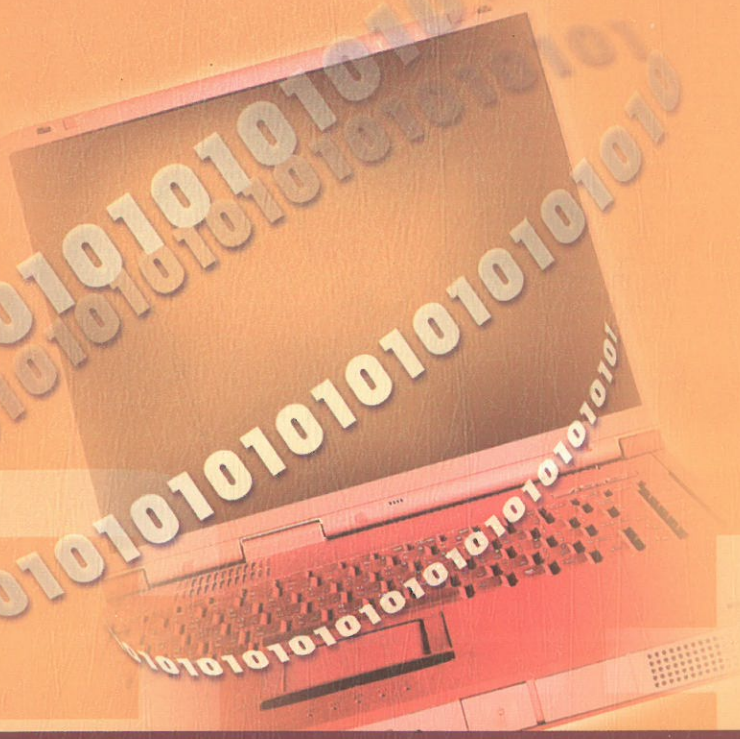
د. حمزة سيد رشوان
د. أبو بكر أحمد السيد



هياكل البيانات بلغة C++

مكتبة الفلاح
للنشر والتوزيع

هياكل البيانات
بلغة C++


بسم الله الرحمن الرحيم

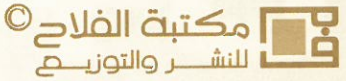
هياكل البيانات
بلغة C++

د. حمزة سيد رشوان
قسم الرياضيات
جامعة الأزهر

د. أبو بكر أحمد السيد
قسم الرياضيات وعلم الحاسوب
جامعة الكويت

مكتبة الفلاح
للنشر والتوزيع

الطبعة الأولى 1427 هـ 2006 م



دولة الكويت
حولي - شارع بيروت - عمارة الأطباء
هاتف: 1985 264 فاكس: 7784 264 965+
ص.ب: 4848 الصفاة - الرمز البريدي 13049

دولة الإمارات العربية المتحدة
العين: - ص.ب 16431 هاتف: 7662189 فاكس: 7657901 3 971+
دبي: - ص.ب: 20438 هاتف: 2630618 فاكس: 2630628 4 971+

جمهورية مصر العربية
37 شارع النصر - امتداد رمسيس 2
مقابل وزارة المالية ومصلحة الجمارك
مدينة نصر - القاهرة
تلفاكس 8143 262 202+
E-mail: alfalah_egypt@ hotmail.com

الموزعون في المملكة الأردنية الهاشمية

العبدلي - مقابل البنك العربي - عمارة الددو
هاتف 5695611 فاكس 1208 568 6 962+
ص.ب 927385 الرمز البريدي 11190
عمان - الأردن
e-mail: dar_honin@yahoo.com

الإخراج وتصميم الغلاف zoom art

الفهرس

الفصل الثاني

المنظومات

الفصل الثالث

الرصات والطوابير

الفصل الرابع

القوائم المترابطة

الفصل الخامس

الأشجار

المقدمة

الحمد لله رب العالمين ، والصلاة والسلام على خاتم المرسلين نبينا محمد وعلى آله وصحبه أجمعين. . وبعد

اللغة العربية من أغنى اللغات العالمية :

إن لغتنا العربية بشهادة اللغويين - العرب والأجانب - من أغنى اللغات وأوفرها حظا من المرونة الاشتقاقية ، ولدينا من وسائل ابتكار المفردات الجديدة* ما يمكننا من مواجهة كل جديد. وقد وسعت العربية كل متطلبات التطور الحضاري إبان النهضة الأولى في العصر العباسي ، كما وفت بالمطلوب في مطلع النهضة الحديثة في مصر في القرن التاسع عشر ، وأمامنا الآن الجامعات السورية تدرس كل علومها بالعربية.

وتعد العربية من أغنى اللغات العالمية نظرا لما تحتويه من مفردات ومعان ، ومن بلاغة وفقه ، ومن إعراب ونحو وصرف ، ومن أساليب تعبيرية رائعة لا تضاهيها لغة من اللغات.

ومن المعلوم أنه إذا تخلف قوم ضعفت لغتهم وربما ضاعت واندثرت وصارت كلاما غير مفهوم ، ولكن تخلف العرب والمسلمين لم يؤثر في اللغة العربية ذلك التأثير الكبير ، فقد بقيت اللغة العربية قوية وثابتة ، ولا تزال يتحدث بها مئات

(*) الاشتقاق ، والاقتراض ، والنحت ، والارتجال ، والقياس ، والمجاز ، والإلصاق ... وغيرها.

الملايين ، ولا زالت قادرة على استيعاب أحدث المصطلحات ، وعلوم العصر ، ويرجع ذلك إلى أن اللغة العربية هي لغة القرآن الكريم ، اللغة التي حملت رسالة الله إلى عباده.. حملت رسالة الإسلام وما فيه من عقيدة وشريعة وعبادات ومعاملات وغير ذلك من الموضوعات فكانت قادرة بإذن الله على التعبير عن هذه المعاني ونقلها إلى كل إنسان ، حتى إن شعوبا كثيرة حفظت هذه اللغة وأتقنتها ولا تزال تستخدمها حتى اليوم.

اللغة العربية. . لغة المستقبل

إن واقع البشرية اليوم يشير إلى أن المستقبل لهذا الدين ، وهذا يعني أن اللغة العربية سوف تكون لغة الحضارة المقبلة ، فينبغي للمسلمين ولأهل العربية أن يزداد اهتمامهم بلغتهم من حيث التعرف على أصولها وقواعدها ودراستها واستنباط الأفكار والمعاني منها ، وصياغة المصطلحات المعاصرة بها ، ونقل أحدث العلوم إلى ساحتها ، وترجمة أمهات الكتب والمراجع في شتى العلوم والمعارف النافعة إليها ، وغير ذلك من الجهود التي تجعل لغتنا افضل اللغات وأقدرها بإذن الله تعالى على أن تكون لغة المستقبل.

اللغة العربية. . لغة علوم. . ولغة آداب

إن التعريب ليس مجرد قضية لغوية ، ولكنه قضية معاصرة ومسايرة للمد الحضاري العالمي الحديث. وهناك بادئ ذي بدء مغالطة ، وهي أن اللغة العربية لغة شعر وأدب وليست بلغة علم ، وإذا كان من المسلم به أن العربية تزخر بالأشعار الرائعة وبالنصوص الأدبية الخالدة فهذا لا يمنعها من كونها أيضا لغة علم ، بل إنها علمية في حروفها ، وفي تسلسل كلماتها ، وفي تركيب جملها ، وفي قواعدها النحوية لدرجة تسمح بتنميط استعمالها في الحواسيب الإلكترونية وغيرها من أحدث الأجهزة والابتكارات العلمية والعلوم العصرية ، وإذا كنا لا نجد في العربية

اليوم مصطلحات بسيطة تقابل ما وضعه الآخرون منذ سنوات عديدة ، أولا نجد مؤلفات بالعربية في فروع عديدة من العلوم الحديثة فإن العيب ليس في اللغة بل في أهلها ومستعمليها الذين دخلوا في سبات عميق منذ نحو ثلاثة قرون ، فلو كان العيب في اللغة لما أعطت الحضارة الإسلامية في عصر شموخها ما أعطته من علوم ومعرفة طيلة قرون منذ أيام الأمويين والعباسيين ، ولنتذكر أن المخطوطات والمؤلفات العربية كانت تعد المرجع الأساسي في أوروبا حتى تاريخ قريب نسبيا. وعلى سبيل المثال فقد ترك محمد بن موسى الخوارزمي مؤلفا أعطى اسمه لفرع من أهم الفروع في الرياضيات وهو الجبر ، واستخدم اسمه عند ابتكار الكلمة الأجنبية «اللوغاريتم» ، والاصطلاح العلمي «الخوارزم» أو «الخوارزمية» (Algorithm) ، والأمثلة على ذلك عديدة ، فالأمر يتوقف على الاجتهاد والجهاد ، وعلينا أن نستبدل بالسكون الحركة ، وألا نكون أعقم خلف لأخصب سلف. علينا أن نسرع وننشط في حركة التأليف والترجمة حتى نواكب ما يستجد من مصطلحات وعبارات ومفاهيم حديثة. واللغة العربية ستعيننا كثيرا في ذلك ، فهي لغة غنية وتتوفر على آليات ضخمة ، فإذا نظرنا مثلا إلى كلمة «حسب» ، وجدنا في حقلها الدلالي : حسَب ، وحاسَب ، وتحاسب ، واحتسب وغيرها من الأفعال ، ثم حساب ، وحسبان ، ومحاسبة ، وتحسب ... الخ. ثم حاسِب ، ومحاسب ، ومحتسب ، وحاسوب ، ... ألخ ، مما يفوق العشرين كلمة ، بينما لا نجد إلا نحو ثلاث كلمات في الحقل الدلالي للكلمة الإنجليزية : calculate أو الكلمة : compute (مثلا : computing ، computer ، compute).

التعريب لا يتعارض مع تعلم اللغات الأجنبية

إن التعريب لا يعني الانغلاق على النفس ، ولا يتعارض مع ضرورة معرفة اللغات الأجنبية ، ويكفي أن نتذكر ما قام به أجدادنا العظام من ترجمات ومن بحث في الحضارات القديمة اليونانية والهندية وغيرها مما كان يتطلب ولا شك

معرفة دقيقة بلغات تلك الحضارات ، فبدأوا باستيعاب جميع أصناف المعارف بواسطة اللغات الأجنبية التي كتبت بها ، ثم قاموا بصياغتها صياغة عربية قبل الابتكار والإبداع واستنباط العلوم العربية وتوطينها ، وما أمر به الخليفة المأمون من حركة لغوية وفكرية وعلمية في بيت الحكمة لدليل على تشجيعه لنقل ما سبق العرب من تراث ، مما يتطلب ضمنيا تشجيعا لمعرفة اللغات ، وأما امتداد العالم الإسلامي من الأندلس إلى وسط آسيا فكان يفرض تنويعا لغويا لا يضر بشيء لغة القرآن الكريم.

وكذلك فإن الأوربيين قد تعلموا بدورهم اللغة العربية لاكتساب العلوم والمعرفة التي وفرتها الحضارة الإسلامية العربية ، وبنوا نهضتهم ، وخرجوا من ظلمات القرون الوسطى بفضل التفتح على حضارتنا ، فلولا ابن الهيثم والبيروني وغيرهما لما سمعنا شيئا عن كوبر نيكوس وجاليليو وغيرهما.

الإنتاج العلمي يعد من تراث اللغة التي كتب بها

وكما قيل «جنسية الفكر هي اللغة» ، فعندما نتحدث مثلا عن الخوارزمي أو ابن خلدون لا نربط إنتاج أي منهما العلمي والفكري بمسقط رأسه داخل العالم الإسلامي ما دام أنه كتبه وخاطبنا باللغة العربية ، وكذلك فإن كل ما يكتبه وينتجه العلماء والمفكرون المسلمون اليوم بلغة أجنبية فسيعد من تراث تلك اللغة التي كتبوا بها ، ولن يعترف التاريخ لهم إلا بهذا الجانب من مساهمتهم ، وأما حقيقة أنهم أصلا مسلمون وعرب فستضيع مع مرور الزمن ، ولن يعرف عنهم في المستقبل البعيد سوى أنهم أنتجوا أبحاثهم بتلك اللغة الأجنبية.

اللغة العربية. . والتماسك بين أبناء الأمة الإسلامية

وإذا كانت اللغة تيسر التواصل والتفاهم والتماسك والتجانس والالتحام بين

أفراد المجتمع البشري ، فإن للغة العربية ميزة عالية - بالإضافة إلى دور اللغة عامة - وهي أنها لغة القرآن الكريم التي تيسر وتضمن التماسك والاعتصام والتضامن بعقيدة واحدة وفي مصير مشترك لأبناء الأمة الإسلامية على اختلاف شعوبهم وقبائلهم.

ومن المعلوم أن ازدواجية اللغة داخل بعض الأقطار الإسلامية حيث تتصارع اللغة العربية مع لغة أجنبية قد أدت إلى انفصام في المجتمع بين نخبة متفرنجة من جهة وجماهير شعبية من جهة أخرى.

تعريب التعليم أساس لتعريب الأمة

إن الدول الاستعمارية تجعل من لغاتها وثقافاتها أولوية الأولويات ، وتخصص ميزانيات هائلة لنشر لغاتها واستنباتها في شتى بقاع الأرض. ومن المعلوم أن التعليم منهج يساعد العقل على اكتساب المعلومات واكتشاف الحقائق ، ومن الطبيعي أن يعمل الإنسان ويُعَلِّم كما تعلَّم ، فلا غرابة أن يستعمل الناس لغة أجنبية في حياتهم المهنية وأحيانا في حياتهم اليومية إذا كان تعليمهم وتكوينهم قد تم بواسطة هذه اللغة الأجنبية ، ومن هنا نرى أن نقطة الانطلاق نحو التعريب الشامل تكمن في تعريب النظام التعليمي بجميع أطواره.

إقصاء العربية عن مجال التعليم حرب على الإسلام

ولم تُثر قضية التعريب في الوطن العربي في العصر الحديث إلا بعد أن استيقظت الأمة العربية على المؤامرة الكبرى التي خطط لها أعداء هذه الأمة من المستعمرين من الإنجليز والفرنسيين حين أخذوا أزِمَّة التخطيط في هذه البلاد في أيديهم ، وعملوا - وهم أصحاب السلطان - على إقصاء اللغة العربية عن مجالات العلم والتعليم ، واحلوا محلها لغتهم الأجنبية ، من أجل أن يتمكنوا بأيسر السبل

من تذويب شخصية الأمة وفرض التبعية لهم عليها. فهذه الحرب الضروس على العربية إنما كانت موجهة في الواقع إلى الإسلام وكتابه ، لأن العربية هي مفتاح هذا الدين : كتابه وثقافته وحضارته ، وقد قالها جلادستون - رئيس وزراء بريطانيا - صراحة حين رفع في مجلس العموم نسخة من القرآن وقال : «ما دام هذا الكتاب يُقرأ فلا قرار لنا في بلاد الشرق». ولعلمهم أن سلطانهم المباشر على تلك البلاد لن يدوم ، فقد خططوا لما بعد ذلك ، فاتخذوا لهم صنائع وعملاء من أبناء البلاد ، أكثرهم صليبيون ، وبذروا بذور الحرب في صورة قضايا علمية وتربوية تصب كلها في مجرى واحد هدفه إزاحة العربية.

التعريب. . قضية صراع على حياة الأمة *

فقضية التعريب إذن هي قضية صراع على حياة هذه الأمة وبقائها ، وعلى عزتها وتقدمها ، وعلى مكانها في موقع القيادة من العالم ، ولنستمع إلى أستاذ أجنبي غير مسلم يكشف لنا الحقيقة في إنصاف ، حيث يقول المستشرق «ماسنيون» : «إن من حق العرب علينا أن نرفع الصوت عاليا ، طالبين إليهم الصمود والمقاومة ليصمدوا وليقاوموا هذه الدعاية المذلة ، التي تسول لهم التنازل عن شرفهم وتراث آبائهم ، والاستسلام أمام القوة الاستعمارية ورؤوس الأموال المصرفية التي تطلب إليهم الانسجام في طريقة تفكيرهم وعملهم مع هذه الحضارة الكاذبة ، حضارة الإنسان الآلي ، التي لم تعد تؤمن بنفسها أو بالذات الإلهية ، وتصبو إلى إخضاع العالم لثقافة أمريكية بلهاء. إن هذا الإنتاج الصناعي المغشوش يوشك أن يسقط ويتردى في الهاوية. وعلى أبناء العربية أن يصمدوا ، فالعالم في حاجة إليهم ، وواجبنا أن نحثهم على احترام عربيتهم ، هذا النظام اللغوي الصافي ، الصالح

(*) من بحث بعنوان «التعريف .. قضية حياة» للاستاذ عبد الوارث سعيد ، نشر بجريدة القبس الكويتية ، العدد ٣٢٥٤ ، بتاريخ ٣/ ٦/ ١٩٨١.

لنقل اكتشافات الفكر عبر القرون ، فلا يحيلوها مسخا مقلدا للغاتنا الآرية. لقد كانت العربية ولا تزال لغة الحرية العليا ، لغة وضوح الذهن ووحي القلب ، لغة المناجاة ، ولغة المعجزة».

إن تصوير قضية التعريب على أنها قضية «أن نكون أو لا نكون» - إن كان الاستشهاد بالأمثال الأجنبية يفيد في هذا المقام - ليس حماسا ولا شعارات ، وإنما هي نتيجة حتمية لمقدمات ومسلمات حقيقية أجمع عليها الباحثون من أهل الاختصاص ، ويشهد عليها الواقع الأليم الذي نعيش فيه.

أهمية القرار الرسمي بالتعريب

وما دام الأمر كذلك فإن مسئولية التعريب لا تقتصر على دائرة المتخصصين في العلوم فحسب وإنما يقع العبء كذلك في اتخاذ قرار في هذا الشأن على الجهات السياسية وكبار المسئولين عن التعليم لأن الأمر يتعلق بمصير الأمة ومستقبلها ، وهذا القرار لا يصدر لينفذ في معهد واحد أو كلية واحدة وإنما في جميع المؤسسات التعليمية حتى أعلى مراحلها ، بل يكون قرارا للأمة كلها ، إنه قرار «تفصيح الأمة وتعريبها» بما في ذلك أجهزتها ومؤسساتها الإعلامية ، وبدون هذا القرار وتنفيذه سيظل عمل الهيئات العلمية في مجال التعريب هزيلا محروما من روافد القوة والتثبيت.

تجنيد الطاقات وأولوية التعريب

فيجب تجنيد الطاقات البشرية والمادية للأمة لإنجاح عملية التعريب إلى أعلى درجة ممكنة في أقصر وقت ممكن. يجب أن تعطى لموضوع التعريب الأولوية على كثير من المشاريع التي تستهلك فيها الآن تلك الطاقات والنعم التي أفاضها الله علينا.

حين نزن الأمر بميزان مصلحة الأمة لابد أن نعطي موضوع التعريب مكان الصدارة والأولوية على مشاريع الفنون والرياضة والسياحة وتجميل المدن وتوفير وسائل الرفاهية ، فأساس البيت وأعمدته وجدرانه أولى بالعناية والبذل من التفنن في أشكال الأبواب والنوافذ وألوان قطع الأثاث ونحوها. إن إنشاء معهد لترجمة العلوم وتوفير الكوادر الفنية ذات الكفاءة وتوفير الإمكانات المادية ليتم نقل العلوم من كافة اللغات إلى العربية لهو أولى ألف مرة من إنشاء أكاديمية للفنون ، وإن تهيئة الإمكانات لإصدار قواميس علمية متخصصة شاملة وحديثة أو دوريات علمية تنقل أحدث ثمرات العلم في العالم إلى لغتنا مساعدة للباحثين والمؤلفين والدارسين لهو أجدى على الأمة ألف مرة من ترجمة كل الإنتاج الأدبي العالمي من شعر وقصص ومسرحيات. وإن مساعدة جامعة عربية - أيا كان موقعها - على أن تنجح في تعريب العلوم فيها أولى وأهم من إنشاء المدن الرياضية العالمية.

موقفنا نشاز

ورغم أهمية الدور الذي تقوم به اللغة في بناء حاضر الأمة ومستقبلها في شتى المجالات ، ورغم اهتمام الغالبية العظمى من دول العالم - حتى ما كان منها حديث النشأة أو الاستقلال - بلغاتها القومية فإن موقف أمتنا - ذات التاريخ الحضاري المجيد - يمثل حالة فريدة تعتبر نشازا بين مواقف الأمم قديما وحديثا.

فما من مرة عرضت فيها قضية التعريب إلا وبرز من بين الصفوف المعارضون والمشككون والمترددون ، ودعاة التأجيل ، والمحذرون والمنذرون. . وتراهم يتصيدون الحجج والمبررات من هنا وهناك لتدعيم موقفهم وليظهروا حرصهم على مصلحة الأمة ومستقبل التقدم فيها. ورغم الجهود المشكورة التي بذلها المؤيدون للتعريب في تفنيد مزاعم المعارضين ومبرراتهم الواهية والمفتعلة إلا أننا مازلنا نرى منهم للأسف إصرارا عليها واجترارا لها في كل مناسبة.

من تجارب الشعوب في التعليم بلغاتها القومية

إن كل الأمم والشعوب التي نجحت في إحياء لغاتها القومية وتطويرها وفرضها لغة للتعلم في كل المراحل انطلقت في عملها هذا انطلاقة قومية تبنتها ودعمتها أعلى سلطة سياسية في البلد ، وكل شعوب الأرض - إلا بقايا توابع للاستعمار الأجنبي - تعلم بلغاتها القومية على ضآلة حظ كثير من تلك اللغات من الإمكانات الذاتية والتعامل مع متطلبات الثقافة والعلوم.

فمثلا كل الدول الأوروبية الحديثة - الشرقية والغربية - على كثرتها وصغر حجمها مساحة وسكانا ، وعلى اختلاف ظروفها السياسية والاقتصادية والتي لا تصل لغاتها إلى مستوى اللغة العربية من حيث عدد الناطقين بها مثل البولندية والهولندية والسويدية والألمانية والمجرية ... الخ تعتز بشخصيتها القومية وبلغتها الوطنية ، وتصر على استعمالها في معاملاتها اليومية وفي معاهدها التعليمية وتدرس في كل المراحل بلغاتها المحلية ، بل إن دولة مثل سويسرا التي تتوزعها لغات ثلاث : الفرنسية والألمانية والإنجليزية ، تدرس لكل منطقة حسب اللغة السائدة فيها ، فنحن أولى بالاعتزاز بشخصيتنا الإسلامية وبلغتنا القرآنية.

ومعلوم أن العبرية لغة سامية شقيقة للعربية ، وبينهما لذلك كثير من وجوه التلاقي في القواعد والمفردات. ولكن العربية ظلت متصلة الحياة - والنمو إلى حد ما - عبر القرون. أما العبرية فقد تعرضت لفترات توقف وضعف حتى كادت أن تصبح لغة ميتة لا يعرفها إلا الأحبار في النصوص الدينية ، ولم تبعث العبرية إلى الحياة إلا في مطلع القرن العشرين بعد نحو ألفي سنة من التوقف لتهيأ وتعد لتكون لغة للدولة التي خططوا لإقامتها على أرض الإسراء ، وأصبحت العبرية لغة حياة بعد الجيل الأول من المهاجرين وهي الآن لغة التعليم الوحيدة في كل المنشآت التعليمية في «إسرائيل» من المرحلة الأولى إلى أعلى المراحل وبكل التخصصات العلمية المتقدمة. أفلا نأخذ من هذا درسا ولو كان من عدونا الذي يجثم في بقعة من أشرف البقاع من ديارنا ؟.

ولقد واجهت أوروبا في عصر النهضة حضارة إسلامية متقدمة في جميع المجالات الفكرية والعلمية والتطبيقية. ولم يكن الفارق بين أوروبا آنذاك في جهلها ، والأمة الإسلامية في تقدمها بأقل من الفارق بيننا في تأخرنا الحالي وأوروبا في تقدمها المادي. فماذا فعلت أوروبا ؟ لقد أرسلت أبناءها وبناتها للدراسة في الأندلس ، ونقلت إلى اللاتينية - لغة العلم والتعليم في أوروبا وقتئذ - ذخائر الحضارة الإسلامية في الفكر والعلم. درس علماؤها العربية ونبغوا فيها حتى صار رجال الدين يشكون من إهمال المسيحيين للغة الإنجيل وإقبالهم على العربية وأدبها. ومع ذلك لم يسمع التاريخ أن أوروبا اتخذت من العربية لغة للتدريس أو التحدث أو تسجيل التراث. كانت اللاتينية هي التي تقوم بهذا الدور إلى أن قويت اللغات المحلية المنبثقة عنها ، كالفرنسية والإيطالية والأسبانية والبرتغالية والرومانية فحلت محلها. نقلوا كثيرا من المصطلحات العلمية إلى لغاتهم ، ومازالت آثار منها باقية حتى الآن ، ولكنهم لم يتخلوا عن لغتهم ليدرسوا بلغة أجنبية. ولم تكن اللاتينية ولا اللغات المحلية - إذا قيست بالعربية آنذاك - بأفضل حظا من العربية اليوم من حيث الثروة اللغوية في المجالات العلمية.

تعريب العلوم. . في الحضارة الإسلامية الشامخة

وحين بدأت حركة الترجمة والنقل في الدولة العباسية عزم المسؤولون آنذاك على نقل ثمار الحضارات الكبرى المعروفة آنذاك : اليونانية والفارسية والهندية والسريانية والصينية. . وقد نقلوا عنها ما كانوا في حاجة إليه وتركوا ما لا يحتاجون وما لا يتلاءم مع عقيدتهم وقيمهم. وتمت حركة رائعة خصبة من النقل بفضل بيت الحكمة ودواوين الترجمة. وفاضت عبقرية العربية في التعريب والتوليد والاشتقاق وغيرها فوسعت مفاهيم علوم تلك الحضارات ، ولم يحدث أبدا أن استخدمت لغة غير العربية في التدريس. كانت العربية تقبل وتهضم ثم تنمو وتنتشر في كل مكان فتلقى من الترحيب ما يثبت أقدامها في كل البلاد ، ولولا بعض النزعات العرقية

المتأخرة ومؤامرات أعدائنا في العصر الحديث لوجدنا العربية حتى اليوم راسخة القدم في كل البلاد الإسلامية ، على الأقل لغة للعلم والثقافة بقبول من الشعوب الإسلامية ورضا.

فالواجب علينا أن نعيد رفع راية الجهاد العلمي كما رفعها آباؤنا الذين أرسوا دعائم تلك الحضارة الإسلامية العلمية الرائعة ، وأن نبذل جهدا كبير في نقل ثمار العلوم الحديثة في شتى الفروع والتخصصات إلى اللغة العربية.

هذا الكتاب

ومن هذه العلوم الحديثة التي يتعين علينا نقلها إلى العربية علم : هياكل البيانات (والذي يطلق عليه أيضا : بنى المعطيات) (Data Structures) وهو علم يتناول مواصفات/ خصائص (specifications) وتصميم (design) وتحليل (analysis) البنيات (structures) والخوارزميات (algorithms) اللازمة لتخزين (storage) وتشغيل (processing) البيانات (data) ، حيث يتناول بصورة أساسية الخصائص والمواصفات التي لا تعتمد على الأجهزة المستخدمة (machine independent specifications) لبنى المعطيات الأساسية (basic data structures) التالية : المنظومات (arrays) ، والقوائم (lists) ، والرصّات (stacks) ، والطوابير (queues) ، والأشجار (trees) ، والمجموعات (sets) ، والرسوم البيانية (graphs) ، وكذلك كيفية تنفيذ / استخدام (implementations) هذه البنى في المسائل والتطبيقات العملية ، والعمليات (operations) المختلفة عليها.

وتعتمد مادة هذا الكتاب «هياكل البيانات بلغة C ++» بصفة أساسية على المذكرات التي أعدها الدكتور حمزة رشوان لسلسلة محاضرات في منهج باللغة الإنجليزية لمدة فصل دراسي واحد عن هذا الفرع من علوم الحاسوب أثناء قيامه بالتدريس في جامعة الكويت ثم صيغت محتويات هذا المنهج باللغة العربية ليعم الانتفاع بها ، ولتثرى بإذن الله تعالى المكتبة العربية في العلوم الحديثة.

فصول الكتاب

ويتناول الفصل الأول بعض المفاهيم الأساسية كدرجة التعقيد الزمنية (time complexity) والمكانية (space complexity) للخوارزميات وكفاءتها ، مع ضرب الأمثلة المختلفة لتطبيق هذه المفاهيم ، كما يتناول الأنواع المختلفة للبيانات واصطلاحات التقارب. وأما الفصل الثاني فهو خاص بدراسة المنظومات أحادية البعد (one-dimensional) ومتعددة الأبعاد (multidimensional) ، والعمليات المختلفة عليها ، وكيفية استخدامها لتنفيذ القوائم الخطية والمرتبة (linear and ordered lists) ، ومن أهم الأمثلة العملية على القوائم التي تقوم المنظومات بتنفيذها والتي يعرضها هذا الفصل : الحدوديات (polynomials) ، والمصفوفات (matrices) ، وبصفة خاصة الحدوديات المتناثرة (sparse polynomials) حيث تشتمل الحدودية على كثير من الحدود الصفرية ، والمصفوفات المتناثرة (sparse matrices) حيث تشتمل المصفوفة على كثير من العناصر الصفرية ، مع دراسة لأكفأ الطرق لإجراء العمليات على الحدوديات (كجمع الحدوديات) والمصفوفات (كتدوير المصفوفات). ونختتم هذا الفصل بمناقشة كيفية تمثيل المنظومات في الذاكرة (memory). ويتعلق الفصل الثالث بالرصات والطوابير المفردة والمتعددة والعمليات المختلفة عليها وكيفية تنفيذها وبعض التطبيقات العملية عليها كتحويل تعبير من التمثيل الرمزي الوسطي (infix) إلى التمثيل الرمزي اللاحق (postfix) ، وحساب قيم التعابير المكتوبة بالتمثيل الرمزي اللاحق (postfix expressions). وأما الفصل الرابع عن القوائم المترابطة (linked lists) فيبدأ ببيان أهمية التمثيل المترابط للقوائم ومقارنته بالتمثيل التتابعي لها ، ثم يستعرض العمليات المختلفة على هذه القوائم وأنواعها : القوائم المترابطة المفردة (single) والمزدوجة (double) والدائرية (circular) والمتعامدة (orthogonal) ، واستخدام القوائم المترابطة لتنفيذ الرصات والطوابير المفردة والمتعددة ، وكذلك تنفيذ الحدوديات والمصفوفات ، مع الإشارة بصورة خاصة إلى الحدوديات المتناثرة والمصفوفات المتناثرة والعمليات المختلفة على كل

منهما ، ومناقشة درجات التعقيد الزمنية والمكانية لهذه العمليات. ويتناول الفصل الأخير موضوع الأشجار وأنواعها المختلفة : الأشجار العامة ، والأشجار الثنائية ، وأشجار البحث الثنائية ، والأشجار المتوازنة ، وأشجار AVL ، وأشجار B ، والغابات ، وكيفية تمثيل الأشجار العامة بأشجار ثنائية نظرا لما للأشجار الثنائية عموما من أهمية عملية. كذلك يتناول هذا الفصل العمليات المختلفة على الأشجار ، وكيفية تمثيلها تمثيلا تتابعيا (باستخدام المنظومات) أو ترابطيا (باستخدام المؤشرات).

شكر وتقدير

ونود أن نسجل هنا شكرنا العميق لكل من ساهم بأي جهد في صدور هذا الكتاب ، ونخص بالشكر «مكتبة الفلاح بالكويت» لجهدها معنا في تيسير نشر وتوزيع الكتب العلمية باللغة العربية مما يساعد على دفع عملية التعريب خطوات حثيثة للأمام ، كذلك نقدم خالص شكرنا للأخ الفاضل / السيد البدوي محمد أحمد بكلية العلوم بجامعة الكويت الذي لم يدخر جهدا في حسن طباعة هذا الكتاب وغيره من الكتب السابقة في هذه السلسلة من كتب علم الحاسوب والرياضيات باللغة العربية ، جعل الله ذلك في ميزان حسناته جزاء صبره ومثابرته وتعاونه المستمر في المساهمة في دعم عملية تعريب العلوم.

كذلك نتوجه بالشكر والتقدير والدعاء بالثواب الجزيل من الله تعالى لكل من يشارك في ركب التعريب : تأليفا أو ترجمة أو تدريسا أو في وضع قرارات وتوصيات التعريب العديدة موضع التنفيذ. فكم من مرة اجتمع وزراء التعليم والثقافة والتعليم العالي العرب ، وأقيمت ندوات ومؤتمرات تحت إشراف الجامعة العربية ، واتحاد الجامعات العربية ، وصدرت عن تلك الاجتماعات قرارات وتوصيات بضرورة البدء في عملية تعريب التعليم الجامعي كله. . ولكن لا حياة لمن تنادي !.

لغة القرآن أمانة في أعناقنا

إن هذه اللغة ليست ملكا للعرب وحدهم ، ولا لمسلمي هذا الجيل وحدهم ، وإنما هي أمانة يتحتم علينا أن نحفظها للأجيال من بعدنا كما تلقيناها عمن قبلنا ، فهي لغة كتاب ربنا وأحاديث رسولنا صلى الله عليه وسلم ، وعلوم وثقافة حضارتنا.. وإذا قال قائل : أوكلما حدثناكم في شئ أقحمتم فيه الإسلام وقلتم : القرآن القرآن. إذا تكلمنا عن تطوير قوانين المجتمع واستحدثنا بعض القوانين من الشرق أو الغرب قلتم : هذه لا تتفق مع شريعة القرآن. إذا تكلمنا عن تطوير الدراسات اللغوية قلتم : إنها لغة القرآن ، حتى إذا تكلمنا عن نقل علوم الغرب من أحياء وفيزياء أو رياضيات وحاسوب قلتم لابد أن نطعمها بمبادئ الإسلام وأن تسري فيها روح القرآن ، لا حجة لكم إلا الإسلام ولا تَعِلَّة لكم سوى القرآن؟ ونحن نقول لهؤلاء : نعم ، القرآن في تقديركم شئ هين يسير ، وهو في تقديرنا عظيم خطير ، فهو دستور أمتنا ومنهاج حياتنا ، به نحيا وفي سبيله نجاهد وعليه نموت ، ونرجو أن يشفع لنا يوم نلقى الله عز وجل.

إن الحياة عندكم نعيم وزخرف ومتاع ثم لا شئ بعد ذلك إلا الفناء ، أما نحن فالحياة عندنا معبر للآخرة وطريق إليها.

إذا كان الأدب عندكم لهوا ومتاعا وخرافات وأوهاما ، فهو عندنا أسمى من ذلك وظيفة وأعز مكانا.

وإذا كانت العلوم عندكم غاية فهي عندنا وسيلة إلى عبادة الله وعمارة الكون. وإن الشعوب التي تتخذ الوسائل غايات ، والعلوم والفنون آلهة تعبد ، وتعشق الحياة والملذات ، وتضعف فيها الإرادة وقوة المقاومة للمغريات ، وينتشر فيها الاستخفاف بفرائض الدين والأخلاق ، فإن هذه الأمة لا تثبت أمام أي عدو زاحف يوما واحدا ، وهذه قصة اليونان وقصة الرومان ، وما أيام نكبتنا في بلادنا العربية منا ببعيد ، فلا قيمة لهذه العلوم ولا وزن لتلك الآداب إذا جردت المجاهد المسلم

من سلاحه الأول سلاح الإيمان وجعلته أعزل ضعيفا. فلنكيف العلوم والآداب مع عناصر ثقافتنا العربية وشخصيتنا الإسلامية ولنخضعها لرسالتنا العالمية الخالـدة ، ونجعـلها جندا من جنودها حتى نكون بحق من جند الله العاملين ، ﴿وَلَقَدْ سَبَقَتْ كَلِمَتُنَا لِعِبَادِنَا الْمُرْسَلِينَ * إِنَّهُمْ لَهُمُ الْمَنصُورُونَ * وَإِنَّ جُندَنَا لَهُمُ الْغَالِبُونَ﴾*. وآخر دعوانا أن الحمد لله رب العالمين.

(*) الصافات : ١٧١ - ١٧٣.

الفصل الأول
مفاهيم أساسية
Basic Concepts

تصميم وتنفيذ البيانات
Data Design and Implementation

1

1 Data Design and Implementation

الفصل الأول

مفاهيم أساسية

Basic Concepts

تصميم وتنفيذ البيانات

Data Design and Implementation

نتناول في هذا الفصل بإذن الله تعالى بعض المفاهيم الأساسية التي نحتاجها في دراسة هياكل البيانات / بنى المعطيات (Data Structures) كتقييم كفاءة (efficiency) خوارزمية ما ، والمقارنة بين الخوارزميات المختلفة من حيث أداء (performance) الخوارزمية ودرجة التعقيد الزمنية والمكانية لها (Time and Space Complexity of an algorithm) ، والإطار المنطقي (logical framework) لكيفية اختبار (examining) هياكل البيانات من حيث كيفية تحديدها (specifying) ، وكيفية تنفيذها (implementing) ، وكيفية استخدامها (using).

تجريد البيانات (Data Abstraction)

نقصد بالبيانات (data) المعلومات (information) التي يقوم برنامج الحاسوب (computer program) بتشغيلها (processing) ، أي الأشياء (objects) التي يقوم

البرنامج بمعالجتها (manipulating) ، وهي عبارة عن مجموعة من الوحدات الثنائية (collection of bits). أما الإنسان فإنه يميل إلى التعامل مع معلومات ذات وحدات أكبر (larger units) كأعداد (numbers) وقوائم (lists). ولذلك فإننا نحاول أن نجعل في برامجنا الأجزاء التي يقرأها الإنسان (human-readable portions) تشير إلى بيانات في صيغة مقبولة لنا. وللفصل (separation) بين نظرة الحاسوب للبيانات (computer's view of data) ونظرتنا لها نستخدم ما يسمى بتجريد البيانات (data abstraction). فالمقصود من :

تجريد البيانات :

هو الفصل (separation) بين الخواص المنطقية لنوع بيانات (data type's logical properties) وبين تنفيذه (its implementation).

فمثلاً نتعامل في برامجنا مع ما يسمى بالأعداد الصحيحة (integers) ، وكل ما نحتاجه هو معرفة كيف نعلن عن متغير من هذا النوع int ، وما هي العمليات المسموح بها على الأعداد الصحيحة : الإسناد (assignment) والجمع (addition) والطرح (subtraction) والضرب (multiplication) والقسمة (division) وحساب الباقي (modulo arithmetic). أما كيفية التمثيل الفيزيائي (physical representation) لهذه الأعداد في الحاسوب ، وبالتالي كيف يقوم الحاسوب بمعالجة هذه الأعداد فأمر لا يهتم به المبرمج عادة ، ولا يضيره عدم معرفة تفاصيله. والتمثيل الفيزيائي للأعداد يختلف من حاسوب لآخر ، ففي ذاكرة حاسوب قد يوجد العدد في الصيغة العشرية الثنائية (Binary-coded decimal) ، وفي ذاكرة آخر قد يوجد في الصيغة الثنائية المباشرة دون إشارة (Unsigned direct binary) ، وفي حاسوب ثالث في صيغة إشارة وقيمة (Sign and magnitude) ، وفي رابع في صيغة مكمِّل - ١ (One's complement) ، وفي خامس في صيغة مكمِّل - ٢ (Two's complement) ، وهكذا. والجدول التالي

(شكل ١-١) يعطي مثالاً لعدد ثنائي ، والعدد العشري المكافئ له في كل من الصيغ الثنائية المختلفة المذكورة سابقا.

العدد العشري المكافئ	عدد ثنائي	10011001			Binary
	99	153	-25	-102	-103
	العشرية - الثنائية Binary-coded decimal	ثنائية مباشرة دون إشارة Unsigned direct binary	إشارة وقيمة Sign and magnitude	مكمل -١ one's comple-ment	مكمل -٢ Two's comple-ment

شكل ١-١

المكافئات العشرية لعدد ثنائي من ٨ وحدات ثنائية

تغليف البيانات (Data encapsulation)

نفرض أن لدينا العبارة

```
Distance = rate * time;
```

مفهوم عملية الضرب (*) لا يعتمد على نوع الكميات المعامَلة (operands) - إن كانت أعداداً صحيحة (integers) أو كميات/ أعدادا حقيقية (real numbers) - رغم أن تنفيذ (implementation) عملية الضرب بالنسبة للأعداد الصحيحة (integer multiplication) قد يختلف بصورة كبيرة عن تنفيذها بالنسبة للأعداد الحقيقية (ذات النقطة العائمة) (floating - point multiplication) في الحاسوب نفسه. وبالطبع لن يكون استخدام الحاسوب سهلاً إن وجب علينا النزول إلى مستوى التمثيل الفيزيائي للأعداد في الحاسوب كلما أردنا إجراء عملية ضرب ،

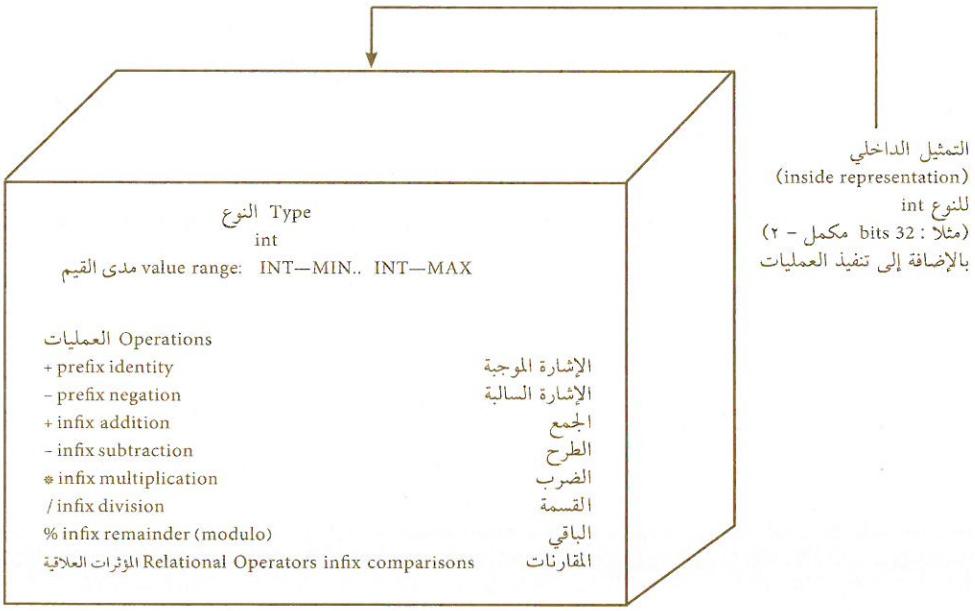
ولكن هذا في الواقع غير ضروري ، حيث أن لغة C++ تحيط (surrounds) نوع البيانات بحزمة (package) وتعطينا فقط المعلومات (information) التي نحتاجها لإنشاء (creating) ومعالجة (manipulating) بيانات من هذا النوع. وكمرادف لكلمة «تحيط» نقول «تغلِّف» أو «تغلِّف بكبسولة» (encapsulates) تشبيها بكبسولة الدواء التي نأخذها من الصيدلي ونعرف تأثيرها كمضاد حيوي مثلا دون أن يهمنا معرفة تفاصيل التركيب الكيميائي للدواء داخلها. أي أن :

تغليف البيانات :

هو الفصل بين تمثيل البيانات والتطبيقات (applications) التي تستخدم (use) البيانات على مستوى منطقي (at a logical level). أي أنه خاصية / سمة (feature) من سمات لغة البرمجة التي تفرض إخفاء المعلومات (information hiding). أي أن تغليف البيانات يعني تغليف التمثيل الفيزيائي لبيانات البرنامج بحيث لا يرى مستخدِم البرنامج تفاصيل التنفيذ ، ولكنه يتعامل مع البيانات فقط بدلالة صورتها المنطقية (logical picture) ، أي تجريدها (abstraction).

وإذا كانت البيانات مغلفة ، فكيف يستطيع المستخدِم الوصول إليها؟ يتم ذلك عن طريق العمليات (operations) التي تسمح له بإنشاء (creating) بيانات والوصول (accessing) إليها وتغييرها. وبخصوص العمليات التي تسمح بها لغة C++ بالنسبة لنوع البيانات المغلَّف int : فيمكننا أولاً إنشاء (creating /constructing) متغيرات من النوع int باستخدام إعلانات (declarations) في البرنامج. ثم يمكننا إسناد قيم لهذه المتغيرات الصحيحة - إما باستخدام مؤثر الإسناد (assignment operator) أو بقراءة قيم فيها - وإجراء عمليات حسابية باستخدام المؤثرات : %, / , * , - , + . والشكل التالي (شكل ١-٢)

يوضح كيف تغلِّف لغة C++ النوع int في حزمة (package) أو ما يعرف بصندوق أسود (black box).



شكل ١-٢

صندوق أسود يمثل عددا صحيحا

(A black box representing an integer)

نوع البيانات المجرد Abstract data type (ADT)

نعلم أن الهدف من التصميم هو تقليل درجة التعقيد (complexity) عن طريق التجريد (abstraction). ويمكننا توسيع هذا الهدف ليشمل أيضا حماية (protection) تجريد البيانات (data abstraction) هذا عن طريق التغليف (encapsulation). ونشير إلى مجموعة (set) جميع القيم المحتملة (possible values) [أي مجال (domain)] لهدف / لكائن بيانات مغلف (encapsulated

(data object) بالإضافة إلى توصيفات (specifications) العمليات (operations) المسموح بها / المتاحة (provided) لإنشاء (creating) ومعالجة (manipulating) البيانات بنوع بيانات مجرد (abstract data type) ، أو اختصارا ADT ، أي أن :

نوع البيانات المجرد (ADT) :

هو نوع بيانات (data type) تُحدَّد (specified) خواصه (properties) [وهي المجال (domain) والعمليات (operations)] بصورة مستقلة (independently) عن أي تنفيذ خاص (particular implementation).

بنى المعطيات / هياكل البيانات (Data Structures)

إذا احتجنا في برنامج ما إلى عدَّاد (counter) أو إلى مجموع (sum) عدة قيم أو إلى مؤشِّر (index) فيكفينا حينئذ عدد صحيح مفرد (a single integer). ولكن عموما نحتاج إلى التعامل مع بيانات تشتمل على أجزاء عديدة (lots of parts) كقائمة (list) مثلا. وفي هذه الحالة نَصِف الخواص المنطقية (logical properties) لهذه المجموعة من البيانات (collection of data) كنوع بيانات مجرد ، ونُطْلِق على التنفيذ الراسخ (concrete implementation) للبيانات بنية معطيات (a data structure). أي أن :

بنية المعطيات (data structure) :

هي مجموعة من عناصر البيانات (collection of data elements) التي يتميز ترتيبها (organization) بعمليات وصول (accessing operations) تستخدم لتخزين (storing) واستعادة (retrieving) عناصر البيانات المفردة (individual). وبأسلوب آخر بنية المعطيات هي تنفيذ (implementation) عناصر البيانات المركبة (composite data members) في نوع بيانات مجرد (abstract data type).

فعندما تتكون معلومات برنامج ما (a program's information) من أجزاء / مركبات (component parts) فحينئذ يجب أن نفكر في بنية معطيات مناسبة.

وهناك عدة سمات/ خواص (features) لبنى المعطيات تجدر ملاحظتها وهي :

(i) يمكن تفكيكها / تحليلها (decomposed) إلى عناصرها المركبة (component elements).

(ii) كيفية ترتيب (arrangement) العناصر سمة من سمات بنية المعطيات التي تؤثر على كيفية الوصول إلى (accessing) أي عنصر.

(iii) يمكن تغليف (encapsulating) كل من ترتيب العناصر وكيفية الوصول إليها.

ويمكن اعتبار المكتبة مثالا عمليا لبنية المعطيات :

(i) فالمكتبة يمكن أن تُحلَّل / تُفكك (decomposed) إلى عناصرها المركبة وهي الكتب.

(ii) ويمكن ترتيب مجموعة الكتب المفردة (collection of individual books) بوضعها على الرفوف بعدة طرق مختلفة : إما عشوائيا دون ترتيب معين (unordered) ، أو ترتيبا أبجديا (alphabetical order) بالنسبة لعناوين الكتب (titles) ، أو أسماء المؤلفين (authors) ، أو تخصصات / موضوعات الكتب (subjects) ،. .. وهكذا. . وبالطبع فإن طريقة وضع الكتب على الرفوف تحدد كيفية البحث عن كتاب معين.

(iii) ونفرض أن هذه المكتبة لا تجعل الأشخاص يبحثون بأنفسهم عن الكتب

التي يطلبونها وإنما على من يطلب كتابا أن يقدم طلبه إلى مسئول المكتبة ليُحضر له الكتاب المطلوب. أي أن كيفية ترتيب الكتب وكيفية الوصول إلى الكتاب المطلوب أمر مخفي (hidden) أو مغَلَّف (encapsulated) بالنسبة للعميل (طالب الكتاب).

ونلاحظ أن البنية الفيزيائية (physical structure) والصورة المجردة (abstract picture) لكتب المكتبة مختلفتان. ففهرس البطاقات (card catalog) مثلا يعطي الصور المنطقية (logical views) للمكتبة (حيث الكتب مرتبة بناء على موضوعاتها أو أسماء مؤلفيها أو عناوينها) والتي تختلف عن ترتيبها الفيزيائي (physical arrangement).

ونستخدم مع بنى المعطيات أسلوبا مماثلا في برامجنا ، حيث نعرِّف أي بنية معطيات (a data structure) بكل من :

(١) ترتيب منطقي (logical arrangement) لعناصر البيانات (data elements) ، و

(٢) مجموعة عمليات (set of operations) نحتاجها للوصول إلى (accessing) العناصر.

ونلاحظ الفارق بين نوع البيانات المجرد وبنية المعطيات. فنوع البيانات المجرد يعد وصفا عالي المستوى (high-level description) يشمل : الصورة المنطقية للبيانات والعمليات التي تعالجها. بينما بنية المعطيات شئ راسخ يشمل : مجموعة عناصر بيانات والعمليات التي تقوم بتخزين واستعادة (storing & retrieving) العناصر المفردة. فنوع البيانات المجرد لا علاقة له بالتنفيذ (implementation independent) ، بينما بنية المعطيات تعتمد على التنفيذ (implementation dependent) ، فهي تعني كيفية تنفيذ البيانات في نوع بيانات مجرد تحتوي قيمه (values) على أجزاء تركيبية (component parts).

والعمليات على نوع البيانات المجرد تُترجم (translated) إلى خوارزميات على بنية المعطيات.

وكنظرة أخرى للبيانات فإننا نهتم بكيفية استخدامها في برنامج ما لحل مسألة معينة ، أي أننا نهتم بتطبيقاتها (applications). فمثلا إذا كنا نكتب برنامجا يتتبع درجات الطلاب ، فإننا نحتاج إلى قائمة بأسماء الطلاب ، وإلى طريقة لتسجيل درجات كل طالب. فقد نأخذ مثلا دفتر أوراق قد سُجِّلت عليها باليد درجات الطلاب ، ثم نعالجها في برنامجنا. وقد تشتمل العمليات على دفتر الدرجات هذا على : إضافة اسم ، أو إضافة درجة ، أو حساب متوسط درجات طالب ، .. .. الخ. وبمجرد كتابة توصيف (specification) لنوع بيانات دفتر الدرجات (grade book data type) فإنه يجب علينا أن نختار بنية معطيات مناسبة لننفذها ونصمم (design) الخوارزميات لتنفيذ (implementation) العمليات على البنية.

ويجب علينا تحديد الصورة المنطقة للبيانات ، واختيار تمثيل البيانات (representation of data) ، وبيان العمليات التي تُغلِّف هذه الترتيبات. وأثناء ذلك كله فإننا نأخذ في الاعتبار المستويات الثلاثة التالية بالنسبة للبيانات :

١) مستوى التطبيق / المستخدِم (application / user level) : ويعني طريقة نمذجة / معالجة البيانات في الحياة العملية (modeling real-life data) في سياق معين (specific context) ، وهو ما يُطلق عليه أيضا مجال المسألة (problem domain).

٢) المستوى المنطقي / المجرد (logical / abstract level) : وهو يمثل نظرة مجردة لقيم البيانات (data values) [المجال (domain)] ومجموعة العمليات (set of operations) المستخدمة لمعالجتها (manipulating).

٣) مستوى التنفيذ (implementation level) : ويعني تمثيلا محددا (specific

representation) للبنية (structure) ، للاحتفاظ (holding) بمفردات البيانات (data items) ، وتشفير (coding) العمليات (operations) في لغة برمجة (a programming language) [إذا لم توفر اللغة أصلا هذه العمليات].

وعادة نشير إلى المستوى الثاني بـ «نوع بيانات مجرد ". ونظرا لأن نوع البيانات المجرد يمكن أن يكون نوعا بسيطا كعدد صحيح (integer) أو رمز (character) ، أو يكون بنية (structure) مركبة من عناصر (component elements) ، فإننا نستخدم أيضا الاصطلاح «نوع بيانات مركب» (composite data type) ليشير إلى نوع البيانات المجرد الذي قد يحتوي على عناصر / مركبات. وأما المستوى الثالث فإنه يصف كيفية التمثيل الفعلي والمعالجة الفعلية (actual representation and manipulation) للبيانات في الذاكرة ، أي يصف بنية المعطيات وخوارزميات العمليات التي تعالج الوحدات الموجودة في البنية.

وعودة إلى مثال المكتبة لبيان المشابهة ، يمكننا أن نقول :

i) على مستوى التطبيق ، نركِّز على وحدات / عناصر (entities) معينة مثل مكتبة دار الحكمة بالقاهرة ، ومكتبة الإسكندرية ، ومكتبة الكتب والمخطوطات النادرة ، ومكتبة اسطنبول ، ومكتبة بغداد.

ii) وعلى المستوى المنطقي : نتعامل مع أسئلة الماهية : ما هو؟ / ما هي؟ / ماذا؟ (what?) : ما هي المكتبة؟ وما هي الخدمات / العمليات التي يمكن للمكتبة أن تقدمها؟ فيمكننا أن ننظر للمكتبة نظرة مجردة (abstract) على أنها مجموعة كتب محدد لها العمليات التالية :

* استعارة (checking out) كتاب من المكتبة.

* إعادة (checking in) كتاب للمكتبة.

* حجز / حفظ (reserving) كتاب عند محاولة استعارته.

* دفع غرامة (fine) لتأخير إعادة كتاب.

* دفع ثمن فقدان كتاب.

ونلاحظ أن هذا المستوى المنطقي لا يهتم بكيفية ترتيب الكتب على الرفوف لأنه يفترض أن العملاء لا يصلون إلى الكتب مباشرة ، وكل ما يُطلب منهم معرفته هو الطريقة الصحيحة لطلب إجراء العملية المطلوبة. فمثلا لإعادة كتاب : سلِّم الكتاب عند نافذة إعادة الكتب في المكتبة التي استُعير منها الكتاب ، واستلم ورقة الغرامة إن تجاوز الكتاب المدة المسموح بها لاستعارته.

iii) وعلى المستوى التنفيذي : نتعامل مع أسئلة الكيفية : كيف (how)؟ كيف يتم تبويب / فهرسة (cataloging) الكتب؟ كيف يتم ترتيبها على الرفوف؟ كيف يتعامل مسئول المكتبة مع كتاب أعيد للمكتبة؟

مثلا الإجابة على السؤال الأخير تعطيها الخوارزمية التالية لتنفيذ عملية إعادة كتاب للمكتبة.

* افحص (examine) آخر موعد مسموح به لإعادة الكتاب (due date) لمعرفة إن كان الكتاب متأخرا (late).

* إن كان الكتاب متأخراً :

احسب قيمة الغرامة (calculate fine)

أصدر ورقة الغرامة (issue fine slip)

* جَدِّد (update) سجلات المكتبة لتبيِّن أن الكتاب قد أُعيد.

* افحص قائمة الكتب المتحفظ عليها (check reserve list) لترى إن كان هناك أحد ينتظر هذا الكتاب.

* إن كان الكتاب على هذه القائمة فضعه على رف الكتب المحجوزة (reserve shelf) وإلا أعد وضع الكتاب على الرف الصحيح المقابل بناءً على نظام ترتيب كتب المكتبة على رفوفها.

وكل هذه الأنشطة بالطبع تُعَد غير مرئية (invisible) بالنسبة لمستخدِم المكتبة (library user) ، فهدف التصميم هو إخفاء (hiding) مستوى التنفيذ عن المستخدم.

العمليات على أنواع البيانات المجردة
Operations on Abstract Data Types

تنقسم العمليات الأساسية التي يمكن إجراؤها على نوع بيانات مجرد إلى أربعة أنواع : المنشِئات (constructors) المحوِّلات (mutators / transformers) والملاحِظات / المراقِبات (observers) والمكرِّرات (iterators).

وبأسلوب آخر فالعمليات الأربع الأساسية هي الإنشاء والتحويل والمراقبة والتكرير ، وتعني ما يلي :

* **المنشئ (constructor) :**

عملية إنشاء هدف/ شئ/ كائن (object / instance) جديد من نوع بيانات مجرد. وعادة نستدعي (invoke) هذه العملية على مستوى اللغة (language level) عن طريق إعلان (declaration) ما.

* والمحوِّل (transformer) :

عملية تغيير حالة (state) قيمة أو أكثر من قيم البيانات (data values) ، كإدخال (inserting) عنصر (item) في هدف (into an object) ، أو حذف (deleting) عنصر (item) من هدف ، أو جعل هدف هدفا خاويا (empty). والعملية التي تأخذ هدفين وتوحدهما (merges) في هدف ثالث يطلق عليها عملية تحويل ثنائية (binary transformer). [ملاحظة : في بعض الكتب يطلق على العمليات التي تقوم بإنشاء أهداف جديدة «منشئات بدائية» (primitive constructors) ، بينما يطلق على المحوِّلات (transformers) «منشئات غير بدائية» (nonprimitive constructors)].

* والمراقِب (observer) :

عملية تسمح لنا بمراقبة حالة قيمة أو أكثر من قيم البيانات (data values) دون تغييرها. وهناك عدة صور للمراقِبات ، منها : العبارات (predicates) التي تسأل ما إذا كانت خاصية (property) معينة متحققة / صحيحة (true) أم لا ، ودوال الانتقاء / الاختيار / الوصول (accessor / selector functions) التي تعيد (return) نسخة (copy) من عنصر معين / وحدة معينة (an item) في هدف (object) ما ، ودوال الملخصات (summary functions) التي تعيد معلومات (information) عن الهدف ككل (as a whole). ومن أمثلة العبارات (predicates) : دالة منطقية تعيد القيمة true إذا كان هدفٌ (object) ما خاويا (empty) والقيمة false إذا كان يحتوي على أي مركبات (components). ومن أمثلة دوال الانتقاء / الوصول : دالة تعيد نسخة من آخر عنصر / وحدة (item) وُضِع في البنية (structure). ومن أمثلة دوال الملخصات : دالة تعيد عدد العناصر / الوحدات في البنية.

* والمكرِّر (iterator) :

عملية تسمح لنا بتشغيل (processing) جميع المركبات (components) في بنية معطيات تتابعيا (sequentially). ومن المكرِّرات عمليات طباعة (printing) عناصر (items) قائمة (list) ، أو إعادة الوحدات المتتابعة في قائمة (successive list items). ويلاحظ أن المكررات لا تُعرَّف إلا على أنواع البيانات المبنية (structured data types).

التجريد والأنواع المبنية داخليا
(Abstraction and Built-In Types)

ذكرنا سابقا أن النوع البسيط الداخلي (built-in simple type) مثل int أو float يمكن أن ننظر إليه على أنه تجريد يُعَرَّف تنفيذه (implementation) بدلالة عمليات على مستوى الآلة (machine-level operations). وتنطبق النظرة نفسها على أنواع البيانات المركبة الداخلية (built-in composite data types) المتوفرة في لغات البرمجة لبناء أهداف بيانات (data objects). ونوع البيانات المجرد هو نوع يعطَي فيه اسم لمجموعة من وحدات/ عناصر بيانات (collection of data items). وأنواع البيانات المركبة (composite) قد تكون مبنية (structured) أو غير مبنية (unstructured). ونوع البيانات غير المبني هو مجموعة من المركبات (collection of components) غير المرتبة (not organized) بالنسبة لبعضها البعض. بينما نوع البيانات المبني هو مجموعة مرتبة من المركبات ، حيث يُحدِّد الترتيب الطريقة التي تُستخدم للوصول إلى مركبات البيانات المفردة.

ومن أمثلة الأنواع المركبة : السجلات (records / structs) ، والطبقات (classes) ، والمنظومات ذوات الأبعاد (dimensions) المختلفة. ويمكن للطبقات والسجلات أن تشتمل على دوال أعضاء (member functions)

بالإضافة إلى بيانات (data) ، والذي يهمنا هنا هو ترتيب البيانات. وتعد الطبقات والسجلات غير مبنية منطقيا (logically unstructured) ، بينما تعد المنظومات مبنية (structured).

وفيما يلي ننظر في هذه الأنواع على مختلف المستويات ، ونلقي أولا نظرة مجردة (abstract view) على البنية : كيف ننشئ (construct) متغيرات من هذا النوع ، وكيف نصل إلى المركبات المفردة في برامجنا. ثم على مستوى التطبيق نرى ماهية أنواع الأشياء التي يمكن أن تُصاغ (modeled) باستخدام كل بنية من هذه البنيات. وأخيرا ننظر في كيفية تنفيذ بعض هذه البنيات : كيفية تحويل دالة الوصول المنطقية (logical accessing function) إلى موضع / موقع (location) في الذاكرة (memory). وبالنسبة للأنواع الداخلية فإن النظرة المجردة هي الصيغة التركيبية (syntax) للنوع نفسه ، ويبقى مستوى التنفيذ مخفيا (hidden) داخل (within) البرنامج المترجم (compiler). ولا يحتاج المبرمج لفهم النظرة التنفيذية لأنواع البيانات المركبة المعرَّفة سلفا (predefined).

السجلات (records)

السجل غير متوفر في بعض لغات البرمجة كالفورتران مثلا ، بينما يستخدم بكثرة في لغة مثل الكوبول. وفي لغة C++ تنفذ السجلات عن طريق (structs). وطبقات C++ تعد تنفيذا آخر للسجلات.

* على المستوى المنطقي :

يعد السجل نوع بيانات مركبا مكوَّنا من مجموعة محدودة (finite collection) من عناصر ليس ضروريا أن تكون متجانسة وتسمى «مجالات» fields أو «عناصر» (elements / members). ويتم الوصول (accessing) مباشرة عن طريق انتقاء اسم العنصر أو المجال.

ونوضح فيما يلي الصيغة التركيبية (syntax) ومعناها (semantics) لمنتقى المركبات (component selector).

مثال ١-١ :

نفرض أن لدينا إعلان السجل (struct declaration) التالي :

```
struct CarType
{
    int year
    char maker[10];
    float price
};
CarType  myCar;
```

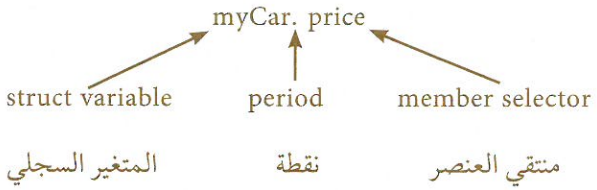
يتكون المتغير السجلي **myCar** (record variable) من ثلاثة عناصر/ مركبات (components) : الأول year من النوع int ، والثاني maker وهو منظومة رموز (array of characters) ، والثالث price من النوع float. وأسماء المركبات تكوِّن مجموعة منتقيى العناصر (member selectors). والشكل التالي (شكل ١-٣) يوضح صورة **myCar**.

. year	2005									
. maker	C	A	P	R	I	C	E	\0		
. price	9785. 75									

شكل ١-٣

السجل myCar

تتكون الصيغة التركيبية لمنتقي المركبة (component selector) من اسم المتغير السجلي ، تتبعه نقطة (period) ، يتبعها منتقي العنصر (member selector) للمركبة التي نختارها ، مثل :



فإذا ظهر هذا التعبير في الطرف الأيمن من عبارة إسناد مثل :

```
pricePaid = myCar. price;
```

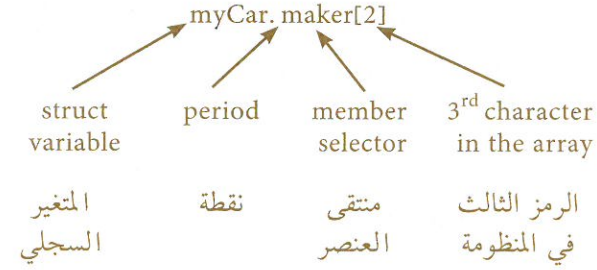
فإن قيمةً تستخرج (extracted) من موضع هذا العنصر.

وإذا ظهر في الطرف الأيسر من عبارة إسناد مثل :

```
myCar. price  = 12009. 500;
```

فإن قيمةً تُخزن في هذا العنصر من السجل.

ونلاحظ في هذا المثال (مثال ١-١) الذي معنا أن myCar. maker منظومة جميع مركباتها/ عناصرها من النوع char. ويمكننا الوصول إما إلى هذا المجال/ العنصر المنظومة (array member) ككل (مثلا myCar. maker) أو إلى أي من الرموز المفردة (individual characters) باستخدام مؤشر (index).



وفي لغة C++ يمكن للسجل (struct) أن :

i) يمرَّر (passed) كوسيط (parameter) لدالة [إما بالقيمة (by value) أو بالإسناد (by reference)].

ii) يُسنَد لسجل آخر من النوع نفسه.

iii) يكون قيمة دالة معادة (function return value).

تمرير الوسطاء في لغة C++ (C++ Parameter Passing)

يوجد في لغة C++ نوعان من الوسطاء الشكليين (formal parameters) : وسطاء ذووا قيمة (value parameters)، ووسطاء إسناد (reference parameters). والوسيط ذو قيمة هو وسيط شكلي يستقبل (receives) نسخة (copy) من محتويات (contents) الوسيط الفعلي المقابل (corresponding actual parameter / argument). ونظراً لأن الوسيط الشكلي يحتوي على (holds) نسخة من الوسيط الفعلي، فإن الوسيط الفعلي نفسه لا يمكن تغييره عن طريق الدالة (function) التي هو وسيط لها. وفي المقابل فإن وسيط الإسناد هو وسيط شكلي يستقبل موضع (location) / عنوان ذاكرة (memory address) الوسيط الفعلي المقابل. ونظراً لأن الوسيط الشكلي يحتوي على عنوان ذاكرة الوسيط الفعلي، لذا فإن الدالة يمكنها تغيير محتويات الوسيط الفعلي. وفي لغة C++ يُفترض (by default) أن تمرَّر المنظومات (arrays) بالإسناد (by reference)، بينما تمرَّر الوسطاء غير المنظومية (nonarray parameters) بالقيمة (by value).

ولتحديد أن وسيطاً شكلياً غَيْرَ منظومي (formal nonarray parameter) هو وسيط إسناد (وليس وسيطا ذا قيمة) فإننا نضع العلامة & (ampersand) يمين

اسم النوع (type name) في قائمة الوسطاء الشكليين (formal parameter list) ، كما هو مبين في المثال التالي :

مثال ١-٢ :

نفرض أن لدينا الإعلانين التاليين عن الدالتين **AdjustForInflation**, **LateModel** :

```
void AdjustForInflation (CarType& car, float perCent)
// Increases price by the amount specified in perCent.
{
   car. price = car. price * perCent + car. price;
}
bool LateModel (CarType car, int date)
// Returns true if the car's model year is later than or
// equal to date; returns false otherwise.
}
   return car. year >= date;
{
```

الدالة **AdjustForInflation** تقوم بتغيير عنصر البيانات (data member) price في الوسيط الشكلي car ، وبالتالي فإن car يجب أن تكون وسيط إسناد. وداخل جسم الدالة (body of the function) يكون car. price هو العنصر price في الوسيط الفعلي.

وأما الدالة **LateModel** فإنها تختبر car دون أن تغيرها ، ولذلك فإن car يجب أن تكون وسيطا ذا قيمة. وداخل الدالة فإن car. year يكون نسخة من الوسيط الفعلي في الدالة المستدعية (caller).

* على مستوى التطبيق : تفيد السجلات كثيرا في صياغة أهداف (modeling objects) لها عدد من الخصائص (characteristics). وهذا النوع من البيانات (data type) يسمح لنا بتجميع أنواع متعددة / مختلفة من البيانات عن هدف معين ، والإشارة إلى الهدف بأكمله (the whole object) باسم مفرد (single name). كما يمكننا أيضا الإشارة إلى الأعضاء / العناصر (members) المختلفة في الهدف بالاسم.

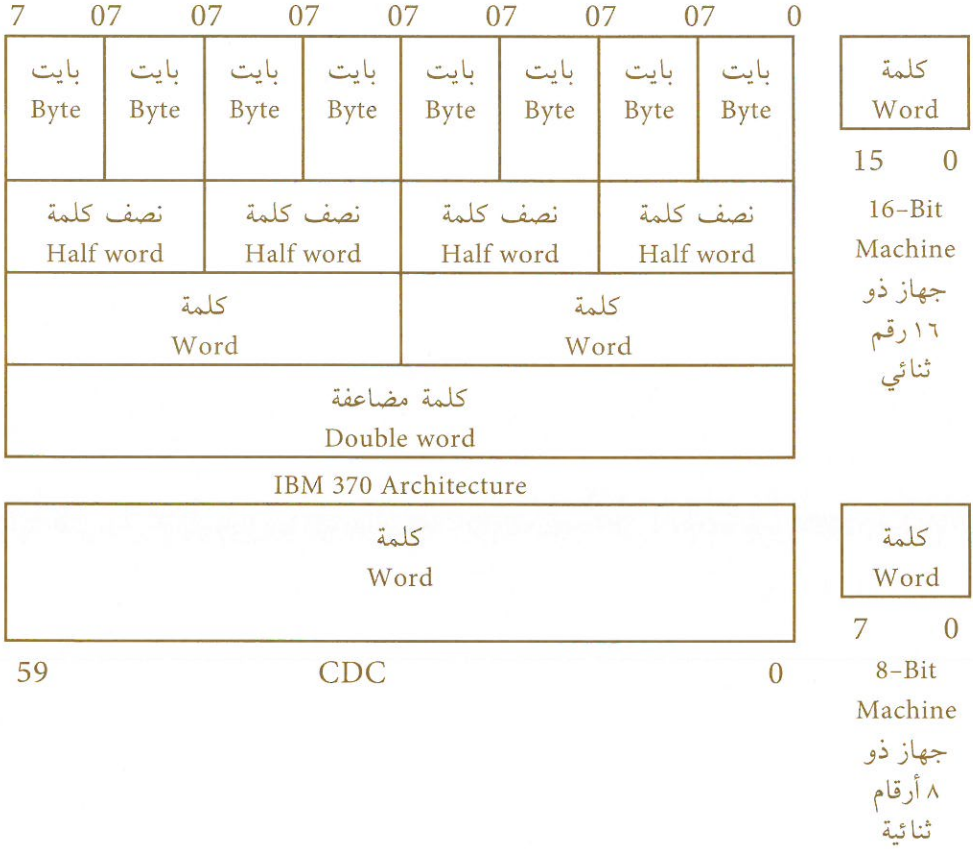
وتفيد السجلات أيضا في تعريف بنى معطيات (data structures) أخرى ، حيث تسمح للمبرمجين بالجمع (combining) بين معلومات (information) عن البنية (structure) ، وتخزين (storage) العناصر (elements). وبهذا الأسلوب نستخدم السجلات بصورة مكثفة حينما نقوم بتمثيل (representation) بنيات المعطيات المعرَّفة في برامجنا الخاصة (our own programmer-defined data structures).

* على مستوى التنفيذ : يجب القيام بأمرين لتنفيذ نوع بيانات مركب داخلي (built-in composite data type) ، وهذان الأمران هما :

(1) حجز (reserving) خلايا ذاكرة (memory cells) للبيانات (data).

(2) تحديد (determining) دالة الوصول (accessing function). ودالة الوصول هي قاعدة (rule) تخبر المترجم (compiler) ونظام وقت التشغيل (run-time system) أين يقع (is located) عنصر معين (individual element) في بنية المعطيات. وقبل أن ندرس مثالا متكاملا دعنا ننظر أولا للذاكرة. ومعلوم أن وحدة الذاكرة (unit of memory) المخصصة (assigned) للاحتفاظ بقيمة (holding a value) تعتمد على الماكينة / الجهاز (is machine dependent).

الشكل التالي يعطي وحدات ذاكرة تكوينية (memory configurations) متنوعة مختلفة.



شكل ١-٤

وحدات تكوينية مختلفة بالذاكرة

different memory configurations

ومن الناحية العملية فإن الشكل التكويني بالذاكرة أمر يهم كاتب البرنامج المترجم (compiler writer). ومن أجل العمومية فسنستخدم الاصطلاح «خلية» (cell) لتمثيل موضع (location) في الذاكرة بدلا من «كلمة» (word) أو «بايت» (byte). وفيما يلي سنفترض أن العدد الصحيح (integer) أو الرمز (character) يتم تخزينه (stored) في خلية واحدة ، وأن العدد ذا النقطة العائمة (floating-point number) يتم تخزينه في خليتين. (وهذا الفرض غير دقيق في لغة C++ ، ولكننا نستخدمه هنا لتبسيط المناقشة).

ومعلوم أن عبارات الإعلان (declaration statements) في أي برنامج تخبر البرنامج المترجم (compiler) عدد الخلايا التي نحتاجها لتمثيل (representing) السجل. ثم إن اسم السجل يرتبط (is associated with) بخصائص السجل (characteristics of the record). وهذه الخصائص تشمل :

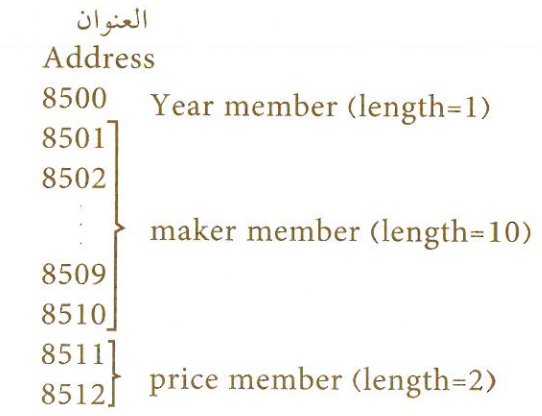
* عنوان الأساس الخاص بالسجل (base address of the record) : وهو الموقع في الذاكرة (location in memory) الخاص بأول خلية من خلايا السجل.

* جدولا (table) يحتوي على عدد المواقع في الذاكرة (number of memory locations) الذي يحتاجه كل عضو / عنصر من عناصر السجل.

والسجل يشغل (occupies) عادة قالبا (block) / مجموعة من الخلايا المتعاقبة (consecutive cells) في الذاكرة. وتقوم دالة الوصول إلى السجل (record's accessing function) بحساب موقع خلية معينة (particular cell) من منتقي عنصر اسمه معطَى (named member selector). والسؤال الأساسي هو : أي خلية (أو خلايا) في هذا القالب من الخلايا المتعاقبة نريد؟

العنوان الأساسي (base address) للسجل - كما ذكرنا - هو عنوان أول عنصر / عضو (number) في السجل. وللوصول إلى أي عنصر نحتاج لمعرفة القدر الذي نتجاوزه / نتخطاه (skip) من السجل حتى نصل إلى العنصر المطلوب. والإشارة (reference) إلى عنصر ما في السجل تجعل البرنامج المترجم (compiler) يفحص جدول الخصائص (characteristics table) لتحديد مدى تشعُّب / انحراف / بُعد / إزاحة (offset) العنصر عن بداية السجل. وعندها يستطيع المترجم حساب عنوان العنصر عن طريق إضافة هذه الإزاحة إلى عنوان الأساس. والشكل التالي (شكل ١-٥) يوضح هذا الجدول للنوع السجلي **CarType**.

العنصر	الطول	الإزاحة
Member	Length	Offset
year	1	0
maker	10	1
price	2	11



شكل ١-٥

مستوى تنفيذ النوع السجلي CarType

Implementation-level view of CarType

إذا فرضا أن العنوان الأساسي للسجل myCar هو 8500 فإننا سنجد مجالات (fields) أو عناصر هذا السجل في العناوين التالية :

Address of myCar. year = 8500 + 0 = 8500
Address of myCar. marker = 8500 + 1 = 8501
Address of myCar. price = 8500 + 11 = 8511

وقد ذكرنا سابقا أن السجل يُعَدُّ نوع بيانات غير مبني (unstructured) ، إلا أن منتقي المركبات (component selector) يعتمد على المواضع النسبية (relative positions) لعناصر السجل وهذا صحيح : فإن السجل يُعَد نوع بيانات مبنيا (structured) إذا نظرنا إليه (viewed) من وجهة نظر التنفيذ (implementation perspective). إلا أنه من وجهة نظر المستخدم (user's view) يعد غير مبني. فإن المستخدم يصل إلى عناصر السجل بالاسم (by name) وليس بالموضع (by position). فمثلا إذا عرَّفنا النوع السجلي **CarType** كما يلي

```
struct CarType
{
    charmake[10];
    float price ;
    intyear;
} ;
```

فإن عبارات البرنامج التي تتعامل مع عناصر **CarType** سوف لا تتغير.

المنظومات أحادية البعد (One-Dimensional Arrays)

المستوى المنطقي (Logical Level) :

المنظومة أحادية البعد هي نوع بيانات مركب مبني (structured composite data type) مكوَّن من مجموعة محدودة ثابتة الحجم من عناصر متجانسة مرتبة (finite, fixed-size collection of ordered homogeneous elements) يسهل الوصول إليها مباشرة. وتشير كلمة «محدودة» (finite) إلى أنه يمكن تحديد / تعريف «عنصر» أخير (a last element). بينما «ثابتة الحجم» (fixed size) تعني أن حجم المنظومة يجب أن يكون معلوما سلفا / مقدما / من البداية (in advance). على أن هذا لا يعني أن جميع الأماكن (slots) في المنظومة يجب أن تحتوي على قيم ذات معنى (meaningful values). وأما كلمة «مرتبة» (ordered) فتعني أن هناك عنصراً أولا (a first element) ، وعنصرا ثانيا (a second element) ،. .. وهكذا [الموضع النسبي (relative position) مرتب ، وليس ضروريا القيم المخزونة هناك]. ونظرا لأن العناصر الموجودة في أي منظومة يجب أن تكون جميعها من النوع نفسه ، فهي متجانسة فيزيائيا (physically homogeneous) ، أي أنها جميعها من نوع البيانات نفسه. وعموما فمن المرغوب فيه أن تكون عناصر المنظومة متجانسة منطقيا (logically) أيضا ، أي أن جميع العناصر يكون لها الغرض نفسه (same purpose). [فمثلا إذا حفظنا قائمة (a list) من العناصر في منظومة أعداد صحيحة بحيث حُفظ طول (length) القائمة - وهو عدد صحيح - في الموضع الأول من المنظومة (first array slot) ، فإن عناصر المنظومة ستكون متجانسة فيزيائيا (كلها أعداد صحيحة) ولكنها غير متجانسة منطقيا (الغرض من العدد في الموضع الأول مختلف عن الغرض من أي عدد آخر)].

وآلية (mechanism) اختيار أي مركبة (component) في المنظومة هي الوصول المباشر (direct access) ، الذي يعني إمكانية الوصول مباشرة إلى أي

عنصر دون الوصول أولا إلى العناصر السابقة له. ويمكن تحديد العنصر المطلوب الوصول إليه عن طريق مؤشر (index) يعطي الموقع النسبي في مجموعة العناصر. وسنناقش فيما بعد كيف تستخدم لغة C++ هذا المؤشر وبعض خصائص المنظومة لتحدد بالضبط أين تجد العنصر في الذاكرة.

وأما بالنسبة للسؤال ، ما هي العمليات (operations) المعرَّفة (defined) بالنسبة للمنظومات؟ فإن افتقرت (lacked) اللغة التي نستخدمها إلى المنظومات المعرَّفة سابقا (predefined arrays) وكنا نحن الذين نُعرِّف المنظومات ، فإننا نحتاج لتعريف / لتحديد ثلاث عمليات على الأقل (ونعرضها هنا كاستدعاءات دوال C++) لإنشاء منظومة ، وتخزين قيمة في منظومة ، واستعادة قيمة من منظومة.

```
CreateArray(anArray, numberOfSlots);
// Create array anArray with numberOfSlots locations.

Store (anArray, value, index) ;
// Store value into anArray at position index.

Retrieve (anArray, value, index) ;
// Retrieve into value the array element found at position index.
```

ونظرا لأن المنظومات هي أنواع بيانا معرَّفة سابقا في لغة البرمجة C++ ، فإن ، هذه اللغة تستخدم طريقة خاصة لإنجاز (performing) كل من هذه العمليات ، حيث تسمح تركيبة (syntax) اللغة باستخدام دالة إنشاء بسيطة (primitive constructor) لإنشاء المنظومات في الذاكرة ، مع استخدام مؤشرات (indexes) للوصول مباشرة إلى أي عنصر في المنظومة.

ففي لغة C++ يؤدي الإعلان (declaration) عن منظومة وظيفة عملية إنشاء بسيطة. فمثلا يمكن الإعلان عن منظومة أحادية البعد بالعبارة.

```
int numbers [10] ;
```

وهذا الإعلان يعرف مجموعة مرتبة خطيا (linearly ordered collection) مكونة من 10 عناصر / أعداد صحيحة (integer items).

وأي عنصر في المنظومة **numbers** يمكن الوصول إليه مباشرة عن طريق موقعه النسبي في المنظومة ، حيث الصيغة التركيبية العامة لمنتقي المركبة (component selector) هي :

array-name [index-expression]

numbers

[0]	First element	العنصر الأول
[1]	Second element	العنصر الثاني
[2]	Third element	العنصر الثالث
⋮	⋮	
[9]	Last element	العنصر الأخير

حيث «تعبير المؤشر» (index-expression) يكون من النوع التكاملي (integral type) [أي : char, short, int, long أو النوع التعددي (enumeration type)]. وقد يكون هذا التعبير بسيطا كثابت (constant) أو اسم متغير ، أو معقدا كمزيج من المتغيرات والمؤثرات (operators) واستدعاءات الدوال (function calls). المهم أن تكون نتيجة التعبير قيمة صحيحة.

وفي لغة C++ يتراوح مدى المؤشر (index range) بين 0 و(حجم المنظومة- ١). ففي حالة المنظومة numbers تقع قيمة المؤشر بين 0 , 9. وفي لغات أخرى يمكن للمستخدم أن يحدد صراحة مدى المؤشر.

وأما معنى (semantics / meaning) منتقى المركبة فهو «عَيِّن موضع العنصر المرتبط بتعبير المؤشر في مجموعة العناصر المُعَرَّفة باسم المنظومة». ويمكن أن يُستخدم منتقى المؤشر بطريقتين :

1) تحديد مكان تُنسَخ فيه قيمة :

مثل :

```
numbers[2] = 5 ;
```

أو

```
cin >> numbers[2] ;
```

2) تحديد مكان تُستعاد / تُسترجع (retrieved) منه قيمة :

مثل :

```
value = numbers[4] ;
```

أو

```
cout << numbers[4] ;
```

وإذا ظهر منتقى المركِّبة في الطرف الأيسر في عبارة الإسناد فإنه يكون مستخدَما كمحوِّل (transformer) حيث أن بنية المعطيات (data structure)

تتغير. بينما إذا ظهر منتقى المركَّبة في الطرف الأيمن في عبارة الإسناد فإنه يكون مستخدَما كملاحظ (observer) حيث أنه يعيد القيمة المخزونة في موضع ما في المنظومة دون تغييرها. ويُعَد الإعلان عن منظومة والوصول إلى عناصرها من العمليات المعرَّفة سلفا في جميع لغات البرمجة عالية المستوى (high-level) تقريبا.

وفي لغة C++ يمكن تمرير المنظومات كوسطاء (parameters) [بالإسناد (by reference) فقط] ، ولكن لا يمكن إسناد بعضها لبعض ، كما لا يمكن استخدامها كنوع القيمة التي تعيدها دالة.

تمرير المنظومات أحادية البعد كوسطاء

C++ One-Dimensional Arrays as Parameters

في لغة C++ يمكن للمنظومات أن تكون وسطاء إسناد (reference parameters) فقط ، ولا يمكن تمرير منظومة بالقيمة (by value). ولذلك فإن الرمز "&" (ampersand) الذي يوضع يمين النوع يُحذف. وعندما تكون منظومة وسيطا شكليا (formal parameter) فإن العنوان الأساسي (base address) للمنظومة [العنوان في الذاكرة الخاص بأول موضع (first slot) في المنظومة] يُمرَّر فعلا إلى دالة. وهذا صحيح سواء كان بُعْد المنظومة أحاديا أو أكثر. وحين نعلن عن وسيط أنه منظومة أحادية البعد فإن البرنامج المترجم (compiler) يحتاج فقط لمعرفة أن الوسيط عبارة عن منظومة دون الحاجة إلى معرفة حجمها (size). وإذا ذكرنا حجم هذا الوسيط الشكلي فإن البرنامج المترجم يهمله. ويجب أن تضمن العبارات التي تظهر في الدالة التي تقوم بتشغيل المنظومة أن مواضع المنظومة (array slots) المسموح بها فقط هي التي تتم الإشارة إليها. ولذلك فإن وسيطا آخر يمرر عادة للدالة لتحديد عدد مواضع المنظومة التي سيتم تشغيلها ، كما توضح ذلك الدالة التالية :

```
int SumValues(int values [ ] , int numberOfValues)
// Returns the sum of values [0] through values [numberOfValues-1].
{
    int sum = 0 ;
    for (int index = 0 ; index < numberOfValues ; index ++)
    sum = sum + values [index] ;
    return sum ;
}
```

وإذا كنا نقوم بتمرير المنظومات بصفة دائمة كوسطاء إسناد فكيف نضمن حماية الوسيط الفعلي من أي تغييرات غير مقصودة؟ فمثلا في الدالة السابقة **SumValues** مطلوب فقط فحص (inspecting) وليس تغيير / تعديل (modifying) الوسيط values ، فكيف نحميه من أي تغيير؟ يمكننا ذلك عن طريق الإعلان عنه أنه وسيط ثابت (const parameter) كما يلي :

```
int SumValues (const int values [ ], int numberOfValues)
```

والآن داخل جسم الدالة (function body) أي محاولة لتغيير محتويات المنظومة values ستؤدي إلى خطأ تركيبي (syntax error).

المستوى التطبيقي (Application Level) :

المنظومة أحادية البعد هي البنية الطبيعية لتخزين قائمة (list) من عناصر بيانات متماثلة كقائمة أرقام تليفونات أو قائمة أسعار أو قائمة سجلات طلاب أو قائمة رموز (characters) [أي سلسلة رموز (a string)].

مستوى التنفيذ (Implementation Level) :

عندما نستخدم منظومة في برنامج بلغة C++ فإن المستخدم لا يشغل باله بتفاصيل التنفيذ ، وبالتالي بالتفاصيل المذكورة فيما يلي.

عبارة الإعلان عن منظومة تخبر البرنامج المترجم كم عدد الخلايا التي نحتاجها لتمثيل هذه المنظومة. ثم يُربط اسم المنظومة بخصائصها. وهذه الخصائص تشمل ما يلي :

* عدد العناصر «Number».
* الموقع في الذاكرة الخاص بأول خلية في المنظومة ، ويُطلَق عليه «العنوان الأساسي للمنظومة» «Base» (base address of the array) .
* عدد المواقع في الذاكرة التي نحتاجها لكل عنصر من عناصر المنظومة «SizeOfElement».

وعادة يتم تخزين معلومات خصائص المنظومة في جدول يُطلَق عليه : «واصف منظومة» (an array descriptor) / «متجه تكميلي / متجه إضافي» (dope vector). وعندما يصادف البرنامج المترجم إشارة إلى عنصر منظومة فإنه يستخدم هذه المعلومات ليولِّد عبارات تحسب موقع العنصر في الذاكرة وقت التشغيل (at run time).

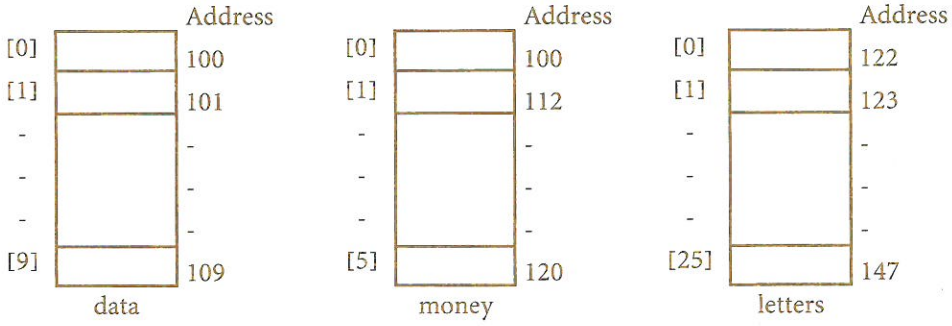
وأما كيف تُستخدم خصائص المنظومة لحساب عدد الخلايا التي نحتاجها ولتوليد دوال الوصول إلى المنظومات التالية ، فإننا - كما سبق - نفرض للبساطة أن العدد الصحيح أو الرمز (character) يتم تخزينه في خلية واحدة ، والعدد ذا النقطة العائمة يتم تخزينه في خليتين. فإذا كانت لدينا الإعلانات التالية :

```
int data [10] ;
float money [6] ;
char letters [26] ;
```

فإن هذه المنظومات تكون لها الخصائص التالية :

	data	money	letters
Number	10	6	26
Base	غير معلوم	غير معلوم	غير معلوم
SizeOfElement	1	2	1

وإذا فرضنا أن البرنامج المترجم في لغة C++ يعطي للمتغيرات خلايا في الذاكرة بترتيب تتابعي (sequential order) ، وأن أول خلية فارغة / متوفرة (available) تالية في الذاكرة حين التعرض للإعلانات السابقة عن المنظومات هي الخلية التي عنوانها 100 ، فإن إسناد خلايا الذاكرة للمنظومات السابقة يبدو كما يلي :



والآن نكون قد حددنا العنوان الأساسي (base address) لكل منظومة : 100 للمنظومة **data** ، 110 للمنظومة **money** ، 122 للمنظومة **letters**. ولذلك فإن ترتيب المنظومات في الذاكرة يؤدي إلى العلاقات المذكورة في الجدول التالي كأمثلة :

عنصر المنظومة المعطَى	العنوان الذي يجب أن يصل إليه البرنامج
Given	The program must access
data [0]	100
data [8]	108
letters [1]	123
letters [25]	147
money [0]	110
money [3]	116

وفي لغة C++ فإن دالة الوصول (accessing function) التي تعطي موضع عنصر في منظومة أحادية البعد مرتبط (associated with) بالتعبير (expression) Index هي :

Address (Index) = Base + Offset of the element at position index

أي أن :

عنوان (Index) = العنوان الأساسي + إزاحة العنصر عن الموضع Index والصيغة العامة لحساب الإزاحة (offset) هي :

Offset = index *SizeOfElement

وبالتالي فإن دالة الوصول كاملة تُعطَى بالعلاقة :

Address(Index) = Base + Index *SizeOfElement

والآن نطبق هذه الصيغة لنحصل على نتائج الجدول السابق الذي يعطي بعض الأمثلة لعناوين بعض العناصر.

العنوان	دالة الوصول	العنصر
Address	Base + Index *SizeOfElement	
= 100	100 + (0 * 1)	data [0]
= 108	100 + (8 * 1)	data [8]
= 123	122 + (1 * 1)	letters [1]
= 147	122 + (25 * 1)	letters [25]
= 110	110 + (0 * 2)	money [0]
= 116	110 + (3 * 2)	money [3]

المنظومات ثنائية البعد (Two-Dimensional Arrays)

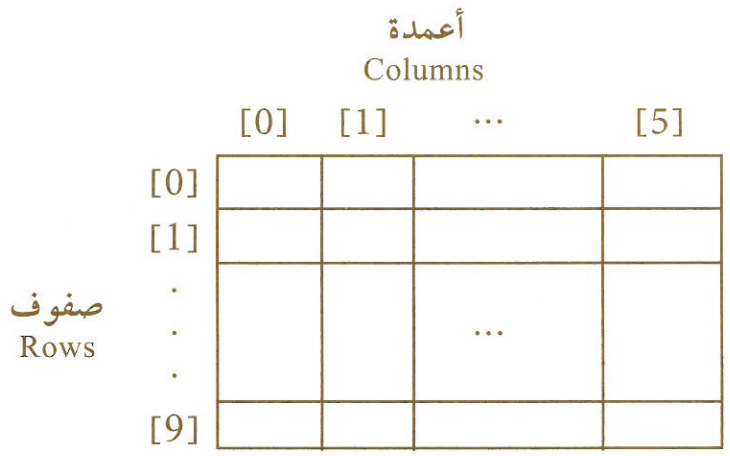
المستوى المنطقي (Logical Level)

معظم ما قيل سابقا عن النظرة المجردة للمنظومة أحادية البعد ينطبق أيضا على المنظومات متعددة الأبعاد. وتعد المنظومة ثنائية البعد نوع بيانات مركب (composite) مكوَّن من مجموعة ثابتة الحجم محدودة / منتهية (finite fixed-size collection) من عناصر متجانسة (homogeneous elements) مُرتَّبة (ordered) في بُعدين. ويتم انتقاء مركباتها عن طريق الوصول المباشر باستخدام مؤشرين (pair of indexes) يحددان العنصر المطلوب بإعطاء موضعه النسبي في كل من البعدين.

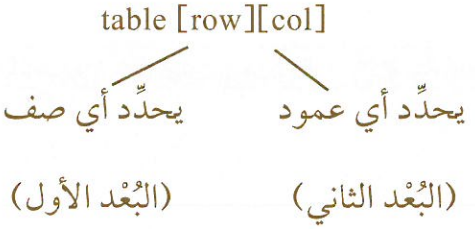
وتعد المنظومة ثنائية البعد طريقة طبيعية لتمثيل البيانات التي يمكن أن ننظر إليها منطقيا كجدول مكون من أعمدة وصفوف. والمثال التالي يوضح الصيغة التركيبية للإعلان عن منظومة ثنائية البعد :

```
int table [10] [6] ;
```

والصورة التجريدية (abstract picture) لهذه البنية عبارة عن شبكة ذات صفوف وأعمدة :



ومنتقى المركِّبة (component selector) بالنسبة للمنظومة ثنائية البعد يظهر بالشكل التالي :



تمرير المنظومات ثنائية البعد كوسطاء

يتم تخزين المنظومات ثنائية البعد في لغة C++ بترتيب الصفوف (in row order) ، بمعنى أن جميع العناصر في صفٍ ما يتم تخزينها ، تتبعها جميع العناصر في الصف التالي ،. . وهكذا. وللوصول إلى أي صف غير الصف الأول فإن البرنامج المترجم يجب أن يكون قادرا على حساب موضع بداية أي صف. ونتيجة هذا الحساب تعتمد على عدد العناصر الموجودة في أي صف. فالصف الثاني يبدأ عند العنوان الأساسي مضافا إليه عدد العناصر الموجودة في أي صف. وكل صف تالي يبدأ عند عنوان الصف السابق مضافا إليه عدد العناصر في أي صف.

فالبُعْد الثاني - وهو عدد الأعمدة - يخبرنا كم عدد العناصر الموجودة في أي صف. ولذلك فإن حجم (size) البُعْد الثاني يجب أن يظهر في الإعلان عن الوسيط الشكلي (formal parameter) بالنسبة لمنظومة ثنائية البُعْد ، كما يلي :

```
int ProcessValues(int values [ ] [5])
{
:
:
}
```

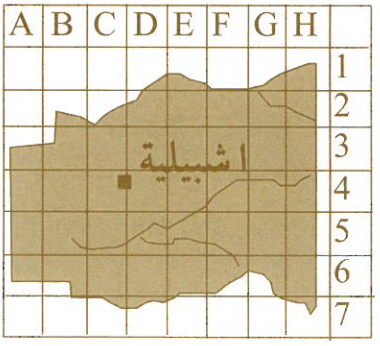
الدالة **ProcessValues** تصلح لمنظومة ذات أي عدد من الصفوف طالما أن عدد أعمدتها 5 بالضبط. أي أن حجم/ سعة (size) البُعْد الثاني لكل من منظومة الوسيط الفعلي ومنظومة الوسيط الشكلي يجب أن يكون هو نفسه ، أي أن يكون الحجمان متطابقين. وحتى نضمن أن يكون لمنظومة الوسيط الفعلي ثنائية البعد ومنظومة الوسيط الشكلي ثنائية البعد الحجم نفسه (same size) فإننا نستخدم عبارة **typedef** لتعريف نوع منظومة ثنائية البعد ، ثم نعلن عن كل من الوسيط الفعلي والوسيط الشكلي من هذا النوع ، كما توضح ذلك مثلا العبارات التالية :

```
const int NUM_ROWS = 5 ;
const int NUM_COLS  = 4 ;
typedef float TableType [NUM_ROWS] [NUM_COLS] ;

int ProcessValues (TableType table) ;

TableType mine ;
TableType yours ;
```

عبارة **typedef** تدلنا على ارتباط منظومة ثنائية البعد من أعداد ذوات نقطة عائمة مكونة من خمسة صفوف وأربعة أعمدة باسم النوع **TableType** (type name والمنظومتان **mine, yours** هما مثالان لهذه المنظومة. وأي وسيط فعلي بالنسبة للدالة **ProcessValues** يجب أن يكون من النوع **TableType**. وتعريف الأنواع بهذه الطريقة يجنبنا حدوث أي مشاكل عدم توافق (mismatch).

المستوى التطبيقي :

كما ذكرنا سابقا فإن المنظومة ثنائية البعد هي بنية المعطيات المثالية المناسبة لنمذجة (modeling) بيانات (data) مبنية منطقيا (logically structured) كجدول به صفوف وأعمدة والبعد الأول يمثل الصفوف والثاني يمثل الأعمدة. وكل عنصر في المنظومة يحتوي على قيمة ، وكل بُعْد يمثل علاقة (relationship). فعلى سبيل المثال نحن عادة نمثل الخريطة (map) كمنظومة ثنائية البعد ، ونحدد موقع شيء ما (مدينة أو شارع أو نهر. ..) بوقوعه في مربع ناتج عن التقاء صف معين وعمود معين.



وكما في حالة المنظومات أحادية البعد فإن العمليات المتاحة بالنسبة لأهداف المنظومات ثنائية البعد (2-dim-array objects) محدودة جدا ، حيث تشتمل فقط على الإنشاء (creation) والوصول المباشر (direct access).

المستوى التنفيذي :

يشتمل تنفيذ المنظومات ثنائية البعد على عملية إسقاط مؤشرين على خلية ذاكرة معينة (mapping of two indexes to a particular memory cell). ودوال الإسقاط (mapping functions) هنا تُعَد أكثر تعقيدا من تلك الدوال في حالة المنظومات أحادية البعد. ولن نشغل أنفسنا هنا في هذا المقرر بكيفية كتابة دوال الوصول (accessing functions) هذه ، فليس هدف دراستنا الآن هو كيف تصبح كاتبا لبرامج الترجمة (compiler writer) ، بل يكفينا هنا أن تدرك قيمة وأهمية إخفاء المعلومات وتغليفها (information hiding and encapsulation).

رأينا سابقا أن المصطلح «نوع بيانات مجرد» ADT (Abstract Data Type) يُقصد به نوع بيانات تُحدَّد (specified) فيه الأشياء والعمليات عليها بطريقة مستقلة لا علاقة لها بـ (independent of) / منفصلة عن (separated from) كيفية تمثيل (representation) الأشياء ، وكيفية تنفيذ (implementation) العمليات ، أي أنه مجرد نموذج رياضي مجرد (abstract math. Model) لا علاقة له بكيفية التمثيل والتنفيذ.

ملاحظة :

هذا التمثيل المجرد يفيد المستخدم (user) الذي لا يهمه التفاصيل الداخلية (ماذا يحدث داخل الحاسب؟ ما هو التمثيل الداخلي (internal representation) للأشياء؟ كيف يتم التنفيذ؟) وإنما يهمه ماذا يعمل ، أما كيفية التنفيذ فيمكن أن تكون بأكثر من طريقة واحدة (تبعا لاعتبارات مختلفة كالكفاءة وغيرها) كما يتضح من المثالين التاليين :

مثال ١-٣ :

الرصة (stack) تعد مثالا لنوع بيانات مجرد ADT.

رصة S

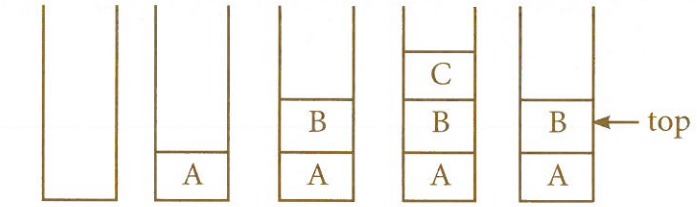
والرصة هي قائمة مرتبة من أشياء / عناصر (ordered list of objects) ، ويتم إدخال (insertion) العناصر في الرصة أو حذفها (deletion) منها من جهة / نهاية واحدة فقط تسمى أعلى الرصة (top of stack) ، وتكون الرصة ابتداءً خالية (empty) ، ويمكننا كتابة العلاقة

$$S = (a_1 , a_2 , ... , a_i , ... , a_n)$$

حيث a_1 هو العنصر السفلي (bottom element) ،

a_n هو العنصر العلوي (top element)

ونستطيع تمثيل الرصة التي تمت فيها عمليات إضافة عنصر A ثم عنصر B ثم عنصر C ثم حذف العنصر C ، بالشكل التالي (حيث نبدأ بالرصة فارغة) :



رصة فارغة

ويمكننا تعريف العمليات التالية على الرصة :

١) **push** : لإضافة (add) عنصر (item) عند قمة / أعلى الرصة.

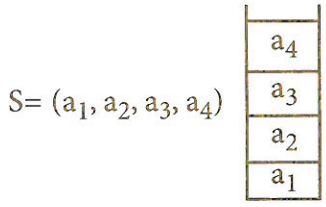
٢) **pop** : لحذف (delete) عنصر من أعلى الرصة.

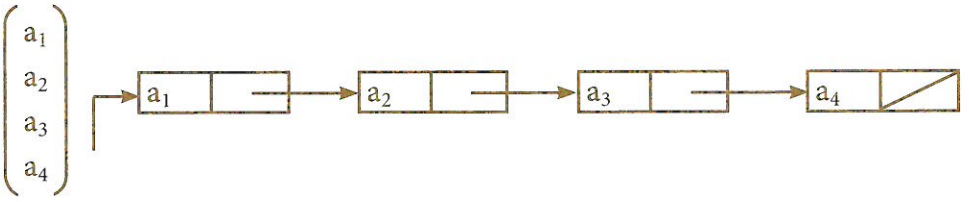
٣) **create—empty—s** : لإنشاء رصة فارغة.

٤) **is—empty** : دالة منطقية لاختبار ما إذا كانت الرصة فارغة أم لا.

٥) **top—of—stack** : دالة تعيد قيمة العنصر الموجود أعلى الرصة.

هذا التحديد (specification) للرصة لا يذكر كيفية تنفيذ الرصة ، ولا يهم مستخدم الرصة كيفية هذا التنفيذ. وقد يكون التنفيذ مثلا عن طريق استخدام منظومة (array) أو قائمة مترابطة (linked list). كما يوضح ذلك الشكل التالي :





تنفيذ S باستخدام قائمة مترابطة

تنفيذ S باستخدام منظومة

مثال ١-٤ : المجموعة (set) تعد مثالا لنوع بيانات مجرد ADT.

وهي عبارة عن مجموعة أشياء (set of objects) وبعض العمليات المعرفة عليها مثل :

* إنشاء مجموعة (create a set)
* الاتحاد والتقاطع والفرق : $\cup, \cap, \setminus$
* مقارنة بين مجموعتين : $\supseteq, \supset, \subset, \subseteq, !=, ==$
* اختبار الانتماء : $\in$

أما كيفية تنفيذ المجموعة فيمكن أن تتم باستخدام إحدى الوسائل / الطرق التالية :

* متجه الوحدات الثنائية (bit vector)
* منظومة (مع دوال functions / عمليات operations)
* قائمة مترابطة (مرتبة أو غير مرتبة)
* شجرة بحث ثنائية BST
* شجرة B
* شجرة AVL
* بعثرة (hashing)

وسوف نشير إلى بعض هذه الوسائل فيما بعد بإذن الله ، وعموما تتفاضل هذه الطرق / الوسائل فيما بينها من حيث الكفاءة (efficiency) أو الأداء (performance). كما أشرنا سابقا إلى أن :

بنية معطيات / هيكل بيانات DS (A Data Structure)

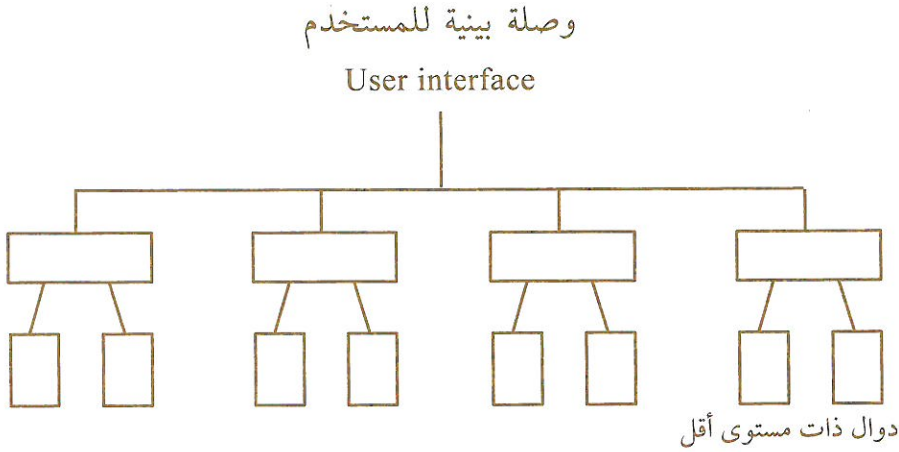
عبارة عن تنفيذ (an implementation) لنوع بيانات مجرد (an ADT) باستخدام عدة أنواع بيانات (collection of data types) ونوع البيانات المجرد ADT. ويجب أن تشتمل أي بنية معطيات DS على ما يلي :

(أ) بنية تخزين (storage structure)

(ب) مجموعة عمليات للمستخدم (user operations) (عبارة عن دوال / إجراءات) ، أي وصلة بينية للمستخدم (user interface) [يصل إليها المستخدم لتنفيذ عمليات نوع البيانات المجرد ADT]

(ج) مجموعة عمليات / دوال ذوات مستوى أقل (lower level routines / operations) للتنفيذ الفعلي لعمليات نوع البيانات المجرد ADT .

ملاحظة : هذه العمليات ذوات المستوى الأقل لا يراها المستخدم ، وهي تساعدنا في تنفيذ مجموعة العمليات (ب).



تقييم بنى المعطيات والخوارزميات

Evaluation of Data Structures and Algorithms

هناك عدة معايير (criteria) أو خصائص تستخدم لتقييم بنى المعطيات والخوارزميات والمقارنة بينها ، وأهم هذه المعايير المستخدمة والتي تؤخذ في الاعتبار :

١) البساطة (simplicity).
٢) إثبات الصحة (proving the correctness).
٣) الكفاءة (efficiency) ، وهذه تأخذ في الاعتبار كلا من :

أ) وقت التنفيذ (CPU time).
ب) حيز التخزين (storage).

تقييم أداء / كفاءة الخوارزميات
(Performance / Efficiency Evaluation of Algorithms)

درجتا التعقيد الزمنية والمكانية لخوارزمية :
(Time Complexity and Space Complexity of an Algorithm)

عادة نستخدم مقياسين للمقارنة بين عدة خوارزميات أو عدة بنى معطيات ، وهذان المقياسان هما : درجة التعقيد الزمنية ، ودرجة التعقيد المكانية ، ونقصد بهما ما يلي :

أ) درجة التعقيد الزمنية (Time Complexity)

هي عدد العمليات (number of operations) التي يحتاجها تنفيذ الخوارزمية لحل مسألة معينة كدالة في حجم المسألة (funtion of the problem size).

ب) درجة التعقيد المكانية (Space Complexity)

هي عدد وحدات / خلايا التخزين / الذاكرة / (number of memory storage cells / bytes) التي يحتاجها تنفيذ الخوارزمية لحل مسألة معينة كدالة في حجم مسألة.

ويمكننا أن نقول باختصار إن درجة التعقيد الزمنية هي كمية الزمن (amount of computer time) اللازم لإنهاء تنفيذ الخوارزمية ، ودرجة التعقيد المكانية هي حيز الذاكرة (amount of computer memory) اللازم لذلك.

وهناك صور أخرى لدرجة التعقيد نذكرها فيما يلي :

(i) درجة التعقيد في أسوأ حالة (Worst_case Complexity)

هي أقصى عدد من العمليات (max. no. of operations) التي يتم إجراؤها بالخوارزمية لحل مسألة معينة [كدالة في حجم المدخلات (function of input size n) / لجميع المدخلات ذات حجم معين n].

(ii) درجة التعقيد المتوسطة (Average_case Complexity)

هي متوسط عدد العمليات التي يتم إجراؤها بالخوارزمية لحل مسألة معينة كدالة في حجم المدخلات.

(iii) درجة التعقيد في أحسن حالة (Best_case Complexity)

هي أقل عدد من العمليات التي يتم إجراؤها بالخوارزمية لحل مسألة معينة كدالة في حجم المدخلات.

وعموما فإن درجة التعقيد المتوسطة هي أفضل مقياس / معيار ، ولكنا عادة نلجأ إلى استخدام درجة التعقيد في أسوأ حالة ، وهي عموما أسهل حسابيا (من الناحية الرياضية) ، وأحيانا أيضا نستخدم درجة التعقيد المتوسطة.

أمثلة على إيجاد درجة التعقيد

في كل من المسائل التالية اكتب خوارزمية / قطعة برنامج لحل المسألة ، واحسب درجات التعقيد في أحسن حالة ، وفي أسوأ حالة ، ودرجة التعقيد المتوسطة.

مثال ١-٥ : البحث التتابعي عن عنصر key في منظومة a مكونة من n عنصر (sequential search of an array) ، وتعيد الخوارزمية ترتيب العنصر (i) إن وجدته ، أو تعيد صفرا إن لم يوجد.

الحل :

لاحظ أن العملية الأساسية هنا هي المقارنة (comparison) ، حيث نقارن العنصر key الذي نبحث عنه مع عناصر المنظومة a عنصرا عنصرا ابتداء من العنصر a_0 إلى أن نجده أو إلى أن نصل إلى نهاية المنظومة

$$\text{key} \quad \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_i \\ \vdots \\ a_{n-1} \end{pmatrix}$$

الخوارزمية (Algorithm)

```
found = false ;
i = 0 ;
while ((i < n) && (! found))
        if (key = = a[i] )
        found = true
    else
        i = i + 1 ;
if (found)
        index = i
else
        index = 0 ;
```

وباختيار عملية المقارنة (فقط) لتمثيل العمليات التي تجريها الخوارزمية فإن الجدول التالي يلخص الاحتمالات المختلفة التي نصل إلى أحدها عند انتهاء الخوارزمية (العنصر الذي نبحث عنه هو أول عنصر ، أو ثاني عنصر ، ... أو العنصر رقم i ، ... ، أو آخر عنصر ، أو غير موجود).

الحالات / الاحتمالات المختلفة	عدد العمليات (المقارنات)
case	no. of operations
$key = a_0$	1
$key = a_1$	2
⋮	⋮
$key = a_i$	i+1
⋮	⋮
$key = a_{n-1}$	n
key غير موجود	n

ومن هذا الجدول يمكننا أن نصل إلى النتائج التالية :

درجة العقيد في أحسن حالة $(key = a_0) = 1$

درجة التعقيد في أسوأ حالة $(key = a_{n-1}$ أو key غير موجود$) = n$

درجة التعقيد المتوسطة $= \frac{1+2+...+...+n+n}{n+1}$

$$= \frac{\left(\sum_{i=1}^{n} i\right)+n}{n+1} = \frac{\frac{n(n+1)}{2}+n}{n+1} = \frac{n}{2} + \frac{n}{n+1}$$

$$\cong \frac{n}{2} \qquad (n >> 1,\ n \to \infty)$$

ملاحظة ١ : افترضنا هنا عند حساب درجة التعقيد المتوسطة أن جميع الحالات متساوية الاحتمالات (uniform probability distribution).

ملاحظة ٢ : يمكننا القول بأن درجة التعقيد المكانية تساوي space complexity ≈ n+1 حيث نحتاج إلى n موضع لعناصر المنظومة (وعددها n عنصرا) وموضع واحد للعنصر key.

مثال ١-٦ : الضرب الداخلي / النقطي / القياسي لمنظومتين / لمتجهين بكل منهما n عنصر

inner / dot / scalar product of two arrays / vectors

الحل : نفرض أن المنظومتين هما b ، a .

$$\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_i \\ \vdots \\ a_{n-1} \\ a \end{pmatrix} \quad \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_i \\ \vdots \\ b_{n-1} \\ b \end{pmatrix}$$

الضرب الداخلي لهاتين المنظومتين تعبر عنه العلاقة التالية :

$$s = a \,.\, b$$

$$= \sum_{i=0}^{n-1} a_i . b_i = a_0 b_0 + a_1 b_1 + \ldots + a_{n-1} b_{n-1} .$$

الخوارزمية :

```
S = 0 ;
for (i = 0; i < n ; i ++)
S = S + a_i * b_i ;
// end for i
```

ولإيجاد العدد الكلي للعمليات التي يتم إجراؤها حتى انتهاء تنفيذ الخوارزمية نجد أن :

العملية	عدد مرات إجرائها
جمع +	n
ضرب *	n
إسناد =	n+1
العدد الإجمالي للعمليات	3n+1

ولكن عادة نختار عملية واحدة مميزة للخوارزمية (characteristic operation) ولتكن مثلا عملية الضرب * (أو عملية الجمع +) ، ونقول إن :

درجة تعقيد الخوارزمية = n

ملاحظات

(١) درجة التعقيد في أحسن حالة = درجة التعقيد في أسوأ حالة = درجة التعقيد المتوسطة = n

(٢) حين تكون درجة التعقيد دالة خطية (linear function) في n فإننا نطلق على الخوارزمية «خوارزمية خطية» (linear algorithm).

(٣) حيث أننا نحتاج إلى n موضع لتخزين كل من عناصر المنظومة a وعناصر المنظومة b ، وموضع واحد لتخزين ناتج الضرب الداخلي S فيمكننا استنتاج أن درجة التعقيد المكانية s.c. ≈ 2n+1

مثال ١-٧ : جمع مصفوفتين من الرتبة n x n

Adding two matrices of order n x n

نفرض أن لدينا مصفوفتين A ، B وأن كلا منهما من الرتبة n x n والمطلوب جمع المصفوفتين للحصول على مصفوفة C ، أي أن

$$C = A + B$$

$$[c_{ij}]_{nxn} = [a_{ij}]_{nxn} + [b_{ij}]_{nxn}$$

الحل :

الخوارزمية :

```
for (i = 0 ; i < n ; i ++)
      for (j = 0 ; j < n ; j ++)
            cij = aij + bij
      // end for j
// end for i
```

العمليات التي تجريها هذه الخوارزمية هي إعطاء قيم ترتيبية (indexing) وإسناد (=) ، وجمع (+). نختار من هذه العمليات عملية مميزة للخوارزمية ، أي عملية تظهر في أكثر العرى داخلية (innermost loop) ، كعملية الجمع. وبالتالي فإن درجة التعقيد (عدد عمليات الجمع) تساوي n^2

ونظرا لأن كل مصفوفة من المصفوفات A , B , C تحتاج إلى n^2 موضع لتخزين عناصرها ، فإن درجة التعقيد المكانية تساوي $s.c. \approx 3n^2$

مثال ١-٨ : **ضرب مصفوفتين مربعتين n x n**

Matrix multiplication of 2 square matrices

نفرض أن المطلوب الحصول على المصفوفة C الناتجة من ضرب المصفوفتين المربعتين A , B حيث كل منهما مصفوفة من الرتبة n x n.

الحل :

$$C = A \cdot B$$

$$C_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj} \qquad \begin{matrix} i = 0, 1, ..., n-1; \\ j=0, 1, ..., n-1. \end{matrix}$$

الخوارزمية :

```
for (i = 0; i < n; i ++)
    for (j = 0; j < n; j ++)
        cij = 0;
        for (k = 0; k < n; k ++)
            cij = cij + aik * bkj
        // end for k
    // end for j
// end for i
```

العمليات التي تجريها هذه الخوارزمية هي : إعطاء القيم الترتيبية ، والإسناد ، والضرب والجمع. نختار عملية الضرب* أو عملية الجمع + كعملية مميزة لهذه الخوارزمية ، وحيث أن عدد مرات إجراء أي من هاتين العمليتين هو n^3 ، لذا فإننا نستنتج أن :

درجة تعقيد الخوارزمية = n^3

وأما عن السعة المكانية التي تتطلبها هذه الخوارزمية فهي n^2 لكل مصفوفة من المصفوفات الثلاث A , B , C وبالتالي فإن حيز التخزين المستخدم (storage used) أو درجة التعقيد المكانية تساوي

$$s.\ c. \approx n^2 + n^2 + n^2 = 3n^2$$

لاحظنا في حل مثال ١-٥ أننا احتجنا عند حساب درجة التعقيد المتوسطة إلى معرفة مجموع المتسلسلة $\sum_{i=1}^{n} i$ للوصول إلى نتيجة مبسطة لدرجة التعقيد ، وفيما يلي نكتب نتائج مجموع بعض المتسلسلات ، ويمكن إثبات أي منها بواسطة الاستنتاج الرياضي (math. induction).

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad ; \qquad n \geq 1$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \quad ; \qquad n \geq 1$$

$$\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1} \quad ; \qquad n \geq 0, r \neq 1$$

$$\sum_{i=1}^{n} 1 = n \quad ; \qquad n \geq 1$$

$$\sum_{i=1}^{n} c = cn \quad ; \qquad n \geq 1$$

$$\sum_{i=m}^{i=n} f(i) = \sum_{i=1}^{i=n} f(i) - \sum_{i=1}^{i=m-1} f(i) \quad ; \qquad m < n$$

اصطلاحات حدود القيم وترتيبها / اصطلاحات التقارب

Order of Magnitude Notation / Asymptotic Notation

O , Ω , Θ

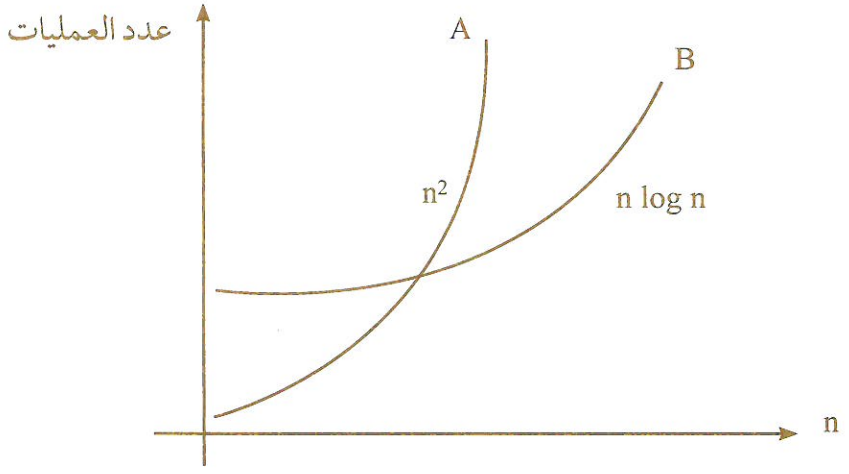
تمهيد :

نفرض أن لدينا خوارزميتين A , B لحل مسألة معينة حجمها n ، ونفرض أن T_A , T_B هما – على الترتيب – درجة التعقيد (أي عدد العمليات) لكل من الخوارزميتين A , B ، ونفرض أن

$$T_A = \frac{1}{2}\ n^2 + n = c_1 n^2 + c_2\ n$$

$$T_B = 10\ n \log_2 n + 100\ n = c_3\ n \log n + c_4\ n$$

من الممكن أن نثبت إما بالرسم (graphically) أو بجدول قيم (table of values) أنه لقيم n الكبيرة (كبرا كافيا large enough) تكون الدالة T_B أصغر من الدالة T_A بغض النظر عن قيم الثوابت c_4 ، c_3 ، c_2 ، c_1 ، أي أن الخوارزمية B تكون أكبر كفاءة (more efficient) من الخوارزمية A.



فالمطلوب أن نستخدم اصطلاحا (notation) معينا يعيننا على استنتاج هذه الملاحظة ، أي اصطلاحا يفيدنا في معرفة حدود قيم الدالة (order of the function) ، فحدود قيم الدالة أو الصورة العامة للدالة (form of the function) أهم من ثوابتها.

ملاحظة : يمكننا الوصول إلى الملاحظة المذكورة سابقا أنه لقيم n الكبيرة تكون الدالة T_B أصغر من الدالة T_A باستخدام العلاقة التالية :

$$\lim_{n \to \infty} \frac{T_B}{T_A} = \lim_{n \to \infty} \frac{10n\log n + 100n}{\frac{1}{2}n^2 + n} = \cdots = 0$$

تعريف الاصطلاح O (big oh)

نفرض أن كلا من f , g دالة في العدد الصحيح n. نقول إن

$f(n) = O(g(n))$ ($f(n)$ is big oh of g of n)

إذا وفقط إذا وُجد ثابت موجب $c > 0$ وعدد صحيح موجب $n_o > 0$ بحيث أن :

$$f(n) \leq c\, g(n) \quad \forall n \geq no$$

أي أن :

$$f(n) = O(g(n)) \text{ iff } \exists\, c\,,\ n_o > 0 \ni$$

$$f(n) \leq c\, g(n) \quad \forall n \geq n_o$$

تعريف مكافئ (equivalent definition)

$f(n) = O(g(n))$ iff

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty \qquad (< \infty)$$

أي أن :

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \qquad (c = \text{const} < \infty)$$

(أي أن c عبارة عن ثابت يساوي إما صفرا أو عددا محدودا

$(c = 0 \text{ or } c = \text{finite}, \ c \neq \infty$

معنى هذا التعريف

التعريف المذكور للاصطلاح O (أو التعريف المكافئ) يعنى أيا من العبارات الثلاث التالية :

* الدالة f لها نفس حدود الدالة g أو أقل

(f has the same order or less than g)

* الدالة f لا تزداد بمعدل أسرع من g

(f grows no faster than g)

* الدالة g حد علوي للدالة f

(g(n) is an upper bound on f(n))

مثال ١-٩ : اثبت صحة ما يلي :

i) $3n + 2 = O(n)$

ii) $3n + 3 = O(n)$

iii) $3n^2 + 2 = O(n^2)$

iv) $3n + 2 = O(n^2)$

v) $3n^2 + 2 \neq O(n)$

vi) $3n^2 + 2n = O(n^2)$

vii) $10^6\, n^2 = O(n^2)$

viii) $10\, n \log n + 100\, n = O(n^2)$

ix) $10\, n \log n + 100\, n = O(n \log n)$

x) $\log \log n = O(\log n)$

xi) $2^n \neq O(n^{1000})$

الحل :

i) $\lim_{n\to\infty} \frac{3n+2}{n} = 3 \neq \infty$

حل آخر

$3n + 2 \leq 4n \qquad \forall\, n \geq 2$

$(c = 4 \text{ ، } n_0 = 2)$

ii) $\lim_{n\to\infty} \frac{3n+3}{n} = 3 \neq \infty$

حل آخر

$3n + 3 \leq 4n \qquad \forall\, n \geq 3$

iii) $\lim_{n\to\infty} \frac{3n^2+2}{n^2} = 3 \neq \infty$

iv) $\lim_{n\to\infty} \frac{3n+2}{n^2} = 0 \neq \infty \qquad (< \infty)$

ملاحظة : العبارة $3n + 2 = O(n^2)$ المطلوب إثباتها (والتي أثبتناها) صحيحة ولكن عادة لا نذكرها لأنها غير مفيدة ولا تعطينا معلومات (not informative) عن الدالة $f(n) = 3n + 2$ أي لا معنى لها (meaningless) ، وذلك لأنه إذا كانت الدالة $g(n) = n^2$ حدا علويا للدالة $f(n)$ (انظر (iii)) فمن البديهي أن تكون $g(n) = n$ حدا علويا أيضا للدالة $f(n)$ لأن $n^2 > n$ ، ولذلك نكتفي بذكر العلاقة $3n + 2 = O(n)$

v) $\lim_{n\to\infty} \frac{3n^2+2}{n} = \infty$

vi) $\lim_{n\to\infty} \frac{3n^2+2n}{n^2} = 3 \neq \infty$

vii) $\lim_{n\to\infty} \frac{10^6 n^2}{n^2} = 10^6 \neq \infty$

viii) $\lim_{n\to\infty} \frac{10\, n \log n + 100\, n}{n^2} = 0 \neq \infty$

ix) $\lim_{n\to\infty} \frac{10\, n \log n + 100\, n}{n \log n} = 10 \neq \infty$

x) $\lim_{n\to\infty} \frac{\log \log n}{\log n} = 0 \neq \infty$

xi) $\lim_{n\to\infty} \frac{2^n}{n^{1000}} = \infty$

تعريف الاصطلاح Ω (big omega)

نفرض أن كلا من f , g دالة في العدد الصحيح n . نقول إن

$$f(n) = \Omega\,(g(n))$$

إذا وفقط إذا وُجد ثابت موجب $c > 0$ وعدد صحيح موجب $n_o > 0$ بحيث أن :

$$f(n) \geq c\, g(n) \qquad \forall n \geq n_o$$

أي أن :

$f(n) = \Omega(g(n))$ iff $\exists\ c\ ،\ n_0 > 0\ \ni$

$f(n) \geq c\ g(n) \quad \forall n \geq n_0$

تعريف مكافئ

$f(n) = \Omega(g(n))$ iff

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq 0$$

معنى التعريف : التعريف المذكور للاصطلاح Ω (التعريف الأصلي أو التعريف المكافئ) يعني أيا من العبارات التالية :

* الدالة f لها نفس حدود الدالة g أو أكبر.
* الدالة f لا تزداد بمعدل أبطأ من g.
(f grows no slower than g)
* الدالة g حد سفلي للدالة f .
(g(n) is a lower bound on f(n))

تعريف الاصطلاح Θ (big theta)

نفرض أن كلا من f , g دالة في العدد الصحيح n. يقال إن

$f(n) = \Theta(g(n))$

إذا وفقط إذا وُجد ثابتان موجبان $c_2 > 0$ ، c_1 وعدد صحيح موجب $n_0 > 0$ بحيث أن

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \quad \forall n \ge n_o$$

أي أن :

$$f(n) = \Theta\,(g(n)) \text{ iff } \exists\, c_1, c_2\,,\ n_o > 0 \ni$$

$$c_1\, g(n) \le f(n) \le c_2\, g(n) \quad \forall n \ge n_o$$

تعريف مكافئ :

$$f(n) = \Theta\,(g(n)) \text{ iff}$$

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c \qquad (c \neq 0,\ c \neq \infty)$$

1

معنى تعريف الاصطلاح Θ :

التعريف المذكور يعني أيا من العبارات التالية :

* الدالة f لها حدود الدالة g نفسها.
* الدالة f تزداد بنفس معدل زيادة الدالة g.
* الدالة g حد علوي وحد سفلي للدالة f.

تعريف الاصطلاح o (small oh)

نفرض أن كلا من f , g دالة في العدد الصحيح n. يقال إن

$$f(n) = o\,(g\,(n))$$

إذا وفقط إذا تحقق الشرط

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

أي أن $g(n) >> f(n)$

معنى تعريف الاصطلاح o

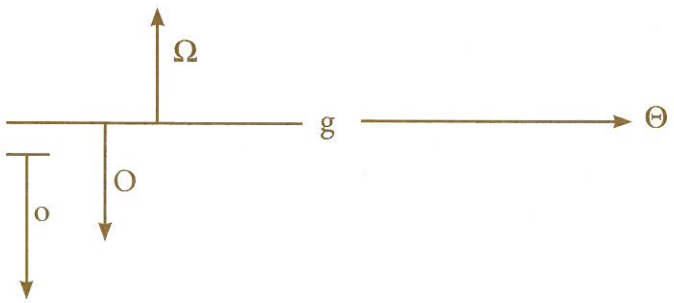
الدالة f تزداد بمعدل أبطأ من معدل زيادة الدالة g

(الدالة g تزداد بمعدل أسرع من f)

يمكننا تلخيص تعريفات الاصطلاحات المذكورة سابقا ومعانيها في الجدول التالي :

الاصطلاح	$\lim_{n\to\infty} \frac{f(n)}{g(n)}$	معنى الاصطلاح	
$f(n) = o(g(n))$	0	تزايد f أبطأ من تزايد g	<
$f(n) = O(g(n))$	$\neq \infty$	تزايد f ليس أسرع من تزايد g	≤
$f(n) = \Theta(g(n)$	$\neq 0 , \neq \infty$	تزايد f مثل تزايد g	=
$f(n) = \Omega(g(n)$	$\neq 0$	تزايد f ليس أبطأ من تزايد g	≥

ويمكننا أن نوضح بيانيا تزايد الدالة f بالنسبة لتزايد الدالة g في الاصطلاحات المختلفة بالشكل التالي :



توضيح بياني لتزايد f بالنسبة لتزايد g

مثال ١-١٠ :

اثبت صحة العلاقات التالية :

i) $(\log_2 n)^2 = o(n)$

ii) $n = \Omega((\log_2 n))^2$

iii) $2^n \neq O(n^k)$

حيث k عدد صحيح موجب ، مهما كانت قيمة k كبيرة.

iv) $n^k = o(2^n)$

الحل :

$$\text{i)} \lim_{n\to\infty} \frac{(\log_2 n)^2}{n} = 2\log_2 e \lim_{n\to\infty} \frac{\log_2 n * \frac{1}{n}}{1}$$

$$= 2(\log_2 e)^2 \lim_{n\to\infty} \frac{\frac{1}{n}}{1} = 0$$

ويمكننا استنتاج العلاقة (ii) من برهان العلاقة (i)

$$\text{iii)} \lim_{n\to\infty} \frac{2^n}{n^k} = \lim_{n\to\infty} \frac{2^n . \ln 2}{k.n^{k-1}} =$$

$$= \frac{(1n_2)^2}{k(k-1)} \lim_{n\to\infty} \frac{2^n}{n^{k-2}} = \cdots = \frac{(1n2)^k}{k1} \lim_{n\to\infty} \frac{2^n}{1} = \infty$$

ويمكن استنتاج العلاقة (iv) من برهان العلاقة (iii).

* * *

درجات التعقيد العملية Practical Complexities

تستخدم دالة درجة التعقيد (Complexity function) للمقارنة بين برنامجين P ، Q يؤديان الوظيفة نفسها. فمثلا إذا فرضنا أن درجة تعقيد البرنامج P هي $\Theta(n)$ بينما درجة تعقيد البرنامج Q هي $\Theta(n^2)$ فيمكننا الحكم بأن تنفيذ البرنامج P أسرع من تنفيذ البرنامج Q لقيم n الكبيرة كبرا كافيا (sufficiently large n). ولبيان صحة ذلك لاحظ أن وقت الحساب الفعلي (actual computing time) للبرنامج P محدود من أعلى (bounded from above) بالمقدار cn حيث c ثابت ما وذلك لجميع قيم n ، حيث $n \geq n_1$ ، بينما وقت الحساب الفعلي للبرنامج Q محدود من أسفل بالمقدار dn^2 حيث d ثابت ما ، وذلك لجميع قيم n حيث $n \geq n_2$ ، وحيث أن $cn \leq dn^2$ لقيم n التي تحقق العلاقة $n \geq c / d$ ، فنستنتج أن البرنامج P أسرع من البرنامج Q كلما كان $c \geq \max\{n1, n2, c/d\}$.

وفيما يلي بعض الدوال الشائعة وهي مكتوبة بترتيب تصاعدي لحدود القيم (increasing order of magnitude).

1 (const) قيمة ثابتة

$\log \log n$

$\log n$

$\sqrt{n}$

$n / \log n$

n (linear fn.) دالة خطية

$n \log n$

$n\sqrt{n}$

$n^2 / \log n$

n^2 (quadratic fn.) دالة تربيعية

n^3 (cubic fn.)دالة تكعيبية

$$\left.\begin{array}{l}\vdots\\ 2^n\\ 3^n\\ \vdots\end{array}\right\}\ \text{دوال أسية}\ \text{(exponential fns.)}$$

$n!$

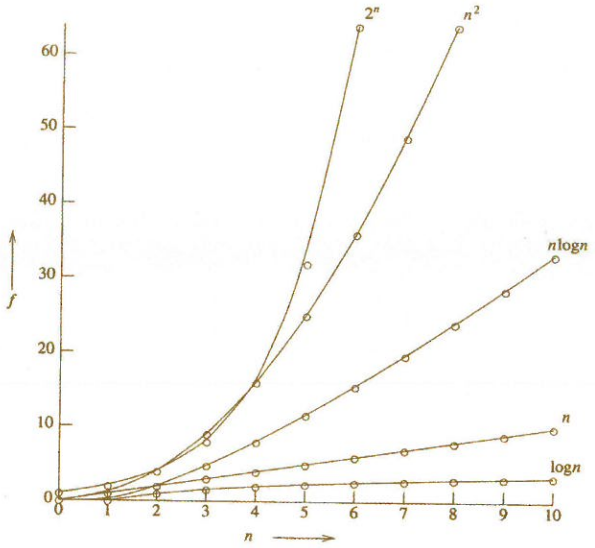
n^n

ونلاحظ عند المقارنة بين برنامجين - مثل البرنامجين P , Q المذكورين سابقا - أننا نقارن بينهما لقيم n الكبيرة كبرا كافيا. ولذا فيجب التأكد من أن n كبيرة فعلا كبرا كافيا ، وإلا كانت نتيجة المقارنة غير صحيحة. فمثلا إذا كان تشغيل البرنامج P يستغرق 10^6n ميللي ثانية (10^6n milliseconds) ، بينما تشغيل البرنامج Q يستغرق n^2 ميللي ثانية ، وكانت $n \leq 10^6$ دائما ، فإننا نفضل استخدام البرنامج Q وليس P (بفرض تساوي جميع العوامل الأخرى).

ويمكننا إدراك مدى تزايد قيم الدوال المختلفة بزيادة قيمة n بإلقاء نظرة فاحصة على كل من الجدول التالي والشكل التالي. ونلاحظ منهما أن الدالة الأسية 2^n تزداد بسرعة كبيرة جدا بزيادة n . فمثلا إذا كان عدد الخطوات المطلوب إجراؤها لتنفيذ برنامج ما يساوي 2^n فعندما تكون n = 40 فإن عدد الخطوات يساوي تقريبا $١٢١٠ \times ١,١$ ، فإذا كان لدينا حاسوب يمكنه تنفيذ بليون خطوة في الثانية الواحدة فإن تنفيذ هذا البرنامج يستغرق نحو ١٨,٣ دقيقة ، وعندما تكون n = 50 فإن تنفيذ البرنامج نفسه يستغرق حوالي ١٣ يوما على هذا الحاسوب ، وعندما n = 60 فعلينا أن ننتظر ٣١٠,٥٦ سنة !! لإتمام تنفيذ البرنامج ، وحينما تصل قيمة n إلى n = 100 فنحتاج إلى ٤×١٣١٠ سنة !!!. وبالتالي يمكننا أن نستنتج أن استخدام البرامج ذات درجة التعقيد الأسية يكون محدودا - من الناحية العملية - لقيم n الصغيرة فقط (تقريبا $n \leq 40$).

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

جدول قيم الدوال



منحنيات قيم الدوال

فالخوارزميات المقبولة (accepted algorithms) عموما هي الخوارزميات التي لا تصل دوالها إلى حد الدوال الأسية ، إلا أن البرامج التي تكون درجة تعقيدها حدودية من درجة عالية (polynomial of high degree) تكون أيضا محدودة الاستعمال ، فمثلا إذا احتاج برنامج إلى n^{10} خطوة فهذا يعني أننا باستخدام الحاسوب ذي البليون خطوة / ثانية نحتاج إلى 10 ثواني عندما$n=10$

و ٣١٧١ سنة !! عندما n = 100 و ٣,١٧ × ١٣١٠ سنة !!! عندما n = 1000. بينما إذا كانت درجة تعقيد البرنامج n^3 خطوة فإننا نحتاج إلى ثانية واحدة فقط عندما n = 1000 ، وإلى ١١٠,٦٧ دقيقة عندما n = 10,000 ، وإلى 11. 57 يوما عندما n = 100,000 .

والجدول الزمني التالي يبين الزمن الذي نحتاجه لتنفيذ برنامج درجة تعقيده f(n) خطوة من التعليمات (instructions) باستخدام الحاسوب ذي البليون خطوة في الثانية الواحدة. ومن الناحية العملية نلاحظ أنه لقيم n الكبيرة كبرا معقولا في التطبيقات العملية (مثلا n > 100) يمكننا فقط تشغيل البرامج ذوات درجات التعقيد المنخفضة (مثل : n^3 , n^2 , n log n , n).

Time for f(n) instructions on a 10^9 instr/sec computer							
n	f(n)=n	$f(n)=\log_2 n$	$f(n)=n^2$	$f(n)=n^3$	$f(n)=n^4$	$f(n)=n^{10}$	$f(n)=2^n$
10	.01μs	.03μs	.1μs	1μs	10μs	10sec	1μs
20	.02μs	.09μs	.4μs	8μs	160μs	2.84hr	1ms
30	.03μs	.15μs	.9μs	27μs	810μs	6.83d	1sec
40	.04μs	.21μs	1.6μs	64μs	2.56ms	121.36d	18.3min
50	.05μs	.28μs	2.5μs	125μs	6.25ms	3.1yr	13d
100	.10μs	.66μs	10μs	1ms	100ms	3171yr	$4*10^{13}$yr
1,000	1.00μs	9.96μs	1ms	1sec	16.67min	$3.17*10^{13}$yr	$32*10^{283}$yr
10,000	10.00μs	130.03μs	100ms	16.67min	115.7d	$3.17*10^{23}$yr	
100,000	100.00μs	1.66ms	10sec	11.57d	3171yr	$3.17*10^{33}$yr	
1,000,000	1.00ms	19.92ms	16.67min	31.71yr	$3.17*10^{7}$yr	$3.17*10^{43}$yr	

مايكرو ثانية	: μs = microsecond = 10^{-6} second
ميللي ثانية	: ms = millisecond = 10^{-3} seconds
ثانية	: sec = seconds
دقيقة	: min = minutes
ساعة	: hr = hours
يوما	: d = days
سنة	: yr = years

الجدول الزمني لتنفيذ f(n) خطوة من التعليمات باستخدام حاسوب ينفذ بليون خطوة في الثانية.

1 Data Design and Implementation

تمرينات رقم ١

١-١ اثبت صحة المتطابقات التالية :

(i) $5n^2 - 6n = \Theta(n^2)$

(ii) $n! = O(n^n)$

(iii) $2n^2 2^n + n \log n = \Theta(n^2 2^n)$

(iv) $4(\log n)^3 \ / \ n = o(n)$

(v) $2^{2^n} + 6 * 2^n = \Theta\left(2^{2^n}\right)$

(vi) $6\, n^3 \ / (\log n + 1) = O(n^3)$

(vii) $n^{k+\varepsilon} + n^k \log n = \Theta(n^{k+\varepsilon}) \qquad \forall\, k \geq 0\,,\ \varepsilon > 0$

(viii) $33\, n^3 + 4\, n^2 = \Omega(n^3)$

١-٢ اثبت عدم صحة المتطابقات التالية :

(i) $10\, n^2 + 9 = o(n)$

(ii) $n^2 \log n = \Theta(n^2)$

(iii) $n^2 \ / \ \log n = \Theta(n^2)$

(iv) $n^3\, 2^n + 6\, n^2\, 3^n = O(n^3\ 2^n)$

١-٣ اثبت صحة العلاقة التالية لأي عدد صحيح ثابت k :

$(\log n)^k = o(n)$

١-٤ أ) ما معنى الاصطلاح $f(n) = o(g(n))$ ؟

ب) اثبت أن $10000 \ln n = o(n / \ln n)$

ج) رتب الدوال التالية ترتيبا تصاعديا حسب قيمها مع بيان السبب :

$n \ln n, \ \ln n, \ n / \ln n$

١-٥ نفرض أن :

$T_1(n) = O(f(n)), \ T_2(n) = O(f(n)),$

اذكر صحة أو خطأ كل من العلاقات التالية ، مع ذكر السبب (أو إعطاء مثال مناقض (counter example)) في حالة العبارة الخاطئة :

(i) $T_1(n) + T_2(n) = O(f(n))$

(ii) $T_1(n) - T_2(n) = o(f(n))$

(iii) $\frac{T_1(n)}{T_2(n)} = O(1)$

(iv) $T_1(n) = O(T_2(n))$

١-٦ أوجد درجة تعقيد (complexity) الخوارزمية التالية :

```
cin >> P;                    // P = price of one gram of gold
Nisab = 85*P ;
for (i = 1; i <= n ; i + +)      // n = number of persons
```

```
{
    cin >> Savings;
    if (Savings >= Nisab)
        Zakat = 0. 025 * Savings;
    else
        Zakat = 0. 0;
    cout << Savings << Zakat;
}
```

١-٧ كم تساوي درجة التعقيد في أسوأ حالة للخوارزمية التالية :

```
for (i = 1;  i <= n;  i + +)
    for (j = 1;  j <= i;  j + +)
        k = i * j ;
```

١-٨ أوجد حدود قيمة C(n) دالة التعقيد (order of magnitude of the complexity function) التي تقيس (measures) زمن تشغيل / تنفيذ (running execution time) كل من الخوارزميات التالية ، حيث n هي حجم البيانات المدخلة (size of the input data). يمكن إعطاء النتيجة بدلالة الاصطلاح O.

```
i) sum = 0;
   for (i = 1;  i <= n;  i + +)
        sum = sum + 1;
ii) sum = 0 ;
   for (i = 1;  i <= n;  i + +)
        for (j = 1;  j <= n;  j + +)
            sum = sum + 1;
iii) sum = 0 ;
    for (i = 1;  i <= n;  i + +)
```

```
    for (j = 1;  j <= n*n;  j + +)
        sum = sum + 1;
iv) sum = 0;
    i = 1;
    while (i <= n)
      {
        sum = sum + 1;
        i = 2*i;
      }
```

١-٩ نفرض أنه بعد تحليل (analyzing) برنامجين A. B وُجد أن زمن التشغيل في أسوأ حالة (worst-case running time) لا يزيد عن $150\ n \log_2 n$ بالنسبة للبرنامج A ، ولا يزيد عن n^2 بالنسبة للبرنامج B. أجب إن أمكن عن الأسئلة التالية :

أ) أي البرنامجين أفضل من حيث ضمان زمن التشغيل (has the better guarantee on the running time) بالنسبة لقيم n الكبيرة ($n > 10,000$)؟ ولماذا؟

ب) أي البرنامجين أفضل من حيث ضمان زمن التشغيل بالنسبة لقيم n الصغيرة ($n < 100$)؟ ولماذا؟

ج) أي البرنامجين سيكون أسرع في التنفيذ (runs faster) لقيم n المتوسطة ، مثل $n = 1000$ ؟ ولماذا؟

د) هل من الممكن أن يكون البرنامج B أسرع تنفيذا من البرنامج A بالنسبة لجميع المدخلات المحتملة (all possible inputs) ؟ ولماذا؟

١٠-١ احسب درجة التعقيد الزمنية [حلل زمن الحسابات (Analyse the computing time) للدالة **sort** التالية التي تطبق طريقة «الترتيب بالاختيار» (selection sort) لترتيب عناصر منظومة أعداد صحيحة a عدد عناصرها n ترتيبا غير تنازلي.

```
for (i = 0;  i < n;  i + +)
{
      examine a[i] to a [n] and suppose the smallest integer is at a[j];
      interchange a[i] and a[j];
}
```

خوارزمية الترتيب بالاختيار (Selection sort)

```
void sort (int a[ ],  int n)
// sort the n integers a[0..  n-1] into nondecreasing order
{ int i,  j,  k,  t;
      for (i = 0;  i < n;  i + +)
      {
            j = i;
            // find smallest integer in a[j..  n-1]
            for (k = j + 1;  k < n;  k + +)
                  if (a [k] < a [j])
                        j = k;
            // interchange
            t = a[i];
            a[i] = a[j];
            a[j] = t;
      } // end for i
} // end of sort
```

1 Data Design and Implementation

الفصل الثاني

المنظومات
Arrays

2

Arrays 2

الفصل الثاني

المنظومـــات
Arrays

تعد المنظومات من أهم هياكل البيانات المتوفرة في لغات البرمجة. ونبدأ بإذن الله تعالى بدراسة المنظومات أحادية البعد.

المنظومات أحادية البعد One-Dimensional Arrays

المنظومة أحادية البعد هي متتابعة / متسلسلة (sequence) من عناصر / أشياء (objects) من النوع نفسه (same type). فالشكل التالي مثلا يرمز إلى منظومة أحادية البعد a مكونة من عدد n من العناصر a_{n-1} , . . . , a_0, a_1, a_2.

ونُعرِّف على هذه العناصر عددا من العمليات تشمل :

العمليات (operations) :

١- إنشاء (create) منظومة خالية.

٢- تخزين (store) قيمة ما x في موضع معين a[i] في المنظومة :

$$a[i] \leftarrow x$$

٣- استرجاع (retrieve) القيمة المخزونة في موضع معين a[i] في المنظومة :

$$x \leftarrow a[i]$$

والرمز i الذي يشير إلى موضع معين في المنظومة يسمى الدليل / المؤشر (index / subscript) ، وهو عادة عدد صحيح (integer) ومن الممكن أيضا أن يكون من أنواع أخرى مثل المدى الجزئي (subrange) ، ويجب أن يكون في مدى طول المنظومة (array length) أي عدد عناصرها.

أهمية المنظومات :

- المنظومات هي بنية معطيات في جميع اللغات عالية المستوى HLL(High Level Languages) .
- وهي بنية المعطيات الوحيدة في بعض اللغات كالفورتران والبيسك.
- والذاكرة الرئيسية (main memory) في الحاسوب هي عبارة عن منظومة أحادية البعد كبيرة الحجم.
- ومن أهم مميزات المنظومات التي تكسبها أهمية خاصة من الناحية العملية الكفاءة العالية للوصول المباشر / العشوائي (efficient random / direct access) لأي عنصر من عناصر المنظومة ، أي أننا نستغرق الوقت نفسه للوصول إلى أي عنصر a[i] بغض النظر عن قيمة المؤشر i ، وذلك بالإضافة إلى الوصول التتابعي (sequential access) .

وعادة يتم تخزين المنظومة في خلايا متلاصقة من الذاكرة (contiguous memory cells) .

القائمة الخطية / المرتبة Ordered / Linear List

تعد القائمة الخطية / المرتبة من أبسط الصور الشائعة للبيانات (data objects) ، والقائمة الخطية إما أن تكون خالية ونرمز لها هكذا : () ، أو تحتوي على متتابعة محدودة (finite sequence) من العناصر ، ونكتبها هكذا :

$$(a_0, a_1, \ldots, a_i, \ldots, a_{n-1})$$

حيث a_i هي عناصر / ذرات (elements / atoms) مجموعة ما S (set) ، فمثلا قائمة الخلفاء هي :

Caliphs = (AbuBakr , Omar , Othman , Ali)

وقائمة الصلوات هي :

Prayers = (Fajr, Zuhr, Asr, Maghreb, Isha)

وهكذا ..

وهناك العديد من العمليات التي يمكن أن تجري على هذه القوائم.

العمليات : فيما يلي قائمة جزئية من العمليات الممكن تنفيذها على القوائم :

١ . إيجاد طول (length) القائمة (أي إيجاد عدد عناصرها).

٢ . اجتياز (traverse / scan) القائمة (مثلا لطباعة عناصرها أو بعض عناصرها).

٣ . استرجاع (retrieve / get) العنصر رقم i في القائمة ، $0 \le i \le n-1$ حيث n يمثل عدد عناصر القائمة.

٤ . تعديل/ تغيير (update / modify) العنصر رقم i في القائمة ، $0 \le i \le n-1$.

٥ . إدخال (insert) عنصر جديد x مثلا في الموضع رقم i ،

فالقائمة $(a_0, a_1, ..., a_{i-1}, a_i, a_{i+1}, ..., a_{n-1})$.

تصبح $(a_0, a_1, ..., a_{i-1}, x, a_i, a_{i+1}, ..., a_{n-1})$.

٦ . حذف (delete) العنصر رقم i من القائمة.

ونلاحظ أن العمليتين ٥ ، ٦ تؤثران على طول القائمة :

ففي العملية ٥ : العناصر $a_i, a_{i+1}, ..., a_{n-1}$ تزاح منطقيا (logically) لليمين خطوة واحدة ،

وفي العملية ٦ : العناصر $a_{i+1}, a_{i+2}, ..., a_{n-1}$ تزاح منطقيا لليسار خطوة واحدة.

التنفيذ (implementation) / التمثيل (representation)

يمكن تنفيذ القائمة الخطية عمليا بطريقة مبسطة عن طريق استخدام منظومة خطية (أي منظومة أحادية البعد) ، وتخزين العنصر رقم i من القائمة في العنصر رقم i من المنظومة. ومن الواضح أن هذا التنفيذ أي هذه البنية للمعطيات (data structure) تجعل العمليات ١) ← ٤) المذكورة سابقا سهلة التنفيذ ومباشرة وتتم بكفاءة عالية ، أما العمليتان ٥) ، ٦) فكفاءة أي منهما لا تكون عالية ، فمثلا العملية ٥) تتطلب (لإدخال العنصر x في الموضع رقم i) إزاحة فيزيائية / عملية / مادية (physical shift) لجميع العناصر $a_i, a_{i+1}, ..., a_{n-1}$ خطوة واحدة لليمين ، بينما تتطلب العملية ٦) (لحذف العنصر رقم i) إزاحة مادية للعناصر $a_{i+1}, a_{i+2}, ..., a_{n-1}$ خطوة واحدة لليسار.

القائمة المرتبة (ordered list) هي قائمة خطية رُتبت عناصرها بناء على قيمة مجال معين يسمى المجال المفتاح (key field) .

وكمثال عملي على القوائم الخطية / المرتبة ندرس فيما يلي الحدوديات.

الحدوديات كقوائم خطية / مرتبة

Polynomials as linear / ordered Lists

نذكر فيما يلي عددا من العمليات التي عادة تُجرى على الحدوديات.

العمليات المطلوبة على الحدوديات

- جمع حدوديتين والحصول على حدودية جديدة.
- طرح حدوديتين والحصول على حدودية جديدة.
- ضرب حدوديتين والحصول على حدودية جديدة.
- قسمة حدودية على حدودية أخرى والحصول على حدودية جديدة.
- إيجاد قيمة حدودية في x عند قيمة معينة للمتغير x.
- إيجاد مشتقة حدودية.

ويمكننا أن ننظر للحدودية على أنها قائمة مرتبة / خطية من حدود (terms) يتكون كل منها من معامل وأس (coefficient , exponent) ، فمثلا الحدودية

$$A(x) = 3x^2 + 2x + 4$$

يمكن أن نعتبرها القائمة المرتبة

$$((3, 2), (2, 1) , (4, 0))$$

وعموما الحدودية من درجة n :

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_o$$

يمكن أن نعتبرها القائمة المرتبة

$((a_n, n), (a_{n-1}, n-1), \ldots, (a_1, 1), (a_0, 0))$

المكونة من $n + 1$ حد (term) حيث $a_n \neq 0$.

كيفية تمثيل حدودية How to represent a polynomial

(أ) الطريقة الأولى : استخدام منظومة لجميع الحدود (جميع المعاملات)

Using an array for all terms (all coefficients)

يمكن أن نمثل الحدودية بطريقة مبسطة عن طريق تخزين / تمثيل (storing / representing) الحدودية باستخدام منظومة للمعاملات ، بينما الأسس / القوى (exponents / powers) تُفهم أو تُعرف من ترتيب أو موضع (index / position) العنصر في المنظومة. فمثلا الحدودية التربيعية $A(x) = 3x^2 + 2x+4$ يتم تخزينها عن طريق تخزين :

(i) الدرجة : 2

(ii) المعاملات

3	2	4	
0	1	2	

(i) degree :

(ii) coef :

وكذلك الحدودية من درجة n

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots a_1 x + a_o$$

يتم تخزينها عن طريق تخزين :

(i) الدرجة : n

(ii) المعاملات

a_n	a_{n-1}	a_{n-2}	...	a_{n-i}	...	a_1	a_0
0	1	2		i		n-1	n

(i) degree :

(ii) coef :

وفيما يلي تعريف نوع (type definition) مناسب لتمثيل حدودية بهذه الطريقة.

```
struct poly
{
    int degree;  // Assume 0 . . MaxDegree
    float coef [MaxDegree];
};
```

حيث MaxDegree ثابت يمثل أعلى درجة ممكنة للحدوديات المطلوب تمثيلها. فإذا كانت A حدودية من النوع poly ، وكانت $n \leq MaxDegree$ فإن A ستُمثل هكذا :

A. degree = n ;

A. coef[i] = a_{n-i} ; $0 \leq i \leq n$

حيث a_{n-i} هو معامل x^{n-i} ، ويتم تخزين المعاملات بترتيب تنازلي للأسس ، ويمكن أيضا استخدام الترتيب التصاعدي.

ومثل هذا التمثيل / التخزين بسيط وسهل ويجعل عمليات / خوارزميات الجمع والطرح والضرب وإيجاد القيمة (evaluation) بسيطة ويسهل تنفيذها ، إلا أنها تؤدي إلى هدر / تبديد ذاكرة الحاسوب (wasting computer storage) ، وتضييع لوقت وحدة التشغيل المركزية (CPU time) ، خاصة عندما تكون معاملات كثيرة أصفارا ، فمثلا الحدودية

$$A(x) = 5\,x^{20} + 3\,x$$

يتم تخزينها هكذا :

degree = 20

coef	5	0	0	...	0	0	3	0
	0	1	2			18	19	20

حيث نستخدم ٢١ خلية (cells) تحتوي معظمها (١٩) على أصفار ، أي أن هناك تبديدا للذاكرة.

وبالنسبة لعمليات مثل جمع وطرح الحدوديات فإنها ستكون بسيطة ، إلا أنها قد لا تكون ذات كفاءة عالية حيث أننا معظم الوقت سنتعامل مع أصفار ، ولذلك فبدلا من تمثيل الحدوديات باستخدام منظومة (جميع) المعاملات ، نلجأ للتمثيل بالطريقة التالية.

2

(ب) الطريقة الثانية : استخدام منظومة للحدود غير الصفرية

Using an array for nonzero terms

في هذه الطريقة ننظر إلى الحدودية على أنها قائمة من حدود غير صفرية (list of nonzero terms) ، حيث يتكون كل حد من زوج مرتب (ordered pair) من (معامل غير صفري وأُس) (nonzero coef. , exponent) ، فمثلا الحدودية $A(x) = 5x^{20} + 3x$

نعتبر أنها القائمة

$$((5، 20), (3, 1))$$

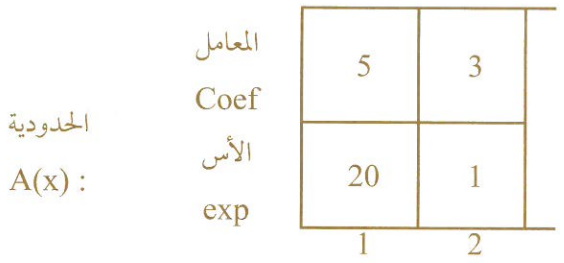
حيث (5 , 20) يعتبر الحد الأول ، و (3 , 1) هو الحد الثاني

والحدودية $A(x) = 15x^{20} + 3x^7$

نعتبرها القائمة

((15، 20), (3, 7)) ، وهكذا .

ثم يمكننا استخدام منظومة سجلات (array of records) لتمثيل (representation) هذه القائمة كما يوضح ذلك الشكل التالي :



حيث تتكون المنظومة من سجلين فقط : السجل الأول (رقم 0) يمثل الحد الأول $5\ x^{20}$ ، والسجل الثاني (رقم 1) يمثل الحد الثاني (غير الصفري) 3x. فإذا رمزنا بالرمز na لعدد الحدود غير الصفرية في الحدودية A(x) ففي هذا المثال تكون na = 2 .

وفيما يلي تعريف نوع مناسب لتمثيل حدودية بهذه الطريقة.

```
const int max_term = 100;  // based on no.  of terms
struct term
{
      float coef;
      int exp;
};
term A[max_term];
```

ومن الواضح أن هذا التمثيل للحدوديات باستخدام منظومة سجلات للحدود غير الصفرية يوفر لنا كثيرا من الذاكرة إذا كانت الحدودية A(x) متناثرة / متفرقة / مبعثرة / هشة (sparse) (أي أن معظم معاملاتها أصفار) ، ولكنه يتطلب نحو ضعف حيز الذاكرة إذا كانت الحدودية ممتلئة / شاملة / كثيفة (dense) (أي أن معظم معاملاتها ليست أصفارا) ، حيث يشتمل كل سجل من سجلاتها على مجالين (أحدهما للمعامل والآخر للأس).

ويمكننا الآن بعد دراسة كيفية تمثيل حدودية أن نبدأ في كتابة بعض الخوارزميات (الدوال) لإجراء عمليات معينة على الحدوديات.

جمع الحدوديات

Addition of Polynomials

نعلم أن جمع الحدوديتين :

$A(x) = x^3 + 2x^5 + 8x^6 + 4x^{15} + 10x^{20} + 2x^{30}$;

$B(x) = 2 + 4x + 4x^5 - 8x^6 + x^{10}$

يعطي الحدودية :

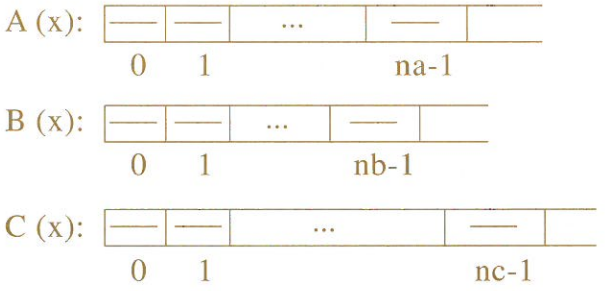
$C(x) = 2 + 4x + x^3 + 6x^5 + x^{10} + 4x^{15} + 10x^{20} + 2x^{30}$.

ونلاحظ أن جمع حدوديتين هو أساسا عملية دمج (merging) قائمتين مرتبتين ، حيث نجمع (add) المعاملات المتقابلة (corresponding coefficients) في الحدود ذات الأسس / القوى المتساوية (equal powers) .

خوارزمية جمع حدوديتين (Algorithm for polynomial addition)

نفرض أن na , nb , nc هي على الترتيب عدد الحدود غير الصفرية في الحدوديات A(x) , B(x) , C(x) . مثلا بالنسبة للحدوديات المذكورة في المثال السابق :

$$na = 6 \ , \ nb = 5 \ , \ nc = 8$$



مثال ٢-١ :

اكتب دالة **poly_add** تقوم بجمع حدوديتين A(x) , B(x) عددا حدودهما غير الصفرية (على الترتيب) na , nb ، وتعطي الدالة الحدودية الناتجة C(x) حيث عدد حدودها غير الصفرية هو nc ، مع افتراض أن الحدود مرتبة تصاعديا بالنسبة للأسس.

الحل :

```
void poly_add (term A[ ], B[ ], C[ ],
                    int na, nb,
                    int& nc)
// input:     A, B, na, nb.
// output:    C, nc.
// assumption: no.  of terms in C( ≡ nc) <= max_term
{
      i=0;   j=0;   k=0;
      while (i < na  && j < nb)
            if ( A[i].exp < B[j].exp )
                  {
                        C[k] = A[i];
                        i = i+1;
                        k = k+1;
                  }
```

```
    else if ( B[j].exp < A[i].exp )
        {
            C[k] = B[j];
            j = j+1;
            k = k+1;
        }
    else     // equal exponents
        {
            t = A[i].coef + B[j].coef;
            if ( t ! = 0 )
                    {
                        C[k].coef = t;
                        C[k].exp = A[i].exp;
                        k = k+1;
                    }
            i = i+1 ;
            j = j+1 ;
        }
// end while }
// Copy remaining terms from A or B into C
        for (i = i; i < na; i + +)
                {
                        C[k] = A[i];
                        k = k+1;
                }
        for (j = j;  j < nb;  j + +)
                {
                        C[k] = B[j];
                        k = k + 1;
                }
```

```
        nc = k;
// end of function poly_add.
```

ملاحظتان :

١) إذا نتج عن عملية جمع حدين متقابلين حد صفري (t = 0) فإنه لا يوضع في الحدودية C(x) ، وإنما نضع في C(x) الحدود غير الصفرية (t!=0) فقط.

٢) سنخرج من عروة while عند انتهاء حدود إحدى الحدوديتين A(x) , B(x) [عندما تكون نتيجتا شرطي while هما true & false] أو عند انتهائهما معا في الوقت نفسه [عندما تكون نتيجتا شرطي while هما false & false] ، وفي أي من الحالتين فإننا ننسخ ما تبقى (إن وجد) من الحدوديتين في الحدودية C(x) [ستبقى حدود من حدودية واحدة فقط أو لا يبقى شئ من كلا الحدوديتين].

مثال ٢-٢ :

اكتب دالة **poly_sub** تقوم بطرح حدوديتين A(x) , B(x) عددا حدودهما na ، nb على الترتيب غير الصفرية ، وتعطي الدالة الحدودية الناتجة C(x) = A(x) - B(x) حيث عدد حدودها غير الصفرية هو nc .

الحل :

خوارزمية الطرح **poly_sub** هي خوارزمية الجمع **poly_add** نفسها ولكن بعد ضرب جميع معاملات الحدودية B(x) في -١.

```
void poly_sub . . . . .
      :
{
      for (j = 0;  j < nb;  j + +)
```

```
            B[j].coef = - B[j].coef;
      :
}
```

حساب درجة تعقيد دالة جمع حدوديتين poly_add

Complexity of function poly_add

نحسب درجة التعقيد - في أسوأ حالة - عن طريق حساب عدد المقارنات بين الأسس (القوى) (exponents / powers) . نفرض أن na , nb هما - على الترتيب - عددا الحدود غير الصفرية في الحدوديتين A(x) , B(x) .

في كل دورة من دورات عروة while إما أن ننسخ حدا من A(x) في C(x) ، أو ننسخ حدا من B(x) في C(x) ، أو نجمع حدين متقابلين (حدا من A(x) وحدا من B(x)) في C(x) .

في أسوأ حالة (كي يكون عدد المقارنات أكبر ما يمكن) :

١- لا يكون هناك أي حدين متقابلين في A(x) , B(x) . أي أنه في كل دورة ننسخ حدا واحدا فقط (إما من A(x) أو من B(x) في C(x)) [هذا يحدث مثلا عندما تكون حدود A(x) فردية القوى ، وحدود B(x) زوجية القوى]. مثلا :

$$A(x) = a_1 x + a_3 x^3 + \ldots + a_{2n+1} x^{2n+1} ;$$

$$B(x) = b_0 + b_2 x^2 + b_4 x^4 + \ldots + b_{2n} x^{2n} ;$$

وبالإضافة إلى ذلك :

٢- لا تبقى بعض الحدود من إحدى الحدوديتين بعد الخروج من عروة while (فتُنسخ هذه الحدود المتبقية كلها مباشرة - دون أي مقارنات - في الحدودية C) ، وإنما تستمر جميع المقارنات داخل العروة حتى تُنسخ

تقريبا جميع حدود الحدوديتين داخل عروة while ، أي أن المؤشرين i, j يصلان معا إلى أقصى قيمتيهما na-1 , nb-1 على الترتيب.

أي أن : عدد العمليات = عدد المقارنات = na + nb - 1

وذلك لأن آخر مقارنة تكون بين آخر حد من A وآخر حد من B ، ويُنسخ أحدهما في C ، ونزيد قيمة المؤشر في الحدودية المقابلة بواحد ، فيصبح شرط while خاطئا ، ونخرج من العروة ، ثم ننسخ الحد الأخير المتبقي من الحدودية الأخرى دون مقارنة ، وبذلك يكون :

عدد المقارنات = عدد حدود إحدى الحدوديتين + (عدد حدود الحدودية الأخرى - ١)

$$= na + nb - 1$$

$$= O(na + nb)$$

مقارنة بين طريقتي تمثيل حدودية لتنفيذ خوارزمية جمع حدوديتين

نقارن فيما يلي بين طريقة تمثيل حدودية باستخدام منظومة لجميع الحدود / المعاملات (الطريقة الأولى أ) وطريقة تمثيلها باستخدام منظومة للحدود غير الصفرية فقط (الطريقة الثانية ب) ، وذلك لتنفيذ خوارزمية جمع حدوديتين ، حيث أننا عند استخدام الطريقة الأولى نجمع الحدوديتين بجمع المعاملات المتقابلة مباشرة ، أما عند استخدام الطريقة الثانية فإننا نجمع الحدوديتين باستخدام خوارزمية (دالة) **poly_add** .

(أ) التخزين (storage)

i) في الطريقة الأولى : نحتاج (درجة الحدودية + 1) خلية / موضع.

ii) في الطريقة الثانية : نحتاج (٢ * عدد الحدود غير الصفرية) خلية

/ موضع ، وذلك لأنه لكل حد غير صفري نحتاج لموضع للمعامل (coef) وآخر للأس (exp) .

فمثلا إذا كانت الحدودية A من درجة n وعدد حدودها غير الصفرية na فإن الحدودية A نحتاج لتخزينها n + 1 خلية في الطريقة الأولى ، ونحتاج لتخزينها 2na خلية في الطريقة الثانية.

فإن كانت الحدودية A متناثرة (sparse) فالطريقة الثانية أفضل ، وإن كانت A كثيفة (dense) فالطريقة الأولى أفضل.

(ب) درجة التعقيد (complexity)

نفرض أن الحدودية A من درجة n وعدد حدودها غير الصفرية na ، وأن الحدودية B من درجة m وعدد حدودها غير الصفرية nb .

عدد العمليات :

i) عند استخدام الطريقة الأولى :

عدد العمليات = max (n , m) = operations ≠

A:

a_0	a_1	a_2	...	a_{n-1}	a_n

B:

b_0	b_1	b_2	...	b_{m-1}	b_m

ii) عند استخدام الطريقة الثانية

عدد العمليات = operations = na + nb - 1

O (na + nb) =

فإن كانت الحدوديتان A , B متناثرتين فالطريقة الثانية أفضل. أما إن كانتا كثيفتين فالطريقة الأولى أفضل.

(ج) بساطة الخوارزمية (simplicity of algorithm)

i) في الطريقة الأولى : الخوارزمية بسيطة جدا.

ii) في الطريقة الثانية : الخوارزمية أعقد قليلا.

يمكننا تلخيص النتائج السابقة في الجدول التالي :

نقطة المقارنة	الطريقة الأولى تخزين جميع المعاملات	الطريقة الثانية تخزين الحدود غير الصفرية
التخزين لحدودية واحدة	درجة الحدودية + ١ = n + 1	٢ * عدد الحدود غير الصفرية = 2 * na
درجة التعقيد ≡ عدد العمليات المطلوبة لجمع حدوديتين من درجتي n , m	max (n , m)	na + nb - 1 = O(na + nb)
بساطة خوارزمية الجمع	بسيطة جدا	أعقد قليلا
متى تفضل ؟	عندما تكون الحدوديتان كثيفتين	عندما تكون الحدوديتان متناثرتين

ملاحظة : عند دراسة خوارزمية جمع حدوديتين لم نأخذ في الاعتبار إمكانية الفيض الزائد عن حيز التخزين المتاح (storage overflow) (والذي يعني أنه لا يوجد لدينا حيز من الذاكرة يكفي لتخزين الحدودية الناتجة (C(x) ، إلا أنه من السهل معالجة هذه المشكلة ، حيث يمكننا مثلا أن نستخدم دالة للتحقق من كفاية الذاكرة ونحن نتعامل مع المنظومات داخل عرى for.

المصفوفات المتناثرة
Sparse Matrices

تمهيد :

نفرض أن A مصفوفة m x n أي تتكون من m صف و n عمود ، الطريقة المعتادة لتخزين المصفوفة A هي أن نستخدم منظومة ثنائية البعد (2-dimensional array) 2D ، حيث نكتب تعريفا / إعلانا مثل :

```
float A [maxrow] [maxcol];
```

فنحتاج إلى mn (أو maxrow * maxcol) خلية (cells) لتخزين المصفوفة A.

واستخدام منظومة ثنائية البعد 2D يعني وصولا عشوائيا / مباشرا ذا كفاءة عالية (efficient direct/random access) إلى عناصر المصفوفة باستعمال مؤشرين i , j (هكذا a_{ij}) يعطيان رقم الصف ورقم العمود حيث يوجد العنصر ، حيث نستغرق الوقت نفسه للوصول إلى أي عنصر ، أي أن هذا الوقت لا يعتمد على قيمتي i , j .

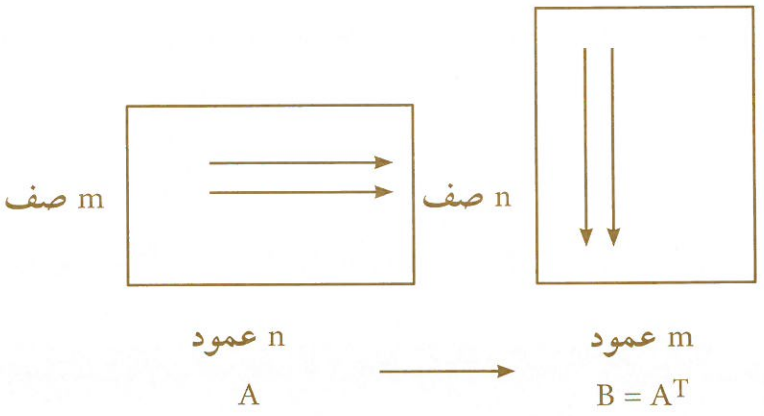
العمليات على المصفوفات :

فيما يلي طائفة من العمليات التي تُجرى عادة على المصفوفات :

- أنشئ create مصفوفة.
- اطبع print مصفوفة.
- اجمع add مصفوفتين.
- اطرح subtract مصفوفتين.
- اضرب multiply مصفوفتين.

- استرجع / خزن retrieve / store العنصر aij رقم i , j .
- دوِّر transpose مصفوفة.
- حل نظاما خطيا Ax = b solve a linear system (مثلا بطريقة جاوس للحذف (Gauss Elimination method) .

عموما هذه العمليات مباشرة وبسيطة ، فمثلا بالنسبة لتدوير المصفوفة :



خوارزمية التدوير (Algorithm for transposing a matrix) :

```
for (i = 0; i < m; i ++)
    for (j = 0; j < n; j ++)
        B_ji = A_ij
```

عدد العمليات (الإسناد"=") = m*n

إذا كانت المصفوفة A كثيفة (dense) (ممتلئة full ، أي أن معظم عناصرها ليست أصفارا) فإن الطريقة المعتادة (القياسية standard) لتمثيل المصفوفات بمنظومات ثنائية البعد 2D ، والخوارزميات المعتادة (القياسية) تكون بسيطة وذات كفاءة عالية ، أما إن كانت المصفوفة متناثرة / مبعثرة / هشة / متفرقة (sparse) فإننا نحجز جزءاً كبيرا من الذاكرة لتخزين أصفار ، وكذلك

يضيع جزء كبير من وقت وحدة التشغيل المركزية (CPU) في التعامل مع أصفار ، وبالتالي تكون الكفاءة عموما منخفضة. وقد تحتوي المصفوفة المتناثرة على عدد كبير من الصفوف والأعمدة (مثلا ١٠,٠٠٠) ويحتوي كل صف على عدد قليل (مثلا ٥ أو ٦) من العناصر غير الصفرية. فلو فرضنا مثلا أن A مصفوفة ١٠,٠٠٠ × ١٠,٠٠٠وفي كل صف خمسة عناصر غير صفرية ، وأننا استخدمنا التمثيل المعتاد للمصفوفات (منظومة ثنائية البعد 2D) فإننا نحتاج إلى ٨١٠ خلية لتخزين المصفوفة وهذا الحجم يتجاوز سعة أكبر الحواسيب. كما أن خوارزميات الجمع والطرح ، ... الخ ستظل معظم الوقت تتعامل مع أصفار. أما إذا قمنا بتخزين العناصر غير الصفرية فقط والتعامل معها فإننا بذلك نوفر حيزا كبيرا من الذاكرة (نحتاج لتخزين ٥٠,٠٠٠ عنصر غير صفري فقط) ، كما نوفر كثيرا من وقت وحدة التشغيل (CPU) ، إلا أن الخوارزميات تصبح أكثر تعقيدا.

وفي المصفوفة المتناثرة قد تكون العناصر غير الصفرية متناثرة / مبعثرة فيها بطريقة عشوائية (apparently random pattern) أو مُجَمَّعة / مكدَّسة (clustered) في مناطق معينة ، مثلا حول القطر.

$$\begin{bmatrix} X & . & . & . & X & . \\ . & . & X & . & . & . \\ . & . & . & . & . & X \\ X & . & . & X & . & . \\ . & X & . & . & . & . \\ . & . & . & . & X & . \end{bmatrix}$$

مصفوفة متناثرة مبعثرة

scattered sparse matrix

$$\begin{bmatrix} X & X & & & & \\ X & X & X & & 0 & \\ & X & X & X & & \\ & & X & X & X & \\ & 0 & & X & X & X \\ & & & & X & X \end{bmatrix}$$

مصفوفة متناثرة قطرية ثلاثية

tridiagonal sparse matrix

تخزين / تمثيل المصفوفة
(Storing / Representing a sparse matrix)

يتم عادة تخزين المصفوفة المتناثرة بتخزين عناصرها غير الصفرية فقط بالطريقة التالية :

نقوم بتخزين العناصر غير الصفرية الموجودة بالصف الأول ، تليها العناصر غير الصفرية الموجودة بالصف الثاني ، وهكذا .. وفي كل صف نقوم بتخزين العناصر بناء على رقم العمود الموجود به العنصر ، حيث يتم تخزين كل عنصر عن طريق تخزين ثلاثة أعداد تمثل : رقم الصف (الموجود به العنصر) ، ورقم العمود ، وقيمة العنصر :

(row no., column no., value)

فمثلا إذا كان $a_{2,5} = 6.0$ فلتخزين هذا العنصر نقوم بتخزين الثلاثية المرتبة (ordered triple) : (2 , 5 , 6.0) . أي أن المصفوفة المتناثرة ستمثلها قائمة من ثلاثيات مرتبة غير صفرية (حيث قيمة العنصر لا تساوي صفرا). ومن المناسب لذلك أن نستخدم لتخزين هذه المعلومات منظومة سجلات كما يلي : [وسنسمي هذه الطريقة للتخزين : التمثيل المتناثر (sparse representation) أو قائمة القيم غير الصفرية (list of nonzeros)] :

```
const int maxnonzeros = 10000;
struct element
{
      int row;
      int col;
      float value;
}
element a[maxnonzeros], b[maxnonzeros];
```

مثال ٢-٣ :

وضح كيفية تخزين المصفوفة المتناثرة A التالية باستخدام التمثيل المتناثر .

$$A = \begin{pmatrix} 0 & 7.0 & 0 & 3.0 & 0 & 0 \\ 4.0 & 0 & 0 & 0 & 0 & 8.0 \\ 2.0 & 0 & 5.0 & 0 & 0 & 0 \\ 0 & 10.0 & 0 & 0 & 0 & 0 \end{pmatrix}_{4x6}$$

لاحظ أن A تحتوي على عدد من العناصر غير الصفرية يساوي t = 7.

الحل :

a_0	0 , 1 , 7.0
a_1	0 , 3 , 3.0
a_2	1 , 0 , 4.0
a_3	1 , 5 , 8.0
a_4	2 , 0 , 2.0
a_5	2 , 2 , 5.0
a_6	3 , 1 , 10.0

سعة التخزين المستخدمة للمصفوفات (storage used)

إذا فرضا أن A مصفوفة m x n (أي تحتوي على m صف و n عمود) فإن التمثيل المعتاد بمنظومة ثنائية البعد 2D يتطلب عددا من الخلايا (cells) يساوي C_{2D} = m * n ، بينما إذا استخدمنا التمثيل المتناثر وكان عدد العناصر غير الصفرية في المصفوفة هو t عنصرا ، فإننا نحتاج لعدد من الخلايا يساوي ، [مثلا في المثال السابق :

$$C_{2D} = 4 \times 6 = 24$$

$$[\ C_s = 3 \times 7 = 21$$

حيث افترضنا أن حيز التخزين المطلوب لتخزين عدد صحيح يساوي الحيز المطلوب لتخزين عدد حقيقي ، وهذا عادة غير صحيح.

مثال ٢-٤ :

اكتب خوارزمية **Search** لاسترجاع (retrieve) قيمة العنصر الذي ترتيبه (i , j) ، أي الموجود في الصف رقم i والعمود رقم j في التمثيل المتناثر (قائمة القيم غير الصفرية / منظومة سجلات القيم غير الصفرية) ، بحيث تعيد الخوارزمية قيمة العنصر إن وجد ، وتعيد صفرا إن لم يوجد. ثم احسب درجة تعقيد الخوارزمية.

الحل : الخوارزمية :

- نبحث أولا عن الصف رقم row = i .
- ثم نبحث عن العمود رقم col = j في هذا الصف رقم i .

[والبحث في الحالتين قد يكون تتابعيا (sequential) أو ثنائيا (binary)]

- إذا وجدنا عنصرا فإن الخوارزمية تعيد قيمته وإلا فإنها تعيد صفرا.

واضح أن هذه الخوارزمية - أو غيرها - (مع منظومة السجلات غير الصفرية) ستكون أكثر تعقيدا من طريقة الوصول المباشر في حالة استخدام المنظومة ثنائية البعد 2D .

درجة التعقيد :

يمكن إثبات أن درجة تعقيد الخوارزمية مع منظومة السجلات (التمثيل المتناثر) ستكون في حالة البحث التتابعي O(t) ، حيث t تمثل عدد القيم غير الصفرية في المصفوفة. [البحث عن الصف رقم i يتطلب في أسوأ حالة t خطوة (عدد السجلات = t) ، والبحث عن العمود رقم j يتطلب في أسوأ حالة n خطوة (عدد الأعمدة = n) ، وعادة t > n) .

مثال ٢-٥ :

اكتب خوارزمية **sparse_insert (i , j , value , A)** لإدخال / تخزين (insert / store) عنصر (غير صفري) (i , j , value) في الموضع i , j في المصفوفة A (ليحل محل عنصر صفري $a_{ij} = 0$) باستخدام التمثيل المتناثر [مثلا لإدخال العنصر (3 , 2 , 5.2) في المصفوفة A السابقة]. واحسب درجة تعقيد هذه الخوارزمية.

الحل : الخوارزمية :

- نحتاج أولا إلى عملية بحث (مثلا باستخدام الخوارزمية **Search** في المثال السابق) للوصول إلى الموضع المطلوب i , j [مثلا : البحث عن الصف رقم 3 ، ثم البحث عن العمود رقم 2 في هذا الصف رقم 3].
- ثم نقوم بعملية إزاحة (shifting) للسجلات عند وأسفل هذا الموضع خطوة واحدة لأسفل (one step down) .
- ثم نقوم بإدخال / تخزين العنصر المطلوب.

درجة التعقيد :

يمكن إثبات أن درجة تعقيد هذه الخوارزمية تساوي في أسوأ حالة worst_case complexity = O(t) حيث t هي عدد العناصر غير الصفرية بالمصفوفة.

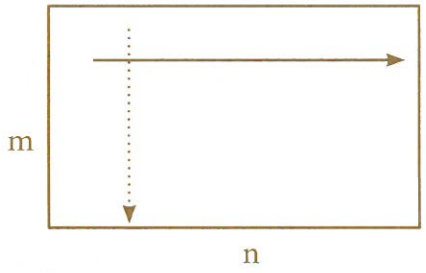
(حتى لو استخدمنا البحث التتابعي فإن درجة التعقيد تساوي t 2 أي تساوي O(t))

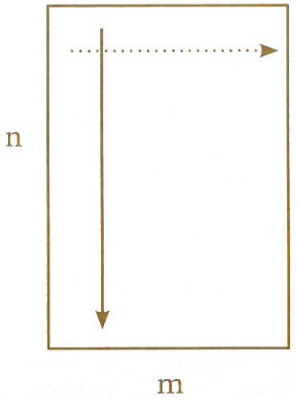
[لاحظ أن البحث يتطلب O(t) والإزاحة لأسفل تتطلب O(t) (في أسوأ حالة)]

تدوير المصفوفات

Matrix Transposition

نفرض أن A مصفوفة m x n ، وأننا نريد إيجاد مدور هذه المصفوفة أي المصفوفة $B = A^T$ وهي مصفوفة n x m .





مثلا إذا كانت المصفوفة A هي المصفوفة المعطاة في مثال 2-3 ، فإن مدورها B = AT هي المصفوفة :

$$B = A^T = \begin{pmatrix} 0 & 4.0 & 2.0 & 0 \\ 7.0 & 0 & 0 & 10.0 \\ 0 & 0 & 5.0 & 0 \\ 3.0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 8 & 0 & 0 \end{pmatrix}$$

وإذا كانت المصفوفة A مخزونة كمنظومة ثنائية البعد 2D فيسهل إيجاد B (كما سبق ذكره) هكذا :

```
for (i = 0; i < m; i ++)
    for (j = 0; j < n; j ++)
```

$b_{ji} = a_{ij}$

// end for j

// end for i

وهذه خوارزمية بسيطة إلا أن درجة التعقيد لها تساوي m x n وهذه قيمة كبيرة نسبيا. وإذا كانت المصفوفة متناثرة فإننا نقوم بتخزين وتحريك عدد كبير من الأصفار.

نفرض الآن أن A مخزونة بالصيغة / الصورة المتناثرة (sparse form) (row , col , value) = (i , j , value) في منظومة أحادية البعد 1D [أي في منظومة أحادية البعد تتكون من العناصر غير الصفرية فقط ، ونفرض أن عددها t]. ونود تخزين B أيضا بالصيغة نفسها ، حيث ستحتوي كل من A , B على عدد t من العناصر غير الصفرية.

سندرس فيما يلي بإذن الله تعالى ثلاث طرق مختلفة للحصول على المصفوفة المدورة B ، ونقارن بين هذه الطرق الثلاث من حيث درجة التعقيد.

(أ) الطريقة المباشرة (الأسلوب القسري)

(Straight-forward Approach / Brute-Force Approach)

- ابدأ بالمنظومة B خالية.
- أدخل عناصر A واحدا واحدا في المنظومة B (في مواضعها الصحيحة) باستخدام الخوارزمية **sparse_insert** هكذا :

```
for (k = 0; k < t; k++)
    {
        i = A[k] . row ;
```

```
        j = A[k] . col ;
        value = A[k] . value ;
        sparse_insert (j , i , value , B)
    }
```

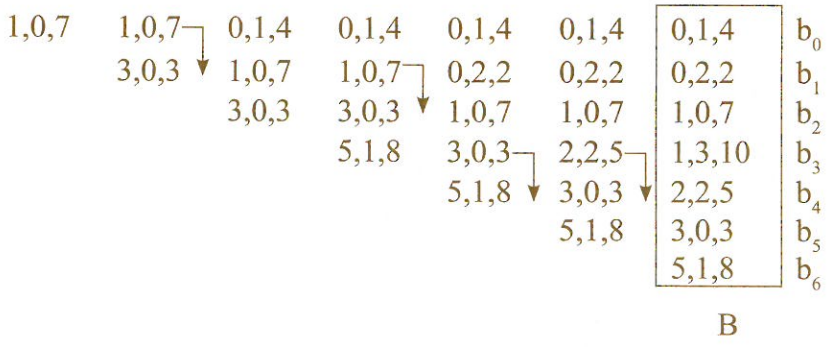

مثال ٢-٦ :

باستخدام الصيغة المتناثرة لتمثيل المصفوفات أوجد مدوَّر المصفوفة المتناثرة A المذكورة في مثال ٢-٣ باتباع الطريقة المباشرة (الأسلوب القسري).

الحل : نبدأ بالتمثيل المتناثر للمصفوفة A :

0 , 1 , 7
0 , 3 , 3
1 , 0 , 4
1 , 5 , 8
2 , 0 , 2
2 , 2 , 5
3 , 1 , 10

باتباع خطوات الطريقة المباشرة نحصل على المصفوفة $B = A^T$ كما يلي :



درجة تعقيد الطريقة المباشرة

في هذه الطريقة نمر على جميع عناصر المنظومة (المصفوفة) A(scan) مرة واحدة فقط ، إلا أن المشكلة الأساسية أو الصعوبة تكمن في كيفية تخزين أو إدخال العناصر في المنظومة B ، حيث يتطلب ذلك تحركات / إزاحات (moves / shifts) عديدة.

ويمكن إثبات أن عدد العمليات يساوي $O(t^2)$ (بالضبط يساوي $\frac{t^2}{2}$) . [لاحظ أن عدد عناصر A غير الصفرية يساوي t ، وإدخال كل عنصر بخوارزمية الإدخال يتطلب O(t) خطوة].

2

(ب) طريقة نقل الأعمدة إلى صفوف :

في هذه الطريقة نقوم بتدوير المصفوفة A عن طريق نقل أعمدتها بالترتيب إلى صفوف B ، أي :

- انسخ عناصر العمود الأول من A في الصف الأول من B .
- ثم انسخ عناصر العمود الثاني من A في الصف الثاني من B .
- .....
- ثم انسخ عناصر العمود رقم j من A في الصف رقم j من B .
- .....

وهكذا إلى أن تنتهي من جميع أعمدة A.

نلاحظ أن عملية إدخال العناصر في B سهلة وبسيطة ، حيث دائما ندخل أي عنصر في النهاية (insert at end) . فمثلا بالنسبة للمنظومة A السابقة (المذكورة في مثال ٢-٣) يكون إدخال عناصر مدورها $B = A^T$ بالترتيب التالي :

				...		
0,1,4	0,1,4	1,0,7				5,1,8
b_0	b_1	b_2				b_6

إلا أنه ستظل هناك صعوبة وهي في عملية استرجاع العناصر من A وذلك لأن عناصر A مخزونة صفا صفا (وليس عمودا عمودا) ، ولذا فيجب أن نجتاز / نمسح / نمر على جميع عناصر A (scan) مرة لكل عمود.

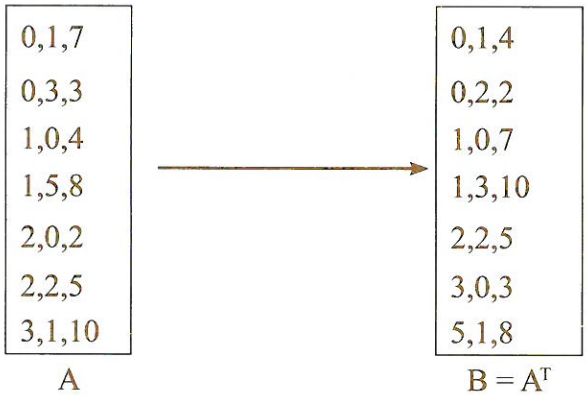
الخوارزمية :

```
void Transpose (element A[maxnonzeros],
                element B[maxnonzeros],
                int t, m, n)
// Transpose of A is placed in B
{
    int j, k, ℓ

    if t > 0          // nonzero matrix
        {
            ℓ = 0;
            for (j = 0; j < n; j++)
                // copy col # j of A into B
                for (k = 0; k < t; k++)
                    if (A[k]. col = = j) // an elt. in col. j is
                        // found; move it to B
            {
                B[ℓ]. row = A[k].col;
                B[ℓ]. col = A[k].row;
                B[ℓ]. value = A[k].value;
                ℓ = ℓ + 1;
            }
        }
}
```

مثال ٢-٧ :

استخدم خوارزمية طريقة نقل الأعمدة إلى صفوف لإيجاد منظومة مدور المصفوفة A (المعطاة في مثال ٢-٣) بفرض استخدام التمثيل المتناثر.

الحل :



درجة التعقيد : واضح أن درجة التعقيد لهذه الخوارزمية تساوي O(nt)

{لدينا عروتان متداخلتان :

```
for (j = 0; j < n; j ++)
    for (k = 0; k < t; k ++)
        .
        .
        .
    // end for k.
// end for j.
```

وهذه عموما أفضل من درجة التعقيد للخوارزمية السابقة (الطريقة المباشرة) [$O(t^2)$] ، وذلك لأنه عموما n < t [مثلا بالنسبة لمصفوفة 100 x 100بكل صف من صفوفها خمسة عناصر غير صفرية : t = 500 > n = 100]

ولكنها مازالت أسوأ من O(mn) وذلك لأنه عادة t > max (m , n) وبالتالي O(nt) > O(mn) .

ملاحظة :

إذا كانت المصفوفة A كثيفة ، أي أن [t ≅ mn] (t = order of mn) ، فإن درجة تعقيد الخوارزمية تصبح :

$$O(nt) = O(n \cdot mn) = O(mn^2)$$

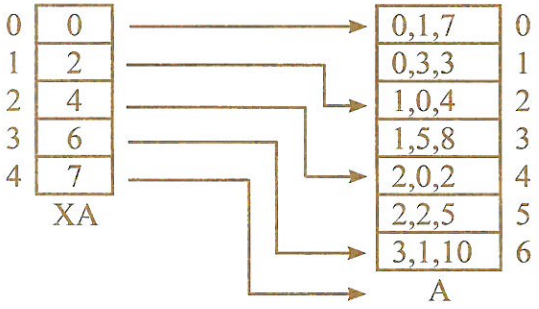
وهذه أسوأ من O(mn) التي نحصل عليها من التمثيل ثنائي البعد 2D.

والسؤال الآن الذي نود الإجابة عليه هو : هل يمكننا الحصول على خوارزمية درجة تعقيدها أفضل ؟ والذي يدفعنا إلى هذا السؤال هو أننا في عملية التدوير نقوم أساسا بتحريك عدد t من العناصر من A إلى B (ولكن يتم هذا التحريك بعد إعادة ترتيب هذه العناصر) ، فأفضل درجة تعقيد يمكن أن نصل إليها أو نطمح إليها (كحد أسفل lower bound) تكون O(t) . والخوارزمية التالية (طريقة التدوير السريع) تصل فعلا درجة تعقيدها إلى هذا الحد الأدنى الذي نصبو إليه.

(ج) طريقة التدوير السريع (Fast Transpose Algorithm)

سنقوم بعمل تعديل طفيف في طريقة تمثيل المصفوفة المتناثرة Am,n ، حيث نقوم (بالإضافة إلى تخزين العناصر غير الصفرية) بتخزين منظومة أعداد صحيحة طولها (أي عدد عناصرها) m + 1 تعمل كمؤشر (index) إلى الموضع الابتدائي (starting location) لكل صف من صفوف المصفوفة ، أي أنها تعمل كمنظومة مؤشرات (array of pointers) ، وسنرمز لها بالرمز XA (الحرف X مأخوذ من كلمة INDEX) ، حيث يشير العنصر XA[i] من هذه المنظومة إلى بداية الصف رقم i من المصفوفة $A_{m,n}$ ، (أي أن قيمة العنصر XA[i] تعطي موضع أول عنصر غير صفري من عناصر الصف رقم i من المصفوفة $A_{m,n}$ في منظومة السجلات A

المستخدمة لتخزين العناصر غير الصفرية ومواضعها) ، كما يوضح ذلك الشكل التالي بالنسبة للمصفوفة المتناثرة A المشار إليها سابقا ، وبالتالي تشير قيمة التعبير XA[i + 1] - 1 إلى نهاية الصف رقم i من المصفوفة $A_{m,n}$ ، أي يعطي هذا التعبير موضع آخر عنصر غير صفري من عناصر الصف رقم i من المصفوفة $A_{m,n}$ في منظومة السجلات A.



مثلا :

XA[2] = 4 أي أن قيمة العنصر رقم 2 في المنظومة XA تساوي 4 ، وهذا يعني أن بداية الصف رقم 2 (i = 2) (أي أول عنصر غير صفري من الصف رقم 2) تقع عند موضع السجل رقم 4 من سجلات المنظومة A.

وكذلك XA[3] - 1 = 6 - 1 = 5 تعني أن الصف رقم 2 (i = 2) ينتهي (أي يقع آخر عنصر غير صفري من عناصره) في الموضع رقم 5 (أي عند السجل رقم 5 من السجلات المخزونة) في المنظومة A .

وتعد هذه الطريقة لتخزين عناصر المصفوفة A وسيلة مناسبة للوصول إلى صفوف A بطريقة مباشرة / عشوائية (random).

ملاحظة : i) في الطريقة الأولى (الأسلوب القسري) :

نضطر إلى عمل إزاحات عديدة حتى نتمكن من تخزين العناصر في أماكنها

الصحيحة. فالصعوبة أساسا هي أننا لا نعرف أين نضع العنصر في B في مكانه بالضبط إلى أن يتم تشغيل جميع العناصر التي تسبقه ، ودرجة التعقيد هنا تكون كبيرة $O(t^2)$.

ii) وفي الطريقة الثانية (نقل أعمدة A إلى صفوف B) :

نتغلب على هذه المشكلة بأن نحرك عناصر A بالترتيب الذي نريده لتخزين هذه العناصر في B ، ولذلك ننقل عناصر أعمدة A بترتيب الأعمدة لأنها ستكون صفوف B .

ولكن المشكلة أو الص عوبة هنا هي أننا لا نحصل على / لا نسترجع عناصر A بسهولة لأنها مخزونة في المنظومة A حسب الصفوف وليس حسب الأعمدة ، ودرجة التعقيد هنا تكون أيضا كبيرة (وإن كانت أقل من السابقة) O(nt) .

iii) وفي هذه الطريقة الثالثة (التدوير السريع) :

نتغلب على مشكلتي الطريقتين السابقتين باتباع الخوارزمية التالية :

خوارزمية طريقة التدوير السريع :

أ. نحدد أولا (في منظومة «S») عدد العناصر - غير الصفرية - في كل عمود من أعمدة A (أي في كل صف من صفوف B) .

ب. ثم نحدد (في منظومة «U» وهي منظومة مؤشرات B) مواضع بدايات صفوف المصفوفة B في المنظومة B.

ج. وبالتالي يسهل تحريك عناصر A - عنصرا عنصرا - بحيث يذهب كل عنصر إلى مكانه الصحيح في المنظومة B مباشرة ، دون عمل أي إزاحات. أي أننا تغلبنا على الصعوبات في الطريقتين السابقتين ، ودرجة التعقيد هنا أقل منها في كل من الطريقتين السابقتين ، حيث أنها هنا تساوي (كما سنرى بإذن الله تعالى).

(أ) تعيين المنظومة S :

وهذه المنظومة تشير إلى حجم (Size) كل صف من صفوف B - أي عدد عناصره غير الصفرية. أي أن :

عدد العناصر غير الصفرية في الصف رقم i في المصفوفة B هو Row = S[i]

Size [i] : أي عدد العناصر غير الصفرية في العمود رقم i من المصفوفة A .

```
// Initialization
for (i = 0; i < n; i ++)
        S[i] = 0 ;

// count no of (nonzero) elements in each col.  of A
// note that total number of (nonzero) elements is t
for (i = 0; i < t; i ++)
        S[A[i] . col] = S[A[i] . col] + 1 ;
```

(ب) تعيين المنظومة U :

وهي المنظومة التي تعد عناصرها - في البداية - RowStart ≡ XB ≡ U مؤشرات إلى مواضع بدايات صفوف B.

أي أن : قيمة موضع العنصر الأول من الصف رقم i من المصفوفة B في المنظومة U[i] : B .

ومن الواضح أنه إذا جمعنا حجم أي صف من صفوف B (أي عدد عناصره غير الصفرية) على موضع بداية هذا الصف ، فإننا نحصل على موضع بداية الصف التالي ، أي أن

```
U[i] = U[i-1] + S[i-1];
```

أي أن عناصر المنظومة U ≡ منظومة نقاط البداية (starting points) لصفوف B نحصل عليها كما يلي :

```
// find values of U[i] ≡ starting position of row i in B.
U[0] = 0;
for (i = 1; i < n; i ++)
        U[i] = U[i - 1] + S[i - 1] ;
```


(ج) نقل عناصر A إلى B :

الآن أصبحنا جاهزين لنقل عناصر A - عنصرا عنصرا - إلى مواضعها الصحيحة مباشرة في B دون أي إزاحات ، مع ملاحظة أنه بعد نقل أي عنصر من A إلى موضعه الصحيح في المنظومة B فإننا نحرك مؤشر بدايات المواضع U [والذي يعمل أيضا كمؤشر لأول موضع خالي تالي (next free position) بالنسبة لعناصر صف معين من B] ، أي نزيد قيمته بواحد ;U[j] = U[j] + 1 لأنه مثلا إذا كان U[2] = 4 , U[1] = 2

فمعنى هذا أن هناك مكانين / موضعين محجوزين لعناصر الصف الثاني (الصف رقم 1) من B ، أي أن هناك المكانين B2 , B3 لعناصر الصف الثاني من B ، وذلك لأن S[1] = 2 ، أي أن الصف الثاني من B (أو العمود الثاني من A) به عنصران غير صفريين (قيمة العنصر الأول 7 وقيمة العنصر الثاني 10) ، فبعد نقل العنصر الأول 7 أي بعد نقل السجل A0 = (0, 1, 7) ليصبح بعد التعديل السجل B2 = (1, 0, 7) ، فإن العنصر الثاني 10 يجب أن يوضع في الموضع الثاني من المواضع المخصصة للصف الثاني من B ، أي أن السجل A6 = (3, 1, 10) ينقل بعد التعديل ليصبح B3 = (1, 3, 10) ، ولا يوضع في الموضع B2 ، ولذلك فبعد نقل العنصر الأول إلى الموضع B2 لأن U[1] = 2 فإننا نعدِّل أو نحرك المؤشر U[1] الذي يشير إلى بداية الصف الثاني (الصف رقم 1) من صفوف B ليصبح :

$$U[1] = U[1] + 1 = 2 + 1 = 3$$

ليشير بذلك إلى الموضع الخالي الذي يوضع فيه أول عنصر قادم فيما بعد من عناصر صفوف B ، أي ليوضع فيه بعد ذلك العنصر الذي قيمته 10 ، أي ليوضع فيه السجل B3 = (1, 3, 10) .

ولذلك فعموما بعد نقل أي عنصر من A إلى B نحرك المؤشر U هكذا :

$$U[j] = U[j] + 1$$

حيث j هو رقم الصف (في B) الذي ينقل إليه عنصر من العمود رقم j (في A) ، أي أن j = A[k].col حيث k هو الرقم الترتيبي للعنصر الذي ننقله من المنظومة A ، أي رقم السجل A_k.

```
// move from A to B
for (k = 0; k < t; k ++)
{
    j = A[k]. col; // find col. in A from which we move the element.
    ℓ = U[j] ; // determine the destination of the element moved to B,
        // i.e. ℓ is the correct new position of the element in B.
    B[ℓ] .row = A[k] . col ;          // = j
    B[ℓ] .col = A[k] . row ;
    B[ℓ] .value = A[k] . value ;
    U[j] = U[j] + 1; // U[j] points to the first next
                     // free position in row number j of B.
}
```

لاحظ أنه من مميزات هذه الخوارزمية :

١) ننقل العناصر من A إلى أماكنها الصحيحة (النهائية) في B مباشرة دون عمل أي إزاحات لأي عناصر (لأي سجلات).

٢) نمر على جميع عناصر A (scan) مرة واحدة فقط صفا صفا [بدلا من عدة مرات في الخوارزمية السابقة (نقل الأعمدة إلى صفوف)].

ملاحظة :

المنظومة U ستُمحى (destroyed) بعد هذا ، ولكن يمكن تكوينها بسهولة باستخدام المنظومة S كما سبق.

فيما يلي الدالة **FastTranspose** التي تطبق هذه الخوارزمية ، يليها تتبع تنفيذها على المصفوفة A السابقة ، ثم نوجد درجة تعقيدها.

```
void FastTranspose (sparse_matrix A, B, int m, n, t)
// The transpose of A is placed in B.
{
    int S[MaxCol], U[MaxCol] ;
        // S[i] : size of row number i,
        // U[i] : starting position of row number i,
    int i , j , k , ℓ ;
    if (t > 0)              // nonzero matrix;
{
    // compute S[i] ≡ no.  of terms in row i of B.
    for (i = 0; i < n; i ++)
    // count no.  of elements in each col.  of A
```

```
        S[A[i] . col] = S[A[i] . col] + 1;
    // find values of U[i] ≡ starting position of row i in B.
    U[0] = 0 ;
    for (i = 1; i < n; i ++)
        U[i] = U[i - 1] + S[i - 1];
    // move from A to B
    for (k = 0; k < t; k ++)
    {
        j = A[k] . col;          // find col.  of A from which we
                                 // move the element.
        ℓ = U[j]; correct position of the elt.  moved to B.

        B[ℓ] . row = A[k] . col;      // =j
        B[ℓ] . col = A[k] . row;
        B[ℓ] . value = A[k] . value;

        U[j] = U[j] + 1;
        // U[j] : points to the first next free position in
        //      row j of B.
    } // end for k.
  } // end if.
} // end FastTranspose
```

مثال ٢-٨ :

تتبع (trace) يدويا تنفيذ الدالة **FastTranspose** التي تقوم باتباع طريقة التدوير السريع لتدوير مصفوفة متناثرة A وتخزين مدورها في مصفوفة متناثرة B ، وذلك بفرض أن المصفوفة المتناثرة A هي المصفوفة المعطاة في مثال ٢-٣ .

الحل :

	row	col	value
0	0	1	7
1	0	3	3
2	1	0	4
3	1	5	8
4	2	0	2
5	2	2	2
6	3	1	10

A
t = 7

أولا : تعيين المنظومة S :

	S
0	0 →1 →2
1	0 →1 →2
2	0 →1
3	0 →1
4	0
5	0 →1

	S
0	2
1	2
2	1
3	1
4	0
5	1

$S[A[0].col] \equiv S[1] = S[1] + 1$

$S[3] = S[3] + 1$

$S[0] = S[0] + 1$

$S[5] = S[5] + 1$

$S[0] = S[0] + 1$

$S[2] = S[2] + 1$

$S[1] = S[1] + 1$

ثانيا : تعيين المنظومة U :

	U
0	0
1	$U_0 + S_0 = 0 + 2 = 2$
2	$U_1 + S_1 = 2 + 2 = 4$
3	$U_2 + S_2 = 4 + 1 = 5$
4	$U_3 + S_3 = 5 + 1 = 6$
5	$U_4 + S_4 = 6 + 0 = 6$

	U
0	0
1	2
2	4
3	5
4	6
5	6

ثالثا : تعيين المنظومة B :

for (k = 0; k < 7; k ++)

k = 0, j = A[k] . col

j = 1, ℓ = U[1] = 2

B[2] . row = A[0] . col = 1

B[2] . col = A[0] . row = 0

B[2] . value = A[0] . value = 7

U[1] = U[1] + 1 = 2 + 1 = 3

k = 1

j = A[1] . col = 3

ℓ = U[3] = 5

B[5].row = A[k] . col = A[1] . col = 3

B[5].col = A[1] . row = 0

B[5].value = 3

U[3] = U[3] + 1 = 5 + 1 = 6

k = 2

j = A[2] . col = 0

ℓ = U[0] = 0

B[0].row = A[2] . col = 0

B[0].col = 1

B[0].value = 4

U[0] = U[0] + 1 = 0 + 1 = 1

k = 3

j = A[3] .col = 5

ℓ = U[j] = U[5] = 6

B[6].row = A[k] . col = A[3] . col = 5

B[6].col = 1

B[6].value = 8

U[5] = U[5] + 1 = 6 + 1 = 7

k = 4

j = A[4] .col = 0

ℓ = U[0] = 1

B[1].row = A[4] . col = 0

B[1].col = 2

B[1].value = 2

U[0] = U[0] + 1 = 1 + 1 = 2

k = 5

j = A[5] .col = 2

ℓ = U[2] = 4

B[4].row = A[5] . col = 2

B[4].col = 2

B[4].value = 5

U[2] = U[2] + 1 = 4 + 1 = 5

k = 6

j = A[6] .col = 1

= U[1] = 3

B[3].row = A[6] .col = 1

B[3].col = 3

B[3].value = 10

U[1] = U[1] + 1 = 3 + 1 = 4

ويمكننا أن نلخص هذا التتبع للدالة **FastTranspose** والذي يتكون أساسا من ثلاث خطوات : تعيين المنظومة S (حجم صفوف B) ، وتعيين المنظومة U (بدايات صفوف B) ، ثم تعيين المنظومة B في حل المثال التالي ، الذي يعتبر هو نفسه المثال السابق بعد إعادة صياغته بصورة أخرى.

مثال ٢-٩ :

أوجد مدوَّر المصفوفة المتناثرة A (المنظومة A) المعطاة في مثال ٢-٣ وذلك باستخدام طريقة التدوير السريع.

الحل :

تعيين B		تعيين U ≡ XB		تعيين S	
عناصر المنظومة B		بدايات صفوف B		حجم صفوف B	
b_0	0, 1, 4	U_0	0 →1→2	S_0	0 →1→2
b_1	0, 2, 2	U_1	2 →3→4	S_1	0 →1→2
b_2	1, 0, 7	U_2	4 →5	S_2	0 →1
b_3	1, 3, 10	U_3	5 →6	S_3	0 →1
b_4	2, 2, 5	U_4	6	S_4	0
b_5	3, 0, 3	U_5	6 →7	S_5	0 →1
b_6	5, 1, 8				

درجة تعقيد الخوارزمية (طريقة التدوير السريع)

نلاحظ أن هناك في الدالة **FastTranspose** أربع عُرَى (4 loops) وأن عدد مرات تنفيذها على الترتيب هو :

for (i = 0; i < n; i ++) : n من المرات

t من المرات : for (i = 0; i < t; i ++)

n - 1 من المرات : for (i = 1; i < n; i ++)

t من المرات : for (k = 0; k < t; k ++)

وفي كل واحدة من هذه العرى يستغرق تنفيذ الدورة الواحدة / التكرير الواحد (each iteration) زمنا معينا (ثابتا) (constant amount of time) ، وبالتالي فإن درجة التعقيد التقاربية تساوي asymptotic complexity = O(n+t) [الوقت المستغرق لإجراء الحسابات (computing time)] .

وعندما تكون t في حدود m × n (of the order of) - أي عندما تكون المصفوفة كثيفة - فإن زمن الحسابات يصبح

computing time = O(n + m * n) = O(mn)

وهي درجة التعقيد نفسها في حالة استخدام المنظومات ثنائية البعد 2D ، إلا أن المعامل الثابت (constant factor) يكون أكبر في حالة طريقة التدوير السريع عنه في حالة التمثيل ثنائي البعد 2D . وعندما يكون t << m*n (أي عندما تكون المصفوفة متناثرة) فإن طريقة التدوير السريع تكون أسرع بكثير. وبالتالي فإننا في هذه الطريقة نوفر كلا من حيز التخزين ووقت التنفيذ ، وهذا ليس صحيحا في حالة الطريقة الثانية (طريقة نقل أعمدة A إلى صفوف B) حيث درجة التعقيد O(nt) ، وعادة تكون t > max(m , n) وبالتالي تكون nt ≥ mn . والمعامل الثابت في خوارزمية هذه الطريقة الثانية أيضا دائما أكبر من 1 (وهو المعامل الثابت في خوارزمية المنظومة ثنائية البعد 2D) .

Complexity|FastTranspose = O(n + t)

= O(t) لأنه عادة تكون n < t

ويلاحظ أن طريقة التدوير السريع تتطلب سعة تخزين (space) أكبر من تلك التي تتطلبها الطريقة الثانية (ب) (نقل الأعمدة إلى صفوف) ، وعموما يمكن تقليل سعة التخزين في طريقة التدوير السريع باستخدام حيز معين من الذاكرة في تخزين / تمثيل كل من المنظومتين S , U ، أي نستفيد (utilize) من الحيز (space) نفسه لتمثيل هاتين المنظومتين.

الجدول التالي يقارن بين درجات تعقيد الخوارزميات سالفة الذكر لتدوير مصفوفة

التمثيل ثنائي البعد 2D	التمثيل المتناثر		
	أ) الطريقة المباشرة : نقل عناصر A واحدا واحدا	ب) نقل أعمدة A بالترتيب إلى صفوف B	ج) التدوير السريع
m * n	$O(t^2)$	O(n.t)	O(n+t) ≅ O(t)

درجة تعقيد عملية تدوير مصفوفة

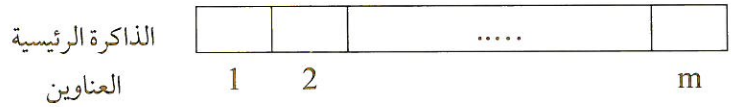
تمثيل المنظومات في الذاكرة

Representation of arrays in memory

المسألة الأساسية التي نود مناقشتها فيما يلي هي أننا إذا قمنا بتخزين منظومة في الذاكرة فإننا نود الحصول على قانون أو صيغة تعطى عنوان أي عنصر من المنظومة (أي موضعه) في الذاكرة إذا علمنا عنوان (موضع) أول عنصر من المنظومة ، وأبعاد المنظومة.

تعد الذاكرة الرئيسية (main memory) للحاسوب منظومة أحادية البعد 1D array من مكونات تسمى كلمات الحاسوب (computer words) [والكلمة الواحدة عادة تتكون من : ٢/ ٤/ ٦/ 8 من البايتات (bytes ٢/ ٤/ ٦/ ٨) ، حيث البايت الواحدة تساوي ٨ بتات (وحدات ثنائية) (1 byte = 8 bits)].

وهذه الكلمات لها عناوين (addresses) متسلسلة 1 , 2 , 3 , ... , n.



وعملية تمثيل بنية منظومة (array structure) هي أساسا عملية إسقاط (mapping) للمنظومة المجردة المكونة من وحدات / عناصر من النوع T (abstract array with components of type T) إلى (onto) الذاكرة التي هي منظومة مكونة من وحدات من النوع word

storage ≡ array with components of type word

ويجب أن يتم هذا الإسقاط (mapping) بحيث يكون حساب عناوين مكونات المنظومة (العناصر) أسهل ما يمكننا (وبالتالي بأكفأ ما يمكن).

a : array [index] of type element (T)

↕ mapping

storage: array [address] of type word

فإذا احتاج العنصر الواحد إلى كلمة واحدة بالضبط فإن تخزينه يكون سهلا ومباشرا [إسقاط واحد لواحد (mapping 1 to 1)] .

نفرض أننا سنقوم بتخزين منظومة A معرفة كما يلي :

element_type A $[u_0]$ $[u_1]$ . . . $[u_{n-1}]$;

أي أن A منظومة نونية البعد nD (n-dimensional array)

حيث n = 1 , 2,....

ونفرض أن العنصر الواحد يتطلب تخزينه عدد ؟ من الكلمات ($\beta = 1$) ، أي أن طوله كلمة واحدة.

سؤال : ما هو عنوان ذاكرة الحاسوب المستخدمة لتخزين العنصر
(address of computer memory of an array element)

$$A [i_0] [i_1] [i_2] \ldots [i_{n-1}]$$

من المنظومة A ، بفرض أن عنوان تخزين العنصر الأول

$$A [0] [0] [0] \ldots [0]$$

من المنظومة هو α (أي أن عنوان خلية الذاكرة (memory cell) المخزون فيها هذا العنصر الأول هو α) ؟ [ملاحظة : المترجم (compiler) هو الذي يقوم بهذه الخطوات].

سنجيب بإذن الله على هذا السؤال :

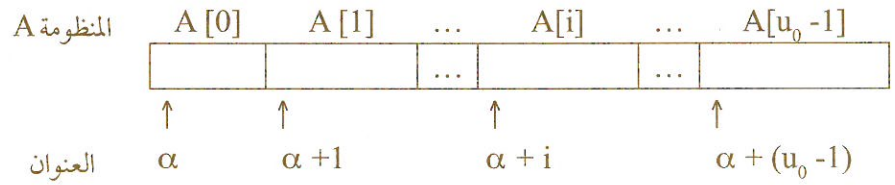
- أولا في الحالة الخاصة n = 1 أي أن A منظومة أحادية البعد 1 D.
- ثانيا في الحالة الخاصة n = 2 أي أن A منظومة ثنائية البعد 2 D.
- ثالثا في الحالة الخاصة n = 3 أي أن A منظومة ثلاثية البعد 3 D.
- رابعا في الحالة العامة ، حيث A منظومة نونية البعد n D .

أولا : ١ = n أي أن A منظومة أحادية البعد 1D

أي أن المنظومة A معرفة كما يلي :

element_type A [u_0];

نفرض أن العنصر A[0] مخزون عند العنوان α



من الواضح أن عنوان تخزين العنصر A[i] هو :

$\alpha + i$

وحيز التخزين الكلي المستخدم لتخزين جميع عناصر المنظومة A هو :

u_0 كلمة words

ثانيا : $n = 2$ أي أن A منظومة ثنائية البعد 2D

أي أن المنظومة A معرفة كما يلي :

element_type A [u_0] [u_1];

وكل عنصر (element) يتطلب تخزينه كلمة واحدة.

يتم تخزين المنظومة ثنائية البعد2D صفا صفا.

[وتسمى هذه الطريقة : التمثيل رئيسي الصف (row major representation)]

وبالتالي فالإعلان (declaration) السابق مكافئ للإعلان عن A كمنظومة من منظومات أحادية البعد ، أي كمنظومة من عناصر عددها u_1 بحيث أن أي عنصر من هذه العناصر عبارة عن منظومة أحادية البعد مكونة من عناصر عددها u_2 ، وكل عنصر (element) من هذه العناصر الأخيرة يتطلب تخزينه كلمة واحدة.

وبناء على هذا فإنه يمكننا استخدام الصيغة التي وصلنا إليها في حالة المنظومة أحادية البعد 1D.

مثلا : إذا فرضنا أن لدينا التعريف :

int A[4] [3];

فإنه يتم تخزين العناصر بالصورة التالية / بالترتيب التالي :

$A_{00}\ A_{01}\ A_{02}$	$A_{10}\ A_{11}\ A_{12}$	$A_{20}\ A_{21}\ A_{22}$	$A_{30}\ A_{31}\ A_{32}$
الصف رقم 0	الصف رقم 1	الصف رقم 2	الصف رقم 3
العنصر رقم 0	العنصر رقم 1	العنصر رقم 2	العنصر رقم 3

والآن لإيجاد عنوان العنصر $A_{i,j}$:

لاحظ أنه في هذا التمثيل المكافئ أحادي البعد 1D :

الطول β' المطلوب لتخزين عنصر واحد (والعنصر الواحد هنا هو منظومة (element_type Arr[u_1]) يساوي :

$$\beta' = u_1$$

وبالتالي فعنوان العنصر رقم i (والعنصر هنا عبارة عن صف) هو :

$$\alpha' = \alpha + i * \beta'$$
$$= \alpha + i * u_1$$

لاحظ أن هذا هو عنوان أول كلمة من الكلمات المحجوزة لتخزين هذا العنصر (الصف) رقم i .

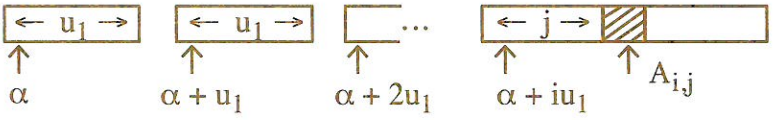
وعنوان (أو موضع) العنصر رقم j في هذا الصف (رقم i) [أي في هذه المنظومة أحادية البعد] أي عنوان العنصر $A_{i,j}$ هو (باستخدام نفس القانون العام للمنظومة أحادية البعد 1D) :

عنوان تخزين العنصر A[ij]:

$$\text{Address}(A_{ij}) = \alpha' + j$$

$$\boxed{\text{Address}(A_{ij}) = \alpha + i\,u_1 + j}$$

وهذه العلاقة يمكن توضيحها بالرسم التالي :



وحيز التخزين الكلي المستخدم لتخزين جميع عناصر المنظومة A هو :

Storage = $u_0\, u_1$ words

ثالثا : n = 3 أي أن A منظومة ثلاثية البعد 3D

نفرض أن المنظومة A معرفة كما يلي :

element_type A [u_0] [u_1] [u_2];

ويتم تخزين المصفوفة بطريقة مماثلة لطريقة التخزين صفا صفا. مثلا إذا كانت A معرفة كما يلي :

float A [3] [3] [2];

فإنه يتم تخزين عناصر A بالترتيب التالي :

A_{000} A_{001} A_{010} A_{011} A_{020} A_{021} A_{100} A_{101} A_{110} A_{111}
A_{120} A_{121} A_{200} A_{201} A_{210} A_{211} A_{220} A_{221}

لاحظ تزايد قيم المؤشرات كأعداد :

000, 001, 010, 011, 020, ... , 211, 220, 221

أي أن آخر مؤشر (last index) هو أسرع مؤشر في التغير ، أي أنه إذا رمزنا للعنصر بالرمز A_{ijk} ، فإن المؤشر k هو أسرع مؤشر في التغير ، يليه المؤشر j ثم المؤشر i.

والآن يمكننا أن ننظر إلى المنظومة ثلاثية البعد المعرفة سابقا على أنها تكافئ منظومة أحادية البعد 1D عنوان العنصر A_i فيها هو :

$$\alpha' = \alpha + i\,\beta'$$

حيث β' هو عدد الكلمات في المنظومة ثنائية البعد 2 D :

(element_type Arr $[u_1]$ $[u_2]$) ;

أي أن :

$$\beta' = u_1\, u_2$$

أي أن :

$$\alpha' = \alpha + i\, u_1\, u_2$$

وموضع العنصر رقم jk في المنظومة ثنائية البعد 2D المشار إليها نحصل عليه بتطبيق الصيغة التي وصلنا إليها في الحالة السابقة (ثانيا : n = 2) ، فنجد أنه :

$$\alpha' + j\, u_2 + k$$

وبالتالي يكون عنوان A_{ijk} هو :

$$\text{Address}\,(A_{ijk}) = \alpha + i\, u_1\, u_2 + j\, u_2 + k$$

وحيز التخزين الكلي المستخدم لتخزين جميع عناصر المنظومة A ثلاثية البعد 3D هو :

$$u_0\, u_1\, u_2$$

رابعا : الحالة العامة : A منظومة نونية البعد nD

نفرض أن المنظومة A معرفة كما يلي :

element_type A $[u_0]$ $[u_1]$ $[u_2]$ . . . $[u_{n-1}]$;

ونفرض أيضا أن كل عنصر يشغل حيزا يساوي كلمة واحدة ، وأن العنوان الأساسي (base) ، أي عنوان أول عنصر هو α.

يمكننا أن ننظر إلى المنظومة A نونية البعد nD على أنها منظومة أحادية البعد كل من عناصرها منظومة بعدها (n-1) ، والمطلوب الحصول على عنوان العنصر

$$A\,[i_0]\,[i_1]\,.\,.\,.\,[i_{n-1}]$$

نعلم أن عنوان Ai_0 هو :

$$\alpha' + \alpha + i_0 \prod_{k=1}^{n-1} u_k$$

وعنوان العنصر $A\,[i_0]\,[i_1]\,.\,.\,.\,[i_{n-1}]$ هو :

$\alpha' + [\{(n-1)\ \text{dim case}\}$ صيغة العنوان من الحالة قبل النونية مباشرة]

ويمكن (بالاستنتاج / الاستقراء الرياضي math. induction) إثبات أن الصيغة العامة لهذا العنوان هي :

$$\text{Address}\,(A[i_0]\,[i_1]\,.\,.\,.\,[i_{n-1}]) =$$

$$= \alpha + i_0 \prod_{k=1}^{n-1} u_k$$

$$+ i_1 \prod_{k=2}^{n-1} u_k$$

$$+ i_2 \prod_{k=3}^{n-1} u_k$$

$$\vdots$$

$$+ i_{n-1}$$

$$\text{Address}(A[i_0][i_1]\ldots[i_{n-1}]) =$$

$$= \alpha + \sum_{j=0}^{n-1} i_j \gamma_j \qquad (*)$$

حيث

$$\gamma_j = \prod_{k=j+1}^{n-1} u_k \; ; \qquad j = 0, 1, \ldots, n-2 \; ;$$

$$\gamma_{n-1} = 1$$

والسعة المكانية المطلوبة لتخزين جميع عناصر المنظومة العامة نونية البعد nD هي :

$$\text{Storage} = \prod_{k=0}^{n-1} u_k$$

ويمكن إثبات أننا نحتاج إلى عدد O(n) من عمليات الضرب لحساب عنوان عنصر ، وذلك لأن الثوابت γ_{n-1} ، γ_{n-2} ، ... ، γ_1 ، γ_0 يمكن حسابها مرة واحدة (وواحدة فقط) وتخزينها ، وواضح أن العلاقة (*) لحساب عنوان عنصر تؤدي إلى درجة تعقيد تساوي O(n).

مثال ٢-١٠ :

أوجد تعبيرا يعطي عنوان العنصر A[i][j] في منظومة A ثنائية البعد معرفة كما يلي :

```
int A[5] [4]
```

بفرض أن كل عنصر من A يحتل كلمة واحدة من كلمات الذاكرة ، وأننا نقوم بتخزين A صفا صفا ، وأن عنوان العنصر A[0] [0] هو 1.

الحل :

$$\alpha = 1, \qquad u_0 = 5, \qquad u_1 = 4$$

$$\text{Address}(A[i][j]) = \alpha + i\,u_1 + j$$

$$= 1 + 4\,i + j$$

Arrays 2

تمرينات رقم ٢

٢-١ نفرض أن لدينا القائمتين المرتبتين

$$A = (a_0 , a_1 , \ldots , a_i , \ldots , a_{n-1}) ,$$

$$B = (b_0 , b_1 , ... , b_j , ... , b_{m-1}) ,$$

يقال إن $A < B$ إذا تحقق أي من الشرطين التاليين :

i) إذا كانت $a_i = b_i$ لجميع قيم i في المدى $0 \le i < j$ ، وكانت $a_j < b_j$.

ii) إذا كانت $a_i = b_i$ لجميع قيم i في المدى $0 \le i \le n$ ، وكانت $n < m$.

اكتب دالة تعيد القيم الثلاث 1+ , 0 , 1- بناء على تحقق أحد الشروط الثلاثة $A < B , A = B , A > B$ على الترتيب.

افترض أنه يمكن مقارنة العناصر / الذرات (atoms) a_i , b_i.

٢-٢ نفرض أننا سنستخدم الدالة poly_add لجمع الحدوديتين

$$A(x) = x^{2n} + x^{2n-2} + \ldots + x^2 + x^0 ,$$

$$B(x) = x^{2n+1} + x^{2n-1} + \ldots + x^3 + x .$$

احسب عدد مرات تنفيذ كل عبارة من عبارات هذا الدالة.

٢-٣ نفرض أننا سنقوم بتمثيل أي حدودية $P(x)$ باستخدام منظومة سجلات للحدود غير الصفرية بترتيب تنازلي تبعا للقوى / للأسس (exponents / powers).

أ) اكتب تعريفا مناسبا لتمثيل هذه الحدوديات.

ب) نفرض أن P(x) حدودية متناثرة من النوع المذكور في أ) ، وأن عدد حدودها غير الصفرية يساوي n. اكتب دالة

void multiply_by_const (P, n, c)

تضرب الحدودية P(x) بعدد ثابت c من النوع float لا يساوي صفرا ، على أن تستبدل الدالة بالحدودية P(x) الحدودية الناتجة cP(x) ، أي تضع الحدودية cP(x) محل الحدودية P(x) .

ج) اكتب دالة

void poly_differentiation (P، n)

(حيث n هي عدد الحدود غير الصفرية في الحدودية P) تستبدل بالحدودية P(x) مشتقتها $\frac{dP}{dx}$ (أي تضع المشتقة $\frac{dP}{dx}$ محل الحدودية P(x)) .

د) كم تساوي درجة تعقيد الخوارزمية / الدالة في ج) ؟

٢-٤ اكتب التمثيل المتناثر للمصفوفة A التالية (أي باستخدام منظومة سجلات ثلاثية للعناصر غير الصفرية) مع منظومة مؤشرات (index array) لبدايات صفوف A.

$$A = \begin{bmatrix} 0 & 1.1 & 0 & 1.2 & 0 \\ 2.1 & 0 & 0 & 2.2 & 2.3 \\ 3.1 & 3.2 & 0 & 0 & 0 \\ 0 & 0 & 4.1 & 0 & 0 \\ 0 & 5.1 & 5.2 & 0 & 0 \end{bmatrix}$$

٢-٥ أ) اكتب التمثيل المتناثر مع منظومة مؤشرات لبدايات الصفوف وذلك للمصفوفة المتناثرة

$$A = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ 2 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{bmatrix}$$

ب) اكتب نتائج تتبع تنفيذ كل من الطرق الثلاث التالية لتدوير المصفوفة المتناثرة A المذكورة في (أ) وإيجاد مدورها $B= A^T$.

i) الطريقة المباشرة (الأسلوب القسري) [مع بيان الإزاحات اللازمة].

ii) طريقة نقل أعمدة A بالترتيب إلى صفوف B.

iii) طريقة التدوير السريع [مع بيان العناصر المخزونة في كل من المنظومتين S,U].

٢-٦ نفرض أن A, B مصفوفتان متناثرتان ممثلتان بمنظومتي الثلاثيات التاليتين :

	row	col	value
a_0	1	1	-2
a_1	1	4	1
a_2	4	1	-1
a_3	4	4	4
a_4	6	69	4

	row	col	value
b_0	1	1	3
b_1	4	1	6
b_2	4	4	2
b_3	69	59	3

اكتب التمثيل المتناثر لمصفوفة C تعطى بالعلاقة C = A.B (أي أن C هي المصفوفة الناتجة من ضرب المصفوفتين A , B) .

٢-٧ نفرض أن a مصفوفة متناثرة m x n عدد عناصرها غير الصفرية يساوي t. ونفرض أنه يتم تخزينa عن طريق استخدام منظومة للعناصر غير الصفرية مع منظومة مؤشرات إلى مواضع بدايات صفوفها.

أ) اكتب خوارزمية لطباعة عناصر الصف رقم i ، واحسب درجة تعقيد الخوارزمية.

ب) اكتب خوارزمية لطباعة عناصر العمود رقم j ، واحسب درجة تعقيد الخوارزمية.

ج) اكتب خوارزمية لطباعة قيمة العنصر رقم (i , j) ، واحسب درجة تعقيد الخوارزمية.

2

٢-٨ نفرض أن A مصفوفة n x n فيها خمسة عناصر غير صفرية في كل صف من صفوفها. أي الطريقتين التاليتين أفضل من حيث سعة التخزين المستخدمة مع بيان السبب ؟

i) طريقة تخزين العناصر غير الصفرية فقط باستخدام التمثيل المتناثر مع منظومة المؤشرات.

ii) طريقة تخزين جميع عناصر المصفوفة باستخدام التمثيل المعتاد بمنظومة ثنائية البعد 2D.

٢-٩ نفرض أن لدينا الإعلانات التالية عن المنظومات متعددة الأبعاد A,B,C (multidimensional arrays).

```
int A[3] [4] ;        int B[5] [21] ;        int C[8] [11] [16]
```

أ) كم عدد عناصر كل من المنظومات A,B,C ؟

ب) نفرض أنه سيتم تخزين عناصر هذه المنظومات في الذاكرة صفا صفا أي بالترتيب رئيسي الصف (row major order) ، كما يوضح ذلك الشكل التالي بالنسبة للمنظومة A.

الترتيب رئيسي الصف للمنظومة A

ونفرض أن عدد الكلمات اللازمة / المطلوبة لكل خلية / موضع / عنصر في الذاكرة (number of words per memory cell) لأي من المنظومات الثلاث هو

$\beta = 1$ word / memory location

i) ما هو عنوان العنصر A[2][2] إذا علمنا أن عنوان أول عنصر A[0][0] في المنظومة A هو Base (A) = 400 ≡ α ؟

ii) ما هو عنوان العنصر B[2][2] إذا علمنا أن عنوان أول عنصر في المنظومة B هو Base (B) = 600 ؟

iii) كم تساوي السعة المكانية المطلوبة لتخزين جميع عناصر المنظومة C ؟

٢-١٠ نفرض أن لدينا منظومة A ثنائية البعد معرفة كما يلي :

```
int A[3][4]
```

وأن كل عنصر من A يحتل كلمة واحدة من كلمات الذاكرة ، وأننا سنقوم بتخزين المنظومة / المصفوفة A صفا صفا [أي نستخدم التمثيل رئيسي الصف (row major representation)] ، وأن عنوان أول عنصر من المنظومة هو 314.

أ) كم يساوي حيز التخزين الكلي المستخدم لتخزين جميع عناصر المنظومة A ؟

ب) أوجد صيغة عامة لحساب عنوان العنصر A[i][j] في الصورة :

$$Address(A[i][j]) = a*i + b*j + c$$

حيث a , b , c ثوابت (أعداد صحيحة).

٢-١١ نفرض أننا سنستخدم التمثيل المعتاد لمصفوفة متناثرة بمنظومة ثلاثيات (array of triples) بالاستعانة بتعريف النوع التالي :

```
struct element
{
        int row;
```

```
        int col;
        float value;
}
element A [100] ;
```

اكتب دالة

void DisplayColumn3 (element A[ ], int m, int n, int t)

لعرض / لكتابة جميع عناصر العمود الثالث في المصفوفة A (بما في ذلك العناصر الصفرية) على الشاشة ، حيث تتكون المصفوفة من m صف و n عمود وعدد عناصرها غير الصفرية يساوي t . [افرض أن المصفوفة تحتوي على الأقل على ثلاثة أعمدة ، وأنه يمكن عرض عمود كامل من المصفوفة على سطر واحد من الشاشة].

Arrays 2

الفصل الثالث

الرصات والطوابير
Stacks and Queues

3

Stacks and Queues 3

الفصل الثالث

الرصات والطوابير

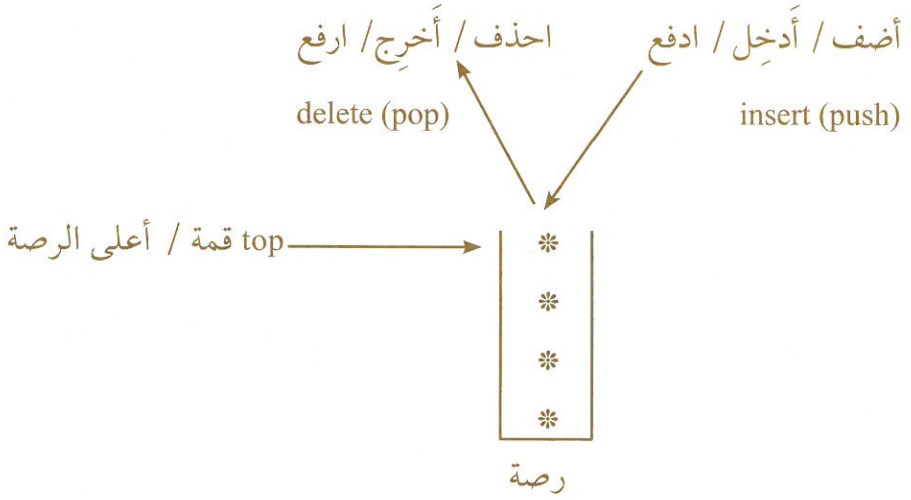
Stacks and Queues

الرصات والطوابير عبارة عن قوائم خطية (linear lists) ذوات شروط خاصة على عمليات الإدخال / الإضافة (insertion) و الحذف (deletion).

أولا : الرصات

Stacks

تعريف : الرصة (Stack) هي قائمة خطية تتم فيها عمليات الإدخال والحذف عند نهاية واحدة ، تسمى قمة الرصة / أعلى الرصة (top of the stack).



وتعرف الرصة بأنها بنية **LIFO** (Last In First Out structure type) أو **FILO** (First In Last Out).

العمليات / الدوال القياسية على الرصات :

فيما يلي ست عمليات قياسية تُجرى عادة على الرصات ، ووظيفة كل منها :

(١) دالة خاوية **create_s(S)** : إنشاء رصة خالية.

(٢) دالة خاوية **push (S,e)** : دفع / إدخال عنصر e في (أعلى) الرصة S بشرط ألا تكون الرصة ممتلئة (full).

(٣) دالة خاوية **pop (S,e)** : رفع / إخراج / حذف عنصر من (أعلى) الرصة S وإعادة قيمته e ، بشرط ألا تكون الرصة خالية (empty).

أو دالة تعيد قيمة **pop (S)** : رفع / إخراج / حذف عنصر من (أعلى) الرصة ، وإعادة قيمته pop بشرط ألا تكون الرصة خالية.

(٤) دالة خاوية **top_s(S,e)** : إعادة قيمة العنصر (e) الموجود أعلى الرصة S (دون حذفه) ، بشرط ألا تكون الرصة خالية.

أو دالة تعيد قيمة **top_s(S)** : إعادة قيمة العنصر الموجود أعلى الرصة S (دون حذفه) ، بشرط ألا تكون الرصة خالية.

(٥) دالة منطقية **empty_s(S)** : تعيد القيمة true إذا كانت الرصة S خالية وتعيد القيمة false إذا كانت الرصة S غير خالية (سواء ممتلئة أو غير ممتلئة).

(٦) دالة منطقية **full_s (S)** : تعيد القيمة true إذا كانت الرصة S ممتلئة ، وتعيد القيمة false إذا كانت الرصة S غير ممتلئة (سواء خالية أو غير خالية). أي تعيد القيمة true إن لم يكن بإمكاننا إضافة أي عنصر.

ملاحظة :

في كل من العمليتين (٢) ، (٣) (**push, pop**) يمكننا إضافة متغير

منطقي (boolean) للدلالة على نجاح العملية المقابلة (الإدخال / الإخراج) أو عدم نجاحها أي للدلالة على تحقق أو عدم تحقق الشرط المذكور في العملية المقابلة.

مثال ٣-١ :

باستخدام نوع البيانات المجردة الخاص بالرصة (stack ADT) ، وبفرض أن الرصة هي رصة رموز (stack of char) ، اكتب برنامجا لقراءة سطر من الرموز ثم عكس (reversing) سلسلة الرموز المدخلة (input string of characters). مثلا :

إذا كانت المدخلات هي : A B C D E input:

فإن المخرجات تكون : E D C B A output:

الحل :

```
int main ( )
// Reversing an input string of characters.
{
        stack_type s;
        char ch;
        create_s (S);
        while (!'\n' && ! full_s (S))
                {
                        cin >> ch;
                        push (S, ch);
                }
        while  (! empty_s (S))
```

E
D
C
B
A

```
        cout << pop (S);
        cout << endl;
        return 0;
}
```

ملاحظة :

العملية **pop** المستخدمة في هذا الحل عبارة عن دالة تعيد قيمة ، وليست دالة خاوية.

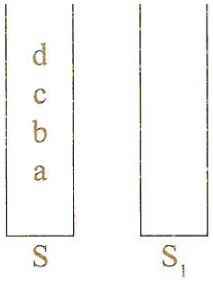
مثال ٣-٢ :

اكتب دالة لإيجاد حجم رصة معطاة (أي إيجاد عدد عناصرها).

الحل :

سنقوم بإنشاء رصة أخرى خالية (مؤقتة) S_1 ، ثم نخرج العناصر من الرصة المعطاة S عنصرا عنصرا ونضعها في الرصة S_1 مع عدها. وبعد الانتهاء من ذلك نعيد العناصر إلى الرصة الأصلية بإخراجها عنصرا عنصرا من S_1 ووضعها ثانية في S.

```
int size_s (stack S)
{
        stack S1;
        int count;
        element_type e;

        create_s (S1);
        // copy S to S1 and count
        count = 0;
        while  ! empty_s (S)
            {
        e = pop (S);
        push (S1, e);
        count ++;
```



```
        }

    // copy S1 back to S
    while ! empty_s (S1)
        {
    e = pop (S1);
    push (S, e);

        }

    size_s = count;
}
```


ملاحظتان :

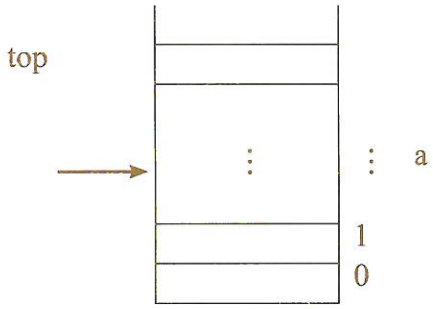
١) هذا التنفيذ (implementation) لا يعتمد على تمثيل معين للرصة (representation independent) ، حيث أننا استخدمنا فقط تعريف الرصة ذات العمليات الست (القياسية) (6 operations stack) .

٢) إذا كانت هذه الدالة سيتم استدعاؤها مرات عديدة فقد يكون من الأفضل أن نضيف هذه الدالة إلى العمليات الست السابقة لتحسين الكفاءة ، أي أنها يجب أن تكون جزءا من مواصفات الرصة (stack specifications).

تنفيذ الرصات (implementation of Stacks)

بِنْيَةُ التخزين Storage Structure

سنستخدم منظومة (array) لتخزين عناصر الرصة مع مؤشر صحيح (integer pointer) يشير إلى أعلى الرصة (أي أنه عدد صحيح عبارة عن ترتيب العنصر الموجود أعلى الرصة) ونجمعهما (المنظومة والمؤشر) في سجل (record) يعرَّف بالنوع التالي :




```
const int maxsize = 100;
struct stack
{
        element_type a[maxsize];
        int top;
};
stack s;
```

ملاحظتان :

١) العنصر الأول (first element) أي العنصر عند قاع الرصة (bottom element) يتم تخزينه في a[0] ، والعنصر الثاني في a[1] ، ... وهكذا.

٢) ابتداءً top = 0 لتعني أن الرصة خالية.

وفيما يلي نكتب العمليات / الدوال التي ذكرنا أسماءها ووظائفها سابقا ، مستخدمين تعريف النوع السجلي **stack**.

مثال ٣-٣ :

باستخدام النوع السجلي **stack** المعرف سابقا اكتب الدوال المقابلة للعمليات الست القياسية على الرصات والمعرفة سابقا.

الحل :

```
1)    void create_s (stack& s)
      {
```

```
  s . top = -1;
}
                              *  *  *
2)  void push (stack& S, element_type e, bool& success)
{
  if full_s (S)
        {
              success = false;
              cout << " stack is full " << endl;
        }
  else
        {
              success = true;
              S.top = S.top + 1;
              S.a[S.top] = e;
        }
}
                              *  *  *
3.a)  void pop (stack& S, element_type& e)
{
  if ! empty_s (S)
        {
              e = a[S.top];
              S.top = S.top - 1;
        }
        else
              cout << " stack is empty " << endl;
  }
                              *  *  *
3.b)  element_type pop (stack S)
{
  if ! empty_s (S)
        {
              pop = S.a[S.top];
```

```
                S.top = S.top - 1;
        }
        else
                cout << " stack is empty " << endl;
}
                        * * *
4)  element_type top_s (stack S)
{
        if empty_s (S)
                cout << " error " << endl;
        else
                typ_s = S.a[S.top];
}
                        * * *
5)  bool empty_s (stack S)
{
        empty_s = (S.top = = -1);
}
                        * * *
6)  bool full_s (stack S)
{
        full_s = (S.top = = maxsize-1);
}
                        * * *
```

نلاحظ أن التمثيل المذكور للرصات بسيط وذو كفاءة عالية حيث تتطلب كل عملية من العمليات المذكورة فترة زمنية محدودة / ثابتة (const amount of time) ، أي O(1) . إلا أنه من المشاكل الخطيرة التي قد تواجهنا مع هذا التمثيل احتمال نفاذ حيز التخزين المتاح أمامنا (run out of storage).

كما أنه من أهم مميزات استخدام السجل (record) في تنفيذ الرصات تلك التي تسمى «إخفاء المعلومات» (information hiding) ، حيث يُسمح للمستخدم أن يستخدم ويرى فقط ما يهمه وما هو ضروري بالنسبة له ، وهو هنا الرصة كَبِنية

كُلِّية (stack as a whole) . كما أن ذلك أيضا يسمح للمبرمج بتغيير وتعديل التنفيذ دون أن يؤثر ذلك على تسلسل الاستدعاءات (calling sequence).

من تطبيقات الرصات
Applications of Stacks

إيجاد قيم التعابير الحسابية في الحاسوب
Evaluation of arithmetic expressions in computer

تمهيد

من أهم التطبيقات التي تستخدم فيها الرصات حساب قيم التعابير الحسابية ، وذلك عن طريق تحويل صيغة التعبير أولا من الصيغة المعتادة والتي تسمى التمثيل الوسطي (infix notation) [لأن المؤثِّر / المعامِل (operator) يظهر وسط أي بين (in-between) المعامَلين (operands) مثل a + b] إلى الصيغة المسماة التمثيل اللاحق (postfix notation) [حيث يظهر المؤثر لاحقا بعد المعامَلين مثل ab +] ، ثم يسهل بعد ذلك – كما سنرى بإذن الله – حساب قيمة التعبير وهو في هذه الصيغة الجديدة لله التمثيل اللاحق لله . ويرجع السبب الرئيسي في هذه السهولة أولا إلى اختفاء الأقواس (parentheses) في هذه الصيغة الجديدة وعدم الحاجة إلى أقواس عند حساب قيمة التعبير في هذه الصيغة ، وثانيا إلى عدم الحاجة إلى تطبيق قاعدة الأولويات عند حساب القيمة ، حيث يتم الحساب بمجرد اجتياز أو المرور على (scanning) التعبير من اليسار إلى اليمين رمزا رمزا فإن كان الرمز معامَلا – أي كمية مؤثَّر عليها – وضعناه في الرصة وإن كان مؤثِّرا / معامِلا حَسَبنا القيمة المطلوبة باستخدام العدد المضبوط المقابل من المعامَلات الموجودة بالرصة ، وأخيرا نضع النتيجة في الرصة ، كما سيتضح بإذن الله بعد قليل من خوارزمية حساب قيمة تعبير.

ويلاحظ أننا نستخدم رصات في كل من عملية تحويل التعبير من التمثيل الوسطي إلى التمثيل اللاحق ، وعملية حساب قيمة التعبير في التمثيل اللاحق. وهكذا :

من التطبيقات التي تستخدم فيها الرصات :

(١) حساب قيمة (evaluation) تعبير مكتوب بالاصطلاح الذي يطلق عليه الاصطلاح / الترميز / التمثيل الرمزي اللاحق / المؤخَّر / المعكوس (postfix notation) ، والذي يطلق عليه أيضا الاصطلاح البولندي المعكوس RPN (Reverse Polish Notation).

(٢) تحويل (conversion) تعبير مكتوب بالاصطلاح الذي يطلق عليه الاصطلاح / الترميز / التمثيل الوسطي (infix notation) إلى تعبير مكتوب بالاصطلاح الرمزي اللاحق (postfix).

(٣) توليد شفرة / عبارات برنامج بلغة التجميع للتعابير المكتوبة بالاصطلاح البولندي (Generation of assembly language code for an RPN).

أنواع التمثيل الرمزي للتعبير الحسابي

هناك ثلاث طرق مختلفة لكتابة تعبير حسابي ويطلق عليها : التمثيل الرمزي الوسطي ، والتمثيل الرمزي اللاحق ، والتمثيل الرمزي السابق. وفيما يلي معنى كل منها :

١) التمثيل الرمزي الوسطي (infix notation) :
وفيه يكون المؤثر (operator) الحسابي بين المعامَلين (operands) كالمعتاد في الرياضيات ، مثل التعبير a + b.

٢) التمثيل الرمزي اللاحق / المؤخَّر / المعكوس (postfix notation) :

وفيه يأتي المؤثر بعد المعاملين ، فمثلا التعبير السابق يكتب هكذا : a b +

٣) التمثيل الرمزي السابق / المقدَّم / البادئ (prefix notation) :

وفيه يأتي المؤثر قبل المعاملين ، فمثلا التعبير السابق يكتب هكذا : + a b

وفيما يلي أمثلة لتوضيح الفارق بين التمثيلين الرمزي الوسطى والرمزي اللاحق.

مثال ٣-٥ :

فيما يلي مجموعة من التعابير الحسابية المكتوبة بالتمثيل الرمزي الوسطى وما يقابل كلا منها بالتمثيل الرمزي اللاحق

التمثيل الرمزي الوسطي Infix	التمثيل الرمزي اللاحق Postfix / RPN
a + b * c	a b c * +
(a + b) * c	a b + c *
a * (b + c)	a b c + *
a * (b + c) * d	a b c + * d *
(a + b) * (c – (d + e))	a b + c d e + - *

وفيما يلي نعطي خوارزميتين : الأولى لحساب قيمة تعبير مكتوب بالتمثيل اللاحق ، والثانية لتحويل تعبير من التمثيل الوسطي إلى التمثيل اللاحق.

(١) خوارزمية حساب قيمة تعبير مكتوب بالاصطلاح الرمزي اللاحق

An Algorithm to evaluate a postfix / RPN expression

تمهيد : نعطي أولا المثال التالي ثم نتناول هذه الخوارزمية.

مثال ٣-٦ :

وضح كيفية حساب قيمة التعبير التالي المكتوب بالتمثيل الرمزي اللاحق.

a b + c d e + - *

واكتب هذا التعبير بالتمثيل الرمزي الوسطي.

الحل :

نتجه من اليسار لليمين وكلما وجدنا مؤثرا (operator) حَسَبنا قيمة التعبير الناتج من التأثير بهذا المؤثر (الثنائي) (هذه العملية) على الرمزين / المتغيرين / المعامَلين (operands) السابقين له (مباشرة) ، كما توضح ذلك الخطوات التالية :

a b + c d e + - *

T_1

$T_1 = a + b$

T_1 c d e + - *

T_2

$T_2 = d + e$

T_1 c T_2 - *

T_3

$T_3 = c - T_2$

$= c - (d + e)$

T_1 T_3 *

T_4

$T_4 = T_1 * T_3$

$= (a + b) * (c - (d + e))$

أي أن التعبير المطلوب بالتمثيل الرمزي الوسطي هو : (a + b) * (c – (d + e))

وفيما يلي نكتب الخوارزمية المطلوبة مستخدمين رصة للاحتفاظ بالمعامَلات (المتغيرات) (using a stack to hold operands).

وتتلخص الخوارزمية فيما يلي :

خوارزمية حساب قيمة تعبير مكتوب بالتمثيل الرمزي اللاحق.

(١) أنشئ رصة خالية بحيث تكون من نوع المعاملات (operands) .

(٢) استمر في قراءة التعبير المعطى رمزا رمزا (عنصرا عنصرا) إلى أن ينتهي ، وإذا كان العنصر معامَلاً (operand) فضعه في الرصة ، أما إن كان مؤثرا (operator) فأخرج من الرصة معامَلا e_2 ومعامَلاً آخر e_1 وأجرِ العملية الحسابية e_1 operator e_2 وضع النتيجة في الرصة.

عند انتهاء الخوارزمية تكون النتيجة النهائية للتعبير هي المعامل الوحيد المتبقي في الرصة.

ويمكننا كتابة هذه الخوارزمية بالأسلوب التالي :

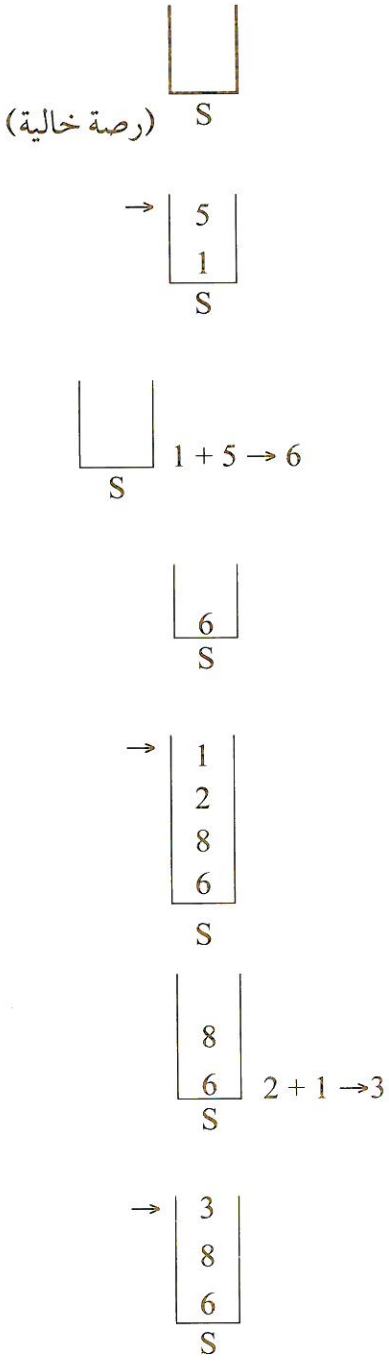
```
// Algorithm to evaluate a postfix expression ;

1.  create_s (s) ;
2.  while not end of postfix expression do
    {
        - get next token from expression ;
        - if the token is an operand then
          push (s , token)
          else {token is an operator}
                        {
                            pop (s , e2) ;
                            pop (s , e1) ;
                            T = e1 token e2 ; // evaluate T
                            push (s , T)
                        }
    }
```

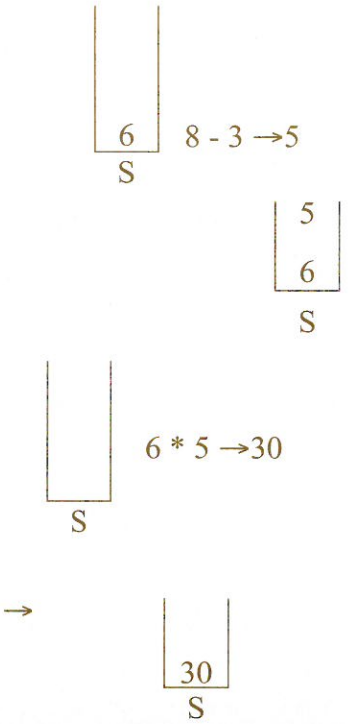
مثال ٣-٧ :

تتبع تنفيذ الخوارزمية السابقة لإيجاد قيمة التعبير التالي المكتوب بالتمثيل الرمزي اللاحق ، وبيِّن محتويات الرصة من بداية إلى نهاية تنفيذ الخوارزمية.

1 5 + 8 2 1 + - *

الحل :





النتيجة : قيمة التعبير المعطى = value =30

ملاحظة : يمكن التأكد من صحة النتيجة بحساب قيمة التعبير المعطى :

value = (1 + 5) * (8 – (2 + 1))

= 6 * (8 – 3) = 6 * 5 = 30

(2) خوارزمية تحويل تعبير من التمثيل الوسطى إلى التمثيل اللاحق

An Algorithm to convert an infix to postfix expression

لكتابة خوارزمية تقوم بهذا التحويل نفترض ما يلي :

- أن المدخلات (التعابير) صحيحة (valid input).
- أن المدخلات تشتمل على :
  - معامَلات (كميات مشغَّلة) (operands) : عبارة عن ثوابت ومتغيرات
  - مؤثرات ثنائية (binary operators) : عبارة عن :

/ , × : لها أولوية عليا high priority

- +, - : لها أولوية سفلى low priority
- قد تشتمل على أقواس وتعطى أولوية لعملية على أخرى.

قاعدة الأولويات (Priority Rule) :

عموما يمكننا تلخيص قاعدة الأولويات في السطور التالية ، حيث العمليات في أي سطر (علوي) لها أولوية قبل العمليات في أي سطر تالي (سفلي) ، والعمليات الأحادية (unary) تأتي في المقدمة من حيث الأولوية (أي لها الأولوية الأولى).

unary - , +

!

* , / , % , & &

+ , - , | | , xor

< , <= , = =, != , > , in

القوس) الذي داخل الرصة

والآن نكتب الخوارزمية المطلوبة لتحويل تعبير من التمثيل الوسطي إلى التمثيل اللاحق.

خوارزمية تحويل تعبير من التمثيل الرمزي الوسطي Infix إلى التمثيل الرمزي اللاحق Postfix

١- أنشئ رصة خالية.

٢- خذ التعبير رمزا رمزا من اليسار لليمين بالترتيب إلى أن ينتهي :
إن كان الرمز (token) معامَلا (operand) (أي كمية مؤثَّر عليها أو متغيرا) فاكتبه في المخرجات مباشرة.
وإن كان الرمز مؤثِّرا (operator) فعلى حسب درجة أولويته :

- إن كانت الرصة فارغة فضعه في الرصة.
- وإن كانت أولويته > أولوية الرمز أعلى الرصة فضعه في الرصة.
- وإن كانت أولويته ≥ أولوية الرمز أعلى الرصة

فأخرج الرموز (رمزا رمزا) التي أولويتها ≤ أولويته ، واكتبها في المخرجات ثم ضع الرمز في الرصة.

وإن كان الرمز :) فضعه في الرصة.

وإن كان الرمز : (فاستمر في إخراج الرموز من الرصة وكتابتها في المخرجات إلى أن يصبح الرمز أعلى الرصة هو :) ثم أخرجه [ولا تكتبه في المخرجات].

٣- استمر في إخراج الرموز المتبقية في الرصة وكتابتها في المخرجات إلى أن تصبح الرصة فارغة.

ويمكننا كتابة هذه الخوارزمية بالأسلوب التالي :

```
// Algorithm to convert an expression from infix to postfix ;
1.    create_s (S) ;
2.    While not end of infix_expression do
      {
            get next token from infix_expression ;
            switch (token)
                  {
                  case operand:  cout << token; // display operand
                  case '(':  push (s, '(' );
                  case ')':  {
                              while (top_s(S) != '(' )
                              {
                                          pop (s, token);
                                          cout << token;
```

```
                    }
                    pop (s, token);      // pop '(' but
                                         // don't write it
                }
            case operator: {
                    while ( ! empty_s(S) &&
                            priority of top_s (S) >=
                            priority of current
                     operator)

                    {pop (s, token);
                    cout << token;
                    }
                     push(s, current operator);
                    }
            }
        }
3.      do
        {
            pop (s, token);
            cout << token;
        } while ! empty_s(S)
```

مثال ٣-٨ :

تتبع تنفيذ الخوارزمية السابقة لتحويل كل من التعابير التالية من التمثيل الرمزي الوسطي إلى التمثيل الرمزي اللاحق ، مع بيان محتويات الرصة في كل حالة

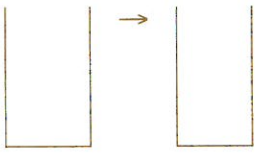
(a) a + b * c

(b) a * (b + c) * d

(c) (a + b) * (c – (d + e))

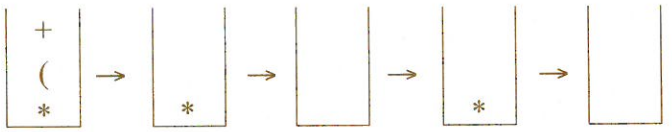
الحل :

(i) المدخلات: a + b * c



المخرجات : a b c * +

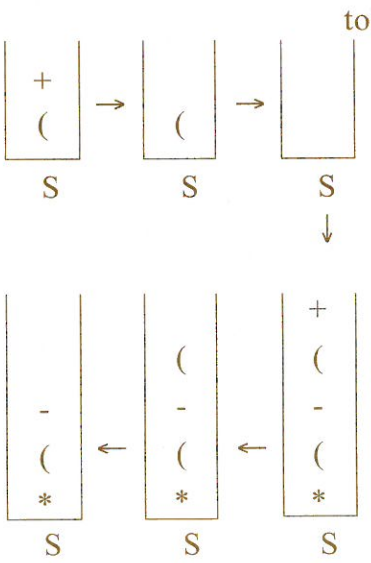
(ii) المدخلات : a * (b + c) * d

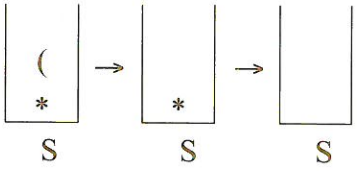


المخرجات : a b c + * d *

(iii) المدخلات : (a + b) * (c – (d + e))

المخرجات : a b + c d e + - *

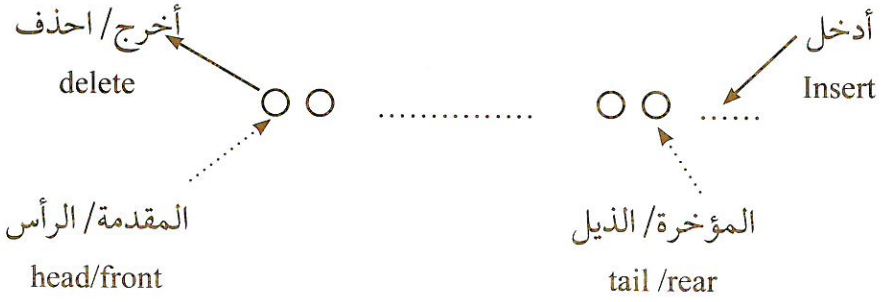




ثانيا : الطوابير

Queues

تعريف : الطابور هو قائمة خطية تتم فيها عمليات إدخال العناصر عند إحدى النهايتين ، وتسمى المؤخرة / الذيل (rear / tail) بينما يتم الإخراج / الحذف عند النهاية الأخرى ، وتسمى المقدمة / الرأس (front / head).



ويعرف الطابور بأنه بنية FIFO (First In First Out)

أو LILO (Last In Last Out)

العمليات القياسية على الطوابير :

١. دالة خاوية **create_q** : إنشاء طابور خال.

٢. دالة خاوية **insert (q , e)** : إدخال عنصر eفي نهاية طابور بشرط ألا يكون الطابور ممتلئا.

٣. دالة خاوية **del (q , e)** : حذف العنصر الأمامي (front element) من الطابور وإعادة قيمته e ، بشرط ألا يكون الطابور خاليا.
(أو دالة تعيد قيمة **del (q)**).

٤. دالة خاوية **front_of_queue (q , e)** : إعادة قيمة العنصر الأمامي من الطابور دون حذفه ، بشرط ألا يكون الطابور خاليا.
(أو دالة تعيد قيمة).

٥. دالة منطقية **empty_q (q)** : تعيد القيمة **true** إن كان الطابور خاليا وتعيد false ما عدا ذلك.

٦. دالة منطقية **full_q(q)** : تعيد القيمة **true** إن كان الطابور ممتلئا وتعيد القيمة **false** ما عدا ذلك.

التنفيذ (التتابعي) لطابور باستخدام منظومة

Array based (sequential) implementation of a queue

سنستخدم سجلا يشتمل على أربعة مجالات : المجال الأول عبارة عن منظومة (a) يتم فيها تخزين عناصر الطابور ، والمجالان الثاني والثالث : عددان صحيحان يشيران إلى موضعي العنصرين في مقدمة الطابور (front) ومؤخرته (rear) ، والمجال الرابع : عدد صحيح يمثل حجم المنظومة / الطابور (size) أي عدد عناصره.

```
const int maxsize = 100;
struct queue
{
        element_type a[maxsize];
        int front;
        int rear;
        int size;        // no. of elements
}
```

وفيما يلي نكتب دوال العمليات المذكورة أسماؤها سابقا.

مثال ٣-٤ :

باستخدام النوع السجلي (queue) المعرف سابقا اكتب الدوال التي تنفذ العمليات الست القياسية المعرفة سابقا على الطوابير.

الحل :

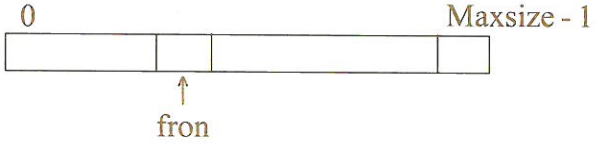
1)
```
void create_q (queue& q)
{
  q.size = 0;
  q.front = 0;
  q.rear = -1
}
```

* * *

2)
```
void insert (queue& q, element_type& e)
{
  if full_q (q)
        cout << "queue is full" << endl;
else
        {
            q.size = q.size + 1;
            q.rear = q.rear + 1;
```

```
                q.a[q.rear] = e;
            }
    }
                        *   *   *

3)  void del (queue& q, element_type& e)
    {
     if empty_q (q)
            cout << " queue is empty" << endl;
     else
            {
                q.size = q.size – 1;
                e = q.a[q.front];
                q.front = q.front + 1;
               }
    }
                        *   *   *
4)  void front_of_q (queue& q, element_type& e)
    {
            if empty_q (q)
                cout << "queue is empty" << endl;
            else
                e = q.a[q.front]
    }
                        *   *   *
5)  bool empty_q (queue& q)
    {
     empty_q = (q.size = = 0)
    }
                        *   *   *
```

```
6)   bool full_q (qeue& q)
     {
       full_q = (q.rear = = maxsize - 1) ;
     }
```



عند استخدام الدالة المكتوبة full_q قد يظهر لنا أن الطابور ممتلئ بينما هو في الحقيقة ليس كذلك (انظر الرسم أعلاه) – أي أن النتيجة يكون فيها تمويه (misleading) – ولكن الحقيقة أن معنى النتيجة (full_q = true) هي أننا لا نستطيع أن نضيف (add) أي عنصر دون أن نزحزح (shift) العناصر المخزونة في الطابور.

* * *

خصائص تنفيذ الرصات / الطوابير باستخدام المنظومات :

(Features of the array based implementation of a stack / queue)

يتميز استخدام المنظومات (لتنفيذ كل من الرصات والطوابير) بأنه يؤدي إلى أن تكون جميع العمليات بسيطة (simple) وذات كفاءة عالية (efficient) [حيث أن درجة تعقيد جميع العمليات تساوي O(1) ، أي لا تعتمد على حجم (طول) الرصة / الطابور] ، ما عدا بعض الحالات ، فمثلا بالنسبة للطوابير :

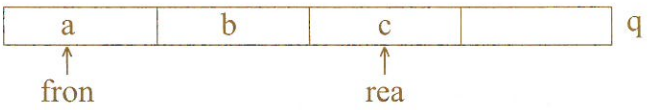
١) حجم الطابور (queue size)محدود ، وبالتالي فقد يصبح الطابور ممتلئا.

٢) قد يبدو الطابور ممتلئا بينما هو في الحقيقة ليس كذلك كما أشرنا سابقا ، فمثلا نفرض أن لدينا طابورا سعته الكلية (حجمه) : maxsize = 4

ونفرض أننا أجرينا العمليات الثلاث المتعاقبة

```
insert (q , a)
insert (q , b)
insert (q , c)
```

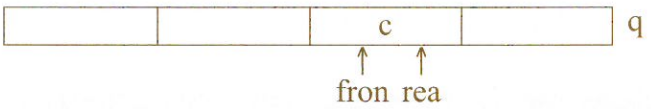
فيصبح الطابور بالشكل التالي :



ثم نفرض الآن أننا أجرينا عمليتي حذف

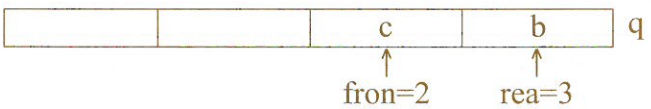
```
del (q,e) , del (q,e)
```

فيصبح الطابور هكذا :



والآن أضف / أدخل d : insert (q , d)

فيصبح الطابور هكذا :



الآن يبدو الطابور ممتلئا (حيث أن rear = 3 ≡ maxsize – 1) ، ولا نستطيع إضافة أي عنصر مثل insert(q , e) ، إلا إذا اتبعنا إحدى الطرق التالية :

الطريقة الأولى :

كلما حذفنا عنصرا أزحنا الطابور إلى اليسار بحيث يظل front = 0 دائما. إلا أن هذه الطريقة مكلفة (expensive) ، حيث أن التكاليف (cost) (أي درجة التعقيد) تساوي O(n) ، حيث n هي حجم / طول الطابور [حيث نزيح جميع عناصر الطابور – وعددها n – موضعا إلى اليسار].

الطريقة الثانية :

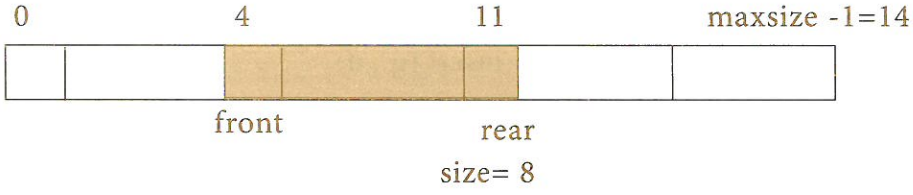
عندما نريد إدخال عنصر ويبدو الطابور ممتلئا ، نحاول إزاحة الطابور إلى اليسار بحيث يصبح front = 0 (وإذا كان front = 0 و rear = maxsize - 1 فإنه لا يمكننا الإزاحة). وهذه الطريقة أفضل من الطريقة الأولى ، إلا أنها أيضا تتطلب O(n) من التكاليف.

الطريقة الثالثة :

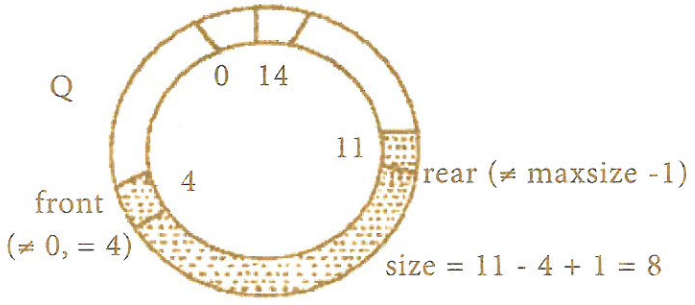
استخدام طابور دائري (circular queue) ، وفيما يلي نوضح معنى هذا الطابور.

الطابور الدائري (Circular Queue) :

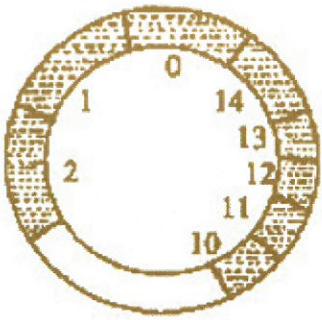
نفرض أن لدينا طابورا بالمواصفات المبينة في الشكل التالي :



يمكننا تصور هذا الطابور دائريا بالشكل التالي :



والآن يمكننا الحذف والإضافة ، وقد نحصل بعد عدد من هذه العمليات (٦ عمليات حذف و٦ عمليات إضافة) على الشكل التالي :



rear = 2
front = 10 (rear < front)
size = 8

من الملاحظ (والمفاجئ !) أنه في الطابور الدائري تبقى جميع العمليات المذكورة سابقا كما هي عدا ما يلي

2) void insert ...

:

:

.....

q.rear = q.rear % maxsize + 1

↓

بدلا من

q.rear = q.rear + 1

3) void del ...

.....

q.rear = q.rear % maxsize + 1

↓

بدلا من

q.front = q.front + 1

6) bool full_q .....

{

full_q = (q.size = = maxsize)

}

* * *

ويلاحظ أن التنفيذ الدائري لطابور له الميزتان الرئيسيتان التاليتان :

١) لا نحتاج معه إلى أي إزاحات للعناصر خلال عمليات الإضافة والحذف.

٢) جميع العمليات تصبح O(1) [أي أن درجة التعقيد تصبح O(1)] إلا أننا قد نصل إلى حالة ملء الطابور كله (وبالتالي لا يمكننا إضافة أي عناصر) ، وذلك لأن المنظومة array ساكنة/استاتيكية بالنسبة لمسألة الحجم (static in size) ، ويمكننا للتخلص من هذه المشكلة استخدام بنيات المؤشرات (pointer structures) والتي هي ديناميكية بالنسبة للحجم ، وهذا هو موضوع الفصل القادم.

3 الرصات والطوابير المتعددة

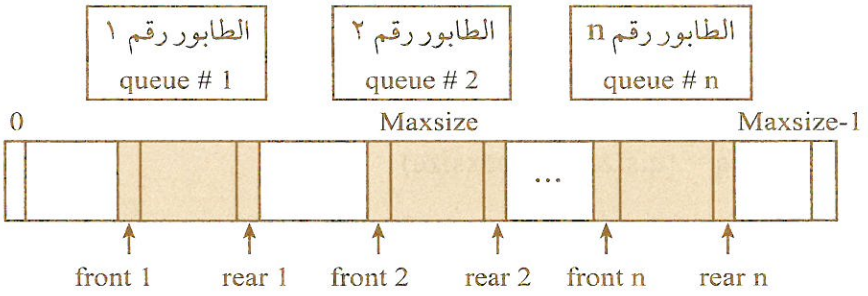
Multiple Stacks and Queues

قد يكون لدينا أكثر من رصة واحدة أو طابور واحد ، ويمكننا تنفيذ هذه الرصات المتعددة أو الطوابير المتعددة إما عن طريق التمثيل التتابعي (أي باستخدام المنظومات أحادية البعد) أو عن طريق التمثيل المترابط (أي باستخدام مؤشرات وقوائم مترابطة ، وهذا التمثيل سنتناوله بإذن الله في الفصل القادم).

التنفيذ باستخدام التمثيل التتابعي (منظومة أحادية البعد 1D array)

Implementation using sequential representation

نفرض أن لدينا n طابورا كما بالشكل التالي :



يمكننا استخدام منظومة لتمثيل هذه الطوابير كما هو موضح بالشكل ، وفي أي لحظة يجب تحقق الشروط التالية :

$$rear\ (i) < front\ (i + 1)\ ; \qquad i = 1\ ,\ 2\ ,\ \ldots\ ,\ n\text{-}1$$

(حيث n تمثل عدد الطوابير).

$$rear\ (n) \leq maxsize\text{-}1$$

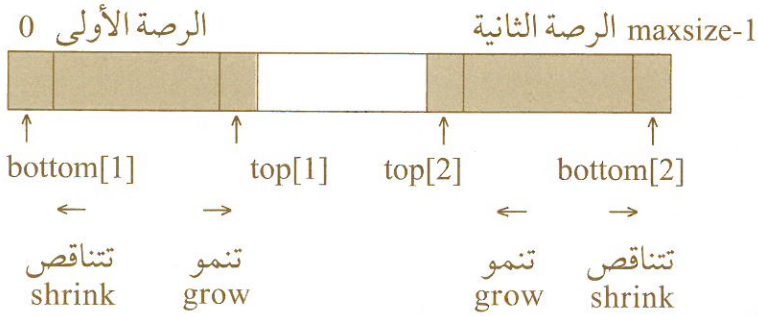
حيث maxsize تمثل أقصى حجم مسموح به للمنظومة المستخدمة (أي سعة المنظومة) .

$$\sum_{i=1}^{n} size\ (i) \leq maxsize$$

حيث size (i) هو حجم (أي طول / عدد عناصر) الطابور رقم i.

فإذا لم يتحقق أي من الشروط السابقة – وكثيرا ما يحدث هذا – فيجب إزاحة بعض الطوابير إلى اليمين (انظر الشكل) ، وهذا يؤدي إلى تقليل كفاءة الخوارزميات المستخدمة. ويمكن التغلب على هذه المشكلة باستخدام التمثيل المترابط كما سنرى في الفصل القادم بإذن الله.

وكمثال آخر للتمثيل التتابعي نفرض أن لدينا رصتين. يمكننا تمثيل الرصتين بمنظومة أحادية واحدة حيث تنمو إحدى الرصتين في اتجاه معين ، وتنمو الرصة الأخرى في الاتجاه المعاكس ، كما يوضح ذلك الشكل التالي :

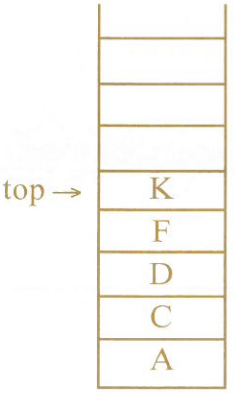


ويلاحظ أنه يجب تحقق الشرط top[1] < top[2] حتى يمكن تطبيق هذا التمثيل.

3 Stacks and Queues

تمرينات رقم ٣

٣-١ نفرض أن s رصة رموز (stack of characters) ممثلة بمنظومة مكونة من ثمانية عناصر / خلايا ذاكرة (8 memory cells) ، وتحتوي الرصة على القيم الخمس المبينة بالشكل (والخلايا الثلاث المتبقية خالية) .



المطلوب : بيان محتويات الرصة بعد تنفيذ كل من الاستدعاءات / العمليات المتعاقبة التالية :

1) pop (s , e) ;
2) pop (s , e) ;
3) push (s , L) ;
4) push (s , P) ;
5) pop (s , e) ;
6) push (s , R) ;
7) push (s , T) ;

٣-٢ نفرض أن s رصة أعداد صحيحة (stack of integers) ممثلة بمنظومة مكونة من ستة عناصر ، وأن الرصة خالية ابتداءً (أي أن top = -1) . المطلوب : تتبع تنفيذ الخوارزمية التالية وبيان مخرجاتها :

```
A = 2 ;          B = 5 ;
push (s , A) ;
push (s , 4) ;
push (s , B + 2) ;
push (s , 9) ;
push (s , A + B) ;
while (top ! = 0)
        {
                pop (s , item) ;
                cout << item << endl ;
        }
```

٣-٣ نفرض أن كلا من Y ، X متغير من نوع (type) العناصر المخزونة في رصة S ، ونفرض أنه قد أُعطيت لكل من Y ، X قيمة ابتدائية . ما تأثير العمليات المتتابعة التالية ؟

```
push (S , X) ;
push (S , Y) ;
pop (S , X) ;
pop (S , Y) ;
```

٣-٤ نفرض أن S رصة سلاسل رموز (stack of strings) وأن V, W, T متغيرات سلاسل رموز (string variables). تتبع تنفيذ الخوارزمية التالية مع بيان المحتويات المتتابعة للرصة S والقيم المتتابعة للمتغيرات V, W, T .

```
create_s (S) ;
push (S , "father") ;
push (S , "your") ;
push (S , "then") ;
pop (S , V) ;
pop (S , T) ;
```

```
push (S , T) ;
push (S , V) ;
push (S , "mother") ;
top_s (S , W) ;
push (S , T) ;
push (S , V) ;
push (S , W) ;
push (S , T) ;
push (S , V) ;
push (S , W) ;
push (S , T) ;
```

٣-٥ أ) نفرض أن S رصة أعداد صحيحة تحتوي على الأعداد المبينة بالشكل التالي ، ونفرض أن temp رصة أعداد صحيحة وأن V متغير صحيح . تتبع تنفيذ خطوات الخوارزمية التالية مع بيان نتائج تنفيذ هذه الخطوات (بيان المحتويات المتتابعة والنهائية لكل من الرصتين S ، temp) . ما وظيفة هذه الخوارزمية ؟

```
create_s (temp) ;
while (! empty_s (S))
        {
                pop (S , V) ;
                if (! empty_s (S))
                        {
                                pop (S , V) ;
                                push (temp , V) ;
                        }
        }
while (! empty_s (temp))
        {
                pop (temp , V) ;
                push (S , V)
{
```

2
5
8
1
4
7
S

ب) نفرض أن S رصة أعداد صحيحة . باستخدام العمليات الست القياسية المعرفة على الرصات اكتب دالة

int StackSum (StackType S)

تعطي مجموع الأعداد الصحيحة المخزونة في S .
{ مثلا إذا كانت S تحتوي على الأعداد المبينة في الشكل السابق (الرصة S) فإن الدالة تعيد القيمة 27}
وإذا كانت الرصة S خالية فإن الدالة تعيد القيمة صفرا . وبعد انتهاء تنفيذ الدالة يجب أن تكون محتويات الرصة S هي نفسها قبل استدعاء الدالة .

٦-٣ نفرض أن S رصة أعداد صحيحة وأننا ننفذها باستخدام منظومة من عشرة أعداد صحيحة . ونفرض أننا نقوم بدفع (push) عددين صحيحين في الرصة ثم رفع (pop) عدد واحد ، وأننا نكرر هذه العملية (دفع عددين ثم رفع عدد) عدة مرات . كم عدد مرات تكرار هذه العملية قبل أن يحدث خطأ فيض (overflow error occurs) ؟

٧-٣ نفرض أن q طابور دائري من الرموز (circular queue of characters) ممثل بمنظومة مكونة من ست خلايا ، وأن الطابور يحتوي على القيم الثلاث المبينة بالشكل التالي (والخلايا الثلاث المتبقية خالية) :

المطلوب : بيان تأثير كل من العمليات المتعاقبة التالية (بيان محتويات الطابور بعد تنفيذ كل منها).

```
1)  insert (q , F) ;    2)  del (q , e) ;     3)  del (q , e) ;
4)  insert (q , K) ;    5)  insert (q , L) ;  6)  insert (q , M) ;
7)  del (q , e) ;       8)  del (q , e) ;     9)  insert (q , R) ;
10) del (q , e) ;       11) del (q , e) ;     12) insert (q , S) ;
13) del (q , e) ;       14) del (q , e) ;     15) del (q , e) ;
16) del (q , e) ;
```

٣-٨ نفرض أن لدينا بيانات مخزونة في طابور دائري ممثل بمنظومة مكونة من خلايا عددها N .

(i) كم عدد العناصر المخزونة في الطابور إذا كان :

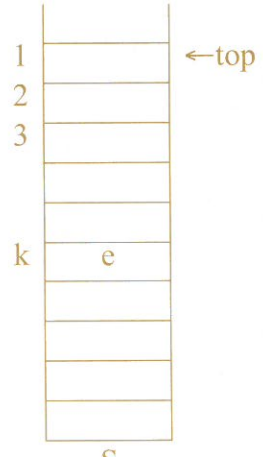
(أ) front = 3 , rear = 9 , N = 12

(ب) front = 9 , rear = 4 , N = 12

(ii) ما هي العلاقة التي تعطي عدد العناصر المخزونة في الطابور الدائري بدلالة front , rear , N ؟

٣-٩ باستخدام العمليات (القياسية) المعرفة على الرصات والطوابير اكتب خوارزمية لعكس (reversing) محتويات رصة. يمكنك الاستعانة برصة أخرى temp_s أو طابور temp_q للتخزين المؤقت.

٣-١٠ باستخدام العمليات (القياسية) المعرفة على الرصات والطوابير اكتب دالة **retrieve_element (S, k, e)** تعيد / تعطي قيمة العنصر (e) الذي ترتيبه k ابتداءً من أعلى رصة معطاة S (انظر الشكل) دون حذفه ، ودون تغيير محتويات الرصة. يمكنك الاستعانة برصة أخرى temp_S أو طابور temp_q للتخزين المؤقت.



٣-١١ نفرض أن الرصة S والطابور Q يحتويان على عناصر من النوع نفسه ، ونفرض كذلك أن X متغير من النوع نفسه . اكتب خوارزمية تستخدم الرصة S لعكس العناصر الموجودة حاليا في الطابور Q .

٣-١٢ اكتب خوارزمية لإيجاد عدد العناصر المخزونة في طابور . استخدم طابورا مؤقتا للوصول إلى ذلك .

٣-١٣ باستخدام العمليات (القياسية) التالية المعرفة على الطوابير :

create_q , insert , del , front_of_queue , empty_q , full_q

اكتب خوارزميات (دوال) لتنفيذ العمليات التالية :

أ) إلحاق (appending) طابور Q_2 بنهاية طابور Q_1 ، مع ترك Q_2 فارغا / خاليا.

ب) تقرير (deciding) ما إذا كان عنصرُ ما x موجودا في طابور Q أم لا ، دون إحداث أي تغيير / تخريب (destruction) في الطابور Q.

ج) دمج (merging) طابورين Q_1 , Q_2 في طابور ثالث جديد Q_3 بالطريقة التالية :

إذا كان الطابور Q_1 يحتوي على العناصر $x_1, x_2, \ldots, x_n$ ، والطابور Q_2 يحتوي على العناصر $y_1, y_2, \ldots, y_m$ ، فإنه يتم تكوين الطابور Q_3 بحيث يحتوي على العناصر $x_1, y_1, x_2, y_2, \ldots$ (أي عنصر من Q_1 وآخر من Q_2 على التعاقب) ، وبحيث يصبح كل من الطابورين Q_1 , Q_2 فارغا بعد عملية الدمج.

٣-١٤ اكتب صيغة الرمز اللاحق / المؤخَّر (postfix form) لكل من التعابير التالية :

a) A * B * C

b) –A + B – C + D

c) A * - B + C

d) (A+B) * D + E/(F + A * D) + C

e) A&& B || C || ! (E > F)

f) ! (A&& ! ((B < C) || (C > D))) || (C < E)

٣-١٥ استخدم رصة مع بيان محتوياتها المتتالية لإيجاد قيمة التعبير التالي المكتوب بالتمثيل الرمزي اللاحق (postfix) :

A B C + * D /

وذلك بفرض أن

A = 8 , B = 2 , C = 3 , D = 4

٣-١٦ ما هو التعبير الرمزي اللاحق (postfix expression) المكافئ للتعبير الرمزي الوسطي (infix expression) :

(أ) (a – b) * (d/e)

(ب) a * (b + d) / e – f * (g + h/k)

٣-١٧ تتبع تنفيذ خوارزمية تحويل التعبير الرمزي الوسطي إلى التعبير الرمزي اللاحق وذلك بالنسبة للتعبير الوسطي

i) a + b * c + (d * e + f) * g

ii) A –(B + C/D * E) – F * G

مع بيان المحتويات المتتابعة للرصة.

٣-١٨ أوجد قيمة كل من التعبيرين الحسابيين التاليين المكتوبين بالاصطلاح الرمزي اللاحق (postfix notation) . وضح خطوات الحل باستخدام رصة.

i) 2 5 2 3 + 8 * + 3 + *

ii) 12 7 3 - / 2 1 5 + * +

٣-١٩ نفرض أن لدينا التعريفات التالية :

```
int     stack [1000] ;
```

```
int     top1, top2, bottom1, bottom2 ;
```

ونفرض أن لدينا رصتين s2 ، s1 وأننا سنمثلهما في المنظومة الأحادية stack ، بحيث أن :

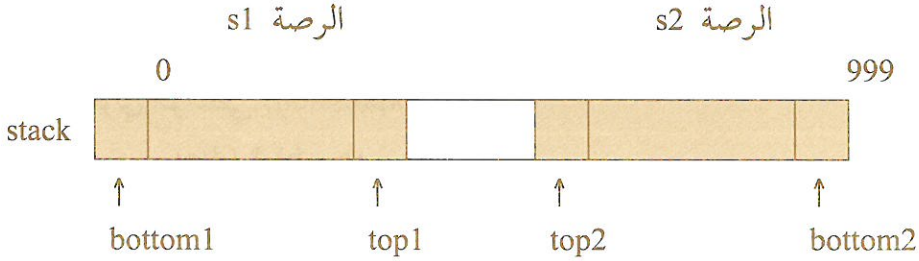
أسفل الرصة s1 هو : bottom1 = 0

أسفل الرصة s2 هو : bottom2 = 999

top1 : عدد صحيح يشير إلى موضع أعلى الرصة s1

top2 : عدد صحيح يشير إلى موضع أعلى الرصة s2

كما هو موضح بالشكل :



اكتب الدوال التالية :

أ) دالة خاوية **void create** لإنشاء رصتين خاليتين .

ب) دالة منطقية **bool empty1** لاختبار أن الرصة الأولى s1 فارغة .

ج) دالة منطقية **bool empty2** لاختبار أن الرصة الثانية s2 فارغة .

د) دالة منطقية **bool full** لاختبار أن المنظومة كلها ممتلئة (وبالتالي لا يمكن إضافة أي عنصر لأي من الرصتين).

هـ) دالة خاوية **void push1(int e)** لإضافة عنصر e للرصة الأولى s1 (بشرط ألا تكون المنظومة كلها ممتلئة) .

و) دالة خاوية **void push2(int e)** لإضافة عنصر e للرصة الثانية s2 (بشرط ألا تكون المنظومة كلها ممتلئة) .

ز) دالة خاوية **void pop1(int& e)** لإخراج عنصر من الرصة الأولى s1 وإعادة قيمته e بشرط ألا تكون الرصة (الأولى) فارغة .

ح) دالة خاوية **void pop2 (int& e)** لإخراج عنصر من الرصة الثانية s2 وإعادة قيمته e بشرط ألا تكون الرصة (الثانية) فارغة .

٣-٢٠ نفرض أن لدينا الإعلانات التالية :

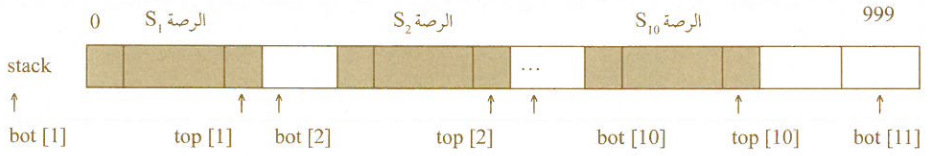
```
int     stack [1000] ;
int     top [11] , bot [12] ;
```

ونفرض أن لدينا 10 رصات $S_1,S_2,\ldots,S_i,\ldots,S_{10}$ (مرقمة من 1 إلى 10) وأننا سنمثلها في المنظومة الأحادية stack ، بحيث أن :

top [i] : عدد صحيح يشير إلى موضع أعلى الرصة S_i {i=1,2,...,10}

bot [i] : عدد صحيح يشير إلى الموضع السابق مباشرة لأسفل الرصة S_i {i=1,2,...,10}

كما هو موضح بالشكل التالي (لاحظ أن bot [1] = -1 ونهمل قيمتي (bot [0], top[0] :



وأننا سنستخدم الشرط bot [i] = top [i] إذا وفقط إذا كانت الرصة S_i خالية (empty).

ونفرض أن الرصة S_i تنمو من bot [i] + 1 وحتى bot [i+1] ، أي أنها تصبح ممتلئة (full) عندما يتحقق الشرط top [i] = bot [i+1].

ولتسهيل كتابة الخوارزميات سنستخدم التعريف bot [11] = 999.

أ) اكتب دالة خاوية

void push (int i, int e)

لإضافة عنصر e للرصة S_i ، بشرط ألا تكون الرصة S_i ممتلئة.

ب) اكتب دالة خاوية

void pop (int i, int& e)

لإخراج عنصر من الرصة S_i وإعادة قيمته e ، بشرط ألا تكون الرصة S_i فارغة.

ج) نفرض أن الرصة S_3 ممتلئة والرصة S_2 غير ممتلئة . اكتب دالة خاوية **void shift3** تزيد حَيِّز (space) الرصة S_3 بموضع واحد وتنقص الحيز المسموح للرصة S_2 بموضع واحد {أي تزيح (shift) الرصة S_3 كلها موضعا واحدا لليسار ، بينما تبقى جميع الرصات الأخرى دون أي تحريك}.

د) اكتب دالة **int free** تحسب العدد الإجمالي للمواضع الخالية (empty locations) في المنظومة stack (أي تحسب عدد المواضع التي لا يشغلها أي عنصر من عناصر الرصات) .

الفصل الرابع

القوائم المترابطة
Linked Lists

4

4 Linked Lists

الفصل الرابع

القوائم المترابطة

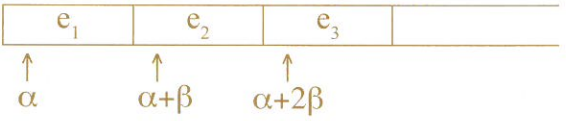
Linked Lists

تمهيد (التمثيل التتابعي والتمثيل المترابط للقوائم) :

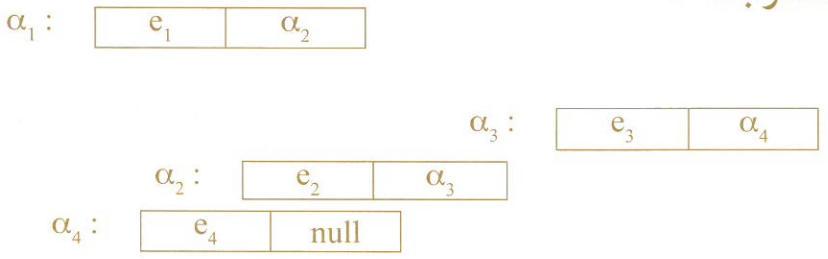
Sequential representation and linked representation of lists

رأينا سابقا أنه إذا تم تخزين قائمة (list) بطريقة تتابعية (stored sequentially) مثلا باستخدام منظومة (array) فإن عمليات الإضافة / الإدخال والحذف (insert / delete) من القائمة تكون نسبيا مكلفة (expensive) وذات كفاءة منخفضة (inefficient) وذلك أساسا بسبب الحاجة إلى عمليات الإزاحة (shift operations). ويمكن التغلب على هذه المشكلة باستخدام القوائم المترابطة التي لا تستخدم التمثيل التتابعي وإنما نضع مع كل عنصر عنوان العنصر التالي له (next / successor element) في القائمة.

التمثيل التتابعي :



التمثيل المترابط :



وعادة ترسم القائمة المترابطة بالشكل التالي :

وعادة يتم تنفيذ القائمة المترابطة عمليا باستخدام المؤشرات (pointers) وأحيانا باستخدام المنظومات (arrays) (خاصة في اللغات التي ليس فيها مؤشرات كلغة الفورتران ولغة البيسك). وتعد القوائم الخطية (linear lists) أكثر القوائم شهرة حيث يُراعَى ترتيب معين لعناصرها ، كأن ترتب مثلا بناء على مجال الاسم (name field).

العمليات على القوائم Operations with lists

فيما يلي مجموعة من العمليات على القوائم :

- إنشاء قائمة خالية (create an empty list).
- تحديث / تغيير (update / change) عنصر / عقدة (node / element) في القائمة.
- استرجاع (retrieve) جزء من البيانات من عنصر بالقائمة.
- حذف (delete) عنصر.
- إدخال (insert) عنصر.
- اجتياز (traverse) القائمة.
- إيجاد طول القائمة.
- دمج (merge) قائمتين.
- شطر (split) قائمة إلى قائمتين بناء على شرط معين.

تعريف النوع :

جزء العنوان	جزء البيانات
next	info

```
typedef node  *node_ptr ;
struct node
   {
        int data ;
        node_ptr next ;
   } ;
node_ptr p, q, t, last ;
```


ونفرض أن القارئ على علم بالإجراءات / المؤثرات التالية التي تستخدم مع المؤشرات :

new, delete, NULL, =, = =, !=

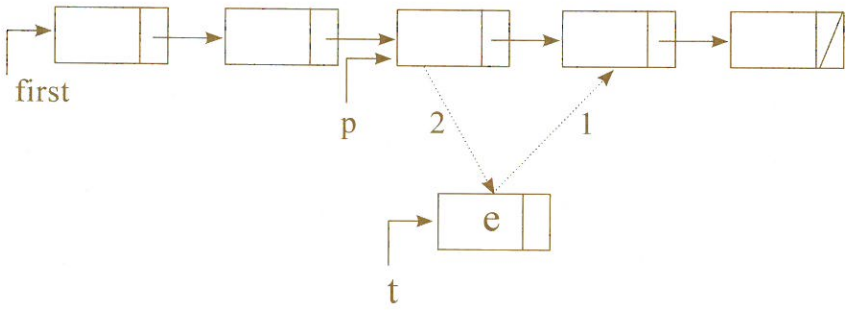
ونتناول فيما يلي بعض خوارزميات / دوال العمليات على القوائم وعناصرها.

إجراء إنشاء عنصر جديد وتخزين بيانات فيه :

```
Void make_new_node (node_ptr& t, int e)
{
      t = new node ;
      t → data = e ;
      t → next = NULL ;
}
```

خوارزمية إدخال (insert) عنصر بعد عنصر معين p

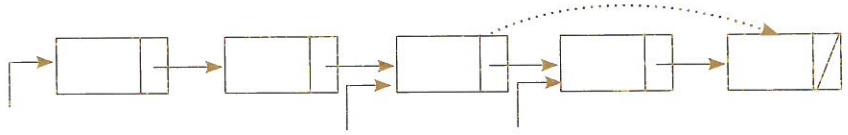
(أي بعد عنصر معين يشير إليه المؤشر p)



```
t = new node ;
t → data = " data to be inserted " ;
t → next = p → next ;
p → next = t ;
```

ملاحظة : قارن هذه الخوارزمية مع خوارزمية إدخال عنصر بعد العنصر رقم i في منظومة معلومة.

خوارزمية حذف (delete) عنصر بعد عنصر معين




```
t = p → next ;
p → next = t → next ;
delete t ;
```

ملاحظة : قارن مع خوارزمية حذف العنصر رقم i من منظومة.

وننصح القارئ بمراجعة عمليات :

* إضافة عنصر عند بداية قائمة.
* حذف عنصر من بداية قائمة.
* إضافة عنصر عند نهاية قائمة.

[إذا تكررت هذه العملية كثيرا كما في حالة الطابور ، فقد يكون من الأنسب الاحتفاظ بمؤشر نطلق عليه مثلا tail / rear / last يشير إلى آخر عنصر في القائمة].

* حذف العنصر الأخير في قائمة.
* إضافة / حذف عنصر وسط قائمة.

مقارنة بين تنفيذ القوائم باستخدام المنظومات واستخدام المؤشرات
Comparison between array & pointer implementation of lists

تعتمد أفضلية استخدام المنظومات أو استخدام المؤشرات لتنفيذ القوائم على العمليات المطلوب تنفيذها وعلى مدى تكرارها (أي عدد مرات تنفيذها) كما يتضح من النقاط التالية :

١- استخدام المنظومات يتطلب معرفة أقصى حجم ممكن (max. possible size) للقائمة سلفا ، بينما استخدام القوائم المترابطة لا يتطلب ذلك ، فالحجم في الحالة الأولى مسألة استاتيكية أما في الحالة الأخيرة فهو مسألة ديناميكية.

٢- عمليات الإضافة والحذف تتم بكفاءة أكبر باستخدام المؤشرات بدلا من استخدام المنظومات ، حيث أننا لا نحتاج لأي عمليات إزاحة ، وإنما نقوم فقط بتغيير بعض المؤشرات.

٣- عمليات البحث في قائمة مرتبة تتم بكفاءة أكبر باستخدام المنظومات ، حيث يمكننا مثلا استخدام البحث الثنائي (binary search) ، أما التمثيل باستخدام القائمة المترابطة فيتطلب إجراء البحث التتابعي (sequential search).

٤- التنفيذ باستخدام المؤشرات يتطلب حيزا أكبر في التخزين (extra storage) ، وذلك لأنه يتطلب تخزين مجال للمؤشر (pointer field) مع مجال المعلومات (information field) لكل عنصر من عناصر القائمة

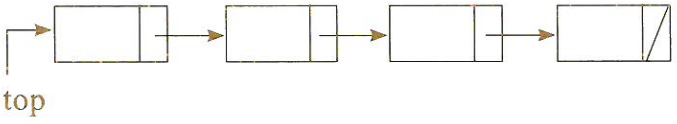
المؤشر	المعلومات
Pointer	info

الرصات المترابطة والطوابير المترابطة
Linked Stacks & Linked Queues

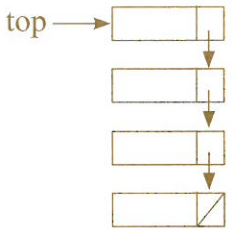
رأينا سابقا أن التمثيل التتابعي للرصة أو الطابور – أي التمثيل باستخدام المنظومات – بسيط وذو كفاءة عالية [ما عدا حالة امتلاء الرصة أو الطابور ، أي امتلاء حيز التخزين المسموح به (storage overflow)]. وكذلك فإن التمثيل المترابط للرصة أو الطابور بسيط وذو كفاءة عالية وبالدرجة نفسها ، وسنقوم بإذن الله بكتابة بعض العمليات (القياسية) التالية باستخدام التمثيل المترابط (أي باستخدام مؤشرات) ونترك كتابة باقي العمليات كتدريب للقارئ :

إنشاء رصة خالية / طابور خالي ، إضافة عنصر ، حذف عنصر ، اختبار أن الرصة خالية / الطابور خالي ، إيجاد قيمة العنصر الأمامي في الطابور / العنصر أعلى الرصة. وبعد ذلك نقوم بتنفيذ الرصات المتعددة والطوابير المتعددة.

(أ)الرصة المترابطة
Linked Stack



تعريف الرصة المترابطة : فيما يلي تعريف نوع للرصة المترابطة ، مع إعادة رسم مخططها بحيث تظهر بالشكل المألوف للرصة (العناصر بعضها فوق بعض).

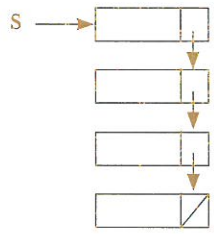



```
typedef node  *node_ptr;
struct node
        {
                int data ;
                node_ptr next ;
        } ;
stack stack
        {
                node_ptr top ;
        } ;
stack s, s1
```

التعريف المذكور يعتبر نوع الرصة stack سجلا يشتمل على مجال واحد top عبارة عن مؤشر إلى سجل node [يشتمل على مجالين : data وفيه البيانات ، والآخر next وهو مؤشر إلى السجل / العنصر التالي].

وفيما يلي تعريف آخر لنوع الرصة (stack) يعتبر stack مؤشرا (وليس سجلا) يشير إلى السجل node .

```
typedef node  *node_ptr ;
struct node
        {
                int data ;
                node_ptr next ;
        } ;
stack = node_ptr ;
stack s, s1 ;
```



وفيما يلي سنستخدم التعريف السابق الذي يَعتبر نوع الرصة stack سجلا (وليس مؤشرا).

تنفيذ العمليات على الرصات :

implementation of stack operations

فيما يلي نكتب العمليات القياسية الست التالية باستخدام التمثيل المترابط

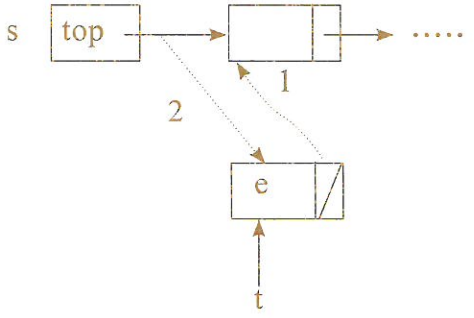
create_s ; **push ;** **pop ;**

top_s ; **full_s ;** **empty_s ;**

مثال ٤-١ :

باستخدام التعريف السابق للرصة المترابطة اكتب إجراء push لإضافة عنصر e أعلى الرصة المترابطة s .

الحل :




```
void push (stack& s, int e) ;
{
```

```
        node_ptr t;
        if full_s (s)
                cout << " stack is full " << endl ;
        else
                {
                        make_new_node (t , e) ;
                        t → next = s.top ;
                        s.top = t
                }
}
```

ملاحظة : إذا استخدمنا التعريف الآخر لنوع الرصة stack ، أي أنها مؤشر (وليس سجلا) ، فإن العبارة المركبة بعد else يمكن تغييرها إلى ما يلي :

```
else
        {
                make_new_node (t , e) ;
                t → next = s ;
                s = t
        }
```

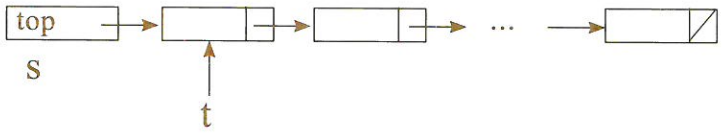
مثال ٤-٢ :

باستخدام التمثيل المترابط المعرف سابقا للرصة اكتب دالة pop لإخراج عنصر من أعلى الرصة المترابطة s ، وإعادة قيمته.

الحل :

```
int pop (stack& s)
{
        int k ;
        if empty_s (s)
                cout << " stack is empty " << endl ;
```

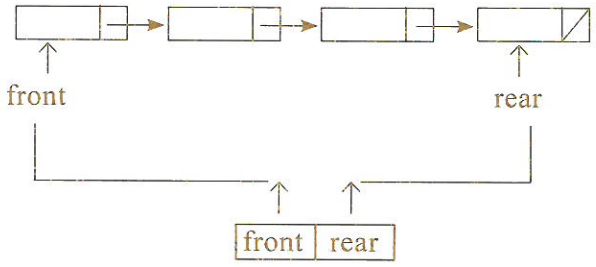
```
        else
                {
                        t = s.top ;
                        k = s.top → data ;
                        s.top = s.top → next ;
        delete t ;
        return k ;
        }
}
```



ملاحظة : المستخدم (user) لا يلاحظ أي فارق بين الرصات المترابطة والرصات التتابعية (أي بين الرصات المنفذة باستخدام التمثيل المترابط والرصات المنفذة باستخدام التمثيل التتابعي).

(ب) الطابور المترابط

Linked Queue



تعريف الطابور المترابط :

يمكننا تعريف التمثيل المترابط للطابور كما يلي ، حيث نعتبر النوع queue سجلا يشتمل على مجالين : أحدهما front مؤشر يشير إلى مقدمة الطابور ، والآخر rear مؤشر يشير إلى مؤخرة الطابور.

```
typedef node  *node_ptr ;
struct node
      {
            int data ;
            node_ptr next ;
       } ;
struct queue
       {
             node_ptr front, rear ;
       } ;
queue q, q1
```

ويمكننا استخدام هذا التعريف للطابور المترابط لتنفيذ أي من العمليات الست القياسية على الطوابير :

create_q , insert , del ,
front_of_q , full_q , empty_q ,

مثال ٤-٣ :

باستخدام التعريف السابق للطابور المترابط اكتب إجراء **insert** لإدخال عنصر e في الطابور المترابط q.

الحل :

```
void insert (queue& q , int  e) ;
{
```

```
        node_ptr t ;
        if full_q (q)
                cout << " queue is full " << endl ;
        else
                {
                        make_new_node (t , e) ;
                        if empty_q (q)    // special case
                                {
                                        q.front = t ;
                                        q.rear = t ;
                                }
                        else              // link at the end.
                                {
                                        q.rear → next = t ;
                                        q.rear = t ;
                                }
                }
}
```

مثال ٤-٤ :

باستخدام التمثيل المترابط - المعرف سابقا - للطابور اكتب دالة **del** لحذف عنصر من الطابور q ، وإعادة قيمة هذا العنصر المحذوف.

الحل :

```
int del (queue& q)
{
        int k ;
        if empty_q (q)
                cout << " queue is empty " << endl ;
        else
                {
                        k = q.front → data ;
```

```
                t = q.front ;
                delete t ;
                if (q.front = = NULL)
                        q.rear = NULL
                return k ;
        }
}
```


ملاحظة : هناك احتمال أن يكون الطابور الأصلي مكونا من عنصر واحد فقط يشير إليه كل من front & rear وبعد حذفه سيصبح front = NULL فيجب أن يصبح rear أيضا NULL.

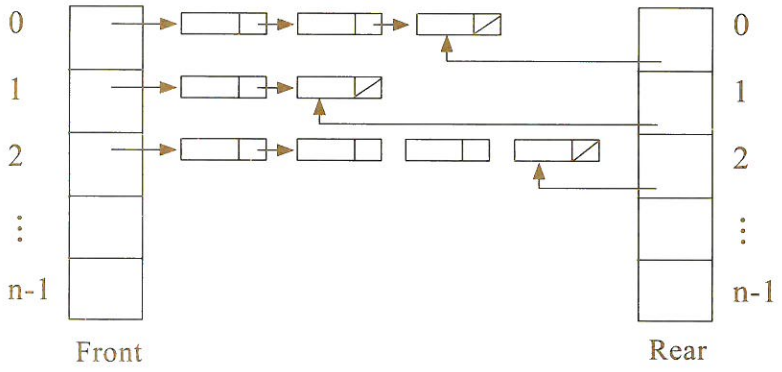
الرصات والطوابير المتعددة
Multiple Stacks and Queues

درسنا في الفصل السابق كيفية تنفيذ الرصات المتعددة والطوابير المتعددة عن طريق التمثيل التتابعي (أي باستخدام المنظومات أحادية البعد) ، وفيما يلي ندرس تنفيذ هذه الرصات والطوابير عن طريق التمثيل المترابط (أي باستخدام مؤشرات وقوائم مترابطة).

التنفيذ باستخدام التمثيل المترابط
Implementation using linked representation

يمكننا تمثيل عدد n من الرصات / الطوابير (حيث n = 1, 2, 3, ...) بسهولة وكفاءة عالية باستخدام المؤشرات. نفرض مثلا أن لدينا n طابورا. يمكننا تمثيل

كل طابور بقائمة مترابطة (مستقلة) ثم نستخدم منظومتين : الأولى منظومة مؤشرات تتكون من n عنصر (array of n pointers) تشير عناصرها – على الترتيب – إلى مقدمات / أوائل (fronts) الطوابير ، والثانية هي أيضا منظومة مؤشرات من n عنصر ولكن عناصرها تشير إلى أواخر / مؤخرات (rears) الطوابير ، كما يبين ذلك الشكل التالي



ويلاحظ أنه في هذا التمثيل يمكننا تشغيل (processing) أي طابور مستقلا عن باقي الطوابير.

تعريف النوع والإعلانات (Type Definition / Declarations)

يمكننا تعريف الطوابير المتعددة باستخدام أحد التعريفين التاليين :

(١) التعريف الأول :

```
typedef node  *node_ptr ;
struct node
        {
                int data ;
                node_ptr next ;
        } ;
node_ptr Front [100] ;
node_ptr Rear [100] ;
```

أو اعتبار Rear ، Front متغيرين كما يلي :

```
typedef  node_ptr header [100] ;
header Front , Rear ;
```

(٢) التعريف الثاني :

```
struct queue
        {
                node_ptr front ;
                node_ptr rear ;
        } ;
typedef  queue  queue_system [100] ;
queue_system  q ;
```


ونلاحظ في هذا التعريف أننا مثَّلنا الطوابير المتعددة بمنظومة سجلات واحدة يشتمل كل سجل فيها على مجالين أحدهما مؤشر يشير إلى مقدمة طابور والآخر مؤشر يشير إلى مؤخرة هذا الطابور ، بينما في التعريف الأول مثَّلنا الطوابير المتعددة بمنظومتين إحداهما منظومة مؤشرات تشير إلى مقدمات الطوابير والأخرى منظومة مؤشرات تشير إلى مؤخرات الطوابير.

ولإنشاء الطوابير الخالية يمكننا استخدام العبارتين التاليتين في حالة التعريف الأول الذي يعتبر Front , Rear متغيرين :

$$\left.\begin{array}{l} \text{Front [i] = NULL;} \\ \text{Rear [i] = NULL;} \end{array}\right\} \quad i = 0,1,2,\ldots, n - 1$$

أو استخدام عبارة الاستدعاء التالية في حالة التعريف الثاني :

```
create_q (q[i] ) ;          i = 0,1,2,..., n - 1
```

ونلاحظ أن العمليات التي تجرى على الطابور رقم i (i^{th} queue) مطابقة تماما للعمليات المذكورة سابقا على طابور واحد.

مثلا لإدخال عنصر e في الطابور رقم i نستدعى الإجراء **insert** هكذا :

```
insert (q[i] , e) ;
```

ولحذف عنصر من الطابور رقم i نستدعى الدالة **del** هكذا :

```
x = del (q[i]) ;
```

الحدوديات
Polynomials

4

تنفيذ الحدوديات باستخدام القوائم المترابطة
(Linked list implementation of Polynomials)

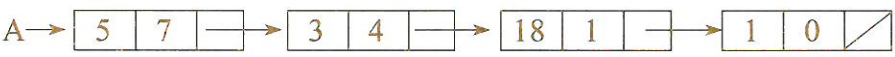
يمكن تنفيذ الحدودية A(x) كقائمة مرتبة (ordered list) من عدة عقد (عناصر) حيث يمثل أي عنصر حدا من حدود الحدودية ، ونعتبر العنصر سجلا مكونا من ثلاثة مجالات.

مؤشر إلى الحد التالي	الأس	المعامل
next	exp	coef

فمثلا الحدودية

$$A(x) = 5\,x^7 + 3\,x^4 + 18\,x + 1$$

يمكن تمثيلها بالقائمة المترابطة



حيث القائمة مرتبة تنازليا حسب الأس (ordered by decreasing exponent) أو القائمة المترابطة

حيث القائمة مرتبة تصاعديا حسب الأس

تعريف النوع :

```
typedef  node  *poly ;
struct node
        {
                int coef ;
                int exp ;
                poly next ;
        } ;
poly A, B, C;
```

ملاحظة : يتميز هذا التمثيل باستخدام القوائم المترابطة عن التمثيل باستخدام المنظومات بأنه لا تنشأ هنا مشكلة امتلاء المنظومة (full array problem).

تمرين :

اكتب إجراء لقراءة حدودية A(x) ، أي لقراءة المعامل والأس (coef , exp) لكل حد من حدودها ، وتكوين القائمة المترابطة التي تمثل A(x) (أي تخزين هذه المعاملات والأسس في قائمة مرتبة) ، مع ملاحظة أن الحدود (المعاملات ، والأسس) قد تصل بأي ترتيب. وتحتاج أساسا لتكرار إدخال (insert) حد جديد في موضعه الصحيح. وإذا تكرر ظهور أس معين فاجمع المعاملات المتقابلة ، فمثلا المدخلات التالية تولِّد الحدودية السابقة A(x).

(1,0) , (3 , 4) , (10 , 1) , (5 , 7) , (8 , 1)

وذلك لأن

(10 , 1) , (8 , 1) ≡ (18 , 1)

أي أن الحد (1 , 18) قد أُدخل كحدين.

جمع الحدوديات (Polynomial Addition)

مثال ٤-٥ :

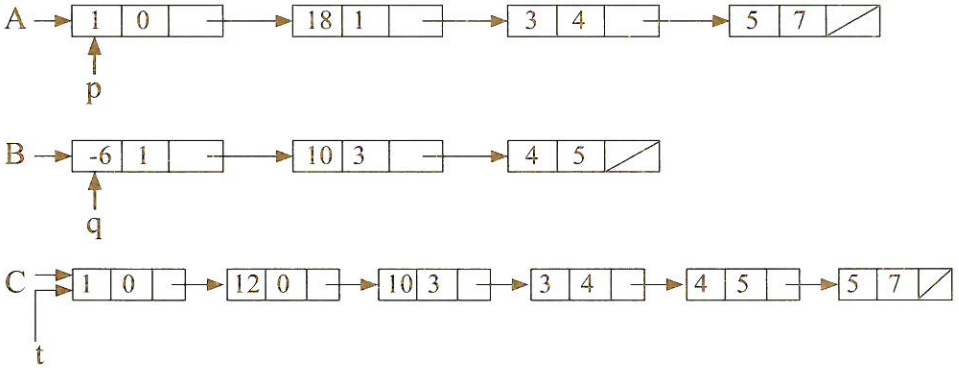
نفرض أن كلا من A(x) , B(x) حدودية معطاة كقائمة مترابطة. اكتب إجراء لجمع الحدوديتين والحصول على الحدودية

$$C(x) = A(x) + B(x).$$

الحل :

نلاحظ أن عملية الجمع هي أساسا عملية دمج (merging) للقائمتين A , B وتكوين القائمة الجديدة C ، بحيث تتم عملية الدمج بناءً على الأسس. كذلك نلاحظ أننا في عملية الدمج هذه ننسخ (copy) حدًّا (term) من A أو من B [وهو الحد الأصغر أسًّا (lower exp)] ونضيفه / ندخله (insert / attach) في نهاية (at the end) القائمة C. ولذلك فيجب الاحتفاظ بمؤشر (pointer) يشير إلى آخر عنصر (last node) من عناصر القائمة C. {سنرمز لهذا المؤشر بالرمز t}.

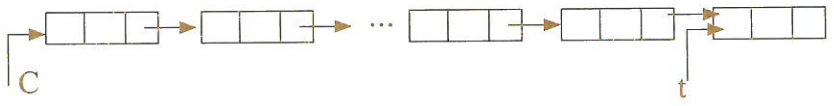
مثال :




```
void poly_add (poly A, B, poly& C)
{
```

```
poly p, q, t ;
int temp ;
// initialization :
C = NULL ;
t = NULL ;      // t points to the last node in C.
p = A;   q = B; // p, q point to the next term of A, B.
while ((p != NULL) && (q != NULL))

  if ((p → exp) < (q → exp))
          {
          insert_at_end (C, t, p → coef, p → exp) ;
          p = p → next ;
          }
  else if ((q → exp) < (p → exp))
          {
          insert_at_end (C, t, q → coef, q → exp) ;
          q = q → next ;
          }
  else    // equal ; add the coefficients.
          {
          temp = p → coef + q → coef;
          if (temp != 0)
                  insert_at_end (C, t, temp, p → exp) ;
          p = p → next ;
          q = q → next ;
          }
while (p != NULL)     // copy rest of A to end of C.
          {
          insert_at_end (C, t, p → coef, p → exp) ;
          p = p → next ;
```

```
        }
    while (q != NULL)    // copy rest of B to end of C.
        {
        insert_at_end (C, t, q → coef, q → exp) ;
        q = q → next ;
        }
}
```



إجراء إنشاء عنصر جديد وإضافته في نهاية قائمة مترابطة تمثل الحدودية C

المؤشر t يشير إلى آخر عنصر من عناصر القائمة C

```
void insert_at_end (poly& C, t, int coef , exp) ;
{
    poly p ;

    p = new node;
    p → coef = coef;
    p → exp = exp;
    p → next = NULL
```

} make _ new _ node(p, coef, exp);

```
    if (C = =  NULL)
        {
            C = p ;
            t = p ;
        }
    else
        {
            t → next = p ;
```

```
                t       = p ;
            }
}
```

درجة تعقيد الدالة poly_add (Complexity of the function)

* تكاليف (cost) / درجة تعقيد (complexity) الخوارزمية **insert_at_end** تساوي O(1).

* بالنسبة للخوارزمية **poly_add** :

في أسوأ حالة (worst case) نأخذ حدا واحدا فقط إما من A أو من B لإضافته إلى C (دون جمع حدين - أحدهما من A والآخر من B - متطابقي الأس) بعد كل مقارنة (if) ، وبالتالي يكون :

عدد العمليات (no. of operations) في أسوأ حالة : O(m + n)

حيث

m : عدد الحدود غير الصفرية في الحدودية A

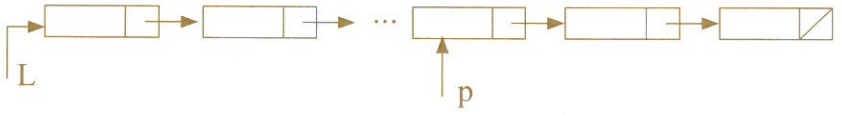
n : عدد الحدود غير الصفرية في الحدودية B

(مثلا : نصل إلى أسوأ حالة عندما تكون A حدودية فردية القوى و B حدودية زوجية القوى).

صور أخرى للقوائم المترابطة

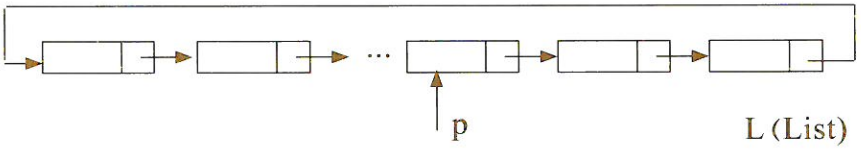
القوائم المترابطة الدائرية

Circular Linked Lists (CLL)



في القائمة المترابطة المفردة لا يمكننا من أي عنصر p إلا أن نتحرك للأمام فقط ، ولا يمكننا التحرك للخلف ، وبالتالي لا يمكننا الوصول إلى أي عنصر سابق إلا إذا بدأنا من أول القائمة.

ويمكننا تحويل هذه القائمة المفردة إلى ما يسمى قائمة دائرية عن طريق تعديلها بأن نضع بدلا من مؤشر التلاشي NULL في آخر عنصر مؤشرا إلى أول عنصر كما بالشكل.



قائمة مترابطة مفردة دائرية CSLL (circular single linked list)

(ملاحظة : الآن لا يوجد في الواقع عنصر أول وعنصر أخير !!)

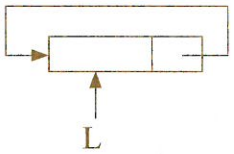
والآن يمكننا أن نبدأ من أي عنصر p ونتحرك للأمام إلى أن نصل إلى أي عنصر آخر. ومن المعتاد أن نجعل المؤشر إلى القائمة كلها " L " يشير إلى " آخر عنصر " (last node) فيها. وبالتالي يتم الوصول إلى أول عنصر بمجرد تحريك المؤشر L إلى العنصر التالي L → next .

اصطلاح :

القائمة المترابطة الفارغة L () يرمز لها هكذا (empty CLL

والقائمة المترابطة الدائرية ذات العنصر الواحد

(CLL with one node /1 node) يرمز لها هكذا



ونلاحظ هنا أن L → next = = L

مثال ٤-٦ : اجتياز القائمة الدائرية Traversal of a

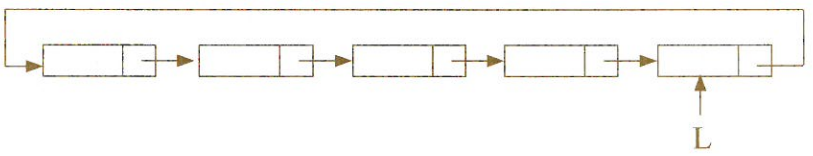
اكتب خوارزمية لاجتياز قائمة مترابطة دائرية L.

الحل :

نلاحظ أنه لا يمكننا الاجتياز عن طريق الخوارزمية

```
p = L ;
while (p != NULL)
  {
        .....
        .....
        .....
  }
```

وذلك لأنه لا يوجد عنصر يشتمل على مؤشر التلاشي.



وإنما يمكننا اجتياز القائمة الدائرية L عن طريق الخوارزمية التالية :

```
if (L != NULL)      // list is not empty.
```

```
{
 p = L → next ;       // we start at first elt.
 while (p !=L)      // L is the last elt.
        {
                visit node p ;
                // do required operations on elt. p.
                p = p → next ;
        }
 // visit the last node.
 visit node p(≡ L)      // do operations on last elt.
 }
```


تمرين : اكتب إجراءات تقوم بالعمليات التالية :

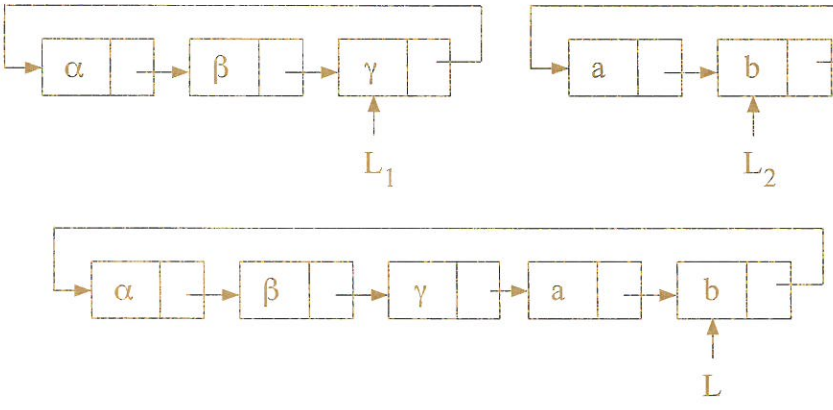
١) إدخال عنصر جديد في بداية قائمة دائرية .

٢) إدخال عنصر جديد في نهاية قائمة دائرية .

٣) حذف عنصر من بداية قائمة دائرية .

٤) حذف عنصر من نهاية قائمة دائرية .

٥) سَلْسَلَة / تسلسل / تعاقب / إلحاق / وصل (concatenation / appending) قائمتين دائريتين L1 ، L2 في قائمة دائرية واحدة L



ملاحظة : في القائمة الدائرية يجب أخذ الحالات التالية في الاعتبار :

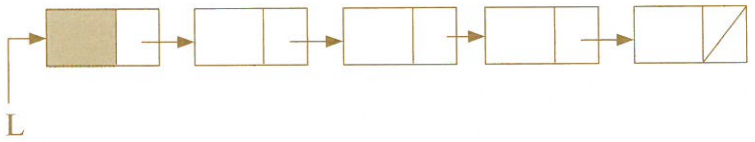
١- القائمة فارغة .

٢- القائمة بها عنصر واحد فقط (خاصة عند الحذف)

٣- القائمة بها أكثر من عنصر واحد .

العناصر الوهمية في واجهة القوائم المترابطة

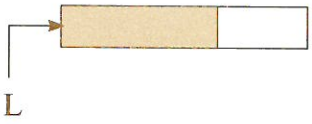
Dummy (header) nodes of linked lists



قائمة مترابطة مفردة من أربعة عناصر (وعنصر وهمي)

أحيانا يكون من الأنسب والأسهل عند كتابة الخوارزميات استخدام عنصر شكلي / وهمي / زائف / كاذب / افتراضي / صناعي في بداية / واجهة / مقدمة القائمة المترابطة (dummy (header) node of a linked list) حيث يُبَسِّط وجود هذا العنصر كتابة العمليات التي تُجرى على القائمة ، وخاصة عمليات الحذف والإضافة حيث لا نشغل أنفسنا بالحالات الخاصة (special cases) مثل : L = = NULL . ويشتمل مجال البيانات (data field) على أي بيانات اختيارية نضعها ، ويمكننا بعد الانتهاء من العمليات المطلوب إجراؤها على القائمة حذف هذا العنصر الوهمي ، وفي أحيان كثيرة نحتفظ بهذا العنصر الوهمي كجزء من هيكل البيانات (data structure). وهذه العناصر الوهمية [التي تعد كعناصر حارسة (sentinel nodes)] قد تضاف عند إحدى نهايتي قائمة أو عند نهايتيها.

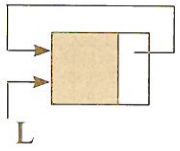
مثلا القائمة الفارغة (ذات العنصر الوهمي) تمثل هكذا :



حيث

$$L \rightarrow next == NULL$$

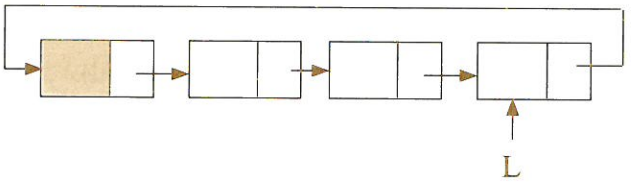
والقائمة الدائرية الفارغة (ذات العنصر الوهمي) تمثل هكذا :



حيث

$$L \rightarrow next == NULL$$

والقائمة الدائرية ذات العناصر الثلاثة (والعنصر الوهمي) تمثل هكذا :



مثال ٤-٧ :

اكتب إجراء لاجتياز قائمة دائرية L تشتمل على عنصر وهمي (في واجهتها).

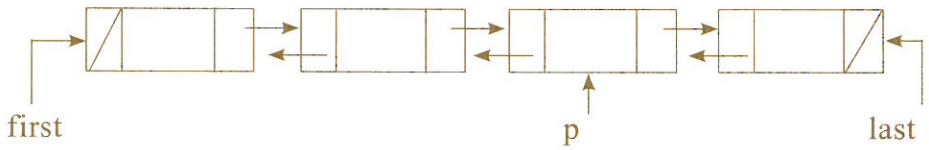
الحل :

```
p = L → next ;
while (p != L)
    {
        process p → data ;
```

```
        p = p → next
}
```

القائمة مزدوجة / ثنائية الارتباط

Double Linked List



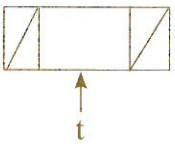
يشتمل كل عنصر في هذه القائمة على مؤشرين : أحدهما next يشير إلى العنصر التالي ، والآخر prev يشير إلى العنصر السابق.

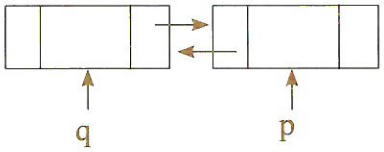

```
typedef node  *node_ptr ;
struct node
        {
                int data ;
                node_ptr next, prev ;
        } ;
```

مثال ٤-٨ :

اكتب خوارزمية لإضافة عنصر t قبل عنصر p (حيث p ليس أول عنصر) في قائمة دائرية مزدوجة الارتباط.

الحل :






```
q = p → prev ;
q → next = t ;
t → prev = q ;

p → prev = t ;
t → next = p
```

مميزات القائمة مزدوجة الارتباط

* سهولة اجتياز القائمة للأمام أو للخلف (forward / backward traversal).
* سهولة إضافة عنصر قبل أو بعد عنصر معين في القائمة (insertion before or after a given node)
* سهولة حذف عنصر معين من القائمة (deletion of a given node)

إلا أن العمليات عموما تكون أطول منها في حالة القوائم المفردة ، ونقوم هنا (في حالة القوائم المزدوجة) بعمل بعض التعديلات في المؤشرات ، بالإضافة إلى أننا نحتاج إلى حيز أكبر في التخزين وذلك للمؤشر الإضافي prev.

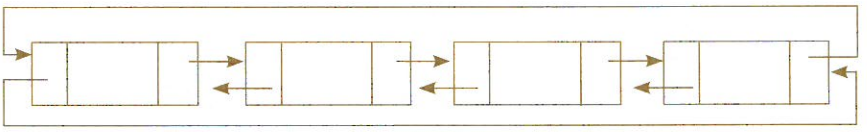
تمرين : اكتب إجراءات تقوم بالتالي :

- اجتياز قائمة مزدوجة dll من اليسار لليمين.
- اجتياز قائمة مزدوجة dll من اليمين لليسار.
- إضافة / حذف عنصر عند بداية / نهاية قائمة مزدوجة .
- إضافة عنصر بعد / قبل عنصر معين.
- حذف عنصر معينp.

القائمة الدائرية مزدوجة الارتباط

Circular Double Linked List CDLL

تمثل القائمة الدائرية مزدوجة الارتباط بالشكل التالي :



ملاحظة : يمكن أن يكون هناك عنصر وهمي عند كل من بداية ونهاية القائمة DLL لتكون متماثلة (symmetric) .

المصفوفات المتناثرة

Sparse Matrices

سنبدأ بإذن الله بتمثيل بسيط - باستخدام القوائم المترابطة - للمصفوفات المتناثرة A_{mxn} ، ثم ننتقل إلى تمثيل أعقد - أيضا باستخدام القوائم المترابطة - ولكنه يؤدي إلى كفاءة أكبر وسهولة من حيث إجراء العمليات على هذه المصفوفات.

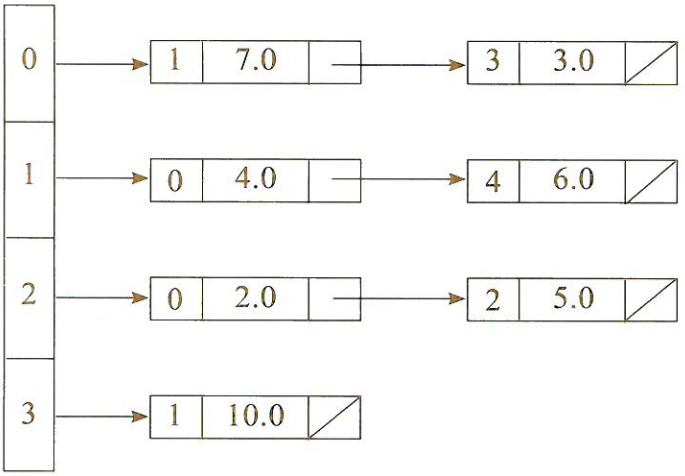
مثال ٤-٩ :

افرض أن لدينا المصفوفة

$$A_{4x5} = \begin{pmatrix} 0 & 7 & 0 & 3 & 0 \\ 4 & 0 & 0 & 0 & 6 \\ 2 & 0 & 5 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 \end{pmatrix}$$

ارسم شكلا تخطيطيا يوضح كيفية تخزين A صفا صفا ، حيث كل صف يُمثَّل بقائمة مترابطة مستقلة بذاتها ، واكتب تعريف نوع مناسب لهذا التمثيل.

الحل :



أي أن كل عنصر في القائمة المترابطة عبارة عن سجل يتكون من ثلاثة مجالات هي :

مؤشر يشير إلى العنصر التالي في الصف next	قيمة العنصر value	رقم العمود col

تعريف النوع

```
const int maxrow = 100 ;
typedef node  *node_ptr ;
struct node
        {
                int col ;      // column number.
```

```
        float value ;
        node_ptr next ;    // next in row.
    } ;
typedef node_ptr pointer_array [maxrow] ;
pointer_array  xa ;
```

حيث maxrow هو حد أعلى (bound) لعدد الصفوف m.

* * *

نلاحظ أن هذا التمثيل يؤدي إلى سهولة وكفاءة العمليات التي تجرى على المصفوفة صفا صفا ، ولكنه غير مناسب بالنسبة للعمليات التي تجرى على المصفوفة عمودا عمودا. فمثلا طباعة عناصر الصف رقم i ستتم بكفاءة عالية ولكن طباعة عناصر العمود رقم j ستكون مكلفة (expensive) حيث أنها تتطلب عملية بحث (searching) في كل صف (عن العنصر الموجود في العمود رقم j).

حيز التخزين المستخدم (Storage used)

نفرض أن A مصفوفة m x n ، وأن عدد عناصرها غير الصفرية t ، وأن الحيز المستخدم لتخزين عدد صحيح يساوي الحيز المستخدم لتخزين عدد حقيقي يساوي الحيز المستخدم لتخزين مؤشر.

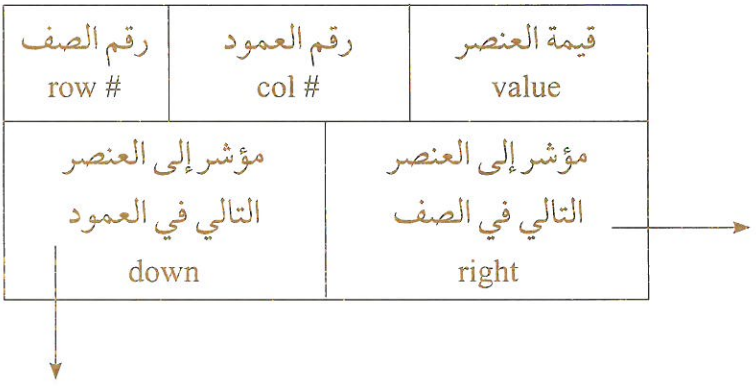
بناء على هذا فإن الحيز الكلي المطلوب للتخزين يساوي

storage = 3 t + m

حيث يتطلب كل عنصر من العناصر غير الصفرية (والتي عددها t) ثلاثة مجالات للتخزين ، ونحتاج m موضعا (حيث m هو عدد صفوف المصفوفة) لتخزين عناصر المنظومة xa .

القوائم المترابطة المتعامدة (Orthogonal Linked Lists)

ذكرنا أن تخزين عناصر المنظومة صفا صفا يؤدي إلى عدم سهولة إجراء العمليات عليها عمودا عمودا ، وللتغلب على هذا فإننا نود تخزين عناصر المصفوفة عمودا عمودا بالإضافة إلى تخزينها صفا صفا ، ويمكننا تحقيق ذلك بأن نحتفظ في سجل كل عنصر بمؤشرين : أحدهما right يشير إلى العنصر التالي (على اليمين) في الصف نفسه ، والآخر down يشير إلى العنصر التالي (إلى أسفل) في العمود نفسه ، وبالتالي تكون صيغة سجل أي عنصر كما يلي :



تعريف النوع

```
typedef node  *node_ptr ;
struct node
        {
                int row, col ;
                float value ;
                node_ptr   down, right ;
        } ;
typedef   node_ptr   pointer_array [100] ;
```

[وذلك بفرض أن 100 حد أعلى لكل من : m : عدد الصفوف و n : عدد الأعمدة]

pointer_array xrow, xcol ;

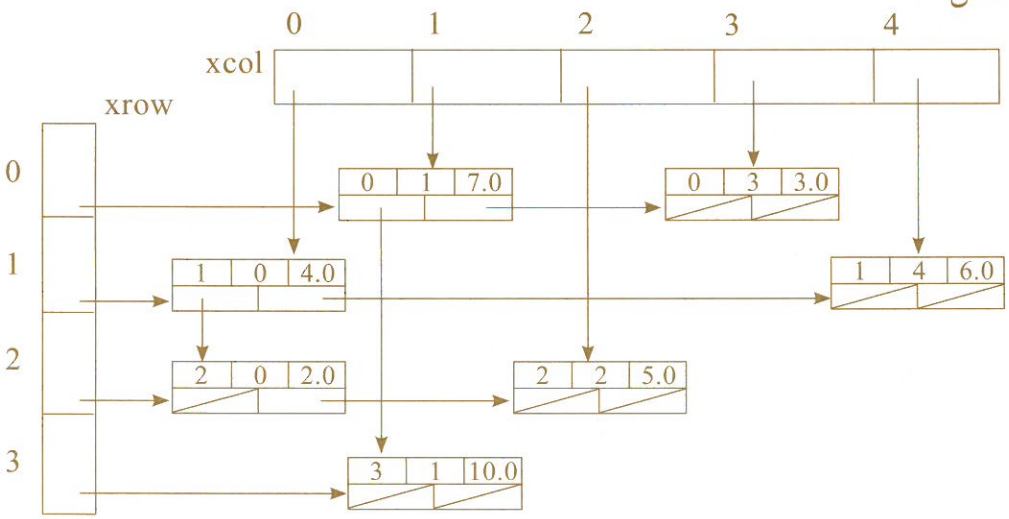
أي أن أي عنصر غير صفري في المصفوفة يرتبط بقائمتين : إحداهما على أساس رقم الصف ، والأخرى على أساس رقم العمود ، فمثلا بالنسبة للمصفوفة A_{4x5} السابقة يكون التمثيل المتعامد (orthogonal representation) كما يلي :

مثال ٤-١٠ :

ارسم التمثيل المتعامد للمصفوفة :

$$A_{4x5} = \begin{pmatrix} 0 & 7 & 0 & 3 & 0 \\ 4 & 0 & 0 & 0 & 6 \\ 2 & 0 & 5 & 0 & 0 \\ 0 & 10 & 0 & 0 & 0 \end{pmatrix} \quad ; \quad t = 7$$

الحل :



ومثل هذه البنية يطلق عليها أحد الاسمين

أ) بنية القوائم المتعامدة (orthogonal list structure)

لأن قوائم الصفوف تظهر في الشكل متعامدة مع قوائم الأعمدة.

ب) بنية القوائم المتعددة (multilist structure)

لأن كل عنصر يظهر في قائمتين في الوقت نفسه : قائمة الصف وقائمة العمود.

حيز التخزين المستخدم

نفرض للسهولة أن الحيز المستخدم لتخزين عدد صحيح يساوي الحيز المستخدم لتخزين عدد حقيقي / عائم يساوي ذلك المستخدم لتخزين مؤشر (= كلمة واحدة مثلا) ، ونفرض أن عدد العناصر غير الصفرية في المصفوفة A يساوي t .

s(integer) = s(real) = s(pointer) {= 1 word}

⇒ storage S(A) = m + n + 5*t

حيث نحتاج m موضعا لتخزين عناصر المنظومة xrow

و n موضعا لتخزين عناصر المنظومة xcol

و 5t موضعا لتخزين العناصر غير الصفرية في المنظومة ، وعدد هذه العناصر t وهناك خمسة مجالات في كل عنصر.

ونلاحظ أن هذا الحيز S(A) = m + n + 5*t أكبر من ذلك الحيز المستخدم في الطريقة السابقة والذي يساوي m + 3*t.

مثال عددي : إذا فرضا أن لدينا مصفوفة متناثرة فيها $m = n = 1000 = 10^3$ أي أنها مصفوفة 1000 x 1000 وبها خمسة عناصر غير صفرية في كل صف ، فإن التمثيل المعتاد بمنظومة ثنائية البعد 2D يتطلب حيزا يساوي $m \times n = 10^6$ ، أما التمثيل المتعامد فإنه يتطلب حيزا يساوي

$$m + n + 5*t = 10^3 + 10^3 + 5 \times 5 \times 10^3 = 27 \times 10^3$$

أي أن النسبة بين الحيزين تساوي $\frac{27}{10^3} = 2.7\ \%$

ملاحظة :

لا معنى لاستخدام هذا التمثيل المتعامد إذا كانت المصفوفة كثيفة (dense).

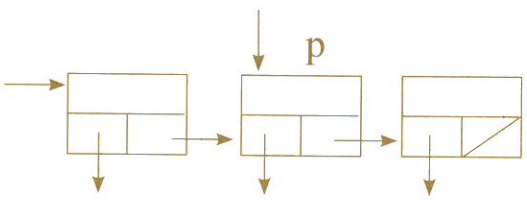
العمليات على المصفوفات ذوات بنية القوائم المتعامدة

* بناء هذه البنية (building the structure) ، أي تخزين / قراءة عناصر المصفوفة (read / input the matrix).
* طباعة عناصر المنظومة (مثلا صفا صفا أو عمودا عمودا).
* تغيير / طباعة / استعادة عنصر معين a_{ij} (change / print / retrieve).
* محو (erasing) مصفوفة.
* جمع مصفوفتين $C = A + B$
* ضرب مصفوفتين $C = A * B$
* تدوير مصفوفة

مثال ٤-١١ :

اكتب إجراء لطباعة صف (يشير إليه المؤشر r) من مصفوفة A بفرض تمثيل المصفوفة بقائمة مترابطة متعامدة.

الحل :

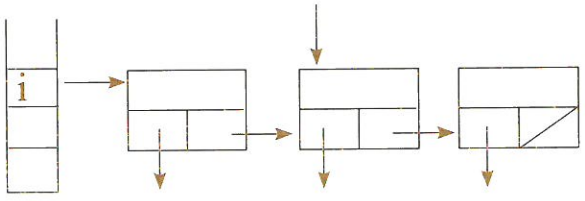


```
void print_row (node_ptr r) ;
{
        node_ptr  p ;
        p = r ;
        while (p != NULL)
                {
                        cout << p → row << p → col << p → value ;
                        p = p → right ;
                }
}
```

فمثلا لطباعة عناصر الصف رقم i نستدعي هذا الإجراء هكذا :

```
print_row (xrow [i])
```

أي أننا نجتاز traverse قائمة الصف (رقم i) التي يشير إليها المؤشر [r = xrow [i] باستخدام المؤشر right.



ولطباعة جميع عناصر المصفوفة صفا صفا نكتب قطعة البرنامج التالية .

مثال ٤-١٢ :

اكتب خوارزمية لطباعة جميع عناصر المصفوفة A المذكورة في المثال السابق.

الحل :

```
for (i = 0 ; i <= m-1 ; i + +)
        if (xrow [i] = = NULL)
```

```
            cout << " row no. " << i << " is empty " << endl ;
        else
            print_row (xrow [i])
```

وبالمثل يمكننا طباعة عمود من المصفوفة أو طباعة جميع عناصر المصفوفة عمودا عمودا.

ملاحظة : يمكننا وضع الشرط أن الصف خالي (ليس به عناصر) داخل الإجراء نفسه هكذا :

```
void print_row (node_ptr r) ;
{
        node_ptr p ;
        if (r = = NULL)
            cout << " empty row " << endl ;
        else
            {
                p = r ;
                while (p != NULL)
                {
                    cout <<p→row<<p→col<<p→value ;
            p = p → right
                }
            }
}
```

وذلك بدلا من وضع هذا الشرط (أن الصف خالي) في قطعة البرنامج التي تستدعي الإجراء.

درجة تعقيد الإجراء Complexity of procedure print_row

واضح أن درجة التعقيد تساوي عدد العناصر غير الصفرية في الصف .

مثال ٤-١٣ :

اكتب دالة لاسترجاع قيمة العنصر a_{ij} في مصفوفة مخزونة بالتمثيل المتعامد بحيث تعيد الدالة قيمة العنصر إن وجدته أو تعيد صفرا إن لم تجده .

الحل :

نبحث في الصف رقم i عن العمود رقم j ،
أو نبحث في العمود رقم j عن الصف رقم i ،
فإذا وجدنا العنصر أعدنا قيمته وإلا أعدنا صفرا

```
float retrieve (node_ptr r, int i, j, bool& found)

{ // we search in row no. i. Function returns aij if it is stored,
  // else it returns zero. found = true  if node exists,
  // false otherwise. Functon uses sequential search to
  // search for col. no. j, in each node in the list of row no. i}
float val ;
node_ptr p ;
p = r ;
found = false ;
while ((p != NULL) && (! found))
        if (p → col = = j)
                found = true
        else
                p = p → right ;
if found
```

```
        val = p → value;
else
        val = 0.0;
return val ;
}
```

ولاستدعاء الدالة **retrieve** لطباعة قيمة العنصر a_{ij} نكتب :

```
cout << retrieve (xrow [i], i, j, found) << endl;
```

ملاحظة : من الأفضل والأكفأ أن نستخدم البحث التتابعي المعدل (modified sequential search) الذي يخرج من عروة while بمجرد أن يصل في البحث التتابعي إلى عمود رقمه ≤ j ، وذلك لأن أرقام الأعمدة – في أي صف – تزايدية .

كيفية بناء المصفوفة المتناثرة في صورة قائمة مترابطة متعامدة

(How to build a sparse matrix structure)

أولا : الإعداد للبدء / إعطاء القيم الابتدائية (Initialization)

ننشئ أولا مصفوفة n x n فارغة :

```
xrow [i] = NULL ;        i = 0, 1, 2, ..., m-1
xcol [j] = NULL ;        j = 0, 1, 2, ..., n-1
```

ثانيا : إدخال العناصر (insertion of nodes)

نقوم بعد ذلك بتكرار إدخال / إضافة العناصر / العقد الجديدة (new nodes) ، حيث كل عنصر عبارة عن ثلاثية (i , j , value). ويجب إدخال العنصر في مكانه الصحيح بالنسبة لصفه وكذلك بالنسبة لعموده.

```
while more input exists
    {
        cin >> i >> j >> value ;
        insert (i , j , value) in its right place
            in row # i & col # j of the structure
    }
```

ونلاحظ في عملية إدخال الثلاثية (i , j , value) ما يلي :

١ . إن لم يسبق إدخال الزوج i , j من قبل ، أي إن لم يكن هناك عنصر في المصفوفة سبق إدخاله حيث row = i , col = j فإننا نقوم بإنشاء عنصر جديد (make a new node) (لهذا الزوج i , j) مرة واحدة فقط ، ثم نربط هذا العنصر الجديد برباطين أحدهما تبعا لصفه والآخر تبعا لعموده.

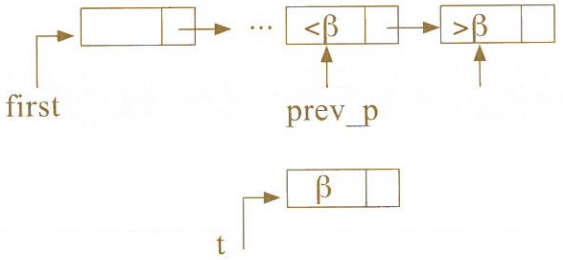
٢ . أما إن سبق إدخال الزوج i , j من قبل ، أي هناك عنصر في المصفوفة سبق إدخاله حيث row = i , col = j ، فإننا نجمع عدديا (numerically add) القيمة الجديدة (new value) الداخلة الآن مع القيمة الموجودة (existing value) التي سبق إدخالها ، بفرض أن هذا المجموع سوف لا يساوي صفرا (numerical cancellation) ، حيث أننا نقوم بتخزين العناصر غير الصفرية فقط.

٣ . يمكننا التحقق من قيم عناصر المصفوفة بعد إدخالها بحيث نحذف العناصر / الحدود ذات المعاملات الصفرية (zero value coefficients) .

٤ . نسمح بإدخال الثلاثيات (i , j , value) بأي ترتيب (any order) ، وهو ما يحدث فعلا في الواقع العملي.

٥ . في تمثيلنا المتعامد نلاحظ أنه :
في أي صف : أرقام الأعمدة متزايدة ، وكذلك
في أي عمود : أرقام الصفوف متزايدة
ولذلك فعند إدخال / ربط (insert / link) عنصر جديد نحتاج لمعرفة مكانه الصحيح بالنسبة لرقم الصف وبالنسبة لرقم العمود.

٦ . راجع كيفية إدخال عنصر في قائمة مترابطة مرتبة (*) ، وأننا نستعين بمؤشر إضافي prev_p – كما هو موضح بالشكل – ليساعدنا في إضافة عنصر قبل العنصر p ، حيث يشير المؤشر prev_p إلى العنصر السابق للعنصر p .



وفي هذه الحالة [حالة استخدام مؤشرين : prev_p ، p] يكون الإدخال كما يلي :

```
if (prev_p != NULL)
        {
                t → next = prev_p → next;    // = = p
                prev_p → next = t ;
        }
else
```

(*) انظر مثلاً فصل : البنيات المترابطة ، كتاب : البرمجة المتقدمة بلغة C++ ، أبو بكر السيد ، دار القلم ، الكويت ٢٠٠٣ .

```
{
        t → next = first ;
    first = t ;
}
```

٧ . وفي مصفوفتنا المتناثرة نحتاج لإجراء الخطوات السابقة مرتين : مرة بالنسبة للصف ومرة أخرى بالنسبة للعمود .

ولذلك سنكتب بإذن الله إجراءين :

* الإجراء الأول **row_search** يبحث في صف معين (i مثلا أو r) عن الموضع الصحيح للعنصر تبعا لرقم عموده j ، ونستعين بمؤشرين :

* row_p : مؤشر يشير في النهاية إلى العنصر - في هذا الصف المعين - الذي رقم عموده ≤ j

* prev_right : مؤشر يشير في النهاية إلى العنصر - في هذا الصف المعين السابق مباشرة للعنصر الذي يشير إليه المؤشر p_wor .

وفي النهاية يتم إدخال العنصر الجديد بين هذين المؤشرين .

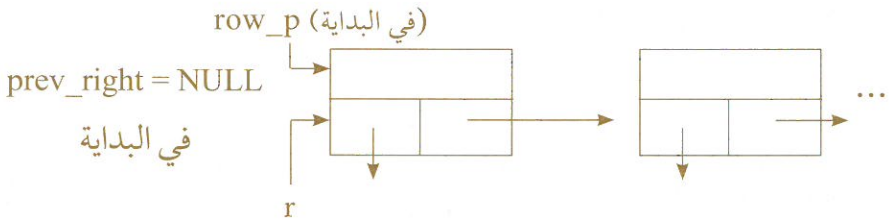
* الإجراء الثاني **col_search** : وظيفته مماثلة للإجراء **row_search** ولكن بالنسبة لعمود معين .

مثال ٤-١٤ :

اكتب إجراء **row_search** يبحث في صف من منظومة معرف بالمؤشر r عن الموضع الصحيح لعنصر معلوم رقم عموده j ، بحيث يعيد الإجراء مؤشرين : أحدهما إلى العنصر " e " - في هذا الصف المعين - الذي رقم عموده ≥ j ، والآخر إلى العنصر - في هذا الصف المعين - السابق مباشرة للعنصر e .

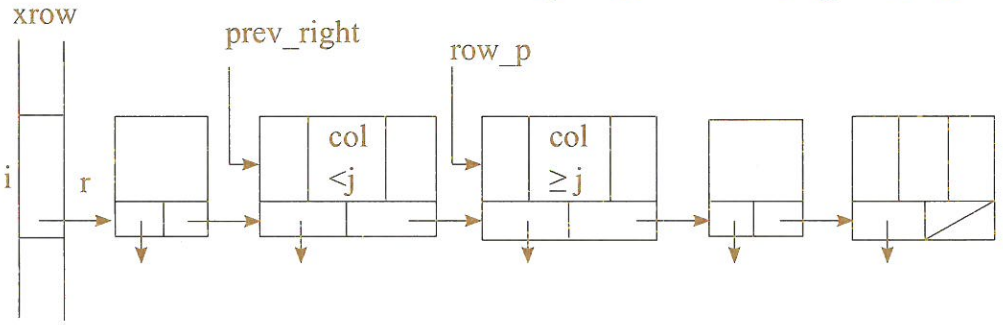
الحل :

```
void row_search (node_ptr r,   // ptr to the first node in the row
                 int j,
                 node_ptr& row_p , prev_right ,
                 bool found)
{
        bool done ;
        row_p = r ;
        prev_right = NULL ;
        found = false ;   done = false ;
        while ((row_p != NULL) && (! done))
              if (row_p → col >= j)
                    {
                          done = true ;
                          found = (row_p → col = = j) ;
                    }
               else    // advance right
                    {
                          prev_right = row_p ;
                          row_p = row_p → right ;
                    }
}
```

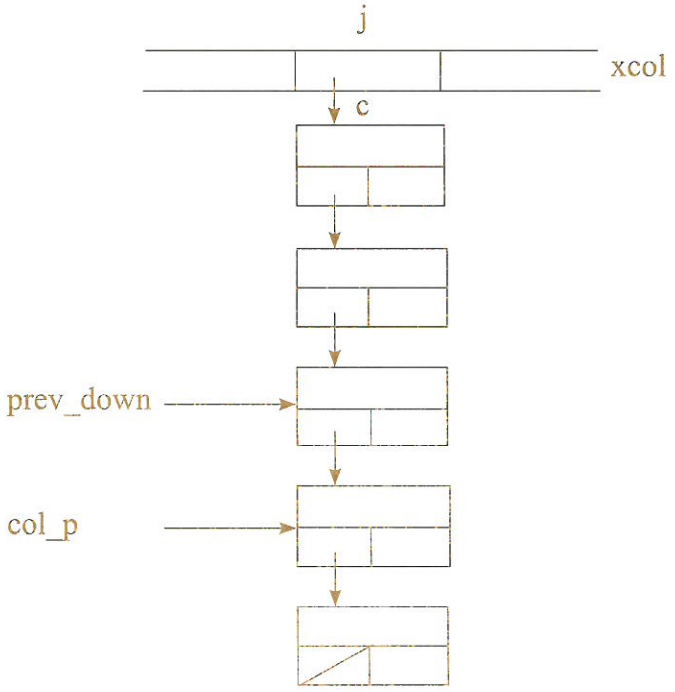


وحين ينتهي تنفيذ هذا الإجراء **row_search**

نصل إلى وضع يوضحه الشكل التالي :



وأما الإجراء **col_search** فهو شبيه جدا بالإجراء **row_searh** مع تبديل دوري الصف والعمود .

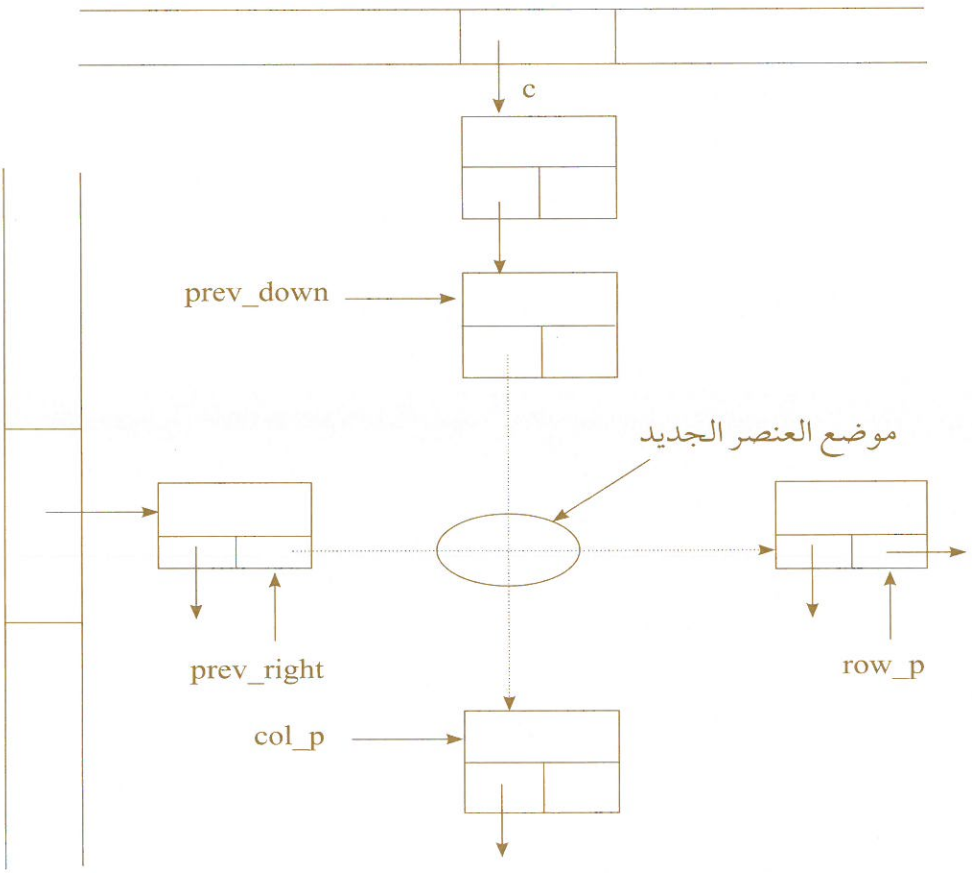



```
void col_search (node_ptr c,   // ptr to the first node in the column
        int i, node_ptr& col_p, prev_down,
        bool& found)
```

والآن بعد تحديد الأربعة مؤشرات

row_p , prev_right , col_p , prev_down

يمكننا بسهولة إدخال العنصر في موضعه الصحيح وربطه بالعناصر التي قبله / بعده في صفه وعموده .



وفيما يلي إجراء إدخال / إضافة العنصر الجديد في موضعه الصحيح . ونعطي أولا جزء تعريف الإجراء ووسطائه ، وكيفية استدعائه من البرنامج الرئيسي لإدخال جميع العناصر (غير الصفرية) في المصفوفة . ثم نعطي بعد ذلك الإجراء كاملا .

```
void matrix_insert (node_ptr& r, c,
      // r : pointer to row linked list , pointer to the insert row
      // c : pointer to column linked list
          int i, j;     // i : row # , j : col #.
          float x       // x : value of the elt.)
      // to insert the triple (i , j , x) in proper row / col.
```

ويمكن استدعاء هذا الإجراء كما يلي لإدخال كافة العناصر :

```
create_empty_matrix ;
while (! eof)
        {
              cin >> i >> j >> value >> endl ;
              matrix_insert (xrow[i], xcol[j], i, j, value)
        }
```

وفيما يلي الإجراء **matrix_insert** كاملا :

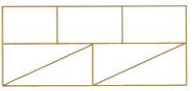
مثال ٤-١٥ :

اكتب إجراء **matrix_insert** لإدخال عنصر قيمته x ورقم صفه i ورقم عموده j ، وذلك في منظومة ممثلة بالتمثيل المتعامد {حيث r : مؤشر إلى صف الإدخال ، c : مؤشر إلى عمود الإدخال } .

الحل

```
void matrix_insert (node_ptr& r, c,
      // r : pointer to row linked list , pointer to the insert row
      // c : pointer to column linked list.
        int i, j     // i : row # , j : col #.
        float x)
{// to insert the triple (i , j , x) in proper row / col.
        node_ptr       t , row_p , prev_right, col_p , prev_down ;
```

```
bool found ;

row_search (r , j , row_p , prev_right , found) ;
if found
        // modify the value , add numerical value.
        row_p → value = row_p → value + x ;
else
        {
          col_search (c , i , col_p , prev_down, found);
           // create a new node.
          make_new_node (t , i , j , x) ;
          // link the new node in the row , then
          // later in the column.
          if (prev_right = = NULL)
                // the new node is the first in its row.
              {
                    t → right = r ;
                    r         = t ;
              }
          else    // insert after prev_right.
              {
                    t → right = row_p ;
                          // = = prev_right → right
                    prev_right → right = t ;
              }
          // Similarly, link in the column.
          if (prev_down = = NULL)
                // the new node is the first in its col.
              {
                    t → down = c ;
```



```
                    c          = t ;
                }
            else
                {
                    t → down = col_p ;
                    prev_down → down = t
                {
        {
{
```

درجة تعقيد بناء المصفوفة Complexity of matrix construction

درجة تعقيد الإجراء Complexity of matrix_insert

عند بناء المصفوفة [أي إعدادها للبدء (إعطاء القيم الابتدائية) ثم إدخال جميع العناصر (غير الصفرية) واحدا تلو الآخر] فإننا نستدعي – من البرنامج الرئيسي – تكراريا (repeatedly) الإجراء **matrix_insert** لإدخال العناصر عنصرا عنصرا .

وحيث أن هذا الإجراء يستخدم كلا من الإجراءين **row_search & col_search** فإن تكاليف (cost) تنفيذ الإجراء **matrix_insert** تساوي مجموع تكاليف تنفيذ هذين الإجراءين (+ تكاليف باقي عبارات **matrix_insert**) .

Cost = cost of row_search + cost of col_search + const

وحيث أنه من المعلوم أن البحث في قائمة (مترابطة) يتكلف (في أسوأ حالة) طول القائمة

(cost of search = length of the list to search)

فإن البحث في صف (عمود) معين يتكلف (على الأكثر) طول هذا الصف (العمود)

cost of searching a row (col) = at most lengh of the row (col)

فإذا فرضنا أن طول الصف رقم i (أي عدد عناصره غير الصفرية) يساوي ℓ_i وأن طول العمود رقم j (أي عدد عناصره غير الصفرية) يساوي k_j ،

أي أن أطوال الصفوف هي : $\ell_1 , \ell_2 , ..., \ell_i , ..., \ell_m$

وأن أطوال الأعمدة هي : $k_1 , k_2 , ... , k_j , ... , k_n$

فإن :

البحث في الصف رقم i يتكلف على الأكثر ℓ_i

والبحث في العمود رقم j يتكلف على الأكثر k_j

ولإدخال insert جميع عناصر الصف رقم i تكون التكاليف كما يلي :

{لاحظ أنه : عند إدخال العنصر الأول يكون طول الصف صفرا ،

وعند إدخال العنصر الثاني يكون طول الصف ١ ،

وعند إدخال العنصر الثالث يكون طول الصف ٢ ،

وهكذا.}

تكاليف إدخال العنصر الأول : 1

تكاليف إدخال العنصر الثاني : 2

تكاليف إدخال العنصر رقم ℓ_i : ℓ_i

$$\text{cost} = 1 + 2 + 3 + ... + \ell_i$$

$$= \frac{\ell_i (\ell_i + 1)}{2} \cong \frac{\ell_i^2}{2}$$

وبالتالي تكون تكاليف إدخال / بناء جميع عناصر جميع الصفوف (أي جميع عناصر المصفوفة)

$$\text{cost}\Big|_{\text{building all rows}} \leq \frac{\ell_1^2}{2} + \frac{\ell_2^2}{2} + ... + \frac{\ell_m^2}{2}$$

{ ≤ : لأن هناك في الإجراء عبارات أخرى }

$$= \frac{1}{2} \left(\ell\frac{2}{1} + \ell\frac{2}{2} + \ldots + \ell\frac{2}{m} \right)$$

$$\leq \frac{1}{2} \left(\ell_{max} \ \ell_1 + \ell_{max} \ \ell_2 + \ldots + \ell_{max} \ \ell_m \right)$$

(ℓ_{max} : طول أطول صف)

$$= \frac{\ell_{max}}{2} \left(\ell_1 + \ell_2 + \ldots + \ell_m \right)$$

$$= \frac{\ell_{max}}{2} * t = O \left(\ell_{max} * t \right)$$

حيث t : عدد العناصر غير الصفرية

[لأن مجموع أطوال الصفوف = عدد جميع العناصر غير الصفرية ، لأن طول أي صف = عدد عناصره (غير الصفرية)]

وبالمثل تكاليف بناء جميع الأعمدة

$$\text{cost} \left| \text{building all columns} = O \left(k_{max} * t \right) \right.$$

وحيث أنه في المصفوفة المتناثرة يكون عدد العناصر غير الصفرية صغيرا (أي أن ℓ_{max} أو k_{max} عدد صغير) ، فبالتالي نحصل على العلاقة :

$$\text{cost} \left| \text{building the matrix} = O \left(t \right) \right.$$

حيث t : عدد العناصر غير الصفرية في المصفوفة .

مثال ٤-١٦ :

اكتب إجراء لحذف العنصر a_{ij} من مصفوفة ممثلة في صورة قائمة مترابطة متعامدة .

الحل :

```
void delete_matrix (node_ptr& r, c, int i, j)
    {
    node_ptr row_p, prev_right, col_p, prev_down) ;
    bool found ;
    row_search (r, j, row_p, prev_right , found) ;
    if (! found)
            cout << " element does not exist " << endl ;
    else
            {
                col_search (c, i, col_p, prev_down, found) ;
                // break row links
                if (prev_right = = NULL)
                        // delete first node in the row.
                        r = r → right ;
                else
                        prev_right → right = row_p → right ;
                // break col links.
                if (prev_down = = NULL)
                        c = c → down ;
                else
                        prev_down → down = col_p → down ;
                // now delete the node.
                delete row_p ;
        {
{
```

بناء قائمة مترابطة مرتبة بناء على مجالين مختلفين

linked list ordered on 2 different keys (fields)

مثال ٤-١٧ :

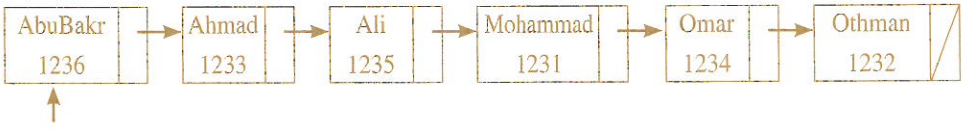
وضِّح كيفية بناء قائمة مترابطة تمثل قائمة سجلات طلاب ، حيث يشتمل السجل على اسم الطالب : name ورقم الطالب : id ، بحيث تكون السجلات مرتبة بناءً على كل من اسم الطالب ورقم الطالب ، أي مرتبة أبجديا بالنسبة لأسماء الطلاب ، ومرتبة تصاعديا بالنسبة لأرقام الطلاب .

الحل :

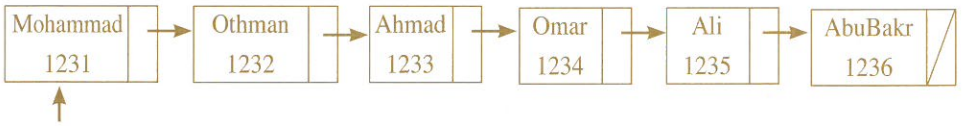
نفرض أن لدينا مثلا مجموعة البيانات التالية لأسماء بعض الطلاب وأرقامهم :

اسم الطالب name	رقم الطالب id
Omar	1234
Ali	1235
Ahmad	1233
Mohammad	1231
Othman	1232
AbuBakr	1236

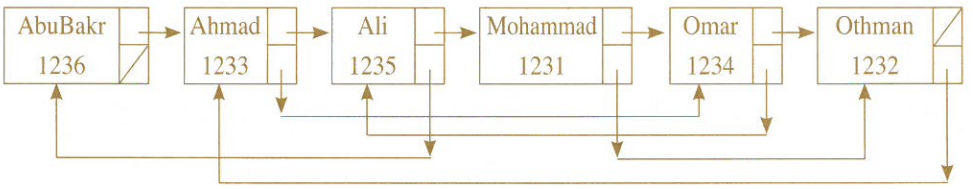
إذا أردنا تخزين القائمة بحيث تكون مرتبة (ordered / sorted) أبجديا أي مرتبة بالنسبة للأسماء فإنها تكون هكذا :



وإذا أردنا تخزينها بحيث تكون مرتبة بناء على رقم الطالب فإنها تكون هكذا :



فإذا أردنا تخزينها بحيث تكون مرتبة بناء على كل من الاسم ورقم الطالب فإننا نحتاج إلى مؤشر آخر :



أي أننا نكوِّن قائمة فيزيائية واحدة (one physical list) ولكن عناصرها (nodes) مترابطة في قائمتين منطقيتين (linked in 2 logical lists) إحداهما مرتبة بناء على الاسم (sorted by name) والأخرى مرتبة بناء على الرقم (sorted by id) .

ملاحظة : كل عنصر (node) يتم إنشاؤه مرة واحدة فقط (created once) ، ولكنه يُدخل / يُربط (inserted / linked) مرتين : مرة في قائمة أسماء والأخرى في قائمة أرقام طلاب . وهناك طرق أخرى لبناء مثل هذه القائمة .

Linked Lists 4

تمرينات رقم E

٤-١ نفرض أن لدينا التعريفات التالية :

```
typedef node  *poly ;
struct node
        {
                int coef ;
                int exp ;
                poly next ;
        } ;
poly A, B, C, L ;
```

(أ) نفرض أننا سنقوم باستدعاء الإجراء poly_add لجمع الحدوديتين

$A(x) = 2 + 4\,x^{8} + x^{12}$;

$B(x) = 3\,x^{5} - 14\,x^{8} - x^{12} + x^{20}$;

والحصول على الحدودية

$C(x) = A(x) + B(x)$

ارسم أشكالا تخطيطية للقوائم المترابطة الثلاث A , B , C التي يتم فيها تخزين الحدوديات الثلاث A , B , C

(ب) نفرض أنه قد تم تخزين الحدودية السابقة A(x) في قائمة مترابطة A ، وأن L مؤشر يشير إلى آخر عنصر من عناصر القائمة A .

(i) تتبع ووضح بالرسم نتائج الاستدعاء Test (20, 18, A، L)
حيث **Test** هو الإجراء التالي :

```
void Test (int c, e, poly& f, last) ;
{
        poly p ;
        p = new node ;
        p → coef = c ;
        p → exp = e ;
        p → next = NULL ;
        if (p = = NULL)
                {
                        f = p ;
                        last = p ;
                }
        else
                {
                        last → next = p ;
                        last = p
                }

}
```

(ii) ما وظيفة الإجراء **Test** ؟

(ج) نفرض أن لدينا الإجراء **Exam** التالي :

```
void Exam (poly&  first) ;
{
        poly p ;
        while (first != NULL)
                {
                        p = first → next ;
```

```
            delete first ;
            first = p ;
        }
    }
```

(i) تتبع ووضح بالرسم نتائج تنفيذ الاستدعاء **Exam(A)** حيث A هي القائمة المترابطة المعرفة سابقا في (أ) .

(ii) ما وظيفة الإجراء **Exam** ؟

(iii) اكتب إجراء

void multiply_const (poly& D, int c)

لضرب حدودية D من النوع المعرف سابقا poly بعدد صحيح c .

٤-٢ نفرض أننا عند تخزين أي مصفوفة سنستخدم طريقة تمثيلها بقائمة مترابطة متعامدة (orthogonal linked list representation) .

١) اكتب تعريف نوع مناسب لهذا التمثيل المتعامد .

٢) ارسم التمثيل المتعامد للمصفوفة $\begin{bmatrix} -1 & 0 & -2 \\ -3 & 0 & 0 \\ 0 & 0 & -4 \end{bmatrix}$

٣) نفرض أن A مصفوفة n x n فيها ٧ عناصر غير صفرية في كل صف . كم سعة التخزين المطلوبة لتخزين المصفوفة A باستخدام هذا التمثيل المتعامد ؟

٤) اكتب إجراء لضرب عناصر العمود رقم j في مصفوفة متناثرة m x n بعدد ثابت غير صفري c (nonzero constant) .

كم تساوي درجة تعقيد هذا الإجراء ؟

هـ) اكتب دالة لحساب قيمة الضرب العددي / الداخلي / النقطي (dot product) للعناصر المتقابلة في الصف رقم i والصف رقم j في مصفوفة متناثرة .

كم تساوي درجة تعقيد هذه الدالة ؟

٤-٣ أ) اكتب دالة تعيد مؤشرا (pointer) إلى العنصر (node) الذي يسبق مباشرة عنصرا معينا q في قائمة مترابطة مفردة (single linked list) وإذا كان q هو أول عنصر في القائمة فإن الدالة تعيد مؤشر التلاشي NULL .

ب) اكتب تعريف نوع مناسب لقائمة مترابطة مزدوجة / ثنائية الارتباط (double linked list) .

جـ) أعد حل الجزء (أ) من السؤال بالنسبة لقائمة مترابطة مزدوجة .

د) اكتب إجراء لحذف عنصر معين q (node) من قائمة مترابطة مزدوجة .

٤-٤ الطابور ذو النهايتين «deque» (double ended queue) هو قائمة list يمكن أن تتم فيها الإضافة أو الحذف deletion / insertion عند أي من النهايتين .

أ) اكتب تمثيلا representation مناسبا لهذه البنية (باستخدام القوائم المترابطة) .

ب) اكتب إجراء لإضافة عنصر عند أي من النهايتين (بناء على اختيار المستخدم) .

٤-٥ أ) اكتب إجراء لاجتياز قائمة دائرية وطباعة محتوياتها (ابتداءً من العنصر الأول) .

ب) اكتب دالة صحيحة لإيجاد طول قائمة دائرية .

جـ) اكتب إجراء لإلحاق قائمة دائرية بنهاية قائمة دائرية أخرى .

٤-٦ أ) اكتب خوارزمية لإدخال / إضافة (inserting) عنصر q عند نهاية قائمة دائرية مترابطة ، حيث تحتوي القائمة على مؤشر خارجي واحد L (one external pointer) يشير إلى آخر عنصر في القائمة .

ب) اكتب خوارزمية لحذف (deleting) العنصر الأول في قائمة دائرية مترابطة .

٤-٧ اكتب إجراء لحذف عنصر معين x من قائمة دائرية مترابطة L ، حيث المؤشر L يشير إلى آخر عنصر (منطقي) في القائمة ، بفرض أن القائمة بها أكثر من عنصر واحد .

٤-٨ اكتب خوارزمية لإدخال عنصر q قبل عنصر p (حيث p != NULL) في قائمة مترابطة مزدوجة / ثنائية الارتباط .

٤-٩ نفرض أن t(x) حدودية معطاة كقائمة مترابطة من النوع poly . اكتب إجراء لمحو الحدودية t وإخلاء (disposing) الذاكرة المستخدمة لتمثيل الحدودية t ، أي إخلاء جميع عناصر / عقد t (free up the nodes in t) عنصرا عنصرا لإمكانية استخدام هذه العناصر لتمثيل حدوديات أخرى .

٤-١٠ اكتب خوارزمية لنسخ (copy) مصفوفة متناثرة ممثَّلة بقائمة مترابطة متعامدة وأوجد درجة التعقيد الزمنية (time complexity) لهذه الخوارزمية .

٤-١١ اكتب برنامجا يعمل بنظام قائمة التعليمات (menu driven program) لإجراء العمليات التالية على المصفوفات المتناثرة المربعة n x n ، حيث تمثَّل المصفوفة بقائمة مترابطة متعامدة :

أ) إدخال بيانات مصفوفة متناثرة ، حيث البيانات في الصورة التالية :

السطر الأول به رتبة المصفوفة : n ، وعدد عناصرها غير الصفرية :t .

كل سطر من السطور التالية به ثلاثية (row ، col ، value) تمثل عنصرا غير صفري ، وتدخل هذه الثلاثيات بأي ترتيب .

ب) طباعة المصفوفة المتناثرة .

ج) جمع مصفوفتين .

د) ضرب مصفوفتين .

هـ) محو مصفوفة .

الفصل الخامس

الأشجـــار
Trees

5

Trees 5

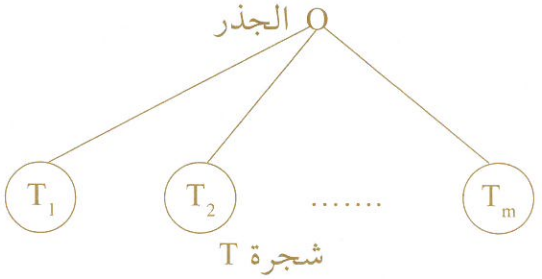
الفصل الخامس

الأشجـــــار
Trees

نبدأ أولا بذكر ملخص لبعض التعريفات الخاصة بالأشجار ..

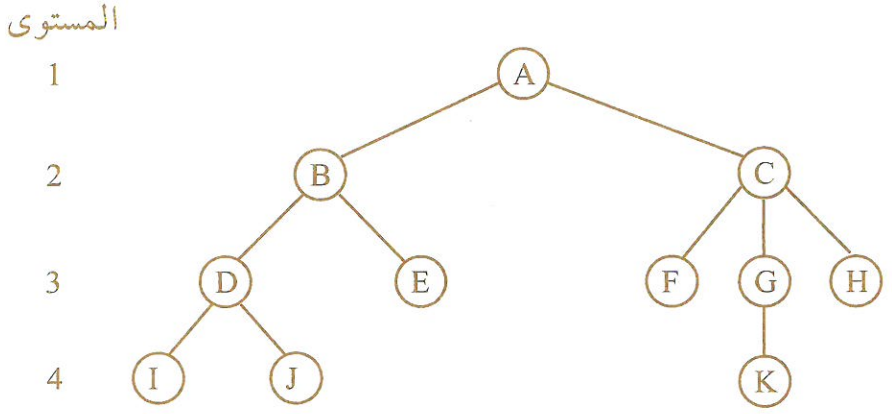
تعريفات :

١) **الشجرة (tree)** : هي مجموعة محدودة (finite set) مكونة من عنصر واحد أو أكثر (one or more nodes) ، من بينها عنصر خاص (special node) يطلق عليه **الجذر (root)** ، وباقي العناصر تُقَسَّم/ تُجَزَّأ (partitioned) إلى مجموعات متباعدة (disjoint sets) $T_1, T_2, \ldots, T_m$ [أي أن $T_i \cap T_j = \phi, i \neq j$] وكل مجموعة منها تعد شجرة ، ويطلق عليها «**شجرة فرعية**» **(subtree)** للجذر R . وجذور الأشجار الفرعية تسمى **توابع (successors of) R**. والعنصر الذي له توابع يطلق عليه «**والد**» **(parent)** هذه التوابع ، وهي تسمَّى **أبناؤه (children)**.



شجرة T

مثال : m = 2



[ملاحظة : الشجرة في الحقيقة هي شكل بياني / مخطط بياني (graph) بدون دورات (without cycles) ، وهي أيضا تسمى **شجرة مرتبة موجَّهة (oriented ordered tree)**]

٢) **العناصر الداخلية (internal nodes)** :

العنصر الداخلي هو العنصر الذي له أبناء / شجرة فرعية (children / subtree)

مثال : العناصر A , B , C , D , G (في الشكل السابق) عناصر داخلية.

٣) **الأوراق (leaves) / العناصر الطرفية (terminal nodes)** :

الورقة / العنصر الطرفي : هو العنصر الذي ليس له أبناء

مثال : العناصر I , J , E , F , K , H

٤) **درجة العنصر (degree of a node)** :

درجة العنصر هي عدد الأشجار الفرعية المتفرعة / الخارجة من العنصر (# of subtrees of the node)

مثلا :

deg (A) = 2 , deg (C) = 3 , deg (I) = 0

deg (G) = 1

٥) درجة الشجرة (degree of a tree) :

هي أكبر قيمة من قيم درجات عناصر الشجرة

$$\deg(\text{Tree}) = \max \deg(\text{node})|_{\text{in tree}}$$

مثلا : بالنسبة للشجرة السابقة

$$\deg(\text{Tree}) = \deg(\text{node C}) = 3$$

٦) الوالد (parent) ، والولد (child) :

العنصر N (node) الذي له توابع (successors) يطلق عليه : الوالد / الأصل / المصدر / السلف / الجد الأعلى (ancestor) لهذه التوابع ، وكل واحد منها يطلق عليه : ولد / ابن / فرع / خلف / سليل / حفيد (descendant) لهذا العنصر N.

٧) اصطلاح المستويات (Notion of levels) :

أ) مستوى أي عنصر = ١ + مستوى والد العنصر

مستوى الجذر = ١.

{ في اصطلاحات أخرى يعتبر مستوى الجذر صفرا وليس واحدا}

$$\text{level (node)} = 1 + \text{level (parent)}$$

$$\text{level (root)} = 1$$

ب) ارتفاع / عمق الشجرة (height / depth of a tree)

هو أكبر قيمة من قيم مستويات عناصرها

$$\text{height (tree)} = \max \text{level (node)}|_{\text{in tree}}$$

مثلا : ارتفاع الشجرة السابقة = ٤ لأن :

$$h \equiv \text{height (tree)} = \text{level (I)} = \text{level (J)} = \text{level (K)} = 4$$

٨) الغابة (Forest) :

هي مجموعة من الأشجار المتباعدة

(a set of disjoint trees)

غابة

ملاحظة : الشجرة هي غابة ذات شجرة واحدة.

وأكثر أنواع الأشجار شيوعا وفائدة من الناحية العملية : الأشجار الثنائية.

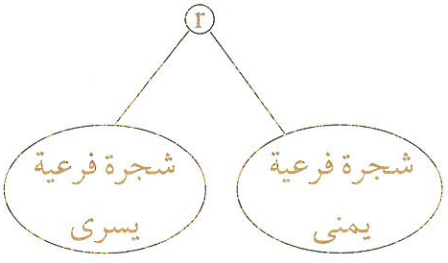
الأشجار الثنائية

Binary Trees (B.T.)

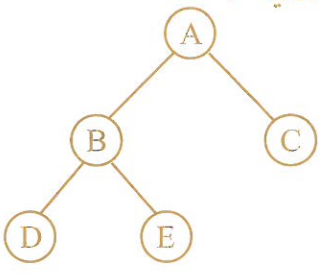
5

تعريف : الشجرة الثنائية (Binary Tree) :

هي مجموعة محدودة من العناصر (finite set of nodes) إما أن تكون خالية (empty) أو تتكون من جذر وشجرة فرعية يسرى(left subtree) وشجرة فرعية يمنى(right subtree) حيث كل منهما شجرة ثنائية (binary tree) .



مثلا : الشكل التالي يمثل شجرة ثنائية :



ملاحظتان :

أ) الشجرتان

تعتبران شجرتين ثنائيتين مختلفتين (different B.T's) ولكنهما تعتبران شجرتين متكافئتين (equivalent trees).

ب) الشجرة  لا تعد شجرة ثنائية (لا يمكن اعتبارها شجرة ثنائية) ولكنها تقبل كشجرة عادية (regular tree)

بعض العلاقات الخاصة بالأشجار الثنائية
Relations about binary trees

نفرض أن n : عدد عناصر شجرة ثنائية (# nodes)

و h : ارتفاع الشجرة (height)

(أ) عدد العناصر في أي مستوى i لا يتجاوز 2^{i-1} حيث ... i = 1, 2,

$$[\ \#\ \text{nodes in level i}\ ] \leq 2^{i-1}$$

$$\boxed{\text{No (nodes , level i)} \leq 2^{i-1}}$$

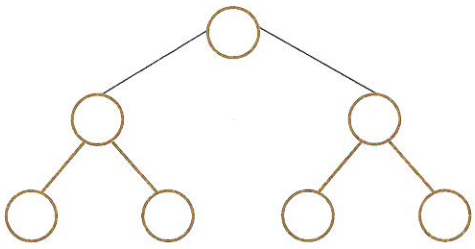
مثلا

$$\text{No (nodes , 3)} \leq 2^2 = 4$$

البرهان :

الجدول التالي يعطي العلاقة بين عدد العناصر والمستوى عندما يكون عدد العناصر أكبر ما يمكن (الشجرة ممتلئة ، أي كل عنصر يتفرع منه عنصران) ، ومن الجدول يسهل استنتاج العلاقة المطلوبة.

عدد العناصر		المستوى
# of nodes		level
1		1
2		2
4		3
8		4
:		:



(ب) عدد العناصر n في شجرة بحث ثنائية ارتفاعها h يحقق العلاقة

$$n \leq 2^h - 1$$

البرهان :

$$n = \sum_{i=1}^{h} \text{No (nodes, level i)}$$

{مجموع عدد العناصر في المستويات المختلفة كلها}

$$\leq \sum_{i=1}^{h} 2^{i-1}$$

$$= 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{h-2} + 2^{h-1}$$

$$= 1 + 2^1 + 2^2 + \quad \ldots + 2^{h-2} + 2^{h-1}$$

$$= S \quad \text{مثلا}$$

$$S = 1 + 2^1 + 2^2 + \ldots + 2^{h-2} + 2^{h-1}$$
$$2S = 2^1 + 2^2 + 2^3 + \ldots + 2^{h-1} + 2^h$$

بالطرح

$$\Rightarrow \quad S = 2^h - 1$$
$$\Rightarrow \quad n \leq 2^h - 1$$

وهو المطلوب.

من الواضح أنه بالنسبة لأي شجرة ثنائية ارتفاعها h وعدد عناصرها n تتحقق العلاقتان :

$$\boxed{h \leq n \leq 2^h - 1}$$

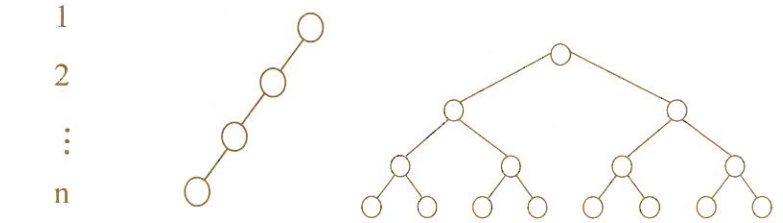
حيث أنه :

في أصغر شجرة ثنائية ممكنة ارتفاعها h (أي شجرة ذات أقل عدد ممكن من العناصر للارتفاع المعطى h) :

$$n = h$$

وفي أكبر شجرة ثنائية ممكنة ارتفاعها h (أي شجرة ذات أكبر عدد ممكن من العناصر للارتفاع المعطى h) :

$$n = 2^h - 1$$



$n = 2^h - 1$
شجرة ثنائية كثيفة (ممتلئة)
full B.T.

$n = h$
شجرة ثنائية خاوية / منحلة / متداعية / هشة degenerate B.T.

تعريف الشجرة الثنائية الكثيفة (الممتلئة) (full B.T.)

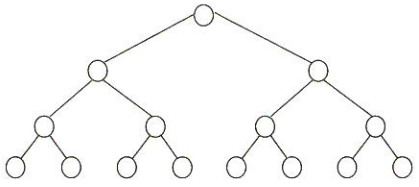
يقال لشجرة ثنائية إنها كثيفة / ممتلئة إذا كانت :

(أ) درجة أي عنصر من عناصرها الداخلية (internal nodes) تساوي ٢.

(ب) جميع أوراقها تقع في آخر مستوى (في المستوى نفسه).

أمثلة للأشجار الثنائية الممتلئة / الكثيفة :

h = 1 h = 2 h = 3



h = 4

وإذا كانت الشجرة كثيفة (full) فإننا نحصل على الحد الأدنى لارتفاع الشجرة h :

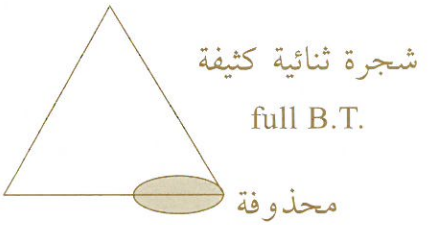
$$h = \lfloor \log_2 n \rfloor + 1 \quad \text{full B.T.}$$

وعموما بالنسبة لشجرة ثنائية عدد عناصرها n فإن ارتفاعها يحقق العلاقة :

$$h \geq \lfloor \log_2 n \rfloor + 1 \quad \text{B.T.}$$

تعريف : الشجرة الثنائية التامة / الكاملة (complete B.T.) :

هي شجرة ثنائية كثيفة / ممتلئة (full B.T.) حُذِفت (deleted) بعض أوراقها من المستوى الأخير (last level) ، أي أنها شجرة ثنائية كثيفة جميع أوراقها تقع في المستويين الأخيرين.



شجرة ثنائية كاملة / تامة

من الواضح انه بالنسبة للشجرة الثنائية الكاملة التي عدد عناصرها n ، فإن ارتفاعها h يحقق العلاقة

$$h = \lfloor \log_2 n \rfloor + 1$$

(تماما مثل الشجرة الثنائية الكثيفة).

(ج) لإيجاد ارتفاع الشجرة h بدلالة عدد عناصرها n :

نفرض أننا أُعطينا أن عدد عناصر شجرة ثنائية ما هو n .

ما هي أقل قيمة ممكنة لارتفاع الشجرة المقابل ؟

$$n \le 2^h - 1$$

$$\Rightarrow \quad 2^h \ge n + 1$$

$$h \ge \log_2 (n+1)$$

وحيث أن h يجب أن تكون عددا صحيحا

$$\Rightarrow \quad h \ge \lceil \log_2 (n+1) \rceil$$

ملاحظة : الاصطلاح $\lceil x \rceil$ يعني : أول عدد صحيح أكبر من أو يساوي x
ويطلق عليه : «سقف x» (ceiling of x / first integer ≥ x / next)
بينما الاصطلاح $\lfloor x \rfloor$ يعني : أول عدد صحيح أصغر من أو يساوي x
ويطلق عليه : «أرضية x» (floor of x / first integer ≤ x / next)

مثلا

$\lceil 3.1 \rceil = 4$ $\quad\quad$ $\lfloor 3.1 \rfloor = 3$

$\lceil 3.9 \rceil = 4$ $\quad\quad$ $\lfloor 3.9 \rfloor = 3$

$\lceil 3 \rceil = 3$ $\quad\quad$ $\lfloor 3 \rfloor = 3$

$$h \geq \lceil \log_2 (n+1) \rceil$$
$$= \lfloor \log_2 n \rfloor + 1$$

$$\Rightarrow \quad \boxed{n \geq h \geq \lfloor \log_2 n \rfloor + 1}$$

أي أن :

$$h \text{ (full B.T.)} = h \text{ (complete B.T.)} = \lfloor \log_2 n \rfloor + 1$$

ولكن

$$n \text{ (full B.T.)} > n \text{ (complete B.T.)}$$

حيث n : عدد العناصر جميعها بالشجرة.

تمثيل الشجرة الثنائية (Representation of B.T.) :

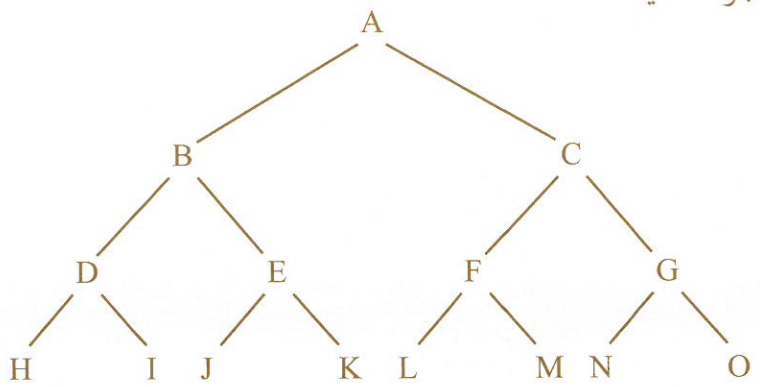
أولا : باستخدام المنظومات (arrays)

ثانيا : باستخدام المؤشرات اليمنى واليسرى (left & right pointers)

أولا : باستخدام المنظومات :

- قُم بتخزين الجذر في الموضع الأول a_0 من المنظومة a
- إذا تم تخزين عنصر (node) في الموضع a_i فقم بتخزين فرعه / ابنه / سليله / ولده الأيسر (left child) في الموضع a_{2i+1} وولده الأيمن (right child) في الموضع a_{2i+2} ، وإذا كان أي منهما غير موجود فقم بتخزين قيمة متلاشية خاصة (special null value).

مثال الشجرة التالية :



يتم تخزينها هكذا

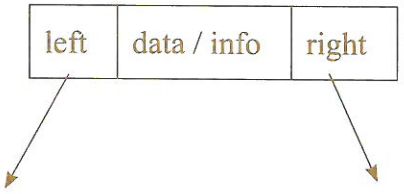
a	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

ملاحظتان :

١- نلاحظ أن هذه الطريقة تقوم بتخزين عناصر الشجرة مستوى مستوى (level by level) من اليسار إلى اليمن (left to right).

٢- يكون هناك هدر كبير في التخزين (waste storage) إذا كان كثير من الأبناء (اليمينيين أو الشماليين) غير موجودين (empty left / right children).

ثانيا : باستخدام المؤشرات اليمنى واليسرى (left & right pointers) :




```
typedef node  *node_ptr;          // global ...
struct  node                      // global ...
 {
        int data;                 // stored data assumed integer
        node_ptr      left, right;
 } ;
```

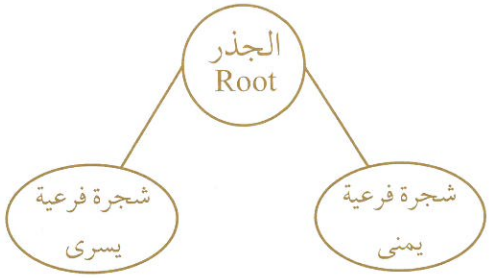
العمليات على الأشجار (Operations with trees) :

فيما يلي أمثلة لبعض العمليات التي نجريها عادة على الأشجار :

- الاجتياز (traversal) وأشهر أنواعه :
  - الاجتياز الترتيبي inorder
  - الاجتياز سابق الترتيب preorder
  - الاجتياز لاحق الترتيب postorder
- البحث عن عنصر searching
- البناء building
- الإدخال / الإضافة inserting
- الحذف deleting

اجتياز شجرة ثنائية Traversal of a B. T.

١) الاجتياز الترتيبي (inorder traversal)



- زيارة (visit) / اجتياز الشجرة الفرعية اليسرى (اجتيازا ترتيبيا)
- زيارة الجذر
- زيارة الشجرة الفرعية اليمنى (اجتيازا ترتيبيا)

٢) الاجتياز سابق الترتيب (preorder traversal)

- الجذر
- اليسرى (باستخدام الاجتياز سابق الترتيب)
- اليمنى (باستخدام الاجتياز سابق الترتيب)

٣) الاجتياز لاحق الترتيب (postorder traversal)

- اليسرى (باستخدام الاجتياز لاحق الترتيب)
- اليمنى (باستخدام الاجتياز لاحق الترتيب)
- الجذر

مثال ٥-١ :

اكتب دالة خاوية للاجتياز الترتيبي لشجرة ثنائية وطباعة عناصرها :

الحل :

```
void inorder (node_ptr  root)
    {
        if (root != NULL)
            {
                inorder (root → left);
```

```
                cout << root → data;
                inorder (root → right);
            }
    }
```

نلاحظ أن هذا الاجتياز اجتياز ارتدادي.

مثال ٥-٢ :

اكتب دالة خاوية للاجتياز سابق الترتيب.

الحل :

```
void preorder (node_ptr  root)
    {
        if (root != NULL)
            {
                cout << root → data;
                preorder (root → left);
                preorder (root → right);
            }
    {
```

نلاحظ أن هذا أيضا اجتياز ارتدادي.

الاجتياز غير الارتدادي لشجرة ثنائية

Non recursive traversal of a B. T.

يمكننا إجراء الاجتياز دون ارتداد باستخدام رصة stack كما يتضح من المثال التالي.

مثال ٥-٣ :

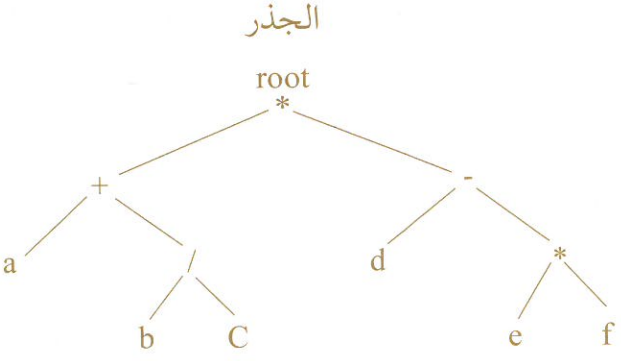
اكتب دالة خاوية (غير ارتدادية) لاجتياز سابق الترتيب لشجرة ثنائية وطباعة عناصرها.

الحل :

```
void preorder1 (node_ptr  root) ;
{
        stack s ;
        create_empty_stack (s) ;
        push (s, root) ;
        while (! empty_s (s))
            {
                pop (s, root) ;
                if (root != NULL)
                    {
                        cout << root → data ;
                        push (s, root → right) ;
                        push (s, root → left) ;
                    }
            }
}
```

مثال ٥-٤ :

استخدم كلا من الأنواع الثلاثة للاجتياز وذلك لاجتياز شجرة التعبير (expression tree) التالية :



ملاحظة : هذه الشجرة تمثل التعبير الحسابي :

(a + b/c) * (d – e*f)

الحل :

أ) الاجتياز الترتيبي (Inorder)

a + b / c * d – e*f

{هو نفسه التعبير (التمثيل) الرمزي الوسطي/ (infix notation expression)}

ب) الاجتياز سابق الترتيب (Preorder)

* + a / b c – d * e f

{هو نفسه التعبير الرمزي السابق (prefix notation)}

ج) الاجتياز لاحق الترتيب (Postorder)

a b c / + d e f * – *

{هو نفسه التعبير الرمزي اللاحق (postfix notation) / (الاصطلاح البولندي) (polish notation)}

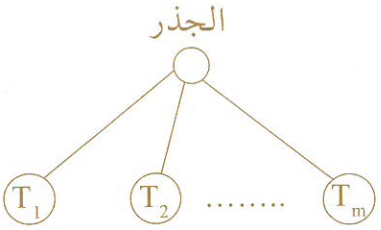
ويمكن حساب قيمة مثل هذا التعبير باستخدام رصة كما سبق ذكره (في فصل الرصات والطوابير).

اجتياز الأشجار العامة (ليس ضروريا الثنائية)

Traversal of trees

يمكننا اجتياز الأشجار العامة عن طريق تعميم الطرق المذكورة سابقا عند اجتياز الأشجار الثنائية كما يلي :

أ) **الاجتياز سابق الترتيب (Preorder)**



- زيارة الجذر
- زيارة الأشجار الفرعية للجذر (subtrees of the root) واحدة واحدة $T_1, T_2, \ldots, T_m$ (باستخدام الاجتياز سابق الترتيب).

ب) **الاجتياز لاحق الترتيب (Postorder)**

- زيارة الأشجار الفرعية للجذر واحدة واحدة $T_1, T_2, \ldots, T_m$ (باستخدام الاجتياز لاحق الترتيب).
- زيارة الجذر

ج) **الاجتياز الترتيبي (Inorder)**

ليس هناك تعريف موحَّد (أي متفق عليه بالإجماع) للاجتياز الترتيبي ، ونذكر هنا التعريف التالي :

- زيارة T_1 (الشجرة الفرعية الأولى ، باستخدام هذا الاجتياز الترتيبي المعدل)
- زيارة الجذر
- زيارة باقي الأشجار الفرعية $T_2, T_3, \ldots, T_m$ (باستخدام هذا الاجتياز الترتيبي المعدل).

تمثيل الأشجار العامة والغابات بأشجار ثنائية
Binary Tree Representation of a Tree / Forest

أولا : تحويل شجرة عامة إلى شجرة ثنائية (B.T.)

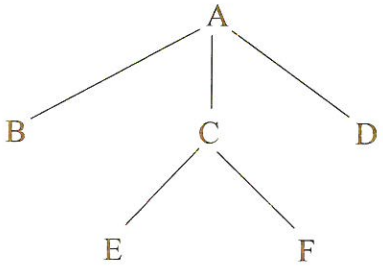
أي تمثيل شجرة عامة بشجرة ثنائية. للقيام بذلك ننفذ خطوتين :

(1) صل جميع الإخوة (brothers) معا [رباط (link) بين كل أخين متتاليين].

(2) احذف جميع الأربطة بين أي عنصر وجميع أبنائه (children) باستثناء الابن الموجود أقصى اليسار (أي أول واحد) (leftmost child/1st child)

مثال ٥-٥ :

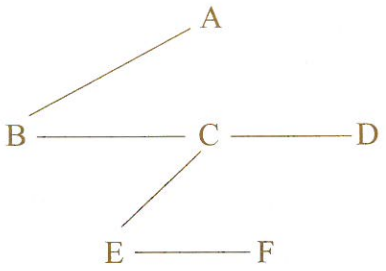
وضح كيفية تمثيل الشجرة العامة التالية بشجرة ثنائية.



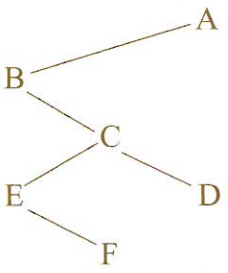
T_1 (شجرة عامة – ليست ثنائية)

الحل :

باتباع الخطوتين المذكورتين لتحويل الشجرة العامة إلى شجرة ثنائية نحصل على الشكل التالي :



ثم نرسم هذه الشجرة الناتجة في الصورة المعتادة :



T_2 (شجرة ثنائية)

نلاحظ أن هذه الطريقة لتمثيل الشجرة العامة بشجرة ثنائية ، أو تحويل (transformation) شجرة عامة إلى شجرة ثنائية تحافظ على نتيجة الاجتياز سابق الترتيب.

(transformation of a tree → B.T. preserves the preorder listing)

مثال ٥-٦ :

طبق أنواع الاجتياز الثلاثة على كل من الشجرة العامة T_1 والشجرة الثنائية المكافئة T_2 ، وقارن بين النتائج.

الحل :

بتطبيق الاجتياز سابق الترتيب على الشجرة T_1 (السابقة) نحصل على النتيجة التالية :

A B C E F D

وبتطبيق الاجتياز سابق الترتيب على الشجرة المكافئة الناتجة T_2 نحصل على النتيجة التالية المطابقة للنتيجة السابقة.

A B C E F D

بينما إذا طبقنا الاجتياز لاحق الترتيب على الشجرة T_1 نحصل على النتيجة التالية :

B E F C D A

وبتطبيق الاجتياز لاحق الترتيب على الشجرة T_2 نحصل على النتيجة المختلفة :

F E D C B A

وبتطبيق الاجتياز الترتيبي على الشجرة T_1 نحصل على النتيجة :

B A E C F D

وبتطبيق الاجتياز الترتيبي على الشجرة T_2 نحصل على النتيجة المختلفة :

B E F C D A

ملاحظة :

نلاحظ أن

نتيجة الاجتياز لاحق الترتيب (T_1) = نتيجة الاجتياز الترتيبي (T_2)

Inorder (T_2) listing = Postorder (T_1) listing

وهذه الملاحظة ليست خاصة بهذا المثال بل هي ملاحظة عامة تنطبق على أي شجرة عامة T_1 والشجرة الثنائية T_2 المكافئة لها.

ثانيا : تحويل غابة إلى شجرة ثنائية (Forest → B.T.)

لتمثيل غابة من الأشجار بشجرة ثنائية مكافئة نتبع الخطوات التالية :

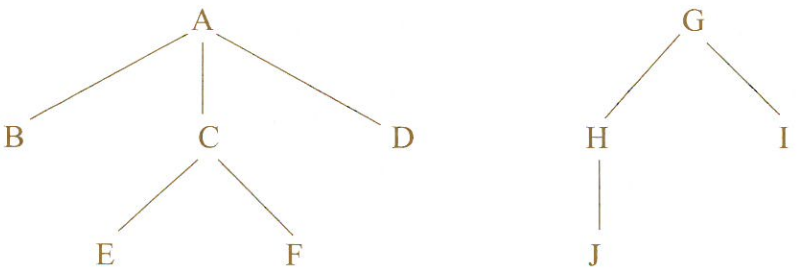
i) صل جذور أشجار الغابة معا.

ii) صل جميع الإخوة في كل شجرة معا.

iii) اقطع جميع الروابط بين أي عنصر وجميع أبنائه عدا أقصى واحد على اليسار.

مثال ٥-٧ :

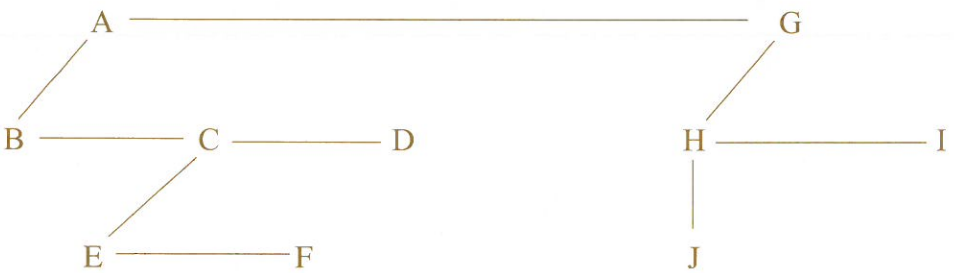
حَوِّل الغابة F التالية إلى شجرة ثنائية .



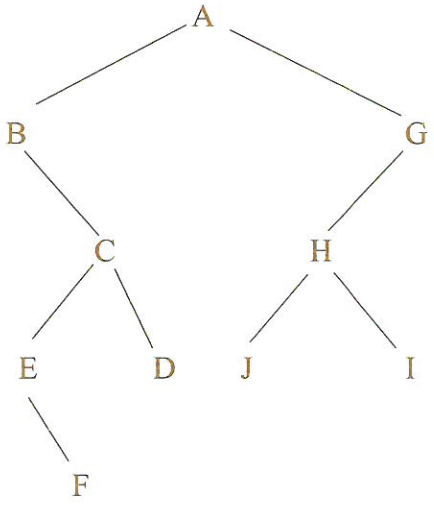
غـابة F

الحل :

باتباع خطوات التحويل المذكورة سابقا نحصل على الشجرة التالية :



ونعيد رسم هذه الشجرة في الصورة المعتادة التالية :



شجرة ثنائية T

تعريف : الاجتياز سابق الترتيب لغابة

(Preorder traversal of a forest)

يتم هذا الاجتياز بالترتيب التالي :

- زيارة T_1 (preorder) (اجتيازا سابق الترتيب)
- ثم زيارة T_2 (اجتيازا سابق الترتيب)

⋮

- ثم زيارة T_m (اجتيازا سابق الترتيب)

تعريف : الاجتياز لاحق الترتيب لغابة

(Postorder Traversal of a forest)

- زيارة T_1 (اجتيازا لاحق الترتيب)
- ثم زيارة T_2 (اجتيازا لاحق الترتيب)

⋮

- ثم زيارة T_m (اجتيازا لاحق الترتيب)

مثال ٥-٨ :

حقق العلاقتين التاليتين بالنسبة للغابة F والشجرة الثنائية المكافئة T :

(i) نتيجة الاجتياز سابق الترتيب للغابة F هي نفسها للشجرة الثنائية المكافئة T.

(ii) نتيجة الاجتياز لاحق الترتيب للغابة F هي نفسها نتيجة الاجتياز الترتيبي للشجرة الثنائية المكافئةT .

الحل :

(i) اجتياز سابق الترتيب (F) :

Preorder (F) : A B C E F D G H J I

(ii) اجتياز سابق الترتيب (T) :

Preorder (T) : A B C E F D G H J I

(iii) اجتياز لاحق الترتيب (F) :

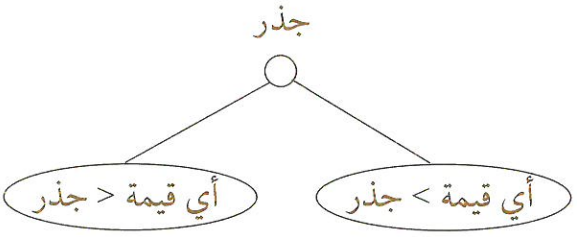
Preorder (F) : B E F C D A J H I G

(vi) اجتياز ترتيبي (T) :

Inorder (T) : B E F C D A J H I G

أشجار البحث الثنائية
Binary Search Trees (BST)

تعريف :



شجرة البحث الثنائية BST : هي شجرة ثنائيةBT يتحقق فيها الشرطان :

أ) أي عنصرkey فيها في الشجرة الفرعية اليسرى أصغر من الجذر ، وأي عنصر فيها في الشجرة الفرعية اليمنى أكبر من الجذر

key (left) < key (root) < key (right)

وكذلك

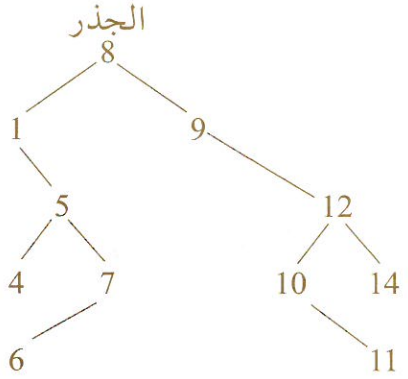
ب) كل من الشجرة الفرعية اليمنى والشجرة الفرعية اليسرى شجرة بحث ثنائيةBST .

مثال ٥-٩ :

كوِّن شجرة بحث ثنائية BST بإدخال (inserting) متتالية العناصر التالية (input sequence of keys) :

8 , 9 , 1 , 12 , 10 , 5 , 7 , 4 , 14 , 11 , 6

الحل :



بعض العمليات على أشجار البحث الثنائية :

١) البحث عن عنصر في شجرة بحث ثنائية (searching a BST) .

٢) إدخال / إضافة عنصر في شجرة بحث ثنائية (inserting an element into a BST) .

٣) حذف عنصر من شجرة بحث ثنائية.

وفيما يلي نقوم بتنفيذ هذه العمليات الثلاث.

١) البحث عن عنصر في شجرة بحث ثنائية (searching a BST)

مثال ٥-١٠ :

اكتب دالة للبحث عن عنصر معين key في شجرة بحث ثنائية ، حيث تقوم الدالة بإرجاع مؤشر للعنصر الذي يحتوي على key.

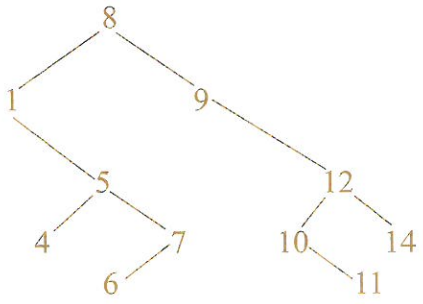
الحل :

```
node_ptr  search_bst (node_ptr  root , int key)
  {
    // to search a bst for the key. We return a pointer to the
```

```
// node that contains the key.
  if (root = = NULL)                    // not found
          return (NULL) ;
    else if (key = = root → data)       // found in the root
          return (root) ;
    else if (key < root → data)         // search in left subtree
          return (search_bst (root → left, key)) ;
    else                                // search in right subtree
          return (search_bst (root → right, key)) ;
}
```

درجة تعقيد دالة البحث (بدلالة عدد عناصر الشجرة n)

للبحث مثلا في الشجرة التالية



عن العناصر 13 , 11 , 14 , 5 , 8 نجد أن عدد العمليات يعطي بالجدول التالي :

العنصر x	عدد العمليات (المقارنات) (بين keys)
8	1
5	3
14	4
11	5
13	5

أي أننا نجد أن عدد العمليات المطلوبة للبحث عن العنصر x يساوي رقم مستوى x

$$\#\ \text{operations (comparisons)} = \#\ \text{level (x)}$$

ونجد أنه في أسوأ حالة فإن عدد المقارنات = ارتفاع الشجرة (مثل البحث عن 6 , 11 , 13)

$$\text{worst\_case complexity} = \text{height (BST)}$$
$$= h$$

وحيث أن h بدلالة n (عدد عناصر الشجرة) تعطى بالعلاقة

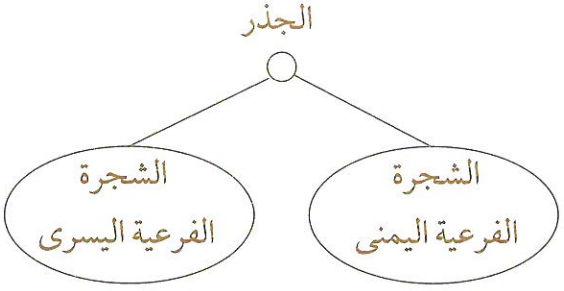
$$\lceil \log (n+1) \rceil \le h \le n$$

فإن درجة التعقيد في أسوأ حالة تقع في هذا المدى.

ولإيجاد درجة التعقيد المتوسطة (average complexity) نحتاج لإيجاد القيمة المتوسطة للارتفاع h ، ويمكن إثبات أن القيمة المتوسطة لارتفاع شجرة البحث الثنائية يتناسب تقريبا مع log n ، أي أن : $[h = O(\log_2 n)]$ ، أي أن درجة التعقيد المتوسطة = $c \log_2 n$ ، وهذا يبين الكفاءة العالية التي تتمتع بها أشجار البحث الثنائية.

البحث التكراري (غير الارتدادي) في شجرة البحث الثنائية

Non recursive search in a BST



نلاحظ أنه عند البحث عن عنصر معين key في شجرة بحث ثنائية ، وبعد مقارنة العنصر key بالقيمة root → data فإننا – إن لم يتساويا – نتجه إما إلى اليمين أو إلى اليسار ولا نحتاج للعودة بعد ذلك إلى الشجرة الفرعية الأخرى (التي لم نتجه إليها) ، وهذا يجعل الحل التكراري (iterative) لمسألة البحث هذه سهلا ، كما يتضح من المثال التالي.

مثال ٥-١١ :

أعد حل المثال السابق (مثال ٥-١٠) باستخدام دالة تكرارية بدلا من الدالة الارتدادية.

الحل :

```
node_ptr  search_bst (node_ptr  root, int key)
  {
    // to search a bst for the key. We return a pointer to the
    // node that contains the key.
      node_ptr  p ;
      p = root ;
      while (p != NULL)
            {
                if (key = = p →  data)
```

```
            return (p) ;
        else if (key < p → data)
            p = p → left ;          // search left subtree
        else
            p = p → right ;         // search right subtree
        }
    return (NULL) ;                 // key is not found
}
```


٢) إدخال / إضافة عنصر في شجرة بحث ثنائية (Insertion into a BST)

مثال ٥-١٢ :

اكتب دالة خاوية ارتدادية لإضافة عنصر key في شجرة بحث ثنائية .

الحل :

```
void insert (node_ptr&  root, int key)
  {
    // to insert the key into a BST. If the key exists, we
    // will not insert it.
      if (root = = NULL) // the new element will be the root}
            root = make_new_node (key) ;
      else if (key = = root → data)
            cout << " will not add , since it exists " << endl;
      else if (key < root → data)
            insert (root → left, key) ;
      else
            insert (root → right, key
  }
```

ملاحظة : في هذه الدالة **insert** استدعينا الدالة make_new_node ،

وفيما يلي بيان لهذه الدالة (في حالة الشجرة الثنائية) :

```
node_ptr  make_new_node (int x)     // as a value returning fn.
        {
                node_ptr  t ;
                t = new node ;
                t → data = x ;
                t → left = NULL ;
                t → right = NULL ;
                return (t) ;
        }
```

وكان يمكننا أيضا استدعاء دالة خاوية - نكتبها فيما يلي - بدلا من دالة تعيد قيمة.

```
void make_new_node (node_ptr&  t, int x)                    // as a void fn.
        {
                t = new node ;
                t → data = x ;
                t → left = NULL ;
                t → right = NULL ;
        }
```

درجة تعقيد الإجراء insert

نلاحظ أن درجة تعقيد هذا الإجراء هي نفسها درجة تعقيد دالة البحث **search_bst** المكتوبة في مثال ٥-١٠ أي أن درجة تعقيد الإجراء **insert** تساوي O(h).

إجراء تكراري (غير ارتدادي) لإدخال (insert) عنصر في شجرة بحث ثنائية

هذا الإجراء شبيه جدا بالإجراء التكراري للبحث في شجرة بحث ثنائية والمذكور في مثال ٥-١١ .

مثال ٥-١٣ :

أعد حل مثال ٥-١٢ ولكن بكتابة دالة خاوية تكرارية (بدلا من الدالة الخاوية الارتدادية) لإضافة عنصر key في شجرة بحث ثنائية .

الحل :

```
void insert (node_ptr&  root, int key)
  {
      // to insert the key into a BST. If the key exists,
      // we will not insert it.

      bool found ;
      node_ptr  p, q, t ;
            p = root ;
            q = NULL ;
      found = false ;
      while ((p != NULL) && (! found))
        if (key = = p → data)
            found = true ;
        else if (key < p → data)  // search in left subtree
                {
                    q = p ;
                    p = p → left ;
                }
        else                              // search in right subtree
                {
```

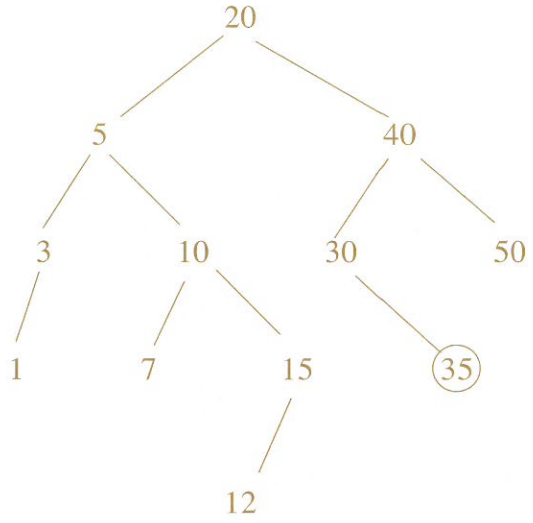
```
                q = p ;
                p = p → right ;
            }
        // end while
    if (! found)
                {
                    t = make_new_node (key) ;
                    if (q = = NULL)
                        root = t ;
                    else if (key < q → data)
                        q → left = t ;
                    else
                        q → right = t ;
                }
    else
        cout << "key already exists in the BST \n";
}
```

٣) إلغاء عنصر من شجرة بحث ثنائية (Deletion form a BST)

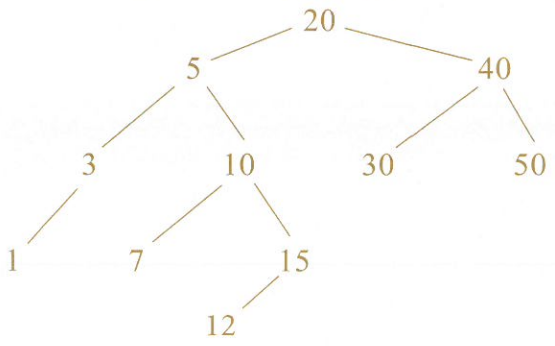
المطلوب في هذه المسألة أن نقوم بإلغاء العنصر الذي يحتوى على قيمة معينة ولتكن x من شجرة بحث ثنائية بحيث تظل الشجرة بعد الإلغاء شجرة بحث ثنائية . يتم الإلغاء بأن نبحث أولا عن العنصر x في الشجرة ، وعملية البحث هذه تنتهي بالوصول إلى حالة من الحالات الأربع التالية :

١- العنصر x غير موجود في الشجرة وبالتالي لا يمكن إلغاؤه.

٢- العنصر x موجود في إحدى الأوراق . والإلغاء هنا بسيط للغاية كما يتضح من إلغاء العنصر 35 من الشجرة التالية :

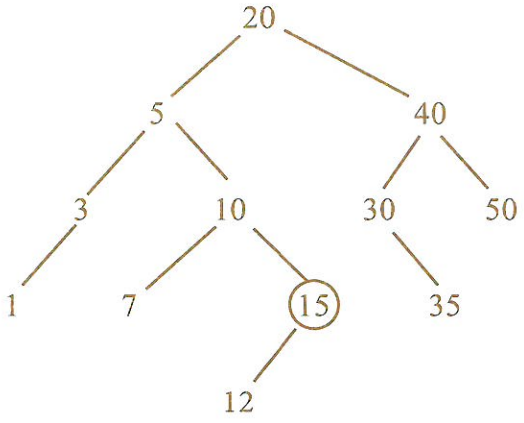


الشجرة قبل إلغاء العنصر 35

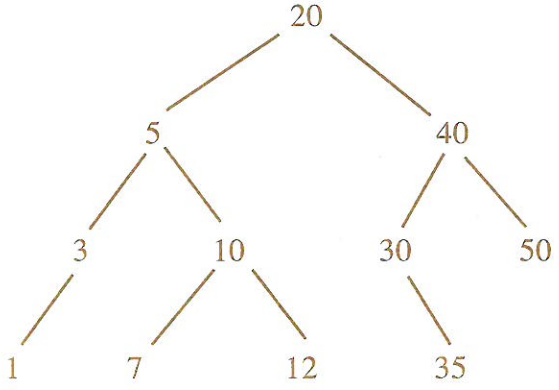


الشجرة بعد إلغاء العنصر 35

٣- العنصر x موجود كعنصر داخلي تتفرع منه شجرة واحدة يسرى أو يمنى. والإلغاء هنا بسيط أيضا وذلك بأن نغير المؤشر الذي يشير للعنصر x بأن نجعله يشير لجذر الشجرة الفرعية الوحيدة المتفرعة من x . ويتضح ذلك من الغاء العنصر 15 من الشجرة التالية :



الشجرة قبل إلغاء العنصر 15

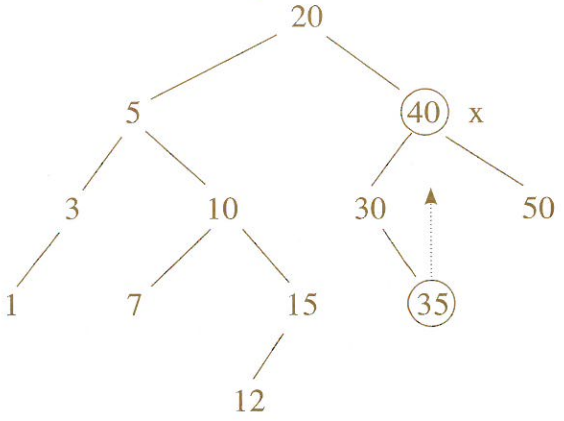


الشجرة بعد إلغاء العنصر 15

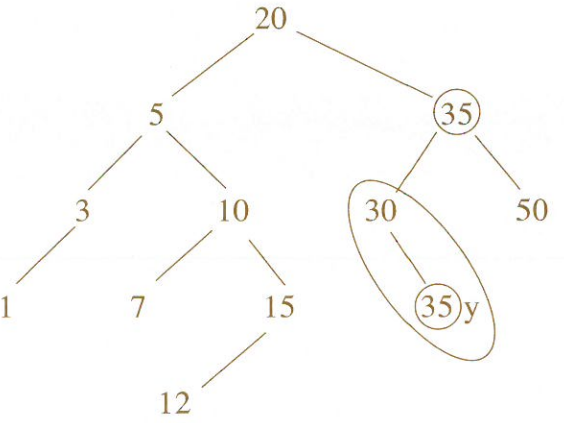
٤- العنصر x المطلوب إلغاؤه هو عنصر داخلي تتفرع منه شجرتان فرعيتان. في هذه الحالة نبحث عن أكبر عنصر وليكن y في الشجرة الفرعية اليسرى للعنصر x (أو أصغر عنصر في الشجرة الفرعية اليمنى) . ثم نقوم باستبدال العنصر y بالعنصر x ، وأخيرا نقوم بإلغاء العنصر y من الشجرة الفرعية اليسرى.

هذه الحالة الأخيرة هي أكثر الحالات الأربع صعوبة ، ولذلك نوضحها بالمثالين التاليين :

المثال الأول : إلغاء العنصر 40 من الشجرة التالية :



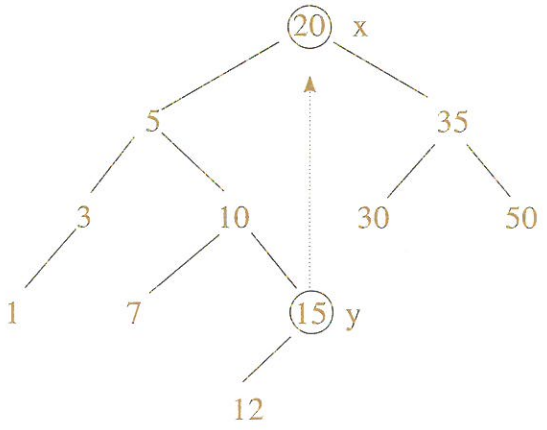
الشجرة قبل إلغاء العنصر 40



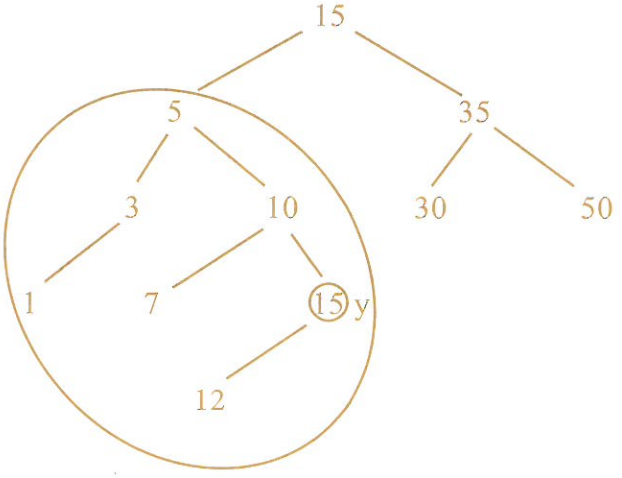
كي يتم الإلغاء نقوم بإلغاء y
من الشجرة الفرعية المحددة.

ونلاحظ هنا أن العنصر y موجود كورقة ومن ثم يسهل إلغاؤه .

المثال الثاني : إلغاء العنصر 20 من الشجرة التالية :



الشجرة قبل إلغاء العنصر 20

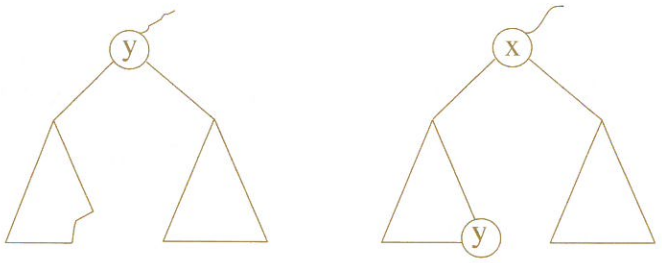


علينا الآن أن نلغي y من الشجرة المحددة

نلاحظ في هذا المثال أن العنصر y موجود كعنصر داخلي تتفرع منه شجرة فرعية واحدة وبالتالي فإن إلغاءه سهل نسبيا .

من هذين المثالين يتضح لنا أن الحالة الرابعة تؤدى دائما إلى إحدى الحالتين الثانية أو الثالثة ولكن مع شجرة أصغر . وذلك يرجع إلى أن أكبر عنصر في شجرة بحث ثنائية غير خالية يقع في أقصى عنصر إلى اليمين في تلك الشجرة وهذا بدوره قد يكون أقصى ورقة إلى اليمين أو أقصى عنصر داخلي إلى اليمين وفى هذه

الحالة لا تتفرع منه إلا شجرة فرعية واحدة إلى اليسار . لا يبقى لدينا إلا أن نرى أن الشجرة الناتجة تظل شجرة بحث ثنائي ، وذلك يتضح بملاحظة أن y أصغر من x ، وبالتالي فإن العنصر y يكون أصغر من عناصر الشجرة الفرعية اليمنى للعنصر x . ومن ناحية أخرى فإن عناصر الشجرة الفرعية اليسرى للعنصر x تكون أصغر من y .



بعد إلغاء x　　قبل إلغاء x

والإجراء التالي يقوم بعملية الإلغاء المطلوبة :

مثال ٥-١٤ :

اكتب دالة خاوية لحذف العنصر الذي يحتوي على قيمة معينة ولتكن x (عدد صحيح) من شجرة بحث ثنائية بحيث تظل الشجرة بعد الحذف شجرة بحث ثنائية أيضا .

الحل :

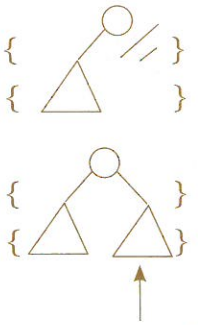
```
void del_bst (node_ptr&  root, int x)
  {
    // to delete the node that contains x in a BST.
        node_ptr  t ;
        int y ;
        if (root = = NULL)                                  // case (1)
                cout << x << " does not exist \n " ;
        else if (x < root → data)
```

```
        del_bst (root → left, x) ;  // delete from left subtree
else if (x > root → data)
        del_bst (root → right, x) ; // delete from right subtree
else // x = = root → data, we arrived at the node to be deleted
        {
            t = root ;
            if ((root → left = = NULL) && (root → right = =
                                                NULL))     // case (2)
                        {
                                root = NULL ;
                                delete t ;
                        }
            else if (root → left = = NULL)                  // case (3)
                                {
                                        root = root → right ;
                                        delete t ;
                                }
            else if (root → right = = NULL)                 // case (3)
                                {
                                        root = root → left ;
                                        delete t ;
                                }
            else                                            // case (4)
             { // let y be the largest element in the left subtree
                        y = largest (root → left) ;
                        root → data = y ;
                        // delete y from the left subtree
                         del_bst (root → left, y) ;
                         }
        }
```

```
    }       // del_bst
```

في هذه الدالة del_bst استخدمنا الدالة largest لإيجاد أكبر عنصر y في شجرة بحث ثنائية غير خالية . وفيما يلى بيان لهذه الدالة :

```
int largest (node_ptr  root)
{ // to find the largest element in a non empty BST.
      if (root → right = = NULL)
        return (root → data) ;
      else
        return (largest (root → right)) ;
}
```



الكبير يقع هنا

درجة تعقيد خوارزمية الحذف

Complexity of the algorithm del_bst

يمكن إثبات أن درجة تعقيد الدالة del_bst تساوي O(h) .

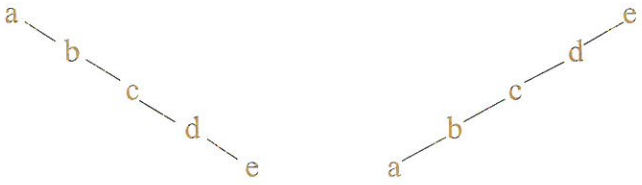
(من الناحية الفعلية complexity ≤ 2h) .

ملاحظة :

لاحظنا من الأمثلة السابقة أن عدد خطوات أي من عمليات البحث والإضافة وكذلك الإلغاء في شجرة بحث ثنائية يعتمد على ارتفاع الشجرة ، وحيث أن القيمة المتوسطة لارتفاع شجرة البحث الثنائية العشوائية التي تحتوي على n من العناصر يتناسب تقريبا مع $\log_2 n$ فمن ذلك نرى أنه في المتوسط تحتاج عملية البحث أو الإضافة أو الإلغاء في شجرة بحث ثنائية إلى $c \log_2 n$ من الخطوات (العمليات) ، وهذا يبين الكفاءة العالية التي تمتاز بها أشجار البحث الثنائية .

الأشجار المتوازنة
Balanced Trees

رأينا أن متوسط عدد المقارنات المطلوبة للبحث عن عنصر (/ لإدخال عنصر / لحذف عنصر) في شجرة بحث ثنائية BST يساوي O(log n) ، وهذا هو الحال بالنسبة لشجرة عشوائية (random BST) ، ولكن في بعض الحالات قد تكون الشجرة متداعية (خاوية / هشة) ويصل عدد العمليات إلى O(n) ، مثل الشجرة الناشئة عن إدخال العناصر التالية : a b c d e أو e d c b a



شجرتان BST متداعيتان (degenerate) حيث الارتفاع = height = n

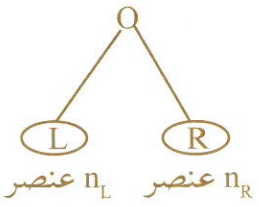
ومن الواضح أن أداء الشجرة المتوازنة [كالشجرة الكثيفة / الممتلئة (full) مثلا ، أو الشجرة الكاملة / التامة (complete B.T.)] أفضل من أداء الشجرة غير المتوازنة.

ملاحظة : في حالة الشجرة المتداعية (كالسابقة) حيث h = n :
البحث عن عنصر يتطلب في المتوسط $\frac{n}{2}$ ،
وفي أسوأ حالة (لاحظ h = n) فإن زمن البحث (search time) يساوي O(n).
بينما في حالة الشجرة BST – كما ذكرنا – فإن زمن البحث يساوي O(log n) (في المتوسط) ، ولذلك نفضل دائما أن تكون الشجرة متوازنة.

نوعا الشجرة الثنائية المتوازنة
(Two kinds of Balanced B.T.) (BST)

(١) الشجرة الثنائية المتوازنة وزنا (weight balanced B.T.) :

يقال إن الشجرة الثنائية متوازنة وزنا إذا كان عدد العناصر (nodes) في الشجرة الفرعية اليمنى يساوي تقريبا عدد العناصر (nodes) في الشجرة الفرعية اليسرى $(n_L \cong n_R)$.



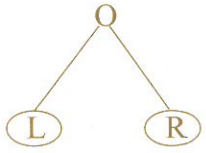
وعادة نضع حدا (limit) على النسبة n_L / n_R لتحديد شرط التوازن.

(٢) الشجرة الثنائية المتوازنة ارتفاعا (height balanced B.T.) :

يقال إن الشجرة الثنائية متوازنة ارتفاعا إذا كان ارتفاع الشجرة الفرعية اليمنى يساوي تقريبا ارتفاع الشجرة الفرعية اليسرى $(h_L \cong h_R)$ ، وعادة نضع الشرط : ألا يزيد الفارق بين الارتفاعين عن ١ (واحد). وتعرف مثل هذه الأشجار أيضا باسم **أشجار AVL** وهي اختصار (Adelson-Velskii, Landis) (1962).

تعريف :

شجرة البحث الثنائية T (BST) تسمى **شجرة AVL أو شجرة ثنائية متوازنة ارتفاعا** (ht. bal. T.) إذا وفقط إذا تحقق أحد الشرطين التاليين :



أ) الشجرة T خالية (empty).

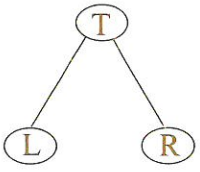
أو ب) إذا كان : (i) $(= 0 \text{ or } 1)$ $|h_L - h_R| \leq 1$

وأيضا (ii) كل من الشجرة الفرعية اليمنى R والشجرة الفرعية اليسرى L شجرة AVL

تعريف :

معامل التوازن لعنصر Balance Factor (B.F.) of a node

معامل التوازن لعنصر L هو قيمة زيادة ارتفاع الشجرة الفرعية اليسرى h_L عن ارتفاع الشجرة الفرعية اليمنى h_R بالنسبة لهذا العنصر ، أي أن :



$$BF(T) = h_L - h_R$$

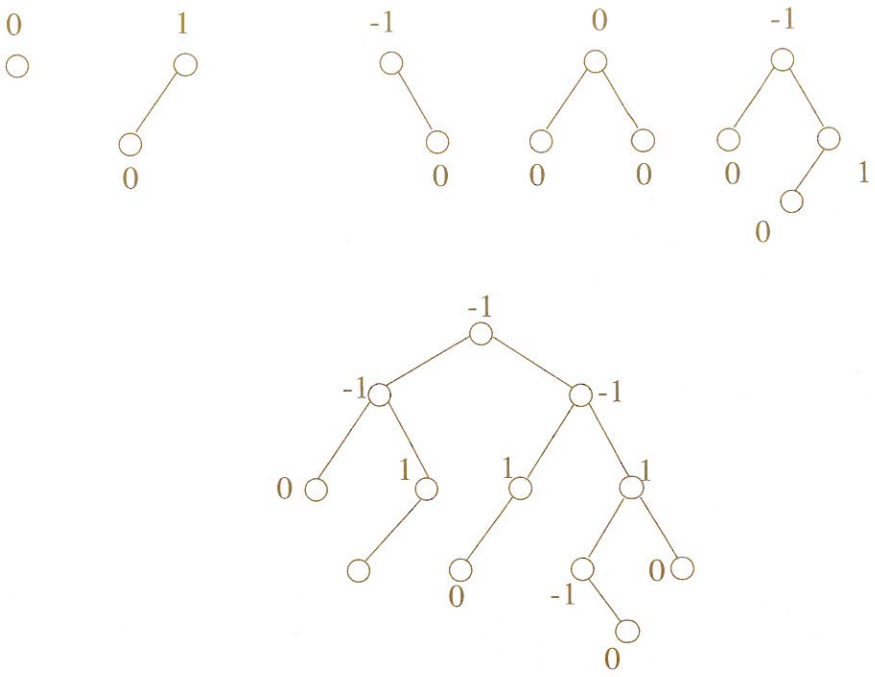
وإذا كان T = NILL فإن BF (T) = 0

{ واضح أنه إذا كان العنصر T ورقة فإن : BF (leaf) = 0}

بناء على هذا التعريف نرى أن معامل التوازن لأي عنصر في شجرة AVL يساوي إحدى القيم الثلاث 1 , 0 , 1-.

أمثلة لأشجار AVL

فيما يلي أمثلة لأشجار AVL مع وضع قيمة معامل التوازن BF لكل عنصر من عناصر الشجرة. ويلاحظ أنه لمعرفة ما إذا كانت شجرة معطاة هي شجرة AVL أم لا ، فإننا نوجد معامل التوازن لكل عنصر من عناصرها ، وإذا وجدنا أن جميع قيم هذا المعامل تساوي 0 أو 1 أو 1- فإن الشجرة تكون شجرة AVL ، وإلا فهي ليست شجرة AVL.



تعد أشجار AVL من الأشجار ذوات الكفاءة العالية من الناحية العملية ، وذلك لأن ارتفاعها O(log n).

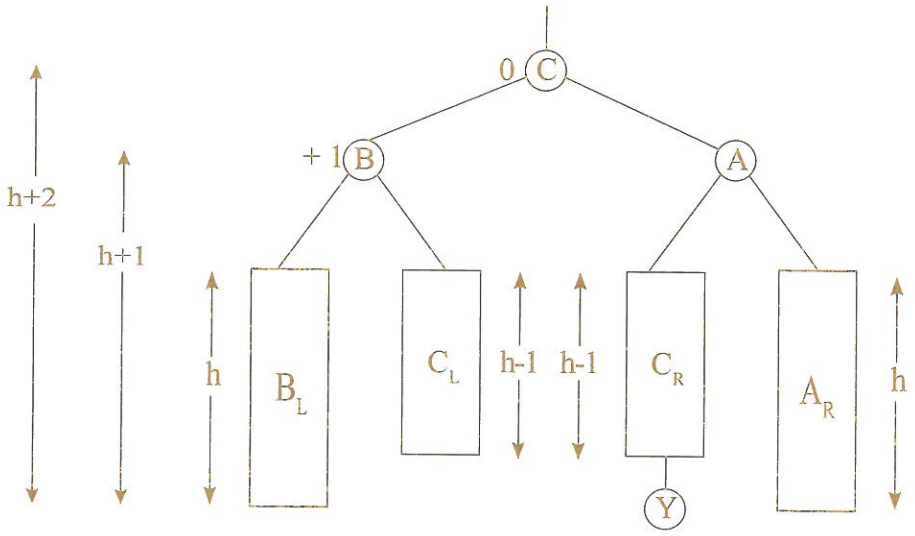
نظرية (دون برهان) : ارتفاع (height) الشجرة AVL (التي عدد عناصرها n) يحقق العلاقة

$$h \cong 1.45 \log_2 n \; ; \qquad n >> 1$$

وهذا يعني أنه في أسوأ حالة فإننا نضمن أن أيا من عمليات البحث أو الإضافة أو الحذف تكون O(log n) .

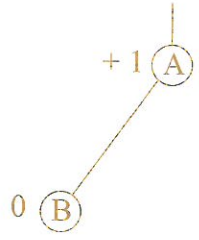
والآن نتناول مسألة الإضافة أو الحذف insert / delete من شجرة AVL بحيث تبقى بعد الإضافة أو الحذف شجرة AVL.

وسيشتمل العنصر في الشجرة AVL على المجالات المبينة بالشكل التالي :



الشجرة بعد إعادة التوازن (LR)

ملاحظة ٢ : إذا كان العنصر B في الشجرة الأصلية ورقة ، أي ليس له شجرة فرعية يمنى أو يسرى ، أي أن الشجرة معطاة بالشكل البسيط التالي :



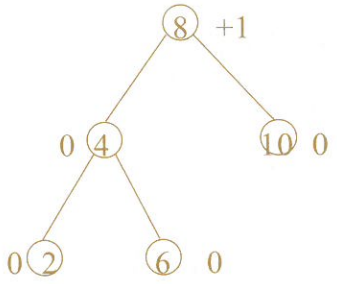
الشجرة قبل الإضافة (متوازنة)

ثم أضفنا العنصر الجديد Y يمين B ، أي جعلنا Y ابنا أيمن للعنصر B ، فإن الشجرة تصبح بالشكل التالي :

بينما إذا أضفنا عنصرا مثل 1 أو 3 فإننا نجد أن توازنها يختل وتحتاج إلى إعادة توازن (rebalancing) لتظل شجرة AVL (وبالطبع شجرة BST) ، كما يتضح من المثال التالي.

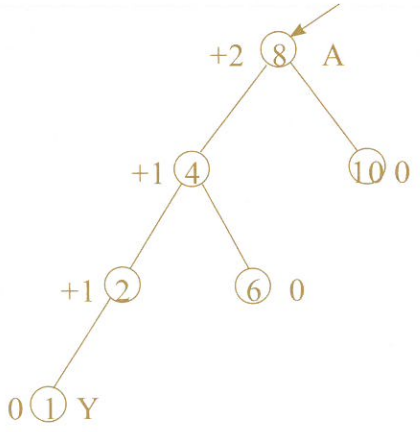
مثال ٥-١٥ :

أدخل العنصر 1 في الشجرة AVL المبينة بالشكل التالي بحيث تظل الشجرة أيضا شجرة AVL بعد الإدخال .



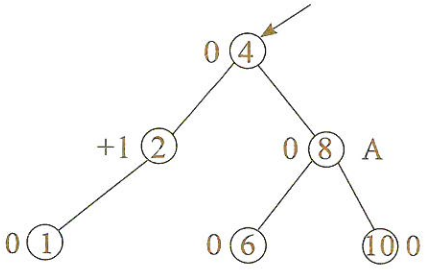
الحل :

بعد إدخال العنصر 1 في الشجرة (كشجرة بحث ثنائية BST) تصبح الشجرة ومعاملات توازن عناصرها كما يلي :



نلاحظ أن الشجرة أصبحت غير متوازنة حيث أصبح معامل توازن العنصر 8 يساوي 2+. ولإعادة توازن الشجرة بحيث تظل أيضا شجرة BST نجعل

الابن الأيسر للعنصر 8 ≡ A الذي اختل توازنه (أي نجعل العنصر 4) جذرا بينما نجعل العنصر 8 (الذي اختل توازنه) ابنا أيمن لهذا الجذر (4) ، ونجعل الابن الأيمن (الأصلي) للعنصر 4 (أي العنصر 6) ابنا أيسر للعنصر8 ، فتصبح الشجرة بالشكل التالي :



نلاحظ أن الشجرة أصبحت متوازنة كما يتضح من قيم معاملات التوازن المبينة بالشكل.

ومثل هذا التعديل الذي قمنا به لإعادة توازن الشجرة يطلق عليه تدوير LL (Left Left Rotation) ، حيث أن العنصر المضاف (Y≡1) الذي أدت إضافته إلى اختلال توازن الشجرة يقع في الشجرة الفرعية اليسرى الواقعة بدورها في الشجرة الفرعية اليسرى لأول عنصر "A" أصبح معامل توازنه

$$(BF(A) = BF(8)=) \pm 2$$

[أي لأقرب عنصر A للعنصر المضاف Y] .

* * *

ويمكننا أن نلخص فيما يلي الحالات المختلفة لاختلال توازن أشجار AVL التي قد تقابلنا ، وقواعد إعادة توازن هذه الأشجار ، ثم نعطي بعد ذلك بعض الأمثلة .

إعادة توازن / تدوير أشجار AVL المختلة التوازن بسبب الإضافة

Rebalancing / Rotating unbalanced AVL trees due to insertion

إذا اختل توازن شجرة AVL بسبب إضافة عنصر Y ونريد إعادة توازنها / تدويرها فيجب أن نحقق ما يلي :

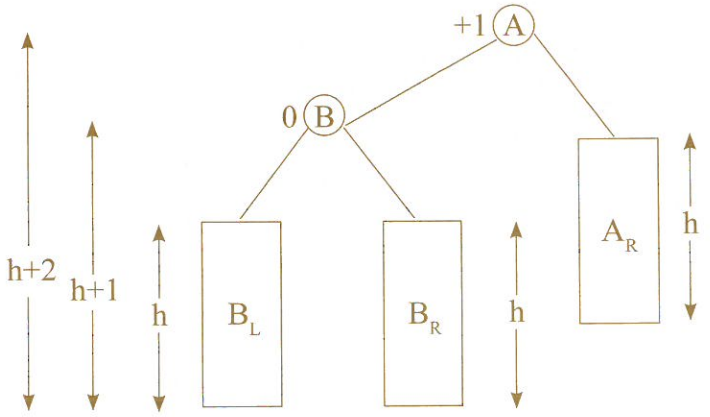
(i) تصبح الشجرة شجرة AVL متوازنة .

(ii) تظل الشجرة شجرة BST (شجرة بحث ثنائية) [أي أن نتيجة الاجتياز الترتيبي (inorder traversal) تظل كما هي قبل وبعد التدوير] .

هناك أربع حالات مختلفة يمكن أن تقابلنا بالنسبة للأشجار AVL المختلة التوازن بسبب الإضافة ، حالتان منها مشابهتان / مماثلتان (symmetric / similar) للحالتين الأخريتين .

(أ) تدوير LL (LL Rotation)

نفرض أن لدينا الشجرة AVL المتوازنة التالية (والتي قد تكون شجرة فرعية) ، حيث A_R شجرة فرعية يمنى للعنصر A ، و B_R شجرة فرعية يمنى للعنصر B ، و B_L شجرة فرعية يسرى للعنصر B ، ونفرض أن ارتفاع كل من هذه الأشجار الفرعية الثلاث يساوي h .

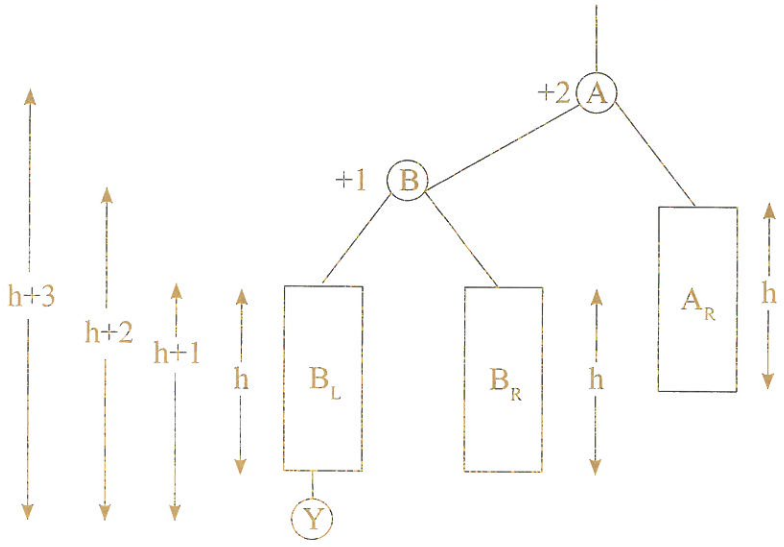


الشجرة قبل الإضافة (متوازنة)

والآن نضيف عنصرا Y في الشجرة الفرعية اليسرى (B_L) للعنصر B الواقع بدوره في الشجرة الفرعية اليسرى للعنصر A.

(Left subtree of the Left subtree of A)

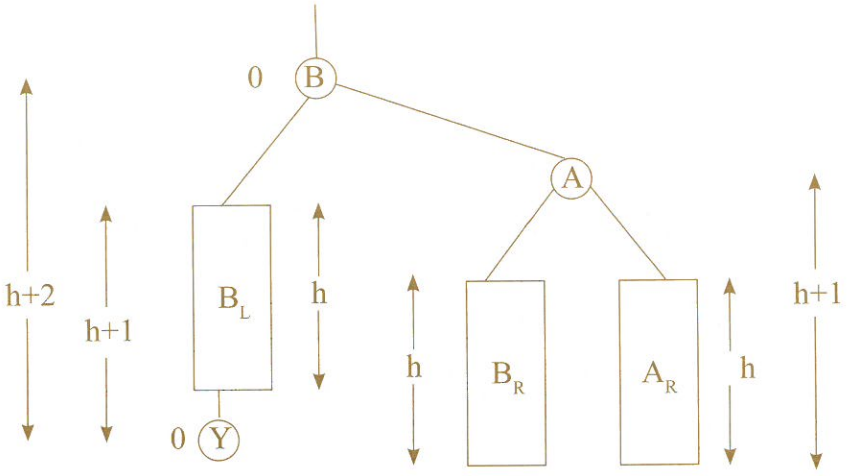
ولهذا يطلق الاصطلاح LL على هذه الحالة).



الشجرة بعد الإضافة (غير متوازنة)

نلاحظ أن الشجرة أصبحت غير متوازنة حيث معامل توازن العنصر A أصبح 2+ . ولإعادة توازن الشجرة نقوم بإجراء الخطوات التالية والتي يطلق عليها تدوير LL :

- العنصر B يصبح جذر هذه الشجرة (الفرعية) ، (أي يصعد لأعلى)
- العنصر A يصبح الابن الأيمن للجذر B (أي ينزل لأسفل)
- الابن الأيمن للعنصر B يصبح الابن الأيسر للعنصر A



الشجرة بعد إعادة التوزان

ونلاحظ من الشكل ومن قيم معامل التوازن أن الشجرة أصبحت متوازنة أي أنها الآن شجرة AVL . {لاحظ أن الارتفاع عند العنصر B هو نفسه الارتفاع السابق}. كما نلاحظ أيضا أن الشجرة مازالت شجرة BST حيث أن الاجتياز الترتيبي للشجرة يعطي النتيجة السابقة نفسها وهي : B_L + العنصر الجديد Y

B

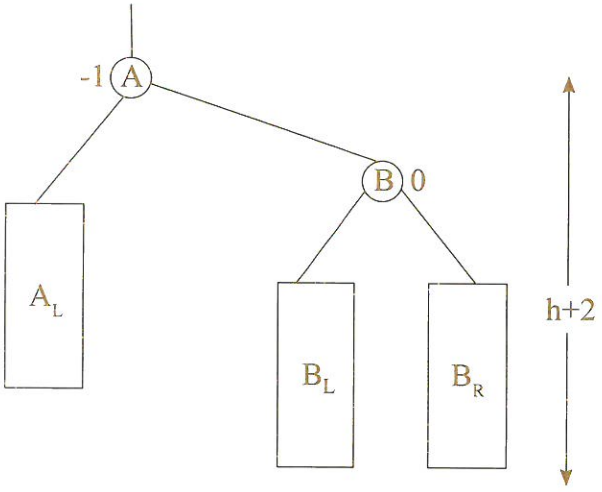
B_R

A

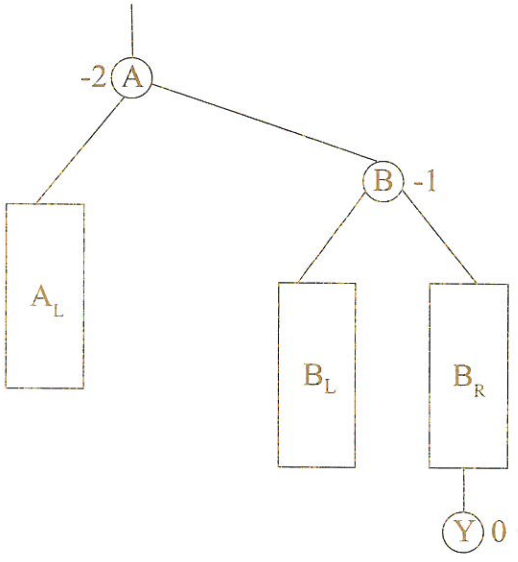
A_R

(ب) تدوير RR (RR Rotation)

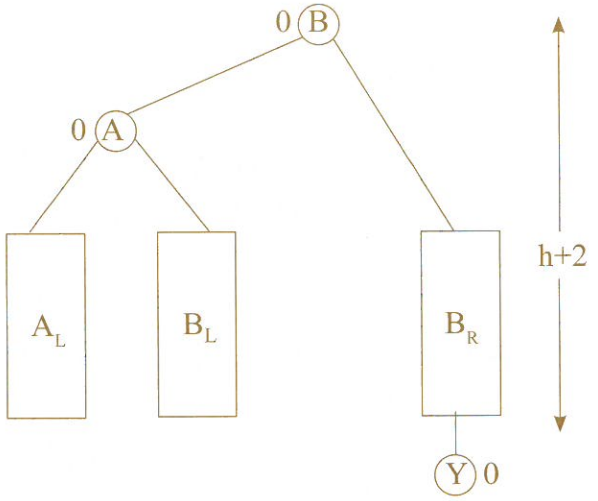
هذه الحالة مماثلة للحالة السابقة (تدوير LL) ولذلك نكتفي هنا برسم الشجرة قبل إضافة العنصر وبعد الإضافة وبعد إعادة التوازن.



الشجرة قبل الإضافة (متوازنة)



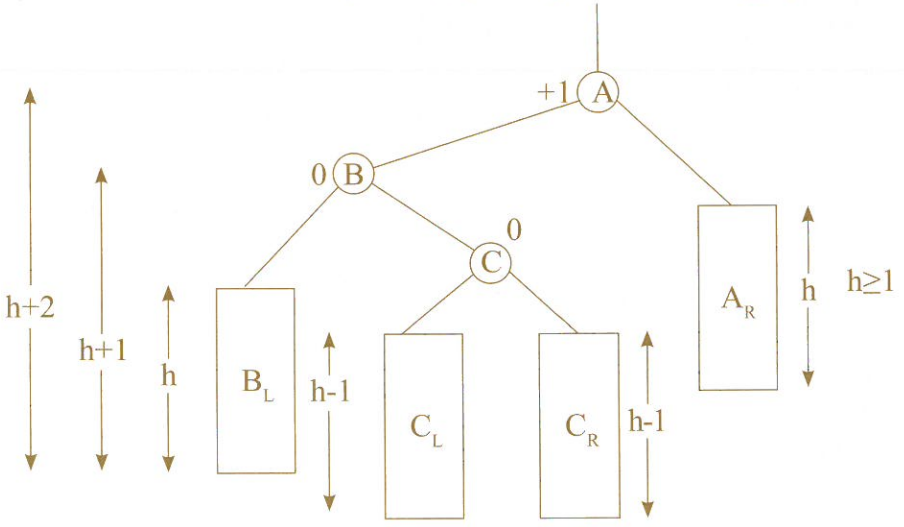
الشجرة بعد الإضافة (غير متوازنة)



الشجرة بعد إعادة التوازن (RR)

(ج) تدوير مزدوج LR (LR Double Rotation)

نفرض أن لدينا الشجرة (الفرعية) التالية :



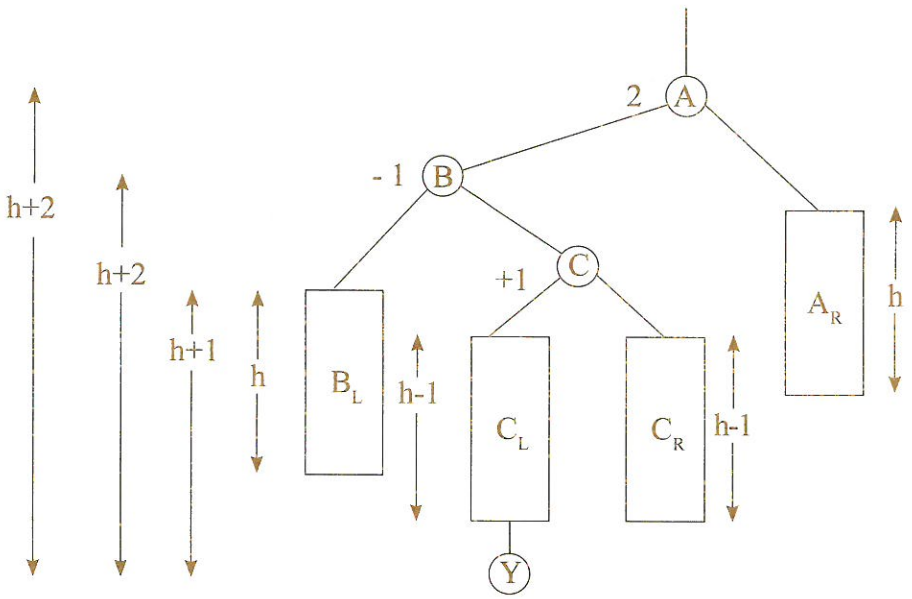
الشجرة قبل الإضافة (متوازنة)

والآن أدخل عنصرا جديدا Y في الشجرة الفرعية التي جذرها C ، أي في الشجرة الفرعية اليمنى للعنصر B الذي هو جذر الشجرة الفرعية اليسرى للعنصر A ، ونقول باختصار إننا ندخل عنصرا في الشجرة الفرعية اليمنى للشجرة الفرعية اليسرى للعنصر A (Right subtree of the left subtree of A) ، ومن هنا يطلق الاصطلاح LR على هذه الحالة .

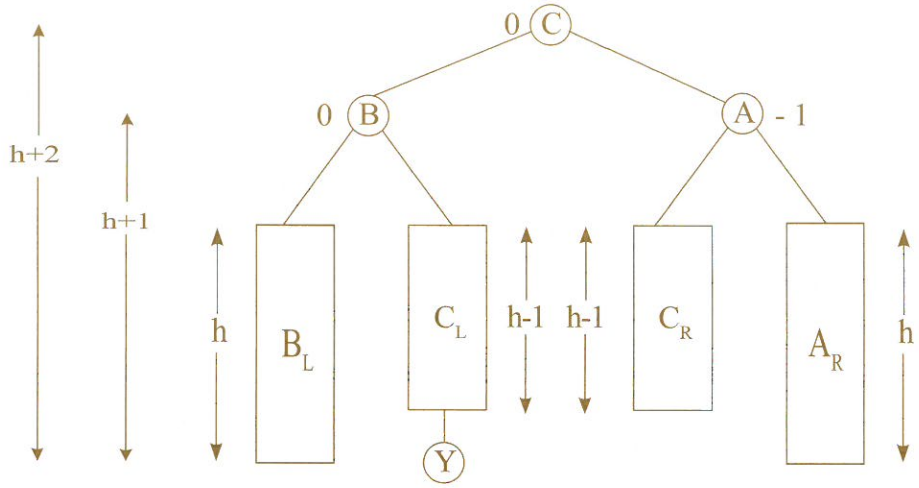
ملاحظة ١ :

يجوز إدخال العنصر في الفرع C_R أو في الفرع C_L :

(i) مثلا إذا أدخلنا العنصر في الفرع C_L تصبح الشجرة ومعاملات توازن عناصرها كما يلي :



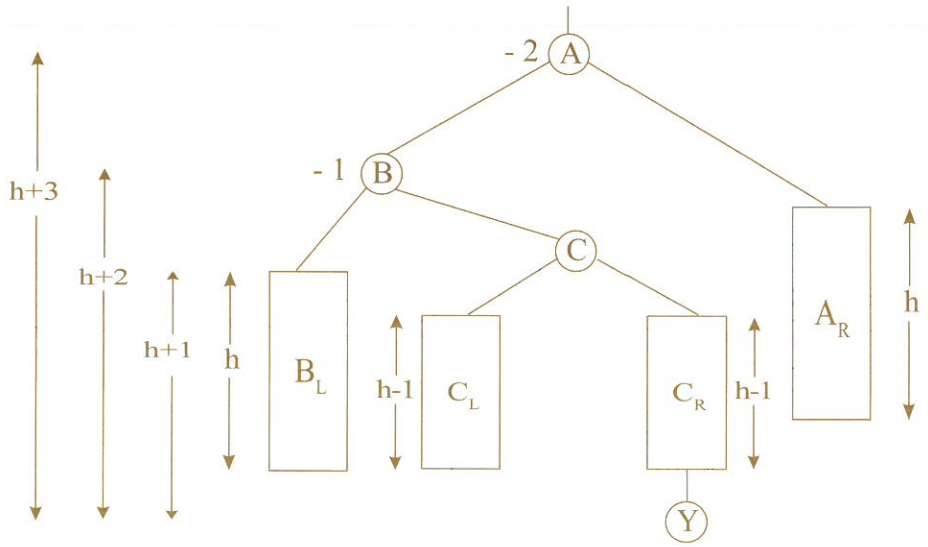
الشجرة قبل الإضافة (غير متوازنة)



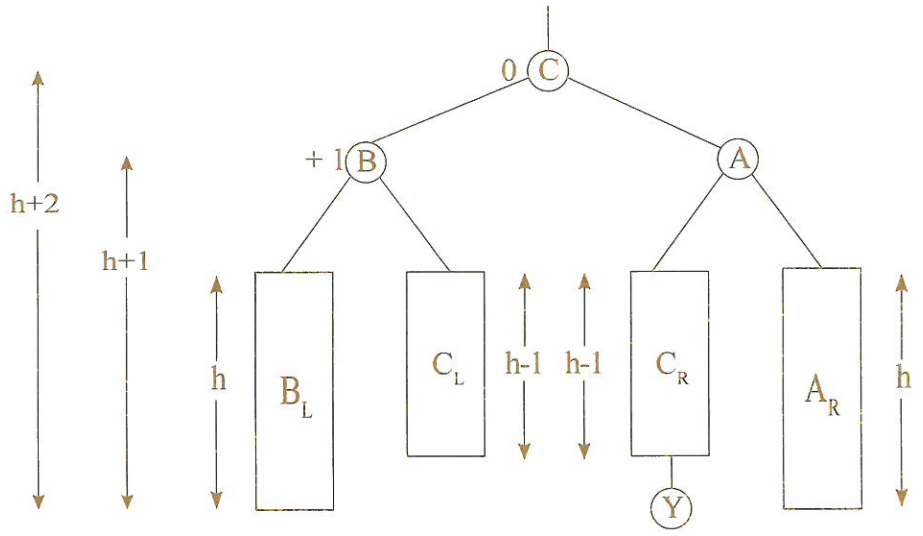
الشجرة بعد إعادة التوازن (LR)

{لاحظ هنا الحركة الرأسية (vertical movement) للعنصر C حيث يصعد رأسيا لأعلى ، ولا توجد له حرة عرضية (lateral movement) }

(ii) بينما إذا أدخلنا العنصر Y في الفرع C_R فإن الشجرة ومعاملات توازن عناصرها تصبح كما يلي :

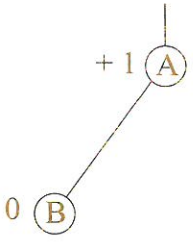


الشجرة بعد الإضافة (غير متوازنة)



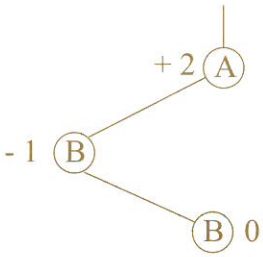
الشجرة بعد إعادة التوازن (LR)

ملاحظة ٢ : إذا كان العنصر B في الشجرة الأصلية ورقة ، أي ليس له شجرة فرعية يمنى أو يسرى ، أي أن الشجرة معطاة بالشكل البسيط التالي :



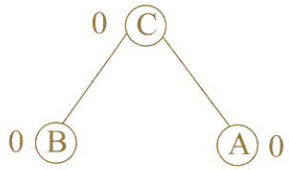
الشجرة قبل الإضافة (متوازنة)

ثم أضفنا العنصر الجديد Y يمين B ، أي جعلنا Y ابنا أيمن للعنصر B ، فإن الشجرة تصبح بالشكل التالي :



الشجرة بعد الإضافة (غير متوازنة)

ونعيد توازنها كما يلي :



الشجرة بعد إعادة التوازن (LR)

(د) تدوير مزدوج RL (RL Double Rotation)

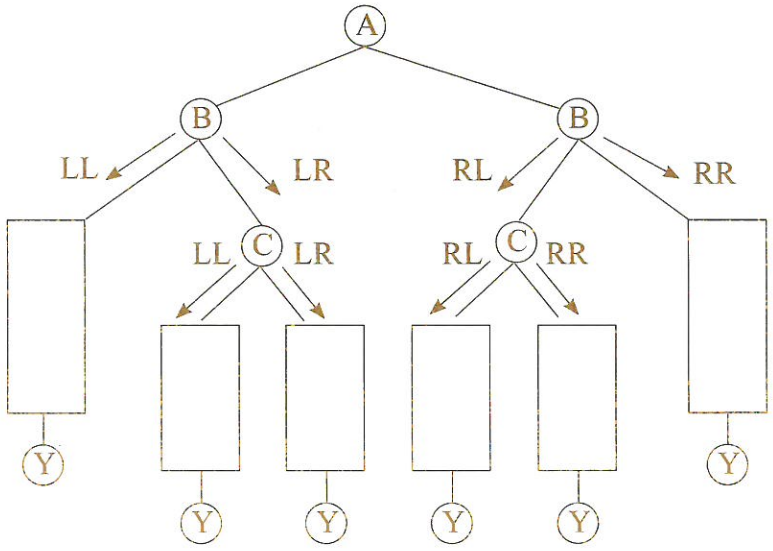
هذه الحالة مشابهة للحالة السابقة فلا داعي لذكر تفاصيلها.

ملاحظات على التدوير وإعادة التوازن :

* التغيرات التي قد تطرأ على قيم معامل التوازن نتيجة إضافة عناصر للشجرة تعد تغيرات محلية (local changes) ، وهي تحدث أساسا في جزء من مسار البحث / الإضافة / الإدخال (search / insertion path) الذي تتبعناه لإضافة العنصر الجديد ، أما باقي عناصر الشجرة التي لا تقع على هذا المسار فلا تتغير قيم معاملات توازنها.
* لحساب القيم المعدلة لمعاملات توازن العناصر الواقعة على مسار الإضافة – والتي قد تتغير قيم معاملات توازنها – بعد إضافة العنصر الجديد Y ، نتتبع المسار راجعين ابتداءً من العنصر الجديد Y ونحسب القيم الجديدة لمعامل

التوازن لهذه العناصر على مسار العودة عنصرا عنصرا إلى أن نصل إلى عنصر A معامل توازنه إما 2+ أو 2- أو إلى ان نصل إلى الجذر root.

* إذا وصلنا إلى عنصر A معامل توازنه ±2 فإن الشجرة تحتاج إلى إعادة توازن ، فنحدد أولا نوع حالة اختلال التوازن أي نوع التدوير المطلوب (LL , RR , LR , RL)
* نقوم بإعادة توازن الشجرة - إن كانت تحتاج لذلك - بتدويرها حول المركز A بناء على القواعد المذكورة سابقا {(أ) - (د)} حسب حالة اختلال توازنها.

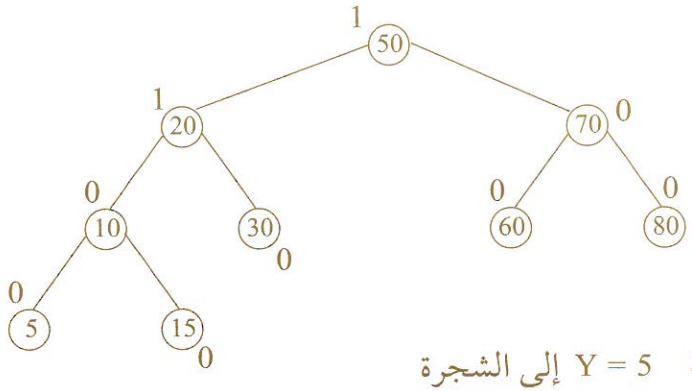


والأمثلة التالية توضح بإذن الله الخطوات السابقة.

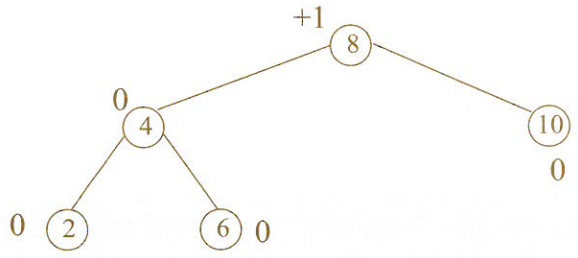
مثال ٥-١٦ :

في كل من الأشجار AVL التالية أدخل العنصر المقابل Y ، بحيث تظل الشجرة أيضا شجرةAVL (أي شجرة متوازنة) بعد الإدخال.

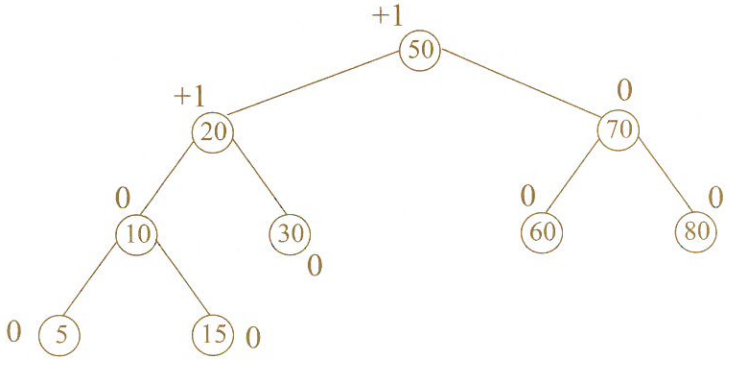
(أ) إضافة Y = 8 إلى الشجرة



(ب) إضافة Y = 5 إلى الشجرة

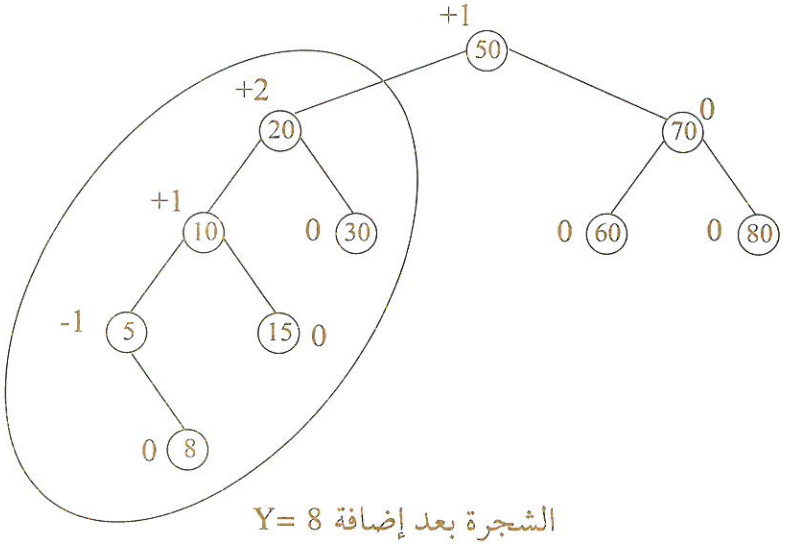


(5) إضافة Y = 18 إلى الشجرة



الحل :

(أ)

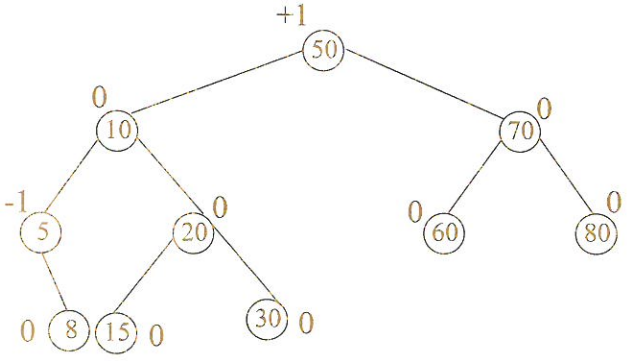


الشجرة بعد إضافة Y= 8

(LL)

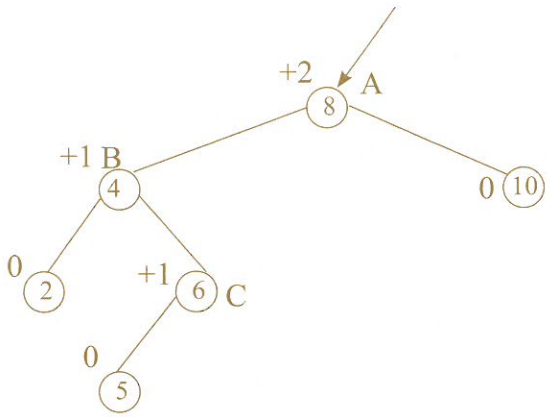
لاحظ أن الشجرة الفرعية داخل المنحنى المغلق تحتوي على عناصر الشجرة والقيم الجديدة لمعاملات توازنها عند تتبع مسار الإضافة من العنصر المضاف (Y = 8) إلى أول عنصر (A = 20) معامل توازنه 2 ±.

ونلاحظ أن الشجرة تحتاج إلى تدوير L L.



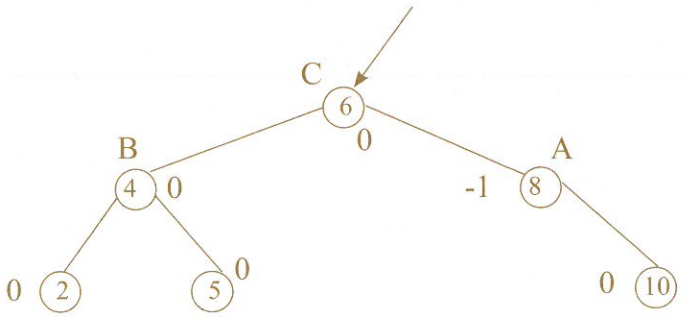
الشجرة بعد إعادة التوازن

(ب)



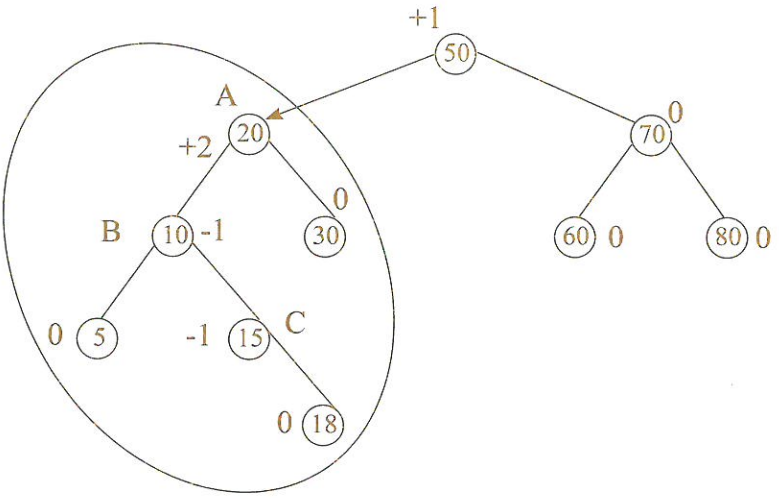
الشجرة بعد إضافة Y= 5

(LR)



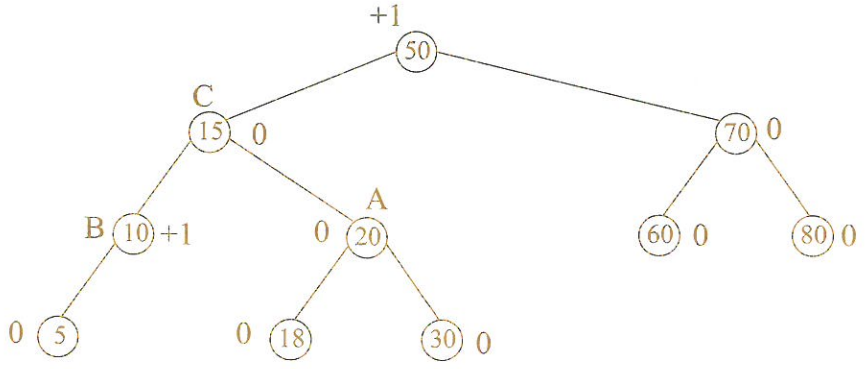
الشجرة بعد اعادة التوازن

(ج)



الشجرة بعد إضافة Y= 18

(LR)



الشجرة بعد إعادة التوازن

نتائج عامة حول أشجار AVL

أولا : من المعلوم نظريا أن ارتفاع شجرة AVL التي عدد عناصرها n يحقق العلاقة / النظرية :

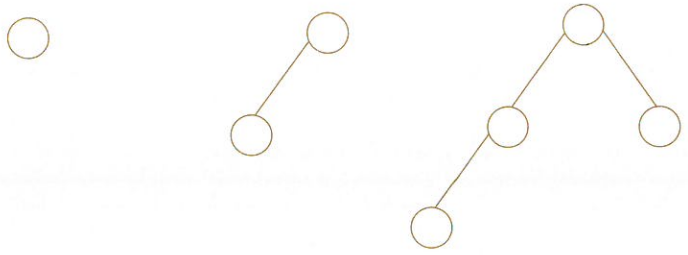
$$h \leq 1.44 \log n + O(1)$$

وهذا يعني - كما سبقت الإشارة إلى ذلك - أن عمليات البحث / الإضافة / الحذف يمكن أن تتم في أسوأ حالة في حدود O(log n)

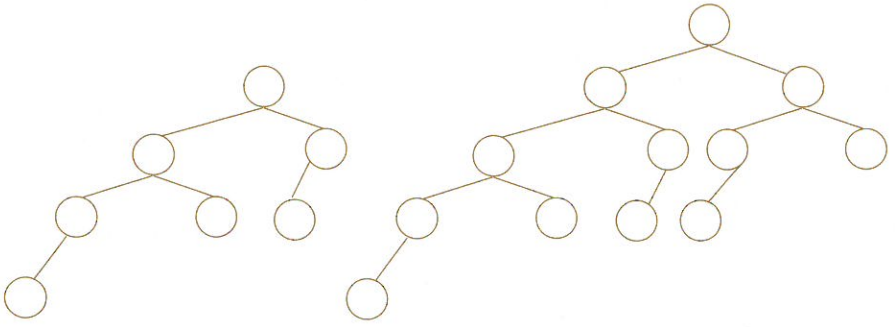
[قارن القيمة O(n) بالنسبة للشجرة BST المتداعية(degenerate) مع القيمة المتوسطة O(log n) بالنسبة للشجرة BST العشوائية].

والعلاقة / النظرية السابقة تتطلب منا أن نحاول إيجاد أسوأ أشجار AVL التي تحتوي أي منها على n عنصر ، وهذه عادة تسمى أشجار «فيبوناتشي» (Fibonacci Trees).

والأشكال التالية تعطي هذه الأشجار.



h = 1 h = 2 h = 3



h = 4 h = 5

أشجار فيبوناتشي لقيم h المختلفة

ثانيا : من النتائج العملية (experimental results) المعلومة بالنسبة لأشجار AVL

(أ) الارتفاع المتوقع (expected height) / القيمة المتوسطة لارتفاع شجرة AVL التي تحتوي على n عنصر يساوي $\log n + O(1)$

(ب) نحتاج لإجراء عملية إعادة اتزان شجرة AVL (عن طريق التدوير) مرة كل عمليتي إضافة للشجرة ومرة كل خمس عمليات حذف من الشجرة.

(ج) من الناحية العملية تعد أشجار BST أفضل من أشجار AVL ما عدا الحالات التي نحتاج فيها لإجراء عمليات بحث في الشجرة أكثر بكثير من عمليات الإضافة أو الحذف.

أشجار B المتوازنة متعددة الطرق
Multiway Balanced B trees

تفيد أشجار B في تخزين (في عمليات البحث / الإضافة / الحذف الخاصة بـ) كميات كبيرة من البيانات على وسائل التخزين المساند / الثانوي ، ويتميز هذا التخزين الخارجي بسعة التخزين الكبيرة ، وبطء الوصول إلى السجلات (أبطأ بكثير من الذاكرة الرئيسية). والهدف هو تقليل زمن الوصول.

تعريف الشجرة B من الرتبة d (B tree of order d)

١- أي عنصر (node) يحتوي على الأكثر على عدد 2d من القيم (keys).

٢- أي عنصر – باستثناء الجذر – يحتوي على الأقل على عدد d من القيم.

٣- الجذر يحتوي على عدد من 1 إلى 2d من القيم.

٤- جميع الأوراق (leaves) تظهر في المستوى نفسه.

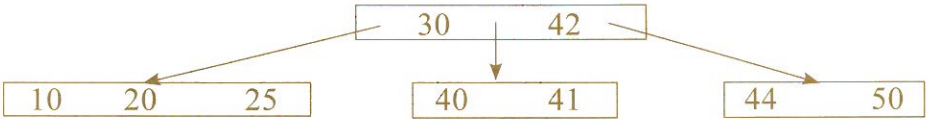
٥- عدد أبناء (children) أي عنصر داخلي (internal node) يساوي
= 1 + عدد قيمه.

[= عدد المؤشرات (pointers) الخارجة من العنصر]

٦- القيم مرتبة في أي عنصر.

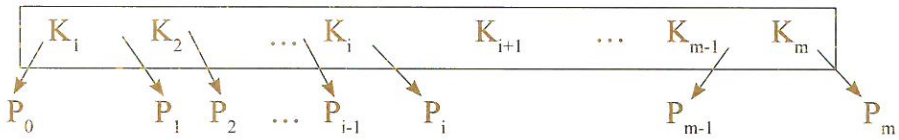
***

في الأمثلة التالية نفترض d = 2 ولكن قيمة d من الناحية العملية قد تصل إلى عدة مئات.



مثال لشجرة B من الرتبة d = 2

الشكل العام لبنية عنصر (structure of a node) في شجرة B



القيم K_i في أي عنصر تكون مرتبة أي أن : $K_1 < K_2 < \ldots < K_m$

عدد المؤشرات في العنصر يساوي $m + 1$ ، والمؤشرات هي : $P_0 , P_1 , \ldots , P_m$

المؤشر P_0 يشير إلى الشجرة الفرعية التي يحقق أي عنصر key فيها العلاقة key $< K_1$

المؤشر P_1 يشير إلى الشجرة الفرعية التي يحقق أي عنصر key فيها العلاقة $K_1 < key < K_2$

⋮

المؤشر P_i يشير إلى الشجرة الفرعية التي يحقق أي عنصر key فيها العلاقة $K_i < key < K_{i+1}$

⋮

المؤشر P_m يشير إلى الشجرة الفرعية التي يحقق أي عنصر key فيها العلاقة K_m < key

البحث عن قيمة x في شجرة B (Searching a B tree for a key x)

- ابحث عن x في جذر الشجرة B (تتابعيا أو ثنائيا sequential / binary)
- إذا وجدته انتهى البحث ، وإلا :

إذا كان $x < K_1$ اتبع المؤشر P_0 وأعد البحث ،

وإذا كان $Ki < x < K_{i+1}$ اتبع المؤشر P_i وأعد البحث ،

وإذا كان $x > K_m$ اتبع المؤشر P_m وأعد البحث.

- وإذا صادفنا مؤشر التلاشي NILL فإن البحث ينتهي بالنتيجة «غير موجود».

مثال ٥-١٧ :

ابحث في الشجرة السابقة (الشجرة B من الرتبة d = 2) عن القيم التالية :

42 , 15 , 20 , 35 , 50 , 60.

الحل :

(موجودة) 20 (غير موجودة) 15 (موجودة) x = 42

(غير موجودة) 60 (موجودة) 50 (غير موجودة) 35

ملاحظة :

إذا كانت القيمة غير موجودة فإننا ننتهي عند ورقة شجرة.

إدخال/ إضافة قيمة x في شجرة B

(Inserting a key/element into a B tree)

تتبع مسار البحث عن x (كما سبق) في الشجرة B إلى أن تصل إلى ورقة شجرة (leaf node) ، أي عنصر هو ورقة من أوراق الشجرة :

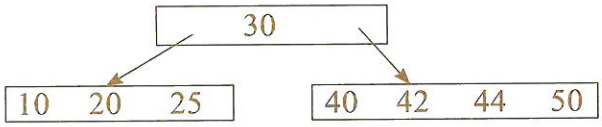
- فإذا كان عدد القيم (keys) بهذا العنصر أقل من 2d فإننا نضيف القيمة x بسهولة في هذا العنصر.
- وإذا كان العنصر ممتلئا (full node) أي يحتوي على 2d قيمة فإننا نجد أن عدد القيم – إذا أضفنا x – سيصبح 2d+1 ولا يسعها العنصر.

* نرتب هذه القيم – (التي عددها 2d+1) – ترتيبا تصاعديا.
* نقسم / نجزئ (split) العنصر إلى عنصرين بكل منهما d قيمة (اليمنى واليسرى).
* وندخل القيمة الوسطى (middle key) في العنصر الوالد (parent node) (إن كان هناك عنصر والد أو ننشئ عنصرا جديدا) إن لم يكن ممتلئا ، فإن كان العنصر الوالد نفسه ممتلئا (أي به 2d عنصر) فإننا نطبق طريقة التقسيم / التجزئة هذه مرة أخرى ، وقد تنتشر عدة مرات.

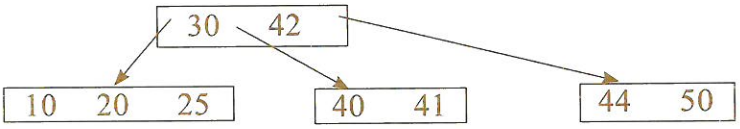
ملاحظة : يسهل برمجة هذه الطريقة ارتداديا.

مثال ٥-١٨ :

أضف 41 إلى الشجرة التالية من الرتبة الثانية (d = 2) :

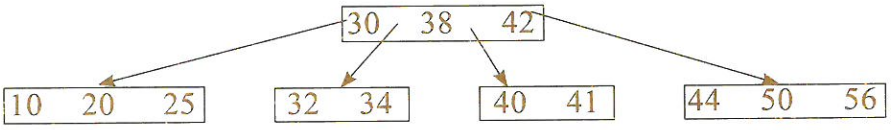


الحل :



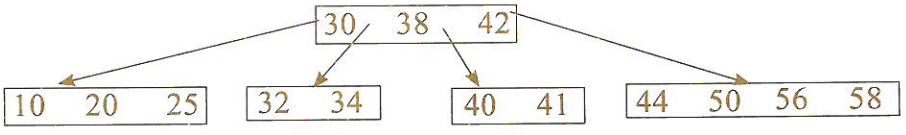
مثال ٥-١٩ :

أضف القيم 46 , 54 , 52 , 60 , 58 بهذا الترتيب المعطى إلى الشجرة التالية من الرتبة الثانية (d = 2).

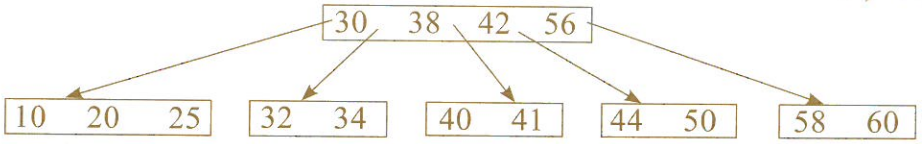


الحل :

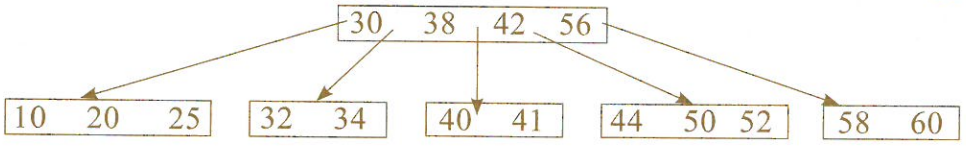
(أ) إضافة 58 (مباشرة) :



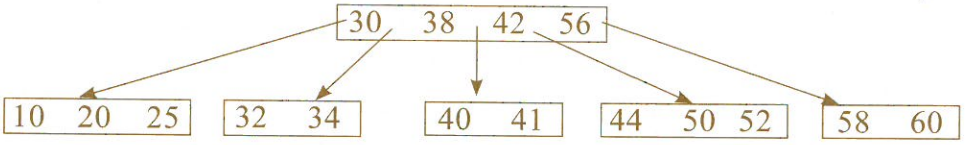
(ب) إضافة 60 :



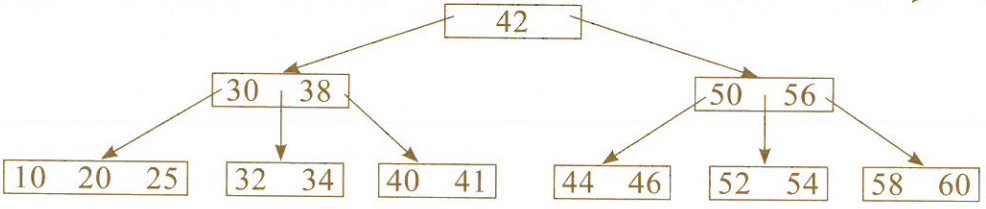
(ج) إضافة 52 :



(د) إضافة 54 :



(هـ) إضافة 46 :



ملاحظتان :

1) الشجرة B تبقى متوازنة محتفظة بجميع خواصها بعد أي عملية إضافة.

2) الشجرة B تنمو (grows) في الاتجاه من الأوراق نحو الجذر عكس اتجاه نمو أشجار BST (والأشجار الطبيعية !).

حذف قيمة x من شجرة B

(Deleting a key/element from a B tree)

نفرض أن الشجرة B من الرتبة d. ولحذف القيمة / العنصر x من الشجرة نبحث أولا عن هذه القيمة ، فإن وجدناها في الشجرة في عنصر z (node) مثلا ، فهناك احتمالان :

(أ) العنصر z الذي يحتوي على القيمة x عبارة عن ورقة (leaf) من أوراق الشجرة.

(ب) العنصر z عبارة عن عنصر داخلي (internal node) وليس ورقة (nonleaf node).

نبحث أولا الحالة الثانية (ب) :

لحذف القيمة x نتبع الخطوات نفسها لحذف عنصر من شجرة بحث ثنائية BST ، أي نوجد أكبر عنصر y في الشجرة الفرعية اليسرى للعنصر x (أو أصغر عنصر y في الشجرة الفرعية اليمنى للعنصر x) ، ثم نستبدل بالعنصر x العنصر y (أي نضع القيمة y موضع القيمة x المطلوب حذفها) ونحذف العنصر الأصلي y (delete). وحيث أن العنصر الأصلي y موجود في ورقة (لأنه أكبر عنصر أو أصغر عنصر في شجرة فرعية) فبذلك نصل في هذه الحالة الثانية (ب) إلى ضرورة حذف عنصر موجود في ورقة وهي الحالة الأولى (أ) نفسها. ولذلك ندرس فيما يلي كيفية حذف قيمة / عنصر من ورقة في شجرة.

حذف قيمة x من ورقة :

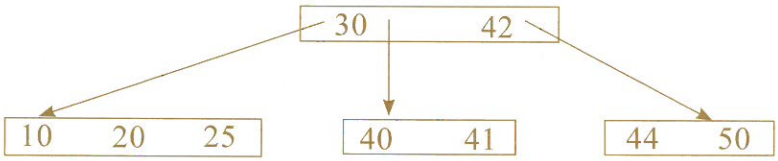
هذه العملية هي أساسا عكس عملية الإضافة / الإدخال. وهناك عدة حالات يمكن أن تنشأ.

(i) القيمة x موجودة في ورقة تحتوي على عدد من القيم $\leq d+1$.

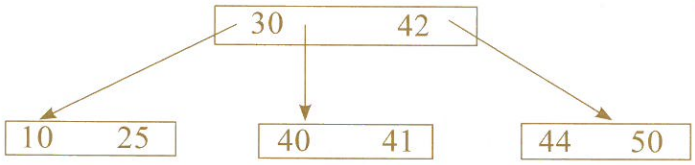
في هذه الحالة نحذف x مباشرة ، وتنتهي عملية الحذف لأن الورقة بعد عملية الحذف تظل عنصرا صحيحا (valid node) لأنها تحتوي على عدد من القيم ≤ d .

مثال ٥-٢٠ :

احذف القيمة 20 من الشجرة B التالية :

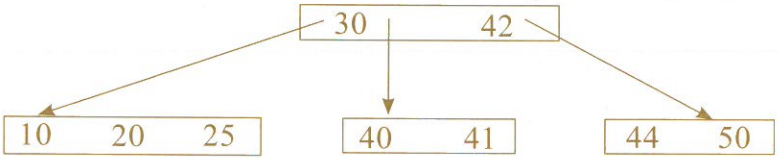


الحل :

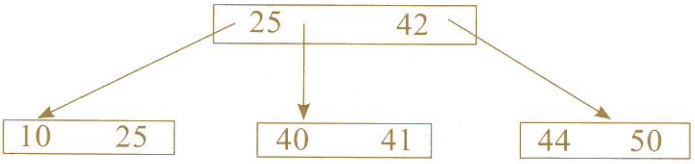


مثال ٥-٢١ :

احذف القيمة 30 من الشجرة B التالية

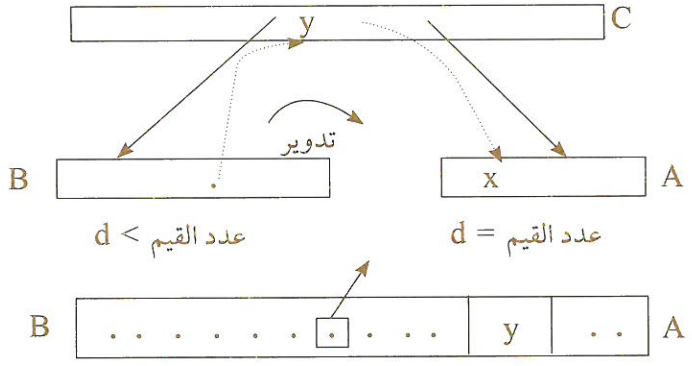


الحل :



بعد أن نستبدل بالقيمة 30 (المطلوب حذفها) القيمة 25 (وهي أكبر قيمة في الشجرة الفرعية اليسرى) يكون حذف القيمة 25 من الورقة التي تحتوي عليها مباشرا وبسيطا وذلك لأن هذه الورقة ستظل تحتوي على قيمتين بعد الحذف.

(ii) القيمة x موجودة في ورقة (A مثلا) تحتوي بالضبط على عدد d من القيم. في هذه الحالة لا نستطيع أن نحذف x مباشرة وذلك لأن الورقة التي تحتوي عليها ستحتوي على عدد d-1 من القيم بعد الحذف ، وهذا غير مسموح به (d-1 < d). ولذلك فإننا في هذه الحالة نبحث عن أخ مباشر (immediate brother) (B مثلا) – أي يمين الورقة مباشرة أو شمالها مباشرة – يحتوي على عدد من القيم > d. فإن وجدنا مثل هذا الأخ فإننا نستعير / نقترض (borrow) منه (قيمة أو أكثر) كما يلي ، بعملية يمكن اعتبارها عملية تدوير (rotation) :

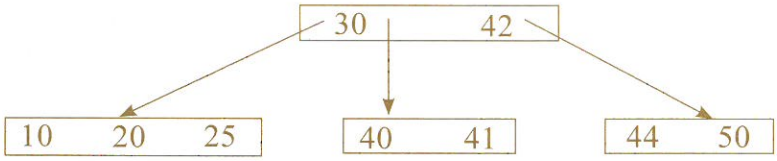


بعد أن نستبعد من الورقة / العنصر A القيمة x (المطلوب حذفها) ، نضع قيم B والقيمة y [من العنصر C الوالد المشترك (joint parent) للعنصرين A , B] وقيم A متتابعة (كما هو مبين بالشكل) ثم نرفع القيمة الوسطى (middle value) إلى أعلى لتحل محل y ، وننزل القيمة y إلى العنصر A ، كما يوضح ذلك المثال التالي.

[ملاحظة : يمكننا استعارة أكثر من قيمة واحدة من B ، وذلك بإعادة توزيع قيم A , B بالتساوي بينهما ، وذلك لتحقيق توازن أفضل].

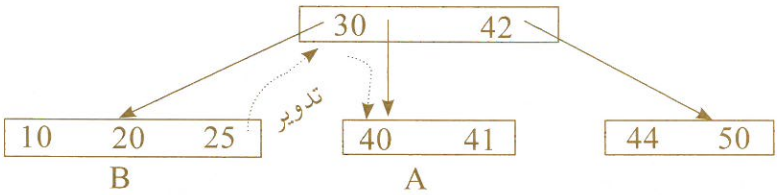
مثال ٥-٢٢ :

احذف القيمة x = 40 من الشجرة التالية :



الحل :

يمكننا الاستعارة من B نظرا لأن عدد القيم في B يساوي ٣ > ٢ (انظر الشكل).

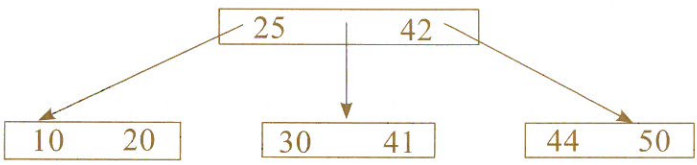


متتابعة القيم :

10 20 25 30 41

القيمة الوسطى

نتيجة التدوير :



(iii) القيمة x موجودة في ورقة تحتوي بالضبط على عدد d من القيم ولا يوجد أخ مباشر يحتوي على عدد من القيم > d (أي أن أي أخ مباشر يحتوي بالضبط على عدد d من القيم).

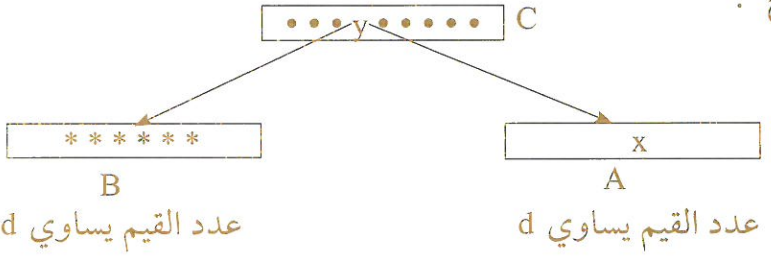
في هذه الحالة ندمج (merge) عنصرين (2 nodes) معا مع قيمة من العنصر الوالد (إذا كان هذا العنصر الوالد يحتوي على عدد من القيم > d).

مثلا إذا أردنا حذف القيمة x من العنصر A الذي يحتوي على عدد d من العناصر ، وكان الأخ المباشر B يحتوي أيضا على عدد d من العناصر ، فإننا بعد حذف x ندمج العنصرين A , B مع قيمة y من العنصر الوالد (C مثلا). وبالتالي يكون عدد القيم في العنصر الجديد الناتج من عملية الدمج هذه هو :

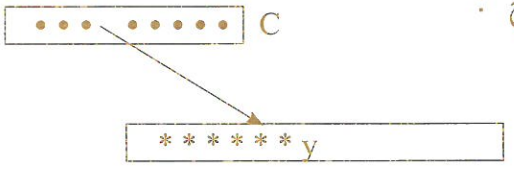
$$(d - 1) + d + 1 = 2d$$

كما يوضح ذلك الشكل التالي :

قبل الدمج :



بعد الدمج :

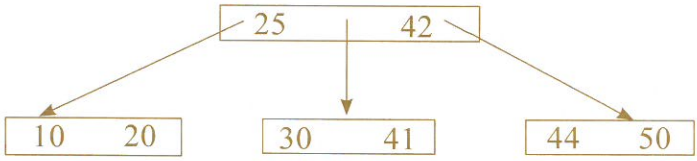


(vi) الحالة الأخيرة : مثل الحالة السابقة (iii) [في أن القيمة x موجودة في ورقة تحتوي بالضبط على عدد d من القيم ولا يوجد أخ مباشر يحتوي على عدد من القيم > d] ولكنها تختلف عنها في أن العنصر الوالد لا يحتوي على عدد من القيم > d (أي يحتوي بالضبط على عدد من القيم يساوي d) وبالتالي فعندما نأخذ من هذا العنصر الوالد قيمة (لعملية الدمج) يصبح عدد قيمه d-1 ، فنضطر للاقتراض (borrowing) – إن أمكن – أو الدمج (merging) ، وهكذا قد تستمر / تنتشر (propagate) عملية الاقتراض / الدمج نحو جذر الشجرة.

[ملاحظة : في الحالات السابقة حينما نذكر أن عنصرا يحتوي بالضبط على عدد d من القيم ونشير إلى أنه لا يمكننا أن نأخذ منه قيمة وإلا أصبح عدد قيمه d-1 أو أننا نضطر إلى أن نقترض له قيمة ليصبح عدد قيمه d ، فبالطبع يستثنى الجذر من ذلك ، أي إذا كان هذا العنصر هو الجذر ، وذلك لأنه يسمح للجذر أن يحتوي على عدد من 1 إلى 2d من القيم].

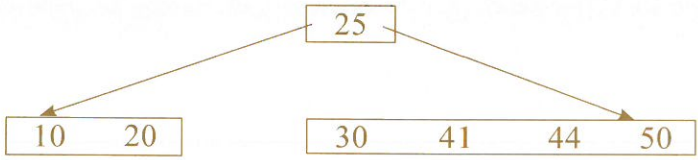
مثال ٥-٢٣ :

احذف القيمة 44 من الشجرة التالية :



الحل :

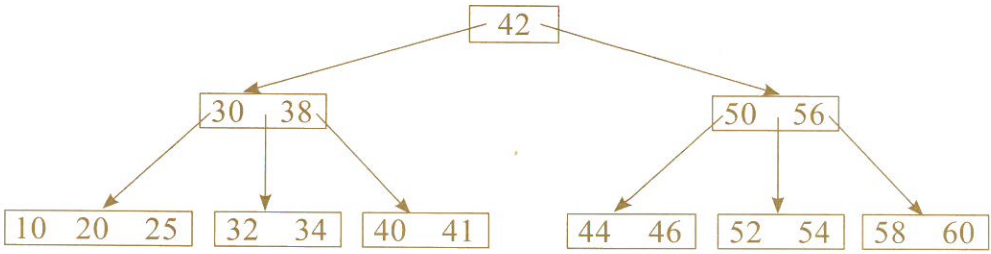
بحذف القيمة 44 وإجراء عملية الدمج نحصل على الشجرة التالية :



ملاحظة : لاحظ أن العنصر الذي يحتوي الآن على القيمة الوحيدة 25 جذر.

مثال ٥-٢٤ :

احذف القيمة 46 من الشجرة التالية :

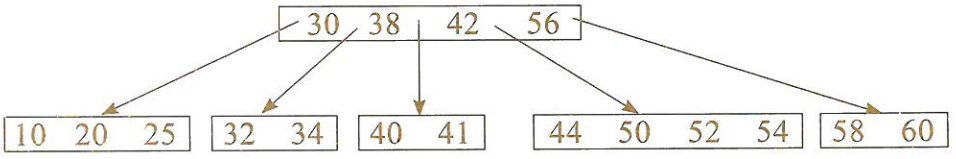


الحل :

لحذف القيمة 46 نقوم بعملية دمج القيم التالية :

44	50	52	54

ولكننا نلاحظ الآن أن العنصر الوالد (بعد أن أخذنا منه القيمة 50) أصبح يحتوي على قيمة وحيدة فقط (القيمة 56) وهو ليس جذرا ، ولا يمكننا الاقتراض له ، فنقوم بعملية دمج أخرى كما يوضح ذلك الشكل التالي :



مثال ٥-٢٥ :

من الشجرة الناتجة بعد حل المثال السابق (مثال ٥-٢٤) احذف القيم التالية بالترتيب - من اليسار لليمين - قيمة قيمة :

54 , 52 , 60 , 58 , 41

الحل :

نكتفي بذكر الإرشاد التالي ، ونترك للقارئ كتابة تفاصيل الحل.

حذف القيمة 54 : مباشر

حذف القيمة 52 : مباشر

حذف القيمة 60 : نحتاج لعملية دمج

حذف القيمة 58 : مباشر

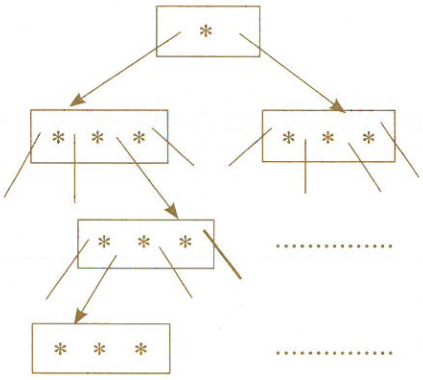
حذف القيمة 41 : نحتاج لعملية اقتراض أو عملية دمج

(بحسب الأخذ في الاعتبار الأخ الأيمن أو الأخ الأيسر)

ارتفاع الشجرة B من الرتبة d (Height of a B-tree of order d)

نستنتج فيما يلي تعبيرا يعطي أقل عدد ممكن من القيم (keys / values) في شجرة B من الرتبة d ارتفاعها h.

من المعلوم أن أقل عدد ممكن من القيم في أي عنصر / عقدة (node) في شجرة B يساوي d ، باستثناء الجذر حيث أقل عدد ممكن من القيم فيه يساوي 1. وفيما يلي أعداد العناصر في المستويات المختلفة (المستويات 1, 2, 3, 4) مع شكل توضيحي للعلاقة بين هذه المستويات وأعداد العناصر المقابلة لها ، ومن هذه الأعداد نستنتج بعدها الجدول الذي يعطي أقل عدد ممكن من القيم في أي مستوى.



رقم المستوى	عدد العناصر / العقد في المستوى
1	1
2	2
3	2 (d+1)
4	$2 (d+1)^2$

ويمكننا بسهولة استنتاج نتائج الجدول التالي الذي يعطي أقل عدد ممكن من القيم في كل مستوى من المستويات المختلفة (من المستوى رقم 1 إلى المستوى رقم h).

رقم المستوى	عدد العناصر / العقد في المستوى	أقل عدد ممكن من القيم في المستوى
1	1	1(الجذر)
2	2	2d
3	2 (d+1)	2d (d+1)
4	2 (d+1)^2	2d (d+1)^2
⋮	⋮	⋮
i	2 (d+1)$^{i-2}$	2d (d+1)$^{i-2}$
⋮	⋮	⋮
h	2(d+1)$^{h-2}$	2d (d+1)$^{h-2}$

وبالتالي يكون أقل عدد ممكن من القيم في الشجرة (من الرتبة d والتي ارتفاعها h) هو :

$$n = 1 + 2d + 2d\,(d+1) + \text{¼} \cdots + 2d\,(d+1)^{i-2} + \text{¼}\ldots + 2d\,(d+1)^{h-2}$$

$$= 1 + 2d \sum_{i=0}^{h-2} (d+1)^i$$

$$1\text{-}2d\,\frac{(d-1)^{h-1}\text{-}1}{(d-1)\text{ -}1}\,2\,(d-1)^{h-1}\text{-}1$$

$$\therefore \quad \frac{n-1}{2} = (d-1)^{h-1}$$

$$\therefore \quad h\text{ -}1 = \log_{d+1}\left(\frac{n+1}{2}\right)$$

$$\therefore \quad h = 1 + \log_{d+1}\left(\frac{n+1}{2}\right)$$

وبالتالي فلأي شجرة B من الرتبة d تحتوي على عدد n من القيم سوف يحقق ارتفاعها h العلاقة

$$h \le 1 + \log_{d+1}\left(\frac{n+1}{2}\right) = 1 + \left\lfloor \log_{d+1}\left(\frac{n+1}{2}\right) \right\rfloor$$

وذلك لأن h يجب أن تكون عددا صحيحا.

فإذا فرضنا مثلا أن d = 100 فإن الجدول التالي يعطي قيم h (أقصى ارتفاع) المقابلة لبعض قيم n

n	h
10^3	2
10^4	2
10^5	3
10^6	3
10^7	4

ملاحظة ١ :

الجدول التالي يقارن بين أداء (performance) كل من شجرة البحث الثنائية التامة / الكاملة (perfect / complete) BST وشجرة B من الرتبة d = 100 ، حيث يعطي قيم الارتفاع (height) (أو زمن الوصول access time) لكل منهما المقابلة لعدة قيم لـ n (عدد القيم الموجودة بالشجرة) { ملاحظة في حالة الشجرة BST الكاملة / التامة يعطى الارتفاع بالتعبير $\lfloor \log_2 n \rfloor + 1$ }.

n	h (full / complete BST)	h (B-tree)
10^3	10	2
10^4	14	2
10^5	17	3
10^6	20	4
10^7	24	4

ومن الجدول نلاحظ أن زمن الوصول (أو الارتفاع) أقل بكثير في حالة أشجار B. وعموما فإنه في حالة أشجار B ذوات قيم d الكبيرة يكون طول المسار (path length) / الارتفاع قصيرا جدا.

ملاحظة ٢ :

يمكننا تمثيل عنصر / عقدة (node) في شجرة B باستخدام إحدى الطرق التالية :

أ) منظومة سجلات عددها 2d.

ب) قائمة مترابطة (مفردة SLL أو مزدوجة DLL أو ...).

ج) شجرة بحث ثنائية BST.

وفي حالة استخدام منظومة قد تنخفض درجة الاستفادة من الذاكرة (storage utilization) إلى ٥٠ ٪ ، حيث أن عدد القيم المخزونة في العنصر قد يصل إلى الحد الأدنى d.

تمرينات رقم 0

٥-١ إذا علم أن نتيجة الاجتياز الترتيبي (inorder) لشجرة ثنائية هي :

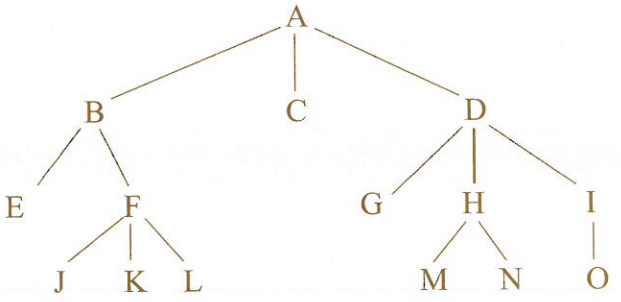
DGBAHEICF

ونتيجة الاجتياز لاحق الترتيب (postorder) للشجرة نفسها هي :

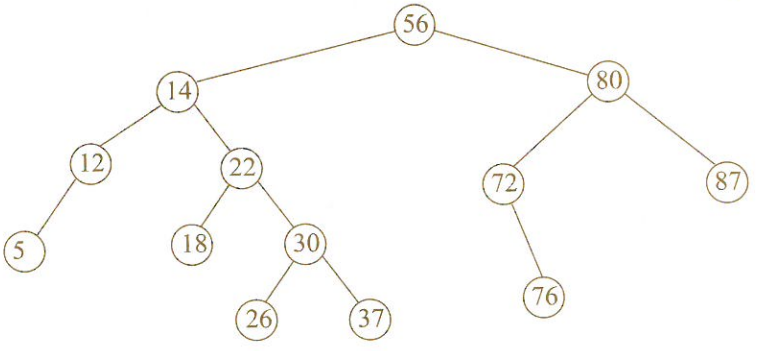
GDBHIEFCA

ارسم هذه الشجرة.

٥-٢ اكتب نتيجة كل من الاجتياز سابق الترتيب (preorder) والاجتياز لاحق الترتيب (postorder) للشجرة التالية :



٥-٣ أ) نفرض أن لدينا الشجرة T المتوازنة ارتفاعا (AVL) المبينة بالشكل التالي :



الشجرة T

i) اكتب قيمة معامل التوازن لكل عنصر من عناصر الشجرة.

ii) أدخل ٢ ثم ٢٨ في الشجرة .

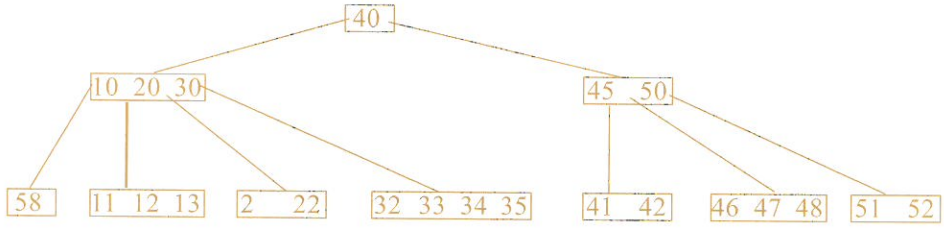
ب) ارسم شجرة البحث الثنائية BST الناتجة عن حذف القيمة 56 من شجرة البحث الثنائية BST المرسومة في الجزء أ) (أي الشجرة T).

٥-٤ المطلوب إنشاء (أ) شجرة بحث ثنائية BST ، (ب) شجرة بحث ثنائية متوازنة ارتفاعا (أي شجرة AVL) بإدخال كل من المتتابعتين التاليتين من العناصر (keys) بالترتيب المعطى :

i) 8 9 10 2 1 5 3 6 4 7 11 12

ii) 1 2 3 4 5 6 7 8 9 10 11 12

٥-٥ أدخل 14 ثم 15 ثم 31 في الشجرة المتوازنة (B) من الرتبة الثانية (d = 2) المبينة في الشكل التالي :



٥-٦أ) أدخل العناصر التالية بالترتيب المعطى في شجرة B خالية ذات الرتبة الثانية (B-tree of order 2) :

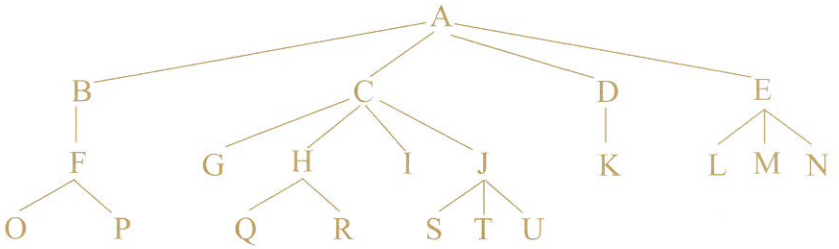
12 100 81 19 5 6 75 12 66 120

54 2 210 70 60 1 3 10 27 90

ب) احذف العناصر التالية بالترتيب المعطى من الشجرة B التي حصلت عليها في أ) :

1 2 3 70 75 100 120 210

٥-٧ أ) ارسم الشجرة الثنائية B1 التي تمثل الشجرة T1 التالية (أي حوِّل الشجرة التالية إلى شجرة ثنائية) .



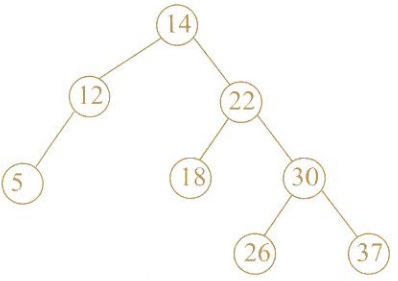
الشجرة T_1

٢) حوِّل الغابة التالية F_1 (ذات الأشجار الثلاث) إلي شجرة ثنائية B_2 .

غابة F_1

٣) اكتب نتيجة الاجتياز سابق الترتيب لكل من الأشجار B2 ، B1 ، T1 والغابة F_1 .

٥-٨ i) اكتب معامل التوازن لكل عنصر من عناصر الشجرة AVL المتوازنة التالية :



ii) أدخل العنصر 24 في الشجرة ثم أعد اتزانها إن لزم الأمر .

iii) اكتب نتيجة الاجتياز الترتيبي لعناصر الشجرة بعد الإضافة وإعادة الاتزان .

٥-٩ يمكن ترتيب عناصر مجموعة مكونة من n عدد صحيح ترتيبا تصاعديا عن طريق بناء شجرة بحث ثنائية BST تحتوي على عناصر المجموعة ثم اجتياز هذه الشجرة اجتيازا ترتيبيا.

١) اكتب إجراء يقرأ قيم عناصر مجموعة S مكونة من n عدد صحيح ثم يطبق هذه الطريقة لترتيب عناصر S ترتيبا تصاعديا ، بفرض أنه متوفر لدينا إجراء insert لإدخال عنصر في شجرة BST .

٢) اعط مثالا لأسوأ حالة لتنفيذ هذا الإجراء بالنسبة لمجموعة مكونة من خمسة عناصر (n = 5) .

أجوبة تمرينات رقم ١

١ - ١

i) $\lim_{n\to\infty} \frac{5n^2-6n}{n^2} = 5 \neq 0,\infty$

ii) $\frac{n!}{n^n} = \frac{1}{n} \cdot \frac{2}{n} \cdot \frac{3}{n} \cdots\cdots \frac{(n-1)}{n} \cdot \frac{n}{n}$

$\frac{n!}{n^n} > 0 \qquad (n \geq 1)$

$\Rightarrow \lim_{n\to\infty} \frac{n!}{n^n} \geq 0 \qquad (1)$

$\frac{2}{n} \cdot \frac{3}{n} \cdots \frac{(n-1)}{n} \cdot \frac{n}{n} < 1 \qquad (n > 2)$

$\Rightarrow \frac{n!}{n^n} < \frac{1}{n} .1 = \frac{1}{n}$

$\Rightarrow \lim_{n\to\infty} \frac{n!}{n^n} \leq \lim_{n\to\infty} \frac{1}{n} = 0 \qquad (2)$

ومن العلاقتين (2) ,(1) نستنتج أن

$\lim_{n\to\infty} \frac{n!}{n^n} = 0 \neq \infty$

iii) $\lim_{n\to\infty} \frac{2n^2.2^2+n \log n}{n^2 2^n} = \lim_{n\to\infty} 2 + \lim_{n\to\infty} \frac{\log n}{n.2^n}$

$= 2 + \lim_{n\to\infty} \frac{1}{n} \cdot \lim_{n\to\infty} \frac{\log n}{2^n}$

$= 2+0.\left\{ \lim_{n\to\infty} \frac{\frac{1}{n}}{2^n (1n2)^n} \right\} = 2 \neq 0,\infty$

iv) $\lim_{n\to\infty} \frac{4(\log n)^3}{n^2} = \lim_{n\to\infty} \frac{12(\log n)^2 . \frac{1}{n}}{2n.1n2}$

$= \frac{6}{1n2} \lim_{n\to\infty} \frac{(\log n)^2}{n^2} = \frac{6}{(1n2)^2} \lim_{n\to\infty} \frac{\log n . \frac{1}{n}}{n}$

$= \frac{6}{(1n2)^2} \lim_{n\to\infty} \frac{\log n}{2^n} = \frac{6}{(1n2)^3} \lim_{n\to\infty} \frac{\frac{1}{n}}{2n} = 0$

v) $$\lim_{n\to\infty} \frac{2^{2^n}-6*2^n}{2^{2^n}} = \lim_{n\to\infty} \left\{1+ \frac{6*2^n}{2^{2^n}}\right\}$$

$$=1+6 \lim_{n\to\infty} \frac{2^n-1n2}{2^{2^n}.2^n(1n2)^2} = 1 \neq 0,\infty$$

vi) $$\lim_{n\to\infty} \frac{6n^3}{(\log n=1)n^3} = 6 \lim_{n\to\infty} \frac{1}{\log n+1} = 0$$

vii) $$\lim_{n\to\infty} \frac{n^{k+\varepsilon} + n^k \log n}{n^{k+\varepsilon}} =1+ \lim_{n\to\infty} \frac{n^k \log n}{n^{k+\varepsilon}}$$

$$=1+ \lim_{n\to\infty} \frac{n^k .\log n}{n^k .n^\varepsilon} =1+ \lim_{n\to\infty} \frac{\log n}{n^\varepsilon}$$

$$=1+ \frac{1}{\varepsilon 1n2} \cdot \lim_{n\to\infty} \frac{\frac{1}{n}}{\varepsilon.n^{\varepsilon-1}}$$

$$=1+ \frac{1}{\varepsilon 1n2} \cdot \lim_{n\to\infty} \frac{1}{n^\varepsilon} =1+ 0 \qquad (\varepsilon > 0 \text{ لاحظ أن})$$

$$= 1 (\neq 0,\infty)$$

viii) $$\cdot \lim_{n\to\infty} \frac{33n^3 +4n^2}{n^3} = 33 \neq 0$$

٢-١

i) $$\lim_{n\to\infty} \frac{10n^2 +9}{n} = \infty$$

ii) $$\lim_{n\to\infty} \frac{n^2 \log n}{n^2} = \lim_{n\to\infty} \log n = \infty$$

iii) $$\lim_{n\to\infty} \frac{n^2 /\log n}{n^2} = \lim_{n\to\infty} \frac{1}{\log n} = 0$$

iv) $$\lim_{n\to\infty} \frac{n^2 /\log n}{n^2} = \lim_{n\to\infty} \frac{1}{\log n} = 0$$

$$=1+ 6 \lim_{n\to\infty} \frac{(1.5)^2 .1n\ 1.5}{1} = \infty$$

$$=1+6\lim_{n\to\infty}\frac{(1.5)^n\cdot 1n\ 1.5}{1}=\infty$$

٣-١

$$\lim_{n\to\infty}\frac{(\log n)^k}{n}=\lim_{n\to\infty}\frac{k(\log n)^{k-1}\cdot\frac{1}{n}}{1}$$

$$=k\lim_{n\to\infty}\frac{(\log n)^{k-1}}{n}=k(k-1)\lim_{n\to\infty}\frac{(\log n)^{k-2}}{n}$$

$$=\ldots=k(k-1)(k-2)\ldots(k-\overline{k-1})\lim_{n\to\infty}\frac{(\log n)^{k-k}}{n}$$

$$=k!\lim_{n\to\infty}\frac{1}{n}=0$$

٤-١ (أ)

$$f(n)=o(g(n))\Leftrightarrow\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$$

$\Leftrightarrow$ g تزايد أبطأ من تزايد f

(ب)

$$\lim_{n\to\infty}\frac{10{,}000\ 1n\ n}{n/1n\ n}\quad\lim_{n\to\infty}\frac{(1n\ n)^2}{n}$$

$$=10{,}000\lim_{n\to\infty}\frac{2(1n\ n).\frac{1}{n}}{1}=10{,}000\lim_{n\to\infty}\frac{1n\ n}{n}$$

$$=20{,}000\lim_{n\to\infty}\frac{\frac{1}{n}}{1}=0$$

(ج) من الجزء (ب) :

$$1n\ n<n/1n\ n$$

$$n>1\Rightarrow 1n\ n<n\ 1n\ n$$

$$n\gg 1\Rightarrow 1n\ n>1\Rightarrow 1n>1>\frac{1}{1n\ n}$$

$$\Rightarrow n\ 1n\ n>\frac{n}{1n\ n}$$

$$\therefore\ 1n\ n<n/1n\ n<n\ 1n\ n$$

١-٥ العلاقات (i), (iii), (iv) صحيحة.

العلاقة (ii) خاطئة

مثال مناقض :

نفرض : $f(n) = n^2, T_1(n) = 5n^2, \quad T_2(n) = 2n^2$

$$T_1(n) - T_2(n) = 3n^2$$

$$\lim_{n\to\infty} \frac{T_1(n) - T_2(n)}{f(n)} = \lim_{n\to\infty} \frac{3n^2}{n^2} = 3 \neq 0$$

$$\Rightarrow T_1(n) - T_2(n) \neq o(f(n))$$

١-٦ $O(n)$

١-٧

i	j	عدد العمليات (ضرب وإسناد)
1	1→ 1	1
2	1→ 2	2
3	1→ 3	3
⋮		⋮
n	1→ n	n

العدد الكلي للعمليات : $\#\text{ operations} = 1 + 2 + 3 + \ldots + n$

$$= \frac{n(n+1)}{n} \cong \frac{n^2}{2} = O(n^2)$$

١-٨

i) $O(n)$ ii) $O(n^2)$ iii) $O(n^3)$

iv)

قيم i	عدد مرات تنفيذ العروة
1	
2	1
4	2
8	3
⋮	⋮
2^k	k

يتم الخروج من عروة while عندما تحقق قيمة i العلاقة $i > n$. وحيث أنه عندما يتم تنفيذ العروة عدد k من المرات تكون قيمة i قد وصلت إلى

$i = 2^k$ ، فبالتالي نستنتج أنه عندما يتم الخروج من العروة يكون قد تم تنفيذ العروة عدد k من المرات ،

حيث $i=2^k > n$ أي $k > \log_2 n$

أي أن درجة التعقيد تساوي O (lolg n)

٩-١ $T_A = 150\ n \log_2 n \quad ; \quad T_B = n^2$

(أ) $$\lim_{n\to\infty} \frac{T_A}{T_B} = \lim_{n\to\infty} \frac{150\ n \log n}{n^2} = 150 \lim_{n\to\infty} \frac{\log n}{n}$$

$$= \text{const} * \lim_{n\to\infty} \frac{\frac{1}{n}}{1} = 0$$

أي أن البرنامج A أفضل.

(ب) $$\frac{T_A}{T_B} = \frac{150\ n \log_2 n}{n^2} = \frac{150 \log_2 n}{n} = \frac{150}{n} . \log_2 n$$

$\frac{150}{n} > 1$ (since $n < 100$) &

$\log_2 n > 1$ (for $n \geq 2$)

$\Rightarrow \frac{T_A}{T_B} > 1 \Rightarrow$ البرنامج B أفضل.

(جـ) $$\frac{T_A}{T_B} = \frac{150 \times 10^3 \times \log 10^3}{10^6} = 0.45 \log_2 10$$

$\cong 1.5 > 1 \Rightarrow$ البرنامج B أفضل.

(د) كي يكون البرنامج B أسرع تنفيذاً من البرنامج A بالنسبة لجميع المدخلات ، يجب أن يتحقق الشرط :

$$n^2 < 150\ n \log_2 n \quad \forall n$$

$$\Rightarrow n < 150 \log_2 n \quad \forall n$$

وهذا غير صحيح بالنسبة لقيم n الكبيرة ، مثلا إذا فرضا $n = 10^6$

$150 \log_2 10^6 = 150 * 6 * 3.2 \cong 3000$

$1000.000 \not< 3000$

أي أن الاحتمال المذكور غير ممكن التحقق.

١٠-١

i	j	k	عدد العمليات (ضرب وإسناد)
1	1	2→ n	n-1
2	2	3→ n	n-2
3	3	4→ n	n-3
⋮	⋮	⋮	⋮
n-1	n-1	n→ n	1
n	n+1	n+1→n	0

العدد الإجمالي للعمليات :

(total # of operations) =

$$1 + 2 + 3 + \ldots + (n-2) + (n-1)$$

$$= \frac{n(n+1)}{2} - n$$

$$= \frac{n^2}{n} = \frac{n}{2} = 0\left(n^2\right)$$

أجوبة تمرينات رقم ٢

٢-١

```
void Compare2Lists (List A, List B, int m, int n, int& result)
{
      int i = 0;
      bool finish = false;

      while ((!finish) && (i < n) && (i < m)
            if (ai = = bi
                  i ++;
            else
                  finish = true;
      if finish
            if (ai > bi)
                  result = 1;
            else
                  result = -1;
      else
            if (n = = m)
                  result = 0;
            else if (n > m)
                  result = 1
               else
                  result = -1
}
```

٢-٣ (أ)

```
const int max _term = 100;
struct term
      {
            float coef;
            int exp;
      }
term poly [max_term];
```

(ب)

```
void multiply _by_const (poly & P, int n, float c)
{
      int i;
```

```
        for (i = 0; i < n; i ++)
            P[i].coef = c * P[i].coef;
}
```

$P(x) = \sum a_i x^{i} = \sum coef \cdot x^{exp}$ (جـ)

$P(x) = \sum i\ a_i x^{i-1}$

```
        coef ← exp * coef
        exp ← exp -1
void poly - differentiation (poly & P, int & n)
{
        int i;
        for (i = 0; i < n; i ++)
            {
                P[i]. coef = P[i].exp * P[i].coef;
                P[i].exp = P[i].exp -1 ;
            }
        if (P[n].exp = = 0)
            n = n-1;
        }
```

(د) درجة التعقيد = O (n)

٤-٢

	row	col	value
0	0	1	1.1
1	0	3	1.2
2	1	0	2.1
3	1	3	1.1
4	1	4	1.3
5	2	0	3.1
6	2	1	3.2
7	3	2	4.1
8	4	1	5.1
9	4	2	5.2

A

0	0
2	1
5	2
7	3
8	4
10	5

XA

(أ) ٥-٢

k	row	col	value
0	0	0	-1
1	0	1	1
2	1	0	2

0	0
2	1
4	2

3	1	2	4	4	3
4	3	2	4	5	4
5	4	4	5	6	5
	A			U≡XA	

(ب) (i) الطريقة المباشرة :

0, 0, -1
1, 0, 1 ←

0, 0, -1
0, 1, 2
1, 9, 1
2, 1, 4
2, 3, 4
4, 4, 5

(ii) طريقة نقل أعمدة A إلى صفوف B :

0, 0, -1
0, 1, 2
1, 0, 1
2, 1, 4
2, 3, 4
4, 4, 5

(ii) طريقة نقل أعمدة A إلى صفوف B :

	S	U	$B \equiv A^T$	
0	0→1→2	0→1→2	0, 0, -1	b_0
1	0→1	2→3	0, 1, 2	b_1
2	0→1→2	3→4→5	1, 0, 1	b_2
3	0	5	2, 1, 4	b_3
4	0→1	5→6	2, 3, 4	b_4
			4, 4, 5	b_5

٦-٢

	0	1	2	3	4	69
0						
1		-2			1	
2						
3						
4		-1			4	
5						
6						4
69						

A

	0	1	2	3	4	59
		3				
		6		2		
						3

B

=

	0	1	2	3	4	59
		0			2	0
		21			8	0
						12

C

	row	col	value
c_0	1	4	2
c_1	4	1	21
c_2	4	4	8
c_3	6	59	12

٧-٢ (أ)

```
typedef     int index [Max];
void print_row (sparse_matrix & A, index & AX, int A)
{// function to print elements of row no. i.
      int k ;
      if (AX [i+1] > AX [i])
            {
                  cout << "Elements of row no.< << "are:" << endl ;
                  for (k = AX [i]; k <= (AX [i+1] -1); k++)
                        cout << i << "" << A [k]. col << ""
                        << A [k]. value << ´,´);
            }
      else
            cout << "All elements of row nuber"
                        << i << "are zeros" << endl ;
}
```

عدد مرات التكرار في تنفيذ عبارة for يساوي :

AX[i+1] -1 - AX[i] +1 = AX[i+1] - AX[i]

أي أن درجة تعقيد خوارزمية طباعة عناصر الصف رقم i تساوي عدد العناصر غير الصفرية في هذا الصف .

(ب)

```
void print_row (element A [], int AX [], int j, m, n, t)
// function to print elements of column number j.
{
      int l ;
      bool found = false ;
      cout << "Elements of column number"  << setw (6)
            << j << "are :" << endl ;
      for (l = 0; l <t; l++)
```

```
        for (l = 0;  l < t;   l ++)
            if (A [ l ].col = = j)
                found = true ;
                count << A [ l ].row << "" << j << " "
                << A [ l ].value << ' , ' <<  endl;
            }
        if (! found)
            cout << "all zeros" << endl;
}
```

واضح أن درجة تعقيد الخوارزمية تساوي O (t) → t (عدد العناصر غير الصفرية في المصفوفة) .

(جـ)

```
void print_element (element A[ ], int AX[ ], int i, j)
// function to print the value of the element A [i,j]
{
    bool found = false ;
    int k ;
    k = AX [i] ;
    cout << "Element number (" << setw (6)
        << i << ' , ' << set (6) << j <<") is: " << endl;
    while ((! found) && (k <= (AX [i+1] - 1)))
        {
            if (A [k].col = = j)
                {
                    found = true ;
                    cout << i << " " << j << " "
                        << A [k].value << endl;
                }
            else
                k = k + 1 ;
        }
    if (! found)
        cout << i << " " << j << " " << ' 0 ' << endl;
}
```

درجة التعقيد (في أسوأ حالة) تساوي عدد ا لعناصر غير الصفرية في الصف رقم i .

أولاً : التمثيل المتناثر

عدد العناصر غير الصفرية = 5n

$\Rightarrow Cs = 3*5n = 15n$

Storage (XA) = n+1

$\Rightarrow C_{sparse}\ Cs|_{total} = 15n + n + 1 = 16n + 1$

ثانياً : التمثيل ثاني البعد 2D

$C_{2D} = n * n = n^2$

من الواضح أنه لقيم n الكبيرة تكون

$C_{sparse} < C_{2D}$

عند n = 16 تتساوى تقريبا سعتا التخزين

عند n < 16 يكون التمثيل ثنائي البعد 2D أفضل حيث سعة التخزين تكون أقل .

عند n > 16 يكون التمثيل المتناثر (مع منظومة مؤشرات) أفضل حيث سعة التخزين تكون أقل .

٩-٢ (أ)

n (A) = 3 * 4 = 12
n (B) = 5 * 21 = 105
n (C) = 8 * 11 * 16 = 1408

(ب)

(i) address $(A_{i,j})$ = + (i - l_1) (u2 - l2 + 1) + (j - l_2)
address $(A_{2,2})$ = 400 + (2-1) (4-1+1) + (2-1)
= 405

(ii) address $(B_{2,2})$ = 600 + (2+2) (22-2+1) + (2-2)
= 684

(iii) Storage (C) = n(C) = 1408

١٠-٢

$u_0 = 3,\ u_1 - 4,\ \alpha = 314$

a) Storage = u0 u1 = 3 x 4 = 12 words

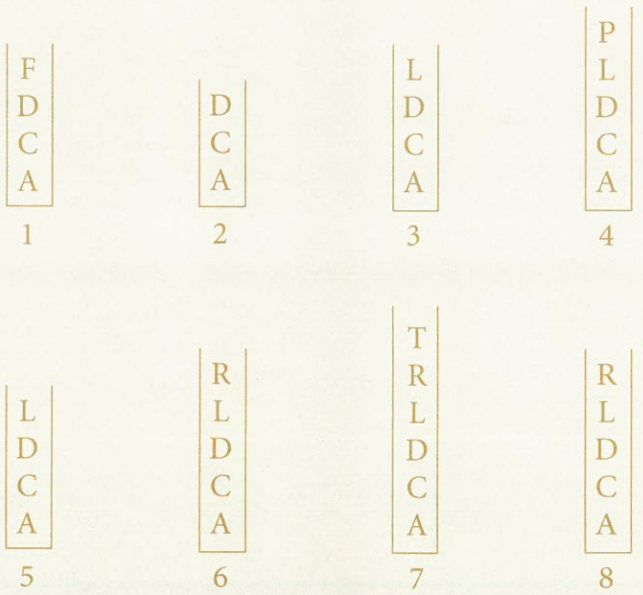
b) Address $(A_{ij}) = \alpha + (i\ u_1 + j)$
$= 314 + (4i + j)$
$= 4\ i + j + 314$
$\Rightarrow a = 4, \quad b = 1, \quad c = 314$

١١-٢

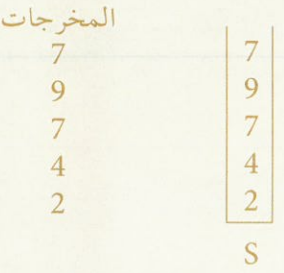
```
void DisplayColumn3 (element A [ ], int m, int n, int t)
// function to display elements of column number 3.
{
      int i, k;
      float Val [m];      // array of the elements of col. no. 3

      for (i = 0; i < m; i ++)
            Val [i] = 0;

      for (k = 0; k < t; k ++)
            if (A [k].col = = 3)
                  Val [A[k].row] = a [k].value;

      for (i = 0; i < m; i ++)
            cout << Val [i];
}
```

أجوبة تمرينات رقم ٣

٣-١

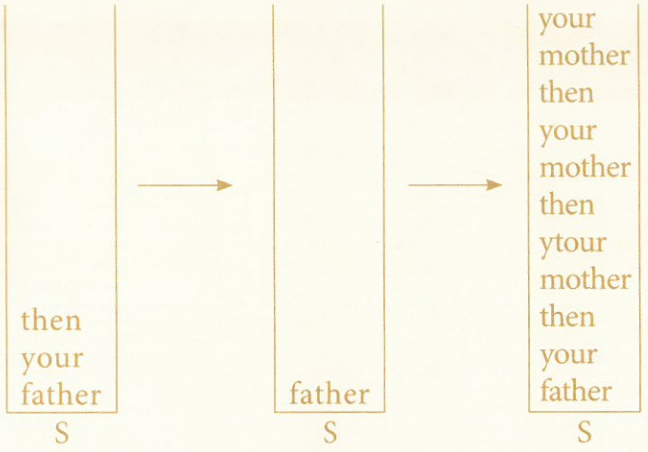


٣-٢

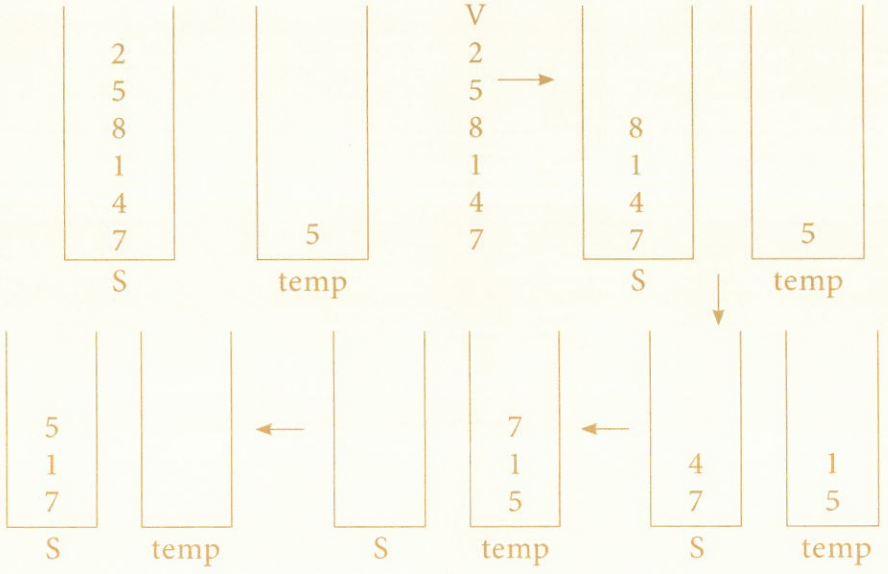


٣-٣ تبديل قيمتي المتغيرين X, Y .

٤-٣



٥-٣ (أ)



وظيفة الخوارزمية : تحذف عنصراً من أعلى الرصة وتترك العنصر الذي يليه ، وتتكرر هذه العملية حتى تصبح الرصة فارغة .

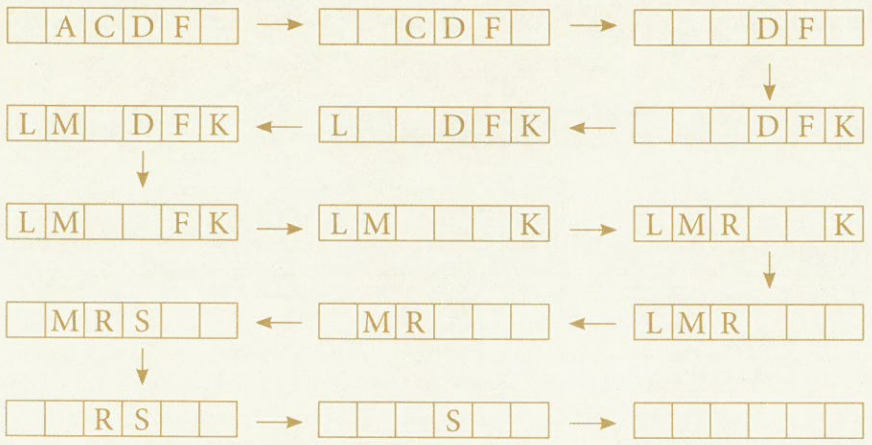
(ب)

```
int StackSum (StackType  S)
{
```

```
        int sum, e;
        StackType temp ;
        create_s (temp);
        sum = 0 ;
        while (! empty_s (S))
                {
                        pop (S, e);
                        sum = sum + e ;
                        push (temp, e)
                }
        while (! empty (temp))
                {
                        pop (temp, e);
                        push (S, e)
                }
        return (sum) ;
}
```

٦-٣ يحدث خطأ فيض (overflow) بعد ٩ عمليات ، حيث أنه في العملية التالية لا نستطيع دفع عنصرين لأنه بعد دفع العنصر الأول تصبح الرصة ممتلئة ولا يمكننا دفع العنصر الثاني .

٧-٣

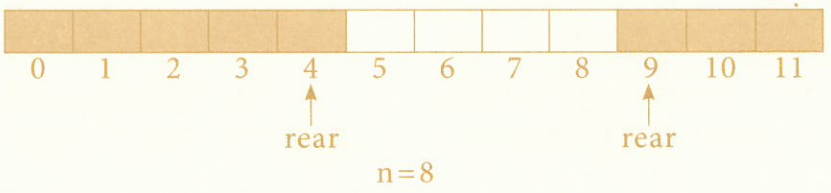


لا يمكن تنفيذ الأمر (16) بحذف عنصر لأن الطابور أصبح فارغاً .

٨-٣ i) (أ)

$$n = rear - front + 1 = 9 - 3 + 1 = 7$$

(ب)



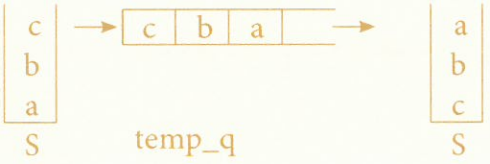
ii)

$$n = (rear - front + 1) \% N$$

للتحقق :

$$[\ n = (4 - 9 + 1) \% 12 = -4 \% 12 = 8$$

٩-٣



نقوم برفع/ بحذف محتويات الرصة المعطاة S وإضافتها / إدخالها على التوالي في طابور مؤقت temp_q ، ثم حذفها من الطابور وإعادتها على التوالي إلى الرصة S ، ونظراً لأن الإضافة في الطابور تكون عند نهايته بينما الحذف منه يكون عند بدايته ، فلذلك تؤدي العملية السابقة (رفع محتويات الرصة وإضافتها الى طابور ثم إعادتها إلى الرصة) إلى عكس محتويات الرصة ، وهو المطلوب .

الخوازرزمية :

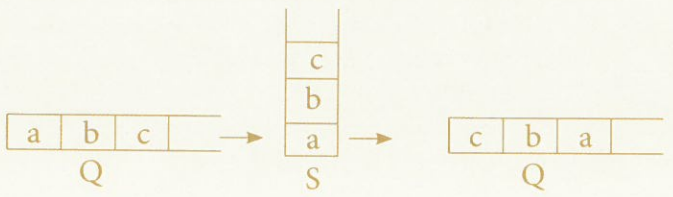
```
void reversing (stack& S)
{
      queue temp_q;
      elt_type e ;
      create_q (temp_q);
      while (! empty_s (S)
            {
                  pop (S, e) ;
```

```
                insert (temp_q, e)
            }
        while (! empty_q (temp_q__
            {
                del (gemp_q, e );
                push (S, e );
            }
}
```

٣-١٠

```
void retrieve_element (stack& S, int k, elt_type& e)
{
    stack temp_s;
    elt_type t;
    int i;
    if (empty_s (S))
        cout << "empty stack, elt. does not exist"
            << endl,
    else
        {
            create_s (temp_s);
            for (i = 1; i <= (k-1); i ++)
            {
                pop (S, t);
                push (temp_s, t);
            }
            tops_s (S, e);
            for (i = 1; i <= (k-1); i ++)
            }
                pop (temp_s, t),
                push (S, t);
            }
        }
}
```

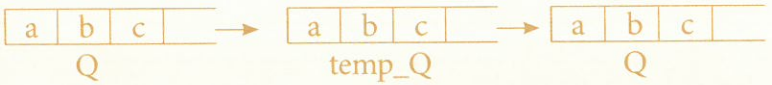
٣-١١



٣-١١ نقوم بحذف محتويات الطابور وإدخالها في الرصة ثم رفعها من الرصة وإعادتها إلى الطابور ، وبالتالي تنعكس ا لعناصر الموجودة بالطابور .

```
Void Reverse (queue& q) ;
{
        stack S ;
        elt_type X ;
        create_s (S) ;
        while (! empty_q (Q))
        {
                del (Q, X);
                push (S, X);
        }
        while (! empty_s (S))
        {
                pop (S, X) ;
                insert (Q, X);
        }
}
```

٣-١٢




```
int Size_Q (queue Q;
{
        queue temp_Q;
        elt_type e;
        int n ;
        if empty_q (Q)
                cout << "Queue Q is empty" << endl,
        else
                {
                        n = 0 ;
                        create_q (temp_Q) ;
                        while (! empty_q (Q))
                        {
                                del (Q, e) ;
                                insert (temp_Q, e) ;
                                n = n + 1 ;
                        }
                        while (! empty_q (temp_Q))
                                {
```

```
                        del (temp_Q, e ) ;
                        insert (Q, e )
                    }
                return n ;
            }
}
```

٣-١٣ (أ)

```
void appending (queue& Q1, Q2) ;
{
    elt_type e;
    while (! empty_q (Q2))
        {
            del (Q2,  e) ;
            insert (Q1, e) ;
        }
}
```

(ب)

```
bool search (queue& Q, elt_type x)
{ // to search for elt. x in queue Q. functions is
  // true if x is found, false if not found.
    queue temp_Q ;
    bool found ;
    found = false ;
    if (front_of_q (Q) = = x)
        found = true ;
    else
        {
            creat_q (temp_Q);
            while ((! found) && (! empty_q (Q))
                {
                    del (Q, e) ;
                    insert (temp_Q, e) ;
                    if (x = = e)
                        found = true ;
                }
            while (! empty_q (Q))
                {
                    del (Q, e) ;
                    insert (temp_Q, e) ;
                }
```

```
                while (! empty_q (temp_Q))
                    {
                        del (temp_Q, e) ;
                        insert (Q, e) ;
                    }
            }
        return found
}
```

(ج)

```
void merge (queue& Q1, Q2, Q3)
{
        elt_type x, y ;
        create_q (Q3) ;
        while ((! empty_q (Q1)) && (! empty_q (Q2))
            {
                del (Q1, x) ;
                insert (Q3, x) ;
                del (Q2, y) ;
                insert (Q3, y) ;
            }
        while (! empty_q (Q1))
            }
                del (Q1, x) ;
                insert (Q3, x) ;
            }
        while (! empty_q (Q2))
            {
                del (Q2, y) ;
                insert (Q3, y) ;
            }
}
```

١٤-٣

a) AB*C@
b) 0A - B + C - D +
c) A0B - * C +
d) AB + D * EF AD * + / + C +
e) AB && C || EF > ! ||
f) ABC < ! && CD > || ! CE < ||

٣-١٥

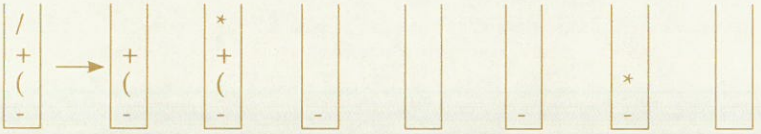
قيمة التعبير = 10

٣-١٦ (أ) (ab-) * (de/) → ab - de / *

(ب) a * (bd +) / e - f* (g + hk/) →
(abd + * e/) - (fghk / + *) →
abd + * e/fghk / + * -
a b c * + d e * f + g * +

٣-١٧ (i

(ii A B C D / E * + - F G * -



٣-١٨ (i

قيمة التعبير = 96

(ii

قيمة التعبير = 15

٣-١٩ (أ)

```
void create ( )
{
      top 1 = -1 ;
```

```
        top 2 : = 1000;
}
```

(ب)

```
bool empty 1
{
        return (top 1 = = -1) ;
}
```

(ج)

```
bool empty 2
{
        return (top2 == 1000);
}
```

(د)

```
bool full
{
        return (top2 = = top1 + 1);
}
```

(هـ)

```
void push1 (int e)
{
        if full
                cout << "stack is full, can not add" << endl ;
        else
                {
                        top1 = top1 + 1 ;
                        stack [top1] = e ;
                }
}
```

(و)

```
void push2 (int e)
{
        if full
                cout << "stack is full, can not add" << endl ;
        else
                {
                        top2 = top2 - 1 ;
                        stack [top2] = e ;
                }
```

```
}
```

(ز)

```
void pop1 (int& e)
{
        if empty1
                cout << "stack s1 is empty, can not pop" << endl ;
        else
                {
                        e = stack [top1];
                        top1 = top1 -1 ;
                }
}
```

(ح)

```
void pop2 (int& e)
{
        if empty2
                cout << "stack s2 is empty, can not pop" << endl ;
        else
                {
                        e = stack [top2];
                        top1 = top2 -1 ;
                }
}
```

٣- ٢٠ (أ)

```
void push (int i, int e)
{ // to add e to the i the stack.
        if (top [i] = = bot [i+1])
                cout << "stack no. i is full" << endl ;
        else
                {
                        top [i] = top [i] + 1 ;
                        stack [top [i]] = e ;
                }
}
```

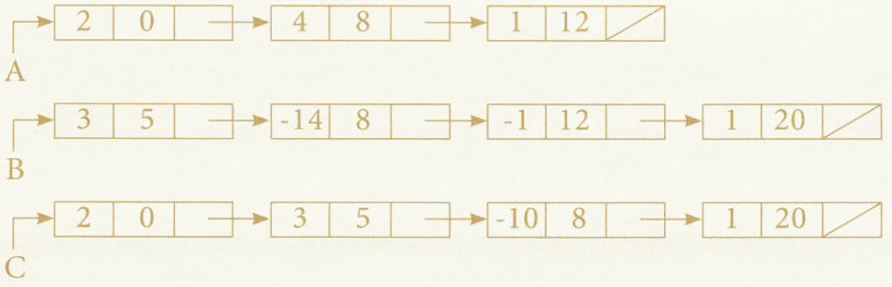
(ب)

```
void pop (int i, int& e)
{ // to delete topmost item of stack no. i.
        if (bot [i] = = top [i])
                cout << "stack no. i is empty" << endl ;
```

```
        else
                {                                                    ( ز )
                        e = stack [top [i]]
                        top [i] = top [i] - 1 ;
                }
}

                                                                     ( ج )
void shift3 ( )
{ // to shift stack no. 3 one location to the left.
        int i ;
        for (i = bot [3]; i <= (top [3] - 1); i ++)
                // top[3] = = bot [4], since stack no. 3 is full.
                stack [i] = stack [i+1] ;
        bot [3] = bot [3] - 1 ;
        top [3] = top [3] -1 ;
}

                                                                     ( د )
int free
{ // counts no. of empty/free locations in the array stack ;
  // i.e., not occupied by any elt. of the 10 stacks.
int i, count ;
count = 0;
        for (i - 1; i <= 10; i + +)
                count = count + (bot [i+1] - top [i]) ;
        return count ;
}

                                                                     حل آخر
{
        int, n     // n is the no. of occupied locations.
        n = 0 ;
        for (i = 1; i<= 10; i + +)
                n = n + (top [i] - bot [i]) ;
        return (1000 - n)
}
```

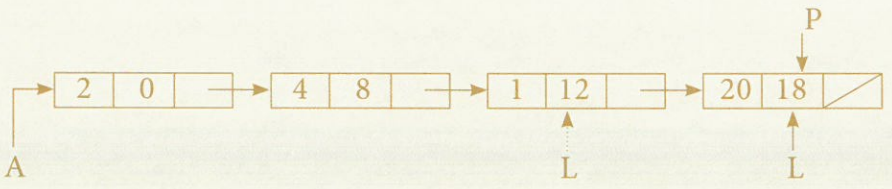
أجوبة تمرينات رقم ٤

٤-١ (أ)



$$C(X) = 2 + 3\,x^5 - 10\,x^8 + x^{20}$$

(ب) (i)



قبل الاستدعاء بعد الاستدعاء

(ii) وظيفة الإجراء Test : إضافة السجل الذي يمثل الحد cx^e إلى نهاية القائمة المترابطة التي تمثل الحدودية المعرفة بالمؤشرين f, last . [فالاستدعاء في (i) يؤدي إلى إضافة الحد $20x^{18}$ إلى الحدودية A (x)]

(ج) (i)

(ii) وظيفة الإجراء Exam : إلغاء القائمة المترابطة التي تمثل الحدودية المعطاة، وذلك عن طريق إلغاء عناصرها عنصراً عنصراً ابتداءً من العنصر الأول .

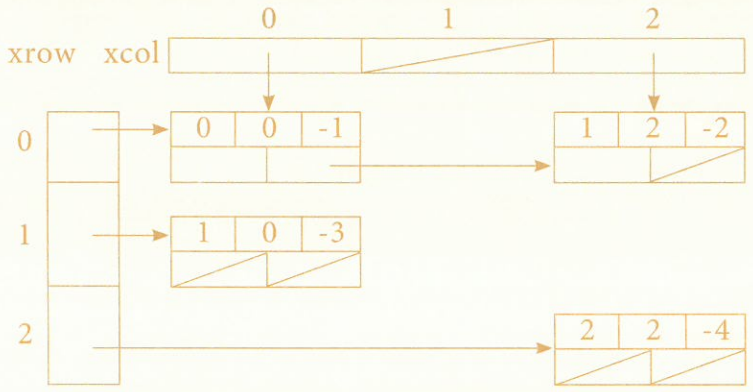
(iii)

```
void multiply_const (poly& D, int c)
{
        poly  p;
        p = D ;
        while (p != NULL)
                {
                        P → coef = p → coef * c ;
                        p = p → next ;
                }
}
```

٢-٤ (أ)

```
typedef node  *node_ptr ;
strucct node
        {
                int row, col ;
                float value ;
                node_ptr down, right ;
        } ;
typedef node_ptr pointer_array [100] ;
pointer_array xrow, xcol ;
```

(ب)



(ج)

Storage (A) = m + n + 5 * t
= n + n + 5 * (7n)
= 37 n

(د)

```
void mul_col_c (pointer_array xcol,
                    int j, float c)
{
    node_ptr col_p ;
    col_p = xcol [j] ;
    while (col_p != NULL)
        {
            col_p → value = col_p → value * c ;
            col_p = col_p → down ;
        }
}
```

درجة تعقيد الإجراء = عدد العناصر غير الصفرية في العمود رقم j

Complexity = O (# nonzero elts. in col. j)

(هـ)

```
float dot_product (pointer_array xrow, int i, j)
{
    node_ptr p, q ;
    float sum ;
    sum = 0.0 ;
    p = xrow [i] ;
    q = xrow [j] ;
    while ((p != NULL) && (q != NULL))
        if (p → col = = q → col)
            {
                sum = sum + p → value * q → value ;
                p = p → right ;
                q = q → right ;
            }
        else if (p → col < q → col)
            p = [ → right ;
        else  // q → col < p → col
            q = q → right ;
    return sum ;
}
```

درجة تعقيد الدالة (في أسوأ حالة) = عدد العناصر غير الصفرية في الصف رقم i

+ عدد العناصر غير الصفرية في الصف رقم j

worst_case complexity = O (# nonzero elts. in row i +
nonzero elts. in row j)

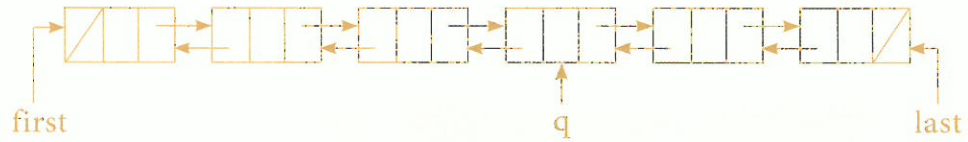
٣-٤ (أ)


```
node_ptr previous (node_ptr ; first, node_ptr q)
{
      node_ptr t, pr ;
      if (first = = NULL)
            cout << "empty list" << endl ;
      else if (q = = first)
            retuurn (NULL) ;
      else
            {
                  t = first ;
                  while ( t → next ! = q)
                        t = t → next ;
                  return (t)
            }
}
```

(ب)

```
typedef nod *node_ptr ;
struct node
      {
            data_type data ;
            node_ptr next, prev ;
      };
```

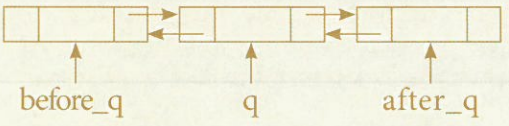
(ج)




```
node_ptr previous (node_ptr ; first, last, q)
{
      node_ptr t, pr ;
```

```
    if (q = = first)
        pr = NULL ;
    else
        pr = q → prev ;
    return (pr) ;
}
```

(د) نفرض أن القائمة المترابطة المزدوجة معرفة بوسيطيها : first (وهو مؤشر إلى بداية القائمة) ، وlast (وهو مؤشر إلى نهايتها) . والمطلوب حذف العنصر q من هذه القائمة (انظر الشكل المرسوم في حل الجزء السابق ج) . هناك ٤ حالات محتملة لهذا العنصر q في القائمة : إما أن يكون عنصراً ما داخل القائمة (أي هناك على الأقل عنصر قبله ، وعلى الأقل عنصر بعده في القائمة ، وإما أن يكون هو العنصر الأول first في القائمة (وهناك ما هو بعده) ، وإما أن يكون هو العنصر الأخير last في القائمة (وهناك ما هو قبله) ، وإما أن يكون هو العنصر الوحيد في القائمة . والإجراء التالي يقوم بحذف العنصر q من القائمة في كل حالة من هذه الحالات الأربع .

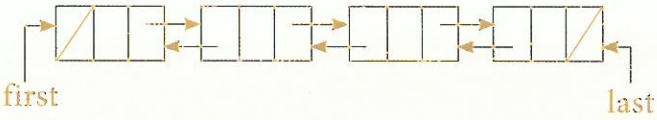



```
void del (node_ptr& first, last, q)
{
    node_ptr before_q, after_q ;
    before_q = q→ prev ;
    after_q = q → next ;

    if ((before_q !=NULL) && (after_q != NULL))
        {
            before_q → next = q → next ;
            after_q → prev = q → prev ;
            delete q
        }
    else if ((before_q = = NULL) && (after_q != NULL))
```

```
                    // q is the first node
            {
                    first = q → next ;
                    first → prev = NULL ;
                    delete q ;
            }
else if ((before_q != NULL) && (after_q = = NULL))
                    // q is the last node
            {
                    last = q → prev ;
                    last →  next = NULL ;
                    delete q ;
            }
      else   // q is the only node
            {
                    first = NULL ;
                    last = NULL ;
                    delet q ;
            }
}
```

٤-٤ (أ) يمكننا تمثيل الطابور ذي النهايتين بقائمة مترابطة مزدوجة ، حيث يسهل الإضافة أو الحذف عند أي من النهايتين ، ونستخدم مؤشرين أحدهما «first» يشير إلى النهاية الأولى للطابور (القائمة المترابطة) والآخر «last» يشير إلى النهاية الأخرى .

```
typedef node *node_ptr;
struct node
      {
            data_type data ;
            node_ptr prev , next ;
      };
node_ptr first, last ;
```



(ب)

```
void inset_deque (node_ptr& first, last,
            node_ptr t,   // elt. to be inserted
```

```
            char end_point    // ' F ' : first end ,
                              // ' L ' : other end)
    {
            if (end_point = = ' F ')
                  if (first = = NULL)
                        {
                              first = t ;
                              last = t
                        }
                  else
                        {
                              t → next = first ;
                              first → prev = t;
                              first = t;
                        }
            else if (end_point = = 'L')
                  if (last = = NULL)
                        {
                              first = t;
                              last = t;
                        }
                  else
                        {
                              t → prev = last;
                              last → next = t;
                              last = t;
                        }
}
```

٤-٥ (أ)

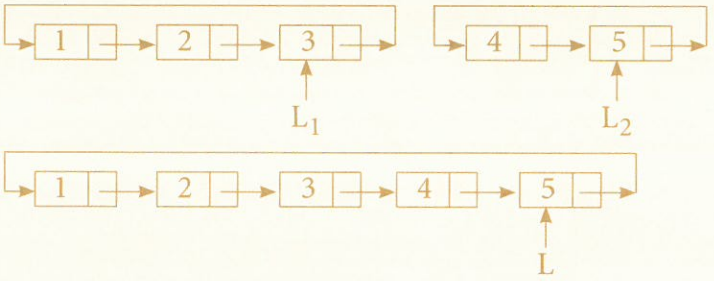
```
void print_circular_list (node_ptr& L)
{
      node_ptr p;
      if (L !=NULL
            {
                  p = L next ;        // begin at first elt.
                  while (p ! = L)     // L is the last elt.
                        {
                              cout << p → data ;
                              p = p → next ;
                        }
```

```
                cout << p → data);  // information of last elt.
            }
}
```

(ب)

```
int length (node_ptr& L)
{
    node_ptr p ;
    int n ;
    n = 0 ;
    if (L != NULL)
        {
            p = L →  next ;
            while (p ! = L)
                {
                    n = n + 1 ;
                    p = p →  next ;
                }
            n = n + 1 ;
        }
    return n ;
}
```

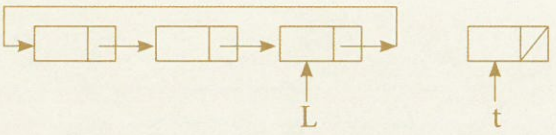
(ج)




```
void append (node_ptr& L1, L2, L);
{
    // to append the circular list L2 to the end of the circular
    // list L1 yielding the circular list L.

    node_ptr p ;
    p = L1 →  next ;
    L1 →  next = L2 →  next ;
```

```
        L2 → next = p ;
        L = L2 ;
}
```

٦-٤ (أ)

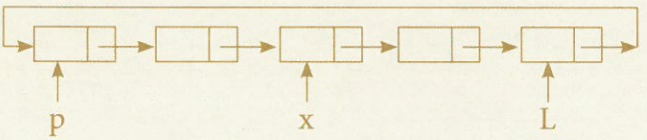



```
void insert_at_end_cir (node_ptr& t, L)
{
        // to insert node t at end of cir. list L.
        t → next = L → next
        L → next = t ;
        L = t ;
}
```

(ب)

```
void del_first_cir (node_ptr& L) ;
{
        node_ptr p ;
        if (L != NULL)
                {
                        p = L → next ;
                        L → next = p → next;
                        delete p
                }
        else
                cout << "List is empty" << endl;
}
```

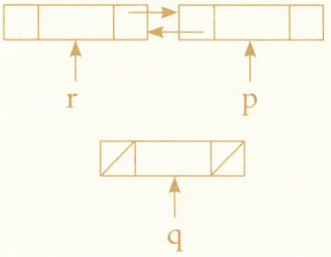
٧-٤




```
void delete_element (node_ptr& L, node_ptr  x)
{
        node_ptr p ;
```

```
        p = L → next;          // p points to first elt.
        if (p = = x)           // delete first elt. , which is x.
                {
                        L → next = p → next ;
                        delet x ;
                }
        else    // move p up to location just before x
                {
                        while (p → next !=x)
                                p = p → next ;
                        if (x = = L)     // delete last elt., which is x.
                                {
                                        p → next = L → next ;
                                        delete x ;
                                        l = p ;
                                }
                        else  // delete x
                                {
                                        p → next = x → next ;
                                        delete x ;
                                }
                }
}
```

٨-٤



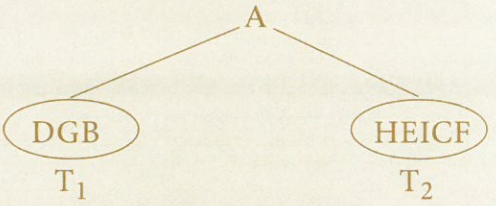

```
r = p → prev ;
if (r ! = NULL)
        {
                r → next = q ;
                q → next = p ;
        }
p → prev = q ;
q → next = p ;
```

٩-٤

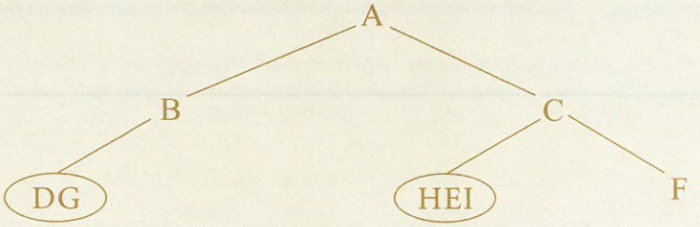
```
void poly_erase (poly& t)
{
        poly p ;
        while (t != NULL)
                {
                        p = t → next ;
                        delete t ;
                        t = p ;
                }
}
```

أجوبة تمرينات رقم ٥

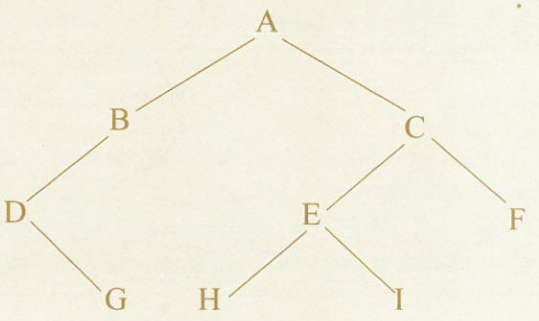
٥-١ من نتيجة الاجتياز لاحق الترتيب نرى أن جذر الشجرة الثنائية المطلوبة هو A ، ثم من نتيجة الاجتياز الترتيبي نستطيع تحديد عناصر كل من الشجرتين : الفرعية اليسرى T_1 والفرعية اليمنى T_2 ، هكذا :



ثم من نتيجة الاجتياز لاحق الترتيب نرى أن جذر T_1 هو B وأن جذر T_2 هو C ، ثم من نتيجة الاجتياز الترتيبي نستطيع تحديد عناصر كل من الشجرتين : الفرعية اليسرى والفرعية اليمنى لكل من الشجرتين الفرعيتين : T_1, T_2 كما يلي :



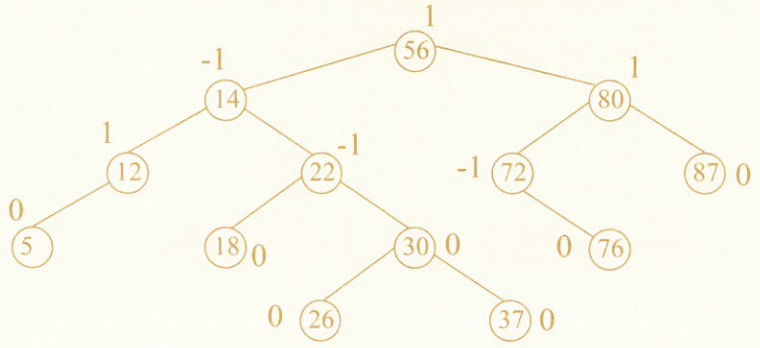
ونستمر بنفس الكيفية إلى أن نحدد جميع عناصر الشجرة الثنائية المطلوبة :

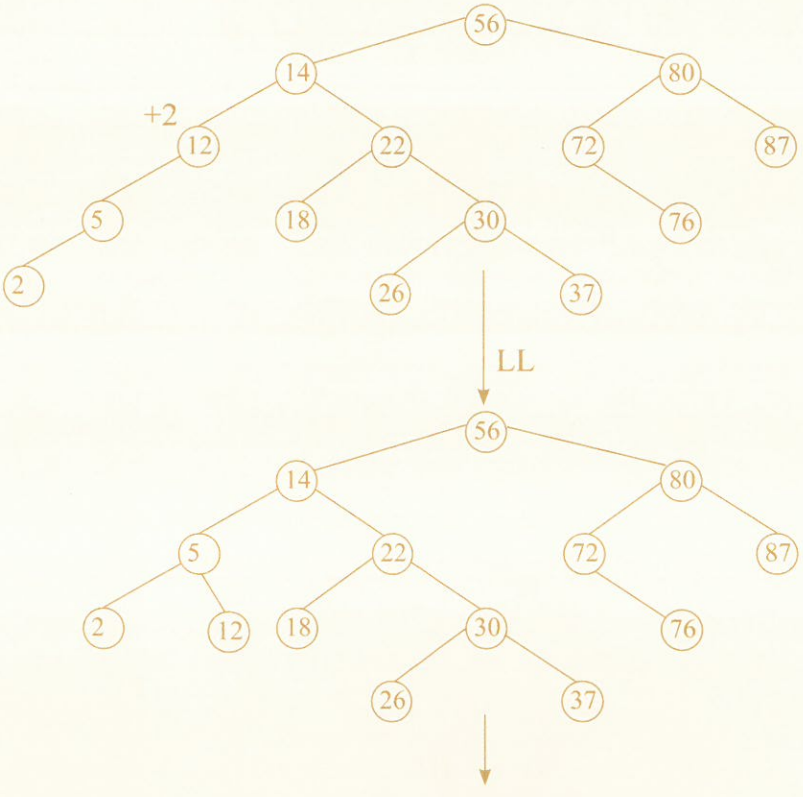


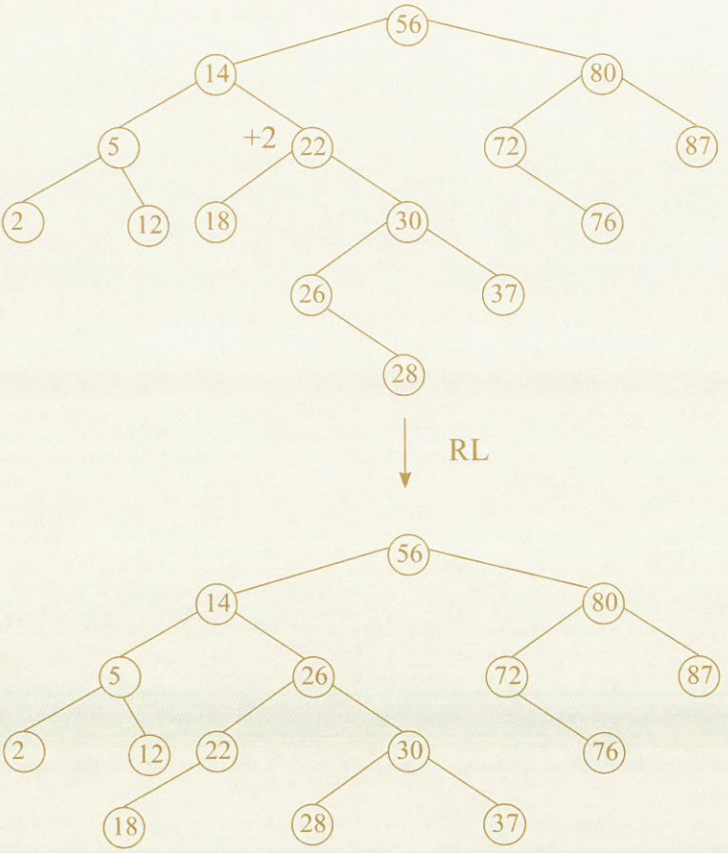
٢-٥ نتيجة الاجتياز سابق الترتيب : ABEFJKLCDGHMNIO

نتيجة الاجتياز لاحق الترتيب : EJKLFBCGMNHOIDA

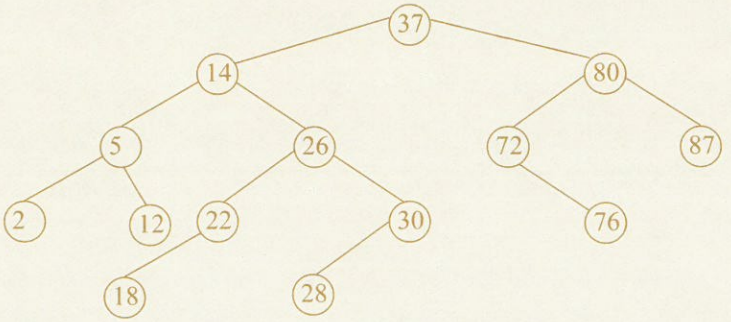
٣-٥ (أ)





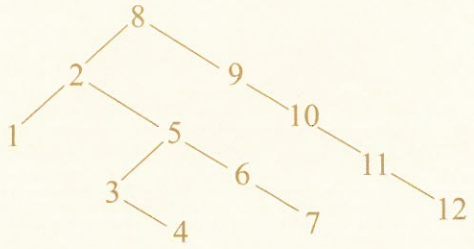

56
14
80
5
+2
22
72
87
2
12
18
30
76
26
37
28
RL
56
14
80
5
26
72
87
2
12
22
30
76
18
28
37

(ب)

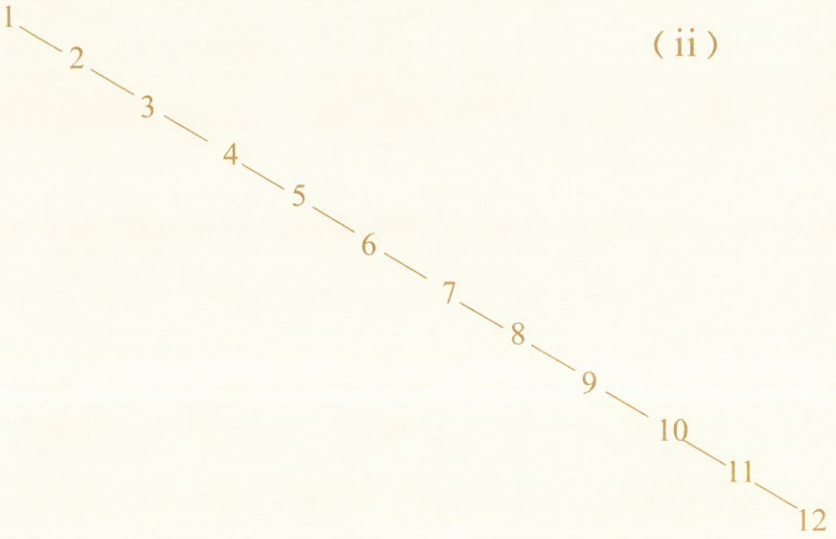

37
14
80
5
26
72
87
2
12
22
30
76
18
28

٥-٤ (أ) إنشاء شجرة بحث ثنائية BST :

(i)

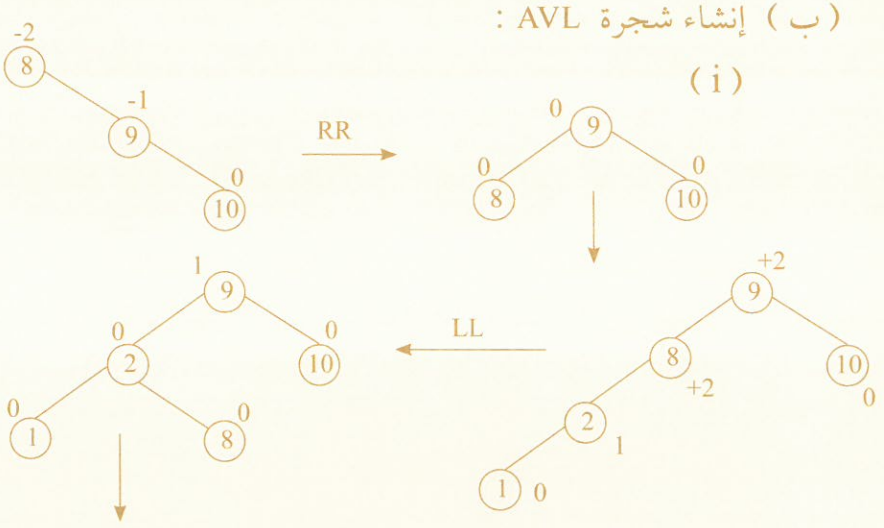


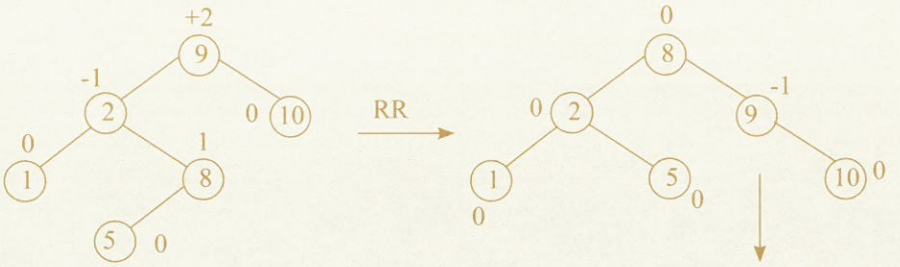
(ii)

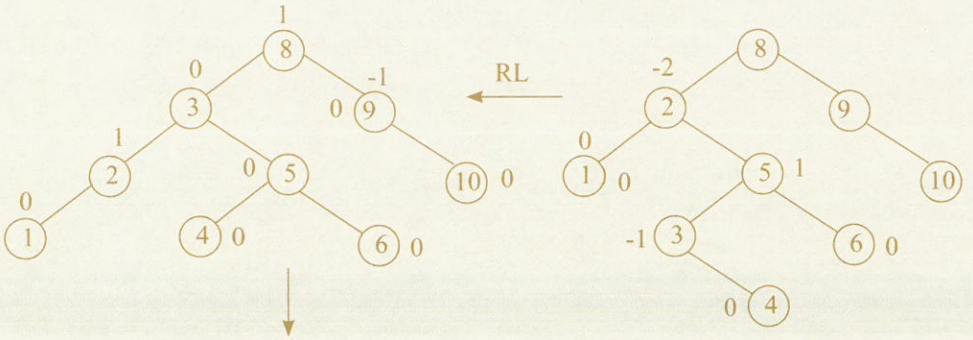


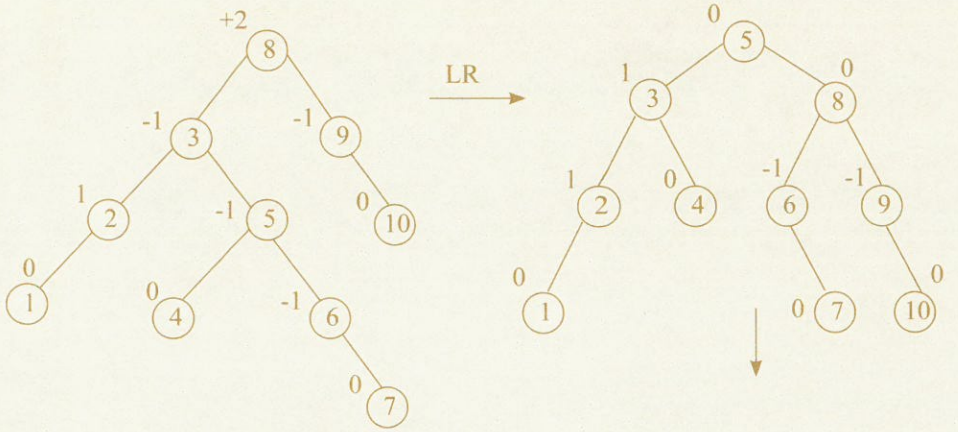
(ب) إنشاء شجرة AVL :

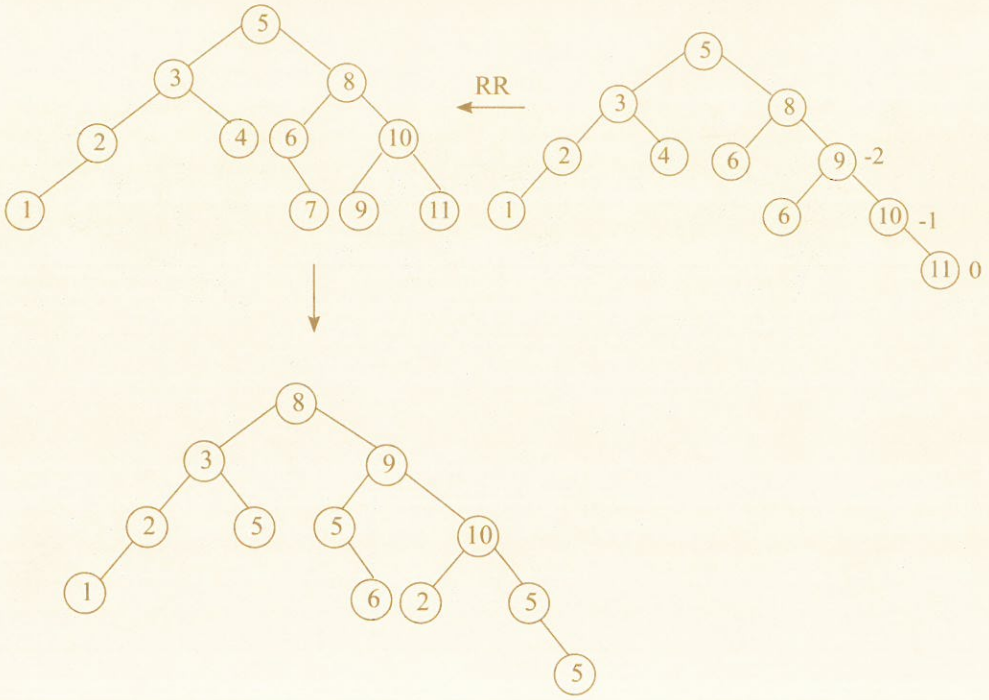
(i)



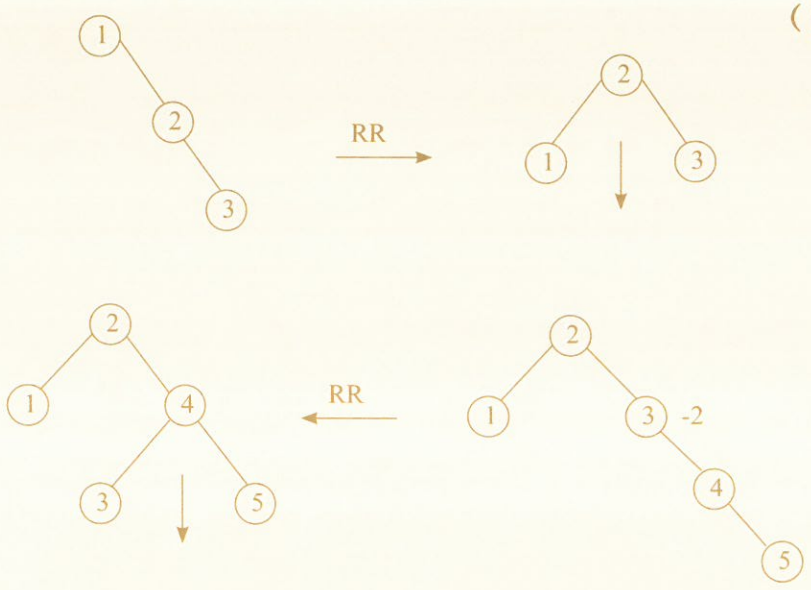

+2
9
-1
2
0
10
0
1
1
8
5
0
RR
0
8
0
2
-1
9
1
0
5
0
10
0
0

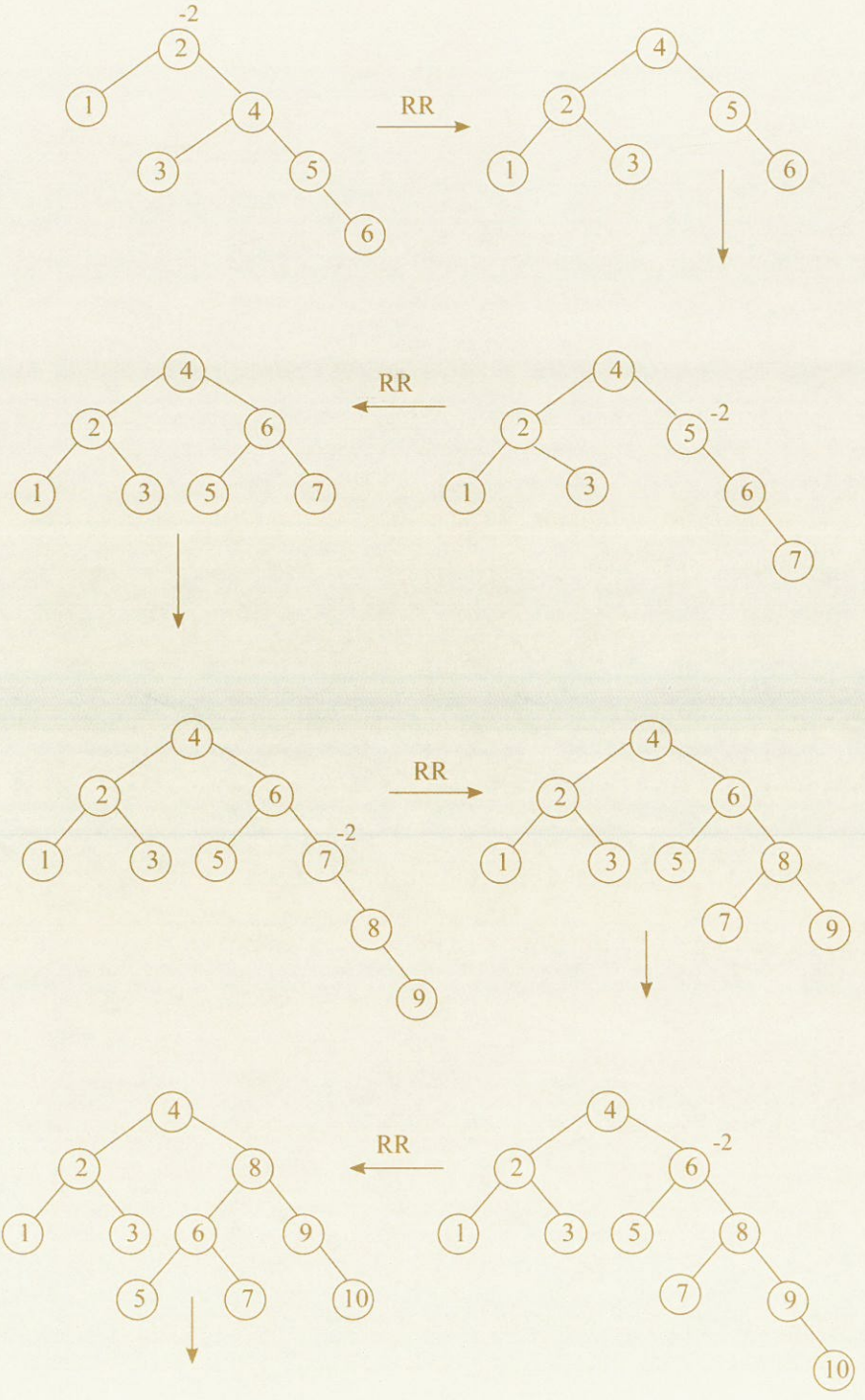

1
8
0
3
-1
0
9
1
2
0
5
10
0
0
1
4
0
6
0
RL
8
-2
2
9
0
1
0
5
1
10
-1
3
6
0
0
4

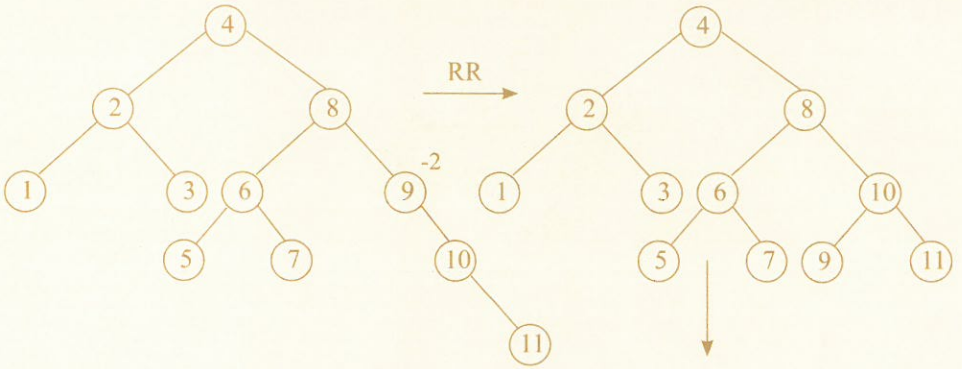

+2
8
-1
3
-1
9
1
2
-1
5
0
10
0
1
0
4
-1
6
0
7
LR
0
5
1
3
0
8
1
2
0
4
-1
6
-1
9
0
1
0
7
0
10
0

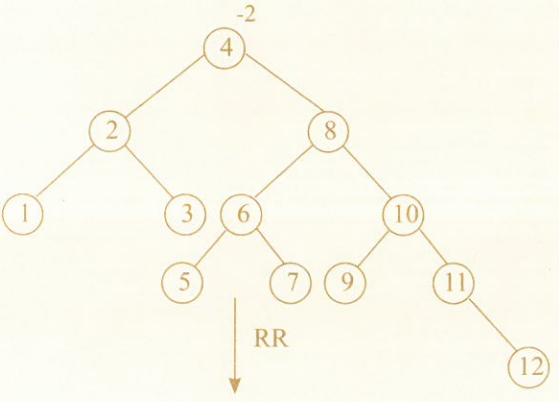


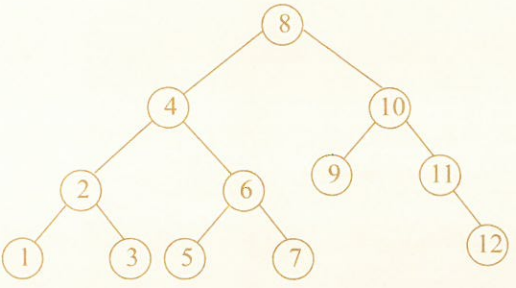
(ii)



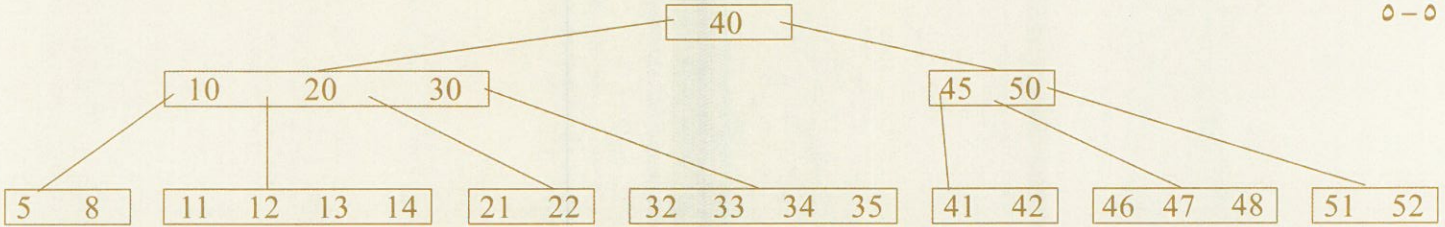

-2
RR
RR
-2
-2
RR
-2

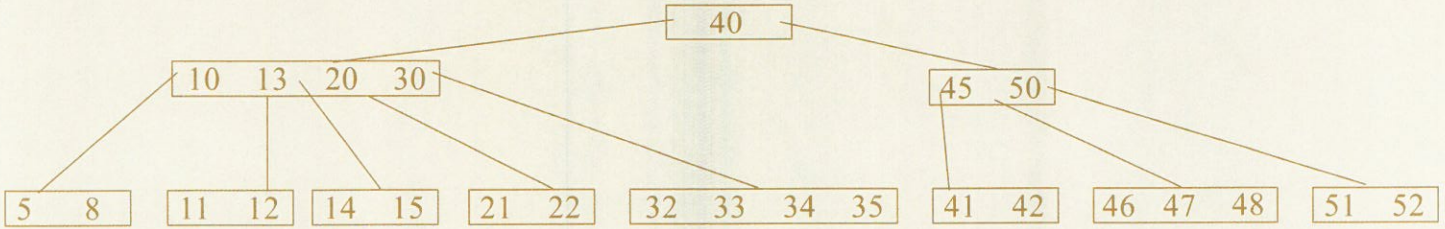

4
2
8
1
3
6
9
5
7
10
11
RR
-2
4
2
8
1
3
6
10
5
7
9
11

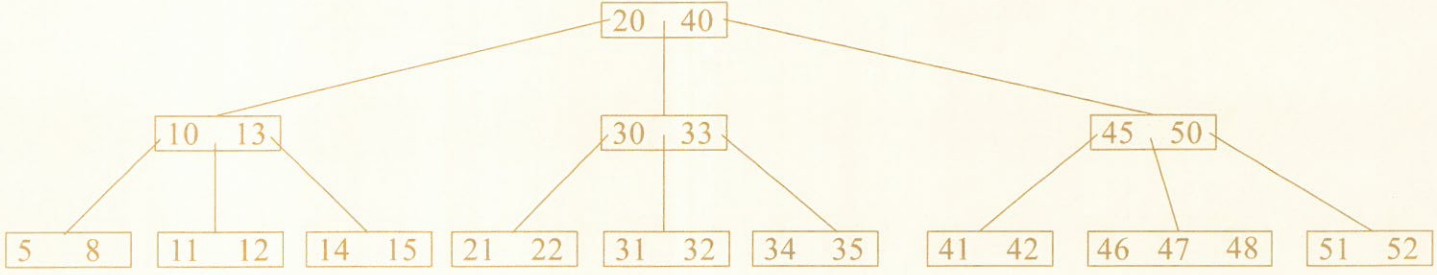

-2
4
2
8
1
3
6
10
5
7
9
11
12
RR


8
4
10
2
6
9
11
1
3
5
7
12

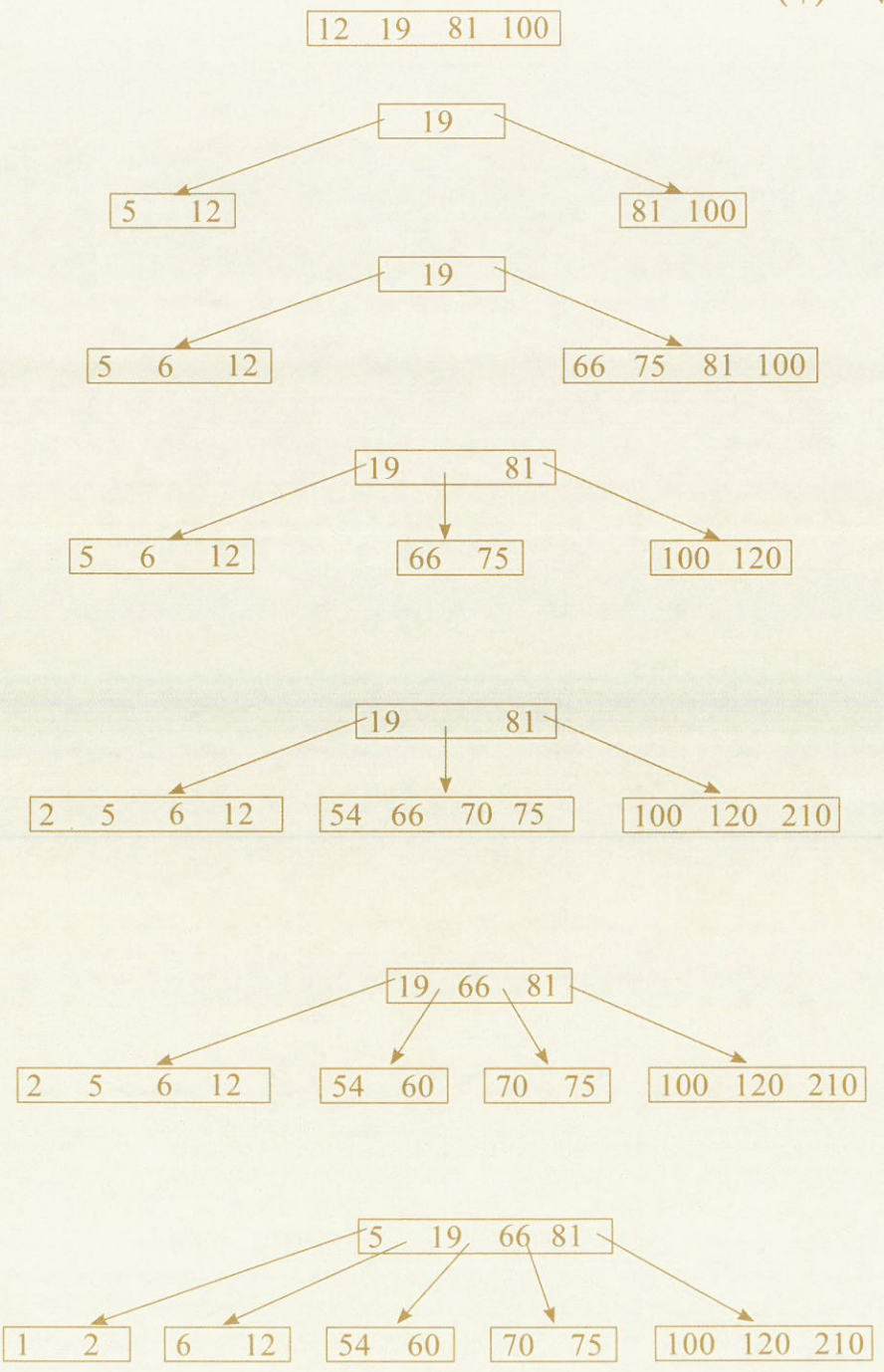
٥-٥

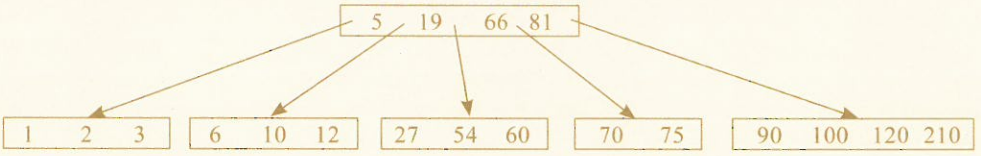

40
10 20 30
45 50
5 8
11 12 13 14
21 22
32 33 34 35
41 42
46 47 48
51 52


40
10 13 20 30
45 50
5 8
11 12
14 15
21 22
32 33 34 35
41 42
46 47 48
51 52

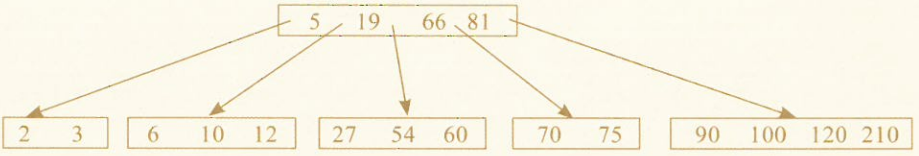

20 40
10 13
30 33
45 50
5 8
11 12
14 15
21 22
31 32
34 35
41 42
46 47 48
51 52

٦-٥ (أ)


12 19 81 100
19
5 12
81 100
19
5 6 12
66 75 81 100
19 81
5 6 12
66 75
100 120
19 81
2 5 6 12
54 66 70 75
100 120 210
19 66 81
2 5 6 12
54 60
70 75
100 120 210
5 19 66 81
1 2
6 12
54 60
70 75
100 120 210



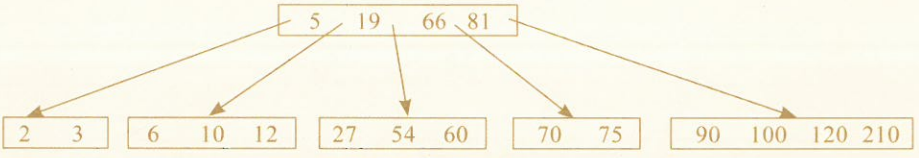
(ب) حذف 1 : مباشر



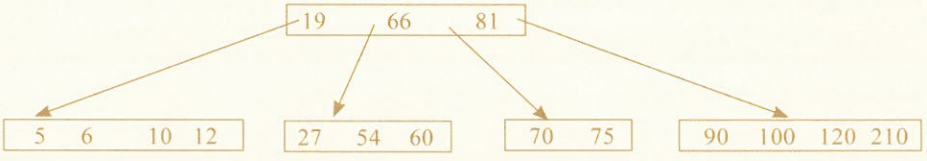
حذف 2 : نفترض من الأخ المجاور

القيم المتتابعة : 3 5 6 10 12

القيمة الوسطى

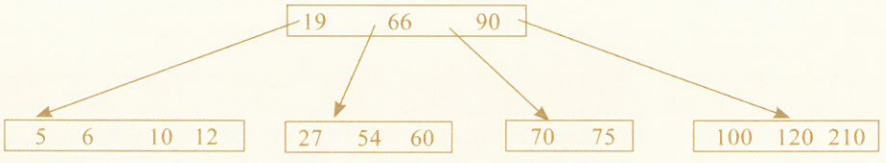


حذف 3 : دمج



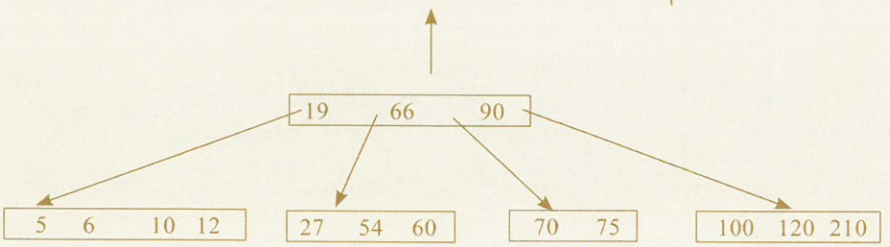
حذف 70 : نفترض من الأخ المجاور الأيمن

القيم المتتابعة : 75 81 90 100 120 210

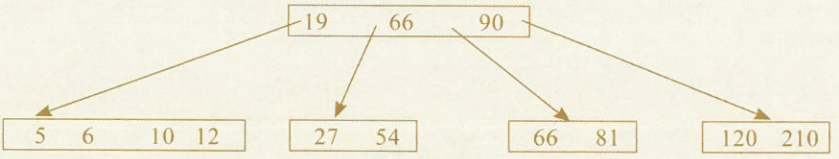


حذف 75 : نفترض من الأخ المجاور الأيسر

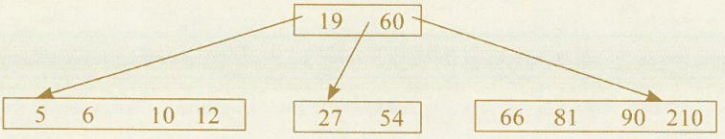
القيم المتتابعة : 81 66 60 54 27



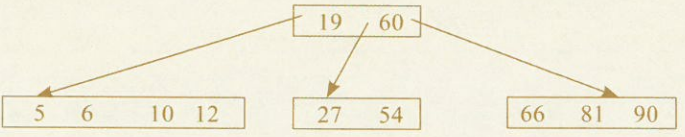
حذف 100 : مباشر



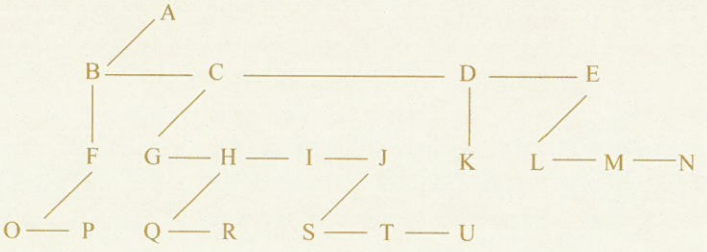
حذف 120 : دمج

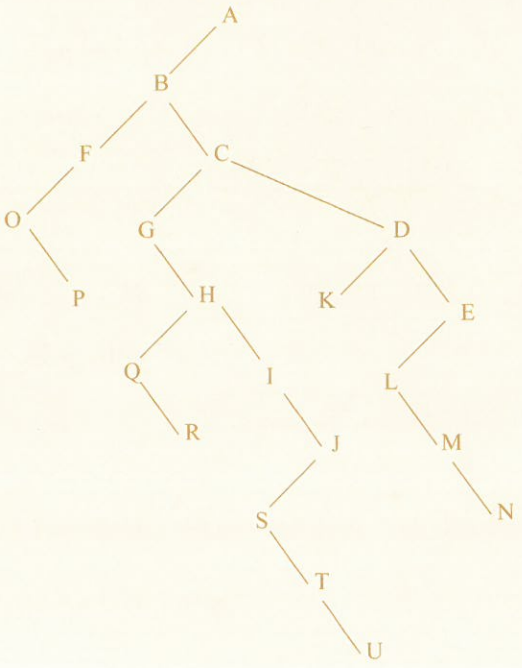


حذف 210 : مباشر



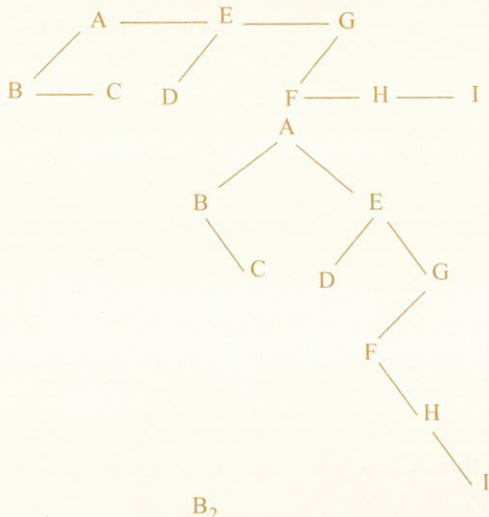
٥-٧ (أ)





B_1

(ب)



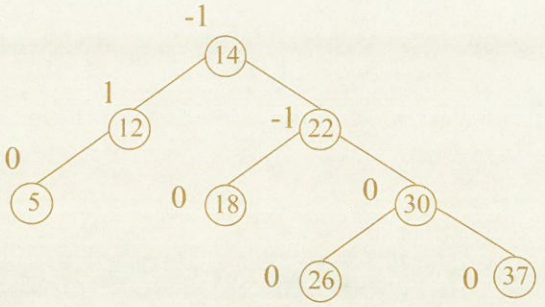
B_2

(ج) نتائج الاجتياز سابق الترتيب :

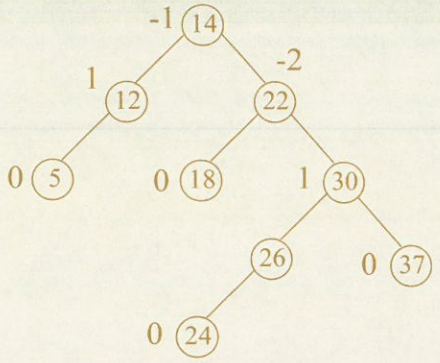
T_1 : A B F O P C G H Q R I J S T U D K E L M N ←
B_2 : A B F O P C G H Q R I J S T U D K E L M N ← =

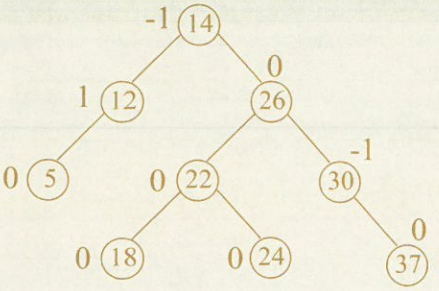
B_2 : A B C E D G F H I ←
F_1 : A B C E D G F H I ← =

٥-٨ (i)



(ii)





(iii) نتيجة الاجتياز الترتيبي :

5 12 14 18 22 24 26 30 37

٥-٩ (أ)

```
void inorder (node_ptr root)
{ // to apply inorder traversal to the tree root.
      if (root ! = NULL)
            {
                  inorder (root → left) ;
                  cout << (root → data) ;
```

```
                inorder (root → right) ;
            }
}
void sorting_ascending (node_ptr& tree_root, int n)
    {
        int i, x;
        tree_root = NULL ;
        couut << "enter the n integers"  << endl;
        for (i = l; i <=n; i + +)
            {
                cin >> x ;
                insert (tree_root, x)
            }
        inorder (tree_root)
    }
```

(ب) حيث أن لدينا عروة for (داخلها الإجراء insert) ، ثم الإجراء inorder ، لذلك فإن درجة التعقيد :

Complexity ≅ O(n*complexity (insert) + complexity (inorder))

والإجراء inorder يمر على جميع العناصر وعددها n (أي n خطوة) وحيث أنه من المعلوم أن عدد خطوات أى من عمليات البحث أو الإضافة فيث شجرة بحث ثنائية يعتمد على ارتفاع الشجرة h أي O(h) ، وعموما h ≤ n ، لذلك ففي أسوأ حالة (كي تكون درجة التعقيد أكبر ما يمكن) يكون h = n = 5

مثال : المجموعة المكونة من الخمسة عناصر المدخلة بالترتيب التالي :

1 5 7 10 14

تؤدي إلى شجرة بحث ثنائية ارتفاعها h = 5

دليل المصطلحات العربية والإنجليزية
Index

فيما يلي قائمة بالمصطلحات الإنجليزية مرتبة ترتيبا أبجديا ، والمصطلحات العربية المقابلة لها ، وكذلك فهرس لهذه المصطلحات.

append	٢٠٤	يُلحِق
applications	٣٤	تطبيقات
argument	٤٨	وسيط (فعلي)
arithmetic expression	١٧٧	تعبير حسابي
array	١٠٥	منظومة
array length	١٠٦	طول المنظومة
array of pointers	١٣٥	منظومة مؤشرات
array of records	١١٣	منظومة سجلات
array structure	١٥٠	بنية منظومة
assembly language	١٧٨	لغة التجميع
associated	٥٢	مرتبط
asymptotic complexity	١٤٨	درجة التعقيد التقاربية
asymptotic notation	٨٢	اصطلاح التقارب
atom	١٠٧	ذرة
average-case complexity	٧٤	درجة التعقيد المتوسطة
AVL tree	٣١٣	شجرة AVL
(balance factor (BF	٣١٤	معامل التوازن
balanced tree	٣١٣	شجرة متوازنة
base address	٥٢	عنوان الأساس ، العنوان الأساسي
best-case complexity	٧٤	درجة التعقيد في أحسن حالة
(BF (balance factor	٣١٤	معامل التوازن
binary	٣٢	ثنائي
binary operator	١٨٣	مؤثِّر ثنائي

circular queue	١٩٤	طابور دائري
class	٤٤	طبقة
clustered	١٢٤	مجمَّع ، مكدَّس
coding	٤٠	تشفير
coefficient	١٠٩	معامل
comparison	٧٥	مقارنة
compiler	٤٥	(البرنامج) المترجم
complete binary tree	٢٨١	شجرة ثنائية تامة / كاملة
complexity	٧٣	درجة التعقيد
complexity function	٩١	دالة (درجة) التعقيد
component selector	٤٧	منتقي المركبات
components	١٥٠	وحدات ، مركبات
composite	٣٦	مركَّب
computer words	١٤٩	كلمات الحاسوب
computing time	٩١	الوقت المستغرق لإجراء الحسابات
concatenation	٢٣٤	سلسلة ، تسلسل ، تعاقب ، إلحاق ، وصل
concrete	٣٦	راسخ
constant	٩١	ثابت
constant factor	١٤٨	معامل ثابت
construct	٣٤	ينشئ
constructor	٤٢	المنشئ
contiguous cells	١٠٧	خلايا متلاصقة
convert	١٧٨	يحوِّل

empty locations	٢٠٨	المواضع الخالية
encapsulation	٣٣	تغليف
entity	٤٠	وحدة ، عنصر
enumeration type	٥٧	نوع تعددي
equivalent	٨٣	مكافئ
equivalent trees	٢٧٧	أشجار متكافئة
erase	٢٤٥	يمحو
evaluation	٧٢، ١٧٨	تقييم ، إيجاد القيمة
execution time	٩٩	وقت / زمن التنفيذ
expected height	٣٣٤	الارتفاع المتوقع
expensive method	١٩٣	طريقة مكلفة
exponent	١٠٩	أس
exponential function	٩٢	دالة أسية
expression tree	٢٨٧	شجرة تعبير
extract	٤٧	يستخرج
Fast Transpose Algorithm	١٣٥	خوارزمية التدوير السريع
feature	٣٤	خاصية ، سِمَة
Fibonacci trees	٣٣٣	أشجار «فيبوناتشي»
field	٤٥	مجال
finite collection	٤٥	مجموع محدودة / منتهية
finite sequence	١٠٧	متتابعة محدودة
finite set	٢٧٣	مجموعة محدودة
floating-point	٥٢	النقطة العائمة
forest	٢٧٥	غابة

formal parameter	٤٩	وسيط شكلي
free position	١٣٩	موضع خالي
front	١٨٨	المقدمة
front element	٢٢١	العنصر الأمامي
full	١٢٣	ممتلئ
full binary tree	٢٨٠	شجرة ثنائية كثيفة / ممتلئة
function	٤٩	دالة
generation	١٧٨	توليد
graphs	٢٧٤	الرسوم البيانية ، المخططات البيانية
grow	٣٣٩	ينمو
guarantee	١٠٠	ضمان
hashing	٧١	بعثرة
head	١٨٨	الرأس
header node	٢٣٥	عنصر في الواجهة / المقدمة / البداية
height balanced tree	٣١٣	شجرة متوازنة ارتفاعا
height of a tree	٢٧٥	ارتفاع الشجرة
high level language	١٠٦	لغة عالية المستوى
high priority	١٨٣	أولوية عليا
homogeneous	٥٥	متجانس
immediate brother	٣٤٢	أخ مباشر
implementation	٣٢	تنفيذ
increasing order	٩١	ترتيب تصاعدي
index	٣٦	دليل ، مؤشر ، ترتيب
indexing	٧٩	إعطاء قيم ترتيبية

key field	١٠٩	المجال المفتاح
large enough	٨٢	كبير كبرا كافيا
lateral movement	٣٢٥	حركة عرضية
leaf	٢٧٤	ورقة (شجرة)
left subtree	٢٧٦	شجرة فرعية يسرى
(length (of a list	١٠٧	طول (قائمة)
level	٣٤	مستوى
linear algorithm	٧٨	خوارزمية خطية
linear function	٧٨	دالة خطية
linear list	١٠٧	قائمة خطية
linear system	١٢٣	نظام خطي
link	٢٥١	رباط ، يربط
linked list	٧٠	قائمة مترابطة
linked queue	٢٢٠	طابور مترابط
linked representation	٢١١	تمثيل مترابط
linked stack	٢١٦	رصة مترابطة
list	٣٢	قائمة
LL (left left) rotation	٣١٨	تدوير LL
local changes	٣٢٧	تغيرات محلية
location	٥٢	موقع ، موضع
logical list	٢٦٣	قائمة منطقية
low priority	١٨٤	أولوية سفلى
lower bound	٨٧	حد سفلي / أسفل
LR (left right) double rotation	٣٢٣	تدوير مزدوج LR

multilist structure	٢٤٤	بنية القوائم المتعددة
multiple queues	١٩٦	الطوابير المتعددة
multiple stacks	١٩٦	الرصات المتعددة
multiway	٣٣٥	متعدد الطرق
mutator	٤٢	المحوِّل
n-dimensional array	١٥٠	منظومة نونية البعد
non-recursive traversal	٢٨٦	اجتياز غير ارتدادي
nonzero term	١١٢	حد غير صفري
notation	٨٢	اصطلاح
null value	٢٨٣	قيمة متلاشية
object	٤٢	شئ ، هدف ، كائن
observer	٤٣	المراقِب
offset	٥٣	تشعُّب ، انحراف ، بُعْد ، إزاحة
one-dimensional array	١٠٥	منظومة أحادية البعد
one's complement	٣٢	مكمِّل-١
operand	٣٣	معامَل ، مؤثَّر عليه ، كمية معامَلة
operation	٣٤	عملية
operator	٣٤	معامِل ، مؤثِّر
order	٦٢، ٧٩	رتبة ، ترتيب
order of magnitude	٨٢	حدود القيم وترتيبها
ordered linked list	٢٦٢	قائمة مترابطة مرتبة
ordered list	٦٩، ١٠٧	قائمة مرتبة
ordered pair	١١٢	زوج مرتب
ordered triple	١٢٥	ثلاثية مرتبة

reference parameter	٥٩	وسيط إسناد
regular tree	٢٧٧	شجرة عادية
relational operator	٣٥	مؤثر علاقي
relations	٢٧٧	علاقات
representation	٣٥	تمثيل
representation of arrays	١٤٩	تمثيل المنظومات
retrieve	٣٨	يسترجع ، يستعيد
reverse	١٧١	يعكس
right subtree	٢٧٦	شجرة فرعية يمنى
RL (right left) double rotation	٣٢٧	تدوير مزدوج RL
root	٢٧٣	الجذر
rotate	٣١٩	يُدَوِّر
row major representation	١٥٢	تمثيل رئيسي الصف
RPN (Reverse Polish Notation)	١٧٨	الاصطلاح البولندي المعكوس
RR (right right) rotation	٣٢١	تدوير RR
run time	٥٠	وقت التشغيل
scalar product	٧٧	الضرب القياسي
scanning	١٠٧	اجتياز ، مسح ، مرور على
search time	٣١٢	زمن البحث
selection sort	١٠١	الترتيب بالاختيار
semantics	٤٦	معنى
sentinel node	٢٣٥	عنصر حارس
sequence	١٠٥	متتابعة / متسلسلة
sequential access	١٠٦	الوصول التتابعي

trace	١٤٣	يتتبع
transformation	٢٩١	تحويل
transformer	٤٣	المحوِّل
transposing a matrix	١٢٣	تدوير مصفوفة
traversal	١٠٧	اجتياز
tree	٢٧٣	شجرة
tridiagonal sparse matrix	١٢٤	مصفوفة متناثرة قطرية ثلاثية
two-dimensional array	٦٤	منظومة ثنائية البعد
two's complement	٣٢	مكمِّل-٢
type	١٠٥	نوع
type definition	١١٠	تعريف نوع
unary	١٨٤	أحادي
unsigned direct binary	٣٢	(الصيغة) الثنائية المباشرة دون إشارة
unstructured	٤٤	غير مبني
updating	٤٢	تحديث ، تعديل ، تغيير ، تجديد
upper bound	٨٤	حد علوي
user	٦٨	المستخدِم
user interface	٧١	وصلة بينية للمستخدم
valid	١٨٣	صحيح
valid node	٣٤١	عنصر صحيح
value	٣٥	قيمة
value parameter	٤٨	وسيط ذو قيمة
vertical movement	٣٢٥	حركة رأسية

كتب للمؤلف
في الرياضيات وعلم الحاسوب

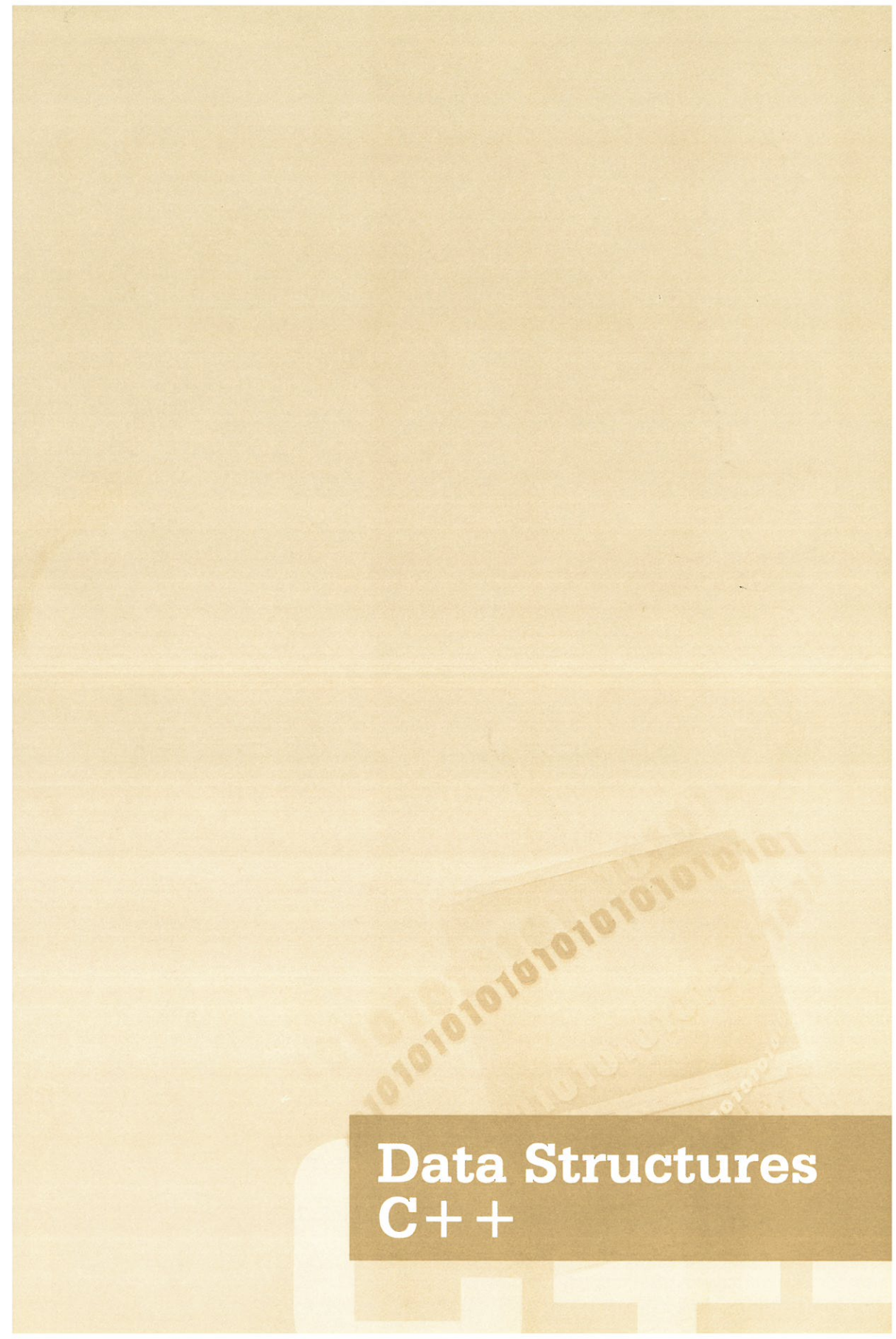
١) برمجة الحاسب بلغة الفورتران ، ط ٤ ، دار القلم . . الكويت ١٩٩٢ .
٢) مقدمة في نظرية المعلومات ، ط ٢ ، دار القلم . . الكويت ١٩٩٣ .
٣) الشبكات الرقمية ، دار القلم . . الكويت ١٩٨٦ .
٤) التحليل العددي ، دار القلم . . الكويت ١٩٨٨ .
٥) الجبر الخطي ، ط ٢ ، دار القلم . . الكويت ١٩٩٥ .
٦) برمجة الحاسب بلغة الباسكال ، ط ٢ ، دار القلم . . الكويت ١٩٨٩ .
٧) البرمجة المتقدمة بلغة الباسكال ، مع د . حمزة رشوان ، دار القلم . . الكويت ، ١٩٩٤ .
٨) الدوائر المتكاملة الرقمية (ترجمة) ، دار القلم . . الكويت ١٩٩٣ .
٩) الخوارزميات والبرمجة بلغة الباسكال ، دار القلم . . الكويت ١٩٩٧ .
١٠) بنى المعطيات ، مع د . حمزة سيد رشوان – دار القلم . . الكويت ١٩٩٨ .
١١) الحلول العددية للمعادلات التفاضلية العادية ، دار القلم . . الكويت ٢٠٠٠ .
١٢) الحلول العددية للمعادلات التفاضلية الجزئية ، دار القلم . . الكويت ٢٠٠١ .
١٣) برمجة الحاسب بلغة C++ ، دار القلم . . الكويت ٢٠٠٢ .
١٤) البرمجة المتقدمة بلغة C++ ، دار القلم . . الكويت ٢٠٠٣ .
١٥) الرياضيات المتقطعة في علم الحاسوب ، مكتبة الفلاح . . الكويت ٢٠٠٥ .
١٦) هياكل البيانات بلغة C++ ، مع د . حمزة رشوان ، مكتبة الفلاح . . الكويت ٢٠٠٦ .
١٧) برمجة الحاسوب بلغة C ، مكتبة الفلاح . . الكويت (تحت الطبع) .
١٨) البرمجة المتقدمة بلغة C ، مكتبة الفلاح . . الكويت (تحت الطبع) .

Data Structures in C++

Dr. Hamza Sayed Rashwan
Dept. of Mathematics
University of Al- Azhar

Dr. Abu- Bakr Ahmad El- Sayed
Dept. of Math. and Computer Sc.
University of Kuwait

مكتبة الفلاح
للنشر والتوزيع


Data Structures
C++

هياكل البيانات بلغة C++

مكتبة الفلاح
للنشر والتوزيع

دولة الكويت
حولي - شارع بيروت - عمارة الأطباء
هاتف: 1985 264 فاكس: 7784 264 965+
ص.ب: 4848 الصفاة-الرمز البريدي 13049

دولة الإمارات العربية المتحدة
العين:- ص.ب 16431 هاتف: 7662189 فاكس: 7657901 3 971+
دبي:- ص.ب: 20438 هاتف: 2630618 فاكس: 2630628 4 971+

جمهورية مصر العربية
37 شارع النصر - امتداد رمسيس2 - مقابل وزارة المالية ومصلحة الجمارك
مدينة نصر - القاهرة - تلفاكس 8143 262 202+
www.alfalahbookshop.com

المطابع المركزية
عمان - الأردن