جَامعة المَنارة
MANARA UNIVERSITY

قسم المعلوماتية - كلية الهندسة

بنى معطيات 1

Data Structure 1

ا. د. علي عمران سليمان

محاضرات الأسبوع السادس
اللوائح 2
lists 2

الفصل الأول 2025-2026

محتوى الفصل

اللوائح 2	Lists 2
1- مقدمة	
2- بعض الأشكال الأخرى للو ائح أحادية الارتباط.	
3- التحقيق المتر ابط لكثيرات الحدود.	
4- hash tables	
5- اللو ائح المتر ابطة المضاعفة والصنف القياسي list في C++.	
6- لو ائح أخرى متعددة الارتباط.	

References

- Deitel & Deitel, Java How to Program, Pearson; 10th Ed(2015)

- د.علي سليمان، بنى معطيات بلغة JAVA، بنى معطيات بلغة C++، بنى معطيات بلغة Pascal جامعة تشرين 2014، 2007، 1998

2- التحقيق المترابط لكثيرات الحدود 1

linked implementation of sparse polynomials1

يملك كثير الحدود P(x) بمتحول واحد x الشكل التالي:

$$P(x)=a_0+a_1x+a_2x^2+a_3x^3+\ldots\ldots+a_nx^n$$

حيث $a_0 , a_1 , a_2 ,\ldots,a_n$ هي معاملات coefficients كثير الحدود، درجة degree كثير الحدود P(x) هي القوة (الأس) الأعلى لـ x التي تظهر في كثير الحدود بمعامل غير الصفري، على سبيل المثال: $P(x)=5+7x-8x^3+4x^5$

هو من الدرجة 5 ومعاملاته هي 5,7,0,-8,0,4. أما كثير الحدود الثابت Q(x)=3.78 فهو من الدرجة 0 وكذلك كثير الحدود الصفري هو من الدرجة صفر. يمكن النظر إلى كثير الحدود على أنه لائحة من المعاملات: $(a_0 , a_1 , a_2 ,\ldots,a_n)$

ويمكن تمثيله بواسطة تحقيقات اللوائح التي تعرفنا عليها. على سبيل المثال كثير الحدود $P(x)=5+7x-8x^3+4x^5$ يمكن كتابته أيضاً كما يلي: $P(x)=5+7x+0x^2-8x^3+0x^4+4x^5+0x^6+0x^7+0x^8+0x^9+0x^{10}$

والذي يمكن تعريفه بلائحة المعاملات: (5,7,0,-8,0,4,0,0,0,0,0)

ويمكن أن تخزن في مصفوفة P حجمها 11 كما يلي:

I	0	1	2	3	4	5	6	7	8	9	10
P[i]	5	7	0	-8	0	4	0	0	0	0	0

2- التحقيق المترابط لكثيرات الحدود 1

linked implementation of sparse polynomials1

إذا كان الفرق بين القوة العليا والقوة الدنيا في كثير الحدود صغيراً وكانت القوى أصغر من السقف الأعلى المفروض وفق حجم المصفوفة وليس فيه معاملات صفرية كثيرة فإن التحقيق السابق قد يكون ناجحاً تماماً، أما من أجل كثيرات الحدود ذات الاختلافات الكبيرة تكون الحدود الصفرية كثيرة فإن التحقيق باعتماد المصفوفات ليس فعالاً، فمثلاً لتخزين كثير الحدود:

$$Q(x)=5+x^{99}$$

أو بشكل مكافئ:

$$Q(x)=5+0x+0x^2+0x^3+.....+0x^{98}+1x^{99}$$

والذي يتطلب مصفوفة بعنصرين غير صفريين، و 98 عنصر صفري فإنه يمكن الحد من الهدر الواضح للذاكرة الناتج عن تخزين كل المعاملات الصفرية وذلك بتخزين المعاملات غير الصفرية فقط.في مثل هذا التحقيق من الواضح أن تمثيله كلائحة عناصرها عبارة عن أزواج معاملات،

$$P(x)=5+7x-8x^3+4x^5\leftrightarrow((5,0),(7,1),(-8,3),(4,5))$$

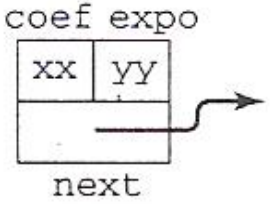
$$Q(x)=5+x^{99}\leftrightarrow((5,0),(1,99))$$

نلاحظ أن الأزواج مرتبة وفق ترتيب القوى المتزايدة.

هذه اللوائح يمكن تحقيقها بواسطة مصفوفات السجلات أو اصناف، كل منها يحتوي حقل المعامل وحقل القوة (الأس)، والأفضل بدل المصفوفة بلائحة لإعطاء مرونه في تحد حجم اللائحة وتلافي ضياعاً ملحوظاً في الذاكرة في مثل هئه التطبيقات.

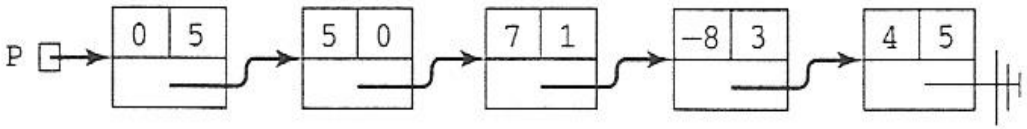
2- التحقيق المترابط لكثيرات الحدود 1

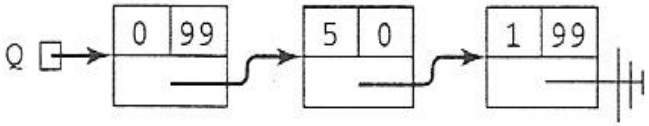
linked implementation of sparse polynomials1



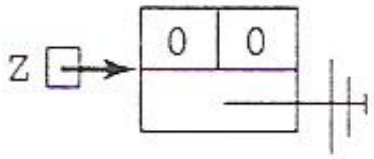
في هذه التطبيقات، فإن التحقيق باستخدام اللائحة المترابطة مناسب أكثر، كل عقدة تأخذ الشكل:

وفيها تخزن الأجزاء الثلاثة coef,expo,next معاملات غير الصفرية، الأس المقابل، ومؤشر إلى العقدة التالية الممثلة للحد التالي.
على سبيل المثال كثيري الحدود السابقين P(x) و Q(x) يمكن تمثيلهما باللائحتين المترابطتين اللتين تملكان عقدة راسية تخزن درجة كثير الحدود في الحقل expo:





أما كثير الحدود الصفري يمكن تمثيله ببساطة بعقدة رأس كما يلي:



نستطيع الآن البدء بتعريف قالب الصنف Polynomial لمثل
كثيرات الحدود المترابطة هذه:

linked implementation of sparse polynomials1

```
#ifndef POLYNOMIAL
#define POLYNOMIAL
template <typename CoefType>
class Polynomial          {/**** Node structure ****/
private: class PolyNode
        {       public:         CoefType  coef;    int    expo;        PolyNode *next;
                PolyNode(CoefType co=0,int ex=0,PolyNode *ptr=0)
                        {       coef=co;        expo=ex;        next=ptr;       }
   };
typedef PolyNode  *PolyPointer;n   /**** function members ****/
public:         .....
/**** data members ****/
private:        PolyPointer  myFirst;       int  myDegree;     ........
};
#endif
```

2- التحقيق المترابط لكثيرات الحدود 1

linked implementation of sparse polynomials1

هنا نستخدم الشكل القياسي لعقد مؤلفة من جزء data وجزء next بجعل نمط الجزء data عبارة عن سجل مؤلف من جزء coef وجزء expo لكن ذلك لا يحقق فوائد حقيقية إذ قد يتطلب اختيار مضاعف للعضو للوصول إلى العناصر أي P->data.coef بدلاً من الكتابة ببساطة P->coef.

تتم معالجة مثل هذه اللوائح المترابطة من كثيرات الحدود، سندرس عملية جمع كثيرات الحدود. مثلاً لجمع كثيري الحدود A(x) و B(x) التاليين:

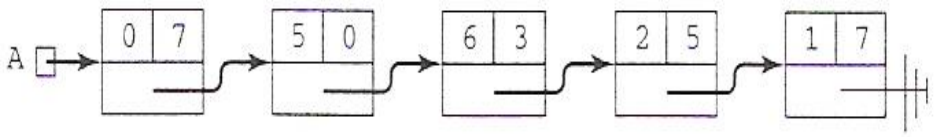
$A(x)=5+6x^3+2x^5+x^7$

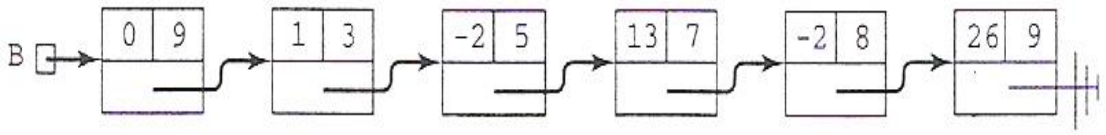
$B(x)=x^3-2x^5+13x^7-2x^8+26x^9$

إن عملية الجمع تتم بجمع معاملات الحدود المتشابهة في درجة x وبالتالي فإن ناتج جمع كثيري الحدود A(x) و B(x) هو:

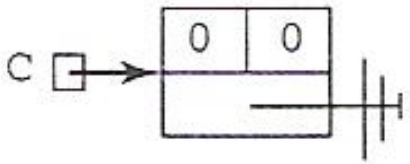
$C(x)=A(x)+B(x)=5+7x^3+14x^7-2x^8+26x^9$

يمكن توضيح التمثيل المترابط لكثيري الحدود مع عقدة رأس.





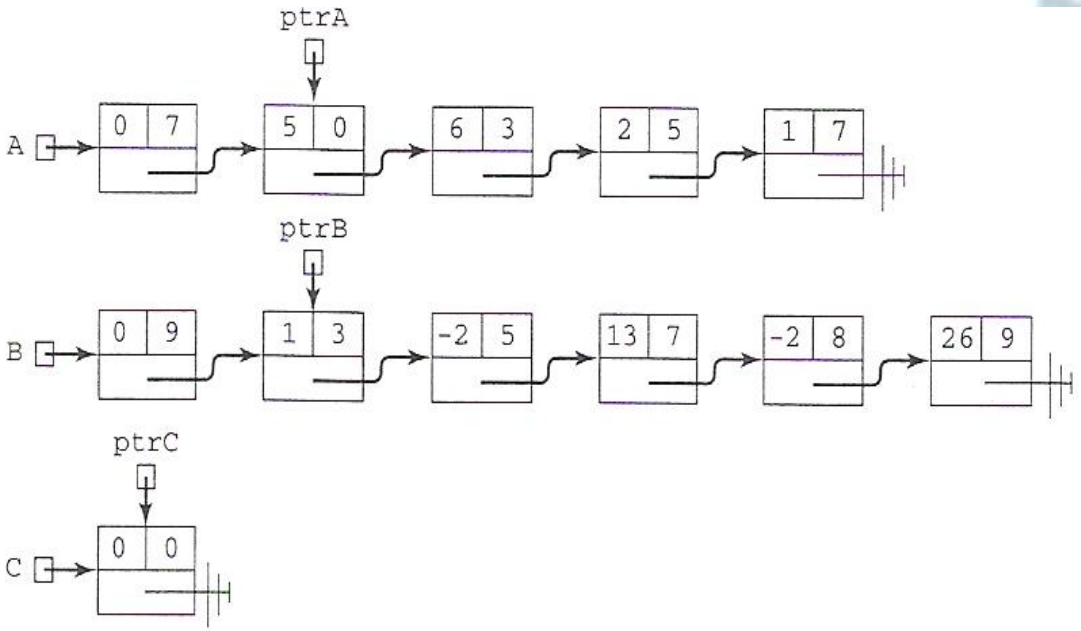
حيث يوضع في جزء المعطيات الأول 0 والثاني أكبر اس في التعبير،



2- التحقيق المترابط لكثيرات الحدود 1

linked implementation of sparse polynomials1

المؤشرات المساعدة ptrA,ptrB,ptrC ستتجول عبر اللوائح A,B,C على الترتيب، سيشير المؤشران ptrA,ptrB إلى العقد الحالية التي تتم معالجتها أما المؤشر ptrC فسيشير إلى آخر عقدة مرتبطة بـ C. وعند الجمع توجد ثلاث حالات:



1- نضع مؤشر إلى بداية كل تعبير ptrA, ptrB, ptrC وندخل حلقة تتكرر حتى ينتهي أحد التعبيرين يتم مقارنة الحدين المؤشر عليهما من قبل ptrA, ptrB ونجد حالتين:

1-1- وجود إختلاف الاس للعقدتين المقارنتين بالمؤشرين ptrA و ptrB فإنه سيتم أخذ عقدة جديدة ونسخ قيم العقدة التي تحوي الاس الأصغر والمعامل المقابل وربطها باللائحة C (مع اختبار أن تكون فارغة أو غير فارغة) ويتم تقديم المؤشر ptrC ومؤشر اللائحة التي تم منها الربط.

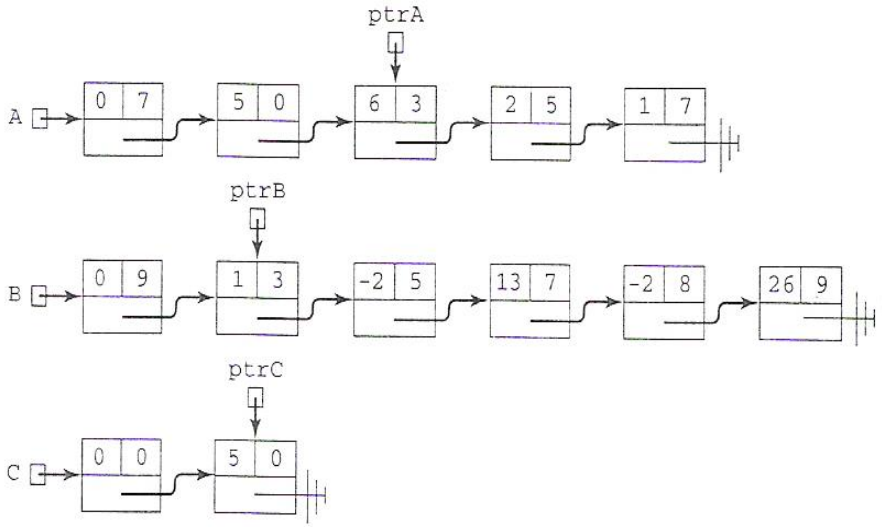
1-2- وجود الاس المتساوي في التعبيرين هنا حالتين :

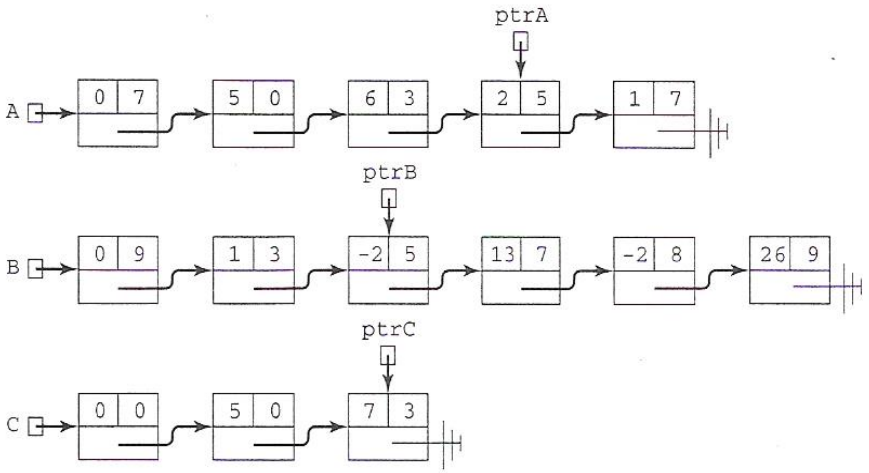
1-2-1-ناتج جمع المعاملات ليس صفر عندها سيتم أخذ عقدة جديدة ونسخ ناتج الجمع والاس إليها ووضعها في C (مع اختبار أن تكون فارغة أو غير فارغة) ويتم إزاحة المؤشرات الثلاث ptrA,ptrB,ptrC

1-2-2- ناتج جمع المعاملات صفراً فإن ptrA وptrB يتم تقديمهما ولا تضاف أي عقدة إلى C:

2- التحقيق المترابط لكثيرات الحدود 1

linked implementation of sparse polynomials1

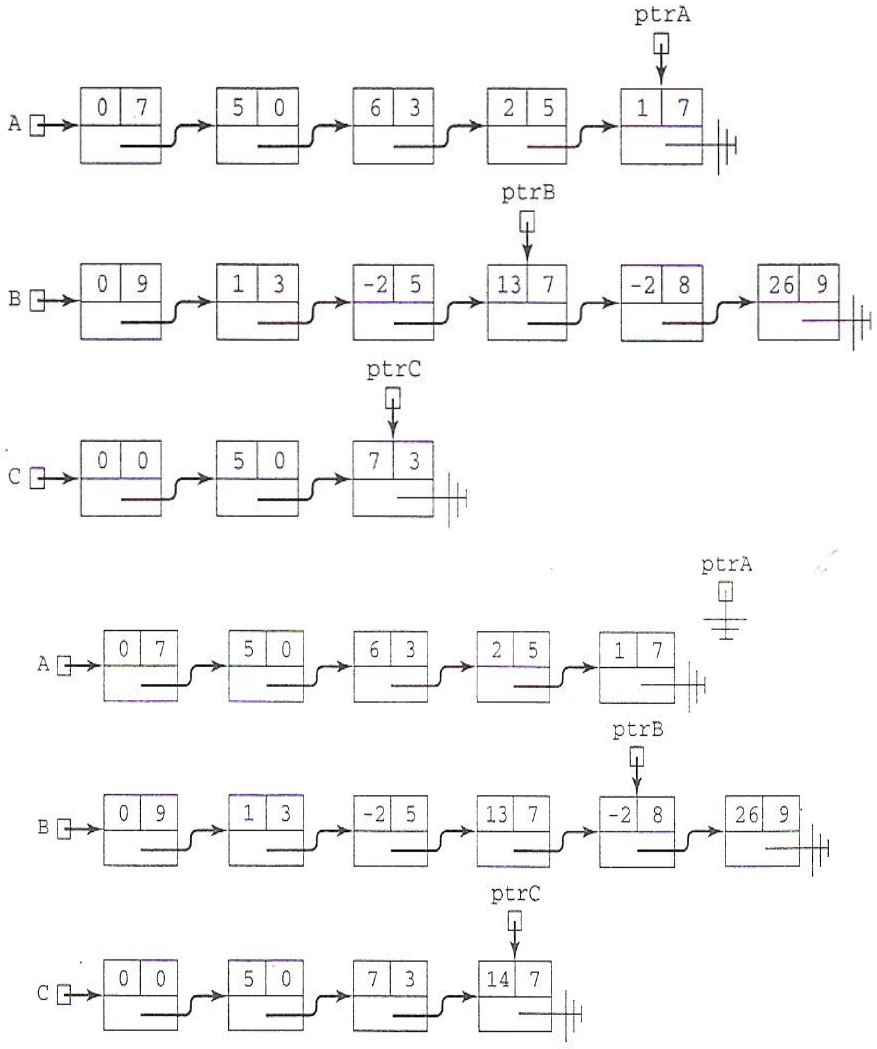




2- حلقة تتكرر حتى ينتهي التعبير تقوم بنسخ حدود التعبير غير المنتهي إلى اللائحة C (مع اختبار أن تكون فارغة أو غير فارغة) ويتم تقديم المؤشر ptrC ومؤشر اللائحة المنسوخ منها ويتم النسخ حداً حداً.

2- التحقيق المترابط لكثيرات الحدود 1

linked implementation of sparse polynomials1

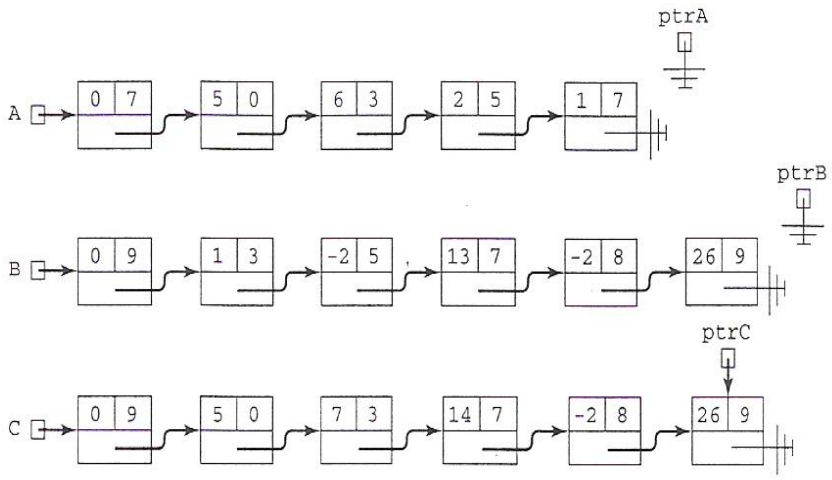


ونتابع بهذه الطريقة لحين الوصول إلى نهاية A أو B، أي إلى أن يشير أحد المؤشرين ptrA أو ptrB إلى العنوان الصفري null وقد يكون كليهما:

إذا تم الوصول إلى نهاية إحدى اللائحتين ولم يتم الوصول إلى نهاية الأخرى، سنقوم بأخذ عقدة جديدة ونسخ العقد المتبقية فيها وربطها إلى C (واخدة تلو الأخرى). بعدئذ ننهي بناء اللائحة المترابطة C التي تمثل المجموع A(x)+B(x) بجعل الجزء Next من العقدة الأخيرة لـ C يشير إلى العنوان الصفري null و الحقل expo في العقدة الرأسية لـ C يأخذ قوة آخر عقدة:

linked implementation of sparse polynomials1

2- التحقيق المترابط لكثيرات الحدود 1



تبين الشيفرة التالية تحقيق هذه الخوارزمية لجمع كثيرات الحدود ويمكن إضافتها إلى قالب الصنف Polynomial كتابع صديق:

```
void merge(linklist firsta,linklist firstb, linklist firstc )
{ Node *ptra=firsta, *ptrb=firstb, *ptrc=firstc; int degree=0;
 while ( ptra !=NULL && ptrb !=NULL)
 {  if( ptra-> expo < ptrb-> expo )  { Node *INN= new Node;  INN-> expo = ptra-> expo; ;  INN-> Coef = ptra-> Coef; ptra=ptra->next;
        if ( ptrc==NULL)    { INN->next = firstc; firstc=INN; ptrc=INN;}    //insert in empty List C
         else { ptrc->next = INN;  ptrc=INN; }                              /* insert in end List C from A */  }
if( ptrb-> expo < ptra-> expo )  { Node *INN= new Node;  INN-> expo = ptrb-> expo; ;  INN-> Coef = ptrb-> Coef;  ptrb=ptrb->next;
```

```
if ( ptrc==NULL)    { INN->next = firstc; firstc=INN; ptrc=INN; }          //insert in empty List C
        else { ptrc->next = INN;  ptrc=INN;  }                            /* insert in end List C  from B*/      }
   else      degree= ptra-> Coef+ ptrb-> Coef ; if (degree==0) {ptra=ptra->next; ptrb=ptrb->next;}
    else {Node *INN= new Node;  INN-> Coef = degree;  INN-> expo = ptra-> expo ; ptra=ptra->next; ptrb=ptrb->next;}
          if ( ptrc==NULL)    { INN->next = firstc; firstc=INN; ptrc=INN; }      //insert in empty List C
       else { ptrc->next = INN;  ptrc=INN; }              /* insert in end List C  from A or B*/      }

while ( ptra !=NULL)        {Node *INN= new Node;  INN-> expo = ptra-> expo;   INN-> Coef = ptra-> Coef; ptra=ptra->next;
       if ( ptrc==NULL)    { INN->next = firstc; firstc=INN; ptrc=INN;  }      //insert in empty List C
        else { ptrc->next = INN;  ptrc=INN;}                                  /* insert in end List C from A */       }

while ( ptrb !=NULL)        {Node *INN= new Node;  INN-> expo = ptrb-> expo;   INN-> Coef = ptrb-> Coef; ptrb=ptrb->next;
       if ( ptrc==NULL)    { INN->next = firstc; firstc=INN; ptrc=INN;  }      //insert in empty List C
        else { ptrc->next = INN;  ptrc=INN;}                                  /* insert in end List C  from B*/   }
```

إن الشيفرة البرمجية لعملية جمع كثيري حدود مترابطين أكثر تعقيداً و أقل قابلية للفهم من الشيفرة المقابلة في حال تحقيق كثير الحدود باستخدام المصفوفات والتي تكتب بالشكل:

```
int MaxDegree=A.myDegree>=B.myDegree?A.myDegree:B.myDegree;

for (int i=0;i<MaxDegree;i++)
        C[i]=A[i]+B[i];
```

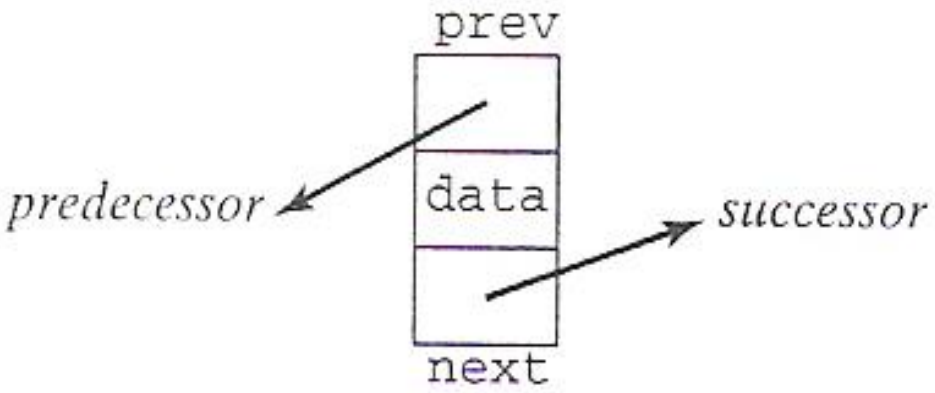
بالمثل فإن التوابع الخاصة بإجراء العمليات الأخرى على كثيرات الحدود، مثل إيجاد قيمة كثير الحدود من أجل قيمة x، وهكذا، أكثر تعقيداً من التوابع المقابلة لها في حال التحقيق باستخدام المصفوفات. ولكن بالمقابل في التطبيقات التي تكون فيها درجة كثيرات الحدود كبيرة فإن توفير الذاكرة سيكون أكثر قيمة من سهولة الخوارزميات.

4- اللوائح المترابطة المضاعفة والصنف القياسي list في C++

4 -doubly-linked lists and the standard C++ list

إن جميع اللوائح التي قمنا بدراستها حتى الآن أحادية الاتجاه unidirectional، وهذا يعني أنه من الممكن الانتقال ببساطة من العقدة إلى العقدة التالية لها. تتطلب بعض المسائل تحديد العنصر السابق بنفس الطريقة التي نحدد فيها العنصر اللاحق، وبالتالي فإن القدرة على التحرك ثنائي الاتجاه bidirectional ضرورية لأن إيجاد العنصر السابق في لائحة مترابطة أحادية غير مجدي على الإطلاق لأنه يتطلب البحث من بداية اللائحة.

نتعرف في هذه الفقرة على كيفية بناء اللوائح ثنائية الاتجاه ومعالجتها، ثم نتعرف على كيفية استخدام الصنف الحاوي القياسي list في لغة C++ ونقوم باستخدامها في مسألة إجراء العمليات الحسابية على الأعداد الصحيحة الضخمة



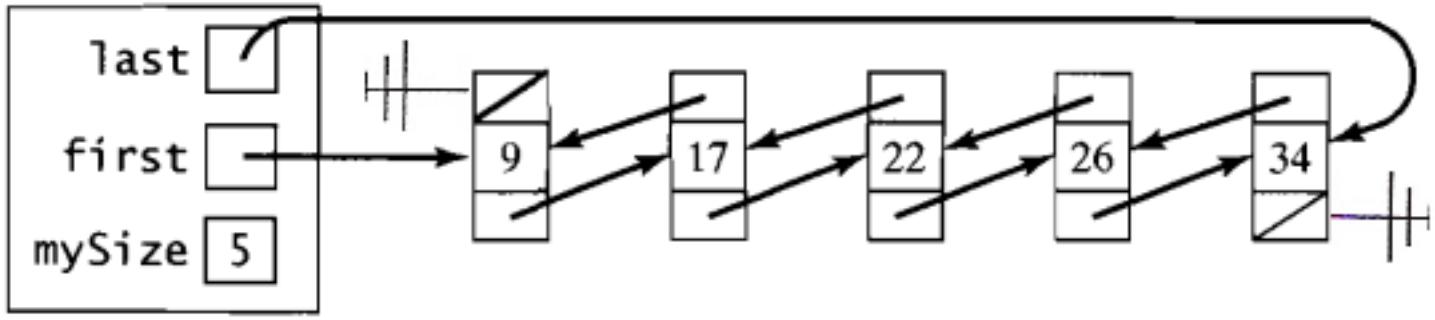
اللوائح المترابطة المضاعفة doubly-linked list:

يمكن بناء اللوائح ثنائية الاتجاه ببساطة باستخدام عقد تحتوي، بالإضافة إلى جزء البيانات، على رابطين رابط أمامي next يشير إلى التالي للعقدة ورابط خلفي prev يشير إلى السابق لها:

4 -doubly-linked lists and the standard C++ list

4- اللوائح المترابطة المضاعفة والصنف القياسي list في C++

اللائحة المترابطة التي تبنى من مثل هذه العقد تدعى اللائحة المترابطة المضاعفة doubly-linked أو اللائحة المترابطة المتناظرة symmetrically-linked. لتسهيل العبور الأمامي والخلفي يستخدم مؤشر first يتيح الوصول إلى العقدة الأولى ومؤشر آخر last يتيح الوصول إلى العقدة الأخيرة، على سبيل المثال، اللائحة المترابطة المضاعفة من الأعداد الصحيحة 9,17,22,26,34 يمكن تمثيلها بالشكل التالي:

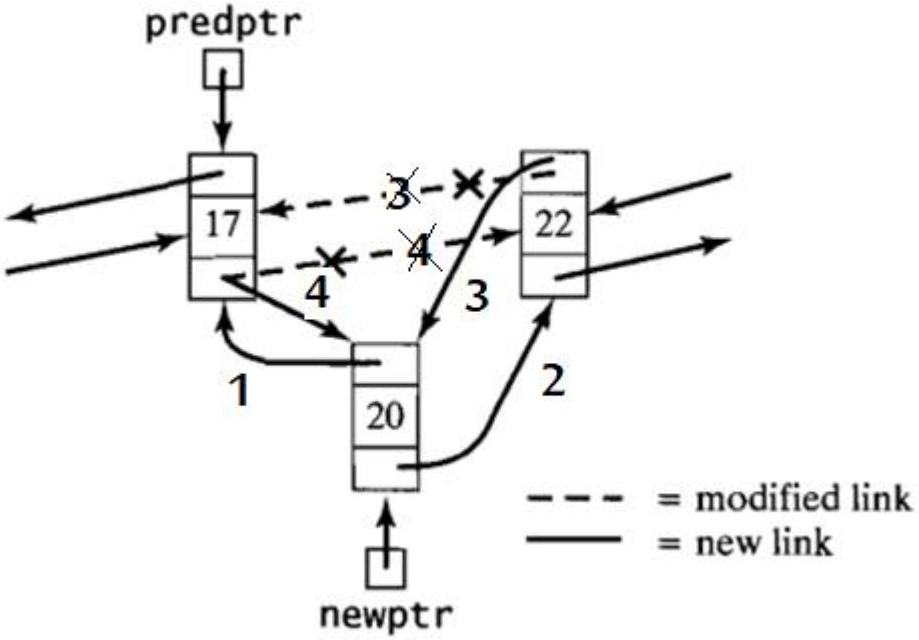


وكما في حال اللائحة المترابطة الأحادية، فإن استخدام عقدة رأسية في اللوائح المترابطة المضاعفة يخلصنا من بعض الحالات الخاصة (مثل اللائحة الفارغة و العقدة الأولى) كما أن جعل اللائحة حلقية يتيح إمكانية الوصول بسهولة إلى أي من نهايتي اللائحة. وكما سنرى لاحقاً في هذه الفقرة فإن هذه البنية مستخدمة في النمط القياسي في لغة C++ المسمى list.

4- اللوائح المترابطة المضاعفة والصنف القياسي list في C++

4 -doubly-linked lists and the standard C++ list

خوارزميات العمليات الأساسية للائحة شبيهة بتلك الخاصة بالحالة أحادية الارتباط، الفارق الأساسي هو الحاجة لبعض الروابط الإضافية، على سبيل المثال، عملية حشر عقدة في لائحة مترابطة مضاعفة يتضمن أولاً جعل رابطها الأمامي والخلفي يشيران إلى العقدة السابقة لها والعقدة اللاحقة، على التوالي، ومن ثم إعادة تعيين الرابط الأمامي للعقدة السابقة لها والرابط الخلفي للعقدة اللاحقة لها يشيران إلى العقدة الجديدة:



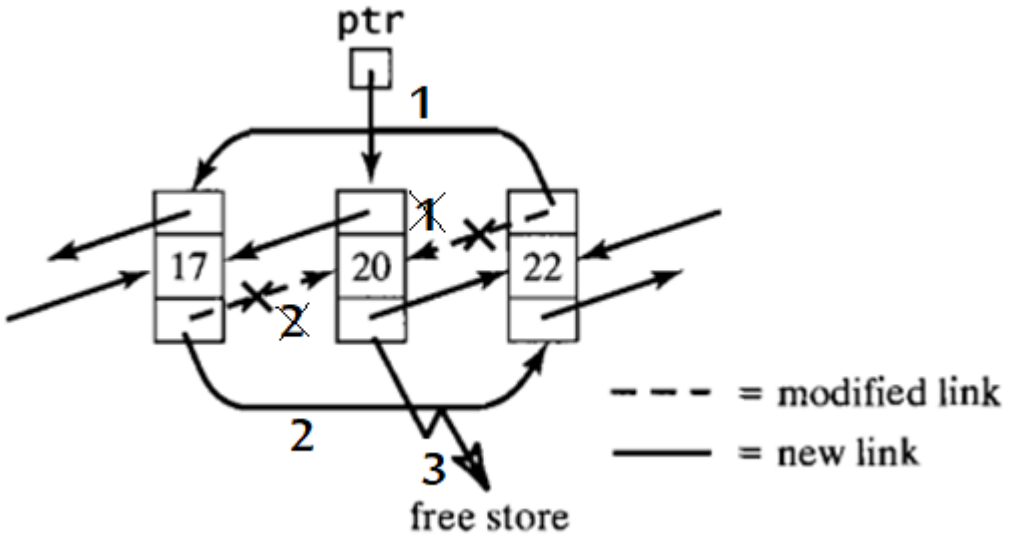
تقوم التعليمات التالية بإنجاز عملية الحشر هذه:

```
newptr->prev=predptr;          // 1
newptr->next=predptr->next;    // 2
predptr->next->prev =newptr;   // 3
predptr->next=newptr;          // 4
```

من المهم جداً أن يتم هذا بالترتيب الصحيح.

4 -doubly-linked lists and the standard C++ list

4- اللوائح المترابطة المضاعفة والصنف القياسي list في C++



يمكن حذف عقدة ببساطة بإعادة تعيين الرابط الأمامي للعقدة السابقة لها والرابط الخلفي للعقدة التالية لها لتجاوز العقدة:

يمكن تحقيق ذلك بثلاث تعليمات:

```
ptr->next->prev=ptr->prev;          // 1
ptr->prev->next=ptr->next;          // 2
delete ptr;                         // 3
```

يتم إجراء عملية التجول بنفس الطريقة كما في اللائحة أحادية الارتباط. التجول الأمامي يتبع الرابط next والتجول الخلفي يتبع الرابط prev. وكذلك التوابع البانية، الهادمة، البانية الناسخة، الإسناد وغيرها من العمليات الأخرى هي نسخ معدلة من تلك الخاصة باللوائح أحادية الارتباط.

5 - لوائح أخرى متعددة الارتباط

other multiply-linked lists

رأينا أن اللوائح المترابطة المضاعفة هي عبارة عن بنى معطيات مفيدة في التطبيقات التي تتطلب التنقل في كلا الاتجاهين، في هذه الفقرة نتعرف على مجموعة من الأشكال الأخرى من معالجة اللوائح وهي تلك التي تحتوي عقد اللائحة المترابطة على أكثر من رابط ، إن هذه الأشكال مدروسة هنا بشكل مختصر ويمكن للراغبين بالتوسع فيها العودة إلى مراجع اللغة.

اللوائح متعددة الترتيب multiply-ordered lists:

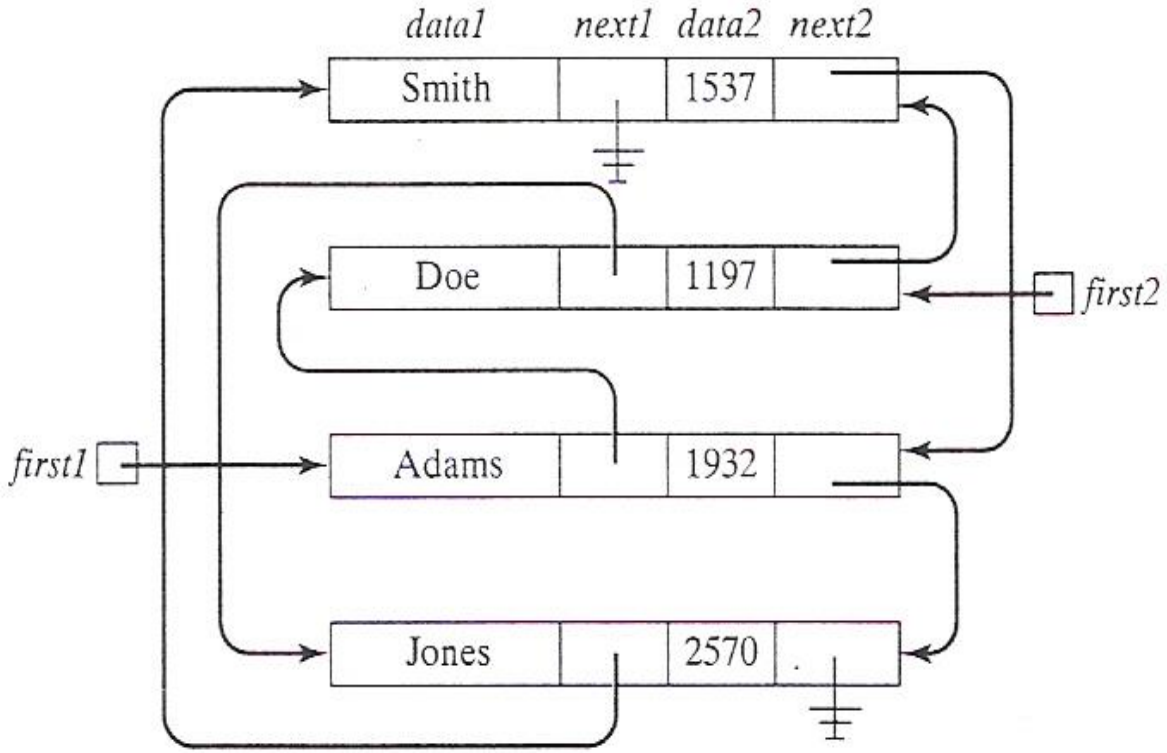
رأينا في الفصل السابق اللوائح المرتبة والتي رتبت فيها العقد بحيث أن عناصر البيانات (أو القيم في بعض الحقول المفتاحية لعناصر البيانات) مخزنة في هذه العقد بترتيب تصاعدي، في حين في بعض التطبيقات، يكون من الضروري الإبقاء على اعلى مجموعة مرتبة بطريقتين مختلفتين أو أكثر، على سبيل المثال، قد نرغب بالحصول على مجموعة من سجلات الطلاب مرتبة بحسب كل من الاسم والرقم الجامعي.

إن إحدى الطرق لتحقيق هذا الترتيب المتعدد هو المحافظة على لوائح مترابطة مرتبة منفصلة، واحدة من أجل كل معيار ترتيب مرغوب. لكن هذه الطريقة غير فعالة وخاصة من أجل السجلات الضخمة، لأن هذا يتطلب على نسخ متعددة من كل سجل.

يمكن استخدام أسلوب أفضل وذلك باستخدام لائحة واحدة تستخدم فيها عدة روابط لربط العقد مع بعضها بتراتيب مختلفة، على سبيل المثال، لتخزين مجموعة من السجلات تحوي أسماء الطلاب وأرقامهم الجامعية حيث الأسماء مرتبة ترتيباً أبجدياً والأرقام الجامعية مرتبة ترتيباً تصاعدياً، يمكن استخدام اللائحة المترابطة المتعددة التالية باستخدام رابطين في كل عقدة:

other multiply-linked lists

5 - لوائح أخرى متعددة الارتباط



إذا تم التجول وعرض حقول البيانات باستخدام المؤشر first1 للإشارة إلى العقدة الأولى و تتبع المؤشرات في الحقل next1 فإن الأسماء ستعطى بترتيب أبجدي:

أما إذا تم التجول باستخدام المؤشر first2 للإشارة إلى العقدة الأولى وتتبع المؤشرات في الحقل next2 يعطي الأرقام الجامعية مرتبة تصاعدياً:

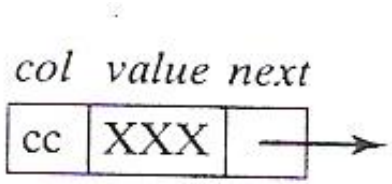
next1			next2	
Adams	1932		Doe	1197
Doe	1197		Smith	1537
Jones	2570		Adams	1932
Smith	1537		Jones	2570

مصفوفات متفرقة sparse matrices:

المصفوفة m×n هي عبارة عن مصفوفة مستطيلة تحتوي m سطر و n عمود، البنية التخزينية العادية للمصفوفات هي المصفوفات ثنائية البعد (أو شعاع vector ثنائي البعد) بما أن المصفوفات موجودة في جميع لغات البرمجة.

في بعض التطبيقات (مثل حل المعادلات التفاضلية) من الضروري معالجة مصفوفات كبيرة جداً تحوي بعض القيم غير الصفرية. إن استخدام مصفوفة ثنائية البعد لتخزين جميع القيم (بما في ذلك الصفرية) لمثل هذه المصفوفات المتفرقة ليس أمراً فعالاً. إن هذه المصفوفات يمكن تخزينها بشكل أكثر فعالية باستخدام بنية مترابطة مشابهة لتلك الخاصة بكثيرات الحدود المتفرقة.

أحد التحقيقات المترابطة الشائعة يتمثل بتمثيل كل صف من المصفوفة كلائحة مترابطة تخزن فقط القيم غير الصفرية في كل صف. تخزن المصفوفة في هذا الأسلوب كمصفوفة من المؤشرات A[m],.....,A[2],A[1] واحد لكل سطر من المصفوفة. كل عنصر في المصفوفة A[i] يشير إلى لائحة مترابطة من العقد، كل منها تخزن قيمة غير صفرية في ذلك السطر ورقم العمود الموجودة فيه، ورابط إلى العقدة التي تحوي القيمة التالية في ذلك السطر: المصفوفة 5×4:



$$A=\begin{vmatrix} 9 & 0 & 0 & 8 & 0 \\ 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & 6 & 0 & -8 & 0 \end{vmatrix}$$

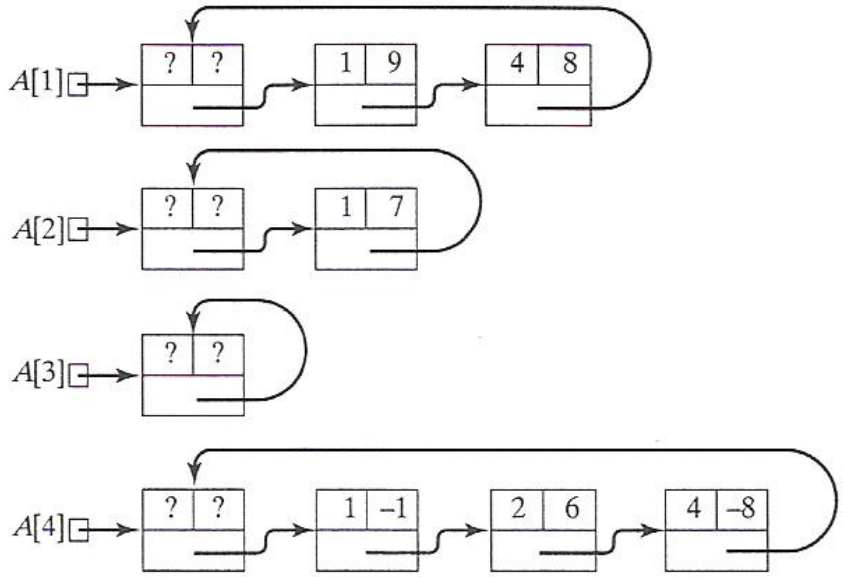
5 - لوائح أخرى متعددة الارتباط

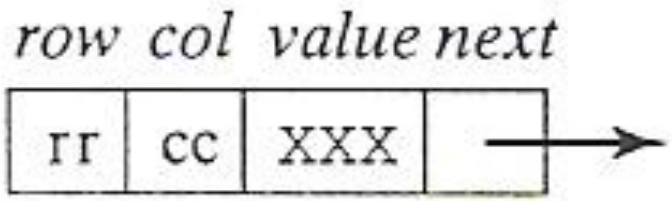
other multiply-linked lists

يمكن تمثيلها بالشكل:

على الرغم من أن استخدام البنية التخزينية المترابطة مع المصفوفات مفيد، فإن حجم المصفوفة يحد من عدد السطور التي يمكن لمثل هذه المصفوفة أن تحتويها، وأكثر من ذلك، فمن أجل المصفوفات الأصغر و/أو تلك التي تحوي عدداً كبيراً من السطور التي تحوي جميعها على قيم صفرية، فإن العديد من العناصر في المصفوفة ستهدر.

يمكن استخدام أسلوب آخر وذلك بإنشاء لائحة مترابطة أحادية، كل منها تحتوي رقم السطر، رقم العمود، القيمة غير الصفرية في ذلك السطر والعمود ورابط إلى العقدة التالية:

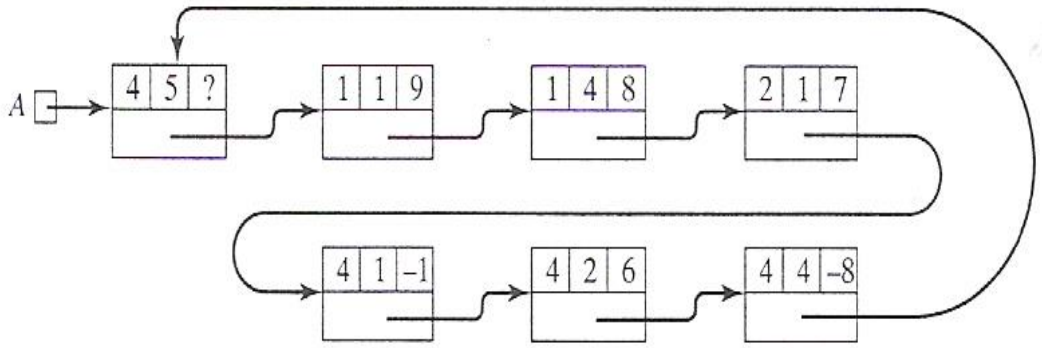


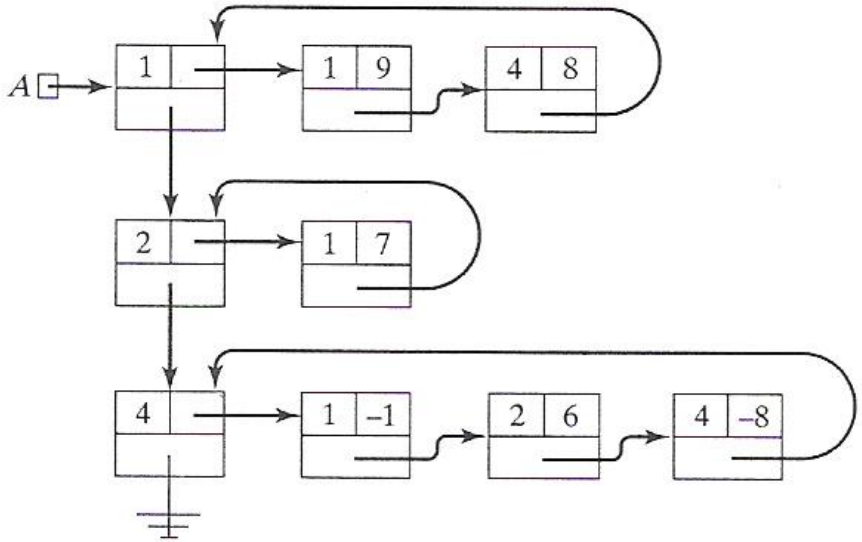


إن هذه العقد مرتبة عادة في اللائحة بحيث أن التجول عبر اللائحة يزور القيم في المصفوفة حسب ترتيب السطور، على سبيل المثال المصفوفة 5×4 السابقة يمكن تمثيلها باللائحة المترابطة الحلقية التالية، التي تستخدم عقدة رأس لتخزين أبعاد المصفوفة:

other multiply-linked lists

5 - لوائح أخرى متعددة الارتباط





بالمقابل، فإن هذا التحقيق يفقد إمكانية الوصول المباشر لكل سطر في المصفوفة.

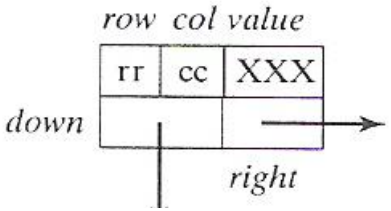
إذا كانت المعالجة بحسب الأسطر مهمة في بعض التطبيقات الخاصة، مثل جمع المصفوفات، فقد يكون من الأفضل استبدال مصفوفة المؤشرات بلائحة مترابطة من سطر عقد رأسية، كل منها يحتوي على مؤشر إلى لائحة سطر غير فارغ، كل عقدة رأسية لسطر ستحتوي أيضاً على رقم ذلك السطر ومؤشر إلى عقدة الرأس التالية، وهذه العقد الرأسية للسطر مرتبة بحسب الترتيب التصاعدي لأرقام السطور.

يمكن تمثيل المصفوفة 5×4 السابقة بهذه الطريقة كما يلي:

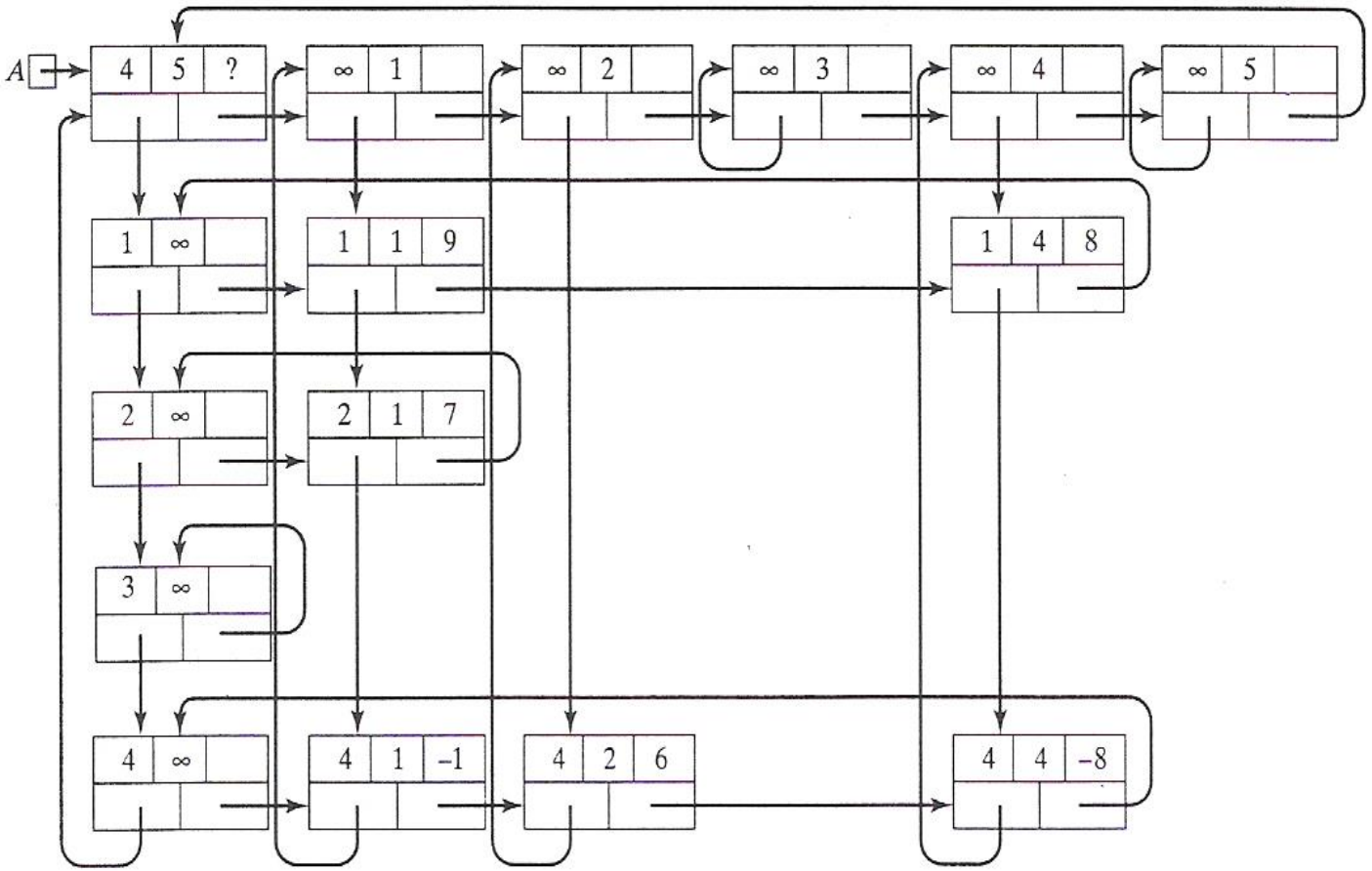
أحد عيوب هذا التحقيق المترابط هي أنه من الصعب معالجة المصفوفة بشكل عمودي عند الضرورة،

5 - لوائح أخرى متعددة الارتباط

other multiply-linked lists



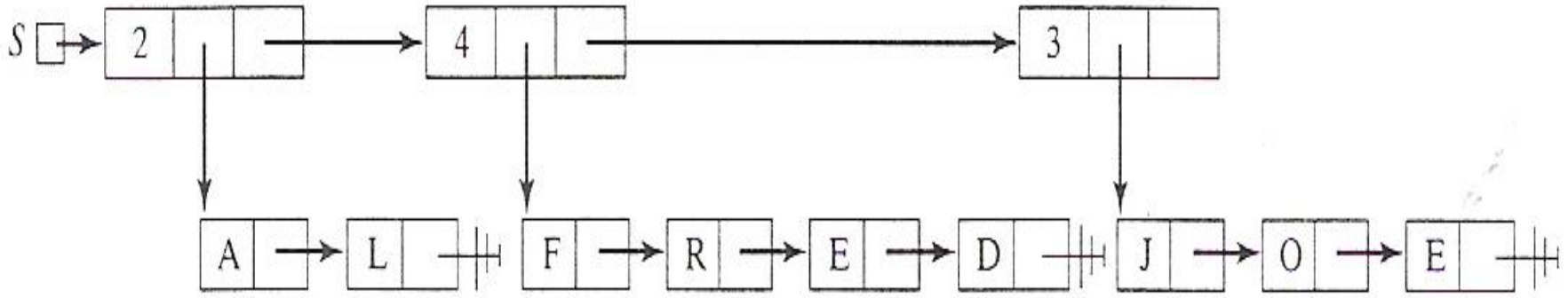
مثالاً: عند ضرب مصفوفتين. أحد البنى المترابطة التي تتيح وصولاً سهلاً لكل من الأسطر والأعمدة من المصفوفة تدعى اللائحة المتعامدة orthogonal list. كل عقدة تخزن رقم السطر، رقم العمود والقيمة غير الصفرية في ذلك السطر والعمود وتظهر على شكل لائحة سطر ولائحة عمود.



تحقيق باستخدام رابطين في كل عقدة، أحدهما يشير إلى العقدة التالية في لائحة السطر والآخر يشير إلى العقدة السابقة في لائحة العمود:

في جميع الأمثلة التي قمنا بدراستها تقريباً، كانت عناصر اللائحة عنصرية atomic الأمر الذي يعني أنها بحد ذاتها ليست لوائح. فقد درسنا، على سبيل المثال، لوائح الأعداد الصحيحة ولوائح سجلات الطلاب. في حين درسنا تمثيل المصفوفات المتفرقة التي هي لائحة من لوائح السطور.

في الحقيقة، يمكن تخزين السلسلة المحرفية في لائحة مترابطة من الأحرف، واللائحة المترابطة من السلاسل المحرفية يمكن أن تكون عندئذ لائحة مترابطة من اللوائح المترابطة. مثلاً: اللائحة S من الأسماء AL,FRED,JOE يمكن تمثيلها بالشكل:



تدعى اللوائح التي عناصرها يمكن أن تكون لوائح باسم اللوائح العامة generalized lists، وكتوضيح، لنفرض الأمثلة التالية من اللوائح:

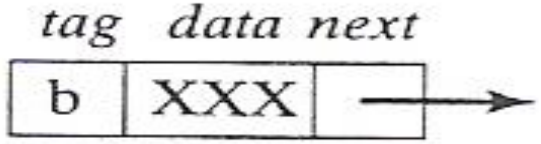
generalized lists

اللوائح العامة

A=(4,6) B=((4,6),8) C=(((4)),6) D=(2,A,A) E=(2,4,E)

هي لائحة عادية تحتوي قيمتين عنصريتين هما العدد الصحيح 4 والعدد الصحيح 6. B أيضاً هي لائحة من عنصرين، لكن العنصر الأول (4,6) هو بحد ذاته لائحة من عنصرين، C أيضاً هي لائحة من عنصرين، عنصرها الأول ((4)) هو لائحة مؤلفة من عنصر واحد (4)، وهذا العنصر بحد ذاته لائحة من عنصر واحد هو العدد الصحيح 4. D هي لائحة من ثلاثة عناصر فيها العنصرين الثاني والثالث هما بحد ذاتهما لوائح، وكذلك اللائحة Eتحوي ثلاثة عناصر، ولكنه تختلف عن D في أنها تحوي نفسها كعضو. مثل هذه اللوائح تدعى اللوائح العودية recursive lists.

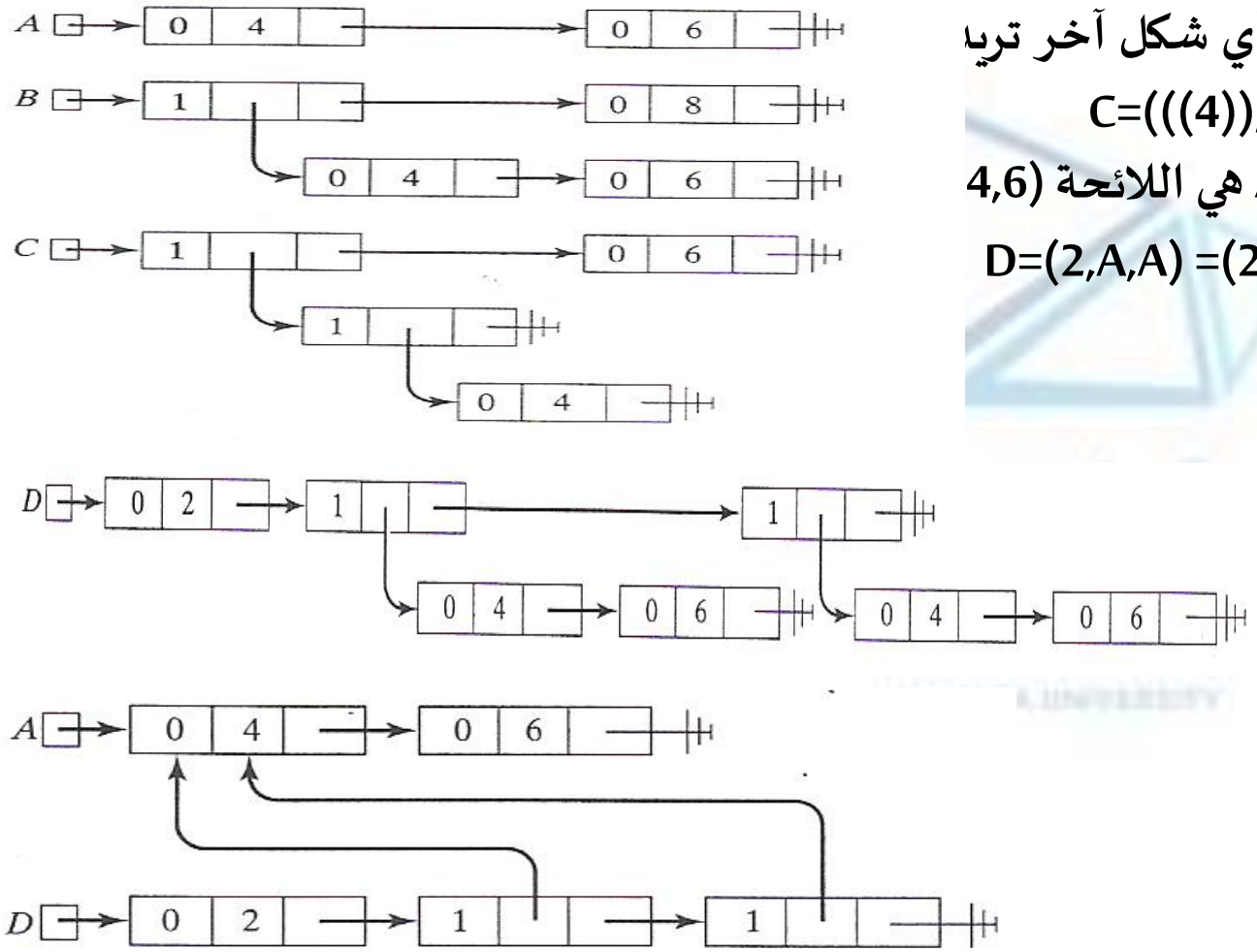
يمكن تمثيل اللوائح العامة كلائحة مترابطة تحوي العقدة فيها حقل مميز tag بالإضافة إلى جزء البيانات وجزء الرابط:



هذا المميز tag يستخدم للإشارة فيما إذا كان حقل البيانات يخزن قيمة عنصرية أو مؤشر إلى لائحة، ويمكن تمثيله كخانة وحيدة single bit، حيث 0 تعبر عن قيمة عنصرية و 1 تعبر عن مؤشر، أو كمتحول منطقي حيث true و false تقابل 0,1 وبالتالي يمكن تمثيل اللوائح A,B,C كما يلي:

generalized lists

اللوائح العامة

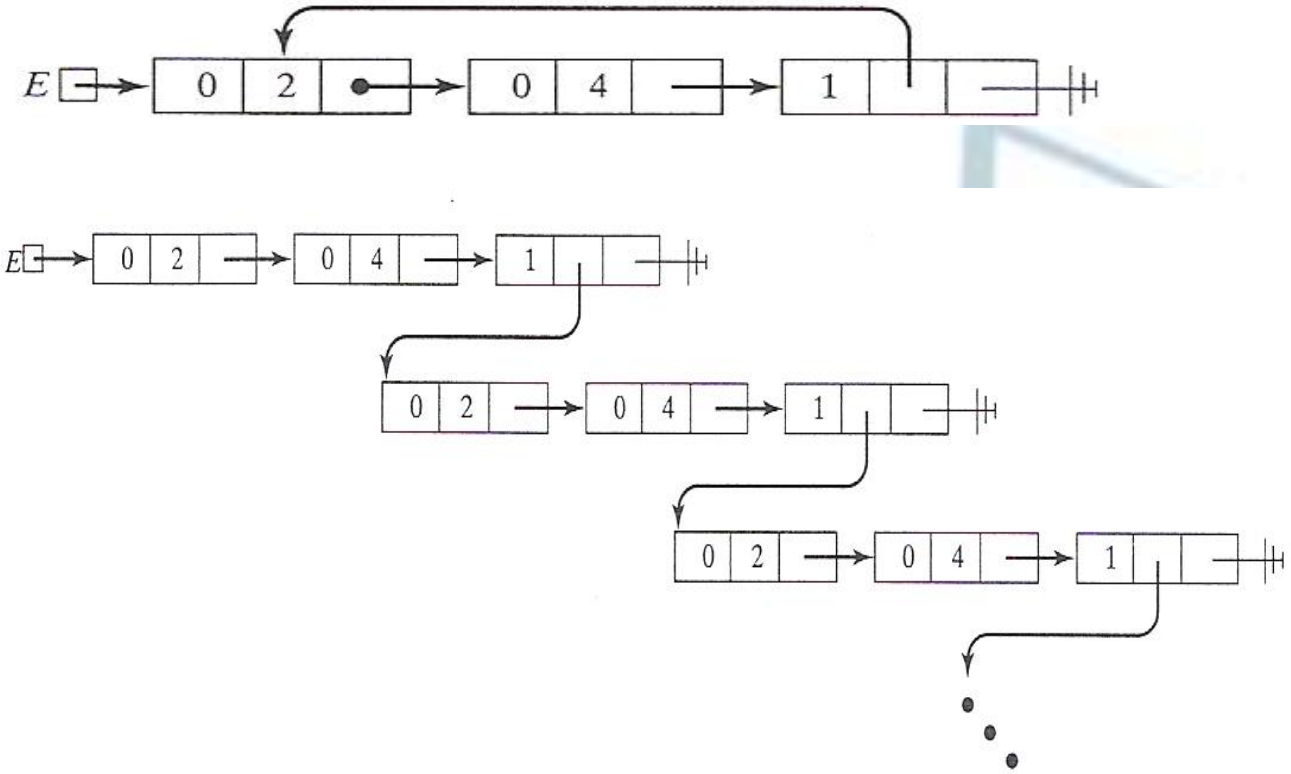


عن بالطبع، هذه اللوائح بعقدة رأس، أو حلقية، أو أي شكل آخر تريـ
استخدامه. A=(4,6) B=((4,6),8) C=(((4)),6)
هناك طريقتان ممكنتان لتحقيق اللائحة D، بما أن A هي اللائحة (4,6
يمكن اعتبار اللائحة D كما يلي: D=(2,A,A) =(2,(4,6),(4,6))
وتمثلها بالشكل التالي:

الطريقة الثانية هي السماح باستخدام اللوائح المشتركة shared lists وتمثيل D وA كما يلي:

generalized lists

اللوائح العامة



لاحظ في هذه الحالة أن تغيير A يغير D أيضاً.
اللائحة العودية E يمكن أن تمثل كبنية حلقية كما يلي:

E=(2,4,E)

وهي مكافئة للبنية اللانهائية التالية:

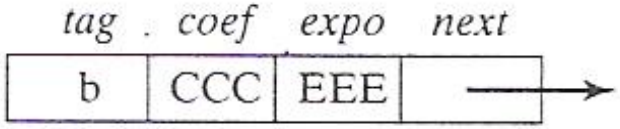
قمنا في فقرة سابقة بتمثيل كثيرات الحدود بمتحول واحد بشكل مترابط، كثيرات الحدود ذات أكثر من متحول واحد يمكن تمثيلها باستخدام اللوائح العامة ويمكن بالتالي تحقيقها باستخدام بنى مترابطة شبيهة لهذه البنى.
على سبيل المثال، ليكن كثير الحدود P(x,y) بمتحولين:

$P(x,y)=3+7x+14y^2+25y^7 9x^2y^7+18x^6y^7$

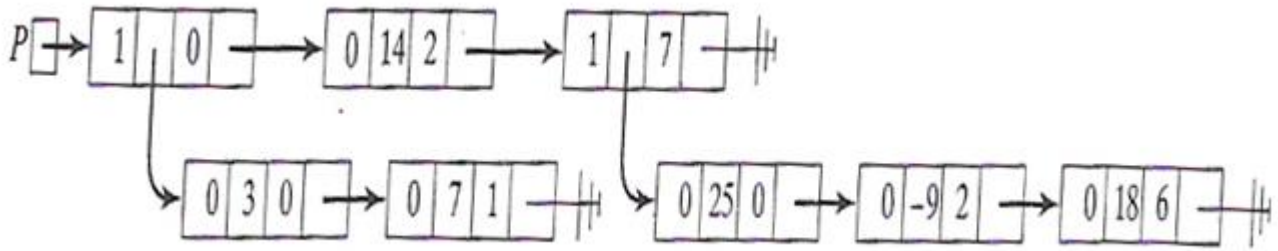
كثير الحدود هذا يمكن كتابته ككثير حدود بمتحول y معاملاته هي كثيرات حدود بدلالة x

$P(x,y)=(3+7x)+14y^2+(25-9x^2+16x^6)y^7$

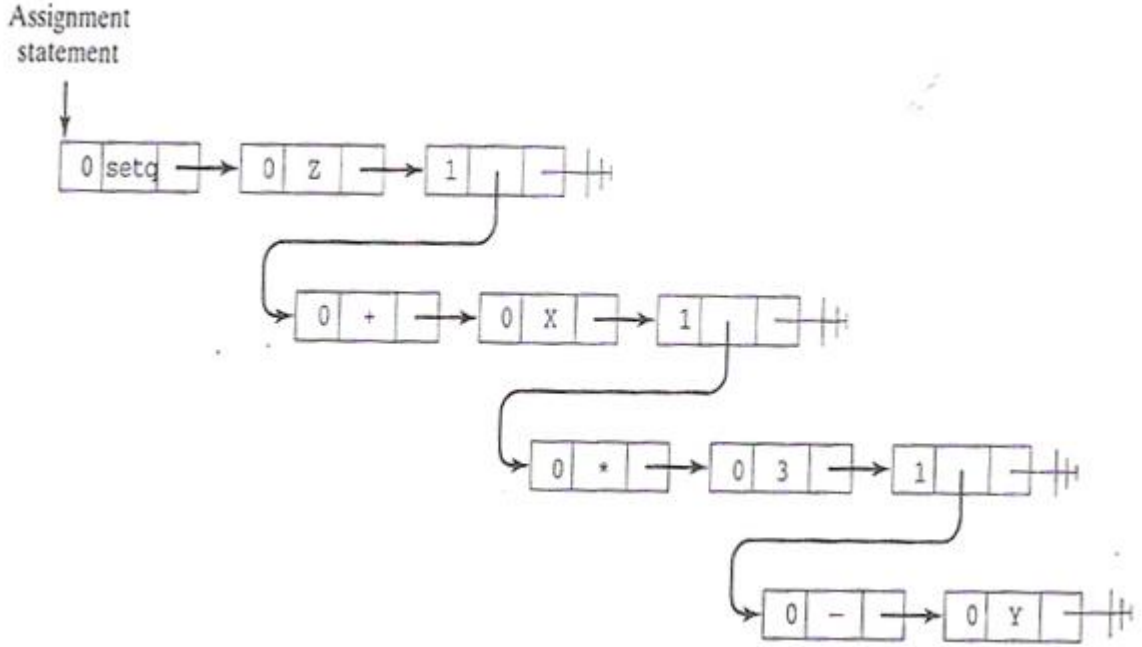
إذا استخدمنا عقداً من الشكل:



حيث tag=0 تشير إلى أن الحقل coef يخزن رقماً و tag=1 تشير إلى أنه يخزن مؤشراً إلى لائحة مترابطة تمثل كثير حدود بدلالة x. وبالتالي يمكن تمثيل كثير الحدود P(x,y) كما يلي:



إن التمثيل المترابط للوائح العامة يستخدم بكثرة في تحقيق لغة البرمجة LISP (LISt Processing) وكتوضيح تعليمة الإسناد التي يمكن كتابتها بلغة C++ كما يلي:



Z=X+3*(-Y) يكتب بلغة LISP كما يلي:

(setq Z(+X(*3(-Y))))

إن هذه في الحقيقة لائحة بثلاثة عناصر، العنصر الأول هو الكلمة المفتاحية setq التي تعني معامل الإسناد في لغة LISP والعنصر الثاني هو المتحول Z الذي تسند إليه قيمة والعنصر الثالث في اللائحة هو (+X(*3(-Y)))) والذي يمكن كتابته بلغة C++ كما يلي: X+3*(-Y) يحتوي هذا العنصر الثالث على معامل الجمع +، المتحول X واللائحة الممثلة للتعبير الجزئي 3*(-Y).

وبشكل مشابه تحتوي هذه اللائحة على المعامل * والعدد الصحيح الثابت 3 وعنصري لائحة يمثلان التعبير الجزئي Y–.

هذه اللائحة تحتوي على معامل الطرح الأحادي كعنصر أول والمتحول Y كنصر ثاني.

إذا استخدمنا عقداً فيها القيمة tag=0 تشير إلى قيمة لائحة عنصرية، مثل الكلمات المفتاحية، أسماء المتحولات والثوابت.

والقيمة tag=1 تشير إلى لائحة. فإن عملية الإسناد السابقة يمكن تمثيلها بالمخطط السابق:

انتهت المحاضرة السابعة

hash tables -3

تقوم خوارزميات البحث الخطي والبحث الثنائي المدروسة في فصل سابق بتحديد موقع عنصر في اللائحة من خلال سلسلة من المقارنات، في هذه المقارنات تتم مقارنة العنصر المراد البحث عنه مع عناصر اللائحة. من أجل مجموعة من n عنصر، يتطلب البحث الخطي O(n) مقارنة، بينما في البحث الثنائي يلزم $O(\log_2 n)$ مقارنة. تعمل هذه الخوارزميات في بعض الأحيان ببطء شديد، فعلي سبيل المثال يتم بناء جدول رموز symbol table من قبل المترجم لتخزين المعرفات المعلومات عنها. إن السرعة التي يتم بها إنشاء هذا الجدول والبحث فيه مرتبطة بسرعة الترجمة.

إن البحث السريع هو هدف الـ hash tableوالتي يتم تحديد موقع العنصر مباشرة بدلالة (كتابع) العنصر بحد ذاته بدلاً من سلسلة من المقارنات. وفي الظروف المثالية يكون الزمن اللازم لتحديد موقع عنصر في hash table هو O(1)، أي زمن ثابت ولا يتعلق بعدد العناصر المخزنة.

hash functions: لتوضيح الفكرة السابقة، بفرض أن 25 عدد صحيح كحد أقصى قيمها ضمن المجال من 0 وحتى 999 تم تخزينها في hash table. إن هذا الجدول يمكن تحقيقه كمصفوفة أعداد صحيحة تدعى table، يتم فيه تهيئة كل عنصر في المصفوفة بقيمة وهمية ما ولتكن 1-، إذا تم اعتبار كل عدد صحيح i كدليل، أي إذا قمنا بتخزين i في table[i] وبالتالي لمعرفة فيما إذا كان عدد ما number تم تخزينه، ما علينا سوى أن نختبر table[number] فيما إذا كان مساوياً لـ number. يدعى التابع h المعرف بـ h(i)=iوالذي يحدد موقع عنصر i في hash table باسم hash function.

Hash table

table[0]	500
table[1]	–1
table[2]	52
table[3]	–1
table[4]	129
table[5]	–1
⋮	⋮
table[23]	273
table[24]	49

يعمل التابع hash functionبشكل ممتاز لأن الزمن اللازم للبحث ضمن الجدول عن قيمة ما ثابت، نحتاج فقط لاختبار موقع واحد، وبالتالي فإن هذا الأسلوب فعال جداً من حيث الزمن ولكنه بالتأكيد غير الفعال من حيث المساحة حيث لا يتم استخدام سوى 25 موقع من المو اقع الـ 1000 المتاحة لتخزين العناصر، ويتم ترك 975 موقع غير المستخدمة أي استخدام 2.5% من المساحة المتاحة وهدر 97.5%.

بما أنه من الممكن تخزين 25 قيمة في 25 موقع، فإن بإمكاننا تحسين مدى استخدام المساحة باستخدام مصفوفة باسم table سعتها 25 وبدلاً من استخدام التابع السابق يتم استخدام التابع h(i)=i modulo 25 أو بلغة C++:

```
int h(int i)   { return i % 25; }
```

لأن هذا التابع ينتج عدداً صحيحاً محصوراً بين 0 و 24. العدد الصحيح 52 يخزن في table[2] حيث 52%25=2، وبشكل مشابه 129,500,273,49 تخزن في المو اقع 4,0,23,24 على التوالي.

استراتيجيات التصادم collision strategies:

إن هناك مشكلة في الجدول السابق وهي أن عملية تصادم collision قد تحدث. فعلى سبيل المثال إذا أردنا تخزين 77 فإنها يجب أن تخزن في h(77)=77%25=2 ولكن هذا الموقع مشغول سلفاً بالقيمة 52، وبنفس الطرقة هناك العديد من القيم قد تتصادم على الموقع نفسه مثل 2,27,102 وبشكل أعم جميع القيم من الشكل 25k+2 تأخذ الموقع 2 وبالتالي يجب وجود بعض الاستراتيجيات لتجاوز هذا التصادم.

أحد الاستراتيجيات البسيطة لمعالجة هذا التصادم تدعى السبر الخطي linear probing، في هذه الطريقة يتم بدء عملية بحث خطي للجدول من الموقع الذي وقع فيه التصادم وتستمر لحين الوصول إلى موقع فارغ يمكن تخزين القيمة فيه، وبالتالي عندما تتصادم القيمة 77 مع القيمة 52 على الموقع 2 فإننا نضع القيمة 77 في الموقع 3، لحشر القيمة 102فإننا نتبع تتالي السبر عن وجود المو اقع 2,3,4,5 حتى نصل أول موقع متاح وبالتالي نخزن 102 في table[5]، إذا وصل البحث إلى أسفل الجدول، نتابع إلى الموقع الأول، فعلى سبيل المثال يتم تخزين 123 في الموقع 1 بما أنها تتصادم مع 273 على الموقع 23 وتتالي السبر هو 23,24,0,1 يجد الموقع الفارغ التالي هو 1.

hash tables -3

table[0]	500
table[1]	123
table[2]	52
table[3]	77
table[4]	129
table[5]	102
:	:
table[23]	273
table[24]	49

لتحديد فيما إذا كانت قيمة ما موجودة في الجدول، نقوم أولاً بتطبيق التابع الـ hash function لحساب الموقع الذي يجب أن توجد فيه القيمة، هناك ثلاث حالات ممكنة: الأولى إذا كان الموقع فارغ فإننا نستطيع الاستنتاج مباشرة أن القيمة ليست في الجدول. الثانية إذا كان الموقع يحوي القيمة المحددة فإن البحث ينتهي مباشرة بنجاح. في الحالة الثالثة، إذا كان الموقع يحوي قيمة غير التي نبحث عنها بسبب الطريقة التي تم استخدامها في حل مسألة التصادم، في هذه الحالة نبدأ عملية بحث خطي حلقي من هذا الموقع ونستمر إلى أن نجد القيمة أو نصل إلى موقع فارغ أو نصل إلى موقع البداية مما يدل على أن القيمة غير الموجودة في الجدول.

إن زمن البحث في الحالتين الأولى والثانية ثابت، أما في الحالة الثالثة ليس كذلك. إذا كان الجدول ممتلئاً تقريباً فإن علينا حقيقة أن نختبر كل موقع قبل أن نجد العنصر أو نستنتج أنه غير اموجود في الجدول.

تحسينات: هناك ثلاثة أشياء يمكن القيام بها لتحسين الأداء:

- زيادة سعة الجدول.
- استخدام استر اتيجية مختلفة لحل التصادم.
- استخدام hash function آخر.

hash tables -3

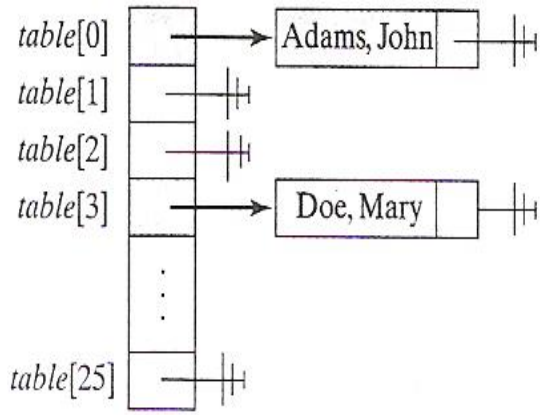
إن جعل سعة الجدول مساوية لعدد العناصر المراد تخزينها، كما في مثالنا الأول يعتبر عادة غير عملي، ولكن استخدام أي جدول أصغر يفتح مجالاً لاحتمال التصادم.

في الحقيقة، حتى لو كان الجدول قادراً على تخزين عدداً من القيم أكبر من اللازم، فإن التصادم سيحصل. على سبيل المثال، في hash table يحوي 365 موقع يراد تخزين 23 قيمة منتقاة عشوائياً، فإن احتمال حصول التصادم يتجاوز 0.5 وبالتالي من غير المنطقي اعتقاد أن هناك أسلوباً يمنع التصادم كلياً، بدلاً من ذلك يمكن أن نكتفي بـ hash tables **يحصل فيها عمليات تصادم محدودة.** تقترح الدراسات التجريبية استخدام جداول سعتها تقريباً ضعف ونصف أو ضعفي عدد العناصر المراد تخزينها.

الطريقة الثانية لتحسين الأداء هي تصميم طريقة أفضل للتعامل مع التصادم. في طريقة السبر الخطي، عند حصول التصادم يتم تخزين القيم المتصادمة في مو اقع يجب أن تكون محجوزة لعناصر يمكن الوصول إليها مباشرة. إن هذه الطريقة تجعل التصادمات المتتالية أكثر احتمالاً. وبالتالي تعقيد المشكلة.

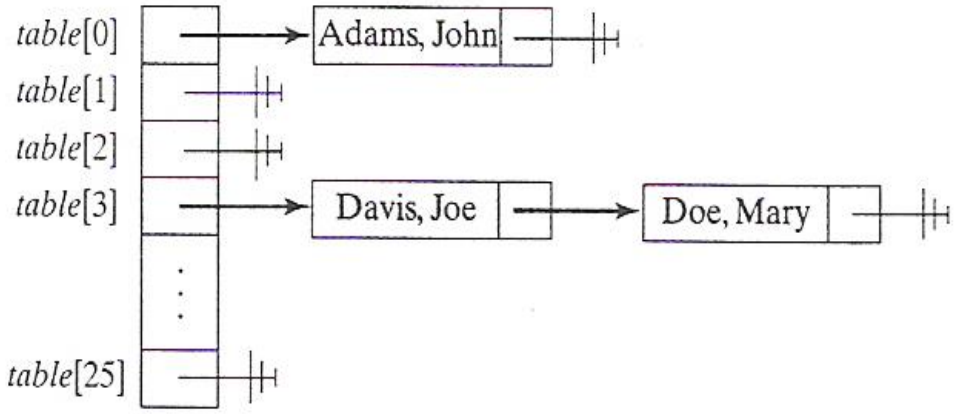
يمكن استخدام طريقة أفضل تدعى chaining تستخدم hash table عبارة عن مصفوفة أو شعاع vector من اللوائح المترابطة. للتوضيح نفرض أننا نرغب بتخزين مجموعة من الأسماء، بإمكاننا استخدام مصفوفة تدعى table مؤلفة من 26 لائحة مترابطة تكون فارغة في البداية، ونستخدم hash function بسيط h(name)=name[0]-'A' أي h(name) هو الصفر إذا كان الاسم يبدأ بحرف 'A'، 1 إذا بدأ بحرف 'B'... 25 إذا بدأ بحرف 'Z' وبالتالي مثلاً "Adams,Jones" و "Doe,Mary" تخزن في عقد مشار إليها بـ table[0] و table[3] على التوالي.

hash tables -3



عندما يحصل التصادم، نقوم ببساطة بحشر العنصر الجديد في اللائحة المترابطة الملائمة، مثالاً: بما أن h("Davis,Joe")=h('D'-'A')=3 سيحصل تصادم عند محاولة تخزين الاسم Davis,Joe وبالتالي نقوم بإضافة عقدة جديدة تتضمن هذا الاسم في اللائحة المترابطة الملائمة المشار إليها بواسطة table[3]:

إن عملية البحث ضمن هذا الجدول هي عملية أمامية مباشرة، نقوم ببساطة بتطبيق الـ hash function على العنصر المراد البحث عنه وعندها نستخدم أحد خوارزميات البحث ضمن اللائحة المترابطة.



هناك العديد من الاستراتيجيات الأخرى التي يمكن استخدامها لحل التصادم، على سبيل المثال، تدعى طريقة السبر الخطي باسم استراتيجية العنونة المفتوحة open addressing، هناك العديد من طرق العنونة المفتوحة التي تحاول تجزئة العناقيد clusters التي تتشكل من السبر الخطي باستخدام تتالي سبر مختلف، على سبيل المثال يستخدم السبر التربيعي quadratic probing:

$i+1^2$، $i-1^2$، $i+2^2$، $i-2^2$، $i+3^2$، $i-3^2$.....

العامل الثالث في تصميم الـ hash tables هو اختيار الـ hash function.

يؤثر سلوك الـ hash function على تكرار التصادمات، على سبيل المثال الـ hash function في المثال السابق ليس خياراً جيداً لأن بعض الأحرف تتكرر أكثر من الأخرى كأحرف أولى في الأسماء، وبالتالي اللائحة المترابطة بالأسماء التي تبدأ بالحرف 'S' ستكون أطول بكثير من تلك التي تحوي الأسماء التي تبدأ بالحرف 'Z' وبالتالي فإن عملية البحث عن أسماء تبدأ بالحرف 'S' ستكون أطول زمناً من الأسماء التي تبدأ بالحرف 'Z'. يمكن استخدام hash function أفضل يقوم بتوزيع الأسماء بطريقة أكثر انتظاماً عبر الـ hash table وهو متوسط الحرف الأول والحرف الأخير في الاسم:

h(name)=(firstLetter+lastLetter)/2

أو يمكن استخدام المتوسط الحسابي لجميع الأحرف، ولكن يجب أن لا يكون الـ hash function معقداً لدرجة أن الزمن اللازم لحسابه يجعل زمن البحث غير مقبولاً.

الـ hash function المثالي هو التابع السهل الحساب الذي يقوم ببعثرة العناصر في الـ hash tableوبالتالي يقوم بتقليل احتمال التصادم.

على الرغم من عدم وجود تابع واحد يقوم بذلك بشكل مثالي في جميع الأحوال فإن هناك طريقة شائعة تعرف باسم random hashing تستخدم تقنية توليد رقم عشوائي لبعثرة العناصر عشوائياً في الـ hash table حيث يتم تحويل العنصر إلى رقم عشوائي صحيح كبير باستخدام المعادلة من الشكل :

```
randomInt=((MULTIPLIER*item)+ADDEND)%MODULUS
```

ثم يتم اختصار هذه القيمة بحساب باقي قسمتها على سعة الجدول لتحديد موقع العنصر:

```
location=randomInt%CAPACITY;
```

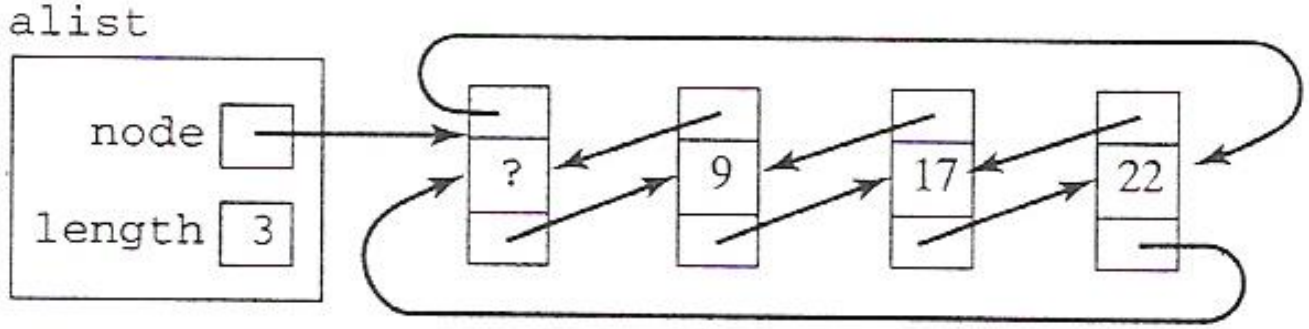
يمكن استخدام هذا التابع مع عناصر غير الأعداد الصحيحة إذا قمنا بترميز مثل هذه العناصر كأعداد صحيحة، على سبيل المثال يمكن ترميز الاسم كمجموع الترميز ASCII لبعض أو كل أحرفه.

4- اللوائح المترابطة المضاعفة والصنف القياسي list في C++

4 -doubly-linked lists and the standard C++ list

نظرة عامة إلى الصنف القياسي list في لغة C++:

القالب القياسي list هو عبارة عن حاوي تتابعي يتيح عملية الحشر والحذف في أي نقطة من هذا التتابع. يقوم هذا القالب بتخزين عناصر اللائحة في لائحة مترابطة مضاعفة حلقية تملك عقدة رأس. على سبيل المثال يمكن تمثيل اللائحة ذات العناصر الثلاثة aList المبنية كما يلي:




```
list <int> aList;
aList.push_back(9);
aList.push_back(17);
aList.push_back(22);
```

يشير عضو البيانات node إلى العقدة الرأسية وعضو البيانات length هو عدد العناصر في اللائحة.

<u>إدارة الذاكرة في الصنف list:</u> بنظرة عامة يبدو الصنف list بسيطاً جداً، في حين أن أسلوبه في تخصيص / إلغاء تخصيص الذاكرة أعقد بشكل ملحوظ من استخدام العمليتين new و delete. يستخدم هذا الصنف أسلوباً خاصاً في إدارة العقد الحرة المتاحة للتقليل من عدم الفعالية في استخدام مدير الكومة heap manager.

4 -doubly-linked lists and the standard C++ list

4- اللوائح المترابطة المضاعفة والصنف القياسي list في C++

يقوم هذا الأسلوب بالمحافظة على العقد الحرة على شكل مكدس مترابط، ويستخدم التابع new فقط عندما تكون هذه اللائحة الحرة فارغة، ومن ثم للحصول على قطاعات ضخمة من الذاكرة التي تجزئتها إلى عقد بحجم مناسب تقوم بوضعها في لائحة العقد الحرة. وتستخدم العملية delete لإعادة العقد إلى مخزن النظام من العقد الحرة عندما ينتهي زمن حياة جميع اللوائح من نمط معين. يمكن التعبير عن الخوارزمية الأساسية لإدارة العقد كما يلي:

for each list of a certain type T:

- when a node is needed:
  - if there is a node on the free list , allocate it.
  - if the free list is empty:
    - call the system's heap manager to allocate a block (called a buffer) of size (typically) 4KB.
    - carve it up into pieces of the size required for a node of a list<T>.
    - put these nodes on the free list.
- when a node is deallocated:

  push it onto the free list.
- when all lists of the type T have been destroyed:

  return all buffers to the heap.

التابع الباني، التابع size( )، التابع empty( ): إن تحقيق أغلب العمليات الأساسية للوائح مشابه تماماً لتلك التي قمنا بشرحها في الفقرة السابقة الخاصة باللوائح المترابطة الأحادية وفي هذه الفقرة من أجل اللوائح المترابطة المضاعفة. على سبيل المثال، من أجل غرض list<T> معرف كما يلي: list<T> alist;

4- اللوائح المترابطة المضاعفة والصنف القياسي list في C++

4 -doubly-linked lists and the standard C++ list

يقوم الباني الافتراضي للصنف list ببناء لائحة فارغة aList بالحصول على عقدة رأس من اللائحة الحرة ويقوم بتخزين عنوان هذه العقدة في عضو البيانات node الخاص بها:

التابع الباني، التابع ( )size، التابع ( )empty: إن تحقيق أغلب العمليات الأساسية للوائح مشابه تماماً لتلك التي قمنا بشرحها في الفقرة السابقة الخاصة باللوائح المترابطة الأحادية وفي هذه الفقرة من أجل اللوائح المترابطة المضاعفة. على سبيل المثال، من أجل غرض list<T> معرف كما يلي: list<T> alist;