الفصل السادس

بناء بنى المعطيات باستخدام الصفوف والمؤشرات

هدف الفصل

- دمج مفهومي الصفوف والمؤشرات في صنع مفاهيم متقدمة
- السيطرة على مفهوم المؤشرات والحجز الديناميكي للذاكرة
- إتقان بناء الصفوف المركبة
- تعلم بناء بنى معطيات مفيدة جداً (كاللوائح المترابطة)
- تعلم إدارة بنى المعطيات

1. مقدمة

إذا عدت إلى الأنماط التي تعرفت إليها في اللغة قبل أن تتعرف إلى المؤشرات، لوجدت أنها لا تسمح بالإدارة الديناميكية للذاكرة، لاسيما عندما يتعلق الأمر بحجز كتل في الذاكرة سواء باستخدام المصفوفات أو الصفوف.

وإذا تأملت في المصفوفات التي تعاملت معها سابقاً، لوجدت أنك يجب أن تحدد طول المصفوفة منذ التصريح عنها وذلك إما بوضع الطول بين القوسين [ ] عند التصريح، أو من خلال عدد القيم الابتدائية المستخدمة. وعند الحاجة إلى التعامل مع المصفوفات في البرامج، وإلى كتابة توابع تمرر لها المصفوفات كموسطات يجب تمرير طول المصفوفة للتابع مع تمرير المصفوفة كما في مثال selectionSort الذي مر معك في فصل

" المؤشرات pointers والإدارة الديناميكية للذاكرة dynamic memory management"

سترى كيف ننتقل بالتدريج وبخطوات متتالية من النمط المعرف في أصل C++ للمصفوفات إلى صفوف تعرف أنماطاً تمثل المصفوفات بخصائص معينة ومنها إلى صفوف تمثل بنى معطيات أكثر شمولية من المصفوفات.

2. الصف الأول vect

بما أن طول المصفوفة هو عدد يلازم عناصر المصفوفة عند كل عملية عليها، فلماذا لانعرف صفاً يحتوي في قسمه الخاص على أعضاء المعطيات التالية:

- حقل يمثل عناصر المصفوفة

```
int arr[];
```

- حقل يمثل طول المصفوفة

```
int size;
```

غير أنه لابد في هذه الحال من تحديد الطول الأعظمي للمصفوفة arr. سنختار هنا على سبيل المثال الطول 100.

وفيما يلي تصريحنا عن الصف vect:

```
#include <iostream>
using namespace std;

class vect
{ public:
Vect();
vect(int s);
int getsize();
int &operator[](int);

private:
int arr[100];
int size;
};
vect::vect()
{
 size=10;
  for(int i=0;i<size;i++) arr[i]=0;
}

vect::vect(int s)
{
if( s>0 && s<=100)
     {size=s;
      for(int i=0;i<=size-1;i++) arr[i]=0};
```

```
 else { cerr << "\nError: size " << s
        <<  "out of range" << endl;
        exit(1);}
}
int vect::getsize()
{return size};

int &vect::operator[](int i)
{
     if(i<size && i>=0)
           return arr[i];
     else
     { cerr << "\nError: Subscript " << i
<<"out of range" << endl;
        exit(1);}
}
int main()
{vect ;int I;
v[0]=5;
for( i=0;i<10;i++)
cout<<v[i]<<"  ";
vect v2(4);
v2[0]=1; v2[1]=4;
v2[2]=7; v2[3]=9;

for( i=0;i<v2.getsize();i++)
cout<<v2[i]<<"  ";

for( i=0;i<v2.getsize();i++)
cin>>v2[i];

for( i=0;i<v2.getsize();i++)
cout<<v2[i]<<"  ";
return 0;
}
```

لاحظ أنك لو صرحت عن متحول vect v(200); أو vect v(-3); لظهرت الرسالة:

Error: size 200 out of range

أو Error: size -3 out of range

وكذلك الأمر لو حاولت الوصول إلى الخانة v2[6] أو v2[300]:

```
cout<<v2[6];  //outputs the message:  Error: Subscript 6 out of range
v2[300]=9; //outputs the message:  Error: Subscript 300 out of range
```

ماميزة هذا النمط مقارنة بنمط المصفوفة المعرف سلفاً في C++؟

1- إن عملية التدقيق التي تجري في الباني vect(int s) لعدم تجاوز الطول الأعظمي للمصفوفة تحمي من حصول التجاوز الذي كان يقع في النمط المعرف في أصل C++ للمصفوفات.

2- إن عملية التحقق من وقوع الدليل بين 0 وطول المصفوفة في التابع int &operator[](int) الذي يسمح بزيادة تحميل المؤثر [ ] تحمي من حصول التجاوز الذي كان يقع في النمط المعرف في أصل C++ للمصفوفات.

3- لقد أصبحت المصفوفة الآن ممثلة بغرض واحد، وعند تمرير المصفوفة كموسط نمرر غرضاً واحداً يتضمن في معطياته الأعضاء عناصر المصفوفة وطولها.

وهكذا يمكننا أن نكتب التابع selectionSort كما يلي:

```
void selectionSort( vect &v )
{
   int smallest; // index of smallest element
   for ( int i = 0; i < v.getsize() - 1; i++)
{
      smallest = i; // first index of remaining array
      // loop to find index of smallest element
      for ( int index = i + 1; index < v.getsize(); index++)
         if ( v[ index ] < v[ smallest])
            smallest = index;
      swap( v[ i ], v[ smallest]);
      for ( int j = 0; j < v.getsize(); j++)
          cout << setw( 4 ) << v[ j ];
      cout<<endl;
   }  // end if
}
```

هل حللنا بتعريف نمط المصفوفة بهذا الشكل جميع نقاط الضعف التي تؤخذ على النمط المعرّف في أصل C++ للمصفوفات؟

لا، فإن تحديد الطول الأعظمي للمصفوفة يعني أننا في حال احتجنا إلى مصفوفة أطول فعلينا تعديل الصف، وإعادة ترجمته.

لحل هذه المشكلة سنلجأ إلى تعريف نمط ثان للمصفوفات Array يسمح بإجراء حجز ديناميكي للمصفوفة عند التنفيذ.

3. الصف الثاني Array

لنصرح عن الصف Array بحيث تتألف معطياته الأعضاء من مؤشر على عدد صحيح ومن عدد يمثل طول المصفوفة.

- حقل يمثل مؤشر على عناصر المصفوفة

```
int *ptr;
```

- حقل يمثل طول المصفوفة

```
int size;
```

إن ptr هو مؤشر يشير إلى كتلة في الذاكرة تتضمن عناصر المصفوفة. يجري حجز هذه الكتلة ديناميكياً عند التنفيذ حيث يحجز في الذاكرة كتلة تتناسب مع عدد العناصر الممرر كمحدد للباني الافتراضي. ويجري تحرير هذه الكتلة كذلك دفعة واحدة عندما يستدعى الهادم.

مثال:

```
#include <iostream>
#include <iomanip>
using namespace std;

class Array
{
public:
   Array( int = 10 ); // default constructor
~  Array();
   int getSize();
   int &operator[](int);

private:
   int *ptr;
   int size;
};
Array::Array( int s)
{
   if (s>0)
         size = s ;
   else size= 10 ;
   ptr = new int[ size];

   for ( int i = 0; i < size; i++)
      ptr[ i ] = 0;
}

// destructor for class Array
Array::~Array()
{
 delete [] ptr;
}
// return number of elements of Array
int Array::getSize ()
{
   return size;
}
int &Array::operator[]( int subscript )
{
   // check for subscript out-of-range error
   if ( subscript < 0 || subscript >= size)
  {
      cerr << "\nError: Subscript " << subscript
<<"out of range" << endl;
```

```
      exit( 1 );
 }
   return ptr[ subscript ]; // reference return
 }
int main()
{
   Array integers1( 7 ); // seven-element Array
   Array integers2; // 10-element Array by default
   int I;
   // print integers1 size and contents
   cout << "Size of Array integers1 is "
      <<integers1.getSize()
         <<"\nArray after initialization:\n";
   for (i=0;i< integers1.getSize();i++)
         cout<<integers1[i];

   // print integers2 size and contents
   cout << "\nSize of Array integers2 is"
        <<integers2.getSize()
        <<"\nArray after initialization:\n";
   for (i=0;i< integers2.getSize();i++)
         cout<<integers2[i];
   // input and print integers1 and integers2
   cout << "\nEnter 17 integers:" << endl;
   for (i=0;i< integers1.getSize();i++)
         cin>>integers1[i];
   for (i=0;i< integers2.getSize();i++)
         cin>>integers2[i];

   cout << "\nAfter input, the Arrays contain:\n"
      <<"integers1:\n";
   for (i=0;i< integers1.getSize();i++)
         cout<<integers1[i];
   cout<< "\nintegers2:\n";
   for (i=0;i< integers2.getSize();i++)
         cout<<integers2[i];
  cout << "\nintegers1[5] is " << integers1[ 5 ];

  cout << "\n\nAssigning 1000 to integers1[5]" << endl;
  integers1[ 5 ] = 1000;
  cout << "integers1:\n";
  for (i=0;i< integers1.getSize();i++)
         cout<<integers1[i];
  // attempt to use out-of-range subscript
   cout << "\nAttempt to assign 1000 to integers1[15]"
<< endl;

   integers1[ 15 ] = 1000; // ERROR: out of range
```

```
  return 0;
}
```

ماميزة النمط Array مقارنة بالنمط vect ؟

يجري تحديد طول المصفوفة ديناميكياً، ولايوجد حد أعظمي لطول المصفوفات التي يمكننا التصريح عنها، وأياً كان الطول المرغوب فيكفي تمريره للباني الافتراضي دون إعادة ترجمة الصف.

هل حللنا بتعريف النمط Array بهذا الشكل جميع نقاط الضعف التي تؤخذ على vect؟

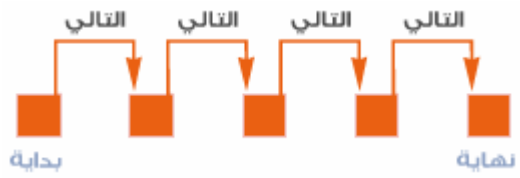
الجواب: لا، فإن تحديد طول المصفوفة ديناميكياً يعني حجز حجم من الذاكرة يوافق الطول المطلوب، ولكن إن احتجنا بعد ذلك إلى إطالة هذه المصفوفة أو تقصيرها حسب عدد العناصر التي تضاف إلى المصفوفة أو تحذف منها أثناء تنفيذ العمليات على المصفوفة، أي لو احتجنا إلى حجز جزء إضافي يلحق بنفس الكتلة أو إعادة جزء منها فإننا لن نتمكن من ذلك لأن عمليتي new[ ] و delete [ ] تحجزان كتلة كاملة أو تعيدانها كاملة.

لحل هذه المشكلة سنلجأ إلى تعريف نمط جديد List يسمح بالحجز الديناميكي عند التنفيذ مع إمكان إضافة أو حذف على مستوى العنصر.

4. الصف الثالث List

<u>مفهوم اللائحة</u>

يعتمد هذا الصف مفهوم اللائحة List. تعرف اللائحة بأنها بنية تحوي عناصر من نفس النمط. وتكون اللائحة متتالية sequential list إذا كان يوجد ترتيب للعناصر فيها، بحيث يوضع العنصر الأول في بداية اللائحة، والعنصر الأخير في نهايتها، ويكون لكل عنصر فيها -عدا العنصر الأخير - عنصر تالٍ له. يمكن الوصول إلى أي عنصر في اللائحة انطلاقاً من بدايتها بالعبور من عنصر إلى العنصر التالي له باتجاه النهاية.



لتمثيل اللائحة المتتالية العناصر سنصرح عن الصف List الذي يتضمن عمليات إقحام عناصر جديدة insert ونزع عناصر من اللائحة remove. لتعريف الصف، نصرح عن الواصفات attributes والتوابع العامة لإدارة اللائحة، أي توابع الإقحام والحذف التي تتعامل مع طرفي اللائحة أي مع بدايتها ونهايته وهما التابعان insertAtFront و insertAtBack للإقحام والتابعان RemoveFromFront و RemoveFromBack للحذف. كما سنعرف التابع isEmpty لفحص إذا كانت اللائحة فارغة، وprint لطباعة اللائحة .

قبل تفصيل كيفية بناء اللائحة، لنتعرف على توابع الصف List لفهم كيفية عملها:

- **الباني:** ينشئ لائحة فارغة

```
List l;
```

- **الإقحام في البداية:** يقحم عنصراً في بداية اللائحة:

```
l.insertAtFront(1 );
```


```
l.insertAtFront(2 );
```

- **الإقحام في النهاية:** يقحم عنصراً في نهاية اللائحة:

```
l.insertAtBack(5 );
```

- **الحذف من البداية:** يحذف عنصراً من بداية اللائحة:

I. removeFromFront (v);

- **الحذف من النهاية:** يحذف عنصراً من نهاية اللائحة:

I. removeFromBack (v);

سندخل الآن في تفاصيل بناء اللائحة.

الصف Node

يتضح من المثال أن اللائحة تتكون من عناصر منفصلة يتضمن كل منها قيمة ما،غير أن وجود الترتيب الذي ذكرناه سابقاً بين العناصر، وعمليات الإقحام الديناميكية التي تجرى في البداية والنهاية يعني وجود تتالٍ معين للعناصر، وهذا ما أشرنا إليه بقولنا لائحة متتالية. لتحقيق هذا المفهوم، وجب أن ينتهي كل عنصر بمؤشر على العنصر الذي يليه، بالطبع عدا العنصر الأخير الذي لاينتهي بمؤشر للعنصر التالي لعدم وجود عنصر تال.

نستنتج إذاً أن كل عنصر من اللائحة يشبه عقدة node في سلسلة، ويتألف من واصفتين:

- واصفة لقيمة العنصر
- واصفة للمؤشر على العنصر التالي

نحتاج إذاً إلى بناء الصف Node ، وهو يمثل الصف الذي تنتمي إليه كل عقدة في اللائحة. يمكن تمثيل عناصر الصف Node كما في الشكل التالي:

وفيما يلي التصريح عن الصف Node :

```
#ifndef Node_H
#define Node_H

//forward declaration of class List required to
announce that class
// List exists so it can be used in the friend
declaration at line 13
class List;

class Node
{
   friend class List; // make List a friend

public:
   Node( const int & ); // constructor
```

```
   int getData() const; // return data in node
private:
   int data; // data
   Node *nextPtr; // next node in list
};// end class Node
 //constructor
 Node::Node( const int &info)
    : data( info ), nextPtr( 0 )
 {
    // empty body
 }   // end Node constructor
 // return copy of data in node
 int Node::getData() const
 {
    return data;
 } // end function getData

 #endif
```

ملاحظات على الصف Node

1- تأمل الأعضاء الخاصة المصرح عنها في الصف Node:

```
   private:
      int data; // data
      Node *nextPtr; // next node in list
```

ماذا تلاحظ؟

إن العضو الثاني nextPtr هو مؤشر إلى عنصر من النمط Node !!

وهكذا أثناء التصريح عن الصف Node نستخدم عضواً هو مؤشر إلى عنصر من النمط Node نفسه !!

يسمى هذا النوع من الصفوف الصف ذاتي المرجعية Self-Referential class، ويعتبر المؤشر الذي يشير إلى عنصر من نمط الصف (وهو هنا nextPtr) بمثابة رابط link بين عناصر الصف.

يستخدم هذا النوع من الصفوف للتمكن من ربط أغراض هذا الصف بعضها مع بعض للحصول على بنى معطيات متقدمة مثل اللوائح والأرتال والمكدسات والأشجار.

2- يتضمن الصف Node تصريحاً عن الصداقة (راجع الفقرة " التوابع الصديقة friend function والصفوف الصديقة friend class" في الفصل الرابع) بينه وبين الصف List . مافائدة هذه الصداقة؟

بالطبع تستخدم الصداقة لتمكين أغراض الصف List من النفاذ مباشرة إلى الأعضاء الخاصة للصف Node دون استخدام توابع النفاذ، أي لخرق خاصة الكبسلة التي يتمتع بها كل غرض من الصف Node ولكن هذا الخرق مسموح فقط لأغراض الصف الصديق List. سنرى مسببات هذا الخرق عند تعريف الصف List.

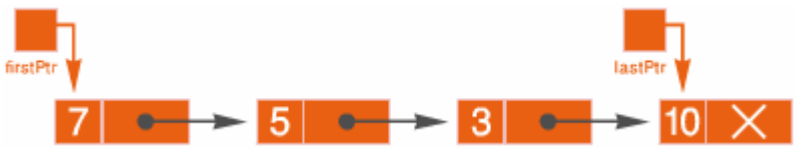
الآن وقد صرحنا عن الصف Node سنستخدمه لنصرح عن الصف List.

الصف List

يمكننا النظر إلى اللائحة على أنها تتألف من سلسلة من العقد، ونظراً لكون عقدتي البداية front والنهاية back تتمتعان بوضع خاص، فإن السلسلة تتضمن مؤشرين إليهما هما firstPtr و lastPtr. تجدر الإشارة إلى أنه لا يوجد عقدة تالية لعقدة النهاية، ومن ثم، فإن قيمة مؤشرها هي NULL. تسمى هذه البنية لائحة مترابطة linked list.

إذاً تتألف اللائحة من واصفتين:

- واصفة للمؤشر على عقدة البداية
- واصفة للمؤشر على عقدة النهاية



لائحة مترابطة

وفيما يلي التصريح عن الصف List :

```
#ifndef LIST_H
#define LIST_H

#include "Listnode.h" // ListNode class definition
#include <iostream>
using namespace std;
class List
{
public:
   List(); // constructor
  ~List(); // destructor
   void insertAtFront( const int  & );
   void insertAtBack( const int & );

   bool removeFromFront( int& );
   bool removeFromBack( int & );
   bool isEmpty() const;
   void print() const;
private:
   Node *firstPtr; // pointer to first node
   Node *lastPtr; // pointer to last node

   //utility function to allocate new node
   Node *getNewNode( const int & );

};   // end class List
//default constructor

List::List ()
   : firstPtr( 0 ), lastPtr( 0 )
{
   // empty body
 }  // end List constructor
// destructor

List::~List()
{
   if ( !isEmpty() ) // List is not empty
     {
      cout << "Destroying nodes ...\n";

      Node *currentPtr = firstPtr;
      Node *tempPtr;
```

```
      while ( currentPtr != 0 ) // delete remaining nodes
      {
         tempPtr = currentPtr;
         cout << tempPtr->data << '\n';
         currentPtr = currentPtr->nextPtr;
         delete tempPtr;
        }//end while
  }// end if

   cout << "All nodes destroyed\n\n";
 }// end List destructor
 // insert node at front of list
 void List::insertAtFront( const int &value)
 {
    Node *newPtr = getNewNode( value ); // new node

    if ( isEmpty() ) // List is empty
       firstPtr = lastPtr = newPtr; // new list has only one node
    else // List is not empty
 {
       newPtr->nextPtr = firstPtr; // point new node to previous
 1st node
       firstPtr = newPtr; // aim firstPtr at new node
    }  //end else
 }// end function insertAtFront
// insert node at back of list
void List::insertAtBack( const int &value )
{
   Node *newPtr = getNewNode( value ); // new node

   if ( isEmpty() ) // List is empty
      firstPtr = lastPtr = newPtr; // new list has only one node
   else // List is not empty
}
      lastPtr->nextPtr = newPtr; // update previous last node
      lastPtr = newPtr; // new last node
   }// end else
 }// end function insertAtBack
 // delete node from front of list
 bool List::removeFromFront( int &value)
 {
    if ( isEmpty() ) // List is empty
       return false; // delete unsuccessful
    else
    {
       Node *tempPtr = firstPtr; // hold tempPtr to delete

       if ( firstPtr == lastPtr )
```

```
         firstPtr = lastPtr = 0; // no nodes remain after removal
      else
         firstPtr = firstPtr->nextPtr; // point to previous 2nd
node

      value = tempPtr->data; // return data being removed
      delete tempPtr; // reclaim previous front node
      return true; // delete successful
}// end else
}// end function removeFromFront
// delete node from back of list
bool List::removeFromBack( int &value )
{
   if ( isEmpty() ) // List is empty
      return false; // delete unsuccessful
   else
{
      Node *tempPtr = lastPtr; // hold tempPtr to delete
      if ( firstPtr == lastPtr ) // List has one element
         firstPtr = lastPtr = 0; // no nodes remain after removal
      else
      {
         Node *currentPtr = firstPtr;
         // locate second-to-last element
         while ( currentPtr->nextPtr != lastPtr )
            currentPtr = currentPtr->nextPtr; // move to next node
         lastPtr = currentPtr; // remove last node
         currentPtr->nextPtr = 0; // this is now the last node
      }// end else
      value = tempPtr->data; // return value from old last node
      delete tempPtr; // reclaim former last node
      return true; // delete successful
   }// end else
}//end function removeFromBack
// is List empty?
bool List::isEmpty() const
{
   return firstPtr == 0;
}//end function isEmpty

//return pointer to newly allocated node
Node *List::getNewNode( const int &value )
{
   return new Node( value);
}//end function getNewNode
// display contents of List
void List::print() const
{
```

```
    if ( isEmpty() ) // List is empty
    {
       cout << "The list is empty\n\n";
       return;
    }//end if
    Node *currentPtr = firstPtr;
    cout << "The list is:";
    while ( currentPtr != 0 ) // get element data
   {
       cout << currentPtr->data <<'  ';
       currentPtr = currentPtr->nextPtr;
   }// end while
    cout << "\n\n";
 }//end function print
#endif
```

شرح التوابع الأعضاء

1- الباني:

يبدأ الباني بإنشاء لائحة فارغة أي ليس فيها أية عقدة، ومعنى هذا أن كلا المؤشرين firstPtr و lastPtr فيهما 0 (NULL)

2- الهادم:

إن المهمة الأساسية للهادم هي إعادة الذاكرة الديناميكية التي جرى حجزها لعقد اللائحة.

تبدأ الإجرائية بالتحقق من وجود عناصر في اللائحة أي من أنها ليست فارغة (!isEmpty()، وفي حال كانت غير فارغة، نستخدم مؤشر يسمى currentPtr يشير إلى عقد اللائحة الواحدة تلو الأخرى اعتباراً من عقدة البداية، وكلما نقلناه إلى العقدة التالية حذفنا العقدة السابقة باستخدام delete.

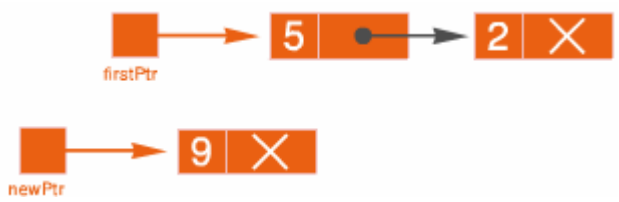
3- الإقحام في بداية اللائحة insertAtFront:

إن المهمة الأساسية لهذا الإجراء هي إنشاء عقدة جديدة ووضعها في بداية اللائحة. ينفذ الإجراء مهمته على مراحل هي كالتالي:

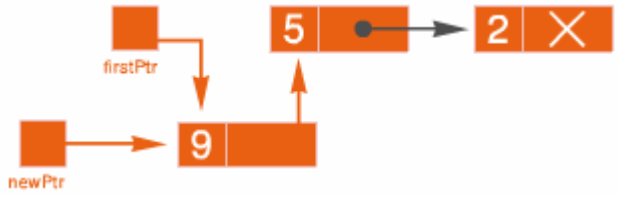
- استدعاء التابع getNewNode مع تمرير القيمة value له وهي القيمة التي نريد خزنها في العقدة الجديدة.

- يستخدم التابع getNewNode المؤثر new لإنشاء عقدة جديدة ويعيد المؤشر الذي يشير إلى هذه العقدة، فيسند هذا المؤشر إلى المتحول newPtr.
- إذا كانت اللائحة فارغة سابقاً فهذا يعني أن العقدة الجديدة هي الوحيدة الموجودة فيها الآن، وعليه يشير كلا المؤشرين firstPtr و lastPtr إلى هذه العقدة.
- إذا لم تكن اللائحة فارغة سابقاً، فهذا يعني أن العقدة الجديدة يجب أن توضع في البداية، وعليه نضع قيمة المؤشر firstPtr في مؤشر العقدة الجديدة، ثم نجعل firstPtr يشير إليها.

(A)



(B)



4- الإقحام في نهاية اللائحة insertAtBack:

إن المهمة الأساسية لهذا الإجراء هي إنشاء عقدة جديدة ووضعها في نهاية اللائحة.

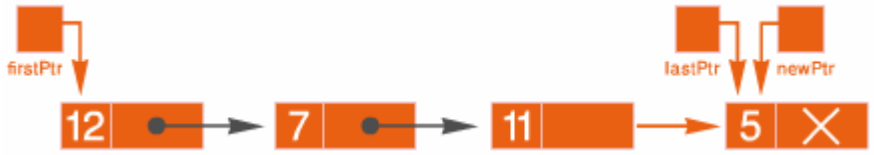
ينفذ الإجراء مهمته على مراحل هي كالتالي:

- استدعاء التابع getNewNode مع تمرير القيمة value له وهي القيمة التي نريد خزنها في العقدة الجديدة.

- يستخدم التابع getNewNode المؤثر new لإنشاء عقدة جديدة ويعيد المؤشر الذي يشير إلى هذه العقدة، فيسند هذا المؤشر إلى المتحول newPtr.
- إذا كانت اللائحة فارغة سابقاً فهذا يعني أن العقدة الجديدة هي الوحيدة الموجودة فيها الآن، وعليه يشير كلا المؤشرين firstPtr و lastPtr إلى هذه العقدة.
- إذا لم تكن اللائحة فارغة سابقاً، فهذا يعني أن العقدة الجديدة يجب أن توضع في النهاية، وعليه نضع قيمة المؤشر newPtr في مؤشر عقدة النهاية، ثم نجعل lastPtr يشير إلى هذه العقدة الجديدة التي أصبحت عقدة النهاية..

(A)

(B)



5- الحذف من بداية اللائحة removeFromFront:

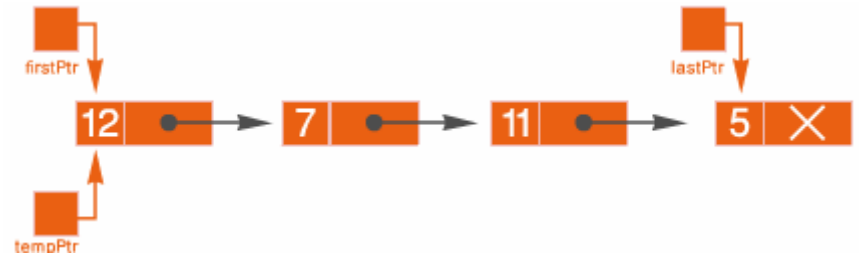
إن المهمة الأساسية لهذا الإجراء هي حذف العقدة الموجودة في البداية، وإسناد قيمة العنصر المحذوف إلى الموسط الممرر بالمرجع لهذا الإجراء. يعيد الإجراء القيمة false في حال جرت محاولة لحذف عنصر البداية من لائحة فارغة، في حين يعيد true عندما تنجح عملية الحذف.

ينفذ الإجراء مهمته على مراحل هي كالتالي:

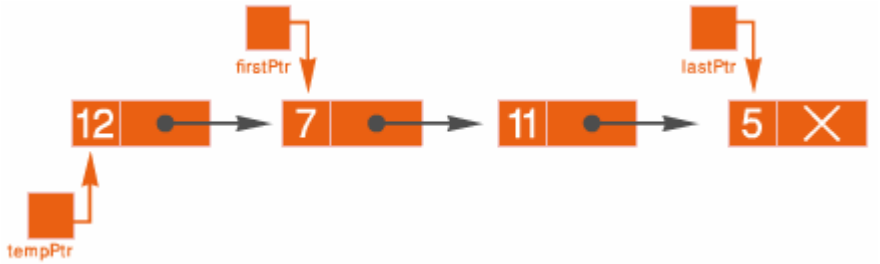
- يضع في المتحول tempPtr عنوان الذاكرة التي يشير إليها المؤشر firstPtr. إن الذاكرة التي يشير إليها tempPtr هي التي سيجري حذفها.

- في حال كان المؤشرين firstPtr و lastPtr متساويين أي أن اللائحة تتألف من عنصر وحيد، يجب وضع 0 في كل من firstPtr و lastPtr لإزالة العقدة من اللائحة.
- في حال كانت اللائحة تحتوي على عدد من العقد يزيد على 1 فإن الإجراء لا يقوم بأي تعديل على lastPtr في حين يجعل firstPtr يشير إلى العقدة التالية وذلك بوضع قيمة firstPtr->nextPtr في firstPtr وهكذا تصبح العقدة الثانية في اللائحة هي العقدة الأولى.
- يجري نسخ قيمة العقدة التي نعمل على حذفها في المتحول value.
- ننفذ تعليمة delete على المؤشر tempPtr الذي يشير إلى العقدة التي نعمل على حذفها، وذلك لإعادة الذاكرة المحجوزة.
- يعيد الإجراء القيمة true للدلالة على نجاح عملية الحذف.

(A)



(B)



6- الحذف من نهاية اللائحة removeFromBack:

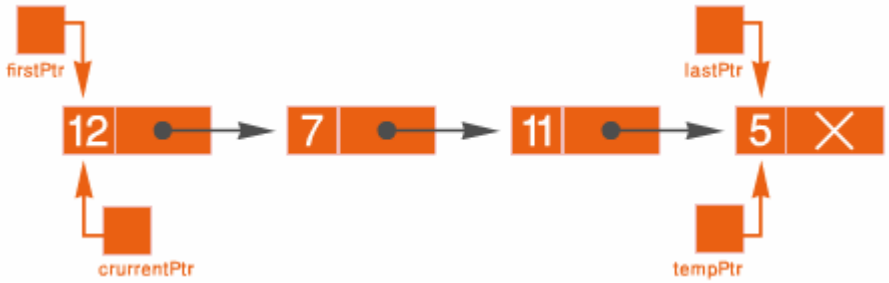
إن المهمة الأساسية لهذا الإجراء هي حذف العقدة الموجودة في النهاية، وإسناد قيمة العنصر المحذوف إلى الموسط الممرر بالمرجع لهذا الإجراء. يعيد الإجراء القيمة false في حال جرت محاولة لحذف عنصر النهاية من لائحة فارغة، في حين يعيد true عندما تنجح عملية الحذف.

ينفذ الإجراء مهمته على مراحل هي كالتالي:

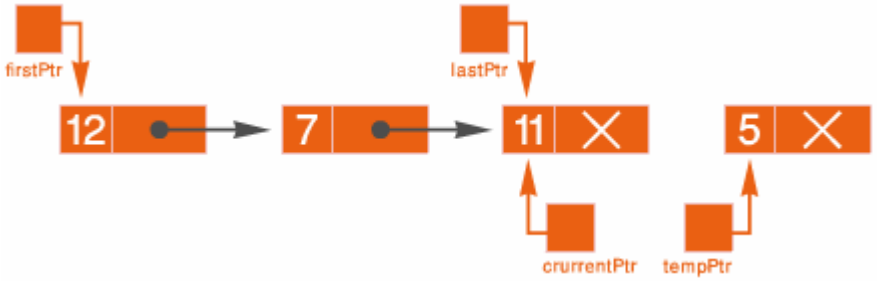
- يضع في المتحول tempPtr عنوان الذاكرة التي يشير إليها المؤشر lastPtr. إن الذاكرة التي يشير إليها tempPtr هي التي سيجري حذفها.

- في حال كان المؤشرين firstPtr و lastPtr متساويين أي أن اللائحة تتألف من عنصر وحيد، يجب وضع 0 في كل من firstPtr و lastPtr لإزالة العقدة من اللائحة.
- في حال كانت اللائحة تحتوي على عدد من العقد يزيد على 1 فإن الإجراء لا يقوم بأي تعديل على firstPtr في حين يجب أن يجعل lastPtr يشير إلى العقدة السابقة. وللوصول إلى العقدة السابقة عليه أن يستخدم مؤشراً (currentPtr) يبدأ بالتأشير على أول عقدة أي فيه قيمة firstPtr.
- يمر currentPtr على جميع عقد اللائحة، حتى يصل إلى العقدة قبل الأخيرة، وذلك بتنفيذ حلقة تضع currentPtr->nextPtr في currentPtr مادام currentPtr->nextPtr لايساوي lastPtr.
- نضع قيمة currentPtr في lastPtr لحذف العقدة الأخيرة من اللائحة.
- نضع القيمة 0 في currentPtr->nextPtr في العقدة الأخيرة الجديدة.
- يجري نسخ قيمة العقدة التي نعمل على حذفها في المتحول value.
- ننفذ تعليمة delete على المؤشر tempPtr الذي يشير إلى العقدة التي نعمل على حذفها.
- يعيد الإجراء القيمة true للدلالة على نجاح عملية الحذف.

(A)



(B)



7- طباعة اللائحة

يفحص هذا الإجراء اللائحة لمعرفة إذا كانت فارغة أم لا. فإن كانت فارغة يظهر على الشاشة جملة موافقة ويعود، وإلا يمر على جميع العقد ويطبع حقل القيمة في كل عقدة. للمرور على كافة العقد يستخدم مؤشر مساعد هو currentPtr الذي يشير بدايةً إلى العقدة الأولى (قيمته الابتدائية = firstPtr)، ومادام currentPtr لايساوي 0 يطبع currentPtr->data ثم ينتقل للعقدة التالية بوضع currentPtr->nextPtr في currentPtr.

البرنامج الرئيسي

يقوم البرنامج التالي باستدعاء التابع testList الذي يأخذ كوسيط لائحة ويظهر على الشاشة مجموعة خيارات (الإقحام والحذف في بداية اللائحة ونهايتها، والخروج)، وبعد كل خيار يطبع قيم عقد اللائحة:

```
//List class test program.
#include <iostream>
#include <string>
#include "List.H" // List class definition
using namespace std;
void instructions();
// function to test a List
void testList( List &listObject)
{
   cout << "Testing a List  \n";
   instructions(); // display instructions
   int choice; // store user choice
   int value; // store input value
   do // perform user-selected actions
{
      Cout<<"? ";
      cin >> choice;
      switch ( choice )
     {
         case 1: // insert at beginning
            cout << "Enter an integer: ";
            cin >> value;
            listObject.insertAtFront( value );
            listObject.print();
```

```
            break;
         case 2: // insert at end
            cout << "Enter an integer :";
            cin >> value;
            listObject.insertAtBack( value );
            listObject.print();
            break;
         case 3: // remove from beginning
            if ( listObject.removeFromFront( value ))
               cout << value << " removed from list\n";
            listObject.print();
            break;
         case 4: // remove from end
            if ( listObject.removeFromBack( value ))
               cout << value << " removed from list\n";
            listObject.print();
            break;
      } //end switch
   } while ( choice != 5 ); // end do...while
   cout << "End list test\n\n";
}  // end function testList
// display program instructions to user
void instructions()
{
   cout << "Enter one of the following:\n"
<<"  1 to insert at beginning of list\n "
<<"  2  to insert at end of list\n "
<<"  3  to delete from beginning of list\n "
<<"  4 to delete from end of list\n "
<<"  5 to end list processing\n”;
}  // end function instructions
int main()
{
   // test List of int values
   List integerList;
   testList( integerList );

   return 0;
}// end main
```

التقويم:

لابد أنك لاحظت في تعريف النمط List أننا استخدمنا مؤشرين firstPtr و lastPtr. وإذا أمعنت النظر يمكن أن يتبين لك أننا نستطيع تنفيذ جميع الوظائف السابقة دون وجود المؤشر lastPtr . والمطلوب:

- إعادة تعريف الصف List الذي يتألف من عضو معطيات واحد هو front الذي يشير إلى البداية.
- تزويد الصف بتابع بحث search عن وجود قيمة يعطيها المستخدم ضمن عقد اللائحة، ويعيد true إذا كانت موجودة و false إن لم تكن موجودة.
- تزويد الصف بتابع حذف remove عقدة في اللائحة إذا كانت تحتوي على قيمة يعطيها المستخدم، أياً كان موقعها (بداية اللائحة أو نهايتها أو داخل اللائحة)

الإجابة:

```
#include <iostream>
using namespace std;

class Node{
  friend class List;
public:
     Node(int, Node* n = 0);
 private ;
  int data;
  Node* nextPtr;
};

class List{
public :
  List();
~ List();
  void insertAtFront(const int &);
```

```
  bool remove(int );
  bool removeFromFront(int &);
  void print();
  bool isEmpty() const;
  bool search(int);
private:
  Node* front;
};
Node::Node(int x ,Node* n){
  data = x;
  nextPtr = n;
}

 List::List(){
  front = 0;
};

bool List::isEmpty() const {
  return front == 0 ? true : false;
}
List::~List()
{   Node* tmp = front;
  while (tmp)
     {      front = tmp->nextPtr;
           delete tmp;
              tmp = front;
     }
 }

void List::insertAtFront(const int &x)
{   Node* p = new Node(x,front);
    front = p};
bool List::search(int x)
{
  if (front == 0) return false;
  Node* p = front;
  bool found = false;
  while ((p) && (!found))
    if (p->data == x)
        found = true;
    else
        p = p->nextPtr;
  return found;
}

void List::print()
 {
  cout <<"( ";
```

```
  for (Node* p = front; p; p = p->nextPtr) cout << p->data<<" ";
  cout << ")" << endl;
}
bool List::removeFromFront(int &x)
{
  if (front == 0)
     return false;
  else {
         x=front->data;
            Node* p = front;
         front = front->nextPtr;
         delete p;
            return true;
           }
}
 bool List::remove(int x)
 {
  if (front == 0){
    cout << "empty List !" << endl;
    return false;
 }
 // research of the element to remove
 Node* p = front;
 Node* pred = 0;
 bool found = false;
 while ((p) && (!found))
   if (p->data == x)
       found = true;
   else {
       pred = p;
       p = p->nextPtr;
  }
 if (!found){
   cout <<  x << " does not exist in the list !" << endl;
   return false;
}
 else { // remove the found element, pointed by p
   if (pred) { // the element to remove has a predecessor
(pred->nextPtr) = (p->nextPtr);
     delete p;
       return true;
  }
   else {      // the element to remove is the first in the list
     front = (p->nextPtr);
     delete p;
       return true;
  }
 }
```

```
}
void main()
{
   List l;
   int i;

   cout << "Enter integers, 0 to finish : " << flush;
   cin >> i;
   while (i)
{
     l.insertAtFront(i);
     cin >> i;
   }
   if (!(l.isEmpty()))
      l.print();
   else
     cout << "empty list !" << endl;
   cout << "Research in the list, 0 to finish : " << flush;
   cin >> i;
   while (i){
     if (l.search(i))
     cout << i << " exist in the list" << endl;
     else
     cout << i << " does not exist in the list" << endl;
     cout << "Research in the list, 0 to finish : " << flush;
     cin >> i;
   }

   cout << "Elimination from the list, 0 to finish : " << flush;
   cin >> i;
   while (i){
     l.remove(i);
     l.print();
     cout << "Elimination from the list, 0 to finish : " << flush;
     cin >> i;
   }
}
```

ملاحظة:

لابد أنك لاحظت أن باني العقدة في هذا البرنامج يضع قيمة في المؤشر nextPtr هي عنوان لذاكرة تمرر أثناء البناء في حين كان الباني في البرنامج السابق يضع 0 في المؤشر.

تقويم:

لتكن لدينا قائمة بأسماء الطلاب في الصف ومعدلاتهم مرتبة أبجدياً. نريد إدخال هذه القائمة إلى الحاسوب في بنية ديناميكية هي اللائحة المترابطة. تقرأ المعلومات بالترتيب الأبجدي لأسماء الطلاب ولكن إدخال المعلومات إلى اللائحة يتم بحيث يراعى الترتيب التنازلي للمعدلات.

والمطلوب: التصريح عن الصف StudentList الذي يتألف من عقد تحتوي كل منها على اسم الطالب، ومعدله ومؤشر على العقدة التالية، وكتابة التوابع الأعضاء التالية:

- إقحام عقدة الطالب في الموضع المناسب لترتيب المعدل
- حذف عقدة طالب يعطى اسمه
- البحث عن معدل طالب يعطى اسمه
- طباعة اللائحة

كما يطلب كتابة برنامج رئيسي يبين استدعاء هذه التوابع.

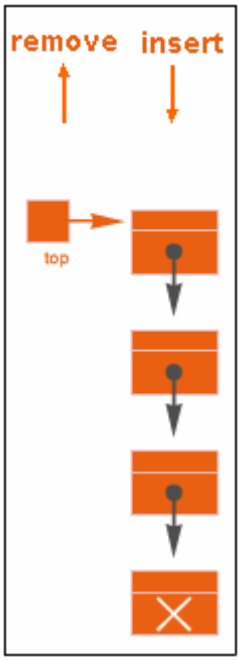
5. بنى المعطيات المتقدمة

بعد أن تعرفت إلى مفهوم اللائحة، فإنه يمكن صنع بنى معطيات عديدة تستند في بنيتها التخزينية على اللائحة، ويختلف بعضها عن بعض بطريقة إدارتها، ومنها المكدسات والأرتال والأشجار.

المكدس

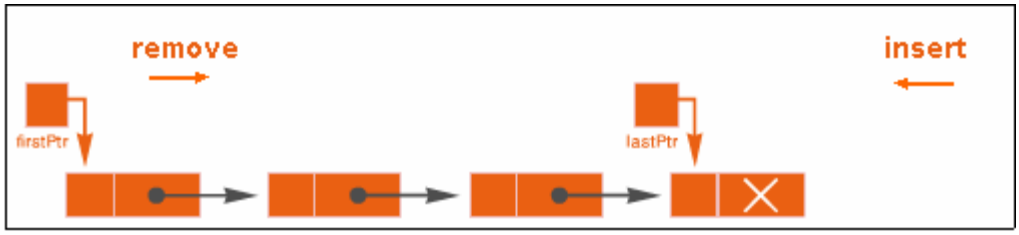
بنية معطيات نقحم العناصر فيها في بدايتها وننزعها من بدايتها فقط. لفهم هذه البنية تذكر كيف نكدس الأطباق بعضها فوق بعض، وعندما نضع طبقاً نضعه في أعلى المكدس وعندما نحتاج إلى سحب طبق نسحبه من الأعلى كذلك. نستخدم بنية معطيات هي لائحة لكن التوابع الأعضاء لاتتضمن سوى توابع تتعامل مع الأعلى (البداية) فقط (لا يوجد مؤشر نهاية، ولا توابع إقحام أو حذف في أي مكان سوى أعلى المكدس). يسمى أسلوب الإدارة هذا

(الداخل آخراً يخرج أولاً (LIFO) last in first out)



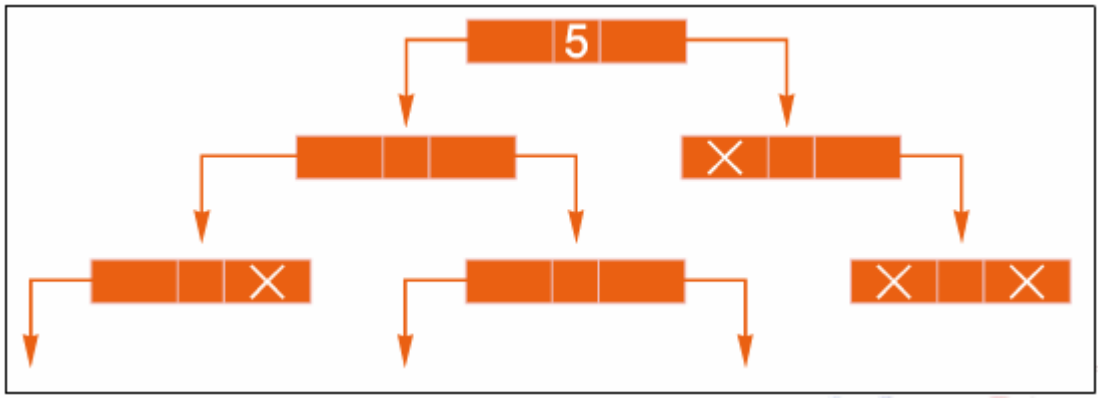
الرتل

بنية معطيات نقحم العناصر فيها في نهايتها وننزعها من بدايتها. لفهم هذه البنية تذكر كيف يقف الزبائن أمام صندوق الدفع في محل تجاري، وكلما أتى زبون يقف في آخر الرتل، في حين يخدم الزبون الذي يقف في أول الرتل. نستخدم بنية معطيات هي لائحة لكن التوابع الأعضاء لاتتضمن سوى توابع الإقحام في النهاية والحذف من البداية (يوجد مؤشران للبداية والنهاية. يسمى أسلوب الإدارة هذا (الداخل أولاً يخرج أولاً (FIFO) first in first out)



الشجرة

بنية معطيات غير خطية، تتألف من عقد، في كل عقدة توجد مؤشرات للعقد الفروع. للتبسيط، ولفهم المبدأ العام، لنأخذ بنية ثنائية البعد. تمثل هذه البنية شجرة ثنائية تتألف من عقد، تحتوي كل عقدة على مؤشرين أحدهما يشير إلى عقدة الفرع الأيمن والثاني يشير إلى عقدة الفرع الأيسر. لهذه البنية استخدامات عديدة ستراها لاحقاً في مقرر الخوارزميات وبنى المعطيات المتقدمة.



إن الدخول في تفاصيل بناء هذه البنى (المكدس، الرتل، الشجرة) واستخداماتها يحتاج إلى فصول. وسيخصص لهذا الغرض مقررات لاحقة تعنى ببنى المعطيات المتقدمة والخوارزميات المبنية عليها، كما يحتاج الدخول في بعض الحالات إلى مفاهيم برمجية غرضية التوجه لم تسنح الفرصة لعرضها هنا كالوراثة والربط الديناميكي. وسيفرد مقرر كامل لهذه المفاهيم البرمجية الهامة في البرمجة الغرضية التوجه.