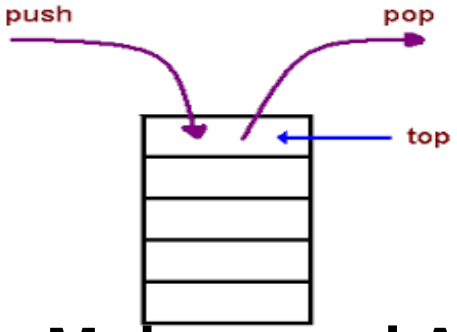
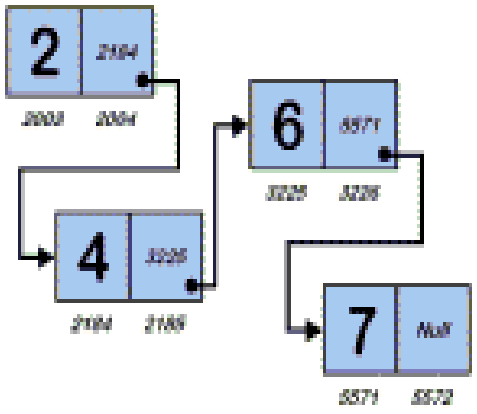
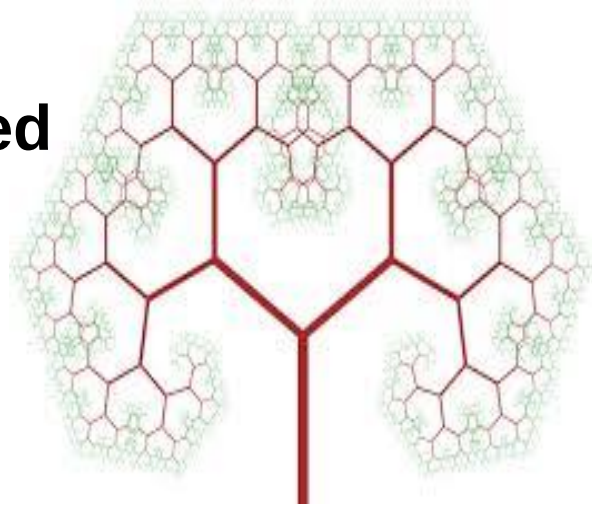
جامعة حماة
الكلية التطبيقية
قسم تقنيات الحاسوب
الفصل الثاني



- المقرر: الخوارزميات وبنى المعطيات
- المحاضرة :السابعة

Dr. Mohammed AL-Mohammed





الجامعة العربية الخاصة للعلوم والتكنولوجيا AUST

كلية الهندسة المعلوماتية Engineering Faculty

قسم تقانة المعلومات IT

- المقرر: الخوارزميات وبنى المعطيات (١)
- المحاضرة : السابعة

Dr. Mohammed AL-Mohammed

Chapter 7

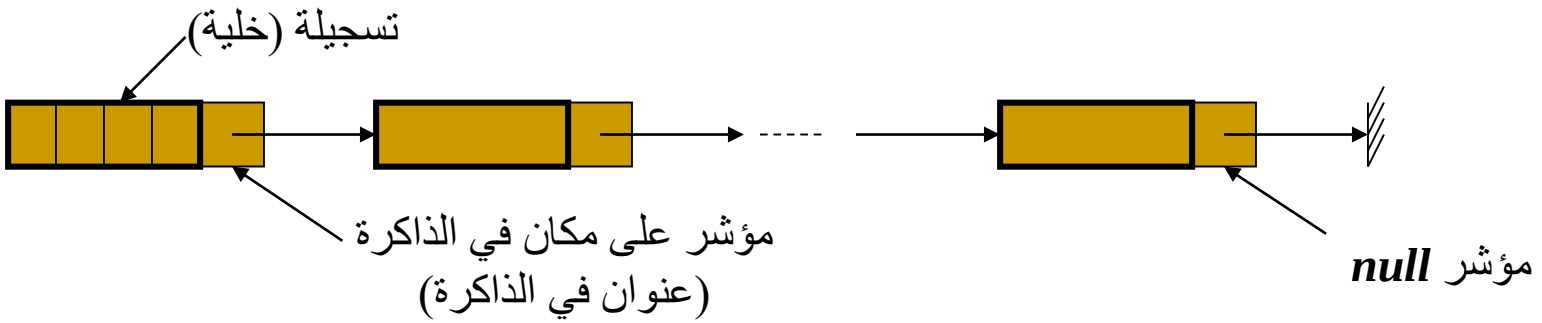
بنى المعطيات

Data structure

Linked list القوائم المترابطة

التنفيذ الديناميكي-اللائحة المترابطة

- لهذه الطريقة السمات التالية:
  - مؤلفة من متتالية من التسجيلات records ليست متجاورة بالضرورة في الذاكرة.
  - تتضمن كل تسجيلة العنصرَ ومؤشراً على تسجيلة تتضمن العنصر التالي (successor) ويسمى مؤشر next.
  - مؤشر التسجيلة الأخيرة يتضمن قيمة مؤشر خاصة وهي null.
  - تتسع وتتقلص في زمن التنفيذ دون هدر في حجم الذاكرة.



القوائم المترابطة *linked list*

كما علمنا أن على المصفوفة عدة قيود أهمها حاجتها لـ *block* كامل ومتسلسل في الذاكرة من أجل حجزها.

وإن من عيوب اللائحة باستخدامها كمصفوفة أننا نقوم منذ البداية بحجز حجم معين للائحة في الذاكرة سواء قمنا باستخدامها كلها أم لا.

لذلك اجتهد العلماء لإيجاد صيغ لبنى البيانات بحجم كبير مع فراغات متسلسلة أقل

⇐ قوائم مترابطة *linked list*

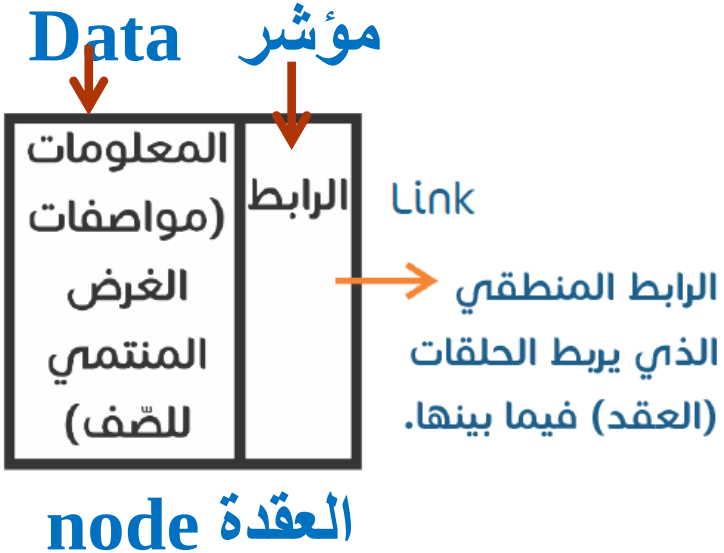
- القوائم المترابطة:

هي سلسلة من الأغراض التي تتبع لصف معين أو نمط بيانات معين وفق ترابط منطقي ما.

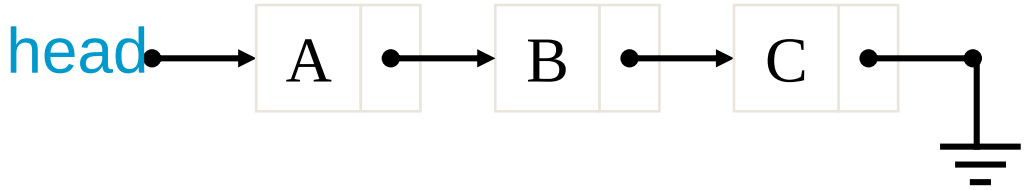
يتم فصل العناصر فيزئياً بحيث يقسم كل عنصر لحقلين $(data - Link)$ بحيث يقوم كل حد بالإشارة إلى الحد الذي يليه زيحتوي على عنوانه وبذلك نستطيع الوصول إلى أي حد بسهولة لأن عنوانه في الذاكرة مسجل في الحد الذي يسبقه لذلك نستخدم المؤشرات في اللوائح المترابطة.

$$int * aptr = \&a;$$

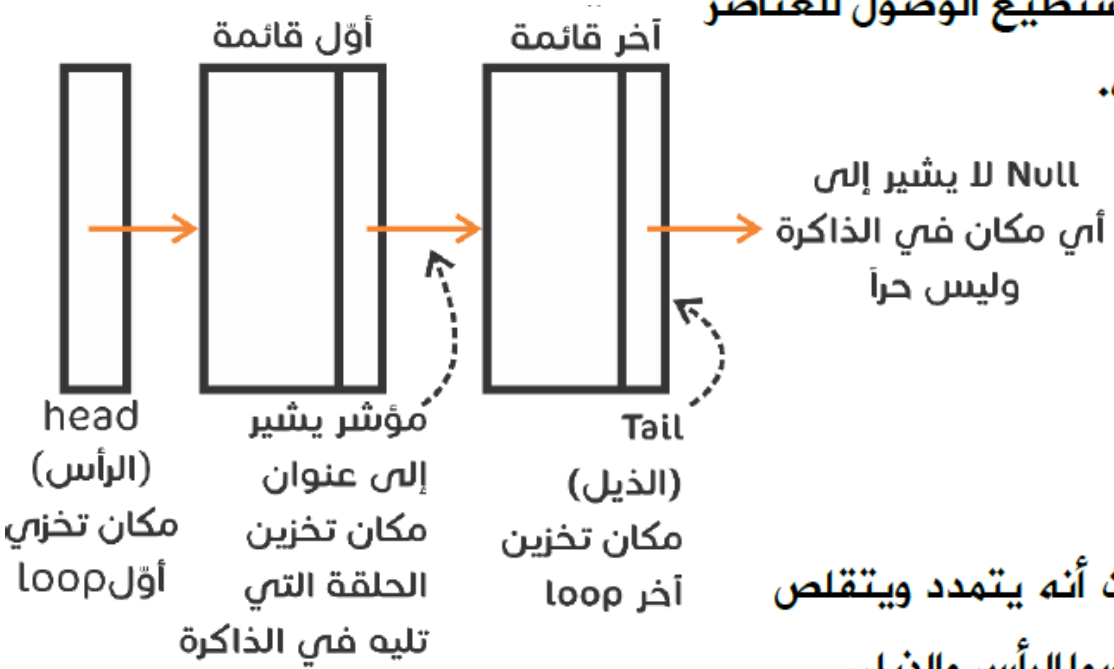
هذا يعني أن المتغير $aptr$ يحوي على عنوان المتغير a وبالتالي نتعامل مع المتغير إما عن طريق اسمه أي a أو عن طريق عنوانه $\&a$.



الرابط: هو مؤشر ***pointer*** يشير إلى قيمة عنوان ما في الذاكرة أو مرجع يستخدم كمؤشر بطريقة مخفيّة.



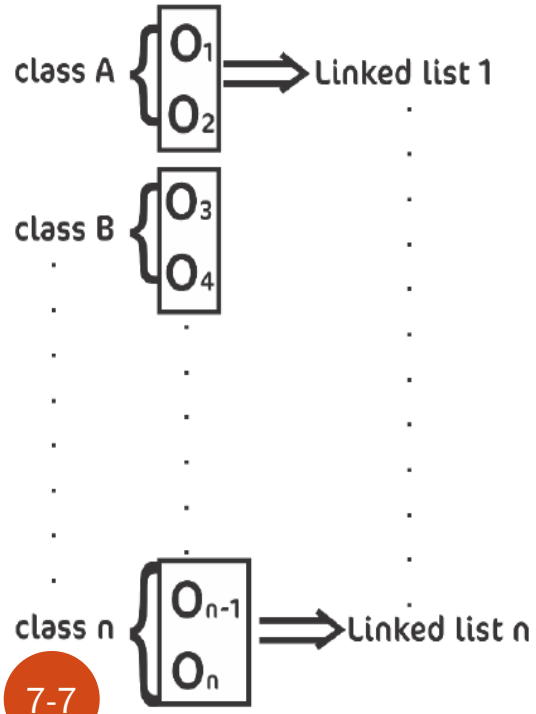
⇐ لوكان عندنا ***loop*** في قائمة مترابطة نستطيع الوصول للعناصر بأي وقت لأنها مترابطة بواسطة المؤشرات.



يمكن تشبيه القائمة المترابطة بنابض بحيث أنه يتمدد ويتقلص بحسب إضافة ***loops*** أو حذفها ⇐ لايهمنا سوا الرأس والذيل.

ملاحظة

- ✓ لا يوجد تسلسل في تخزين العناوين بالذاكرة لقائمة مترابطة ما (ما عدا المصفوفات) ⇐ كل ***loop*** من القائمة موجودة في مكان ما من الذاكرة ⇐ لانحتاج إلى ***block*** كامل متسلسل في الذاكرة ⇐ خففنا من ظاهرة التشكل الحالي لأننا نستطيع القفز فوراً إلى عنوان الحلقة التالية عبر قيم ***pointers*** الغير مرتبة والغير متسلسلة أيضاً.
- ✓ المقدمة والنهاية (الرأس والذيل) لا يحويان معلومات من القائمة المترابطة فلا يهمنا إلا المعلومات في العناصر التي بينهما، مثل قطار مثلاً لا يهمّنا قاطرة السائق ولا القاطرة الخلفية إنّما قاطرات الركاب في المنتصف.
- ✓ يفضل أن تنتمي الأغراض ***objects*** في اللائحة الواحدة لنفس الصف ***class*** لماذا؟



لأن الرابط بينهم عبارة عن مؤشرات لا علاقة لبعضهم ببعض

⇐ ***link*** لايتعلق بالصف ***class*** ونحن سنأخذ بعين الاعتبار العمليات الأساسية كالإضافة والحذف والوصول،

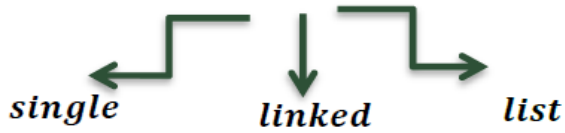
وبالتالي سيكون الجهد الذي سنبذله عند أخذ كل غرض من صف ما أكبر وبالتالي الفائدة أقل

⇐ سنجمع الأغراض التي نحتاجها في بنية ما ونربطهم برابط منطقي معين.

وهنا سنحتاج إلى صفين:

1) صف العقدة ***Node***:

اسم الصف ***sLLNode***



2) صف القائمة المترابطة:

اسم الصف ***sLList***

إن ***structure*** تتضمن المعلومات فقط وليس الأعضاء ***function***.

```
class sLLNode
{
Private:
int info;    (مواصفات الغرض وهي تعريف بمجموعة بيانات)
sLLNode*next;(الرابط المنطقي)
```

يجب أن يكون الرابط من نفس نوع البيانات المخزنة

⇐ يجب أن يكون ***object*** من هذا الصف ***class***.

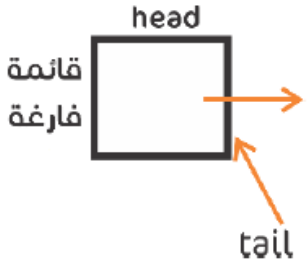
```
  public:
  sLLNode ( ) {    باني افتراضي
  info=0;
  next=0;   ≡ Null     (الموشر صفر)
    }
   sLLNode (int e1) {    باني عادي
   info=e1;
   next=null;}
   };
```

```
Class sLList
{
Private:
sLLNode *head, tail;
public:
sLL ist()

{head=tail=null;}
إضافة
حذف
وصول

    }};
```

يحوي بواني و أعضاء أولية (عمليات أساسية)



توصيف العمليات

```
RDT FunName (الباراميترات);
```

أي تحديد القيمة المعادة واسم الدالة وباراميتراتها حتى تقوم بالتنفيذ إذا فهو يتضمن مناقشة للأسماء وثم الباراميترات ..... إلخ.

عملية الإضافة

سنضيف ***loop*** ما وستكون خيارات الإعادة بعد الإضافة حسب الدالة:

void: لاتعيد للدالة أي شيء.

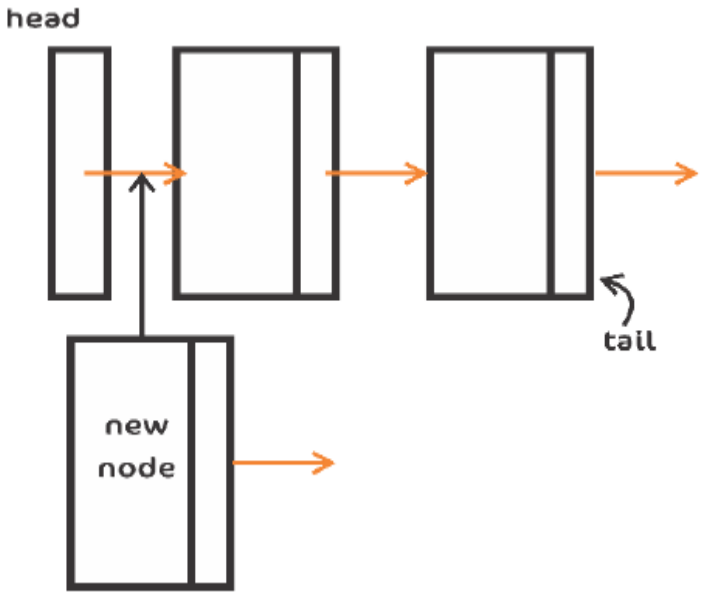
bool: تعيد للدالة صفر أو واحد.

⇐ هذا يشير إلى تمام عملية الإضافة.

ماهي المعلومات التي تحتاجها الدالة حتى تنفذ الإضافة؟

1- معرفة العقدة ***Node*** التي سنضيفها لذلك نحتاج قيمها (قيم ومواصفات الغرض المضاف)

2- مكان الإضافة: أين سأضيف؟

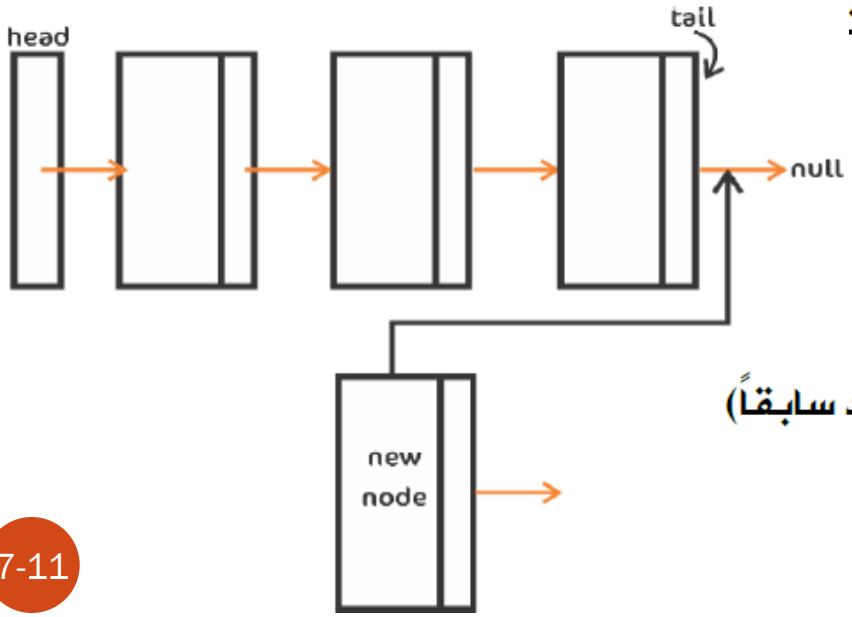


أ- في بداية القائمة: ***Add TO head*** أي بعد الرأس مباشرة.

1. ننشئ عقدة جديدة، أي نحجز في الذاكرة مكان لعقدة جديد
2. نسخ قيمة المتغير للعقدة الجديدة.
3. نسخ الرابط الخارجي في الحقل ***Link*** للعقدة الجديدة.
4. نغير مكان الرابط الخارجي إلى نقطة في العقدة الجديدة.

ب- في نهاية القائمة: ***Add to tail*** أي بعد آخر ***Node***.

1. ننشئ عقدة جديدة، أي نحجز في الذاكرة مكان لعقدة جديدة.
2. نسخ قيمة المتغير للعقدة الجديدة.
3. تغيير حقل الـ***next*** في آخر عقدة وجعلها تؤشر على القعدة الجديدة.
4. جعل الحقل ***next*** في العقدة الجديدة يدل على أن آخر عقدة في اللائحة.
5. جعل المؤشر ***tail*** يؤشر على العقدة الجديدة.



ج- في وسط القائمة: ***Add*** في أي مكان أرغب فيه (محدد سابقاً) بين الرأس ***head*** والذيل ***tail***.

ولو كانت القائمة مرتبة يجب أن نحافظ على الترتيب.

عملية الحذف

ماهي المعلومات اللازمة لتنفيذ الحذف؟

1- فحص لمعرفة فيما إذا كانت المصفوفة فارغة

⇐ لا معنى للحذف أصلاً، أو إذا كانت مؤلفة من عقدة واحدة

⇐ $head = tail$.

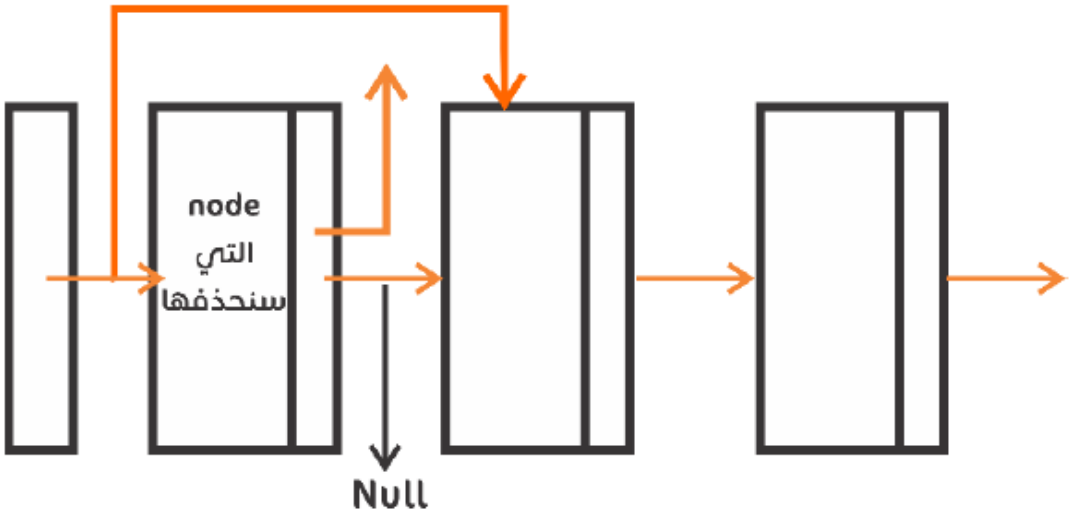
ملاحظة: لا معنى للقائمة $Full$ في بنية اللوائح المترابطة لأنها تشبه الراصور (نابض) بالإمكان الحذف منها والإضافة إليها متى شئنا، ولكن بإمكاننا تحديد الحجم الأعظمي لها وذلك من أجل حل بعض المشاكل في البرامج.

2- مكان الحذف: من أين سأحذف؟

أ- من بداية القائمة: $delete\ from\ head$

ب- من نهاية القائمة: $delete\ from\ tail$

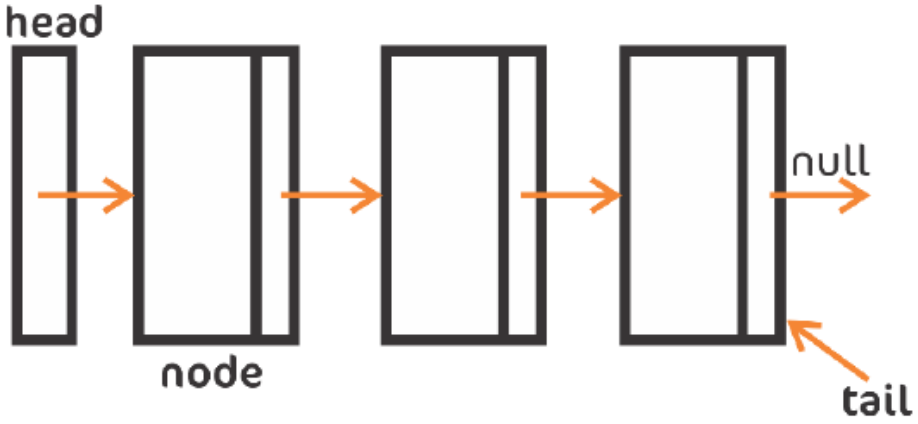
ت- من وسط القائمة: $delete$



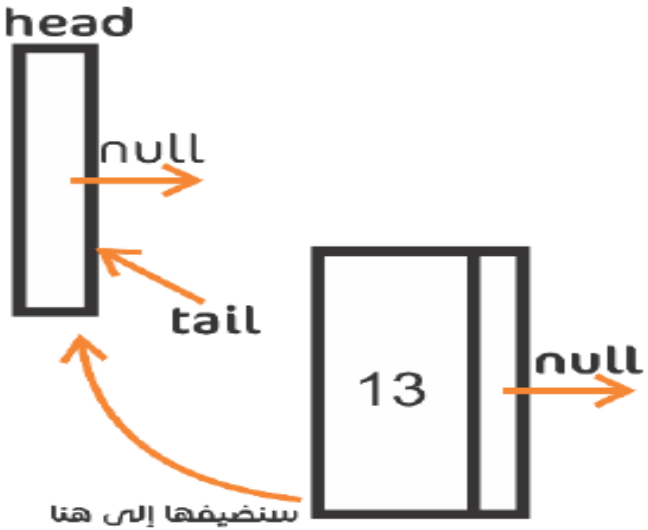
ملاحظة:

- ✓ سندرس فيما إذا كانت اللائحة مترابطة في حال الإضافة والحذف.
- ✓ عملية الإضافة ليست إلا تغييراً في المؤشرات فقط وكذلك الحذف.

العمليات على القوائم المرتبطة

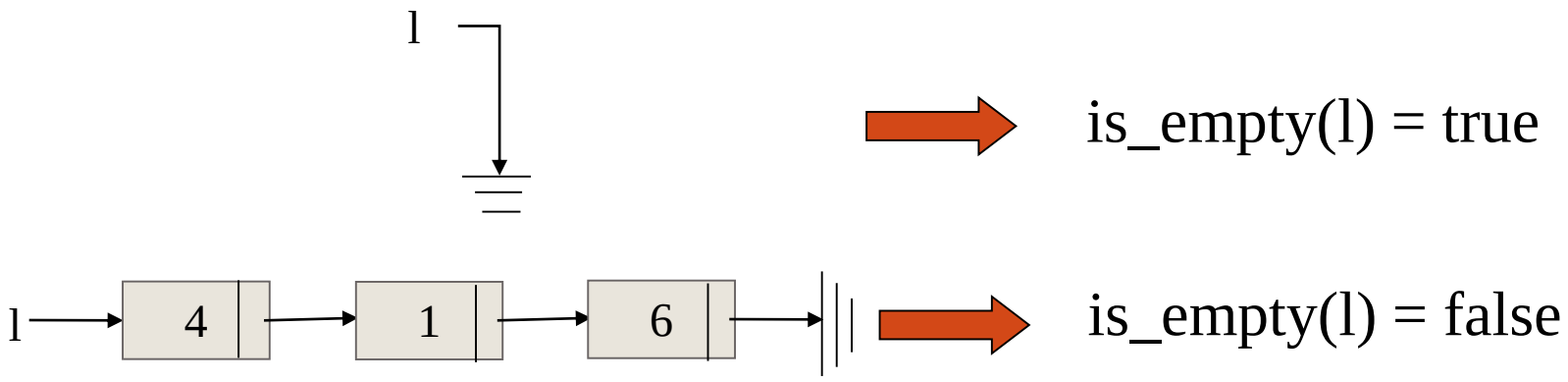


✓ القائمة فارغة: أي لايوجد أية $Nodes$ ⇐ يوجد قائمة ولكنها لاتحوي عناصر⇐ $Head = Tail = Null$



اللائحة المترابطة (Is_empty)

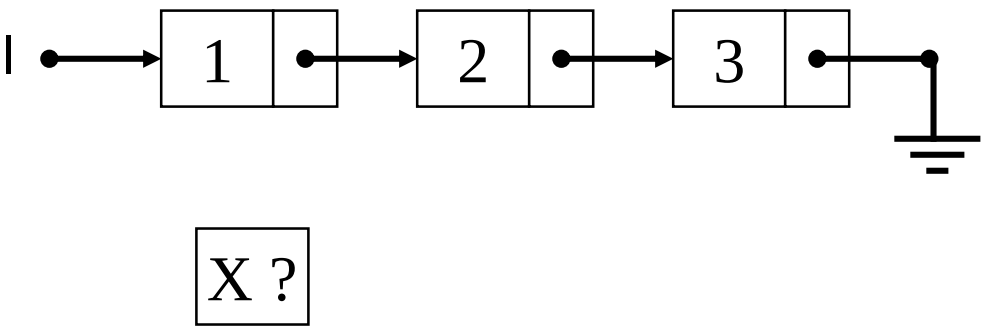
is_empty(l): تابع يتحقق كون اللائحة المعطاة فارغة أم لا.




```
bool is_empty(List l)
   { return l == 0 ? true : false; }
```

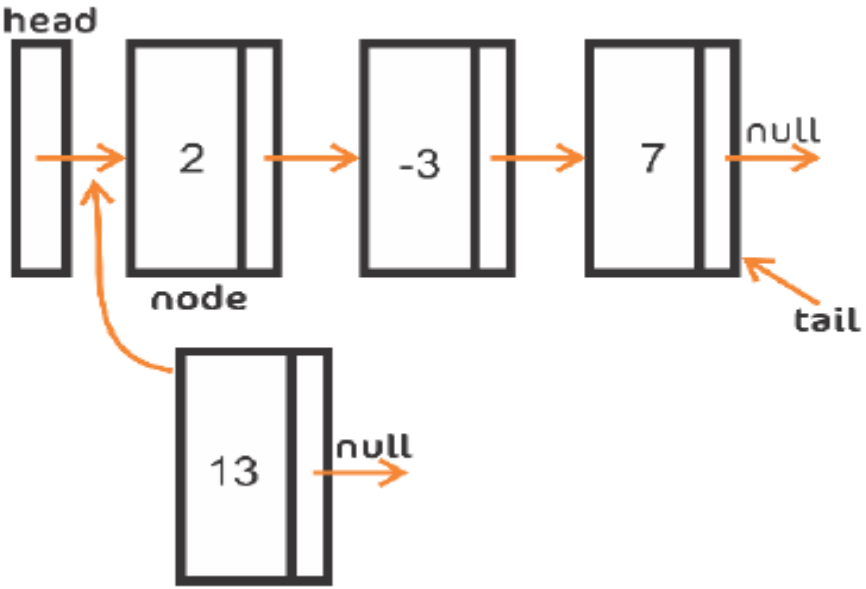
اللائحة المترابطة (exist)

exist(l,x): تابع يتحقق كون العنصر x موجوداً في اللائحة أم لا.




```
bool exist(List l, Type x)
{
  if (l == 0)
       return false;
  Item* p = l;
  bool found = false;
  while ((p) && (!found))
    if (p->val == x)
        found = true;
    else
        p = p->next;
  return found;
}
```

✓ القائمة غير فارغة: يوجد عناصر



⇐في الحالتين سنضيف العقدة الجديدة بعد الرأس $head$ أي أننا سندخلها ضمن الارتباط المنطقي للائحة.

خطوات الإضافة

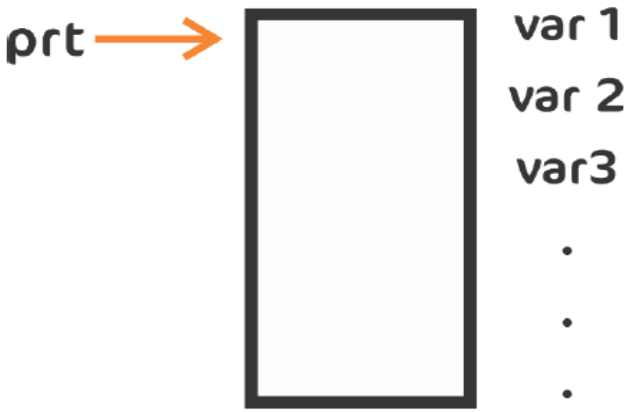
1) إنشاء عقدة جديدة $new\ Node$،
وذلك عبر استدعاء أحد بواني الصف $sLLNode$.

الوصول إلى 2 : $head \rightarrow next$

الوصول إلى -3 : $head \rightarrow next \rightarrow next$

الوصول إلى 7 : $head \rightarrow next \rightarrow next \rightarrow next$

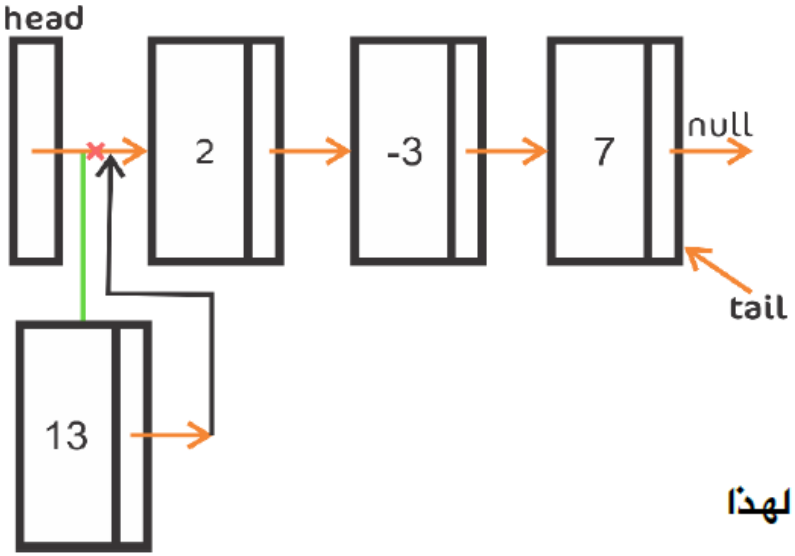
الوصول إلى Null: $head \rightarrow next \rightarrow next \rightarrow next \rightarrow next$




```
prt -> var 1;
prt -> var 2;
```

2) تغيير قيم المؤشرات:

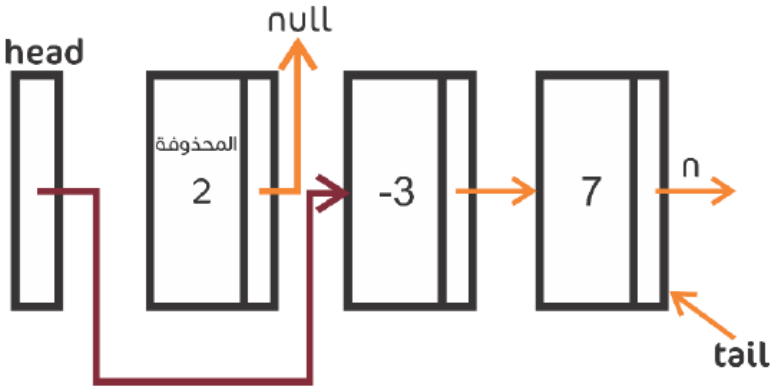
```
tail→next.
sLLNode
new Node=new sLLNode(13);
```



Delete:

إن الحذف فقط يحرر القيم من الذاكرة وهو الاستخدام الأخير لهذا الـ Node في البرنامج فإمّا أن يعيد خاصة من الخصائص أو قيمة ما فهي تبقى موجودة إلى أن نكتب فوقها:

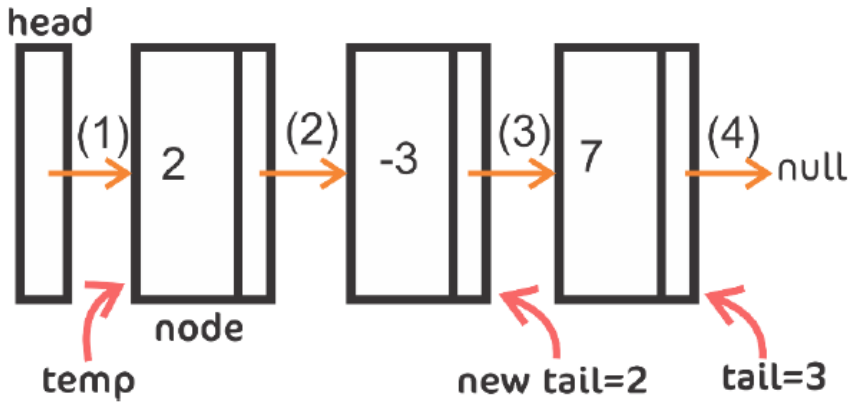
```
int delete from head ( )
{
if (head=tail)
return −1;
else
head=head->next;}
```



ملاحظة:

إن المؤشر $next$ للعنصر المحذوف يجب أن يشير إلى $Null$ رغم أنها محررة من الذاكرة ولكن قد يكون هذا نقطة ضعف في البرنامج لأن قيمة المؤشر للـ $Node$ المحذوفة ستدلنا على كل اللائحة ⇐ قد تستغل ذلك برامج التجسس فتصل إلى معلومات مهمة.

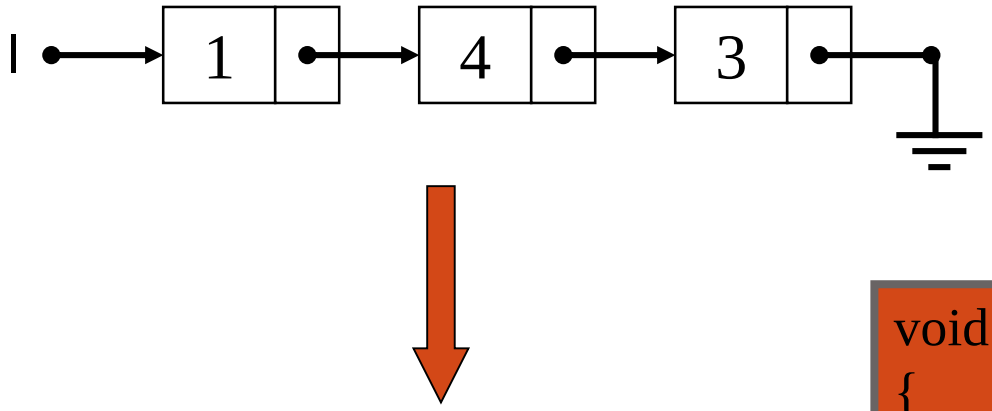
```
int delete from tail ( )
{
sLLNode*Temp=head;
while (Temp->next!=Null)
{
Temp=Temp->next;
}
```



- في البداية كان $Tail \equiv 3$ وبعد عملية $Delete\ from\ tail$ أصبح $Tail \equiv 2$.

اللائحة المترابطة (display)

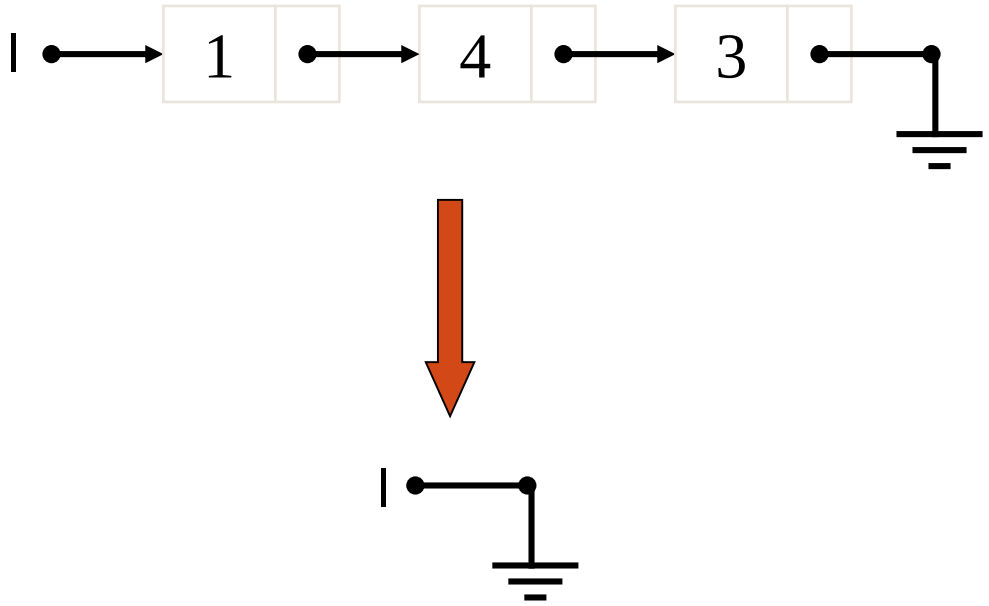
display(l): تابع يطبع اللائحة l.




```
void display(List l)
{
  cout<<"(  ";
  for (Item* p = l; p; p = p->next)
    cout << p->val <<“ ”;
  cout <<")";
}
```

اللائحة المترابطة (destruct)

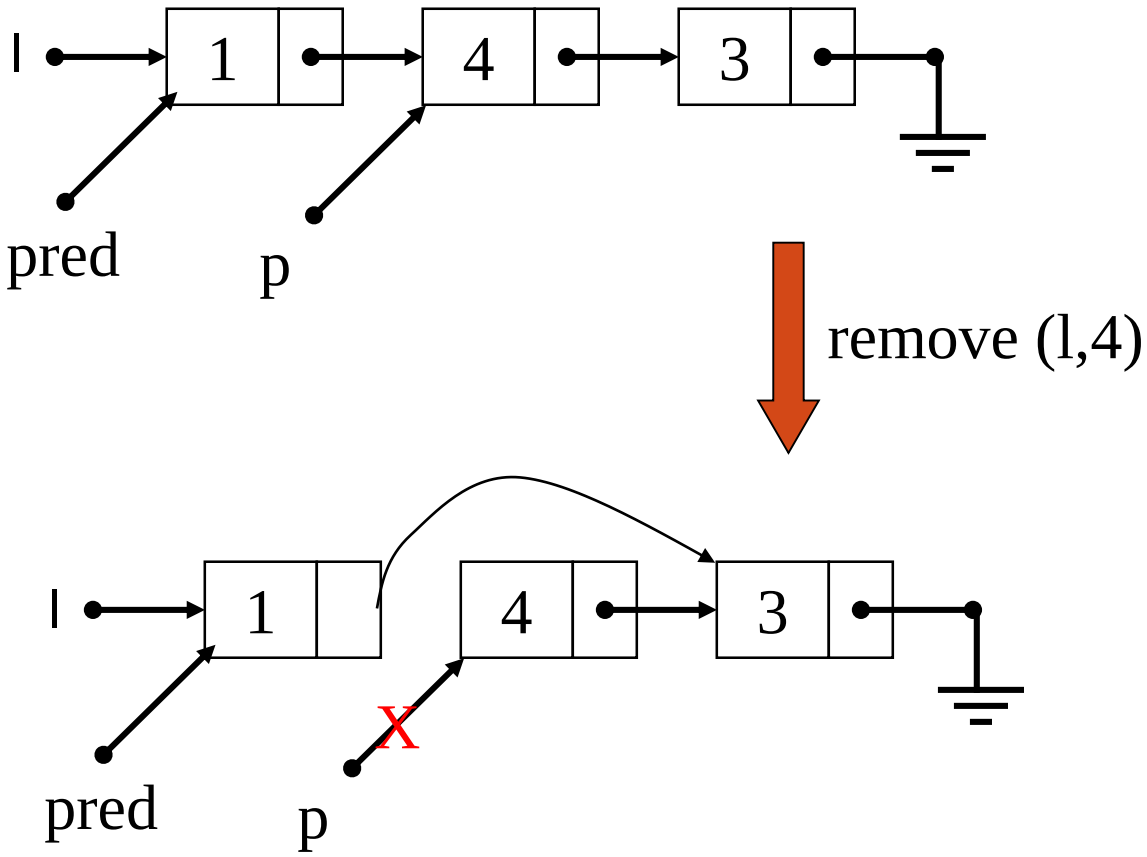
destruct(l): تابع يهدم اللائحة المعطاة ويعيد الذاكرة المستخدمة إلى النظام.




```
void destruct(List &l)
{
  List p = l;
  List tmp = l;
  while (tmp)
  {
     tmp = p->next;
     delete p;
     p = tmp;
  }
  l = 0;
}
```

اللائحة المترابطة (remove)

remove(l,x): تابع يحذف العنصر x من اللائحة المعطاة.



Linked List اللائحة المترابطة

```
typedef int Type;
struct Item {
  Type val;
  Item* next;
   };
typedef Item* List;
enum bool { false = 0, true };

void init(List&);                  // initialize the  list
void insert(List&, Type);          // insert one element in the head of the list
void display(List);                // displays the list
void destruct(List&);              // destructs the list
void remove(List&, Type);          // removes x from the  list
bool is_empty(List);
bool exist(List, Type);
```

تنجيز اللائحة المترابطة – الحذف

```
void Remove(List &l, Type x)
{ if (l == 0) {
          cout << "Empty list!“ << endl;
           return; }
  // search for the element to delete
  List p = l;
  List pred = 0;
  bool found = false;
  while ((p) && (!found))
    if (p->val == x)
       found = true;
    else {
      pred = p;
      p = p->next; }
  if (!found) {
    cout << "The element << x <<  is not in the list! “<< endl;
    return;}
  else { // delete the found element pointed by p
    if (pred) { // the element to be deleted has  a predecessor
      (pred->next) = (p->next);
      delete p;}
    else { // the element to be deleted is the first  of the list
      l = (p->next);
      delete p; } } }
```

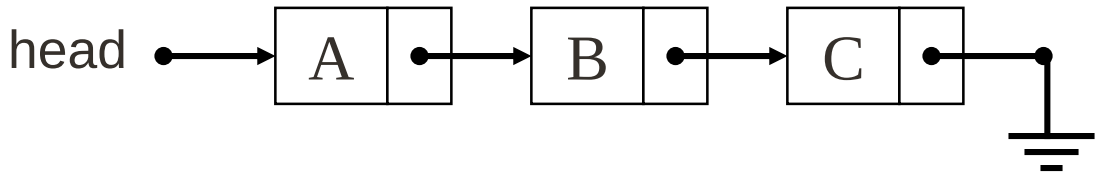
كيف يمكن تبسيط هذه الطريقة؟

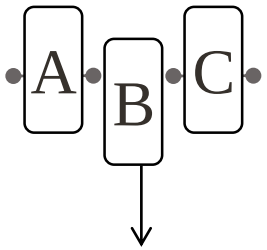
تنجيز اللائحة المترابطة – الحذف (تبسيط)

- استخدام طريقة بحث تعيد مؤشراً إلى العنصر إذا كان موجوداً في اللائحة. يمكن أن نسميها Find.
- تستدعي الطريقة Remove الطريقة Find بدلاً من أن تقوم بذلك بنفسها.

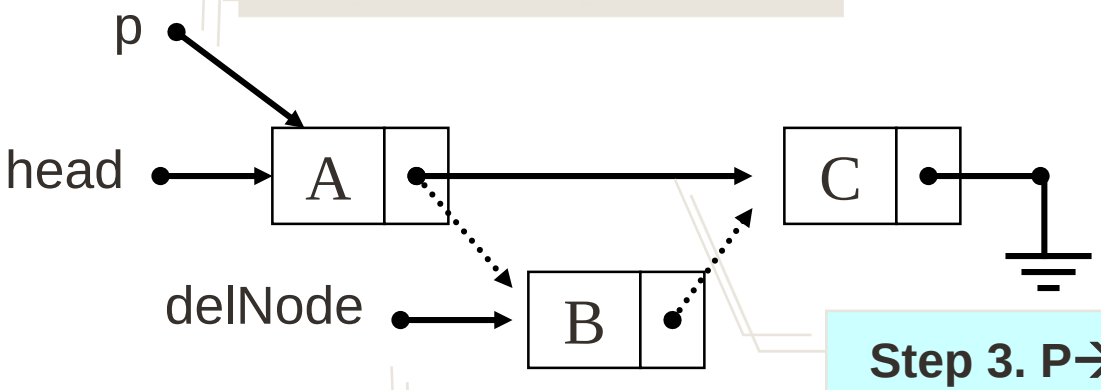
حذف العنصر الموجود في موقع معين من لائحة

RemovePos(2, L)





Step 1. Move p to pos-1

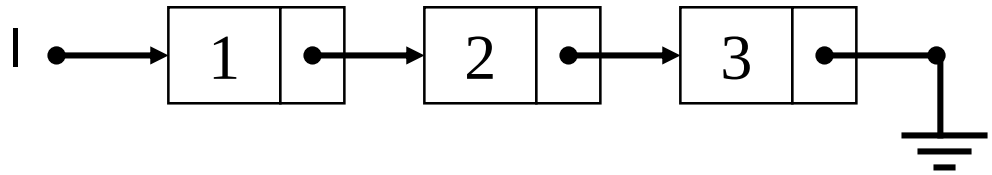


Step 3. P→next = delNode→next

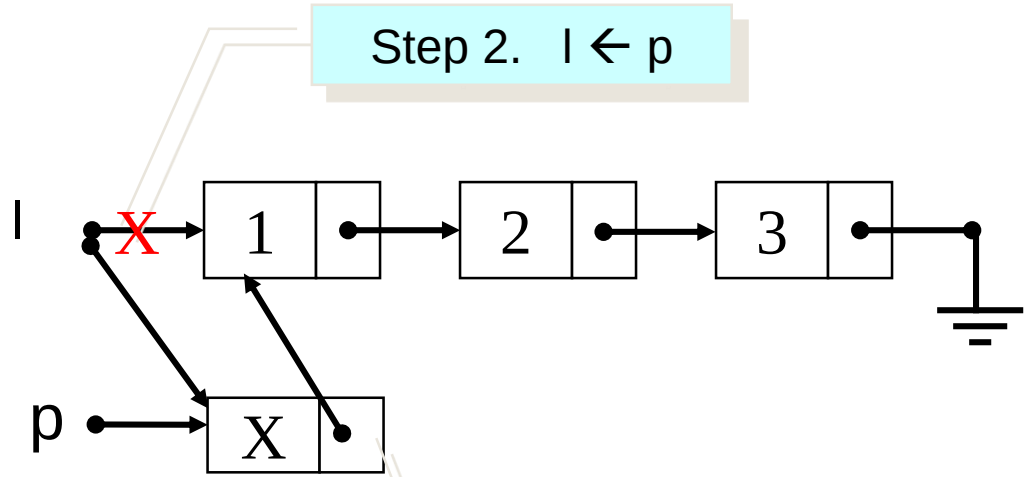
Step 2. delNode = p→next

تنجيز اللائحة المترابطة- الإدراج Insert

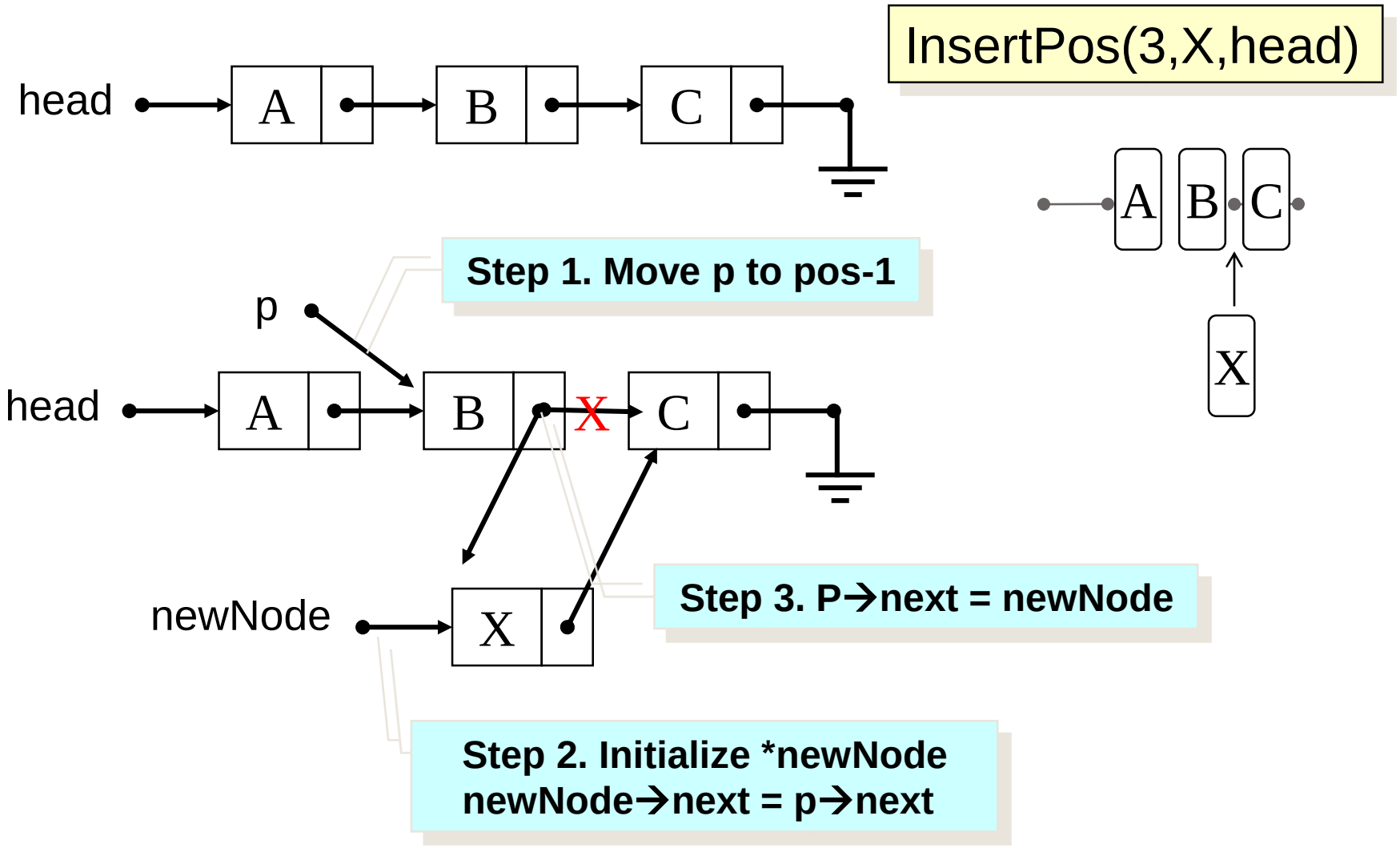
insert(l,x): تابع يدرج العنصر X في بداية اللائحة المعطاة.




```
void insert(List &l, Type x)
 {
    Item* p = new Item;
    p->val = x;
    p->next = l;
    l = p;
 }
```



تنجيز اللائحة المترابطة- إدراج عنصر في موقع معين



تنجيز اللائحة المترابطة- إدراج عنصر ضمن لائحة مرتبة

```
void LinkedList::Insert(int newData){
  node   *previous;
  node   *current = List;
  node   *newNode = new node(newData);
  if (List == NULL)  List = newNode;
  else if (List->data >= newData){
       newNode->next = List;
       List = newNode;}
  else{
       while(current) {
          if (current->data >= newData) {
                    previous->next = newNode;
                    newNode->next = current;
                    break; }
          else {
                    previous = current;
                    current = current->next;  } }
       if (current == NULL)   previous->next = newNode;}}
```

مميزات اللائحة المترابطة Linked List

- يجري الوصول إلى اللائحة عبر أول تسجيلة (خلية).
- يجري البحث عن قيمة معينة بزمن خطي O(n).
- الوصول إلى العنصر رقم k أبطأ من الصفيفة

  (O(k) وفي أسوأ الحالات O(n)).

مميزات اللائحة المترابطة Linked List

- يتطلب الحذف تغيير مؤشر واحد فقط.
- يتطلب الإدراجِ خليةً جديدةً من النظام system وتغييراً في مؤشرين.
- عند انتهاء الحاجة إلى مكان مخصص ديناميكياً، يمكن استخدام أمر Delete(P) (P تؤشر إلى null) لإعلام النظام أن بإمكانه استخدام المكان.

مقارنة بين التنجيز باستخدام الصفيفة واللائحة المترابطة

	linked list	array
get i- th element	***O(n)***	***O(1)***
find length	***O(n)***	***O(1)***
delete from sorted list	***O(n)***	***O(n)***
add to sorted list	***O(n)***	***O(n)***
find	***O(n)***	***O(n)***
delete last	***O(n)***	***O(1)***
delete first	***O(1)***	***O(n)***
add to end	***O(n)***	***O(1)***
add to beginning	***O(1)***	***O(n)***

عمليات أخرى؟

- قد يحتاج تطبيق ما إلى إضافة عناصر إلى نهاية اللائحة مراراً.
  مثال: خط الانتظار
- قد نحتاج إلى طباعة العناصر من الأخير إلى الأول مراراً.
  مثال: طباعة العناصر داخل مكدس وفق ترتيب دخولها
- قد نريد مراراً حذف خلية من وسط لائحة (لدينا مؤشر خارجي عليها)

ما تعقيد هذه العمليات باستخدام اللائحة المرتبطة؟

ما نمط المعطيات الذي يسمح بإجراء هذه العمليات بتعقيد أقل؟

اللوائح المضاعفة الارتباط ⟵

اللائحة المرتبة The Inorder List

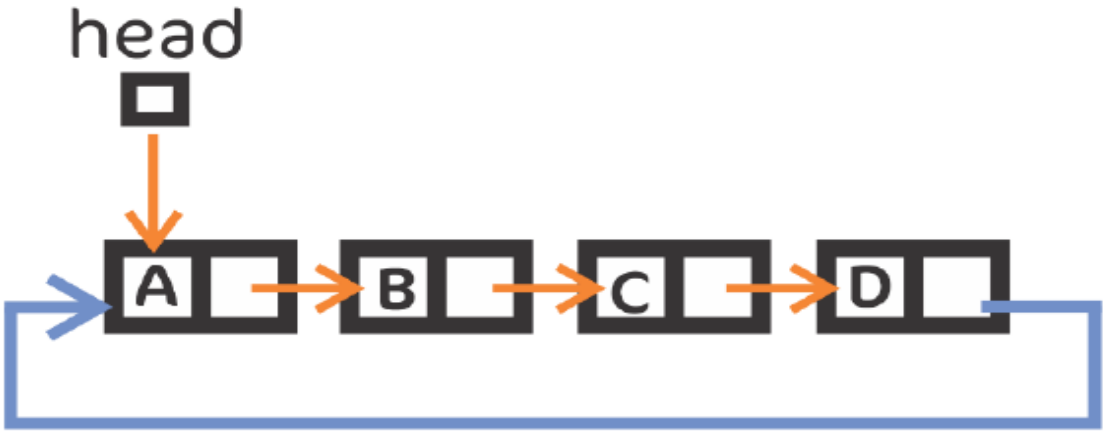
- ✓ **المميزات:** هو لائحة تحتوي عل عناصر من نوع $ListElementype$ مرتبة في الكامل.

الحدود في اللائحة تكون مرتبة بحيث أنه إذا كان a وb حدود في اللائحة و $a > b$ عندها فإنّ b سيكون قبل a في اللائحة.

- ✓ **شرط مسبق:** يجب أن يكون نوع المعطيات الأساسي لهذه اللائحة يقبل المقارنة أي الإشارات $=, >, <$.
- ✓ عمليات توصيف التوابع في اللائحة المرتبة.

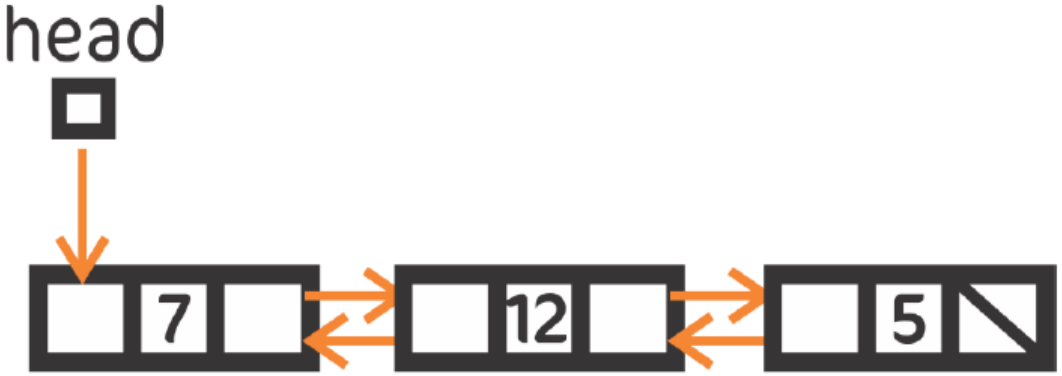
اللائحة الدائرية المترابطة

يخزن عنوان الحد الأول في الحد الأخير أي أنه بدلاً من أن يكون آخر حد $null$ فهو يحتوي على عنوان أول عقدة



اللائحة المترابطة ربط مضاعف *Doubly Linked Lists*

في اللائحة المترابطة ربط بسيط لا نستطيع الرجوع للوراء أي معرفة قيم الحدود السابقة أما في اللائحة المترابطة ربط مضاعف فنستطيع ذلك والسبب أنني أستطيع في العقدة أن أسجل عنوان الحد السابق وعنوان الحد التالي.



اللائحة الديناميكية الخطية *A Dynamic Linear List*

تفرق عن اللائحة الديناميكية العادية أن الأخيرة هي عبارة عن لائحة مترابطة بسيطة أما اللائحة الديناميكية الخطية فهي عبارة عن مصفوفة ولكن يتم تحديد حجم المصفوفة عن طريق باني.