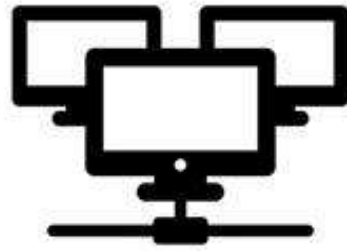


ملخص لمادة:

تركيب بيانات



عضو الجمعية الطلابية
عبد الرحمن ابو زيد

خدمتكم عبادة نتقرب بها الى الله

Team
ZerOne
Computer & Networks Engineers

Data Structure:

A data structure is a specialized format for organizing and storing data. هيكل البيانات هي صيغة خاصة لتنظيم وتخزين البيانات.

General data structure types include the array, the file, the tree, and so on. البيانات العام المجموعة، الملف، الشجرة، الخ.....

Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

□ تم تصميم أي هيكل بيانات ليناسب مع غرضه محدد بحيث يمكن الوصول إليها والتعامل معها بطرق مناسبة.

In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms.

في برمجة الكمبيوتر يمكن اختيار بنية البيانات لتخزين البيانات لغرض العمل على ذلك مع مختلف الخوارزميات.

★ بعض المفاهيم ينبغي للطالب معرفتها قبل البدء ★

Pointer: مؤشر

- Address of a variable in memory عنوان متغير في الذاكرة
- Allows us to indirectly access variable يسمح لنا بالوصول بشكل غير مباشر للمتغير

*In other word, we can talk about its address rather than value.

يستخدم المؤشر لغة C++ كعنوان المتغير في الذاكرة ، احد الاستعمالات المهمة للمؤشرات
هو التخصيص الديناميكي للذاكرة حيث يتم استعمال المؤشرات لإنشاء بنية بيانات لتخزين
البيانات في الذاكرة.

- The pointer is define before use in program .

Array:

- A list of value arranged sequentially in memory.

Example: a list of telephone numbers a[4] refers of the 4th elements of array a.

Address and value:

1- some time we want to deal with the address of memory location, rather than the value it contain .

بعض الوقت نحتاج التعامل مع عنوان موقع الذاكرة، بدلاً من القيمة التي تحتوي عليها.

Ex: R1 is pointer it contains address of -first location- data we are interested in .

R1

	Value	Address
X3100	X 3107	X 3100
	X 2819	X 3101
	X 0110	X 3102

2-consider the following function that is supposed to swap the values of its arguments. مثال لتبديل بين القيم

```
Void swap(int firstval , int secval)
```

```
{Int temp val =firstval;
```

```
firstval =secval;
```

```
secval=tempval;}
```

Team
ZeroOne

Before call

swap	4	First val
	3	Sec val
main	3	A
	4	B

after call

swap	4	Temp val	These value change
	3	First val	
	4	Sec val	
main	4	A	But these didn't
	3	B	

Swap needs addresses of variables out side its activation record

Let us talk about manipulate pointer as variable and in expressions:

Ex: `int*p; // *p is pointer to an int`

نعمله عن المؤشر `p` ليشير الى المتغير من نوع `int` وكل متغير يعمل عنه كمؤشر يجب ان يكتب في الاعلان مسبقا ب `*` ، مثلا :

`Float *X, double*y ,char*z,.....`

يشير الى موقع من نوع `char, float.....`

`Int*x,y;` غير صحيحة

`Int *x ,*y;` صحيحة (يمكن اعطاءه قيمة ابتدائية ويمكن لا)

- تذكر دائما عند الاعلان عن اي مؤشر ان تسبق `*` كل مؤشر على حدة.

Operators:

***p:** return the value pointed to by p *يرجع القيمة التي يحملها معاملة p*

&Z: return the address of variable z

يستعمل لمعرفة العنوان يحملته متغير ما (يرجع عنوان)

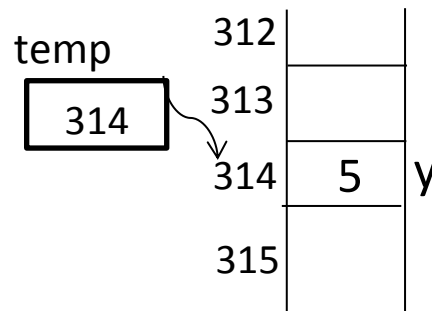
Ex: int y=5;

int * temp,

temp=&y;

cout<<temp ; //314 عنوان y

cout<<* temp ; // 5 قيمة y



Ex: int a=7; int*address; address=&a;

cout <<&a; //0xff4 عنوان a

cout<<address; //0xff4 عنوان a

cout<<a; //7 قيمة a

cout<<*address; //7 قيمة a

cout<<&*address; //0xff4

Cout<<*&address; //0xff4

*العنوان الذي نضعه في المؤشر يجب ان يكون منه نفس نوع المؤشر لا يمكنه تعيين عنوان متغير float الى مؤشر int

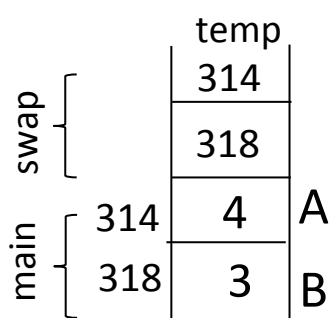
*passing a pointer into a function allows the function to read chsnge memory out side its activation record.

Ex: void swap(int*first,int*sec)

{int temp =*first;

*first= *sec;

*sec= temp;}



int*sec: arguments are integer pointers
caller posses addresses of variable that it
want function to change

يتم تمرير عناوين المتغيرات المراد تطبيق العمليات عليها

Ex: char word Elo];

char*cptr;

Cptr=word; // *points to word Eo]

Cptr=cptr+1; //can change the content of cptr

*(cptr+3)=13; //word[4]=13

مثلا لو كان عنوان cptr 301 يصعب 302 لأنها char

لو كانت int يزيد 4 فيصعب 305 حسب الحجم الذي تجهزه في الذاكرة

Cptr	Word	&word[0]
Cptr+n	Word+n	&word[n]
*cptr	*word	Word[0]
*(cptr+n)	*(word+n)	Word[n]

Factor new:

يخصص العامل new مساحة ذات حجم معين ويعد مؤشر نقطة بداية تلك المساحة .

P-var=new type

P-var: مؤشر يتم فيه تخزين بداية المساحة المحبوزة

New type: نسمح بتخزينه متغير منه نوع type

Team
ZeroOne

Factor delete:

إذا تم حجز العديد من المساحات في الذاكرة بواسطة **new** سيتم حجز كل الذاكرة

الموافرة وسيتم حذف الحاسوب عن العمل لذلك يرافقة العامل **new** عامل **delete**

```
Char*str="It is the best";
```

```
Int len=strlen (str);
```

```
Char*ptr;
```

```
Ptr=new char [len+1];
```

Char → نوع المتغير
Len+1 → عدد المتغيرات

```
Strepy (ptf,str);
```

```
Cout<<"ptr="<<ptr; // It is the best
```

```
Delete [ ] ptr ;
```

تعيد للنظام كمية الذاكرة التي يسير اليها المؤشر

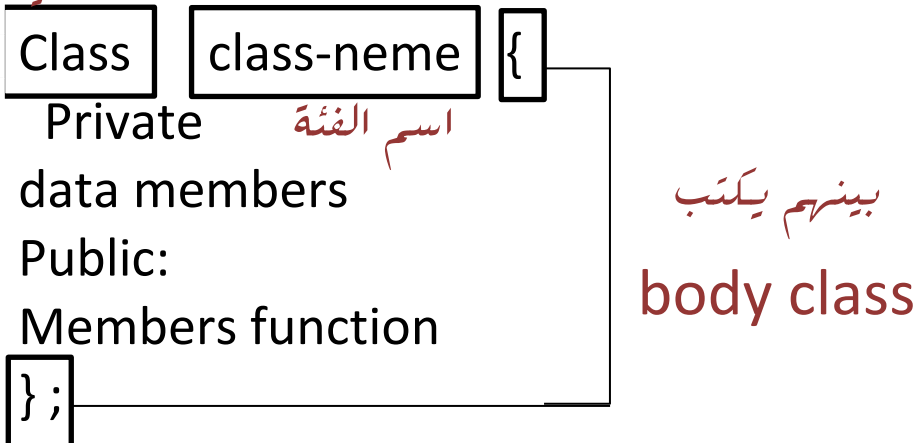
=====

Team
ZerOne

Classes:

اساس البرامج المكتوبة بلغة C++ هي كائنات يتم انشاؤها بواسطة فئة تستعمل كقالب
فعندما يكون الكثير من الكائنات المتطابقة في البرامج لا يكون منطقيا وصف كل واحدة على حدة
فمنه الافضل تطوير مواصفات واحدة لكل هذه الكائنات وبعد تحديد هذه المواصفات يمكنه

استخدامها لانشاء قدر ما نحتاج اليه من الكائنات او هذا ما يسمى **class** كلمة أساسية



data member: اعضاء بيانية

يتم كتابتها بنفس الطريقة التي يتم كتابة المتغيرات الا انه لا يمكننا اعطاها قيمة

ابتدائية عند الاعلان عنها ويمكنه ان تكون من اي نوع من البيانات

int stack[size]/int x/char y/ flout z.....

Membes function : اعضاء دالية

Team
ZeroOne

يمكنه من انجاز العديد من العمليات على الكائنات التابعة لها ☐

Void push(int i) /int pop() /void print().....☐

يمكن الوصول إليها من أي مكان داخل البرنامج **Public:**

يمكن الوصول إليها فقط من الأعضاء الداخلية داخل **private: public**

يتم إنشاء الكائنات باستخدام نفس التركيب المستخدم لإنشاء متغير من نوع **int** مثلاً وذلك أن الكائنات تتم معاملتها كأنواع متغيرات كما تتم معاملة الفئات كأنواع بيانات.

لإنشاء كائنه تابع ل **class stack** ← **stack L1;**

عند التنفيذ بحسب البرنامج حجم الكائنه ويعطيه مساحة **1** سم

- يتم التفاعل مع الكائنات من خلال استدعاء احد اعضائها الداخلية نحتاج تركيب مؤلف من قسمين

اسم الكائنه واسم العضو الداخلي ويتم الربط بينهم بواسطة (.) **L1.print();**

=====

Unsorted list "Array Implementation"

```
#include <iostream.h>
```

```
template <class T, const int MAX_ITEMS>
class UnsortedList
```

```
{
public:
    UnsortedList () ;
    int IsFull () const ;
    int Lengths () const ;
    void RetrievalItem (T& item, int& found );
    void InsertItem (T item );
    void DeletionItem (T item);
    void ResetList () ;
    void printList () const;
    T GetNextItem () ;
```

```
private:
    int length;
    T info[MAX_ITEMS] ;
    int currentPos;
};
```

```
template <class T, const int MAX_ITEMS>
    UnsortedList<T, MAX_ITEMS>::UnsortedList ()
        {length = 0; ResetList () ; }
```

```
template <class T, const int MAX_ITEMS>
    int UnsortedList<T, MAX_ITEMS>::IsFull () const
        { return (length == MAX_ITEMS) ; }
```

```
template <class T, const int MAX_ITEMS>
    int UnsortedList<T, MAX_ITEMS>::Lengths () const
        { return length ; }
```

```
template <class T, const int MAX_ITEMS>
    void UnsortedList<T, MAX_ITEMS>::InsertItem (T item)
        { if(!IsFull())
            info[length++]=item ; }
```

```
template <class T, const int MAX_ITEMS>
    void UnsortedList<T, MAX_ITEMS>::ResetList ()
        { currentPos = -1; }
```

```
template <class T, const int MAX_ITEMS>
    T UnsortedList<T, MAX_ITEMS>::GetNextItem ()
        { return info[++currentPos] ; }
```

```
template <class T, const int MAX_ITEMS>
    void UnsortedList<T, MAX_ITEMS>::RetrievalItem (T& item, int& found)
        { found = 0; // false
          for (int location = 0; location < length; location++ )
              if (item == info[location] )
                  found = 1; //true
          }
```

```
template <class T, const int MAX_ITEMS>
    void UnsortedList<T, MAX_ITEMS>::DeletionItem (T item )
        {
            for (int location = 0; item != info[location]; location++ ) ;
            info[location] = info[--length];
        }
```

```
template <class T, const int MAX_ITEMS>
    void UnsortedList<T, MAX_ITEMS>::printList () const
```

```

{
    for (int location = 0;location < length;location++ )
        cout << info[location]<< " ";
    cout << endl;
}

```

```
void main()
```

```

{
    UnsortedList <int,10>L1;
    L1.InsertItem(10) ;
    L1.InsertItem(70) ;
    L1.InsertItem(30) ;
    L1.InsertItem(90) ;
    L1.InsertItem(50) ;

    L1.printList () ;    // 10
70 30 90 50
    cout << L1.GetNextItem () << " " ;    // 10
    cout << L1.GetNextItem () << " " ;    //70
    cout << L1.GetNextItem() << " " ;    //30
    cout << L1.GetNextItem () << " " ;    //90
    cout << L1.GetNextItem() << "\n n " ;    // 50

    UnsortedList <char,5>L2;
    L2.InsertItem('a' ) ;
    L2.InsertItem('b' ) ;
    L2.InsertItem('c' ) ;
    L2.InsertItem('d ' ) ;
    L2.InsertItem('e' ) ;

    L2.printList () ;    // a b c d e
}

```

0	1	2	3	4	5	6	7	8	9

10									
----	--	--	--	--	--	--	--	--	--

10	70								
----	----	--	--	--	--	--	--	--	--

10	70	30							
----	----	----	--	--	--	--	--	--	--

10	70	30	90						
----	----	----	----	--	--	--	--	--	--

10	70	30	90	50					
----	----	----	----	----	--	--	--	--	--

0	1	2	3	4

a				
---	--	--	--	--

a	b			
---	---	--	--	--

a	b	c		
---	---	---	--	--

a	b	c	d	
---	---	---	---	--

a	b	c	d	e
---	---	---	---	---

1- Unsorted list "Array Implementation"

Array: some member have the same type

-building array:

Int []x= new int [6] **int** عندما تطبقها يتم حجز مكان في الذاكرة يتسع ل 6 عناصر

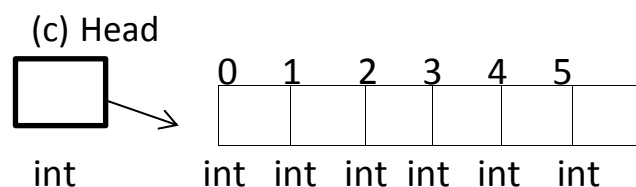
To build array must satisfy the following conditions :

- Fixed size حجم ثابت
- Identical جميع العناصر من نفس النوع
- Sequential جميع العناصر تكون مرتبة بشكل متتالي ولا يوجد بينهم فراغ

* Every array have plus data ^{معلومات إضافية} it is what is known such as "head" it is the address of the first element in array

* we use head to make it easy to access the element in array

نستخدم head حتى يسهل علينا الوصول لعناصر المصفوفة



int size = 32 bit

المساحة المحجوزة = $32 * 6$

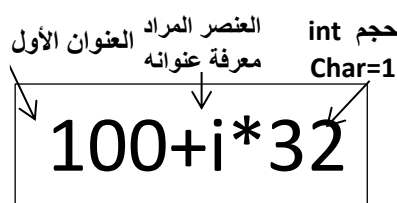
C= 32 bit=4byte

لو افترضنا أن عنوان $x[0]=1000$ bit

فإن $x[1]=1032$

$x[2]=1064$

من خلال هذه القانون يمكن معرفة عنوان أي عنصر في المصفوفة



عند التدقيق في الكود السابق نجد ان التطبيقات التي يتم اجراءها عند انشاء كائنه هي :

“تهيئة list * طباعة طول list * اضافة عنصر, والاضافة تتطلب معرفة ان كانت ممثلة ام لا. * حذف عنصر * طباعة عناصر * بحث عنه عناصر” .

-template < class T ,const int MAX-ITEMS>

تم استخدام **templat** لتكون ال **list** اعم حيث بإمكاننا فيه نفس الكود تعريف

doulde list, char list, int list

Constructor:

template <class T, const int MAX-ITEMS>

unsorted list<T,MSX-ITEMS> ::unsorted list()

{length=0;Reset list() ;}

عند انشاء **new list** لابد منه تصغير الطول لها واستدعاء **Reset list** الذي يقوم بدوره

في اعطاء قيمة ابتدائية للمتغير **currentpos** الذي نستخدمه في عمليات اخرى

template<class T, const int MAX-ITEMS>

-void unsortedlist <T,MAX-ITEMS> :: Reset list()

{current pos=-1;}

استخدامنا **void** لأنه لا يتم إرجاع قيمة

Void → not return value

T , int , char , → return value

template<class T,const int Max-ITEMS>

T unsortedlist< T,MAX-ITEMS>::Get next item()

{return info[++currentpos];}

استخدام **T** في تعريف ال **function** لانه يرجع القيمة ونوع القيمة تتغير منه **list** الى اخرى ,

يستخدم في طباعه عنصر في كل مره 😊

```
template<class T, const int MAX-ITEMS>
```

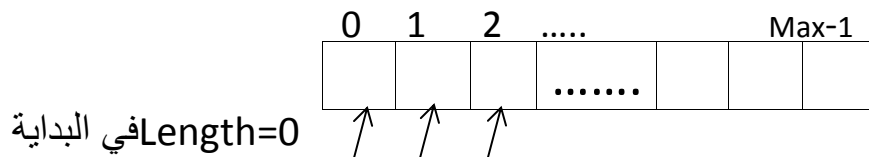
```
-void unsorted list<T, MAX-ITEMS>::insert item(T item)
```

عند اضافة عنصر يتم السؤال اذا كانت ال **list** قد امتلأت ام لا

```
{if(! is full())  
info [length++]=item;}
```

وذلك باستدعاء كود **is full** وبعدها تتم الاضافة بالطريقة

المعتادة في تخزينه عناصر المصفوفة مع زيادة **length** قيمة واحدة حتى ينتقل لل خانة التالية



تكون full عندما
length=max

```
template <class T, const int MAX-ITEMS>
```

```
void unsorted list<T, MAX-ITEMS>:: retrieve item(T& item,int&found)
```

```
{Found=0
```

```
For(int loc=0, loc<length,loc++)
```

```
If(item==info[loc])
```

```
Found=1 ;}
```

عندما نريد البحث عنه عنصر نخزن قيمة (1) في المتغير **found** ان وجد و(0) ان لم يوجد, لذلك

نعطي **found** قيمة ابتدائية 0 على فرضه ان العنصر غير موجود ثم نبدأ بالتنقل بين العناصر

جميعها لذلك نحتاج جملة **for** نبدأ من 0 وننتهي مع انتهاء العناصر اي لقيمة **length** ثم مقارنة

القيم المخزنة بالقيمة التي نبحث عنها ان وجدت نعدل على **found**

Team
ZerOne

```
template<class T, const int MAX-ITEMS>
```

```
-void unsorted list<T, MAX-ITEMS>::delete item(T item)
```

```
{For (int loc=0, item!= info[loc], loc ++);
```

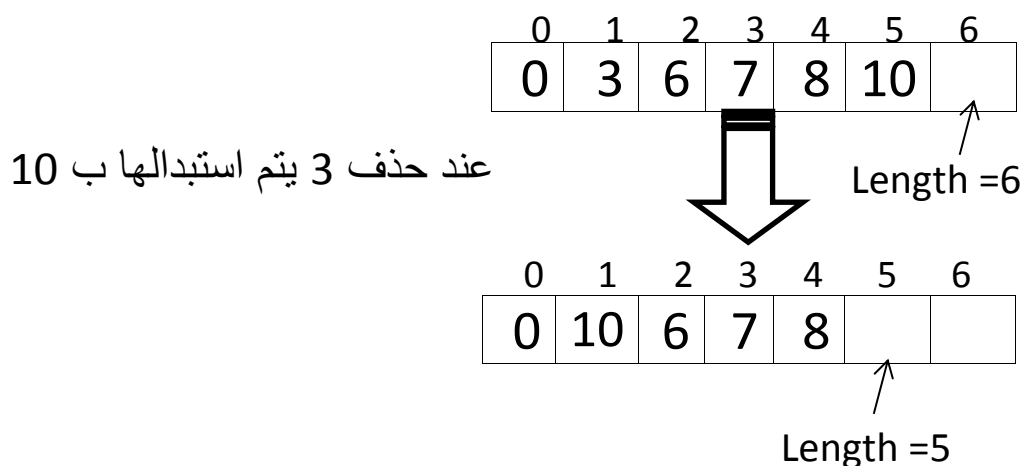
```
info ,[loc]=info[--length];}
```

عند حذف عنصر معين من list لا يتم البحث عنه في الصفوفة من خلال جملة **for** تبدأ من اول عنصر

وتنتهي عند العنصر المراد حذفه "لاحظ انه في جملة **for** لا يتم تنفيذ اي جملة سوى المرور على

العناصر". عند الوصول للعنصر المراد حذفه يتم استبدال القيمة المراد حذفها مع آخر قيمة ثم انقاص

طول ال list.



```
-void unsorted list <T, MAX-ITEMS>::print list()
```

```
{For(int loc=0, loc<length, loc++)
```

```
cout<<info[loc];}
```

عند طباعة العناصر يتم المرور على جميع العناصر من خلال **for** وطباعتها.

sorted list

```
#include <iostream.h>

template <class T, const int MAX_ITEMS=100>
class SortedList
{
public:
    SortedList () ;
    int IsFull () const ;
    int LengthIs () const ;
    void Retrieveltem (T& item, int& found) ;
    void InsertItem(T item) ;
    void Deleteltem(T item) ;
    void ResetList () ;
    void printList () const;
    T GetNextItem () ;
private:
    int length ;
    T info[MAX_ITEMS] ;
    int currentPos ;
};
/*****/

template <class T, const int MAX_ITEMS>
    SortedList<T, MAX_ITEMS>::SortedList ()
        { length = 0;ResetList () ; }

template <class T, const int MAX_ITEMS>
    int SortedList<T, MAX_ITEMS>::IsFull () const
        { return (length == MAX_ITEMS) ; }

template <class T, const int MAX_ITEMS>
    int SortedList<T, MAX_ITEMS>::LengthIs () const
        { return length ; }

template <class T, const int MAX_ITEMS>
    void SortedList<T, MAX_ITEMS>::ResetList ()
        { currentPos = -1; }

template <class T, const int MAX_ITEMS>
    T SortedList<T, MAX_ITEMS>::GetNextItem ()
        { return info[++currentPos] ; }

template <class T, const int MAX_ITEMS>
    void SortedList<T, MAX_ITEMS>::Retrieveltem(T& item, int& found)
        {
            found = 0;// false
            for (int location = 0;location < length;location++ )
                if (item == info[location] )
                    found = 1;//true
        }

template <class T, const int MAX_ITEMS>
    void SortedList<T, MAX_ITEMS>::printList () const
        {
            for (int location = 0;location < length;location++ )
                cout << info[location]<< " " ;
            cout << endl;
        }
```

```

template <class T, const int MAX_ITEMS>
void SortedList<T, MAX_ITEMS>::InsertItem(T item)
{
    if(IsFull ( ) )
        return;
    for (int i=0; i<length; i++ )
        if (info[i]>item)
        {
            for (int j=length;j>i;j-- )
                info[j]=info[j-1] ;
            break;
        }
    info[i]=item;
    length++ ;
}

```

```

template <class T, const int MAX_ITEMS>
void SortedList<T, MAX_ITEMS>::DeleteItem(T item)
{
    int location;
    for ( location = 0; item != info[location];location++ ) ;
    for (int i=location; i<length; i++ )
        info[i]=info[i+1] ;
    --length;
}

/*****/

```

```

void main ( )
{

```

```

    SortedList <int,10>L1;

```

```

    L1.InsertItem(10);

```

10									
----	--	--	--	--	--	--	--	--	--

```

    L1.InsertItem(70) ;

```

10	70								
----	----	--	--	--	--	--	--	--	--

```

    L1.InsertItem(30);

```

10	30	70							
----	----	----	--	--	--	--	--	--	--

```

    L1.InsertItem(90);

```

10	30	70	90						
----	----	----	----	--	--	--	--	--	--

```

    L1.InsertItem(50);

```

10	30	50	70	90					
----	----	----	----	----	--	--	--	--	--

```

    L1.DeleteItem(30);

```

10	30	30	50	70	90				
----	----	----	----	----	----	--	--	--	--

```

    L1.printList ( ) ;

```

```

    cout << L1.GetNextItem ( ) << " " ;

```

```

    cout << L1.GetNextItem ( ) << " " ;

```

```

    cout << L1.GetNextItem ( ) << " " ;

```

```

    cout << L1.GetNextItem ( ) << " \n" ;

```

```

    SortedList <char>L2;

```

```

    L2.InsertItem('a') ;

```

a									
---	--	--	--	--	--	--	--	--	--

```

    L2.InsertItem('c') ;

```

a	c								
---	---	--	--	--	--	--	--	--	--

```

    L2.InsertItem('e') ;

```

a	c	e							
---	---	---	--	--	--	--	--	--	--

```

    L2.InsertItem('d') ;

```

a	c	d	e						
---	---	---	---	--	--	--	--	--	--

```

    L2.InsertItem('b') ;

```

a	b	s	d	e					
---	---	---	---	---	--	--	--	--	--

```

    L2.printList ( ) ;}

```

2-Sorted list

لا يختلف كود sorted list عنه unsorted list في شيء سوى في الإضافة والحذف

```
-template <class T, const int MAX-ITEMS>
```

```
int sorted list <T, MAX-ITEMS>::Insert Item(T item)
```

```
{If (Is full())
```

يُقارن بين القيمة المراد إضافتها مع القيم الموجودة في list ,

```
return 0;
```

إذا وجد عنصر أكبر منه المراد إضافته يرجع كل العناصر التي

```
for (int i=0; i<length; i++)
```

بقيت خانة واحدة ونخرج منه جملة for كجملة لوجود break

```
if (info[i]>item)
```

ثم نضيف العنصر الجديد

```
{for (int j=length, j>i, j--)
```

```
info[j]=info[j-1];
```

لتحضير موقع للقيمة الجديدة

```
break;
```

```
}
```

```
info[i]=item;
```

```
length++;
```

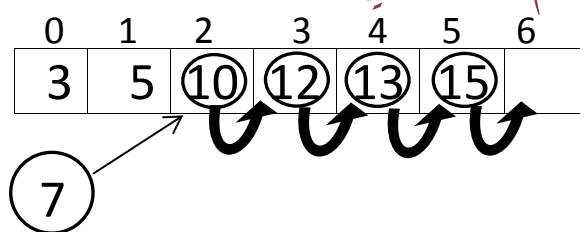
```
return }
```

* عند الإضافة يجب مراعاة الترتيب

عند الإضافة تتم مقارنة العناصر في list مع القيمة المراد إضافتها item إلى أن نصل

إلى قيمة أكبر منه item ثم يتم إزاحة العناصر الأكبر منه item خانة واحدة لليمين

ثم إضافتها في المكان المناسب لها.



عند إضافة 7 سيكون موقعها بين 10 و 5

فسوف يتم إزاحة

خانة واحدة ثم ادخال 7 مكان 10

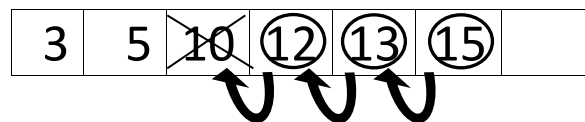
```

-template<class T, const int MAX-ITEMS>
void sorted list<T, MAX-ITEMS>::delete(T item)
{For(int i=0; item !=info[i];i++); ←———— بحث عنه مكان العنصر
For(int y=i; y<length ; y++)
info[y]=info[y+1];
--length;}

```

عند الحذف يتم تحديد العنصر المراد حذفه (جملّة for الأولى)

ثم تتم إزاحة العناصر الموجودة على يمين العنصر خانة واحدة لليسار (جملّة for الثانية)



Team
ZeroOne

Some applications on array :

تطبيقات على الـ array يمكنه ان تأتي بالامتحان على شكل كتابة كود

1-reverse array

x s E l e c t

Have this array . Write a program to reverse elements.

t c E l e s

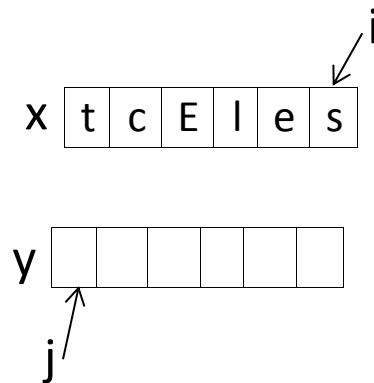
نحتاج برنامج يقوم بعكس ترتيب العناصر حتى نصبح بهذا الشكل

1- نحتاج الى انشاء array جديدة حتى يتم تخزين الترتيب الجديد بها

2- نحتاج الى متغيريه احدهم يخزن القيمة الاخيرة في المصفوفة الاولى والثاني القيمة الاولى

في المصفوفة الثانية "الجديدة"

```
int size=6;
Char[] x= new char [size]
X[]=select;
char [] y= new char [size]
int j=0
for (int i=size-1; i>= 0; i--)
y[j++] =x [i];
```



2-swapping variables

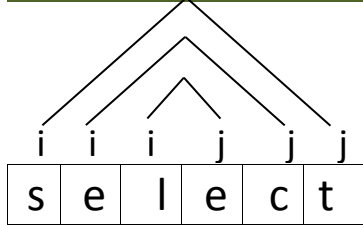
0 1 2 3 4 5
s e l e c t

تبديل مكان عنصريه

```
Char temp ;
temp= x[0];
x[0]=x[5];
x[5]=temp;
```

Team
ZeroOne

3-reverse array in place



عكس العناصر في نفس المكان هي عبارة عنه عملية تبديل swap العناصر

عدد عمليات التبديل = $1/2 \text{ size}$

لذلك سنقوم ب 3 عمليات تبديل

```
for (int i=0, j =size-1 ;i<(size/2) ;i++,j--)  
{char temp =x[i];  
x[j]=temp;}
```

4-sorting array

0	1	2	3	4	5
8	6	9	4	2	5

ترتيب العناصر بعد ادخالها غير مرتبة

بنفكر بطريقة تلف على العناصر كلها بنأخذ العنصر الاول وبنقارنه مع العناصر حتى نوجد اقل

عنصر وبعدها بنعمل تبديل بينه وبين الاول بعدها بنأخذ العنصر الثاني

ونفس العملية بنعيد لها كمان مرة

Min = ~~8~~ ~~6~~ ~~4~~ 2

اول مرة بتأخذ 8 وبنخليها min

بعد الدوران على كافة العناصر تكون قيمة min=2 ثم نعمل تبديل "swap" مع 8

```
int min=x[0]; int i,j, temp ,k
```

```
for (i=0;i<size ;i++) ← الدوران على العناصر
```

```
{for( j=0 ; j<size ; j++)← مقارنة العناصر
```

```
if (x[j] <min)
```

```
{min= x[j] ;k=j;}
```

```
temp =x[i];
```

```
x[i]=x[k];
```

```
X[k]= temp;}
```

تبديل العناصر

Team
ZeroOne

Stack "using array"

```
#include <iostream>
using namespace std;
template <class T, const int MAX_ITEMS=100>
class Stack
{ public:
    Stack() ;
    int isFull() const;
    int isEmpty() const;
    int length() const;
    void push(T item) ;
    T pop() ;
private:
    int length;
    T info[MAX_ITEMS] ;    };
/*****/
template <class T, const int MAX_ITEMS>
Stack<T, MAX_ITEMS>::Stack()
{length = 0;}

template <class T, const int MAX_ITEMS>
int Stack<T, MAX_ITEMS>::isFull() const
{ return (length == MAX_ITEMS) ; }

template <class T, const int MAX_ITEMS>
int Stack<T, MAX_ITEMS>::isEmpty() const
{ return (length == 0);}

template <class T, const int MAX_ITEMS>
int Stack<T, MAX_ITEMS>::length()const
{ return length; }

template <class T, const int MAX_ITEMS>
void Stack<T, MAX_ITEMS>::push(T item)
{    if(! isFull())
    info[length++]=item;    }

template <class T, const int MAX_ITEMS>
T Stack<T, MAX_ITEMS>::pop()
{    if(! isEmpty())
    return info[--length] ;
    return 0;
}
/*****/
int main()
{    Stack <int,4>L1;
    L1.push(10);
    L1.push(20);
    L1.push(30);
    L1.push(40);
    cout << L1.pop() << " " ; \40
    cout << L1.pop () << " " ; \30
    cout << L1.pop() << " " ; \20
    cout << L1.pop()<< " \n" ; \10

    Stack <char>L2;
    L2.push('a');
    L2.push('b');
    L2.push('c');
    L2.push('d');
    L2.push('e');
    cout << L2.pop() << " " ;
    cout << L2.pop() << " " ;
    cout << L2.pop() << " " ;
    cout << L2.pop() << " \n" ;
}
```

40
30
20
10

e
d
c
b
a

3- Stack " Using Array "

_A stack is an example of data structure

- . A method of organising data
- . Defined structure and operation

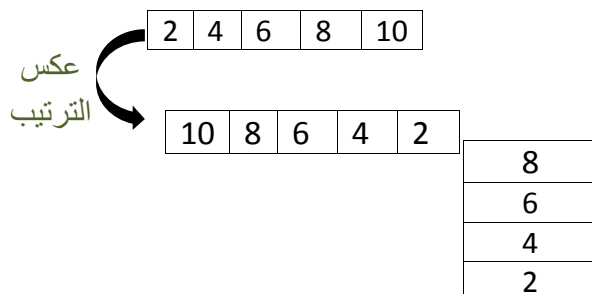
_Stack typically used for temporary storag of data

_Analogous to stack of paper or a stack or a stack of cards

_Some rule :

- . push : place cards on the top of the stack
 - . pop : remove cards from the top of the stack
 - . LIFO : last in is the first out
- compare with FIFO : first in first out

□ هو عبارة عنه طريقة تخزينه للبيانات بتنسيب. أن البيانات تدخل منه مدخل واحد وتخرج منه نفس المدخل : **stack** *



* المصفوفة "list" اذا ادخلنا 2 ثم 4 ثم 6 , 8, 1 <--
* المكس "stack" اذا ادخلنا 2 ثم 4 ثم 6, ... <--

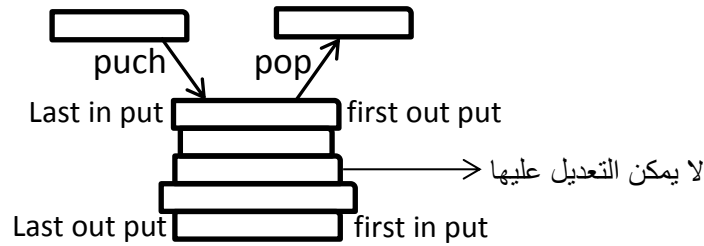
* يتبع المكس مفهوم "LIFO" أي العنصر الذي يدخل أخيراً يخرج أولاً وعند ادخال عنصر نضعه فوق اعلى عنصر

* ادخال عنصر تسمى "push"

* سحب عنصر تسمى "POP"

* كيف نعرف اعلى عنصر ؟

نحتاج الى مؤشر للمصفوفة "عدد صحيح" لمعرفة من هو اعلى عنصر "top", في الحقيقة top ليس مؤشر وإنما هو عدد int ولكن نستخدمه كدليل الى العنصر الاعلى في المصفوفة



To implement a stack data structure we need :

- . An area of memory to store the data items
- . A stack pointer register to point to the top of the stack
- . Some well defined operations : initialize ,push ,pop

To push a word on to the stack :

- . decrement the stack pointer by **4 byte** (1 word = 32 bits = 4 byte)
- . store the word in memory at the location pointed to by the stack pointer

بالعودة الى كود stack نجد انه يحتوي على قسم للاضافة والحذف مرتبط ب `isFull` , `isEmpty`

عند استثناء كائن تابع ل `stack` يتم اعطاء قيمة ابتدائية للطول

```
_stack ( ) { Length= 0 ; }
```

- لمعرفة ان كانت ال `stack` قد امتلأ ام لا بمقارنة الطول الذي وصلنا له مع قيمة `Max` فارغة عندما تكون قيمة `(length= 0)`

```
_int isFull ( ) const
{return (length == MAX - ITEMS);}
```

```
_int isEmpty ( ) const
{return (length == 0);}
```

يتم استدعاء الكودان عند الاضافة او الحذف

```

_Void push (T item)
{ if (! IsFull() )
Info [length ++] = item; }

```

- عند اضافة عنصر الى *stack* تم انشاء يتم السؤال عن اذا كانت ممثلة ام لا "العودة الى كود *isFull* ثم تتم الاضافة كالاضافة الى مصفوفة عادية مع زيادة الطول

1

```

_T pop ()
{ if (! IsEmpty () )
return info [--length];
return 0; }

```

- عند الحذف يتم السؤال ان كانت فارغة باستدعاء *isEmpty* ثم يرجع القيمة التي تم حذفها مع انقاص الطول 1 قبل الحذف او يرجع 0 اذا كانت فارغة

logic error لو قمنا بعملية خامسة ل *stack* يتسع ل 4 عناصر لا ننفذ بشكل صحيح

لكم البرنامج لا يتوقف وينفذ ما يليه امر

يفحص *isFull* ويعطي T ينفذ ويضيف

يعطي F لا ينفذ

لو قمنا بعملية حذف ل *stack* فارغة

يفحص *isEmpty* ويعطي T يرجع 0

يعطي F يرجع اول قيمة

Void main ()

{*stack* < int ,4> L1 ; تم تحديد نوعها int وحجمها 4

Stack < int> L2 ;

تحديد نوعها int فقط ولم يتم تحديد حجمها لذلك تأخذ قيمة max بالنسبة للحجم

Team
ZeroOne

Stack Application

```
#include<iostream.h>
#include<conio.h>
class Stack
{private:
    char s[100] ;
    int depth;
public:
    Stack(){ ( ) depth=0; }
    void push (char x)
    { s[depth++]=x;}

    char pop()
    {   if(depth<0)
        depth=0;
        if(depth>=1)
            return s[--depth];
        else
            return 0;    }

    int IsEmpty() const { return (depth==0) ;}
    char head() {
        if(depth>0)
            return s[depth-1];
        else
            return 0;
    } };

void main ( ) {
    Stack stack;
    char *infix;
    char expression[50];
    cin>>expression;
    infix=expression;
    while(*infix!=0) {
        switch(*infix)    {
            case:'+':
                if(stack.head() != 0)
                    while(!stack.IsEmpty( ) )
                        cout<<stack.pop( ) ;
                stack.push;('+')
                break;
            case:'-':
                if(stack.head() != 0)
                    while(!stack.IsEmpty( ) )
                        cout<<stack.pop ( ) ;
                stack.push;('-')
```

```

        break;
    case: '*'
        if(stack.head()=='*' || stack.head() == '/' )
            while(!stack.isEmpty ( ) )
                cout<<stack.pop ( ) ;
            stack.push('*')
            break;
    case: '/'
        if(stack.head()=='*' || stack.head() == '/' )
            while(!stack.isEmpty ( ) )
                cout<<stack.pop ( ) ;
            stack.push('/') ;
            break;
    default:
        cout<<(*infix) ;
} // end of switch
* (infix++ ) ;
} // end of while

while(!stack.isEmpty ( ) )
    cout<<stack.pop ( ) ;

getch ( ) ;
return;
}

```

4~ Stack Application :

Operator precedence الأولويات

Team
ZeroOne

Priority	Operator	Example	
1	()	(a+b)/c	parenthesis
2	! ~ ++ -- * & sizeof	a=&b	plus/minus/NOT/compliment increment/decrement/sizeof
3	* / %	a*b	multiply/divide/modulus
4	+ -	a+b	add/subtract
5	<< >>	a=b>>c	shift left or right
6	< <= > >=	a>=b	greater/less/equal than
7	== != &	a=b&c	bitwise AND
8		a=b c	bitwise OR
9	^	a=b^c	bitwise XOR
10	&&	a&&b	logical AND
11		a b	logical OR
12	= += -= *= /=	a=b	assignment

operand operator
Ex: 2+3*4
Sol: 2+12
14

يتم إجراء عملية الضرب ثم
الجمع حسب الأولويات

Ex: 5-3*6+100

Sol: 5-18+100
-13+100 = 87

6/2-3+4*2

3-3+8
=8

طريقة الحل اليدوي تسمى infix

In تعني ان الإشارة تكون بين المتغيريه 3+5

لكم لا يفهم الكمبيوتر هذه الطريقة

□ لغة الكمبيوتر تسمى Post fix

Post : الإشارة بعد المتغيريه 35+

$$5+3 \rightarrow 53+$$

معاملين بعدهم اشارة

$$6*7 \rightarrow 67*$$

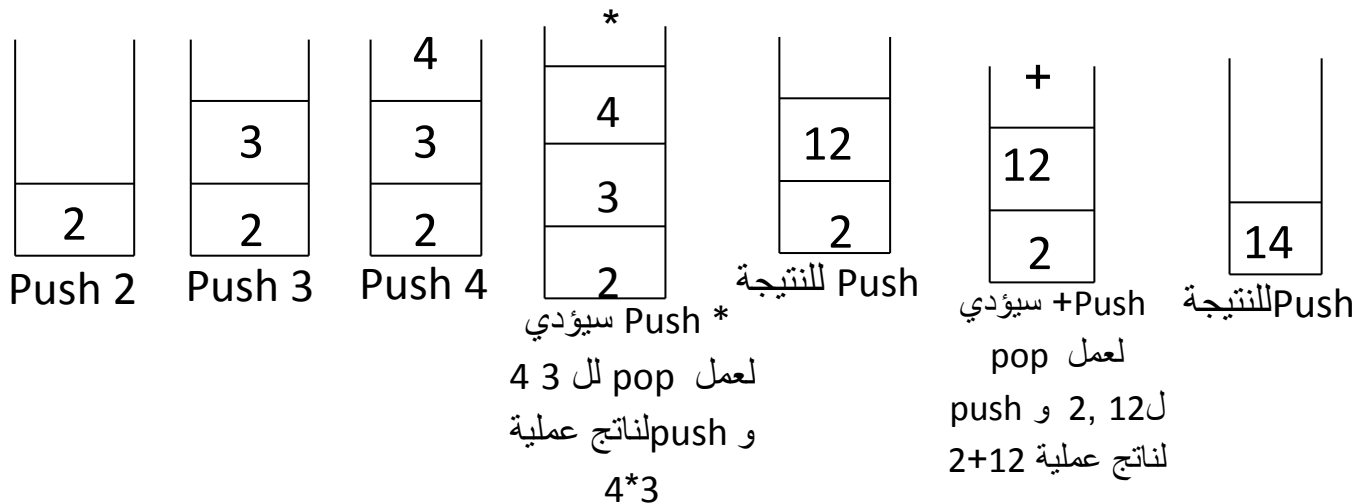
$$2+3*4 \rightarrow 234*+$$

Team
ZeroOne

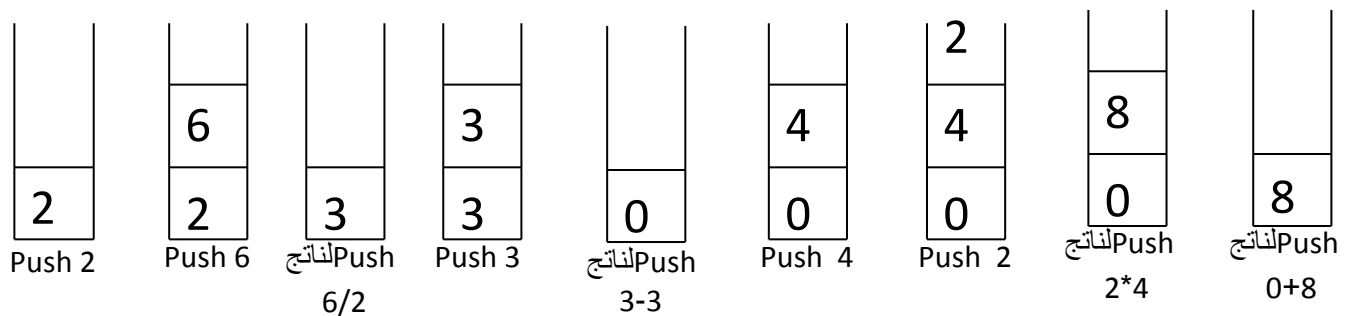
يمكن حلها عن طريق الـ stack؟؟

$$2\ 3\ 4\ * \ +$$

كل ما يأتي operand متغير نجري عملية push وعند الوصول لـ operator إشارة
نجري عملية pop لمتغيرين ويجري عليهم العملية operator

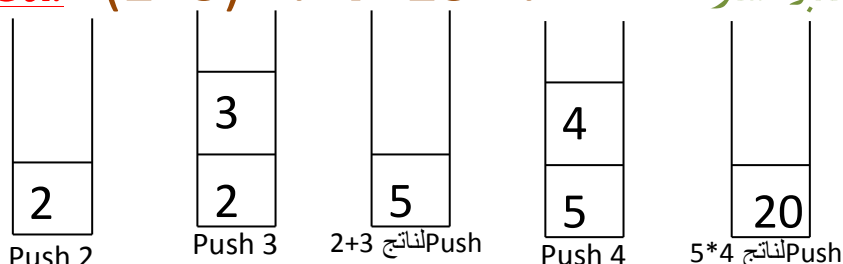


Ex: $6/2-3+4*2 \rightarrow 62/3-42*+$



Ex: $(2+3)*4 \rightarrow 23+4*$

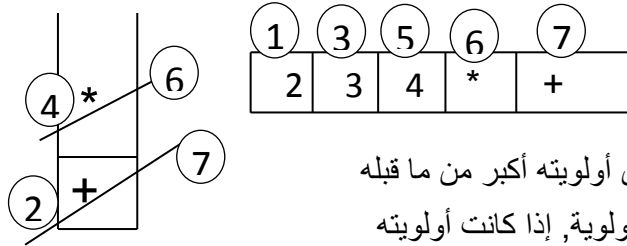
ملاحظة : الأقواس تعبر إشارة



Expression evaluation infix to postfix:

كيف سنحوه ل post fix ؟؟ $2+3*4 \rightarrow$

كل ما يأتي operator إشارة تجري عملية push
وعند الوصول ل operand متغير يوضع على السطر



1- أضع 2 على السطر

2- أضع + في stack

3- أضع 3 على السطر

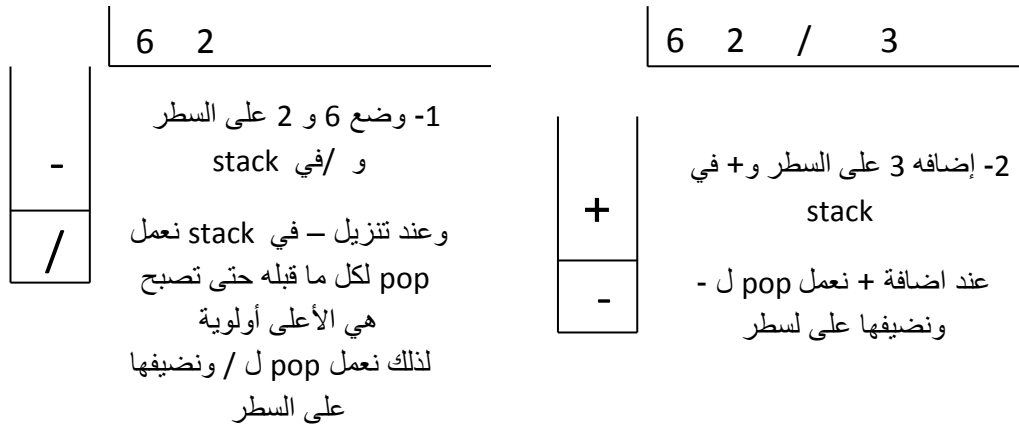
* عندما نريد وضع * في stack يجب أن تكون أولويته أكبر من ما قبله
في ال stack فقط حتى لو كانت مساوية له بالأولوية، إذا كانت أولويته

أقل أو مساوية نعمل pop لكل ما قبله حتى يصبح الأعلى أولوية

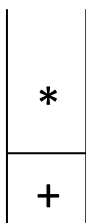
5- أضع 4 على السطر

6- عند نهاية المسئلة pop لكل ما في ال stack

Ex: $6/2-3+4*2 \rightarrow ??$



6 2 / 3 - 4 2



3- إضافه 4 على السطر و*
في stack ثم 2 على السطر
عند الانتهاء من المسألة نعمل
pop لما في ال stack
ونضيفها على السطر

6 2 / 3 - 4 2 * +

Ex: $30+10*5/2 \rightarrow$ 30 10 5 * 2 / +

Ex: $6/(2-3+4)*2+10 \rightarrow$ 6 2 3 - 4 + / 2 * 10 +

القوس يعتبر إشارة فعتند إضافتها لل stack وكأنه بدأنا stack ثانية , وبنهاية

القوس تنتهي ال stack الثانيه ونعود للأولى

Ex: $13+20*((5+2)*3)-2 \rightarrow$ 13 20 5 2 + 3 * * + 2 -

Linear Queue

```
#include <iostream>
using namespace std;
template <class T>
class Queue { public:
    Queue(int max=100);
    ~Queue() { delete [ ] items; }
    int isEmpty( ) const { return (rear == front); }
    int isFull ( ) const { return ((rear +1) == maxQue); } //r+1=max
    void makeEmpty() { front = rear = - 1; }
    void printAll ( ) const;
    int enqueue(T newItem);
    int dequeue(T& item);
private:
    int front, rear;
    T* items;
    int maxQue; };
/*****/
template <class T>
    Queue<T>::Queue(int max){
maxQue = max ;
front = rear = -1;
items = new T[maxQue];}

template <class T>
int Queue<T>::enqueue(T newItem)
// If (queue is not full) newItem is at the rear of the queue
{ if (isFull())
    return 0;
    rear++;
items[rear] = newItem;
return 1;}

template <class T>
int Queue<T>::dequeue(T& item)
// If (queue is not empty) the front of the queue has been removed and returned
{ if (isEmpty())
    return 0;
    front++;
    item = items[front];
    return 1;}

template <class T>
void Queue<T>::printAll( ) const
{for( int i=front+1; i<=rear; i++)
    cout<<items[i]<< " ";
    cout<<endl;}
/*****/

int main(){
    Queue <int> line(3);    int x;
    line.enqueue(10); f=0 , r=-1
    line.enqueue(20); f= 1 , r=-1
    line.dequeue(x); f=1 r=0 x=10
    line.enqueue(30); f=2 r=0
    line.enqueue(40); f=0 r=0

    line.dequeue(x); cout << x << " "; \\20
    line.dequeue(x); cout << x << " "; \\ 30
    line.dequeue(x); cout << x << " "; }\\ 40
```


5- linear Queue

هو عبارة عنه طابور أول عنصر يدخل هو آخر عنصر يخرج وآخر عنصر يدخل هو آخر عنصر يخرج

*First in First out "FIFO"

*Last in Last out "LILO"

عملية الإضافة : enqueue . عملية الحذف : dequeue .



After dequeuer () حذف

* الإضافة تكون من النهاية

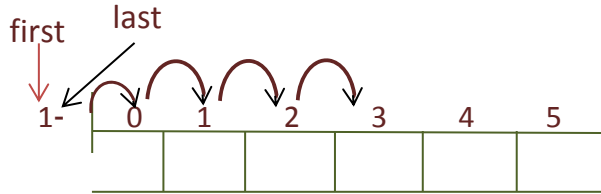


After enqueueer (83) إضافة

والحذف يكون من البداية

حتى نتحكم من الإضافة أو الحذف في المكان الصحيح نحتاج إلى 2 reference أحدهم يشير إلى البداية

للإجراء عملية الحذف والثاني يشير إلى النهاية للإجراء عملية الإضافة



First → front

Last → rear

First : تتحرك مع كل عملية حذف خطوة وتسير في البداية إلى القيمة -1

Last : تتحرك مع كل عملية إضافة

الحجم ثابت ويساوي max

First, last = -1 → empty ∴ First=last

Last = max-1 → full

* مشكلة linear queue أنه يمكنه أن أتصل إلى مرحلة تكون فارغة وممتلئة في نفس الوقت وذلك عنده إضافة جميع

العناصر فيصعب last=m-1 وعند حذف جميع العناصر يصعب last=first لذلك تم تحسينها لcircular queue

```
template <class T>
```

```
Queue<T>::Queue(int max){
```

```
maxQue = max ;
```

```
front = rear = -1;
```

```
items = new T[maxQue];}
```

عند انشاء كائن جديد لا بد من اعطاءه حجم معين
 "قيمة الماكس" واذا لم يتم اعطاءه قيمه يأخذ
 القيمة المفترضة *100 ثم يخزنها في
 المتغير *maxQue* , واعطاء قيمه
rear و *front* قيمة 1- ابتدائية



```
template <class T>
```

```
int Queue<T>::enqueue(T newItem)
```

```
// If (queue is not full) newItem is at the rear of the queue
```

```
{ if (isFull())
```

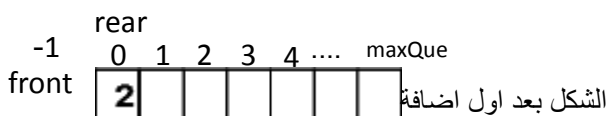
```
    return 0;
```

```
    rear++;
```

```
    items[rear] = newItem;
```

```
    return 1;}
```

بعد انشاء الكائن نبدأ في اضافة العناصر , عند
 الاضافه لا بد من معرفة ان كانت ممتلئة ام لا
 باستدعاء *Isfull* كود
 يرجع قيمه 0 ان كانت ممتلئة وإن لم تكن يزيد
 قيمه *rear* واحد ويخزن القيمة المراد اضافتها
 ويرجع قيمة 1 ليدل أنه تمت الإضافة بنجاح



```
template <class T>
```

```
int Queue<T>::deQueue(T& item)
```

```
// If (queue is not empty) the front of the queue has been removed and returned
```

```
{ if (isEmpty())
```

```
    return 0;
```

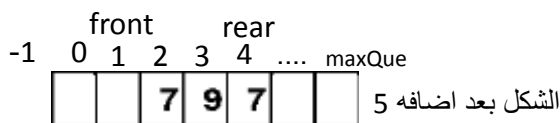
```
    front++;
```

```
    item = items[front];
```

```
    return 1;}
```

بعد اضافة العناصر وأردنا حذف إحداها ..
 يتم الحذف من البداية " اول عنصر يضاف اول عنصر يحذف"
 عند الحذف لا بد من معرفة ان كانت فارغة ام لا
 باستدعاء *IsEmpty* كود

يرجع قيمه 0 ان كانت فارغة وإن لم تكن
 يزيد قيمه *front* واحد ويحذف القيمة
 الموجودة في البداية ويرجع قيمة 1 ليدل أنه تم الحذف بنجاح



مرات وحذف مرتين

* لا يشترط أن يتم الحذف بعد إضافة جميع العناصر

* في عملية الحذف يتم تخزين القيمة المحذوفة في متغير *iteam*

* عملية الحذف هي شكلية لأنه يتم انقاص من حجم *queue* لأن حدودها هي ما بين *front* & *rear*

```
template <class T>
void Queue<T>::printAll( ) const
{for( int i=front+1; i<=rear; i++)
    cout<<iteam[i]<< “ “;
    cout<<endle;}
```

الطباعة تتم من المكان الذي يشير إليه *front*، لأنه فعليا هو يشير لأول عنصر في الـ *queue* ويتنتهي عند *rear* لأنه يشير إلى آخر عنصر في الـ *queue* من خلال جملة *for* تبدأ من عند *front+1* وتنتهي عند *rear* نهاية الـ *queue* بوجود *cout* داخلها

In class :

```
template <class T>
int Queue<T>::deQueue(T& item)
// If (queue is not empty) the front of the queue has been removed and returned
{ if (isEmpty())
    return 0;
    front++;
    item = items[front];
    return item;}
```

يمكن طباعه عنصر واحد فقط من خلال جملة طباعه تطبع ناتج حذف العنصر .. مع مراعاة ان يكون كود الحذف يرجع القيمة وليس 0,1

In main :

```
Cout<<deQueue();
```

واذا أردنا مثلا طباعه آخر قيمه يتم حذف جميع العناصر التي قبلها ثم طباعتها

Circular Queue

```
#include <iostream>
using namespace std;
template <class T>
class Queue { public:
    Queue(int max=100);
    ~Queue() { delete [] items; }
    int isEmpty() const { return (rear == front); }
    int isFull () const { return ((rear +1) % maxQue == front); } //r+1=max
    void makeEmpty() { front = rear = maxQue - 1; } //f=r=-1;
    int enqueue(T newItem);
    int dequeue(T& item);
private:
    int front, rear;
    T* items;
    int maxQue; };
/*****
```

```
template <class T>
    Queue<T>::Queue(int max){
    maxQue = max+1 ;
    front = rear = maxQue-1; // -1
    items = new T[maxQue];}
```

عند تعريف *queue* جديدة نعطي قيمة الماكس اكبر ب 1 من

الحجم الأصلي وقيمة r, f قيمه -1 او قيمه الماكس

```
template <class T>
int Queue<T>::enqueue(T newItem)
// If (queue is not full) newItem is at the rear of the queue
{ if (isFull())
    return 0;
    rear = (rear +1) % maxQue; // rear++;
    items[rear] = newItem;
    return 1;}
```

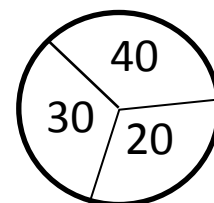
عند الاضافة يتم فحص انك كانت ممثلة ام لا ثم
انتقال مؤشر *front* خطوة للأمام وتخزين القيمة

```
template <class T>
int Queue<T>::dequeue(T& item)
// If (queue is not empty) the front of the queue has been removed and returned
{ if (isEmpty()) return 0;
    front = (front + 1) % maxQue; // front++;
    item = items[front];
    return 1;}
```

عند الحذف يتم فحص انك كانت فارغة ام لا ثم
انتقال مؤشر *rear* خطوة للأمام وحذف القيمة

```
/*****
int main(){
    Queue <int> line(3);    int x;
    line.enqueue(10); f=0 , r=3
    line.enqueue(20); f= 1 , r=3
    line.dequeue(x); f=1 r=0 x=10 يتم تخزين القيمة المحذوفة في المتغير x
    line.enqueue(30); f=2 r=0
    line.enqueue(40); f=0 r=0

    line.dequeue(x); cout << x << " "; \\ 20
    line.dequeue(x); cout << x << " "; \\ 30
    line.dequeue(x); cout << x << " "; }\\ 40
```



Counter Circular Queue

```
#include <iostream>
using namespace std;
template <class T>
class Queue { public:
    Queue(int max=100);
    ~Queue()    { delete [] items; }
    int isEmpty() const { return (count == 0); } // rear == front
    int isFull () const { return coun == maxQuaue); }
    void makeEmpty()  { front = rear = - 1; }      //not neaded
    void printAll ( ) const ;
    int enqueue(T newItem);
    int dequeue(T& item);
private:
    int front, rear, count;
    T* items;
    int maxQue; };
/*****/
template <class T>
    Queue<T>::Queue(int max){
maxQue = max ;
front = rear = -1
items = new T[maxQue];
count=0}

template <class T>
int Queue<T>::enqueue(T newItem)
// If (queue is not full) newItem is at the rear of the queue
{ if (isFull())
    return 0;
rear = (rear +1) % maxQue; // rear++;
items[rear] = newItem;
count++;
return 1;}

template <class T>
int Queue<T>::dequeue(T& item)
// If (queue is not empty) the front of the queue has been removed and returned
{ if (isEmpty())    return 0;
front = (front + 1) % maxQue; // front++;
item = items[front];
count--;
return 1;}

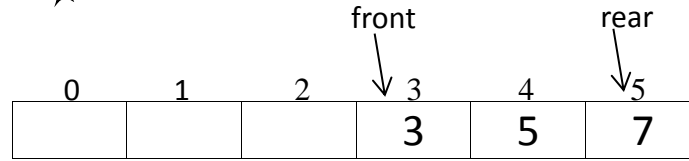
template <class T>
void Queue<T>::printAll ( )const
for ( int i=0 ; i<count ; i++)
    cout<<<item [( i+front+1 ) %maxQue] << "  ";
cout<<endle;}

/*****/
```

```
int main(){
    Queue <int> line(3);    int x;
    line.enqueue(10); f=0 , r=3
    line.enqueue(20); f= 1 , r=3
    line.dequeue(x); f=1 r=0  x=10
    line.enqueue(30); f=2  r=0
    line.enqueue(40); f=0  r=0

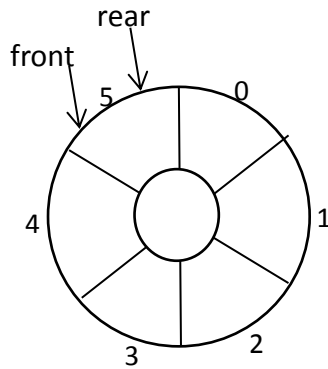
    line.dequeue(x); cout << x << " "; \\ 20
    line.dequeue(x); cout << x << " "; \\ 30
    line.dequeue(x); cout << x << " "; }\\ 40
```

6~Circular queue



بعد اجراء 6 عمليات اضافة و 3 عمليات حذف نصيب قيمة $front=3$, $rear=length-1$ اي انه اذا اردنا اجراء عملية اضافة 7 سيعتبر انها مملئة لتحقيق الشرط $rear=length-1$ على الرغم من وجود 3 اماكن فارغة , لذلك نحتاج الى طريقة تعمل على ارجاع المؤشر $rear$ الى البداية....

نعمل التفاف للشكل ليصبح دائرة



$front == rear \rightarrow \text{empty}$

or $front == rear == -1 \rightarrow \text{empty}$

$(rear + 1) \% \text{Max queue} == front \rightarrow \text{full}$

الفرق الوحيد بين $linear$ و $circular$ هي طريقة الاضافه على $front$ و $rear$ ومعرفة متى تكون مملئة ومتى تكون فارغة

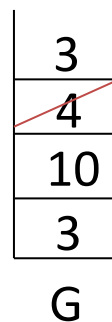
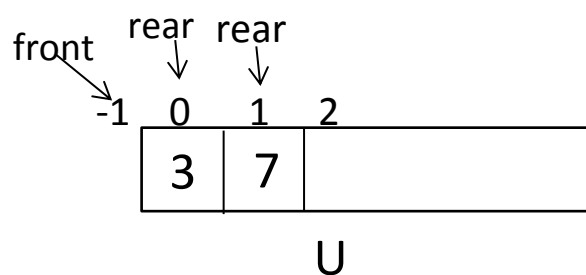
7-counter Circular queue

حل اخر لمشكلة تعارضه انها فارغة و مملئة في نفس الوقت هو استخدام العداد تزداد قيمته 1 مع كل اضافه وتنقص 1 مع كل حذف, وعندما يصبح $count = \text{max} - 1$ تكون $queue$ قد امتلأت و $count = 0$ فارغة

Some example (stack.. List.. Queue..)

1-write the output

```
void main(){  
Stack <int> G;  
G. Push (3);  
G. Push (10);  
G. Push (7);  
U. Enqueue (32);  
U. Enqueue (G. Pop());  
G. Push(u.dequeue());}
```



2-in unsorted list. . Write program to remove item.

(x index item to be deleted)

نذكر عند حذف عنصر من unsorted list نعمل على استبدال القيمة المحذوفة بأخر قيمة في الlist

Sol : info[i]=info[--length];

4- on which principle does stack work?

Sol : LIFO / FILO

5-Queue follows the order

Sol : FIFO... LILO

Team
ZeroOne

6- You have a circular queue the max number of element have added 9 times and twice been deleted

what is the value of rear and front?

الإضافة الأولى $f=7$. $R=0$

الإضافة السادسة $f=7$. $R=5$

الإضافة الثانية $f=7$. $R=1$

الإضافة السابعة $f=7$. $R=6$

الإضافة الثالثة $f=7$. $R=2$

الإضافة الثامنة $f=7$. $R=7$

الإضافة الرابعة $f=7$. $R=3$

الإضافة التاسعة $f=7$. $R=7$

لا تتم الإضافة
لأنها full

الإضافة الخامسة $f=7$. $R=4$

الجواب النهائي $F=1$. $R=7$ ثاني حذف $f=0$. $R=7$ أول حذف

7- Write programme to find the minimum number in a templated static stack.

Sol : `void stack <t, max - item> :: Max () const`

```
{ int min = info [0];  
for( int i=1 ; i<length ; i++)  
    if( info [i] <min)  
        min= info [i] m;  
cout << "min= " << min;}
```

Team
ZeroOne

8-write the output , user input are as folowing

9 13 7 48 -4 73 50 80 12

```
Int main( ){  
  
    Queue<int> Q ;  
  
    Int i=2;  
  
    While(i>0){  
  
        Cin>>i;  
  
        If (i%2!=0)  
  
            Q.Enqueue(i);  
  
        Cin>>i;  
  
    }  
  
    Q.Print_All();}
```

Sol: The output is: 9 7 1

9.....method is used to remove element from a stack .

Sol: pop()

Linked list

```
#include <iostream.h>

template <class T>
class node
{
public:
    node(T d=0, node* n=NULL)
    {data=d; next=n;}
    node* next;
    T data;
};

template <class T>
class LinkList
{
public:
    LinkList() head=new node<T>(0);
    int isEmpty () const {return (head->data == 0);}
    int addItem (T item);
    int delItem (T item);

private:
    node <T>*head;
};
/*****

template <class T>
int LinkList<T>::addItem (T item)
{
    Try{
        node<T>* newItem = new node<T> (item);
        for (node<T>* i=head->next; i != NULL; i=i->next);
        i->next=newItem;
        (head->data++);
        return 1;
    }
    catch(...)
    {
        return 0;
    }
}

template <class T>
int LinkList<T>::delItem (T item)
{
    if (isEmpty ( ) )
        return 0;
    try
    {
        node<T>* i,*j=head;
        for (i=head->next; i != NULL; i=i->next)
        {
            if (i->data == item)
                break;
            j=i;
        }
        j->next=i->next;
        delete i;
        return 1;
    }
    catch(...)
    {
        return 0;
    }
}
```

```
/*  
*****  
*/
```

```
void main()
```

```
{
```

```
    LinkedList <int> l1;
```

```
    LinkedList <char> l2;
```

```
    LinkedList <float> l3;
```

```
    l1.addItem(1); l1.addItem(2);
```

```
    l1.addItem(3); l1.addItem(4);
```

```
    l1.addItem(5); l1.addItem(6);
```

```
    l1.removeItem (4);
```

```
}
```

Linked list unsorted

```
#include <iostream.h>

template<class ItemType>
struct node
{
    ItemType info;
    node* next;
};

template <class ItemType>
class UnsortedLinkList
{
private:
    int length;    //optional
    node<ItemType>* listData;
    node<ItemType>* currentPos;
public:
    UnsortedLinkList( ) { length = 0; listData = NULL; }
    ~UnsortedLinkList ( ) {makeEmpty() ;}
    void resetList ( ) { currentPos = NULL; }
    int lengthIs() const { return length ; }
    int isFull() const;
    void makeEmpty() ;
    int retrievalItem(ItemType item) ;
    void getNextItem (ItemType& item);
    int insertItem (ItemType item);
    int deleteItem (ItemType item);
};

template<class ItemType>
int UnsortedLinkList<ItemType>::isFull() const
{
    node<ItemType>* location;
    try{
        location = new node<ItemType>;
        delete location;
        return 0;
    }
    catch (...){ return 1;}
}

template <class ItemType>
void UnsortedLinkList<ItemType>::makeEmpty()
{
    node<ItemType>* tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}

template <class ItemType>
int UnsortedLinkList<ItemType>::retrievalItem(ItemType item)
{
    node<ItemType>* location;
    for (location = listData; location != NULL; location = location->next)
        if (item == location->info)
            return 1;
    return 0;
}
```

```

template <class ItemType>
int UnsortedLinkList<ItemType>::insertItem(ItemType item)
{
    if (isFull ( ) )
        return 0;
    node<ItemType>* location;
    location = new node<ItemType>;
    location->info = item;
    location->next = listData;
    listData = location;
    length++;
    return 1;
}

template <class ItemType>
int UnsortedLinkList<ItemType>::deleteItem(ItemType item)
{
    if (listData==NULL)
        return 0;
    node<ItemType>* location = listData;
    node<ItemType>* tempLocation;
    if (item == listData->info)
    {
        tempLocation = location;
        listData = listData->next; // Delete first node.
    }
    else
    {
        while (!(item==(location->next)->info))
            location = location->next;
        //Delete node at location->next.
        tempLocation = location->next;
        location->next = (location->next)->next;
    }
    delete tempLocation;
    length - - ;
    return 1;
}

template <class ItemType>
void UnsortedLinkList<ItemType>::getNextItem(ItemType& item)
{
    if (currentPos == NULL)
        currentPos = listData;
    else
        currentPos = currentPos->next;
    item = currentPos->info;
}
/*****/
void main()
{
    UnsortedLinkList <int>  I1;
    //UnsortedLinkList <char> I2;
    //UnsortedLinkList <float> I3;

    I1.insertItem(1); I1.insertItem(2);
    I1.insertItem(3); I1.insertItem(4);
    I1.insertItem(5); I1.insertItem(6);

    I1.deleteItem (4);
}

```

Linked list sorted

```
#include <iostream.h>

template <class ItemType>
class sortedLinkedList
{
private :
    struct node
    {
        ItemType info;
        node* next;
    };
    int length;    //optional
    node* listData;
    node* currentPos;
public:
    sortedLinkedList() { length = 0; listData = NULL ; }
    ~sortedLinkedList() { makeEmpty ( ) ;}
    void resetList ( ) {currentPos = NULL;}
    int lengthIs() const { return length ;}
    int isFull() const;
    void makeEmpty() ;
    int retrievalItem(ItemType item) ;
    void getNextItem (ItemType& item);
    int insertItem (ItemType item);
    int deleteItem (ItemType item);
    void print() const;
};

template<class ItemType>
int sortedLinkedList<ItemType>::isFull() const
{
    node* location;
    try{
        location = new node;
        delete location;
        return 0;
    }
    catch (...){ return 1; }
}

template <class ItemType>
void sortedLinkedList<ItemType>::makeEmpty()
{
    node* tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}

template <class ItemType>
int sortedLinkedList<ItemType>::retrievalItem(ItemType item)
{
    node* location;
    for (location = listData; location != NULL; location = location->next(
        if (item == location->info)
            return 1;
    return 0;
}
```

```

template <class ItemType>
int sortedLinkedList<ItemType>::insertItem(ItemType item)
{
    if (isFull ( ) )
        return 0;

    node* location;
    node* befor=listData, *after=listData;

    for (after=listData;after!=NULL;after=after->next)
        if (after->info > item)
            break;
        else
            befor=after;

    location = new node;
    location->info = item;
    location->next = after;

    if(listData==NULL)
        listData=location;
    else
        befor->next=location;
    length++;
    return 1;
}

template <class ItemType>
int sortedLinkedList<ItemType>::deleteItem(ItemType item)
{
    if (listData==NULL)
        return 0;
    node* location = listData;
    node* tempLocation;
    if (item == listData->info)
    {
        tempLocation = location;
        listData = listData->next; // Delete first node.
    }
    else
    {
        while (!(item==(location->next)->info))
            location = location->next;
        //Delete node at location->next.
        tempLocation = location->next;
        location->next = (location->next)->next;
    }
    delete tempLocation;
    length - - ;
    return 1;
}

template <class ItemType>
void sortedLinkedList<ItemType>::getNextItem(ItemType& item)
{
    if (currentPos == NULL)
        currentPos = listData;
    else
        currentPos = currentPos->next;
    item = currentPos->info;
}

```



```
template <class ItemType>
void sortedLinkedList<ItemType>::print() const
{
    node* location;
    for (location = listData; location != NULL; location = location->next)
        cout <<location->info;
}
/*****/
void main()
{
    sortedLinkedList <int> l1;
    //sortedLinkedList <char> l2;
    //sortedLinkedList <float> l3;

    l1.insertItem(1); l1.insertItem(3);
    l1.insertItem(6); l1.insertItem(4);
    l1.insertItem(5); l1.insertItem(2);

    l1.deleteItem (4);

    l1.print ( ) ;

}
```

8~ link list

Array:

*Fixed size حجم ثابت

☹️ *Sequential العناصر مخزنة بشكل متتالي

*Bad removal عملية الحذف تحتاج وقت وخاصة اذا كانت طويلة

*Bad insertion عملية الإضافة تحتاج لوقت خاصة الإضافة بالترتيب

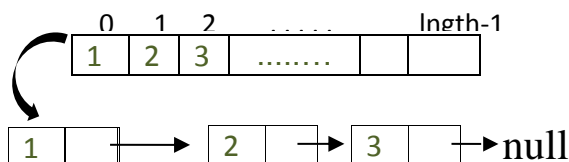
😊 * Fixed fast access الوصول لأي عنصر أو جميعها كانت بسهولة مه خلال

جمل بسيطة مثلا لطباعة العنصر الرابع (cout<<info [3])

*كانت المشاكل التي تواجهنا في ال (array) محدودة الحجم أي انه ثابت لا يمكنه الزيادة عليه بعد إعطائه حجم أولي , و أن العناصر يجب أن تكون متتالية في الذاكرة , أي أنه إذا أردنا إنشاء array طولها 100 ولا يوجد في الذاكرة سوى 99 عنصر متتالي , مع وجود مساحات في الذاكرة تتسع لأكثر مه عنصر لكنهم غير متتاليات فلا يمكنه إنشاؤها , بالإضافة لعملية الحذف و الإضافة وخاصة بالترتيب
أما الحسنة فكانت بسرعة الوصول للعناصر

*لذلك كان الحل هو استخدام link لحل مشاكل محدودية الحجم و تتالي العناصر و الإضافة و الحذف لكنه تبقى مشكلة واحدة أنه لا يتم الوصول للعناصر بسهولة .

ال link list عكس ال array في كل شيء



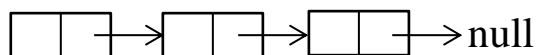
→ Array

→ link list

شكل العناصر مختلف , العناصر في link list عبارة عن node يتكون مه جزئين الأول للقيمة المراد تخزينها data و الثاني مؤشر reference للعنصر الذي يليه.

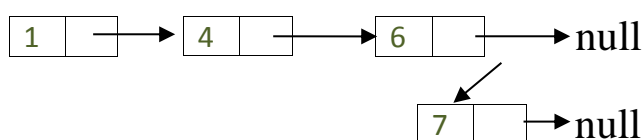
- في link list لا يشترط إعطاء حجم ثابت في البداية لطول أي أنه تتم الزيادة و النقصان على الطول متى أردنا كل ما عليه إنشاء node جديد و تغيير مكان المؤشر.

- لا يشترط أن تكون العناصر متتالية في الذاكرة، فيمكنه أن يكون عنوان الأول 301 و الثاني 310 و الثالث 400 .



First node has the data and reference to the next node "second node"

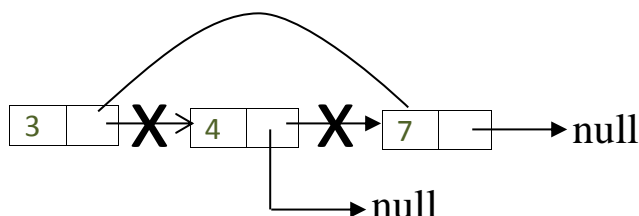
★الإضافة :



عند إضافة 7

- إنشاء new node و تخزين القيمة بداخله. ثم نجعل مؤشر آخر عنصر الذي يشير لـ null يشير إليه و مؤشره يصعب يشير للـ null

★الحذف :



عند حذف الـ node الذي يحمل القيمة 4

- نجعل مؤشر الـ 3 يشير إلى 7 فقط ، و يمكنه جعل مؤشر القيمة المراد حذفها تشير إلى null.

** كما أنه يوجد reference لأول عنصر حتى نتحكم به معرفة مكان وجود الـ list ويسمى head .

عند استدعاء العنصر الرابع مثلا لا بد من المرور بالعناصر التي قبلها ، لأن كل عنصر ينقلنا للعنصر الذي يليه إلى أن نصل للعنصر الذي نريد . و كل عنصر هو node .

طريقة كتابة الكودات قد تختلف لكنها تبقى صحيحة ما دامت تؤدي لنتيجة واحدة



عند كتابة برنامج link list لابد منه وجود 2 classes الأول نستخدمه في إنشاء nodes التي تكون عناصر في الـ list والثاني في إنشاء الـ list .

```
Template < class T>
Class node
{ public:
Node (Td=0 , node*n=null)
{data=d ; next=n;}
Node*next;
T data;}
```

عند إنشاء أي نود جديد نخزن قيمة ابتدائية و لتكن (0) في القسم الأول و نوعها T حتى نتمكن من تخزين أي نوع بياني.
أما القسم الثاني ,,next,, يكون نوعه node لأنه يشير إلى node كما قلنا سابقا أن المؤشر يكون نوعه من نوع ما يشير إليه و هو يشير إلى node , فيكون بداية يشير إلى null .
-في برامج أخرى تم تعريف class node باستخدام struct

عادة ما يتم تعريفها private , لكن هنا تعرف في public حتى نتمكن من الوصول إليهم من class آخر .

*كتابة class الـ link list

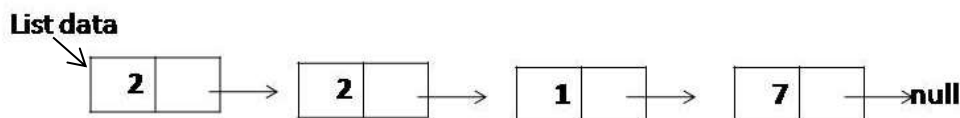
1-محتاج إلى constructor يتم فيه تمهيد الطول لـ 5 و المؤشر الذي يشير إلى عنصر list data or head

```
Unsorted linklist () {length=0 ; list data = null;}
```

يتم تعريف النوع البياني لـ length على أنه int و list data على أنه node في private أما نحن الآن نكتب بـ public

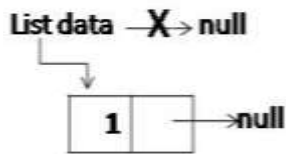
2- كما يحتاج destructor و يتم فيه استدعاء كود make empty الذي يجعل المصفوفة فارغة .

```
~unsorted linklist () {make empty();}
```

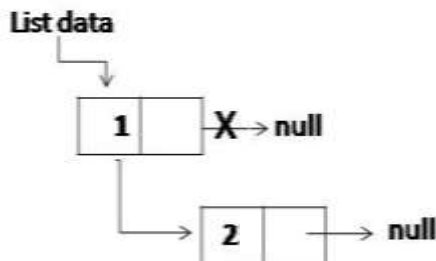


```
Void make empty()
{While (list data !=null)
{ node<T>*temp;
Temp = list data;
List data=list data -> next;
Delete temp;}
length=0;}
```

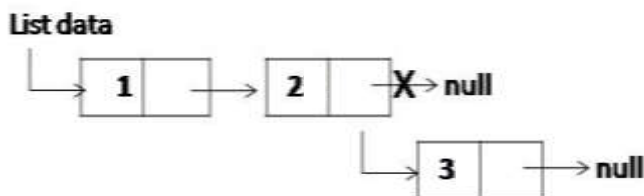
لنفرض انه لدينا هذه المصفوفة و نريد حذفها وليس لدينا مرجع نصل من خلاله للمصفوفة إلا list data لذلك نستخدمه في الحذف , نحتاج نقل مؤشره من عنصر لآخر حتى نهاية list , أي حتى الوصول لـ null
(إنشاء مؤشر من نوع node وجعله مساوي للـ list data ثم أحرك مؤشر list data للعنصر الثاني و احذف الـ node الجديد و نكرر العملية ما دامت list data لا تشير إلى null)



-عند إضافة العنصر الأول يتم إنشاء Node وتخزين القيمة فيه وجعل مؤشر list data يشير إلى null كما نجعل مؤشر list data يشير للعنصر الأول إذا كان يشير ل null (هذا فقط للعنصر الأول)



-عند إضافة العنصر الثاني يتم إنشاء node وتخزين القيم فيه وجعل مؤشر يشير إلى null وجعل مؤشر آخر عنصر يشير إليه.



-عند إنشاء العنصر الثالث يتم إنشاء node وتخزين القيمة فيه وجعل مؤشر يشير إلى Null وجعل مؤشر آخر عنصر يشير إليه .

*العمليات التي أجريت على كل عنصر أضيف :

1. إنشاء node جديد.
2. تخزين القيمة فيه وجعل مؤشر يشير إلى null.
3. إذا كان أول عنصر أي أن list data تشير إلى null فنريد جعلها تشير للعنصر الأول وإذا لم تكن نريد أن أصل لآخر عنصر وأجعل مؤشره يشير للعنصر الجديد وذلك عبر جملة for توصلنا لنهاية list دون أن تطبق أمر آخر

← لا يمكنني تحريك list data من مكانها لأنه ستضيع المصفوفة , لذلك يتم إنشاء node مؤشره يشير إلى ما يشير إليه list data

```
int insertItem(T item)
{node <T> * new node ;
newnode = newnode<T>;
newnode -> next=null;
list data = new node;
else
{for(node<T>*I;(i-> next)!=null ; i=i->next) ;
i-> next = new node;}
length ++;
return 1;}
```

```

int insertItem(T item)
{
    node <T> * new node ;
    new node = new node<T>;
    new node -> info=item;
    new node -> next=list data;
    list data = new node;
    length ++;}

```

← **العمليات السابقة تضيف العنصر في النهاية , يمكن أن تكون الإضافة بالبداية :**
 1. إنشاء node جديد.

2. تخزين القيمة فيه
 3. جعل مؤشره يشير لما يشير إليه list data
 4. جعل list data تساوي الـ node الجديد.

قبل الإضافة لا بد من معرفة إن كان بالإمكان الإضافة , سابقاً كنا نقارن بين الطول و بين عدد العناصر التي أضفناها لكنه هنا لا يوجد لدينا طول لنقارن , لكنه يصعب طريقته جديد لمعرفة ما إذا كان يمكن الإضافة أو لا باستخدام كود try , catch .

*يضيف اذا كان هناك وسع في الذاكرة .

```

int is full ( ) const
{
    node <T> *location;
    Try
    {
        location = new node <T> ;
        delete location ;
        return 0;
    }
    Catch (...) { return 1;}}

```

يتم إنشاء نود جديد *location*
 يدخل جملة *try* اذا طبقها بدون وجود أخطاء ((الأخطاء التي تحدث بسبب إنشاء عنصر لا يوجد متسع له في الذاكرة)) يحذف النود الجديد لأنه لا داعي له و إنما كان مجرد تجربه و يرجع 0 أما إذا واجه أخطاء يخرج من جملة *try* وينفذ جملة *catch* ويرجع 1 على أنها ممتلئة .
 دون أن يتوقف البرنامج

← يتم استدعاء كود *if full()* في بداية جملة الإضافة

لأنه يرجع قيمه

```

int retrieve Item (t item)
{
    node <T> * location;
    for ( location = listdata ; location != null;
        location = location -> next )
    {
        if (item==location->info)
        {
            return t;
        }
    }
    return 0;
}

```

للتنقل بين العناصر
 لمقارنة العناصر
 يرجع 1 إن وجدت
 يرجع 0 إن لم توجد

البحث عن عناصر

عند البحث عن عنصر سنحتاج إلى مؤشر للتنقل بين العناصر *new node* ثم مقارنة بين القيم التي أبحث عنها و القيم المخزنة في المصفوفة .

القيمة المراد حذفها

int delete Item (T item)

{if (list data == null) ← إذا كانت فارغة

return 0;

node <T> *location = list data;

node <T> *Temp location;

if(item==list data → info) {
temp location = location;
list data= list data→next;} } إذا كانت القيمة في أول node

else

{while (!(item==(location →next)→info)) } بحث عن العنصر
location=location→next; يمكن استخدام جملة for

templocation=location→next;

location → next=(location→next)→next;

}

delete templocation;

length --;

return 1;}

حذف عناصر

عند حذف قيمة نبحث عنها في العنصر الأول إن لم تكن موجودة ينتقل للذي يليه و إذا وجدها يجعل مؤشر الذي قبلها يشير للذي يليه لذلك نحتاج إلى مؤشرين الأول يشير للنود المراد حذفه و الثاني للنود قبل المراد حذفه .

Team
ZeroOne

void print () cost

{ if (list data1==null)

for (node<T> *i=list data; i!=null; i=i→ next)

cout<<i→ data;

}

طباعة عناصر

نحتاج في الطباعة إلى مؤشر ينتقل بين العناصر ثم يطبعها .

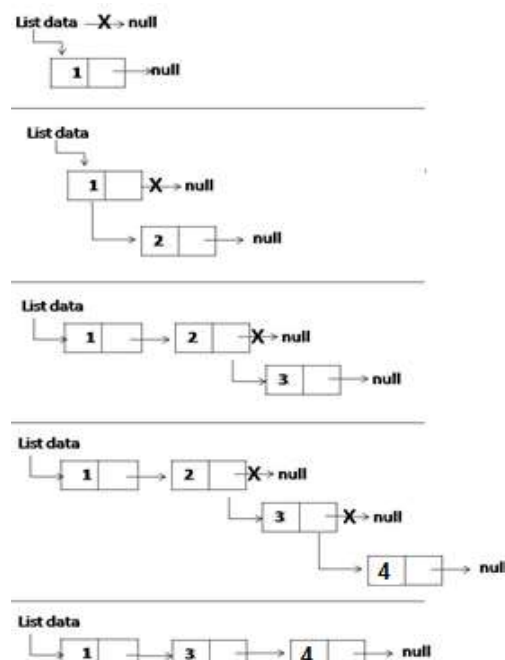
void main () {

unsorted link list <int> l1;

L1.insert item(1) ; L1insertitem(2);

L1.insert item(3) ; L1insertitem(4);

L1.delete item(2);}



الإضافة بالترتيب stored

```
int insert item (T item)
```

```
(if (is full() )
```

```
return 0;
```

```
node*new node;
```

```
node*befor=list data;
```

```
node*after=listdata;
```

```
for(after=listdata;after!=null;after=after → next)
```

```
if(after→info>item)
```

```
break;
```

```
befor = after;
```

```
new node =new node<T>;
```

```
new node → info = item;
```

```
new node → next = after;
```

```
if (list data = null)
```

```
list data = new node;
```

```
else
```

```
befor → next = new node;
```

```
length++;
```

```
return 1;
```

```
}
```

عندما أردنا حذف عنصر استخدمنا مؤشرين أحدهما يشير للعنصر المراد حذفه و الآخر يشير للذي قبله , وكذلك عند الإضافة بالترتيب لابد من وجود مؤشرين أحدهما للعنصر و الآخر للذي يسبقه.

- اذا كان العنصر الأول لا مشكلة في إضافته
- اذا كان العنصر الثاني الذي نريد إضافته أكبر من العنصر الأول نجعل مؤشر الأول يشير إليه و هو يشير لل *null* و اذا كان أقل منه نجعله مكان الأول و يصبح هو يشير للعنصر الذي كان في البداية و

ال *list data* يشير إليه

☺ مثال إضافة (4 7 1 3 2) ابتداء ب2

عند إضافة 2 يتم إنشاء *node* 3 أحدهم للعنصر الجديد واثنان يساعدان في العملية

List data = befor & after

نذهب لجملة *for* يخرج منها بدون عمل أي خطوة لعدم تحقق الشرط لأن ال *list* ما زالت فارغة, نخزن بالنود القيمة و يشير إلى ما يشير إليه *after*

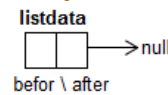
يذهب لجملة *if* يتحقق الشرط و تصبح *list data* تشير ل 2

عند إضافة 3 يتم إنشاء *node* 3 يذهب لجملة *for* لا يتحقق

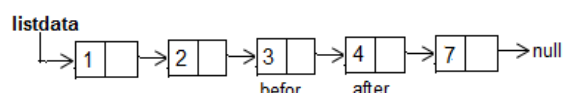
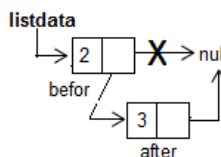
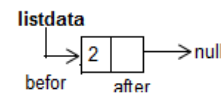
الشرط في المرة الأولى $3 > 2$ يخرج من الجملة في المرة الثانية

لعدم تحقق شرط ال *for* يخزن القيمة ومؤشره يشير إلى *after* .

بجملة *if* لا يتحقق الشرط فينتغير مؤشر *befor* ليشير إلى العنصر الجديد



← ويستمر لباقي العناصر



Linked stack

```
#include <iostream.h>
template <class T>
struct node
{
    T info;
    node* next;
};
template <class T>
class StackType
{
public:
    StackType( ) {topPtr = '0\'; }
    ~StackType ( );
    int Push( T );
    int Pop ( );
    T Top () const;
private:
    node<T>* topPtr;
    int IsEmpty() const { return (topPtr == '0\'; ) ; }
    // int IsFull () const;
};
template <class T>
StackType<T>::~~StackType()
{
    node<T>* tempPtr;
    while (topPtr != '\0' )
    {
        tempPtr = topPtr;
        topPtr = topPtr->next;
        delete tempPtr;
    }
}

template <class T>
int StackType<T>::Push(double newItem)
{
    Try{
        node<T>* location;
        location = new node<T>;
        location->info = newItem;
        location->next = topPtr;
        topPtr = location;
        return 1;
    }
    catch(...)
    {return 0;}
}

template <class T>
int StackType<T>::Pop()
{
    if (IsEmpty ( ))
        return 0;
    else
    {
        node<T>* tempPtr;
        tempPtr = topPtr;
        topPtr = topPtr->next;
        delete tempPtr;
        return 1;}}

template <class T>
double StackType<T>::Top() const
{

```

```

if (IsEmpty ( ))
    return 0;
else
    return topPtr->info;
}

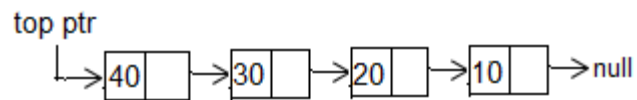
```

```

void main()
{
    StackType <int> s;
    s.Push(10); s.Push(20) ;
    s.Push(30); s.Push(40);

    cout << s.Top() << ' ' ; \\40
    cout << s.Top() << ' ' ; \\30
    cout << s.Top() << ' ' ; \\20
    cout << s.Top() << ' ' ; \\10
}

```



```

*/
int StackType::IsFull() const
//Returns true if there is no room for another node object
//on the free store and false otherwise.
{
    node<T>* location;
    try
    {
        location = new node<T>;
        delete location;
        return 0;
    }
    catch(...)
    {
        return 1;
    }
}
/*

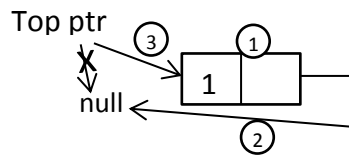
```

8~ linked stack

لا تختلف مبادئ *stack* عن *linked stack*

Push → إضافة *pop* → حذف *LIFO* → last in is the first out

عند إنشاء *linked stack* جديدة يتم تهيئة *node* يشير لأول عنصر *topptr* و يشير ل *null*
عند إضافة العنصر الأول يتم إنشاء *node* تخزن في القسم الأول القيمة و الثاني "المؤشر" يشير لما يشير إليه *topptr* ثم جعل *topptr* توضح للعنصر الجديد
عند إضافة العنصر الثاني يتم إنشاء *node* جديد نخزن في القسم الأول القيمة و القسم الثاني يشير لما يشير إليه *topptr* ثم جعل *topptr* توضح للعنصر الجديد



← تبقى هذه العملية تتكرر بكل إضافة

ترجمته في البرنامج ←

إنشاء *linked stack* وتهيئتها "constructor"

Stack type () { topptr = '\0'; } '\0' → NULL

إضافة عناصر :-

int push (Titem)

*{node <T> *location = new node<T>; → إنشاء node*

location → info = item; → تخزين القيمة

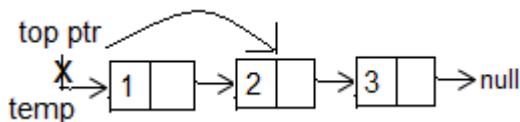
location → next = topptr; → جعله يشير إلى ما يشير له topptr

topptr = location ; return 1; } → جعل ال topptr يشير إلى النود الجديد

الحذف :-

- لا يمكن حذف سوى آخر عنصر تمت إضافته حسب المبدأ (*LIFO*)

- للحذف نستخدم *pop*



- عند إجراء عملية الحذف يتم إنشاء *node* يشير لما يشير إليه *topptr*

إليه *topptr* للعنصر الحالي

Int pop ()

{if (is empty ())
Return 0;

السؤال إذا كانت فارغة و
إرجاع قيمة 0 إن كانت فارغة

Int isEmpty () const

{return(topptr=='\0');}

Else

{node <T>*temp; → إنشاء نود جديد

Temp=topptr; → جعله يشير لل *topptr* الأول

Topptr = topptr → next; → تحريك مؤشر ال *topptr* خطوة

Delete temp; → حذفه فلا يوجد سبب لوجوده

Return 1; → إرجاع 1 على أنه تمت العملية بنجاح

}}

عند إنشاء *stack* لابد من حذفها

Constructor ↔ destructor

الحذف هي عملية *pop* متكررة إلى أن تصل قيمة

topptr إلى '\0'

في عملية *pop* يتم إنشاء *node* و جعله يشير ل

topptr ثم تحريك مؤشر *topptr* خطوة و حذف ال

node

نحتاج لجملة *while* أو *for* لتكرير عملية ال *pop*.

طباعة عناصر :-

1) Void print () const

{ for (node <T>*I = topptr ; i!= '\0'; i = i →
next)

Cout << i → info;}

2) Void print () const

{while (topptr != '\0')

{cout<<pop(x);

}}

T pop ()

{ if (! Is empty ())

{Tx;

X=topptr → info;

Node<T>*temp;

Temp = topptr;

Topptr = topptr → next;

Delete temp;

}}

برنامجين مقترحين في طباعة عناصر *stack* ولا
يعني أنهم الوحيدين ☺

Linked Queue

```
#include <iostream.h>

template <class ItemType>
struct node
{
    ItemType info ;
    node* next;
};

template <class ItemType>
class Queue
{
public:
    Queue () front = rear = NULL;
    ~Queue () makeEmpty () ;
    int isEmpty()const { return (front == NULL) ;}
    int isFull() const;
    int enqueue(ItemType );
    int dequeue(ItemType& ) ;
    void makeEmpty() ;
private:
    node<ItemType>* front;
    node<ItemType>* rear;
};

/*****/
template <class ItemType>
void Queue<ItemType>::makeEmpty()
{
    node<ItemType>* tempPtr;
    while (front != NULL)
    {
        tempPtr = front;
        front = front->next;
        delete tempPtr;
    }
    rear = NULL;
}

template<class ItemType>
int Queue<ItemType>::isFull() const
{
    node<ItemType>* location;
    try{ location = new node<ItemType>;
        delete location;
        return 0;
    }
    Catch (...) {return 1;}
}

//Adds newItem to the rear of the queue.
//Pre: Queue has been initialized.
//Post: If (queue is not full), newItem is at the rear of the queue;
//otherwise, a FullQueue exception is thrown.
template <class ItemType>
int Queue<ItemType>::enqueue(ItemType newItem)
{
    if (isFull ( ))
        return 0;
    else
    {
        node<ItemType>* newNode;
        newNode = new node<ItemType>;
        newNode->info = newItem;
        newNode->next = NULL;
        if (rear == NULL)
            front = newNode;
        else
    }
```

```

        rear->next = newNode;
        rear = newNode;
    }
    return 1;
}
//Removes front item from the queue and returns it in item.
//Pre: Queue has been initialized
//Post: If (queue is not empty), the front of the queue has been
//removed and a copy returned in item;
//otherwise, an EmptyQueue exception is thrown.
template <class ItemType>
int Queue<ItemType>::deQueue(ItemType& item)
{
    if (isEmpty ( ) )
        return 0;
    else
    {
        node<ItemType>* tempPtr;
        tempPtr = front;
        item = front->info;
        front = front->next;
        if (front == NULL)
            rear = NULL;
        delete tempPtr;
    }
    return 1;
}

```

/*****

```

void main()
{
    Queue <int> q1;
    q1.enqueue(1);
    q1.enqueue(2);
    q1.enqueue(3);
    q1.enqueue(4);

    for (int x; !q1.isEmpty(); cout << x >> ' ' ;
        q1.deQueue(x);
    }
}

```

10- linked queue

في *Circular queue or liner queue* - *Enqueue* اضافة , *dequeue* حذف , *FIFO \ LILO* ,

- كنا نستخدم مؤشرين احدهم يتحرك مع كل اضافة "*rear*" والاخر يتحرك مع الحذف "*front*"
- وفي البداية كانت قيمتهم = 1 -

- ستبقى طريقة تحريك المؤشرين كما هي لكن القيمة البدائية تصبح *null*

Queue () {front =rear =null;}

-الاضافة:-

عند اضافة اول عنصرين يتم انشاء *node* نخزن فيها القيمة ونجعل مؤشرها لـ *null* , كما انه لابد من تحريك *rear* لتشير الى *new node* , لكن اذا كان اول عنصر لابد من جعل مؤشر *front* يشير الى العنصر الأول كذلك واذا لم يكن نجعل مؤشر العنصر المضاف قبل العنصر الحالي الذي كان اسمه *rear* ليشير الى العنصر الجديد

int enqueue(T time)

{ if (is full ()) → try & catch استدعاء *is full* وتم شرحه سابقاً بجملته

return 0;

else

*{ node <T> *new node= new node <T> ; →* انشاء *node*

new node → info = item ; new node → next = null ; →

if (rear == null) } تخزين القيمة وجعل المؤشر يشير لـ *null*

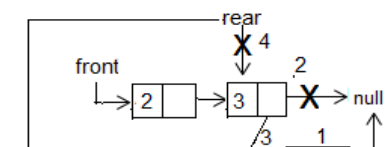
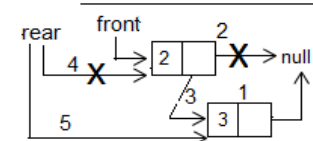
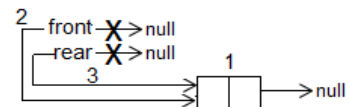
front= new node ; } اذا كان العنصر المضاف اول عنصر يجعل *front = new node*

else

rear → next = new node ; } اذا لم يكن يجعل المؤشر السابق "مؤشر آخر *node* اضعف " يشير للـ *node* الجديد

rear = new node; → تحريك مؤشر الـ *rear* مع كل اضافة

} return 1; }



* اذا اردنا ادخال العناصر بالترتيب فهو نفس

كود الاضافة بالترتيب في *link list*

وتم شرحه سابقاً

الحذف de queue

- عند الحذف يتحرك مؤشر front خطوة مع وجود متغير يخزن القيمة المحذوفة .
- اذا كانت عملية الحذف للعنصر الاخير فلا بد من تحريك مؤشر rear ليشير الى null دالاعلى انها فارغة .
- فليس من المنطق أن تشير front انها فارغة و rear تشير الى انها تملك عناصر .

```
T dequeue ( T and time)
{ if ( is empty () )
return o;
else
{ item = front → info ;
front = front → next ;
if ( front == null ) } → (لفحص ان كان اخر عنصر )
rear =null ;}
return item ; }
```

Make empty

جعلها فارغة بتكرار عملية الحذف dequeue لتصل الى اخر عنصر باستخدام جملة for or while

```
Void make empty ()
{ while ( front != null )
node <T> * temp ;
temp=front ;
front =front → next ;
delete temp;}
rear = null ;}
```

طباعة العناصر

```
1)Void print ( ) const
{ while ( front != null)
Cout<<dequeueer();}
```

```
2)Void print ( ) const
{ for( node<t> * i=front; i!= null ; i= i->next)
Cout<<i->info;}
```



```
Ex:Void main () {  
Queue <int> Q1;  
Q1.enqueue(1);  
Q1.enqueue(2);  
Q1.enqueue(3);  
Q1.enqueue(4);  
For (int x; ! Q1.isEmpty ( ) ; cout<<x<< ' ' )  
Q1.dequeue(x);}
```

Sol: For 1 : q1.isEmpty () → false

Q1>dequeue(x) → 1

Cout <<x → \\1

For 2 : q1.isEmpty () → false

Q1>dequeue(x) → 2

Cout <<x → \\2

For 3 : q1.isEmpty () → false

Q1>dequeue(x) → 3

Cout <<x → \\3

For 4 : q1.isEmpty () → false

Q1>dequeue(x) → 4

Cout <<x → \\4

For 5 : q1.isEmpty () → true

ينتهي البرنامج

Sorted circular Link list

```
#include <iostream.h>
template <class ItemType>
class sortedLinkList
{
private:
    struct node
    {
        ItemType info;
        node* next;
        node* prev;
    };
    int length;    //optinal
    node* listData;
    node* currentPos;
public :
    sortedLinkList(){ length = 0; listData = NULL;}
    ~sortedLinkList(){ makeEmpty( ) ; }
    void resetList () { currentPos = NULL;}
    int lengthIs() const { return length ;}
    int isFull() const;
    void makeEmpty() ;
    int retrievalItem(ItemType item);
    void getNextItem (ItemType& item);
    int insertItem (ItemType item);
    int deleteItem (ItemType item);
    void print() const;
};

template<class ItemType>
int sortedLinkList<ItemType>::isFull() const
{
    node* location;
    try{
        location = new node;
        delete location;
        return 0;
    }
    catch (...){ return 1 ;}
}

template <class ItemType>
void sortedLinkList<ItemType>::makeEmpty()
{
    node* tempPtr;
    while (listData != NULL)
    {
        tempPtr = listData;
        listData = listData->next;
        delete tempPtr;
    }
    length = 0;
}

template <class ItemType>
int sortedLinkList<ItemType>::retrievalItem(ItemType item)
{
    node* location;
    for (location = listData; location != NULL; location = location->next)
        if (item == location->info)
            return 1;
    return 0;
}

template <class ItemType>
```

```

int sortedLinkedList<ItemType>::insertItem(ItemType item)
{
    if (isFull( ))
        return 0;

    node* location;
    node* before=listData, *after=listData;

    for (after=listData;after!=NULL;after=after->next)
        if (after->info > item)
            break;
        else
            before=after;

    location = new node;
    location->info = item;
    location->next = after;

    if(listData==NULL)
        listData=location;
    else
        before->next=location;
    length++ ;
    return 1;
}

```

```

template <class ItemType>
int sortedLinkedList<ItemType>::deleteItem(ItemType item)
{
    if (listData==NULL)
        return 0;
    node* location = listData;
    node* tempLocation;
    if (item == listData->info)
    {
        tempLocation = location;
        listData = listData->next; // Delete first node.
    }
    else
    {
        while (!(item==(location->next)->info))
            location = location->next;
        //Delete node at location->next.
        tempLocation = location->next;
        location->next = (location->next)->next;
    }
    delete tempLocation;
    length - - ;
    return 1;
}

```

```

template <class ItemType>
void sortedLinkedList<ItemType>::getNextItem(ItemType& item)
{
    if (currentPos == NULL)
        currentPos = listData;
    else
        currentPos = currentPos->next;
    item = currentPos->info;
}

```

```

template <class ItemType>

```

```
void sortedLinkedList<ItemType>::print() const
{
    node* location;
    for (location = listData; location != NULL; location = location->next(
        cout <<location->info;
    }
    /*****/
void main()
{

    sortedLinkedList <int> l1;
    //sortedLinkedList <char> l2;
    //sortedLinkedList <float> l3;

    l1.insertItem(1); l1.insertItem(3);
    l1.insertItem(6); l1.insertItem(4);
    l1.insertItem(5); l1.insertItem(2);

    l1.deleteItem (4);

    l1.print( ) ;

}
```

Specialized

```
#include <iostream.h>
template <class ItemType>
class SpecializedList
{
private:
    struct node
    {
        ItemType info;
        node *next, *back;
    };
    int length;    //optinal
    node* list;
    node *currentNextPos,
        * currentBackPos;

public:
    SpecializedList ( ) {length = 0; list = NULL; . }
    // ~ SpecializedList(); // Class destructor.

    void ResetForward( ) {currentNextPos = NULL;}
    void ResetBackward( ) {currentBackPos = NULL;}

    void GetNextItem (ItemType& item, int& finished);
    void GetPriorItem(ItemType& item, int& finished);

    void InsertFront(ItemType item);
    void InsertEnd (ItemType item);
    void InsertItem (ItemType item);
    int deleteItem (ItemType item);

    int LengthIs() const { return length;}
    void printForward ( ) const;
    void printBackward() const;
};

template<class ItemType>
void SpecializedList<ItemType>::GetNextItem(ItemType& item, int& finished)
{
    if (currentNextPos == NULL)
        currentNextPos = list->next;
    else
        currentNextPos = currentNextPos->next;
    item = currentNextPos->info;
    finished = (currentNextPos == list);
}

template<class ItemType>
void SpecializedList<ItemType>::GetPriorItem(ItemType& item,int& finished)
{
    if (currentBackPos == NULL)
        currentBackPos = list;
    else
        currentBackPos = currentBackPos->back;
    item = currentBackPos->info;
    finished = (currentBackPos == list->next);
}

template<class ItemType>
void SpecializedList<ItemType>::InsertFront(ItemType item)
{
    node* newNode;
    newNode = new node;
    newNode->info = item;
    if (list == NULL)
    { //list is empty.
        newNode->back = newNode;
        newNode->next = newNode;
    }
}
```

```

        list = newNode;
    }
    else
    {
        newNode->back = list;
        newNode->next = list->next;
        list->next->back = newNode;
        list->next = newNode;
    }
    length++;
}

template<class ItemType.
void SpecializedList<ItemType>::InsertEnd(ItemType item)
{
    InsertFront(item);
    list = list->next;
}

template<class ItemType>
void SpecializedList<ItemType>::InsertItem(ItemType item){
node* newNode;
newNode = new node;
newNode->info = item;
if (list == NULL)
{ // list is empty.
    newNode->back = newNode;
    newNode->next = newNode;
    list = newNode;
}
else
{
    node *i;
    for (i=list->next; i!=list; i=i->next)
        if( i->info > item )
        {
            newNode->back = i->back;
            newNode->next = i;
            (i->back)->next = newNode;
            i->back=newNode;
        }
        if( list->info >= item )
        {
            i=list; // optional
            newNode->back = i->back;
            newNode->next = i;
            (i->back)->next = newNode;
            i->back=newNode;
        }
        else
        {
            i=list; // optional
            list=newNode;
            newNode->back = i;
            newNode->next = i->next;
            (i->next)->back = newNode;
            i->next=newNode;
        }
    }
}

template<class ItemType>
int SpecializedList<ItemType>::deleteItem(ItemType item)
{
    if (list==NULL)
        return 0;
    node* i;

```

```

for (i=list->next; i!=list; i=i->next)
    if( i->info == item )
    {
        (i->back)->next=i->next;
        (i->next)->back=i->back;
        delete i;
        return 1;
    }

    if( list->info == item )
    {
        i=list; // optinal
        list=list->back;
        list->next=i->next;
        (i->next)->back=list;
        delete i;
        return 1;
    }
return 0;
}

template<class ItemType>
void SpecializedList<ItemType>::printForward() const
{ if (list==NULL)
    return;
  for (node* i=list->next; i!=list; i=i->next)
    cout << i-> info<< ' ';
  cout<< list->info << ' ';
}

template<class ItemType>
void SpecializedList<ItemType>::printBackward() const
{
  if (list==NULL)
    return;
  cout<< list->info << ' ';
  for (node* i=list->back; i!=list; i=i->back)
    cout << i-> info << ' ';
}

/*****/
template <class t>
void printBack(SpecializedList<t> l,t item)
{
  int& finished=1;
  l.GetNextItem(item, finished);
  if (!finished)
    printBack(l,item);
  cout << item << ' ';
}

void main()
{
  SpecializedList <int> l1;
  l1.ResetForward() ;
  l1.ResetBackward() ;

  l1.InsertFront(1); l1.InsertEnd(3);
  l1.InsertFront(6); l1.InsertEnd(4);
  l1.InsertFront(5); l1.InsertEnd(2);

  l1.deleteItem (6);
  l1.deleteItem (2);

  l1.printForward() ;
  cout <<endl;
  l1.printBackward () ;

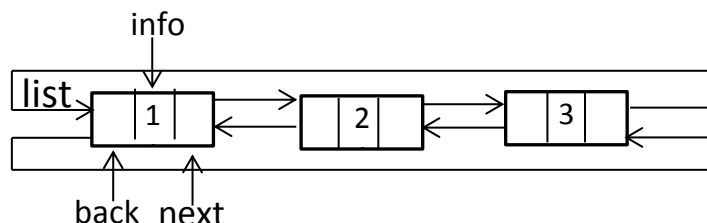
  cout << endl;
  int x=10;
  printBack(l1,x); }

```

11- specialized list

- سابقا كنا نتحدث عن single linked ام هنا سنتحدث عن double linked

- العنصر كان يتكون من قسمين أحدهم للقيمة والآخر مؤشر للعنصر الذي يليه , اما هنا فيتكون من 3 أقسام أحدهم للقيمة ومؤشران (مؤشر للعنصر الذي يليه ومؤشر للعنصر الذي يسبقه)



الاضافة:-

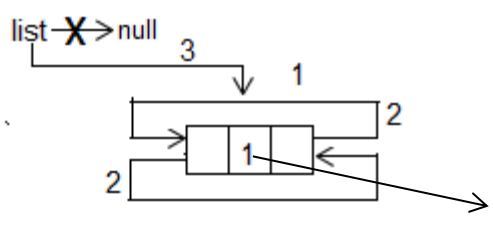
- عند اضافة العنصر الأول يتم انشاء node وتخزن فيه القيمة , تكون list تشير ل null يصبح مؤشران next و back العنصر الجديد يشيران للعنصر نفسه و تصبح list تشير للعنصر

- اضافة العنصر الثاني يتم انشاء node وتخزن فيه القيمة ويصبح الـ back للعنصر الجديد يشير لـ list و next تشير لما يشير له list من ناحية next ثم جعل مؤشر list->next للعنصر الجديد و back لـ list للعنصر الجديد , ثم تغيير مؤشر list لـ new node

1- انشاء node وتخزين القيمة فيه .

1- جعل مؤشر back للـ new node يشير الى list

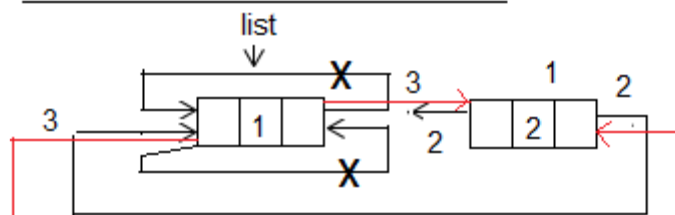
list → null



2- جعل مؤشر next للـ new node يشير لـ list->next

list->next

3- جعل مؤشر list->next يشير الى new node



4- (1) اذا اردنا الاضافة للأمام نغير مؤشر list مع كل اضافته لـ list->next=new node

(2) اذا اردنا الاضافة تزداد لجهة اليسار نغير مؤشر list مع كل اضافة لـ list->next = list بعد كتابة الجملة السابقة

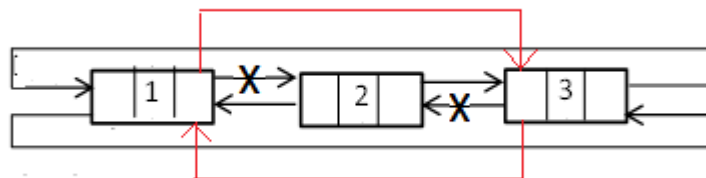

```
void Insert ( t item ){
node * new node =new node <T> ;
new node info = item ;
if ( list == null )
{ new node → back = new node
new node → next = new node
list = new node ;}
else { new node → back = list ;
new node → next = list → next ;
(new node → next ) → back = new node;
( new node → back ) → next = new node ;
}
length ++; }
```

عد للكود الكامل وادرس كود *insertFront* , *insertEnd* وقارن بين الطريقتين

بافتراض ارقام والتطبيق عليه ☺

الحذف

- عند حذف عنصر محدد نحتاج لمؤشر ينتقل بين العناصر للبحث عن العنصر و عندما يجده يبدل المؤشرات التي كانت تشير له ، مؤشر ال back للعنصر الذي بعده يصبح يشير للعناصر الذي قبله ، و مؤشر ال next للعنصر الذي قبله يصبح يشير للعنصر الذي بعده



واذا كانت القيمة المراد حذفها مخزنة في list نجعل المؤشر يشير لـ list ثم تغير مكان ال list لتنتقل العنصر الى العنصر قبله ثم جعل مؤشره next يشير لـ i → next و i → next → back

يشير لـ list

Int delete (T item)

{ if (list == null)

return 0 ;

*node * i;*

if (i=list→next ; I != list ; i=i→next)

if (i→info ==item)

{ (i→back) → next = i→next;

(i→ next) →back=i→back;

delete I ; return 1 ;}

if (list → info==item)

{ i=list ; list=list→ back ; list→ next= i→ next ;

(i→next) → back = list; delete I ; return 1 ; } return 0 ;}

Zero^{Team}**One**

الطباعة :

في الطباعة تطبق نفس الخطوات الماضية , فإذا اردنا الطباعة من اليمين لليسار نمشي $i=i \rightarrow \text{back}$

وإذا اردنا من اليمين لليسار $i=i \rightarrow \text{next}$

```
void print() forward () const
{ if ( list == null)
return ;
for ( node * i=list  $\rightarrow$  next ; i != list ; i=i  $\rightarrow$  next )
cout<< i  $\rightarrow$  info << ' ' ;
cout << list  $\rightarrow$  info ; }
void print backward () const
{ if ( list ==null )
return ;
for ( node * i=list  $\rightarrow$  back ; i != list ; i =i  $\rightarrow$  back )
cout << i  $\rightarrow$  info << ' ' ;
cout << list  $\rightarrow$  info ;
}
```

Some example :

1- what is the out put of the following code?

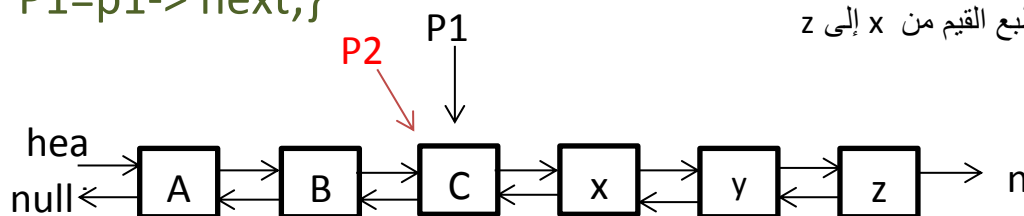
```
{p1=p1-> next;
P2=p1->back;
While(p2->back != null)
{cout<<p2->info;
P2=p2->back;}
Cout<<endl;
While(p1->next != null)
{cout<<p1->info;
P1=p1-> next;}
```

1- عند تطبيق جملة $p1=p1 \rightarrow next$ ينتقل مؤشر $p1$ ليشير على x

2- عند تطبيق جملة $p2=p1 \rightarrow next$ ينتقل مؤشر $p2$ ليشير على c

3- في جملة `while` الاولى يطبع القيم من C إلى A

4- في جملة `while` الثانية يطبع القيم من x إلى z

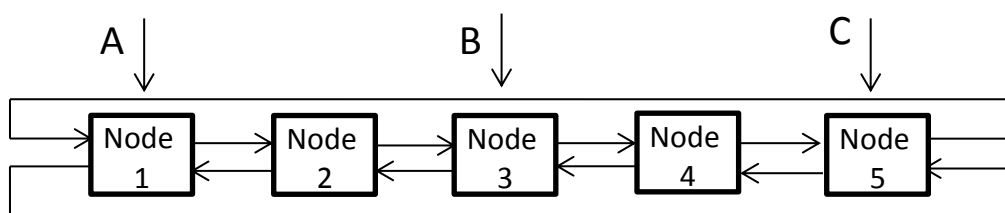


Out put:

C B

x y

2- using the circular doubly linked list , give the expression corresponding to each of the following descriptions.



1- the info member of node 1 , reference from pointer c

`(C->next)->info;`

2- the next member of node 2 , reference from pointer A

`(A->next)-> next;`

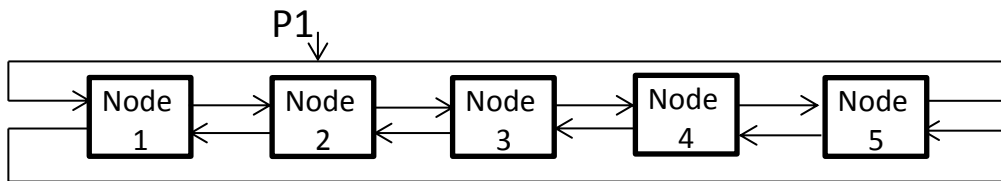
3- the pointer member of node 1 , reference from pointer B

`(B->back)->next;`

4- the back member of node 4 , reference from pointer C

`(C->back)->back;`

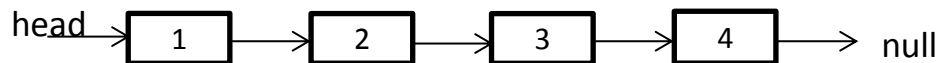
3- you have circular doubly linked list and pointer p1 which points to the second node in it , return a pointer to the immediate successor of last node .



نفترض ان معنا هذا الشكل و p1 يشير الى ثاني نود , يريد منا باستخدام p1 جعله يشير الى النود الذي يلحق اخر نود (يعني next\آخر node) وهو يكون اول نود المؤشر يؤشر لثاني نود (النود الاول back للنود الثاني)

$P1 = p1 \rightarrow back ;$

4- pointer p points to the head of the sorted linked list write to y the contain of the last node .



نفترض انه معنا sorted linked list بغض النظر عن القيم المخزنة داخلها و p يشير الى head (يعني اول عنصر) يريد منا تخزين قيمة y في اخر عنصر.

حتى نصل الى اخر node علينا التنقل من عنصر لآخر بواسطة next حتى نصل الى اخر عنصر ثم نخزن القيمة .

For (p= head ;(p→ next) != null ; p=p → next) ;

Y = p → info ;

عندما تكون $p \rightarrow next == null$ هذا يعني انها وصلت لآخر عنصر بينما $p=null$ فهذا يعني انها تعد آخر عنصر.

5- pointer p points to any node in the linked queue remove the last node .



لنفترض ان هذا الشكل لدينا , يريد منا حذف اخر عنصر , وقاعدة ال LIFO queue لذلك لا بد من حذف كل العناصر التي قبله حتى نصل اليه :: باختصار يريد منا تفريغها من العناصر , عند الحذف يتحرك front

While (front != null)

{node < t> *temp ;

Temp = front ;

Front = front → next ;

Delete temp;}

Rear=null ;

وعندما تصبح فارغة فلا بد من تعديل مؤشر rear

6- write of c++ statements to swap the front element with last element in linked queue .

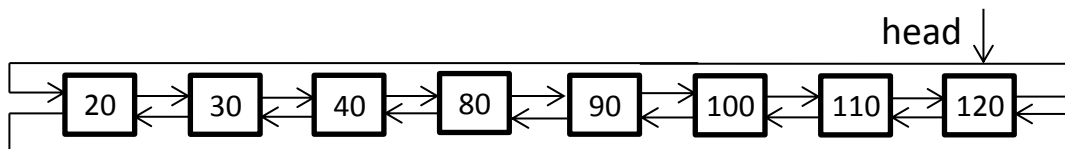
يريد تبديل بين قيمة front مع اخر قيمة اضيفت (rear)

```
Node < t> *temp ;
Temp → info = front → info ;
Front → info = rear → info ;
Rear → info = temp → info ;
Delete temp;
```

7- if you have circular doubly linked list with the following data

(20 30 40 80 90 100 110 120) , (head point to 1200)

Write statements to display (110 → 120 → 20 → 30)



نحتاج ل node يشير للعنصر السابق ل head ثم ينتقل عبر مؤشر next لباقي العناصر

```
Int x=1
While ( x !=5) حتى تطبع 4 عناصر فقط
{ cout << temp → info ;
Temp =temp → next ;
X++;}
```

```
Node < t> *temp ;
Temp = head → back ;
While ( temp !=(( head → next ) → next ) جملة تمر بالعناصر إلى أن تصل 30 وتقف
{ cout << temp → info ;
Temp = temp → next ;}
```

طريقان للطباعة ويمكن كتابة غيرهم بطرق اخرى وما دامت النتيجة صحيحة فالكود صحيح ,

للتأكد من ان البرنامج صحيح تتبعه وان كانت النتيجة صحيحة فالبرنامج صحيح .

8- T any function (node <t> *p ,node <t> *l)

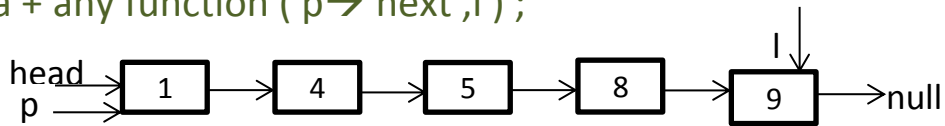
{ if (p==l)

Return p → data ;

Else

Return p → data + any function (p → next ,l) ;

}



What is the value returned from the function ?

Return 1 : p → 1 , l → 9

1+ return 2

1+ return 2 = 1+26 = 27

Return 2 : p → 4 , l → 9

4+ return 3

4+ return 3 = 4+22 = 26

Return 3 : p → 5 , l → 9

5+ return 4

5+ return 4 = 5+17 = 22

Return 4 : p → 8 , l → 9

8+ return 5

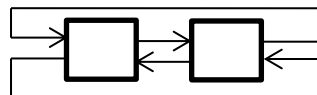
8+ return 5 = 8+9 = 17

Return 5 : p → 9 , l → 9

9

9- in circular doubly linked list , insertion of a new node between two entire node involve the modification of .

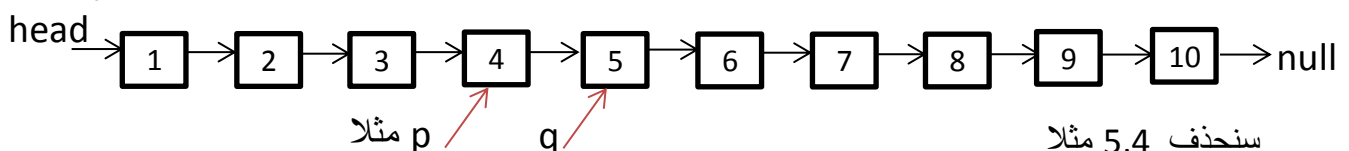
عند اضافة عنصر بين عنصرين غير مؤشرين الواقعين بين العنصرين بالاضافة لمؤشرين العنصر الجديد



فيصبح عدد المؤشرات المتغير 4 pointers .

10 - write code will delete tow successive node after p node from single linked list ?

(p points to a node , q pointer to next node of p node and list has 10 nodes)



سنحذف 5,4 مثلا

تغيير مؤشر ال p (next) لخانتين

P → next = p → next → next → next ;

سنحول مؤشر 4 > null

Q → next = null;

سنحول مؤشر 5 > null

q → next → next = null ;

11- if the circular doubly linked list

head → Ali ahmad mohammad anas kamel majed)

```
Node <t>*t= head ;
```

```
head → next → next → data = t → back → back → data ; //
```

```
cout << head → data << ";" << head → next → next → data ;
```

```
cout << ";" << t → back → back → data ;
```

** output : ali ; kamel ; kamel

12- linked queue before executing the following code contain [12, 14, 7]

```
Int x=5 , y=3 ;
```

```
q. enqueue (4) ; q. enqueue (x) ; q. enqueue (x+1)
```

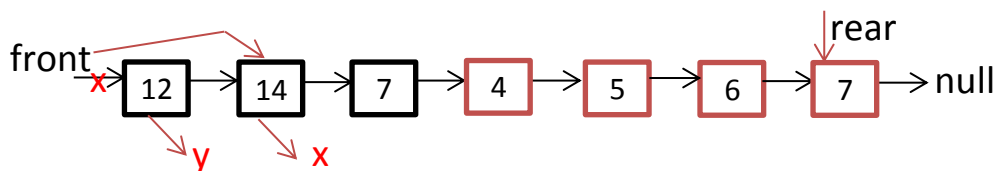
```
q. dequeue (Y) ;
```

*تخزن القيمة المحذوفة في y

```
q. enqueue (x+Y) ;
```

```
q. dequeue (x) ;
```

```
cout << "x=" << x << "y =" << y ; // 14, 12
```



13- if address of 8th nodes in doubly linked list is 2028 and each node has data field length = 10 byte and address space of memory 3 byte , what is address of 9th node?

A) 2041 b)2038 c)2040 d)none

لان العناصر في linklist ليس شرطاً أن تكون مواقعها في الذاكرة متتاليه


```
#include <iostream.h>

template <class ItemType>

class Tree{

public:

    Tree() { root = NULL ;}

    ~ Tree () { Destroy(root) ;}

    int isEmpty() const      { return root == NULL;}

    void insertItem (ItemType item){ Insert(root, item);}

    int deleteItem (ItemType item ) { return Delete(root, item ) ;}

    void NLR () const { cout <<" Pre-Order(NLR): "; recursiveNLR(root) ;}

    void LNR () const { cout <<" In-Order(LNR): "; recursiveLNR(root) ;}

    void LRN () const { cout <<"Post-Order(LRN): "; recursiveLRN(root) ;}

    int retrieveItem(ItemType& item) const{ return Retrieve(root, item) ;}

    int lengthIs() const {return CountNodes(root) ;}

private:

    struct node

    {
        ItemType info;

        node *left, *right; };

    node* root;

    //////////Helper Methode\\\\\\\\\\\\\\

    void Destroy (node * & );

    void recursiveNLR (node*) const;

    void recursiveLNR (node*) const;

    void recursiveLRN (node*) const;

    void Insert(node*&, ItemType);

    int Delete(node*&, ItemType);

    void DeleteNode(node*& tree);

    int CountNodes(node*) const;

    int Retrieve (node* ,ItemType& item) const;

};

/*****/

template<class ItemType>

void Tree<ItemType>::Destroy(node*& tree)

{ if (tree != NULL)

{
    Destroy(tree->left);
```

```
    Destroy(tree->right);
```

```
    delete tree ;   } }
```

```
template<class ItemType>
```

```
void Tree<ItemType>::Insert(node*& tree, ItemType item)
```

```
{ if (tree == NULL)
```

```
{ // Insertion place found.
```

```
    tree = new node;
```

```
    tree->right = NULL;
```

```
    tree->left = NULL;
```

```
    tree->info = item ;   }
```

```
else if (item < tree->info)
```

```
    Insert(tree->left, item); // Insert in left subtree.
```

```
else
```

```
    Insert(tree->right, item); // Insert in right subtree .}
```

```
template<class ItemType> // 498
```

```
int Tree<ItemType>::Delete(node*& tree, ItemType item){
```

```
    if (tree== NULL)        // Node not found; return false
```

```
        return 0;
```

```
    else if (item == tree->info) // Node found; call DeleteNode.
```

```
        { DeleteNode(tree); return 1;   }
```

```
    else if (item < tree->info)
```

```
        return Delete(tree->left, item); // Look in left subtree.
```

```
    else // if (item > tree->info)
```

```
        return Delete(tree->right, item); // Look in right subtree.
```

```
}
```

```
template<class ItemType>
```

```
void Tree<ItemType>::DeleteNode(node*& tree)
```

```
{ node* tempPtr = tree;
```

```
    if (tree->left == NULL)
```

```
{ tree = tree->right;
```

```
    delete tempPtr;}
```

```
    else if (tree->right == NULL)
```

```
{ tree = tree->left;
```

```
    delete tempPtr;}
```

```
else
```

```

{    node* Predecessor=tree->left;
    while (Predecessor->right != NULL)
        Predecessor = Predecessor->right;
    tree->info = Predecessor->info;
    Delete(tree->left, Predecessor->info); // Delete predecessor node
}
}

```

```

template<class ItemType>
void Tree<ItemType>::recursiveLNR (node* subRoot) const
{ if (subRoot != NULL)
    {    recursiveLNR(subRoot->left); // Print left subtree.
        cout << subRoot->info << ' ' ;
        recursiveLNR(subRoot->right); // Print right subtree.
    }
}

```

```

template<class ItemType>
void Tree<ItemType>::recursiveNLR (node* subRoot) const
{ if (subRoot != NULL)
    {    cout << subRoot->info << ' ' ;
        recursiveNLR(subRoot->left); // Print left subtree.
        recursiveNLR(subRoot->right); // Print right subtree.
    }
}

```

```

template<class ItemType>
void Tree<ItemType>::recursiveLRN (node* subRoot) const
{ if (subRoot != NULL)
    {    recursiveLRN(subRoot->left); // Print left subtree.
        recursiveLRN(subRoot->right); // Print right subtree.
        cout << subRoot->info << ' ' ;
    }
}

```

```

template<class ItemType>
int Tree<ItemType>::CountNodes(node* tree) const
{if (tree == NULL)
    return 0;
else
    return CountNodes(tree->left) + CountNodes(tree->right) + 1;
}

```

```

template<class ItemType>

```

```

int Tree<ItemType>::Retrieve(node* tree,ItemType& item) const
{ if (tree == NULL)
    return 0; // item is not found.
else if (item < tree->info)
    return Retrieve(tree->left, item); // Search left subtree.
else if (item > tree->info)
    return Retrieve(tree->right, item); // Search right subtree.
else
    return 1;}

```

```

/*****

```

```

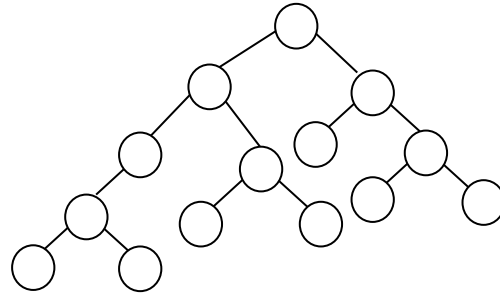
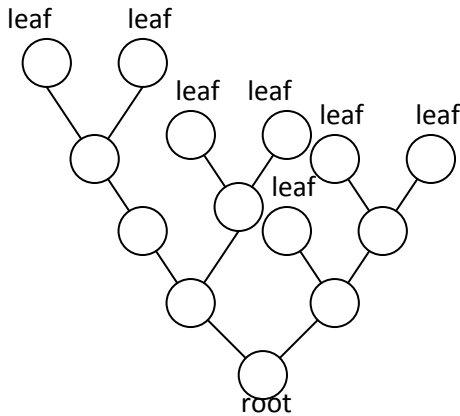
void main() {
    Tree <int>t1;
    t1.insertItem(5); t1.insertItem(9);
    t1.insertItem(7); t1.insertItem(3);
    t1.insertItem(8); t1.insertItem(12);
    t1.insertItem(6); t1.insertItem(4);
    t1.insertItem(20); t1.insertItem(1);
    t1.NLR(); cout << endl;
    t1.LNR(); cout << endl;
    t1.LRN(); cout << endl;
    cout<" >> n;"
    t1.deleteItem(12); t1.deleteItem(4);
    t1.deleteItem(9); t1.deleteItem(1000);
    t1.NLR(); cout << endl;
    t1.LNR(); cout << endl;
    t1.LRN(); cout << endl;}
    */The Output
    Pre-Order(NLR): 5 3 1 4 9 7 6 8 12 20
    In-Order(LNR): 1 3 4 5 6 7 8 9 12 20
    Post-Order(LRN): 1 4 3 6 8 7 20 12 9 5

    Pre-Order(NLR): 5 3 1 8 7 6 20
    In-Order(LNR): 1 3 5 6 7 8 20
    Post-Order(LRN): 1 3 6 7 20 8 5
    /*

```

12~trees

أي شجرة يكون لها جذر (*root*) ويكون له أفرع والفرع الواحد يكون له أفرع



nodes : جميع عناصر الشجرة

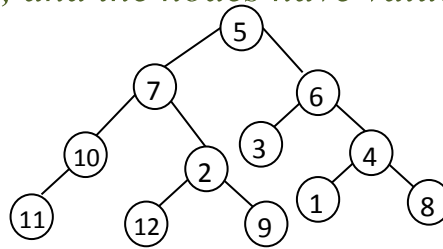
root : اول عنصر في الشجرة

leaf : اخر عنصر في الشجرة ولا يتفرع منه شيء

* كيف يتم تخزين البيانات في الشجرة ??

البيانات "data" ستخزن داخل ال *nodes* .

Trees have some nodes , and the nodes have value and have children or not .

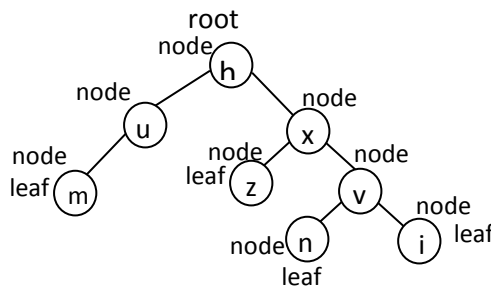


Create a tree of these values " h , u , x , z , v , m , n , I "

• سيتم ترتيب العناصر في شجرة دون مراعاة أي شرط .

root : h

leaf : m , z , n , i



تسمى هذه الشجرة " *Binary tree* " ماذا يعني ??

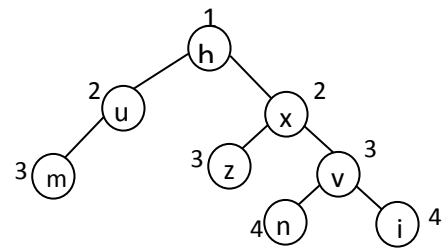
كل *node* له على الأكثر 2 children

لو كان للعنصر 3 children يصبح اسمها *thinary tree*

length tree

* طول الشجرة هو اطول مسار من ال $leaf \rightarrow root$

$$\# Level = length - 1$$



أطول مسار 4

$$Length = 4$$

$$\# Level = 3$$

$$Size\ tree = 8$$

Size tree

عدد ال $nodes$ في الشجرة

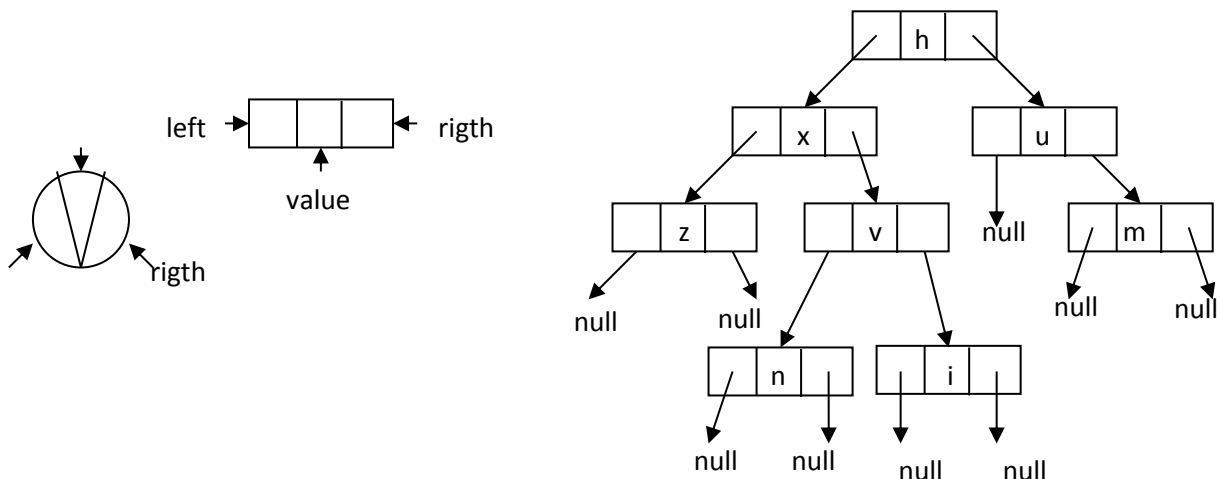
$$Maximum\ \# of\ node\ in\ this\ level = 2^{\#level}$$

$$Level\ 3 \rightarrow 2^3 = 8$$

$$\# of\ node\ in\ tree = 2^n + (2^n - 1)$$

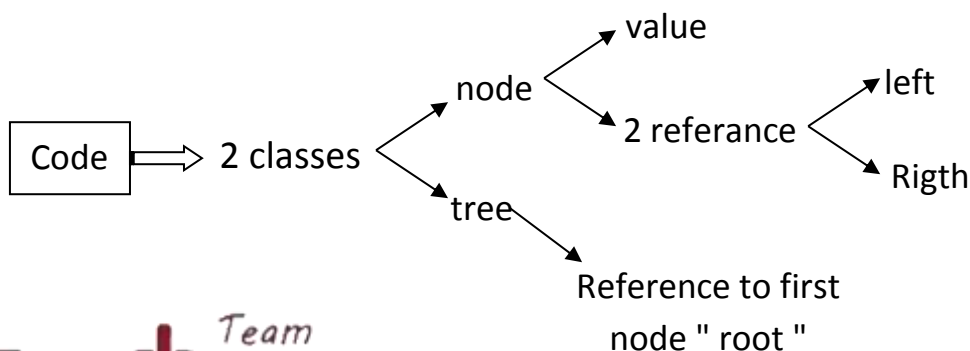
↑ ↑
عدد عناصر عدد عناصر
أكبر مستوى المستويات قبله

برمجة ال tree



tree code

لكتابة كود الشجرة نحتاج إلى 2 classes احدهم للإنشاء العنصر نفسه والاخر لاستخدام العناصر في انشاء الشجرة



Team
ZeroOne

* عند تهيئة المؤشرات جميعها تشير إلى null

```

#include <iostream.h>
template <class ItemType>
class Tree{
public:
    Tree () { root = NULL;}
    ~Tree () {Destroy(root) ; }
    int isEmpty() const {return root == NULL; }
    void insertItem (ItemType item) { Insert(root, item) ; }
    int deleteItem (ItemType item) { return Delete(root, item) ; }
    void NLR () const {cout <<" Pre-Order(NLR): "; recursiveNLR(root) ; }
    void LNR () const {cout <<" In-Order(LNR): "; recursiveLNR(root) ; }
    void LRN () const {cout <<"Post-Order(LRN): "; recursiveLRN(root) ; }
    int retrieveItem(ItemType& item) const { return Retrieve(root, item) ; }
    int lengthIs () const { return CountNodes(root) ; }

private:
    struct node
    {
        ItemType info;
        node *left, *right; };
    node* root;
    //Helper Methode
    void Destroy (node * n);
    void recursiveNLR (node*) const;
    void recursiveLNR (node*) const;
    void recursiveLRN (node*) const;
    void Insert(node*&, ItemType) ;
    int Delete(node*&, ItemType) ;
    void DeleteNode(node*& tree) ;
    int CountNodes(node*) const;
    int Retrieve (node*, ItemType& item) const;
};

```

Team
ZeroOne

Class tree

“public”

- تم تهيئة الشجرة بحيث يكون مؤشر اسمه *root* يشير إلى *“null constructor”*

- عند حذف الشجرة يتم استدعاء كود *“destructor”* *destroy*

- وجود بعض الاضافات سيتم شرحها لاحقا *“isEmpty, insertItem, deleteItem, retrieveItem, length, NLR, LNR, LRN”*

“private”

- تعريف النودات عن طريق *struct* وهي طريقة أخرى بديله عن انشاء *class* جديد واستدعاء

- تعريف مؤشر *root*

- تعريف *function* داخل *private* لا يمكن إلا استخدامها داخل ال *class* نفسه

Empty or not

عدم وجود عناصر \rightarrow root == null

وجود عناصر \rightarrow root != null

```
{ int isEmpty() const
return (root == NULL); }
```

يرجع 1 اذا كانت فارغة و 0 اذا كانت غير ذلك

معرفة اذا كان العنصر leaf or not

Leaf \rightarrow left & right == null

Leaf \rightarrow left & right != null

```
Int isLeaf (node*& tree)
{ if ((tree->left == NULL) && (tree->right == null))
return 1;
else
return 0; }
```

Some method

معرفة طول الشجرة

```
int lengthIs ( ) const { return CountNodes(root) ; }
int CountNodes(node* tree) const
{if (tree == NULL)
return 0;
else
return CountNodes(tree->left) + CountNodes(tree->right) + 1;}
```

في هذا الكود سيتم حساب عدد العناصر في الشجرة ومن خلال التطبيق على القانون سيتم معرفة الطول

Tree traversal الممرور على جميع العناصر

عن طريق 3 طرق

قبل Pre order

وسط in order

بعد post order

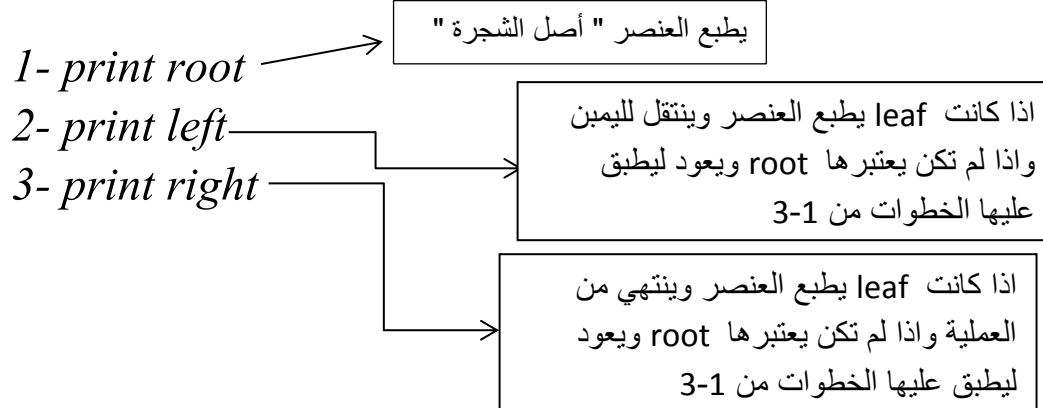
للطباعة

بحث عن عنصر

Max \ min

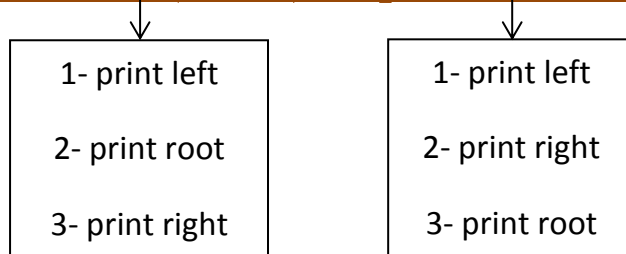
Print tree

Pre order (NLR)



In order (LNR) & post order (LRN)

نفس الخطوات السابقة لكن تختلف بالترتيب



```
void LNR ( ) const { cout << " In-Order(LNR): "; recursiveLNR(root) ; }
template<class ItemType>
void Tree<ItemType>::recursiveLNR (node* subRoot) const
{ if (subRoot != NULL)
{ recursiveLNR(subRoot->left); // Print left subtree.
cout << subRoot->info << ' ';
recursiveLNR(subRoot->right); // Print right subtree.}}
```

```
void LRN ( ) const { cout << "Post-Order(LRN): "; recursiveLRN(root) ; }
template<class ItemType>
void Tree<ItemType>::recursiveLRN (node* subRoot) const
{ if (subRoot != NULL)
{ recursiveLRN(subRoot->left); // Print left subtree.
recursiveLRN(subRoot->right); // Print right subtree.
cout << subRoot->info << ' ' ; }}
```

```
void NLR ( ) const { cout << " Pre-Order(NLR): "; recursiveNLR(root) ; }
template<class ItemType>
void Tree<ItemType>::recursiveNLR (node* subRoot) const
{ if (subRoot != NULL)
{ cout << subRoot->info << ' ' ;
recursiveNLR(subRoot->left); // Print left subtree.
recursiveNLR(subRoot->right); // Print right subtree.}}
```

* سيتم الشرح للطريقة الأولى وابني عليها للطريقة الثانية والثالثة مع مراعاة الترتيب

“LNR”

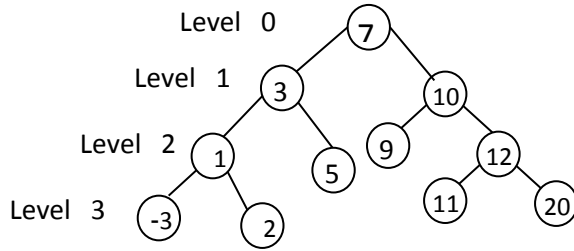
- عند الطباعة يتم استدعاء كود " recursiveNLR

- في هذا الكود يبدأ بالمرور على العناصر من البداية " root " اذا كان يشير ل null تكون فارغة ويخرج من الكود لعدم تحقق شرط جملة if

- اذا تحقق الشرط يستدعي الكود نفسه مره اخرى لكن ينتقل للعنصر الذي يكون على يسار ال root " ويعتبره root يبقى يستدعي بيسار العناصر حتى يصل لآخر عنصر من جهة اليسار الذي اذا استدعي يساره يكون null فيعود للكود ثم يطبعه "

- استدعاء الكود نفسه مره اخرى لكن ينتقل لجهة اليمين لآخر عنصر طبعه , اذا كان يشير ل null يعود لطباعة ال node الذي طبعنا يساره ثم ينتقل لليمينه , يعتبر اليمين root جديده وينتقل ليسارها ' اذا كان اليسار null فيعود لطباعتها ثم ينتقل ليمينها وهكذا حتى آخر عنصر

* خلال التطبيق على الكود يصبح هناك تداخل بين الأوامر لذلك عليك الانتباه والتركيز قليلا



Length = 4 # level = 3 #maximum of node in tree = $2^3 + 2^3 - 1 = 15$

NLR <-- Print pre order : Pre order

طباعة النود الاول <7> له يمين ويسار " ينتقل لليساار اولاً , اليسار عبارته عن نود يطبعه <3> " له يمين ويسار " فينتقل ليساره , اليسار عبارته عن نود يطبعه <1> " له يمين ويسار " فينتقل لليساار , اليسار عبارته عن نود يطبعه <-3> " ليس له يمين ولا يسار " فيرجع للنود "1" ينتقل ليمينه , اليمين عبارته عن نود يطبعه <2> " ليس له لا يمين ولا يسار فيرجع للنود "3" وينتقل ليمينه , اليمين عبارته عن نود يطبعه <5> " ليس له لا يمين ولا يسار " فيعود للنود "7" وينتقل ليمينها بعد أن أنهينا جهة اليسار له , اليمين عبارته عن نود يطبعه <10> " لها يمين ويسار فينتقل لليساار , اليسار عبارته عن نود يطبعه <9> " ليس له يمين ولا يسار " فيعود للنود "10" , وينتقل ليمينه , اليمين عبارته عن نود يطبعه <12> " له يمين ويسار " فينتقل لليساار , اليسار عبارة عن نود يطبعه <11> " ليس له يمين ولا يسار " فيعود للنود "12" وينتقل لليمين , اليمين عبارته عن نود يطبعه <20> " ليس له يمين ولا يسار " وبذلك نكون مررنا على جميع العناصر وطبعناها

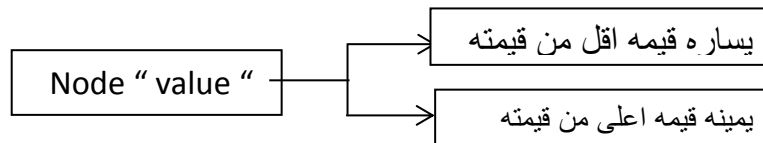
النتيجة " 7, 3, 1, -3, 2, 5, 10, 9, 12, 11, 20 "

Print in order : " -3 , 1, 2, 3, 5, 7, 9, 10, 11, 12, 20 "

Print post order : " -3.2.1.5.3.9.11.20.12.10.7 "

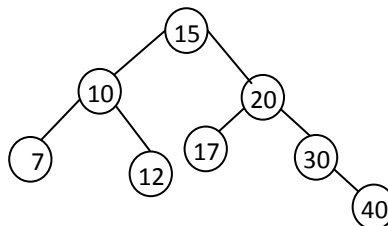
binary search trees "BST"

هي شجرة كل عنصر له على الأكثر 2 children لكن خاصية جديدة هي أن كل node له قيمه "value" على يساره قيم أقل من قيمته وعلى يمينه قيم أكبر من قيمته

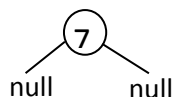


اليسار $\leftarrow 15 > 10, 12, 7$

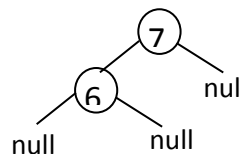
اليمين $\leftarrow 15 < 20, 17, 30, 40$



Ex: Create a tree of these values "7, 6, 4, 10, 15, 2, 20, 5, 8, 13, 16, 3"



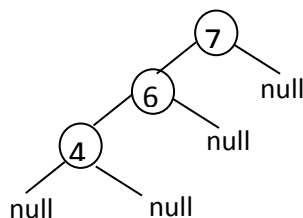
أول رقم "7" root



6 < 7 خطوة يسار

يسار = null انشاء node

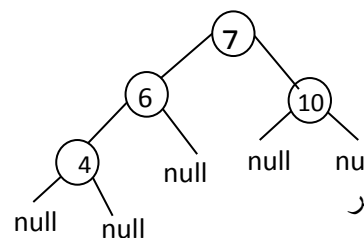
وجعل يسار "7" يشير للعنصر الجديد



4 < 7 ينتقل لليسار 4 < 6 ينتقل لليسار

يسار = null انشاء node

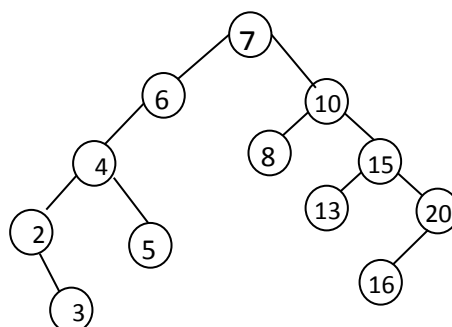
وجعل يسار "6" يشير للعنصر الجديد



10 > 7 ينتقل لليمين

يمين = null انشاء node

وجعل يمين "7" يشير للعنصر الجديد



يستمر بالإضافه على نفس المبدأ إلى أن ينهي جميع العناصر فيكون الشكل

Team
ZeroOne

يتم الاستفادة من هذه الطريقة في تسهيل عملية البحث عن عنصر

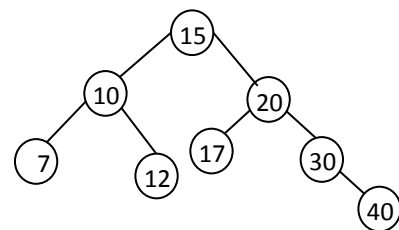
- مثلاً لو أردنا البحث عن العنصر 40 سنقوم بتعريف مؤشر يشير إلى أول عنصر وبعدها يقارن القيم إذا كانت أكبر يسلك الجزء الأيمن من الشجرة ويهمل الجزء الأيسر والعكس إذا كانت أقل , ثم ننقل المؤشر ليشير للنود الذي انتقلنا له فيعتبره *root* جديده ويعيد الكرة نفسها بالمقارنه وإهمال جزء ومواصلة الخطوات على الجزء الآخر

1- referace to root

2- $value\ root < 40 \rightarrow$ يكمل البحث عاليمين ويهمل الجزء اليسار

$Value\ root > 40 \rightarrow$ يكمل البحث على اليسار ويهمل اليمين

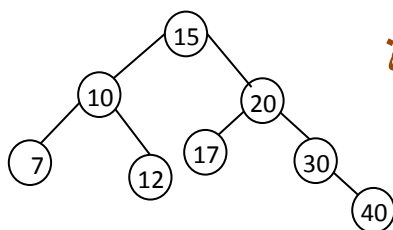
3- يكرر الخطوات 1,2 إلى أن يجد العنصر



أكبر عدد مرات يمكن البحث فيها عن عنصر $\log_2 n =$ حيث $n =$ عدد عناصر الشجرة

$$n=1000000 \quad \log_2 1000000 = 20 = \text{عدد مرات البحث}$$

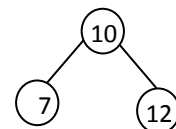
يمكن أن أجد العنصر بأقل من 20 مرة لكن هذا إن كان العنصر *leaf* في آخر *level*



نريد البحث عن 12 داخل هذه الشجرة

1- $12 < 15$ يهمل الجزء الأيمن وننتقل للجزء الأيسر

2- $12 > 10$ ننتقل للجزء الأيمن ونهمل الأيسر



3- وجدنا العنصر في الجزء الأيمن وتنتهي عملية البحث ☺

*عملية الإضافه تشبه عملية البحث في مبدئها *

```

int searchtem (ItemType item, return search(root, item) : }
template<class ItemType>
int Tree<ItemType>::search(node*& tree, ItemType item)
{
    if (tree== NULL)           // Node not found; return false
        return 0;
    else if (item == tree->info) // Node found
        cout<<tree->info;
        else if (item < tree->info)
            return search(tree->left, item); // Look in left subtree.
        elseif (item > tree->info)
            return search(tree->right, item); // Look in right subtree.
}

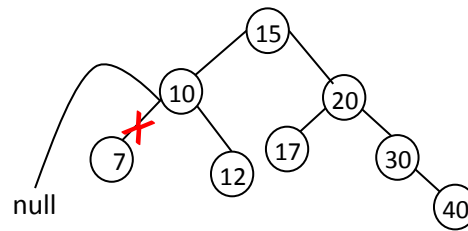
```

Binary tree deletion:

1- Delete node has no children

* جعل المؤشر الذي يشير له يشير لـ null

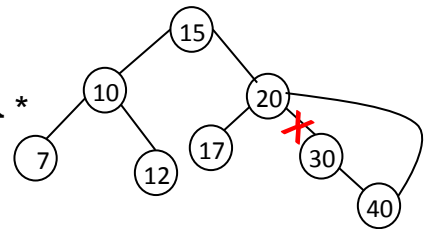
حذف الـ 7 مثلاً بتغيير مؤشر يسار الـ 10 ليشير لـ null



2- Delete node has one children

* جعل المؤشر الذي يشير إليها يشير لطفلاها بدلاً منها

* حذف الـ 30 مثلاً وذلك بتغيير مؤشر يمين الـ 20 ليشير إلى 40 بدلاً من 30



3- Delete node has two children

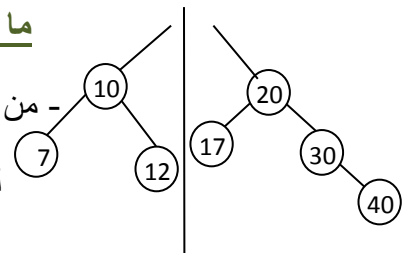
* حذفه مباشرة سيؤدي لضياع جزء من الشجرة المرتبط بها

لذلك سيتم استبدال القيمة بقيمه أخرى كحذف الـ root مثلاً

ما هي القيمة التي يمكن أن تحل محل الـ root أو أي عنصر يرتبط به عنصرين؟؟

- من خصائص الـ root أن كل العناصر التي على يمينه أكبر منه وعلى يساره أقل منه

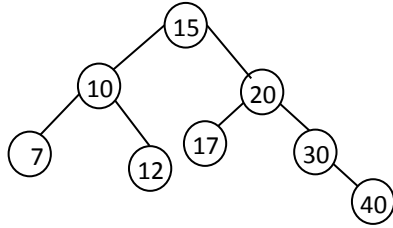
لذلك يتم استبدال القيمة إما بأكبر قيمة من جهة اليسار , أو أقل قيمة من جهة اليمين



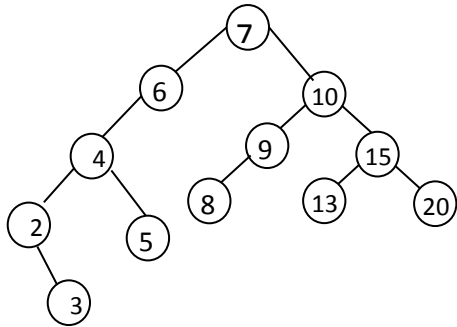
تصبح شجرتان ولا مرجع لأحدهما يمكننا من الوصول للعناصر

* إذا أردنا حذف 15 :

- لنأخذ أكبر قيمة من جهة اليسار بداية , ننتقل يسار ثم أقصى يمين فنجد أن ال 12 هي أكبر قيمة من جهة اليسار



- إذا أردنا استبداله بأقل قيمة من جهة اليمين , ننتقل يمين ثم أقصى يسار فنجد أن 17 هي أقل قيمة من جهة اليمين



مثال آخر : استبدال 7 بأقل قيمة من جهة اليسار , ننتقل يمين "10" ثم أقصى يسار مروراً ب "9" لنصل بالنهاية ل 8

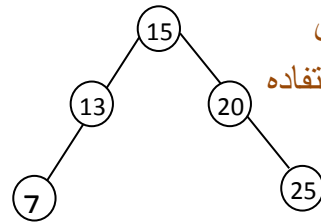
استبداله بأكبر قيمة من جهة اليسار , ننتقل يسار "6" لنجد أنه ليس لها يمين فنغير مؤشر اليمين لل 6 يشير ل 10

AVL trees



في هذه الشجرة نجد أنها أصبحت شبيهة ل link list وفقدت الكثير من الخصائص التي توفرها طريقة ترتيب العناصر في شجرة

ترتيب آخر



بهذا الشكل يمكن الإستفادة مما توفره الشجرة

- search]
- insert] عدد مرات البحث او الاضافة او الحذف لعنصر
- delete] مساوي لعدد العناصر

- search]
- insert] عدد مرات البحث او الاضافة او الحذف لعنصر
- delete] يصبح على القانون $\log_2 n$

يسمى الشكل الأول الشبيه ب link list " not balanced tree "

أما الشكل الثاني " balanced tree "

كيف يتم معرفة أنها balanced or not ؟

1- نأخذ كل نود لوحده ونقارن الطول بين جهة اليمين واليسار , اذا كان الفرق قليل " 0 , 1 , -1 " تكون *balance*

2- نجد المقارنه لكل عنصر

* عملية المقارنة تسمى *balanced factor*

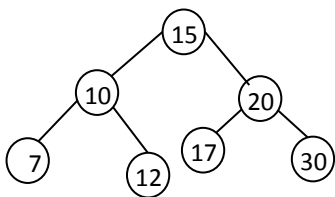
$$\text{Balanced factor} = \text{length left} - \text{length right}$$

* باختصار : يمكن معرفة أنها متوازنة " *balanced* " إذا كان أغلب ال *leaf* في آخر *level* او ما قبله

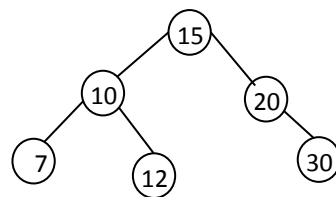
Full tree , complete tree

Full tree → # node in last leaf = 2^n $n = \# \text{ level}$

Complete tree → → # node in last leaf $\neq 2^n$ $n = \# \text{ level}$



Full tree



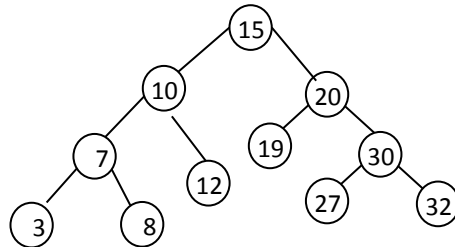
Complete tree

أي *full tree* هي *complete tree*

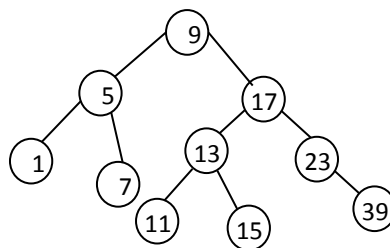
وليس كل *complete tree* هي *full tree*

Questions years

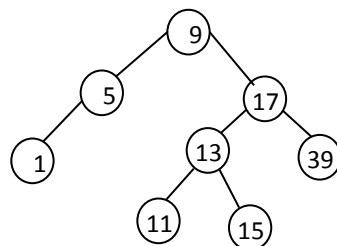
1- Create a tree of these values (15, 10, 20, 19, 12, 7, 3, 30, 27, 17, 32, 8)



2- a- Create a tree of these values (9, 17, 13, 5, 15, 23, 11, 7, 39, 1)



b-Redraw the tree after delete item (23) and (7) .



3- you have tree find the out put:

a- pre order

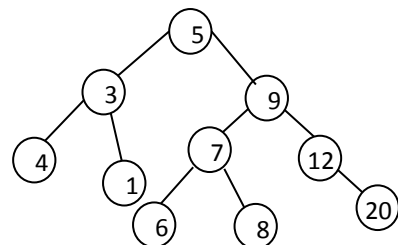
" 5 3 4 1 9 7 6 8 12 20 "

b- in order

" 4 3 1 5 6 7 8 9 12 20 "

c- post order

" 4 1 3 6 8 7 20 12 9 5 "



4- find the output :

If (p→right == null)

cout << p → right → info;

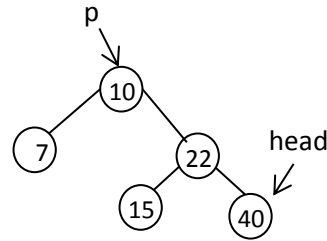
else

head = p → right → right;

while (head != null)

{ cout << head → info;

head = head → left; }



Out put :

40

5- pointer p point to root , write code to save in pointer left right the root >

لدينا مؤشر يشير ل root يريد منا تخزين داخل امؤشر يساريمين ال root

P= root → right → left ;

6- which of the foolowing ways below is a pre order traversal:

1-root->left->right

2- root ->right->left

7- which of the foolowing ways below is a pre order traversal:

1-root->left->right

2- root ->right->left

8- tree cannot cotain cycles:

1- true

2- false