كلية علوم الحاسوب وتكنولوجيا المعلومات

المرحلة الثانية

مادة الخورازميات

م.م فرح معاذ جاسم

١. Introduction

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms:

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

1.2 Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics:

- **Unambiguous**: Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input**: An algorithm should have 0 or more well-defined inputs.
- **Output**: An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness**: Algorithms must terminate after a finite number of steps.
- **Feasibility**: Should be feasible with the available resources.

- **Independent**: An algorithm should have step-by-step directions, which should be independent of any programming code.

1.3 How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is a problem and resource-dependent. Algorithms are never written to support a particular programming code.

Algorithm writing is a process and is executed after the problem domain is well-defined. So it should know the problem domain, for which we are designing a solution.

The common constructs such as like loops (do, for, while), flow-control (if-else), etc are all programming languages share these basic codes .so, it can be used to write an algorithm.

Example:

Design an algorithm to add two numbers and display the result.

Step 1 − START
Step 2 − declare three integers **a**, **b** & **c** ← Input
Step 3 − define values of **a** & **b**
Step 4 − add values of **a** & **b**
Step 5 − store output of step 4 to **c**
Step 6 − print **c** ← Output
Step 7 − STOP ← terminate

1.4 Algorithm Analysis

The efficiency of an algorithm can be analyzed at two different stages, before implementation, and after implementation. They are the following

1. Priori Analysis :(execution time) this is a theoretical analysis of an algorithm. The efficiency of an algorithm is measured such as processor speed, are constant and has no effect on the implementation.

2. Posterior Analysis :(running time) this is an empirical analysis of an algorithm. The selected algorithm is implemented using a programming language. This is then executed on the target computer machine. In this analysis, actual statistics like running time and space required, are collected.

1.5 Algorithm Complexity

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

- **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

1.5.1Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

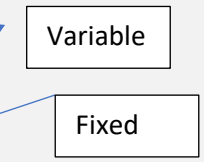
- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory
- allocation, recursion stack space, etc.

Space complexity S(x) of any algorithm x is S(x) = C + S(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I.

Following is a simple example that tries to explain the concept.

Algorithm: SUM(A, B)
Step 1 - START
Step 2 - C ← A + B + 10
Step 3 - Stop



S(x) = C + S(I) → s(x)=1+2→=3

1.5.2 Time complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

the time required by an algorithm falls under three types :

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

Homework

Homework1: is the following steps is an algorithm ? justifying your answer

Step 1: Start

Step 2: Declare variables a,b,c and d,

Step3:Declear x=1.

Step 4: Read variables a,b and c.

Step 5: If a > b

If a > c

a+c

Else

b=a

Else

If b > c

Display b is the largest number.

Else

Display c is the greatest number.

Step 6: while x>0

x=x+1

Step 6: Stop

****

Homework 2: the following an algorithm to find all roots of a quadratic equation a(x)^2+bx+c=0.

compute the Space complexity of this algorithm

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

D = b2-4ac

Step 4: If D =0

r1 =(-b+D)/2a

r2 = (-b-D)/2a

Display r1 and r2 as roots.

Else

Calculate real part and imaginary part

rp = b/2a

ip =(-D)/2a

Display rp+j(ip) and rp-j(ip) as roots

Step 5: Stop

كلية علوم الحاسوب وتكنولوجيا المعلومات

المرحلة الثانية

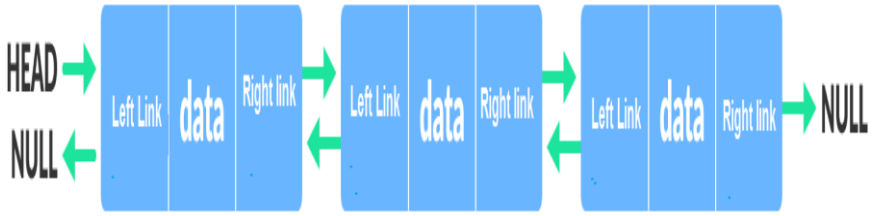
مادة الخورازميات

م.م فرح معاذ جاسم

DOUBLE LINKED LIST (DLL)

- A double linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- Left link.
- Data.
- Right link.



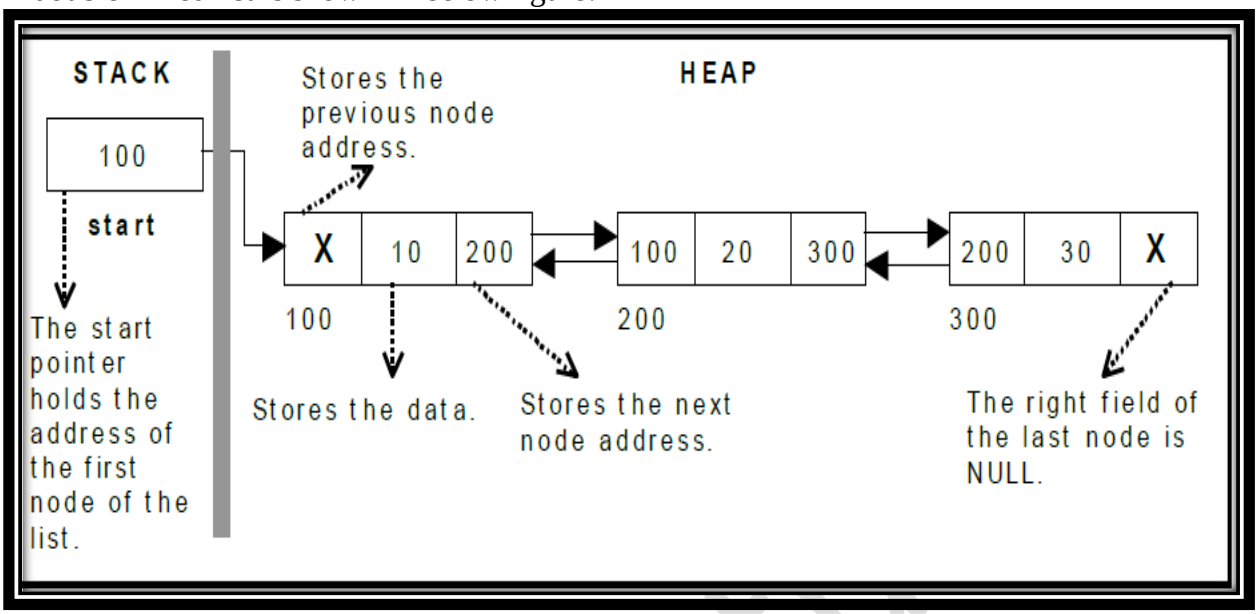
- The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

- Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

- The basic operations in a double linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

A double linked list is shown in below figure:



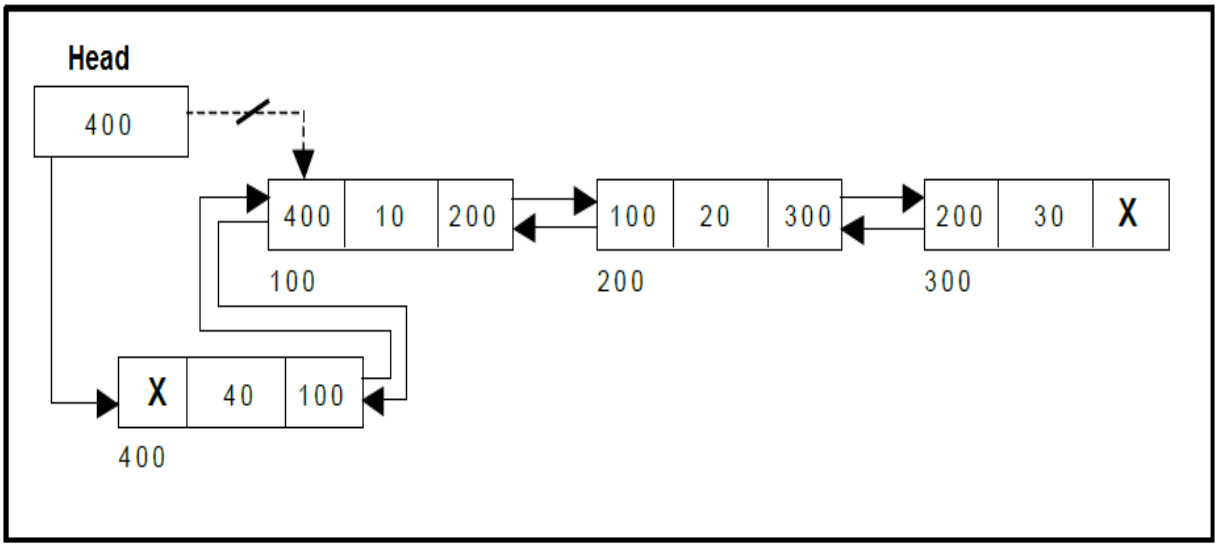
- The following code gives the structure definition

```
struct node
{
        int Data;
        node *LL,*RL;
};
node *head=NULL;
```

INSERT NODE AT FIRST OF DOUBLE LINKED LIST:

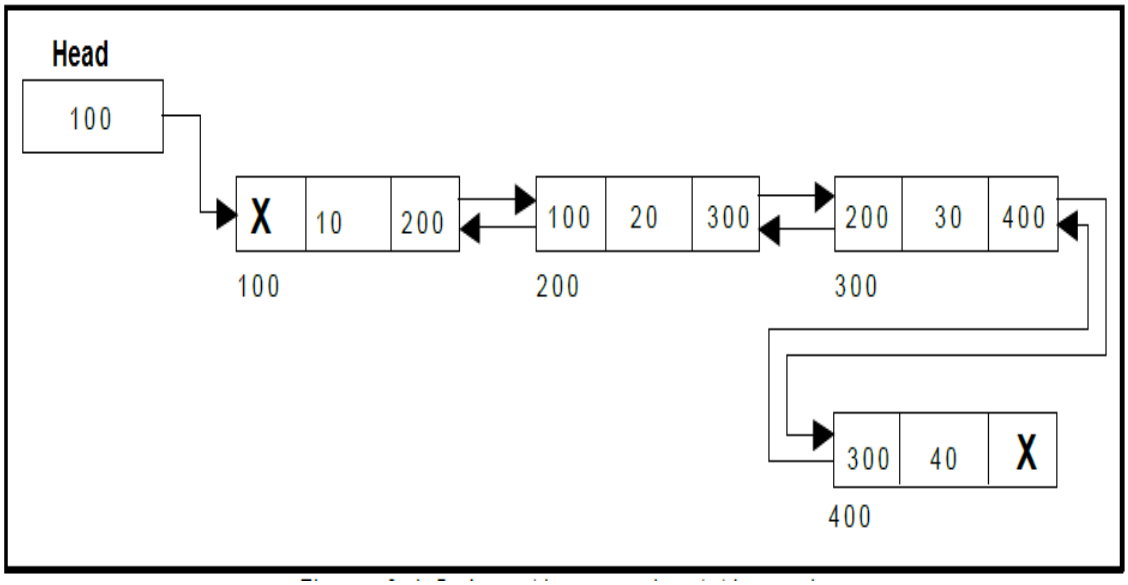
```
void insertF(int n)
{
        node *X=new node;
        X->Data=n;
        X->LL=NULL;

        if (head==NULL)
        X->RL=NULL;
        else
        {X->RL=head;
        head->LL=X;
        }
        head=X;
}
```



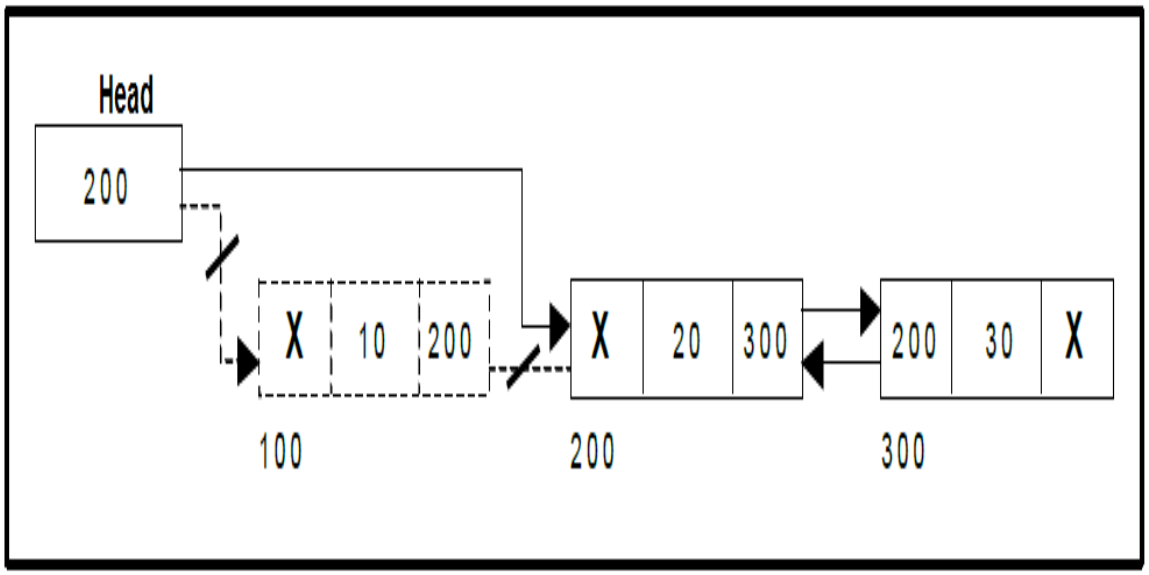
INSERT NODE AT THE END OF DOUBLE LINKED LIST:

```
void insertEnd(int n)
{
        node *X=new node;
        X->Data=n;
        X->RL =NULL;
        node *q=head;
        while (q->RL!=NULL)
        q=q->RL;
        q->RL =X;
        X->LL =q;
}
```



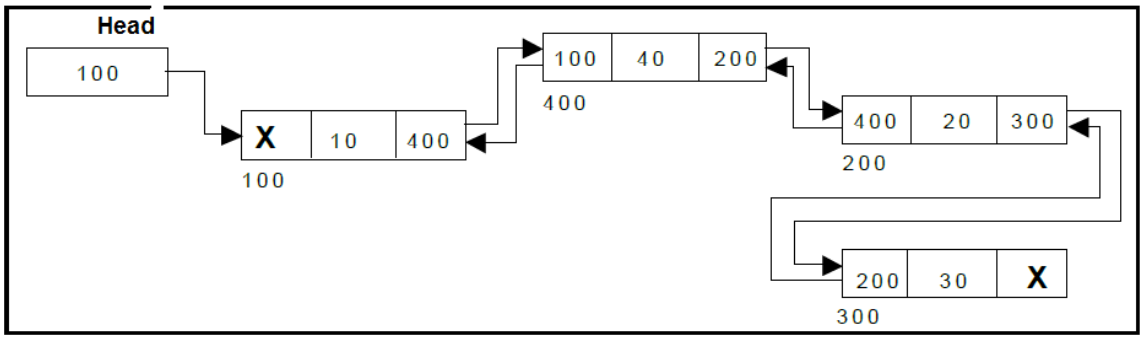
DELETE THE FIRST NODE ON DLL

```
void deletF()
{
 if (head!=NULL)
 {
        head=head->RL;
 head->LL=NULL;
 }
 else
 head=NULL;
}
```



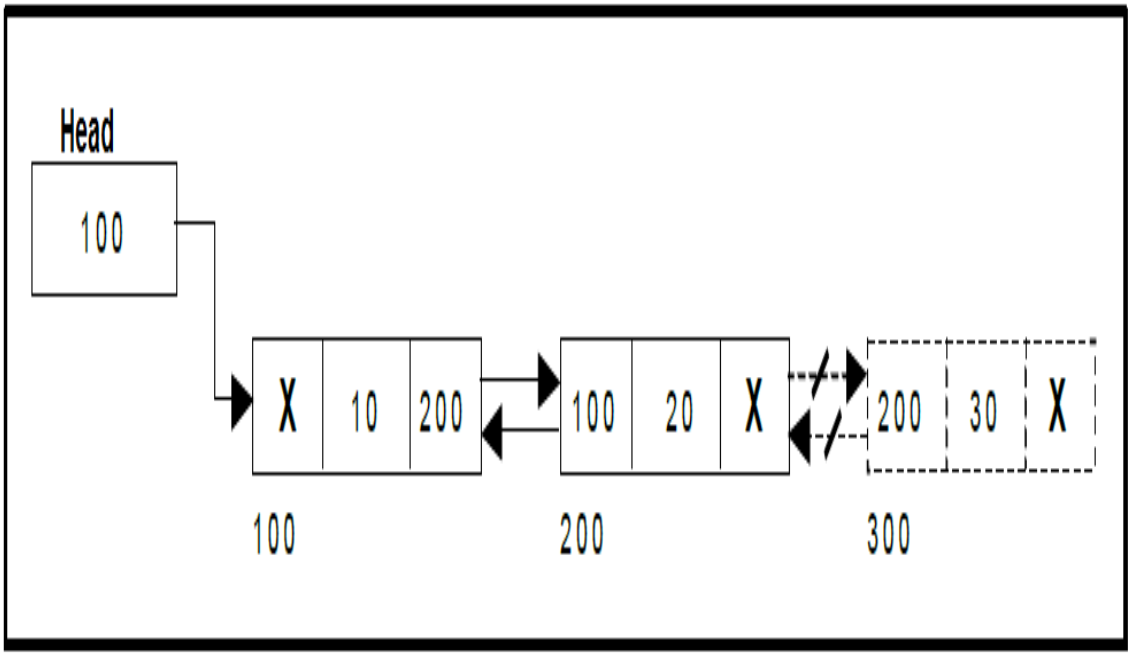
INSERT NODE AT THE MID OF DOUBLE LINKED LIST:

```
void insertMid(int n, int y)
{
        node *X=new node;
        X->Data=n;
        node *q=head;
        while (q->Data!=y)
        q=q->RL;
        X->RL=q->RL;
        q->RL->LL=X;
        q->RL=X;
        X->LL=q;
}
```



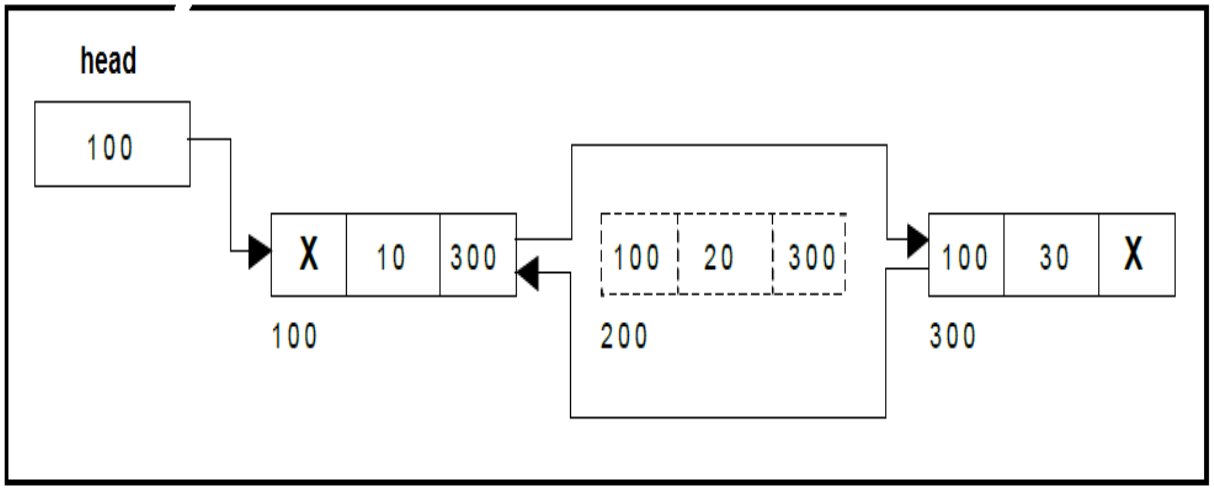
DELETE THE END NODE ON DLL

```
void DeletEnd()
{
        node *q=head;
        while (q->RL->RL!=NULL)
        q=q->RL;
        q->RL=NULL;
}
```



DELETE ANY MID NODE ON DLL

```
void DeleteMid(int y)
{
node *q=head;
while(q->data!=y)
q=q->RL;
q->RL=q->RL->RL;
q->RL->RL->LL=q;
}
```



Homework:

1.create Double Linked List (DDL) with those nodes(Data, Adderess) ,(3,100),(5,200),(8,300),(45,400),(30,600) .

2.insert the node (20,600)to the end of DDL

3.Delete the first node

4.Delete node with 45=value

5.insert first (4,60),

Draw every step

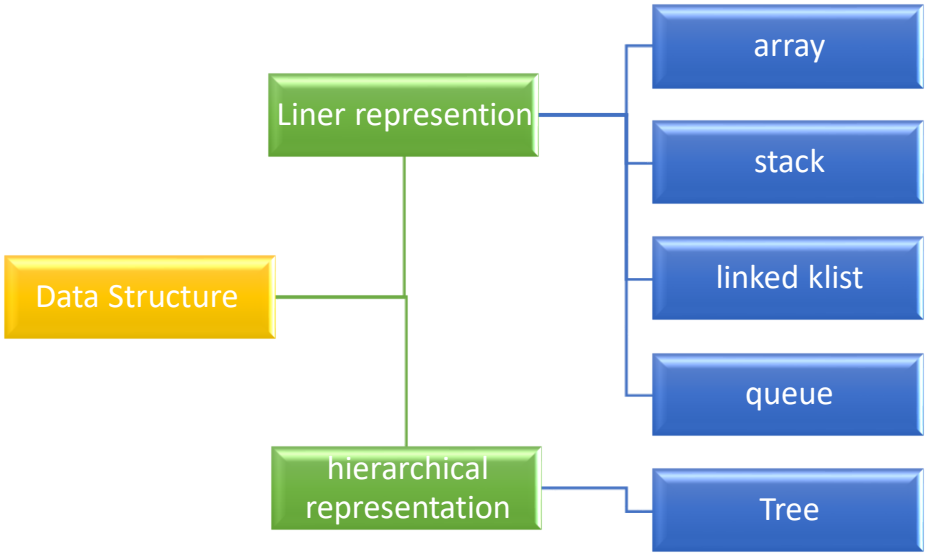
كلية علوم الحاسوب وتكنولوجيا المعلومات

المرحلة الثانية

مادة الخورازميات

م.م فرح معاذ جاسم

1. Introduction:



2. Tree

A tree is a data structure that has hierarchal relationships between its individual data items.

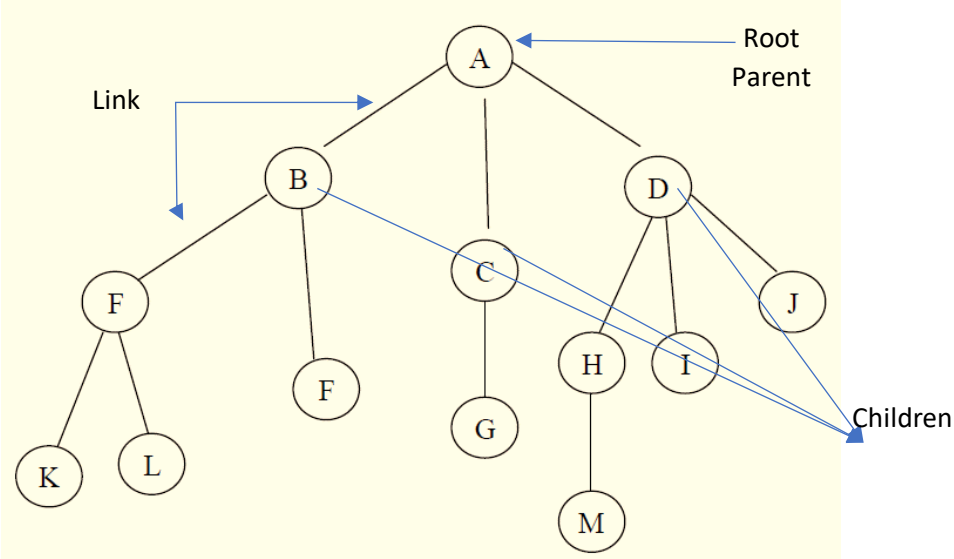


Figure (1)

Important notes:

- The higher node A of the tree in figure(1) is the root.

-The nodes B, C and D which are directly connected to the root node are the children of the root.

- The link between a parent and its child is called branch.

- The root of the tree is the ancestor of all nodes in the tree.

- Each node may be the parent of any number of nodes in the tree.
- The root of the tree (2) has level 0, and the level of any other node in the tree is one more than the level of its father.
- The depth of the tree is the maximum level of any leaf in the tree.

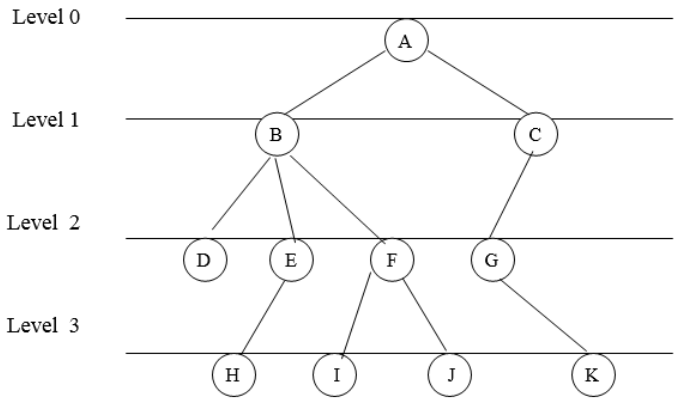


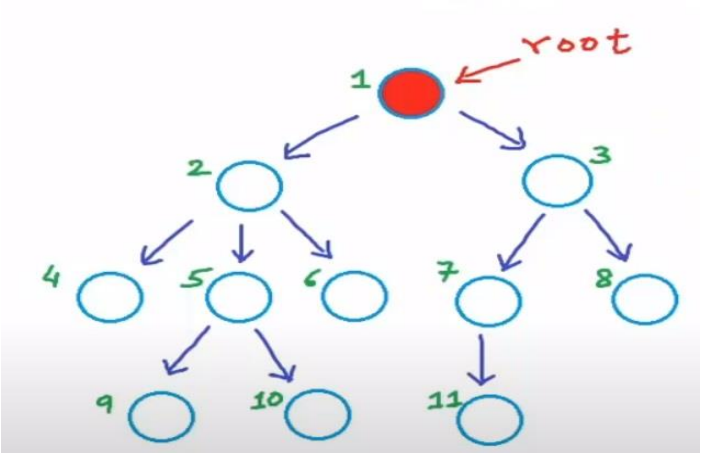
Figure (2)

-all nodes in the last level of a tree is called a leaf node (H, I, J, K).

- A tree is a recursive data structure because every node in a tree may have children so that consider a subset tree of the primary tree.

- The children of a given parent is the set of sub trees. The node in the tree of figure(2) A=[B,C,D,E,F,G,H,I,J,K] ,B=[D,E,F,H,I,J], C=[G,K], E=[H],F=[I,J], G=[K]. where B,C,E,F,G are sub tree of A

Example: represented array A=[1,2,3,4,5,6,7,8,9,10,11] in a tree .



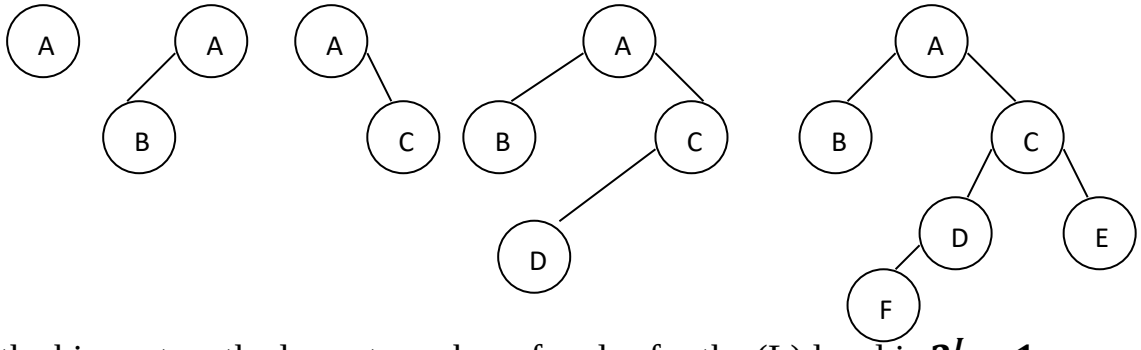
Examples tree in real life:

• Family tree • Table of contents of a book • Class inheritance hierarchy

• Computer file system (folders and subfolders)

3. Binary Tree

A binary tree is the tree that is characterized by the fact that any node can have at most two branches. (i.e) there is no node with degree greater than two). A binary may be empty or consist of a root and two disjoint binary tree called the left subtree and the right subtree.



In the binary tree the largest number of nodes for the (L) level is $\mathbf{2^L - 1}$

Example :tree have 3 level what are the largest number of nodes?

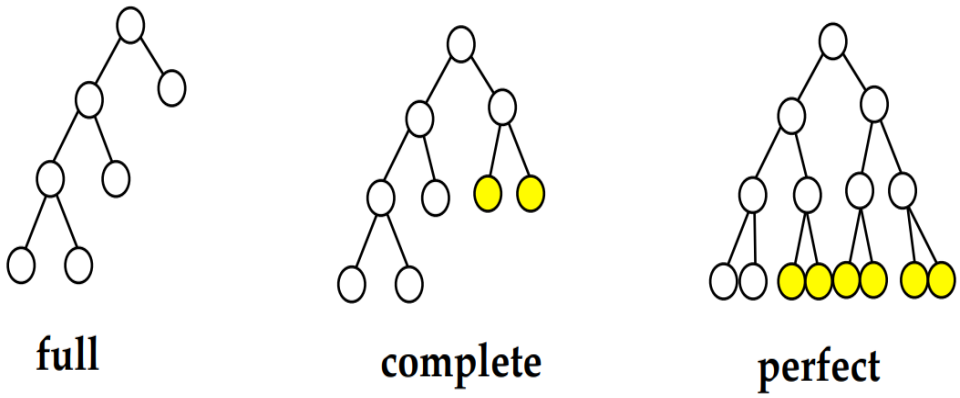
Node num= $2^3 - 1 = 7$

4.Full , Complete , Perfect Binary Trees

• If every node has either 0 or 2 children, a binary tree is called **full.**

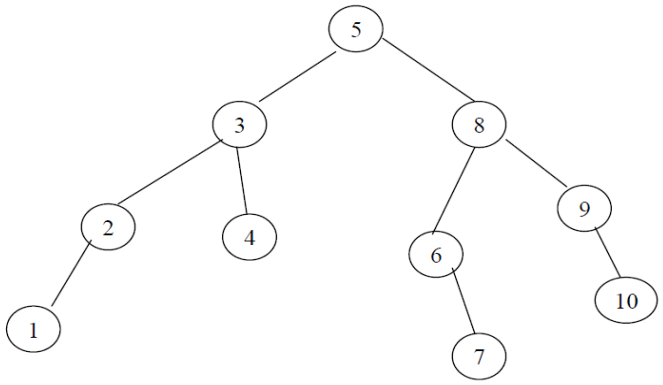
• If the lowest d-1 levels of a binary tree of height d are filled and level d is partially filled from left to right, the tree is called **complete.**

• If all d levels of a height-d binary tree are filled, the tree is called **perfect.**



5. Linear Representation

The linear representation method of a binary Search tree uses a one – dimensional array of size (2^{d+1}-1) where s is the depth of the tree. In the following tree, d =3 and this tree require an array of size (2^{3+1}-1) = 15 to be represented.



Once the size of the array has been determine, the following method is used is represent the tree:

1- Store the root in the 1st location of the array.

2- If a node is in the1st location of the array, Store its left child at location (2n), and its right child at location (2n+1).

The main advantages of this method are:

- Its simplicity and the fact that given a child node, its parent node can be determined immediately .If a child node is at location N in the array, then its parent node can be determined immediately. If a child is at location N in the array, then its parent node is at location N / 2.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5	3	8	2	4	6	9	1	-	-	-	-	7	-	10

The disadvantages of this method:

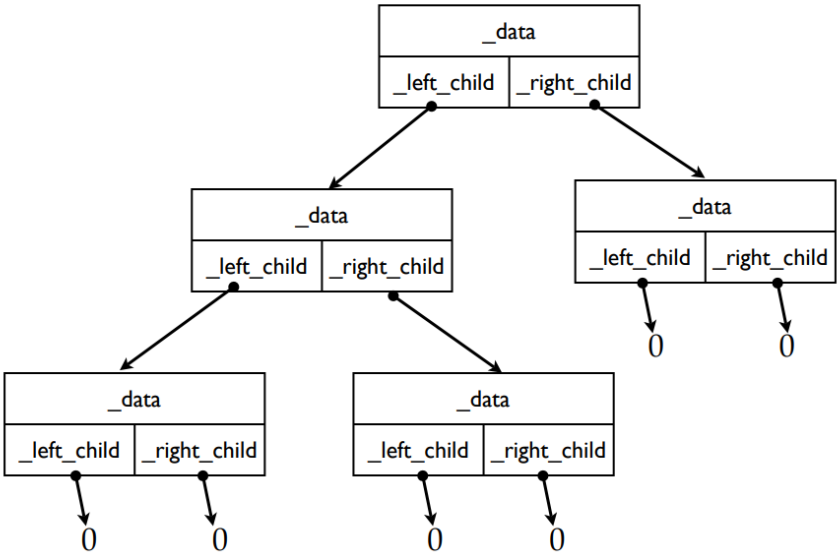
- Insertion and deletion of a node causes Considerable data monument up and down the array, using an excessive amount of processing time, because insertions and deletions are the major data processing activities.
- Wasted memory location.

6. Linked Representation of a Binary Tree

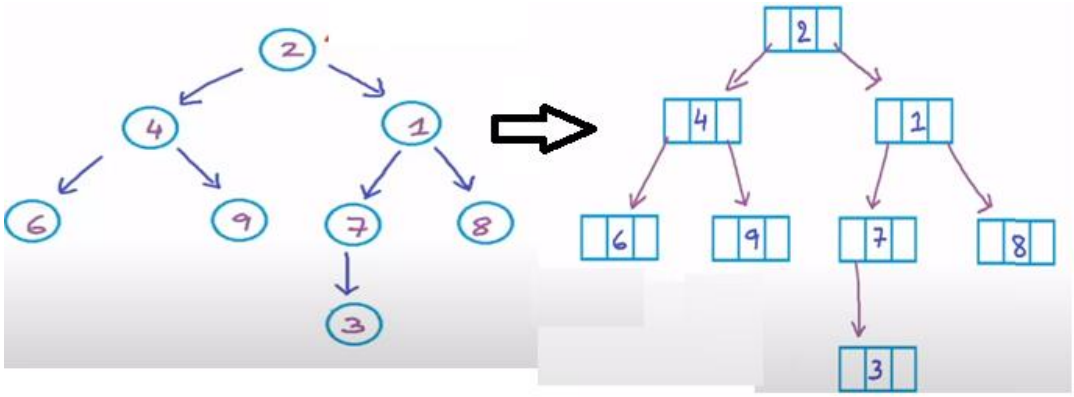
Because each node in a binary tree may have 2 children, a node in a linked representation has 2pointer fields, one for each child, and one or more data fields containing specific information about the node itself. When anode has no children the corresponding pointer fields are NULL. As in a linear linked list, the first node in the tree is pointed to by an external pointer.

Tree in c++ by using Linked list

```
Struct  Node{
 Int data;
Node*   left;
Node * right;        };
```



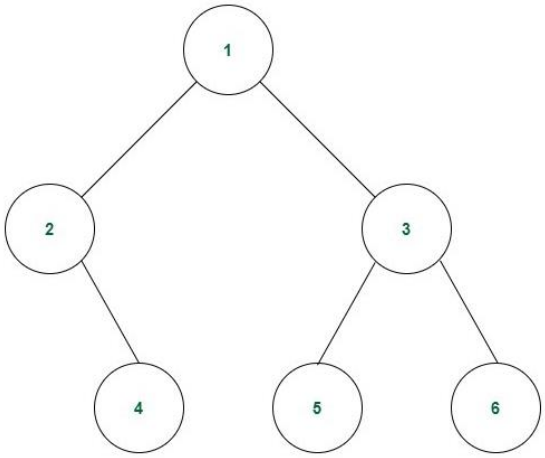
Example:



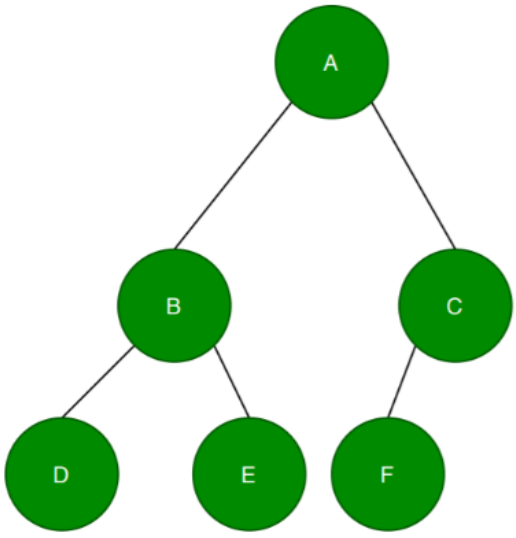
Homework

1. what are the disadvantages of represent a binary Search tree by using linear representation method (one -dimensional array)?

2. The following tree is not a perfect binary tree .why? convert it to perfect tree?



3. Is this tree a full binary tree.?? Justify your answer.



4.we have an array [10, 2,5,8,4] convert this value to a binary tree.

1. Binary Search Trees

It is the binary tree in which the value of the element of the left branch (child) of any node is less than the value of the element of that node as the father (father) and the value of the element of the right branch (child) is greater than the value of the element of the node (father).

2. Operations on Binary Search Trees

- Insert: add a new node to a binary search tree.
- Print: Binary tree traversal prints all elements in the binary search tree.
- Search: search for a node at the binary search tree.
- Delete: delete a node from the binary search tree.

1.Insert function

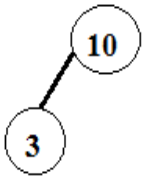
```
Node*  insert (Node* root, int data )
{
        if (root==NULL)
        {
                        root=get_newnode(data);
        }
        else if ( data<= root->data)
        {
                root->left=insert(root->left,data);
        }
else
{
        root->right=insert(root->right,data);
}

        return root;
}
Node* get_newnode(int data)
{
        Node* newnode= new Node();
        newnode->data=data;
        newnode->left=NULL;
        newnode->right=NULL;
        return newnode;
}
```

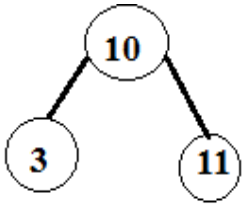
Example : show the Binary Tree search for the following :

1. insert(root,10)

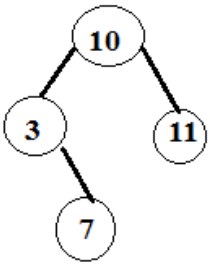
2. insert(root,3)



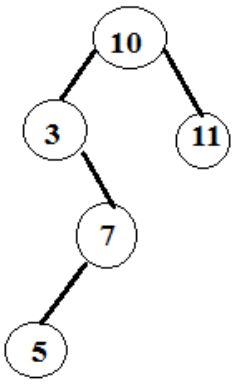
3. insert(root,11)



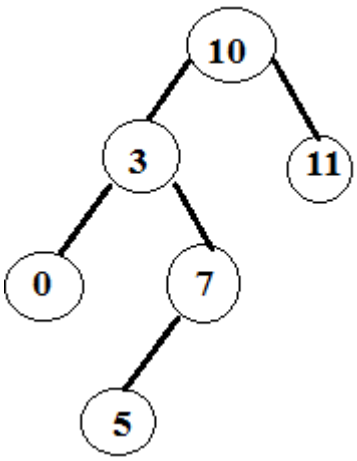
4. insert(root,7)



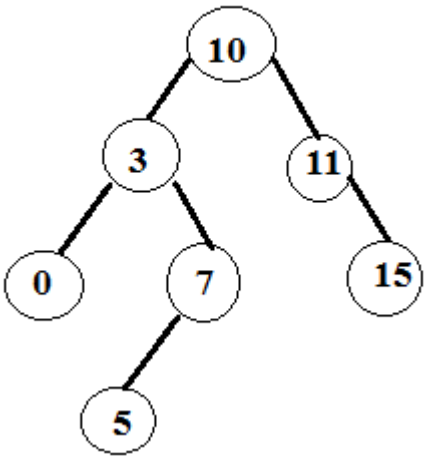
5. insert(root,5)



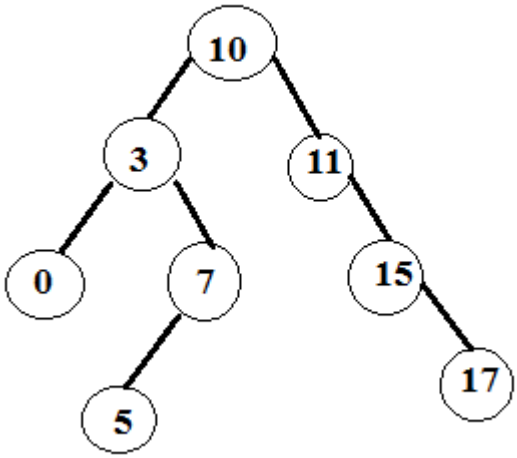
6. insert(root, 0)



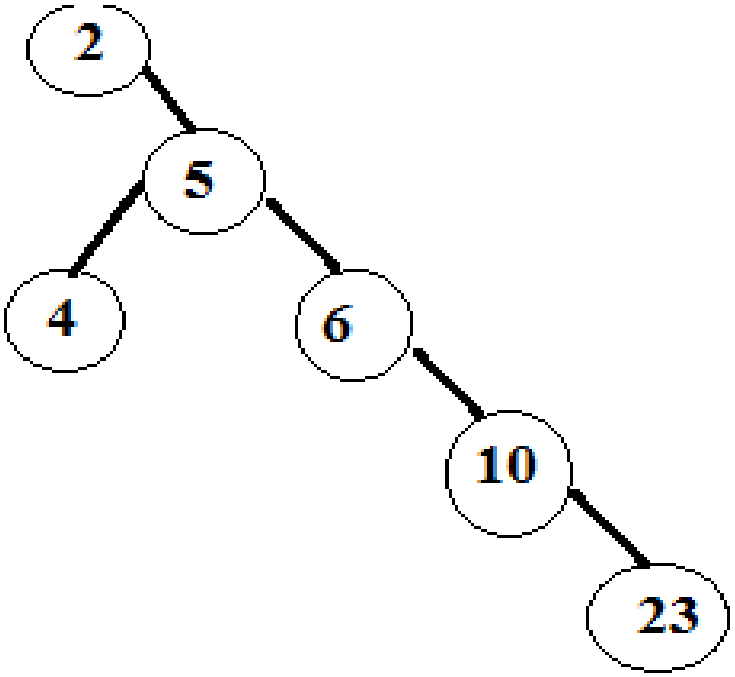
7. insert (root,15)



8.insert (root,17)

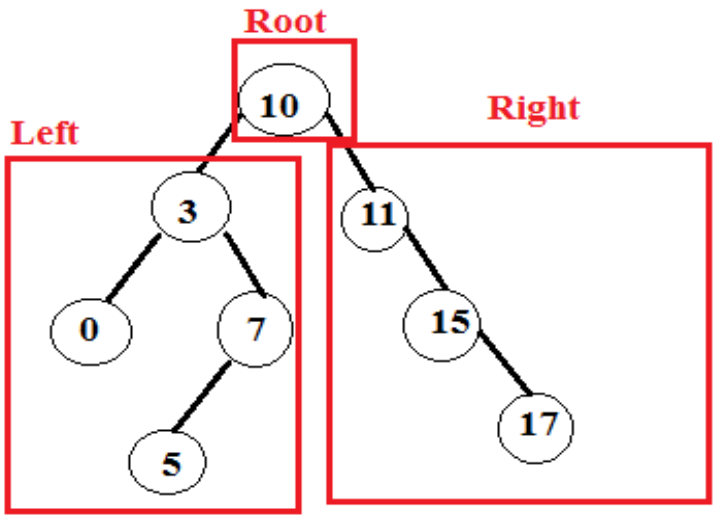


Example2: show the Binary Tree search for the following :2,5,6,10,23,4



2.Binary Tree Traversal

To print the information of nodes within a tree we need to visit each of these nodes and then print its information. There are three different method to visit the tree nodes, each of them prints nodes information in different order.



A- In order print function (left – root – right)

These method means traverse and print from the smallest to the largest values. We first need to print the roots left subtree, then we print the value I the root node finally print the value in the root's right subtree.

Inorder print = 0 3 5 7 10 11 15 17

```
void inorder(Node* p)
{
        if (p!= NULL)
        {
inorder(p->left);
cout<< p->data<<"=>";
inorder(p->right);
        }
}
```

B-Preorder print function (Root - left - right)

Preorder method traverse end print each node information before its left and right subtrees. Therefore, the information of the tree in the previous example are print as follows: **preorder print=10 3 0 7 5 11 15 17**

```
void preorder(Node* p)
{
if (p!= NULL)
{
cout <<p->data<<"=>";
preorder(p->left);
preorder(p->right);
}
}
```

C.Postorder print function (left – right – root)

Postorder method traverse and

print each node information after its left and right subtree. The information of the same example of the previous two methods are print as follows: **Postorder** = **0 5 7 3 17 15 11 10**

```
void postorder (Node* p)
{
if (p!= NULL)
{
postorder (p->left);
postorder (p->right);
cout <<p->data<<"=>";
}
```

١Homework

.Which of the following is false about a binary search tree؟

a) The left child is always lesser than its parent

b) The right child is always greater than its parent

c) The left and right sub-trees should also be binary search trees

d) In order sequence gives decreasing order of elements

.٢How to search for a key in a binary search tree؟

.٣What does the following piece of code do؟

```
void ?????(Node* p(
}
if (p!= NULL(
}
cout <<p->data؛"<=">>
)؟؟؟؟؟p->left؛(
)؟؟؟؟؟p->right؛(
{
{
```

a)Preorder traversal

b) Inorder traversal

c) Postorder traversal

d) Level order traversal

.٤Construct a binary search tree with the below information.

The key of a binary search tree are 15,4,11,0,3,2,8,18.

.٥Construct a binary search tree with the below information.

The preorder traversal of a binary search tree 10, 4, 3, 5, 11, 12.

الانتباه على صيغة السؤال جيدا

.٦The number of edges from the root to the node is called _____ of the tree.

a) Height

b) Depth

c) Length

d) Width

.٧The number of edges from the node to the deepest leaf is called _____ of the tree.

a) Height

b) Depth

c) Length

d) Width

.٨What is a full binary tree؟

a) Each node has exactly zero or two children

b) Each node has exactly two children

c) All the leaves are at the same level

d) Each node has exactly one or two children

.٩What does the following piece of code do؟

```
void ???(Node* p(
}
        if (p!= NULL(
        }
)؟؟؟p->left؛(
cout<< p->data؛"<=">>
)؟؟؟p->right؛(
        {

{
```

a)Preorder traversal

b) Inorder traversal

c) Postorder traversal

d) Level order traversal

.١٠What is a complete binary tree؟

a) Each node has exactly zero or two children

b) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from right to left

c) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right

d) A tree In which all nodes have degree 2

Construct a binary tree by using postorder and inorder sequences given below.

١١. Construct a binary search tree with the below information.

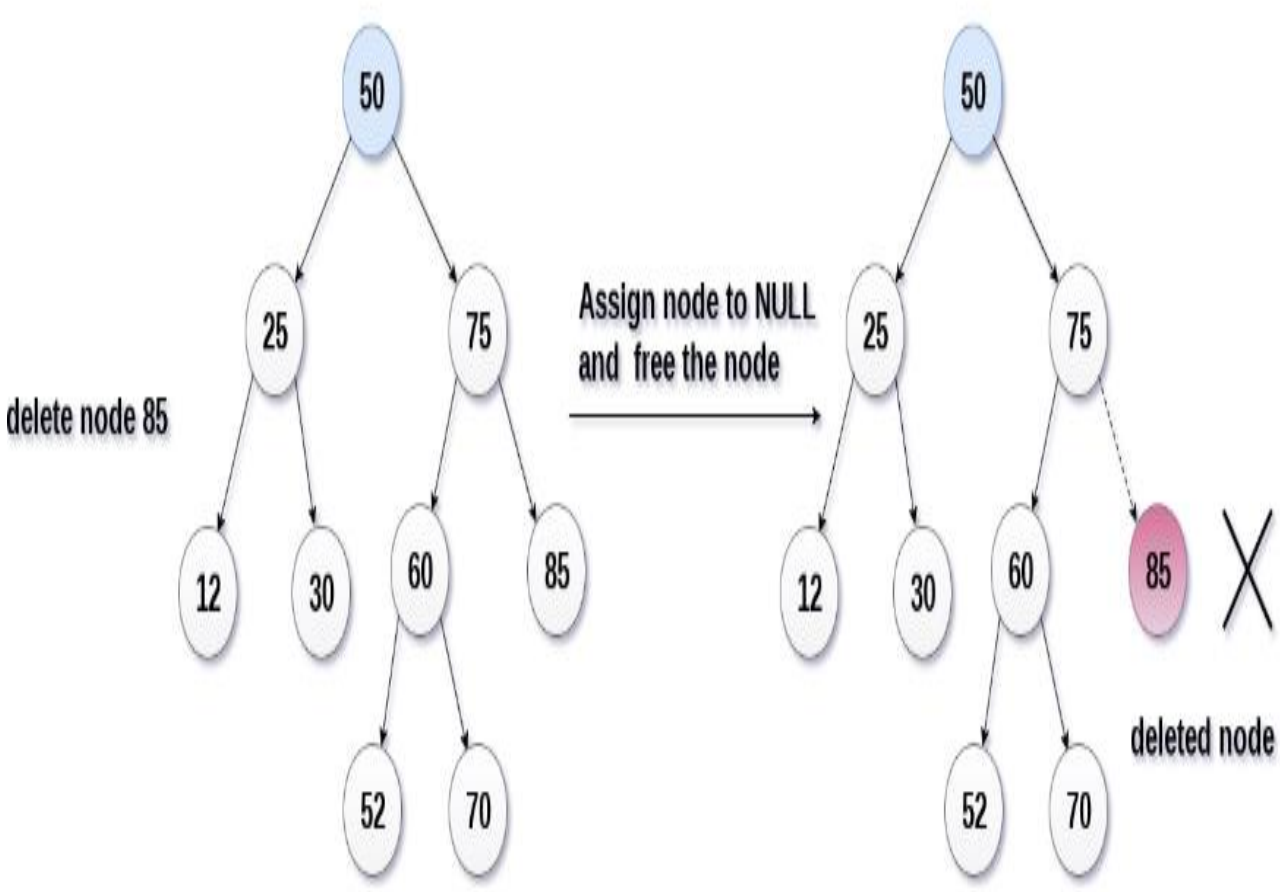
The key of a binary search tree are p,x,a,r,f,z,i

Delete Node from Binary tree search

Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary search tree doesn't violate. There are three situations of deleting a node from binary search tree**:**

1: The node to be deleted is a leaf node

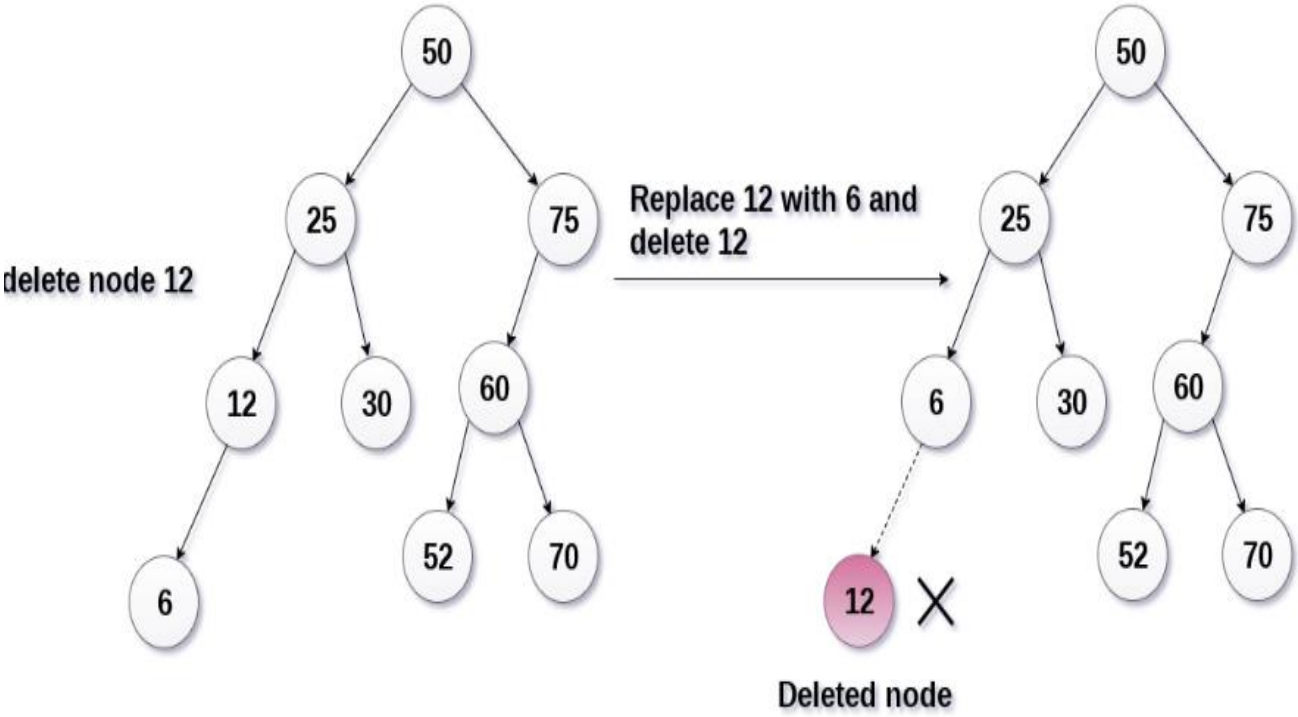
It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space. In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



2: The node to be deleted has only one child:

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

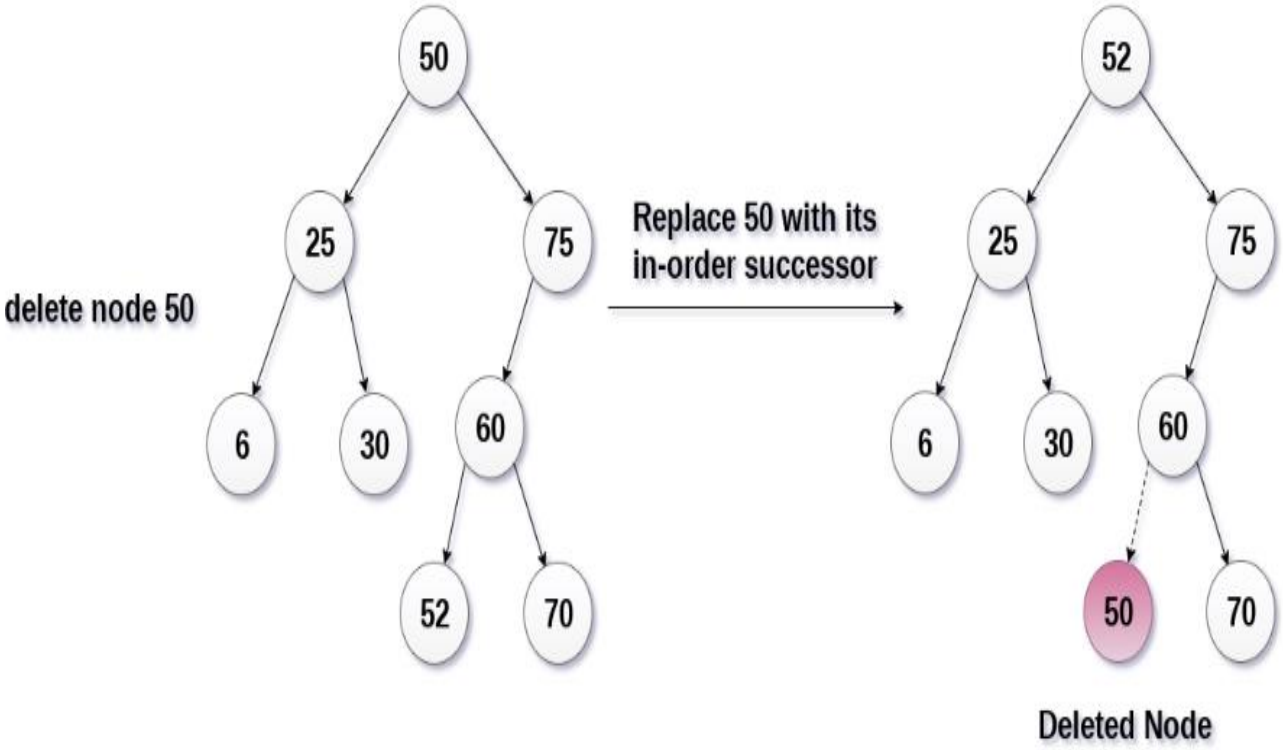
In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



3:The node to be deleted has two children.:

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.6, 25, 30, 50, 52, 60, 70, 75. replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



```
Node* Delete(struct Node *root, int data)
{
        If (root == NULL) return root ;
        else if(data <  root->data)     root->left = Delete(root->left,data);
        else if (data > root->data)    root->right = Delete(root->right,data);
        //... I found you, Get ready to be deleted
        else {

                 //Case 1:  No child
                If (root->left == NULL && root->right == NULL )
                      {
                        delete root;
                        root = NULL;
                }

                //Case 2: One child

                else if(root->left == NULL )

                {
                        struct Node *temp = root;
                        root = root->right;
                        delete temp;
                }
                else if(root->right == NULL)
                {
                        struct Node *temp = root;
                        root = root->left;
                        delete temp;
                {
                 //case 3: 2 children

                else
                      {
                        struct Node *temp = FindMin(root->right);
                        root->data = temp->data;
                        root->right = Delete(root->right,temp->data);
                }
        }

        return root;

}
```

```
Node* FindMin(Node* root)
{
while(root->left != NULL) ؛
root = root->left
return root؛
}
```

كلية علوم الحاسوب وتكنولوجيا المعلومات

المرحلة الثانية

مادة الخورازميات

م.م فرح معاذ جاسم

6.1 Sorting Algorithm

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order.

Other Definition: **Sorting** is the process of arranging a set of graphical elements according to the value of a field (or fields) called a key (ascending) or descending.

6.2 Sorting Applications and Purposes

- Uniqueness testing
- Deleting duplicates: solve the problem of similarity restrictions.
- Prioritizing events :To simplify the processing of files
- Frequency counting
- Reconstructing the original order
- Set intersection/union
- Finding a target pair x, y such that x+y = z
- Efficient searching: To increase the efficiency of the search algorithm for an item.

5.3 Steps in the sorting process

The steps of the sorting algorithm are summarized in the following stages:

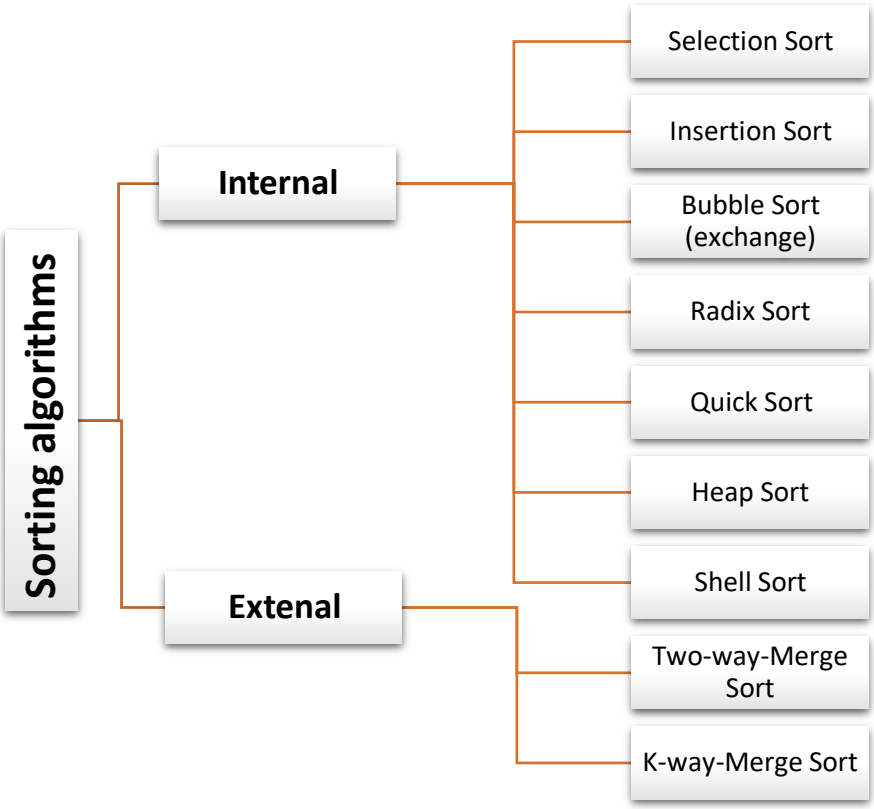
1- Reading the key field. Which mean the input data that will sort it.

2- Inference (deduction) the location of the element in the new arrangement.

3- Move the sorted element to its new location.

6.4 Types of sorting algorithms

There are two types of internal sort (in the main memory) ,and external sort (the secondary storage).see the figure below:



6.5 key determinants of the sorting algorithm selection

The testing of any of the ranking algorithms should be in light of a number of factors, the most important of which are:

1 - The volume of data stored.

2. Storage type (main memory, disk, and tape).

3. Degree of data order (unordered, semi-ordered)

6.6 Selection sort

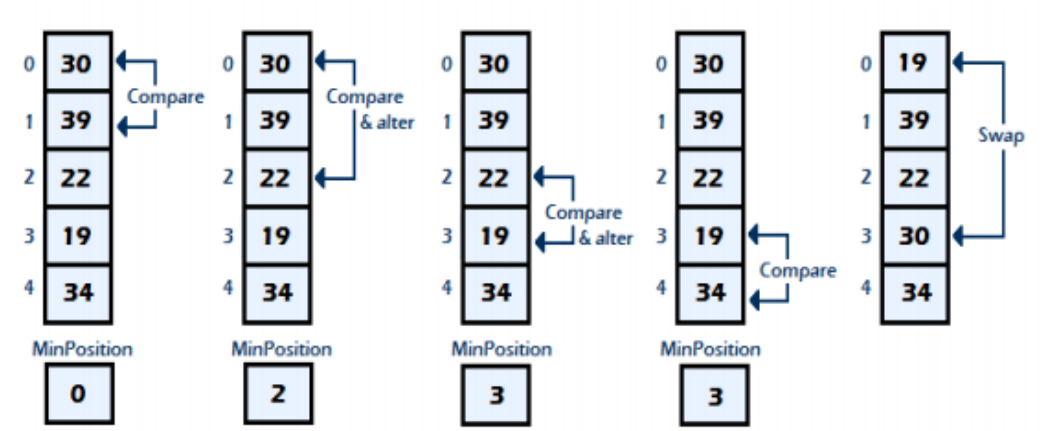
The algorithm for this arrangement is summarized by the following steps:

1- Find the smallest item in the list and replace it from its location with the item in the first location in the list.

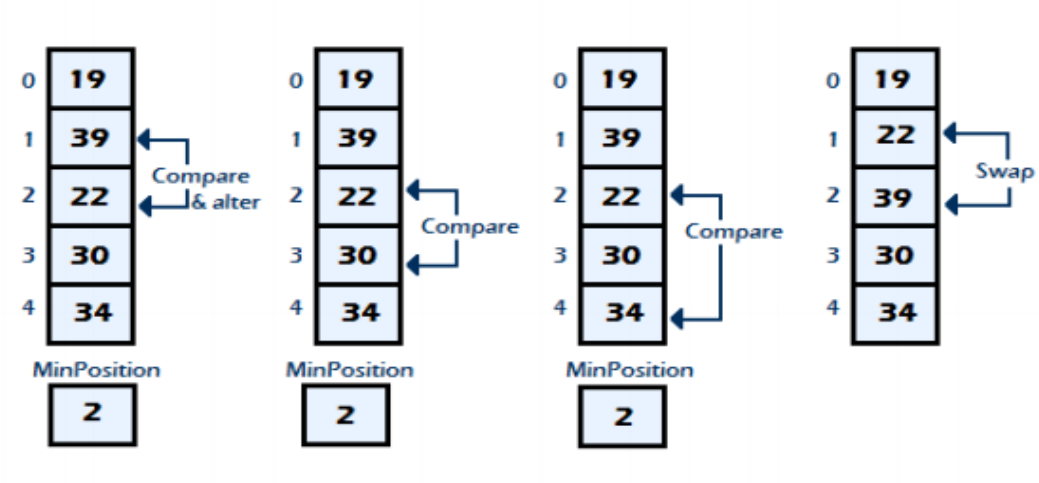
2- Find the smallest element in the remaining part of the list and replace it from its location with the element in the second location in the list.

3- We continue in this process until we reach the end of the list.

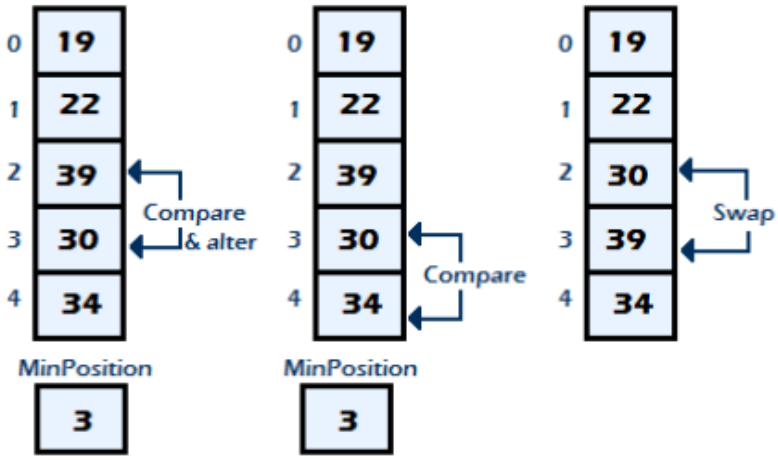
Example: sort the following list in ascending order (30 39 22 19 34)



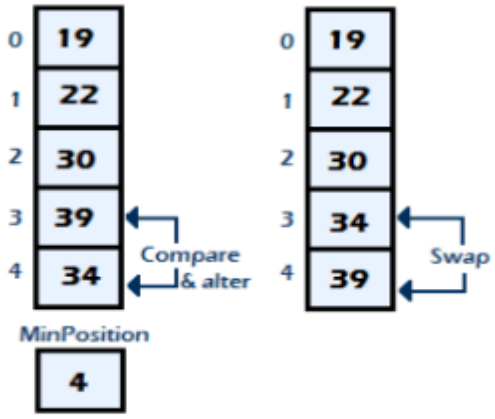
First Pass



Second Pass



Third Pass



Fourth Pass

Example: sort the following list in ascending order (8 3 9 7 2 6 4).

6	5	4	3	2	1	القائمة الأصلية
2	2	2	2	2	2	8
3	3	3	3	3	3	3
4	4	4	4	9	9	9
6	6	6	7	7	7	7
7	7	8	8	8	8	2
8	8	7	6	6	6	6
9	9	9	9	4	4	4

Example: sort the following list in ascending order (22 3 16 7 0 5 9).

6	5	4	3	2	1	القائمة الأصلية
0	0	0	0	0	0	22
3	3	3	3	3	3	3
5	5	5	5	16	16	16
7	7	7	7	7	7	7
9	9	22	22	22	22	0
16	16	16	16	5	5	5
22	22	9	9	9	9	9

Example: sort the following list in ascending order **(re, xy, zn, or, py, cz, ab)**

6	5	4	3	2	1	القائمة الأصلية
ab	ab	ab	ab	ab	ab	re
cz	cz	cz	cz	cz	xy	xy
or	or	or	or	zn	zn	zn
py	py	py	zn	or	or	or
re	re	zn	py	py	py	py
xy	xy	xy	xy	xy	cz	cz
zn	zn	re	re	re	re	ab

Selection sort function in c++

```
void slctsort(int ar[n],int n)
{
 int i,k,j,item,x,y;
 for(i=0;i<n-1;i++)
  {
   k=i;
   item=ar[i];
   for(j=i+1;j<n;j++)
    {
     if(ar[j]<item)
      {
       x=ar[j];
       ar[j]=item;
       item=x;
      }
    }
   y=item;
   item=ar[k];
   ar[k]=y;
  }
}
```

هنا الدالة تأخذ متغيرين مصفوفة وعدد عناصرها

هذا ال loop لكي يبحث على كل العناصر من العنصر الاول الى الاخير ممكن نبدأ من ١ لل n

Kمتغير اخزن بي القيمة الي وصلت بيها اقارن يعني رقم ال pass الي اذا وجدت رقم اصغر ابدل بمكانه واذا لا يبقى بدون تغير

هذا ال loop

Item = العنصر الي اقارن بي يعني رقم ال pass

اذا وجد رقم اصغر ممنه يعمل تبديل واذا لا يبقى ar[k] بدون تغير يعني خطوة ال no change

كلية علوم الحاسوب وتكنولوجيا المعلومات

المرحلة الثانية

مادة الخورازميات

م.م فرح معاذ جاسم

6.1 Inserting Sort

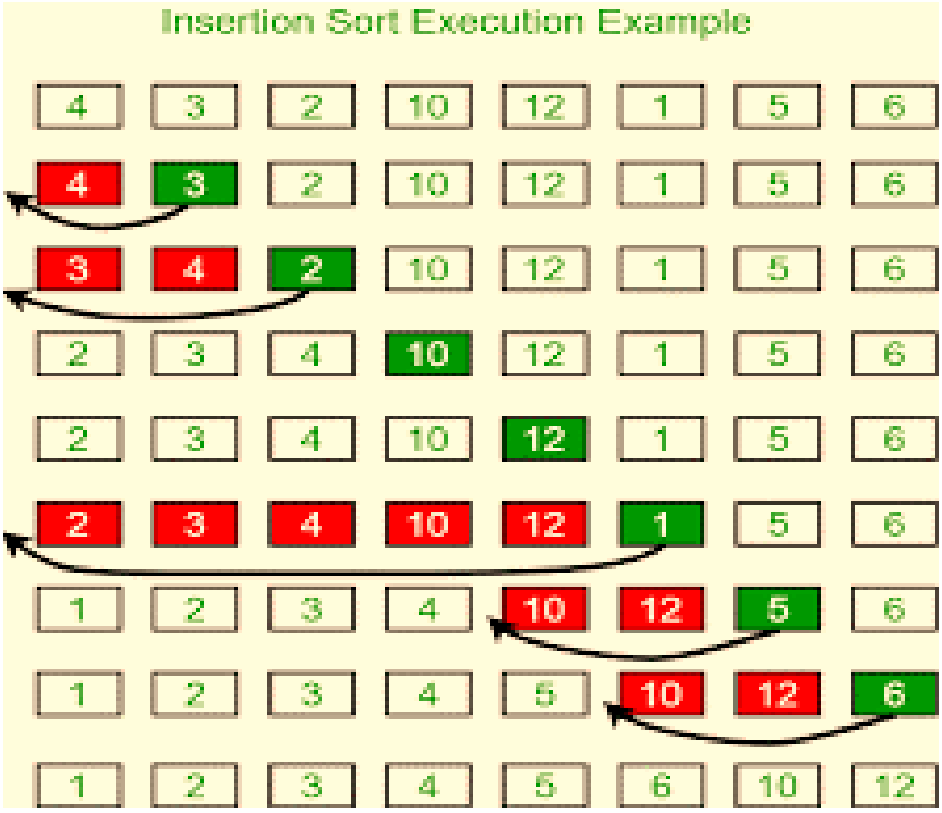
Inserting Sort is one of the internal type sort. It is used when the data is semi-ordered .The steps of this algorithm are summarized as follows:

1- We start with the second element i = 2 in the original list and compare it with the first element i = 1 and put them in order and be ascending to the top of the list.

2- We take the third element i = 3 in the original list and compare it with the introduction to the list that contains the first and second element and put it in its correct location with them.

3- We take the fourth element i = 4 in the original list and compare it with the introduction to the list that contains the three elements and put it in its correct position between them.

4- We continue in this process until the last component and we will get the list in order.



Example1: sort the following list items in ascending order (8 3 9 7 2 6 4)

Pass0	**8**	**3**	**9**	**7**	**2**	**6**	**4**	**القائمة الاصلية**
Pass1	**3**	**8**	**9**	**7**	**2**	**6**	**4**	**نفحص الموقع الاول مع الثاني من هو الاصغر ونبدل المكان ونزحف الباقي**
Pass2	**3**	**8**	**9**	**7**	**2**	**6**	**4**	**عندما وصل العداد لل 9 فهي اكبر من 3 و8 فبقيت في مكانها**
Pass3	**3**	**7**	**8**	**9**	**2**	**6**	**4**	**وصل العداد لل 2 فقام بفحصها من البداية وجدها اصغير من 3 فوضع ال2 مكان ال3 وزحف الباقي**
Pass4	**2**	**3**	**7**	**8**	**9**	**6**	**4**	**وصل العداد لل 6 فقارنها مع 2و3و7 ووضعها قبل ال7 وزحف الباقي**
Pass5	**2**	**3**	**6**	**7**	**8**	**9**	**4**	**وصل العداد لل 4 فقارنها مع 2 و3و6 فوجد مكانه الصحيح قبل ال 6 توضع ال4 قبل ال6 وتزحف الباقي**
Pass6	**2**	**3**	**4**	**6**	**7**	**8**	**9**	

Example2- sort (77, 33, 44, 11, 88, 22, 66, 55)

Pass0	**77**	**33**	**44**	**11**	**88**	**22**	**66**	**55**
Pass1	**33**	**77**	**44**	**11**	**88**	**22**	**66**	**55**
Pass2	**33**	**44**	**77**	**11**	**88**	**22**	**66**	**55**
Pass3	**11**	**33**	**44**	**77**	**88**	**22**	**66**	**55**
Pass4	**11**	**33**	**44**	**77**	**88**	**22**	**66**	**55**
Pass5	**11**	**22**	**33**	**44**	**77**	**88**	**66**	**55**
Pass6	**11**	**22**	**33**	**44**	**66**	**77**	**88**	**55**
Pass7	**11**	**22**	**33**	**44**	**55**	**66**	**77**	**88**

Example3- sort in descending order (0 4, 8, 11, 100, 22, 15, 55,2)

Pass0	**0**	**4**	**8**	**11**	**100**	**22**	**15**	**55**	**2**
Pass1	**4**	**0**	**8**	**11**	**100**	**22**	**15**	**55**	**2**
Pass2	**8**	**4**	**0**	**11**	**100**	**22**	**15**	**55**	**2**
Pass3	**11**	**8**	**4**	**0**	**100**	**22**	**15**	**55**	**2**
Pass4	**100**	**11**	**8**	**4**	**0**	**22**	**15**	**55**	**2**
Pass5	**100**	**22**	**11**	**8**	**4**	**0**	**15**	**55**	**2**
Pass6	**100**	**22**	**15**	**11**	**8**	**4**	**0**	**55**	**2**
Pass7	**100**	**55**	**22**	**15**	**11**	**8**	**4**	**0**	**2**
Pass8	**100**	**55**	**22**	**15**	**11**	**8**	**4**	**2**	**0**

Example 4: sort the following items(xray,rab, for,if ,car)

Pass0	**xray**	**rab**	**for**	**if**	**car**
Pass1	**rab**	**xray**	**for**	**if**	**car**
Pass2	**for**	**rab**	**xray**	**if**	**car**
Pass3	**for**	**if**	**rab**	**xray**	**car**
Pass4	**for**	**car**	**if**	**rab**	**xray**

6.2 Advantages for Insertion sort

1. Implementation of insertion sort is very easy as compared to sorting algorithms like quick sort, merge sort or heap sort.
2. Very efficient in the case of a small number of elements.
3. If the elements are already in sorted order it won't spend much time in useless operations and will deliver a run time of O(n).
4. It is a stable sorting technique, that is, the order of keys is maintained.

5. It requires constant "additional" memory, no matter the number of elements.
6. It can sort the elements as soon as it receives them.

6.3 Disadvantage

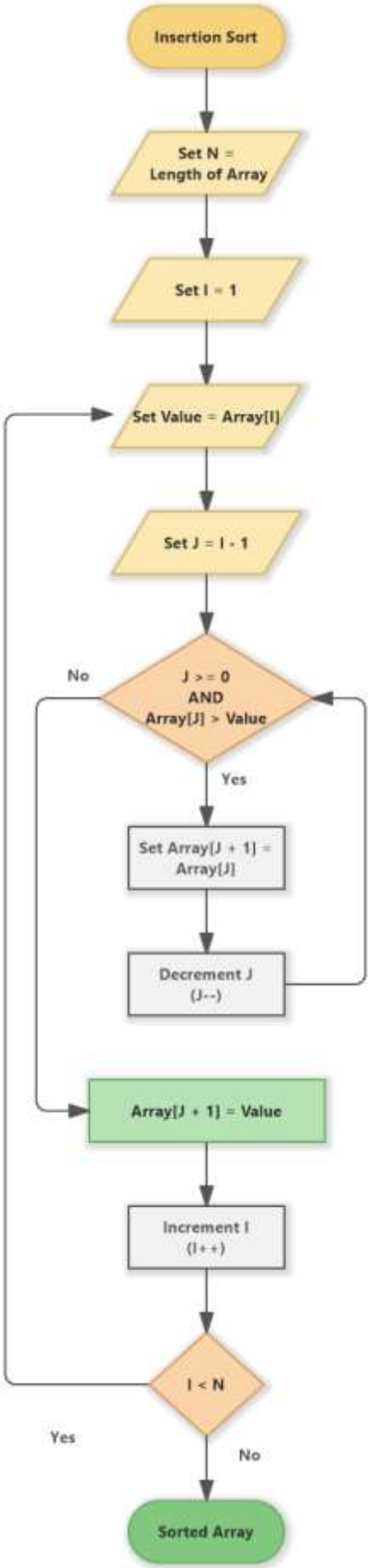
1. It is less efficient on list containing more number of elements.

2. As the number of elements increases the performance of the program would be slow.

3. Insertion sort needs a large number of element shifts.

Inserting Sort function in c++

```
const size=20;
int line[size],int i,m;
void insertionsort (int data[size],int n)
{
 int i,j,item;
 i=1;  عدد المراحل pass
while(i<n)
 {
  j=i;
  while((j>=1) && (data[j]<data[j-1]))
     {
         item= data[j];
         data[j]=data[j-1];
         data[j-1]=item;
          j--;
        }
   i++;
    }
  }
```


Insertion Sort
Set N = Length of Array
Set I = 1
Set Value = Array[I]
Set J = I - 1
J >= 0 AND Array[J] > Value
No
Yes
Set Array[J + 1] = Array[J]
Decrement J (J--)
Array[J + 1] = Value
Increment I (I++)
I < N
Yes
No
Sorted Array

كلية علوم الحاسوب وتكنولوجيا المعلومات

المرحلة الثانية

مادة الخورازميات

م.م فرح معاذ جاسم

6.1 Bubble Sort

The bubble sort used when the data items is not huge and semi-ordered. The idea of this method involves testing the smallest values and placing them in the list (that is, the small value floats to the surface.)

1. In the first stage (first pass) :We compare the two elements in the two locations (n-1), (n) and we exchange their location to be the smallest before the other, and we continue to the top of the list until we reach the comparison of the element in the second location with the element in the first site.
2. In the second pass: We compare in the same way as the previous one, but from the element in the location (n) to the element in the second site because the first site was chosen where the least valuable element in the previous step
3. Mention the above steps for (n-1) stages.

Example: sort the list by 8, 3, 9, 7, 2 ascending:

The fourth Pass	The third Pass i=3	The second Pass i=2	The first Pass i=1	The input
1	2 1	3 2 1	4 3 2 1	n
2	2 2	2 2 2	2 **8** 8 8	8
3	3 3	3 **8** 8	8 **2** **3** 3	3
7	7 **8**	8 **3** **3**	3 3 **2** **9**	9
8	**8** **7**	**7** 7 **7**	**9** 9 9 **2**	**7 (n-1)**
9	**9** 9	**9** 9 9	**7** 7 7 7	**2 (n)**

Example: sort the list by 13,20,5,4,3,0 ascending using bubble sort algorithm:

The fifth Pass Is the Out put	The fourth pass		The third pass			The second pass				The first pass					The input
1	2	1	3	2	1	4	3	2	1	**5**	**4**	**3**	**2**	**1**	**j**
0	0	0	0	0	0	0	0	0	0	**0**	**13**	13	13	13	13
3	3	3	3	3	3	**3**	**13**	13	13	13	**0**	**20**	20	20	20
4	4	4	4	**13**	13	13	**3**	**20**	20	20	20	**0**	**5**	5	5
5	5	**13**	13	**4**	**20**	20	20	**3**	**5**	5	5	5	**0**	**4**	4
13	**13**	**5**	**20**	20	**4**	**5**	5	5	**3**	**4**	4	4	4	**0**	**3**
20	**20**	20	**5**	5	5	**4**	4	4	4	**3**	3	3	3	3	**0**

هنا نقارن من الاسفل لغاية العنصر الرابع ويتوقف عن المقارنة

هنا ايضا نبدأ من الاسفل الى ان نصل للعنصر الثالث لان العنصر الثاني صار فيه اصغر .

هنا تحدث المقارنة من الاسفل الى الموقع الثاني وليس الاول لان الاول اصبح فيه اصغر قيمة ، لهذا نلاحظ ان الاعمدة قلت ب 1

في هذه المرحلة يبدأ من الموقع 6 و5

ولكن في كل عامود تحصل مقارنة بالعدد الاصغر ويبدل المواقع الى ان يصل الى الموقع 1 ويضع به الرقم الاصغر

Example: sort the list by 70, 101, 13, 0, 2 ascending:

The fourth Pass i=4 The our put	The third Pass i=3		The second Pass i=2			The first Pass i=1				The input
1	2	1	3	2	1	4	3	2	1	n
0	0	0	0	0	0	0	**70**	70	70	70
2	2	2	2	**70**	70	70	**0**	**101**	101	101
13	13	**70**	70	**2**	**101**	101	101	**0**	**13**	13
70	**70**	**13**	**101**	101	**2**	**13**	13	13	**0**	**0**
101	**101**	101	**13**	13	13	**2**	2	2	2	**2**

نلاحظ العامود الاول تكرر لان حصلت عملية مقارنة ولكن اصلا العدد اصغر من الي قبله فيكتب العامود بدون تغيير

Bubble Sort function in c++

```
void bubblesort(int ar[20],int n)
{
 int i,j;
 int item;
 for(i=0;i<n;i++)
  {
   for(j=n-1;j>i;--j)
    {
    if(ar[j]<ar[j-1])
     {
      item=ar[j];
      ar[j]=ar[j-1];
      ar[j-1]=item;
     }
    }
  }
}
```

هذا الجزء من ال loop عدد مراحله يساوي عدد عناصره وهو المتغير n ويبدأ i بالتناقص الي يمثل عدد ال pass

هنا عملية المقارنة من الاسفل اذا اصغر يبدل واذا لا لايدخل عملية التبديل ويعبر على العنصر الذي بعده

حيث العداد j يبدا من الاسفل لان J=n-1 ويتوقف الى ان يصل لل i

6.2 balanced two –way merge

This method is one of the types of external sorting algorithm, and the algorithm is summarized in the following steps:

1- Divide the original list (data) into two roughly equal lists, let it be a, b

2- We compare the first element of list with its counterpart the first element of list b and put them in order in list c.

3- We compare the second element of list a with its counterpart the second element of list b and put them in order in list d

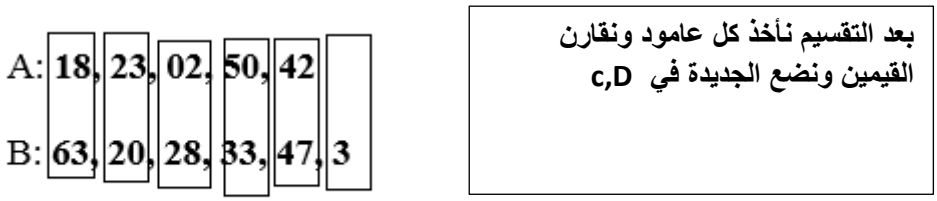
4- We repeat steps 2.3 and we get string of length 2 in each of the two lists c,d .

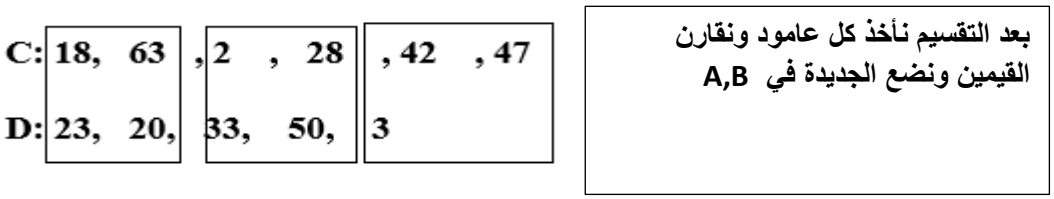
5- In the same way we combine the elements of lists c and d and put them in lists a,b and we will have their elements of length 4.

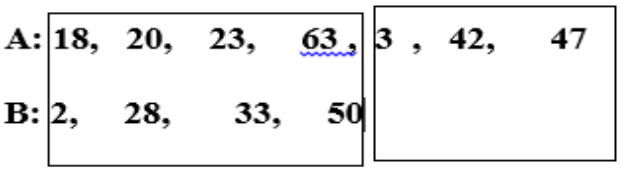
6- We repeat the method by merging the elements of lists a,b and putting them in lists c,d and their elements will be 8 .

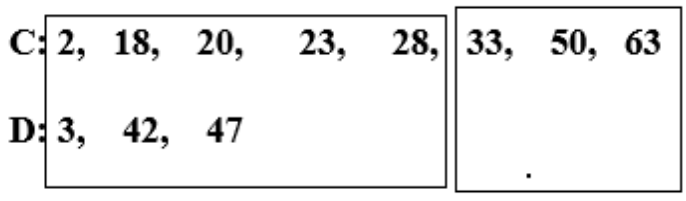
7- We will continue in this manner until the final ranked list is obtained.

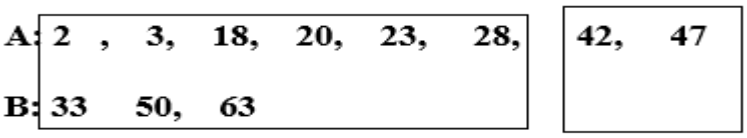
Example: sort in ascending order this list that used balanced two – way merge algorithm ((18, 23, 02, 50, 42, 63, 20, 28, 33, 47, 3))

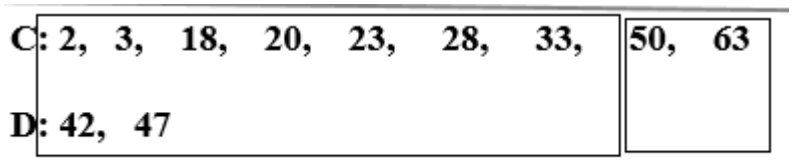


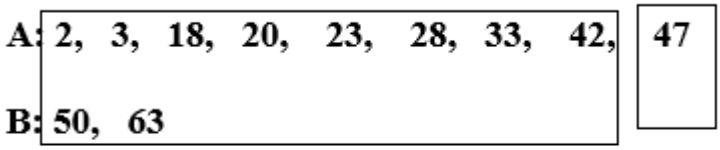


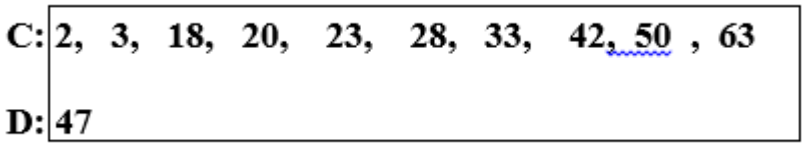












A: 2, 3, 18, 20, 23, 28, 33, 42, 50 , 47, 63

6.1 Search Algorithms

Before consider specific search techniques, let define some terms. A table or a file is group of elements, each of which is called a record. Associated with each record is a key, which is used to differential among different records.

For every file there is at least one set of keys (possible more) that is unique (that is, no two records have the same key). Such a key is called primary key. For example, if the file is stored as an array, the index within the array of an element is a unique external key for that element.

A searching algorithm is an algorithm that accepts an argument and tries to find a record whose key is a. The algorithm may return entire record or, more commonly; it may return a pointer to that record. It is possible that the search for a particular argument in a table is unsuccessful; that is, there is no record I the table with that argument as its key.

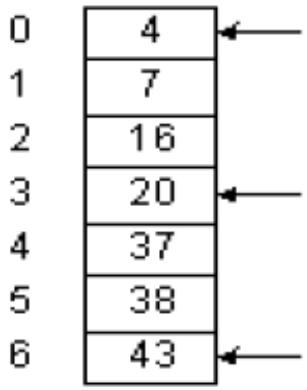
6.2Types of Search Algorithm:

1. Sequential search
2- Binary search
3- Binary tree search

6.3 Sequential search

It is the process of searching for a specific item in a list of items through (reviewing) all the list items from its beginning and in sequence until the required element is reached in its presence or reaching the end of the list when it is not present, so the average number of comparisons will be (n / 2) meaning that the time of implementation This algorithm will be O (n).

Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Linear Search

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

6.4 Binary search

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

The algorithm of this research assumes searching for a specific item in a sorted list according to a specific sequence and can be summarized in the following steps:

1- Locate the item, which is located approximately in the middle of the list.

2- Compare the item you want to search for x with the victory in the middle of the list.

3- If the required element x is equal to the element in the mean, the search process will end here.

4- If the required element x is less than the value of the element that is located in the middle, then the search will be limited to the part that includes the smaller values, and let the part be in the left section.

5- If the required element x is greater than the value of the element that is in the middle, then the search will be limited to the part that includes the largest values, and let the part that falls into the right section be.

6- In either case (5,4), that part is treated in the same way, i.e. choosing the midpoint and comparison until the required element is reached.

In this algorithm, each comparison will reduce the number of subsequent comparisons by half, and therefore the largest number of comparisons will reach (log2n) when searching in the list of the number of its components n, noting that the elements must be stored in an array because they will be in successive locations.

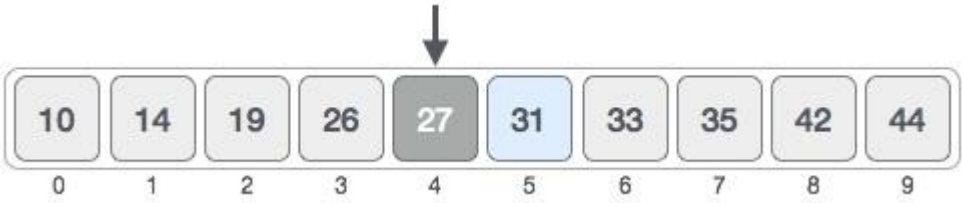
6.5 How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again.

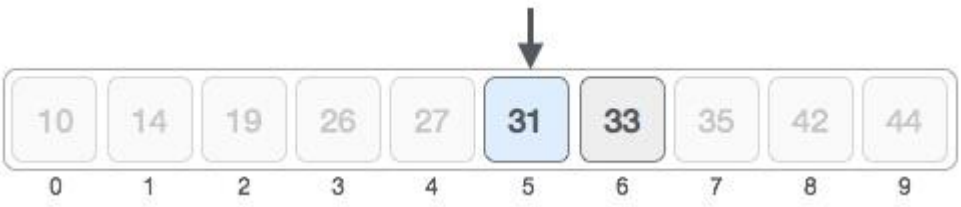
low = mid + 1

mid = low + (high - low) / 2

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

```
const n=20;
int a[n];
void binsearch(int a[n],int x,int n,int j)
{
 int upper,lower,mid;
 int found;
 lower=1;
 upper=n-1;
 found=0;
 while((lower<=upper)&&(!found))
   {
    mid=(lower+upper)/2;
    switch (compare(x,a[mid]))
     {
      case'>':lower=mid+1;break;
      case'<':upper=mid-1;break;
      case'=':
       {
          j=mid;
          found=1;
       }
      break;
     }
   }
}
```

```
char compare(int x,int y)
{
 if(x>y)
 return('>');
 else
  {
   if(x<y)
   return('<');
   else return('=');
  }
}
```

Search In Binary Search Tree

Search operations in binary search trees will be very similar to that. Let's say we want to search for the number **X**.

- We start at the root, and then we compare the value to be searched with the value of the root,
    - If it's equal we are done with the search if it's smaller we know that we need to go to the left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
    - Searching an element in the binary search tree is basically this traversal, at each step we go either left or right and at each step we discard one of the sub-trees.

```
bool search (Node* root,int data)
{
if (root==NULL) return false;
else if (root->data== data) return true ;
else if (data<= root->data) return (search(root->left,data));
else return search(root->right,data);
}
```