University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

Searching& Sorting Algorithms
خوارزميات البحث والترتيب

Lect. Alaa A. Hashim
م.علاء عبدالحسين هاشم

cs.uotechnology.edu.iq

1. Recursion الاستدعاء الذاتي

الاستدعاء الذاتي : هو قابلية البرنامج الفرعي () لاستدعاء نفسه , وهو طريقة رياضية واسلوب برمجي فعال يمكن استخدامه بدل استخدام اسلوب التكرار.

هناك العديد من الصيغ الرياضية يمكن التعبير عنها باستخدام الاستدعاء الذاتي . مثال:

دالة مضروب العدد التي تعرف رياضيا كالاتي :

Factorial of n → $n! = n * (n-1) * (n-2) * (n-3) \dots * 1$

$$n! = \begin{cases} 1 & , n = 0 \\ n * (n-1)! & , n > 0 \end{cases}$$

n=0 **الجزء الاساس**

n>0 **الجزء الاستقرائي**

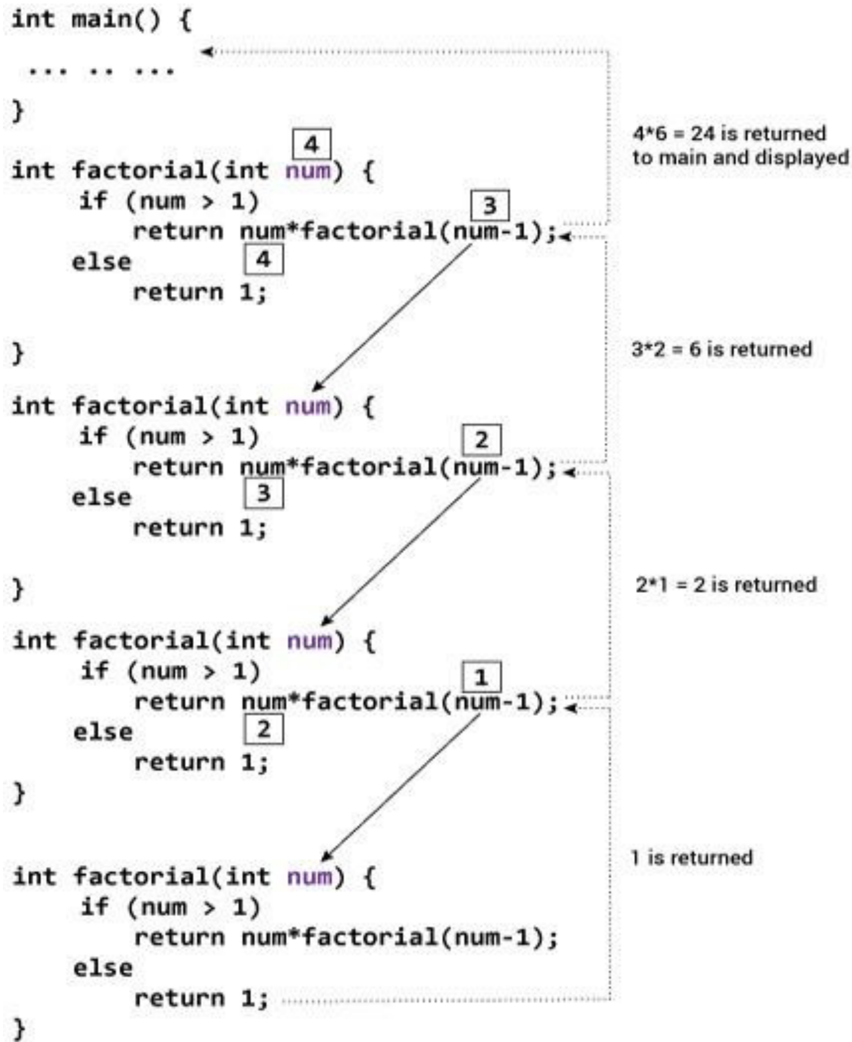
```
  // Factorial of n = 1*2*3*...*n

#include <iostream>
using namespace std;
int factorial (int n)
{
   If    (n > 1)
      return n * factorial (n - 1);
   else
      return 1;
}
int main()
{
   int n;
   cout<<"Enter a number to find factorial: ";
   cin >> n;
   cout << "Factorial of " << n <<" = " << factorial(n);
   return 0;
}
```

Enter a number to find factorial: 4

Factorial of 4 = 24

Explanation: How this example works?



Suppose the user entered 4, which is passed to the `factorial()` function.

1. In the first `factorial()` function, test expression inside if statement is true. The `return num*factorial(num-1);` statement is executed, which calls the second `factorial()` function and argument passed is `num-1`which is 3.

2. In the second `factorial()` function, test expression inside if statement is true. The `return num*factorial(num-1);` statement is executed, which calls the third `factorial()` function and argument passed is `num-1` which is 2.

3. In the third `factorial()` function, test expression inside if statement is true. The `return num*factorial(num-1);` statement is executed, which calls the fourth `factorial()` function and argument passed is `num-1` which is 1.

4. In the fourth `factorial()` function, test expression inside if statement is false. The `return 1;` statement is executed, which returns 1 to third `factorial()` function.

5. The third `factorial()` function returns 2 to the second `factorial()` function.

6. The second `factorial()` function returns 6 to the first `factorial()` function.

7. Finally, the first `factorial()` function returns 24 to the `main()` function, which is displayed on the screen.

Examples:

1. How to find the power of X^m?

$$X^m = \begin{cases} 1 & , \boldsymbol{m} = 0 \\ x * x^{m-1} & , \boldsymbol{m} > 0 \end{cases}$$

```
int power (int x,  int m)
{
   if (m == 0)
      return 1;

   else
      return   x * power(x, (m-1));
}
```

2. The Fibonacci Sequence is the series of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

في الرياضيات، متتالية فيبوناتشي أو أعداد فيبوناتشي نسبة إلى عالم الرياضيات الإيطالي ليوناردو فيبوناتشي, ،اي عدد فيها يكون مساويا لمجموع العددين السابقين عدا العدد الاول = 0 والعدد الثاني = 1

$$F_n = F_{n\text{-}1} + F_{n\text{-}2}$$

$$F_1 = 1 \text{ و } F_0 = 0$$

Example: the 8th term is

the 7th term plus the 6th term:

$F_8 = F_7 + F_6$

```
int fib(int n)
{
    if (n <= 1)
        return (n);
    return fib(n-1) + fib(n-2);
}
```

1.1 Introduction to Graph

Graph is a nonlinear data structure, it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

A graph is defined as follows:

Graph is a collection of vertices and arcs which connects vertices in the graph.

Graph is a collection of nodes and edges which connects nodes in the graph.

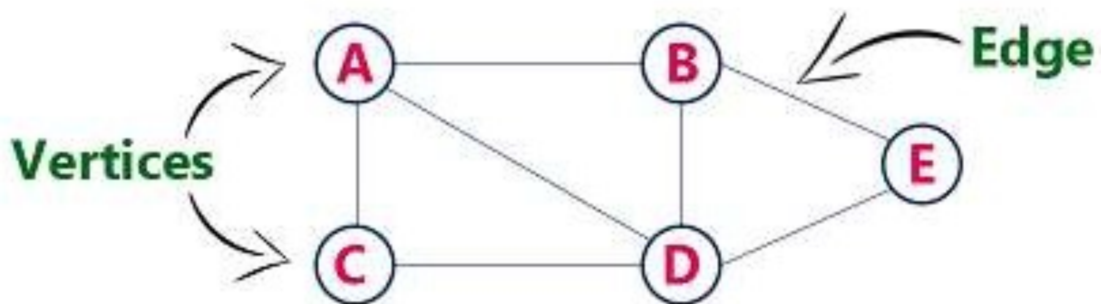
Generally, a graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example

The following is a graph with 5 vertices and 6 edges.

This graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(D, E)}.



1.1.1 Graph Terminology

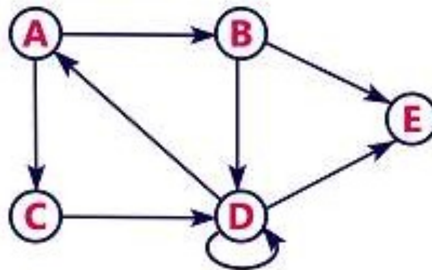
We use the following terms in graph data structure...

Vertex: A individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

Edge: An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (startingVertex, endingVertex). For example, in above graph, the link between vertices A and B is represented as (A,B). In above example graph, there are 7 edges (i.e., (A,B), (A,C), (A,D), (B,D), (B,E), (C,D), (D,E)).

Undirected Graph: A graph with only undirected edges is said to be undirected graph as in the above.

Directed Graph: A graph with only directed edges is said to be directed graph as in the figure below:



Connected graph: A graph G is called connected if every two of its vertices are connected.

Disconnected graph: A graph that is called not connected if some of its vertices is disconnected.

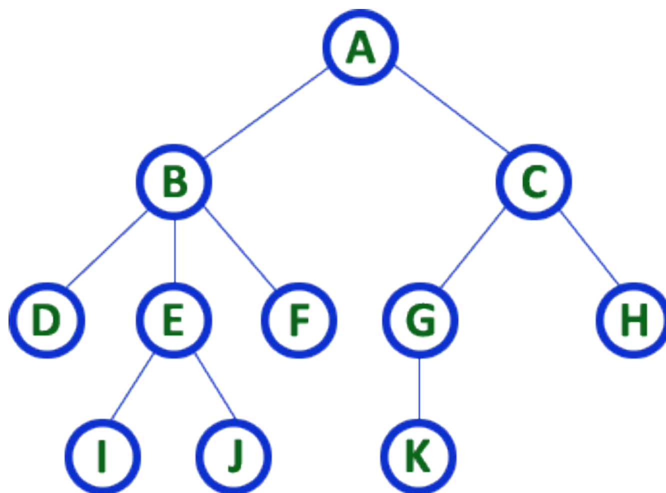
1.2 Trees

A tree data structure can be defined as follows...

A **connected acyclic graph** is called a tree. In other words, tree is a connected graph with no cycles .

In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N-1** number of links.

Example

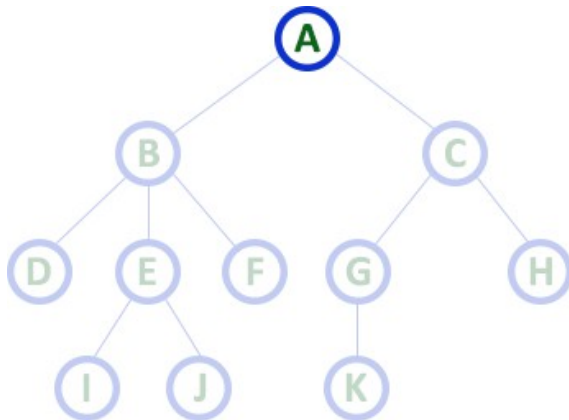


TREE with 11 nodes and 10 edges

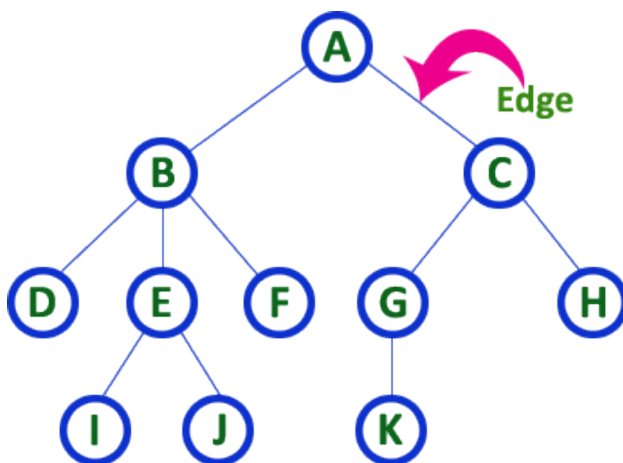
- **In any tree with 'N' nodes there will be maximum of 'N-1' edges**
- **In a tree every individual element is called as 'NODE'**

In a tree data structure, we use the following terminology...

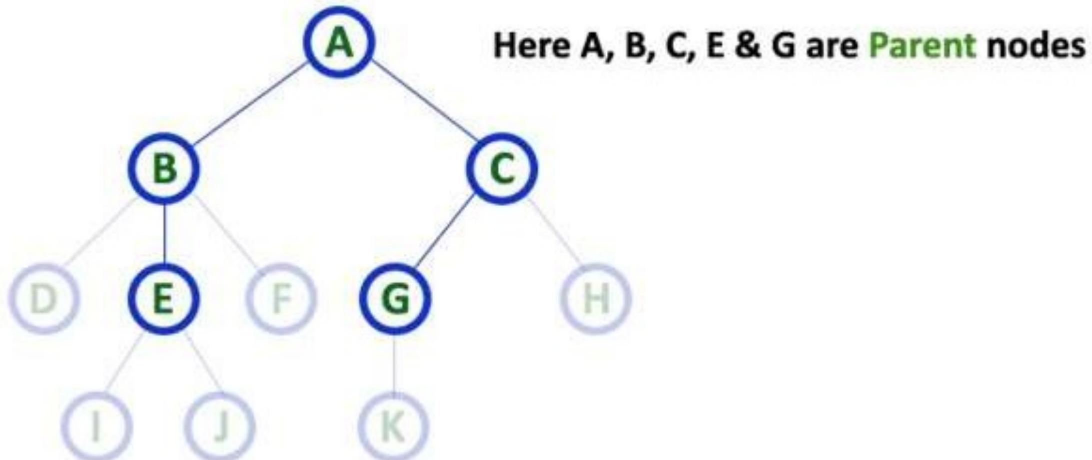
1. Root : In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree, there must be only one root node.



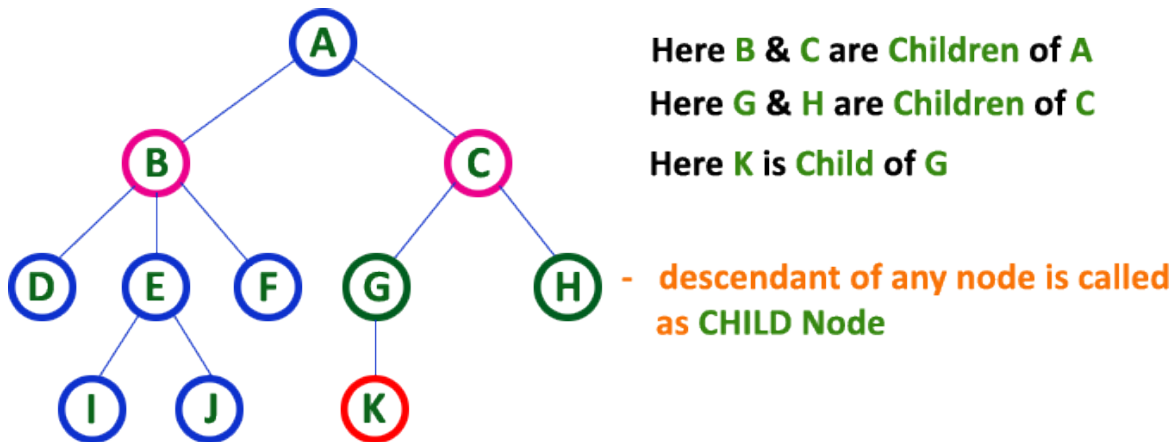
2. Edge: In a tree data structure, the connecting link between any two nodes is called as **EDGE.** In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



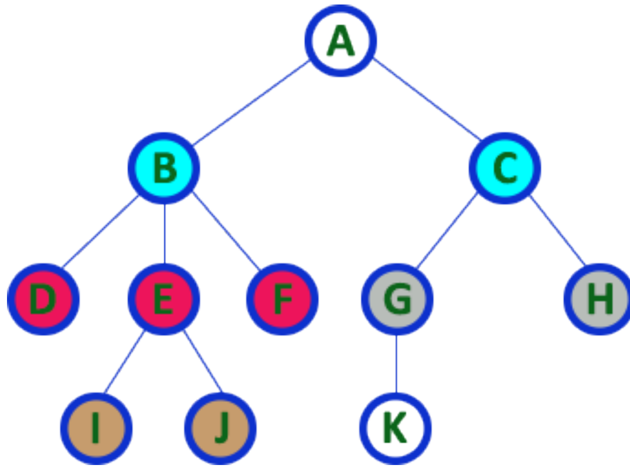
3. Parent: In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".



4. Child: In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



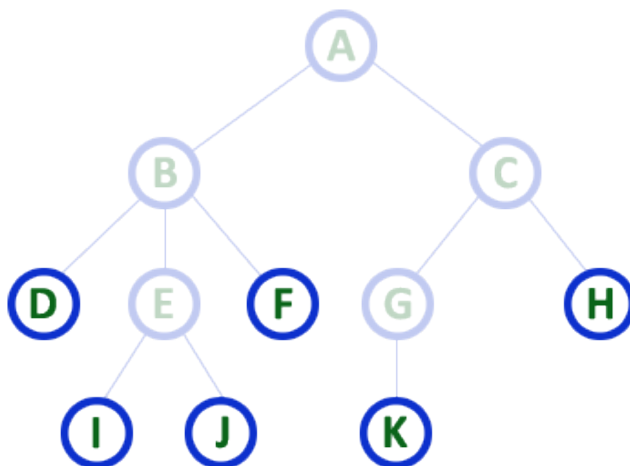
5. Siblings: In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

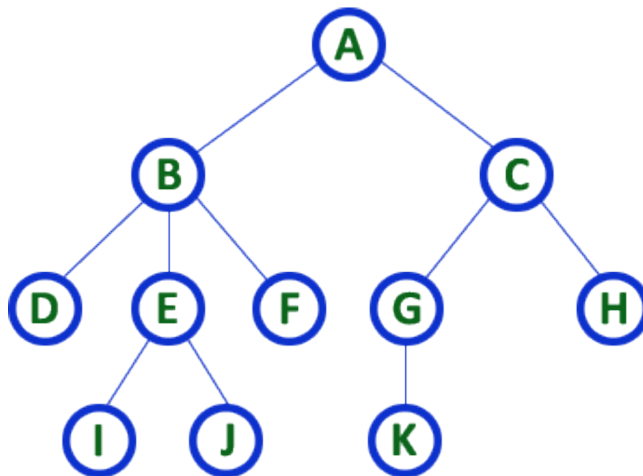
6. Leaf: In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.
leaf node is also called as '**Terminal**' node.



Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Degree: In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



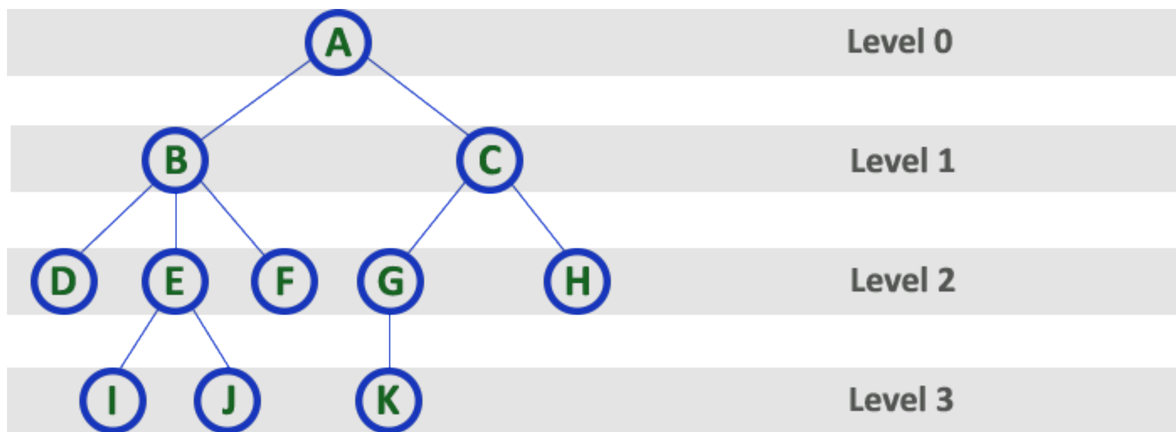
Here Degree of B is 3

Here Degree of A is 2

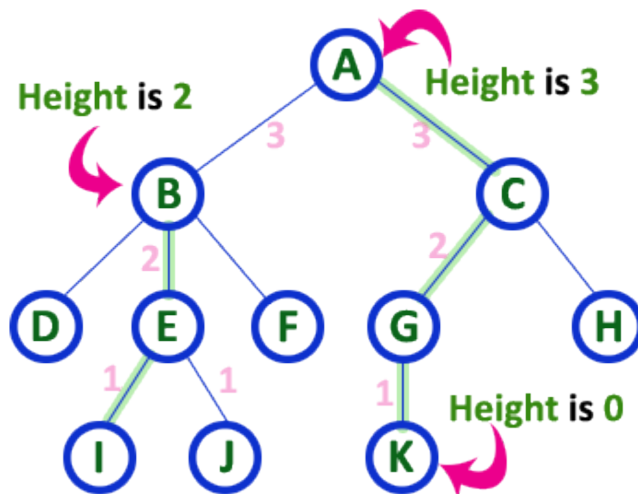
Here Degree of F is 0

- In any tree, 'Degree' a node is total number of children it has.

8. Level: In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



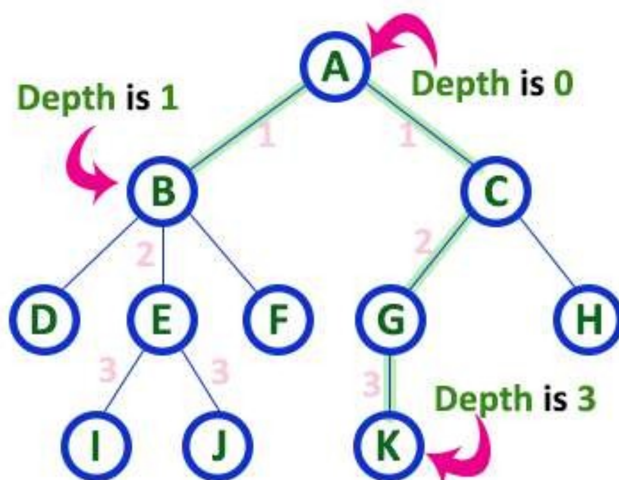
9. Height: In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**



Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

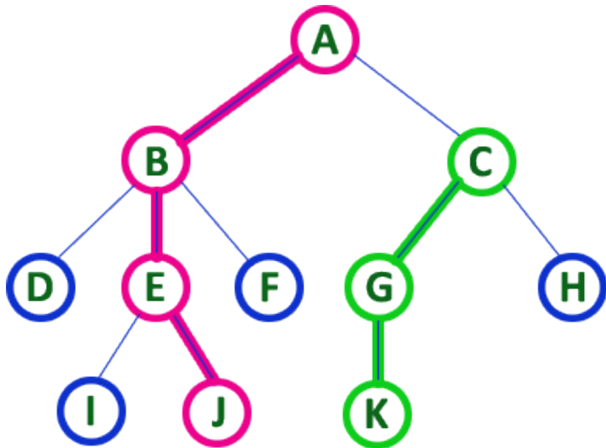
10. Depth: In a tree data structure, the total number of edges from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**



Here Depth of tree is 3

- In any tree, 'Depth of Node' is total number of Edges from root to that node.
- In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.

11. Path: In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **<u>Length of a Path</u>** <u>is</u> total number of nodes in that path. In below example **the path A - B - E - J has length4.**



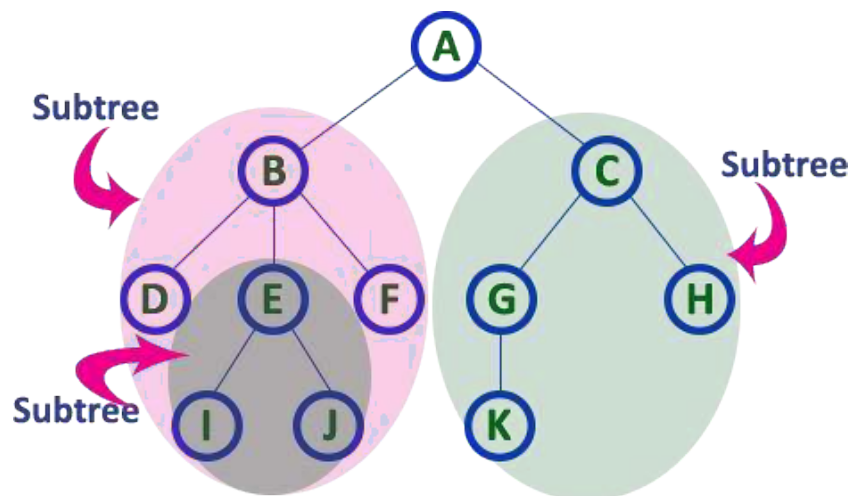
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

12. Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



2.1 Binary Tree

In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children**. One is known as left child and the other is known as right child.

Binary Tree: is a tree in which every node can have a maximum of two children.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example

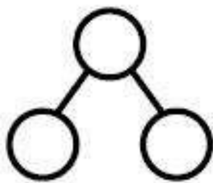


Figure 1:

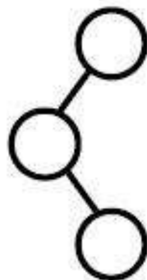


Figure 2:

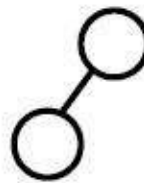


Figure 3:

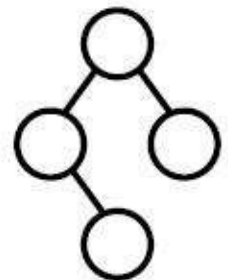
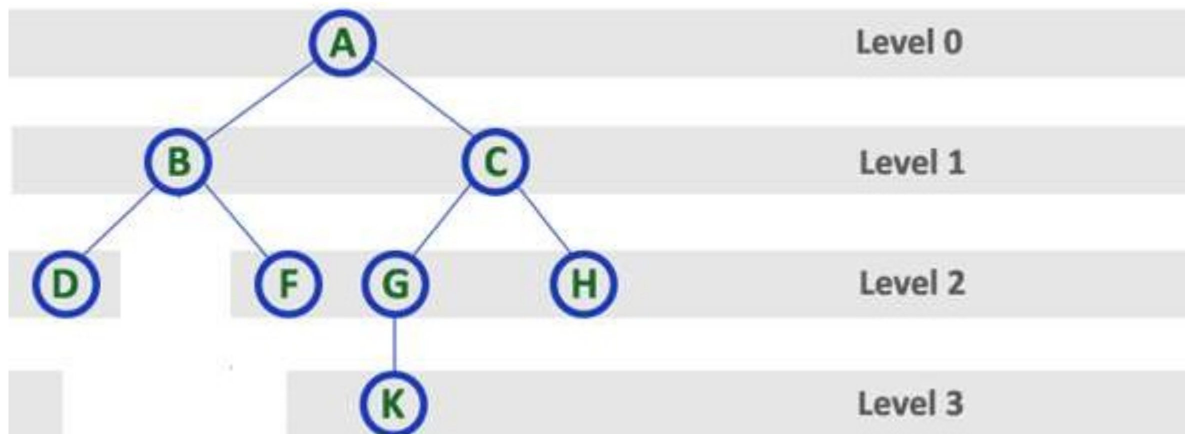


Figure 4:

Example



1. The Maximum number of nodes in any level L
   ($Max\ Nodes\ in\ any\ level = 2^L$)
2. The maximum number of the nodes in the binary tree ($2^{h+1} - 1$) where h is the height of the tree so in the example ($2^{3+1} - 1$=15) and the real number is 8.
3. The number of the leaves of the binary tree is equal to
   No. of leaves= (no. of nodes which have degree 2)+1
   In the above example 3+1=4.

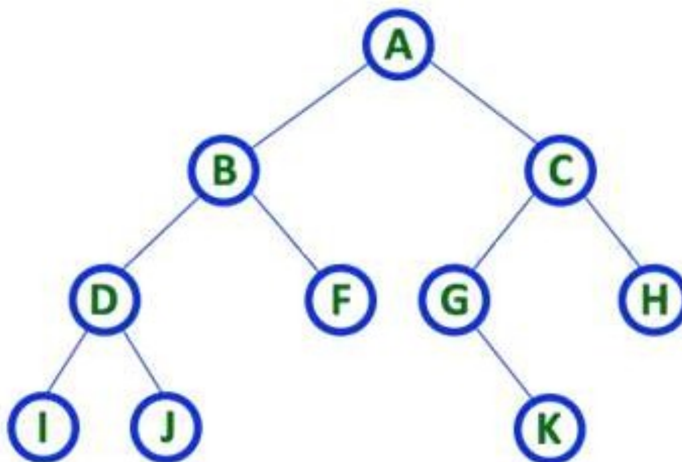
2.2 Binary Tree Representations:

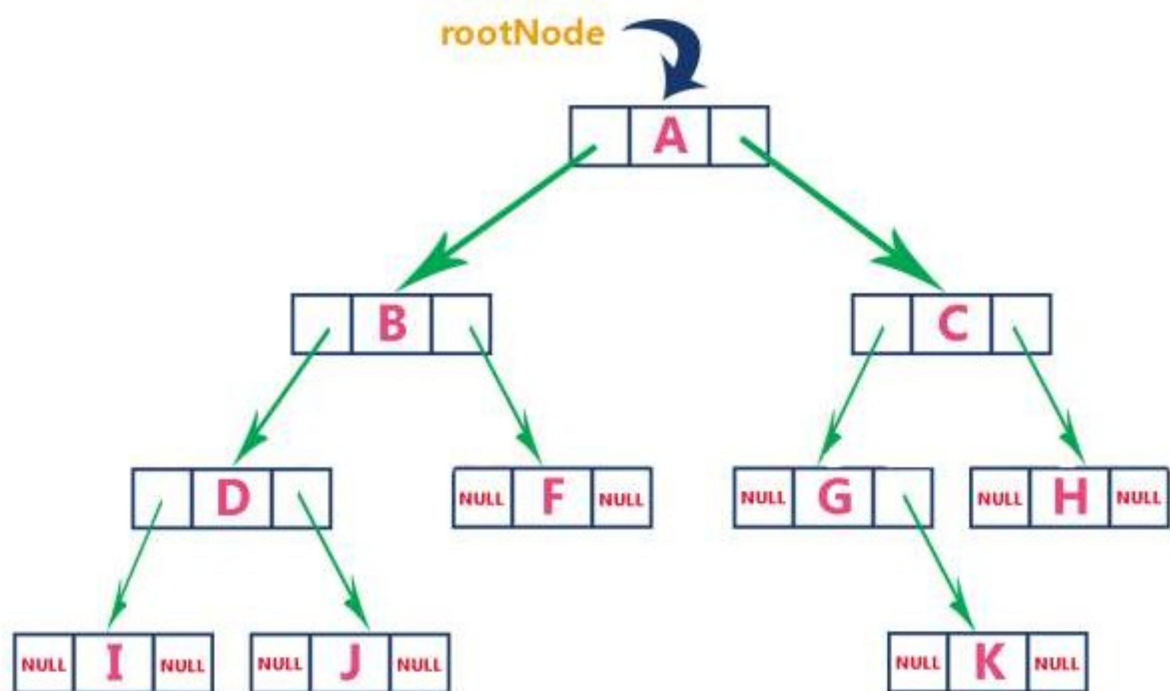
A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
In this linked list representation, a node has the following structure...

The below example of binary tree represented using Linked list representation is shown as follows...





2.3 Binary Tree Traversals

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

Displaying (or) visiting order of nodes in a binary tree is called as **Binary Tree Traversal.**

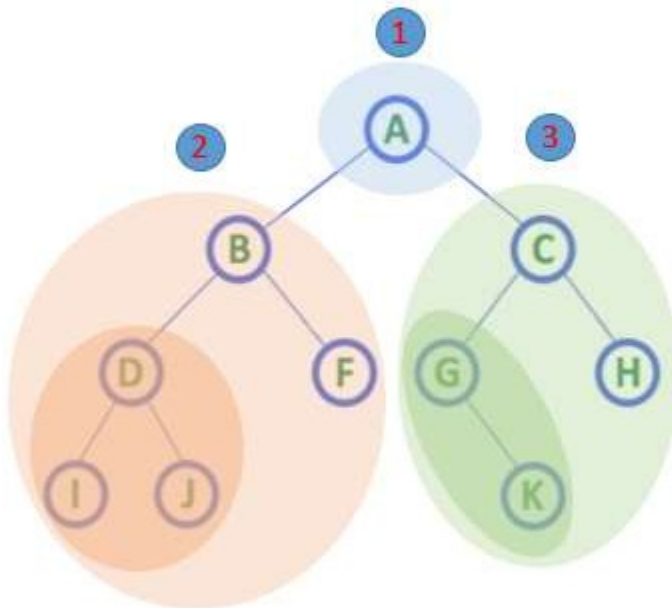
There are three types of binary tree traversals.

1. Pre - Order Traversal

2. In - Order Traversal

3. Post - Order Traversal

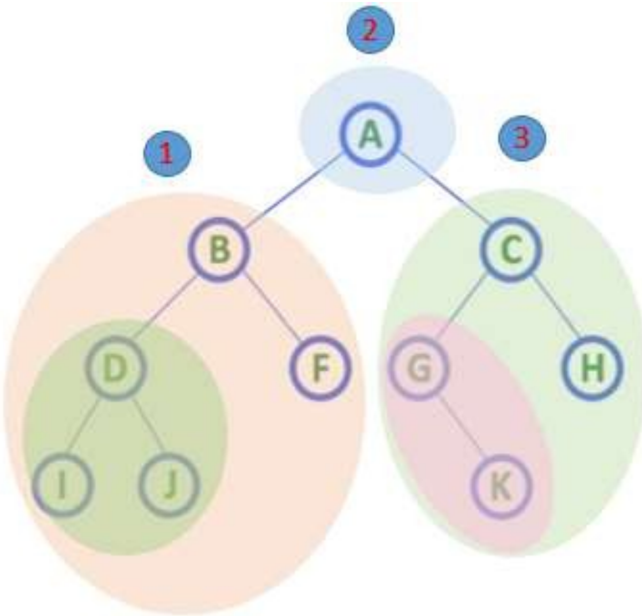
Consider the following binary tree...



1. Pre - Order Traversal (root – left Child – right Child)

In Pre-Order traversal, the root node is visited before left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree. In the above example of binary tree, first we visit root node 'A' then visit its left child 'B' which is a root for D and F. So we visit B's left child 'D' and again D is a root for I and J. So we visit D's left child 'I' which is the left most child. So next we go for visiting D's right child 'J'. With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child 'F'. With this we have completed root and left parts of node A. So we go for A's right child 'C' which is a root node for G and H. After visiting C, we go for its left child 'G' which is a root for node K. So next we visit left of G, but it does not have left child so we go for G's right child 'K'. With this we have completed node C's root and left parts. Next visit C's right child 'H' which is the right most child in the tree. So we stop the process.

Pre-Order Traversal for above example binary tree is
A - B - D - I - J - F - C - G - K – H



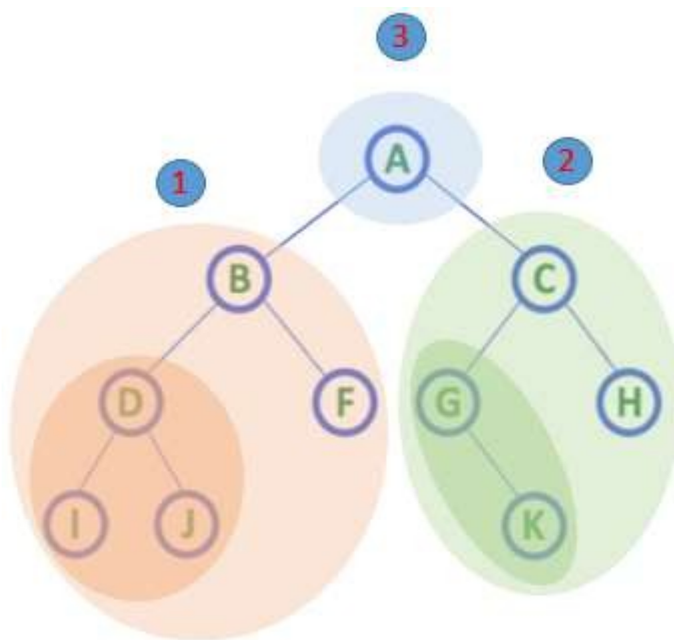
2. In - Order Traversal (left Child - root – right Child)

In In-Order traversal, the root node is visited between left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes I and J. So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for its root node 'D' and later we visit D's right child 'J'. With this we have completed the left part of node B. Then visit 'B' and next B's right child 'F' is visited. With this we have completed left part of node A. Then visit root node 'A'. With this we have completed left and root parts of node A. Then we go for right part of the node A. In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G. But G does not have left part so we visit 'G' and then visit G's right child K. With this we have completed the left part

of node C. Then visit root node 'C' and next visit C's right child 'H' which is the right most child in the tree so we stop the process.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C – H



3. Post - Order Traversal (left Child – right Child - root)

In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.

Post-Order Traversal for above example binary tree is
I - J - D - F - B - K - G - H - C – A

2.4 The functions of the Binary tree

As we mentioned before that, the structure of the tree consists of subtrees so, that means the part looks like the all, so here we can make use of the recursion to represent the functions of the tree.

1. In - Order Traversal

```
void inorder(nodeptr t)
{     if(t!=0)
     {
          inorder(t->l);
          cout<<t->info<<' ';
          inorder(t->r);
     }
}
```

2. Pre - Order Traversal

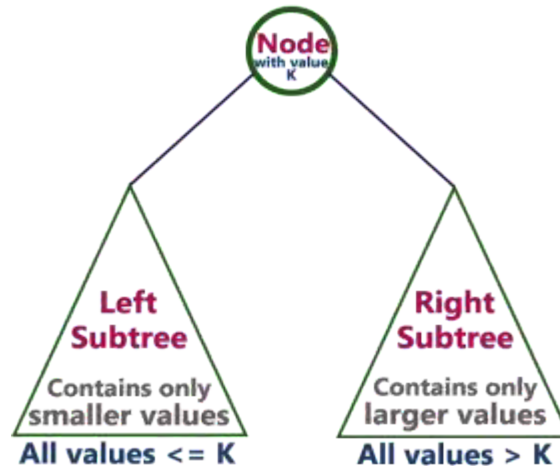
```
void preorder (nodeptr& t)
{     if(t!=0)
     {
          cout<<t->info<<' ';
          preorder(t->l);
          preorder(t->r);
     }
}
```

3. Post - Order Traversal

```
void postorder(nodeptr t)
{    if(t!=0)
    {
          postorder(t->l);
          postorder(t->r);
          cout<<t->info<<' ';
    }
}
```

3.1 Binary Search Tree

Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

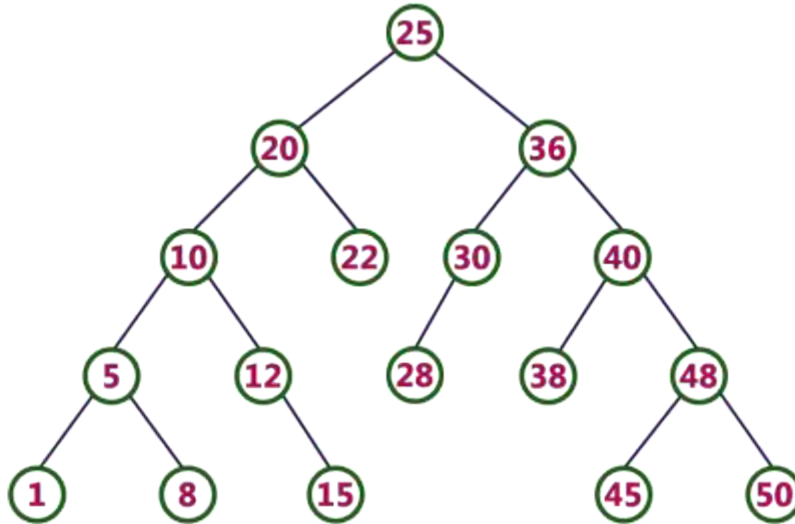


Note: Every Binary Search Tree is a binary tree but **NOT** all the Binary Trees are binary search trees.

Example:

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains

larger values

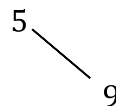


3.2 Insertion to BST

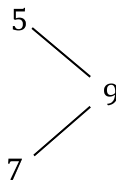
Example: Draw the BST for the following elements:

5,9, 7, 3,8,12, 6, 20

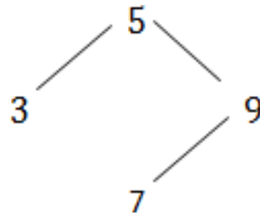
1- Take (5) as a root.
2- Take (9) as a right child because it is greater than the root.



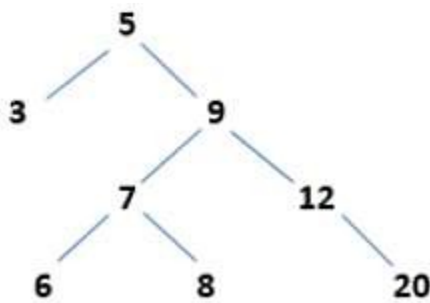
3- The next element (7) is greater than the root so we choose the right branch since it less than 9 so it will be the left child of 9.



4- Take 3 which it less than 5 put it in the left side.



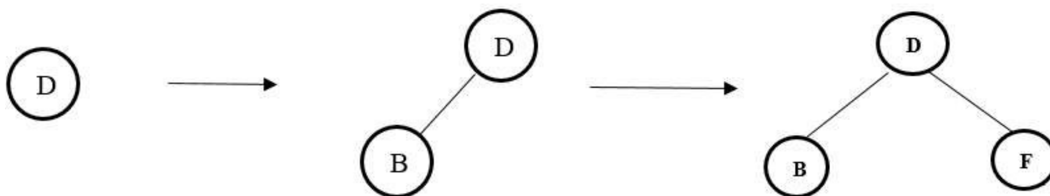
Continue in the same way take the new element and compare it with the tree started from the root , then we will get the final tree as below :

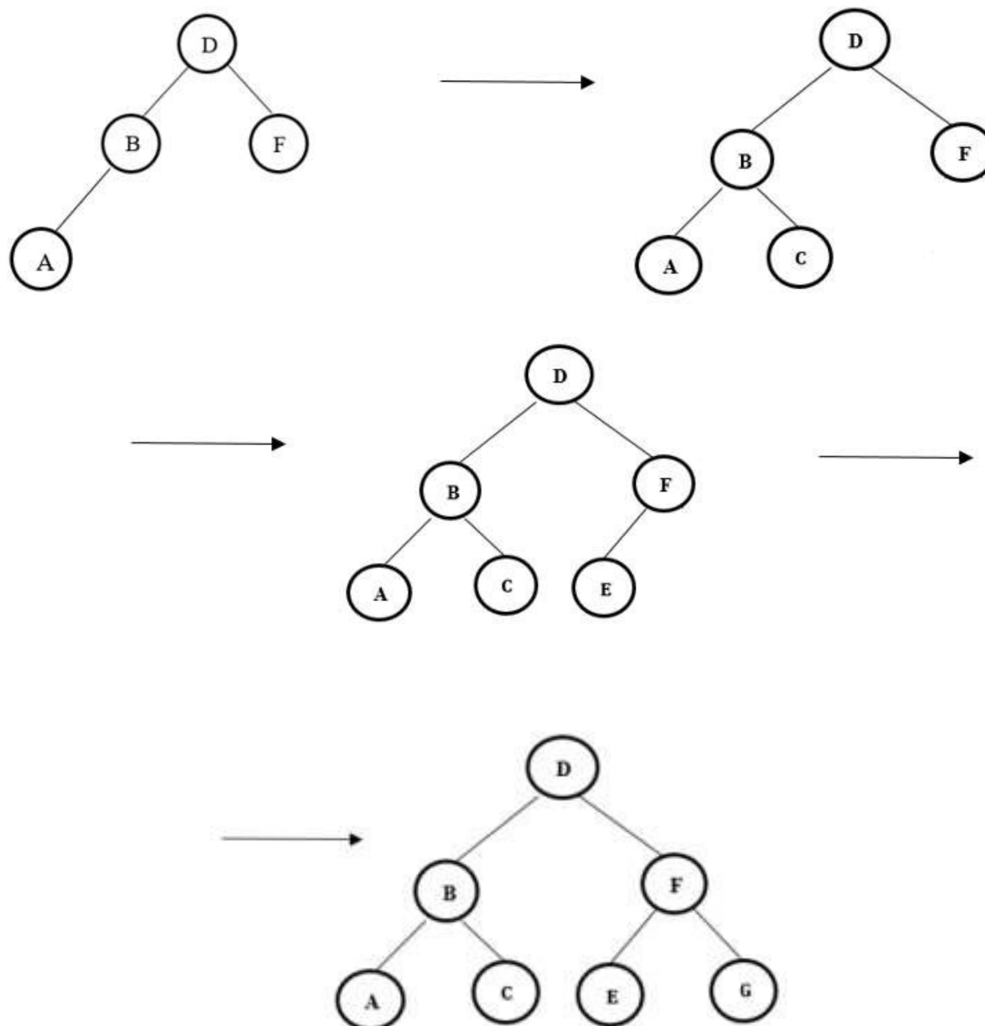


Example: draw the BST for those elements.

NOTE: the ascii code for A=65.

D, B, F, A, C, E, G





Home work: Draw the BST for those elements.

1. B, A, D, C, G, F, E
2. A, B, C, D, E, F, G

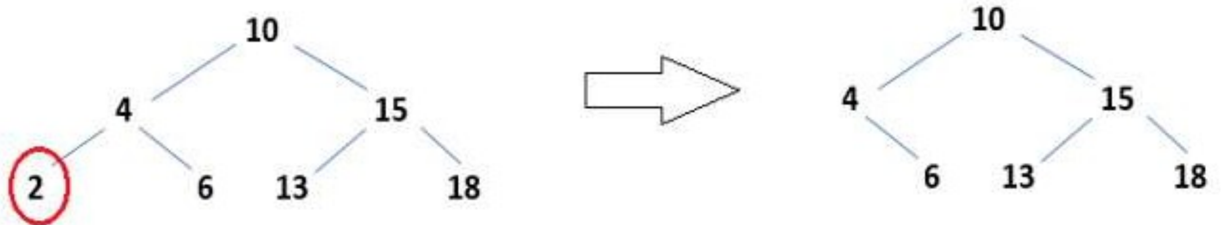
3.3 Deletion Operation in BST

Deleting a node from Binary search tree has following three cases...

Case 1: Deleting a Leaf node (A node with no children)
Case 2: Deleting a node with one child
Case 3: Deleting a node with two children

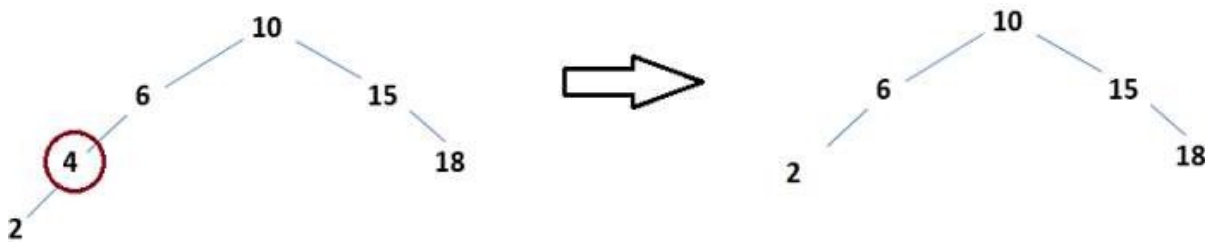
Case 1: Deleting a leaf node

Step 1: Find the node to be deleted.
Step 2: Delete the node and make the father node point to null.



Case 2: Deleting a node with one child:

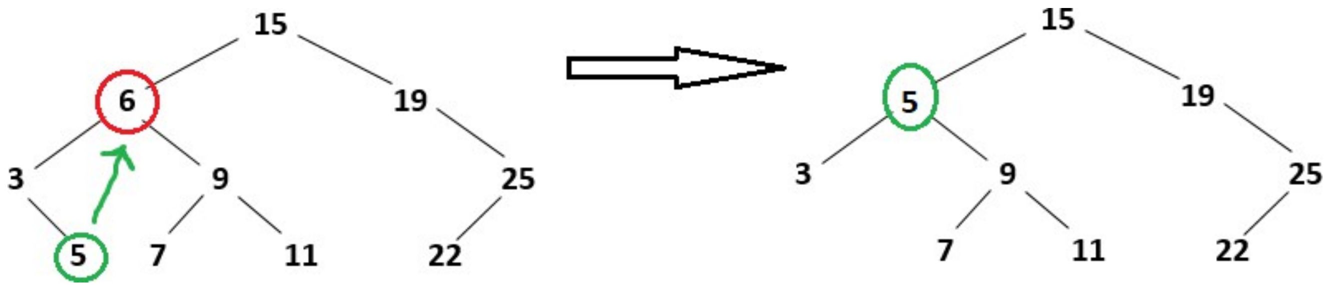
Step 1: Find the node to be deleted.
Step 2: Create a link between its parent and child node.



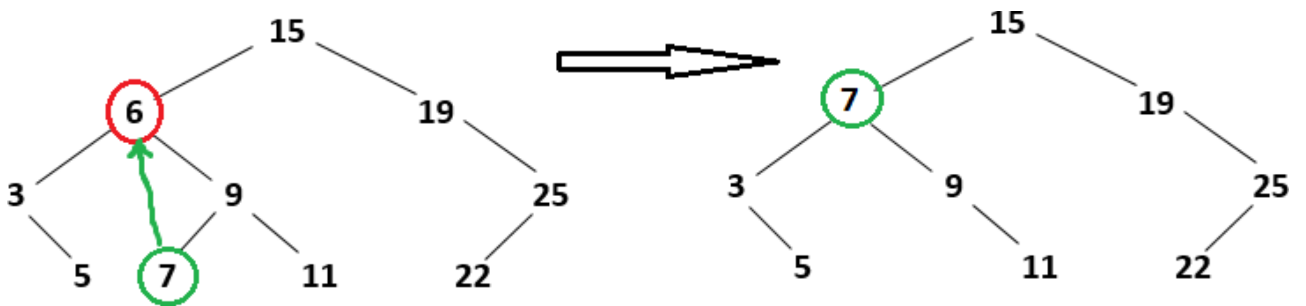
Case 3: Deleting a node with two children

Step 1: Find the node to be deleted.
Step 2: find the **max** node in its left subtree, **OR** the **min** node in its right subtree.

Example: Delete node 6



OR



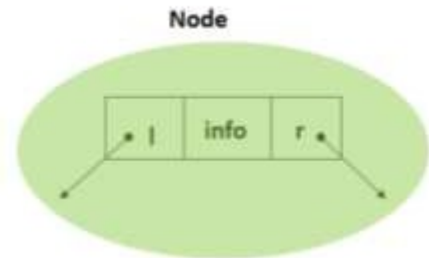
Exercise:

Delete the node 15.

Declaration of Tree by using linked list

نفس طريقة تعريف القائمة الموصولة الفرق اضافة حقل جديد
ليصبح حقل للفرع الايمن وحقل للفرع الايسرمن نوع مؤشر.

```
struct node
{
        int info;
        struct node *l,*r;
};
typedef struct node *nodeptr;
```



Creat the BST

```
void creat(nodeptr& t)
{
  char ch;
  if(t==0)
     {
       t=new node();
       cin>>t->info;
       t->l=0;
       t->r=0;
     }
  cout<<"Do you want to add from left ("<<t->info<<") (Y,N):";
  cin>>ch;
  if(ch=='y')
    creat(t->l);
  cout<<"Do you want to add from right ("<<t->info<<") (Y,N):";
  cin>>ch;
  if(ch=='y')
    creat(t->r);
}
```

عملية خلق اول عقدة في الشجرة (الجذر)

عملية اضافة عقدة في الجهة اليسرى

عملية اضافة عقدة في الجهة اليمنى

Printing the tree by using of the following:

1. In - Order Traversal

```
void inorder(nodeptr t)
{     if(t!=0)
      {
          inorder(t->l);
          cout<<t->info<<' ';
          inorder(t->r);
      }
}
```

2. Pre - Order Traversal

```
void preorder (nodeptr& t)
{     if(t!=0)
      {
          cout<<t->info<<' ';
          preorder(t->l);
          preorder(t->r);
      }
}
```

3. Post - Order Traversal

```
void postorder(nodeptr t)
{   if(t!=0)
    {
          postorder(t->l);
          postorder(t->r);
          cout<<t->info<<' ';
    }
}
```

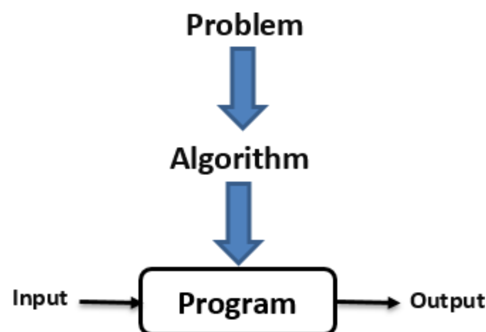
4. Sorting and Searching Algorithms خوارزميات البحث والترتيب

4.1 What is an algorithm?

In normal language, the **algorithm** is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Algorithms are used to convert the problem solution into step by step statements. These statements can be converted into computer programming instructions which form a program. This program is executed by a computer to produce a solution. Here, the program takes required data as **input**, **processes** data according to the program instructions and finally produces a **result** as shown in the following figure.



Specifications of Algorithms

Every algorithm must satisfy the following specifications...

1. **Input -** Every algorithm must take zero or more number of input values from external.
2. **Output -** Every algorithm must produce an output as result.
3. **Definiteness -** Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation).
4. **Finiteness -** For all different cases, the algorithm must produce result within a finite number of steps.
5. **Effectiveness -** Every instruction must be basic enough to be carried out and it also must be feasible.

ما هي الخوارزمية؟

الخوارزمية هي سلسلة من التعليمات التي لا غموض (كبرنامج) فيها تُستخدم لحل مشكلة ما ، والتي يمكن تنفيذها على جهاز الحاسوب.

يتم استخدام الخوارزميات لتحويل حل المشكلة إلى عبارات بيانية خطوة بخطوة. يمكن تحويل هذه العبارات إلى تعليمات برمجية التي تشكل برنامجًا . يتم تنفيذ هذا البرنامج بواسطة الحاسوب لإنتاج الحل . يأخذ البرنامج البيانات المطلوبة كمدخلات ، ويعالج البيانات وفقًا لإرشادات البرنامج وينتج النتيجة أخيرًاكما موضح في الشكل اعلاه .

4.2 Sorting Algorithms: خوارزميات الترتيب

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Why we need sorting?

1. To increase the efficiency of the search algorithm for an item
2. To simplify the processing of files
3. To solve the problem of similarity of data restriction

Sorting Algorithms

1. Selection sort الترتيب بالاختيار
2. Insertion sort الترتيب بالاضافة
3. Bubble sort ترتيب الفقاعة
4. Quick sort الترتيب السريع
5. Heap sort الترتيب الكومي
6. Merge sort ترتيب الدمج

We will explain three algorithms

1. Bubble sort الترتيب بالاضافة
2. Insertion sort الترتيب بالاختيار
3. Quick sort الترتيب السريع

1. Bubble sort الترتيب بالفقاعة

```
Void bubbleSort (int data [], int n)
{
    int temp;
  for (i = 0; i<(n-1); i++) {
    for (j = n-1; j< i; --j)
      if(data[j] < data [j-1])
      {
         temp = data [j];
         data [j] = data [j-1];
        data[j-1] = temp;
      }
}
```

2. Insertion sort algorithm الترتيب بالاختيار

```
void insertionSort(int data[], int n)
{
    int i, key, j;
    for (i = 1; i < n; i++)
    {
        key = data[i];
        for  (j = i; j>0 && key <data[j-1]; j--)
              data[j ] = data[j-1];

        data[j] = key;
    }
}
```

3. Quick sort algorithm الترتيب السريع

```
Void quickSort(int data [], int first, int last)
{
    While (last > first)
    {
       int lower=first; int upper=last;
       int bound= data[first];
       while (lower< upper)
        {
           While(data[upper]> bound)
             Upper--;
          data[lower]=data[upper];
          while ((lower< upper) && (data[lower]<= bound))
              lower++;
          data[upper]= data[lower];
        }
       data[lower]= bound;
        quickSort(data, first, lower-1);
        first=lower+1;
    }
}
```

4.3 Searching Algorithms: خوارزميات البحث

Search : is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

The search process may be positive when the required element exists and may be negative in case that the element is not found in the search list, the search process is effective when the search list is arranged according to a specific format.

Searching Algorithms

1. Sequential Search algorithm خوارزمية البحث التسلسلي

```
void  sequentialsearch(int arr[size], int k)
{
    int i, pos=-1;
    for (i = 0; i < (size -1); i++)
        if (arr[i] == k)
             pos=i;

        if (pos==-1)
          cout <<"the key is not found" ;
        else
           cout <<"the key is  found in the position " << pos;

}
```

2. Binary Search algorithm خوارزمية البحث الثنائي

```
int binarySearch(int data[]), int k , int lower , int upper)
 {
     int pos= -1; int mid;

     if (lower <= upper)
     {
        mid = ((lower + upper)/2);
        if k == (data[mid])
         {
            pos= mid;
            return pos;
         }

        else
         {
             if (k<data[mid])
               binarySearch(data,k,lower,mid-1);

             else
               binarySearch(data,k,mid+1,upper);

         }

      } return pos;

 }
```