KTU NOTES

The learning companion.

KTU STUDY MATERIALS | SYLLABUS | LIVE NOTIFICATIONS | SOLVED QUESTION PAPERS

DATA STRUCTURES –ITT 201 (S3 IT)

MODULE - 4

	Module 4: Trees and graphs	10 hrs
4.1	**Trees:** Basic terminologies, Binary Trees, Properties of binary trees, linear and linked representations, Complete and full Binary Tree.	2
4.2	**Binary Tree Traversals**: Preorder -In order and post order (Recursive, non-recursive)-problems	1
4.3	**Binary tree Applications:** Expression tree creation, heap trees (concepts), Binary search tree – creation, insertion and deletion and search operations	3
4.4	**Graph**: Terminologies, set representations, linked/adjacency list representation, Adjacency matrix linear representation **Graph traversal**: Breadth First Search (BFS), Depth First Search (DFS) - related problems.	2
4.5	**Graph Applications:** Shortest Path Problem-Dijkstras Algorithm	2

Tree Data Structure | Tree Terminologies
Tree Data Structure-

Tree data structure may be defined as-

Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.
OR

A tree is a connected graph without any circuits.

OR

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.
Properties-

The important properties of tree data structure are-

There is one and only one path between every pair of vertices in a tree.
A tree with n vertices has exactly (n-1) edges.
A graph is a tree if and only if it is minimally connected.
Any connected graph with n vertices and (n-1) edges is a tree.
Tree Terminology-
The important terms related to tree data structure are-

1. Root-

The first node from where the tree originates is called as a root node.
In any tree, there must be only one root node.
We can never have multiple root nodes in a tree data structure.

2. Edge-

The connecting link between any two nodes is called as an edge.
In a tree with n number of nodes, there are exactly (n-1) number of edges.

3. Parent-

The node which has a branch from it to any other node is called as a parent node.
In other words, the node which has one or more children is called as a parent node.
In a tree, a parent node can have any number of child nodes.

4. Child-

The node which is a descendant of some node is called as a child node.
All the nodes except root node are child nodes

5. Siblings-

Nodes which belong to the same parent are called as siblings.
In other words, nodes with the same parent are sibling nodes.

6. Degree-

Degree of a node is the total number of children of that node.
Degree of a tree is the highest degree of a node among all the nodes in the tree.

7. Internal Node-

The node which has at least one child is called as an internal node.
Internal nodes are also called as non-terminal nodes.
Every non-leaf node is an internal node.

8. Leaf Node-

The node which does not have any child is called as a leaf node.
Leaf nodes are also called as external nodes or terminal nodes.

9. Level-

In a tree, each step from top to bottom is called as level of a tree.
The level count starts with 0 and increments by 1 at each level or step.

10. Height-

Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node.
Height of a tree is the height of root node.
Height of all leaf nodes = 0

11. Depth-

Total number of edges from root node to a particular node is called as depth of that node.
Depth of a tree is the total number of edges from root node to a leaf node in the longest path.
Depth of the root node = 0
The terms “level” and “depth” are used interchangeably.

12. Subtree-

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

13. Forest-

A forest is a set of disjoint trees.

BINARY TREE:

A binary tree is a hierarchal data structure in which each node has at most two children. The child nodes are called the left child and the right child.

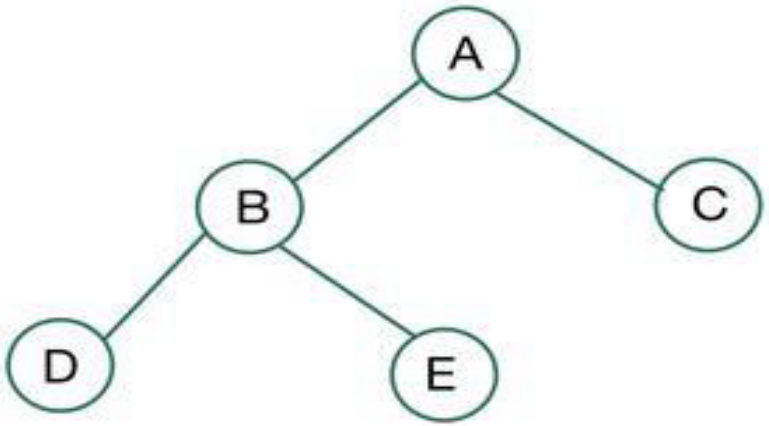
Properties:
some basic properties of a binary tree:

A binary tree can have a maximum of 2^{l} nodes at level l if the level of the root is zero.
When each node of a binary tree has one or two children, the number of leaf nodes (nodes with no children) is one more than the number of nodes that have two children.
There exists a maximum of (2^{h}-1) nodes in a binary tree if its height is h, and the height of a leaf node is one.
If there are L leaf nodes in a binary tree, then it has at least \lceil \log_2 L \rceil and at most L+1 levels.
A binary tree of n nodes has \log_{2}(n+1) minimum number of levels or minimum height.
The minimum and the maximum height of a binary tree having n nodes are \lceil \log_{2}n \rceil and n, respectively.
A binary tree of n nodes has (n+1) null

FULL BINARY TREE

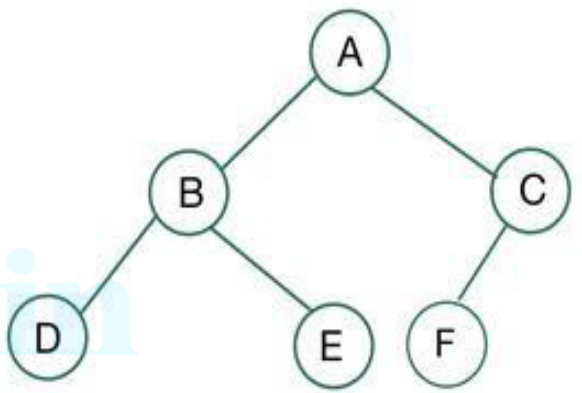
- If each node of binary tree has either two children or no child at all,is said to be a FULL BINARY TREE.

Full binary tree is also called a strictly binary tree.



COMPLETE BINARY TREE

- When all of the levels of a binary tree are entirely filled, except for the last level, which can contain 1 or 2 children nodes and is **filled from the left**, it is said to be a complete binary tree.
- Complete binary is also called as Perfect binary tree.
- In a complete binary tree,every internal node has exactly two children and all leaf nodes are at same level.
- For eg:-



At level 2,there must be 2^2=4 nodes and at level 3 there must be 2^3=8 nodes.

There are 2 points that you can recognize from here,

- *The leftmost side of the leaf node must always be filled first.*
- *It isn't necessary for the last leaf node to have a right sibling.*

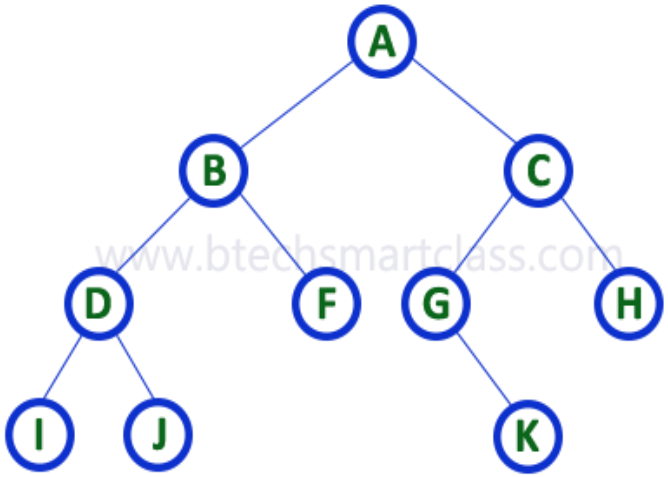
DIFFERENCE B/W COMPLETE AND FULL BINARY TREE

COMPLETE BINARY TREE	FULL BINARY TREE
• In a complete binary tree, a node in the last level can have only one child. • In a complete binary tree, the node should be filled from the left to right. • Complete binary trees are mainly used in heap-based data tree. • A complete binary tree is also called almost complete binary tree • A complete binary tree must have the entire leaves node in the exact same depth.	• In a full binary tree, a node cannot have just one child. • There is no order of filling nodes In a full binary tree. • Full binary tree has no application as such but is also called a proper binary tree. • A full binary tree also called proper binary tree or 2-tree • In full binary tree leaf level not necessarily have to be in the same depth.

BINARY TREE REPRESENTATION

A Binary tree data structure can be represented in two methods. Those methods are as follows...

1. Using Array
2. Using Linked list



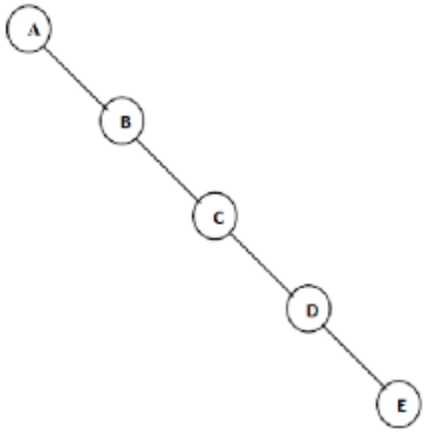
1. Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Consider the above example of a binary tree and it is represented as follows...

To represent a binary tree of depth **'n'** using array representation, we need one dimensional array with a maximum size of **2n + 1**.

❖**Consider the following tree:-**



	A	B		C		D		E	

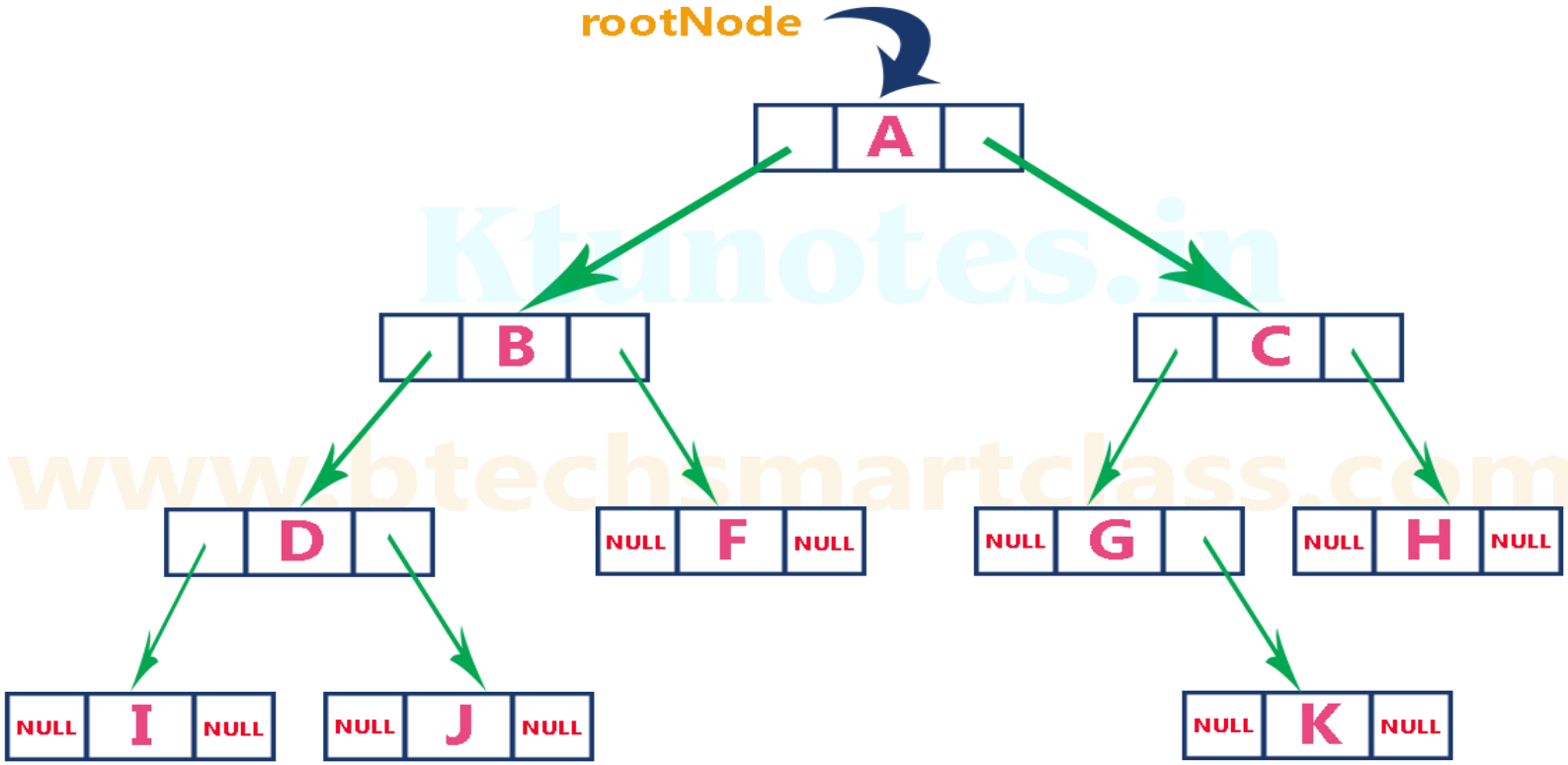
•Skewed representation of a tree can be implemented in an array.But this representation is inefficient.S o we use linked list representation of the tree.

2. Linked List Representation of Binary Tree

We use a 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 above example of the binary tree represented using Linked list representation is shown as follows...

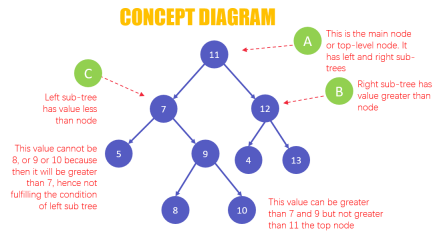


BINARY SEARCH TREE

A binary tree T is termed as binary search tree (or binary sorted tree)if each node N of T satisfies the following properties:

-the value at N is greater than every value in the left sub tree of N and is

-less than every value in the right sub tree of N



BST primarily offers the following three types of operations for your usage:

Search: searches the element from the binary tree

Insert: adds an element to the binary tree

Delete: delete the element from a binary tree

1:SEARCHING A KEY IN BST

-searching for a data in a binary search tree is much faster than in arrays or linked-lists

Algorithm searchBST(ROOT,ITEM): Ptr=ROOT,flag=FALSE

While(ptr!=NULL)and(flag=FALSE)do

Case:ITEM<ptr->DATA

ptr=ptr->L CHILD

Case:ptr->DATA=ITEM

flag=TRUE

Case:ITEM>ptr->DATA

ptr=ptr->R CHILD

Endcase

EndWhile

if(flag=TRUE)then

print"ITEM has found at the node",ptr

else

print"ITEM does not exist:search is unsuccessful "

Endif

stop

2:insertion into BST

-it is very simple

-infact one step more than the searching operation

-to search a node with data,say ITEM into a tree,the tree is to be searched starting from the root node.

-if ITEM is found do nothing, otherwise ITEM to be inserted at the dead end where search halts.

Algorithm insert BST(ROOT,ITEM)

ptr=ROOT,flag=FALSE

While(ptr=NULL)and(flag=FALSE)do

```
    Case:ITEM<ptr->DATA
            ptr1=ptr
            ptr=ptr->L CHILD
    Case:ITEM>ptr->DATA
            ptr1=ptr
            ptr=ptr-> R CHILD
    Case:ptr->DATA=ITEM
            flag=TRUE
            print"ITEM already exists"
            Exit
    Endcase
EndWhile
if(ptr=NULL)then
  new=GetNode(NODE)
  new->DATA=ITEM
  new->L CHILD=NULL
  new->R CHILD=NULL
  if(ptr1->DATA<ITEM)
      ptr1->R CHILD=new
  else
      ptr1->L CHILD=new
  Endif
Endif
Stop
```


3:deletion from BST

For deleting a Node N ,3 cases are to be considered

1.N is a leaf node

2.N has exactly 1 child

3.N has 2 children

Algorithm DeleteBST(ROOT, ITEM)

```
1.  ptr = ROOT, flag = FALSE
2.  While (ptr ' NULL) and (flag = FALSE) do          // Step to find the location of the node
3.      Case: ITEM < ptr→DATA
4.          parent = ptr
5.          ptr = ptr→LCHILD
6.      Case: ITEM > ptr→DATA
7.          parent = ptr
8.          ptr = ptr→RCHILD
9.      Case: ptr→DATA = ITEM
10.         flag = TRUE
11.     EndCase
12. EndWhile
13. If (flag = FALSE) then                           // When the node does not exist
14.     Print "ITEM does not exist: No deletion"
15.     Exit                                         // Quit the execution
16. EndIf
    /* DECIDE THE CASE OF DELETION */
17. If (ptr→LCHILD = NULL) and (ptr→RCHILD = NULL) then   // Node has no child
18.     case = 1
19. Else
20.     If (ptr→LCHILD ' NULL) and (ptr→RCHILD ' NULL) then
                                                     // Node contains both the child
21.         case = 3
22.     Else                                         // Node contains only one child
23.         case = 2
24.     EndIf
25. EndIf
```

```
    /* DELETION: CASE 1 */
26. If (case = 1) then
27.     If (parent→LCHILD = ptr) then                // If the node is a left child
28.         parent→LCHILD = NULL                     // Set pointer of its parent
29.     Else
30.         parent→RCHILD = NULL
31.     EndIf
32.     ReturnNode(ptr)                              // Return deleted node to the memory bank
33. EndIf
```

```
    /* DELETION: CASE 2 */
34. If (case = 2) then                                          // When the node contains only one child
35.     If (parent→LCHILD = ptr) then                           // If the node is a left child
36.         If (ptr→LCHILD = NULL) then
37.             parent→LCHILD = ptr→RCHILD
38.         Else
39.             parent→LCHILD = ptr→LCHILD
40.         EndIf
41.     Else
42.         If (parent→RCHILD = ptr) then
43.             If (ptr→LCHILD = NULL) then
44.                 parent→RCHILD = ptr→RCHILD
45.             Else
46.                 parent→RCHILD = ptr→LCHILD
47.             EndIf
48.         EndIf
49.     EndIf
50.     ReturmnNode(ptr)                                        // Return deleted node to the memory bank
51. EndIf
```

```
    /* DELETION: CASE 3 */
52. If (case = 3)                                               // When contains both child
53.     ptr1 = SUCC(ptr)                                        // Find the in order successor of the
        node
54.     item1 = ptr1→DATA
55.     Delete_BST (item1)                                      // Delete the inorder successor
56.     ptr→DATA = item1                                        // Replace the data with the data of in order successor
57. EndIf
```

DIFFERENT TYPES OF TRAVERSALS OF A BINARY TREE (using recursion)

- **Inorder traversal**
- **Preorder traversal**
- **Postorder traversal**

INORDER TRAVERSAL(left-root-right)

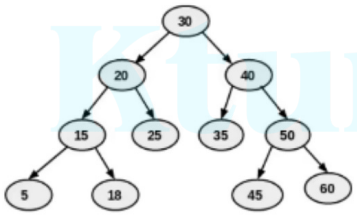
1. **Traverse the left sub-tree of the root node R in inorder.**
2. **Visit the root node R.**
3. **Traverse the right sub-tree of the root node R in inorder.**

Algorithm

Steps:

1. **Ptr=ROOT**
2. **if(ptr!=NULL)then**
3. **Inorder (ptr->LC)**
4. **Visit(ptr)**
5. **Inorder(ptr->RC)**
6. **Endif**
7. **Stop**

Example of inorder traversal



- **we start recursive call from 30(root) then move to 20 (20 also have sub tree so apply in order on it),15 and 5.**
- **5 have no child .so print 5 then move to it's parent node which is 15 print and then move to 15's right node which is 18.**
- **18 have no child print 18 and move to 20 .print 20 then move it right node which is 25 .25 have no subtree so print 25.**
- **print root node 30 .**
- **now recursively traverse to right subtree of root node . so move to 40. 40 have subtree so traverse to left subtree of 40.**
- **left subtree of 40 have only one node which is 35. 35 had no further subtree so print 35. move to 40 and print 40.**
- **traverse to right subtree of 40. so move to 50 now have subtree so traverse to left subtree of 50 .move to 45 , 45 have no further subtree so print 45.**
- **move to 50 and print 50. now traverse to right subtree of 50 hence move to 60 and print 60.**
- **our final output is {5 , 15 , 18 , 20 , 25 , 30 , 35 , 40 , 45 , 50 , 60}**

APPLICATION OF INORDER TRAVERSAL

- **In-order traversal is used to retrives data of binary search tree in sorted order.**

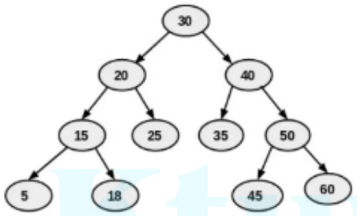
PREORDER TRAVERSAL (root-left-right)

1. **Visit the root node R.**
2. **Traverse the left sub-tree of the root node R in inorder**
3. **Traverse the right sub-tree of the root node R in inorder**

Algorithm
Steps:

1. **Ptr=ROOT**
2. **if(ptr!=NULL)then**
3. **Visit(ptr)**
4. **preorder(ptr->LC)**
5. **Preorder (ptr->RC)**
6. **Endif**
7. **Stop**

Example of preorder traversal



- **Start with root node 30 .print 30 and recursively traverse the left subtree.**
- **next node is 20. now 20 have subtree so print 20 and traverse to left subtree of 20 .**
- **next node is 15 and 15 have subtree so print 15 and traverse to left subtree of 15.**
- **5 is next node and 5 have no subtree so print 5 and traverse to right subtree of 15.**
- **next node is 18 and 18 have no child so print 18 and traverse to right subtree of 20.**
- **25 is right subtree of 20 .25 have no child so print 25 and start traverse to right subtree of 30.**
- **next node is 40. node 40 have subtree so print 40 and then traverse to left subtree of 40.**
- **next node is 35. 35 have no subtree so print 35 and then traverse to right subtree of 40.**
- **next node is 50. 50 have subtree so print 50 and traverse to left subtree of 50.**
- **next node is 45. 45 have no subtree so print 45 and then print 60(right subtree) of 50.**
- **our final output is {30 , 20 , 15 , 5 , 18 , 25 , 40 , 35 , 50 , 45 , 60}**

APPLICATION OF PRE-ORDER TRAVERSAL

- ***Preorder traversal is used to create a copy of the tree.***

- ***Preorder traversal is also used to get prefix expression of an expression tree.***

POSTORDER TRAVERSAL (left-right-root)

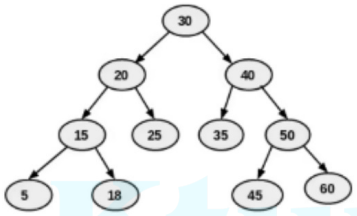
1. **Traverse the left sub-tree of the root node R in inorder**
2. **Traverse the right sub-tree of the root node R in inorder**
3. **Visit the root node R.**

Algorithm

Steps:

1. **Ptr=ROOT**
2. **if(ptr!=NULL)then**
3. **postorder(ptr->LC)**
4. **Postorder (ptr->RC)**
5. **Visit(ptr)**
6. **Endif**
7. **Stop**

Example of postorder traversal



- **We start from 30, and following Post-order traversal, we first visit the left subtree 20. 20 is also traversed post-order.**
- **15 is left subtree of 20 .15 is also traversed post order.**
- **5 is left subtree of 15. 5 have no subtree so print 5 and traverse to right subtree of 15 .**
- **18 is right subtree of 15. 18 have no subtree so print 18 and then print 15. post order traversal for 15 is finished.**
- **next move to right subtree of 20.**
- **25 is right subtree of 20. 25 have no subtree so print 25 and then print 20. post order traversal for 20 is finished.**
- **next visit the right subtree of 30 which is 40 .40 is also traversed post-order(40 have subtree).**
- **35 is left subtree of 40. 35 have no more subtree so print 35 and traverse to right subtree of 40.**
- **50 is right subtree of 40. 50 should also traversed post order.**
- **45 is left subtree of 50. 45 have no more subtree so print 45 and then print 60 which is right subtree of 50.**
- **next print 50 . post order traversal for 50 is finished.**
- **now print 40 ,and post order traversal for 40 is finished.**
- **print 30. post order traversal for 30 is finished.**
- **our final output is {5 , 18 , 15 , 25 , 20 , 35 , 45 , 60 , 50 , 40 , 30}**

APPLICATION OF POSTORDER TRAVERSAL

- ***Postorder traversal is used to delete the tree.***
- ***Postorder traversal is also used to get the postfix expression of an expression tree.***

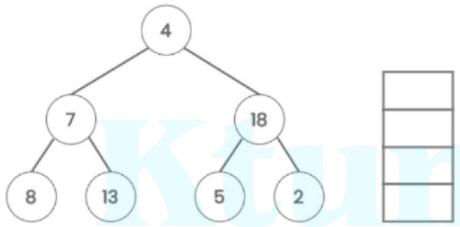
NON RECURSIVE BINARY TREE TRAVERSE

Using Stack is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack.

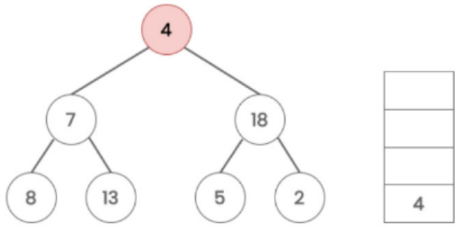
PREORDER TRAVERSAL

- ***Start with root node and push onto stack.***
- ***Repeat while the stack is not empty***
- ***POP the top element (PTR) from the stack and process the node.***
- ***PUSH the right child of PTR onto to stack.***
- ***PUSH the left child of PTR onto to stack.***

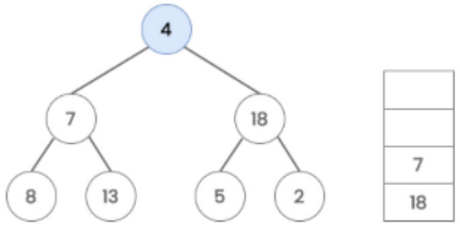
Consider the following tree.



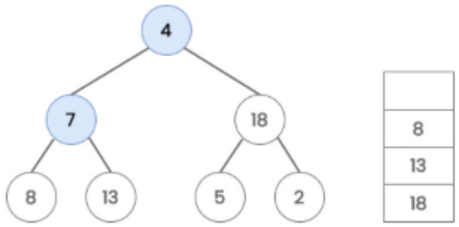
Start with node 4 and push it onto the stack.



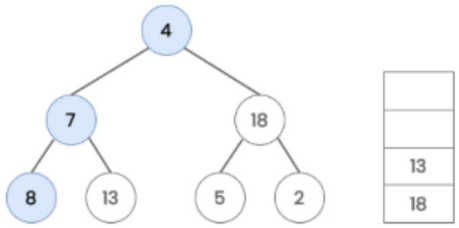
Since the stack is not empty, POP 4 from the stack, process it and PUSH its left(7) and right(18) child onto the stack.



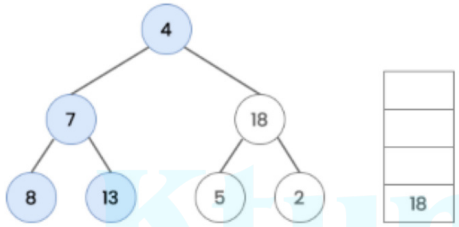
Repeat the same process since the stack is not empty. POP 7 from the stack, process it and PUSH its left(8) and right (13) child onto the stack.



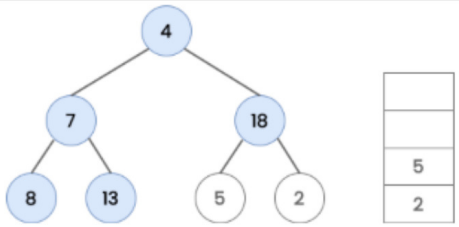
Again, POP 8 from the stack and process it. Since it has no right or left child we don't have to PUSH anything to the stack.



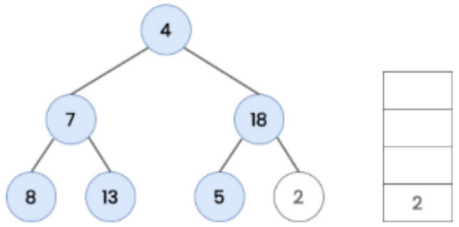
Now POP 13 from the stack and process it. We don't have to PUSH anything to the stack because it also doesn't have any subtrees.

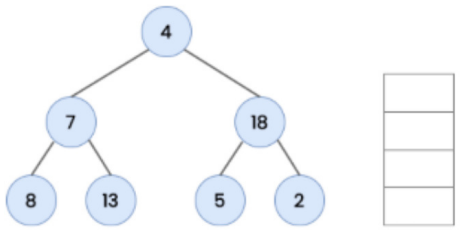


POP 18 from the stack, process it and PUSH its left(5) and right(2) child to the stack



Similarly POP 5 and 2 from the stack one after another. Since both these nodes don't have any child, we don't have to PUSH anything onto the stack.



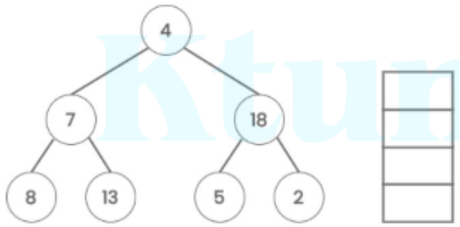


The nodes are processed in the order [4, 7, 8, 3, 18, 5, 2]. This is the required preorder traversal of the given tree.

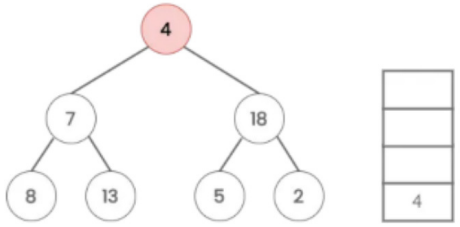
INORDER TRAVERSAL

1. **Start from the root, call it PTR.**
2. **Push PTR onto stack if PTR is not NULL.**
3. **Move to left of PTR and repeat step 2.**
4. **If PTR is NULL and stack is not empty, then Pop element from stack and set as PTR.**
5. **Process PTR and move to right of PTR , go to step**

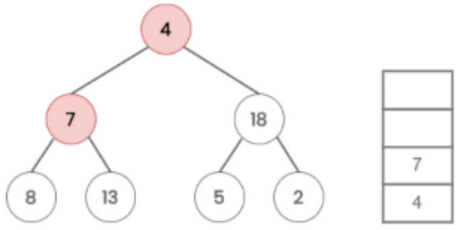
Consider the following tree



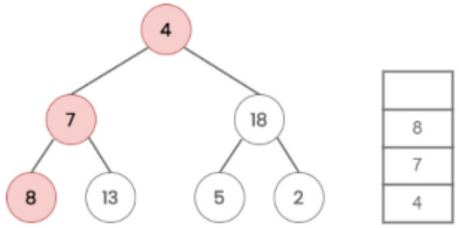
Start with node 4 and call it PTR. Since PTR is not NULL, PUSH it onto the stack.



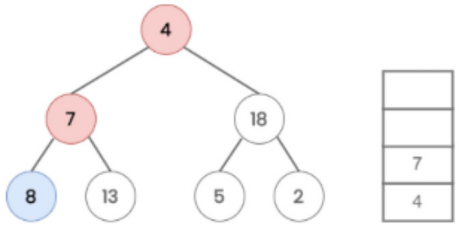
Move to the left of node 4. Now PTR is node 7, which is not NULL. So PUSH it onto the stack.



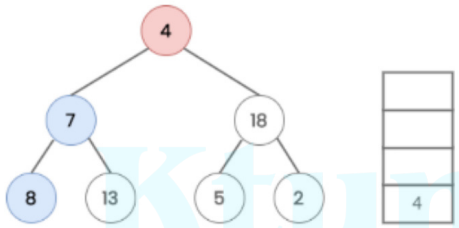
Again, move to the left of node 7. Now PTR is node 8, which is not NULL. So PUSH it onto the stack



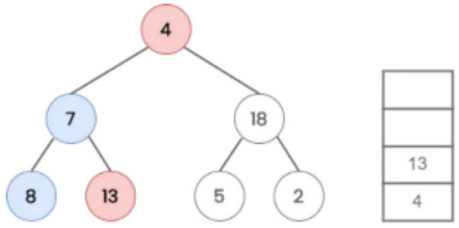
When we move again to the left of node 8, PTR becomes NULL. So POP 8 from the stack. Process PTR (8).



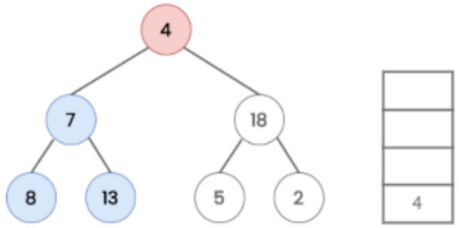
Move to the right child of PTR(8), which is NULL. So POP 7 from the stack and process it. Now PTR points to 7.



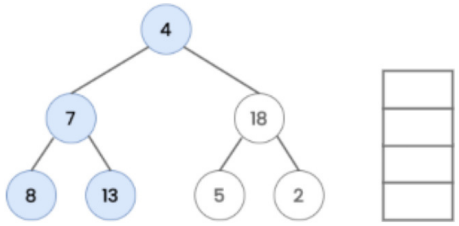
Move to the right child(13) of PTR and PUSH it onto the stack.



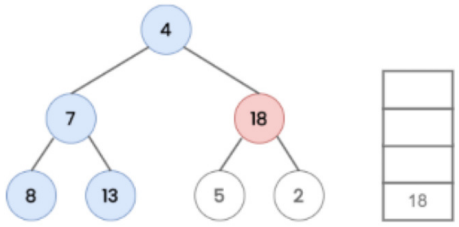
Move to the left of node 13, which is NULL. So POP 13 from the stack and process it.



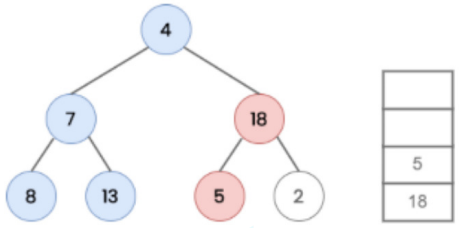
Since node 13 don't have any right child, POP 4 from the stack and process it. Now PTR points to node 4.



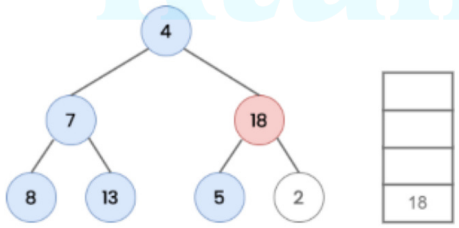
Move to the right of node 4 and put it on to the stack. Now PTR points to node 18.



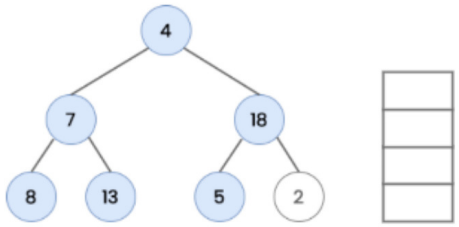
Move to the left(5) child of 18 and put it onto the stack. Now PTR points to node 5.



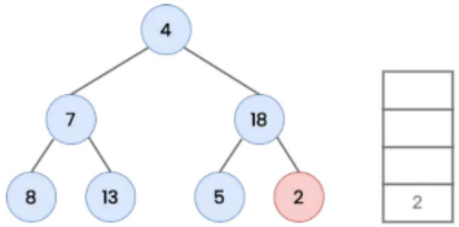
Move to the left of node 5, which is NULL. So POP 5 from the stack and process it.



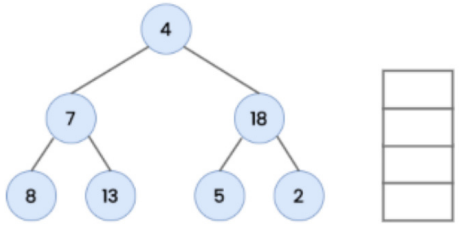
Now, move to the right of node 5, which is NULL. So POP 18 from the stack and process it.



Move to the right(2) of node 18 and PUSH it on to the stack.



Since node 2 has left child, POP 2 from the stack and process it. Now the stack is empty and node 2 has no right child. So stop traversing.



The nodes are processed in the order [8, 7, 13, 4, 5, 18, 2] . This is the required in order traversal of the given tree.

POSTORDER TRAVERSAL

The post order traversal algorithm is more difficult to implement than the previous two algorithms. The following operations are performed to traverse a binary tree in post-order using a stack:

1. **Start from the root, call it PTR.**
2. **Push PTR onto stack if PTR is not NULL.**
3. **Move to left of PTR and repeat step 2.**
4. **If PTR has a right child R, then PUSH -R onto the stack.**
5. **Pop and process positive element from stack and set as PTR.**
6. **If a negative node is popped, (PTR = -N), then set PTR = N and go to step 2.**

PREVIOUS YEAR QUESTIONS

1.What is the difference between recursive and iterative algorithms?(April 2018)

ans:

Recursive.

- **In recursive function call itself until the condition is met**
- **Implemented using function calls**
- **Recursive algorithm uses a branching structure**
- **Recursive solution are often less efficient in times of time and space**
- **It requires extra memory on the stack for each recursive call**
- **Slower in execution**

Iterative

- **The function repeats until the condition fails**
- **An iterative algorithm uses a looping construct**
- **Implemented using loops**

- **More efficient**
- **It terminates when loop continuation condition falls**
- **Faster in execution**

2. Write a recursive algorithm for pre-order traversal in a binary tree.(April 2018,Dec 2018)

ans:

1. **Visit the root node R**
2. **Traverse the left sub-tree of the root node R in inorder**
3. **Traverse the right sub-tree of the root node R in inorder**

3. Write a non recursive algorithm/pseudocode to perform preorder traversal of a binary tree (Dec 2019,Sep 2020,Jan2017,Dec 2020)

ans:

- ***Start with root node and push onto stack.***
- ***Repeat while the stack is not empty***
- ***POP the top element (PTR) from the stack and process the node.***
- ***PUSH the right child of PTR onto to stack.***
- ***PUSH the left child of PTR onto to ststack.***

***4. Write* an iterative algorithm to perform inorder traversal of a binary tree (May 2019)**

ans:

1. **Start from the root, call it PTR.**
2. **Push PTR onto stack if PTR is not NULL.**
3. **Move to left of PTR and repeat step**
4. **If PTR is NULL and stack is not empty, then Pop element from stack and set as PTR.**
5. **Process PTR and move to right of PTR , go to step**

5. Here is a small binary tree:

What is the output obtained after preorder, inorder and postorder traversal of the following tree.

b) Write the non-recursive algorithm for post order traversal of tree.(Dec 2017)

ans:

a) The output obtained after preorder traversal of the given tree would be: 14, 2, 1, 3, 7, 11, 10, 30, 40

The output obtained after inorder traversal of the given tree would be: 1, 2, 3, 7, 14, 10, 11, 30, 40
The output obtained after postorder traversal of the given tree would be: 1, 7, 3, 2, 10, 40, 30, 11, 14

b) Here is a non-recursive algorithm for postorder traversal of a tree:

1. **Create an empty stack and push the root node to it.**
2. **While the stack is not empty, do the following:**
    - **Pop the top node from the stack.**
    - **Push the popped node's left and right children (if they exist) to the stack.**
    - **Add the popped node to the result list.**

3.Return the result list.

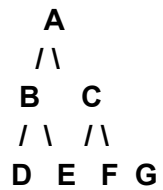
This algorithm works by traversing the tree depth-first and adding nodes to the result list after their children have been processed. This ensures that the nodes are processed in postorder.

6. Write preorder,inorder and post order traversal of following binary tree(Sep 2020,Jan2017)

ans:

Preorder traversal: 5, 7, 3, 1, 6, 9, 2, 4, 8
Inorder traversal: 1, 3, 6, 7, 5, 2, 4, 9, 8
Postorder traversal: 1, 6, 3, 7, 2, 8, 4, 9, 5

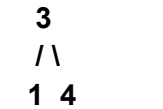
7. Write preorder,inorder and post order traversal of following binary tree(Dec 2020)

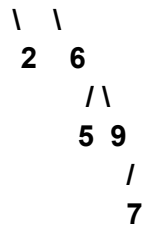


ans:

Preorder traversal: A, B, D, E, C, F, G
Inorder traversal: D, B, E, A, F, C, G
Postorder traversal: D, E, B, F, G, C, A

8. Show the result of inorder, preorder and postorder traversal of given tree(Jan 2020)





ans:
Inorder traversal: 1, 2, 3, 4, 5, 6, 7, 9
Preorder traversal: 3, 1, 2, 4, 6, 5, 9, 7
Postorder traversal: 2, 1, 5, 7, 9, 6, 4, 3

9. What is the characteristics of a recursive algorithm? write non recursive version of binary search tree (Dec 2020)

ans:
A recursive algorithm is an algorithm that solves a problem by breaking it down into smaller subproblems and solving those subproblems recursively. Recursive algorithms are often characterized by the following features:

1. **A base case: This is the simplest version of the problem that can be solved directly without recursion.**
2. **A recursive step: This is a process that reduces the problem to one or more smaller subproblems and then calls itself to solve those subproblems.**

Here is a non-recursive version of a binary search tree (BST) algorithm:

1. **Initialize a current node pointer to the root of the BST.**
2. **While the current node is not null, do the following:**

- **If the value being searched for is less than the value at the current node, set the current node to its left child.**
- **If the value being searched for is greater than the value at the current node, set the current node to its right child.**
- **If the value being searched for is equal to the value at the current node, return the node.**
- **If the current node is null, return null to indicate that the value was not found in the BST.**

3. **If the current node is null, return null to indicate that the value was not found in the BST.**

This algorithm works by iteratively traversing the BST, starting at the root, and following the left or right child based on whether the value being searched for is less than or greater than the current node's value. When the value is found or the end of the tree is reached, the algorithm returns the node or null, respectively.

EXPRESSION TREE

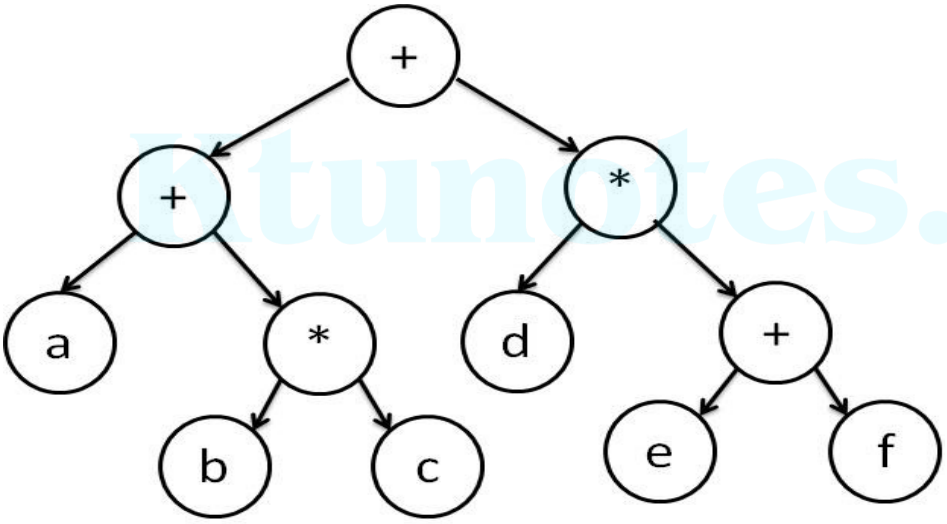
Expression Tree is used to represent expressions.

It is a tree with leaves as operands of the expression and nodes contain the operators.

Expression trees are mainly used for analyzing, evaluating and modifying expressions, especially complex expressions.

An expression and expression tree shown below

```
a + (b * c) + d * (e + f)
```



CONSTRUCTION OF AN EXPRESSION TREE

Let us consider a postfix expression is given as an input for constructing an expression tree. Following are the step to construct an expression tree:

1. Read one symbol at a time from the postfix expression.

2. Check if the symbol is an operand or operator.

3. If the symbol is an operand, create a one node tree and push a pointer onto a stack

4. If the symbol is an operator, pop two pointers from the stack namely T1 & T2 and form a new tree with root as the operator, T1 & T2 as a left and right child

5. A pointer to this new tree is pushed onto the stack

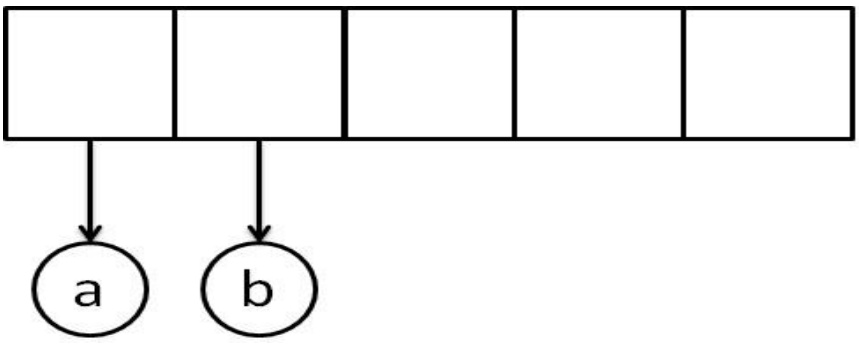
Thus, An expression is created or constructed by reading the symbols or numbers from the left. If operand, create a node. If operator, create a tree with operator as root and two pointers to left and right sub tree.

Example - Postfix Expression Construction

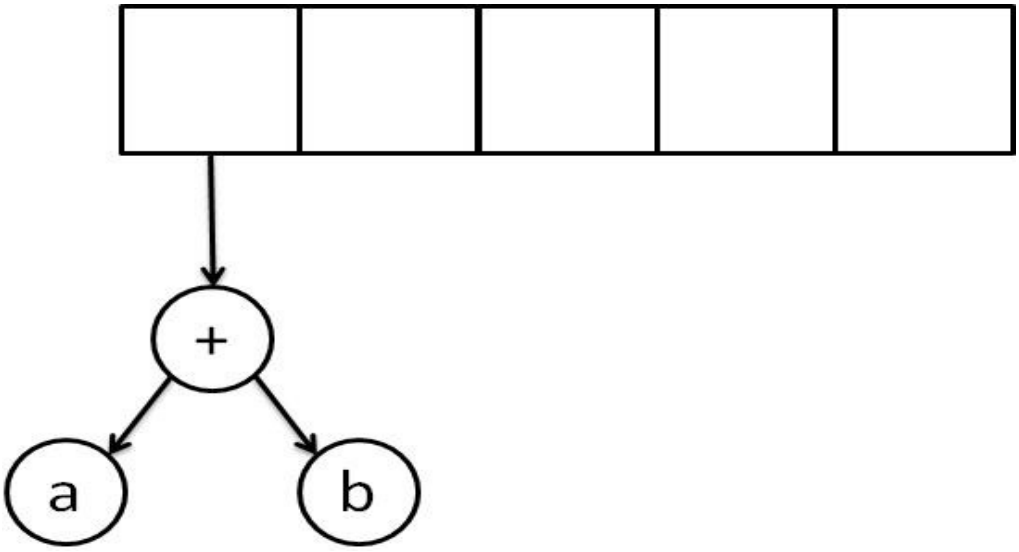
The input is:

```
a b + c *
```

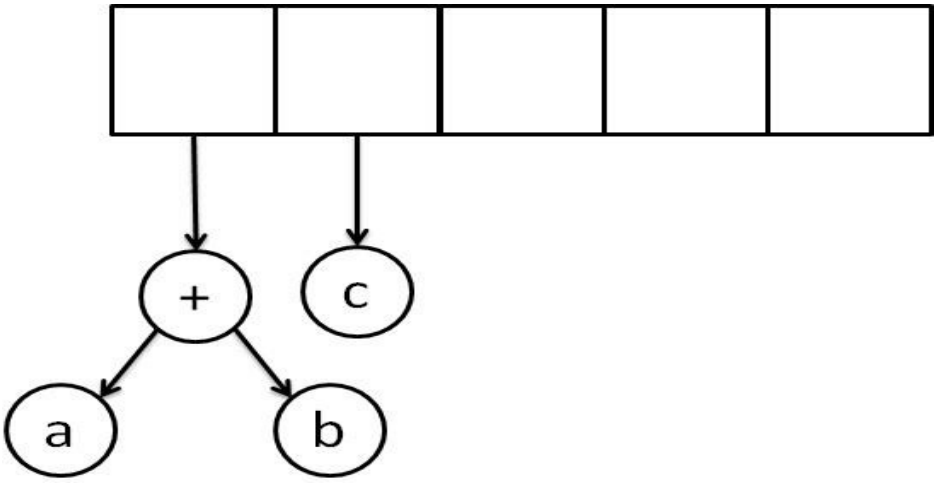
The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



Next, read a**'+'** symbol, so two pointers to tree are popped, a new tree is formed and push a pointer to it onto the stack.

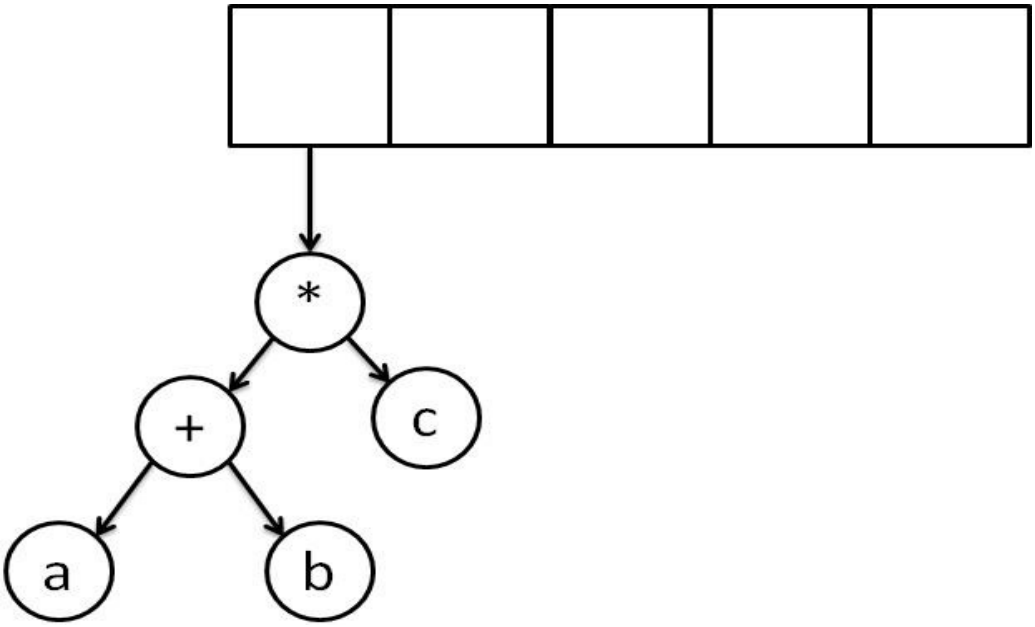


Next, 'c' is read, we create one node tree and push a pointer to it onto the stack.



Finally, the last symbol is read ' * ', we pop two tree pointers and form a new tree with a, ' * ' as root, and a pointer to the final tree

remains on the stack.



Algorithm to build expression tree

Let T be an expression tree

If T is not NULL:

If T->data is an operand:

return T.data

A = solve(T.left)

B = solve(T.right)

--> Calculate operator for 'T.data' on A and B, and call recursively,

return calculate(A, B, T.data)

HEAP TREE

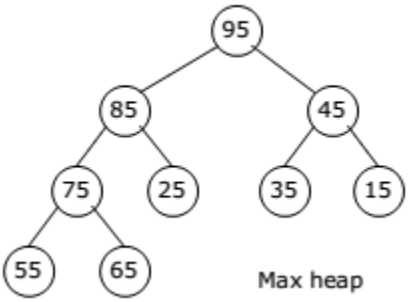
A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

Operations of Heap Data Structure:

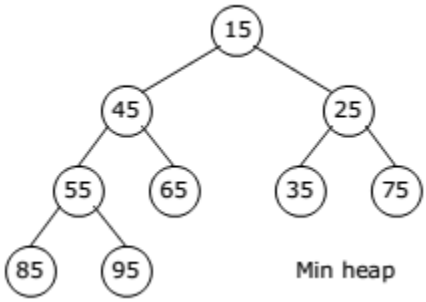
- Heapify: a process of creating a heap from an array.
- Insertion: process to insert an element in existing heap time complexity O(log N).
- Deletion: deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
- Peek: to check or find the most prior element in the heap, (max or min element for max and min heap).

Types of Heap Data Structure

1. Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.
2. Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.



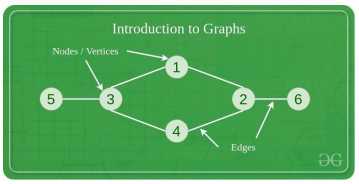
Max heap



Min heap

GRAPH

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).



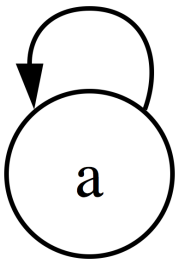
Components of a Graph

- Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph.

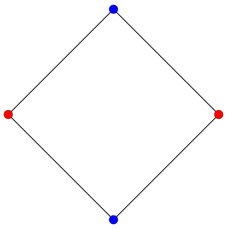
- Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.

End Vertices: Two edges associated with any edge are called end vertices.

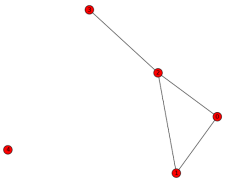
Self loop: Any edge that have only one vertex are called self loop.



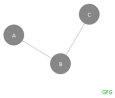
Regular Graph: If the degree of all vertex is same, it is called regular graph.



Isolated vertex: A vertex having no incident edge is called an isolated vertex.



Pendant Vertex: A vertex of degree one is called a pendant vertex or an end vertex.



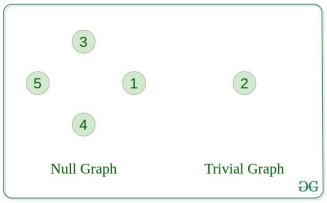
Types Of Graph

- **Null Graph**

  A Graph without any edge is called a null Graph.

- **Trivial Graph**

  Graph having only a single vertex, it is also the smallest graph possible.

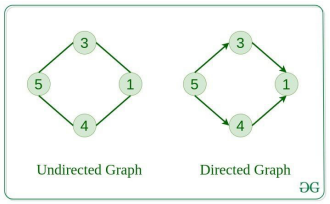


- **Undirected Graph**

  A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

- **Directed Graph**

  A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.

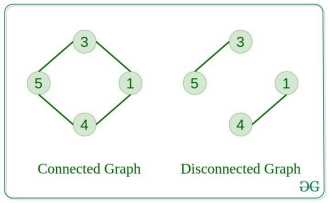


- **Connected Graph**

  The graph in which from one node we can visit any other node in the graph is known as a connected graph.

- **Disconnected Graph**

  The graph in which at least one node is not reachable from a node is known as a disconnected graph.

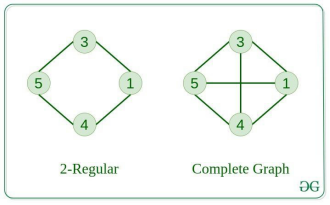


- **Regular Graph**

  The graph in which the degree of every vertex is equal to K is called K regular graph.

- **Complete Graph**

  The graph in which from each node there is an edge to each other node.

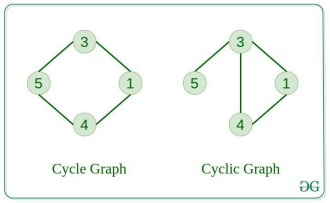


- **Cycle Graph**

  The graph in which the graph is a cycle in itself, the degree of each vertex is 2.

- **Cyclic Graph**

  A graph containing at least one cycle is known as a Cyclic graph.

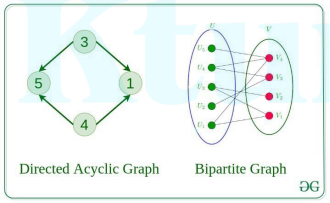


- **Directed Acyclic Graph**

  A Directed Graph that does not contain any cycle.

- **Bipartite Graph**

  A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



- **Weighted Graph**

  A graph in which the edges are already specified with suitable weight is known as a weighted graph. Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

Graph representation

By Graph representation, we simply mean the technique to be used to store some graph into the computer's memory.

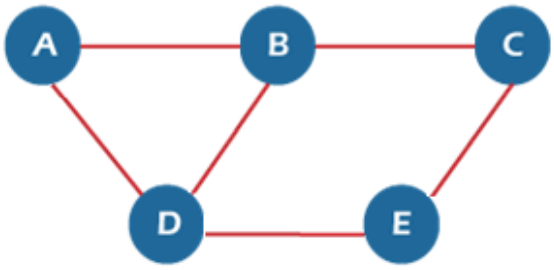
A graph is a data structure that consist a sets of vertices (called nodes) and edges. There are two ways to store Graphs into the computer's memory:

- **Sequential representation** (or, Adjacency matrix representation)
- **Linked list representation** (or, Adjacency list representation)

Adjacency Matrix:

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w. The adjacency matrix for the above example graph is:

Now, let's see the adjacency matrix representation of an undirected graph.



Undirected Graph

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

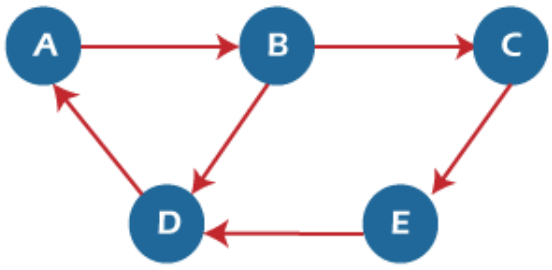
In the above figure, an image shows the mapping among the vertices (A, B, C, D, E), and this mapping is represented by using the adjacency matrix.

There exist different adjacency matrices for the directed and undirected graph. In a directed graph, an entry A_{ij} will be 1 only when there is an edge directed from V_i to V_j.

Adjacency matrix for a directed graph

In a directed graph, edges represent a specific path from one vertex to another vertex. Suppose a path exists from vertex A to another vertex B; it means that node A is the initial node, while node B is the terminal node.

Consider the below-directed graph and try to construct the adjacency matrix of it.



Directed Graph

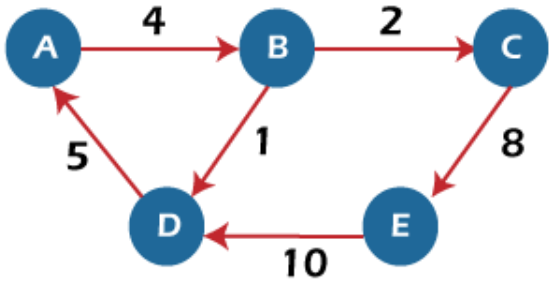
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

In the above graph, we can see there is no self-loop, so the diagonal entries of the adjacent matrix are 0.

Adjacency matrix for a weighted directed graph

It is similar to an adjacency matrix representation of a directed graph except that instead of using the '1' for the existence of a path, here we have to use the weight associated with the edge. The weights on the graph edges will be represented as the entries of the adjacency matrix. We can understand it with the help of an example. Consider the below graph and its adjacency matrix representation. In the representation, we can see that the weight associated with the edges is represented as the entries in the adjacency matrix.



weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

Adjacency Matrix

In the above image, we can see that the adjacency matrix representation of the weighted directed graph is different from other representations. It is because, in this representation, the non-zero values are replaced by the actual weight assigned to the edges.

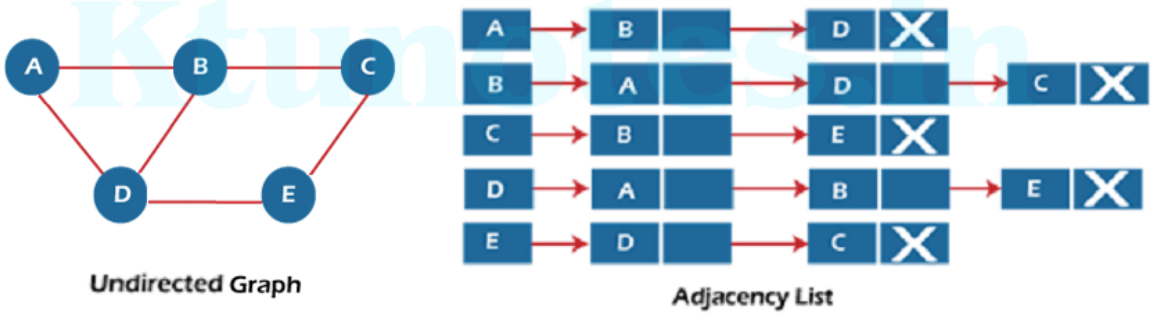
Adjacency matrix is easier to implement and follow. An adjacency matrix can be used when the graph is dense and a number of edges are large.

Though, it is advantageous to use an adjacency matrix, but it consumes more space. Even if the graph is sparse, the matrix still consumes the same space

Adjacency List

An adjacency list is used in the linked representation to store the Graph in the computer's memory. It is efficient in terms of storage as we only have to store the values for edges.

Let's see the adjacency list representation of an undirected graph.

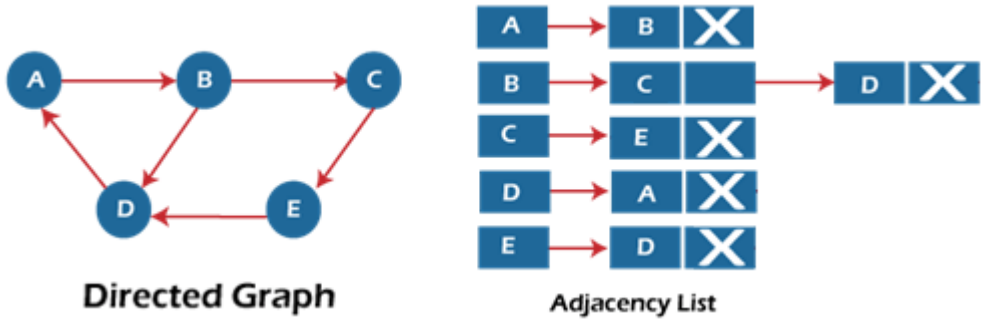


In the above figure, we can see that there is a linked list or adjacency list for every node of the graph. From vertex A, there are paths to vertex B and vertex D. These nodes are linked to nodes A in the given adjacency list.

An adjacency list is maintained for each node present in the graph, which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed, then store the NULL in the pointer field of the last node of the list.

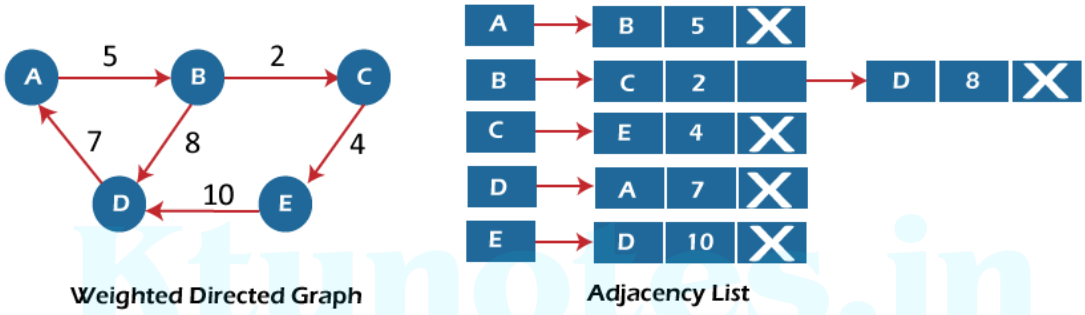
The sum of the lengths of adjacency lists is equal to twice the number of edges present in an undirected graph.

Now, consider the directed graph, and let's see the adjacency list representation of that graph.



For a directed graph, the sum of the lengths of adjacency lists is equal to the number of edges present in the graph.

Now, consider the weighted directed graph, and let's see the adjacency list representation of that graph.



In the case of a weighted directed graph, each node contains an extra field that is called the weight of the node.

In an adjacency list, it is easy to add a vertex. Because of using the linked list, it also saves space.

Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Breadth First Search (BFS)

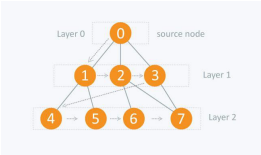
There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

First move horizontally and visit all the nodes of the current layer
Move to the next layer.

Consider the following diagram. enter image description here



The distance between the nodes in layer 1 is comparitively lesser than the distance between the nodes in layer 2. Therefore, in BFS, you must traverse all the nodes in layer 1 before you move to the nodes in layer 2.

Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

In the earlier diagram, start traversing from 0 and visit its child nodes 1, 2, and 3. Store them in the order in which they are visited. This will allow you to visit the child nodes of 1 first (i.e. 4 and 5), then of 2 (i.e. 6 and 7), and then of 3 (i.e. 7) etc.

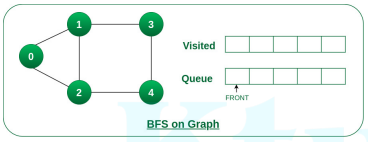
To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neigbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Follow the below method to implement BFS traversal.

Declare a queue and insert the starting vertex.
Initialize a visited array and mark the starting vertex as visited.
Follow the below process till the queue becomes empty:
Remove the first vertex of the queue.
Mark that vertex as visited.
Insert all the unvisited neighbors of the vertex into the queue.

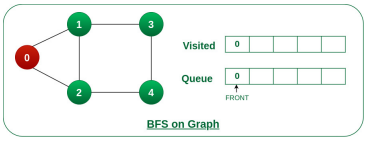
Illustration:

Step1: Initially queue and visited arrays are empty.



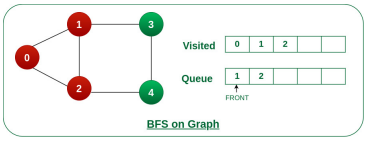
Queue and visited arrays are empty initially.

Step2: Push node 0 into queue and mark it visited.



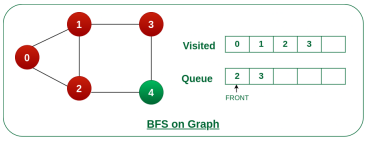
Push node 0 into queue and mark it visited.

Step 3: Remove node 0 from the front of queue and visit the unvisited neighbours and push them into queue.



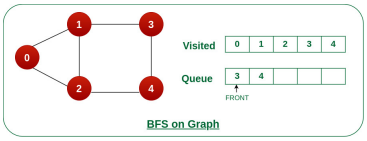
Remove node 0 from the front of queue and visited the unvisited neighbours and push into queue.

Step 4: Remove node 1 from the front of queue and visit the unvisited neighbours and push them into queue.



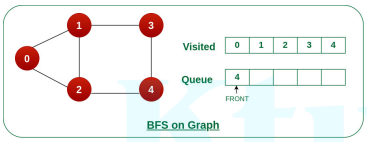
Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.



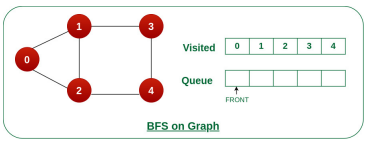
Remove node 2 from the front of queue and visit the unvisited neighbours and push them into queue.

Step 6: Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.
As we can see that every neighbours of node 3 is visited, so move to the next node that are in the front of the queue.



Remove node 3 from the front of queue and visit the unvisited neighbours and push them into queue.

Steps 7: Remove node 4 from the front of queue and and visit the unvisited neighbours and push ithem into queue.
As we can see that every neighbours of node 4 are visited, so move to the next node that is in the front of the queue.

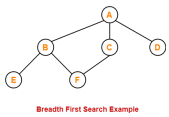


Remove node 4 from the front of queue and and visit the unvisited neighbours and push ithem into queue.

Now, Queue becomes empty, So, terminate these process of iteration.

BFS Example-

Consider the following graph-



Breadth First Search Example

The breadth first search traversal order of the above graph is-

A, B, C, D, E, F

Breadth First Search Algorithm-

BFS (V,E,s)

for each vertex v in V – {s}

do

color[v] ← WHITE

d[v] ← ∞

π[v] ← NIL

color[s] = GREY

d[s] ← 0

π[s] ← NIL

Q ← { }

ENQUEUE (Q,s)

While Q is non-empty

do v ← DEQUEUE (Q)

for each u adjacent to v

do if color[u] ← WHITE

then color[u] ← GREY

d[u] ← d[v] + 1

π[u] ← v

ENQUEUE (Q,u)

color[v] ← BLACK

Explanation-

The above breadth first search algorithm is explained in the following steps-

Step-01

Create and maintain 3 variables for each vertex of the graph.

For any vertex ‘v’ of the graph, these 3 variables are-

1. color[v]-

This variable represents the color of the vertex v at the given point of time.
The possible values of this variable are- WHITE, GREY and BLACK.
WHITE color of the vertex signifies that it has not been discovered yet.
GREY color of the vertex signifies that it has been discovered and it is being processed.
BLACK color of the vertex signifies that it has been completely processed.

2. Π[v]-

This variable represents the predecessor of vertex ‘v’.

3. d[v]-

This variable represents the distance of vertex v from the source vertex.

Step-02

For each vertex v of the graph except source vertex, initialize the variables as-

color[v] = WHITE
π[v] = NIL
d[v] = ∞

For source vertex S, initialize the variables as-

color[S] = GREY
π[S] = NIL
d[S] = 0

Step-03

Now, enqueue source vertex S in queue Q and repeat the following procedure until the queue Q becomes empty-

1. Dequeue vertex v from queue Q.

2. For all the adjacent white vertices u of vertex v,

do-

color[u] = GREY

d[u] = d[v] + 1

π[u] = v

Enqueue (Q,u)

3. Color vertex v to black.

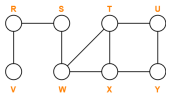
BFS Time Complexity-

The total running time for Breadth First Search is O (V+E).

PRACTICE PROBLEM BASED ON BREADTH FIRST SEARCH-

Problem-

Traverse the following graph using Breadth First Search Technique-



Consider vertex S as the starting vertex.

Solution-

Step-01:

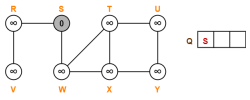
For all the vertices v except source vertex S of the graph, we initialize the variables as-

color[v] = WHITE
π[v] = NIL
d[v] = ∞

For source vertex S, we initialize the variables as-

color[S] = GREY
π[S] = NIL
d[S] = 0

We enqueue the source vertex S in the queue Q.



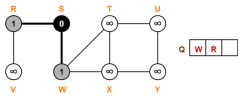
Step-02:

Dequeue vertex S from the queue Q
For all adjacent white vertices ‘v’ (vertices R and W) of vertex S, we do-

color[v] = GREY

d[v] = d[S] + 1 = 0 + 1 = 1
π[v] = S
Enqueue all adjacent white vertices of S in queue Q

color[S] = BLACK

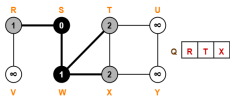


Step-03:

Dequeue vertex W from the queue Q
For all adjacent white vertices ‘v’ (vertices T and X) of vertex W, we do-

color[v] = GREY
d[v] = d[W] + 1 = 1 + 1 = 2
π[v] = W
Enqueue all adjacent white vertices of W in queue Q

color[W] = BLACK

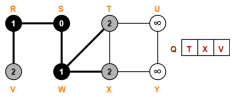


Step-04:

Dequeue vertex R from the queue Q
For all adjacent white vertices ‘v’ (vertex V) of vertex R, we do-

color[v] = GREY
d[v] = d[R] + 1 = 1 + 1 = 2
π[v] = R
Enqueue all adjacent white vertices of R in queue Q

color[R] = BLACK

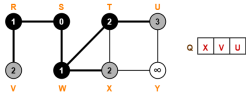


Step-05:

Dequeue vertex T from the queue Q
For all adjacent white vertices ‘v’ (vertex U) of vertex T, we do-

color[v] = GREY
d[v] = d[T] + 1 = 2 + 1 = 3
π[v] = T
Enqueue all adjacent white vertices of T in queue Q

color[T] = BLACK

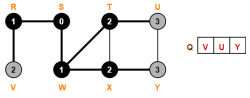


Step-06:

Dequeue vertex X from the queue Q
For all adjacent white vertices ‘v’ (vertex Y) of vertex X, we do-

color[v] = GREY
d[v] = d[X] + 1 = 2 + 1 = 3
π[v] = X
Enqueue all adjacent white vertices of X in queue Q

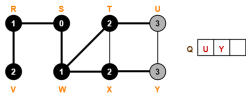
color[X] = BLACK



Step-07:

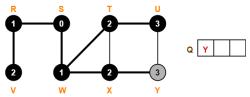
Dequeue vertex V from the queue Q

There are no adjacent white vertices to vertex V.
color[V] = BLACK



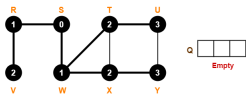
Step-08:

Dequeue vertex U from the queue Q
There are no adjacent white vertices to vertex U.
color[U] = BLACK



Step-09:

Dequeue vertex Y from the queue Q
There are no adjacent white vertices to vertex Y.
color[Y] = BLACK



Since, all the vertices have turned black and the queue has got empty, so we stop.

This is how any given graph is traversed using Breadth First Search (BFS) technique.

To gain better understanding about Breadth First Search Algorithm.

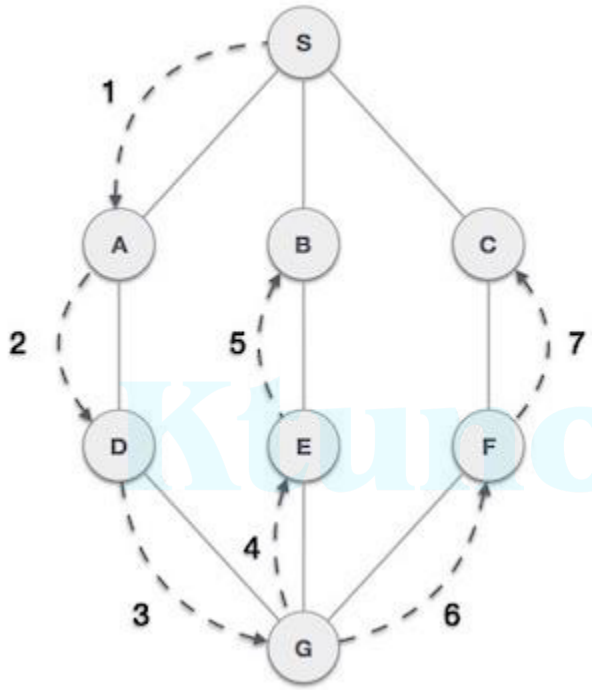
GRAPH TRAVERSAL

Graph traversal (also known as graph search) refers to the process of visiting (checking and/or updating) each vertex in a graph. Such traversals are classified by the order in which the vertices are visited. Tree traversal is a special case of graph traversal.

DEPTH FIRST SEARCH

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

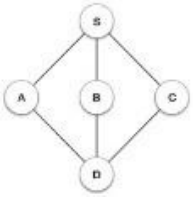
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first,



then to F and lastly to C. It employs the following rules.

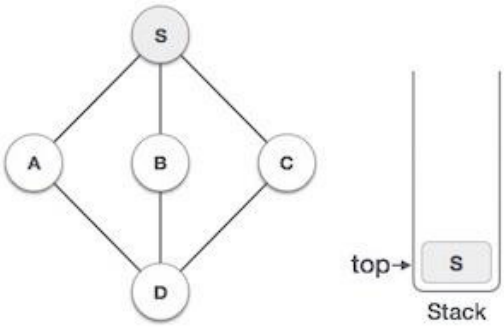
- **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step 1:



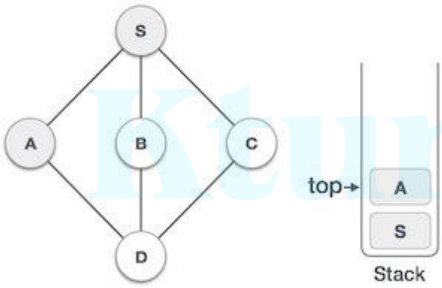
Initialize the stack

Step 2:



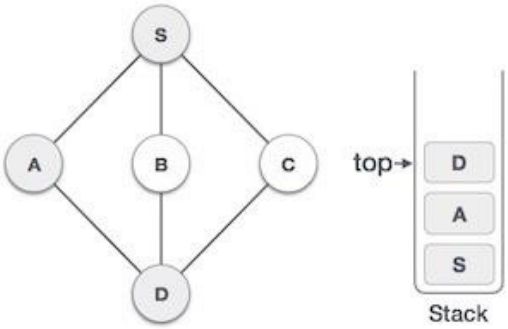
Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.

Step 3:



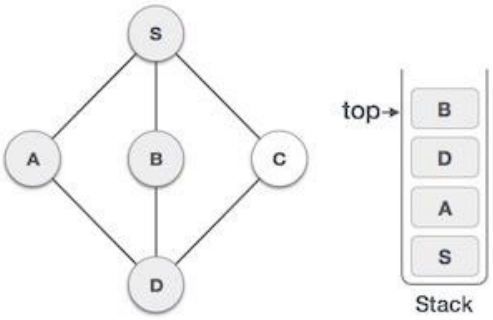
Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.

Step 4:



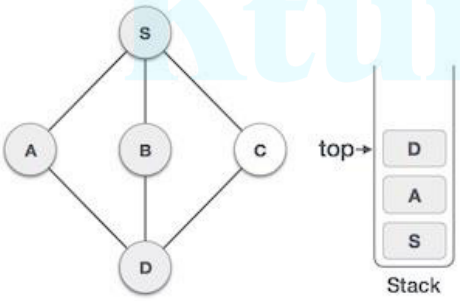
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

Step 5:



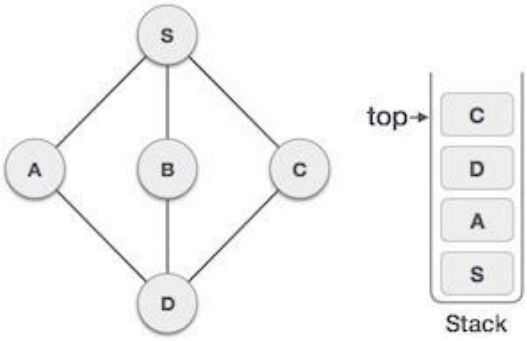
Choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

Step 6:



Check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

Step 7:



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so keep popping the stack until find a node that has an unvisited adjacent node. In this case, there's none and keep popping until the stack is empty.

IMPLEMENTATION IN C

```
// DFS algorithm in C

#include <stdio.h>
#include <stdlib.h>

struct node {
  int vertex;
  struct node* next;
};

struct node* createNode(int v);

struct Graph {
  int numVertices;
  int* visited;

  // We need int** to store a two dimensional array.
  // Similary, we need struct node** to store an array of Linked lists
  struct node** adjLists;
};

// DFS algo
void DFS(struct Graph* graph, int vertex) {
  struct node* adjList = graph->adjLists[vertex];
  struct node* temp = adjList;

  graph->visited[vertex] = 1;
  printf("Visited %d \n", vertex);

  while (temp != NULL) {
    int connectedVertex = temp->vertex;

    if (graph->visited[connectedVertex] == 0) {
      DFS(graph, connectedVertex);
    }
```

```
    temp = temp->next;
  }
}

// Create a node
struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}

// Create graph
struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));

  graph->visited = malloc(vertices * sizeof(int));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
    graph->visited[i] = 0;
  }
  return graph;
}

// Add edge
void addEdge(struct Graph* graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  // Add edge from dest to src
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}

// Print the graph
void printGraph(struct Graph* graph) {
  int v;
  for (v = 0; v < graph->numVertices; v++) {
    struct node* temp = graph->adjLists[v];
    printf("\n Adjacency list of vertex %d\n ", v);
```

```
    while (temp) {
      printf("%d -> ", temp->vertex);
      temp = temp->next;
    }
    printf("\n");
  }
}

int main() {
  struct Graph* graph = createGraph(4);
  addEdge(graph, 0, 1);
  addEdge(graph, 0, 2);
  addEdge(graph, 1, 2);
  addEdge(graph, 2, 3);

  printGraph(graph);

  DFS(graph, 2);

  return 0;
}
```

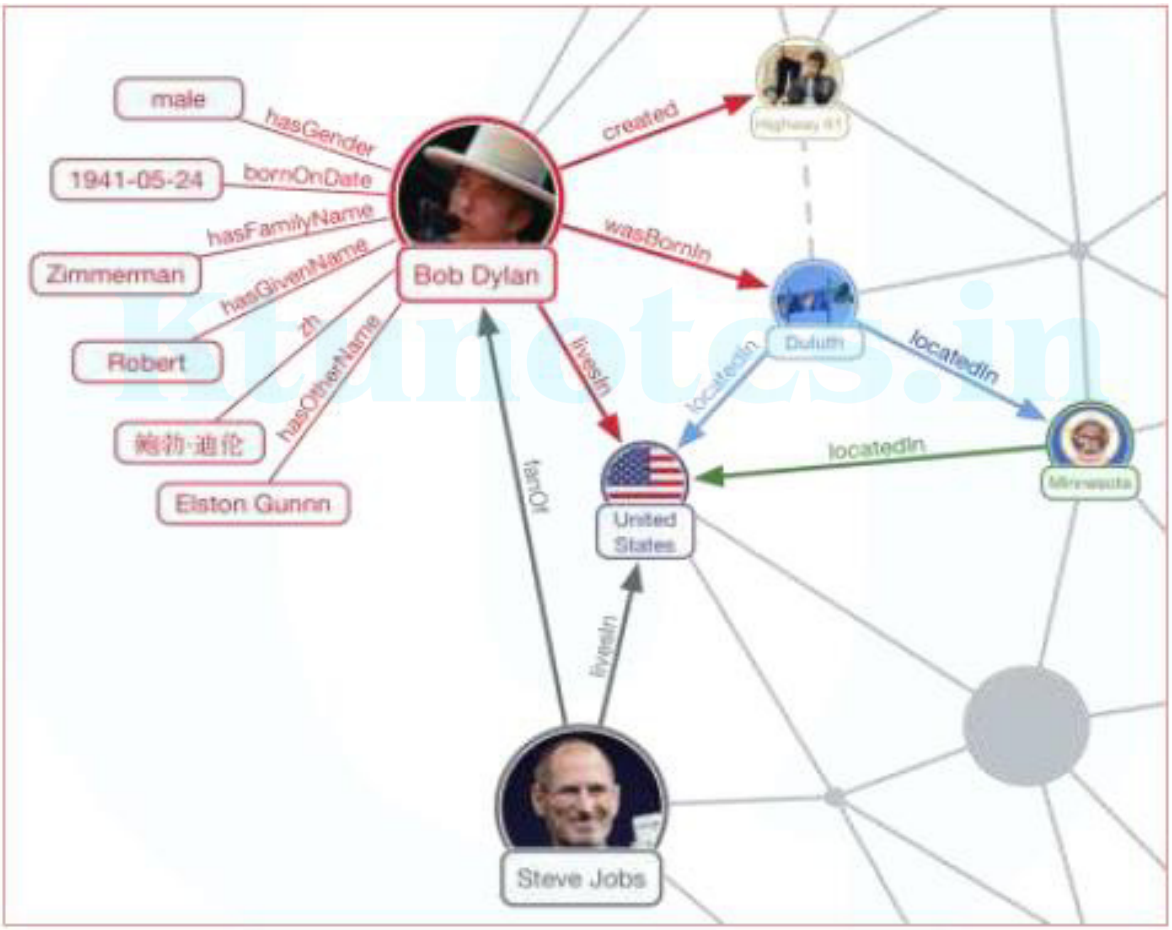

Application of DFS Algorithm

- For finding the path
- To test if the graph is bipartite
- For finding the strongly connected components of a graph
- For detecting cycles in a graph

Application of Graphs

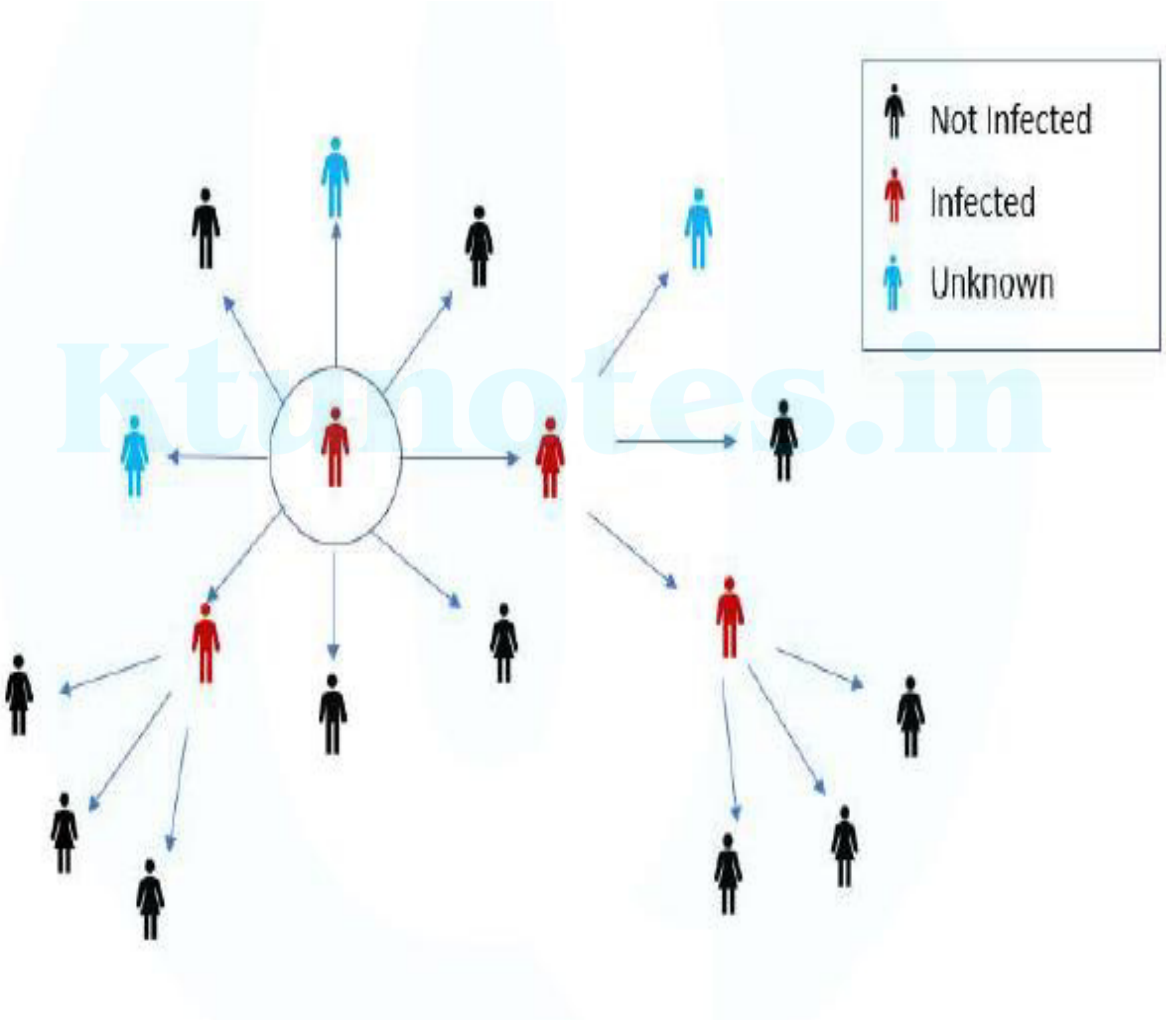
- Knowledge Graphs
- Social Graphs Web Document Graphs
- Neural Networks
- Road Networks
- Blockchain Networks
- Path Optimization
- Contact Tracing

Example for a Knowledge Graph

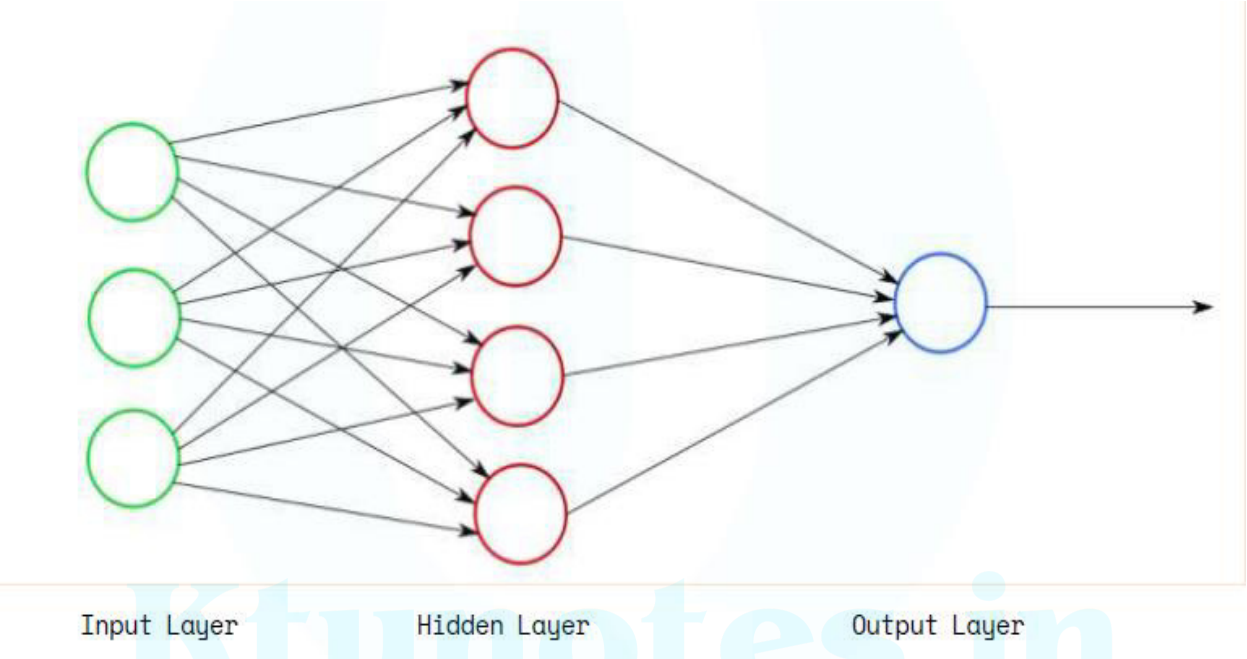


Example for a Social Graph

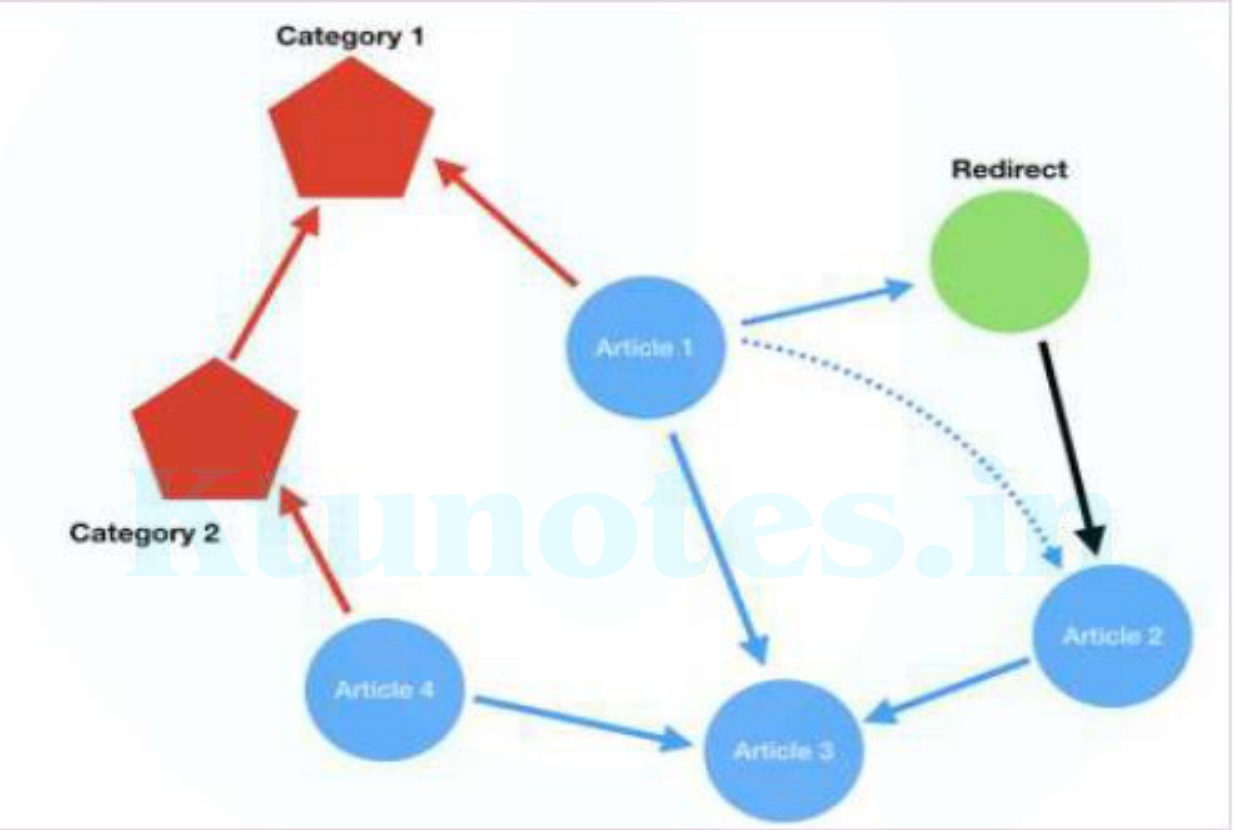
Example for a Contact Tracing Graph



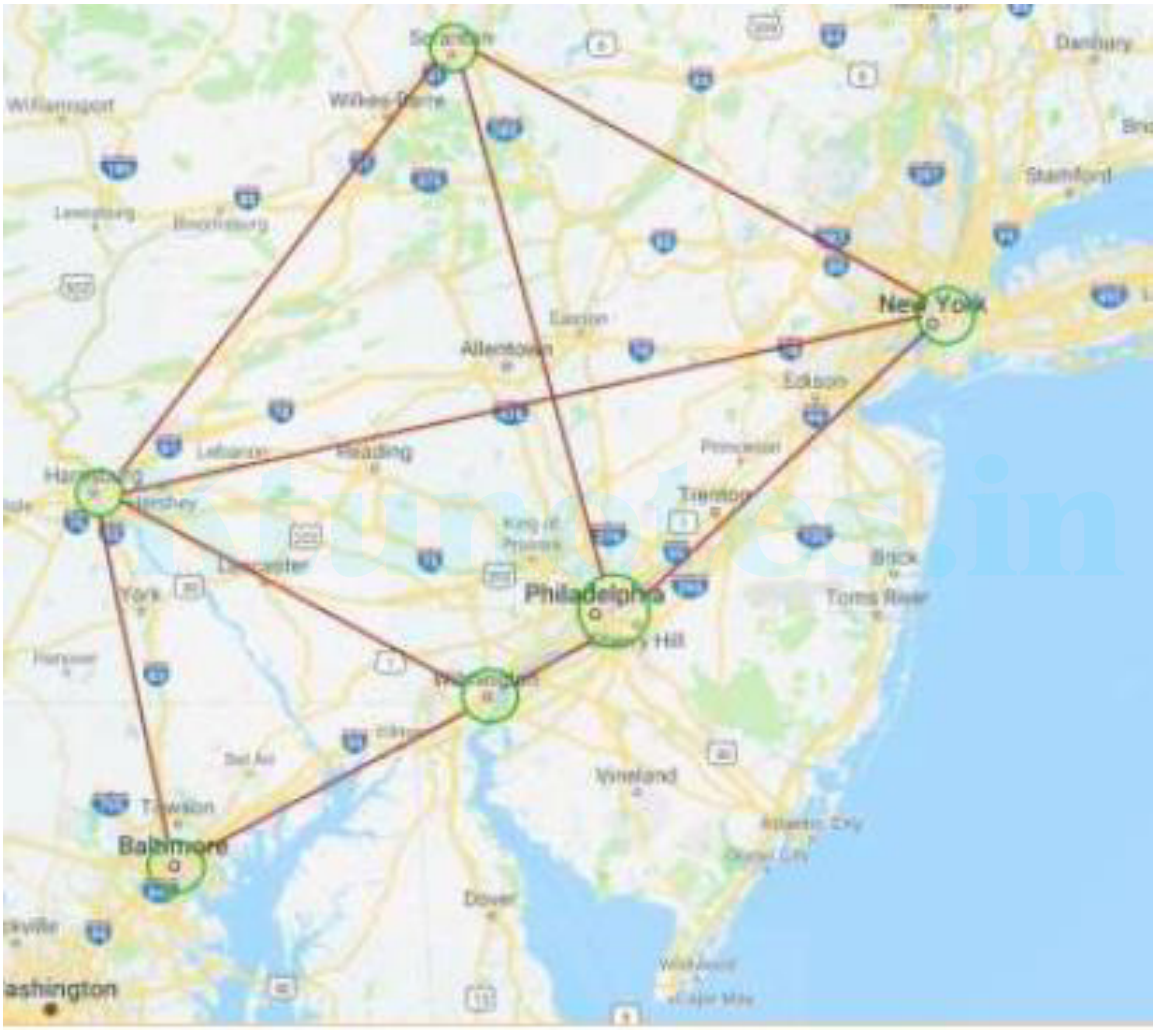
Example for a Neural Network Graph



Example for a Web Document Graph



Example for a Road Network Graph



1. Social networks: Graphs can be used to represent social networks and study the relationships between people.
2. Transportation networks: Graphs can be used to represent transportation networks, such as roads and highways, and to optimize the routes taken by vehicles.
3. Biological networks: Graphs can be used to represent biological networks, such as protein-protein interaction networks and metabolic networks.
4. Computer science: Graphs are used in computer science to represent algorithms, data structures, and the relationships between them.
5. Data visualization: Graphs are often used to visualize data and present it in a more understandable way.
6. Network analysis: Graphs are used to study and analyze networks, including social networks, transportation networks, and biological networks.
7. Machine learning: Graphs are used in machine learning to represent relationships between data points and to build predictive models.
8. Image processing: Graphs are used in image processing to represent the relationships between pixels in an image.
9. Natural language processing: Graphs are used in natural language processing to represent the relationships between words and phrases in a text.
10. Internet: Graphs are used to represent the structure of the internet and the relationships between websites.

Dijkstra's Algorithm

Dijkstra's algorithm allows us to find the shortest path between any two vertices of a graph.

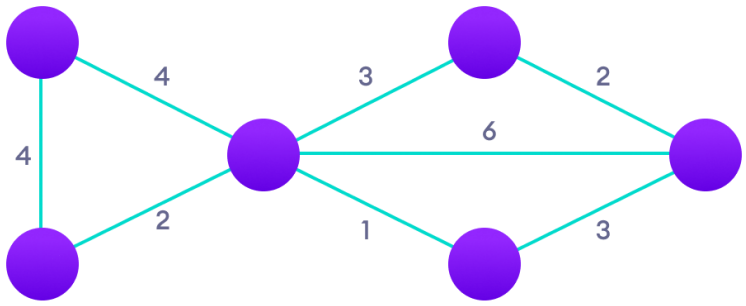
It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

How Dijkstra's Algorithm works

Dijkstra's Algorithm works on the basis that any sub path `B -> D` of the shortest path `A -> D` between vertices A and D is also the shortest path between vertices B and D.
Djikstra used this property in the opposite direction i.e we overestimate the distance of each vertex from the starting vertex. Then we visit each node and its neighbors to find the shortest subpath to those neighbors.

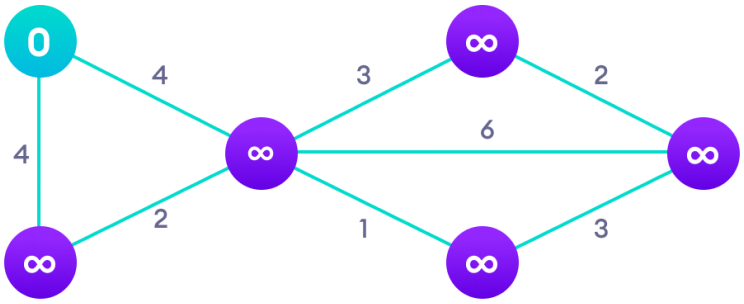
The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

Example of Dijkstra's algorithm



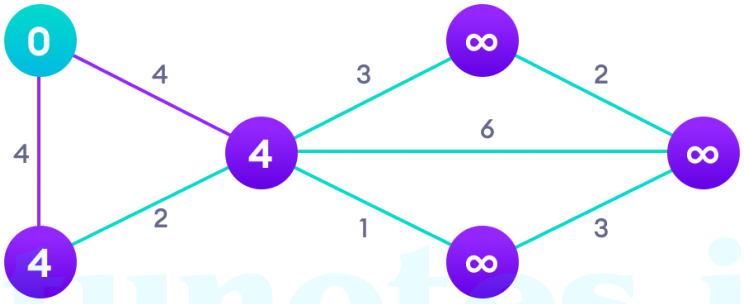
Step: 1

Start with a weighted graph



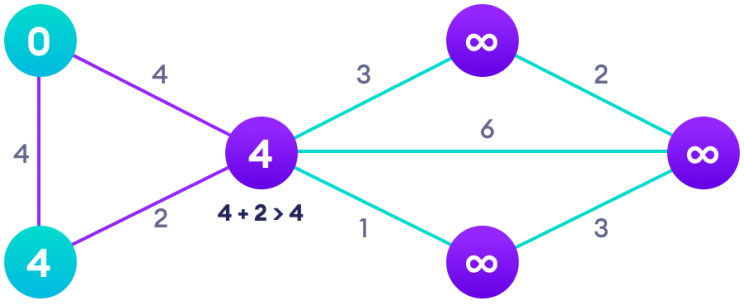
Step: 2

Choose a starting vertex and assign infinity path values to all other devices



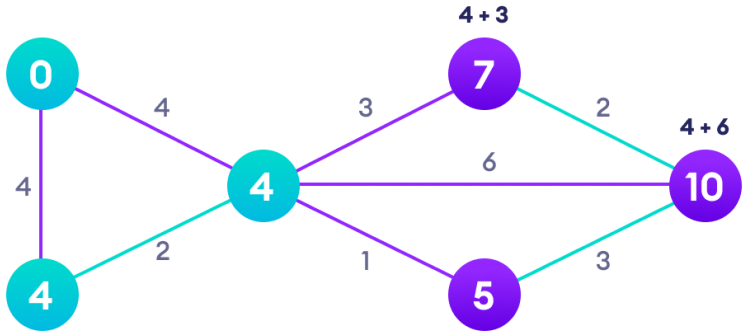
Step: 3

Go to each vertex and update its path length



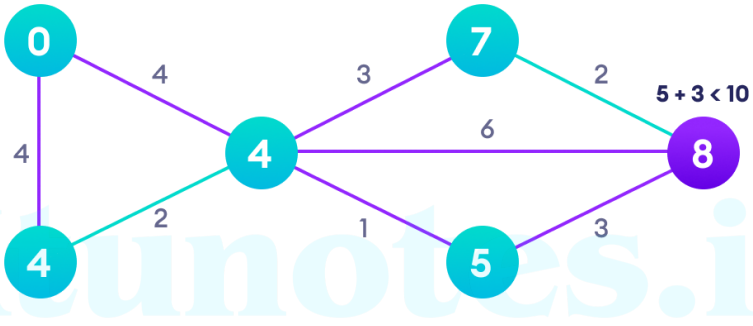
Step: 4

If the path length of the adjacent vertex is lesser than new path length, don't update it



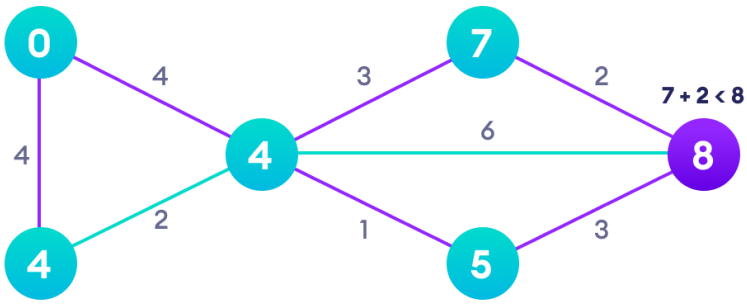
Step: 5

Avoid updating path lengths of already visited vertices



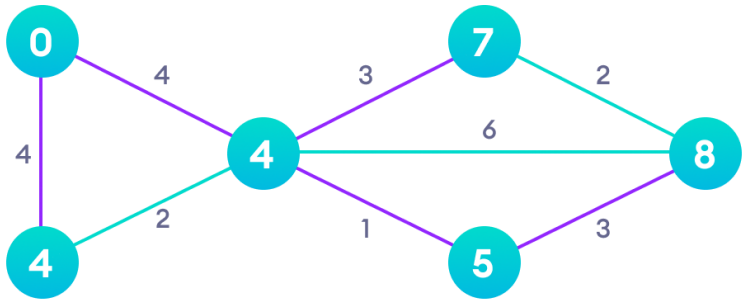
Step: 6

After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Step: 7

Notice how the rightmost vertex has its path length updated twice



Step: 8

Repeat until all the vertices have been visited

Djikstra's algorithm pseudocode

Once the algorithm is over, we can backtrack from the destination vertex to the source vertex to find the path.

A minimum priority queue can be used to efficiently receive the vertex with least path distance.

We need to maintain the path distance of every vertex. We can store that in an array of size v, where v is the number of vertices.

We also want to be able to get the shortest path, not only know the length of the shortest path. For this, we map each vertex to the vertex that last updated its path length.

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q
    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
```

```
                distance[V] <- tempDistance
                previous[V] <- U
    return distance[], previous[]
```

Dijkstra's Algorithm Complexity

Time Complexity: `O(E Log V)`
where, E is the number of edges and V is the number of vertices.

Space Complexity: `O(V)`

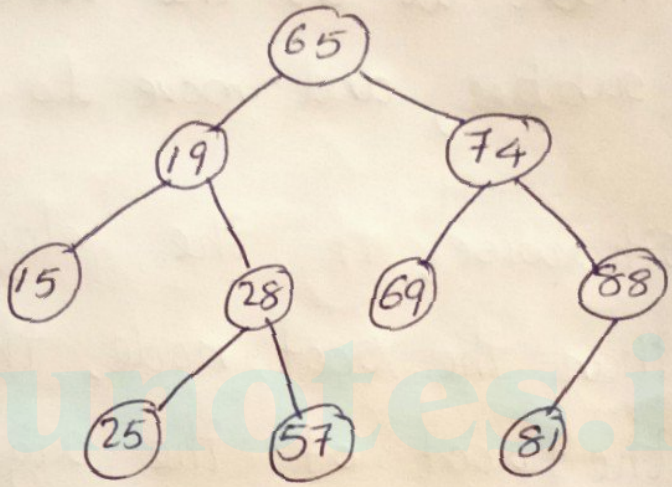
Dijkstra's Algorithm Applications

- To find the shortest path
- In social networking applications
- In a telephone network
- To find the locations in the map

Binary Search Tree

→ The value of a node (N) is greater than every value in the left sub-tree of node (N) and is less than every value in the right subtree of node (N).

eg:-



Advantages:-

→ Searching an element is easy.

→ Insertion and deletion are faster as compared with array and linked lists

Creation

Q. Create a BST of the following values.

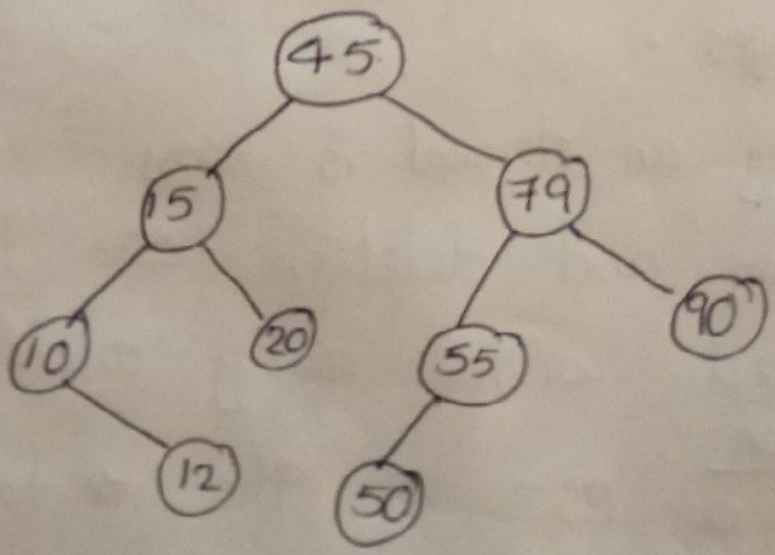
45, 15, 79, 90, 10, 55, ~~20~~ 12, 20, 50

* First, we have to insert first value into the tree as the root of the tree.

* Then read the next element;

  If it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.

* Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

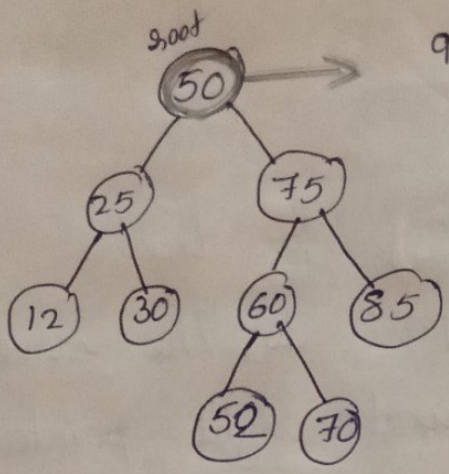


Insertion

steps :-

1) Allocate the memory for tree.
2) set the data part to the value and set the left and right pointer of tree, point to NULL.
3) If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4) Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5) If this is false, then perform this operation recursively with the right subtree of the root.

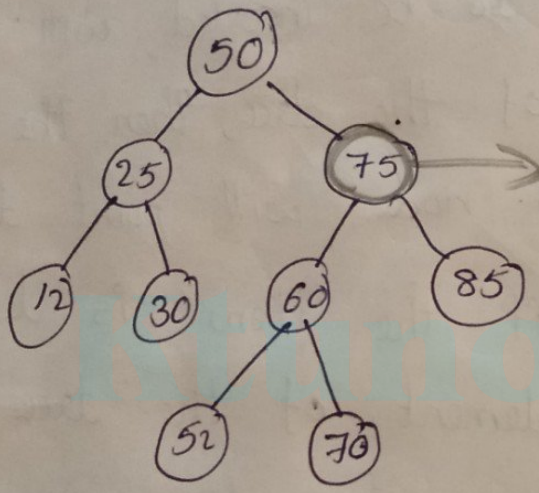
Eg:- Insert 95 into the given tree.



95 > 50

∴ root = root → right

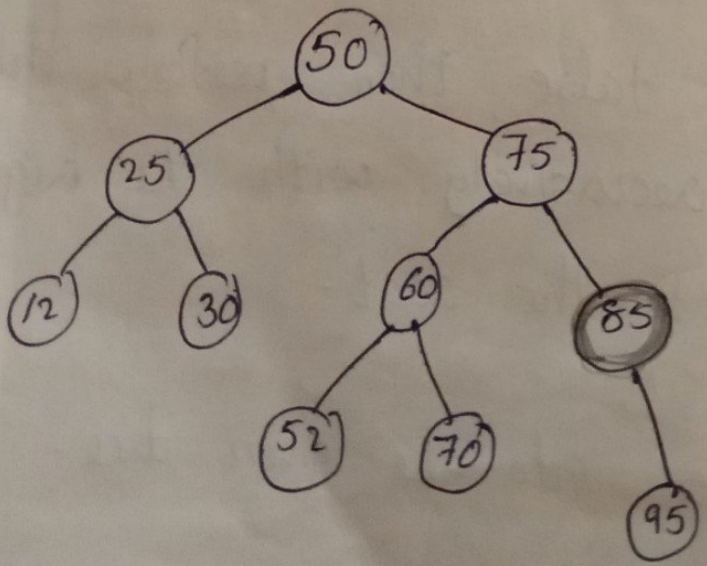
ie, root = 75



95 > 75

∴ root = root → right

= 85



95 > 85

∴ root = root → right

= NULL

∴ Insert the item 95 to the right of 85.

Deletion

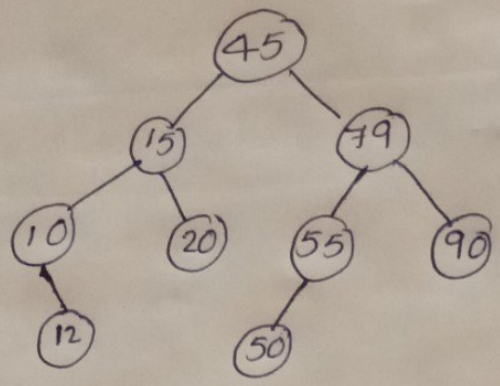
→ ~~There are three~~

Searching

steps :-

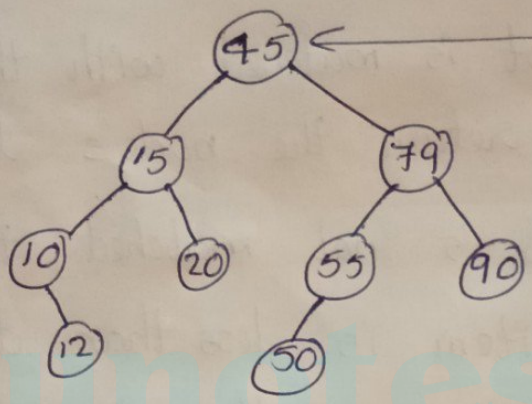
1) First, compare the element to be searched with the root element of the tree.
2) If root is matched with the target element, then return the node's location.
3) If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4) If it is larger than the root element, then move to the right subtree.
5) Repeat the above procedure recursively until the match is found.
6) If the element is not found or not present in the tree, then return NULL.

Eg :-



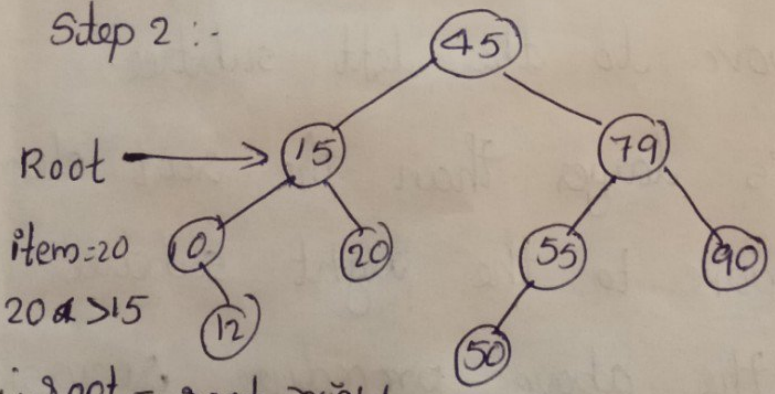
Suppose we have to find node 20.

Step 1 :-



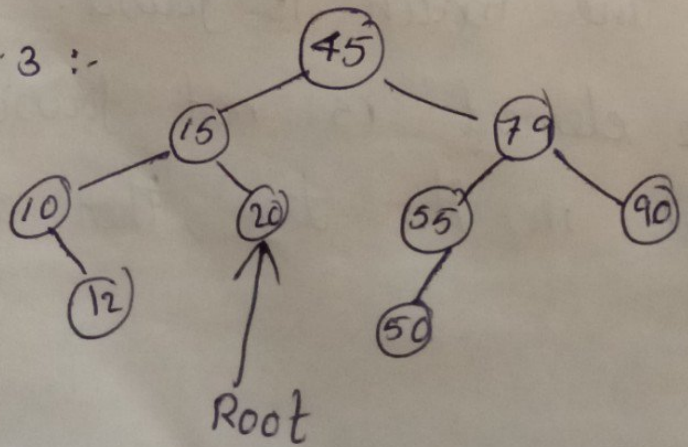
Root
Item = 20
20 < 45
∴ Root = Root → left

Step 2 :-



∴ root = root → right

Step 3 :-



item = 20
20 = root
return root

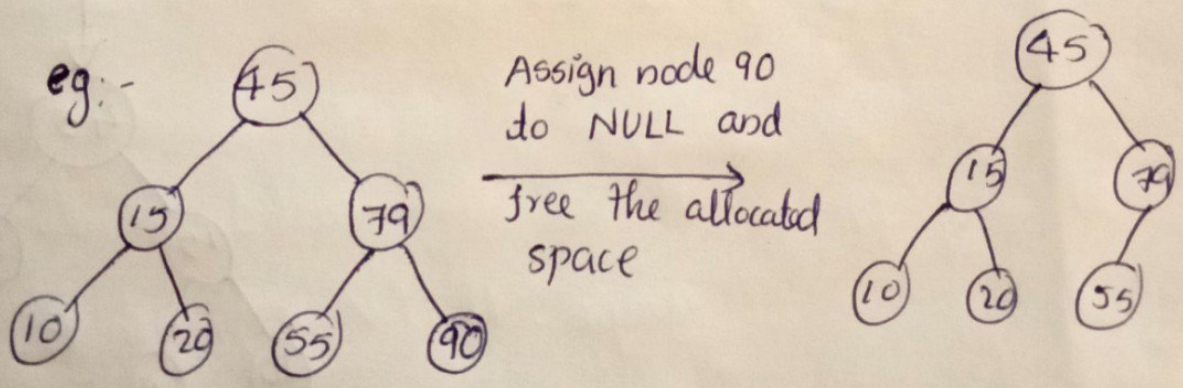
Deletion

→ There are 3 possible situations.

1) The node to be deleted is the leaf node, or
2) The node to be deleted has only one child
3) The node to be deleted has two children.

★ When the node to be deleted is the leaf node

→ Here, we have to replace the leaf node with NULL and simply free the allocated space.

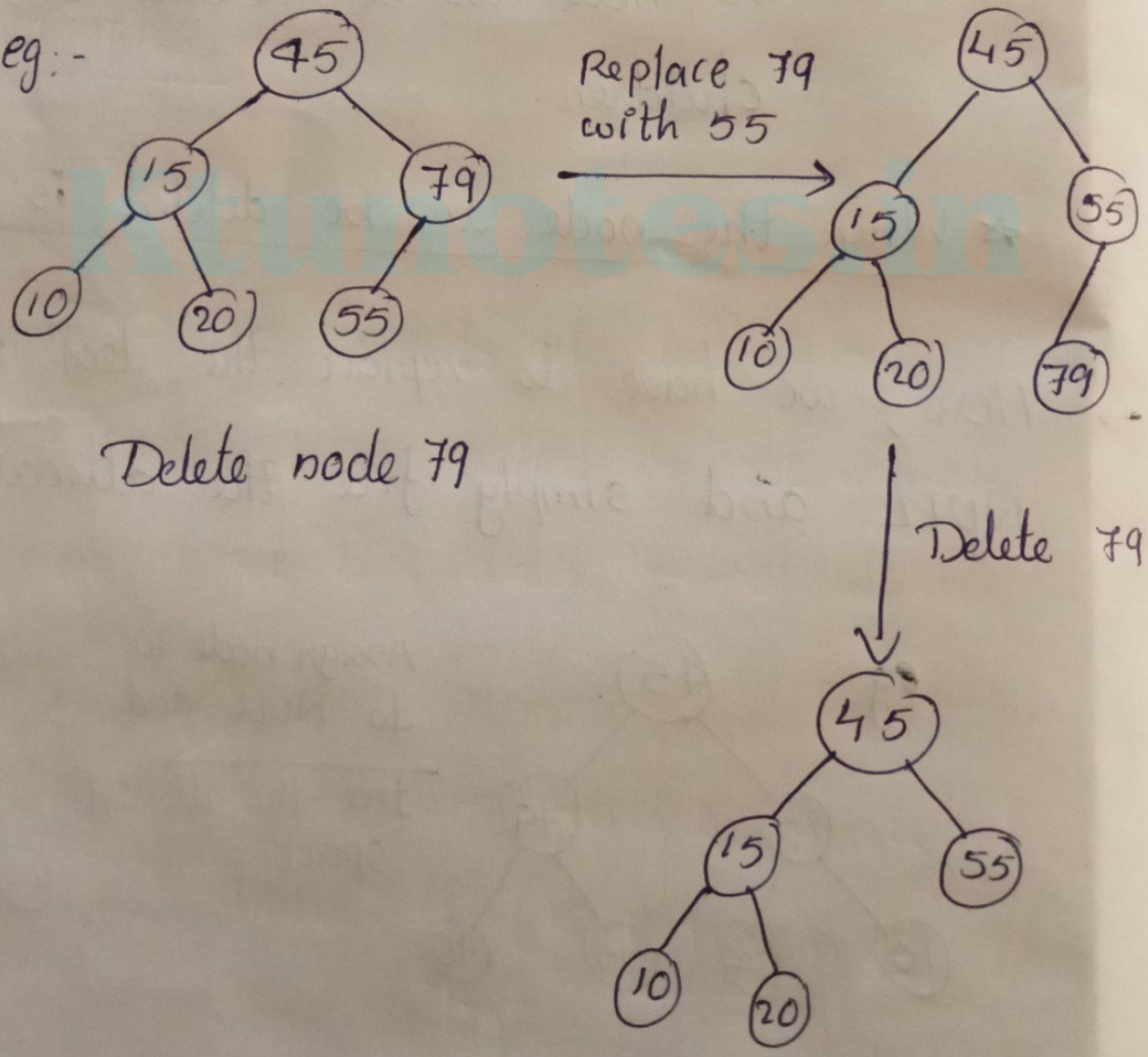


Delete node 90

When the node to be deleted has only one child

→ In this case, we have do replace the target node with its child, and then delete the child node.

eg:-

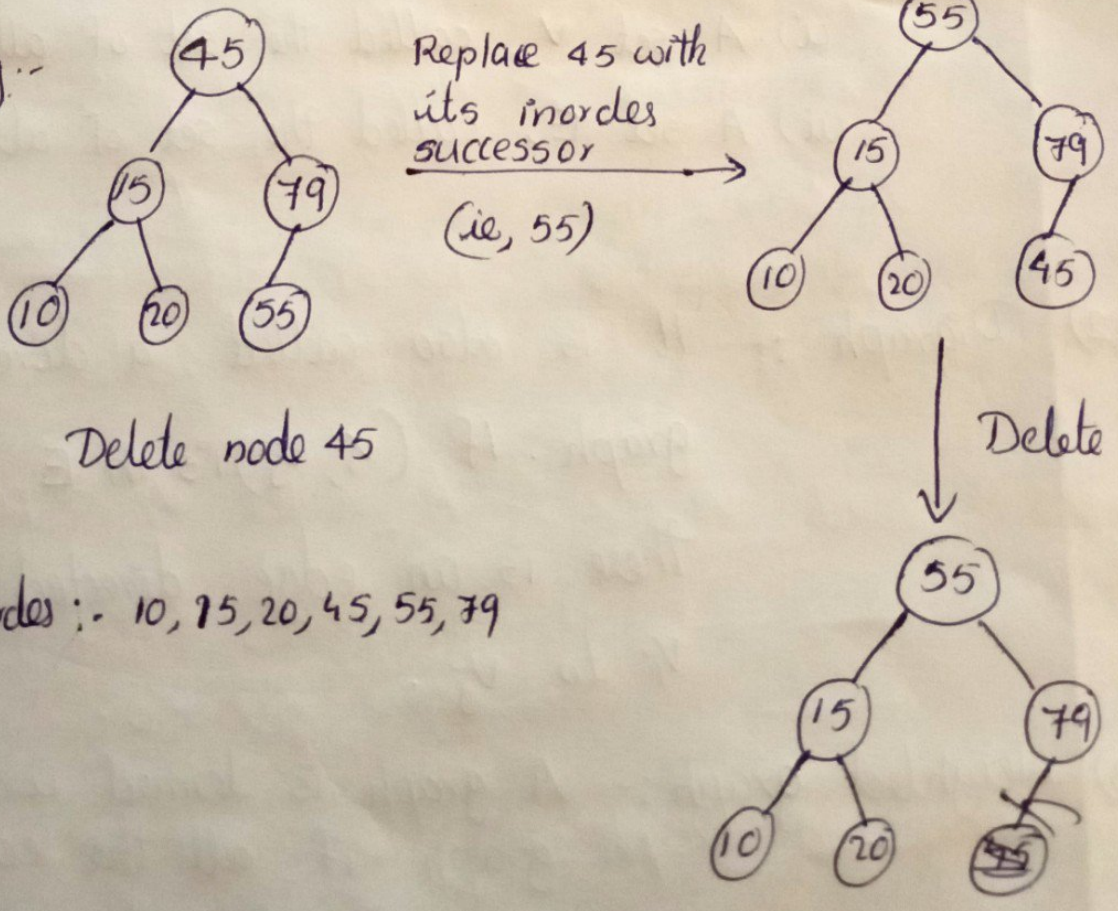


When the node to be deleted has two children

Steps :-

1) First, find the inorder successor of the node to be deleted.
2) After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
3) And at last, replace the node with NULL and free up the allocated space.

eg:-



Inorder :- 10, 15, 20, 45, 55, 79

→ The inorder successor is required when the right child of the node is not empty.

→ We can obtain the inorder successor by finding the minimum element in the right child of the node.