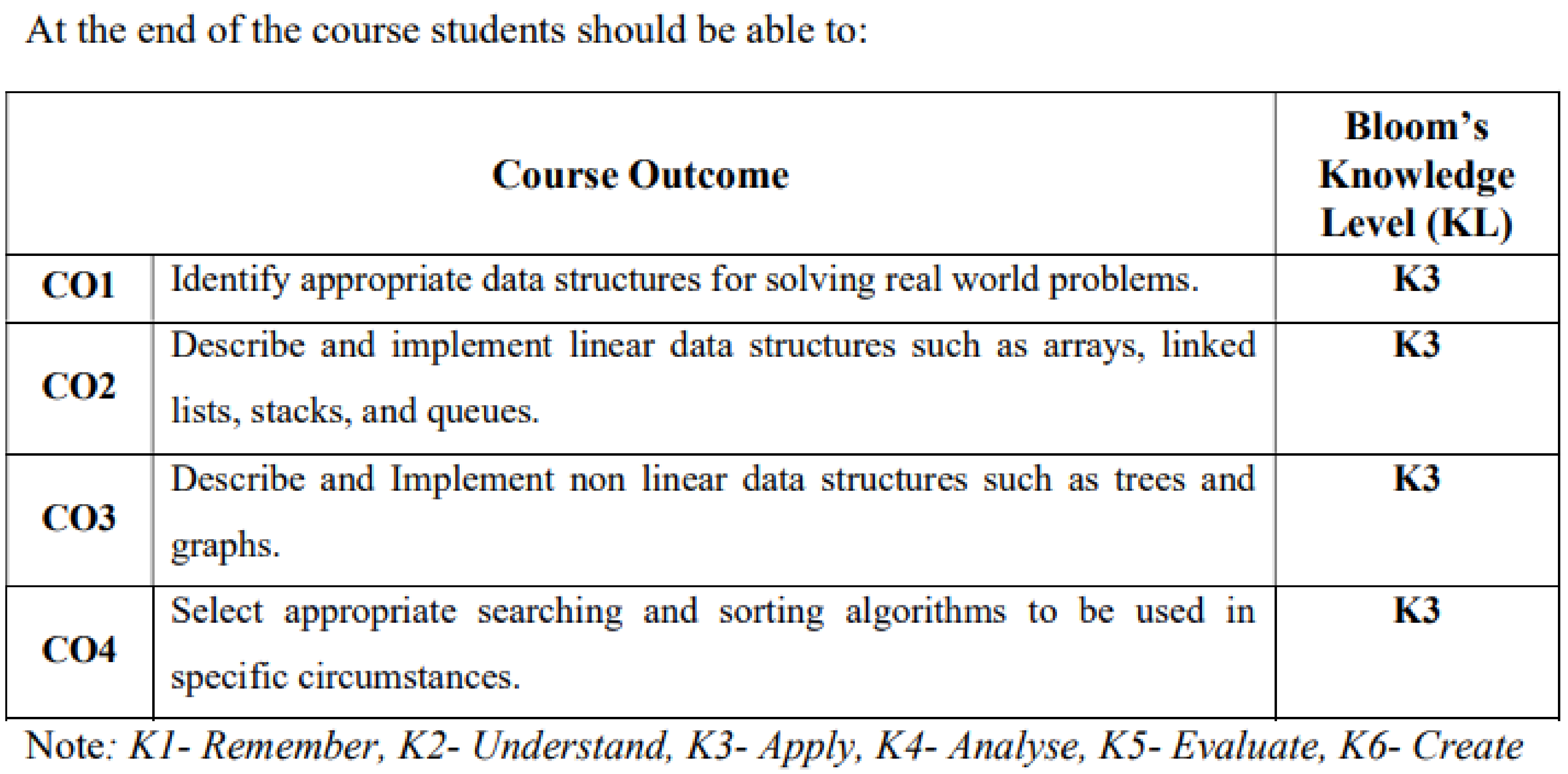
PCCST303DATA STRUCTURES AND ALGORITHMS

PREPARED BY SHARIKA T R, ASSISTANT PROFESSOR, ASIET

Course Outcome



At the end of the course students should be able to:

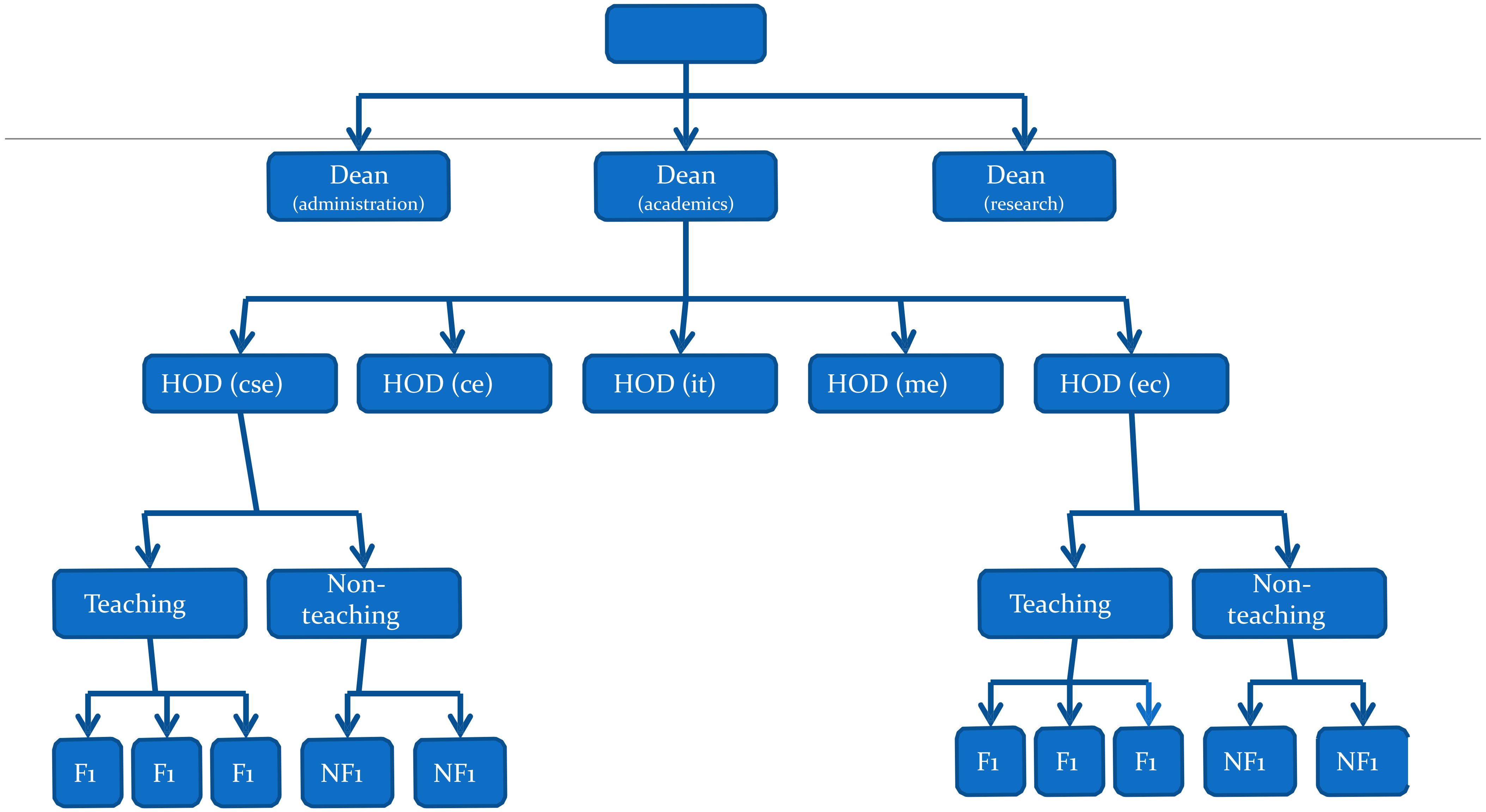
Course Outcome		Bloom's Knowledge Level (KL)
CO1	Identify appropriate data structures for solving real world problems.	**K3**
CO2	Describe and implement linear data structures such as arrays, linked lists, stacks, and queues.	**K3**
CO3	Describe and Implement non linear data structures such as trees and graphs.	**K3**
CO4	Select appropriate searching and sorting algorithms to be used in specific circumstances.	**K3**

Note: *K1- Remember, K2- Understand, K3- Apply, K4- Analyse, K5- Evaluate, K6- Create*

Module 3- Trees and Graphs

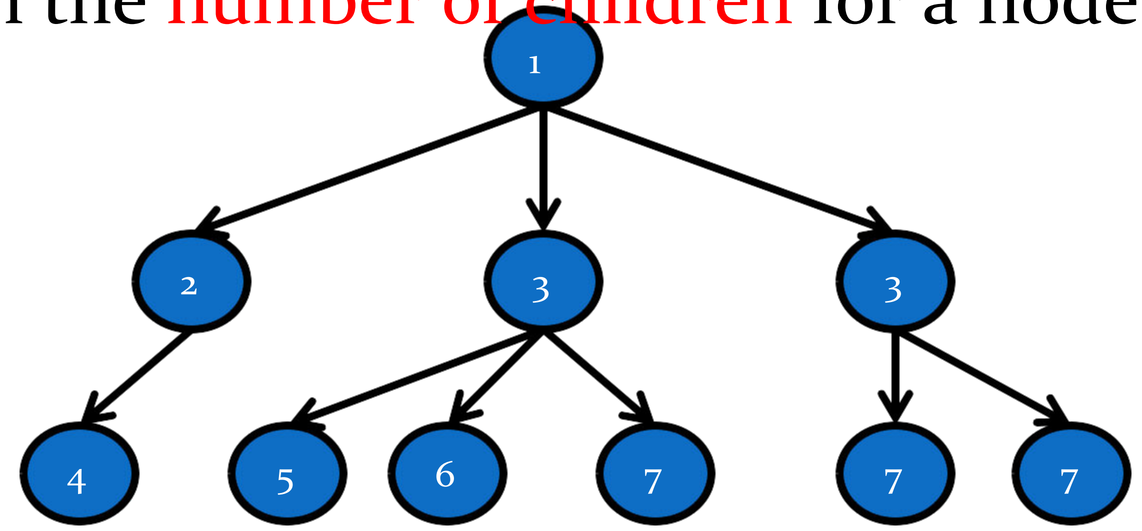
Trees :- Representation Of Trees; Binary Trees - Types and Properties, Binary Tree Representation, Tree Operations, Tree Traversals; Expression Trees; Binary Search Trees - Binary Search Tree Operations; Binary Heaps - Binary Heap Operations, Priority Queue.

Graphs :- Definitions; Representation of Graphs; Depth First Search and Breadth First Search; Applications of Graphs - Single Source All Destination.


Dean
(administration)
Dean
(academics)
Dean
(research)
HOD (cse)
HOD (ce)
HOD (it)
HOD (me)
HOD (ec)
Teaching
Non-teaching
F1
F1
F1
NF1
NF1
Teaching
Non-teaching
F1
F1
F1
NF1
NF1

TREE

- Finite collection of special data items called nodes
- Nodes of the tree are arranged in hierarchical structures to represent the parent-child relationship
- No restriction on the number of children for a node

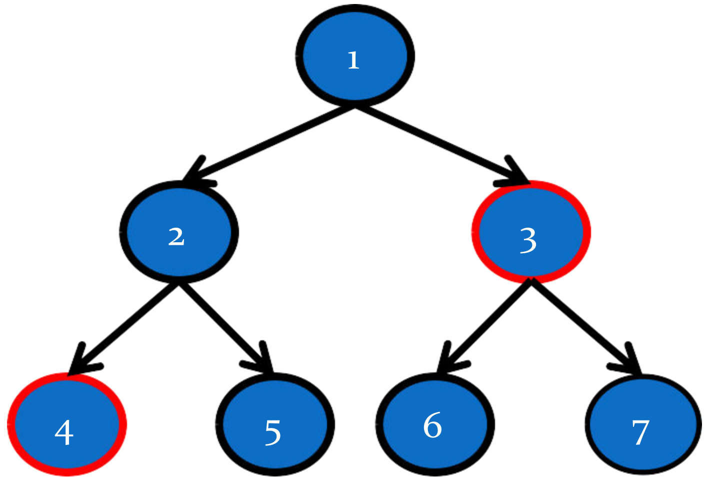


DEGREE of a node: Number of nodes connected to any node

IN-DEGREE of a node: Number of edges that ends at a particular node

OUT-DEGREE: Number of edges that starts from a particular node

Degree = in-degree+ out-degree

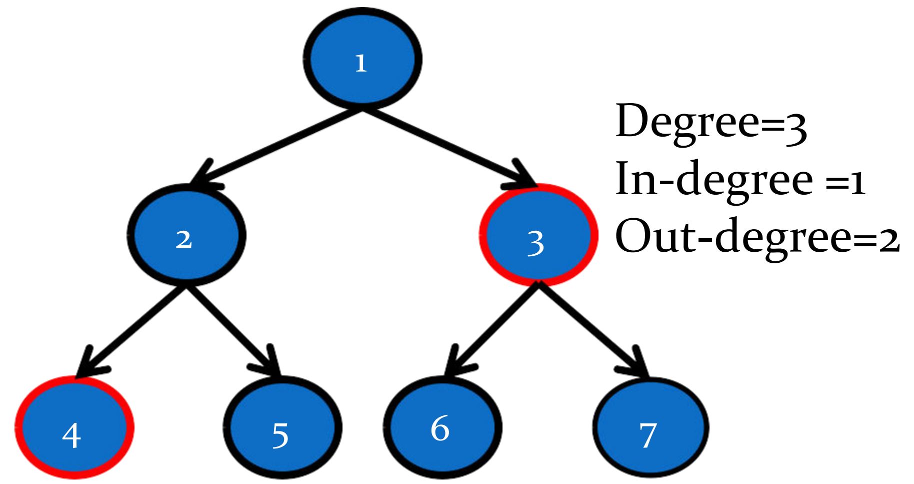


DEGREE of a node: Number of nodes connected to any node

IN-DEGREE of a node: Number of edges that ends at a particular node

OUT-DEGREE: Number of edges that starts from a particular node

Degree = in-degree+ out-degree



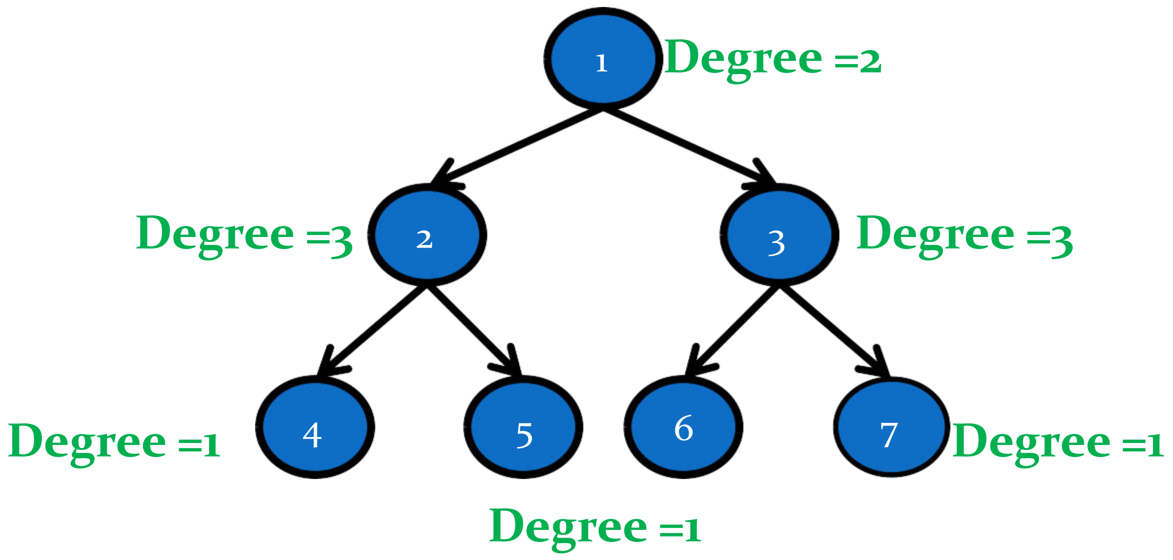
Degree=3
In-degree =1
Out-degree=2

Degree=1
In-degree =1
Out-degree=0

Terminologies

Degree of a tree:

Maximum degree of any node

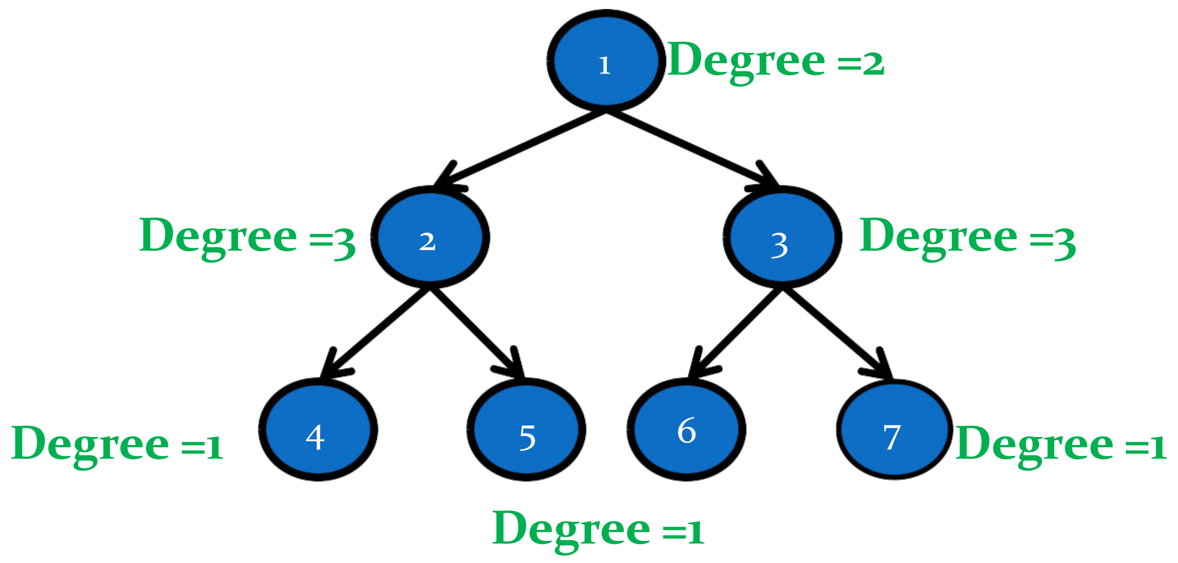


Terminologies

Degree of a tree:

Maximum degree of any node

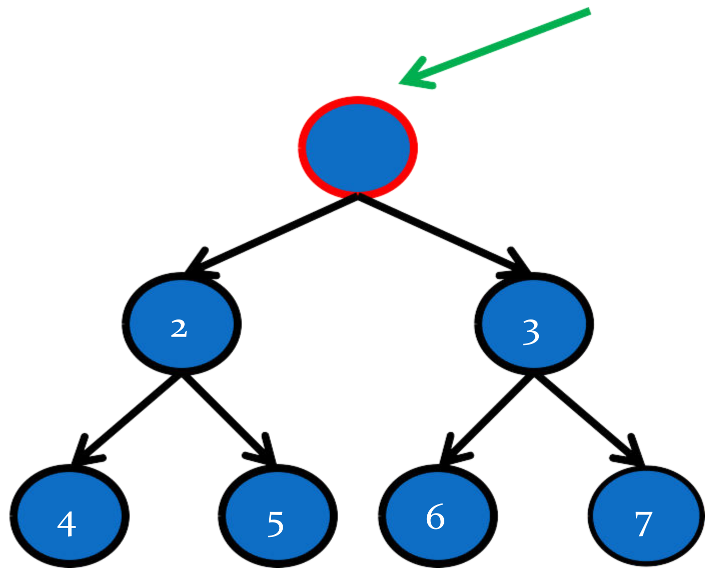
Degree of tree = 3



Terminologies

ROOT

First node which does not have any parent Node with In-degree =0



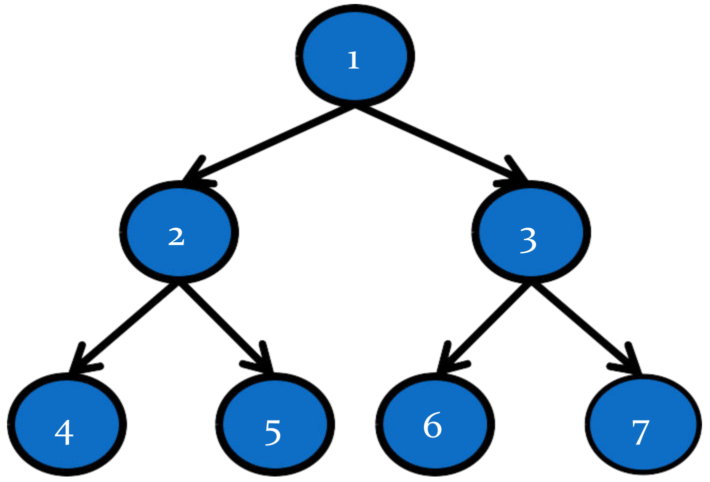
ROOT

Terminologies

LEAF NODE or TERMINAL NODE

Last node of the tree which does not have any descendants

Node with out-degree= 0

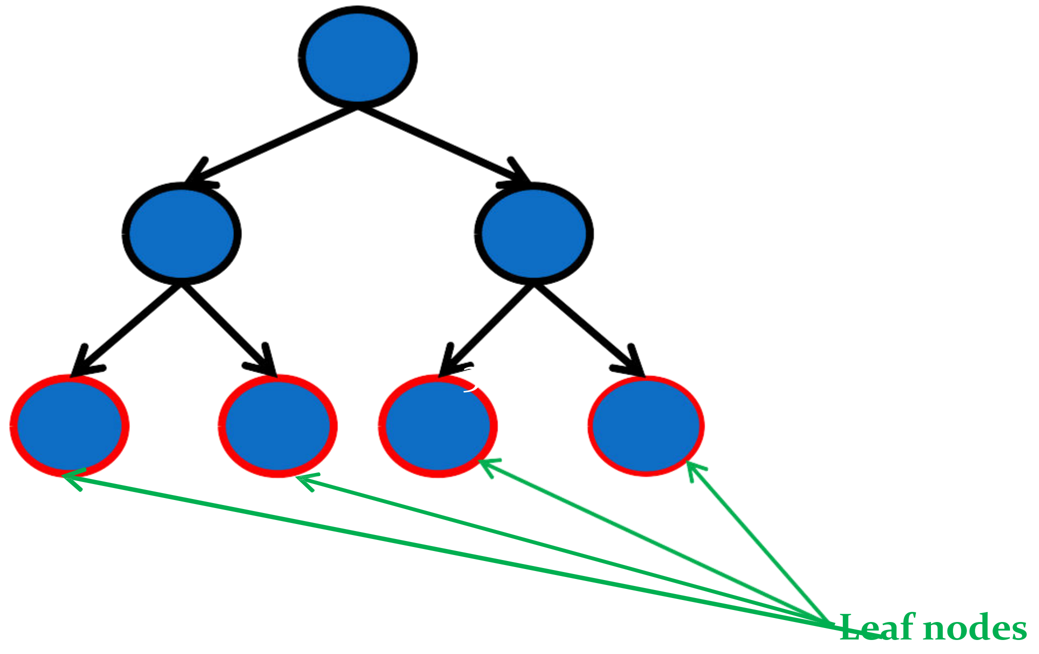


Terminologies

LEAF NODE or TERMINAL NODE

Last node of the tree which does not have any descendants

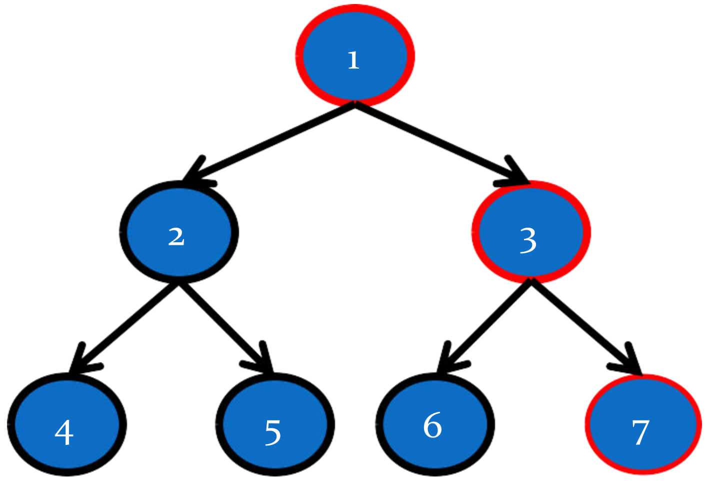
Node with out-degree= 0



Terminologies

PATH

Sequence of edges between any two nodes



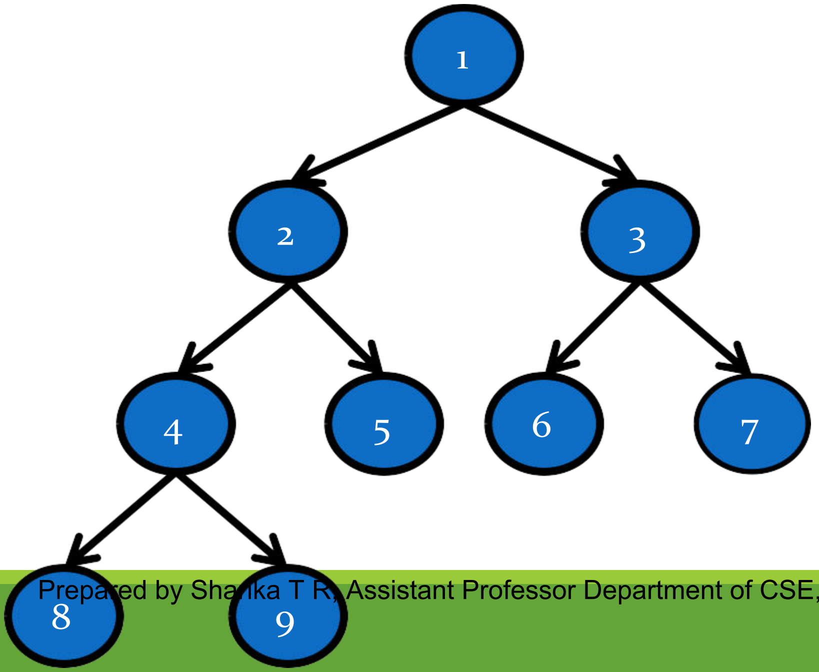
Path : 1-3, 3-7

Terminologies

LEVEL of a node

Root is always assigned a level zero

Level of any other node = Level of its parent +1

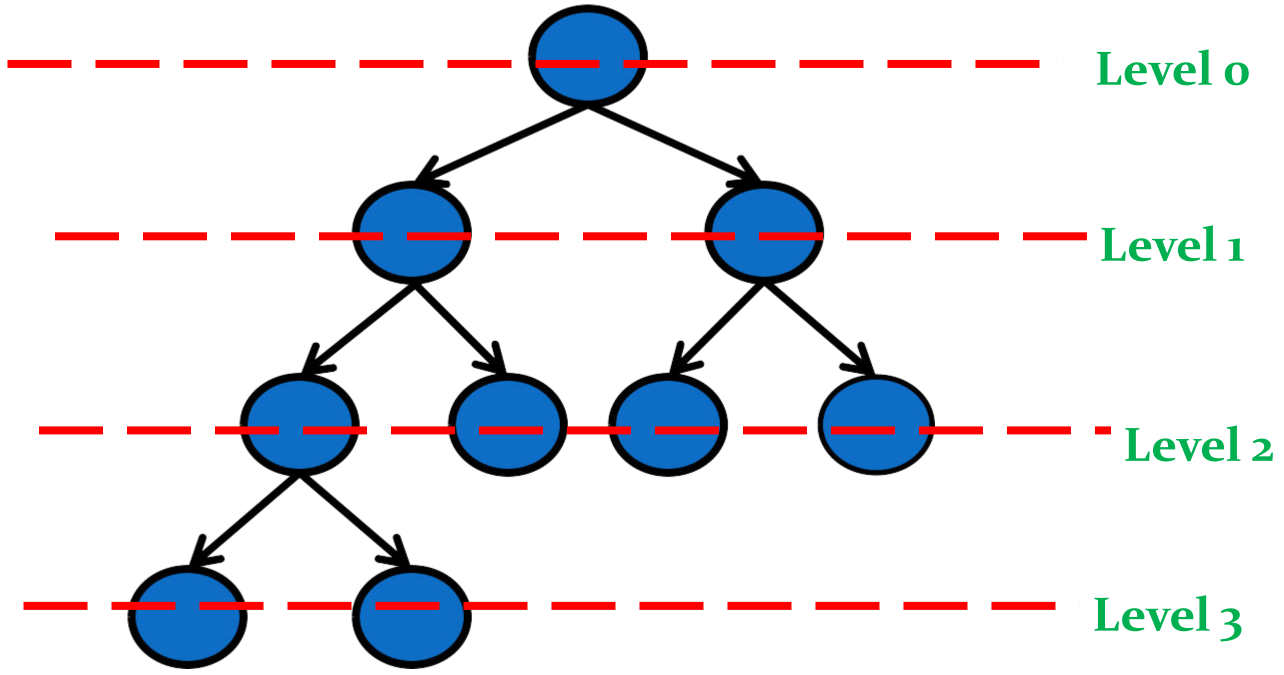


Terminologies

LEVEL of a node

Root is always assigned a level zero

Level of any other node = Level of its parent +1



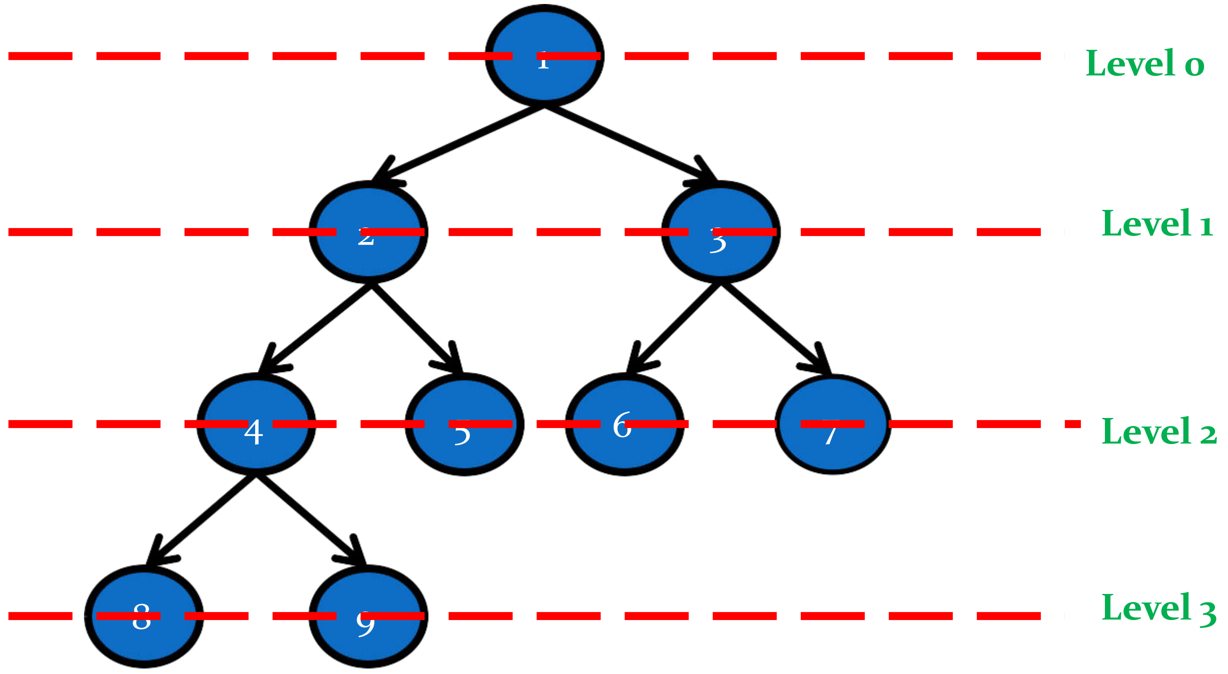
Terminologies

DEPTH of a node

No. of nodes in the longest path from the root to any terminal node

Depth of a node = level of a node +1

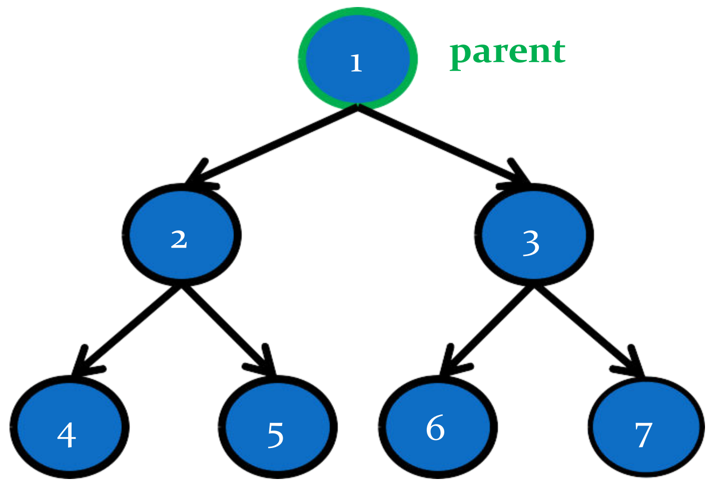
Depth of a tree = largest level of a tree +1



Terminologies

Siblings

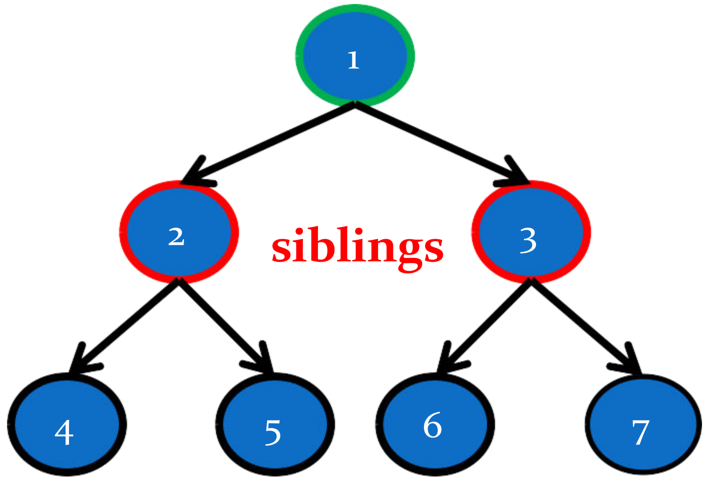
Children of same parent node



Terminologies

Siblings

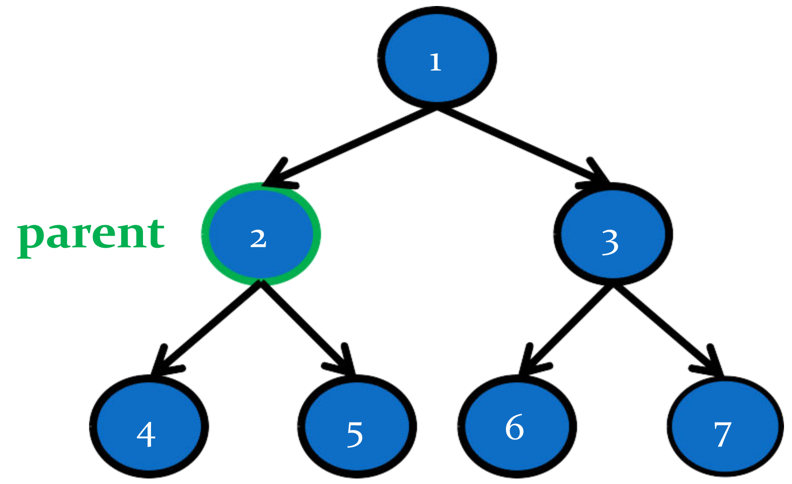
Children of same parent node



Terminologies

Siblings

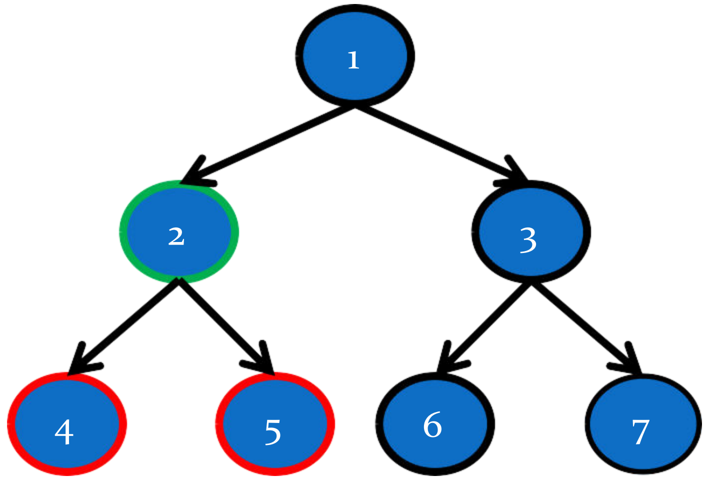
Children of same parent node



Terminologies

Siblings

Children of same parent node



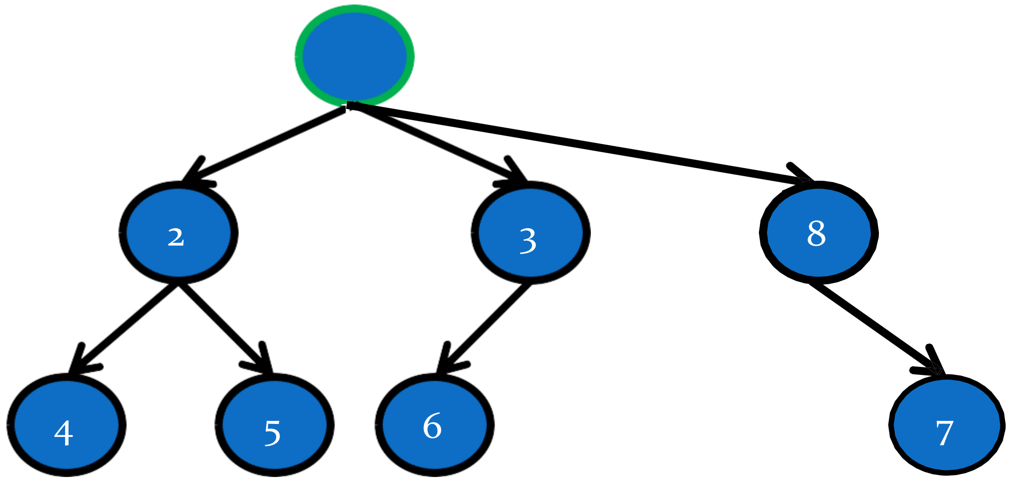
siblings

TREE: Definition

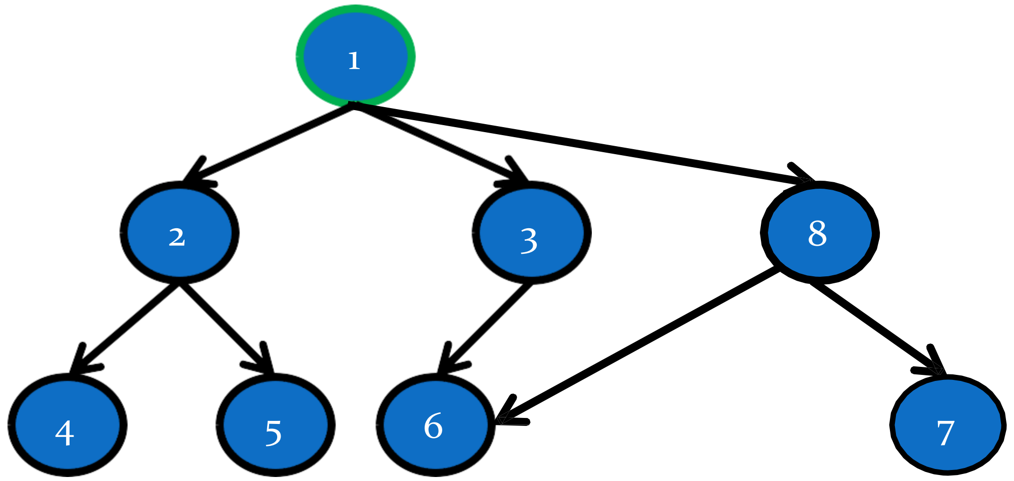
- A tree is a finite set of one or more nodes such that
  1. There is a specially designated node called the root
  2. The remaining nodes are partitioned into n>=0 disjoint sets T1,

T2, ...Tn where each of these sets is a tree

T1, T2, ...Tn are called the subtrees of the root



Is this a tree ??

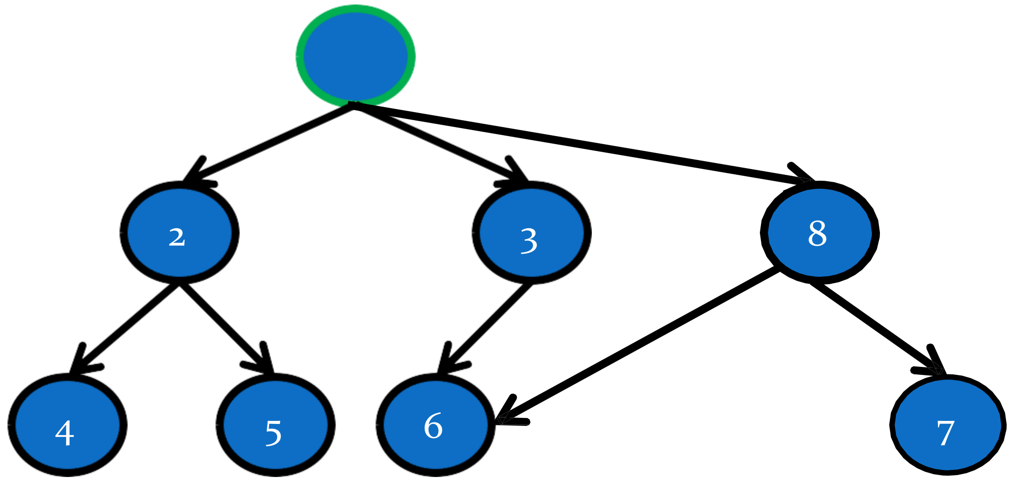


Is this a tree ??

- A tree is a finite set of one or more nodes such that
    1. There is a specially designated node called the root
    2. The remaining nodes are partitioned into n>=0 disjoint sets T1,

T2, ...Tn where each of these sets is a tree

T1, T2, ...Tn are called the subtrees of the root

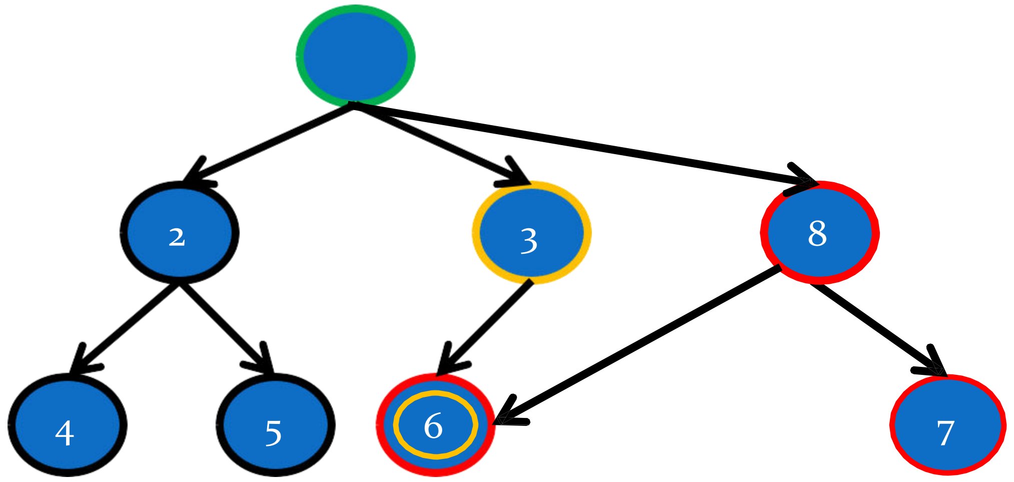


Is this a tree ?? - NO

- A tree is a finite set of one or more nodes such that
  1. There is a specially designated node called the root
  2. The remaining nodes are partitioned into n>=0 disjoint sets T1,

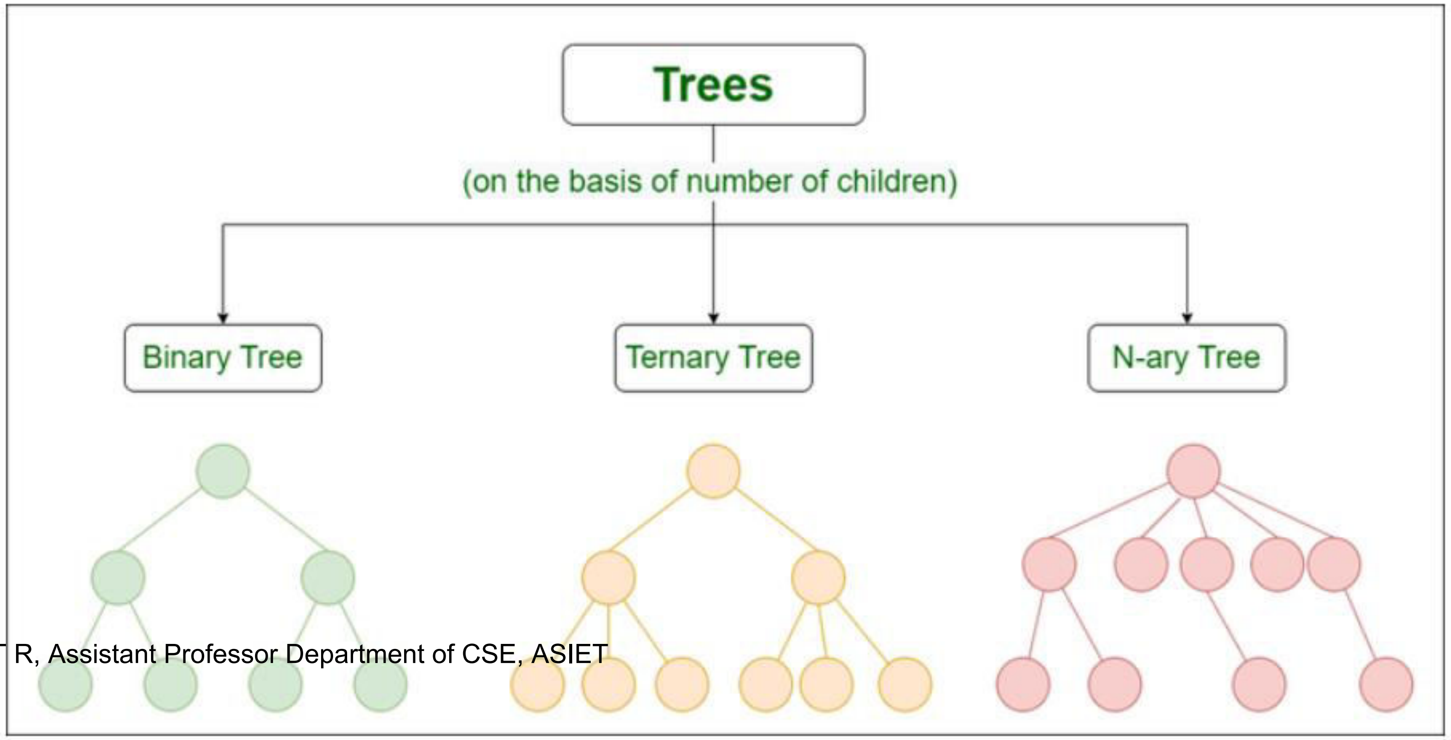
T2, ...Tn where each of these sets is a tree

T1, T2, ...Tn are called the subtrees of the root



Types of Tree data structures:

- **Binary tree**: In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
- **Ternary Tree**: A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
- **N-ary Tree or Generic Tree**: Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

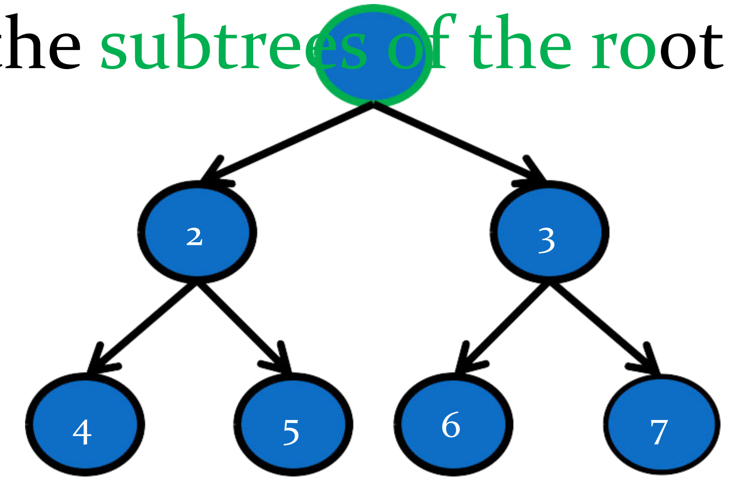


BINARY TREE: Definition

- A binary tree is a finite set of one or more nodes such that

1. There is a specially designated node called the root
2. The remaining nodes are partitioned into N=0,1 or 2 disjoint sets T1 and T2 where each of these sets is a tree

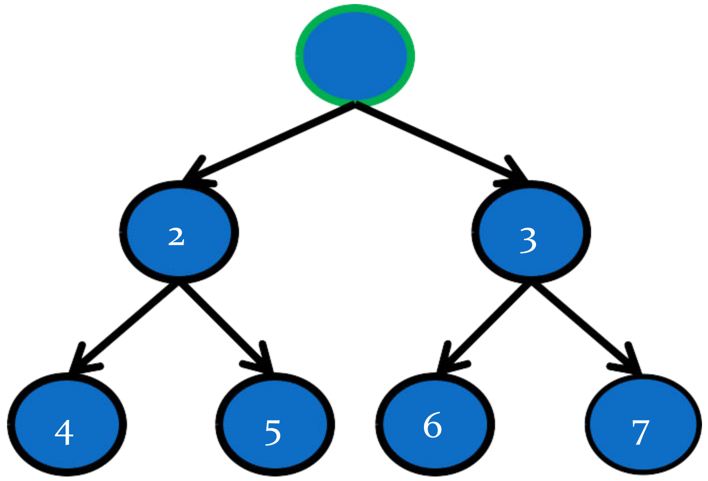
T1 and T2 are called the subtrees of the root



Terminologies

Binary TREE

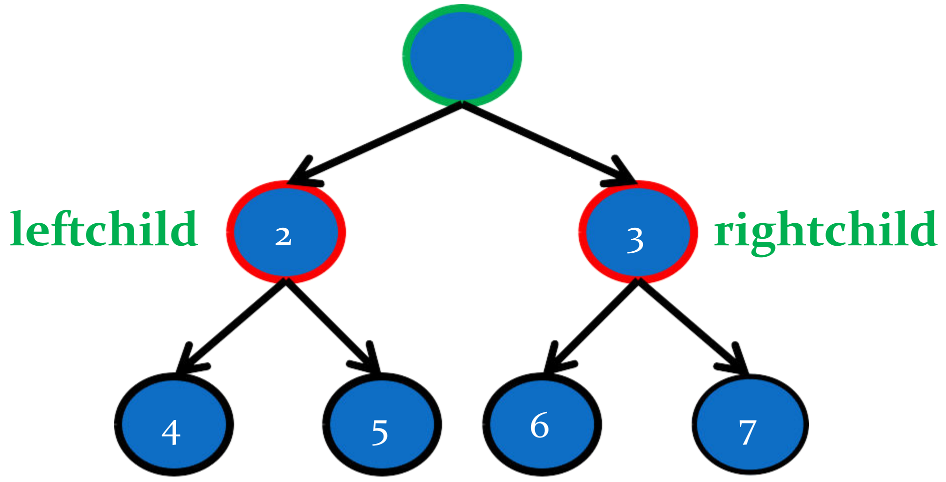
- a tree which is either empty or each node can have maximum 2 children
- each node can have either 0 children, 1 child or 2 children
- Leftchild - Node on the left of the parent
- Rightchild - Node on the right of the parent



Terminologies

Binary TREE

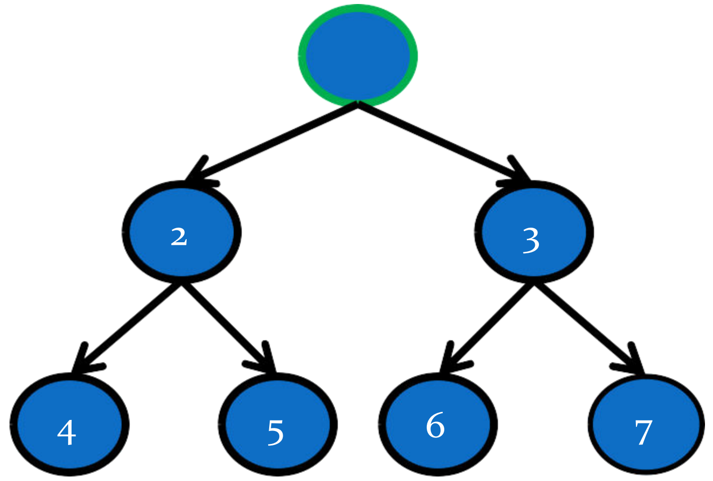
- a tree which is either empty or each node can have maximum 2 children
- each node can have either 0 children, 1 child or 2 children
- Leftchild - Node on the left of the parent
- Rightchild - Node on the right of the parent



Terminologies

Binary TREE

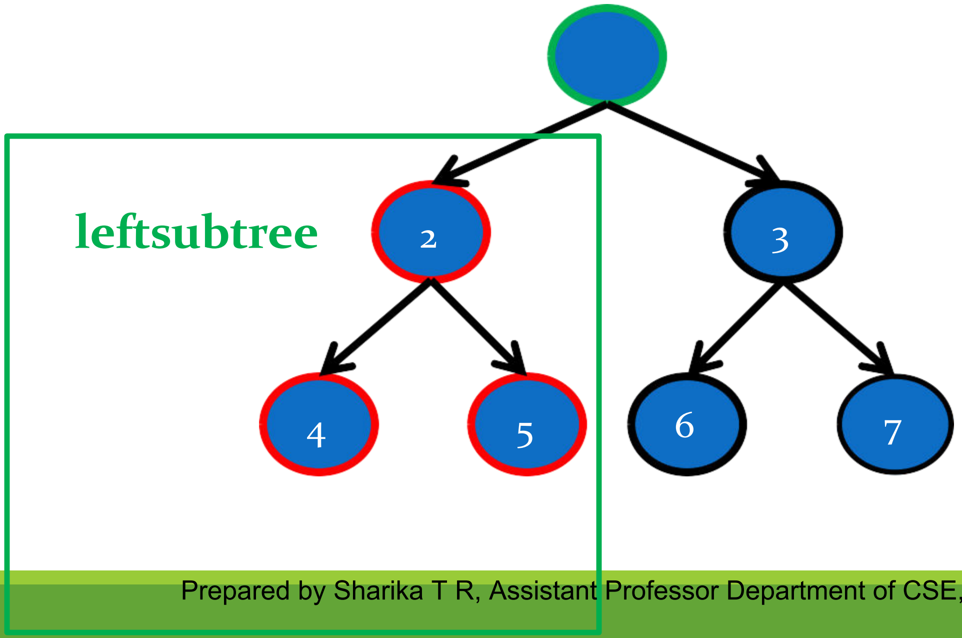
- Left**subtree** - Tree on the left of the parent
- Right**subtree** - Tree on the right of the parent



Terminologies

Binary TREE

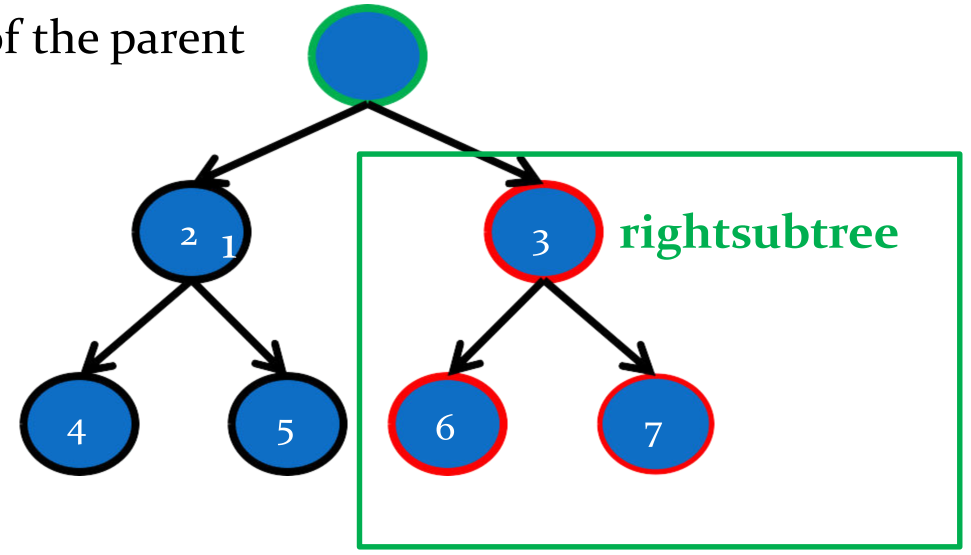
- a tree which is either empty or each node can have maximum 2 children
- each node can have either 0 children, 1 child or 2 children
- Leftsubtree - Tree on the left of the parent
- Rightsubtree - Tree on the right of the parent



Terminologies

Binary TREE

- a tree which is either empty or each node can have maximum 2 children
- each node can have either 0 children, 1 child or 2 children
- Leftsubtree - Tree on the left of the parent
- Rightsubtree - Tree on the right of the parent

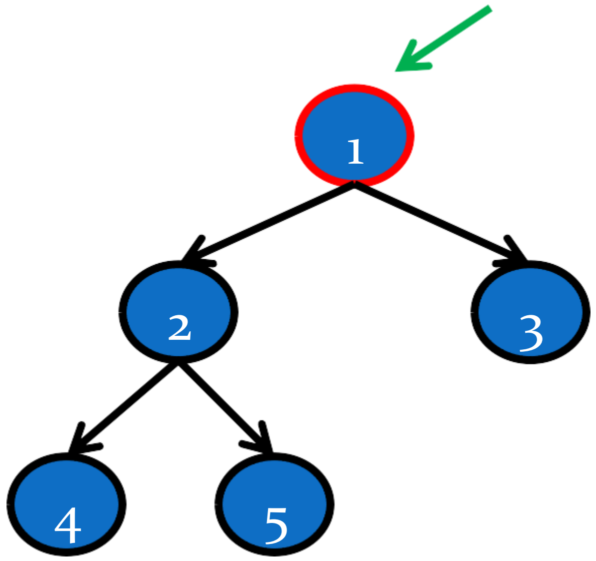


CLASSIFICATION OF BINARY TREE

1. **Strictly binary tree**

A binary tree in which every nonleaf node has non empty left and right sub tree A strictly binary tree with N leaves always contains 2N-1 nodes

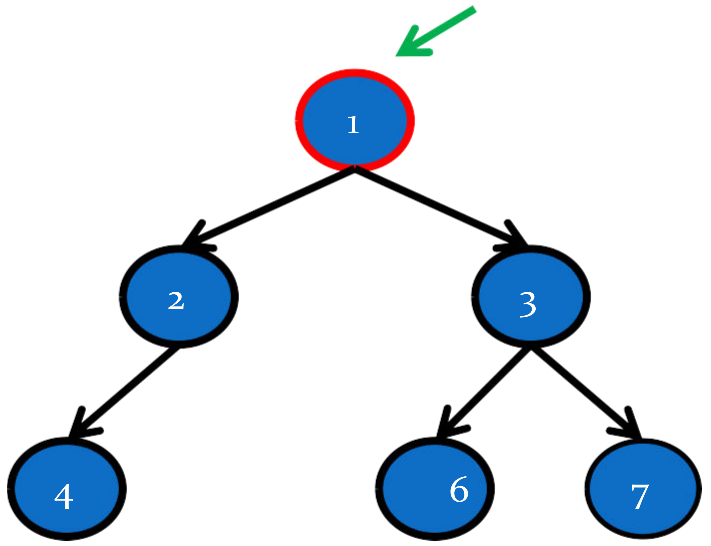
A strict binary tree, also known as a full or proper binary tree, is a type of binary tree where each node has either zero or two children.



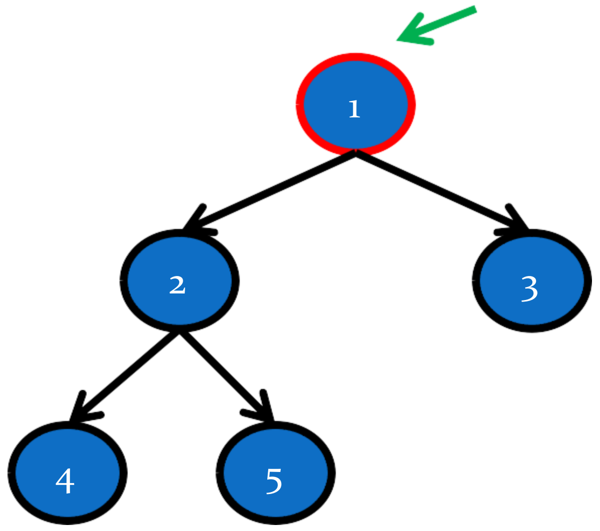
CLASSIFICATION OF BINARY TREE

2. **Complete binary tree**

- A binary tree with all levels except the last level contains the maximum number of possible nodes and all nodes in the last level appear as left as possible



Incomplete

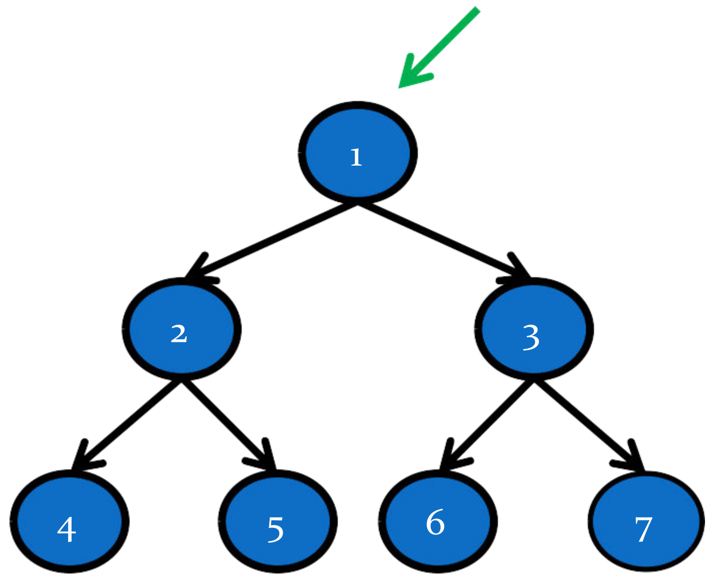


complete

A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. I

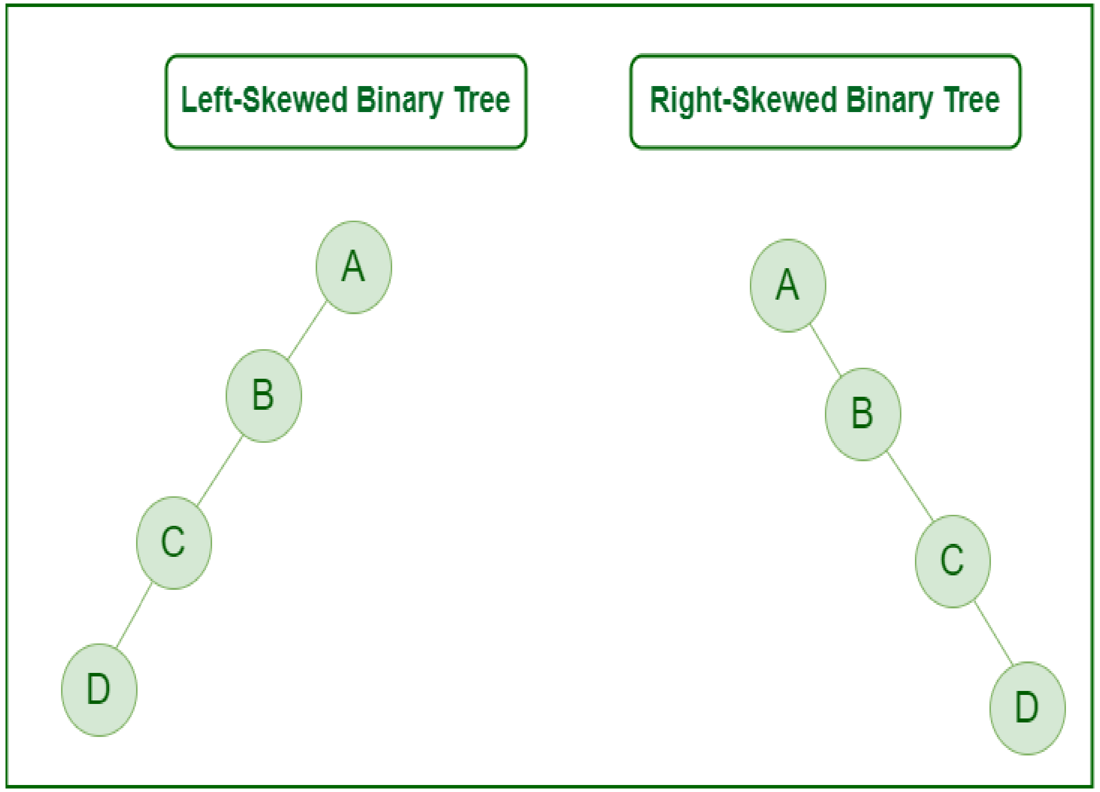
CLASSIFICATION OF BINARY TREE

3. **Full binary tree**
   - A binary tree that contains maximum possible number of nodes at all levels
   - A tree of depth d will have 2^d-1 nodes



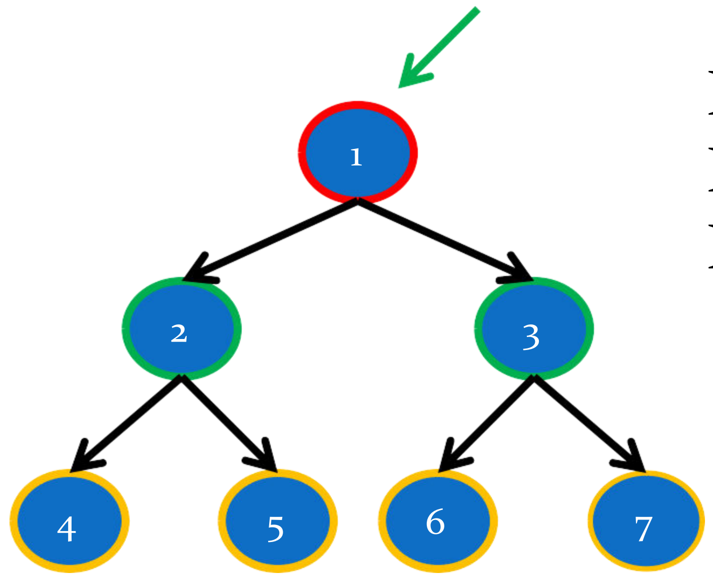
D=3
Nodes = $2^3-1 = 7$

4. Skewed Binary Trees



PROPERTIES OF BINARY TREE

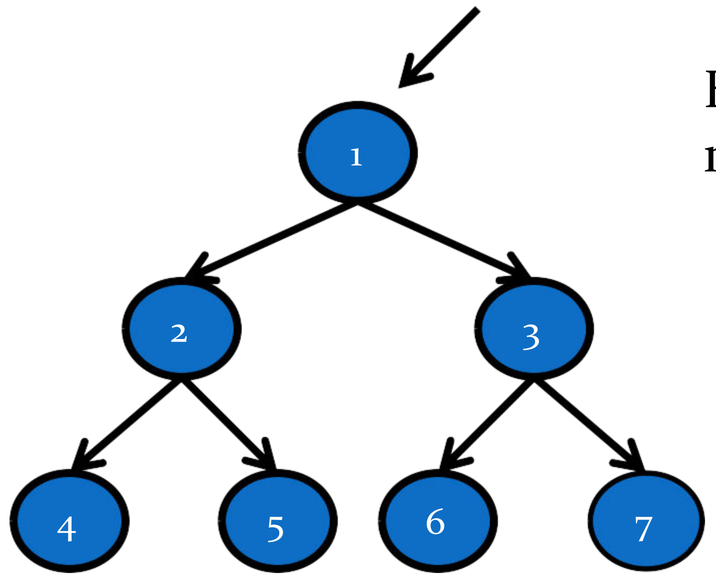
1. Maximum number of nodes on level l is 2^l, where l>=0



L=0, nodes = $2^0=1$
L=1, nodes = $2^1=2$
L=2, nodes = $2^2=4$

PROPERTIES OF BINARY TREE

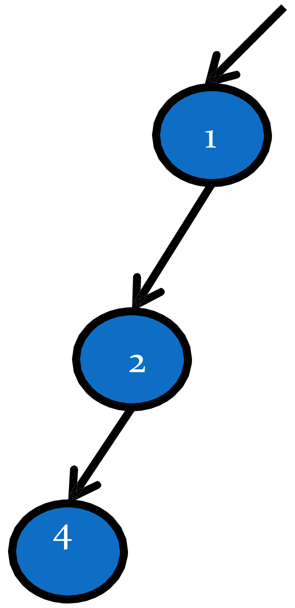
2. Maximum number of nodes possible in a binary tree of height h is 2^h-1.



H=3
nodes= $2^3-1 = 7$

PROPERTIES OF BINARY TREE

3. Minimum number of nodes possible in a binary tree of height h is h.



H=3
nodes= 3

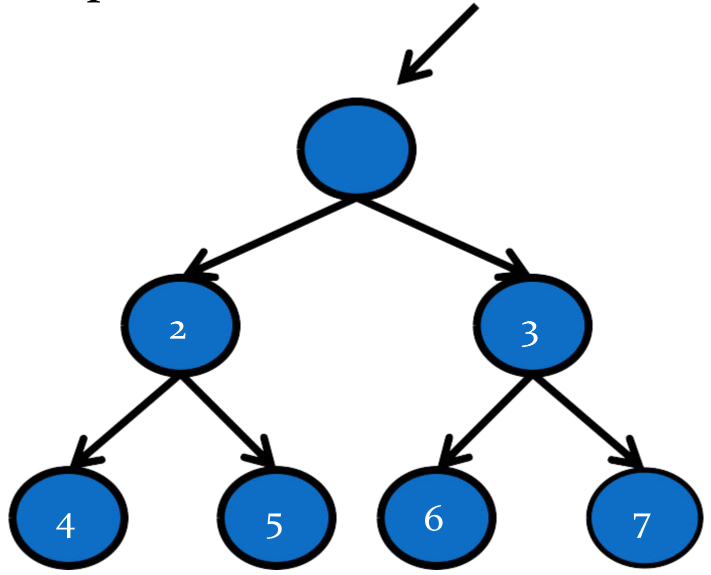
REPRESENTATIONS OF BINARY TREE

1. Linear or sequential representation
2. Linked or pointer representation

REPRESENTATIIONS OF BINARY TREE

1. Linear or sequential representation

- A block of memory for an array is allocated before storing the actual tree in it.
- Once memory is allocated, the size of the tree is restricted as permitted by the memory.
- In this representation, the nodes are stored level by level, starting from the zero level where only the root node is present.



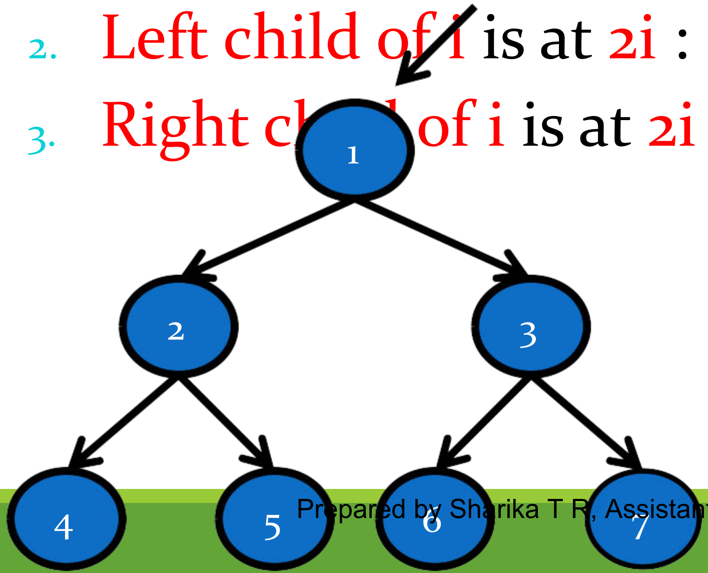
A[1]	1
A[2]	2
A[3]	3
A[4]	4
A[5]	5
A[6]	6
A[7]	7

REPRESENTATIIONS OF BINARY TREE

Linear or sequential representation

Following rules can be used to decide the location of any node

1. The root node is at index 1
2. For any node with index I,
   1. Parent (i) is at i /2
   2. Left child of i is at 2i :
   3. Right child of i is at 2i + 1:



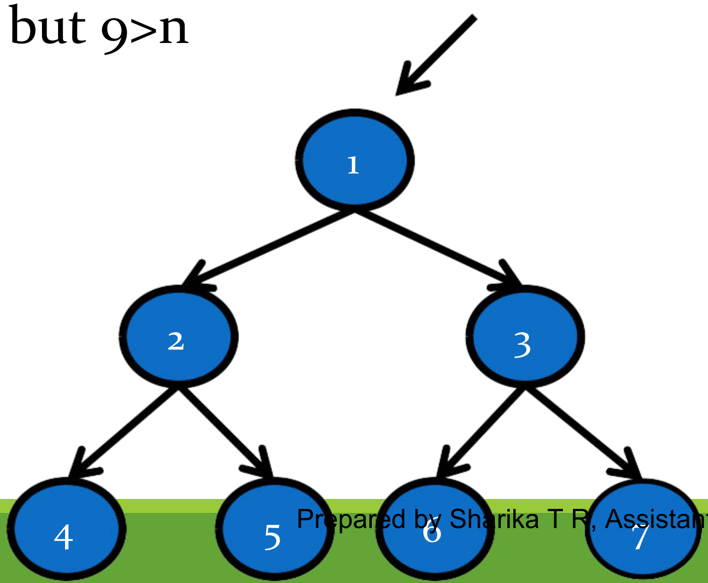
A[1]	1
A[2]	2
A[3]	3
A[4]	4
A[5]	5
A[6]	6
A[7]	7

REPRESENTATIIONS OF BINARY TREE

Linear or sequential representation

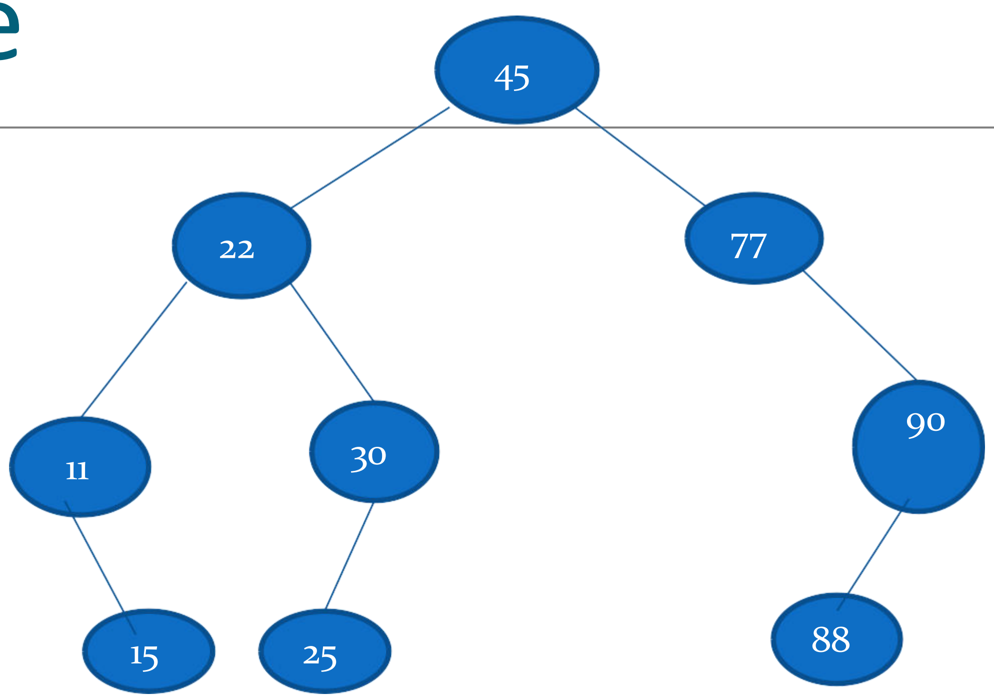
Left child of i is at 2i :

- If 2i > n , then the node does not have a left child.
- Eg: i=4 , then 2i = 8, but 8>n
- no left child Right child of i is at 2i +1:
- If 2i + 1 > n , then the node does have a right child.
- Eg: i=4 , then 2i+1 = 9, but 9>n
- no right child



A[1]	1
A[2]	2
A[3]	3
A[4]	4
A[5]	5
A[6]	6
A[7]	7

Sequential representation: Example



1	2	3	4	5	6	7	8	9	10	11	12	13	14
45	22	77	11	30		9 0		15	25				88

REPRESENTATIONS OF BINARY TREE

Linear or sequential representation

Advantages of linear representation

- Any node can be accessed by calculating the index
- For each node, only data is stored – no need to store any pointers

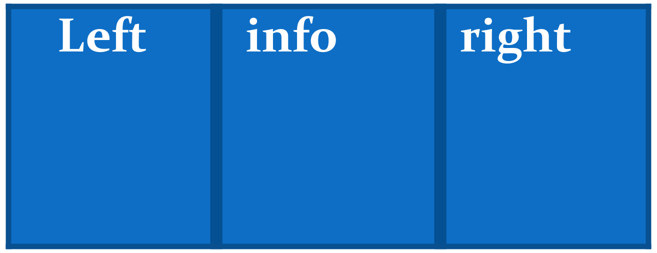
Disadvantages

- Other than for full binary tree, majority of the array entries may be empty
- No way to increase the size of the tree during execution
- Inserting or deleting a node is inefficient

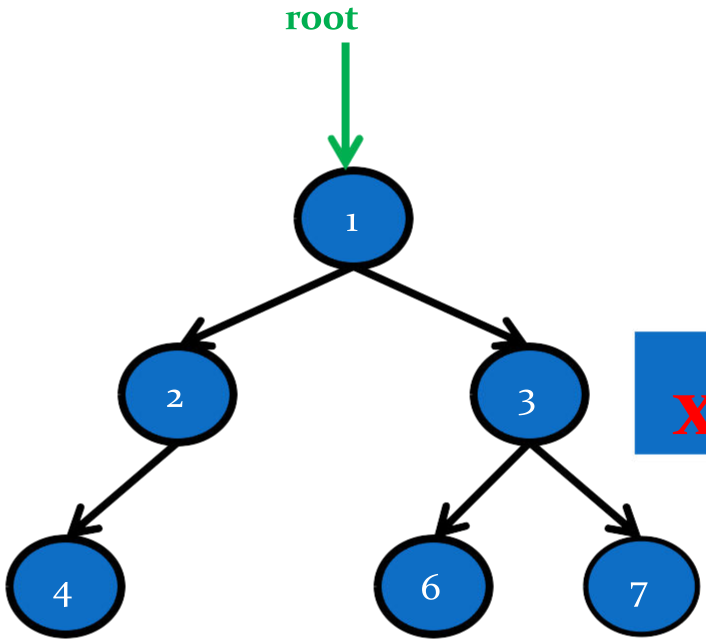
REPRESENTATIONS OF BINARY TREE

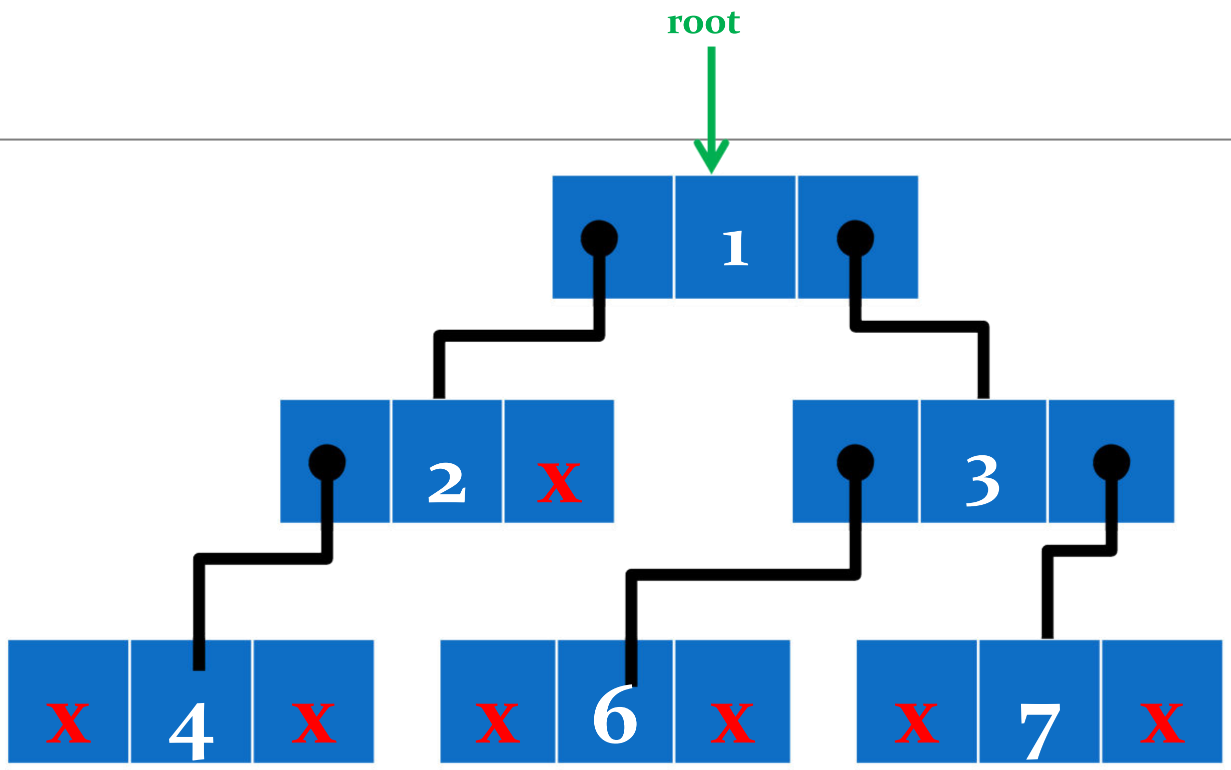
2. Linked or pointer representation

- Tree can also be defined as a finite collection of nodes where each node is divided into 3 parts containing
  - Left child address
  - Information (data)
  - Right child address



BINARY TREE





Operations on binary tree

- Insertion as a leaf node
- Deletion of a leaf node
- traversal

Insertion as a leaf node

```
1. l =Search(k)
  - Search(k) function:
    - Returns the index of the node with key k if found,
otherwise returns -1.
2. If l == -1:
    - Print "Search is unsuccessful"
  Else:
    a. Read option to insert as left child (LC) or right
child (RC).
    b. If option == RC:
        - If a[(2 * l) + 1] == NULL:
            - a[(2 * l) + 1] = x
        - Else:
            - Print "Insertion not possible"
        - EndIf
    c. Else if option == LC:
        - If a[(2 * l)] == NULL:
            - a[(2 * l)] = x
        - Else:
            - Print "Insertion not possible"
        - EndIf
    d. EndIf
  EndIf
```

Search a key

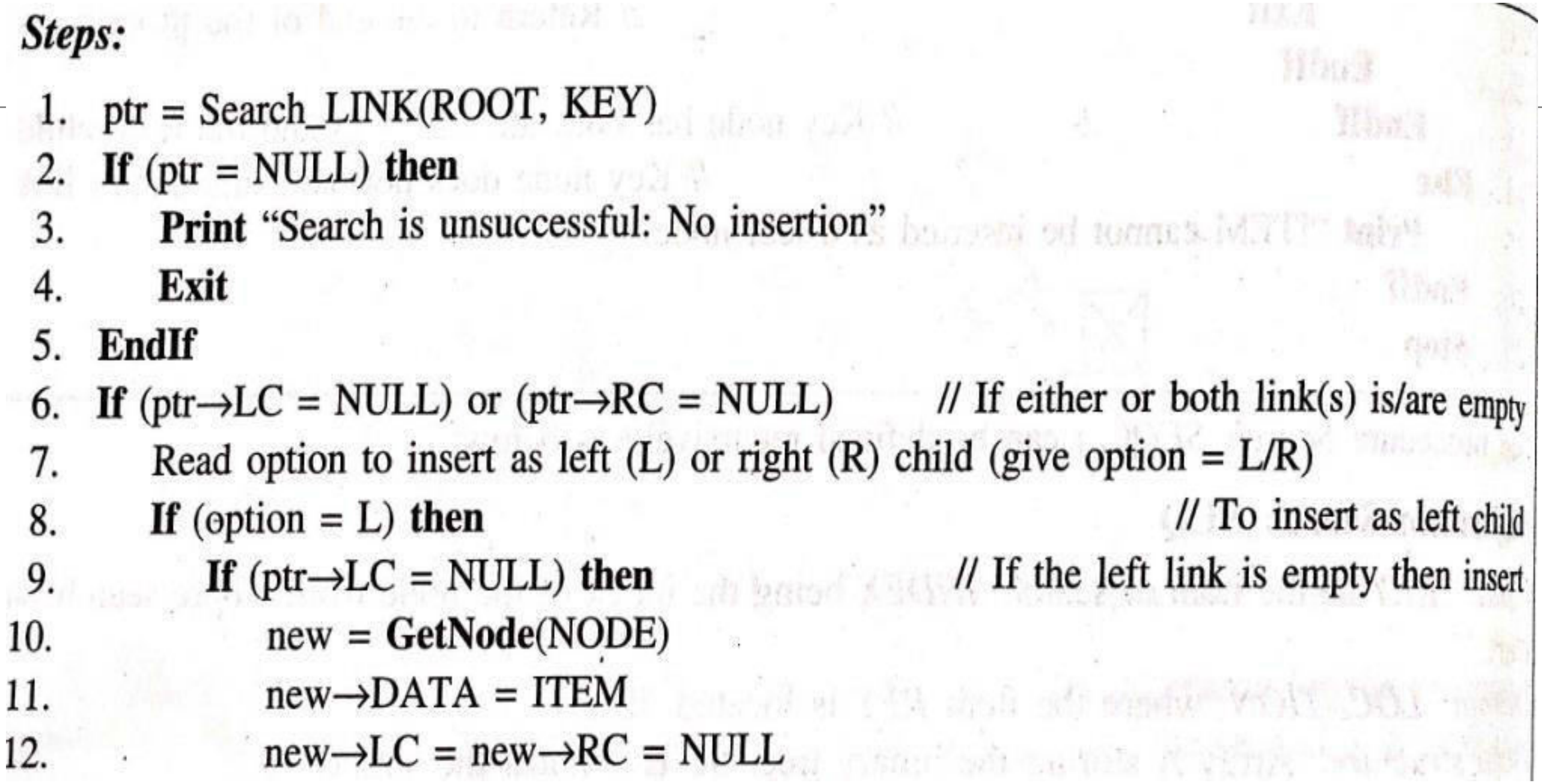
<u>search(int k)</u>

```
For     i=1 to size
   if a[i]==k
      return i
   end if
End for
return -1
```

Deletion of a leaf node

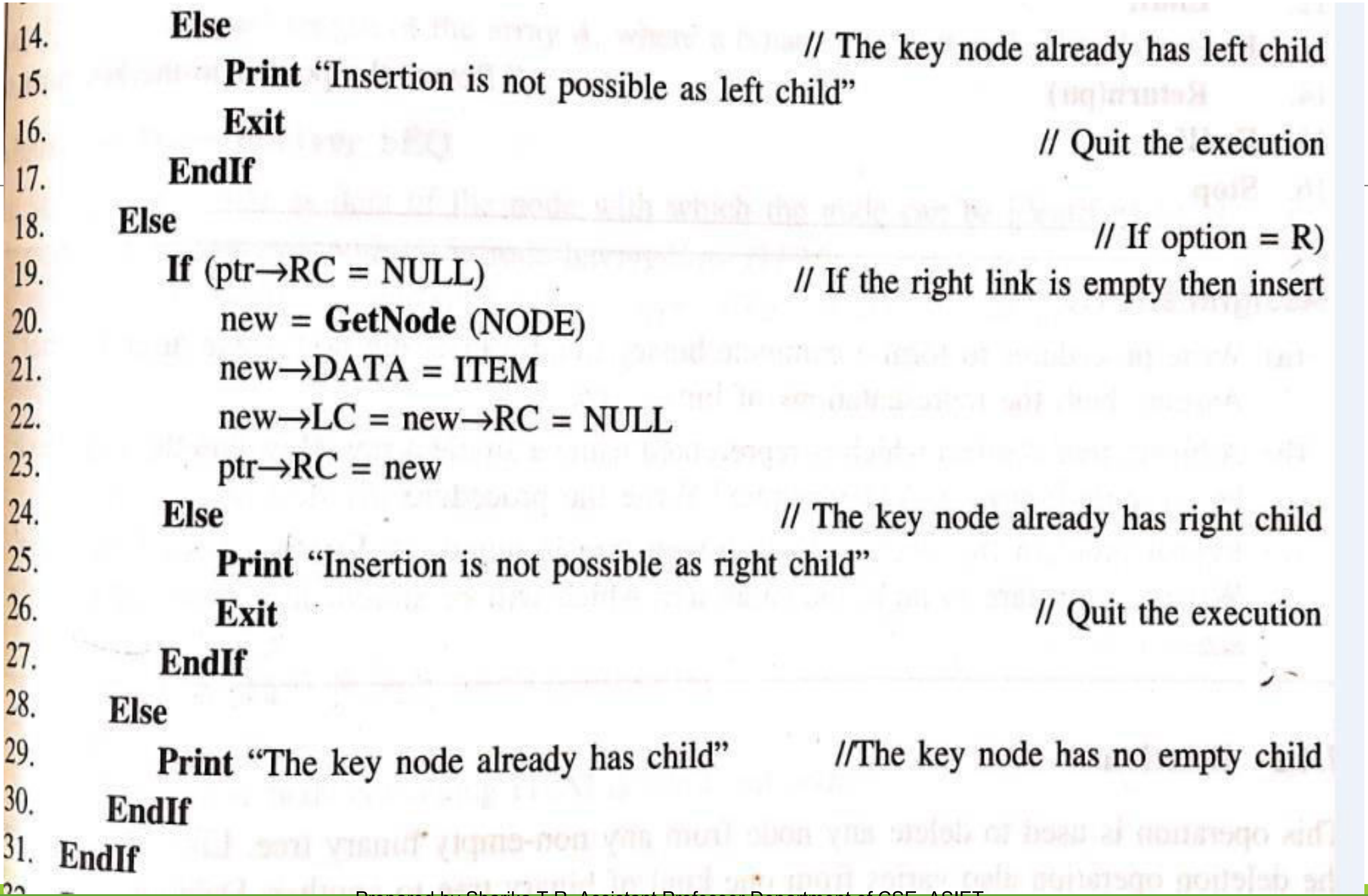
```
del(int x)
l=search(x)
If  l!=-1
   if a[(2*l)]==null && a[(2*l)+1]==null
          a[l] is deleted
          a[l]=null
   else
           print item is not a leaf node
   end if
else
           print item does not exist
End if
```

Insertion of leaf node in a linked list representation



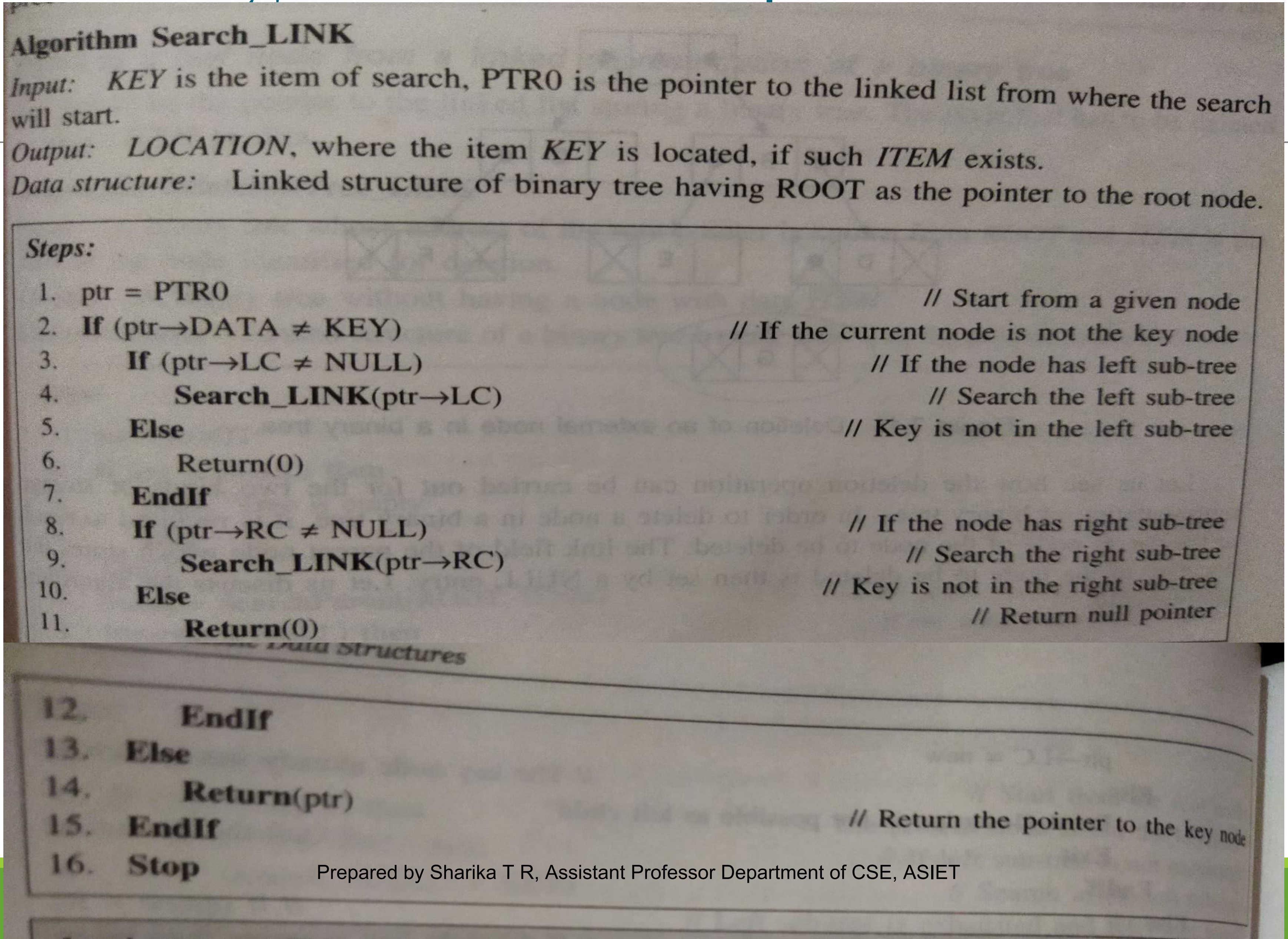
Steps:

```
1.  ptr = Search_LINK(ROOT, KEY)
2.  If (ptr = NULL) then
3.      Print "Search is unsuccessful: No insertion"
4.      Exit
5.  EndIf
6.  If (ptr→LC = NULL) or (ptr→RC = NULL)        // If either or both link(s) is/are empty
7.      Read option to insert as left (L) or right (R) child (give option = L/R)
8.      If (option = L) then                                    // To insert as left child
9.          If (ptr→LC = NULL) then                 // If the left link is empty then insert
10.             new = GetNode(NODE)
11.             new→DATA = ITEM
12.             new→LC = new→RC = NULL
```



```
14.         Else                           // The key node already has left child
15.             Print "Insertion is not possible as left child"
16.             Exit                                       // Quit the execution
17.         EndIf
18.     Else                                                  // If option = R)
19.         If (ptr→RC = NULL)            // If the right link is empty then insert
20.             new = GetNode (NODE)
21.             new→DATA = ITEM
22.             new→LC = new→RC = NULL
23.             ptr→RC = new
24.         Else                          // The key node already has right child
25.             Print "Insertion is not possible as right child"
26.             Exit                                       // Quit the execution
27.         EndIf
28.     Else
29.         Print "The key node already has child"   //The key node has no empty child
30.     EndIf
31. EndIf
```

Searching in a linked list representation



Algorithm Search_LINK

Input: *KEY* is the item of search, PTR0 is the pointer to the linked list from where the search will start.

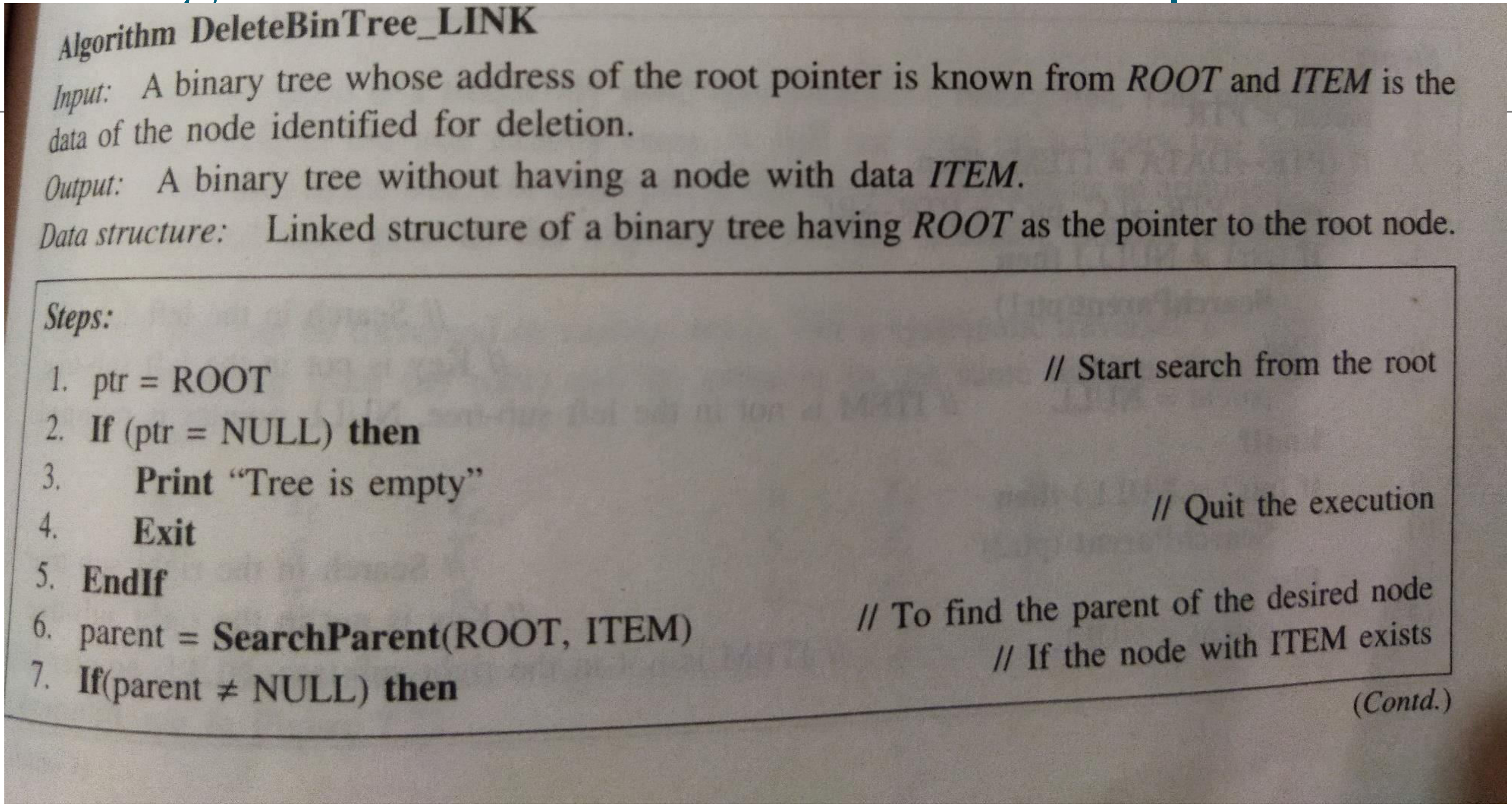
Output: *LOCATION*, where the item *KEY* is located, if such *ITEM* exists.

Data structure: Linked structure of binary tree having ROOT as the pointer to the root node.

Steps:

```
1.  ptr = PTR0                              // Start from a given node
2.  If (ptr→DATA ≠ KEY)                     // If the current node is not the key node
3.      If (ptr→LC ≠ NULL)                  // If the node has left sub-tree
4.          Search_LINK(ptr→LC)             // Search the left sub-tree
5.      Else                                // Key is not in the left sub-tree
6.          Return(0)
7.      EndIf
8.      If (ptr→RC ≠ NULL)                  // If the node has right sub-tree
9.          Search_LINK(ptr→RC)             // Search the right sub-tree
10.     Else                                // Key is not in the right sub-tree
11.         Return(0)                       // Return null pointer
12.     EndIf
13. Else
14.     Return(ptr)                         // Return the pointer to the key node
15. EndIf
16. Stop
```

Deleting a leaf node from a linked list representation



Algorithm DeleteBinTree_LINK

Input: A binary tree whose address of the root pointer is known from *ROOT* and *ITEM* is the data of the node identified for deletion.

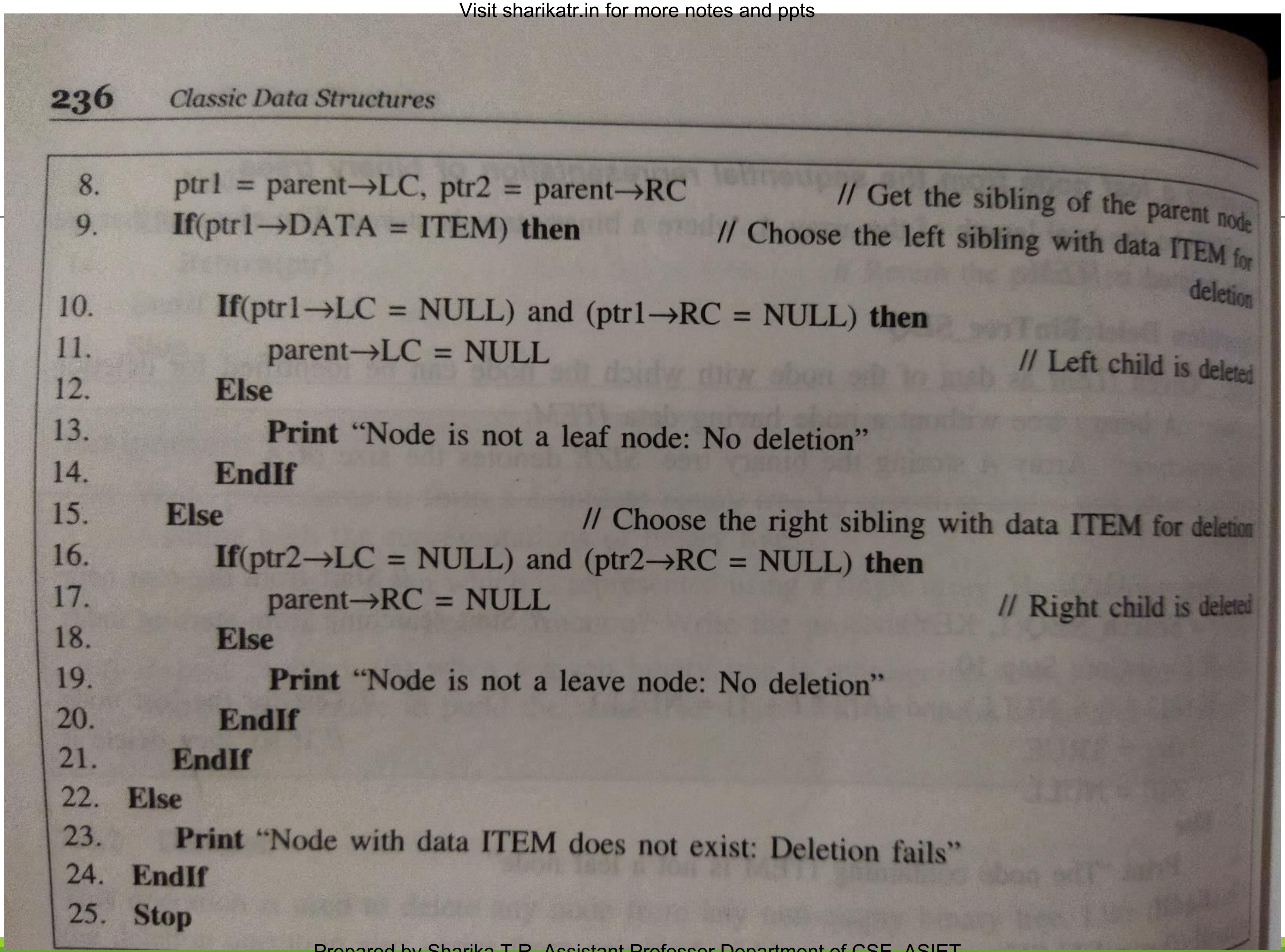
Output: A binary tree without having a node with data *ITEM*.

Data structure: Linked structure of a binary tree having *ROOT* as the pointer to the root node.

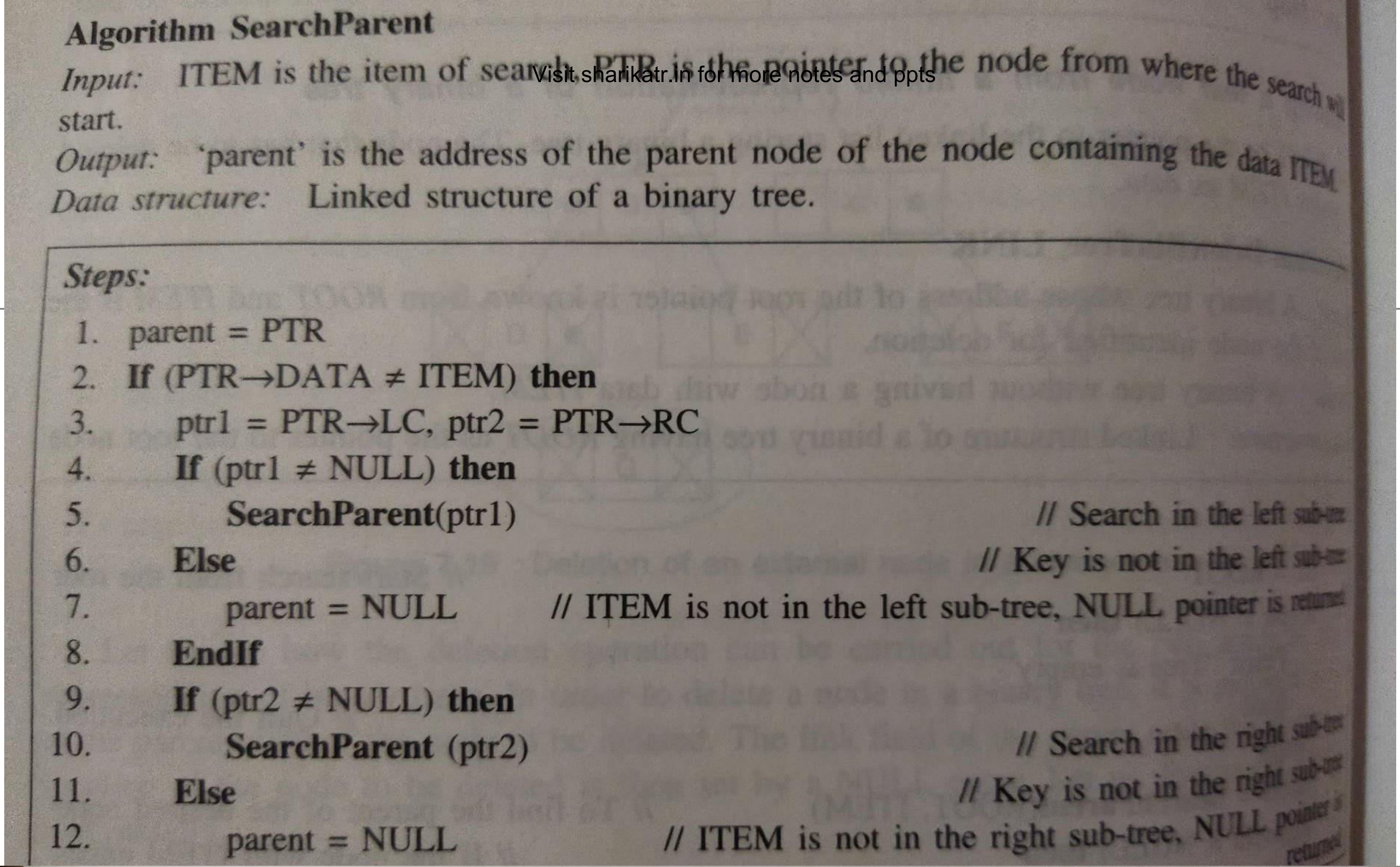
Steps:

```
1.  ptr = ROOT                                        // Start search from the root
2.  If (ptr = NULL) then
3.      Print "Tree is empty"
4.      Exit                                          // Quit the execution
5.  EndIf
6.  parent = SearchParent(ROOT, ITEM)                 // To find the parent of the desired node
7.  If(parent ≠ NULL) then                            // If the node with ITEM exists
```

(Contd.)



```
8.      ptr1 = parent→LC, ptr2 = parent→RC             // Get the sibling of the parent node
9.      If(ptr1→DATA = ITEM) then                      // Choose the left sibling with data ITEM for
                                                          deletion
10.        If(ptr1→LC = NULL) and (ptr1→RC = NULL) then
11.            parent→LC = NULL                          // Left child is deleted
12.        Else
13.            Print "Node is not a leaf node: No deletion"
14.        EndIf
15.     Else                                  // Choose the right sibling with data ITEM for deletion
16.        If(ptr2→LC = NULL) and (ptr2→RC = NULL) then
17.            parent→RC = NULL                          // Right child is deleted
18.        Else
19.            Print "Node is not a leave node: No deletion"
20.        EndIf
21.     EndIf
22. Else
23.     Print "Node with data ITEM does not exist: Deletion fails"
24. EndIf
25. Stop
```



Algorithm SearchParent

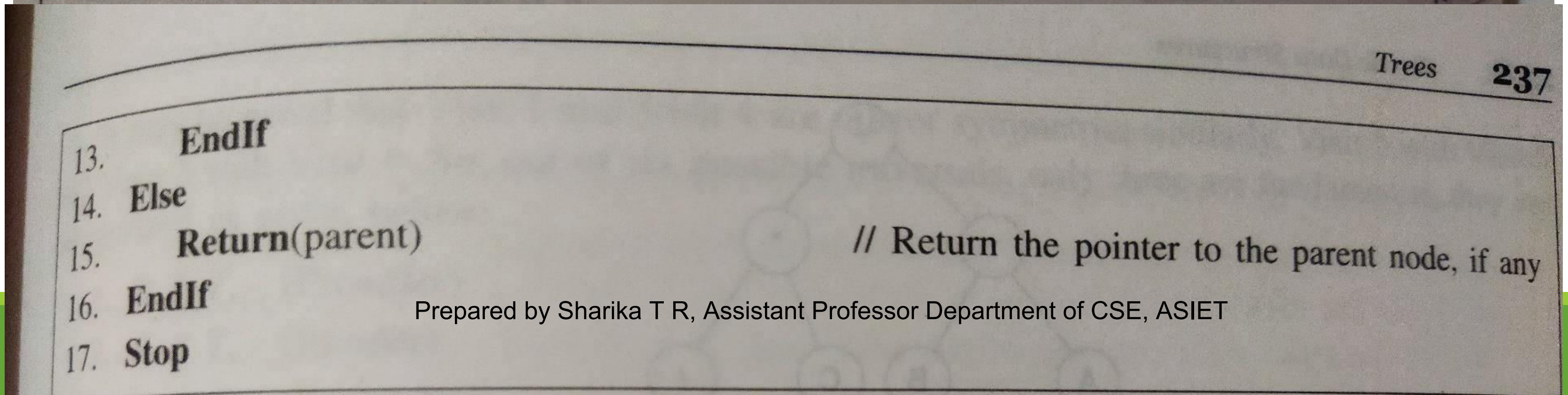
Input: ITEM is the item of search, PTR is the pointer to the node from where the search will start.

Output: 'parent' is the address of the parent node of the node containing the data ITEM.

Data structure: Linked structure of a binary tree.

Steps:

```
1.  parent = PTR
2.  If (PTR→DATA ≠ ITEM) then
3.      ptr1 = PTR→LC, ptr2 = PTR→RC
4.      If (ptr1 ≠ NULL) then
5.          SearchParent(ptr1)                    // Search in the left sub-tree
6.      Else                                      // Key is not in the left sub-tree
7.          parent = NULL     // ITEM is not in the left sub-tree, NULL pointer is returned
8.      EndIf
9.      If (ptr2 ≠ NULL) then
10.         SearchParent (ptr2)                   // Search in the right sub-tree
11.     Else                                      // Key is not in the right sub-tree
12.         parent = NULL     // ITEM is not in the right sub-tree, NULL pointer is returned
```



```
13.     EndIf
14. Else
15.     Return(parent)          // Return the pointer to the parent node, if any
16. EndIf
17. Stop
```

Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

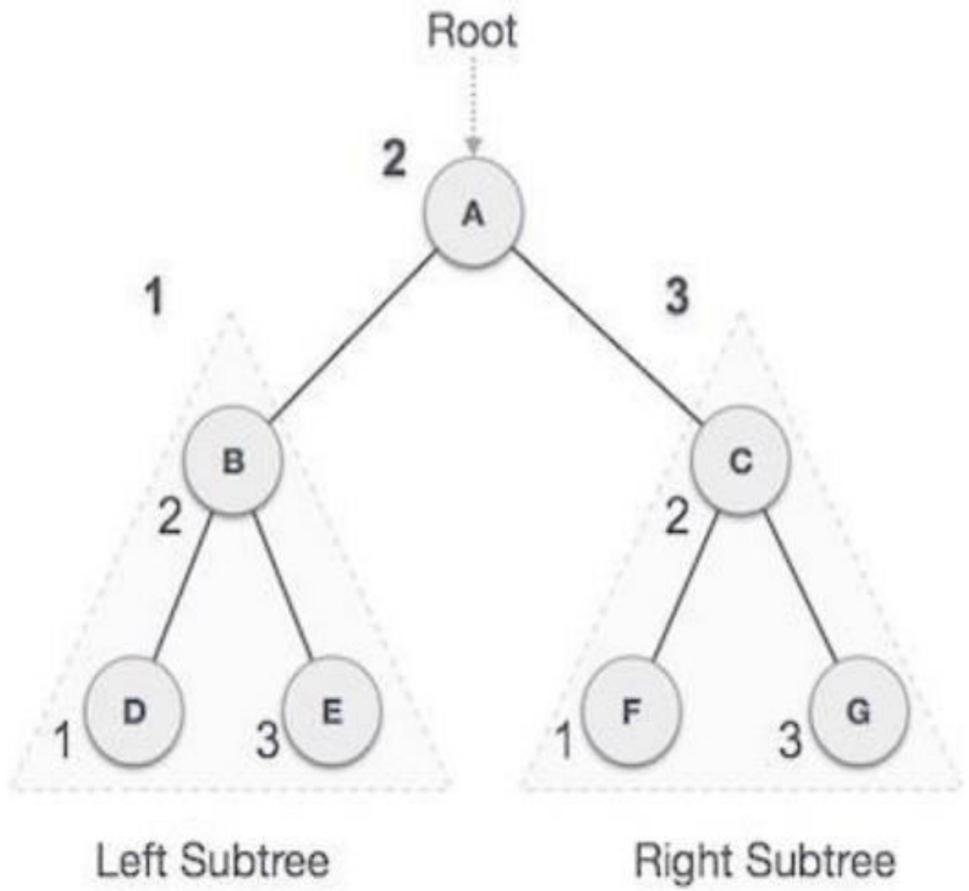
In-order Traversal : In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.

D → B → E → A → F → C → G

Algorithm

Until all nodes are traversed –

Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

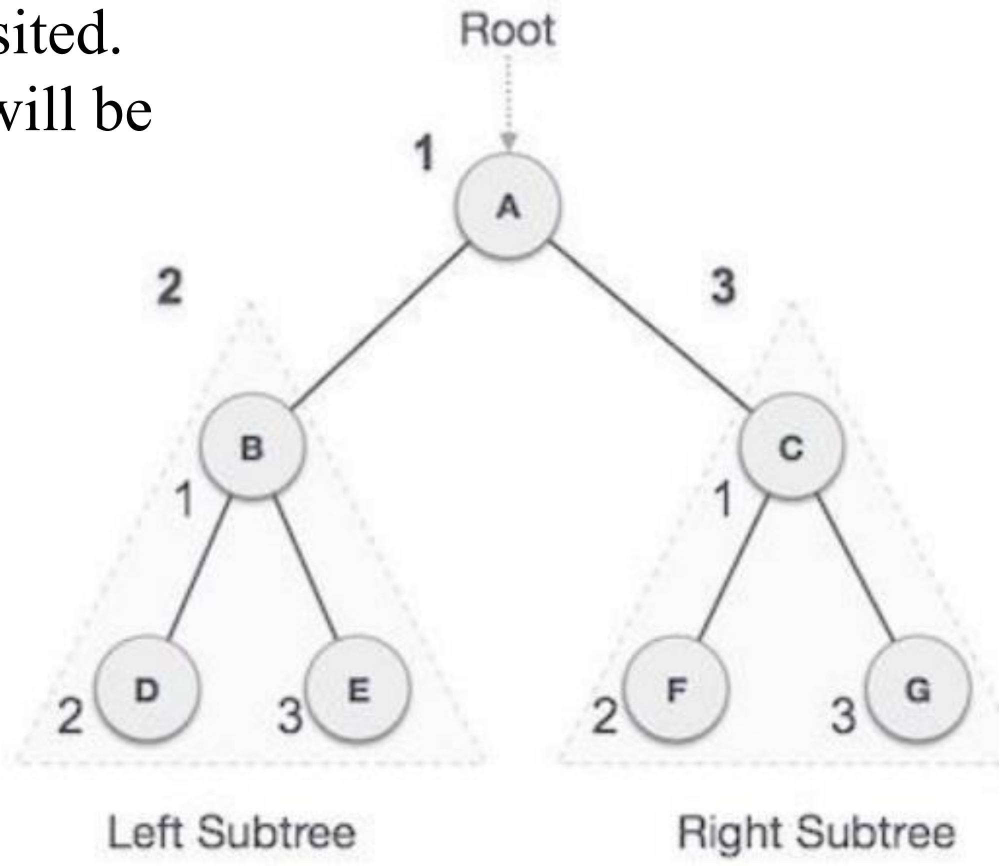


Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited.
The output of pre-order traversal of this tree will be

− A → B → D → E → C → F → G



Algorithm
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited.

The output of post-order traversal of this tree will be

− D → E → B → F → G → C → A

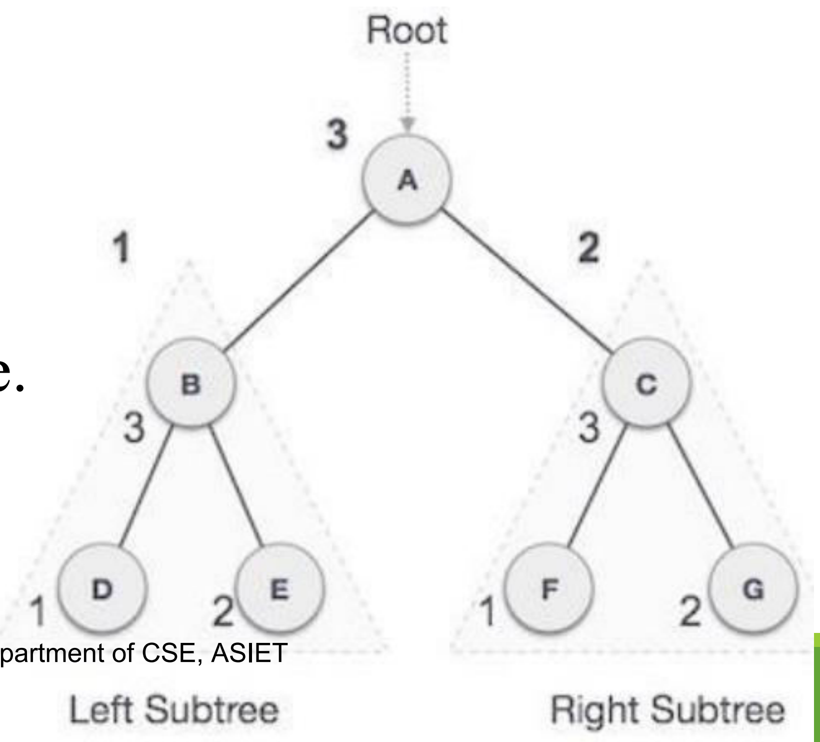
Algorithm

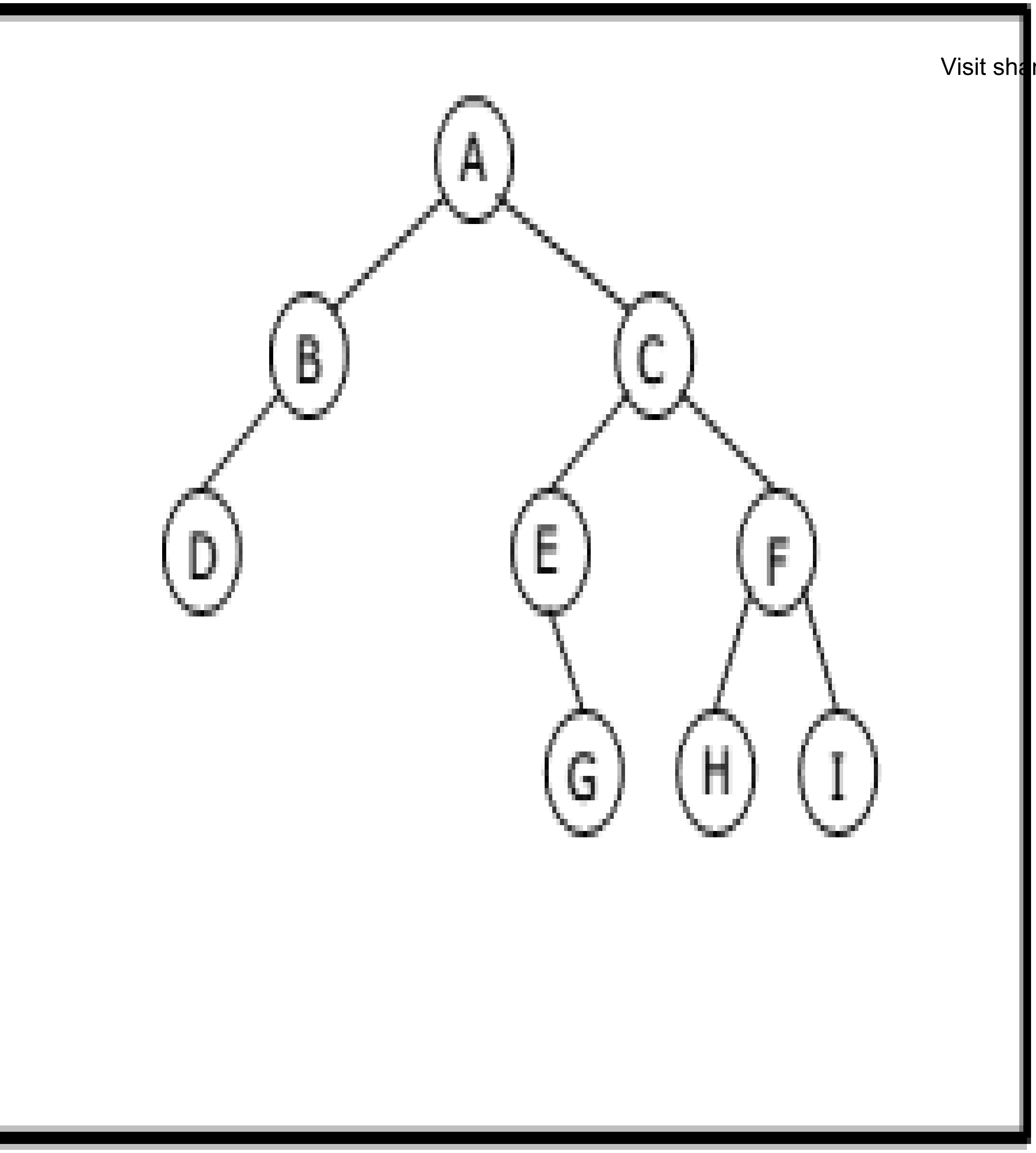
Until all nodes are traversed –

Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

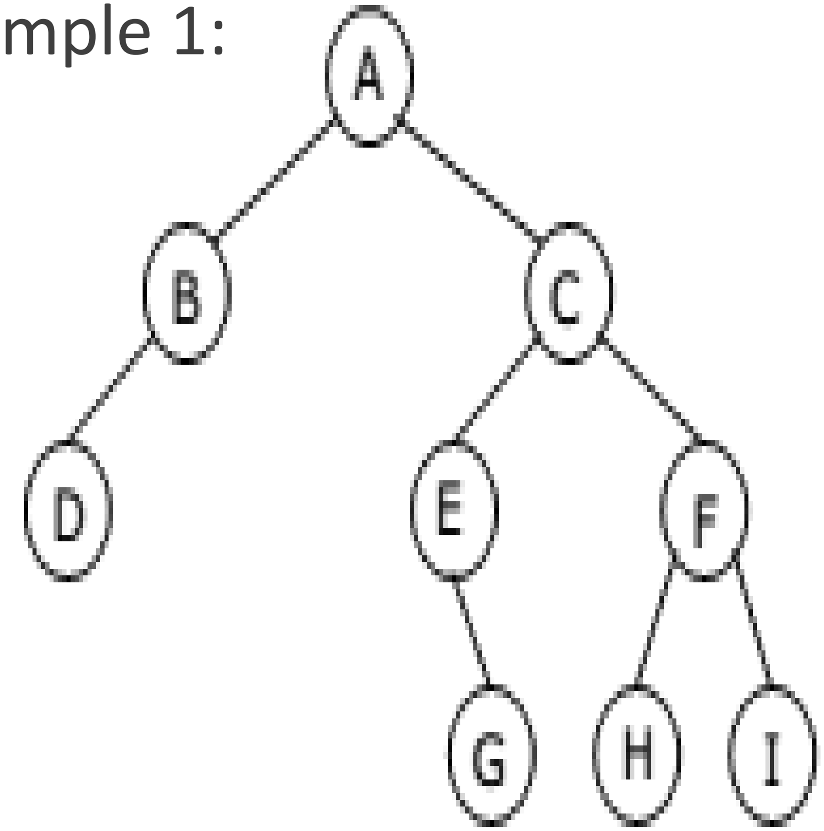
Step 3 − Visit root node





Binary Tree

Example 1:

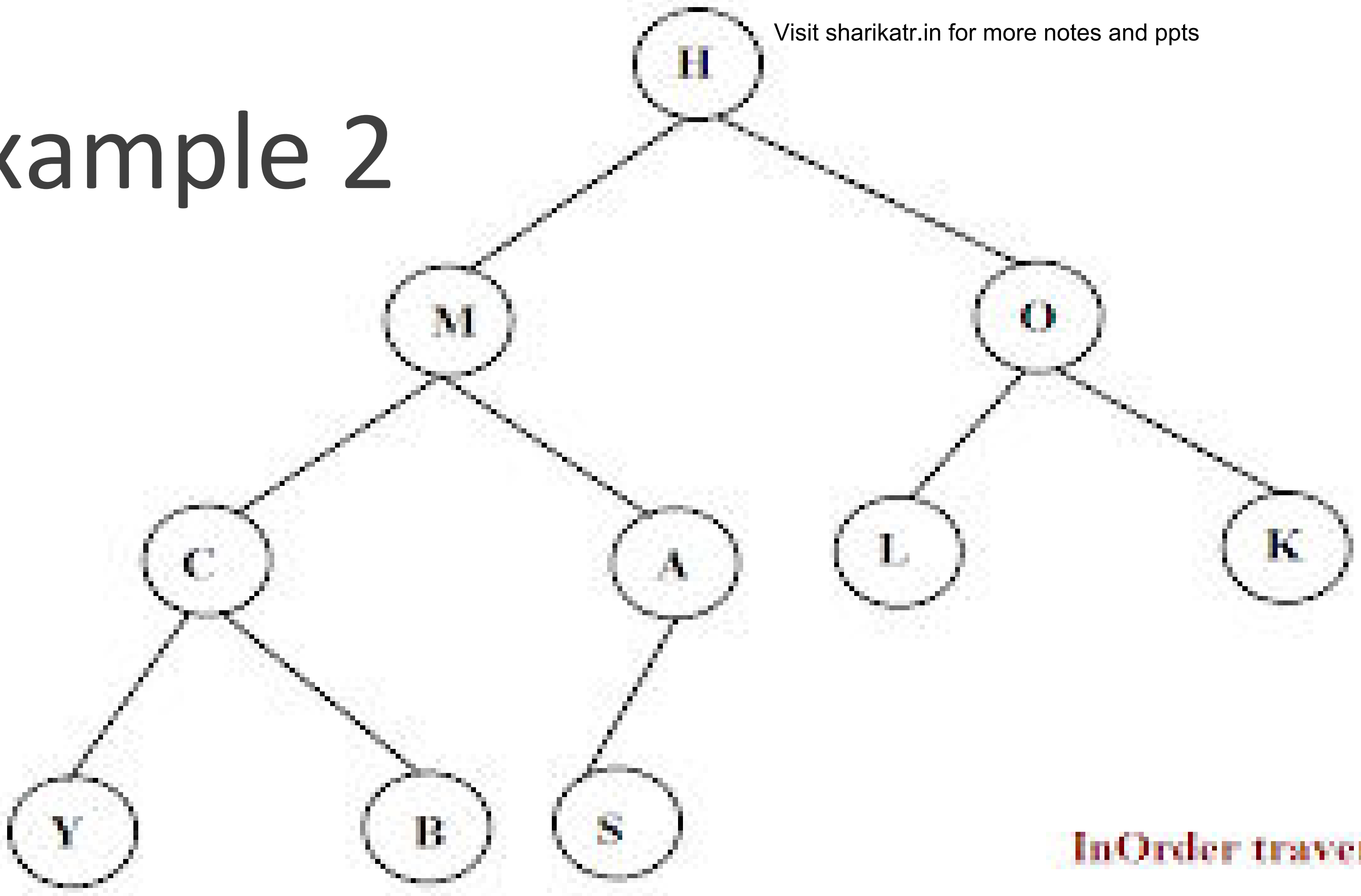


- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Binary Tree

Pre, Post, Inorder and level order Traversing

Example 2



InOrder traversal: Y C B M S A H L O K

Preorder traversal: H M C Y B A S O L K

Postorder traversal: Y B C S A M L K O H

Level order traversal: H M O C A L K Y B S

Tree Traversal

- Used to visit each node in the tree exactly once.
- Three types of traversal
  1. Preorder
  2. Inorder
  3. Postorder

Tree Traversal

- Used to visit each node in the tree exactly once.
- Three types of traversal
  1. **Preorder**
     - Visit root
     - Visit left subtree in preorder
     - Visit right subtree in preorder
  2. Inorder
  3. Postorder

TREE TRAVERSAL

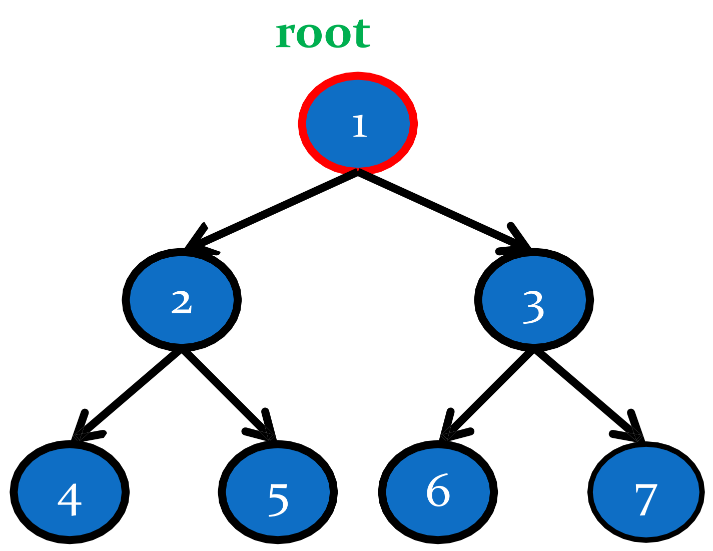
Tree Traversal

- Used to visit each node in the tree exactly once.
- Three types of traversal
  1. Preorder
  2. **Inorder**
     - Visit left subtree in inorder
     - Visit root
     - Visit right subtree in inorder
  3. Postorder

Tree Traversal

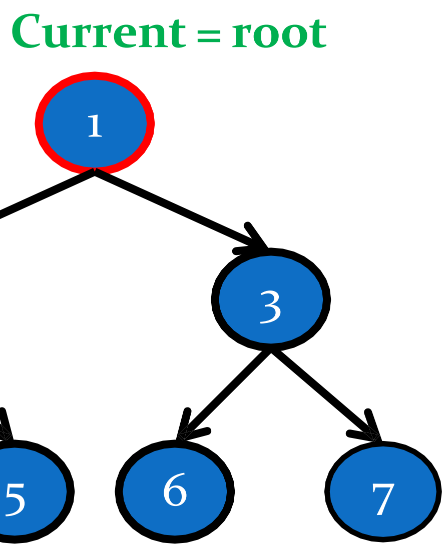
- Used to visit each node in the tree exactly once.
- Three types of traversal
  1. Preorder
  2. Inorder
  3. **Postorder**
     - Visit left subtree in inorder
       Visit right subtree in inorder
       Visit root

Preorder traversal



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

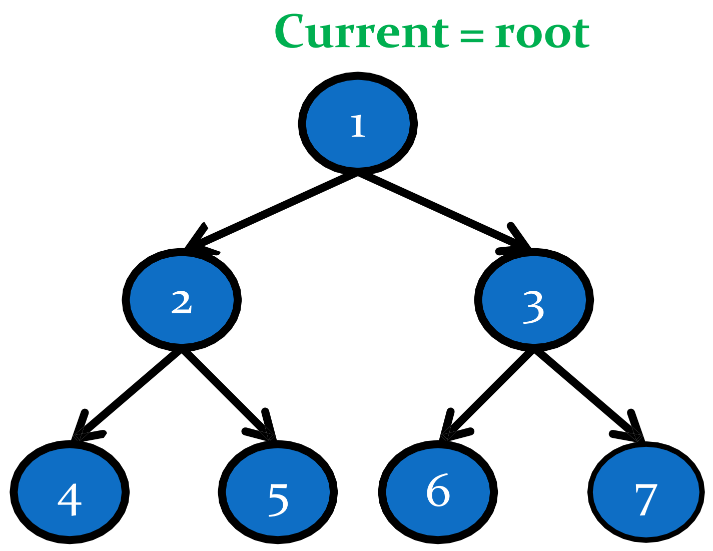


Preorder (root)

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

output

1



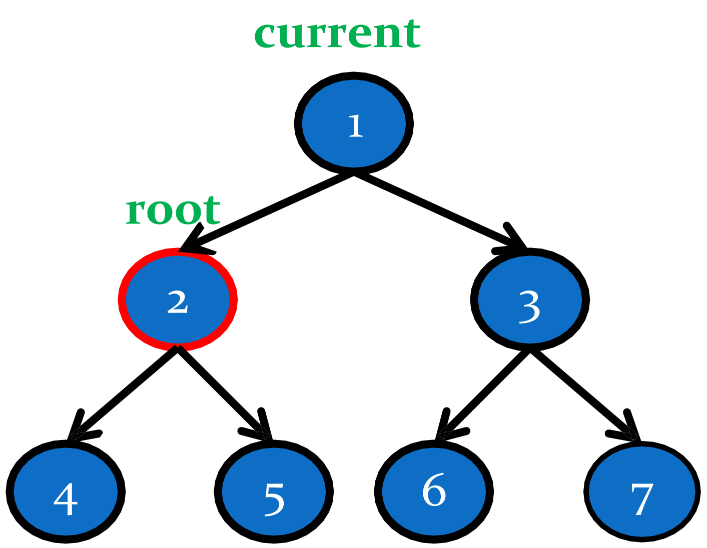
Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

1



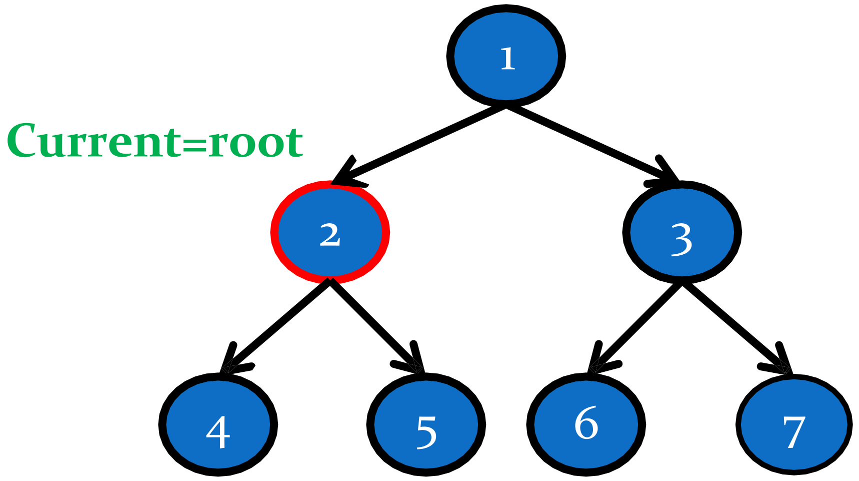
Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

1



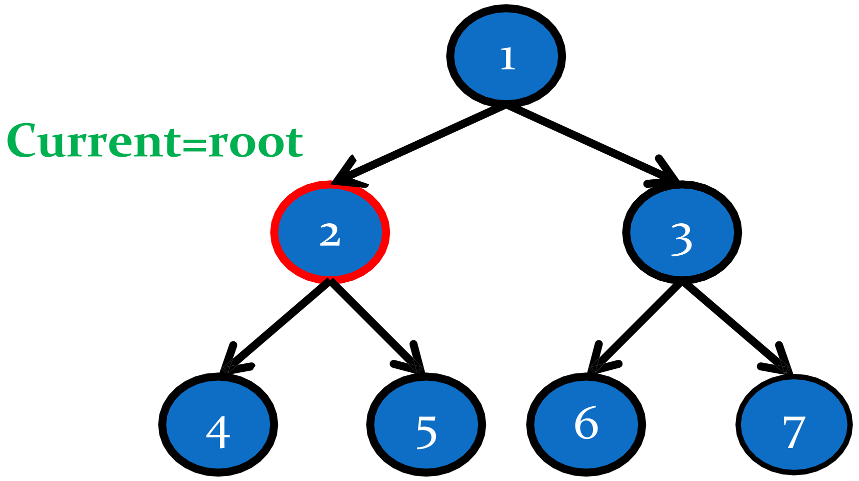
Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

12



Preorder (root)

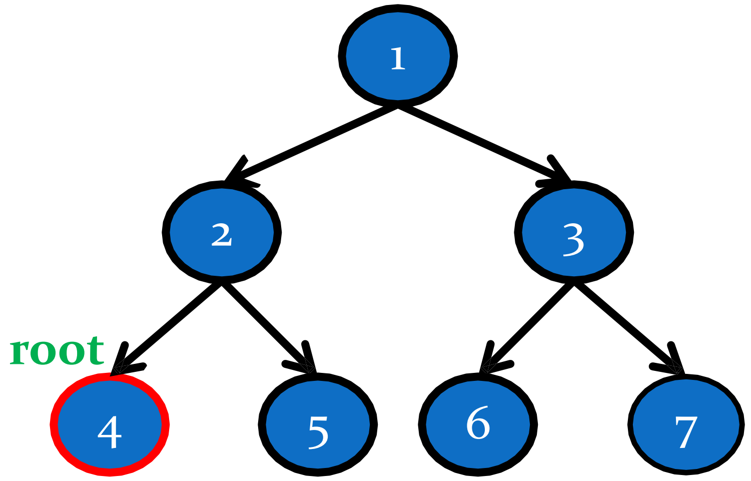
```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

12

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```



Preorder (root)

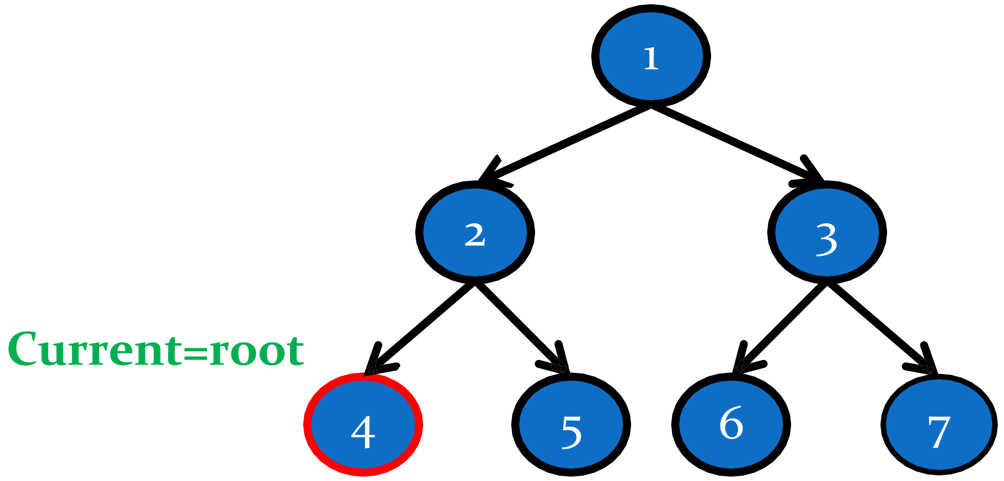
```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

12

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```



Preorder (root)

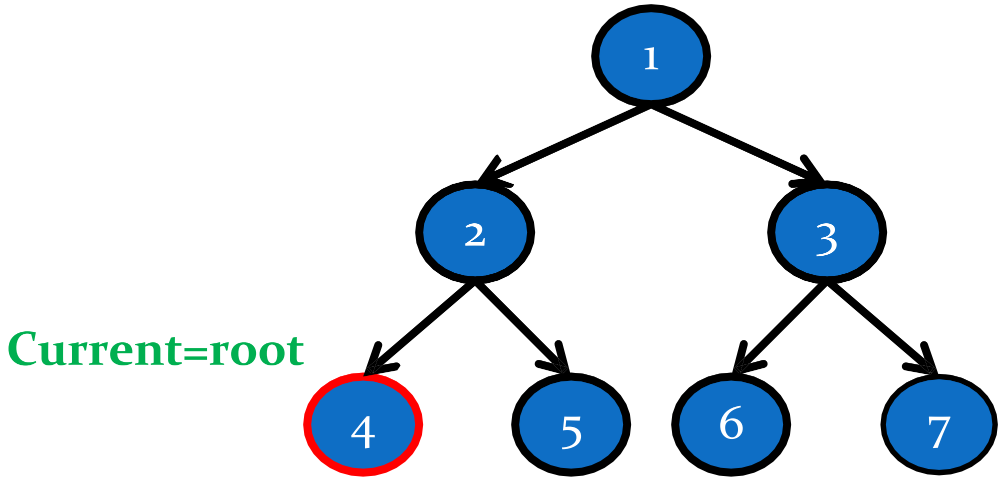
```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

124

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```



Preorder (root)

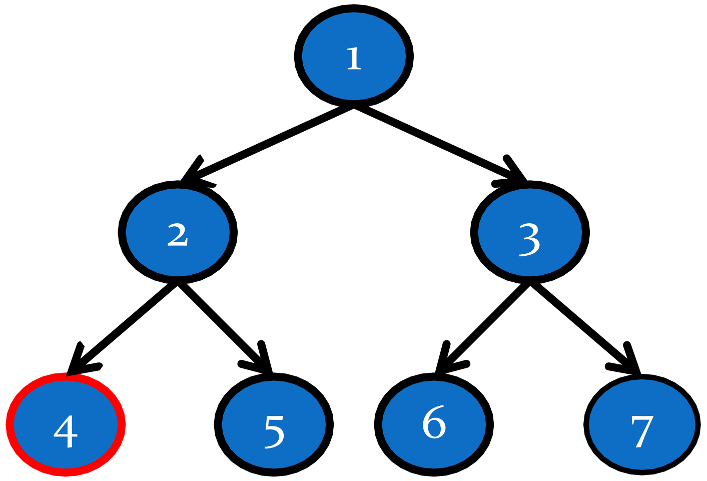
```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

124

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```



```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL

Preorder (root)

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

output

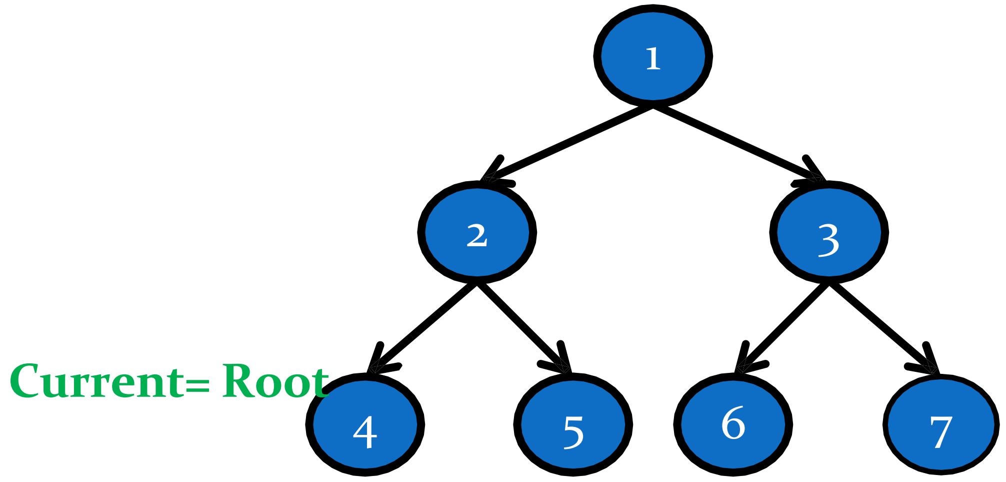
124

Root=4

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

Root=NULL

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

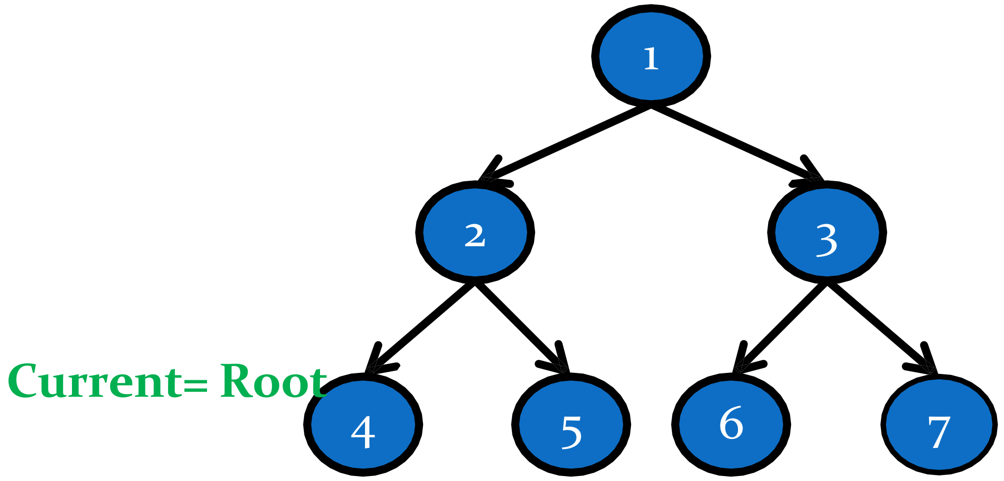
124

Root=4

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

output

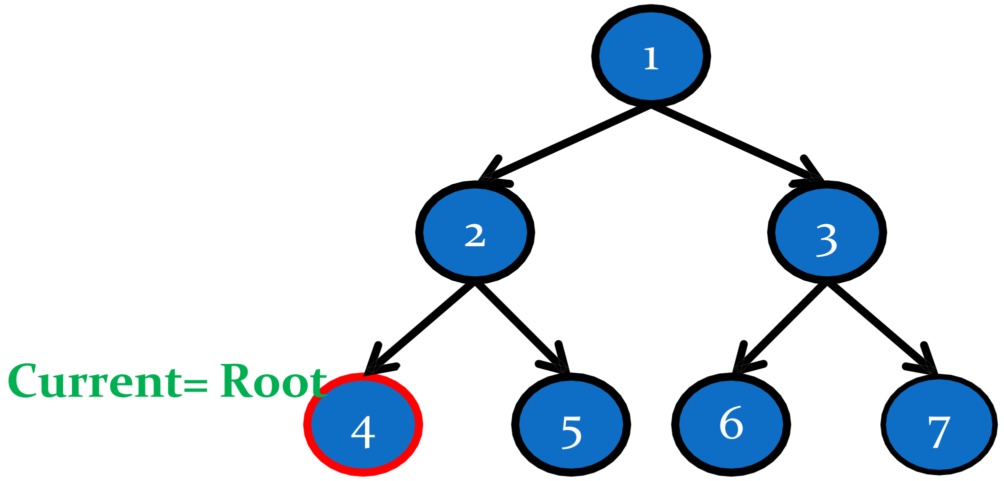
124

Root=4

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

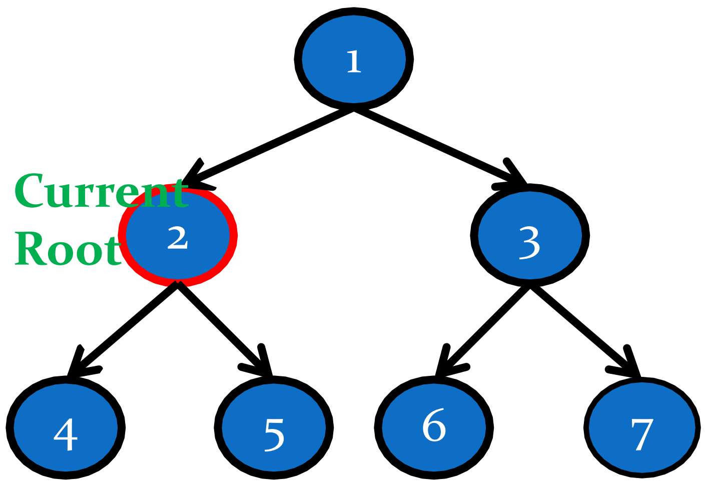
output

124

Root=4

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

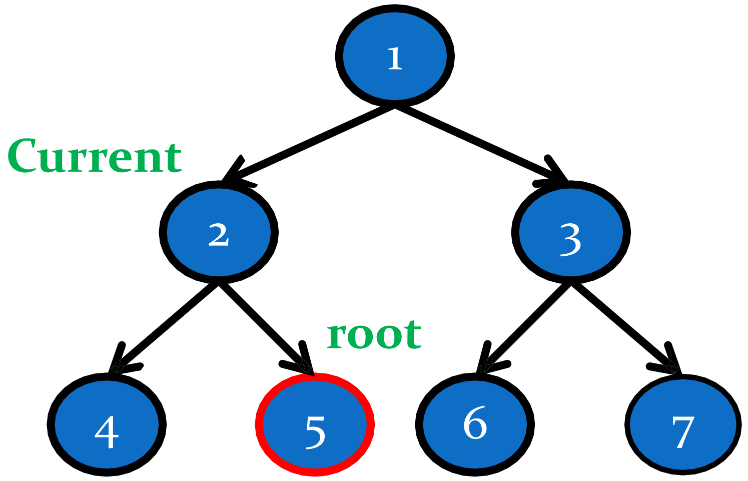
output

124

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

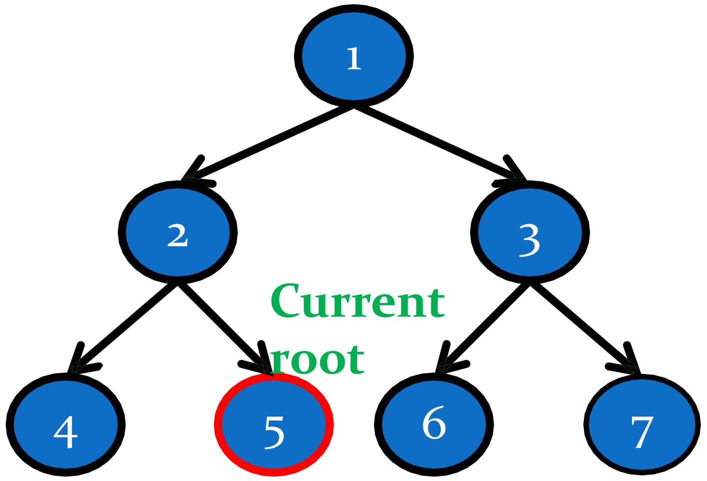
output

124

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

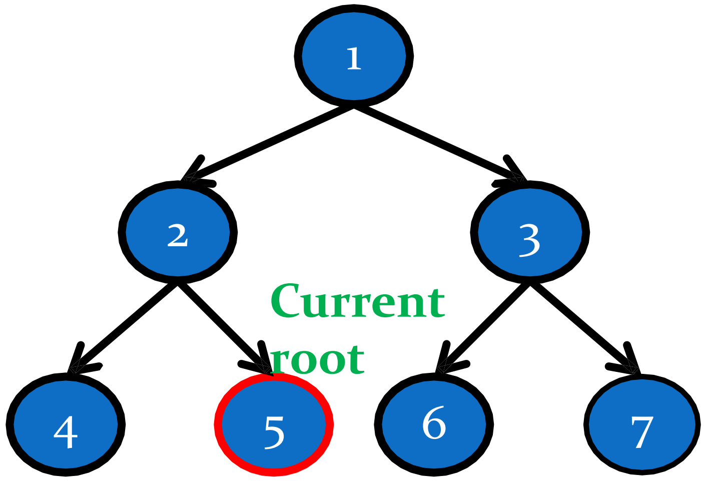
output

1245

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

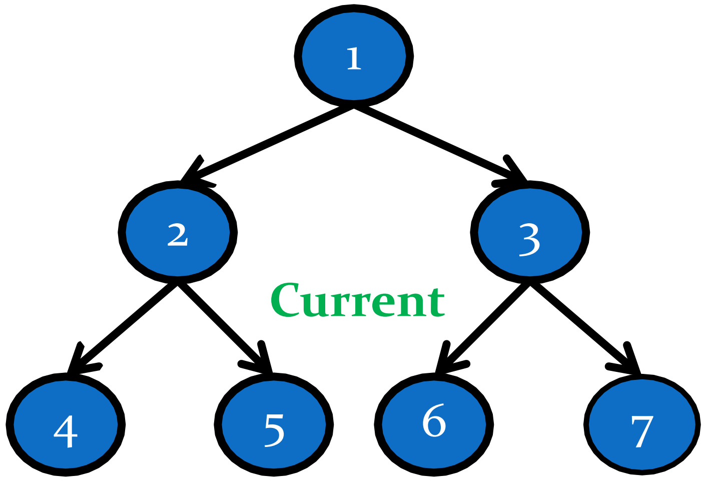
output

1245

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Root = NULL

Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

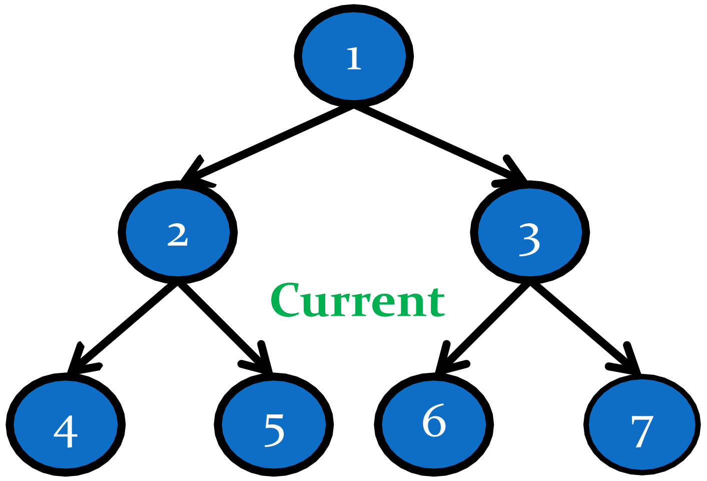
output

1245

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Root = NULL

Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

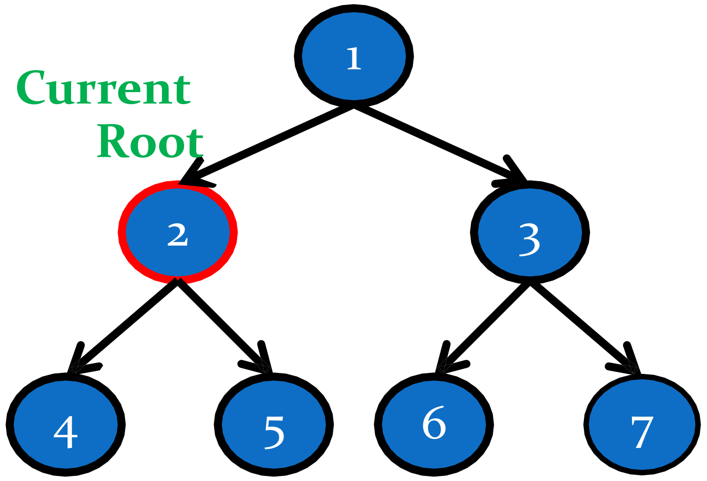
output

1245

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=2

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

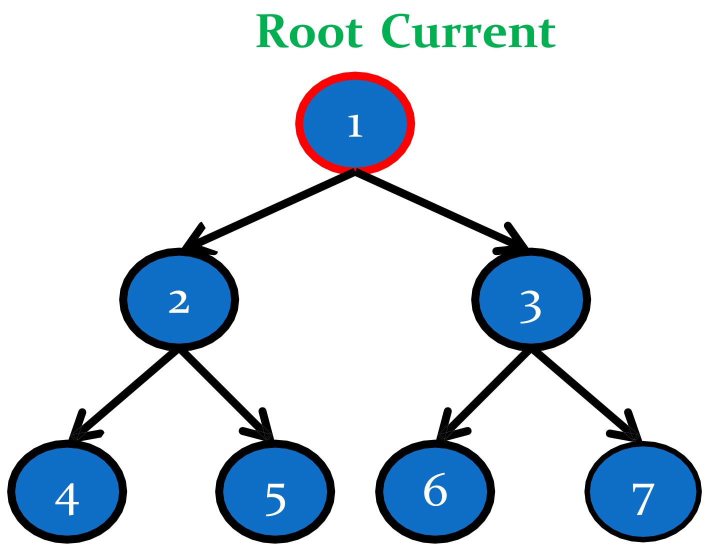
output

1245

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

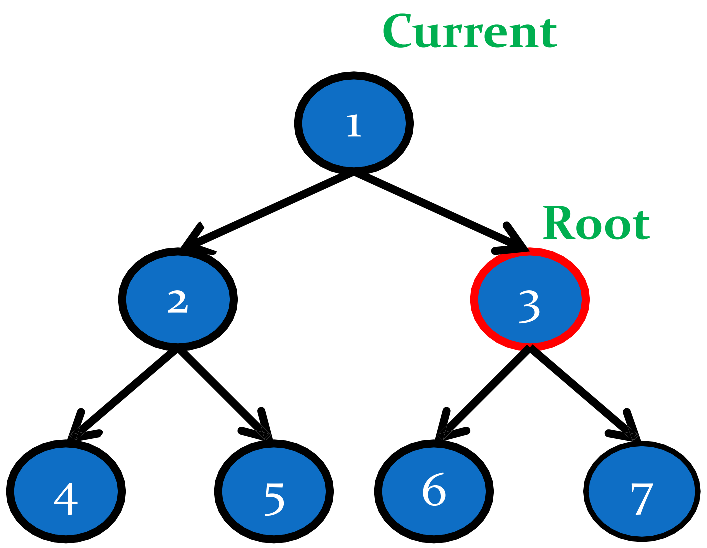
output

1245

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

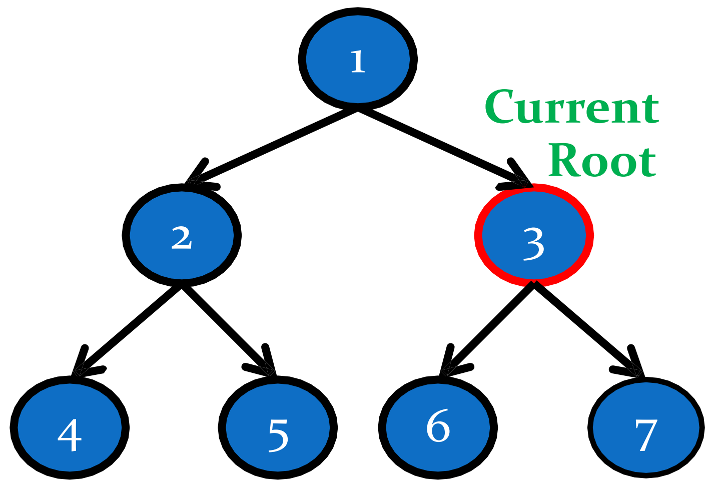
output

1245

Root=5

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

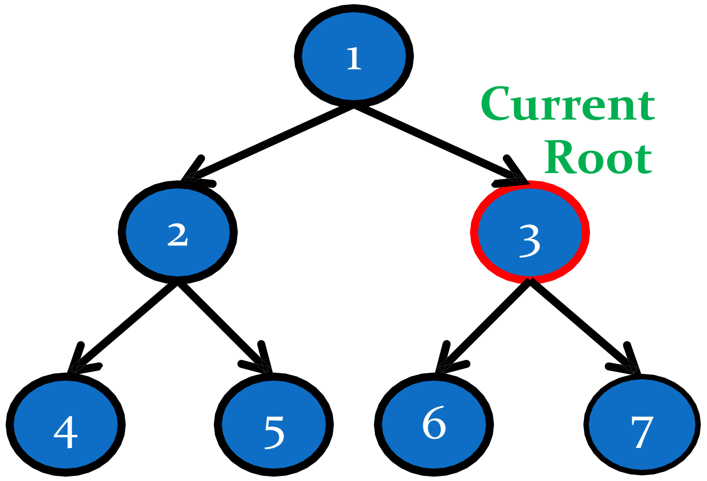
output

12453

Root=5

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

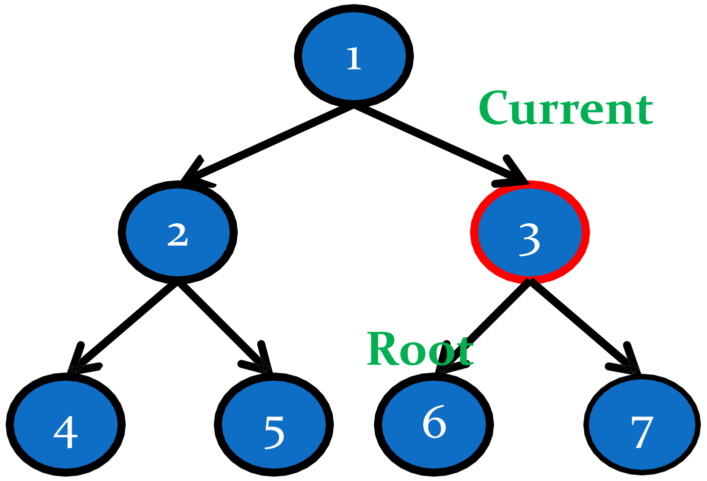
output

12453

Root=6

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

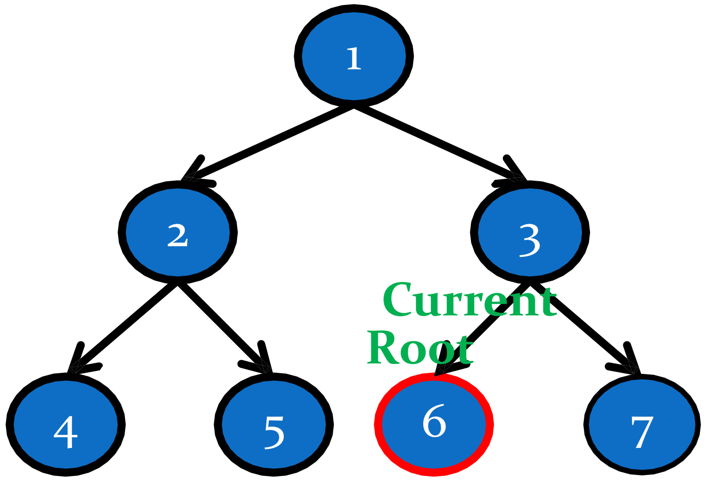
output

124536

Root=6

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

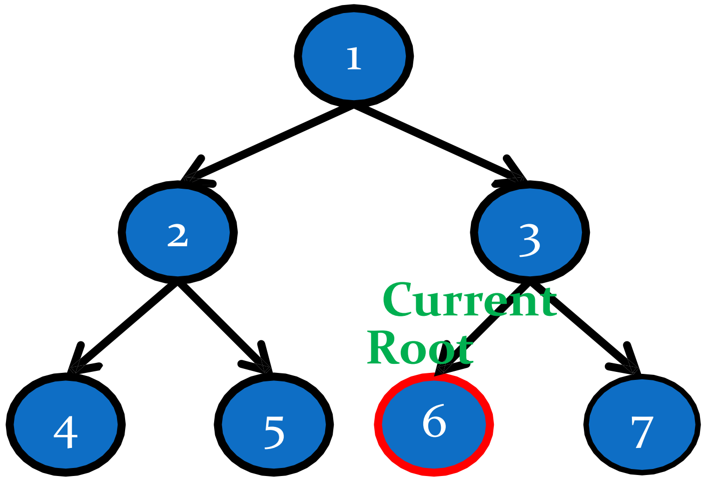
output

124536

Root=6

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

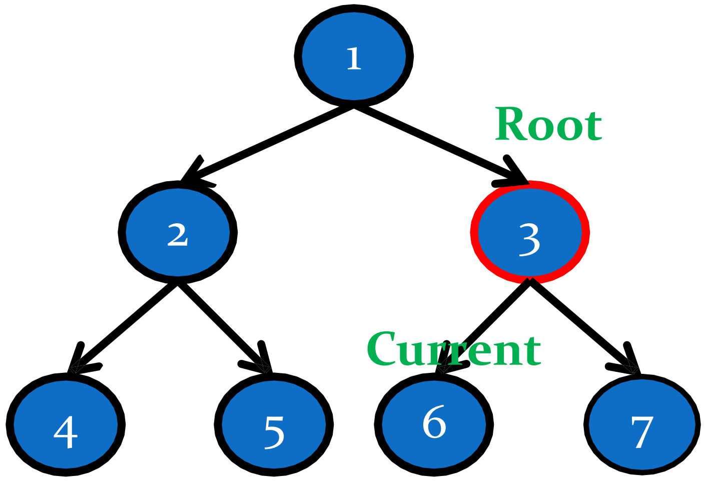
output

124536

Root=6

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

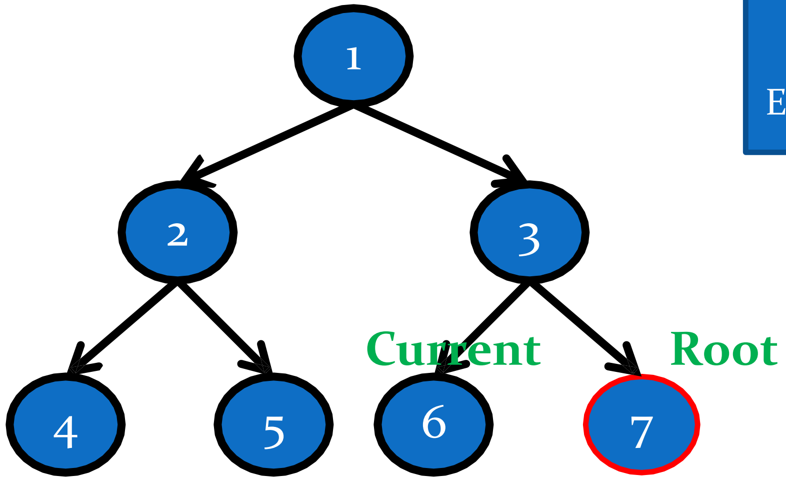
output

124536

Root=6

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

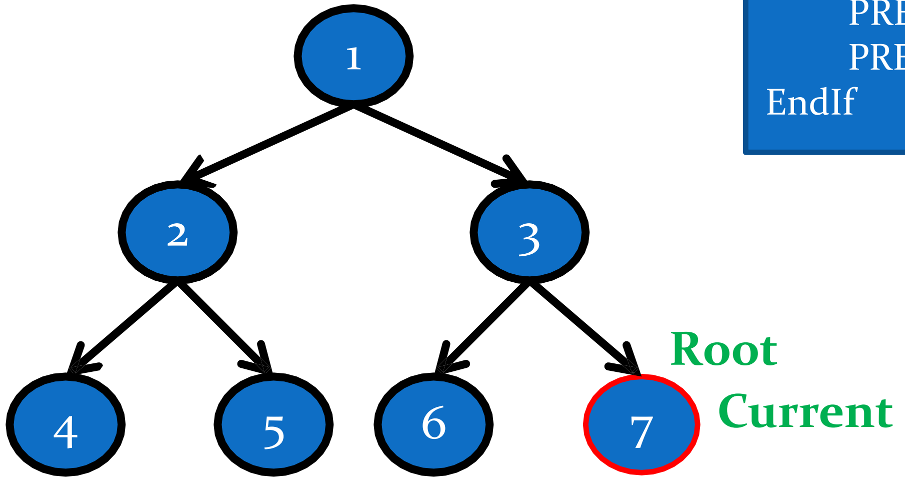
output

124536

Root=6

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

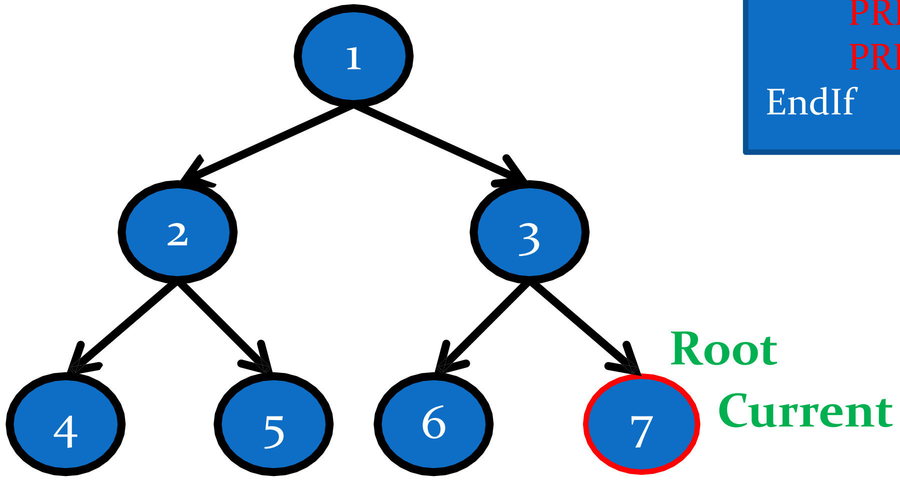
output

1245367

Root=6

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

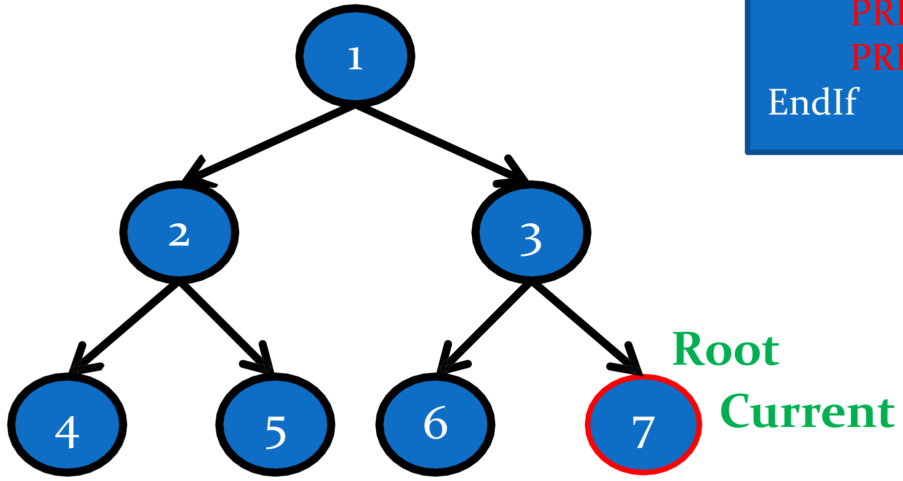
output

1245367

Root=6

```
Current=root
If(current!=NULL)
      VISIT(current)
      PREORDER(current.llink)
      PREORDER(current.rlink)
EndIf
```

Root=NULL



Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=3

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

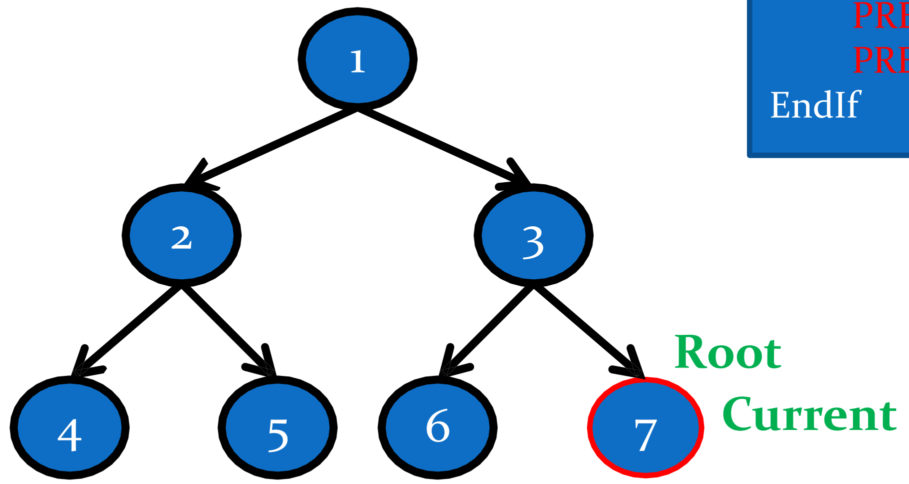
output

1245367

Root=6

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

Root=NULL

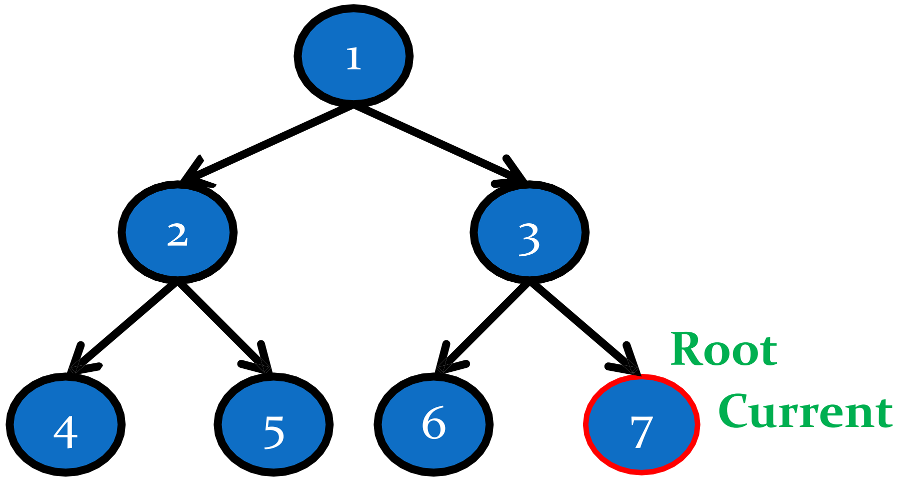


Preorder (root)

```
Current=root
If(current!=NULL)
    VISIT(current)
    PREORDER(current.llink)
    PREORDER(current.rlink)
EndIf
```

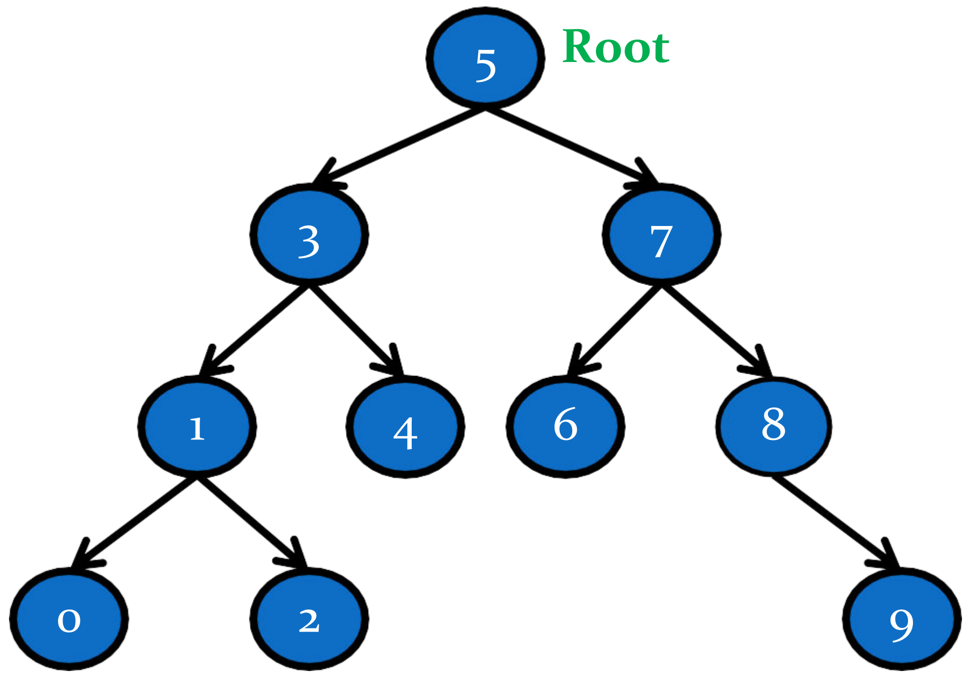
output

1245367



Question :

- **Write the preorder, inorder and postorder traversal of**

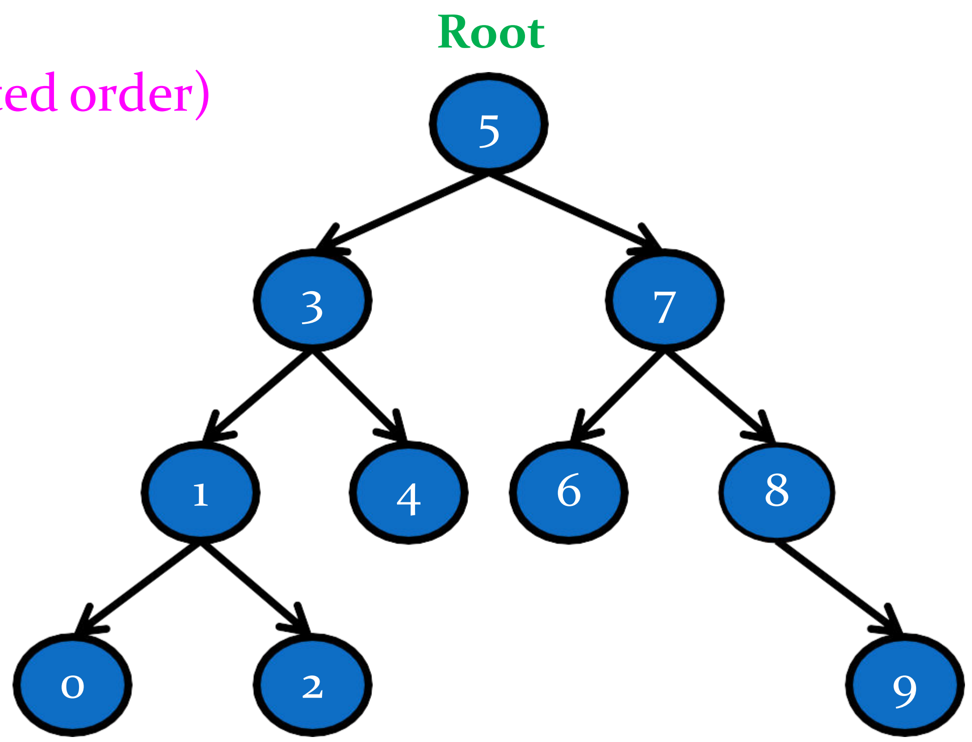


Answer:

PREORDER = 5 3 1 0 2 4 7 6 8 9

INORDER= 0 1 2 3 4 5 6 7 8 9 (sorted order)

POSTORDER= 0 2 1 4 3 6 9 8 7 5



1. Preorder traversal

Algorithm **PREORDER(root)**

Input: root is the pointer to the root node of the binary search tree

Output: Visiting of all the nodes in preorder fashion

Data structure: linked structure

Steps:

```
1. Current=root
2. If(current!=NULL)
    1. VISIT(current)
    2. PREORDER(current.llink)
    3. PREORDER(current.rlink)
3. EndIf
4. Stop
```

2. Inorder Traversal

Algorithm INORDER(root)

Input: root is the pointer to the root node of the binary search tree

Output: Visiting of all the nodes in inorder fashion

Data structure: linked structure

Steps:

```
1. Current=root
2. If(current!=NULL)
    1. INORDER(current.llink)
    2. VISIT(current)
    3. INORDER(current.rlink)
3. EndIf
4. Stop
```

3. Postorder Traversal

Algorithm **POSTORDER(root)**

Input: root is the pointer to the root node of the binary search tree

Output: Visiting of all the nodes in postorder fashion

Data structure: linked structure

Steps:

```
1. Current=root
2. If(current!=NULL)
    1. POSTORDER(current.llink)
    2. POSTORDER(current.rlink)
    3. VISIT(current)
3. EndIf
4. Stop
```

Construct tree from the inorder and preorder traversals

PREORDER = 5 3 1 0 2 4 7 6 8 9

INORDER= 0 1 2 3 4 5 6 7 8 9

Construct tree
from the following inorder and preorder traversals

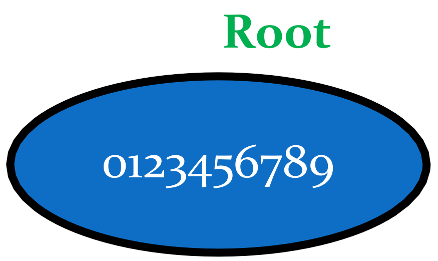
PREORDER = 5 3 1 0 2 4 7 6 8 9

INORDER= 0 1 2 3 4 5 6 7 8 9

- Preorder starts with the root
- Now split the inorder traversal into 3
  - Left subtree, root, right subtree
- Repeat the same process on each subtrees

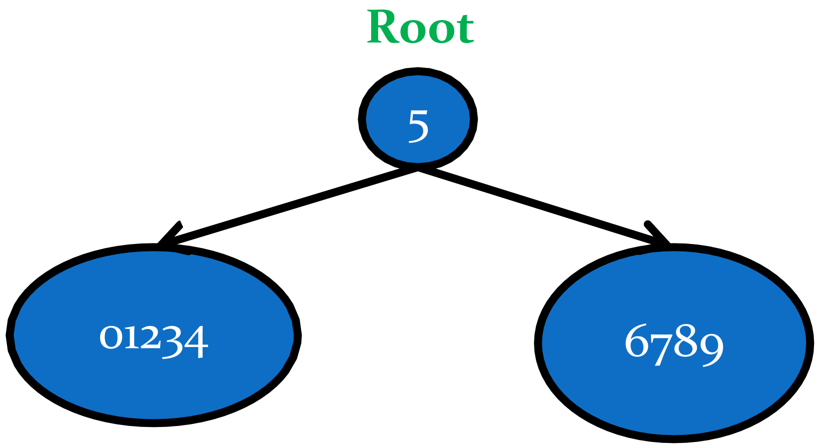
PREORDER = 5 3 1 0 2 4 7 6 8 9
INORDER= 0 1 2 3 4 5 6 7 8 9

Root = 5
[01234] [**5**] [6789]



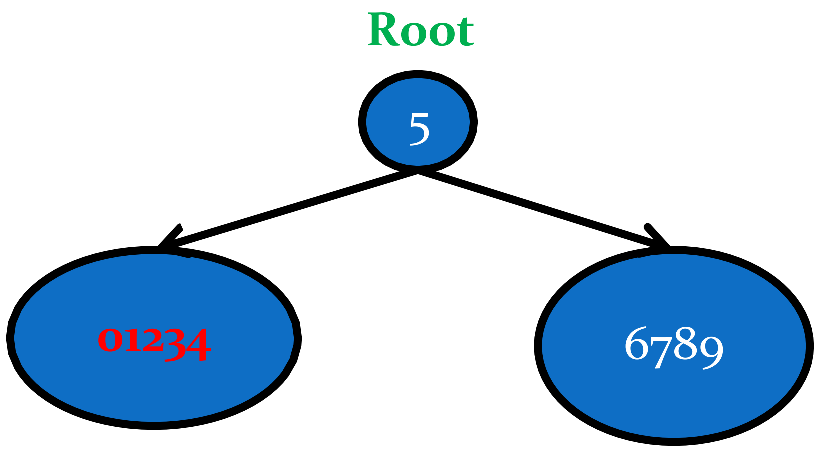
PREORDER = 5 3 1 0 2 4 7 6 8 9
INORDER= 0 1 2 3 4 5 6 7 8 9

Root = 5
[01234] [5] [6789]



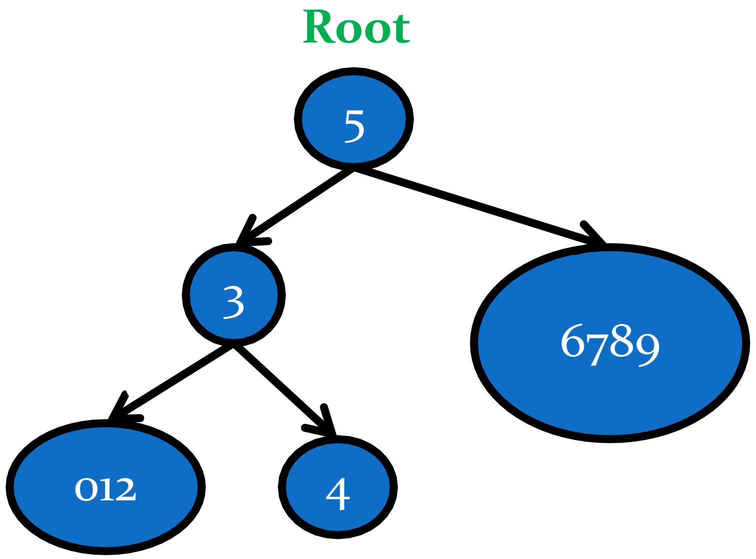
PREORDER = 5 3 1 0 2 4 7 6 8 9

INORDER= 0 1 2 3 4 5 6 7 8 9



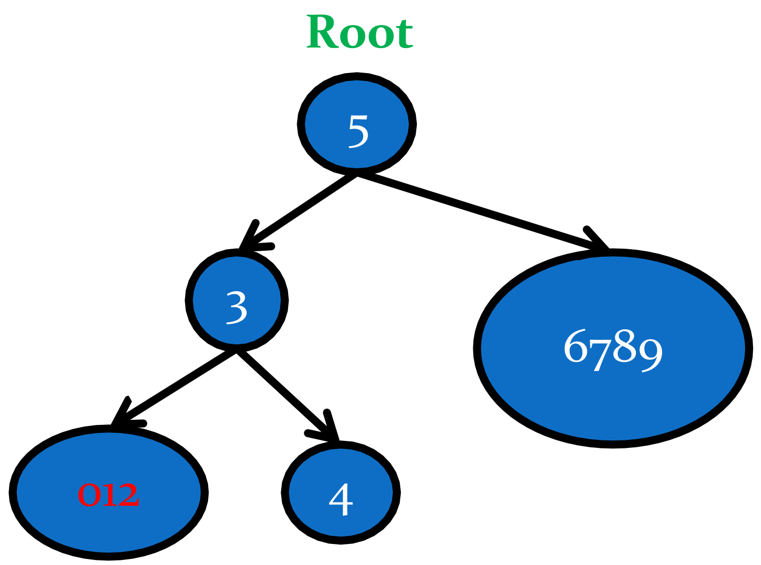
PREORDER = 5 3 1 0 2 4 7 6 8 9
INORDER= 0 1 2 3 4 5 6 7 8 9

Root = 3
[012] [3] [4]



PREORDER = 5 3 1 0 2 4 7 6 8 9

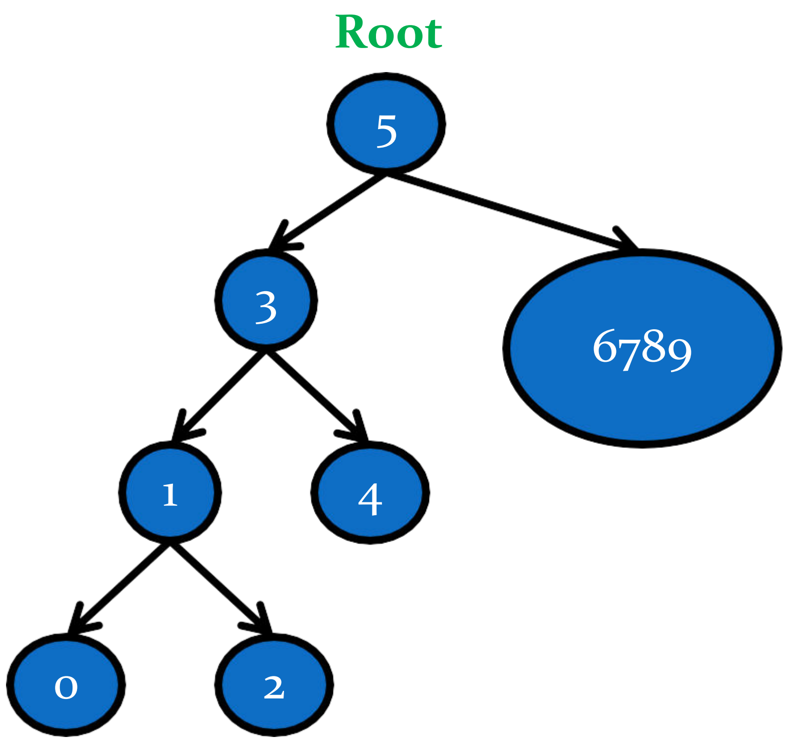
INORDER= 0 1 2 3 4 5 6 7 8 9



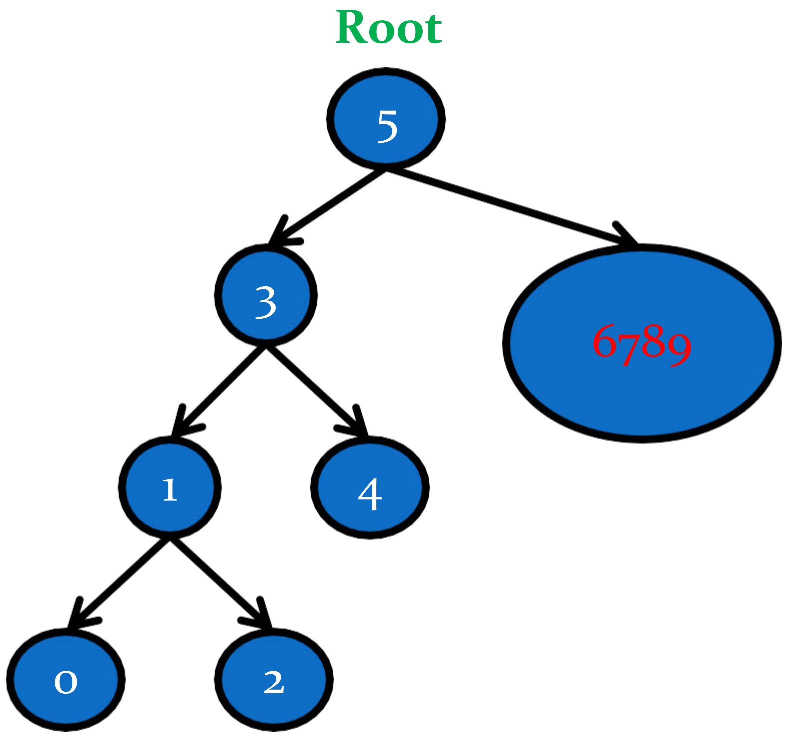
PREORDER = 5 3 1 0 2 4 7 6 8 9
INORDER= 0 1 2 3 4 5 6 7 8 9

Root = 1
[0] [**1**] [2]

No more elements to be divided
In the left subtree



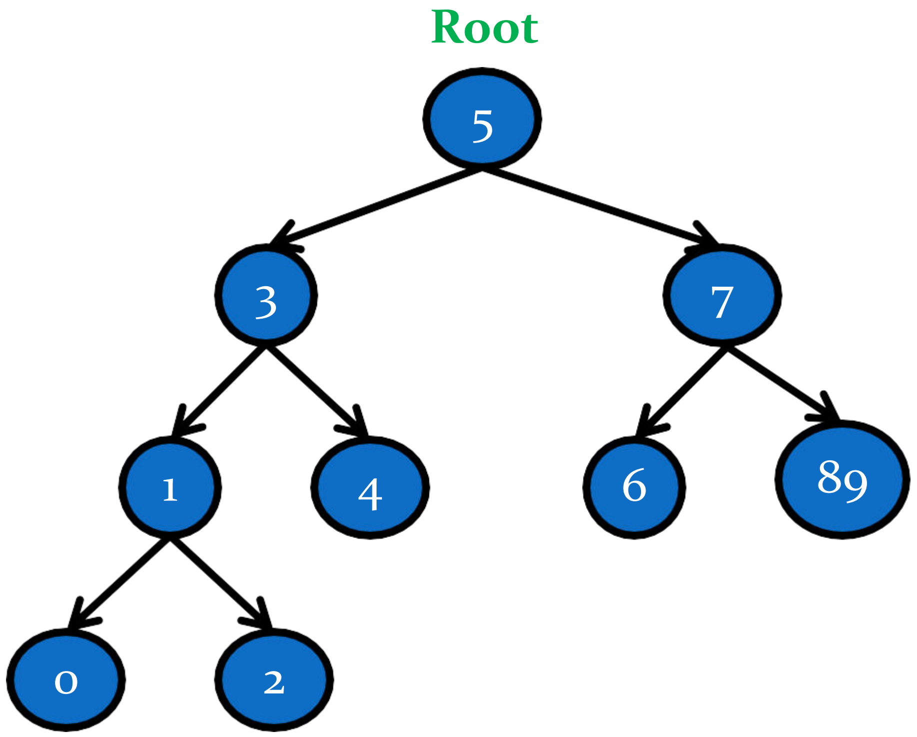
PREORDER = 5 3 1 0 2 4 7 6 8 9
INORDER= 0 1 2 3 4 5 6 7 8 9



PREORDER = 5 3 1 0 2 4 7 6 8 9
INORDER= 0 1 2 3 4 5 6 7 8 9

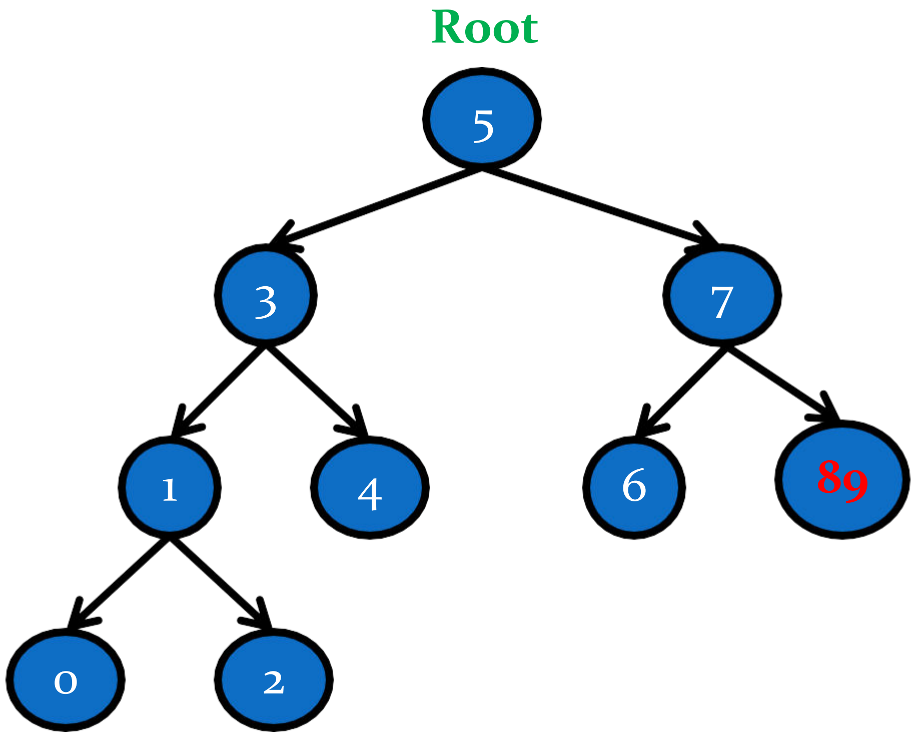
Root =7

[6] [**7**] [89]



PREORDER = 5 3 1 0 2 4 7 6 8 9

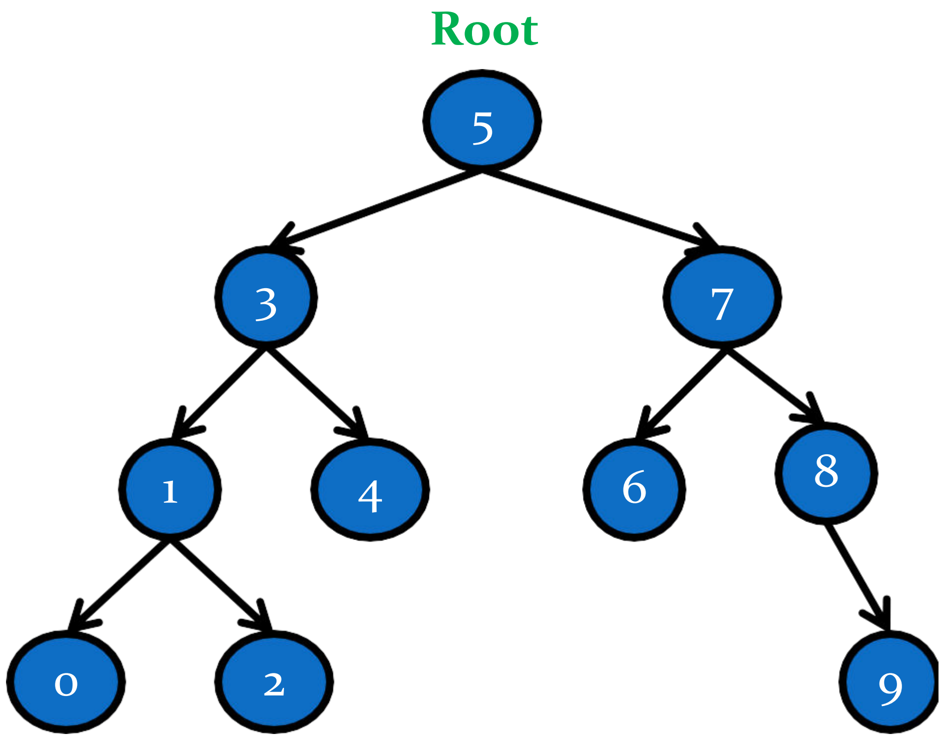
INORDER= 0 1 2 3 4 5 6 7 8 9



PREORDER = 5 3 1 0 2 4 7 6 8 9
INORDER= 0 1 2 3 4 5 6 7 8 9

Root =8
 [**8**] [9]

No more elements to be divided
In the right subtree



Question:

- **Construct the tree from the following inorder and postorder traversals**

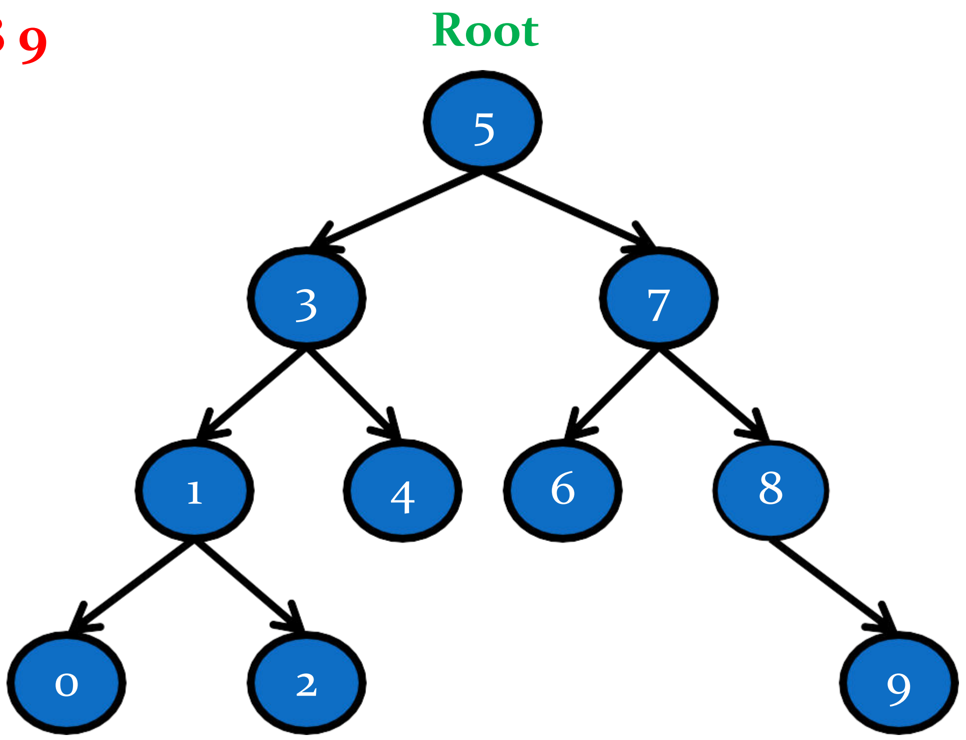
POSTORDER = 0 2 1 4 3 6 9 8 7 5

INORDER= 0 1 2 3 4 5 6 7 8 9

Answer:

POSTORDER = 0 2 1 4 3 6 9 8 7 5

INORDER = 0 1 2 3 4 5 6 7 8 9



Given the following **postorder** and **inorder** traversals:

POSTORDER: 4 5 2 6 7 3 1
INORDER: 4 2 5 1 6 3 7

- Construct the binary tree from these traversals.

```
   1
  / \
 2   3
/ \ / \
4  5 6  7
```

Given the following **inorder** and **preorder** traversals:

Inorder: 9 3 15 20 7
Preorder: 3 9 20 15 7

- Construct the binary tree from these traversals.

```
  3
 / \
9   20
   /  \
  15   7
```

Given the following **inorder** and **preorder** traversals:

Inorder: 8 4 9 2 10 5 1 6 11 3 12 7
Preorder: 1 2 4 8 9 5 10 3 6 11 7 12

- Construct the binary tree from these traversals.

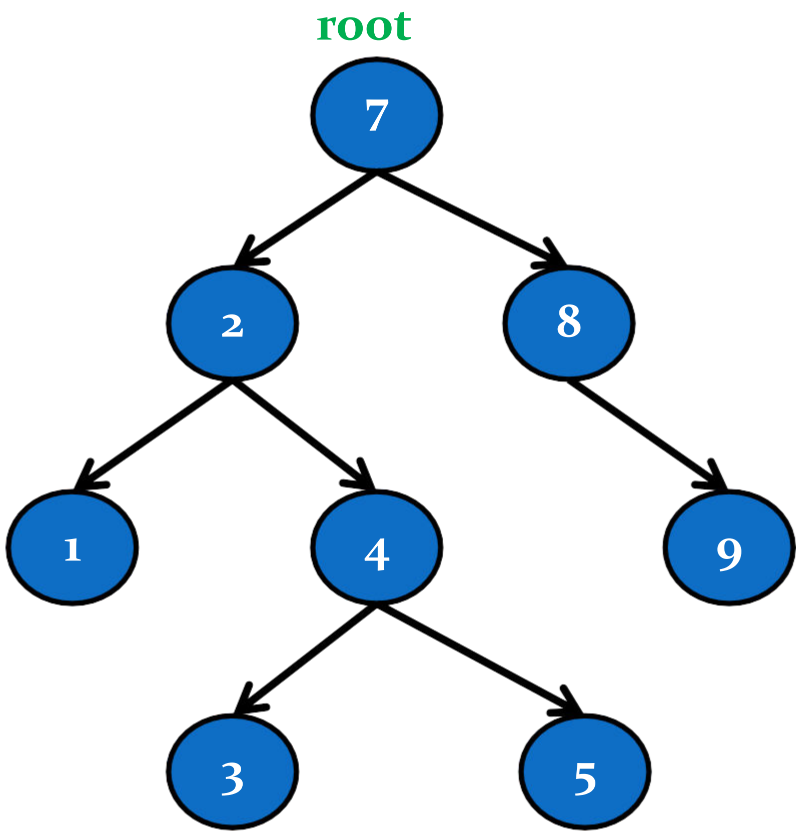
```
       1
      / \
    2     3
   / \  /   \
  4   5 6     7
 / \  /  \    /
8  9 10   11 12
```

BINARY SEARCH TREE

A Binary search tree is a binary tree.

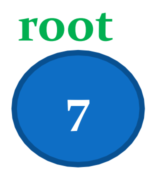
- It may be empty.
- If it is not empty, then it satisfies the following properties:
    1. Every element has a key and no two elements have the same key (no duplicates)
    2. The keys (if any) in the left subtree are smaller than the key in the root
    3. The keys (if any) in the right subtree are larger than the key in the root
    4. The left and right subtrees are also binary search trees

Eg: BINARY SEARCH TREE



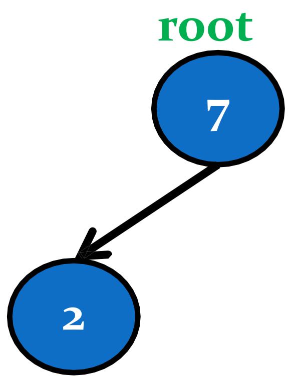
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



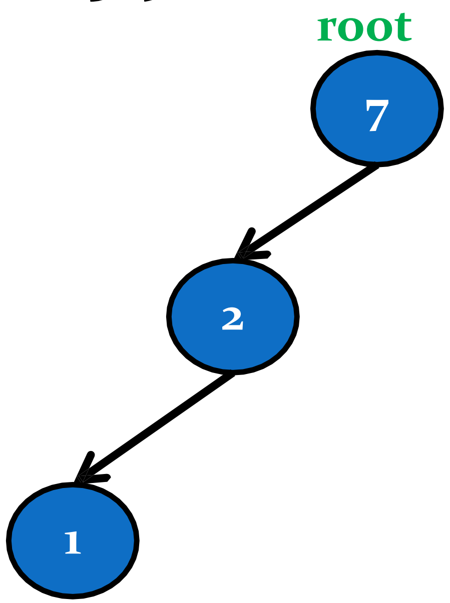
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



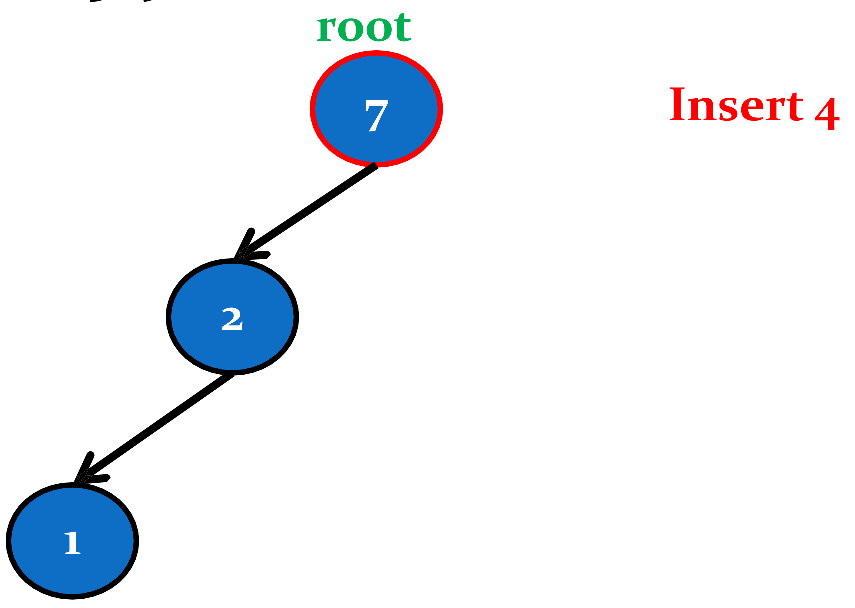
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



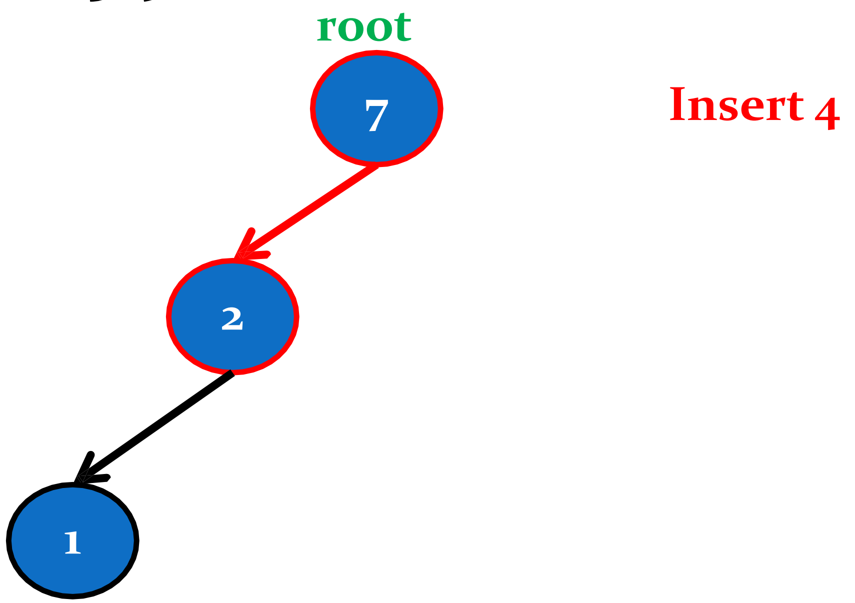
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



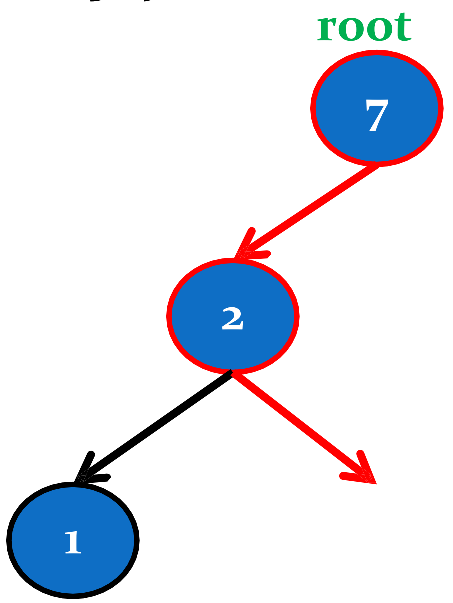
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



Construction of binary search tree

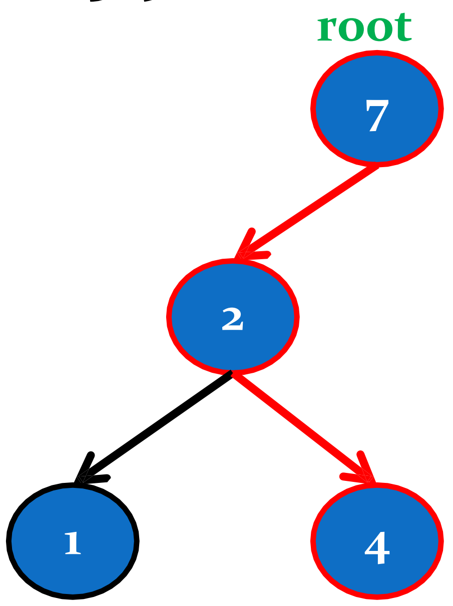
Insert 7,2,1,4,3,8,5,9



Insert 4

Construction of binary search tree

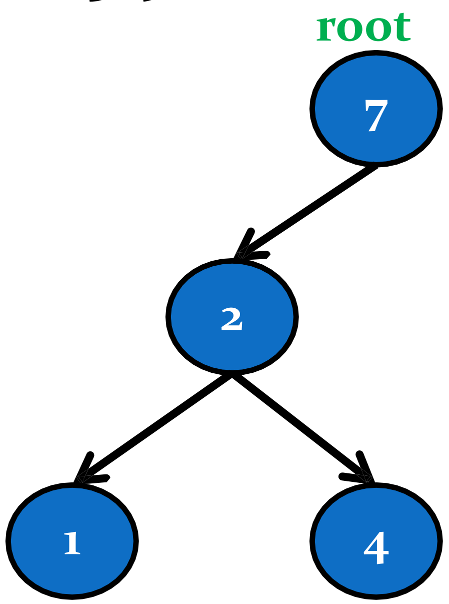
Insert 7,2,1,4,3,8,5,9



Insert 4

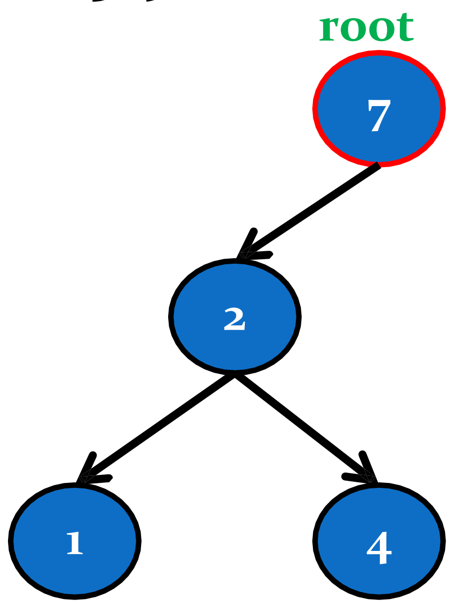
Construction of binary search tree

Insert **7**,**2**,1,4,3,8,5,9



Construction of binary search tree

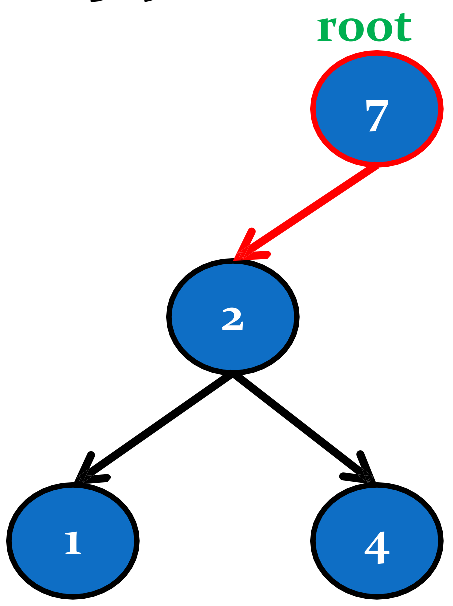
Insert 7,2,1,4,3,8,5,9



Insert 3

Construction of binary search tree

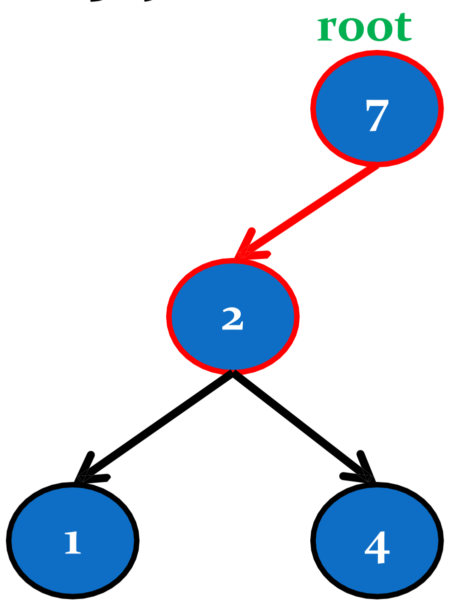
Insert 7,2,1,4,3,8,5,9



Insert 3

Construction of binary search tree

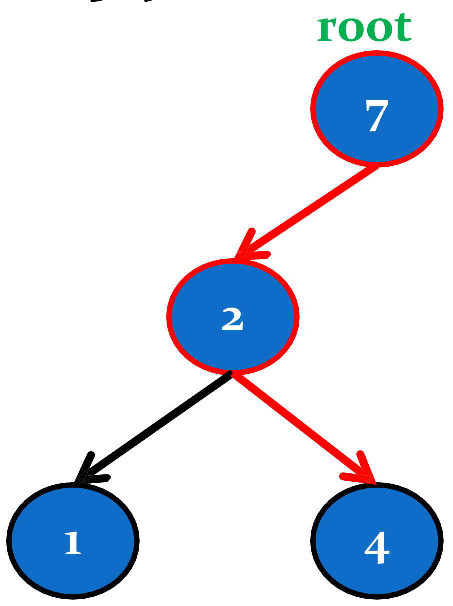
Insert 7,2,1,4,3,8,5,9



Insert 3

Construction of binary search tree

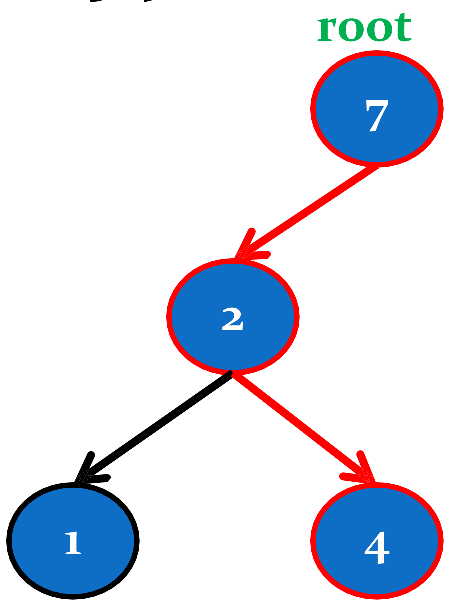
Insert 7,2,1,4,3,8,5,9



Insert 3

Construction of binary search tree

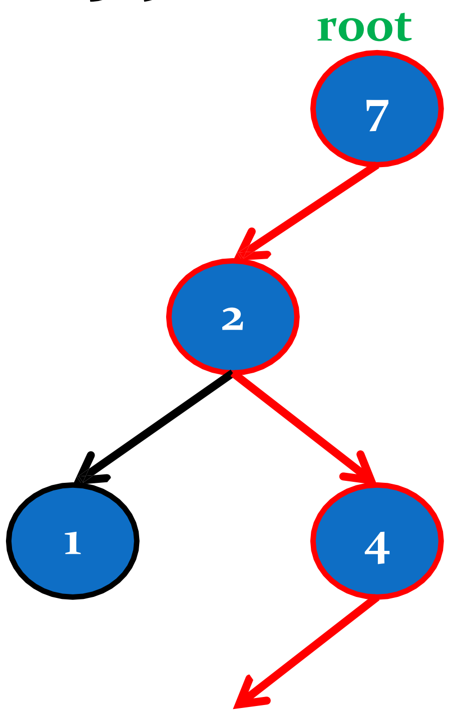
Insert 7,2,1,4,3,8,5,9



Insert 3

Construction of binary search tree

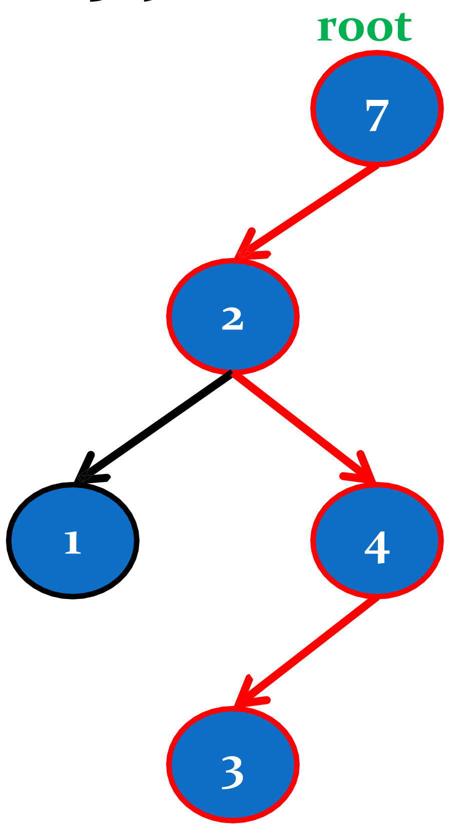
Insert 7,2,1,4,3,8,5,9



Insert 3

Construction of binary search tree

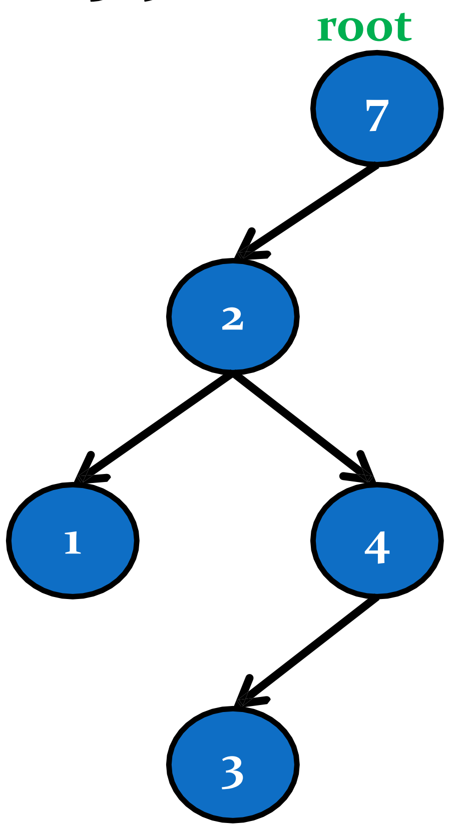
Insert 7,2,1,4,3,8,5,9



Insert 3

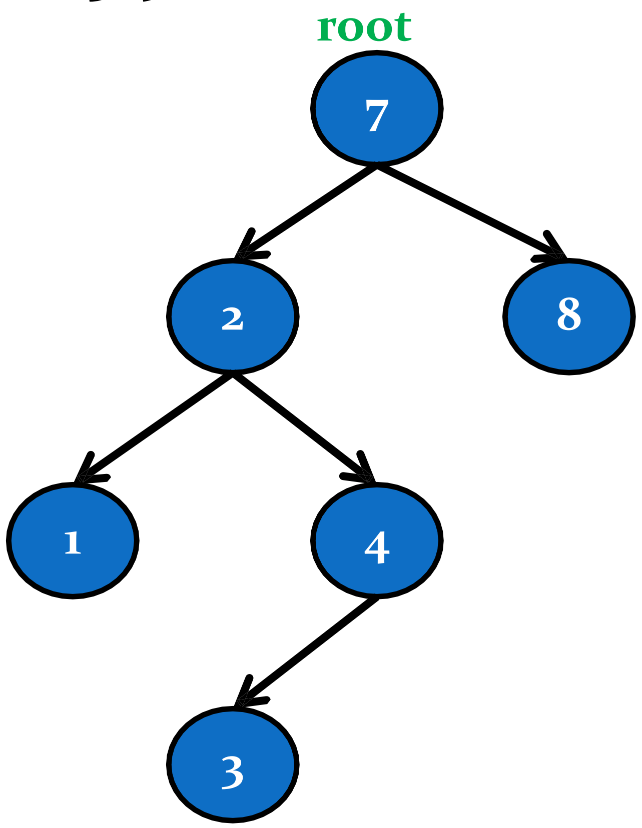
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



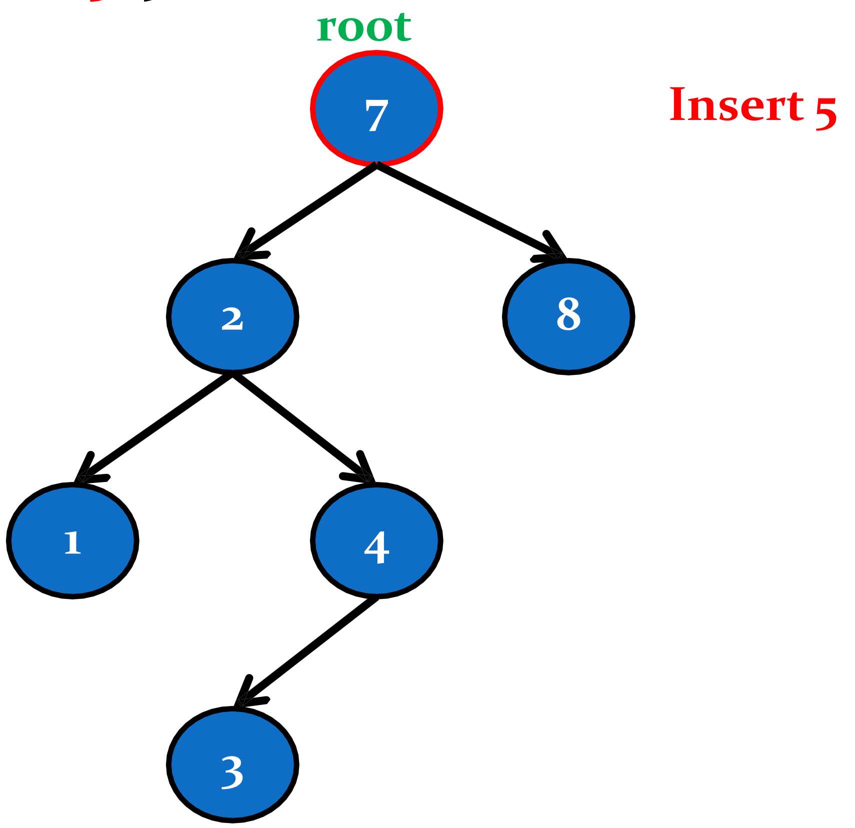
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



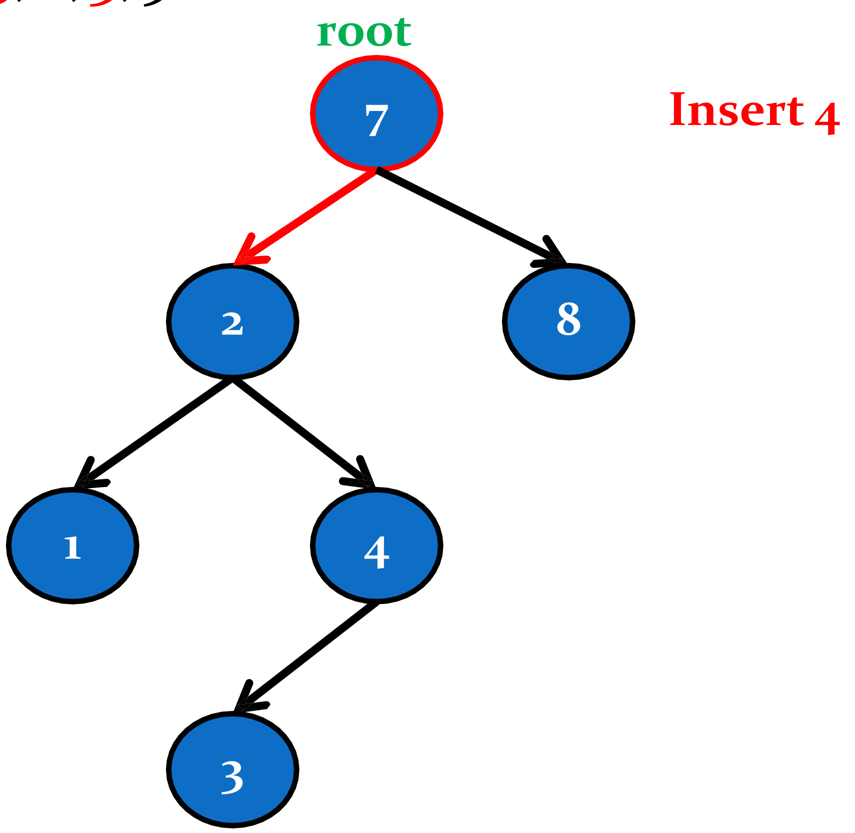
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



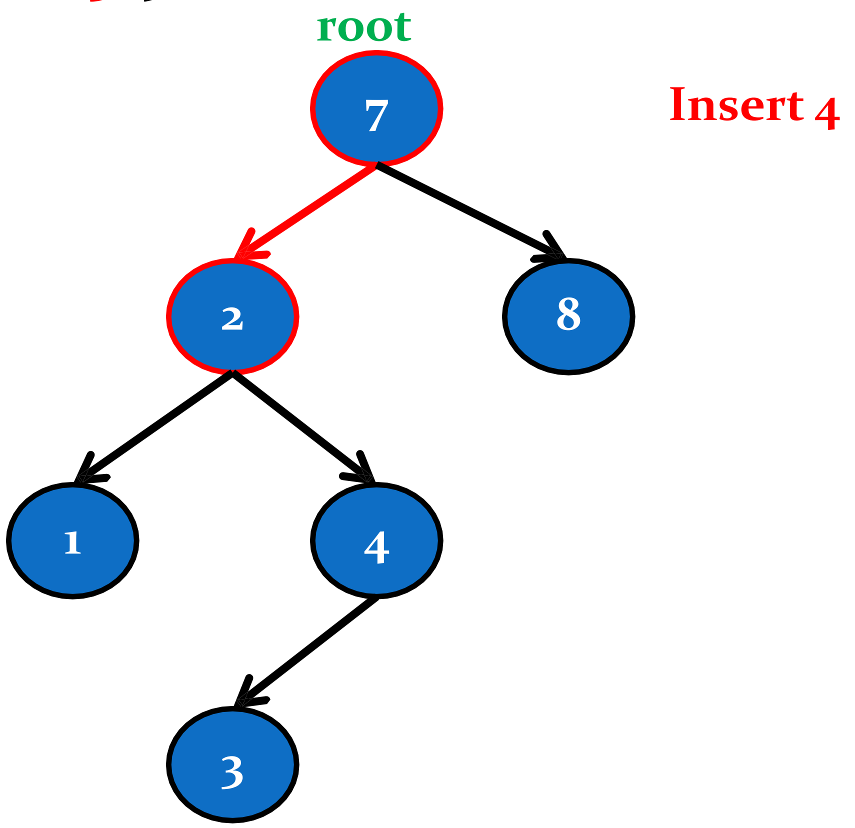
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



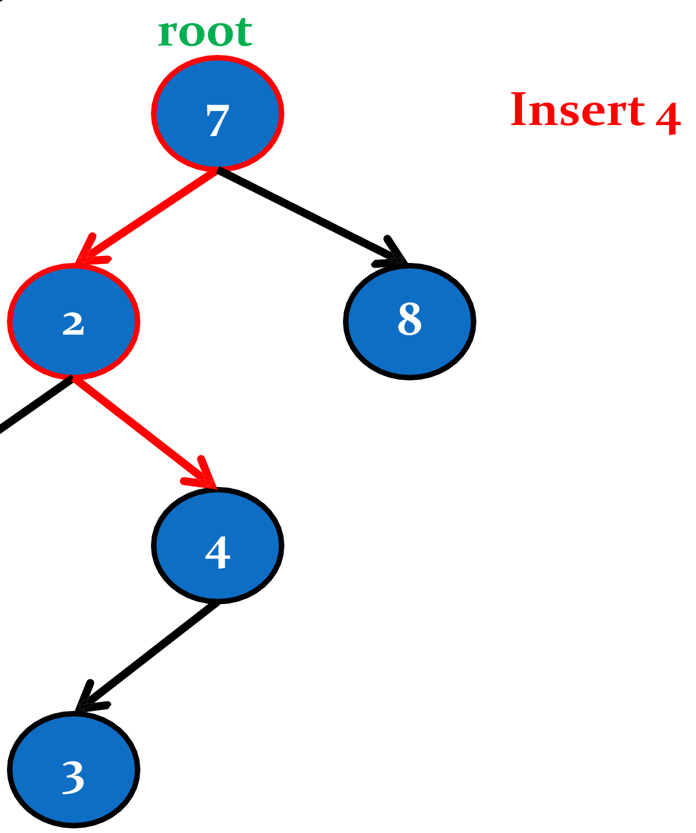
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



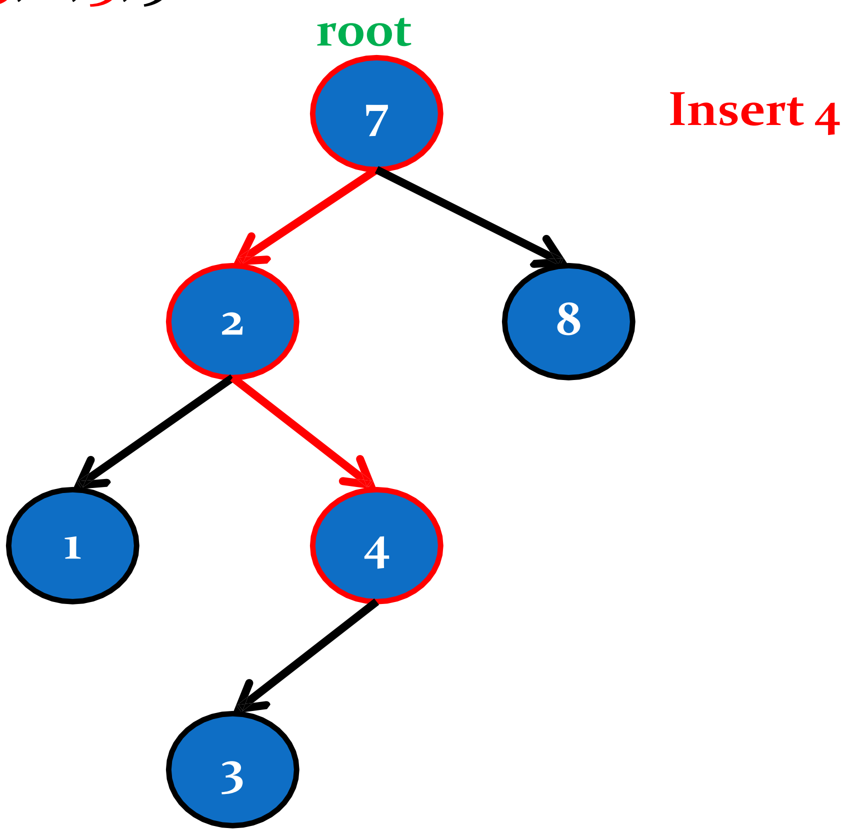
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



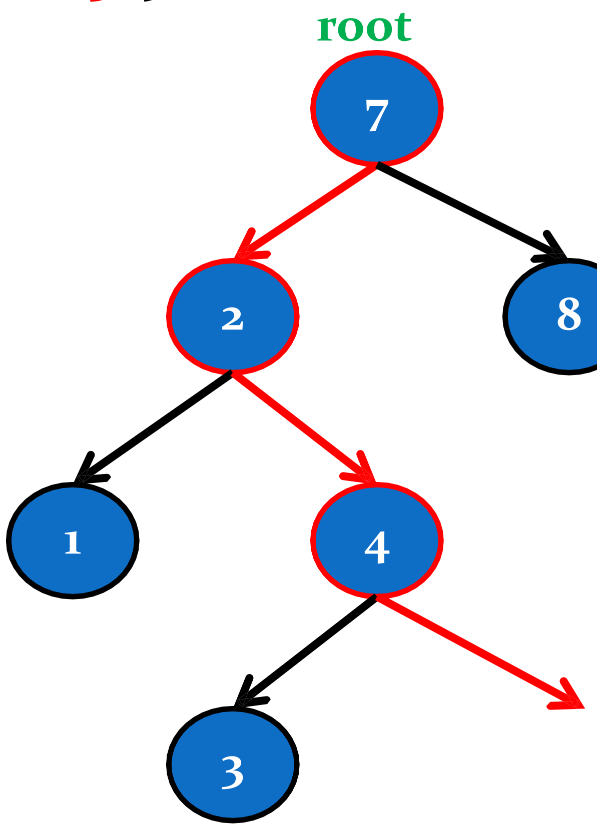
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



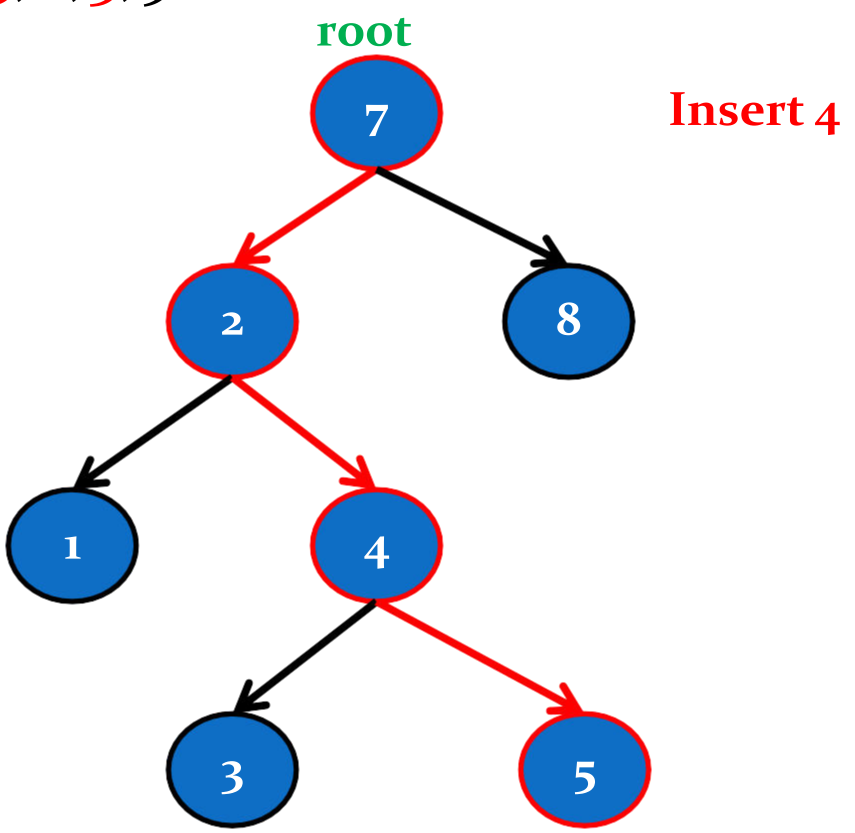
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



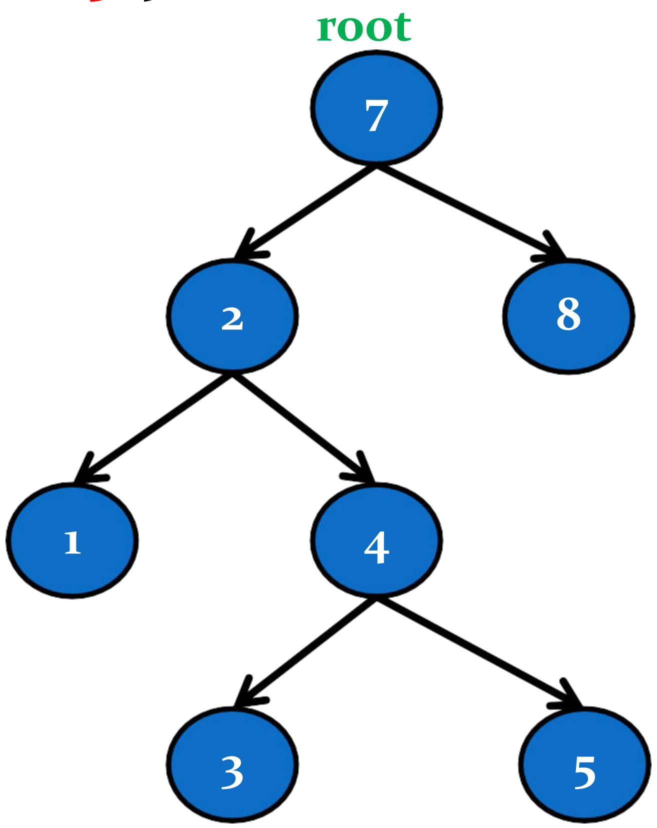
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



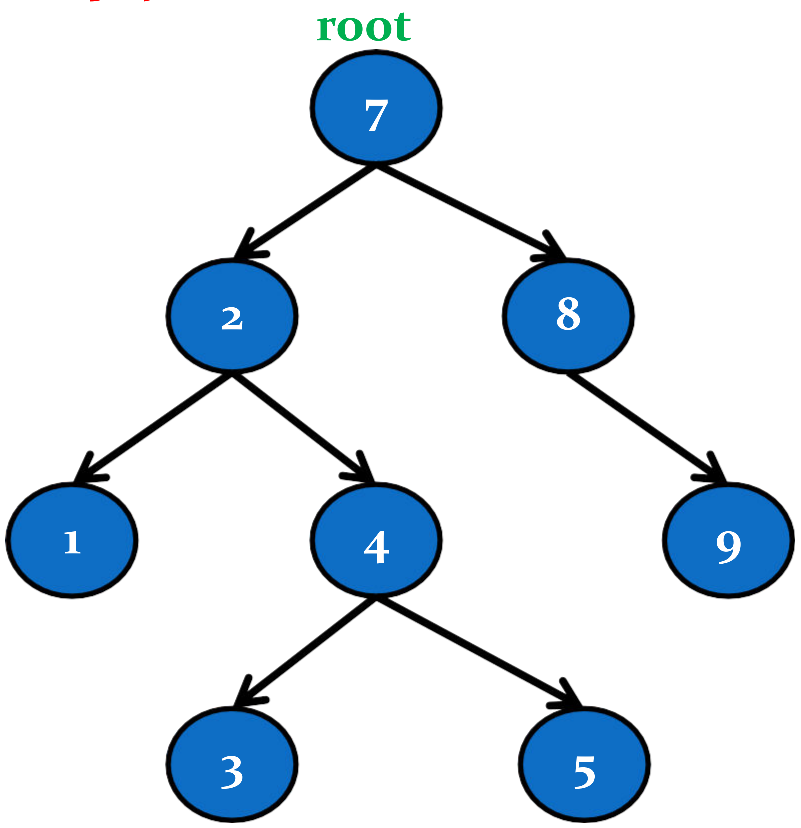
Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



Construction of binary search tree

Insert 7,2,1,4,3,8,5,9



Construct a binary search tree

BST : INSERT(5)

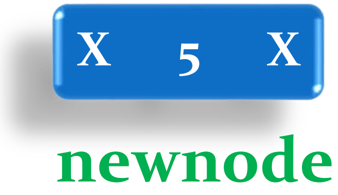
Newnode=malloc(newnode)

Newnode.data=item

Newnode.leftl=NULL

Newnode.rightl=NULL

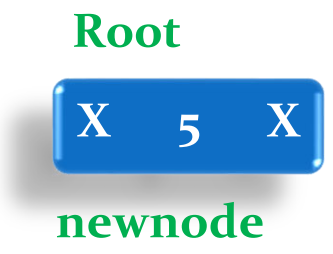
Root =NULL



BST : INSERT(5)

Root =NULL

```
If(root=NULL)
    Root=newnode
```



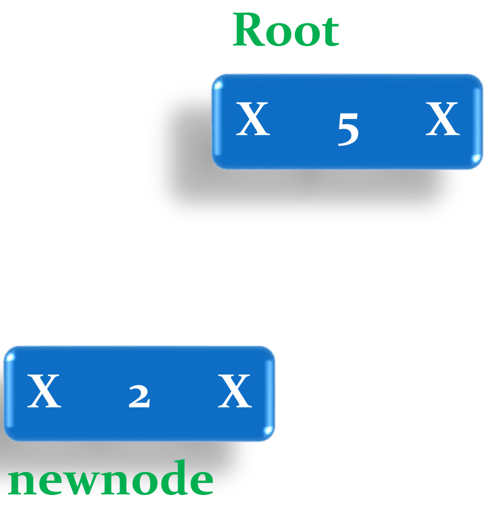
BST : INSERT(2)

Newnode=malloc (newnode)

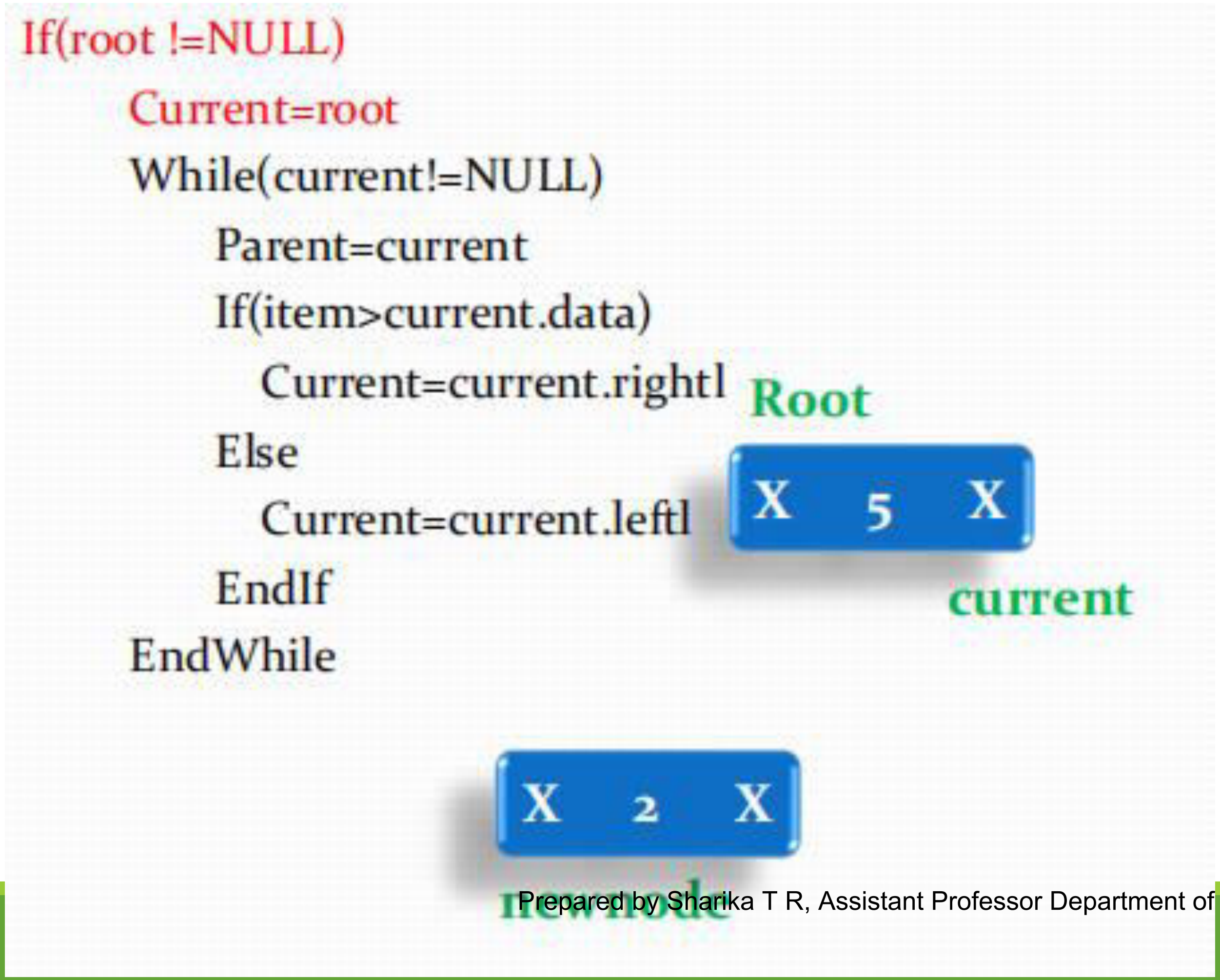
Newnode.data=item

Newnode.leftl=NULL

Newnode.rightl=NULL



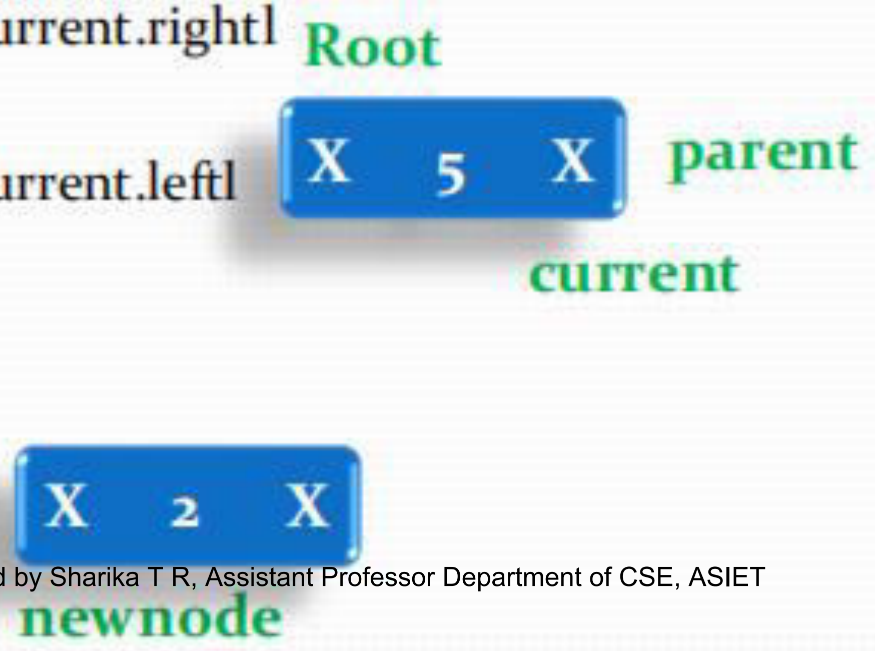
BST : INSERT(2)



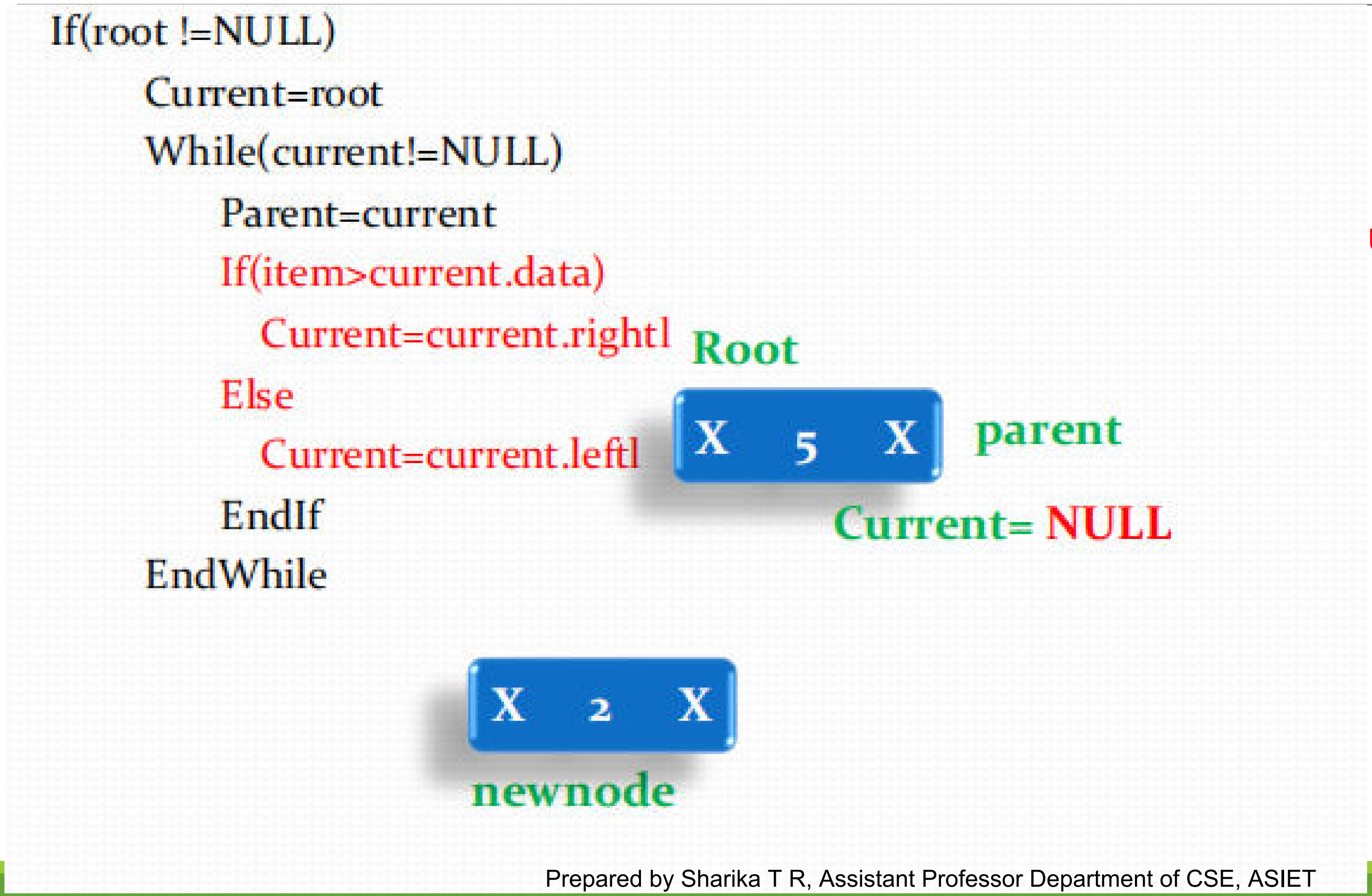
```
If(root !=NULL)
    Current=root
    While(current!=NULL)
        Parent=current
        If(item>current.data)
            Current=current.rightl
        Else
            Current=current.leftl
        EndIf
    EndWhile
```

BST : INSERT(2)

```
If(root !=NULL)
        Current=root
        While(current!=NULL)
              Parent=current
              If(item>current.data)
                 Current=current.rightl
              Else
                 Current=current.leftl
              EndIf
        EndWhile
```



BST : INSERT(2)

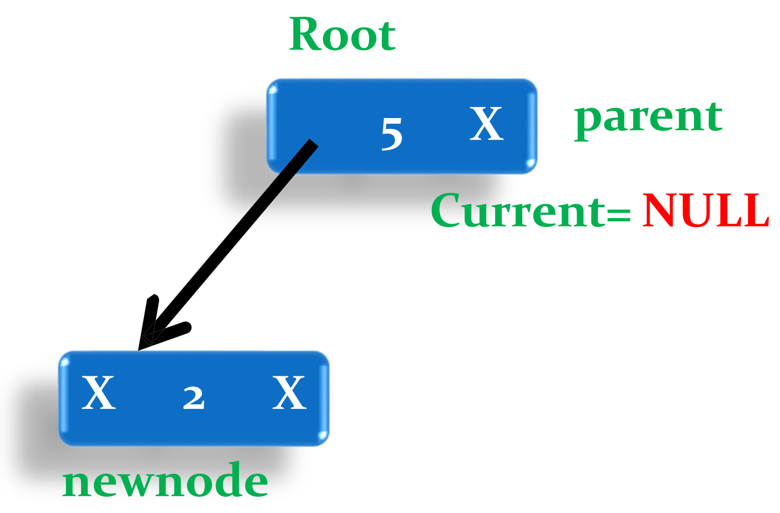


```
If(root !=NULL)
    Current=root
    While(current!=NULL)
        Parent=current
        If(item>current.data)
          Current=current.rightl
        Else
          Current=current.leftl
        EndIf
    EndWhile
```

urrent.data)

BST : INSERT(2)

```
If(item>parent.data)
    Parent.rightl=newnode
Else
    Parent.leftl=newnode
EndIf
```



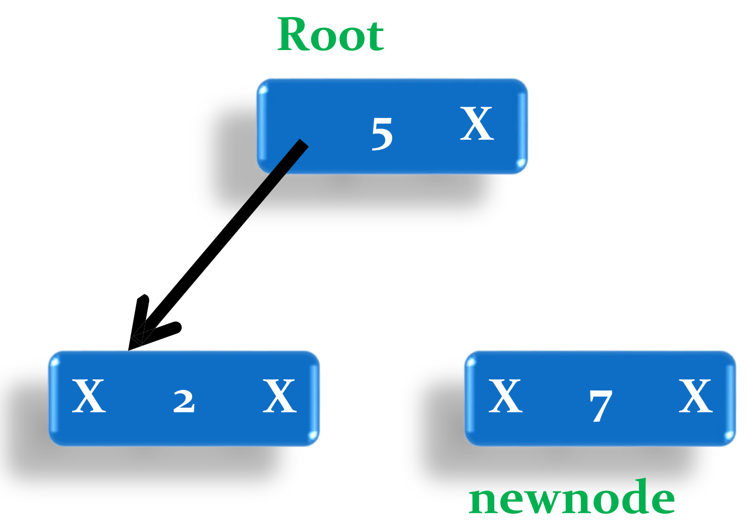
BST : INSERT(7)

Newnode=malloc (newnode)

Newnode.data=item

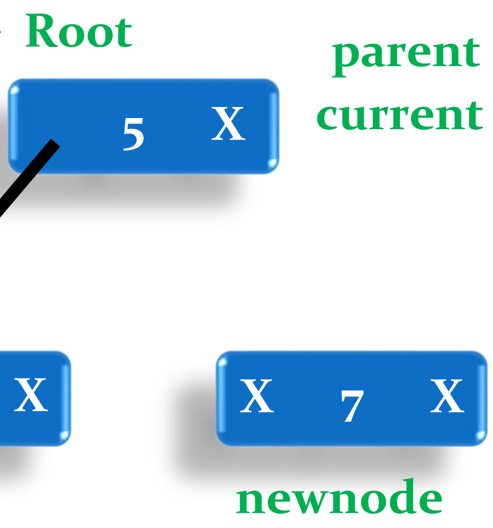
Newnode.leftl=NULL

Newnode.rightl=NULL



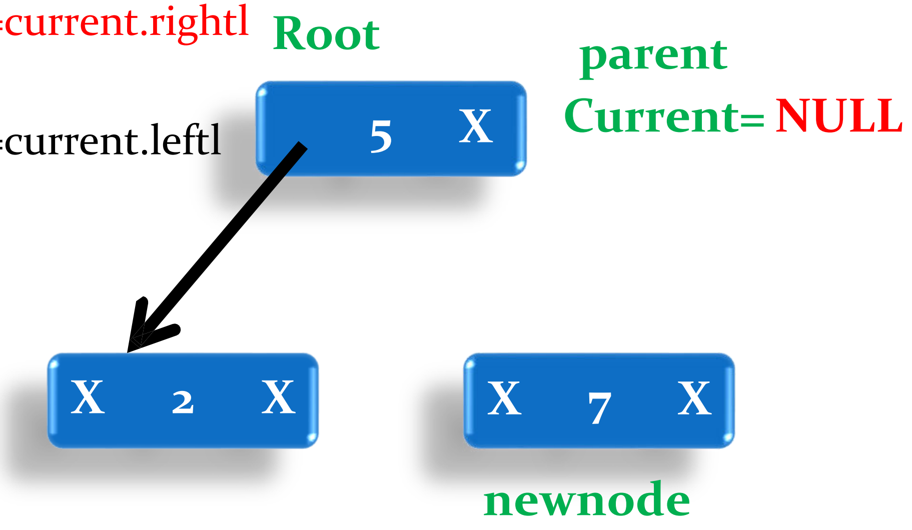
BST : INSERT(7)

```
If(root !=NULL)
    Current=root
    While(current!=NULL)
        Parent=current
        If(item>current.data)
            Current=current.rightl
        Else
            Current=current.leftl
        EndIf
    EndWhile
```



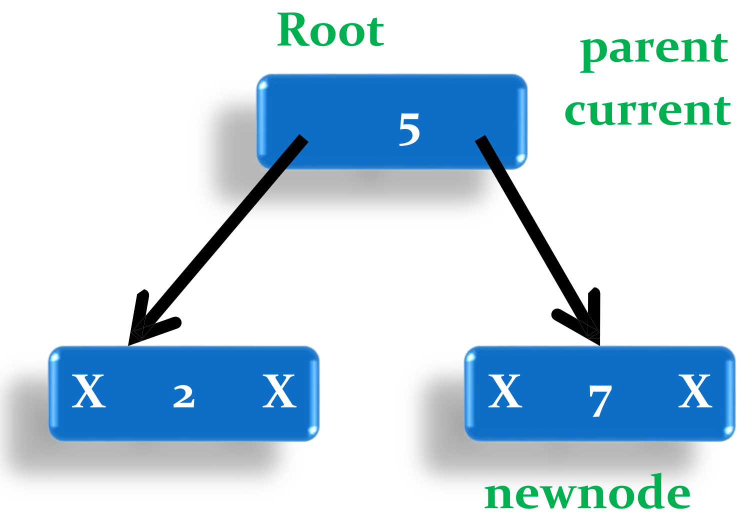
BST : INSERT(7)

```
If(root !=NULL)
    Current=root
    While(current!=NULL)
        Parent=current
        If(item>current.data)
            Current=current.rightl
        Else
            Current=current.leftl
        EndIf
    EndWhile
```



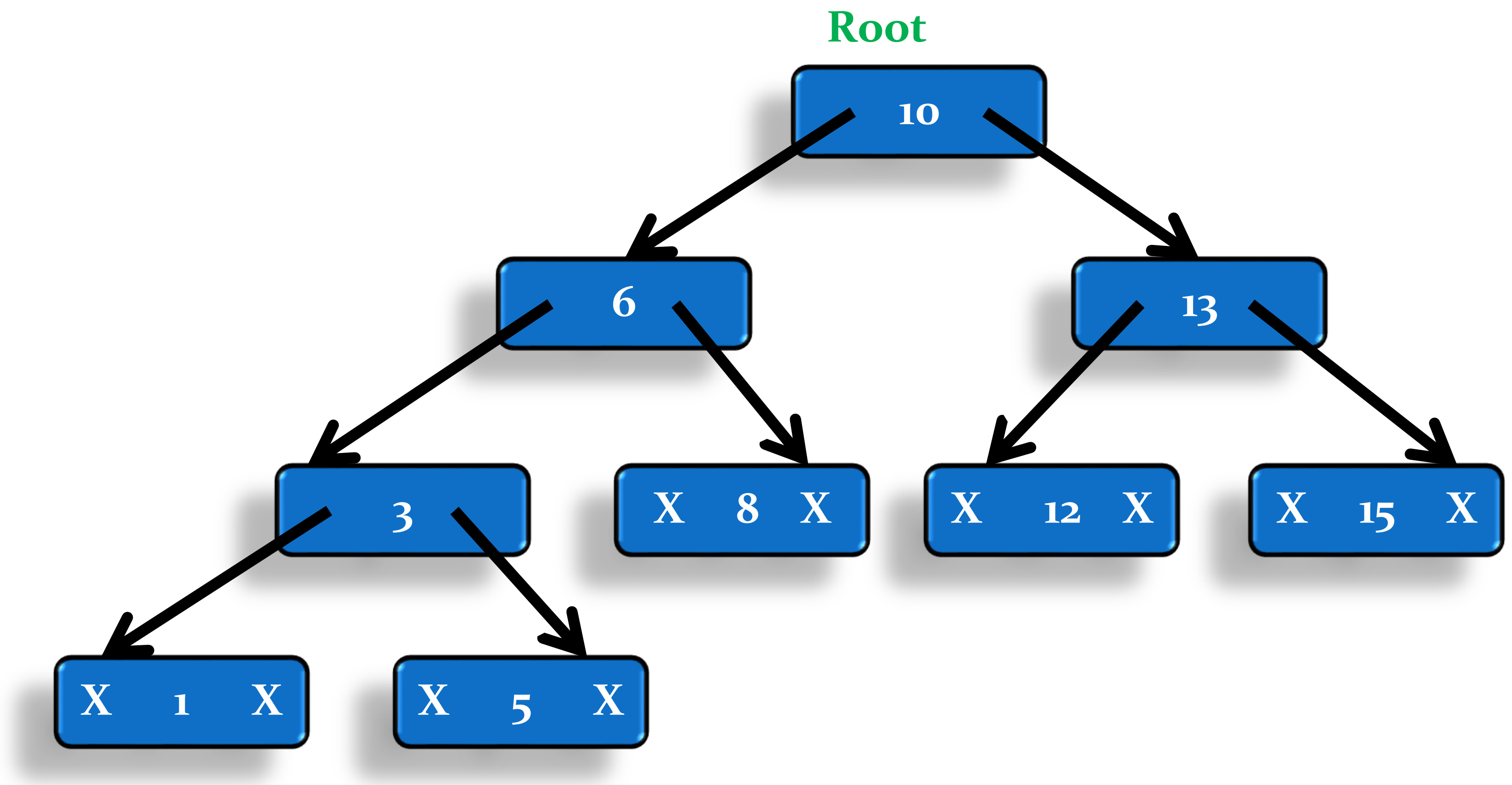
BST : INSERT(7)

```
If(item>parent.data)
    Parent.rightl=newnode
Else
    Parent.leftl=newnode
EndIf
```



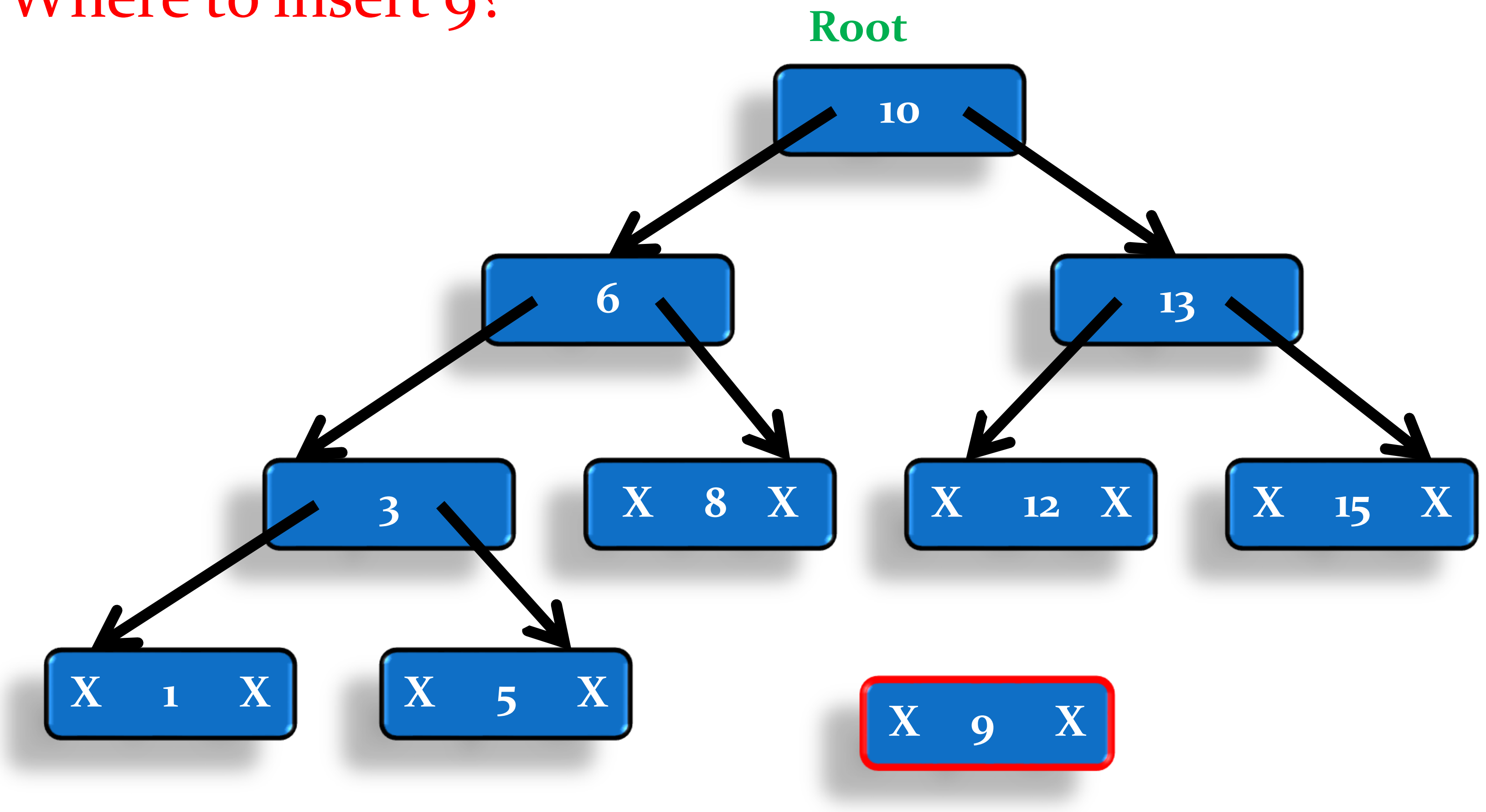
Insert an element into a binary search tree

BST : Algorithm



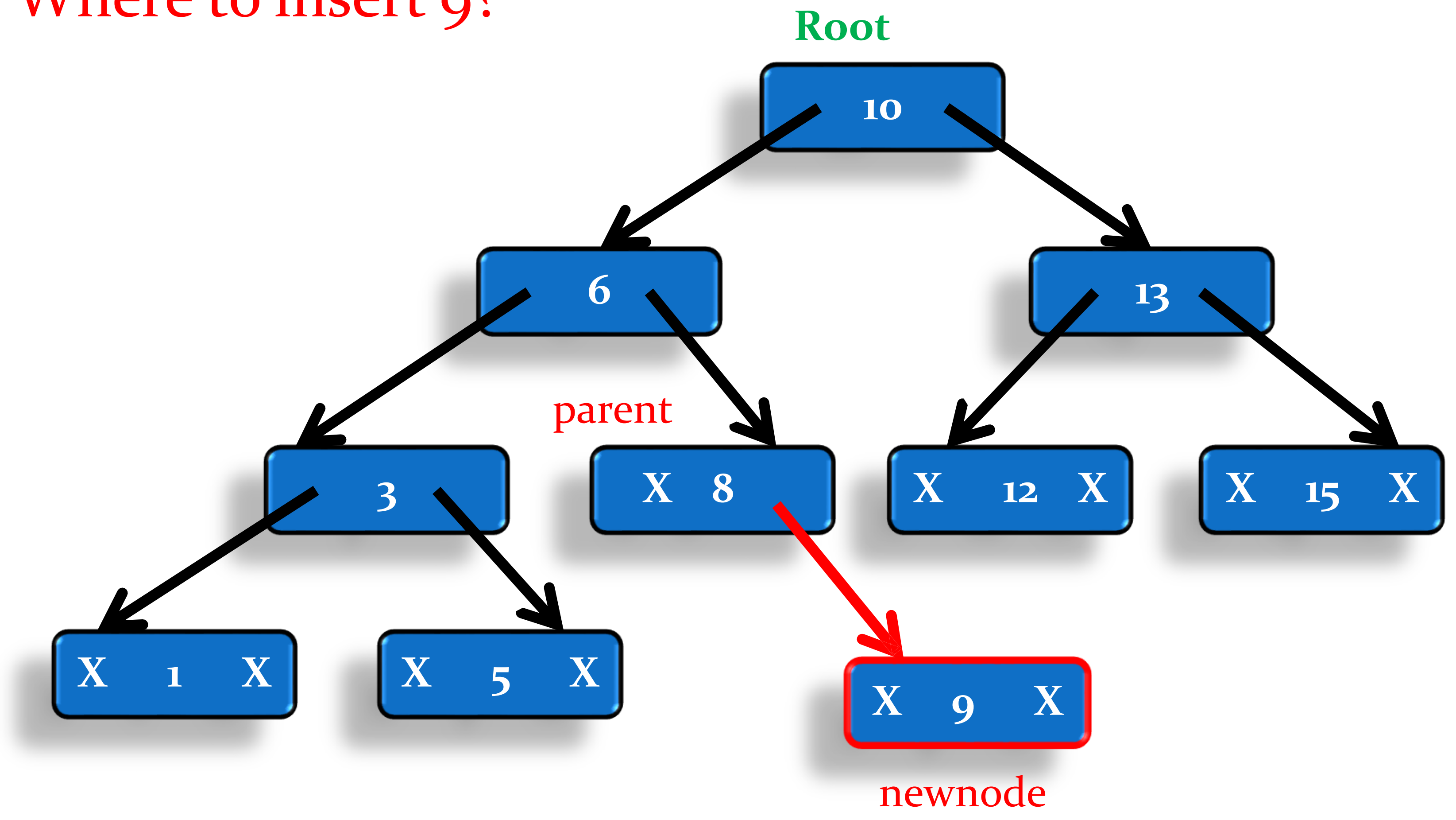
BST : insert (9)

Where to insert 9?



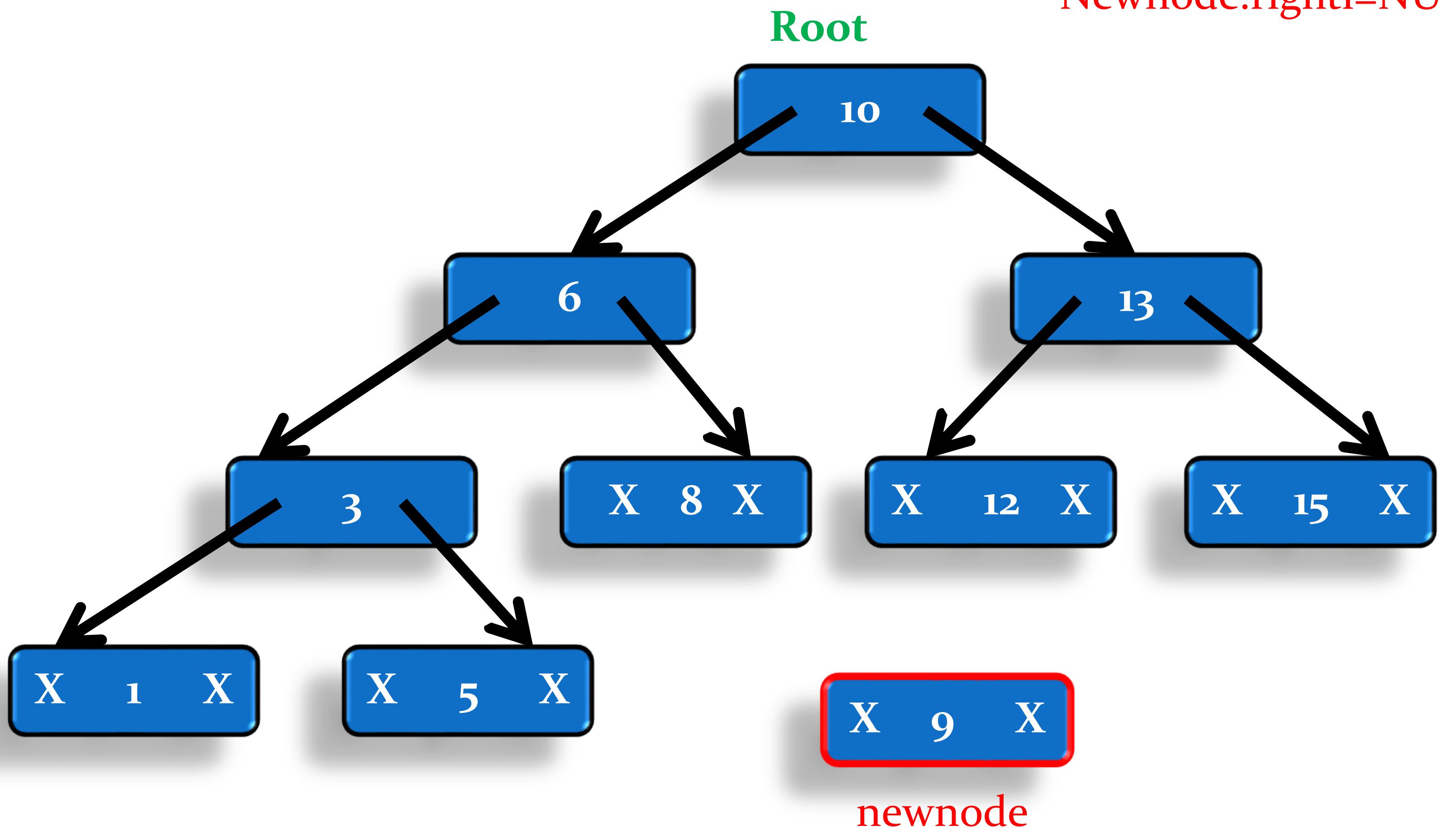
BST : insert (9)

Where to insert 9?



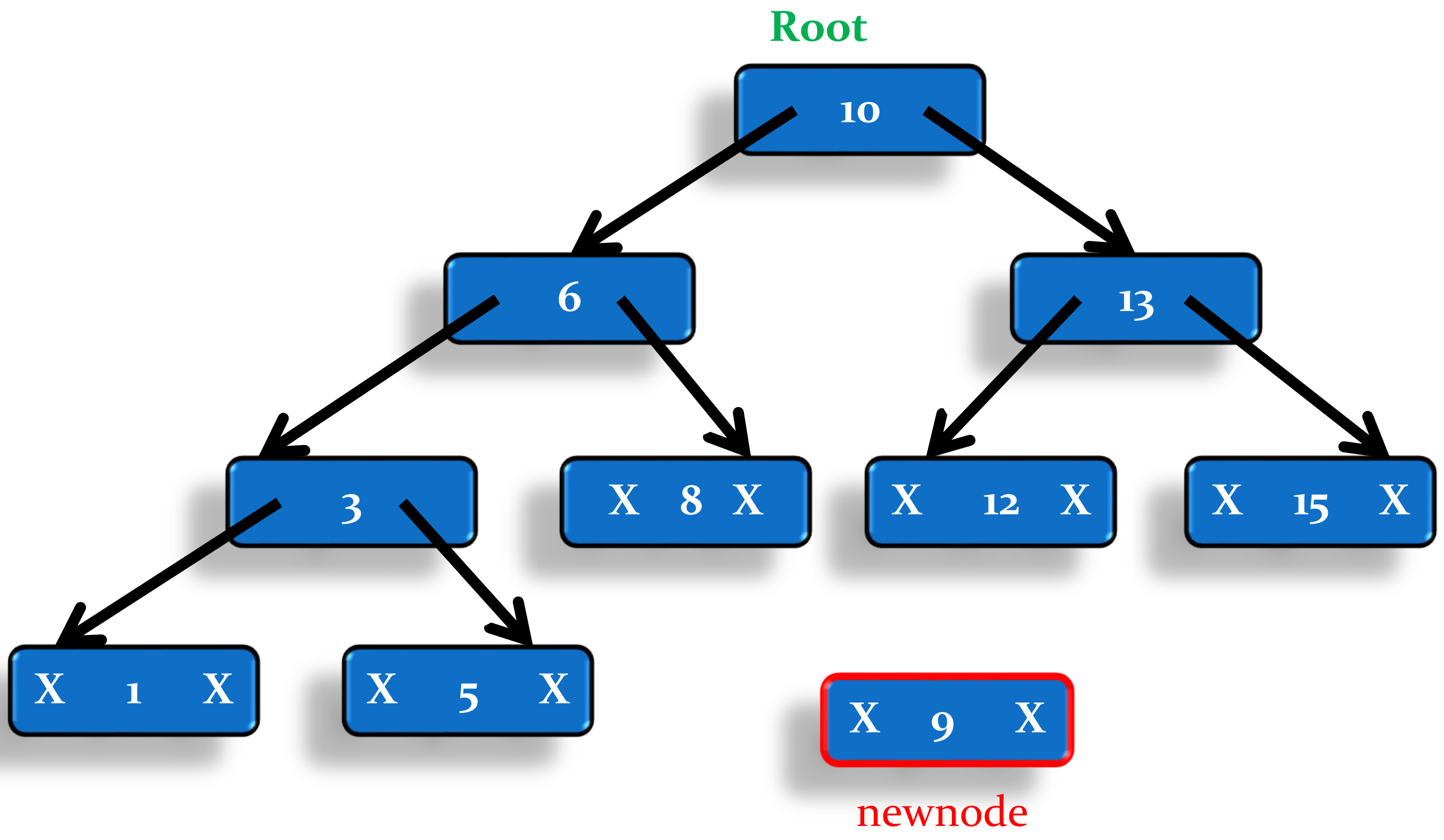
BST : insert (9)

Newnode=malloc (newnode)
Newnode.data=item
Newnode.leftl=NULL
Newnode.rightl=NULL



BST : insert (9)

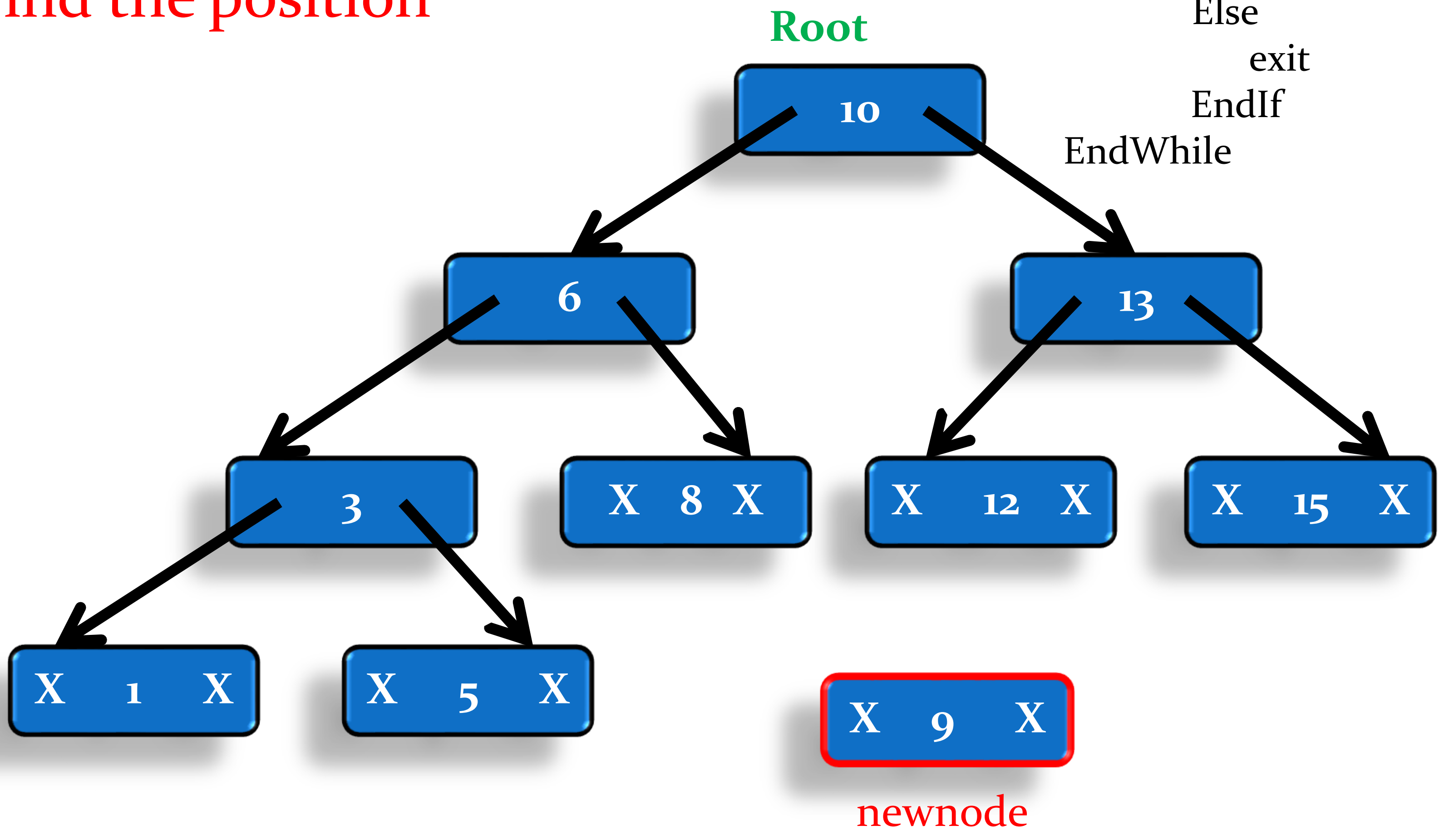
```
If(root=NULL)
      Root=newnode
```



BST : insert (9)

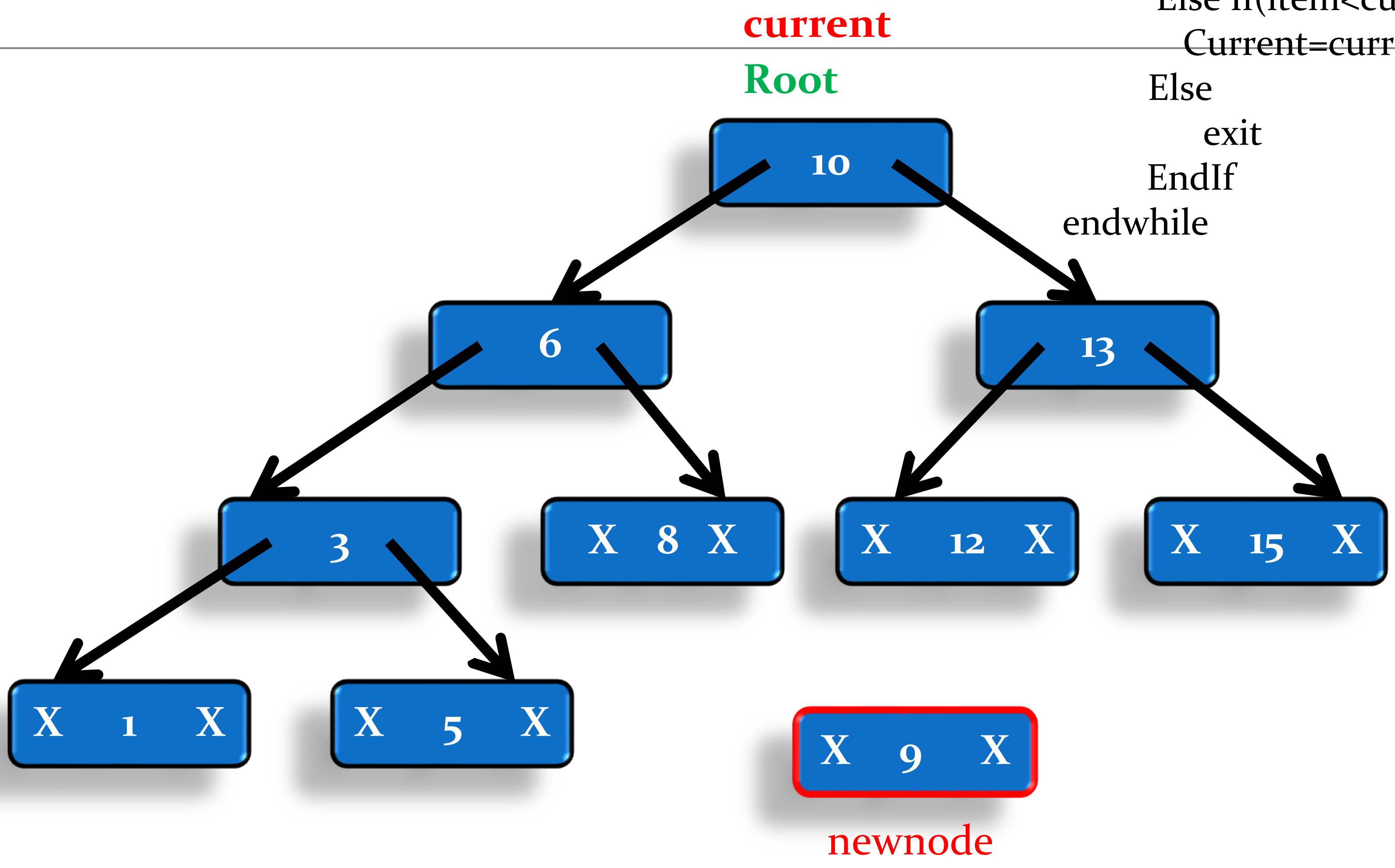
Find the position

```
Current=root
While(current!=NULL)
        Parent=current
        If(item>current.data)
           Current=current->rightl
        Else If(item<current.data)
           Current=current->leftl
        Else
             exit
        EndIf
EndWhile
```



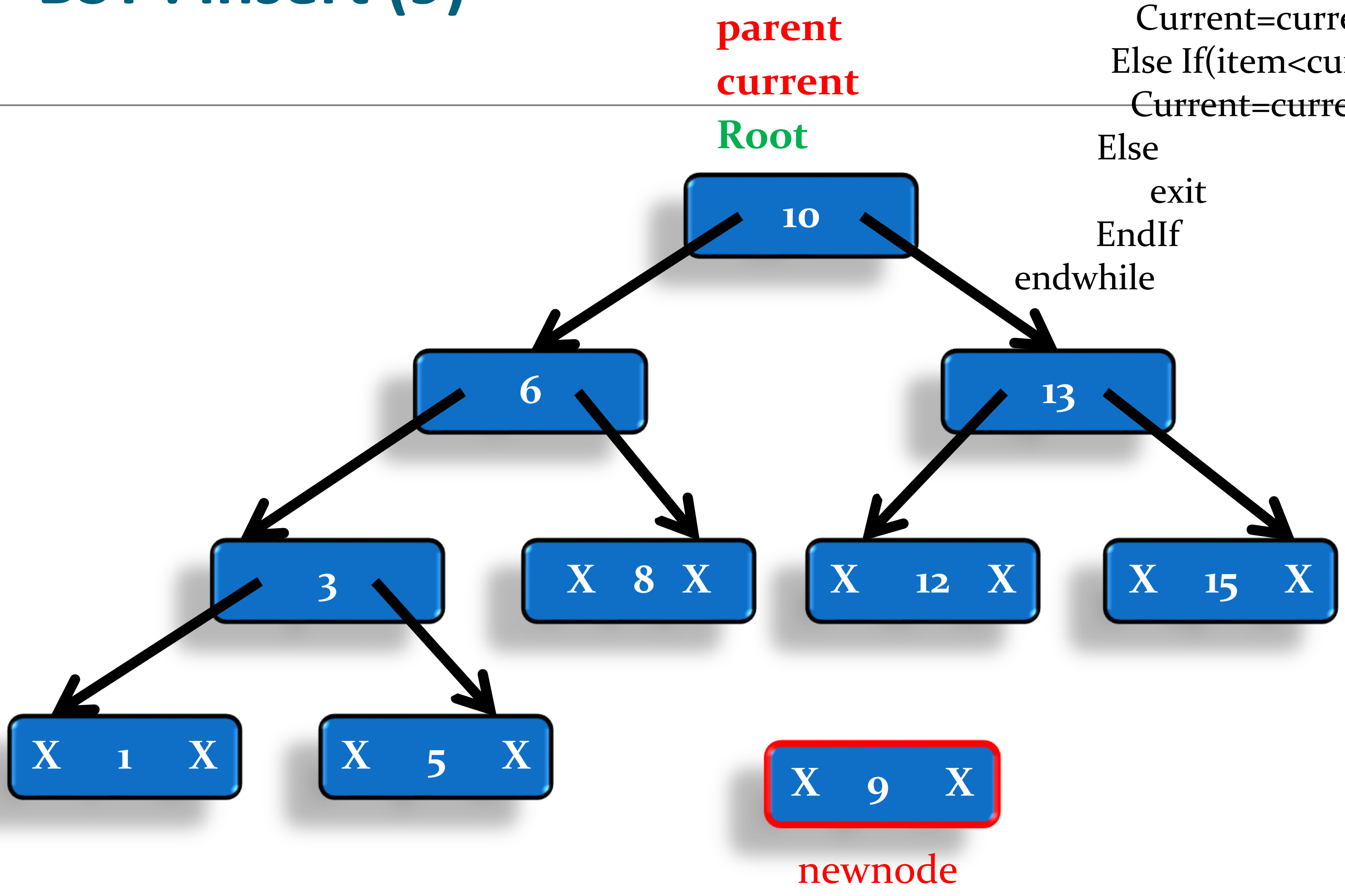
BST : insert (9)

```
Current=root
While(current!=NULL)
        Parent=current
        If(item>current.data)
            Current=current->rightl
        Else If(item<current.data)
            Current=current->leftl
        Else
            exit
        EndIf
endwhile
```



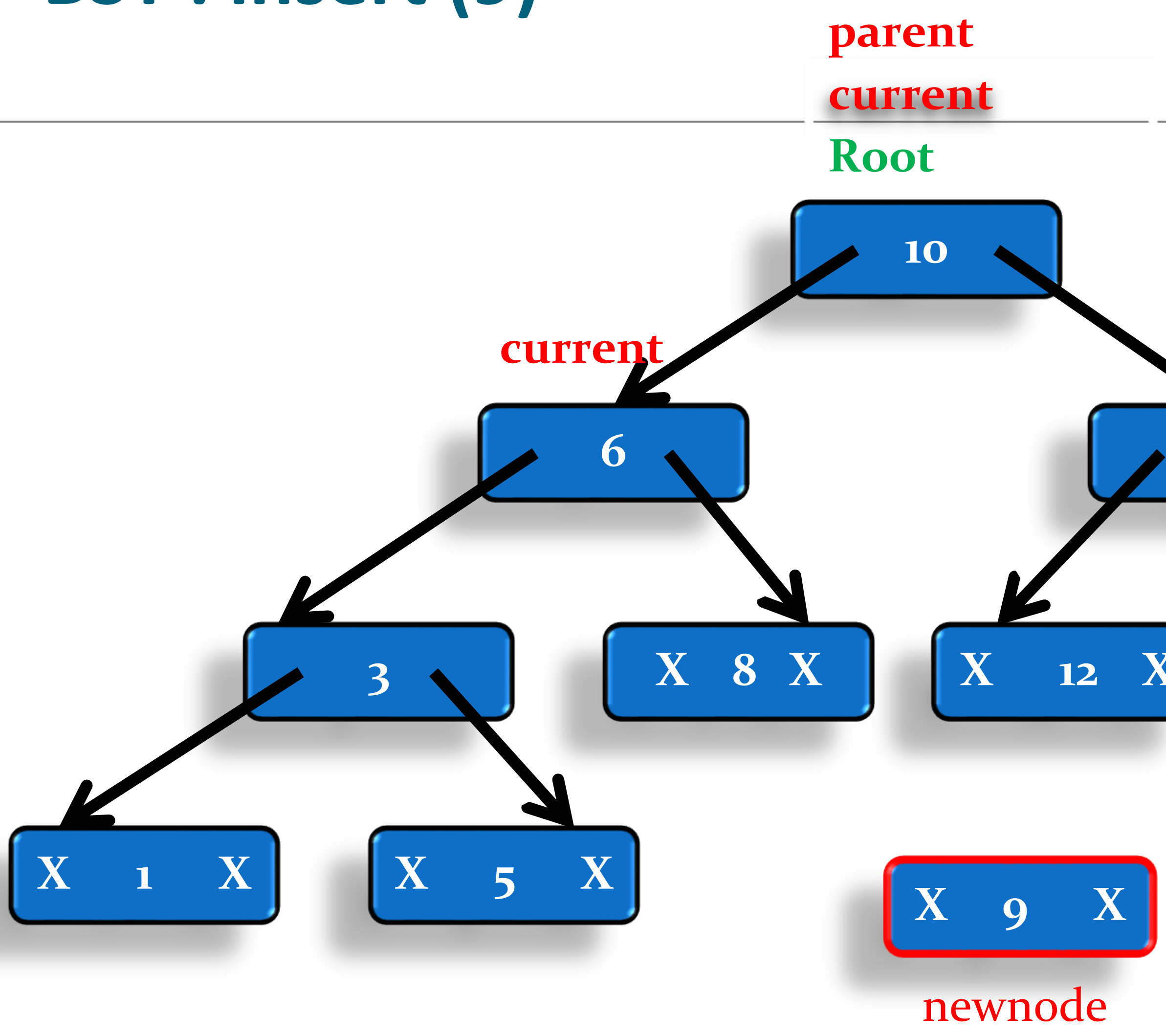
BST : insert (9)

```
Current=root
While(current!=NULL)
        Parent=current
        If(item>current.data)
           Current=current->rightl
         Else If(item<current.data)
           Current=current->leftl
        Else
            exit
        EndIf
endwhile
```



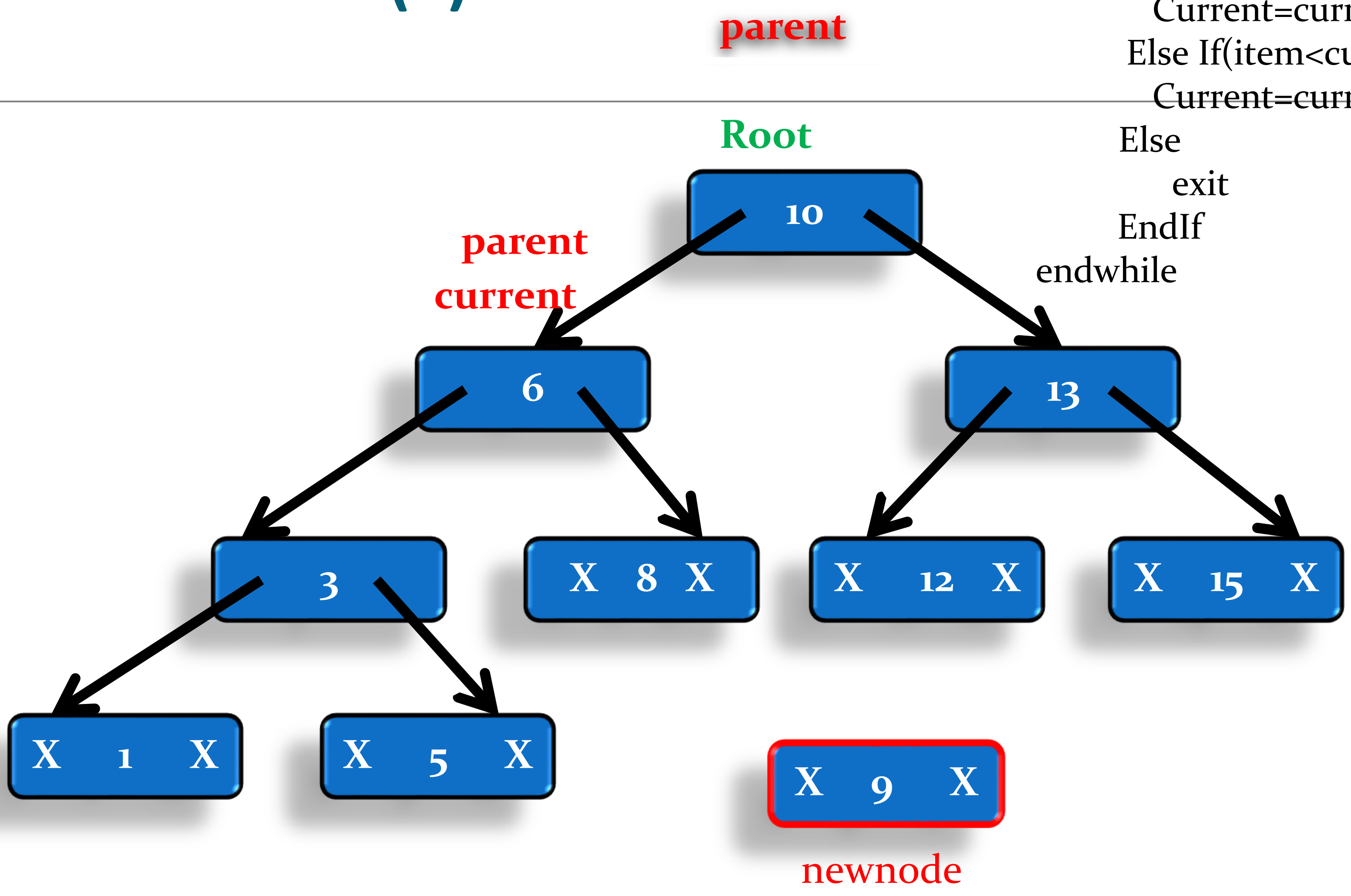
BST : insert (9)

```
Current=root
While(current!=NULL)
        Parent=current
        If(item>current.data)
          Current=current->rightl
        Else If(item<current.data)
          Current=current->leftl
        Else
            exit
        EndIf
endwhile
```



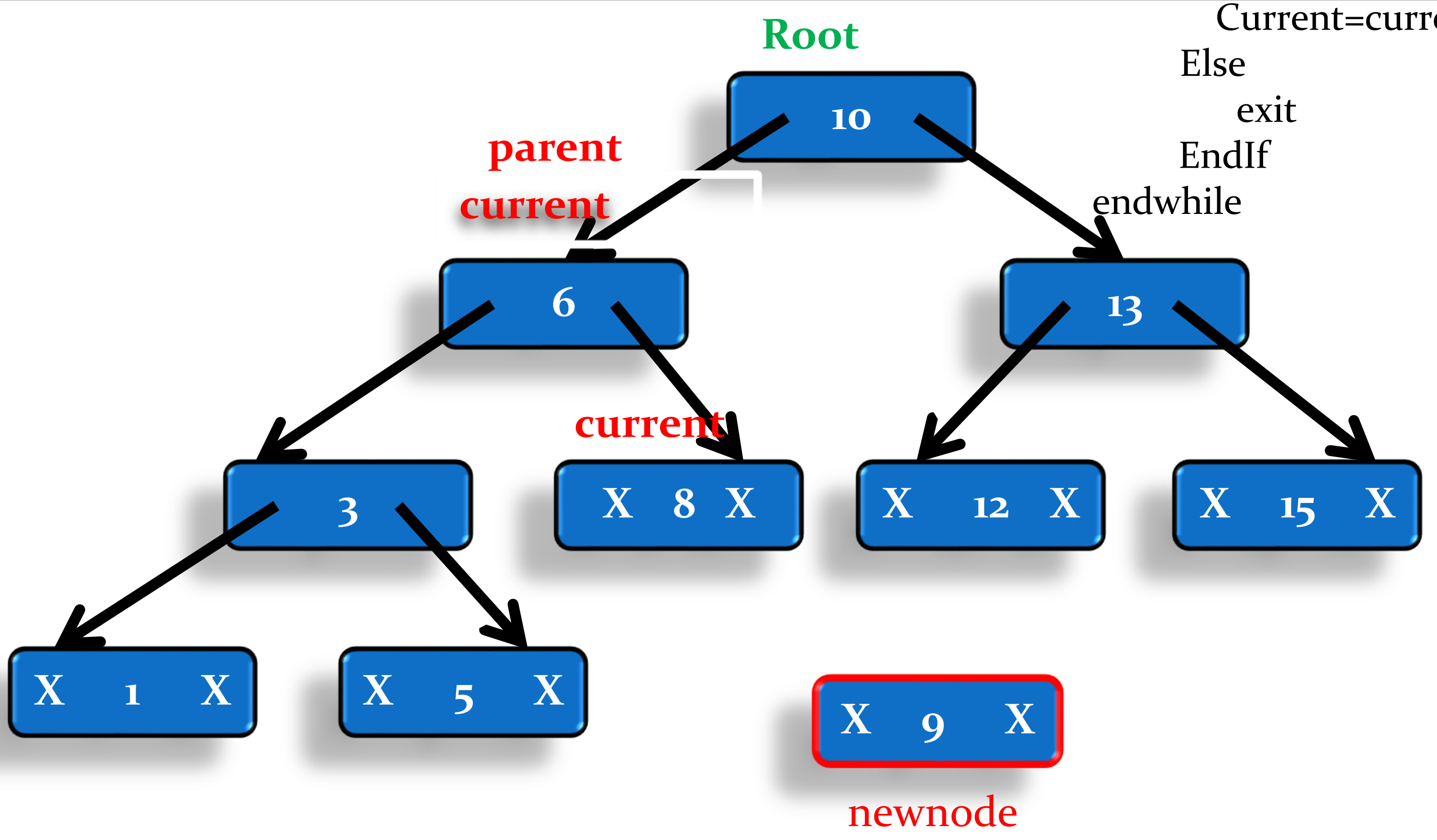
BST : insert (9)

```
Current=root
While(current!=NULL)
        Parent=current
        If(item>current.data)
            Current=current->rightl
         Else If(item<current.data)
            Current=current->leftl
        Else
            exit
        EndIf
endwhile
```



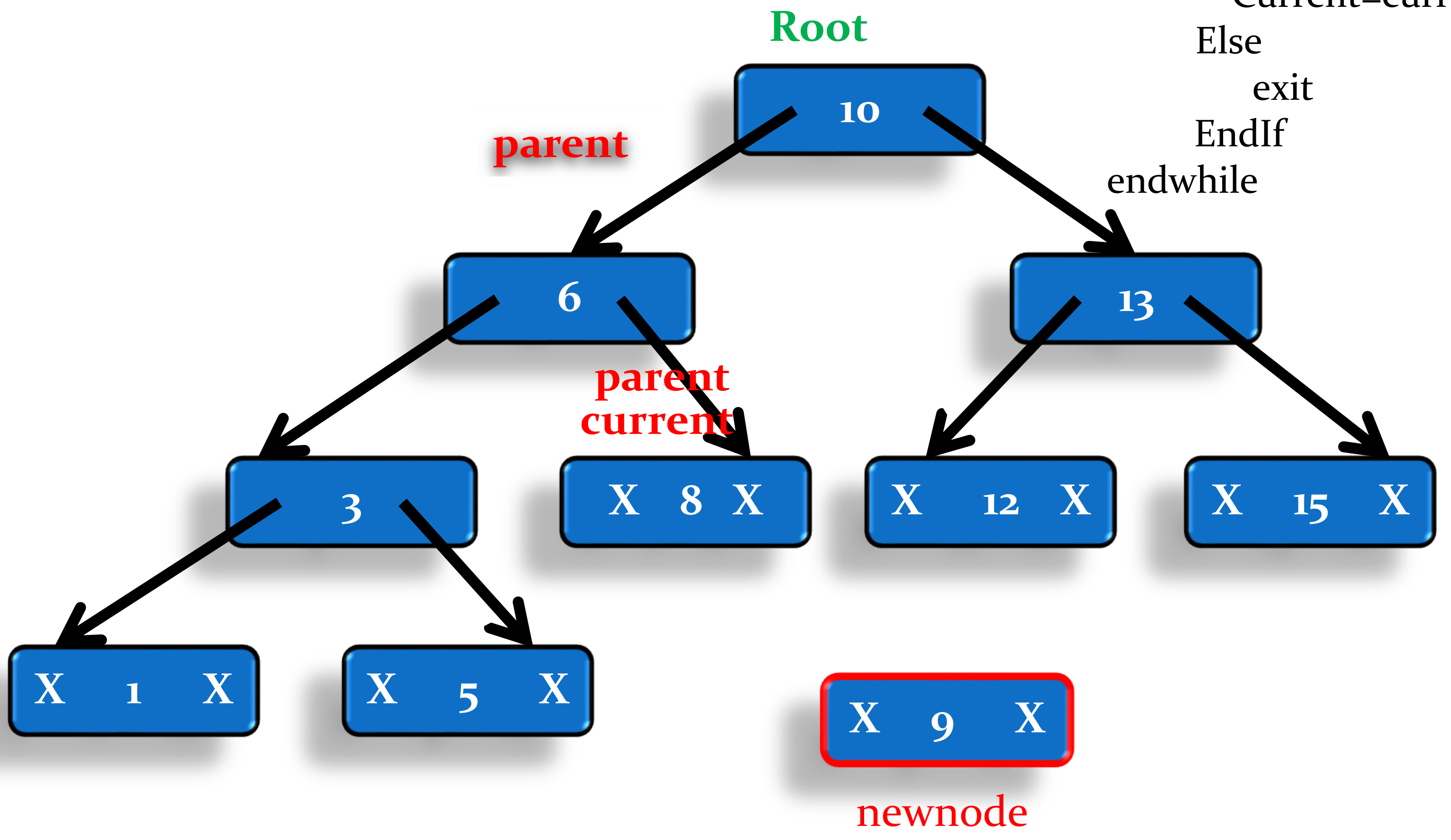
BST : insert (9)

```
Current=root
While(current!=NULL)
        Parent=current
        If(item>current.data)
          Current=current->rightl
         Else If(item<current.data)
          Current=current->leftl
        Else
            exit
        EndIf
endwhile
```



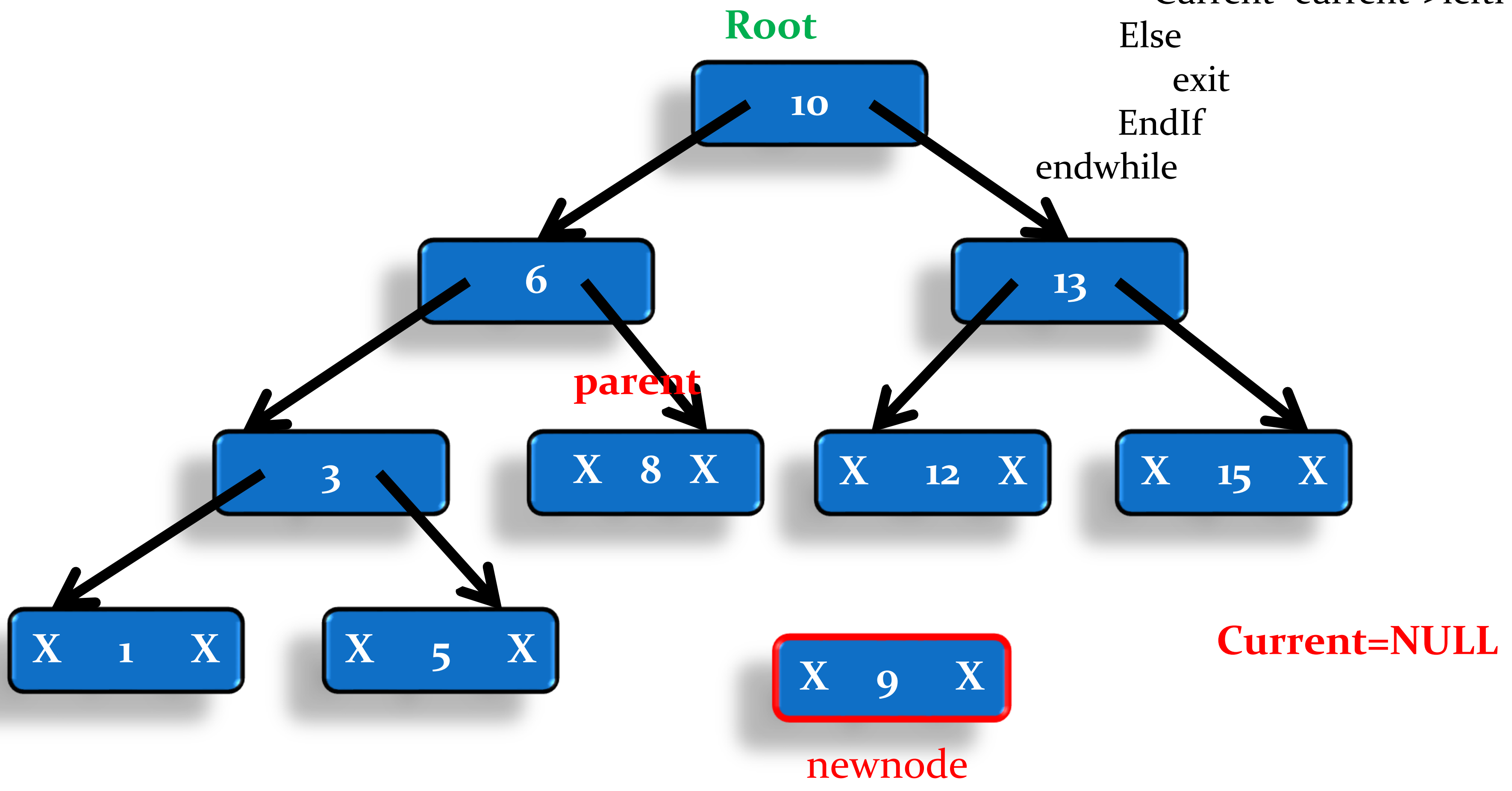
BST : insert (9)

```
Current=root
While(current!=NULL)
        Parent=current
        If(item>current.data)
            Current=current->rightl
         Else If(item<current.data)
            Current=current->leftl
        Else
              exit
        EndIf
endwhile
```



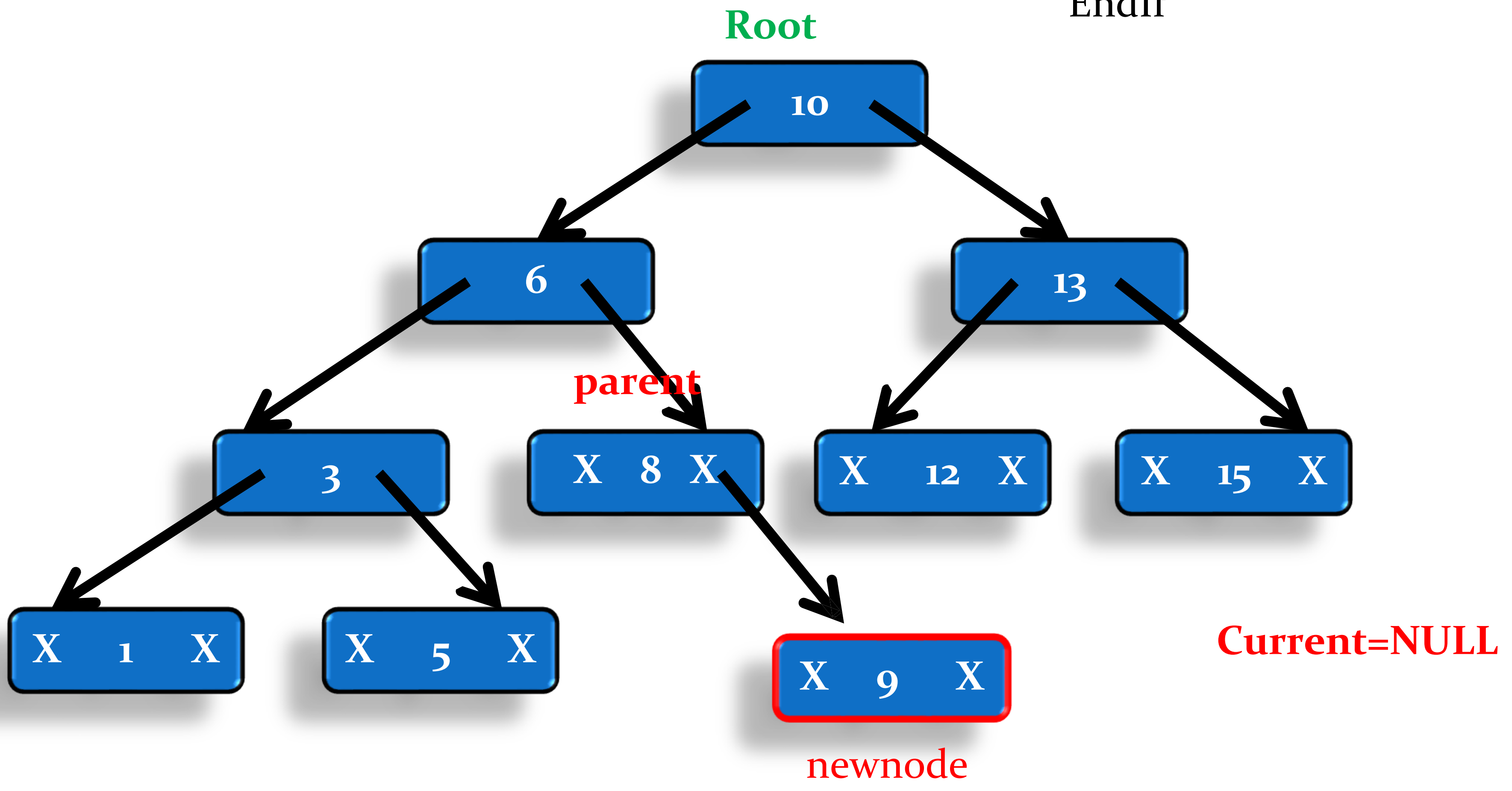
BST : insert (9)

```
Current=root
While(current!=NULL)
        Parent=current
        If(item>current.data)
          Current=current->rightl
        Else If(item<current.data)
          Current=current->leftl
        Else
            exit
        EndIf
endwhile
```

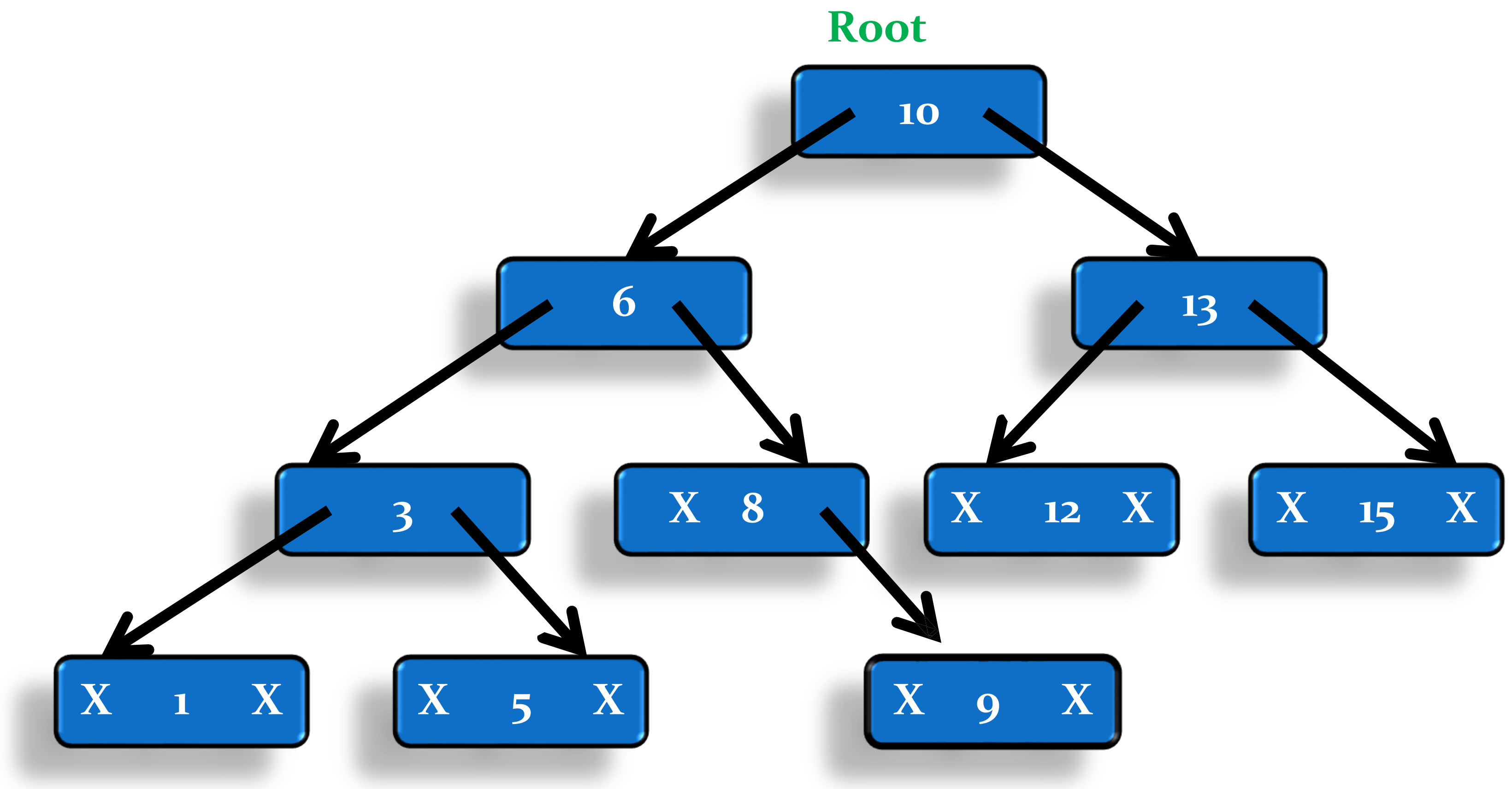


BST : insert (9)

```
If(item>parent.data)
    Parent.rightl=newnode
Else
    Parent.leftl=newnode
EndIf
```



BST : 9 inserted



BST : INSERTION

Algorithm **BST_INSERT(item)**

Input: Item is the data component of a node that has to be inserted.

Output: Item inserted in to the proper place of the tree.

Data structure: Linked structure

Steps:

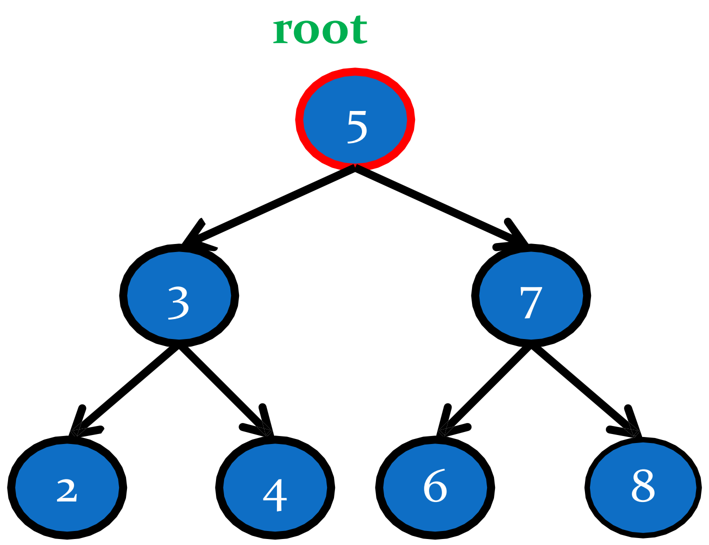
```
1.    Newnode=malloc(newnode)
2.    Newnode.data=item
3.    Newnode.leftl=NULL
4.    Newnode.rightl=NULL
5.    If(root=NULL)
      1.    Root=newnode
6.    Else
      1.    Current=root
      2.    While(current!=NULL)
            1.    Parent=current
            2.    If(item>current.data)
                  1.    Current=current->rightl
            3.    Else If(item<current.data)
                  1.    Current=current->leftl
            4.    Else
                  1.    Print("duplicate node");
                  2.    exit
            5.    EndIf
      3.    EndWhile
      4.    If(item>parent.data)
            1.    Parent.rightl=newnode
      5.    Else
            1.    Parent.leftl=newnode
      6.    EndIf
7.    EndIf
8.    Stop.
```

Searching in a binary search tree

- Searching for a data in a binary search tree is much faster than in arrays or linked lists.
- So the applications where frequent searching operations are to be performed, this data structure is used.

- To search for an item
  1. start from the root node,
  2. if item < root node data
     1. proceed to its left child.
  3. If item > root node data
     1. proceed to its right child.
  4. Continue step 2 and 3 till the item is found or we reach to a dead end (leaf node)

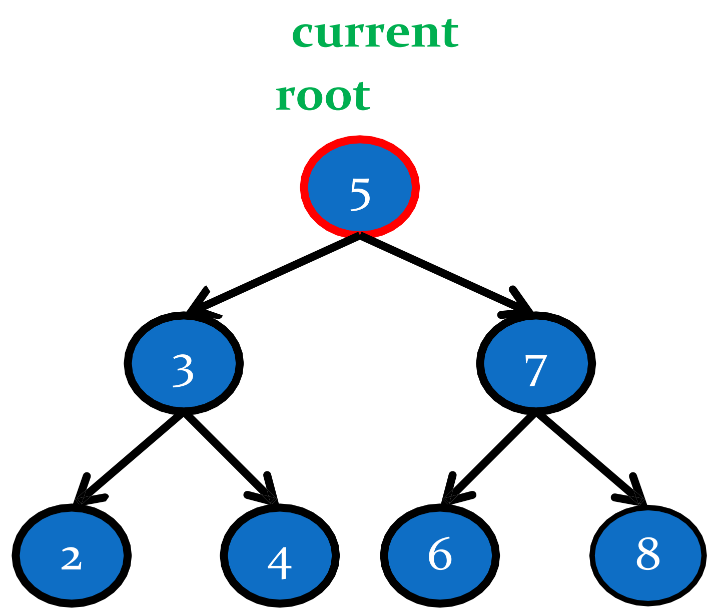
Search 6



Steps:

Search 6

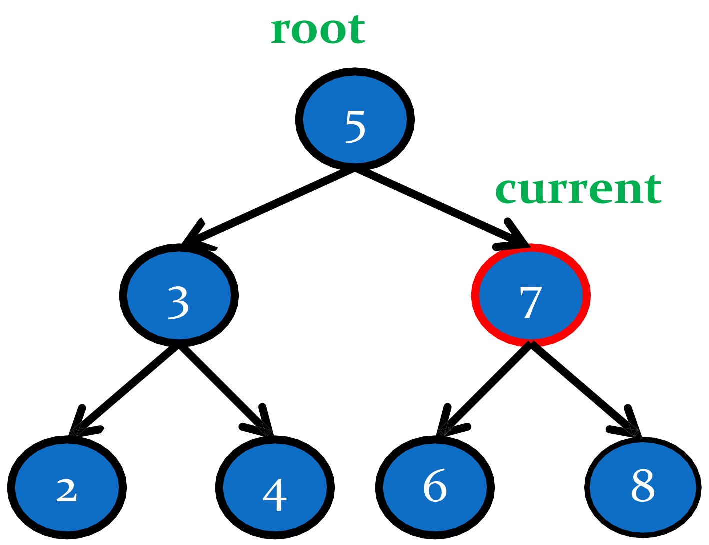
```
1. Current=root
2. While(current!=NULL and current.data != item)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```



Search 6

Steps:

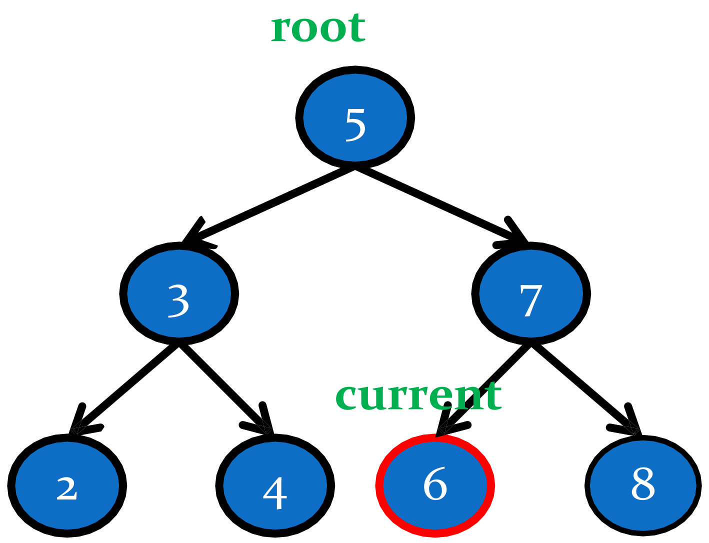
```
1. Current=root
2. While(current!=NULL and current.data != item)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. Exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```



Search 6

Steps:

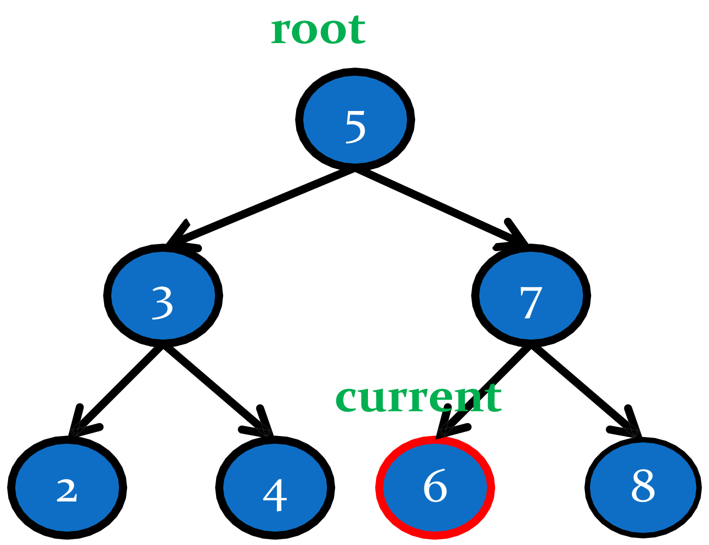
```
1. Current=root
2. While(current!=NULL and current.data != item)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```



Search 6

Steps:

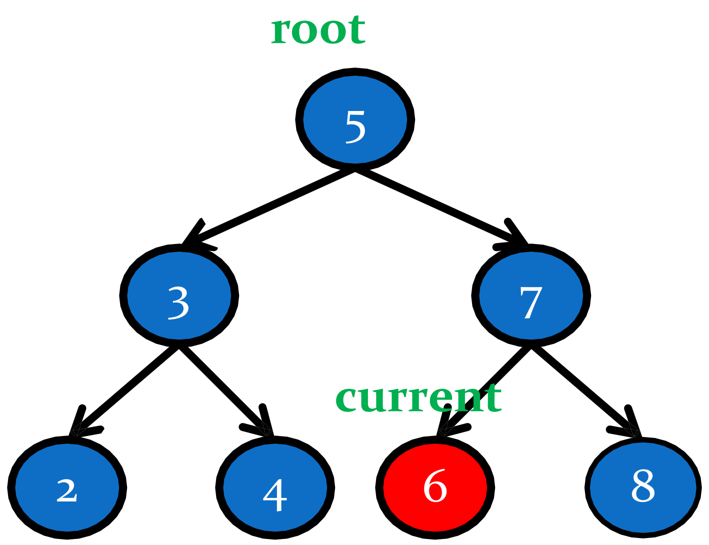
```
1.  Current=root
2.  While(current!=NULL and current.data != item)
    1.  If( item>current.data)
        1.  Current=current.rlink
    2.  Else
        1.  Current=current.llink
    3.  EndIf
3.  EndWhile
4.  If(current=NULL)
    1.  Print("Item notFound")
    2.  exit
5.  Else
    1.  Print("Item Found")
6.  EndIf
7.  Return (current)
```



Search 6

Steps:

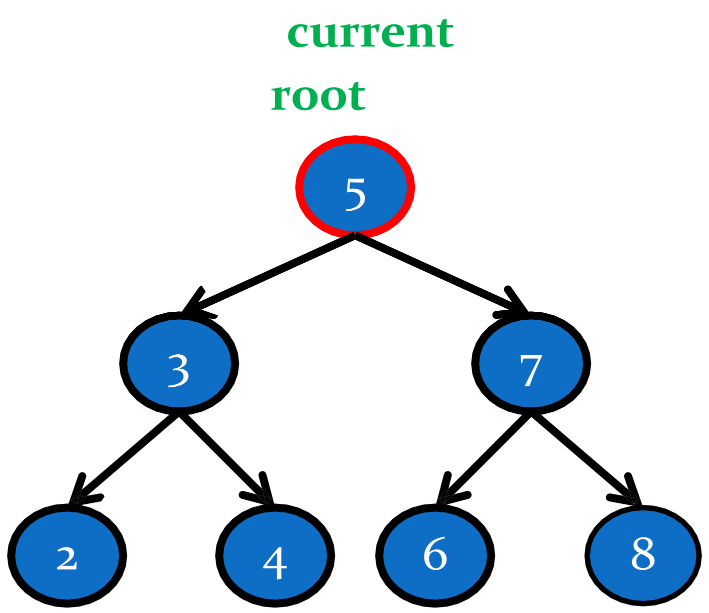
```
1. Current=root
2. While(current!=NULL and **current.data != item**)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```



Again Search 9

Steps:

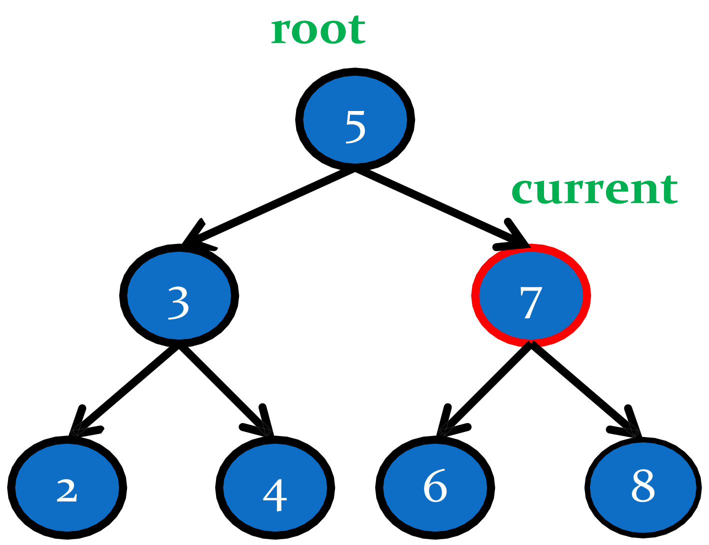
```
1.  Current=root
2.  While(current!=NULL and current.data != item)
    1.  If( item>current.data)
        1.  Current=current.rlink
    2.  Else
        1.  Current=current.llink
    3.  EndIf
3.  EndWhile
4.  If(current=NULL)
    1.  Print("Item notFound")
    2.  exit
5.  Else
    1.  Print("Item Found")
6.  EndIf
7.  Return (current)
```



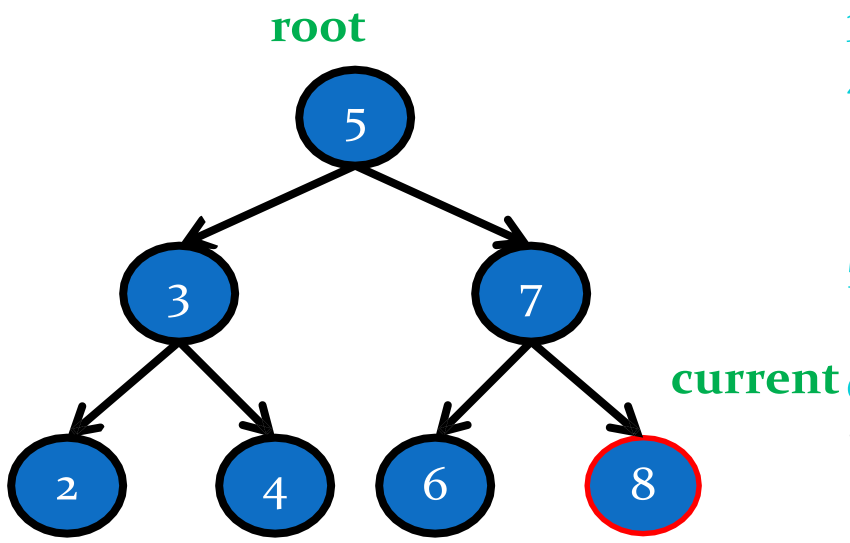
Search 9

Steps:

```
1. Current=root
2. While(current!=NULL and current.data != item)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```



Search 9



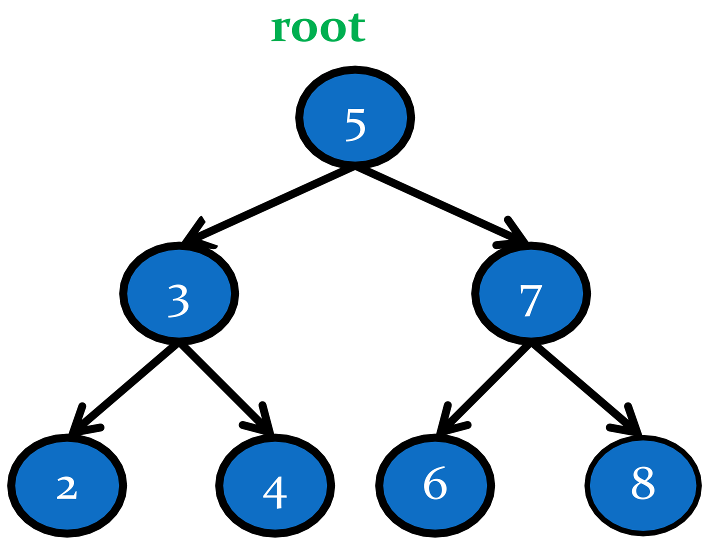
Steps:

```
1. Current=root
2. While(current!=NULL and current.data != item)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```

Search 9

Steps:

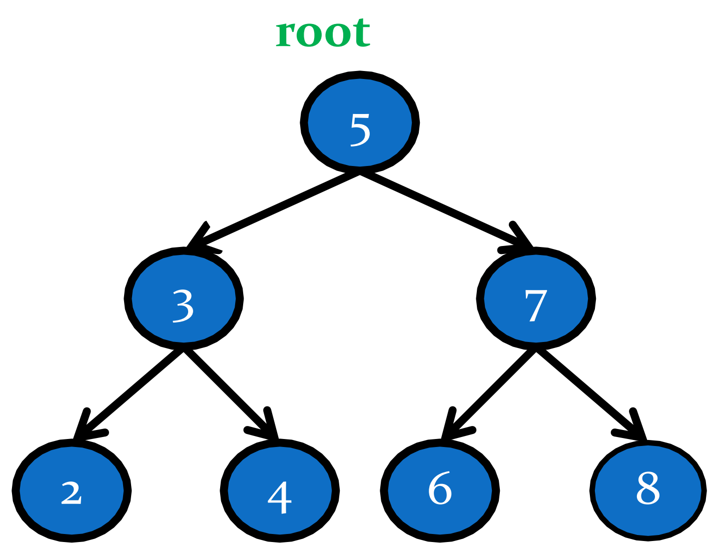
```
1. Current=root
2. While(current!=NULL and current.data != item)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```



Current=NULL

Search 9

- Item not found



Steps:

```
1. Current=root
2. While(current!=NULL and current.data != item)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```

Current=NULL

Algorithm **BST_SEARCH(item)**

Input: item is the data to be searched.

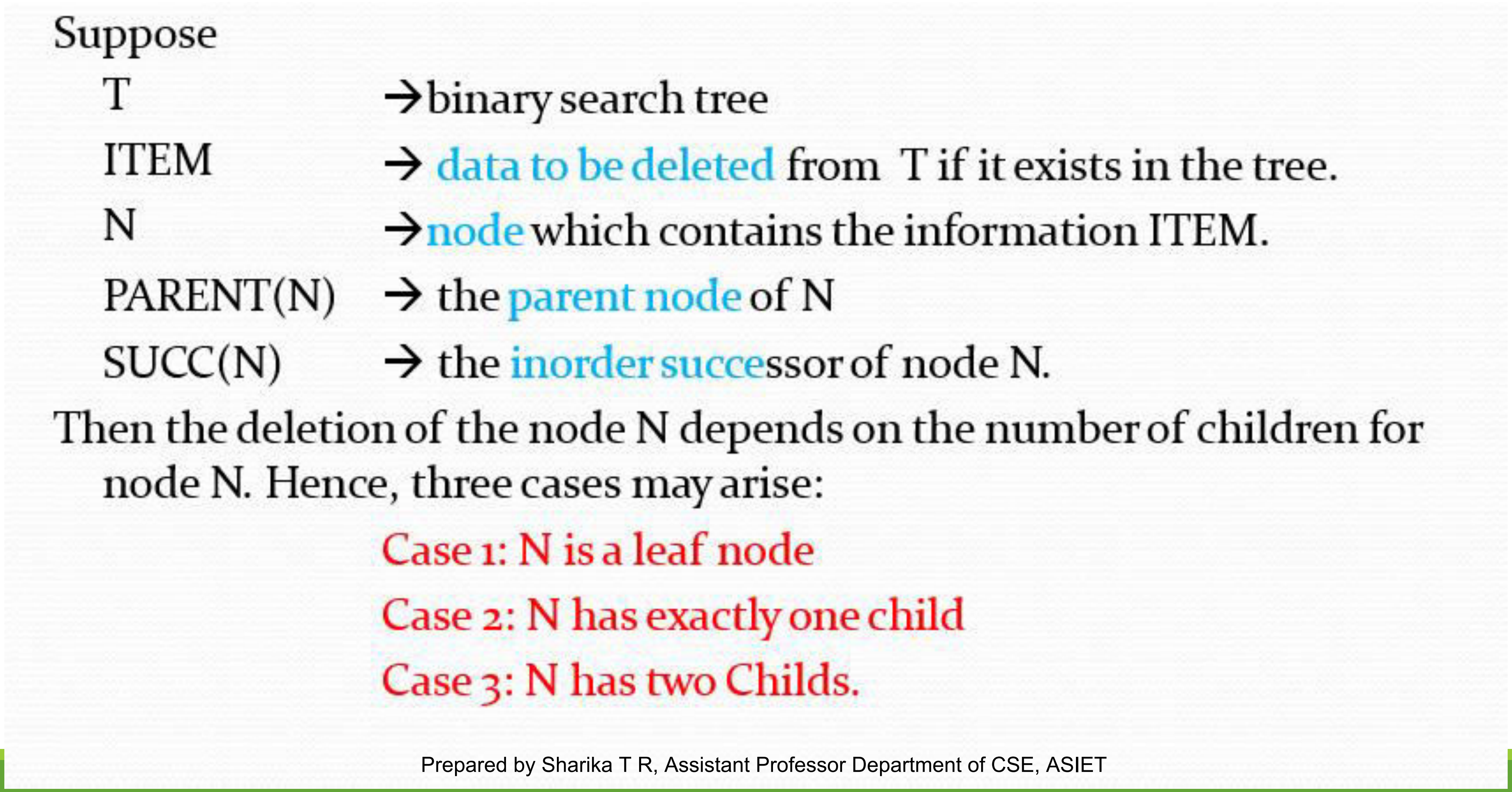
Output: if found then pointer to the node containing data.

Data Structure: Linked Structure.

Steps:

```
1. Current=root
2. While(current!=NULL and current.data != item)
    1. If( item>current.data)
        1. Current=current.rlink
    2. Else
        1. Current=current.llink
    3. EndIf
3. EndWhile
4. If(current=NULL)
    1. Print("Item notFound")
    2. exit
5. Else
    1. Print("Item Found")
6. EndIf
7. Return (current)
```

Deleting a node from a binary search tree



Suppose

T	→ binary search tree
ITEM	→ data to be deleted from T if it exists in the tree.
N	→ node which contains the information ITEM.
PARENT(N)	→ the parent node of N
SUCC(N)	→ the inorder successor of node N.

Then the deletion of the node N depends on the number of children for node N. Hence, three cases may arise:

Case 1: N is a leaf node

Case 2: N has exactly one child

Case 3: N has two Childs.

Deleting a node from a binary search tree

Steps:

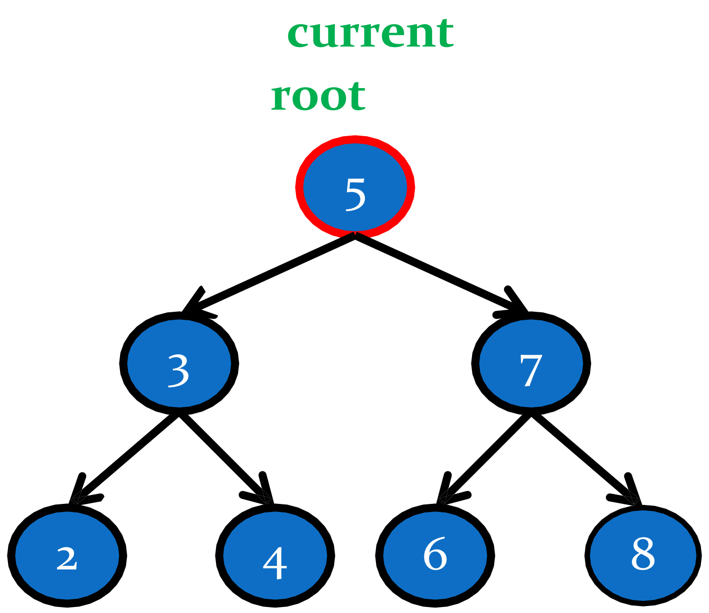
1. Search for the particular node
2. If item Not found
    1. Print a message
    2. Exit
3. Else
    1. Case 1: N is a leaf node
    2. Case 2: N has exactly one child and it's a left child
    3. Case 3: N has exactly one child and it's a right child
    4. Case 4: N has two Childs.
4. Stop

IMPLEMENTATION

Deleting a node from a binary search tree

Delete 6

Search :

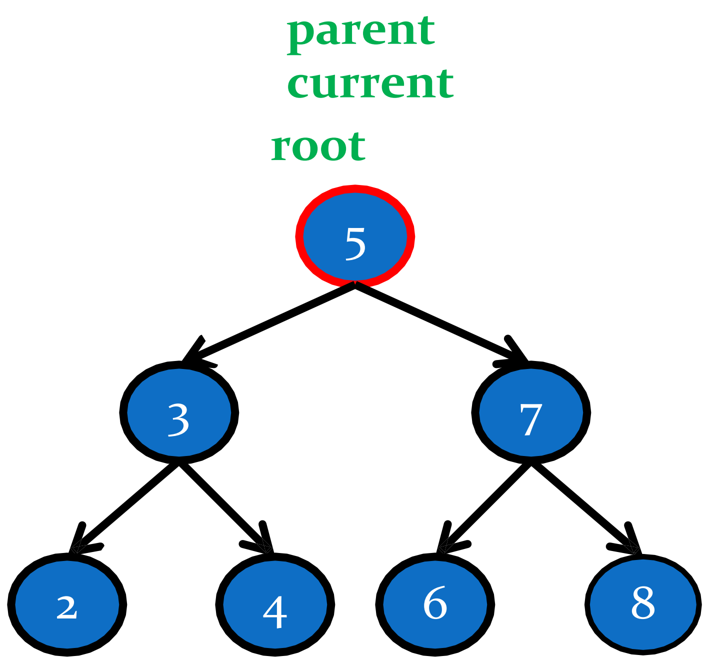


```
1. Flag=0
2. Current=root
3. While(current!=NULL)
    1. If(current.data=item)
        1. Flag=1
        2. ExitWhile
    2. Else
        1. Parent=current
        2. If(item>current.data)
            1. Current=current.rlink
        3. Else Current=current.llink
        4. EndIf
    3. EndIf
4. EndWhile
5. If(flag=0)
    1. Print(" item NOT found in the tree")
    2. Exit
6. EndIf
```

IMPLEMENTATION

Deleting a node from a binary search tree

Delete 6 Search:

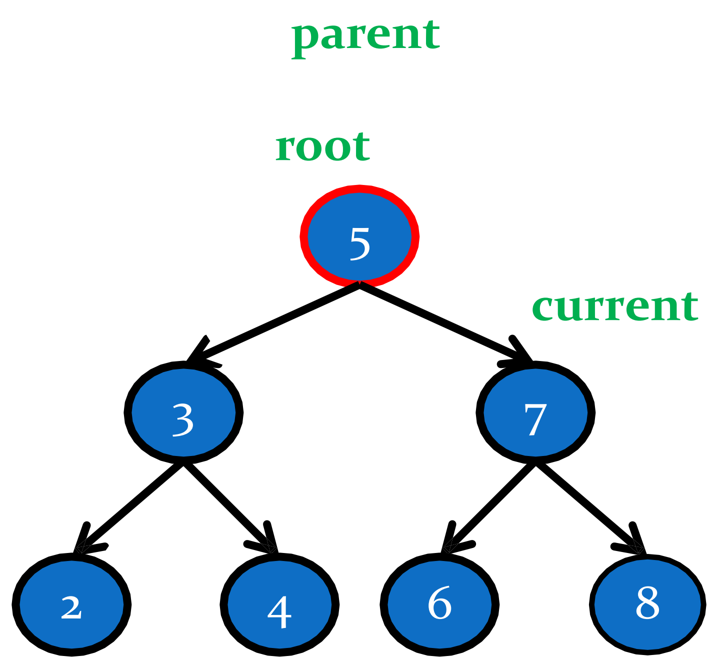


```
1. Flag=0
2. Current=root
3. While(current!=NULL)
    1. If(current.data=item)
        1. Flag=1
        2. ExitWhile
    2. Else
        1. Parent=current
        2. If(item>current.data)
            1. Current=current.rlink
        3. Else Current=current.llink
        4. EndIf
    3. EndIf
4. EndWhile
5. If(flag=0)
    1. Print(" item NOT found in the tree")
    2. Exit
6. EndIf
```

Deleting a node from a binary search tree

Delete 6

Search :

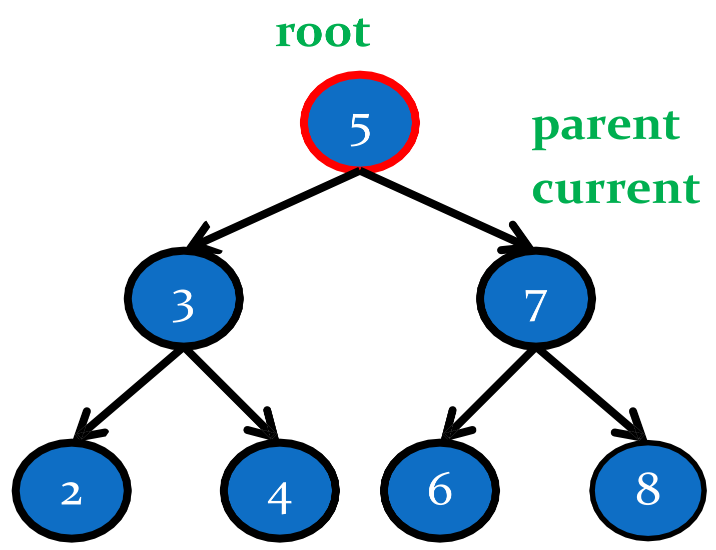


```
1. Flag=0
2. Current=root
3. While(current!=NULL)
    1. If(current.data=item)
        1. Flag=1
        2. ExitWhile
    2. Else
        1. Parent=current
        2. If(item>current.data)
            1. Current=current.rlink
        3. Else Current=current.llink
        4. EndIf
    3. EndIf
4. EndWhile
5. If(flag=0)
    1. Print(" item NOT found in the tree")
    2. Exit
6. EndIf
```

Deleting a node from a binary search tree

Delete 6

Search :

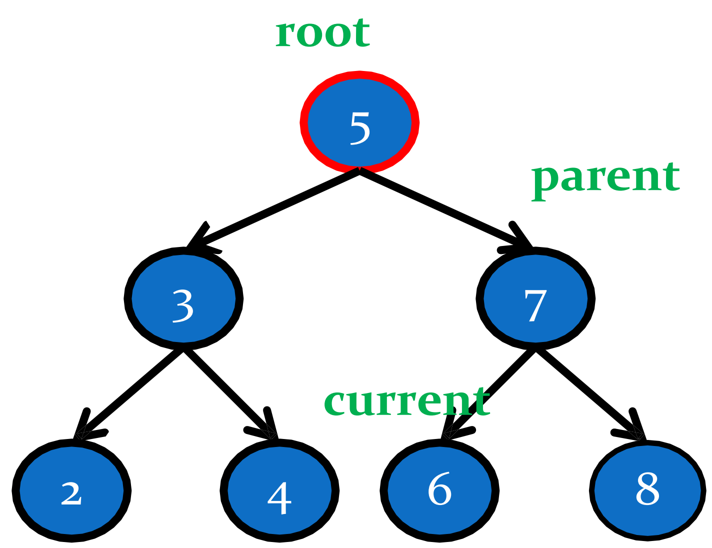


```
1. Flag=0
2. Current=root
3. While(current!=NULL)
   1. If(current.data=item)
      1. Flag=1
      2. ExitWhile
   2. Else
      1. Parent=current
      2. If(item>current.data)
         1. Current=current.rlink
      3. Else Current=current.llink
      4. EndIf
   3. EndIf
4. EndWhile
5. If(flag=0)
   1. Print(" item NOT found in the tree")
   2. Exit
6. EndIf
```

Deleting a node from a binary search tree

Delete 6

Search:

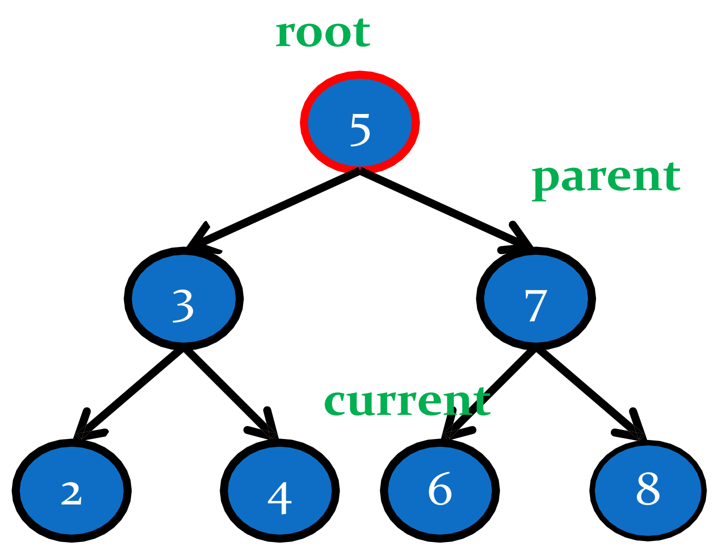


```
1. Flag=0
2. Current=root
3. While(current!=NULL)
   1. If(current.data=item)
      1. Flag=1
      2. ExitWhile
   2. Else
      1. Parent=current
      2. If(item>current.data)
         1. Current=current.rlink
      3. Else Current=current.llink
      4. EndIf
   3. EndIf
4. EndWhile
5. If(flag=0)
   1. Print(" item NOT found in the tree")
   2. Exit
6. EndIf
```

IMPLEMENTATION
Deleting a node from a binary search tree

Delete 6

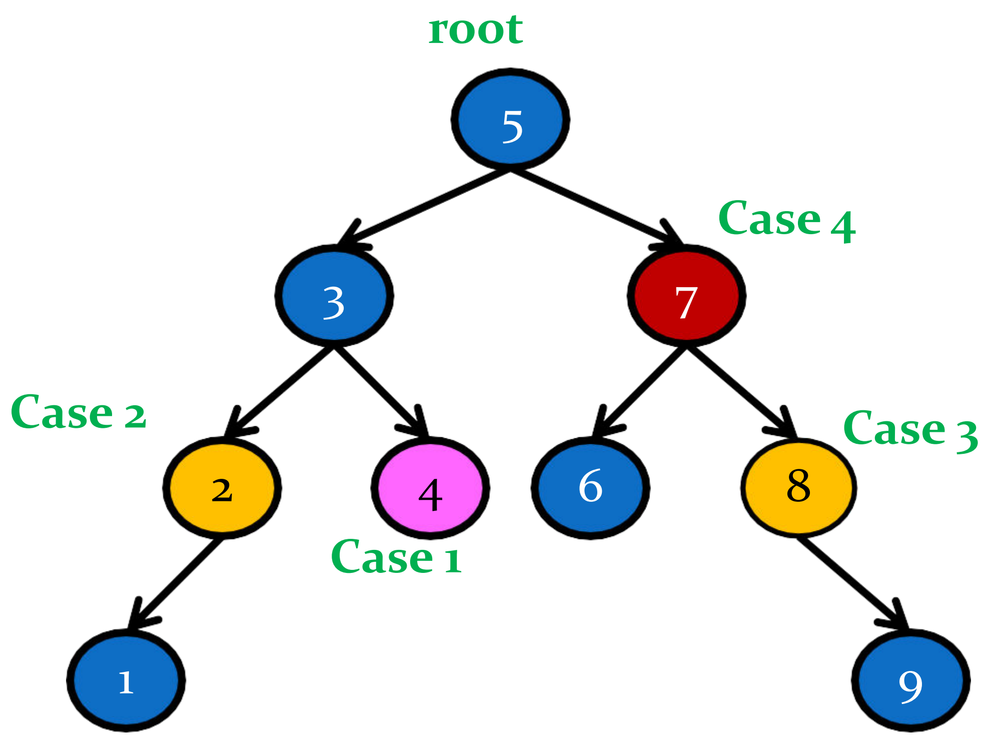
Search : **Flag=1**



```
1. Flag=0
2. Current=root
3. While(current!=NULL)
   1. If(current.data=item)
      1. Flag=1
      2. ExitWhile
   2. Else
      1. Parent=current
      2. If(item>current.data)
         1. Current=current.rlink
      3. Else Current=current.llink
      4. EndIf
   3. EndIf
4. EndWhile
5. If(flag=0)
   1. Print(" item NOT found in the tree")
   2. Exit
6. EndIf
```

Deleting a node from a binary search tree

1. Case 1: N is a leaf node
2. Case 2: N has exactly one child and it's a left child
3. Case 3: N has exactly one child and it's a right child
4. Case 4: N has two Childs.



Deleting a node from a binary search tree

1. Case 1: N is a leaf node
    1. N is the root
    2. N is the left child of its parent
    3. N is the right child of its parent
2. Case 2: N has exactly one child and it's a left child
    1. N is the root
    2. N is the left child of its parent
    3. N is the right child of its parent
3. Case 3: N has exactly one child and it's a right child
    1. N is the root
    2. N is the left child of its parent
    3. N is the right child of its parent
4. Case 4: N has two Childs.

Deleting a node from a binary search tree

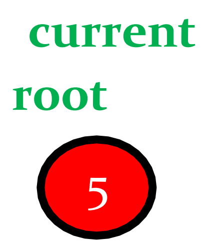
Case 1: N is a leaf node

1. N is the root
2. N is the left child of its parent
3. N is the right child of its parent

Case 1: N is a leaf node

N is the root

(Delete 5)



Case 1: N is a leaf node

N is the root

(Delete 5)

current

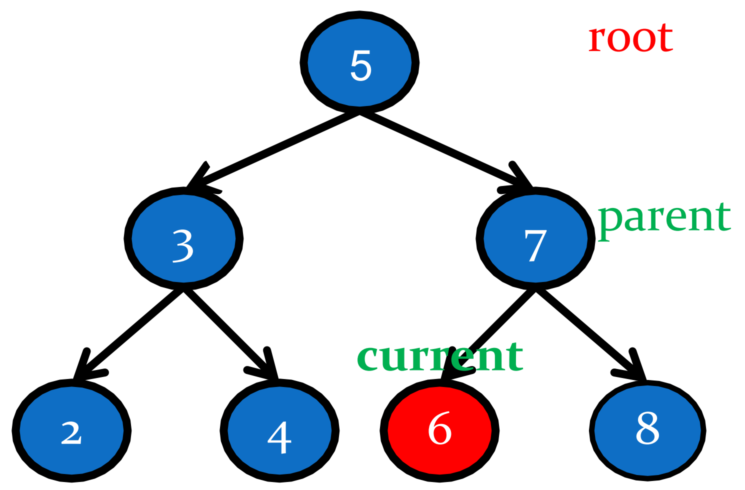
Root = NULL

```
If(current.llink=NULL && current.rlink=NULL)
1.    If(root=current)
    1.    FREE (current)
    2.    Root=NULL
2.    ElseIf(parent.llink=current)
    1.    Parent.llink=NULL
    2.    FREE (current)
3.    ElseIf(parent.rlink=current)
    1.    Parent.rlink=NULL
    2.    FREE (current)
4.    EndIf
```

Case 1: N is a leaf node
N is the left child of its parent

(Delete 6)

N is deleted by simply setting the pointer of N in the parent node PARENT(N) to a NULL value

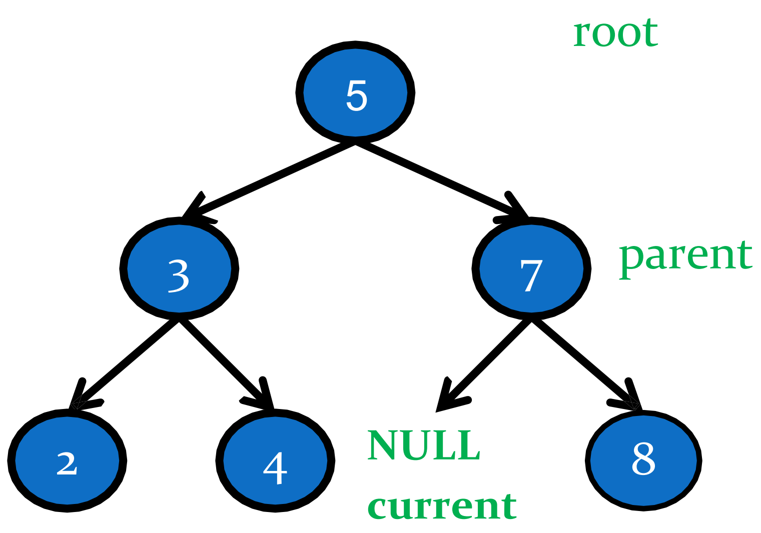


Case 1: N is a leaf node

N is the left child of its parent

(Delete 6)

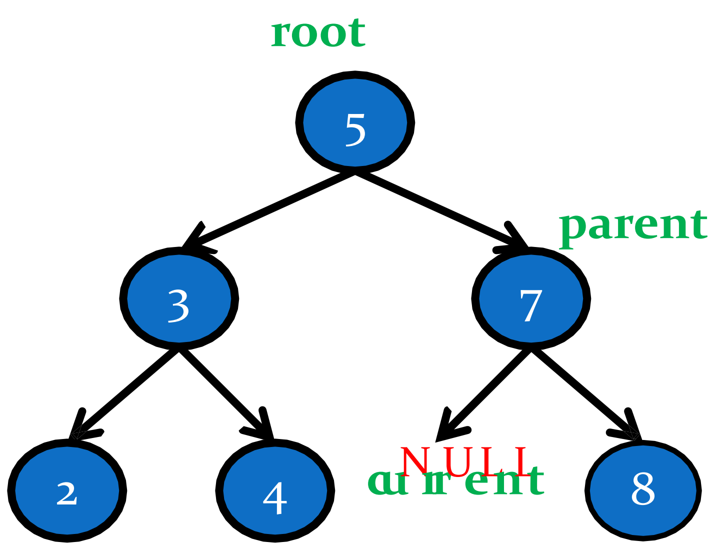
N is deleted by simply setting the pointer of N in the parent node PARENT(N) to a NULL value



Case 1: N is a leaf node

N is the left child of its parent

(Delete 6)

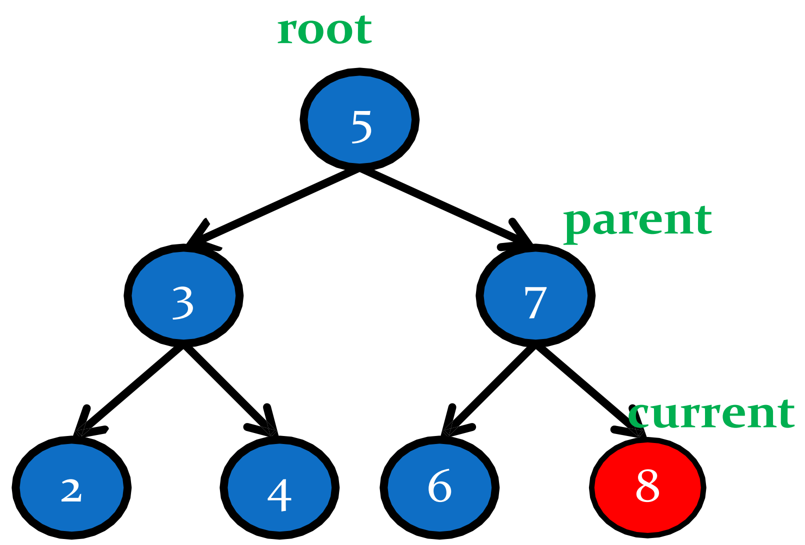


```
If(current.llink=NULL && current.rlink=NULL)
1.      If(root=current)
    1.      FREE (current)
    2.      Root=NULL
2.      ElseIf(parent.llink=current)
    1.      Parent.llink=NULL
    2.      FREE (current)
3.      ElseIf(parent.rlink=current)
    1.      Parent.rlink=NULL
    2.      FREE (current)
4.      EndIf
```

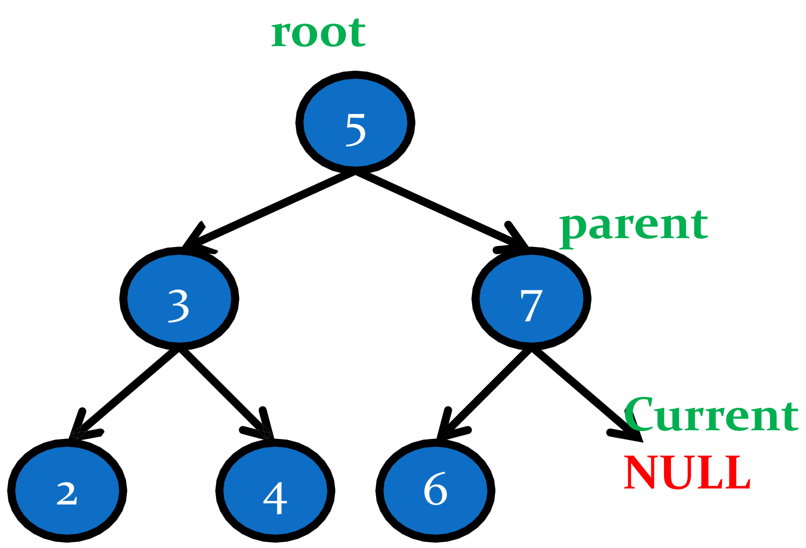
Case 1: N is a leaf node

N is the right child of its parent

(Delete 8)



Case 1: N is a leaf node
N is the right child of its parent
(Delete 8)



```
If(current.llink=NULL && current.rlink=NULL)
1.    If(root=current)
      1.    FREE (current)
      2.    Root=NULL
2.    ElseIf(parent.llink=current)
      1.    Parent.llink=NULL
      2.    FREE (current)
3.    ElseIf(parent.rlink=current)
      1.    Parent.rlink=NULL
      2.    FREE (current)
4.    EndIf
```

Deleting a node from a binary search tree

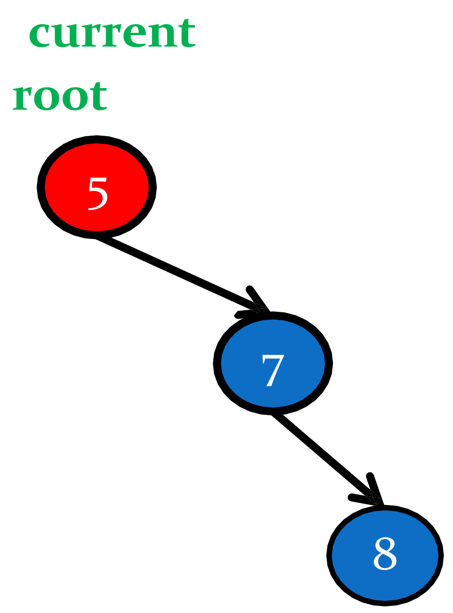
Case 2: N has exactly one child and it's a right child

1. N is the root
2. N is the left child of its parent
3. N is the right child of its parent

N has exactly one child and it’s a right child

N is the root

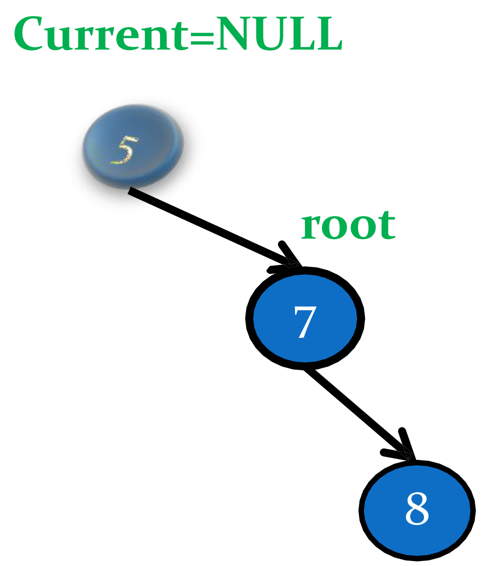
Delete 5



N has exactly one child and it's a right child

N is the root

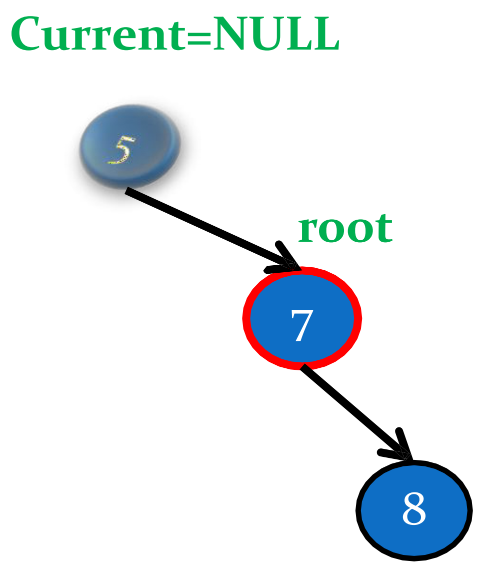
Delete 5



N has exactly one child and it's a right child

N is the root

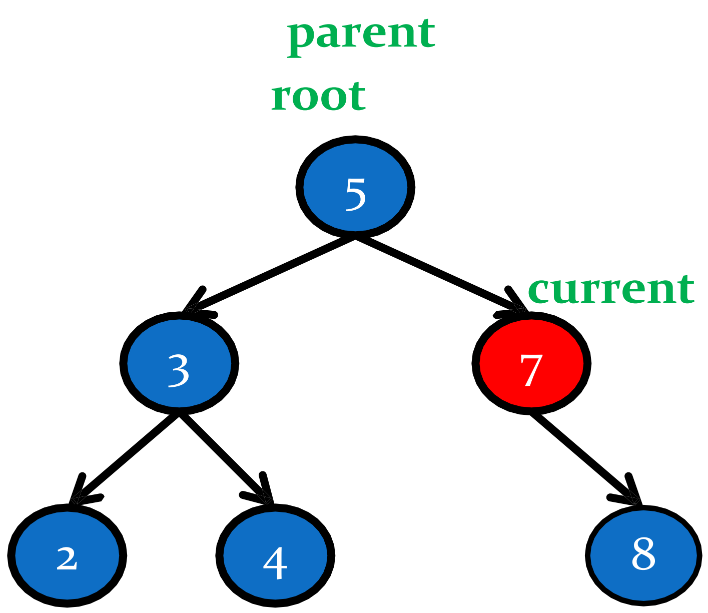
Delete 5



```
current.llink == NULL && current.rlink != NULL
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a right child
N is the right child of its parent

Delete 7



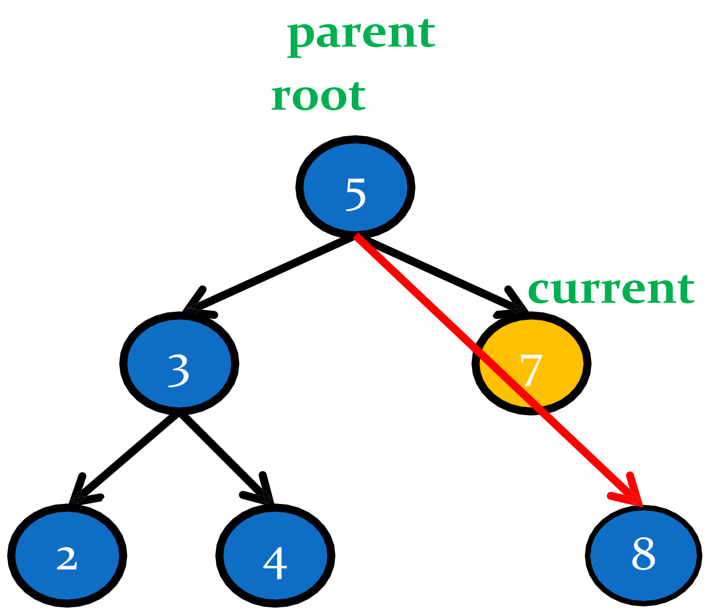
current.llink == NULL && current.rlink != NULL

```
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a right child

N is the right child of its parent

Delete 7

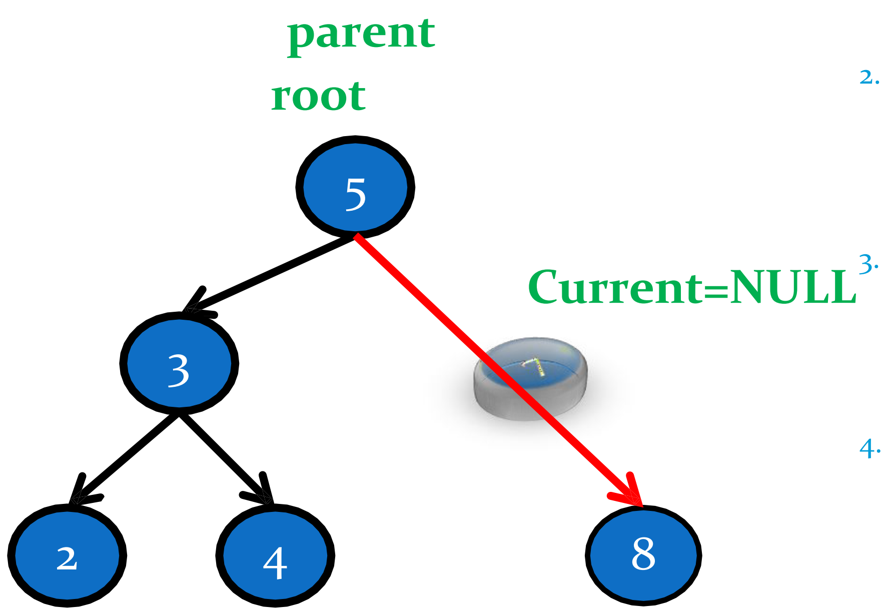


current.llink == NULL && current.rlink != NULL

```
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a right child
N is the right child of its parent

Delete 7

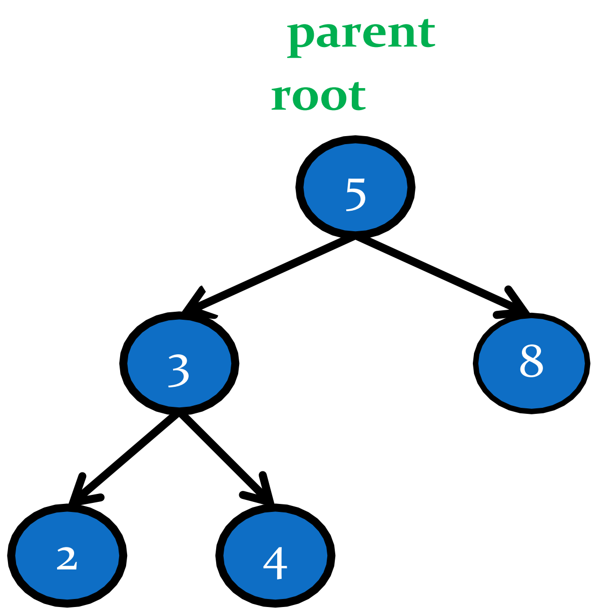


current.llink == NULL && current.rlink != NULL

```
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a right child
N is the right child of its parent

Delete 7



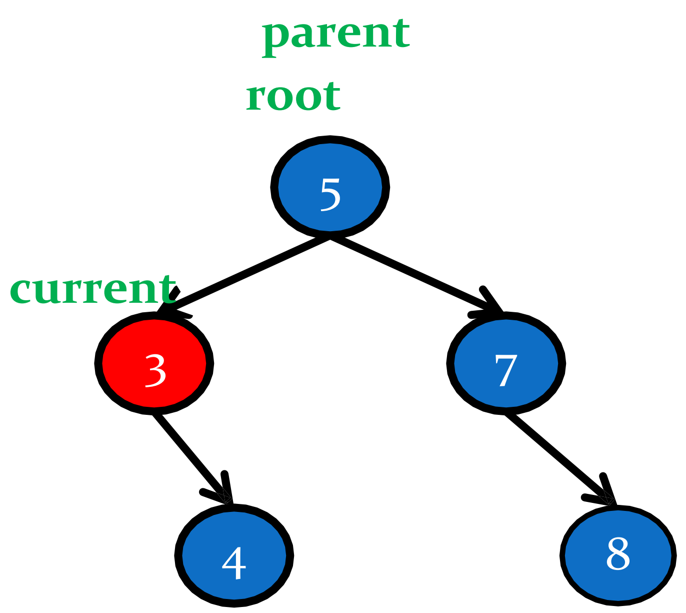
current.llink == NULL && current.rlink != NULL

```
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a left child

N is the left child of its parent

Delete 3

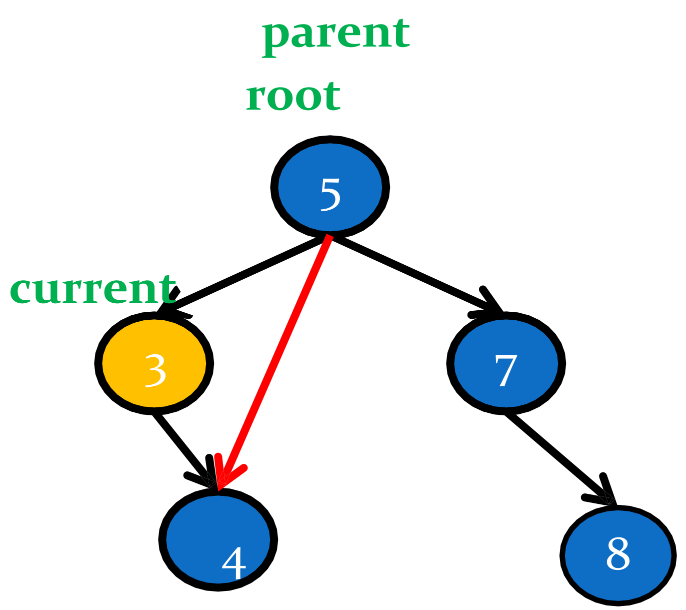


current.llink == NULL && current.rlink != NULL

```
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a left child
N is the left child of its parent

Delete 3



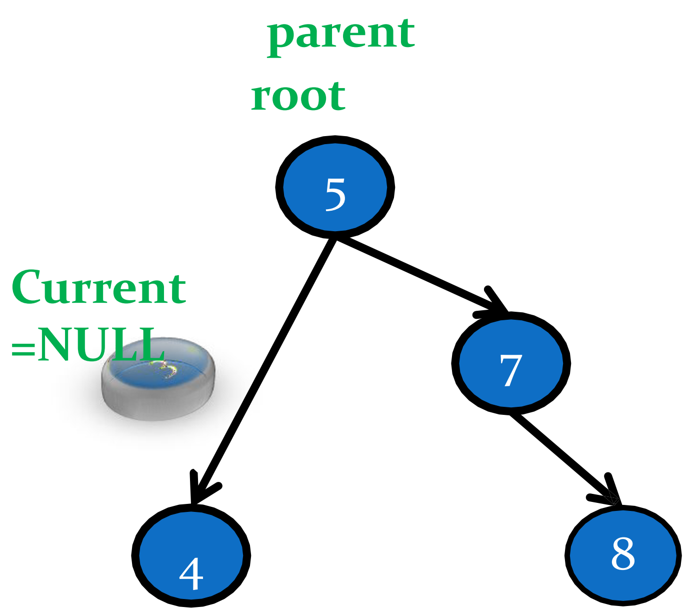
current.llink == NULL && current.rlink != NULL

```
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a left child

N is the left child of its parent

Delete 3

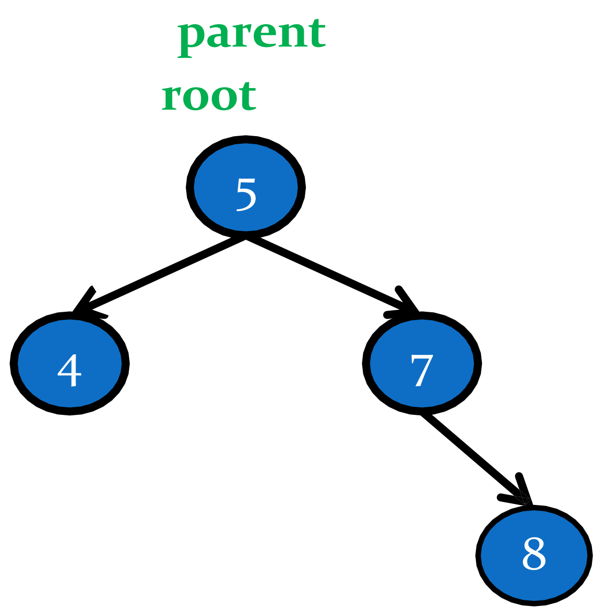


current.llink == NULL && current.rlink != NULL

```
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a right child
N is the left child of its parent

Delete 3



current.llink == NULL && current.rlink != NULL

```
1. If(root=current)
    1. Root=root.rlink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.rlink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.rlink
    2. FREE (current)
4. EndIf
```

Deleting a node from a binary search tree

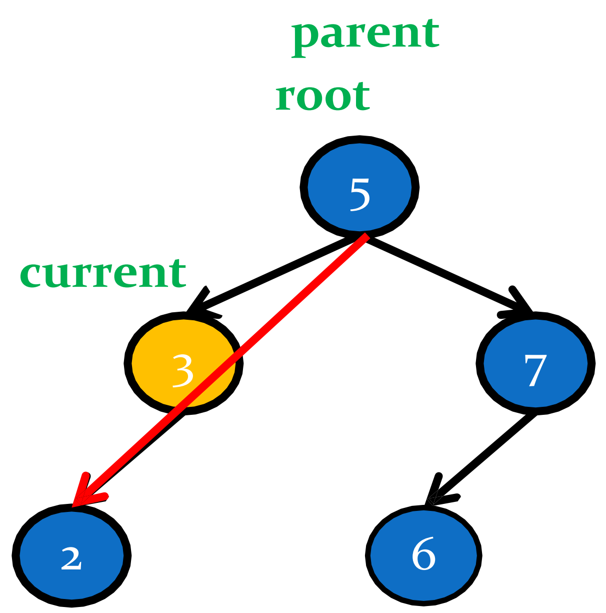
Case 3: N has exactly one child and it's a left child

1. N is the root
2. N is the left child of its parent
3. N is the right child of its parent

N has exactly one child and it's a left child

N is the left child of its parent

Delete 3



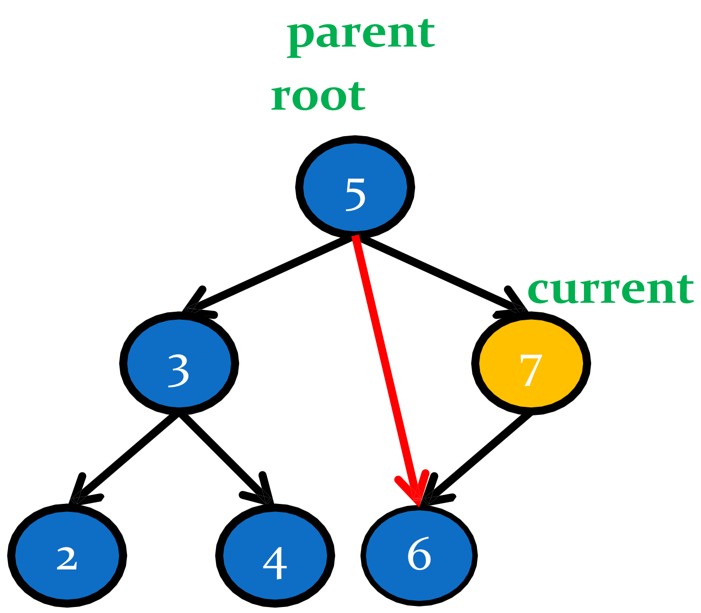
current.llink != NULL && current.rlink == NULL

```
1. If(root=current)
    1. Root=root.llink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.llink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.llink
    2. FREE (current)
4. EndIf
```

N has exactly one child and it's a left child

N is the right child of its parent

Delete 7



current.llink != NULL && current.rlink == NULL

```
1. If(root=current)
    1. Root=root.llink
    2. FREE (current)
2. ElseIf(parent.llink=current)
    1. Parent.llink=current.llink
    2. FREE (current)
3. ElseIf(parent.rlink=current)
    1. Parent.rlink=current.llink
    2. FREE (current)
4. EndIf
```

Deleting a node from a binary search tree

Case 3: N has two Childs.

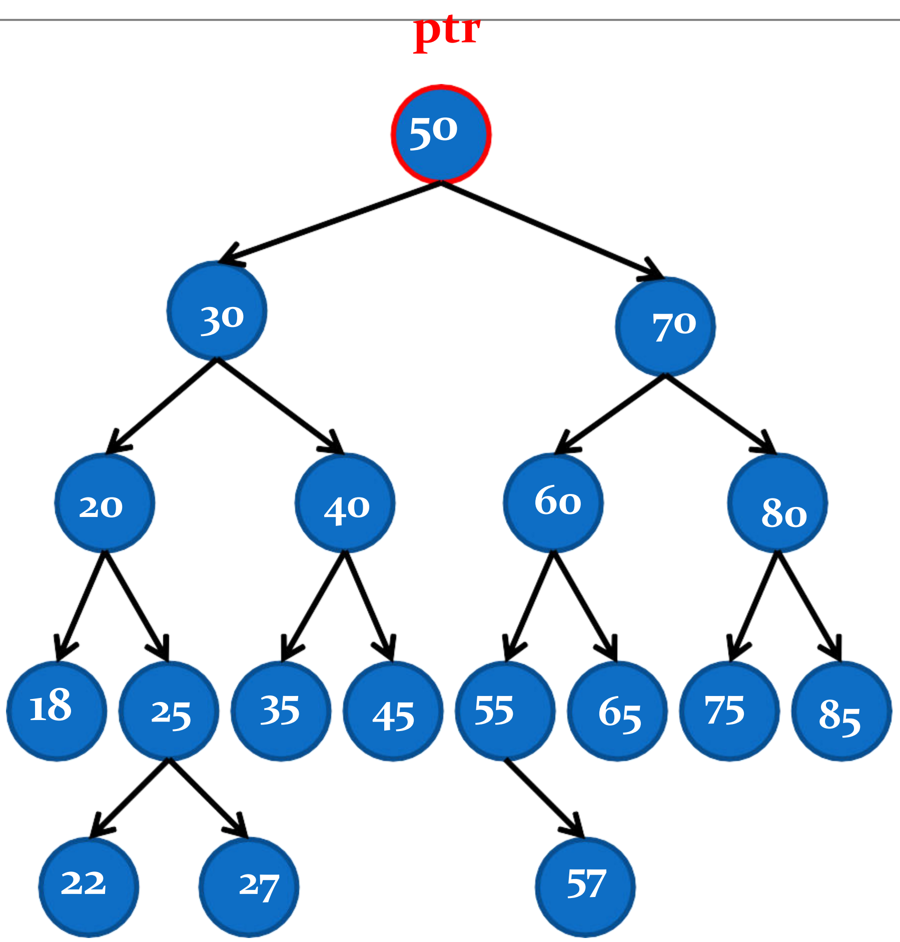
Deleting a node from a binary search tree

Case 3: N has two Childs.

Steps:

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)
4. Reset the leftchild of the parent of SUCCESSOR(N) by the rightchild of SUCCESSOR(N)

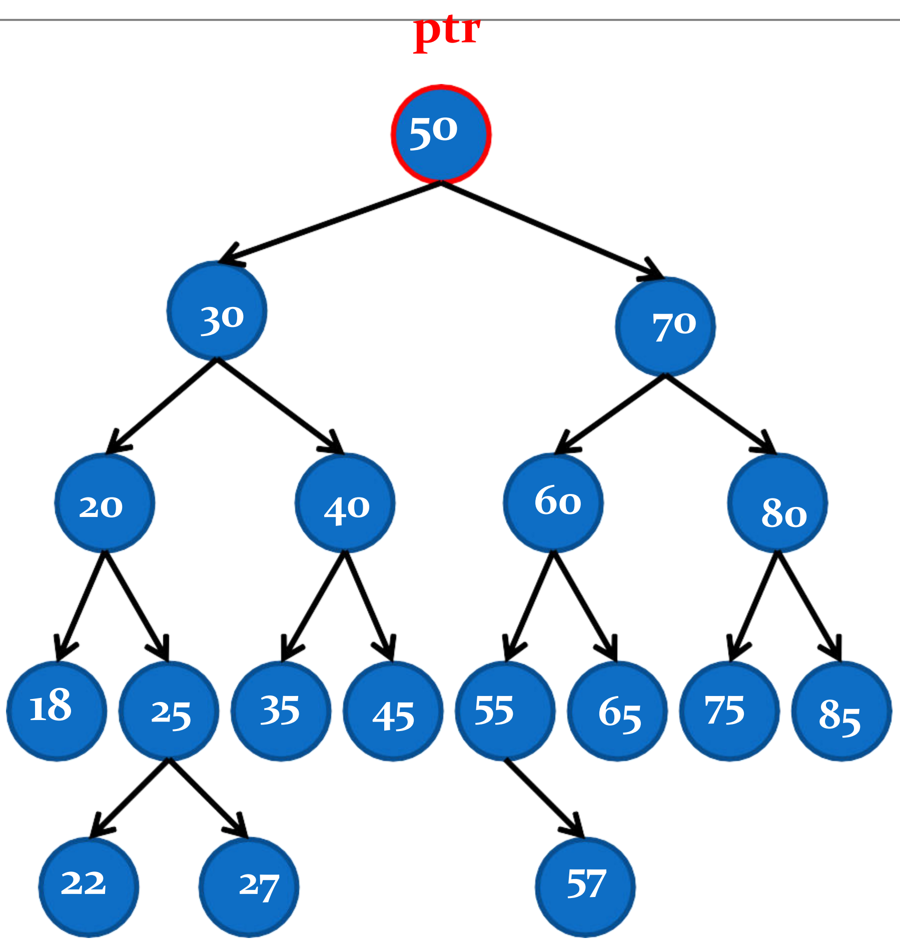
SUCCESSOR(ptr)



Successor (N)

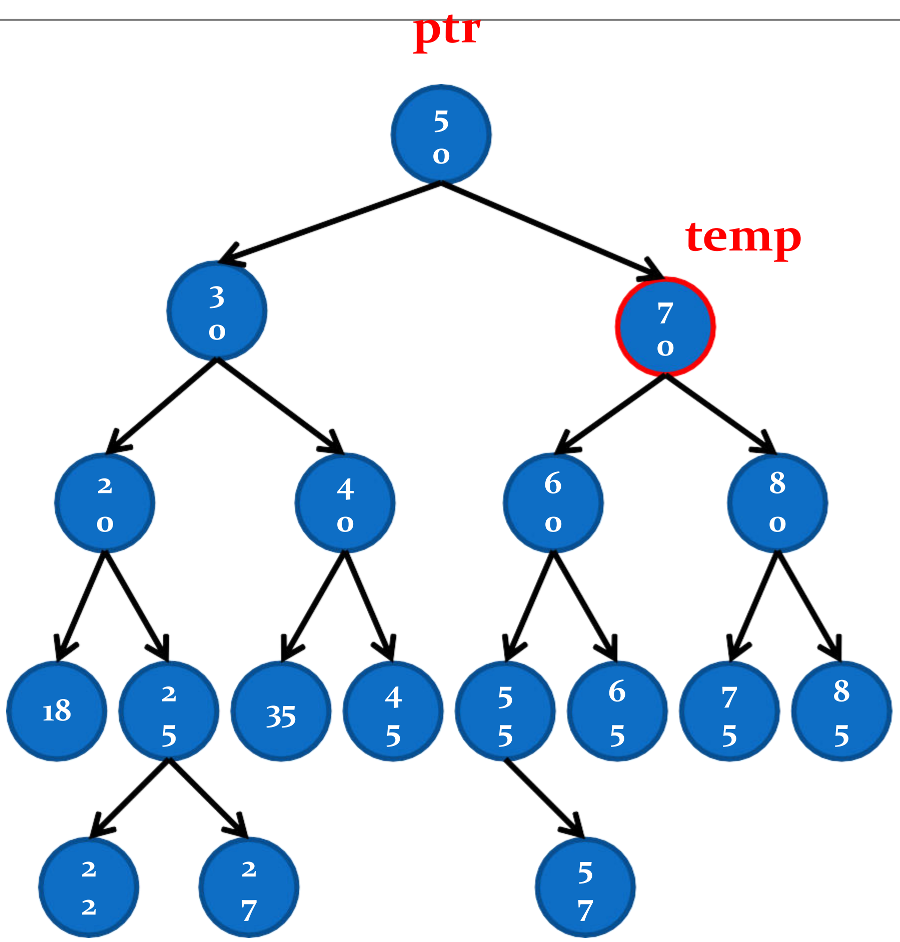
Leftmost child of the right child of N

SUCCESSOR(ptr)



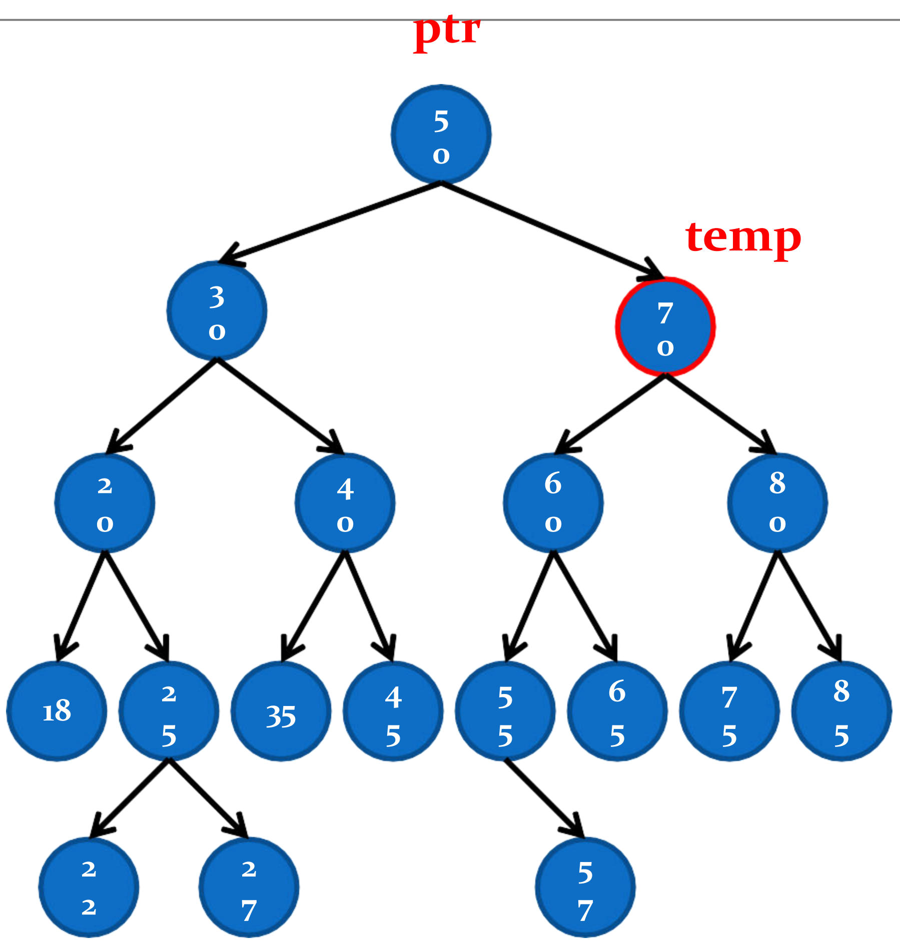
```
1. Temp = ptr.rightlink
2. If (temp !=NULL)
    1. While(temp.leftlink !=NULL)
        1. temp=temp.leftlink
    2. Endwhile
3. Endif
4. Return (temp)
5. stop
```

SUCCESSOR(ptr)



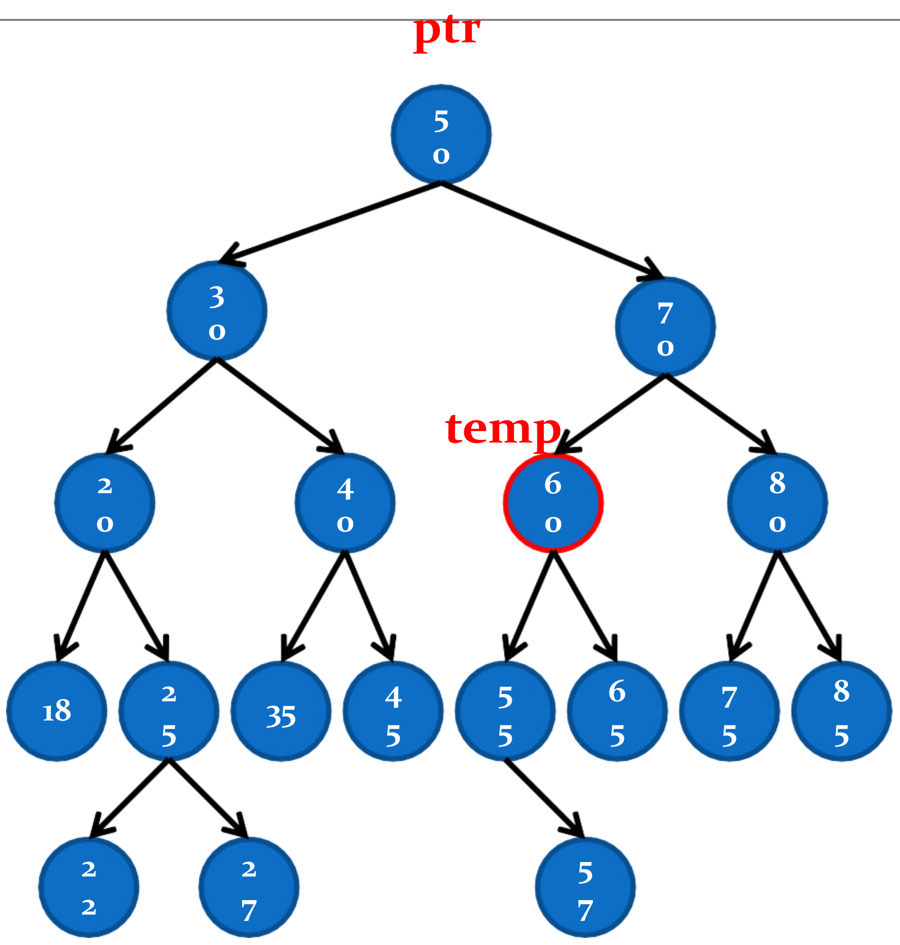
```
1. Temp = ptr.rightlink
2. If (temp !=NULL)
    1. While(temp.leftlink !=NULL)
        1. temp=temp.leftlink
    2. Endwhile
3. Endif
4. Return (temp)
5. stop
```

SUCCESSOR(ptr)



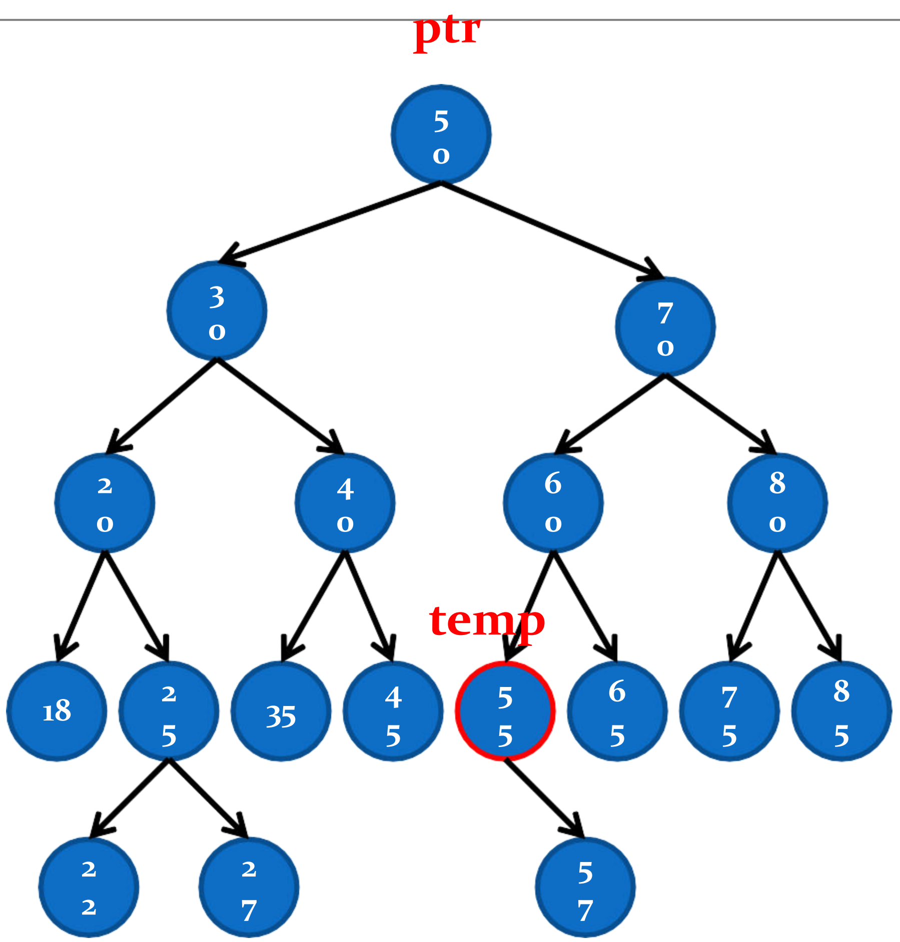
```
1. Temp = ptr.rightlink
2. If (temp !=NULL)
    1. While(temp.leftlink !=NULL)
        1. temp=temp.leftlink
    2. Endwhile
3. Endif
4. Return (temp)
5. stop
```

SUCCESSOR(ptr)



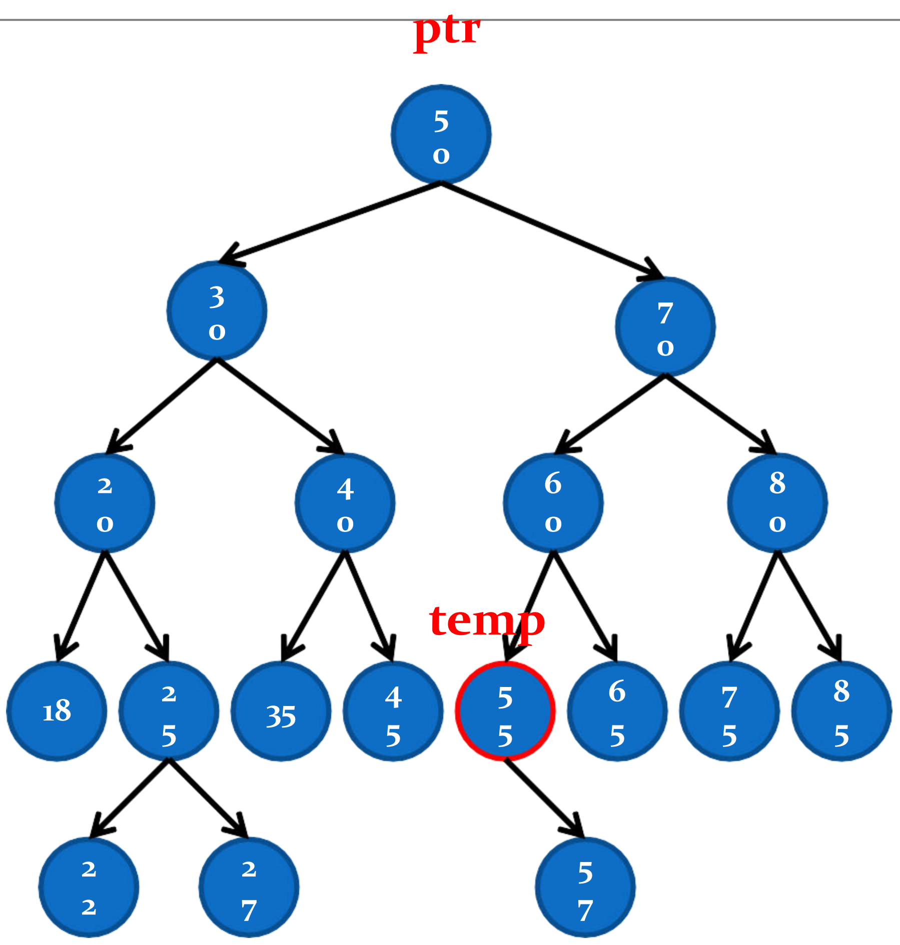
```
1. Temp = ptr.rightlink
2. If (temp !=NULL)
   1. While(temp.leftlink !=NULL)
      1. temp=temp.leftlink
   2. Endwhile
3. Endif
4. Return (temp)
5. stop
```

SUCCESSOR(ptr)



```
1. Temp = ptr.rightlink
2. If (temp !=NULL)
    1. While(temp.leftlink !=NULL)
        1. temp=temp.leftlink
    2. Endwhile
3. Endif
4. Return (temp)
5. stop
```

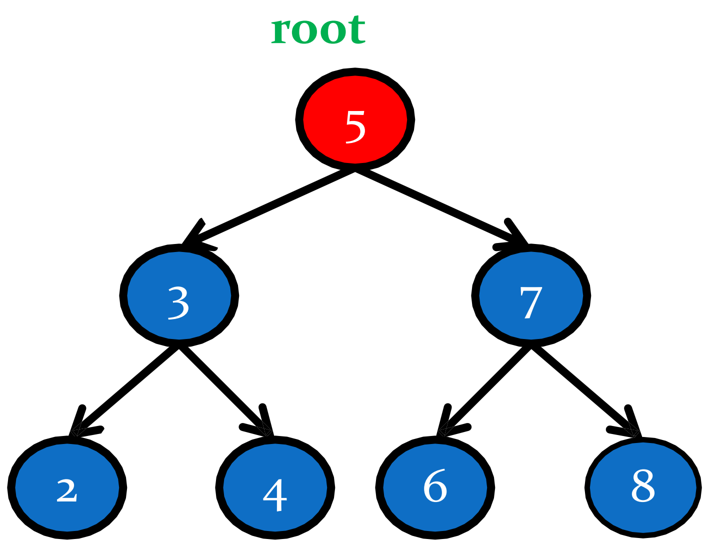
SUCCESSOR(ptr)



```
1. Temp = ptr.rightlink
2. If (temp !=NULL)
    1. While( temp.leftlink !=NULL )
        1. temp=temp.leftlink
    2. Endwhile
3. Endif
4. Return (temp)
5. stop
```

SUCCESSOR(50)=55

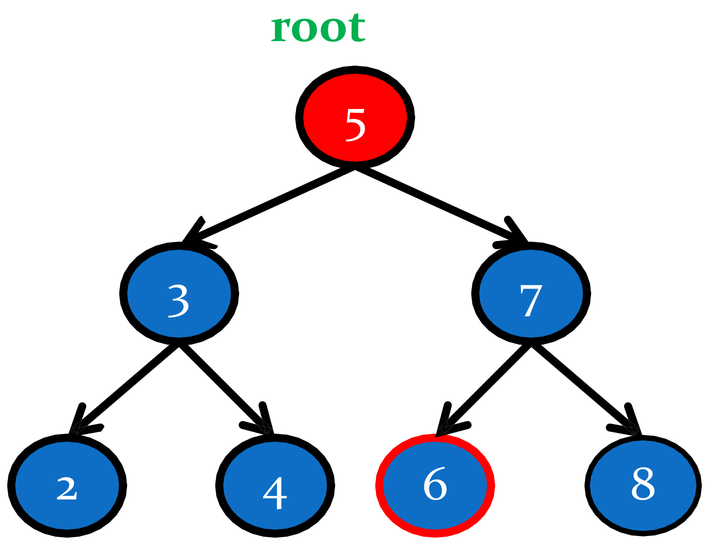
DELETE 5



N has two Childs.

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Reset the leftchild of the parent of SUCCESSOR(N) by the rightchild of SUCCESSOR(N)
4. Delete SUCCESSOR(N)

DELETE 5



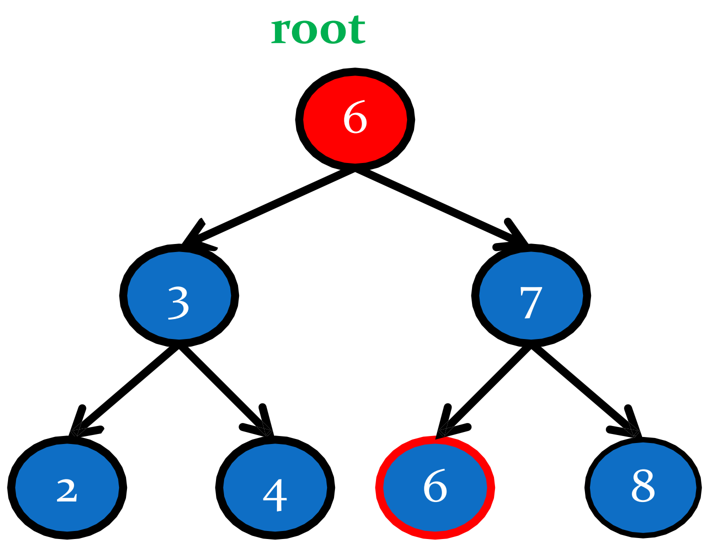
N has two Childs.

1. Find SUCCESSOR (N)
    1. SUCCESSOR(5) = NODE(6)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)

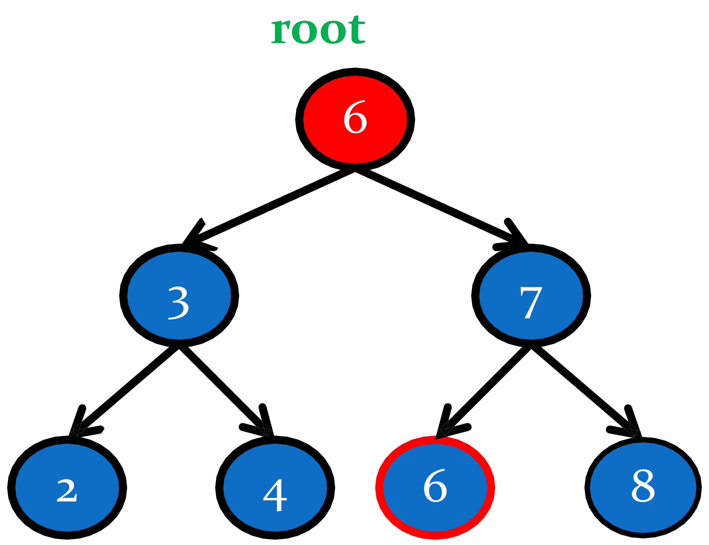
DELETE 5

N has two Childs.

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)



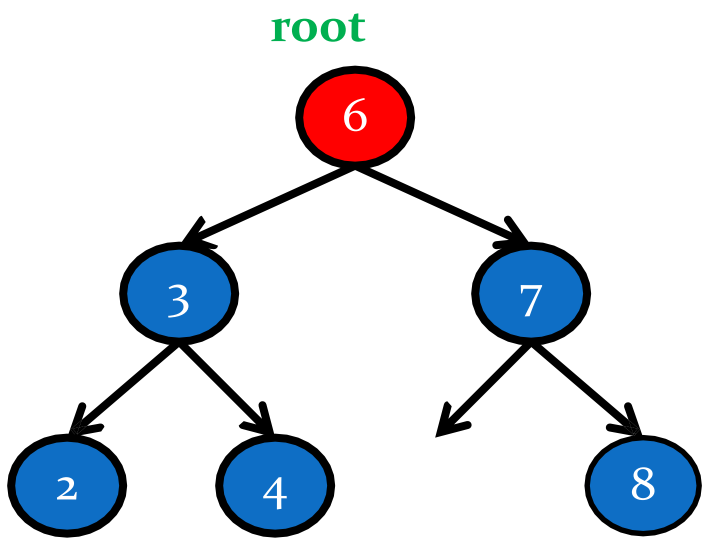
DELETE 5



N has two Childs.

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)
    - 3 cases
        - No child
        - Only right child
        - **No chance of a left child**

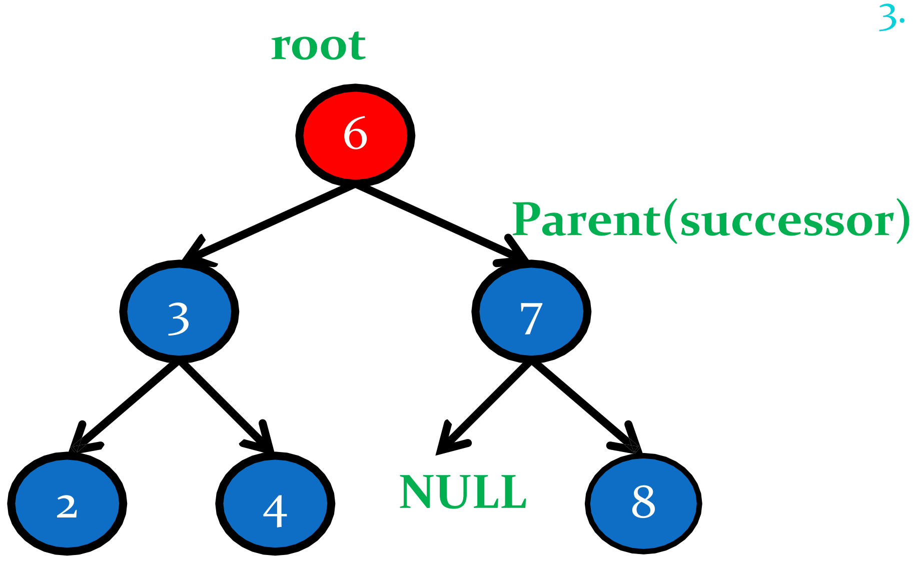
DELETE 5



N has two Childs.

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)
   - 3 cases
     - No child
     - Only right child
     - No chance of a left child

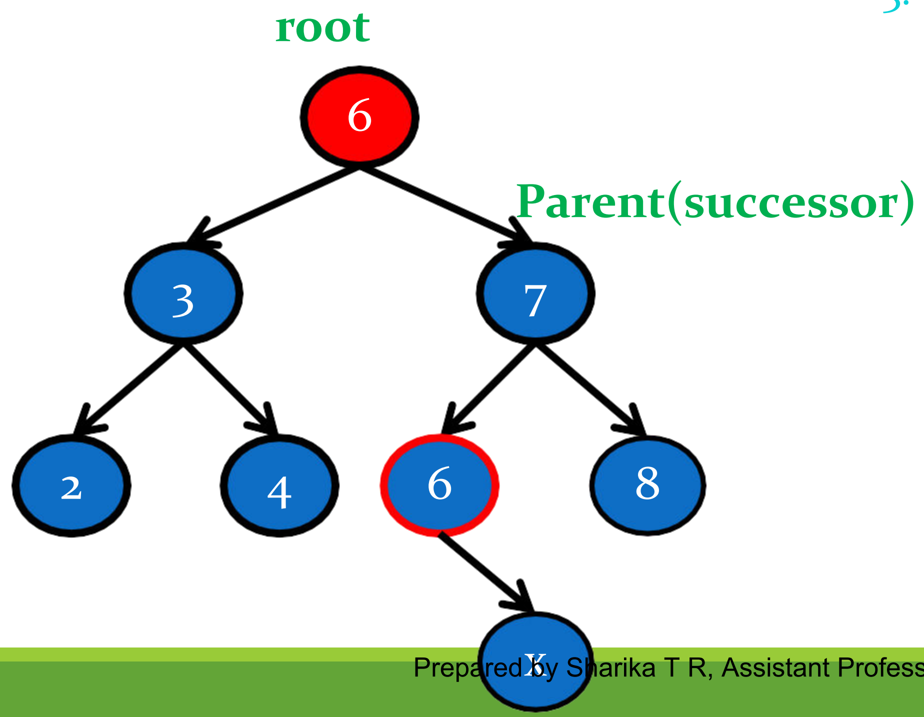
DELETE 5



N has two Childs.

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)
   - 3 cases
     - No child
     - Only right child
     - No chance of a left child

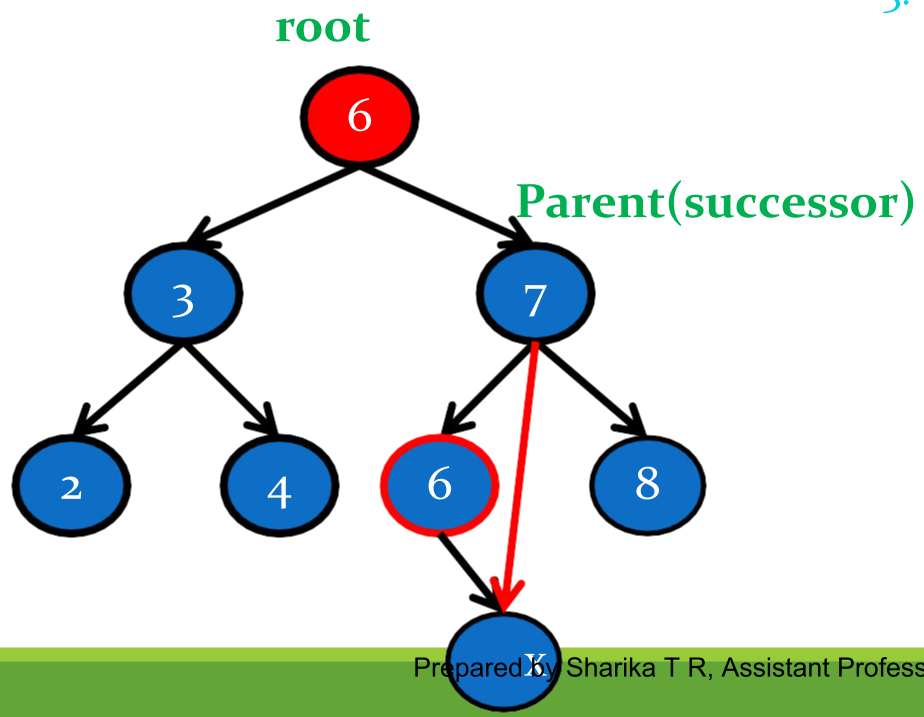
DELETE 5 – if the successor have a right child ??



N has two Childs.

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)
   - 3 cases
     - No child
     - Only right child
     - No chance of a left child

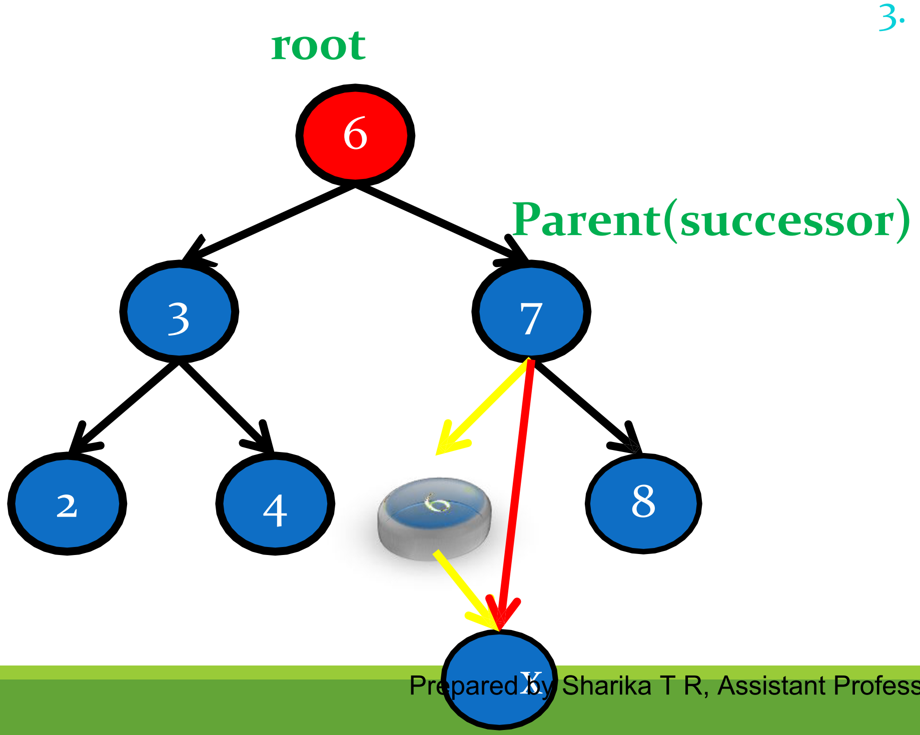
DELETE 5 – if the successor have a right child ??



N has two Childs.

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)
    - 3 cases
        - No child
        - Only right child
        - No chance of a left child

DELETE 5 – if the successor have a right child ??



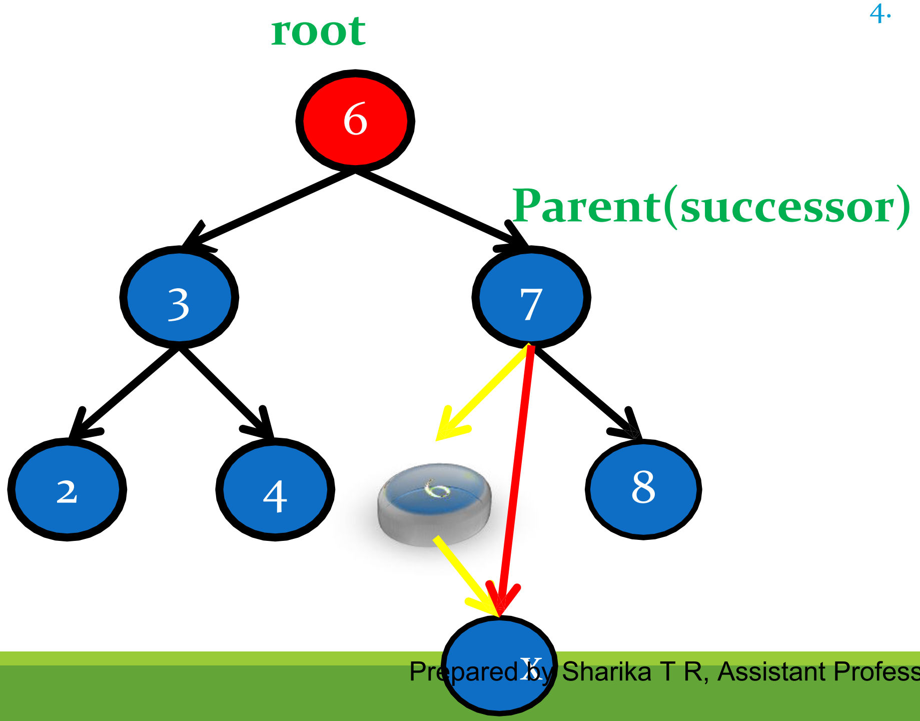
N has two Childs.

1. Find SUCCESSOR (N)
2. Copy data of SUCCESSOR(N) to NODE(N)
3. Delete SUCCESSOR(N)
   - 3 cases
     - No child
     - Only right child
     - No chance of a left child

DELETE 5 – if the successor have a right child ??

current.llink != NULL && current.rlink != NULL

1. Succ = SUCCESSOR (current)
2. item = succ.data
3. Delete (item)
4. Current.data=item



Algorithm BST_DELETE(item)

Input:

item is the data to be removed

Output:

if the node with data as item exist it is deleted else a message.

Data Structure:

Linked structure of binary tree, root will point to the root node

Steps:

```
1.  Flag=0
2.  Parent=NULL
3.  If(root=NULL)
    1.  Print("Tree is Empty")
    2.  Exit
4.  EndIf
5.  Current=root
6.  While(current!=NULL)
    1.  If(current.data=item)
        1.  Flag=1
        2.  ExitWhile
    2.  Else
        1.  Parent=current
        2.  If(item>current.data)
            1.  Current=current.rlink
        3.  Else
            1.  Current=current.llink
        4.  EndIf
    3.  EndIf
7.  EndWhile
8.  If(flag=0)
    1.  Print("No item found in the tree")
    2.  Exit
9.  EndIf
```

```
10. If(current.llink=NULL AND current.rlink=NULL)
    1. If(parent=NULL)
        1. FREE (current)
        2. Root=NULL
    2. ElseIf(parent.llink=current)
        1. Parent.llink=NULL
        2. FREE (current)
    3. ElseIf(parent.rlink=current)
        1. Parent.rlink=NULL
        2. FREE (current)
    4. EndIf
11. ElseIf(current.llink=NULL AND current.rlink!=NULL)
    1. If(parent=NULL)
        1. Root=root.rlink
        2. FREE (current)
    2. ElseIf(parent.llink=current)
        1. Parent.llink=current.rlink
        2. FREE (current)
    3. ElseIf(parent.rlink=current)
        1. Parent.rlink=current.rlink
        2. FREE (current)
    4. EndIf
```

```
12. ElseIf(current.llink!=NULL AND current.rlink=NULL)
    1. If(parent=NULL)
        1. Root=root.llink
        2. FREE (current)
    2. ElseIf(parent.llink=current)
        1. Parent.llink=current.llink
        2. FREE (current)
    3. ElseIf(parent.rlink=current)
        1. Parent.rlink=current.llink
        2. FREE (current)
    4. EndIf
13. Else if (current.llink != NULL && current.rlink != NULL)
    1. Succ =  SUCCESSOR (current)
    2. item = succ.data
    3. BST_DELETE  (item)
    4. Current.data=item
14. Endif
15. stop
```

Graph

•A graph is an abstract data type (ADT) that consists of a set of objects that are connected to each other via links.

•These objects are called **vertices** and the links are called **edges**.

•Usually, a graph is represented as G = {V, E}, where G is the graph space, V is the set of vertices and E is the set of edges.

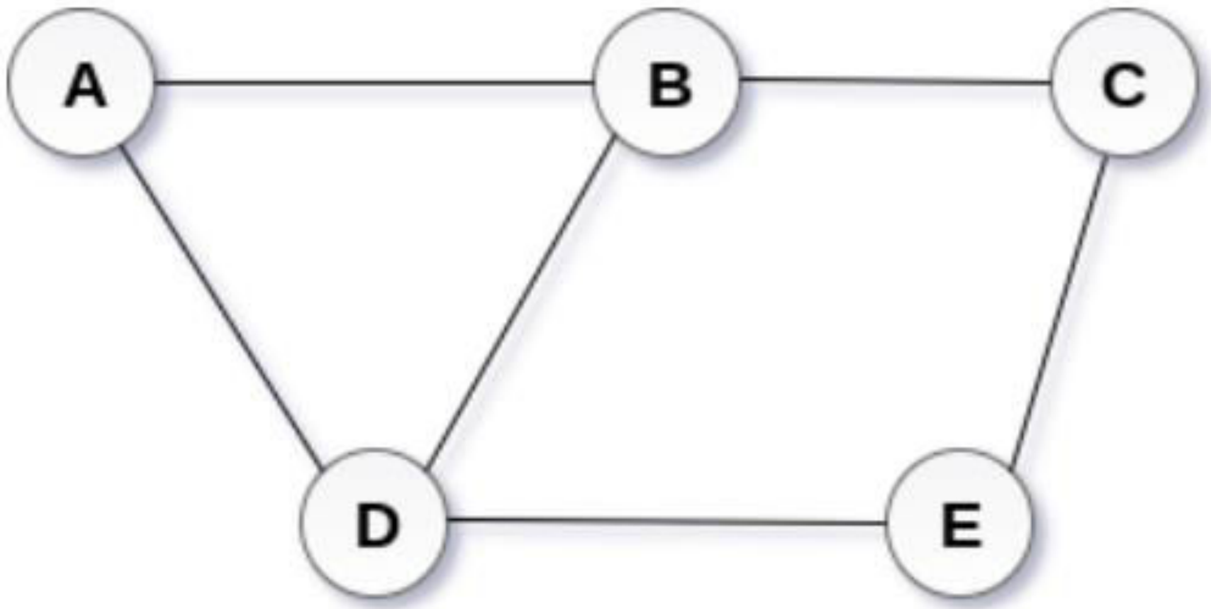
• If E is empty, the graph is known as a **forest**.

•A graph can be defined as group of vertices and edges that are used to connect these vertices.

•A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

***Definition** : A graph G can be defined as an ordered set G(V, E) where V(G) represents the set of vertices and E(G) represents the set of edges which are used to connect these vertices.*

A Graph G(V, E) with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



Graph Terminology

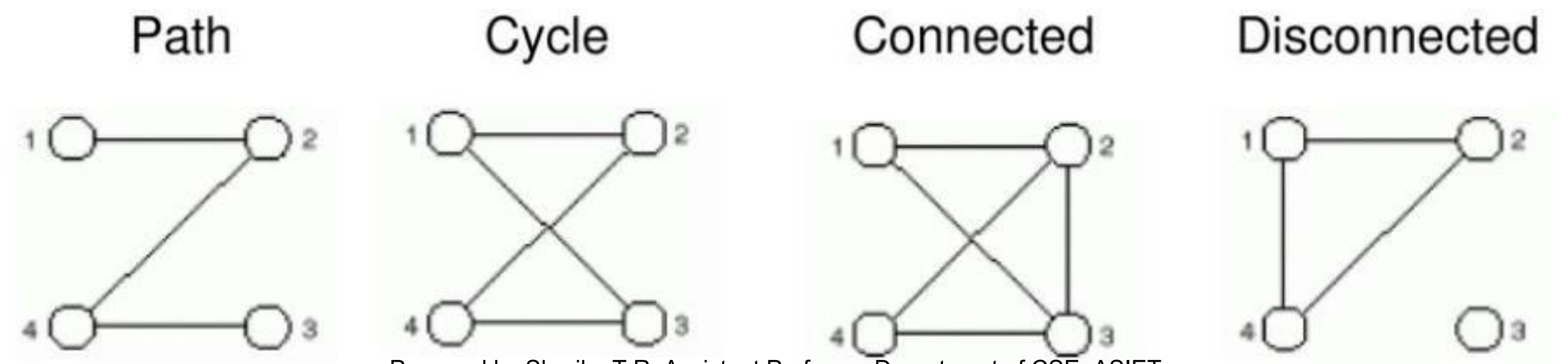
Path - A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Closed Path -A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0=V_N$.

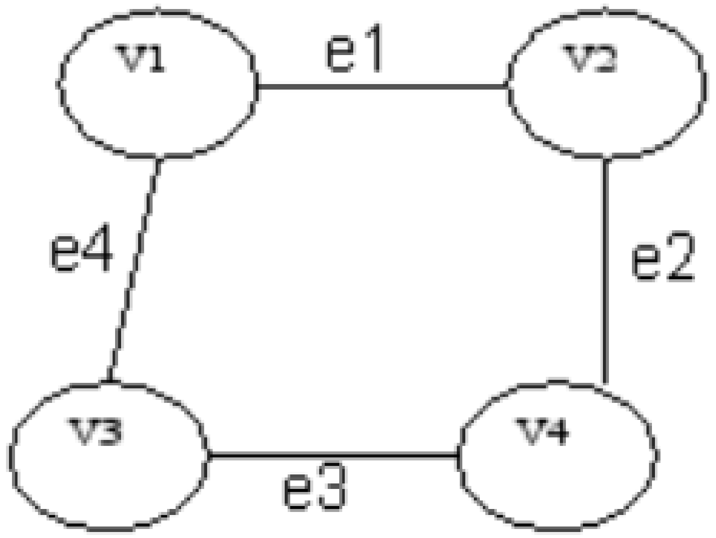
Simple Path - If all the nodes of the graph are distinct with an exception $V_0=V_N$, then such path P is called as closed simple path.

Cycle -A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.

Degree In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it.

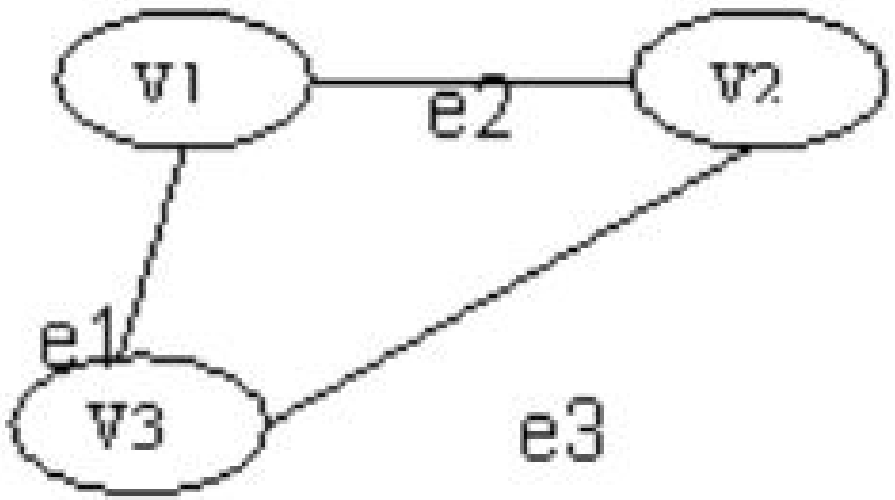


•**Walk -**A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it. Closed walk A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk.



If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2 e2 v4 e3 v3 e4 v1 be its closed walk or circuit

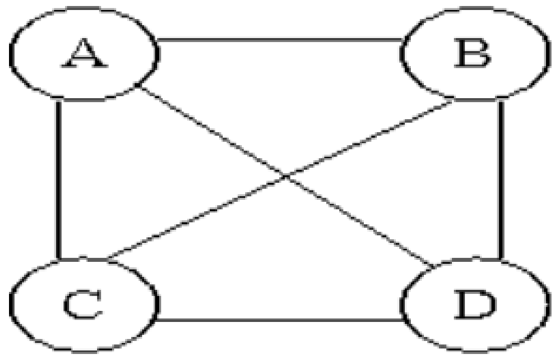
•**Circuit:** A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit. A circuit having three vertices and three edges.



Types of Graph

1.Undirected Graph

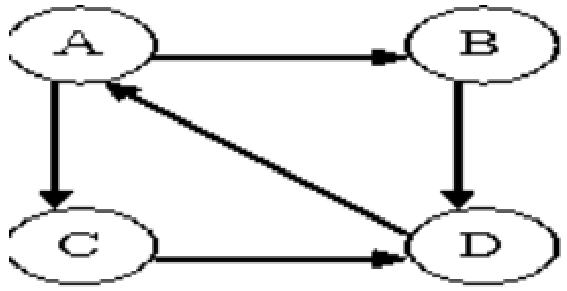
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2.Directed Graph

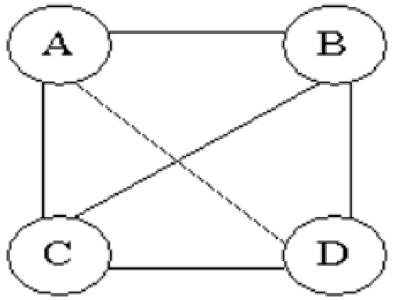
A graph with only directed edges is said to be directed graph.



Directed Graph.

3.Complete Graph

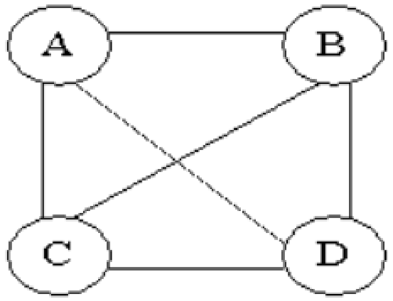
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

4.Regular Graph

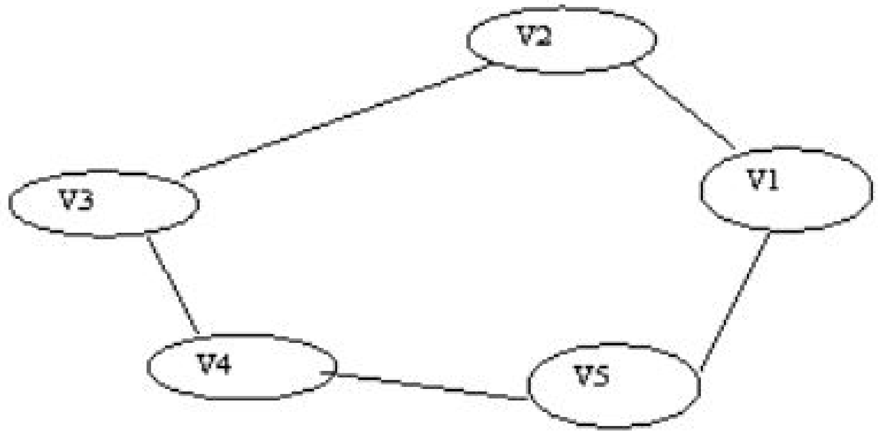
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

5.Cycle Graph

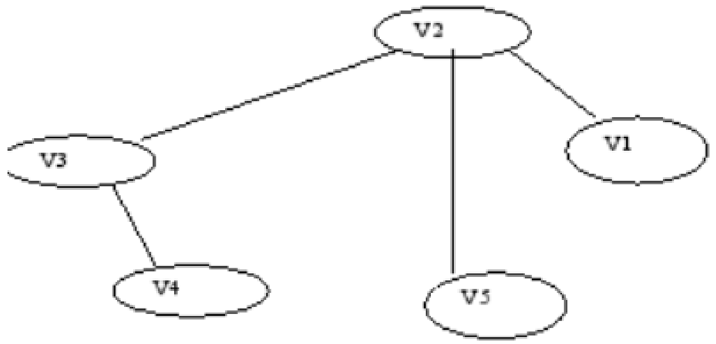
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle



A cycle graph

6.Acyclic Graph

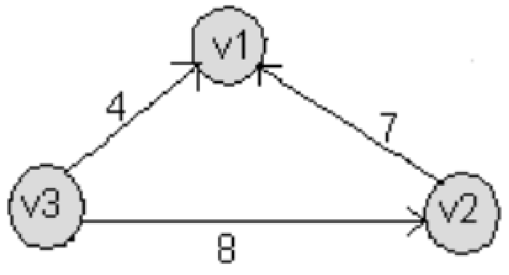
A graph without cycle is called acyclic graphs.



A acyclic graph

7. Weighted Graph

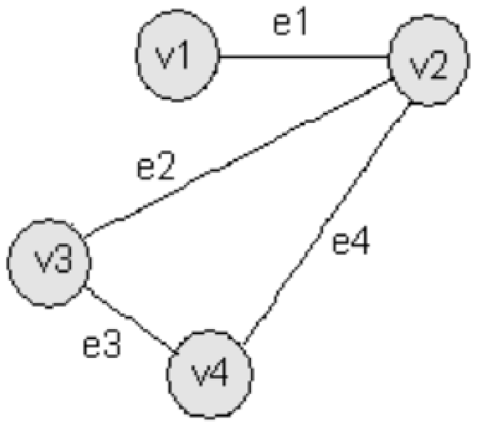
A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



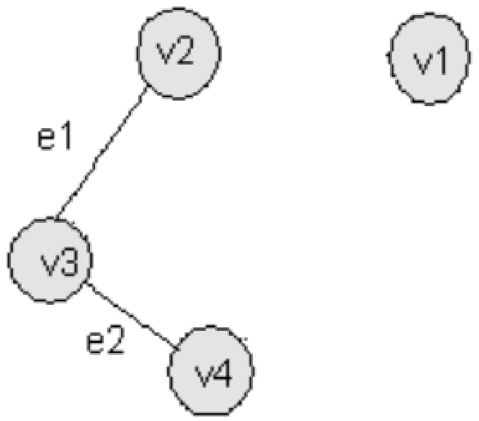
A weighted graph

Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G is disconnected.



A connected graph G



A disconnected graph G

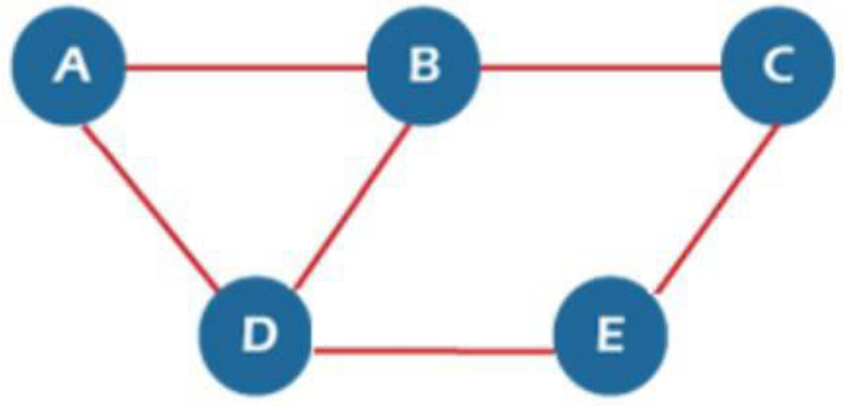
Graph representation

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 representation

- If adj[i][j] = w, it means that there is an edge exists from vertex i to vertex j with weight w.
- An entry A_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if an edge exists between V_i and V_j.

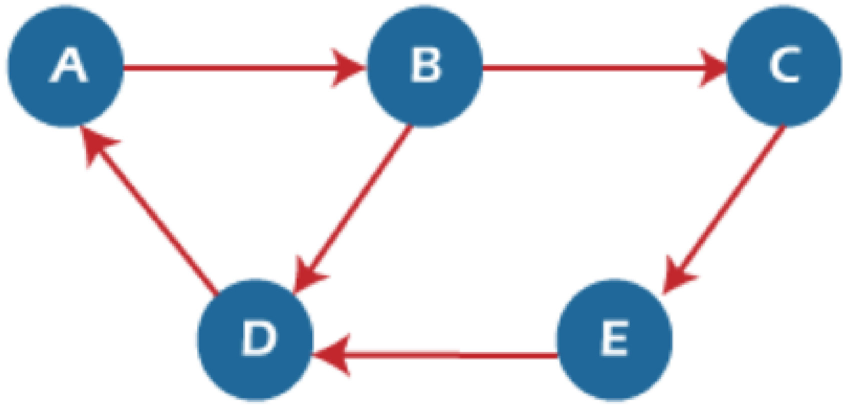


Undirected Graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

Adjacency Matrix

Adjacency matrix for a directed graph

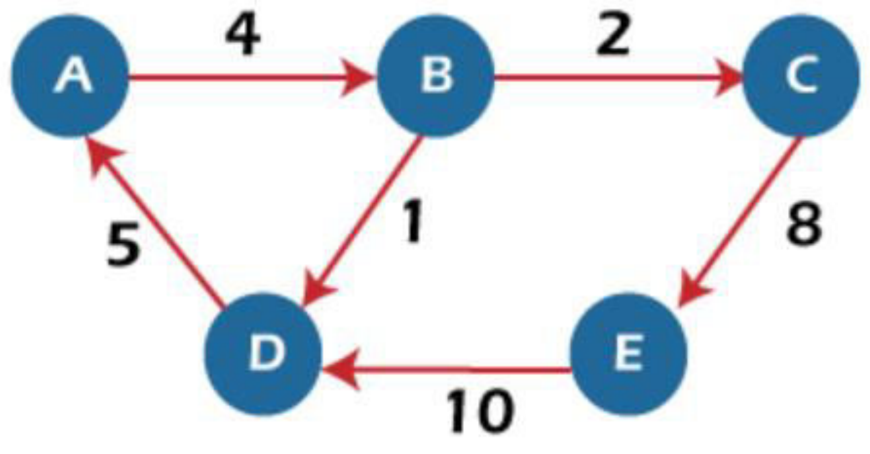


Directed Graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

Adjacency Matrix

Adjacency matrix for a weighted directed graph



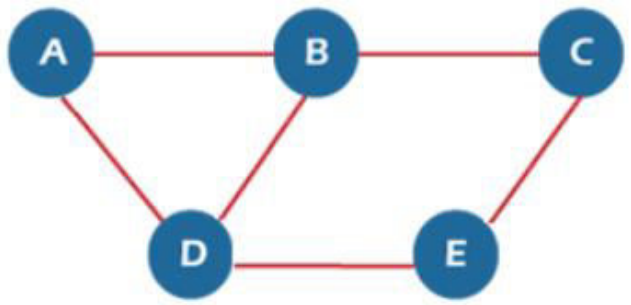
weighted Directed Graph

$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

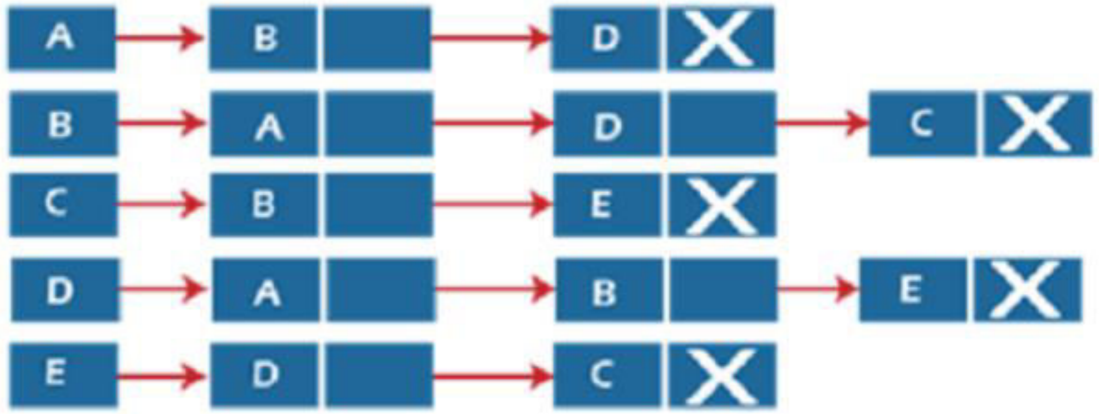
Adjacency Matrix

Linked list representation – Undirected graph

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.

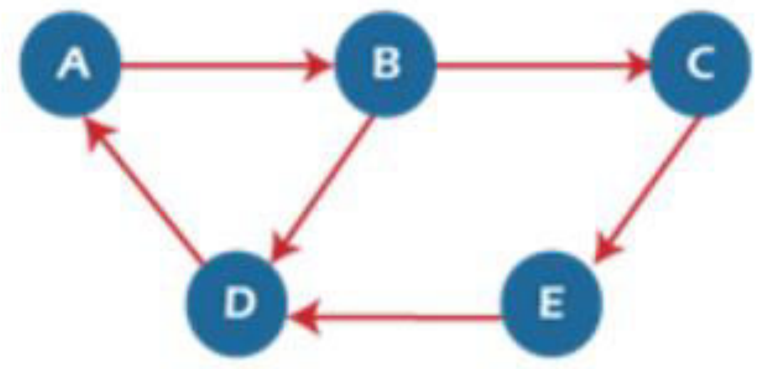


Undirected Graph

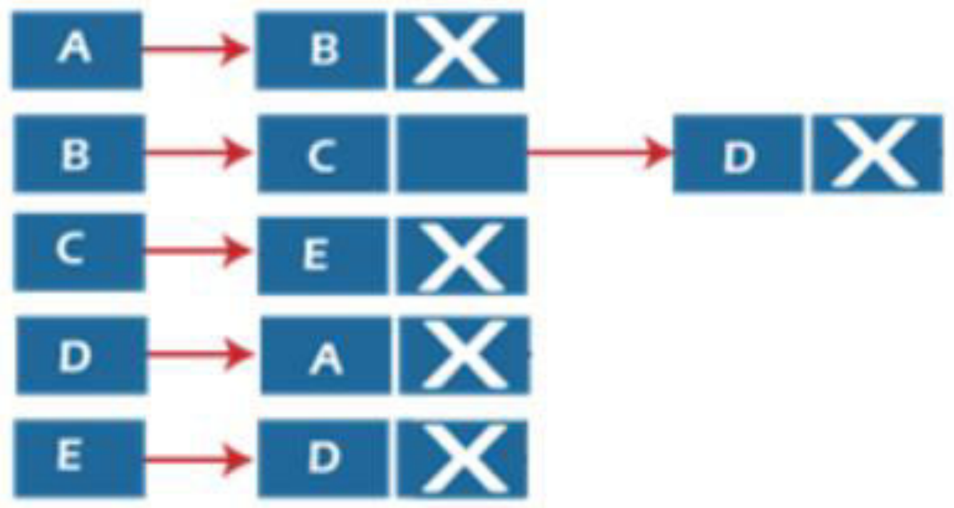


Adjacency List

Linked list representation – Directed Graph

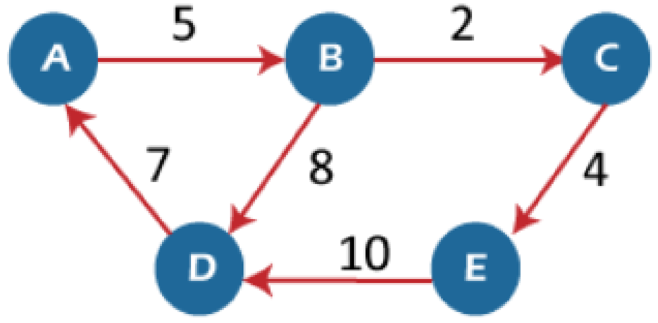


Directed Graph

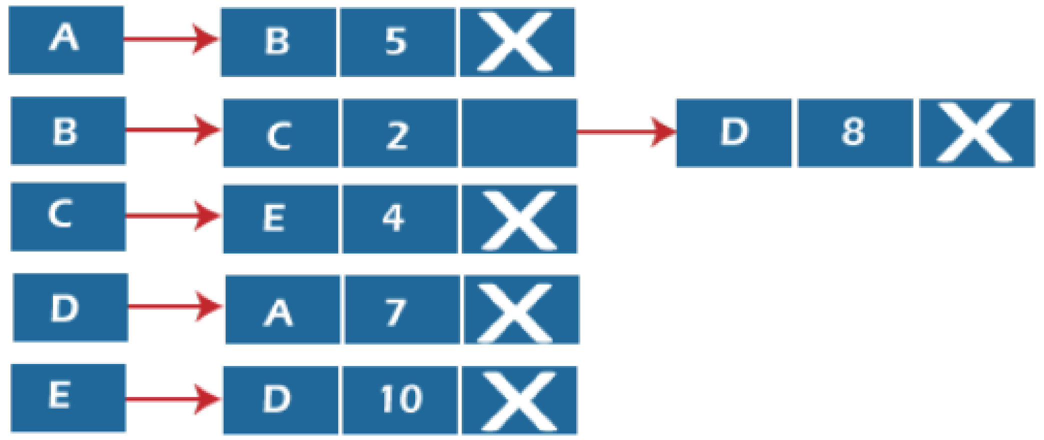


Adjacency List

Linked list representation – Weighted Graph



Weighted Directed Graph



Adjacency List

Graph Operations

The most common graph operations are:

- Check if the element is present in the graph
- Graph Traversal
- Add elements(vertex, edges) to graph
- Finding the path from one vertex to another

Graph Traversal (Depth/Breadth First Search)

- BFS or Breadth First Search starts from the top node in the graph and travels down until it reaches the root node.

- DFS or Depth First Search starts from the top node and follows a path to reaches the end node of the path.

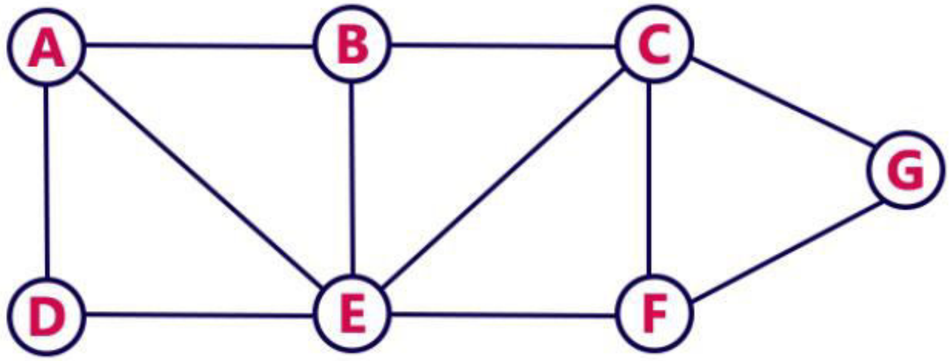
What is BFS?

•**Breadth First Search (BFS) algorithm** traverses a graph in a breadth-ward motion and uses a **queue** to remember to get the next vertex to start a search when a dead end occurs in any iteration.

•BFS is basically a node based algorithm which is used to find the shortest path in the graph between two nodes. BFS moves through all of its nodes which are connected to the individual nodes.

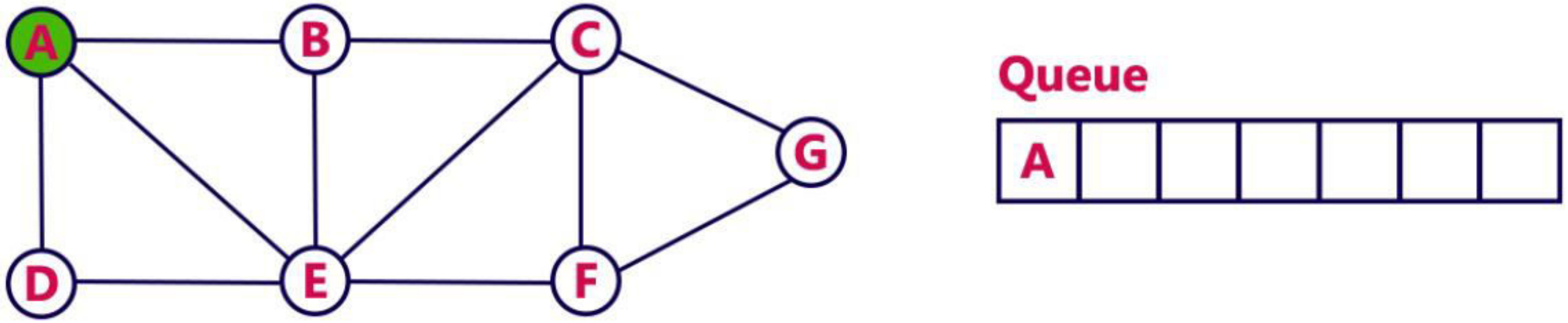
•BFS uses the FIFO (First In First Out) principle while using the Queue to find the shortest path. However, BFS is slower and requires a large memory space.

Consider the following example graph to perform BFS traversal



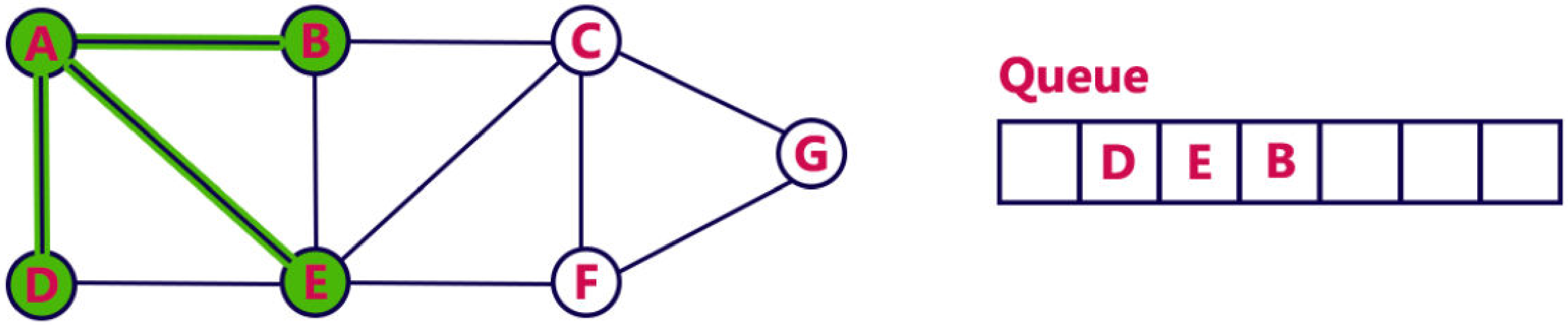
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



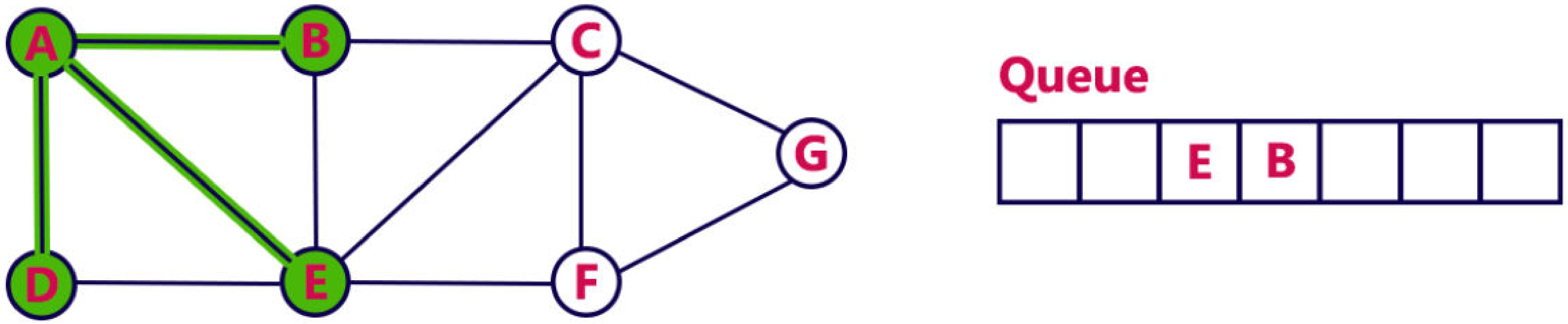
Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



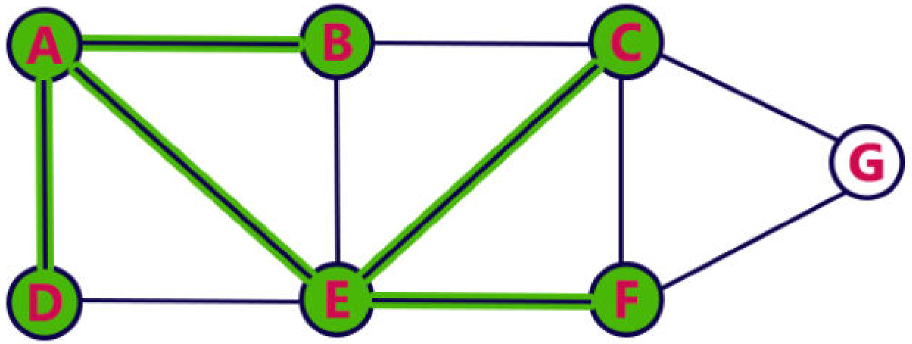
Step 3:

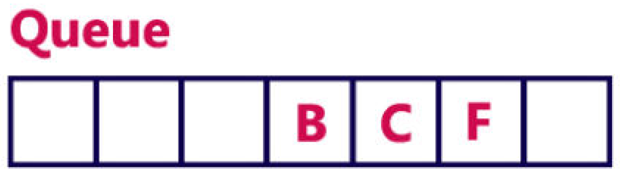
- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.



Step 4:

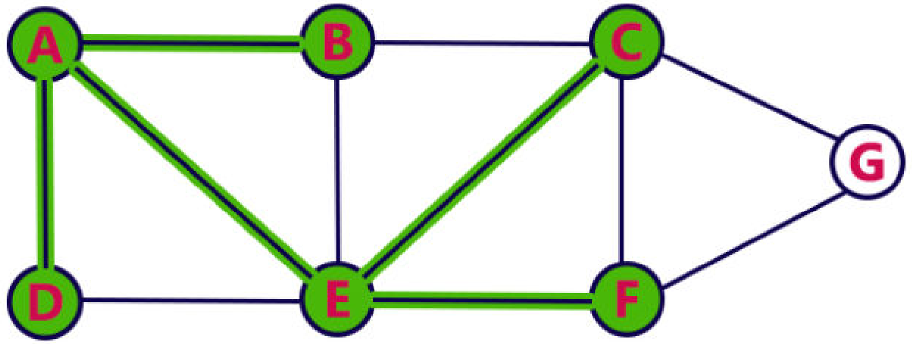
- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

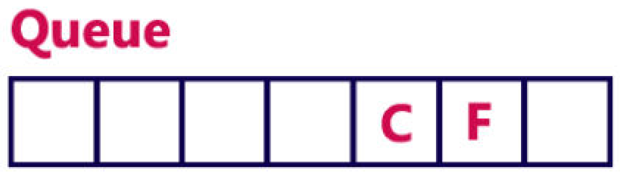




Step 5:

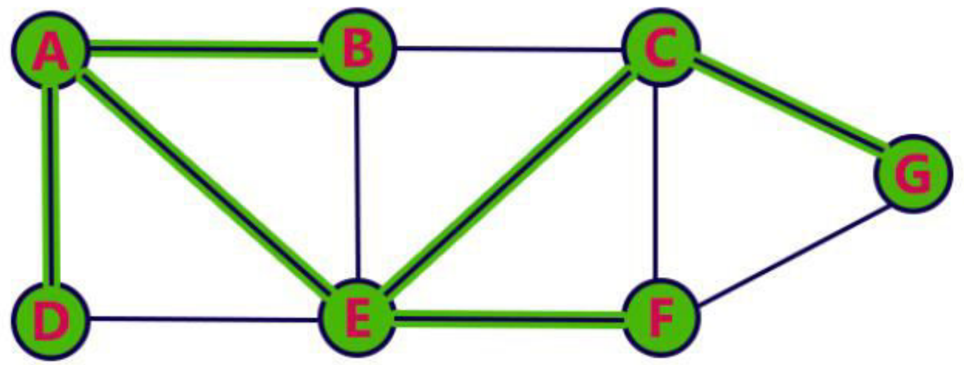
- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

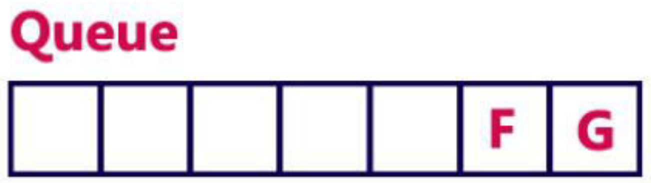




Step 6:

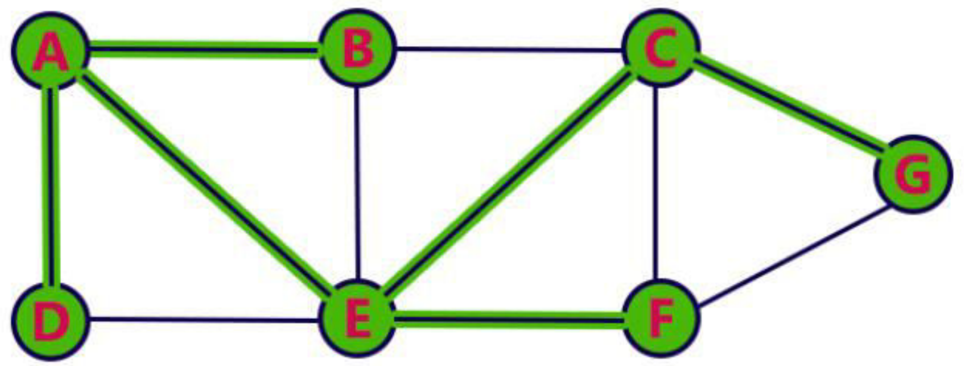
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

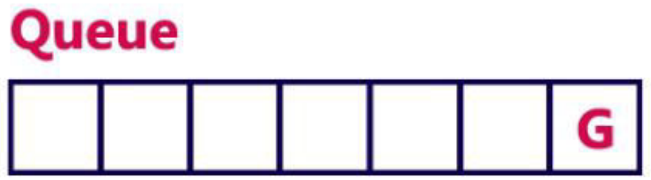




Step 7:

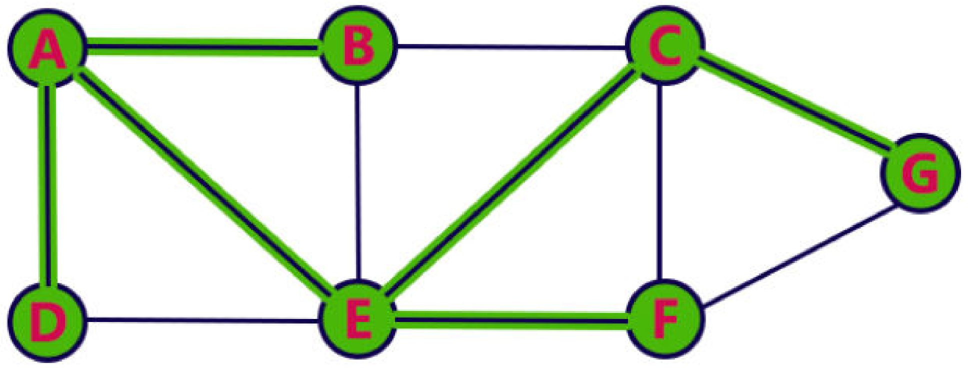
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



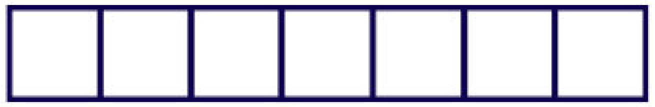


Step 8:

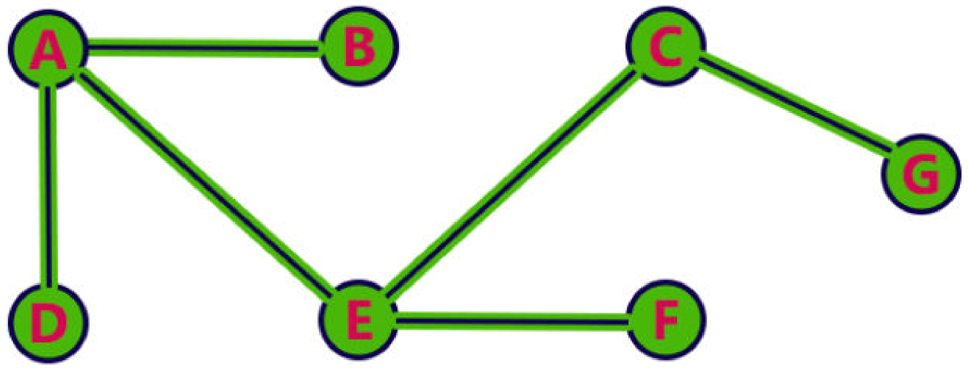
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



We use the following steps to implement BFS traversal...

Step 1 - Define a Queue of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

Step 5 - Repeat steps 3 and 4 until queue becomes empty.

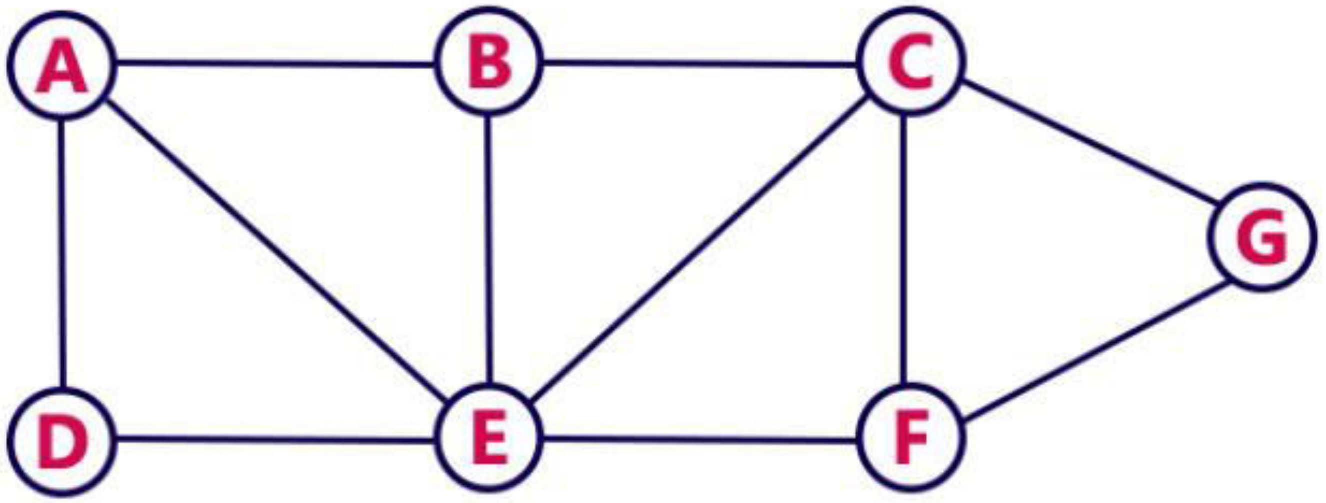
Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

What is DFS?

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

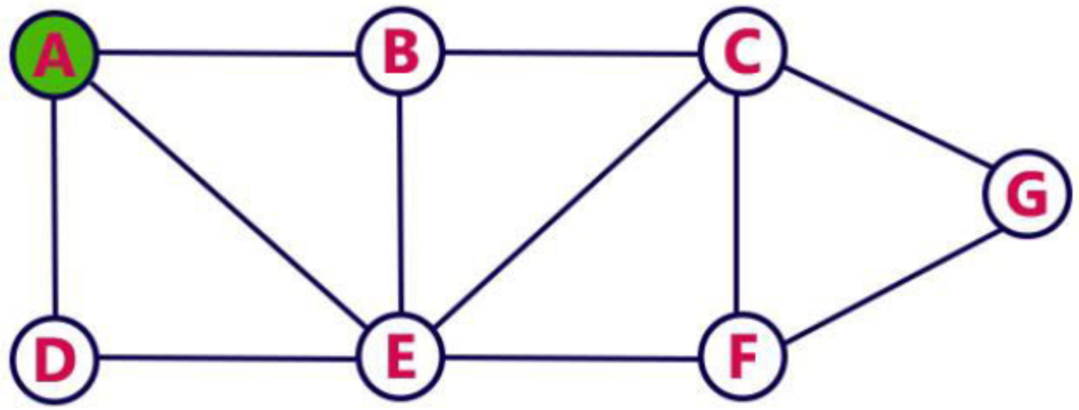
•DFS uses LIFO (Last In First Out) principle while using Stack to find the shortest path. DFS is also called **Edge Based Traversal** because it explores the nodes along the edge or path. DFS is faster and requires less memory. DFS is best suited for decision trees.

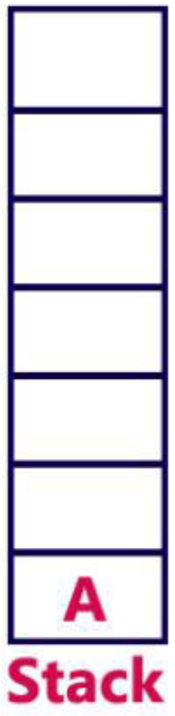
Consider the following example graph to perform DFS traversal



Step 1:

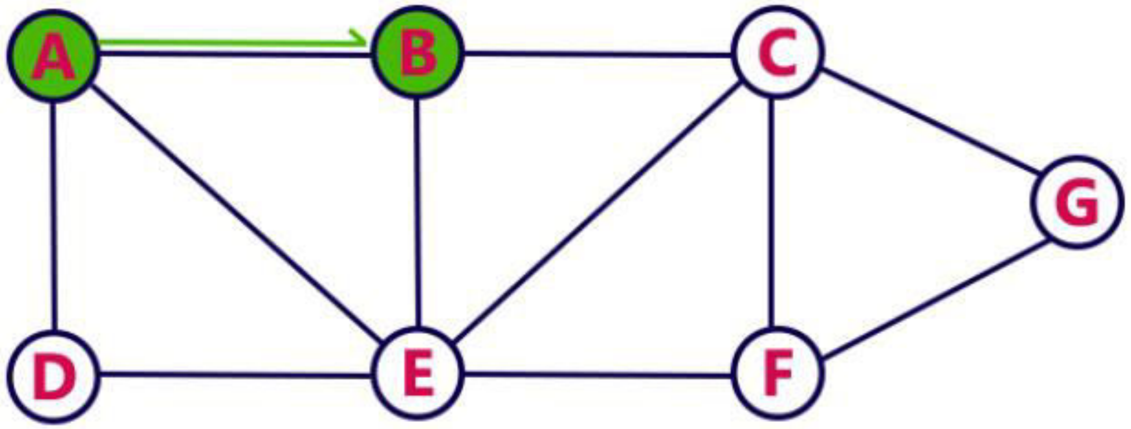
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

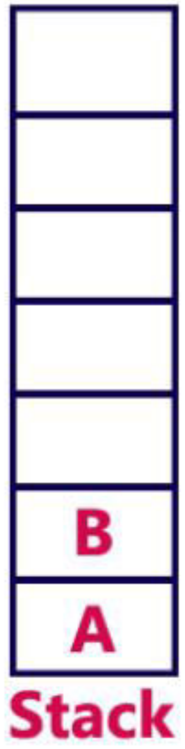




Step 2:

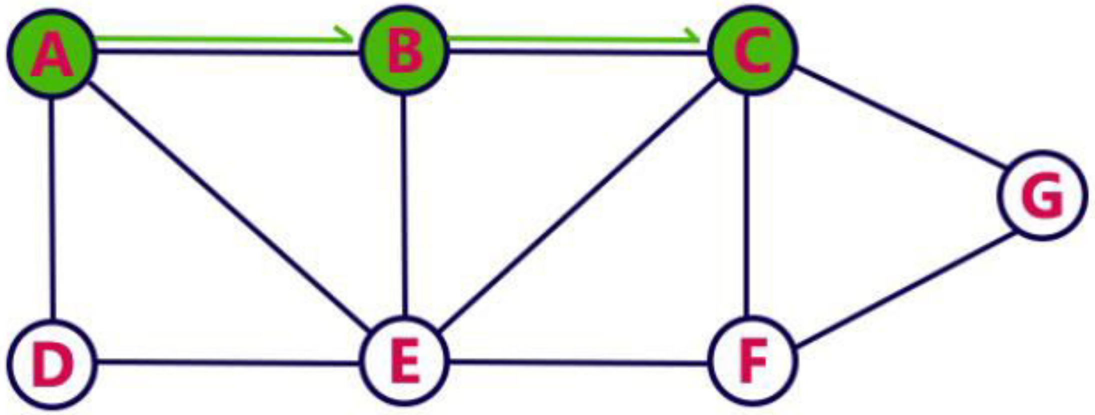
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

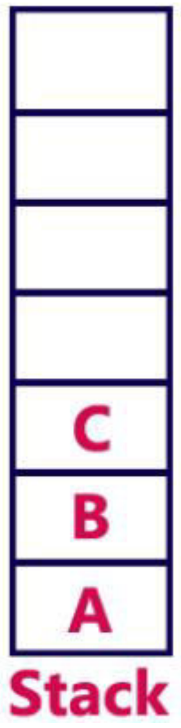




Step 3:

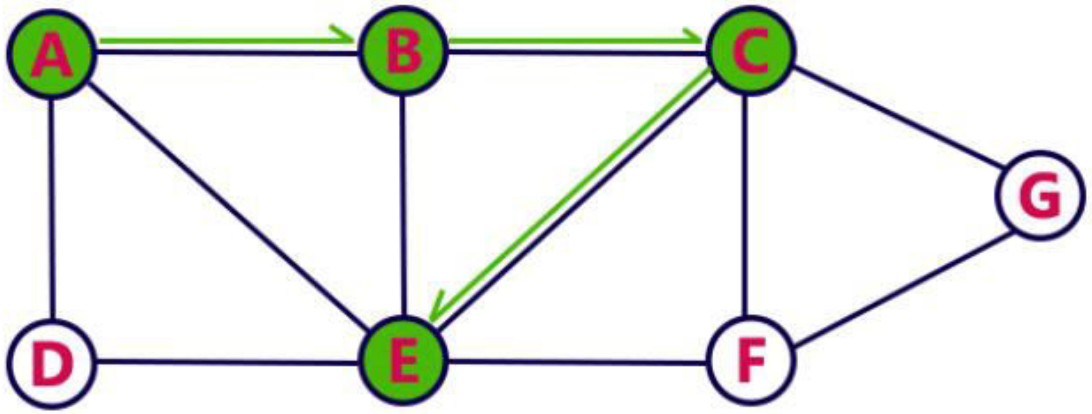
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.

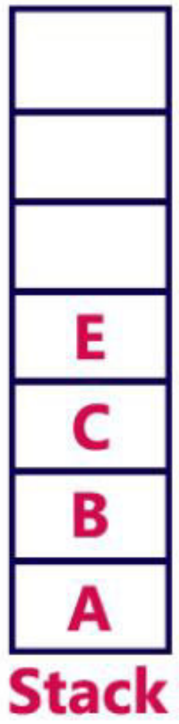




Step 4:

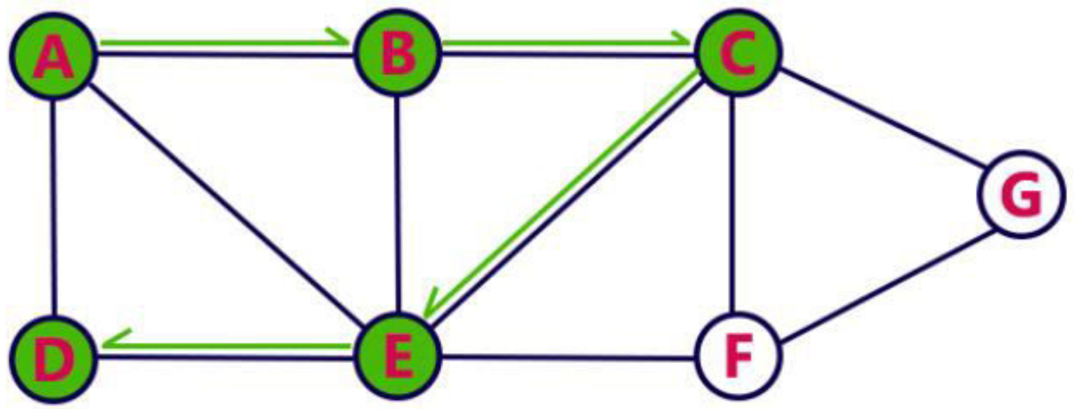
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

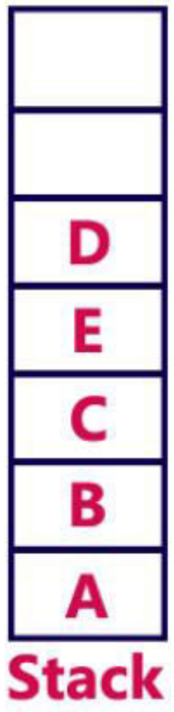




Step 5:

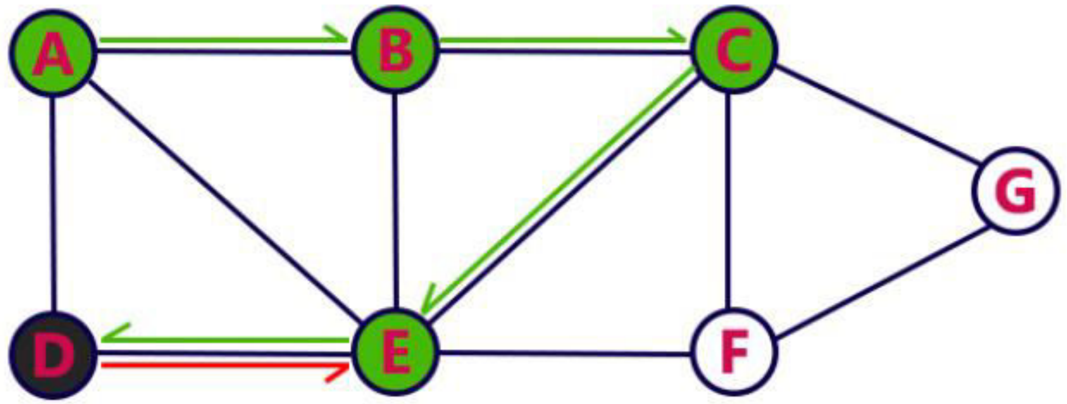
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

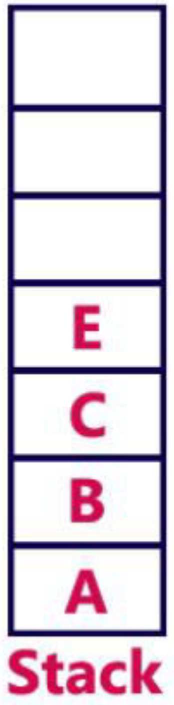




Step 6:

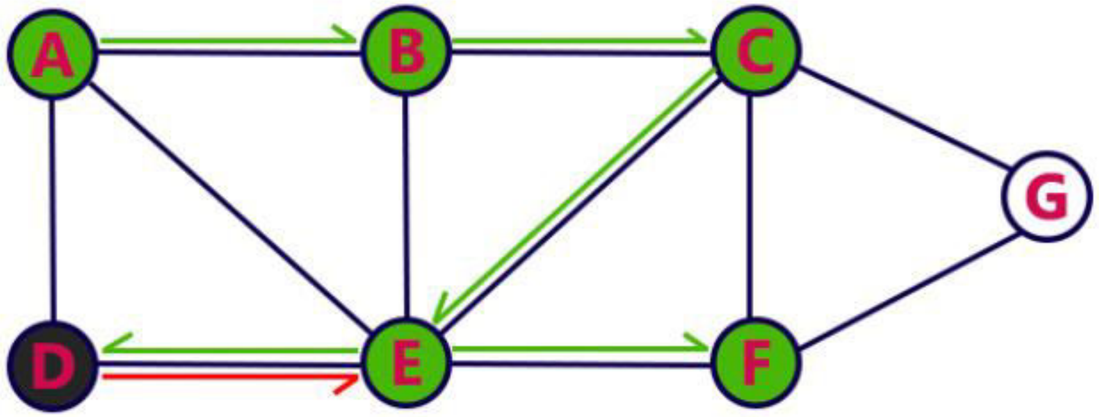
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

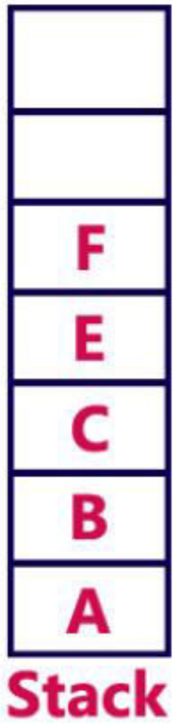




Step 7:

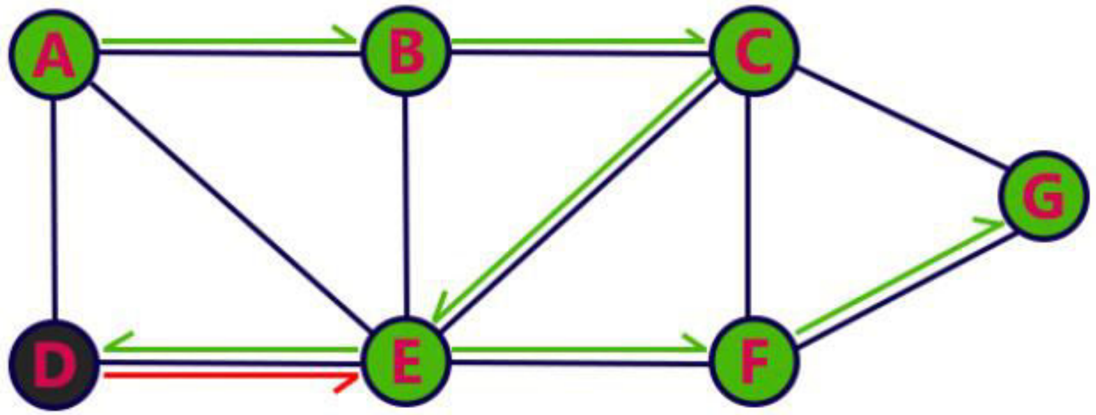
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

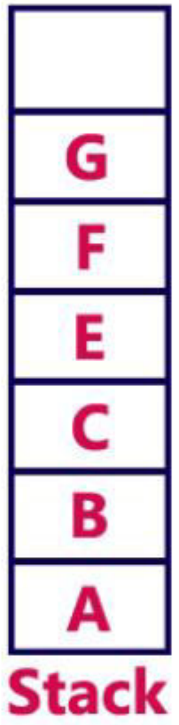




Step 8:

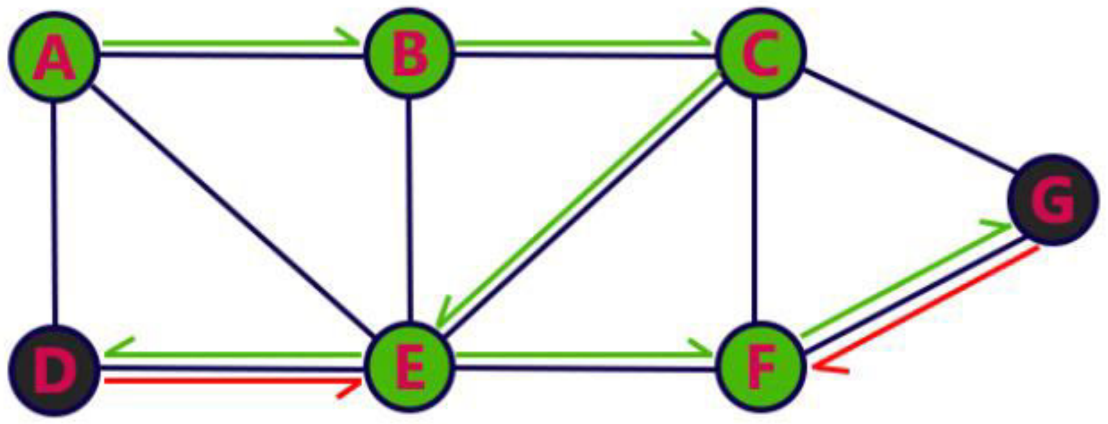
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

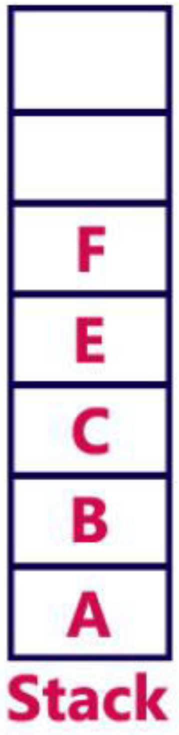




Step 9:

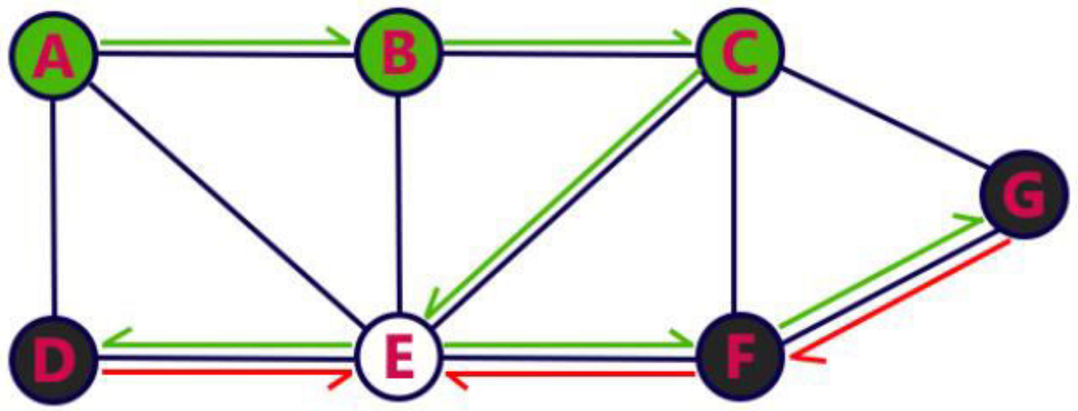
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

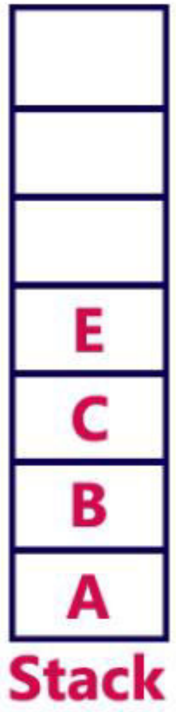




Step 10:

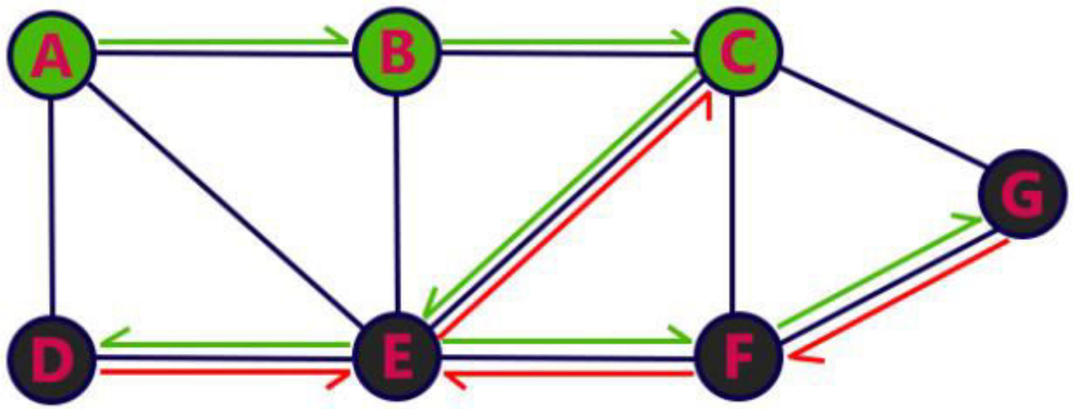
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

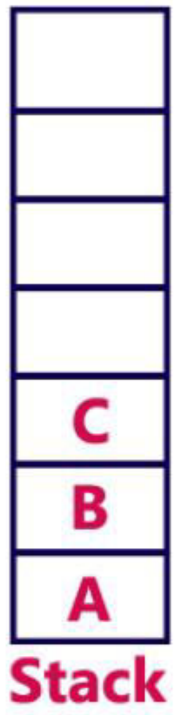




Step 11:

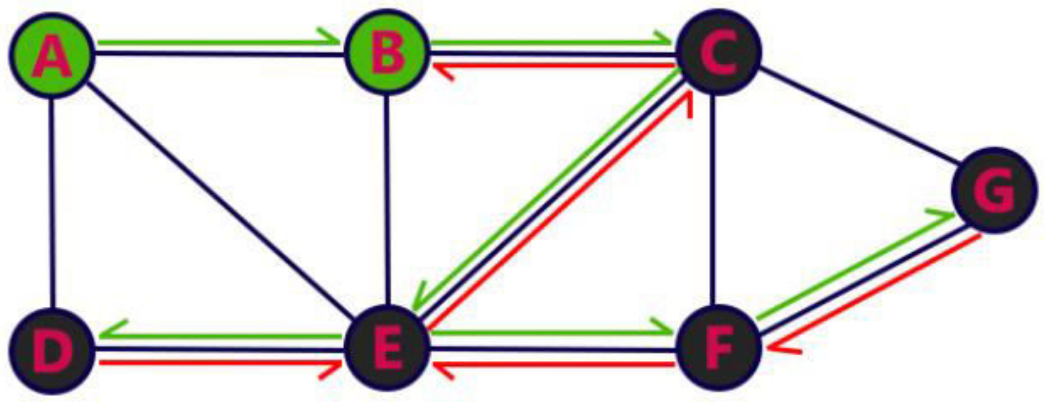
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

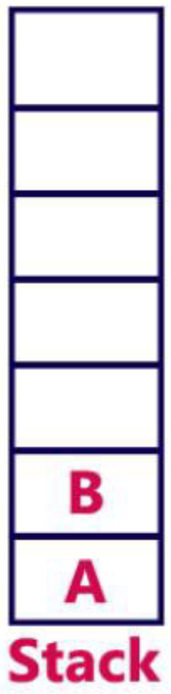




Step 12:

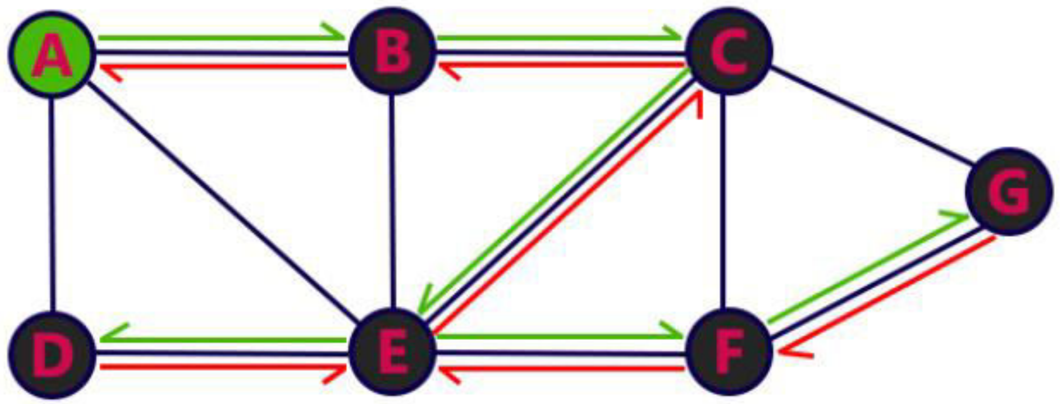
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

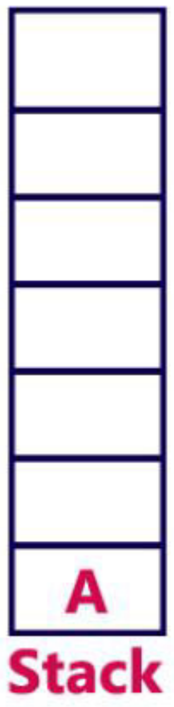




Step 13:

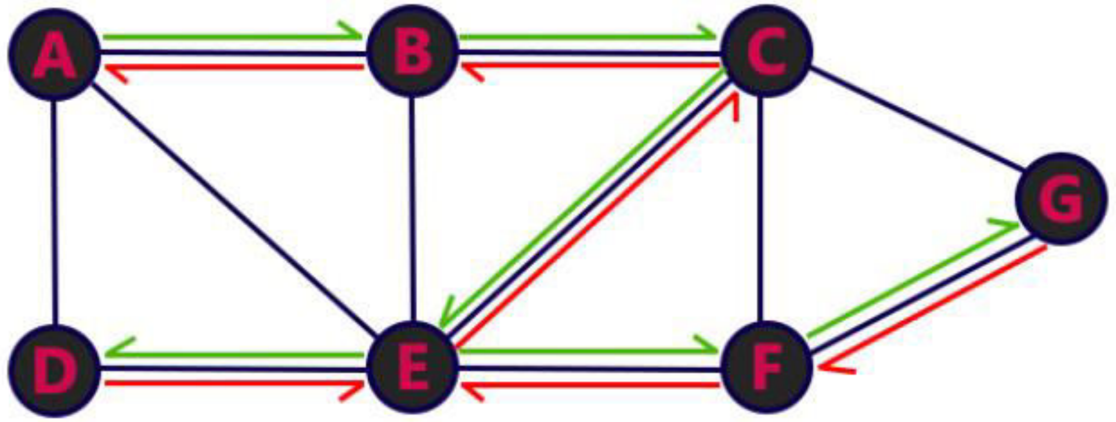
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

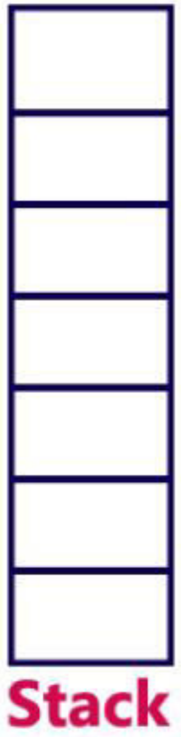




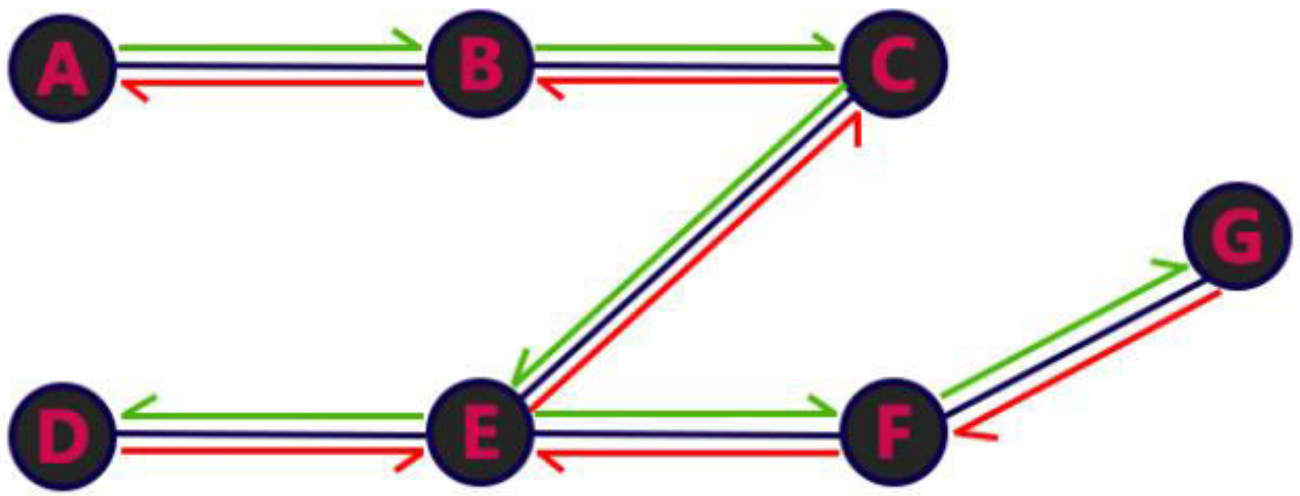
Step 14:

- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.





- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



We use the following steps to implement DFS traversal...

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

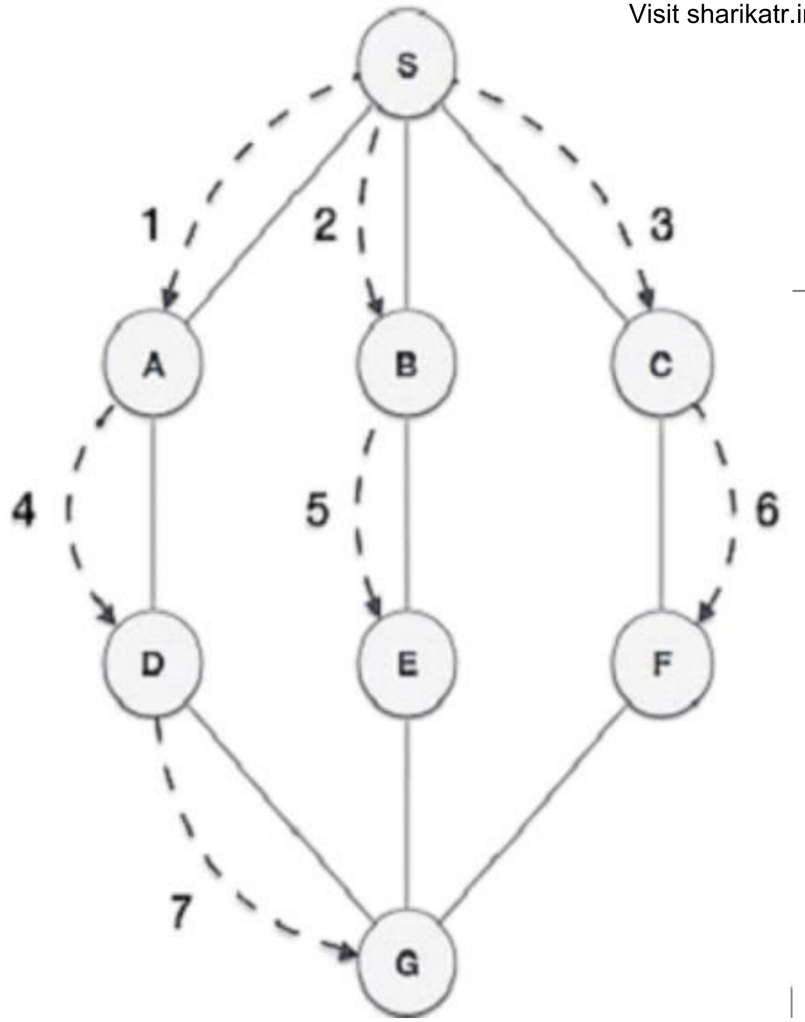
Step 3 - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

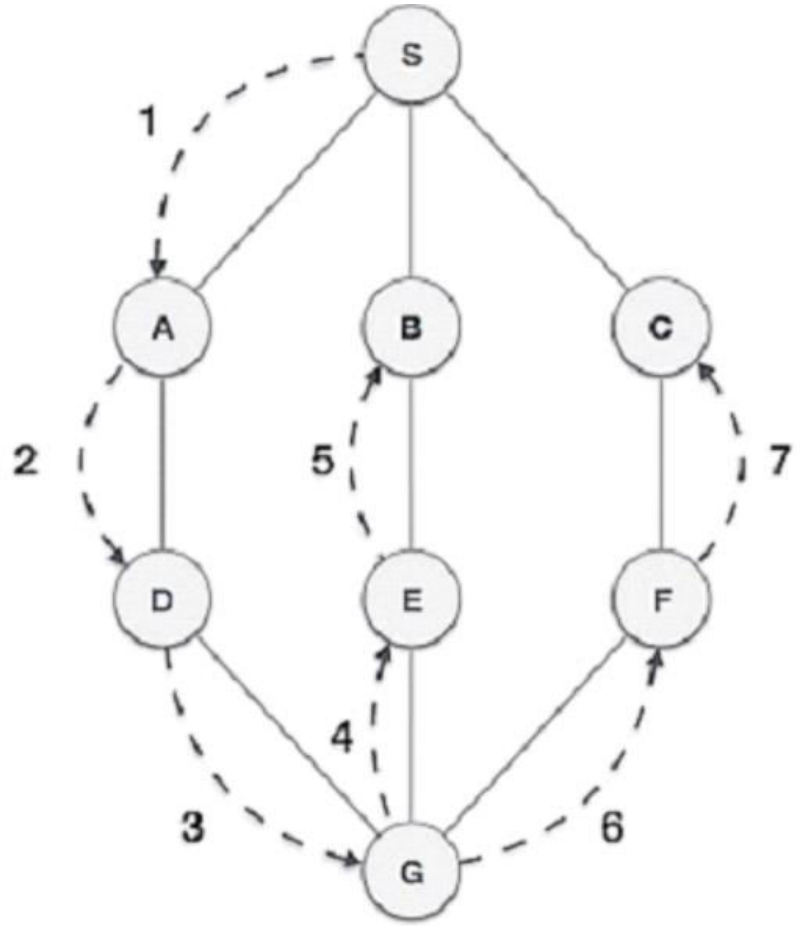
Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

Step 5 - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph





Example of BFS

Example of DFS

Difference between BFS and DFS

Key	BFS	DFS
Definition	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data structure	BFS uses a Queue to find the shortest path.	DFS uses a Stack to find the shortest path.
Source	BFS is better when target is closer to Source.	DFS is better when target is far from source.
Suitability for decision tree	As BFS considers all neighbour so it is not suitable for decision tree used in puzzle games.	DFS is more suitable for decision tree. As with one decision, we need to traverse further to augment the decision. If we reach the conclusion, we won.
Speed	BFS is slower than DFS.	DFS is faster than BFS.
Time Complexity	Time Complexity of BFS = O(V+E) where V is vertices and E is edges.	Time Complexity of DFS is also O(V+E) where V is vertices and E is edges.
Memory	BFS requires more memory space.	DFS requires less memory space.
Tapping in loops	In BFS, there is no problem of trapping into finite loops.	In DFS, we may be trapped into infinite loops.
Principle	BFS is implemented using FIFO (First In First Out) principle.	DFS is implemented using LIFO (Last In First Out) principle

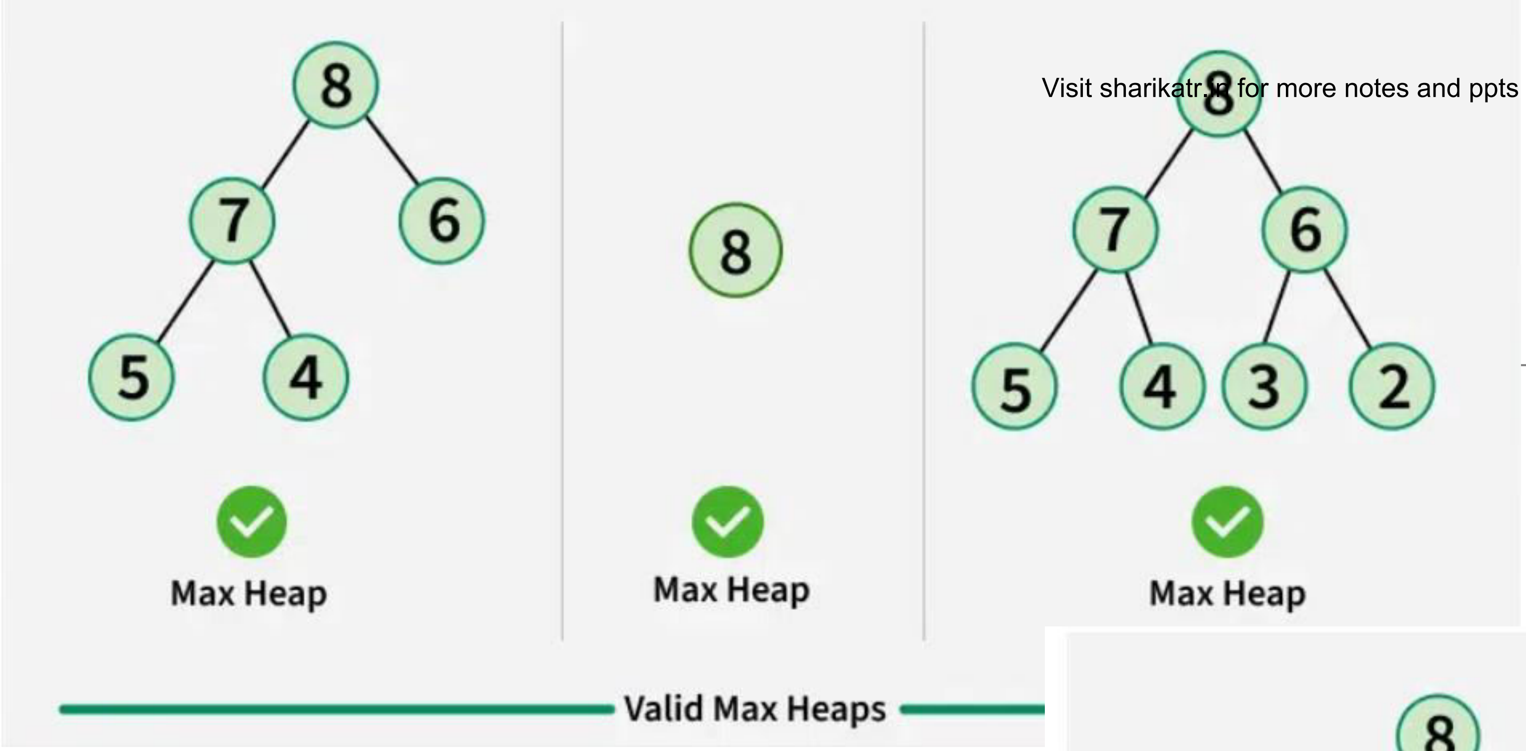
Binary Heap

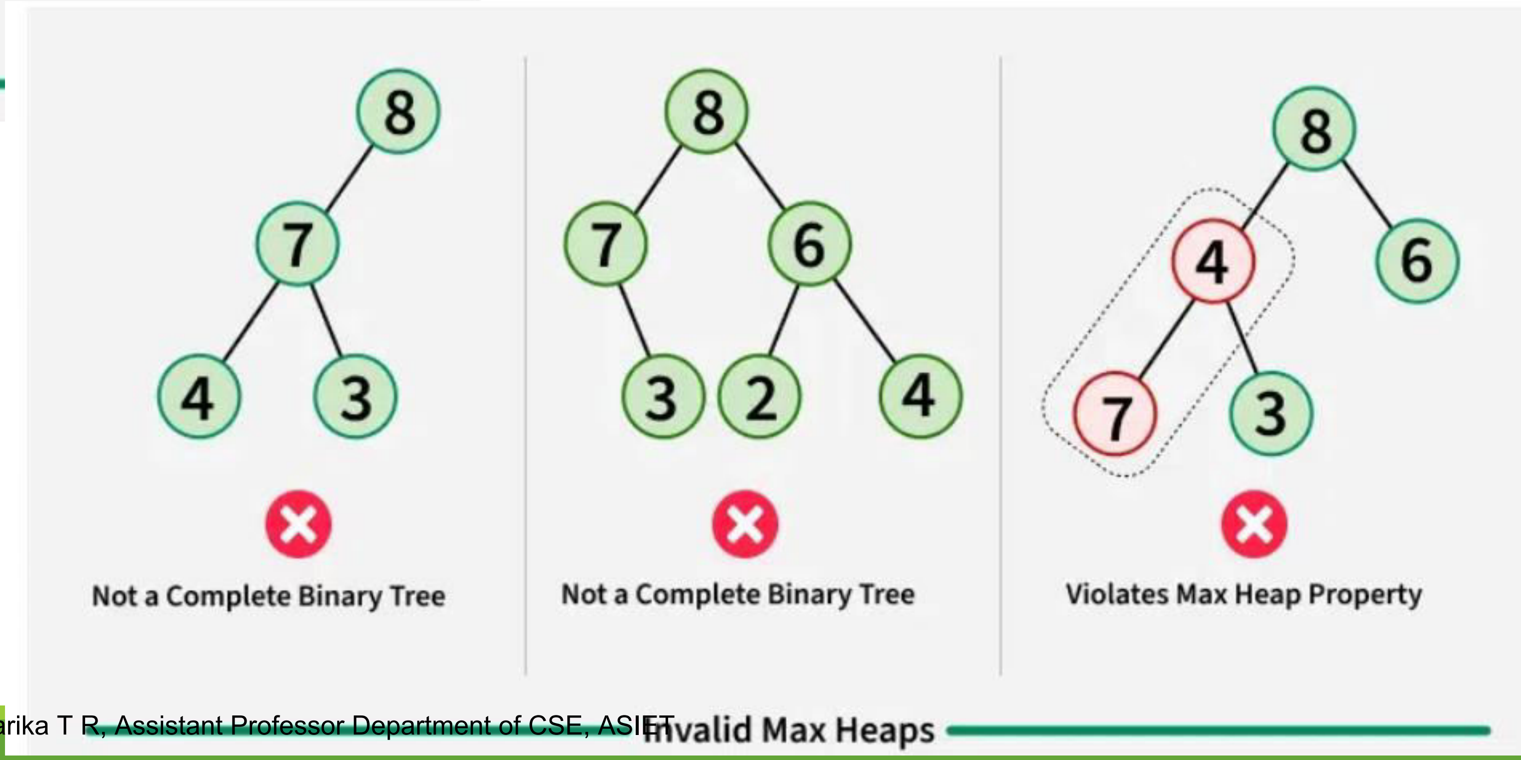
A Binary Heap is a complete binary tree that stores data efficiently, allowing quick access to the maximum or minimum element, depending on the type of heap.

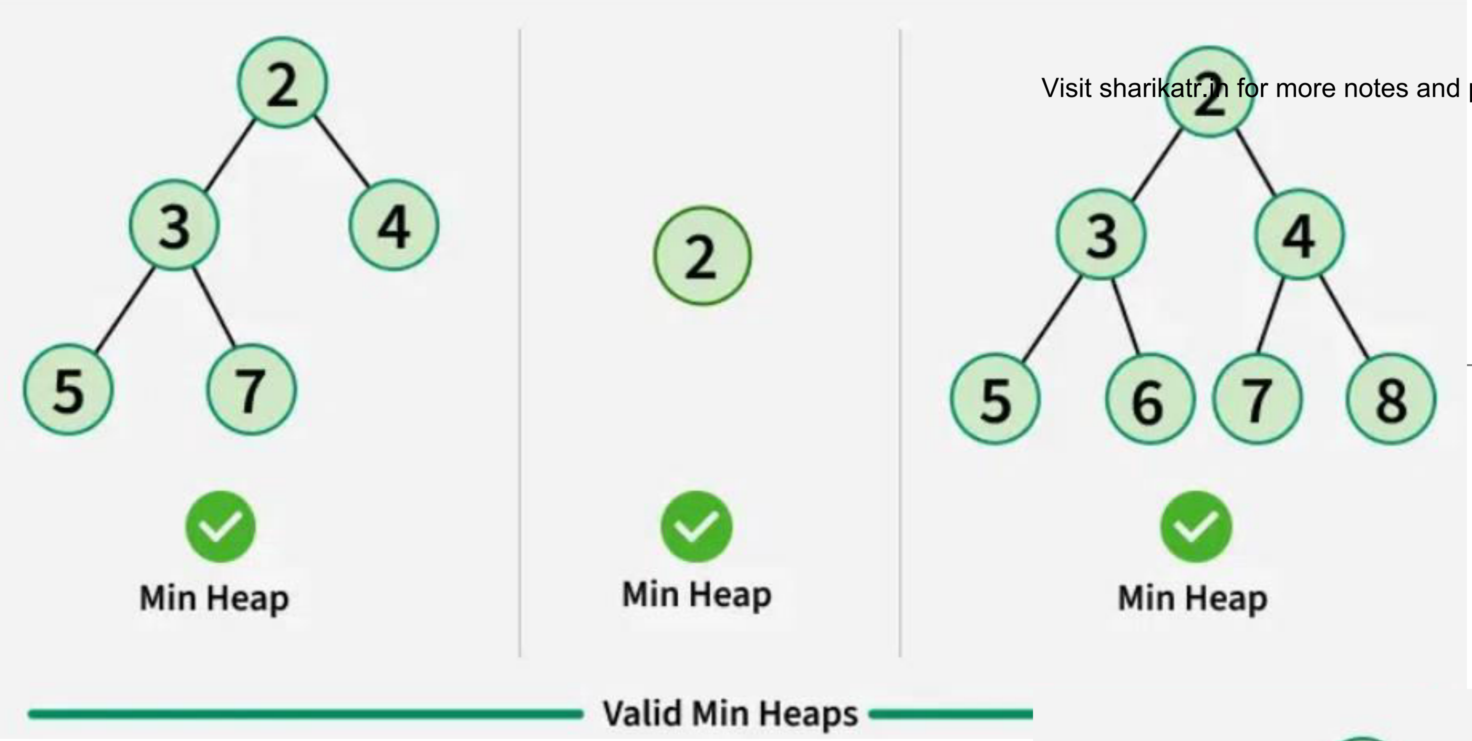
It can either be a Min Heap or a Max Heap.

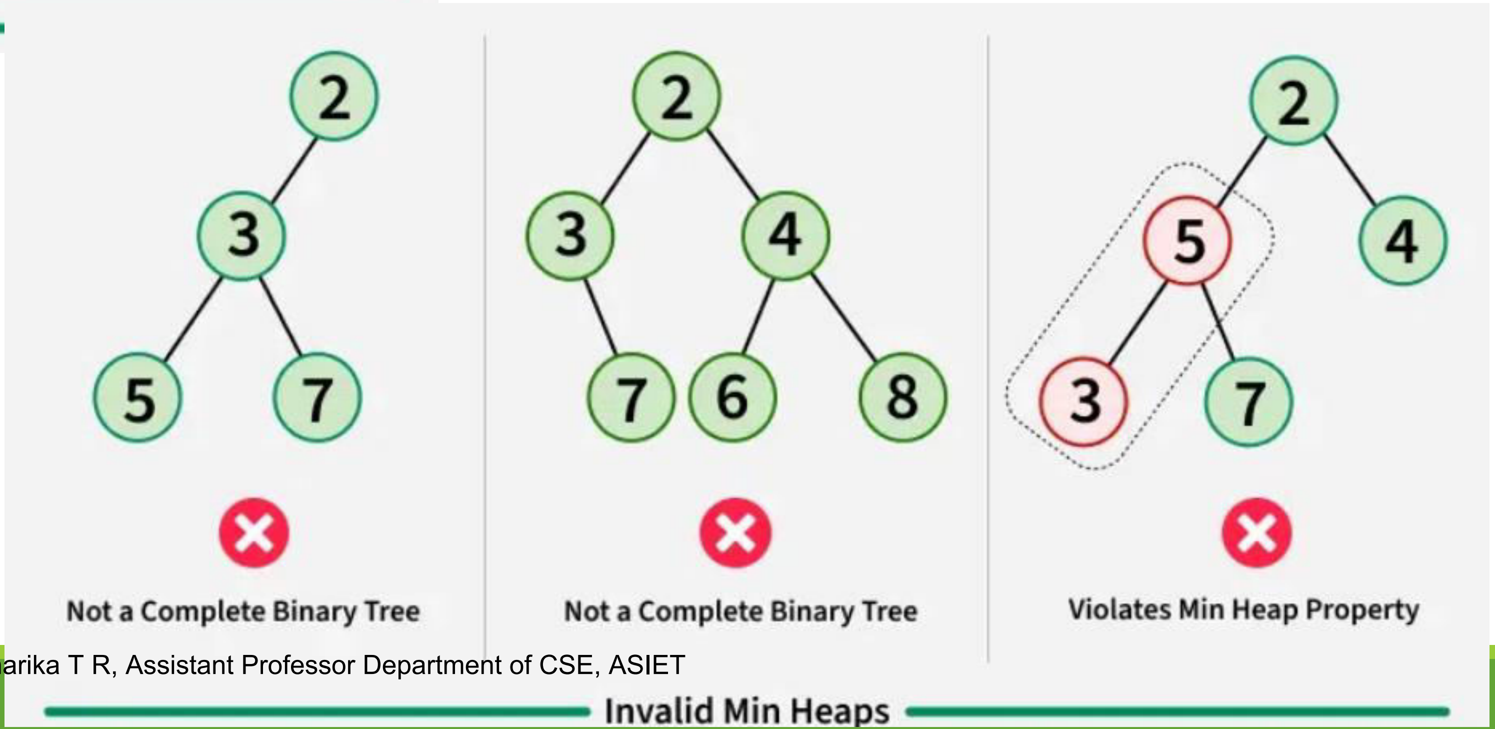
Min Heap→ the key at the root must be the smallest among all the keys in the heap, and this property must hold true recursively for all nodes.

Max Heap→ follows the same principle, but with the largest key at the root.

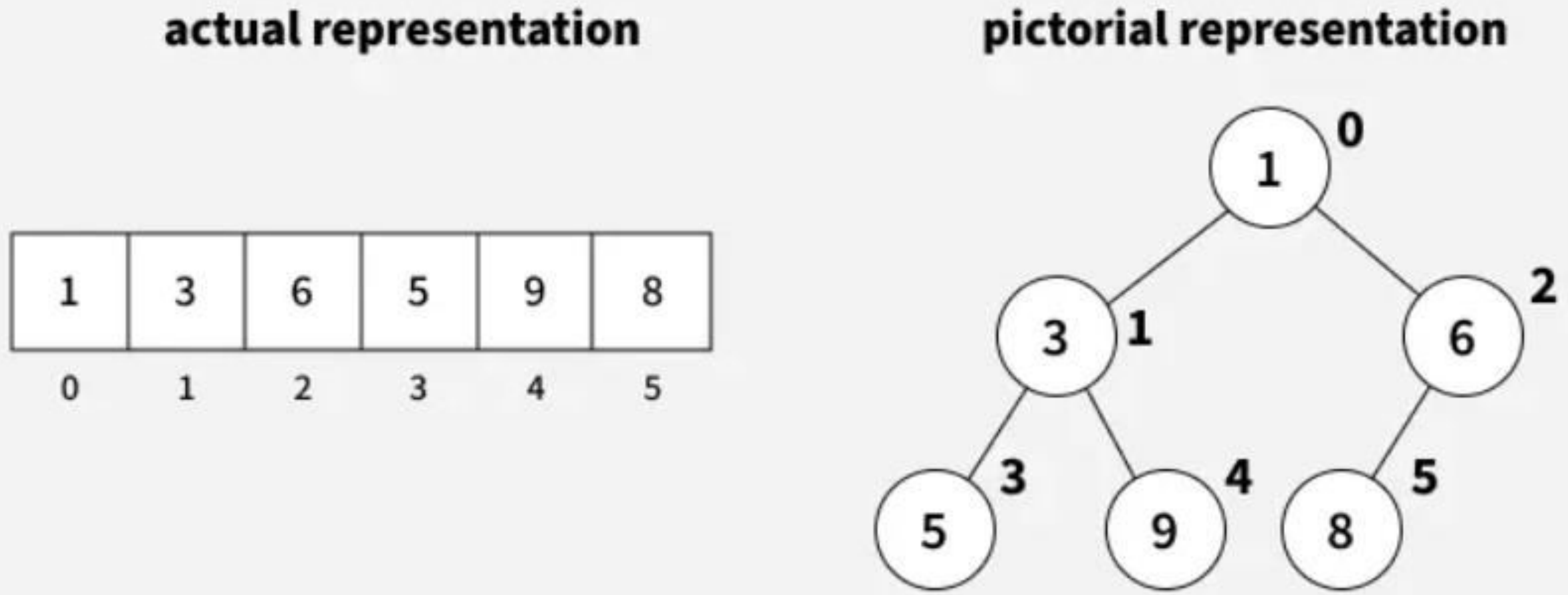

Max Heap
Max Heap
Max Heap
Valid Max Heaps


Not a Complete Binary Tree
Not a Complete Binary Tree
Violates Max Heap Property
Invalid Max Heaps


Min Heap
Min Heap
Min Heap
Valid Min Heaps


Not a Complete Binary Tree
Not a Complete Binary Tree
Violates Min Heap Property
Invalid Min Heaps

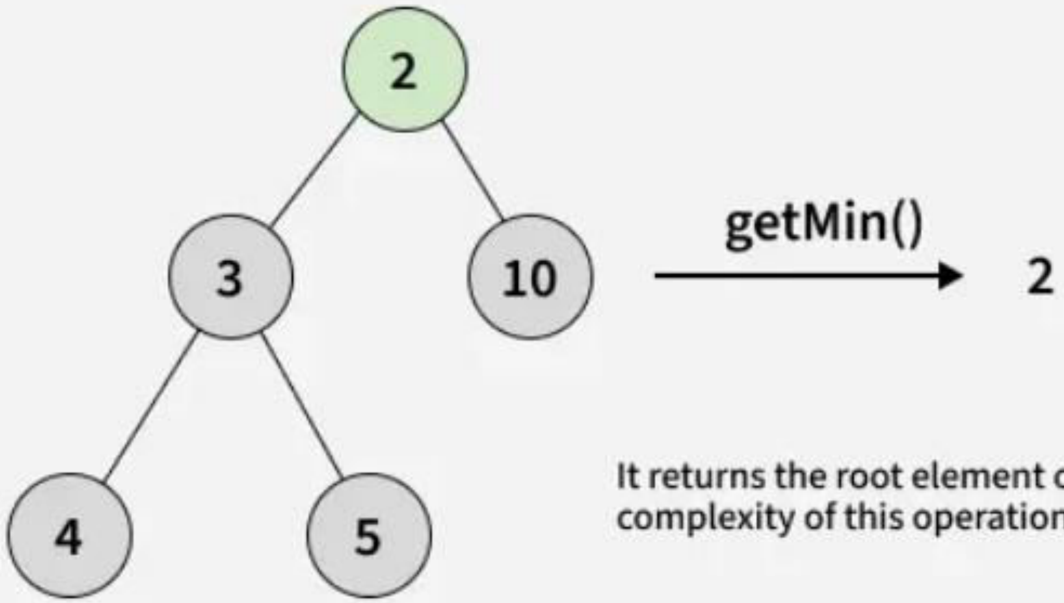
A heap is often represented as an array rather than a tree structure. In this array-based representation, for any element at index i, the left child is at index 2i + 1 and the right child is at index 2i + 2. The parent of any element at index i is at index (i-1)/2



Representation of a Binary Heap

Operations on Heap

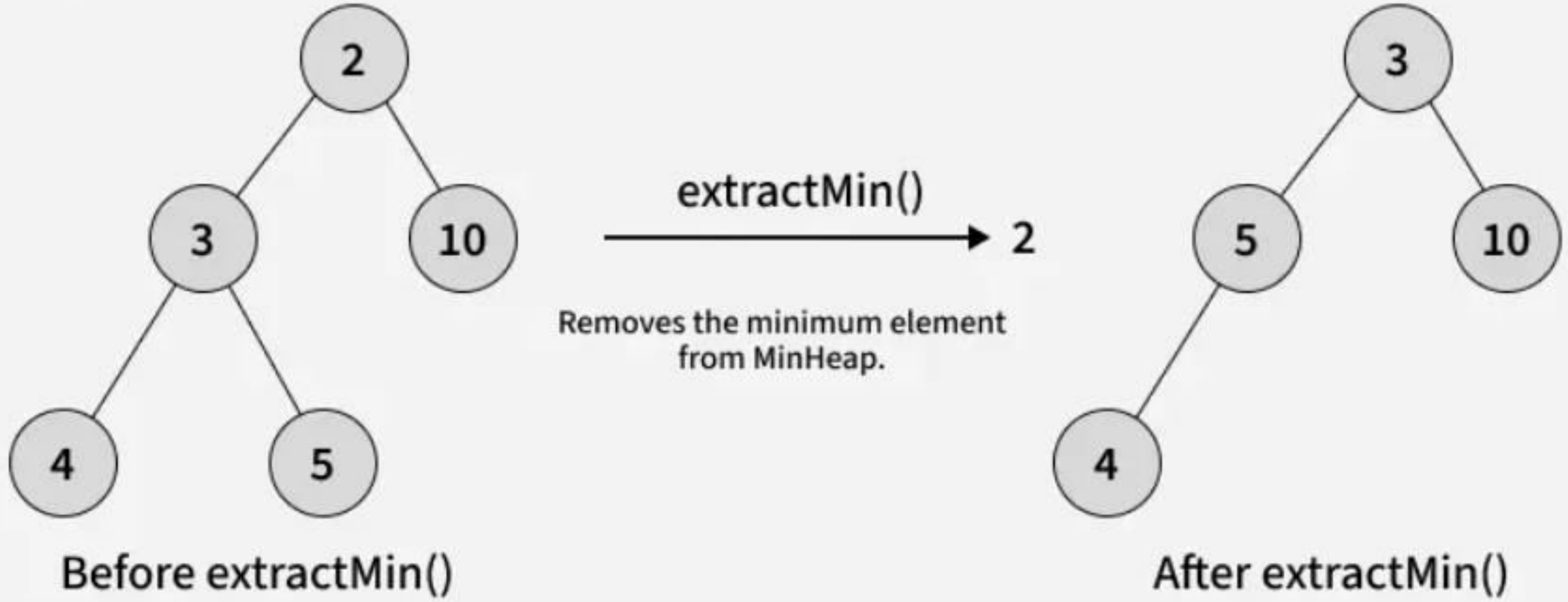
01 | getMin()



It returns the root element of Min Heap. The time complexity of this operation is O(1).

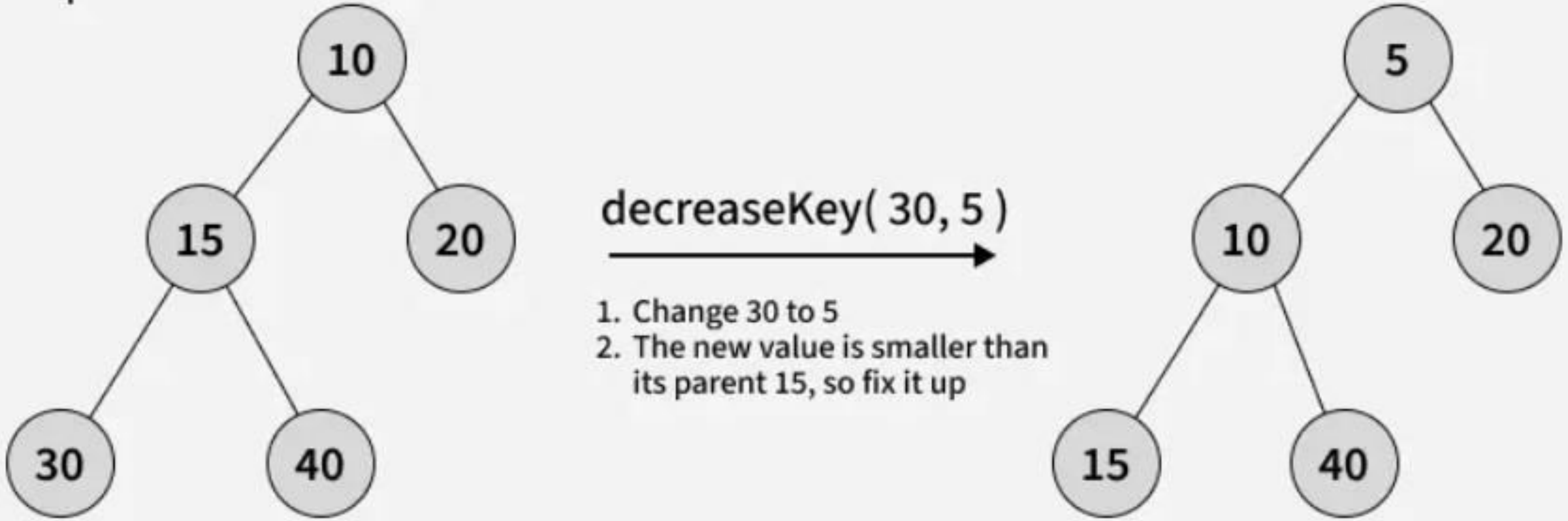
Operations on Heap

02 | extractMin()



Operations on Heap

03 | decreaseKey()

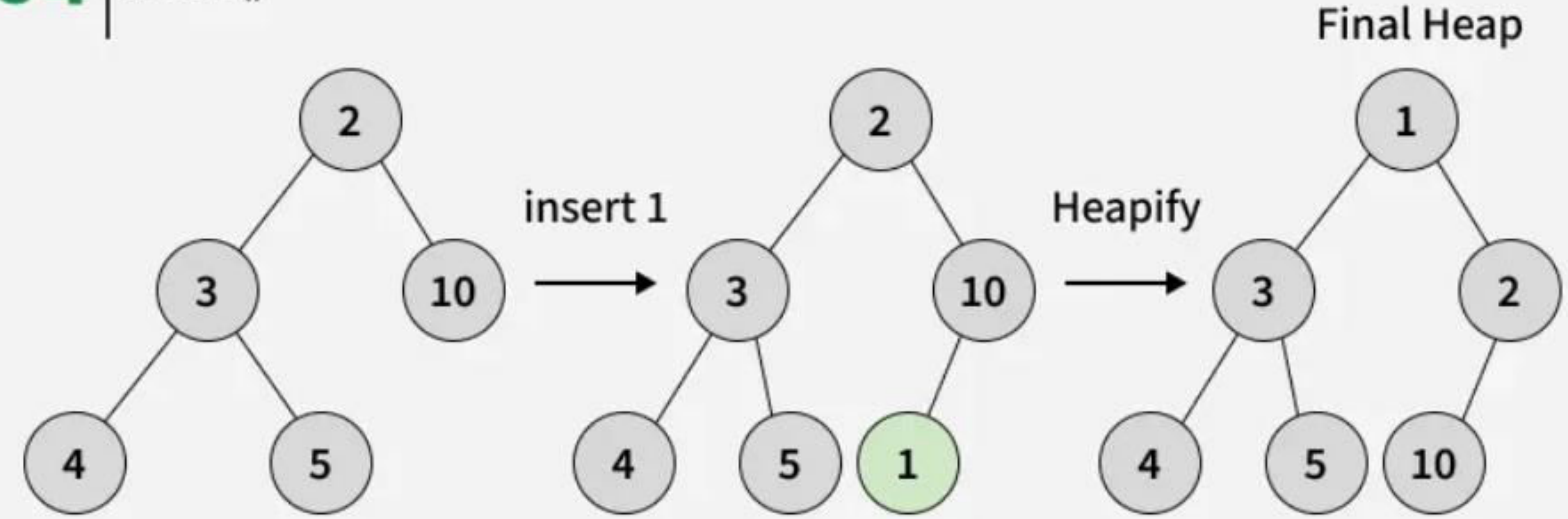


1. Change 30 to 5
2. The new value is smaller than its parent 15, so fix it up

Decrease the value of the key. If the decreased key value of a node is greater than the parent of the node, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

Operations on Heap

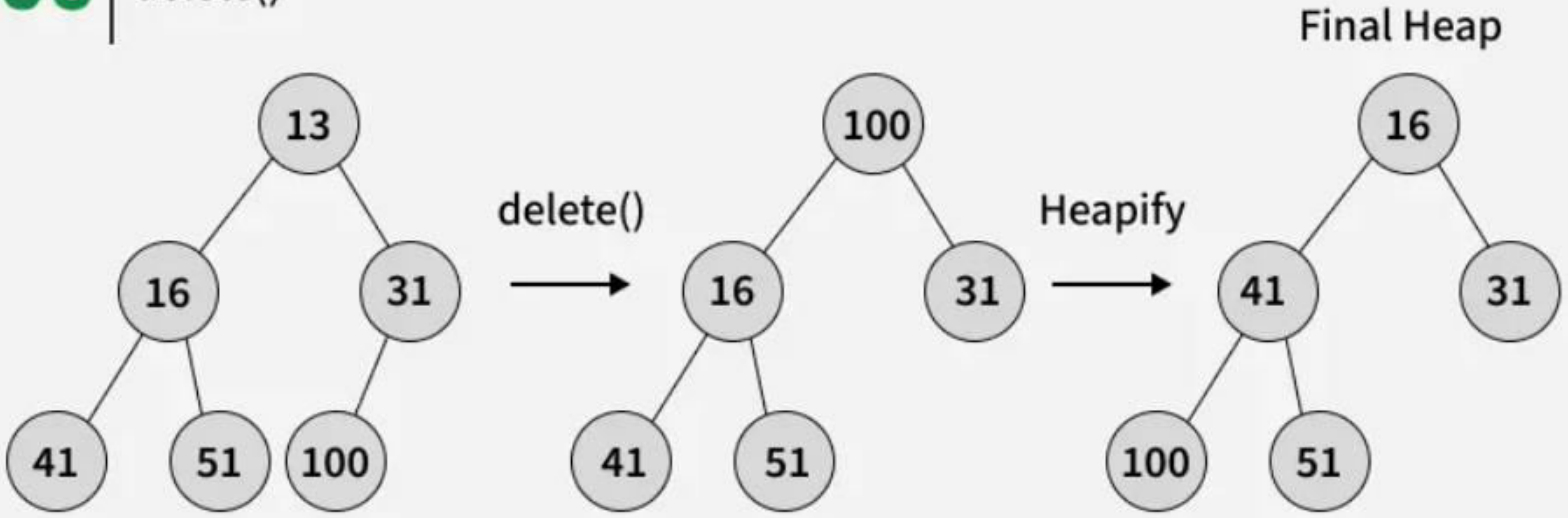
04 insert()



Add a new key at the end of the tree. If the new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.

Operations on Heap

05 | delete()



Replace the root element with the last element in the heap and remove the last element. Since the new root may violate the heap property, heapify the root to restore the heap structure.

Operations on Heap

MODULE 3 ENDS