Binary Search Tree

Day 8 : Algorithms and Data Structures

Topic: Introduction to ADS

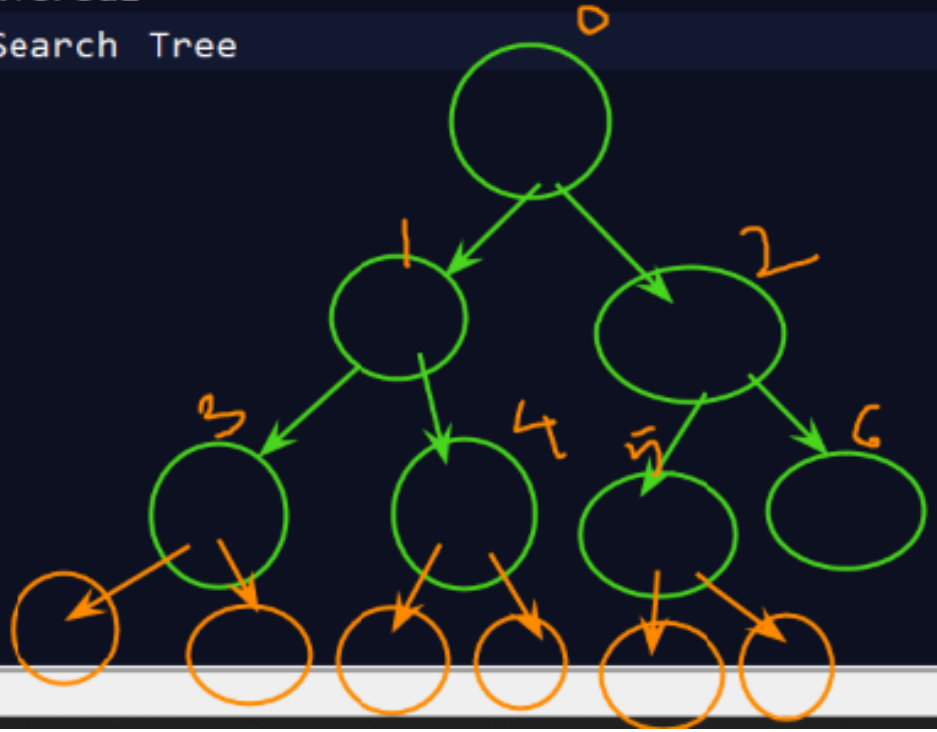
Date: 02/10/2024

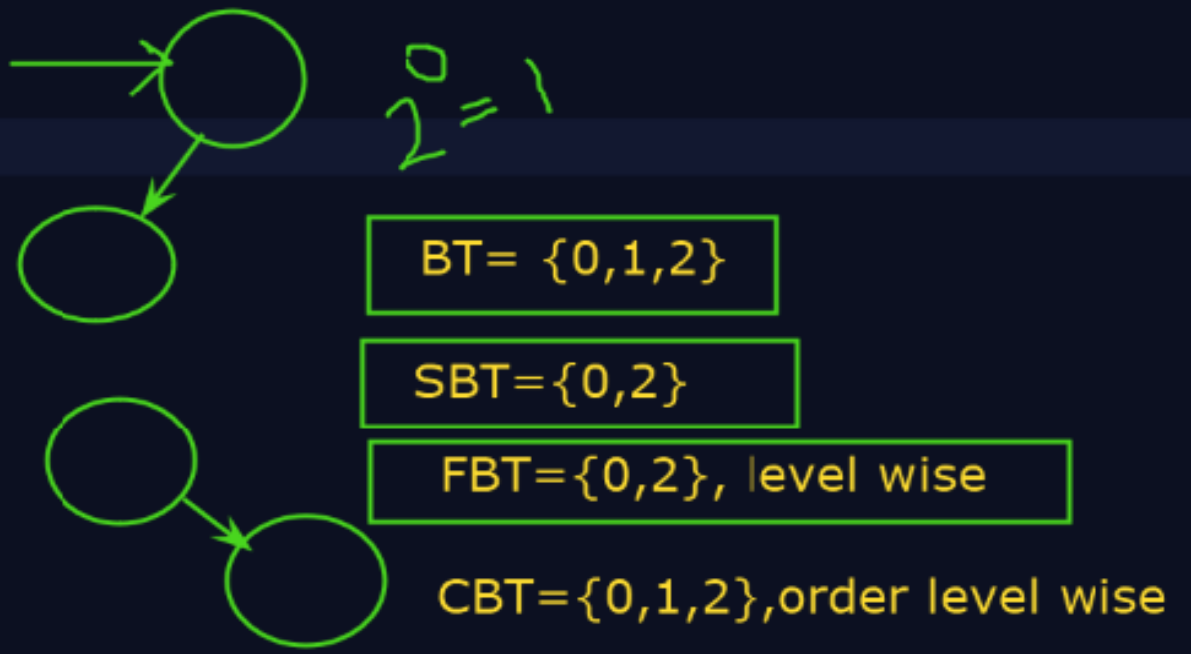
Meeting ID: 832 1579 8576

Passcode: 806920

Topics:

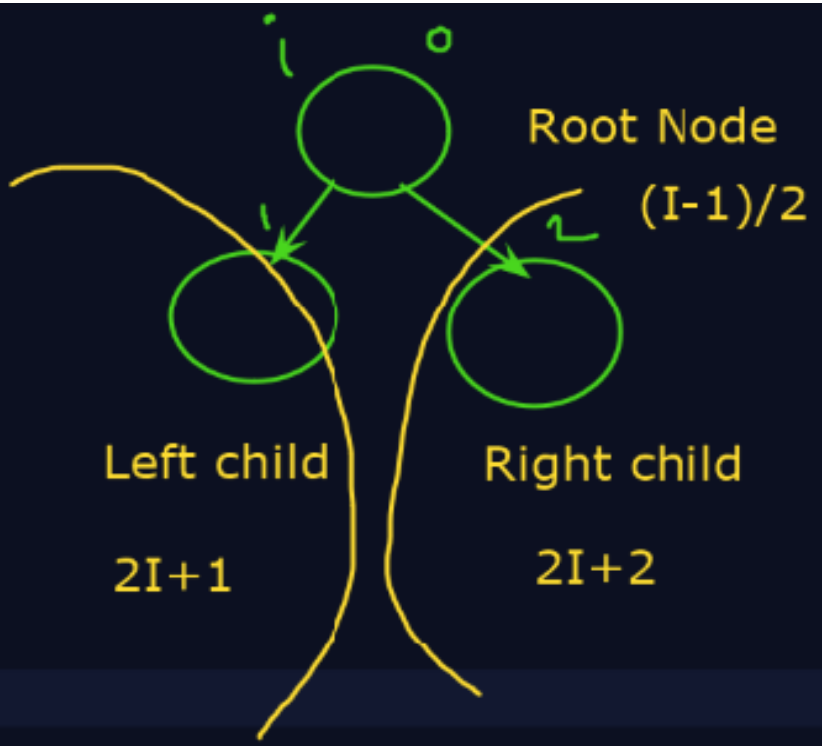
- -Tree
- -Binary Tree
- -Tree Traversal
- -Binary Search Tree





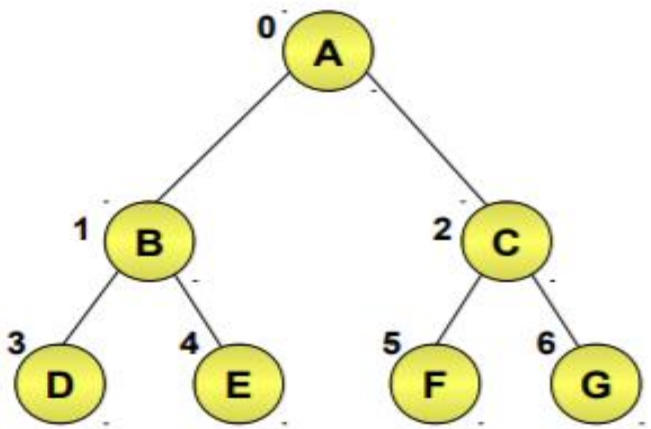
length : 28,012 lines : 1,151 Ln : 1,150 Col : 24 Pos : 28,010 Windows (CR

```
----------------------------------------
Day 8 : Algorithms and Data Structures
----------------------------------------
Topic: Introduction to ADS
Date: 02/10/2024
Meeting ID: 832 1579 8576
Passcode: 806920
----------------------------------------
Topics:
     -Tree
     -Binary Tree
     -Tree Traversal
     -Binary Search Tree
```

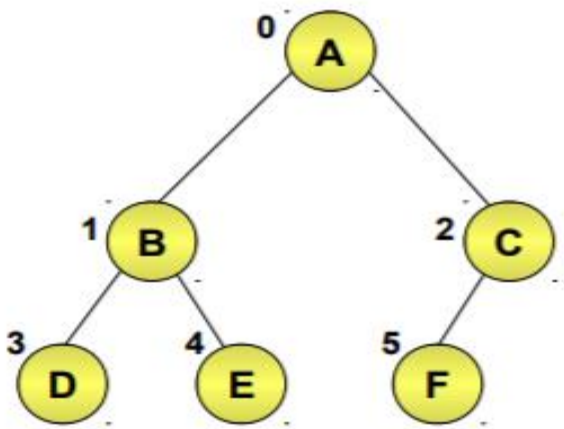


Defining Binary Trees (Contd.)

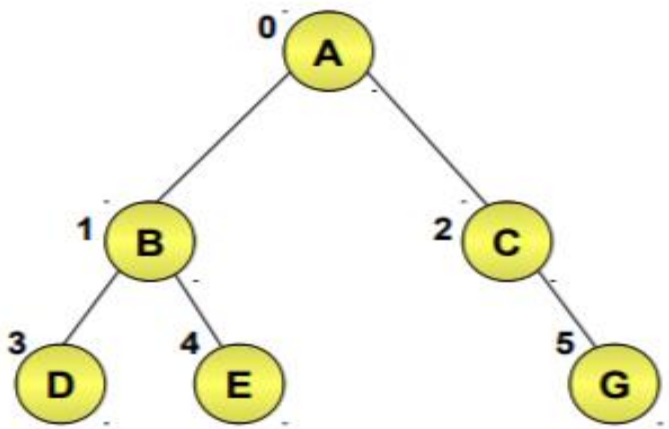
- Complete binary tree:
  - A binary tree with n nodes and depth d whose nodes correspond to the nodes numbered from 0 to n − 1 in the full binary tree of depth k.



Full Binary Tree



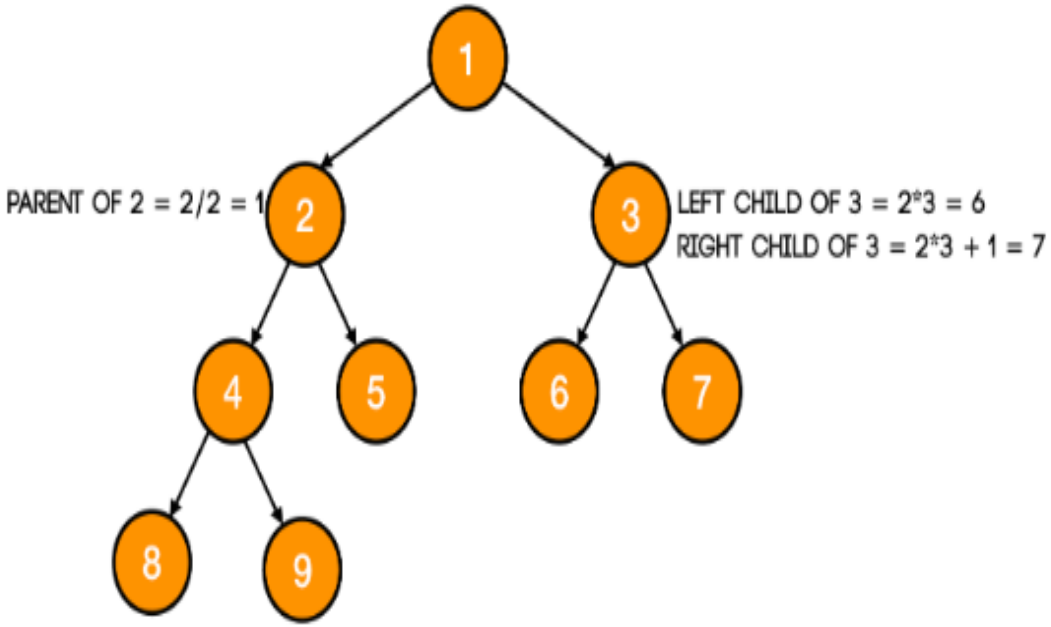
Complete Binary Tree



Incomplete Binary Tree

A complete binary tree also holds some important properties. So, let's look at them.

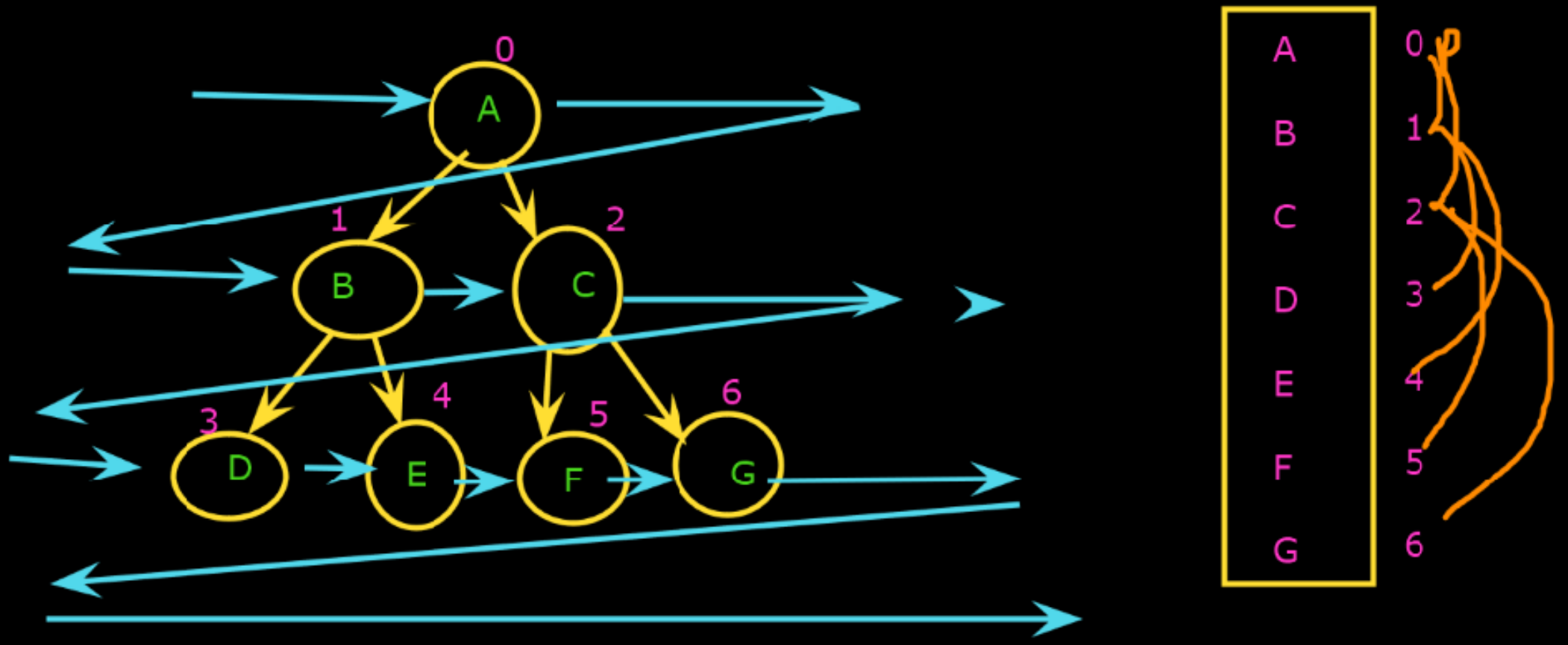
- The **parent of node i** is $\left\lfloor \frac{i}{2} \right\rfloor$. For example, the parent of node 4 is 2 and the parent of node 5 is also 2.
- The **left child of node i** is $2i$.
- The **right child of node i** is $2i + 1$



Binary Tree:

A binary tree is a tree in which every node has at most two children.

0,1,2,2



Array Representation of Binary Tree

-Binary Search Tree

Relationship of Parent and LC and RC:

LC= 2i + 1

RC =2i + 2

Parent = (i-1)/2

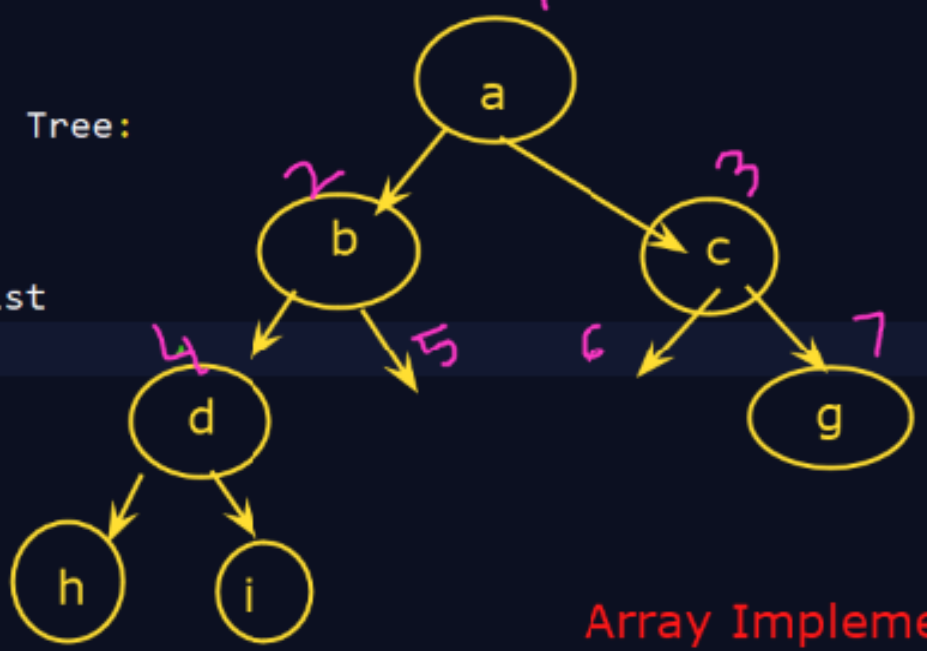
Representation of Tree:

1.Using Array

2.Using Linked list

Root= (1-1)/2=0

1/2



Root Node

(I-1)/2

Left child

2I+1

2i

Right child

2I+2

2i+1

I=6

Array Implementation of Tree

Green =index=0

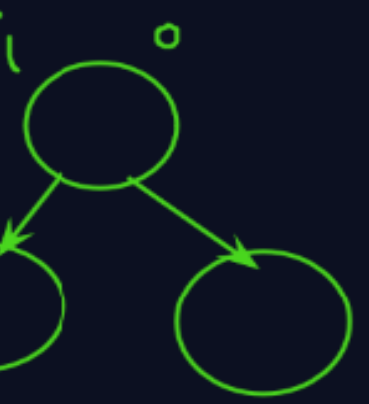
Pink =index=1

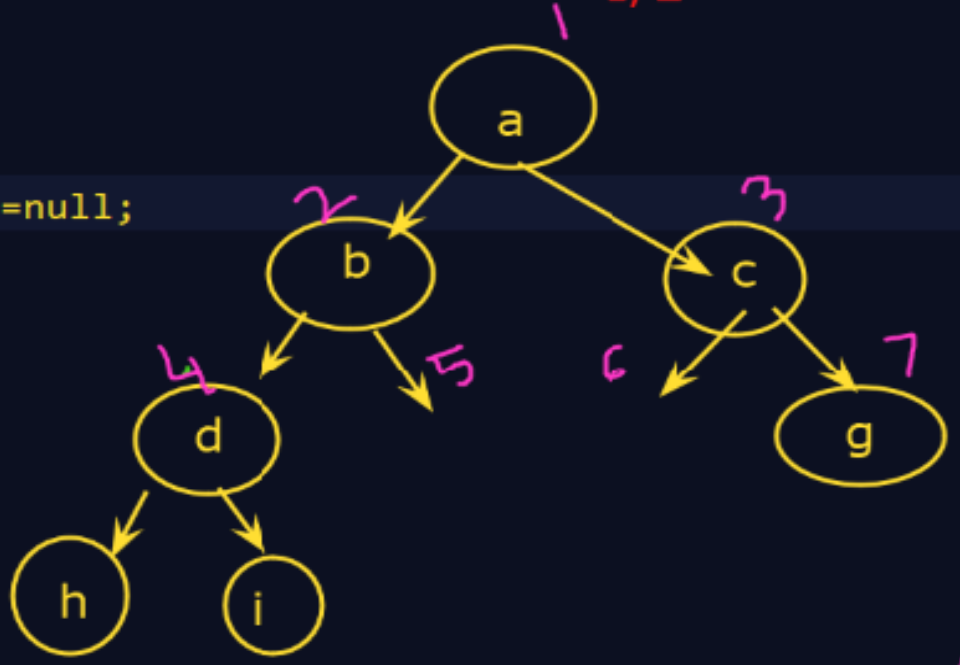
RLC

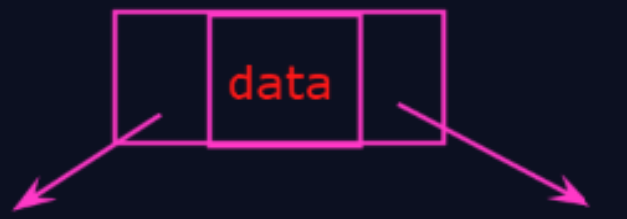
```
class Node{
int data;
Node left;
Node right;

Node(int d)
{
    data = d;
    left=regiht =null;
}
}
```

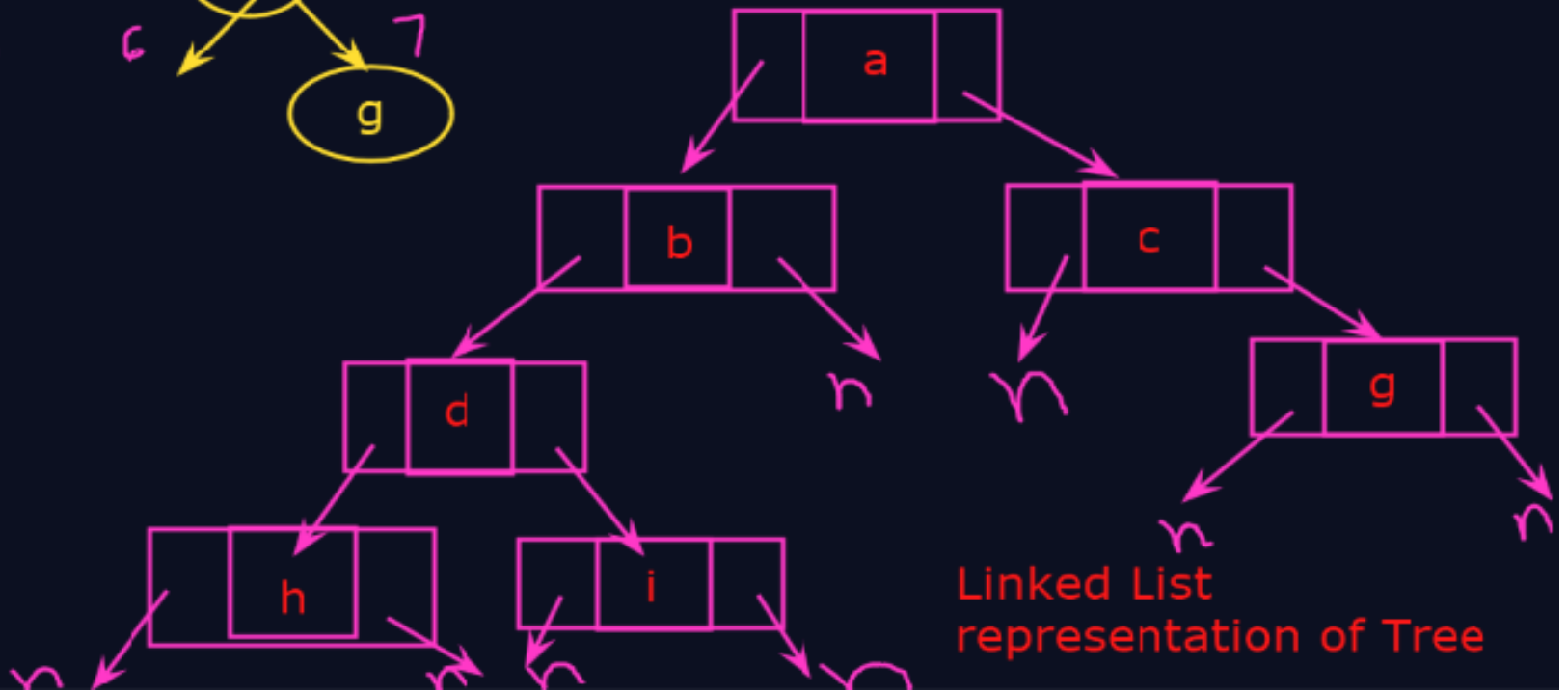
Root= (1-1)/2=0

1/2

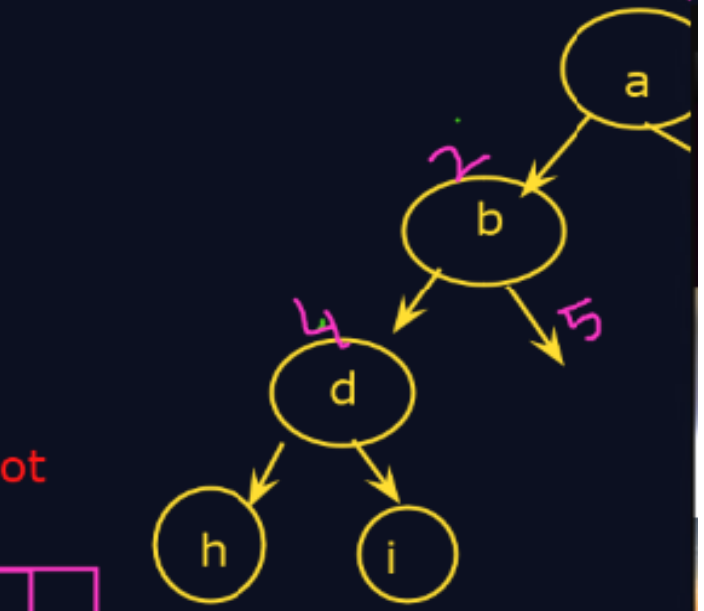


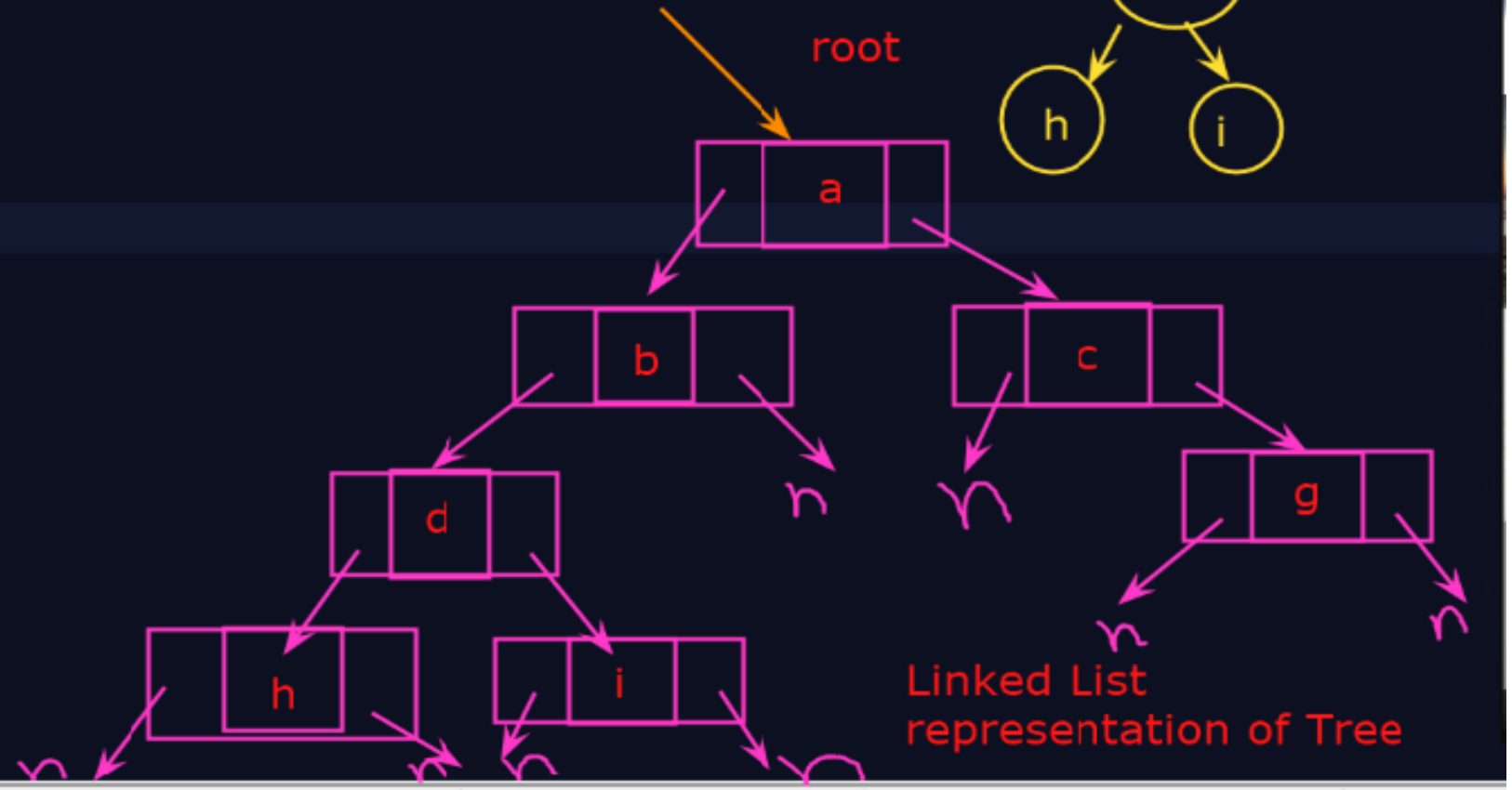






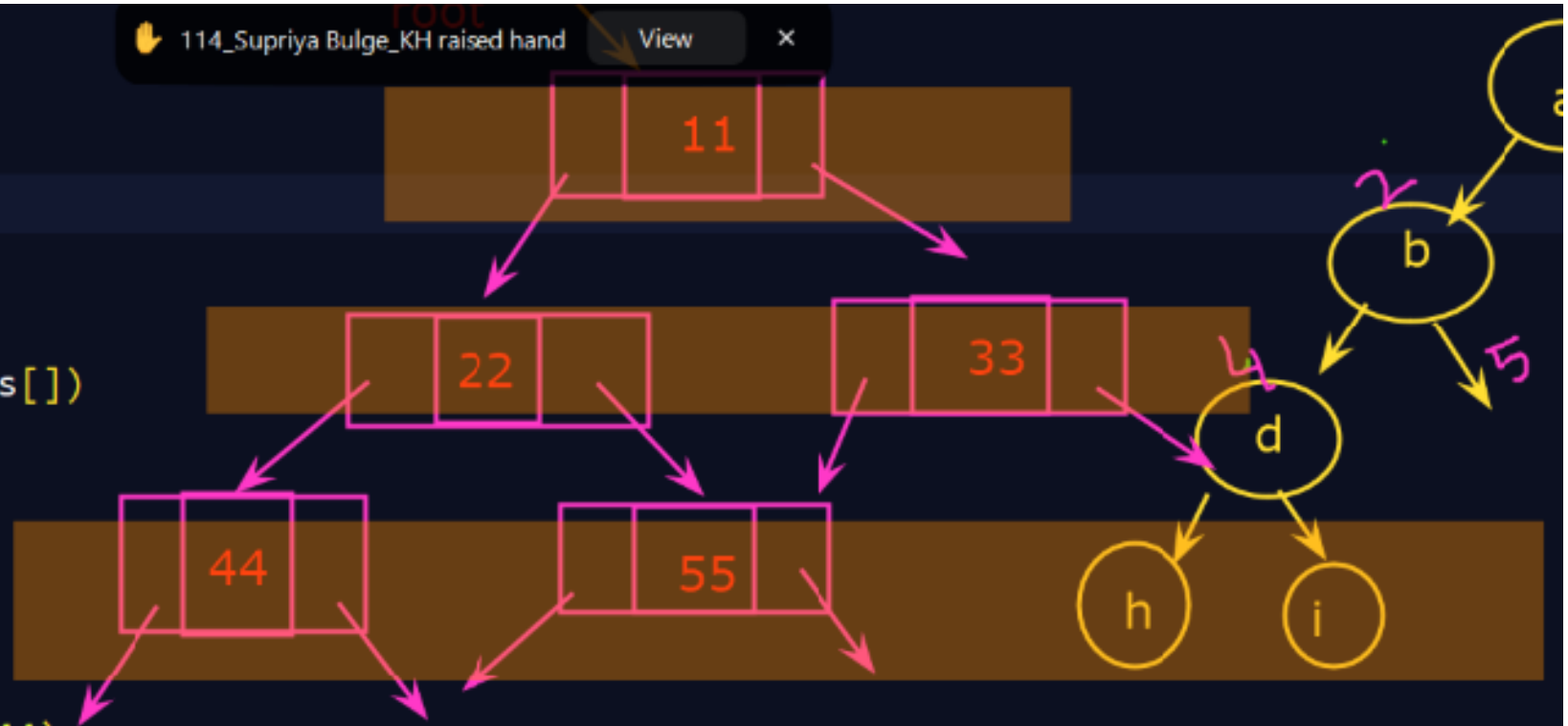
```
class BT{
    Node root;//starting point of tree

    static class Node{
    int data;
    Node left,right;

    Node(int d)
    {
        data = d;
        left=right=null;
    }
    }

    BT()
    {
        root = null;
    }

    BT(int d)
    {
        root = new Node(d);
    }
```





length : 246 lines : 24 Ln : 13 Col : 6 Pos : 164 Windows (CR LF)

```
BT(int d)
{
    root = new Node(d);
}

public static void main(String args[])
{
    BT t1 = new BT();
    t1.root = new Node(11);
    t1.root.left = new Node(22);
    t1.root.right = new Node(33);
    t1.root.left.left = new Node(44);
    t1.root.left.right = new Node(55);
}
```

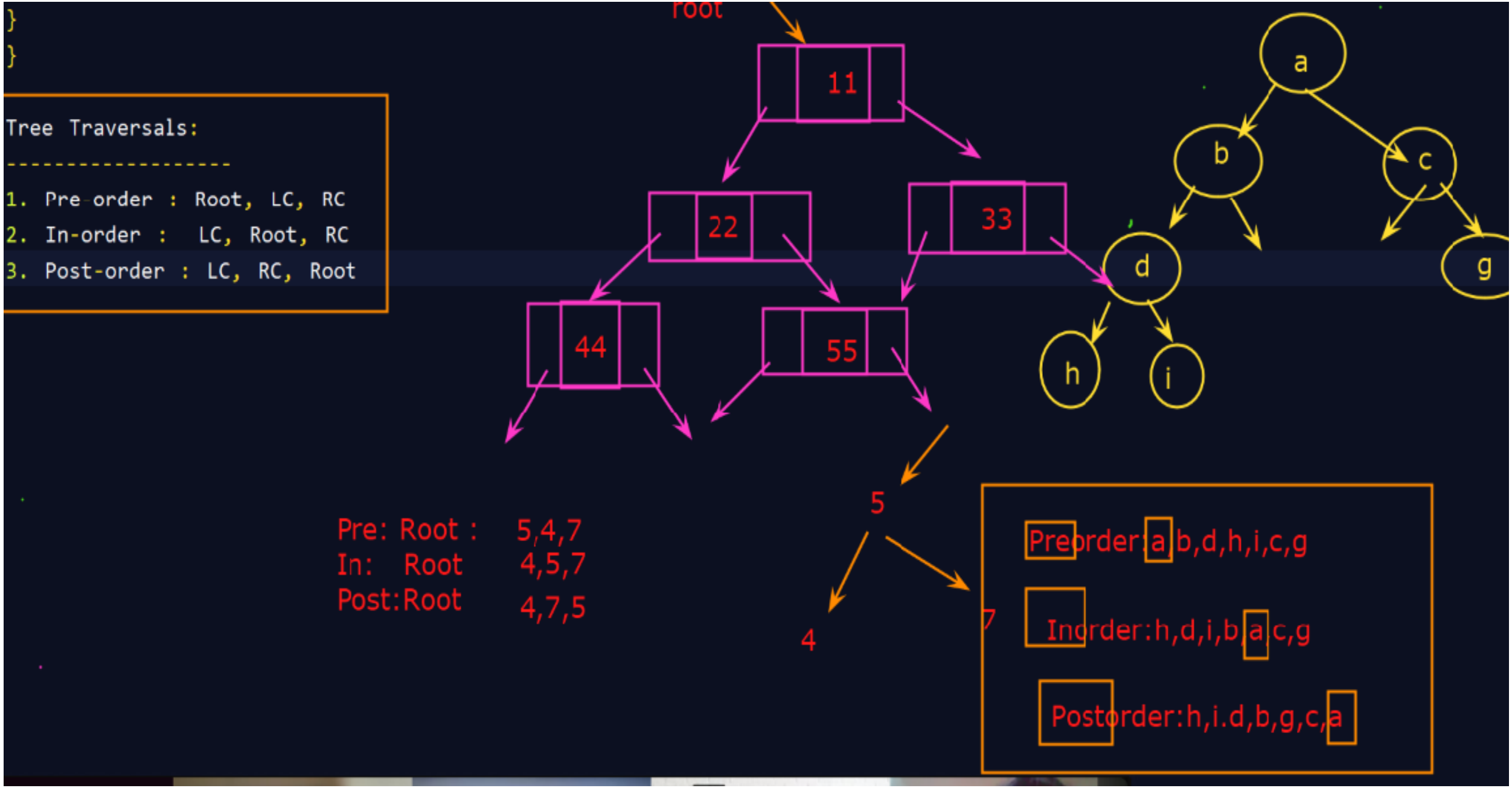



OPERATIONS ON TREES

Traversing a Binary Tree

1)TRAVERSING

- You can implement various operations on a binary tree.
- A common operation on a binary tree is traversal.
- Traversal refers to the process of visiting all the nodes of a binary tree once.
- There are three ways for traversing a binary tree:
  - Inorder traversal
  - Preorder traversal
  - Postorder traversal

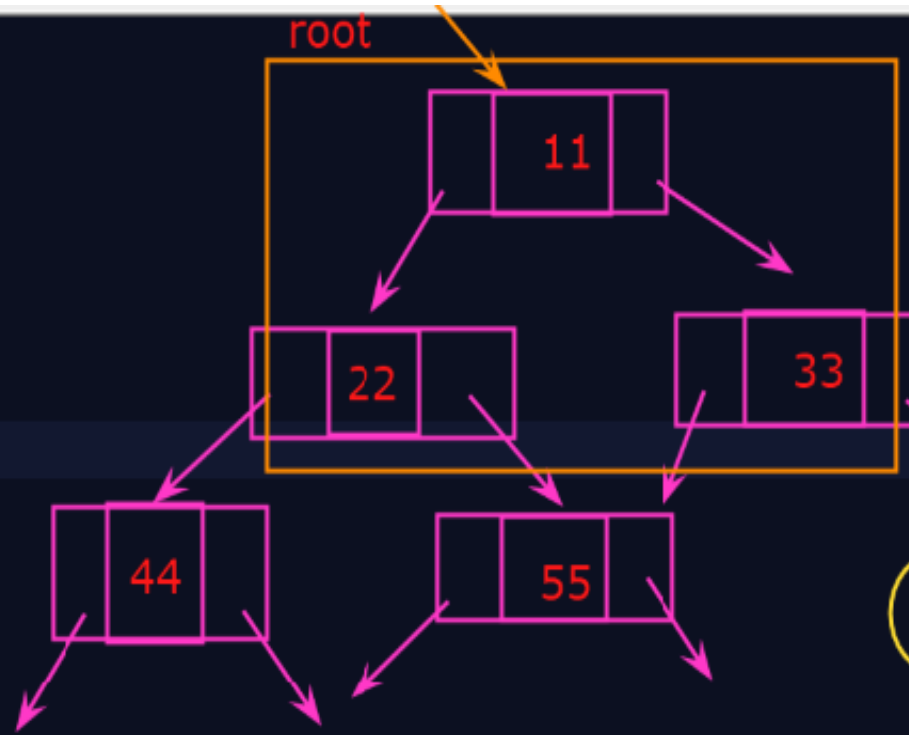

}
}
Tree Traversals:
1. Pre-order : Root, LC, RC
2. In-order : LC, Root, RC
3. Post-order : LC, RC, Root
root
11
22
33
44
55
5
4
7
Pre: Root : 5,4,7
In: Root 4,5,7
Post:Root 4,7,5
a
b
c
d
g
h
i
Preorder:a,b,d,h,i,c,g
Inorder:h,d,i,b,a,c,g
Postorder:h,i.d,b,g,c,a

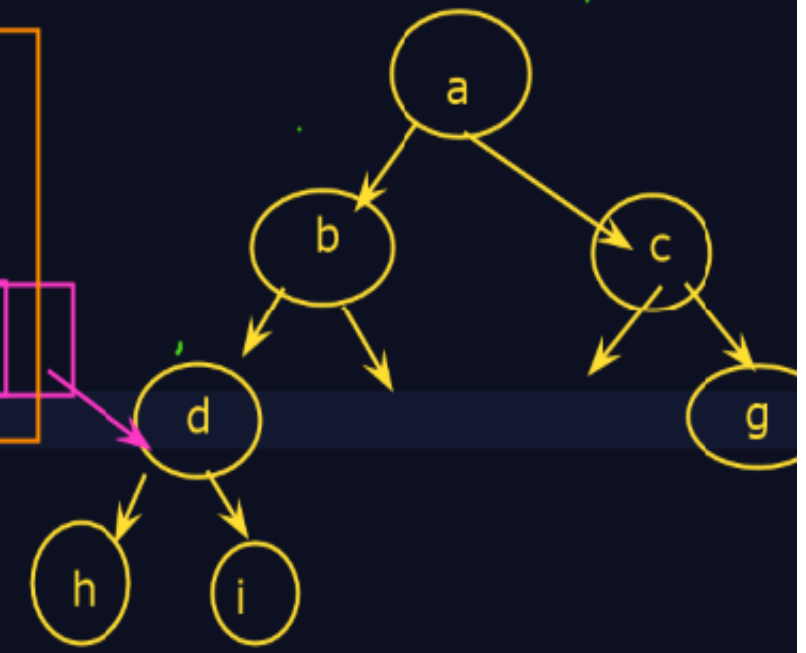
```
}
}
```

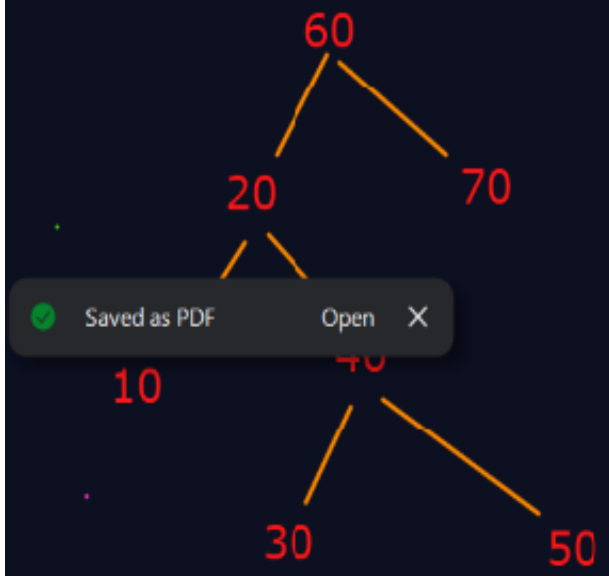
Tree Traversals:

---

1. Pre order : Root, LC, RC
2. In-order : LC, Root, RC
3. Post-order : LC, RC, Root







Saved as PDF Open

Inorder: 44,22,55,11,33

Preorder:11,22,44,55,33

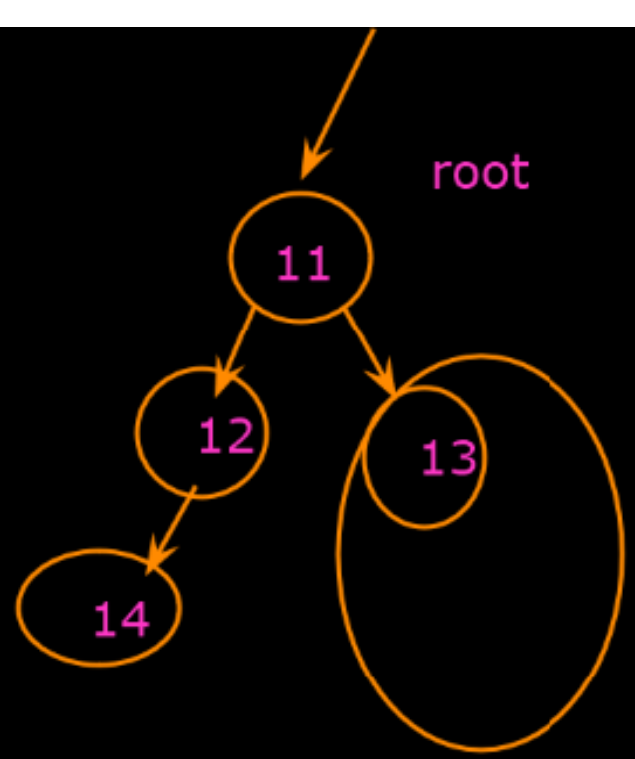
Postorder: 44,55,22,33,11

Preorder:a,b,d,h,i,c,g

Inorder:h,d,i,b,a,c,g

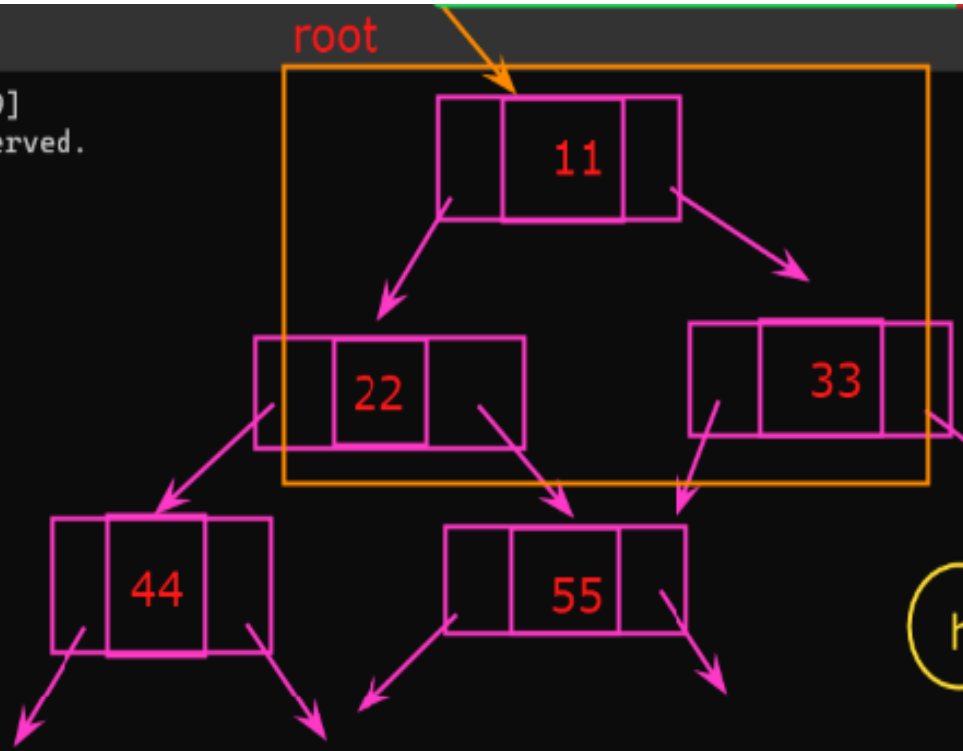
Postorder:h,i.d,b,g,c,a

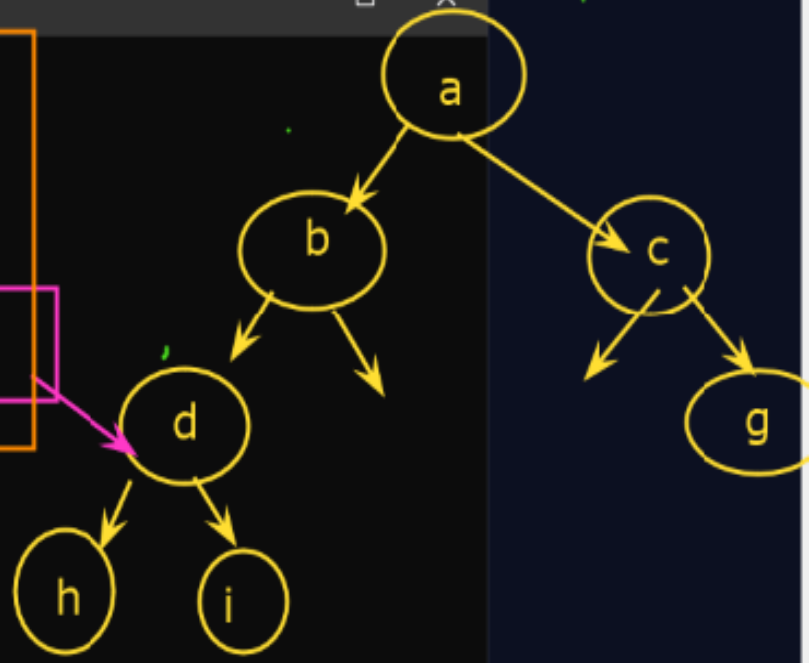
```
    System.out.println(root.data+ " ");
    printInorder(root.right);

    }

    void printPreorder(Node root)
{
    if(root == null)
        return;

    System.out.println(root.data+ " ")
    printPreorder(root.left);
    printPreorder(root.right;

}
```




```
void printPostorder(Node root)
{
    if(root == null)
        return;
```

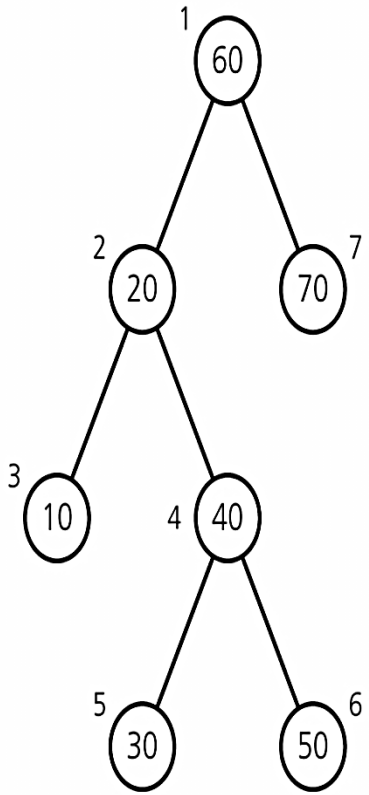
```
C:\Windows\system32\c
Microsoft Windows [Version 10.0.22631.4169]
(c) Microsoft Corporation. All rights reserved.

D:\Test>javac BT.java

D:\Test>java BT
PreOrder:
11 22 44 55 33
InOrder:
44 22 55 11 33
PostOrder:
44 55 22 33 11

D:\Test>
```



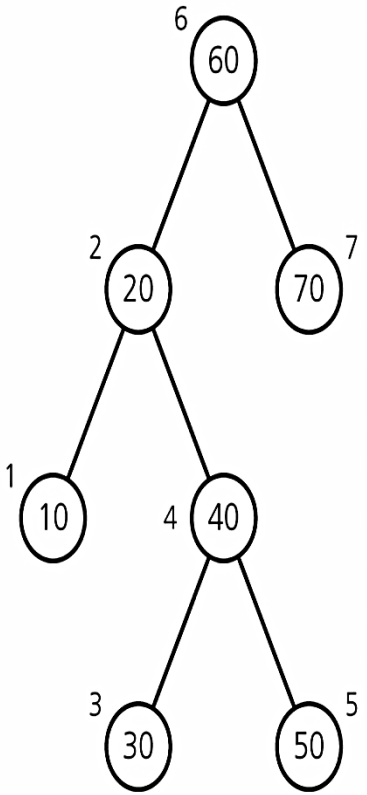



```
}
void
{
}
void
{
}
void
{
    if(node == null)
```

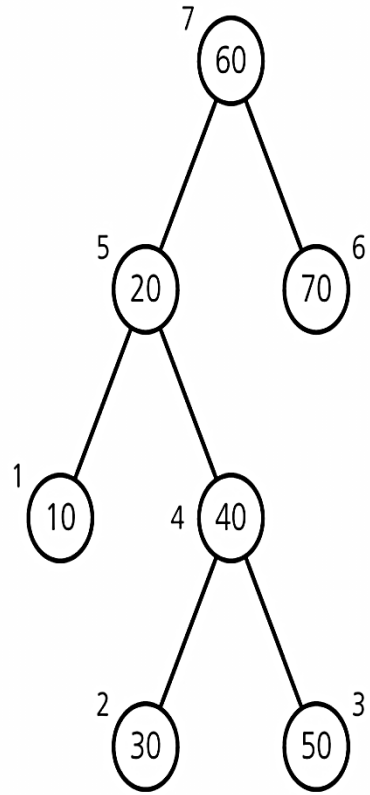
Binary Tree Traversals



(a) Preorder: 60, 20, 10, 40, 30, 50, 70



(b) Inorder: 10, 20, 30, 40, 50, 60, 70



(c) Postorder: 10, 30, 50, 40, 20, 70, 60

(Numbers beside nodes indicate traversal order.)

InOrder(root) visits nodes in the following order:

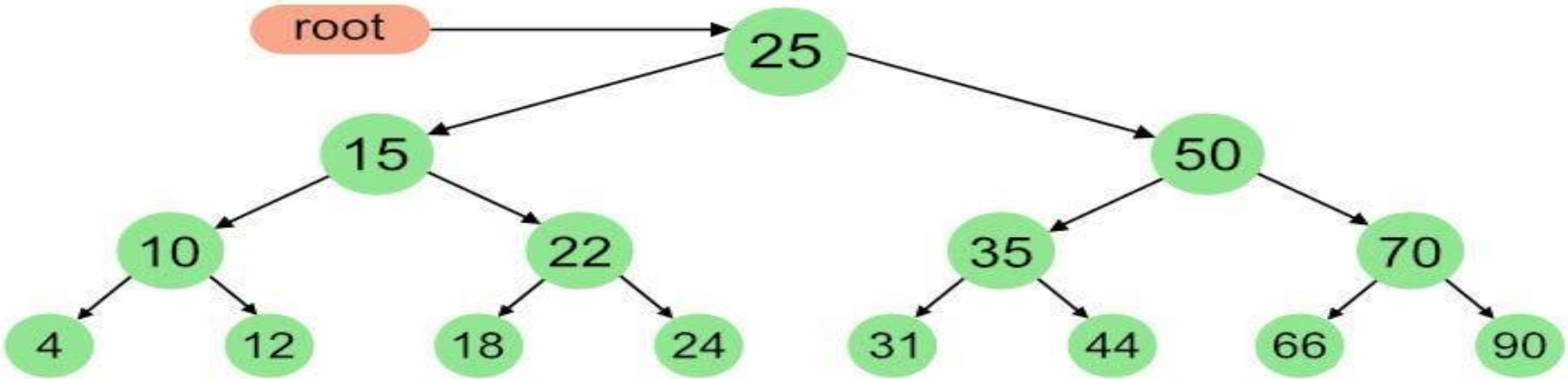
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

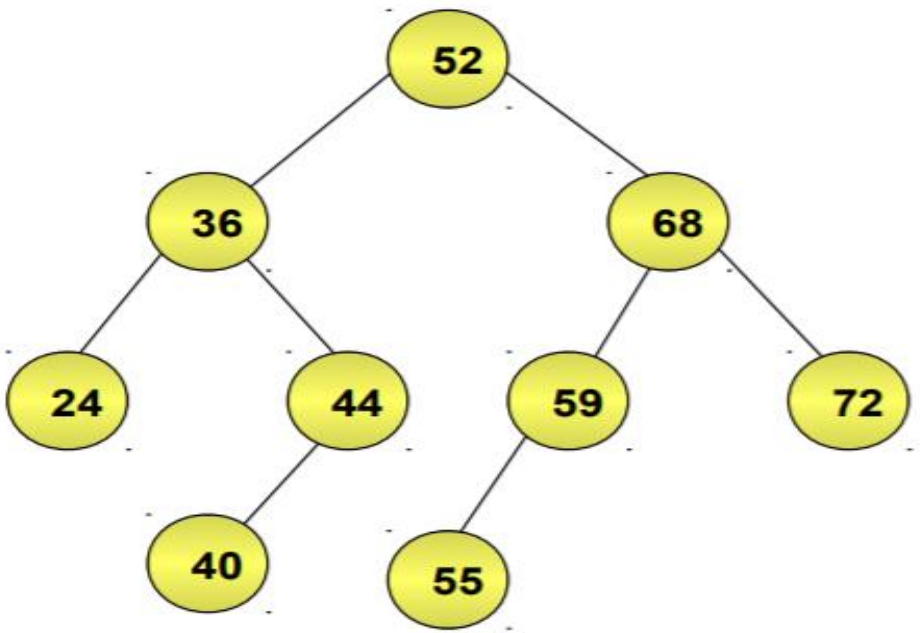
A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Binary Search Tree

- Binary search tree is a binary tree in which every node satisfies the following conditions:
  - All values in the left subtree of a node are less than the value of the node.
  - All values in the right subtree of a node are greater than the value of the node.
- The following is an example of a binary search tree.



Operations on a Binary Search Tree

- The following operations are performed on a binary earch tree...
  - Search
  - Insertion
  - Deletion
  - Traversal

Insertion of a key in a BST

```
Algorithm:- InsertBST (info, left, right, root, key, LOC)
{
   key is the value to be inserted.
   1. call SearchBST ( info, left, right, root, key, LOC , PAR )    // Find the parent of the new node
   2. If ( LOC != NULL)
   2.1 Print " Node alredy exist"
   2.2 Exit
   3. create a node [ new1 =  ( struct node*) malloc ( sizeof( struct node) ) ]
   4. new1 -> info = key
   5. new1 -> left = NULL , new1 -> right = NULL
   6. If ( PAR = NULL ) Then
   6.1 root = new1
   6.2 exit
     elseif ( new1 -> info < PAR -> info)
   6.1 PAR -> left = new1
   6.2 exit
     else
   6.1 PAR -> right = new1
   6.2 exit
 }
```

Deleting Nodes from a Binary Search Tree

- Write an algorithm to locate the position of the node to deleted from a binary search tree.

- Delete operation in a binary search tree refers to the process of deleting the specified node from the tree.
- Before implementing a delete operation, you first need to locate the position of the node to be deleted and its parent.
- To locate the position of the node to be deleted and its parent, you need to implement a search operation.

Deleting Nodes from a Binary Search Tree (Contd.)

- Once the nodes are located, there can be three cases:
  - **Case I**: Node to be deleted is the leaf node
  - **Case II**: Node to be deleted has one child (left or right)
  - **Case III**: Node to be deleted has two children

Deletion of a key from a BST

```
Algorithm:- Delete1BST (info, left, right, root, LOC, PAR)
            // When leaf node has no child or only one child
{
   1.  if ( ( LOC -> left = NULL)  and  ( LOC -> right = NULL))
          1.1 Child = NULL
       elseIf ( LOC -> left != NULL)
          1.1 Child = LOC -> left
       else
          1.1 Child = LOC -> right
   2. if ( PAR != NULL)
          2.1   if ( LOC = PAR -> left )
                     2.1.1 PAR -> left = Child
          2.1   else
                     2.1.1 PAR -> right = Child
       else
          2.1 root = Child
 }
```

Deletion of a key from a BST

```
Algorithm:- Delete2BST (info, left, right, root, LOC, PAR)
                // When leaf node has both child
{
   1. ptr1 = LOC
   2. ptr2 = LOC -> right
   3. While ( ptr2 -> left != NULL )
                3.1 ptr1 = ptr2
                3.2 ptr2 = ptr2 -> left
   4. call Delete1BST (info, left, right, root, ptr2, ptr1)
   5. If ( PAR != NULL)
                5.1   If LOC = PAR -> left
                              5.1.1 PAR -> left = ptr2
                5.1 else
                                        5.1.1 PAR -> right = ptr2
        else
                5.1 root = ptr2
   6. ptr2 -> left = LOC -> left
   7. ptr2 -> right = LOC -> right
}
```

Deletion of a key from a BST

Algorithm:- DeleteBST (info, left, right, root, key)

```
{
key is the value to be deleted.
    1. call SearchBST ( info, left, right, root, key, LOC, PAR )
            // To find the location LOC and parent  PAR of the
                                node to be deleted.
    2. If ( LOC = NULL ) Then
            2.1 Print " Node does not exist"
            2.2 exit
    3.  if ( ( LOC -> left != NULL)  and  ( LOC -> right != NULL))
                                        // when the node to be deleted has both child
            3.1 call Delete2BST (info, left, right, root, LOC, PAR)
       else
            3.1 call Delete1BST (info, left, right, root, LOC, PAR)
     }
```

```
private boolean search(BSTNode r, int val)
    {
        boolean found = false;
        while ((r != null) && !found)
        {
            int rval = r.getData();
            if (val < rval)
                r = r.getLeft();
            else if (val > rval)
                r = r.getRight();
            else
            {
                found = true;
                break;
            }
            found = search(r, val);
        }
        return found;
    }
```

Thanks