Tree

1-Trees are a very common data structure in computer science.

2-A tree is a nonlinear data structure that is used to store data in a hierarchical manner.

3-Binary trees are often chosen over more fundamental structures, such as arrays and linked lists, because you can search a binary tree quickly (as opposed to a linked list) and you can quickly insert data and delete data from a binary tree (as opposed to an array).

4- A *tree* is a set of *nodes* storing elements in a parent-child relationship.

Figure below displays tree that defines a few terms we need when discussing trees.

1-The top node of a tree is called the *root* node.

2-If a node is connected to other nodes below it, the top node is called the parent, and the nodes below it are called the parent's children.

4-A node can have zero, one, or more nodes connected to it.

5-A node without any child node is called a *leaf* (terminal) .

4-Special types of trees, called *binary* trees, restrict the number of children to no more than two.

6-Binary trees have certain computational properties that make them very efficient for many operations.

7-you can travel from one node to other nodes that are not directly connected.

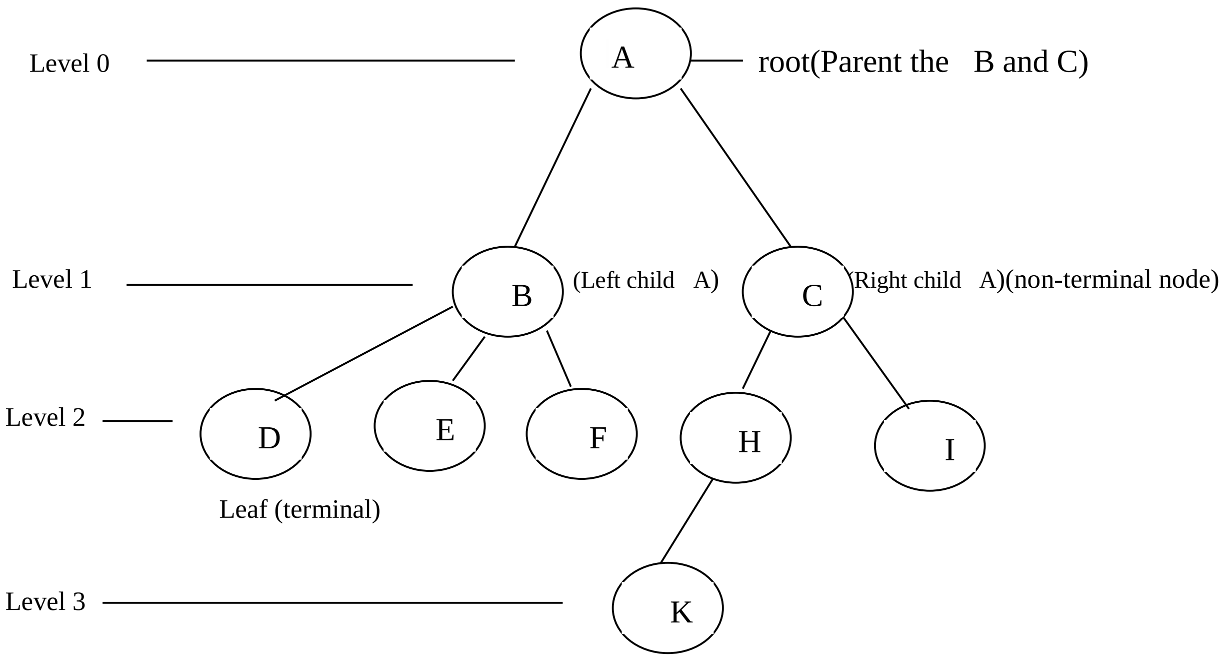
8-The series of edges you follow to get from one node to another is called a *path*

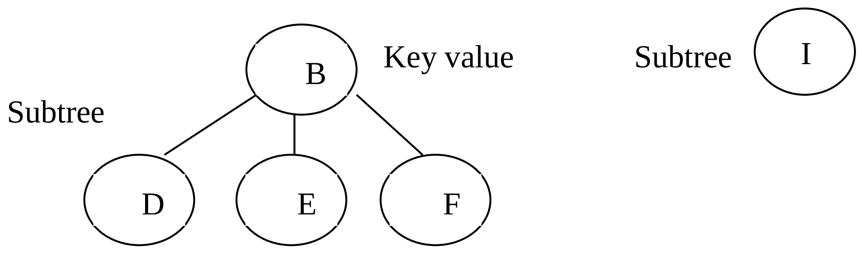
9-Visiting all the nodes in a tree in some particular order is known as a tree *transversal*.

10-A tree can be broken down into *levels*. The root node is at Level 0, its children at Level 1, those node's children are at Level 2, and so on. A node at any level is considered the root of a *subtree*, which consists of that root node's children, its children's children, and so on.

11-We can define the *depth* of a tree as the number of layers in the tree.

12-Finally, each node in a tree has a value. This value is sometimes referred to as the *key* value.





Root : A

Terminal node (Leaf node) : D,E,F,I,K

Non-Terminal node : B ,C ,H

Node Level:

Level 0: A

Level 1 : B ,C

Level 2 : D,E,F,H,I

Level 3 : K

-The degree of a node is the number of sub trees (children) of this node.

The degree of node A is 2; the degree of node B is 3. Degree of node I is 0 (Leaf or terminal node be zero degree).

The degree of the tree : (Maximum degree of the Node) = 3 (B three children)

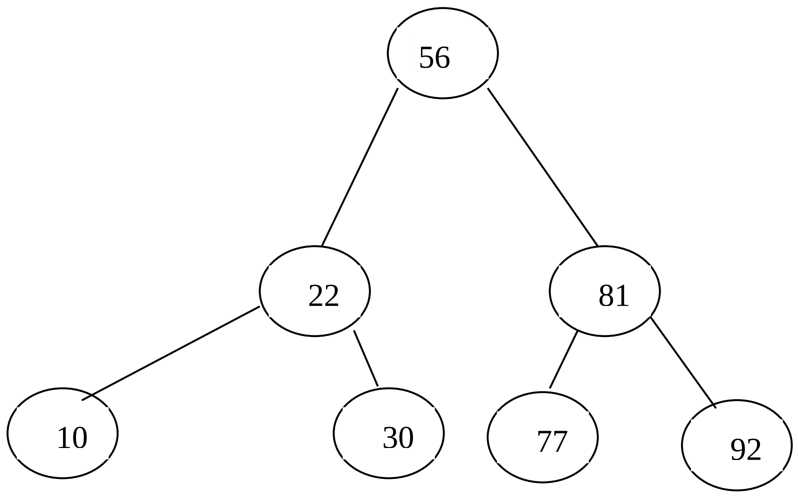
The Depth of the tree : = 3 (Number of Levels)

BINARY TREES

A binary tree is defined as a tree where each node can have no more than two children. By limiting the number of children to 2, we can write efficient programs for inserting data, deleting data, and searching for data in a binary tree.

Before we discuss building a binary tree in C#, we need to add two terms to our tree lexicon. The child nodes of a parent node are referred to as the *left* node and the *right* node. For certain binary tree implementations, certain data values can only be stored in left nodes and other data values must be stored in right nodes. An example binary tree is shown in Figure below.

A binary search tree is a binary tree where data with lesser values are stored in left nodes and values with greater values are stored in right nodes. This property provides for very efficient searches, as we shall soon see.



Binary Tree (Binary Search Tree, Balanced binary tree, Full Binary Tree , Complete Binary Tree)

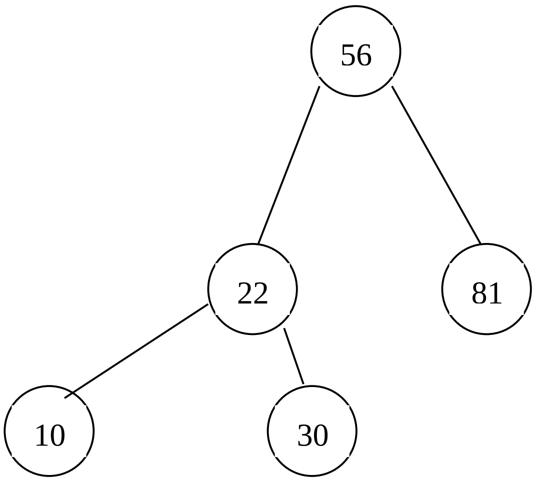
Maximum Number of Nodes in BT

1. The maximum number of nodes on level i of a binary tree is 2^i, i>=0.
2. The maximum number of nodes in a binary tree of depth k is 2^{k+1}-1, k>=0.

Types of Trees

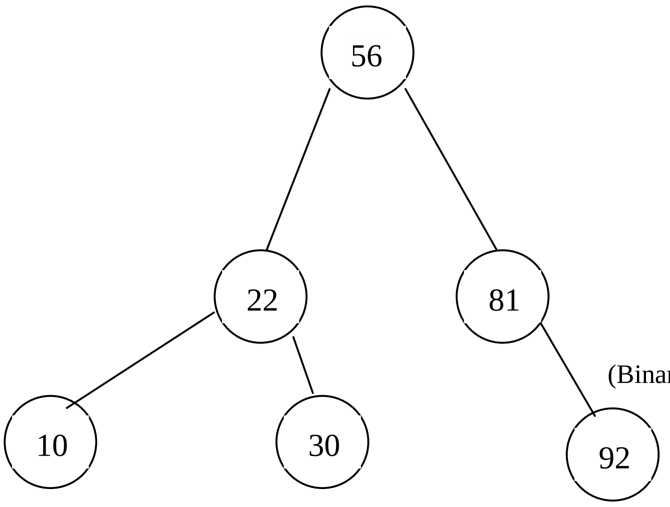
1-Balanced binary tree:

Is the tree that all its leaves(terminals) at one level



non- Balanced binary tree

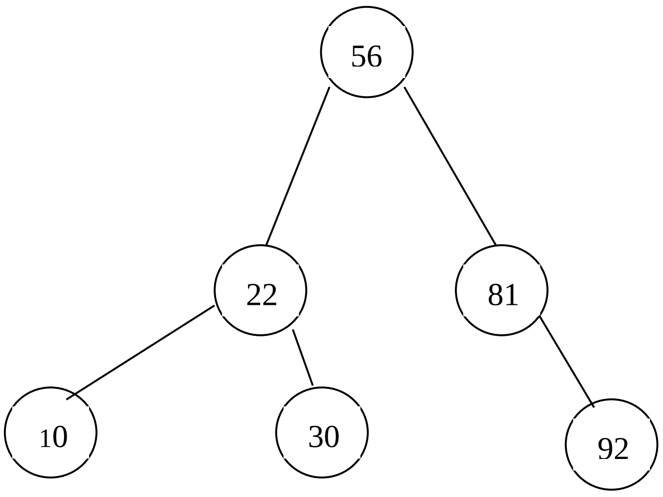
(Binary Search Tree,non- Full Binary Tree, Complete Binary Tree)



Balanced binary tree

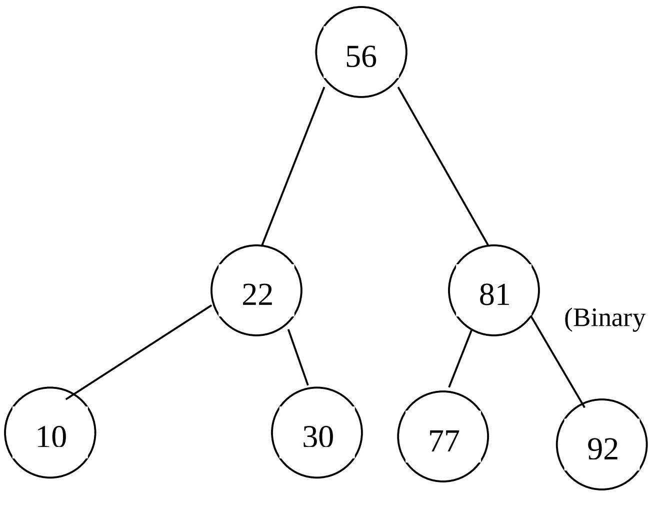
(Binary Search Tree, non- Full Binary Tree , non-Complete Binary Tree)

2- Full Binary Tree : Is the tree that all its leaves at one level, and each node has two children.



non- Full Binary Tree

(Binary Search Tree, Balanced binary tree, non- Complete Binary Tree)

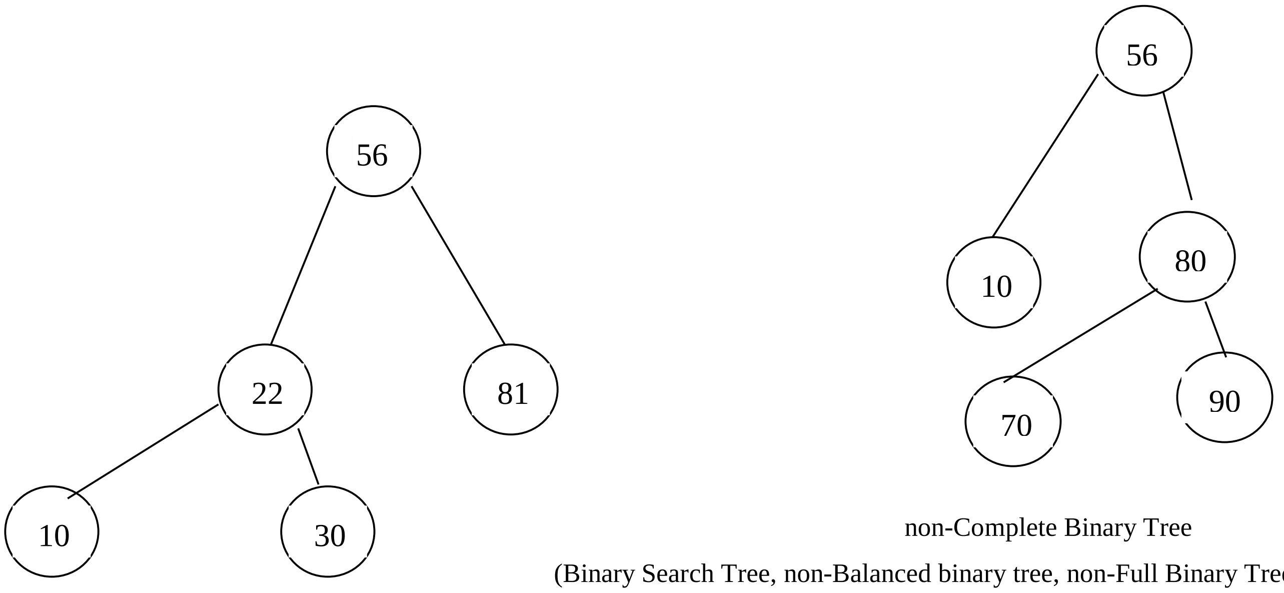


Full Binary Tree

(Binary Search Tree, Balanced binary tree, Complete Binary Tree)

3- Complete Binary Tree

It is a tree that is either full-tree or full of reducing the level before last, and be the last level in the far left-sheets

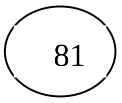


non-Complete Binary Tree

(Binary Search Tree, non-Balanced binary tree, non-Full Binary Tree)

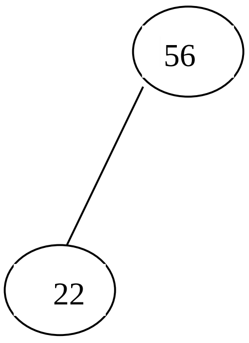
Complete Binary Tree

(Binary Search Tree, non- Balanced binary tree, non-Full Binary Tree)



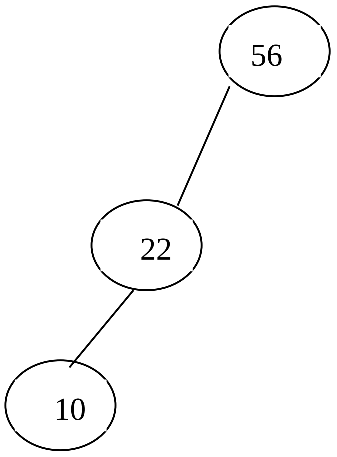
Complete Binary Tree

Full Binary Tree



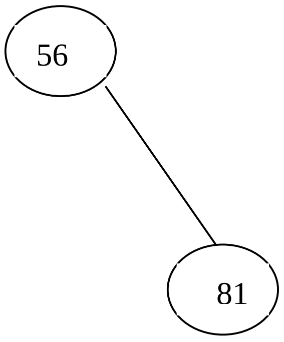
Complete Binary Tree

Non-Full Binary Tree



Non-Complete Binary Tree

Non-Full Binary Tree



Non-complete, Non Full

Binary Tree

Example :

You have the figures below. Answer the following questions:

1-What is the type of tree?(Binary Search Tree, Balanced binary tree, Full Binary Tree, complete binary tree)

2-What is the depth of the tree?

3-What is the maximum number of nodes in the tree

4-What is the level of node D in this tree

5-What is the maximum number of nodes in the level 2.

6- What is the terminal nodes in figure a.

7- What is the non-terminal nodes in figure b.

Figure a

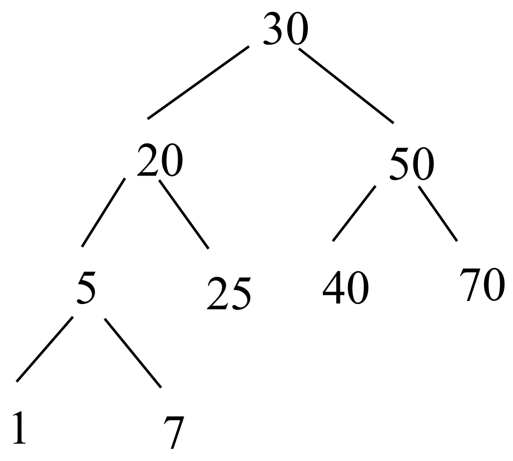
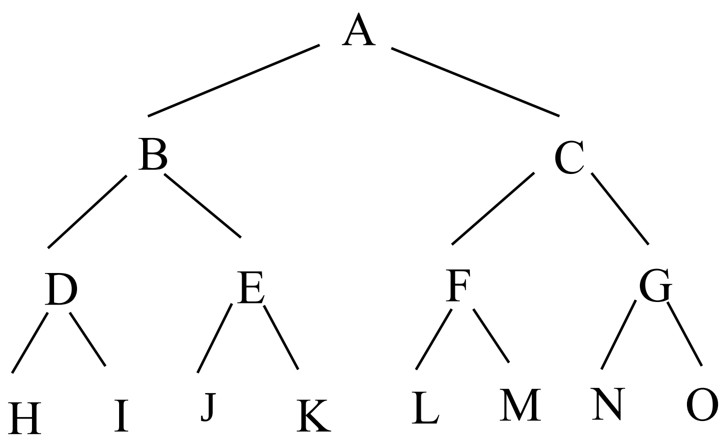


Figure b



Binary Tree Representations

1-Array Representation:

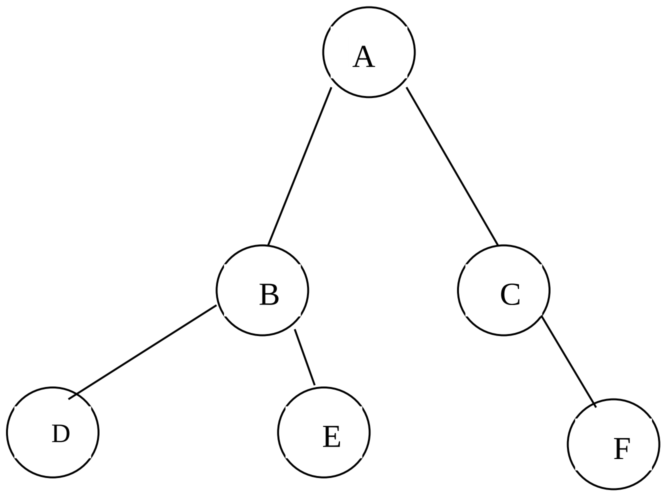
The size of array n : The maximum number of nodes in a binary tree of depth k is

$n = 2^{k+1}-1$ if $k>=0$.

is represented sequentially, then for any node with index i, 0<=i<n, we have:

- parent(i) is at (i/2 -1) if i is even else i/2 .

  If i=0, i is at the root and has no parent.
- left_child(i) is at 2*i+1 if 2*i<n. If 2*i>n, then i has no left child.
- right_child(i) is at 2*i+2 if 2*i +2 <n. If 2*i +2 >=n, then i has no right child.



The size of Array $n = 2^{2+1} -1 = 7$

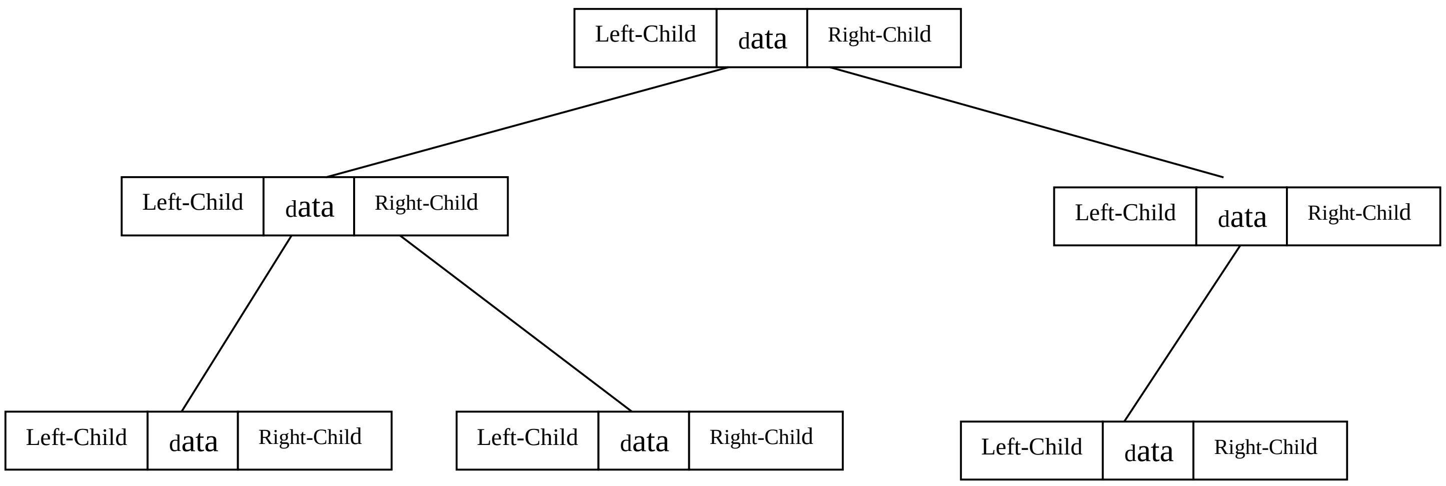
0	1	2	3	4	5	6
A	B	C	D	E	-	F

Example: Draw the following binary tree, :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
60	50	85	30	55	70	90	2	40	-	-	-	-	88	-

What is the type of tree?(Binary Search Tree, Balanced binary tree, Full Binary Tree, complete binary tree)

2- Linked List Representation: In this way linked list used to represent the tree . two pointers represent the node where the one pointer of them refers to the left child and the other refers to the right child.



<u>Building a Binary Search Tree</u>

A binary search tree is made up of nodes, so we need a Node class that is similar to the Node class we used in the linked list implementation.

Class **Node** specification :

Node
+Data:int or object **+ Left:Node** **+Right:Node**

+Node()
+ Node (object)
+ DisplayNode()

We include Public data members for the data stored in the node and for each child node. This particular Node class holds integers, but we could adopt the class easily to hold any type of data, or even declare Data of Object type if we need to.

Next we're ready to build a BinarySearchTree (BST) class.

Class BinarySearchTree (BST) class specification :

BST
-root:Node
+BST() **+ insert(object):void**

1-The class consists of just one data member a Node object that represents the root node of the BST.

2-The default constructor method for the class sets the root node to null, creating an empty node.

3-Insert method to add new nodes to our tree. This method is somewhat complex and will require some explanation.

The first step in the method is to create a Node object and assign the data the Node holds to the iData variable. This value is passed in as the only argument to the method.

The second step to insertion is to see if our BST has a root node. If not, then this is a new BST and the node we are inserting is the root node. If this is the case, then the method is finished. Otherwise, the method moves on to the next step.

If the node being added is not the root node, then we have to prepare to traverse the BST in order to find the proper insertion point. This process is similar to traversing a linked list. We need a Node object that we can assign to the current node as we move from level to level. We also need to position ourselves inside the BST at the root node.

Once we're inside the BST, the next step is to determine where to put the new node. This is performed inside a while loop that we break once we've found the correct position for the new node. **The algorithm for determining the proper position for a node is as follows:**

1. Set the parent node to be the current node, which is the root node.

2. If the data value in the new node is **less than** the data value in the current node, set the current node to be the left child of the current node. If the data value in the new node is **greater than t**he data value in the current node, skip to Step 4.

3. If the value of the left child of the current node is null, insert the new node here and exit the loop. Otherwise, skip to the next iteration of the While loop.

4. Set the current node to the right child node of the current node.

5. If the value of the right child of the current node is null, insert the new node here and exit the loop. Otherwise, skip to the next iteration of the While loop.

Let's look at the code for the Node class and BST class:

```
public class Node
 {
   public int Data;
    public Node Left;
    public Node Right;
    public Node()
                 {
```

```
                          Data=null;
                           Left=null;
                            Right=null
                        }

      public Node(int item)
                       {
                          Data=item;
                           Left=null;
                            Right=null
                        }
                 public void DisplayNode()
                   {
                textBox1.Text+= Data.ToString()+" ; ";          // print data
                 }
public class BST
{
    public Node root;
     public BST ()
       {
           root = null;
         }
      public void Insert(int i)
      {
           Node newNode = new Node();
            newNode.Data = i;
            if (root == null)
                root = newNode;
           else
            {
                Node current = root;
                 Node parent;
               while (true)
                {
                     parent = current;
```

```
                    if (i < current.Data)
                      {
                          current = current.Left;
                          if (current == null)
                            {
                               parent.Left = newNode;
                               break;
                              }
                          else
                            {
                                current = current.Right;
                               if (current == null)
                                   {
                                     parent.Right = newNode;
                                    break;
                                   }
                             }
               }
        }
 }
```

Example1: create binary search tree from the following numbers:

65 87 76 63 51 92 54 57 62 61 60

Example: create binary search tree from the following numbers:

90 80 70 60 50 30

Traversing a Binary Search Tree

Binary Tree Traversals

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal

  LVR, LRV, VLR, VRL, RVL, RLV
- Adopt convention that we traverse left before right, only 3 traversals remain

LVR,	LRV,	VLR
inorder,	postorder,	preorder

Here's the code for a **inorder traversal method:**

```
public void InOrder(Node theRoot)
 {
       if (!(theRoot == null))
          {
             InOrder(theRoot.Left);
             theRoot.DisplayNode();
              InOrder(theRoot.Right);
           }
}
```

To demonstrate how this method works, let's examine a program that inserts a series of numbers into a BST. Then we'll call the inOrder method to display the numbers we've placed in the BST. Here's the code:

```
static void Main()
 {
       BST nums = new BST ();
        nums.Insert(23);
        nums.Insert(45);
        nums.Insert(16);
        nums.Insert(37);
       nums.Insert(3);
       nums.Insert(99);
       nums.Insert(22);
      // display Inorder traversal
     nums.inOrder(nums.root);
}
```

Here's the output:

```
Inorder traversal:
3   16   22    23   37    45     99
```

Now let’s examine the code for a **preorder traversal:**

```
public void PreOrder(Node theRoot)
 {
     if (!(theRoot == null))
{
    theRoot.displayNode();
     preOrder(theRoot.Left);
     preOrder(theRoot.Right);
}
}
```

Notice that the only difference between the preOrder method and the inOrder method is where the three lines of code are placed. The call to the displayNode method was sandwiched between the two recursive calls in the inOrder method and it is the first line of the preOrder method.

If we replace the call to inOrder with a call to preOrder in the previous sample program, we get the following output:

```
Preorder traversal:
23    16    3    22     45     37      99
```

Finally, we can write a method for performing **postorder traversals:**

```
public void PostOrder(Node theRoot)
{
      if (!(theRoot == null))
   {
       PostOrder(theRoot.Left);
        PostOrder(theRoot.Right);
       theRoot.DisplayNode();
}
}
```

Again, the difference between this method and the other two traversal methods is where the recursive calls and the call to displayNode are placed.

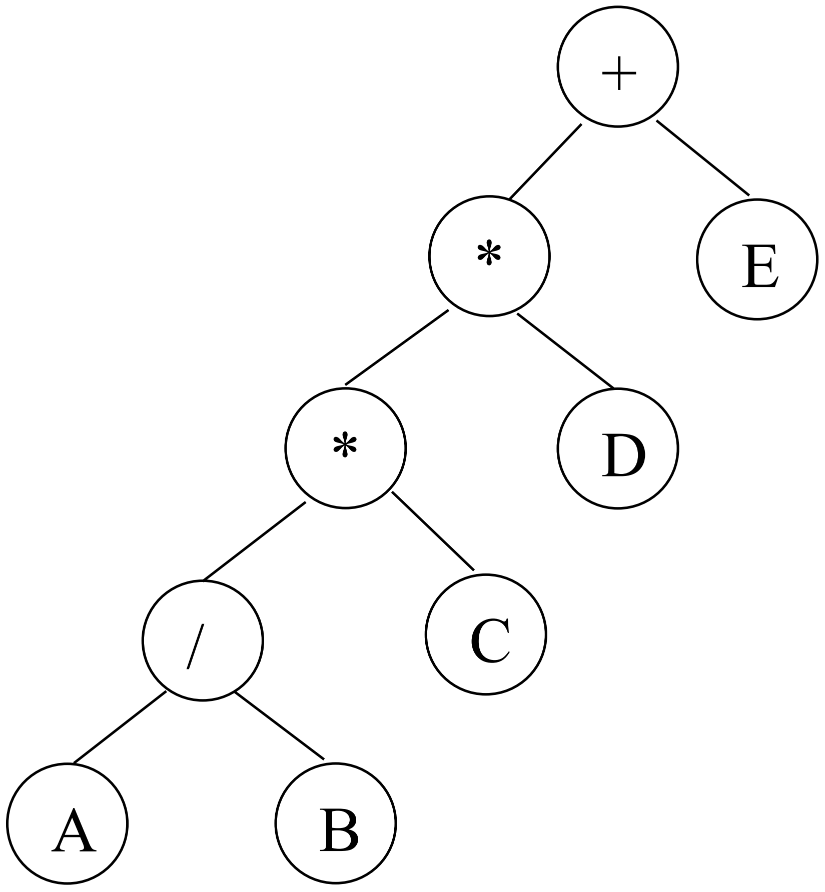
In a postorder traversal, the method first recurses over the left subtrees and then over the right subtrees. Here’s the output from the postOrder method:

```
Postorder traversal:
3    22     16   37    99    45    23
```

We'll look at some practical programming examples using BSTs that use these traversal methods later in this chapter.

Representation of Arithmetic Expressions using Binary Tree

Binary trees is used to represent the mathematical expressions. The operators represent non-terminal node either arithmetic operators represent leaf (terminal) . Note the levels of the tree reflect the priorities in the implementation of the calculations that arithmetic expression.



inorder traversal

A / B * C * D + E infix expression

preorder traversal

+ * * / A B C D E prefix expression

postorder traversal

A B / C * D * E + postfix expression

level order traversal

+ * E * D / C A B

Finding a Node and Minimum/Maximum Values in a Binary Search Tree

Three of the easiest things to do with BSTs are find a particular value, find the minimum value, and find the maximum value. We examine these operations in this section.

The code for finding the minimum and maximum values is almost trivial in both cases, due to the properties of a BST. The smallest value in a BST will always be found at the last left child node of a subtree beginning with the left child of the root node. On the other hand, the largest value in a BST is found at the last right child node of a subtree beginning with the right child of the root node.

We provide the code for finding the minimum value first:

```
public int FindMin()
 {
     Node current = root;
     while (!(current.Left == null))
     current = current.Left;
   return current.Data;
}
```

The method starts by creating a Node object and setting it to the root node of the BST. The method then tests to see if the value in the left child is null. If a non-Nothing node exists in the left child, the program sets the current node to that node. This continues until a node is found whose left child is equal to null. This means there is no smaller value below and the minimum value has been found.

Now here’s the code for finding the maximum value in a BST:

```
public int FindMax()
{
       Node current = root;
      while (!(current.Right == null))
         current = current.Right;
     return current.Data;
}
```

This method looks almost identical to the FindMin() method, except the method moves through the right children of the BST instead of the left children.

The last method we'll look at here is the Find method, which is used to determine if a specified value is stored in the BST. The method first creates a Node object and sets it to the root node of the BST. Next it tests to see if the key (the data we're searching for) is in that node. If it is, the method simply returns the current node and exits. If the data isn't found in the root node, the data we're searching for is compared to the data stored in the current node.
If the key is less than the current data value, the current node is set to the left child. If the key is greater than the current data value, the current node is set to the right child. The last segment of the method will return null as the return value of the method if the current node is null (Nothing), indicating the end of the BST has been reached without finding the key. When the While loop ends, the value stored in current is the value being searched for.

Here's the code for the Find method:

```
public Node Find(int key)
 {
       Node current = root;
        while (current.iData != key)
        {
           if (key < current.iData)
                current = current.Left;
         else
               current = current.Right;
      if (current == null)
        return null;
}
return current;
}
```

Removing a Leaf Node From a BST

The operations we've performed on a BST so far have not been that complicated, at least in comparison with the operation we explore in this section— removal. For some cases, removing a node from a BST is almost trivial; for other cases, it is quite involved and demands that we pay special care to the code we right, otherwise we run the risk of destroying the correct hierarchical order of the BST.
Let's start our examination of removing a node from a BST by discussing the simplest case—removing a leaf. Removing a leaf is the simplest case since there are no child nodes to take into

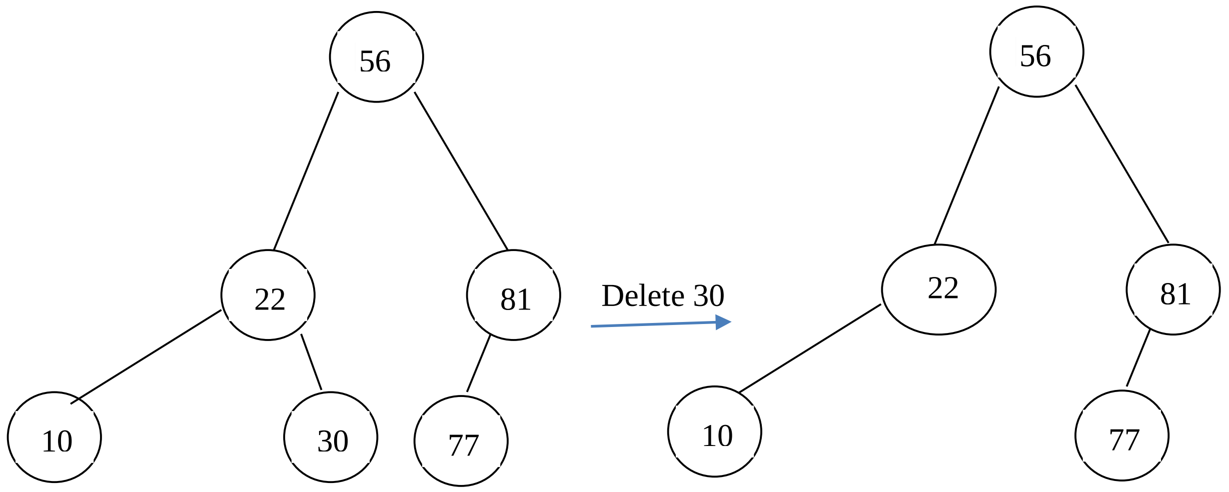
consideration. All we have to do is set each child node of the target node's parent to null. Of course, the node will still be there, but there will not be any references to the node. The code fragment for deleting a leaf node is as follows (this code also includes the beginning of the Delete method, which declares some data members and moves to the node to be deleted):

```
public Node Delete(int key)
{
    Node current = root;
     Node parent = root;
     bool isLeftChild = true;
     while (current.Data != key)
     {
        parent = current;
        if (key < current.Data)
         {
           isLeftChild = true;
         current = current.Right;
          }
     else
       {
         isLeftChild = false;
         current = current.Right;
       }
if (current == null)
return false;
}
if ((current.Left == null) & (current.Right == null))
if (current == root)
root == null;
else if (isLeftChild)
parent.Left = null;
else
parent.Right = null;
}
// the rest of the class goes here
}
```

The while loop takes us to the node we're deleting. The first test is to see if the left child and the right child of that node are null. Then we test to see if this node is the root node. If so, we set it to null, otherwise, we either set the left node of the parent to null (if isLeftChild is true) or we set the right node of the parent to null.

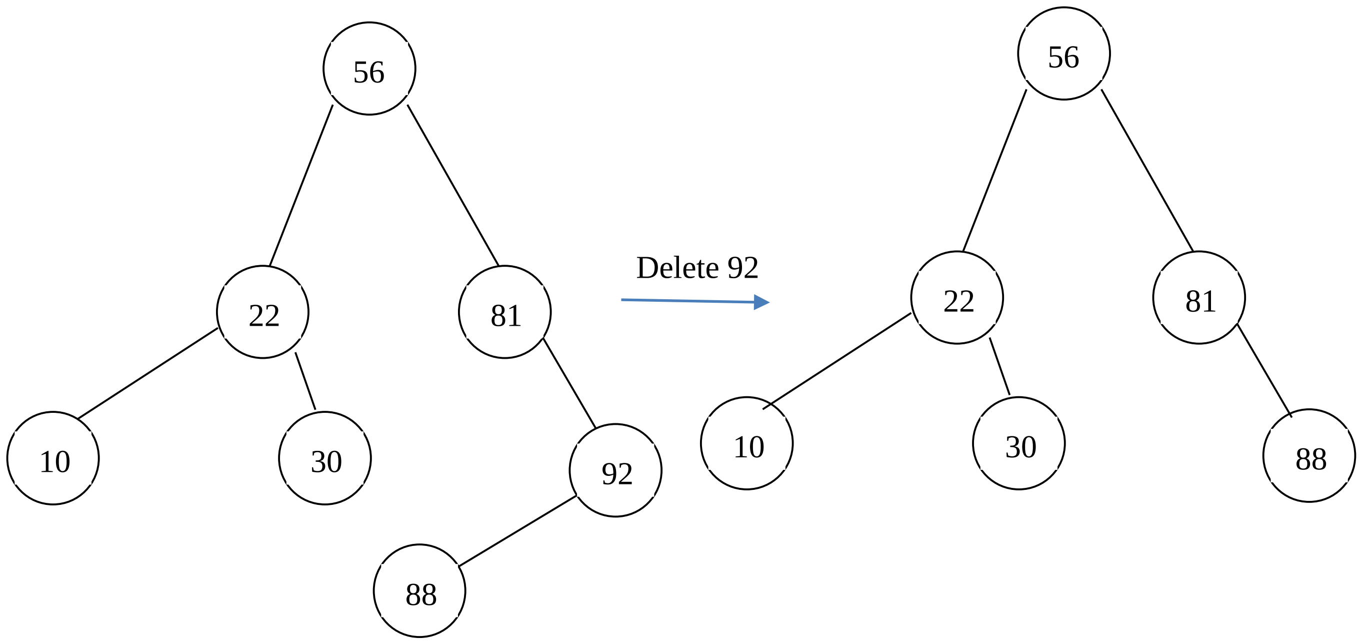
Steps to delete an item from the binary search tree:

Delete leaf node : Take the node and make the value of the node index parent is null .



Deleting a Node With One Child

Make parent node index refers to the child node.



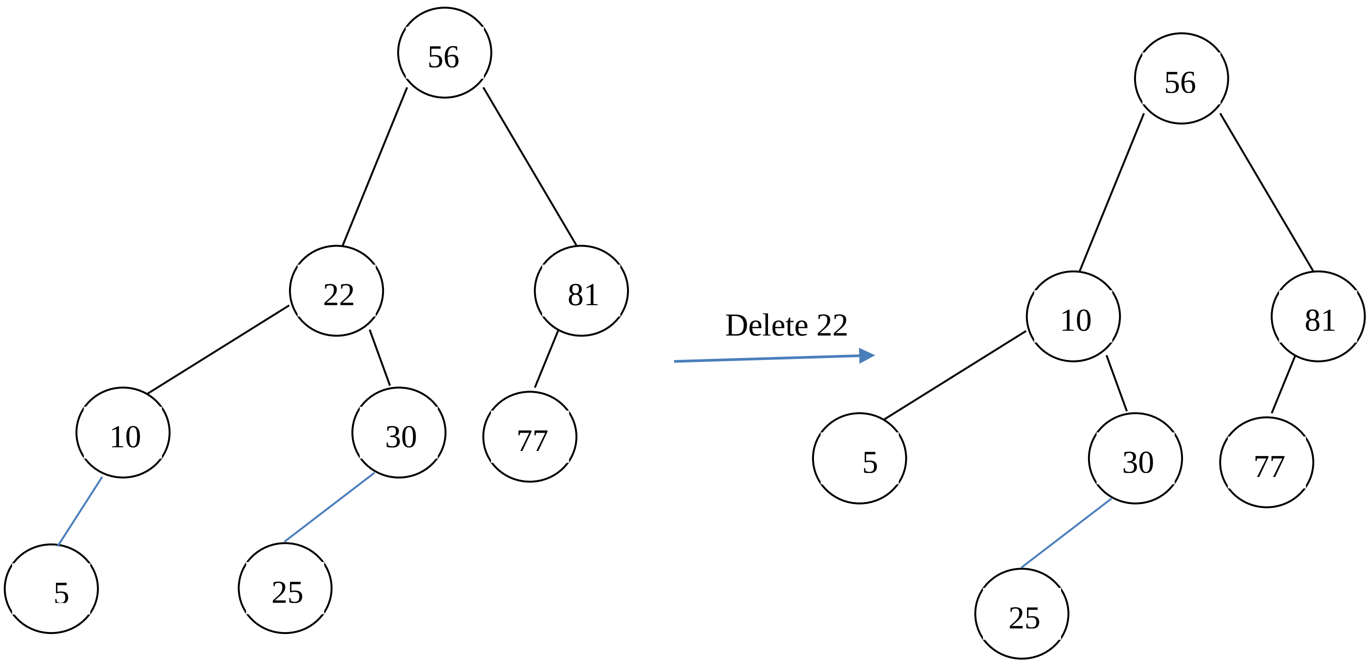
Deleting a Node With Two Children

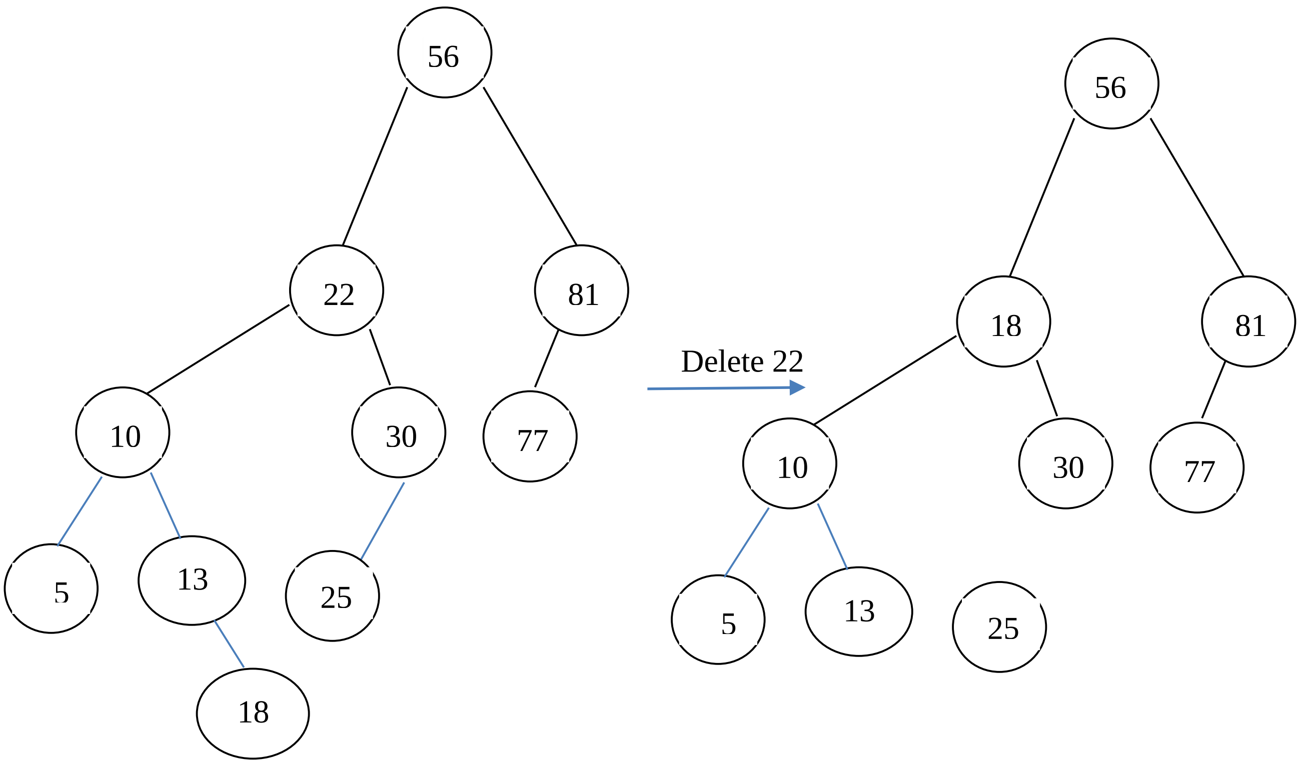
1- replaced node is required to delete the following node her worth and this is aggregated from subtree Left or right for sub-tree node.

2. We take the left sub-tree node (ie node in the left node is required to delete).

+ If you do not have a branch right   they become alternative

+ If we have a branch right we take node at the far right to become the alternative





Exercises

1-Draw the following binary tree then write this expression in infix form:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	….		21	22
-	*	/	+	/	*	T	W	G	P	-	Z	A				R	M

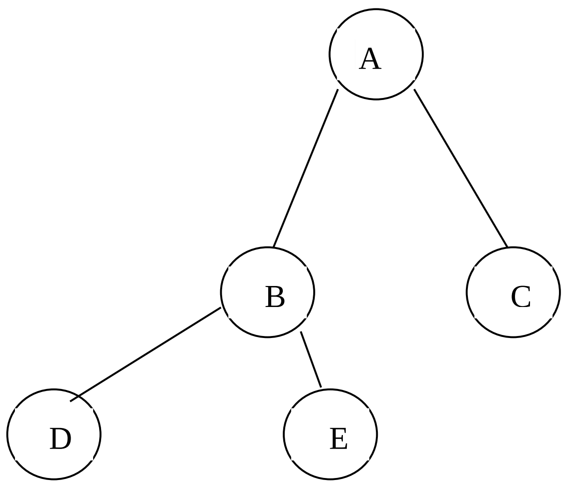
2- create binary search tree from the following numbers:

65 87 76 63 51 92 54 57 62 61 60

Then draw this tree after delete 63 , and traverse this tree in postorder (after deletion)

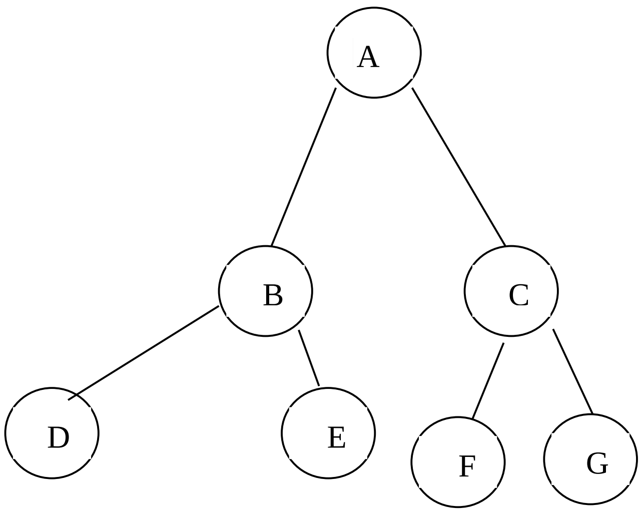
3- Which option best describes this tree ?

a. Not a tree
b. Tree
c. Binary tree
d. Complete binary tree
e. Balanced binary tree



4- Which option best describes this tree ?

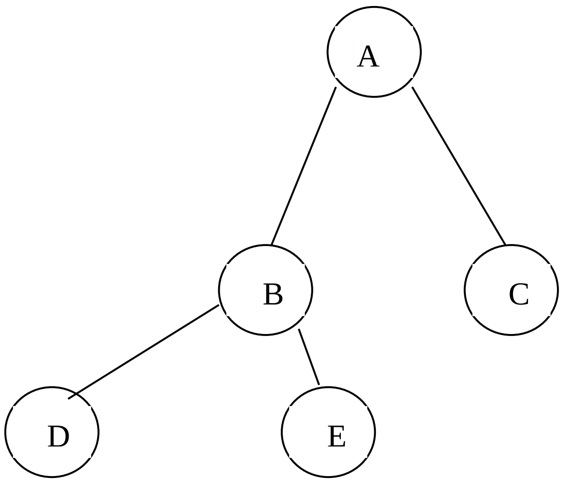
a. Not a tree
b. Tree
c. Binary tree
d. Full Binary tree
e. Complete binary tree
f. Balanced binary tree



5- Which traversal around the tree will give the order D – E – B –F – G –C – A

6- Which traversal around the tree will give the order A – B – D –E – C– F – G

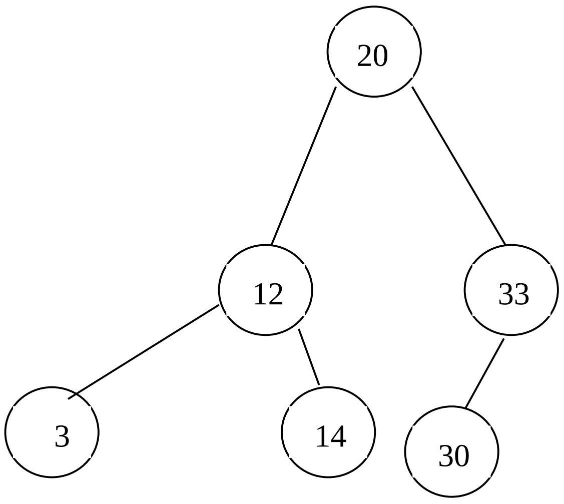
7- Which traversal around the tree will give the order D – B – E – A– F –C – G



8-Select the one FALSE statement about binary trees :

a. Every binary tree has at least one node.
b. Every non-empty tree has exactly one root node.
c. Every node has at most two children .
d. Every non-root node has exactly one parent .

9- Given the following binary tree, answer the following :



1-What is the type of tree?(Binary Search Tree, Balanced binary tree, Full Binary Tree, complete binary tree)

2-What is the depth of the tree?

3-What is the maximum number of nodes in the tree

4-What is the level of node 14 in this tree

5-What is the maximum number of nodes in the level 1.

6- What is the terminal nodes in this tree .

7- What is the non-terminal nodes in this tree .

8- List the values that can be inserted to the right of 30 .

9- List the values that can be inserted to the left of 3 .

10- List the values that can be inserted to the right of 33 .

11- Show the preorder traverse from left to right .

12- Show the post order traverse from left to right .

13- Where can you find the largest data value in a binary search tree.

14- Show the descending order .