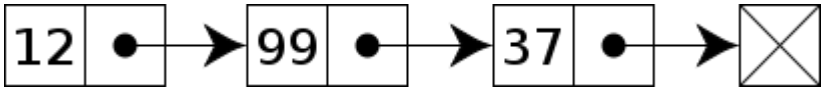
Binary Tree

Binary trees are one of the fundamental data structures used in programming.

1- What is meant by data structure?

Data structure is a way of storing data in a computer so that it can be used efficiently.

- An **array** stores a number of elements in a specific order. They are accessed using an integer to specify which element is required (Index) .it must have fixed length.
- **Linked list** is a data structure consisting of a group of nodes which together represent a sequence. Under the simplest form, each node is composed of a datum and a reference (in other words, a *link*) to the next node in the sequence.

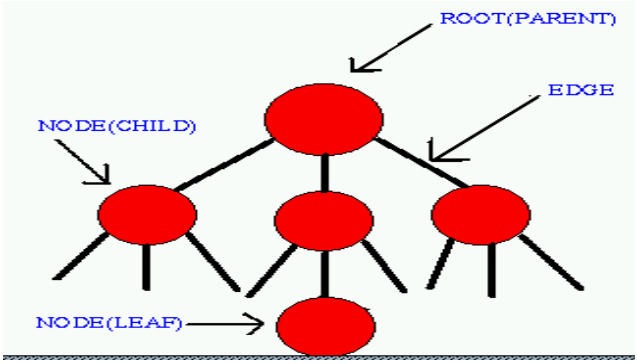


2- Why use binary trees?

Because, it combines the advantages of two other structures: an ordered array and linked list. You can search a tree quickly, as you can an ordered array, and you can insert or delete items quickly as you can with a linked list.

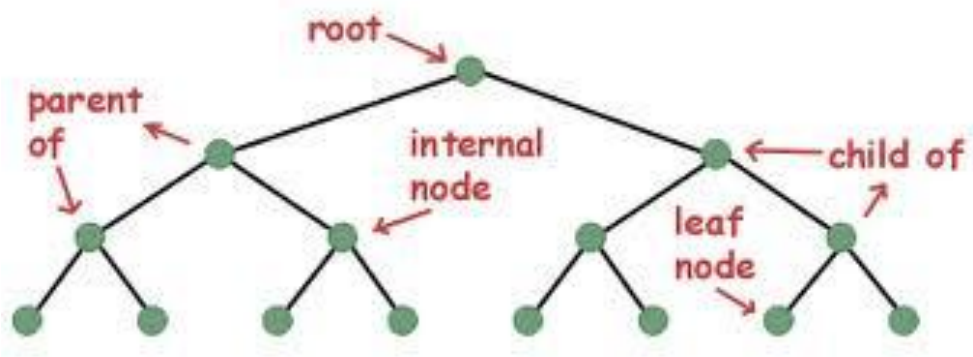
3- What is a tree?

4- A **tree** consists of nodes connected by edges.



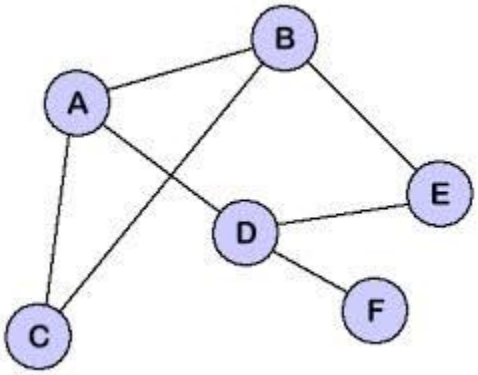
- Nodes are represented by objects.
- Edges represent the way the nodes are related.(are represented in a program by ***references*** (or by pointers if the program written in c++)).

Tree Terminology



- Root

  The node at the top of the tree , there is only one root in a tree. There must be **ONE** and only one path from the root to any other node.



A non-tree

- Path

  Think of someone walking from node to node along the edges that connect them. The resulting sequence of nodes is called a *path.*

- Parent

  Any node (except the root) has exactly one edge running upward to another node. The node above it is called the *parent* of the node.

- Child

  Any node may have one or more lines running downward to other nodes. These node below a given node are called its *children*.
- Leaf

  A node that has no children is called a *leaf* node.

- Subtree

  Any node may be considered to be the root of a *subtree,* which consists of its children, and its children's children.
- Traversing

  To *traverse* a tree means to visit all nodes in some specified order.
- Levels

  The level of a particular node refers yo how many generations the node is from the root.
- Keys

  We've seen that one data field in an object is usually designated a *key value*. This value is used to search for the item or perform other operations on it. In tree diagrams, when a circle represents a node holding a data item, the key value of the items is typically shown in the circle.

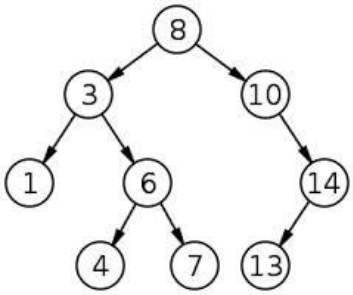
Binary tree

Is a special case of tree, if every node in a tree can have at most two children, the two children of each node are called ***left*** child and ***right*** child.

The kind of binary tree we'll be dealing with in the discussion is technically called a binary search tree.

Binary search tree

A node's left child must have a key less than its parent and a node's right child must hace a key greater than or equal the parent.



it means that
all elements on right of root are greater than it and all elements on left of root are smaller than it

Before beginning in know how to represent the tree in java code, we need to revise some definitions:

- What is the difference between primitive variable and reference variable?

- Primitive types
  - `byte, short, int, long, float, double, Boolean , char.`
  - **primitive variables store primitive values**
- reference types
  - `String, Scanner ,int[],` etc.
  - **reference variables store addresses.**

Primitive variable example :

```
public static void main(String[] args) throws IOException {
    int x = 5;
    int y ;
    y=x;
    x++;
    System.out.println(x+"  "+y);
}
```

Output:

```
run:
6  5
BUILD SUCCESSFUL (total time: 0 seconds)
```

Reference variable example:

```
public static void main(String[] args) throws IOException {
    square x = new square();
    x.length=2;
    x.width=2;
    square y ;
    y=x;
     System.out.println(y.length+" "+y.width);
    y.length++;
     System.out.println(x.length+" "+x.width);

}
static class square
{
    int length;
    int width;
}
```

```
run:
2 2
3 2
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Object oriented example

```
public static void main(String[] args)  {
   person man = new person ();
   man.name="Ahmed";
   man.age=24;
   man.parent=new person();
   person hisFather = man.parent;
   hisFather.name="Aly";
   hisFather.age=55;
   hisFather.parent=new person();
   person hisGrandFather = hisFather.parent;
   hisGrandFather.name="Mohamed";
   hisGrandFather.age=83;
   System.out.println(man.name+" "+man.parent.name+" "+man.parent.parent.name);
    System.out.println(man.name+" "+hisFather.name+" "+hisGrandFather.name);

}
static class person
{
   int age;
   String name;
   person parent ;
}
```

Output

```
run:
Ahmed Aly Mohamed
Ahmed Aly Mohamed
BUILD SUCCESSFUL (total time: 0 seconds)
```

- Now how to represent the tree in java code?

The most common to store the nodes at unrelated locations in memory and connect them using references in each node that point to its children.

1-Node class

```
public class Node {
    int key ;
    Node left ;
    Node right ;
}
```

2-Tree class

```
public class Tree {
    Node root ;
    public void insert (int key){
        //to be continued
    }
    public Node find (int key){
        //to be continued
        return null;
    }
    public boolean delete (int key){
        //to be continued
        return false;
    }
    public void traverse (int traverseType){
        //to be continued
    }
    private void inOrder (Node localRoot){
        //to be continued
    }
    private void preOrder (Node localRoot){
        //to be continued
    }
    private void postOrder (Node localRoot){
        //to be continued
    }
}
```

- "Find" method:

```
public Node find (int key){
    Node current = root ;
    while(current.key!=key){
       if(key<current.key){
           current=current.left;
       }
       else{
           current=current.right;
       }
       if(current==null)
           return null;
    }
    return current;
}
```

- "Insert" method:

```
public void insert(int key) {
    Node newNode = new Node();
    newNode.key = key;
    if (root == null) {
        root = newNode;
    } else {
        Node current = root;
        Node parent;
        while (true) {
            parent = current;
            if (key < current.key) {
                current = current.left;
                if (current == null) {
                    parent.left = newNode;
                    break;
                }
            } else {
                current = current.right;
                if (current == null) {
                    parent.right = newNode;
                    break;
                }
            }
        }
    }
}
```

- “inOrder” method:

  An inorder traversal of a binary search tree will cause all nodes to be visited in *ascending order,* based on their key value.

```
private void inOrder(Node localRoot) {
    if(localRoot!=null)
    {
        inOrder(localRoot.left);
        System.out.println(localRoot.key+"  ");
        inOrder(localRoot.right);
    }
}
```

- “preOrder” method:

```
private void preOrder(Node localRoot) {
    if(localRoot!=null){
        System.out.println(localRoot.key+"  ");
        preOrder(localRoot.left);
        preOrder(localRoot.right);
    }
}
```

- “postOrder” method

```
private void postOrder(Node localRoot) {
    if(localRoot!=null){
        postOrder(localRoot.left);
        postOrder(localRoot.right);
        System.out.println(localRoot.key+"  ");
    }
}
```

- "traverse" method

```
public void traverse(int traverseType) {
    if(traverseType==1)
        inOrder(root);
    else if (traverseType==2)
        preOrder(root);
    else if (traverseType==3)
        postOrder(root);
}
```

- "Delete" method

1-find the node.

2-if the node is the root, make the root equal to null (empty the tree).

3- if the node is leaf node which doesn't have any children , make the node equal to false.

4-if the node has one child,

- If it is a left node make its parent's left node equal to his children.
- If it is a right node make its parent's right node equal to his children.

5-if it has two children,

- Search about "successor" which is the smallest of the set of nodes that are larger than the original node.
- Replace the node with its successor.

- “getSuccessor” Method

```
private Node getSuccessor (Node delNode){
    Node successorParent = delNode;
    Node successor = delNode;
    Node current = delNode.right;
    while(current!=null){
            successorParent=successor;
            successor=current;
            current=current.left;
    }
    if(successor!=delNode.right)
    {
        successorParent.left=successor.right;
        successor.right=delNode.right;
    }
    return successor;
}
```

- “Delete” method

```
public boolean delete(int key) {
    Node current = root;
    Node parent = root;
    boolean isLeftChild = true;
    while (current.key != key) // search for node
    {
        parent = current;
        if (key < current.key) {
            isLeftChild = true;
            current = current.left;
        } else {
            isLeftChild = false;
            current = current.right;
        }
        if (current == null) {
            return false;
        }
    }
    if (current.left == null && current.right == null) {
        current = null;
    } else if (current.left == null) {
        if (current == root) {
            root = current.right;
        } else if (isLeftChild) {
            parent.left = current.right;
        } else {
            parent.right = current.right;
        }
    } else if (current.right == null) {
```

```
            if (current == root) {
                root = current.left;
            } else if (isLeftChild) {
                parent.left = current.left;
            } else {
                parent.right = current.left;
            }

        }else{
            Node Successor = getSuccessor(current);
            if(root==current)
                root=Successor;
            else if (isLeftChild){
                parent.left=Successor;
            }
            else{
                parent.right=Successor;
            }
            Successor.left=current.left;
        }

        return true;
    }
```