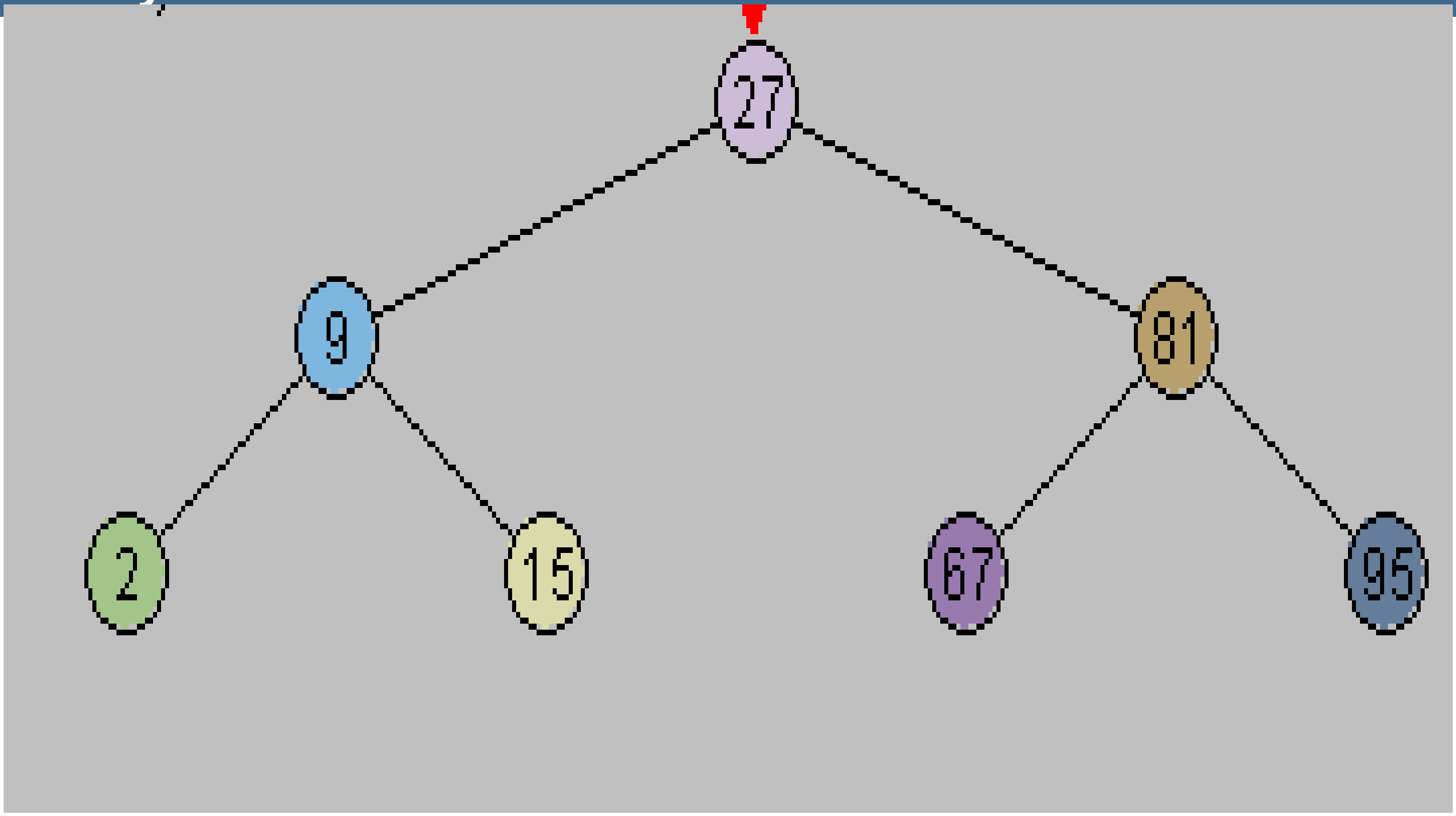
Trees

Binary Search Trees

- **Trees** are one of the **fundamental data structure**.
- Combines the **advantage** of **array** and a **linked list**.
- They are called that, because if you try to visualize the structure, it kind of looks like a tree (root, branches, and leafs).
- Trees are **node based** data structures, meaning that they're made out of small parts called nodes**.**
- Tree Nodes have **two or more child** nodes.
- **Nodes** are **connected by edges**.

Binary Search Trees



Binary Search Trees

- **Recursive structures**
- **Sub-trees** are **disjoint**. That is they **don't share any nodes**. In other word, there is a **unique path** from the **root** of a tree to any other **node** of the tree. This means that **every node** (except the root) has a **unique parent**.

Binary Search Trees

- In **computer programs**, **nodes** often represent such **entities** as people, car, airline reservations, and so on.
- In an **OOP** language like **Java** these real-word entities are represented by **objects**.
- The lines (**edges**) between the nodes represent the way the nodes are related.

Binary Trees

Binary Search Trees

- Each **node** is capable of **two children**.
- Represent an important technique for handling structures such as files and directories, dictionaries, and symbol tables.
- If every node in a tree can have at most **two children**, the tree is called a **binary tree**. The two children of each node in a binary tree are called the **left child and the right child**. A node in binary tree doesn't necessarily have the maximum of two children; it may have **only a left child**, or on a **right child**, or it can have **no children at all**.

Binary Search Trees

- **Binary tree** is a **dynamic data structure**, that is **memory** for its nodes is **allocated** and **de-allocated** during **program execution**.
- **Maximum** number of **node** at any level n is **2^n**
- The **maximum** number of **level** in **n** number of node tree is **n**.
- The **minimum** number of **level** in **n** number of node tree is **log (n).**

Binary Search Tree(BST)

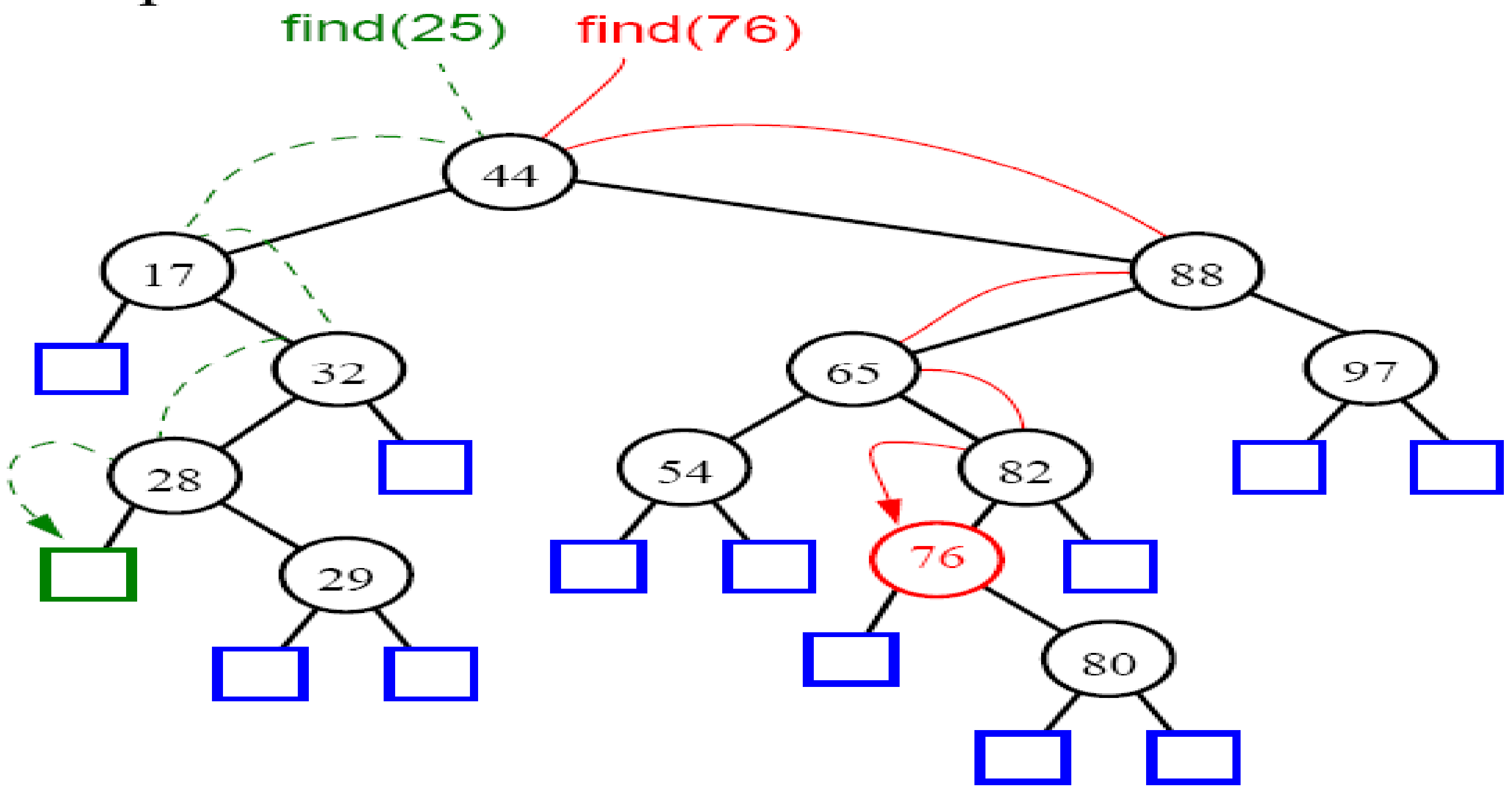
Binary Search Trees

- To support **O (log n) search**, we add a **property** to **binary tree**:
  - **Smaller** and **equal** value to **left**, and **greater** values to **right**.
- In this case **BST** has the **benefits** of both **sorted array** and **linked lists**.
- BSTs are suitable for **applications** in which **search time** must be **minimized** or in which the nodes are not necessarily processed in **sequential order**.
- Tradeoff: **BSTs** with its **extra reference** in each node, takes up more **memory space** than a single linked list. In addition, the **algorithms** for manipulating the tree are somewhat more **complicated**.
- In its **worst case** if the elements were inserted in order from smallest to largest or vice versa, the tree won't really be a a tree at all, it would be a **linear list**. This called **degenerate tree**.

Binary Search Trees

Search (cont.)

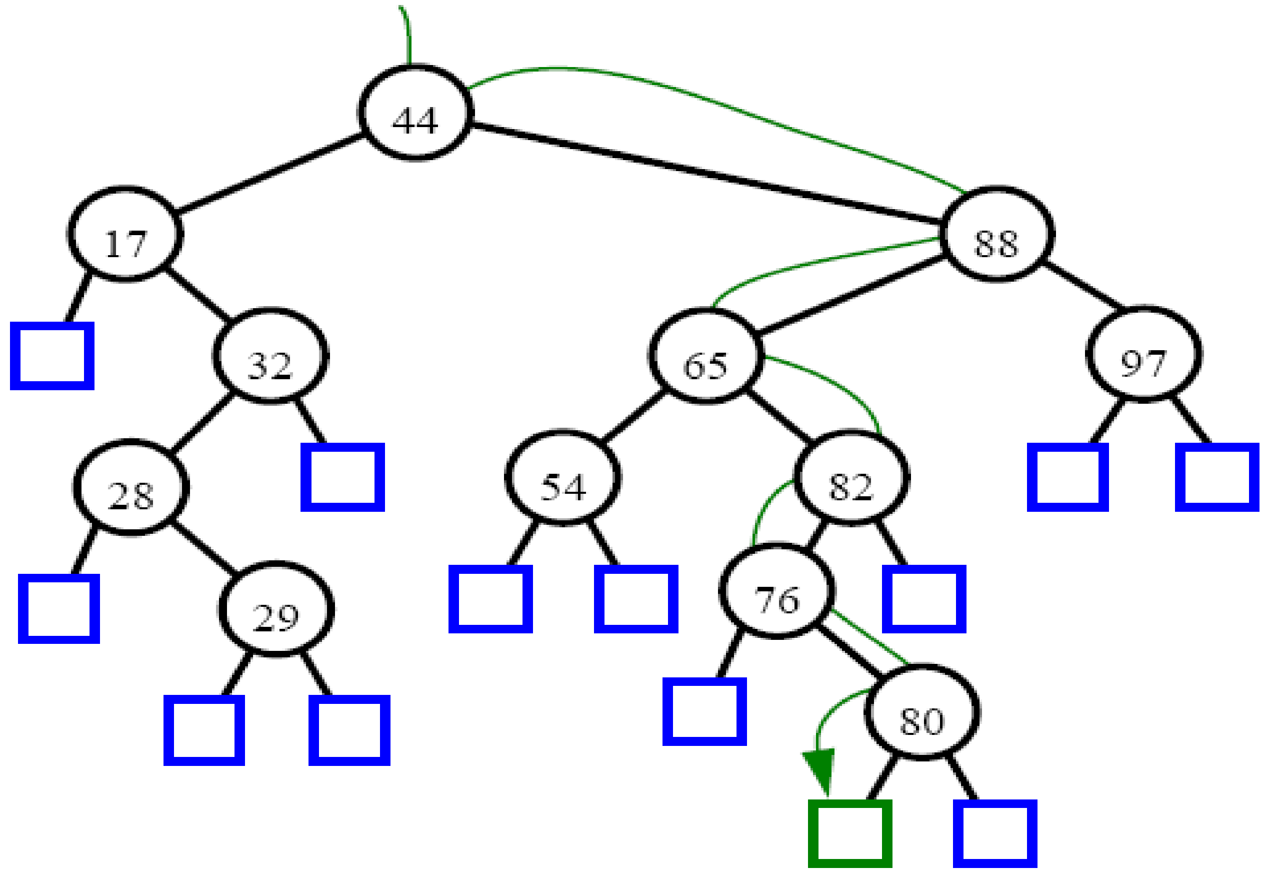
- A picture:



Binary Search Trees

- Insertion of an element with key 78:

a)



Balancing a BST

Binary Search Trees

- **O (log n)** → if tree is **balanced**.
- In **worst case O (n).**
- **Full binary tree** → a binary tree in which **all** of the **leaves** are on the **same level** and every **non-leaf** node has **2 children**.
- **Complete binary tree** : a binary tree that is **either full** or full through the **next-to-last level** with the leaves on the last level as far to the left as possible.

Tree terminology

Binary Search Trees

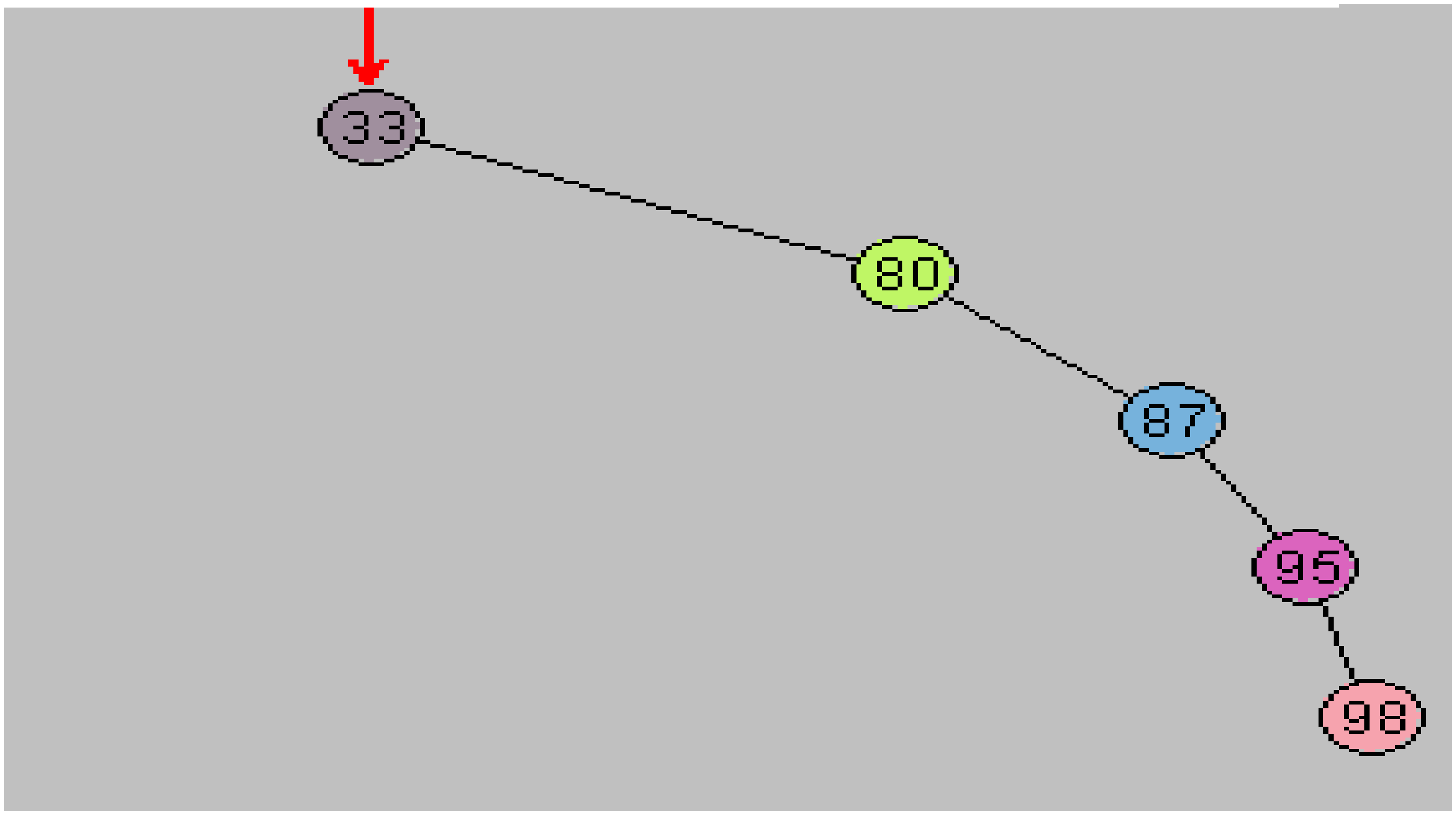
- **Path**: Sequence of nodes.
- **Root**: The node at the top of the tree is called the root. There is only one root in a tree.
- **Parent**: the node above it is called the parent of the node.
- **Child**: The nodes below a given node are called its children.
- **Leaf**: a node that has no children is called a leaf node or simply a leaf.
- **Sub-tree**: Any node may be considered to be the root of a sub-tree, which consists of its children and children's children, and so on.
- **Visiting**: A node is visited when program control arrives at the node, usually for the purpose of carrying out some operation on the node, such as checking the value of one of its data fields or display it.

Binary Search Trees

- **Traversing**: To traverse a tree means to visit all the nodes in some specified order. For example, you might visit all the nodes in order of ascending key value.
- **Levels**: The level of a particular node refers to **how many generations** the node is from the root.
- **Keys**: One **data field** in an object is usually designated a key value. This value is used to **search** for the item or perform other operation on it.
- **Unbalanced Trees**: Some of the trees you generate are unbalanced; that is, they have most of their nodes on **one side** of the root or the other.

Un-balanced Binary Search Tree

Binary Search Trees



Binary Search Trees

- **Representing the Tree in Java code:**
- There are **several approaches** to representing a tree in the **computer's memory**. The most common is to store the nodes at **unrelated locations** in memory, and connect them using **references** in each node that point to its children.
- **The node class:**
- First, we need a **class of node object**. These objects contain the data representing the objects being stored and also references to each of the node's two children.

```
class element
{
   int  idno;
   String stname;
   element leftchild;
   element rightchild;
   }
```

Binary Search Trees

- There is another way to define node:

```
class element
{
    student st1;
     element leftchild;
     element rightchild;
}
class student
{
    int idno;
    String stname;
}
```

Java code for Inserting a Node

Binary Search Trees

```
public void insert(int id, String name)
   {
       element newstudent = new element ();
       newstudent.idno= id;
       newstudent.stname = name;
       if(root == null)
          root = newstudent;
       else
       {
           element current = root;
           element parent;
          while(true)
          {
              parent = current;
              if(id < current.idno)
              {
                  current = current.leftChild;
```

Binary Search Trees

```
if(current == null)
          {
            parent.leftChild = newstudent;
            return;
          }
        }
        else
        {
          current = current.rightChild;
          if(current == null)
          {
            parent.rightChild = newstudent;
            return;
          }
        }
      }
    }
  }
```

Java code for finding a Node

Binary Search Trees

```
public element find(int key)
    {
       element current = root;
        while(current.idno != key)
        {
            if(key < current.idno)
                current = current.leftChild;
            else
                current = current.rightChild;
            if(current == null)
                return null;
        }
        return current;
    }
```

Deleting an Item from a BST

- **Most complicated** common operation required for binary search trees.
- Start by **finding** the node you want to delete. When you have found the node, there are **three cases** to consider:
  - The node to be deleted is a **leaf** (has no children).
  - The node to be deleted **has one child**.
  - The node to be deleted **has two children**.
- The **first case is easy**. You simply change the appropriate child field in the node's parent to point to null.

Binary Search Trees

- The **second case** isn’t so bad either. The node has only two connections: to its parent & to its only child. You have to connect its parent directly to its child.
- If the deleted node hade **two children**, you can’t just replace it with one of these children, at least if the child has its own children. To delete a node with two children, **replace** the node with its **in-order successor.**

Binary Search Trees

```
public boolean delete(int key)
    {
        element current = root;
        element parent = root;
        boolean isleftChild = true;
```

Binary Search Trees

```
while(current.idno != key)
      {
         parent = current;
         if(key < current.idno)
         {
            isleftChild = true;
            current = current.leftChild;
         }
         else
         {
            isleftChild = false;
            current = current.rightChild;
         }
         if(current == null)
            return false;
      }
```

Binary Search Trees

```
if(current.leftChild == null && current.rightChild == null)
        {
            if(current == root)
                root = null;
            else if(isleftChild)
                parent.leftChild = null;
            else
                parent.rightChild = null;
        }
        else if(current.rightChild == null)
            if(current == root)
                root = current.leftChild;
            else if(isleftChild)
                parent.leftChild = current.leftChild;
            else
                parent.rightChild = current.leftChild;
```

Binary Search Trees

```
else if(current.leftChild == null)
            if(current == root)
                root = current.rightChild;
            else if(isleftChild)
                parent.leftChild = current.rightChild;
            else
                parent.rightChild = current.rightChild;
        else
        {
 element successor = getSuccessor(current);
        if(current == root)
            root = successor;
        else if(isleftChild)
            parent.leftChild = successor;
        else
            parent.rightChild = successor;
        successor.leftChild = current.leftChild;
    }
    return true;
}
```

Binary Search Trees

```
private element getSuccessor(element delelement) {
 element successorParent = delelement;
 element successor = delelement;
 element current = delelement.rightChild;
    while(current !=null) {
        successorParent = successor;
        successor = current;
        current = current.leftChild;
    }
    if(successor != delelement.rightChild) {
        successorParent.leftChild = successor.rightChild;
        successor.rightChild = delelement.rightChild;
    }
    return successor;
}
```

Finding maximum and minimum values

Binary Search Trees

```
Pubic element maximum( )
{
   node current, last;
   current = root;
   while (current !=null)
   {
      last = current;
      current = current.rightchild;
   }
   return last;
}
```

Finding maximum and minimum values

Binary Search Trees

```
Pubic element minimum( )
{
 element current, last;
   current = root;
   while (current !=null)
   {
      last = current;
      current = current.leftchild;
   }
   return last;
}
```

Java code for displaying Tree

Binary Search Trees

```
public void displayTree()
{
    stack globalStack = new stack(25);
    globalStack.push(root);
    int nBlanks = 32;
    boolean isRowEmpty = false;
    System.out.println("................................");
    while(isRowEmpty == false)
    {
        stack localStack = new stack(25);
        isRowEmpty = true;
        for(int j=0; j<nBlanks; j++)
            System.out.print("  ");
```

Binary Search Trees

```
while(globalStack.isempty() == false) {
        treeitem temp = globalStack.pop();
        if(temp != null) {
            System.out.print(temp.idno);
            localStack.push(temp.leftchild);
            localStack.push(temp.rightchild);
            if(temp.leftchild != null  || temp.rightchild != null)
                isRowEmpty = false;
        }
```

Binary Search Trees

```
else {
            System.out.print(".....");
            localStack.push(null);
            localStack.push(null);
         }
         for(int j=0; j<nBlanks*2-2;j++)
            System.out.print("  ");
      }
      System.out.println();
      nBlanks /= 2;
      while(localStack.isempty() == false)
         globalStack.push(localStack.pop() );
   }
      System.out.println("..................");
 }
```

Binary Search Trees

- **Traversing the Tree:**
- Traversing a tree means **visiting each node** in a specified order. There are **three simple ways** to traverse a tree:
  - **Preorder,**
  - **Inorder,**
  - **Postorder,**
- The simplest way to carry out a traversal is the use of **recursion**.

Binary Search Trees

In-order Traversal:

- Tree Nodes have two or more child nodes; unlike our list node, which only had one child.
- An in-order traversal of a binary search tree will cause all the nodes to be visited in **ascending order** based on their key values. If you want to create a sorted list of the data in a binary tree, this is one way to do it. The simplest way to carry out a traversal is the use of **recursion**. The method needs to do only three things:

1. **call itself to traverse the node's left subtree.**
2. **visit the node .**
3. **call itself to traverse the node's right subtree.**

Preorder and postorder traversals:

- These traversal are useful if you are writing programs that parse or analyze **Algebric expression.**

Binary Search Trees

```
public void traverse(int traverseType)
{
    switch(traverseType)
    {
        case 1: System.out.print("\nPreorder traversal: ");
        preOrder(root);
        break;
        case 2: System.out.print("\nInorder traversal:  ");
        inOrder(root);
        break;
        case 3: System.out.print("\nPostorder traversal:  ");
        postOrder(root);
        break;
    }
    System.out.println();
}
```

Binary Search Trees

```
private void preOrder(element localRoot)
{
    if(localRoot != null)
    {
        System.out.print(localRoot.iData + "  ");
        preOrder(localRoot.leftChild);
        preOrder(localRoot.rightChild);
    }
}
```

Binary Search Trees

```
private void inOrder(element localRoot)
{
    if(localRoot != null)
    {
        inOrder(localRoot.leftChild);
        System.out.print(localRoot.iData + "  ");
        inOrder(localRoot.rightChild);
    }
}
```

Binary Search Trees

```
private void postOrder(element localRoot)
{
    if(localRoot != null)
    {
        postOrder(localRoot.leftChild);
        postOrder(localRoot.rightChild);
        System.out.print(localRoot.iData + "   ");
    }
}
```

Complete Java program for Tree

Binary Search Trees

```
class element
{
   int  idno;
   String stname;
   element leftchild;
   element rightchild;
   }
```

Binary Search Trees

```
class binaryTree {
   public element root;
    public void insert(int id, String name)
    {
        element newstudent = new element ();
        newstudent.idno= id;
        newstudent.stname = name;
        if(root == null)
            root = newstudent;
        else
        {
             element current = root;
             element parent;
```

Binary Search Trees

```
while(true)
        {
            parent = current;
            if(id < current.idno)
            {
                current = current.leftchild;
                if(current == null)
                {
                    parent.leftchild = newstudent;
                    return;
                }
            }
            else
            {
                current = current.rightchild;
                if(current == null)
                {
                    parent.rightchild = newstudent;
                    return;
                }
            }
        }
    }
}
```

```
public void displayTree() {
        inOrder(root);
    }
public void inOrder(element
localRoot) {
    if(localRoot != null)  {
        inOrder(localRoot.leftchild);
        System.out.print(localRoot.idno
+ "  ");
        inOrder(localRoot.rightchild);
    }
}


}
```

Binary Search Trees

```
class binaryTreeApp {
       public static void main(String [] args) {
        binaryTree tree1 = new binaryTree();
        tree1.insert(80, "Sharif");
        tree1.insert(30, "Shams");
        tree1.insert(120, “Jalal");
        tree1.insert(125, “Jamal");
        tree1.insert(100, “Jawed");
        tree1.insert(90, “Jamil");
        tree1.insert(95, “Zabi");
        tree1.insert(60, “Wali");
        tree1.insert(20, “Khan");
        tree1.insert(10, “Karim");
        tree1.insert(45, “Sultan");
        tree1.insert(40, “Zobair");
        tree1.displayTree();

    }
}
```

Binary Search Trees

- ***The efficiency of Binary Trees:***
- As you have seen, most operations with trees involve **descending** the tree from level to level to find a particular node. How long does it take to do this? In a **full tree**, about **half the nodes** are on the **bottom level.**
- Thus, about half of all **searches** or **insertions** or **deletions** require finding a node on the **lowest level.** An additional **quarter** of these operations require finding the node on the **next-to-lowest level**, & so on. During a search we need to visit one node on each level. So we can get a good idea how long it takes to carry out these operations by knowing **how many levels there are:**

Binary Search Trees

Number of nodes	Number of levels
1	1
3	2
7	3
15	4
31	5
… …	
1023	10
32767	15
… …	
1048575	20
… …	
33554432	25
… …	
1073741824	30

Binary Search Trees

- This situation is very much like the **ordered array**. In that case the number of **comparison** for a binary search was approximately equal to the **base 2 logarithm** of the number of cells in the array. Thus, the time needed to carry out the common tree operations is **proportional** to the **base 2 log of N**.