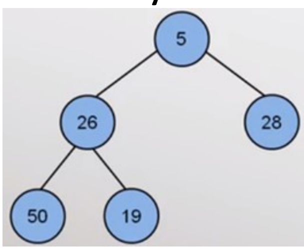
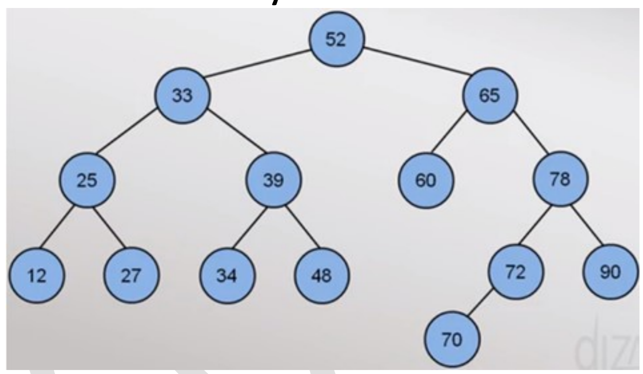
Binary Search Trees (BST)

- **Problem**: searching in binary tree takes **O(n)**.
- **Solution**: forming a binary search tree.
- In a **binary search tree** for every node , **X**, in the tree, the values of all the items in its **left subtree** are smaller than the item in **X**, and the values of all the items in its **right subtree** are larger (***or equal if duplication is allowed***) than the item in **X**.

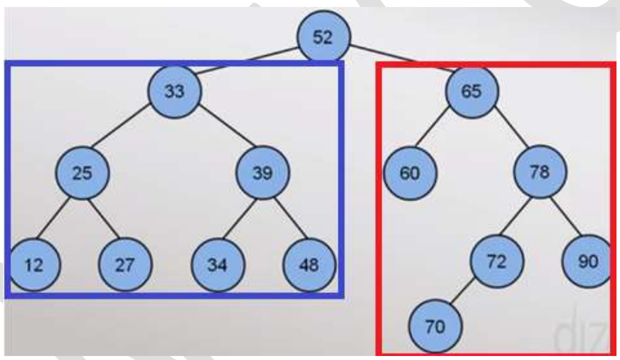
Binary Tree



Binary Search Tree



- Every node in a binary search tree is the root of a binary search tree.



- **Search for an item:**

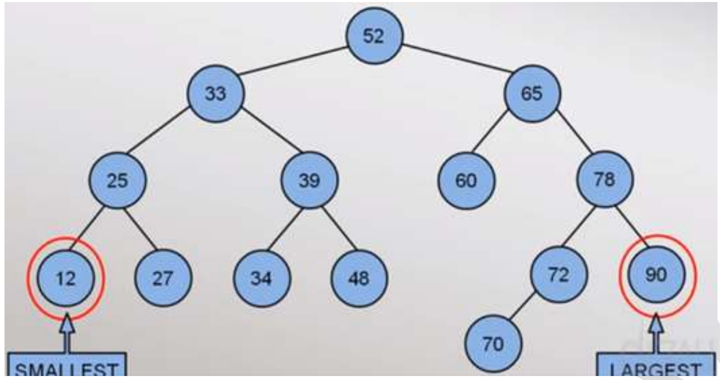
Example: find(52) , find(39) , find(35)

```java
public TNode find(T data) {  return find(data, root);   }
public TNode find(T data, TNode node) {
   if (node!= null) {
      int comp = node.data.compareTo(data);
      if (comp == 0)
         return node;
      else if (comp > 0 && node.hasLeft())          return find(data, node.left);
      else if (comp < 0 && node.hasRight())         return find(data, node.right);
   }
   return null;
}
```

Efficiency: Searching a binary search tree of height **h** is **O(h)**

However, to make searching a binary search tree as efficient as possible, tree must be as <u>**short**</u> as possible.

Finding Max and Min Values:



- The find **Min** operation is performed by following left nodes as long as there is a **left** child.
- The find **Max** operation is similar.

```
public TNode largest() {  return largest(root);   }
public TNode<T> largest(TNode node) {
   if(node!= null){
      if(!node.hasRight())
         return (node);
      return largest(node.right);
   }
   return null;
}

public TNode smallest() {  return smallest(root);   }
public TNode<T> smallest(TNode node) {
   if(node!= null){
      if(!node.hasLeft())
         return (node);
      return smallest(node.left);
   }
   return null;
}
```

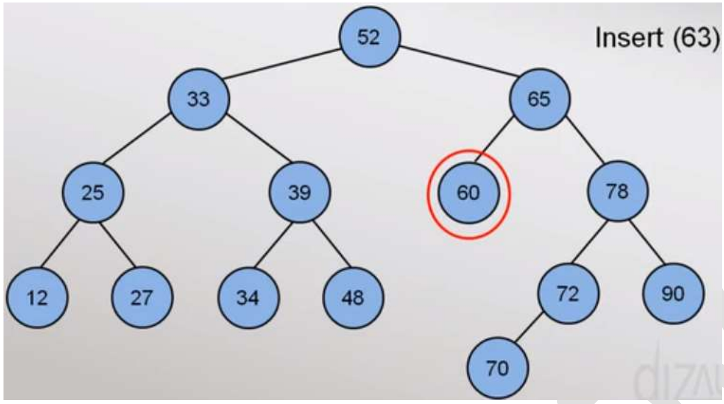
Tree Height:

```
public int height() {  return height(root); }
public int height(TNode node) {
   if (node == null) return 0;
   if (node.isLeaf()) return 1;
   int left = 0;
   int right = 0;
   if (node.hasLeft())
            left = height(node.left);
   if (node.hasRight())
            right = height(node.right);
   return (left > right) ? (left + 1) : (right + 1);
}
```

H.W. Implement Tree Size

Insert in Binary Search Tree:

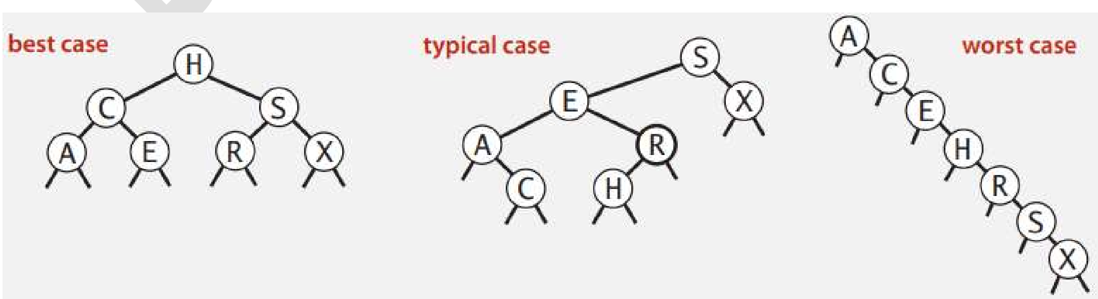
Example: insert(63)




```java
public void insert(T data) {
   if (isEmpty())
      root = new TNode(data);
   else
      insert(data, root);
}
public void insert(T data, TNode node) {
   if (data.compareTo((T) node.data) >= 0) { // insert into right subtree
      if (!node.hasRight())
         node.right = new TNode(data);
      else
         insert(data, node.right);
   } else {       // insert into left subtree
      if (!node.hasLeft())
         node.left = new TNode(data);
      else
         insert(data, node.left);
   }
}
```

Tree Shape:

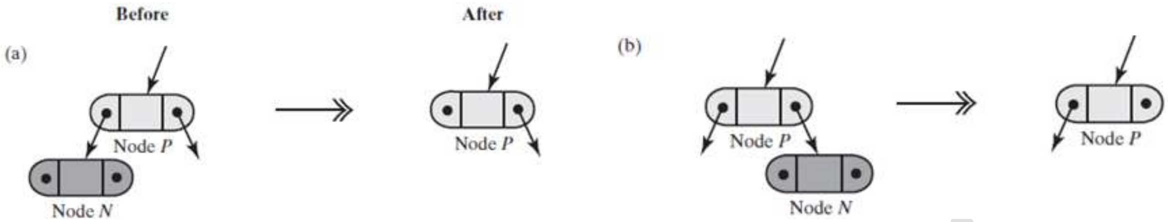
Tree shape depends on order of insertion.



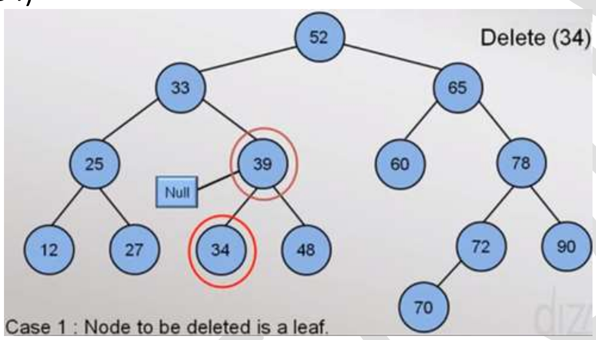
Deleting a Node:

Case 1: Node to be deleted is a **leaf**. Two possible configurations of a leaf node N:

Being a **left** child or a **right** child:



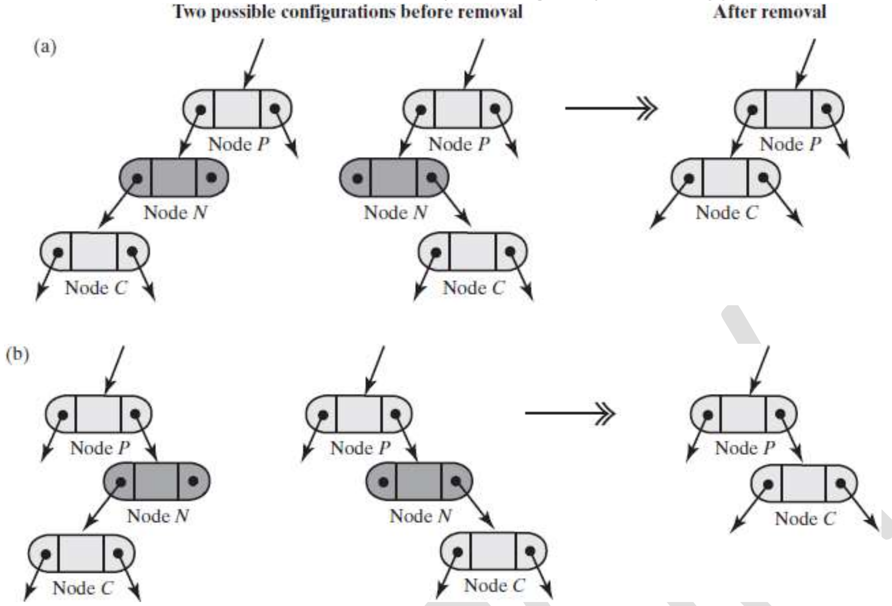
Example: delete(34)



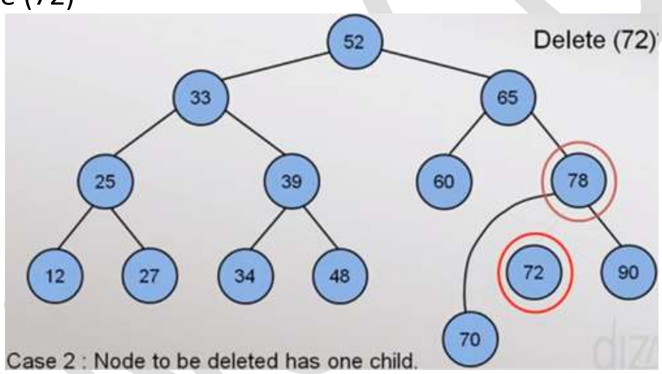
Case 1 : Node to be deleted is a leaf.

```
public TNode delete(T data) {
   TNode current = root;
   TNode parent =  root;
   boolean isLeftChild = false;

   if (isEmpty()) return null;     // tree is empty
   while (current != null && !current.data.equals(data)) {
      parent = current;
      if (data.compareTo((T)current.data) < 0) {
         current = current.left;
         isLeftChild = true;
      } else {
         current = current.right;
         isLeftChild = false;
      }
   }
   if (current == null) return null; // node to be deleted not found

   // case 1: node is a leaf
   if (!current.hasLeft() && !current.hasRight()) {
      if (current == root)  // tree has one node
         root = null;
      else {
         if (isLeftChild)             parent.left = null;
         else                         parent.right = null;
      }
   }

   // other cases
   return current;
}
```

Case 2: If a node has **one child**, it can be removed by having its parent **bypass** it.



Example: delete (72)

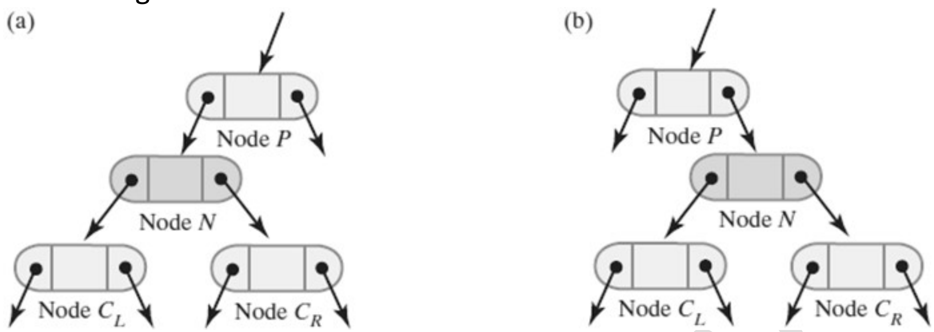


Note: The **root** is a special case because it does not have a parent.

```
// Case 2 broken down further into 2 separate cases
else if (current.hasLeft()) {  // current has left child only
   if (current == root) {
      root = current.left;
   } else if (isLeftChild) {
      parent.left = current.left;
   } else {
      parent.right = current.left;
   }
} else if (current.hasRight()) {  // current has right child only
   if (current == root) {
      root = current.right;
   } else if (isLeftChild) {
      parent.left = current.right;
   } else {
      parent.right = current.right;
   }
}
```

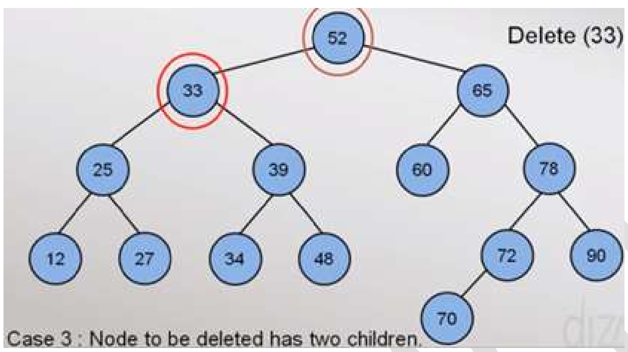
Case 3:

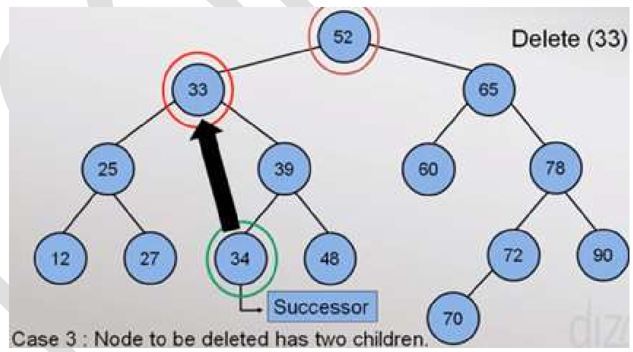
- Two possible configurations of a node N that has two children:

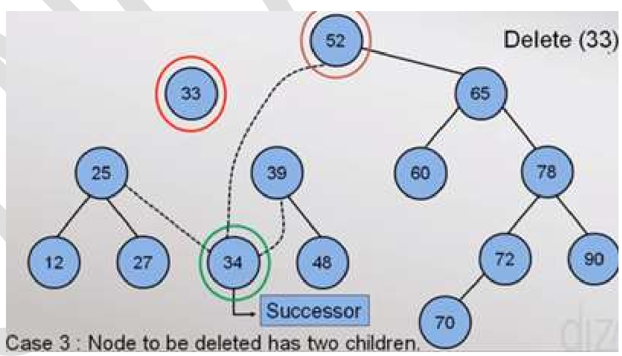


- A node with two children is replaced by using the **smallest** item in the right subtree (**Successor**).

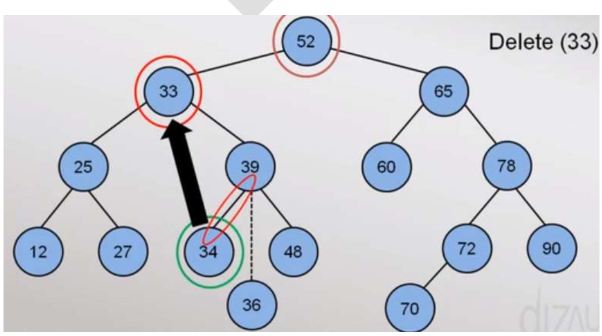
Example: delete(33)



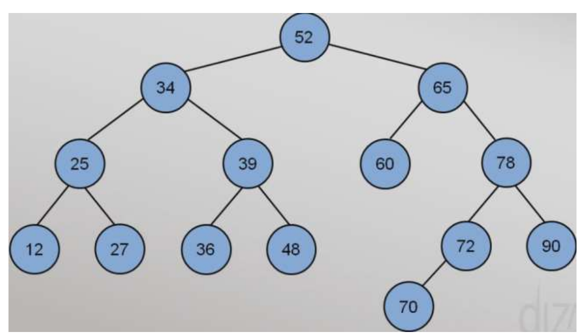




What if node **34** has a right child (e.g. **36**)?



➔



```
// case 3: node to be deleted has 2 children
else {
   Node successor = getSuccessor(current);
   if (current == root)
      root = successor;
   else if (isLeftChild) {
         parent.left  = successor;
      } else {
         parent.right = successor;
      }
   successor.left = current.left;
}
```

```
private Node getSuccessor(Node node) {
   Node parentOfSuccessor = node;
   Node successor = node;
   Node current = node.right;
   while (current != null) {
      parentOfSuccessor = successor;
      successor = current;
      current = current.left;
   }
   if (successor != node.right) {  // fix successor connections
      parentOfSuccessor.left = successor.right;
      successor.right = node.right;
   }
   return successor;
}
```

Soft Delete (lazy deletion):

When an element is to be deleted, it is left in the tree and simply **marked** as being deleted.

- If a deleted item is reinserted, the overhead of allocating a new cell is avoided.

Efficiency of Operations:

- For tree of height ***h***
  - The operations **add**, **delete**, and **find** are **O(h)**
- If tree of ***n*** nodes has height ***h = n***
  - These operations are **O(n)**
- Shortest tree is **complete**
  - Results in these operations being **O(log n)**