BIRZEIT UNIVERSITY

COMP242

Data Structure

Lectures Note: Binary Trees

Prepared by: **Dr. Mamoun Nawahdah**

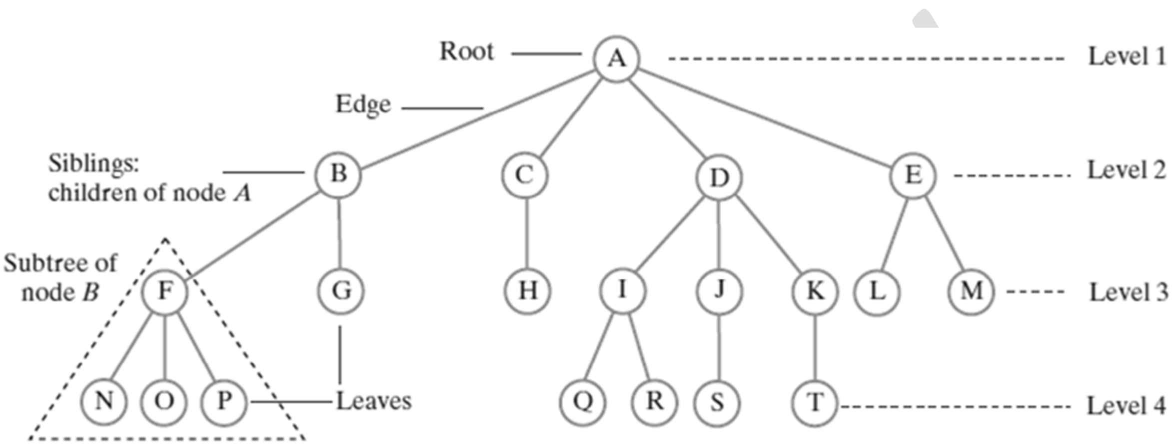
2016/2017

Trees

Revision:

	Sorted Arrays	Sorted Linked List
Search	**Fast O(log n)**	**Slow O(n)**
Insert	**Slow O(n)**	**Slow O(n)**
Delete	**slow O(n)**	**Slow O(n)**

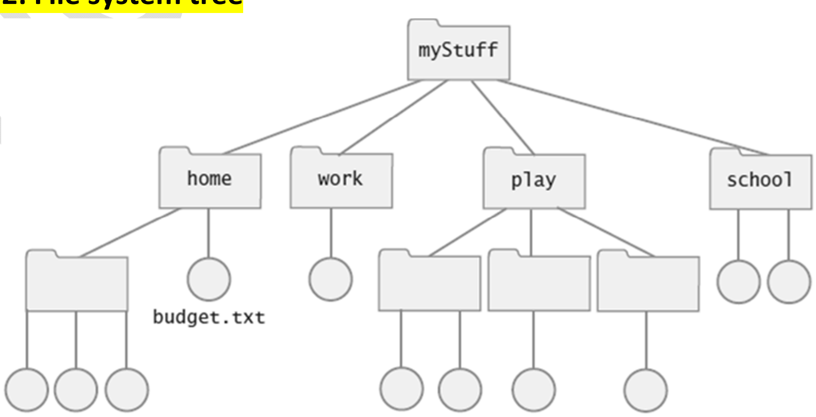
Tree



- A **tree** is a collection of *N* **nodes**, one of which is the **root**, and *N* 1 **edges**.
- Every node except the **root** has one **parent**.
- Nodes with no children are known as **leaves**.
- An **internal node (parent)** is any node that has at least one non-empty child.
- Nodes with the same parent are **siblings**.
- The ***depth of a node*** in a tree is the length of the path from the **root** to the node.
- The ***height*** of a tree is the number of levels in the tree.

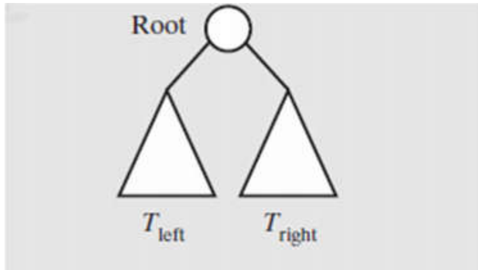
Example 1: Family Trees (one parent)

Example 2: File system tree



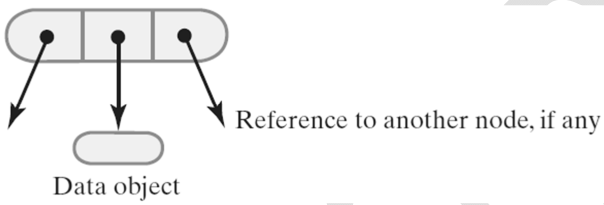
Binary Trees

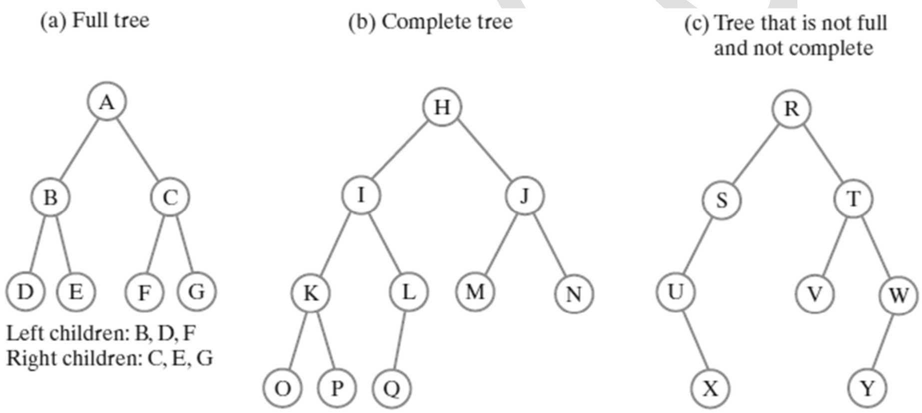
- A **binary tree** is a tree in which no node can have **more** than **two** children:



where T_{left} and T_{right} are binary trees.

- Binary Tree **Node**:





(a) Each node in a **full binary tree** is either:

(1) an internal node with **exactly** two non-empty children or

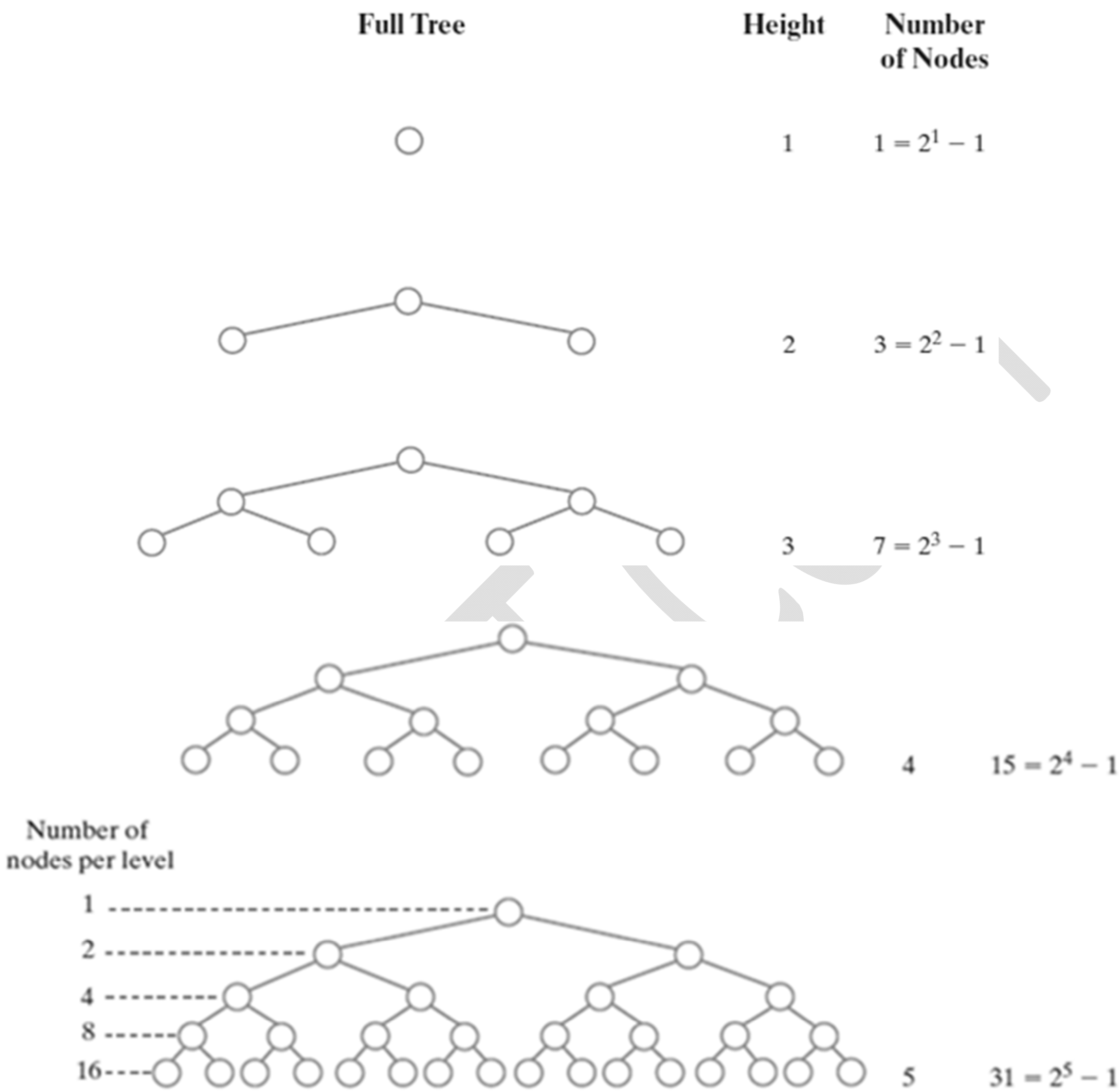
(2) a leaf.

(b) A **complete binary tree** has a restricted shape obtained by starting at the root and filling the tree by levels from **left** to **right**.

(a) This tree is full (but not complete).

(b) This tree is complete (but not full).

- The maximum number of nodes in a full binary tree as a function of the tree's height **= 2^h-1**



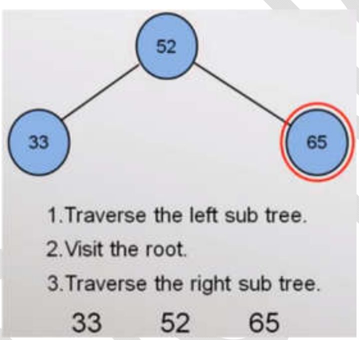
Implementation:

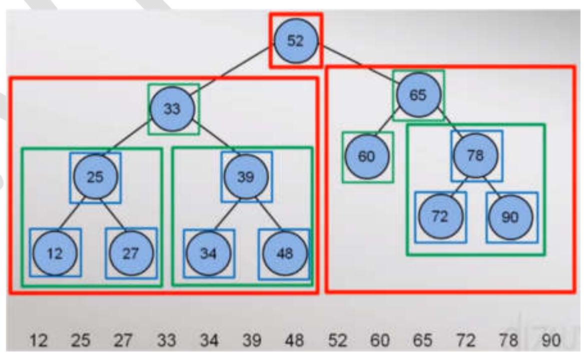
```
public class TNode<T extends Comparable<T>> {
   T data;
   TNode left;
   TNode right;

   public TNode(T data) {    this.data = data; }
   public void setData(T data)    {  this.data = data;  }
   public T getData()    {  return data;  }
   public TNode getLeft() {  return left;  }
   public void setLeft(TNode left) {  this.left = left; }
   public TNode getRight() {  return right;  }
   public void setRight(TNode right) { this.right = right;}
   public boolean isLeaf(){   return (left == null && right == null);   }
   public boolean hasLeft(){  return left != null;  }
   public boolean hasRight(){  return right != null;  }
   public String toString() {      return "[" + data + "]";    }
}
```

Tree Traversal

Definition: visit, or process, each data item exactly once.

- **In-Order Traversal:** Visit **root** of a binary tree between visiting nodes in root's subtrees.

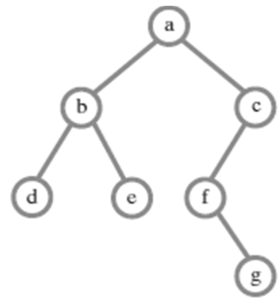




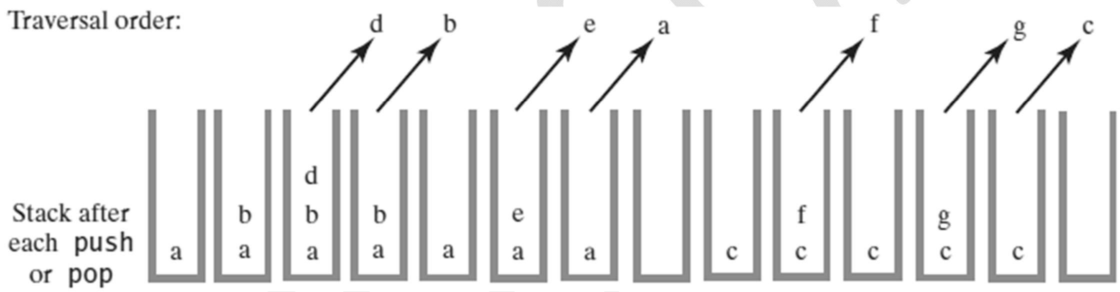
  - **Recursive implementation:**

```
public void traverseInOrder() {  traverseInOrder(root);   }
public void traverseInOrder(TNode node) {
   if (node != null) {
      if (node.left != null)
         traverseInOrder(node.left);
      System.out.print(node + " ");
      if (node.right != null)
         traverseInOrder(node.right);
   }
}
```

- **Using a stack to perform an in-order traversal <u>iteratively</u>: (Optional)**

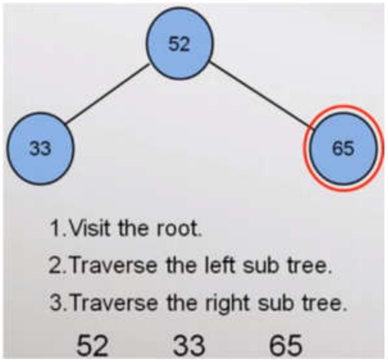


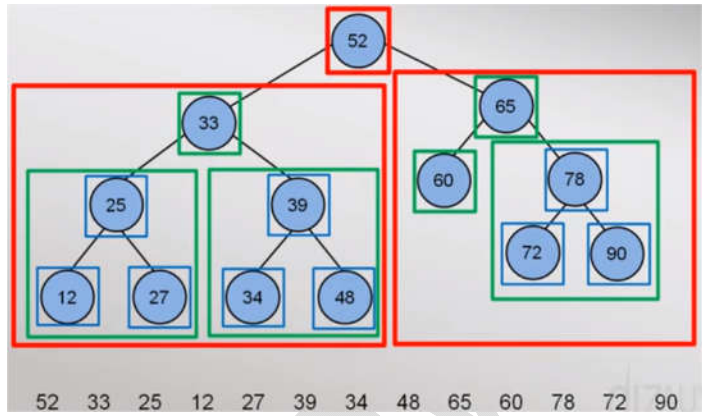
1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current→left until current is NULL
4) If current is NULL and stack is not empty then
   a) Pop the top item from stack.
   b) Print the popped item, set current = popped_item→right
   c) Go to step 3.
5) If current is NULL and stack is empty then we are done.



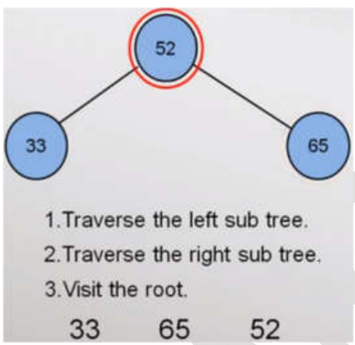

```
void traverseInOrder () {
        if (root == null) return;
        Stack<Node> stack = new Stack<Node>();
        Node node = root;
        //first node to be visited will be the left one
        while (node != null) {
            stack.push(node);
            node = node.left;
        }
        // traverse the tree
        while (!stack.isEmpty()) {
            // visit the top node
            node = stack.pop();
            System.out.print(node.data + " ");
            if (node.right != null) {
                node = node.right;
                // the next node to be visited is the leftmost
                while (node != null) {
                    stack.push(node);
                    node = node.left;
                }
            }
        }
    }
```

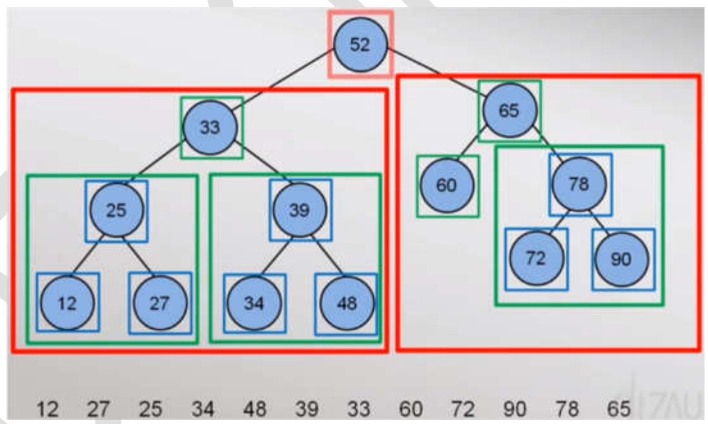
- **Pre-Order Traversal:** Visit **root** before we visit root's subtrees.



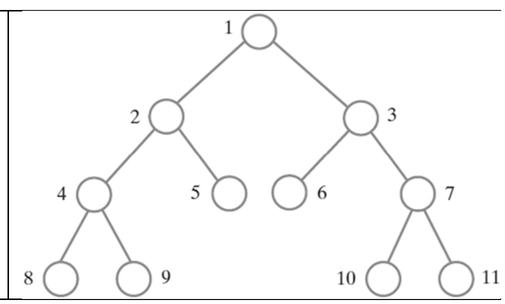


- **Post-Order Traversal:** Visit **root** of a binary tree after visiting nodes in root's subtrees.





- **Level-Order Traversal:** Begin at **root** and visit nodes one level at a time.
    - The visitation order of a level-order traversal:

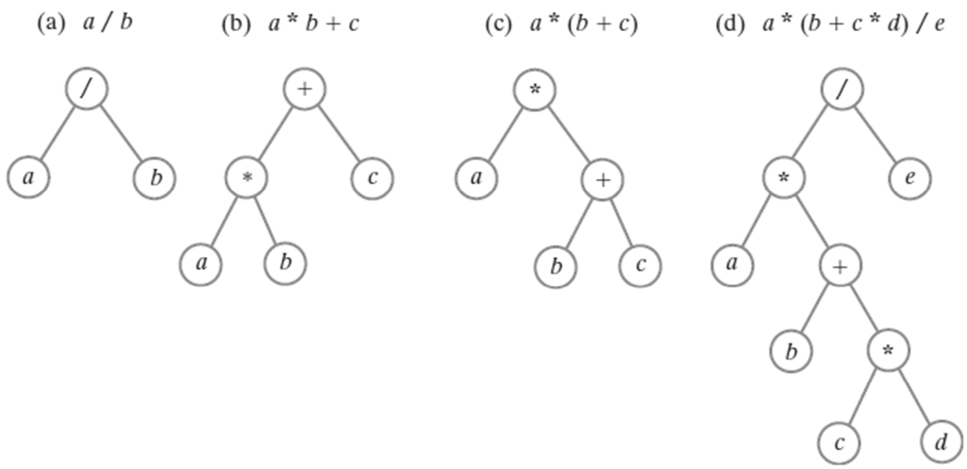
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
  a) print temp_node->data.
  b) Enqueue temp_node's children (first left then right children) to q
  c) Dequeue a node from q and assign it's value to temp_node
	

- Level-order traversal is implemented via a **queue**.
- The traversal is a breadth-first search.

HW: implement level-order traversal

Expression Trees

- The leaves of an expression tree are **operands**, such as **constants** or **variable** names, and the other nodes contain **operators**.
- It is also possible for a node to have only one child, as is the case with the **unary minus** operator.
- We can evaluate an expression tree by applying the **operator** at the **root** to the values obtained by **recursively** evaluating the **left** and **right** subtrees.



Algorithm for evaluation of an expression tree:

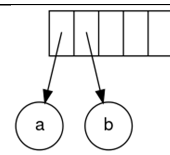
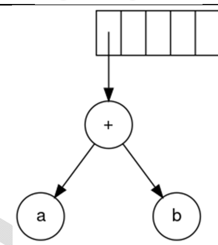
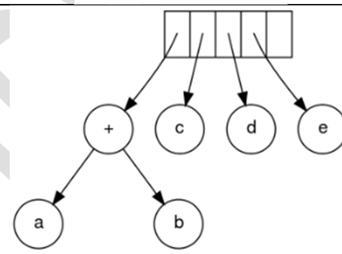
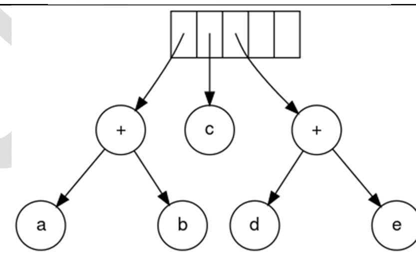
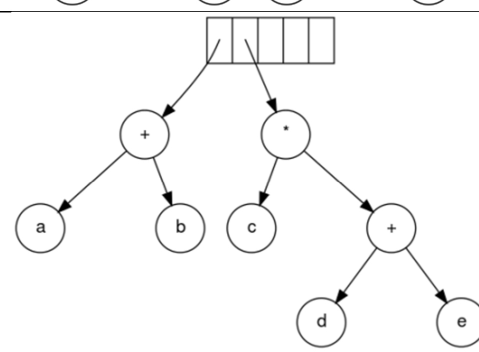
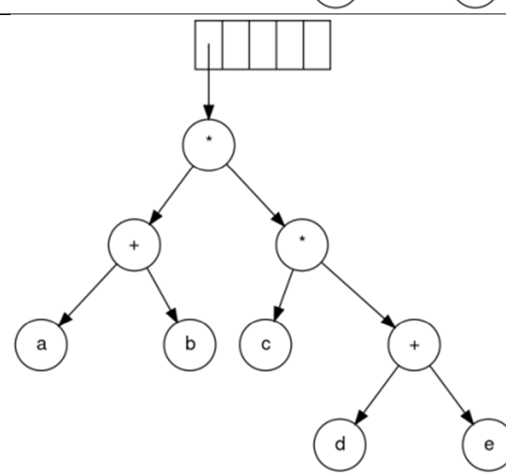
```
Algorithm evaluate(expressionTree)
if (expressionTree is empty)
    return 0
else
{
    firstOperand = evaluate(left subtree of expressionTree)
    secondOperand = evaluate(right subtree of expressionTree)
    operator = the root of expressionTree
    return the result of the operation operator and its operands firstOperand
           and secondOperand
}
```

Constructing an expression tree:

The construction of the expression tree takes place by reading the **postfix expression** one symbol at a time:

- If the symbol is an **operand**, one-node tree is created and a pointer is pushed onto a **stack**.
- If the symbol is an **operator**,
  - Two pointers trees **T1** and **T2** are popped from the stack
  - A new tree whose root is the **operator** and whose **left** and **right** children point to **T2** and **T1** respectively is formed .
  - A pointer to this new tree is then pushed to the Stack.

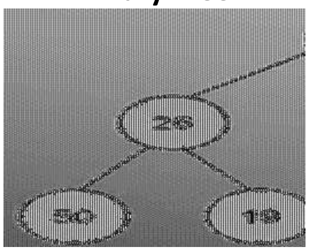
Example: (a b + c d e + * *)

• Since the first two symbols are operands, one-node trees are created and pointers are pushed to them onto a stack.	
• The next symbol is a '+'. It pops two pointers, a new tree is formed, and a pointer to it is pushed onto to the stack.	
• Next, **c**, **d**, and **e** are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.	
• Continuing, a '+' is read, and it merges the last two trees.	
• Now, a '*' is read. The last two tree pointers are popped and a new tree is formed with a '*' as the root.	
• Finally, the last symbol is read. The two trees are merged and a pointer to the final tree remains on the stack.	

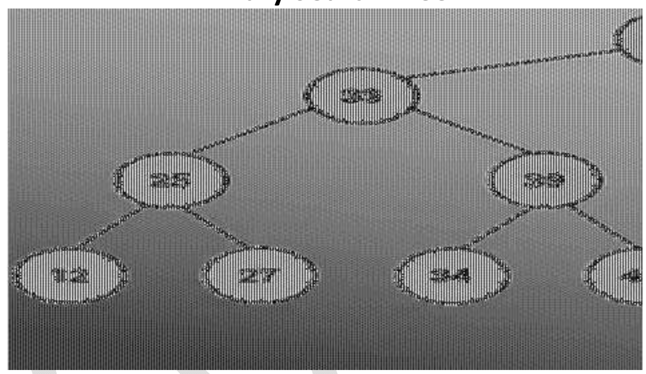
Binary Search Trees (BST)

- **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**.

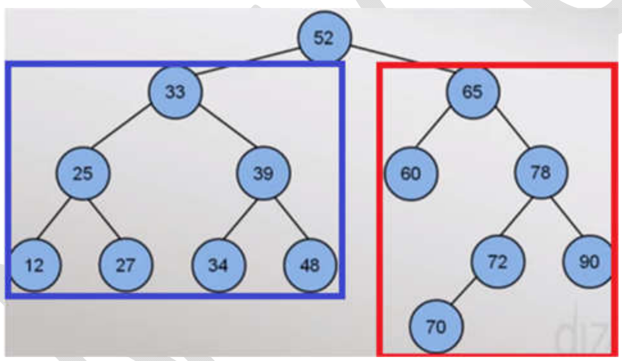
Binary Tree



Binary Search Tree



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



- **Search for an item:**

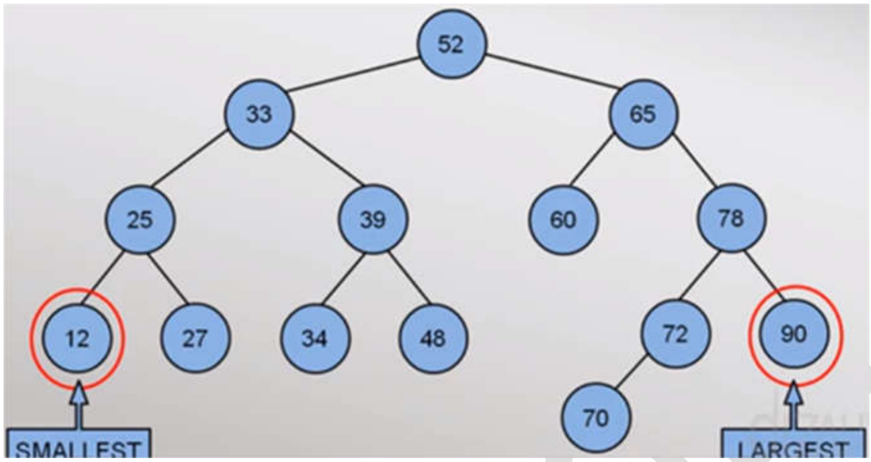
Example: find(52) , find(39) , find(35)

```
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 **short** as possible.

Finding Max and Min Values:

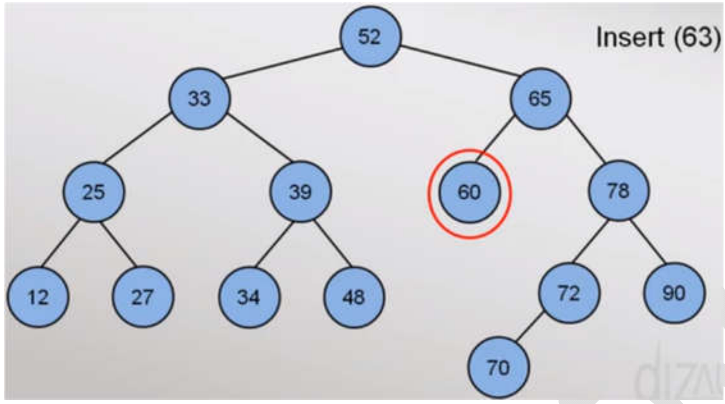


- The find **Min** operation is performed by following left nodes as long as there is a **left** child.
- The find **Max** operation is similar.

```
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;
}
```

Insert in Binary Search Tree:

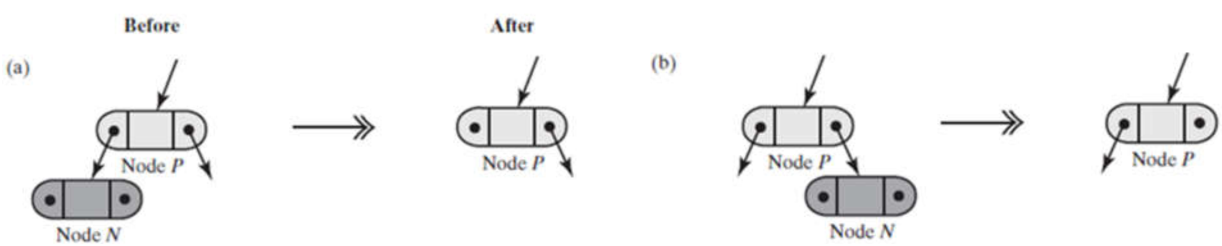
Example: insert(63)



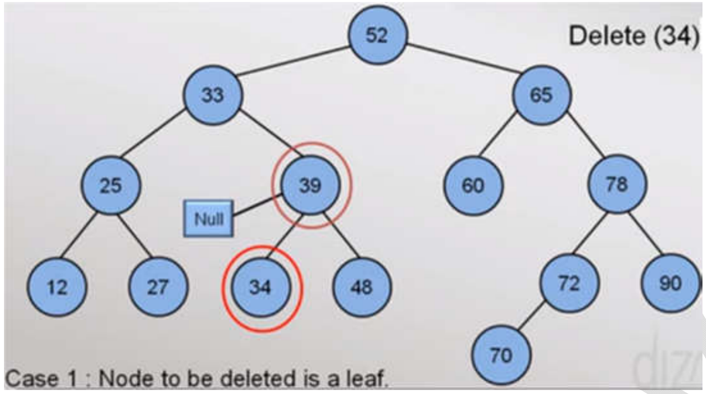

```java
public void insert(T data) {
   if (isEmpty())
      root = new TNode(data);
   else
      insert(data, root);
}
public void insert(T data, TNode node) {
   if (data.compareTo((T) node.data) >= 0) { // insert into right subtree
      if (!node.hasRight())
         node.right = new TNode(data);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

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:

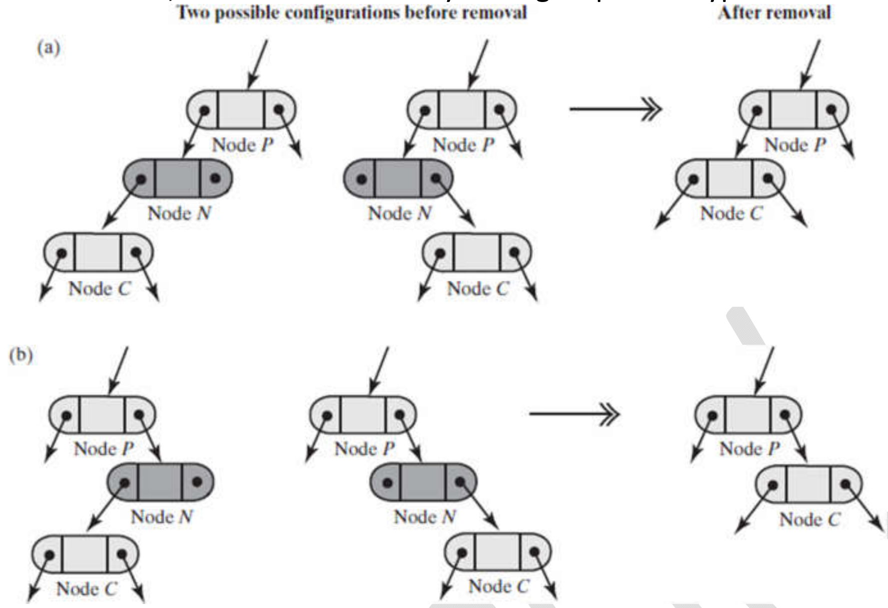


Example: delete(34)

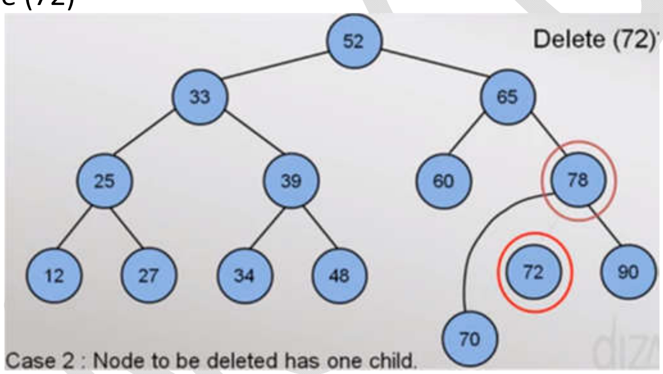



```java
public TNode delete(T data) {
   TNode current = root;
   TNode parent =  root;
   boolean isLeftChild = false;

   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)

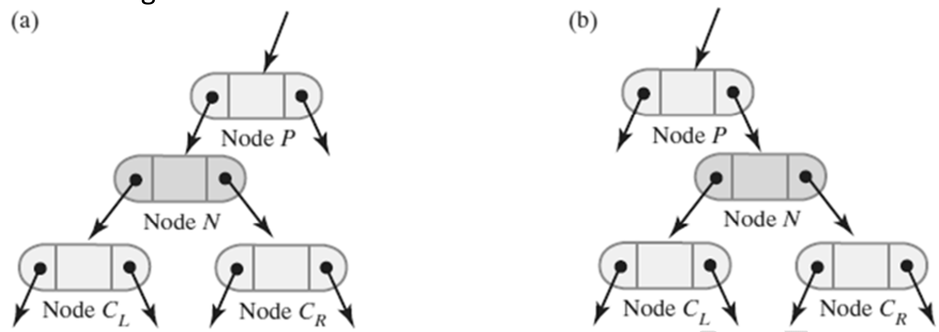


Note: The **root** is a special case because it does not have a parent.

```java
// 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;
   }
}
```

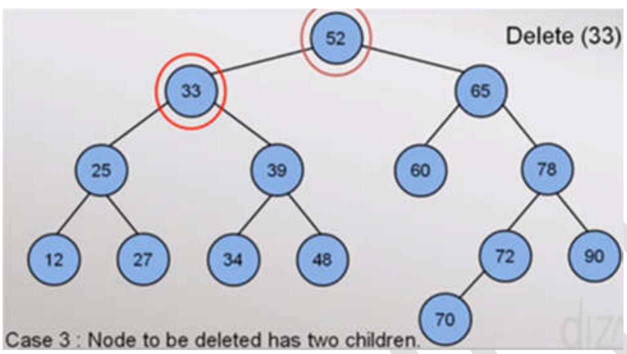
Case 3:

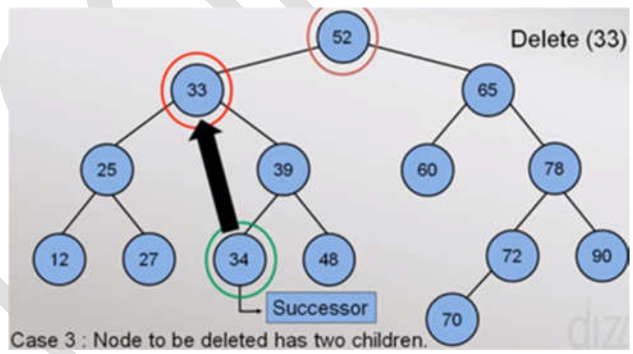
- Two possible configurations of a node N that has two children:

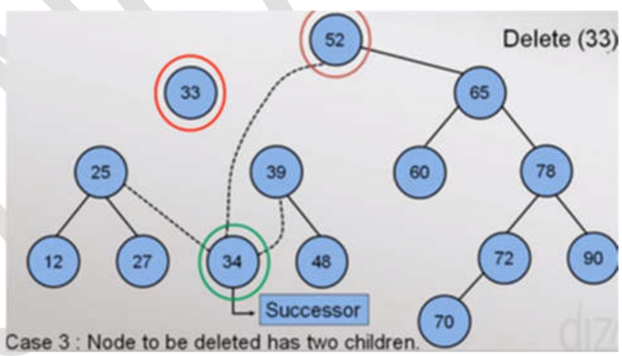


- A node with two children is replaced by using the **smallest** item in the right subtree (**Successor**).

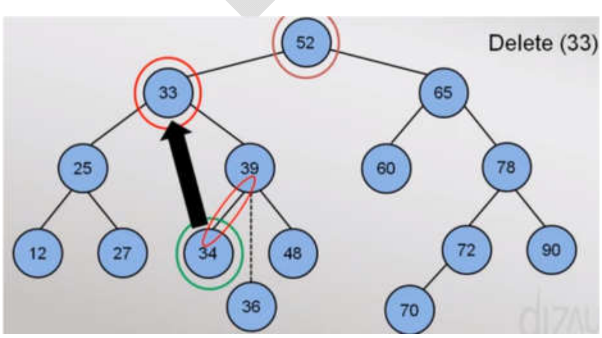
Example: delete(33)



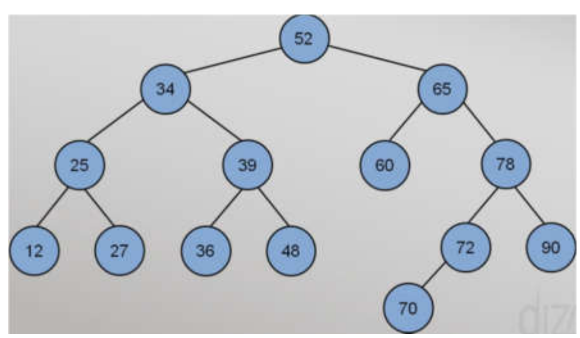




What if node **34** has a right child (e.g. **36**)?



➔



```
// 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.

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);
}
```

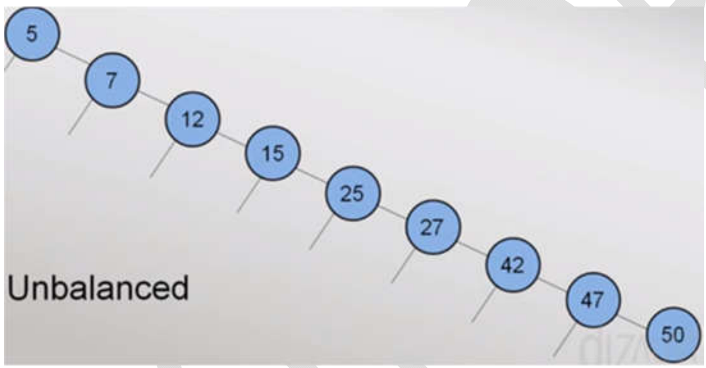
Efficiency of Operations:

- 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)**

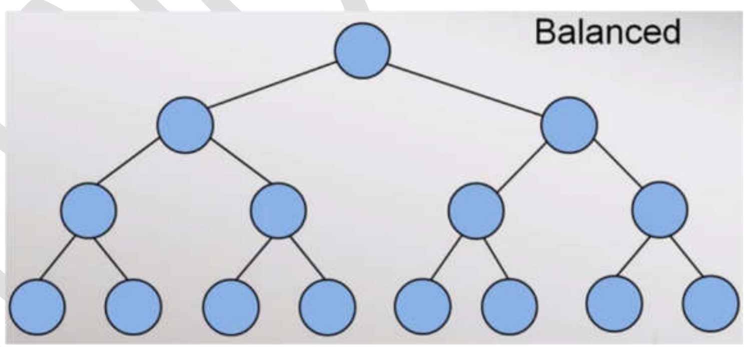
Unbalanced Tree:

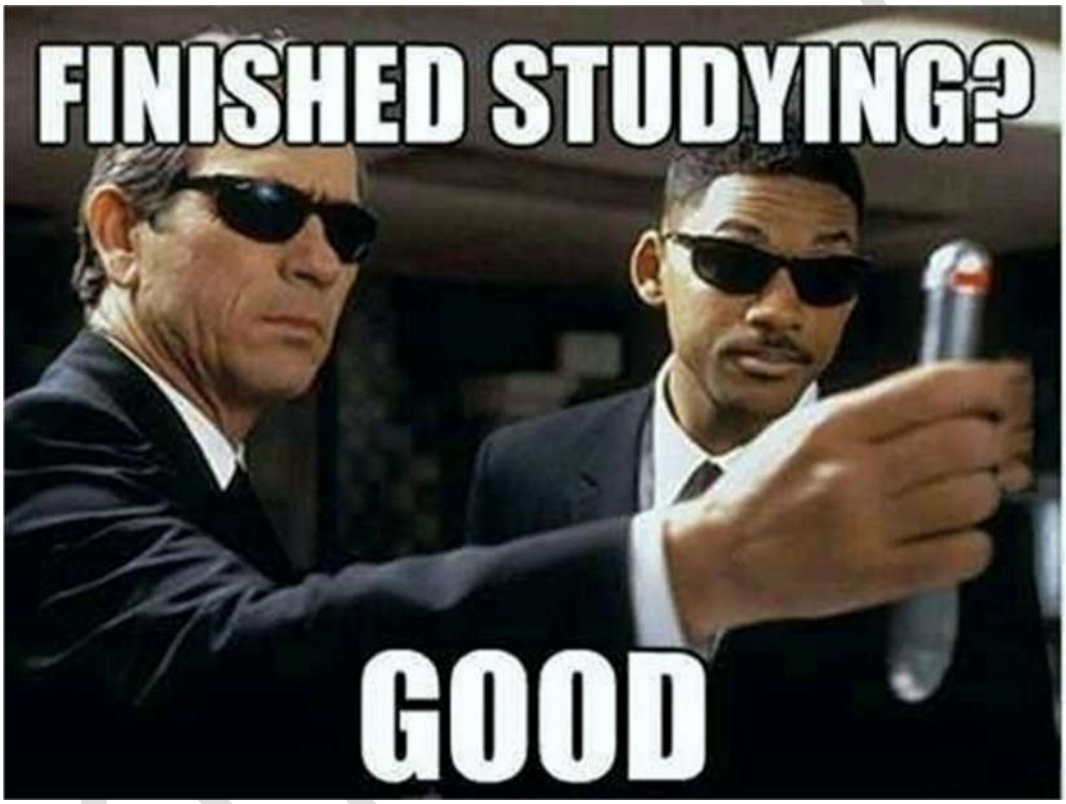
- The order in which you add entries to a binary search tree affects the shape of the tree.

  Example: add 5, 7, 12, 15, 25, 27, 42, 47, 50



- If you add entries into an initially empty binary search tree, do not add them in sorted order.




FINISHED STUDYING?
GOOD