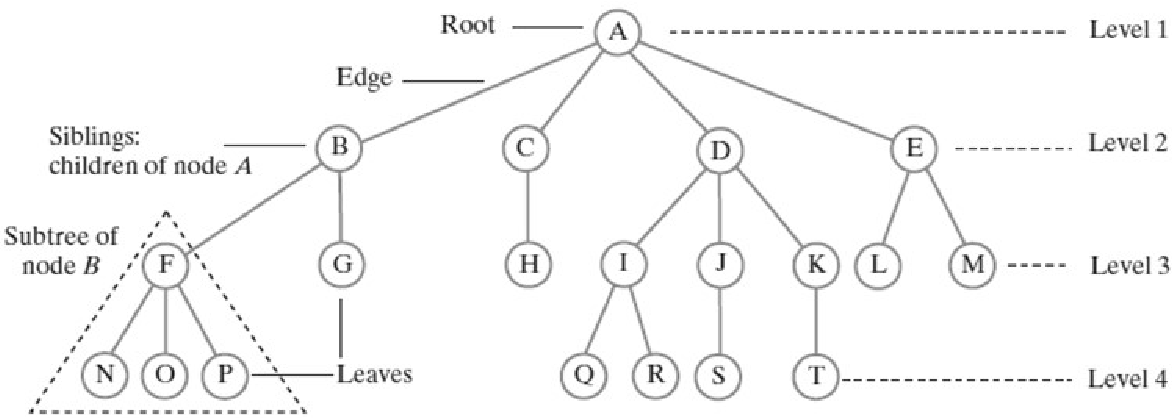
Trees

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)

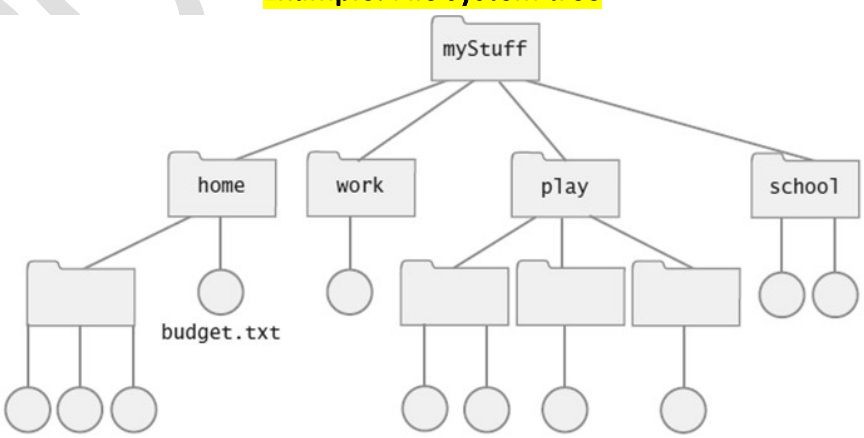
Tree



- 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.

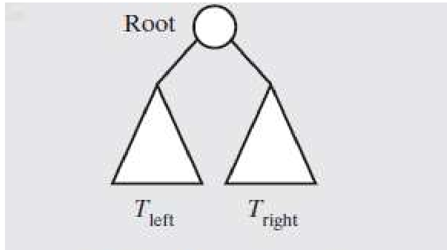
Example: Family Trees (one parent)

Example: File system tree



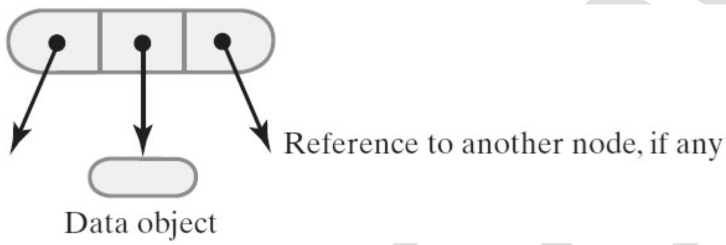
Binary Trees

- 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**:

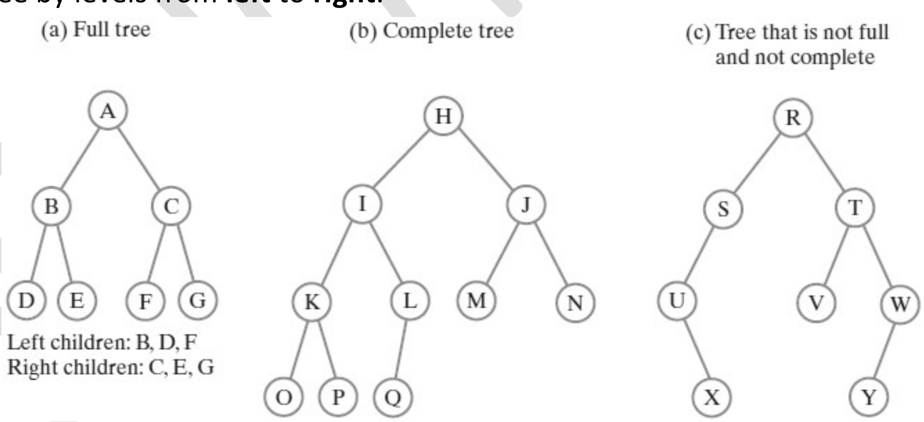


Full Binary tree: Each node in a **full binary tree** is either:

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

(2) A leaf.

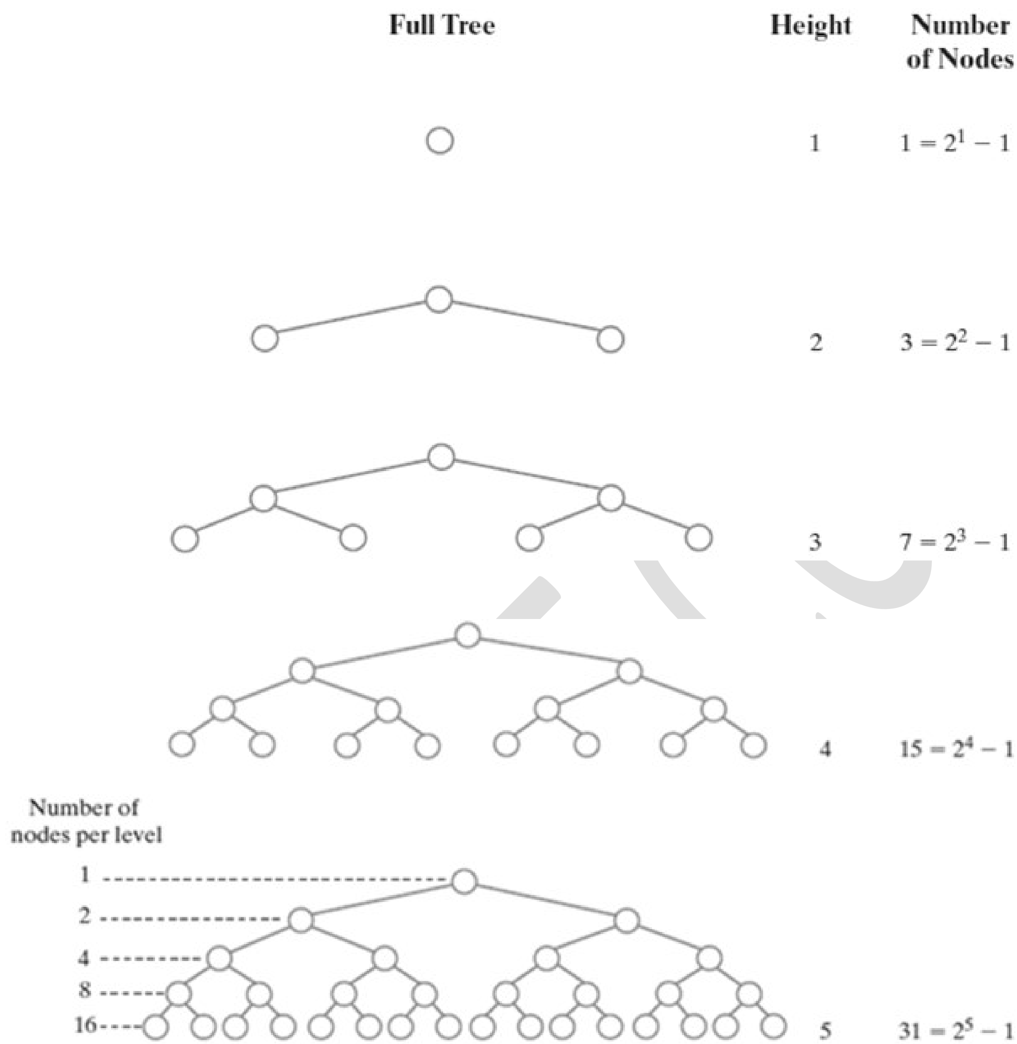
Complete binary tree: A **complete binary tree** has a restricted shape obtained by starting at the root and filling the tree by levels from **left** to **right**.



(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**



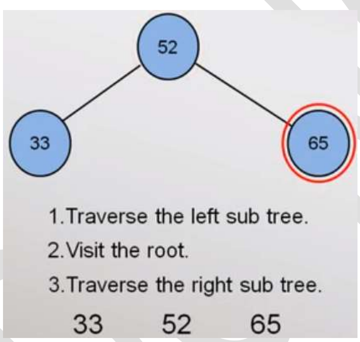
Implementation:

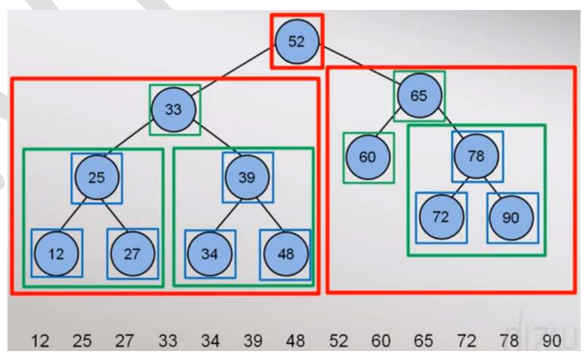
```
public class TNode<T extends Comparable<T>> {
   T data;
   TNode left;
   TNode right;

   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.

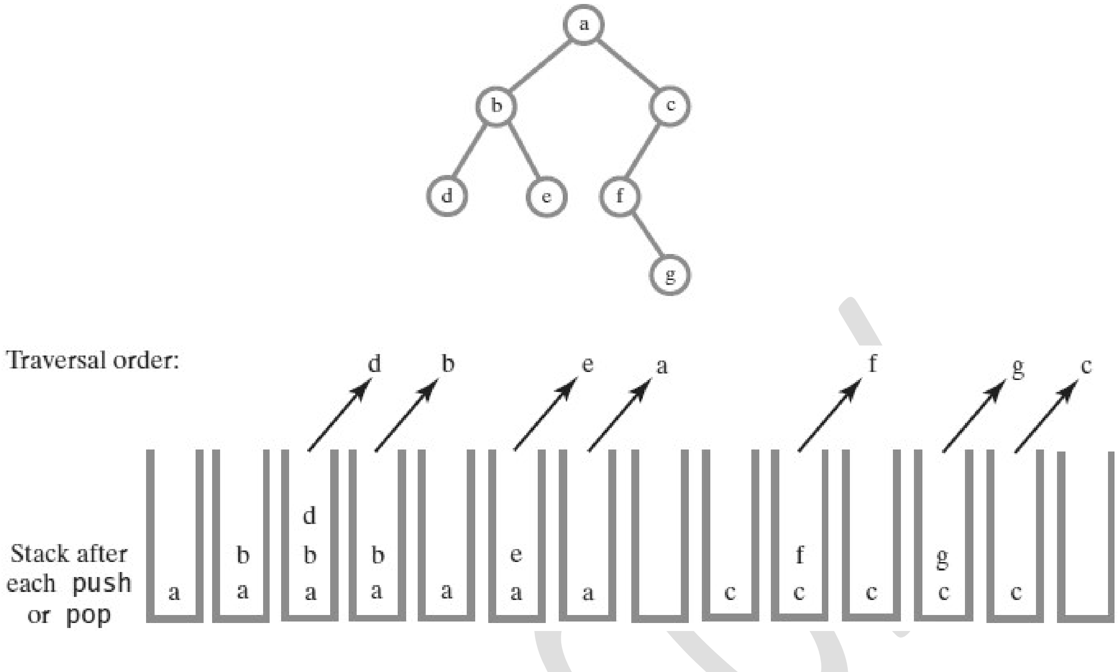




  - **Recursive implementation:**

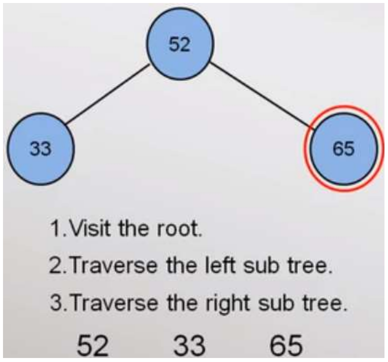
```
public void traverseInOrder() {  traverseInOrder(root);   }
private void traverseInOrder(TNode node) {
   if (node != null) {
      if (node.left != null)
         traverseInOrder(node.left);
      System.out.print(node + " ");
      if (node.right != null)
         traverseInOrder(node.right);
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

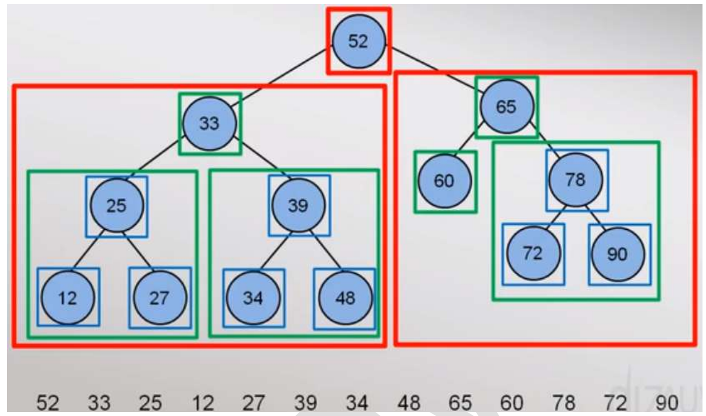
- **Using a stack to perform an in-order traversal iteratively: (Optional)**



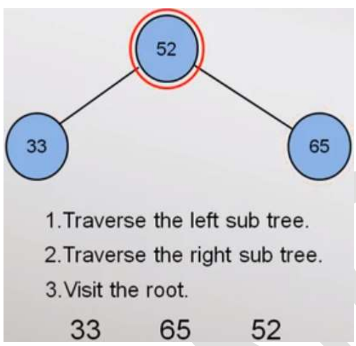

```java
public void iterativeInorderTraverse()
{
   StackInterface<BinaryNodeInterface<T>> nodeStack = new LinkedStack<>();
   BinaryNode<T> currentNode = root;

   while (!nodeStack.isEmpty() || (currentNode != null))
   {
      // Find leftmost node with no left child
      while (currentNode != null)
      {
         nodeStack.push(currentNode);
         currentNode = currentNode.getLeftChild();
      } // end while

      // Visit leftmost node, then traverse its right subtree
      if (!nodeStack.isEmpty())
      {
         BinaryNode<T> nextNode = nodeStack.pop();
         assert nextNode != null;  // Since nodeStack was not empty
                                   // before the pop

         System.out.println(nextNode.getData());
         currentNode = nextNode.getRightChild();
      } // end if
   } // end while
} // end iterativeInorderTraverse
```

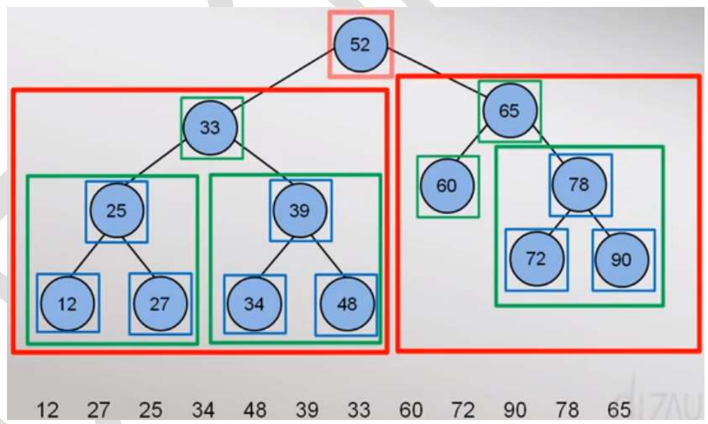
- **Pre-Order Traversal:** Visit **root** before we visit root's subtrees.





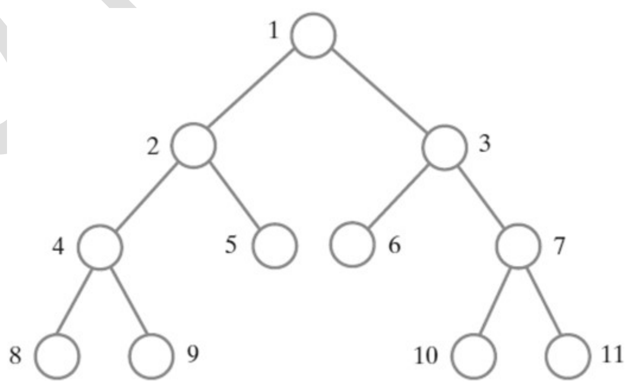
- **Post-Order Traversal:** Visit **root** of a binary tree after visiting nodes in root's subtrees.





- **Level-Order Traversal:** Begin at **root** and visit nodes one level at a time.

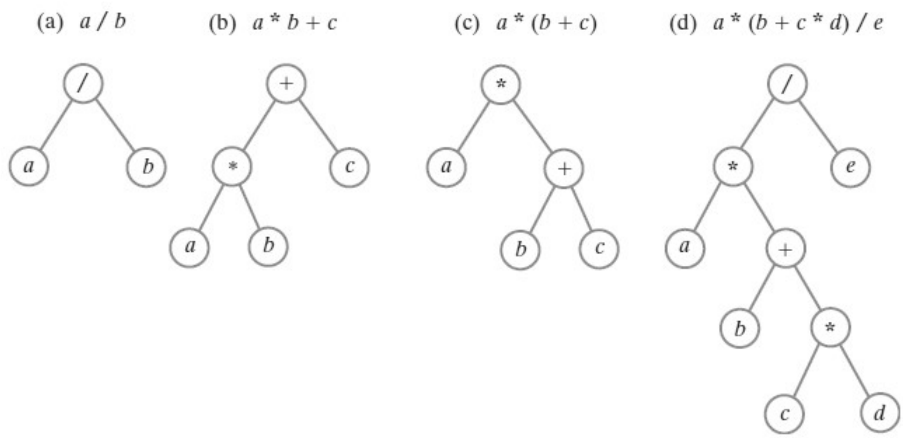
  - The visitation order of a level-order traversal:



  - 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:

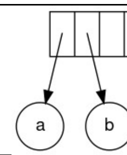
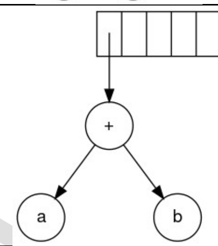
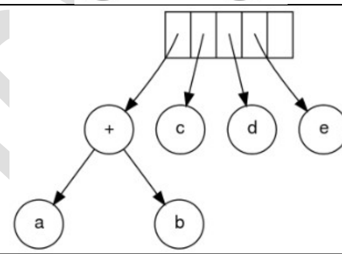
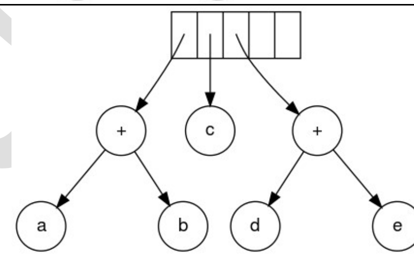
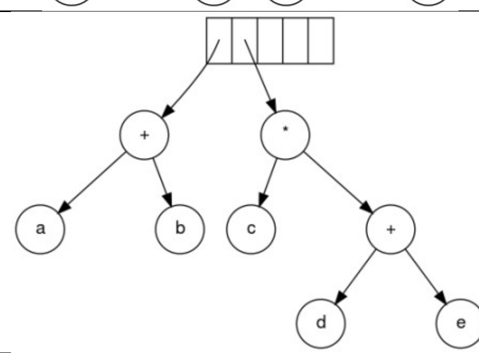
```
Algorithm evaluate(expressionTree)
if (expressionTree is empty)
    return 0
else
{
    firstOperand = evaluate(left subtree of expressionTree)
    secondOperand = evaluate(right subtree of expressionTree)
    operator = the root of expressionTree
    return the result of the operation operator and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