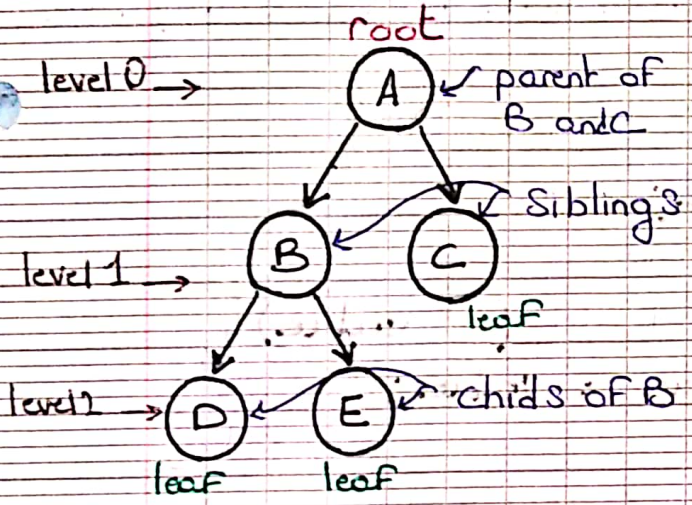
Binary Trees

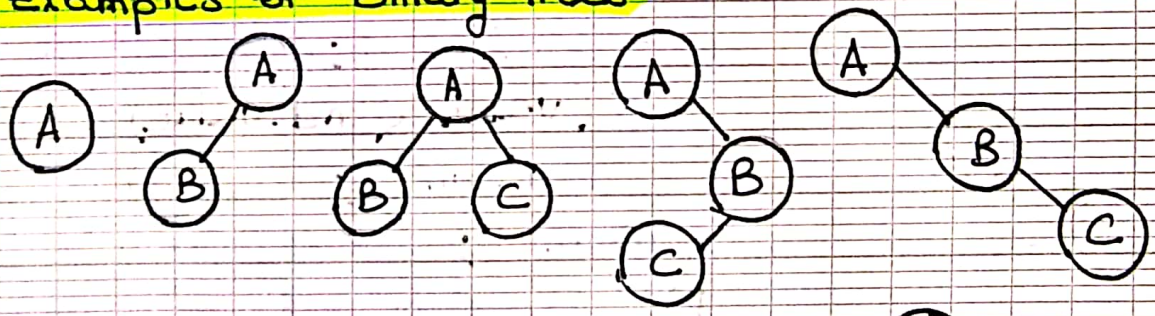
→ What is a Binary Tree

A binary tree is a hierarchical data structure composed of nodes where each node has at most two children / subtrees



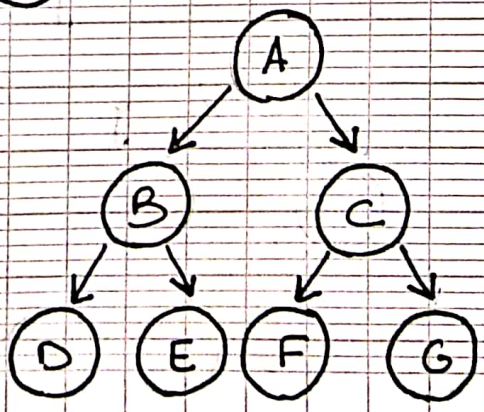
- **degree:** number of subtrees of the node
- **leaf:** node with no children
- **root:** node with no parent
- **branch:** non leaf node
- **moment:** nb of nodes in the tree
- **weight:** nb of leaves in the tree
- **height:** nb of levels in the tree

→ Examples of Binary Trees



→ Complete Binary Tree

A complete Binary Tree is a binary tree where each node, except the leaves, has exactly two children / subtrees



→ Implementation of Binary Tree in Java

To represent a Binary Tree we should write 3 classes: NodeData, TreeNode, and BinaryTree

1) NodeData Class

This class represent the data that will be stored in each node of the binary tree

Code:

```
public class NodeData {
    int value;
    // Constructor
    public NodeData (int value) {
        this.value = value;
    }
    // Compare 2 NodeData objects
    public int compareTo (NodeData nd) {
        if (value == nd.value)
            return 0;
        if (value < nd.value)
            return -1;
        return 1;
    }
}
```

2) TreeNode Class

- This class represents the nodes of the binary tree. Each node contains a 'NodeData' object, and links to its left and right children

code:

```
public class TreeNode {
    NodeData data;
    TreeNode left, right;
    // Constructor
    public TreeNode (NodeData nd) {
        data = nd;
        left = right = null;
    }
}
```

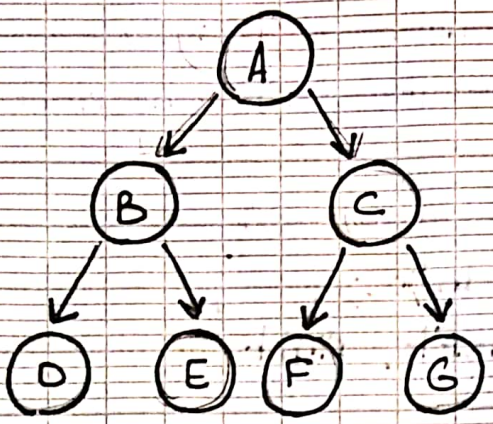
3) Binary Tree Class (or: BST class)

- This class is responsible for managing the overall structure of the binary Tree.
- The constructor initializes the root of the tree.

code:

```
public class BST {
    TreeNode root;
    //Constructor
    public BST() {
        root = null;
    }
    // Check if the tree is empty
    public boolean isEmpty() {
        return root == null;
    }
    ... more methods
}
```

→ Traversing a Binary Tree



Considering the following binary Tree, we can traverse it (visit it) in 3 different ways: pre_order, in-order, and postorder.

↳ Pre_order traversal (NLR)

- pre_order traversal involves visiting the root first, then recursively traversing the left subtree and finally traversing the right subtree.
  (root → left → right)
  => output: A B D E C F G
- Code: (in the BST class)

```
public void preOrder() {
    preOrderTraversal(root);
}
private void preOrderTraversal(TreeNode current){
    if (current != null) {
        System.out.println(current.data + " ");//N
        preOrderTraversal(current.left); //L
        preOrderTraversal(current.right); //R
    }
}
```

In_order Traversal (LNR)

- In-order traversal involves visiting the left subtree, then the current node and finally the right subtree
  (left → root → right)
  => output: D B E A F C G

- Code:

```
public void inOrder(){
    inOrderTraversal(root);
}
private void inOrderTraversal(TreeNode current){
    if(current != null){
        inOrderTraversal(current.left); // L
        System.out.println(current.data + " "); // N
        inOrderTraversal(current.right); // R
    }
}
```

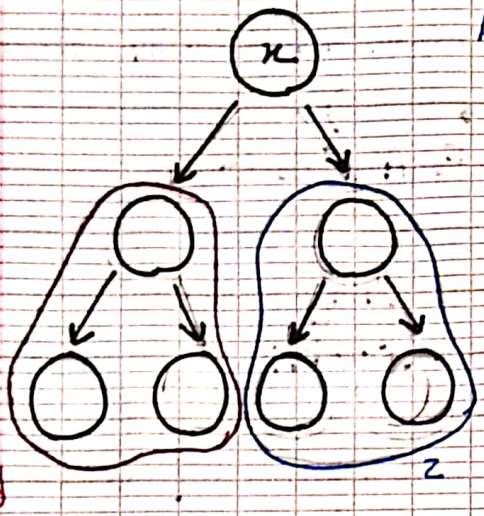
Post_Order Traversal (LRN)

- Post-order traversal involves visiting the left subtree, then the right subtree, and finally the root
  (left → right → root)
  => output: D E B F G C A

- Code

```
private void postOrder(){
    postOrderTraversal(root);
}
public void postOrderTraversal(TreeNode current){
    if(current != null){
        inorderTraversal(current.left); // L
        inorderTraversal(current.right); // R
    } System.out.println(current.data + " "); // N
}
```

→ Binary Search Tree



A Binary Search Tree follows some order to arrange the elements. In a Binary Search Tree, the value of the left node must be smaller than parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.

=> In the binary search tree above we must have $y < x$ and $z > x$.

↳ Searching in BST

Approach:

1. Compare the element to be searched with the root.
2. If the root is equal to the target element, then return true.
3. If the target value is less than the root, recursively search in the left subtree.
4. If the target value is greater than the root, recursively search in the right subtree.
5. Continue these steps until a match is found or a leaf node (null) is reached, indicating that the target value is not present in the tree.

• Code:

```
public class Search (NodeData item){
    TreeNode current = root;
    // Check if the tree is empty
    if ( isEmpty ()) {
        System.out.println ("Operation failed : BST empty");
    }else {
        while ( current != null) {
            if (item.compareTo (current.data) == 0)
                return true;
            else if (item.compareTo (current.data < 0)
                current = current.left;
            else
                current = current.right
        }
    }
    return false;
}
```

↳ Inserting an element into a BST

• approach:

1. Check if the tree is empty: in this case insert the new item as the root.
2. Search or Insert in a Non-Empty Tree:
   - If the tree is not empty, start from the root and traverse down the tree
   - while traversing: if the item is equal to the current

node's data, it's already in the tree. Print a message and stop.

- If the item is less than the current node's data move to the left child.
- If the item is greater than the current node's data move to the right child.

3. If the item is not found during traversal create a new node with the item's data. Decide whether to attach the new node to the left or right of the last visited node based on the comparison.

→ Code:

```
public void findOrInsert (NodeData item){
    TreeNode current = root, parent = null, newNode;
    boolean found = false;
    // Check if the tree is empty
    if (isEmpty())
        root = new TreeNode (item);
    // Search or Insert in a Non Empty Tree
    else {
        while (current != null){
            if (item.CompareTo (current.data) == 0){
                System.out.println ("The item is already in the BST");
                found = true;
                break;
            }
```

```
        else if (item.compareTo(current.data) < 0) {
                parent = current;
                current = current.left;
        }
        else {
                parent = current;
                current = current.right;
        }
    } // end while

    // If the item is not found
    if (!found) {
        newNode = new TreeNode(item);
        if (item.compareTo(parent.data) < 0) {
            parent.left = newNode
        }
        else {
            parent.right = newNode;
        }
    } // end else
}
```

→ Level-Order Traversal

- Level Order Traversal is an algorithm that visits the nodes of a binary tree level by level, starting from the root and moving to the next level left to right.
- The level Order Traversal can be implemented using a queue to keep track of the nodes at each level

Approach:

1. Create an empty queue of a TreeNode Object
2. Check if the tree is empty: in this case exit
3. If not, Enqueue the root node into the queue
4. Iterate Until the queue is Empty:
   - while queue is not empty
     - Dequeue a node from the front of the queue
     - Process the dequeue node (print its value)
     - Enqueue its left and right children (if any) into the queue

   Continue until the tree become empty

Code:

```
public void levelOrder() {
 Queue queue = new Queue();
 TreeNode current;
 // Check if the tree is empty
 if (root == null) [
     System.out.println("The tree is empty");
     System.exit(1);
 }
```

```
// If the tree is not empty
    queue.enqueue(root);
    while (!queue.isEmpty()) {
        current = queue.dequeue();
        System.out.println(current.data.value + " ");
        if (current.left != null)
            queue.enqueue(current.left);
        if (current.right != null)
            queue.enqueue(current.right);
    }
}
```

→ **Non-recursive Traversal**

Non-recursive tree traversal involves visiting each node in a binary tree without using recursion.

Approach (For In-Order)

1. Start from the root and initialize an empty stack
2. While the current node is not null or the stack is not empty:
   - Traverse down the left subtree by pushing nodes onto the stack
   - pop a node from the stack and move to its right subtree
   - Repeat until all nodes are visited

. Code:

```
public void nonRecursiveInorder() {
    Stack s = new Stack();
    TreeNode current = root;
    boolean stop = false;

    while (!stop) {
        while (current != null) {
            s.push(current);
            current = current.left;
        }
        if (s.empty())
            stop = true;
        else {
            current = s.pop();
            System.out.println(current.data.value + " ");
            current = current.right;
        }
    }
}
```

→ Some Useful Tree functions

↳ Count the nb of nodes in a tree

```
public int numNodes () {
    return countNodes (root);
}

private int countNodes (TreeNode current) {
    if (current == null)
        return 0;
    return 1 + countNodes (current.left) + countNodes (current.right);
}
```

↳ Count the nb of leaves in the tree.

```
public int numLeaves () {
    return countLeaves (root);
}

public int countLeaves (TreeNode current) {
    if (current == null)
        return 0;
    if (current.left == null && current.right == null)
        return 1;
    return countLeaves (current.left) + countLeaves (current.right);
}
```

→ Calculate the height (nb of levels) of the tree

```
public int height(){
    return countLevels(root);
}
public int countLevels(TreeNode current){
    if(current == null)
        return 0;
    return 1 + Math.max(countLevels(current.left),
                        countLevels(current.right));
}
```

→ BST Deletion

The method delete(NodeData item) is used to delete an item from the binary search tree if it is found

Approach:

1. Initialize variables including pointers for the parent, child, and current node, and a boolean flag to track if the item is found
2. Check if the BST is empty. If so print an error message
3. Use a loop to search the node containing the specified item
4. If the item is not found, print an error message
5. If the item is found handle 3 cases
   - If the node has two children, find the successor (smallest node in the right subtree), exchange its data with the current node, and then delete the successor
   - If the node has one child (left or right) or no children update the parent's link to bypass the current node and connect to its child
   - If the node to be deleted is the root, update the root to be the child

- Code:

```
public void delete (NodeData item) {
  TreeNode parent = null , child , temp, current = root;
  boolean found = false;
  //Check if the BST is empty
  if (root == null) {
    System.out.println ("The BST is empty");
    return ;
  }
  //If not loop to check if the element is already in the tree
  While ( current != null && !found) {
    if (item.compareTo(current.data) == 0)
      found = true;
    else if (item.compareTo(current.data < 0){
      parent = current ;
      current = current .left;
    }
    else {
      parent = current;
      current = current.right;
    }
  } // end while
  // if the item is not found
  if (!found)
    System.out.println("Element is not in the BST");
```

```
// If the item is found
else {
  // Case 1: two children
  if (current.left != null && current.right != null) {
    temp = current; // Keep a pointer to the node
                    containing the item
    // find the successor of the item
    parent = current;
    current = current.right;
    while (current.left != null) {
      parent = current;
      current = current.left;
    } // Current now point to the successor
    // Exchange the data of temp and that of current
    temp.data = current.data;
  } // end case 1
  // Case 2 : one child or no child
  child = current.left;
  if (child == null)
    child = current.right;
  if (current == root)
    root = child;
  else if (parent.left == current)
    parent.left = child;
  else
    parent.right = child;
  }
}
```