Traversal

```
import java.util.Queue;

import java.util.LinkedList;


public class BinaryTreeDemo {


    // Node class

    static class Node {

        String data;

        Node left, right;


        Node(String data) {

            this.data = data;

            this.left = this.right = null;

        }

    }


    // Build example tree:

    //        A

    //       / \

    //      B   C

    //     / \   \

    //    D   E   F

    private static Node buildExampleTree() {

        Node root = new Node("A");

        root.left  = new Node("B");

        root.right = new Node("C");
```

```
    root.left.left  = new Node("D");
    root.left.right = new Node("E");

    root.right.right = new Node("F");

    return root;
}

// Inorder (Left, Node, Right)
public static void inorder(Node node) {
    if (node == null) return;
    inorder(node.left);
    System.out.print(node.data + " ");
    inorder(node.right);
}

// Preorder (Node, Left, Right)
public static void preorder(Node node) {
    if (node == null) return;
    System.out.print(node.data + " ");
    preorder(node.left);
    preorder(node.right);
}

// Postorder (Left, Right, Node)
public static void postorder(Node node) {
    if (node == null) return;
    postorder(node.left);
    postorder(node.right);
```

```java
    System.out.print(node.data + " ");
}

// Breadth-First (Level-Order)
public static void breadthFirst(Node root) {
    if (root == null) return;
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        Node node = queue.remove();
        System.out.print(node.data + " ");
        if (node.left  != null) queue.add(node.left);
        if (node.right != null) queue.add(node.right);
    }
}

public static void main(String[] args) {
    Node root = buildExampleTree();

    System.out.print("Inorder traversal:       ");
    inorder(root);
    System.out.println();

    System.out.print("Preorder traversal:      ");
    preorder(root);
    System.out.println();

    System.out.print("Postorder traversal:     ");
    postorder(root);
```

```
        System.out.println();

        System.out.print("Depth-first (Preorder):   ");
        preorder(root);
        System.out.println();

        System.out.print("Breadth-first (Level-order): ");
        breadthFirst(root);
        System.out.println();
    }
}
```

Output

Inorder traversal: D B E A C F

Preorder traversal: A B D E C F

Postorder traversal: D E B F C A

Depth-first (Preorder): A B D E C F

Breadth-first (Level-order): A B C D E F

TreeDemo

```
// TreeInterface.java
public interface TreeInterface<T> {
    /** @return the data at the root of the tree (throws if empty) */
    T getRootData();

    /** @return the height of the tree (number of levels) */
    int getHeight();
```

```
    /** @return total count of nodes in the tree */
    int getNumberOfNodes();

    /** remove all nodes */
    void clear();

    /** @return true if the tree has no nodes */
    boolean isEmpty();
}

// BinaryTree.java
public class BinaryTree<T> implements TreeInterface<T> {
    private Node root;

    // --- node class ---
    private class Node {
        T      data;
        Node   left, right;
        Node(T data) { this(data, null, null); }
        Node(T data, Node left, Node right) {
            this.data  = data;
            this.left  = left;
            this.right = right;
        }
    }

    // --- constructors ---
    public BinaryTree() {
```

```java
    root = null;
}

/** create a leaf */
public BinaryTree(T rootData) {
    root = new Node(rootData);
}

/** create a tree whose root is rootData, with the given left/right subtrees */
public BinaryTree(T rootData, BinaryTree<T> leftTree, BinaryTree<T> rightTree) {
    Node leftNode  = (leftTree  == null ? null : leftTree.root);
    Node rightNode = (rightTree == null ? null : rightTree.root);
    root = new Node(rootData, leftNode, rightNode);
}

// --- TreeInterface methods ---

@Override
public T getRootData() {
    if (root == null)
        throw new IllegalStateException("Tree is empty");
    return root.data;
}

@Override
public boolean isEmpty() {
    return root == null;
}
```

```
@Override
public void clear() {
   root = null;
}

@Override
public int getHeight() {
   return computeHeight(root);
}

private int computeHeight(Node node) {
   if (node == null) return 0;
   int leftH  = computeHeight(node.left);
   int rightH = computeHeight(node.right);
   return 1 + Math.max(leftH, rightH);
}

@Override
public int getNumberOfNodes() {
   return countNodes(root);
}

private int countNodes(Node node) {
   if (node == null) return 0;
   return 1 + countNodes(node.left) + countNodes(node.right);
}

// — No add(...) or remove(...) here —
// Those belong only to specialized tree classes
```

```java
}

// TreeDemo.java
public class TreeDemo {
   public static void main(String[] args) {
      // Build the example tree from the slides:
      //        A
      //       / \
      //      B   C
      //     / \   \
      //    D   E   F

      BinaryTree<String> dTree = new BinaryTree<>("D");
      BinaryTree<String> eTree = new BinaryTree<>("E");
      BinaryTree<String> bTree = new BinaryTree<>("B", dTree, eTree);

      BinaryTree<String> fTree = new BinaryTree<>("F");
      BinaryTree<String> cTree = new BinaryTree<>("C", null, fTree);

      BinaryTree<String> aTree = new BinaryTree<>("A", bTree, cTree);

      // Demonstrate TreeInterface methods:
      System.out.println("Root data:       " + aTree.getRootData());
      System.out.println("Is empty?        " + aTree.isEmpty());
      System.out.println("Height:          " + aTree.getHeight());
      System.out.println("Number of nodes:  " + aTree.getNumberOfNodes());

      // Clear and test again
      aTree.clear();
```

```
        System.out.println("After clear...");
        System.out.println("  Is empty?      " + aTree.isEmpty());
    }
}
```

```
Root data:      A
Is empty?       false
Height:         3
Number of nodes:  6
After clear...
 Is empty?      true
```

Tree Example

```
public class BuildTreeExample {
    public static void main(String[] args) {
        // 1) leaves
        BinaryTreeInterface<String> dTree = new BinaryTree<>();
        dTree.setTree("D");
        BinaryTreeInterface<String> fTree = new BinaryTree<>();
        fTree.setTree("F");
        BinaryTreeInterface<String> gTree = new BinaryTree<>();
        gTree.setTree("G");
        BinaryTreeInterface<String> hTree = new BinaryTree<>();
        hTree.setTree("H");

        // 2) empty placeholder
        BinaryTreeInterface<String> emptyTree = new BinaryTree<>();
```

```
        // 3) subtrees
        BinaryTreeInterface<String> eTree = new BinaryTree<>();
        eTree.setTree("E", fTree, gTree);

        BinaryTreeInterface<String> bTree = new BinaryTree<>();
        bTree.setTree("B", dTree, eTree);

        BinaryTreeInterface<String> cTree = new BinaryTree<>();
        cTree.setTree("C", emptyTree, hTree);

        // 4) root
        BinaryTreeInterface<String> aTree = new BinaryTree<>();
        aTree.setTree("A", bTree, cTree);

        // (You can now traverse aTree, inspect getHeight(), etc.)
    }
}
```

Output

```
      A
    /   \
   B     C
  / \     \
 D   E     H
    / \
   F   G
```

Expression Evaluator

```
public class ExpressionEvaluator {

    // Node class for the expression tree
    static class Node {
        String data;
        Node left, right;

        // Leaf constructor
        Node(String data) {
            this(data, null, null);
        }

        // Internal node constructor
        Node(String data, Node left, Node right) {
            this.data = data;
            this.left  = left;
            this.right = right;
        }
    }

    // Recursively evaluates the expression tree
    public static double evaluate(Node node) {
        // Base case: leaf node (operand)
        if (node.left == null && node.right == null) {
            return Double.parseDouble(node.data);
        }
```

```
    // Recursive case: internal node (operator)
    double L = evaluate(node.left);
    double R = evaluate(node.right);
    switch (node.data) {
        case "+":
            return L + R;
        case "-":
            return L - R;
        case "*":
            return L * R;
        case "/":
            return L / R;
        default:
            throw new IllegalArgumentException("Unknown operator: " + node.data);
    }
}

public static void main(String[] args) {
    // Build an expression tree for: 3 + (4 * 5)
    Node three    = new Node("3");
    Node four     = new Node("4");
    Node five     = new Node("5");
    Node multiply = new Node("*", four, five);
    Node plus     = new Node("+", three, multiply);

    // Evaluate and print
    double result = evaluate(plus);
    System.out.println("Expression: 3 + (4 * 5)");
    System.out.println("Result: " + result);
```

```
   }
}
```

Output

Expression: 3 + (4 * 5)

Result: 23.0

Building Library Book Tree/PreInPost Traversal/Maze application on 15-node alphabet tree

```
import java.util.*;


public class Lab10Trees {


   /** Simple binary-tree node */
   static class Node {
      String data;
      Node left, right;


      Node(String data) {
         this(data, null, null);
      }
      Node(String data, Node left, Node right) {
         this.data  = data;
         this.left  = left;
         this.right = right;
```

```
    }
}

// --- Traversals ---
static void inorder(Node root) {
    if (root == null) return;
    inorder(root.left);
    System.out.print(root.data + " | ");
    inorder(root.right);
}

static void preorder(Node root) {
    if (root == null) return;
    System.out.print(root.data + " | ");
    preorder(root.left);
    preorder(root.right);
}

static void postorder(Node root) {
    if (root == null) return;
    postorder(root.left);
    postorder(root.right);
    System.out.print(root.data + " | ");
}

// --- Maze path finder ---
/** Finds a path from root to target; path is built in `stack` */
```

```java
static boolean findPath(Node node, String target, Deque<String> stack) {
    if (node == null) return false;
    stack.push(node.data);
    if (node.data.equals(target)) return true;
    if (findPath(node.left,  target, stack) ||
        findPath(node.right, target, stack)) {
        return true;
    }
    stack.pop();
    return false;
}

public static void main(String[] args) {
    // === 1) Library-books tree ===
    Node childSec    = new Node("Children's Section");
    Node selectSec   = new Node("Selected Books Section");
    Node firstFloor  = new Node("First Floor", childSec, selectSec);

    Node scienceSec  = new Node("Science Section");
    Node novelsSec   = new Node("Novels Section");
    Node historySec  = new Node("History Section");
    // Group novels & history under a dummy node (binary-tree workaround)
    Node otherSecs   = new Node("Other Sections", novelsSec, historySec);
    Node secondFloor = new Node("Second Floor", scienceSec, otherSecs);

    Node library     = new Node("Library", firstFloor, secondFloor);
```

```java
System.out.println("=== Library Tree Traversals ===");
System.out.print("Inorder:   ");  inorder(library);  System.out.println();
System.out.print("Preorder:  ");  preorder(library); System.out.println();
System.out.print("Postorder: ");  postorder(library);System.out.println();
System.out.println();

// === 2) Maze: a 15-node BST of letters A–Z ===
// Build leaves
Node A = new Node("A"), E = new Node("E");
Node G = new Node("G"), I = new Node("I");
Node N = new Node("N"), Q = new Node("Q");
Node W = new Node("W"), Z = new Node("Z");
// Build next level
Node C = new Node("C", A, E);
Node H = new Node("H", G, I);
Node P = new Node("P", N, Q);
Node Y = new Node("Y", W, Z);
// Build level above
Node F = new Node("F", C, H);
Node T = new Node("T", P, Y);
// Entrance of the maze
Node J = new Node("J", F, T);

System.out.println("=== Maze Paths ===");
String[] targets = { "A", "I", "N", "Q", "Z" };
for (String tgt : targets) {
  Deque<String> path = new ArrayDeque<>();
```

```
        if (findPath(J, tgt, path)) {
          // Stack has root→…→target, but in reverse (target on top)
          System.out.print("Path to " + tgt + ": ");
          while (!path.isEmpty()) {
            System.out.print(path.removeLast());
            if (!path.isEmpty()) System.out.print(" → ");
          }
          System.out.println();
        } else {
          System.out.println(tgt + " not found in maze.");
        }
      }
    }
}
```

Output

=== Library Tree Traversals ===

Inorder: Children's Section | First Floor | Selected Books Section | Library | Science Section | Second Floor | Novels Section | Other Sections | History Section |

Preorder: Library | First Floor | Children's Section | Selected Books Section | Second Floor | Science Section | Other Sections | Novels Section | History Section |

Postorder: Children's Section | Selected Books Section | First Floor | Science Section | Novels Section | History Section | Other Sections | Second Floor | Library |

=== Maze Paths ===

Path to A: J → F → C → A

Path to I: J → F → H → I

Path to N: J → T → P → N

Path to Q: J → T → P → Q

Path to Z: J → T → Y → Z

HeapSort

```
import java.util.ArrayList;

import java.util.Arrays;


public class MaxHeapDemo {

   /** A simple max-heap with 0-based indexing */

   static class MaxHeap {

      private ArrayList<Integer> heap;


      /** Constructs an empty heap */

      public MaxHeap() {

         heap = new ArrayList<>();

      }


      /** Builds a max-heap from the given array (bottom-up) */

      public MaxHeap(int[] arr) {

         heap = new ArrayList<>();

         for (int v : arr) {

            heap.add(v);

         }

         buildMaxHeap();

      }


      /** Restores max-heap order for the subtree rooted at i */
```

```
private void siftDown(int i, int heapSize) {
    while (true) {
        int left  = 2 * i + 1;
        int right = 2 * i + 2;
        int largest = i;

        if (left < heapSize && heap.get(left) > heap.get(largest)) {
            largest = left;
        }
        if (right < heapSize && heap.get(right) > heap.get(largest)) {
            largest = right;
        }
        if (largest != i) {
            swap(i, largest);
            i = largest;
        } else {
            break;
        }
    }
}

/** Converts the internal array into a valid max-heap in O(n) time */
public void buildMaxHeap() {
    int n = heap.size();
    // Last parent index = (n-2)/2
    for (int i = (n - 2) / 2; i >= 0; i--) {
        siftDown(i, n);
```

```
    }
}

/** Inserts a new key, restoring heap order by sifting up */
public void insert(int key) {
    heap.add(key);
    siftUp(heap.size() - 1);
}

/** Moves the element at i up until heap-order is restored */
private void siftUp(int i) {
    while (i > 0) {
        int parent = (i - 1) / 2;
        if (heap.get(i) > heap.get(parent)) {
            swap(i, parent);
            i = parent;
        } else {
            break;
        }
    }
}

/** Swaps two elements in the heap array */
private void swap(int i, int j) {
    int tmp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, tmp);
```

```
    }

    /** Prints the heap array in level order */
    public void printHeap() {
      System.out.println(heap);
    }
  }

  public static void main(String[] args) {
    int[] data = { 3, 1, 6, 5, 2, 4 };
    System.out.println("Original array:     " + Arrays.toString(data));

    // Build a max-heap from the array
    MaxHeap heap = new MaxHeap(data);
    System.out.print  ("After buildMaxHeap:  ");
    heap.printHeap();

    // Insert a new element
    heap.insert(10);
    System.out.print  ("After insert(10):    ");
    heap.printHeap();
  }
}
```

Output

Original array: [3, 1, 6, 5, 2, 4]

After buildMaxHeap: [6, 5, 4, 3, 2, 1]

After insert(10):   [10, 5, 6, 3, 2, 1, 4]