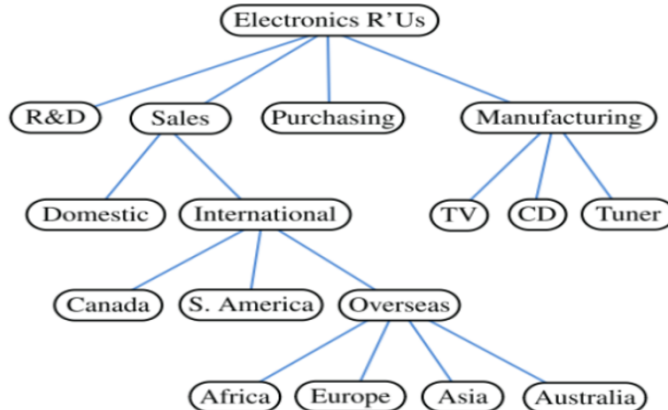


Laboratory-07

Experiment-Implement the given tree applications-

1. Write a program using a tree to display the tree in the following manner. Add the numbering and indentation with each depth



2. Consider the directory shown in the figure. Create node representing file information
 - name
 - type (0-file, 1-directory)
 - size
 - date

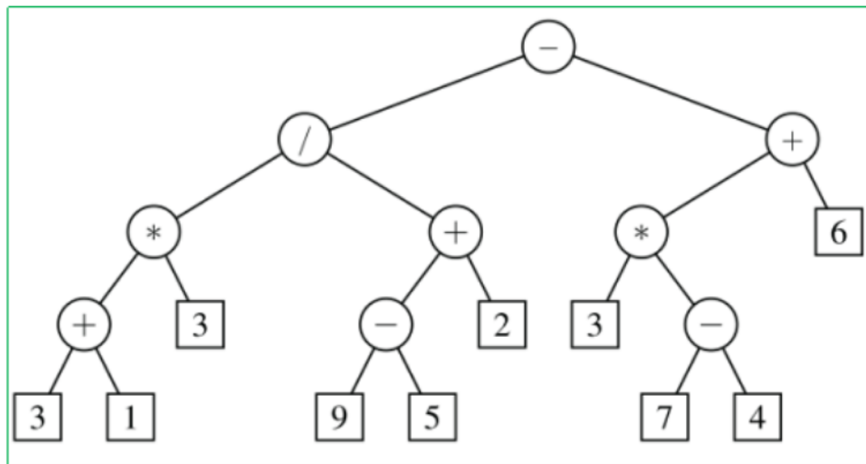
Write a program to display directory contents and its total disk space as shown in figure

```
E:\Department Course\2020-July-Dec-Data Structure Sem -III\PPT>dir
Volume in drive E has no label.
Volume Serial Number is 520E-7749

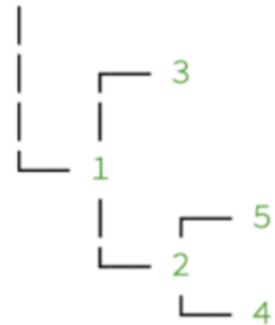
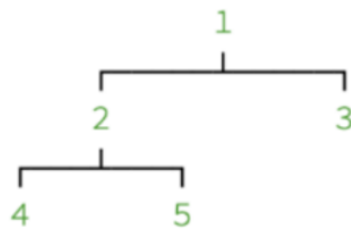
Directory of E:\Department Course\2020-July-Dec-Data Structure Sem -III\PPT

06-09-2020  12:10    <DIR>          .
06-09-2020  12:10    <DIR>          ..
08-08-2020  21:59    <DIR>          Godrich-book-slides
03-08-2020  05:45             58,978 Lecture-1- Overview.pptx
22-08-2020  20:17          403,557 Lecture-10- Circular Linked List.pptx
23-08-2020  05:27          774,294 Lecture-11-Recursion.pptx
26-08-2020  05:39          522,678 Lecture-12-Recursion Analysis.pptx
02-09-2020  09:45        1,101,808 Lecture-13-Positional Lists.pptx
29-08-2020  23:55        1,762,623 Lecture-14-Trees.pptx
05-09-2020  19:27        1,167,485 Lecture-15-Binary Trees.pptx
06-09-2020  11:07          772,218 Lecture-16-Trees Traversal.pptx
06-09-2020  12:10          312,615 Lecture-17-Trees Applications.pptx
03-08-2020  23:36          242,812 Lecture-2- OOPL-Java.pptx
10-08-2020  11:28          295,498 Lecture-3-Array Data Structure.pptx
10-08-2020  11:50          418,106 Lecture-4-Sorting Algorithms-1.pptx
11-08-2020  11:04          413,984 Lecture-5-AnalysisOfAlgorithm-1.pptx
12-08-2020  10:01          586,870 Lecture-6-AnalysisOfAlgorithm-2.pptx
17-08-2020  10:59          940,020 Lecture-7-Stack Data Structure.pptx
18-08-2020  10:54        1,192,213 Lecture-8-Queue Data Structure.pptx
19-08-2020  09:56          411,720 Lecture-9- Linked List.pptx
              17 File(s)          11,377,479 bytes
              3 Dir(s)        38,928,846,848 bytes free
```

3. Write a Java code to display arithmetic expression for a given tree as follows:
 $((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))$



4. Write a Java program to display the binary tree on console in any one of the shown methods using tree traversal method.



Java Constructs used-

Here are some Java constructs and concepts used in the provided code:

Classes and Objects:

- class: Defines a class (Information, Application2, Application3, Application4, ExtendedTree, BinaryTree etc).
- new: Instantiates objects of classes.

File Handling:

- File: Represents a file or directory path.
- File() constructor: Creates a new File instance.

Data Types:

- String: Represents sequences of characters.
- boolean: Represents a boolean value.
- long: Represents a 64-bit integer.

Data Structures:

- GeneralTree, LinkedBinaryTree, ExtendedTree: Represents different tree structures.
- TreeNode, Position: Nodes in the tree structure.

Generics:

- <T>: Generic type parameter used in class and method definitions.

Exception Handling:

- IllegalStateException: Thrown to indicate that a method has been invoked at an illegal or inappropriate time.

Date and Time Formatting:

- DateFormat, SimpleDateFormat: Used for formatting dates.

Input/Output:

- System.out.println(): Prints output to the console.

Looping:

- for loop: Used for iterating over collections.

Conditional Statements:

- if-else: Used for conditional branching.

Method Invocation:

- methodName(): Invoking methods to perform specific actions.

Recursion:

- Recursive methods (constructTree, printPreorderLabeled):

Methods that call themselves.

Inheritance:

- extends: Indicates that a class is intended to be a subclass of another class.

Interface Implementation:

- implements: Indicates that a class is implementing an interface.

Annotations:

- @Override: Indicates that a method is intended to override a method in a superclass.

Packages:

- java.io, java.text, java.util: Imported packages for input/output, text formatting, and utility classes.

Static Members:

- static: Indicates that a variable or method belongs to the class rather than instances of the class.

Collections:

- Iterable, ArrayList: Used for iterating over a collection of elements.

Lambda Expressions:

- Used implicitly in the for-each loop for iterating over collections.

Stack Usage:

- op.push(), op.pop(): Operations on a stack.

Overriding Methods:

- Methods like createNode, addRoot, diskSpace, etc., override methods from parent classes or interfaces.

These are some of the key Java constructs and concepts used in the provided code.

Code- Application1-

```
import java.util.*;
```

```
// Position interface representing a node in the tree
```

```
interface Position<E> {  
    E getElement();  
}
```

```
// Tree interface defining basic tree operations
```

```
interface Tree<E> extends Iterable<E> {  
    Position<E> root();//Returns position of root of the tree(null if empty)
```

```
    Position<E> parent(Position<E> p) throws IllegalArgumentException;//Returns position of  
    parent of position p of the tree(null if empty)
```

```
    Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;//Returns  
    iterable collection containing children at position p
```

```
    int numChildren(Position<E> p) throws IllegalArgumentException;//Returns no. of children at  
    position p
```

```
    boolean isInternal(Position<E> p) throws IllegalArgumentException;//Returns true if there is  
    atleast 1 child
```

```
    boolean isExternal(Position<E> p) throws IllegalArgumentException;//Returns true if there is  
    no child
```

```
    boolean isRoot(Position<E> p) throws IllegalArgumentException;//True if p is the root
```

```
    int size();//Return no. of positions and hence elements in a tree
```

```

boolean isEmpty();//Returns true if tree does not contain any positions

Iterator<E> iterator();//Returns iterator for all elements in the tree

Iterable<Position<E>> positions();//Returns iterable collection of all positions
}

// AbstractTree class providing a base implementation for common tree operations
abstract class AbstractTree<E> implements Tree<E> {

    // Common methods can be implemented here which are used for tree operations

    // Validate a position and convert it to TreeNode
    private TreeNode<E> validate(Position<E> p) throws IllegalArgumentException {
        if (!(p instanceof TreeNode)) {
            throw new IllegalArgumentException("Invalid position type");
        }
        TreeNode<E> node = (TreeNode<E>) p;
        if (node.getParent() == node) {
            throw new IllegalArgumentException("Position no longer in the tree");
        }
        return node;
    }

    // Abstract method to add a child node to a given parent position
    public abstract Position<E> addChild(Position<E> p, E element);

    // Abstract method to display the tree
    public abstract void displayTree();

    // Method to get siblings of a node
    public Iterable<Position<E>> siblings(Position<E> p) {
        // Get the parent of the given node
        Position<E> parent = parent(p);

        // Check if the node is the root or has no parent
        if (parent == null)
            return Collections.emptyList(); // p is the root, return an empty list
        else {
            // Get the children of the parent
            Iterable<Position<E>> siblings = children(parent);

            // Create and return a filtered iterable that excludes the original node
            List<Position<E>> filteredSiblings = new ArrayList<>();

```

```

        for (Position<E> sibling : siblings) {
            if (!sibling.equals(p)) {
                filteredSiblings.add(sibling);
            }
        }
        return filteredSiblings;
    }
}

```

```

// Method to list leaves of the tree
public List<Position<E>> leaves() {
    List<Position<E>> leafList = new ArrayList<>();
    for (Position<E> p : positions()) {
        if (isExternal(p)) {
            leafList.add(p);
        }
    }
    return leafList;
}

```

```

// Method to list internal nodes of the tree
public List<Position<E>> internalNodes() {
    List<Position<E>> internalNodesList = new ArrayList<>();
    for (Position<E> p : positions()) {
        if (isInternal(p)) {
            internalNodesList.add(p);
        }
    }
    return internalNodesList;
}

```

```

// Method to list edges of the tree
public List<Edge<E>> edges() {
    List<Edge<E>> edgeList = new ArrayList<>();
    for (Position<E> p : positions()) {
        for (Position<E> c : children(p)) {
            edgeList.add(new Edge<>(p, c));
        }
    }
    return edgeList;
}

```

```

// Method to get the path for a given node

```

```

public List<Position<E>> path(Position<E> p) {
    List<Position<E>> pathList = new ArrayList<>();
    while (p != null) {
        pathList.add(p);
        p = parent(p);
    }
    Collections.reverse(pathList);
    return pathList;
}

```

// Method to get the depth of a node

```

public int depth(Position<E> p) {
    if (isRoot(p))
        return 0;
    else
        return 1 + depth(parent(p));
}

```

// Method to get the height of the tree

```

public int height(Position<E> p) {
    int h = 0;
    for (Position<E> c : children(p)) {
        h = Math.max(h, 1 + height(c));
    }
    return h;
}

```

// Method to get a subtree rooted at a given node

//subTree method generates a new tree representing the subtree rooted at a given node

```

public Tree<E> subTree(Position<E> p) {
    TreeNode<E> rootNode = copySubTree(validate(p), null); //copySubTree method is a
    recursive helper method used to copy the structure of the subtree.
    MyTree<E> subtree = new MyTree<>(); //MyTree<E> object named subtree is created.
    subtree.root = rootNode; //rootNode is set as the root of the subtree.
    return subtree;
}

```

// Helper method to recursively copy the subtree

```

private TreeNode<E> copySubTree(Position<E> originalNode, TreeNode<E> parent) {
    // A new TreeNode called newNode with the same element as the originalNode and the
    specified parent.
    TreeNode<E> newNode = new TreeNode<>(originalNode.getElement(), parent);
    /// For each child, it recursively calls copySubTree to copy the child and its subtree.
    for (Position<E> child : children(originalNode)) {

```



```

        TreeNode<E> newChild = copySubTree(child, newNode);
        newNode.getChildren().add(newChild); //The newly created child node (newChild) is
        added to the list of children of the newNode.
    }

    return newNode;
}

}

```

```

// PositionalElement interface
interface PositionalElement {
    void setX(int x);
    void setY(int y);
}

```

```

class TreeNode<E> implements PositionalElement {
    private E element;
    private TreeNode<E> parent;
    private List<TreeNode<E>> children;
    private int x; // Added field for X-coordinate
    private int y; // Added field for Y-coordinate

    // Constructor for creating a tree node with an element and parent
    public TreeNode(E element, TreeNode<E> parent) {
        this.element = element;
        this.parent = parent;
        this.children = new ArrayList<>();
    }

    // Method to get the element of the node
    public E getElement() {
        return element;
    }

    // Method to get the parent of the node
    public TreeNode<E> getParent() {
        return parent;
    }

    // Method to get the children of the node
    public List<TreeNode<E>> getChildren() {
        return children;
    }
}

```

```

    }

    // Method to get the left child of the node
    public TreeNode<E> getLeftChild() {
        return children.isEmpty() ? null : children.get(0);
    }

    // Method to get the right child of the node
    public TreeNode<E> getRightChild() {
        return children.size() < 2 ? null : children.get(1);
    }

    // Method to get the X-coordinate
    public int getX() {
        return x;
    }

    // Method to set the X-coordinate
    public void setX(int x) {
        this.x = x;
    }

    // Method to get the Y-coordinate
    public int getY() {
        return y;
    }

    // Method to set the Y-coordinate
    public void setY(int y) {
        this.y = y;
    }
}

// Edge class representing a connection between two positions
class Edge<E> {
    private Position<E> source;
    private Position<E> target;

    // Constructor for creating an edge between two positions
    public Edge(Position<E> source, Position<E> target) {
        this.source = source;
        this.target = target;
    }
}

```

```

// Method to get the source position of the edge
public Position<E> getSource() {
    return source;
}

// Method to get the target position of the edge
public Position<E> getTarget() {
    return target;
}
}

```

MyTree.java

```

import java.util.*;

// MyTree class implementing the AbstractTree interface
public class MyTree<E> extends AbstractBinaryTree<E>{

    @Override
    public Position<E> left(Position<E> p) throws IllegalArgumentException {
        TreeNode<E> node = validate(p);
        return node.getLeftChild();
    }

    @Override
    public Position<E> right(Position<E> p) throws IllegalArgumentException {
        TreeNode<E> node = validate(p);
        return node.getRightChild();
    }

    // The root of the tree
    protected TreeNode<E> root;

    // Constructor for creating an empty tree
    public MyTree() {
        this.root = null;
    }

    // Implementing methods from the Tree interface

    // Returns the root position of the tree

```

```

public Position<E> root() {
    return root;
}

// Returns the root as a TreeNode
public TreeNode<E> getRoot() {
    return root;
}

// Returns the parent position of a given position
public Position<E> parent(Position<E> p) throws IllegalArgumentException {
    TreeNode<E> node = validate(p);
    return node.getParent();
}

// Returns the children of a given position
// Updated return type for children method
public Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException {
    TreeNode<E> node = validate(p);
    return new ArrayList<>(node.getChildren());
}

// Returns the number of children of a given position
public int numChildren(Position<E> p) throws IllegalArgumentException {
    TreeNode<E> node = validate(p);
    return node.getChildren().size();
}

// Checks if a given position is internal
public boolean isInternal(Position<E> p) throws IllegalArgumentException {
    return numChildren(p) > 0;
}

// Checks if a given position is external
public boolean isExternal(Position<E> p) throws IllegalArgumentException {
    return numChildren(p) == 0;
}

// Checks if a given position is the root of the tree
public boolean isRoot(Position<E> p) throws IllegalArgumentException {
    TreeNode<E> node = validate(p);
    return node == root;
}

```

```

// Returns the total number of positions in the tree
public int size() {
    return size(root);
}

// Recursive helper method to calculate the size of the subtree rooted at a given position
private int size(Position<E> p) {
    int count = 1; // count for the root
    for (Position<E> child : children(p)) {
        count += size(child);
    }
    return count;
}

// Checks if the tree is empty
public boolean isEmpty() {
    return size() == 0;
}

// Returns an iterator over the elements of the tree
public Iterator<E> iterator() {
    return new Treeliterator();
}

// Returns an iterable collection of all positions in the tree
public Iterable<Position<E>> positions() {
    List<Position<E>> positions = new ArrayList<>();
    collectPositions(root, positions);
    return positions;
}

// Iterator for the tree
private class Treeliterator implements Iterator<E> {
    private Iterator<Position<E>> positionIterator = positions().iterator();

    public boolean hasNext() {
        return positionIterator.hasNext();
    }

    public E next() {
        return positionIterator.next().getElement();
    }
}

```

```

// Add child node to a given parent position
public Position<E> addChild(Position<E> p, E element) {
    TreeNode<E> parent = validate(p);
    TreeNode<E> child = new TreeNode<>(element, parent);
    parent.getChildren().add(child);
    return child;
}

// Display the tree
public void displayTree() {
    displayTree(root, 0);
}

// Recursive method to display the tree structure
private void displayTree(Position<E> node, int depth) {
    if (node != null) {
        for (int i = 0; i < depth; i++) {
            System.out.print(" ");
        }
        System.out.println(node.getElement());
        for (Position<E> child : children(node)) {
            displayTree(child, depth + 1);
        }
    }
}

// Recursive method to collect positions in the tree
private void collectPositions(TreeNode<E> node, List<Position<E>> positions) {
    if (node != null) {
        positions.add(node);
        for (TreeNode<E> child : node.getChildren()) {
            collectPositions(child, positions);
        }
    }
}

// Helper method to validate a position
private TreeNode<E> validate(Position<E> p) throws IllegalArgumentException {
    if (!(p instanceof TreeNode)) {
        throw new IllegalArgumentException("Invalid position type");
    }
    TreeNode<E> node = (TreeNode<E>) p;
    if (node.getParent() == node) {
        throw new IllegalArgumentException("Position no longer in the tree");
    }
}

```

```

    }
    return node;
}
}

```

//Application1.java

```

import java.util.*;
public class Application1<T> extends MyTree<T> {

//Application 1 Printing Pre-Order indentation
    public void printPreorderLabeled(Application1<String> T, TreeNode<String> p,
ArrayList<Integer> path) {
        int d = path.size();
        for (int i = 1; i <= d; i++) {
            System.out.print(" ");
        }
        for (int j = 0; j < d; j++) System.out.print(path.get(j) + (j == d - 1 ? " " : "."));
        System.out.println(p.getElement());
        path.add(1);
        int childNumber = 1;
        for (TreeNode<String> c : p.getChildren()) {
            path.set(d, childNumber);
            printPreorderLabeled(T, c, path);
            childNumber++;
        }
        path.remove(d);
    }

    public void printIndentedTree(Application1<String> tree) {
        // tree is an instance of ExtendedTree<String> in TreeTestApplication
        printPreorderLabeled(tree, tree.root, new ArrayList<>());
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Application1<String> tree = new Application1<>();

        int choice;
        do {
            displayMenu();
            System.out.print("Enter your choice (1-15 to perform operations, 0 to exit): ");
            choice = scanner.nextInt();

```

```

scanner.nextLine();

switch (choice) {
    case 1:
        System.out.print("Enter root value: "); //Enter the root value
        String rootValue = scanner.nextLine();
        tree.root = new TreeNode<>(rootValue, null); // Set the root of the tree
        break;

    case 2:
        System.out.print("Enter parent value: "); //Enter parent value
        String parentValue = scanner.nextLine();
        Position<String> parentNode = findNode(tree, parentValue); //Finds the parent node
        if (parentNode != null) {
            System.out.print("Enter child value: "); //Enter the child value
            String childValue = scanner.nextLine();
            tree.addChild(parentNode, childValue); // Add a child to the specified parent
            System.out.println("Child added successfully.");
        } else {
            System.out.println("Parent node not found."); //In case parent node not found
        }
        break;

    case 3:
        System.out.println("Tree:");
        ((MyTree<String>) tree).displayTree();
        break;

    case 4:
        System.out.println("Indented Tree Structure:");
        tree.printIndentedTree(tree);
        break;

    default:
        if (choice != 0) {
            System.out.println("Invalid choice. Please try again.");
        }
    }
} while (choice != 0);
scanner.close();
}

```



```

private static void displayMenu() {
    System.out.println("\nMenu:");
    System.out.println("1. Set root");
    System.out.println("2. Add child node");
    System.out.println("3. Display tree");
    System.out.println("4. Print Indented Tree Structure");
    System.out.println("0. Exit");
}

private static Position<String> findNode(Tree<String> tree, String nodeValue) {
    // Implements logic to find a node with the given value in the tree
    // Return the node if found, otherwise return null
    for (Position<String> node : tree.positions()) {
        if (node.getElement().equals(nodeValue)) {
            return node;
        }
    }
    return null;
}
}

```

Output-

Menu:

1. Set root
2. Add child node
3. Display tree
4. Print Indented Tree Structure
0. Exit

Enter your choice (1-4 to perform operations, 0 to exit): 4

Indented Tree Structure:

Electronics R'us

- 1 R&D
- 2 Sales
 - 2.1 Domestic
 - 2.2 International
 - 2.2.1 Canada
 - 2.2.2 S.America
 - 2.2.3 Overseas
 - 2.2.3.1 Africa
 - 2.2.3.2 Europe
 - 2.2.3.3 Asia
 - 2.2.3.4 Australia
- 3 Purchasing
- 4 Manufacturing
 - 4.1 TV
 - 4.2 CD
 - 4.3 Tuner

Menu:

1. Set root
2. Add child node
3. Display tree
4. Print Indented Tree Structure
0. Exit

Enter your choice (1-4 to perform operations, 0 to exit): 0

PS C:\Users\abhim\OneDrive\Desktop\Abhi Mehta\TreeTest> █

Application2-

```
//Abstractclass.java
import java.util.*;
// Position interface representing a node in the tree
interface Position<E> {
    E getElement();
}

// Tree interface defining basic tree operations
interface Tree<E> extends Iterable<E> {
    Position<E> root();//Returns position of root of the tree(null if empty)

    Position<E> parent(Position<E> p) throws IllegalArgumentException;//Returns position of
    parent of position p of the tree(null if empty)

    Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;//Returns
    iterable collection containing children at position p

    int numChildren(Position<E> p) throws IllegalArgumentException;//Returns no. of children at
    position p

    boolean isInternal(Position<E> p) throws IllegalArgumentException;//Returns true if there is
    atleast 1 child

    boolean isExternal(Position<E> p) throws IllegalArgumentException;//Returns true if there is
    no child

    boolean isRoot(Position<E> p) throws IllegalArgumentException;//True if p is the root

    int size();//Return no. of positions and hence elements in a tree

    boolean isEmpty();//Returns true if tree does not contain any positions

    Iterator<E> iterator();//Returns iterator for all elements in the tree

    Iterable<Position<E>> positions();//Returns iterable collection of all positions
}

// AbstractTree class providing a base implementation for common tree operations
abstract class AbstractTree<E> implements Tree<E> {

    // Common methods can be implemented here which are used for tree operations

    // Validate a position and convert it to TreeNode
```

```

private TreeNode<E> validate(Position<E> p) throws IllegalArgumentException {
    if (!(p instanceof TreeNode)) {
        throw new IllegalArgumentException("Invalid position type");
    }
    TreeNode<E> node = (TreeNode<E>) p;
    if (node.getParent() == node) {
        throw new IllegalArgumentException("Position no longer in the tree");
    }
    return node;
}

// Abstract method to add a child node to a given parent position
public abstract Position<E> addChild(Position<E> p, E element);

// Abstract method to display the tree
public abstract void displayTree();

// Method to get siblings of a node
public Iterable<Position<E>> siblings(Position<E> p) {
    // Get the parent of the given node
    Position<E> parent = parent(p);

    // Check if the node is the root or has no parent
    if (parent == null)
        return Collections.emptyList(); // p is the root, return an empty list
    else {
        // Get the children of the parent
        Iterable<Position<E>> siblings = children(parent);

        // Create and return a filtered iterable that excludes the original node
        List<Position<E>> filteredSiblings = new ArrayList<>();
        for (Position<E> sibling : siblings) {
            if (!sibling.equals(p)) {
                filteredSiblings.add(sibling);
            }
        }
        return filteredSiblings;
    }
}

// Method to list leaves of the tree
public List<Position<E>> leaves() {
    List<Position<E>> leafList = new ArrayList<>();

```

```

    for (Position<E> p : positions()) {
        if (isExternal(p)) {
            leafList.add(p);
        }
    }
    return leafList;
}

// Method to list internal nodes of the tree
public List<Position<E>> internalNodes() {
    List<Position<E>> internalNodesList = new ArrayList<>();
    for (Position<E> p : positions()) {
        if (isInternal(p)) {
            internalNodesList.add(p);
        }
    }
    return internalNodesList;
}

// Method to list edges of the tree
public List<Edge<E>> edges() {
    List<Edge<E>> edgeList = new ArrayList<>();
    for (Position<E> p : positions()) {
        for (Position<E> c : children(p)) {
            edgeList.add(new Edge<>(p, c));
        }
    }
    return edgeList;
}

// Method to get the path for a given node
public List<Position<E>> path(Position<E> p) {
    List<Position<E>> pathList = new ArrayList<>();
    while (p != null) {
        pathList.add(p);
        p = parent(p);
    }
    Collections.reverse(pathList);
    return pathList;
}

// Method to get the depth of a node
public int depth(Position<E> p) {
    if (isRoot(p))

```

```

        return 0;
    else
        return 1 + depth(parent(p));
    }

// Method to get the height of the tree
public int height(Position<E> p) {
    int h = 0;
    for (Position<E> c : children(p)) {
        h = Math.max(h, 1 + height(c));
    }
    return h;
}

// Method to get a subtree rooted at a given node
//subTree method generates a new tree representing the subtree rooted at a given node
public Tree<E> subTree(Position<E> p) {
    TreeNode<E> rootNode = copySubTree(validate(p), null); //copySubTree method is a
    recursive helper method used to copy the structure of the subtree.
    MyTree<E> subtree = new MyTree<>(); //MyTree<E> object named subtree is created.
    subtree.root = rootNode; //rootNode is set as the root of the subtree.
    return subtree;
}

// Helper method to recursively copy the subtree
private TreeNode<E> copySubTree(Position<E> originalNode, TreeNode<E> parent) {
    // A new TreeNode called newNode with the same element as the originalNode and the
    specified parent.
    TreeNode<E> newNode = new TreeNode<>(originalNode.getElement(), parent);
    /// For each child, it recursively calls copySubTree to copy the child and its subtree.
    for (Position<E> child : children(originalNode)) {
        TreeNode<E> newChild = copySubTree(child, newNode);
        newNode.getChildren().add(newChild); //The newly created child node (newChild) is
        added to the list of children of the newNode.
    }

    return newNode;
}

}

// PositionalElement interface
interface PositionalElement {

```

```
void setX(int x);  
void setY(int y);  
}
```

```
class TreeNode<E> implements Position<E> {  
    private E element;  
    private TreeNode<E> parent;  
    private List<TreeNode<E>> children;  
    private int x; // Added field for X-coordinate  
    private int y; // Added field for Y-coordinate  
  
    // Constructor for creating a tree node with an element and parent  
    public TreeNode(E element, TreeNode<E> parent) {  
        this.element = element;  
        this.parent = parent;  
        this.children = new ArrayList<>();  
    }  
  
    // Method to get the element of the node  
    public E getElement() {  
        return element;  
    }  
  
    // Method to get the parent of the node  
    public TreeNode<E> getParent() {  
        return parent;  
    }  
  
    // Method to get the children of the node  
    public List<TreeNode<E>> getChildren() {  
        return children;  
    }  
  
    // Method to get the left child of the node  
    public TreeNode<E> getLeftChild() {  
        return children.isEmpty() ? null : children.get(0);  
    }  
  
    // Method to get the right child of the node  
    public TreeNode<E> getRightChild() {  
        return children.size() < 2 ? null : children.get(1);  
    }  
  
    // Method to get the X-coordinate
```

```

public int getX() {
    return x;
}

// Method to set the X-coordinate
public void setX(int x) {
    this.x = x;
}

// Method to get the Y-coordinate
public int getY() {
    return y;
}

// Method to set the Y-coordinate
public void setY(int y) {
    this.y = y;
}
}

// Edge class representing a connection between two positions
class Edge<E> {
    private Position<E> source;
    private Position<E> target;

    // Constructor for creating an edge between two positions
    public Edge(Position<E> source, Position<E> target) {
        this.source = source;
        this.target = target;
    }

    // Method to get the source position of the edge
    public Position<E> getSource() {
        return source;
    }

    // Method to get the target position of the edge
    public Position<E> getTarget() {
        return target;
    }
}

```


MyTree.java

```
import java.util.*;
```

```
// MyTree class implementing the AbstractTree interface
public class MyTree<E> extends AbstractBinaryTree<E>{
```

```
    @Override
```

```
    public Position<E> left(Position<E> p) throws IllegalArgumentException {
        TreeNode<E> node = validate(p);
        return node.getLeftChild();
    }
```

```
    @Override
```

```
    public Position<E> right(Position<E> p) throws IllegalArgumentException {
        TreeNode<E> node = validate(p);
        return node.getRightChild();
    }
```

```
    // The root of the tree
    protected TreeNode<E> root;
```

```
    // Constructor for creating an empty tree
    public MyTree() {
        this.root = null;
    }
```

```
    // Implementing methods from the Tree interface
```

```
    // Returns the root position of the tree
    public Position<E> root() {
        return root;
    }
```

```
    // Returns the root as a TreeNode
    public TreeNode<E> getRoot() {
        return root;
    }
```

```
    // Returns the parent position of a given position
    public Position<E> parent(Position<E> p) throws IllegalArgumentException {
        TreeNode<E> node = validate(p);
        return node.getParent();
    }
```

```

}

// Returns the children of a given position
// Updated return type for children method
public Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException {
    TreeNode<E> node = validate(p);
    return new ArrayList<>(node.getChildren());
}

// Returns the number of children of a given position
public int numChildren(Position<E> p) throws IllegalArgumentException {
    TreeNode<E> node = validate(p);
    return node.getChildren().size();
}

// Checks if a given position is internal
public boolean isInternal(Position<E> p) throws IllegalArgumentException {
    return numChildren(p) > 0;
}

// Checks if a given position is external
public boolean isExternal(Position<E> p) throws IllegalArgumentException {
    return numChildren(p) == 0;
}

// Checks if a given position is the root of the tree
public boolean isRoot(Position<E> p) throws IllegalArgumentException {
    TreeNode<E> node = validate(p);
    return node == root;
}

// Returns the total number of positions in the tree
public int size() {
    return size(root);
}

// Recursive helper method to calculate the size of the subtree rooted at a given position
private int size(Position<E> p) {
    int count = 1; // count for the root
    for (Position<E> child : children(p)) {
        count += size(child);
    }
    return count;
}

```

```

// Checks if the tree is empty
public boolean isEmpty() {
    return size() == 0;
}

// Returns an iterator over the elements of the tree
public Iterator<E> iterator() {
    return new Treeliterator();
}

// Returns an iterable collection of all positions in the tree
public Iterable<Position<E>> positions() {
    List<Position<E>> positions = new ArrayList<>();
    collectPositions(root, positions);
    return positions;
}

// Iterator for the tree
private class Treeliterator implements Iterator<E> {
    private Iterator<Position<E>> positionIterator = positions().iterator();

    public boolean hasNext() {
        return positionIterator.hasNext();
    }

    public E next() {
        return positionIterator.next().getElement();
    }
}

// Add child node to a given parent position
public Position<E> addChild(Position<E> p, E element) {
    TreeNode<E> parent = validate(p);
    TreeNode<E> child = new TreeNode<>(element, parent);
    parent.getChildren().add(child);
    return child;
}

// Display the tree
public void displayTree() {
    displayTree(root, 0);
}

```

```

// Recursive method to display the tree structure
private void displayTree(Position<E> node, int depth) {
    if (node != null) {
        for (int i = 0; i < depth; i++) {
            System.out.print(" ");
        }
        System.out.println(node.getElement());
        for (Position<E> child : children(node)) {
            displayTree(child, depth + 1);
        }
    }
}

// Recursive method to collect positions in the tree
private void collectPositions(TreeNode<E> node, List<Position<E>> positions) {
    if (node != null) {
        positions.add(node);
        for (TreeNode<E> child : node.getChildren()) {
            collectPositions(child, positions);
        }
    }
}

// Helper method to validate a position
private TreeNode<E> validate(Position<E> p) throws IllegalArgumentException {
    if (!(p instanceof TreeNode)) {
        throw new IllegalArgumentException("Invalid position type");
    }
    TreeNode<E> node = (TreeNode<E>) p;
    if (node.getParent() == node) {
        throw new IllegalArgumentException("Position no longer in the tree");
    }
    return node;
}
}

```

```

//Application2.java
import java.io.*;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
class Information{
    public String name;
    public boolean type;
    public long size;
}

```

```

    public long date;
}
public class Application2
{
    public static Application1<Information> infoTree = new Application1<Information>();
    public static void main(String[] args)
    {
        Application1<Information> infoTree = new Application1<>();
        File f = new File("C:\\Users\\abhim\\OneDrive\\Desktop\\Abhi Mehta");
        Information r = new Information();
        setInformation(r, f);
        infoTree.root=new TreeNode<>(r, null);
        constructTree(infoTree.root(), f);
        Iterable<Position<Information>> iter = infoTree.positions();
        DateFormat sdf = new SimpleDateFormat("MMMM dd, yyyy hh:mm a");
        for(Position<Information> i : iter){
            String Type;
            if(i.getElement().type)
                Type = "Directory";
            else
                Type = "File";
            System.out.println(i.getElement().size + "\\t" + sdf.format(i.getElement().date) + "\\t" +
i.getElement().name + "\\t" + Type);
        }
    }

    public static Information setInformation(Information i, File root){
        i.name = root.getName();
        i.date = root.lastModified();
        i.type = root.isDirectory();
        i.size = root.length();
        return i;
    }

    public static Position<Information> addNode(Position<Information> parent, File root){
        Information i = new Information();
        i = setInformation(i, root);
        return infoTree.addChild(parent, i);
    }

    public static void constructTree(Position<Information> parent,File root){
        if(root.isDirectory()) {
            for (String childname : root.list()) {
                File child = new File(root, childname);
                Position<Information> temp = addNode(parent, child);
                constructTree(temp, child);
            }
        }
    }
}

```

Output-

```

PS C:\Users\abhim\OneDrive\Desktop\Abhi Mehta\TreeTest> cd "c:\Users\abhim\OneDrive\Desktop\
8192 November 25, 2023 09:57 PM Abhi Mehta Directory
383201 February 20, 2023 02:49 AM 10.1.png File
329051 February 20, 2023 02:50 AM 10.2.png File
250974 January 16, 2023 12:47 AM 3.2 corrected.png File
211492 January 22, 2023 07:30 PM 5.1 correctttt.png File
341760 January 22, 2023 07:28 PM 5.2 correctttt.png File
261143 January 22, 2023 08:50 PM 6.1.png File
273750 January 22, 2023 06:59 PM 6.2 correcttt.png File
218243 February 27, 2023 03:40 AM 8888888888888888.png File
26226504 November 07, 2023 09:01 PM aerial_modelcheckpoint.h5 File
184069 January 01, 2023 08:16 PM Drawing 1.1.png File
137182 January 01, 2023 10:46 PM Drawing 1.2.png File
187994 January 01, 2023 03:43 AM Drawing 2.1.png File
363191 January 01, 2023 03:56 AM drawing 2.2.png File
121342 January 01, 2023 05:50 PM Drawing 3.1.png File
221773 January 01, 2023 06:07 PM Drawing 3.2.png File
218869 January 01, 2023 07:08 PM Drawing 4.1.png File
296754 January 01, 2023 07:42 PM Drawing 4.2.png File
330982 January 17, 2023 10:23 PM drawing 6.2.png File
487843 February 20, 2023 02:46 AM ex1.png File
548196 February 20, 2023 02:47 AM ex2.png File
304739 February 20, 2023 02:48 AM ex3.png File
347860 February 27, 2023 01:34 AM exc81.png File
348473 February 27, 2023 02:33 AM EXC82.png File
237350 February 27, 2023 02:58 AM EXC83.png File
249895 February 13, 2023 08:01 AM iso1.png File
400715 February 13, 2023 08:10 AM iso2.png File
235216 February 13, 2023 08:14 AM isoo1.png File
174280 November 12, 2023 10:47 PM lab6.pdf File
393180 October 16, 2023 10:20 PM sih presentation 101.pdf File
8192 November 25, 2023 10:21 PM Tree7 Directory
1062 November 25, 2023 10:09 PM AbstractBinaryTree.class File
1175 November 25, 2023 10:07 PM AbstractBinaryTree.java File
1021 November 25, 2023 10:09 PM AbstractTree$ElementIterator.class File
2424 November 25, 2023 10:09 PM AbstractTree.class File
2668 November 25, 2023 10:01 PM AbstractTree.java File
844 November 25, 2023 10:10 PM Application3.class File
1324 November 25, 2023 10:09 PM Application3.java File
602 November 25, 2023 10:30 PM Application4.class File
485 November 25, 2023 10:21 PM Application4.java File
403 November 25, 2023 10:09 PM BinaryTree.class File
291 November 25, 2023 10:06 PM BinaryTree.java File
1041 November 25, 2023 10:30 PM LinkedBinaryTree$ElementIterator.class File
1533 November 25, 2023 10:30 PM LinkedBinaryTree$Node.class File
6541 November 25, 2023 10:30 PM LinkedBinaryTree.class File
17746 November 25, 2023 10:30 PM LinkedBinaryTree.java File
280 November 25, 2023 10:09 PM Position.class File
82 November 25, 2023 10:03 PM Position.java File
987 November 25, 2023 10:19 PM Tree.class File
898 November 25, 2023 10:10 PM Tree.java File
8192 November 25, 2023 10:55 PM TreeTest Directory
1716 November 23, 2023 02:48 AM AbstractBinaryTree.class File
1647 November 23, 2023 01:05 AM AbstractBinaryTree.java File
3468 November 25, 2023 09:03 PM AbstractTree.class File
8410 November 25, 2023 09:01 PM AbstractTree.java File
3952 November 25, 2023 10:57 PM Application1.class File
3906 November 25, 2023 10:57 PM Application1.java File
3110 November 25, 2023 11:09 PM Application2.class File
2038 November 25, 2023 10:55 PM Application2.java File

```

```

403      November 23, 2023 02:48 AM      BinaryTree.class      File
537      November 23, 2023 12:45 AM      BinaryTree.java File
607      November 25, 2023 09:03 PM      Edge.class      File
3953     November 25, 2023 10:54 PM      ExtendedTree.class      File
286      November 25, 2023 11:09 PM      Information.class      File
1820     November 23, 2023 01:15 PM      LinkedBinaryTree$Node.class      File
3826     November 23, 2023 01:15 PM      LinkedBinaryTree.class      File
7323     November 25, 2023 09:35 PM      LinkedBinaryTree.java      File
923      November 25, 2023 08:23 PM      MyTree$TreeIterator.class      File
3886     November 25, 2023 08:23 PM      MyTree.class      File
5196     November 25, 2023 08:21 PM      MyTree.java      File
224      November 25, 2023 09:03 PM      Position.class      File
147      November 25, 2023 09:03 PM      PositionalElement.class      File
995      November 25, 2023 09:03 PM      Tree.class      File
1559     November 25, 2023 09:03 PM      TreeNode.class      File
3987     November 25, 2023 07:57 PM      TreeTestApplication.class      File
2995     November 25, 2023 10:55 PM      TreeTestApplication.java      File
PS C:\Users\abhim\OneDrive\Desktop\Abhi Mehta\TreeTest>

```

Application3-

```

public interface Position<E>{
    E getElement() throws IllegalStateException;
}
import java.util.Iterator;
/* We need to print indents before nodes to indicate that they are child */
/* We will need a class tree to make a tree */
/* We will need functions that will help us print the indents */
//Tree.java
public interface Tree<E> extends Iterable<E>
{
    Position<E> root( );//
    Position<E> parent(Position<E> p) throws IllegalArgumentException;//
    Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;//
    int numChildren(Position<E> p) throws IllegalArgumentException;//
    boolean isInternal(Position<E> p) throws IllegalArgumentException;//
    boolean isExternal(Position<E> p) throws IllegalArgumentException;//
    boolean isRoot(Position<E> p) throws IllegalArgumentException;//
    int size( );//
    boolean isEmpty( );//
    Iterator<E> iterator( );//
}

```

```

    Iterable<Position<E>> positions();//
}
//AbstractTree.java
import java.util.Iterator;
import java.util.ArrayList;
import java.util.List;
public abstract class AbstractTree<E> implements Tree<E>
{
    public boolean isInternal(Position<E> p)
        { return numChildren(p) > 0; }
    public boolean isExternal(Position<E> p)
        { return numChildren(p) == 0; }
    public boolean isRoot(Position<E> p)
        { return p == root( ); }
    public boolean isEmpty( )
        { return size( ) == 0; }

    public int depth(Position<E> p)
    {
        if (isRoot(p))
            return 0;
        else
            return 1 + depth(parent(p));
    }
    // private int heightBad( )
    // { // works, but quadratic worst-case time
    //     int h = 0;
    //     for (Position<E> p : positions( ))
    //         if (isExternal(p)) // only consider leaf positions
    //             h = Math.max(h, depth(p));
    //     return h;
    // }
    public int height(Position<E> p)
    {
        int h = 0; // base case if p is external
        for (Position<E> c : children(p))
            h = Math.max(h, 1 + height(c));
        return h;
    }
    private class ElementIterator implements Iterator<E>
    {
        Iterator<Position<E>> posIterator = positions( ).iterator( );
        public boolean hasNext( ) { return posIterator.hasNext( ); }
        public E next( ) { return posIterator.next( ).getElement( ); } // return element!
        public void remove( ) { posIterator.remove( ); }
    }
}

```



```

    }
    public Iterator<E> iterator( ) { return new ElementIterator( ); }
    private void preorderSubtree(Position<E> p, List<Position<E>> snapshot)
    {
        snapshot.add(p); // for preorder, we add position p before exploring subtrees
        for (Position<E> c : children(p))
            preorderSubtree(c, snapshot);
    }
    public Iterable<Position<E>> preorder( )
    {
        List<Position<E>> snapshot = new ArrayList<>( );
        if (!isEmpty( ))
            preorderSubtree(root( ), snapshot); // fill the snapshot recursively
        return snapshot;
    }
    public Iterable<Position<E>> positions( ) { return preorder( ); }
    private void postorderSubtree(Position<E> p, List<Position<E>> snapshot)
    {
        for (Position<E> c : children(p))
            postorderSubtree(c, snapshot);
        snapshot.add(p); // for postorder, we add position p after exploring subtrees
    }
    public Iterable<Position<E>> postorder( )
    {
        List<Position<E>> snapshot = new ArrayList<>( );
        if (!isEmpty( ))
            postorderSubtree(root( ), snapshot); // fill the snapshot recursively
        return snapshot;
    }
}

//BinaryTree.java
public interface BinaryTree<E> extends Tree<E>
{
    Position<E> left(Position<E> p) throws IllegalArgumentException;
    Position<E> right(Position<E> p) throws IllegalArgumentException;
    Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}

import java.util.*;
public abstract class AbstractBinaryTree<E> extends AbstractTree<E> implements
    BinaryTree<E>

```

```

{
    public Position<E> sibling(Position<E> p)
    {
        Position<E> parent = parent(p);
        if (parent == null) return null; // p must be the root
        if (p == left(parent)) // p is a left child
            return right(parent); // (right child might be null)
        else // p is a right child
            return left(parent); // (left child might be null)
    }

    /** Returns the number of children of Position p. */
    public int numChildren(Position<E> p)
    {
        int count=0;
        if (left(p) != null)
            count++;
        if (right(p) != null)
            count++;
        return count;
    }

    /** Returns an iterable collection of the Positions representing p's children. */
    public Iterable<Position<E>> children(Position<E> p)
    {
        List<Position<E>> snapshot = new ArrayList<>(2); // max capacity of 2
        if (left(p) != null)
            snapshot.add(left(p));
        if (right(p) != null)
            snapshot.add(right(p));
        return snapshot;
    }
}

```

```

//LinkedListBinaryTree.java
import java.util.Iterator;
import java.util.Stack;
public class LinkedListBinaryTree<E> extends AbstractBinaryTree<E>
{
    protected static class Node<E> implements Position<E>
    {
        private E element; // an element stored at this node
        private Node<E> parent; // a reference to the parent node (if any)
    }
}

```

```

private Node<E> left; // a reference to the left child (if any)
private Node<E> right; // a reference to the right child (if any)
/** Constructs a node with the given element and neighbors. */
public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild)
{
    element = e;
    parent = above;
    left = leftChild;
    right = rightChild;
} // accessor methods
public E getElement( )
{ return element; }
public Node<E> getParent( )
{ return parent; }
public Node<E> getLeft( )
{ return left; }
public Node<E> getRight( )
{ return right; } // update methods
public void setElement(E e)
{ element = e; }
public void setParent(Node<E> parentNode)
{ parent = parentNode; }
public void setLeft(Node<E> leftChild)
{ left = leftChild; }
public void setRight(Node<E> rightChild)
{ right = rightChild; }
} //----- end of nested Node class -----

protected Node<E> createNode(E e, Node<E> parent, Node<E> left, Node<E> right)
{ return new Node<E>(e, parent, left, right);}
protected Node<E> root = null; // root of the tree
private int size = 0; // number of nodes in the tree
public LinkedBinaryTree( ) { } // constructs an empty binary tree
protected Node<E> validate(Position<E> p) throws IllegalArgumentException
{
    if (!(p instanceof Node)) throw new IllegalArgumentException("Not valid position type");
    Node<E> node = (Node<E>) p; // safe cast
    if (node.getParent( ) == node) // our convention for defunct node
        throw new IllegalArgumentException("p is no longer in the tree");
    return node;
}
public int size( )
{ return size;}
public Position<E> root( )

```

```

    { return root;}
public Position<E> parent(Position<E> p) throws IllegalArgumentException
{
    Node<E> node = validate(p);
    return node.getParent( );
}
public Position<E> left(Position<E> p) throws IllegalArgumentException
{
    Node<E> node = validate(p);
    return node.getLeft( );
}
public Position<E> right(Position<E> p) throws IllegalArgumentException
{
    Node<E> node = validate(p);
    return node.getRight( );
}
// update methods supported by this class
/** Places element e at the root of an empty tree and returns its new Position. */
public Position<E> addRoot(E e) throws IllegalStateException
{
    if (!isEmpty( )) throw new IllegalStateException("Tree is not empty");
    root = createNode(e, null, null, null);
    size = 1;
    return root;
}
/** Creates a new left child of Position p storing element e; returns its Position. */
public Position<E> addLeft(Position<E> p, E e) throws IllegalArgumentException
{
    Node<E> parent = validate(p);
    if (parent.getLeft( ) != null) throw new IllegalArgumentException("p already has a left
child");
    Node<E> child = createNode(e, parent, null, null);
    parent.setLeft(child);
    size++;
    return child;
}
/** Creates a new right child of Position p storing element e; returns its Position. */
public Position<E> addRight(Position<E> p, E e) throws IllegalArgumentException
{
    Node<E> parent = validate(p);
    if (parent.getRight( ) != null) throw new IllegalArgumentException("p already has a right
child");
    Node<E> child = createNode(e, parent, null, null);
    parent.setRight(child);
}

```

```

        size++;
        return child;
    }
    /** Replaces the element at Position p with e and returns the replaced element. */
    public E set(Position<E> p, E e) throws IllegalArgumentException
    {
        Node<E> node = validate(p);
        E temp = node.getElement( );
        node.setElement(e);
        return temp;
    }
    /** Attaches trees t1 and t2 as left and right subtrees of external p. */
    public void attach(Position<E> p, LinkedBinaryTree<E> t1, LinkedBinaryTree<E> t2) throws
    IllegalArgumentException
    {
        Node<E> node = validate(p);
        if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");
        size += t1.size( ) + t2.size( );
        if (!t1.isEmpty( ))
        { // attach t1 as left subtree of node
            t1.root.setParent(node);
            node.setLeft(t1.root);
            t1.root = null;
            t1.size = 0;
        }
        if (!t2.isEmpty( ))
        { // attach t2 as right subtree of node
            t2.root.setParent(node);
            node.setRight(t2.root);
            t2.root = null;
            t2.size = 0;
        }
    }
    /** Removes the node at Position p and replaces it with its child, if any. */
    public E remove(Position<E> p) throws IllegalArgumentException
    {
        Node<E> node = validate(p);
        if (numChildren(p) == 2) throw new IllegalArgumentException("p has two children");
        Node<E> child = (node.getLeft( ) != null ? node.getLeft( ) : node.getRight( ) );
        if (child != null)
            child.setParent(node.getParent( )); // child's grandparent becomes its parent
        if (node == root)
            root = child; // child becomes root
        else
        {

```

```

        Node<E> parent = node.getParent( );
        if (node == parent.getLeft( ))
            parent.setLeft(child);
        else
            parent.setRight(child);
    }
    size--;
    E temp = node.getElement( );
    node.setElement(null); // help garbage collection
    node.setLeft(null);
    node.setRight(null);
    node.setParent(node); // our convention for defunct node
    return temp;
}
private class ElementIterator implements Iterator<E>
{
    Iterator<Position<E>> posIterator = positions( ).iterator( );
    public boolean hasNext( ) { return posIterator.hasNext( ); }
    public E next( ) { return posIterator.next( ).getElement( ); } // return element!
    public void remove( ) { posIterator.remove( ); }
}
/** Returns an iterator of the elements stored in the tree. */
public Iterator<E> iterator( ) { return new ElementIterator( ); }
// public void parenthesize(Tree<E> T, Position<E> p) //to print normal elements in
parenthesis form
Stack<E> op=new Stack<>();
public void parenthesize(Tree<E> T, Position<E> p) //to print expression in parenthesis form
{
    if (p.getElement()=="+"||p.getElement()=="-"||p.getElement()=="*"||p.getElement()=="/")
        op.push(p.getElement());
    else
        System.out.print(p.getElement());
    if (T.isInternal(p))
    {
        boolean firstTime = true;
        for (Position<E> c : T.children(p))
        {
            System.out.print( (firstTime ? "(" : op.pop()) );
            firstTime = false;
            parenthesize(T, c); // recur on child
        }
        System.out.print(")");
    }
}
}

```

```

public void traverseNodes(StringBuilder sb, String padding, String pointer, BinaryTree<E> T,
Position<E> p, boolean hasRightSibling)
{
    if (p != null)
    {
        sb.append("\n");
        sb.append(padding);
        sb.append(pointer);
        sb.append(p.getElement());

        StringBuilder paddingBuilder = new StringBuilder(padding);
        if (hasRightSibling) {
            paddingBuilder.append("| ");
        } else {
            paddingBuilder.append(" ");
        }

        String paddingForBoth = paddingBuilder.toString();
        String pointerRight = "└─";
        String pointerLeft = (T.right(p) != null) ? "├─" : "└─";

        traverseNodes(sb, paddingForBoth, pointerLeft, T, T.left(p), T.right(p) != null);
        traverseNodes(sb, paddingForBoth, pointerRight, T, T.right(p), false);
    }
}

public String traversePreOrder(BinaryTree<E> T, Position<E> p)
{
    if (p == null) {
        return "";
    }

    StringBuilder sb = new StringBuilder();
    sb.append(p.getElement());

    String pointerRight = "└─";
    String pointerLeft = (T.right(p) != null) ? "├─" : "└─";

    traverseNodes(sb, "", pointerLeft, T, T.left(p), T.right(p) != null);
    traverseNodes(sb, "", pointerRight, T, T.right(p), false);

    return sb.toString();
}

```

```

public void printTree(BinaryTree<E> T, Position<E> p)
{
    System.out.println(traversePreOrder(T,p));
}
}

```

//Application3.java

```

public class Application3
{
    public static void main(String[] args)
    {
        LinkedBinaryTree<String> T1 = new LinkedBinaryTree<>();
        Position<String> root = T1.addRoot("-");
        Position<String> node1=T1.addLeft(root,"/");
        Position<String> node2=T1.addRight(root,"+");

        Position<String> node1l=T1.addLeft(node1,"*");
        Position<String> node1r=T1.addRight(node1,"+");

        Position<String> node2l=T1.addLeft(node2,"*");
        Position<String> node2r=T1.addRight(node2,"6");

        Position<String> node1ll=T1.addLeft(node1l,"+");
        Position<String> node1lr=T1.addRight(node1l,"3");

        Position<String> node1rl=T1.addLeft(node1r,"-");
        Position<String> node1rr=T1.addRight(node1r,"2");

        Position<String> node1lll=T1.addLeft(node1ll,"3");
        Position<String> node1llr=T1.addRight(node1ll,"1");

        Position<String> node1rll=T1.addLeft(node1rl,"9");
        Position<String> node1rlr=T1.addRight(node1rl,"5");

        Position<String> node2ll=T1.addLeft(node2l,"3");
        Position<String> node2lr=T1.addRight(node2l,"-");

        Position<String> node2lrl=T1.addLeft(node2lr,"7");
        Position<String> node2lrr=T1.addRight(node2lr,"4");

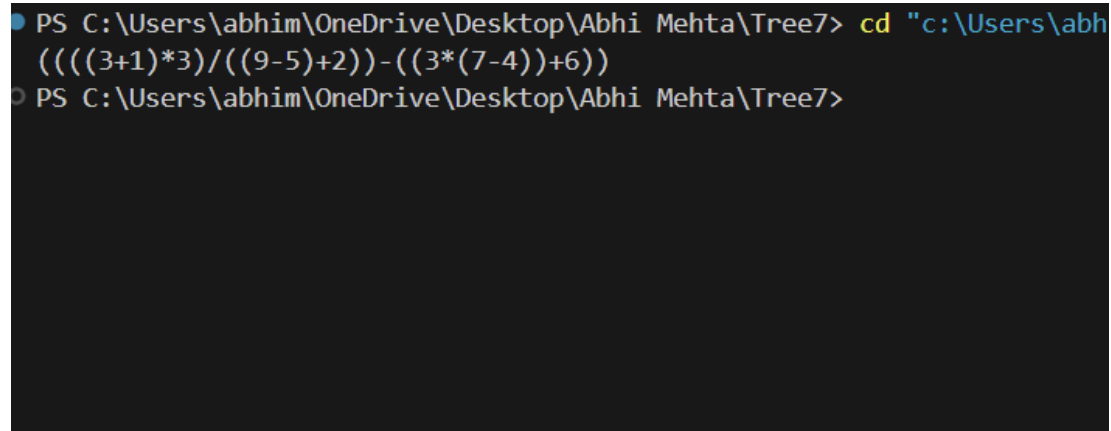
        T1.parenthesize(T1, root);
    }
}

```



```
}  
  
}
```

Output-



```
PS C:\Users\abhim\OneDrive\Desktop\Abhi Mehta\Tree7> cd "c:\Users\abh  
(((3+1)*3)/((9-5)+2))-((3*(7-4))+6))  
PS C:\Users\abhim\OneDrive\Desktop\Abhi Mehta\Tree7>
```

Application4-

```
public interface Position<E>{  
    E getElement() throws IllegalStateException;  
}  
import java.util.Iterator;  
/* We need to print indents before nodes to indicate that they are child */  
/* We will need a class tree to make a tree */  
/* We will need functions that will help us print the indents */  
//Tree.java  
public interface Tree<E> extends Iterable<E>  
{  
    Position<E> root( );//  
    Position<E> parent(Position<E> p) throws IllegalArgumentException;//  
    Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;//  
    int numChildren(Position<E> p) throws IllegalArgumentException;//  
    boolean isInternal(Position<E> p) throws IllegalArgumentException;//  
    boolean isExternal(Position<E> p) throws IllegalArgumentException;//  
    boolean isRoot(Position<E> p) throws IllegalArgumentException;//  
    int size( );//  
    boolean isEmpty( );//  
    Iterator<E> iterator( );//
```

```

        Iterable<Position<E>> positions();//
    }
//AbstractTree.java
import java.util.Iterator;
import java.util.ArrayList;
import java.util.List;
public abstract class AbstractTree<E> implements Tree<E>
{
    public boolean isInternal(Position<E> p)
        { return numChildren(p) > 0; }
    public boolean isExternal(Position<E> p)
        { return numChildren(p) == 0; }
    public boolean isRoot(Position<E> p)
        { return p == root( ); }
    public boolean isEmpty( )
        { return size( ) == 0; }

    public int depth(Position<E> p)
    {
        if (isRoot(p))
            return 0;
        else
            return 1 + depth(parent(p));
    }
    // private int heightBad( )
    // { // works, but quadratic worst-case time
    //     int h = 0;
    //     for (Position<E> p : positions( ))
    //         if (isExternal(p)) // only consider leaf positions
    //             h = Math.max(h, depth(p));
    //     return h;
    // }
    public int height(Position<E> p)
    {
        int h = 0; // base case if p is external
        for (Position<E> c : children(p))
            h = Math.max(h, 1 + height(c));
        return h;
    }
    private class ElementIterator implements Iterator<E>
    {
        Iterator<Position<E>> posIterator = positions( ).iterator( );
        public boolean hasNext( ) { return posIterator.hasNext( ); }
        public E next( ) { return posIterator.next( ).getElement( ); } // return element!
        public void remove( ) { posIterator.remove( ); }
    }
}

```

```

    }
    public Iterator<E> iterator( ) { return new ElementIterator( ); }
    private void preorderSubtree(Position<E> p, List<Position<E>> snapshot)
    {
        snapshot.add(p); // for preorder, we add position p before exploring subtrees
        for (Position<E> c : children(p))
            preorderSubtree(c, snapshot);
    }
    public Iterable<Position<E>> preorder( )
    {
        List<Position<E>> snapshot = new ArrayList<>( );
        if (!isEmpty( ))
            preorderSubtree(root( ), snapshot); // fill the snapshot recursively
        return snapshot;
    }
    public Iterable<Position<E>> positions( ) { return preorder( ); }
    private void postorderSubtree(Position<E> p, List<Position<E>> snapshot)
    {
        for (Position<E> c : children(p))
            postorderSubtree(c, snapshot);
        snapshot.add(p); // for postorder, we add position p after exploring subtrees
    }
    public Iterable<Position<E>> postorder( )
    {
        List<Position<E>> snapshot = new ArrayList<>( );
        if (!isEmpty( ))
            postorderSubtree(root( ), snapshot); // fill the snapshot recursively
        return snapshot;
    }
}

//BinaryTree.java
public interface BinaryTree<E> extends Tree<E>
{
    Position<E> left(Position<E> p) throws IllegalArgumentException;
    Position<E> right(Position<E> p) throws IllegalArgumentException;
    Position<E> sibling(Position<E> p) throws IllegalArgumentException;
}

import java.util.*;
public abstract class AbstractBinaryTree<E> extends AbstractTree<E> implements
    BinaryTree<E>

```

```

{
    public Position<E> sibling(Position<E> p)
    {
        Position<E> parent = parent(p);
        if (parent == null) return null; // p must be the root
        if (p == left(parent)) // p is a left child
            return right(parent); // (right child might be null)
        else // p is a right child
            return left(parent); // (left child might be null)
    }

    /** Returns the number of children of Position p. */
    public int numChildren(Position<E> p)
    {
        int count=0;
        if (left(p) != null)
            count++;
        if (right(p) != null)
            count++;
        return count;
    }

    /** Returns an iterable collection of the Positions representing p's children. */
    public Iterable<Position<E>> children(Position<E> p)
    {
        List<Position<E>> snapshot = new ArrayList<>(2); // max capacity of 2
        if (left(p) != null)
            snapshot.add(left(p));
        if (right(p) != null)
            snapshot.add(right(p));
        return snapshot;
    }
}

```

```

//LinkedBinaryTree.java
import java.util.Iterator;
import java.util.Stack;
public class LinkedBinaryTree<E> extends AbstractBinaryTree<E>
{
    protected static class Node<E> implements Position<E>
    {
        private E element; // an element stored at this node
        private Node<E> parent; // a reference to the parent node (if any)
    }
}

```

```

private Node<E> left; // a reference to the left child (if any)
private Node<E> right; // a reference to the right child (if any)
/** Constructs a node with the given element and neighbors. */
public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild)
{
    element = e;
    parent = above;
    left = leftChild;
    right = rightChild;
} // accessor methods
public E getElement( )
{ return element; }
public Node<E> getParent( )
{ return parent; }
public Node<E> getLeft( )
{ return left; }
public Node<E> getRight( )
{ return right; } // update methods
public void setElement(E e)
{ element = e; }
public void setParent(Node<E> parentNode)
{ parent = parentNode; }
public void setLeft(Node<E> leftChild)
{ left = leftChild; }
public void setRight(Node<E> rightChild)
{ right = rightChild; }
} //----- end of nested Node class -----

protected Node<E> createNode(E e, Node<E> parent, Node<E> left, Node<E> right)
{ return new Node<E>(e, parent, left, right);}
protected Node<E> root = null; // root of the tree
private int size = 0; // number of nodes in the tree
public LinkedBinaryTree( ) { } // constructs an empty binary tree
protected Node<E> validate(Position<E> p) throws IllegalArgumentException
{
    if (!(p instanceof Node)) throw new IllegalArgumentException("Not valid position type");
    Node<E> node = (Node<E>) p; // safe cast
    if (node.getParent( ) == node) // our convention for defunct node
        throw new IllegalArgumentException("p is no longer in the tree");
    return node;
}
public int size( )
{ return size;}
public Position<E> root( )

```

```

    { return root;}
    public Position<E> parent(Position<E> p) throws IllegalArgumentException
    {
        Node<E> node = validate(p);
        return node.getParent( );
    }
    public Position<E> left(Position<E> p) throws IllegalArgumentException
    {
        Node<E> node = validate(p);
        return node.getLeft( );
    }
    public Position<E> right(Position<E> p) throws IllegalArgumentException
    {
        Node<E> node = validate(p);
        return node.getRight( );
    }
    // update methods supported by this class
    /** Places element e at the root of an empty tree and returns its new Position. */
    public Position<E> addRoot(E e) throws IllegalStateException
    {
        if (!isEmpty( )) throw new IllegalStateException("Tree is not empty");
        root = createNode(e, null, null, null);
        size = 1;
        return root;
    }
    /** Creates a new left child of Position p storing element e; returns its Position. */
    public Position<E> addLeft(Position<E> p, E e) throws IllegalArgumentException
    {
        Node<E> parent = validate(p);
        if (parent.getLeft( ) != null) throw new IllegalArgumentException("p already has a left
child");
        Node<E> child = createNode(e, parent, null, null);
        parent.setLeft(child);
        size++;
        return child;
    }
    /** Creates a new right child of Position p storing element e; returns its Position. */
    public Position<E> addRight(Position<E> p, E e) throws IllegalArgumentException
    {
        Node<E> parent = validate(p);
        if (parent.getRight( ) != null) throw new IllegalArgumentException("p already has a right
child");
        Node<E> child = createNode(e, parent, null, null);
        parent.setRight(child);
    }

```

```

        size++;
        return child;
    }
    /** Replaces the element at Position p with e and returns the replaced element. */
    public E set(Position<E> p, E e) throws IllegalArgumentException
    {
        Node<E> node = validate(p);
        E temp = node.getElement( );
        node.setElement(e);
        return temp;
    }
    /** Attaches trees t1 and t2 as left and right subtrees of external p. */
    public void attach(Position<E> p, LinkedBinaryTree<E> t1, LinkedBinaryTree<E> t2) throws
    IllegalArgumentException
    {
        Node<E> node = validate(p);
        if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");
        size += t1.size( ) + t2.size( );
        if (!t1.isEmpty( ))
        { // attach t1 as left subtree of node
            t1.root.setParent(node);
            node.setLeft(t1.root);
            t1.root = null;
            t1.size = 0;
        }
        if (!t2.isEmpty( ))
        { // attach t2 as right subtree of node
            t2.root.setParent(node);
            node.setRight(t2.root);
            t2.root = null;
            t2.size = 0;
        }
    }
    /** Removes the node at Position p and replaces it with its child, if any. */
    public E remove(Position<E> p) throws IllegalArgumentException
    {
        Node<E> node = validate(p);
        if (numChildren(p) == 2) throw new IllegalArgumentException("p has two children");
        Node<E> child = (node.getLeft( ) != null ? node.getLeft( ) : node.getRight( ) );
        if (child != null)
            child.setParent(node.getParent( )); // child's grandparent becomes its parent
        if (node == root)
            root = child; // child becomes root
        else
        {

```

```

        Node<E> parent = node.getParent( );
        if (node == parent.getLeft( ))
            parent.setLeft(child);
        else
            parent.setRight(child);
    }
    size--;
    E temp = node.getElement( );
    node.setElement(null); // help garbage collection
    node.setLeft(null);
    node.setRight(null);
    node.setParent(node); // our convention for defunct node
    return temp;
}
private class ElementIterator implements Iterator<E>
{
    Iterator<Position<E>> posIterator = positions( ).iterator( );
    public boolean hasNext( ) { return posIterator.hasNext( ); }
    public E next( ) { return posIterator.next( ).getElement( ); } // return element!
    public void remove( ) { posIterator.remove( ); }
}
/** Returns an iterator of the elements stored in the tree. */
public Iterator<E> iterator( ) { return new ElementIterator( ); }
// public void parenthesize(Tree<E> T, Position<E> p) //to print normal elements in
parenthesis form
Stack<E> op=new Stack<>();
public void parenthesize(Tree<E> T, Position<E> p) //to print expression in parenthesis form
{
    if (p.getElement()=="+"||p.getElement()=="-"||p.getElement()=="*"||p.getElement()=="/")
        op.push(p.getElement());
    else
        System.out.print(p.getElement());
    if (T.isInternal(p))
    {
        boolean firstTime = true;
        for (Position<E> c : T.children(p))
        {
            System.out.print( (firstTime ? "(" : op.pop()) );
            firstTime = false;
            parenthesize(T, c); // recur on child
        }
        System.out.print(")");
    }
}
}

```



```

public void traverseNodes(StringBuilder sb, String padding, String pointer, BinaryTree<E> T,
Position<E> p, boolean hasRightSibling)
{
    if (p != null)
    {
        sb.append("\n");
        sb.append(padding);
        sb.append(pointer);
        sb.append(p.getElement());

        StringBuilder paddingBuilder = new StringBuilder(padding);
        if (hasRightSibling) {
            paddingBuilder.append("| ");
        } else {
            paddingBuilder.append(" ");
        }

        String paddingForBoth = paddingBuilder.toString();
        String pointerRight = "└─";
        String pointerLeft = (T.right(p) != null) ? "├─" : "└─";

        traverseNodes(sb, paddingForBoth, pointerLeft, T, T.left(p), T.right(p) != null);
        traverseNodes(sb, paddingForBoth, pointerRight, T, T.right(p), false);
    }
}

public String traversePreOrder(BinaryTree<E> T, Position<E> p)
{
    if (p == null) {
        return "";
    }

    StringBuilder sb = new StringBuilder();
    sb.append(p.getElement());

    String pointerRight = "└─";
    String pointerLeft = (T.right(p) != null) ? "├─" : "└─";

    traverseNodes(sb, "", pointerLeft, T, T.left(p), T.right(p) != null);
    traverseNodes(sb, "", pointerRight, T, T.right(p), false);

    return sb.toString();
}

```

```

public void printTree(BinaryTree<E> T, Position<E> p)
{
    System.out.println(traversePreOrder(T,p));
}
}

```

```

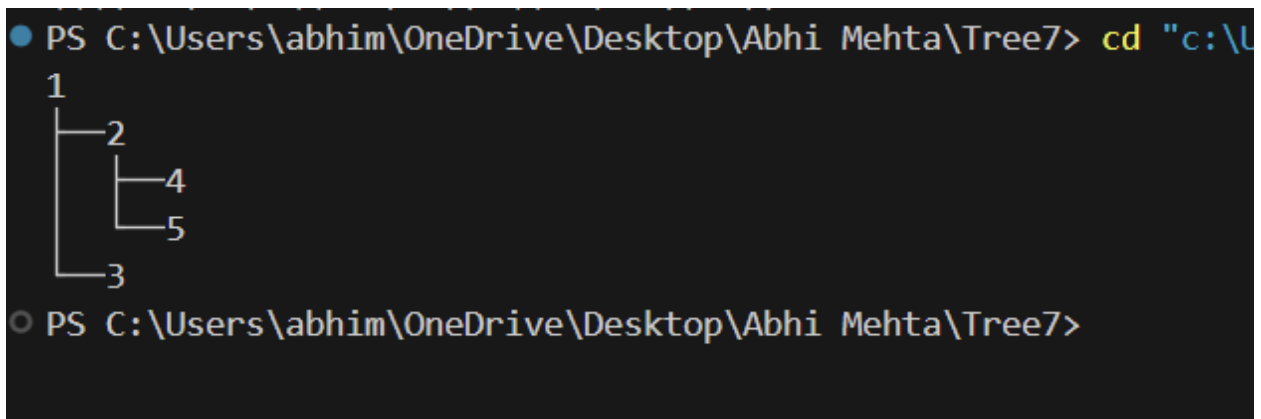
//Application4.java
public class Application4
{
    public static void main(String[] args)
    {
        LinkedBinaryTree<String> T = new LinkedBinaryTree<>();
        Position<String> root =T.addRoot("1");//Adding root value
        Position<String> node1=T.addLeft(root,"2");//Adding left node for node1
        Position<String> node2=T.addRight(root,"3");//Adding right node for node1

        Position<String> node1l=T.addLeft(node1,"4");//Adding Left node for node1
        Position<String> node1r=T.addRight(node1,"5");//Adding right node for node1

        T.printTree(T, root);//Method already defined in LinkedTree
    }
}

```

Output-



```

PS C:\Users\abhim\OneDrive\Desktop\Abhi Mehta\Tree7> cd "c:\U
1
|
├── 2
│   ├── 4
│   └── 5
└── 3
PS C:\Users\abhim\OneDrive\Desktop\Abhi Mehta\Tree7>

```

Errors Encountered-

- 1.Errors occurred due to neglect of imports preventing the implementation of functions reliant on those modules
- 2.Errors occurred due a validate statement in MyTree and LinkedTree and position validation did not match for which it showed errors.
- 3.Addressing errors involved ensuring the passage of accurate parameters with the correct data types according to the function's requirements.
- 4.Encountering basic issues such as indentation and semicolon errors to some degree during the process
- 5.Error occurred while adding the root because of Node and TreeNode being different so validation error led to this error.
- 6.Error occurred when input was not getting stored by the program and root and child value were not displayed.

Learnings-

I have acquired a comprehensive understanding of the intricacies associated with implementing a tree data structure. This comprehension extends to the nuanced roles played by fundamental components like the root and nodes. In contrast to binary trees, the introduction of the capability for each node to accommodate multiple child nodes, each with its distinct set of children, introduces a heightened level of complexity, demanding thorough scrutiny and analysis. The diversity in implementation methods, utilizing various data structures, was apparent. Applying the structure for particular applications such as indenting, disc operation, parentheses and tree display provided a great opportunity to learn about trees, binary trees as well as linked binary trees for the applications. I also learnt about tree traversal methods necessary to implement the given applications such as pre-order, postorder and inorder traversal which has been implemented in the code.

Conclusion-

In conclusion, the encountered errors in the implementation process were diverse, ranging from issues with imports to challenges in parameter validation. Addressing these errors required a meticulous approach, ensuring accurate parameter types and correct data passing. Basic syntax errors, such as indentation and semicolon issues, were also part of the learning experience.

A key learning point was the importance of a comprehensive understanding of tree data structures, considering the distinctive roles of root, nodes, and leaves. The flexibility introduced by allowing nodes to have multiple children added complexity, requiring careful analysis. The implementation of various applications, including indenting, disk operation, parentheses representation, and tree display, provided practical insights into applying tree structures for specific purposes. Additionally, learning about tree traversal methods, such as pre-order, post-order, and in-order, was essential for implementing the given applications.

Overall, this experience deepened my knowledge of tree structures, binary trees, and linked binary trees, as well as their applications and traversal methods. It underscored the importance of attention to detail and a methodical approach in working with complex data structures.