Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 4 REPORT

AHMET YASIR NACAK
161044042

Course Assistant: Mehmet Burak KOCA

1 INTRODUCTION

1.1 Problem Definition

In this homework, the problem is to create two tree structures. One of them is an extension of the Binary Tree class that represents a general tree in binary tree form. After representing this tree, another requirement is to add traversal and search operations that runs on this binary tree, but the search and traverse logic is acts like the tree is a general tree. The second data structure is a multidimensional binary search tree. This tree has a certain amount of data for each of its nodes and the when there is an item being added to this tree, the comparison takes the depth as the number of element and this operation wraps around when the depth passes the amount of dimension.

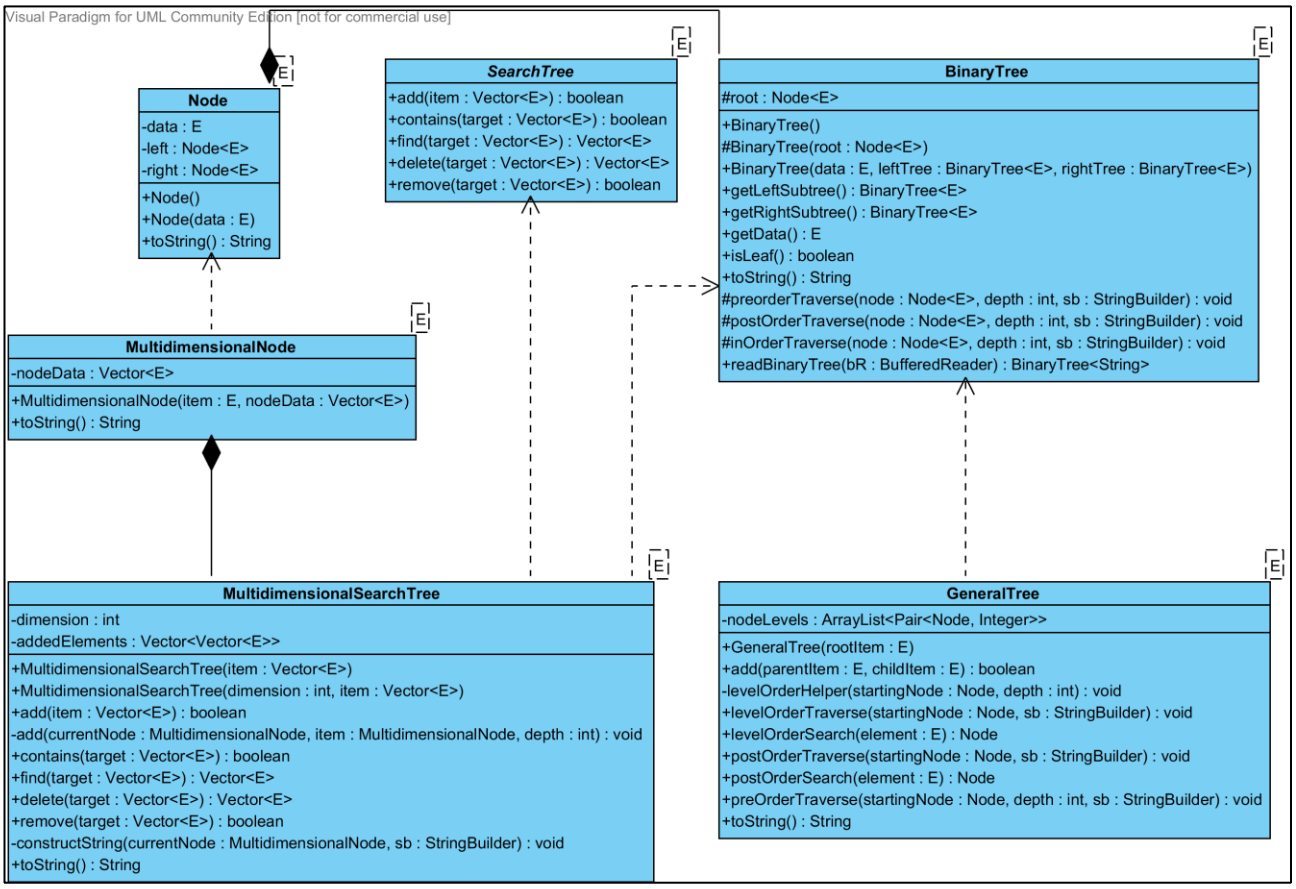
1.2 System Requirements

To create a general tree that is being represented as a binary tree, we need to create a way to hold more than one child in a binary tree. To do that, we use the common approach and assume that each right child is the sibling of the node and each left child is the child of the node. This guarantees an understandable binary tree structure that represents a general tree. After that, in order to create traversal operations that acts as the tree is a general tree, we need to analyze the way that a traversal algorithm acts on a general tree and replicate it in our own tree structure.

To create a multidimensional binary search tree that has more than one data for each node, we need to modify the node structure but since damaging the main node class is not a good approach, we can extend it in our multidimensional tree class and modify it that way. After that, we need to create an operation that can add a list of elements that is wrapped in a data structure. To do that, we first need to create a multidimensional node out of that list of elements and find its correct position in our tree. This procedure works as if the comparison is being made with an element in the tree is at top of the tree, we compare their first elements, if the comparison is being made with one of the children of the root, we compare their second value and so on. This comparison wraps around itself once the depth is same as the dimension of the tree.

2 METHOD

2.1 Class Diagram


Visual Paradigm for UML Community Edition [not for commercial use]
E
Node
-data : E
-left : Node<E>
-right : Node<E>
+Node()
+Node(data : E)
+toString() : String
E
SearchTree
+add(item : Vector<E>) : boolean
+contains(target : Vector<E>) : boolean
+find(target : Vector<E>) : Vector<E>
+delete(target : Vector<E>) : Vector<E>
+remove(target : Vector<E>) : boolean
E
BinaryTree
#root : Node<E>
+BinaryTree()
#BinaryTree(root : Node<E>)
+BinaryTree(data : E, leftTree : BinaryTree<E>, rightTree : BinaryTree<E>)
+getLeftSubtree() : BinaryTree<E>
+getRightSubtree() : BinaryTree<E>
+getData() : E
+isLeaf() : boolean
+toString() : String
#preorderTraverse(node : Node<E>, depth : int, sb : StringBuilder) : void
#postOrderTraverse(node : Node<E>, depth : int, sb : StringBuilder) : void
#inOrderTraverse(node : Node<E>, depth : int, sb : StringBuilder) : void
+readBinaryTree(bR : BufferedReader) : BinaryTree<String>
E
MultidimensionalNode
-nodeData : Vector<E>
+MultidimensionalNode(item : E, nodeData : Vector<E>)
+toString() : String
E
MultidimensionalSearchTree
-dimension : int
-addedElements : Vector<Vector<E>>
+MultidimensionalSearchTree(item : Vector<E>)
+MultidimensionalSearchTree(dimension : int, item : Vector<E>)
+add(item : Vector<E>) : boolean
-add(currentNode : MultidimensionalNode, item : MultidimensionalNode, depth : int) : void
+contains(target : Vector<E>) : boolean
+find(target : Vector<E>) : Vector<E>
+delete(target : Vector<E>) : Vector<E>
+remove(target : Vector<E>) : boolean
-constructString(currentNode : MultidimensionalNode, sb : StringBuilder) : void
+toString() : String
E
GeneralTree
-nodeLevels : ArrayList<Pair<Node, Integer>>
+GeneralTree(rootItem : E)
+add(parentItem : E, childItem : E) : boolean
-levelOrderHelper(startingNode : Node, depth : int) : void
+levelOrderTraverse(startingNode : Node, sb : StringBuilder) : void
+levelOrderSearch(element : E) : Node
+postOrderTraverse(startingNode : Node, sb : StringBuilder) : void
+postOrderSearch(element : E) : Node
+preOrderTraverse(startingNode : Node, depth : int, sb : StringBuilder) : void
+toString() : String

2.2 Problem Solution Approach

2.2.1 Binary Tree and Basic Node

The binary tree class and its inner node class is a premade class that can be found in our book. This class provides the user a simple binary tree class that has nodes in it. This structure has subtree methods and simple traversal methods. All of these methods already work correctly for a basic binary tree but since our purpose is to create two tree structures that are not simple binary trees, we need to override some methods and write new ones. There is also another missing part of this class and that is the add method. Since the simple binary class has no strategy for adding new nodes, this method is not implemented. In this binary class, there is a basic node class that is generic and has its data, its left child and its right child.

2.2.2 General Tree

The general tree class inherits from binary tree class and acts a binary tree representation of a general tree. This class needs to add an add method that takes a parent item and the item that is needed to be added to the tree. This method first looks for the given parent if it is in the list and if this control is a success, it then looks at the first left child of the parent. If this node is empty, it adds it there. If not, it looks all the right children of the first left child of the parent node. When the method finds an empty node in there, it adds the given children to that position. Running time of this method is dependent on the number of children of the given parent, the worst case running time is O(N). After the addition method, there is the traversal and search methods. Since the counterparts of these methods in the parent class in configured for a simple binary tree, our general tree structure can not utilize this method and it rewrites/overrides it. There are 2 search methods and 1 traversal method we need to write. First search method is level order search. In a general tree, all the siblings are in the same level but in a binary tree representation, all the right subtree of a node is its same generation so to do a level order search, we need to look for the elements in a right to left manner, always looking for the right children and then advance to the left child of those nodes and look for their right children. This creates a recursive structure and its running time O(n^2). After the level order search, we need to create a post order search. The post order search goes for the leftmost node of the tree and looks for its siblings, then its parents. After that the search looks for the sibling of the parent of the leftmost node and looks for its children first and the search goes on like that. To achieve this search in a binary tree representation of a general tree, we need to create a recursive search structure that controls if the left node of our current node exists, calls the method with that node, controls the current node and finally looks for any of the right nodes and calls the method with that node as well. The time complexity of this search is O(n^2) because of the 2 recursive calls in the method. And finally, we need to create a preorder traversal method that provides the user a text representation of the tree. This method is a basic modification of the post order search and takes the same amount of the time to be completed.

2.2.3 Search Tree

Search tree structure is an interface with simple search tree methods given in it. Since this data structure is not suitable for a multidimensional tree structure, we need to modify the signatures of its method. In a real-life scenario, this could get done in a more software engineer manner but since the project we are creating does not require the search tree interface to be used in anywhere else, we can modify it.

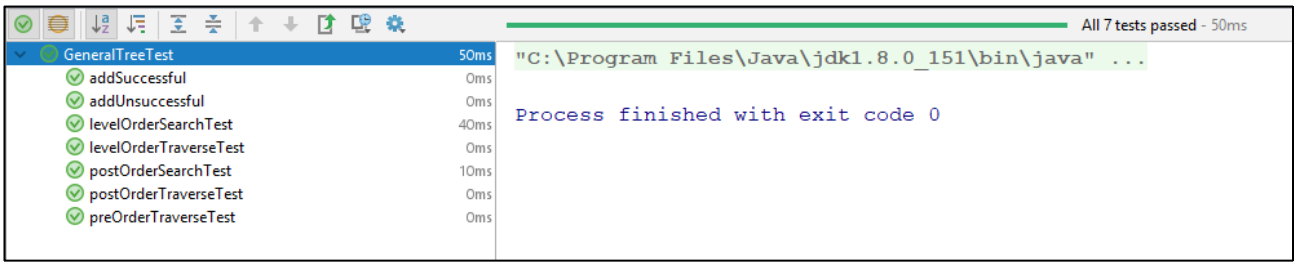
2.2.4 Multidimensional Tree and Multidimensional Node

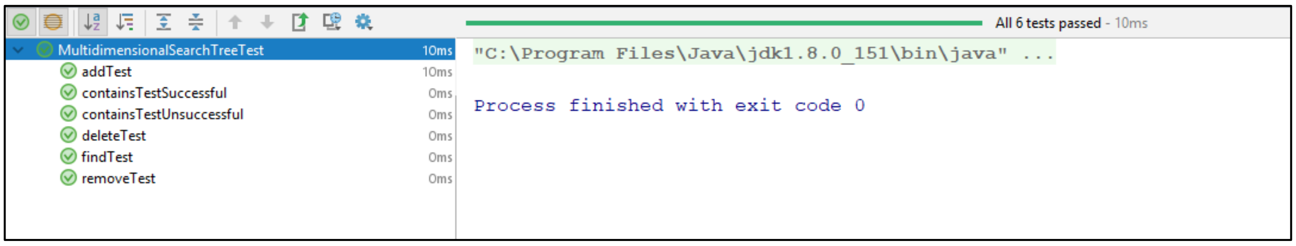
The multidimensional tree structure is a both a binary tree and a search tree that means that it is a BST. To create a data structure like this, we need to implement the search tree interface and extend the binary tree class. But to achieve a multidimensional structure, we need to modify the basic node class of the binary tree. We can crate an inner class that extends the basic node and adds a vector of generic type that can represent all the elements in a node. We also need to create a constructor for this class, so it can be initialized correctly. After creating the node class, we need to implement all the methods that are in the Search Tree interface. The most important methods of this class are the add and remove methods. To add an item to the tree, we need to take a list of elements and do a multidimensional comparison to find the correct position of this item. Multidimensional comparison works as the **depth *modulo* dimension** is the one being compared to decide whether to go left or right in the tree for a correct position. To remove an item from the list, I decided to hold a list of nodes that have added to the tree in their specific order, call this nodeList, and whenever the user wants to remove an element from the tree, the method firstly removes the element from nodeList and reconstructs the tree without that node. This approach keeps the BST property of the tree and removes an element from the list. This complexity of addition and removal is O(n).

3 RESULT

3.1 Test Cases

Here are the results of the test codes. Each of them is described in the Javadoc.


```
All 7 tests passed - 50ms
GeneralTreeTest            50ms
  addSuccessful             0ms
  addUnsuccessful           0ms
  levelOrderSearchTest     40ms
  levelOrderTraverseTest    0ms
  postOrderSearchTest      10ms
  postOrderTraverseTest     0ms
  preOrderTraverseTest      0ms

"C:\Program Files\Java\jdk1.8.0_151\bin\java" ...

Process finished with exit code 0
```

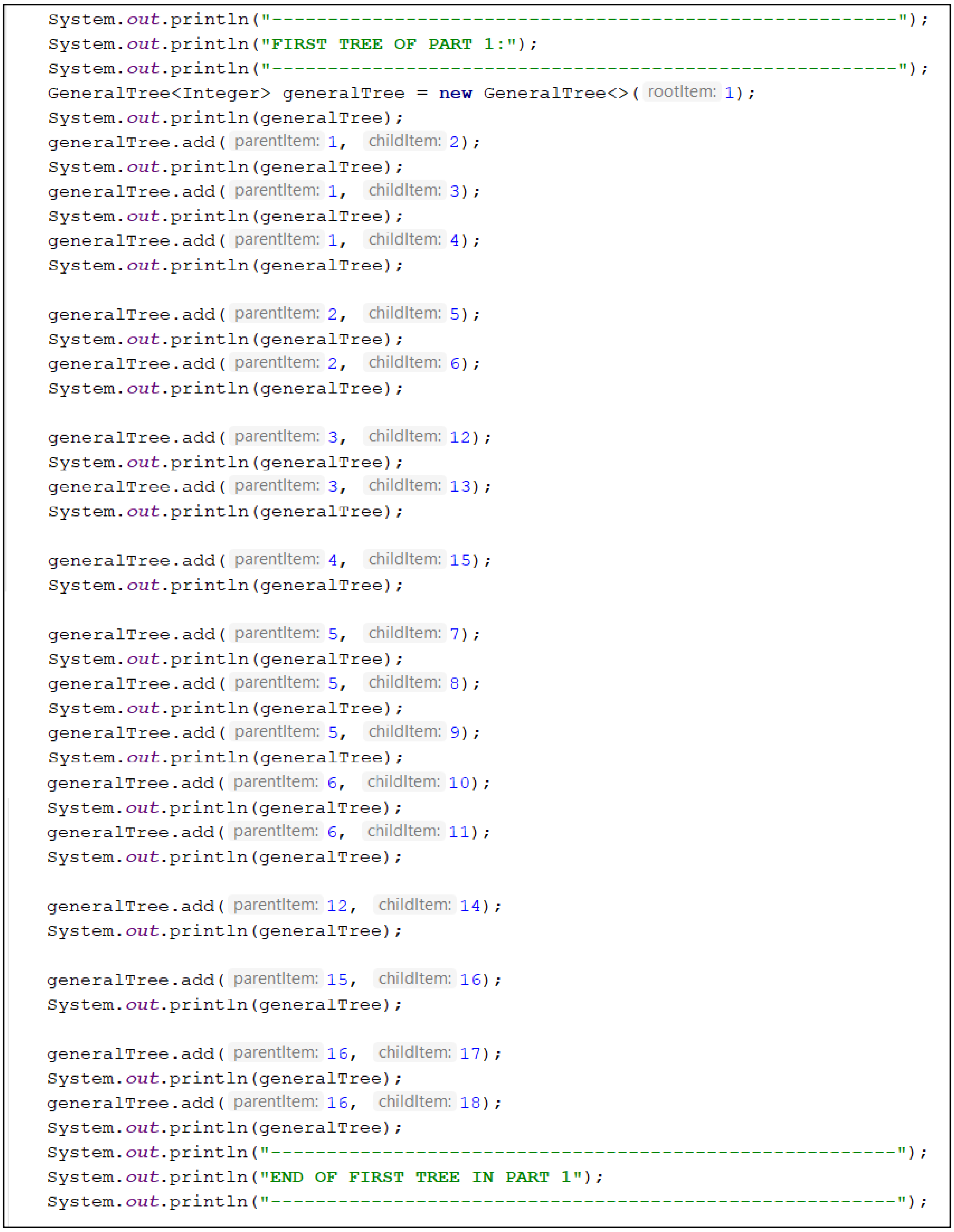

```
All 6 tests passed - 10ms
MultidimensionalSearchTreeTest  10ms
  addTest                       10ms
  containsTestSuccessful         0ms
  containsTestUnsuccessful       0ms
  deleteTest                     0ms
  findTest                       0ms
  removeTest                     0ms

"C:\Program Files\Java\jdk1.8.0_151\bin\java" ...

Process finished with exit code 0
```

3.2 Running Results

These are the sample test codes

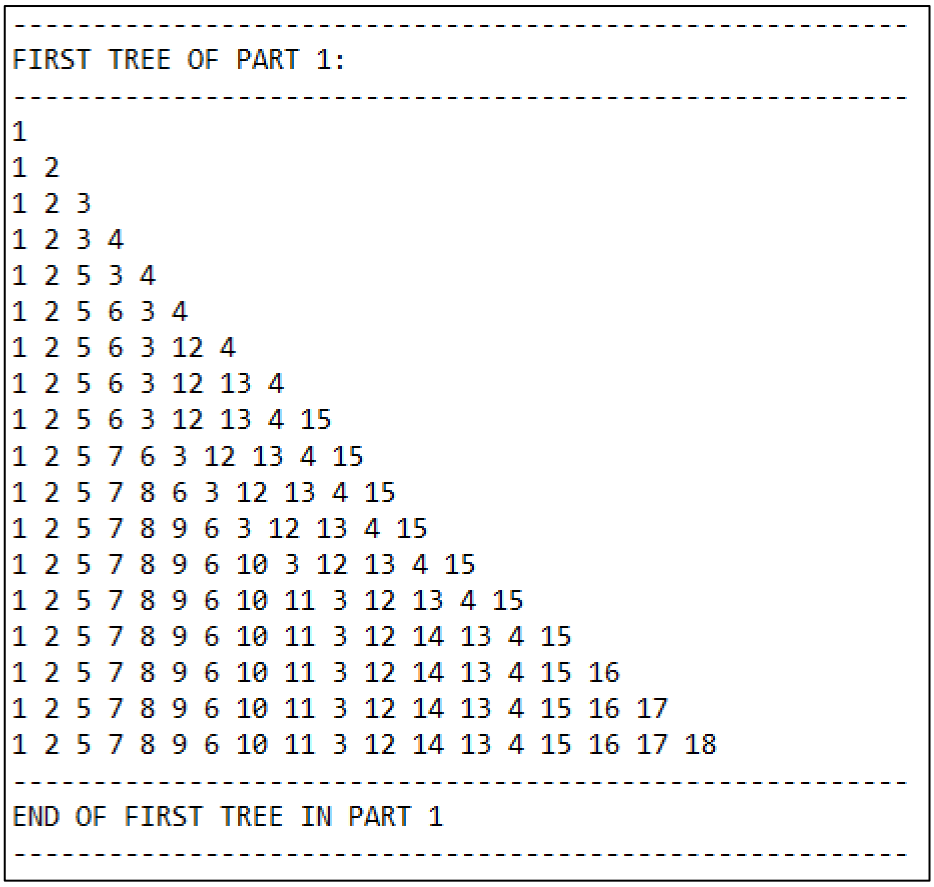
General Tree 1:

CODE:



```
System.out.println("--------------------------------------------------------");
System.out.println("FIRST TREE OF PART 1:");
System.out.println("--------------------------------------------------------");
GeneralTree<Integer> generalTree = new GeneralTree<>( rootItem: 1);
System.out.println(generalTree);
generalTree.add( parentItem: 1,  childItem: 2);
System.out.println(generalTree);
generalTree.add( parentItem: 1,  childItem: 3);
System.out.println(generalTree);
generalTree.add( parentItem: 1,  childItem: 4);
System.out.println(generalTree);

generalTree.add( parentItem: 2,  childItem: 5);
System.out.println(generalTree);
generalTree.add( parentItem: 2,  childItem: 6);
System.out.println(generalTree);

generalTree.add( parentItem: 3,  childItem: 12);
System.out.println(generalTree);
generalTree.add( parentItem: 3,  childItem: 13);
System.out.println(generalTree);

generalTree.add( parentItem: 4,  childItem: 15);
System.out.println(generalTree);

generalTree.add( parentItem: 5,  childItem: 7);
System.out.println(generalTree);
generalTree.add( parentItem: 5,  childItem: 8);
System.out.println(generalTree);
generalTree.add( parentItem: 5,  childItem: 9);
System.out.println(generalTree);
generalTree.add( parentItem: 6,  childItem: 10);
System.out.println(generalTree);
generalTree.add( parentItem: 6,  childItem: 11);
System.out.println(generalTree);

generalTree.add( parentItem: 12,  childItem: 14);
System.out.println(generalTree);

generalTree.add( parentItem: 15,  childItem: 16);
System.out.println(generalTree);

generalTree.add( parentItem: 16,  childItem: 17);
System.out.println(generalTree);
generalTree.add( parentItem: 16,  childItem: 18);
System.out.println(generalTree);
System.out.println("--------------------------------------------------------");
System.out.println("END OF FIRST TREE IN PART 1");
System.out.println("--------------------------------------------------------");
```

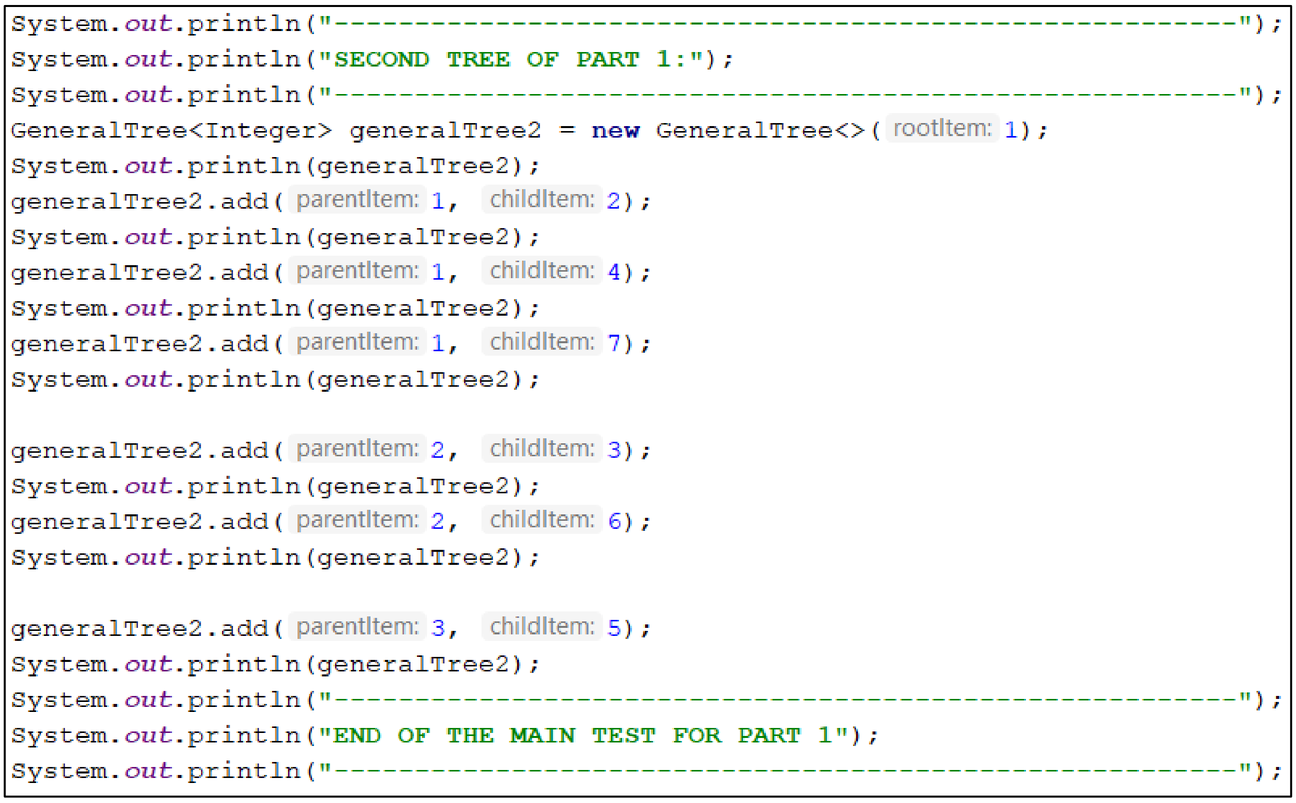
OUTPUT:



```
--------------------------------------------------------
FIRST TREE OF PART 1:
--------------------------------------------------------
1
1 2
1 2 3
1 2 3 4
1 2 5 3 4
1 2 5 6 3 4
1 2 5 6 3 12 4
1 2 5 6 3 12 13 4
1 2 5 6 3 12 13 4 15
1 2 5 7 6 3 12 13 4 15
1 2 5 7 8 6 3 12 13 4 15
1 2 5 7 8 9 6 3 12 13 4 15
1 2 5 7 8 9 6 10 3 12 13 4 15
1 2 5 7 8 9 6 10 11 3 12 13 4 15
1 2 5 7 8 9 6 10 11 3 12 14 13 4 15
1 2 5 7 8 9 6 10 11 3 12 14 13 4 15 16
1 2 5 7 8 9 6 10 11 3 12 14 13 4 15 16 17
1 2 5 7 8 9 6 10 11 3 12 14 13 4 15 16 17 18
--------------------------------------------------------
END OF FIRST TREE IN PART 1
--------------------------------------------------------
```

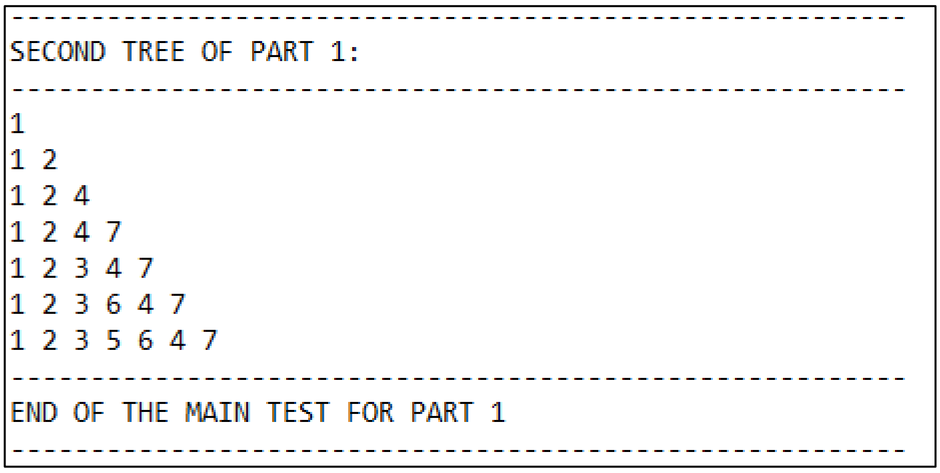
General Tree 2:

CODE:



```java
System.out.println("--------------------------------------------------------");
System.out.println("SECOND TREE OF PART 1:");
System.out.println("--------------------------------------------------------");
GeneralTree<Integer> generalTree2 = new GeneralTree<>( rootItem: 1);
System.out.println(generalTree2);
generalTree2.add( parentItem: 1,  childItem: 2);
System.out.println(generalTree2);
generalTree2.add( parentItem: 1,  childItem: 4);
System.out.println(generalTree2);
generalTree2.add( parentItem: 1,  childItem: 7);
System.out.println(generalTree2);

generalTree2.add( parentItem: 2,  childItem: 3);
System.out.println(generalTree2);
generalTree2.add( parentItem: 2,  childItem: 6);
System.out.println(generalTree2);

generalTree2.add( parentItem: 3,  childItem: 5);
System.out.println(generalTree2);
System.out.println("--------------------------------------------------------");
System.out.println("END OF THE MAIN TEST FOR PART 1");
System.out.println("--------------------------------------------------------");
```

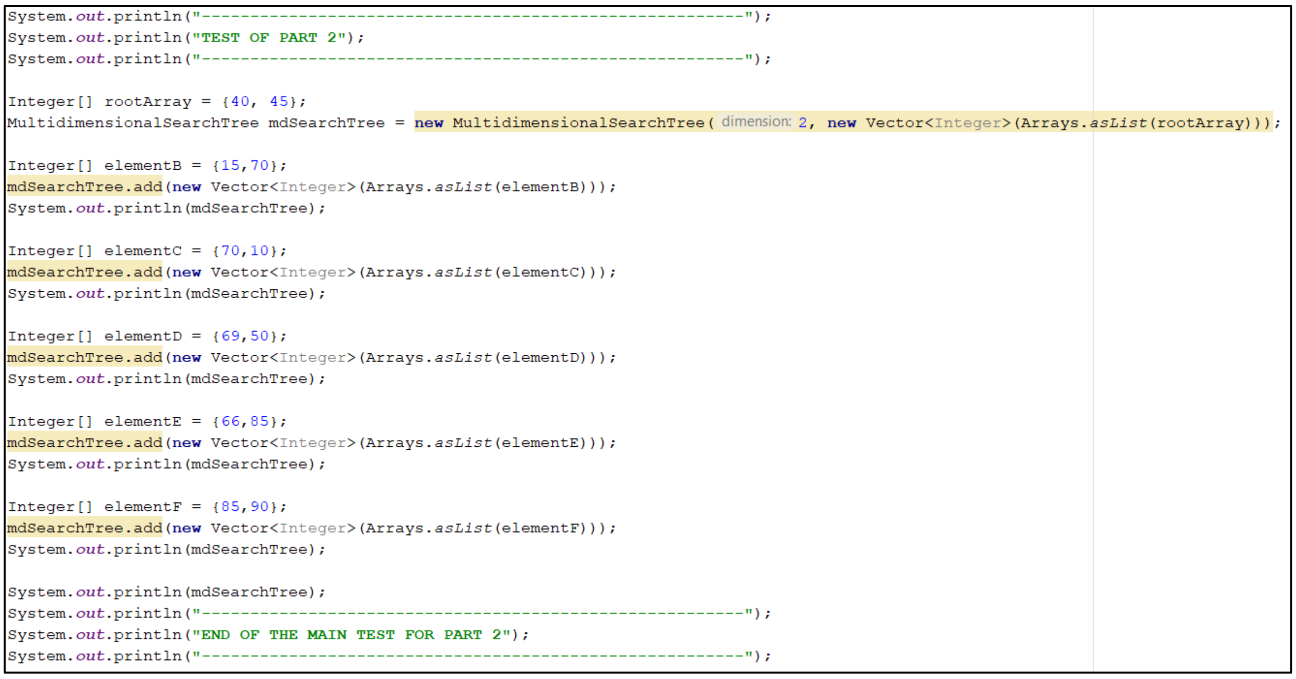
OUTPUT:



```
--------------------------------------------------------
SECOND TREE OF PART 1:
--------------------------------------------------------
1
1 2
1 2 4
1 2 4 7
1 2 3 4 7
1 2 3 6 4 7
1 2 3 5 6 4 7
--------------------------------------------------------
END OF THE MAIN TEST FOR PART 1
--------------------------------------------------------
```

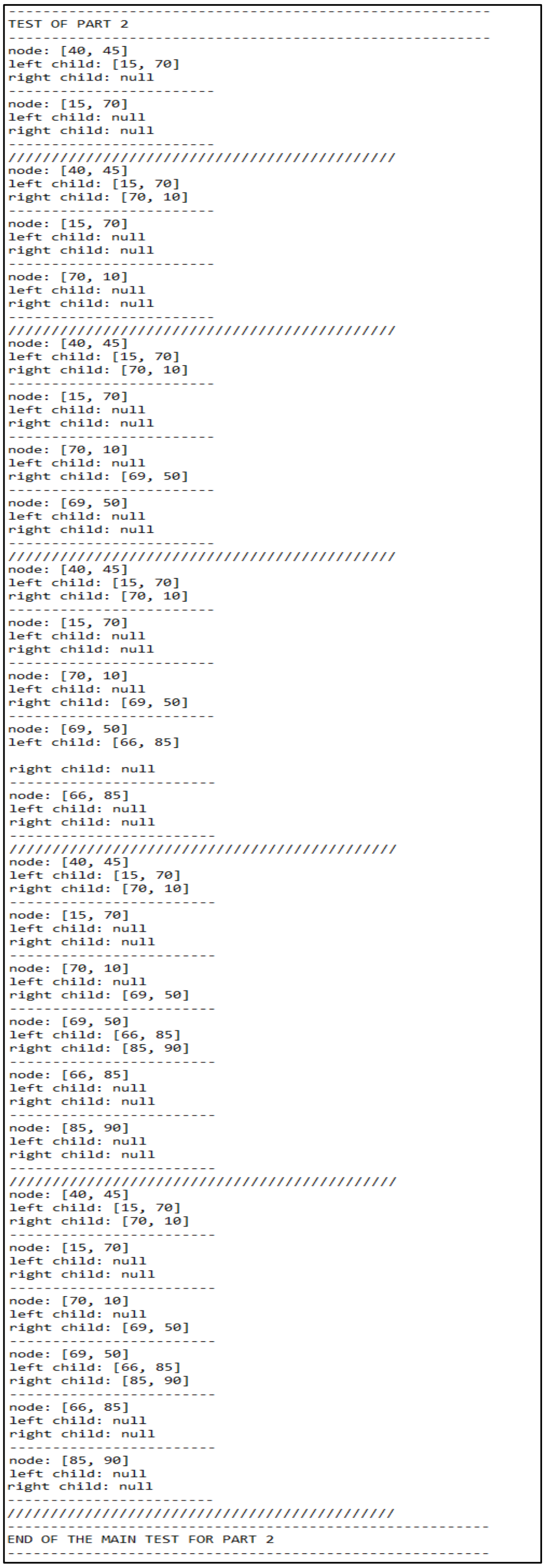
Multidimensional Tree:

CODE:



```java
System.out.println("--------------------------------------------------------");
System.out.println("TEST OF PART 2");
System.out.println("--------------------------------------------------------");

Integer[] rootArray = {40, 45};
MultidimensionalSearchTree mdSearchTree = new MultidimensionalSearchTree( dimension: 2, new Vector<Integer>(Arrays.asList(rootArray)));

Integer[] elementB = {15,70};
mdSearchTree.add(new Vector<Integer>(Arrays.asList(elementB)));
System.out.println(mdSearchTree);

Integer[] elementC = {70,10};
mdSearchTree.add(new Vector<Integer>(Arrays.asList(elementC)));
System.out.println(mdSearchTree);

Integer[] elementD = {69,50};
mdSearchTree.add(new Vector<Integer>(Arrays.asList(elementD)));
System.out.println(mdSearchTree);

Integer[] elementE = {66,85};
mdSearchTree.add(new Vector<Integer>(Arrays.asList(elementE)));
System.out.println(mdSearchTree);

Integer[] elementF = {85,90};
mdSearchTree.add(new Vector<Integer>(Arrays.asList(elementF)));
System.out.println(mdSearchTree);

System.out.println(mdSearchTree);
System.out.println("--------------------------------------------------------");
System.out.println("END OF THE MAIN TEST FOR PART 2");
System.out.println("--------------------------------------------------------");
```

OUTPUT:



```
--------------------------------------------------------
TEST OF PART 2
--------------------------------------------------------
node: [40, 45]
left child: [15, 70]
right child: null
------------------------
node: [15, 70]
left child: null
right child: null
------------------------
/////////////////////////////////////////////
node: [40, 45]
left child: [15, 70]
right child: [70, 10]
------------------------
node: [15, 70]
left child: null
right child: null
------------------------
node: [70, 10]
left child: null
right child: null
------------------------
/////////////////////////////////////////////
node: [40, 45]
left child: [15, 70]
right child: [70, 10]
------------------------
node: [15, 70]
left child: null
right child: null
------------------------
node: [70, 10]
left child: null
right child: [69, 50]
------------------------
node: [69, 50]
left child: null
right child: null
------------------------
/////////////////////////////////////////////
node: [40, 45]
left child: [15, 70]
right child: [70, 10]
------------------------
node: [15, 70]
left child: null
right child: null
------------------------
node: [70, 10]
left child: null
right child: [69, 50]
------------------------
node: [69, 50]
left child: [66, 85]

right child: null
------------------------
node: [66, 85]
left child: null
right child: null
------------------------
/////////////////////////////////////////////
node: [40, 45]
left child: [15, 70]
right child: [70, 10]
------------------------
node: [15, 70]
left child: null
right child: null
------------------------
node: [70, 10]
left child: null
right child: [69, 50]
------------------------
node: [69, 50]
left child: [66, 85]
right child: [85, 90]
------------------------
node: [66, 85]
left child: null
right child: null
------------------------
node: [85, 90]
left child: null
right child: null
------------------------
/////////////////////////////////////////////
node: [40, 45]
left child: [15, 70]
right child: [70, 10]
------------------------
node: [15, 70]
left child: null
right child: null
------------------------
node: [70, 10]
left child: null
right child: [69, 50]
------------------------
node: [69, 50]
left child: [66, 85]
right child: [85, 90]
------------------------
node: [66, 85]
left child: null
right child: null
------------------------
node: [85, 90]
left child: null
right child: null
------------------------
/////////////////////////////////////////////
--------------------------------------------------------
END OF THE MAIN TEST FOR PART 2
--------------------------------------------------------
```