Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 4 REPORT

CENGİZ TOPRAK
161044087

Course Assistant:

BURAK KOCA

1 INTRODUCTION

1.1 Problem Definition

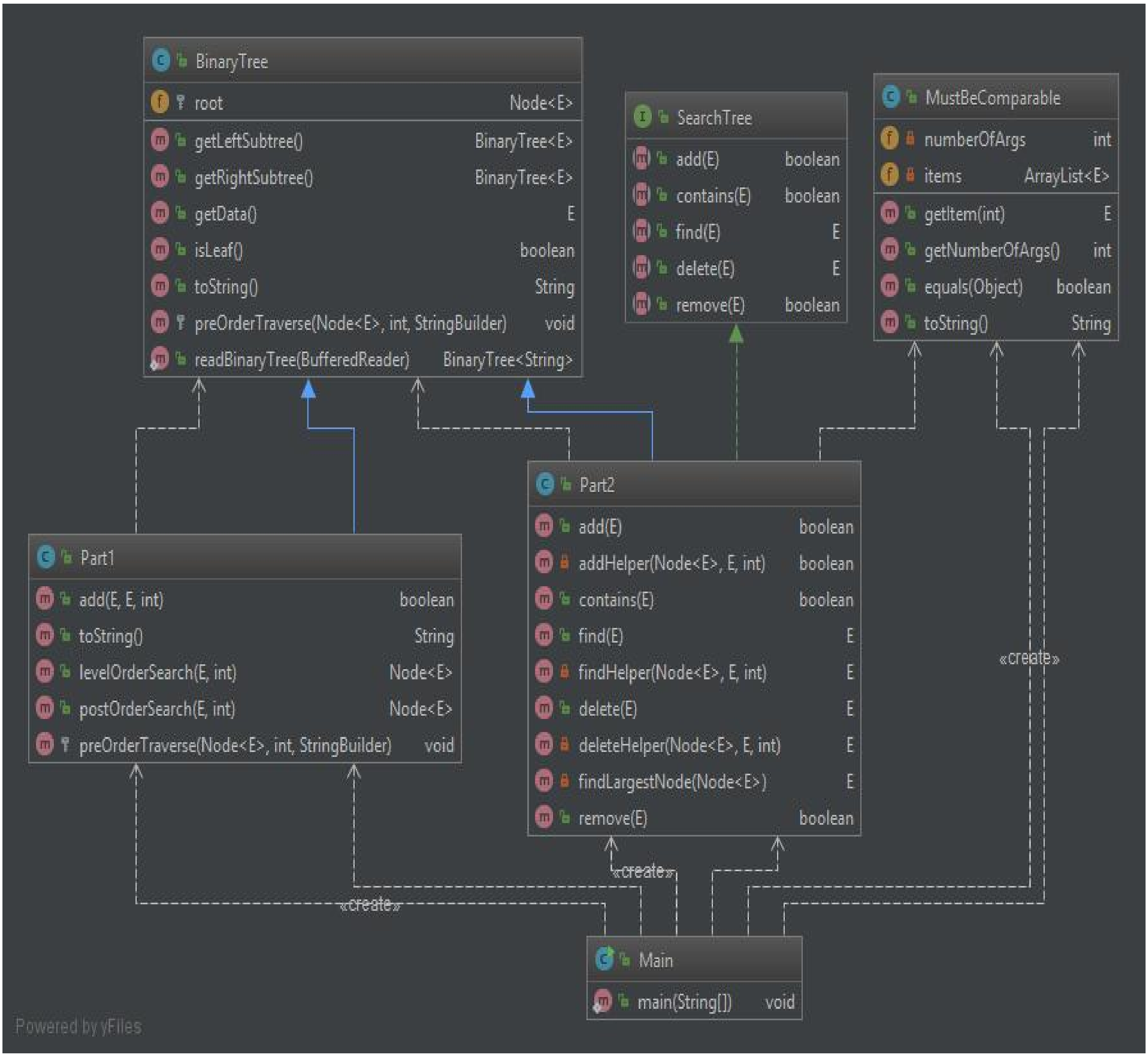
The problem is to understand well consept of binary tree and , to implement general tree by using provided binary tree class ,to implement others binary searh algorithm like level order search,post order search and preorder traverse, and to implement multi dimensionle tree by using provided binary tree class and search tree interface.

1.2 System Requirements

It is needed a binary tree class for implementing general tree in part 1 and a binary tree class , a search tree interface and a class that has a constructor , can take multiple arguments , for implementing multi dimensionel tree in part2

2 METHOD

2.1 Class Diagrams



Generic part1 class extends binary tree class ,and implements add method and some search methods(level order search and post order search) which add method can use one of them when find reference of parent node.

Generic part2 class extends binary tree , implements search tree.

MustBeComparable class is created for part2 class and extends Comparable.

2.2 Problem Solution Approach

For implementing general tree ,it is needed a binary tree class.

Generic part1 class extends binary tree class , override pre order traverse method and implement some useful methods(add,level order search) whose perform operations on the general tree.

- boolean add(E parent,E child,int option) : add child to tree and return true if parent is on the tree ,false otherwise.The option argument determine that method used level order search or pre order search when was finding the parent.
- Node<E> levelOrderSearch(E item) : search item level by level with respect to general tree.if it finds item ,it returns reference of item in tree,null otherwise.
- Node<E> preOrderSearch(E item) : search item according to preorder algorithm with respect to general tree.if it finds item ,it returns reference of item in tree,null otherwise.

For implementing multi dimensionel tree,it is needed a binary tree class ,a search tree interface and a MustBeComparable class.

Generic part2 class extends binary tree class and implements search tree interface for implementing multi dimensionel tree.

MustBeComparable class extends Comparable ,and has special constructor that has multiple arguments like MustBeComparable(E item1,E item2,...) In the otherwords MustBeComparable(E... args).

Generic item in the part2 class must be MustbeComparable class or extend MustbeComparable class .

- boolean add(E item) : add item to tree and return true if item is not in tree,false otherwise. The add method use another private addhelper method for traversing tree recursively.
- E find(E item) : return reference of item in the tree if item is present in the tree ,null otherwise. The find method use another private findHelper method for traversing tree recursively.
- boolean contains(E item) : return true in the tree if item is present in the tree ,false otherwise. The contains method just use find method inside.

- E delete(E item) : delete item and return reference of item in the tree if item is present in the tree ,null otherwise. The delete method use other private deleteHelper method for traversing tree recursively and use another private helper method named findLargestNode so that structure of mds tree is not break down when item is deleted.
- boolean remove(E item) : delete item and return true in the tree if item is present in the tree ,false otherwise. The remove method just use delete method inside.

3 RESULT

3.1 Test Cases

1) PART1

- Adding child that has no parent in tree is failed.(add method return type boolean)
- Adding child that has parent in the tree is succesfull.
- Searching item which is present in tree is succesfull in level order or post order search.(level and post order method return type reference)
- Searching item which is not present in tree is failed in level order or post order search.(level and post order method)

2) PART2

- Adding item that is present already in tree is failed .(add method return type boolean)
- Adding item that is not present in tree is successfull.
- Deleting item is present in the tree is successfull.(delete method return type a reference of deleted item)
- Deleting item does not exist in the tree is failed.
- Removing item is present in the tree is successfull.(remove method return type boolean)
- Removing item does not exist in the tree is failed.
- Finding item is present in the tree is successfull.(find method return type a reference of found item)
- Finding item does not exist in the tree is failed.

- Finding item is present in the tree is successfull.(contains method return type boolean)
- Finding item does not exist in the tree is failed.

failed : method return false or null according to return type of method.

successfull: method return true or a reference of deleted or found items in tree according to return type of method.

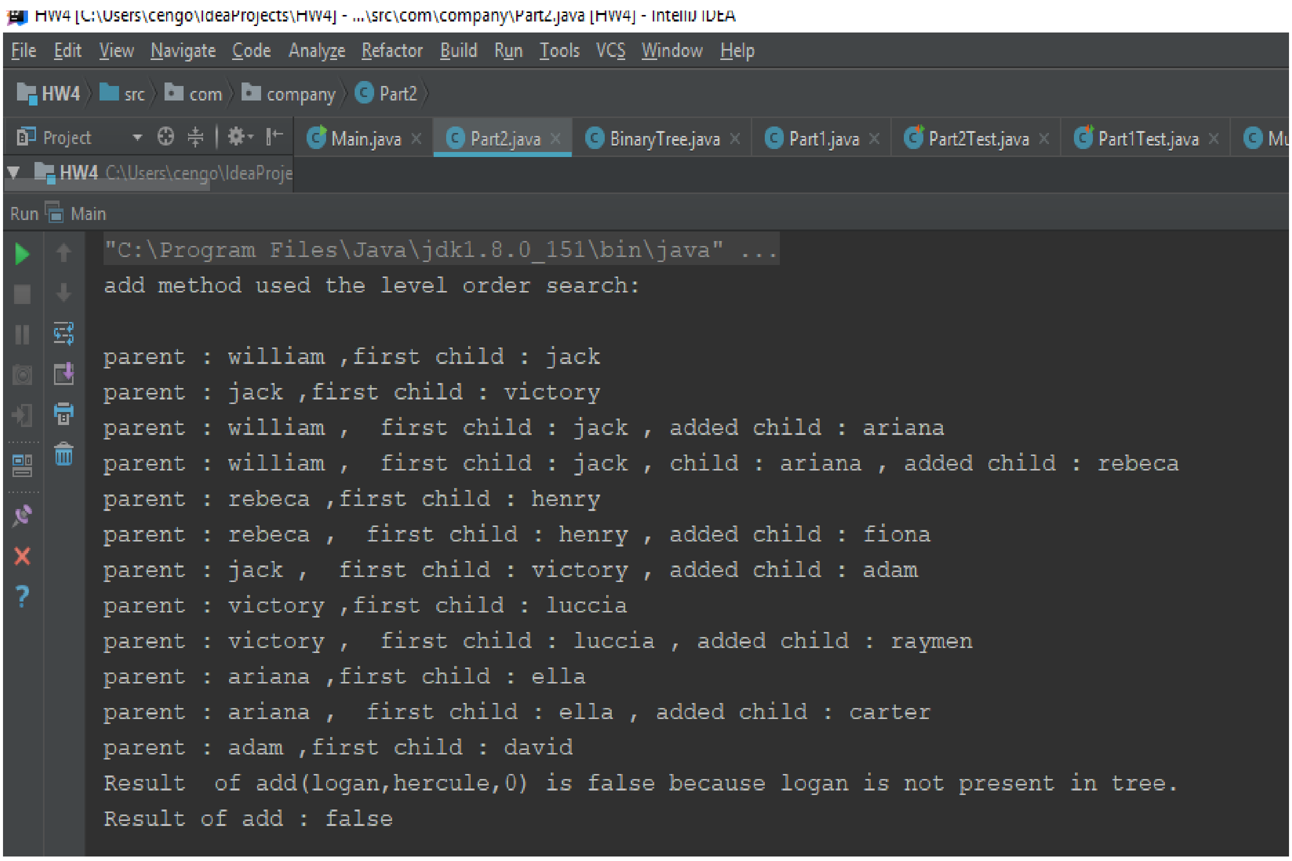
3.2 Unit Tests

It is writen test classes for part1 and part2 ,and can be seen result of unit test of methods by running test classes.

3.3 Running Results

PART1 RESULTS :

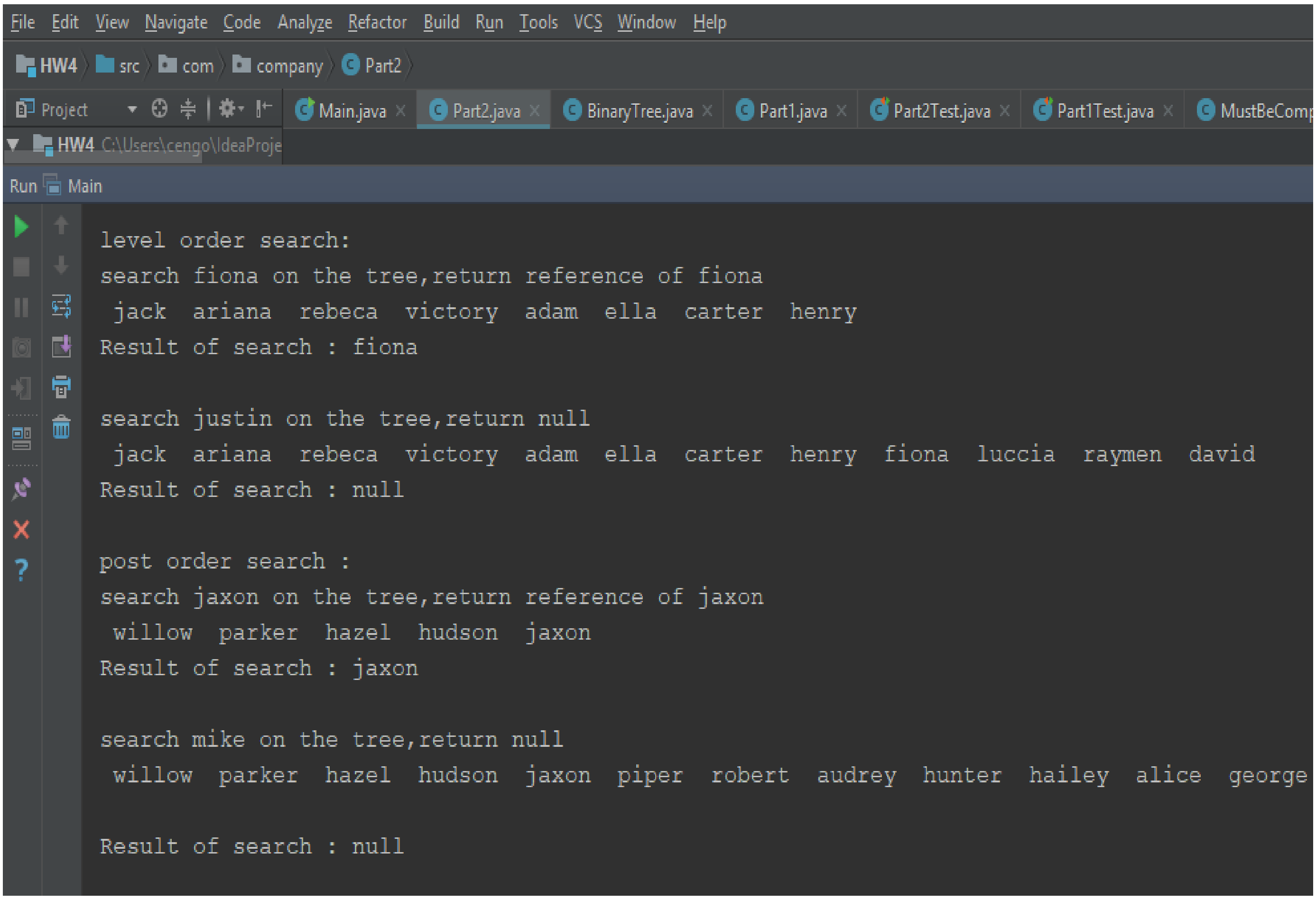
add method outputs.



As seen above,add method return false if parent is not in the tree such as,add(logan,hercule,0) and seen that parent has how many childs.

If parent has no child ,it is added child to leftside of parent,if parent has child or chids it is added childs to rightside of last child.

LevelOrderSearch and postOrderSearch methods output:



As show above,when search fiona on tree by using level order search fiona compares ,in turn,jack ,ariana,rebeca in the first level and then compares with victory,adam,ella,carter, henry and finally fiona in second level.Thereby,fiona is found at second level.

PART 2 RESULTS:

add , find and contains methods outputs:

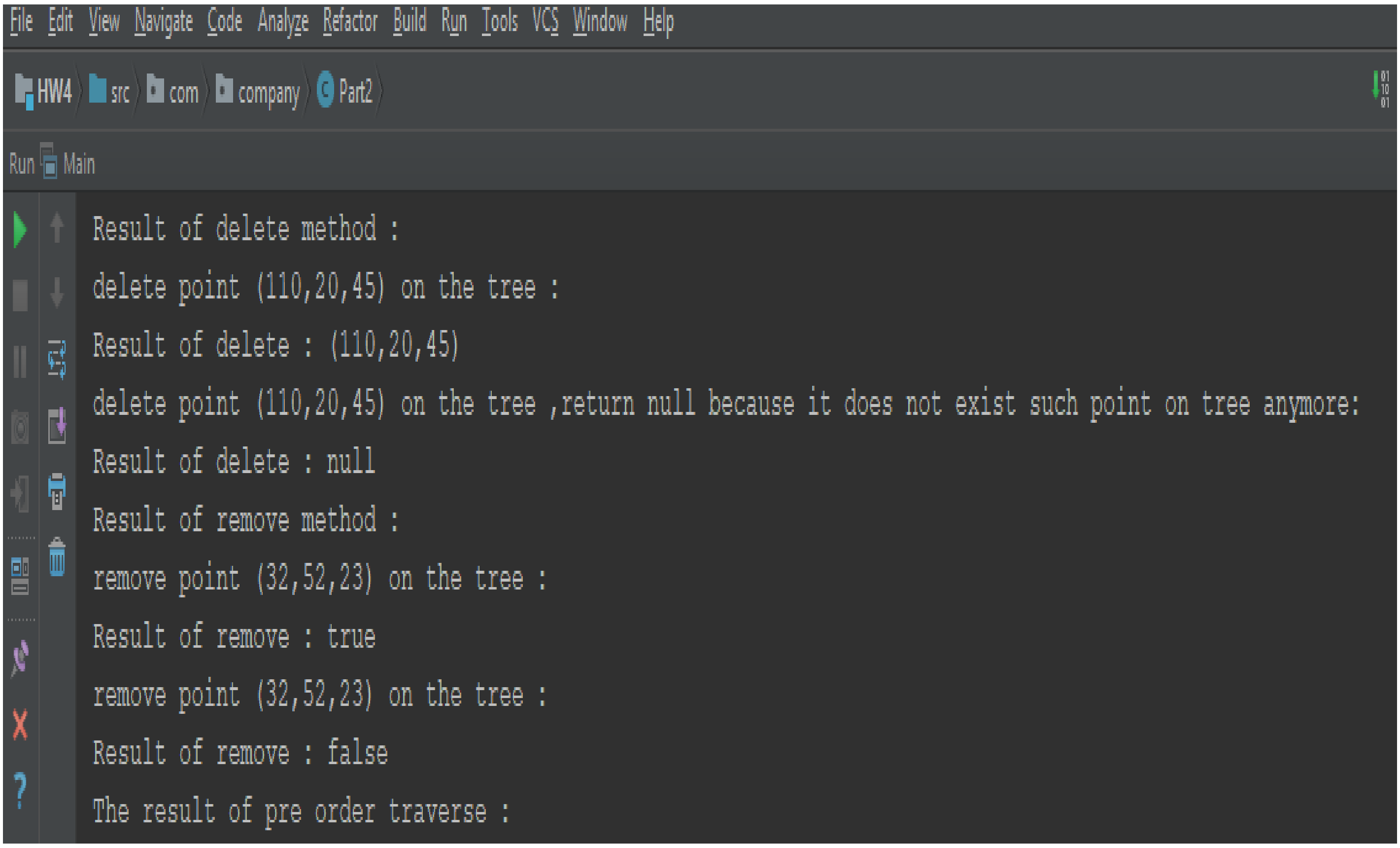
```
Part 2 :
Result of add method:
Root :   (10,20,30)
RightSide:   node   :   (10,20,30)  ,   point   :   (11,25,31)  dimension   :   0
LeftSide:    node   :   (10,20,30)  ,   point   :   (9,18,40)  dimension   :   0
LeftSide:    node   :   (9,18,40)  ,   point   :   (-9,-96,0)  dimension   :   1
RightSide:   node   :   (-9,-96,0)  ,   point   :   (7,15,33)  dimension   :   2
RightSide:   node   :   (11,25,31)  ,   point   :   (32,52,23)  dimension   :   1
LeftSide:    node   :   (11,25,31)  ,   point   :   (110,20,45)  dimension   :   1
RightSide:   node   :   (9,18,40)  ,   point   :   (0,23,37)  dimension   :   1
RightSide:   node   :   (7,15,33)  ,   point   :   (8,17,99)  dimension   :   0

Result of find method :
find point (110,20,45) on the tree :
Result of find : (110,20,45)
find point (9,30,45) on the tree :
Result of find : null
Result of contains method :
find point (32,52,23) on the tree :
Result of contains : true
find point (0,70,-5) on the tree :
Result of contains : false
```

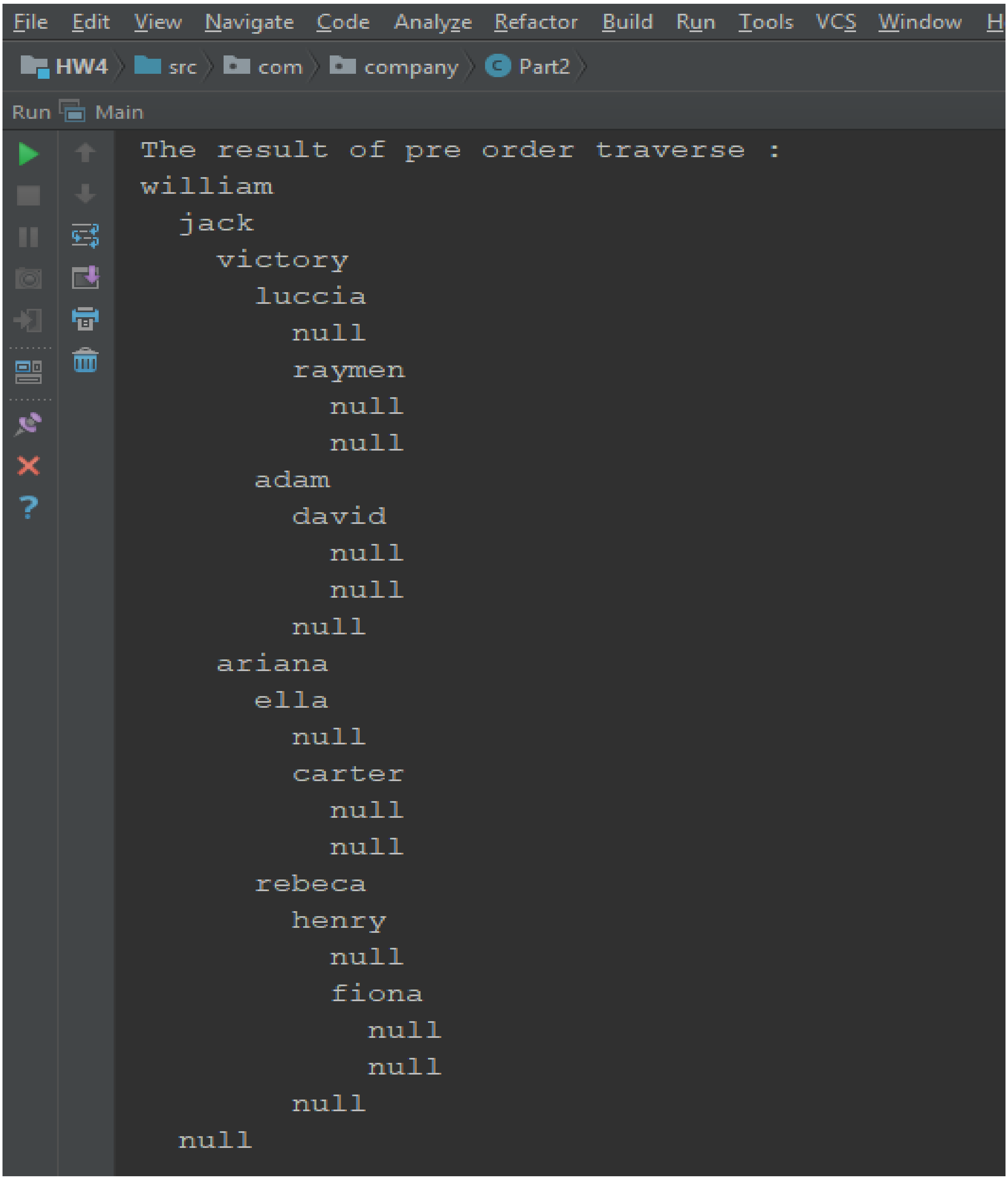
All files are up-to-date (today 16:49)

As seen above, add method adds apoint appropiate location on tree with respect to dimension value.find method return reference of given item if item is present in tree,null otherwise,and then contains method works like find methods ,but it return boolean value.

delete and remove methods outputs :



preOrderTraverse output:



3.4 Analyze time complexity of methods of part1

➔ Level order search:

```
public Node<E> levelOrderSearch(E item,int result) {
    if (root.data.equals(item)) return root;
    Queue<Node<E>> queue = new LinkedList<>();
    Node<E> root1 = this.root;
    queue.add(root1.left);
    if (result == 1) {
        while (!queue.isEmpty()) {
            Node<E>  node = queue.remove();
            if (node.data.equals(item)) return node;
                while (node != null) {
                    if (node.data.equals(item)) return node;
                    System.out.printf(" %s ",node.toString());
                    if (node.left != null) queue.add(node.left);
                    node = node.right;
                }
        }
        return null;
    }
    while (!queue.isEmpty()) {
        Node<E> node = queue.remove();
        if (node.data.equals(item)) return node;
        while (node != null) {

            if (node.data.equals(item)) return node;
            if (node.left != null) queue.add(node.left);
            node = node.right;
        }
    }
    return null;
}
```

result argument is needed for wheather value of node is printing or not when searching.

It is required to calculate time complexity of two while loop : inner while loop and outer while loop.

The best case of the this function is that item is equal to root.So ,$T_B(n) = \Theta(1)$.

The worst case of this function is that every parent has more one than child and child is bottom of tree(the last level of tree.)

inner loop => input size m (for example parent has m child)

$T_{inner}(n) = O(m)$

Outter loop => input size n (for example queue.size() is n)

$T_{outter}(n) = O(n)$

$T_{worst\ case}(n) = T_{outter}(n)*T_{inner}(n) = O(m*n) = O(n^2)$

➔ **Post order search :**

```java
*/
public Node<E> postOrderSearch(E item,int option) {

    if(root == null) return null;

    Stack<Node<E>> tree = new Stack<>();
    Node<E> temp = root;

    while(!tree.empty() || temp!=null){

        if(temp!=null)
        {
            tree.push(temp);
            temp=temp.left;
        }
        else
        {
            Node<E> node = tree.pop();
            if (option == 1) System.out.printf(" %s ",node.toString());
            if (node.data.equals(item)) return node;
            temp = node.right;
        }
    }
    if (option == 1)System.out.println();
    return null;
}
```

The best case of this function is that every parent has just one child.So,It searches linearly on the tree.

$T_{best}(n)$ is O(n);

The worst case of this function is that every parent has one or more childs on the tree.

Assume that,number of parent in tree is n and maximum number of childs of every parent is m.

$T_{worst}(n) = T(m)*T(n) = O(m*n)$

T(n) is O(n) and T(m) is O(n)

thereby $T_{worst}(n)$ is $T(n)*T(m) = O(m*n) = (n^2)$.

➔ add method :

```java
public boolean add(E parent,E child,int option) {
    if (option == 0) {
        if (isLeaf() && root.data.equals(parent)) {
            root.left = new Node<>(child);
            System.out.printf("parent : %s ,first child : %s\n",root.toString(),child.toString());
            return true;
        }
        else  if (isLeaf() && !root.data.equals(parent))
            return false;

        Node<E> temp = levelOrderSearch(parent,0);
        if (temp != null) {
            if (temp.left == null) {
                temp.left = new Node<>(child);
                System.out.printf("parent : %s ,first child : %s\n",temp.toString(),child.toString());//parent
                return true;
            }
            System.out.printf("parent : %s , ",parent.toString());
            temp = temp.left;
            System.out.printf(" first child : %s ,",temp.toString()); // first child.
            while (temp.right!= null) {
                temp = temp.right;
                System.out.printf(" child : %s ,",temp.toString()); // other children
            }
            temp.right = new Node<>(child);
            System.out.printf(" added child : %s\n",child.toString()); // new added child
            return true;
        }
        return false;
    }
```

```java
    if (isLeaf() && root.data.equals(parent)) {
        root.left = new Node<>(child);
        System.out.printf("parent : %s , first child : %s\n",root.toString(),child.toString());
        return true;
    }
    else  if (isLeaf() && !root.data.equals(parent))
        return false;

    Node<E> temp = postOrderSearch(parent,0);
    if (temp != null) {
        if (temp.left == null) {
            temp.left = new Node<>(child);
            System.out.printf("parent : %s , first child : %s\n",temp.toString(),child.toString());
            return true;
        }
        System.out.printf("parent : %s , ",parent.toString());
        temp = temp.left;
        System.out.printf(" first child : %s ,",temp.toString()); // first child.
        while (temp.right!= null) {
            temp = temp.right;
            System.out.printf(" child : %s ,",temp.toString()); // other children
        }
        temp.right = new Node<>(child);
        System.out.printf(" added child : %s\n",child.toString()); // new added child
        return true;
    }
    return false;
}
```

The best case of this function is that parent is root and has no child.So,

$T_{best}(n)$ is $\Theta(1)$

The worst case of this function is $\max(T_{level\text{-}worst\ case}(n), T_{post\text{-}worst\ case}(n)) + T_{find\ last\ clhild\ of\ parent}(n)$

$T_{level\text{-}worst\ case}(n) = O(n^2)$

$T_{post\text{-}worst\ case}(n) = O(n^2)$

$T_{find\ last\ clhild\ of\ parent}(n) = O(n)$

$T_{worst}(n) = \max(O(n^2), O(n^2)) + O(n)$

$\max(O(n^2), O(n^2)) = O(n^2)$

$T_{worst}(n) = O(n^2) + O(n) = \max(O(n^2), O(n)) = O(n^2).$