DSA Coursework 2 Report

Part A: Binary search tree

Task 1: Below is the code of class MemberBST

A Binary Search Tree is implemented in this code to store Member objects. The code's main section and its function description are provided here.

```
package CHC5223;


public class MemberBST implements IMemberDB {
```

The nodes of a binary search tree are represented by the secret internal class known as Node. A member object, a left child node, and a right child node are all present in every node.

```
private class Node {
    Member member; // save message
    Node left; // left node
    Node right;// right node


    Node(Member member) {
    this.member = member;
    this.left = null;
    this.right = null;
}
}
```

Member variables of memberBST:

root: root node of the binomial search tree.

size: the number of nodes in the binary search tree.

```
private Node root; // root node
private int size;
```

Constructor:

```
public MemberBST() {
    root = null;
    size = 0;
    System.out.println("Binary Search Tree");
}
```

Member methods:

1.put(Member member): Insert a member object into the binomial search tree. According to the name of the member object, find the appropriate position to insert the node according to the rules of the binomial search tree.

```
public Member put(Member member) {
    assert member != null : "Member to be inserted should not be null";
    assert !member.getName().isEmpty() : "Member name should not be empty";
    System.out.print("Adding member: " + member.getName());
    System.out.print(", Node visited: ");
    root = put(root, member);
    System.out.println();
    return null;
```

```
}


private Node put(Node node, Member member) {
        // edge judge
        if (node == null) {
        size++;
        return new Node(member);
}


        System.out.print(node.member.getName() + " ");
        // member vs node->member
        int cmp = member.getName().compareTo(node.member.getName());


        // member < node->member, put to left
        if (cmp < 0) {
        if (node.left != null && member.getName().equals(node.left.member.getName())
{
        System.out.println("Duplicate member found: " + member.getName());
}
        node.left = put(node.left, member);
}
        // member > node->member, put to right
        else if (cmp > 0) {
        if (node.right != null &&
member.getName().equals(node.right.member.getName())) {
        System.out.println("Duplicate member found: " + member.getName());
}
        node.right = put(node.right, member);
}
        // member == node->member, ok
        else {
        node.member = member; // replace existing member
}


        // return node
        return node;


}
```

2. get(String name): Get the member object from the binary search tree according to the member name.

```
public Member get(String name) {
        assert name != null && !name.trim().isEmpty() : "Name to be gotten should not
be null or empty";
        System.out.print("Getting member: " + name);
        System.out.print(", Node visited: ");
        Node node = get(root, name);
        System.out.println();
        return node == null ? null : node.member;
}


private Node get(Node node, String name) {
        if (node == null) {
        return null;
}
        System.out.print(node.member.getName() + " ");
        int cmp = name.compareTo(node.member.getName());

        if (cmp < 0) {
        return get(node.left, name);
        } else if (cmp > 0) {
        return get(node.right, name);
        } else {
        return node;
}
}
```

3.remove(String name): According to the member name, delete the member object from the binary search tree.

```
public Member remove(String name) {
        assert name != null && !name.isEmpty() : "Name to be removed should not be
null or empty";

        Node parent = null, del, p = null, q = null;
        Member result;
        del = root;
        while (del != null && !del.member.getName().equals(name)) {
```

```
        parent = del;
        if (name.compareTo(del.member.getName()) < 0)
        del = del.left;
        else
        del = del.right;
}

        if (del != null) {
        if (del == root) {
        System.out.println("Removing member: " + name + ", Node visited: " +
del.member.getName());
        System.out.println("(This deleted node is the root node and the node visited is
itself)");
        } else {
        System.out.println("Removing member: " + name + ", Node visited: " +
del.member.getName());
}

        if (del.right == null) p = del.left;
        else if (del.right.left == null) {
        p = del.right;
        p.left = del.left;
        } else {
        p = del.right;
        while (p.left != null) {
        q = p;
        p = p.left;
}
        q.left = p.right;
        p.left = del.left;
        p.right = del.right;
}


        if (del == root) root = p;
        else if (del.member.getName().compareTo(parent.member.getName()) < 0)
        parent.left = p;
        else
        parent.right = p;

        result = del.member;
        } else {
        System.out.println("Node not found");
        result = null;
}
                size--;
        return result;
}
```

4. Use displayDB() to alphabetically list every member object in the binomial search tree.

```
public void displayDB() {
        if (root == null) {
```

```
        System.out.println("The database is empty.");
        return;
}


        System.out.println("Members in alphabetical order:");
        displayInOrder(root);
}


private void displayInOrder(Node node) {
        if (node != null) {
        displayInOrder(node.left);
        System.out.println("Name: " + node.member.getName() + ", Affiliation: " +
node.member.getAffiliation());
        displayInOrder(node.right);
}
}
```

5.checks whether the binary search tree containsName(String name) has a member with the given name.

```
public boolean containsName(String name) {
        assert name != null && !name.trim().isEmpty() : "Name to be searched should
not be null or empty";
        return containsName(name, root);
}


private boolean containsName(String name, Node node) {
        if (node == null) {
```

```
        return false;
}


        int compare = name.compareTo(node.member.getName());
        if (compare < 0) {
        return containsName(name, node.left);
        } else if (compare > 0) {
        return containsName(name, node.right);
        } else {
        return true;
}
}
```

6. clearDB()：Remove all nodes from the binary search tree to clear it.

```
public void clearDB() {
        root = null;
        size = 0;
}
```

7.size()：Returns the number of nodes in the binary search tree.

```
public int size() {
        return this.size;
}
```

8.isEmpty()：Check if the binary search tree is empty.

```
public boolean isEmpty() {
        return size == 0;
}}
```

Task 2: Assert statement

2.1 public Member put(Member member) {

```
assert member != null : "Member should not be null";
assert !member.getName().isEmpty() : "Member name should not be empty";
```

This assertion language is used to check whether member are null and member name are not empty string. If they are, the assertion language will be showed with "AssertionError".

2.2 public Member get(String name) {

```
assert name != null && !name.trim().isEmpty() : "Name to be gotten should not be
null or empty";
```

This assertion language is used to check whether name are null and empty string. If they are, the assertion language "Name to be gotten should not be null or empty" will be showed with "AssertionError".

2.3 public Member remove(String name) {

```
assert name != null && !name.isEmpty() : "Name to be removed should not be null or
empty";
```

This assertion language is used to check whether name to be moved are null and empty. If they are, the assertion language "Name to be removed should not be null or empty" will be showed with "AssertionError.

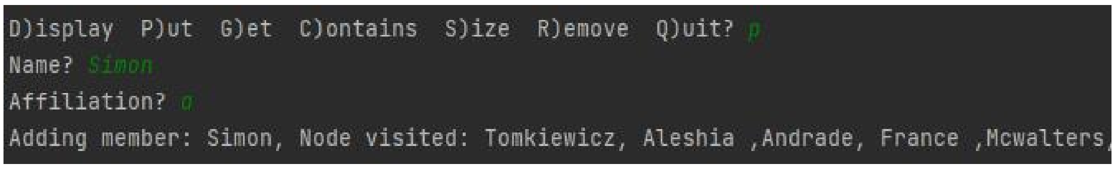
2.4 public boolean containsName(String name) {

```
assert name != null && !name.trim().isEmpty() : "Name to be searched should not be null or empty";
```

This assertion language is used to check whether name to be searched are null and empty. If they are, the assertion language "Name to be searched should not be null or empty" will be showed with "AssertionError.

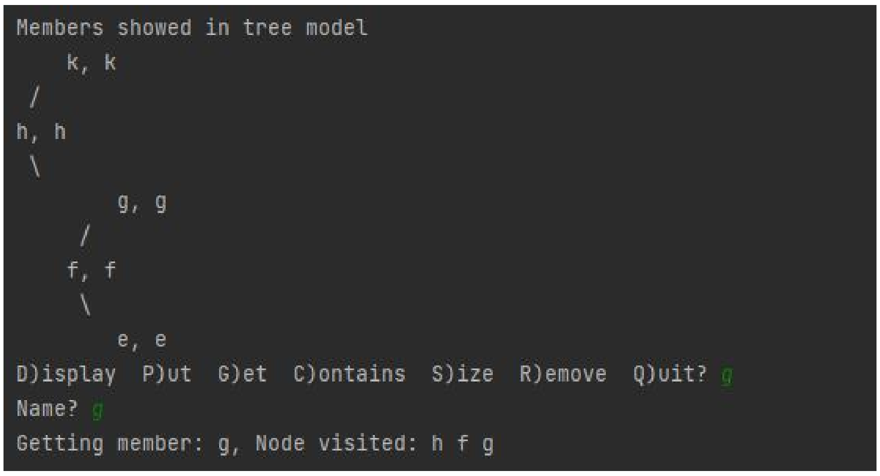
Task3: Log monitoring information

Put a member



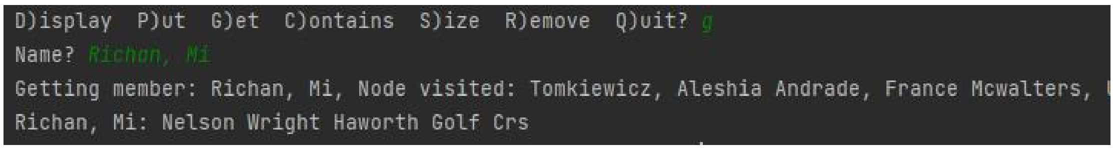
```
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? p
Name? Simon
Affiliation? a
Adding member: Simon, Node visited: Tomkiewicz, Aleshia ,Andrade, France ,Mcwalters,
```

Here we insert "Simon" into the binomial tree. The log message shows that it adds members and all the nodes it traverses.



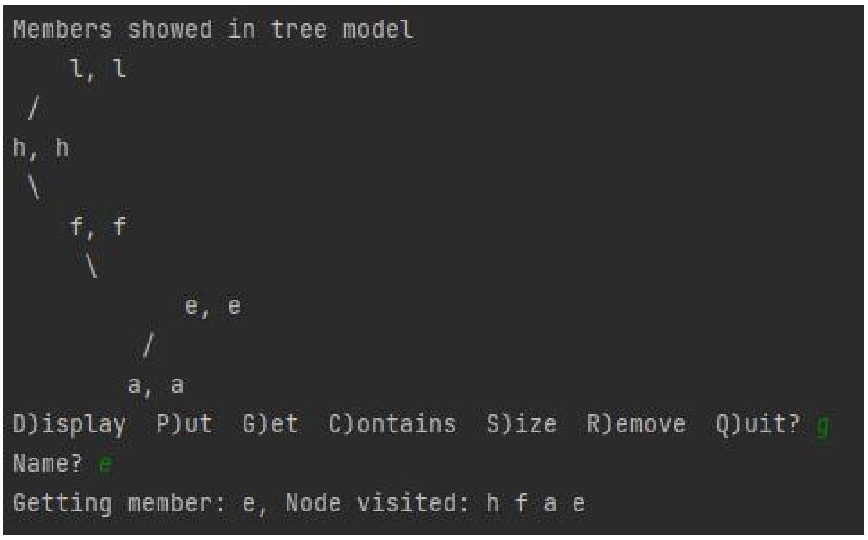
```
Members showed in tree model
    k, k
 /
h, h
 \
        g, g
     /
    f, f
     \
        e, e
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? g
Name? g
Getting member: g, Node visited: h f g
```

Get the member



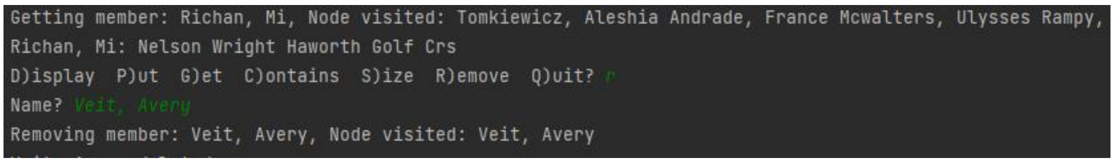
```
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? g
Name? Richan, Mi
Getting member: Richan, Mi, Node visited: Tomkiewicz, Aleshia Andrade, France Mcwalters,
Richan, Mi: Nelson Wright Haworth Golf Crs
```

Here we are looking for the member "Richan,Mi" in a binary tree. The log message shows the member it looks for and all the nodes it traverses.



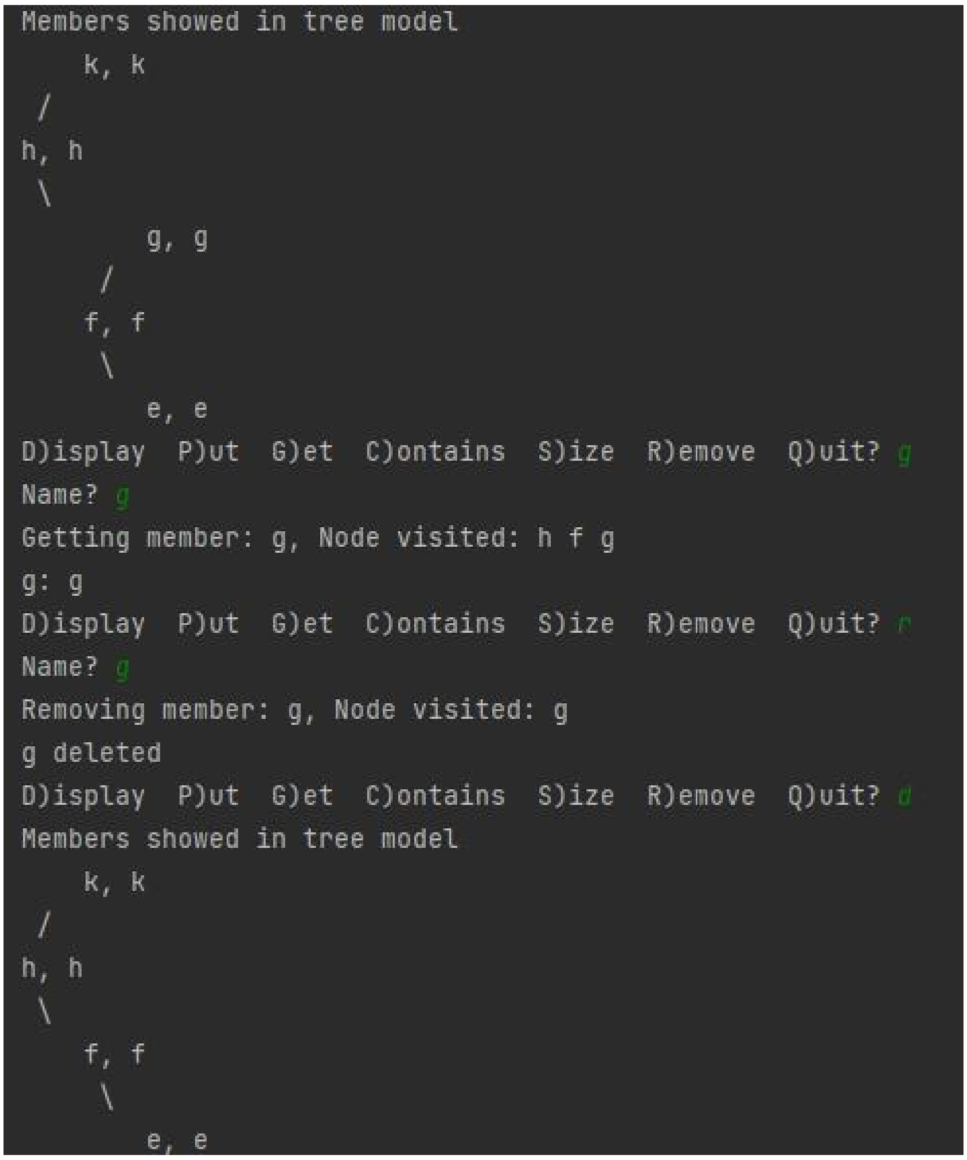
```
Members showed in tree model
   l, l
 /
h, h
 \
   f, f
    \
          e, e
        /
       a, a
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? g
Name? e
Getting member: e, Node visited: h f a e
```

Remove the member



```
Getting member: Richan, Mi, Node visited: Tomkiewicz, Aleshia Andrade, France Mcwalters, Ulysses Rampy,
Richan, Mi: Nelson Wright Haworth Golf Crs
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? r
Name? Veit, Avery
Removing member: Veit, Avery, Node visited: Veit, Avery
```

Here we remove the member "Veit, Avery" from the binary tree. The log message shows the members it deleted and all the nodes it traversed

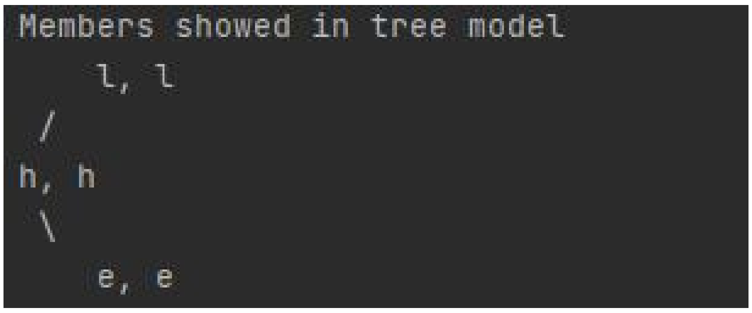


```
Members showed in tree model
    k, k
 /
h, h
 \
        g, g
    /
    f, f
     \
        e, e
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? g
Name? g
Getting member: g, Node visited: h f g
g: g
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? r
Name? g
Removing member: g, Node visited: g
g deleted
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? d
Members showed in tree model
    k, k
 /
h, h
 \
    f, f
     \
        e, e
```

Task4 and 5: Test of BST:

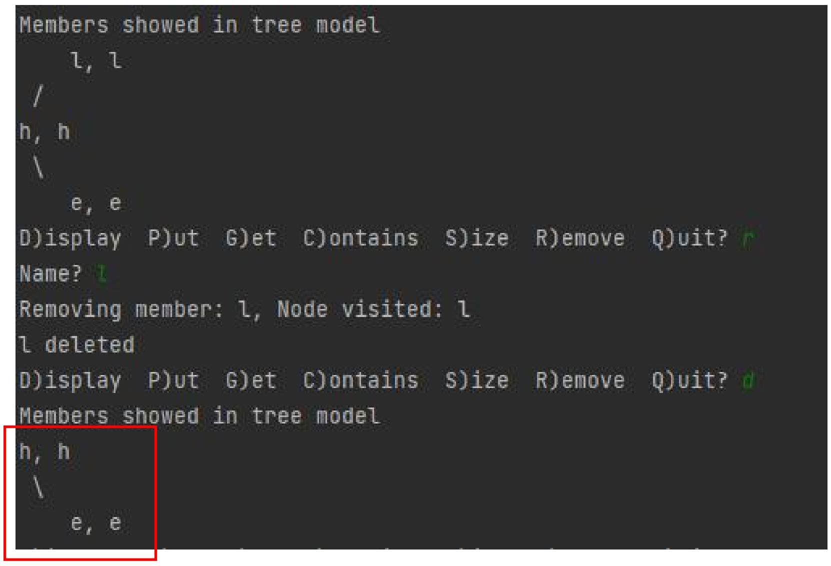
1. **Deleting a leaf node:**

We add member h, e and l to the binary search tree below is the tree:



```
Members showed in tree model
    l, l
 /
h, h
 \
    e, e
```

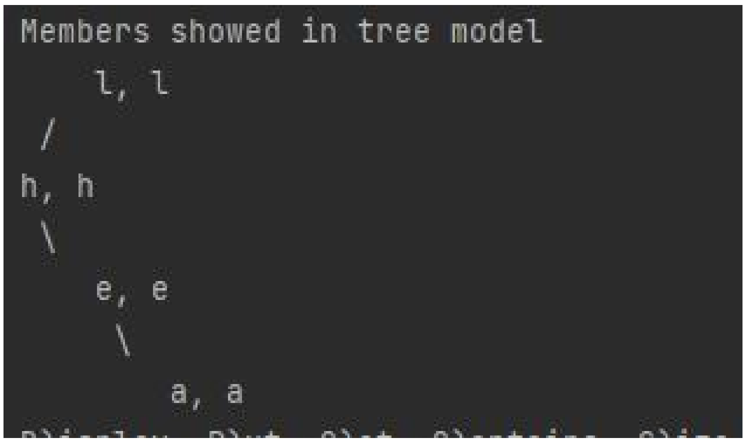
Then e and l are both leaf nodes which not contain both left child nodes and right child nodes. Then we choose to delete node l. (e is in the same)



```
Members showed in tree model
    l, l
 /
h, h
 \
    e, e
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? r
Name? l
Removing member: l, Node visited: l
l deleted
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? d
Members showed in tree model
h, h
 \
    e, e
```

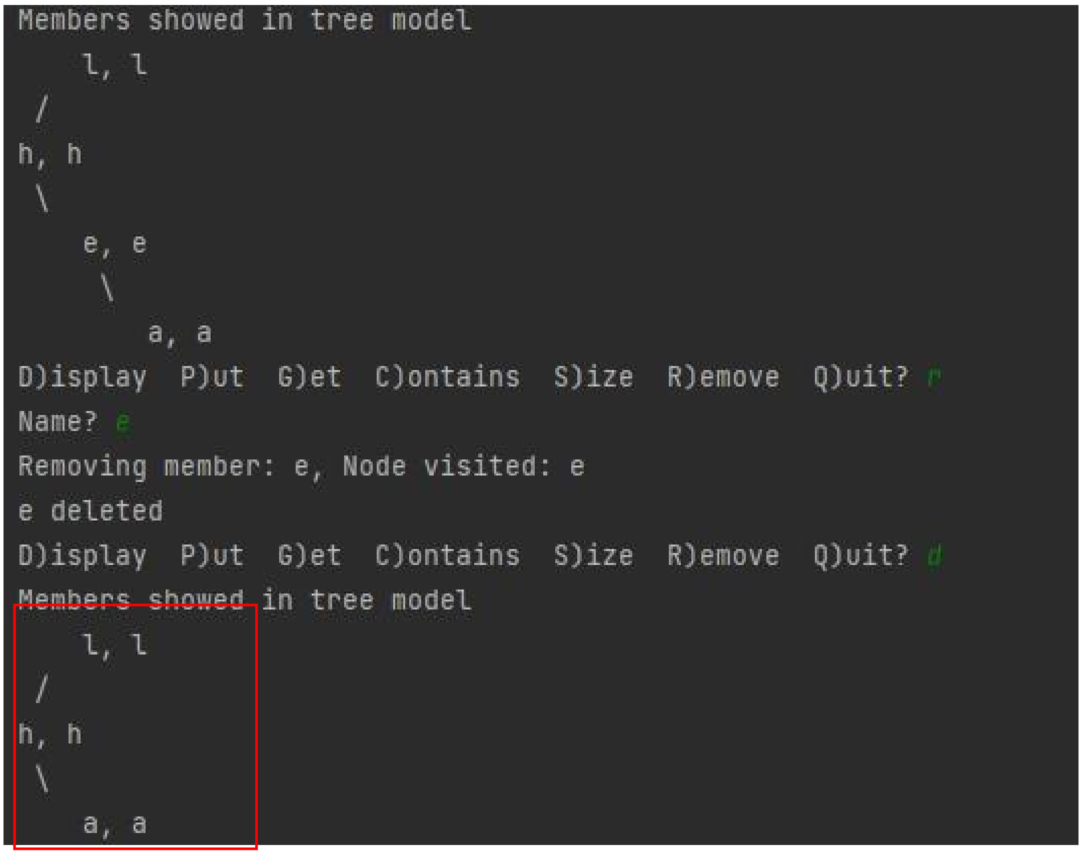
2. **Deleting a node with one descendant**

Then we add member a to the binary search tree based on the above tree.



```
Members showed in tree model
    l, l
 /
h, h
 \
    e, e
     \
        a, a
```

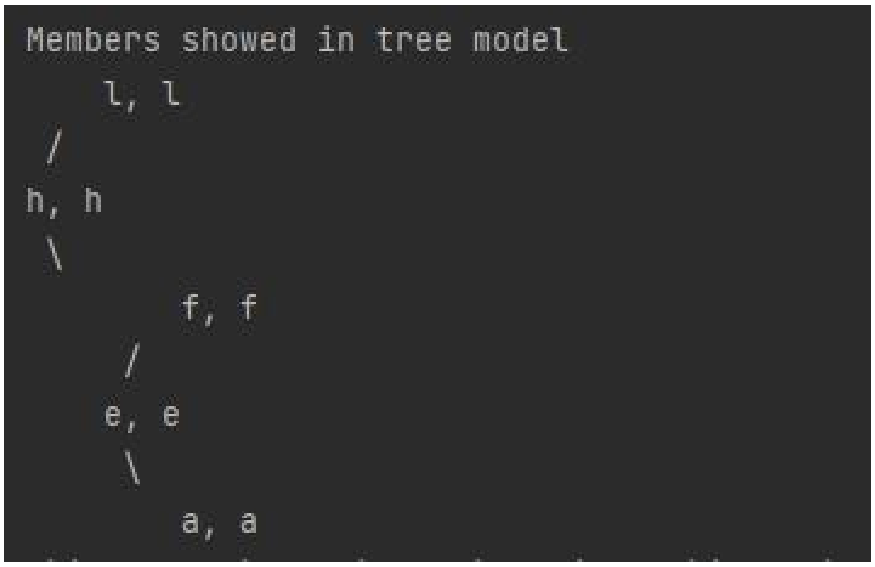
Here e is a node with one descendant, so we delete it. Below is the result.



```
Members showed in tree model
    l, l
 /
h, h
 \
    e, e
     \
        a, a
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? r
Name? e
Removing member: e, Node visited: e
e deleted
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? d
Members showed in tree model
    l, l
 /
h, h
 \
    a, a
```

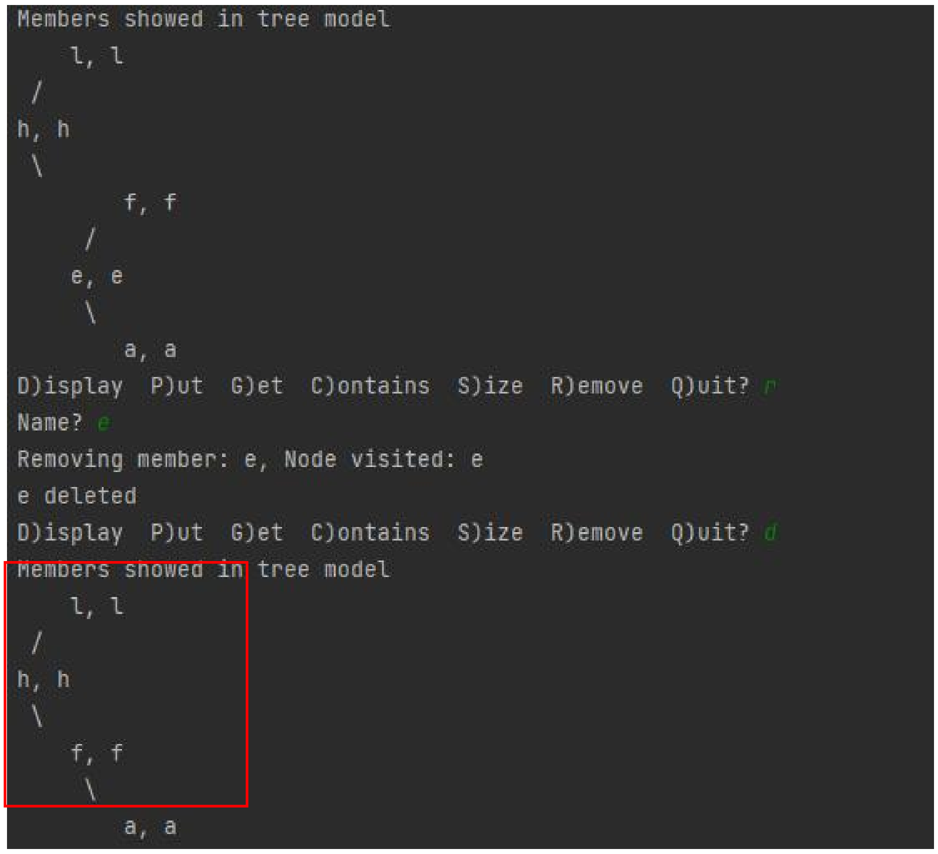
3. Deleting a node with two descendants

Then we add f node to above binary search tree.



```
Members showed in tree model
    l, l
 /
h, h
 \
        f, f
     /
    e, e
     \
        a, a
```

Here e is a node with two descendants, then we delete it. Below is the result.



```
Members showed in tree model
    l, l
 /
h, h
 \
        f, f
     /
    e, e
     \
        a, a
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? r
Name? e
Removing member: e, Node visited: e
e deleted
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? d
Members showed in tree model
    l, l
 /
h, h
 \
    f, f
     \
        a, a
```

Task 6:

Time efficiency:

put(Member member) function:

In the worst case, i.e., when the tree is not balanced, the time complexity of the insertion operation is O(N), where N is the number of nodes in the tree

get(String name) function:

In the worst case, i.e., when the tree is not balanced, the time complexity of the insertion operation is O(N), where N is the number of nodes in the tree

remove(String name) function:

In the worst case, i.e., when the tree is not balanced, the time complexity of the insertion operation is O(N), where N is the number of nodes in the tree

displayDB() function：

This function traverses the entire binomial search tree and displays information about the member objects in alphabetical order.

The time complexity depends on the number of nodes in the binomial search tree and is O(N), where N is the number of nodes in the tree.

containsName(String name) function：

The time complexity depends on the number of nodes in the binomial search tree and is O(N), where N is the number of nodes in the tree.

clearDB() function：

The time complexity is O(1) because only the root node needs to be set to null.

size() function：

The time complexity is O(1) because only the number of stored nodes needs to be returned.

isEmpty() function：

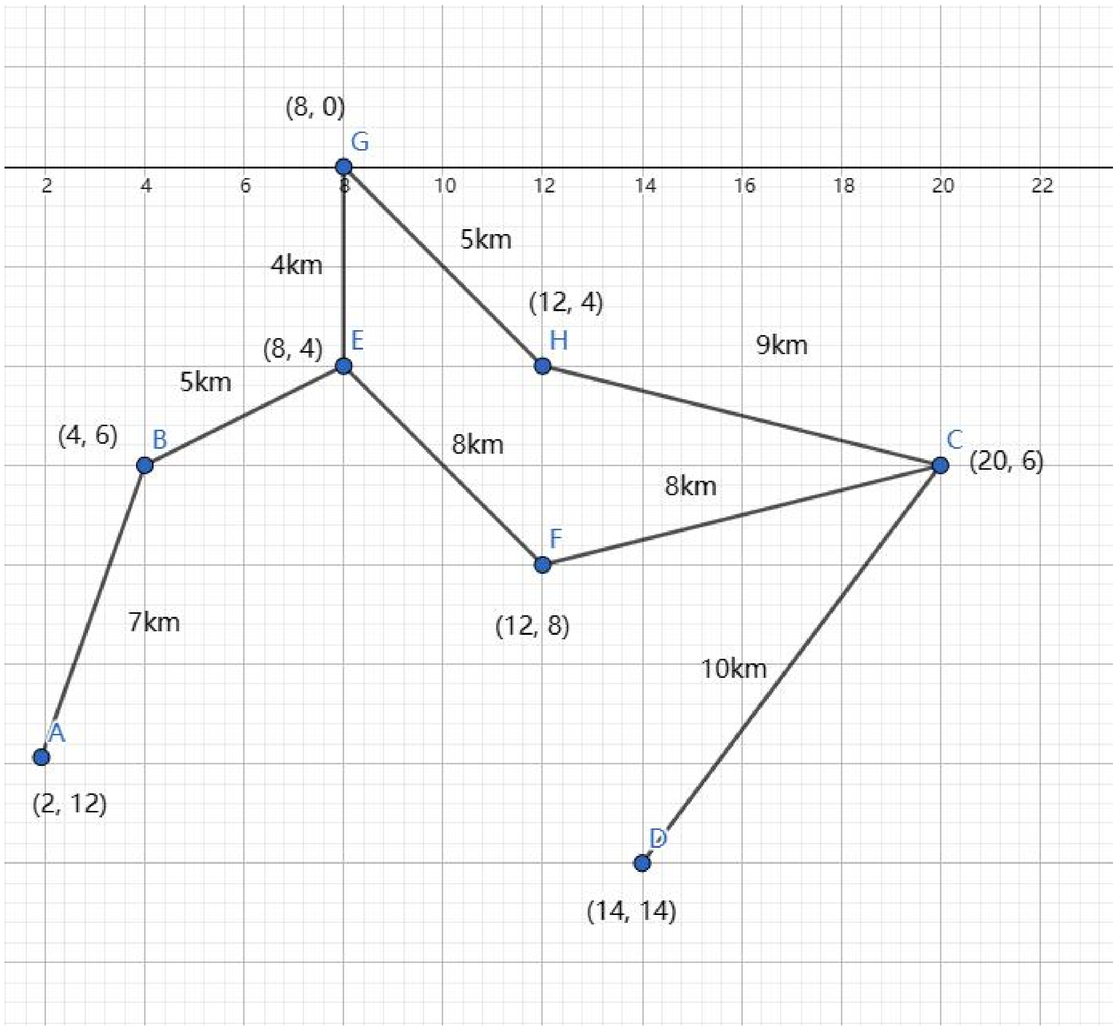
The time complexity is O(1), because it is only necessary to determine whether the number of nodes is 0 or not.

Space efficiency: The binary search tree uses a chain table structure to represent the nodes, and the given function implementation maintains the member objects and references to the left and right child nodes. As a result, the space complexity primarily depends on how many nodes are stored in the binary search tree. The space complexity of a binary search tree with N nodes is O(N). The necessary storage space grows linearly as the number of nodes increases since each node needs a specific amount of space to store member objects and references to left and right child nodes. Please take note that the space complexity is based on the number of nodes and excludes the actual space occupied by the member objects.

In task1, I create a class called MemberBST interface the ImemberDB and Use recursion to implement the function of adding, deleting, and searching in a binary search tree. In this class I created two disply methods, one for easy printing when the number of members is large and the other for showing the tree when the number of members are not very large using a middle-order traversal. In task2, I using some assert language to avoid some special situation like the name is null or member is null and on. In task 3, I print a log information when we put, get and remove a member.In task 4 and 5 test the programme well. In task 6, I write down and compare the complexity of each function.

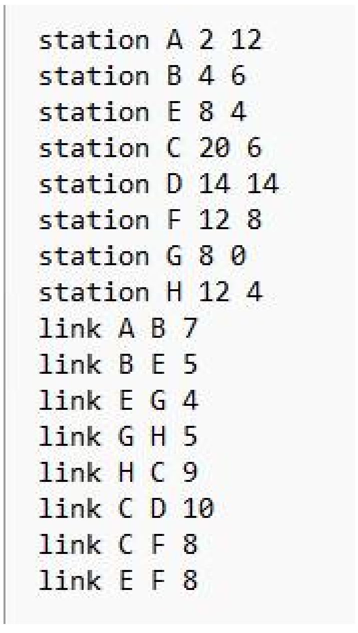
Part B: Binary search tree

Task 1:



Task 2:

This is my text file format:


```
station A 2 12
station B 4 6
station E 8 4
station C 20 6
station D 14 14
station F 12 8
station G 8 0
station H 12 4
link A B 7
link B E 5
link E G 4
link G H 5
link H C 9
link C D 10
link C F 8
link E F 8
```

Task 3:

Class ListInt:

```
}package CHC5223;


public class ListInt extends AbsListInt {
	public ListInt(int capacity){
		super(capacity);
	}


	@Override
	public void append(int n) {
		assert this.size != this.capacity : "List is full, cannot append value";
		this.list[this.size++] = n;
	}


	@Override
	public boolean contains(int x) {
		for(int i = 0; i < size; i++) {
			if(this.list[i] == x) {
				return true;
			}
```

```
        }



        return false;
    }
}
```

Class QueueInt:

```
 package CHC5223;
public class QueueInt extends AbsQueueInt {


    public QueueInt(int capacity) {
        super(capacity);
    }


    @Override
    public void addToBack(int n) {
        assert getSize() < getCapacity() : " Queue is full, cannot add to back";



        this.queue[this.size] = n;
        this.size++;
        assert this.queue[this.size - 1] == n : " n is not at the back of the queue";
    }

    @Override
    public int removeFromFront() {
        assert getSize() > 0 : " Queue is empty, cannot remove from front";
        int frontElement = this.queue[0];
        System.arraycopy(this.queue, 1, this.queue, 0, this.size - 1);
        this.size--;

        assert !this.contains(frontElement) : " Removed element is still in the
queue";

        return frontElement;
    }

    private boolean contains(int element) {
        for (int i = 0; i < this.size; i++) {
            if (this.queue[i] == element) {
                return true;
            }
        }
        return false;
    }
}
```

Class SetInt:

```
package CHC5223;
public class SetInt extends AbsSetInt {


    public SetInt(int capacity) {
        super(capacity);
    }


    @Override
    public boolean contains(int x) {
        for (int i = 0; i < this.size; i++) {
            if (this.set[i] == x) {
                return true;
            }
        }
        return false;
    }

    @Override
    public void include(int n) {
        assert contains(n) || getSize() != getCapacity() : "Set is full or element
already exists";

        if (!contains(n)) {
            this.set[this.size] = n;
            this.size++;
        }


        assert contains(n) : "Element was not included in the set";
    }


    @Override
    public void exclude(int n) {
        int index = -1;
        for (int i = 0; i < this.size; i++) {
            if (this.set[i] == n) {
                index = i;
                break;
            }
        }


        if (index != -1) {
            for (int i = index; i < this.size - 1; i++) {
                this.set[i] = this.set[i+1];
            }
            this.size--;
        }
        assert !contains(n) : "Element was not removed from the set";
    }
}
```

Class StackInt:

```
package CHC5223;

public class StackInt extends AbsStackInt {


    public StackInt(int capacity) {

        super(capacity);

    }


    @Override

    public void push(int n) {

        assert this.size != this.capacity : "Stack is full, cannot push";

        this.stack[this.size] = n;

        this.size++;

    }


    @Override

    public int pop() {

        assert this.size != 0 : "Stack is empty, cannot pop";

        int poppedValue = this.stack[this.size - 1];

        this.size--;

        return poppedValue;

    }


    @Override
    public int peek() {
        assert this.size != 0 : "Stack is empty, cannot peek";
        return this.stack[this.size - 1];
    }
}
```

Task 4:

Assert language:

ListInt:

```
assert this.size != this.capacity : "List is full, cannot append value";
```

In order to avoid array overruns and other related issues in list, it is used to make sure the list doesn't go over its predetermined capacity limit in append method.

QueueInt:

```
assert getSize() < getCapacity() : "Pre-condition violation: Queue is full, cannot add to back";
```

In order to avoid array overruns and other related issues in the queue, it is used to make sure the list doesn't go over its predetermined capacity limit in addToBack method.

```
assert this.queue[this.size - 1] == n : "n is not at the back of the queue";
```

This assert language check whether the n have been added at the back of the queue in addToBack method.

```
assert getSize() > 0 : "Queue is empty, cannot remove from front";
```

It is used to make sure that at least one element is present in the queue before an element is removed from the front of the queue in removeFromFront method.

```
assert !this.contains(frontElement) : " Removed element is still in the queue";
```

It is used to make sure that an element no longer exists in the queue once it has been removed from the front of the queue in removeFromFront method.

SetInt:

```
assert contains(n) || getSize() != getCapacity() : "Set is full or element already exists";
```

Here we calling method contain(int x). It is used to check if there is space in the collection or that the element n to be added either doesn't exist in the collection in include method.

```
assert contains(n) : "Element was not included in the set";
```

Here we calling method contain(int x). It is used to ensure that the set already contains the specified elements n in include method.

```
assert !contains(n) : "Element was not removed from the set";
```

Here we calling method contain(int x).   It is used to check to see if the designated element has been correctly eliminated from the set n in exclude method.

StackInt:

```
assert this.size != this.capacity : "Stack is full, cannot push";
```

It is used to ensure that there is still space available on the stack before a push operation is performed in push method.

```
assert this.size != 0 : "Stack is empty, cannot pop";
```

It is used to ensure that at least one element exists in the stack before the pop operation is performed in pop method.

```
assert this.size != 0 : "Stack is empty, cannot peek";
```

It is used to ensure that at least one element exists in the stack before peek operation in peek method.

Task 5:

Before is my junit test plan:

testStackInt():

```
@org.junit.Test
public void testStackInt() {
        AbsStackInt    stackInt = new StackInt(10);
        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        for (int i = 0; i < a.length; i++) {
            stackInt.push(a[i]);
        }
        Assert.assertEquals(10, stackInt.getCapacity());
        Assert.assertEquals(10, stackInt.getSize());
        Assert.assertEquals(10, stackInt.peek());
        Assert.assertEquals(10, stackInt.pop());
        Assert.assertEquals(9, stackInt.pop());
        Assert.assertEquals(8, stackInt.pop());
        Assert.assertEquals(7, stackInt.getSize());


    }
```

1. Create a StackInt object and the capacity is specified as 10.
2. Use the push() method to push elements of the integer array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] to the stack.
3. At this time the expected capacity and size are 10 so we test getSize() and getCapacity() We use this two lines Assert.assertEquals(10, stackInt.getCapacity());

1. Assert.assertEquals(10, stackInt.getSize()) to test this two method;. And the assertEqual() to judge whether the result is equal to the expected value. And also test the push method because it can test whether the integer are add to stack.

4. The peek value is 10, then we use this line Assert.assertEquals(10, stackInt.peek());

to check whether the result is equal to the expected value and to test this method.

5. Because the feature of the stack last in first out, so the expected value after stackInt.pop() is 10, then we use Assert.assertEquals(10, stackInt.pop()); to check
6. whether the result is equal to the expected value and to test this method.
7. The left lines have same regularity.

testQueueInt():

```
    @org.junit.Test
    public void testQueueInt() {
        AbsQueueInt queueInt = new QueueInt(10);
        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        for (int i = 0; i < a.length; i++) {
            queueInt.addToBack(a[i]);
        }
        Assert.assertEquals(10, queueInt.getCapacity());
        Assert.assertEquals(10, queueInt.getSize());
        Assert.assertEquals(1, queueInt.removeFromFront());
        Assert.assertEquals(2, queueInt.removeFromFront());
        Assert.assertEquals(3, queueInt.removeFromFront());
        queueInt.addToBack(11);
        Assert.assertEquals(8, queueInt.getSize());

    }
```

2. Create a queueInt object and the capacity is specified as 10.
3. Use the addToBack() method to add elements of the integer array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] to the queue.

4. At this time the expected capacity and size are 10 so we test getSize() and getCapacity() using this two lines Assert.assertEquals(10, queueInt.getCapacity());Assert.assertEquals(10, queueInt.getSize());to test this two method;. And the assertEqual() to judge whether the result is equal to the expected value and to test the two method.And also test the addToBack method because it can test whether the integer are add to queue.

8. The feature of the queue is first in first out. After queueInt.removeFromFront(), the expected value is1, so we use Assert.assertEquals(1, queueInt.removeFromFront());to check whether the result is equal to the expected value and to test removeFromFront method.

5. Then do queueInt.removeFromFront() again, the the expected value is 2,then we use Assert.assertEquals(2, queueInt.removeFromFront());to check whether the result is equal to the expected value.

9. Then do queueInt.addToBack(11);, the size will be increment 1, so the expected value is 10-3+1=8, using Assert.assertEquals(8, queueInt.getSize());to check whether the result is equal to the expected value to test addToBack method.

testSetInt():

```
    @org.junit.Test
        public void testSetInt() {
            AbsSetInt setInt = new SetInt(10);
            int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
            for (int i = 0; i < a.length; i++) {
                setInt.include(a[i]);
            }
            Assert.assertEquals(10, setInt.getCapacity());
            Assert.assertEquals(10, setInt.getSize());
```

```
                Assert.assertTrue(setInt.contains(10));

                setInt.exclude(10);

                Assert.assertFalse(setInt.contains(10));

                Assert.assertEquals(9, setInt.getSize());

                setInt.exclude(9);

                Assert.assertFalse(setInt.contains(9));

                Assert.assertEquals(8, setInt.getSize());

            }
```

1. Create a setInt object and the capacity is specified as 10.

2. Use the include() method to add elements of the integer array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] to the set.

3.At this time the expected capacity and size are 10 so we test getSize() and getCapacity() using this two lines Assert.assertEquals(10, setInt.getCapacity()); Assert.assertEquals(10, setInt.getSize()); to test this two method;. And the assertEqual() to judge whether the result is equal to the expected value. And also test the include method because it can test whether the integer are add to set.

4.After setInt.contains(10) , the expected value should be true. Then use Assert.assertTrue(setInt.contains(10));to check whether the result is equal to the expected value and to test .contains method.

5.After setInt.exclude(10); , the 10 will be delete from the set so that the set will not contain 10.The expected value is false. We use Assert.assertFalse(setInt.contains(10)); to check whether the result is equal to the expected value and to test exclude method.

testListInt():

```
    @org.junit.Test

        public void testListInt() {
                AbsListInt listInt = new ListInt(10);

                int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

                for (int i = 0; i < a.length; i++) {

                      listInt.append(a[i]);

                }

                Assert.assertEquals(10, listInt.getCapacity());

                Assert.assertEquals(10, listInt.getSize());

                for (int i = 0; i < a.length; i++) {

                      Assert.assertTrue(listInt.contains(a[i]));

                }

        }

  }
```

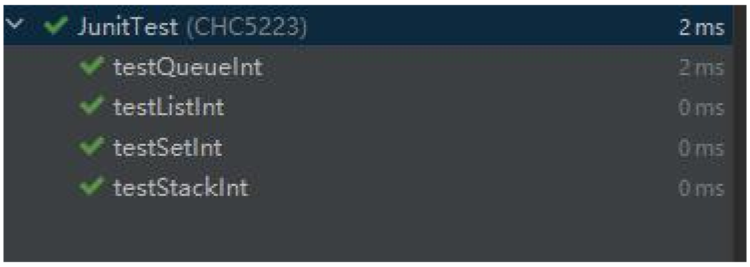
1. Create a setInt object and the capacity is specified as 10.

2. Use the append() method to add elements of the integer array [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] to the list.

3. getCapacity() using this two lines Assert.assertEquals(10, listInt.getCapacity()); Assert.assertEquals(10, listInt.getSize()); to test this two method;. And the assertEqual() to judge whether the result is equal to the expected value. And also test the append method because it can test whether the integer are add to list.

4. Making use of for loops to check whether each of integer of array are in list to check contain method.The expected value is true. Then use Assert.assertTrue to check whether the result is equal to the expected value

Results:



```
✓ JunitTest (CHC5223)    2 ms
    ✓ testQueueInt       2 ms
    ✓ testListInt        0 ms
    ✓ testSetInt         0 ms
    ✓ testStackInt       0 ms
```

Four class are all passed! The function of method included in four class can be implemented successfully.

Task 6:

Below is the code of my stationInfo class:

```
    }package CHC5223; // or whatever

    public class StationInfo implements IStationInfo {

        private String name;
        private int xPos;
        private int yPos;

        public StationInfo(String name, int xPos, int yPos) {
            if (xPos < 0 || xPos >= 256) throw new
IllegalArgumentException("Invalid x position");
            if (yPos < 0 || yPos >= 256) throw new
IllegalArgumentException("Invalid y position");
            this.name = name;
            this.xPos = xPos;
            this.yPos = yPos;
        }


        @Override
        public String getName() {
                return this.name;
        }


        @Override
        public int getxPos() {
                return this.xPos;
        }


        @Override
        public int getyPos() {
                return this.yPos;
        }
}
```

Task 7:

Main part of AdjacencyMatrix class:

```
 public class AdjacencyMatrix {
        final double NO_LINK = Double.MAX_VALUE; // was erroneous
double.MAX_VALUE
        int numStations; // 0 <= numStations <= capacity
        double distance[][];


    public AdjacencyMatrix(int capacity) {
                this.numStations = 0; // Initialize with no stations
                distance = new double[capacity][capacity];
                stationInfos = new StationInfo[capacity];    // Initialize stationInfos

                // Initialize all distances to NO_LINK
                for (int i = 0; i < capacity; i++) {
                        for (int j = 0; j < capacity; j++) {
                                distance[i][j] = NO_LINK;
                        }
                }
        }
```

All codes will be appended in the appendix.

Task 8:

```
    StationInfo[] stationInfos;
```

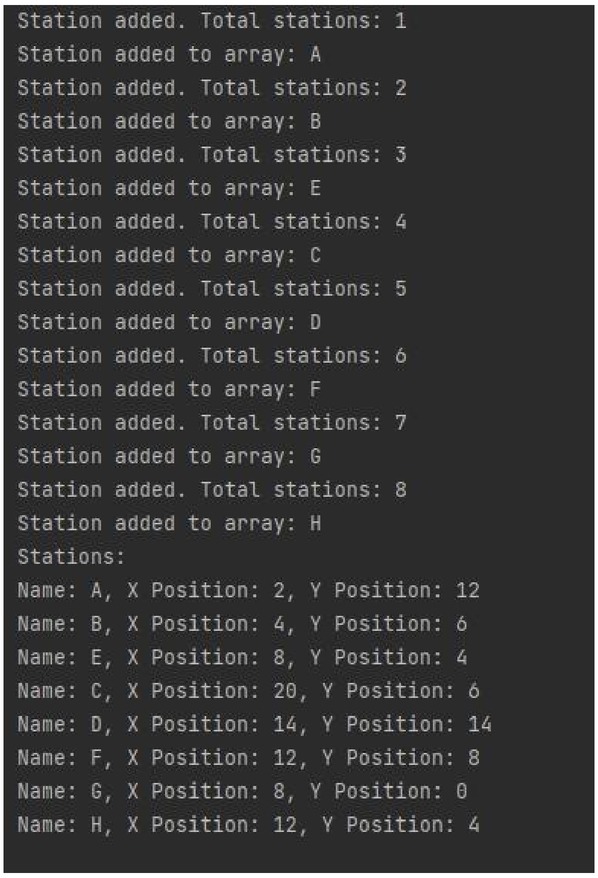
List of objects of class StationInfo held in an array

Task 9:

```
            }
        }
```

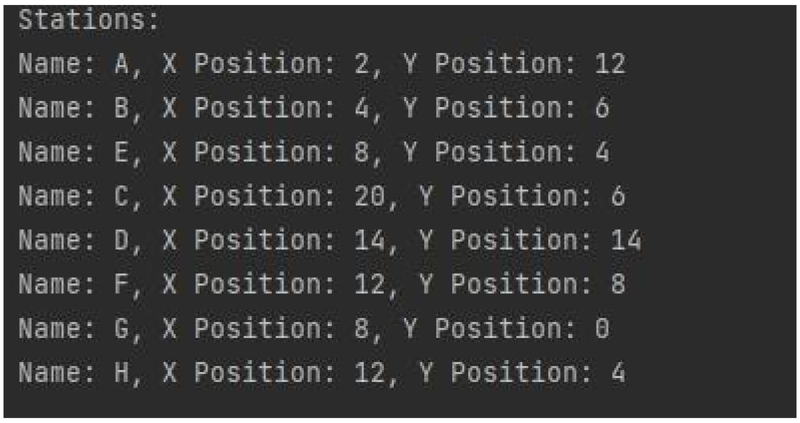
```
    public void readNetwork(String filename) throws IOException {
            Map<String, Integer> stationIndices = new HashMap<>();
            BufferedReader reader = new BufferedReader(new
FileReader(filename));
            String line;

            while ((line = reader.readLine()) != null) {
                // Trim the line and skip if it is empty.
                line = line.trim();
                if (line.isEmpty()) {
                    continue;
                }

                String[] parts = line.split(" ");
                if (parts.length < 3) {
                    throw new IllegalArgumentException("Invalid line: " + line);
                }

                String type = parts[0];



                if (type.equals("station")) {
                    if (parts.length != 4) {
                        throw new IllegalArgumentException("Invalid line for
station: " + line);
                    }
                    String name = parts[1];
                    int x = Integer.parseInt(parts[2]);
                    int y = Integer.parseInt(parts[3]);
                    StationInfo station = new StationInfo(name, x, y);
                    addStation();
                    stationInfos[numStations - 1] = station;
                    stationIndices.put(name, numStations - 1);
                    System.out.println("Station added to array: " +
station.getName());
                } else if (type.equals("link")) {
                    if (parts.length != 4) {
                        throw new IllegalArgumentException("Invalid line for
link: " + line);
                    }
                    String station1Name = parts[1];
                    String station2Name = parts[2];
                    double dist = Double.parseDouble(parts[3]);

                    Integer station1Index = stationIndices.get(station1Name);
                    Integer station2Index = stationIndices.get(station2Name);

                    if (station1Index == null || station2Index == null) {
                        throw new IllegalArgumentException("Link references
unknown station: " + line);
                    }
                    addLink(station1Index, station2Index, dist);
                } else {
                    throw new IllegalArgumentException("Unknown type: " +
type);
                }
            }

            reader.close();
        }
```

Result:



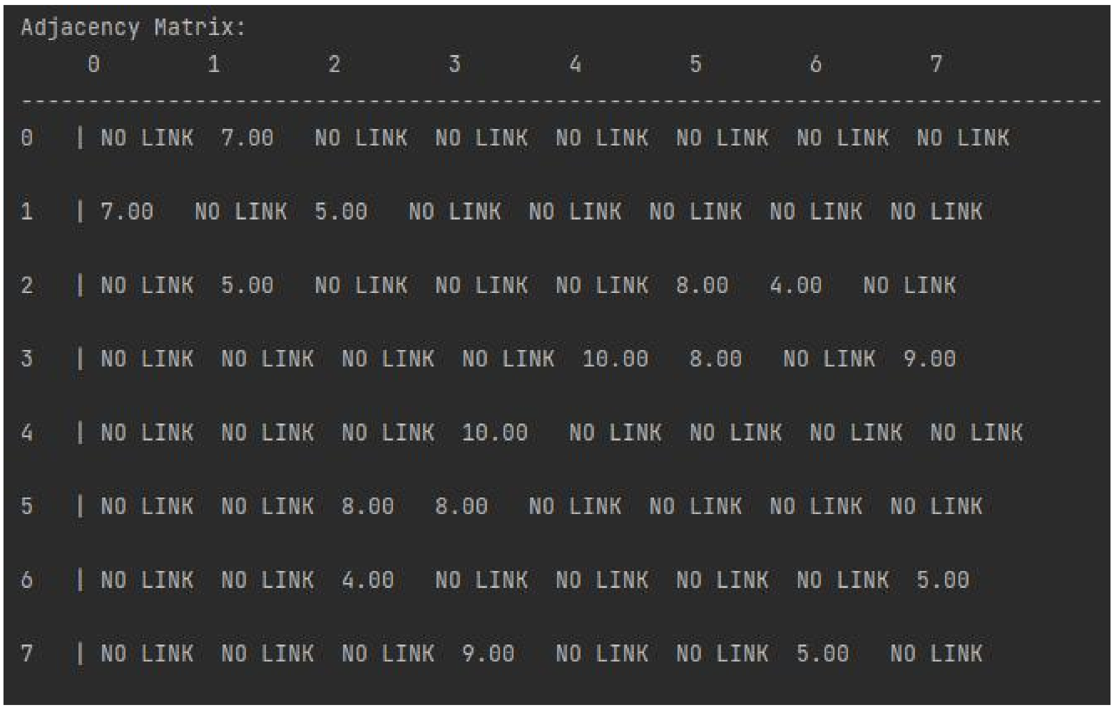
```
Station added. Total stations: 1
Station added to array: A
Station added. Total stations: 2
Station added to array: B
Station added. Total stations: 3
Station added to array: E
Station added. Total stations: 4
Station added to array: C
Station added. Total stations: 5
Station added to array: D
Station added. Total stations: 6
Station added to array: F
Station added. Total stations: 7
Station added to array: G
Station added. Total stations: 8
Station added to array: H
Stations:
Name: A, X Position: 2, Y Position: 12
Name: B, X Position: 4, Y Position: 6
Name: E, X Position: 8, Y Position: 4
Name: C, X Position: 20, Y Position: 6
Name: D, X Position: 14, Y Position: 14
Name: F, X Position: 12, Y Position: 8
Name: G, X Position: 8, Y Position: 0
Name: H, X Position: 12, Y Position: 4
```

Task 10: Here is print of the content of list of stations



```
Stations:
Name: A, X Position: 2, Y Position: 12
Name: B, X Position: 4, Y Position: 6
Name: E, X Position: 8, Y Position: 4
Name: C, X Position: 20, Y Position: 6
Name: D, X Position: 14, Y Position: 14
Name: F, X Position: 12, Y Position: 8
Name: G, X Position: 8, Y Position: 0
Name: H, X Position: 12, Y Position: 4
```

Here is the matrix build by my program.



```
Adjacency Matrix:
     0        1        2        3        4        5        6        7
--------------------------------------------------------------------------------
0   | NO LINK  7.00   NO LINK  NO LINK  NO LINK  NO LINK  NO LINK  NO LINK

1   | 7.00   NO LINK  5.00   NO LINK  NO LINK  NO LINK  NO LINK  NO LINK

2   | NO LINK  5.00   NO LINK  NO LINK  NO LINK  8.00   4.00   NO LINK

3   | NO LINK  NO LINK  NO LINK  NO LINK  10.00   8.00   NO LINK  9.00

4   | NO LINK  NO LINK  NO LINK  10.00   NO LINK  NO LINK  NO LINK  NO LINK

5   | NO LINK  NO LINK  8.00   8.00   NO LINK  NO LINK  NO LINK  NO LINK

6   | NO LINK  NO LINK  4.00   NO LINK  NO LINK  NO LINK  NO LINK  5.00

7   | NO LINK  NO LINK  NO LINK  9.00   NO LINK  NO LINK  5.00   NO LINK
```

Task 11: Depth-first-Traversal and Breadth-first-Traversal

1. Depth-first-Traversal

```
public List<StationInfo> depthFirstTraversal(int startStation) {

        Stack<Integer> S = new Stack<>();

        List<StationInfo> L = new ArrayList<>();

        S.push(startStation);
        while (!S.isEmpty()) {
            int T = S.pop();
            if (!L.contains(stationInfos[T])) {
                L.add(stationInfos[T]);



                for (int T2 = 0; T2 < numStations; T2++) {
                    if (distance[T][T2] != NO_LINK
&& !L.contains(stationInfos[T2])) {
                        S.push(T2);
                    }
                }
            }
        }

        return L;
    }

```

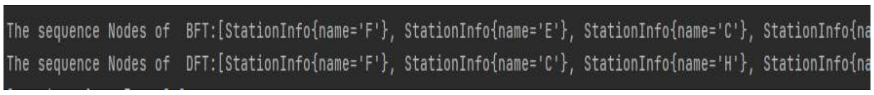
Beginning at a specific node, the depth-first algorithm first explores as far as it can along each branch before turning around. The procedure pushes the beginning station into the stack, initializes the stack and list, and then starts the loop. Each time the loop is executed, the top station is removed from the stack, checked to see if it has already been visited, and if not, marked as such. If not, all nearby stations that have not yet been visited are then pushed onto the stack. Until the stack is empty and all sites that may be accessed have been viewed, the method will keep running.

2. Breadth-first-Traversal

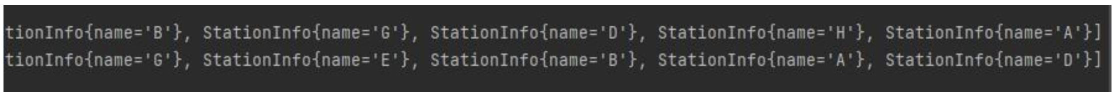
```
public List<StationInfo> breadthFirstTraversal(int startStation) {
            Queue<Integer> Q = new LinkedList<>();
            List<StationInfo> L = new ArrayList<>();


            Q.add(startStation);


            while (!Q.isEmpty()) {
                int S = Q.poll();


                if (!L.contains(stationInfos[S])) {
                    L.add(stationInfos[S]);
                    for (int S2 = 0; S2 < numStations; S2++) {
                        if (distance[S][S2] != NO_LINK
&& !L.contains(stationInfos[S2])) {
                            Q.add(S2);
                        }
                    }
                }
            }


            return L;
        }
```

The queue and list are both initialized by the breadth-first approach before the beginning station is queued and the loop is started. In each iteration, a station is taken from the front of the line, checked to see whether it has been visited, and if not, marked as visited. Then, all nearby stations that have not been visited are queued up until the line is empty.

Result:



```
The sequence Nodes of  BFT:[StationInfo{name='F'}, StationInfo{name='E'}, StationInfo{name='C'}, StationInfo{na
The sequence Nodes of  DFT:[StationInfo{name='F'}, StationInfo{name='C'}, StationInfo{name='H'}, StationInfo{na
```



```
tionInfo{name='B'}, StationInfo{name='G'}, StationInfo{name='D'}, StationInfo{name='H'}, StationInfo{name='A'}]
tionInfo{name='G'}, StationInfo{name='E'}, StationInfo{name='B'}, StationInfo{name='A'}, StationInfo{name='D'}]
```

The results format is too long so that i divide it into two parts. The first line is the sequence node of BFT and the second line is the sequence node of DFT.

Task 12: Dijkstra’s algorithm:

Below is the code of Dijkstra’s algorithm:

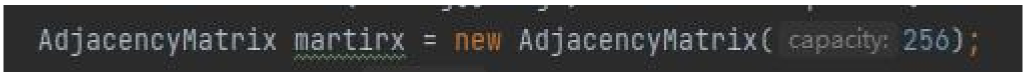
```
public double dijkstra(int start, int end) {
        assert 0 <= start && start < numStations : "start out of range!";
        assert 0 <= end && end < numStations : "end out of range!";

        boolean[] closed = new boolean[numStations];
        boolean[] open = new boolean[numStations];
        double[] g_value = new double[numStations];
        int[] previous = new int[numStations];

        for (int i = 0; i < numStations; i++) {
            closed[i] = false;
            open[i] = true;
            g_value[i] = NO_LINK;
            previous[i] = -1;
        }

        g_value[start] = 0;
```

```
int Counter = 0;    // Initialize the iteration counter
while (open[end]) {
    Counter++;    // Increment the counter

    double lowest_g = NO_LINK;
    int x = -1;
    for (int i = 0; i < numStations; i++) {
        if (open[i] && (x == -1 || g_value[i] < lowest_g)) {
            x = i;
            lowest_g = g_value[i];
        }
    }

    assert x != -1 : "Lowest g value error";

    open[x] = false;

    if (x != end) {
        for (int i = 0; i < numStations; i++) {
            if (open[i] && distance[x][i] != NO_LINK) {
                double new_g = g_value[x] + distance[x][i];
                if (new_g < g_value[i]) {
                    g_value[i] = new_g;
                    previous[i] = x;
```

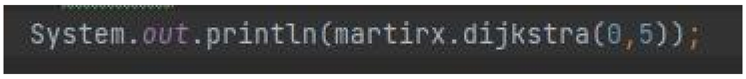
```
                            }
                        }
                    }
                }
            }

            System.out.println("Iteration Count: " + Counter);    // Print the
iteration count

            if (previous[end] != -1) {
                // The end station is reachable. Display the path.
                System.out.print("Shortest path: ");
                List<Integer> path = new ArrayList<>();
                for (int node = end; node != -1; node = previous[node]) {
                    path.add(node);
                }
                Collections.reverse(path);    // Reverse to get the correct order
                for (int node : path) {
                    System.out.print(node + " ");
                }
                System.out.print(    " the distance between them is: ");
            }

            return g_value[end];
        }}
```

The closed array is used to track the sites that have been visited, the open array is used to track the sites to be visited, the g_value array stores the minimum cost from the starting site to each site, and the previous array stores the previous site for each site and is used to backtrack the path**.**All sites are initially in the OPEN state, and all sites' g-values—aside from the starting site—are set to NO_LINK. Then it enters a loop where it first processes the site x with the lowest g-value out of all the unprocessed sites, putting it in the CLOSE state, and then it updates the g-values of all of x's nearby sites. All sites are initially in the OPEN state, and all sites' g-values—aside from the starting site—are set to NO_LINK. Then it enters a loop where it first processes the site x with the lowest g-value out of all the unprocessed sites, putting it in the CLOSE state, and then it updates the g-values of all of x's nearby sites until the end station are close state. Now, we gain the shortest path from the start station and end station.

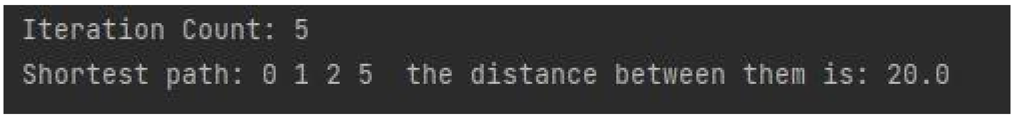
Test: The shortest path from 0 index to 5



```
AdjacencyMatrix martirx = new AdjacencyMatrix( capacity: 256);
```

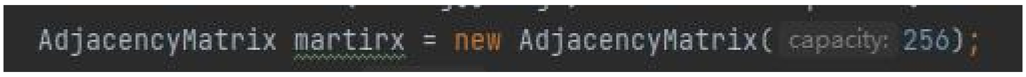


```
System.out.println(martirx.dijkstra(0,5));
```

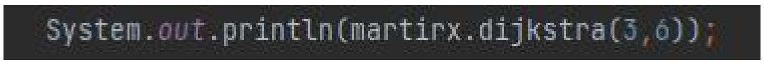


```
Iteration Count: 5
Shortest path: 0 1 2 5  the distance between them is: 20.0
```

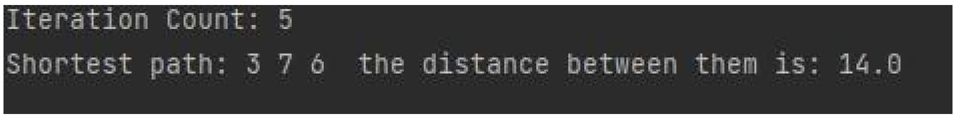
The shortest path from 3 index to 6



```
AdjacencyMatrix martirx = new AdjacencyMatrix( capacity: 256);
```



```
System.out.println(martirx.dijkstra(3,6));
```



```
Iteration Count: 5
Shortest path: 3 7 6  the distance between them is: 14.0
```

Task 13: Comparison between　Dijkstra’s algorithm and A* algorithm

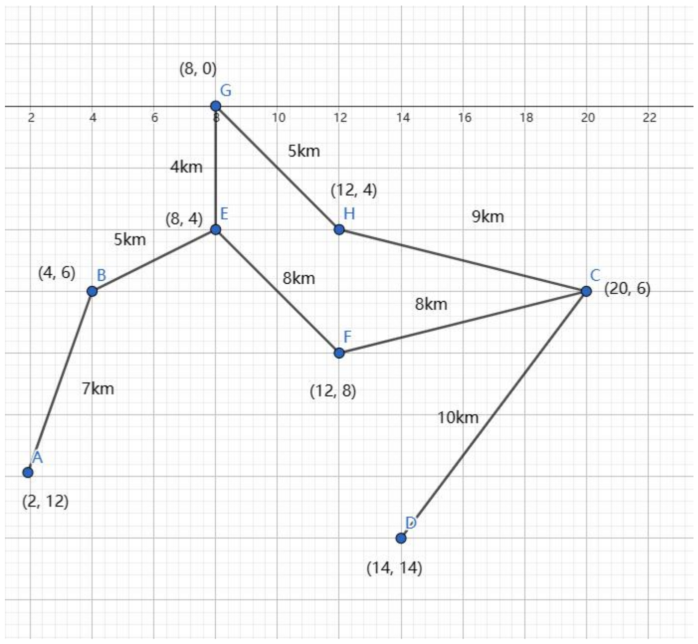
In Dijkstra's algorithm, it selects the current node with the shortest distance from the starting node for expansion. This ensures that the global shortest path is found.

However, when the graph is large or the target node is far from the starting node, the algorithm may waste a lot of time exploring regions far from the target.

The A* algorithm is a modification of Dijkstra's algorithm. It considers both the estimated distance-h-value from the current node to the target node as well as the distance g-value from the starting node to the current node. As a result, throughout the search process, the A* algorithm can traversal more deliberately in the search process , considerably increasing the search efficiency.

In my network:

In my network, the Dijkstra's algorithm will traverse nodes B, E, G, H and so on which are close to the start station, closing to the end station gradually ,if I want to calculate the shortest path from A to D. If we use the A* algorithm which contain the heuristic functions estimating the distance between the start point and the end point, it will move more directly toward node D and find the shortest path faster. For example, it may explore node B first, then E, F and C, and finally D, thus avoiding useless exploration of G and H.1



Task 14:Comment

Although the pseudo-code of Dijkstra's algorithm is provided in the appendix, it was still hard for me to understand. Therefore, I spent a long time searching for information and watching learning videos to understand the logic of the algorithm, and finally solved the problem.Additionally, I believe that my depth-first and breadth-first traversals are effectively implemented. I will send more time on the logic of the data structure in the future and do more exercise like the coursework because it helps me a lot.

Appendix

Class AdjacencyMatrix :

```
package CHC5223;


    import java.io.*;

    import java.util.*;


    public class AdjacencyMatrix {

        final double NO_LINK = Double.MAX_VALUE;

        int numStations; // 0 <= numStations <= capacity

        double distance[][];


        // List of objects of class StationInfo held in an array

        StationInfo[] stationInfos;


        public AdjacencyMatrix(int capacity) {

            this.numStations = 0;

            distance = new double[capacity][capacity];

            stationInfos = new StationInfo[capacity];
            // Initialize all distances to NO_LINK
            for (int i = 0; i < capacity; i++) {
                for (int j = 0; j < capacity; j++) {
                    distance[i][j] = NO_LINK;
                }
            }
        }



        public void addStation() {
            if (numStations >= distance.length) throw new
IllegalStateException("Capacity reached");
            numStations++;
            System.out.println("Station added. Total stations: " + numStations); //
Add this line
        }

        public void addLink(int station1, int station2, double distance) {
            if (station1 < 0 || station2 < 0 || station1 >= numStations || station2 >=
numStations)
                throw new IllegalArgumentException("Invalid station number");

            this.distance[station1][station2] = distance;
            this.distance[station2][station1] = distance;
        }
        public void readFile(String filename) throws IOException {
            Map<String, Integer> stationIndices = new HashMap<>();
            BufferedReader reader = new BufferedReader(new
FileReader(filename));
            String line;

            while ((line = reader.readLine()) != null) {
                line = line.trim();
                if (line.isEmpty()) {
                    continue;
                }

                String[] parts = line.split(" ");
                if (parts.length < 3) {
                    throw new IllegalArgumentException("Invalid line: " + line);
                }

                String type = parts[0];


                if (type.equals("station")) {
                    if (parts.length != 4) {
                        throw new IllegalArgumentException("Invalid line for
station: " + line);
                    }
                    String name = parts[1];
                    int x = Integer.parseInt(parts[2]);
                    int y = Integer.parseInt(parts[3]);
                    StationInfo station = new StationInfo(name, x, y);
                    addStation();
                    stationInfos[numStations - 1] = station;
                    stationIndices.put(name, numStations - 1);
                    System.out.println("Station added to array: " +
station.getName());
                } else if (type.equals("link")) {
                    if (parts.length != 4) {
                        throw new IllegalArgumentException("Invalid line for
link: " + line);
                    }
                    String station1Name = parts[1];
                    String station2Name = parts[2];
                    double dist = Double.parseDouble(parts[3]);

                    Integer station1Index = stationIndices.get(station1Name);
                    Integer station2Index = stationIndices.get(station2Name);

                    if (station1Index == null || station2Index == null) {
                        throw new IllegalArgumentException("Link references
unknown station: " + line);
                    }
```

```
                    addLink(station1Index, station2Index, dist);
                } else {
                    throw new IllegalArgumentException("Unknown type: " +
type);
                }
            }

            reader.close();
        }



        public void printStations() {
            System.out.println("Stations:");
            for (int i = 0; i < numStations; i++) {
                StationInfo station = stationInfos[i];
                System.out.println("Name: " + station.getName() + ", X Position:
" + station.getxPos() + ", Y Position: " + station.getyPos());
            }
        }

        public void printAdjacencyMatrix() {
            System.out.println("\nAdjacency Matrix:");
            System.out.print("          ");
            for (int i = 0; i < numStations; i++) {
                System.out.printf("%-9d", i);
            }
            System.out.println();
            StringBuilder separator = new StringBuilder();
            for (int i = 0; i < (numStations + 1) * 9; i++) {
                separator.append('-');
            }
            System.out.println(separator.toString());

            for (int i = 0; i < numStations; i++) {
                System.out.printf("%-4d| ", i);
                for (int j = 0; j < numStations; j++) {
                    if (distance[i][j] == NO_LINK) {
                        System.out.print("NO LINK    ");
                    } else {
                        System.out.printf("%.2f      ", distance[i][j]);
                    }
                }
                System.out.println("\n");
            }
        }


        public List<StationInfo> breadthFirstTraversal(int startStation) {
            Queue<Integer> Q = new LinkedList<>();
            List<StationInfo> L = new ArrayList<>();
```

```
            Q.add(startStation);


            while (!Q.isEmpty()) {
                int S = Q.poll();


                if (!L.contains(stationInfos[S])) {
                    L.add(stationInfos[S]);
                    for (int S2 = 0; S2 < numStations; S2++) {
                        if (distance[S][S2] != NO_LINK
&& !L.contains(stationInfos[S2])) {
                            Q.add(S2);
                        }
                    }
                }
            }


            return L;
        }




        public List<StationInfo> depthFirstTraversal(int startStation) {
            Stack<Integer> S = new Stack<>();
            List<StationInfo> L = new ArrayList<>();
            S.push(startStation);
            while (!S.isEmpty()) {
                int T = S.pop();
                if (!L.contains(stationInfos[T])) {
                    L.add(stationInfos[T]);



                    for (int T2 = 0; T2 < numStations; T2++) {
                        if (distance[T][T2] != NO_LINK
&& !L.contains(stationInfos[T2])) {
                            S.push(T2);
                        }
                    }
                }
            }

            return L;
        }

        public double dijkstra(int start, int end) {
            assert 0 <= start && start < numStations : "start out of range!";
            assert 0 <= end && end < numStations : "end out of range!";

            boolean[] closed = new boolean[numStations];
            boolean[] open = new boolean[numStations];
            double[] g_value = new double[numStations];
            int[] previous = new int[numStations];


            for (int i = 0; i < numStations; i++) {
                closed[i] = false;
                open[i] = true;
                g_value[i] = NO_LINK;
                previous[i] = -1;
            }


            g_value[start] = 0;


            int Counter = 0;
            while (open[end]) {
                Counter++;


                double lowest_g = NO_LINK;
                int x = -1;
                for (int i = 0; i < numStations; i++) {
                    if (open[i] && (x == -1 || g_value[i] < lowest_g)) {
                        x = i;
                        lowest_g = g_value[i];
                    }
                }


                assert x != -1 : "Lowest g value error";
                open[x] = false;

                if (x != end) {
                    for (int i = 0; i < numStations; i++) {
                        if (open[i] && distance[x][i] != NO_LINK) {
                            double new_g = g_value[x] + distance[x][i];
                            if (new_g < g_value[i]) {
                                g_value[i] = new_g;
                                previous[i] = x;
                            }
                        }
                    }
                }
            }

            System.out.println("Iteration Count: " + Counter);

            if (previous[end] != -1) {
                System.out.print("Shortest path: ");
                List<Integer> path = new ArrayList<>();
                for (int node = end; node != -1; node = previous[node]) {
                    path.add(node);
                }
                Collections.reverse(path);
```

```
                for (int node : path) {

                        System.out.print(node + " ");

                }

                System.out.print(    " the distance between them is: ");

        }


        return g_value[end];

}}
```

Class MemberBST:

```
package CHC5223;


public class MemberBST implements IMemberDB {

        private class Node {

                Member member;

                Node left;

                Node right;


                Node(Member member) {

                        this.member = member;

                        this.left = null;

                        this.right = null;

                }

        }
```

```
	private Node root;

	private int size;



		public MemberBST() {

			root = null;

			size = 0;

			System.out.println("Binary Search Tree");

		}


	@Override


	public Member put(Member member) {

		assert member != null : "Member should not be null";

		assert !member.getName().isEmpty() : "Member name should not be
empty";

		System.out.print("Adding member: " + member.getName());

		System.out.print(", Node visited: ");

		root = put(root, member);

		System.out.println();

		return null;

	}
```

```
        private Node put(Node node, Member member) {
                // edge judge
                if (node == null) {
                        size++;
                        return new Node(member);
                }

                System.out.print(node.member.getName() + " ");
                // member vs node->member
                int cmp = member.getName().compareTo(node.member.getName());

                // member < node->member, put to left
                if (cmp < 0) {
                        if (node.left != null &&
member.getName().equals(node.left.member.getName())) {
                                System.out.println("Duplicate member found: " +
member.getName());
                        }
                        node.left = put(node.left, member);
                }
                // member > node->member, put to right
                else if (cmp > 0) {
                        if (node.right != null &&
member.getName().equals(node.right.member.getName())) {
                                System.out.println("Duplicate member found: " +
member.getName());
```

```
                }
                node.right = put(node.right, member);
            }
            // member == node->member, ok
            else {
                node.member = member;
            }
            return node;
        }

        public Member get(String name) {
            assert name != null && !name.trim().isEmpty() : "Name to be gotten
should not be null or empty";
            System.out.print("Getting member: " + name);
            System.out.print(", Node visited: ");
            Node node = get(root, name);
            System.out.println();
            return node == null ? null : node.member;
        }

        private Node get(Node node, String name) {
            if (node == null) {
                return null;
            }
            System.out.print(node.member.getName() + " ");
            int cmp = name.compareTo(node.member.getName());
            if (cmp < 0) {
                return get(node.left, name);
            } else if (cmp > 0) {
                return get(node.right, name);
            } else {
                return node;
            }
        }


        public Member remove(String name) {
            assert name != null && !name.isEmpty() : "Name to be removed
should not be null or empty";

            Node parent = null, del, p = null, q = null;
            Member result;
            del = root;
            while (del != null && !del.member.getName().equals(name)) {
                parent = del;
                if (name.compareTo(del.member.getName()) < 0)
                    del = del.left;
                else
                    del = del.right;
            }
            if (del != null) {
                if (del == root) {
                    System.out.println("Removing member: " + name + ", Node
visited: " + del.member.getName());
                    System.out.println("(This deleted node is the root node and
the node visited is itself)");
                } else {
                    System.out.println("Removing member: " + name + ", Node
visited: " + del.member.getName());
                }

                if (del.right == null) p = del.left;
                else if (del.right.left == null) {
                    p = del.right;
                    p.left = del.left;
                } else {
                    p = del.right;
                    while (p.left != null) {
                        q = p;
                        p = p.left;
                    }
                    q.left = p.right;
                    p.left = del.left;
                    p.right = del.right;
                }
```

```
                        if (del == root) root = p;
                        else if
(del.member.getName().compareTo(parent.member.getName()) < 0)
                            parent.left = p;
                        else
                            parent.right = p;

                        result = del.member;
                    } else {
                        System.out.println("Node not found");
                        result = null;
                    }

                    return result;
                }
    //
    //Use for show member in alphabetical order
                public void displayDB() {
                    if (root == null) {
                        System.out.println("The database is empty.");
                        return;
                    }

                    System.out.println("Members in alphabetical order:");
```

```
            displayInOrder(root);
        }

        private void displayInOrder(Node node) {
            if (node != null) {
                displayInOrder(node.left);
                System.out.println("Name: " + node.member.getName() + ", 
Affiliation: " + node.member.getAffiliation());
                displayInOrder(node.right);
            }
        }




        public boolean containsName(String name) {
            assert name != null && !name.trim().isEmpty() : "Name to be searched 
should not be null or empty";
            return containsName(name, root);
        }

        private boolean containsName(String name, Node node) {
            if (node == null) {
                return false;
            }

            int compare = name.compareTo(node.member.getName());
            if (compare < 0) {
                return containsName(name, node.left);
            } else if (compare > 0) {
                return containsName(name, node.right);
            } else {
                return true;
            }
        }



        public void clearDB() {
            root = null;
            size = 0;
        }

        public int size() {
            return this.size;
        }

        public boolean isEmpty() {
            return size == 0;
        }



    //This display method is used to showed the structure of tree
```

```
//	public void displayDB() {
//		if (root == null) {
//			System.out.println("The database is empty.");
//			return;
//		}
//
//		System.out.println("Members showed in tree model");
//		displayInOrder(root, "");
//	}
//
//	private void displayInOrder(Node node, String prefix) {
//		if (node != null) {
//			displayInOrder(node.right, prefix + "        ");
//
//			if (node.right != null) {
//				System.out.println(prefix + " /");
//			}
//
//			System.out.println(prefix + node.member.getName() + ", " +
node.member.getAffiliation());
//
//			if (node.left != null) {
//				System.out.println(prefix + " \\");
//			}
//
//                      displayInOrder(node.left, prefix + "        ");

//              }

//      }

}
```

Class CHC5223_PartB_test:

```
package CHC5223;


        import java.io.IOException;


        public class CHC5223_PartB_test {


          public static void main(String[] args) throws IOException {

                  AdjacencyMatrix martirx = new AdjacencyMatrix(256);

                  martirx.readFile("graphInfo.txt");

                  martirx.printStations();

                  martirx.printAdjacencyMatrix();

                  System.out.println("The sequence Nodes of    BFT:" +
martirx.breadthFirstTraversal(5).toString());

                  System.out.println("The sequence Nodes of    DFT:"
+martirx.depthFirstTraversal(5));

                  System.out.println(martirx.dijkstra(3,6));
```

```
    }}
```