CHC5223_CW2_Report

Student ID: 202118010418

Catalogue

一、 Part A – binary search tree

Task_1

To accomplish this task, I create a Java class called MemberBST that implements the IMemberDB interface and uses a binary search Tree (BST) as its underlying data structure. Here are my design ideas and rationale:

- Class Definition: The MemberBST class is defined to implement the IMemberDB interface.
- Inner Node Class: Inside the MemberBST class, we define a private inner class Node to represent a node in the binary search tree. Each node contains a Member object (as the data of the node) and two references of type Node (pointing to the left and right child nodes).
- Root Node and Size: In the MemberBST class, we also maintain a reference to the root node and an integer representing the size of the tree (i.e., the number of nodes).
- Constructor: The constructor of MemberBST initializes the root node (as null, indicating an empty tree) and the size of the tree (as 0). It also prints a message indicating the creation of a binary search tree.
- Insert Operation: The put method is used to insert a new member into the BST. It starts from the root node and traverses down the tree until it finds an appropriate spot to insert the new node. If the new member's name is the same as an existing node's name, then the new member will replace

the existing node's data.

- Search Operation: The get method is used to retrieve a member from the BST. It starts from the root node and traverses down the tree until it finds the node with the specified name.
- Delete Operation: The remove method is used to remove a member from the BST. It locates the node to be removed and then takes different actions based on the number of child nodes of that node: if the node has two child nodes, it is replaced by its successor node; if the node has only one child node, it is replaced by its only child node; if the node has no child nodes, it is directly deleted.
- Other Methods: The MemberBST class also provides several other methods, such as containsName (to check whether a member with a given name exists), size (to return the size of the BST), isEmpty (to check whether the BST is empty), displayDB (to print out all the member information in order), and clearDB (to clear the BST).
- Logging: To track the execution of each operation, we also define a log method, which writes detailed information about the operation (including the operation type, member name, and sequence of visited nodes) into a log file.

```
package CHC5223;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
```

```java
import java.util.List;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;


// The MemberBST class that implements the IMemberDB interface
public class MemberBST implements IMemberDB {
    // A list to store the sequence of visited nodes
    private final List<String> visitedNodes;

    // Inner class Node representing a node in the binary search
tree
    private class Node {
        // The data stored in the node, which is a Member object
        private Member data;
        // The left and right child nodes of the current node
        private Node left, right;

        // Constructor to initialize the node data with a Member
object
        public Node(Member data) {
            this.data = data;
            left = null;
            right = null;
        }
    }

    // The root node of the binary search tree
    private Node root;
    // The size of the binary search tree, indicating the number
of nodes
    private int size;

    // Constructor to initialize the binary search tree
    public MemberBST() {
        // Print a message to the console indicating the creation
of a binary search tree
        System.out.println("Binary Search Tree");
        // Initialize the root node as null, indicating an empty
tree
        root = null;
        // Initialize the size of the tree as 0
        size = 0;
        // Initialize the list of visited nodes as an empty
```

```
ArrayList
        visitedNodes = new ArrayList<>();
    }


    // Method to check if a member with the specified name exists
in the database
    @Override
    public boolean containsName(String name) {
        // Assert that the name is not null and not an empty
string
        assert name != null && !name.isEmpty() : "Name cannot be
null or empty";
        // Return true if the member with the given name is found,
false otherwise
        return get(name) != null;
    }

    // Method to insert a member into the database
    @Override
    public Member put(Member member) {
        // Assert that the member and the member's name are not
null and not an empty string
        assert member != null && member.getName() != null
&& !member.getName().isEmpty() : "Member or member name cannot be
null or empty";

        // If the root is null (i.e., the tree is empty)
        if (root == null) {
            // Create a new node with the member as the data and
assign it to the root
            root = new Node(member);
            // Add a log entry indicating that the root was empty
            visitedNodes.add("(Empty root node)");
            // Increment the size of the tree
            size++;
            // Log the operation
            log("put", member.getName());
            // Clear the list of visited nodes
            visitedNodes.clear();
            // Return null as there was no previous member with
the same name
            return null;
        }
```

```
        // Initialize the current node to the root and the parent
node to null
        Node current = root;
        Node parent = null;
        // Initialize a variable to hold the result of the
comparison between the member's name and the current node's
data's name
        int cmp = 0;

        // While the current node is not null
        while (current != null) {
            // Set the parent node to the current node
            parent = current;
            // Compare the member's name with the current node's
data's name
            cmp =
member.getName().compareTo(current.data.getName());

            // If the member's name is less than the current
node's data's name
            if (cmp < 0) {
                // Add a log entry indicating that we're going to
the left child
                visitedNodes.add(current.data.getName() + "(L)");
                // Move to the left child
                current = current.left;
            }
            // If the member's name is greater than the current
node's data's name
            else if (cmp > 0) {
                // Add a log entry indicating that we're going to
the right child
                visitedNodes.add(current.data.getName() + "(R)");
                // Move to the right child
                current = current.right;
            }
            // If the member's name is equal to the current
node's data's name
            else {
                // Add a log entry indicating that we've found
the node
                visitedNodes.add(current.data.getName());
                // Store the current node's data in a temporary
```

```
variable
                Member oldData = current.data;
                // Replace the current node's data with the new
member
                current.data = member;
                // Log the operation
                log("put", member.getName());
                // Clear the list of visited nodes
                visitedNodes.clear();
                // Return the old data
                return oldData;
            }
        }

        // Create a new node with the member as the data
        Node newNode = new Node(member);
        // If the member's name is less than the parent node's
data's name
        if (cmp < 0) {
            // Add a log entry indicating that we're adding a
left child
            visitedNodes.add("(Empty left child)");
            // Add the new node as the left child of the parent
node
            parent.left = newNode;
        }
        // If the member's name is greater than the parent node's
data's name
        else {
            // Add a log entry indicating that we're adding a
right child
            visitedNodes.add("(Empty right child)");
            // Add the new node as the right child of the parent
node
            parent.right = newNode;
        }

        // Increment the size of the tree
        size++;
        // Log the operation
        log("put", member.getName());
        // Clear the list of visited nodes
        visitedNodes.clear();
        // Return null as there was no previous member with the
```

```
same name
        return null;
    }

    // Method to remove a member from the database
    @Override
    public Member remove(String name){
        // Assert that the name is not null and not an empty
string
        assert name != null && !name.isEmpty() : "Name cannot be
null or empty";
        // Define the parent node, the node to be deleted, and
two auxiliary nodes
        Node parent = null, del, p = null, q = null;
        // Define the result member
        Member result;
        // Start searching from the root node
        del = root;
        // While the node to be deleted is not null and the name
of the data in the node to be deleted is not equal to the input
name, continue searching
        while (del != null && !del.data.getName().equals(name)) {
            // Update the parent node to the current node to be
deleted
            parent = del;
            // Add the name of the data in the node to be deleted
to the visitedNodes list
            visitedNodes.add(del.data.getName());
            // If the input name is less than the name of the
data in the node to be deleted, search in the left subtree
            if (name.compareTo(del.data.getName()) < 0)
                del = del.left;
                // Otherwise, search in the right subtree
            else
                del = del.right;
        }

        // If the node to be deleted is found
        if(del != null) {
            // Add the name of the data in the node to be deleted
to the visitedNodes list
            visitedNodes.add(del.data.getName());

            // If the right child of the node to be deleted is
```

```
null, p points to the left child of the node to be deleted
            if (del.right == null) {
                p = del.left;
                // Add the name of the data in the p node to the
visitedNodes list
                if (p != null) {
                    visitedNodes.add(p.data.getName());
                }
            }
            // If the left child of the right child of the node
to be deleted is null, p points to the right child of the node to
be deleted, and the left child of the node to be deleted is
assigned to the left child of p
            else if (del.right.left == null) {
                p = del.right;
                // Add the name of the data in the p node to the
visitedNodes list
                visitedNodes.add(p.data.getName());
                p.left = del.left;
            } else {
                // Otherwise, p points to the right child of the
node to be deleted, and then searches to the leftmost node
                p = del.right;
                while (p.left != null) {
                    // Add the name of the data in the p node to
the visitedNodes list
                    visitedNodes.add(p.data.getName());
                    q = p;
                    p = p.left;
                }
                // Add the name of the data in the p node to the
visitedNodes list
                visitedNodes.add(p.data.getName());
                // The left child of q points to the right child
of p, the left child of p points to the left child of the node to
be deleted, and the right child of p points to the right child of
the node to be deleted
                q.left = p.right; p.left = del.left; p.right =
del.right;
            }
            // If the node to be deleted is the root node, the
root node points to p
            if(del == root) root = p;
                // If the name of the data in the node to be
```

```
    deleted is less than the name of the data in the parent node, the
    left child of the parent node points to p
                else if
    (del.data.getName().compareTo(parent.data.getName()) < 0)
                    parent.left = p;
                    // Otherwise, the right child of the parent node
    points to p
                else parent.right = p;
                // The result member is the data of the node to be
    deleted
                result = del.data;
                // Decrement the size of the binary search tree
                size--;
            }
            // If the node to be deleted is not found, the result
    member is null
            else result = null;

            // Log the operation
            log("remove", name);
            // Clear the visitedNodes list
            visitedNodes.clear();
            // Return the result member
            return result;
        }

        // Method to get a member with the specified name
        @Override
        public Member get(String name) {
            // Assert that the name is not null and not an empty
    string
            assert name != null && !name.isEmpty() : "Name cannot be
    null or empty";
            // Call the getNode method to find the node with the
    specified name starting from the root
            Node node = getNode(name, root);
            // Log the operation
            log("get", name);
            // Clear the list of visited nodes
            visitedNodes.clear();
            // If the node is null (i.e., not found), return null;
    otherwise, return the data of the node
            return node == null ? null : node.data;
        }
```

```
    // Private method to get a node with the specified name
    private Node getNode(String name, Node node) {
        // Assert that the name is not null and not an empty
string
        assert name != null && !name.isEmpty() : "Name cannot be
null or empty";
        // If the node is null (i.e., we've reached a leaf node
and haven't found the specified name), return null
        if (node == null) {
            visitedNodes.add("(Empty node)");
            return null;
        }
        // Compare the specified name with the name of the data
in the node
        int cmp = name.compareTo(node.data.getName());
        // If the specified name is less than the name of the
data in the node, search in the left subtree
        if (cmp < 0) {
            visitedNodes.add(node.data.getName()+"(L)");
            return getNode(name, node.left);
        }
        // If the specified name is greater than the name of the
data in the node, search in the right subtree
        else if (cmp > 0) {
            visitedNodes.add(node.data.getName()+"(R)");
            return getNode(name, node.right);
        }
        // If the specified name is equal to the name of the data
in the node, return the node
        else {
            visitedNodes.add(node.data.getName());
            return node;
        }
    }

    // Method to get the size of the database
    @Override
    public int size() {
        // Return the size of the binary search tree
        return size;
    }

    // Method to check if the database is empty
```

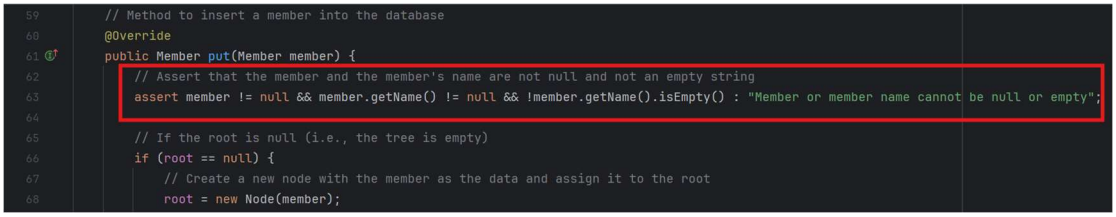
```
    @Override
    public boolean isEmpty() {
        // Return true if the size of the binary search tree is 0,
false otherwise
        return size == 0;
    }

    // Method to display all members in the database
    @Override
    public void displayDB() {
        // Call the private displayDB method with the root node
as the argument
        displayDB(root);
    }

    // Private recursive method to display all nodes in the
binary search tree
    private void displayDB(Node node) {
        // If the node is not null
        if (node != null) {
            // Recursively call the method with the left child of
the current node
            displayDB(node.left);
            // Print the name and affiliation of the member in
the current node
            System.out.println(node.data.getName() + " " +
node.data.getAffiliation());
            // Recursively call the method with the right child
of the current node
            displayDB(node.right);
        }
    }

    // Method to clear the database
    @Override
    public void clearDB() {
        // Set the root node to null
        root = null;
        // Set the size of the binary search tree to 0
        size = 0;
    }

    // Define a private method for logging operations
    private void log(String operation, String name) {
```

```
        // Specify the log file name
        String logFile = "LogInformation.txt";
        // Use try-with-resources to ensure the BufferedWriter is
closed at the end
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(logFile, true))) {
            // Get the current date and time
            LocalDateTime now = LocalDateTime.now();
            // Create a DateTimeFormatter
            DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
            // Format the current date and time as a string
            String nowString = now.format(formatter);

            // Construct the log message with the current date
and time, operation, member name, and visited node sequence
            String logMessage = "Time: "+ nowString + " |
Operation: " + operation + " | Member Name: " + name + " |
Visited Node Sequence : ";
            // Write the log message to the log file
            writer.write(logMessage);
            // Call the private method to write the node sequence
to the log file
            writeNodeSequence(writer, visitedNodes);
            // Write a new line to the log file
            writer.newLine();
            // Print the log message to the console
            System.out.println(logMessage + String.join("->",
visitedNodes));
        } catch (IOException e) {
            // Print the stack trace for any IOException
            e.printStackTrace();
        }
    }

    // Define a private method for writing the node sequence to
the log file
    private void writeNodeSequence(BufferedWriter
writer,List<String> visitedNodes) throws IOException {
        // For each visited node
        for (String visitedNode : visitedNodes) {
            // Write the visited node to the log file, prefixed
with "->"
            writer.write("->" + visitedNode);
```

```
                }
            }
        }
```

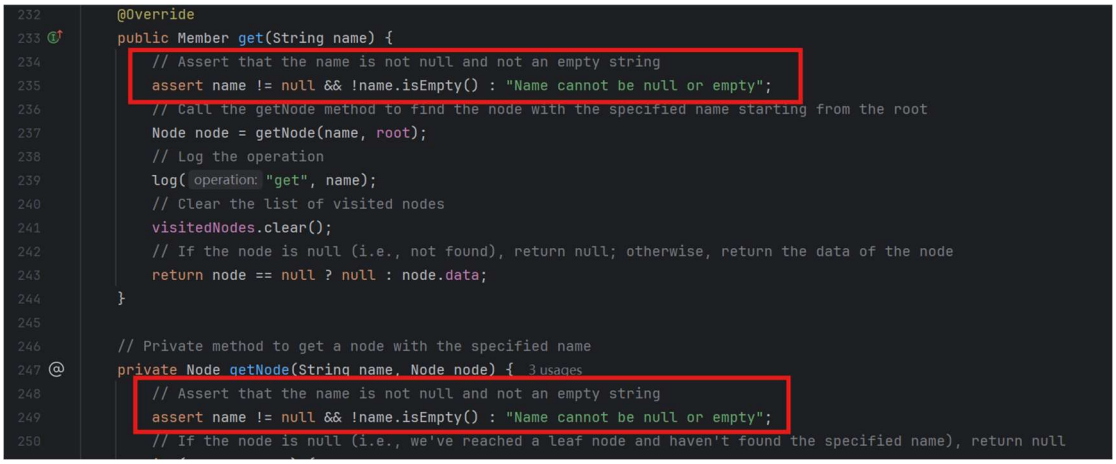
Task_2

- In the ***put*** method, an assertion is used to ensure that the member object and the member's name are not null and not an empty string. This is a precondition for the put operation because the member's name is used as a key in the binary search tree. If the member or its name were null or an empty string, the put operation would not be able to proceed correctly.

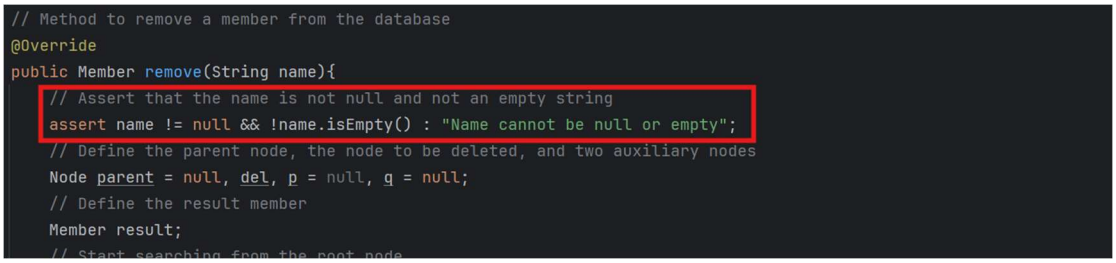


```
59      // Method to insert a member into the database
60      @Override
61      public Member put(Member member) {
62          // Assert that the member and the member's name are not null and not an empty string
63          assert member != null && member.getName() != null && !member.getName().isEmpty() : "Member or member name cannot be null or empty";
64
65          // If the root is null (i.e., the tree is empty)
66          if (root == null) {
67              // Create a new node with the member as the data and assign it to the root
68              root = new Node(member);
```

- In the ***get***, ***getNode*** and ***remove*** methods, an assertion is used to ensure that the name parameter is not null and not an empty string. This is a precondition for these operations because the name is used to locate a member in the binary search tree. If the name were null or an empty string, these operations would not be able to proceed correctly.



```java
@Override
public Member get(String name) {
    // Assert that the name is not null and not an empty string
    assert name != null && !name.isEmpty() : "Name cannot be null or empty";
    // Call the getNode method to find the node with the specified name starting from the root
    Node node = getNode(name, root);
    // Log the operation
    log( operation: "get", name);
    // Clear the list of visited nodes
    visitedNodes.clear();
    // If the node is null (i.e., not found), return null; otherwise, return the data of the node
    return node == null ? null : node.data;
}

// Private method to get a node with the specified name
private Node getNode(String name, Node node) {  3 usages
    // Assert that the name is not null and not an empty string
    assert name != null && !name.isEmpty() : "Name cannot be null or empty";
    // If the node is null (i.e., we've reached a leaf node and haven't found the specified name), return null
```



```java
// Method to remove a member from the database
@Override
public Member remove(String name){
    // Assert that the name is not null and not an empty string
    assert name != null && !name.isEmpty() : "Name cannot be null or empty";
    // Define the parent node, the node to be deleted, and two auxiliary nodes
    Node parent = null, del, p = null, q = null;
    // Define the result member
    Member result;
    // Start searching from the root node
```

Task_3

The MemberBST class already includes a logging mechanism that logs the operations put, get, and remove. The logging is done in the ***log*** method, which writes the operation, the member name, and the sequence of visited nodes to a text file named LogInformation.txt. The sequence of visited nodes is stored in the visitedNodes list, which is cleared after each operation. The put, get, and remove methods all call the log method after performing their operations. The log method writes a log entry to the LogInformation.txt file, which includes the current date and time, the operation performed, the member name, and the sequence of visited nodes. The sequence of visited nodes is represented as a string, with each node separated by "->".

The log method: In the MemberBST class, the implementation of logging is

mainly done through the log method. This method accepts two arguments: the type of operation (for example, "put", "get", "remove") and the member name. The main task of this method is to write the details of the operation to a text file called LogInformation.txt. In the log method, we first get the current date and time and format it as a string. Then, a log message is constructed, including the current date and time, operation type, member name, and the sequence of nodes visited. This sequence of nodes is obtained through the visitedNodes list, which is emptied after each operation. Then, the log message is written to the LogInformation.txt file using BufferedWriter. During the write process, each visited node is added to the log message, separated by "->". Finally, log messages are printed to the console.

```
// Define a private method for logging operations
private void log(String operation, String name) {
    // Specify the log file name
    String logFile = "LogInformation.txt";
    // Use try-with-resources to ensure the BufferedWriter is closed
at the end
    try (BufferedWriter writer = new BufferedWriter(new
FileWriter(logFile, true))) {
        // Get the current date and time
        LocalDateTime now = LocalDateTime.now();
        // Create a DateTimeFormatter
        DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        // Format the current date and time as a string
        String nowString = now.format(formatter);

        // Construct the log message with the current date and time,
operation, member name, and visited node sequence
        String logMessage = "Time: "+ nowString + " | Operation: " +
operation + " | Member Name: " + name + " | Visited Node Sequence : ";
        // Write the log message to the log file
        writer.write(logMessage);
```

```
        // Call the private method to write the node sequence to the
log file
        writeNodeSequence(writer, visitedNodes);
        // Write a new line to the log file
        writer.newLine();
        // Print the log message to the console
        System.out.println(logMessage + String.join("->",
visitedNodes));
    } catch (IOException e) {
        // Print the stack trace for any IOException
        e.printStackTrace();
    }
}

// Define a private method for writing the node sequence to the log
file
private void writeNodeSequence(BufferedWriter writer,List<String>
visitedNodes) throws IOException {
    // For each visited node
    for (String visitedNode : visitedNodes) {
        // Write the visited node to the log file, prefixed with "->"
        writer.write("->" + visitedNode);
    }
}
```

A list that records the order of the visited nodes:

```
// The MemberBST class that implements the IMemberDB interface
public class MemberBST implements IMemberDB {  3 usages
    // A list to store the sequence of visited nodes
    private final List<String> visitedNodes;  24 usages
```

Each time the method is called, the time and member name are printed first, and they are inserted into the record list when the node is cyclically visited. Finally, the order of visited nodes is printed. At the end of the method, the list of visited nodes is emptied for the next method to record. For example,

```java
// Method to insert a member into the database
@Override
public Member put(Member member) {
    // Assert that the member and the member's name are not null and not an empty string
    assert member != null && member.getName() != null && !member.getName().isEmpty() : "Member or member name cannot be null or empty";

    // If the root is null (i.e., the tree is empty)
    if (root == null) {
        // Create a new node with the member as the data and assign it to the root
        root = new Node(member);
        // Add a log entry indicating that the root was empty
        visitedNodes.add("(Empty root node)");
        // Increment the size of the tree
        size++;
        // Log the operation
        log( operation: "put", member.getName());
        // Clear the list of visited nodes
        visitedNodes.clear();
        // Return null as there was no previous member with the same name
        return null;
    }

    // Initialize the current node to the root and the parent node to null
    Node current = root;
    Node parent = null;
    // Initialize a variable to hold the result of the comparison between the member's name and the current node's data's name
    int cmp = 0;

    // While the current node is not null
    while (current != null) {
        // Set the parent node to the current node
        parent = current;
        // Compare the member's name with the current node's data's name
        cmp = member.getName().compareTo(current.data.getName());
```

```java
            // If the member's name is less than the current node's data's name
            if (cmp < 0) {
                // Add a log entry indicating that we're going to the left child
                visitedNodes.add(current.data.getName() + "(L)");
                // Move to the left child
                current = current.left;
            }
            // If the member's name is greater than the current node's data's name
            else if (cmp > 0) {
                // Add a log entry indicating that we're going to the right child
                visitedNodes.add(current.data.getName() + "(R)");
                // Move to the right child
                current = current.right;
            }
            // If the member's name is equal to the current node's data's name
            else {
                // Add a log entry indicating that we've found the node
                visitedNodes.add(current.data.getName());
                // Store the current node's data in a temporary variable
                Member oldData = current.data;
                // Replace the current node's data with the new member
                current.data = member;
                // Log the operation
                log( operation: "put", member.getName());
                // Clear the list of visited nodes
                visitedNodes.clear();
                // Return the old data
                return oldData;
            }
        }

        // Create a new node with the member as the data
```

```
        // Create a new node with the member as the data
        Node newNode = new Node(member);
        // If the member's name is less than the parent node's data's name
        if (cmp < 0) {
            // Add a log entry indicating that we're adding a left child
            visitedNodes.add("(Empty left child)");
            // Add the new node as the left child of the parent node
            parent.left = newNode;
        }
        // If the member's name is greater than the parent node's data's name
        else {
            // Add a log entry indicating that we're adding a right child
            visitedNodes.add("(Empty right child)");
            // Add the new node as the right child of the parent node
            parent.right = newNode;
        }

        // Increment the size of the tree
        size++;
        // Log the operation
        log( operation: "put", member.getName());
        // Clear the list of visited nodes
        visitedNodes.clear();
        // Return null as there was no previous member with the same name
        return null;
    }
```

(Log Information)

```
Time: 2024-05-26 16:11:40 | Operation: put | Member Name: Tomkiewicz, Aleshia | Visited Node Sequence : ->(Empty root node)
Time: 2024-05-26 16:11:40 | Operation: put | Member Name: Zigomalas, Evan | Visited Node Sequence : ->Tomkiewicz, Aleshia(R)->(Empty right child)
Time: 2024-05-26 16:11:40 | Operation: put | Member Name: Andrade, France | Visited Node Sequence : ->Tomkiewicz, Aleshia(L)->(Empty left child)
```

Task_4

To design the implemented test plan, I used Junit to create test cases covering different scenarios of deleting nodes in a binary search tree. Here's the test case:

- Test Deleting a Leaf Node: This test case will check if the implementation correctly handles the deletion of a leaf node. A leaf node is a node that has no children. The test will involve creating a binary search tree, adding nodes to it, deleting a leaf node, and then checking if the node was correctly removed. When removing a leaf node, the goal of the algorithm is to find the node to remove and set the corresponding child pointer of its parent to null. A leaf node is a node that has no children, so removing it won't affect the rest of the tree.

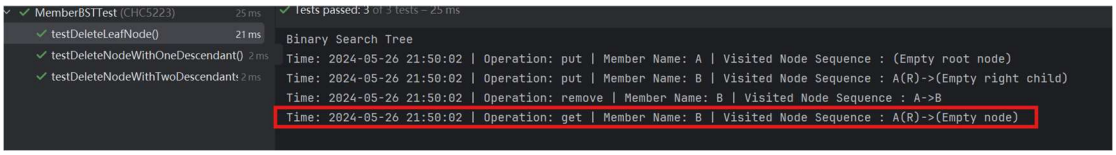
Actual design:

Create and put in, in turn, node A and node B. In this case, node B will be a leaf node of the tree and will not have any descendants. Delete the B node and the tree will contain no B.

The result shows that after deletion, the B leaf node has been successfully deleted as a leaf node, and the original position has been null by calling get (B). The test was successful.

```
package CHC5223;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MemberBSTTest {
    private MemberBST memberBST;  18 usages

    @BeforeEach
    public void setUp() { memberBST = new MemberBST(); }

    @Test
    public void testDeleteLeafNode() {
        Member member1 = new Member( name: "A",  affiliation: "aaaaa");
        Member member2 = new Member( name: "B",  affiliation: "bbbbb");
        memberBST.put(member1);
        memberBST.put(member2);
        memberBST.remove( name: "B");
        assertNull(memberBST.get("B"));
    }
```

Test Result:



- Test Deleting a Node with One Descendant: This test case will check if the implementation correctly handles the deletion of a node that has only one child. The test will involve creating a binary search tree, adding nodes to it, deleting a node with one child, and then checking if the node was correctly removed and the child node is correctly linked to the rest of the tree. When deleting a node with one descendant, the goal of the algorithm is to find a replacement node to replace the deleted node while maintaining the nature of a binary search tree. This replacement node is usually the only child of the removed node.

  Here are the steps in detail:

1. First, we find the node to remove. During this process, we record the visited nodes for subsequent logging.
2. If the node to be removed is found, we check whether its right child is empty. If it is empty, it means that the removed node has only one left child, and we use this left child as a replacement node.
3. After finding the replacement node, we need to link the replacement node into the tree. If the removed node is the root node, we set the replacement node as the new root node. Otherwise, we need to decide whether the removed node is the left or right child of its parent and link

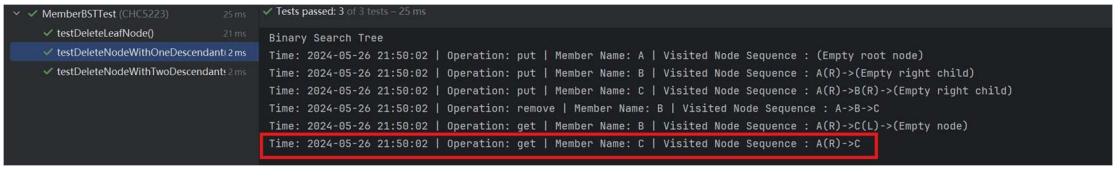
the replacement node to the parent accordingly.

Actual design:

Create and place, in turn, node A, node B and node C. At this point, node B will be a descendant of node C in alphabetical order. Deleting node B removes B from the tree, and node C should take B's place.

The result shows that when node B is removed, the tree does not contain node B, and node C becomes A descendant of node A, taking its place. The test was successful.

```
@Test
public void testDeleteNodeWithOneDescendant() {
    Member member1 = new Member( name: "A",  affiliation: "aaaaa");
    Member member2 = new Member( name: "B",  affiliation: "bbbbb");
    Member member3 = new Member( name: "C",  affiliation: "ccccc");
    memberBST.put(member1);
    memberBST.put(member2);
    memberBST.put(member3);
    memberBST.remove( name: "B");
    assertNull(memberBST.get("B"));
    assertNotNull(memberBST.get("C"));
}
```



Test deletion of nodes with two descendants: When deleting a node with two descendants, the goal of the algorithm is to find a replacement node to

replace the deleted node while maintaining the binary search tree property. This replacement node is usually the leftmost node of the right subtree of the removed node, or the rightmost node of the left subtree. In this implementation, we chose the leftmost node in the right subtree as a replacement.

Here are the steps in detail:

1. First, we find the node to remove. During this process, we record the visited nodes for subsequent logging.
2. If the node to be removed is found, we check whether the left child of its right child is empty. If it is empty, it means that the right child is the replacement node we are looking for. We assign the left child of the removed node to the left child of the replacement node.
3. If the left child of the right child is not empty, we need to find the leftmost node in the right subtree as a replacement node. In this process, we also record the visited nodes.
4. After finding the replacement node, we need to adjust the structure of the tree. We assign the right child of the replacement node to the left child of its parent node, and then assign the left and right child of the removed node to the left and right child of the replacement node, respectively.
5. Finally, we need to link alternative nodes into the tree. If the removed node is the root node, we set the replacement node as the new root node. Otherwise, we need to decide whether the removed node is the left or

right child of its parent and link the replacement node to the parent accordingly.

Actual design:

Create and place, in turn, node B, node A and node C. At this point, node B will be the node with two descendants of node C and node A in alphabetical order. If node B is deleted, the tree no longer contains node B, and node C should take its place.

The result shows that when node B is removed, the tree does not contain node B, and node C becomes the root node, taking its place. The test was successful.

```
public void testDeleteNodeWithTwoDescendants() {
    Member member1 = new Member( name: "B", affiliation: "bbbbb");
    Member member2 = new Member( name: "A", affiliation: "aaaaa");
    Member member3 = new Member( name: "C", affiliation: "ccccc");
    memberBST.put(member1);
    memberBST.put(member2);
    memberBST.put(member3);
    memberBST.remove( name: "B");
    assertNull(memberBST.get("B"));
    assertNotNull(memberBST.get("A"));
    assertNotNull(memberBST.get("C"));
}
```

```
MemberBSTTest (CHC5223) 25 ms
  testDeleteLeafNode() 21 ms
  testDeleteNodeWithOneDescendant 2 ms
  testDeleteNodeWithTwoDescendant 2 ms
Tests passed: 3 of 3 tests – 25 ms

Binary Search Tree
Time: 2024-05-26 21:50:02 | Operation: put | Member Name: B | Visited Node Sequence : (Empty root node)
Time: 2024-05-26 21:50:02 | Operation: put | Member Name: A | Visited Node Sequence : B(L)->(Empty left child)
Time: 2024-05-26 21:50:02 | Operation: put | Member Name: C | Visited Node Sequence : B(R)->(Empty right child)
Time: 2024-05-26 21:50:02 | Operation: remove | Member Name: B | Visited Node Sequence : B->C
Time: 2024-05-26 21:50:02 | Operation: get | Member Name: B | Visited Node Sequence : C(L)->A(R)->(Empty node)
Time: 2024-05-26 21:50:02 | Operation: get | Member Name: A | Visited Node Sequence : C(L)->A
Time: 2024-05-26 21:50:02 | Operation: get | Member Name: C | Visited Node Sequence : C
```

Class MemberBSTTest:

```
package CHC5223;
```

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MemberBSTTest {
    private MemberBST memberBST;

    @BeforeEach
    public void setUp() {
        memberBST = new MemberBST();
    }

    @Test
    public void testDeleteLeafNode() {
        Member member1 = new Member("A", "aaaaa");
        Member member2 = new Member("B", "bbbbb");
        memberBST.put(member1);
        memberBST.put(member2);
        memberBST.remove("B");
        assertNull(memberBST.get("B"));
    }

    @Test
    public void testDeleteNodeWithOneDescendant() {
        Member member1 = new Member("A", "aaaaa");
        Member member2 = new Member("B", "bbbbb");
        Member member3 = new Member("C", "ccccc");
        memberBST.put(member1);
        memberBST.put(member2);
        memberBST.put(member3);
        memberBST.remove("B");
        assertNull(memberBST.get("B"));
        assertNotNull(memberBST.get("C"));
    }

    @Test
    public void testDeleteNodeWithTwoDescendants() {
        Member member1 = new Member("B", "bbbbb");
        Member member2 = new Member("A", "aaaaa");
        Member member3 = new Member("C", "ccccc");
        memberBST.put(member1);
        memberBST.put(member2);
        memberBST.put(member3);
        memberBST.remove("B");
```

```
        assertNull(memberBST.get("B"));
        assertNotNull(memberBST.get("A"));
        assertNotNull(memberBST.get("C"));
    }
}
```

Task_5

Test: The provided main program (initially loaded using the provided text file)

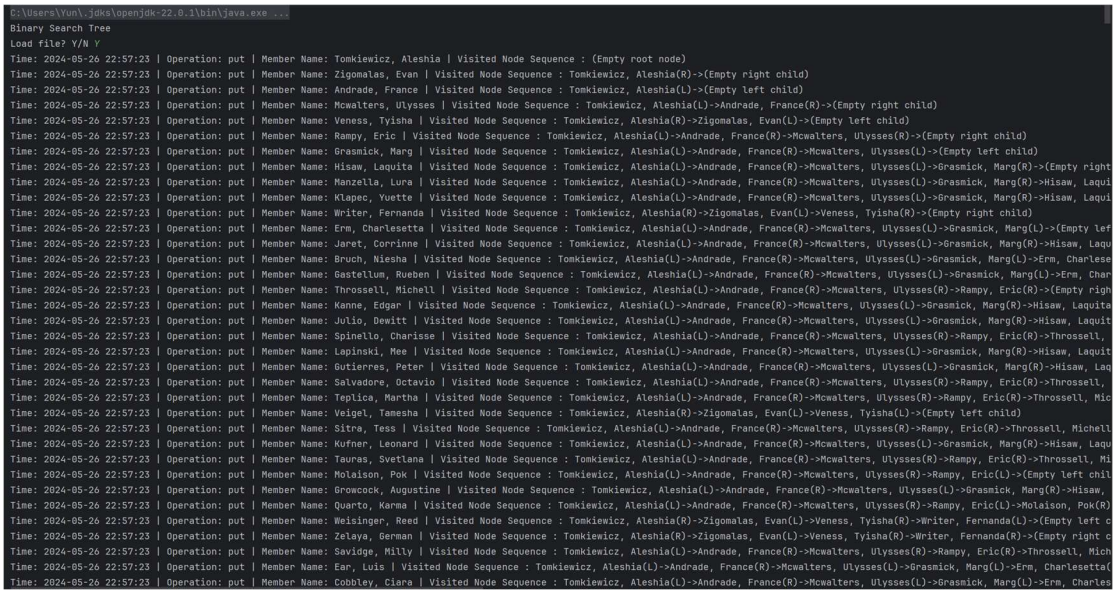
Test plan:

Step 1: Enter Y and load the sample file;

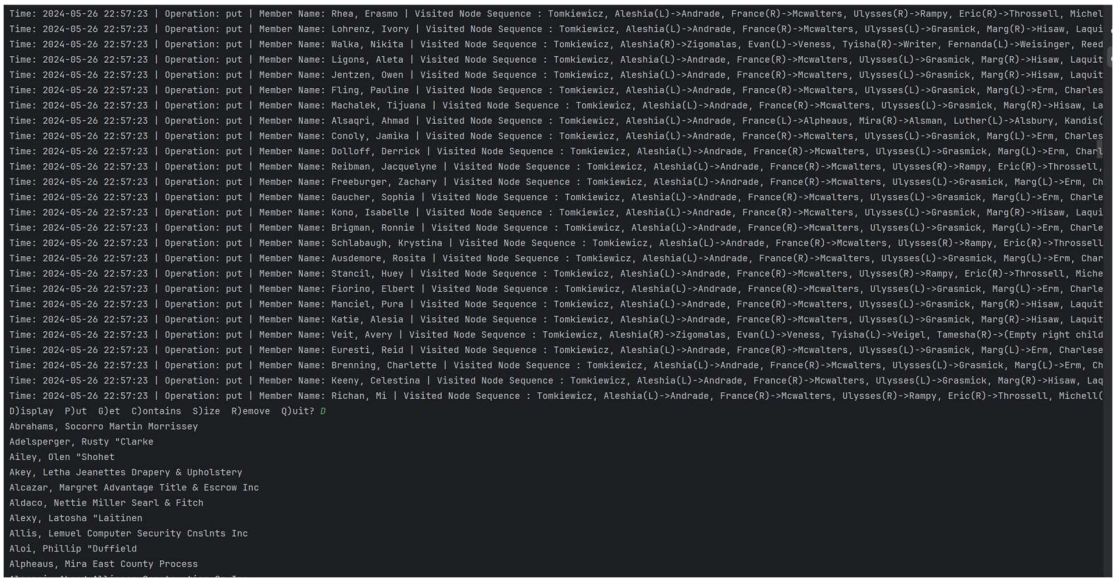
Test data: "sampleMembersUK.csv";

Expected result: log information displayed in the console and loaded into a log file ("LogInformation.txt").

Actual results: (as expected)



(Omit the middle)

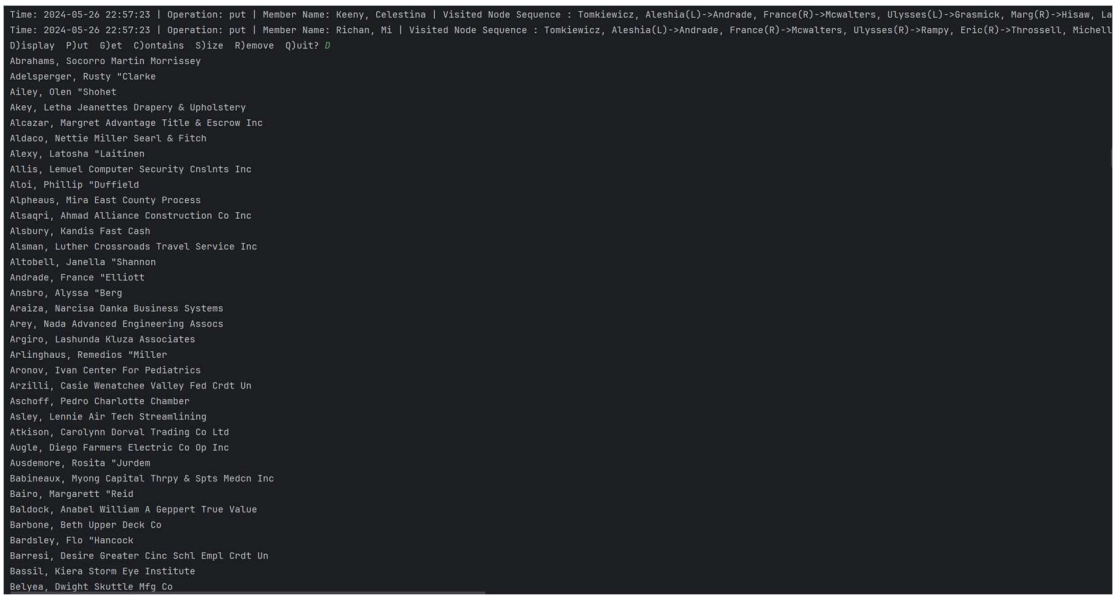


Step 2: Enter D to display the database;

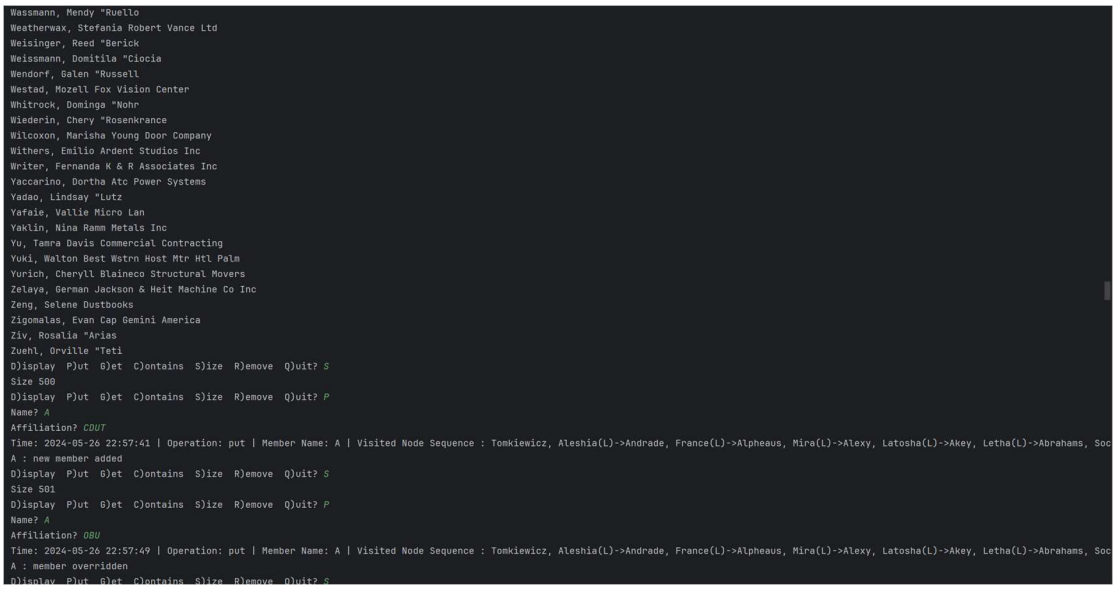
Test data: "sampleMembersUK.csv";

Expected result: Displays the database data, in alphabetical order.

Actual results: (as expected)



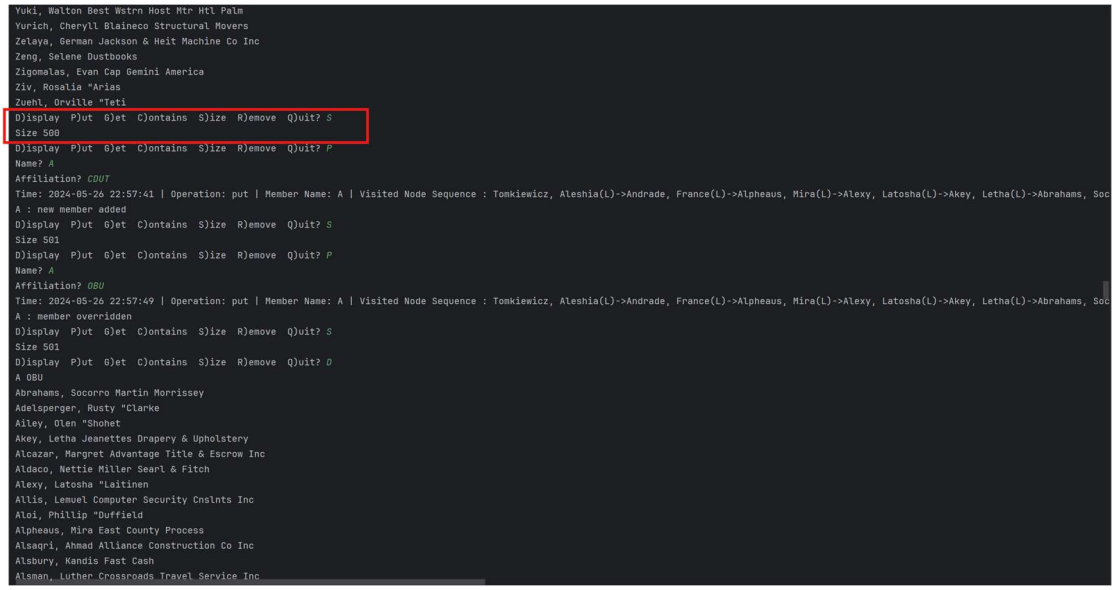
(Omit the middle)



Step 3: Input S and print the database size;

Expected result: size 500

Actual results: (as expected)



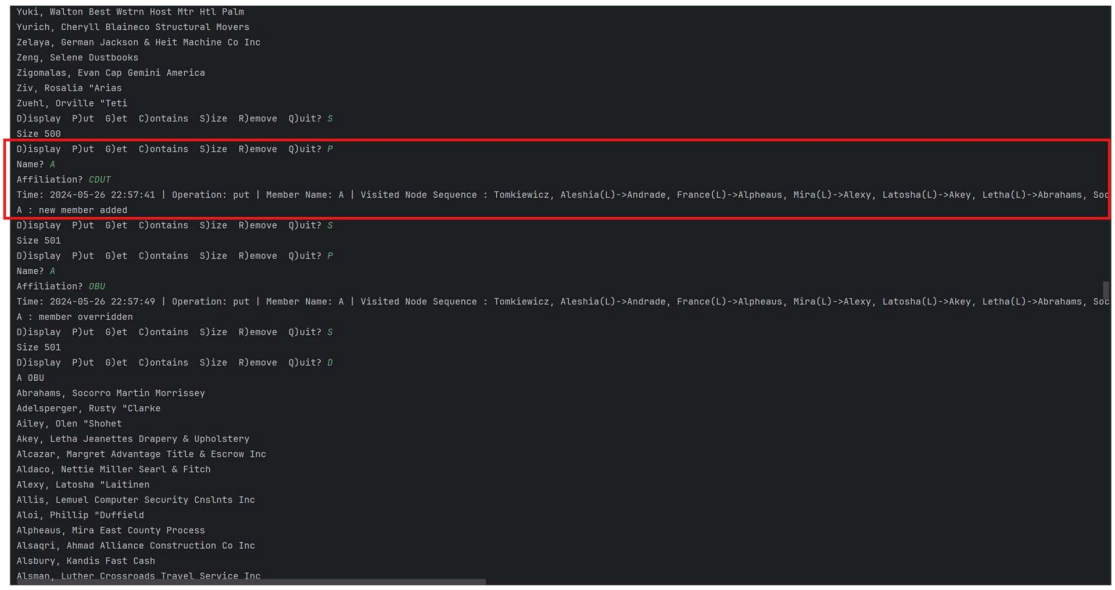
Step 4: Enter P and add new data to the database;

Test data: Name: A | Affiliation: CDUT;

Expected result: Displays "new member added". log information is displayed in

the console as well as loaded into a log file ("LogInformation.txt").

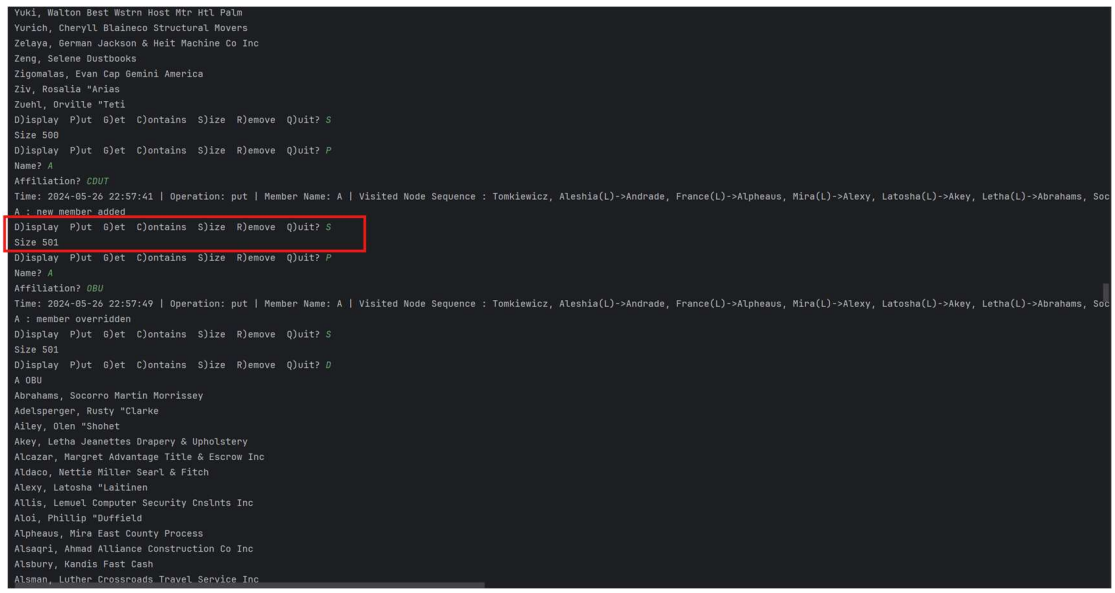
Actual results: (as expected)



Step 5: Input S and print the database size;

Expected result: size 500

Actual results: (as expected)

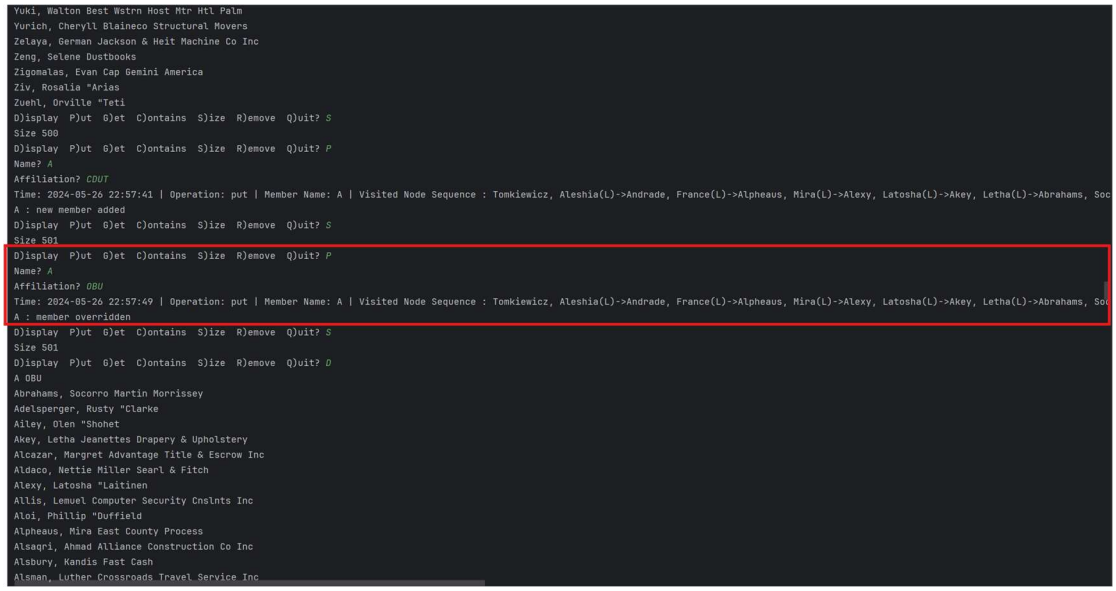


Step 6: Input P, add new data to the database;

Test data: Name: A | Affiliation: OBU;

Expected result: Displays "member overridden". log information is displayed in the console as well as loaded into a log file ("LogInformation.txt").

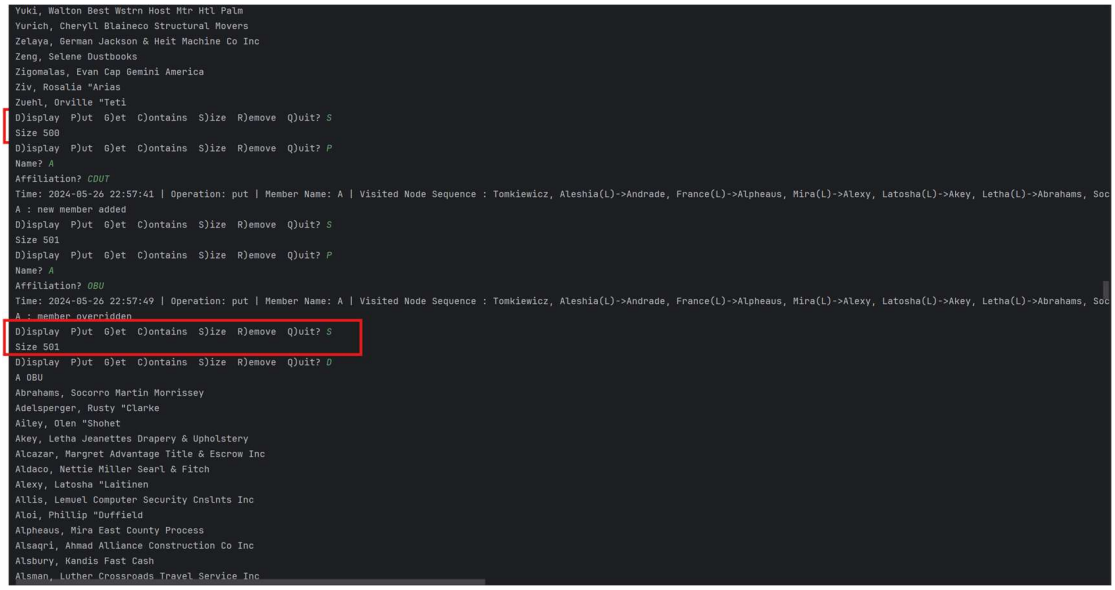
Actual results: (as expected)



Step 7: Enter S to display the database size;

Expected result: size 501

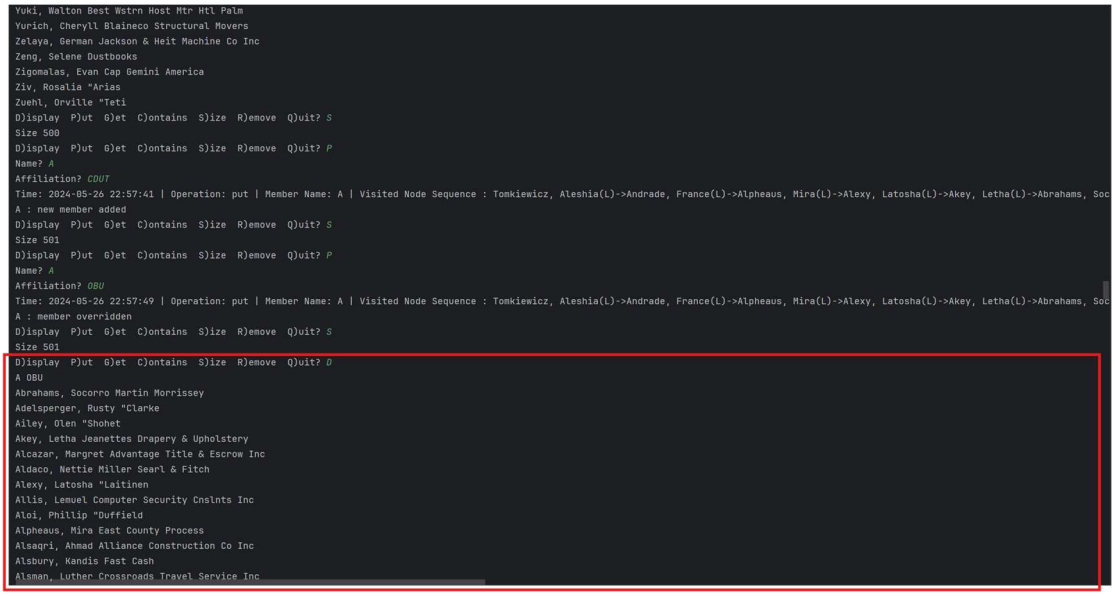
Actual results: (as expected)



Step 8: Enter D to display the database;

Expected result: "A OBU" is at the top and the rest is the same as the result of step 1.

Actual results: (as expected)



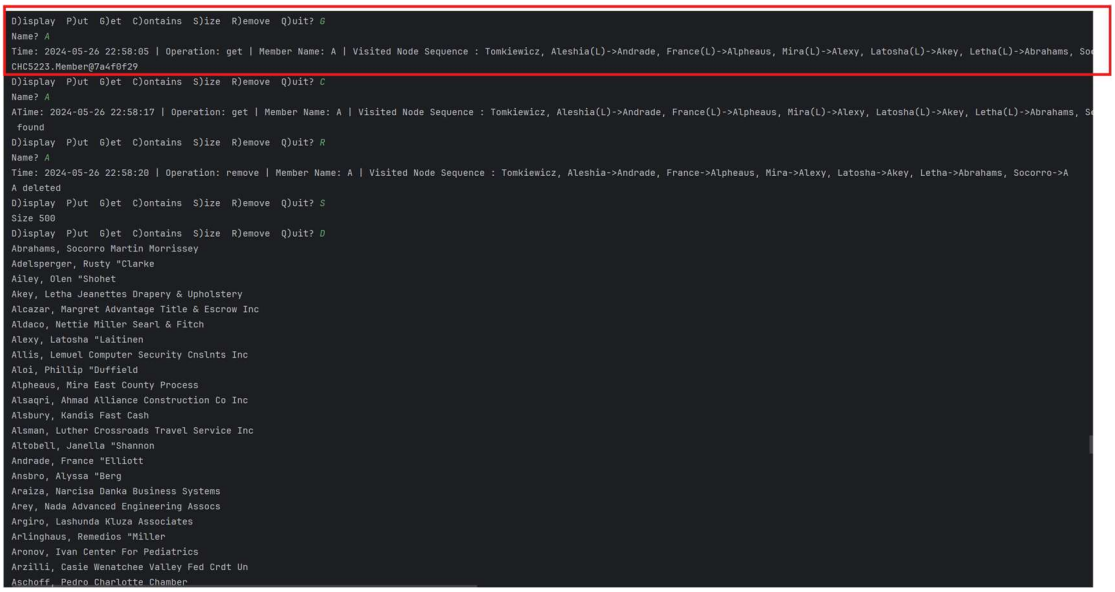
(Omit the middle)

Step 9: Input G and look for A;

Test data: "A";

Expected result: Display point A information. log information is displayed in the console as well as loaded into a log file ("LogInformation.txt").

Actual results: (as expected)

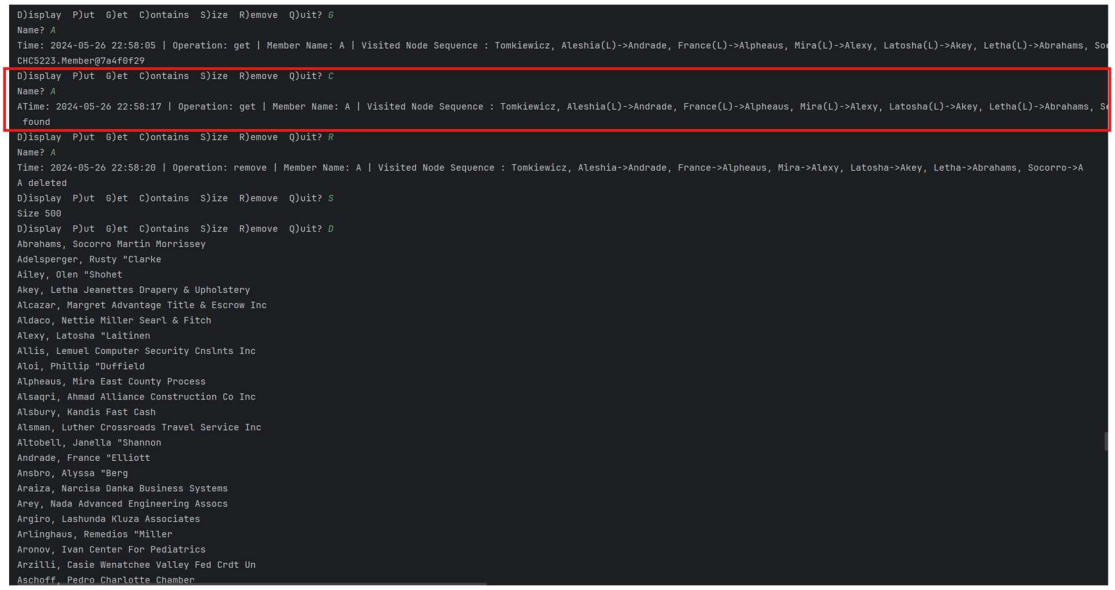


Step 10: Enter C and return whether A is included.

Test data: "A";

Expected result: "Found".

Actual results: (as expected)

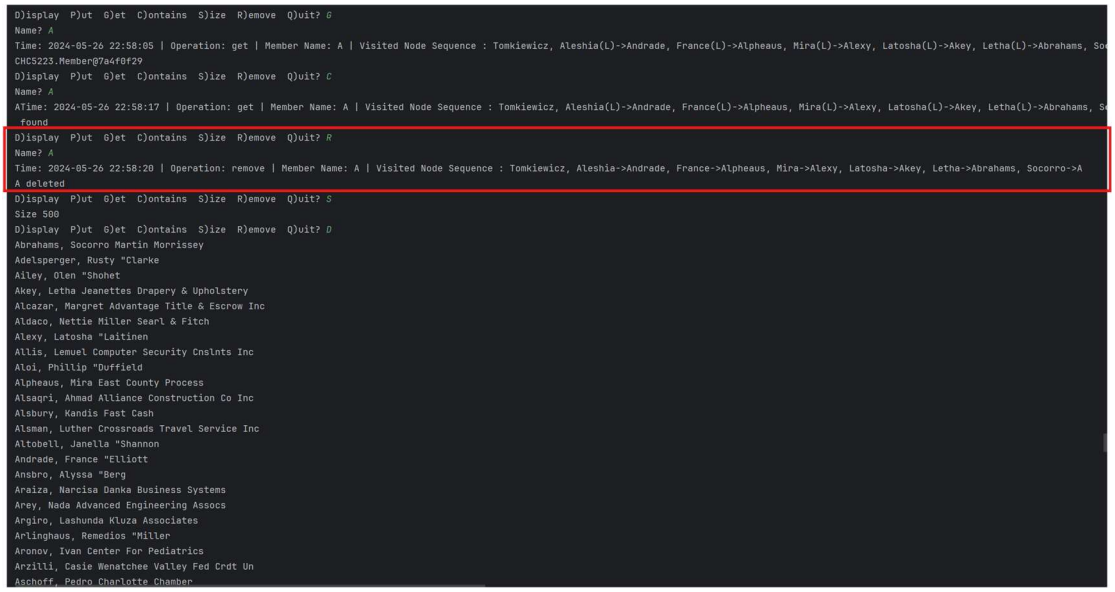


Step 11: Enter R and remove A;

Test data: "A";

Expected result: "A deleted". log information is displayed in the console as well as loaded into a log file ("LogInformation.txt").

Actual results: (as expected)



Step 12: Enter S and return the database size;

Expected result: Size 500

Actual results: (as expected)

```
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? G
Name? A
Time: 2024-05-26 22:58:05 | Operation: get | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia(L)->Andrade, France(L)->Alpheaus, Mira(L)->Alexy, Latosha(L)->Akey, Letha(L)->Abrahams, So
CHC5223.Member@7a4f0f29
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? C
Name? A
ATime: 2024-05-26 22:58:17 | Operation: get | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia(L)->Andrade, France(L)->Alpheaus, Mira(L)->Alexy, Latosha(L)->Akey, Letha(L)->Abrahams, S
 found
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? R
Name? A
Time: 2024-05-26 22:58:20 | Operation: remove | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia->Andrade, France->Alpheaus, Mira->Alexy, Latosha->Akey, Letha->Abrahams, Socorro->A
A deleted
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? S
Size 500
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? D
Abrahams, Socorro Martin Morrissey
Adelsperger, Rusty "Clarke
Ailey, Olen "Shohet
Akey, Letha Jeanettes Drapery & Upholstery
Alcazar, Margret Advantage Title & Escrow Inc
Aldaco, Nettie Miller Searl & Fitch
Alexy, Latosha "Laitinen
Allis, Lemuel Computer Security Cnslnts Inc
Aloi, Phillip "Duffield
Alpheaus, Mira East County Process
Alsaqri, Ahmad Alliance Construction Co Inc
Alsbury, Kandis Fast Cash
Alsman, Luther Crossroads Travel Service Inc
Altobell, Janella "Shannon
Andrade, France "Elliott
Ansbro, Alyssa "Berg
Araiza, Narcisa Danka Business Systems
Arey, Nada Advanced Engineering Assocs
Argiro, Lashunda Kluza Associates
Arlinghaus, Remedios "Miller
Aronov, Ivan Center For Pediatrics
Arzilli, Casie Wenatchee Valley Fed Crdt Un
Aschoff, Pedro Charlotte Chamber
```

Step 13: Enter D to display the database information;

Expected result: Same as Step 1.

Actual results: (as expected)

```
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? G
Name? A
Time: 2024-05-26 22:58:05 | Operation: get | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia(L)->Andrade, France(L)->Alpheaus, Mira(L)->Alexy, Latosha(L)->Akey, Letha(L)->Abrahams, So
CHC5223.Member@7a4f0f29
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? C
Name? A
ATime: 2024-05-26 22:58:17 | Operation: get | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia(L)->Andrade, France(L)->Alpheaus, Mira(L)->Alexy, Latosha(L)->Akey, Letha(L)->Abrahams, S
 found
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? R
Name? A
Time: 2024-05-26 22:58:20 | Operation: remove | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia->Andrade, France->Alpheaus, Mira->Alexy, Latosha->Akey, Letha->Abrahams, Socorro->A
A deleted
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? S
Size 500
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? D
Abrahams, Socorro Martin Morrissey
Adelsperger, Rusty "Clarke
Ailey, Olen "Shohet
Akey, Letha Jeanettes Drapery & Upholstery
Alcazar, Margret Advantage Title & Escrow Inc
Aldaco, Nettie Miller Searl & Fitch
Alexy, Latosha "Laitinen
Allis, Lemuel Computer Security Cnslnts Inc
Aloi, Phillip "Duffield
Alpheaus, Mira East County Process
Alsaqri, Ahmad Alliance Construction Co Inc
Alsbury, Kandis Fast Cash
Alsman, Luther Crossroads Travel Service Inc
Altobell, Janella "Shannon
Andrade, France "Elliott
Ansbro, Alyssa "Berg
Araiza, Narcisa Danka Business Systems
Arey, Nada Advanced Engineering Assocs
Argiro, Lashunda Kluza Associates
Arlinghaus, Remedios "Miller
Aronov, Ivan Center For Pediatrics
Arzilli, Casie Wenatchee Valley Fed Crdt Un
Aschoff, Pedro Charlotte Chamber
```

(Omit the middle)

```
Weatherwax, Stefania Robert Vance Ltd
Weisinger, Reed "Berick
Weissmann, Domitila "Ciocia
Wendorf, Galen "Russell
Westad, Mozell Fox Vision Center
Whitrock, Dominga "Nohr
Wiederin, Chery "Rosenkrance
Wilcoxon, Marisha Young Door Company
Withers, Emilio Ardent Studios Inc
Writer, Fernanda K & R Associates Inc
Yaccarino, Dortha Atc Power Systems
Yadao, Lindsay "Lutz
Yafaie, Vallie Micro Lan
Yaklin, Nina Ramm Metals Inc
Yu, Tamra Davis Commercial Contracting
Yuki, Walton Best Wstrn Host Mtr Htl Palm
Yurich, Cheryll Blaineco Structural Movers
Zelaya, German Jackson & Heit Machine Co Inc
Zeng, Selene Dustbooks
Zigomalas, Evan Cap Gemini America
Ziv, Rosalia "Arias
Zuehl, Orville "Teti
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? G
Name? A
Time: 2024-05-26 22:58:29 | Operation: get | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia(L)->Andrade, France(L)->Alpheaus, Mira(L)->Alexy, Latosha(L)->Akey, Letha(L)->Abrahams, Soc
A not found
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? C
Name? A
ATime: 2024-05-26 22:58:33 | Operation: get | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia(L)->Andrade, France(L)->Alpheaus, Mira(L)->Alexy, Latosha(L)->Akey, Letha(L)->Abrahams, So
 not found
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? R
Name? A
Time: 2024-05-26 22:58:45 | Operation: remove | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia->Andrade, France->Alpheaus, Mira->Alexy, Latosha->Akey, Letha->Abrahams, Socorro
A not found
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? Q

Process finished with exit code 0
```

Step 14: Input G and look for A;

Test data: "A";

Expected result: "A not found." log information is displayed in the console as well as loaded into a log file ("LogInformation.txt").

Actual results: (as expected)

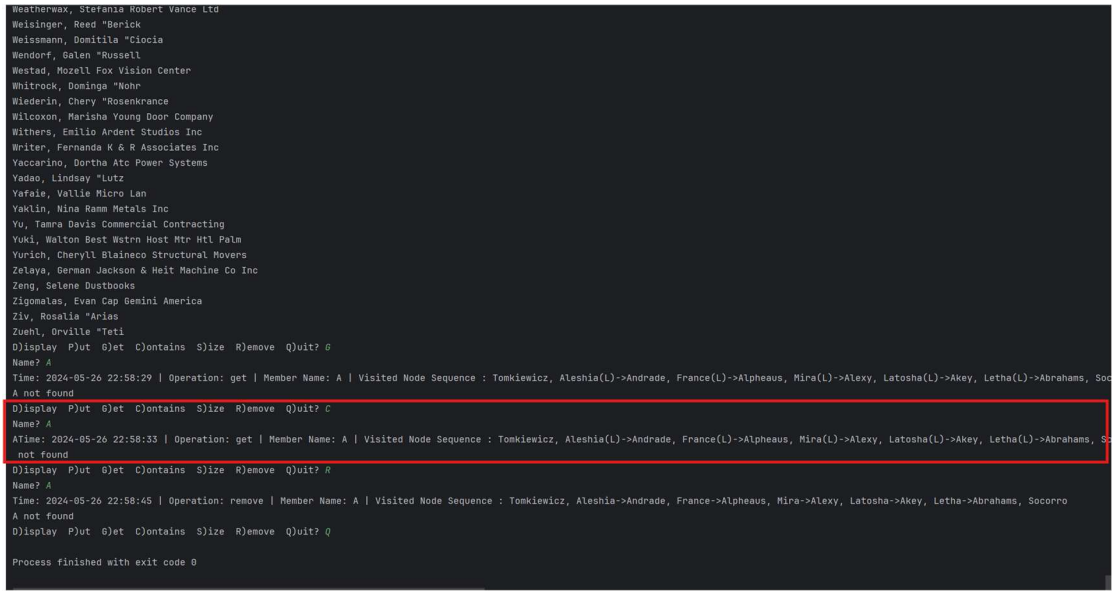
```
Weatherwax, Stefania Robert Vance Ltd
Weisinger, Reed "Berick
Weissmann, Domitila "Ciocia
Wendorf, Galen "Russell
Westad, Mozell Fox Vision Center
Whitrock, Dominga "Nohr
Wiederin, Chery "Rosenkrance
Wilcoxon, Marisha Young Door Company
Withers, Emilio Ardent Studios Inc
Writer, Fernanda K & R Associates Inc
Yaccarino, Dortha Atc Power Systems
Yadao, Lindsay "Lutz
Yafaie, Vallie Micro Lan
Yaklin, Nina Ramm Metals Inc
Yu, Tamra Davis Commercial Contracting
Yuki, Walton Best Wstrn Host Mtr Htl Palm
Yurich, Cheryll Blaineco Structural Movers
Zelaya, German Jackson & Heit Machine Co Inc
Zeng, Selene Dustbooks
Zigomalas, Evan Cap Gemini America
Ziv, Rosalia "Arias
Zuehl, Orville "Teti
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? G
Name? A
Time: 2024-05-26 22:58:29 | Operation: get | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia(L)->Andrade, France(L)->Alpheaus, Mira(L)->Alexy, Latosha(L)->Akey, Letha(L)->Abrahams, So
A not found
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? C
Name? A
ATime: 2024-05-26 22:58:33 | Operation: get | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia(L)->Andrade, France(L)->Alpheaus, Mira(L)->Alexy, Latosha(L)->Akey, Letha(L)->Abrahams, Sc
 not found
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? R
Name? A
Time: 2024-05-26 22:58:45 | Operation: remove | Member Name: A | Visited Node Sequence : Tomkiewicz, Aleshia->Andrade, France->Alpheaus, Mira->Alexy, Latosha->Akey, Letha->Abrahams, Socorro
A not found
D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove  Q)uit? Q

Process finished with exit code 0
```

Step 15: Enter C and find whether it contains A;

Test data: "A";

Expected result: "A not found." log information is displayed in the console as well as loaded into a log file ("LogInformation.txt").

Actual results: (as expected)



Step 16: Enter R and remove A;

Test data: "A";

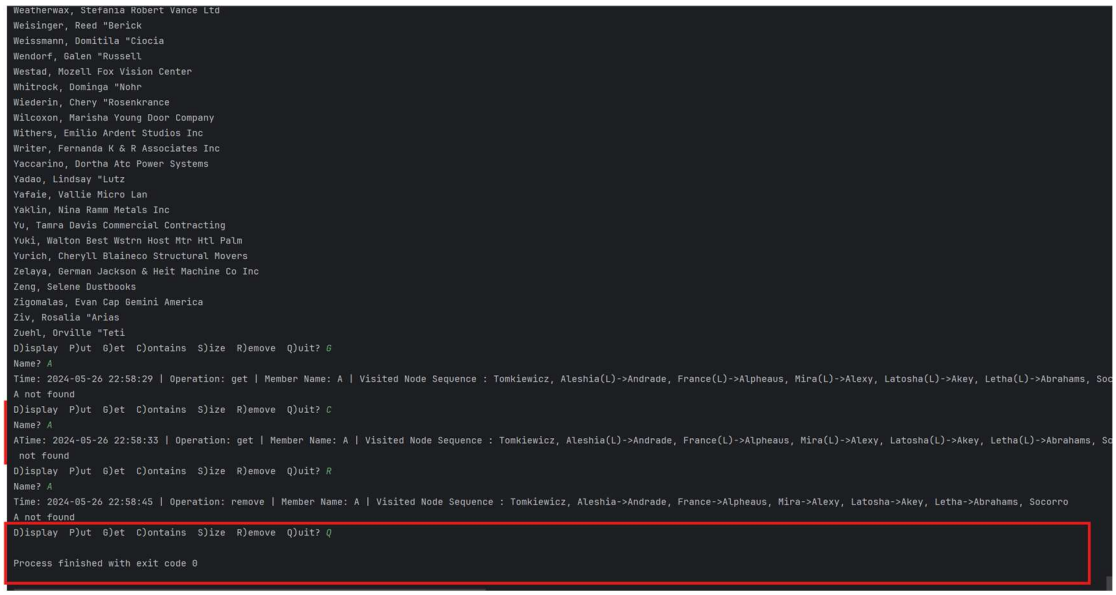
Expected result: "A not found".

Actual results: (as expected)

Step 17: Enter Q to end the program;

Expected result: Exit from the program.

Actual results: (as expected)



Attachment: "LogInformation.txt", which contains all the log information

Task_6

- Time Efficiency: The time efficiency of a binary search tree (BST) depends on its height. In the best case, the BST is perfectly balanced and its height is log(n), where n is the number of nodes. In this case, the time complexity for search, insert, and delete operations is O(log(n)). However, in the worst case, the BST becomes a linked list, its height is n, and the time complexity for these operations is O(n).
- Space Efficiency: The space efficiency of a BST is O(n), where n is the number of nodes. This is because each node in the tree occupies a constant amount of space, and there are n nodes in total.
- Rationale: The BST is a dynamic data structure that allows fast lookup, addition, and removal of items. It keeps its elements in sorted order for fast in-order traversal and other operations. However, the time efficiency can degrade to O(n) if the tree becomes unbalanced. Various types of self-balancing BSTs are available to mitigate this issue.

Source Code

Class MemberBST

```
package CHC5223;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
```

```java
// The MemberBST class that implements the IMemberDB interface
public class MemberBST implements IMemberDB {
    // A list to store the sequence of visited nodes
    private final List<String> visitedNodes;

    // Inner class Node representing a node in the binary search tree
    private class Node {
        // The data stored in the node, which is a Member object
        private Member data;
        // The left and right child nodes of the current node
        private Node left, right;

        // Constructor to initialize the node data with a Member
object
        public Node(Member data) {
            this.data = data;
            left = null;
            right = null;
        }
    }

    // The root node of the binary search tree
    private Node root;
    // The size of the binary search tree, indicating the number of
nodes
    private int size;

    // Constructor to initialize the binary search tree
    public MemberBST() {
        // Print a message to the console indicating the creation of
a binary search tree
        System.out.println("Binary Search Tree");
        // Initialize the root node as null, indicating an empty tree
        root = null;
        // Initialize the size of the tree as 0
        size = 0;
        // Initialize the list of visited nodes as an empty ArrayList
        visitedNodes = new ArrayList<>();
    }


    // Method to check if a member with the specified name exists in
the database
```

```
    @Override
    public boolean containsName(String name) {
        // Assert that the name is not null and not an empty string
        assert name != null && !name.isEmpty() : "Name cannot be null
or empty";
        // Return true if the member with the given name is found,
false otherwise
        return get(name) != null;
    }

    // Method to insert a member into the database
    @Override
    public Member put(Member member) {
        // Assert that the member and the member's name are not null
and not an empty string
        assert member != null && member.getName() != null
&& !member.getName().isEmpty() : "Member or member name cannot be
null or empty";

        // If the root is null (i.e., the tree is empty)
        if (root == null) {
            // Create a new node with the member as the data and
assign it to the root
            root = new Node(member);
            // Add a log entry indicating that the root was empty
            visitedNodes.add("(Empty root node)");
            // Increment the size of the tree
            size++;
            // Log the operation
            log("put", member.getName());
            // Clear the list of visited nodes
            visitedNodes.clear();
            // Return null as there was no previous member with the
same name
            return null;
        }

        // Initialize the current node to the root and the parent
node to null
        Node current = root;
        Node parent = null;
        // Initialize a variable to hold the result of the comparison
between the member's name and the current node's data's name
        int cmp = 0;
```

```
        // While the current node is not null
        while (current != null) {
            // Set the parent node to the current node
            parent = current;
            // Compare the member's name with the current node's
data's name
            cmp = member.getName().compareTo(current.data.getName());

            // If the member's name is less than the current node's
data's name
            if (cmp < 0) {
                // Add a log entry indicating that we're going to the
left child
                visitedNodes.add(current.data.getName() + "(L)");
                // Move to the left child
                current = current.left;
            }
            // If the member's name is greater than the current
node's data's name
            else if (cmp > 0) {
                // Add a log entry indicating that we're going to the
right child
                visitedNodes.add(current.data.getName() + "(R)");
                // Move to the right child
                current = current.right;
            }
            // If the member's name is equal to the current node's
data's name
            else {
                // Add a log entry indicating that we've found the
node
                visitedNodes.add(current.data.getName());
                // Store the current node's data in a temporary
variable
                Member oldData = current.data;
                // Replace the current node's data with the new
member
                current.data = member;
                // Log the operation
                log("put", member.getName());
                // Clear the list of visited nodes
                visitedNodes.clear();
                // Return the old data
```

```
                return oldData;
            }
        }

        // Create a new node with the member as the data
        Node newNode = new Node(member);
        // If the member's name is less than the parent node's data's
name
        if (cmp < 0) {
            // Add a log entry indicating that we're adding a left
child
            visitedNodes.add("(Empty left child)");
            // Add the new node as the left child of the parent node
            parent.left = newNode;
        }
        // If the member's name is greater than the parent node's
data's name
        else {
            // Add a log entry indicating that we're adding a right
child
            visitedNodes.add("(Empty right child)");
            // Add the new node as the right child of the parent node
            parent.right = newNode;
        }

        // Increment the size of the tree
        size++;
        // Log the operation
        log("put", member.getName());
        // Clear the list of visited nodes
        visitedNodes.clear();
        // Return null as there was no previous member with the same
name
        return null;
    }

    // Method to remove a member from the database
    @Override
    public Member remove(String name){
        // Assert that the name is not null and not an empty string
        assert name != null && !name.isEmpty() : "Name cannot be null
or empty";
        // Define the parent node, the node to be deleted, and two
auxiliary nodes
```

```
        Node parent = null, del, p = null, q = null;
        // Define the result member
        Member result;
        // Start searching from the root node
        del = root;
        // While the node to be deleted is not null and the name of
the data in the node to be deleted is not equal to the input name,
continue searching
        while (del != null && !del.data.getName().equals(name)) {
            // Update the parent node to the current node to be
deleted
            parent = del;
            // Add the name of the data in the node to be deleted to
the visitedNodes list
            visitedNodes.add(del.data.getName());
            // If the input name is less than the name of the data in
the node to be deleted, search in the left subtree
            if (name.compareTo(del.data.getName()) < 0)
                del = del.left;
                // Otherwise, search in the right subtree
            else
                del = del.right;
        }

        // If the node to be deleted is found
        if(del != null) {
            // Add the name of the data in the node to be deleted to
the visitedNodes list
            visitedNodes.add(del.data.getName());

            // If the right child of the node to be deleted is null,
p points to the left child of the node to be deleted
            if (del.right == null) {
                p = del.left;
                // Add the name of the data in the p node to the
visitedNodes list
                if (p != null) {
                    visitedNodes.add(p.data.getName());
                }
            }
            // If the left child of the right child of the node to be
deleted is null, p points to the right child of the node to be
deleted, and the left child of the node to be deleted is assigned to
the left child of p
```

```
            else if (del.right.left == null) {
                p = del.right;
                // Add the name of the data in the p node to the
visitedNodes list
                visitedNodes.add(p.data.getName());
                p.left = del.left;
            } else {
                // Otherwise, p points to the right child of the node
to be deleted, and then searches to the leftmost node
                p = del.right;
                while (p.left != null) {
                    // Add the name of the data in the p node to the
visitedNodes list
                    visitedNodes.add(p.data.getName());
                    q = p;
                    p = p.left;
                }
                // Add the name of the data in the p node to the
visitedNodes list
                visitedNodes.add(p.data.getName());
                // The left child of q points to the right child of p,
the left child of p points to the left child of the node to be
deleted, and the right child of p points to the right child of the
node to be deleted
                q.left = p.right; p.left = del.left; p.right =
del.right;
            }
            // If the node to be deleted is the root node, the root
node points to p
            if(del == root) root = p;
                // If the name of the data in the node to be deleted
is less than the name of the data in the parent node, the left child
of the parent node points to p
            else if
(del.data.getName().compareTo(parent.data.getName()) < 0)
                parent.left = p;
                // Otherwise, the right child of the parent node
points to p
            else parent.right = p;
            // The result member is the data of the node to be
deleted
            result = del.data;
            // Decrement the size of the binary search tree
            size--;
```

```
        }
        // If the node to be deleted is not found, the result member
is null
        else result = null;

        // Log the operation
        log("remove", name);
        // Clear the visitedNodes list
        visitedNodes.clear();
        // Return the result member
        return result;
    }

    // Method to get a member with the specified name
    @Override
    public Member get(String name) {
        // Assert that the name is not null and not an empty string
        assert name != null && !name.isEmpty() : "Name cannot be null
or empty";
        // Call the getNode method to find the node with the
specified name starting from the root
        Node node = getNode(name, root);
        // Log the operation
        log("get", name);
        // Clear the list of visited nodes
        visitedNodes.clear();
        // If the node is null (i.e., not found), return null;
otherwise, return the data of the node
        return node == null ? null : node.data;
    }

    // Private method to get a node with the specified name
    private Node getNode(String name, Node node) {
        // Assert that the name is not null and not an empty string
        assert name != null && !name.isEmpty() : "Name cannot be null
or empty";
        // If the node is null (i.e., we've reached a leaf node and
haven't found the specified name), return null
        if (node == null) {
            visitedNodes.add("(Empty node)");
            return null;
        }
        // Compare the specified name with the name of the data in
the node
```

```
        int cmp = name.compareTo(node.data.getName());
        // If the specified name is less than the name of the data in
the node, search in the left subtree
        if (cmp < 0) {
            visitedNodes.add(node.data.getName()+"(L)");
            return getNode(name, node.left);
        }
        // If the specified name is greater than the name of the data
in the node, search in the right subtree
        else if (cmp > 0) {
            visitedNodes.add(node.data.getName()+"(R)");
            return getNode(name, node.right);
        }
        // If the specified name is equal to the name of the data in
the node, return the node
        else {
            visitedNodes.add(node.data.getName());
            return node;
        }
    }

    // Method to get the size of the database
    @Override
    public int size() {
        // Return the size of the binary search tree
        return size;
    }

    // Method to check if the database is empty
    @Override
    public boolean isEmpty() {
        // Return true if the size of the binary search tree is 0,
false otherwise
        return size == 0;
    }

    // Method to display all members in the database
    @Override
    public void displayDB() {
        // Call the private displayDB method with the root node as
the argument
        displayDB(root);
    }
```

```
    // Private recursive method to display all nodes in the binary
search tree
    private void displayDB(Node node) {
        // If the node is not null
        if (node != null) {
            // Recursively call the method with the left child of the
current node
            displayDB(node.left);
            // Print the name and affiliation of the member in the
current node
            System.out.println(node.data.getName() + " " +
node.data.getAffiliation());
            // Recursively call the method with the right child of
the current node
            displayDB(node.right);
        }
    }

    // Method to clear the database
    @Override
    public void clearDB() {
        // Set the root node to null
        root = null;
        // Set the size of the binary search tree to 0
        size = 0;
    }

    // Define a private method for logging operations
    private void log(String operation, String name) {
        // Specify the log file name
        String logFile = "LogInformation.txt";
        // Use try-with-resources to ensure the BufferedWriter is
closed at the end
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(logFile, true))) {
            // Get the current date and time
            LocalDateTime now = LocalDateTime.now();
            // Create a DateTimeFormatter
            DateTimeFormatter formatter =
DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
            // Format the current date and time as a string
            String nowString = now.format(formatter);

            // Construct the log message with the current date and
```

```
time, operation, member name, and visited node sequence
            String logMessage = "Time: "+ nowString + " | Operation:
" + operation + " | Member Name: " + name + " | Visited Node
Sequence : ";
            // Write the log message to the log file
            writer.write(logMessage);
            // Call the private method to write the node sequence to
the log file
            writeNodeSequence(writer, visitedNodes);
            // Write a new line to the log file
            writer.newLine();
            // Print the log message to the console
            System.out.println(logMessage + String.join("->",
visitedNodes));
        } catch (IOException e) {
            // Print the stack trace for any IOException
            e.printStackTrace();
        }
    }

    // Define a private method for writing the node sequence to the
log file
    private void writeNodeSequence(BufferedWriter writer,List<String>
visitedNodes) throws IOException {
        // For each visited node
        for (String visitedNode : visitedNodes) {
            // Write the visited node to the log file, prefixed with
"->"
            writer.write("->" + visitedNode);
        }
    }
}
```

Class MemberBSTTest

```
package CHC5223;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class MemberBSTTest {
    private MemberBST memberBST;
```

```
    @BeforeEach
    public void setUp() {
        memberBST = new MemberBST();
    }

    @Test
    public void testDeleteLeafNode() {
        Member member1 = new Member("A", "aaaaa");
        Member member2 = new Member("B", "bbbbb");
        memberBST.put(member1);
        memberBST.put(member2);
        memberBST.remove("B");
        assertNull(memberBST.get("B"));
    }

    @Test
    public void testDeleteNodeWithOneDescendant() {
        Member member1 = new Member("A", "aaaaa");
        Member member2 = new Member("B", "bbbbb");
        Member member3 = new Member("C", "ccccc");
        memberBST.put(member1);
        memberBST.put(member2);
        memberBST.put(member3);
        memberBST.remove("B");
        assertNull(memberBST.get("B"));
        assertNotNull(memberBST.get("C"));
    }

    @Test
    public void testDeleteNodeWithTwoDescendants() {
        Member member1 = new Member("B", "bbbbb");
        Member member2 = new Member("A", "aaaaa");
        Member member3 = new Member("C", "ccccc");
        memberBST.put(member1);
        memberBST.put(member2);
        memberBST.put(member3);
        memberBST.remove("B");
        assertNull(memberBST.get("B"));
        assertNotNull(memberBST.get("A"));
        assertNotNull(memberBST.get("C"));
    }
}
```

Class CHC5223

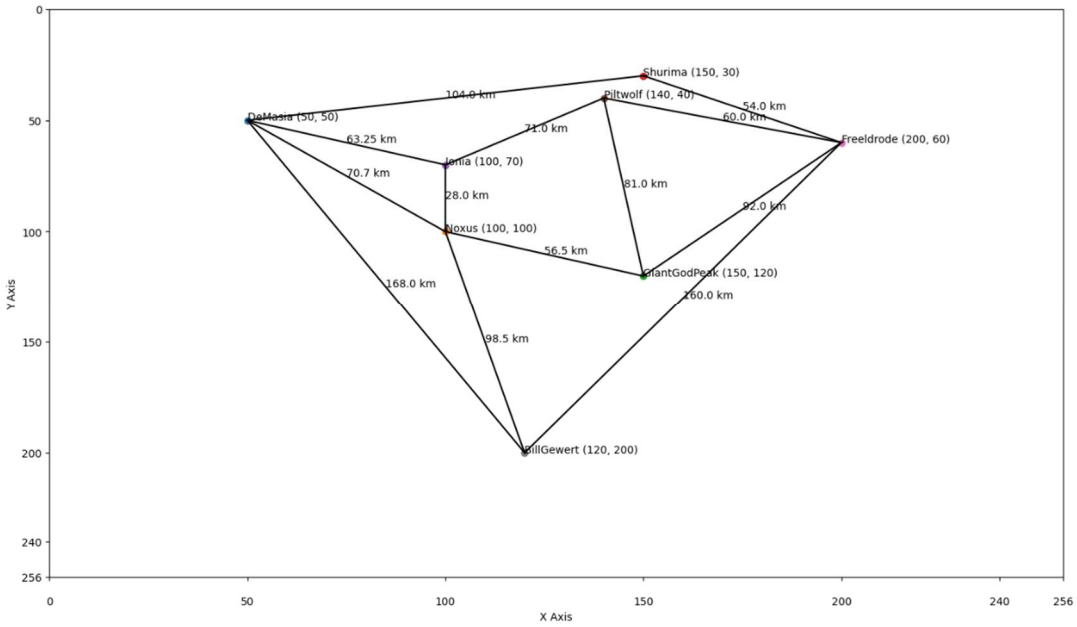
```
package CHC5223;

import java.util.*;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
/**
 * @author
 */

public class CHC5223 {
static Scanner kb;
static IMemberDB db = new MemberBST();

// simply change this to
// static IMemberDB db = new MemberBST()
// for Assignment 2

static String fileName = "sampleMembersUK.csv";

private static String readNonEmpty() {
    // read non-null, non-empty string;
    String s = kb.nextLine().trim();
    while (s == null || s.equals("")) {
        System.out.print("must not be blank -- try again: ");
        s = kb.nextLine().trim();
    }
    assert s != null && !s.trim().equals("");
  return s;
}

public static void main(String[] args) {
    String option, load, name, affiliation;
    Member resp;
    kb = new Scanner(System.in);
    System.out.print("Load file? Y/N ");
    load = readNonEmpty();
    if(load.charAt(0) == 'Y' || load.charAt(0) == 'y')
        loadFile();
    System.out.print("D)isplay  P)ut  G)et  C)ontains  S)ize  R)emove
Q)uit? ");
    option = readNonEmpty();
    while(option.charAt(0)!= 'Q' && option.charAt(0)!= 'q') {
```

```
        switch (option.charAt(0)) {
            case 'D': case'd':  // display
              db.displayDB();
              break;

            case 'P': case'p':  // put
              System.out.print("Name? "); name = readNonEmpty();
              System.out.print("Affiliation? "); affiliation =
readNonEmpty();
              Member member = new Member(name, affiliation);
              resp = db.put(member);
              System.out.print(name);
              if(resp == null) System.out.println(" : new member
added");
              else System.out.println(" : member overridden");
              break;

            case 'S': case's':  //size
              System.out.println("Size "+ db.size());
              break;

            case 'G': case'g':  // get
              System.out.print("Name? "); name = readNonEmpty();
              resp = db.get(name);
              if(resp != null) System.out.println(resp.toString());
              else System.out.println(name + " not found");
              break;

            case 'C': case'c':  // contains
              System.out.print("Name? "); name = readNonEmpty();
              System.out.print(name);
              if(db.containsName(name)) System.out.println(" found");
              else System.out.println(" not found");
              break;

            case 'R': case'r':  // remove
              System.out.print("Name? "); name = readNonEmpty();
              resp = db.remove(name);
              System.out.print(name);
              if(resp != null) System.out.println(" deleted");
              else System.out.println(" not found");
              break;

            default: //?
```

```
                System.out.println("unknown option");

            } // switch
            System.out.print("D)isplay  P)ut  G)et  C)ontains  S)ize
R)emove  Q)uit? ");
            option = readNonEmpty();
        } // while
}

    private static void loadFile() {
            String cvsSplitBy = ",";
            Scanner file;
            Member member;
            try {
              FileInputStream streamIn = new FileInputStream(fileName);
              file = new Scanner(streamIn);
              while (file.hasNextLine()) {
                String line = file.nextLine();
                String[] parts = line.split(cvsSplitBy);
                String surname = parts[0].trim();
                String firstNames = parts[1].trim();
                String affiliation = parts[2].trim();
                member = new Member(surname + ", " + firstNames,
affiliation);
                db.put(member);
              }
              file.close();
            } catch (FileNotFoundException ex) {
                System.out.println("Can't find file " + fileName);
            }
    }
}
```

二、 Part B – graphs and pathfinding

Task_1



The python code to create the image is as follows:

```
import matplotlib.pyplot as plt

# Create a dictionary for the stations
stations = {
    "DeMasia": (50, 50),
    "Noxus": (100, 100),
    "GiantGodPeak": (150, 120),
    "Shurima": (150, 30),
    "Ionia": (100, 70),
    "Piltwolf": (140, 40),
    "Freeldrode": (200, 60),
    "BillGewert": (120, 200)
}

# Create a list of tuples for the links
links = [
    ("DeMasia", "Noxus", 70.70),
    ("DeMasia", "Ionia", 63.25),
    ("DeMasia", "BillGewert", 168.00),
    ("DeMasia", "Shurima", 104.00),
    ("Noxus", "GiantGodPeak", 56.50),
    ("Noxus", "BillGewert", 98.50),
```

```
    ("Noxus", "Ionia", 28.00),
    ("GiantGodPeak", "Piltwolf", 81.00),
    ("GiantGodPeak", "Freeldrode", 92.00),
    ("Shurima", "Freeldrode", 54.00),
    ("Ionia", "Piltwolf", 71.00),
    ("Piltwolf", "Freeldrode", 60.00),
    ("Freeldrode", "BillGewert", 160.00)
]

# Plot the stations
for station, (x, y) in stations.items():
    plt.scatter(x, y)
    plt.text(x, y, f'{station} ({x}, {y})')  # Add coordinates to the
station label

# Plot the links
for station1, station2, weight in links:
    x1, y1 = stations[station1]
    x2, y2 = stations[station2]
    plt.plot([x1, x2], [y1, y2], 'k-')
    plt.text((x1 + x2) / 2, (y1 + y2) / 2, f'{weight} km')

# Set the axis limits and invert the y-axis
plt.axis([0, 256, 0, 256])
plt.gca().invert_yaxis()

# Add x-axis and y-axis
plt.axhline(0, color='black')
plt.axvline(0, color='black')

# Set the x-axis and y-axis ticks
plt.xticks([0, 50,100,150,200,240,256], ['0',
'50','100','150','200','240','256'])
plt.yticks([0, 50,100,150,200,240,256], ['0',
'50','100','150','200','240','256'])

# Set the axis labels
plt.xlabel('X Axis', position=(100, 1.05))
plt.ylabel('Y Axis')

# Show the grid
plt.grid(False)

# Adjust the tick parameters
```

```
plt.tick_params(axis='x', which='major', labelsize=10, colors='black',
pad=15)

# Show the plot
plt.show()
```

Task_2

"station" DeMasia 50 50

"station" Noxus 100 100

"station" GiantGodPeak 150 120

"station" Shurima 150 30

"station" Ionia 100 70

"station" Piltwolf 140 40

"station" Freeldrode 200 60

"station" BillGewert 120 200

"link" DeMasia Noxus 70.70

"link" DeMasia Ionia 63.25

"link" DeMasia BillGewert 168.00

"link" DeMasia Shurima 104.00

"link" Noxus GiantGodPeak 56.50

"link" Noxus BillGewert 98.50

"link" Noxus Ionia 28.00

"link" GiantGodPeak Piltwolf 81.00

"link" GiantGodPeak Freeldrode 92.00

"link" Shurima Freeldrode 54.00

"link" Ionia Piltwolf 71.00

"link" Piltwolf Freeldrode 60.00

"link" Freeldrode BillGewert 160.00

Task_3

StackInt

```
/**
 * StackInt is a concrete implementation of the abstract class
AbsStackInt.
 * It represents a stack of integers with a fixed capacity.
 * The stack is implemented using an array.
 */
public class StackInt extends AbsStackInt {

    /**
     * Constructs a new StackInt with the specified capacity.
     *
     * @param capacity the maximum number of elements the stack can
hold
     */
    public StackInt(int capacity) {
        super(capacity);
    }

    /**
     * Pushes an integer onto the top of the stack.
     * If the stack is full, an AssertionError is thrown.
     *
     * @param n the integer to be pushed onto the stack
     */
    @Override
    public void push(int n) {
        assert getSize() != getCapacity() : "Stack is full";
        stack[size++] = n;
    }
```

```
    /**
     * Removes and returns the integer at the top of the stack.
     * If the stack is empty, an AssertionError is thrown.
     *
     * @return the integer at the top of the stack
     */
    @Override
    public int pop() {
        assert getSize() != 0 : "Stack is empty";
        return stack[--size];
    }

    /**
     * Returns the integer at the top of the stack without removing
it.
     * If the stack is empty, an AssertionError is thrown.
     *
     * @return the integer at the top of the stack
     */
    @Override
    public int peek() {
        assert getSize() != 0 : "Stack is empty";
        return stack[size - 1];
    }
}
```

The "StackInt" class extends the abstract class "AbsStackInt". It represents a stack data structure that stores elements and fits the LIFO order. The constructor takes an integer parameter to set the capacity of the stack.
The "push" method adds an element to the top of the stack, while the "pop" method removes and returns the element at the top. The "peek" method returns the element at the top of the stack. All methods include asserting statements that ensure the stack is not full (push method) or empty (pop and peek methods) before performing their operations.

QueueInt

```
/**
 * QueueInt is a concrete implementation of the abstract class
AbsQueueInt.
 * It represents a queue of integers with a fixed capacity.
 * The queue is implemented using an array.
 */
```

```java
public class QueueInt extends AbsQueueInt {

    /**
     * Constructs a new QueueInt with the specified capacity.
     *
     * @param capacity the maximum number of elements the queue can
hold
     */
    public QueueInt(int capacity) {
        super(capacity);
    }

    /**
     * Adds an integer to the back of the queue.
     * If the queue is full, an AssertionError is thrown.
     *
     * @param n the integer to be added to the back of the queue
     */
    @Override
    public void addToBack(int n) {
        assert getSize() != getCapacity() : "Queue is full";
        queue[size++] = n;
    }

    /**
     * Removes and returns the integer from the front of the queue.
     * If the queue is empty, an AssertionError is thrown.
     *
     * @return the integer from the front of the queue
     */
    @Override
    public int removefromFront() {
        assert getSize() != 0 : "Queue is empty";
        int removed = queue[0];
        for (int i = 0; i < getSize() - 1; i++) {
            queue[i] = queue[i + 1];
        }
        size--;
        return removed;
    }
}
```

The class "QueueInt" extends an abstract class "AbsQueueInt", which satisfies the FIFO order. It has a constructor that takes an integer "capacity" as an argument and

calls the constructor of the superclass "AbsQueueInt".
The "addToBack" method adds an integer "n" to the back of the queue, as long as the queue is not full (i.e., use the assert statement to ensure the size of the queue is less than the capacity).
The "removeFromFront" method removes and returns the element at the front of the queue, as long as the queue is not empty (i.e., use assertion to ensure the size of the queue is greater than 0). First, the element to be removed is set to 0. The queue then loops through the last element in turn, is assigned to the one before it, clears the last element to reduce its size, and returns the result "del".

ListInt

```
/**
 * ListInt is a concrete implementation of the abstract class
AbsListInt.
 * It represents a list of integers with a fixed capacity.
 * The list is implemented using an array.
 */
public class ListInt extends AbsListInt {

    /**
     * Constructs a new ListInt with the specified capacity.
     *
     * @param capacity the maximum number of elements the list can
hold
     */
    public ListInt(int capacity) {
        super(capacity);
    }

    /**
     * Appends an integer to the end of the list.
     * If the list is full, an AssertionError is thrown.
     *
     * @param n the integer to be appended to the list
     */
    @Override
    public void append(int n) {
        assert getSize() != getCapacity() : "List is full";
        list[size] = n;
        size++;
    }

    /**
```

```
     * Checks if the list contains a specific integer.
     * If the list is empty, an AssertionError is thrown.
     *
     * @param x the integer to be checked
     * @return true if the list contains the integer, false otherwise
     */
    @Override
    public boolean contains(int x) {
        assert getSize() != 0 : "List is empty";
        for (int i = 0; i < getSize(); i++) {
            if (list[i] == x) {
                return true;
            }
        }
        return false;
    }
}
```

The "ListInt" class extends a superclass called "AbsListInt" and overrides its methods.

- ListInt(int capacity): This is the constructor of the ListInt class. It initializes a new ListInt object with a specified capacity.
- append(int n): This method appends an integer n to the end of the list. If the list is already full, it throws an AssertionError.
- contains(int x): This method checks if the list contains a specific integer x. It returns true if the integer is found in the list, and false otherwise. If the list is empty, it throws an AssertionError.

SetInt

```
public class SetInt extends AbsSetInt {

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

    /**
     * @param "x" -- value to be sought
     * @pre true
     * @return true iff x is in list*/
    @Override
    public boolean contains(int n) {
        if (getSize() == 0) {
            return false;
```

```
        }
        for (int i = 0; i < getSize(); i++) {
            if (set[i] == n) {
                return true;
            }
        }
        return false;
    }

    /**
     * @param n node to be added
     * @pre contains(n) || getSize() != getCapacity()
     * @post contains(n)
     */
    @Override
    public void include(int n) {
        assert !contains(n) : "Element already exists in the set";
        assert getSize() != getCapacity() : "Set is full";
        set[size++] = n;
    }

    /**
     * @pre true
     * @post !contains(n)
     */
    @Override
    public void exclude(int n) {
        assert contains(n) : "Element does not exist in the set";
        for (int i = 0; i < getSize(); i++) {
            if (set[i] == n) {
                for (int j = i; j < getSize() - 1; j++) {
                    set[j] = set[j + 1];
                }
                size--;
                break;
            }
        }
    }
}
```

Class 'SetInt' which implements' AbsSetInt '. This class represents a collection of integers with a fixed capacity. Collections are implemented using arrays.
Here's a detailed explanation:
1. Constructor 'SetInt(int capacity)' : This is the constructor function of the 'SetInt' class that takes a single parameter 'capacity', which represents the maximum capacity of the collection. This constructor initializes the collection by calling the constructor of the parent class 'AbsSetInt'.

2. The 'contains(int n)' method: This method is used to check if the collection contains a specific integer 'n'. If the collection is empty, then we simply return 'false'. Otherwise, iterate over each element in the collection and if an element equal to 'n' is found, then 'true' is returned. If no element equal to 'n' is found after iterating through all elements, then 'false' is returned.

3. 'include(int n)' method: This method is used to add an integer 'n' to the collection. Before adding it, we check if 'n' is already in the collection, and if it is, we throw an 'AssertionError'. Then, we check if the collection is full, and if it is, we throw an 'AssertionError' as well. If there is no 'n' in the set, and the set is not yet full, then add 'n' to the set.

4. 'exclude(int n)' method: This method is used to remove an integer 'n' from the collection. Before removing, we check if the collection contains' n ', and if it doesn't, we throw an 'AssertionError'. If the set contains' n ', then iterate over each element in the set, find the element equal to 'n', and shift all elements after it one bit forward, then decrement the size of the set by one. Tip: Items in a collection are unique and can't have duplicates, so you need to check to see if an item already exists when adding an item and check to see if an item already exists when removing an item.

The idea of this class is to use an array to store the elements of a collection, and then use various methods to operate on the array to implement various operations on the collection. The design principle of this class is that the elements in the collection are unique and there can be no duplicate elements, so you need to check if an element already exists when adding an element and check if an element exists when removing an element.

Task_4

StackInt

In the StackInt class, I use assertions to ensure that the stack is not full before pushing an element and not empty before popping or peeking an element.

```
24      @Override  13 usages
        public void push(int n) {
25          assert getSize() != getCapacity() : "Stack is full";
26          stack[size++] = n;
27      }
28
29      /**
30       * Removes and returns the integer at the top of the stack.
31       * If the stack is empty, an AssertionError is thrown.
32       *
33       * @return the integer at the top of the stack
34       */
35      @Override  7 usages
36      public int pop() {
37          assert getSize() != 0 : "Stack is empty";
38          return stack[--size];
39      }
40
41      /**
42       * Returns the integer at the top of the stack without removing it.
43       * If the stack is empty, an AssertionError is thrown.
44       *
45       * @return the integer at the top of the stack
46       */
47      @Override  5 usages
48      public int peek() {
49          assert getSize() != 0 : "Stack is empty";
50          return stack[size - 1];
51      }
```

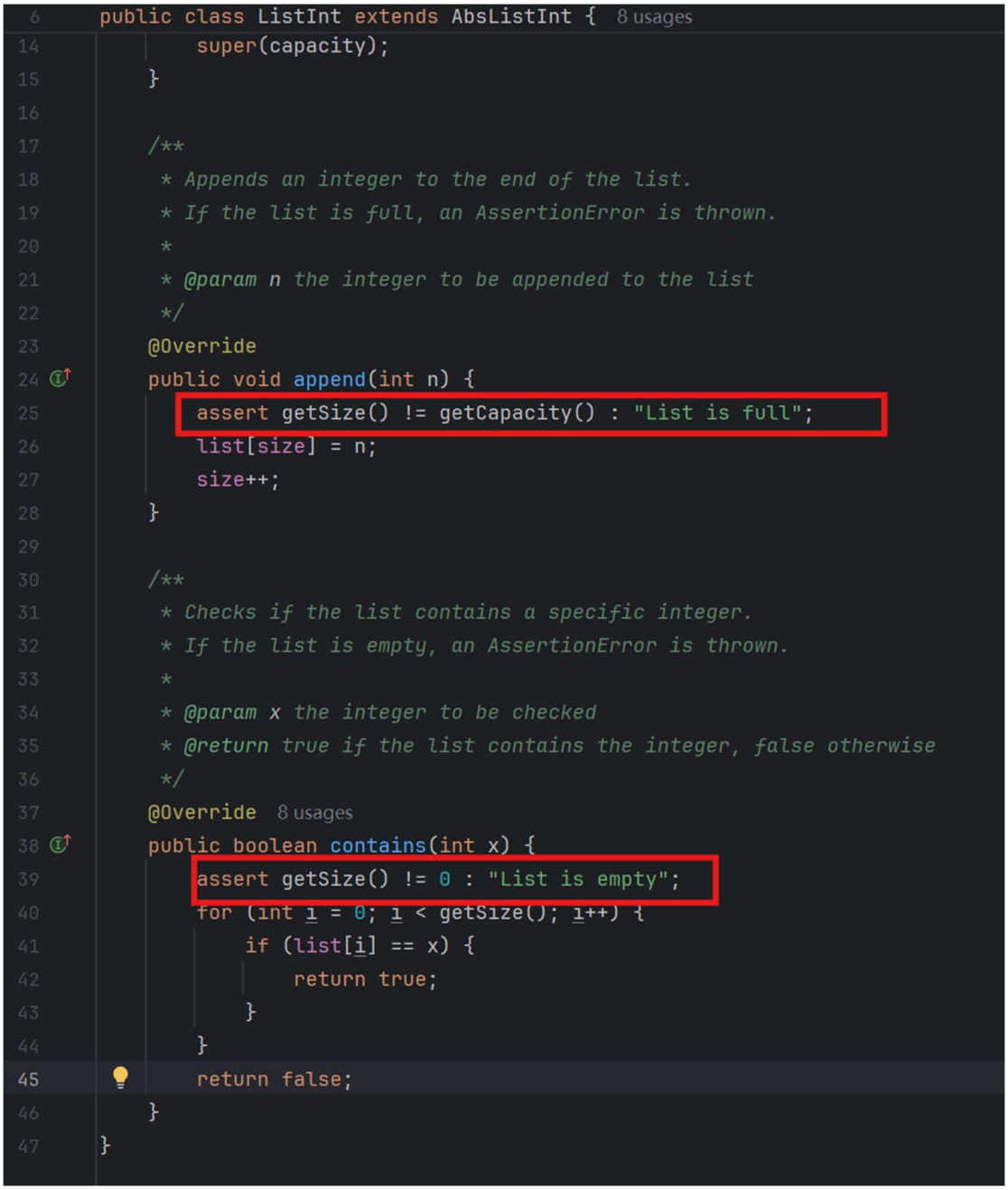
QueueInt

In the QueueInt class, I use assertions to ensure that the queue is not full before adding an element and not empty before removing an element.

```java
public class QueueInt extends AbsQueueInt {  4 usages
     * @param capacity the maximum number of elements the queue can hold
     */
    public QueueInt(int capacity) {  2 usages
        super(capacity);
    }

    /**
     * Adds an integer to the back of the queue.
     * If the queue is full, an AssertionError is thrown.
     *
     * @param n the integer to be added to the back of the queue
     */
    @Override  6 usages
    public void addToBack(int n) {
        assert getSize() != getCapacity() : "Queue is full";
        queue[size++] = n;
    }

    /**
     * Removes and returns the integer from the front of the queue.
     * If the queue is empty, an AssertionError is thrown.
     *
     * @return the integer from the front of the queue
     */
    @Override  3 usages
    public int removefromFront() {
        assert getSize() != 0 : "Queue is empty";
        int removed = queue[0];
        for (int i = 0; i < getSize() - 1; i++) {
            queue[i] = queue[i + 1];
        }
        size--;
        return removed;
    }
}
```

ListInt

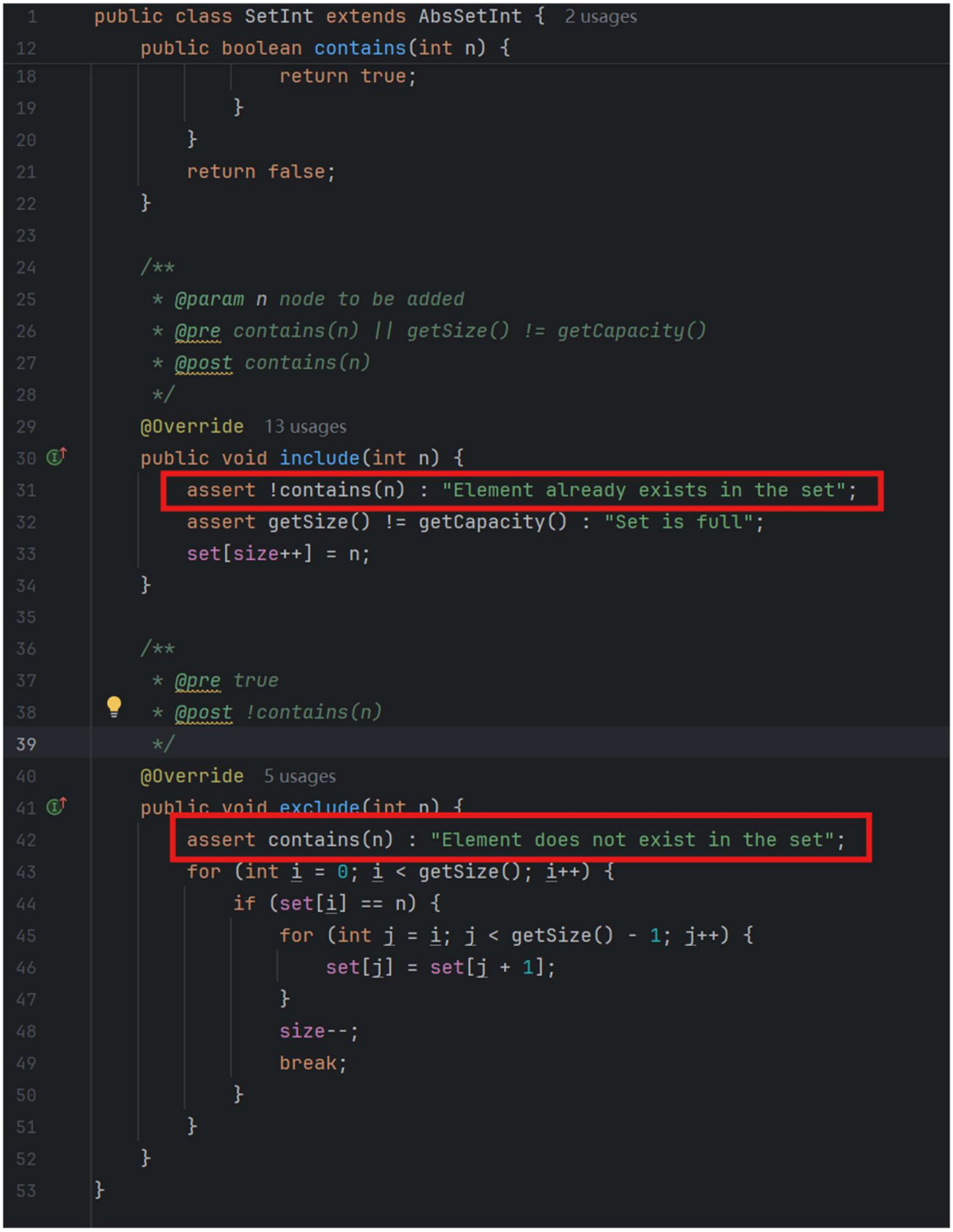
In the ListInt class, I use assertions to ensure that the list is not full before appending an element and not empty before checking if an element exists.



```
6   public class ListInt extends AbsListInt {  8 usages
14          super(capacity);
15      }
16
17      /**
18       * Appends an integer to the end of the list.
19       * If the list is full, an AssertionError is thrown.
20       *
21       * @param n the integer to be appended to the list
22       */
23      @Override
24      public void append(int n) {
25          assert getSize() != getCapacity() : "List is full";
26          list[size] = n;
27          size++;
28      }
29
30      /**
31       * Checks if the list contains a specific integer.
32       * If the list is empty, an AssertionError is thrown.
33       *
34       * @param x the integer to be checked
35       * @return true if the list contains the integer, false otherwise
36       */
37      @Override  8 usages
38      public boolean contains(int x) {
39          assert getSize() != 0 : "List is empty";
40          for (int i = 0; i < getSize(); i++) {
41              if (list[i] == x) {
42                  return true;
43              }
44          }
45          return false;
46      }
47  }
```

SetInt

In the SetInt class, I use assertions to ensure that the set does not already contain an element before including it and that it does contain an element before excluding it.



```
1   public class SetInt extends AbsSetInt {  2 usages
12      public boolean contains(int n) {
18                  return true;
19              }
20          }
21          return false;
22      }
23
24      /**
25       * @param n node to be added
26       * @pre contains(n) || getSize() != getCapacity()
27       * @post contains(n)
28       */
29      @Override  13 usages
30      public void include(int n) {
31          assert !contains(n) : "Element already exists in the set";
32          assert getSize() != getCapacity() : "Set is full";
33          set[size++] = n;
34      }
35
36      /**
37       * @pre true
38       * @post !contains(n)
39       */
40      @Override  5 usages
41      public void exclude(int n) {
42          assert contains(n) : "Element does not exist in the set";
43          for (int i = 0; i < getSize(); i++) {
44              if (set[i] == n) {
45                  for (int j = i; j < getSize() - 1; j++) {
46                      set[j] = set[j + 1];
47                  }
48                  size--;
49                  break;
50              }
51          }
52      }
53  }
```

Task_5

To test the implementation of StackInt, QueueInt, ListInt, and SetInt, I used JUnit.

Here's the overall test plan:

- Test Plan: For each class, we will create a series of tests that cover all the methods in the class. We will test both the normal operation of the methods (with valid inputs) and their error handling (with invalid inputs).
- Test Data: We will create objects of each class with a capacity of 10. We will use a variety of integers for the test data.
- Expected Results: For each test, we will determine the expected result based on the specifications of the method being tested.
- Actual Results: We will compare the actual results of the tests with the expected results (actual results of the tests with the expected results).

1. StackIntTest

This test class is StackIntTest, which is the JUnit test class used to test the StackInt class. This test class contains a set of test methods that test all the methods of the StackInt class.

Here's the test plan, test data, expected and actual results for each test method in the 'StackIntTest' class:

1. 'testPushAndPop()' :

- Test plan: Test that the 'push' and 'pop' methods operate properly.

- Test data: 'push' an element onto the stack.

- Expected result: The 'pop' method should return 1, which is the element we 'push' into.

- Actual result: Run the test and check if the result returned is 1.

2. 'testPushAndPeek()' :

- Test plan: Test the 'push' and 'peek' methods.

- Test data: 'push' an element onto the stack.

- Expected result: The 'peek' method should return 1, which is the element we 'push' into, but won't remove it from the stack.

- Actual result: Run the test and check if the result returned is 1.

3. 'testPushAndPopAndPeek()' :

- Test plan: Test combinations of 'push', 'pop', and 'peek' methods.

- Test data: 'push' an element 1 onto the stack, then 'pop', then 'push' an element 2.

- Expected result: The 'peek' method should return 2, which is the last element we 'push' into.

- Actual result: Run the test and check if the result returned is 2.

4. 'testPushAndPopAndPeek2()' :

- Test plan: Test combinations of 'push', 'pop', and 'peek' methods.

- Test data: 'push' two elements 1 and 2 onto the stack, then 'pop'.

- Expected result: The 'peek' method should return 1, which is now the element at the top of the stack.

- Actual result: Run the test and check if the result returned is 1.

5. 'testPushAndPopAndPeek3()' :

- Test plan: Test combinations of 'push', 'pop', and 'peek' methods.

- Test data: 'push' three elements 1,2, and 3 onto the stack, then 'pop'.

- Expected result: The 'peek' method should return 2, which is now the top element of the stack.

- Actual result: Run the test and check if the result returned is 2.

6. 'testPopOnEmptyStack()' :

- Test plan: Test that trying to 'pop' elements throws an 'AssertionError' when the stack is empty.

- Test data: an empty stack.

- Expected result: An 'AssertionError' should be thrown.

- Actual result: runs the test and checks if an 'AssertionError' is thrown.

7. 'testPeekOnEmptyStack()' :

- Test plan: Test that trying to 'peek' an element will throw an 'AssertionError'

when the stack is empty.

- Test data: an empty stack.

- Expected result: An 'AssertionError' should be thrown.

- Actual result: runs the test and checks if an 'AssertionError' is thrown.

8. 'testPushBeyondCapacity()' :

- Test plan: Test that trying to 'push' elements throws an 'AssertionError' when the stack is full.

- Test data: 'push' 11 items onto the stack (exceeds stack capacity).

- Expected result: An 'AssertionError' should be thrown.

- Actual result: runs the test and checks if an 'AssertionError' is thrown.

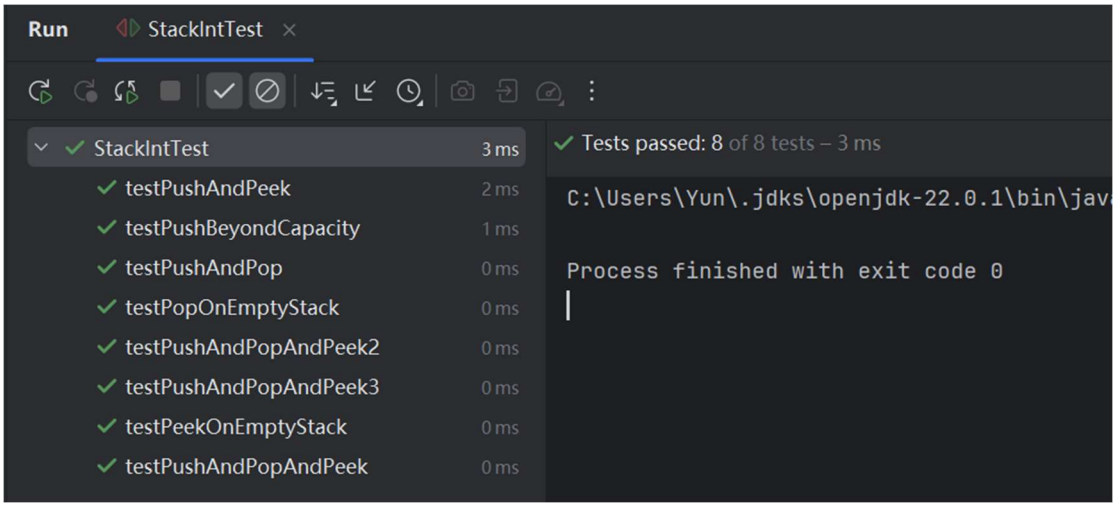
The purpose of this test class is to ensure that all the methods of the StackInt class work as expected, including correctly throwing exceptions in case of error conditions.

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class StackIntTest {
    private StackInt stack;

    @Before
    public void setUp() {
        stack = new StackInt(10);
    }

    @Test
    public void testPushAndPop() {
```

```
        stack.push(1);
        assertEquals(1, stack.pop());
    }

    @Test
    public void testPushAndPeek() {
        stack.push(1);
        assertEquals(1, stack.peek());
    }

    @Test
    public void testPushAndPopAndPeek() {
        stack.push(1);
        assertEquals(1, stack.pop());
        stack.push(2);
        assertEquals(2, stack.peek());
    }

    @Test
    public void testPushAndPopAndPeek2() {
        stack.push(1);
        stack.push(2);
        assertEquals(2, stack.pop());
        assertEquals(1, stack.peek());
    }

    @Test
    public void testPushAndPopAndPeek3() {
        stack.push(1);
        stack.push(2);
        stack.push(3);
        assertEquals(3, stack.pop());
        assertEquals(2, stack.peek());
    }

    @Test(expected = AssertionError.class)
    public void testPopOnEmptyStack() {
        stack.pop();
    }

    @Test(expected = AssertionError.class)
    public void testPeekOnEmptyStack() {
        stack.peek();
    }
```

```
    @Test(expected = AssertionError.class)
    public void testPushBeyondCapacity() {
        for (int i = 0; i < 11; i++) {
            stack.push(i);
        }
    }
}
```

Actual result:



2. QueueIntTest

Here are the test plans, test data, expected and actual results for each test method:

1.TestAddToBackAndRemovefromFront () :

Test plan: Test the addToBack and removefromFront methods for normal operation.

Test: Add elements 1, 2, and 3 to the queue.

Expected result: The removefromFront method should return 1, which is the element we added to the queue first.

The actual result: Runs the test and checks if it returns 1.

2.TestRemovefromFrontOnEmptyQueue () :

Test plan: Test that trying to use removefromFront throws an AssertionError when the queue is empty.

Test data: An empty queue.

Expected result: An AssertionError should be thrown.

Actual result: Run the test and check if an AssertionError is thrown.

3.TestAddToBackBeyondCapacity () :

Test plan: Test that trying to use addToBack throws an AssertionError when the queue is full.

Test data: Add 11 elements to the queue (more than the capacity of the queue)

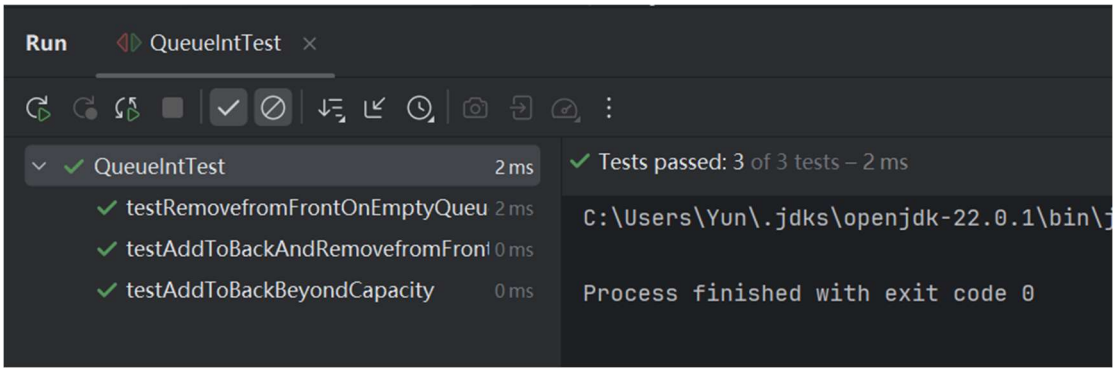
Expected result: An AssertionError should be thrown.

Actual result: Run the test and check if an AssertionError is thrown.

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class QueueIntTest {
    private QueueInt queue;

    @Before
    public void setUp() {
        queue = new QueueInt(10);
    }

    @Test
    public void testAddToBackAndRemovefromFront() {
        queue.addToBack(1);
        queue.addToBack(2);
```

```
        queue.addToBack(3);
        assertEquals(1, queue.removefromFront());
    }

    @Test(expected = AssertionError.class)
    public void testRemovefromFrontOnEmptyQueue() {
        queue.removefromFront();
    }

    @Test(expected = AssertionError.class)
    public void testAddToBackBeyondCapacity() {
        for (int i = 0; i < 11; i++) {
            queue.addToBack(i);
        }
    }
}
```

Actual result:



3. ListIntTest

Here are the test plans, test data, expected and actual results for each test method:

1.testAppend () :

Test plan: Test the proper operation of the append method.

Test: Add element 1 to the list.

Expected result: The contains method should return true, indicating that the

list contains the item we added.

Actual result: Runs the test and checks if it returns true.

2.testAppendBeyondCapacity() :

Test plan: Test that trying to use the append method throws an AssertionError when the list is full.

Test data: Add 11 items to the list (more than the list capacity).

Expected result: An AssertionError should be thrown.

Actual result: Run the test and check if an AssertionError is thrown.

3.testContains() :

Test plan: Test the contains method for normal operation.

Test: Add element 1 to the list, and then check whether the list contains elements 1 and 2.

Expected result: The contains method should return true for element 1. For element 2, the contains method should return false.

Actual results: Run the tests and check that the results returned are as expected.

4.testContainsOnEmptyList() :

Test plan: Test that trying to use the contains method throws an AssertionError when the list is empty.

Test data: An empty list.

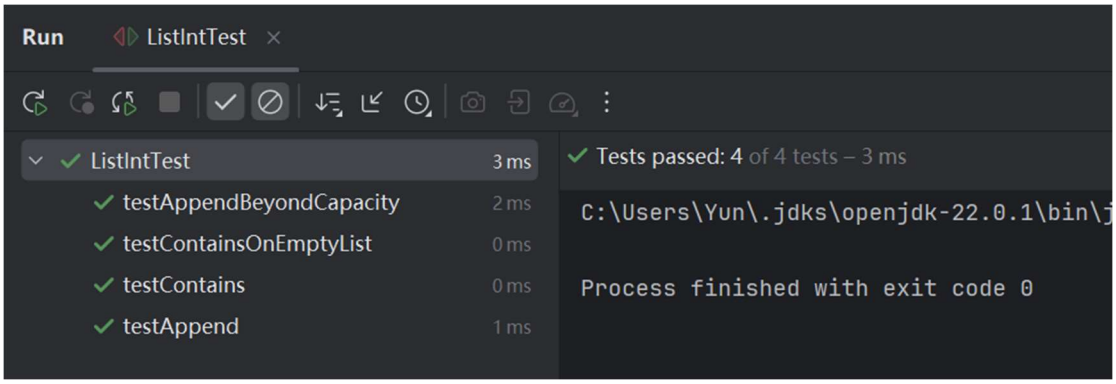
Expected result: An AssertionError should be thrown.

Actual result: Run the test and check if an AssertionError is thrown.

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class ListIntTest {
    private ListInt list;

    @Before
    public void setUp() {
        list = new ListInt(10);
    }

    @Test
    public void testAppend() {
        list.append(1);
        assertTrue(list.contains(1));
    }

    @Test(expected = AssertionError.class)
    public void testAppendBeyondCapacity() {
        for (int i = 0; i < 11; i++) {
            list.append(i);
        }
    }

    @Test
    public void testContains() {
        list.append(1);
        assertTrue(list.contains(1));
        assertFalse(list.contains(2));
    }

    @Test(expected = AssertionError.class)
    public void testContainsOnEmptyList() {
        list.contains(1);
```

```
    }
}
```

Actual Result:



4. SetIntTest

First, we write a test method for each method of the SetInt class. These test methods will include normal operation testing (using valid inputs) and error handling testing (using invalid inputs).

Here are the test plans, test data, expected and actual results for each test method:

1.testIncludeAndContains() :

Test plan: Test the include and contains methods for normal operation.

Test: Add element 1 to the collection.

Expected result: The contains method should return true, indicating that the collection contains the element we added.

Actual result: Runs the test and checks if it returns true.

2.TestIncludeBeyondCapacity () :

Test plan: Test that trying to use include throws an AssertionError when the

collection is full.

Test: Add 11 items to the collection (more than the collection capacity).

Expected result: An AssertionError should be thrown.

Actual result: Run the test and check if an AssertionError is thrown.

3.testExclude() :

Test plan: Test the exclude method for normal operation.

Test: Add element 1 to the collection, then remove element 1.

Expected result: The contains method should return false, indicating that the collection does not contain the item we removed.

Actual result: Runs the test and checks if it returns false.

4.TestExcludeNonExistentElement () :

Test plan: Test that trying to use the exclude method throws an AssertionError when an element is not present in the collection.

Test: Tries to remove element 1 from the set, but the set doesn't contain element 1.

Expected result: An AssertionError should be thrown.

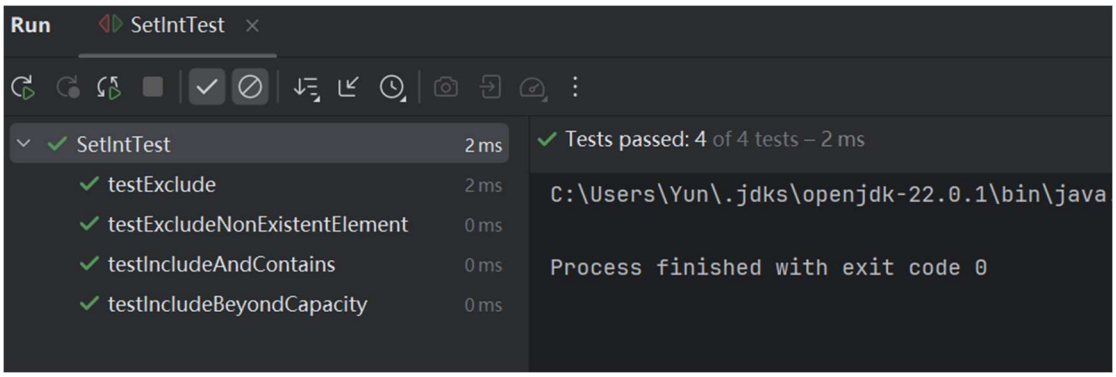
Actual result: Run the test and check if an AssertionError is thrown.

Here is the implementation of the SetIntTest test class:

```
import org.junit.Before;
import org.junit.Test;
```

```
import static org.junit.Assert.*;

public class SetIntTest {
    private SetInt set;

    @Before
    public void setUp() {
        set = new SetInt(10);
    }

    @Test
    public void testIncludeAndContains() {
        set.include(1);
        assertTrue(set.contains(1));
    }

    @Test(expected = AssertionError.class)
    public void testIncludeBeyondCapacity() {
        for (int i = 0; i < 11; i++) {
            set.include(i);
        }
    }

    @Test
    public void testExclude() {
        set.include(1);
        set.exclude(1);
        assertFalse(set.contains(1));
    }

    @Test(expected = AssertionError.class)
    public void testExcludeNonExistentElement() {
        set.exclude(1);
    }
}
```

Actual Result:



Task_6

```java
public class StationInfo implements IStationInfo {
    private String name;
    private int xPos;
    private int yPos;

    public StationInfo(String name, int xPos, int yPos) {
        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;
    }
}
```

The current class is' StationInfo ', which implements the 'IStationInfo' interface.

This class is used to represent information about a site, including its name, x coordinates, and y coordinates.

Here's an explanation of each method:

- 'StationInfo(String name, int xPos, int yPos)' : This is a class constructor that creates a new 'StationInfo' object. You need to provide the name, x-coordinate, and y-coordinate of the site.

- 'getName()' : This method returns the name of the site.

- 'getxPos()' : This method returns the x-coordinate of the site.

- 'getyPos()' : This method returns the y-coordinate of the site.

The purpose of this class is to encapsulate information about a site and provide a way to get it.

Task_7 & Task _8

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Network {  2 usages
    final double NO_LINK = Double.MAX_VALUE;  9 usages
    int numStations;  26 usages
    double[][] distance;  11 usages
    StationInfo[] stations;  12 usages

    public Network(int capacity) {  1 usage
        numStations = 0;
        distance = new double[capacity][capacity];
        stations = new StationInfo[capacity];
        for (int i = 0; i < capacity; i++) {
            for (int j = 0; j < capacity; j++) {
                if (i == j) {
                    distance[i][j] = NO_LINK;
                } else {
                    distance[i][j] = NO_LINK;
                }
            }
        }
    }
```

In the current Network class, the requirements of tasks 7 and 8 have been

implemented.

Task 7 requires the creation of a Java class to define the adjacency matrix. In the Network class, we define a two-dimensional array distance as the adjacency matrix. For all i and j (0 <= i, j < numStations), distance[i][j] denotes the distance from station numbered i to station numbered j. distance[i][j] is equal to distance[j][i], indicating that the distance is bidirectional. For all i (0 <= i < numStations), distance[i][i] is equal to NO_LINK, indicating that a station is infinitely far from itself.

Task 8 requires the definition of a list of StationInfo objects, which are stored in an array. In the Network class, we define an array of StationInfo type stations to store the station information.

In the constructor of the Network class, we initialize distance and stations. For distance, we initialize all elements to NO_LINK. For stations, we initialize it to a StationInfo array of a specified capacity

Task_9

The readNetworkFromFile method in the Network class is used to read a text file containing a network description and build a list of sites and a matrix of links.

The validity checks of this method include:

checking if the coordinates of the site can be resolved as integers,

checking if the name of the site already exists,

checking if the maximum number of sites has been exceeded,

checking if the distance of the link can be resolved as a floating point number, checking if the two sites of the link exist, and checking if the link already exists

The implementation of this method is as follows:

```
public void readNetworkFromFile(String filename) throws IOException {
    BufferedReader reader = new BufferedReader(new
FileReader(filename));
    String line;
    while ((line = reader.readLine()) != null) {
        String[] words = line.split(" ");
        if (words[0].equals("\"station\"")) {
            String name = words[1];
            int xPos;
            int yPos;
            try {
                xPos = Integer.parseInt(words[2]);
                yPos = Integer.parseInt(words[3]);
            } catch (NumberFormatException e) {
                throw new IllegalArgumentException("Invalid
coordinates for station: " + name);
            }
            if (findStationIndex(name) != -1) {
                throw new IllegalArgumentException("Duplicate station:
" + name);
            }
            if (numStations >= stations.length) {
                throw new IllegalArgumentException("Exceeded capacity
of stations");
            }
            stations[numStations] = new StationInfo(name, xPos, yPos);
            numStations++;
        } else if (words[0].equals("\"link\"")) {
            String name1 = words[1];
            String name2 = words[2];
            double dist;
            try {
                dist = Double.parseDouble(words[3]);
            } catch (NumberFormatException e) {
                throw new IllegalArgumentException("Invalid distance
for link: " + name1 + " - " + name2);
            }
            int index1 = findStationIndex(name1);
```

```
            int index2 = findStationIndex(name2);
            if (index1 == -1 || index2 == -1) {
                throw new IllegalArgumentException("Invalid link: " +
name1 + " - " + name2);
            }
            if (distance[index1][index2] != NO_LINK ||
distance[index2][index1] != NO_LINK) {
                throw new IllegalArgumentException("Duplicate link: "
+ name1 + " - " + name2);
            }
            distance[index1][index2] = dist;
            distance[index2][index1] = dist;
        }
    }
    reader.close();
}
```

This method first opens a file reader and then reads the file line by line. For each line, it first segments it into words. It then performs different actions depending on whether the first word is "station" or "link".

If the first word is "station", it creates a new StationInfo object and adds it to the stations array. In the process, it checks if the coordinates of the site are valid (i.e., if they can be resolved to integers), if the name of the site already exists, and if the maximum number of sites has been exceeded.

If the first word is "link", it adds a new link to the distance matrix. In the process, it checks if the distance of the link is valid (i.e., if it can be resolved as a floating point number), if the two sites of the link exist, and if the link already exists. If at any point an error is encountered, it throws an IllegalArgumentException with a message describing the error.

This method closes the file reader when it is finished reading the file.

Also, in the 'Network' class, the 'readNetworkFromFile' method is responsible for reading a text file containing a description of the network and building the corresponding list of sites and adjacency matrix.

When a line starts with '"station"' is encountered, this method creates a new 'StationInfo' object and adds it to the 'stations' array. This array is the list of sites.

When a line starts with '"link"' is encountered, this method adds a new link to the 'distance' matrix. This matrix is the adjacency matrix and represents the distance between sites. If there is no link between the two sites, then the corresponding matrix element has the value 'NO_LINK', which means infinity.

By reading every line in the file, the 'readNetworkFromFile' method can build up a list of sites and an adjacency matrix that represents the entire network.

Task_10

Two methods are used to print network information to the console:

```
public void printStations() {
    System.out.printf("%-20s %-10s %-10s\n", "Name", "X Position", "Y
Position");
    for (int i = 0; i < numStations; i++) {
        StationInfo station = stations[i];
        System.out.printf("%-20s %-10d %-10d\n", station.getName(),
station.getxPos(), station.getyPos());
    }
}

public void printMatrix() {
    // Determine the maximum length of station names
    int maxNameLength = 0;
    for (int i = 0; i < numStations; i++) {
```

```
        maxNameLength = Math.max(maxNameLength,
stations[i].getName().length());
    }

    // Print column labels
    System.out.print(String.format("%" + (maxNameLength + 2) + "s",
""));
    for (int i = 0; i < numStations; i++) {
        System.out.printf("%" + (maxNameLength + 2) + "s",
stations[i].getName());
    }
    System.out.println();

    // Print rows
    for (int i = 0; i < numStations; i++) {
        // Print row label
        System.out.printf("%-" + (maxNameLength + 2) + "s",
stations[i].getName());

        // Print row data
        for (int j = 0; j < numStations; j++) {
            if (distance[i][j] == NO_LINK) {
                System.out.printf("%" + (maxNameLength + 2) + "s",
"NO LINK");
            } else {
                System.out.printf("%" + (maxNameLength + 2) + ".2f",
distance[i][j]);
            }
        }
        System.out.println();
    }
}
```

The 'printStations' method iterates through the' stations' array and prints the name, x - and y-coordinates of each station. This method first prints the list header, then iterates through the 'stations' array, and for each station, prints its name, x coordinates, and y coordinates.

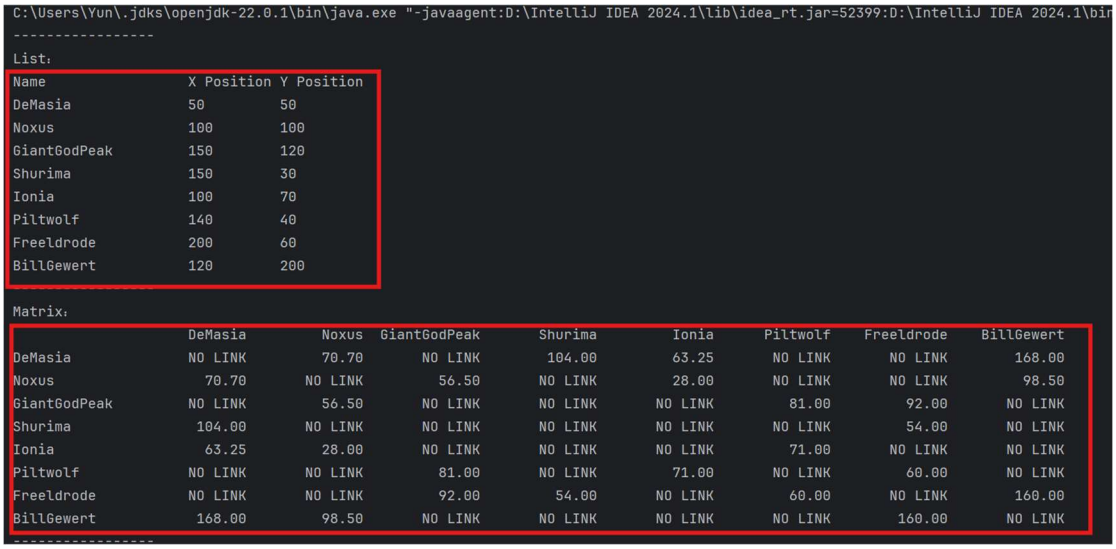
The 'printMatrix' method prints out an adjacency matrix representing the

distance between sites. This method first calculates the maximum length of the site name and then prints the column label (that is, the site name). Next, it iterates over the adjacency matrix, and for each row, it prints the trip label (the station name) and then the data for that row (the distance to the other stations). If there is NO LINK between the two sites, then the value of the corresponding matrix element is' NO_LINK ', which means infinity, and "NO LINK" is printed. Otherwise, print the distance to the other stations.

Main code:

```
import java.io.IOException;

public class Main {
    public static void main(String[] args) throws IOException {
        Network network = new Network( capacity: 8);
        network.readNetworkFromFile( filename: "network.txt");
        System.out.println("-----------------");
        System.out.println("List: ");
        network.printStations();
        System.out.println("-----------------");
        System.out.println("Matrix: ");
        network.printMatrix();
        System.out.println("-----------------");
        network.breadthFirstTraversal( start: "DeMasia");
        System.out.println("-----------------");
        network.depthFirstTraversal( start: "DeMasia");
        System.out.println("-----------------");
        network.dijkstraShortestPath("DeMasia", "Freeldrode");
    }
}
```

Actual result:



```
C:\Users\Yun\.jdks\openjdk-22.0.1\bin\java.exe "-javaagent:D:\IntelliJ IDEA 2024.1\lib\idea_rt.jar=52399:D:\IntelliJ IDEA 2024.1\bi
----------
List:
Name            X Position Y Position
DeMasia         50         50
Noxus           100        100
GiantGodPeak    150        120
Shurima         150        30
Ionia           100        70
Piltwolf        140        40
Freeldrode      200        60
BillGewert      120        200

Matrix:
                DeMasia      Noxus  GiantGodPeak      Shurima       Ionia     Piltwolf    Freeldrode    BillGewert
DeMasia         NO LINK      70.70       NO LINK       104.00       63.25      NO LINK       NO LINK       168.00
Noxus             70.70    NO LINK         56.50      NO LINK       28.00      NO LINK       NO LINK        98.50
GiantGodPeak    NO LINK      56.50       NO LINK      NO LINK      NO LINK       81.00        92.00      NO LINK
Shurima          104.00    NO LINK       NO LINK      NO LINK      NO LINK      NO LINK        54.00      NO LINK
Ionia             63.25      28.00       NO LINK      NO LINK      NO LINK       71.00       NO LINK      NO LINK
Piltwolf        NO LINK    NO LINK         81.00      NO LINK       71.00      NO LINK        60.00      NO LINK
Freeldrode      NO LINK    NO LINK         92.00        54.00      NO LINK       60.00       NO LINK       160.00
BillGewert       168.00      98.50       NO LINK      NO LINK      NO LINK      NO LINK       160.00      NO LINK
----------
```

Both methods use the 'printf' method to format the output to ensure that the output is tidy.

Task_11

In the 'Network' class, we have implemented both depthFirstTraversal and breadthFirstTraversal methods.

The depthFirstTraversal 'method is implemented as follows:

```
public ListInt depthFirstTraversal(String start) {
    StationInfo startStation = stations[findStationIndex(start)];
    StackInt stack = new StackInt(numStations);
    ListInt visited = new ListInt(numStations);

    stack.push(findStationIndex(startStation.getName()));

    System.out.println("Depth First Traversal");
    System.out.println("Started: " + startStation.getName());

    while (stack.getSize() != 0) {
        StationInfo station = stations[stack.pop()];

        if (!visited.contains(findStationIndex(station.getName())))
{
```

```
            visited.append(findStationIndex(station.getName()));
            System.out.print("->" + station.getName());
            for (int i = 0; i < numStations; i++) {
                if
(distance[findStationIndex(station.getName())][i] != NO_LINK
&& !visited.contains(i)){
                    stack.push(i);
                }
            }
        }
    }
    System.out.println();
    return visited;
}
```

This method first pushes the starting site onto the stack, and then when the stack is not empty, pops the site at the top of the stack, marks it as visited if it has not been visited, and pushes the adjacent unvisited site onto the stack. This process continues until the stack is empty, that is, all reachable sites have been visited.

The breadthFirstTraversal method is implemented as follows:

```
public ListInt breadthFirstTraversal(String start) {
    StationInfo startStation = stations[findStationIndex(start)];
    QueueInt queue = new QueueInt(numStations);
    ListInt visited = new ListInt(numStations);

    queue.addToBack(findStationIndex(startStation.getName()));

    System.out.println("Breadth First Traversal");
    System.out.println("Started: " + startStation.getName());

    while (queue.getSize() != 0) {
        StationInfo station = stations[queue.removefromFront()];

        if (!visited.contains(findStationIndex(station.getName()))) {
            visited.append(findStationIndex(station.getName()));
            System.out.print("->" + station.getName());

            for (int i = 0; i < numStations; i++) {
```

```
                if
(distance[findStationIndex(station.getName())][i] != NO_LINK
&& !visited.contains(i)) {
                    queue.addToBack(i);
                }
            }
        }
    }
    System.out.println();
    return visited;
}
```

This method first adds the starting station to the queue, and then, if the queue is not empty, removes a station from the front of the queue, marks it as visited if it has not been visited, and adds its neighboring unvisited station to the back of the queue. This process continues until the queue is empty, that is, all reachable sites have been visited.

Main code:

```java
import java.io.IOException;

public class Main {
    public static void main(String[] args) throws IOException {
        Network network = new Network( capacity: 8);
        network.readNetworkFromFile( filename: "network.txt");
        System.out.println("-----------------");
        System.out.println("List: ");
        network.printStations();
        System.out.println("-----------------");
        System.out.println("Matrix: ");
        network.printMatrix();
        System.out.println("-----------------");
        network.breadthFirstTraversal( start: "DeMasia");
        System.out.println("-----------------");
        network.depthFirstTraversal( start: "DeMasia");
        System.out.println("-----------------");
        network.dijkstraShortestPath("DeMasia", "Freeldrode");
    }
}
```

Actual result:

```
-----------------
Breadth First Traversal
Started: DeMasia
->DeMasia->Noxus->Shurima->Ionia->BillGewert->GiantGodPeak->Freeldrode->Piltwolf
-----------------
Depth First Traversal
Started: DeMasia
->DeMasia->BillGewert->Freeldrode->Piltwolf->Ionia->Noxus->GiantGodPeak->Shurima
-----------------
```

Both methods return a ListInt indicating the order in which sites were visited during the traversal.

Task_12

In the Network class, Dijkstra's algorithm, which is used to find the shortest path between two stations in a network, has been implemented. The name of

this method is dijkstraShortestPath.

Here is the implementation of the method:

```
public void dijkstraShortestPath(String start, String end) {
    // Find the index of the start and end stations
    int startIdx = findStationIndex(start);
    int endIdx = findStationIndex(end);

    // Initialize the distances array with NO_LINK (infinity)
    double[] distances = new double[numStations];
    for (int i = 0; i < numStations; i++) {
        distances[i] = NO_LINK;
    }
    // Set the distance from the start station to itself as 0
    distances[startIdx] = 0;

    // Initialize the previous array to keep track of the path
    int[] previous = new int[numStations];
    for (int i = 0; i < numStations; i++) {
        previous[i] = -1;
    }

    // Initialize the visited array to keep track of which stations
have been visited
    boolean[] visited = new boolean[numStations];
    for (int i = 0; i < numStations; i++) {
        visited[i] = false;
    }

    // Initialize the open array to keep track of which stations are
open for visiting
    boolean[] open = new boolean[numStations];
    for (int i = 0; i < numStations; i++) {
        open[i] = false;
    }

    // Mark the start station as open
    open[startIdx] = true;

    // Initialize the iterations counter
    int iterations = 0;

    // While the end station has not been visited
```

```
    while (!visited[endIdx]) {
        // Find the open station with the smallest distance
        double minDist = NO_LINK;
        int minIdx = -1;
        for (int i = 0; i < numStations; i++) {
            if (open[i] && distances[i] < minDist) {
                minDist = distances[i];
                minIdx = i;
            }
        }
        // If no such station exists, break the loop
        if (minIdx == -1) {
            break;
        }
        // Mark the station as visited and not open
        open[minIdx] = false;
        visited[minIdx] = true;

        // If the end station has been visited, break the loop
        if (minIdx == endIdx) {
            break;
        }
        // Otherwise, update the distances to the neighboring
stations
        else {
            for (int i = 0; i < numStations; i++) {
                if (distance[minIdx][i] != NO_LINK && !visited[i]) {
                    double newDist = distances[minIdx] +
distance[minIdx][i];
                    if (newDist < distances[i]) {
                        distances[i] = newDist;
                        previous[i] = minIdx;
                        open[i] = true;
                    }
                }
            }
        }
        // Increment the iterations counter
        iterations++;
    }

    // If no path was found, print a message
    if (distances[endIdx] == NO_LINK) {
        System.out.println("No path found from " + start + " to " +
```

```
end);
    }
    // Otherwise, print the shortest path and its distance
    else {
        System.out.println("Dijkstra Shortest Path");
        System.out.println("Shortest path from " + start + " to " +
end + ":");
        int idx = endIdx;
        StackInt stack = new StackInt(numStations);
        // Push the stations on the shortest path onto a stack
        while (idx != -1) {
            stack.push(idx);
            idx = previous[idx];
        }
        // Pop the stations from the stack and print them
        while (stack.getSize() != 0) {
            System.out.print(stations[stack.pop()].getName());
            if (stack.getSize() != 0) {
                System.out.print("->");
            }
        }
        System.out.println();
        System.out.println("Distance: " + distances[endIdx]);
    }
    // Print the number of iterations
    System.out.println("Number of iterations: " + iterations);
}
```

Main code:

```java
public class Main {
    public static void main(String[] args) throws IOException {
        Network network = new Network( capacity: 8);
        network.readNetworkFromFile( filename: "network.txt");
        System.out.println("-----------------");
        System.out.println("List: ");
        network.printStations();
        System.out.println("-----------------");
        System.out.println("Matrix: ");
        network.printMatrix();
        System.out.println("-----------------");
        network.breadthFirstTraversal( start: "DeMasia");
        System.out.println("-----------------");
        network.depthFirstTraversal( start: "DeMasia");
        System.out.println("-----------------");
        network.dijkstraShortestPath("DeMasia", "Freeldrode");
    }
}
```

Actual Result:

```
-----------------
Dijkstra Shortest Path
Shortest path from DeMasia to Freeldrode:
DeMasia->Shurima->Freeldrode
Distance: 158.0
Number of iterations: 6

Process finished with exit code 0
```

The method first initializes a few arrays: The distances array is used to store the shortest distance from the start site to each site, the previous array is used to store the previous site for each site in the shortest path, the visited array is used to indicate whether each site has already been visited, and the open array is used to indicate whether each site can still be visited.

This method then goes into a loop until the end site is visited. In each loop, it

first finds the site with the smallest distance among the unvisited sites, then marks this site as visited and updates the distances of other sites that can be directly reached from this site.

Finally, the method prints the shortest path and distance from the start station to the end station, as well as the number of iterations. The design and implementation of this method follow the principle of Dijkstra's algorithm, and also consider the needs of practical applications, such as printing the shortest path, the shortest distance, and the number of cycles.

Task_13

Both the Dijkstra algorithm and the A* algorithm are algorithms used to find the shortest path between two nodes in a graph, but they work differently.

The Dijkstra algorithm starts from the starting node and gradually expands to all reachable nodes until the goal node is reached. At each step, it selects the node with the shortest distance from the starting node to that node among the currently unvisited nodes, and then updates the distances of other nodes that can be directly reached from that node. This process continues until all nodes have been visited or the shortest path to the destination node has been found.

The A* algorithm also starts from the start node, but when selecting the next node to visit, it takes into account not only the distance from the start node to that node, but also a heuristic function that is the estimated distance from that node to the goal node. In this way, the A* algorithm can visit the nodes closer to the target node earlier, so that in a graph with a large number of nodes, it may be faster to find the shortest path than the Dijkstra algorithm.

However, in my network, since all the stations lie on a two-dimensional plane and the coordinates of each station are known, the Euclidean distance can be used as a heuristic function. However, due to the relatively small number of sites in your network, Dijkstra and A* are likely to be less efficient in practice.

In general, both Dijkstra algorithm and A* algorithm can be used to find the shortest path between two stations in the network, but A* algorithm may have better performance when dealing with large-scale problems. However, this requires a suitable heuristic function, which may not be readily available in some problems.

Source Code

Class Main

```
import java.io.IOException;

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

```
        Network network = new Network(8);
        network.readNetworkFromFile("network.txt");
        System.out.println("-----------------");
        System.out.println("List: ");
        network.printStations();
        System.out.println("-----------------");
        System.out.println("Matrix: ");
        network.printMatrix();
        System.out.println("-----------------");
        network.breadthFirstTraversal("DeMasia");
        System.out.println("-----------------");
        network.depthFirstTraversal("DeMasia");
        System.out.println("-----------------");
        network.dijkstraShortestPath("DeMasia", "Freeldrode");
    }
}
```

Class StackInt

```
/**
 * StackInt is a concrete implementation of the abstract class
AbsStackInt.
 * It represents a stack of integers with a fixed capacity.
 * The stack is implemented using an array.
 */
public class StackInt extends AbsStackInt {

    /**
     * Constructs a new StackInt with the specified capacity.
     *
     * @param capacity the maximum number of elements the stack can
hold
     */
    public StackInt(int capacity) {
        super(capacity);
    }

    /**
     * Pushes an integer onto the top of the stack.
     * If the stack is full, an AssertionError is thrown.
     *
     * @param n the integer to be pushed onto the stack
     */
```

```
    @Override
    public void push(int n) {
        assert getSize() != getCapacity() : "Stack is full";
        stack[size++] = n;
    }

    /**
     * Removes and returns the integer at the top of the stack.
     * If the stack is empty, an AssertionError is thrown.
     *
     * @return the integer at the top of the stack
     */
    @Override
    public int pop() {
        assert getSize() != 0 : "Stack is empty";
        return stack[--size];
    }

    /**
     * Returns the integer at the top of the stack without removing
it.
     * If the stack is empty, an AssertionError is thrown.
     *
     * @return the integer at the top of the stack
     */
    @Override
    public int peek() {
        assert getSize() != 0 : "Stack is empty";
        return stack[size - 1];
    }
}
```

Class ListInt

```
/**
 * ListInt is a concrete implementation of the abstract class
AbsListInt.
 * It represents a list of integers with a fixed capacity.
 * The list is implemented using an array.
 */
public class ListInt extends AbsListInt {

    /**
```

```
     * Constructs a new ListInt with the specified capacity.
     *
     * @param capacity the maximum number of elements the list can
hold
     */
    public ListInt(int capacity) {
        super(capacity);
    }

    /**
     * Appends an integer to the end of the list.
     * If the list is full, an AssertionError is thrown.
     *
     * @param n the integer to be appended to the list
     */
    @Override
    public void append(int n) {
        assert getSize() != getCapacity() : "List is full";
        list[size] = n;
        size++;
    }

    /**
     * Checks if the list contains a specific integer.
     * If the list is empty, an AssertionError is thrown.
     *
     * @param x the integer to be checked
     * @return true if the list contains the integer, false otherwise
     */
    @Override
    public boolean contains(int x) {
        assert getSize() != 0 : "List is empty";
        for (int i = 0; i < getSize(); i++) {
            if (list[i] == x) {
                return true;
            }
        }
        return false;
    }
}
```

Class QueueInt

```java
/**
 * QueueInt is a concrete implementation of the abstract class
AbsQueueInt.
 * It represents a queue of integers with a fixed capacity.
 * The queue is implemented using an array.
 */
public class QueueInt extends AbsQueueInt {

    /**
     * Constructs a new QueueInt with the specified capacity.
     *
     * @param capacity the maximum number of elements the queue can
hold
     */
    public QueueInt(int capacity) {
        super(capacity);
    }

    /**
     * Adds an integer to the back of the queue.
     * If the queue is full, an AssertionError is thrown.
     *
     * @param n the integer to be added to the back of the queue
     */
    @Override
    public void addToBack(int n) {
        assert getSize() != getCapacity() : "Queue is full";
        queue[size++] = n;
    }

    /**
     * Removes and returns the integer from the front of the queue.
     * If the queue is empty, an AssertionError is thrown.
     *
     * @return the integer from the front of the queue
     */
    @Override
    public int removefromFront() {
        assert getSize() != 0 : "Queue is empty";
        int removed = queue[0];
        for (int i = 0; i < getSize() - 1; i++) {
            queue[i] = queue[i + 1];
        }
        size--;
```

```
        return removed;
    }
}
```

Class SetInt

```
public class SetInt extends AbsSetInt {

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

    /**
     * @param "x" -- value to be sought
     * @pre true
     * @return true iff x is in list*/
    @Override
    public boolean contains(int n) {
        if (getSize() == 0) {
            return false;
        }
        for (int i = 0; i < getSize(); i++) {
            if (set[i] == n) {
                return true;
            }
        }
        return false;
    }

    /**
     * @param n node to be added
     * @pre contains(n) || getSize() != getCapacity()
     * @post contains(n)
     */
    @Override
    public void include(int n) {
        assert !contains(n) : "Element already exists in the set";
        assert getSize() != getCapacity() : "Set is full";
        set[size++] = n;
    }

    /**
     * @pre true
```

```
     * @post !contains(n)
     */
    @Override
    public void exclude(int n) {
        assert contains(n) : "Element does not exist in the set";
        for (int i = 0; i < getSize(); i++) {
            if (set[i] == n) {
                for (int j = i; j < getSize() - 1; j++) {
                    set[j] = set[j + 1];
                }
                size--;
                break;
            }
        }
    }
}
```

Class StackIntTest

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class StackIntTest {
    private StackInt stack;

    @Before
    public void setUp() {
        stack = new StackInt(10);
    }

    @Test
    public void testPushAndPop() {
        stack.push(1);
        assertEquals(1, stack.pop());
    }

    @Test
    public void testPushAndPeek() {
        stack.push(1);
        assertEquals(1, stack.peek());
    }
```

```
    @Test
    public void testPushAndPopAndPeek() {
        stack.push(1);
        assertEquals(1, stack.pop());
        stack.push(2);
        assertEquals(2, stack.peek());
    }

    @Test
    public void testPushAndPopAndPeek2() {
        stack.push(1);
        stack.push(2);
        assertEquals(2, stack.pop());
        assertEquals(1, stack.peek());
    }

    @Test
    public void testPushAndPopAndPeek3() {
        stack.push(1);
        stack.push(2);
        stack.push(3);
        assertEquals(3, stack.pop());
        assertEquals(2, stack.peek());
    }

    @Test(expected = AssertionError.class)
    public void testPopOnEmptyStack() {
        stack.pop();
    }

    @Test(expected = AssertionError.class)
    public void testPeekOnEmptyStack() {
        stack.peek();
    }

    @Test(expected = AssertionError.class)
    public void testPushBeyondCapacity() {
        for (int i = 0; i < 11; i++) {
            stack.push(i);
        }
    }
}
```

Class ListIntTest

```
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class ListIntTest {
    private ListInt list;

    @Before
    public void setUp() {
        list = new ListInt(10);
    }

    @Test
    public void testAppend() {
        list.append(1);
        assertTrue(list.contains(1));
    }

    @Test(expected = AssertionError.class)
    public void testAppendBeyondCapacity() {
        for (int i = 0; i < 11; i++) {
            list.append(i);
        }
    }

    @Test
    public void testContains() {
        list.append(1);
        assertTrue(list.contains(1));
        assertFalse(list.contains(2));
    }

    @Test(expected = AssertionError.class)
    public void testContainsOnEmptyList() {
        list.contains(1);
    }
}
```

Class QueueIntTest

```java
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class QueueIntTest {
    private QueueInt queue;

    @Before
    public void setUp() {
        queue = new QueueInt(10);
    }

    @Test
    public void testAddToBackAndRemovefromFront() {
        queue.addToBack(1);
        queue.addToBack(2);
        queue.addToBack(3);
        assertEquals(1, queue.removefromFront());
    }

    @Test(expected = AssertionError.class)
    public void testRemovefromFrontOnEmptyQueue() {
        queue.removefromFront();
    }

    @Test(expected = AssertionError.class)
    public void testAddToBackBeyondCapacity() {
        for (int i = 0; i < 11; i++) {
            queue.addToBack(i);
        }
    }
}
```

Class SetIntTest

```java
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class SetIntTest {
    private SetInt set;

    @Before
```

```
    public void setUp() {
        set = new SetInt(10);
    }

    @Test
    public void testIncludeAndContains() {
        set.include(1);
        assertTrue(set.contains(1));
    }

    @Test(expected = AssertionError.class)
    public void testIncludeBeyondCapacity() {
        for (int i = 0; i < 11; i++) {
            set.include(i);
        }
    }

    @Test
    public void testExclude() {
        set.include(1);
        set.exclude(1);
        assertFalse(set.contains(1));
    }

    @Test(expected = AssertionError.class)
    public void testExcludeNonExistentElement() {
        set.exclude(1);
    }
}
```

Class StationInfo

```
public class StationInfo implements IStationInfo {
    private String name;
    private int xPos;
    private int yPos;

    public StationInfo(String name, int xPos, int yPos) {
        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;
    }
}
```

Class Network

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Network {
    final double NO_LINK = Double.MAX_VALUE;
    int numStations;
    double[][] distance;
    StationInfo[] stations;

    public Network(int capacity) {
        numStations = 0;
        distance = new double[capacity][capacity];
        stations = new StationInfo[capacity];
        for (int i = 0; i < capacity; i++) {
            for (int j = 0; j < capacity; j++) {
                if (i == j) {
                    distance[i][j] = NO_LINK;
                } else {
                    distance[i][j] = NO_LINK;
                }
            }
        }
    }

    public void readNetworkFromFile(String filename) throws
```

```
IOException {
        BufferedReader reader = new BufferedReader(new
FileReader(filename));
        String line;
        while ((line = reader.readLine()) != null) {
            String[] words = line.split(" ");
            if (words[0].equals("\"station\"")) {
                String name = words[1];
                int xPos;
                int yPos;
                try {
                    xPos = Integer.parseInt(words[2]);
                    yPos = Integer.parseInt(words[3]);
                } catch (NumberFormatException e) {
                    throw new IllegalArgumentException("Invalid
coordinates for station: " + name);
                }
                if (findStationIndex(name) != -1) {
                    throw new IllegalArgumentException("Duplicate
station: " + name);
                }
                if (numStations >= stations.length) {
                    throw new IllegalArgumentException("Exceeded
capacity of stations");
                }
                stations[numStations] = new StationInfo(name, xPos,
yPos);
                numStations++;
            } else if (words[0].equals("\"link\"")) {
                String name1 = words[1];
                String name2 = words[2];
                double dist;
                try {
                    dist = Double.parseDouble(words[3]);
                } catch (NumberFormatException e) {
                    throw new IllegalArgumentException("Invalid
distance for link: " + name1 + " - " + name2);
                }
                int index1 = findStationIndex(name1);
                int index2 = findStationIndex(name2);
                if (index1 == -1 || index2 == -1) {
                    throw new IllegalArgumentException("Invalid link:
" + name1 + " - " + name2);
                }
```

```
                if (distance[index1][index2] != NO_LINK ||
distance[index2][index1] != NO_LINK) {
                    throw new IllegalArgumentException("Duplicate
link: " + name1 + " - " + name2);
                }
                distance[index1][index2] = dist;
                distance[index2][index1] = dist;
            }
        }
        reader.close();
    }

    private int findStationIndex(String name) {
        for (int i = 0; i < numStations; i++) {
            if (stations[i].getName().equals(name)) {
                return i;
            }
        }
        return -1;
    }

    public void printStations() {
        System.out.printf("%-20s %-10s %-10s\n", "Name", "X Position",
"Y Position");
        for (int i = 0; i < numStations; i++) {
            StationInfo station = stations[i];
            System.out.printf("%-20s %-10d %-10d\n",
station.getName(), station.getxPos(), station.getyPos());
        }
    }

    public void printMatrix() {
        // Determine the maximum length of station names
        int maxNameLength = 0;
        for (int i = 0; i < numStations; i++) {
            maxNameLength = Math.max(maxNameLength,
stations[i].getName().length());
        }

        // Print column labels
        System.out.print(String.format("%" + (maxNameLength + 2) +
"s", ""));
        for (int i = 0; i < numStations; i++) {
            System.out.printf("%" + (maxNameLength + 2) + "s",
```

```
stations[i].getName());
        }
        System.out.println();

        // Print rows
        for (int i = 0; i < numStations; i++) {
            // Print row label
            System.out.printf("%-" + (maxNameLength + 2) + "s",
stations[i].getName());

            // Print row data
            for (int j = 0; j < numStations; j++) {
                if (distance[i][j] == NO_LINK) {
                    System.out.printf("%" + (maxNameLength + 2) + "s",
"NO LINK");
                } else {
                    System.out.printf("%" + (maxNameLength + 2) +
".2f", distance[i][j]);
                }
            }
            System.out.println();
        }
    }

    public ListInt breadthFirstTraversal(String start) {
        StationInfo startStation = stations[findStationIndex(start)];
        QueueInt queue = new QueueInt(numStations);
        ListInt visited = new ListInt(numStations);

        queue.addToBack(findStationIndex(startStation.getName()));

        System.out.println("Breadth First Traversal");
        System.out.println("Started: " + startStation.getName());

        while (queue.getSize() != 0) {
            StationInfo station = stations[queue.removefromFront()];

            if
(!visited.contains(findStationIndex(station.getName()))) {
                visited.append(findStationIndex(station.getName()));
                System.out.print("->" + station.getName());

                for (int i = 0; i < numStations; i++) {
                    if
```

```
(distance[findStationIndex(station.getName())][i] != NO_LINK
&& !visited.contains(i)) {
                        queue.addToBack(i);
                    }
                }
            }
        }
        System.out.println();
        return visited;
    }

    public ListInt depthFirstTraversal(String start) {
        StationInfo startStation = stations[findStationIndex(start)];
        StackInt stack = new StackInt(numStations);
        ListInt visited = new ListInt(numStations);

        stack.push(findStationIndex(startStation.getName()));

        System.out.println("Depth First Traversal");
        System.out.println("Started: " + startStation.getName());

        while (stack.getSize() != 0) {
            StationInfo station = stations[stack.pop()];

            if
(!visited.contains(findStationIndex(station.getName())))  {
                visited.append(findStationIndex(station.getName()));
                System.out.print("->" + station.getName());
                for (int i = 0; i < numStations; i++) {
                    if
(distance[findStationIndex(station.getName())][i] != NO_LINK
&& !visited.contains(i)){
                        stack.push(i);
                    }
                }
            }
        }
        System.out.println();
        return visited;
    }

    public void dijkstraShortestPath(String start, String end) {
        // Find the index of the start and end stations
        int startIdx = findStationIndex(start);
```

```
        int endIdx = findStationIndex(end);

        // Initialize the distances array with NO_LINK (infinity)
        double[] distances = new double[numStations];
        for (int i = 0; i < numStations; i++) {
            distances[i] = NO_LINK;
        }
        // Set the distance from the start station to itself as 0
        distances[startIdx] = 0;

        // Initialize the previous array to keep track of the path
        int[] previous = new int[numStations];
        for (int i = 0; i < numStations; i++) {
            previous[i] = -1;
        }

        // Initialize the visited array to keep track of which
stations have been visited
        boolean[] visited = new boolean[numStations];
        for (int i = 0; i < numStations; i++) {
            visited[i] = false;
        }

        // Initialize the open array to keep track of which stations
are open for visiting
        boolean[] open = new boolean[numStations];
        for (int i = 0; i < numStations; i++) {
            open[i] = false;
        }

        // Mark the start station as open
        open[startIdx] = true;

        // Initialize the iterations counter
        int iterations = 0;

        // While the end station has not been visited
        while (!visited[endIdx]) {
            // Find the open station with the smallest distance
            double minDist = NO_LINK;
            int minIdx = -1;
            for (int i = 0; i < numStations; i++) {
                if (open[i] && distances[i] < minDist) {
                    minDist = distances[i];
```

```
                    minIdx = i;
                }
            }
            // If no such station exists, break the loop
            if (minIdx == -1) {
                break;
            }
            // Mark the station as visited and not open
            open[minIdx] = false;
            visited[minIdx] = true;

            // If the end station has been visited, break the loop
            if (minIdx == endIdx) {
                break;
            }
            // Otherwise, update the distances to the neighboring
stations
            else {
                for (int i = 0; i < numStations; i++) {
                    if (distance[minIdx][i] != NO_LINK && !visited[i])
{
                        double newDist = distances[minIdx] +
distance[minIdx][i];
                        if (newDist < distances[i]) {
                            distances[i] = newDist;
                            previous[i] = minIdx;
                            open[i] = true;
                        }
                    }
                }
            }
            // Increment the iterations counter
            iterations++;
        }

        // If no path was found, print a message
        if (distances[endIdx] == NO_LINK) {
            System.out.println("No path found from " + start + " to "
+ end);
        }
        // Otherwise, print the shortest path and its distance
        else {
            System.out.println("Dijkstra Shortest Path");
            System.out.println("Shortest path from " + start + " to "
```

```
+ end + ":");
            int idx = endIdx;
            StackInt stack = new StackInt(numStations);
            // Push the stations on the shortest path onto a stack
            while (idx != -1) {
                stack.push(idx);
                idx = previous[idx];
            }
            // Pop the stations from the stack and print them
            while (stack.getSize() != 0) {
                System.out.print(stations[stack.pop()].getName());
                if (stack.getSize() != 0) {
                    System.out.print("->");
                }
            }
            System.out.println();
            System.out.println("Distance: " + distances[endIdx]);
        }
        // Print the number of iterations
        System.out.println("Number of iterations: " + iterations);
    }
}
```