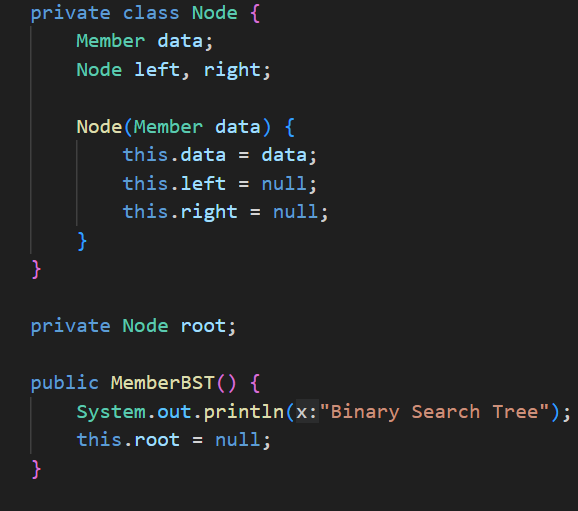
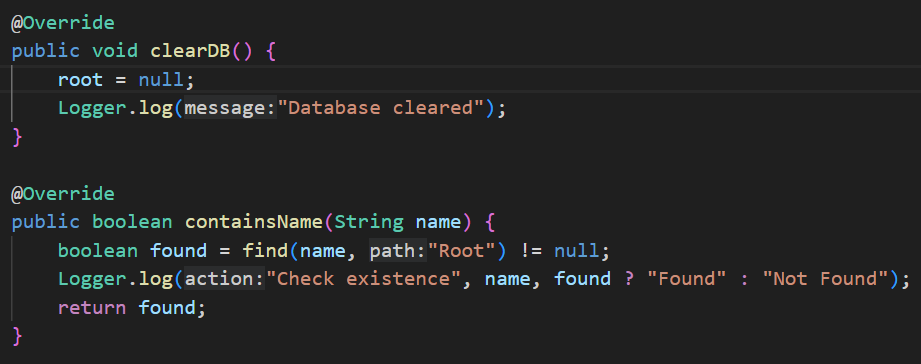
Coursework2

Part A：

# Task1:

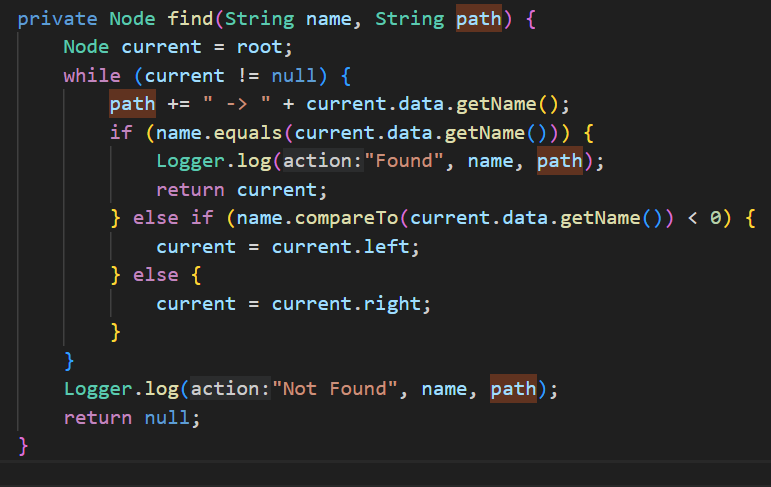
1. 

First of all, I define a Node which include Member type data to stores the actual data of the node, which is an object of the Member class. Then Node left, right represent references to the left child and the right child of the current node

1. 

clearDB():Resets the tree by removing all nodes, root = null; directly sets the tree's root to null, which discards the entire tree structure.

containsName(String name): Checks if a member with the specified name exists in the tree. `find(name, "Root") != null;` utilizes the `find` method to check for the existence of a node.

1. 

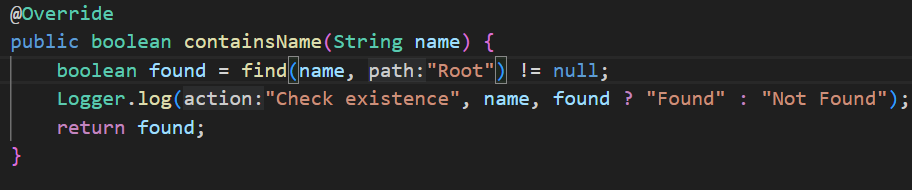
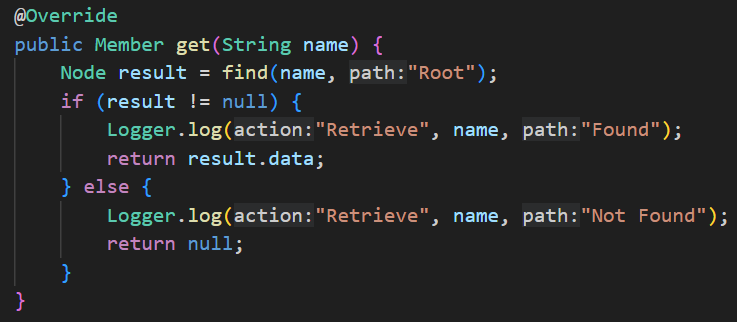
The purpose is to locate a Node that contains a specific name.

Traverse the tree, starting from the root.

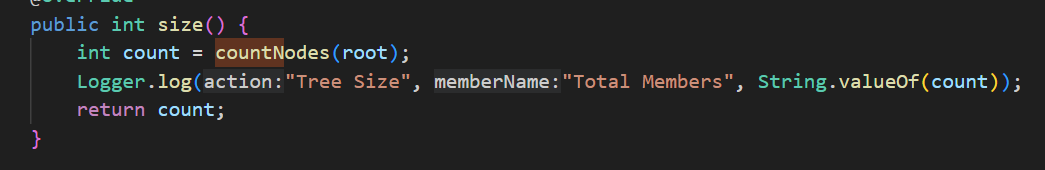
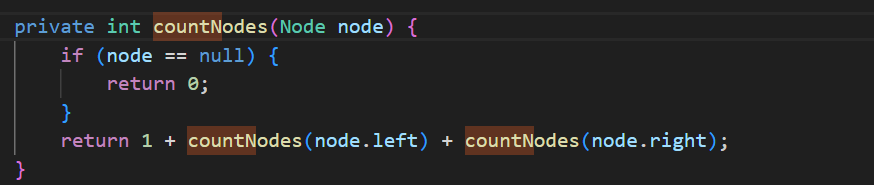
If current becomes null, search has reached a leaf node's null child, and the member is not found in the tree.

path += " -> " + current.data.getName(); updates the path string by appending the current node's member name.

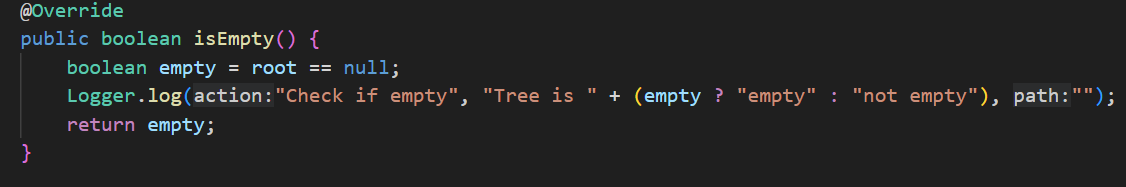
If the search name is less than the current node's name (based on lexicographical comparison), the search moves to the left child of the current node. Otherwise, it moves to the right child.

1.  Use the find method to check whether the member with the specified name exists in the tree. And display the result of the lookup operation.
2. 

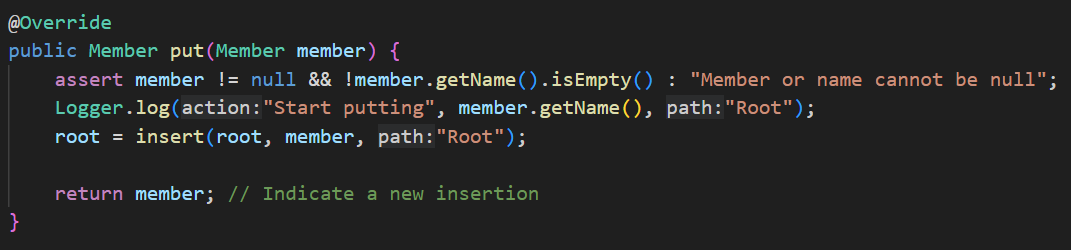
`get(String name)`:Retrieves the member with the specified name. use `find` for direct retrieval. If the name exist display the log and return the name, otherwise return null. Direct access to member data if present, with null checks ensuring safety against non-existent entries.

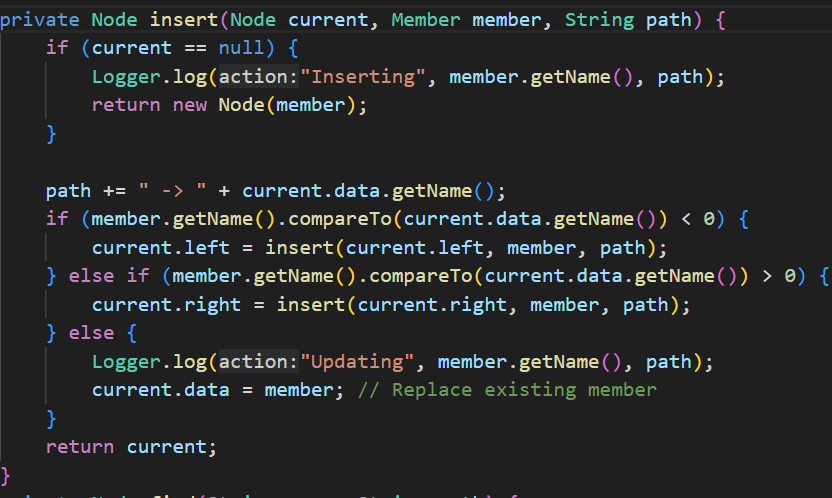
1.  

Returns the total number of members, counts nodes recursively from the root. Because count nodes from the root so need add 1 at first,

1. isEmpty():

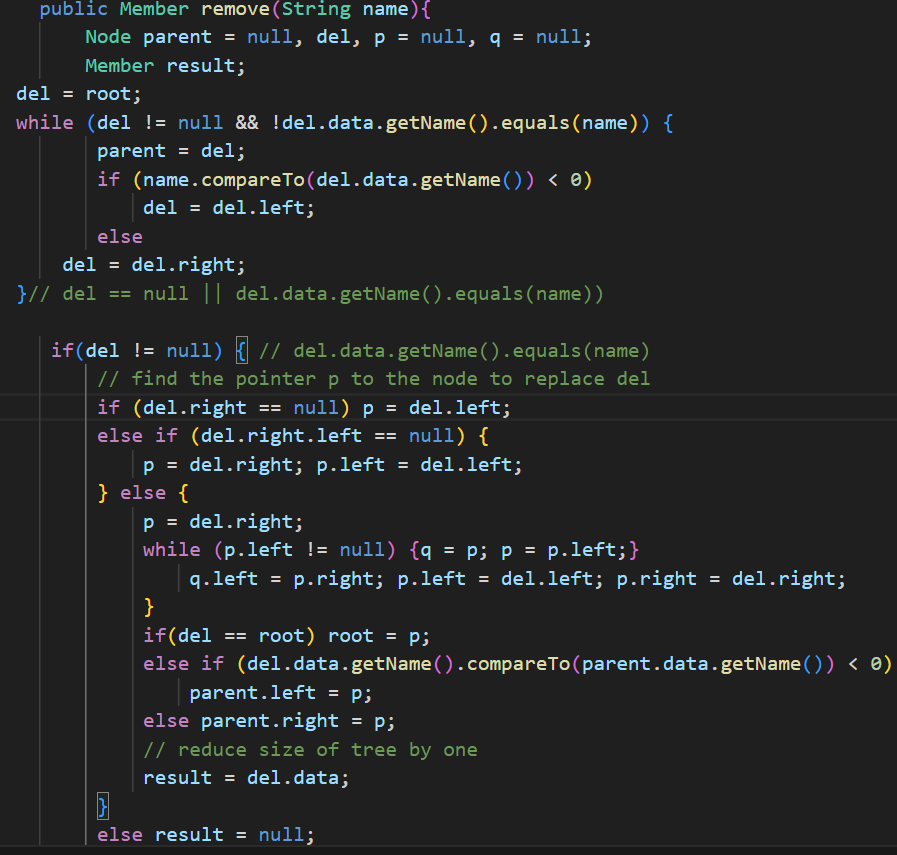
Checks if the tree is empty

1. 



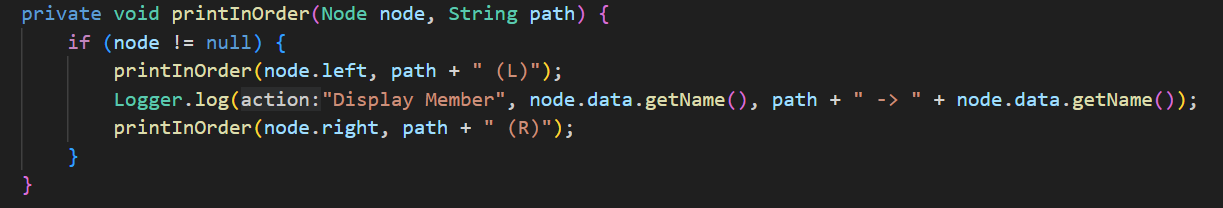
If member or name is not empty it will inserts or updates member.

path += " -> " + current.data.getName(); updates the path string by appending the current node's member name. If member.getName().compareto (current.data.getName()) < 0 is true, the name of the member to be inserted is smaller than the name of the current node, so the member should be inserted into the left subtree of the current node. current.left = insert(current.left, member, path); This line of code inserts the member recursively into the left subtree and updates the left subnode. Finally, If the name is the same, update the current node member and record the update operation

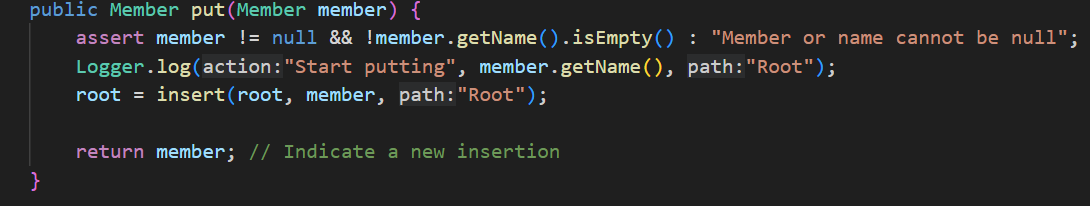
1. 

First initialization variable, then searches for the node (del) to remove by comparing the name. It traverses the tree, moving left or right based on the comparison, Next, If del is not null, it means the node to be deleted has been found. If del.right is null, simply lift the left child (p = del.left). If the right child of del has no left child, the right child replaces del, and the left child of del becomes the left child of p. If the right child has a left child, find the minimum node (p) from the right subtree, which will replace del. This is done by moving left until no more left children are found. q is used to keep track of the parent of p. If del is the root, then p becomes the new root. Otherwise, depending on whether del was a left or right child of its parent, p is connected appropriately. Store the data from the deleted node in result to return.

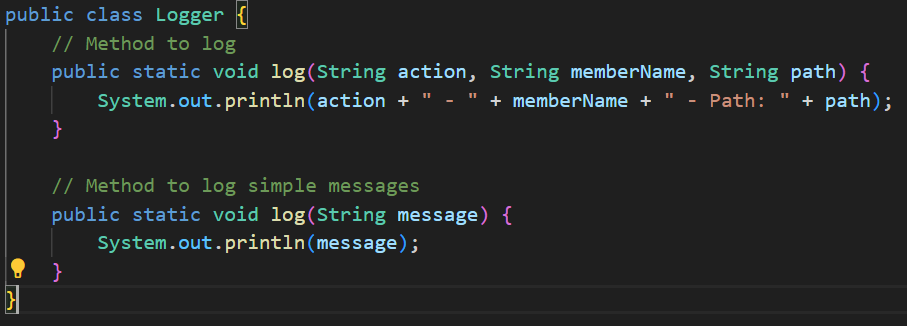
1. 



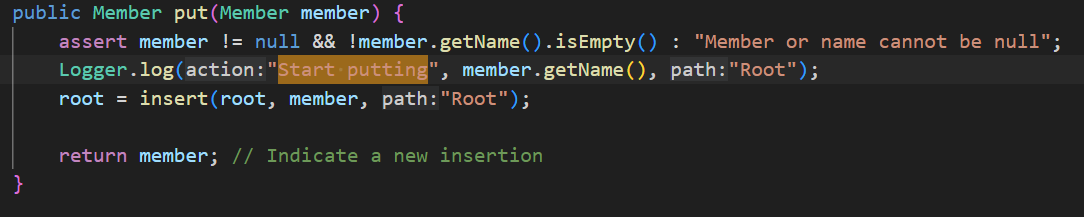
Printinorder method perform an in-order traversal of tree. (node ! = null): Ensure non-empty nodes need to be processed. Recursively traverses the left subtree first, with the path marked "(L)" indicating a move to the left. And display the exact position of each member in the tree. Recursively traverses the right subtree, with the path marked "(R)" indicating a move to the right, which handles all the nodes that have keys greater than the current node’s key.

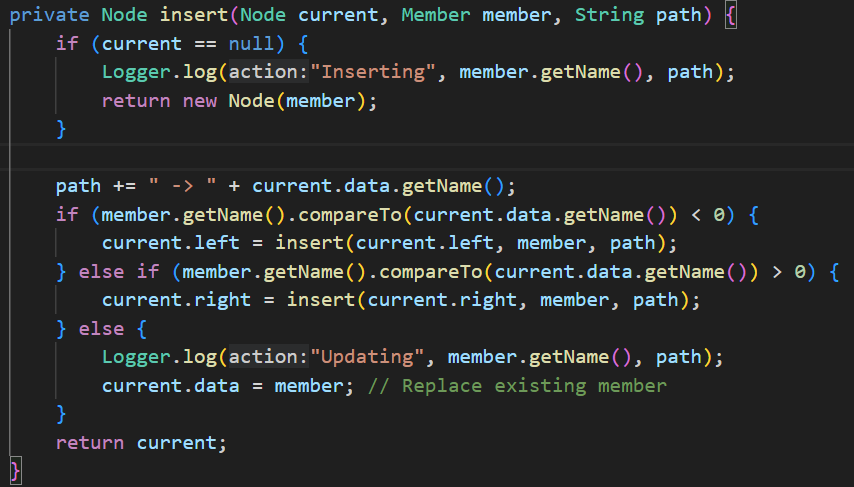
Task2  The assert ensure that the input member and its name are not empty

# Task3：



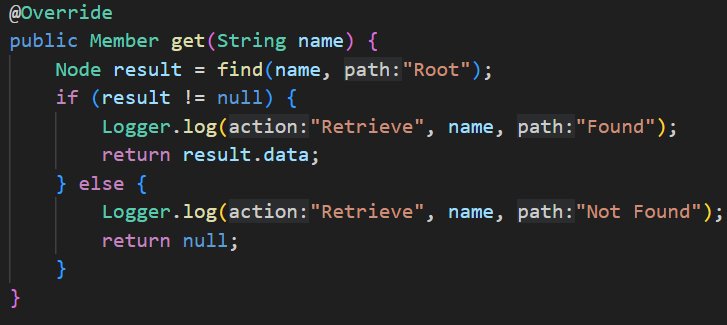
This is a log method to display what uesr done in BST.





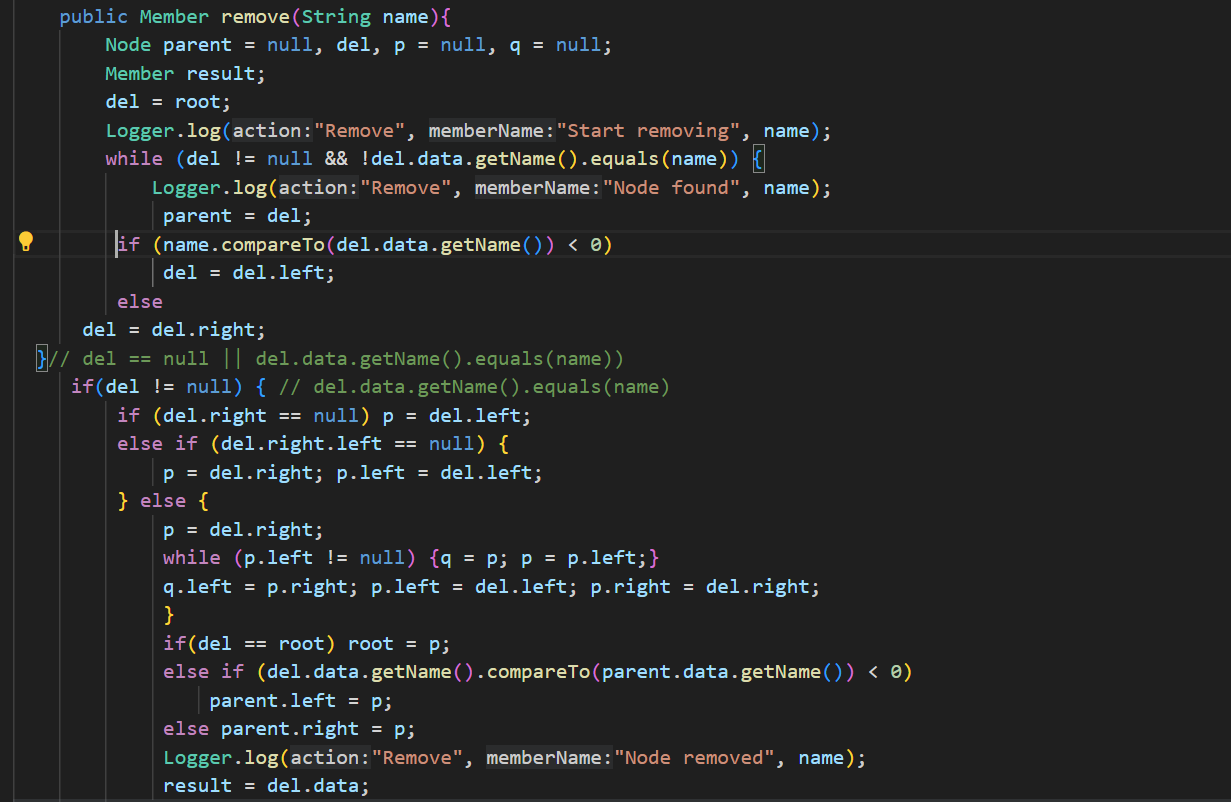
This is log in put method: Logger.log("Inserting", member.getName(), path); logs the actual insertion of a new member at a specific path in the tree.

Logger.log("Updating", member.getName(), path); logs the updating of an existing member’s information.



Logger.log("Retrieve", name, "Found"); logs the successful retrieval of a member’s information.

Logger.log("Retrieve", name, "Not Found"); logs when a member’s information is not found during retrieval.

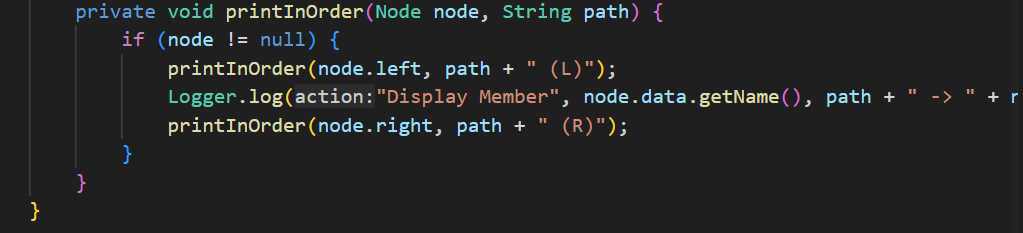


Logger.log("Remove", "Start removing", name); logs the initiation of the removal process for a member.

Logger.log("Remove", "Node found", name); logs each step of searching for the node to be removed.

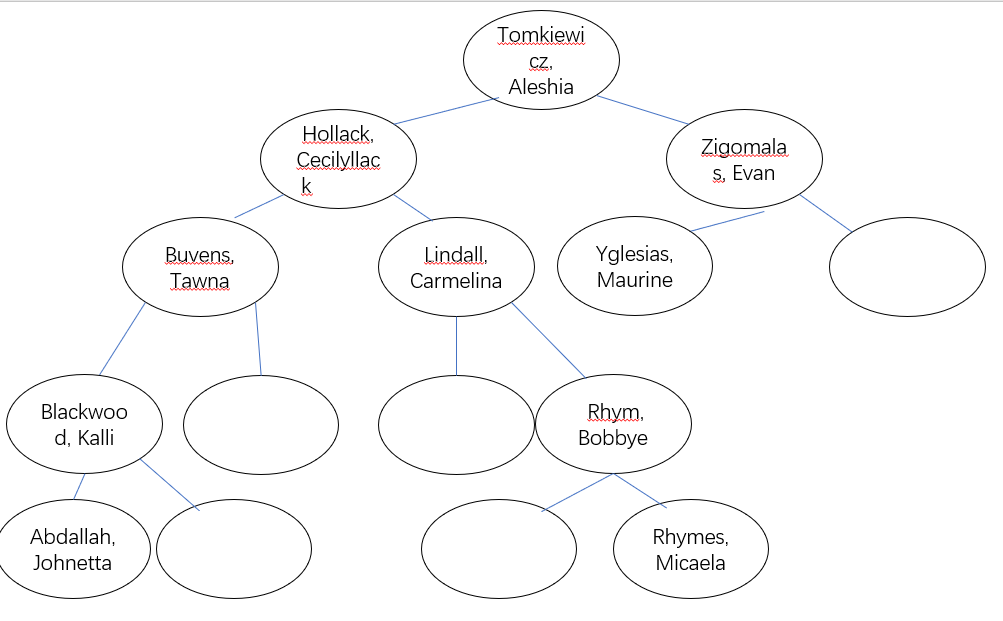
Logger.log("Remove", "Node removed", name); logs the successful removal of a member.

Logger.log("Remove", "Node not found", name); logs when a member to be removed is not found in the database.

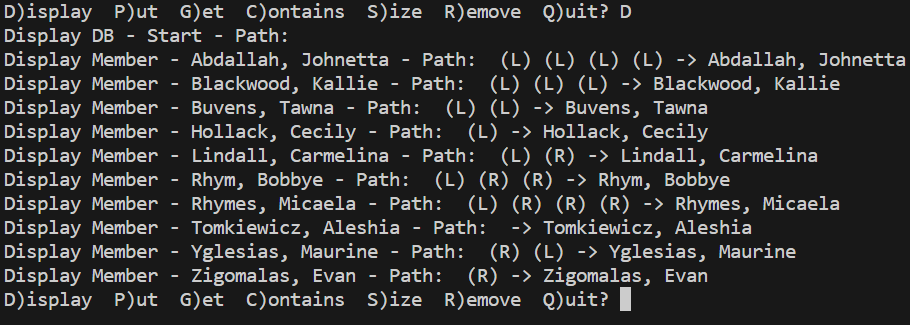


Logger.log("Display Member", node.data.getName(), path + " -> " + node.data.getName()); logs each member’s information during the in-order traversal of the database.

# Task 4:



The figure is designed to test my BST. White space means the node is empty.



private void printInOrder(Node node, String path) {

        if (node != null) {

            printInOrder(node.left, path + " (L)");

            Logger.log("Display Member", node.data.getName(), path + " -> " + node.data.getName());

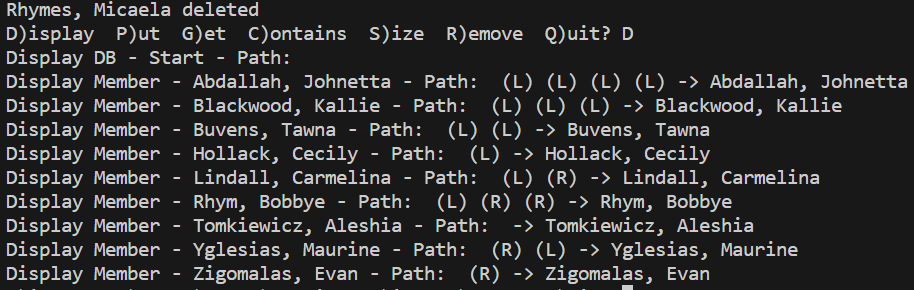
            printInOrder(node.right, path + " (R)");

        }

    }

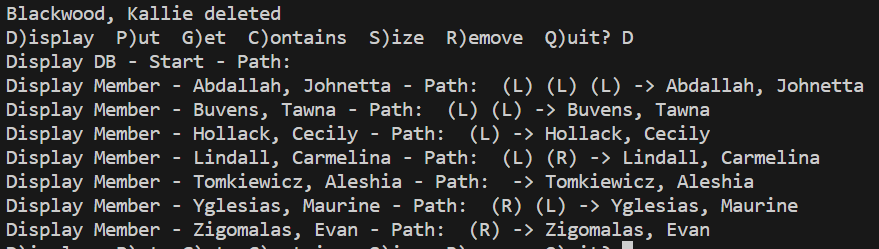
Because of the printinorder which I explain in task1 recursively traverses the left subtree first, with the path marked "(L)" indicating a move to the left. And display the exact position of each member in the tree. Recursively traverses the right subtree, with the path marked "(R)" indicating a move to the right.

## Deleting leaf node



I delete Rhymes, Micaela and obviously it is successful.

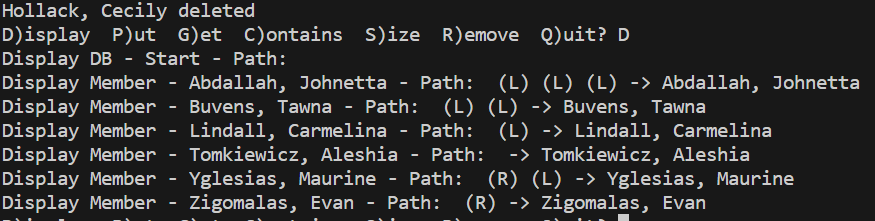
## deleting a node with one descendant



As figure shown Blackwood, Kallie has a descendant Abdallah,Johnetta

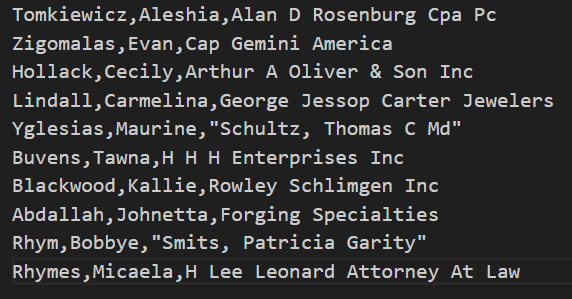
Which still exist after delete Blackwood, Kallie.

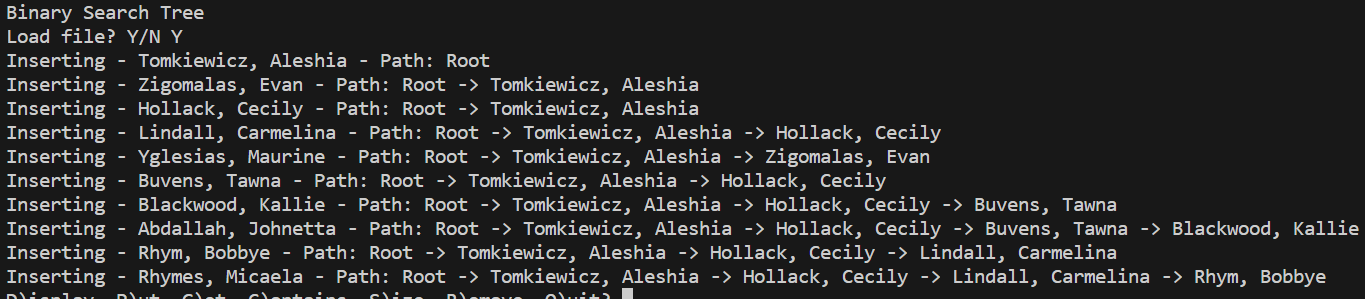
## deleting a node with two descendants

 I delete Hollack, Cecily and the set right sub tree as new node.

# Task 5;

## Test data:



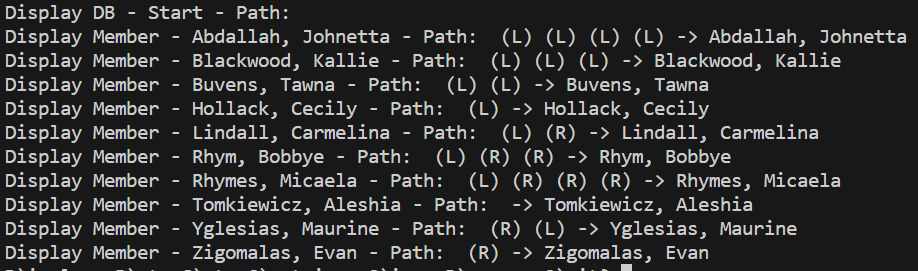


The data insert BST successfully.

## Test plan:

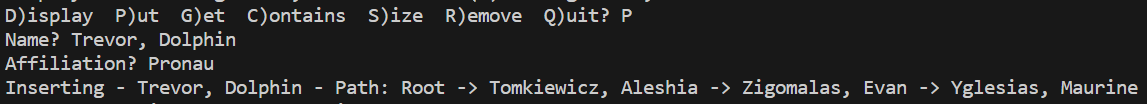
*  put: Insert various members into the BST, Members should be inserted correctly and the tree structure should reflect the binary search property.
*  remove: It should have many case tests for remove such as deleting a leaf node, deleting a node with one descendant, deleting a node with two descendants and deleting the root. However, I already test them in Task 4. So, I will simply test remove one random member from the tree.
*  get: It should get the member based on the name if it was found in the tree.
*  containsName: Traverse the tree based on the name. It will return a Boolean type value to show whether the member exist in the tree.
*  size: It will return the size of the tree.
*  isEmpty: Return the Boolean type value to show whether the tree is empty.
*  clearDB: Clean the tree and set the size to 0.
*  displayDB: Display the tree utilizing String.compareTo to compare strings lexicographically.

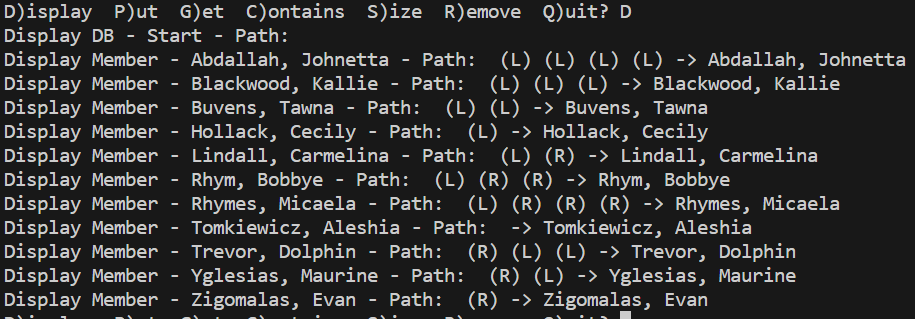
### Display



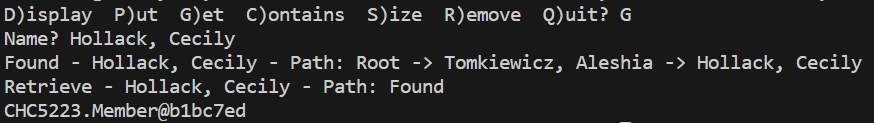
It will display the data as tree type. For example, the output `Display Member - Aschoff, Pedro - Path: (L) -> Aschoff, Pedro` shows that "Aschoff, Pedro" is a left child of the root, and no further left children exist before him. This display format provides a clear, visual representation of the tree's structure and the relative positions of its nodes, enhancing both debugging and understanding of the data structure’s layout.

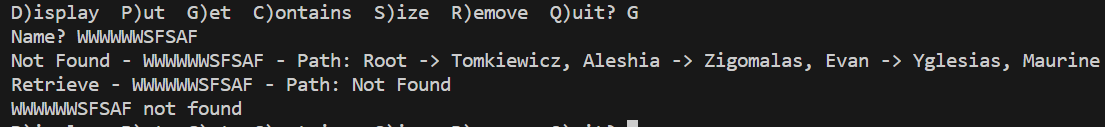
### Put



It will show the insert path and Node has inserted. The Trevor dolphin put successfully as the test plan expect. 

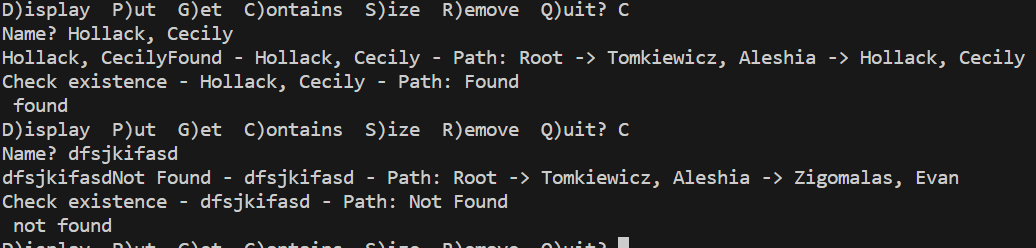
### Get





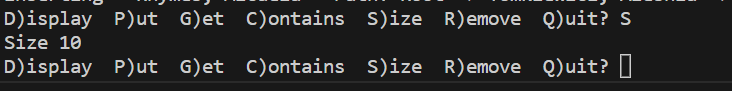
If found, it will return the member and logging the retrieved result, else return not found and path is shown.

### Contains



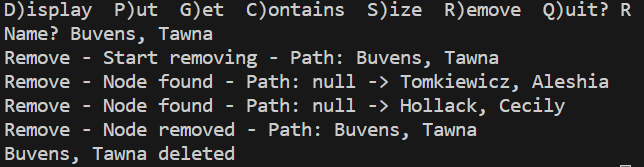
If found the node, it will return the name “found”, else return not found and the path from start to the end.

### Size

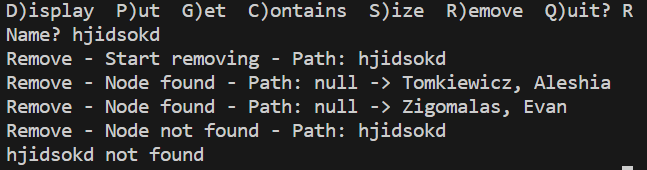


Show the size correct.

### Remove



It will show the process path first go to Tomkiewicz, then go to Hollack ndoe, finally show the node has been remove.



If not found it will show not found, and more case tests are shown in Test 4.

# Task 6:

### Put:

Time Efficiency: The average case time complexity for insertion is O(log n). However, in the worst case (unbalanced tree), it can be O(n).

Space Efficiency: The space complexity for insertion is O(log n) due to the recursive call stack.

### containsName and get:

Time Efficiency: Both search operations have an average case time complexity of O(log n). In the worst case, they can take O(n) time.

Space Efficiency: The space complexity for search is O(log n) due to the recursive call stack.

### remove

Time Efficiency: The average case time complexity for removal is O(log n). In the worst case, it can be O(n).

Space Efficiency: The space complexity for removal is O(log n) due to the recursive call stack.

### Size:

Time Efficiency: Calculating the size of the tree requires visiting all nodes, so the time complexity is O(n).

Space Efficiency: No additional space is used beyond the recursive call stack.

### isEmpty:

Time Efficiency: Checking if the tree is empty takes constant time, so it is O(1).

Space Efficiency: No additional space is used beyond the function call.

### BST

Time Efficiency (Average Case): The average time complexity for most BST operations is O(log n), where n is the number of nodes in the tree. However, in the worst case (unbalanced tree), the height of the BST can be O(n), resulting in linear time complexity for operations. For example, if the tree all nodes have only one child, the BST degenerates into a linked list.

Space Efficiency: The space complexity of a BST is O(n),

Part B

# Task1:

I create eight nodes:

"station" NodeA 50 50

"station" NodeB 200 50

"station" NodeC 50 200

"station" NodeD 200 200

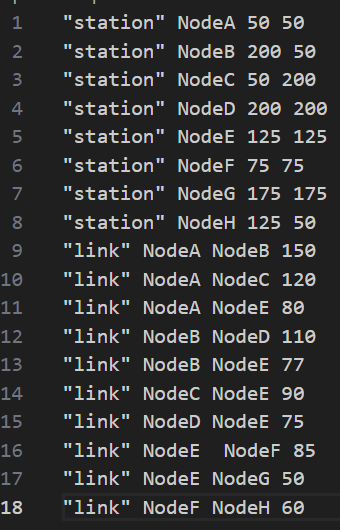
"station" NodeE 125 125

"station" NodeF 75 75

"station" NodeG 175 175

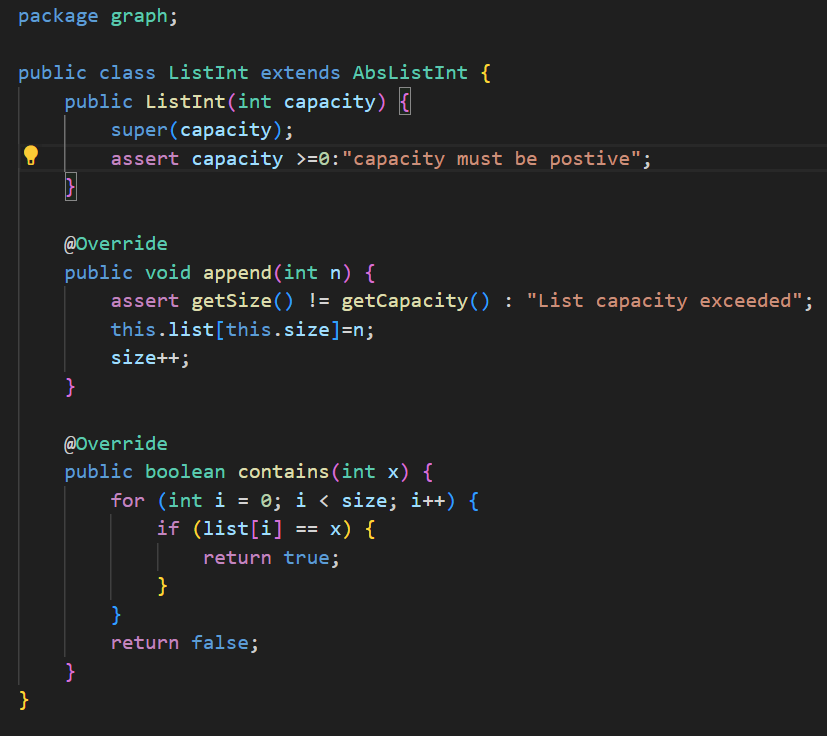
"station" NodeH 125 50

# Task2:



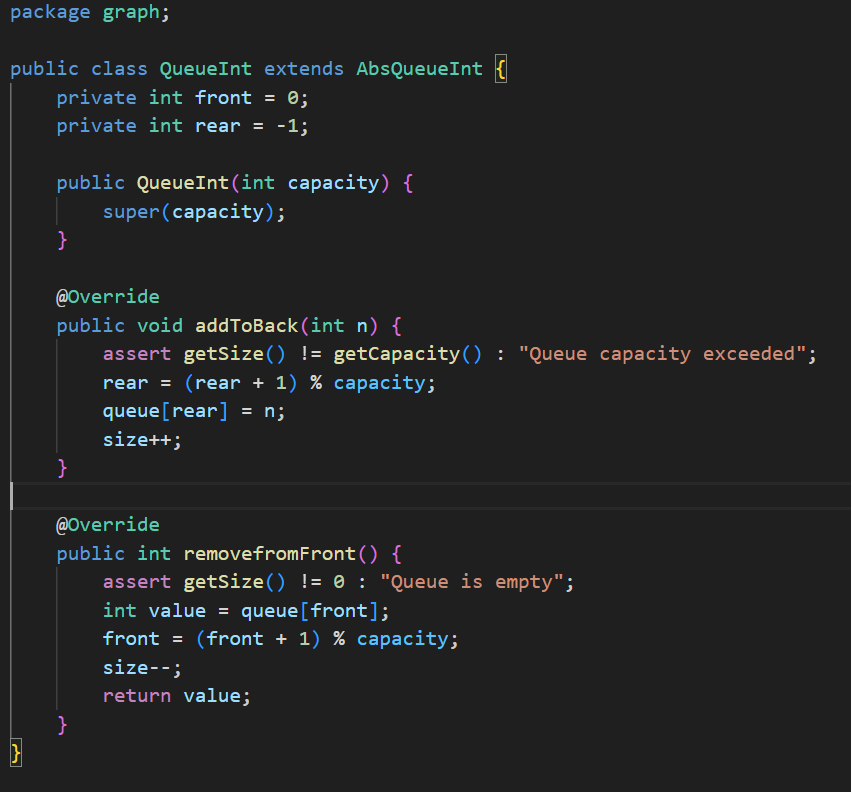
# Task3:

### ListInt:



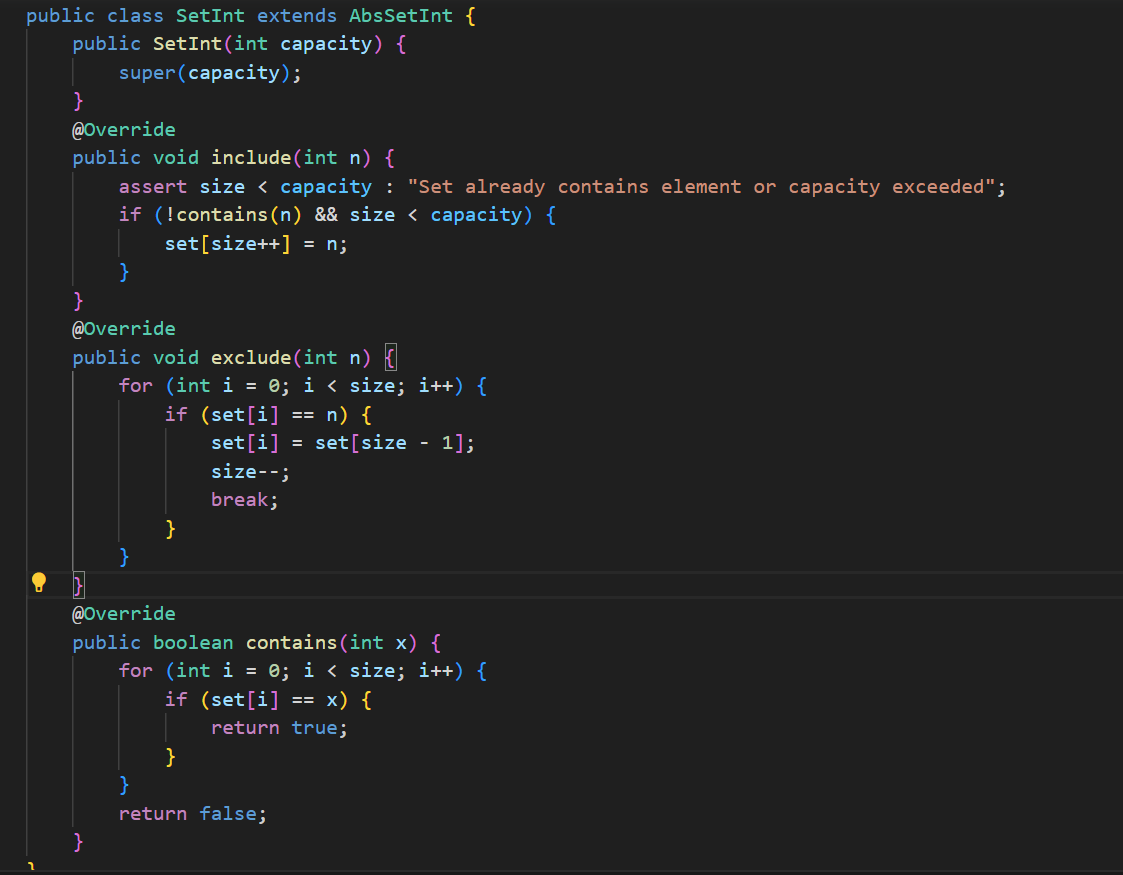
The ListInt class is designed to manage a collection of integers in a list format, allowing for dynamic insertion at the end.The append method adds an integer to the end of the list, ensuring that the list’s capacity is not exceeded. Contains method checks if a given integer is present in the list by iterating through the elements.

### QueueInt:



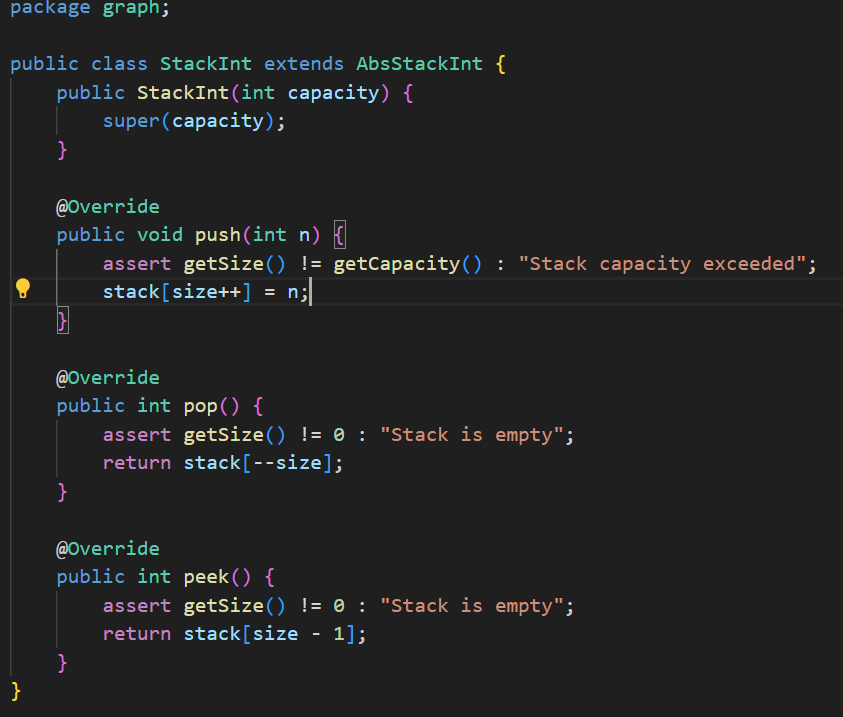
The QueueInt class implements a queue data structure using an array, supporting FIFO (First-In-First-Out) operations. AddToBack method enqueues an integer at the rear of the queue, wrapping around the array if necessary. RemovefromFront method dequeues an integer from the front of the queue, also wrapping around the array.

### SetInt:



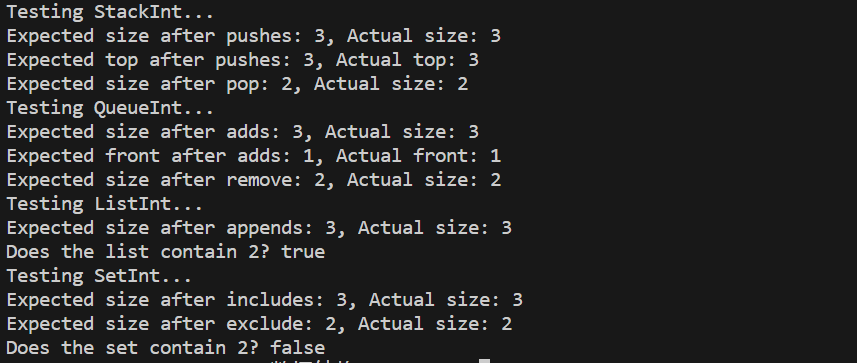
The SetInt class represents a set of unique integers, ensuring no duplicates are stored.The include method adds an integer to the set if it’s not already present and if there’s space available.The exclude method removes an integer from the set by replacing it with the last element and reducing the size.

### StackInt:

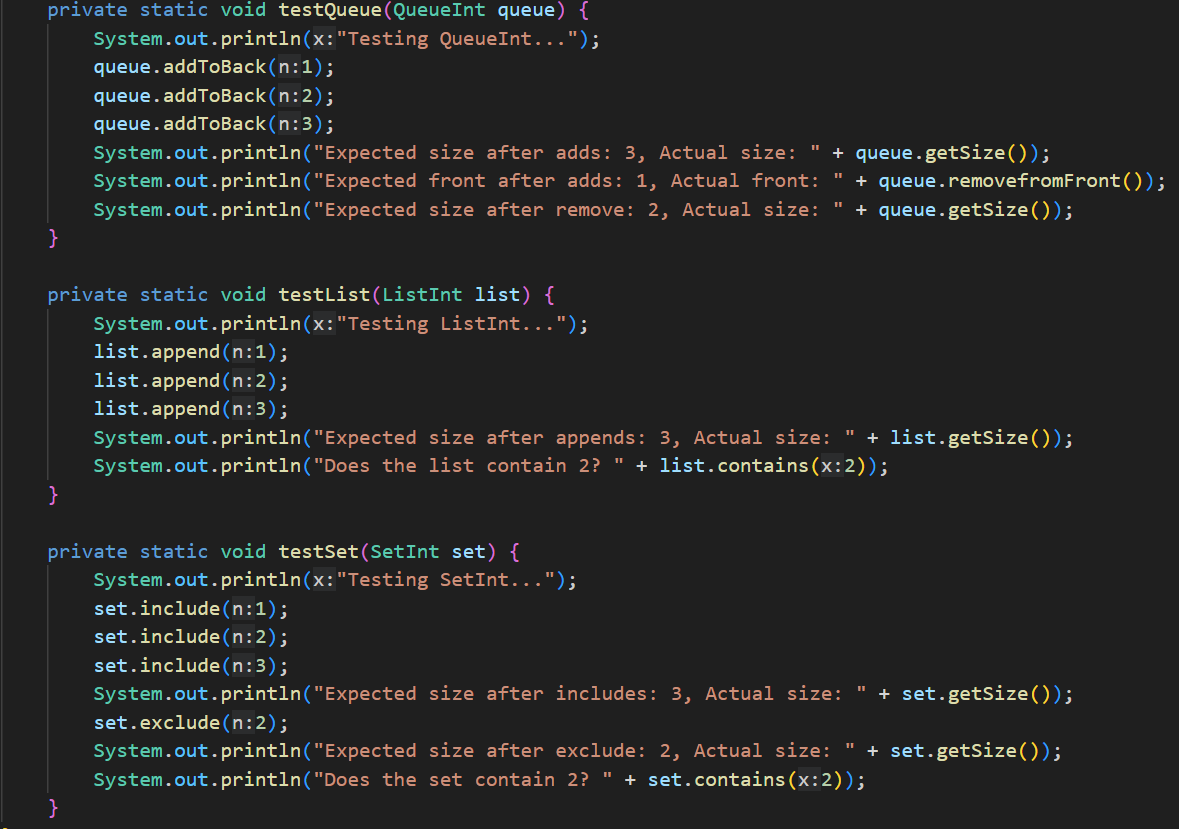


The StackInt class implements a stack data structure, allowing LIFO (Last-In-First-Out) operations. push method pushes an integer onto the top of the stack, checking for capacity limits.The pop method pops the top integer off the stack and returns it.The peek method returns the top integer without removing it from the stack.

# Task5







## Test Plan

Put integer in these structure, test every function working as expect.

### Test Data:

Push elements: 1, 2, 3

### Testing StackInt:

Push elements, Pop one element, Actual Results are as expected.

Size after pushes: 3

Top after pushes: 3

Size after pop: 2

### Testing QueueInt:

Add elements, Remove one element, Actual Results are as expected.

Size after adds: 3

Front after adds: 1

Size after remove: 2

### Testing ListInt:

Append elements: 1, 2, 3, Check if list contains 2. Actual Results are as expected.

Size after appends: 3

List contains 2

### Testing SetInt:

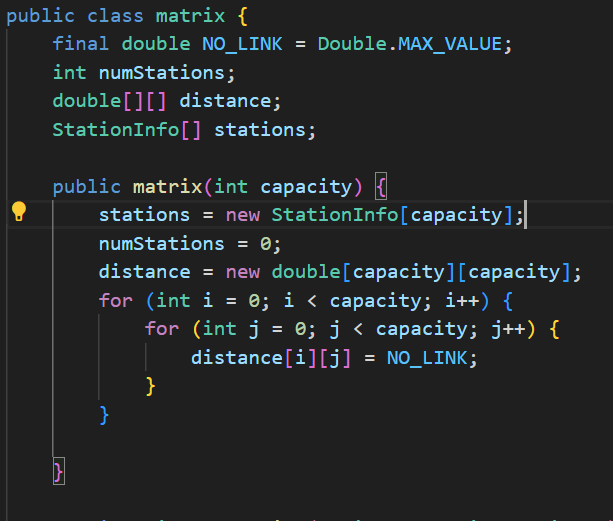
Include elements: 1, 2, 3, Exclude element: 2, Check if set contains 2 Actual Results are as expected.

Size after includes: 3

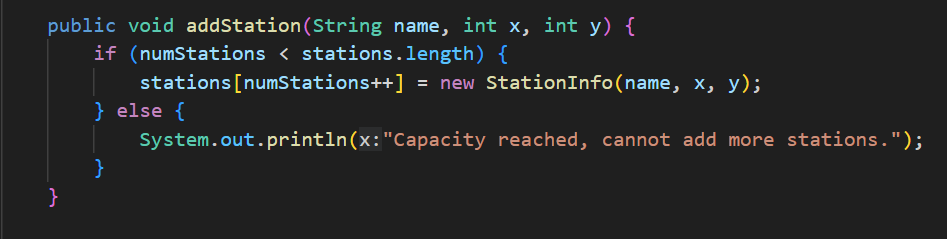
Size after exclude: 2

Set does not contain 2

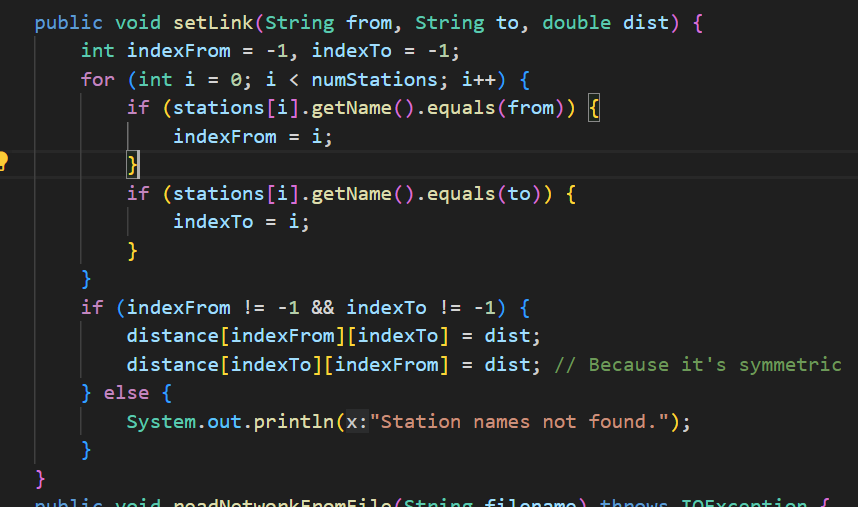
# Task9:



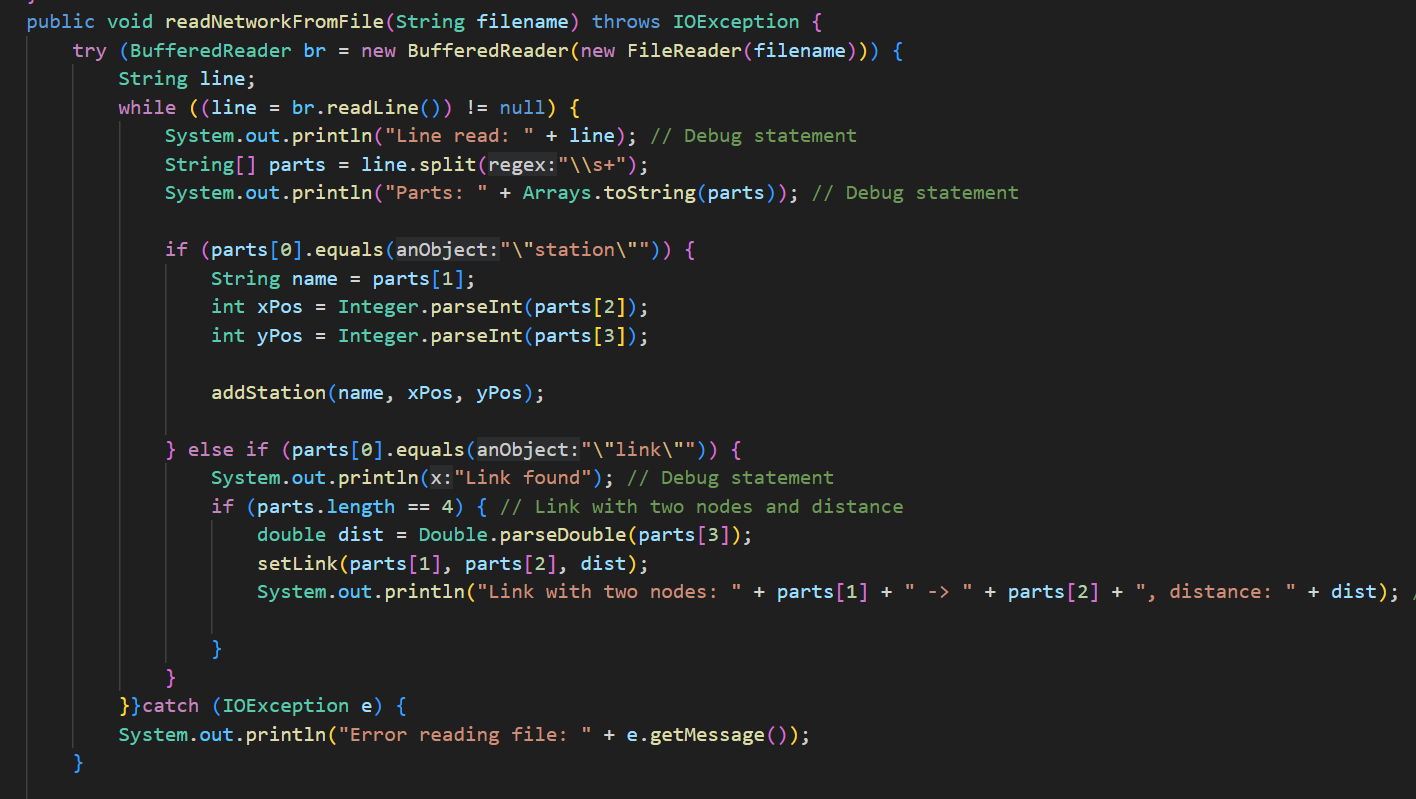
Initializes the stations array and the distance matrix. Sets the initial number of stations (numStations) to zero. The NO\_LINK constant represents an absence of a link. All codes are perform demands,



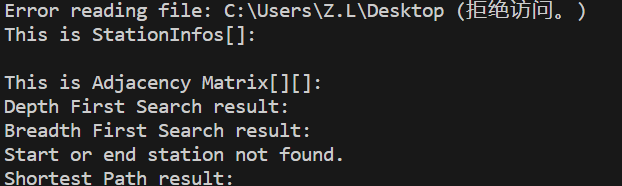
Adds a new station to the network. Checks if the current number of stations is less than the capacity. If the capacity is reached, it will print a error message.



Sets the distance between two stations. Searches for the indices of the specified station names (from and to). If either station name is not found, you print an error message.



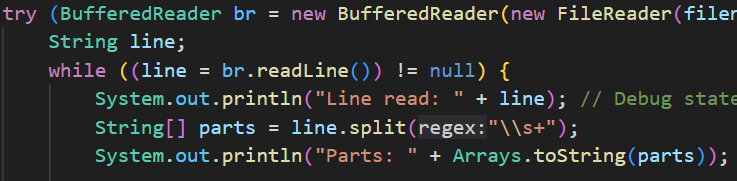
Successfully handle cases where the input file format is incorrect



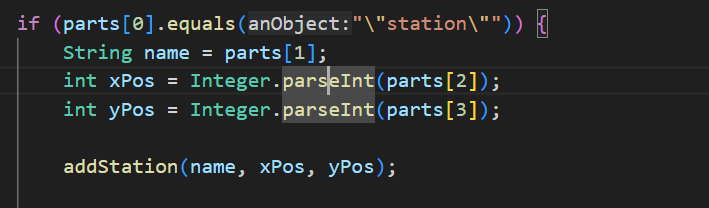
This line prints the content of the current line read from the file.

String[] parts = line.split("\\s+");

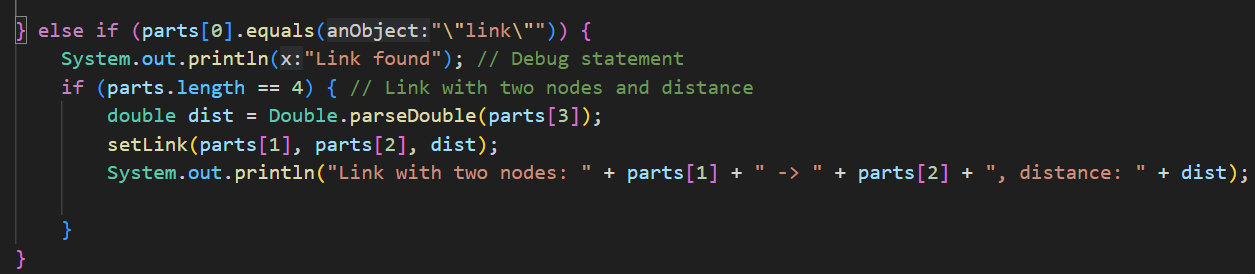
This splits the current line into an array of strings (parts). The \\s+ regular expression matches one or more whitespace characters.



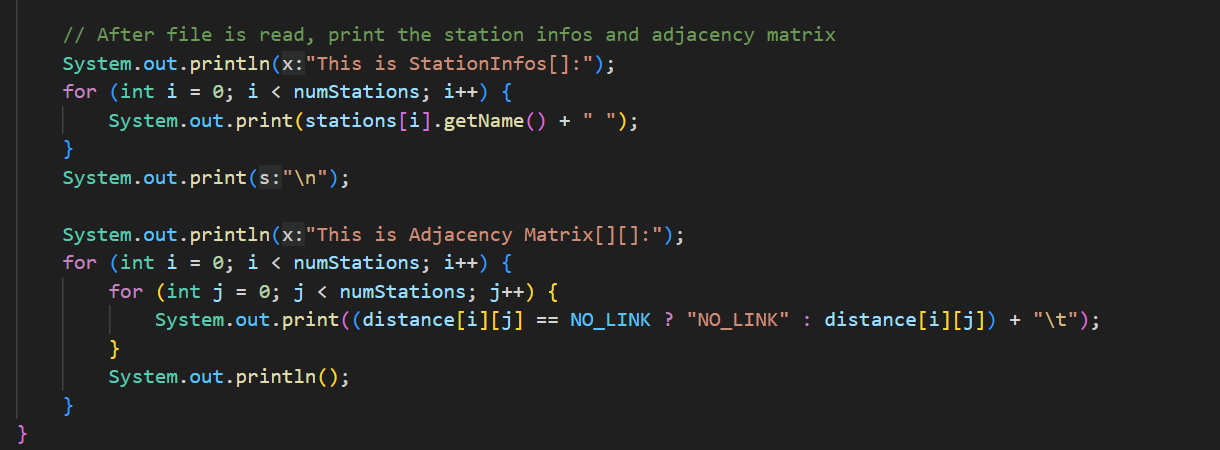
If begin with “station” it will converts the element of parts to an integer. Calls the addStation method with the station name (name) and the parsed x and y coordinates, which will adds a new station to the network

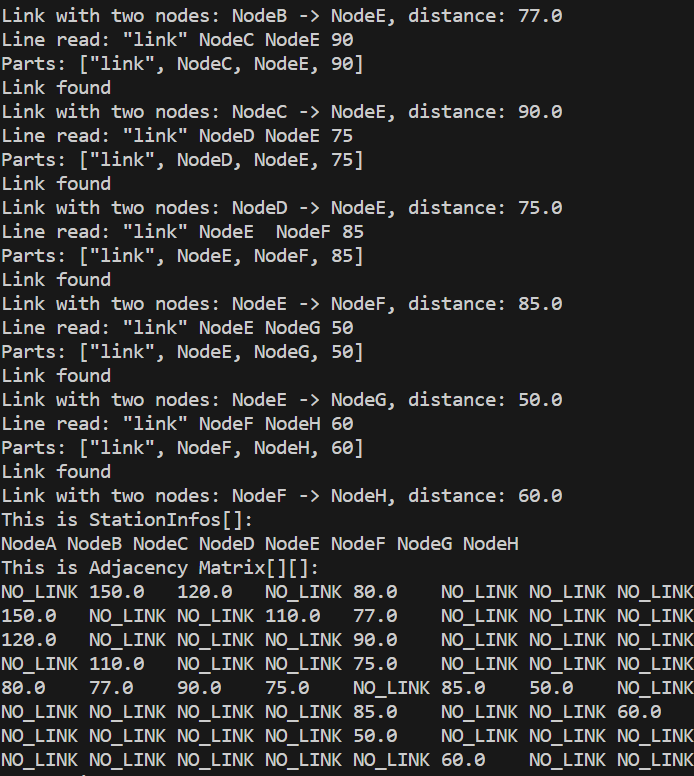


Checks if the first element of parts is equal to "link" and if there are exactly four elements which indicates a link definition with two nodes and a distance. Then, call the setLink method to set the link between the specified nodes in the adjacency matrix.

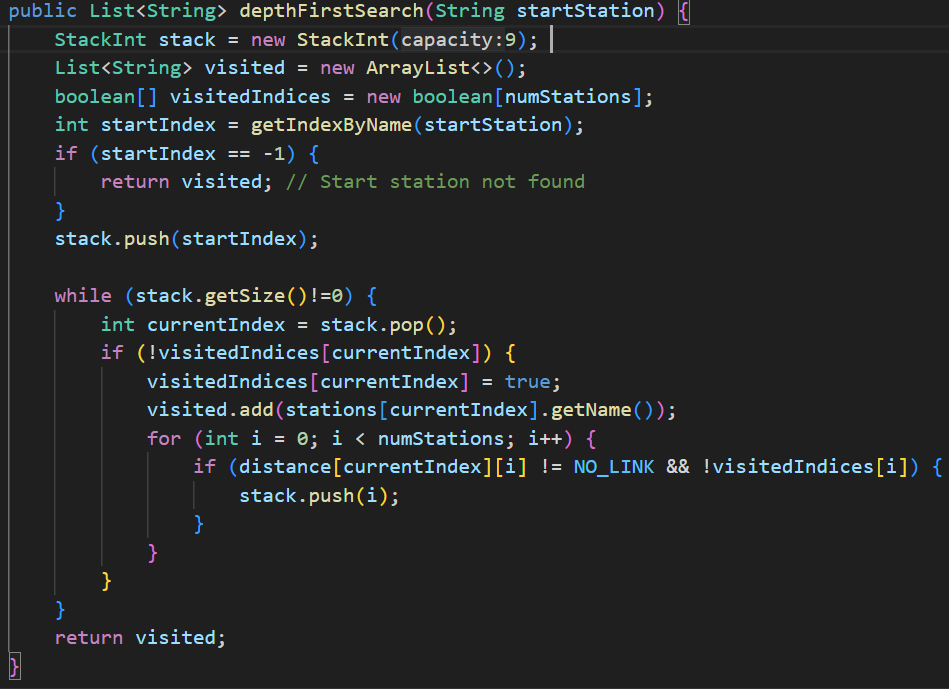


Iterate over all pairs of stations and check if the distance between stations i and j is equal to NO\_LINK, it prints "NO\_LINK". Otherwise, it prints the actual distance value.

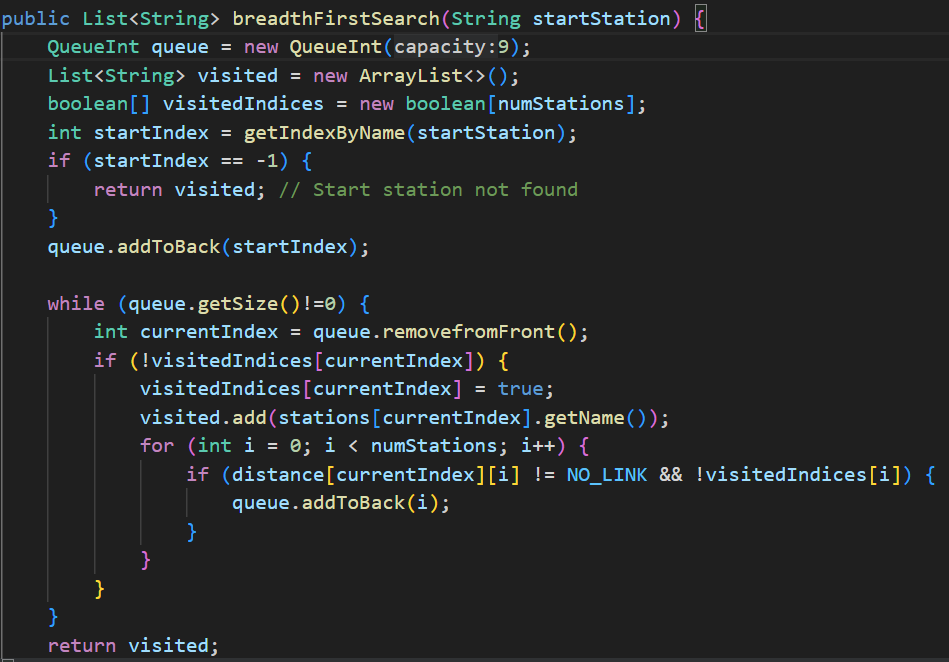




# Task 11:

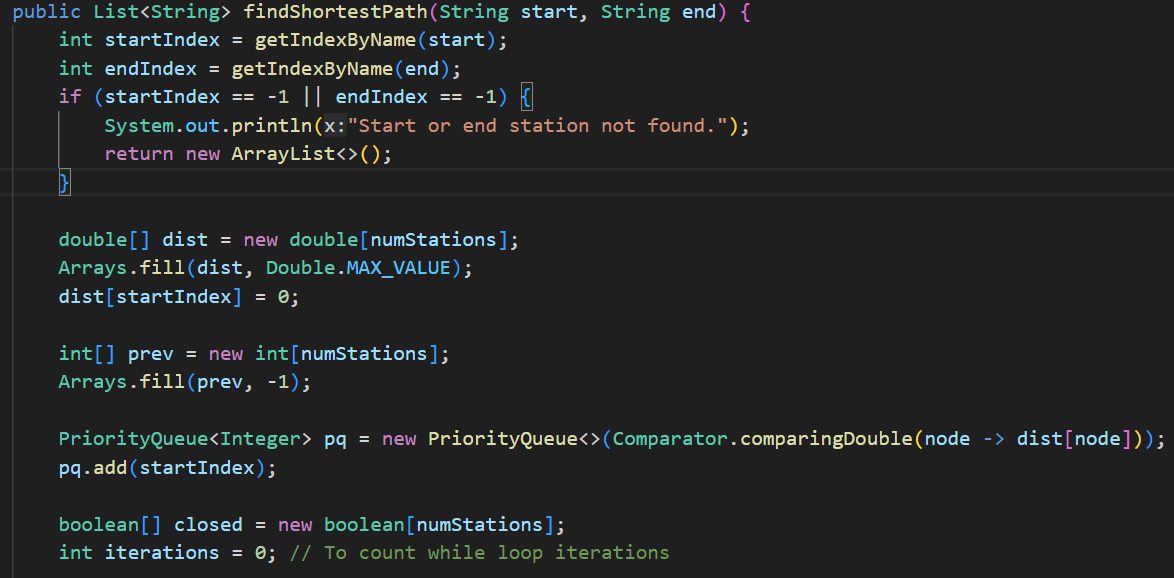


Initializes a stack to manage the traversal order. visited keeps track of the stations visited in the order they were visited. visitedIndices is a boolean array to mark stations as visited to avoid revisiting them. Then, Retrieves the index of the start station. If the station is not found, returns an empty list. stack.push(startIndex);pushes the index of the start station onto the stack. Then, If the current station hasn’t been visited, marks it as visited and records its name. Finally, iterates through all stations to find unvisited neighbors connected to the current station and pushes them onto the stack.

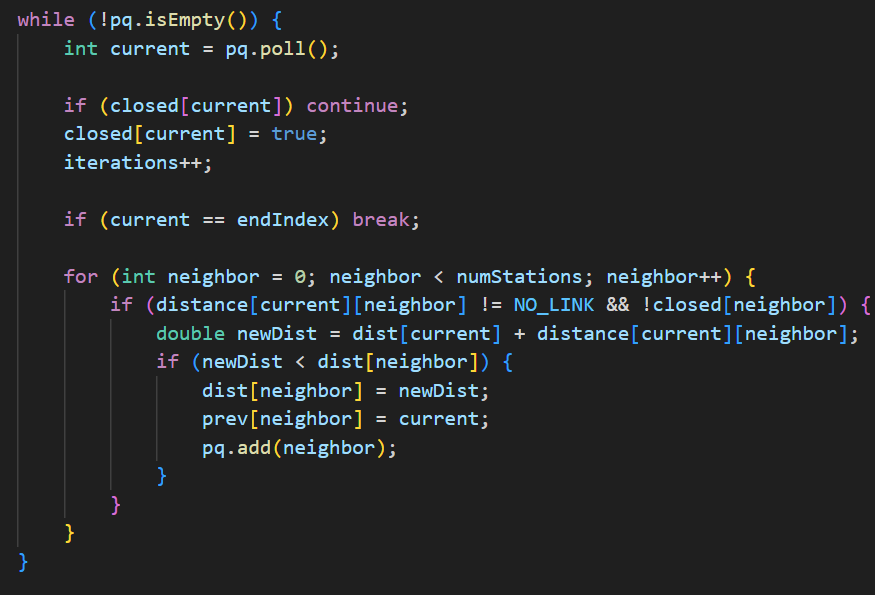


First, initializes a queue to manage the traversal order. Keeps track of visited stations and marks stations in list. Then, Retrieves the start station index and enqueues it until the queue is empty. Removes the front element. Then, Marks the current station as visited and records its name. Finally, iterates through all stations to find unvisited neighbors connected to the current station and enqueues them and returns the list of visited stations

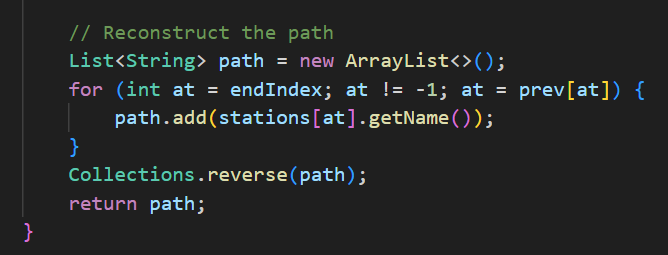
# Task 12

Initialization: 

startIndex and endIndex represent the index of the start and end stations. Check if the provided start and end station names exist. If not, we return an empty list. Then, dist is an array to store the shortest distanced, which initialize positive infinity except for the start station. The prev array backtrack from the end station to the start node. It stores the previous node on the shortest path for each node. We initialize it with -1. pq is a priority queue to keep track of the stations to visit next, ordered by their current distance.closed is a boolean array to mark whether a station has been visited.



While the priority queue is not empty, it will extract the station with the minimum distance from the priority queue. If it’s already closed, skip to the next iteration. Then mark it as closed, until it’s the end station, break the loop. If there is a direct link between the current station and the neighbor (distance[current][neighbor] != NO\_LINK) and the neighbor is not closed. Then, it will calculate the new distance. If this new distance is shorter than the previously recorded distance, update it. Update the prev array to remember the path. Add the neighbor to the priority queue.



After the loop, we have the shortest distances and the previous stations. Get shortest path from the end station to start station follow the prev, then, reverse the path to get the correct order.

# Task 13:

Dijkstra’s algorithm treats all nodes equally and explores them in a systematic order based solely on their current distances. It doesn’t consider any information about the target node. If the network has uniform edge weights, it will have good performance.

A\* algorithm takes into account both the actual cost (distance) and an estimate of the remaining cost (heuristic) to reach the target node.If you have additional information (e.g., congestion data, or historical travel times) that can serve as a heuristic, A\* can exploit it. A\* would likely perform better than Dijkstra’s algorithm in terms of exploration efficiency, especially if there are significant variations in edge weights or if the network has a large number of nodes.

In summary, Dijkstra’s algorithm is simpler and more widely applicable, but it can’t use any other heuristic.A\* algorithm is more complex and can be significantly faster in certain scenarios, but it requires a good heuristic.