Document version: 1.1 (2015-11-15)

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Chai	Student ID:	21186114
Other name(s):	Michael Chon Yun		
Unit name:	Data Structures and Algorithms	Unit ID:	COMP1002
Lecturer / unit coordinator:	Terence Tan Peng Lian		
Assessment:	Assignment		

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: Chai   Date of signature: 27 May 2023

(By submitting this form, you indicate that you agree with all the above text.)

Executive Summary

This report is written is for the COMP1002 Data Structures & Algorithms' assignment of creating a Java program to monitor bushfires using unmanned aerial vehicles (UAVs). UAVs will fly over various locations collecting environmental data including temperature, humidity, and wind speed. This bushfire monitoring program will not only store the locations and their associated data, but also process the data to quantify the level of bushfire risk of each area.

Location data will be stored using a graph data structure. Each vertex will represent a location, while each weighted edge will connect two locations with a distance. The provided *location.txt* file will be used as sample data, consisting of both vertices and weighted edges. The graph will be able to be displayed as an adjacency list.

An interactive user-interface menu will allow the user to insert new, remove, and/or search locations to the graph, making it scalable. Additional features have been added as well.

Data called by the UAVs will be stored in a hash table data structure. The provided *UAVdata.txt* will be used as sample data, consisting of data of each location. Rather than a list or an array, a hash table is used due to better storing and retrieval efficiency, which will be discussed later.

A heap data structure will process the UAV data, quantifying the bushfire risk level of each location. As well as sorting out the data into an array, ranking the locations starting from the location with the highest bushfire risk, all the way down to the lowest risk location.

Finally, UML diagrams, testing methodology and results, program limitations, and potential areas for improvement will be detailed in this report.

Table of Contents

UML Diagrams

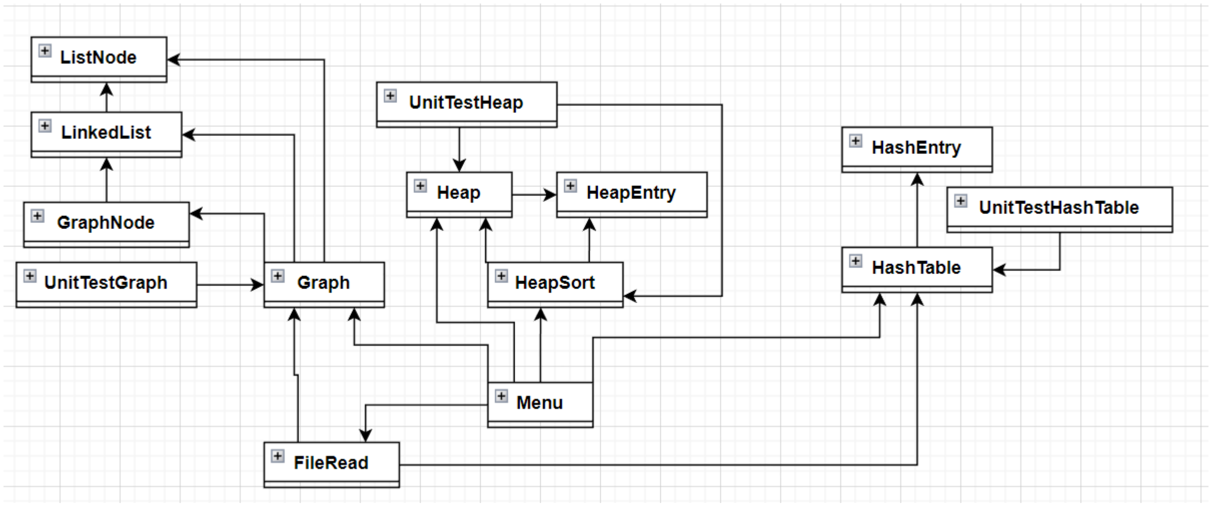


Figure 1: Class Relationships

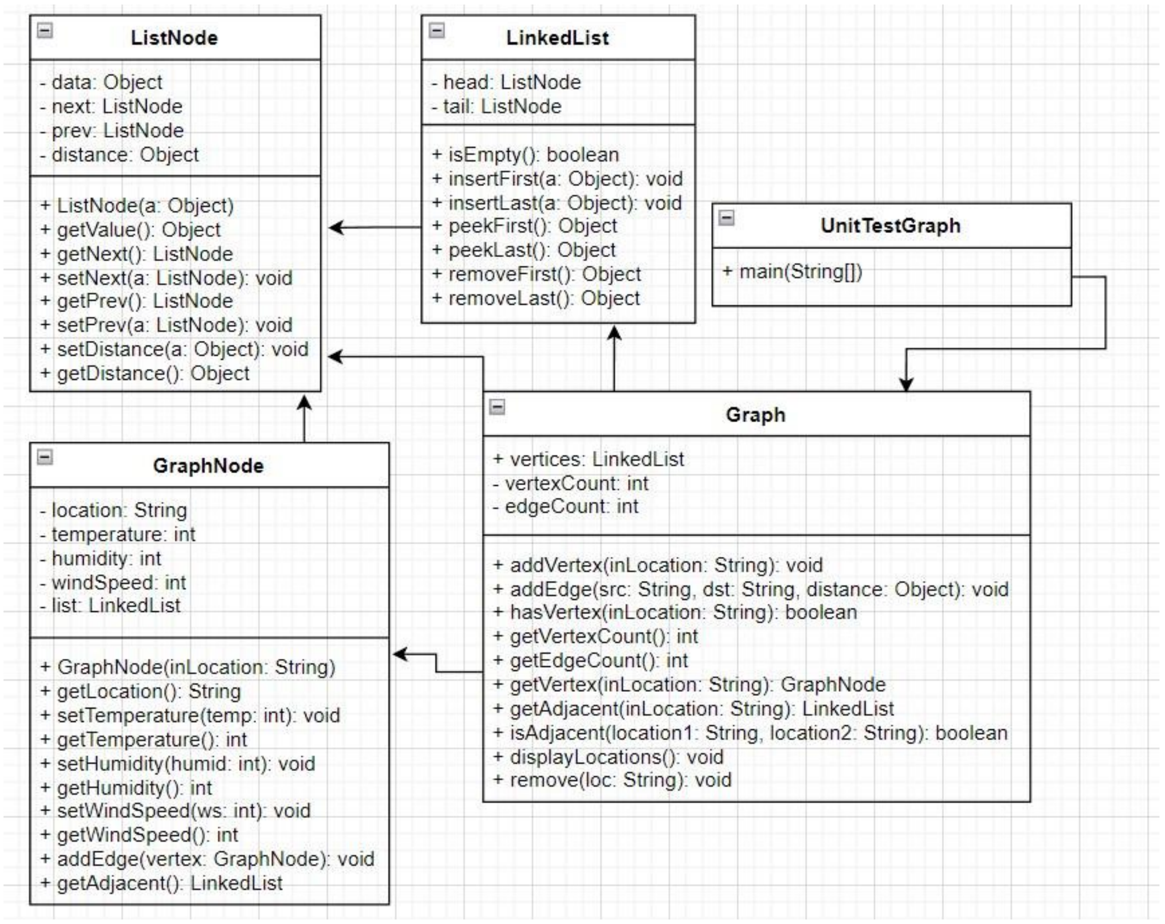
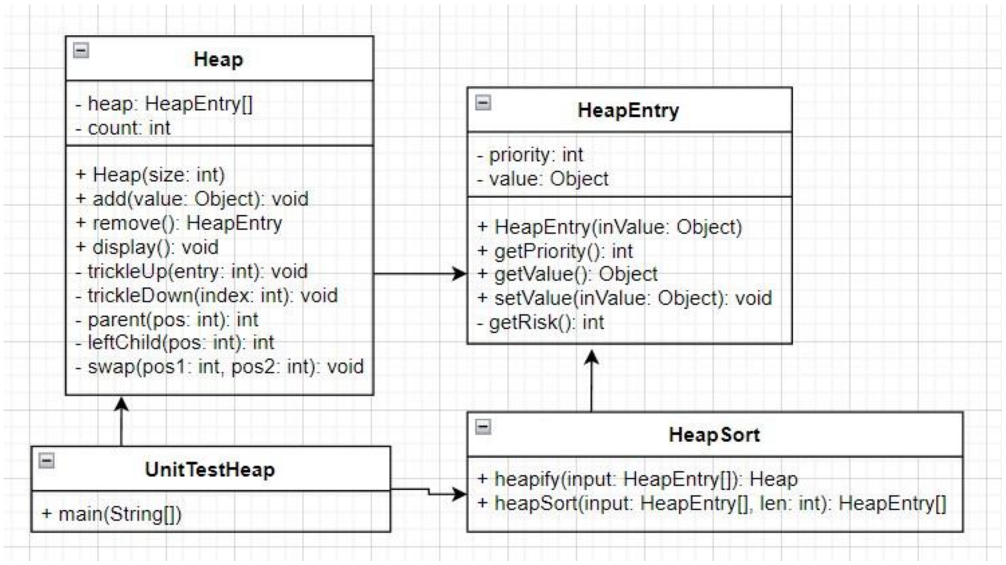
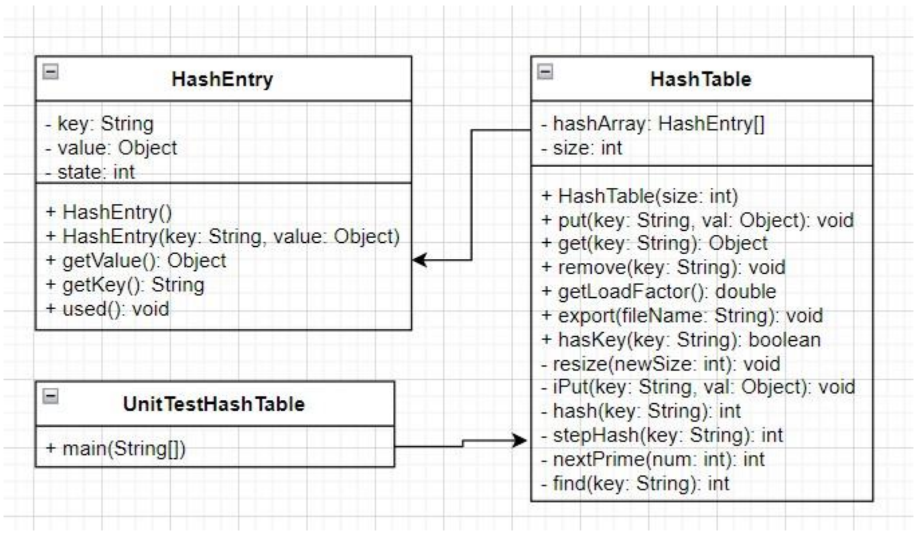
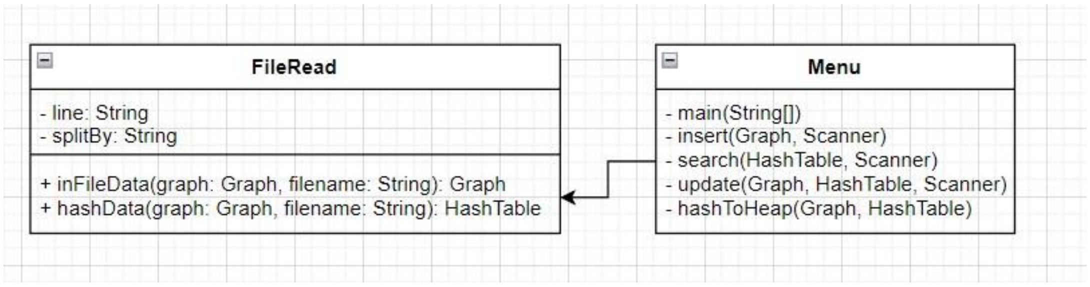

ListNode
- data: Object
- next: ListNode
- prev: ListNode
- distance: Object
+ ListNode(a: Object)
+ getValue(): Object
+ getNext(): ListNode
+ setNext(a: ListNode): void
+ getPrev(): ListNode
+ setPrev(a: ListNode): void
+ setDistance(a: Object): void
+ getDistance(): Object
LinkedList
- head: ListNode
- tail: ListNode
+ isEmpty(): boolean
+ insertFirst(a: Object): void
+ insertLast(a: Object): void
+ peekFirst(): Object
+ peekLast(): Object
+ removeFirst(): Object
+ removeLast(): Object
UnitTestGraph
+ main(String[])
GraphNode
- location: String
- temperature: int
- humidity: int
- windSpeed: int
- list: LinkedList
+ GraphNode(inLocation: String)
+ getLocation(): String
+ setTemperature(temp: int): void
+ getTemperature(): int
+ setHumidity(humid: int): void
+ getHumidity(): int
+ setWindSpeed(ws: int): void
+ getWindSpeed(): int
+ addEdge(vertex: GraphNode): void
+ getAdjacent(): LinkedList
Graph
+ vertices: LinkedList
- vertexCount: int
- edgeCount: int
+ addVertex(inLocation: String): void
+ addEdge(src: String, dst: String, distance: Object): void
+ hasVertex(inLocation: String): boolean
+ getVertexCount(): int
+ getEdgeCount(): int
+ getVertex(inLocation: String): GraphNode
+ getAdjacent(inLocation: String): LinkedList
+ isAdjacent(location1: String, location2: String): boolean
+ displayLocations(): void
+ remove(loc: String): void

Figure 2: UML Classes 1


Heap
- heap: HeapEntry[]
- count: int
+ Heap(size: int)
+ add(value: Object): void
+ remove(): HeapEntry
+ display(): void
- trickleUp(entry: int): void
- trickleDown(index: int): void
- parent(pos: int): int
- leftChild(pos: int): int
- swap(pos1: int, pos2: int): void
HeapEntry
- priority: int
- value: Object
+ HeapEntry(inValue: Object)
+ getPriority(): int
+ getValue(): Object
+ setValue(inValue: Object): void
- getRisk(): int
UnitTestHeap
+ main(String[])
HeapSort
+ heapify(input: HeapEntry[]): Heap
+ heapSort(input: HeapEntry[], len: int): HeapEntry[]
Figure 3: UML Classes 2


HashEntry
- key: String
- value: Object
- state: int
+ HashEntry()
+ HashEntry(key: String, value: Object)
+ getValue(): Object
+ getKey(): String
+ used(): void
HashTable
- hashArray: HashEntry[]
- size: int
+ HashTable(size: int)
+ put(key: String, val: Object): void
+ get(key: String): Object
+ remove(key: String): void
+ getLoadFactor(): double
+ export(fileName: String): void
+ hasKey(key: String): boolean
- resize(newSize: int): void
- iPut(key: String, val: Object): void
- hash(key: String): int
- stepHash(key: String): int
- nextPrime(num: int): int
- find(key: String): int
UnitTestHashTable
+ main(String[])
Figure 4: UML Classes 3


FileRead
- line: String
- splitBy: String
+ inFileData(graph: Graph, filename: String): Graph
+ hashData(graph: Graph, filename: String): HashTable
Menu
- main(String[])
- insert(Graph, Scanner)
- search(HashTable, Scanner)
- update(Graph, HashTable, Scanner)
- hashToHeap(Graph, HashTable)
Figure 5: UML Classes 4

Task 1: Creating a Graph

Class Description

In order to create a graph for Task 1, four classes are needed. Graph.java, GraphNode.java, LinkedList.java, and ListNode.java. All of which have been referenced from the author's own Practical 4 and Practical 6 in previous weeks.

LinkedList.java and ListNode.java each resemble conventional code for the Linked List data structure. In this case, a double-ended doubly linked list is used for easier traversal between List Nodes. One addition was made to ListNode.java is that a "Object: *distance"* field has been added to store the distance, presumably in kilometres, between locations in an edge. For instance, in the edge of A->B, A being the source and B being the destination, the ListNode object assigned to B will be holding the distance value.

Each GraphNode object will represent one unique location. It consists of five fields, a *location* field that will store the unique location ID, and a *list* field that holds a LinkedList to hold its adjacent GraphNode, if any. The remaining three fields will hold UAV data, those being temperature, humidity, and windspeed. These UAV data fields will not be utilised in the Graph itself but later in Task 4's Hash Table.

Graph.java will use its vertices to represent locations, and edges to represent paths between locations and their respective distances. In addition, a handful of methods exists to manage the locations and their relationships.

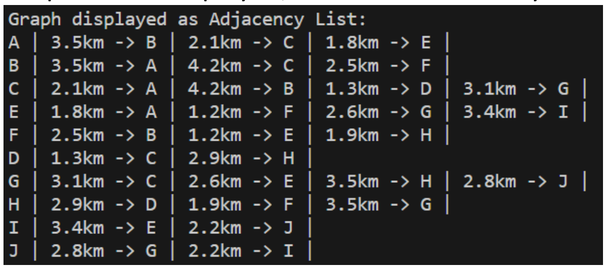
Methods to add vertices and edges are present. Getters have been included for the vertices and edges. As well as Boolean methods to check if a specific vertex or edge exist.

This graph is an undirected graph, as no one location can have a one-way path to another location in nature. The addEdge() method consists of a wrapper because when the user enters an edge, say A->B, the program will have to create two edges, A->B and B->A. Maintaining the undirected nature.

The graph can be displayed using displayLocations(), which would display the graph using adjacency lists.

Testing Methodology & Results

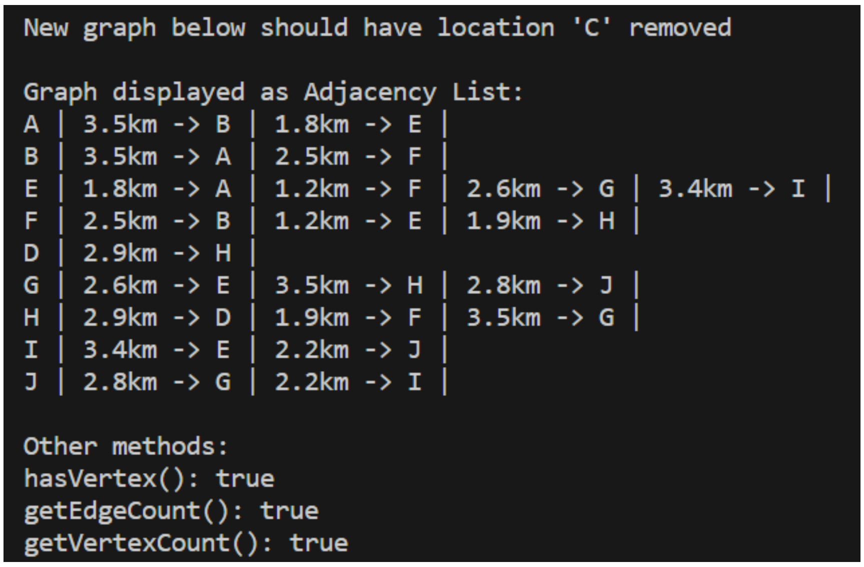
UnitTestGraph.java will utilise the provided *location.txt* file to test the Graph. To test, an empty graph is created, and loaded with the data from *location.txt* using the inFileData() from FileRead.java. Graph is then displayed, and the results verify the functionality of the graph.



```
Graph displayed as Adjacency List:
A | 3.5km -> B | 2.1km -> C | 1.8km -> E |
B | 3.5km -> A | 4.2km -> C | 2.5km -> F |
C | 2.1km -> A | 4.2km -> B | 1.3km -> D | 3.1km -> G |
E | 1.8km -> A | 1.2km -> F | 2.6km -> G | 3.4km -> I |
F | 2.5km -> B | 1.2km -> E | 1.9km -> H |
D | 1.3km -> C | 2.9km -> H |
G | 3.1km -> C | 2.6km -> E | 3.5km -> H | 2.8km -> J |
H | 2.9km -> D | 1.9km -> F | 3.5km -> G |
I | 3.4km -> E | 2.2km -> J |
J | 2.8km -> G | 2.2km -> I |
```

Figure 6: Graph is displayed using adjacency lists, all vertices and edges are present and distances are correct

Other methods are tested as well. Location 'C' is removed, and verified from displaying the graph again. Getters for the vertex and edge count are verified, as well as Boolean check hasVertex().



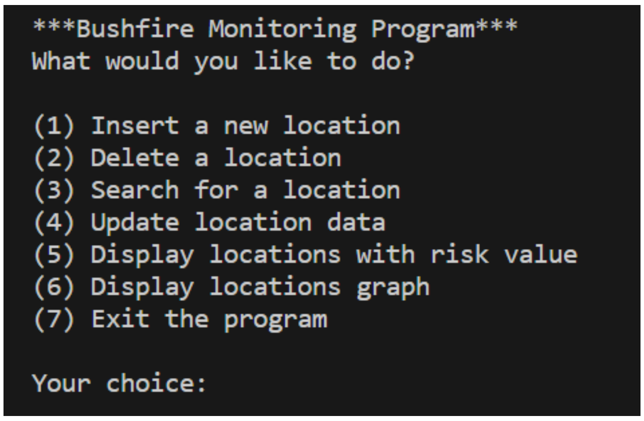
```
New graph below should have location 'C' removed

Graph displayed as Adjacency List:
A | 3.5km -> B | 1.8km -> E |
B | 3.5km -> A | 2.5km -> F |
E | 1.8km -> A | 1.2km -> F | 2.6km -> G | 3.4km -> I |
F | 2.5km -> B | 1.2km -> E | 1.9km -> H |
D | 2.9km -> H |
G | 2.6km -> E | 3.5km -> H | 2.8km -> J |
H | 2.9km -> D | 1.9km -> F | 3.5km -> G |
I | 3.4km -> E | 2.2km -> J |
J | 2.8km -> G | 2.2km -> I |

Other methods:
hasVertex(): true
getEdgeCount(): true
getVertexCount(): true
```

Figure 7: Location 'C' no longer exists in the graph, and other methods are verified

Task 3: Insert, Delete, and Searching Operations

Class Description

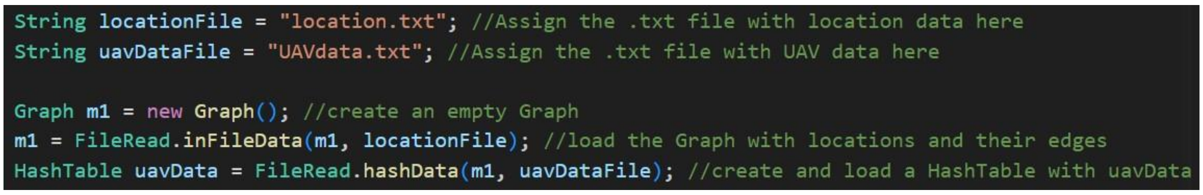
The user can carry out insert, delete, and searching operations directly from the interactive user menu in Menu.java, operations number 1, 2, and 3 respectively. The other operations will be covered later in this report.



```
***Bushfire Monitoring Program***
What would you like to do?

(1) Insert a new location
(2) Delete a location
(3) Search for a location
(4) Update location data
(5) Display locations with risk value
(6) Display locations graph
(7) Exit the program

Your choice:
```

Figure 8: Interactive user menu

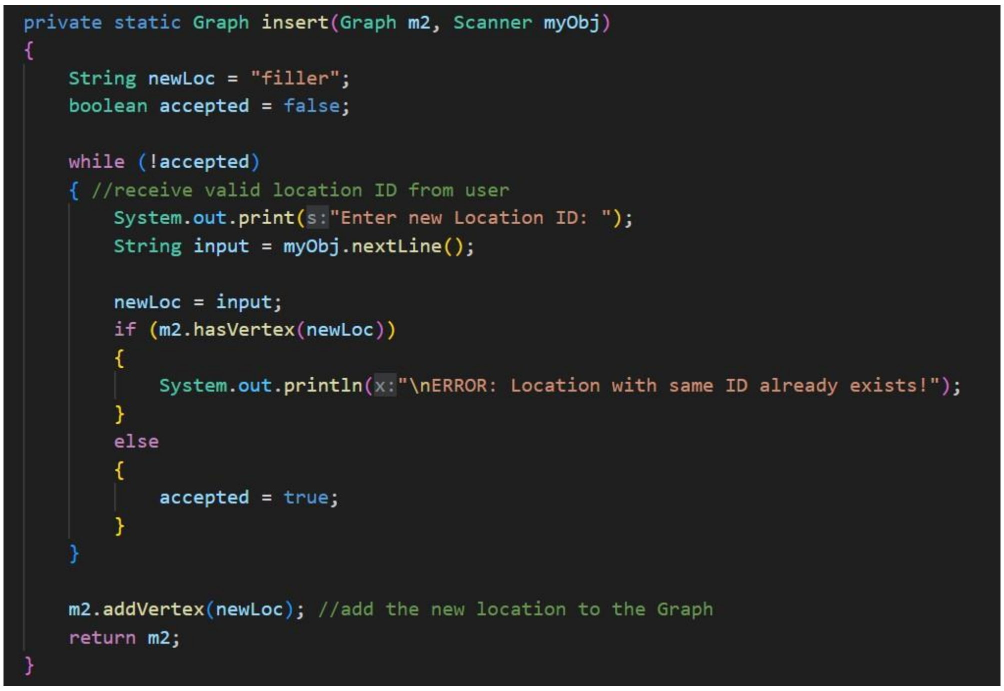
At the start of running Menu.java, a graph and hash table will be created and loaded with the data from *location.txt* and *UAVdata.txt* respectively. Loading the hash table also requires the graph to be passed in first, this is to ensure later that UAV data is not inserted into the hash table for a non-existent location in the graph.



```java
String locationFile = "location.txt"; //Assign the .txt file with location data here
String uavDataFile = "UAVdata.txt"; //Assign the .txt file with UAV data here

Graph m1 = new Graph(); //create an empty Graph
m1 = FileRead.inFileData(m1, locationFile); //load the Graph with locations and their edges
HashTable uavData = FileRead.hashData(m1, uavDataFile); //create and load a HashTable with uavData
```

Figure 9: Graph & Hash Table created and loaded

The insert method is stored inside the Menu.java class as a private static method which is called when the user selects operation 1.

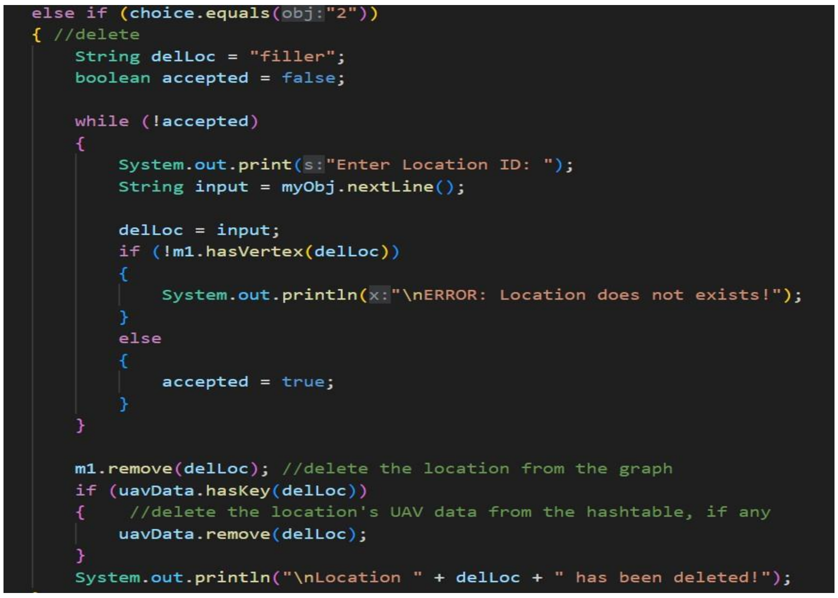


```java
private static Graph insert(Graph m2, Scanner myObj)
{
    String newLoc = "filler";
    boolean accepted = false;

    while (!accepted)
    { //receive valid location ID from user
        System.out.print(s:"Enter new Location ID: ");
        String input = myObj.nextLine();

        newLoc = input;
        if (m2.hasVertex(newLoc))
        {
            System.out.println(x:"\nERROR: Location with same ID already exists!");
        }
        else
        {
            accepted = true;
        }
    }

    m2.addVertex(newLoc); //add the new location to the Graph
    return m2;
}
```

Figure 10: Insert method

The pre-existing graph created at the start of the program, along with Scanner object is passed into the insert method. A while-loop will receive the user input and ensure that the user does not use a location ID that is already used in the graph. Once the input is valid, the new location will be added to the graph, which is later returned back to the program, updated with the new location.

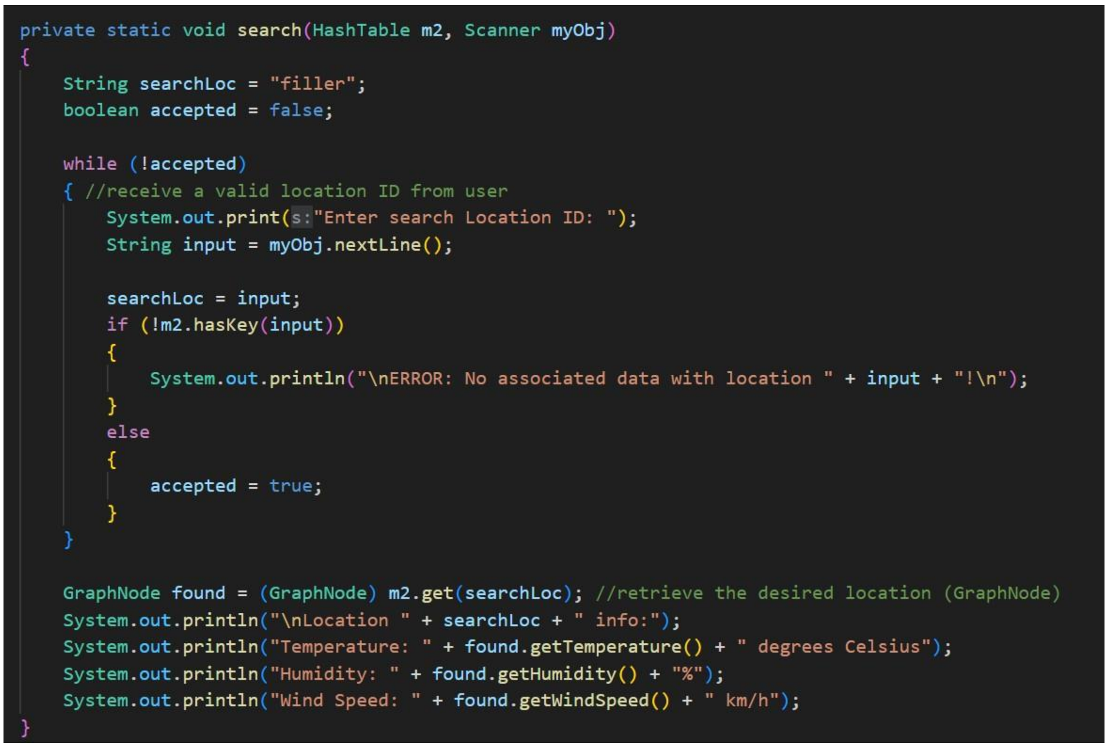
Remove function will delete a user-chosen location from both the graph and hash table. The operation is stored within the Main method of Menu.java, it will accept and validate a user input, ensuring it does exist in the graph to begin with. Then it will proceed to call the graph's and hash table's respective remove methods.



```
else if (choice.equals(obj:"2"))
{ //delete
    String delLoc = "filler";
    boolean accepted = false;

    while (!accepted)
    {
        System.out.print(s:"Enter Location ID: ");
        String input = myObj.nextLine();

        delLoc = input;
        if (!m1.hasVertex(delLoc))
        {
            System.out.println(x:"\nERROR: Location does not exists!");
        }
        else
        {
            accepted = true;
        }
    }

    m1.remove(delLoc); //delete the location from the graph
    if (uavData.hasKey(delLoc))
    {   //delete the location's UAV data from the hashtable, if any
        uavData.remove(delLoc);
    }
    System.out.println("\nLocation " + delLoc + " has been deleted!");
```

Figure 11: Delete operation within Menu.java's Main method

Similarly, to the insert operation, searching operation is stored as a private static method within the Menu.java class as well and is called when the user opts for it in the main method.



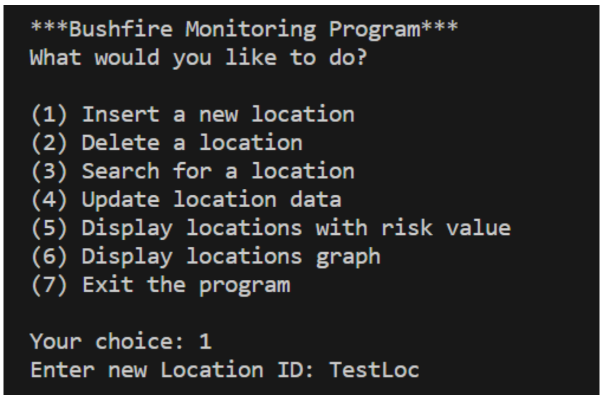
```
private static void search(HashTable m2, Scanner myObj)
{
    String searchLoc = "filler";
    boolean accepted = false;

    while (!accepted)
    { //receive a valid location ID from user
        System.out.print(s:"Enter search Location ID: ");
        String input = myObj.nextLine();

        searchLoc = input;
        if (!m2.hasKey(input))
        {
            System.out.println("\nERROR: No associated data with location " + input + "!\n");
        }
        else
        {
            accepted = true;
        }
    }

    GraphNode found = (GraphNode) m2.get(searchLoc); //retrieve the desired location (GraphNode)
    System.out.println("\nLocation " + searchLoc + " info:");
    System.out.println("Temperature: " + found.getTemperature() + " degrees Celsius");
    System.out.println("Humidity: " + found.getHumidity() + "%");
    System.out.println("Wind Speed: " + found.getWindSpeed() + " km/h");
}
```

Figure 12: Search operation

Upon calling the search method, hash table and Scanner object will be passed in. User input for the wanted location will be validated, then the location's data will be printed to the screen.

Testing Methodology & Results

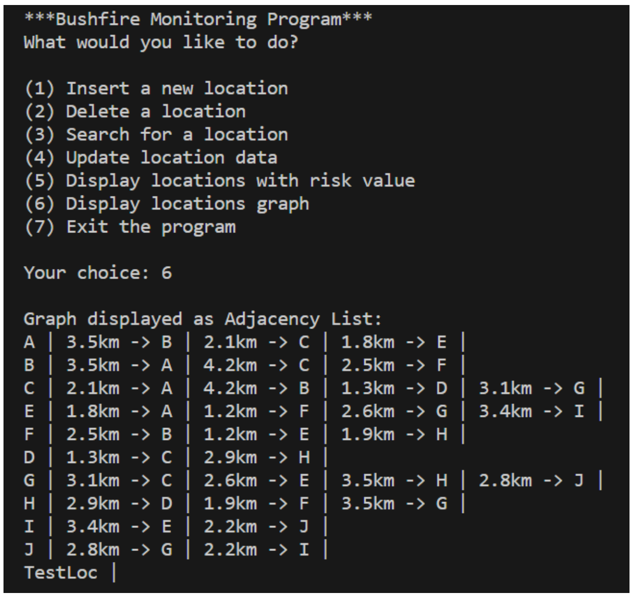
All three operations are tested simply by running the menu program itself. First, a new location labelled "TestLoc" is inserted to the graph.



```
***Bushfire Monitoring Program***
What would you like to do?

(1) Insert a new location
(2) Delete a location
(3) Search for a location
(4) Update location data
(5) Display locations with risk value
(6) Display locations graph
(7) Exit the program

Your choice: 1
Enter new Location ID: TestLoc
```

Figure 13: "TestLoc" location inserted

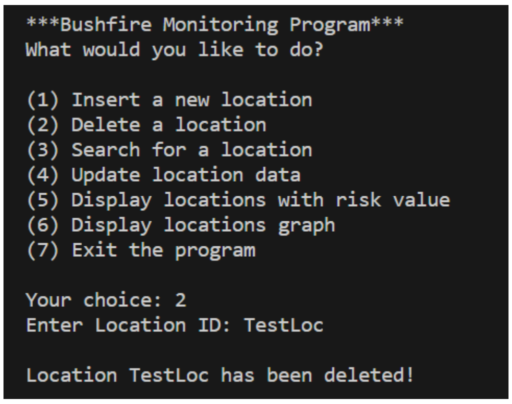
Later verified by displaying the graph, as the newly added "TestLoc" location is present.



```
***Bushfire Monitoring Program***
What would you like to do?

(1) Insert a new location
(2) Delete a location
(3) Search for a location
(4) Update location data
(5) Display locations with risk value
(6) Display locations graph
(7) Exit the program

Your choice: 6

Graph displayed as Adjacency List:
A | 3.5km -> B | 2.1km -> C | 1.8km -> E |
B | 3.5km -> A | 4.2km -> C | 2.5km -> F |
C | 2.1km -> A | 4.2km -> B | 1.3km -> D | 3.1km -> G |
E | 1.8km -> A | 1.2km -> F | 2.6km -> G | 3.4km -> I |
F | 2.5km -> B | 1.2km -> E | 1.9km -> H |
D | 1.3km -> C | 2.9km -> H |
G | 3.1km -> C | 2.6km -> E | 3.5km -> H | 2.8km -> J |
H | 2.9km -> D | 1.9km -> F | 3.5km -> G |
I | 3.4km -> E | 2.2km -> J |
J | 2.8km -> G | 2.2km -> I |
TestLoc |
```

Figure 14: New "TestLoc" location is present at the bottom of the graph

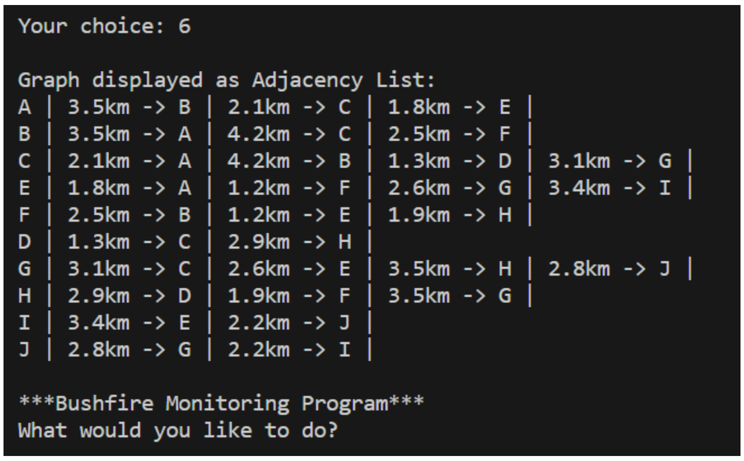
Next up, delete operation is tested within the same runtime. Earlier added "TestLoc" location will be removed.



```
***Bushfire Monitoring Program***
What would you like to do?

(1) Insert a new location
(2) Delete a location
(3) Search for a location
(4) Update location data
(5) Display locations with risk value
(6) Display locations graph
(7) Exit the program

Your choice: 2
Enter Location ID: TestLoc

Location TestLoc has been deleted!
```

Figure 15: "TestLoc" location is deleted

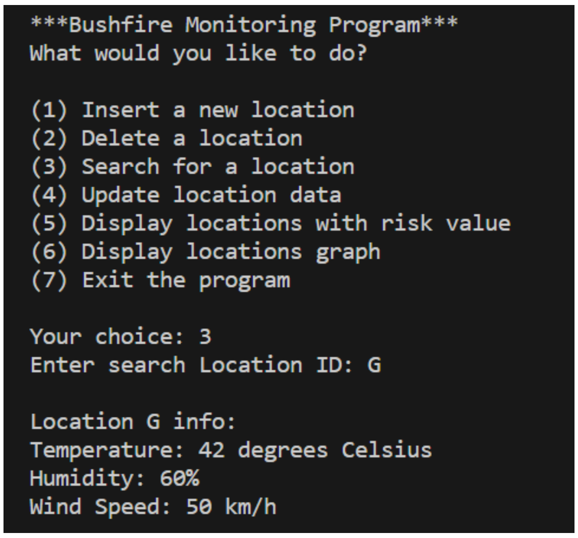
Graph is displayed again revealing that deleted “TestLoc” location is gone.



```
Your choice: 6

Graph displayed as Adjacency List:
A | 3.5km -> B | 2.1km -> C | 1.8km -> E |
B | 3.5km -> A | 4.2km -> C | 2.5km -> F |
C | 2.1km -> A | 4.2km -> B | 1.3km -> D | 3.1km -> G |
E | 1.8km -> A | 1.2km -> F | 2.6km -> G | 3.4km -> I |
F | 2.5km -> B | 1.2km -> E | 1.9km -> H |
D | 1.3km -> C | 2.9km -> H |
G | 3.1km -> C | 2.6km -> E | 3.5km -> H | 2.8km -> J |
H | 2.9km -> D | 1.9km -> F | 3.5km -> G |
I | 3.4km -> E | 2.2km -> J |
J | 2.8km -> G | 2.2km -> I |

***Bushfire Monitoring Program***
What would you like to do?
```

Figure 16: "TestLoc" is gone

Lastly, search operation is tested by searching for Location ‘G’.



```
***Bushfire Monitoring Program***
What would you like to do?

(1) Insert a new location
(2) Delete a location
(3) Search for a location
(4) Update location data
(5) Display locations with risk value
(6) Display locations graph
(7) Exit the program

Your choice: 3
Enter search Location ID: G

Location G info:
Temperature: 42 degrees Celsius
Humidity: 60%
Wind Speed: 50 km/h
```

Figure 17: Location 'G' successfully searched

Task 4: Hashing the Data

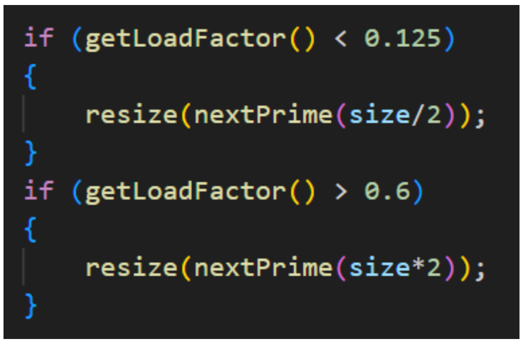
Class Descriptions

HashTable.java and HashEntry.java classes have been referenced by author’s own Practical 8.

Each HashEntry object will represent one location and its associated data. The conventional key-value HashEntry is used with the location’s ID used as the key, and the entire location graph node as the value. This will allow the user to look up locations by entering a location ID, which the program will use to look for the specific hash entry, and return the location’s graph node in which the associated data may be obtained from.

The hash table in question utilises double-hashing rather than linear probing to effectively handle collisions. This will prevent the formation of large data chunks in the Hash Array which can quickly degrade the searching operation’s O(1) time complexity to O(N) in the worst case.

Resizing of the Hash Table has been implemented as well to either increase or decrease the size of the hash table based on the load factor. A load factor threshold of 0.125 to 0.6 is used.



```java
if (getLoadFactor() < 0.125)
{
    resize(nextPrime(size/2));
}
if (getLoadFactor() > 0.6)
{
    resize(nextPrime(size*2));
}
```

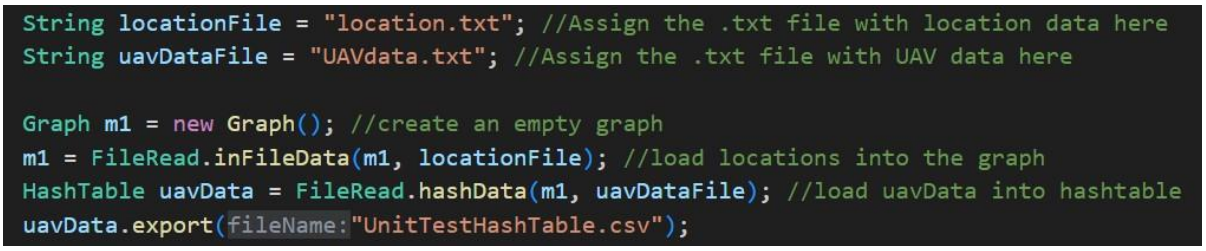
Figure 18: Load Factor threshold

In layman terms, no more than 60% of the hash table can be used up, and no less than 12.5% of the hash table can be used up. This ensures the hash table's time complexity is not degraded due to more collisions, and space complexity is not degraded due to more empty spaces in the hash table.

Other conventional methods such as Boolean methods to check for existing entries, remove entries methods, and getters have been implemented. An export method is implemented as well to help with unit testing.

Testing Methodology & Results

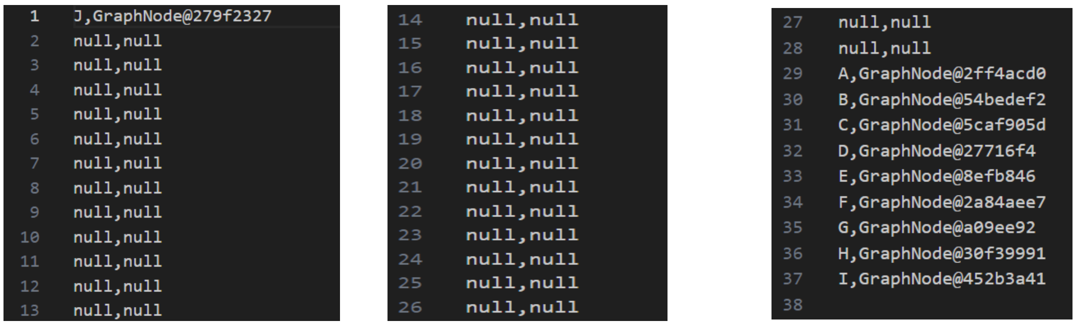
The Hash Table is tested in UnitTestHashTable.java with both data from *UAVdata.txt* and sample data provided by the author. Starting with data from *UAVdata.txt,* a graph is first created and loaded with data from *location.txt,* then it is used to create the hash table. As seen earlier, file reading takes place by calling the file reading methods from FileRead.java



```java
String locationFile = "location.txt"; //Assign the .txt file with location data here
String uavDataFile = "UAVdata.txt"; //Assign the .txt file with UAV data here

Graph m1 = new Graph(); //create an empty graph
m1 = FileRead.inFileData(m1, locationFile); //load locations into the graph
HashTable uavData = FileRead.hashData(m1, uavDataFile); //load uavData into hashtable
uavData.export(fileName:"UnitTestHashTable.csv");
```

Figure 19: Test Hash Table with UAVdata.txt

The hash table is then exported to a CSV file to verify its contents.

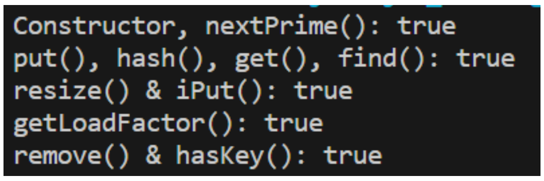


```
J,GraphNode@279f2327
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
null,null
A,GraphNode@2ff4acd0
B,GraphNode@54bedef2
C,GraphNode@5caf905d
D,GraphNode@27716f4
E,GraphNode@8efb846
F,GraphNode@2a84aee7
G,GraphNode@a09ee92
H,GraphNode@30f39991
I,GraphNode@452b3a41

```

Figure 20: UAVdata CSV Output

As seen, all entries are present, and the hash table size fits within the load factor threshold.

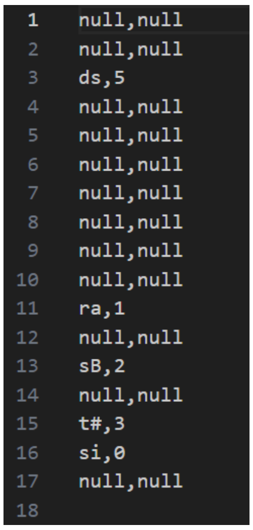
Next up, a small set of sample data is tested to further test the hash table, referenced from author's own Practical 8 as well.



```
Constructor, nextPrime(): true
put(), hash(), get(), find(): true
resize() & iPut(): true
getLoadFactor(): true
remove() & hasKey(): true
```

Figure 21: All methods verified

Similarly, the sample data is exported to a CSV file for verification.



```
null,null
null,null
ds,5
null,null
null,null
null,null
null,null
null,null
null,null
null,null
ra,1
null,null
sB,2
null,null
t#,3
si,0
null,null
```

Figure 22: Alternative test data CSV output

As seen, the test data is evenly spread out, and the hash table size is within the load factor threshold.

Hash Table over List or Array

Hash Tables are deemed far more efficient in searching/retrieval operations in comparison to Linked Lists and Arrays.

Mainly because hash tables employ hashing to map keys to their designated values. Therefore, when a user enters a key, the key will be hashed into its hash index, then the value can be directly accessed. Giving the search operation an average time complexity of O(1), constant time.

Where as in Linked Lists or Arrays, searching is done by linearly iterating from the start of a list or array until the end, comparing each element to the element in search. Giving this a time complexity of O(N).

Furthermore, the hash table in this case utilises double hashing to effectively prevent chunk formation during collisions. As well as resizing based on optimal load factor thresholds ensuring that its average time complexity of O(N), constant time, is maintained as the amount of data increases or decreases.

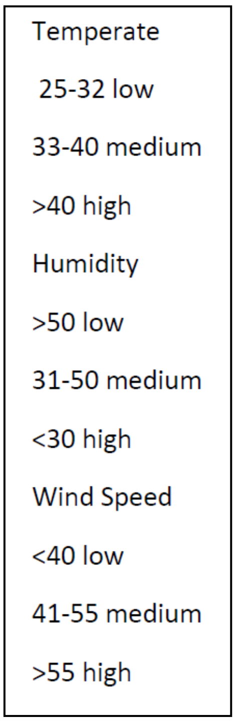
Task 5: Using a Heap

Class Descriptions

The heap utilises classes Heap.java, HeapEntry.java, and HeapSort.java. All of which have been referenced from author's own Practical 9.

Each HeapEntry object will represent one location and its risk value. It utilises the conventional priority and value pairing. Its constructor has been modified to only accept one parameter, the value, which in this case is a location's graph node object. The priority value will represent the location's risk value, which is computed within the HeapEntry object itself by analysing the location's UAV data from the inserted graph node using getRisk().

The risk threshold used would be based on the sample threshold provided in the assignment specification.



Temperate

25-32 low

33-40 medium

>40 high

Humidity

>50 low

31-50 medium

<30 high

Wind Speed

<40 low

41-55 medium

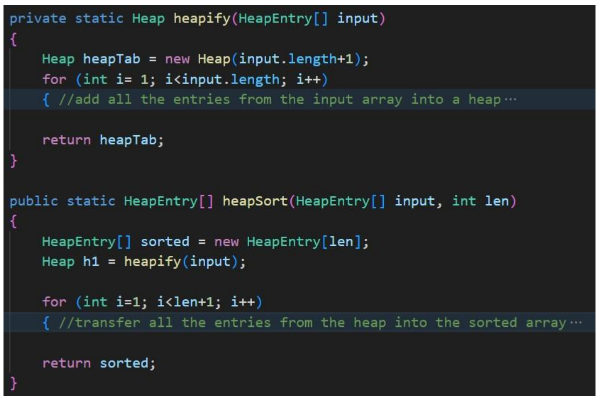
>55 high

Figure 23: Sample threshold from assignment specification

The risk value (*priority variable*) will be an integer score from 3-9. Scoring is provided by three aspects, temperature, humidity, and wind speed. For each aspect, low risk would generate one point, medium risk would generate two points, and high risk would generate three points.

Heap.java class continues to utilise the same conventional methods, including add(), remove(), display(), and other private methods for its own use.

HeapSort has been implemented to sort locations in a HeapEntry array, in descending order based on their corresponding risk value.



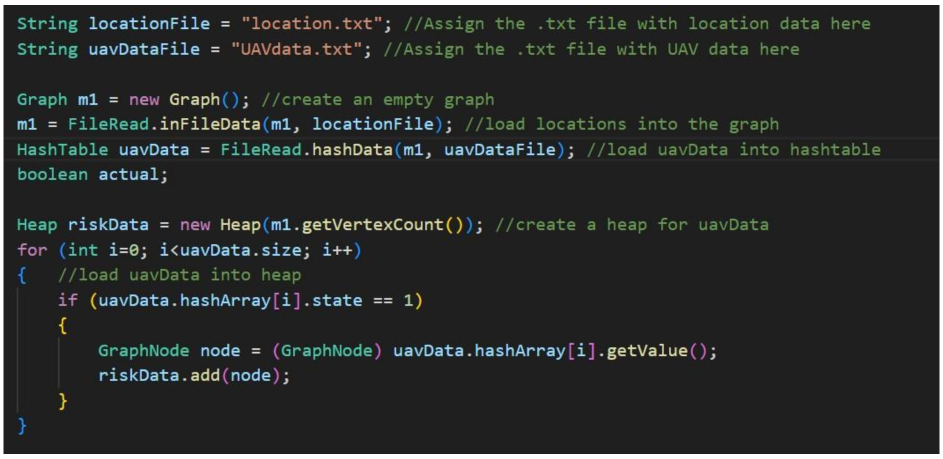
```java
private static Heap heapify(HeapEntry[] input)
{
    Heap heapTab = new Heap(input.length+1);
    for (int i= 1; i<input.length; i++)
    { //add all the entries from the input array into a heap···

    return heapTab;
}

public static HeapEntry[] heapSort(HeapEntry[] input, int len)
{
    HeapEntry[] sorted = new HeapEntry[len];
    Heap h1 = heapify(input);

    for (int i=1; i<len+1; i++)
    { //transfer all the entries from the heap into the sorted array···

    return sorted;
}
```

Figure 24: HeapSort.java class

The heapSort() method will accept a heapEntry array and its length, from another heap. It will proceed to call heapify() and pass in the heapEntry input. Heapify() will then add all the entries from the input array into a heap, and return the heap to heapSort(). Lastly, heapSort() will transfer all the entries from the heap into a sorted array. Sorted array now has all the locations sorted by their risk value in descending order.

Testing Methodology & Results

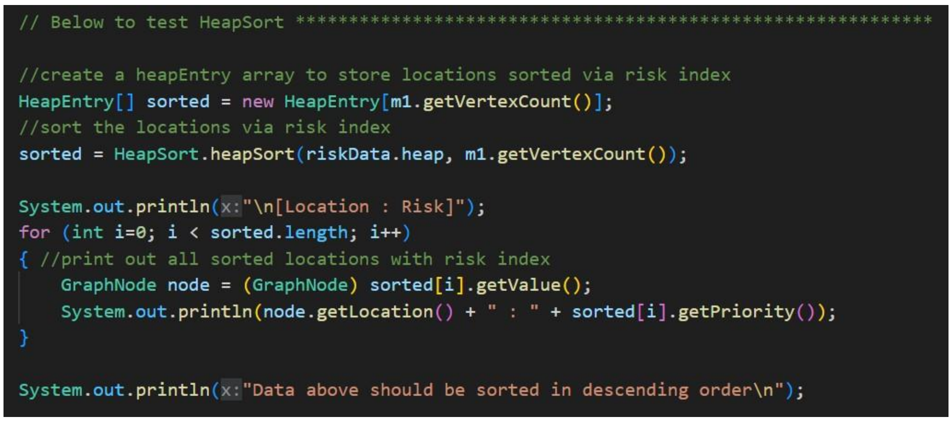
Heap testing is done in UnitTestHeap.java. Similarly, to Graph and Hash Table, sample data used is from *locations.txt* and *UAVdata.txt,* so the graph and hash table must be created and loaded first. Then the heap can be created using the location data from the hash table.



```java
String locationFile = "location.txt"; //Assign the .txt file with location data here
String uavDataFile = "UAVdata.txt"; //Assign the .txt file with UAV data here

Graph m1 = new Graph(); //create an empty graph
m1 = FileRead.inFileData(m1, locationFile); //load locations into the graph
HashTable uavData = FileRead.hashData(m1, uavDataFile); //load uavData into hashtable
boolean actual;

Heap riskData = new Heap(m1.getVertexCount()); //create a heap for uavData
for (int i=0; i<uavData.size; i++)
{   //load uavData into heap
    if (uavData.hashArray[i].state == 1)
    {
        GraphNode node = (GraphNode) uavData.hashArray[i].getValue();
        riskData.add(node);
    }
}
```

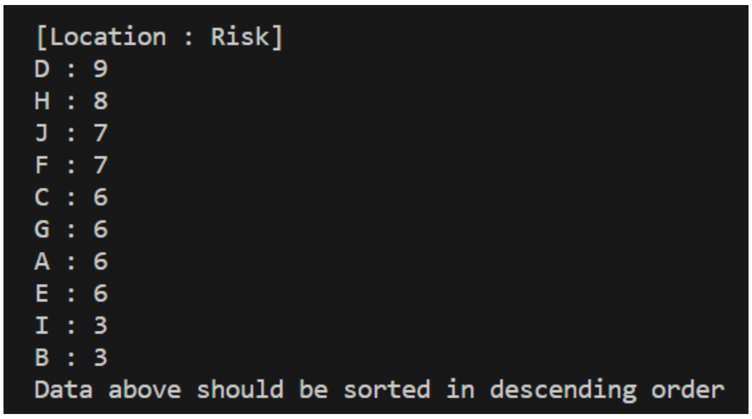
Figure 25: Creating a Heap

Now that the heap is created, HeapSort() is called to create a sorted array of the locations.



```java
// Below to test HeapSort ***********************************************************

//create a heapEntry array to store locations sorted via risk index
HeapEntry[] sorted = new HeapEntry[m1.getVertexCount()];
//sort the locations via risk index
sorted = HeapSort.heapSort(riskData.heap, m1.getVertexCount());

System.out.println(x:"\n[Location : Risk]");
for (int i=0; i < sorted.length; i++)
{ //print out all sorted locations with risk index
    GraphNode node = (GraphNode) sorted[i].getValue();
    System.out.println(node.getLocation() + " : " + sorted[i].getPriority());
}

System.out.println(x:"Data above should be sorted in descending order\n");
```

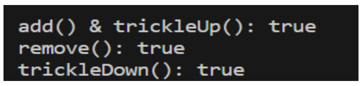
Figure 26: HeapSort called and tested



```
[Location : Risk]
D : 9
H : 8
J : 7
F : 7
C : 6
G : 6
A : 6
E : 6
I : 3
B : 3
Data above should be sorted in descending order
```

Figure 27: Sorted array verified

Sorted array is verified as all locations are correctly arranged in descending order, starting from the location with the highest risk. Other Heap.java methods are tested as well.



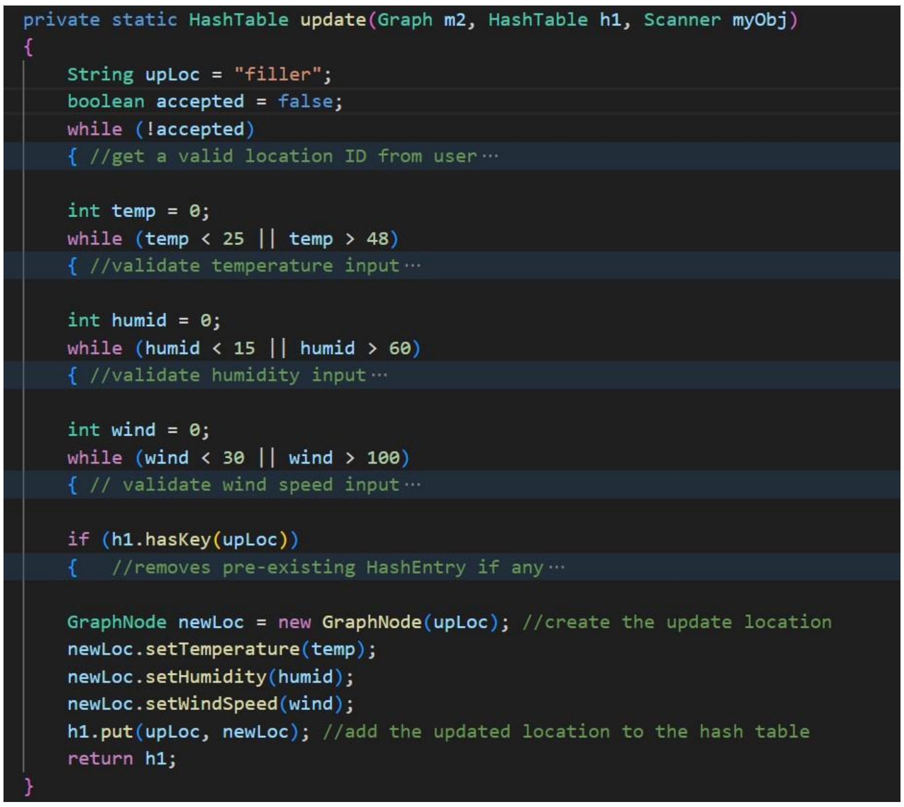
```
add() & trickleUp(): true
remove(): true
trickleDown(): true
```

Figure 28: Other heap methods tested

Task 7: Writing & Testing the Program

As witnessed in Task 3's Testing Methodology & Results section, the interactive menu has so far been tested to verify the insert, delete, and searching operations. However, the interactive menu still has additional features, those being, updating location information and displaying locations arranged by risk values.

The update location operation is stored as a private static method within Menu.java. It will accept the graph, the hash table, and Scanner object as parameters, then later return the updated hash table back to main method.



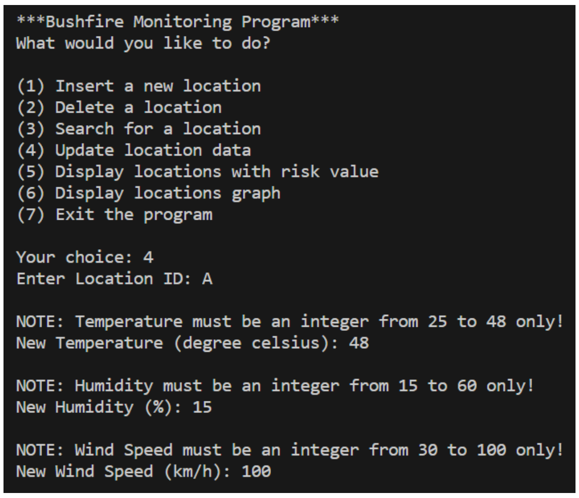
```java
private static HashTable update(Graph m2, HashTable h1, Scanner myObj)
{
    String upLoc = "filler";
    boolean accepted = false;
    while (!accepted)
    { //get a valid location ID from user···

    int temp = 0;
    while (temp < 25 || temp > 48)
    { //validate temperature input···

    int humid = 0;
    while (humid < 15 || humid > 60)
    { //validate humidity input···

    int wind = 0;
    while (wind < 30 || wind > 100)
    { // validate wind speed input···

    if (h1.hasKey(upLoc))
    {   //removes pre-existing HashEntry if any···

    GraphNode newLoc = new GraphNode(upLoc); //create the update location
    newLoc.setTemperature(temp);
    newLoc.setHumidity(humid);
    newLoc.setWindSpeed(wind);
    h1.put(upLoc, newLoc); //add the updated location to the hash table
    return h1;
}
```

Figure 29: update method

Once called, it will request the user to pick a location he/she would like to update, validate it ensuring it does exist in the graph to begin with. Then request the user for the new updated location data which is also validated.

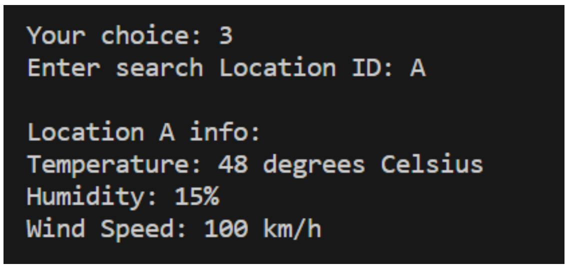
Finally, it will delete the old location data from the hash table, if any. Create the new location data and add it to the hash table. The updated hash table is then returned to the main method.

To put it to the test, location 'A' with the initial data values of [temp: 32, humidity: 45, wind: 90], will be updated to [48, 15, 100]. Then verified by using the search operation.



```
***Bushfire Monitoring Program***
What would you like to do?

(1) Insert a new location
(2) Delete a location
(3) Search for a location
(4) Update location data
(5) Display locations with risk value
(6) Display locations graph
(7) Exit the program

Your choice: 4
Enter Location ID: A

NOTE: Temperature must be an integer from 25 to 48 only!
New Temperature (degree celsius): 48

NOTE: Humidity must be an integer from 15 to 60 only!
New Humidity (%): 15

NOTE: Wind Speed must be an integer from 30 to 100 only!
New Wind Speed (km/h): 100
```

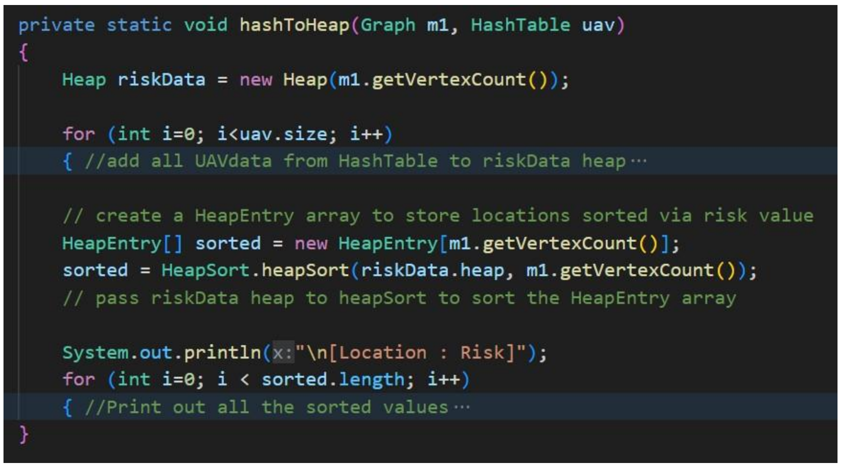
Figure 30: Update Location 'A' data



```
Your choice: 3
Enter search Location ID: A

Location A info:
Temperature: 48 degrees Celsius
Humidity: 15%
Wind Speed: 100 km/h
```

Figure 31: Location 'A' updated data validated

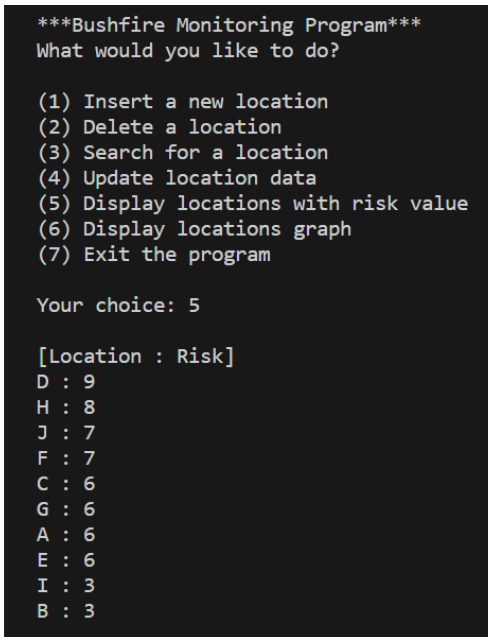
Finally, the last operation possible, display locations with risk value, is also stored as a private static method inside of Menu.java, which accepts the hash table and graph as parameters, then prints out all the locations sorted by risk value.



```java
private static void hashToHeap(Graph m1, HashTable uav)
{
    Heap riskData = new Heap(m1.getVertexCount());

    for (int i=0; i<uav.size; i++)
    { //add all UAVdata from HashTable to riskData heap···

    // create a HeapEntry array to store locations sorted via risk value
    HeapEntry[] sorted = new HeapEntry[m1.getVertexCount()];
    sorted = HeapSort.heapSort(riskData.heap, m1.getVertexCount());
    // pass riskData heap to heapSort to sort the HeapEntry array

    System.out.println(x:"\n[Location : Risk]");
    for (int i=0; i < sorted.length; i++)
    { //Print out all the sorted values···
}
```

Figure 32: HashToHeap method

It first creates a heap of the size of number of locations. Add all the UAV data from the hash table to the heap. Then calls the heapSort() from HeapSort.java to sort the locations, then print the sorted locations to the screen.

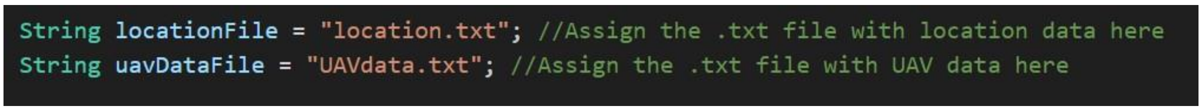


```
***Bushfire Monitoring Program***
What would you like to do?

(1) Insert a new location
(2) Delete a location
(3) Search for a location
(4) Update location data
(5) Display locations with risk value
(6) Display locations graph
(7) Exit the program

Your choice: 5

[Location : Risk]
D : 9
H : 8
J : 7
F : 7
C : 6
G : 6
A : 6
E : 6
I : 3
B : 3
```

Figure 33: Display locations with risk value tested

Further Information

Scalability was taken into account during the development of this program. Therefore, the input files for location and UAV data can be easily modified from the Menu.java class.

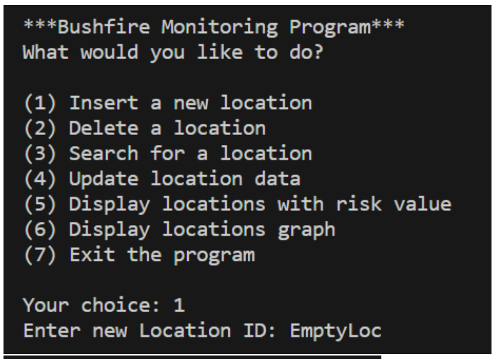


```
String locationFile = "location.txt"; //Assign the .txt file with location data here
String uavDataFile = "UAVdata.txt"; //Assign the .txt file with UAV data here
```

Figure 34: Change the input files from the main method of Menu.java

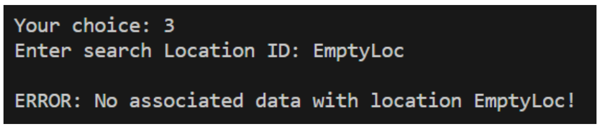
Simply swap the filenames for locationFile and uavDataFile, with the new files. The program will work as normally granted the new files are of the same style and format as *location.txt* and *UAVdata.txt.*

In the main menu, when a new location is inserted. It would have no UAV data values by default, thus, it is not registered in the hash table or heap. Until it is updated later with new values.



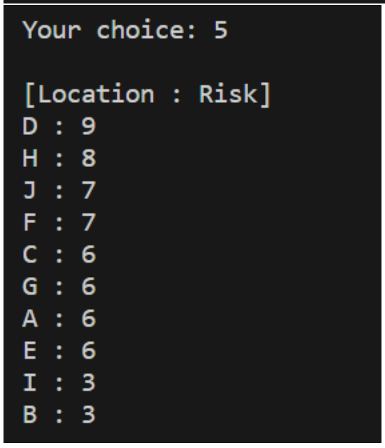
```
***Bushfire Monitoring Program***
What would you like to do?

(1) Insert a new location
(2) Delete a location
(3) Search for a location
(4) Update location data
(5) Display locations with risk value
(6) Display locations graph
(7) Exit the program

Your choice: 1
Enter new Location ID: EmptyLoc
```

Figure 35: New location "EmptyLoc" is inserted



```
Your choice: 3
Enter search Location ID: EmptyLoc

ERROR: No associated data with location EmptyLoc!
```

Figure 36: Searching for "EmptyLoc" will yield no results



```
Your choice: 5

[Location : Risk]
D : 9
H : 8
J : 7
F : 7
C : 6
G : 6
A : 6
E : 6
I : 3
B : 3
```

Figure 37: "EmptyLoc" will not appear in Display locations with risk values either

Potential areas of improvements or extensions include implementation of Task 2' BFS and DFS, as well as Task 6's Itinerary. However, the rest of the assignment's tasks and specification have been fulfilled so far with sufficient error handling.