COSC 222: Data Structures
Lab 8 – Minimum Spanning Tree, Hashing and Sorting

For this final lab, you will gain an understanding of how collisions are handled in a hash set, and practice coding methods for collision handling in a hash set and performance of sorting algorithm.

Question 1 (Hashing): Hash Functions [6 marks]
Suppose we have a hash table of size N = 7, and we would like to store integer values there. We have a hash function `hash(i) = i % N`.

In a step-by-step manner, draw the tables (three separate tables for a, b, and c) that result after inserting the following values in the given order: **17, 25, 11, 46, 14** Assume that collisions are handled by

a. linear probing
b. quadratic probing (use the function discussed in the lecture)
c. separate chaining

You will need to submit a PDF file with your answers for this question. This can be handwritten and scanned, drawn on a tablet, or created using a diagramming program and word processor. Name your file **Lab8Question1.pdf**.

Question 2 (Hashing): Collision Handling [10 marks]
Complete a program which uses each of the above methods to handle collisions when entering integers into a hash table. Use the provided files: **HashTable.java** and **HashTableTest.java**.

As usual, **HashTableTest.java** contains the class to test your methods using a given data set. You will need to complete a few methods in the **HashTable.java** file. Notice that several methods are already complete: primarily the constructor, the `hash()` function, and an `add()` method. However, notice that the `add()` method calls `handleCollision()`, which then calls one of the methods discussed in lecture to handle collisions: linear probing, quadratic probing, and double hashing. Your task is to complete these three methods as described below.

You may make the following assumptions: (a) zeroes will not be added to the hash table, (b) negative numbers will not be added to the hash table, (c) rehashing will not be required, and (d) no elements will be removed.

Part A [3 marks]:
`linearProbing()`: This is the most straightforward method of the three. If the hash function indicates that a value should be stored at an already full (non-zero) index of the array, this function increments the index until an available spot is found, then inserts the new element at that index.

Part B [3 marks]:

`quadraticProbing()`: This method is similar to linear probing, except instead of incrementing the index by one until an available spot is found, this method increments by a quadratically increasing amount. In other words, if the initial index returned by the hash function is `i`, the next index tried is $i+1^2$, then $i+2^2$, and so on until an available spot is found.

Part C [4 marks]:

`doubleHashing()`: This final method is slightly more complicated. A secondary hash function is calculated in the case of a collision. This function often is in the form of `d(k) = q - k % q` for a given key `k`, when `q` is a prime number. Use this secondary hash function format for this question. Notice that the `nextHighestPrime()` function has been provided to return a suitable value for `q`. After this second hash code has been calculated, add it to the original hash code to get the new index. If this index is also full, add the second hash code again, and repeat until a vacant position is found.

If you complete these methods correctly, you should see the following output when running **HashTableTest.java** with `testArray1()`.

Sample Output:

```
Linear Probing:
[ 37 50 54 __ __ __ 19 __ __ __ 49 11 24 ]
Quadratic Probing:
[ __ __ 37 54 __ __ 19 50 __ __ 49 11 24 ]
Double Hashing:
[ __ __ 54 50 __ 37 19 24 __ __ 49 11 __ ]
```

Question 3 (Sorting): Comparing performance [4 marks]

For this problem, you will test and evaluate different sorting algorithms by comparing their relative runtimes. Begin with the file **Sorting.java**. The code in this file is complete, but you should examine it anyway.

The main method creates a random array of the size specified by the int value `LENGTH`, then sorts it using one of several sorting methods - quick, merge, insertion, selection, and bubble sort. The sorting process is timed. When the array has been sorted, the length of the array and the amount of time taken to sort the array (in milliseconds) is printed. Then the `LENGTH` value is doubled, a new random array is created with the new size, and the sorting process is repeated. This process is performed as many times as the int value `RUNS` specifies. Note that running the whole loop may take several minutes, depending on the performance of your device, among other things. Once the loop has run the set number of times, the program then proceeds to sort the array using the next sorting method.

An Excel spreadsheet is provided for you: **Results.xlsx**. Your task is to run the file at least three times and collect the runtime data from each of these trials. Copy the results of each trial into the Excel spreadsheet. The average of the trials for each method at each array size should be

automatically calculated in the table below the space for trial results, and then a graph should be automatically constructed from the averages.

The average Big-O complexities of each of these algorithms are either $O(n^2)$ and $O(n \log(n))$. Examine the graph you've created and see if you can guess the Big-O complexities of each of these sorting algorithms. Note your hypothesis in the Excel spreadsheet. For full marks, make a text comment in the Excel spreadsheet below your hypothesis, explaining your thinking.

Question 4 (Minimum Spanning Tree): [6 marks (BONUS)]

The minimum spanning tree of a graph is an acyclic connected graph with the fewest possible edges (or if the graph is weighted, the smallest sum of edge weights) that can be made by removing edges of the given graph. In this question, you will write a program which takes a weighted graph and returns the graph's minimum spanning tree. To do this, we will use Prim's Algorithm.

Begin with the files **MSTGraph.java** and **MSTTest.java**. The MSTTest.java file will be used to test the methods written in the MSTGraph.java file. The test file calls MSTGraph methods to generate an undirected graph (either designed or random), print it, find its minimum spanning tree, and then print that as well. To make these functions work, you must complete two methods in the **MSTGraph.java** file.

Part A [1 mark]:

`printList()`: This method prints the graph's adjacency list. Note that the adjacency list no longer just stores the adjacent nodes, but rather stores Edges. You can see the class which defines this new object at the bottom of the **MSTGraph.java** file.

Part B [5 marks]:

`getMST()`: This method creates and returns a new MSTGraph which represents the minimum spanning tree of the original graph. You should use Prim's Algorithm to solve this problem, which you have learned about in lecture.

Essentially, this algorithm selects an arbitrary starting vertex (you should start with vertex 0), then evaluates each edge from this initial vertex. It selects the edge with the lowest weight that does not create a cycle and makes note of the vertex at the other end of the selected edge. This process is then repeated by searching for the lowest weighted edge from either of these two vertices that does not create a cycle. Once this next edge has been selected and the newly linked vertex saved, the process is repeated from the three visited vertices, and so on until every node in the new MSTGraph is connected.

There are four methods in the **MSTTest** class to help you test your getMST() method: `testGraph1()`, `testGraph2()`, `testGraph3()`, and `randomGraph()`. We recommend that you test with each of these methods individually. However, notice that the `randomGraph()` method will occasionally generate disconnected graphs. Read the comment in the `randomGraph()` method before using it.

Submission Instructions:

- Create a folder called "Lab8_<*student_ID* >" where <*student_ID* > is your student number/ID (e.g. **Lab8_12345678**). Only include the mentioned java files in the folder. **DO NOT** include any other files (e.g., .class, .java~ or any other files)
    - For Question 1, include your **Lab8Question1.pdf** file.
    - For Question 2, include your **HashTable.java** file.
    - For Question 3, include your **Results.xlsx** file.
    - For Question 4, include your **MSTGraph.java** file.
- Make a **zip file** of the folder and upload it to Canvas.
- To be eligible to earn full marks, your Java programs **must compile and run** upon download, without requiring any modifications.
- These assignments are your chance to learn the material for the exams. **Code your assignments independently.**

Congratulations on making it to the end of the semester, and good luck on your final exam!