Assignment 6

This assignment has 5 parts.

Part 1 (60 points) The goal of this assignment is to give students an opportunity to observe differences among three data structures in Java – HashMap, ArrayList, LinkedList – in terms of insertion time and search time.

Students are required to write a program that implements the following pseudocode:

create a HshMap instance myMap
create an ArrayList instance myArrayList
create a LinkedList instance myLinkedList

Repeat the following 10 times and calculate average total insertion time and average total search time for each data structure

generate 100,000 random integers in the range [1, 1,000,000] and store them in the array of integers keys[ ]

// begin with empty myHap, myArrayList, and myLinkedList each time

// Insert keys one at a time but measure only the total time (not individual insert
// time)
// Use *put* method for HashMap
// Use *add* method for ArrayList and Linked List

insert all keys in keys[ ] into myMap and measure the total insert time
insert all keys in keys[ ] into myArrayList and measure the total insert time
insert all keys in keys[ ] into myLinkedList and measure the total insert time

generate 100,000 random integers in the range [1, 2,000,000] and store them in the array keys[ ] (or in a different array).

// Search keys one at a time but measure only total time (not individual search
// time)
// Use *containsKey* method for HashMap
// Use *contains* method for ArrayList and Linked List

search myMap for all keys in keys[ ] and measure the total search time
search myArrayList for all keys in keys[ ] and measure the total search time
search myLinkedList for all keys in keys[ ] and measure the total search time

Print your output on the screen using the following format:

```
Number of keys = 100000
```

```
HashMap average total insert time = xxxxx
ArrayList average total insert time = xxxxx
LinkedList average total insert time = xxxxx

HashMap average total search time = xxxxx
ArrayList average total search time = xxxxx
LinkedList average total search time = xxxxx
```

You can generate *n* random integers between 1 and N in the following way:

```
Random r = new Random(System.currentTimeMillis() );
for i = 0 to n – 1
    a[i] = r.nextInt(N) + 1
```

When you generate random numbers, it is a good practice to reset the seed. When you first create an instance of the Random class, you can pass a seed as an argument, as shown below:

```
Random r = new Random(System.currentTimeMillis());
```

You can pass any long integer as an argument. The above example uses the current time as a seed.

Later, when you want to generate another sequence of random numbers using the same Random instance, you can reset the seed as follows:

```
r.setSeed(System.currentTimeMillis());
```

You can also use the *Math.random*( ) method. Refer to a Java tutorial or reference manual on how to use this method.

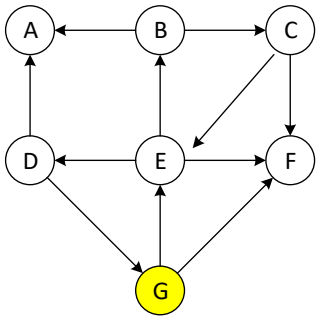
We cannot accurately measure the execution time of a code segment. However, we can estimate it by measuring an elapsed time, as shown below:

```
long startTime, endTime, elapsedTime;
startTime = System.currentTimeMillis();
// code segment
endTime = System.currentTimeMillis();
elapsedTime = endTime - startTime;
```

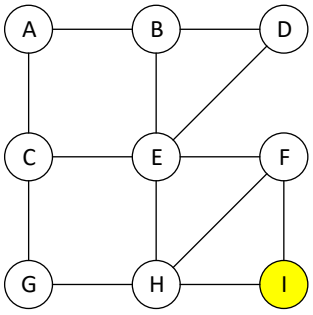
We can use the *elapsedTime* as an estimate of the execution time of the code segment.

Name the program *InsertSearchTimeComparison.java*.

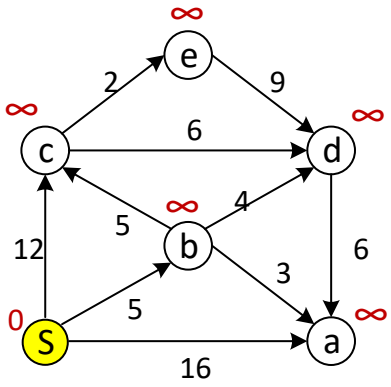
Part 2 (10 points). Run the DFS on the following graph beginning at node G and show the sequence of nodes generated by the search. When you have two or more choices as the next node to visit, choose them in the alphabetical order.



Part 3 (10 points). Run the BFS on the following graph beginning at node I and show the sequence of nodes generated by the search. When you have two or more choices as the next node to visit, choose them in the alphabetical order.

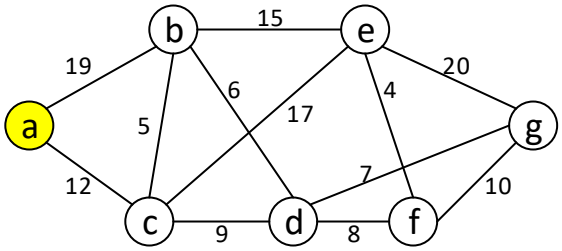


Part 4 (10 points). Run the Dijkstra's algorithm on the following graph beginning at node S.



Part 4-(1). After each iteration, show the D values of all nodes (initial D values are shown above each node in red).
Part 4-(2). Show the shortest path from S to every other node generated by the algorithm

Part 5 (10 points). Run the Prim-Jarnik algorithm on the following graph beginning at node *a*.



Part 5-(1). Show the sequence of nodes in the order that are brought into the "cloud."
Part 5-(2). Show the minimum spanning tree T, generated by the algorithm, as a set of edges.

Deliverables

Part 1.

- *InsertSearchTimeComparison.java* file
- A separate documentation that includes your conclusion/observation/discussion of this experiment.

Part 2 – Part 5

- *Hw6_others.pdf* file, which includes answers to Part 2, Part 3, Part 4, Part 5.

Combine the program file, additional files (if any), and the documentation file into a single archive file, such as a *zip* file or a *rar* file, and name it *LastName_FirstName_hw6.EXT*, where *EXT* is an appropriate file extension (such as *zip* or *rar*). Upload it to Blackboard.

Grading

Part 1:

- There is no one correct output. As far as your output is consistent with generally expected output, no point will be deducted.
- Up to 15 points will be deducted for wrong insert times
- Up to 15 points will be deducted for wrong search times.
- If your conclusion/observation/discussion is not substantive, points will be deducted up to 5 points.
- If there are no sufficient inline comments, points will be deducted up to 5 points.

Part 2: Up to 6 points will be deducted if your answer is wrong.
Part 3: Up to 6 points will be deducted if your answer is wrong.
Part 4

- Part 4-(1): Up to 3 points will be deducted if your answer is wrong.
- Part 4-(2): Up to 3 points will be deducted if your answer is wrong.

Part 5

- Part 5-(1): Up to 3 points will be deducted if your answer is wrong.
- Part 5-(2): Up to 3 points will be deducted if your answer is wrong.