Concordia University
Engineering and Computer Science

COMP 352: Data Structures and Algorithms
Winter 2021 - Assignment 4
Due date and time: Sunday April 18, 2021 by 23:59
This due date is firm and will not be extended

Heads-Up

- You must submit the answers to all the questions below. However, only one or more questions, possibly chosen at random, will be corrected and will be evaluated to the full 50 marks.
- The programming question can be done in a team of two, if you wish. However, the theory questions must be completed and submitted individually.

1 Written Questions (50 marks)

Question 1.

(a) Construct the BST that results when the following integers are inserted in the order given:

50, 20, 40, 55, 45, 30 10, 15, 35, 60, 5, 25

(b) Perform preorder, inorder, and postorder traversals of the BST above.

(c) Construct the BST that results when the following integers are deleted from the BST in the order given:

60, 30, 20

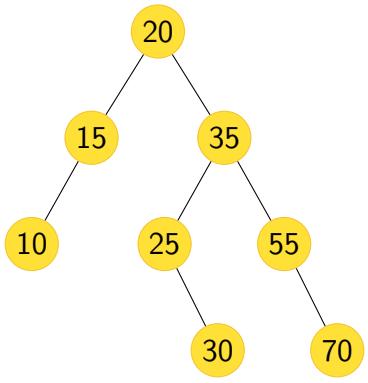
(d) Perform preorder, inorder, and postorder traversals of the BST in item (c) above.

Question 2.

For each of the following questions start with the AVL tree shown below.

Show the AVL tree that results from performing the following operations.

To be clear, in addition to preserving the BST property, you will need to preserve the AVL property of the tree by rebalancing it after **each** insertion and **each** deletion operation.



(a) Add 75 and then 65
(b) Add 75, 23, and then 32
(c) remove 15 and then 35
(d) remove 25 and then 30

Question 3.

Sort the following list of numbers by hand.

13, 11, 4, 40, 7, 34, 25, 17, 50, 17, 14

For each item (a), (b), and (c) below, show the list's contents each time it's elements are rearranged as well as showing all the steps you followed according to each sorting algorithm.

(a) insersion sort

(b) selection sort

(c) heapsort

(d) mergesort

Use Bottom-Up (Nonrecursive) Merge-Sort (Page 543) to sort the list in place. Here is an example of in place mergesort animation: Merge sort

To facilitate grading, your work should be arranged to look like Figure 12.1.b on page 533 but upside down and with boxes replaced by square brackets, as shown at right.

```
[85] [24] [63] [45] [17] [31] [96] [50]
[24  85] [45  63] [17  31] [50  96]
[24  45  63  85] [17  31  50  96]
[17  24  31  45  50  63  85  96]
```

(e) quicksort Select the pivot to be the last element in in the list.

To facilitate grading, your work should be arranged to look like Figure 12.14 Page 554 using square brackets and text instead of boxes and arrows, as shown at right.

Apply the same process to the left partition and then to the right partition.

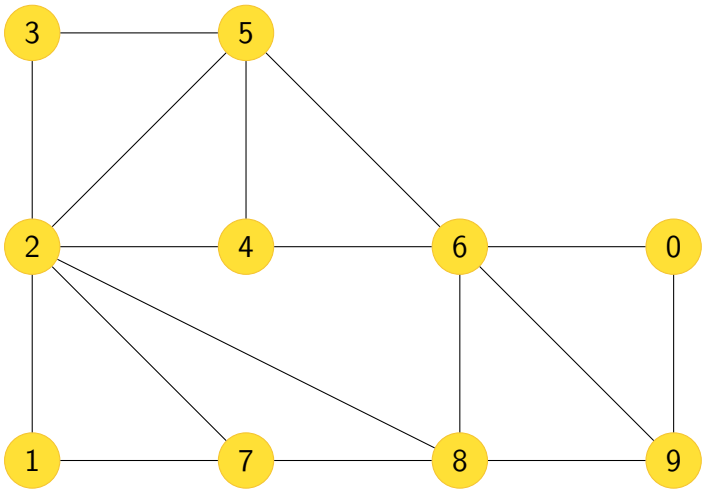
```
pivot = 50 at index 7
[85 24 63 45 17 31 96 50]

pivot = 50, L=0, R=6, swap
[31 24 63 45 17 85 96 50]

pivot = 50, L=1, R=4, no swap
[31 24 63 45 17 85 96 50]

pivot = 50, L=2, R=4, swap
[31 24 17 45 63 85 96 50]

pivot = 50, L=3, R=3, no swap
[31 24 17 45 63 85 96 50]

pivot = 50, L=4, R=3, exit loop
[31 24 17 45 63 85 96 50]

Swap pivot=50 and 63 at  L=4
[31 24 17 45] [63] [85 96 50]
```

Requirements

1. Each of methods above must sort the list in place, using no other auxiliary arrays, lists, etc.
2. In each case count the number of assignments and comparisons.

Question 4.

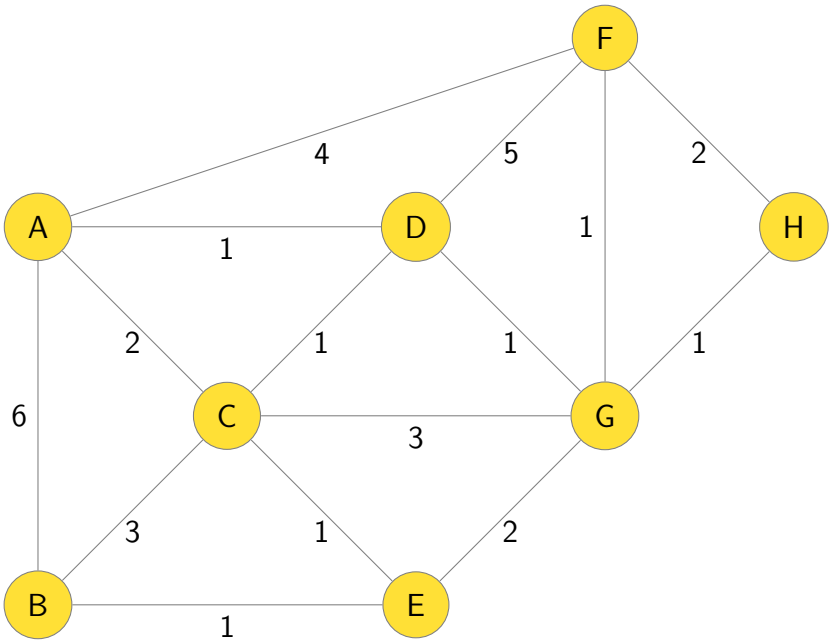
- Consider the graph shown below:



1. Give the adjacency matrix representing this graph.
2. Give the adjacency lists representing this graph.
3. Show the breadth-first search trees for the graph starting at node 0.
4. Show the depth-first search trees for the the graph starting at node 0.

Question 5.

- Consider the graph shown below:



1. Applying Kruskal's algorithm, find a minimal spanning tree for the graph.

   Use a table that includes column headers shown below (and possibly other columns of your choice) to show the progress of the algorithm at each step.

Step	Edge Considered	Cost	Accepted/Rejected

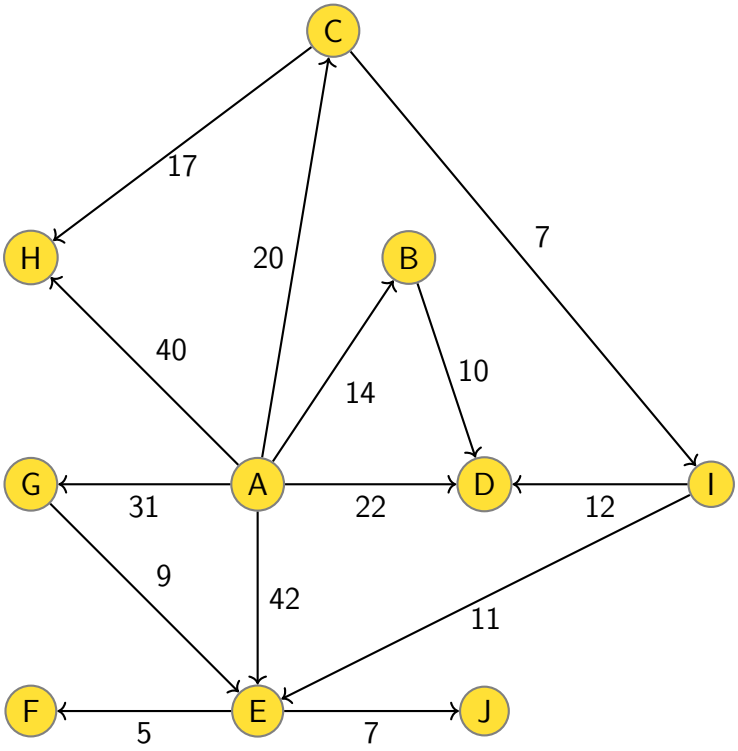
2. Applying Prim's algorithm, find a minimal spanning tree for the graph.

   Use a table that includes the following columns (and possibly other columns of your choice) to show the progress of the algorithm at each step.

Vertex	Visited?	Cost	Predecessor

Question 6.

Consider the graph shown below.



(a) Find depth-first and breadth-first topological orderings for the graph.

(b) Applying Dijkstra's algorithm, find the shortest distance from node A to every other node in the graph.

Use a table that includes the following columns (and possibly other columns of your choice) to show the progress of the algorithm at each step.

Vertex	Visited?	Cost	Predecessor

Programming Questions (50 marks)

In this assignment you will implement a map using a hash table, handling collisions via separate chaining and exploring the map's performance using hash table load factors. (The ratio $\lambda = n/N$ is called the load factor of the hash table, where N is the hash table capacity, and n is the number of elements stored in it.)

Class Entry

Write a class **Entry** to represent <**Key, Value**> entry pairs in the hash map. This will be a non-generic implementation. Specifically, **Key** is of type integer, while **Value** can be any type of your choice. Your class must include the following methods:

- A constructor that generates a new **Entry** object using a random integer (key). The value component of the <**Key, Value**> pair may be supplied as a parameter or it may be generated randomly, depending on you choice of the **Value** type.
- An override for class **Object**'s compression function **public int hashCode()**, using any of the strategies covered in section **10.2.1** (Hash Functions, page 411).

Abstract Class AbsHashMap

This abstract class models a hash table without providing any concrete representation of the underlying data structure of a table of "buckets." (See pages 410 and 417.)

The class must include a constructor that accepts the initial capacity for the hash table as a parameter, and uses the function $h(k) = k \mod N$ as the hash (compression) function.

The class must include the following abstract methods

size() Returns the number of entries in the map

isEmpty() Returns a Boolean indicating whether the map is empty

get(k) Returns the value **v** associated with key **k**, if such an entry exists; otherwise return null.

Put(k,v) if the map does not have an entry with key **k**, then adds entry **(k,v)** to it and returns null; else replaces with **v** the existing value of the entry with key equal to **k** and returns the old value.

remove(k) Removes from the map the entry with key equal to **k**, and returns its value; if the map has no such entry, then it returns null.

Class MyHashMap

Write a concrete class named **MyHashMap** that implements **AbsHashMap**. The class must use separate chaining to resolve key collisions. You may use Java's **ArrayList** as the buckets to store the entries.

For the purpose of output presentation in this assignment, equip the class to print the following information each time the method **put(k,v)** is invoked:

- the size of the table,
- the number of elements in the table after the method has finished processing **(k,v)** entry,
- the number of keys that resulted in a collision
- the number of items in the bucket storing **v**

Additionally,

- each invocation of **get(k)**, **put(k,v)**, and **remove(k)** should print the time used to run the method. If any **put(k,v)** takes an excessive amount of time, handle this with a suitable exception.

Class HashMapDriver

This class should include the following static void methods:

1. **void validate()** must perform the following:
   (a) Create a local Java.util **ArrayList** (say, **data**) of 50 random <**Key, Value**> pairs.
   (b) Create a **MyHashMap** object using 100 as the initial capacity (N) of the hash map.

   > **Heads-up**: you should never use a non-prime hash table size in practice, but do this for the purposes of this experiment.

   (c) Add all 50 entries from the **data** array to the map, using the **put(k,v)** method, of course.
   (d) Run **get(k)** on each of the 50 elements in **data**.
   (e) Run **remove(k)** on the first 25 keys, followed by **get(k)** on each of the 50 keys.
   (f) Ensure that your hash map functions correctly.

2. **void experiment_interpret()** must perform the following:
   (a) Create a hash map of initial capacity 100
   (b) Create a local Java.util **ArrayList** (say, **data**) of 150 random <**Key, Value**> pairs.
   (c) For $n \in \{25, 50, 75, 100, 125, 150\}$
      - Describe (by inspection or graphing) how the time to run **put(k,v)** increases as the load factor of the hash table increases, and provide reason to justify your observation.
      - If your **put(k,v)** method takes an excessive amount of time, describe why this is happening and why it happens at the value it happens at.

Note

- Feel free to include other methods of your own and choice if you feel they would facilitate your tasks in this assignment. Justify inclusion of such methods.

Deliverables

- A written specification of each of the classes you implemented, providing any information about design decisions.
- A written report with the trial run of **validate()**, and answers to questions in item (c) of **experiment_interpret()**.
- Well-formatted and documented Java source code

Answers to Theory Questions

For all questions, including the programming questions, the parts that require written answers, pseudo-code, graphs, etc., you can express your answers into any number of files of any format, including image files, plain text, hand-writing, Word, Excel, PowerPoint, etc.

However, no matter what program you are using, you must convert each and every file into a PDF before submitting. This is to ensure the markers and you have exact same view of your work regardless of the original file formats.

Whether or not you are in a team, you must each submit a copy of your written answers.

Solutions to Programming Question

Developing your programming work using a Java IDE such as NetBeans, Eclipse, etc., you are required to submit the project folder(s) created by your Java IDE, together with any input files used and output files produced by your program(s).

If you are in a team, you must to submit only ONE copy of your program project folders(s) and input/output files.

What and How to Submit

1. Letting the # character denote the assignment number, create a folder as follows:
   (a) If your working individually, name your folder `A#_studentID`, where `studentID` denotes your student ID. Then, copy the PDF files you prepared for the theory questions together with your project folder(s) and input/output files to the `A#_studentID` folder.
   (b) If you are working in a team, name your folder `A#_studentID1_studentID2`, where `studentID1` denotes the ID number of the student who is responsible for submitting the programming solutions for both of you, and `studentID2` denotes the ID number of the other team member. Then, the student whose ID number is `studentID1` must follow item (a), as if she/he completed the programming question individually. The other team member must also follow item (a) above, but she/he must NOT include the team's programming solutions.
2. Compress the folder you created in (1) into a zip file with the same name as that of the folder being compressed.
3. Upload the compressed file you created in item (2) to Moodle.

1.1 Last but not Least

To receive credit for your programming solutions, you must demo your program to your marker, who will schedule the date and time for the demo for you (please refer to the course outline for full details). If working in a team, both members of the team must be present during the demo. Please notice that failing to demo your programming solution will result in zero mark regardless of your submission.