Concordia University
Engineering and Computer Science

COMP 352: Data Structures and Algorithms
Winter 2021 - Assignment 3

Due date and time: Friday March 26th, 2021 by midnight

1. **Theory Questions (50 marks)**

Please read carefully: You must submit the answers to <u>all</u> the questions below. However, only a couple of questions, possibly chosen at random, will be corrected and will be evaluated to the full 50 marks.

<u>Question 1</u>

a) Draw a <u>single</u> binary tree that gave the following traversals:

Inorder: S A E U Y Q R P D F K L M
Preorder: F A S Q Y E U P R D K L M

a) Assume that the binary tree from the above- part (a)- is stored in an array-list as a complete binary tree as discussed in class. Specify the contents of such an array-list for this tree.

<u>Question 2</u>

Draw the min-heap that results from the bottom-up heap construction algorithm on the following list of values:

`10, 27, 15, 35, 42, 19, 45, 16, 12, 8, 18, 14, 13, 9, 20, 11, 13`

Starting from the bottom layer, use the values from left to right as specified above. Show <u>all</u> the steps and the final tree representing the min-heap. Afterwards perform the operation `removeMin` four (4) times and show the resulting min-heap after each step.

<u>Question 3</u>

Assume a hash table that utilizes an array of 13 elements and where collisions are handled by <u>*separate chaining*</u>. Considering the hash function is defined as: $h(k)=k \bmod 13$.

a) Draw the contents of the table after inserting elements with the following keys:
`{32, 147, 265, 195, 207, 180, 21, 16, 189, 202, 91, 94, 162, 75, 37, 77, 81, 48}`
b) What is the maximum number of collisions caused by the above insertions?

<u>Question 4</u>

Assume an <u>*open addressing*</u> hash table implementation, where the size of the array $N = 19$, and the <u>*double hashing*</u> is performed for collision handling. The second hash function is defined as: $d(k) = q - k \bmod q$, where k is the key being inserted in the table and the prime number $q = 11$. Use simple modular operation ($k \bmod N$) for the first hash function.

a) Show the content of the table after performing the following operations, in order:
`put(42), put(19), put(48), put(20), put(72), put(18), put(48),put(27), put(9).`
b) What is the size of the longest cluster caused by the above insertions?
c) What is the number of occurred collisions as a result of the above operations?
d) What is the current value of the table's *load factor*?

Question 5

Assume the utilization of *linear probing* instead of *double hashing* for the implementation given in Question 4. Still, the size of the array `N = 19`, and that simple modular operation (`k mod N`) is used for the hash function.

a) Show the contents of the table after performing the following operations, in order: `put(29), put(53), put(14), put(95), remove(53), remove(29), put(32), put(19), remove(14), put(30), put(12), put(72).`
b) What is the size of the longest cluster caused by the above insertions? Using Big-O notation, indicate the complexity of the above operations.
c) What is the number of occurred collisions as a result of the above operations?

2. Programming Problem (50 marks)

In this programming part, you will implement a standard *Priority Queue* interface using four basic strategies and then evaluate and compare their operational performance.

Storing a collection of prioritized elements, a *Priority Queue* is an ADT characterized as follows:

- In general, the elements are ordered pairs[1] *(k,v)* with key *k* prioritizing value *v* in the priority queue.
- It can store elements of any type, as long as the keys satisfy a **total ordering** relationship. (See text book, page 363)
- It supports arbitrary insertions of elements.
- It allows the removal of the element that has highest priority.
  - In this assignment :
    - an element with the smallest key has the highest priority.
    - if multiple elements have the same priority, then the removal process may arbitrary remove any of those elements.
- It supports the following MyPQ<K,V> interface (Page 361):
  - *insert(k, v):* Creates an entry (element) with key *k* and value *v* in the priority queue.
  - *min( ):* Returns (but does not remove) a priority queue entry *(k,v)* having minimal key; returns null if the priority queue is empty.
  - *removeMin( ):* Removes and returns an entry *(k,v)* having minimal key from the priority queue; returns null if the priority queue is empty.
  - size( ): Returns the number of entries in the priority queue.
  - isEmpty( ): Returns a boolean indicating whether the priority queue is empty.

Programming Requirements:

1. Implement the MyPQ<K,V> interface above four ways: array-based and linked list-based, each sorted and unsorted.

   Specifically, implement the following four classes.

   a) Array-Based
      i. MyPQUnsortedArray<K,V>

[1] For example, Emergency Room, (P_i, Patient Record), where P_i= 1, 2, 3, …

1. Uses an array to store the elements, with array's initial capacity = 1.
2. During a call to *insert(k, v)*, immediately knows where to insert an element in the array,
3. During a call to *removeMin( ) or min( ),* searches the array for the element with the smallest key.
4. Before inserting an element during a call to *insert(k, v)*, the array capacity is doubled if it is full.
5. After removing the smallest element during a call to *removeMin( )*, the array capacity is halved if its size is less than 25% of the capacity.

ii. MyPQSortedArray<K,V>
1. Uses an array to store the elements, with array's initial capacity = 1.
2. During a call to *insert(k, v)*, search among the sorted elements in the array for an appropriate position to insert the element there.
3. During a call to *removeMin( ) or min( ),* immediately knows where the element with the smallest key is located.
4. Before inserting an element during a call to *insert(k, v)*, the array capacity is doubled *)* if it is full.
5. After removing the smallest element during a call to *removeMin( )*, the array capacity is halved if its size is less than 25% of the capacity.

b) Linked List-Based
i. MyPQUnsortedList<K,V>
1. Uses a doubly linked list to store the elements,
2. During a call to *insert(k, v)*, immediately knows where to insert an element in the array,
3. During a call to *removeMin( ) or min( ),* searches the array for the element with the smallest key.

ii. MyPQSortedList<K,V>
1. Uses a doubly linked list to store the elements,
2. During a call to *insert(k, v)*, search among the sorted elements in the list for an appropriate position to insert the element there.
3. During a call to *removeMin( ) or min( ),* immediately knows where the element with the smallest key is located.

2. Your implementations above may not use the Java Collections class and the java collection framework; that is, you are expected to roll out you own code! It seems like a lot, but as you can see it repeats the same theme four times. Please note that your array-based and linked list-based classes do not need to be as expansive as Java's ArrayList or LinkedList classes beyond what is the bare minimum to support the MyPQ<K,V> interface.

3. Write a program named PQTester to test drive your classes as follows:

a) For each N in `{10, 100, 1000, 10000, 100000, 1000000}`
i. Using the input files posted with this assignment, prepare an input file:
1. If N is in the range `[1, 10000]` use **`elements_test_file1`**, which has `10000` lines of strings.
2. If N is in the range `[10000, 100000]`, then use **`elements_test_file2`**, which has `100000` lines of strings.

3. If N is in the range [100000, 1000000], then use **elements_test_file3**, which has 1000000 lines of strings.

ii. Starting with four empty priority queue objects, one of each of your classes above, measure how long it takes to insert *N* elements of type <Integer, String> into each queue, where the Integer keys are random integers in the range [1,N], and the String values come from the input file prepared above.
iii. Using the priority queues created in part a, measure how long it takes to remove all of the N elements.
iv. Create and fill a table of values that looks as follows (the (ms) timing values below are just an illustration and do not relate to any real measurements):

N = 10	Insert(k,v) (ms)	RemoveMin(k,v) (ms)
MyPQUnsortedArray	15	35
MyPQUnsortedList	45	11
MyPQSortedArray	15	71
MyPQSortedList	48	86

v. Save the table to a file named **pqtestrun.txt.**
vi. Make sure you reset the timer (or save the intermediate time before the next measurement). In other words, make sure that your measured time contains only the time to perform one set of operations (insert or remove) that was supposed to be timed

4. In case the operations for big N numbers take too long (e.g., more than 50s) you may reduce the number to a smaller one or eliminate it (so that you will have a range from, say, 1 to 100000)

5. As always, it is imperative that you test your classes, making sure that your program can handle boundary cases (e.g., calling *min*() or *minRemove*() on an empty queue) and other error conditions.

6. Remember to submit your **pqtestrun.txt** output file with your other deliverables

1. Deliverables

In this assignment, the programming question can be done in a team of two, if you wish. However, the theory questions must be completed and submitted individually.

1.1 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.

1.2 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 submit only ONE copy of your program project folders(s) and input/output files.

1.3 What and How to Submit

1. Letting the # character denote the assignment number, create a folder as follows:
   a) If you are 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.

1.4 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.