Assignment 1

Jackson Wiebe 3519635

December 15, 2021

Answer question 1, question 2, and **any other 2** questions from questions 3 to 6 – maximum 100 marks. You must score at least 50 to pass the assignment.

1. (25 + 15 = 40 marks) You have learned some fundamental data structure concepts such as array, queue and priority queue, stack, list and linked list, sequence, and unordered set, and you understand the concept of interface or abstract data type that defines the set of operations supported by a data structure and the semantics, or meaning, of those operations. You can use the interface of one particular data structure to define or implement the operations of a different data structure.
   1. (25 marks total) Describe the meaning of the essential methods add(x), deleteMin(), and size() that are supported by the priority queue interface (5 marks).   
      Implement those methods using a singly-linked list (5 marks for each method).   
      Analyze the running time of the add(x) and deleteMin() operations based on this implementation (5 marks).

Please see attached executable demonstration file Q1a.java for direct implementation using java. I felt it best to write my own implementation to demonstrate the requested knowledge.

**Add(x)**

With my implementation of a priority queue with singly linked lists the head of the list is always maintained as the 0 slot; or highest priority queue object. The new information is added where the smallest integer is in slot 0 and the remainder of the list is linked from there. Priority is given as an integer where the lower numbers have higher priority. In my implementation other data could be added to the Node object for later processing by whatever program is using the PQ.

Worst case complexity is O(n) since we need to traverse the entire list for insertion. Best case complexity will be O(1) since it gets inserted to the head.

**deleteMin()**

This method returns the highest priority item from the queue and deletes it from the linked list. In my implementation the head of the list is also our highest priority (given by a low integer). As such, removing this Node is simply done by assigning head to the next item. We also return a copy of the item to the user if they wish to do some processing like displaying/processing the data. In Java the garbage collector will free the memory however in other languages some release of the memory may be needed to avoid bloat.

Compute complexity for this operation is O(1) in all cases since the head of the list is always the highest priority. Only 1 loop of the function is needed.

**size()**

This method simply returns an integer holding the current count of the queue. The private variable size is incremented and decremented by the above methods Add(x) and deleteMin() as objects are added to the list. If data corruption was a concern this method could also be made more advanced by traversing the list and counting items individually however that would quickly reduce the efficiency from O(1) to O(n).

* 1. (15 marks total) Implement the stack methods push(x) and pop() using two queues (5 marks for each method).   
     Analyze the running time of the push(x) and pop() operations based on this implementation (5 marks).

Please see attached executable demonstration file Q1b.java for direct implementation using java. I felt it best to write my own implementation to demonstrate the requested knowledge.

**Push(x)**

From my research the best way to implement this was keeping Queue as a “Virtual” stack wherein it would always maintain our desired object order. Queue #2 would be used for adding new items and reversing the order. For adding new items to the stack: Move all the existing items to stack #2, add the new item, them reverse the order back into Queue #1 so to maintain the desired structure.

Complexity here will be O(2n) in most cases. To add an item to the virtual stack we must move all the data to a secondary queue, add the item, then move all the data back, 2 full operations for each item. The only exception will be where only 1 item is stacked since it can be directly placed into Stack#1

**Pop()**

In my implementation the stack is always represented in queue 1. We simply need to remove the first element in queue 1. The complexity of this operation will always be O(1) since only one step is required.

1. (10 + 10 = 20 marks) Swap two adjacent elements in a list by adjusting only the links (and not the data) using
   1. singly-linked list (10 marks).

See my implementation in file Q2a.java. First we establish a linked list with a head and 5 tail elements. We print out the list to confirm the data. Then we swap nodes 2 and 3 by changing the respective nextPtr. And again print the list to confirm the change. 3 pointer changes are need to swap the nodes. See drawing below.

* 1. doubly-linked list (10 marks).

See my implementation in file Q2b.java. This is done in much the same method as the first swap only we need to handle the extra reverse links. First we establish a linked list with a head and 5 tail elements. We print out the list to confirm the data. Then we swap nodes 2 and 3 by changing the respective nextPtr and previousPtr variables. And again print the list to confirm the change. 6 pointer changes are need to swap the nodes. Alternative to my implementation you could also traverse the list with only forward links and set the reverse link with each traverse however that would add O(n) complexity as the whole list would need to be traversed. See drawing below.

1. (20 marks) Exercise 1.5. Using a USet, implement a Bag. A Bag is like a USet—it supports the add(x), remove(x), and find(x) methods—but it allows duplicate elements to be stored. The find(x) operation in a Bag returns some element (if any) that is equal to x. In addition, a Bag supports the findAll(x) operation that returns a list of all elements in the Bag that are equal to x.

I would like some clarification on this question. From what I understand, the task is to create a bag using a unique set. From my understanding in Java this could be implemented with a HashMap where the Key represents our unique data and Value represents the count of unique data. For an add(x) operation we would search the HashMap for that item and increment its counter (or add as new unique). And FindAll(x) simply finds the unique data and returns the total count represented by the Value of that hash set.

Am I correct in my understanding of the above question?

1. (20 marks) Exercise 2.3. Design and implement a *RandomQueue*. This is an implementation of the Queue interface in which the remove() operation removes an element that is chosen uniformly at random among all the elements currently in the queue. (Think of a RandomQueue as a bag in which we can add elements or reach in and blindly remove some random element.) The add(x) and remove() operations in a RandomQueue should run in constant time per operation.

See implementation in Q4.java.

My code generates a new Random Queue with code from question 1. Adding items are done with O(1) complexity by simply adding new items to the tail of our linked list. Removing items are done by randomly picking an item from 0 to size and iterating from the tail until we find the item to return. I did not find a way to have O(1) complexity for removing a random item. At maximum we will have O(n) complexity since we may need to search the entire list for our item.

1. (20 marks) Exercise 3.12. Write a method, reverse(), that reverses the order of elements in a DLList.

I did not complete this however I would like to verify if my thought process is correct. To implement this task I would create a new doubly linked list. The sudo code would be to remove objects from the head of the first list setting a new head each time. Then adding that object to the second list. Added objects would become the new head effectively pushing items down the list. This would have a net effect of a new list in reverse order.

1. (20 marks) Exercise 3.14. Design and implement a MinStack data structure that can store comparable elements and supports the stack operations push(x), pop(), and size(), as well as the min() operation, which returns the minimum value currently stored in the data structure. All operations should run in constant time.

I was unsure if the min() function was intended to remove the object from the stack or only peek at the object but leave it in the stack. My implementation is a peek and leaves the object in the stack.

See file Q6.java for implementation.

My general approach to this stack was to keep a linked list of objects built from a Top object. Thus the pop and push functions would only need to deal with the top object of the list. The size() function uses a counter and the min value is discussed below.

Since a stack will always be built or removed from one end we can keep track of the minimum value Thus far in each object. This will result in the top object containing the minimum value of the entire stack. As we add new objects they will hold the minimum value of the entire stack. And visa versa as objects are removed the top object will always hold the min for the current stack of objects.

I implemented this by adding a new value min to my LLNode object and a new function that could store the minimum of 2 given values. As we build our stack the object will default to its created value. The interface will then feed the minimum value thus far for comparison and storing.