CSCI 3230, Fall 2018 Assignment 2

Posted: September 1 **Due before 11:00 pm, Friday, September 21, 2018**

Upload to "Assignment 2" folder in Folio Dropbox with file name "**Lastname_ID.pdf**".

(Mandatory assignment cover-sheet; without it, your work will not be marked.)

Submitting student: _____ Your mark: ☐ out of 100

Collaborating classmates: _____

Other collaborators: _____

References other than the textbook and handouts:

_____

_____

_____

_____

Regardless of the collaboration method allowed, you must always properly acknowledge the sources you used and people you worked with. Your professor reserves the right to give you an exam (oral, written, or both) to determine the degree that you participated in the making of the deliverable, and how well you understand what was submitted. For example, you may be asked to explain any solution that was submitted and why you choose to write it that way. This may impact the mark that you receive for the deliverable.

So, whenever you submit a deliverable, especially if you collaborate, you should be prepared for an individual inspection/walkthrough in which you explain what every line of assignment does and why you choose to write it that way.

1. **(20 points)** Answer "True" or "False":

   (a) Storing information in a doubly linked list structure is a bad idea if we plan to do a lot of insertions.

   (b) Storing information in a doubly linked list structure is a bad idea if we plan to do a lot of deletions.

   (c) Storing information in a doubly linked list structure is a bad idea if we plan to print the entire list frequently.

   (d) An array implementation of a queue is more difficult to manage than the array implementation of a stack.

   Solution:

   (a) False

   (b) False

   (c) False

   (d) True

2. **(20 points)** Describe how to implement the **stack ADT** using a **single queue** as an instance variable, and only constant additional local memory within the method bodies. What is the running time of the `push()`, `pop()`, and `top()` methods for your design?

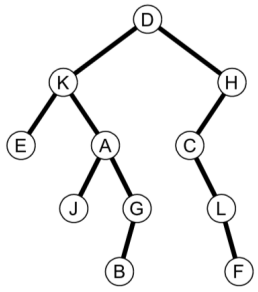
   Solution:

   One approach to implement the stack ADT using a queue Q, simply enqueues elements into Q whenever a push call is made. This takes $O(1)$ time to complete. For pop calls, we make $n-1$ calls to `Q.enqueue(Q.dequeue())`, where n is the current size, and then return `Q.dequeue()`, as that is the most recently inserted element. This requires `O(n)` time. We can use a similar approach for top, but rotating the answer back to the end of the queue.

   A better approach is to align the elements of the queue so that the "top" is aligned with the front of the queue. To push an element, we enqueue it and then immediately make $n-1$ calls to `Q.enqueue(Q.dequeue())`. This requires $O(n)$ time, but with that orientation, both pop and top can be implemented in $O(1)$ time by a respective call to dequeue or front.

3. **(20 points)**

(a) (2 points) What is the preorder traversal of the following binary tree?



(b) (2 points) What is the inorder traversal of the above binary tree?

(c) (2 points) What is the postorder traversal of the above binary tree?

(d) (2 points) What is the minimum and maximum number of nodes that a binary tree with height of h can contain? Write the number of nodes in terms of h.

(e) (6 points) Write a recursive function that returns the height of a binary tree.

(f) (6 points) Write a recursive function that returns a count of the number of leaf nodes in a binary tree.

Construct testing examples to test your code.

(Include screenshots of your return results and submit your **java code** in a separate file.)

Solution:

(a) DKEAJGBHCLF

(b) EKJABGDCLFH

(c) EJBGAKFLCHD

(d) Min: h, Max: $2^h - 1$

(e)
```
int height(BinNode node) {
  if (node == NULL) return 0;  // Empty subtree
  return 1 + max(height(node.left()), height(node.right()));
}
```

(f)
```
int countLeaf(BinNode node) {
  if (node == NULL) return 0;  // Empty subtree
  if (node.isLeaf()) return 1; // A leaf
  return 1 + count(node.left()) + count(node.right());
}
```

4. **(20 points)** Consider the following code fragment for **priority queue**.

```
// delMax() deletes the largest element in current queue.
MaxPQ<Integer> pq = new MaxPQ<Integer>();
int N = a.length;
for (int i = 0; i < N; i++) {
   pq.insert(a[i]);
   if (pq.size() > k) pq.delMax();  /* MARK */
}
for (int i = 0; i < k; i++)
   System.out.println(pq.delMax());
```

Assume that $a[]$ is an array of integers, *MaxPQ* is implemented using a binary heap, and $N \geq k \geq 1$.

(a) What does it output?

(b) What is the order of growth of its worst-case running time. Circle the best answer:

$$k \log k \quad k \log N \quad N \log k \quad N \log N \quad N^2$$

(c) Now suppose the marked line was deleted. What does it output?

(d) Now suppose the marked line was deleted. What is the order of growth of its worst-case running time. Circle the best answer:

$$k \log k \quad k \log N \quad N \log k \quad N \log N \quad N^2$$

Solution:

(a) Prints the k smallest values in descending order.

(b) $N \log k$. The maximum size of the heap is k.

(c) Prints the k largest values in descending order.

(d) $N \log N$. The maximum size of the heap is N.

5. **(20 points)** Consider the following binary max heap (*i.e.*, the array-representation of a heap-ordered *complete* binary tree). Here assume $A < B < \ldots < Z$.

0	1	2	3	4	5	6	7	8	9	10	11	12
Z	W	Y	T	G	K	V	R	S	F	A	-	-

(a) Delete the maximum key. Give the resulting binary heap. Circle those values that changed.

0	1	2	3	4	5	6	7	8	9	10	11	12

(b) Insert the key X into the **original** binary heap. Give the resulting binary heap. Circle those values that changed.

0	1	2	3	4	5	6	7	8	9	10	11	12

Solution:

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12
Y	W	*V*	T	G	K	*A*	R	S	F	*-*	-	-

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12
Z	W	Y	T	G	*X*	V	R	S	F	A	*K*	-