CS2040 2021/2022 Semester 2 Midterm

MCQ

This section has 10 questions and is worth 30 marks. 3 marks per question.

Do all questions in this section.

1. You are given an empty hash table of size 11 which uses quadratic probing and hash function h(key) = key % 11. The following integer keys were then inserted one at a time into the hash table, though the exact order in which the keys were inserted is unknown:

   12, 21, 22, 36, 47, 50, X

   where X is an unknown integer. No deletions were involved throughout the process.

   After inserting all keys, the hash table now looks as follows:

Slot	0	1	2	3	4	5	6	7	8	9	10
Key	22	12		47	X		50	36			21

   Which of the following could be valid values of X? A value of X is considered valid if there is a sequence of keys to be inserted that can produce the hash table above.

   i. 25
   ii. 32
   iii. 39

   a. **i only**
   b. i and iii
   c. ii and iii
   d. i, ii and iii

Consider the slots which each key is initially mapped to, and the slot which they actually occupy in the hash table to determine the probing sequence that occurs for the keys to reach its eventual position:

12: 1 (1)
21: 10 (10)

22: 0 (0)
36: 3 (3 -> 4 -> 7)
47: 3 (3)
50: 6 (6)

Notice that for the key 36 to be in slot 7, slot 4 must already be occupied. Therefore, X must have been inserted before key 36.

Now, consider the slots which each option for X are initially mapped to, and the probing sequence needed for them to be put into slot 4:

25: 3 (3 -> 4)
32: 10 (10 -> 0 -> 3 -> 8 ->4)
39: 6 (6 -> 7 -> 10 -> 4)

If X is 25, there is no issue (slot 3 is easily occupied by key 47, with no collisions).
If X is 32, there is an issue (slot 8 is empty, so it is impossible for this probing sequence to occur).
If X is 39, there appears to be no issue at first (slots 6, 7 and 10 are all occupied). However, for slot 7 to be occupied, the probing sequence requires that slot 4 is already occupied (ie. X has been inserted already), and therefore it is impossible for X to be 39.

2. You are given a hash table of size 7, which contains 11 distinct keys. No further information about the hash table is provided. The load factor of the hash table (rounded to 2 decimal places) is:

a. 0.64
b. 0.85
c. 1.57
d. It is impossible for such a hash table to exist

11 keys / 7 slots = 1.57 load factor. The load factor exceeding 1 can be explained by the use of separate chaining.

3. You are given the following integer array:

Index	0	1	2	3	4	5	6	7
Value	3	7	8	10	1	14	16	17

Which of the following sorting algorithms (as taught in lecture) would be most suited to sorting this specific array in ascending order?

a. Selection Sort
b. Insertion Sort
c. Improved Bubble Sort
d. Quick Sort

Selection sort always takes $O(n^2)$ time regardless of input.

Insertion sort would not need to perform any swaps, except for when 1 is selected as the next element to be sorted. Traversing the entire array takes O(n) time, and it takes at worst O(n) time to move 1 to the front of the array. Total time taken is O(n).

Improved bubble sort would need about n/2 iterations to move 1 to the front of the array (since 1 is near the middle of the array). Running n/2 iterations of bubble sort would take $O(n^2)$ time.

Quick sort would run in $O(n^2)$ time, as running the partition method would result in the right partition shrinking by 1 to 2 elements each time (array is almost sorted). Even assuming that the right partition always shrinks by 2 elements each time, the total time taken to sort the right partition would be $n + (n-2) + (n-4) + \ldots = O(n^2)$. The time taken to sort the left partition would be largely negligible owing to its small size.

4. You are given the following linked lists which contain **n** elements (where **n** is at least 4). You want to remove both the head and the tail of the linked list. Which of these linked lists **cannot** perform both operations correctly in O(1) time?

i. Basic linked list
ii. Tailed linked list
iii. Circular linked list

All lists are assumed to use the lecture implementation/descriptions.

a. i only
b. i and ii
c. ii and iii
d. i, ii and iii

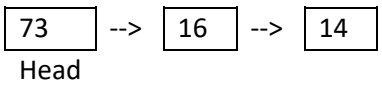
A basic linked list cannot remove the tail in O(1) time, as it needs to iterate over the entire length of the list to reach the tail, taking O(n) time.

A tailed linked list cannot remove the tail in O(1) time, as it has to find the second last node in the list (and set the next pointer of this node to null). This cannot be sped up by using the tail pointer, and once again we will need to iterate starting from the head to reach this node, taking O(n) time.

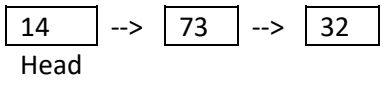
For similar reasons, a circular linked list cannot remove the tail in O(1) time, as finding the second last node in the list will require iterating from the head, taking O(n) time.

PS: this question was inspired by bread loaves, where the head/tail slices tend to be removed by people who don't like them

5. You are given the following basic linked list, which consists of three singly linked ListNodes:



You want to change the linked list to the form below:



Each ListNode has an **item** and a **next** attribute. A **head** variable has also been initialised to point to the head node (initially, the node containing 73) correctly, and must be updated correctly.

Which of the following code fragments will be able to change the linked list correctly?

a.
```
head.next.next.item = 32;
head.next.next.next = head;
head = head.next.next;
head.next.next.next = null;
```

b.
```
head.next.item = 32;
head.next.next = null;
head.next.next.next = head;
head = head.next.next;
```

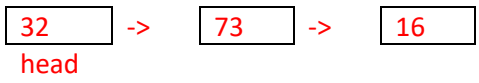
c.
```
head.next.next.next = head;
head.next.item = 32;
head = head.next.next;
head.next.next.next = null;
```

d.
```
head.next.next.next = head;
head = head.next.next;
```

```
head.next.next.next = null;
head.next.item = 32;
```

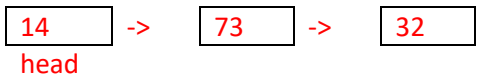
The results of each code fragment is shown below:

a.

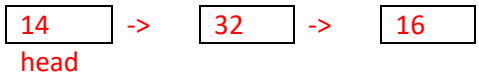


b. Null pointer exception on 3rd line

c.



d.



6. Given an array of **N** integers (which may not be distinct) sorted in ascending order, you want to find the integer that appears the most frequently in the array. It is known that this integer appears in more than 50% of the array (and therefore, there are no ties for the most frequently appearing integer). This can be done in **worst case**:

   a. O(1) time
   b. O(logN) time
   c. O(N) time
   d. O(NlogN) time

Simply return the element at index (n+1)/2. If n is odd, this element is exactly in the middle of the array. If n is even, this element is the element just to the right of the middle of the array (notice that for an even array, the middle of the array is in between two elements).

Since the array is sorted, it means that integers with the same value must appear next to each other in one continuous segment of the array. For such a segment to take up > 50% of the array, it must cover this particular element (towards the middle of the array) regardless of where the segment starts and ends.

7. What is the time complexity of the following method?

```
int someIt(int n) {

    int m = 1;

    while (n > 0) {
        n = n - m;
        m = m + 2;
    }
    return m / 2;
}
```

a. O(log n)
b. O($\sqrt{n}$)
c. O($\sqrt[3]{n}$)
d. O(n)

It can be seen that the value of n is subtracted by a constantly increasing number. We can attempt to track the total value which n was subtracted by at each iteration:

Iteration number	0	1	2	3	4
Value of m (before increment)	1	3	5	7	9
Total value n was subtracted by	$1 = 1^2$	$4 = 2^2$	$9 = 3^2$	$16 = 4^2$	$25 = 5^2$

Notice that the total value n was subtracted by is equal to i^2, where i is the number of iterations of the while loop. As such, the loop terminates for the first integer value of I where $i^2 >= n$. By taking the square root of both sides, we can see that the number of iterations is bounded by O($\sqrt{n}$)

8. What is the time complexity of the following method?

```
int someRe(int n) {

    int i = n;
    int ans = 0;

    while (i > 1) {
        ans = ans + someRe(n/2);
        i = i / 2;
    }
    return ans;
}
```

a. O(log n)
b. O(log(n!))
c. O((log n)!)
d. O((log n)^2)

Let x be log(n). We can see that at the first level, there is only one call (parameter value n). At the second level (parameter value n/2), there are x calls. At the third level (parameter value n/4), there are (x * (x-1)) calls. This pattern repeats until we hit the last level, which has (x * (x-1) * … * 1) calls. Therefore, the total number of calls is:

$$\frac{x!}{x!}+\frac{x!}{(x-1)!}+\frac{x!}{(x-2)!}+\cdots+\frac{x!}{2!}+\frac{x!}{1!}+\frac{x!}{0!}$$

This can be rewritten as:

$$x!\left(\frac{1}{x!}+\frac{1}{(x-1)!}+\frac{1}{(x-2)!}+\cdots+\frac{1}{2!}+\frac{1}{1!}+\frac{1}{0!}\right)$$

The part in brackets approaches a constant. Recall from tutorials that

$$\left(\frac{1}{2^x}+\frac{1}{2^{x-1}}+\frac{1}{2^{x-2}}+\cdots+\frac{1}{2^2}+\frac{1}{2^1}+\frac{1}{2^0}\right)$$

Approaches a constant, and that the growth rate of x! is greater than 2^x, and therefore, the summation of

$$\left(\frac{1}{x!}+\frac{1}{(x-1)!}+\frac{1}{(x-2)!}+\cdots+\frac{1}{2!}+\frac{1}{1!}+\frac{1}{0!}\right)$$

Must be bounded by this constant, and therefore can be considered a constant itself. Therefore, the total time taken in terms of x is O(x!). Rewriting this in terms of n, the time complexity is O((log n)!)

9. You are given a circular linked list (lecture implementation) consisting of at least 2 nodes. Each node in the list contains distinct integers. Unfortunately, one of the nodes was corrupted, and its .next attribute was set to *null*. We now want to fix the linked list such that the node correctly points to the next node in the list. This can be done in worst case:

a. O(1) time
b. O(log n) time
c. O(n) time
d. Impossible to perform this operation in general except for some special cases

This cannot be done in general. While it is possible to find the node with the corrupted `.next` attribute, it is impossible to find the other node which it should point to, owing to the lack of a previous pointer to find this node (unless it's the tail and so we know next should point to head).

10. Which of the following operation names are not related to stacks or queues?

a. offer()
b. take()
c. peek()
d. pop()

Mostly self explanatory. offer() is a queue operation, pop() is a stack operation while peek() can be used for both stacks and queues.

Analysis

This section has 3 questions and is worth 12 marks. 4 marks per question.

Please select True or False and then type in your reasons for your answer.

Correct answer (true/false) is worth 2 marks.

Correct explanation is worth 2 marks. Partially correct explanation worth 1 marks.

Do all questions in this section.

11. Even though quadratic probing might not be able to find an empty slot for insertion when load factor >= 0.5, we can still determine when we will not find an empty slot and terminate the insertion instead of probing indefinitely.

    True.

    This is because regardless of when the probing starts to cycle among the occupied slots (if they do start to cycle) or whether if the table size is prime or not, if we consider the slot indices generated by each probe as a sequence of numbers, the sequence can only have a period of at most m (i.e after m numbers, the entire sequence will start repeating).

    Note the formula for probe sequence is $(i+k^2)\%m$ where k = 0,1,2,3…

    Once k = m we have $(i+m^2)\%m$ which by the property of modular arithmetic is $(i\%m+m^2\%m)\%m = (i+0)\%m$ so we basically start from k = 0 again, and this keeps repeating, so the probe sequence only has a period of length m. Thus we can simply probe for m times and if no empty slot found, terminate the insertion.

In fact, you need to use at most m/2 probes. The proof is as follows:

Expansion of $(a-b)^2 = (a^2 – 2ab – b^2)$
If we consider the last half of the m terms and re-write them as $(m-x)^2$ for x from 0 to m/2 (give or take 1) this expands to $m^2-2xm+x^2$. m^2 and 2xm % m is clearly 0 so this leaves x^2. Since x ranges from 0 to m/2, this is exactly the same as the 1st half of the terms, just mirrored. So the last m/2 terms will reach the same slots as the 1st m/2 terms.

Grading scheme:

2m: Mentioning having to go through at least m/2 terms in the probing sequence, where m is the size of the hash table.

1m: Mentioning checking if the current slot has already been visited. This only works if the hash table size is prime.

1m: Mentioning checking if a pattern repeats in the probing sequence. This is theoretically possible, but checking for a repeating pattern is not particularly straightforward, and answers in this category do not explain how it would be possible to find a repeating pattern.

0m: Giving any answer that suggests the need to go through the entire probing sequence to determine if any free slots can be found. This is effectively determining if the probing sequence will repeat infinitely by running the probing sequence (which can itself repeat infinitely).

0m: Giving an answer that checks only for whether the initial slot is occupied (it is still possible to find a free slot via probing even if the initial slot has a collision).

12. Quicksort can be made a stable sort by using an extra O(N) space in the partition function (thus making it non in-place) and still maintain O(N) time complexity of the partition function where N is the size of the subarray being partitioned.

    True.

    Simply use 2 extra arrays A and B of size equal to the current size of the subarray to be partitioned. A to store all items < pivot and B to store all items >= pivot while the partitioning is done. Now the relative ordering of the items in A and in B will be maintained. After going through all items in the subarray, copy back to the subarray from A then place the pivot before copying back from B. The total time is still O(N) since we only copy all N items to A and B in O(N) during the partitioning then copy back to the subarray again in O(N) time.

Grading scheme:

2m: Suggesting adding additional information to each element in the array to include its original index as well, thereby ensuring all elements can be considered distinct when partitioning. Note that this must be correctly described, otherwise this goes into the 1m section below.

2m: Modifying partitioning to use two separate arrays/lists for each partition (as in the provided answer).

1m: Only partial handling of swaps in the partition method. As an example:

- Only performing swapping of elements which have the same value as the pivot, after the original partitioning. Swapping also occurs when moving an element into the section < pivot, which is not addressed here.

1m: Suggesting a way to include the original index that doesn't really work. Some examples:

- Storing all element -> index pairs in a HashMap, then using this to determine the original index of an element. Suppose we originally had this array:
  3, 5, 7, 7, 4, 9
  Naturally, you would have a mapping of 7 -> (2, 3) in the HashMap, to indicate that you originally had a 7 in indices 2 and 3. Unfortunately, with only this information, you can't exactly tell which 7 is which (is it the red one or the blue one? From the perspective of the program, they are all 7s)
- Storing the original array in a separate array, then using this to determine the positions of elements with the same value. Again, the program has no way of distinguishing multiple elements with the same value apart in this manner.

0m: Any answer that causes the partition method to run in > O(N) time.

13. The following method **boom** will run in **worst case O(N)** time.

```
public static void boom(ArrayList<ArrayList<Integer>> list2D) {
  // list2D points to a 2D ArrayList in which there are
  // N inner ArrayLists with 1 item each

  int N = list2D.size();
  for (int gap = 2; gap < N; gap *= 2) {
          int home = 0;
          for (int idx = 1; idx < N; idx++) {
                  if (idx == home + gap) {
                          home += gap;
                          continue;
                  }

                  // x.addAll(y) adds every item in y
                  // in sequence to the back of x,
                  // one item at a time. Assume time taken to perform
                  // addAll is worst case O(size of y)
                  list2D.get(home).addAll(list2D.get(idx));

                  // clear will remove all items at index idx of list2D in
                  // worst case O(1) time
                  list2D.get(idx).clear();
          }
   }
}
```

False.

The time complexity of boom runs in O(N log N) time
The algorithm is somewhat similar to a less efficient version of kattis joinstrings problem, where we join:
every 2 lists of size 1 to form lists of size 2, leaving 1 out of every 2 inner lists with items
then every 2 lists of size 2 to form lists of size 4, leaving 1 out of every 4 inner lists with items
...
finally every 2 lists of size N/4 to form lists of size N/2, leaving 1 out of every N/2 inner lists with items

Each join (a.addAll(b) followed by b.clear()) runs in O(size of b) time. Each pass of the outer loop does O(N) operations:

addAll() of 1st pass does around (N/2)*1 operations, and goes through N items in the outer dimension

addAll() of 2nd pass does around (N/4)*2 operations, and goes through N items in the outer dimension
...
addAll() of last pass does around 2*(N/4) operations, and goes through N items in the outer dimension

With O(log N) levels, each doing O(N) operations, the time complexity is O(N log N)

Grading Scheme:

2m: Mentioning that the code runs in O(N log N), with sufficient explanation (eg. mentioning the two time complexities of the inner and outer for loops, and that they are independent). Answers not mentioning that the time complexities are not independent used to be given 1m only, but this was deemed too harsh.

1m: Mentioning that the code runs in O(N log N) without further elaboration, or with mistakes in the explanation.

1m: Mentioning a time complexity greater than O(N) (but not O(N log N)).

0m: Mentioning a time complexity smaller than O(N).

Structured Questions

This section has 4 questions and is worth 58 marks.

Write in **pseudo-code**.

Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes.

Partial marks will be awarded for correct answers not meeting the time complexity required.

*Note: If you use hashing in your answer, you can assume that insert, find and delete operation takes **worst case O(1) time**.

14. Given a List containing N integers which is to be reversed (e.g if the List is {1,2,3} then the reversed List is {3,2,1}), determine an appropriate underlying Linked List implementation of the List that takes O(N) space (singly linked list, tailed linked list ... etc) and give an algorithm to reverse the List in worst case O(N) time.

    You can only use an extra O(1) space (some extra variable and not another data structure of size N) to help you reverse the List, and you cannot use recursion. **[15 marks]**

    Many ways of solving this problem (any LL variant can work in O(N) time).

    Easiest Linked List variant to use is the doubly linked list:

```
If (tail == head)
   do nothing
else
   Let cur = tail
   while (cur != null)
      Let temp = cur.next
      cur.next = cur.prev
      cur.prev = temp
      cur = cur.next
   Let temp1 = head
   head = tail
   tail = temp1
```

    Comments:

    General:
    - misreading the question, assuming that there are 2 Lists - a general List and another LinkedList
    - iterating from 0..N when you want to iterate N times (either ...N-1 or <N)
    - writing A[i]... Are you referring to A.get(i) which takes O(i) time?

    If swapping values:
    - swapping pairs twice, resulting in original sequence
    - not swapping pairs of elements properly

    Playing with references / LL operations:
    - Moving 1 or N-1 elements to the back
    - Treating 2 LLs like stacks (with either but same end as top) and just moving from one to the other and back
    - Stopping just as you reach first/last node and not handling the first and/or last appropriately
    - Algorithm doesn't work for small number of nodes e.g. 0 1 2 3 nodes

Grading Scheme:

Correct solution

1. O(1) extra space and worst case O(N) time ← 15 marks
2. O(1) extra space and worst case O(N^2) time ← 9 marks
3. Using extra O(N) space ← 3 marks
4. Deferred reversal ← 5 marks

Incorrect solutions (using extra O(1) space and worst case O(N) time)

1. swaps attributes of head and tail only ← 2 marks
2. swaps attributes, iterates thru all/almost all nodes and performs meaningful ptr manipulation but doesn't correctly swap ← 5 marks

Deductions

1. for each of 0-3 nodes cases not being handled properly ← -1 mark
2. # of iterations slightly off on each end ← -2 marks
3. for each other minor mistake (losing many nodes is not a minor mistake) ← -1 mark
4. correctly swaps one direction but neglects the other ← -3 marks
5. neglects swapping the head and tail reference ← -3 marks

15. You were supposed to be given a 2D array A of size N*N containing integer values between 0 and 1,000,000 such that the integers are sorted in ascending order from left to right, top to bottom, so that the smallest is at A[0][0] (top left) and the largest at A[N-1][N-1] (bottom right) and if you concatenate all the rows from 0 to N-1 you will get a sorted 1D array in ascending order. An example is shown below for a 3*3 array.

3	10	20
27	33	145
200	303	1023

However, due to some glitch in the computer, the A instead became A' which has the following property:

1. The min value in column j >= The max value in column j' for all j and j' where 0 <= j' < j <= N-1

An example of A' is as shown below for another 3*3 array:

20	44	100
0	25	170
10	30	300

Now give an algorithm to find the integer that should be at A[b][c] (i.e the values in the correctly sorted 2D array) for some valid b and c in **worst case O(N)** time. **[12 marks]**

Key observation:
if you transpose A' all the values in each row are the correct values that should be in the row for A, but they might not be in the correct ordering.

Given any b and c to find A[b][c]:

1. Find column b in A' and sort all the values in that column using Radix sort in O(N) time
2. Return the value at A'[c][b].

Comments
- Mis-interpreting N (there are N^2 elements in the grid, not N)
- Mis-interpreting that A[b][c] is an input, finding the indexes b and c instead
- Thinking that "O(N^2)" sorts on N^2 elements takes O(N^2) time
- Wrongly assuming that rows in A' are partitions (should be columns)
- Wrongly assuming that elements in each column in A' are already sorted
- Manually transposing the entire grid (instead of just using the transposed index)
- Sorting every column (recognizes the property) despite only accessing a value in one column

- Wrongly taking the average case of (randomized pivot) QuickSelect and QuickSort over N elements to be O(N) and O(N log N) worst case respsectively, when the worst case is really O(N^2)
- HashMap isn't necessary since arrays already provide random access (using array as DAT), and all your indexes are sequential from 0..(N-1)

other remarks:
- if you are using column-major ordering (i.e. arr[col][row] instead of arr[row][col]) please explicitly state so
- this question uses 0-based indexes. stick to the convention given in the question

Grading Scheme:

Correct solution

1. O(N) time ← 12 marks
2. O(NlogN) time ←10marks
3. $O(N^2)$ time ←6 marks
4. $O(N^2 logN)$ time ← 5 marks

Deductions

1. flipped row & col indexes ← -3 marks

16. Implement a new ADT with the following operations:

1. **insert(i)**: Inserts an integer i ($0 \leq i \leq 1{,}000{,}000$) into the ADT

2. **removeMaxFreq()**: remove the integer that appears the most frequently from the ADT. If there are multiple integers that appear most frequently, removing any one of them will do. <u>All copies of the integer will have to be removed</u>.

insert(i) should run in worst case not more than **O(n)** time, where n is the number of items in the ADT.
removeMaxFreq() should run in worst case **O(1)** time.

Now use no more than 2 queues and no more than **O(n)** space to implement the above 2 operations. You can only make sure of the **enqueue**/**offer**, **dequeue**/**poll**, **peek** and **isEmpty()** operations of the queue ADT.

If you use any other data structure(s)/ADT(s) in replacement or in addition to the 2 queues that uses more than O(1) space, you will only get <u>half of the marks</u> for a correct solution. **[15 marks]**

Let q1 and q2 be 2 queues. q1 will store the integers sorted by decreasing frequency, while q2 is a helper queue. Let each queue store a pair <x,y> where x is the integer inserted into the ADT and y is its frequency.

```
Insert(i): // O(n) time
        p = <i,1>
        While (!q1.isEmpty()) // find if there is already an entry for i in O(n) time
           p1 = q1.deqeue()
           If (p1.x == i)
              p.y = p1.y+1
           else
              q2.enqueue(p1)
        flag = true
        While (!q2.isEmpty())  // find right position to insert p in O(n) time
           p1 = q2.dequeue()
           if (p.y > p1.y && flag)
              q1.enqueue(p)
              flag = false
           q1.enqueue(p1)
        if (flag)  // insert at back
           q1.enqueue(p)

removeMaxFreq(): // O(1) time
        return q1.dequeue()
```

Comments:

1. Due to there being a restriction on the range of values the integer inserted into the ADT can take (should have been removed ….), all solutions that uses DSes that takes O(1000000) = O(1) space is considered to be correct.

2. A lot of students only keep track of the integer A with the max frequency in the insert operation and don't immediately update it when A is removed, or they do not order the integers based on decreasing frequency, thus when multiple calls to removeMaxFreq are consecutively made, only the 1st call is correct and the rest are wrong.

3. Trying to radix sort the queues. Cannot radix sort as is, since the underlying implementation is not necessarily an array and also the queues can only be manipulated by the queue operations

4. Assuming linked list implementation of queue and just manipulating the linked list directly. This means you are simply using a linked list and not a queue.

Grading Scheme:

Insert operation (5 marks):

Graded according to

2. time complexity
3. Correctness: whether each insert(i) ensures that all integers that are previously in the ADT are still there and i is also inserted correctly).

Time complexity

<= O(N) ← 5 marks
O(NlogN) ← 3 marks
>= O(NlogN) ← 2 marks

Correctness

Major problems

1. if i is enqueued not once but multiple times or 0 times into the queue ← -3 marks
2. if ADT is messed up after each insertion (the integers in the ADT are changed) ← -4 marks

Other problems
-1 mark for each minor mistake.
-2 marks for each normal mistake.
-2 marks for each unclear/vague description of the algorithm.

deduction not applied when marks is <= 1 (if a complete solution given).

removeMaxFreq operation (10 marks):

Graded according to

1. Time complexity
2. Correctness: whether each call to removeMaxFreq will remove all copies of the number with the current max frequency from the ADT.

Time complexity

O(1) time ← 10 marks
O(N) time ← 7 marks
> O(N) time ← 3 marks

Correctness

Major problems

1. consecutive calls to removeMaxFreq will result in wrong integer being removed and only the 1st call is correct ← (-6,-4,-1) marks
2. calls to removeMaxFreq giving wrong answer because the integers inserted are not in the correct order due to errors in the algorithm for insert or removeMaxFreq, but the algorithm to get the integers in the correct order is still discernable. ← (-6,-4,-1) marks
3. No attempt to get the integers in the correct order or the algorithm to get the integers in the correct order is totally wrong (no way to discern what intended algorithm is due to too many mistakes) ← (-9,-6,-2) marks

Where (x,y,z) above correspond to deduction from the O(1) solution, O(N) solution and > O(N) solution respectively.

Other problems

-1 mark for each minor mistake.
-2 marks for each normal mistake.
-2 marks for each unclear/vague description of the algorithm.

deduction not applied when marks is <= 1 (if a complete solution given).

Using other DSes/ADT other than Queue that require > O(1) space: total marks calculated as above divided by 2

17. Mary has invented a new ADT which contains a sequence of integer numbers. The ADT has the following operations:

    1. **insert()**: Insert n to the back of the sequence for the nth call of insert. This means that for the first call to insert, 1 will be added to the back of the sequence, for the 2nd call, 2 will be inserted so on and so forth.

    2. **swap(k)**: Will take the first k numbers in the current sequence and place them to the back of the sequence in the same order. You may assume that k is always valid ($0 < k <= N$) where N is current size of the sequence. For example in the given sequence

    {1,2,3,4,5,6,7,8}

    if swap(3) is called, then we will have

    {4,5,6,7,8,1,2,3}

    3. **maxToRight(i)**: return the largest number that is in an increasing <u>continuous</u> subsequence to the right of the number i, including i. You may assume that i is always a valid number in the sequence. For example if we have the following sequence

    {8,1,2,3,4,5,6,7}

    and maxToRight(8) is called then 8 itself should be returned since after 8 is 1 so there is a decrease instead of an increase. For the following sequence

    {5,6,7,8,1,2,3,4,9}

    if maxToRight(6) is called then 8 should be returned since 8 is the largest in an increasing continuous subsequence from 6, and if maxToRight(2) is called then 9 should be returned.

    Now implement the above 3 operations such that **insert()** and **maxToRight(i)** will run in **worst case O(1)** time and **swap(k)** will run in **worst case O(k)** time.

    [16 marks]

Use a queue Q and hashtable H for this problem.
Q will contain the current sequence of numbers.
The <key,value> pair for H is such that key is each number inserted in the ADT and value is a reference to a node with an attribute val that stores the largest number that is in an increasing subsequence from key.

Let b be the number at the back Q currently. b = 0 at the start
Let count be the number that should be inserted when insert is next called. count = 1 at the start

Insert(): // O(1) time

```
If (b != 0)
   node = H.get(b)  // get the node associated with the key b in O(1) time
else
   create a new node
node.val = count // update val to count since count is currently largest in an increasing sequence
H.put(count,node) // point reference to node in O(1) time
b = count
count++
```

swap(k): // O(k) time

```
Repeat the following k times
        num = Q.poll() // O(1) time
        if num > b
           node = H.get(b) // O(1) time
           node.val = num
           H.put(num,node) // O(1) time
        else // start of a new increasing sequence
           create a new node where node.val is initialized to num
           H.put(num,node) // O(1) time
        b = num
        Q.offer(num) // O(1) time
```

maxToRight(i): // O(1) time

```
return H.get(i).val
```

Grading scheme:

The maximum marks awarded are as follows (before further deductions due to mistakes):

3m: 1 method only

7m: 2 methods only

11m: 3 methods, but at least one method is slower than the expected answer

16m: 3 methods, working correctly and meets the expected runtime (in theory only; no students were awarded full marks for this question)

Some common mistakes:

insert():

This method should not take in a parameter (you are expected to keep track of the number of calls/the value to be inserted instead of it being given to you)

Adding the value = (last element in ADT + 1). The last element may not be the largest element in the ADT due to swap(k) calls.

swap(k):

Assuming that the swapped elements are always in ascending order. This need not be the case.

(For circular array implementations): Simply shifting the front and back indices, without actually shifting any elements in the array. Such an implementation assumes that the circular array is completely full for it to work, which will likely result in conflicts with insert(), where insert() is now expected to add an element to the back of the circular array (which may not be the last element in the usual 0-based array), requiring shifting of elements to make room for the new element, and hence is not in O(1) time.

maxToRight(i):

Assuming i is the index of the element and not the value of the element. For most implementations, you will need to derive the index of the element given i (either via iterating through the ADT or via other means (eg. HashMap)).

Iterating over the array, and stopping when you find an element < i (instead of < curr element). Suppose we call maxToRight(5) on the following (partial) ADT. The expected answer is 9 and not 11:

…5, 7, 8, 9, 6, 10, 11, 4, …