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

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

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

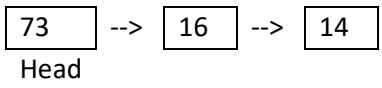
4. You are given the following linked lists which contain **n** elements (where **n** is at least 4). You want to remove both the head and the tail of the linked list. Which of these linked lists **cannot** perform both operations correctly in O(1) time?

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

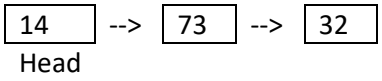
All lists are assumed to use the lecture implementation/descriptions.

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

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;
```

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

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)

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. $O((\log n)^2)$

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

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

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

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.

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.

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();
          }
   }
}
```

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. 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.

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. All copies of the integer will have to be removed.

    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 half of the marks for a correct solution.

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.