CS2040 2022/2023 Sem 4 Midterm

MCQ: 40 Marks, 10 Questions

1. You are given an unknown data structure X, which can be either a stack or a queue. This data structure only supports integers from 0 to 9 (both inclusive), and currently has 2 unknown values stored within. Additionally, this data structure has the following methods:

   insert(e): Runs push(e) if X is a stack, or offer(e) if X is a queue. This method has no return value. The parameter e must be an integer from 0 to 9.
   remove(): Runs and returns the result of pop() if X is a stack, or poll() if X is a queue.
   check(): Runs and returns the result of peek() regardless of whether X is a stack or a queue.

   A call to any of these methods count as one operation. Determine the minimum number of operations needed to determine in all possible cases whether X is a stack or a queue.

a. 2
b. 3
c. 4
d. 5

2. Refer to the method below to answer this and Q3:

```
int gcd(int a, int b) {
    int ans = 1;
    int mult = 1;

    while (mult < a) {
        mult++;
        if (a % mult == 0 && b % mult == 0) {
            ans = ans * mult * gcd(a/mult, b/mult);
            break;
        }
    }
    return ans;
}
```

The method gcd(a, b) is called with 2 parameters a and b, which are distinct positive integers with values of >= n and <= 2n for some variable n. Determine the best case time complexity of gcd(a, b) in terms of n.

a. O(log n)
b. $O(n^{0.5})$
c. O(n)
d. O(n log n)

3. Determine the worst case time complexity of gcd(a, b) in terms of n.

a. O(log n)
b. $O(n^{0.5})$
c. O(n)
d. O(n log n)

4. A sorting algorithm (using lecture implementation) is used to sort the first array given below. The resulting array is shown as the second array.

Initial:

4	7	3	2	4	1	6	5

Final:

1	2	3	4	4	5	6	7

At some point while the sorting algorithm was running, the array looked like this:

2	3	4	7	4	1	6	5

The above only shows the array itself, and does not show any temporary variables or data structures that were used in the sorting algorithm. Determine the sorting algorithm that could have been used.

a. Bubble Sort
b. Selection Sort
c. Quick Sort
d. Merge Sort

5. You are given an empty hash table of size 11, with h(key) = key % 11 as the hash function, and quadratic probing as the collision resolution technique.

   The following keys are to be inserted one at a time (though not necessarily in that order) into the hash table:

   3, 32, 41, 53, 66, 75, 87

   Determine the minimum number of collisions that can occur when inserting these keys.

a. 2
b. 3
c. 4
d. 5 or more (this includes infinity (ie. no slot can be found for a key even in the best case))

6. You are given a sorted array A of size 2n (where n is a positive integer). Additionally, this array has a special property:
   Every pair of elements A[i] and A[2n-i-1] (where i is an integer between 0 to 2n-1 inclusive) sum up to the same value p.
   Unfortunately, the array was corrupted, and now one element has been randomly replaced with a different value, resulting in one pair of elements not summing up to p.

   You are asked to determine the indices of the pair that does not sum up to p. You are provided the corrupted array A, the size (2n) and the value of p.
   The fastest algorithm to do this runs in worst case:

a. O(1) time
b. O(log n) time
c. O(n) time
d. O(n log n) time or worse (this includes the case where the problem is impossible to solve)

7. You are given a linked list with the following properties:
   1. When iterating over the list, beginning from the head, the values encountered are 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3… (this patten repeats for a very long time).
   2. The linked list could either be a circular linked list (with tail reference) or a tailed linked list.
   3. The size of the list is finite.

   Which of the following code would **always** return true if the linked list is a circular linked list, and **never** return true if the list is **not** circular? Select all that apply. It is guaranteed that at least one option is correct, and so this question should not be left blank.

a.
```
return num_nodes == 3; // num_nodes is assumed to contain the
                       // correct number of nodes in the list
```

b.
```
curr = head;
while (curr != null && curr != head) {
    curr = curr.next;
}
return curr == head;
```

c.
```
return tail.next == head;
```

d.
```
return tail.value == 3
```

8. You are given the following hash table:

Index	0	1	2	3	4	5	6	7	8	9	10
Key		26			23				79		

It is known that:

1. The hash table uses h(key) = key % 11 as the hash function.
2. Collision resolution is done by double hashing, with h2(key) = (key % 7) + 1 as the second hash function.
3. The hash table was initially empty upon creation.
4. Various keys were then inserted one at a time into the hash table, and deleted one at a time from the table. For deleted keys, their old positions appear as empty in the table above (functionally, they are marked as deleted, but you cannot deduce their positions from the table above).

Note that for step 4, there is no requirement that all keys are inserted before all deletions occur (ie. insertions and deletions can be mixed in with each other).

Determine the minimum number of deletions that is required to produce this table.

a. 4
b. 5
c. 6
d. 7 or more (this includes the case where the table is impossible to produce regardless of the number of deletions)

9. You are given the following unsorted array, where each element consists of an integer value and a string:

4 "A"	3 "D"	4 "C"	2 "B"	1 "A"	8 "C"	3 "B"	5 "G"

You are allowed to pick any sorting algorithm (using lecture implementation) to sort this array. All sorting algorithms will use only the integer value as the sort key, but will move the entire element (ie. both the integer and the string) as a whole.

Determine the number of different resulting arrays that can be achieved (here, two arrays B and C are different if for any valid index i, B[i].value != C[i].value
or !B[i].string.equals(C[i].string)).

a. 1
b. 2
c. 3
d. 4

10. You are given a basic linked list with 3 elements, and the following code fragment:

```
head.next.next = head.next.next.next
head.next = head.next.next
head = head.next
```

What happens to the linked list at the end of the code above? You may assume there is no need to update the num_nodes value.

a. The first two elements are removed.
b. The middle element is removed.
c. The last two elements are removed.
d. All elements are removed.

Analysis: 18 marks, 3 questions (6 marks each)

11. **[6 marks]** Jasper the tree frog is excellent at leaping. In one leap, he can achieve a distance of **x** centimeters. However Jasper cannot maintain this distance if he leaps continuously, and he will eventually run out of stamina and thus can only leap consecutively for **n-1** times (**n > 2**). Assuming the first leap is the 0th leap where Jasper can cover a distance of **x** centimeters, the distance covered by Jasper will be $\left(1 - (\frac{1}{n-i})\right) * x$ for the ith leap after the 0th leap.

Based on the above we can conclude that a tight upperbound on the distance covered by Jasper will be **O(nlogn)*x**.

a. This is false. Tight upper bound should be O(logn)*x since the distance of each leap is reduced by a constant fraction of the distance of the previous leap.

b. This is false. Upper bound should be O(n)*x. The reason is as follows:
If Jasper does not have any reduction in his leaping distance then he will leap a total of (n-1)*x distance, which is upper bounded by O(n)*x.
However each time he leaps his distance is reduced by some amount and the sum of this total reduction in leaping distance is upperbounded by O(logn)*x.
So total distance covered is (n-1)*x-(logn)*x which is still tightly upper bounded by O(n)*x.

c. This is false. Upper bound should be O(1)*x. The reason is as follows:
If Jasper does not have any reduction in his leaping distance then he will leap a total of (n-1)*x distance.
However each time he leaps his distance is reduced by some amount and the sum of this total reduction in leaping distance is (n-2)*x.
So total distance covered is (n-1)*x-(n-2)*x which is tightly upper bounded by O(1)*x.

d. This is true. Since Jasper can leap n-1 times and each leap will average out to be a distance of O(logn)*x, so total distance is tightly upper bounded by O(nlogn)*x

Question 12 to 13 refers to the following problem

12. **[3 marks]** For a hash table using separate chaining as a collision resolution technique, if a sorted arraylist instead of a linked list was used, then **insertion**, **deletion** and **retrieval** can be done in **worst case O(logn)** time instead of **worst case O(n)** time.

    The statement is true or false?

13. **[3 marks]** Give your rationale for your answer to the previous question.

Question 14 to 15 refers to the following problem

14. **[3 marks]** Given the following function:

```
somefunction(int[] arr):

        sum = 0

        for (int i = 0; i < arr.length; i++)

            sum += arr[i]

        return sum
```

    John concludes that the time complexity of the function is O(n) where n is the length of arr. So for all arrays which are of length 0, the function will take O(0) = 0 time.

    John is correct.

15. **[3 marks]** Give your rationale for your answer to the previous question.

Structured Questions: 4 questions

This section is worth 42 marks. Answer all questions.

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.

16. **[6 marks]** John wants to implement an ADT with the following operations:

    1. **void insertFirst(int val)** - insert a positive integer value val as the first item in the ADT
    2. **void insertLast(int val)** - insert a positive integer value val as the last item in the ADT
    3. **void deleteFirst()** - delete the first item in the ADT. If the ADT is empty nothing is done.
    4. **void deleteLast()** - delete the last item in the ADT. If the ADT is empty nothing is Done
    5. **int getVal(int i)** - return the item at index i (using 0-based indexing, i.e first item is at index 0). if i is not valid, i.e i < 0 or >= n where n is the size of the ADT, return -1

    such that each of the above operations can run in average O(1) time or better (note that worst case O(logn) is not better then average case O(1)).

    The following DSes can be used to implement the John's ADT so that they run in the required time complexity (choose all correct options that apply):

    a. A Queue + some extra variables
    b. A Stack + some extra variables
    c. A Linked List + some extra variables
    d. A Hashtable + some extra variables
    e. A Array List + some extra variables
    f. None of the above options can implement the required operations for John's ADT so that they run in the required time complexity.

17. **[6 marks]** There is a queue in a town for a local attraction. As a way to “spice up” the queueing experience, the organisers have decided to introduce a special rule: at times, their mascot may join the queue, and during this time, the next person who gets to enter the attraction will be the person immediately in front of the mascot, instead of the person at the front of the queue. During this time, no new people are allowed to join the queue.

    You are to write a program to simulate this queue. Your program should support the following methods:

    1. queue(String x): the person with name x joins the back of the queue if the mascot is not in the queue, otherwise nothing is done.
    2. dequeue(): if the mascot is not in the queue, the person at the head of the queue leaves the queue. Otherwise, the person immediately in front of the mascot leaves the queue. In either case, this method is expected to output the name of the person that leaves the queue and remove the person from the queue. It is guaranteed that a valid person to leave the queue exists.
    3. mascotJoin(): the mascot joins the back of the queue. It is guaranteed that the mascot is not in the queue when this is called.
    4. mascotLeave(): the mascot leaves the queue. It is guaranteed that the mascot is in the queue when this is called.

    You may assume that everyone in the town has a unique name, and once they have left the queue, they will not rejoin the queue again.

    As an example, suppose the current people in the queue are as follows:

    Bob, Sam, Amy, Pat

    Where the leftmost person is at the front of the queue. If dequeue is called, the queue now looks like this:

    Sam, Amy, Pat

    Where Bob is the person that has left the queue. Suppose the mascot has joined the queue at this time, thereby forming:

    Sam, Amy, Pat, Mascot

    After another dequeue, the queue would be:

    Sam, Amy, Mascot

    As Pat is the person immediately in front of the mascot, they are the next person to leave the queue. Finally, the mascot leaves, causing the queue to be:

    Sam, Amy

    After another dequeue, Sam leaves, leaving Amy as the only person still in the queue.

    Determine the possible data structures that can support all operations in worst case O(1) time. Select all that apply:

a. A doubly linked list + additional variables
b. A tailed linked list + additional variables
c. A basic linked list + additional variables
d. A stack and a queue + additional variables
e. A hashtable + additional variables

18. **[14 marks]** You are given a doubly linked list and node class with the following attributes:

    doubly linked list class attributes:
    head - reference to the first node of the doubly linked list
    tail - reference to the last node of the doubly linked list
    num_nodes – the number of nodes in the doubly linked list

    node class attributes:
    val - item (an integer value) contained in the node
    next - reference to the next node
    prev - reference to the previous node

    The doubly linked list class has the following implemented operations (which you can use as is):

    1. insert(index, item)
    2. insertLast(item)
    3. insertFirst(item)
    4. delete(index)
    5. deleteFirst()
    6. deleteLast()
    7. getNodeAtIndex(index)
    8. getFirstNode()
    9. getLastNode()
    10. size()

    Now implement a new operation for the doubly linked list class described as follows:

    void translocation() -
    Split the doubly linked list into half, the last index of the 1st half = (length of list)/2 where / is integer division.
    Switch both halves so that the first half becomes the 2nd half and the 2nd half becomes the 1st half of the list.
    If there is <= 1 items in the list then no change should be made to the list.

    E.g given a doubly linked list containing the integers as follows from left to right

    1, 2, 3, 4, 5

    after applying the translocation operation it will be as follows:

    4, 5, 1, 2, 3

    <u>You can directly access the attributes in the doubly linked list class and node class in order to manipulate them and implement translocation()</u>

An example is given below:

```
int example_operation() // implementation of an example operation in the doubly linked list class
  first_item = head.val
  second_item = head.next.val

  return first_item+second_item
```

19. **[16 marks]** A sequence of positive numbers representing signals from space are being recorded by the government agency DDEA (Department to discover extraterrestrial activity) at regular intervals.
In order to properly and efficiently record and make sense of the signals, DDEA requires you to implement a ADT that consists of the following operations:

1. **insert(int x)**: insert a signal x into the ADT as the latest signal being recorded.

2. **insertAndTrack(int x)**: insert a signal x into the ADT as the latest signal being recorded and also track the largest signal from this point onwards.This is called a **tracking**.

For example, if the following signals are inserted with the bolded and asterisked one being inserted and tracked

1,2431,14,41,**33***,8178,221,13,11

then at the point when 11 is inserted, the largest signal being tracked will be 8178.

Another example:

231,**1411***,13,131,111

at the point when 111 is inserted, the largest signal being tracked will be 1411.

Note that multiple trackings can be done. An example is as follows
321, 32131, **132***, 3113, 773111, **13119***, 31781, 1313

when 1313 is inserted, the tracking that starts at 132 will have 773111 as the largest signal being tracked, while the tracking that starts at 13119 will have 31781 as the largest signal being tracked.

3. **LargestSignalOfLatestTracking()**: return the largest signal of the latest tracking.

For example:

1,2431,14,41,**33***,8178,221,13,11 → there is only 1 tracking with largest signal tracked being 8178, so 8178 is returned.

Another example:

321, 32131, **132***, 3113, 773111, **13119***, 31781, 1313 → there are 2 tracking with one having largest signal 773111 and the other (the latest tracking) having largest signal 31781. Thus 31781 is returned.

4. **delete()**: This will delete the latest recorded signal. If there is a tracking associated with the signal (i.e the signal was inserted using **insertAndTrack()**) remove the tracking too.

Determine the most appropriate DS(es) to use and implement each of the above operations so that they will run in **worst case O(1) time**.