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

Ans:

This can be done by the following:

1. Run check() and store the result.
2. Insert(e), where e is an integer different from the result of step 1.
3. Run check() again and store the result.

If the numbers in step 1 and 3 are the same, this is a queue. Otherwise, this is a stack.

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)

Ans:

For Q2, the best case scenario is when a is a power of 2, a == n, and therefore b can be a value == 1.5a (since a is a power of 2, it is even and (a * 1.5) will still give an integer). In this case, the method will quickly encounter factors of both numbers and hence take O(1) time per recursive call. There will be a total of O(log n) recursive calls (since a is a power of 2), giving a total time complexity of O(log n).

For Q3, the worst case scenario is when a and b are prime number (or coprime numbers). In this case, no common factors can be found, and the algorithm will run until mult == a, which takes O(n) time.

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

Ans:

Bubble sort and selection sort, as taught in lectures, always works by moving the largest elements to the right of the array. As 7 is not in the rightmost position of the array, these algorithms could not have been used.

For quick sort, it works by picking the first 4 as the pivot. Once it has finished partitioning with 4 as the pivot, all elements to the left of it should be < 4, and all elements to the right of it should be >= 4. These properties do not hold for either element of value 4 in the array, as 1 lies to the right of both of them.

For merge sort, this is possible if, splitting the initial array into half, we have finished sorting the first half of the array. Owing to how the recursive calls are done in merge sort, the second half of the array need not necessarily be sorted yet, which allows for the array above to exist.

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))

Ans:

First, we start by determining which slot each key should be added to:

Slot 0: 66
Slot 3: 3
Slot 8: 41
Slot 9: 53, 75
Slot 10: 32, 87

Clearly, there must be at least 2 collisions due to there being 2 keys for slots 9 and 10. Next, we look at the probe sequence for slots 9 and 10 to determine the possibility of further collisions:

9 -> **10** -> 2 -> 7 -> **3** -> 1 -> …
10 -> **0** -> **3** -> **8** -> 4 -> 2 -> …

The probe sequence of slot 10 is highly problematic, as it covers many slots early on which will be used by other keys. Therefore, we want to minimize the number of times this probe sequence is used, by ensuring slot 10's first key does not encounter a collision. Therefore, either 32 or 87 should be inserted first. The other key can then be inserted, causing it to be put at slot 0 (+**1** collision).

Doing so causes key 66 to encounter a collision when inserted, putting it into slot 1 (+**1** collision). The remaining keys can then be inserted in any order. Either 53 or 75 will encounter **2** collisions (at slots 9 and 10), requiring itself to be put into slot 2 instead, for a total of **4** collisions.

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)

Ans:

Despite the special property being given, there is still no way to determine the pair other than trying all possible pairs (which takes O(n) time), and determining which pair does not sum up to p.

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
```

Ans:

Option a: if the list is circular, it could involve more than 3 nodes (eg, a circular linked list containing 1, 2, 3, 1, 2, 3). This therefore cannot cover all cases of circular linked lists.

Option b: While this may appear to work at a glance, the condition of the while loop is faulty (checks for curr != head, when curr was initialised to head), and therefore the code always returns true.

Option b: This is the definition of a circular linked list.

Option d: If the list is circular, then the tail must have a value of 3 (or it becomes impossible to have the sequence above). However, if the list was non-circular, it could still have a tail value of 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)

Ans:

Start by determining where each key would be placed if there were no collisions, followed by their probe sequence:

23: 1 -> 4
26: 4 -> **10** -> **5** -> **0** -> **6** -> 1
79: **2** -> **5** -> 8

Notice that there are 5 slots (0, 2, 5 (appears twice), 6, 10) which appear in the probe sequence, but are empty. This implies that these 5 slots must have been deleted at some point.

However, also note the relation between 23 and 26. 23 requires slot 1 to be occupied (currently occupied by key 26), while 26 requires slot 4 to be occupied (currently occupied by key 23). Therefore, one of those slots must have stored a different key before subsequently being deleted later, requiring another deletion, thereby giving us a total of 6 deletions.

A possible sequence of insertions and deletions could therefore be as follows:

0, 1, 2, 5, 6, 10, 79, 23, D0, D1, D2, D5, D6, D10, 26, where (no letter) indicates insertions, and D indicates 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

Ans:

This is a test of the stability of sorting algorithms. Any stable sort would give the same result, hence only one of such sorts needs to be done. The result is:

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

Now, we need to run each of the unstable sorting types. Selection sort gives us:

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

Quick sort gives us:

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

Therefore, the total number of different resulting arrays is 3.

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.

Ans:

Initially, head.next.next.next is null (the element after the last node in the list). By substituting null into head.next.next, we see that head.next is also set to null, and subsequently, in the last line, head is set to null, therefore the linked list is now empty.

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

Ans:
The sum of distances covered is:
1*x + (1-1/(n-1))*x + (1-1/(n-2))*x + ... + (1-1/(n-(n-2)))*x

first term is the distance of the 1st jump and the last term is the distance of (n-1)th jump.
This is equal to

1*x + (1-1/(n-1))*x + (1-1/(n-2))*x + ...(1-1/3)*x + (1-1/2)*x
= x[1 + (1-1/(n-1)) + (1-1/(n-2)) + ...(1-1/3) + (1-1/2)]
= x [n-1 - (1/(n-1) + 1/(n-2) + .. + 1/3 + 1/2)]
= x [n-1 - O(logn)] -> 1/2 + 1/3 + ... + 1/(n-1) is a harmonic series that sum to O(logn)
= x * O(n)

thus the upper bound is x*n = O(n)*x and not x*nlogn = 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?

Ans:

False

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

Ans:

Find can be done in O(logn) time in a sorted arraylist. However once an item is found, deletion will take O(n) time due to the need to close the gap (shift O(n) items in the arraylist to do so in the worst case).

Also insertion will take O(n) time if the key to be inserted is not found since there is a need to introduce a gap for the insertion (again you need to shift O(n) items in the arraylist to do so in the worst case).

Grading scheme:

One mark is given for the correct worst-case time complexity analysis for each operation: retrieval, insertion and deletion.
The analysis of retrieval is important, because the runtime affects deletion as well.

Some notable answers:

1. Mentioned that insertion and deletion takes worst case O(n) time, and/or retrieval takes worst case O(lg n) time, but does not specify how. No marks is given.
2. Suggested the need to expand the array when array is full during insertion, and the answer is accepted.
3. Gave assumptions that contradicts the question, such as assuming that array is not sorted, or assuming hashtable only maps integers. No marks is given.
4. Suggested sorting the array after inserting to the back of the array instead of inserting into the correct position within the array, resulting in worse runtime. No marks is given.
5. Extending from Point 3/4, suggested sorting the array using a comparison-based sort runs in O(lg n) or O(n) time. This is a very critical misconception

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.

Ans:

False.

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

Ans:

Even though the big-O time complexity is O(nlogn) the actual function is nlogn+c where c is some constant since there is still constant time taken to call the function regardless of the value of n and also to return sum. Thus even when the array size is 0, the time taken is O(1) meaning that some constant time or constant number of operations is required to execute the function.

Grading scheme:

As long as there is a mention of constant work being done / some time needed for either function call, variable declaration and initialization, for-loop computation, return statements etc., full credit is given.

Incomplete rationale gets 1 mark.

Rationalizing that the function has a time complexity of O(0) or O(1) will not get credit.

We can only claim that with an empty array, the time taken to run the function is O(1).

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.

Ans:

Option d only.

Use a hashtable ht with 2 extra variable, start and size. The key value pair is <index,val> where val is the item inserted in the the ADT and
index is the "index" of the item in the ADT.

start and size is initalized to 0

```
insertFirst(int val): // ave O(1) time
  if (start != 0)
    start -= 1
  ht.insert(<start,val>)
  size += 1

insertLast(int val): // ave O(1) time
  ht.insert(<start+size,val>)
  size += 1

deleteFirst(): // ave O(1) time
  if size > 0
    ht.remove(start)
    if start != 0
      start += 1
    size -= 1

deleteLast(): // ave O(1) time
  if size > 0
    ht.remove(start+size)
    size -= 1

getVal(int i): // ave O(1) time
  return ht.get(start+i)
```

Queue/Stack/Arraylist can do either insertFirst or insertLast and deleteFirst or deleteLast in O(1) time but the other will need O(n) time. Linked List (whichever variation) cannot do general access in O(1) time same with queue and stack.

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

Ans:

Option a only.

For option a, we can use the head of the list as the front of the queue. We can add people to the tail of the list, and remove people from the head of the list. If the mascot is present, we can remove people from the back of the list in worst case O(1) time.

For option b, this is not possible as removal from the back of a tailed linked list cannot be done in O(1) time, thereby preventing O(1) dequeuing while the mascot is present. The same applies to option c.

For option d, it may initially be possible to add a new person to both the stack and the queue, and remove from the queue if the mascot is not present, or remove from the stack if the mascot is present. In this case, we can define a person as present if they appear in both the queue and the stack, and absent if they appear in only one of the 2 ADTs. However, this will eventually lead to the queue ADT containing many elements which represent people who are no longer in the actual queue, requiring cleaning up which takes more than O(1) worst case time (in a manner similar to Tutorial 3 Q3).

For option e, all operations will run in worst case O(n) time, and hence cannot support any operations in worst case O(1) time.

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

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

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
```

Ans:

```
void translocation()
  if size() <= 1
    return
  mid = size()/2
  cur = getNodeAtIndex(mid)
  tail.next = head
  head.prev = tail
  cur.next.prev = null
  head = cur.next
  cur.next = null
  tail = cur
```

or even simpler

```
void translocation()
  mid = size()/2
  for i from 0 to mid
      node = deleteLast()
      insertFirst(node.val)
```

Grading Scheme:

No marks are deducted for missing the midpoint if it is off by 1. As the original question description does not split the list evenly in half if the size is even (eg. for size 4, the list is split into 3 and 1 elements), students who took the description of “half” to mean this should be split 2-2 will not be penalised.

Answers generally are split into two types: those which manipulate pointers, and those which utilise the add/remove methods provided in the question. Each has their own possible errors:

Pointers:

1 mark is deducted for each incorrect pointer at the end of the operation, assuming your answer is close to correct (this condition is necessary, as a list that is not modified has only 6 incorrect pointers, and is not worth 8 marks).

Generally, the pointers that are missed are:

Old/new head .prev pointers (note that the list explicitly stated to be a doubly linked list)

Head/tail pointers of the list

Add/remove:

2 marks are deducted for issues with looping over the list. Typically, this is caused by eg. running a for loop with variable i from 0 to mid, and performing delete(i) on the list. Note that due to deletion, the indices of elements in the list may have shifted, resulting in deleting the incorrect elements. This also applies to loops which directly compare with list.size() (ie. the method), instead of first storing it as a variable somewhere, as list.size() will return a different result each time as elements in the list are deleted.

For less complete answers, simply finding the midpoint correctly awards at least 4 marks.

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**.

Ans:

Idea:
Use a stack to do tracking the largest value encountered so far. If there is no tracking yet, the largest value encountered will be -1.

Solution:
Let S be a stack that contains the pair <x,largest> where x is the current signal being inserted and largest is the largest signal so far from the 1st tracking. largest will be -1 if there is no tracking yet.

```
insert(x):
    if S.empty() or S.peek().largest == -1
        S.push(<x,-1>)
    else
        S.push(<x,max(S.peek().largest,x)>)

insertAndTrack(x):
    S.push(<x,x>)

LatestLargestSignalTracked():
    if S.empty()
        return -1
    return S.peek().largest

delete():
    if !S.empty()
        S.pop()
```

Grading Scheme:

The maximum amount of marks (before additional deductions) depending on the number of implemented operations are as follows:

1 operation: 3m
2 operations: 7m
3 operations: 11m
4 operations: 16m

Rationale: the solution is immediately much simpler if one operation is ignored.

For each operation that runs worse than O(1) worst case (including the use of hash tables, or unrealistically sized DAT), up to O(n) time, 1 mark is deducted.

To avoid penalising hash tables too severely, deductions due to operations that would take O(1) average time for using a hash table are capped at 2m total.

For each operation that runs in O(n log n) time or worse, 2 marks are deducted.

Other common issues:

Some solutions require integers to be distinct to work correctly. This incurs a 1m penalty.

insertAndTrack():

Note that this signal itself can be the largest signal found so far (2nd example in the question). Missing this case incurs a 1m penalty.

largestSignalOfLatestTracking():

Note that this is specifically the largest signal of the latest tracking, not the largest signal across the entire list of signals. Missing this incurs a 1m penalty.

delete():

Assuming that the signal being deleted is always a signal inserted by insertAndTrack() incur a 3m penalty. Unable to correctly get the next largest signal if the largest signal is deleted incurs this same penalty as well. As an example, suppose we have the following signals already:

4*, 7, **43***, 87, 46, 164

At this point, the value that should be returned by largestSignalOfLatestTracking() should be 164. However, after deletion, this should be 87. Some common incorrect answers would return:

164 (the value is unchanged)
46 (the value immediately before this signal)
7 (the largest signal found before the most recent tracking)