Fall 2024 - Data Structures Course Midterm Test (Sample Answers)

MC 1. Which of the following descriptions is <u>incorrect?</u> ( )

A. The meaning of the algorithm working in-place is that it does not need any additional auxiliary space.
B. Under the same scale, the algorithm with complexity O(n) is always superior to the algorithm with complexity $O(2^n)$ in terms of time.
C. The so-called time complexity refers to an upper bound of the estimated algorithm execution time in the worst case.
D. For the same algorithm, the higher the level of implementation language, the lower the execution efficiency.

MC 2. Evaluate the following postfix expression: *7 8 + 3 2 + /*, what is the final result after evaluating this expression? ( )

A. 1
B. 3
C. 5
D. 7

MC 3. Analyze this code, Which of the following options about the code's execution time and time complexity analysis are <u>incorrect</u>? ( )

```
a = 5, b = 6, c = 10

for i in range(n):
    for j in range(n):
        x = i * i
        y = i * j
        z = i * j

for k in range(n):
    w = a * k + 45
    v = b * k
d = 33
```

i. Adding an inner loop `for m in range(n)` inside double loop, with a constant-time computation inside it, would change the overall time complexity to $O(n^3)$.
ii. If n is a very large value (e.g., $n=10^6$), the constant term 4 will significantly affect the overall execution time of the code.
iii. If an O(log n) operation is added at the end of each double loop iteration, the overall time complexity of the code will change to $O(n^2 \log n)$.

iv. Changing `for k in range(n)` to `for k in range(n²)` will make the overall time complexity $O(n^3)$

A. iii, iv
B. ii, iii, and iv
C. ii, iv
D. iv

MC 4. The suffix expression for *a∗(b+c)/e−d* is ( )

A. abc+∗e/d−
B. abc+∗de/−
C. ab+c∗e/d−
D. abcd+/e∗−

MC 5. Which of the following <u>is NOT</u> a basic operation of a stack? ( )

A. Push
B. Pop
C. Peek
D. Sort

MC 6. In the simulation of the Hot Potato game using a queue, a group of players stands in a circle and passes the hot potato around.

When the count reaches a specified number, the player holding the hot potato is out.
Which statements about using a queue to simulate this process is correct? ( )

A. To implement circular passing, a double-ended queue (Deque) is required, with players dequeued from the front and enqueued at the back.
B. A standard queue can be used, with the player holding the hot potato being moved to the back of the queue by dequeuing and then re-enqueuing.
C. To simulate a circular queue, two pointers must be added to mark the beginning and the end of the queue.
D. The enqueue and dequeue operations of the queue both have a time complexity of O(1), making it the optimal choice for implementing the hot potato game.

MC 7. Which statements about linear data structures and queues is incorrect? ( )

A. A linear data structure organizes data items in a certain linear order.
B. A queue maintains elements in the order of FIFO.
C. In a queue, elements can be inserted and removed at any position.
D. Dual-end Deque can have both the capabilities of stack and queue

MC 8. In a chain queue, assuming that the queue head pointer is front, the queue tail pointer is rear, and the element pointed to by x needs to be enqueued, which of the following operations should be performed? ( )

A. `front = x; front = front -> next`
B. `x -> next = front -> next; front = x`
C. `rear -> next = x; rear = x`
D. `rear -> next = x; x -> next = null; rear = x`

MC 9. In a singly linked list, each node contains two fields: data (which stores data) and next (a reference to the next node). We have a linked list *L* with nodes storing the following values: *L: 17 -> 26 -> 31 -> 54 -> 77 -> 93*, and:
We need to delete the node with the value 31. The deletion process involves two steps:
1. Locate the node with the value 31 and its predecessor node.
2. Make the predecessor node's next point to the successor node of 31.

- After completing the deletion operation, what will the state of the linked list L be?
- And, In what order should pointer operations occur?

A. *L: 17 -> 26 -> 54 -> 77 -> 93*, where the predecessor's next is directly pointed to 31's successor.
B. *L: 17 -> 26 -> 54 -> 77 -> 93*, after deletion, the next pointer of the deleted node 31 is set to None.

C. *L: 17 -> 26 -> 31 -> 54 -> 93*, only the next pointer of 31's predecessor is updated, with no change to subsequent nodes.
D. *L: 17 -> 26 -> 77 -> 93*, where both intermediate nodes 54 and 77 are skipped in the deletion of node 31.

MC 10. Suppose there's a recursive function `countWays(n, coins)` **that calculates the number of ways to make an amount n using coin denominations [1, 5, 10, 25]:**

- So, when calculating, e.g., `countWays(63, [1, 5, 10, 25])`,
- which option correctly describes the nature of the recursive calls? ( )

A. Each recursive call removes one coin denomination and recursively computes combinations with the remaining denominations.
B. Each level of recursion uses only one coin denomination to calculate possibilities.
C. The recursion uses coin denomination combinations without decreasing the target `n`.
D. All of the above are incorrect.

MC 11. Which is Not one of the three essential components of recursion? ( )

A. Defining the base case
B. Dividing the problem into smaller subproblems
C. Implementing a loop structure
D. Making a recursive call to itself

MC 12. The total number of moves required for the Tower of Hanoi with 5 disks is ( )

A. 7
B. 15
C. 31
D. 127

MC 13. During the execution of recursive algorithms, the data structure that the computer system must use is ( ).

A. Binary Tree
B. Queue
C. Linked List
D. Stack

MC 14. Use *S* to represent the stacking operation and *X* to represent the stacking operation. If the stacking order of elements is 12345, to obtain 13425 stacking order, what is the corresponding operation sequence of *S* and *X*? ( )

A. SXSXSSXXSX
B. SXSSXSXXSX
C. SSSXXSXSXX
D. SXSSXXSXSX

MC 15. Given a recursive function `sumNestedList(nestedList)` calculates sum of all integers in a nested list, where the list can contain integers or other nested lists.

- e.g., list [1, [2, [3, 4]], 5], the sum is 1 + 2 + 3 + 4 + 5 = 15.
- **If the input list is [1, [2, [3, [4, 5]]]], what is the depth of recursive calls made by `sumNestedList`?** ( )

A. 3
B. 4
C. 5
D. 14

Part B - Long Questions

(Total: 40 Marks)

The following are the very good or even perfect answers.

Q.1

LQ 1. In the stack implementations shown, the version on the left has the *push* and *pop* operations with a time complexity of O(n), while the version on the right has a time complexity of O(1) for these operations. **(12 Marks)**

```
class Stack_left:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.insert(0, item)
    def pop(self):
        return self.items.pop(0)
    ...
```

```
class Stack_right:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    ...
```

a) Explain why the *push* and *pop* operations in the left version have **O(n)** complexity, whereas they have **O(1)** complexity in the right version.

In general, since we use the different end of the list, so we have different operation when doing "push" and "pop".

① For the left one, when "push", we do that in the head and the element backward statement executes n times with O(n) time. When "pop", it requires moving all but the head element with an O(n) time complexity.

② For the right one, when "push", we just insert the new element at the end, the element backward statement will not execute, and the time complexity is O(1).
When "pop", we just delete the tail element, without moving the element, with a time complexity O(1).

b) Which implementation is more efficient for a stack structure, and why?

right one because if have a clearly structure and pop() coresponding to a normal stack it follow FIFO

Ok, OK it's frequently oper

Q.2

LQ 2. Read this code and answer the questions. **(12 Marks)** Is less.

```
def what_function_of_this (n, b):
    S = "0123456789ABCDEF"
    if n < b:
        return S [n]
    else:
        return what_function_of_this(n // b, b) + S[n % b]
print(what_function_of_this(1453, 16))
```

a) Which classic problem we have studied does this code, or `def`, solve?

Convert integers to an arbitrary decimal system, such as binary and hexidecimal.

b) For this problem, in addition to the solution in the code, we have also learned another solution. Please briefly describe the core idea of the other solution.

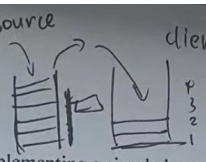
Tips: two key points about the method, structure and its characters

Structure: We firstly create a string holding from 0 to F, and a stack to hold the numbers.
Then, we design a "while" loop. Every loop, we module the number by base and push the remainder to stack.
Then we devide the number by base and just have the integer part. When number is less or equal to 0, stop loop.
Then, pop the element in stack one by one and connect to a string.

Characters: The process of "getting the remainder" is obtained in order from low to high but the output is from high to low. Stack is "LIFO". We use a stack to store the numbe to reverse it.

Q.3

Fig1: resource client



LQ 3. You are implementing a simple browser history feature that supports visiting new pages, going back to the previous page, and moving forward to the next page. **(16 Marks)**

a) Which basic data structure is suited for this navigation functionality? Explain why.

Tips: Queues or Stacks

(a) Stack is suited for this navigation functionality. By the FIFO rules, we can easily going back to the previous page. The characteristic of stack is well fit for save the history of viewing and get the new page.

b) Describe how to use this data structure to implement the visit, back, and forward operations.

Tips: by basic operations

(b) we can use two stack to operate. One stack is represent the resoure we are going to view and the other ~~was~~ is our client to view the resource. Like the Fig1 on the top, we get the resource by pop() and push() it into our client stack. That's viewing new pages. And pop() in our client and push() it into the resource, then we can see our previous page without ruining the order. Then keeping push() and pop() into client, or changing the resource stack, we can perform

LQ 3. You are implementing a simple browser history feature that supports visiting new pages, going back to the previous page, and moving forward to the next page. **(16 Marks)**

a) Which basic data structure is suited for this navigation functionality? Explain why.

Tips: Queues or Stacks

Stacks. The reason is as follows. Stack's a "LIFO" ds, which can restore the pages.

① When ~~the~~ visiting a new page, this page (or URL) is pushed into a browser stack, being top of the stack. Old pages has been restored in the stack

② If we want to go back, then we need to pop out the top pages and get previous pages. (maybe one or many pages)

③ The only thing we need to do to move forward to next page's to push them into stack, get the peek of stack and then read it.

b) Describe how to use this data structure to implement the visit, back, and forward operations.

Tips: by basic operations

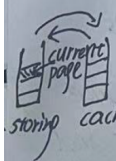
Use 2 stacks, one for storing the history, one for cache (缓存)

① visit: if it's a new page, then push it into storing stack. if it's old page, try to search this page in 2 stacks.

~~② back: pop out the element until we get what we want.~~

② back: In storing stack, pop out the ~~stack~~ elem and push them into cache

~~③ forward:~~ until get target page. In cache stack, pop out the elem and push them into storing stack until get target page. This ensures the order of all pages we visited.



② back: pop out the element in storing stack and push it into cache.

③ moving forward: pop out the element in cache stack and push it into

c) If you need to display the entire list of visited pages in order (start to end or end to start), how would you design or modify the data structure to support this feature?

I'll modify the structure to a dequeue.
Since when we want to display all the visited pages, we need to display it in time order, which is the character of a queue. And dequeue has the features of both stack and queu
When we do the basic browser operations, we do it just at one end, such as Front.
"Visiting new page": d= Deque() create a dequeue
"Going back": d.removeFront() delete the current one.
"Moving forward": d.addFront(next page). add a next page.
When we want to display history pages, we do it at the other end one by one. We do the "d.removeRear()" for n times where n is the number of pages.