CS2040S: Data Structures and Algorithms

Tutorial Problems for Week 5: Lists, Stacks and Queues

For: 5 Sept 2022, Tutorial 3

Solution: Secret! Shhhh... This is the solutions sheet.

Problem 1. True or False?

For each of the following, determine if the statement is True or False (time complexity given is worst case time complexity), justifying your answer with appropriate explanation. The Linked List and variations mentioned are as given in the lecture notes.

a) Deletion in any Linked List can always be done in $O(1)$ time.

b) A search operation in a Doubly Linked List will only take $O(\log n)$ time.

c) All operations in a stack are $O(1)$ time when implemented using an array.

d) A stack can be implemented with a Singly Linked List with no tail reference with $O(1)$ time for all operations.

e) All operations in a queue are $O(1)$ time when implemented using a Doubly Linked List with no modification.

f) Three items A, B, C are inserted (in this order) into an unknown data structure X. If the first element removed from X is B, X can be a queue.

Solution: Note that the True/False questions may not have a unique solution. As long as you have a valid justification for the solution, it will also be correct. This will also apply to all future tutorial questions.

a) False. Deletion only $O(1)$ at head or tail (only for doubly linked list). $O(n)$ otherwise.

b) False. Search is always $O(n)$, since it is unsorted. Even if it is sorted, you still cannot achieve $O(\log n)$ time using binary sort since you cannot directly access a node at a particular index in $O(1)$ time unlike the case of an array.

c) False. On average, insertion is $O(1)$ time, but in the worst case individual insertions can be $O(n)$ time due to resizing of the array. Note that if we consider amortisation, we can prove that a stack implemented with an array has worst case amortised $O(1)$ time complexity for insertion. use amortized time for array implementation of stack/list/q

d) True. Insertion and deletion only required to be done at the head of the linked list.

e) True. Doubly linked list by default has a tail reference (as given in the lecture notes) so it can add to the back in $O(1)$ time. *Some can argue based on other references or books that not having a tail is default, but we will go along with having a tail as default for this course.* ***In midterms and finals, you can make your own assumptions and answer according to those assumptions. If your assumptions is valid and clearly not contradicted by the question or the content taught in CS2040 and your answer is correct according to your assumptions, your answer will be accepted.***

f) False. First element removed should be A if X is a queue.

Problem 2. Circular Linked List

Implement a method `swap(int index)` in the `CircularLinkedList` class given to you below, to swap the node at the given index with the next node. The `ListNode` class (as given in the lectures) contains an integer value.

```
class CircularLinkedList {

    public int size;
    public ListNode head;
    public ListNode tail;

    public void addFirst(int element) {
        size++;
        head = new ListNode(element, head);
        if (tail == null)
            tail = head;
        tail.setNext(head);
    }

    public void swap(int index) { ... }
}
```

Case 1: size < 2
Do nothing since there is nothing to swap!
• Case 2: size == 2
Simply swap head and tail.
• Case 3: size > 2
Use modulo operation to reduce the index (we don't have to iterate through the entire linked list multiple times).
Note that the order of changing the references matters as you may lose nodes if the references are not changed properly.
curr.setNext(succ.getNext());
succ.setNext(curr);
prev.setNext(succ);

A pre-condition is that the index will be non-negative (index ≥ 0). If the index is larger than the size of the list, then the index wraps around. For example, if the list has 13 elements, then `swap(15)` will swap nodes at indices 2 and 3.

Restriction: You are NOT allowed to:

- Create any new nodes.
- Modify the element in any node.

Hint: Consider all cases, and remember to update the necessary instance attributes/variables!

Solution: Note that we can use modulo operation to reduce the index value so that we need not iterate through the entire linked list multiple times. For swapping two nodes in a singly linked list, you will need to consider three nodes, the two nodes being swapped and the node before the two nodes. Adhering to the restriction, you should change the references on the three nodes to swap the order. Note that the order of changing the references matters as you may lose nodes if the references are not changed properly. The solution should also ensure that edge cases are handled. See `CircularLinkedList.java` for a sample solution.

Problem 3. Waiting Queue

In our day-to-day life, it is common to wait in a queue/line, be it buying a hamburger at McDonald's, or waiting to pay for accommodation at a residence. People join the queue sequentially, and are served in a first-come-first-served manner. However, using pure queue operations such as `enqueue`, `dequeue` and `isEmpty` is not enough, as people in the queue might grow impatient and leave.

In this problem, you are to implement a **WaitingQueue** which contains the names of the people in the queue. In addition to the standard queue behavior, the **WaitingQueue** has a `leave(String personName)` operation that allows a person in the queue with name `personName` to leave at any time. An **array** is used as the underlying data structure. You may assume that the names of

people in the queue are unique, and that no one can join the queue if it is full. Think of **at least two different ways** of implementing the `leave` operation and explain how it works, along with any other changes that are made. Give the time complexity of the `leave` operation and any other operations that have changed.

Solution: Three possible solutions are given below. Note that you will need to maintain two additional integer variables, `front` and `back`, that denotes the index of the start and end of the queue.

- `leave(String personName)` is implemented by searching for the person with the name `personName` from the start of the queue to the end of the queue. If the person is found, we remove it from the array. We now need to also left shift all the remaining elements in the array so that our queue elements are contiguous. The time complexity of `leave` is $O(n)$. See `WaitingQueue.java` for a sample solution.

- (Lazy deletion) Each element in the queue has a boolean flag indicating whether a person has left the queue, or not. If we create a `Person` class, let the flag be one of its attributes, then we can indicate that a person has left. However, `dequeue` will suffer, as we now have to access more than one element in order to clear the deleted objects at the front of the queue. The time complexity of `leave` is however still $O(n)$ since we still need to search for and set the flag for the person unless we already have a reference to it (then it will be $O(1)$) and `dequeue` is $O(n)$ since everyone may have left the queue and we have to dequeue the entire queue.

- We can store the names of the people who want to leave the queue in a separate data structure. When a person is served, the collection is searched to find a matching person. We will learn how to implement a collection that allows elements to be added and searched efficiently later in the semester. The efficiency of leave() is improved to $O(1)$, but the method requires more space. `dequeue` also deteriorates to $O(n)$, as we may have to remove multiple elements until we find someone who has not already left the queue. Meanwhile, the person that already left still takes up one position in this queue before it served, which reduces the valid length of the queue.

Problem 4. Stack Application – Expression Evaluation

In the Lisp programming language, each of the four basic arithmetic operators appears before an arbitrary number of operands, which are separated by spaces. The resulting expressions are enclosed in parentheses. There is only one operator in a pair of parentheses. The operators behave as follows:

- `( + a b c )` returns the sum of all the operands, and `( + )` returns 0.
- `( - a b c )` returns `a - b - c - ...` and `( - a )` returns `0 - a`. The minus operator must have at least one operand.
- `( * a b c )` returns the product of all the operands, and `( * )` returns 1.
- `( / a b c )` returns `a / b / c / ...` and `( / a )` returns `1 / a`, using double division. The divide operator must have at least one operand.

You can form larger arithmetic expressions by combining these basic expressions using a fully parenthesized prefix notation. For example, the following is a valid Lisp expression:
`( + ( - 6 ) ( * 2 3 4 ) )`

The expression is evaluated successively as follows:
`( + -6.0 ( * 2.0 3.0 4.0 ) )`
$\Longrightarrow$ `( + -6.0 24.0 )`
$\Longrightarrow$ `18.0`

Design and implement an algorithm that uses stacks to evaluate a legal Lisp expression with n tokens composed of the four basic operators, integer operands, and parentheses. The expression is well formed (i.e. no syntax error), there will always be a space between 2 tokens, and we will not divide by zero. **Output the result, which will be one double value. What is the time complexity of your algorithm?**
Hint: How many stacks do you need to use?

one for the entire expression and
another to evaluate sub expressions

Solution: The algorithm requires two stacks. We start by pushing the tokens one by one into the first stack until we see the first ")". We then pop tokens in the first stack and push them into the second stack one by one until we pop the element "(". (Note that the tokens are pushed into the second stack in reverse order.) Now in the second stack, the operator is the first tokens to be removed followed by the tokens to be operated on, and we can remove the tokens inside one by one and evaluate the expression in the same order as they were given in the input. The result of the expression is pushed back into the first stack, and we repeat the above steps until all tokens have been processed, and the final answer will be the one remaining value inside A. The second stack is important because not all the operations (subtraction and division) are commutative. The time complexity of the algorithm is $O(n)$, because each token will only be added or removed from a stack not more than 4 times.

Things to take note include handling the edge cases such as `/ a` and `- a`. A sample solution is given in `ExpEval.java`.

An example with the expression `( + ( - 6 ) ( * 2 3 4 )`. We denote the first stack as A and the second stack as B. In the diagrams below, the top of the stack is on the right.

1. The main stack pushes the tokens one by one until it reads ")".

A:

(	+	(	-	6.0				

B:

2. The main stack transfers its tokens to the temporary stack for evaluation.

A:

(	+							

B:

6.0	-							

3. The temporary stack pushes back the result after performing subtraction.

A:

(	+	-6.0						

B:

4. Main stack continues to push tokens until it reads ")".

A:

(	+	-6.0	(	*	2.0	3.0	4.0	

B:

5. Main stack transfers tokens to temporary stack one by one.

A:

(	+	-6.0						

B:

4.0	3.0	2.0	*					

6. Temporary stack pushes back the result after calculation.

A:

(	+	-6.0	24.0					

B:

7. Main stack pushes until ")".
   No change in diagram from 6.

8. Main stack transfers to temporary stack.

A:

B:

24.0	-6.0	+						

9. Temporary stack pushes back final result.

A:

18.0								

B:
