CS2040: Data Structures and Algorithms

Tutorial Problems for Week 5: Lists, Stacks and Queues

For: 12 Sep 2024, Tutorial 3

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 worst case $O(1)$ time.

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

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

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

e) All operations in a queue are worst case $O(1)$ 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.

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) { ... }
}
```

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!

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.

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?