Computer Science 62

Bruce/Mawhorter – Fall ‘16

Midterm Examination

October 5, 2016

<table>
<thead>
<tr>
<th>Question</th>
<th>Points</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

TOTAL 52

SOLUTIONS

Your name (Please print)

____________________________________
1. Suppose you are given a singly-linked list class that holds strings and that maintains pointers to both the head and the tail of the list. Its fields and constructors are as follows:

```java
public class SinglyLinkedList {
    protected ListNode head;
    protected ListNode tail;

    public SinglyLinkedList() {
        this.head = null;
        this.tail = null;
    }

    ...
}
```

The `ListNode` class looks like this:

```java
public class ListNode {
    private String value;
    private ListNode next;

    public ListNode(String value, ListNode next) {
        this.value = value;
        this.next = next;
    }

    public String getValue() {
        return this.value;
    }

    public ListNode getNext() {
        return this.next;
    }

    public String setValue(String newValue) {
        this.value = newValue;
    }

    public ListNode setNext(ListNode newNext) {
        this.next = newNext;
    }
}
```
Please add a new method to the class SinglyLinkedList with header:

```java
public void keep(int howMany) {
    which should modify the list so it only keeps the first howMany elements,
    dropping the rest of the elements from the list. E.g., if myList originally contains
    10 elements, then executing myList.keep(6) should result in myList having
    only the first 6 elements of the list. You don't need to worry about keeping track of
    the discarded nodes as long as you cut them off from the rest of the list.
```

a. Write the pre- and post- conditions for the keep method. Just describe them
   in English.

   **Pre: howMany > 0**
   (adding howMany < size or howMany <= size is okay too)

   **Post: list has <= howMany elements**
   (<= if we accept howMany > size and do nothing)

b. List at least one special case that either violates your preconditions or requires
   special handling.

   **HowMany = 0**
   howMany == size
   howMany < 0

   c. Write the code for keep on the next page (you don't need to worry about
      comments). Remember that you should check your preconditions (you can use
      "RuntimeError" if you need to throw any exceptions).
public void keep(int howMany) {
    if (howMany < 0) {
        throw new RuntimeError("Can’t keep a negative number of elements.");
    } else if (howMany == 0) {
        this.head = null;
        this.tail = null;
    }
    this.tail = this.head; // set tail to head → reduce list to size 1
    while (howMany > 1 && this.tail != null) {
        this.tail = this.tail.next; // set tail to next element, adding 1 to kept size
        howMany -= 1; // decrement counter
    }
    // now we just need to chop off the rest of the list:
    this.tail.next = null;
    // that’s it. We don’t have a size variable to modify or anything like that
}
2. You have a singly linked list with only a head pointer (see the figure below). The `insert()` method for the list inserts new values into the list so that the elements remain in sorted order using the obvious algorithm. In other words, after each insertion, the list is in sorted order. Assume you are given a sequence of \( n \) values to insert one at a time into the list. What do you expect the total worst-case running time to be, using big-O notation, for inserting all of the values into the list? Give a brief (one to two sentence) justification for your answer.

To insert a sequence of \( n \) values will take \( O(n^2) \) time. The reason for this is that on average, inserting the \( n \)th element will take \( n/2 \) time (scanning through the list to find the right place which on average is the center of previously inserted elements). So the runtime is the sum from \( i = 0 \) to \( n \) of \( i \) times a constant \((1/2)\) which is \( O(n^2) \).

b. Suppose that the sequence of \( n \) values to be inserted just happen to be in reverse sorted order. E.g., you might be given the elements 47, 23, 19, 13, 7, 6, and finally 2. What do you expect the running time to be, using big-O notation, for inserting all of \( n \) values into the list? Give a brief (one to two sentence) justification for your answer.

Now the run-time for inserting \( n \) elements will be \( O(n) \), because each insert will be \( O(1) \). This is because each insertion will be smaller than the first element of the list, and so it'll live there without the need to do more than 1 comparison.

Note that for this problem and the one above, the question is asking about the time to insert all \( n \) values, not the time to insert a single value.
4. An advantage of using stacks and queues is that their limited number of operations allows more efficient representation than more general data structures. Please answer the following questions about the complexity of operations on queues, expressing all answers in big-O notation.

A queue may be represented by a “circular” implementation on an array (or ArrayList) or by a singly linked list with a reference to both the front and rear. Please give the complexity of the following queue operations on a queue of size n for each representation, including one or two sentences justifying your answer.

i) Enqueuing an element at the rear of the queue with a circular array:

This is $O(1)$. You just add the element to an empty spot in the array after the current tail and increment your tail index. If the queue is out of space it’s $O(n)$, but with a doubling policy this happens infrequently enough that enqueueuing is still amortized $O(1)$ per operation.

linked list:

This is $O(1)$. You just add a new node to the end and advance the tail pointer.

ii) Dequeing an element from the front of the queue with a circular array:

Still $O(1)$. Just advance the head index by 1.

linked list:

Also $O(1)$ (there’s a pattern here). Just remove head and set new head to old head.next.