Computer Science 62

Bruce/Mawhorter – Fall ‘16

Midterm Examination

October 5, 2016

Question	Points	Score
1	15	____
2	10	____
3	10	____
4	8	____
5	9	____
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:

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

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

    ...
}
```

The ListNode class looks like this:

```
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:

```
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.

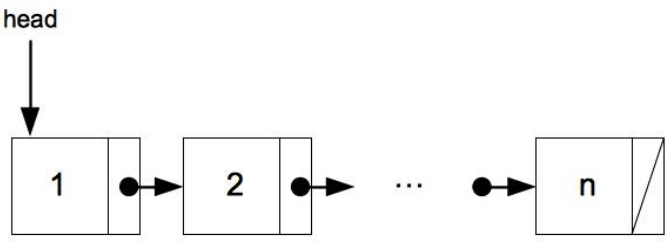
HowMany = 0
howMany == size
howMany < 0

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.

   a. 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 nth 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 (½) 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.

3. Suppose we have a list of ints held as an ArrayList. As you may recall, the elements of the list are held in an array elts, with instance variable eltsFilled keeping track of the number of elements in the list. The standard implementation of an iterator for an ArrayList presents the elements of the array (one at a time) from slot 0 to slot eltsFilled -1.

   However, for a particular application we want to design a special iterator (call it inOrderIterator) that presents the elements of the list in numeric order. Please describe in words (no Java code necessary) how you would initialize the iterator (i.e., what the constructor would do), and how you would write the hasNext() and next() methods.

   You may assume that the iterator is an inner class of the ArrayList class (thus giving it access to the instance variables of ArrayList). The initialization of the iterator should take no more than O (n log n) time in the worst case (where n is the size of the list) and the methods next() and hasNext() should each be O(1). The contents of the array elts should not be modified by the construction or use of the iterator. You may use any extra storage needed to create the iterator.

   To initialize the iterator, you make a copy of the array contents and sort that copy. The hasNext and next methods then just use an index into that sorted array and return the appropriate value from it each time.

   This is O(n log n) for initializing the iterator and O(1) for each call to hasNext and/or next.

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 enqueueing 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.

5. Short answer

a. Describe carefully in words what happens when you insert an element into an ArrayList when it is already filled to capacity.

A new array (not a new ArrayList) is allocated with double the old capacity and all of the old elements are copied over to this new array in an O(n) operation.

b. We noted that when using Java graphics, we must call the method repaint (which the programmer doesn't write) in order to get the computer to eventually call the method paint, which is the one the programmer actually writes. Please explain why this happens and what data structure that we have discussed in class is used to make this all work.

This happens because paint() is a callback: Java is in charge of deciding when to call it. Repaint() is just a way to tell Java: "Hey you know that paint method? Could you please call it?" As for the data structure, the mechanism for this interaction is the event queue that we discussed in class. Repaint puts an event on that queue, and whenever that event is processed, it will trigger a paint call.

c. Explain how the run-time stack changes when a method is invoked and when the method completes execution.

When a method is invoked, an activation record is pushed onto the stack. When a method returns, that record is popped from the stack (this question is just looking for the basics).