CS 220 Software Design II Spring 2022

Final Exam Practice Problems

University of Wisconsin - La Crosse *April 27*

1. Given the `SinglyLinkedList` class on the attached Reference Classes page, write the instance method `replaceByValue` described below (this method will be located in the `SinglyLinkedList` class).

```
/**
 * Replaces all instances of the first given value from the list with the
  second given value and returns true.
 * If the value is not in the list, the list will remain unchanged and should
 * return false.
 * @param oldValue The value of the generic type E to be replaced
 * @param newValue The value of the generic type E to replace with
 * @return true if one or more values were replaced, false if nothing was
 *         replaced
 */
```

2. Given the `DoublyLinkedList` class on the attached Reference Classes page, write the instance method `halfSumsEqual` described below (this method will be located in the `DoublyLinkedList` class). Note that your method **may not** use the `size()` method or the `size` attribute.

```
/**
 * Returns true if and only if the sum of the elements in the first half
 * of the list is equal to the sum of the elements in the second half
 * of the list.  If the list has an odd number of elements, the middle
 * element should be ignored.
 * @return true iff the sum of the first half of the list is equal to the sum
   of the second half
 */
```

3. What is the asymptotic runtime (using O-notation) of your algorithm?

4. Implement a `public` method `listSelectionSort` that has a return type of `SinglyLinkedList<E>`. It creates a new list that contains the nodes from the list to which it was applied in non-decreasing order and consumes the list to which it was applied (i.e. that list now contains no nodes other than the sentinel node). Though implementation details will be different, the method should function in a similar way to the Selection Sort algorithm we examined in class.

```
/**
 * SelectionSort implementation for a SinglyLinkedList.
 * @return a new SinglyLinkedList<E> in sorted order
 */
```

5. What is the asymptotic runtime (using O-notation) of your algorithm?

Reference Classes

```java
public class SinglyLinkedList<E> {
    private int size;
    private SingleListNode firstNode;

    public SinglyLinkedList() {
        size = 0;
        // assumes the use of a sentinel node
        firstNode = new SingleListNode(null);
    }

    private class SingleListNode {
        private E data;
        private SingleListNode nextNode;

        public SingleListNode(E i) {
            data = i;
            nextNode = null;
        }
    }
}
```

```java
public class DoublyLinkedList {
    private int size;
    private DoubleLinkNode head;
    private DoubleLinkNode tail;

    public DoublyLinkedList() {
        head = new DoubleLinkNode(-1, null, null);
        tail = new DoubleLinkNode(-1, head, null);
        head.next = tail;
        size = 0;
    }
    public int size() {
        return size;
    }

    private class DoubleLinkNode {
        private int data;
        private DoubleLinkNode prev;
        private DoubleLinkNode next;

        public DoubleLinkNode(int i, DoubleLinkNode p, DoubleLinkNode n) {
            data = i;
            prev = p;
            next = n;
        }
    }
}
```

6. Write a `public static` non-void method called `countChar` described in the Javadoc comment below. Your solution must be recursive – if you use loops, summer will be canceled and it will be all your fault.

```
/**
 * Computes the number of occurrences of a provided target character in a
 * provided String.
 * @param input The String to be searched
 * @param target The char to search for
 * @return Number of occurrences of target in input
 */
```

7. Write an implementation of the above method that uses a different recursive structure.

8. Recall the recursive implementation of the Fibonacci numbers. Despite the elegance of the implementation, it is very inefficient. Explain the cause of the inefficiency.

9. Write a `public static void` method `removeValue` that goes through a queue and removes all occurrences of a particular value. You may not assume any particular underlying representation for the queue (i.e. you don't know if it's based on a linked list or an ArrayList or something else). You may declare primitive variables but may not declare any data structures.

```
/**
 * Removes all occurrences of a provided value from a provided queue.
 * @param q The queue: (Queue)
 * @param value The target value to be removed from the queue: (E)
 */
```

10. Suppose that you have a working queue implementation and that you want to base a stack implementation on it (presumably because you've suffered a head injury). This means that the stack will use the queue as its underlying data structure). What, if any, methods do you need to add to the queue implementation to make this feasible?