Linked List Mini-Project

Due: Tues, June 26 by 11:30AM

Total Points: 115 (100 for main code; 15 for experiments and writeup)

On Piazza you will find the file `SLL.java` which we have been using in lecture. At the bottom of `SLL.java` you will find ten new methods. Nine of these methods are just "stubs" – i.e., they are not fully implemented. Some of them have dummy return statements so that the file can compile. You should make **no modifications** to `SLL.java` above the stubs (i.e., to the already existing and implemented methods).

You will implement each of these nine methods and submit your new `SLL.java` file. The semantics of the methods are given in comments above each stub. You must also note the runtime requirements!

The methods you need to implement are (recall that `T` is the generic type specifier for the data stored in the list):

```
int size()

T get(int i)

int count(T x)

void clear()

SLL<T> clone()

void reverse()

void concatenateWith(SLL<T> suffix)

// these three go together...
int removeAll(T x)
SLL<String> badCase4SRA(int n);
void removeAllTest(int n);
```

See the comments in the source code for details on semantics, runtime requirements and points allotted to each item. But a few key points here:

- **Only** the `clone` method is allowed to create new list nodes. All other methods must work with the existing nodes only.
- Some methods can be done iteratively or recursively. You are free to choose.
- All runtime requirements are for *worst-case* runtime.
- There is one method – `slowDeleteAll()` which is already implemented. You will not modify this method, but it will play a role in a short writeup (see below).

Removal

The `SLL` class allows duplicates – i.e., a list may have more than one occurrence of the same value. However, the `delete` method already defined only deletes the first occurrence of the given value. What if we want to delete *every* occurrence of the given value?

The method `slowRemoveAll` achieves this goal by repeatedly calling `contains` and `remove`.

QUESTIONS (15 points): Include with the written part of the assignment an answer to the following question:

1. What is the *best case* runtime of `slowDeleteAll`? Describe a situation in which the best case occurs.
2. What is the *worst case* runtime of `slowDeleteAll`? Describe a situation in which the worst case occurs.

   Your implementation of `badCase4SRA(n)` generates such instances.
3. Explain how your implementation of `removeAll` achieves the $O(n)$ worst case runtime.
4. Include data (e.g. as a table or as a graph) experimentally confirming the behavior of `slowRemoveAll` and `removeAll` (collect data using the `removeAllTest()` method you are to complete.

Testing Tips

- Consider writing sanity checker methods which verify properties you expect to always be true (invariants) – e.g., if you find the last element of a list by searching from the head, the result had better match what the tail says.
- Create test cases that exercise boundary cases (e.g. involving first/last elements in a list; lists with consecutive elements that are equal, etc.)
- Create test cases that are perform sequences of operations (e.g., two reverse operations in a row, etc.)
- Mix and match the above suggestions.

What to submit

Submit your source code and your writeup in a single Blackboard submission. Name your writeup file clearly (e.g., `writeup.txt`). Your writeup can be a plain text file or a pdf.