COMP2402: Data Structures

Assignment 3

Grading

This assignment will be tested and graded by a computer program (and you can submit as many times as you like). For this to work, there are some important rules you must follow:

- Keep the directory structure of the provided zip file. If you find a file in the subdirectory `comp2402a3` leave it there.
- Keep the package structure of the provided zip file. If you find a `package comp2402a2;` directive at the top of a file, leave it there.
- Do not rename or change the visibility of any methods already present. If a method or class is `public` leave it that way.
- Submit early and often. The submission server compiles and runs your code, and gives you a mark. You can submit as often as you like and only your best submission will count. There is no excuse for submitting code that does not compile or does not pass tests.
- Write efficient code. The submission server places a limit on how much time it will spend executing your code even when performing millions of operations. For some questions it also places a limit on the stack size. These limits will easily be obtained by data structures and algorithms that are asymptotically optimal: This means $O(\log n)$ time operations for question 1 and $O(n)$ time $O(1)$-stack-size algorithms for question 2.

Submitting and Testing

The submission server is now accepting submissions. Please try it out.

Warning: Do not wait until the last minute to submit your assignment. There is a hard 5 second limit on the time each test has to complete. For the largest tests, even an optimal implementation takes 3 seconds, and may take longer if the server is heavily loaded.

The Assignment

Start by downloading the Assignment 3 Zip File, which contains a skeleton of the code you need to write.

1. Sorted sets with access by rank

For this assignment, you will have to test your own code thoroughly. The submission server will consume anything that you try to print to System.out or System.err so the testing done on the server will be almost completely opaque. It will only give you a generic error of the form "`get(i)/rank(i)/find(x)` returns incorrect value", "an exception occurred", or "test failed after 5 seconds". On the positive side, testing is something you can do easily because `SlowSkiplistRankedSSet` is a correct implementation that you can use to test against. Be sure to test a good mix of operations that includes and interleaves `add(x)`, `remove(x)`, `get(i)` and `rank(x)`.

The `RankedSSet` interface is an extension of the `SSet` interface described in the textbook that supports two extra operations

- `get(i)`: return the value `x` in the set that is larger than exactly `i` other elements in the set.
- `rank(x)`: return the number, i, of elements in the set that are less than `x` (note that `x` is not necessarily in the set).

Currently there are two identical implementations of the `RankedSSet` interface included in the assignment called `SlowSkiplistRankedSSet` and `FastSkiplistRankedSSet`. Both

of them are slow: each of them has an implementation of `get(i)` and `rank(x)` that runs in $\Omega(n)$ time, in the worst case.

Modify the `FastSkiplistRankedSSet` implementation so that the `get(i)` and `rank(x)` implementations each run in $O(\log n)$ expected time and none of the other operations (`add(x)`, `remove(x)`, `find(x)`, etc.) have their running-time increased by more than a small constant factor. Looked to the `SkiplistList` implementation discussed in class for inspiration on how to achieve this.

Suggestion: Once you've decided on your representation, implement a `display()` method for your `FastSkiplistRankedSSet` that outputs an ASCII-graphics representation of your data structure on `System.out`. Here's example output from one that I created while making sample solutions:

```
*-----------------------------------*
*-----------------------------------*
*-----------------------------------*
*-----------------------------------*
*-----------------------------------*-------------*
*-------------------*-----*----------*-------------*
*-------------------*-----*-----*-----*---*-----*-----*
*-------------------*-----*-----*-*---*---*---*-*-----*
*-----------*---*---*-*---*-----*-*---*---*---*-*-----*
*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

Doing this will force you to think about your representation and will also help you debug your implementation. With the right representation, the `display()` method is actually easy to implement, requiring only a pair of nested `for` loops.

2. Tree Traversal Tricks

For this part of the assignment, you will be working with the `BinaryTree` class provided in the zip file. It contains four uncompleted functions that you are supposed to complete. For full marks, each of these functions should run in $O(n)$ time and *should not use recursion*. See the

function `traverse2()` in `BinaryTree.java` that we also discussed in class for an example of how to do tree traversal without recursion.

The `BinaryTree` class implements a static method called `randomBST(n)` that returns a random looking binary tree. `BinaryTree` also implements the `toString()` method so you can use `System.out.println(t)` to view a representation of a `BinaryTree` that is closely related to question 3. You can use this for testing your functions. Again, you should test your own code thoroughly since the testing done on the server will be mostly opaque. On the server, your code will be tested on a Java virtual machine with a limited stack size. You can do similar testing yourself by using the java -Xss argument (although you won't have to worry about this *if you don't use recursion*.).

1. The method `totalDepth()` should return the sum of the depths of all nodes in the tree, or `-1` if the tree has no nodes at all.
2. The method `totalLeafDepth()` should return the sum of the depths of all the leaves in the tree, of `-1` if the tree has no leaves. (A leaf is a node that has no left child and no right child.)
3. The method `bracketSequence()` should return a string that gives the *dot-bracket representation* of the binary tree. The dot-bracket representation of a binary tree can be defined as follows:
    - the dot-bracket representation of a tree with no nodes is the string `"."`
    - the dot-bracket representation of a binary tree with root node `r` consists of an open bracket `(` followed by the dot-bracket representation of `r.left` followed by the dot-bracket representation of `r.right` followed by a closing bracket `)`
      Some examples:
    - the dot-bracket representation of the binary tree with only one node is `(..)`
    - the dot-bracket representation of a 2-node binary tree is either `((..).)` or `(.(..))` depending on whether the root has no right child or no left child
    - the dot-bracket representation of a the height-1 binary tree with two leaves is `((..)(..))`

4. The method `prettyPrint(w)` prints an ASCII drawing of the binary tree on the `PrintWriter w` (using `w.print(s)` and `w.println(s)`, like in Assignment 1). In this representation,
   - each node is represented by an asterisk `*`
   - an edge from a node to its left child is represented by a vertical bar `|`
   - an edge from a node to its right child is represented by a sequence of one or more hyphens `-`

     For each edge from a node `u` to its right child `v`, the number of hyphens used to represent the edge `uv` should be just enough so that the drawing of the subtree rooted at `v` is separated from the drawing of the subtree rooted at `u.left` by exactly one column. See the example here:

```
*-------*-*-----*-*
|       | |     |
*---*-* * *-*-* *
|   |     |
*-* *     *
|
*
```