The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algorithms

Assignment 5

Date Due: February 28, 2017, 10:00pm

Total Marks: 63

1 Submission Instructions

- Assignments must be submitted using Moodle.
- Responses to written (non-programming) questions must be submitted in a PDF file, plain text file (`.txt`), Rich Text file (`.rtf`), or MS Word's `.doc` or `.docx` files. Digital images of handwritten pages are also acceptable, provided that they are **clearly** legible.
- Programs must be written in Java.
- If you are using IntelliJ (or similar development environment), do not submit the Module (project). Hand in only those files identified in Section 5. Export your `.java` source files from the workspace and submit only the `.java` files. **Compressed archives are not acceptable.**
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.

2 Background

2.1 Efficient Insertion and Deletion in AVL Trees

For AVL trees, we need to be able to identify critical nodes to determine if a rotation is required. After each recursive call to insert, we need to check whether the current node is critical (the `restoreAVLProperty` algorithm). This means we need to know the node's imbalance, which means we need to know the heights of its subtrees. If we compute the subtree heights with the recursive post-order traversal we saw in class, we are in trouble, because this algorithm costs $O(n)$ time, where n is the number of nodes in the tree. Since insertion requires $O(\log n)$ checks for critical nodes, computing imbalance in this way makes insertion $O(n \log n)$ in the worst case. To avoid this cost, in each node of the AVL tree we have to store the heights of both of its subtrees, and update these heights locally with each insertion and rotation.

The insertion algorithm from the AVL tree slides becomes:

```
// Recursively insert data into the tree rooted at R
Algorithm insert(data, R)
data is the element to be inserted
R is the root of the tree in which to insert 'data'

// This algorithm would only be called after making sure the tree
// was non-empty.  Insertion into an empty tree is a special case.

if data <= R.item()
    if( R.leftChild == null )
        R.leftChild = new node containing data
    else
        insert(data, R.left)
    recompute R.leftSubtreeHeight  // Can be done in constant time!
else
    if( R.rightChild == null )
        R.rightChild = new node containing data
    else
        insert(data, R.right)
    recompute R.rightSubtreeHeight // Can be done in constant time!

restoreAVLProperty(R)
```

Observe that if we recurse the left subtree, then when that recursion finishes, we recompute the left subtree height but not the right subtree height (since we know the right subtree didn't change!). Likewise, if we recurse right, then only the right subtree's height need be recomputed when the recursive call returns (the left subtree didn't change!).

As the above algorithm indicates, the key is to recompute subtree heights in constant time. Let p be a node, and let l and r be p's left and right children, respectively. As long as l and r's subtree heights are correct, then the subtree heights for p can be recomputed in constant time. For example, if l.leftHeight and l.rightHeight are correct, then we can recompute p's left subtree height as:

$$\max(l.\text{leftHeight}, l.\text{rightHeight}) + 1$$

Since we are adjusting subtree heights starting from the bottom level of the tree where the new node was inserted, we can correct subtree heights at each level in constant time as the recursion unwinds back up the tree because all children of the current node will already have correct subtree heights.

It is important to note that, in addition to recomputing subtree heights as the recursive calls to `insert` unwind, it is also necessary to correct subtree heights whenever a rotation occurs. Rotations rearrange nodes, and change the heights of subtrees. For both the left and right rotations, there are exactly two nodes involved for which one subtree height has to be recomputed. Again, this can be done in constant time similar to as above. It will be up to you to figure out the exact details of adjusting the subtree height during rotations. Double rotations should pose no additional problems since they can be written in terms of two single rotations.

3 Your Tasks

Question 1 (63 points):

Implement an AVL tree ADT. The design of the class is up to you. You may use as much or as little of `lib280-asn5` as you desire (including taking pieces of code for your own methods, or not using any of `lib280-asn5` at all) as long as the following requirements are met:

- Your ADT must support insertion of new elements, one at a time. Elements may be of any object type, but all elements in a single tree are the same type.
- The implementation of insertion must be $O(\log n)$ in the worst case. This means you have to store subtree heights in the nodes (see previous section) to avoid incurring the linear-time cost of a recursive `height` method.

  Part marks will be given for solutions that are at worst $O(n \log n)$, but such a solution will receive a major deduction. No marks will be given for solutions where insertion and deletion is worse than $O(n \log n)$.
- The tree must restore the AVL property after insertions using (double) left and right rotations.
- Your ADT must have an operation for determining whether a given element is in the tree.
- Your ADT must have an operation that deletes an element. The mechanism for specifying the item to delete is up to you.
- The implementation of deletion must be $O(\log n)$ in the worst case (as insertion) and must restore the AVL property after deletions using (double) left and right rotations.

  Again, part marks will be given for solutions that are at worst $O(n \log n)$, but such a solution will receive a major deduction. No marks will be given for solutions where insertion and deletion is worse than $O(n \log n)$.
- You must include a test program (in a `main()` method) that demonstrates the correctness of your implementation. The purpose of this test program is to demonstrate to the markers that your ADT meets the above requirements. Your program's output should be designed to convince the marker that your insertion, rotation, and lookup, and search methods work correctly. Note that the goals here are different from a normal "regression test", so console output even when there are no errors is acceptable. **The onus is on you to use your test program to demonstrate that your implementation works.**
- You may not modify any existing classes in `lib280-asn5`. If you want to change something, create new class or interface, inherit what you want form lib280, and add/override things as needed.

Evaluation

For this question, 16 marks will be awarded based on the quality of the design of your class(es). Your use of modularization, encapsulation, and choices when using or not using protected/private methods will be considered. That said, there is no need to be overly fancy. If appropriate, make use of existing classes in `lib280-asn5`, but consider your options before you start and think about which of the possible approaches will be easier to implement.

12 Marks are allocated to good commenting. This includes both inline comments and Javadoc comments for each method header and instance variable.

The remaining 35 marks will be for allocated for the correctness (as demonstrated by your unit test) and efficiency of the implementation of the insertion (15 marks), lookup (5 marks), and deletion (15 marks) operations.

Hints and Notes

- There's a reason that there is only one question on this assignment: it's a longer question and requires a lot more thought than the problems you've been given up to this point. It's not that the solution is particularly difficult, but it requires planning! Make sure you give yourself ample time to attempt it, and ask for help if you get stuck. Seriously, I'm not kidding here! While I have given you ample time to do this assignment, if you wait until the day before the due date begin, there is a high probability that you will fail or not complete the assignment in time.
- Your early design choices can have an impact on how hard the implementation is (indeed this is true of **any** non-trivial software project!). Think things through before you begin coding. Sketch out the class architecture (UML diagrams are good for this!), and/or algorithms on paper first.
- Steal the `toStringByLevel` method (found in `LinkedSimpleTree280`) and modify it so that prints the left and right subtree heights along with each node's contents. This will help immensely with debugging because even if you implement insertions and rotations correctly, you'll get incorrect results if the subtree heights are recorded incorrectly. This is because incorrect subtree heights will trigger rotations when none are actually needed, or prevent necessary rotations from occurring.
- If you choose to use a cursor, your ADT need not have methods that allow the user full control over the cursor position (e.g. `goFirst()`, `goForth()`, `before()`, etc.).
- Make sure everything else is working flawlessly before you attempt the delete operation. If you design things well, the delete operations should be able to re-use all of the work you did on rotations for the insertion operation.

4 Files Provided

lib280-asn5: A copy of lib280 which includes:

- solutions to assignment 4;

5 What to Hand In

AVLTree280.java: Your AVL tree implementation.

a4q1.txt: A copy of your test program's output, cut and pasted from the IntelliJ console window.

Other .java files: Any other .java files that you might have created for the implementation of your AVL tree.