The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algoirthms

Assignment 4

Date Due: February 26, 2019, 9:00pm

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 project or module folder. Hand in only those files identified in Section 5. Export your `.java` source files from the workspace and submit only the `.java` files.
- No late assignments will be accepted. See the course syllabus for the full late assignment policy for this class.

2 Background

2.1 Heaps

A *heap* is a binary tree which has the following *heap property*: the item stored at a node must be at least as large as any of its descendents (if it has any). In a heap, when an item is removed, it is always the largest item (the one stored at the root) that gets removed. Also, the only item that is allowed to be inspected is the top of the heap, in much the same way that the only item of a stack that may be inspected is the top element. Stacks, queues, and heaps are all examples of collections of data items that we call *dispensers*. You can put stuff into a dispenser, but the user doesn't get to specify where – the collection decides according to some rule(s). Likewise, you can take something out of a dispenser, but the dispenser decides what item you get. Dispensers maintain a current item using an internal cursor, but the dispenser always decides what is the current item, and thus the item that will next be *dispensed* when a user asks to remove or inspect the current item. Dispensers do not have public methods to control the cursor position because the user is not supposed to control this; it's up to the dispenser. In a stack, the "current" item is always the item at the top of the stack. In a queue it is the item at the front of the queue. In a heap it is the item at the root of the heap.

In question 1 you will implement a heap by writing a class called `ArrayedHeap280` that extends the abstract class `ArrayedBinaryTree280<I>` and implements the `Dispenser280<I>` interface. Here are brief pseudocode sketches of the `insert` and `deleteItem` algorithms:

```
Algoirthm insert(H, e)
Inserts the element e into the heap H.

Insert e into H normally, as in ArrayedBinaryTreeWithCursors280<I>
// (put it in the left-most open position at the bottom level of the tree)

while e is larger than its parent and is not at the root:
   swap e with its parent
```

```
Algorithm deleteItem(H)
Removes the largest element from the heap H.

// Since the largest element in a heap is always at the root...
Remove the root from H normally, as in ArrayedBinaryTreeWithCursors280<I>
// (copy the right-most element in the bottom level, e, into the root,
// remove the original copy of e.)

while e is smaller than its largest child
   swap e with its largest child
```

2.2 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 (10 points):

Implement a heap by writing a class called `ArrayedHeap280` that extends the existing abstract class `ArrayedBinaryTree280<I>` and implements the `Dispenser280<I>` interface. The only methods you should need to write are a constructor, and the `insert` and `deleteItem` methods required by `Dispenser280<I>`, but you can use additional private methods if you think it makes sense to do so. The algorithms for insertion and deletion are given in Section 2.1 of this document.

Note that since you need to compare items in the heap to each other, you must write the generic type parameter of your class' header so that it is required to be a subclass of Java's `Comparable` interface. This is necessary to ensure that only items that implement the `Comparable` interface can be stored in the heap. As a consequence, you'll have to make sure that your constructor initializes the instance variable `items` (inherited from `ArrayedBinaryTree280<I>`) to an array of `Comparable`. If this is all done properly, then any item `x` in the heap can be compared to another item `y` using `x.compareTo(y)`.

A regression test is provided for you in the `heaptests.txt`. Copy the two functions therein into your `ArrayedHeap280` class. You do not need to write your own test. The test functions will likely exhibit compiler errors if your class does not have the correct class header. Your implementation is highly likely to be correct if it passes the provided regression test.

While you should always practice good commenting habits, we will not be grading javadoc or inline comments on this question.

Question 2 (53 points):

Implement an AVL tree ADT. The design of the class is up to you. You may choose to use as much or as little of `lib280-asn4` as you desire (including extending existing classes, implementing existing interfaces, taking pieces of code for your own methods, or not using any of `lib280-asn4` at all) but you will be graded on these choices, so make good choices.

Regardless of your high-level design decisions, you can earn full marks for correctness of implementation as long as the following requirements are met:

- Your AVL tree ADT must be able to contain elements of any comparable object type, but all elements in a single tree are the same type.
- Your ADT must support insertion of new elements, one at a time. Additionally, the insertion algorithm implementation must have the following properties:
  - It 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.
  - You are free to choose whether or not to allow duplicate items in the AVL tree.
- 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. Additionally, the deletion algorithm implementation must have the following properties:
  - It must be $O(\log n)$ in the worst case. 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 $O(n \log n)$.
  - It must restore the AVL property after deletions using (double) left and right rotations.

- 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 not only acceptable, but required. For example, you can print out the tree, describe what operation is about to be performed, and then print the resulting tree. **The onus is on you to use your test program to demonstrate that your implementation works.**. Thus you should demonstrate cases that invoke all of the different kinds of rotations and special cases that might arise. Your examples should be non-trivial, but simple enough that they can be easily verified by inspection.
- You may not modify any existing classes in `lib280-asn4`. But you may make new classes as you see fit.

Hints and Notes

- Start on this question early! 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. If you wait until the day before the due date begin, there is a high probability that you will not complete this question. You don't need to **finish** early, but you should start planning early. Also keep in mind that partial solutions can earn partial marks.
- 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.). That is, your class does not need to implement `LinearIterator280<I>`.
- Make sure everything else is working correctly 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.

Evaluation of Question 2

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

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-asn4`, but consider your options before you start and think about which of the possible approaches will be easier to implement.

The remaining marks will be for allocated for the correctness (as demonstrated by your test program) and efficiency of the implementation of the insertion, lookup, and deletion operations. The detailed grading rubric can be found on Moodle.

4 Files Provided

lib280-asn4.zip: Contains the `ArrayedBinaryTree280<I>` class that you will extend in Question 2.

heaptests.txt: Functions for testing your `ArrayedHeap280<I>` implementation.

5 What to Hand In

ArrayedHeap280.java: Your arrayed heap implementation.

AVLTree280.java: Your AVL tree implementation.

a4q3.txt: A copy of your AVL tree 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.