The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algorithms

Assignment 5

Date Due: March 6, 2018, 10:00pm

Total Marks: 54

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 Restrictions of ADTs

We have seen how we can extend the functionality of a data structure that we already have implemented by creating a subclass using the `extends` keyword in Java. This allows us to take an existing data structure class and add new functionality or modify existing ADT functionality (e.g. extension of `ArrayedTree280` to `ArrayedHeap280`), or to create a specialization of a more generic ADT (e.g. extension of `LinkedSimpleTree280<String>` to `ExpressionTree`).

Another ADT programming paradigm is *restriction*. Restriction is used to make an existing ADT appear to be another ADT that has **less** functionality. Consider the following example. Suppose we need a Stack ADT, but our data structure library, whatever it is, doesn't have one. Further suppose that our data structure library **does** have a list ADT which allows insertions and deletions from either end of the list. Such a list has *more* than the necessary functionality of a stack. A stack is really just a list where we can only add (push) and remove (pop) items at one end. We could just use the list itself as a stack, trusting ourselves to only add and remove at one end. But this does not eliminate the possibility that the "stack" could be used in ways that a stack should not be used when all we really need is a stack. Perhaps another programmer comes along and doesn't realize that you were using a list to get the behaviour of a stack and does something very un-stack-like to the list without realizing it! What is really needed in this situation is an ADT that has *less* functionality than the one we already have. We can't get that by extending the existing list ADT, but we can use the idea of *restriction* to make the list appear to the outside world as nothing more than a stack!

The solution is to *restrict* the list class by writing a stack class which includes a list as an instance variable, and methods that consist of only the interface for a stack. In other words, the list is the internal data structure for the stack ADT, but the public operations for the ADT consist of only those for a stack. These methods then "translate" the stack operations into the corresponding list operations, thus resulting in a new class that looks exactly like a stack to the outside world, but implements the stack using a list.[1] If you take a look at the `Stack280` class in lib280, you'll see that this is precisely what `Stack280` does. Its methods translate stack operations into operations on an internal list. Implementing this restriction is far less work than writing a new linked or array-based stack class from scratch.

You will practice the concept of restriction in Question 1, below.

2.2 Hash Tables

In this question you will work with an implementation of a *chained hash table*. `lib280-asn5` contains the class `KeyedChainedHashTable280`. You will use this class to solve a problem. `KeyedChainedHashTable280` is an example of a *keyed dictionary* where each item is associated with a unique *key*. A keyed hash table computes the hash of an item from **only** the item's key, even though the item itself might contain other data. Additional background and details about chained hash tables are part of Lecture 15 and will be a component of this week's tutorials.

[1] Some might call this a "wrapper" class because all it does is "wrap" one ADT in a more restrictive interface, and passes all the work on to the inner ADT.

3 Your Tasks

Question 1 (34 points):

Recall from assignment 2 that a priority queue is a queue in which there is a numeric priority associated with every item in the queue. The item at the head of a priority queue is always the item with highest priority. If we think about it, a heap actually has some of the functionality of the priority queue ADT we specified in assignment 2. We can insert items into a heap which are kept organized by their size (equivalent to enqueue!), and a heap always "dispenses" the largest item (equivalent to getting the item at the front of the priority queue), and it allows us to delete the largest item (equivalent to a dequeue!). **The problem we would like to solve is to implement the priority queue ADT specification from assignment 2 by re-using as much of our existing code as we can.** You can find the priority queue specification in the appendix of this document.

In Assignment 4 we wrote a class `ArrayedHeap280` (the instructor's `ArrayedHeap280` is included in `lib280-asn5`), which has some of the functionality we need for a priority queue. But we cannot just extend it because:

1. it has methods that are priority queue ADT doesn't have, like `itemExists`;
2. it has methods that have the right functionality, but the wrong name, like `deleteItem`, which, for our priority queue, is called `deleteMax`; and
3. it doesn't have an iterator, which we need in order to determine the item with the smallest priority (we need to be able to inspect all of the items in the heap to find the smallest one without moving the internal cursor away from the root of the heap).

The solution will be as follows:

(a) Write a class `ArrayedBinaryTreeIterator280` which extends `ArrayedBinaryTreePostion280` and implements the `LinearIterator280` interface. This will be an iterator for the `ArrayedBinaryTree280` class. `ArrayedBinaryTreeIterator280` should be fairly easy to implement since you can pretty much copy all of the methods required by the `LinearIterator280` interface from `ArrayedBinaryTreeWithCursors280`, with perhaps some small modification. A mostly incomplete `ArrayedBinaryTreeIterator280.java` file has been provided in the `tree` package of `lib280-asn5` to start you off.

(b) Extend `ArrayedHeap280` to a class `IterableArrayedHeap280` and write the following methods in `IterableArrayedHeap280`:

- Add a `deleteAtPosition` method to delete a specific item in the heap (which need not be the root). The item to be deleted should be specified by passing a reference to an `ArrayedBinaryTreeIterator280` object. The algorithm for this is just a slight modification of the heap deletion algorithm where you swap the item at the end of the array with the item to be deleted, and then swap it with its larger child until it is larger than both its children. This method should be very similar to `deleteItem` from `ArrayedHeap280`.
- Add a method to `IterableArrayedHeap280` called `iterator` which returns a new `ArrayedBinaryTreeIterator280` object for the tree.

A mostly incomplete `IterableArrayedHeap280.java` file has been provided in the `tree` package `lib280-asn5` to start you off.

(c) Write a class `PriorityQueue280` which is a *restriction* (as defined in Section 2.1) of `IterableArrayedHeap280` which implements the priority queue ADT specification given in the Appendix of this document. This means that `PriorityQueue280` should have an `IterableArrayedHeap280` as an instance variable, and it is in the heap that the queue items are stored.

In this way we can hide the functionality of `IterableArrayedHeap280` that we don't want exposed, as well as add the functionality that it lacks.

Some of the priority queue methods, like `isFull` and `deleteMax`, require identical behavior to existing methods in the `IterableArrayedHeap280` and can be written as a single call to a method in the `IterableArrayedHeap280` instance variable.

Other methods, like `minItem` and `deleteAllMax`, require functionality that doesn't exist in `IterableArrayedHeap280` which will be up to you to implement – this will still be done by calling methods of the internal heap, but a single call won't be enough, for example, you may need to iterate over the heap using an iterator.

I have provided you with a mostly incomplete `PriorityQueue280.java` file in the `dispenser` package `lib280-asn4` which contains a regression test for your convenience (you do not have to write your own).

(d) Comment all of the methods in all classes that you wrote by adding a javadoc comment header, and inline comments where appropriate.

Implementation Hints

Since items in the `IterableArrayedHeap280` must implement `Comparable`, your priority queue may assume that the `compareTo` method of the items compares items based on their priority.

Question 2 (20 points):

Almost every modern video game in the roleplaying genre provides the player with a *quest log* which is essentially a list of tasks that the player's character has to perform to advance the story or to obtain rewards.[2] In this question we are going to create a data structure for a quest log based on a chained hash table. For our purposes, a *quest log entry* will consist of the following pieces of information:

- Name of the quest.
- Name of the area of the world in which the quest takes place.
- Recommended minimum character level that should attempt the quest.
- Recommended maximum character level that should attempt the quest.

For the purposes identifying quests, **quest log entries are keyed on the name of the quest.** A class called `QuestLogEntry` that can hold these pieces of information is provided. Notice how the `key()` method of the `QuestLogEntry` class returns the name of the quest.

For this question, you are provided with a complete IntelliJ module called `QuestLog-Template` in which you will modify one of the classes. It includes the `QuestLogEntry` class and some `.csv` (comma-separated value) files which will be used as input data. The `QuestLog-Template` module requires access to the `lib280-asn5` project, set this up as a module dependency as per the self-guided tutorial on the class website for setting up an IntelliJ module to use `lib280`. You've already done this sort of thing previously with other assignments.

The entirety of your work will be to finish implementing methods in the `QuestLog` class provided in the `QuestLog-Template` project, and to write a couple of interesting tests. Note that the `QuestLog` class is a specialized extension of `KeyedChainedHashTable280`, so it is a chained hash table. Here is a list of what you have to do:

(a) Complete the implementation of the `QuestLog.keys()` method. This method must return an array of the keys (i.e. the quest names) of each `QuestLogEntry` instance in the hash table. The keys may appear in the returned array in any order.

(b) Complete the implementation of the `QuestLog.toString()` method. This method should return a printable string consisting of the complete contents of all of the `QuestLogEntry` objects in the hash table *in alphabetical order by the quest names*, one per line. Here is an example string returned by `toString()` from a quest log containing four entries:

```
Defeat Goliad : Candy Kingdom, Level Range: 20-25
Locate the Lich's Lair : Costal Wasteland, Level Range: 35-40
Make an Amazing Sandwich : Finn's Treehouse, Level Range: 1-5
Win Wizard Battle : Wizard Battle Arena, Level Range: 2-4
```

Remember that the hash table makes no promises whatsoever about the ordering of the quest log entries in the chains of the hash table. Hint: the `keys()` method from part (a) will be handy for this method, as will knowing that `Arrays.sort()` can sort the elements of an array.

(c) Complete the implementation of the `QuestLog.obtainWithCount()` method. This method takes a quest name as input and returns a `Pair280` object (found in `lib280.base`) which must contain the `QuestLogEntry` object from the quest log which matches the given quest name (if it exists) and the number of `QuestLogEntry` objects that were examined while searching for the desired one. The latter number must be present whether the quest name was found in the quest log or not.

Hints: A `Pair280` object has two generic type parameters, the first of which specifies the type of the first element of the pair, and the second of which specifies the type of the second element in the pair. For example, if I wanted a pair consisting of an `Integer` and a `Float` I might write:

[2] Back in '80s and early '90s, games didn't have quest logs. If you wanted to remember what you were supposed to do, or something that a character in the game said, you **wrote it down** with an ancient mystical device called a **pencil**. And there was no Internet with detailed wikis for every game to get hints from if you got stuck!

```
Pair280<Integer, Float> p = new Pair280<Integer, Float>( 5, 42.0 );
```

The components of the pair can be accessed using the `firstItem()` and `secondItem()` methods:

```
System.out.println(p.firstItem());    // prints the integer 5
System.out.println(p.secondItem());   // the floating point number 42.0
```

(d) Now take a look at the `main()` program in `QuestLog.java`. As given, it already does the following things:

1. Creates a new, empty `QuestLog` instance called `hashQuestLog`.
2. Creates a new, empty `OrderedSimpleTree280` instance (a binary search tree) called `treeQuestLog` that can hold items of type `QuestLogEntry`.
3. A `.csv` file containing the data for a number of `QuestLogEntry` objects is opened, and read in, creating a `QuestLogEntry` instance for each quest, and adding each such instance to both both the `hashQuestLog` and `treeQuestLog` data structures.
4. The complete contents of `hashQuestLog` and `treeQuestLog` are printed out using their respective `toString()` methods. You'll know when your `Questlog.toString()` method from part (b) is working when its output matches that of `treeQuestLog.toStringInorder()`.

At the end of `main()` are two TODO markers. For the first one, you need to write code that calls `hashQuestLog.obtainWithCount()` for each quest in the hashed quest log and determines the average number of `QuestLogEntry` objects that were examined over all such calls.

Finally, for the second TODO marker, you have to do the same thing for `treeQuestLog`. You can do this by calling the `searchCount()` method of `treeQuestLog` for each quest stored in the log. Note that `searchCount()` requires that you pass in the actual `QuestLogEntry` object that you are looking for rather than just the quest name (you can obtain these from the hashed quest log). `searchCount()` returns the number of items that were examined while trying to position the tree's cursor at the given `QuestLogEntry` object.

Once you have computed the average number of `QuestLogEntry` objects examined for each of the two data structures, print out the results. Something like this will do:

```
Avg. # of items examined per query in the hashed quest log with 4 entries: 1.25
Avg. # of items examined per query in the tree quest log with 4 entries: 2.0
```

(e) Run your completed `main()` program for each of the `.csv` files provided in the project (just change the filename in quotes that is passed to `FileReader`). Each `.csv` file contains in its filename the number of quest entries in the file. For each `.csv` file, record the reported average number of items examined per query in each data structure. In a text file called `a4q2.txt` (or other acceptable file format) answer the following questions:

1. List the reported averages that you recorded for each .csv input file in a table that looks something like this (filling in the rest of the table of course):

Filename	Avg. Queries for `hashQuestLog`	Avg. Queries for `treeQuestLog`
quests4.csv	1.25	2.0
quests16.csv		
quests250.csv		
quests1000.csv		
quests100000.csv		

2. If you had to choose a simple function (i.e. from the list of functions used in big-O notation) to characterize the behaviour of the the average number of items examined per query for the **hashed** quest log as the number of quests (n) in the log increases, what would it be?
3. If you had to choose a simple function (i.e. from the list of functions used in big-O notation) to characterize the behaviour of the the average number of items examined per query for the **tree** quest log as the number of quests (n) in the log increases, what would it be?

4. If your primary use of the quest log was to display all of the quests in the log in alphabetical order, would you prefer the hashed quest log or the tree quest log? Why?
5. If your primary use of the quest log was to periodically look up the details of specific quests in no particular order, would you prefer the hashed quest log or the tree quest log? Why?

4 Files Provided

lib280-asn5: A copy of lib280 which includes:

solutions to assignment 4;

ArrayedBinaryTreeIterator280.java A mostly incomplete implementation of a linear iterator for arrayed binary trees;

IterableArrayedHeap280.java A mostly incomplete implementation of a heap which provides an iterator and allows any item to be deleted; and

PriorityQueue280.java: A mostly incomplete implementation of our priority queue ADT (in the lib280 dispenser package).

QuestLog-Template.zip: A complete IntelliJ module containing everything you need for Question 2.

5 What to Hand In

ArrayedBinaryTreeIterator280.java Your completed implementation of a linear iterator for arrayed binary trees.

IterableArrayedHeap280.java Your completed implementation of a heap which provides an iterator and allows any item to be deleted.

PriorityQueue280.java: Your completed implementation of the priority queue ADT.

QuestLog.java: Your completed quest log based on a hash table (parts (a), (b), and (c) of Q2), and completed additions to `main()` (part (d) of Q2).

a4q2.txt: Your answers to the questions posed in part (e) of Q2.

Appendix – Priority Queue ADT Specification

This is the Priority Queue ADT specification from Assignment 2, but with the `frequency` operation omitted. You need to implement only the operations shown here.

Name: PriorityQueue<G>

Sets: Q : set of priority queues containing elements from G.

G : set of items that can be in a priority queue.

B : $\{true, false\}$

$\mathbb{N}$: set of positive integers.

$\mathbb{N}_0$: set of non-negative integers.

Signatures: newPriorityQueue<G>: $\mathbb{N} \rightarrow Q$

Q.insert(g): $G \not\rightarrow Q$

Q.isFull: $\rightarrow B$

Q.isEmpty: $\rightarrow B$

Q.count: $\rightarrow \mathbb{N}_0$

Q.maxItem: $\not\rightarrow G$

Q.minItem: $\not\rightarrow G$

Q.deleteMax: $\not\rightarrow Q$

Q.deleteMin: $\not\rightarrow Q$

Q.deleteAllMax: $\not\rightarrow Q$

Preconditions: For all $q \in Q$, $g \in G$,

q.insert(g): queue is not full

q.maxItem: queue is not empty

q.minItem: queue is not empty

q.deleteMax: queue is not empty

q.deleteMin: queue is not empty

q.deleteAllMax: q must not be empty.

(Operations without preconditions are omitted)

Semantics: For all $n \in N$, $g \in G$, $n \in \mathbb{N}$

newPriorityQueue<G>(n): create a new queue with capacity n.

q.insert(g): insert item g into t in priority order with the highest number being the highest priority.

q.isFull: return *true* if t is full, $false$ otherwise

q.isEmpty: return *true* if t is empty, $false$ otherwise

q.count: obtain number of items in q

q.maxItem: return the largest (highest priority) item in q.

q.minItem: return the smallest (lowest priority) item in q.

q.deleteMax: remove the largest (highest priority) item in q from q.

q.deleteMin: remove the smallest (lowest priority) item in q from q.

q.deleteAllMax: all occurrences of the highest priority item are deleted from q.