The University of Saskatchewan

Saskatoon, Canada

Department of Computer Science

CMPT 280– Intermediate Data Structures and Algoirthms

# Assignment 3

Date Due: January 31, 2017, 10:00pm

Total Marks: 55

## 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 Eclipse (or similar development environment), do not submit the workspace (project). 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 3 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
```

Additional background and examples on heaps are available in this week's tutorial.

## 2.2 *m*-ary Trees

An *m*-ary tree is one in which a node may have up to *m* children. Your `lib280-asn3` project has a class called `BasicMAryTree280<I>` which implements an *m*-ary tree. It has some similarities with `LinkedSimpleTree280<I>` in that, like `LinkedSimpleTree280<I>`, you have to build up larger trees from smaller trees, rather than inserting individual elements, because since *m*-ary trees have no defined structure in general and thus there is no obvious algorithm for automatically deciding where a new element should go. You will use this class in Question 4. More details and some examples on how to use this class are provided in this week's tutorial.

## 3 Your Tasks

### Question 1 (15 points):

A priority queue is a queue where a numeric priority is associated with each element. Access to elements that have been inserted into the queue is limited to inspection and removal of the elements with smallest and largest priority only. A priority queue may have multiple items that are of equal priority.

Give the ADT specification for a bounded priority queue using the specification method described in Topic 5 of the lecture notes. By "bounded", it is meant that the priority queue has a maximum capacity specified when it is created, and it can never contain more than that number of items.

Your specification must specify the following operations:

**newPriorityQueue:** make a new queue

**insert:** inserts an element with a certain priority

**isEmpty:** test if the queue is empty

**isFull:** test if the queue is full

**maxItem:** obtain the item in the queue with the highest priority

**minItem:** obtain the item in the queue with the lowest priority

**deleteMax:** remove from the queue the item with the highest priority

**deleteAllMax:** remove from the queue all items that are tied for the highest priority

**deleteMin:** remove from the queue the item with the lowest priority

**frequency:** obtain the number of times a certain item occurs in the queue (with **any** priority)

### Question 2 (6 points):

Use the **statement counting approach** to show that the worst-case time complexity of the heap insertion and deletion algorithms given above in Section 2.1 are $O(\log n)$.

### Question 3 (16 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.

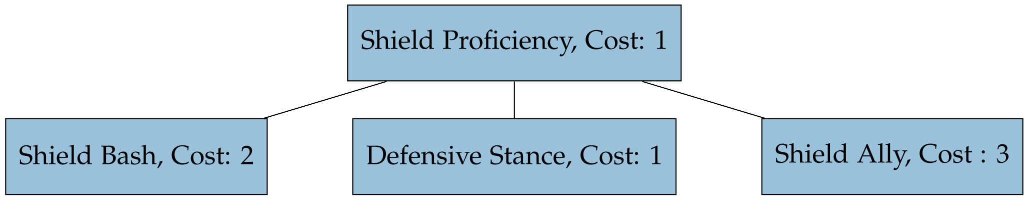
Note that since you need to compare items in the heap to each other, you must write the generic type parameter of your class' header so that it is required to be a subclass of Java's `Comparable` interface. This is necessary to ensure that only items that implement the `Comparable` interface can be stored 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)`.

Finally, you are required to write a `main` method for the `ArrayedHeap280` class that performs a regression test to make sure the class functions properly. You need only test the methods in the `ArrayedHeap280` class, as one would normally assume that the regression test for the parent class has already been run.

**For this question, there will be marks awarded for suitable commenting of code, and for writing appropriate javadoc comments.**

## Question 4 (18 points):

In video games, especially those in the roleplaying genre, it is common that characters in the game are advanced in power through the use of a *skill tree*. Generally, a skill tree defines the prerequisite for the various skills that your character in the game might acquire. For example, in a hypothetical game, if the *Shield Bash*, *Defensive Stance*, and *Shield Ally* skills all require that your character first have the skill *Shield Proficiency*, then this might be represented by the following skill tree:



More formally, a skill in the skill tree can only be gained if the character first gains all of the skills which are ancestors of that skill in the tree.[1]

Your task in this question is to write a class called `SkillTree` which extends `BasicMAryTree280<Skill>` (an $m$-ary tree of `Skill` objects; a complete `Skill.java` is provided). A template for the `SkillTree` class is provided. It contains a constructor and a couple of useful methods. You will add additional methods to this class in the following steps, which you should complete in order:

(a) Write a `main()` method in the `SkillTree` class in which you construct your own skill tree for your own hypothetical video game. Your tree must contain at least 10 skills. However, for the sanity of everyone involved, try to keep it under 15 skills. Be creative! There is no reason why any two students should hand in exactly the same (or even very similar) skill trees, nor should you just duplicate the skill tree shown in the sample output. Print your tree to the console using the `toStringByLevel()` method inherited from `BasicMAryTree280`.

(b) Write a method in the `SkillTree` class called `skillDependencies` which takes a skill name as input and returns an instance of `LinkedList280<Skill>` which contains all the of the skills which are prerequisites for obtaining the input skill (including the input skill itself!). A `RuntimeException` exception should be thrown if the tree does not contain the given skill. A good implementation approach for this method is to use a recursive traversal of the tree to find the named skill, and then add skills to the output list as the recursion unwinds. This week's tutorial will include some discussion of recursive traversal of $m$-ary trees. Add to your `main()` program a few tests of this method, and print out the lists that is returned (you can use the list's `toString()` method for this). Be sure to test the case where the named skill does not exist in the tree.

(c) Write a method in the `SkillTree` class called `skillTotalCost` which takes a skill name as input and returns the total number of skill points that a player must invest to obtain the given skill. If the named skill is not in the skill tree, then the `skillTotalCost` method should throw a `RuntimeException` exception. *Hint: this method is quite easy to implement if it makes use of the* `skillDependencies` *method.*

For example, in the above skill tree, if a character wants the *Shield Ally* skill they would need to spend 1 skill point to get *Shield Proficiency*, and then spend 3 skill points to get *Shield Ally* for an overall investment of $1 + 3 = 4$ points, so for the above tree, `skillTotalCost("Shield Ally")` should return 4. Note that the `Skill` object contains the cost of the skill.

Add to your `main()` program a few tests of `skillTotalCost`, and print out the total costs returned. Be sure to test the case where the named skill does not exist in the tree.

(d) Run your `main()` program. Cut and paste the console output to a text file and submit it with your assignment. See the sample output on the next page.

[1]In the video game world, the term "skill tree" sometimes refers to things that actually aren't trees; a noteworthy example is the skill tree in the ARPG Path Of Exile, which, if you click the link, can see is clearly **not** a tree, even though they call it that. **Here in question 4, we used the term "skill trees" to mean skill trees that are, in fact, actual trees.**

## 3.1 Sample Output

Here is an example of what the output of your program might look like. Remember, you are expected to be creative in designing your skill tree, and your submission should not attempt to duplicate what you see here aside from the general formatting (the formatting can be the same, but the data should be different!). Note that the formatting of output of the skill tree contents is done by the `toStringByLevel()` method of `BasicMAryTree280`.

```
My Skill Tree:

1: Slash, Cost: 1
     2: Mighty Blow, Cost: 2
     2: Shield Bash, Cost: 1
          3: Shield Charge, Cost: 2
          3: Parry, Cost: 2
               4: Shield Wall, Cost: 4
               4: -
               4: -
               4: -
          3: -
          3: -
     2: Cleave, Cost: 2
          3: Whirlwind, Cost: 3
               4: Berzerk, Cost: 5
               4: -
               4: -
               4: -
          3: -
          3: -
          3: -
     2: Mobility, Cost: 1

Dependencies for Shield Wall:
Slash, Cost: 1, Shield Bash, Cost: 1, Parry, Cost: 2, Shield Wall, Cost: 4,
Dependencies for Mobility:
Slash, Cost: 1, Mobility, Cost: 1,
Dependencies for Slash:
Slash, Cost: 1,
Dependencies for FakeSkill:
FakeSkill not found.
To get Whirlwind you must invest 6 points.
To get Mighty Blow you must invest 3 points.
To get Slash you must invest 1 points.
FakeSkill not found.
```

## 4 Files Provided

**lib280-asn3:** A copy of lib280 which includes the `ArrayedBinaryTree280<I>` class needed for Question 3, and the `BasicMAryTree280` class which is needed for Question 4.

**Skill.java** : A complete implementation of the `Skill` class needed for Question 4.

**SkillTree.java** : A template for your implementation of the `SkillTree` class in Question 4.

## 5 What to Hand In

You must submit the following files:

**Q1-2.doc/docx/rtf/pdf/txt** - your answers to questions 1 and 2. Digital images of handwritten pages are also acceptable, provided that they are **clearly** legible.

**ArrayedHeap280.java** - Your implementation of an arrayed heap.

**SkillTree.java** - Your finished implementation of the skill tree.