ACS-2947-002

Assignment 3

Due by Friday, March 25, 11:59 pm

Instructions

- Submit your `.java` files (together in a `Assign3.zip` file) via Nexus.
- Include your name and student number as a comment in every file.
    - Document the classes using Javadoc notation.
    - Include comments as needed.
    - Use appropriate exception handling where necessary.

PART A (40 marks)

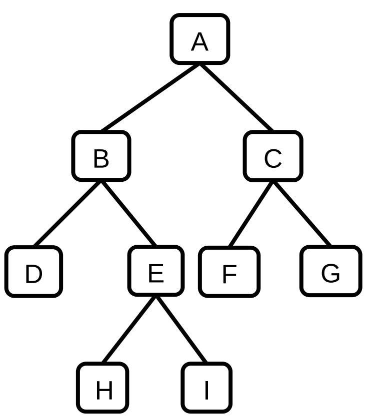
1. Using your Lab 6 `LinkedBinaryTree` implementation, add a position and element iterator.
    a. Have the `Tree` interface extend `Iterable`. Add the abstract methods `iterator()` and `positions()` that return `Iterator<E>` and `Iterable<Position<E>>`, respectively.

    b. Add the nested class and methods from your notes/text.
        - `ElementIterator`, `iterator()`, `positions()` and methods required for your tree traversals.

        * Note that your classes require the `java.util.Iterator` package

    c. Override the toString method to display a tree in the following format:

Structure:



Output:

```
- A
    - B
        - D
        - E
            - H
            - I
    - C
        - F
        - G
```

2. Create an interactive program that asks the user to work through a Yes/No decision tree with height of at least 3. **You must come up with a decision tree of your own.** Name your driver class: `PartA_Driver`. Also display the content of the tree using the toString method.

Example of decision tree from your notes (Textbook figure 8.5 on p. 317 or L10 slide 32)

Sample output:

```
Tree
-----
...  (As described in  1c)


Are you nervous? (yes/no)
no
Will you need to access most of the money within the next 5 years?
(yes/no)
no
Are you willing to accept risks in exchange for higher expected returns?
(yes/no)
yes
Final Decision: Stock portfolio
```

3. In the same driver class, create a binary tree for the arithmetic expression from the notes (L10_) and evaluate the value arithmetic expression tree. Your implementation must:

    a. Display the the postorder representation of the tree

    b. Use a **stack** to evaluate the value of the tree based on the postorder representation. *You may assume that the tree has only integer numbers.*

Sample output:

```
Postorder: [3, 1, +, 3, *, 9, 5, -, 2, +, /, 3, 7, 4, -, *, 6, +, -]
Tree value: -13
```

Notes:

- Iterator implementation:

a. In your tree interface:
   - Extend Iterable and add 2 abstract methods: `iterator()` that returns `Iterator<E>` and `positions()` that returns `Iterable<Position<E>>`

b. In the AbstractTree class:
   - add the nested ElementIterator class from your notes. Implement the iterator() method by returning a new instance of ElementIterator

c. Add the code for 3 traversal algorithms:
   - `preorder()` and its associated recursive private method
   - `postorder()` and its associated recursive private method
   - `breadthfirst()`

- import List and ArrayList from Java Class Libraries.
- For Queue you can use one from `java.util`, or add one of our implementations from class in your package e.g. ArrayQueue from Assignment 1

d. Implement the `positions()` method by returning an `Iterable<Position<E>>` from one of the above. Select your preferred default, and override `toString()` to return a simple list view of your tree in the default traversal order.

- toString:
  - "indentation" of each item depends on its position's depth in the tree
  - your algorithm should work with any tree
  - test this with your tree from Lab 6

- decision tree:
  - Build the tree by assigning/re-assigning positions as you go along.
  - Map your yes/no to left/right child, and work through the decisions until an answer is reached (external node)
  - Your code should work for any linked binary decision tree: starts at the root as the first question and advances to left/right depending on the user input i.e. do not hardcode questions/answers to work only with your tree

PART B (50 marks)

Implement the Priority Queue ADT using a heap. The heap will use your ***LinkedBinaryTree*** (LBT) from PART A and a comparator.

1. Create a class called `LinkedHeapPriorityQueue` (LHPQ) that implements the given `PriorityQueue` and `Entry` interfaces. Include the `AbstractPriorityQueue` class from your textbook for your Linked Priority Queue to extend.

2. Create a driver program to show you working with your priority queue in a simple simulation of a ***real-world*** example (e.g. airline standby list). **You must include a custom class and a custom comparator.**

Notes:
- Before starting Part B, make sure that your `LinkedBinaryTree` is fully implemented. You should have your tree traversal algorithms set and `toString()` in place.

- First, have a good understanding of the array-based `HeapPriorityQueue` from your notes/textbook. Here, the parameters are indices that represent the level number of each entry. With a linked tree-based PQ the parameters will be `Position` objects. Instead of using indices to access entries in the tree, we will determine the positions of these elements relatively.

- Start building your LHPQ. Declare a `LinkedBinaryTree` called `heap` that holds `Entry` objects as its elements. Add the constructors in the same manner as your textbook `HeapPriorityQueue` (HPQ), and make sure that a `DefaultComparator` is included in your package. The next 5 protected utilities of HPQ are not required in the LinkedHeap version because all of this information can be either directly accessed or quickly determined via the LBT methods.

- Next, look at the protected `swap` utility: instead of indices (int), you will have `Position` objects as parameters. In an ArrayList, you swap the *elements* in the given array indices. How would you swap the *elements* in given positions? Use this to form a basis of how to convert from array-based to LBT-based.

Suggestions/Notes:

- Override the `toString()` method to help with debugging
    - Should be quick if the `toString()` in your LBT is in place
    - Using the breadth-first traversal algorithm can be handy with PQs.

- The first method that you need to get running is `insert` (which needs to have `upheap` and `size` in place): use simple sample data for your driver as you are building/testing e.g., use K-V pairs: 8-8, 6-6, 7-7, 5-5, 3-3, 0-0, 9-9.
    - This way you will insert entries that may or may not need upheaping, and values outputted are easier to understand and map
    - Jot down what the heap should look like and compare when debugging

- You will need to find a way to insert the next node to satisfy the complete binary tree property: think of how a *binary* tree works:
    - How can we insert a new entry in the next position? i.e. how do we find the parent to add this new position to, and whether we add to left or right?
    - **Requirement: use a Stack in your solution**

- Once your `LinkedHeapPriorityQueue` is in place and working accurately then you can start working on your PQ simulation.

Submission

Submit your **`Assign3.zip`** file that includes all the assignment files (`Tree.java, BinaryTree.java, Position.java, AbstractTree.java, AbstractBinaryTree.java, LinkedBinaryTree.java, DefaultComparator.java, PartA_Driver.java, PriorityQueue.java, Entry.java, AbstractPriorityQueue.java, LinkedHeapPriorityQueue.java, PartB_Driver.java`, and your custom classes) via **Nexus**.