Homework 2: Tree Structures

CS 1332 Section C

Fall 2021

1 Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines, you risk receiving a **0** for the entire assignment

1. All submitted code must compile under **JDK 11**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, instance/global variables, or method signatures. For example, do not add **throws** to the method headers since they are not necessary.
4. Do not add additional public methods. You may create helper methods, but any helper method you create should be **private**.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use java.util.ArrayList for an ArrayList assignment. Ask if you are unsure.)
6. Always consider the efficiency of your code. Even if your method is *O(n)*, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is very rare).
7. You must submit your source code - the .java files. Do not submit compiled code - the .class files.
8. Only the last submission will be graded. Make sure your last submission has all required files. Resubmitting voids prior submissions.

2 Style and Formatting

It is important that your code is not only functional, but written clearly and with good programming style. Your code will be checked against a style checker. The style checker is provided to you and it located on Canvas. A point is deducted for every style error that occurs. Please double check before you submit that your code is in the appropriate style so that you don't lose any unnecessary points!

2.1 Javadocs

Javadocs should be written for any private helper methods that you create. They should follow a style similar to the existing javadocs on the assignment. Any javadocs you write must be useful and describe the contract, parameters, and return value of the method. Random or useless javadocs added only to appease checkstyle will lose points.

2.2 Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies to all aspects of your code, such as comments, variable names, and javadocs.

2.3 Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong.** "Error", "Oof - Bad things are happening", and "FAIL" are *not* good messages. Additionally, the name of an exception itself is not a good message.

In addition, you may not use try catch blocks to catch an excpetion unless you are catching an exception you have explicitly thrown yourself with the **throw new ExceptionName('Exception Message");** syntax

2.4 Generics

If available, use the generic type of the class; do not use the raw type of the class. For example, use new **LinkedList<Integer>()** instead of **new LinkedList()**. Using the raw type of the class will result in a penalty.

3 Forbidden Statements

You may not use any of the following in your code at any time in CS 1332. If you are not sure whether you can use something, and it is not explicitly listed here, just ask. Debug print statements are fine, but should be either removed or commented out prior to submission. If print statements are left in, assignments are messy to grade, and checkstyle points will be deducted.

- package
- System.arraycopy()
- clone()
- assert()
- Arrays class
- Thread class
- Collections class
- Collection.toArray()
- Refleciton APIs
- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

4 JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code, nor do they guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub.

5 Deliverables

You must submit all of the following file(s) to the corresponding assignment on Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission is graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present.

1. BST.java
2. MaxHeap.java

5.1 BST

You are to code a **BST**, which is a collection of nodes, each having a data item and reference pointing to left and right child nodes. The BST must follow the order property: for any given node, its left child's data and all children of the left child must be less than the current node while its right child's data and all children of the right node must be greater than the current node. In order to compare the data, all elements added to the tree must implement Java's generic **Comparable** interface.

The **BST** will have two constructors: a no-argument constructor (which should initialize an empty tree), and a constructor which takes in a collection of data to be added to the tree, and initializes the tree with this collection of data. You may import Java's **LinkedList/ArrayList** classes for the 4 traversal methods, but only for use in these methods.

5.1.1 Recursion

Since trees are naturally recursive structures, all methods that are not O(1) **must be implemented recursively**, except for level order traversal. You'll also notice that a lot of the public method stubs we've provided do not contain the parameters necessary for recursion to work, so these public methods act as "wrapper methods" for the user to use. You will have to write private recursive helper methods and call them in these wrapper methods. **All of these helper methods must be private**. To reiterate, do not change the method headers for the provided methods

For methods that change the structure of the tree in some way, you should use a technique taught in class called pointer reinforcement. More on this will be taught during recitation. The basic concept is to "unset" pointers as you recurse down the tree and to reset them as you backtrack on the recursive stack.

5.1.2 Nodes

The binary search tree consists of nodes. A class BSTNode is provided to you. BSTNode has getter and setter methods to access and mutate the structure of the nodes.

5.1.3 Methods

You will implement all standard methods for a Java data structure (add, remove, etc.) in addition to a few other methods (such as traversals). You must follow the requirements stated in the javadocs of each method you implement. Again, when performing these methods (especially remove), use pointer reinforcement.

5.1.4 Traversals

You will implement 4 different ways of traversing a tree: pre-order traversal, in-order traversal, postorder traversal, and level-order traversal. The first 3 MUST be implemented recursively; level-order is best implemented iteratively. For a level-order traversal, you may use Java's Queue interface (and an implementing class for it such as LinkedList).

5.1.5 Height

You will implement a method to calculate the height of the tree. The height of any given node is max(left child node's height, right child node's height) + 1. When doing this calculation, a leaf node has a height of 0 and a null node has a height of -1.

5.1.6 Comparable

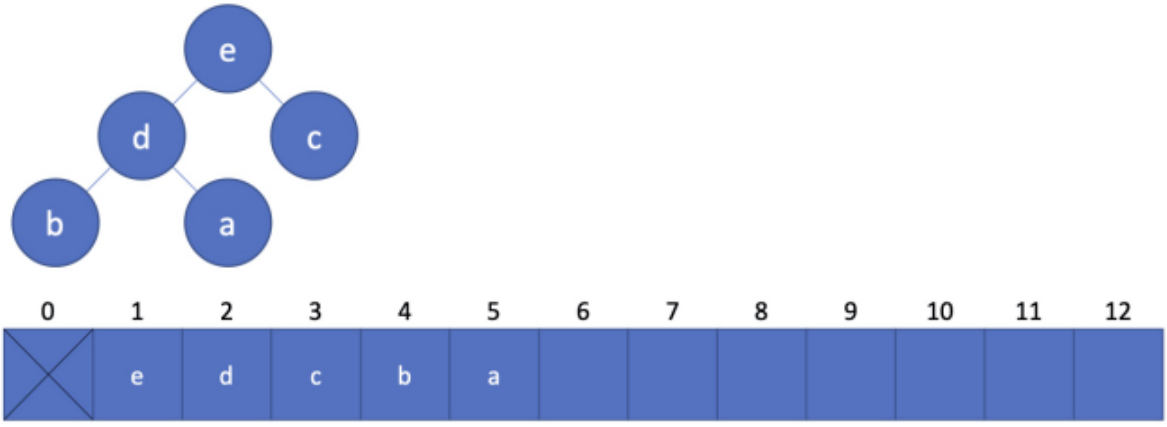
As stated, the data in the BST must implement the Comparable interface. As you'll see in the files, the generic typing has been specified to require that it implements the Comparable interface. You use the interface by making a method call like data1.compareTo(data2). This will return an int, and the value tells you how data1 and data2 are in relation to each other

- If the int is positive, then data1 is larger than data2.
- If the int is negative, then data1 is smaller than data2.
- If the int is zero, then data1 equals data2.

Note that the returned value can be any integer in Java's int range, not just -1, 0, 1.

5.2 Heap

You are to code a Heap, specifically a **MaxHeap**, that is backed be an array of contiguous elements. Here is a tree and array representation of the same MaxHeap:



5.2.1 Shape Property

The tree must be **complete**. No levels of the tree can contain null except the bottom-most level. The bottom-most level must have data filled from left to right. In other words, the backing array should always have contiguous data starting at index 1. **Leave index 0 set to null always**.

5.2.2 Order Property

Each node's data is larger than the data in its two children. There is no explicit relationship between sibling nodes and their corresponding subtrees.

These properties guarantee that the largest element in the heap will be at the root of the heap.

5.2.3 Backing Array

Although heaps are usually classified as a type of tree, they are commonly implemented using an array due to their completeness. In your implementation, you should leave index 0 empty and begin your heap at index 1. This will make the arithmetic to find parents and children simpler.

To find the children of index i, $2i$ is the index of the left child (if one exists) and $2i + 1$ is the index of the right child (if one exists). Make sure to be careful that you do not run into index out of bounds exceptions when attempting to access children. By extension, the parent of a given node will exist at index $i/2$.

5.2.4 Constructors

You should implement two constructors for this heap. One constructor initializes an empty heap with the capacity specified by a constant in MaxHeap.java. The other constructor should implement the BuildHeap algorithm that was taught in class, which is an algorithm that creates a heap in $O(n)$ time. Simply adding the elements one by one will not receive credit since it would be $O(nlog(n))$ in the worst case; see the javadocs for this constructor for more specifications.

5.2.5 General Notes

You may assume that your implementation does not need to handle duplicate elements. That is, the add method will never be passed duplicates and the remove method will never have to deal with the heap having duplicates. To be clear, your implementation would most likely work even if we were to test for duplicates; however, this will help remove ambiguity surrounding grading and testing your implementation.

Additionally, you will not be required to use recursion for this part of the assignment. Use whatever you find most intuitive - recursion, iteration, or both. However, regardless of the technique you use, make sure to meet efficiency requirements as discussed in lecture and recitation.

6 Grading

Here is the grading breakdown for the assignment. There are various deductions not listed that are incurred when breaking the rules listed in the PDF and in other various circumstances.

- BST − 45pts
- MaxHeap − 45pts
- Checkstyle − 10 pts
- Total: 100 pts