**CSCI 3302 Programming Assignment 06 (100 Points)**

**Due: Nov 18, 8:00 AM**

**GITHUB Link:** [**Program 06**](https://classroom.github.com/a/X4LK4BDY)

**Objectives:**

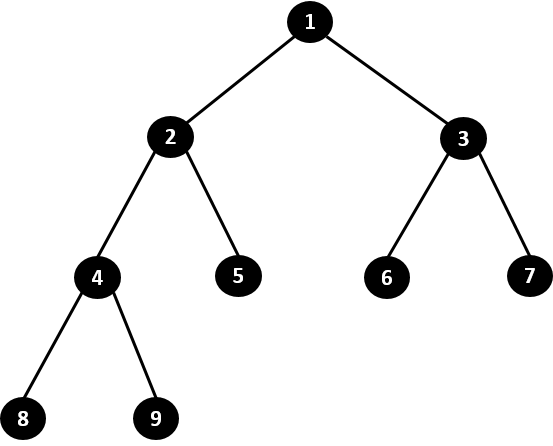
* Demonstrate how to implement operations on a reference-based tree data structure.
* Demonstrate an understanding of a complete binary tree.
* Demonstrate understanding of traversing a binary tree.
* Demonstrate Java programming proficiency for generics.

**Assignment Assistance:**

* This homework assignment is due prior to the date and time specified above.
* This assignment is restricted to individual effort. As per our syllabus, the use of AI is prohibited. You may not receive help from any other person except the instructor or the AARC (help from the AARC must be well documented!).
* Any resource used (other than Dr. Becnel or the course text) must be documented in the code (as comments) detailing the source and describing exactly what was learned and how that information was used. Submissions will be severely penalized if copied in part or in whole from any source.
* If you need help, visit your instructor during his posted office hours. If your schedule cannot accommodate any of these times, then email your instructor to schedule a different time.

**Problem Description:**

1. As discussed in class, a complete binary tree of height *h* is a binary tree that is full down to level *h* – 1, with level *h* filled in from left to right, as the figure below illustrates:



**Figure 1: A Complete Binary Tree with 9 nodes**

1. More formally, a binary tree *T* of height *h* is complete if:
   1. all nodes at level *h* – 2 and above have two children each;
   2. when a node at level *h* – 1 has children, all nodes to its left at the same level have two children each; and
   3. when a node at level *h* – 1 has one child, it is a left child.
2. In this assignment, you will implement the methods identified in the stub code below in support of a complete binary tree (CBT). This code is in your GitHub repo when you start the assignment.

// Student Name

// Date

// CSCI 3302 Section 001

//

// Files:

//

// Description:

public class **CBT**<T> {

  private T item; // the item stored at the root of this tree

  private int size; // the number of nodes in the subtree

  private CBT<T> leftChild; // reference to the left subtree

  private CBT<T> rightChild; // reference to the right subtree

  //===============Constructors=================

  /\*\*

   \* This method creates an empty CBT by setting the size to 0

   \* and leaving all fields null.

   \*/

  public CBT() {

**this**.size = 0;

   }

  /\*\*

   \* This constructor adds the new item, sets both children to

   \* empty trees (new CBT) and the size (of the subtree

   \* rooted at this node) to 1.

   \* @param newItem - item stored at root of tree

   \* A reference to the tree is returned.

   \*/

  public CBT(T newItem) {

    // your code here

  } // end constructor

  //==================Methods=====================

  /\*\*

   \* This method returns the number of items within the subtree

   \* rooted at this node. This method should execute in constant

   \* time; i.e. O(1).

   \* @return - returns an integer representing the number of elements in the tree

   \*/

  public int size() {

    // your code here

  } // end size

   /\*\*

    \* This method returns the height of the binary tree. The method

    \* should run in O(log n) time. So, you canNOT use the recursive

    \* procedure we did in class. Hint: Consider the relationship

    \* between the height and the size (powers of 2).

    \* @return

    \*/

   public int height() {

     int h = 0;     // height to return

     int s = **this**.size; // local variable for size of tree

     // you code here

   }

  /\*\*

   \* This method is a helper function used by the add method that

   \* determines, at a given CBT node, whether the next traversal

   \* should occur on the left subtree or not.

   \* @return - true if we should traverse left and false otherwise

   \*

   \*/

  private boolean goLeft() {

    // your code here

  } // end goLeft

  /\*\*

   \* This method creates a new CBT node containing the newItem and

   \* adds it to the complete tree as a leaf in the next available

   \* location.

   \* This method should execute in O(log n) where n is

   \* the number of elements in the tree

   \*

   \* @param - given the item to add to the tree

   \*

   \*/

  public void add(T newItem) {

    // your code here

  } // end add

  /\*\*

   \* This method determines whether or not checkItem is in the tree.

   \* This method should execute in O(n) where n is the number of

   \* elements in the tree.

   \* @param checkItem - reference to item to determine if in the tree

   \* @return true if given item is in the tree, false otherwise

   \*/

  public boolean contains(T checkItem) {

    // your code here

  } // end contains

  /\*\*

   \* Here is a toString method that may be useful for testing. You

   \* are not responsible for this code.

   \* @return - string representation of the tree.

   \*/

   public String toString() {

      StringBuilder buffer = new StringBuilder(50);

      if (!**this**.isEmpty())

          buildTreeString(buffer, "", "");

      return buffer.toString();

  }

  private void buildTreeString(StringBuilder buffer, String prefix, String childrenPrefix) {

      buffer.append(prefix);

      buffer.append(**this**.item.toString());

      buffer.append('\n');

      if (!**this**.leftChild.isEmpty())

          if (!**this**.rightChild.isEmpty())

              leftChild.buildTreeString(buffer, childrenPrefix + "├── ", childrenPrefix + "│   ");

          else

              leftChild.buildTreeString(buffer, childrenPrefix + "└── ", childrenPrefix + "    ");

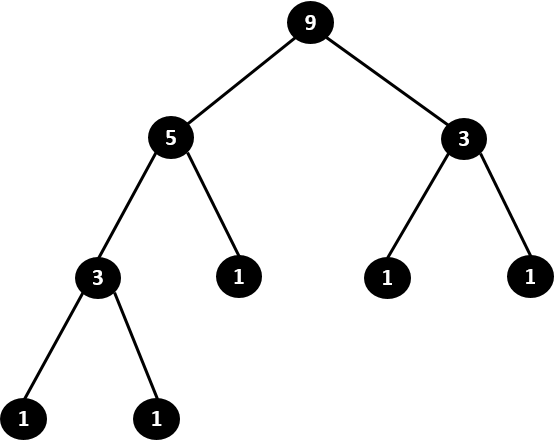
      if (!**this**.rightChild.isEmpty())

          rightChild.buildTreeString(buffer, childrenPrefix + "└── ", childrenPrefix + "    ");

  }

} // end CBT

1. The methods that you are required to implement should, for the most part, be self-explanatory from examining the provided code above. The comments provide some guidance. Additional guidance for each part follows:
   1. The size field corresponds to the size of the tree rooted at this node. For example, the labels for nodes in the following figure represent the values of the size fields for each node:



**Figure 2: Each node is labeled with its appropriate size field.**

* 1. The height method should return the height of the tree. It should run in O(log n) time. Use the relationship between the height of the tree and its size. In the example for Figure 2, the height is 4.
  2. The contains method should test whether the current node contains the item or if either subtree contains the item.
  3. The add method adds a new node into the next available leaf position that will allow the binary tree to maintain its completeness property (in accordance with the definition in 2, above).
     + Thus, in Figure 1, the label for each node corresponds to the order that the nodes were added. For example, Node 7 was the seventh node added to the tree.
     + The add method relies on the goLeft method. The add method traverses down the tree until it reaches the appropriate parent node for adding the new node. In Figure 1, the method should traverse down to Node 5 and then add the new node as the left child of Node 5.
     + As the add method traverses the tree, the goLeft method should be called to determine whether the next visited node should occur on the left subtree or the right subtree.
  4. The goLeft method works by calculating, based on the size of the subtree rooted at the current node, whether any remaining spots exist in the left subtree.
     + To understand how to implement the goLeft method, try to understand the mathematical relationship between the value of the size attribute and the structure of the subtree. How many elements would be in the next **full binary tree**? How many leaf node slots are there and how many are currently filled?
     + This is done by determining how many leaf slots exist in the subtree at the bottom level (hint: always a power of two) and how many of these slots are occupied.
     + If the left subtree is not full (less than half of the total leaf node slots are filled), then true is returned. Otherwise, false is returned.
     + Don’t forget to consider the case that the current CBT is full (i.e., all leaf node slots are filled). If this is the case, then the next node should be added to the leftmost slot in the left subtree; that is, goLeft should return true.
     + For example, in Figure 1, if the add method was currently examining the root (Node 1), it could determine, based on size = 9, that there are 8 leaf node slots along the bottom level, and that there are currently 2 leaf nodes occupied. Since less than half of the 8 total slots are occupied, then true would be returned by goLeft.
  5. The toString method is implemented for you for testing purposes. You should also consider adding an in-order traversal of the tree to assist in testing.

1. Your program should work in the GitHub codespace (Linux environment) and locally (Windows environment).
2. You may write any private helper methods, if needed.

**Hints:**

* Your data structure should use generics. Notice that item is of type T. Thus, your implementation should not have anything specified as type Object. When done correctly, your code should compile with no warnings (nor errors).
* Notice that, unlike the other reference-based data structures that we have examined (both Program 04 and Program 05), this data structure does not make use of a separate Node class. A CBT is guaranteed to contain at least one node storing an item.
* The add and contains methods can be written either iteratively or recursively. Feel free to use either approach.
* No tests are provided for you. Use what you learned this semester to test your program. You are welcome to use JUnit testing (like in previous assignments) or a simple test program that makes use of a main method as we have done in some paired programming assignments and in class. Include your test file(s) in your submission.

**Submission:**

* Review the Evaluation below to ensure you have met all the requirements.
* Submit an electronic copy of CBT.java to GitHub along with the program(s) you use for testing. Upload a backup copy to D2L. This file CBT.java should NOT contain a main method or any extraneous testing code. You should include test files in your repository; however, these will not be considered when grading. If you wish to include non-working code for insight into your thought process, make sure to contain it within comment blocks and ensure that the submission successfully compiles.

**Evaluation**

|  |  |
| --- | --- |
| * **Automatic Deductions:** |  |
| Late/Not Submitted | -100 |
| Code not submitted to GitHub | -30 |
| Code does not run/compile | -50 |
|  |  |
| **Earn Points for the following:** |  |
| Code has comment header with name, section, date | 5 pts |
| Code organization, structure, and indention is appropriate (SHFT + ALT + F in VS Code) | 5 pts |
| Code is well and meaningfully commented. | 5 pts |
| Appropriate variable and method names that follow Java conventions | 5 pts |
| Instructions correctly followed for fields, class, methods | 10 pts |
| size and constructor | 5 pts |
| add, contains, goLeft, height with correct big O | 15 pts each |
| Testing program or unit test included and test all methods | 5 |