UNC-CH COMP 410

Binary Search Tree Implementation

Basic Binary Search Tree (BST)

You will write code that implements and exercises **binary search trees**. We are eventually going to build **splay trees** but will get there one chunk at a time, and the first chunk is the basic BST.

There are plenty of implementations of "tree" stuff out there. Please do not copy those. You may read them for ideas and understanding, but I want you to write your own code and follow the structure I outline below.

The learning in this activity comes from actually powering through all the reasoning needed to get the pointers (object addresses) and recursive calls hooked up correctly.

Delegation

In this assignment we are learning about a common program design pattern called *delegation*. Delegation is common in many object designs. In a delegation situation, one object contains the methods that are called "publicly", and that object may pass on the calls to a helper object (or several) after examining the circumstances of the method call. Often the helper objects will identical (or very similar) methods to those of the main object. For example, we call method "doSomething()" on the main object, and in order to complete that call the main object will call "doSomething()" on an appropriate helper object.

In our program, we will have one object as "the BST". We will call the various functions such as insert, delete, etc. on that object. The BST object will then examine the circumstances around the call and in most cases call a similar methods the cell that is the root of the data hierarchy... an object that is referenced by one of the fields in the BST object. The BST object will directly handle the calls for special cases, for initial cases or clean up cases. For "normal" cases, it will pass along the call -- delegate the work -- to the data cells.

The BST implementation examples in your text are not structured this way directly. We are requiring this structure so you learn some BST and you learn some patterns.

What to Hand-in

Hand in your code as per the instructions under Assignment 2 in Sakai. There will be zipping and naming instructions there.

Interfaces

For uniformity, and to aid timely grading, you must write your program with these specific structures.

Interface BST_Interface

```
/**
 * COMP 410
 *
 * Make your class and its methods public!
 *
 * Your BST implementation will implement this interface.
 *
*/

package BST_A2;

/*
  Interface: The BST will provide this collection of operations:

  insert:
    in: a string (the element to be stored into the tree)
    return: boolean, return true if insert is successful, false otherwise
    effect: if the string is already in the tree, then there is no change to
              the tree state, and return false
            if the string is not already in the tree, then a new tree cell/node
              is created, the string put into it as data, the new node is linked
              into the tree at the proper place; size is incremented by 1,
              and return a true
  remove:
    in: a string (the element to be taken out of the tree)
    return: boolean, return true if the remove is successful, false otherwise
            this means return false if the tree size is 0
    effect: if the element being looked for is in the tree, unlink the node containing
              it and return true (success); size decreases by one
            if the element being looked for is not in the tree, return false and
              make no change to the tree state
  contains:
    in: a string (the element to be seaarched for)
    return: boolean, return true if the string being looked for is in the tree;
            return false otherwise
            this means return false if tree size is 0
    effect: no change to the state of the tree

  findMin:
    in: none
    return: string, the element value from the tree that is smallest
    effect: no tree state change
    error: is tree size is 0, return null


  findMax:
    in: none
    return: string, the element value from the tree that is largest
    effect: no tree state change
    error: is tree size is 0, return null

  size:
    in: nothing
    return: number of elements stored in the tree
    effect: no change to tree state

  empty:
    in: nothing
    return: boolean, true if the tree has no elements in it, true if size is 0
            return false otherwise
    effect: no change to tree state

  height:
    in: none
    return: integer, the length of the longest path in the tree from root to a leaf
    effect: no change in tree state
    error: return -1 is tree is empty (size is 0, root is null)

  getRoot:
    in: none
    return: a tree cell/node, the one that is the root of the entire tree
            means return a null if the tree is empty
    effect: no change to tree state

*/

// ADT operations

public interface BST_Interface {
  public boolean insert(String s);
  public boolean remove(String s);
  public String findMin();
  public String findMax();
  public boolean empty();
  public boolean contains(String s);
  public int size();
  public int height();
  public BST_Node getRoot();
}
```

Class BST_Node

```
package BST_A2;

public class BST_Node {
  String data;
  BST_Node left;
  BST_Node right;

  BST_Node(String data){ this.data=data; }

  // --- used for testing  ----------------------------------------------
  //
  // leave these 3 methods in, as is

  public String getData(){ return data; }
  public BST_Node getLeft(){ return left; }
  public BST_Node getRight(){ return right; }

  // --- end used for testing -------------------------------------------
  // --- fill in these methods ------------------------------------------
  //
  // at the moment, they are stubs returning false
  // or some appropriate "fake" value
  //
  // you make them work properly
  // add the meat of correct implementation logic to them

  // you MAY change the signatures if you wish...
  // make the take more or different parameters
  // have them return different types
  //
  // you may use recursive or iterative implementations

  /*
  public boolean containsNode(String s){ return false; }
  public boolean insertNode(String s){ return false; }
  public boolean removeNode(String s){ return false; }
  public BST_Node findMin(){ return left; }
  public BST_Node findMax(){ return right; }
  public int getHeight(){ return 0; }
  */

  // --- end fill in these methods --------------------------------------


  // --------------------------------------------------------------------
  // you may add any other methods you want to get the job done
  // --------------------------------------------------------------------

  public String toString(){
    return "Data: "+this.data+", Left: "+((this.left!=null)?left.data:"null")
            +",Right: "+((this.right!=null)?right.data:"null");
  }
}
```

Class BST

```
package BST_A2;

public class BST implements BST_Interface {
  public BST_Node root;
  int size;

  public BST(){ size=0; root=null; }

  @Override
  //used for testing, please leave as is
  public BST_Node getRoot(){ return root; }

}
```

Class BST_Playground

```
package BST_A2;

public class BST_Playground {
/*
 * you will test your own BST implementation in here
 *
 * we will replace this with our own when grading, and will
 * do what you should do in here... create BST objects,
 * put data into them, take data out, look for values stored
 * in it, checking size and height, and looking at the BST_Nodes
 * to see if they are all linked up correctly for a BST
 *
*/

  public static void main(String[]args){

   // you should test your BST implementation in here
   // it is up to you to test it thoroughly and make sure
   // the methods behave as requested above in the interface

   // do not simple depend on the oracle test we will give
   // use the oracle tests as a way of checking AFTER you have done
   // your own testing

   // one thing you might find useful for debugging is a print tree method
   // feel free to use the printLevelOrder method to verify your trees manually
   // or write one you like better
   // you may wish to print not only the node value, and indicators of what
   // nodes are the left and right subtree roots,
   // but also which node is the parent of the current node

  }

  static void printLevelOrder(BST tree){
  //will print your current tree in Level-Order...
  //https://en.wikipedia.org/wiki/Tree_traversal
    int h=tree.getRoot().getHeight();
    for(int i=0;i<=h;i++){
      printGivenLevel(tree.getRoot(), i);
    }

  }
  static void printGivenLevel(BST_Node root,int level){
    if(root==null)return;
    if(level==0)System.out.print(root.data+" ");
    else if(level>0){
      printGivenLevel(root.left,level-1);
      printGivenLevel(root.right,level-1);
    }
  }
  static void printInOrder(BST_Node root){
  //will print your current tree In-Order
  if(root!=null){
    printInOrder(root.getLeft());
    System.out.print(root.getData() + " ");
    printInOrder(root.getRight());
  }
  }
}
```

Class RunTests

```
package BST_A2;
import gradingTools.comp410s19.assignment2.testcases.Assignment2Suite;

public class RunTests {
  public static void main(String[] args){ //runs Assignment 2 oracle tests
    Assignment2Suite.main(args);
  }
}
```

Grading Notes

We want your code structured as shown above... it is an object structure called "delegation". When a method in the BST class is called, it will handle special cases and edge cases (like the very first add node when the tree is empty); if no edge case applies, then the methods "delegates" the work by calling a similar methods in the BST_Node class (by calling that method on the root node). The methods in the BST_Node class will then do the hard work of linking up tree cells correctly on insert, delete, etc. You may implement the BST_Node methods recursively or iteratively.

For example, consider this:

```
BST myTree = new BST();

myTree.insert("hello");
myTree.insert("world");
```

The "hello" is the first insert being done. At this point the root field in myTree is null. The insert method in myTree will check for that null root and make the new BST_Node to hold the "hello" value. When the "world" insert is done, the insert method in myTree will note that root is NOT null (since we have one cell, containing "hello") and will then call root.insert("world")... it will *delegate* the insert work to the root node (to the tree linked cells).

Furthermore, please note that just because the Oracle gives you 100, you are not guaranteed any specific grade on the assignment. We do try to put as many things as we can in there edge-case wise, but testing your programs is your responsibility. The main goal of the Oracle is to provide *some direction*. However, it is not the end-all be-all determiner of correctness. Please do not use this tool as a crutch.

TIPS:

- If you want to implement recursively and want a refresher on recursion, implementing height and testing on a manually built tree might be a good idea. You can manually build a tree cell hierarchy like this:

```
BST atree = new BST();
atree.root= new BST_Node("gamma");
atree.root.left = new BST_Node("delta");
atree.root.right = new BST_Node("theta");
atree.root.left.left = new BST_Node("alpha");
atree.root.right.left = new BST_Node("pi");

system.out.println(BST.height()); // this will delegate to the height method
                                  // in BST_Node class

// etc...
```

- Remove is by far the hardest method. Try to get all the other methods done first! Particularly do insert and build a tree, and print it out to make sure you are hooking it all up properly according to the BST properties the tree must maintain.

- Lastly, when comparing strings note that the String.compareTo method compares two strings lexically and should be used in this assignment. Don't waste time building your own compare function!

Supporting Code Examples

Random strings

You may find this code useful. You can build larger trees for testing by generating a lot of (random) string inut rather than doing a ton of input typing. It is a class that generates random strings of characters from "a" to "z". It will generate strings that are 5 to 25 characters in length by default if you call MyRandom.nextString() with no arguments. If you wish to control the string lengths better, then you can call it with arguments. Calling MyRandom.nextString(3,11) for example will generate string that vary in length from 3 to 11 characters. The nextString methods are declared static so you can call them as shown with the class name qualifier and don't need to instantiate an object in your code.

Keyboard input

You may find this code useful. Hoever this code is aging. There are other ways to do input perhaps more easily. Since we don't do manual drivers currently it my not be something you need. In past classes we wrote an interactive test driver to use to manually build a tree one command at a time, examining its structure with tree printing. The sample code shows one way to do keyboard input in Java. Another way is to use the Scanner class, which wraps the various streams shown in the sample code. It is even simpler to use. Scanner includes methods like "next" and "nextLine" to get the next String, or an entire line of text (to the return key), as well as methods like "nextInt" and "nextFloat" to parse various types from the input text directly. Here is a code example.
