UNC-CH COMP 410

Splay Tree ADT (SPLT)

What to Do

- **Step 1:** Obtain a working version of the BST code from Assignment 2. You may do this several ways:

  a. By using your current correct and working code from Assn 2.
  b. Use your own code from Assn 2, but first correct the errors you made and get it working properly.
  c. Get a copy of the working BST code we are distributing. No charge, no penalty for doing this. (Will be a zip attached to sakai assignment prompt)

- **Step 2:** Implement a Splay Tree (SPLT) according to the interfaces and classes given below.
  The basic activity is to augment the working BST code by adding a splay method, and the calling it properly. The splay will be a private (not public) method that is called at appropriate times by some of the other public methods of the BST (see below "when to splay"). The splay method will move the node being splayed to the root of the tree, adjusting along the way in order to make the tree maintain proper BST relationships among its node values.

- **Step 3:** Test your code yourself, as we have been doing. Use the JAR file of oracle tests (when we release it) as a further check on the correctness of your code. Use the animation page for splay trees to give you examples of correctly splayed trees from the various access operations.

The "splay" method
You will most likely want to write a splay method that can be called at the right times inside your existing BST methods (such as insert, find, delete, etc.). Note however that the BST interface really is not changed. We have chosen to name the interface "SPLT_Interface" but it has all the methods of the BST_Interface. This means there will be no public method "splay" that client code can call. Splaying is simply a balancing method that is used internally during execution of the public methods of the tree.

What to Hand-in

Hand in your code as per the instructions under Assignment 4 in Sakai.

Interfaces

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

Interface SPLT_Interface

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

package SPLT_A4;

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

  insert:
    in: a string (the element to be stored into the tree)
    return: void, if you need to update size when delegating, consider using a boolean as I mention in BST_Node
    effect: if the string is already in the tree, then there is no change to
              the tree state (save for splaying), and return
            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
  remove:
    in: a string (the element to be taken out of the tree)
    return: void
    effect: see description on "when to splay"

  contains:
    in: a string (the element to be searched 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 (save for splaying found node or what would be its parent)

  findMin:
    in: none
    return: string, the element value from the tree that is smallest
    effect: no tree state change (save for splaying)
    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 (save for splaying)
    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 SPLT_Interface {
  public void insert(String s);
  public void 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 SPLT_A4;

public class BST_Node {
  String data;
  BST_Node left;
  BST_Node right;
  BST_Node par; //parent...not necessarily required, but can be useful in splay tree
  boolean justMade; //could be helpful if you change some of the return types on your BST_Node insert.
            //I personally use it to indicate to my SPLT insert whether or not we increment size.

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

  BST_Node(String data, BST_Node left,BST_Node right,BST_Node par){ //feel free to modify this constructor to suit your needs
    this.data=data;
    this.left=left;
    this.right=right;
    this.par=par;
    this.justMade=true;
  }

  // --- used for testing  ----------------------------------------------
  //
  // leave these 3 methods in, as is (meaning also make sure they do in fact return data,left,right respectively)

  public String getData(){ return data; }
  public BST_Node getLeft(){ return left; }
  public BST_Node getRight(){ return right; }
  // --- end used for testing -------------------------------------------


  // --- Some example methods that could be helpful ------------------------------------------
  //
  // add the meat of correct implementation logic to them if you wish

  // you MAY change the signatures if you wish...names too (we will not grade on delegation for this assignment)
  // make them take more or different parameters
  // have them return different types
  //
  // you may use recursive or iterative implementations

  /*
  public BST_Node containsNode(String s){ return false; } //note: I personally find it easiest to make this return a Node,(that being the node splayed to root), you are however free to do what you wish.
  public BST_Node insertNode(String s){ return false; } //Really same logic as above note
  public boolean removeNode(String s){ return false; } //I personal do not use the removeNode internal method in my impl since it is rather easily done in SPLT, feel free to try to delegate this out, however we do not "remove" like we do in B
  public BST_Node findMin(){ return left; }
  public BST_Node findMax(){ return right; }
  public int getHeight(){ return 0; }

  private void splay(BST_Node toSplay) { return false; } //you could have this return or take in whatever you want..so long as it will do the job internally. As a caller of SPLT functions, I should really have no idea if you are "splaying or
                      //I of course, will be checking with tests and by eye to make sure you are indeed splaying
                      //Pro tip: Making individual methods for rotateLeft and rotateRight might be a good idea!
  */

  // --- end example methods --------------------------------------



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


}
```

Class SPLT

```
package SPLT_A4;

public class SPLT implements SPLT_Interface{
  private BST_Node root;
  private int size;

  public SPLT() {
    this.size = 0;
  }

  public BST_Node getRoot() { //please keep this in here! I need your root node to test your tree!
    return root;
  }

}
```

Class SPLT_Playground

```
package SPLT_A4;

public class SPLT_Playground {
  public static void main(String[] args){
    genTest();
  }

  public static void genTest(){
    SPLT tree= new SPLT();
    tree.insert("hello");
    tree.insert("world");
    tree.insert("my");
    tree.insert("name");
    tree.insert("is");
    tree.insert("blank");
    tree.remove("hello");
    System.out.println("size is "+tree.size());

    printLevelOrder(tree);
  }

    static void printLevelOrder(SPLT tree){
    //will print your current tree in Level-Order...Requires a correct height method
    //https://en.wikipedia.org/wiki/Tree_traversal
      int h=tree.getRoot().getHeight();
      for(int i=0;i<=h;i++){
        System.out.print("Level "+i+":");
        printGivenLevel(tree.getRoot(), i);
        System.out.println();
      }

    }
    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){
      if(root!=null){
      printInOrder(root.getLeft());
      System.out.print(root.getData()+" ");
      printInOrder(root.getRight());
      }
  }

}
```

Class RunTests

```
package SPLT_A4;
import gradingTools.comp410s19.assignment4.testcases.Assignment4Suite;

public class RunTests {
  public static void main(String[] args){ //runs Assignment 4 oracle tests
    Assignment4Suite.main(args);
  }
}
```

When to Splay

Splaying is a balancing operations that is done when nodes are accessed in the tree. The basic splay operation causes a node to move to the root position by doing rotations that preserve the BST relations amoung the node values. We move a node to the root position on the assumption that more accesses to that node are likely in the near future; that node will be near the root when those subsequent accesses happen.

First, write an internal method "splay" for your SPLT. It may take an argument that is a tree cell (the node to be moved to the root). When done, the tree will have that cell as its new root, and the other cells will all still be in proper BST arrangement. It may be useful to separate out a couple of methods for rotateLeft and rotateRight operations.

One you have the splay method, you call it at these points:

- **insert**: do a normal BST insert, and then splay the inserted node. If the node being inserted is already in the tree, then it is not added again, but it is splayed to the root. Either way, when done the root node has the inserted value in it.
- **Pro Tip:** You will notice that in the BST_Node example methods, we have insertNode return a BST_Node. While you do not have to do this, think of the potential advantage of returning a Node. If you return the newly splayed root Node to SPLT, you can now set that to Root! You can then utilize some sort of boolean check to see if this new root was recently created, or simply was a dup existing before-hand to decide whether or not you increment size. Hint: see the constructor for BST_Node.

- **contains**: search for the node with the indicated value; the last node accessed in this search is splayed. This means if the search is successful, then the found node is splayed and becomes the new root. If the value being searched for is not in the tree, then the last node accessed prior to reaching the NULL pointer is splayed and becomes the new root.
- **Pro Tip:** You will notice that in the BST_Node example methods, we have containsNode return a BST_Node. While you do not have to do this, think of the potential advantage of returning a Node. If you return the newly splayed root Node to SPLT, you can now set that to Root! You can then do a check to see if the new root is the same as what was requested as a parameter to the SPLT contains method to return true or false.

- **findMin (Max):** this is an access, so when the min (max) is found, that node is splayed to the root.

- **remove**: We do this in several steps.
  - do a "contains" on the node to be removed; this will splay it to the root if it is in the tree; if it is not in the tree, the "contains" splays the last accessed node to the root, and we are done... no node is deleted.
  - unhook the root node, leaving you with two subtrees: the left child L, and the right child R
  - do a "findMax" on L; this will splay the max node to the root, and will also leave that new root of L with no right child subtree (since the value there is larger than all the others in L).
  - make R the right child subtree of L.
  - **pro tip:** If you choose to use the delegated example of BST we provide, note that remove is much easier than last time and doesn't really get much out of delegating. You need not delegate on this method (or at all), to get the desired results.

Test Data

Use the Splay Tree animation (see the link under the "Examples" tab) to show you what trees result after various combinations of insert, remove, contains, etc. operations. Slow the animation way down to see how the operations do tree rotations.

We, as before, will release a JAR file of Junit tests that you can also use to check your work. This release will be about half-way through the assignment time frame, so do not wait to get started. Write your code and test it as you see fit, and use the JAR file as a further check once we release it.

Supporting Code Examples

Random strings

You may find this code useful. 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.