De Anza College
Summer 2016

CIS 22C: Data Abstraction and Structures

Programming Project 2 (Binary Search Trees)

For this programming project, you will be implementing a templatized binary search tree (BST) class. We assume that the objects stored in the BST are unique and have all their comparison operators defined. The instructions are as follows:

1. Implement the **BinarySearchTree** template class using your textbook.

2. Note that not all of the implementations are shown so be sure to implement all of the functions in the **BinarySearchTree** class.

3. After you have finished implementing the **BinarySearchTree** class, add three new methods to the **BinarySearchTree** for printing the elements in *pre-order, in-order, and post-order.*

4. Add a fourth method, called **printTraversals(),** to the BinarySearchTree class, that calls the three different traversal methods. For example, if our current BST is the one shown in Figure 4.22 on page 130, then calling the **printTraversals()** method would produce the following output:

```
Pre-order:  6 2 1 4 3 8
In-order:   1 2 3 4 6 8
Post-order: 1 3 4 2 8 6
```

5. Add a new method, called **balance()**, to the **BinarySearchTree** class, that balances the BST by creating a new tree with the middle element as the root, and then recursively does the same with the left and right subtrees of the new tree using the elements that are less than the middle element and greater than the middle element, respectively. The algorithm for balancing the BST is as follows:

   (a) Copy the elements of the existing BST `T`, in order, into a temporary vector v (so that the elements are sorted in increasing order).

   (b) Create a new empty BST `newT`.

   (c) Insert the elements of `vector v` into `newT` using the following recursive algorithm:

```
recursiveInsert (v, lowIdx, highIdx, newT)
    if (lowIdx <= highIdx)
       midIdx = floor((lowIdx + highIdx) / 2)
       insert v[midIdx] to newT
       recursiveInsert(v, lowIdx, midIdx - 1, newT)
       recursiveInsert(v, midIdx + 1, highIdx, newT)
```

   (d) Delete the old tree `T`, and set `T` to the new tree `newT`.

6. You will test your **`BinarySearchTree`** class by building several BSTs. For this, create a main program (e.g. `main.cpp`) that reads the contents of a text file named `input.txt`. The contents of the input file `input.txt` follows the following format:

```
<number of BSTs to generate>
<number of items in the first BST> <item 1> <item 2> <item 3> ... <item x>
<number of items in the second BST> <item 1> <item 2> <item 3> ... <item y>
 ...
 ...
<number of items in the last BST> <item 1> <item 2> <item 3> ... <item z>
```

In building a `BST`, we assume that the items are inserted into the `BST` in the order that they are enumerated in the input file, i.e. `<item 1>` is inserted first, followed by `<item 2>`, followed by `<item3>`, and so on. After you have inserted all the items for a `BST`, you should:

(a) Print the three different traversals of the `BST`;

(b) Balance the `BST` by calling the `balance()` method on the `BST`; and

(c) Print the three different traversals of the `BST` after it has been balanced.

As an example, suppose the contents of the input file `input.txt` are as follows:

```
3
10  1  2  3  4  5  6  7  8  9  10
10  10  9  8  7  6  5  4  3  2  1
6  6  2  1  4  3  8
```

The output of the program would be:

```
Tree 1:

Pre-order:      1 2 3 4 5 6 7 8 9 10
In-order:       1 2 3 4 5 6 7 8 9 10
Post-order:     10 9 8 7 6 5 4 3 2 1

Balanced Tree 1:

Pre-order:      5 2 1 3 4 8 6 7 9 10
In-order:       1 2 3 4 5 6 7 8 9 10
Post-order:     1 4 3 2 7 6 10 9 8 5

Tree 2:

Pre-order:      10 9 8 7 6 5 4 3 2 1
In-order:       1 2 3 4 5 6 7 8 9 10
Post-order:     1 2 3 4 5 6 7 8 9 10
```

```
Balanced Tree 2:

Pre-order:       5 2 1 3 4 8 6 7 9 10
In-order:        1 2 3 4 5 6 7 8 9 10
Post-order:      1 4 3 2 7 6 10 9 8 5

Tree 3:

Pre-order:       6 2 1 4 3 8
In-order:        1 2 3 4 6 8
Post-order:      1 3 4 2 8 6

Balanced Tree 3:

Pre-order:       3 1 2 6 4 8
In-order:        1 2 3 4 6 8
Post-order:      2 1 4 8 6 3
```

The following are the additional instructions for this programming project:

- This programming project is to be done individually.
- Create a readme.txt file that describes exactly how to compile and execute your program.
- Collect your source codes, readme file, and other files needed to compile and execute your program into one ZIP file called YourFirstName_YourLastName_prog2.zip. **Please DO NOT include any executable files in your ZIP file.**
- Make sure you follow good object-oriented programming approach, good coding style, and proper documentation.

The grading for this programming project will be based not only on the correctness of the program, but also on the program's overall design, coding style, and documentation.

Submitting Assignments:

- Submit the ZIP file on Catalyst