CS14 Winter 2016 Lab 4
Fun with Binary Search Trees

February 6, 2016

DUE: Submit by iLearn by 11:59pm Sunday, February 21, 2016
LATE DEADLINE (50% penalty) 11:59pm Monday, February 22, 2016

1 Introduction

The purpose of this assignment is to give you practice using binary search trees (BSTs). Your program will begin by constructing *two* BSTs, say `A` and `B`, by reading data from an input file in the usual way. Then your program will call a newly-designed function called `splinter` which has *four* BSTs in its parameter list:

1. `A` the first input BST, which is not modified by your function
2. `B` the second input BST, which is not modified by your function
3. `C` the first output BST, which is initially NULL but during execution of the `splinter` function it grows to include all nodes common to both input trees,
4. `D` the second output BST, which is initially NULL but during execution of the `splinter` function it grows to include all nodes that belong to *either one* of the input trees, but *not both*.

Using the terminology from set theory, `C` represents the *intersection* of `A` and `B`, while `D` represents their *symmetric difference*.

Recall from the definition of a BST, that it cannot have more than one node with the same value because `insert` ignores duplicates. However, all nodes in `C` are duplicates between `A` and `B`. Thus if `size()` returns the number of nodes it a BST, then

```
size(A) + size(B) == size(D) + 2 * size(C).
```

As a starting point, you may use *the sample code from the textbook*. If your program uses code from the textbook (or some other public source), then **make sure you clearly identify which parts of the code you wrote yourself, and which source(s) were used for your remaining parts.**

Note that a *very good solution* to this problem would use a balanced AVL tree instead of a plain binary search tree (and then just hoping for the best with respect to balancing the search tree). However, implementing AVL trees is ***not required*** for this assignment, but may earn you up to 20% extra credit. On the other hand, please don't try for the extra credit until you get the basic solution finished, because a garbled implementation of AVL trees is worth considerably less than a working implementation using plain binary trees.

2 Sketch of the `splinter` algorithm

2.1 Step 1, assign the elements of `A` to the correct output

In this step, you must compare each of the values stored in BST `A`, one at a time, with the entire contents of `B`. Whenever you find a value from `A` is *also* in `B`, it must be inserted into `C`, the intersection BST. Otherwise,

the value must be inserted into `D`, the symmetric difference BST. Be sure your code examines the values from `A` in **preorder**. (Can you see what's bad about using an **inorder** traversal?)

HINT: Modify the standard `insert` algorithm for a BST to solve this value-splintering operation. Both problems start off the same way (i.e., a target value and a reference to a BST node), but they act differently when you reach the end of the search phase.

2.2 Step 2, assign the elements of `B` to the correct output

At the end of step 1, the output BST `C` will be complete, and the output BST `D` will contain those values from `A` not also in `B`. However, output BST `D` does not yet contain those values from `B` not also in `A`.

A naïve implementation could finish the job by repeating Step 1 with the roles of `A` and `B` reversed, thanks to the fact that insert will ignore your attempt to add duplicate values to the intersection BST `C`. However, there is a much better approach, which is both much faster and not much more difficult to program.

This time, simply compare each of the values stored in BST `B`, one at a time, with the entire contents of the intersection BST `C`. Whenever you find a value from `B` is *also* in `C`, discard it. Otherwise, the value must be inserted into `D`, the symmetric difference BST.

3 Recursive BST printing function

Write a recursive function `prettyprint` and use it to display each of the the four BSTs generated in your program using the output format shown in the following illustration of a 15-node perfectly-balanced BST:

```
      / 1
    / 2
   |  \ 3
  / 4
 | |  / 5
 |  \ 6
 |    \ 7
< 8
 |    / 9
 |  / 10
 | |  \ 11
  \ 12
   |  / 13
    \ 14
      \ 15
```

Notice that each node from the BST is printed on a separate line, and the order of the nodes is determined by an *in-order traversal* of the BST (assuming left means first on the page). The level of indenting for a node is determined by its depth in the BST, and the closest non-blank symbol to the left of the node value is its `nodeTag`, which indicates the direction of the connection to its parent (if any). Columns of vertical bars are used to fill the gap in the connection between a node and its parent if they are separated by intermediate rows for displaying the node's own descendants.

The key to writing `prettyprint` is how to manage the string of characters that comes in front of each node's value when it is printed, and how to pass it from one recursive call to another through the `linePrefix` parameter. To help clarify the details on how `linePrefix` is used, in this section I will represent each space character in the string by the symbol `"␣"`, which looks like an underline with its ends turned up slightly and is called the "visible space character".

Your `prettyprint` function should include the following parameters:

- `BinaryNode *t` which points to the root node of the current (sub)tree
- `string linePrefix` which is initialized to `""` for the root and thereafter grows in length by *two characters* for each level of recursion, according to the following rules:

  - The root adds "␣␣" to the string passed to both of its recursive calls.
  - For all other nodes, the extension is "␣␣" for the recursive call in the *outside direction*, and "|␣" for the recursive call in the *inside direction*. (See below for a definition.)

- `string nodeTag` which is "␣<" if `t` is the tree root, "␣/" if `t` is a left child (and hence must be printed before/above its parent), and "␣\" if `t` is a right child (and hence must be printed after/below its parent)

When the function is ready to visit node `t`, it forms an output row that consists of `linePrefix`, `nodeTag` and finally `t`'s value.

To illustrate the method, let us return to the example BST shown above. Notice that node 8 passes `linePrefix="␣␣"` in its recursive calls to both nodes 4 and 12 because 8 is the tree root. However, node 4 passes `linePrefix="␣␣␣␣"` in its recursive call to node 2, because 2 is its *outside child*, i.e., node 2 is the left child of a left child and hence further than 4 from 4's parent. Conversely, node 4 passes `linePrefix="␣␣|␣"` in its recursive call to node 6, because 6 is its *inside child*, i.e., node 6 is the right child of a left child and closer than 4 to 4's parent. This difference is responsible for the left-most vertical bar visible on rows 5–7, but does not extend to rows 1–3.

Applying the same logic to the next level, node 6 passes `linePrefix="␣␣|␣|␣"` to its inside child, node 5, and `linePrefix="␣␣|␣␣␣"` to its outside child, node 7. Beware of how far to the left of node 5 is the last vertical bar added to `linePrefix` by node 6! When node 5 is visited, this bar will actually be printed one character further left than 5's grandparent, node 4, because its function is to extend the link between nodes 4 and 6.

4 Program input and output

Your main program should accept *one* input filename as a command-line argument and generate *one* similarly-named output file with the original filename extension (if any) changed to "`.out`". For example, if the input file were `sample_input.txt`, then the output file would be `sample_input.out`.

The input file is a plain ASCII text file, consisting of a variable number of test cases (with two rows of data per test case), terminating with end-of-file.

For each test case, the first row contains a series of one or more numbers (not necessarily integers) separated by white space, giving the elements belonging to BST `A`. The second row contains another series of one or more numbers separated by white space, giving the elements belonging to BST `B`.

Once you have created the two input BSTs `A` and `B` use your `splinter` function to create the output BSTS `C` (representing intersection) and `D` (representing symmetric difference).

For each test case, your program must produce the following output.

- A line which says "TEST CASE: " followed by the test case number
- Four *single lines*, each containing a list of all the elements from one of the four BSTs `A`, `B`, `C`, and `D`. The values on each line should be numbers separated by white space, *sorted into increasing order*. [HINT: use a simple in-order traveral to extract them from the BST.]
- one blank line followed by a `prettyprint` version of BST `A`
- one blank line followed by a `prettyprint` version of BST `B`
- one blank line followed by a `prettyprint` version of BST `C`
- one blank line followed by a `prettyprint` version of BST `D`

Note that the single-line output format makes it easy to check the correctness of your intersection and symmetic-difference calculations. The `prettyprint` output format allows us to see the layout of those BSTs. This is especially important if you implement balanced AVL trees.