CS14 Winter 2016 Lab 4
Fun with Hyper Balancing Trees

November 9, 2016

DUE: Submit by iLearn by 1:00am Wednesday, November 23, 2016
LATE DEADLINE (50% penalty) 1:00am Saturday, November 26, 2016

1 Introduction

The purpose of this assignment is to give you practice manipulating *ordered binary trees* (OBTs). The definition of an OBT is exactly the same as a binary search tree (BST) except that duplicate nodes are allowed. In other words, OBTs satisfy the following recursive ordering property:

> For any node `x` in an OBT, the value `x->element` is *greater than or equal to* the value of every node in its left sub-tree, and *less than or equal to* the value of every node in its right subtree.

(BSTs satisfy an equivalent ordering rule, except that duplicate elements are not allowed, so all comparisons use strict inequalities.)

Your program will read a sequence of test cases, and for each one construct three versions of an OBT:

1. A *completely unbalanced* tree, where there are no restrictions on where new elements are inserted, and no effort is made to balance the tree even if it becomes highly skewed.
2. An *AVL* tree, which uses the tree-balancing algorithm described in section 4.4 of the textbook to control the *difference in heights* between the two subtrees of every node.
3. A *hyper balanced* tree, which uses the algorithm described below in section 3 to control the *difference in the total number of nodes* between the two subtrees of every node.

As a starting point, **you may use the sample BST and AVL code from the textbook**

- C++ 98 version from 3rd Edition of text
- C++ 11 version from 4th Edition of text.

However, 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.**

1.1 Node Structure

Use the following node structure for all three versions of your OBT code (i.e., unbalanced, AVL, and hyper balanced). It is the same as the sample BST code from the textbook, except for three adding `int` fields: `height` (required by AVL) and both `lsize` and `rsize` (required by hyper balanced). For example here is the modified structure for the C++ 98 version of the sample code:

```
private:
  struct HB_Node
  {
      Comparable element;
      HB_Node *left;
      HB_Node *right;
      int height;
      int lsize;
      int rsize;

      HB_Node( const Comparable & theElement, HB_Node *lt, HB_Node *rt,
              int h = 0, int ls = 0, int rs = 0 )
          : element( theElement ), left( lt ), right( rt ),
              height ( h ), lsize( ls ), rsize( rs ) { }
  };

  HB_Node *root;
```

2 Recursive binary tree printing function

Write a recursive function `prettyprint` and use it to display each of the three OBTs generated in your program using the output format shown in the following illustration of a 15-node perfectly-balanced OBT.

I strongly recommend that you implement this printing function *early* during your program development, and certainly before you write the code for hyper balanced OBTs described below. This printing function will provide you with an important debugging tool, so you can see how each step of your code modifies the shape of a tree as it runs.

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

Figure 1: `prettyprint` output for a 15-node perfectly-balanced tree

Notice from the example shown in Figure 1 that the values for each node from the OBT are printed on separate lines, and that a pattern of line segments and indenting levels is used to indicate the how each node is connected to the rest of the tree. The printing order is chosen to look like a "standard" drawing of an ordered binary tree (i.e., the root is on top and each node is above its children, with its smaller child is on the left and larger child on the right) but the entire diagram has been *rotated* 90° *in the counterclockwise direction*.

Since the *first* (top) printed line contains the numerically *last* data value, the order in which the nodes are printed does not follow one of the usual tree traveral algorithms (i.e., pre-order, in-order and post-order).

Instead, you must follow a traversal called *reverse in-order*, where at node `t` you: (i) traverse `t->right`; (ii) visit `t`; and finally (iii) traverse `t->left`.

The level of indenting for a node is determined by its depth in the OBT, 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:

- `HB_Node *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.)
  - TIP! The vertical bar is used to construct the line from this node to its *parent*, not its children.
- `string nodeTag` which is "␣<" if `t` is the tree root, "␣/" if `t` is a right child (and hence must be printed before/above its parent), and "␣\" if `t` is a left 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 OBT shown in Figure 1. Starting from the tree root, node 8 always passes `linePrefix="␣␣"` in both of its recursive calls, first to node 12 and later to node 4 because the tree root doesn't have a connection to an even-higher level node above or below the currently visible tree. However, node 12 passes `linePrefix="␣␣␣␣"` in its first recursive call to node 14, because 14 is its *outside child*, i.e., node 14 is the right child of a right child and hence further than 12 from 12's own parent. Conversely, node 12 passes `linePrefix="␣␣|␣"` in its second recursive call to node 10, because 10 is its *inside child*, i.e., node 10 is the left child of a right child and closer than 12 to 12's own parent. This difference is responsible for the left-most vertical bar visible on rows for data items 9–11, and is used to build the connection from 12 back to its parent 8. This same line does *not* extend upwards to the rows for data items 13–15 because the connection to 12's own parent does not go in that upwards direction.

Applying the same logic to the next level, node 10 passes `linePrefix="␣␣|␣|␣"` to its inside child, node 11, and `linePrefix="␣␣|␣␣␣"` to its outside child, node 9. Beware of how far to the left of node 11 is the last vertical bar added to `linePrefix` by node 10! When node 11 is visited, this bar will actually be printed one character further left than 11's grandparent, node 12, because its function is to extend the link between nodes 10 and 12.

Figure 2 shows another example of `prettyprint` output, but this time color coding has been added to show the contributions of each recursive call to the final output. Starting from the initial call to the root (node 4, bright green), the function immediately makes a recursive call to its right subtree (node 7, dark pink). Because 4 is the root, the recursive call adds two blanks (shown in bright green) to the `linePrefix` parameter. This second function call immediately makes a recursive call to its right subtree (node 9, purple), while adding two blanks (shown in purple) to the `linePrefix` parameter because node 9 is its *outside child* (i.e., child 9 and parent 4 are on opposite sides of node 7). This third function call prints its root node value (9 in purple) because it has no right subtree, then makes a fourth recursive call to its left subtree (node 8, grey). This time node 9 adds a vertical bar to `linePrefix` because node 8 is its *inside child* (i.e., child 8 and parent 7 are on the same side of node 9).

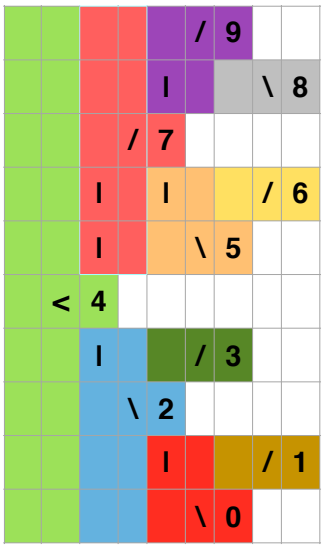

Figure 2: An example of `prettyprint` applied to an irregularly-shaped OBT.

3 Description of the hyper balancing algorithm for OBTs

For any node `t`, let `t->lsize` and `t->rsize` be the *total number of nodes* in its left and right subtrees:

- if `t->left == nullptr` then `t->lsize = 0`;
- otherwise `t->lsize = 1 + t->left->lsize + t->left->rsize`.

The value for `t->rsize` uses a mirror image of this definition.

We say that an OBT is *hyper balanced* if every node `t` satisfies: `abs (t->lsize - t->rsize) < 2`. In other words, only these three possibilities are allowed:

1. `t->lsize = (t->rsize - 1)`, i.e, right subtree has one extra node
2. `t->lsize = t->rsize`, i.e, both subtrees have the same number of nodes
3. `t->lsize = (t->rsize + 1)`, i.e, left subtree has one extra node

Every other possibility can be reduced to one of these three, by shifting nodes from one subtee to the other.

3.1 Functions for updating a hyper balanced OBT

In this assignment, you must be able to *build* an OBT by inserting a sequence of elements into an OBT, which is initially empty. You *will not* be required to *shrink* an existing OBT after it has been built by deleting any of its elements. However, you *will* need the capability of *moving* an existing node from one location in the OBT to another in order to restore hyper balance to the OBT after inserting a new element and/or moving an existing element to another location.

You will need to implement the following recursive functions:

- `void insert( HB_Node * & x, HB_Node * & t )`

  This function connects a currently-disconnected node `x` into the appropriate location in the subtree rooted at node `t`, which satisfies the ordering of data elements.

  – Unlike the textbook's BST code, we assume that the first parameter is a `HB_Node` object that already holds the data value in its `x->element` field. This change allows you to use a single function for both inserting new data values into the OBT as well as moving existing data values to different locations while rebalancing the tree.

- Note that `insert` must update the values of `t->lsize` and `t->rsize` for every node `t` it visits during the insertion process to account for the presence of this newly-inserted node.
- If the updated values of `t->lsize` and `t->rsize` no longer satisfy the hyper balancing property, then you must call `rebalance(t)` (described below) to restore the property.

- `void rebalance( HB_Node * & t )`

  This function should never be called unless `t->lsize = (t->rsize+2)` or `t->lsize = (t->rsize-2)`; anything else means something bad has happened to your program.

  If `t->lsize = (t->rsize+2)` then your function must:

  - call the `removeMax` function to disconnect the node, `z` say, containing the largest element from `t->left`;
  - swap the data fields between subtree root `t` and disconnected node `z`; and finally
  - call the `insert` function to reconnect the node `z` into the correct location within `t->right`.

  Otherwise, `t->lsize = (t->rsize-2)` so your function must:

  - call the `removeMin` function to disconnect the node, `z` say, containing the smallest element from `t->right`;
  - swap the data fields between subtree root `t` and disconnected node `z`; and finally
  - call the `insert` function to reconnect the node `z` into the correct location within `t->left`.

  Note that `rebalance` should *never call itself directly.* However, its calls to `removeMax`, `removeMin` and `insert` may cause those functions to detect an out-of-balance condition at a different location in the tree, thus triggering another call to `rebalance`.

- `HB_Node * removeMax( HB_Node * & t )`

  This function finds the node containing largest element in the subtree rooted at `t`, disconnects the node from the tree, and returns a pointer to that node.

  - Note that `removeMax` must update the values of `t->lsize` and `t->rsize` for every node `t` it visits during the removal process to account for the absence of this newly-removed node.
  - If the updated values of `t->lsize` and `t->rsize` no longer satisfy the hyper balancing property, then you must call `rebalance(t)`

- `HB_Node * removeMin( HB_Node * & t )`

  This function finds the node containing smallest element in the subtree rooted at `t`, disconnects the node from the tree, and returns a pointer to that node.

  - Note that `removeMin` must update the values of `t->lsize` and `t->rsize` for every node `t` it visits during the removal process to account for the absence of this newly-removed node.
  - If the updated values of `t->lsize` and `t->rsize` no longer satisfy the hyper balancing property, then you must call `rebalance(t)`

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 rows of numbers (not necessarily integers) separated by white space, terminating with end-of-file.

Each row of numbers represents a separate test case. Apply each of the three algorithms described above to construct a separate OBT for each test case.

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

- A line which says "TEST CASE: " followed by the test case number
- one blank line followed by a `prettyprint` version of OBT produced by your *completely unbalanced tree* algorithm
- one blank line followed by a `prettyprint` version of OBT produced by your *AVL tree* algorithm
- one blank line followed by a `prettyprint` version of OBT produced by your *hyper balanced tree* algorithm