CS14 Winter 2017 Lab 6
Creating and Managing a Generic (i.e, *not*-Binary) Tree Class

February 14, 2017

LIVE DEMO [20 points]: **Demonstrate 2.1(a)-(c), 3, 4 to your TA in Lab or his office hours**
The **entire assignment** should be submitted by iLearn by 1:00am Tuesday morning, February 21, 2017
LATE DEADLINE (50% penalty) 1:00am Friday morning, February 24, 2017

1 Introduction

Chapter 4 from the textbook is about trees, but spends most of its time on the more specialized topic of *binary search trees*. Thus, in this assignment you will be working with *generic trees* where the data elements have *not* been ordered to allow the find/insert/delete operations to be implemented efficiently. In particular:

(a) Each node in a generic tree can have *any number* of children, and the order of the children is *not significant*. In other words, if node A has 3 children F, C, D then the order in which we show those children doesn't matter (i.e., D, F, C is equally good, and both represent the same information).

(b) Because the *number of children may be different for every node*, your program must use the `firstChild`, `nextSibling` representation which is illustrated in the diagram of Figure 4.4 from the textbook and given in the code structure of Figure 4.3, i.e.,

```
struct TreeNode
{
  Comparable   element;
  TreeNode *firstChild;
  TreeNode *nextSibling;
  int height; // added because you need it later
}
```

(c) You may use **the binary search tree class skeleton from the textbook** (Figure 4.16 and beyond) as a starting point for your code, but **be aware that many of its algorithms will not work correctly for your application**. As usual, you must identify which parts of the code came from other sources, including the textbook, and which parts you wrote by yourself.

2 Generic tree construction

Your program must be able to construct a generic tree by reading data from an input file. For example, **here is a file** which can be used to recreate Figure 4.2 from the textbook.

The first line of the file starts with a single character:

- The character "`d`" or "`D`" indicates that all the values stored in this tree are *numbers* (i.e., any combination of integers and/or real numbers), which your program must store in a `GenericTree<double>`.

- The character "s" or "S" indicates that all the values stored in this tree are *strings* (i.e., sequences of letters and/or digits without any included white space and without being enclosed in quotation marks), which your program must store in a `GenericTree<string>`.

Also on the first line, but separated from the first character by white space, is the *value for the root*. Thus, to create a new generic tree of `double` values, using the value stored in `root_val` for the root, you would execute a statement like this:

```
GenericTree<double> t(root_val);
```

Each of the remaining lines from the input file will consist of *exactly two values* of the specified type separated by white space, representing a new node to be added to the existing tree.

- The value of the new node
- The value of the parent of the new node

Using these input values, you would then add the new node to the tree with a statement like this:

```
what_happened = t.insert(new_val, parent_val);
```

Note that the value returned, indicates whether `insert` failed because of an error, namely (i) the value provided for the new node is *already* in the tree, or (ii) the value provided for its parent is *not* in the tree.

2.1 Public member functions for the `GenericTree` class

(a) `GenericTree ( Comparable root_val );`

is the *constructor* for initializing a new `GenericTree`. You *must* supply a `Comparable` value to initialize the root's `element` field. The constructor creates the root node, sets its `firstChild` and `nextSibling` pointers to null and sets its `height` to `-1` (indicating that its height has not yet been determined).

(b) `int insert ( Comparable new_val, Comparable parent_val );`

creates a node with `element` field `new_val`, and adds the node to the `GenericTree` as the next child of an existing node with `element` field `parent_val`. This function is described below in section 3.

(c) `void print_Tree();`

displays the contents of your entire `GenericTree` in an "indented list" style. This function is described below in section 4.

(d) `int set_height();`

initializes the `height` *field* for every node in the `GenericTree`. This function is described below in section 5.

3 The `insert` member function [30 points]

The `insert` public member function uses the two `Comparable` values provided by the input file as its input arguments, which represent the `element` values of the new node and its parent, respectively. It uses the `contains` recursive private member function (described immediately below) to (i) verify that the new value is not already in the `GenericTree`, and (ii) obtain a pointer to the parent of the new node. Thus:

- If `new_value` **is found** in the tree, then `insert` *fails*, and returns value `-1` to indicate that `new_val` was already in the tree.
- Otherwise, if `parent_val` **is not found** in the tree, then `insert` *fails*, and returns value `-2` to indicate that `parent_val` was missing from the tree.

- Otherwise, `parent_val` **is found** in the tree at `key_loc`, so `insert` *succeeds*, and returns `+1` to indicate that a new node was created and added to the tree as a child of the node at `key_loc`. In addition, the new node is initialized by
  - setting its `element` value to `new_val`
  - setting its `firstChild` and `nextSibling` pointers to null
  - setting its `height` to -1 (indicating that its height has not yet been determined)

3.1 The `contains` private member function

`bool contains (const Comparable & key_val, TreeNode *rt, TreeNode * & key_loc);`
is a recursive private member function that is called as a helper function by the `insert` public member function, starting from the root of the `GenericTree`. It searches the sub-tree rooted at node `*rt`, looking for a node whose `element` value is the same as its argument `key_val`:

- If none of the nodes in this sub-tree has `key_val`, then the function returns `false` and the value of `key_loc` remains untouched.
- Otherwise, exactly one node must have `key_val`, because *duplicate values are not allowed* in the tree. In this case, the function returns `true` and sets `key_loc` to point to the node holding this value.

Note that Figure 4.18 from the textbook, which shows a `contains` function for binary search trees, is **not a good hint for writing your function**. *Your problem* is *not* a binary tree, the nodes in *your tree* are *not* ordered, and *your function* has the *additional requirement* of returning a pointer to the location of `key_val`.

Since the values in the tree are not ordered in any way, your `contains` function must be prepared to carry out an *preorder traversal* of the entire tree to determine that none of its nodes has `element` value equal to `key_val`. However, if it finds that a particular node `*key_loc` *does have* `key_val` as its `element` value, then the function may be able to quit the traversal early.

4 Printing the generic tree [25 points]

In order to help you visualize the generic trees you are creating, you must write the following print function. Sadly, because generic trees have no recognizable shape, the printed tree will use an "indented list" style, similar to Figure 4.7 in the textbook, but with extra node labels to help you keep track of the structure. More specifically:

- You *must* print all nodes in the order determined by a *preorder traversal* of the entire tree

  Make sure you understand how to get a preorder traversal with our tree representation!
- From the main program, call your print function using the following public member function:

  `t.print_Tree ()`

  This function initializes the *hierarchical tag value* for the root:

  `vector<int> nodeLabel; nodeLabel.push_back(1);`

  then calls its recursive private helper function (described immediately below) starting from the root of the `GenericTree`.

4.1 The `print_Tree` private member function

`void print_Tree (const TreeNode *rt, vector<int> & nodeLabel)`
is a recursive private member function which carries out the actual printing task for the sub-tree rooted at node `*rt`. Notice that its function signature has the same name as the public `print_Tree` member function, but its parameter list has been changed.

Each time the recursive helper function is called, its argument list already provides it with all the necessary information to print *one line of output* describing the node `*rt`, which is the *root* of the current sub-tree.

- First, it prints the entire *hierarchical tag value* it received as an input argument, as a series of integer values, each followed by a single period.
- Next, it prints an opening square bracket, `"["`, the `height` field for node `*rt`, followed by a closing square bracket, colon and space, `"]: "`.
- Finally, it prints the `element` value for node `*rt`, followed by `endl`.

Thus, for example, the line printed for the root of the tree in Figure 4.2 of the textbook should consist of the following parts, one after the other:

- hierarchical tag for the root: `"1."`
- its height (not yet initialized): `"[-1]: "`
- its element value: `"A"`

As the tree traversal continues, however, the recursive helper function must continually update the vector that determines the hierarchical tag value, according to the following rules:

- The first element of the vector is always 1 because our trees always have start with a single root.
- The remaining elements of the vector represent the path from the root to node `*rt`, where tag element value for each node along that path corresponds to its "birth order" within its own family, i.e.,
  - Since the birth order for a *first child* is 1, the vector for a first child is the same as its parent, but with one extra element with value 1 added to the end;
  - Similarly, since the birth order for a *second child* is 2, the vector for a second child is the same as its parent, but with one extra element with value 2 added to the end, or equivalently, the same as the *first child* sibling with the value of the last element changed from 1 to 2.
  - Continuing with the same pattern, the vector for a *third child* is the same as the *second child* to its left with the value of the last element changed from 2 to 3, and so on.

  For example, `"1.4."` is the 4th child of the root, and `"1.4.3."` is the 3rd child of that node.
- Thus, when making the recursive call that moves one level down the tree to a node's *first* child:
  - You must *expand* the tag via: `nodeLabel.push_back(1);` *before* the recursive call
  - You must *reduce* the tag via: `nodeLabel.pop_back();` *after* the call returns.
- Similarly, when making the recursive call the moves horizontally to a node's next sibling:
  - You must *increment* the last element of the vector *before* the recursive call.
  - You must *decrement* the last element of the vector *after* the call returns.

Notice that are updates to the `nodeLabel` vector are very similar to a stack, since we always add and remove elements from the "top" end. However, it is not quite a "pure" stack because we must access the entire vector to print it, and because we use increment/decrement of the last element, instead of push/pop, when the recursive calls move sideways between siblings instead vertically between generations.

Here is an example of what printing the tree from Figure 4.2 (or, equivalently, 4.4) should look like (assuming the `height` fields have not yet been properly initialized):

```
1.[-1]: A
1.1.[-1]: B
1.2.[-1]: C
1.3.[-1]: D
1.3.1.[-1]: H
1.4.[-1]: E
1.4.1.[-1]: I
```

```
1.4.2.[-1]: J
1.4.2.1.[-1]: P
1.4.2.2.[-1]: Q
1.5.[-1]: F
1.5.1.[-1]: K
1.5.2.[-1]: L
1.5.3.[-1]: M
1.6.[-1]: G
1.6.1.[-1]: N
```

5 Establishing the height of each node [25 points]

The `set_height` public member function initializes the `height` field for every node in your `GenericTree`, and returns the height of the root.

The previous values of the `height` field (if any) are ignored and replaced by new values determined by the usual definition: The *height* of a node is the length (i.e., number of edges) in the longest downward path from this node to a leaf. This definition can be converted into the following algorithm:

- The `height` for a *leaf node* is 0.
- The `height` of a *non-leaf node* is 1 plus the maximum height for any of its children.

"Clearly" the way to apply the algorithm is to create a recursive private member function to carry out a *postorder traversal* of the tree. (Note that you want *postorder*, rather than *preorder*, so the heights of your children will be calculated before you need them.)

```
void set_height( TreeNode *rt )
```

HINT: the structure of your `print_Tree` recursive private member function may be a good starting point for writing the recursive `set_height` function, because you already figured out how to visit all the nodes in the entire `GenericTree`. However, you will need to move the code for "visiting" the current node from the beginning to the end of the function body, in order to change the traversal order from preorder to postorder.

Now suppose your function is currently working on the sub-tree rooted at node `*rt`. Your *first task* is to make recursive calls to all the children of node `*rt` because you must carry out a postorder traversal of the tree. After all of those recursive calls have been completed, you know that the `height` fields for all of the children of node `*rt` must now hold the correct values. Thus, your *second task* is to examine the `height` fields for all of your children to find the largest value, then add one and store the result as your own `height`. You should be able to find the maximum `height` from among all your children without resorting to recursion – just a simple loop is enough!

6 Summary of Tasks

For the *live demo* you must complete sections 2.1(a)–2.1(c), 3, and 4. In other words, you need to be able to

- construct a new `GenericTree` by reading an input file (section 3), and
- print the entire tree as an indented list with hierarchical tags (section 4).

For the *final submission*, you must complete all sections of the assignment. That is, you need to be able to

- construct a new `GenericTree` by reading an input file (section 3),
- print the entire tree as an indented list with hierarchical tags (section 4),
- fill in the values for the height field of each node (section 5)
- print the entire tree again (section 4), to demonstrate that the height values were assigned correctly.