CPSC 131 Project 4: countdown trees

Fall 2015
Prof. Kevin Wortman, CSU Fullerton
kwortman@fullerton.edu

Introduction

In this project you will implement a *countdown tree,* which is a kind of self-balancing binary search tree. Countdown trees are significantly simpler than AVL trees, but their performance is slightly worse.

Countdown trees

Countdown trees are a relatively obscure form of self-balancing binary search tree. They are described briefly in Exercise 8.2.9 of Open Data Structures. This section explains them in more detail.

Recall that AVL trees always check their balance invariant after each insert and remove operation. Whenever the invariant is violated, the tree immediately rebalances itself through an elaborate rotation process. So an AVL tree's height is always $O(\log n)$. The rebalancing process takes only $O(\log n)$ time. So the get, put, and erase operations each take $O(\log n)$ *worst-case* time in an AVL tree.

Countdown trees use a simpler policy, based on the idea of *scheduled maintenance.* Each node in a countdown tree has a timer that keeps track of how "worn out" the node is. When the timer says that the node is due for maintenance, its subtree is ripped apart and rebuilt as a perfect binary subtree. This is similar to how dental cleanings work; most patients have a dental checkup every 6 months, like clockwork, regardless of the precise condition of each tooth.

The good news is that the algorithm for rebuilding a binary tree is substantially simpler to describe and implement than is the AVL rebalancing algorithm. The bad news is that rebuilding a subtree can take as much as $O(n)$ time. So the get, put, and erase operations take $O(\log n)$ *amortized* time in a countdown tree, which is slightly worse than the performance of AVL trees.

A countdown tree class contains the same two data members as a plain BST:

1. a pointer to the root node, which is NULL when the tree is empty; and
2. an int for the size of the tree.

A countdown tree node has the same data members as a plain binary node, plus a timer:

1. a key;
2. a value;
3. two pointers to left and right child nodes, which may be NULL; and
4. an integer *timer.*

When a new node is created, its timer is initialized to 1. A node's timer is decremented whenever another node is added or removed from its subtree. Put another way, when a node is inserted or removed, all nodes on the path from the root to the affected node have their timers decremented. Whenever some node u has a zero timer, the entire subtree rooted at u is rebuilt as a perfectly-balanced subtree. Each node x involved in the rebuilding has its timer reset to $size(x)/3$, where $size(x)$ is the size of the subtree rooted at x.

The following countdown tree operations do not interact with the node timers, so work exactly the same as in a plain BST:

1. create empty (constructor);
2. clear;
3. destructor;
4. is_empty;
5. size getter; and
6. get (search).

The put and erase operations are the only ones that affect timers, so those need to be different from plain BST operations. The erase operation is not required for this project. So most of your effort will go toward re-implementing the `insert` private helper function that adds a node to a BST.

Put and insert

Your countdown tree must have a public `put` function which, as usual, searches for a node containing a given key. If a matching node is found, the node's value is updated; otherwise a new leaf node is added to the tree using a private recursive `insert` function.

The following pseudocode explains how this insert function works in a countdown tree. As with our plain BST and AVL tree, it takes a pointer to a node as an argument, and returns the new node object that takes it place.

```
insert(key, value, subtree):
      if subtree is NULL:
            return new node containing key, value, with timer initialized to 1
      else if key < subtree->key():
            subtree->set_left(insert(key, value, subtree->left()))
            return maintain(subtree)
      else:
```

```
            subtree->set_right(insert(key, value, subtree->right()))
            return maintain(subtree)
```

The previous pseudocode is identical to that of the plain BST, except that it uses a new subroutine `maintain`. `maintain`'s job is to decrement subtree's timer and rebuild the subtree when necessary. It returns the node that takes subtree's place, just as the `insert` function does.

```
maintain(subtree):
      decrement subtree's timer
      if subtree->timer() > 0:
            return subtree // no rebuild necessary
      else: // rebuild
            k = size(subtree)
            array = new k-element array of node pointers
            tree_to_array(array, 0, subtree)
            subtree = array_to_tree(array, 0, k)
            delete array
            return subtree
```

That pseudocode makes use of three more helper functions: `size`, `tree_to_array`, and `array_to_tree`.

`size` computes the number of nodes in a subtree. We actually covered this function in class, as an example of recursion.

```
size(subtree):
      if subtree is NULL:
            return 0
      else:
            return 1 + size(subtree->left()) + size(subtree->right())
```

`tree_to_array` initializes an array of node pointers with all the nodes of a subtree in order. It is based on a standard recursive inorder traversal. In addition to the array object and subtree node pointer, it also takes the starting index to use as an argument. It returns the next index that needs to be initialized.

```
tree_to_array(array, subtree, start_index):
      if subtree is NULL:
            return start_index
      else:
            start_index = tree_to_array(array, subtree->left(), start_index)
            array[start_index] = subtree
            return tree_to_array(array, subtree->right(), start_index+1)
```

Finally, `array_to_tree` takes an array of node pointers, `start_index`, and `end_index` as arguments. It rearranges all the nodes whose index `i` is in the range `start_index` ≤ `i` < `end_index` into a well-balanced tree, and returns the root of that tree.

```
array_to_tree(array, start_index, end_index):
      k = (end_index - start_index) // number of nodes in this subtree
      if k == 0:
            return NULL // empty subtree
      else:
            middle = (start_index + end_index) / 2
            root = array[middle]
            root->set_left(array_to_tree(array, start_index, middle))
            root->set_right(array_to_tree(array, middle+1, end_index))
            if k >= 3: // make sure k/3 doesn't round down to 0
                  root->set_timer(k / 3)
            else:
                  root->set_timer(1)
            return root
```

Altogether, `maintain` and its helpers rebuild a subtree into a perfect BST (or as close to perfect as is possible). It takes only $O(k)$ time, where k is the size of the rebuilt subtree. While there are more helper functions than in AVL trees, each is considerably simpler. None of these functions are "scary" in the way that the AVL rotation process is.

The code

A ZIP archive file, `project4.zip`, is available in TITANium. It contains the following files:

1. `cdtree.hh` is a header file that declares a `CDNode` and `CDTree` class. Currently, this code "cheats" by being a copy-paste of our plain BST class. You will need to refactor the code to actually be a countdown tree instead of a plain BST.
2. `test_main.cc` is a source file for a program that uses the `assert(...)` function to test whether the countdown tree works. First, it puts many random keys and values into the tree, and checks that the countdown tree gets the same results as an STL `map`. Then, it inserts strictly increasing keys, which triggers the worst-case performance of a plain BST. This step should become much, much faster once your countdown tree rebalances properly.

As presented, each file should compile cleanly, and all tests should pass.

What to do

As described above, the code in `cdtree.hh` compiles and passes its tests, but is not actually a countdown tree. You will need to modify the `CDNode` and `CDTree` classes so that they actually implement a countdown tree. You should leave the declarations of all the `CDTree` member functions the same, so that when you're done, all the tests in `test_main.cc` still compile and pass.

Sample output

The following is the output of the test program, when all tests succeed.

```
validation with std::map...
clear...
worst case...
```

Deliverables

Upload the following file to TITANium:

1. `cdtree.hh`

Each file must be uploaded as a plain text source file. That means, for example, that you must not upload a ZIP or DOC file.

For full credit, your code must compile, link, and run cleanly against the provided `test_main.cc`. I may use the output of the test program to confirm the correctness of your code while grading. If your code passes all those tests, it is likely to be correct. However, the test program does not check every single aspect of the module. (For instance, it does not check for memory leaks.) So passing all the tests does not guarantee a 100% grade.

Deadline

The project deadline is Friday, December 18, 11:55 pm. Late submissions will not be accepted.

License