Review from Lecture 16 and Lab 9

- Binary Trees, Binary Search Trees, & Balanced Trees
- STL set container class (like STL map, but without the pairs!)
- Finding the smallest element in a BST.
- Overview of the ds_set implementation: begin and find.

Today’s Lecture

- Warmup / Review: destroy_tree
- A very important ds_set operation: insert
- In-order, pre-order, and post-order traversal; Breadth-first and depth-first tree search
- Finding the in-order successor of a binary tree node, tree iterator increment

17.1 Warmup Exercise

- Write the ds_set::destroy_tree private helper function.

17.2 Insert

- Move left and right down the tree based on comparing keys. The goal is to find the location to do an insert that preserves the binary search tree ordering property.
- We will always be inserting at an empty (NULL) pointer location.
- Exercise: Why does this work? Is there always a place to put the new item? Is there ever more than one place to put the new item?

- IMPORTANT NOTE: Passing pointers by reference ensures that the new node is truly inserted into the tree. This is subtle but important.
- Note how the return value pair is constructed.
- Exercise: How does the order that the nodes are inserted affect the final tree structure? Give an ordering that produces a balanced tree and an insertion ordering that produces a highly unbalanced tree.
17.3 In-order, Pre-order, Post-order Traversal

- Reminder: For an exactly balanced binary search tree with the elements 1-7:
  - In-order: 1 2 3 (4) 5 6 7
  - Pre-order: (4) 2 1 3 6 5 7
  - Post-order: 1 3 2 5 7 6 (4)

- What is the traversal order of the destroy_tree function we wrote earlier?

17.4 Depth-first vs. Breadth-first Search

- We should also discuss two other important tree traversal terms related to problem solving and searching.
  - In a depth-first search, we greedily follow links down into the tree, and don’t backtrack until we have hit a leaf.
  - When we hit a leaf we step back out, but only to the last decision point and then proceed to the next leaf.
  - This search method will quickly investigate leaf nodes, but if it has made “incorrect” branch decision early in the search, it will take a long time to work back to that point and go down the “right” branch.
  - In a breadth-first search, the nodes are visited with priority based on their distance from the root, with nodes closer to the root visited first.
  - In other words, we visit the nodes by level, first the root (level 0), then all children of the root (level 1), then all nodes 2 links from the root (level 2), etc.
  - If there are multiple solution nodes, this search method will find the solution node with the shortest path to the root node.
  - However, the breadth-first search method is memory-intensive, because the implementation must store all nodes at the current level – and the worst case number of nodes on each level doubles as we progress down the tree!

- Both depth-first and breadth-first will eventually visit all elements in the tree.

- Note: The ordering of elements visited by depth-first and breadth-first is not fully specified.
  - In-order, pre-order, and post-order are all examples of depth-first tree traversals.
    - Note: A simple recursive tree function is usually a depth-first traversal.
  - What is a breadth-first traversal of the elements in our sample binary search tree above?

17.5 General-Purpose Breadth-First Search/Tree Traversal

- Write an algorithm to print the nodes in the tree one tier at a time, that is, in a breadth-first manner.

- What is the best/average/worst-case running time of this algorithm? What is the best/average/worst-case memory usage of this algorithm? Give a specific example tree that illustrates each case.
17.6 Tree Iterator Increment/Decrement - Implementation Choices

- The increment operator should change the iterator’s pointer to point to the next TreeNode in an in-order traversal — the “in-order successor” — while the decrement operator should change the iterator’s pointer to point to the “in-order predecessor”.

- Unlike the situation with lists and vectors, these predecessors and successors are not necessarily “nearby” (either in physical memory or by following a link) in the tree, as examples we draw in class will illustrate.

- There are two common solution approaches:
  - Each node stores a parent pointer. Only the root node has a null parent pointer. [method 1]
  - Each iterator maintains a stack of pointers representing the path down the tree to the current node. [method 2]

- If we choose the parent pointer method, we’ll need to rewrite the insert and erase (which we’ll write next lecture!) member functions to correctly adjust parent pointers.

- Although iterator increment looks expensive in the worst case for a single application of operator++, it is fairly easy to show that iterating through a tree storing $n$ nodes requires $O(n)$ operations overall.

**Exercise:** [method 1] Write a fragment of code that given a node, finds the in-order successor using parent pointers. Be sure to draw a picture to help you understand!

**Exercise:** [method 2] Write a fragment of code that given a tree iterator containing a pointer to the node and a stack of pointers representing path from root to node, finds the in-order successor (without using parent pointers).

**Exercise:** What are the advantages & disadvantages of each method?
// OUTPUT & PRINTING
friend std::ostream& operator<<(std::ostream& ostr, const ds_set<T>& s) {s.print_in_order(ostr, s.root_);ostr;}
void print_sideways_tree(std::ostream& ostr, const print_sideways_tree(ostr, root_, 0);}

// ITERATORS
iterator begin() const { print_sideways_tree(ostr,root_,0); return iterator(NULL); }
ext() const { return iterator(NULL); }

// REPRESENTATION
int TreeNode* root_;  ds_set<T>& old) { root_ = this->copy_tree(old.root_); }  ds_set() : root_(NULL), size_(0) {}  ds_set(const ds_set<T>& old) : size_(old.size_) { root_ = this->copy_tree(old.root_); }  ds_set(T& key_value, TreeNode* p, &parent(NULL)) { p = new TreeNode(key_value); this->size_++;

// FIND, INSERT & ERASE
iterator find(const T& key_value) { return find(key_value, root_); }
std::pair<iterator, bool> insert(T const& key_value) { return insert(key_value, root_); }
int erase(T const& key_value) { return erase(key_value, root_); }

// PRIVATE HELPER FUNCTIONS
TreeNode* copy_tree(TreeNode* old_root) { /* Implemented in Lab 9 */ }  void destroy_tree(TreeNode* p) { /* Implemented in Lecture 17 */ }