CSCI-1200 Data Structures — Spring 2018
Lecture 17 – Trees, Part II

Review from Lecture 16 and Lab 10

- 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`.

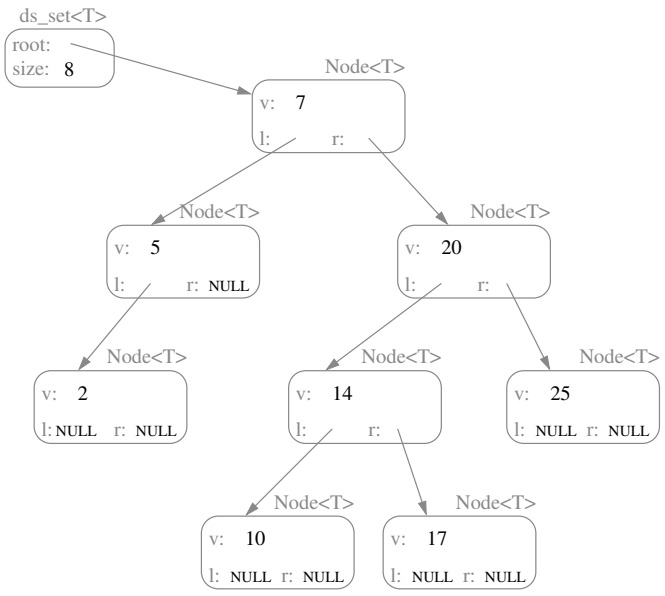
Today's Lecture

- 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 itertor 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)`
- Now let's write code to print out the elements in a binary tree in each of these three orders. These functions are easy to write recursively, and the code for the three functions looks amazingly similar. Here's the code for an in-order traversal to print the contents of a tree:

```
void print_in_order(ostream& ostr, const TreeNode<T>* p) {
  if (p) {
    print_in_order(ostr, p->left);
    ostr << p->value << "\n";
    print_in_order(ostr, p->right);
  }
}
```

- How would you modify this code to perform pre-order and post-order traversals?

- 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 an "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 trees 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.

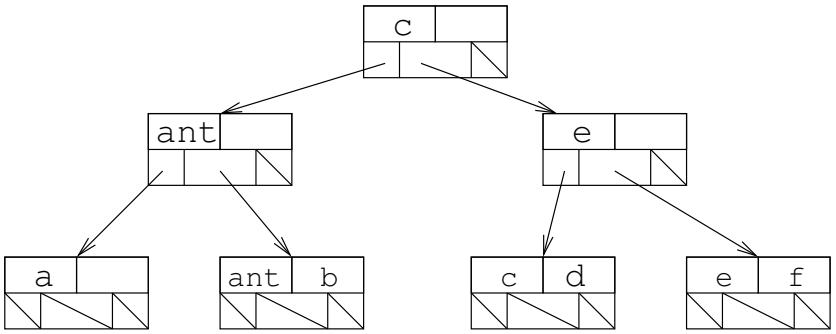
17.6 Limitations of Our BST Implementation

- The efficiency of the main insert, find and erase algorithms depends on the height of the tree.
- The best-case and average-case heights of a binary search tree storing n nodes are both $O(\log n)$. The worst-case, which often can happen in practice, is $O(n)$.
- Developing more sophisticated algorithms to avoid the worst-case behavior will be covered in Introduction to Algorithms. One elegant extension to the binary search tree is described below...

17.7 B+ Trees

- Unlike binary search trees, nodes in B+ trees (and their predecessor, the B tree) have up to b children. Thus B+ trees are very flat and very wide. This is good when it is very expensive to move from one node to another.
- B+ trees are supposed to be associative (i.e. they have key-value pairs), but we will just focus on the keys.
- Just like STL `map` and STL `set`, these *keys* and *values* can be any type, but *keys* must have an `operator<` defined.
- In a B tree value-key pairs can show up anywhere in the tree, in a B+ tree all the key-value pairs are in the leaves and the non-leaf nodes contain duplicates of some keys.
- In either type of tree, all leaves are the same distance from the root.
- The keys are always sorted in a B/B+ tree node, and there are up to $b-1$ of them. They act like $b-1$ binary search tree nodes mashed together.
- In fact, with the exception of the root, nodes will always have between roughly $\frac{b}{2}$ and $b-1$ keys (in our implementation).
- If a B+ tree node has k keys $key_0, key_1, key_2, \ldots, key_k$, it will have $k+1$ children. The keys in the leftmost child must be $< key_0$, the next child must have keys such that they are $\geq key_0$ and $< key_1$, and so on up to the rightmost child which has only keys $\geq key_k$.

- HW8 will focus on implementing some of the functionality of a B+ tree. It won't be enough to replace a real B+ tree, but it will be enough to understand how the tree works and construct trees.



- Note: "a" will come before "ant" lexicographically, in other words "a" < "ant"
- Considerations in a full implementation:
  - What happens when we want to add a key to a node that's already full?
  - How do we remove values from a node?
  - How do we ensure the tree stays balanced?
  - How to keep leaves linked together? Why would we want this?
  - How to represent key-value pairs?

Exercise: Draw a B+ tree with $b = 3$ with values inserted in the order $1, 2, 3, 4, 5, 6$. Now draw a B+ tree with $b = 3$ and values inserted in the order $6, 5, 4, 3, 2, 1$. Hint: The two trees have a different number of levels.

17.8 HW8 Hints

- You are *not* implementing a full B+ tree. Read the homework assignment carefully and keep it in mind if you look up videos/notes on B+ trees.
- You should put your implementation at the bottom of the .h file - do not change the forward declaration.
- Use the provided .h file - don't start from scratch. Since this is a templated class your entire implementation can go in the .h file.
- *find()* will only return NULL for an empty tree. Otherwise it will always return a leaf pointer. If the value is in the tree, the pointer should point to the leaf containing the value. If the value is not in the tree, the leaf should point to the last node visited in find(), which should be a leaf.
- You will **not** be graded on test cases that you write. You do **not** need to submit any test cases. We will only use your .h file and we will only read your .h file and README
- Don't try to use std::sort() to keep internal pointers sorted - because the nodes are templated, this creates a big mess. You're on your own if you want to pass a templated function to std::sort() as a 3^{rd} argument.
- *PrintSideways()* makes the split at $b/2$ nodes using integer division. Our tree printing functions use tabs (\t) instead of spaces.
- Compile with -Wall. Read all warnings. Fix all warnings. Read any compiler warnings on Submitty. Fix those warnings too.