Computer Science II — CSci 1200
Lecture 17 — Trees, Part 2

Review from Lecture 16

- Binary trees and binary search trees. They have a close tie to recursion.
- Tree nodes
- Basic tree algorithms: traversals
- Overview of tree implementation of `cs2set` class.
- Implementation of `begin` and `find`.

We will complete the discussion of `begin` and `find` in this lecture.

Review in more detail: cs2set class overview

- The classes are templated.
- There is an auxiliary `TreeNode` class
- The only member variables of the `cs2set` class are the root pointer and the size (number of tree nodes).
- The iterator class is declared internally, and is effectively a wrapper on `TreeNode` pointers.
  - Note that `operator*` returns a `const` reference because the keys may not be changed.
  - The increment and decrement operators are missing. These will be discussed more later in the lecture.
- The main public member functions just call a private (and often recursive) member function (passing the root pointer) that does all of the work.
- Because the class stores and manages dynamically allocated memory, it must provide a copy constructor, an `operator=`, and a destructor in order to work correctly.

Today's Lecture

- `cs2set` operations: insert, destroy, printing, erase
- Tree height
- Increment and decrement operations on iterators
- Limitations of our implementation

Insert

Major components of the insert algorithm:

- Move left and right down the tree based on comparing keys. The goal is to find the single location to do an insert *that preserves the binary search tree ordering property.*
- Inserting at an empty pointer location.
- 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.

Printing

There are two output methods:

- One outputs one key per line of output based on an in-order traversal.
- The second prints the tree sideways — rotated counter-clockwise by 90 degrees.
- This is accomplished by a "reversed" in-order traversal while keeping track of the tree height.
- We will look at a few examples in class to get a feel for how this works.

Exercise

Write the `destroy_tree` member function. This should effectively be a post-order traversal, with a node being destroyed after its left and right subtrees are destroyed.

Erase

- The first step is finding the node to remove
- Once it is found, the actual removal is easy if the node has no children or only one child.
- It is harder if there are two children. The trick is to
  - Find the node with the greatest value in the left subtree or the node with the smallest value in the right subtree.
  - The value in this node may be safely moved into the current node because of the tree ordering.
  - Then we recursively apply erase to remove that node — which is guaranteed to have at most one child.
- **Exercise:** Write a recursive version of erase.

Height and Height Calculation Algorithm

The following is not used in the `cs2set` class, but is an important exercise in understanding trees and recursion.

- The height of a node in a tree is the length of the longest path down the tree from the node to a leaf node.
  - The height of a leaf is therefore 0.
  - We will think of the height of a null pointer as -1.
- The height of the tree is the height of the root node, and therefore if the tree is empty the height will be -1.
- We will write a simple recursive algorithm in class to calculate the height of a tree.

Tree Iterators, Revisited

- 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 iterator maintains a list of pointers representing the path down the tree to the current node.
  - Each node stores a parent pointer. Only the root node has a null parent pointer.

  We will focus on the parent pointer version of the implementation.
- Implementing this requires that the `insert` and `erase` member functions correctly adjust parent pointers.
  - Handling all of the special cases involved is made easier if there is a dummy node at the root of the tree, just as with some linked-list implementations.
- We will focus our remaining discussion on the algorithm for finding the in-order successor of a node.
- Although this will look 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.

Limitations of Our BST Implementation

- The efficiency of the main insert, find and erase algorithms depends on the height of the tree (the height of the root).
- 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 $O(n)$. We will think about what causes this.
- Developing more sophisticated algorithms to avoid the worst-case behavior will be covered in Data Structures and Algorithms.