CSCI-1200 Data Structures — Fall 2009
Lecture 17 – Trees, Part II

Review from Lecture 16

- Binary trees and binary search trees. They have a close tie to recursion.
- We use auxiliary `TreeNode` and `tree_iterator` classes.
- The only member variables of the `ds_set` 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 can't change.
  - The increment and decrement operators are missing from the implementation but we will discuss how they could be added.
- 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.
- Last time we saw basic tree traversal algorithms and implemented `begin` and `find`.

Today's Lecture

- `ds_set` operations: insert, destroy, printing, erase
- Tree height
- Increment and decrement operations on iterators
- Limitations of our implementation
- Homework 8 Discussion

17.1 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.*
- 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.

17.2 Printing - Two different 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'll look at a few examples in class to get a feel for how this works.

17.3 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.

17.4 Erase

First we need to find 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:

- 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.

17.5 Height and Height Calculation Algorithm

- The height of a node in a tree is the length of the longest path down the tree from that 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.

Exercise: Write a simple recursive algorithm to calculate the height of a tree.

17.6 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 stack 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.
- If we choose the parent pointer method, we'll need to rewrite the `insert` and `erase` 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: Implement an algorithm for finding the in-order successor of a node.

17.7 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.

```
// ------------------------------------------------------------------
// TREE NODE CLASS
template <class T>
class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL) {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL) {}
  T value;
  TreeNode* left;
  TreeNode* right;
};

// ------------------------------------------------------------------
// TREE NODE ITERATOR CLASS
template <class T>
class tree_iterator {
public:
  tree_iterator() : ptr_(NULL) {}
  tree_iterator(TreeNode<T>* p) : ptr_(p) {}
  tree_iterator(const tree_iterator& old) : ptr_(old.ptr_) {}
  ~tree_iterator() {}
  tree_iterator& operator=(const tree_iterator& old) { ptr_ = old.ptr_;  return *this; }
  // operator* gives constant access to the value at the pointer
  const T& operator*() const { return ptr_->value; }
  // comparions operators are straightforward
  friend bool operator== (const tree_iterator& lft, const tree_iterator& rgt) { return lft.ptr_ == rgt.ptr_; }
  friend bool operator!= (const tree_iterator& lft, const tree_iterator& rgt) { return lft.ptr_ != rgt.ptr_; }
  // increment & decrement will be discussed in Lectures 17 & 18

private:
  // representation
  TreeNode<T>* ptr_;
};

// ------------------------------------------------------------------
// DS_SET CLASS
template <class T>
class ds_set {
public:
  ds_set() : root_(NULL), size_(0) {}
  ds_set(const ds_set<T>& old) : size_(old.size_) {   root_ = this->copy_tree(old.root_); }
  ~ds_set() {  this->destroy_tree(root_); }
  ds_set& operator=(const ds_set<T>& old) {
    if (old != *this) {
      this->destroy_tree(root_);
      root_ = this->copy_tree(old.root_);
      size_ = old.size_;
    }
    return *this;
  }
  typedef tree_iterator<T> iterator;
  int size() const { return size_; }
  bool operator==(const ds_set<T>& old) const { return (old.root_ == this->root_); }

  // 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_); }

  // OUTPUT & PRINTING
  friend std::ostream& operator<< (std::ostream& ostr, const ds_set<T>& s) {
    s.print_in_order(ostr, s.root_);
    return ostr;
  }
  void print_as_sideways_tree(std::ostream& ostr) const { print_as_sideways_tree(ostr, root_, 0); }
```

```
  // ITERATORS
  iterator begin() const {
    if (!root_) return iterator(NULL);
    TreeNode<T>* p = root_;
    while (p->left) p = p->left;
    return iterator(p);
  }
  iterator end() const { return iterator(NULL); }

private:
  // REPRESENTATION
  TreeNode<T>* root_;
  int size_;

  // PRIVATE HELPER FUNCTIONS
  TreeNode<T>*  copy_tree(TreeNode<T>* old_root) { /* Implemented in Lab 10 */ }
  void destroy_tree(TreeNode<T>* p) { /* Implemented in Lecture 17 */ }

  iterator find(const T& key_value, TreeNode<T>* p) {
    if (!p) return iterator(NULL);
    if (p->value > key_value)
      return find(key_value, p->left);
    else if (p->value < key_value)
      return find(key_value, p->right);
    else
      return iterator(p);
  }

  std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p) {
    if (!p) {
      p = new TreeNode<T>(key_value);
      this->size_++;
      return std::pair<iterator,bool>(iterator(p), true);
    }
    else if (key_value < p->value)
      return insert(key_value, p->left);
    else if (key_value > p->value)
      return insert(key_value, p->right);
    else
      return std::pair<iterator,bool>(iterator(p), false);
  }

  int erase(T const& key_value, TreeNode<T>* &p) {
    iterator itr = find(key_value);     // locate element to remove
    // Completed in Lecture 17
  }

  void print_in_order(std::ostream& ostr, const TreeNode<T>* p) const {
    if (p) {
      print_in_order(ostr, p->left);
      ostr << p->value << "\n";
      print_in_order(ostr, p->right);
    }
  }
  void print_as_sideways_tree(std::ostream& ostr, const TreeNode<T>* p, int depth) const {
    if (p) {
      print_as_sideways_tree(ostr, p->right, depth+1);
      for (int i=0; i<depth; ++i) ostr << "    ";
      ostr << p->value << "\n";
      print_as_sideways_tree(ostr, p->left, depth+1);
    }
  }
};
```