CSCI-1200 Data Structures — Fall 2010
Lecture 17 – Trees, Part II

Announcements

- Turing Award winner & MIT CS Professor Barbara Liskov will be giving a talk 1 week from today, Thursday Oct 11th @ 4pm in the CBIS Auditorium (the auditorium on the west end of the Biotech center).

  `http://www.cs.rpi.edu/news/flaherty/2010Liskov.pdf`

  Talk Title: The Power of Abstraction

  Talk Abstract: Abstraction is at the center of much work in Computer Science. It encompasses finding the right interface for a system as well as finding an effective design for a system implementation. Furthermore, abstraction is the basis for program construction, allowing programs to be built in a modular fashion. This talk will discuss how the abstraction mechanisms we use today came to be, how they are supported in programming languages, and some possible areas for future research.

Review from Lecture 16

- STL `set` container class (like STL `map`, but without the pairs!)
- Binary trees and binary search trees & recursion.
- Overview of the `ds_set` implementation and the `begin` and `find` operations.

Today's Lecture

- `ds_set` operations: insert, destroy, printing, erase
- Tree traversal order, tree height, & the increment and decrement operations on tree iterators
- Limitations of our `ds_set` implementation

17.1 `ds_set`: Class Overview

- There is an auxiliary `TreeNode` class, and a `tree_iterator` class. The classes are templated.
- The only member variables of the `ds_set` class are the root and the size (number of tree nodes).
- The iterator class is declared internally, and is effectively a wrapper on the TreeNode pointers.
  - Note that `operator*` returns a `const` reference because the keys can't change.
  - As just discussed the increment and decrement operators are missing.
- The main public member functions just call a private (and often recursive) member function (passing the root node) that does all of the work.
- Because the class stores and manages dynamically allocated memory, a copy constructor, `operator=`, and destructor must be provided.

17.2 Exercises

- Draw a diagram of a *possible* memory layout for a `ds_set` containing the numbers 16, 2, 8, 11, and 5.
- Is there only one valid memory layout for this data as a `ds_set`? Why?
- In what order should a forward iterator visit the data?
- Draw an *abstract* table representation of this data (omits details of `TreeNode` memory layout).

17.3 In-order, Pre-Order, Post-Order Traversal

- One of the fundamental tree operations is "traversing" the nodes in the tree and doing something at each node. The "doing something", which is often just printing, is referred to generically as "visiting" the node.
- There are three general orders in which binary trees are traversed: pre-order, in-order and post-order.
- **Exercises:** Draw an "exactly balanced" binary search tree with the elements 1-7.

  - What is the *in-order traversal* of this tree? Hint: it is monotonically increasing, which is always true for an in-order traversal of a binary search tree!
  - What is the *post-order traversal* of this tree? Hint, it ends with "4" and the 3rd element printed is "2".
  - What is the *pre-order traversal* of this tree? Hint, the last element is the same as the last element of thein-order traversal (but that is not true in general!)

- These are usually written 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);
  }
}
```

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

17.4 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.

17.5 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.

17.6 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.

17.7 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.

17.8 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 using parent pointers.

17.9 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);
    }
  }
};
```