CSCI-1200 Data Structures — Fall 2011
Lecture 17 – Trees, Part II

Review from Lecture 16

- STL `set` container class (like STL `map`, but without the pairs!)
- Overview of the `ds_set` implementation and the `begin` and `find` operations.
- In-order, pre-order, and post-order traversal; Breadth-first and depth-first tree search

Today's Lecture

- `ds_set` operations: insert, destroy, printing, erase; also tree height
- Finding the *in-order successor* of a binary tree node

17.1 `ds_set` Warmup/Review 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).

- 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.

17.2 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 inserting at an empty (NULL) pointer location. 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 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.

Exercise: How does the order that nodes are deleted affect the tree structure? Starting with a mostly balanced tree, give an erase ordering that yields an unbalanced tree.

17.4 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` 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?

```
// ------------------------------------------------------------------
// 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);
    }
  }
};
```