

CSCI-1200 Data Structures — Fall 2019

Lecture 16 – Trees, Part I

Review from Lecture 15

- Maps containing more complicated values. Example: index mapping words to the text line numbers on which they appear.
- Maps whose keys are class objects. Example: maintaining student records.
- Summary discussion of when to use maps.
- Lists vs. Graphs vs. Trees
- Intro to Binary Trees, Binary Search Trees, & Balanced Trees

Today's Lecture

- Finish Intro to Binary Trees, Binary Search Trees, & Balanced Trees
- STL `set` container class (like STL `map`, but without the pairs!)
- Implementation of `ds_set` class using binary search trees
- In-order, pre-order, and post-order traversal
- Breadth-first and depth-first tree search

16.1 Standard Library Sets

- STL sets are *ordered* containers storing unique “keys”. An ordering relation on the keys, which defaults to `operator<`, is necessary. Because STL sets are ordered, they are technically not traditional mathematical sets.
- Sets are like maps except they have only keys, there are no associated values. Like maps, the keys are **constant**. This means you can't change a key while it is in the set. You must remove it, change it, and then reinsert it.
- Access to items in sets is extremely fast! $O(\log n)$, just like maps.
- Like other containers, sets have the usual constructors as well as the `size` member function.

16.2 Set iterators

- Set iterators, similar to map iterators, are bidirectional: they allow you to step forward (`++`) and backward (`--`) through the set. Sets provide `begin()` and `end()` iterators to delimit the bounds of the set.
- Set iterators refer to const keys (as opposed to the pairs referred to by map iterators). For example, the following code outputs all strings in the set `words`:

```
for (set<string>::iterator p = words.begin(); p!= words.end(); ++p)
    cout << *p << endl;
```

16.3 Set insert

- There are two different versions of the `insert` member function. The first version inserts the entry into the set and returns a pair. The first component of the returned pair refers to the location in the set containing the entry. The second component is true if the entry wasn't already in the set and therefore was inserted. It is false otherwise. The second version also inserts the key if it is not already there. The iterator `pos` is a “hint” as to where to put it. This makes the insert faster if the hint is good.

```
pair<iterator,bool> set<Key>::insert(const Key& entry);
iterator set<Key>::insert(iterator pos, const Key& entry);
```

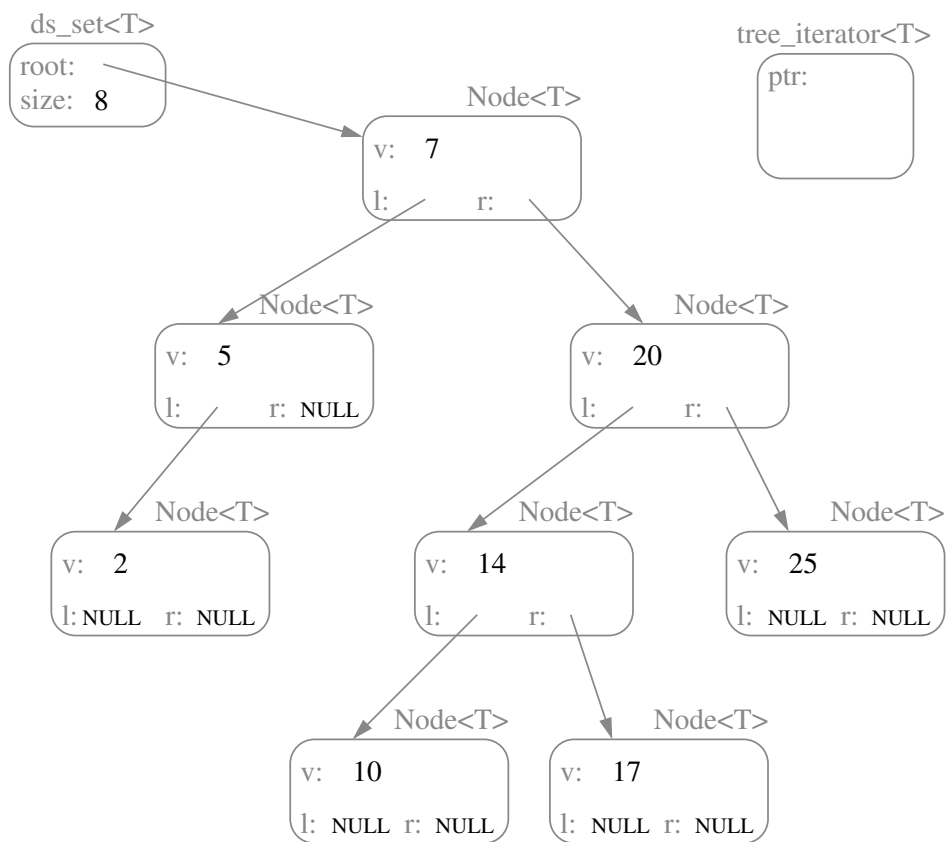
16.4 Set erase

- There are three versions of `erase`. The first `erase` returns the number of entries removed (either 0 or 1). The second and third erase functions are just like the corresponding erase functions for maps. Note that the `erase` functions do not return iterators. This is different from the `vector` and `list` erase functions.

```
size_type set<Key>::erase(const Key& x);
void set<Key>::erase(iterator p);
void set<Key>::erase(iterator first, iterator last);
```


16.9 ds_set: Class Overview

- There is two auxiliary classes, `TreeNode` and `tree_iterator`. All three 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.
 - The increment and decrement operators are missing (we'll fill this in next lecture!).
- 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.



- 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?

16.12 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 “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.
 - What is a breadth-first traversal of the elements in our sample binary search tree above? (We'll write and discuss code for breadth-first traversal next lecture!)

```

// Partial implementation of binary-tree based set class similar to std::set.
// The iterator increment & decrement operations have been omitted.
#ifdef ds_set_h
#define ds_set_h
#include <iostream>
#include <utility>
// -----
// 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;
};

template <class T> class ds_set;
// -----
// 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; }
    const T& operator*() const { return ptr_>value; }
    // operator* gives constant access to the value at the pointer
    // comparions operators are straightforward
    bool operator==(const tree_iterator& r) const { return ptr_ == r.ptr_; }
    bool operator!=(const tree_iterator& r) const { return ptr_ != r.ptr_; }
    // Increment & decrement will be discussed in Lecture 19 and Lab 10
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_); root_ = NULL; }
    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_sideways_tree(std::ostream& ostr) const { print_sideways_tree(ostr, root_, 0); }

// ITERATORS
iterator begin() const {
    // Implemented in Lecture 16
}

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 9 */ }
void destroy_tree(TreeNode<T>* p) { /* Implemented in Lecture 17 */ }

iterator find(const T& key_value, TreeNode<T>* p) {
    // Implemented in Lecture 16
}

std::pair<iterator, bool> insert(const T& key_value, TreeNode<T>* &p)
{ /* Discussed in Lecture 17 */ }
int erase(T const& key_value, TreeNode<T>* &p) { /* Implemented in Lecture 17 */ }
void print_in_order(std::ostream& ostr, const TreeNode<T>* p) const {
    // Discussed in Lecture 16
    if (p) {
        print_in_order(ostr, p->left);
        ostr << p->value << "\n";
        print_in_order(ostr, p->right);
    }
}

void print_sideways_tree(std::ostream& ostr, const TreeNode<T>* p, int depth) const {
    // Discussed in Lecture 17 */ }
};

#endif

```