

CSCI-1200 Computer Science II — Spring 2006

Lecture 21 — Trees, Part I

COURSE EVALUATIONS: Tuesday April 25th. Bring a #2 pencil.

Review from Lectures 20 & 21 — Linked Lists

- Singly-linked lists, doubly-linked lists, standard list operations
- Discuss `cs2list::erase`.

Today's Lecture — Binary Trees and Binary Search Trees

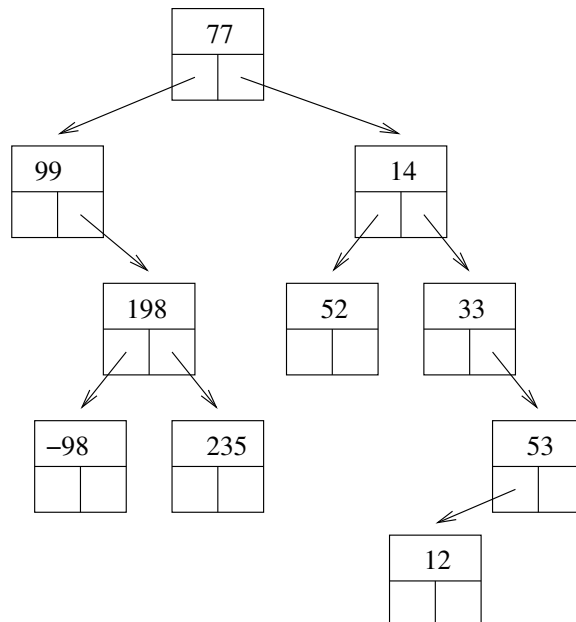
- Definition
- Basic operations
- Implementation of `cs2set` class using binary search trees
- Lots more tree stuff in CSCI 2300 Data Structures & Algorithms (DSA)!

22.1 Lists vs. Trees vs. Graphs

- Trees create a *hierarchical organization* of data, rather than the *linear organization* in linked lists (and arrays and vectors).
- Binary search trees are the mechanism underlying maps & sets (and multimaps & multisets).
- In HW8 you are building a graph with bi-directional edges to represent trains between cities. A tree is a special graph that has no cycles. The edges that connect nodes in trees and graphs may be *directed* or *undirected*.

22.2 Definition: Binary Trees

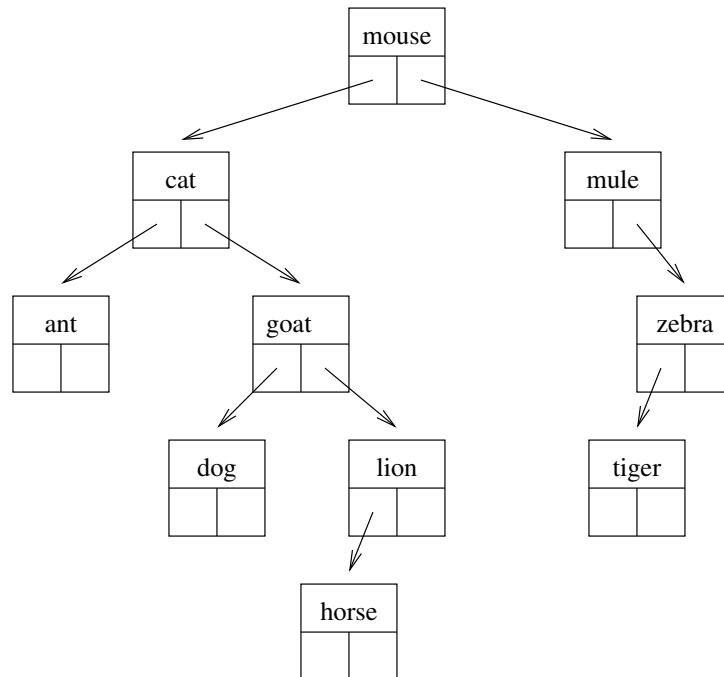
- A binary tree (strictly speaking, a “rooted binary tree”) is either empty or is a node that has pointers to two binary trees.
- Here's a picture of a binary tree storing integer values. In this figure, each large box indicates a tree node, with the top rectangle representing the value stored and the two lower boxes representing pointers. Pointers that are null are shown with an empty box.



- The topmost node in the tree is called the *root*.
- The pointers from each node are called *left* and *right*. The nodes they point to are referred to as that node's (left and right) *children*.
- The (sub)trees pointed to by the left and right pointers at *any* node are called the *left subtree* and *right subtree* of that node.
- A node where **both** children pointers are null is called a *leaf node*.
- A node's *parent* is the unique node that points to it. Only the root has no parent.

22.3 Definition: Binary Search Trees

- A *binary search tree* is a binary tree where **at each node** of the tree, the *value* stored at the node is
 - greater than or equal to all values stored in the left subtree, and
 - less than or equal to all values stored in the right subtree.
- Below is a picture of a binary search tree storing string values.



22.4 Exercise

Consider the following values:

4.5, 9.8, 3.5, 13.6, 19.2, 7.4, 11.7

1. Draw a binary tree with these values that *is NOT* a binary search tree.
2. Draw *two different* binary search trees with these values. Important note: This shows that the binary search tree structure for a given set of values is not unique!

22.5 The Tree Node Class

Here is the class definition for nodes in the tree. We will use this for the tree manipulation code we write.

```
template <class T>
class TreeNode {
public:
    TreeNode() : left(0), right(0) {}
    TreeNode(const T& init) : value(init), left(0), right(0) {}
    T value;
    TreeNode* left;
    TreeNode* right;
};
```

Sometimes a 3rd pointer — to the parent `TreeNode` — is added.

22.6 In 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.
- 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);
    }
}
```

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

22.7 Exercises

1. Write a templated function to find the smallest value stored in a binary search tree whose root node is pointed to by `p`.

2. Write a function to count the number of odd numbers stored in a binary tree (not necessarily a binary search tree) of integers. The function should accept a `TreeNode<int>` pointer as its sole argument and return an integer. Try to think recursively!


```

// Partial implementation of binary-tree based set class similar to std::set.
// The iterator increment & decrement operations have been omitted.
#ifndef cs2set_h_
#define cs2set_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 cs2set;

// -----
// 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; }
    // comparisons operators are straightforward
    friend bool operator==(const tree_iterator& l, const tree_iterator& r) { return l.ptr_ == r.ptr_; }
    friend bool operator!=(const tree_iterator& l, const tree_iterator& r) { return l.ptr_ != r.ptr_; }

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

// -----
// CS2 SET CLASS
template <class T>
class cs2set {
public:
    cs2set() : root_(NULL), size_(0) {}
    cs2set(const cs2set<T>& old) : size_(old.size_) {
        root_ = this->copy_tree(old.root_); }
    ~cs2set() { this->destroy_tree(root_); root_ = NULL; }
    cs2set& operator=(const cs2set<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 cs2set<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 cs2set<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 {
    // Implemented in Lecture 22

}
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 13 */ }
void destroy_tree(TreeNode<T>* p) { /* Implemented in Lecture 23 */ }

iterator find(const T& key_value, TreeNode<T>* p) {
    // Implemented in Lecture 22

}

std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p) { /* Discussed in Lecture 23 */ }
int erase(T const& key_value, TreeNode<T>* &p) { /* Implemented in Lecture 23 */ }

void print_in_order(std::ostream& ostr, const TreeNode<T>* p) const {
    // Discussed in Lecture 22
    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 {
    /* Discussed in Lecture 23 */ }
};

#endif

```