

Computer Science II — CSci 1200

Lecture 16 — Trees, Part 1

Test 2 Results

- Average (max of 90): 63.8
- Score ranges:

| Range | Num Tests |
|-------|-----------|
| 80-87 | 18 |
| 70-79 | 53 |
| 60-69 | 42 |
| 50-59 | 28 |
| < 50 | 25 |

- Curve: if s is your raw score out of 90, then your scaled score s' out of 100 is

$$s' = 1.05s + 9.54.$$

- If you believe your test was incorrectly graded, please write a note on the question(s) **and** on the front of the exam and resubmit. Your entire exam will be reconsidered.
- Please check all of your on-line grades for correctness. Contact your lab TA to correct any mistakes.

Today's Lecture — Binary Tree and Binary Search Trees

- Definition
- Basic operations
- Implementation of `cs2set` class using binary search trees

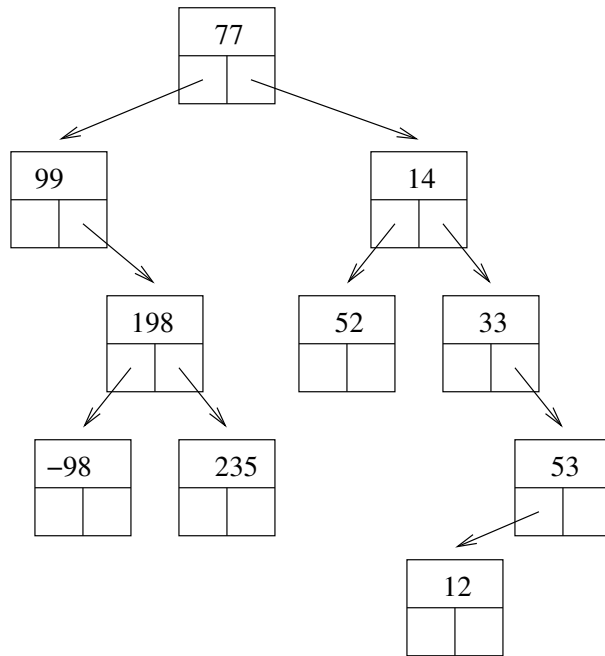
Binary Trees and Binary Search Trees

- Trees create a hierarchical organization of data, rather than the linear organization in linked lists. Trees are a special case of the graphs you investigated in HW 6.
- Binary search trees are the mechanism underlying maps, multimaps, sets and multi-sets.
- Many other problems can be addressed using trees.
 - One example is building data structures to represent spatial information.

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.

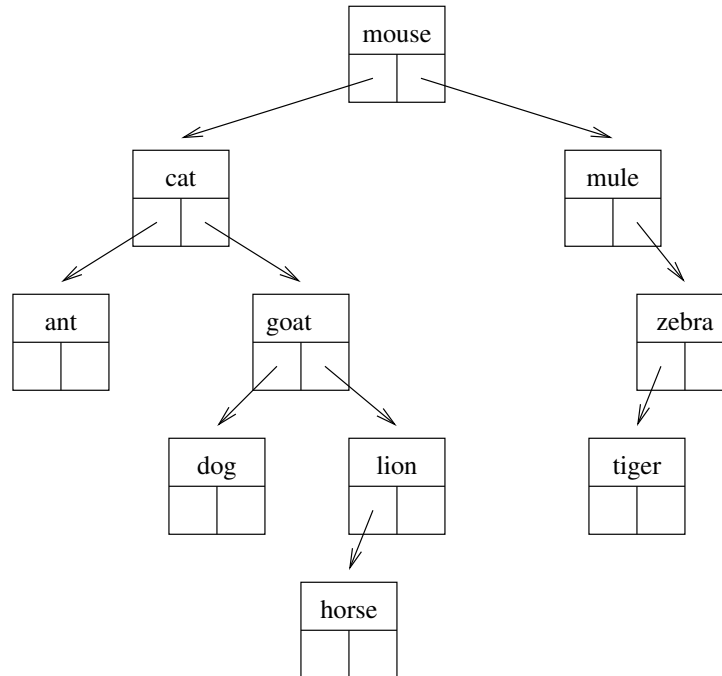


In this figure, each large box indicates a tree node, with the top rectangle representing the value stored and the two lower boxes representing pointers. Pointers that are null are shown with an empty box.

- 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.

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.



Exercise

Consider the following values:

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

1. Show these values placed in a binary tree that is NOT a binary search tree.
2. Show these values placed in two different binary search trees (all values in each tree).
This shows that the binary search tree structure for given set of values is not unique.

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.

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 );
    }
}
```

- We will look at this and discuss pre-order and post-order traversals in class.

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!

cs2set and Binary Search Tree Implementation

- A limited implementation of a set using a binary search tree is in the code attached.
 - Missing from this are the increment and decrement operations for iterators and the `erase` function. We will discuss both, but not in terms of their class implementation.
- We will use this as the basis both for understanding an initial selection of tree algorithms and for thinking about how standard library sets really work.

cs2set: Class Overview

- The classes are templated.
- There is an auxiliary `TreeNode` class
- The only member variables of the `cs2set` 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.
 - Node that `operator*` returns a `const` reference because the keys can not be changed while they are in the set.
 - 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.

Exercises

1. Provide the implementation of the member function `cs2set<T>::begin`. This is essentially the problem of finding the node in the tree that has stores the smallest value.
2. Write a recursive version of the function `find`.