CSCI-1200 Data Structures — Fall 2014
Lecture 16 – Trees, Part I

Review from Lectures 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.

Today's Lecture

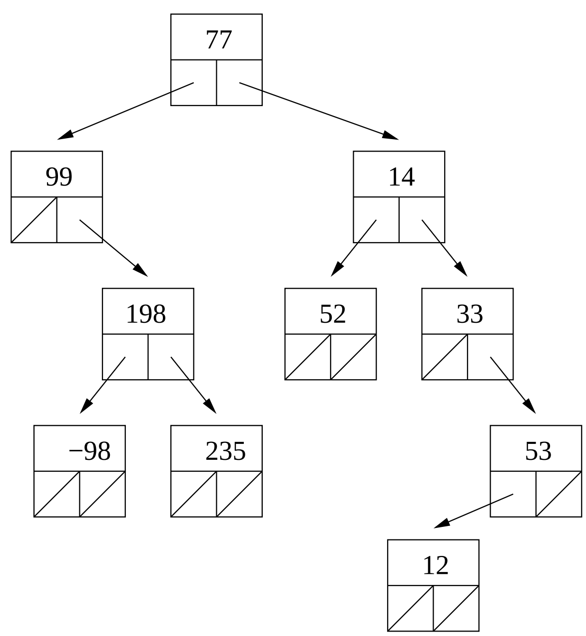
- STL `set` container class (like STL `map`, but without the pairs!)
- Binary Trees and Binary Search Trees
- Definition & basic operations
- Implementation of `ds_set` class using binary search trees
- In-order, pre-order, and post-order traversal
- Breadth-first and depth-first tree search

16.1 Overview: 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).
- Mathematically speaking: A *graph* is a set of vertices connected by edges. And a tree is a special graph that has no *cycles*. The edges that connect nodes in trees and graphs may be *directed* or *undirected*.

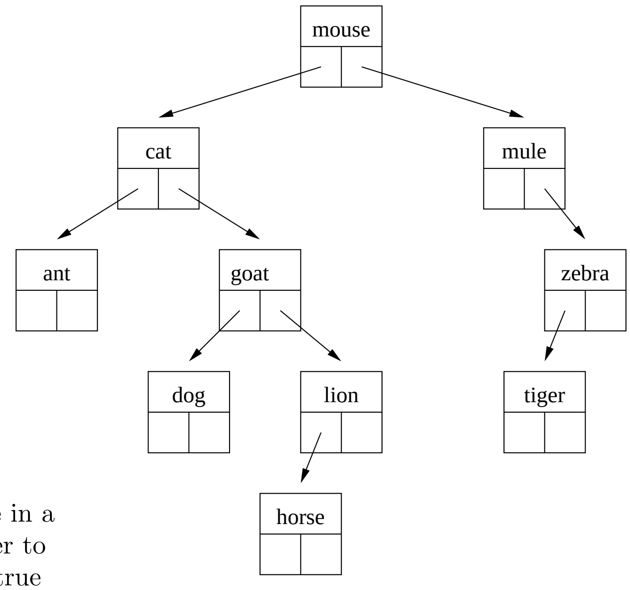
16.2 Definition: Binary Trees

- 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 a slash through the 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.



16.3 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.
- Here is a picture of a binary search tree storing string values.



16.4 Definition: Balanced Trees

- The number of nodes on each subtree of each node in a "balanced" tree is *approximately* the same. In order to be an *exactly* balanced binary tree, what must be true about the number of nodes in the tree?
- In order to claim the performance advantages of trees, we must assume and ensure that our data structure remains approximately balanced. (You'll see much more of this in Intro to Algorithms!)

16.5 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!

3. How many *exactly balanced* **binary search trees** exist with these numbers? How many *exactly balanced* **binary trees** exist with these numbers?

16.6 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.

16.7 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;
```

16.8 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);
```

16.9 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);
```

16.10 Set `find`

- The find function returns the `end` iterator if the key is not in the set:

```
const_iterator set<Key>::find(const Key& x) const;
```

16.11 Beginning our implementation of `ds_set`: 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(NULL), right(NULL) {}
    TreeNode(const T& init) : value(init), left(NULL), right(NULL) {}
    T value;
    TreeNode* left;
    TreeNode* right;
  };
```

- Note: Sometimes a 3rd pointer — to the parent TreeNode — is added.

16.12 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. Hint: think recursively!

16.13 `ds_set` and Binary Search Tree Implementation

- A partial implementation of a set using a binary search tree is in the code attached. We will continue to study this implementation in tomorrow's lab & the next lecture.
- The increment and decrement operations for iterators have been omitted from this implementation. Next lecture we will discuss a couple strategies for adding these operations.
- 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.

16.14 `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.

16.15 Exercises

1. Provide the implementation of the member function `ds_set<T>::begin`. This is essentially the problem of finding the node in the tree that stores the smallest value.

2. Write a recursive version of the function `find`.

16.16 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.
- In order to explain these, let's first 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 the in-order traversal (but that is not true in general! why not?)

- 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?

16.17 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.
#ifndef 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; }
  // operator* gives constant access to the value at the pointer
  const T& operator*() const { return ptr_->value; }
  // comparions operators are straightforward
  bool operator==(const tree_iterator& r) { return ptr_ == r.ptr_; }
  bool operator!=(const tree_iterator& r) { return ptr_ != r.ptr_; }
  // increment & decrement will be discussed in Lecture 17 and Lab 11

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_as_sideways_tree(std::ostream& ostr) const { print_as_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 10 */  }
  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_as_sideways_tree(std::ostream& ostr, const TreeNode<T>* p, int depth) const {
    /* Discussed in Lecture 17 */  }
};

#endif
```