CSCI-1200 Data Structures — Fall 2021
Lecture 17 – Trees, Part I

Announcements

- We are now collecting puzzles to use in the Homework 6 Contest.
  Optional contest puzzle submission deadline is Tuesday, November 2nd @ 11:59pm!
- Deadline for the Homework 6 Contest is Sunday, November 7th @ 11:59pm.
  Contest submission will be worth 2 points on the regular Homework 6 grade.

Review of Associative Containers (STL Maps) from Lecture 15 & 16

- Maps are an association between two types, one of which (the key) must have a `operator<` ordering on it.
- The association may be immediate:
  - Words and their counts.
  - Words and the lines on which they appear
- Or, the association may be created by splitting a type:
  - Splitting off the name (or student id) from rest of student record.

Today's Lecture

- STL `set` container class (like STL `map`, but without the pairs!)
- Lists vs. Graphs vs. Trees
- Intro to Binary Trees, Binary Search Trees, & Balanced Trees
- Implementation of `ds_set` class using binary search trees
- WARNING: Trees are complicated, and we will spend 4 total lectures discussing the implementation details of trees in preparation for Homework 8.

17.1 Standard Library Sets (STL Maps with only keys, no values)

- 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.

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

17.3 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);
```

17.4 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. Newer versions of STL specify that the `erase` functions return an iterator to the element after the removed element. *Which matches the behavior of the `vector` and `list` erase functions.*

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

17.5 Set `find`

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

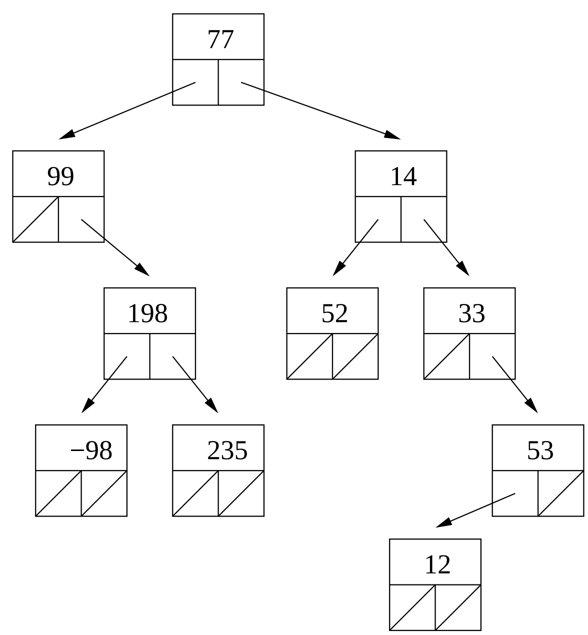
```
const_iterator set<Key>::find(const Key& x) const;
```

17.6 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*.

17.7 Definition: Binary Trees

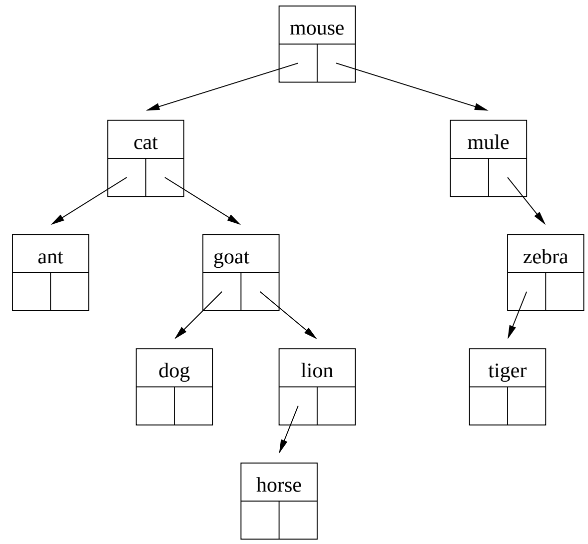
- A binary tree (strictly speaking, a "rooted binary tree") is either empty or is a node that has pointers to two binary trees.
- Here's a picture of a binary tree storing integer values. In this figure, each large box indicates a tree node, with the top rectangle representing the value stored and the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

17.8 Definition: Binary Search Trees

- A *tree* is a graph without cycles (loops). Exactly one path between every 2 nodes.
- Every node in a *binary tree* has 2 children (one or both might be NULL).
- 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.



17.9 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!)

17.10 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?

17.11 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.

17.12 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`.

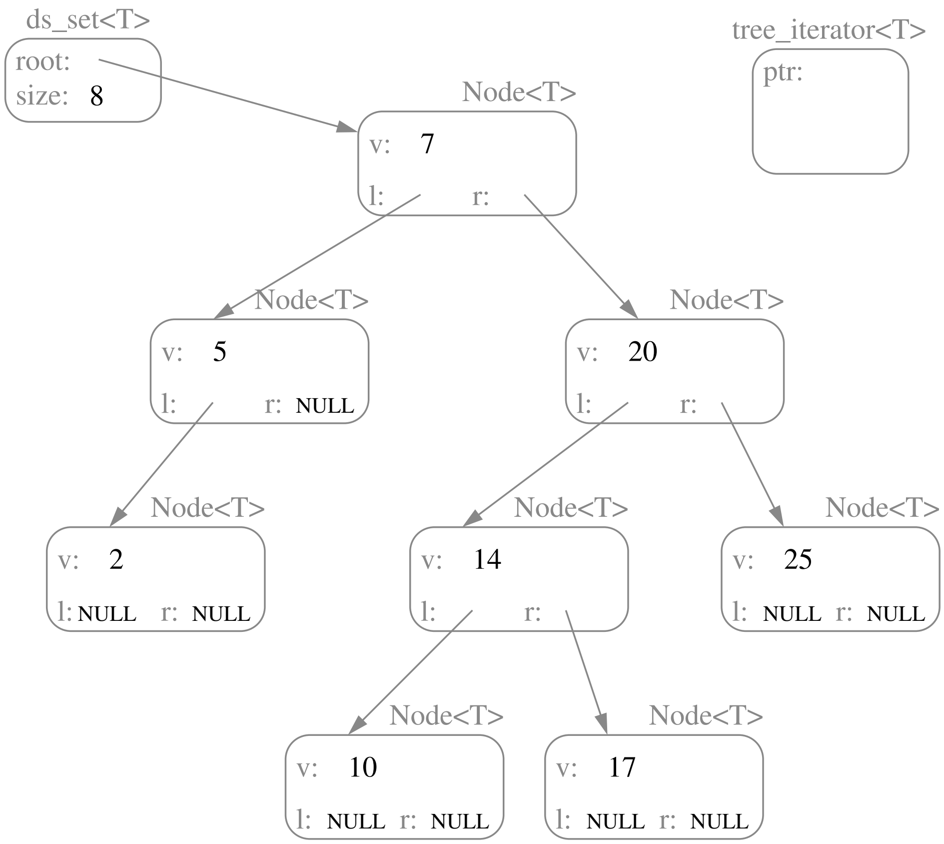
2. Write a function to count the number of odd numbers stored in a binary tree (not necessarily a binary search tree) of integers. The function should accept a `TreeNode<int>` pointer as its sole argument and return an integer. Hint: think recursively!

17.13 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.

17.14 `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.



17.15 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`.

```cpp
// 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 20 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 17
  }
  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 18 */  }

  iterator find(const T& key_value, TreeNode<T>* p) {
    // Implemented in Lecture 17
  }

  std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p)
  {  /* Discussed in Lecture 18 */  }
  int erase(T const& key_value, TreeNode<T>* &p) {  /* Implemented in Lecture 18 */  }
  void print_in_order(std::ostream& ostr, const TreeNode<T>* p) const {
    // Discussed in Lecture 17
    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 18 */  }
};

#endif
```