Review from Lecture 19 (& Lectures 17 & 18)

- General-Purpose Breadth-First, Tree Traversal Code

```cpp
void breadth_first_traverse( Node* root ) {
    int level = 0;
    std::vector<Node*> current_level;
    std::vector<Node*> next_level;
    if (root == NULL) { return; }
    current_level.push_back(root);
    while (current_level.size() != 0) {
        std::cout << "level " << level << " \":";
        for (unsigned int i = 0; i < current_level.size(); i++) {
            if (current_level[i]->left != NULL)
                next_level.push_back(current_level[i]->left);
            if (current_level[i]->right != NULL)
                next_level.push_back(current_level[i]->right);
            std::cout << " " << current_level[i]->value;
        }
        current_level = next_level;
        level++;
        next_level.clear();
        std::cout << std::endl;
    }
}
```

- BST / ds_set iterator increment (operator+)

- Every node stores Node parent pointer or
  iterator stores a vector of Node pointers (the path from root Node).

Today’s Lecture

- Some more practice exercises with trees & Big O Notation
- Limitations of our ds_set implementation, brief intro to red-black trees
- BONUS TOPIC: Template Specialization

20.1 Height and Height Calculation Algorithm

- The height of a node in a tree is the length of the longest path down the tree from that node to a leaf node. The height of a leaf is 1. We will think of the height of a null pointer as 0.

- The height of the tree is the height of the root node, and therefore if the tree is empty the height will be 0.

**Exercise:** Write a simple recursive algorithm to calculate the height of a tree.

- What is the best/average/worst-case running time of this algorithm? What is the best/average/worst-case memory usage of this algorithm? Give a specific example tree that illustrates each case.
20.2 Shortest Paths to Leaf Node

- Now let’s write a function to instead calculate the shortest path to a NULL child pointer.

- What is the running time of this algorithm? Can we do better? *Hint:* How does a breadth-first vs. depth-first algorithm for this problem compare?

20.3 A Practice Test Tree Problem

A *trinary tree* is similar to a binary tree except that each node has at most 3 children. Write a *recursive* function named `EqualsChildrenSum` that takes one argument, a pointer to the root of a trinary tree, and returns true if the value at each non-leaf node is the sum of the values of all of its children and false otherwise. In the examples below, the tree on the left will return true and the tree on the right will return false.

```cpp
class Node {
public:
  int value;
  Node* left;
  Node* middle;
  Node* right;
};
```

20.4 Limitations of Our BST Implementation

- The efficiency of the main insert, find and erase algorithms depends on the height of the tree.

- The best-case and average-case heights of a binary search tree storing $n$ nodes are both $O(\log n)$. The worst-case, which often can happen in practice, is $O(n)$.

- Developing more sophisticated algorithms to avoid the worst-case behavior will be covered in Introduction to Algorithms. One elegant extension to binary search tree is described below...
20.5 Red-Black Trees

In addition to the binary search tree properties, the following red-black tree properties are maintained throughout all modifications to the data structure:

1. Each node is either red or black.
2. The NULL child pointers are black.
3. Both children of every red node are black.
   Thus, the parent of a red node must also be black.
4. All paths from a particular node to a NULL child pointer contain the same number of black nodes.

What tree does our ds.set implementation produce if we insert the numbers 1-14 in order? The tree at the right is the result using a red-black tree. Notice how the tree is still quite balanced. Visit these links for an animation of the sequential insertion and re-balancing:

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
http://www.youtube.com/watch?v=vDHFF4wjWYU&noredirect=1

• What is the best/average/worst case height of a red-black tree with \( n \) nodes?
• What is the best/average/worst case shortest-path from root to leaf node in a red-black tree with \( n \) nodes?

20.6 Exercise \[ /6 \]

Fill in the tree on the right with the integers 1-7 to make a binary search tree. Also, color each node “red” or “black” so that the tree also fulfills the requirements of a Red-Black tree.

Draw two other red-black binary search trees with the values 1-7.

Note: Red-Black Trees are just one algorithm for self-balancing binary search tree. Others include: AVL trees, Splay Trees, (& more!).
20.7 BONUS TOPIC: Template Specialization Example

Writing templated functions is elegant and powerful, but sometimes we do not want to handle all types in exactly the same way. Sometimes we want to write different versions of the function depending on the type:

- Let’s study and discussion the following code:

```cpp
// We'll use this templated function (unless we find a specialized // implementation for our type)
template <class T>
void print_vec (const std::vector<T> &v) {
    std::cout << "count= " << v.size() << " data=";
    for (unsigned int i = 0; i < v.size(); i++) {
        std::cout << v[i];
    }
    std::cout << std::endl;
}

// This will match doubles (but not floats)
void print_vec (const std::vector<double> &v) {
    std::cout << "count= " << v.size() << " data=";
    for (unsigned int i = 0; i < v.size(); i++) {
        std::cout << std::setprecision(1) << std::fixed << v[i];
    }
    // unset the formatting
    std::cout << std::defaultfloat << std::endl;
}

int main() {
    // note: this syntax for initialization of vector contents is available with C++11
    std::vector<int> int_v = { 1, 2, 3, 4, 5 };  
    std::vector<double> double_v = { 1, 2, 3, 4, 5 };  
    std::vector<float> float_v = { 1, 2, 3, 4, 5 };  
    std::vector<std::string> string_v = { "1", "2", "3", "4", "5" };  
    print_vec(int_v);  
    print_vec(double_v);  
    print_vec(float_v);  
    print_vec(string_v);  
}

// This would match strings... but because it's placed after the // usage in main it's not used!?!?!
void print_vec (const std::vector<std::string> &v) {
    std::cout << "count= " << v.size() << " data=";
    for (unsigned int i = 0; i < v.size(); i++) {
        std::cout << "\"" << v[i] << "\"";
    }
    std::cout << std::endl;
}

- If we commented out the specialized implementations of print_vec for the double and string types:

  count=5  data= 1 2 3 4 5  
  count=5  data= 1 2 3 4 5  
  count=5  data= 1 2 3 4 5  
  count=5  data= 1 2 3 4 5  

- If we run the original code:

  count=5  data= 1 2 3 4 5  
  count=5  data= 1 2 3 4 5  
  count=5  data= 1 2 3 4 5  
  count=5  data= 1.0 2.0 3.0 4.0 5.0  
  count=5  data= 1 2 3 4 5  
  count=5  data= 1 2 3 4 5  

- If we swap the order of the main function and the string version of print_vec:

  count=5  data= 1 2 3 4 5  
  count=5  data= 1.0 2.0 3.0 4.0 5.0  
  count=5  data= 1 2 3 4 5  
  count=5  data= 1.0 2.0 3.0 4.0 5.0  
  count=5  data= 1 2 3 4 5  
  count=5  data= "1" "2" "3" "4" "5"
```cpp
// ITERATORS
iterator begin() const
if (!root_) return iterator(NULL, this);
TreeNode* p = root_; while (p->left) p = p->left;
return iterator(p, this);
}
iterator end() const {
return iterator(NULL, this); }  

private:
// REPRESENTATION
TreeNode* root_;  
// PRIVATE HELPER FUNCTIONS
TreeNode* copy_tree(TreeNode* old_root, TreeNode* the_parent) {
return NULL;
TreeNode *answer = new TreeNode();    answer->value = old_root->value;    answer->left = copy_tree(old_root->left,answer);    answer->right = copy_tree(old_root->right,answer);    answer->parent = the_parent;
}
void destroy_tree(TreeNode* p) {
if (p) { destroy_tree(p->left); destroy_tree(p->right); delete p; }}

// TREE NODE CLASS
class TreeNode {
int value;    TreeNode* left;    TreeNode* right;
TreeNode() : left(NULL), right(NULL), parent(NULL) {}    TreeNode(const T& init) : value(init), left(NULL), right(NULL), parent(NULL) {}    TreeNode& operator=(const TreeNode&) const { return *this; }  

public:
// CONSTRUCTORS, ASSIGNMENT OPERATOR, DESTRUCTOR
ds_set() : root_(NULL), size_(0) {}  
~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_,NULL);      size_ = old.size_;    }
}
int size() const { return size_; }  
bool operator==(const ds_set<T>& other) const { return (old.root_ == other.root_); }

// FIND, INSERT & ERASE
iterator find(const T& key_value) { return find(key_value, root_); }    int erase(T const& key_value, TreeNode* &p) {
/* implemented in Lecture 20 */
}
std::pair<iterator,bool> insert(const T& key_value, TreeNode* &p, TreeNode* the_parent) {
if (!p) {
    p = new TreeNode(key_value);      p->parent = the_parent;      this->size_++;
    return std::pair<iterator,bool>(iterator(p,this), true);  }
else if (p->value > key_value) return insert(key_value, p->left, p);
else if (p->value < key_value) return insert(key_value, p->right, p);
else
    return iterator(p, this);
}

void destroy_tree(TreeNode* p) {
if (p) { destroy_tree(p->left); destroy_tree(p->right); delete p; }}

// one way to allow implementation of iterator increment & decrement
TreeNode* parent;
// comparisons operators are straightforward
bool operator==(const iterator& rgt) { return ptr_ == rgt.ptr_; }
bool operator!=(const iterator& rgt) { return ptr_ != rgt.ptr_; }

// increment & decrement operators
iterator & operator++() { iterator temp(*this); ++(*this);    return temp; }
iterator& operator--() { iterator temp(*this); --(*this);    return temp; }

private:
// representation
TreeNode* ptr_;    const ds_set* set_;  

// ITERATORS
iterator begin() const {
if (!root_) return iterator(NULL, this);
TreeNode* p = root_; while (p->left) p = p->left;
return iterator(p, this);
}
iterator end() const { return iterator(NULL, this); }  

// TREE NODE CLASS
class TreeNode {
int value;    TreeNode* left;    TreeNode* right;
TreeNode() : left(NULL), right(NULL), parent(NULL) {}    TreeNode(const T& init) : value(init), left(NULL), right(NULL), parent(NULL) {}    TreeNode& operator=(const TreeNode&) const { return *this; }  

public:
// CONSTRUCTORS, ASSIGNMENT OPERATOR, DESTRUCTOR
ds_set() : root_(NULL), size_(0) {}  
~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_,NULL);      size_ = old.size_;    }
}
int size() const { return size_; }  
bool operator==(const ds_set<T>& other) const { return (old.root_ == other.root_); }

// FIND, INSERT & ERASE
iterator find(const T& key_value) { return find(key_value, root_); }    int erase(T const& key_value, TreeNode* &p) {
/* implemented in Lecture 20 */
}
std::pair<iterator,bool> insert(const T& key_value, TreeNode* &p, TreeNode* the_parent) {
if (!p) {
    p = new TreeNode(key_value);      p->parent = the_parent;      this->size_++;
    return std::pair<iterator,bool>(iterator(p,this), true);  }
else if (p->value > key_value) return insert(key_value, p->left, p);
else if (p->value < key_value) return insert(key_value, p->right, p);
else
    return iterator(p, this);
}

void destroy_tree(TreeNode* p) {
if (p) { destroy_tree(p->left); destroy_tree(p->right); delete p; }}

// one way to allow implementation of iterator increment & decrement
TreeNode* parent;
// comparisons operators are straightforward
bool operator==(const iterator& rgt) { return ptr_ == rgt.ptr_; }
bool operator!=(const iterator& rgt) { return ptr_ != rgt.ptr_; }

// increment & decrement operators
iterator & operator++() { iterator temp(*this); ++(*this);    return temp; }
iterator& operator--() { iterator temp(*this); --(*this);    return temp; }

private:
// representation
TreeNode* ptr_;    const ds_set* set_;  

// ITERATORS
iterator begin() const {
if (!root_) return iterator(NULL, this);
TreeNode* p = root_; while (p->left) p = p->left;
return iterator(p, this);
}
iterator end() const { return iterator(NULL, this); }  

// TREE NODE CLASS
class TreeNode {
int value;    TreeNode* left;    TreeNode* right;
TreeNode() : left(NULL), right(NULL), parent(NULL) {}    TreeNode(const T& init) : value(init), left(NULL), right(NULL), parent(NULL) {}    TreeNode& operator=(const TreeNode&) const { return *this; }  
```

```cpp
```