Review from Lecture 19 (& Lectures 17 & 18)

- Template Specialization
- 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 ":"; "
        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;
    }
}
```

- Discussed how to `erase` items from a BST (incomplete)

Today’s Lecture

- Some more practice exercises with trees & Big O Notation
- Implement `erase` from a `ds_set`
- Limitations of our `ds_set` implementation, brief intro to red-black trees

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 Erase

First we need to find the node to remove. Once it is found, the actual removal is easy if the node has no children or only one child. Draw picture of each case!

<table>
<thead>
<tr>
<th>no children</th>
<th>only a left child</th>
<th>only a right child</th>
</tr>
</thead>
<tbody>
<tr>
<td>(with potentially a big subtree)</td>
<td>(with potentially a big subtree)</td>
<td></td>
</tr>
</tbody>
</table>
It is harder if there are two children:

- Find the node with the greatest value in the left subtree or the node with the smallest value in the right subtree.
- The value in this node may be safely moved into the current node because of the tree ordering.
- Then we recursively apply erase to remove that node — which is guaranteed to have at most one child.

**Exercise:** Write a recursive version of erase.
*Note: ignore parent pointers initially!*

---

**Exercise:** How does the order that nodes are deleted affect the tree structure? Starting with a mostly balanced tree, give an erase ordering that yields an unbalanced tree.

---

### 20.4 A Note about Parent Pointers...

- If we choose to implement the iterators using parent pointers, we will need to:
  - add the parent to the Node representation
  - revise `insert` to set parent pointers (see attached code)
  - revise `copy_tree` to set parent pointers (see attached code)
  - revise `erase` to update with parent pointers

### 20.5 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...
20.6 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?

20.7 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!).
20.8 More Tree Exercises

1. Draw a diagram of a possible memory layout for a ds_set containing the numbers 16, 2, 8, 11, and 5.

- Is there only one valid memory layout for this data as a ds_set? Why?

- In what order should a forward iterator visit the data? Draw an abstract table representation of this data. (This is the “user of STL map” diagram of the data, which omits details of TreeNode memory layout).

2. Draw a balanced binary tree that contains the values: 6, 13, 9, 17, 32, 23, and 20.

3. What is the height of a balanced binary tree storing \( n \) elements?

4. Draw a binary search tree that has post-order traversal: 6 13 9 17 32 23 20.

How many other correct answers are possible for the previous question?

20.9 Yet Another 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;
};
```
// ITERATORS
iterator begin() const {
    if (!root_) return iterator(NULL,this);
    while (p->left) p = p->left;
    TreeNode* p = root_;  
    return iterator(p,this);  
}
  iterator end() const { return iterator(NULL,this); }
private:
  // PRIVATE REPRESENTATION
  TreeNode* root_;  
  int size_;  
  // PRIVATE HELPER FUNCTIONS
  TreeNode* copy_tree(TreeNode* old_root, TreeNode* the_parent) {
    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;
    return answer;
  }
  void destroy_tree(TreeNode* p) {
    if (p) {
      destroy_tree(p->left); destroy_tree(p->right); delete p;  
    }
  }
  iterator find(const T& key_value, TreeNode* p) {
    if (!p) return end();
    if (p->value > key_value) return find(key_value, p->left);
    else if (p->value < key_value) return find(key_value, p->right);
    else return iterator(p,this);
  }
  std::pair<iterator,bool> insert(const T& key_value, TreeNode*& p, TreeNode* the_parent) {
    if (&old != this) {
      this->destroy_tree(root_);      root_ = this->copy_tree(old_root_,NULL);      size_ = old.size_;    }
    return std::pair<iterator,bool>(iterator(p,this), true);
  }
  int erase(T const& key_value, TreeNode* &p) { /* Implemented in Lecture 20 */
    if (p) {
      if (key_value < p->value) {
        return insert(key_value, p->right, p);
      } else if (key_value > p->value) {
        return insert(key_value, p->left, p);
      } else if (key_value == p->value) {
        return insert(key_value, p->right, p);
      } else return iterator(p,this);
    }
  }
public:
  // Template
  template <class T>
  class ds_set {
    // CONSTRUCTORS, ASSIGNMENT OPERATOR, DESTRUCTOR
  ds_set() : root_(NULL), size_(0) {}  
  ds_set(const ds_set<T>& old) : size_(old.size_) { root_ = 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_,NULL);      size_ = old.size_;    }
  
    return *this;
  }  
  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_,NULL);}  
  int erase(T const& key_value) { return erase(key_value, root_); }  

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

// DS_SET CLASS -- WITH NESTED TREE NODE & TREE ITERATOR CLASSES (ALTERNATE STYLE)
}