Review from Lecture 17 & 18 and Lab 10

• Definitions & Drawing: Trees, Binary Trees, Binary Search Trees, Balanced Trees, etc.
• Overview of the ds_set implementation
• begin, find, destroy_tree, insert
• In-order, pre-order, and post-order traversal; Breadth-first and depth-first tree search

Today’s Lecture

• ds_set & BST warmup exercises
• Implementation of a breadth-first tree traversal
• Iterator increment/decrement implementation, a.k.a. finding the in order successor to a node: add parent pointers – or – add a list/vector/stack of pointers to the iterator.
• Last piece of ds_set: removing an item, erase
• Tree height, longest-shortest paths – choice of depth-first vs. breadth-first search
• To support increment/decrement: Copy tree, Insert, and Erase with parent pointers

19.1 ds_set Warmup/Review Exercises

• 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 set/map” diagram of the data, which omits details of BST/TreeNode memory layout).
19.2 General-Purpose Breadth-First Search/Tree Traversal

- Write an algorithm to print the nodes in the tree one tier at a time, that is, in a breadth-first manner.

- 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.

19.3 Tree Iterator Increment/Decrement - Implementation Choices

- The increment operator should change the iterator’s pointer to point to the next TreeNode in an in-order traversal — the “in-order successor” — while the decrement operator should change the iterator’s pointer to point to the “in-order predecessor”.

- Unlike the situation with lists and vectors, these predecessors and successors are not necessarily “nearby” (either in physical memory or by following a link) in the tree, as examples we draw in class will illustrate.

- There are two common solution approaches:
  - Each node stores a parent pointer. Only the root node has a null parent pointer. [method 1]
  - Each iterator maintains a stack of pointers representing the path down the tree to the current node. [method 2]

- If we choose the parent pointer method, we’ll need to rewrite the insert and erase member functions to correctly adjust parent pointers.

- Although iterator increment looks expensive in the worst case for a single application of operator++, it is fairly easy to show that iterating through a tree storing n nodes requires $O(n)$ operations overall.
Exercise: [method 1] Write a fragment of code that given a node, finds the in-order successor using parent pointers. Be sure to draw a picture to help you understand!

Either version can be extended to complete the implementation of increment/decrement for the `ds_set` tree iterators.

Exercise: What are the advantages & disadvantages of each method?

Exercise: [method 2] Write a fragment of code that given a tree iterator containing a pointer to the node and a stack of pointers representing path from root to node, finds the in-order successor (without using parent pointers).
19.4 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!

- **no children**
- **only a left child** (with potentially a big subtree)
- **only a right child** (with potentially a big subtree)

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.
19.5 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.

19.6 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?*

19.7 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
// 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:
// REPRESENTATION
int size_; TreeNode* root_;  // PRIVATE HELPER FUNCTIONS
TreeNode* copy_tree(TreeNode* old_root, TreeNode* the_parent) {
    return NULL; return answer;
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; }
}  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); }
public:
// CONSTRUCTORS, ASSIGNMENT OPERATOR, DESTRUCTOR
ds_set() : root_(NULL), size_(0) {}
ds_set<T>& old) : root_(NULL), size_(0) { root_=this->copy_tree(old.root_,NULL); } }
""'
iterator operator++(int) { iterator temp(*this); ++(*this); return temp; } iterator operator--(int) { iterator temp(*this); --(*this); return temp; }
private:
// representation
TreeNode* ptr_;
const ds_set* set_;  // one way to allow implementation of iterator increment & decrement
TreeNode* parent;  }{