CSCI-1200 Data Structures — Fall 2016
Lecture 19 – Trees, Part II

Review from Lecture 18 and Lab 10

- Binary Trees, Binary Search Trees, & Balanced Trees
- STL `set` container class (like STL `map`, but without the pairs!)
- Finding the smallest element in a BST.
- Overview of the `ds_set` implementation: `begin` and `find`.

Today's Lecture

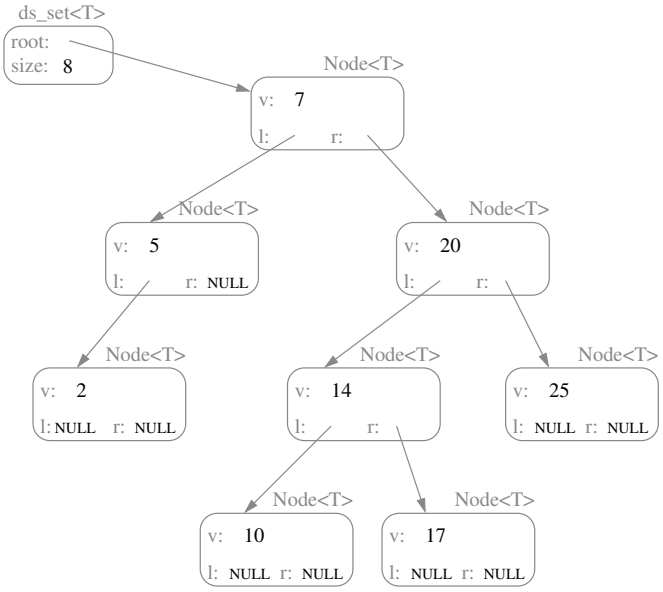
- Warmup / Review: `destroy_tree`
- A very important `ds_set` operation insert
- In-order, pre-order, and post-order traversal; Breadth-first and depth-first tree search
- Finding the *in-order successor* of a binary tree node, tree itertor increment

19.1 Warmup Exercise

- Write the `ds_set::destroy_tree` private helper function.

19.2 Insert

- Move left and right down the tree based on comparing keys. The goal is to find the location to do an insert *that preserves the binary search tree ordering property.*
- We will always be inserting at an empty (NULL) pointer location.
- **Exercise:** Why does this work? Is there always a place to put the new item? Is there ever more than one place to put the new item?



- IMPORTANT NOTE: Passing pointers by reference ensures that the new node is truly inserted into the tree. This is subtle but important.
- Note how the return value pair is constructed.
- **Exercise:** How does the order that the nodes are inserted affect the final tree structure? Give an ordering that produces a balanced tree and an insertion ordering that produces a highly unbalanced tree.

19.3 In-order, Pre-order, Post-order Traversal

- Reminder: For an exactly balanced binary search tree with the elements 1-7:
  - In-order: `1 2 3  (4)  5 6 7`
  - Pre-order: `(4)  2 1 3  6 5 7`
  - Post-order: `1 3 2  5 7 6  (4)`
- 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?

- What is the traversal order of the `destroy_tree` function we wrote earlier?

19.4 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.

    Note: A simple recursive tree function is usually a depth-first traversal.
  - What is a breadth-first traversal of the elements in our sample binary search trees above?

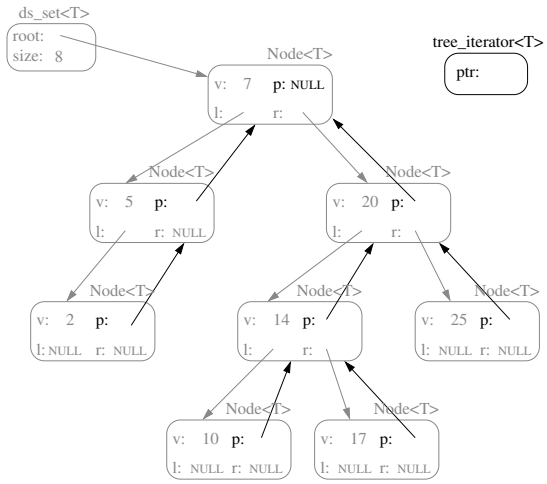
19.5 General-Purpose Breadth-First Search/Tree Traversal

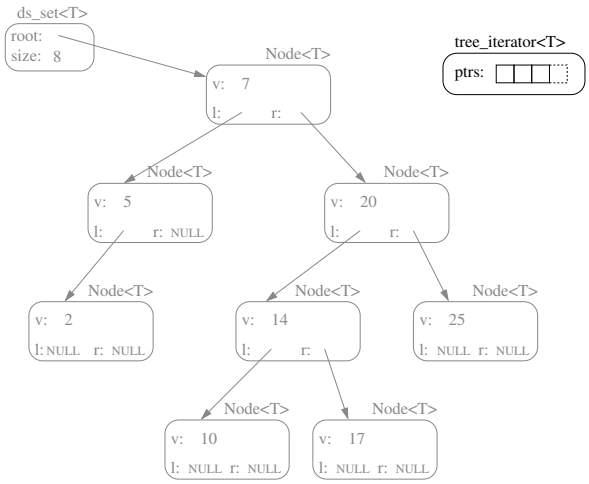
- Write an algorithm to print the nodes in the tree one tier at a time, that is, in a *breadth-first* manner.

- 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.

19.6 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` (*which we'll write next lecture!*) 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!

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).

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

```
// ------------------------------------------------------------------
// TREE NODE CLASS
template <class T>
class TreeNode {
public:
  TreeNode() : left(NULL), right(NULL)/*, parent(NULL)*/ {}
  TreeNode(const T& init) : value(init), left(NULL), right(NULL)/*, parent(NULL)*/ {}
  T value;
  TreeNode* left;
  TreeNode* right;
  // one way to allow implementation of iterator increment & decrement
  // TreeNode* parent;
};

// ------------------------------------------------------------------
// 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& rgt) { return ptr_ == rgt.ptr_; }
  bool operator!= (const tree_iterator& rgt) { return ptr_ != rgt.ptr_; }
  // increment & decrement operators
  tree_iterator<T> & operator++() { /* discussed & implemented in Lecture 19 */









    return *this;
  }
  tree_iterator<T> operator++(int) {  tree_iterator<T> temp(*this);  ++(*this);  return temp;  }
  tree_iterator<T> & operator--() { /* implementation omitted */ }
  tree_iterator<T> operator--(int) {  tree_iterator<T> temp(*this);  --(*this);  return temp;  }

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_,NULL); }
  ~ds_set() { this->destroy_tree(root_); root_ = NULL; }
  ds_set& operator=(const ds_set<T>& old) { /* implementation omitted */ }

  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;
  }

  // ITERATORS
  iterator begin() const {
    if (!root_) return iterator(NULL);
    TreeNode<T>* p = root_;
    while (p->left) p = p->left;
    return iterator(p);
  }
  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 19 */





  }

  iterator find(const T& key_value, TreeNode<T>* p) {  /* Implemented in Lecture 18 */  }

  std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p) {
    // NOTE: will need revision to support & maintain parent pointers
    if (!p) {
      p = new TreeNode<T>(key_value);
      this->size_++;
      return std::pair<iterator,bool>(iterator(p), true);
    }
    else if (key_value < p->value)
      return insert(key_value, p->left);
    else if (key_value > p->value)
      return insert(key_value, p->right);
    else
      return std::pair<iterator,bool>(iterator(p), false);
  }

  int erase(T const& key_value, TreeNode<T>* &p) { /* Implemented in Lecture 20 */ }

  void print_in_order(std::ostream& ostr, const TreeNode<T>* p) const {
    if (p) {
      print_in_order(ostr, p->left);
      ostr << p->value << "\n";
      print_in_order(ostr, p->right);
    }
  }
};
```