CSCI-1200 Data Structures — Fall 2021
Lecture 18 – Trees, Part II

Review from Lecture 17

- 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.

Today's Lecture

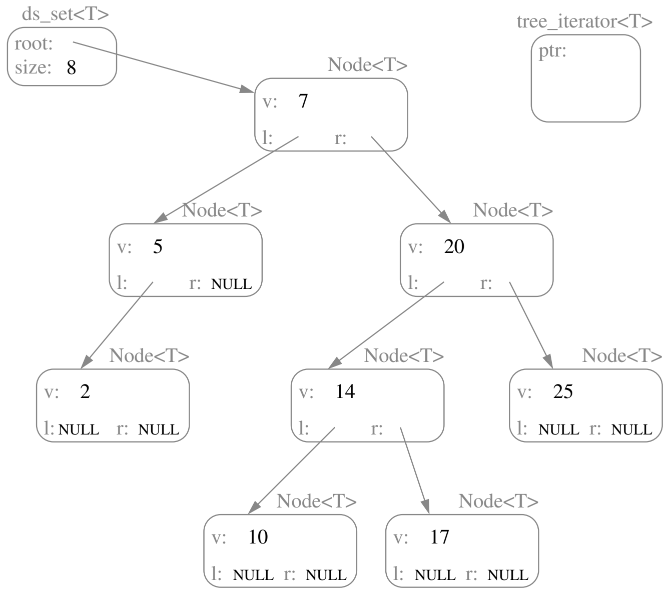
- Overview of the `ds_set` implementation
- Exercises: `begin` and `find` and `destroy_tree`
- A very important `ds_set` operation: `insert`
- In-order, pre-order, and post-order traversal
- Breadth-first and depth-first tree search
- ... and more Big O Notation practice!

18.1 `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 couple lectures.
- 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.

18.2 `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.



18.3 Exercises

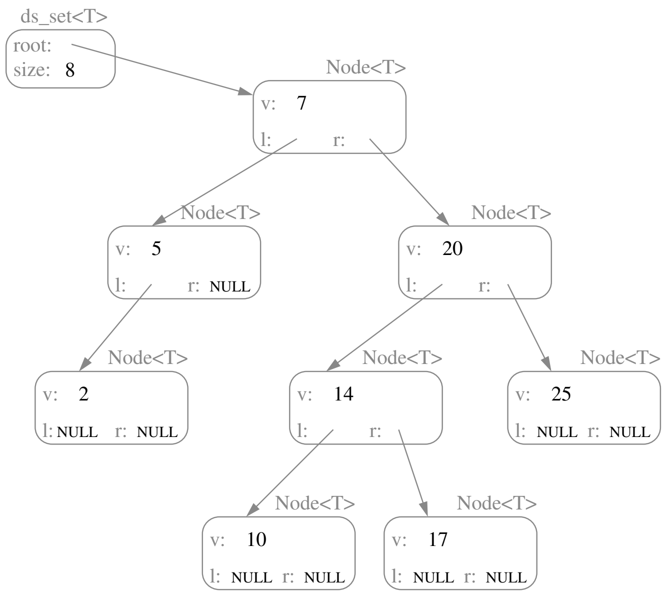
1. Provide the implementation of the member function `ds_set<T>::begin`. This is essentially the problem of finding the node in the tree that stores the smallest value.

2. Write a recursive version of the function `find`.

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

18.4 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.

18.5 In-order, Pre-order, and Post-order Traversal

- One of the fundamental tree operations is "traversing" the nodes in the tree and doing something at each node. The "doing something", which is often just printing, is referred to generically as "visiting" the node.
- There are three general orders in which binary trees are traversed: pre-order, in-order and post-order.
- These are usually written 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);
  }
}
```

- Draw an exactly balanced binary search tree with the elements 1-7:
- The traversals for tree you just drew are:
  - 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 modify the `print` function above to perform pre-order and post-order traversals.
- What is the traversal order of the `destroy_tree` function we wrote earlier?

18.6 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 tree above?

18.7 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.

```
// -------------------------------------------------------------------
// DS_SET CLASS -- WITH NESTED TREE NODE & TREE ITERATOR CLASSES (ALTERNATE STYLE)
template <class T>
class ds_set {
public:

  // -------------------------------------------------------------------
  // TREE NODE CLASS
  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
  class iterator {
  public:
    iterator() : ptr_(NULL) {}
    iterator(TreeNode* p) : ptr_(p) {}
    iterator& operator=(const 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 iterator& rgt) { return ptr_ == rgt.ptr_; }
    bool operator!= (const iterator& rgt) { return ptr_ != rgt.ptr_; }
    iterator & operator++() { /* discussed & implemented in Lecture 18 & 19 */










      return *this;
    }
    iterator operator++(int) { iterator temp(*this); ++(*this); return temp; }
    iterator & operator--() { /* implementation omitted */ }
    iterator operator--(int) { iterator temp(*this); --(*this); return temp; }
  private:
    // representation
    TreeNode* ptr_;
  };

  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_); }
  ds_set& operator=(const ds_set<T>& old) { /* implementation omitted */ }
  int size() const { return size_; }

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

private:
  // REPRESENTATION
  TreeNode* root_;
  int size_;

  // PRIVATE HELPER FUNCTIONS
  TreeNode*  copy_tree(TreeNode* old_root) { /* Implemented in Lab 10 */ }
  void destroy_tree(TreeNode* p) { /* Implemented in Lecture 18 */





  }
  iterator find(const T& key_value, TreeNode* p) { /* Implemented in Lecture 18 */  }
  std::pair<iterator,bool> insert(const T& key_value, TreeNode*& p) {
    // NOTE: will need revision to support & maintain parent pointers
    if (!p) {
      p = new TreeNode(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* &p) { /* Implemented in Lecture 19 & 20 */ }
  void print_in_order(std::ostream& ostr, const TreeNode* p) const {
    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* p, int depth) const {
    if (p) {
      print_sideways_tree(ostr, p->right, depth+1);
      for (int i=0; i<depth; ++i) ostr << "    ";
      ostr << p->value << "\n";
      print_sideways_tree(ostr, p->left, depth+1);
    }
  }
};
```