CSCI-1200 Data Structures — Fall 2025
Lecture 19 – Trees, Part II

Review from Lecture 18

- Binary Trees, Binary Search Trees, & Balanced Trees
- In-order, pre-order, post-order, depth-first, breadth-first traversal
- Breadth-first and depth-first tree search
- STL `set` container class (like STL `map`, but without the pairs!)

Today's Lecture

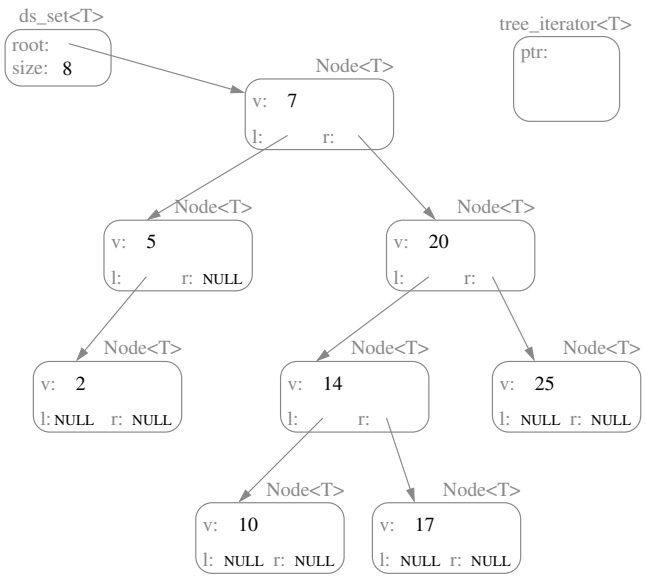
- Overview of the `ds_set` implementation
- Finding the smallest element in a BST.
- Exercises: `begin` and `find` and `destroy_tree`
- A very important `ds_set` operation: `insert`
- In-order, pre-order, and post-order traversal
- Breadth-first and depth-first tree search
- ... and more Big O Notation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

19.1 `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 and write functions for this implementation in future lectures & labs.
- 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 and why they the key operations are $O(\log n)$.

19.2 `ds_set`: Class Overview

- There is two helper classes, `TreeNode` and `tree_iterator`, that are *nested* within the "manager" `ds_set` class. All three classes are templated over two types.
- The only member variables of the `ds_set` class are the root and the size (number of tree nodes).
- The iterator class and is effectively a wrapper on the TreeNode pointers.
- `operator*` returns a `const` reference because the keys can't change.
- We'll tackle the implementation of increment and decrement operators in the 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.



19.3 Lecture Exercises

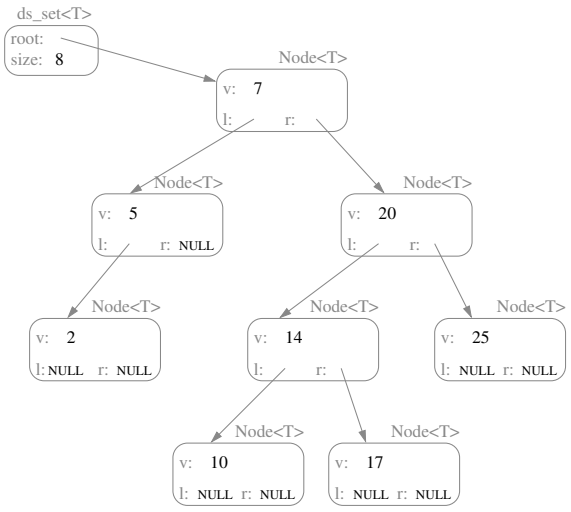
1. Provide the implementation of the member function `ds_set<T>::begin`. This is essentially the problem of finding the node in the tree that stores the smallest value.

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

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

19.4 Insert Overview

- Move left and right down the tree 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?



19.5 Insert Implementation

- Now implement `insert`:

```
std::pair<iterator,bool> insert(const T& key_value, TreeNode*& p) {




}
```

- 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.
  We'll discuss how to mitigate unbalanced trees in a future lecture.

19.6 Review: In-order, Pre-order, and Post-order

- For an exactly-balanced binary search tree with integers 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)`
- These traveral orderings are all *depth-first* traversals.
- 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.

19.7 Exercise

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

19.8 Breadth-first Search

- 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.

19.9 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.
- For our exactly balanced BST with elements 1-7, here is the desired output:

```
tier 1: 4
tier 2: 2 6
tier 3: 1 3 5 7
```

- 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 example tree with n nodes that illustrates each case.

```
// ------------------------------------------------------------------
// DS_SET CLASS -- WITH NESTED TREE NODE & TREE ITERATOR CLASSES

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++() { /* implemented in future lecture */ }
    iterator operator++(int) { /* implemented in future lecture */ }
    iterator & operator--() { /* implemented in future lecture */ }
    iterator operator--(int) { /* implemented in future lecture */ }
  private:
    // representation
    TreeNode* ptr_;
  };


  // DS_SET CONSTRUCTORS, ASSIGNMENT OPERATOR, & DESTRUCTOR
  ds_set() : root_(NULL), size_(0) {}
  ds_set(const ds_set<T>& old) : size_(old.size_) { root_ = this->copy_tree(old.root_); }
  ds_set& operator=(const ds_set<T>& old) { /* implementation omitted */ }
  ~ds_set() {  this->destroy_tree(root_); }

  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 {
    // Implemented in today's lecture




  }
  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 today's lecture




  }
  iterator find(const T& key_value, TreeNode* p) {
    // Implemented in today's lecture




  }
  std::pair<iterator,bool> insert(const T& key_value, TreeNode*& p) {
    // Implemented in today's lecture




  }
  int erase(T const& key_value, TreeNode* &p) { /* Implemented in future lecture */ }
  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);
    }
  }
};
```