CSCI-1200 Data Structures — Fall 2024
Lecture 18 – Trees, Part III

Review from Lecture 16 & 17 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
- Implementation of a breadth-first tree traversal

```
template <class T>
void breadth_first_print(TreeNode<T>* root) {
  int counter = 1;
  if (root == NULL) return;
  std::list< TreeNode<T>* > current;  // list of all nodes on a specific level
  current.push_back(root);
  std::list< TreeNode<T>* > next;     // list of all nodes on the next level
  while ( current.size() > 0 ) {      // print everything at this level
    std::cout << "level " << counter << ": ";
    typename std::list<TreeNode<T>*>::iterator itr = current.begin();
    while ( itr != current.end() ) {  // and collect items for next level
      TreeNode<T> *tmp = *itr;
      std::cout << tmp->value << " ";
      if (tmp->left != NULL) { next.push_back(tmp->left); }
      if (tmp->right != NULL) { next.push_back(tmp->right); }
      itr++;
    }
    current = next;                   // move on to the next level!
    next.clear();
    counter++;
    std::cout << std::endl;
  }
}
```

Today's Lecture

- `ds_set` & BST warmup exercises
- 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

18.1 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?

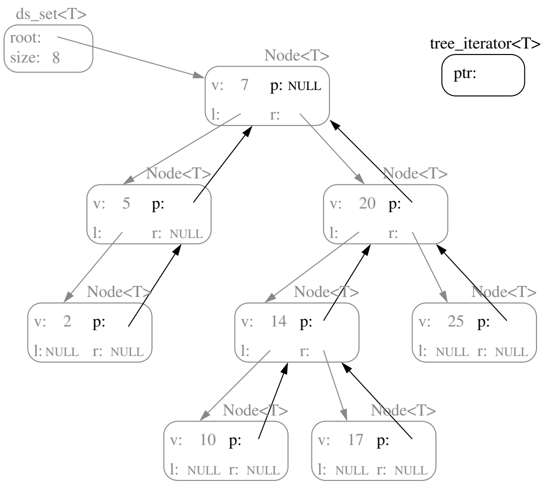
- In what order should a forward iterator visit the data?

- 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).

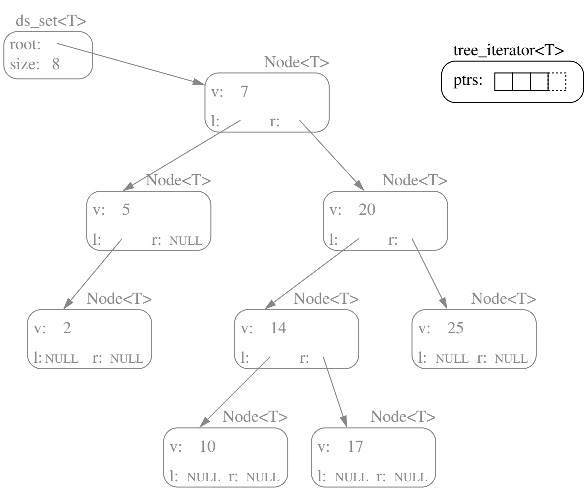
18.2 Tree Iterator Increment/Decrement - Implementation Choices

- 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!



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).



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?

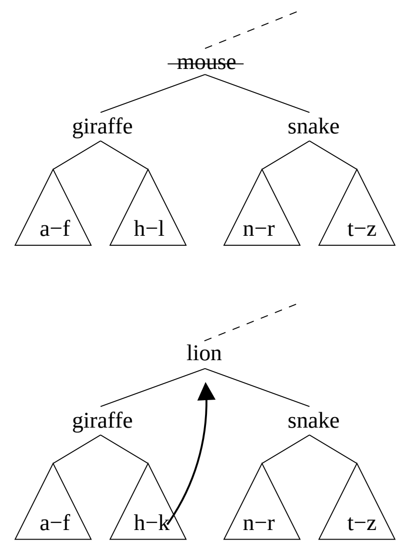
18.3 Erase

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!*

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.

18.4 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.

18.5 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?*

18.6 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

```
#ifndef ds_set_h_
#define ds_set_h_
#include <iostream>
#include <utility>
// -------------------------------------------------------------------
// DS_SET CLASS -- WITH NESTED NODE & ITERATOR CLASSES (ALTERNATE STYLE)
template <class T>
class ds_set {
public:
  // NODE CLASS
  class Node {
  public:
    Node() : left(NULL), right(NULL), parent(NULL) {}
    Node(const T& init) : value(init), left(NULL), right(NULL), parent(NULL) {}
    T value;
    Node* left;
    Node* right;
    Node* parent; // to allow implementation of iterator increment & decrement
  };
  // ITERATOR CLASS
  class iterator {
    public:
    iterator() : ptr_(NULL) {}
    iterator(Node* p) : ptr_(p) {}
    // 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_; }
    // pre & post increment & decrement operators
    iterator & operator++();                                           // ++itr
    iterator operator++(int) { iterator temp(*this); ++(*this); return temp; } // itr++
    iterator & operator--() { /* implementation omitted */ }                // --itr
    iterator operator--(int) { iterator temp(*this); --(*this); return temp; } // itr--
  private:
    // representation
    Node* ptr_;
  };

  // DS_SET CONSTRUCTORS, ASSIGNMENT OPERATOR, & DESTRUCTOR
  ds_set() : root_(NULL), size_(0) {}
  ds_set(const ds_set& old) : size_(old.size_) { root_ = copy_tree(old.root_,NULL); }
  ~ds_set() { destroy_tree(root_); root_ = NULL; }
  ds_set& operator=(const ds_set<T>& old) { /* implementation omitted */ }
  // SET FUNCTIONALITY
  int size() const { return size_; }
  iterator begin() const { /* implementation omitted */ }
  iterator end() const { return iterator(NULL,this); }
  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_, N
ULL); }
  int erase(T const& key_value) { return erase(key_value, root_); }
private:
  // REPRESENTATION
  Node* root_;
  int size_;
  // PRIVATE HELPER FUNCTIONS
  Node* copy_tree(Node* old_root, Node* the_parent);
  void destroy_tree(Node* p) { /* implementation omitted */ }
  iterator find(const T& key_value, Node* p) { /* implementation omitted */ }
  std::pair<iterator,bool> insert(const T& key_value, Node*& p, Node* the_parent);
  int erase(T const& key_value, Node* &p);
};
```

```
// DS_SET::ITERATOR FUNCTIONS
template <class T>
typename ds_set<T>::iterator& ds_set<T>::iterator::operator++() {
  /* implemented in Lecture 18 */









}

// DS_SET FUNCTIONS
template <class T>
typename ds_set<T>::Node* ds_set<T>::copy_tree(Node* old_root, Node* the_parent) {
  if (old_root == NULL)
    return NULL;
  Node *answer = new Node();
  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;
}
template <class T>
std::pair<typename ds_set<T>::iterator,bool>
ds_set<T>::insert(const T& key_value, Node*& p, Node* the_parent) {
  if (!p) {
    p = new Node(key_value);
    p->parent = the_parent;
    size_++;
    return std::pair<iterator,bool>(iterator(p,this), true);
  }
  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 std::pair<iterator,bool>(iterator(p,this), false);
}
template <class T>
int ds_set<T>::erase(T const& key_value, Node* &p) {
  /* Implemented in Lecture 18 */


















}
#endif
```