CSCI-1200 Data Structures — Spring 2023
Lecture 19 – Trees, Part III

Review from Lecture 17 & 18

- Overview of the `ds_set` implementation
- `begin`, `find`, `destroy_tree`, `insert`
- In-order, pre-order, and post-order traversal;
- Iterator implementation. Finding the in order successor to a node: add parent pointers *or* add a list/vector/stack of pointers to the iterator.
- B+ Tree Overview

Today's Lecture

- Last piece of `ds_set`: removing an item, `erase` (leetcode 450)
- Breadth-first and depth-first tree search (leetcode 102, 107, 429)
- Tree height, longest-shortest paths, breadth-first search (leetcode 104, 111)
- Erase with parent pointers, increment operation on iterators
- Limitations of our `ds_set` implementation

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?

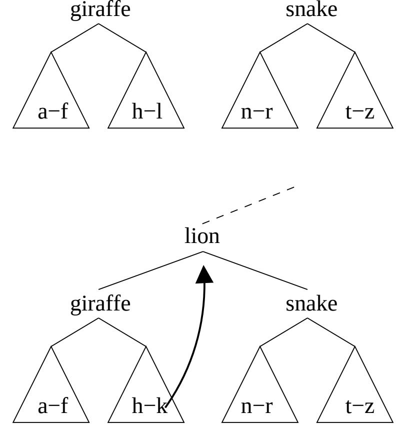
- In what order should a forward iterator visit the data? Draw an *abstract* table representation of this data (omits details of `TreeNode` memory layout).

19.2 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!

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.

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.

19.3 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 an "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?

19.4 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.

19.5 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 Erase (now with parent pointers)

- If we choose to use parent pointers, we need to add to the Node representation, and re-implement several `ds_set` member functions.
- **Exercise:** Study the new version of `insert`, with parent pointers.
- **Exercise:** Rewrite `erase`, now with parent pointers.

```
// -------------------------------------------------------------------
// 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;
  TreeNode* parent; // to allow implementation of iterator increment & decrement
};
template <class T> class ds_set;
// -------------------------------------------------------------------
// TREE NODE ITERATOR CLASS
template <class T> class tree_iterator {
public:
  tree_iterator() : ptr_(NULL), set_(NULL) {}
  tree_iterator(TreeNode<T>* p, const ds_set<T> * s) : ptr_(p), set_(s) {}
  // 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++() {
    if (ptr_->right != NULL) { // find the leftmost child of the right node
      ptr_ = ptr_->right;
      while (ptr_->left != NULL) { ptr_ = ptr_->left; }
    } else { // go upwards along right branches...  stop after the first left
      while (ptr_->parent != NULL && ptr_->parent->right == ptr_) { ptr_ = ptr_->parent; }
      ptr_ = ptr_->parent;
    }
    return *this;
  }
  tree_iterator<T> operator++(int) { tree_iterator<T> temp(*this);  ++(*this); return temp; }
  tree_iterator<T> & operator--() {
    if (ptr_ == NULL) { // so that it works for end()
      assert (set_ != NULL);
      ptr_ = set_->root_;
      while (ptr_->right != NULL) { ptr_ = ptr_->right; }
    } else if (ptr_->left != NULL) { // find the rightmost child of the left node
      ptr_ = ptr_->left;
      while (ptr_->right != NULL) { ptr_ = ptr_->right; }
    } else { // go upwards along left brances... stop after the first right
      while (ptr_->parent != NULL && ptr_->parent->left == ptr_) { ptr_ = ptr_->parent; }
      ptr_ = ptr_->parent;
    }
    return *this;
  }
  tree_iterator<T> operator--(int) { tree_iterator<T> temp(*this); --(*this); return temp; }
private:
  // representation
  TreeNode<T>* ptr_;
  const ds_set<T>* set_;
};
// -------------------------------------------------------------------
// 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) {
    if (&old != this) {
      this->destroy_tree(root_);
```

```
      root_ = this->copy_tree(old.root_,NULL);
      size_ = old.size_;
    }
    return *this;
  }
  typedef tree_iterator<T> iterator;
  friend class tree_iterator<T>;
  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_, NULL); }
  int erase(T const& key_value) { return erase(key_value, root_); }
  // ITERATORS
  iterator begin() const {
    if (!root_) return iterator(NULL,this);
    TreeNode<T>* p = root_;
    while (p->left) p = p->left;
    return iterator(p,this);
  }
  iterator end() const { return iterator(NULL,this); }
private:
  // REPRESENTATION
  TreeNode<T>* root_;
  int size_;
  // PRIVATE HELPER FUNCTIONS
  TreeNode<T>*  copy_tree(TreeNode<T>* old_root, TreeNode<T>* the_parent) {
    if (old_root == NULL) return NULL;
    TreeNode<T> *answer = new TreeNode<T>();
    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;
  }
  void destroy_tree(TreeNode<T>* p) {
    if (!p) return;
    destroy_tree(p->right);
    destroy_tree(p->left);
    delete p;
  }
  iterator find(const T& key_value, TreeNode<T>* 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);
  }
  std::pair<iterator,bool> insert(const T& key_value, TreeNode<T>*& p, TreeNode<T>* the_parent) {
    if (!p) {
      p = new TreeNode<T>(key_value);
      p->parent = the_parent;
      this->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);
  }
  int erase(T const& key_value, TreeNode<T>* &p) {
    /* Implemented in Lecture 20 */
  }
};
```