

CSCI-1200 Data Structures — Spring 2020

Lecture 17 – Trees, Part II

Review from Lecture 16

- 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

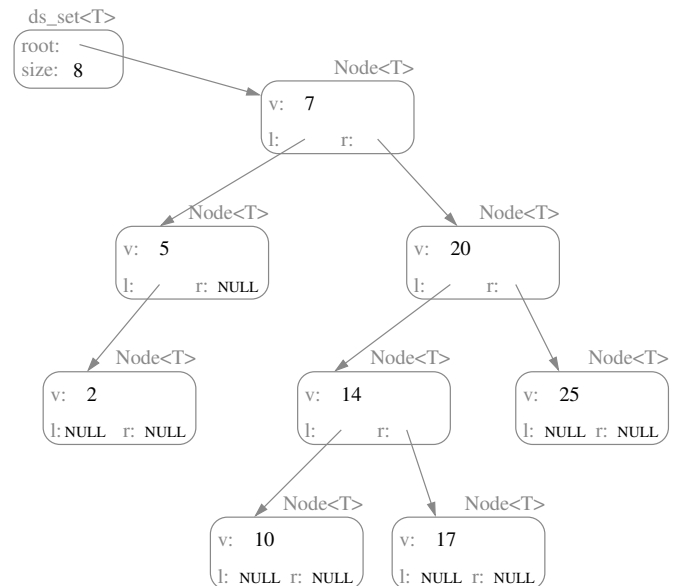
- Warmup / Review: `destroy_tree`
- A very important `ds_set` operation `insert`
- In-order, pre-order, and post-order traversal
- Finding the *in-order successor* of a binary tree node, tree iterator increment
- Advanced tree structure discussion

17.1 Warmup Exercise

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

17.2 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.

17.3 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?

17.4 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?

17.5 String Performance

- To motivate this new data structure, let's first think back to the idea of a string.
- Specifically, we will consider a string as either `char*` or as `vector<char>`
- Let's suppose we have a string of length 500. For each of these cases, what is the running time, using n for length of the string:
 1. `insert()/erase()` at the end, assuming we have room for at least 501 characters already
 2. `insert()/erase()` at the end, but we don't have extra room at the end of the array/vector
 3. `insert()/erase()` at middle (index 249 in our example), assuming we have room for at least 501 characters already?
- Many data structures involve a trade-off: we make something faster at the expense of additional space, or by making other operations slower. We will look at a way to speed up `insert()/erase()` but it will slow down our `operator[]`. However, it will not be as bad as if we fixed this by using a `list<char>` and we will be able to quickly use an index instead of iterators.
- Some of this material will be handed out Friday instead of today.
Sources are wikipedia, e.g. [https://en.wikipedia.org/wiki/Rope_\(data_structure\)#/media/File:Vector_Rope_index.svg](https://en.wikipedia.org/wiki/Rope_(data_structure)#/media/File:Vector_Rope_index.svg) - be careful since outside resources may not exactly match our definitions.

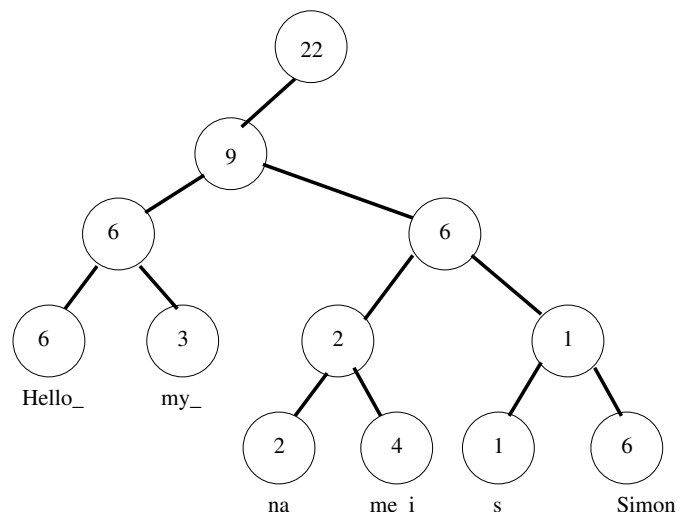
17.6 Rope Node class

- We'll use a tree-like structure. First consider this modified `Node` class:

```
class Node{
public:
    Node() : left{NULL}, right{NULL}, parent{NULL}, weight{0} {}
    Node* left;
    Node* right;
    Node* parent;
    int weight;
    std::string value;
};
```

17.7 Rope Example

- Now we can construct a `Rope` by using the following rules:
 1. Only leaves will have a non-empty **value** member.
 2. The **weight** of a leaf is the length of its **value** member.
 3. For any non-leaf, the **weight** is the sum of the length of every **value** in its left subtree.
- To the right is one possible way we could make a `Rope` for the string "Hello my name is Simon". To make it clear where spaces are, our figure uses `'_'`, but the real **value** uses `' '`.



17.8 Rope *index()*

- In binary search trees, we used `operator<` to decide whether to go left or right. In a Rope that won't work.
- See the homework handout for details
- Let's walk through calls to `index(0)`, `index(8)`, and `index(20)`, keeping track of i and whether we go left or right at every step.

17.9 Rope *concat()*

- Concatenation is a pretty simple operation.
- We must carefully consider if we want to copy nodes or just glue existing nodes together.
- We often talk about a left-hand side (LHS) and right-hand side (RHS) when we have two operands
- See the homework handout for details
- The version in our code should copy *rhs*'s values, because otherwise `~lhs()` and `~rhs()` will try to free the same nodes.
- However a version that doesn't copy anything will be useful for `split()`.

17.10 Rope *split()*

- Breaking a Rope into two smaller Ropes is another important operation.
- We'll start by considering the "main" case, where the split will happen between two nodes, for example $i = 11$
- See the homework handout for details
- During the concatenation of severed nodes, should we make copies? Why or why not? *Hint: Think about what `~lhs()` and `~rhs()` can and can't reach.*
- If the split happens in the middle of a leaf, such as with $i = 12$, we just need to split the leaf so that $i = 12$ is at the start of a leaf. This is like concatenate in reverse.

17.11 Our Rope Limitations

- Normally to stay efficient, a balanced tree is used
- Whenever we do an operation like `split()`, `insert()`, or `erase()`, we would normally rebalance the tree.
- We're going to skip that for now, but try and think about how this might impact the performance of our rope.
- Another issue is iterators - we will be using the same iterator we used today for BSTs.
- A common task is to just move between leaves in a rope, so real implementations often have a separate iterator for leaf traversal
- This still requires a fair amount of work, but for very large strings with many operations performed on them, n vs. $\frac{n}{2}$ can be a huge savings.

```

// -----
// 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; }
    // comparisons 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 9 */ }
void destroy_tree(TreeNode<T>* p) {
    /* Implemented in Lecture 18 */
}

iterator find(const T& key_value, TreeNode<T>* p) { /* Implemented in Lecture 17 */ }

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 19 */ }

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);
    }
}

};

```