CSCI-1200 Data Structures — Spring 2023
Lecture 18 – Trees, Part II

Review from Lecture 17

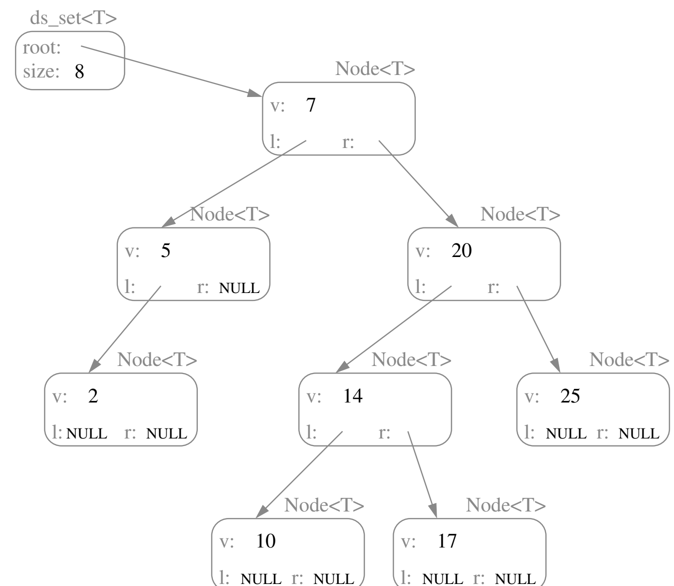
- Binary Trees, Binary Search Trees, & Balanced Trees
- STL `set` container class (like STL `map`, but without the pairs!)
- Finding the smallest element in a BST.
- Overview of the `ds_set` implementation: `begin` and `find`. (leetcode 700)

Today's Lecture

- Warmup / Review: `destroy_tree`
- A very important `ds_set` operation insert (leetcode 701)
- 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 (leetcode 559, 589, 590)
- HW8 discussion

18.1 Warmup Exercise

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



18.2 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.

18.3 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?

18.4 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.*

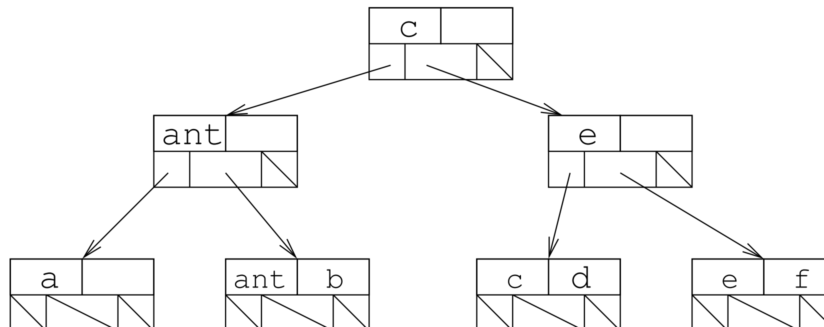
Exercise: What are the advantages & disadvantages of each method?

18.5 Limitations of Our BST Implementation

- The efficiency of the main insert, find and erase algorithms depends on the height of the tree.
- The best-case and average-case heights of a binary search tree storing n nodes are both $O(\log n)$. The worst-case, which often can happen in practice, is $O(n)$.
- Developing more sophisticated algorithms to avoid the worst-case behavior will be covered in Introduction to Algorithms. One elegant extension to the binary search tree is described below...

18.6 B+ Trees

- Unlike binary search trees, nodes in B+ trees (and their predecessor, the B tree) have up to b children. Thus B+ trees are very flat and very wide. This is good when it is very expensive to move from one node to another.
- B+ trees are supposed to be associative (i.e. they have key-value pairs), but we will just focus on the keys.
- Just like STL `map` and STL `set`, these *keys* and *values* can be any type, but *keys* must have an `operator<` defined.
- In a B tree key-value pairs can show up anywhere in the tree, in a B+ tree all the key-value pairs are in the leaves and the non-leaf nodes contain duplicates of some keys.
- In either type of tree, all leaves are the same distance from the root.
- The keys are always sorted in a B/B+ tree node, and there are up to $b-1$ of them. They act like $b-1$ binary search tree nodes mashed together.
- In fact, with the exception of the root, nodes will always have between roughly $\frac{b}{2}$ and $b-1$ keys (in our implementation).
- If a B+ tree node has k keys $key_0, key_1, key_2, \ldots, key_{k-1}$, it will have $k+1$ children. The keys in the leftmost child must be $< key_0$, the next child must have keys such that they are $\geq key_0$ and $< key_1$, and so on up to the rightmost child which has only keys $\geq key_{k-1}$.
- HW8 will focus on implementing some of the functionality of a B+ tree. It won't be enough to replace a real B+ tree, but it will be enough to understand how the tree works and construct trees.



- Note: "a" will come before "ant" lexicographically, in other words "a" $<$ "ant"
- Considerations in a full implementation:
  - What happens when we want to add a key to a node that's already full?
  - How do we remove values from a node?
  - How do we ensure the tree stays balanced?
  - How to keep leaves linked together? Why would we want this?
  - How to represent key-value pairs?

Exercise: Draw a B+ tree with $b = 3$ with values inserted in the order $1, 2, 3, 4, 5, 6$. Now draw a B+ tree with $b = 3$ and values inserted in the order $6, 5, 4, 3, 2, 1$. Hint: The two trees have a different number of levels.

18.7 HW8 Hints

- You are *not* implementing a full B+ tree. Read the homework assignment carefully and keep it in mind if you look up videos/notes on B+ trees.
- You should put your implementation at the bottom of the .h file - do not change the forward declaration.
- Use the provided .h file - don't start from scratch. Since this is a templated class your entire implementation can go in the .h file.
- *find()* will only return NULL for an empty tree. Otherwise it will always return a leaf pointer. If the value is in the tree, the pointer should point to the leaf containing the value. If the value is not in the tree, the leaf should point to the last node visited in find(), which should be a leaf.
- You will **not** be graded on test cases that you write. You do **not** need to submit any test cases. We will only use your .h file and we will only read your .h file and README
- Insertion order can make a HUGE difference. Try inserting a,b,c,d,e,f into a tree with b=3 and then try inserting f,e,d,c,b,a into a tree with b=3. What happens and why?
- Don't try to use std::sort() to keep internal pointers sorted - because the nodes are templated, this creates a big mess. You're on your own if you want to pass a templated function to std::sort() as a 3rd argument.
- *PrintSideways()* makes the split at $b/2$ nodes using integer division. Our tree printing functions use tabs (\t) instead of spaces.
- Compile with -Wall. Read all warnings. Fix all warnings. Read any compiler warnings on Submitty. Fix those warnings too.

```
// -------------------------------------------------------------------
// 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; }
  // 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++() { /* 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);
    }
  }
};
```