CSCI-1200 Data Structures — Fall 2020
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; Breadth-first and depth-first tree search

Today's Lecture

- Bonus C++ Topic: Template Specialization (for Homework 8)
- Finish Last Lecture
  - Implementation of a breadth-first tree 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.
- Last piece of `ds_set`: removing an item, `erase`
- Tree height, longest-shortest paths, breadth-first search
- To support increment/decrement: Copy tree, Insert, and Erase with parent pointers

19.1 Template Specialization Example

Writing templated functions is elegant and powerful, but sometimes we do not want to handle all types in exactly the same way. Sometimes we want to write different versions of the function depending on the type:

- Let's study and discussion the following code:

```
// We'll use this templated function (unless we find a specialized
// implementation for our type)
template <class T>
void print_vec (const std::vector<T> &v) {
  std::cout << "count=" << v.size() << "  data=";
  for (unsigned int i = 0; i < v.size(); i++) {
    std::cout << " " << v[i]; }
  std::cout << std::endl;
}

// This will match doubles (but not floats)
void print_vec (const std::vector<double> &v) {
  std::cout << "count=" << v.size() << "  data=";
  for (unsigned int i = 0; i < v.size(); i++) {
    std::cout << std::setprecision(1) << std::fixed << " " << v[i]; }
  // unset the formatting
  std::cout << std::defaultfloat << std::endl;
}

int main() {
  // note: this syntax for initialization of vector contents is available with C++11
  std::vector<int> int_v = { 1, 2, 3, 4, 5 };
  std::vector<double> double_v = { 1, 2, 3, 4, 5 };
  std::vector<float> float_v = { 1, 2, 3, 4, 5 };
  std::vector<std::string> string_v = { "1", "2", "3", "4", "5" };
  print_vec(int_v);
  print_vec(double_v);
  print_vec(float_v);
  print_vec(string_v);
}
```

```
// This would match strings...  but because it's placed after the
// usage in main it's not used!?!?!
void print_vec (const std::vector<std::string> &v) {
  std::cout << "count=" << v.size() << "  data=";
  for (unsigned int i = 0; i < v.size(); i++) {
    std::cout << " \"" << v[i] << "\""; }
  std::cout << std::endl;
}
```

- If we commented out the specialized implementations of `print_vec` for the double and string types:

```
count=5  data= 1 2 3 4 5
count=5  data= 1 2 3 4 5
count=5  data= 1 2 3 4 5
count=5  data= 1 2 3 4 5
```

- If we run the original code:

```
count=5  data= 1 2 3 4 5
count=5  data= 1.0 2.0 3.0 4.0 5.0
count=5  data= 1 2 3 4 5
count=5  data= 1 2 3 4 5
```

- If we swap the order of the main function and the string version of `print_vec`:

```
count=5  data= 1 2 3 4 5
count=5  data= 1.0 2.0 3.0 4.0 5.0
count=5  data= 1 2 3 4 5
count=5  data= "1" "2" "3" "4" "5"
```

19.2 General-Purpose Breadth-First Search/Tree Traversal

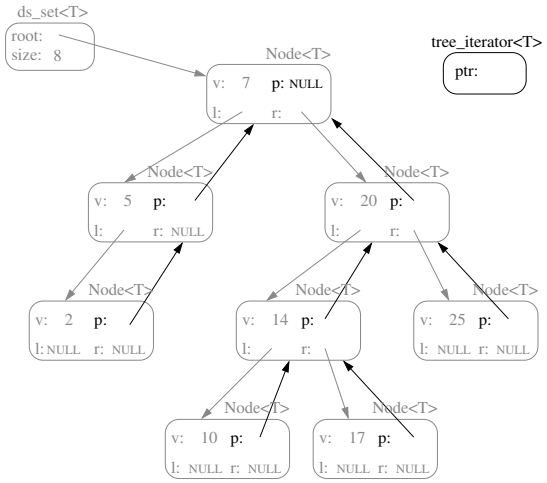
- Write an algorithm to print the nodes in the tree one tier at a time, that is, in a *breadth-first* manner.

- 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.

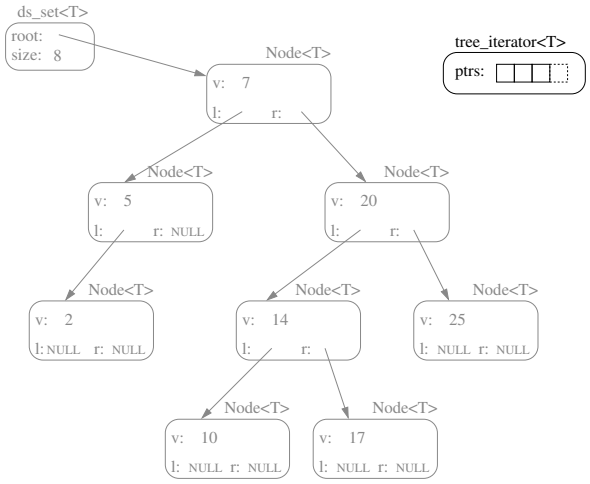
19.3 Tree Iterator Increment/Decrement - Implementation Choices

- 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?

19.4 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!*

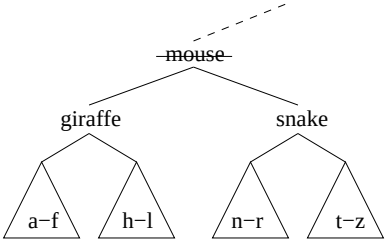
no children

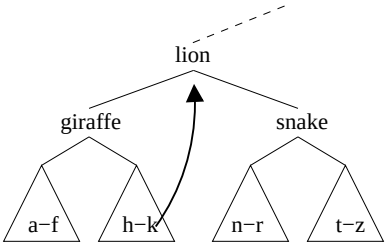
only a left child
(with potentially a big subtree)

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.

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 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

```
// ----------------------------------------------------------------
// DS_SET CLASS -- WITH NESTED TREE NODE & TREE ITERATOR CLASSES (ALTERNATE STYLE)
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), set_(NULL) {}
    iterator(TreeNode* p, const ds_set * 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 iterator& rgt) { return ptr_ == rgt.ptr_; }
    bool operator!= (const iterator& rgt) { return ptr_ != rgt.ptr_; }
    // increment & decrement operators
    iterator & operator++() {
       /* discussed & implemented in Lecture 19 */




    }
    iterator operator++(int) { iterator temp(*this); ++(*this); return temp; }
    iterator& operator--() {  /* implementation omitted */ }
    iterator operator--(int) { iterator temp(*this); --(*this); return temp; }
  private:
    // representation
    TreeNode* ptr_;
    const ds_set* 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_,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;
  }
  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* p = root_;
    while (p->left) p = p->left;
    return iterator(p,this);
  }
  iterator end() const { return iterator(NULL,this); }
private:
  // REPRESENTATION
  TreeNode* root_;
  int size_;
  // PRIVATE HELPER FUNCTIONS
  TreeNode*  copy_tree(TreeNode* old_root, TreeNode* the_parent) {
    if (old_root == NULL) return NULL;
    TreeNode *answer = new TreeNode();
    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* p) {
    if (p) { destroy_tree(p->left); destroy_tree(p->right); delete p; }}
  iterator find(const T& key_value, TreeNode* 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*& p, TreeNode* the_parent) {
    if (!p) {
      p = new TreeNode(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* &p) {
    /* Implemented in Lecture 19 */




  }
};
```