CSCI 2270
Data Structures and Algorithms
Lecture 17

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128
Wed 1-2pm
Fri 2-3pm

Administrivia

Thursday 2-3 office hours have moved to Friday 2-3

HW2 will post on Tuesday (part 1)

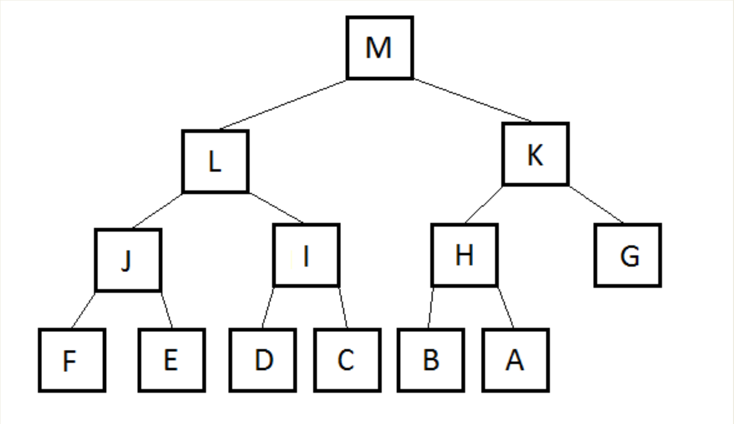
linked list implementation of a biiig integer in any base

Lab this week: review for test

One bug in doubly linked list code: posted correction

Tree traversals

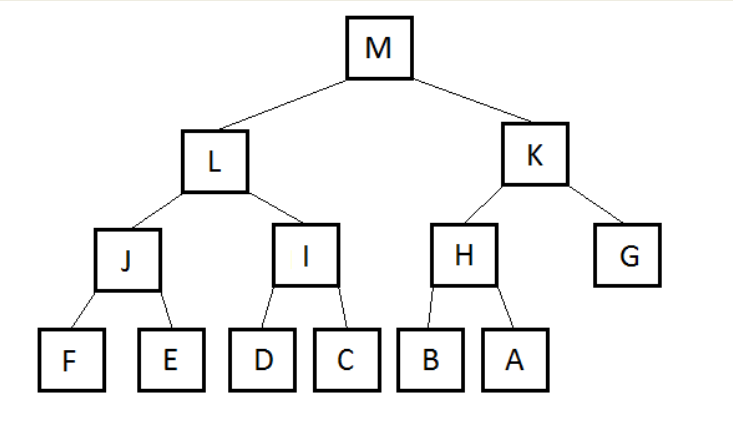
Recursive process




```
preorder_print (const binary_tree_node* bintree)
        if (bintree is not empty)
        {
                print out data at root
                preorder_print(left subtree of bintree's root)
                preorder_print(right subtree of bintree's root)
        }
```

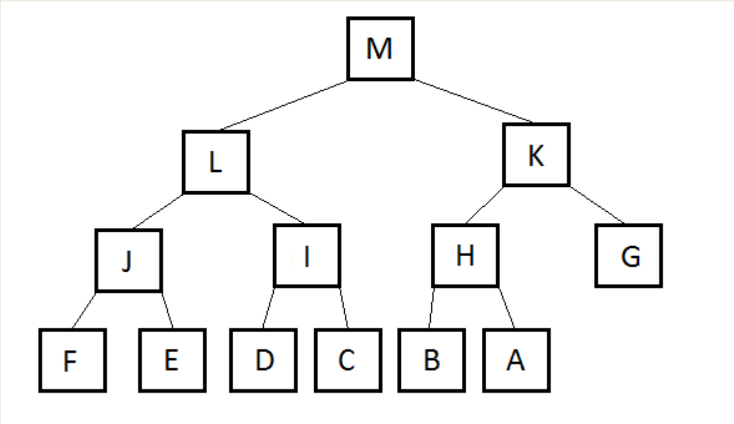
Tree traversals

Recursive process



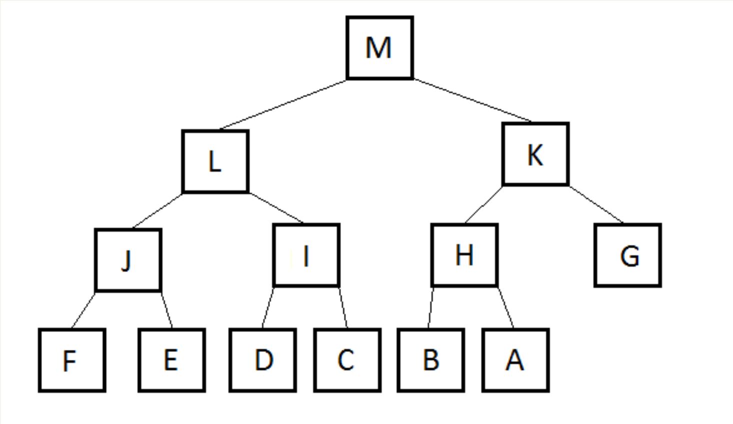

```
preorder_print (const binary_tree_node* bintree)
        if (bintree is not empty)
        {
                print out data at root
                preorder_print(left subtree of bintree's root)
                preorder_print(right subtree of bintree's root)
        }
```

Tree traversals




```
inorder_print (const binary_tree_node* bintree) :
        if (bintree is not empty)
        {
                inorder_print(left subtree of bintree's root)
                print out data at root
                inorder_print(right subtree of bintree's root)
        }
```

Tree traversals




```
postorder_print (const binary_tree_node* bintree) :
        if (bintree is not empty)
        {
                postorder_print(left subtree of bintree's root)
                postorder_print(right subtree of bintree's root)
                print out data at root
        }
```

Destroying a binary tree

```
void tree_clear(binary_tree_node*& root_ptr) {
        binary_tree_node* child;
        if (root_ptr != nullptr)
        {
                child = root_ptr->left;
                tree_clear( child );
                child = root_ptr->right;
                tree_clear( child );
                delete root_ptr;
                root_ptr = nullptr;
        }
  }
```

Destroying a binary tree

```
void tree_clear(binary_tree_node*& root_ptr) {
        binary_tree_node* child;
        if (root_ptr != nullptr)    // what happens if it is nullptr?
        {           // destroy child nodes first (else we lose them)
                child = root_ptr->left;
                tree_clear( child );
                child = root_ptr->right;
                tree_clear( child );
                delete root_ptr;
                root_ptr = nullptr;
        }
}
```

Copying a binary tree

```
binary_tree_node* tree_copy(const  binary_tree_node*
        root_ptr)   {
        binary_tree_node* l_ptr; binary_tree_node* r_ptr;
        if (root_ptr == nullptr)  return nullptr;
        else {
           l_ptr = tree_copy( root_ptr->left );
           r_ptr = tree_copy( root_ptr->right );
           binary_tree_node* new_tree = new binary_tree_node;
           new_tree->data = root_ptr->data;
           new_tree->left = l_ptr; new_tree->right = r_ptr;
           return new_tree;
        }
}
```

Traversal order matters (*post order*)

Tree_clear:

```
child = root_ptr->left; tree_clear( child );
child = root_ptr->right; tree_clear( child );
delete root_ptr; root_ptr = nullptr;
```

Tree_copy:

```
l_ptr = tree_copy( root_ptr->left );
r_ptr = tree_copy( root_ptr->right );
binary_tree_node* new_tree = new
        binary_tree_node;
new_tree->data = root_ptr->data;
new_tree->left = l_ptr; new_tree->right = r_ptr;
return new_tree;
```

Trees offer the chance for log(n) performance if you're lucky

For a binary search tree of n nodes,

- Search: O(log n) average, O(n) worst case
- Insert: O(log n) average, O(n) worst case
- Remove: O(log n) average, O(n) worst case
- Traverse: O(n)

Another tree ADT, the B-tree, ensures that the tree is perfectly full—guarantees the O(log n) performance, worst case