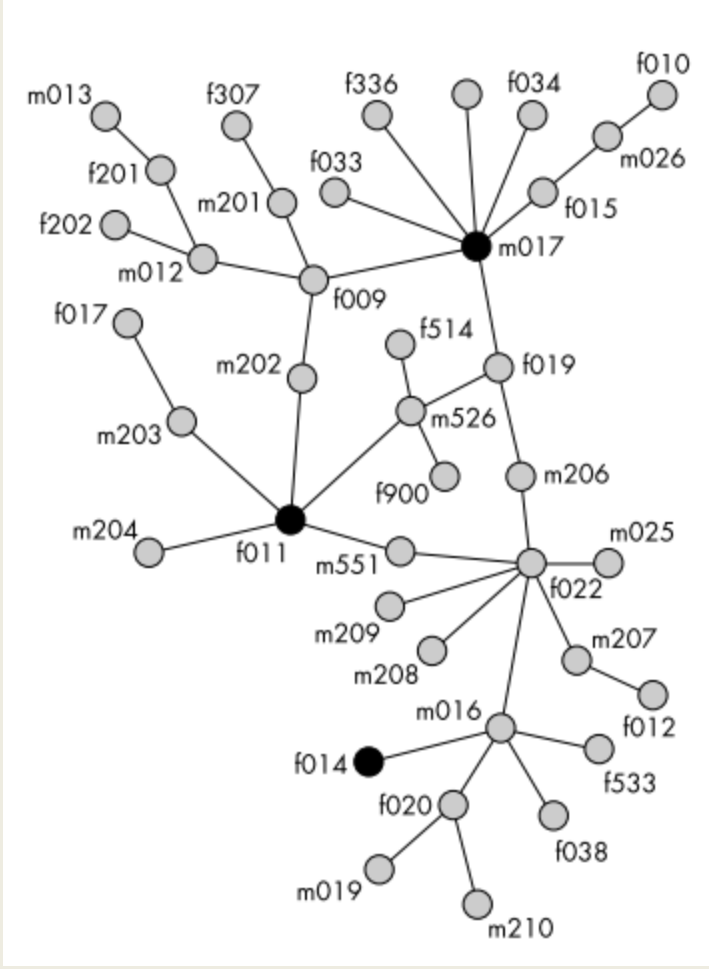
CSCI 2270
Data Structures and Algorithms
graph lecture 2

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



Administrivia

Last homework on graphs, due Saturday night

Find shortest-hop path between 2 cities

depth first or breadth first search

Lab this week: buffer overrun

Today: exam, programming depth first search

Wednesday's lecture demo: hash tables and timings

Read 11.1-11.4, on hashing.

Don't get bogged down in the details.

Assignment

```
big_number a(87);
big_number b(78);
big_number c;
c = a;
++a;
cout << c << " " << a << endl;
```

What prints?

Default assignment operator

If we never wrote the operator = for big_number, C++ would supply us with one. (I got you, baby!) Look what it does:

```
this->positive = m.positive;          // ok
this->digits = m.digits;              // ok
this->base = m.base;                  // ok
this->head_ptr = m.head_ptr;          // oh no
this->tail_ptr = m.tail_ptr;          // also no
```

```
big_number a(87);  big_number b(78); big_number c;
c = a;
++a;
cout << c << " " << a << endl;
```

What prints?

Default assignment operator

“Hey, that’s weird. I have the exact same dog.”

Default assignment operator

```
big_number a(87);  big_number b(78); big_number c;
c = a;
++a; ++a; ++a; ++a; ++a;
++a; ++a; ++a; ++a; ++a;
cout << c << “ “ << a << endl;
```

What could possibly go wrong?

Your assignment operator

Your assignment operator gives you a deep copy

each big_number has its own digits list

```
// assignment operator
big_number& big_number::operator=(const big_number& m)
{
    if (this == &m) return *this;
    clear_list(head_ptr, tail_ptr);
    base = m.base;
    copy_list(m.head_ptr, head_ptr, tail_ptr);
    positive = m.positive;
    digits = m.digits;
    return *this;
}
```

Your copy constructor

Your copy constructor also gives you a deep copy

each big_number has its own digits list

```
// copy constructor
big_number::big_number(const big_number& m)
{
        head_ptr = nullptr;
        tail_ptr = nullptr;
        *this = m;                  <- operator = runs here
}
```

Depth first search 1

```
bool graph::does_dfs_path_exist(const string& city1,
        const string& city2)  {
        deque<vertex*> yet_to_explore;       // the 'stack'
        map<vertex*, bool> visited;          // breadcrumbs
        map<vertex*, vertex*> path;          // route
        // initialize visited to all false, path to all nullptrs
        // find vertex* v for city1 and vertex* u for city2
        // add v to your deque and mark its visited as true
        return depth_first_search(u, visited, yet_to_explore,
                path);
}
```

Depth first search 2

```
bool graph::depth_first_search(vertex* u, map<vertex*, bool>&
        visited, deque<vertex *>& yet_to_explore,
        map<vertex*, vertex*>& path)  {
        // if the deque is not empty,
        //        find the last item in the deque, w
        //        pop this item w off the deque
        //        if that item's u, we're done (base case)
        //        else
        //                push w's unvisited neighbors on the deque
        //                set path[neighbor] = w
        //                mark each neighbor you pushed as visited
        //        return depth_first_search(u, visited,
        //                yet_to_explore, path);
        // else  give up                    }        <- bracket
```