CS 132 - Intro to Computer Science II - Spring 2004
WEEK #14 LAB EXERCISE and Homework #11

Week #14 Lab Exercise due: Wednesday, April 27th, END of lab
HW #11 due: Wednesday, May 4th, 1:00 pm

Purpose: Thinking/experience with graphs

WEEK #14 LAB EXERCISE

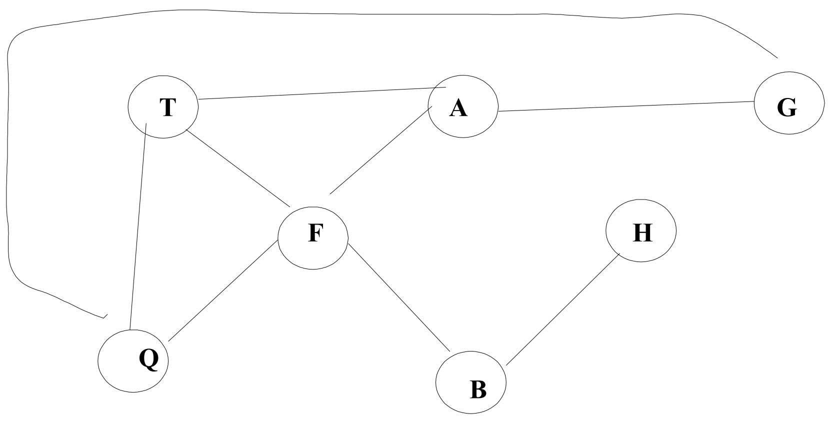
Answer the following questions on paper; after coming up with your initial answers, you may discuss them with another student before getting your answers checked, if you wish. When you are ready, put your name on the "Next:" list on the board so that your answers can be checked. [Note: check **carefully**! There will be **points docked** for errors, this time, to encourage you to double-check carefully *before* getting your work checked.]

Consider the following **pseudocode** for **depth-first search:**

```
// PSEUDOCODE - RECURSIVE VERSION
// dfs
// Purpose: traverses a graph g beginning at vertex v by using a depth-first
//          search:
//          Recursive version
template <typename Item>
void dfs (graph<Item> g, Item v)
{
    // mark v as visited
    g.mark(v);
    cout << "visited: " << v << endl;

    for (each unvisited vertex u adjacent to v)
    {
        dfs(g, u);
    }
}
```

Assume further that you have the following graph g1:



1. Write this graph in **G = {V, E}** form.

____

____

____

2. Assume that, for professorial sanity, when you have a choice of adjacent unvisited nodes, you choose the next node in increasing alphabetical order.

Then, write what could be printed to the screen as a result of:

(a) dfs(g1, 'G'); **(b) dfs(g1, 'F');**

3. Now consider this pseudocode for breadth-first-search:

```
// PSEUDOCODE
// bfs
// Purpose: traverses a graph beginning at vertex v by using a breadth-first
//          search

template <typename Item>
void bfs(graph<Item> g, Item v)
{
    queue<Item> myQ;
    Item w, u;

    // add v to queue and mark it
    myQ.enqueue(v);
    g.mark(v);
    cout << "visited: " << v << endl;

    while (!myQ.empty())
    {
        w = myQ.dequeue( );

        // loop invariant: there is a path from vertex w to every vertex in
        // the queue myQ
        for (each unvisited vertex u adjacent to w)
        {
            // mark u as visited
            g.mark(u);
            cout << "visited: " << u << endl;
            myQ.enqueue(u);
        }
    }
}
```

Again, assume that, for professorial sanity, when you have a choice of adjacent unvisited nodes, you choose the next node in increasing alphabetical order.

And now show what would be printed for the calls:

(a) **bfs(g1, 'G');** **(b)** **bfs(g1, 'F');**

____ ____

____ ____

____ ____

____ ____

____ ____

____ ____

____ ____

<u>HOMEWORK #11:</u>

1. Now, copy all of the files accompanying this assignment handout on the public course web page into your desired current working directory on cs-server.

You now have a (possibly-buggy, hopefully-not) implementation of a template class **graph**, implemented using adjacency lists (which can be done in a way rather reminiscent of a buckets-and-chaining hash table! But I digress...) You also have some other handy ADT's from previous assignments.

Show that you have everything you need for the graph implementation, at least, by:

(a) Adding a a **cout** statement containing your name to **try_graph.cpp**,

(b) compiling it, running it, and running it redirecting its output into **try_graph_out1**:

```
try_graph > try_graph_out1
```

...and submitting **try_graph_out1**.

2. I want to make sure you are at least a little familiar with the capabilities of this provided **graph.h** and **graph.template**. So, write a small program **graph_ex.cpp** that simply uses **graph.h** and **graph.template** to create the graph used in the lab exercise, and to print it using **graph**'s **print_graph** member function.

[Of course, you should precede that **print_graph** call with a printout saying what you expect to see --- but, you may use the style used in **try_graph.cpp** for this. That is, it is sufficient to list the expected vertices and edges; they needn't be formatted/look exactly the same as print_graph depicts them, nor must each edge be listed twice as print_graph does. The point is that the reader can tell if the expected graph and actual graph are the same in essence.]

Run:

```
graph_ex > graph_ex_out
```

...and submit your resulting **graph_ex.cpp** and **graph_ex_out .**

3. If I wanted you to *implement* bfs and dfs --- I couldn't, yet, with the provided **graph**. It has no way to **mark** nodes as visited.

SO --- we're going to **modify** it so that it does.

Modify **graph.h** and **graph.template** such that you:

* add a separate array of type **bool** and size **MAXIMUM** called **vertex_markings** that is initially all false --- it represents the current markings for all vertices in the graph.

* (note that get_vert_index(label) returns the index into array **vertices** for vertex label --- this value should also be label's index into **vertex_markings**.)

* add a member function **unmark_all** which re-marks all vertices as false;

* add a member function **mark** which takes an Item **vert** and sets the mark for the vertex **vert** to true.

* add a member function **get_mark** which takes an Item **vert** and returns the current marking for **vert**.

Add appropriate tests of **mark**, **unmark_all**, and **get_mark** to **try_graph.cpp**. Run:

try_graph > try_graph_out2

...and submit versions of your modified **graph.h**, **graph.template**, **try_graph.cpp**, and **try_graph_out2**

4. Implement the pseudocode for **dfs** given in the lab exercise. [HOWEVER, you do not have to implement the professorial-sanity clause! You can choose the next adjacent unvisited node in any way that you like.]

In **babytest_dfs.cpp**, run your **dfs** function on the graph from homework problem #1 using both calls from the lab exercise, putting the output into **babytest_dfs_out** and submitting your **dfs.template**, **babytest_dfs.cpp** and **babytest_dfs_out**.

[yes, **babytest_dfs_out** should still print out actual and expected results --- here, though, just summarizing the order that you expect the nodes to be visited before each **dfs** call will suffice. The "expected" doesn't have to put 1 node per line with "visited:" as **dfs** will actually do. ASK ME if you are not sure what I mean by this.]

5. Implement the pseudocode for **bfs** given in the lab exercise. [AGAIN, you do not have to implement the professorial-sanity clause! You can choose the next adjacent unvisited node in any way that you like.]

In **babytest_bfs**, run your **bfs** function on the graph from homework problem #1 using both calls from the lab exercise, putting the output into **babytest_bfs_out** and submitting your **bfs.template**, **babytest_bfs.cpp** and **babytest_bfs_out**.

[yes, **babytest_bfs_out** should still print out actual and expected results --- again, though, just summarizing the order that you expect the nodes to be visited before each **bfs** call will suffice. The "expected" doesn't have to put 1 node per line with "visited:" as **bfs** will actually do. ASK ME if you are not sure what I mean by this.]

And, when you are satisfied with all of the above, submit them using **~st10/132submit** on cs-server:

try_graph_out1
graph_ex.cpp, graph_ex_out
graph.h, **graph.template, try_graph.cpp**, and **try_graph_out2**
dfs.template, **babytest_dfs.cpp, babytest_dfs_out**
bfs.template, babytest_bfs.cpp, babytest_bfs_out