CSCI 2270 Data Structures Recitation 11
Instructors: Hoenigman/Zagrodzki/Zietz

Graph Traversal

Learning Objectives:

- Using standard libraries — queue and stack
- Traverse a graph using BFS and DFS

1. Breadth First Search

1.1 Using Standard Libraries: Queue

Remember a queue is a data structure that implements "FIFO", that is, first-in-first-out. A quick reminder: if we enqueue the following numbers in the order:

```
1  4  7  2  3 ,
```

when we dequeue a number from this queue, we will get number **1**. Next time when we perform dequeue operation, we get number **4**. Although we can create our own queue structure by using array or vector, the standard library in C++ has already implemented a queue class that we can directly use. The following code is an example of using queue library. Notice how to include the library, define a queue, and several functions defined in the queue class.

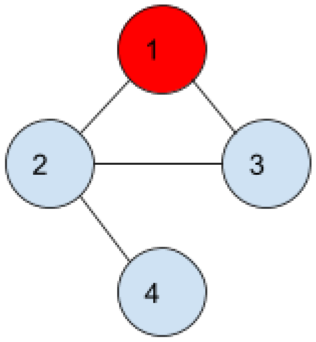
```
#include <iostream>
#include <queue>

using namespace std;

int main() {
      queue<int> the_queue;
      the_queue.push(1);
      the_queue.push(4);
      the_queue.push(7);
      the_queue.push(2);
      the_queue.push(3);

      while(!the_queue.empty()) {
       the_queue.pop();
       queue<int> copy_queue = the_queue;
```

```
        cout << "Current queue: ";
        while (!copy_queue.empty()) {
              cout << copy_queue.front() << " ";
              copy_queue.pop();
        }
        cout << endl;
      }
      return 0;
}
```

1.2. Breadth First Search Algorithm

We will use queue structure to keep track of node traversals in our graph traversal algorithm. Recall BFS in a tree structure. Briefly, we start from the root of the tree. Assume we're at node *i*. Then we visit all the children of node *i*. After this step, we start again from all the children *j* of node *i* and visit all the children of each node *j*, and repeat this step.

Similarly, in graph, we apply the same idea. Since there's no root node in graph, we usually specify a starting node. We also need two variables for BFS: **a queue** and a **"visiting" array**. Say we have the following graph where we start at node 1. After visiting node 1, we want to visit its adjacent nodes *in order*, which is 2 first and then 3. In other words, we push all the adjacent nodes of node 1 in an order, and visit them in the same order. This is exactly a queue structure — we push node 2 first and then 3, therefore we visit 2 first (dequeue or pop) and then visit 3.



Different from a tree structure where any node can only be visited once, a node in a graph structure can be visited multiple times. For example, in the above graph, when we visit node 2, we need to create a queue and push all the adjacent nodes into the queue — 1, 3, 4. The node 1, however, has already been visited before, so we don't need to visit it twice and we need to skip it. To implement this, we use an array to record if node has been visited or not. Say we define an array, `visited`, whose size is the number of nodes. If node `i` has been visited, we set `visited[i] = 1`. Otherwise `visited[i] = 0`. Alternatively, we can also define a variable `visited` in a node structure.

To sum, we create a queue and a visiting array. When visiting a node, we examine all the adjacent nodes of the current node. If an adjacent node has not been visited, i.e., `visited[i] = 0`, we add it to the queue. When the queue is empty, meaning we have visited all the nodes, we complete BFS algorithm.

2. Depth First Search

2.1 Using Standard Libraries: Stack

Recall that the idea of stack is "LIFO", last-in-first-out. Assume we push numbers in a stack as follows:

```
1 4 7 2 3 ,
```

then when we pop a number, we get **3** which is the last one pushed in. Similarly, we can also use the standard template library provided in C++. To use it, we need to include the library:

```
#include <stack>
```

The manner of defining a stack and the functions in a stack is similar to the usage of queue. A good reference can be found in `http://www.cplusplus.com/reference/stack/stack/`.

2.2 Depth First Search Algorithm

DFS in graph is very similar to that in a tree structure. In a tree, we start from the root *r*. We visit tree nodes all the way down to the bottom of the tree where we visit a leaf node of the tree *c*. Then we back to the leaf node's parent *p* and keep going down to the leaf node by starting at *p*'s other children.

In this case, we notice a pattern of a stack — when we go down from a starting node to the bottom of a tree, we push all the nodes in a stack in that order. Then after we reach the leaf node, we want to start with the node that's last pushed into the stack.

Using DFS in a graph has similar idea, and as in BFS, we also need a "visited" array to keep track of node visitings, or equivalently, use a variable `visited` in each node structure.

3. Exercise

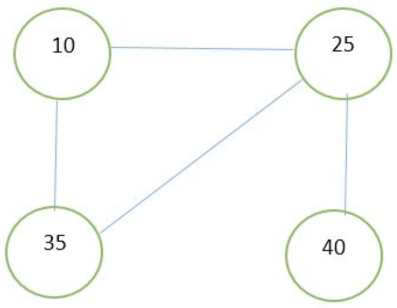
Use the starter code provided on Moodle to finish the following programming exercises. Show TA your work and get credit for this lab.

3.1 BFS. Write a method that prints the vertices in a graph visited in a breadth-first search (BFS) and the vertex with the maximum number of adjacent nodes.

```
void Graph::findNodeWithMaximumAdjacent(string);
```

The function should print the BFS sequence and the node that has the maximum number of adjacent nodes. The argument of the function is the starting point of BFS.

Example. For the following graph,



The expected output by calling `findNodeWithMaximumAdjacent("10")` should be:

```
10 25 35 40
25
```

The first line prints out the BFS traversal sequence, while the second line prints out the node(s) that has maximum number of adjacent nodes. When multiple nodes have same number of maximum number of adjacent nodes, print out all nodes.

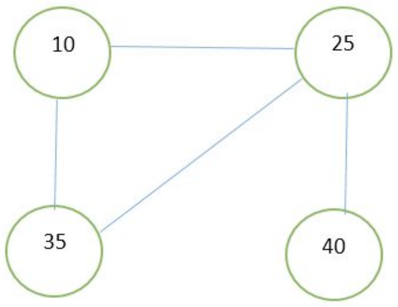
Hint. Use BFS algorithm to traverse the graph, while keep track of number of adjacent nodes for each node.

3.2 DFS. Write a function using the following prototype

```
void Graph::DFSpath(string src, string dst);
```

This function takes two strings as the source and destination nodes, and uses DFS to print out the path searched using DFS. Note that we assume the graph is fully connected.

Example. For the following graph,



The expected output by calling `DFSpath(“40”, “10”)` should be

```
40 --> 25 --> 10
```

Question. Can the output of DFSpath function be different given the same source and destination?