```
#ifndef _GRAPH_H
#define _GRAPH_H


#include <string>

#include "hmap.h"
#include "tvector.h"

/**
 * This Graph class uses an adjacency list representation
 * of a graph.  The lists of edges for a vertex are returned
 * to client programs in vectors rather than as linked lists
 *
 * All vertices are represented as strings.  Each string has
 * a corresponding number, and the relationship between
 * strings and numbers is obtained via functions:
 *
 *  o getNum(string) -- the number associated with a vertex string
 *  o getName(int)   -- the string associated with a number
 *
 *    Note: getName(getNum(s)) == s
 *    Time: getName and getNum are O(1) operations
 *
 * A list of vertices is accessible via getVertices
 * a copy of the names of all vertices is returned in a vector
 *
 * Similarly getAdjacent returns a list of vertices adjacent to
 * a named vertex. The names are returned, edges can be formed
 * in client programs from the names of ’to’ vertices
 *
 * Edges are added using addEdge(from,to)  which adds a directed edge.
 * Similarly, addUndirectedEdge adds an undirected edge which is
 * simulated in this class by adding two edges from−&gt;to and
 * to−gt;from.
 *
 * Edges can be removed by calling removeEdge(from,to).
 *
 * All edges can be removed by calling clear() [which doesn’t remove
 * vertices, but does remove all edges]
 *
 */

class Graph
{
  public:
    Graph();
    ~Graph();

    int  addVertex(const string& s);
    void addEdge(const string& from, const string& to);
    void addEdge(const string& from, const string& to, double weight);
    void addUndirectedEdge(const string& from, const string& to);
    void addUndirectedEdge(const string& from,
                           const string& to, double weight);

    void removeEdge(const string& from, const string& to);

    int    getNum(const string& s) const;
    string getName(int index)         const;
    int    vertexSize()            const;
    int    edgeSize()            const;
    double getWeight(const string& from, const string& to) const;
    double getWeight(int from, int to)      const;
    bool hasEdge(const string& from, const string& to) const;

    void getVertices(tvector<string>& list) const;
    void getAdjacent(const string& vertex, tvector<string>& list) const;
```

```
    void clear();

    struct Vertex
    {
        string name;
        double weight;
        explicit Vertex(const string& s="")   // default
            : name(s), weight(1.0)
        { }
        Vertex(const string& s, double w)
            : name(s), weight(w)
        { }
        bool operator < (const Vertex& rhs) const
        {
            return weight < rhs.weight;
        }
    };

  private:

    tvector<tvector<Vertex> > myAlist;
    HMap<int>                  myMap;
    int myVertexCount;
    int myEdgeCount;

    // disable copy/assignment
    Graph(const Graph& g);
    const Graph& operator = (const Graph& g);
};


#endif
```

```
#ifndef _GRAPHALG_H
#define _GRAPHALG_H


/**
 * Limited but useful class for doing graph algorithms.
 * Client classes subclass GraphAlg to get a generic class that has
 * vectors for distance, visited, path-stuff. This information
 * is accessible in protected vectors myDistance, myVisited, myPath,
 * respectively (note: no accessor/mutator function, access data
 * directly)
 *
 * The pure-virtual/abstract class processGraph is meant to be a
 * place-holder for generic graph algorithms, e.g., breadth-first
 * search, shortest-path, etc.
 *
 * Derived classes should override appInitialize, but
 * call GraphAlg::initialize() which will call the sub-class
 * application specific appInitialize, then initialize the vectors
 * of a graph.
 *
 * The initialize/appInitialize pair implement the GOF template pattern
 *
 * contract:
 *
 * initialize() -- initializes the graph to be used by processGraph
 *                 derived classes should call GraphAlg::initialize()
 *                 after constructing the  graph to ensure that
 *                 bookkeeping vectors are the right size
 *
 * author: Owen Astrachan
 *         April 16, 2000
 *         modified November 30, 2000
 */

#include "tgraph.h"
#include <cfloat>

class GraphAlg
{
  public:

    virtual ~GraphAlg() { }
    void initialize()
    // post: working area initialized
    {
        appInitialize();
        myState.resize(myGraph.vertexSize());
    }

    virtual void processGraph() = 0;    // do something to a graph

  protected:

    virtual void appInitialize() = 0;   // application specific init

    Graph  myGraph;

    static const double INFINITY = DBL_MAX;
    struct VertexState
    {
        int    path;
        double distance;
        bool   visited;
        VertexState()
            : path(-1),
              distance(INFINITY),
              visited(false)
        { }
```

```
        VertexState(int p, double d, bool b)
            : path(p),
              distance(d),
              visited(b)
        { }
    };

    tvector<VertexState> myState;
};


#endif
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>  // for sort
using namespace std;

#include "tvector.h"
#include "tmatrix.h"
#include "tqueue.h"
#include "sortall.h"  // for bsearch

/**
 * This version of wordladder doesn’t use a graph
 * abstraction, but shows how to do a breadth first
 * search on a raw adjacency−matrix version of a graph
 *
 * First, create graph of word−ladders from file.
 * File read should consist of unique wordList (e.g., a dictionary)
 *
 * Each word is added to a graph, and edges from the word
 * to all other wordList "one−away" are created. Here one away
 * means change one letter of a word and get another word
 * e.g., slat −> slot −> plot −> plow −> flow −> glow
 *
 * the user is prompted for a word, and a breadth first
 * search from the user−entered word is performed
 *
 * this means finding a word ladder from −> to is done by
 * starting at ’to’ and going backwards to ’from’
 *
 * author: Owen Astrachan
 * 4/8/2002
 * modified 4/6/2003 to turn code into a class
 */


class LadderMatrix
{
  public:
    LadderMatrix();
    void makeGraph(ifstream& input);
    void doSearch(const string& from, const string& to);

  private:
    void clear();
    void printLadder(const string& to,
                     const tvector<int>& path);
    tmatrix<bool>   myGraph;
    tvector<string> myWordList;

};

LadderMatrix::LadderMatrix()
{

}

void LadderMatrix::clear()
// post: no edges in graph of words
{
    for(int j=0; j < myGraph.numrows(); j++) {
        for(int k=0; k < myGraph.numcols(); k++){
            myGraph[j][k] = false;
        }
    }
}

void LadderMatrix::makeGraph(ifstream& input)
// post: edges connecting all words in input are created
```

```
//       where connection means words are one away
{

    // read words, store, make graph have connections
    string word;
    while (input >> word) {
        myWordList.push_back(word);
    }
    sort(myWordList.begin(), myWordList.end());

    // graph initially has no edges

    myGraph.resize(myWordList.size(), myWordList.size());
    clear();

    // now connect edges where words are one−away
    for(int k=0; k < myWordList.size(); k++) {
        string s = myWordList[k];

        // for every char in word, change to all chars ’a’ .. ’z’

        for(unsigned j=0; j < s.length(); j++) {
            string copy = s;
            for(char ch = ’a’; ch <= ’z’; ch++) {
                if (ch != s[j]) {
                    copy[j] = ch;
                    int index = bsearch(myWordList, copy);
                    if (index != −1) {
                        myGraph[k][index] = true;
                    }
                }
            }
        }
        if (k % 100 == 0) {
            cout << k << " out of " << myWordList.size() << endl;
        }
    }
}

void LadderMatrix::doSearch(const string& from, const string& to)
// post: ladder from −> to found/printed if it exists
{

    // visited vector tells whether vertex/int has been seen before
    // if visited[k] == true, path[k] is vertex/how we got to visit

    tvector<int> path(myWordList.size(),−1);
    tvector<bool> visited(myWordList.size(), false);

    int fromIndex = bsearch(myWordList, from);
    int toIndex = bsearch(myWordList, to);
    if (fromIndex == −1 || toIndex == −1) {
        cout << "one of " << from << " " << to
             << " not found" <<endl;
        return;
    }

    tqueue<string> q;
    string word;
    q.enqueue(from);
    visited[fromIndex] = true;

    while (! q.isEmpty()) {
        q.dequeue(word);
        if (word == to) {
            cout << "found ladder" << endl;
            printLadder(to,path);
            return;
        }
        int index = bsearch(myWordList, word);
        for(int k=0; k < myWordList.size(); k++) {
            if (myGraph[index][k] && ! visited[k]) {
                q.enqueue(myWordList[k]);
                visited[k] = true;
                path[k] = index;
            }
        }
    }
    cout << "ladder not found" << endl;
}

void LadderMatrix::printLadder(const string& to,
                              const tvector<int>& path)
{
    int index = bsearch(myWordList,to);
    while (index != -1) {
        cout << myWordList[index] << endl;
        index = path[index];
    }
}

int main(int argc, char * argv[])
{
    string filename;
    if (argc > 1) {
        filename = argv[1];
    }
    else {
        cout << "filename: ";
        cin >> filename;
    }
    ifstream input(filename.c_str());
    LadderMatrix ladder;
    ladder.makeGraph(input);

    string from, to;
    while (true) {
        cout << "words ";
        cin >> from >> to;
        ladder.doSearch(from,to);
    }
}
```

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

#include "tvector.h"
#include "tgraph.h"
#include "tqueue.h"

/**
 * This version of wordladder uses the Graph class
 * in tgraph.h which uses an adjacency list backed
 * by a hashmap for int->string vertex/name conversion
 * (and back again)
 *
 * First, create graph of word-ladders from file.
 * File read should consist of unique wordList (e.g., a dictionary)
 *
 * Each word is added to a graph, and edges from the word
 * to all other wordList "one-away" are created. Here one away
 * means change one letter of a word and get another word
 * e.g., slat -> slot -> plot -> plow -> flow -> glow
 *
 * the user is prompted for a word, and a breadth first
 * search from the user-entered word is performed
 *
 * this means finding a word ladder from -> to is done by
 * starting at 'to' and going backwards to 'from'
 *
 * author: Owen Astrachan
 * 4/7/2003
 */


class LadderGraph
{
  public:
    LadderGraph();
    void makeGraph(ifstream& input);
    void doSearch(const string& from, const string& to);

  private:
    void breadth(const string& from, const string& to);

    void printLadder(const string& to,
                     const tvector<int>& path);

    Graph *         myGraph;
};

LadderGraph::LadderGraph()
    : myGraph(new Graph())
{

}

void LadderGraph::makeGraph(ifstream& input)
// post: edges connecting all words in input are created
//       where connection means words are one away
{

    // read words, store, make graph have connections
    string word;
    while (input >> word) {
        myGraph->addVertex(word);
    }

    // now connect edges where words are one-away
```

```

    for(int k=0; k < myGraph->vertexSize(); k++) {

        string s = myGraph->getName(k);

        // for every char in word, change to all chars 'a' .. 'z'

        for(unsigned j=0; j < s.length(); j++) {
            string copy = s;
            for(char ch = 'a'; ch <= 'z'; ch++) {
                if (ch != s[j]) {
                    copy[j] = ch;
                    int index = myGraph->getNum(copy);
                    if (index != -1) {
                        myGraph->addEdge(s,copy);
                    }
                }
            }
        }
        if (k % 100 == 0) {
            cout << k << " out of " << myGraph->vertexSize() << endl;
        }
    }
}

void LadderGraph::doSearch(const string& from, const string& to)
// post: ladder from -> to found/printed if it exists
{
    int fromIndex = myGraph->getNum(from);
    int toIndex   = myGraph->getNum(to);
    if (fromIndex == -1 || toIndex == -1) {
        cout << "one of " << from << " " << to
             << " not found" <<endl;
        return;
    }
    breadth(from,to);

}

void LadderGraph::breadth(const string& from, const string& to)
{
    // visited vector tells whether vertex/int has been seen before
    // if visited[k] == true, path[k] is vertex/how we got to visit

    tvector<int> path(myGraph->vertexSize(),-1);
    tvector<bool> visited(myGraph->vertexSize(), false);
    int fromIndex = myGraph->getNum(from);

    tqueue<string> q;
    q.enqueue(from);

    visited[fromIndex] = true;

    string word;
    while (! q.isEmpty()) {
        q.dequeue(word);
        if (word == to) {
            cout << "found ladder" << endl;
            printLadder(to,path);
            return;
        }
        tvector<string> adj;
        myGraph->getAdjacent(word,adj);
        int wIndex = myGraph->getNum(word);

        for(int k=0; k < adj.size(); k++) {
            int kIndex = myGraph->getNum(adj[k]);
            if (! visited[kIndex]) {
                q.enqueue(adj[k]);
                visited[kIndex] = true;
                path[kIndex] = wIndex;
            }
        }
    }
    cout << "ladder not found" << endl;
}

void LadderGraph::printLadder(const string& to,
                              const tvector<int>& path)
{
    int index = myGraph->getNum(to);
    while (index != -1) {
        cout << myGraph->getName(index) << endl;
        index = path[index];
    }
}

int main(int argc, char * argv[])
{
    string filename;
    if (argc > 1) {
        filename = argv[1];
    }
    else {
        cout << "filename: ";
        cin >> filename;
    }
    ifstream input(filename.c_str());
    LadderGraph ladder;
    ladder.makeGraph(input);

    string from, to;
    while (true) {
        cout << "words ";
        cin >> from >> to;
        ladder.doSearch(from,to);
    }
}
```

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#include "depth.h"
#include "makeladder.cpp"

int main(int argc, char * argv[])
{
    string filename;
    if (argc > 1) {
        filename = argv[1];
    }
    else {
        cout << "filename: ";
        cin >> filename;
    }
    ifstream input(filename.c_str());
    DepthFirst depthalg;
    Graph * graph = makeLadderGraph(input);
    depthalg.setup(graph);

    int total = 0;
    for(int k=0; k < graph->vertexSize(); k++){
        int count= depthalg.depth(k);
        if (count != 0){
            cout << count << "\t" << graph->getName(k) << endl;
            total++;
        }
    }
    cout << "total # components = " << total << endl;
}
```

```
#ifndef _DEPTH_H
#define _DEPTH_H

#include "graphalgo.h"
#include "tstack.h"

class DepthFirst : public GraphAlgorithm
{
  public:

    int depth(int vertex){
        int count = doDepth(vertex);
        return count;
    }
  protected:
    int doDepth(int vertex){

        int count = 0;
        tstack<int> st;
        st.push(vertex);
        markVisited(vertex);

        while (st.size() > 0){

            st.pop(vertex);
            tvector<string> adj;
            string name = myGraph->getName(vertex);
            myGraph->getAdjacent(name,adj);

            for(int k=0; k < adj.size(); k++){
                int num = myGraph->getNum(adj[k]);
                if (! isVisited(num)) {
                    markVisited(num);
                    st.push(num);
                    count++;
                }
            }
        }
        return count;
    }

};


#endif
```