```cpp
#include "library.h"
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <queue>
#include <limits>
#include <cmath>
#include <assert.h>


using namespace std;

struct place {
	int code;
	string state;
	string name;
	int pop; //population
	double area;
	double lat; //latitude
	double lon; //longitude
	int inter; //intersection code
	double dist; //distance to intersection
	place(int c, string s, string n, int p, double a, double la, double lo, int in, double d) {
		code = c;
		state = s;
		name = n;
		pop = p;
		area = a;
```

```
        lat = la;
        lon = lo;
        inter = in;
        dist = d;
    }
};
struct ht_items {
    string* key; //key will be place name
    place* data; //data will be all the data associated to that name
    ht_items(place* p) {
        key = &p->name;
        data = p;
    }
};
class LL {
protected:
    struct Link {
        ht_items* item;
        Link* next;
        Link(ht_items* i, Link* n) {
            item = i;
            next = n;
        }
    };
    Link* first, * last;
    int size;
public:
    LL() {
        first = NULL;
```

```
        last = NULL;
        size = 0;
}
void add_to_front(ht_items* x) {
        first = new Link(x, first);
        if (last == NULL) {
                last = first;
        }
        size++;
}
void add_to_back(ht_items* x) {
        if (last != NULL) {
                last->next = new Link(x, NULL);
                last = last->next;
        }
        else {
                last = new Link(x, last);
                first = last;
        }
        size++;
}
place* find(string name, string state) { //retuns the named place
        place* p = NULL;
        Link* ptr = first;
        while (ptr != NULL) {
                if ((ptr->item->data->name == name) && (ptr->item->data->state == state)) {
                        p = ptr->item->data;
                        break;
                }
```

```
                ptr = ptr->next;
            }
            return p;
        }
        vector<place*> find(string name) { //retuns vector of all same named places
            vector<place*> solution;
            place* p = NULL;
            Link* ptr = first;
            while (ptr != NULL) {
                if (ptr->item->data->name == name) {
                    p = ptr->item->data;
                    solution.push_back(p);
                }
                ptr = ptr->next;
            }
            return solution;
        }
};
class HT {//Hash table
private:
        LL item_list[30000]; //anything higher exceeds stack size of compiler (Visual Studio)
        int size;
        int count;
        int hash_function(string* key) {
            const int p = 53;
            const int m = 30000;
            long hash_value = 0;
            long p_pow = 1;
            for (char c : *key) {
```

```
                hash_value = (hash_value + (c - '\'' + 1) * p_pow) % m;
                p_pow = (p_pow * p) % m;
            }
            //cout << "key: " << key << " hash: " << hash_value << endl;
            return hash_value;
        }
public:
        HT() {
            size = 30000;
            count = 0;
        }
        void add(place* p) {
            ht_items* ptr = new ht_items(p);
            int position = hash_function(&ptr->data->name);
            item_list[position].add_to_back(ptr);
            count++;
        }
        void readFile() {
            ifstream fin;
            fin.open("NamedPlaces318.txt");
            if (fin.fail()) {
                cout << "Unable to open NamedPlaces318.txt" << endl;
                return;
            }

            while (!fin.eof()) {
                place* p;
                string buf;
```

```
getline(fin, buf);

string tempCode(buf, 0, 8);
string tempState(buf, 8, 2);
string tempNamePopMix(buf, 10, 56);
string tempArea(buf, 66, 14);
string tempLat(buf, 80, 10);
string tempLon(buf, 90, 11);
string tempInter(buf, 101, 5);
string tempDist(buf, 106, 8);

string code = "";
string state = "";
string name = "";
string pop = "";
string area = "";
string lat = "";
string lon = "";
string inter = "";
string dist = "";

for (int i = 0; i < tempCode.size(); i++) {
        if (tempCode.at(i) != ' ') {
                code += tempCode.at(i);
        }
}
for (int i = 0; i < tempState.size(); i++) {
        if (tempState.at(i) != ' ') {
                state += tempState.at(i);
```

```
            }
        }
        for (int i = 0; i < tempNamePopMix.size(); i++) {
            if (tempNamePopMix.at(i) >= '0' && tempNamePopMix.at(i) <= '9') {
                pop += tempNamePopMix.at(i);
            }
        }
        for (int i = 0; i < tempNamePopMix.size() - 1; i++) {
            //can't be a number or double space, also there will always be a space
at the end of the name if using "NamedPlaces318.txt"
            if (!(tempNamePopMix.at(i) == ' ' && tempNamePopMix.at(i + 1) == ' ')
&& !(tempNamePopMix.at(i) >= '0' && tempNamePopMix.at(i) <= '9')) {
                name += tempNamePopMix.at(i);
            }
        }
        name.erase(name.begin() + name.size() - 1); // to get rid of the extra space at
end
        for (int i = 0; i < tempArea.size(); i++) {
            if (tempArea.at(i) != ' ') {
                area += tempArea.at(i);
            }
        }
        for (int i = 0; i < tempLat.size(); i++) {
            if (tempLat.at(i) != ' ') {
                lat += tempLat.at(i);
            }
        }
        for (int i = 0; i < tempLon.size(); i++) {
            if (tempLon.at(i) != ' ') {
                lon += tempLon.at(i);
```

```
                }
            }
            for (int i = 0; i < tempInter.size(); i++) {
                if (tempInter.at(i) != ' ') {
                    inter += tempInter.at(i);
                }
            }
            for (int i = 0; i < tempDist.size(); i++) {
                if (tempDist.at(i) != ' ') {
                    dist += tempDist.at(i);
                }
            }

            p = new place(stoi(code), state, name, stoi(pop), stod(area), stod(lat), stod(lon),
stoi(inter), stod(dist));

            add(p);
        }

        fin.close();

    }
    place* find(string name, string state) {
        int position = hash_function(&name);
        return item_list[position].find(name, state);
    }
    vector<place*> find(string name) {
        int position = hash_function(&name);
        return item_list[position].find(name);
    }
```

```
};

int LOCATION_TOTAL = 0;
struct connection {
	string name, code;
	int isecA, isecB; //intersection A and B
	double length; //in miles
	connection(string n, string c, int a, int b, double l) {
		name = n;
		code = c;
		isecA = a;
		isecB = b;
		length = l;
	}
};
struct intersection {
	string state, name; //nearest named place name and state
	double lon, lat, dist; // distance from nearest named place
	intersection(string st, string na, double lo, double la, double dis) {
		state = st;
		name = na;
		lon = lo;
		lat = la;
		dist = dis;
	}
};
struct node {//equivalent to location
	vector<connection*> v_conn_data;
	intersection* inter_data;
```

```
	int number;

	/// used for dijkstras algorithm
	double spe = numeric_limits<double>::infinity();
	node* predecessor = NULL;
	bool visited = false;
	bool mpqAdded = false;
	///

	node(intersection* inter, int num) {
		inter_data = inter;
		number = num;
	}

};
struct graph {
	vector<node*> v_nodes;
};

//<helperFunctions>
node* returnNodePointer(graph* g, int locationCode) {
	if (locationCode >= 0 && locationCode < g->v_nodes.size()) {
		return g->v_nodes.at(locationCode);
	}
	return NULL;
}
bool checkNumber(string str) {
	for (int i = 0; i < str.length(); i++) {
		if (!isdigit(str[i])) {
```

```
                return false;
            }
        }
        return true;
}
connection* connectionBetween(node* a, node* b) {
        if (a == NULL || b == NULL) {
            return NULL;
        }
        for (int i = 0; i < a->v_conn_data.size(); i++) {
            if (a->v_conn_data.at(i)->isecB == b->number) {
                return a->v_conn_data.at(i);
            }
        }
        return NULL;
}
string CardinalAndOrdinalDirectionsAtoB(node* a, node* b) {

        double pi = 3.14159265;
        double x = b->inter_data->lon - a->inter_data->lon;
        double y = b->inter_data->lat - a->inter_data->lat;
        double r = sqrt(x * x + y * y);
        double theta;

        if (y >= 0 && r != 0) {
            theta = acos(x / r);
        }
        if (y < 0) {
            theta = -acos(x / r);
```

```
}
if (r == 0) {
	return "ERROR";
}


if (theta >= 0 && theta <= pi / 8) {
	return "E";
}
else if (theta > pi / 8 && theta < 3 * pi / 8) {
	return "NE";
}
else if (theta >= 3 * pi / 8 && theta <= 5 * pi / 8) {
	return "N";
}
else if (theta > 5 * pi / 8 && theta < 7 * pi / 8) {
	return "NW";
}
else if (theta >= 7 * pi / 8 && theta <= pi) {
	return "W";
}

else if (theta <= 0 && theta >= -pi / 8) {
	return "E";
}
else if (theta < -pi / 8 && theta > -3 * pi / 8) {
	return "SE";
}
else if (theta <= -3 * pi / 8 && theta >= -5 * pi / 8) {
```

```
		return "S";
	}
	else if (theta < -5 * pi / 8 && theta > -7 * pi / 8) {
		return "SW";
	}
	else if (theta <= -7 * pi / 8 && theta >= -pi) {
		return "W";
	}

	return "DNE ERROR";
}
//</helperFunction>

//<graphModifiers>
void createGraphIntersections(graph& g) {
	ifstream interIN;
	interIN.open("intersections.txt");
	if (interIN.fail()) {
		cout << "unable to open intersections.txt" << endl;
		exit(1);
	}

	string s, n; double lo, la, d;
	string name; node* nptr; intersection* iptr;

	while (!interIN.eof()) {
		interIN >> lo >> la >> d >> s;
		getline(interIN, name);
		name.erase(0, 1); //removes space from beginning of name
```

```
        iptr = new intersection(s, name, lo, la, d);

        nptr = new node(iptr, LOCATION_TOTAL);//location 0 is "No Such Place Exists"

        g.v_nodes.push_back(nptr);

        LOCATION_TOTAL++;
    }
    assert(!interIN.fail());

    interIN.close();
}
void connectGraph(graph& g) {
    ifstream conIN;
    conIN.open("connections.txt");
    if (conIN.fail()) {
        cout << "unable to open connections.txt" << endl;
        exit(1);
    }

    string N, C; int A, B; double L; connection* cptrA, * cptrB;

    while (!conIN.eof()) {
        conIN >> N >> C >> A >> B >> L;
        cptrA = new connection(N, C, A, B, L);
        g.v_nodes.at(A)->v_conn_data.push_back(cptrA);

        cptrB = new connection(N, C, B, A, L);
```

```
		g.v_nodes.at(B)->v_conn_data.push_back(cptrB);
	}
	assert(!conIN.fail());

	conIN.close();

}
//</graphModifiers>

void dijkstrasAlgorithm(graph* g, int SourceNodeCode) {
	/// This function implements Dijkstra's shortest path algorithm
	/// See Introduction to Algorithms 4th edition pg 620 for reference
	/// All page references shown in this program will refer to this book

	//NOTE: pirority queues assume their content will remain constant
	//priority_queue<node*, vector<node*>, compare> mpq;//adds complexities and longer
runtime when implemented
	vector<node*> mpq; //minimum priority queue, will use a vector as the minimum priority
queue's container

	/// "INITIALIZE-SINGLE-SOURCE"
	node* source = returnNodePointer(g, SourceNodeCode);
	if (source != NULL) {
		source->spe = 0;
		source->visited = true;
		source->mpqAdded = true;
		mpq.push_back(source);
	}
	else {
		cout << "\nError dijkstrasAlgorithm: Source Uninitialized\n" << endl;
```

```
            return;
        }
        ///

        /// "INSERT"
        for (int i = 0; i < g->v_nodes.size(); i++) {
            if (g->v_nodes.at(i) != source) {//source already inserted, we dont want a duplicate
                g->v_nodes.at(i)->visited = false;
                g->v_nodes.at(i)->mpqAdded = false;
                g->v_nodes.at(i)->spe = numeric_limits<double>::infinity(); //reinitialize every
node except source to infinity,

                                    //needed when dijkstras algorithm used more than once
                //mpq.push_back(g->v_nodes.at(i)); //insert all other nodes
            }
        }
        ///

        string sfoo = "foo"; double dfoo = -1;
        intersection* ifoo = new intersection(sfoo, sfoo, dfoo, dfoo, dfoo);
        node* temp = new node(ifoo, dfoo);
        temp->spe = numeric_limits<double>::infinity();
        while (!mpq.empty()) {
            //cout << mpq.size() << endl;
            //cout << mpqTotalUnvisited << endl;
            /// "EXTRACT-MIN"
            int selected = -1;
            node* minimum = temp;
            for (int i = 0; i < mpq.size(); i++) {
                if ((minimum->spe > mpq.at(i)->spe)) {
```

```
                minimum = mpq.at(i);
                selected = i;
            }
        }

        //in the while loop the invariant Q = V - S must be kept, this is done by keeping track which nodes have been visited and used as minimum
        if (minimum != temp) {
            minimum->visited = true;
        }
        else {
            cout << "temp error" << endl;
            continue;
        }
        if (selected == -1) {
            cout << "selected error" << endl;
            continue;
        }
        else {
            mpq.erase(mpq.begin() + selected);
        }
        ///

        for (int i = 0; i < minimum->v_conn_data.size(); i++) { //for each node adjacent to minimum that has not already been visited
            /// preliminary work for "RELAX" function
            node* nodeA = minimum;
            node* nodeB = returnNodePointer(g, minimum->v_conn_data.at(i)->isecB);
            connection* connData = connectionBetween(nodeA, nodeB);
```

```
                    double weight = connData->length; // the "weight" of a connection is just its
length

                    ///

                    if (nodeB != NULL) {

                        /// "RELAX"

                        if (nodeB->spe > (nodeA->spe + weight)) {

                            nodeB->spe = (nodeA->spe + weight);

                            nodeB->predecessor = nodeA;

                            //cout << nodeB->number << "-" << nodeB->inter_data->name
<< " predecessor = " << nodeA->number << "-" << nodeA->inter_data->name << endl;

                        }

                        if (!nodeB->visited && !nodeB->mpqAdded) {

                            nodeB->mpqAdded = true;

                            mpq.push_back(nodeB);

                        }

                        ///

                    }

                    /// "DECREASE-KEY" pg 173

                    //this is done automatically in the "RELAX" function

                    //theres no need to write anything here only because we are using a vector,
where the contents can be dynamic (at least what they point to)

                    //we would have to implement an actual "DECREASE-KEY" function if we used a
priority queue implemented from a c++ library (with its added complexities)

                    ///

                }

            }

            delete ifoo;
```

```
	delete temp;

}
vector<node*> shortestPath(graph* g, int nodeAcode, int nodeBcode) {
	dijkstrasAlgorithm(g, nodeAcode);
	node* nodeA = returnNodePointer(g, nodeAcode);
	node* nodeB = returnNodePointer(g, nodeBcode);
	vector<node*> empty;
	vector<node*> path;

	if (nodeA == NULL || nodeB == NULL) {
		return empty;
	}

	node* temp = nodeB;
	while (true) {
		if (temp == NULL) {
			return empty;
		}
		if (temp == nodeA) {
			path.push_back(temp);
			return path;
		}
		path.push_back(temp);
		temp = temp->predecessor;
	}
}
void coutShortestPath(graph* g, int pointANodeCode, int pointBNodeCode) {
	vector<node*> v = shortestPath(g, pointANodeCode, pointBNodeCode);
```

```
	if (v.size() == 0) {
		cout << "No path Exists" << endl;
	}
	for (int i = v.size(); i > 1; i--) {
		node* tempA = v.at(i - 1);
		node* tempB = v.at(i - 2);
		connection* connData = connectionBetween(tempA, tempB);
		cout << "from intersection " << tempA->number;
		cout << " take " << connData->name << " " << connData->length << " miles " <<
CardinalAndOrdinalDirectionsAtoB(tempA, tempB) << " to intersection " << tempB->number << endl;
	}


}
string findMap(node* nodeA, node* nodeB) {
	ifstream fin;
	fin.open("coverage.txt");
	if (fin.fail()) {
		printf("unable to open coverage.txt\n");
		exit(1);
	}
	int top, bottom, left, right;
	string map;
	while (!fin.eof()) {
		fin >> top >> bottom >> left >> right >> map;
		if (((nodeA->inter_data->lat < top) && (nodeA->inter_data->lat >= bottom) && (nodeA-
>inter_data->lon >= left) && (nodeA->inter_data->lon < right))
			&& ((nodeB->inter_data->lat < top) && (nodeB->inter_data->lat >= bottom) &&
(nodeB->inter_data->lon >= left) && (nodeB->inter_data->lon < right))) {
			return map;
```

```
		}
	}
	return "";
	fin.close();
}
void titlePage() {

	cout << "**************************************************" << endl;
	cout << "|                                                 |" << endl;
	cout << "|            Graphical Shortest Path              |" << endl;
	cout << "| By: Brandon Rubio, ECE 318, University of Miami |" << endl;
	cout << "|                                                 |" << endl;
	cout << "**************************************************" << endl;
	cout << endl;
}

void main() {
	titlePage();
	printf("Loading...\n");
	HT HashTable;
	HashTable.readFile();
	graph G;
	createGraphIntersections(G);
	connectGraph(G);

	string inputA, inputB;
	while (true) {
		bool inputAisInt = false;
		bool inputBisInt = false;
```

```
cout << "Enter two intersection numbers" << endl;
cout << "Intersection A: ";
cin >> inputA;
if (inputA == "exit") {
	cout << "exiting program" << endl;
	break;
}
else if (!checkNumber(inputA)) {
	cout << "Please enter a positive integer" << endl;
}
else {
	inputAisInt = true;
	cout << "Intersection B: ";
	cin >> inputB;
	if (inputB == "exit") {
		cout << "exiting program" << endl;
		break;
	}
	else if (!checkNumber(inputB)) {
		cout << "Please enter a positive integer" << endl;
	}
	else {
		inputBisInt = true;
	}
}
if (inputAisInt && inputBisInt) {
	int stoiA = stoi(inputA);
	int stoiB = stoi(inputB);
	cout << "Searching.." << endl;
```

```
                        string map = findMap(returnNodePointer(&G, stoiA), returnNodePointer(&G,
stoiB));
                        if (map == "") {
                                printf("No map found\n");
                                exit(1);
                        }
                        else {


                        }
                        coutShortestPath(&G, stoiA, stoiB);
                }
        }

}
```