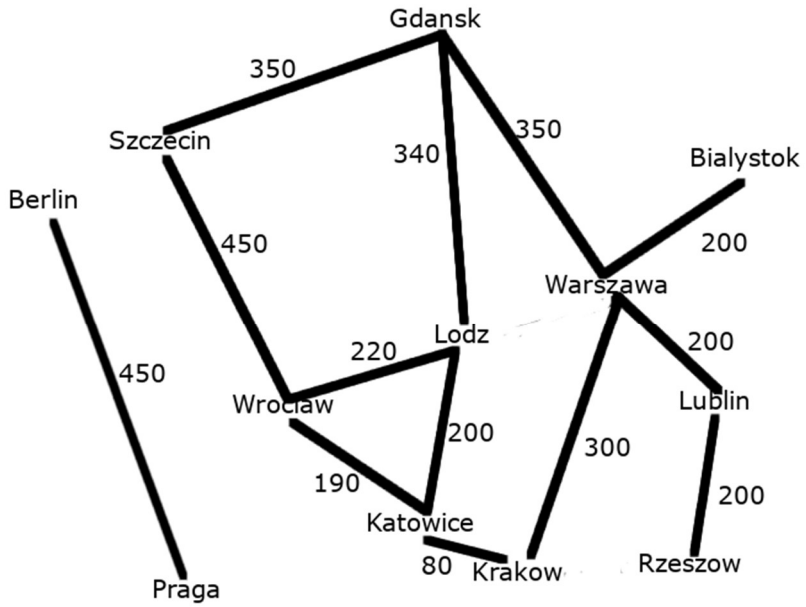
The Map, “Story”

The Map is library that allows you to create a map of cities, for example map of cities in Poland. The user will be able to add new Cities to the map and link some cities, specifying the distance between them. Map will allow you to check how to get from one city to another. For example, if you check how to get from Krakow to Warsaw, you may see that the best route is go to Katowice, then to Łodź and then to Gdansk.

The Map library is designed to use when map is modified rarely, but you very often check how to get from one city to another. After modification of Map (adding a city or changing a distance between two cities) user, to get the correct best path from one city to another, will have to explicitly call function “recalculate”. Function recalculate finds the best path from every city in the map to every other city in the map (if any connection between two cities is possible). After recalculation each city remembers how to get to every other city (if any path is possible, cities can be in two disconnected graphs. On the beginning function “recalculate” force each city in the map to forget all it's connections.Let’s consider such map



(map1).

If there are at least two convenient paths connecting two cities, after recalculation each of this two cities will know the best and second the best path to other city. So in the case of map1 after recalculation city “Gdansk” will know that the best city to Krakow is through Lodz and Katowice, and second the best path to Krakow is through Warszawa. City “Krakow” will also know two routes to Gdansk (same as from Gdansk to Krakow reversed).

If there is only one path connecting two cities then each of those two cities will know this path and will know that this is the only convenient path. So in case of map1 city

“Rzeszow”, after recalculation, will know that the only convenient path to Bialystok is through Lublin and Warszawa. City “Bialystok will know know that the only convenient path to Rzeszow is through Warszawa and Lublin.

If there is no convenient way from two cities each of those two cities will know that there in so way to get to other city.

After recalculation user may call function “how_to_get”, passing names of two cities as arguments, to check how to get from one city to another.

```
bool how_to_get(std::string from, std::string destination, std::ostream& stream)
```

If any of two passed cities is not on the map fuction ”how_to_get” will print this information.

If there is no way to get from one city to another fuction ”how_to_get” will print this information.

If there is only one convenient path from one city to another, fuction ”how_to_get” will print this path and print that this is the only possible path.

If there are at least two possible paths connecting those two cities fuction ”how_to_get” will print those path as best_path and second_best_path.

User is building a map by calling a function “link” (bool link(std::string city1_name, std::string city2_name, unsigned int distance)). If any of passed cites is not yet on the map, function first creates new city on new map with given name. If cities were unlink before (one wasn’t a neighbour of other) function links them (adds second city to list of neighbours of city1 and vice versa). If cities were linked before function reset distance between them, so user can use this function to change distance between cities. Function returns true if new link was created, false otherwise.

User can unlink two cities using function “unlink” (bool unlink(std::string city1_name, city2_name);). Function returns false and do nothing if one of the cities wasn’t on the map, or if cities wasn’t linked. If given cities were linked, function unlinks them and returns true. Unlinking means removing first city of neighbours list of second city and removing second city from neighbours list of first city.

The Map “Case study”

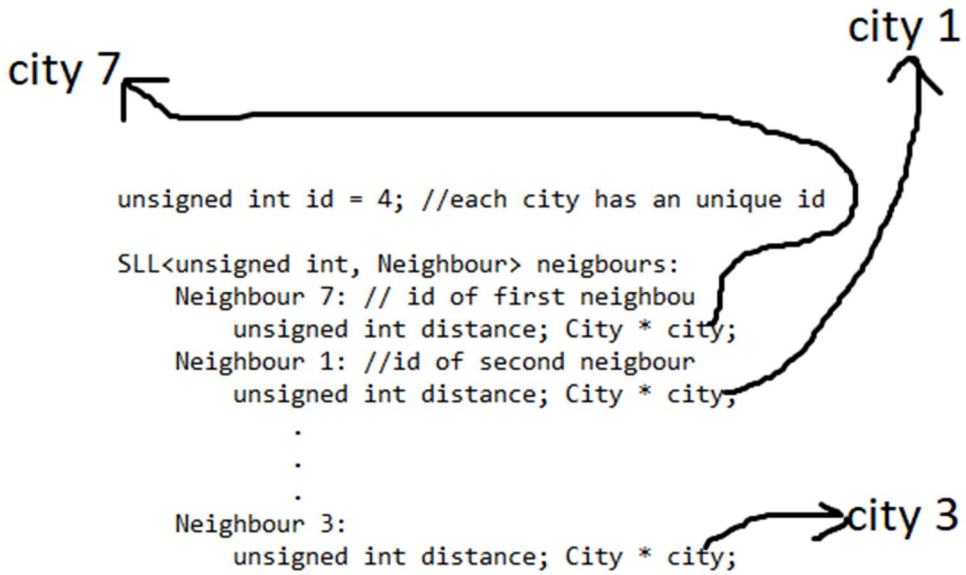
How recalculation is done:

Calculation is done by “sending paths” from on city to another. Path is represented by object of class Path. Path contains array of id’s of cities which are in the path and distances between them. Let’s consider map1. Let’s give some cities an id: Bialsytok is nr 1, Warszawa, Lublin, Krakow, and Gdansk are in order 2,3,4,5. Firstly city nr1 (Bialystok) send path which contains id 1 and distance 200 to it’s only neighbour, city nr 2 (Warszawa). City nr 2 then checks on the connection table. If it is necessary it adds path to it’s routing table. Then city nr 2 sends to all it's neighbour the copy of the path with addition of neighbour id and distance between them. So it’s send path with id 1, distance 200, id 4, distance 300 to Kraków and so on… Each city that receives the path check it’s own connection table and if necessary adds path to the connection table. Then it send the copy of the path with addition of neighbour id and distance to **each neighbour which isn’t already in the path. This way no loops are created.**

After this each city know the best path (or two path) to Bialystok. Then operation is repeated, but now starting from city nr2, then from city nr3… Because of many operations recalculation of maps is time-consuming.

Important note: Paths are send as pointers to Path objects. The city that receive the path is responsible to delete this path, if necessary.

Program structure:




```
SLL<unsigned int; Connection> connections:
    Connection 1: //connection to city of id 1
        Path* best_path; Path* second_best_path;
    Connection 5:
        Path* best_path; Path* second_best_path;
            .
            .
            .
    Connection 2:
        best path; second_best_path;
    //if ther is no possible path to other city this city is simply not
    //in the connection table. IF there is only one route to other city,
    //second path is nullptr. There is no connection in whole connection
    //table whose best_path is nullptr
```

A member of class Map object is single linked list of keys type std::string (city names) and value types City.

The Map, c++ code.

Here are all headers files with comments describing all function.

Note: There is a need to change some nonstatic methods to constant method and change type of some arguments to constant.

Class Path:

```
#ifndef PATH_H
#define PATH_H
#include <iostream>
#include <vector>

class Path{
public:
	Path(unsigned int id, unsigned int distance); //creates a new path with one city and one distance (hop number is 1)
	Path(const Path* source); // This constcructor creates a deep copy of a path.
	~Path(); //free memory alocated for this path;

	bool is_in(unsigned int id); //checks if city of passed id is in the path. Returns true if city is in the path, false otherwise.
	Path* add(unsigned int id, unsigned int distance);
	/*	Creates a copy of path with addition of passed id and distance.
		Dynamically creates a new path of hops nr greater by one than itself's hops number.
		First n cities id and first n distances are copied from this object to newly created object.
		n-th+1 id of newly creted object is passed id/
		n-th+1 distance of newly created distance is passed distances.
		This function does not frees memmory allocated for this Path!s
	*/
	void print(std::ostream& my_stream, std::vector<std::string> & names);
	/*	Print the path with cities names and distances beetween them.
		Arguments:
			std::ostream& my_stream:
				The stream on which path will be printed (like std::cout).
			std::vector<std::string> & names:
				Because of the fact that Path knows only ids of the city, no names while printing
				path object we need to some howe tell what is the name of each city in the path.
				Names is vector which indices are cities ids and values are cities names. For example
				name of city of id 7 is names[7].
		This function prints path in reversed order. If city nr 9 receives path to city nr 4 the first city on the path
		is city nr 4. If the user want's to know how to get from city nr 9 to city nr 4 it wants output it to be in order.
		The exemplatory output on stream from this function may be:
		--> Katowice (80) --> Lodz (200) --> Gdansk (340)
	*/
	unsigned int from(); //returns an id of origin city of the path (an id of first city on the path)
	unsigned int get_length(); //return the total length of the path, the sum of all distances in the path.

private:
	unsigned int n; //hops numober (how many cities are in the path)
	unsigned int* the_path;
		/* Variable the_path is one deminsional array. The length of this array is always 2 times n.
```

```
            In this array bu turns are id's of cities and distances between
them.
            So for example if n is equal 4 (there are four cities in the array,
the array may look like this:)
            [4, 234, 2, 450, 7, 750, 13, 123].
            It means that the origin of path is city nr 4, the distance betweend
city 4 and city 2 is 450, second
            city on the path is city nr 2 and so on. This path may be received
by for example city nr 9. This way sity nr 9 knows
            how to get to city nr 2. Because each city knows it's own id there
is no need for destinaton id.
            */

      unsigned int length; // the total length of the path, the sum of all
distances in the path.
      Path(unsigned int* the_path, unsigned int n, unsigned int length);
      /* Creates a new Path with given values.
      NOTE: THIS concructor does not copy context of the_path array to it's owna
the path_array, it's just set
      own's the_path pointer to be same as the_path agument.
      */
};


#endif //PATH_H
```

Class Connection:

```
#ifndef CONNECTION_H
#define CONNECTION_H
#include "path.h"
#include <iostream>
#include <vector>

class Connection
{
public:
	Connection(Path* the_path);
	/* This constructor creates a new Connection object. Best path is set to the_path and second_best_path
		is set to nullptr. Because there is no sense in keepeing in connections table a Connection object with
		both best_path and second_best_path being nullptr (connection with no path) this class does not allow existance of
		such a Connection object.
	*/
	Connection(const Connection& source); //copy constructor, creates a deep copy
	~Connection(); //Destructor frees memmory allocated for both paths and this object
	void update(Path* the_path);
	/* Update functions compares passed path (the_path) with the best_path and second_the best path
		of this object.
		In case thath passed path is better (shorter) than at least one of the object's path this functions
		removes one of object's paths and add the_path to the object.
		In case that path is worse (longer) than each of the object paths this frees memory allocated for passed path.
		Function throws an exception if first city on the passed path it's different that first city of current best path of this object.
	*/
	void print(std::ostream& stream, std::vector<std::string> & names);
	/*
	Print's paths to the destination.
	In case that there is only one known path to destination this information will be printed. (when second_best_path is nullptr)
	In case that there are two knows path they both will be printed.
	*/

private:
	Path* best_path; //this is the shortets known path to destination
	Path* second_best_path;
	/*In case of only one known path this is set to nullptr. Otherwise it is second best path. Second best path is
		always equal to or shorter than best path.
	*/
	unsigned int destination;
	/*The first city to be on best_path and on the second_best_path. They must be the same cities.
	*/
};

#endif // !CONNECTION_H
```

```
Class SLL (singly linked list)
#ifndef SLL_H
#define SLL_H
#include <stdexcept>

template <typename key_type, typename value_type>
class SLL{
public:
	class IteratorBase; class IteratorConst; class IteratorVar;

protected:
	class ListElement
	{
		friend class SLL;
		friend class IteratorBase;
		friend class IteratorConst;
		friend class IteratorVar;

	public:
		ListElement(key_type key, value_type value, ListElement* next =
nullptr);
		~ListElement();
		key_type key;
		value_type value;
		ListElement* next;
	};

public:
	class IteratorBase
	{
	public:
		IteratorBase(ListElement* current);

		IteratorBase& operator++();
		IteratorBase operator++(int);

		bool operator==(const IteratorBase& b); //{ return a.current ==
b.current; }
		bool operator!=(const IteratorBase& b); // { return a.current !=
b.current; }

	protected:
		ListElement* current;
	};

class IteratorVar : public IteratorBase
{
public:
	IteratorVar(ListElement* current);
	value_type& operator*();
	key_type& operator&();
};
class IteratorConst : public IteratorBase
{
public:
	IteratorConst(ListElement* current);
	value_type operator*() const;
	key_type operator&() const;
};

public:
	SLL();
	~SLL();
```

```
        int get_element_nr() const;
        void clear();
        bool remove(const key_type key);
        bool is_in(const key_type key) const;
        value_type operator[](const key_type key) const;
        value_type & operator[](const key_type key);
        value_type* insert_back(const key_type key, const value_type value);
        IteratorConst begin() const;
        IteratorVar begin();



protected:
        ListElement* first;
        ListElement* last;
        int elements_nr;
        ListElement* find(const key_type key) const;
};
#endif // SLL_H
```

```
#ifndef CITY_H
#define CITY_H
#include "path.h"
#include <iostream>
#include "sll.h"
#include "neighbour.h"
#include "connection.h"
#include <vector>


class City {
public:
        City(unsigned int id); //creates a new city with given id
        void update(Path* the_path);
        /*
                If the origin city (first city) of the_path is not yet in connection
table the new Connection object with
                best_path as the_path is created and added to connection table.
                If origin city of the path is in the connection table the update
method is called on Connection object
                represented the connection to the origin city. See void
Connection::update(Path* the_path) for more info.

                Then to each neighbour city which is not yet in on the path thisthe
copy of the path with addition of
                neighbour id and distance is send. Sending path means calling the
update function (recursively) on city object.
        */
        bool add_neigbour(unsigned int id, City* neighbour, unsigned int distance);
        /*      If city of given id is not yet in  neighbour cities list this city
is added to the list. Funciton
        * retunrns true in this case.
                If city of given id is already in given list distance between
neighbour and this city is set on
                argument distance value.
        */
        bool remove_me(unsigned int id);
        /*If city of given id is in neigbours list of this city, city of given id
is removed from neigbours list.
        * Function retunrs true in that case.
        *
        * If given city is not in the neighbour list function do nothing and return
false.
```

```
	*/
	void print_neghbours(std::ostream& stream, std::vector<std::string> &
names);
	/*This function print on passed stream all neighbours of  this city.
	Arguments:
		std::ostream& my_stream:
			The stream on which path will be printed (like std::cout).
		std::vector<std::string> & names:
			Because of the fact that Path knows only ids of the city, no
names while printing
			path object we need to some howe tell what is the name of each
city in the path.
			Names is vector which indices are cities ids and values are
cities names. For example
			name of city of id 7 is names[7].
	*/
	void print_all_connections(std::ostream& stream, std::vector<std::string>&
names);
	bool print_connection(std::ostream& stream, std::vector<std::string> &
names, unsigned int id);
	/*This function print on passed stream a best path (and if exists second
best path) to given city.
	If connection to given city is not on the connection table function prints
that information on screan and
	returns false.
	Otherwise function returns false.
	Arguments:
		std::ostream& my_stream:
			The stream on which path will be printed (like std::cout).
		std::vector<std::string> & names:
			Because of the fact that Path knows only ids of the city, no
names while printing
			path object we need to some howe tell what is the name of each
city in the path.
			Names is vector which indices are cities ids and values are
cities names. For example
			name of city of id 7 is names[7].
		unsigned int id:
			The distination, the id of the city to which we want to print
connection.
	Exemplatory otput on stream from this function:

	From city Gdansk :
	Best path to Wroclaw (total lengh is 560 ):
	--> Lodz (340) --> Wroclaw (220)
	Alternative path to Wroclaw (total lengh is 800 ):
	--> Szczecin (350) --> Wroclaw (450)
	*/
	void forget_all_connections(); //forces to forget all connections (to clear
connection table)
	unsigned int get_id(); //returns the id of this city
	void call_update();
	/*The path sending process needs to start somewhere. This function calls
update mehtod on all it's neighbours.
	* (Its sends the path conating id of this city and distance to given
neighbour to all it's neigbour)
	*/

private:
	const unsigned int id; //every city has an unique id number
	SLL<unsigned int, Neighbour>  neighbours; //the neigbours list
	SLL<unsigned int, Connection> connections; //the connection list
};
```

```
#endif
```

Class City:

```
#ifndef CITY_H
#define CITY_H
#include "path.h"
#include <iostream>
#include "sll.h"
#include "neighbour.h"
#include "connection.h"
#include <vector>


class City {
public:
	City(unsigned int id); //creates a new city with given id
	void update(Path* the_path);
	/*
		If the origin city (first city) of the_path is not yet in connection table the new Connection object with
		best_path as the_path is created and added to connection table.
		If origin city of the path is in the connection table the update method is called on Connection object
		represented the connection to the origin city. See void Connection::update(Path* the_path) for more info.

		Then to each neighbour city which is not yet in on the path thisthe copy of the path with addition of
		neighbour id and distance is send. Sending path means calling the update function (recursively) on city object.
	*/
	bool add_neigbour(unsigned int id, City* neighbour, unsigned int distance);
	/*	If city of given id is not yet in  neighbour cities list this city is added to the list. Funciton
	* retunrns true in this case.
		If city of given id is already in given list distance between neighbour and this city is set on
		argument distance value.
	*/
	bool remove_me(unsigned int id);
	/*If city of given id is in neigbours list of this city, city of given id is removed from neigbours list.
	* Function retunrs true in that case.
	*
	* If given city is not in the neighbour list function do nothing and return false.
	*/
	void print_neghbours(std::ostream& stream, std::vector<std::string> & names);
	/*This function print on passed stream all neighbours of  this city.
	Arguments:
		std::ostream& my_stream:
			The stream on which path will be printed (like std::cout).
		std::vector<std::string> & names:
			Because of the fact that Path knows only ids of the city, no names while printing
			path object we need to some howe tell what is the name of each city in the path.
			Names is vector which indices are cities ids and values are cities names. For example
			name of city of id 7 is names[7].
	*/
	void print_all_connections(std::ostream& stream, std::vector<std::string>& names);
```

```
	bool print_connection(std::ostream& stream, std::vector<std::string> &
names, unsigned int id);
	/*This function print on passed stream a best path (and if exists second
best path) to given city.
	If connection to given city is not on the connection table function prints
that information on screan and
	returns false.
	Otherwise function returns false.
	Arguments:
		std::ostream& my_stream:
			The stream on which path will be printed (like std::cout).
		std::vector<std::string> & names:
			Because of the fact that Path knows only ids of the city, no
names while printing
			path object we need to some howe tell what is the name of each
city in the path.
			Names is vector which indices are cities ids and values are
cities names. For example
			name of city of id 7 is names[7].
		unsigned int id:
			The distination, the id of the city to which we want to print
connection.
	Exemplatory otput on stream from this function:

	From city Gdansk :
	Best path to Wroclaw (total lengh is 560 ):
	--> Lodz (340) --> Wroclaw (220)
	Alternative path to Wroclaw (total lengh is 800 ):
	--> Szczecin (350) --> Wroclaw (450)
	*/
	void forget_all_connections(); //forces to forget all connections (to clear
connection table)
	unsigned int get_id(); //returns the id of this city
	void call_update();
	/*The path sending process needs to start somewhere. This function calls
update mehtod on all it's neighbours.
	* (Its sends the path conating id of this city and distance to given
neighbour to all it's neigbour)
	*/

private:
	const unsigned int id; //every city has an unique id number
	SLL<unsigned int, Neighbour>  neighbours; //the neigbours list
	SLL<unsigned int, Connection> connections; //the connection list
};

#endif
```

Class Map:

```
#ifndef MAP_H
#define MAP_H
#include <vector>
#include <iostream>
#include "sll.h"
#include "city.h"


class Map{
public:
	Map(); //creates new Map with no cities
	bool link(std::string city1, std::string city2, unsigned int distance,
std::ostream& stream);
	/*If any of passed cites is not yet on the map, function first creates new
city on new map with given name.
	If cities were unlink before (one wasn't a neighbour of other) function
links them (adds second city to list of neighbours of city1 and vice versa).
	If cities were linked before function reset distance between them, so user
can use this function to change distance between cities.
	Function returns true if new link was created, false otherwise.
	Functon prints on stream relevent information (about city creation or link
creation)
	*/
	bool unlink(std::string city1, std::string city2, std::ostream& stream);
	/*Function returns false and do nothing if at least one city wasn't on the
map.
	Function returns false and do nothing if cities were unlinked. (but both
were on the map).
	If given cities were linked, function unlinks them and returns true.
Unlinking means removing first city of
	neighbours list of second city and removing second city from neighbours
list of first city.
	Function prints relevant information on stream (wheather cities were on
map, wheather they were linked before...s
	*/
	bool how_to_get(std::string from, std::string destination, std::ostream&
stream);
	/*If any of two passed cities is not on the map fuction "how_to_get" will
print this information.
	If there is no way to get from one city to another fuction "how_to_get"
will print this information.
	If there is only one convenient path from one city to another, fuction
"how_to_get" will print this path and print that this is the only possible path.
	If there are at least two possible paths connecting those two cities
fuction "how_to_get" will print those path as best_path and second_best_path.
	*/
	void recalculate_map();
	//calculating the best paths from each city to each other city
	//see do main doc "story" for more info

	void print_all_cities(std::ostream& stream);
	/*prints names of all cities that are actually on the map
	Exemplatory otput on stream from this function:
	1) Krakow  2) Warszawa  3) Rzeszow  4) Gdansk 5) Szczecin
	6) Gdynia 7) Czestochowa
	*/

	void print_city_neighbours(std::string city_name, std::ostream& stream);
	//prints all neighbours of given city. If city is not on the map this
information will be printed.
	void print_city_connections(std::string city_name, std::ostream& stream);
	//prints all connections (where you can get from this city) of given city.
```

```
	//If city is not on the map this information will be printed.

private:
	SLL<std::string, City> cities;
	/*the singles linkded list representing all cities which are on the map.
	the keys are cities names (like Warszawa) and values are cities objects.
	Each time new city is added on the map the new city object is pushed at the
end of this list.*/
	unsigned int next_id;
	//this id will be given to each new added on the map. Id of first city will
be 0, id of second city will be 1 and so on
	//this value is incremented after each new city additions
	std::vector<std::string> names;
	/*Because of the fact that each city, connetcion or path object knows only
ids of the cities, no names, while printing
	any connecton or path  we need to some howe tell what is the name of each
city.
	Names is vector which indices are cities idsand values are cities names.
	For example name of city of id 7 is names[7].
	This vector is passed as reference to each path, connecton or city object
method which prints sth on some strem.
	Each time new city is added on the map the new name is pushed at the end of
this vector.
	*/
};

#endif // !MAP_H
```