Middle East Technical University - Department of Computer Engineering

CENG 213

Data Structures

Fall '2021-2022

Programming Assignment 3

Due Date: 17 January 2021, Monday, 00:05
Late Submission Policy will be explained below

Objectives

In this programming assignment, you are expected to implement functional equivalent of a maps application (i.e. Google Maps, Yandex Maps etc.). In order to implement such application, you will additionally implement *Graph* and *Hash Table* data structures. Details of these data structures will be explained in the following sections.
Keywords: C++, Data Structures, Graphs, Dijkstra Shortest Path Algorithm, Hash Table, Quadratic Probing

1 Graph Implementation (40 pts)

The Graph data data structure that will be used in this assignment, follows the adjacency list style of implementation with slight differences. Graph Data Structure has non-negative weights for its edges. Additionally, edges on this data structure is **bi-directional** meaning that graph is **not** a directional graph. Additional functionality consists of masking an edge in the graph and finding multiple "shortest" paths using masking functions. Data layout of Graph class and its helper structures can bee seen on Listing 1.
Instead of linked list, graph holds its data (in this case edges and vertices) in dynamic arrays. For dynamic array implementation, class utilizes `std::vector<T>` class from the *Standard Template Library (STL)*. Graph class and its helper structures are declared in *Graph.h* header file and their implementations (although most of it is empty) are defined in *Graph.cpp* file.

1.1 GraphVertex Struct

GraphVertex structure holds the information about a vertex. A GraphVertex instance holds *edgeIds* for which edges are originated from this vertex.
At most a vertex can have **MAX_EDGE_PER_VERTEX** (currently this is set to 8) amount of edges. Since maximum edge count is defined as a compile time constant, *edgeIds* variable is declared as a static array. Thus, *edgeCount* variable holds the actual edge count in *edgeIds* array. Please note that an edgeId in *edgeIds* array is the index of the *Graph::edgeList* dynamic array. Finally *name* variable holds the unique name of that vertex.

1.2 GraphEdge Struct

GraphEdge structure holds the information about an edge. *weight* variable holds the weight of the vertex, *masked* flag holds whether this edge is masked or not. Finally *vertexId0* and *vertexId1* hold the vertex indices that this edge connects.
Just like edgeId, *vertexId0* and *vertexId1* are indices of the *Graph::vertexList* dynamic array.

```cpp
// Compile time constants
#define MAX_EDGE_PER_VERTEX 8
#define INVALID_INDEX -1
// Large number can be use to set as initial weights.
// Weights of the edges are guaranteed to not exceed this value
#define LARGE_NUMBER 0x7FFFFFFF

struct GraphVertex
{
    int             edgeIds[MAX_EDGE_PER_VERTEX];   // Adjacency List
    int             edgeCount;                      // Current edge count
    std::string     name;                           // Name of the vertex
};

struct GraphEdge
{
    int     weight;     // Weight of the edge (used on shortest path)
    bool    masked;     // Whether this edge is masked or not
    int     vertexId0;  // Information about which two vertices this edge
    int     vertexId1;  // connects.
};

class Graph
{
    private:
    std::vector<GraphVertex>    vertexList;
    std::vector<GraphEdge>      edgeList;

    ...
};
```

Listing 1: Graph and Helper Structure data layout

1.3 Graph Class

Graph class implements non-directional, non-negative weighted graph. It only allows single edge between two vertices. It utilizes two dynamic arrays for its data, and it is capable of some functionalities that will be explained shortly.

1.3.1 `Graph();`

Default constructor constructs an empty graph. It relies on the default constructors of the std::vector. **This function is implemented for you**.

1.3.2 `Graph(const std::string& filePath);`

This constructor reads a ".map" file and constructs the Graph data structure using the information in the file. **This function is implemented for you however; it relies on** *`InsertVertex(...)`* **and** *`ConnectVertices(...)`* **functions.**

1.3.3 `void InsertVertex(const std::string& vertexName);`

This function tries to insert a vertex named "vertexName" without any edge. However, it should throw **"DuplicateVertexNameException"** if a same named vertex already exists on the graph.

1.3.4 `bool ConnectVertices(const std::string& fromVertexName, const std::string& toVertexName, int weight);`

This function tries to set an edge between two vertices. If such named vertex is not found, it should throw **"VertexNotFoundException"**.
If edge count will exceed the **MAX_EDGE_PER_VERTEX**, it should throw **"TooManyEdgeOnVertexExecption"**. Additionally, this function returns true if the insertion is successful. Insertion may fail when these two vertices already have an edge. Then this function will return false.

1.3.5 Mask Functions

Unlike traditional graphs, this graph implementation will consist masking functions. Mask functions mask the particular edge(s) in the graph. Masked edges are not used on the calculations of "`PrintAll`" and "`ShortestPath`" algorithms. Masked edges does **not** alter the behavior of any other function (i.e "`ConnectVertices`" function still returns false even if the edge between those two vertices are masked)
There is no error when user tries to mask/unmask an already masked/unmasked edge or tries to mask/unmask an unavailable edge.

1.3.6 `void MaskEdges(const std::vector<StringPair>& vertexNames);`

This function masks edges between vertex name pairs on the array *vertexNames*. Definition of *StringPair* can be seen on Listing 2.

```
struct StringPair
{
    std::string s0;
    std::string s1;

    ...
};
```

Listing 2: StringPair struct

If any of the names does not correspond to a vertex in the graph, this function should throw **"VertexNotFoundException"**.

1.3.7 `void UnMaskEdges(const std::vector<StringPair>& vertexNames);`

This function does the complete opposite of the function explained in Section 1.3.6.

1.3.8 `void UnMaskAllEdges();`

This function unmasks all the edges in the graph.

1.3.9 `MaskVertexEdges(const std::string& name);`

This function masks all the edges coming out of the vertex with the name *name*. If name does not correspond to a vertex in the graph, this function should throw **"VertexNotFoundException"**.

1.3.10 `UnMaskVertexEdges(const std::string& name);`

This function does the complete opposite of the function explained in Section 1.3.9.

1.3.11 `bool ShortestPath(std::vector<int>& orderedVertexIdList, const std::string& from, const std::string& to) const`

This function is used to find shortest path between two vertices. It returns the shortest path as an ordered vertex id list in the *orderedVertexIdList* variable. If a shortest path exists, this variable's first element should be the vertex id of the vertex named "from" and the last element should be the vertex id of the vertex named "to". If a path cannot be found, then *orderedVertexIdList* should remain empty. Return value of the function is true when a shortest path is found and false when there is no path exists between those two vertices.
If either "to" or "from" is not available in the graph, function should throw **"VertexNotFoundException"**.
Remarks

- You may want to use *`std::priority_queue`* STL library class. A helper class **DistanceVertexIdPair** is provided for you to be used in a *`std::priority_queue`*.
- In order remain consistent between implementations, do **not** update the path weight if it is **equal** to previously found path weight. This means that the "first" shortest path will be chosen as the shortest path if any other equally weighted shortest path exists.
- Still some inconsistencies may occur depending on the priority queue (heap) implementation. Because of that, we would strongly suggest you to use `std::priority_queue` and **DistanceVertexIdPair** instead of implementing your own heap data structure.
- While running, this function must **not** consider masked edges as a valid edge.

1.3.12 `int MultipleShortPaths(std::vector<std::vector<int>>& orderedVertexIdList, const std::string& from, const std::string& to, int numberOfShortestPaths);`

This function returns *numberOfShortestPaths* amount of short paths according to the algorithm which will be defined below.
The algorithm can be expressed as follows:

- Unmask all the edges
- Find the shortest path normally
- Add the found path to the *orderedVertexIdList*
- Refer to it as the last found path
- Until *numberOfShortestPaths* is reached
  - Mask the highest weighted edge on the last found path
  - Run shortest path on the graph
  - If any other shortest path could **not** be found; terminate.
  - Add the found path to the *orderedVertexIdList*
  - Refer the found path as last found path.
- Unmask all the edges

For curious students, this algorithm is **not** an exact solution for the multiple shortest path algorithm (formally known as k-shortest path) can be implemented using Yen's Algorithm. You can find the paper of the algorithm here.

Algorithm is quite similar to the algorithm defined above but masking and unmasking operations are different and complex. Do **not** implement this algorithm. Implement the algorithm described above.

Argument *orderedVertexIdList* is dynamic array of dynamic array of integers. Each dynamic array of integer should hold a valid path as defined the the Section 1.3.11. You may assume that *orderedVertexIdList* is empty initially. This function should return the number of shortest paths available in the *orderedVertexIdList*.
If either "to" or "from" is not available in the graph, function should throw **"VertexNotFoundException"**.

1.3.13 `void ModifyEdge(const std::string& vName0, const std::string& vName1, float newWeight);`

Modifies the edge weight between two vertices. If either "to" or "from" is not available in the graph, function should throw **"VertexNotFoundException"**. If no edge is present between these vertices, function should fail silently.

1.3.14 `void ModifyEdge(int vId0, int vId1, float newWeight);`

Modifies the edge weight between two vertices. If no edge is present between these vertices and/or any of the ids are not valid, function should fail silently.

1.3.15 `void PrintAll() const;`

Prints the graph on to the standard output. If an edge is masked, print function does not print that edge. **This function is implemented for you**.

1.3.16 `void PrintPath(const std::vector<int>& orderedVertexIdList, bool sameLine = false) const;`

Prints the orderedVertexIdList (which probably is the output of "ShortestPath" or "MultipleShortPaths") to the standard output. Please notice that this function does not check if an edge exists between adjacent vertices on the list. If no edge exists between these vertices, it prints "`-##->`". If *sameLine* is true, this function prints on a single line. **This function is implemented for you**.

1.3.17 `int TotalVertexCount() const;`

This function should return the total number of vertices in the graph.

1.3.18 `int TotalEdgeCount() const;`

This function should return the total number of edges in the graph.

1.3.19 `std::string VertexName(int vertexId) const;`

This function should return name of the vertex with the given id. If no such vertex is available, this function should return an empty string.

1.3.20 `int TotalWeightInBetween(std::vector<int>& orderedVertexIdList, int start, int end);`

Accumulates the edge weights between vertices in the *orderedVertexIdList*. If a vertex with the given id is not available in the graph, this function should throw **"VertexNotFoundException"**. If no edge exists between any of the vertices function should return `-1`.

2 Hash Table (30 pts)

In addition to the Graph data structure maps application requires a hash table for storing found paths. You are going to implement *KeyedHashTable* class which stores string and integer vector pairs. The string will be used to determine the location of the data using hash function. Data layout of the *KeyedHashTable* can be seen on Listing 3. Hash table will resolve its collisions using **quadratic probing**.
KeyedHashTable class and its helper structures are declared in *HashTable.h* header file and their implementations (although most of it is empty) are defined in *HashTable.cpp* file.

```
// Expand threshold is the multiplicative inverse of the 1/3 (%33)
// in order to prevent floating point math
#define EXPAND_THRESHOLD 3
// Only first 100 primes are defined, test cases would not exceed the maximum
// prime in this table
#define PRIME_TABLE_COUNT 100

struct HashData
{
    std::vector<int>    intArray;
    std::string         key;

    ...
};

class KeyedHashTable
{
    private:
    // Stores the first 100 primes
    static const int            PRIME_LIST[PRIME_TABLE_COUNT];
    ...

    // Hash Table Storage
    HashData*                   table;
    int                         tableSize;
    int                         occupiedElementCount;

    ...
};
```

Listing 3: Hash Table Structures

2.1 KeyedHashTable Class

KeyedHashTable class has the main implementation. This class holds its data as an array on a simple allocation which is pointed by the *table* variable. Entire capacity of the hash table is on *tableSize* variable. Variable *occupiedElementCount* holds how many entries are on the hash table. Empty entries should be initialized with an empty string and an empty vector.

2.1.1 `static int FindNearestLargerPrime(int requestedCapacity);`

This function should return the nearest but larger prime from the *PRIME_LIST* static variable. You can assume *requestedCapacity* is never greater or equal to 100^{th} prime which is 541. Even if

requestedCapacity is a prime, this function should still return the **next prime**.

2.1.2 `int Hash(const std::string& key) const;`

This function returns the hashed value of a string. Equation 1 shows the mathematical formula of the hash function

$$\begin{aligned} hash &= \sum_{k=1}^{n} string[k] * PRIME\_LIST[k] \\ hash &= (hash \mod tableSize) \end{aligned} \tag{1}$$

n is the string length. You can assume that the length of the string never exceeds 100.

2.1.3 `void ReHash();`

Rehash function is called when `(occupiedElementCount * EXPAND_THRESHOLD >= tableSize)`. When called; this function allocates a new table with a new prime. This prime is found by using `int FindNearestLargerPrime(...)`. Argument for `int FindNearestLargerPrime(...)` will be the **doubled** table size. Then it rehashes all of the entries from the old table to the new table. Finally it deletes the old table.

2.1.4 `KeyedHashTable();`

Default constructor will create the hash table with the smallest prime number which is 2.

2.1.5 `KeyedHashTable(int requestedCapacity);`

This constructor will create the hash table with the nearest but larger prime of *requestedCapacity*.

2.1.6 `KeyedHashTable(const KeyedHashTable&);`

Copy constructor should do **deep copy.**

2.1.7 `KeyedHashTable& operator=(const KeyedHashTable&);`

Copy assignment operator should do a **deep copy.**

2.1.8 `~KeyedHashTable();`

Destructor should delete all the allocated data.

2.1.9 `bool Insert(const std::string& key, const std::vector<int>& intArray);`

This function inserts a value with key *key* and an integer array *intArray*. Collisions should be resolved using **quadratic probing**. After insertion, this function should call `void ReHash()` when `(occupiedElementCount * EXPAND_THRESHOLD >= tableSize)`.
If insert operation is successful, function should return true. Insert operation may fail when there is an entry with a same key. Then this function should return false.

2.1.10 `bool Remove(const std::string& key);`

This functions tries to remove an entry with a key named *key*. If there is no such keyed entry present on the table it returns false. If the entry is found and deleted, this function returns true. In order to "delete" an entry, its key should be set to empty string and its array should be set to an empty array.

2.1.11 `void ClearTable();`

This function clears all of the entries in the table. This function does **not** deallocate the table. It sets all of occupied entries to empty string and empty array.

2.1.12 `bool Find(std::vector<int>& valueOut, const std::string& key) const;`

This function tries to find an entry with a key named *key*. If such entry is present, function copies that entry's integer array to the *valueOut* variable and returns true. If such entry is not available, function returns false.

2.1.13 `void Print() const;`

Prints the hash table. This function only prints the occupied elements. **This function is implemented for you**.

3 METU Maps (30 pts)

"METUMaps" class combines the Graph data structure and KeyedHashTable data structure. Data layout of the class can be seen on Listing 4. Continuing with the same pattern; METUMaps class is declared on *METUMaps.h* header file and its definitions are in *METUMaps.cpp* file.

```
class METUMaps
{
    private:
    Graph           map;
    KeyedHashTable  cachedPaths;
    // How many paths should be provided
    // by the maps array
    int             potentialPathCount;

    // States
    bool                inJourney;
    std::string         startingLoc;
    std::string         currentLoc;
    std::string         destination;
    std::vector<int>    currentRoute;

    ...
}
```

Listing 4: METUMaps Class

3.1 METU Maps Class

METUMaps class is basically a state machine. This means that call order of the functions is important. In order to properly use the class, user first set its destination and starting location. Then it should start the journey. During journey, user can update its current location and call display function to see its current route and timing values. Finally user should end the journey. MetuMaps is responsible to warn the user when such order is not satisfied using the print functions (Section 3.1.1).

3.1.1 Print Functions

There are many print operations in this class. In order to provide consistency, all print operations are provided as functions. These functions are not explained here but all of which are self-explanatory. **These functions are:**

- `void PrintNotInJourney() const;`
- `void PrintUnableToChangeDestination() const;`
- `void PrintUnableToChangeStartingLoc() const;`
- `void PrintAlreadyInJourney() const;`
- `void PrintJourneyIsAlreadFinished() const;`
- `void PrintLocationNotFound() const;`
- `void PrintJourneyCompleted() const;`
- `void PrintCachedLocationFound(const std::string& location0, const std::string& location1)`
- `void PrintCalculatingRoutes(const std::string& location0, const std::string& location1) const;`

3.1.2 `static std::string GenerateKey(const std::string& location0, const std::string& location1);`

Generates a key that will be used in the hash table. **This function is implemented for you**.

3.1.3 `METUMaps(int potentialPathCount, const std::string& mapFilePath);`

Constructor will generate a graph using the string *mapFilePath* and sets the *potentialPathCount* member variable to the same named argument. Initially class is not in a journey. Initially, hash table should be constructed with the multiplication of total location count in the map and *potentialPathCount*.

3.1.4 `void SetDestination(const std::string& name);`

This function tries to set *destination*. This function succeeds only if the user is not in a journey. If this function is called during journey it should print `PrintUnableToChangeDestination()`.

3.1.5 `void SetStartingLocation(const std::string& name);`

This function tries to set *startingLoc*. This function succeeds only if the user is not in a journey. If this function is called during journey it should print `PrintUnableToChangeStartingLoc()`.

3.1.6 `void StartJourney();`

This function starts a journey. When a journey starts, this function finds the shortest paths between *startingLoc* and *destination*. Before calculating the shortest paths, it prints `PrintCalculatingRoutes(...)`. How many paths that are going to be calculated is determined by the *potentialPathCount* variable. After paths are found, this function caches these paths on to the hash table. Caching operation will be explained in Section 3.1.10. Additionally, it sets *currentLoc* to *startingLoc*. Finally, it sets the *currentRoute* to the first route that is found.
If this function is called during journey, it should print `PrintAlreadyInJourney()`. If either *startingLoc* and *destination* not the map, it should print `PrintLocationNotFound()`.

3.1.7 `void EndJourney();`

This function ends the journey. If this function is **not** called during a journey, it should print `PrintJourneyIsAlreadFinished()`. Otherwise; this function should clear the cache table, *destination*, *startingLoc* and *currentLoc* variables.

3.1.8 `void UpdateLocation(const std::string& name);`

This function will be called when user reached on a certain location on the map. User may or may not call this from a location that is already on the current path. Algorithm-wise, this function will check the destination *name* between *destination* is already in the cache. If such path is in hash table it should print `PrintCachedLocationFound()` and set *currentRoute* to the cached route. If location can not be found in the hash table it should print `PrintCalculatingRoutes(...)` then it should find shortest path between *name* and *destination* just like in `StartJourney()` function. Finally, this function should cache found locations to the hash table which will be explained in Section 3.1.10.
If this function is **not** called during journey, it should print `PrintNotInJourney()`. If *name* is not on the map it should print `PrintLocationNotFound()`. If user reached to his/her destination, function should print `PrintJourneyCompleted()`.

3.1.9 `void Display();`

This function prints the current journey parameters. This function should be called during journey otherwise this function will print `PrintNotInJourney()`. **This function is implemented for you.**

3.1.10 Caching

When multiple short paths are found, METUMaps class should cache these on to the hash table. **All** sub paths towards the destination should be cached. For example, assuming two paths `(A->B->C->D)` and `(A->E->C->D)` are found, `(A->B->C->D)`, `(B->C->D)`, `(C->D)`, `(A->E->C->D)`, `(E->C->D)`, `(C->D)` should be cached on to the hash table. Key value of such paths are found by using `void GenerateKey()` function. First argument on this function should be the starting location on the sub path and second argument should be the *destination* (Key value of `B->C->D` will be the output of `GenerateKey(B, D)`. You should ignore the duplicates.

Regulations

1. **Programming Language:** You will use C++.
2. Standard Template Library is not allowed except "`std::priority_queue`" "`std::string`" and "`std::vector`". "`std::priority_queue`" should only be used on the appropriate functions as a temporary data structure.
3. Using external libraries other than those already included are not allowed.
4. Changing or modifying already implemented functions are not allowed
5. You can add any private member functions unless it is explicitly stated that you should not.
6. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.
7. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the university regulations. Remember that students of this course are bounded to code of honor and its violation is subject to severe punishment.

8. **Newsgroup:** You must follow the Forum (odtuclass.metu.edu.tr) for discussions and possible updates on a daily basis.

Submission

- Submission will be done via CengClass, (cengclass.ceng.metu.edu.tr).
- Don't write a "main" function in any of your source files. It will clash with the test case(s) main function and your code will not compile.
- A test environment will be ready in CengClass :
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.