Middle East Technical University Department of Computer Engineering

CENG 213

Data Structures

Fall 2020-2021

Programming Assignment 3

Due date: 14.01.2021, Thursday, 23:59

1 Introduction

In this assignment, you are going to practice on hash table and graph implementations in two stages. First, you are asked to implement a generic hash table along with its accompanying hash function using C++. Notice that the runtime of insertion, deletion, and retrieval operations are expected to be constant time for hash tables. Afterwards, you need to develop a program that allows users to do some operations on the world metal trade dataset in 1994 which is provided on Cengclass. For this purpose, you will first construct a weighted, directed graph from the dataset and then implement the given methods for users to be able to analyze the dataset.

Keywords: *Hash table, Quadratic probing, Directed graphs, C++*

2 World Metal Trade Dataset

The dataset contains information on trade miscellaneous manufactures of metal among 80 countries in 1994. All countries with entries in the paper version of the Commodity Trade Statistics published by the United Nations were included, but for some countries, the 1993 data (Austria, Seychelles, Bangladesh, Croatia, and Barbados) or 1995 data (South Africa and Ecuador) were used because they were not available for 1994. You can read more about the dataset at:

`http://vlado.fmf.uni-lj.si/pub/networks/data/esna/metalWT.htm`

The nodes represent countries with its additional information such as the country id, country name, continent, and gross domestic product (gdp). The edges represent imports by one country from another for the class of commodities designated as 'miscellaneous manufactures of metal', which represents high technology products or heavy manufacture. The absolute value of imports (in 1000 US$) is used but imports with values less than 1% of the country's total imports were omitted. There is no self loop within the dataset. Each field is separated by tab character in the dataset. The file structure is given below:

```
*Vertices 80  // header for the vertices, 80 indicates the number of vertices in the dataset
1 Algeria Africa  1209 // 1 is the country id, "Algeria" is the country name, "Africa" is the ↩
    continent of Algeria, "1209" is the gdp per capita for Algeria
.
.
.
*Edges // header for the edges
78  25  556 // there is a directed edge originated from 78 towards 25 where both 78 and 25 are ↩
    the country ids. 556 is the edge weight that represents the metal import value
...
```

In this assignment, you are supposed to use the following class definitions to create nodes and edges. Note that, node and edge objects are created in the main function while parsing the dataset, so you are just expected to use the generated objects in your hash table and graph implementations. You are not allowed to modify these classes.

```
class Node {

public:
    // Constructors
    Node() : country(""), continent("") {}
    Node(int vid_, const std::string &country_, const std::string &continent_, long gdp_) :
        vid(vid_), country(country_), continent(continent_), gdp(gdp_) {}

    // Getters & setters
    int getVid() const { return vid;}

    void setVid(int vid) { Node::vid = vid; }

    const std::string &getCountry() const { return country; }

    void setCountry(const std::string &country_) { country = country_; }

    const std::string &getContinent() const { return continent; }

    void setContinent(const std::string &continent_) { continent = continent_; }

    long getGdp() const { return gdp; }

    void setGdp(long gdp_) { gdp = gdp_; }

private:
    int vid; // country id
    std::string country; // country name
    std::string continent; // continent of the country
    long gdp; // gross domestic product per capita
};

class Edge {
public:

    // Constructors
    Edge() {}

    Edge(const Node &tailNode, long import_) : tailNode(tailNode), import(import_) {}


    // Getters & setters
    const Node &getTailNode() const { return tailNode; }

    void setTailNode(const Node &tailNode_) { tailNode = tailNode_; }

    long getImport() const { return import; }

    void setImport(long import_) { import = import_; }

private:
    Node tailNode; // the country that imports the metal
    long import; // import value (in 1000 US$)

};
```

3 Part 1: Hash Table Implementation (30 POINTS)

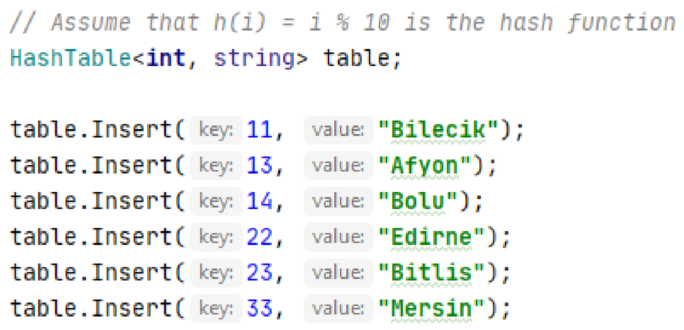
In the first step, you will design a generic hash table that should be able to perform the following operations:

- Inserting an item with a key into the hash table
- Deleting an item with a key from the hash table
- Getting an item with a key from the hash table

3.1 Specifications

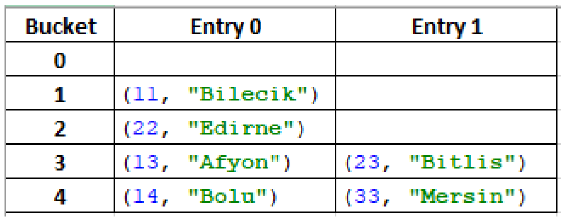
- In this hash table, for every possible place in the table, you are supposed to store at most 2 values, called a bucket. If the bucket for a key is used, then your hash table should use the standard open addressing (quadratic probing) to find the next bucket for that key. Note that you should perform quadratic probing only for inter-bucket traversal. Since a bucket consists of only two entries, you do not need such operation within the bucket.

Example



```cpp
// Assume that h(i) = i % 10 is the hash function
HashTable<int, string> table;

table.Insert( key: 11,  value: "Bilecik");
table.Insert( key: 13,  value: "Afyon");
table.Insert( key: 14,  value: "Bolu");
table.Insert( key: 22,  value: "Edirne");
table.Insert( key: 23,  value: "Bitlis");
table.Insert( key: 33,  value: "Mersin");
```



Bucket	Entry 0	Entry 1
0		
1	(11, "Bilecik")	
2	(22, "Edirne")	
3	(13, "Afyon")	(23, "Bitlis")
4	(14, "Bolu")	(33, "Mersin")

There is a sample hash table implementation given above. For simplicity, we assume that the hash function is h(key) = key % 10. Therefore, (13, "Afyon"), (23, "Bitlis") and (33, "Mersin") should be located in the same bucket. However, since the bucket 3 is full, (33, "Mersin") tuple is located in Bucket 4 using quadratic probing.

- You are given a utility class, "HashUtils" which contains two hash functions for the keys of type string and int respectively, and a function for finding the next appropriate size for the hash table. It should not be modified in any case.

```cpp
// Returns the hash value of the given key
int Hash(const std::string& key);

// Returns the hash value of the given key
int Hash(int key);

// Finds the next appropriate hash table size from a given number
int NextCapacity(int prev);
```

- You will implement the functions (marked with "TODO") in the "HashTable.h", and must use open addressing with **quadratic probing** as a collision resolution strategy. The bare header file, "HashTable.h", is given below. You are free to add any private helper variables/functions to it. However, your hash table should support at least the given functions.

```cpp
class ItemNotFoundException : public std::exception{
public:
    const char * what() const throw() { return "Item Not Found in the Hash Table!"; }
};

template <class K, class T>
class HashTable {
    struct Entry {
        K Key;              // the key of the entry
        T Value;    // the value of the entry
        bool Deleted;          // flag indicating whether this entry is deleted
        bool Active;           // flag indicating whether this item is currently used

        Entry() : Key(), Value(), Deleted(false), Active(false) {}
    };

    struct Bucket {
        Entry entries[2];
    };

    int _capacity; // INDICATES THE TOTAL CAPACITY OF THE TABLE
    int _size; // INDICATES THE NUMBER OF ITEMS IN THE TABLE

    Bucket* _table; // THE HASH TABLE
    // == DEFINE HELPER METHODS & VARIABLES BELOW ==


public:
    // TODO: IMPLEMENT THESE FUNCTIONS.
    // CONSTRUCTORS, ASSIGNMENT OPERATOR, AND THE DESTRUCTOR
    HashTable();
    // COPY THE WHOLE CONTENT OF RHS INCLUDING THE KEYS THAT WERE SET AS DELETED
    HashTable(const HashTable<K, T>& rhs);
    HashTable<K, T>& operator=(const HashTable<K, T>& rhs);
    ~HashTable();

    // TODO: IMPLEMENT THIS FUNCTION.
    // INSERT THE ENTRY IN THE HASH TABLE WITH THE GIVEN KEY & VALUE
    // IF THE GIVEN KEY ALREADY EXISTS, THE NEW VALUE OVERWRITES
    // THE ALREADY EXISTING ONE. IF THE LOAD FACTOR OF THE TABLE IS GREATER THAN 0.6,
    // RESIZE THE TABLE WITH THE NEXT PRIME NUMBER.
    void Insert(const K& key, const T& value);

    // TODO: IMPLEMENT THIS FUNCTION.
    // DELETE THE ENTRY WITH THE GIVEN KEY FROM THE TABLE
    // IF THE GIVEN KEY DOES NOT EXIST IN THE TABLE, THROW ItemNotFoundException()
    void Delete(const K& key);

    // TODO: IMPLEMENT THIS FUNCTION.
    // IT SHOULD RETURN THE VALUE THAT CORRESPONDS TO THE GIVEN KEY.
    // IF THE KEY DOES NOT EXIST, THROW ItemNotFoundException()
    T& Get(const K& key) const;

    // TODO: IMPLEMENT THIS FUNCTION.
    // AFTER THIS FUNCTION IS EXECUTED THE TABLE CAPACITY MUST BE
    // EQUAL TO newCapacity AND ALL THE EXISTING ITEMS MUST BE REHASHED
    // ACCORDING TO THIS NEW CAPACITY.
    // WHEN CHANGING THE SIZE, YOU MUST REHASH ALL OF THE ENTRIES
    void Resize(int newCapacity);

    // THE IMPLEMENTATION OF THESE FUNCTIONS ARE GIVEN TO YOU
    // DO NOT MODIFY!
    int Capacity() const; // RETURN THE TOTAL CAPACITY OF THE TABLE
    int Size() const; // // RETURN THE NUMBER OF ACTIVE ITEMS
    void getKeys(K* keys); // PUTS THE ACTIVE KEYS TO THE GIVEN INPUT PARAMETER
};
```

- You can assume that country names and country ids for each record will be unique.
- **You have to call hash method (with either string or integer keys) in the "HashUtils.h" to calculate the hash value of an item.**
- **"NextCapacity" function takes an unsigned integer and returns a bigger prime number. You have to call this method while resizing the table.**
- While calculating the load factor, you should use bucket_ size (2) *_capacity as the total number of slots in the hash table.
- In this part, you are not allowed to use vector or any other standard libraries except the ones that are included.

3.2 Entry Struct

- Each "Entry" object stores information regarding a single key-value pair in the hash table.
- "Key" variable should store the original key variable given in the "Insert" method of the entry
- "Value" variables stores the value given in the "Insert" method.
- "Active" variable denotes that the key-value pair stored in this entry is valid. For example, 0^{th} entry of the 0^{th} bucket in the example above is not valid, however 0^{th} entries of the 1^{st}, 2^{nd}, 3^{rd} and 4^{th} bucket are valid and their "Active" fields are set as true. Initially, "Active" variable of each "Entry" object is set to false.
- "Deleted" variable stores whether this entry has been deleted before. Initially, every "Entry" object has this variable as false.

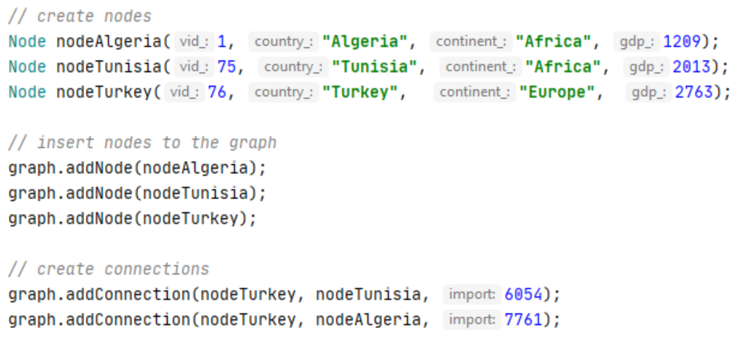
4 Part 2: Graph Implementation (70 POINTS)

In the second phase, you will implement the "TODO" functions in "Graph.cpp". In the graph class, each vertex represents a country and each directed edge represents imports by one country from another. The edge should be directed towards the country that imports the commodity. Each node has a country id, country name, continent, and gdp. Each edge has an import value in the US dollar which you should consider as the edge weight. The adjacency list of the graph should actually be stored in a hash map as follows:

$$HashTable < string, list < Edge >> adjList$$

You can assume that country id and country name fields are unique. Therefore you can also select country id as the key. In other words, you can change the types of key-value pairs of "adjList", it depends on your design. You can declare more than one hash tables, to store different key-value pairs which may facilitate your work. In graph class, you are expected to implement constructor, copy constructor, assignment operator, destructor and 8 more functions whose specifications are given below:

- **void addNode(const Node& node);**
  addNode function gets a node as a parameter and then adds this node to the hash table, where country name (or country id) is the key and an empty Edge list (list<Edge>) is the value.
- **void addConnection(const Node& headNode, const Node& tailNode, int import);**
  addConnection function adds a new Edge to the end of list of headNode using tailNode and import data. You must add the new edge to the back of headNode's list<Edge>. For example, lets create 3 nodes for Algeria, Tunisia and Turkey respectively. Then, each node is added to the graph using addNode function. At this stage, their adjacency lists are empty. Finally, we execute the last two statements and insert new edges from Turkey to Tunisia and to Algeria.

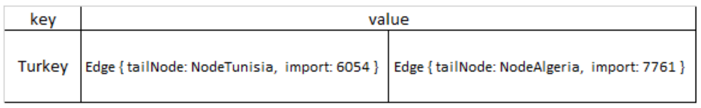


```
// create nodes
Node nodeAlgeria( vid_: 1,  country_: "Algeria",  continent_: "Africa",  gdp_: 1209);
Node nodeTunisia( vid_: 75,  country_: "Tunisia",  continent_: "Africa",  gdp_: 2013);
Node nodeTurkey( vid_: 76,  country_: "Turkey",   continent_: "Europe",   gdp_: 2763);

// insert nodes to the graph
graph.addNode(nodeAlgeria);
graph.addNode(nodeTunisia);
graph.addNode(nodeTurkey);

// create connections
graph.addConnection(nodeTurkey, nodeTunisia,  import: 6054);
graph.addConnection(nodeTurkey, nodeAlgeria,  import: 7761);
```

The table representation for Turkey is given below. Since the relationship between Turkey and Tunisia is created first, it appears in front.

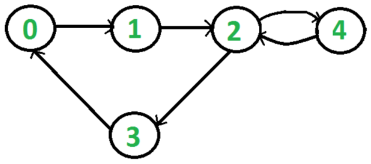


key	value	
Turkey	Edge { tailNode: NodeTunisia, import: 6054 }	Edge { tailNode: NodeAlgeria, import: 7761 }

- **list<Node> getAdjacentNodes(const Node& node);**
  getAdjacentNodes function gets a node as a parameter and returns all adjacent nodes of the given node as a list of Node. You should only consider the given node's adjacency list. Therefore, only consider the edges that are originated from the given node. If the input parameter does not exist in the graph, throw ItemNotFoundException().
- **long getTotalImports(const Node& node);**
  getTotalImports function gets a node as a parameter and returns the total imports from the given node. In other words, it returns the sum of the values of the edges that are originated from the given node.
- **void deleteNode(const Node& node);**
  deleteNode function deletes the given node from the graph together with its incident connections (edges). You may use "getKeys" function of "HashTable" to retrieve all active keys and delete the given node from their adjacency lists.
- **list<string> findLeastCostPath(const Node& srcNode, const Node& destNode);**
  findLeastCostPath function gets two nodes as parameter, calculates the least cost path from srcNode to destNode and returns the calculated path as a list of country names from srcNode to destNode. For this function, you should consider import values as the edge cost. If there are more than one shortest path between these nodes, it is enough to return only one of them. You may implement Dijkstra's weighted shortest path algorithm using priority queue of C++ Standard Template Library (STL). For instance the leastCostPath from Honduras to Guatemala is

  Honduras → El Salvador → Panama → Guatemala

- **bool isCyclic();**
  If the graph contains any cycles return true, otherwise return false. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$ and $k > 2$. For instance, in the given figure below, 0,1,2,3 and 2,4 are the graph cycles. Therefore, for this graph, isCyclic function must return true. You can use either DFS or BFS to traverse and you should mark which nodes you have visited so far. You may use additional data structures such as queue, stack, vector of stl library that are included in the Graph.h.



- **list<string> getBFSPath(const Node& srcNode, const Node& destNode);**
  getBFSPath function takes srcNode and destNode as parameters and returns the BFS path from srcNode to destNode as a list of country names. You may use queue of stl library to find the path. Note that in order to find correct result, in addConnection function, you should insert new edges to the end of the list. For example, lets find the BFS path from Turkey to Honduras. The adjacency list of Turkey is as follows:

  [Tunisia, Algeria, Greece, Romania, Jordan]

  The order of the adjacency list is preserved by the order of insertion. The only connection of Tunisia is Morocco and Algeria has no connections. Next, we need to check Greece's incident edges. Greece's adjaceny list is as follows:

  [Cyprus, Honduras, Romania, Jordan]

  Therefore, the output of getBFSPath should look like (in this order):

  Turkey, Tunisia, Algeria, Greece, Romania, Jordan, Morocco, Cyprus, Honduras

5 Regulations

1. You will use C++.
2. You are not allowed to use map or any other standard libraries except the included ones.
3. Use the classes that are currently included; do not include any class (other than those specified) from the C++ standard library
4. You are not allowed to modify already implemented functions and variables.
5. You can define private methods and variables if necessary.
6. If your submission fails to follow the specifications or does not compile, there will be a "significant" penalty of grade reduction.
7. In HashTable.h, those who do the operations (insert, delete, resize, get) without utilizing the hashing, there will be a "significant" penalty of grade reduction.
8. In Graph class, those who do not store adjacency list as HashTable, there will be a "significant" penalty of grade reduction.
9. Those who modify already implemented functions and those who insert other data variables or public functions and those who change the prototype of given functions will be penalized as well.
10. **Late Submission:** Late submission policy is stated in the course syllabus.
11. **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. Note that students of this course are bounded to code of honor and its violation is subject to severe punishment.
12. **Newsgroup:** You must follow the Forum (in Odtuclass) for discussions and possible updates on a daily basis.

6 Submission

1. Submission will be done via Moodle (cengclass.ceng.metu.edu.tr).
2. Do not submit a main function in any of your source files.
3. A test environment will be ready in Moodle.
   - You can submit your source files to Moodle and test your work with a subset of evaluation inputs and outputs.
   - Another set of test cases will be used for evaluation of your final grade. So, your actual grades may be different than the ones you get in Moodle.
   - Only the last submission before the deadline will be graded.