ENSF 694 – Summer 2024
Lab 5

Department of Electrical & Software Engineering
University of Calgary
Written by: M. Moussavi, PhD, PEng.

Objective:

This lab focuses on two important data structure concepts:

1. Hash Table, also known as a hash map, is a data structure that implements an associative array, a structure that can map keys to values. It is designed to provide efficient data retrieval by using a hash function.
2. AVL tree, which is a self-balancing binary search tree. It maintains the tree balanced to ensure the height of the tree remains logarithmic with respect to the number of nodes.
3. Graph data structures

Marking Scheme:

Exercise A	20 marks
Exercise B	20 marks
Exercise C	15 marks

Total marks: 55

Due Date: Friday August 2nd before 11:59 PM.

Exercise A – Building and Using a Hash Table

Introduction:
objective of this lab is to become familiar with hash tables. Hash tables are one of the common data structures categorized under the set structures. The idea is to provide a hash function that uses the key field of the record and maps the key to an address into a table called hash table. A good hash function spreads the records fairly randomly among the available addresses (ie, avoid functions that cluster records around certain addresses). If the entries in hash table spread uniformly among the buckets, they can be accessed in about O(1) time. In simple words, the hash function takes an input (the key) and returns an integer, which is typically used as the index in an array. This process converts the key into a hash code. The goal of the hash function is to distribute keys uniformly across the hash table to minimize collisions. Collisions occur when two keys hash to the same index. There are several strategies to handle collisions:

1. **Chaining:** Each bucket contains a list or another data structure that can store multiple key-value pairs. When a collision occurs, the new key-value pair is simply added to the list at the appropriate bucket.
2. **Open Addressing:** When a collision occurs, the algorithm searches for the next available bucket according to a certain probing sequence (e.g., linear probing, quadratic probing, double hashing).

One of common measures for a hash table packing density that indicates how full the hash table is. It is defined as the ratio of the number of stored elements to the total number of slots or buckets in the hash table. The other measure important measure is hashing efficiency that refers to how effectively the hash table performs its operations such as insertion, and lookup. Efficiency is largely influenced by the load factor and the method used to handle collisions.

The program in this assignment is supposed to:

1. Call a hashing function which takes the flight identifier as its input and produces an integer table address as its output. Experiment until you find a function that gives you a hash efficiency of at least 50%.
2. Create the hash table with the appropriate size. Allocate the array using dynamic memory allocation.
3. Use chaining to resolve collisions. Populate the hash table with data from the input text file, using a two-pass load.
4. Once the table is filled, calculate the packing density and the hash efficiency, and print these to screen.
5. Allow the user to interactively search for any record in the hash table and print the entire record to screen. The search will use the same hashing function as in step 1 and will use chaining to resolve collisions.

The program must be invoked from the command line as follows:

```
hashtable input.txt
```

Where `hashtable` is the name of the executable and the `input.txt` is the name of the input file. If the command line arguments are improperly specified, the program should display the following message on the screen: "`usage: hashtable input_file`", and abort. Be sure the specified file can be opened.

What to Do:
Copy the text files: `HashTable_tester.cpp`, `Node.h, List.h, List.cpp, Point.h, Point.cpp, Flight.h, Flight.cpp, HashTable.h`, and the text file `input.txt` that contains a number of records in the following format:

```
Flight Identifier         8 characters
Point of Origin           3 characters
Destination               3 characters
Departure Date            6 characters
Departure Time            4 characters
Craft Capacity            3 characters
```

If you read this file carefully, you will see that the implementation of member functions for class `HashTable`, defined in the header file `HashTable.cpp` are missing. Therefore, you should read the given function interface comments in the header file and write the implementation of all member functions. Also, in the file `HashTable_tester.cpp`, there is a global function called `read_flight_info`, which its implementation is missing, and you need to complete it. If you complete the missing functions, and compile and run the program, the following example of output put must be displayed, followed by interactive part with the user:

```
Packing Density: 2
Hash Efficiency: 0.833333
Hash Table Contents:
Bucket 0:
Flight Number: DELTA2332, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 10:45,
Capacity: 200
Flight Number: AMA11232, Origin: Otawa, Destination: Toronto, Date: 2024-05-30, Time: 00:45,
Capacity: 576
Flight Number: WJ12301, Origin: Calgary, Destination: Toronto, Date: 2024-05-30, Time: 2:45,
Capacity: 476
...
...

Bucket 2:
Flight Number: DELTA233, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 10:45,
Capacity: 200
...
...
Bucket 5:
Flight Number: WJ1230, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time: 2:45,
Capacity: 476

Enter flight number to search (or 'exit' to quit): WJ1230
Record found: Flight Number: WJ1230, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30, Time:
2:45, Capacity: 476

Enter flight number to search (or 'exit' to quit): DELTA233
Record found: Flight Number: DELTA233, Origin: Calgary, Destination: Edmonton, Date: 2024-05-30,
Time: 10:45, Capacity: 200

Enter flight number to search (or 'exit' to quit): quit
```

What to submit:

1. The copy of followings as part of you lab report in PDF format:
   a) your source code for HashTable member functions, and global function read_flight_info
   b) screenshot of your program output
   c) The calculation of the packing density and hash efficiency.
   d) Brief descriptions of your hashing function, including a justification for the technique you chose, and a discussion of how the function might be improved.
2. Your source code (cpp and .h files), in a zipped file

Exercise B – AVL tree

In this exercise you are going to complete an incomplete implementation of a C++ program that builds an AVL tree, allows to insert or search for a record (a key/value pair).

What to Do:

Download files `AVL_tree.h, AVL_tree.cpp,` and `AVL_tree_tester.cpp`. If you read these files carefully, you will see that implementation of some of the member functions are missing. Your task, in this exercise to complete the missing parts in the file `AVL_tree.cpp.`

Please notice that some the function implementations are supposed to be recursive. Also, in some cases such as function insert and find there are overloaded functions (two functions with the same name). One function is declared as public interface and the other function is declared as a helper function (a private function).

Please don't make any changes to the files: `AVL_tree.h,` and `AVL_tree_tester.cpp`
Your also recommended to start with completing the implementation of function insert, followed by its associated files righRotate and leftRotate, then other function.

What to Submit:

1. Submit your source code, `AVL_tree.cpp`, and a screenshot of your program's output, as part of your lab report in PDF format.
2. Your source code (.cpp and .h files) in zipped file.

Exercise C – Graph Data Structure

The objective of this exercise is to become familiar with the concept of graph structures. Graphs are important data structures in many engineering, science and business applications.

What to Do:

1. Copy files `graph.cpp, graph.h, test_graph.cpp, graph.txt,` from D2L. Then, compile, run, and observe the program output.

2. Study the header file called `graph.h` and try to understand the details of `class Graph`, two structures `Edge`, and `Vertex`, and the class called `PriorityQueue`. The priority will be used for the purpose finding shortest path in the function `Dijkstra`. It is also can be used in `unweighted` function that finds the shortest path for an unweighted graph (it sets the distance to 0 before calculates the shortest path.

3. When you run the give code, because of a few missing functions your program should generate the following output (red text is the user input):

```
Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 1

Enter the start vertex: A
A -> A     0   A
A -> B     inf   No path
A -> E     inf   No path
A -> C     inf   No path
A -> D     inf   No path
A -> M     inf   No path

Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 2
```

```
Enter the start vertex: A
A -> A     0   A
A -> B     inf   No path
A -> E     inf   No path
A -> C     inf   No path
A -> D     inf   No path
A -> M     inf   No path

Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2):3
// End of program
```

4. Noe, draw an activation record (AR) diagram for the program at point one in `addEdge` (you don't need to show the file I/O object in your diagrams). Also, **You don't need to submit this diagram, but drawing this AR diagram can help you to understand how the program builds graph objects.**

5. Complete the definition of member function `Dijkstra` that uses Dijkstra's algorithm to calculate the shortest distance from a designated vertex v to any vertex w connected to v. And, function unweighted that calculates unweighted-shortest path.

The program must receive the file name from command line:

```
graph graph.txt
```

Where `graph` is the name of the executable, and `graph.txt` is the name of the input file. The input file contains the graph data in the following format:

```
A B 10
A E 5
B C 1
B E 2
C D 4
D A 7
D C 6
E B 3
E C 9
E D 2
E M 100
```

The first column in this file is the name of a source vertex, the second column is the name of the destination vertex, and the third column is the cost/weight for an edge, from source vertex to the destination vertex.

To better understand how the program interacts with the user to display the possible distances between any vertex to any other vertex, the following lines show a sample run, with sample user inputs (in red). The first column is the start and end vertex, second column shows the distance and the third column show the path from start to end vertex.

```
Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 1

Enter the start vertex: A
A -> A     0   A
A -> B     1   A B
A -> E     1   A E
A -> C     2   A E C
A -> D     2   A E D
A -> M     2   A E M

Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 2

Enter the start vertex: A
A -> A     0   A
```

```
A -> B      8    A E B
A -> E      5    A E
A -> C      9    A E B C
A -> D      7    A E D
A -> M      105    A E M

Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 2

Enter the start vertex: C
C -> A      11    C D A
C -> B      19    C D A E B
C -> E      16    C D A E
C -> C      0     C
C -> D      4     C D
C -> M      116   C D A E M

Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 1

Enter the start vertex: C
C -> A      2    C D A
C -> B      3    C D A B
C -> E      3    C D A E
C -> C      0    C
C -> D      1    C D
C -> M      4    C D A E M

Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 1

Enter the start vertex: M
M -> A      inf    No path
M -> B      inf    No path
M -> E      inf    No path
M -> C      inf    No path
M -> D      inf    No path
M -> M      0    M

Choose the type of graph:
1. Unweighted Graph
2. Weighted Graph
3. Quit
Enter your choice (1 or 2): 3

// Program ended
```

What to hand in:

1. The copy of your source code, and the screenshot of your program output, as part of your lab report in PDF format. We may give you a different input file with more data. Which in this case you need to submit your program output, using this data set.
2. Your source code (.cpp and .h files), in a zipped file.