NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

Practical Examination (PE)

CS2040C Data Structure and Algorithm

8 Nov 2023 Time Allowed: 2 hours

VERY IMPORTANT:

- Please make sure your code can be compiled and runnable after you have submitted to Couresmology. Any crashing code will be given a zero mark.
- You are allowed to make at most 10 submissions for each part. Make sure to test your code in your own compiler before submitting to Coursemology. Do not use Coursemology as a debugger to test your code!
- You should stop modifying your code and start submitting 10 min before the end of the PE in order to avoid the "jam" on Coursemology. Best practice is to submit a version whenever you complete a task. It is your own responsibility to submit the code to Coursemology on time!
- Only standard input and output libraries are allowed. No other libraries, including STL, may be used.
- The PE consists of three parts. In each part, you will be provided with a MSVS project. In each project file, there will be ***one file*** to be submitted. You should modify and submit that file only by "select all" (ctrl-A) in that file and copy and paste it to the corresponding part in Coursemology. Your submission must be able to be compiled and run with the rest of the files as they were in the original project.
    - Except for Part 3 that you only need to submit one member function.
- **We will close the PE submission sharply when time runs out.** Make sure you save and submit your work a few minutes before it ends. The network will be jammed at the last few min of the assessment because everyone is submitting at the same time.
- *Leave the Coursemology **webpage open** at all the times.*
- Any variables used must be declared within some functions. You are not allowed to use global variables (variables that are declared outside all the functions). Heavy penalty will be given (see below) if you use any global variable.

Advice

- Manage your time well! Do not spend excessive time on any task.
- Read all the questions and plan before you start coding.
- Please save and backup your code regularly during the PE.
- It is a bad idea to do major changes to your code at the last 10 minutes of your PE.

Part 1 Linked List with O(1) Tail Access (30 marks)

You are provided with the trimmed skeleton code of Assignment 1. Your job is to implement the following three features. Note that, You will only get full marks if you can do the tasks in O(1) time.

Task 1 Insert at Tail (15 marks)

Implement the function `insertTail(x)` that will insert an item `x` at the tail (last item) of the list in O(1) time. You will get full marks only if other functions such as `headItem()` and `removeHead()` still works.

Task 2 Get the Middle Item (15 marks)

Implement the function `midItem()` to return the "middle" item in the linked list, that can be called at any time with O(1) running time. If there are n items in the list, the middle item is the i^{th} item with

$$i = \text{floor}(n/2).$$

Your function should work perfectly with all the given functions in the given List class.

You can assume the list is not empty in Task 1 and Task 3. Here is an example:

```
List l;

for (int i = 0; i < 10; i++)
{
        l.insertTail(i * 2 + 1);
        cout << "List: ";
        l.print();
        cout << "Mid = " << l.midItem() << endl;
}
```

```
List: 1
Mid = 1
List: 1 3
Mid = 1
List: 1 3 5
Mid = 3
List: 1 3 5 7
Mid = 3
List: 1 3 5 7 9
Mid = 5
List: 1 3 5 7 9 11
Mid = 5
List: 1 3 5 7 9 11 13
Mid = 7
List: 1 3 5 7 9 11 13 15
Mid = 7
List: 1 3 5 7 9 11 13 15 17
Mid = 9
List: 1 3 5 7 9 11 13 15 17 19
Mid = 9

Remove Head
List: 3 5 7 9 11 13 15 17 19
Mid = 11
List: 5 7 9 11 13 15 17 19
Mid = 11
List: 7 9 11 13 15 17 19
Mid = 13
List: 9 11 13 15 17 19
Mid = 13
List: 11 13 15 17 19
Mid = 15
List: 13 15 17 19
Mid = 15
List: 15 17 19
Mid = 17
List: 17 19
Mid = 17
List: 19
Mid = 19
```

Part 1 Submission

Copy and paste the entire `simpleLinkedList.cpp` file into Coursemology.

Part 2: Hash Table Linear Probing with Auto-resizing (36 marks)

You are provided with a class Hash Table with a hash function. In the beginning, your hash table will be initialized by a given table size = 8 and allocated some memory dynamically. You can assume that

- The hash function `h()` is implemented for you, in which , it is just the sum of the digits.
- We will only insert integers > 0 into a table
- An entry of the table is 0 if it's empty
- An entry of the table is -1 if it's deleted.
- All access is with the open addressing with linear probing scheme.

You should not modify any existing output for your given code (do not change all the lines with '`cout`'.).

Submission: Copy and paste your entire `HashTable.cpp` file into Coursemology.

However, you may or may not submit the two functions `hastTabletest1()` and `hastTabletest2()` because they will not be called and used to be grade.

Task 1 Linear Probing Hashing Operations (12 marks)

Your job is to implement linear probing for the functions insert, remove and exist. You can assume we will not insert duplicate items into the table.

- `insert(int x)`: Insert an integer x into the table.
- `remove(int x)`: Remove an integer x from the table. Print "`Fail to remove:`" (Given in the code) if the integer is not in the table before trying to remove it.
- `exist(int x)`: Check if a certain integer x is in the table. Return true or false accordingly.

If you call `hastTabletest1()`, there will be no collisions. However, if you call `hastTabletest2()`, there will be collisions. And the output should be like the followings:

```
Inserting 1
The new hash table: 0 1 0 0 0 0 0 0

Inserting 10
Collision at index 1
The new hash table: 0 1 10 0 0 0 0 0

Inserting 100
Collision at index 1
Collision at index 2
The new hash table: 0 1 10 100 0 0 0 0

Inserting 1000
Collision at index 1
Collision at index 2
Collision at index 3
The new hash table: 0 1 10 100 1000 0 0 0

Inserting 10000
Collision at index 1
Collision at index 2
Collision at index 3
Collision at index 4
The new hash table: 0 1 10 100 1000 10000 0 0

Inserting 100000
Collision at index 1
Collision at index 2
Collision at index 3
Collision at index 4
Collision at index 5
The new hash table: 0 1 10 100 1000 10000 100000
0
```

```
Removing 1
The new hash table: 0 -1 10 100 1000 10000 100000
0

Removing 10
The new hash table: 0 -1 -1 100 1000 10000 100000
0

Removing 100
The new hash table: 0 -1 -1 -1 1000 10000 100000
0

Removing 1000
The new hash table: 0 -1 -1 -1 -1 10000 100000 0

Removing 10000
The new hash table: 0 -1 -1 -1 -1 -1 100000 0

Removing 100000
The new hash table: 0 -1 -1 -1 -1 -1 -1 0
```

Note that you need to add the line "`Collision at index i`" in your code whenever there is a collision occurs in the insertion. (Please note that you do not need to print out that statement in `remove()` and `exist()`.) The string is given to you by the constant "`COL_STR`". Please just add the statement

```
cout << COL_STR << idx << endl;
```

if the collision is at index `idx`.

Task 2 Auto-resizing (24 marks)

Your hash table should auto-resize itself by the following rules. The table will be initialized with a size of `MIN_TABLE_SIZE` (= 8). Let the current size of the table be m and the number of items be n.

- After an insertion, if the number of items n > floor(m/2), then double the table size to 2m
- After removing an item, if the number of items n < floor(m/4), then half the table size to m/2
- You have to manage the memory well. The array size of your hash table (`_ht`) must be exactly m.

Note that:

- You will rehash the values in the original hash table into the new resized table according to their orders in the original hash table, not the order of insertion in their original table.
- The table size cannot be less than MIN_TABLE_SIZE (=8). So, if there are only 2 items in the table and m = 8, the table will not be shrunken.
- Also you need to print out the resizing statement correct by using the string `RESIZE_STR` (Given in the function `_resize()`.

Sample Outputs for `hastTabletest2 ()`.

```
Inserting 1
The new hash table: 0 1 0 0 0 0 0 0

Inserting 10
Collision at index 1
The new hash table: 0 1 10 0 0 0 0 0

Inserting 100
Collision at index 1
Collision at index 2
The new hash table: 0 1 10 100 0 0 0 0

Inserting 1000
Collision at index 1
Collision at index 2
Collision at index 3
The new hash table: 0 1 10 100 1000 0 0 0

Inserting 10000
Collision at index 1
Collision at index 2
Collision at index 3
Collision at index 4
Resizing to 16
Inserting 1
Inserting 10
Collision at index 1
Inserting 100
Collision at index 1
Collision at index 2
Inserting 1000
Collision at index 1
Collision at index 2
Collision at index 3
Inserting 10000
Collision at index 1
Collision at index 2
Collision at index 3
Collision at index 4
The new hash table: 0 1 10 100 1000 10000 0 0 0 0 0 0 0 0 0 0

Inserting 100000
Collision at index 1
Collision at index 2
Collision at index 3
Collision at index 4
Collision at index 5
The new hash table: 0 1 10 100 1000 10000 100000 0 0 0 0 0 0 0 0 0
```

```
Removing 1
The new hash table: 0 -1 10 100 1000 10000 100000 0 0 0 0 0 0 0 0 0

Removing 10
The new hash table: 0 -1 -1 100 1000 10000 100000 0 0 0 0 0 0 0 0 0

Removing 100
Resizing to 8
Inserting 1000
Inserting 10000
Collision at index 1
Inserting 100000
Collision at index 1
Collision at index 2
The new hash table: 0 1000 10000 100000 0 0 0 0

Removing 1000
The new hash table: 0 -1 10000 100000 0 0 0 0

Removing 10000
The new hash table: 0 -1 -1 100000 0 0 0 0

Removing 100000
The new hash table: 0 -1 -1 -1 0 0 0 0
```

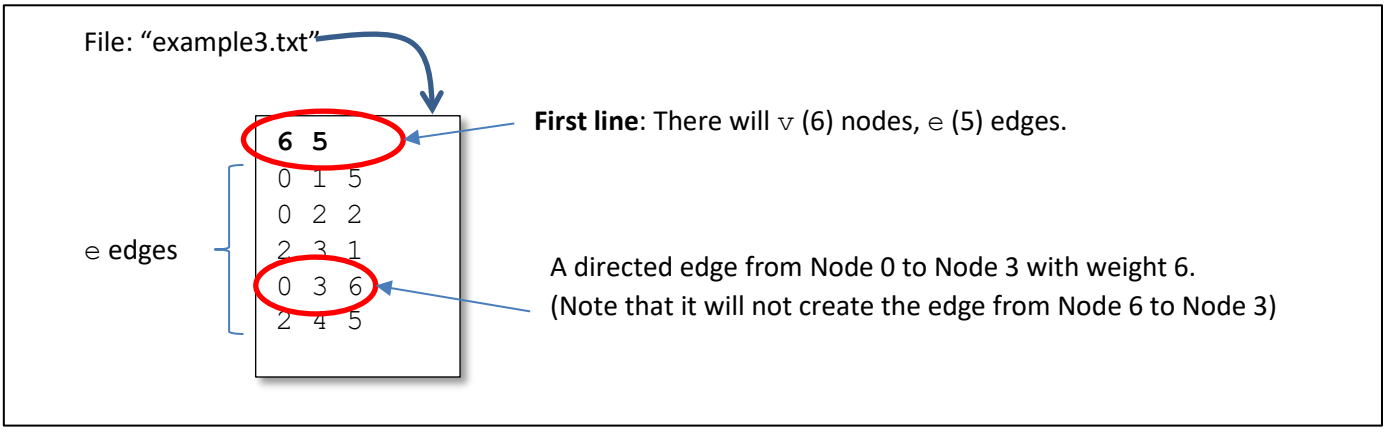
You should change the test code to change the integers added to the hash table and increased the number of numbers to be inserted into the hash table.

Part 3 SSSP & Negative Cycle Detection with Bellman-Ford Algorithm (34 marks)

In this part, we will compute the path and the distance of the shortest path between two nodes. You can assume our graphs are:

- Directed
- At least one node but no limitation on the number of edges.
- Each edge has a weight w that is less than 1000000. Namely, there will be edges with weight 0 or less
- You can assume all computations are in integers.
- You can assume the "infinity" estimated distance is 100000000 (`INFINITY_WEIGHT` given in the .h file).

Your graph (together with some SSSP queries) will be given in a text file. You can see there are a few files named "`example?.txt`" in your folder. Here is an example (example3.txt):



The first line will tell you the number of nodes, edges and SSSP queries. If there are n nodes, the node indices will be from 0 to n – 1.Then followed by `e` lines of numbers that each line represents an edge with a weight if `e` is the number of edges. Each edge has three number as the source, the destination and the weight of the edge.

Skeleton Code

You will be given the following code:

- The `Graph` Class: (`GraphBF.{h,cpp}`)
- The driver code (`main.cpp`) to read in the input file and start your computations.

And you only need to submit the shortest distance function `Graph::BF()` in the file `GraphBF.cpp`.

The Weighted Edge Class (Given)

There is a class `WeightedEdge` that has 3 public members:

- v1: the starting vertex of the edge
- v2: the ending vertex of the edge
- weight: the weight of the edge

The Graph Class (Given)

Instead of using adjacency lists or matrices, we just use an array of `WeightedEdge` to remember all the edges in the graph. The constructor will take in a file name and load in all the edges for you into the attribute `_edges`.

In this part, you should not change any other code except the function `Graph::BF()`.

Task

Write the function `Graph::BF(int v)` to compute the shortest path from the vertex v to all other nodes by the Bellman-Ford Algorithm. However, you need to:

- You should initialize the estimated distance to be `INFINITY_WEIGHT` except the source.
- Relax all the edges by the order of input (the order in _edges).
- After you relax one round of all the edges, you need to print out all estimated distance from the node v to all other nodes separated by commas.
- Repeat the relaxation. However, your BF algorithm should stop when there is no further improvement
- Stop and print the message `NEGATIVECYCLESTRING` if there is a negative cycle detected.

For example, with the given file of example2.txt, you should print out the followings

```
0,5,14,20,9,13,26,8
0,5,14,17,9,13,25,8
```

And example6.txt:

```
0,-3,100000000,100000000,100000000,100000000,100000000,100000000,100000000,100000000
0,-3,-2,100000000,100000000,100000000,100000000,100000000,100000000,100000000
0,-3,-2,0,100000000,100000000,100000000,100000000,100000000,100000000
0,-3,-2,0,3,100000000,100000000,100000000,100000000,100000000
0,-3,-2,0,3,7,100000000,100000000,100000000,100000000
0,-3,-2,0,3,7,12,100000000,100000000,100000000
0,-3,-2,0,3,7,12,18,100000000,100000000
0,-3,-2,0,3,7,12,18,25,100000000
0,-3,-2,0,3,7,12,18,25,33
```

And example7.txt:

```
-8,-4,-7,2
-16,-12,-15,-6
-24,-20,-23,-14
-32,-28,-31,-22
-40,-36,-39,-30
Negative Cycle!
```

Part 3 Submission

Please only copy and paste the function `void Graph::BF(int source)`. Do not submit the whole file.

--- End of Paper ---