NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING

Practical Examination (PE)

TIC2001 Data Structure and Algorithm

22 June 2019 Time Allowed: 2 hours

VERY IMPORTANT:

1. Please remember your coursemology login and password. You need to log in to coursemology at the start of the PE with the password and login id. All submissions are through coursemology.
2. You will be logged into a special Windows account at the beginning of the PE. Do not use your own NUSNET account. **Do not log off** until the end of the PE. Do not close your coursemology in your browser.
3. You are advised to start your Visual Studio even before you download your files form coursemology because it takes about 5 min or more to start VS for the first time.

INSTRUCTIONS

1. You are advised to arrive at the venue 10 min before the PE starts. You will be assigned to a specific seat and the seating plan will be posted in front of your venue.
2. This is NOT an open-book assessment. You can only bring one piece of A4 size paper, but **not** any electronic devices. You have to switch off/silence your mobile phone and **keep it out of view**.
3. Please read carefully and follow all instructions in each question. If in doubt, please ask. Raise your hand and the invigilator will attend to you.
4. Any form of communication with other students or the use of unauthorised materials is considered cheating and you are liable to disciplinary action.
5. When you are told to stop, please do so **immediately**, or you will be penalised.
6. **You will be forced to log out at the time the PE ends SHARPLY**. 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.
7. Please check and take your belongings (especially your student card) before you leave.
8. ***Your program must be able to be compiled!*** Or you will receive zero mark for that part.
9. If you are using iMac, you are advised to use VC 2015 for compatibility.
10. 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.
11. You may write additional function(s) not mentioned in the task statement if you think it is necessary.

Advice

- Manage your time well! Do not spend excessive time on any task.
- 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: Hash Table

A zipped file of the solution files for MS Visual Studio is provided which contains:

- The Hash Table code.
    - `HashTable.h`
    - `HashTable.hpp`
- The main driver class containing test codes.
    - `main.cpp`

IMPORTANT:

To facilitate grading, you may only modify the functions given in `HashTable.hpp`, and the hash function (h) given in HashTable.h (on top of the HashTable class). In other words, you are not allowed to modify anything else, add or remove your own classes.

Preliminaries

The following information must be considered when constructing your Hash Table.

1. We are only dealing with the insertion and deletion of positive integers (1, 2, 3, …)
    a. In other words, we will not be dealing with floats, negative numbers or 0.
2. Initially, all hash table entries are empty. Denote empty unwritten entries as 0.
3. Deleted entries are denoted by -1.
4. There are N entries in your Hash Table. This will be initialized when initializing your HashTable class.
    a. That is, HashTable ht(50) will initialize the Hash Table with a capacity of 50 items. Similarly, HashTable ht(x) will initialize with a capacity of x items.
    b. For the purposes of this question, assume that 1 <= N <= 100.
5. Collisions are resolved through linear probing.
6. There are no duplicate entries in the hash table. If a duplicate is inserted, then ignore it.
7. Test cases, with expected outputs, are given in main.cpp.

Question 1: Implementing the Hash Function

Implement the hash function `h(int x)` in `HashTable.h`, where x = positive integer to be hashed

Given `h(x)`, it returns the sum of digits in x

For example,

1. `h(25)` returns 7
2. `h(0)` returns 0.
3. `h(999)` returns 27

Question 2: Constructor of the Hash Function

In `HashTable.hpp`, complete the constructor of the `HashTable` class `HashTable(int n)`.

The constructor will initialize an empty `HashTable` of size n. You can assume that 1 <= n <= 100.

For example,

1. Calling `HashTable hashtable(10)` will initialize an empty hash table of size 10.
2. Similarly, calling `HashTable ht(50)` will initialize an empty hash table of size 50.

Question 3: Inserting Without Collision

In HashTable.hpp, complete the `insertWithoutCollision(int n)` function. This function will assume that there are no collisions, and insert n in the index given by the hash function h (from Question 1).

For example, given a hashTable of size 50 (N = 50)

1. `insertWithoutCollision(2)` will insert 2 in index 2.
2. Similarly, `insertWithoutCollision(11)` and `insertWithoutCollision(20)` will also insert into position 2. And the later will overwrite the former.
3. `InsertWithoutCollision(999995)` will insert into position 0 as `h(999995) mod 50 = 0`, if the size of the hash table is 50.

When copying your question to coursemology, you may assume that the hash function h has been defined and can be called from the function. Namely, you do not need to copy your hash function code.

Question 4: Delete Without Collision

In HashTable.hpp, complete the `deleteWithoutCollision(int n)` function. Again, this function assumes no collision and will delete the item n from the hash table if it is present. In addition, n may or may not already exist in the hash table.

Question 5: Exist Without Collision

In `HashTable.hpp`, complete the `existWithoutCollision(int n)` function. Again, this function assumes no collision and will return true if n is in the hash table, and false otherwise.

Question 6: Inserting With Collision

In `HashTable.hpp`, complete the `insertWithCollision(int n)`. This is similar to Question 3 (inserting without collision). But collisions are now resolved by linear probing.

For example, given a `hashTable` of size 50 (N = 50)

1. `insertWithCollision(2)` will insert 2 in index 2.
2. Similarly, given an empty hash table, `insertWithCollision(12)` and `insertWithoutCollision(21)` insert 12 and 21 into index 3 and 4, respectively
3. `InsertWithCollision(999995)` will insert into position 0 as `h(999995) mod 50 = 0`. Calling `insertWithCollision(55)` will insert 55 into index 10

You can assume that the Hash Table will have at least one empty bucket when performing insertion.

Question 7: Delete With Collision

In `HashTable.hpp`, complete the `deleteWithCollision(int n)` function. This is similar to Question 4, but now, collisions are resolved by linear probing.

If necessary, you can assume the `existWithCollision(n)` function works (which is to be defined in Question 8) and call it for this function when submitting your code on Coursemology.

Question 8: Exist With Collision

In HashTable.hpp, complete the `existWithCollision(int n)` function. Collisions are now resolved by linear probing.

Any other method of searching will be awarded 0 marks (for example, iterating through the entire hash table array).

Question 9: Is h a good hash function?

Is the hash function h given above a good hash function? Give two reasons to support your argument.

Explanation of Some Test Cases

Test cases from the main function can be commented out to test a particular test case. Expected output will be printed out also.

1. testInsertWithoutCollision
    a. Inserts (1, 2, 3, 4, 5, 6, 7, 8, 9, 444, 555, 88, 1234567, 9994) in that order. All the numbers won't collide with one another
2. testDeleteWithoutCollision
    a. Inserts (1, 2, 3, 4, 5, 6, 7, 8, 9) in that order.
    b. Then deletes 1, 3, 5, 9, 99
3. testExistWithoutCollision
    a. Inserts (1, 2, 3, 4, 5, 6, 7, 8, 9, 444, 555, 88, 1234567, 9994) then deletes 2
    b. Then check if { 1, 2, 5, 31, 444, 1234567, 93294 } exists in the hash table.
4. testInsertWithCollision
    a. Inserts (1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, 44, 555, 96, 11)
    b. Note that there are both duplicated and collision entries. Eg 11 collides with 2
5. testDeleteWithCollision
    a. Inserts (1, 2, 3, 4, 5, 6, 7, 8, 9, 44, 555, 96, 11)
    b. Deletes (1, 44, 555, 96, 9876)
6. testExistWIthCollision
    a. Insert (1, 2, 3, 4, 5, 6, 7, 8, 9, 44, 555, 96, 11) and deletes (1, 44, 555, 96, 9876)
    b. Check if `(1, 2, 5, 10, 11, 12, 14, 44, 9876) is in the hash table.`

Part 2: Sorting Linked Lists

In this problem, we will be implementing Bubble Sort and Merge Sort on a Single Linked List, implemented similarly with Lab Assignment 2.

A zipped file of the solution files for MS Visual Studio is provided which contains:

- The Linked List class similar to Lab Assignment 2.
    - `linkedlist.h`, `linkedlist.hpp`
- The main driver class containing test codes.
    - `main.cpp`

To facilitate grading, you may only modify the `bubbleSort()`, `split()`, `merge()` and `recursiveMergeSort()` functions. You may add your own functions, but you should not modify any other functions.

Part 2 Task 1: Implementing Bubble Sort

Given a Single Linked List, the function `bubbleSort()` sorts the linked list using bubble sort algorithm. This function takes in a boolean parameter `printAtEveryIteration`, which prints the entire linked list after each pass in the outer repetition statement (for-loop / while-loop).

```
Problem 2 Task 1: Bubble Sort
=============================
Current List with 10 elements: 90 80 70 60 50 40 30 20 10 0

Sorting the list using bubbleSort, with printing after each pass:
80 70 60 50 40 30 20 10 0 90
70 60 50 40 30 20 10 0 80 90
60 50 40 30 20 10 0 70 80 90
50 40 30 20 10 0 60 70 80 90
40 30 20 10 0 50 60 70 80 90
30 20 10 0 40 50 60 70 80 90
20 10 0 30 40 50 60 70 80 90
10 0 20 30 40 50 60 70 80 90
0 10 20 30 40 50 60 70 80 90

Sorted List with 10 elements: 0 10 20 30 40 50 60 70 80 90


Current List with 5 elements: 40 10 30 20 -20

Sorting the list using bubbleSort, without printing after each pass.

Sorted List with 5 elements: -20 10 20 30 40
```

Part 2 Task 2: Implementing splitting of linked list

The function `split()` divides the linked list into two linked lists. This function takes in a List pointer `otherList`. After splitting, the first half of the initial linked list will be pointed by the list that calls the function, and `otherList` will point to the second half. You can assume `otherList` will be always empty initially. You can also assume the initial list contains at least two nodes. If the initial list has odd number of elements, you should split the list as evenly as possible (e.g. 9 elements split into 5 and 4 elements, or 4 and 5 elements).

For example, the test function calls `firstList.split(&secondList)` which results in the following output.

```
Problem 2 Task 2: Split
=======================
First List with 10 elements: 90 80 70 60 50 40 30 20 10 0

Second List with 0 elements:

Splitting the list into firstList and secondList.

First List: 90 80 70 60 50

Second List: 40 30 20 10 0
```

Part 2 Task 3: Implementing merging of linked list

The function `merge()` takes in a List pointer `otherList`, and combines two non-empty sorted linked lists in ascending order into one linked list. After combining, the merged linked list is sorted in ascending order. You can assume both lists are non-empty initially. The `otherList` should be empty after merging.

For example, the test function calls `firstList.merge(&secondList)` which results in the following output.

```
Problem 2 Task 3: Merge
=======================
First List with 5 elements: 10 30 50 70 90

Second List with 5 elements: 0 20 40 60 80

Merging the two lists together.

Merged list with 10 elements: 0 10 20 30 40 50 60 70 80 90

Second list with no elements:
```

Hint: By inserting values through the head in ascending order, the resulting list will be in descending order. There is a function for you to reverse the list.

Part 2 Task 4: Implementing Merge Sort

The function mergeSort() calls a recursive function recursiveMergeSort(), which sorts the List in ascending order using merge sort algorithm. Make use of your implementation in the previous task to implement this function.

```
Problem 2 Task 4: Merge Sort
============================
Current List with 10 elements: 90 80 70 60 50 40 30 20 10 0

Sorting the list using mergeSort.

Sorted List with 10 elements: 0 10 20 30 40 50 60 70 80 90
```