CSE30: Lab #12 – Hash Tables

Overview

In this lab, we will explore a new useful data structure: **Hash Tables**. A hash table is a very efficient data structure to store and retrieve **unsorted data**. They are frequently used to implement unordered maps and dictionaries (multimaps). In this lab, you will implement a simple form of hash tables.

Getting started

Create a new directory in your main development directory (probably on Desktop/CSE30) called Lab_12. Try to use the terminal on your own, without getting help from the TA to setup the new directories (try to learn/remember the terminal commands).

The g++ syntax to compile classes is slightly different than for a single program comprised of a main (and potential functions):

```
g++ class1.h class1.cpp class2.h class2.cpp mainSource.cpp -o executable
```

where:

- g++ is the compiler (installed on your Linux system) of C++ source files,
- ***mainSource.cpp*** is the source file for your main program (main function),
- ***class1.h*** is the class declaration for your 1st class,
- ***class1.cpp*** is your 1st class definition,
- ***class2.h*** is the class declaration for your 2nd class,
- ***class2.cpp*** is your 2nd class definition (and so on...),
- -o tells the compiler that you want to give the executable its own name, and
- ***executable*** is the name you want to give your program.

As an example, if we have a main source file called ***Exercise1.cpp***, the class declaration called ***LinkedList.h***, the class definition called ***LinkedList.cpp***, and want to create an executable called ***aTestProgram***, you would type:

```
g++ LinkedList.h LinkedList.cpp Exercise1.cpp -o aTestProgram
```

Assuming that your program compiled successfully (i.e. no errors were found), you can run your program as you normally would by typing "***./aTestProgram***" in the terminal/console.

Good coding practices (worth 2 points!)

Writing code that is understandable by humans is as important as being correct for compilers. Writing good code will help you complete the code, debug it and … get good grades. It is very important to learn as soon as possible, because bad habits are hard to get rid of and good habits become effortless. Someone (guess who) reads your code will be in a better mood if it is easy to understand … leading to better grades! This lab will include 2 points (10% for code quality):

- Explanations with comments
- Meaningful names
- Indenting of blocks { } and nesting …
- Proper use of spaces, parentheses, etc. to
- Visible, clear logic
- One / simple statements per line
- Anything that keeps your style consistent

(Exercise)

In this part of the lab, you will be implementing the basic functions for a Hash Table, which are provided in the class declaration **HTable.h** (on UCMCROPS). In other words, you have to **create and implement the class implementation in a file called *HTable.cpp***. The main file you have to use for this lab is also provided on UCMCROPS (***Exercise.cpp***). ***DO NOT modify the class declaration (HTable.h) or main file (Exercise.cpp)***. Looking at the class declaration, you will find that a **data** is defined as a structure comprised of a ***key*** of type **int** and a ***value*** of type **string**. You will also notice (under the class declaration of HTable) that an array of data (***dt***) with a maximum size (***max_size***) is used as the hash table. A variable, ***numel***, is used to keep track of the number of data stored in the hash table. Note: ***numel*** is not always the same as ***max_size***.

In this part of the lab, you will need to implement the following functions for the **HTable** class:

- **Default Constructor**
    - Initializes the table (array) with a default size of ***23***.
    - An unoccupied space of the table will have a -1 as key and "N/A" as the value.
    - ***numel*** is set to 0.
- **Alternative Constructor**
    - Initializes the table (array) with a size equal to the input parameter.
    - An unoccupied space of the table will have a -1 as key and "N/A" as the value.
    - ***numel*** is set to 0.

- **hash(int &k)**
    - Evaluates the hash code according to the value of k.
    - Since k is an integer, we will simply use the modulo operator (%) to evaluate the hash code. It returns the hash code, which is the remainder of k divided by the

size of the table. This hash code will be used as the index value of the hash table when storing the data.

- **int rehash(int &k)**
    - In an event when a collision (multiple keys result to the same hash code) happens, a rehashing of the hash code is performed. We will implement a linear probing as the rehashing function. It returns a new hash code, which is the remainder of (k+1) divided by the size of the table. This will shift the index value of the table to the next available space.
- **int add(data &d)**
    - It adds a ***data*** structure into the table according to its key.
    - It evaluates the hash code according to the key of data.
    - If the space at the index represented by the hash code is available (key = -1), ***data*** is added to this space by setting the corresponding key and value of ***data*** to the table.
    - If the space is not available, rehash the hash code repeatedly until an available space is found or the end of table is reached.
    - It returns a 0 when a pair of data is added successfully; otherwise a -1 is returned when the table is full.
    - You need to keep track of the number of data added to the table by incrementing ***numel*** correctly. You also need to check if the table is full before adding any data to it.
- **int remove(data &d)**
    - It removes a ***data*** structure from the table according to the key of ***data***.
    - It locates the data by evaluating the hash code of the key. **DO NOT** simply remove the data according to the hash code! Data may not be stored at that location because of collisions. You need to compare the keys before removing any data. Rehash the hash code if necessary until you have reached the location where the key stored is the same as the key of ***data***.
    - To remove a data pair, set the data pair in the table to the initial values (key to -1 and value to "N/A").
    - You need to keep track of the number of data left in the table by decrementing ***numel*** correctly.
    - It returns a 0 when data is removed successfully; otherwise a -1 is returned when a data pair is not found. A data pair is not found when the repetition of rehashing reaches the end of the table and the key of data is not found.
- **output()**

- It prints out the content of the table, one data pair per line.
- It also prints out the number of data stored in the table.

Sample output from Exercise.cpp:

```
Data added.
Data added.
Data added.
Data added.
Data added.
Data added.
Data added.
Data added.
Data added.
Data added.
Data added.
Cannot add data...table is full.
Data removed.
Cannot remove data...Key not found.
Content of table:
0 ->  22     Books
1 ->  44     Clothing
2 ->  13     Movies
3 ->  33     Pets
4 ->  59     Computers
5 ->  70     Home Improvement
6 ->  -1    N/A
7 ->  18     Appliances
8 ->  41     Auto Parts
9 ->  31     Furniture
10 ->  32     Electronics
There are 10 data in the table.
```

What to hand in

When you are done with this lab assignment, you are ready to submit your work. Make sure you have included the following ***before*** you press Submit:

- A compressed file that includes **HTable.h**, **HTable.cpp**, **Exercise.cpp**, and a list of Collaborators.
- Documentation (in a text file) of code you used from the internet. You may want to cut-and-paste the original code as well.