# ECE250 Project 2: Hashing

**Due Date: Monday, November 4, 2024 @11:00pm to the Dropbox on Learn**

## Overview

In this project, you will implement a hash table data structure for storing simulated file blocks. File blocks are represented using a unique integer ID and a "payload" of at most 500 bytes. The payload is to be stored as an array of chars. In addition to the ID and payload, the block contains a "checksum", which is an integer whose value depends on the values of the bytes in the payload. When a file block is stored on the disk, a checksum is computed and stored along with it. Then, when the block is read, the checksum is recomputed based on what was read. If the stored checksum and the newly computed checksum match, we have reasonable confidence that the data was not corrupted[1]. In our case, we are computing the checksum as:

$$C = \left( \sum_{i=0}^{500} c_i \right) \%256$$

Where C is the checksum and $c_i$ is the integer value of the $i^{th}$ character. For this to work, your file block must be filled with zeros before any write operation occurs, so that bytes that are not part of the actual payload do not affect the calculation. You may notice that this is a form of hashing – it is! Although this isn't a great checksum function, computing checksums is a way to use hashing to check for errors.

File blocks are stored on a "disk", which we will represent as an array of fixed size containing file block objects. The array index of a given file block is found by hashing its ID.

You must create a C++ implementation for a hash table data structure. In this data structure, values are mapped to a position in a table using a hash function. For this project, you will implement hash tables in which collisions resolve using two different techniques: (i) open addressing using double hashing, and (ii) separate chaining where the chains are ordered by ID. We will provide you with primary and secondary hash functions in the next section, and both functions use the division method.

**You are permitted to use the vector class from the STL. No other STL classes may be used.**

### Hash Functions

**Primary Hash Function**

$h_1(k) = k \ mod \ m$, where *k* is the key (in this case our file block ID) and *m* is the size of the hash table. You may assume that m is a power of 2 for this project.

**Secondary Hash Function**

For open addressing, you will implement the hash table using the double hashing technique to resolve collisions. The secondary hash function is $h_2(k) = \left\lfloor \frac{k}{m} \right\rfloor mod \ m$. Since $h_2(k)$ must be an odd number, you will add 1 to the resulting value if this value is even.

## Program Design and Documentation

You must use proper Object-Oriented design principles to implement your solution. You will create a design using classes which have appropriately private/protected data members and appropriately public services (member functions). It is not acceptable to simply make all data members public. **Structs are not permitted.** Write a short description of your design to be submitted along with your C++ solution files for marking according to the template posted to Learn.

[1] Real-world warning: this is only true if your checksum function is good – that is, if there is a high probability that if the data has been corrupted the checksum will be different from what it was before the corruption. In this assignment we are simplifying the function to make it easier to implement and we really can't trust that corrupted data won't result in the same checksum.

# Input/Output Requirements

In this project, you must create a single test program, which must read commands from standard input and write the output to standard output. The commands are described below. Since there are two methods of hashing that are used, the first command in the input will always indicate the hashing method that is used for all subsequent commands.

The file IDs (keys) follow no specific format other than the fact that they will fit into a single unsigned integer. The charstring contents may contain any valid ASCII character, including spaces, except for the ! character.

| Command | Parameters | Description | Output |
|---|---|---|---|
| **NEW** | N T | Create a new hash table with size N. You may assume that this command appears exactly once, as the first command in the test file, and that N > 0 always. T is always either 0 (indicating that the file assumes you are using open addressing) or 1 (indicating the file assumes you are using separate chaining). | **success** |
| **STORE** | ID charstring | Stores a file block with ID given by ID and payload given by charstring, which you must read one character at a time. The charstring always ends with the character '!' (note that in reality this should be the zero character, but that is difficult to do in a text file). You may assume that no further output occurs after the exclamation mark on the given line. Do not store the exclamation mark. Upon successful insertion, compute the checksum for this block. Assume the charstring is always strictly less than 501 characters. The ! character does not count toward this length. The charstring will always have at least one non '!' character in it. | **success** if the insertion was successful: space exists in the hash table and the ID is not already present<br>**failure** if the insertion was unable to complete since the table was full or the key was already there |
| **SEARCH** | ID | Searches for the key ID in the table | **found** ID **in p:** if the desired key was found in the position *p* of the hash table. For separate chaining, indicate the chain's index, not the index in the chain.<br>**not found:** if the desired key was not found |
| **DELETE** | ID | Deletes the key ID from the table and its associated file block | **success** if the deletion was successful<br>**failure** if the deletion was unable to complete since the key was not found in the hash table |
| **CORRUPT** | ID charstring | CORRUPT operates the same as STORE except:<br>1. We are operating on an existing file block, found via its ID, ID<br>2. We set all chars in the file block to zero before writing<br>3. We write the new charstring but do not recompute the checksum | **success** if a file block with the given ID exists<br>**failure** if no file block with the given ID exists |
| **VALIDATE** | ID | Finds the file block with the given ID, if it exists. Computes the checksum and compares it with the stored checksum. | **valid** if a file block with the given ID exists and the newly computed checksum matches the existing one<br>**invalid** if a file block with the given ID exists but the newly computed checksum does not match the existing one<br>**failure** if no block exists with the given ID |
| **Command** | **Parameters** | **Description** | **Output** |

| | | | |
|---|---|---|---|
| **PRINT** | i | This command is only for separate chaining. It will not be used to test double addressing.<br><br>Prints the chain of IDs that starts at position i. Keys in the chain are separated by one space. Indicates that the chain is empty if that is the case. IDs are printed in numerically ascending order. You may assume i is a valid index for the chain. | Example 1 (PRINT 10, assuming that position 10 has 3 keys): **312 546 985**<br><br>Example 2 (PRINT 10, assuming the chain is empty): **chain is empty** |
| **EXIT** | | This command produces no output. This command will always end all input files. Once you encounter it, you may exit the program. | **This command produces no output.** |

You must analyze the average runtime of your algorithms in your design document. Assume for a given hash value, there are at most m collisions and m << T, the total number of elements inserted, and m is O(1). Then the runtime for each STORE, SEARCH, and DELETE operation is O(1). Prove that your implementation for both types of hashing achieves this runtime.

## Valgrind and Memory Leaks, Formatting, and Commenting

10% of the grade of this project will be allocated to memory leaks. We will be using the Valgrind utility to do this check. The expected behaviour of Valgrind is to indicate 0 errors and 0 leaks possible, with all allocated bytes freed. A penalty of 10% will be applied for poor commenting or code organization. A penalty of 15% will be applied if non-permitted header files or structs are used.

## Test Files

Learn contains some examples input files with the corresponding output. The files are named test01.in, test02.in and so on with their corresponding output files named test01.out etc.

All lines in the input files end with a UNIX newline character (\n), and there are no spaces before this character. This means that the last line of the file will be blank, and you may use *getline* to read input.

## Submitting your Program

Once you have completed your solution and tested it comprehensively on your own computer or the lab computers, you should transfer your files to the eceUbuntu server and test there. We perform automated tested on this platform, so if your code works on your own computer but not on eceUbuntu it will be considered incorrect. **A makefile is required for this project since the exact source structure you use will be unique to you. A video has been posted to Learn about how to create it.**

Once you are done your testing you must create a compressed file in the tar.gz format, that contains:

- A typed document, maximum of two pages, describing your design. Submit this document in PDF format. The name of this file should be xxxxxxxx_design_pn.pdf where xxxxxxxx is your maximum 8-character UW user ID (for example, I would use my ID “mstachow”, not my ID “mstachowsky”, even though both are valid UW IDs), and n is the project number. In my case, my file would be mstachow_design_p2.pdf.
- A cpp file containing your main() function. Your class implementations may not go into this file.
- Required header files that you created.
- Any additional support files that you created.
- A makefile, named Makefile, with commands to compile your solution and create an executable. This makefile will be run using the “make” utility. Do not specify an output file name, the default of a.out is used in the automated testing.

The name of your compressed file should be **xxxxxxxx_p2.tar.gz**, where xxxxxxxx is your UW ID as above.