**ECE 250 - Project 2 – Hashing tables – Design Document**

Easson Weisshaar, UW UsedID: etcweiss

**November 5th**

## Overview of classes

**Class:**

**student** - A class that allows for the storage of all relevant student information

**Private member variables:**

**SN**: a unsigned int that holds the student number, **lastname**: a string that holds the students lastname, **got_deleted**: a bool that stores whether the specific object was deleted, allows the program to recognize if a entry that was “inbetween collisions” was deleted

**Public member functions:**

**get_SN**: a function that returns an unsigned int of the entry’s student number, **get_LN**: a function that returns an entry’s lasntname as a string, **swap_deletion**: changes the status of a entry’s **got_deleted** value from false to true, **deleted**: returns whether a entry is deleted or not

**Class:**

**h_double** - A class that implements the usage of double hashing and includes appropriate functions to utilize

**Private member variables:**

***double_table**: a pointer to a vector of type student, **double_size**: a int that holds table size

**Public member functions:**

**add**: void function that adds a new **student** object to an appropriate vector in the table by calling hashing functions to determine the correct index, takes student number and lastname parameters, it overwrites an entry if the **deleted** status is true otherwise, runs until empty slot is located, **find**: a void function that searches the table for a given student number parameter by repeating the same hashing functions steps to locate the number, **remove**: very similar process to the **find** function except if located, the function calls **swap_deletion** on the **student** object to change its **deleted** status to true, **hash_1**: primary hash function that returns a parameter **key** modulus **double_size**, **hash_2**: The secondary hash function that returns the floor of the given parameter **key** divided by **double size**

**Class:**

**h_chaining** - A class that implements the usage of hashing by chaining and includes appropriate functions to utilize

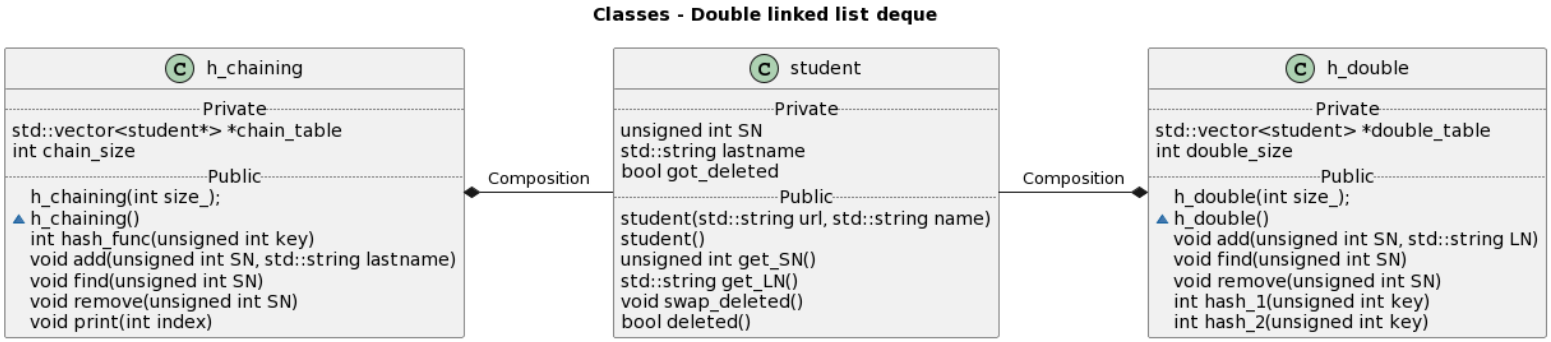
**Private member variables:**

***chain_table**: a pointer to a vector of type student, **chain_size**: a int that holds table size

**Public member functions:**

**hash_func**: the only used hashing function that returns an int from the parameter **key** modulus **chain_size** , **add**: a void function that takes in a student number and lastname as parameters, it creates a new student object with the given parameters, it determines a index to be added to from **hash_func** and will insert itself into the proper location in the array based on the student numbers stored previously, if an identical student number is stored, operation is terminated , **find**: a void function that takes in a student number parameter to search for, it determines an index from the use of **hash_func** and searches the correlating vector to determine a if it exists, **remove**: a void function that takes in a student number parameter to search for, it determines an index from the use of **hash_func** and searches the correlating vector to determine a if it exists. If found, **swap_deletion** gets called on the object to “delete” the entry , **print**: a void function that takes in a index number of the table to print and prints all the entries stored in descending order if any exist.

## UML diagram



**ECE 250 - Project 2 – Hashing tables – Design Document**

Easson Weisshaar, UW UsedID: etcweiss

**November 5th**

## Constructor/Destructor

**Student:**

- Given a unsigned int **SN** and string **lastname** argument, it sets the **SN** and **lastname** parameters to their respective arguments, also initializes **got_deleted** to false
- Default constructor that sets **lastname, SN,** and **got_deleted** to default values of “”, 0 and false respectively

**h_double:**

- Given an int; size, it creates a new vector of given size
- The destructor deallocates the memory used to make the table

**h_chaining:**

- Given an int; size, it creates a new vector of given size
- The destructor deallocates the memory used to make the table

## Test Cases

**student:**

- Given arguments, sets member variables to correct arguments

**h_double:**

add:

- Doesn’t allow more than 1 of a unique student number
- Prints failure when collisions equals table size (no more slots left)
- Able to read add a previously deleted student number
- Inserts to correct table slot
- After first collision correct double hashing procedures occur

find:

- Able to find a element after several collisions occurred
- Able to find element if a deletion occurred at a earlier slot that the element would’ve been added to had it been available
- Doesn’t find elements not inserted or elements previously deleted
- Doesn’t get stuck in infinite loops searching
- Finds correct elements

remove:

- Able to remove an element after many collisions
- Able to delete element if a deletion occurred at a earlier slot that the element would’ve been added to had it been available
- Deletes correct element
- Unable to “redelete” a previously deleted element
- Doesn’t delete non-present elements
- Doesn’t get stuck in infinite loops

**h_chaining:**

add:

- Doesn’t add duplicates
- Inserts elements in the correct index and corrent order within vector stored at index
- Able to re-add previously deleted elements

find:

- Searches correct index
- Doesn’t find non-present elements
- Doesn’t find elements after a deleted occurs

remove:

- Removes correct element in the ordered lists and doesn’t break the order after deletion

**ECE 250 - Project 2 – Hashing tables – Design Document**

Easson Weisshaar, UW UsedID: etcweiss

**November 5th**

- Doesn’t delete recently deleted elements
- Unable to ‘redelete’ a recently deleted element
- Doesn’t delete wrong elements
- Fails if element is not found

## Performance considerations

Under the assumption of uniform hashing the average runtime being constant was achieved. This is because under uniform hashing, the any key has a 1/size chance of being inserted into all the slots and therefore, on average, adding, searching and deleting would all be constant as there would theoretically be a runtime of $(1 + \alpha)$ where $\alpha$ is (number of elements)/(table size) which is constant be constant. Of course this is an ideal so that is why we consider the average runtime over the actual runtime. For example, adding a key in a slot on average results in no collisions (if we assume no. of elements = table size) so the index the key is added in would be solely dependant on the result of the hash function and not based on previous insertions, the same logic applies for finding and searching as finding the index would simply be the result of the hash function and no further steps based on the scanning an ordered list within or applying multiple iterations of a secondary hash function