ECE 250 Project 2: Hashing

Name: Kyle Lee
Student ID: 20892255
Date: Nov 7, 2022

1. Overview of Classes

a. Class Student

This class is composed of two private member variables called number and name that store the student number and student's last name, respectively. HashTable classes use this class to store student information. Once it is initialized, you can't alter the value of the number or name since there are no setters.

b. Template Class HashTable

This class represents a generic hash table that has a user-specified maximum capacity. It is designed as a template class to store the data of any data type. By implementing this class as a template class, it can be a parent class of both OpenHashTable and OrderedHashTable. This class is composed of basic member variables and functions that both OpenHashTable and OrderedHashTable have in common. The bucket of HashTable is an array of a template class. Since the size of the bucket never changes throughout the execution, I decided to implement it using a dynamically allocated array.

Member variables/functions:

- # size (dtype: int): Stores the number of entries in the bucket
- # max_size (dtype: int): Stores the maximum size of the bucket
- # array (dtype: T[ ]): Acts as a bucket of the table; stores data of type T
- # int hash_1 (unsigned int k): Primary hash function; returns (k mod max_size)

c. Class OpenHashTable

This class represents a hash table that resolves collisions using a technique called open addressing using double hashing. It is implemented by inheriting from the HashTable class. Since the data stored in this class is a Student class, its mother class is HashTable<Student>.

Member variables/functions:

- **+** void insert (unsigned int student_id, const string student_name):
  1. Check if the table is full; if it is full, print out "failure"
  2. Using hash functions ((h1(k) + i.h2(k)) mod max_size), find the appropriate index to insert the key
  3. If the key is already in the table, print out "failure"
  4. Else, insert the key and value and print out "success"
- **+** void search (unsigned int student_id):
  1. Using hash functions, search for the key in the table
  2. If the key was found in the position p of the table, print out "found LN in p"
  3. Else, print out "not found"
- **+** void remove (unsigned int student_id):
  1. Using hash functions, search for the key in the table
  2. If the key was found in the position p of the table, remove the key from the array and print out "success"
  3. Else, print out "failure"
- **-** int find (unsigned int student_id):
  1. Helper function for functions search and remove
  2. Using hash functions ((h1(k) + i.h2(k)) mod max_size), find the key in the table
  3. If the key was found in the positions p of the table, return p
  4. Else, return -1
- **-** int hash_2 (unsigned int k): Secondary hash function; returns ($\left\lfloor \frac{k}{\text{max\_size}} \right\rfloor$ mod max_size)

d. Class OrderedHashTable

This class represents a hash table that resolves collisions using a technique called separate chaining where the chains are ordered by student number. It is implemented by inheriting from the HashTable class. Since the data stored in this class is a vector<Student>, its mother class is HashTable<vector<Student>>. I chose to use vector class as a chaining container since its size can change dynamically, and it has methods such as emplace and erase where I can perform insertion and deletion from the position I want.

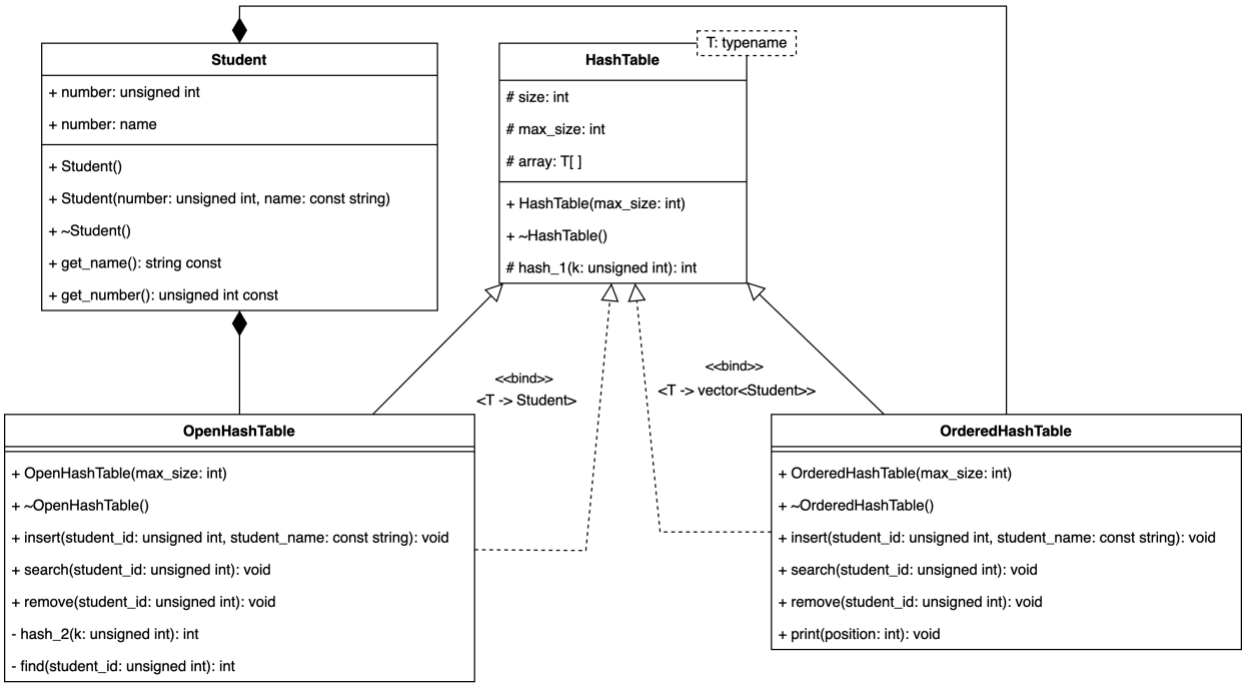
Member variables/functions:

- **+** void insert (unsigned int student_id, const string student_name):
  1. Using the primary hash function, find the appropriate index to insert the key and value
  2. If the key is the first key to be inserted in such position, simply emplace_back to the chain vector
  3. If the key is already in the chain, print out “failure”
  4. Else, by comparing the student_id (key) with keys in the chain, find the appropriate position to insert
  5. Insert the key and value and print out “success”
- **+** void search (unsigned int student_id):
  1. Using the primary hash function, find the chain in which the key is located from the table
  2. If the key was found in the chain at position p of the table, print out “found LN in p”
  3. Else, print out “not found”
- **+** void remove (unsigned int student_id):
  1. Using the primary hash function, find the chain in which the key is located from the table
  2. If the key was found in the chain, erase the key from the chain vector and print out “success”
  3. Else, print out “failure”
- **+** void print (int position):
  1. Print the chain of keys that starts at “position” in descending order
  2. Separate keys in the chain by one space.
  3. If the chain is empty, print “chain is empty”

Return types of all public member functions of Open/OrderedHashTable classes are void, which means, these classes never return a value. Instead, they will print out a message indicating the result of the operation in accordance with the project requirements.

*** +: public / #: protected / -: private**

2. UML Class Diagram



3. Design Decisions Details

a. Class Student

Constructor

- If parameters are passed, assigns passed parameters to number and name, else, assigns 0 and empty string to number and name, respectively.

Destructor

- Assigns 0 and empty string to number and name, respectively.

Function Examination

Since both getter functions just returns private member variables (no change in object), “const” keyword has been used to both functions.

b. Template Class HashTable

Constructor

- Dynamically allocates memory for array of class T

Destructor

- Deallocates the memory assigned to array

Function Examination

Function	Parameter Examination
hash_1	k (dtype: unsigned int): key to be hashed

c. Class OpenHashTable

Constructor

- Dynamically allocates memory for Student array

Destructor

- Deallocates the memory assigned to the array

Function Examination

Function	Parameter Examination
insert	student_id (dtype: unsigned int): key to be inserted student_name (dtype: const string): associated last name
search	student_id (dtype: unsigned int): key to be searched
remove	student_id (dtype: unsigned int): key to be deleted
find	student_id (dtype: unsigned int): key to be found
hash_2	k (dtype: unsigned int): key to be hashed

*Parameters that are passed using **“**const” keyword have no change in data during the function operation

d. Class OrderedHashTable

Constructor

- Dynamically allocates memory for vector<Student> array

Destructor

- Clears all keys in each vector and swaps with an empty vector
- Deallocates the memory assigned to the array

Function Examination

Function	Parameter Examination
insert	student_id (dtype: unsigned int): key to be inserted student_name (dtype: const string): associated last name
search	student_id (dtype: unsigned int): key to be searched
remove	student_id (dtype: unsigned int): key to be deleted
print	position (dtype: int): position of the chain to be printed

*Parameters that are passed using **“**const” keyword have no change in data during the function operation

4. Test Cases

It is important to test edge cases to confirm all functions are working as expected.

Open Addressing:

- Capacity Test: Insert a new key into the full table
- Collision Test: Insert a key that the result of primary hashing gives collision (secondary hash function should be called and find the appropriate index to insert the key)
- Duplication Test: Insert a key that is already in the table
- Deletion Test: Delete a key that is in the table or not in the table
- Search Test: Search for a key that is in the table, not in the table or deleted from the table

Separate Chaining:

- Duplication Test: Insert a key that is already in the table
- Collision Test: Insert a key that the result of primary hashing gives collision (key should be inserted into the chain in descending order)
- Deletion Test: Delete a key that is in the table or not in the table
- Search Test: Search for a key that is in the table, not in the table or deleted from the table
- Print Test: Print a key from an empty chain / Print to check if the keys are ordered in a descending order

5. Performance

Commands with a time complexity of O(1) are insert, search, and delete. A command “insert” inserts a new key into the hash table using the hash function. For both open addressing and separate chaining, assuming uniform hashing, finding the appropriate position to insert can be done in O(1), and inserting key/value also has a time complexity of O(1). Hence the resulting time complexity is O(1). A command “search” searches a key from the hash table by utilizing the hash function. Assuming uniform hashing, a searching operation can be done in O(1) for both open addressing and separate chaining. Finally, the command “delete” deletes a key from the hash table if it exists. Since it behaves like a “search” except for deleting the found key from the table, considering that deleting the key is executed with the time complexity of O(1), deleting operation can also achieve a time complexity of O(1).

Command "print" can’t achieve a constant time complexity since it needs to traverse the whole chain to print out all the keys. Hence, assuming uniform hashing, the length of each chain will be n/m. Then, the time complexity would be O(n/m).