Design - P2

Kehan Xing
UW id: k5xing Student#:20783902

(A) open addressing using linear probing

Class:

I designed a class named HashTable in this project. This class contains a vector member variable, and the variable type inside the vector is int. There is an int variable called current_size in the class as well, this variable is used to determine whether the hash table is full. There is also a constructor, a destructor and five functions in this class.

Member functions:

a. setsize function: this is a function which can set the size of the hash table, it requires an int parameter which represent size, and there's no output(void).

b. search function: this is a function which can check whether the input key is already existed in the hash table and get the position of the key if it is. It takes the key as an int parameter, and it returns the position as an integer as well. If the key is not in the hash table, it will return -1.

c. insert function: this function can insert a key using linear probing algorithm. It requires an int parameter that represent the key value, and there's not output(void) for the function.

d. erase function: this function takes an integer as the value of the key, and it will delete the key with that value (if there is one). It takes an integer as the input, and there's no output(void).

e. print function: this function is written for debugging. It can print the hash table, and there's no input required and no output(void).

Constructor and Destructor:

There's one constructor and one destructor. The constructor is a default constructor which takes no input, and it set the variable current_size to 0 and the size of the vector hashtable to 0. The destructor takes no input as well, and it deletes the hash table by setting current_size to -1 and the size of the vector hashtable to -1.

Design decision related to running time:

In this project, it is expected that the running time for every function is O(1). Therefore, my search function will start searching from hashtable[key % size], and it will continue searching until it finds the specific value or it reaches to a spot with value -1 which represent this spot is empty and never used. I used this algorithm instead of searching the whole hashtable because this will reduce the running time, especially when the size of the hashtable is big.

Also, the method I used to determine whether the hashtable is full is creating a variable called current_size, this variable will increase 1 after a successful insertion and decrease 1

after a successful deletion. Instead of going through the entire hashtable, this method takes less running time.

(B)separate chaining where the chains are unordered

Classes:

I designed two classes in this algorithm: HashList and HashTable.

(i) Class HashList:

This class creates a linked list for the keys have the same mod value (which means key % size). Inside the HashList class, there's a struct called HashNode, and it represents every node in the linked list. The HashNode struct has an int variable called key and a HashNode* variable called next. The HashList class has two HashNode* variables which are called head and back, and they represent the first and the last key in the linked list. There are also three functions, a constructor and a destructor in this class

Member functions in HashList:

a. search_list function: this function can check whether the input key is already existed in the linked list. It takes an integer input as the key value, and the variable type of the output is bool which represent whether the key exists.

b. insert_list function: this function can insert a key to the back of the linked list. It requires an int parameter which is the key value, and the output type is bool which represents whether the insertion is successful.

c. erase_list function: this function takes an integer as the value of the key, and it will delete the key with that value (if there's one in the linked list). It takes an integer as the input, and the output type is bool which represents whether the deletion is successful.

.

Constructor and Destructor in HashList:

There's one constructor and one destructor. The constructor is a default constructor, and it takes no input. The constructor will set the variable front and back to NULL. The destructor takes no input as well, and it will delete the elements in the linked list by setting front and back to NULL.

(ii) Class HashTable

This class represents the hash table, and it contains a vector member variable called hashtable, and the variable type inside the vector is HashList. There are also 4 functions, a constructor and a destructor.

Member functions in HashTable:

a. setsize function: this function can set the size of the hash table, it requires an int parameter which represent the required size, and there's no output(void).

b. search function: this is a function which can check whether the input key is already existed in the hash table. It will find the corresponding linked list by finding the mod

value (key % size) and use the search_list function for that particular linked list. The function requires an int parameter as the key value, and there's no output (void).

c. insert function: this function can insert a key to the hash table. It can find the corresponding linked list by finding the mod value (key % size) and use the insert_list function to insert the key to the back of the linked list. This function takes an integer input which is the key value, and there's no output (void).

d. erase function: this function takes an integer as the value of the key, and it will delete the key with that value by finding the corresponding linked list and using the erase_list function. It takes an integer as the input, and there's no output(void).

Constructor and Destructor in HashTable:

There's one constructor and one destructor. The constructor is a default constructor which takes no input, and it will set the size of the vector hashtable to 0. The destructor takes no input as well, and it deletes the hash table by setting the size of the vector hashtable to -1.

Design decision related to running time:

In every linked list, we must go through the entire linked list to search since the keys are unordered. However, I made sure the program only needs to search through one linked list in the hash table. I found the mod value (key % size) before searching, inserting and deleting, and the program will only deal with the linked list of hashtable[key % size], and this will reduce the running time.