Chrystal Mingo

CSC 21200

Final Project

Github: https://github.com/chrystalmingo/CSC212Final

Hash Tables

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. Hashing is the transformation of a string of characters into a usually *shorter fixed-length value or key* that represents the original string.

For my project I implemented a Hash Table which could do the following: create a hash, add items to the hash table, prints the number of indexes and the number of items in that location, print the Hash Table, print all the items that fall under one index in the hash table, a function that would allow me to search for my key, and remove items from my Hash Table.
In the end my project collected student names, and used the name of their college as a key. My hash table did a great job of allowing me to store the student's information as well, to search for it immediately thanks to the hash function.

I am going to explain the purpose of each of my functions, found in my hash.cpp. I implemented a hash table because to me it was the perfect data structure to store information, and allows me to easily and efficiently access my elements within the table. The first function was the hash function. This function called *hash*, is what hashes the strings. Taking the key, then its length. The for loop adds up all the values that represent the each character of the string. I took the sum of the word and mod it by the size of the hash table, to find a unique hash key (or

index) within range for the string. This is the process of hashing, with a time complexity of O(n).

```
int hash::Hash(string key){
    int hash = 0;
    int index;
    index = key.length(); //index is represented by the lenght of the string
    for (int i = 0; i< key.length(); i++){ //Adds up all the characters in the string
        hash = hash + (int)key[i];
     }
     //key[0]; would print the first letter of the string
     //now when you do (int)key[0 or any other index] you get the integer that represents that number
     index = hash % tableSize;
     //example if key total = 402 / 100(table size) = 4 r 2 and you store that remainder value as the index in the hash table
```

The second function called *AddItem,* would add Items to the Hash Table, the item would include the person's name and college. This function has a time complexity of O(n)

```
void hash::AddItem(string name, string college){
    int index = Hash(name);//hold the location in the hash table that stores the int of that string
    if(HashTable[index]->name == "empty"){ //over writing
        HashTable[index]->name = name;
        HashTable[index]->college = college;

    }
    else {
        item* Ptr = HashTable[index]; //point to the begginning of that table
        item* n = new item; //Adding new item with name & college
        n->name = name;
        n->college = college;
        n->next = NULL; //make the next item point to NULL;
        while(Ptr->next != NULL){//will make the pointer transverse till the end of the list/hashTable
            Ptr = Ptr->next;
        }
        Ptr->next = n; //links the last item in that list with the new item in the hash table
    }
}
```

The next item was called the *NumberOfItemsInIndex* which informs you of how much Items are in one Index of the Hash Table. This function has a time complexity of O(

```
int hash::NumberOfItemsInIndex(int index){
    int count = 0; //initialize the counter at 0
    if(HashTable[index]->name == "empty"){ //if we haven't put anything in then return the default
        return count;
    }
    else{
        count ++; //incrementing count
        item *Ptr = HashTable[index]; //point to beginning of the list
        while(Ptr->next != NULL){ //As long as the next pointer is not empty
            count ++;//increment as long as something is attached
            Ptr = Ptr->next;//continue to transverse to the next
        }
    }
```

The next two called *PrintTable* and *PrintItemsInIndex*, the first one printed the entire hash table, and it's items. This first function has a time complexity of O(n). The next one would print the Items if a Index had more than one item. This function has a time complexity of O(n)

```
void hash::PrintTable(){//Will Print out all the information in the HashTable
    int number; //hold the num of elements in each buckets
    for(int i =0; i < tableSize; i++){
        number = NumberOfItemsInIndex(i);
        cout <<"------------------\n";
        cout<<"index = " << i << endl;
        cout<<HashTable[i]->name<<endl;
        cout<<HashTable[i]->college<<endl;
        cout<<"# of Items = "<<number<<endl;
        cout <<"------------------\n";
    }
}
```

```
void hash::PrintItemsInIndex(int index){
    item* Ptr = HashTable[index]; //points to the first item in the bucket/hash table
    if(Ptr->name == "empty"){
        cout<<"index = "<< index << " is empty"<< endl;
    }
    else{
        cout <<"index = "<<index<< "contains the following items"<<endl;
        while(Ptr != NULL){
            cout <<"------------------\n";
            cout<<Ptr->name<< endl;
            cout<<Ptr->college<<endl;
            cout <<"------------------\n";
            Ptr = Ptr->next;
        }
    }
```

The remove function, was able to delete the entire hash table, and specific cases. The last function was called FindCollege, as I stated before College was my key, with this function I was able to search for my key, which is one of the great capabilities of hash tables, since it allows one to freely search for a element, or item, or in this case someone's information relating to their name and institution they attend. This function has a time complexity of O(n).

```
void hash::FindCollege(string name){
    int index = Hash(name);
    bool foundName = false; //Name not found
    string college;
    item* Ptr = HashTable[index]; //pointing to the first item in the bucket
    while (Ptr!= NULL){
        if(Ptr->name == name){
            foundName = true;
            college = Ptr->college;
        }
        Ptr = Ptr->next;
    }
    if(foundName == true){
        cout<<"College = "<<college<<endl;
    }
    else{
        cout<< name <<" not found in the hash table"<<endl;
    }

}
```

I decided to implement the Hash Tables to learn about how hashing works, how to create a hash, display the table with information, add information, just learn overall. It was definitely worth it, because it was quite similar to linked list and served as a refresher. Since I implemented a specific type of hash that's called chained hashing. I used chained hashing to be able to deal with the collisions my function might face if for example, people had the same name and went to the same college. Also it was amazing to see what different applications can be done to the program, I used it to gather information on students names and their colleges, but it could have gathered more information like the person's last name, instead of college looked into what country or nationality these individuals are from. I found hash table to be a perfect data structure to store and search for items easily. I also feel like I learned so much from implementing it, and overall feel like a stronger programmer in c++.

I used the Hash table because it could allow me to store and search for items efficiently. The Hash data structure capability to hash giving an element a unique key, fascinated me. I choose it to store information on students, but hashing is used day to day for real world applications such as Contact List, Cryptography, Message Digest, Hash values or codes, Blockchain, Hash Tables like I used it for, Password Verification and there is so much other uses for hashing.

The biggest issue with Hashing is collisions. As a result I used chained hashing, I dealt with collisions by creating a linked list within each array. Therefore if someone had the same name, and college I would would find each one by referencing the index, each one would be its own item within the hash table, and could be found in the same index, however the first one entered was in the first node, and the second one below. The challenge was this issue with collision but the chained hashing helped deal with the issues if items fell under the same "unique" key, the uniquenesses of the key stayed true because they fell within their own unique index within that unique key.

The overall benefit of hash tables is their *speed*, its capability to access elements in a short amount of time because the elements are hashed away and the unique keys allow faster access to elements. With a time complexity of O(n) hash tables are quite efficient. For this project I could have used Binary Search Trees to search for my elements, however I choose a hash table because of its ability to store large sums of information, and allows for easy access.