GROUP A

Assignment 01

Problem Statement:

Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up cliens telephone number

Objectives:

1. To understand concept of Hashing
2. To understand to find record quickly using hash function.
3. To understand concept & features of object oriented programming.

Learning Objectives

To understand concept of hashing.

To understand operations like insert and search record in the database.

To understand the collision handling technique.

Learning Outcome

Learn object oriented Programming features

Understand & implement concept of hash table .

Theory:

Hash tables are an efficient implementation of a keyed array data structure, a structure sometimes known as an associative array or map. If you're working in C++, you can take advantage of the STL map container for keyed arrays implemented using binary trees, but this article will give you some of the theory behind how a hash table works.

Keyed Arrays vs. Indexed Arrays

One of the biggest drawbacks to a language like C is that there are no keyed arrays. In a normal C array (also called an indexed array), the only way to access an element would be through its index number. To find element 50 of an array named "employees" you have to access it like this:

```
1employees[50];
```

In a keyed array, however, you would be able to associate each element with a "key," which can be anything from a name to a product model number. So, if you have a keyed array of employee records, you could access the record of employee "John Brown" like this:

```
1employees["Brown, John"];
```

One basic form of a keyed array is called the hash table. In a hash table, a key is used to find an element instead of an index number. Since the hash table has to be coded using an indexed array, there has to be some way of transforming a key to an index number. That way is called the hashing function.

Hashing Functions

A hashing function can be just about anything. How the hashing function is actually coded depends on the situation, but generally the hashing function should return a value based on a key and the size of the array the hashing table is built on. Also, one important thing that is sometimes overlooked is that a hashing function has to return the same value every time it is given the same key.

Let's say you wanted to organize a list of about 200 addresses by people's last names. A hash table would be ideal for this sort of thing, so that you can access the records with the people's last names as the keys.

First, we have to determine the size of the array we're using. Let's use a 260 element array so that there can be an average of about 10 element spaces per letter of the alphabet.>

Now, we have to make a hashing function. First, let's create a relationship between letters and numbers:

A --> 0

key mod Table size

B --> 1

sum (string) mod 7.

C --> 2

D --> 3

eg. ab = 1+2 = 3 mod 7 = 3
cd = 3+4 = 7 mod 7 = 0
efg = 5+6+7 = 18 mode 7 = 4

...

and so on until Z --> 25.

0	1	2	3	4	5	6	7
cd			ab/	efg			

The easiest way to organize the hash table would be based on the first letter of the last name.

Since we have 260 elements, we can multiply the first letter of the last name by 10. So, when a key like "Smith" is given, the key would be transformed to the index 180 (S is the 19 letter of the alphabet, so S --> 18, and 18 * 10 = 180).

Since we use a simple function to generate an index number quickly, and we use the fact that the index number can be used to access an element directly, a hash table's access time is quite small. A linked list of keys and elements wouldn't be nearly as fast, since you would have to search through every single key-element pair.

Basic Operations

Following are the basic primary operations of a hash table.

Search – Searches an element in a hash table.

Insert – inserts an element in a hash table.

delete – Deletes an element from a hash table.

DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem

{

int data;

int key;

};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){

return key % SIZE;

}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Example

```
struct DataItem *search(int key)

{

//get the hash

int hashIndex = hashCode(key);


//move in array until an empty

while(hashArray[hashIndex] != NULL) {


if(hashArray[hashIndex]->key == key)

return hashArray[hashIndex];


//go to next cell

++hashIndex;


//wrap around the table

hashIndex %= SIZE;

}


return NULL;
}
```

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Example

```
void insert(int key,int data)

{

struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));

item->data = data;

item->key = key;


//get the hash

int hashIndex = hashCode(key);


//move in array until an empty or deleted cell


while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) { //go to next cell


++hashIndex;


//wrap around the table

hashIndex %= SIZE;

}

hashArray[hashIndex] = item;

}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
int key = item->key;


//get the hash
int hashIndex = hashCode(key);
//move in array until an empty
while(hashArray[hashIndex] !=NULL) {
if(hashArray[hashIndex]->key == key) {
struct DataItem* temp = hashArray[hashIndex];
//assign a dummy item at deleted position
hashArray[hashIndex] = dummyItem;
return temp;
}
//go to next cell
++hashIndex;
//wrap around the table
hashIndex %= SIZE;
}


return NULL;
}
```

Collisions and Collision Handling

Problems, of course, arise when we have last names with the same first letter. So "Webster" and "Whitney" would correspond to the same index number, 22. A situation like this when two keys get sent to the same location in the array is called a collision. If you're trying to insert an element, you might find that the space is already filled by a different one.

Of course, you might try to just make a huge array and thus make it almost impossible for collisions to happen, but then that defeats the purpose of using a hash table. One of the advantages of the hash table is that it is both fast and small.

Conclusion: In this way we have implemented Hash table for quick lookup using C++.

Assignment 02

Problem Statement:

Implement all the functions of a dictionary (ADT) using hashing.

Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique

Standard Operations: Insert (key, value), Find(key), Delete(key)

Objectives:

1. To understand Dictionary (ADT)
2. To understand concept of hashing
3. To understand concept & features like searching using hash function.

Theory:

Dictionary ADT

Dictionary (map, association list) is a data structure, which is generally an association of unique keys with some values. One may bind a value to a key, delete a key (and naturally an associated value) and lookup for a value by the key. Values are not required to be unique. Simple usage example is an explanatory dictionary. In the example, words are keys and explanations are values.

Dictionary Operations

- **Dictionary create()**
  creates empty dictionary
- **boolean isEmpty(Dictionary d)**
  tells whether the dictionary **d** is empty

- **put(Dictionary d, Key k, Value v)**
  associates key **k** with a value **v**; if key **k** already presents in the dictionary old value is replaced by **v**

- **Value get(Dictionary d, Key k)**
  returns a value, associated with key **k**or null, if dictionary contains no such key

- **remove(Dictionary d, Key k)**
  removes key **k** and associated value

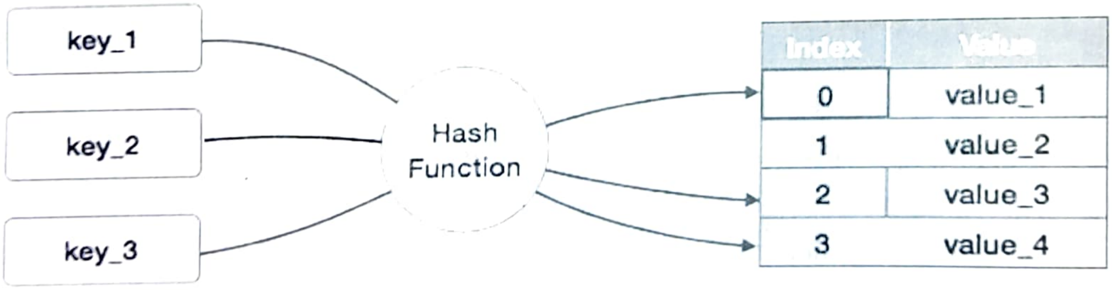
- **destroy(Dictionary d)**
  destroys dictionary **d**

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



Basic Operations of hash table

Following are the basic primary operations of a hash table.

- **Search** – Searches an element in a hash table.
- **Insert** – inserts an element in a hash table.

- **delete** – Deletes an element from a hash table.

1. DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
  int data;
  int key;
};
```

2. Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
  return key % SIZE;
}
```

3. Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

4. Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

5. Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
   int key = item->key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] !=NULL) {

      if(hashArray[hashIndex]->key == key) {
         struct DataItem* temp = hashArray[hashIndex];

         //assign a dummy item at deleted position
         hashArray[hashIndex] = dummyItem;
         return temp;
      }

      //go to next cell
      ++hashIndex;
```

Software Required: Dev C++ compiler- / 64 bit windows

Input: No. of. elements with key and value pair.

Output: Create dictionary using hash table and search the elements in table.

Conclusion: This program gives us the knowledge of dictionary(ADT).