ADVANCED DATA STRUCTURE

GROUP C

ASSIGNMENT 9

BATCH B1

YEAR: 2017-18

COLLEGE: VIIT

DATE OF COMPLETION : 30/03/2018

**Aim:**

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

**Objective:**

To construct Hash table and lookup for certain entry.

**Software And Hardware Requirement:**

1. 64-bit Open source Linux or its derivative.

2. Open Source C++ Programming tool like G++/GCC.

**Theory:**

Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed eﬃciently:

1. Insert a phone number and corresponding information.

2. Search a phone number and fetch the information.

3. Delete a phone number and related information.

We can think of using the following data structures to maintain information about different phone numbers.

1. Array of phone numbers and records.

2. Linked List of phone numbers and records.

3. Balanced binary search tree with phone numbers as keys.

4. Direct Access Table.

For arrays and linked lists, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.

With balanced binary search tree, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.

Another solution that one can think of is to use a direct access table where we make a big array and use phone numbers as index in the array. An entry in array is NIL if phone number is not present, else the array entry stores pointer to records corresponding to phone number. Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.

This solution has many practical limitations. First problem with this solution is extra space required is huge. For example if phone number is n digits, we need O(m \* 10n) space for table where m is size of a pointer to record. Another problem is an integer in a programming language may not store n digits. Due to above limitations Direct Access Table cannot always be used.

Hashing is the solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table. Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

A good hash function should have following properties

1. Eﬃciently computable.

2. Should uniformly distribute the keys (Each table position equally likely for each key)

For example for phone numbers a bad hash function is to take ﬁrst three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways. Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry. Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

1. Chaining:The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

2. Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

**Algorithm:**

Phone Directory Hashing

1. Enter a choice ’ch’ to choose from the choice menu.

2. If ch=1, insert ’name’ and ’phone number’ in the record.

To insert, first find the appropriate position in the hash table by passing ’name’ as argument to the hash funtion.

If a collision occurs then use separate chaining to resolve.

3. If ch=2, display all the records.

4. If ch=3, Search a specific record by the ’name’.

**Code :**

#include<iostream>

using namespace std;

int i,m,j,k;

int temp,key;

string c;

class hash\_t{

public:

int no;

string name;

hash\_t(){

no=0;

name='-';

}

}o[10];

class hash : public hash\_t{

int arr[20];

string n[20];

int index,v;

public:

int create();

void display();

void create\_hasht();

int search(int );

int search1(string);

int hashf(int temp){

index=temp%10;

return index;

}

};

int hash :: create(){

cout<<"\nEnter the telephone number : ";

cin>>arr[i];

cout<<"Enter name : ";

cin>>n[i];

}

void hash :: display(){

for(i=0;i<10;i++)

{

//cout<<n[i]<<"\t"<<arr[i];

cout<<o[i].no<<"\t"<<o[i].name<<endl;

}

}

void hash :: create\_hasht(){

i=0;

while(i<10 && i<m){

temp=arr[i];

c=n[i];

j=hashf(temp);

if(o[j].no==0)

{

o[j].no =temp;

o[j].name =c;

}

else

{

int k=0;

while(k<10)

{

if(o[j].no!=0)

j=(j+1)%10;

else

{

o[j].no =temp;

o[j].name =c;

break;

}

}

}

i++;

}

}

int hash :: search(int key){

j=hashf(key);

i=0;

int flag=0;

while(i<10 && i<m){

if(o[j].no==key)

{

cout<<"\nNumber found\n";

cout<<"Number : "<<o[j].no;

cout<<"\nName : "<<o[j].name;

flag=1; v=2;

break;

}

else

{

j=(j+1)%10;

}

i++;

}

if(flag==0)

cout<<"Number not found ";

}

int hash :: search1(string s){

for(i=0;i<m;i++){

if(n[i]==s)

search(arr[i]);

if(v==2)

return 0;

}

}

int main(){

int ans,temp1;

char choice;

hash b;

cout<<"Enter number of records : ";

cin>>m;

for(i=0;i<m;i++)

b.create();

//v=i;

b.create\_hasht();

do{

cout<<"\n1: Display hash table\n";

cout<<"2: Search\n";

cout<<"3: Update\n";

cout<<"4: Exit\n";

cout<<"Enter your choice : ";

cin>>ans;

switch(ans){

case 1 : b.display(); break;

case 2 : cout<<"Enter the telephone number to be searched : ";

cin>>key;

b.search(key);

break;

case 3 : cout<<"\na: Update telephone number ";

cout<<"\nb: Update name";

cout<<"\nEnter your choice : ";

cin>>choice;

switch(choice){

case 'a' : cout<<"Enter name : ";

cin>>c;

b.search1(c);

cout<<"Enter updated number : ";

cin>>temp1;

k=b.hashf(temp1);

if(o[k].no!=0)

{

for(i=0;i<10;i++)

{

k=(k+1)%10;

if(o[k].no==0)

break;

}

}

o[k].no=temp1;o[k].name=c;

o[j].no=0; o[j].name='-';

break;

case 'b' : cout<<"Enter number : ";

cin>>temp1;

b.search(temp1);

cout<<"Enter updated name : ";

cin>>c;

o[j].name=c;

break;

}break;

}

}while(ans!=4);

}

**Output:**

**Enter number of records : 2**

**Enter the telephone number : 2580**

**Enter name : vijay**

**Enter the telephone number : 9989**

**Enter name : divya**

**1: Display hash table**

**2: Search**

**3: Update**

**4: Exit**

**Enter your choice : 1**

**2580 vijay**

**0 -**

**0 -**

**0 -**

**0 -**

**0 -**

**0 -**

**0 -**

**0 -**

**9989 divya**

**1: Display hash table**

**2: Search**

**3: Update**

**4: Exit**

**Enter your choice : 2**

**Enter the telephone number to be searched : 2580**

**Number found**

**Number : 2580**

**Name : vijay**

**1: Display hash table**

**2: Search**

**3: Update**

**4: Exit**

**Enter your choice : 2**

**Enter the telephone number to be searched : 5677**

**Number not found**

**1: Display hash table**

**2: Search**

**3: Update**

**4: Exit**

**Enter your choice : 3**

**a: Update telephone number**

**b: Update name**

**Enter your choice : a**

**Enter name : divya**

**Enter updated number : 1234**

**1: Display hash table**

**2: Search**

**3: Update**

**4: Exit**

**Enter your choice :4**

**Conclusion**

Through this assignment we could implement telephone directory using hashing and perform operations like insert and lookup.