**DATA STRUCTURES AND FILES**

**Assignment-9 : Dictionary using Hashing.**

1. **Problem Statement:**

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

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

Standard operations:

* + Insert(key, value)
  + Find(key)
  + Delete(key)

# **Objective:**

Able to understand and use:

* + Basic concepts of hash tables and hash functions.
  + Basic collision resolution techniques in hash tables.
  + Implementation of various forms of hash tables i.e. separate chaining and linear probing.

# **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.

Methods that a dictionary can have:

* + Dictionary create()
  + boolean isEmpty(Dictionary d)
  + void put(Dictionary d, Key k, Value v)
  + void get(Dictionary d, Key k)
  + Key remove(Dictionary d,Key k)
  + void destroy(Dictionary d)

## **Hash tables:**

A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list *Θ*(*n*) time in the worst case in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is *O*(1).

## **Hash functions:**

Hash function is very important part of hash table design. Hash function is considered to be good, if it provides uniform distribution of hash values. Other hash function’s properties, required for quality hashing will be examined in detail later. The reason, why hash function is a subject to the principal concern, is that poor hash functions cause collisions and some other unwanted effects, which badly affect hash table overall performance.

## **Hash table and load factor:**

Basic underlying data structure used to store hash table is an array. The load factor is the ratio between the number of stored items and array’s size. Hash table can whether be of a constant size or being dynamically resized, when load factor exceeds some thresh- old. Resizing is done before the table becomes full to keep the number of collisions under certain amount and prevent performance degradation.

## **Collisions and Resolutions:**

What happens, if hash function returns the same hash value for different keys? It yields an effect, called collision. Collisions are practically unavoidable and should be considered when one implements hash table. Due to collisions, keys are also stored in the table, so one can distinguish between key-value pairs having the same hash. There are various ways of *collision resolution* . Basically, there are two different strategies:

* + **Closed addressing (open hashing):**

Each slot of the hash table contains a link to another data structure (i.e. linked list), which stores key-value pairs with the same hash. When collision occurs, this data structure is searched for key-value pair, which matches the key.

* + **Open Addressing (closed hashing):**

Each slot actually contains a key-value pair. When collision occurs, open addressing algorithm calculates another location (i.e. next one) to locate a free slot. Hash tables, based on open addressing strategy experience drastic performance decrease, when table is tightly filled (load factor is 0.7 or more).

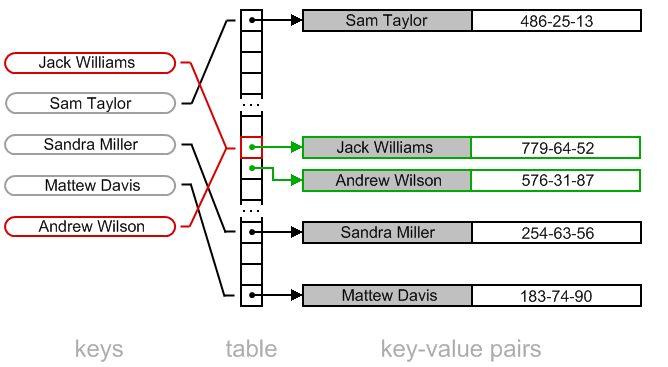
* + 

fig: Linear probing (Open addressing)

# **Algorithms:**

1. void input()
   1. Start
   2. while char c == ’y’:
      1. Create new node
      2. Take input the word and it’s meaning
      3. call hash(word) and hash at index recieved
      4. Continue? (y/n)
   3. Stop
2. int hash(string word)
   1. Start
   2. while i is less than length of word:
   3. Add the ASCII values of characters of word
   4. mod the sum by 10 to find the hash index
   5. return hash index
3. void deletefunc()
   1. Start
   2. Take input the word and calculate it’s index by hash(word)
   3. starting from the index found, find the word to be deleted
   4. if found, delete the node and return
   5. else, print “Element not found” and return (f) Stop
4. void search()
   1. Start
   2. Input the word
   3. find the index of the word by hash(word)
   4. Starting from the index, look for the given word
   5. if found, print it’s meaning (f) else, “print word not found”

(g) Stop

1. void display()
   1. Start
   2. while i is less than size of hash table:
      1. if node is empty, skip node
      2. else, print it’s word and meaning
   3. Stop

# **Code:**

# 

# #include<iostream>

# #include<string>

# using namespace std;

# class hashing

# {

# typedef struct node

# {

# string word;

# string meaning;

# struct node\* next;

# }node;

# node \*New,\*temp;

# int i;

# node \*arr[10];

# public:

# hashing()

# {

# i=0;

# New=temp=NULL;

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

# {

# arr[i]=NULL;

# }

# }

# void input()

# {

# char ans;

# do

# {

# New=new node;

# New->next=NULL;

# cout<<"\nEnter word :";

# cin>>New->word;

# cout<<"\nEnter meaning :";

# cin>>New->meaning;

# int x=hash(New->word);

# if(arr[x]==NULL)

# arr[x]=New;

# else

# {

# temp=arr[x];

# while(temp->next!=NULL)

# temp=temp->next;

# temp->next=New;

# }

# cout<<"\nContinue?";

# cin>>ans;

# }while(ans=='y');

# }

# void deletefunc()

# {

# string delword;

# int flag=0;

# cout<<"\nEnter the word to be deleted :";

# cin>>delword;

# int x=hash(delword);

# temp=arr[x];

# while(temp!=NULL && temp->word!=delword)

# {

# temp=temp->next;

# }

# if(temp==NULL)

# cout<<"\nWord not present in dictionary.\n";

# else

# {

# node \*p;

# if(flag==0)

# {

# p=temp;

# temp=temp->next;

# arr[x]=temp;

# delete(p);

# display();

# }

# else

# {

# p=temp->next;

# temp->next=(temp->next)->next;

# delete(p);

# display();

# }

# }

# }

# int hash(string word)

# {

# int i=0,sum=0,x=0;

# while(i<word.length())

# {

# x=word[i];

# sum+=x;

# i++;

# }

# sum=sum%10;

# return sum;

# }

# void search()

# {

# string person;

# cout<<"\nEnter the word to find it's meaning :";

# cin>>person;

# int x=hash(person);

# temp=arr[x];

# while(temp!=NULL && temp->word!=person)

# temp=temp->next;

# if(temp==NULL)

# cout<<"\nWord not present in dictionary.\n";

# else

# cout<<"Required meaning is :"<<temp->meaning;

# }

# void display()

# {

# int j=0;

# while(j<10)

# {

# temp=arr[j];

# if(temp==NULL)

# {

# j++;

# continue;

# }

# else

# {

# while(temp!=NULL)

# {

# cout<<temp->word<<"\t";

# cout<<temp->meaning<<endl;

# temp=temp->next;

# }

# }

# j++;

# }

# }

# };

# int main()

# {

# hashing s;

# s.input();

# s.display();

# s.deletefunc();

# s.search();

# return 0;

# }

# **6 Input And Output:**

Enter word :a

Enter meaning :apple

Continue?y

Enter word :b

Enter meaning :ball

Continue?y

Enter word :c

Enter meaning :cat

Continue?n

a apple

b ball

c cat

Enter the word to be deleted :a

b ball

c cat

Enter the word to find it's meaning :c

Required meaning is :cat

**7 Conclusion:**

By performing the above assignment, we learned the concepts of implementation of hash tables, the collisions and used a collision resolution method called seperate chaining.

**8 Applications:**

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, [distinct elements](https://www.geeksforgeeks.org/print-distinct-elements-given-integer-array/), counting frequencies of items, finding duplicates, etc.

There are many other applications of hashing, including modern day cryptography hash functions. Some of these applications are listed below:

* Message Digest
* Password Verification
* Data Structures(Programming Languages)
* Compiler Operation
* Rabin-Karp Algortithm
* Linking File name and path together