LAB NO 09: HASHING

Overview

- Introduction
- Advantages
- Hash Function
- Hash Table
- Collision Resolution Techniques
- Separate Chaining
- Linear Chaining
- Quadratic Probing
- Double Hashing
- Application
- Reference

Introduction

- Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects.
- Some examples of how hashing is used in our lives include:
- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.
- In both these examples the students and books were hashed to a unique number.

Introduction

Assume that you have an object and you want to assign a key to it to make searching easy.

To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use hashing.

Introduction

- In hashing, large keys are converted into small keys by using hash functions.
- The values are then stored in a data structure called hash table.
- The idea of hashing is to distribute entries (key/value pairs) uniformly across an array.
- Each element is assigned a key (converted key).
- By using that key you can access the element in O(1) time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted.

Introduction

- Hashing is implemented in two steps:
- An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.
- The element is stored in the hash table where it can be quickly retrieved using hashed key.
- hash = hashfunc(key)

index = hash % array_size

Advantages

- The main advantage of hash tables over other table data structures is speed.
- This advantage is more apparent when the number of entries is large (thousands or more).
- Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

Hash function

• A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a fixed size, which falls into the hash table.

• The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

Hash function

- To achieve a good hashing mechanism, It is important to have a good hash function with the following basic requirements:
- Easy to compute
- Uniform distribution
- Less Collision

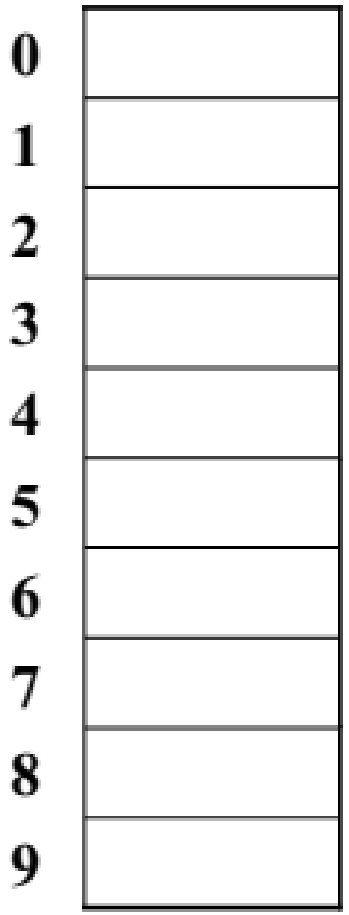
Hash Table

- A hash table is a data structure that is used to store keys/value pairs. It uses a hash function to compute an index into an array in which an element will be inserted or searched.

- By using a good hash function, hashing can work well. Under reasonable assumptions, the average time required to search for an element in a hash table is O(1).

Hash Table

- key space = integers
- TableSize = 10
- h(K) = K mod 10
- Insert: 7, 18, 41, 94



Hash Table

Data Item	Value % No. of Slots	Hash Value
26	26 % 10 = 6	6
70	70 % 10 = 0	0
18	18 % 10 = 8	8
31	31 % 10 = 1	1
54	54 % 10 = 4	4
93	93 % 10 = 3	3

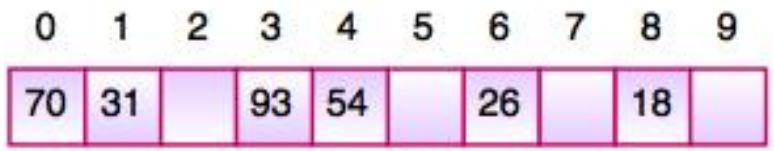


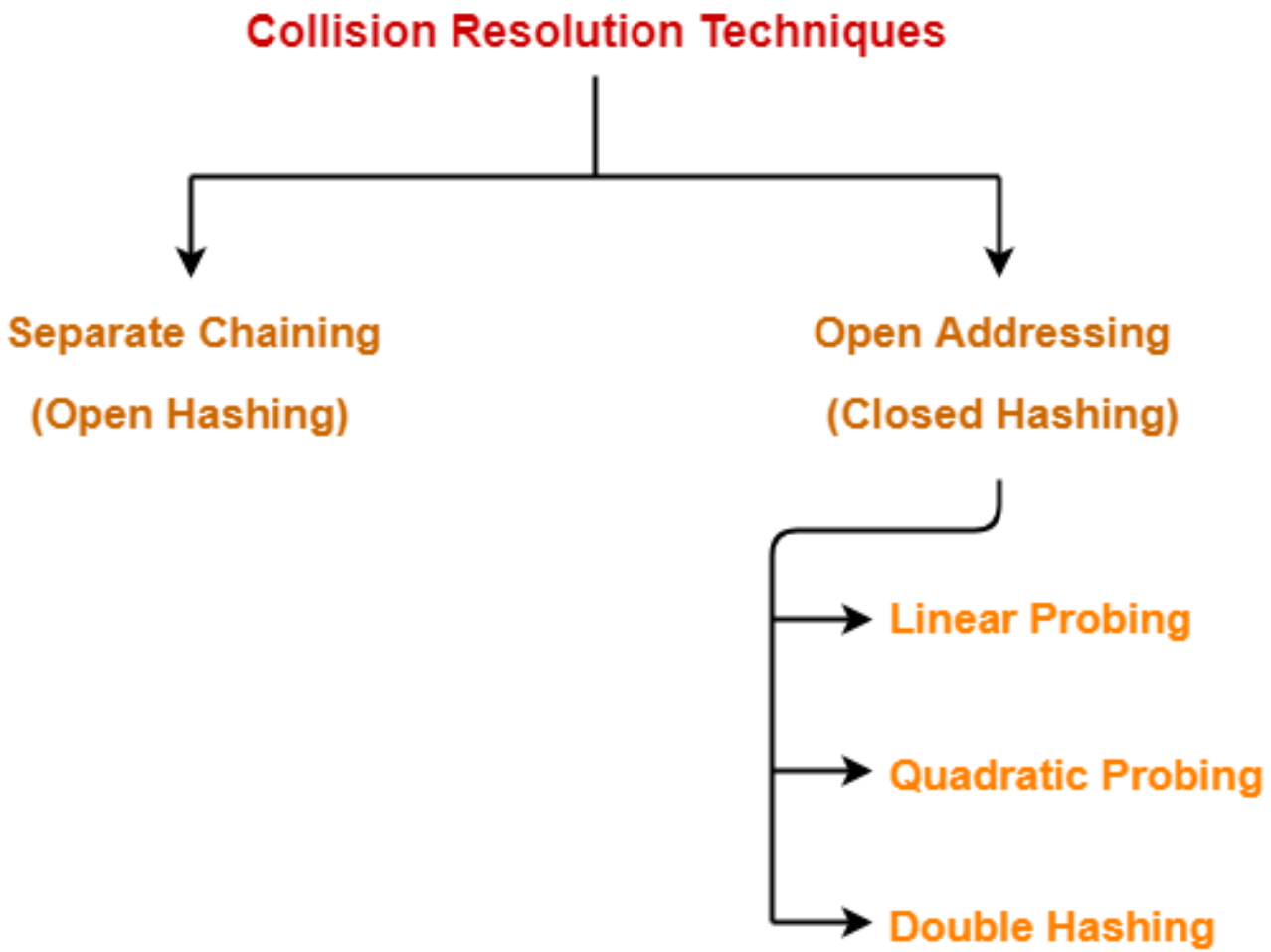
Fig. Hash Table

Collision resolution techniques

• If x_1 and x_2 are two different keys, it is possible that $h(x_1) = h(x_2)$. This is called a collision. Collision resolution is the most important issue in hash table implementations.

• Choosing a hash function that minimizes the number of collisions and also hashes uniformly is another critical issue.
• Separate chaining (open hashing)
• Linear probing (open addressing or closed hashing)
• Quadratic Probing
• Double hashing

Collision resolution techniques



Separate Chaining (Open Hashing)

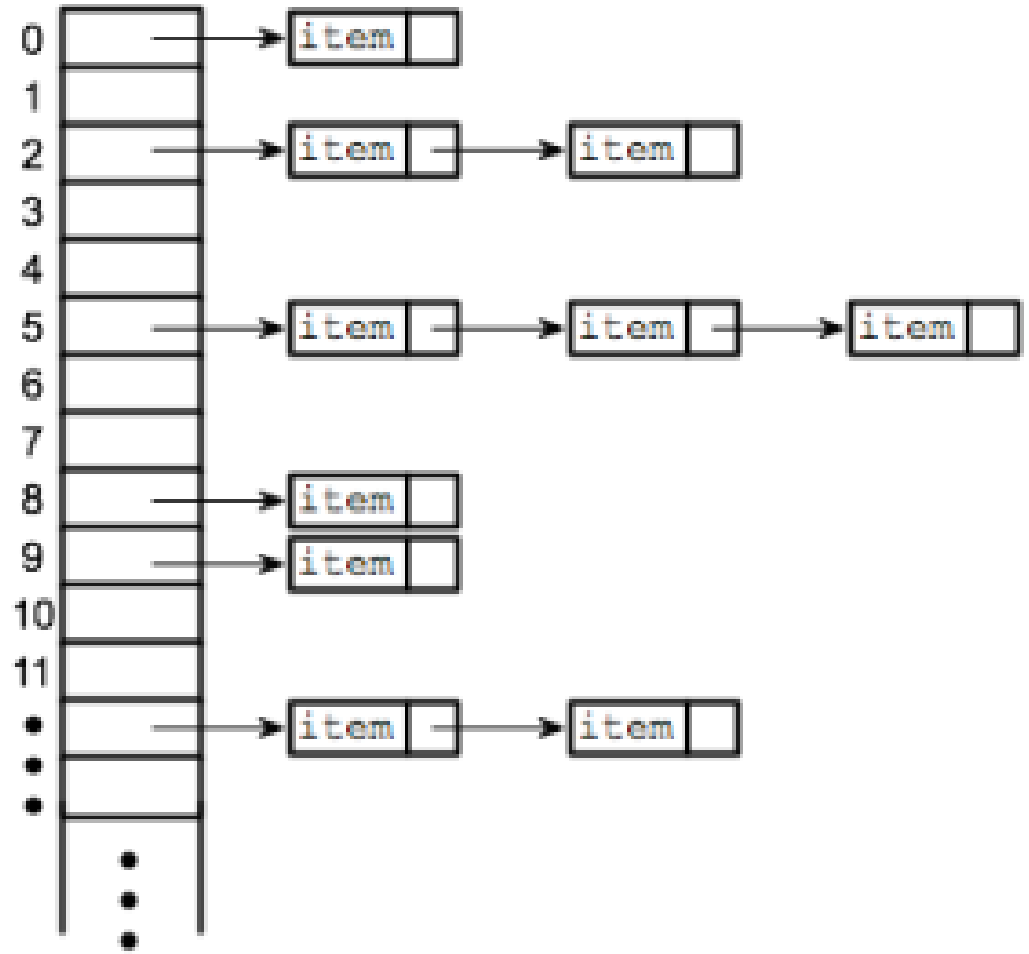
- Separate chaining is one of the most commonly used collision resolution techniques.

- It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list.

- To store an element in the hash table you must insert it into a specific linked list.

- If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

Separate Chaining (Open Hashing)



Linear Probing

- In open addressing, instead of in linked lists, all entry records are stored in the array itself.

- When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index).

- If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

Linear Probing

- The probe sequence is the sequence that is followed while traversing through entries. In different probe sequences, you can have different intervals between successive entry slots or probes.

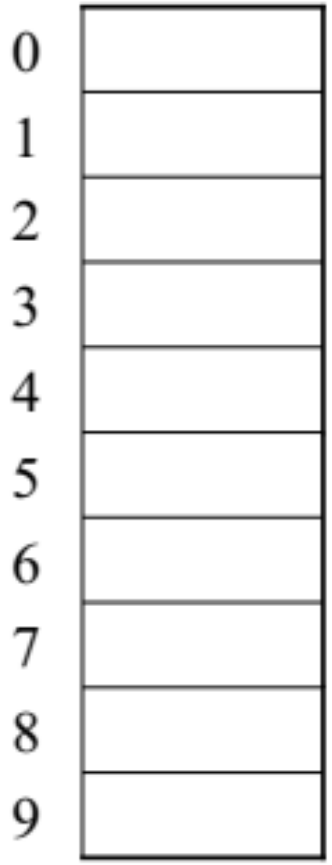
- When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

Linear Probing

- Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is index. The probing sequence for linear probing will be:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize

Linear Probing



Insert:
38
19
8
109
10

- **<u>Linear Probing</u>**:
  after checking spot h(k), try spot h(k)+1, if that is full, try h(k)+2, then h(k)+3, etc.

Quadratic Probing

- Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots.
- Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot.
- The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Quadratic Probing

- Let us assume that the hashed index for an entry is index and at index there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + 1^2) % hashTableSize
index = (index + 2^2) % hashTableSize
index = (index + 3^2) % hashTableSize

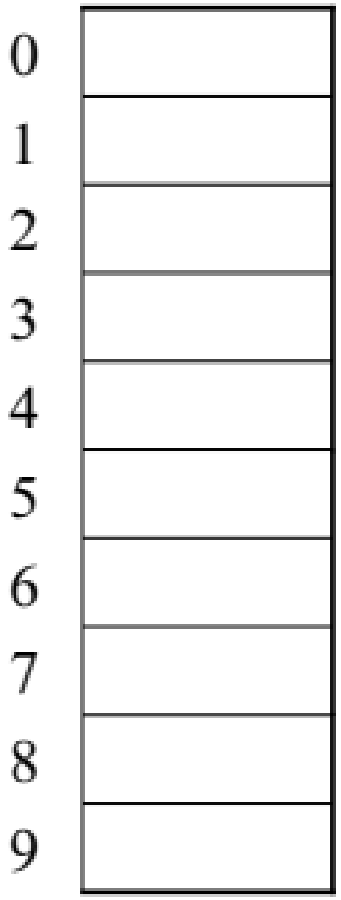
Quadratic Probing

$f(i) = i^2$

- Probe sequence:

0th probe = h(k) mod TableSize
1th probe = (h(k) + 1) mod TableSize
2th probe = (h(k) + 4) mod TableSize
3th probe = (h(k) + 9) mod TableSize
. . .
ith probe = (h(k) + i^2) mod TableSize

Quadratic Probing



Insert:
89
18
49
58
79

Double hashing

- Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

- Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

Double hashing

```
index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;
```

Double Hashing

f(i) = i * g(k)
where g is a second hash function

- Probe sequence:

0th probe = h(k) mod TableSize
1th probe = (h(k) + g(k)) mod TableSize
2th probe = (h(k) + 2*g(k)) mod TableSize
3th probe = (h(k) + 3*g(k)) mod TableSize
. . .
ith probe = (h(k) + i*g(k)) mod TableSize

h(k) = k mod 7 and g(k) = 5 – (k mod 5)

Separate Chaining Vs Open Addressing

Separate Chaining	Open Addressing
Keys are stored inside the hash table as well as outside the hash table.	All the keys are stored only inside the hash table. No key is present outside the hash table.
The number of keys to be stored in the hash table can even exceed the size of the hash table.	The number of keys to be stored in the hash table can never exceed the size of the hash table.
Deletion is easier.	Deletion is difficult.

Separate Chaining Vs Open Addressing

Extra space is required for the pointers to store the keys outside the hash table.	No extra space is required.
Cache performance is poor. This is because of linked lists which store the keys outside the hash table.	Cache performance is better. This is because here no linked lists are used.
Some buckets of the hash table are never used which leads to wastage of space.	Buckets may be used even if no key maps to those particular buckets.

Summary

Which is the Preferred Technique?

The performance of both the techniques depend on the kind of operations that are required to be performed on the keys stored in the hash table-

Summary

Separate Chaining-

Separate Chaining is advantageous when it is required to perform all the following operations on the keys stored in the hash table-

- Insertion Operation
- Deletion Operation
- Searching Operation

NOTE-

- Deletion is easier in separate chaining.
- This is because deleting a key from the hash table does not affect the other keys stored in the hash table.

Summary

Open Addressing-

Open addressing is advantageous when it is required to perform only the following operations on the keys stored in the hash table-

- Insertion Operation
- Searching Operation

NOTE-

- Deletion is difficult in open addressing.
- This is because deleting a key from the hash table requires some extra efforts.
- After deleting a key, certain keys have to be rearranged.

C++ Program to Implement Hash

```
#include <iostream>
#include<list>
using namespace std;
class hashclass
{
   int bucket; //no of buckets you want
   list<int>*hashtable;//container
   public:
   hashclass(int a)//constructor
    {
      bucket=a;
      hashtable=new list<int>[bucket];
    }
```

C++ Program to Implement Hash

```
void insert_element(int key)//function used to insert elements to the hashtable
   {
      //to get the hash index of key
      int indexkey=hashFunction(key);
      hashtable[indexkey].push_back(key);

   }
```

C++ Program to Implement Hash

```
void delete_element(int key)//function used to delete elements from the hashtable
  {
     int indexkey=hashFunction(key);
     list<int>::iterator i=hashtable[indexkey].begin();
     for(;i!=hashtable[indexkey].end();i++)
     {
        if(*i==key)
           break;     }

if(i!=hashtable[indexkey].end())
     {

        hashtable[indexkey].erase(i);       }
```

C++ Program to Implement Hash

```
 int hashFunction(int a)//function used to map
values to key
   {
      return (a%bucket);

   }
```

C++ Program to Implement Hash

```
void display_table()//used to display hashtable values
   {

      for(int i=0;i<bucket;i++)
      {
         cout<<i;
         list<int>::iterator j=hashtable[i].begin();
      for(;j!=hashtable[i].end();j++)
         {
            cout<<"---->"<<*j;
         }
         cout<<endl;
      }
```

C++ Program to Implement Hash

```
 void search_element(int key)//used to search
element
   {
      int a=0;//will be 1 if element exist
otherwise 0
      int indexkey=hashFunction(key);
            list<int>::iterator
i=hashtable[indexkey].begin();
      for(;i!=hashtable[indexkey].end();i++)
      {
         if(*i==key)
         {
            a=1;
            break;          }}
if(a==1)
         {
            cout<<"The element you wanted to
search is present in the hashtable."<<endl;
         }
         else
         {
            cout<<"Element not present"<<endl;
         }

   }
```

C++ Program to Implement Hash

```
~hashclass()//destructor
    {
        delete[]hashtable;

    } };
```

C++ Program to Implement Hash

```
int main()
{
int bucketn,ch,element;

cout<<"enter the no of buckets"<<endl;
cin>>bucketn;
hashclass hashelement(bucketn);
```

C++ Program to Implement Hash

```
while(1)
{
    cout<<"1.insert element to the hashtable"<<endl;
    cout<<"2.search element from the hashtable"<<endl;
    cout<<"3.delete element from the hashtable"<<endl;
    cout<<"4.display elements of the hashtable"<<endl;
    cout<<"5.exit"<<endl;
    cout<<"enter your choice"<<endl;
    cin>>ch;
```

C++ Program to Implement Hash

```
switch(ch)
  {
     case 1:cout<<"enter the element"<<endl;
     cin>>element;
     hashelement.insert_element(element);
     break;
     case 2:cout<<"enter the element you want to search"<<endl;
     cin>>element;
     hashelement.search_element(element);
     break;
     case 3:cout<<"enter the element to be deleted"<<endl;
     cin>>element;
     hashelement.delete_element(element);
     break;
```

C++ Program to Implement Hash

```
 case 4:hashelement.display_table();
        break;
        case 5:return 0;
        default:cout<<"enter a valid value"<<endl;
}

}
  return 0;
}
```

OUTPUT:

```
enter the no of buckets
4
1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
enter your choice
1
enter the element
13
1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
enter your choice
1
enter the element
16
```

OUTPUT:

1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
enter your choice
1
enter the element
20

1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
enter your choice
4
0---->16---->20
1---->13
2
3

OUTPUT:

1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
enter your choice
2
enter the element you want to search
20

The element you wanted to search is present in the hashtable.

OUTPUT:

1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
enter your choice
3
enter the element to be deleted
20

OUTPUT:

```
1.insert element to the hashtable
2.search element from the hashtable
3.delete element from the hashtable
4.display elements of the hashtable
5.exit
enter your choice
4
0---->16
1---->13
2
3
```