



HASHING

HASHING

The implementation of hash tables is frequently called hashing. Hashing is a technique used for performing insertions, deletions, and finds in constant average time.

Types of Hashing

There are two types of hashing namely:

1. Static hashing
2. Dynamic hashing

Static Hashing

In static hashing, the hash function maps search key values to a fixed set of locations.

Dynamic Hashing

In dynamic hashing, the hash table can grow to handle more items. The associated hash the function must change as the table grows.

HASH TABLE

A hash table is a data structure, which is implemented by a hash function and used for searching elements in quick time. In a hash table, hash keys act as the addresses of the elements.

Applications of Hash Tables

The applications of hash table are:

1. Compilers
2. Graph theory problem
3. Online spelling checkers etc.
4. Database systems
5. Symbol tables
6. Data dictionaries
7. Network processing algorithms
8. Browse caches

HASH FUNCTION

Hashing finds the location of an element in a data structure **without making any comparisons**.

In contrast to the other comparison-based searching techniques, like linear and binary search, hashing uses a **mathematical function** to determine the location of an element.

This **mathematical function** called **hash function** accepts a value, known as **key**, as input and generates an output known as **hash key**.

The hash function generates hash keys and stores elements corresponding to each hash key in the **hash table**.

A file containing information for five employees of a company. Each record in that file contains the name and a three-digit numeric Employee ID of the employee.

The hash function will implement a hash table of five slots using Employee IDs as the keys.

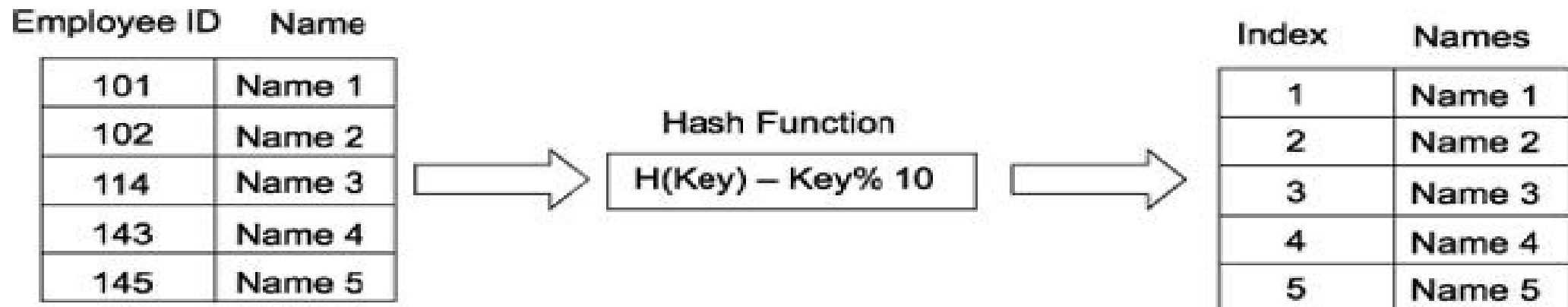


Fig. Generating hash keys

In the hash table generated in the above example, the hash function is $\text{Employee ID} \% 10$.

Employee ID 101, hash key will be calculated as 1. Therefore, Name1 will be stored at position 1 in the hash table and so on for Name 2, Name 3 and Name 4 will be stored at position 4, 3, and 5 respectively.

Whenever an employee record is searched using the Employee ID, the hash function will indicate the exact position of that record in the hash table.

HASH FUNCTION

A good hash functions should:

- Minimize collisions
- Be easy and quick to compute
- Distribute key values evenly in the hash table
- Use all the information provided in the key

METHODS OF HASHING FUNCTIONS

1. Mid square method
2. Modulo division or division remainder method
3. Folding method
4. Pseudo random number generator method
5. Digit or character extraction method
6. Radix transformation method

MID SQUARE METHOD

In this method, the key is squared and the middle part of the result based on the number of digits required for addressing is taken as the hash value.

This method works well if the keys do not contain a lot of leading and trailing zeros.

$H(X)$ = return middle digits of X^2

For example:

Map the key 2453 into a hash table of size 1000, there, $X = 2453$.

$$X^2 = 6017209$$

Extract middle value 172 as the hash value.

MODULO DIVISION OR DIVISION REMAINDER METHOD

This method computes the hash value from the key using the modulo (%) operator.

Here, the table size that is the power of 2 like 32, 64 and 1024 should be avoided as it leads to more collisions. It is always better to select table size not close to the power of 2.

$H(\text{Key}) = \text{return Key \% TableSize}$

For example:

Map the key 4 into a hash table of size 4.

$$H(4) = 4 \% 4 = 0$$

FOLDING METHOD

This method involves **splitting keys into two or more parts** each of which has the **same length** as the required address with the possible exception of the last part and then **adding the parts to form the hash address**.

There are two types of folding methods:

1. Fold shifting method
2. Fold boundary method

FOLD SHIFTING METHOD

Key is broken into several parts of same length of the required address and then added to get the hash value. Final carry is ignored.

For example:

Map the key 123203241 to a range between 0 to 999.

Let $X = 1\ 2\ 3\ 2\ 0\ 3\ 2\ 4\ 1$

Position X into 123, 203, 241, then add these three values.

$123 + 203 + 241 = 567$ to get the hash value.

FOLD BOUNDARY METHOD

Key is broken into several parts and reverse the digits in the outermost partitions and then add the partition to form the hash value.

For example:

$X = 123203241$

Partition = 123, 203, 241

Reverse the boundary partition = 321, 203, 142

Add the partition = 321 + 203 + 142

Hash value = 666.

PSEUDO RANDOM NUMBER GENERATOR METHOD

This method generates a random number given a seed as a parameter and the resulting random number is then scaled into the possible address range using modulo division.

It must ensure that it always generates the same random value for a given key. The random number produced can be transformed to produce a valid hash value.

$$X_{i+1} = A X_i \% \text{TableSize}$$

PSEUDO RANDOM NUMBER GENERATOR (PRNG)

Pseudo Random Number Generator(PRNG) refers to an algorithm that uses mathematical formulas to produce sequences of random numbers.

PRNGs generate a sequence of numbers approximating the properties of random numbers. A PRNG starts from an arbitrary starting state using a seed state.

Many numbers are generated in a short time and can also be reproduced later if the starting point in the sequence is known.

Hence, the numbers are **deterministic** and **efficient**.

PSEUDO RANDOM NUMBER GENERATOR (PRNG)

Linear Congruential Generator is the most common and oldest algorithm for generating pseudo-randomized numbers. The generator is defined by the recurrence relation:


$X_{n+1} = (aX_n + c) \bmod m$ where X is the sequence of pseudo-random values

$m, 0 < m$ - modulus

$a, 0 < a < m$ - multiplier

$c, 0 \leq c < m$ - increment

$x_0, 0 \leq x_0 < m$ - the seed or start value



```
#include<stdio.h>
```

```
#include<stdlib.h>
```

```
#include<time.h>
```

```
int main()
```

```
{
```

```
    srand(time(NULL));
```

```
    int i;
```

```
    for(i = 0; i<5; i++)
```

```
        printf("%d\t", rand()%10);
```

```
}
```


DIGIT OR CHARACTER EXTRACTION METHOD

This method **extracts the selected digits** from the **key** and used it as the address or it can be reversed to give the hash value.

For example:

Map the key **123203241** to a range between 0 to 9999.

Select the digits from the positions: **2, 4, 5, and 8**.

123203241

Therefore the hash value = **2204**.

Similarly, the hash value can also be obtained by considering the above-digit positions and reversing it, which yields the hash value of **4022**.

RADIX TRANSFORMATION METHOD

Transform the number into another base and then divide by the maximum address.

Example: Addresses from 0 to 99,

key = 453 in base 11 = 382.

Hash address = $382 \bmod 99 = 85$.

PROBLEMS WITH HASHING

Collision: No matter what the hash function, there is the possibility that two different keys could resolve to the same hash address. This situation is known as a collision.

Handling the Collisions: The following techniques can be used to handle the collisions.

- Chaining
- Double hashing (Re-hashing)
- Open Addressing (Linear probing, Quadratic probing, Random probing), etc.

COLLISION

The hash function takes some key values as input, performs some mathematical calculation, and generates hash key to ascertain the position in the hash table where the record corresponding to the key will be stored.

However, it is quite possible that the hash function generates **same hash keys** for **two different key values**. That means, two different records are indicated to be stored at the same position in the hash table. This situation is termed as **collision**.

VARIOUS TECHNIQUES OF HASHING

The various techniques of hashing are:

- Separate chaining
- Open addressing
 - Linear probing
 - Quadratic probing
 - Double hashing
- Rehashing
- Extendible hashing

SEPARATE CHAINING

The separate chaining technique uses linked lists to overcome the problem of collisions.

In this technique, all the keys that resolve to the same hash values are maintained in a linked list. Whenever a collision occurs, the key value is added at the end of the corresponding list.

Thus, each position in the hash table acts as a header node for a linked list. To perform a search operation, the list indicated by the hash function is searched sequentially.

Insert 0, 1, 81, 4, 64, 25, 16, 36, 9, 49.

INSERTION

To perform an insert, we traverse down the appropriate list to check whether the element is already in place (if duplicates are expected, an extra field is usually kept, and this field would be incremented in the event of a match).

If the element turns out to be new, it is inserted either at the front of the list or at the end of the list, whichever is easiest.

$$H(\text{Key}) = \text{Key} \% \text{TableSize}$$

Insert 0 $H(0) = 0 \% 10 = 0$

Insert 1 $H(1) = 1 \% 10 = 1$

Insert 81 $H(81) = 81 \% 10 = 1$

The **element 81 collides** to the same hash value 1. To place the value at this position, perform the following:

- Traverse the list to check whether it is already present.
- Since it is not already present, insert at the end of the list.

Similarly, the rest of the elements are inserted.

$$\text{Insert 4 } H(4) = 4 \% 10 = 4$$

$$\text{Insert 64 } H(64) = 64 \% 10 = 4$$

$$\text{Insert 25 } H(25) = 25 \% 10 = 5$$

$$\text{Insert 16 } H(16) = 16 \% 10 = 6$$

$$\text{Insert 36 } H(36) = 36 \% 10 = 6$$

$$\text{Insert 9 } H(9) = 9 \% 10 = 9$$

$$\text{Insert 49 } H(49) = 49 \% 10 = 9$$

ROUTINE TO PERFORM INSERTION

```
void Insert(int Key, HashTable H)
{
    Position Pos, NewCell;
    List L;
    Pos = Find(Key, H);
    if(Pos == NULL)
    {
        NewCell = malloc(sizeof(struct ListNode));
        if(NewCell == NULL)
            printf("Out of space!!!");
        else
        {
            L = H->TheLists[Hash(Key, H->TableSize)];
            NewCell->Next = L->Next;
            NewCell->Element = Key;
            L->Next = NewCell;
        }
    }
}
```

FIND

To perform a find, we use the hash function to determine which list to traverse.

We then traverse this list in the normal manner, returning the position where the item is found.

ADVANTAGE

- More elements can be inserted as it uses array of linked lists.
- Collision resolution is simple and efficient

DISADVANTAGE

- It requires pointers, which occupies more memory space.
- It takes more effort to perform a search since it takes time to evaluate the hash function and also to traverse the list.

TITLE LOREM IPSUM DOLOR



LOREM IPSUM DOLOR SIT AMET,
CONSECTETUER ADIPISCING
ELIT.



NUNC VIVERRA IMPERDIET ENIM.
FUSCE EST. VIVAMUS A TELLUS.



PELLENTESQUE HABITANT
MORBI TRISTIQUE SENECTUS ET
NETUS.