**What is hashing**

* Hashing means to chop and mix.
* A bucket in a hash file is a unit of storage (typically a disk block) that can hold one or more records.
* The hash function, h, is a function from the set of all search-keys, K, to the set of all bucket addresses, B.
* Insertion, deletion, and lookup are done in constant time.
* To insert a record into the structure compute the hash value h(Ki), and place the record in the bucket address returned.
* For lookup operations, compute the hash value as above and search each record in the bucket for the specific record.
* To delete simply lookup and remove.

**Basic Hashing techniques**

* In **direct hashing**, the key is the address without any algorithmic manipulation

Store numbers 0 thru 5 in an array

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
| Value | 0 | 1 | 2 | 3 | 4 | 5 |

* In **subtraction hashing** the key is transformed to an address by subtracting a fixed number from it.

Store numbers 10-15 in an array

10-10=0

11-10=1

12-10=2

13-10=3

14-10=4

15-10=5

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 | 4 | 5 |
| Value | 10 | 11 | 12 | 13 | 14 | 15 |

* In **modulo division hashing**, the key is divided by the list size (which is recommended to be a prime number) and the reminder plus 1 is used as the address

In practice, keys tend to be correlated.

If divisor is an even number, odd integers hash into odd home buckets and even integers into even home buckets.

20%14 = 6, 30%14 = 2, 8%14 = 8

15%14 = 1, 3%14 = 3, 23%14 = 9

Divisor is an odd number, odd (even) integers may hash into any home.

20%15 = 5, 30%15 = 0, 8%15 = 8

15%15 = 0, 3%15 = 3, 23%15 = 8

Example:

Insert 7, 6, 5, 4

7%4=3

6%4=2

5%4=1

4%4=0

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 |
| value | 4 | 5 | 6 | 7 |

**Collisions**

Except in direct and subtraction methods, collisions are unavoidable. Collisions occur when a new key is hashed into an address that is already occupied. The key detail being: lots of collisions means that it takes longer to find the value! (the time it takes to find a value in a bin is proportional to the number of elements in that bin)

Example:

Insert 7, 11

7%4=3

11%4=3 where would 11 go????

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Index | 0 | 1 | 2 | 3 |
| value |  |  | 7 |  |

* The distribution should be uniform.
  + An ideal hash function should assign the same number of records in each bucket.
* The distribution should be random.
  + Regardless of the actual search-keys, the each bucket has the same number of records on average
  + Hash values should not depend on any ordering or the search-keys

**(Inside the table)** Closed hashing (open addressing). Closed hashing is preferred in database systems It is confusing because open hashing means the opposite of open addressing. (Within table) Closed hashing stores all records directly in the hash table.

**(Outside the table)** Open hashing (separate chaining)(outside the table)

Records within a slot's list can be ordered in several ways: by insertion order, by key value order, or by frequency-of-access order.

**Closed Hashing algorithms (Open addressing)**

Ideally, choose large prime number b (Size of table)

**Linear probing**: Map key to integer i. If not free, then try i+1, i+2, etc.

**Example 1:**

Table size=17

Insert the following keys : 6,12, 34, 29,28,11, 23, 7, 0 ,33 ,30 ,45

Hash function: key % 17

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 34 | 0 | 45 |  |  |  | 6 | 23 | 7 |  |  | 28 | 12 | 29 | 11 | 30 | 33 |

**Example 2:**

Table size=10

Insert the following keys: 81, 70, 97, 60, 51, 38, 89, 68, 24

Hash function: key % 10

0 1 2 3 4 5 6 7 8 9

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 70 | 81 | 60 | 51 | 68 |  |  | 97 | 38 | 89 |

60 causes collision, check bin 1, check bin 2

51 causes collision, check 1, 2, 3

68 causes collision, check 1, 2, 3, 4.

What happens when we search? Inefficient. What happens when we delete? We could rescan all the numbers and shift them again to see if things can become more efficient. We could leave it in the same place, change the 89 to like D and then probe until we get to an empty spot.

**Quadratic probe** hi(x) = (h(x) + i^2) % D

* + If there is a collision at hash address h, this method probes the table at locations h+1, h+4, h+9, ..., that is, at locations h + i^2 (mod tablesize) for i = 1, 2, .... That is, the increment function is i^2. Quadratic probing substantially reduces clustering, but it is not obvious that it will probe all locations in the table.

In **digit extraction hashing**, selected digits are extracted from the key and used as an address

Multiply by some prime number and extract

14\*117=1638 Bin 8

29\*117=3393 Bin 3

43\*117=5031 Bin 1

19\*117=2223 Bin 3 collision, jump by the next digit which is 2, (2+3)%10=Bin 5

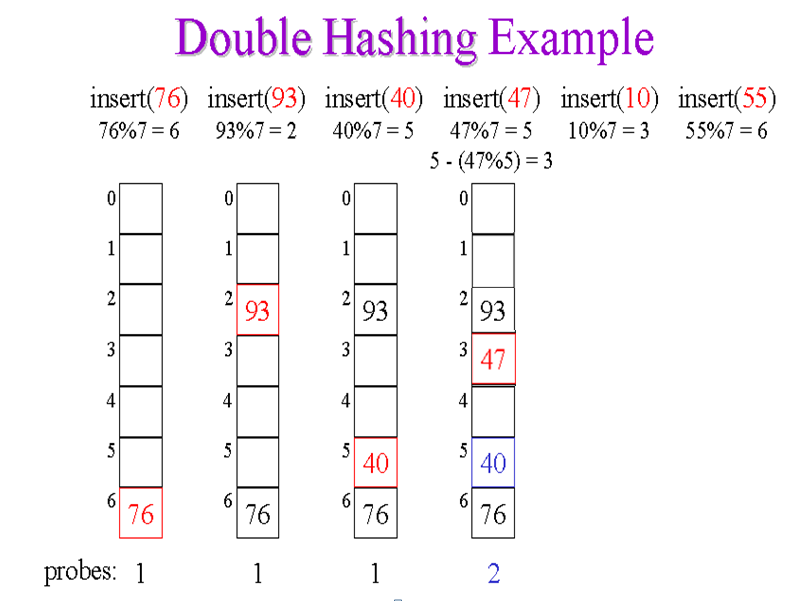
74\*117=8658 Bin 8 jump 5 (5+8) % 10 =3 taken. Oops

Jump size and table size should be relatively prime.

**Double Hashing**

Double Hashing uses nonlinear probing by computing different probe increments for different keys.

It uses two functions. The first function computes the original address, if the slot is available (or the record is found) we stop there, otherwise, we apply the second hashing function to compute the step value.



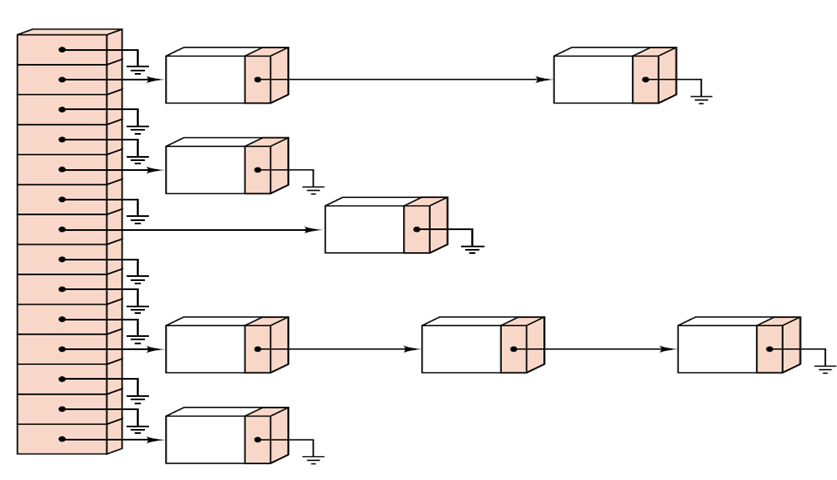
Other methods

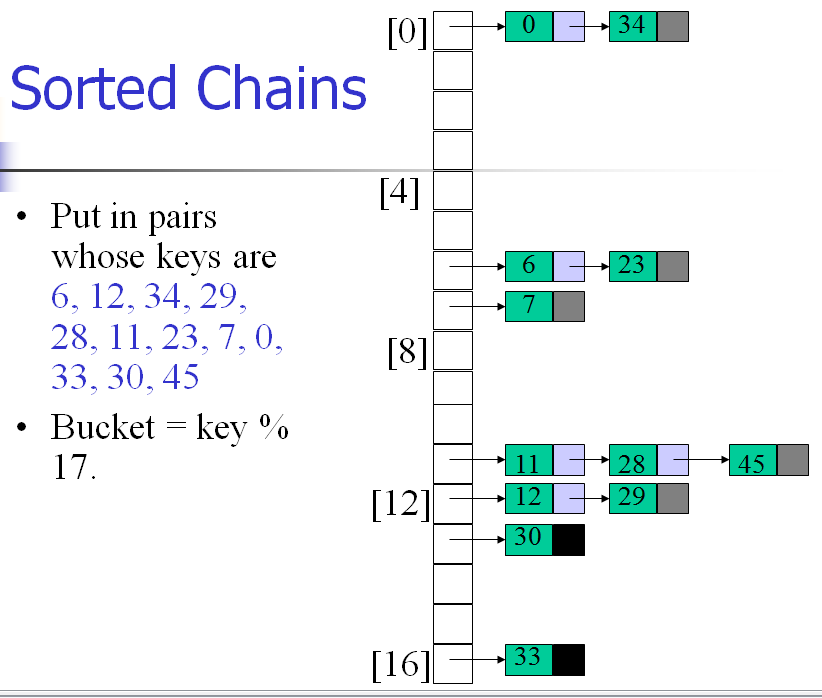
* In **mid-square hashing**, the key is squared and the address is selected from the middle of the result. 4321 multiplied by itself gives us 18671041, extract 3 digits and then we have 710
* In **fold shift hashing**, the key is divided into parts whose sizes match the size of the required address. Then the parts are added to obtain an address. In the folding method, the key is divided into two parts that are then combined or folded together to create an index into the table. This is done by first dividing the key into parts where each of parts of the key will be the same length. Like an ssn 987654321 would be 987 654 321 add together to come up with 1962 and then we use divide or extraction to get 3 digits. We could also do boundary folding with some parts being reversed. 987 456 321 we get 1764 then mod or extract. e.g. x=12320324111220; partition x into 123,203,241,112,20; then return the address 123+203+241+112+20=699
* In **fold boundary** hashing, the key is divided into parts whose size match the size of the required address. Then the left and right parts are reversed and added to the middle part to obtain the address.

**Open Hashing**

**Chaining**

One way of resolving collisions is to maintain M linked lists, one for each possible address in the hash table. A key K hashes to an address i = h(k) in the table. At address i, we find the head of a list containing all records having keys that have hashed to i. This list is then searched for a record containing key K.





**Load Factor**

We also want to keep track of the load factor, λ (the ratio of elements in the table to the size of

the table), so that we double the size if λ exceeds a threshold (typically 2/3 or so).

LoadFactor = *noOfElements* /sizeOfTable

* If **open addressing** is used, then each table slot holds at most one element, therefore, the loading factor can **never** be greater than 1.
* If **external chaining** is used, then each table slot can hold many elements, therefore, the loading factor **may be** greater than 1.