Hashing

Direct-address table >>

- Universe of keys, ***U* = { 0, 1, … … , m-1 }** where m is not too large.
- Uses a direct-address table, denoted by ***T* [0..m-1]** in which each slot corresponds to a key in the universe ***U.***
- An element with key ***k*** is stored in slot ***k***.
- Search, Insert, Delete operation takes ***O(1)*** time.
- Storage requirement: **Θ(|U|)**

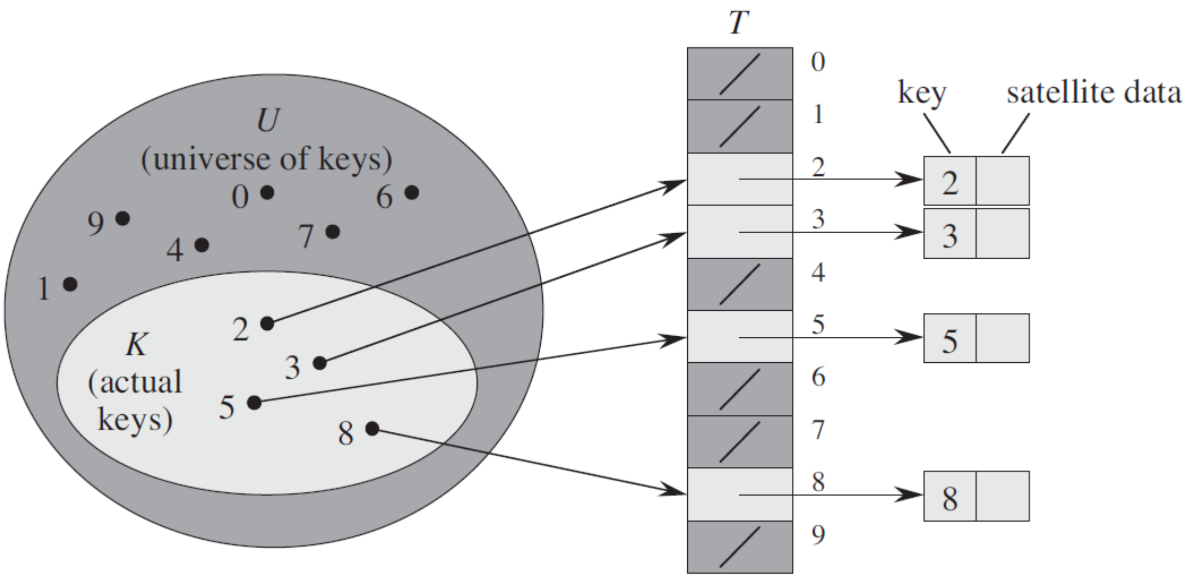


Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Problems:

- If the universe ***U*** is large, storing a table ***T*** of size **|U|** may be impractical, or even impossible, given the memory available on a typical computer.
- Furthermore, the set ***K*** of keys actually stored may be so small relative to ***U*** that most of the space allocated for ***T*** would be wasted.

Hash Table >>

- Storage requirement: **Θ(m)**
- Average search time: **O(1)**
- With hashing, an element with key ***k*** is stored in slot ***h(k),*** we use hash function ***h*** to compute the slot from the key ***k***.
- Uses a hash table ***T* [0..m-1]** where ***h*** maps ***U*** to the slot number,

h : U -> { 0, 1, … , m-1 }

- Hash function reduces the range of array indices and hence the size of the array. Instead of a size of **|U|**, the array can have size **m**.

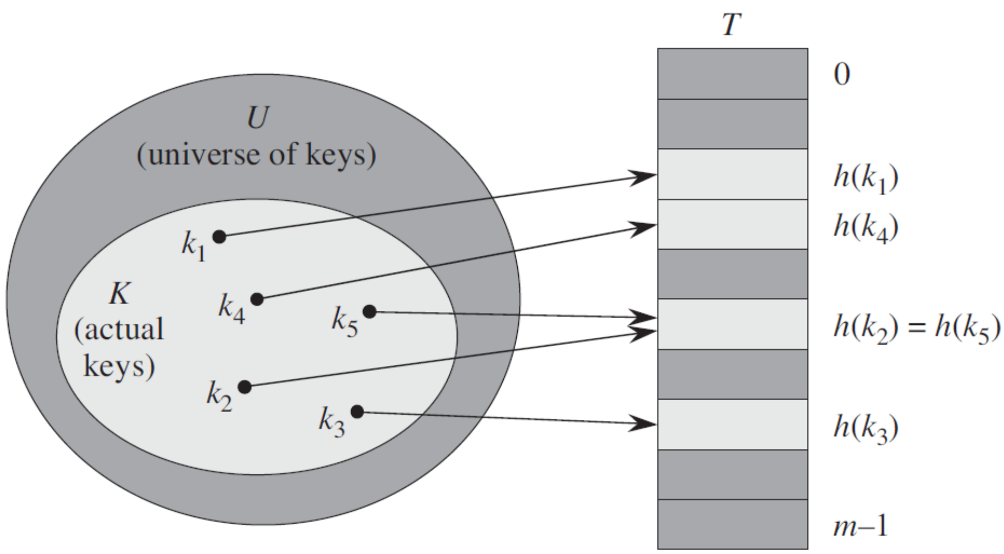


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

Good Hash Function Characteristics >>

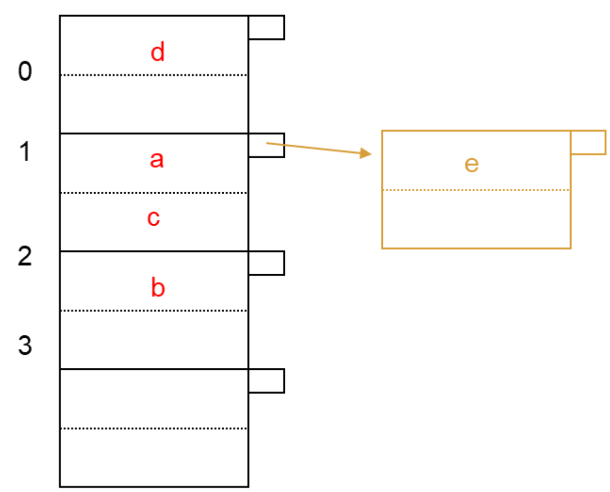
- Satisfies the assumption of simple **uniform hashing**: each key is equally likely to hash to any of the ***m*** slots, independently of where any other key has hashed to.
- Keys that are close in some sense must yield hash values that are far apart, **independent of any patterns** that might exist in the data.
- Number of **collisions should be less** while placing the data in the hash table.
- Average search time: ***O(1)***

Collision Resolution >>

1. Chaining -> Open Hashing/Separate Chaining :

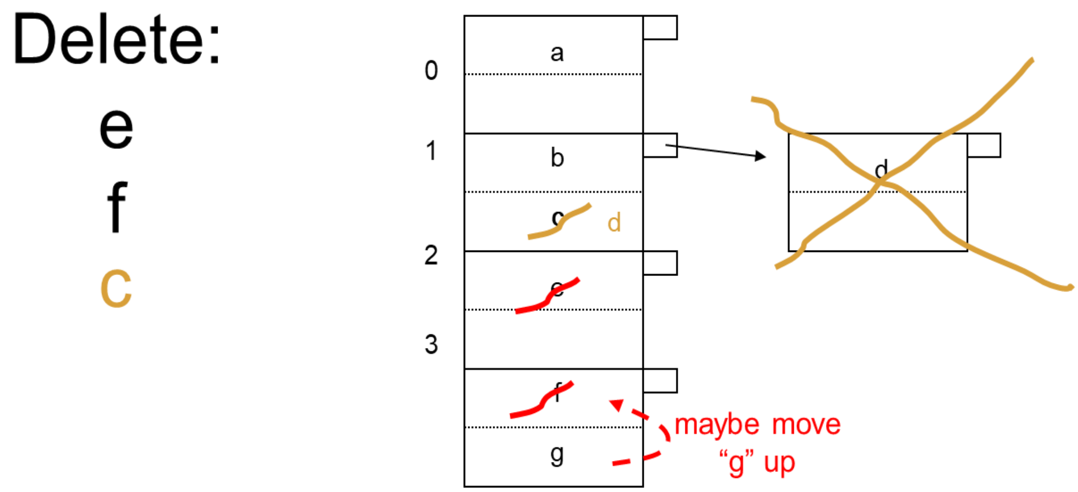
Insertion:
2 keys / bucket

INSERT:
h(a) = 1
h(b) = 2
h(c) = 1
h(d) = 0



h(e) = 1

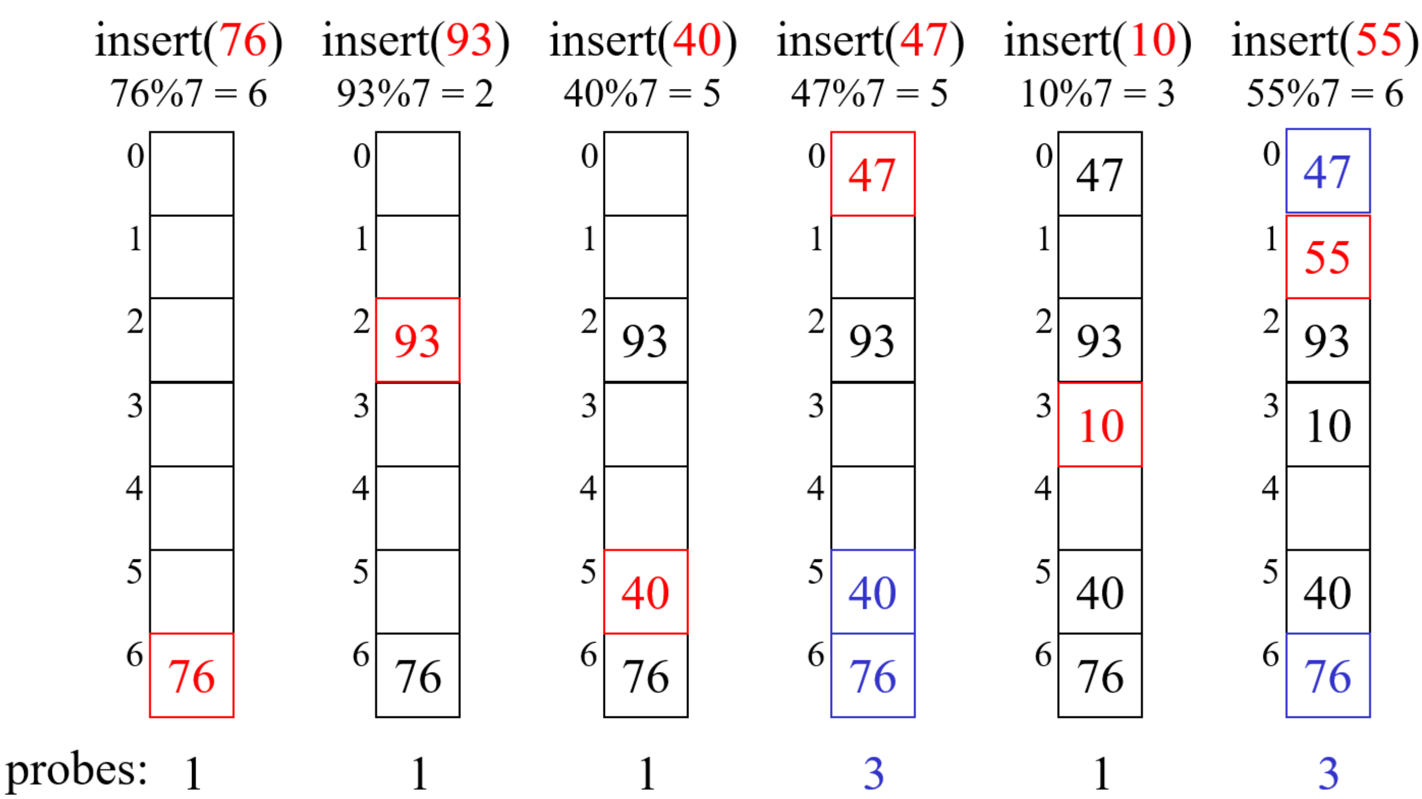
Deletion:



2. Probing -> Closed Hashing/Open Addressing:

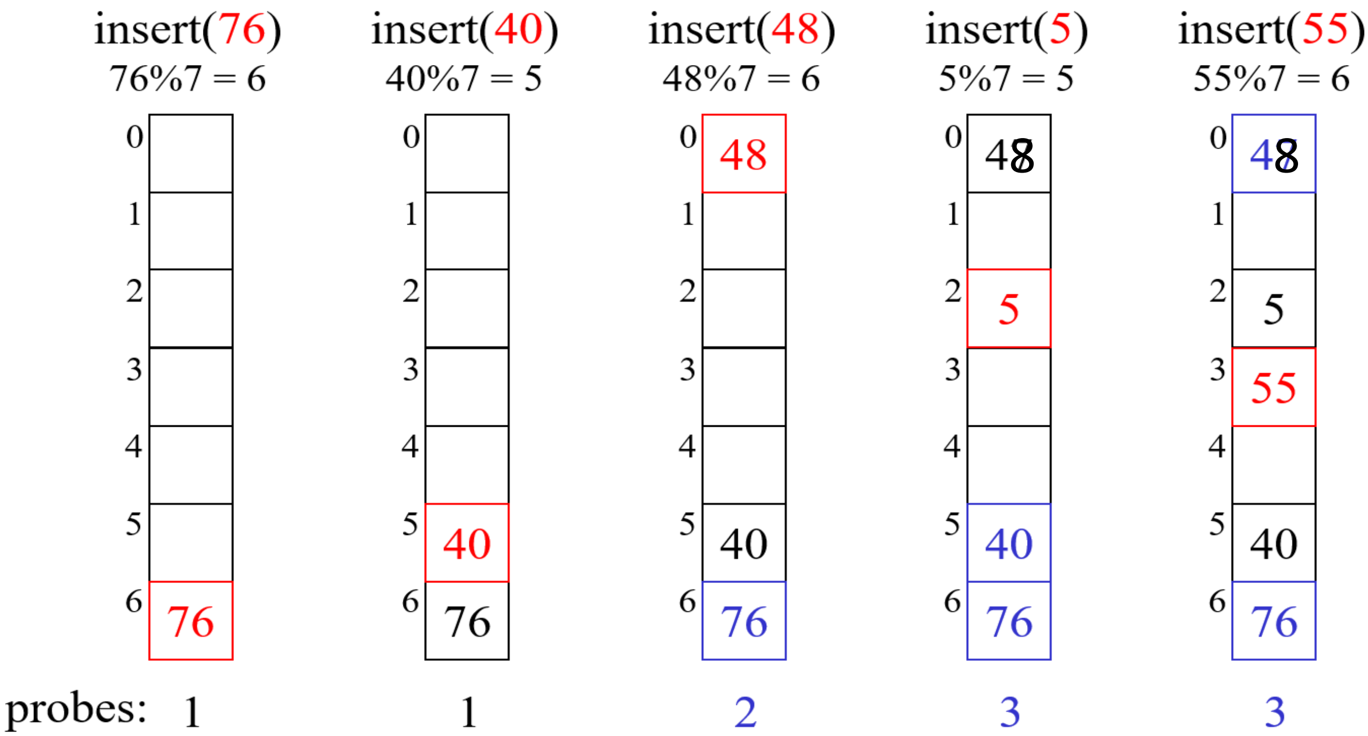
- **Linear Probing:**
  - Auxiliary hash function: h′ : U -> { 0, 1, … , m-1 }
  - Hash function: h(k, i) = (h′(k) + i) mod m; i = 0,1, … , m-1
  - First tries to insert into h′(k), if collusion found then search the next available slots for insertion.

Here, 1 key/bucket, auxiliary hash fn: h′(k) = k % 7, hash fn: h(k, i) = ((k%7) + i) % 7 ; i = 0, 1 , … , 6



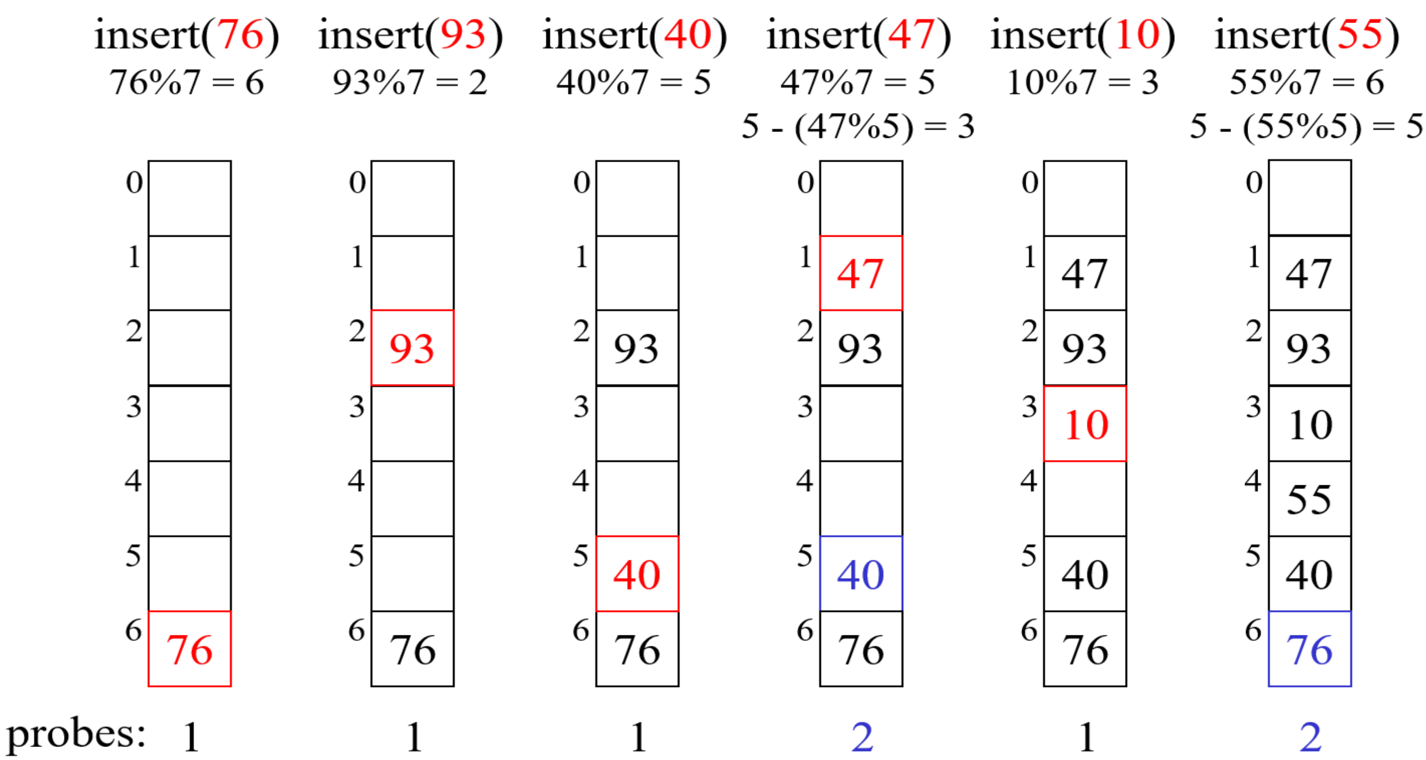
- **Quadratic Probing:**
  - Hash function: h(k, i) = (h'(k) + $c_1 * i$ + $c_2 * i^2$) mod m; i = 0, 1, … , m-1

    Here, 1 key/bucket, c_1 = 0 and c_2 = 1, auxiliary hash fn: h'(k) = k % 7, hash fn: h(k, i) = ((k%7) + i^2) % 7



- **Double Hashing:**
  - Hash function: h(k, i) = ($h_1(k)$ + $i * h_2(k)$) mod m ; i = 0, 1, … , m-1

    Let, 1 key/bucket, $h_1(k)$ = k % 7 , $h_2(k)$ = 5 – (k % 5), hash function: h(k, i) = ($h_1(K)$ + $i * h_2(k)$) % 7



Mohammad Imam Hossain, Lecturer, Dept. of CSE, UIU. Email: imambuet11@gmail.com

Static Hashing >>

The hash tables we have examined so far are called static hash tables, because the number of bucket addresses never changes that is hash table size is fixed.

Example: Chaining and Probing.

Dynamic Hashing >>

The hash table size is allowed to vary so that it can add/remove data buckets depending on the number of records.

Two types:

- Extensible Hashing
- Linear Hashing

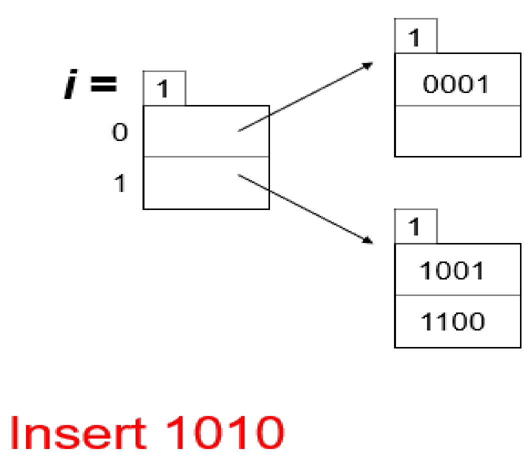
Extensible Hashing >>

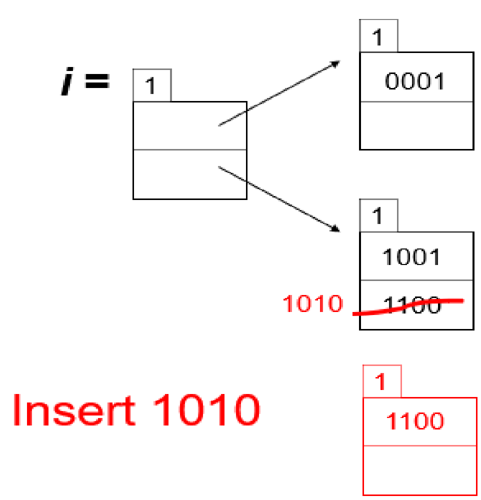
- There is an array of pointers to buckets.
- The array of pointers can grow. Its length is always a power of 2, so in a growing step the number of buckets doubles.
- There doesn't have to be a data block for each bucket. Certain buckets can share a block if the total number of records in those buckets can fit in the block.
- The hash function h computes for each key a sequence of k bits for some large k, say 32.
- The bucket numbers will at all times use some smaller number of bits, say **i** bits, from the beginning of this sequence of k bits. (i.e. $\mathbf{2}^{i}$ maximum no of entries).
- The number(**j**) appearing in the nub of each buckets, indicates how many bits of the hash function's sequence is used to determine membership of records in this block.

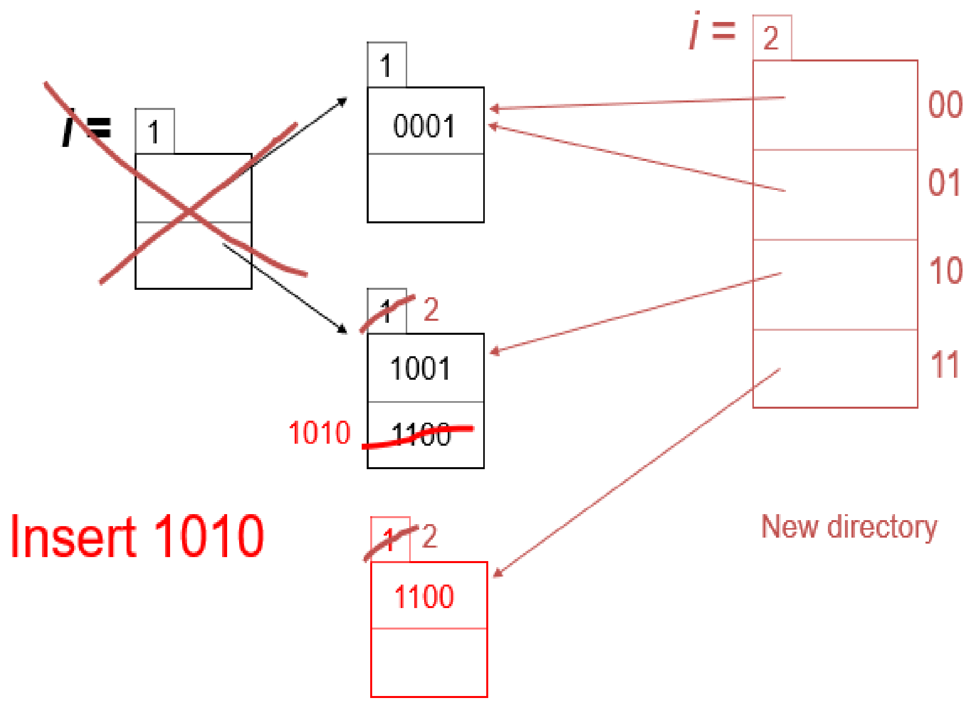
- Online Simulator Link: https://devimam.github.io/exhash/

Insertion:

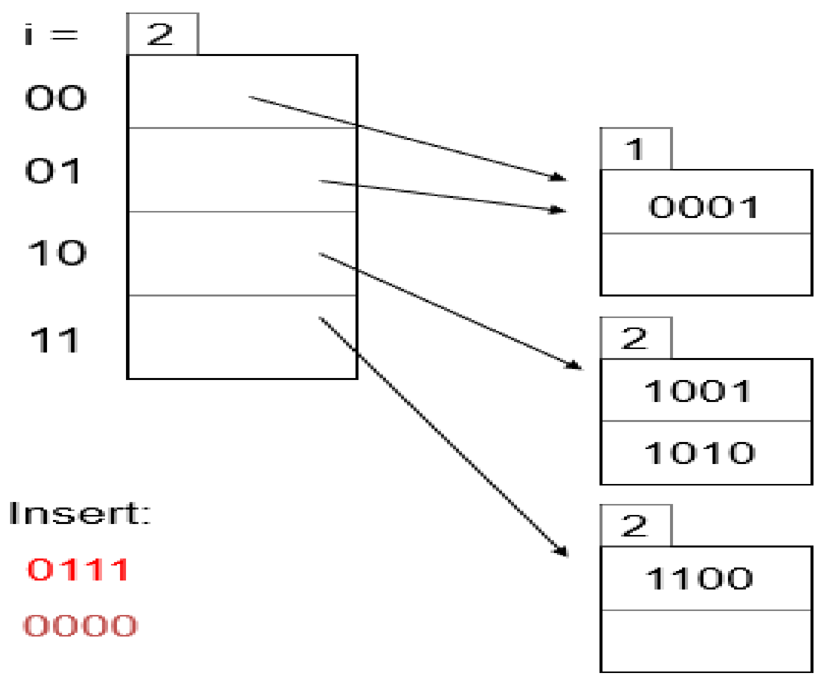
- **Inserting 1010**



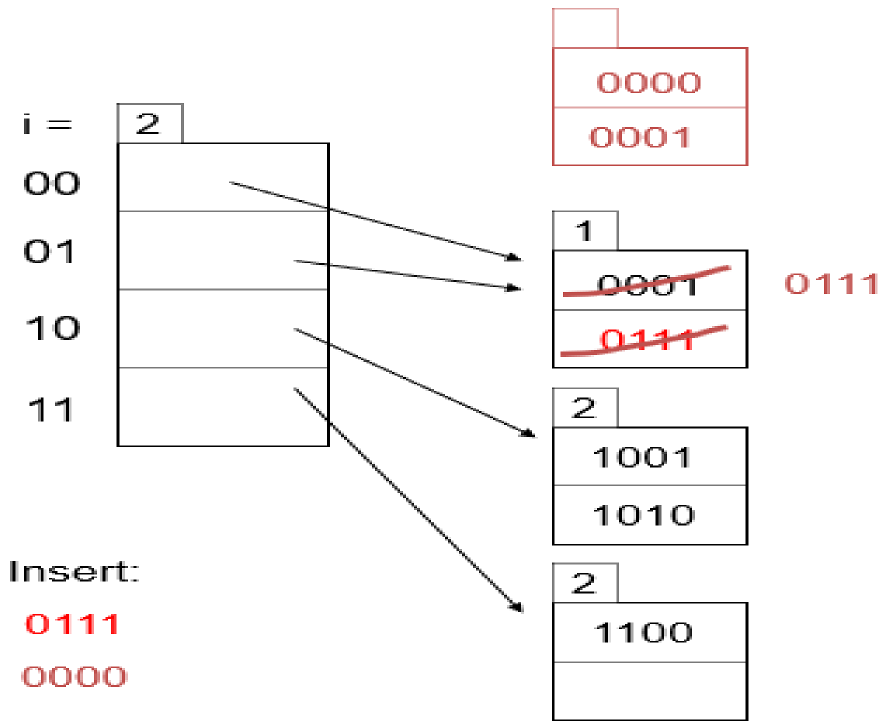


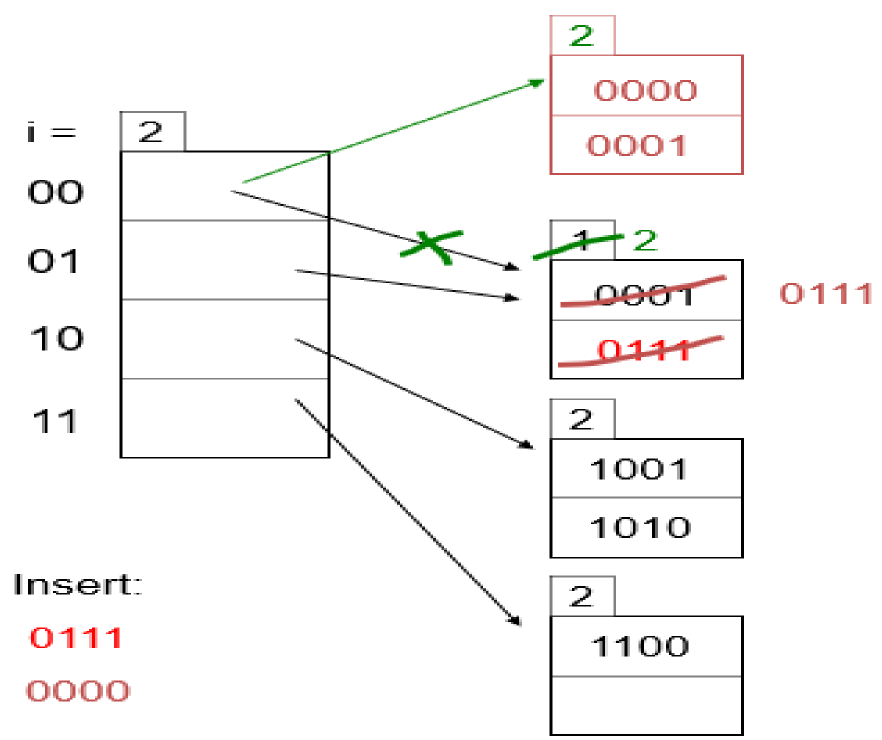


Final structure:

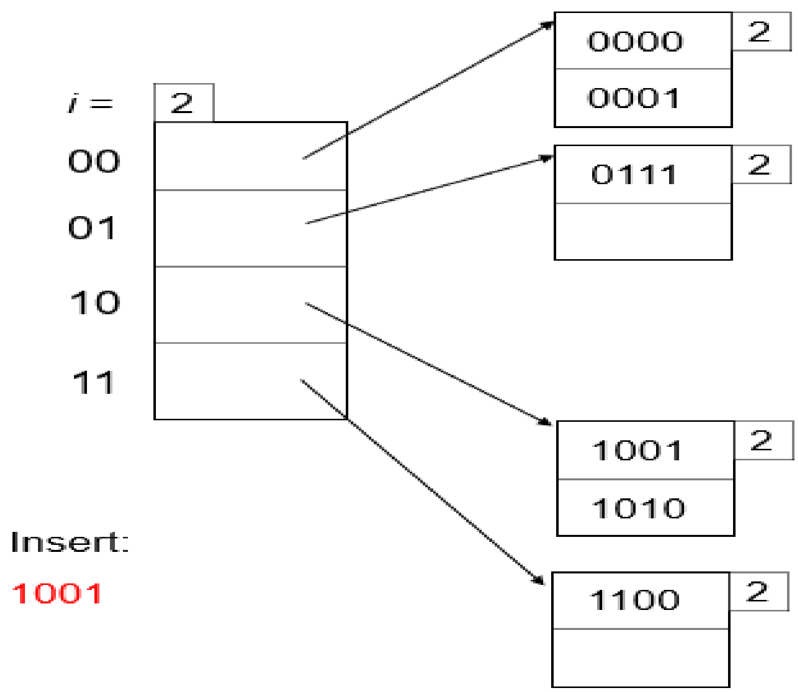


- **Inserting 0111 , 0000**

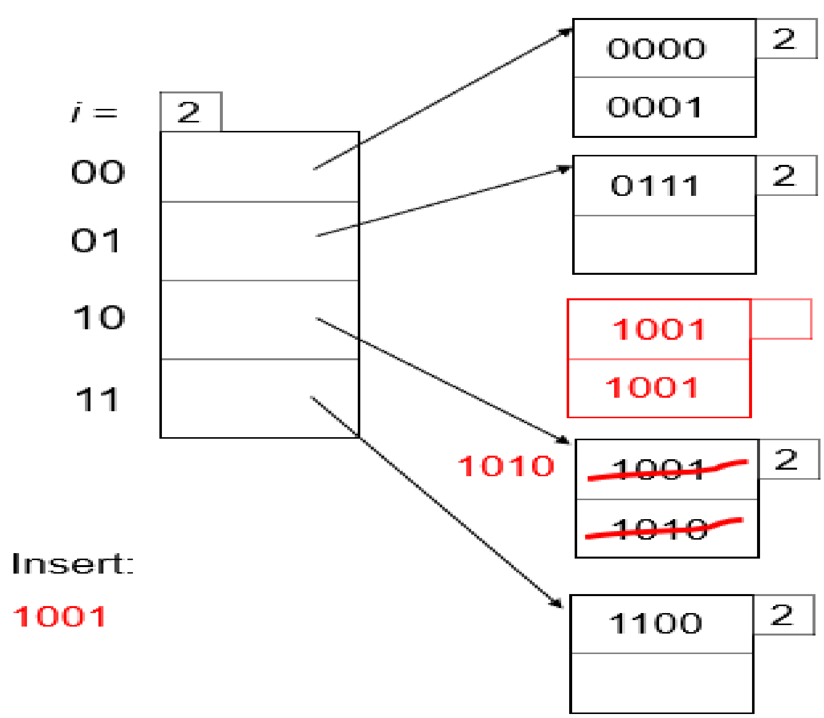




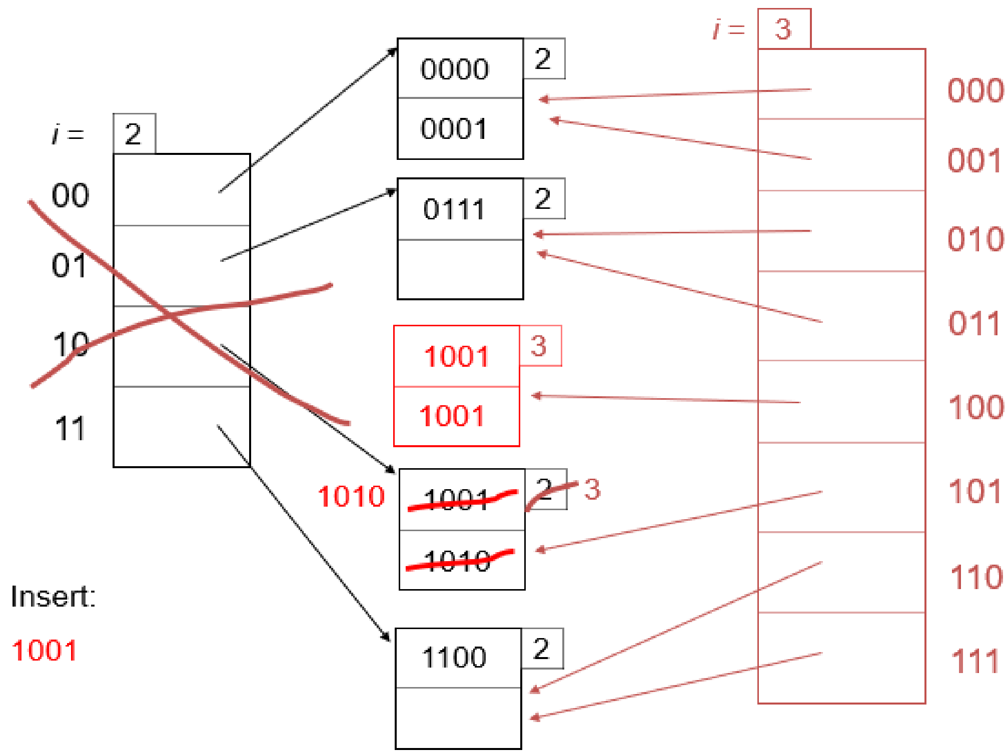
Final structure:



- **Inserting 1001**



Final structure:



B+ tree vs Hashing >>

- Hashing good for given key values, no need to access index structure

  Example:
  SELECT * FROM Sells WHERE price = 20;

- B+ Trees and conventional indexes good for range queries:

  Example:
  SELECT * FROM Sells WHERE price > 20;

Practices:

1. Suppose that we are using extendable hashing on a file that contains records with the following search key values:

   2, 3, 5, 7, 11, 17, 19, 23, 29, 31

   Show the extendable hash structure for this file if the hash function is ***h(x) = x mod 8*** and buckets can hold three records.

2. Suppose you have a file that contains records with the following searck key values:

   42, 71, 43, 70, 12, 22, 91, 62, 53, 82

   Now build an Extensible hash index structure by inserting these search keys sequentially where the hash function is h(search-key) = (reverse the digits of the search key) mod 8 and each bucket can hold at most three records.

3. Consider the Hash Function h(x) = x mod 8 and the given search keys:

   0, 7, 1, 2, 3, 11, 13, 22

   Insert the search keys one by one into an extensible hash structure using the given hash function where each bucket can contain at most 3 keys.