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|)**

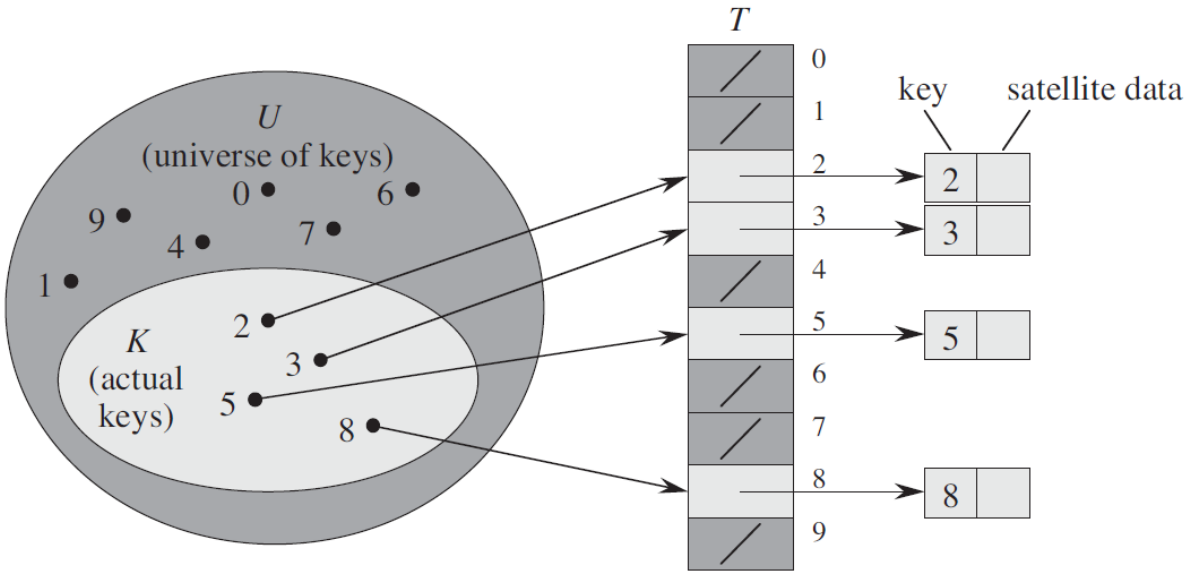


Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded,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

Problem:

- 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**.

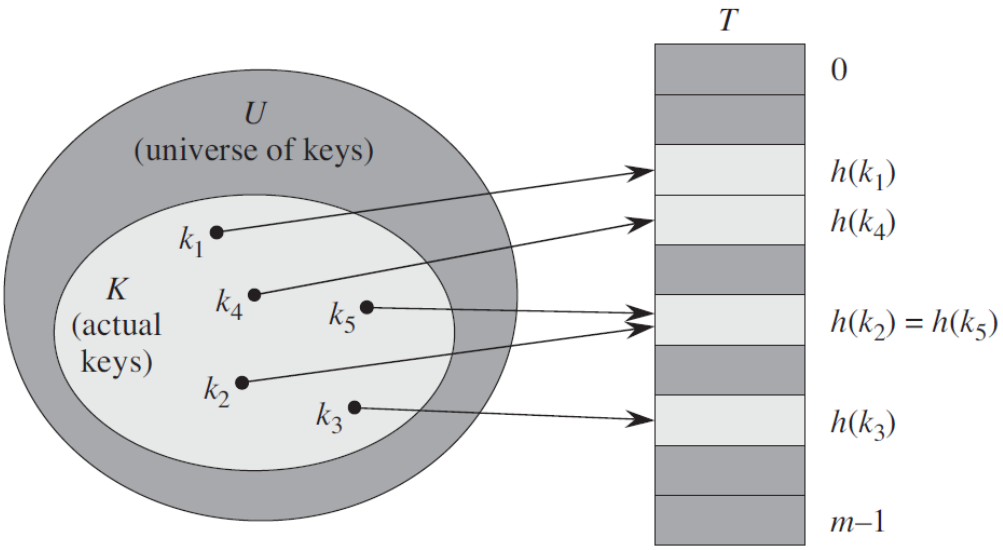


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

Characteristics of a good hash function>>

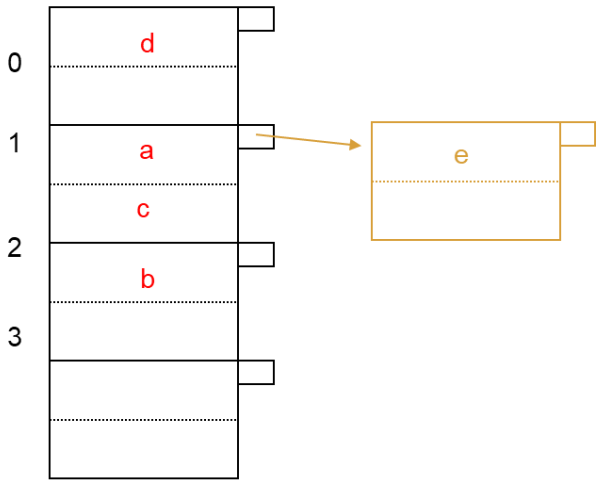
- Satisfies the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.
- Keys that are close in some sense must yield hash values that are far apart, independent of any patterns that might exist in the data.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

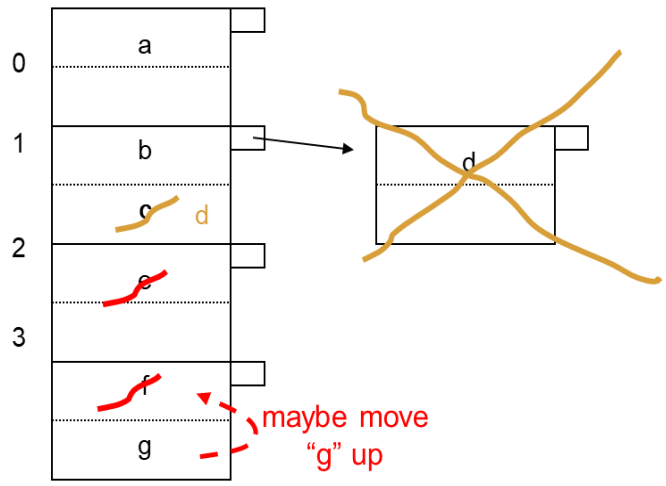
Deletion:

Delete:

e

f

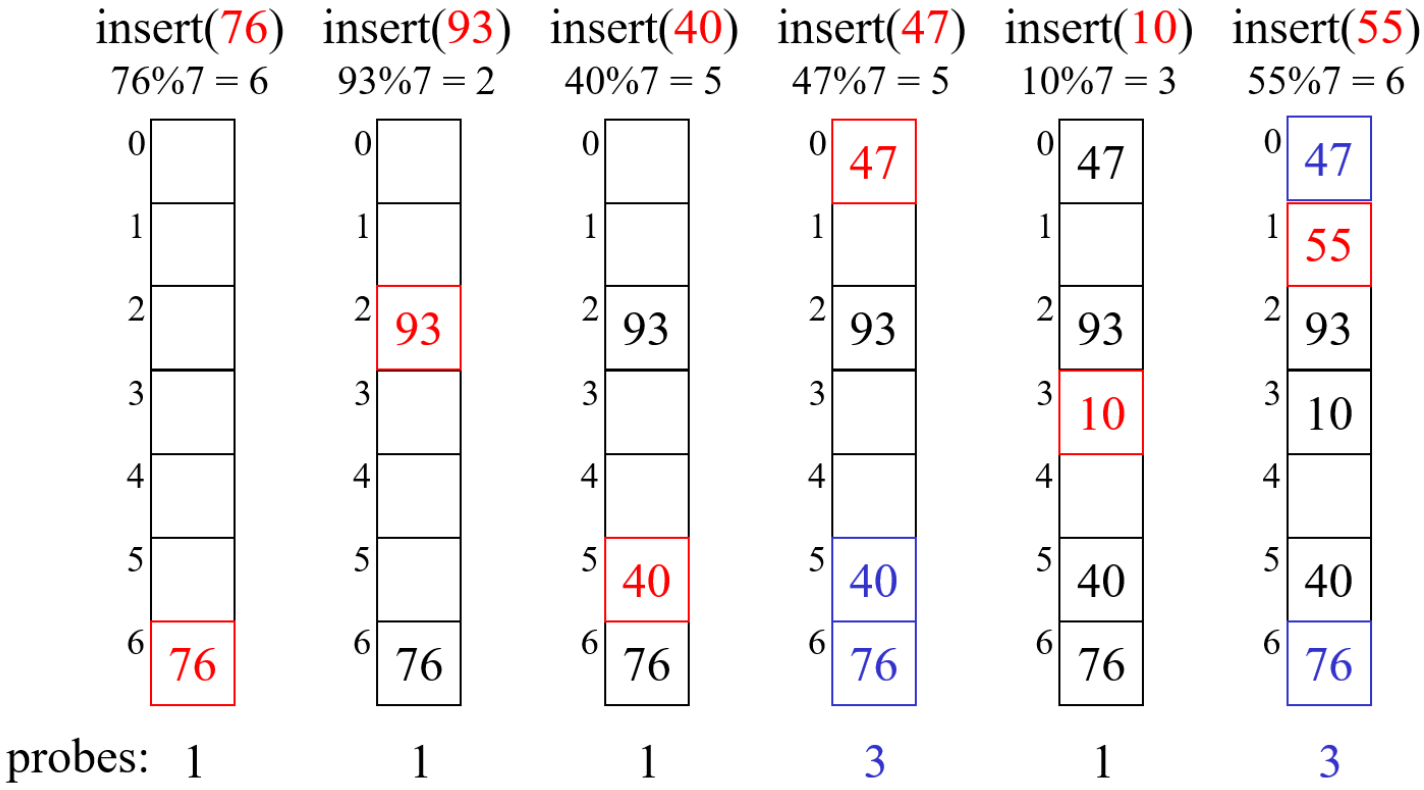
c



2. Probing -> Closed Hashing/Open Addressing:

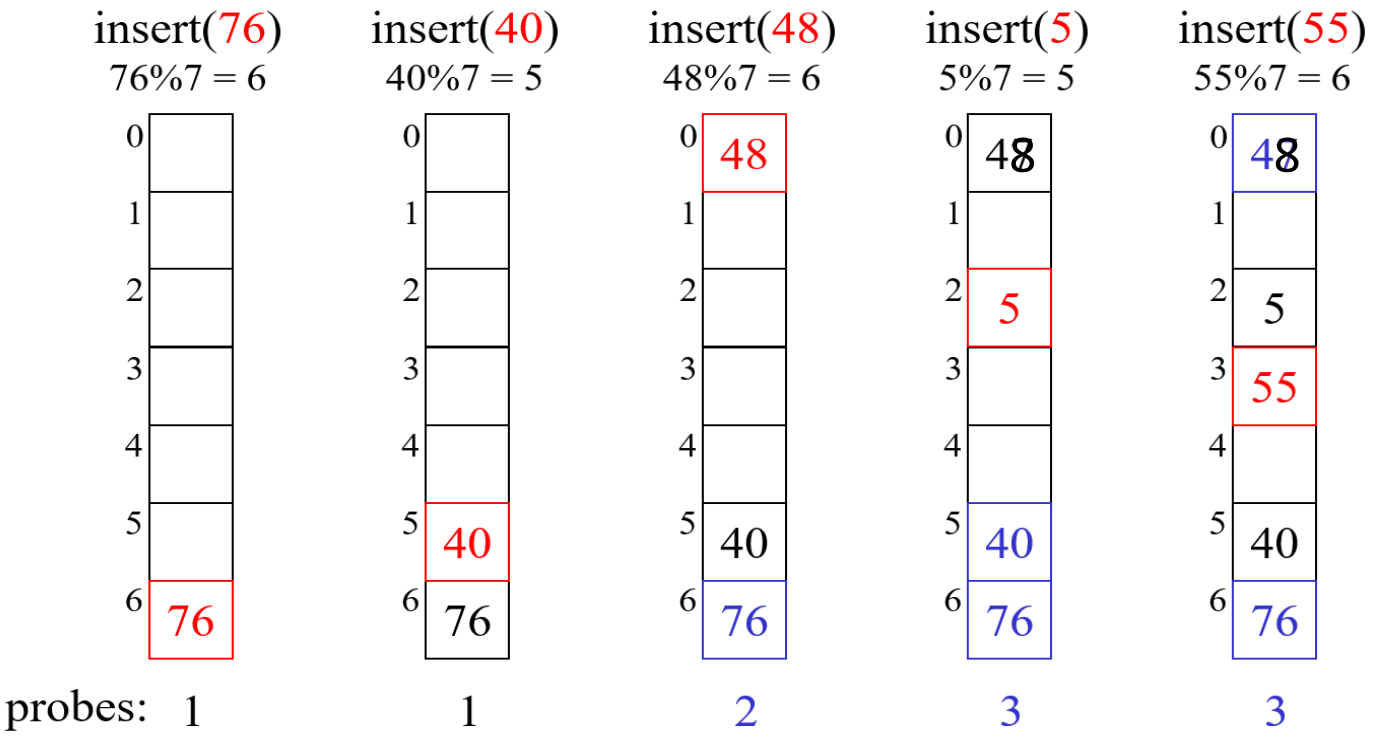
- Linear Probing:
  - auxiliary hash function: h′ : U -> { 0, 1, … , m-1 }
  - hash function: h(k,i) = (h'(k)+i) mod m ; i = 0,1, … , m-1
  - First tries to insert into h'(k), if collusion found then search the next available slots for insertion.

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



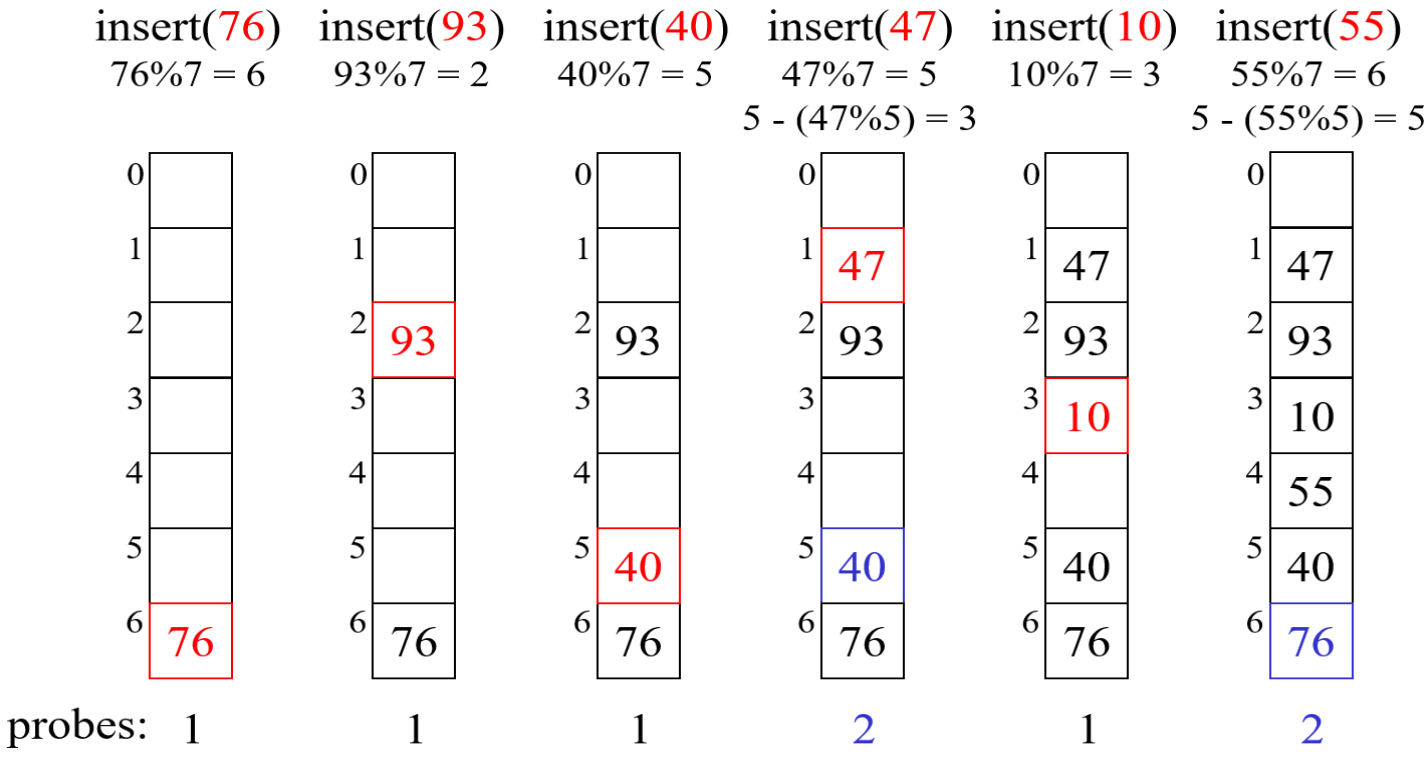
- Quadratic Probing:
  - hash function: h(k,i) = (h'(k) + c_1* i + c_2*i^2) mod m; i = 0, 1, … , m-1

Here, 1 key / bucket, c1 = 0 and c2 = 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



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.

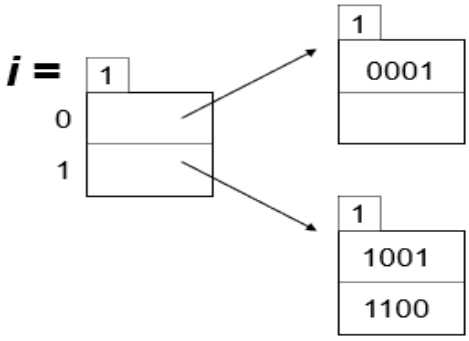
Two types:

- Extensible Hashing
- Linear Hashing

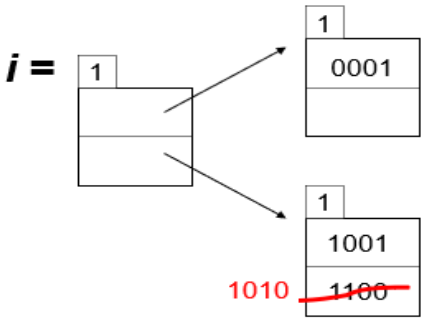
Extensible Hashing:

- 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. $\mathbf{2}^{\mathbf{i}}$ 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.

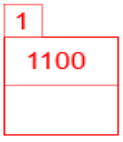
Insertion:

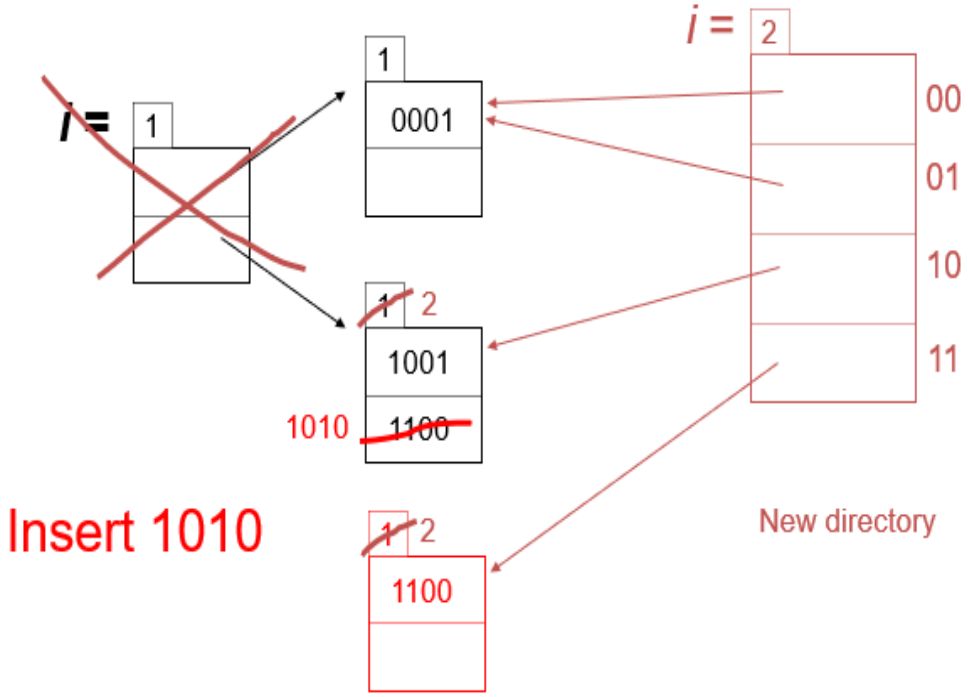


Insert 1010

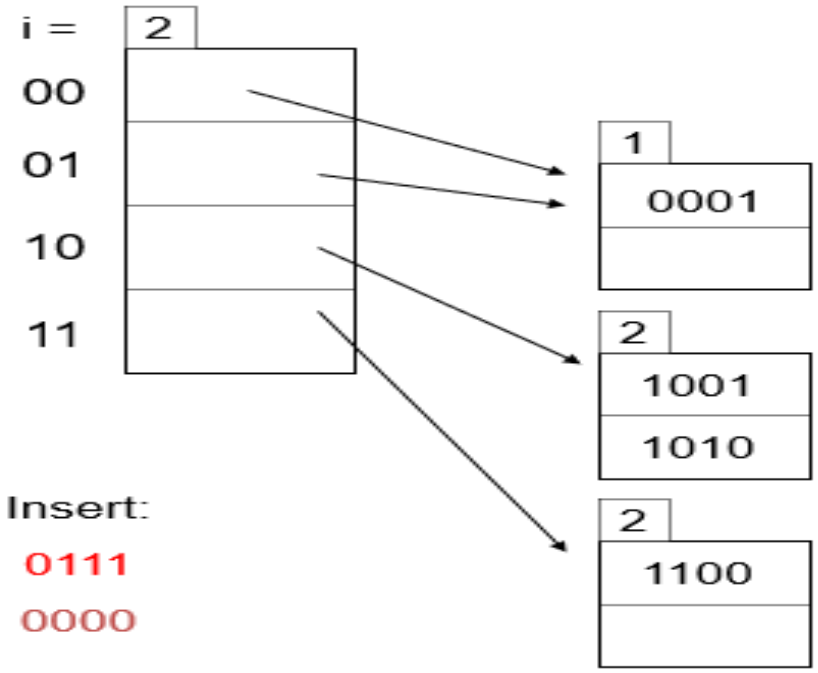


Insert 1010

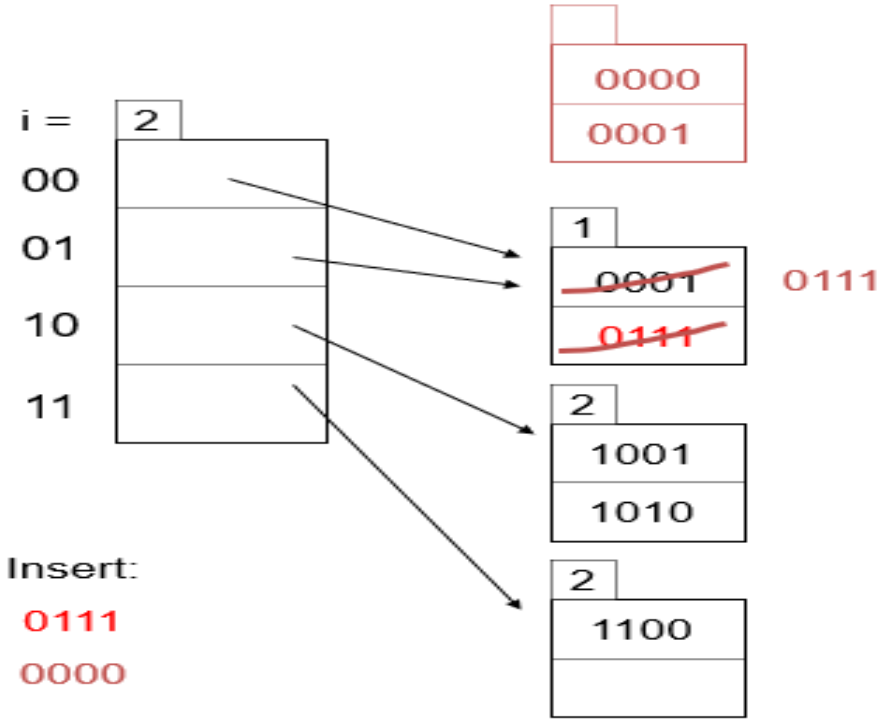


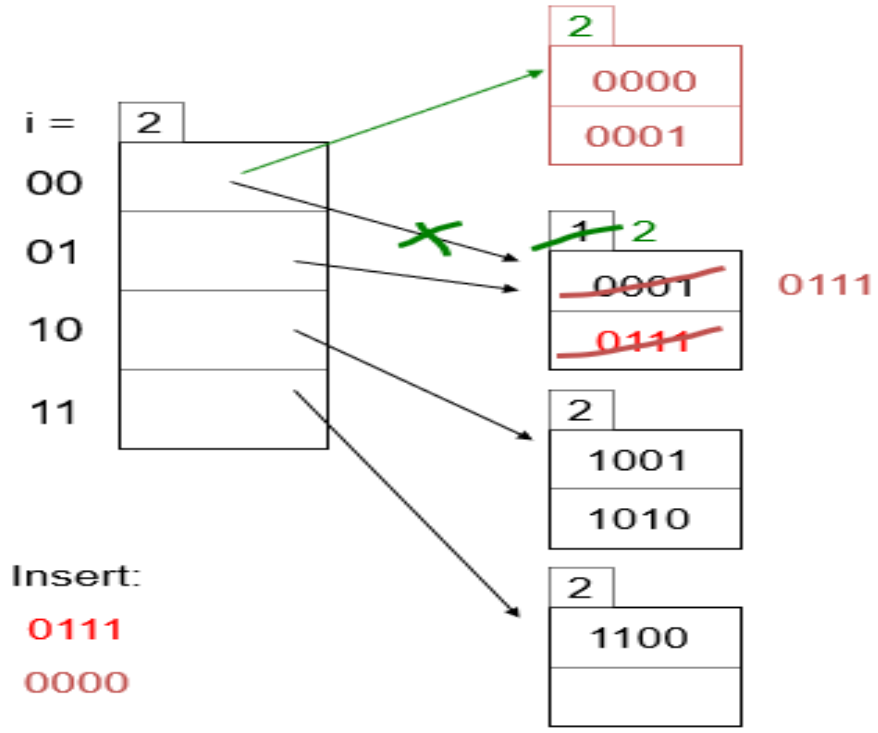


Final structure:

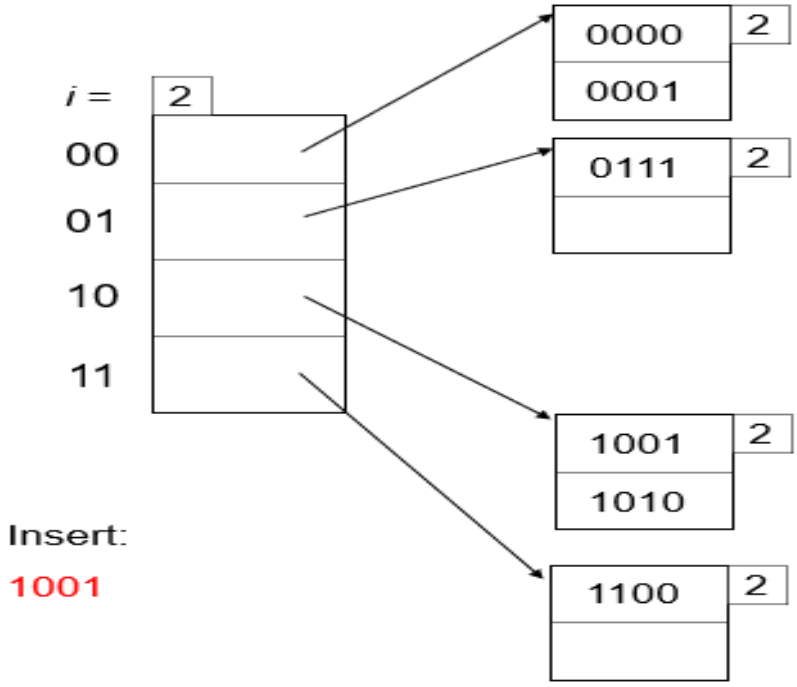


insert 0111 , 0000 =>

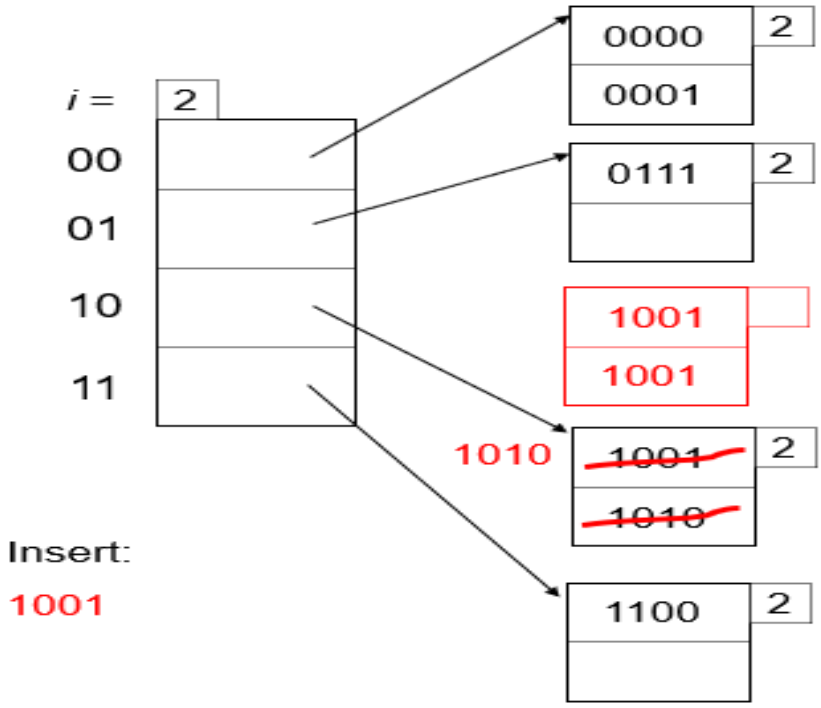




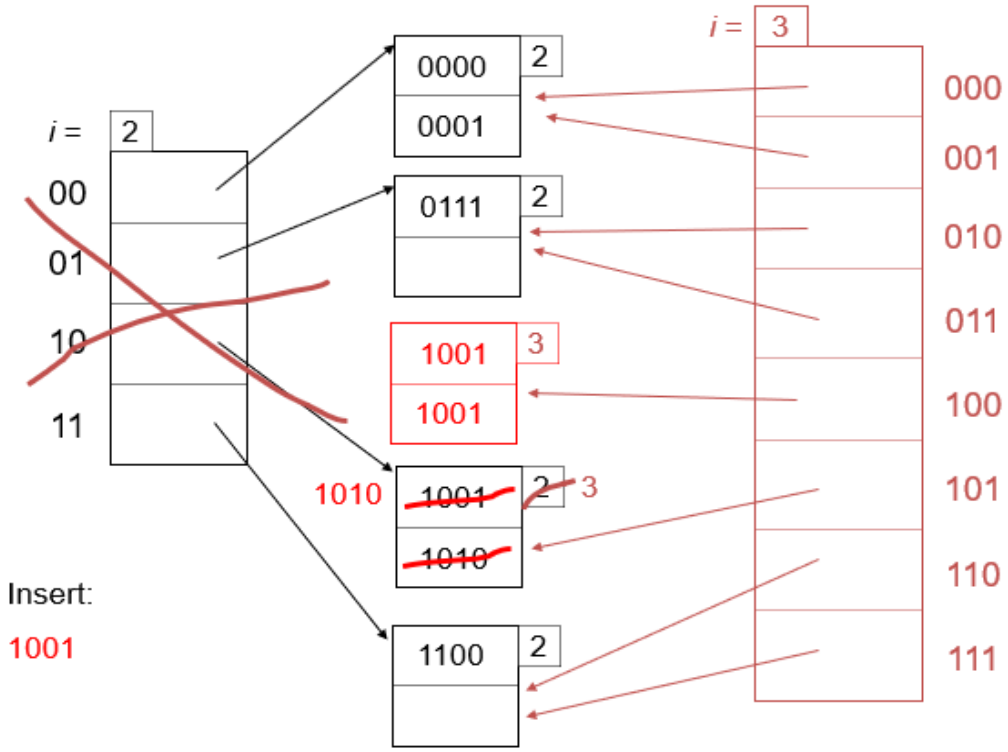
final solution:



insert 1001 =>



final solution:



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;