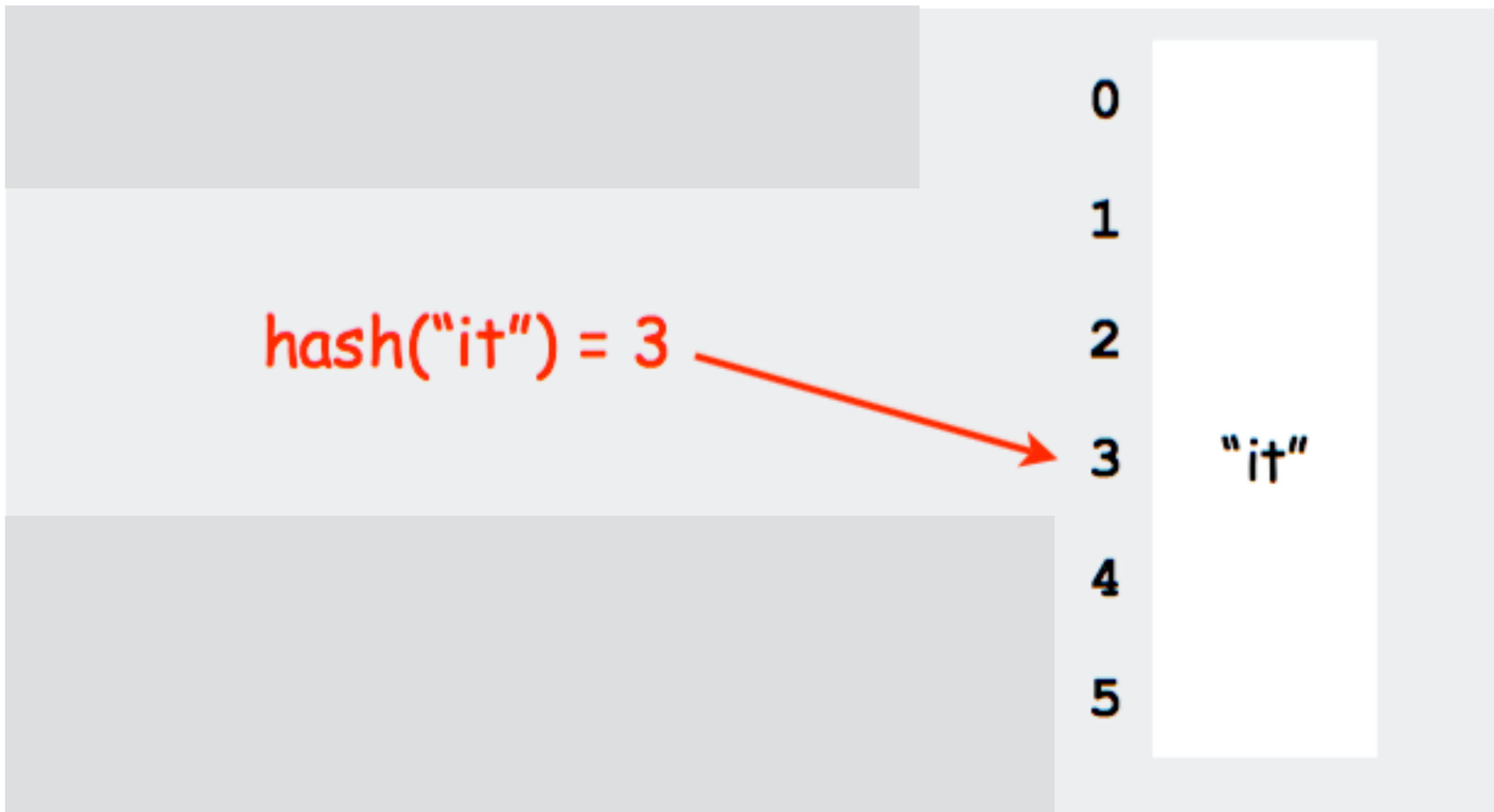
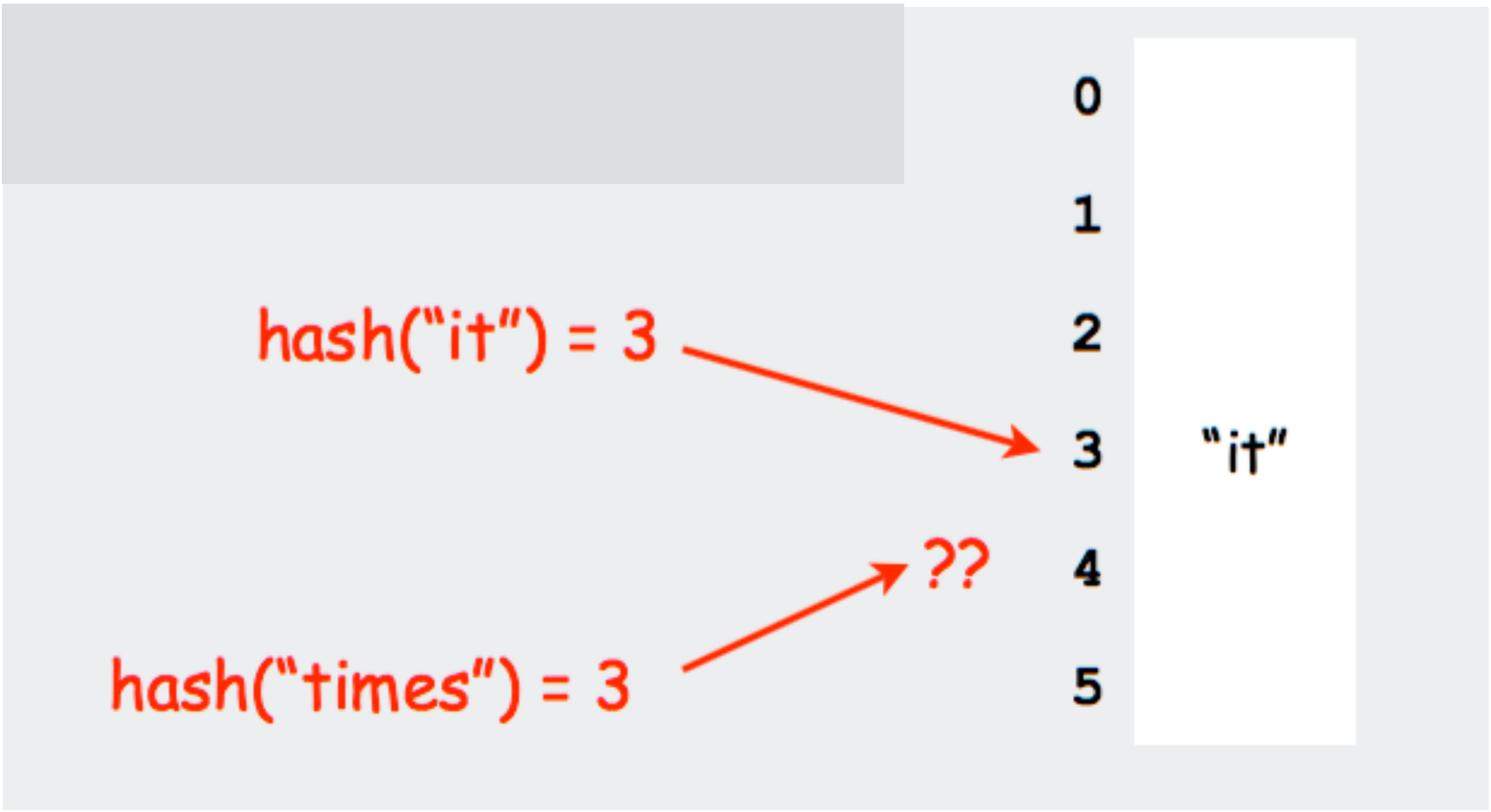
Data Structure:
Hashing

hashing



hashing



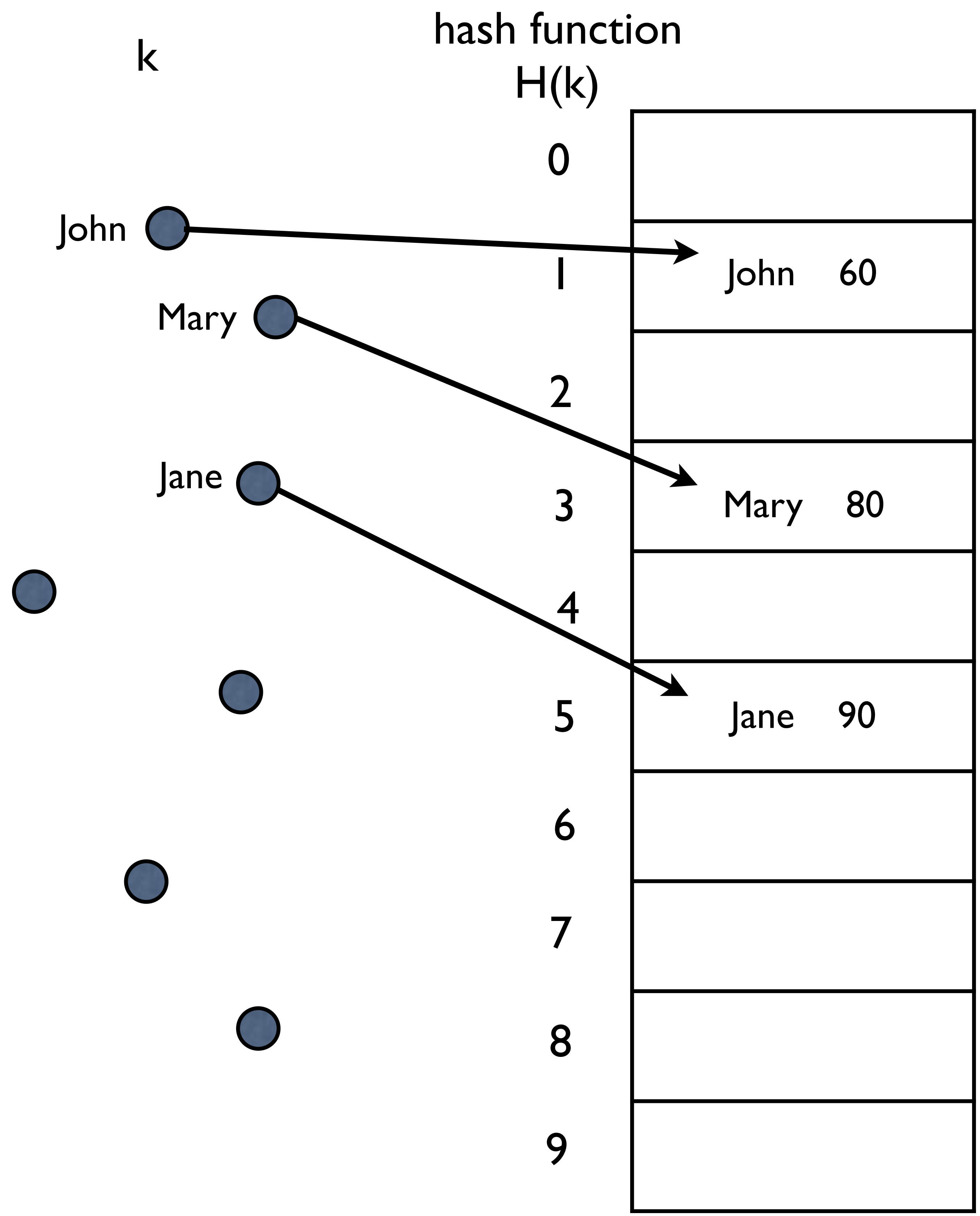
hashing

- hashing is a technique used for performing insertion, deletion, and finding in constant time
- tree operations such as FindMin, FindMax, and the printing all elements in sorted order are not supported

- hash table is an array of fixed size, containing the keys
- hash function maps each key to some cell in the hash table
  - should be easy to compute
  - should minimize the number of collision
  - uniform hash function, the probability of $h(k) = i$ is $1/b$ for all i (b is bucket size)
- collision occurs when different keys are mapped to the same cell

Hashing

H(John) = 1



Hash functions

- adding all characters (alphabets) in the key
  - for example, h(abc) = h(bca) = 1+2+3 = 6 (a=1, b=2, c=3)
  - all ordering information is lost
  - the number of hash function value is too small, considering the number of possible keys
    - for example, length(key) = 8
    - the number of hash function value H(key) = 26 * 8 = 208
    - the number of possible keys = 26^8

- polynomial function (using horner's rule)
  - $h(k) = k_1 + 27k_2 + 27^2k_3 = ((k_3) * 27 + k_2) * 27 + k_1$
  - number gets easily too big

- division
  - h(k) = k mod m, where m is the size of hash table
  - good choice for m is a prime number

resolving collision

separate chaining:
put keys that collide in a list associated with index

open addressing:
when a new key collides, find next empty slot and put it there

resolving collision: separate chaining (open hashing)

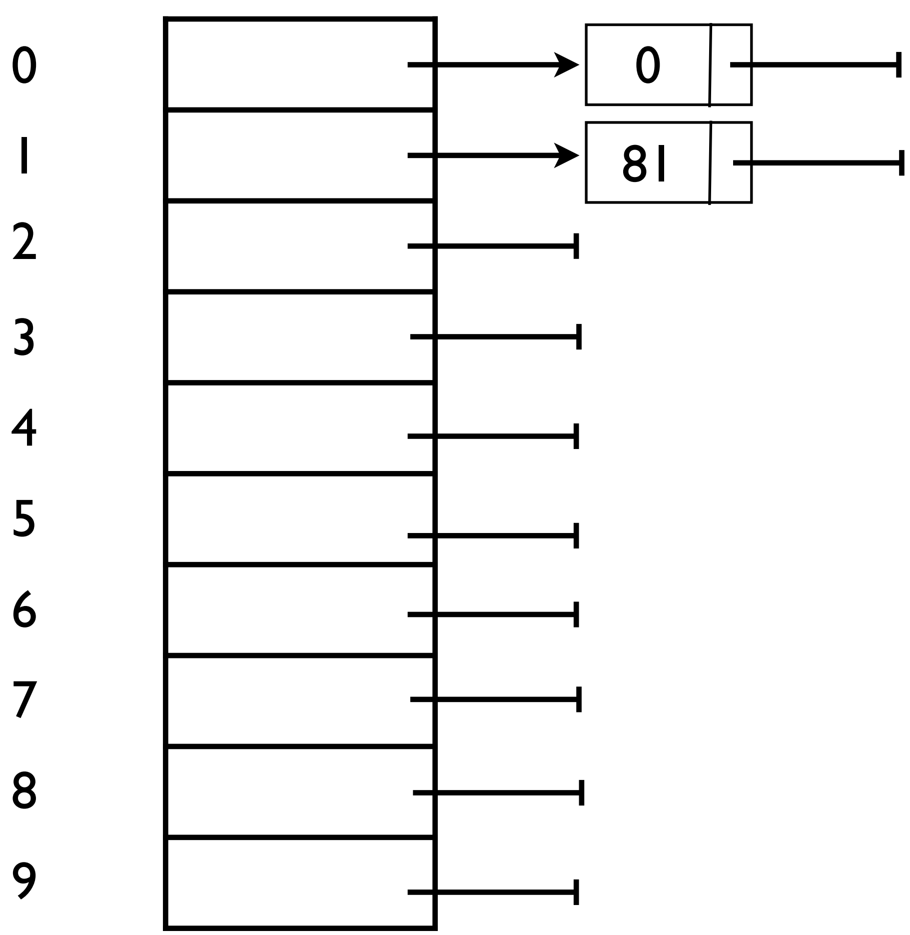
- keep a list of all elements that hash to the same value
- operations
  - Find: use hash function to determine which list to traverse
  - Insert: traverse down the list to check whether the element is in the list

    if not, it is inserted at the front (or at the end)

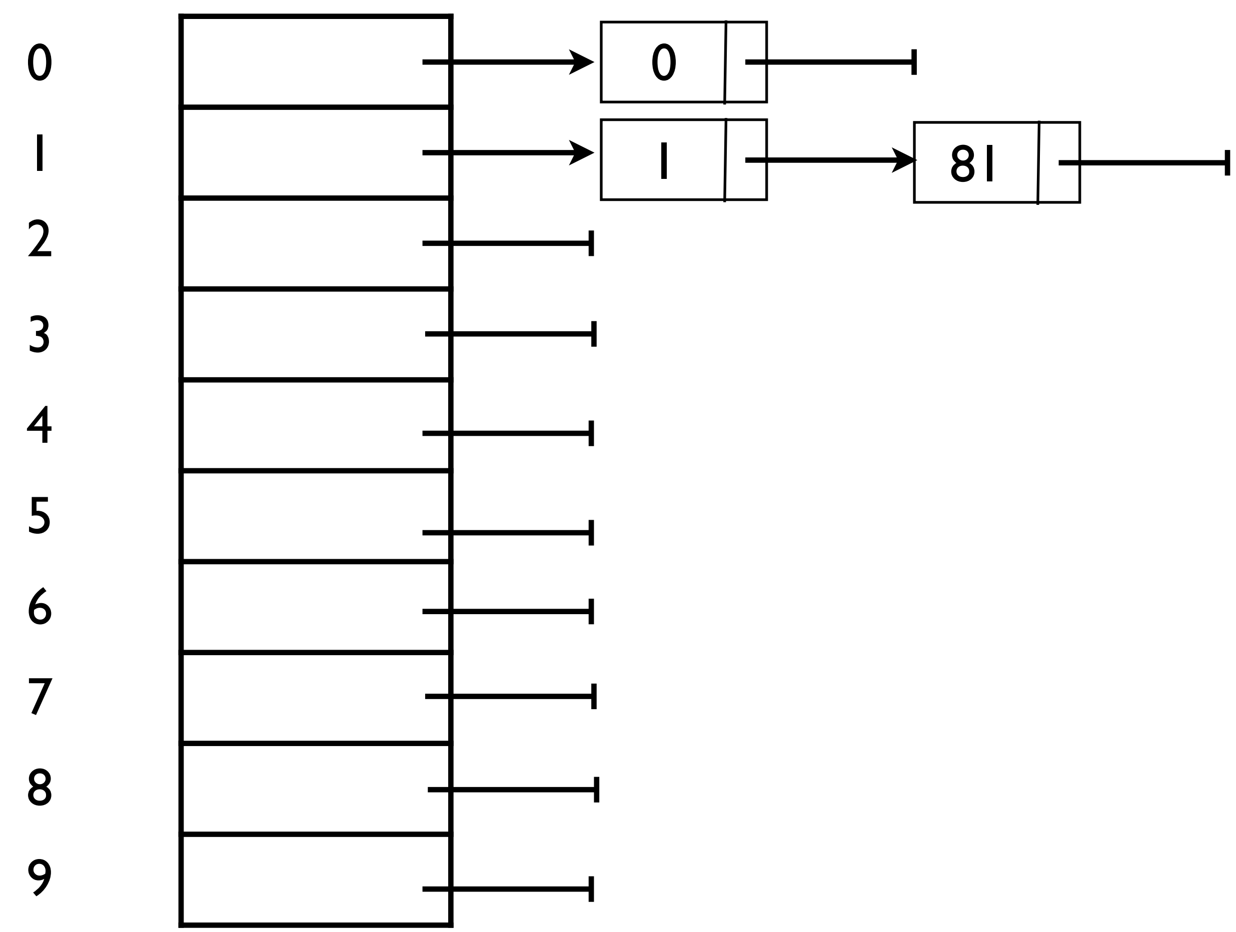
resolving collision: separate chaining (open hashing)

H(x) = x %10

insert (81)
insert(0)

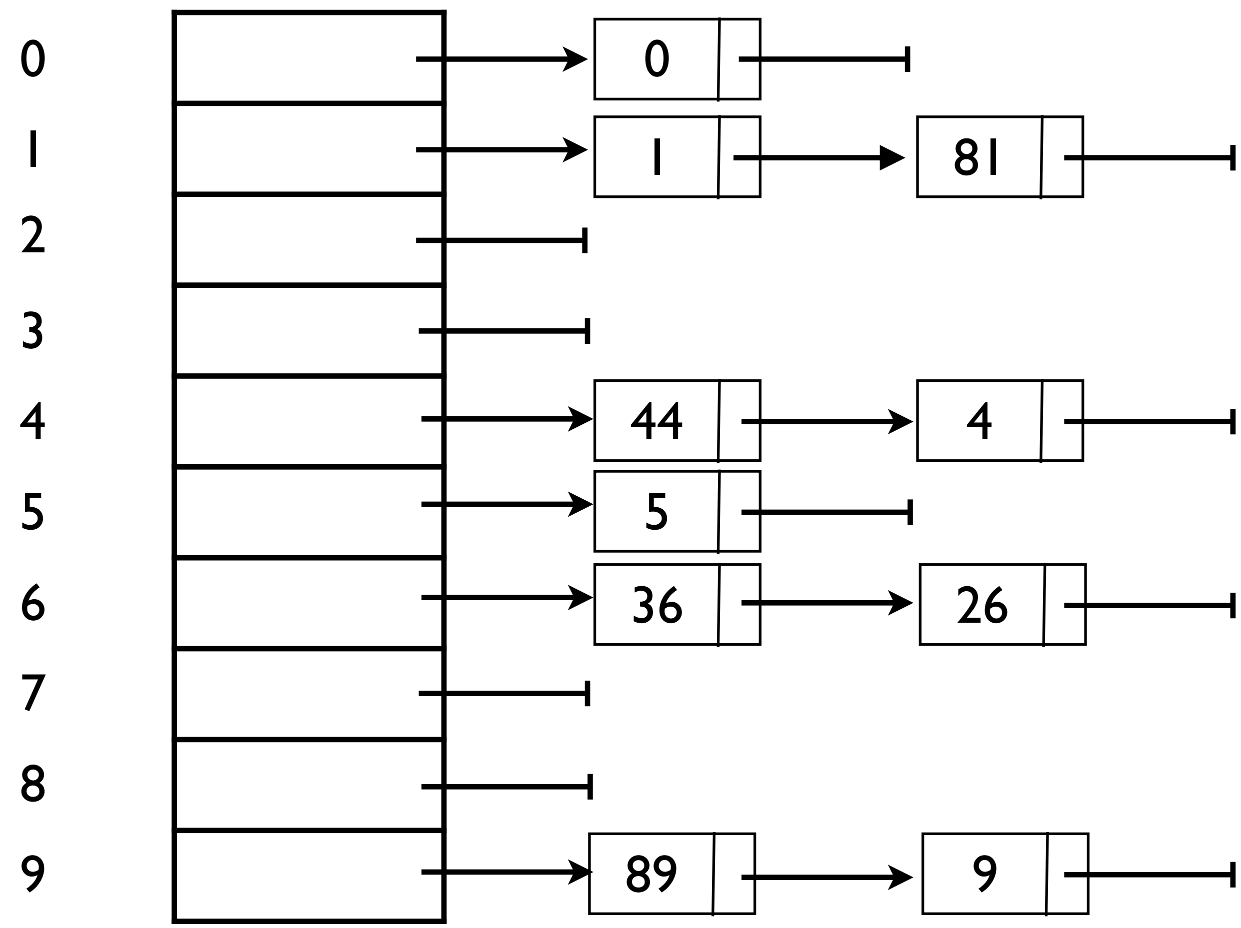


insert(1)



resolving collision: separate chaining (open hashing)

insert (81)
insert(0)
insert(1)
insert(4)
insert(26)
insert(5)
insert(9)
insert(44)
insert(36)
insert(89)



resolving collision: separate chaining

```
typedef struct  ListNode* Position;
typedef Position List;

struct ListNode {
        ElementType    Element;
        Position   Next;
}

struct HashTbl{
        int TableSize;
        List*  TheLists;
}
```

resolving collision: separate chaining

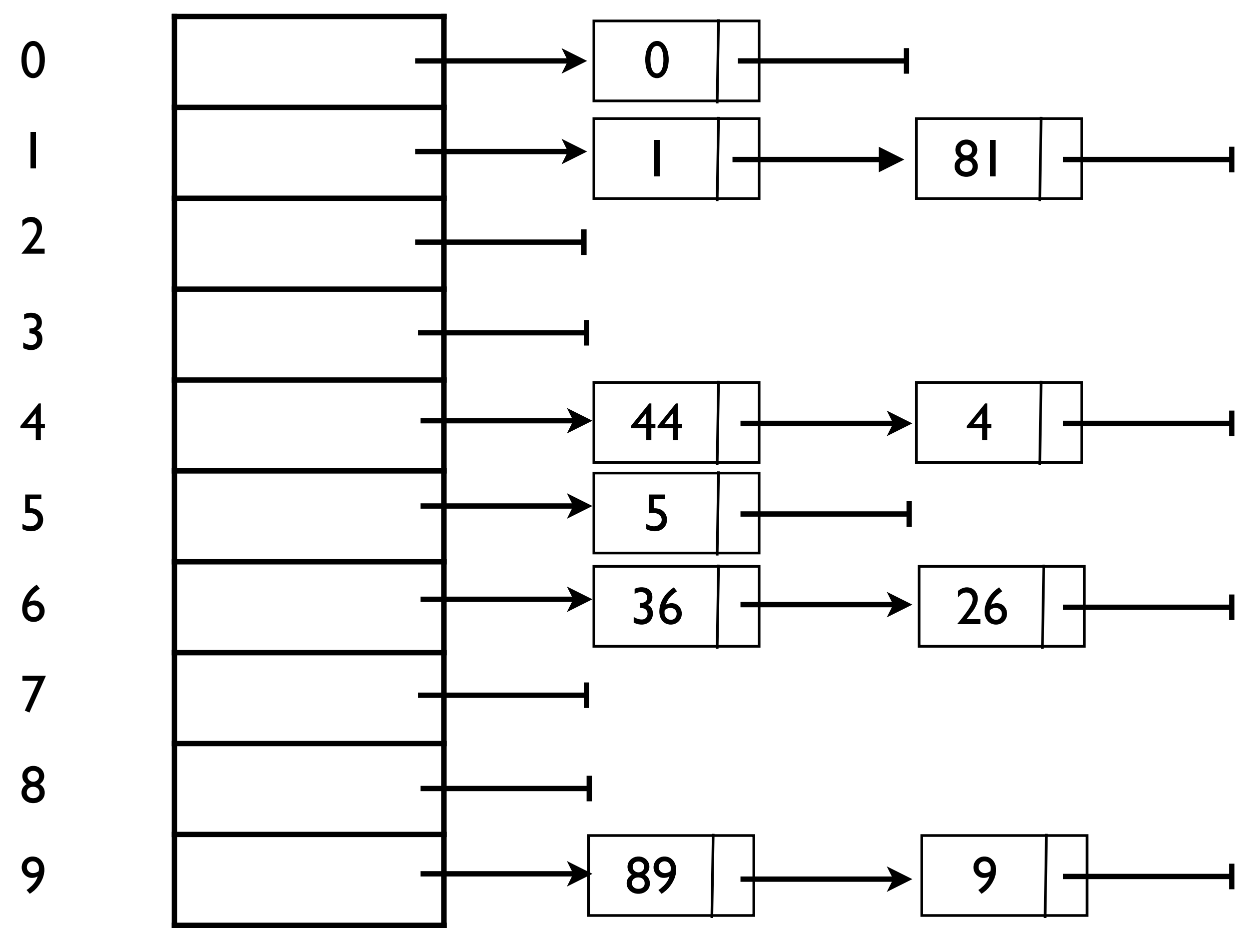
```
Position Find (ElementType Key, HashTable H){

    Position P;
    List L;

    L = H -> TheLists [ Hash(key,  H->TableSize)];
    P = L -> Next;

    while (P != NULL && P->Element != Key)
        P = P->Next;

    return P;
}
```



resolving collision: separate chaining

```
void Insert (ElementType Key,   HashTable H){

        Position Pos, newCell;
        List L;

        Pos = Find(Key,   H);

        if (Pos == NULL){

                NewCell = malloc(sizeof (struct ListNode));
                NewCell ->Element = Key;

                L = H->TheLists[Hash(Key, H->TableSize)];
                NewCell ->Next = L->Next;
                L->Next = NewCell;
        }
}
```

resolving collision: separate chaining

- load factor: the ratio of the number of elements in the hash table to the table size

  $\lambda = n / m$

  n is the number of keys in the table, m is the size of the table

- successful search (i.e. no clustering): 1 (hash function) + (λ/2) =O(1)
- unsuccessful search: 1 + λ =O(1)
- needs extra space and operation for pointers and new nodes

resolving collision: open addressing (closed hashing)

- all the keys are stored in the table without pointers
- if a collision occurs, alternative cells are tried until an empty cell is found
- try $h_0(key), h_1(key), h_2(key), \ldots$
  - where $h_i(key) = (Hash(key) + F(i)) \bmod m$
  - $F(i)$ is the collision resolution strategy
  - linear probing: $F(i)$ is a linear function, $F(i) = i$

    for example, $h_1(key) = (Hash(key) + 1)$, $h_2(key) = (Hash(key) + 2), \ldots$
  - quadratic probing: $F(i)$ is a quadratic function, $F(i) = i^2$

    for example, $h_1(key) = (Hash(key) + 1)$, $h_2(key) = (Hash(key) + 4), \ldots$

resolving collision: linear probing

- F(i) is a linear function. for example, F(i) = i

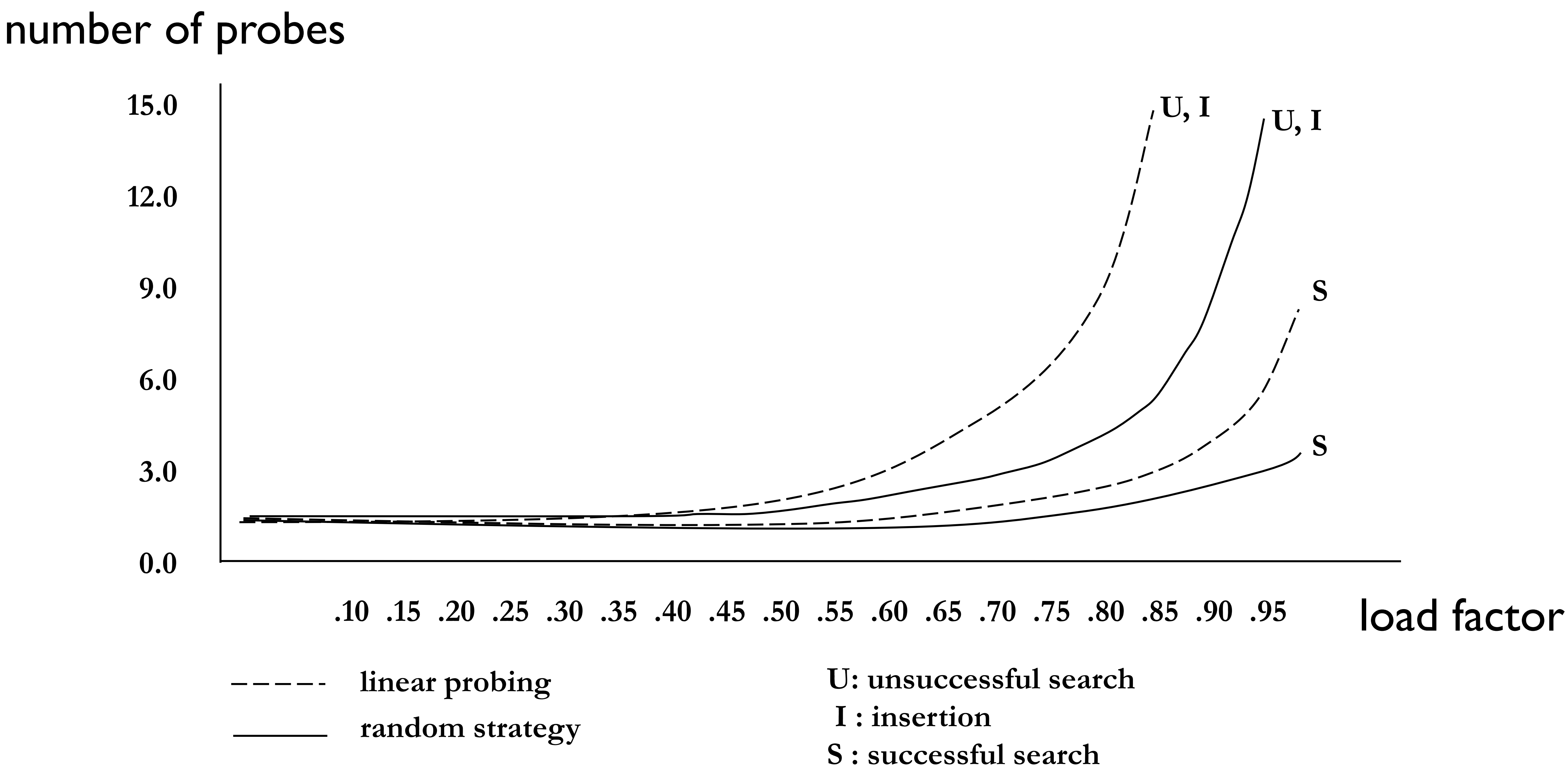
  inserting keys: 89, 18, 49, 58, 69

0		0	49	0	49	0	49
1		1		1	58	1	58
2		2		2		2	69
3		3		3		3	
4		4		4		4	
5		5		5		5	
6		6		6		6	
7		7		7		7	
8	18	8	18	8	18	8	18
9	89	9	89	9	89	9	89

resolving collision: linear probing

- primary clustering: any key that hashes into the cluster will require several attempts to resolve the collision and then it will add to the cluster
- expected number of probes
  - successful search $$S = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$
  - unsuccessful search $$U = \frac{1}{2}\left(1 + \left(\frac{1}{1-\lambda}\right)^2\right)$$
  - as λ approaches to 1, the search time grows to infinity
  - linear probing does well if the table is less than 75% full

resolving collision: linear probing



resolving collision: quadratic probing

- a collision resolution method that eliminates the primary clustering problem of linear probing
- collision function $F(i) = i^{2},$ $h_i(key) = (Hash(key) + F(i)) \bmod m$

inserting keys: 89, 18, 49, 58, 69

0		49	49	49
1				
2			58	58
3				69
4				
5				
6				
7				
8	18	18	18	18
9	89	89	89	89

resolving collision: double hashing

- use other hash function for random probing
- for example, $(h_i(key) = (Hash(key) + F(i)) \bmod m)$

  $Hash(key) = key \bmod m$

  $F(i) = i * Hash_2(key)$, $Hash_2(key) = R - (key \bmod R)$

R=7

inserting keys: 89, 18, 49, 58, 69

0				69
1				
2				
3			58	58
4				
5				
6		49	49	49
7				
8	18	18	18	18
9	89	89	89	89

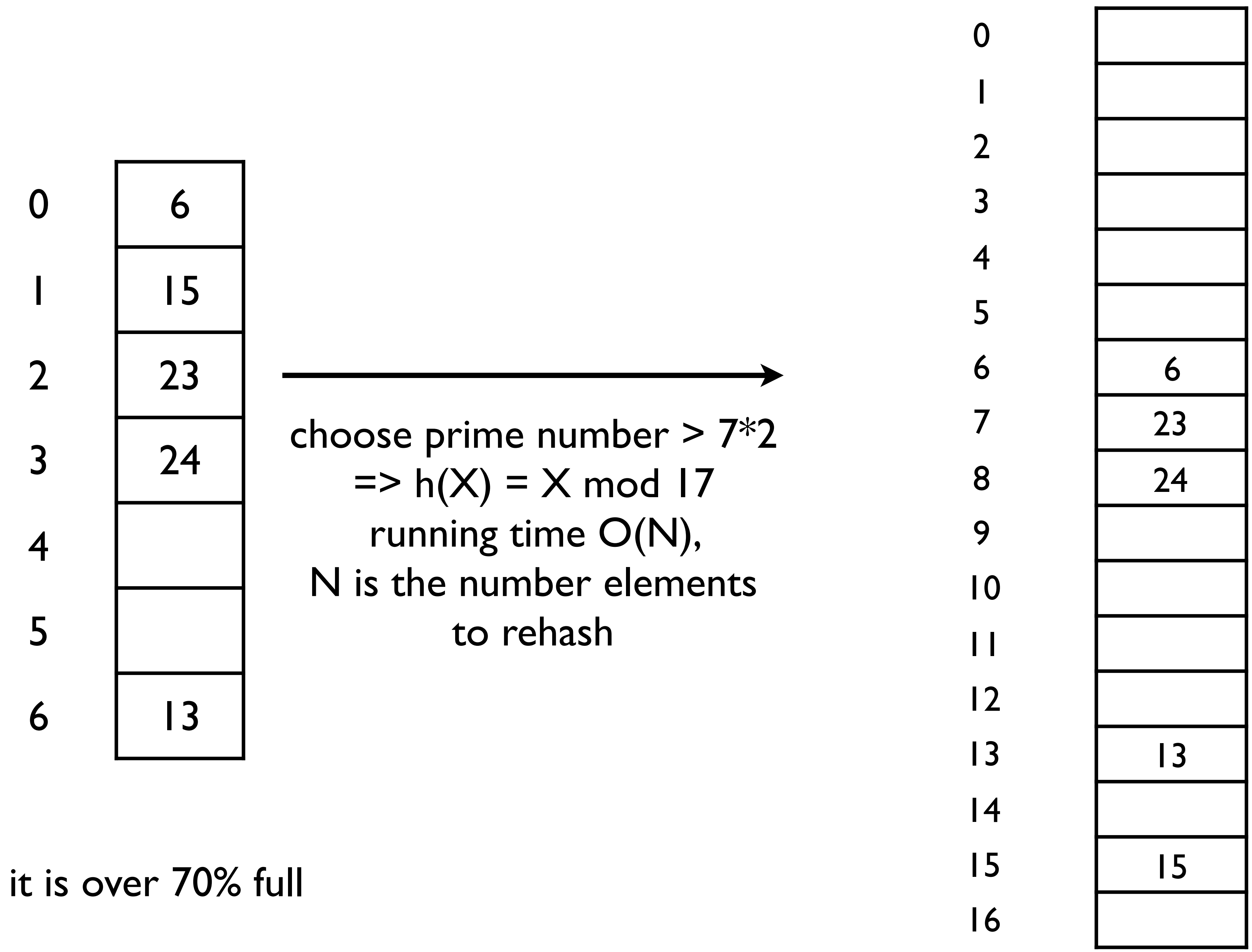
49: $Hash_2(49) = 7 - 0 = 7$

58: $Hash_2(58) = 7 - 2 = 5$

69: $Hash_2(69) = 7 - 6 = 1$

rehashing

- if the table gets too full, the running time for the operations start taking too long
- build another table that is about twice as big

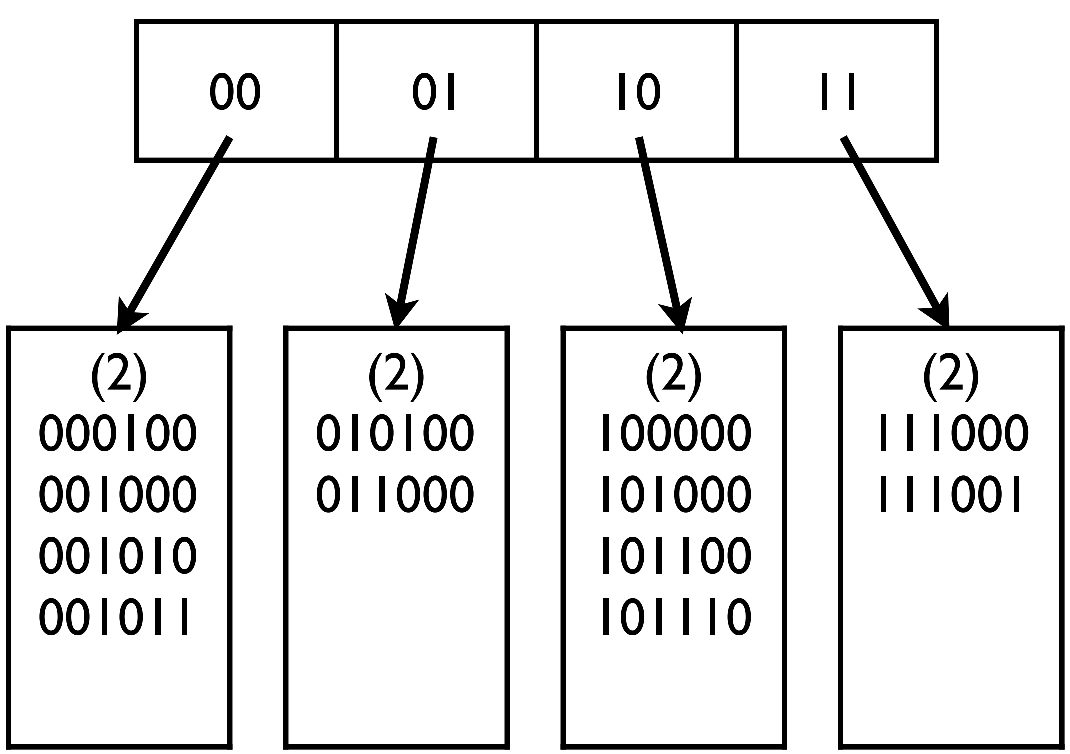


extendible hashing

- what if the hash table is too large to fit in main memory?
  - locality is important for large data structure since disk access is costly but memory access is cheap
  - efficient probing is the lack of locality
  - need a method to reduce the number of disk access

extendible hashing

- The hash table is broken into a number of smaller hash tables, each is called a *bucket*.
- The maximum size of each bucket is the size of a disk page.
- To find which bucket to search for, we store a data structure called *directory* in main memory, and each entry in the directory holds a disk address of the corresponding bucket.
- Each bucket can hold as many records that can be fit in one page, and we will try to keep each bucket at least half full.



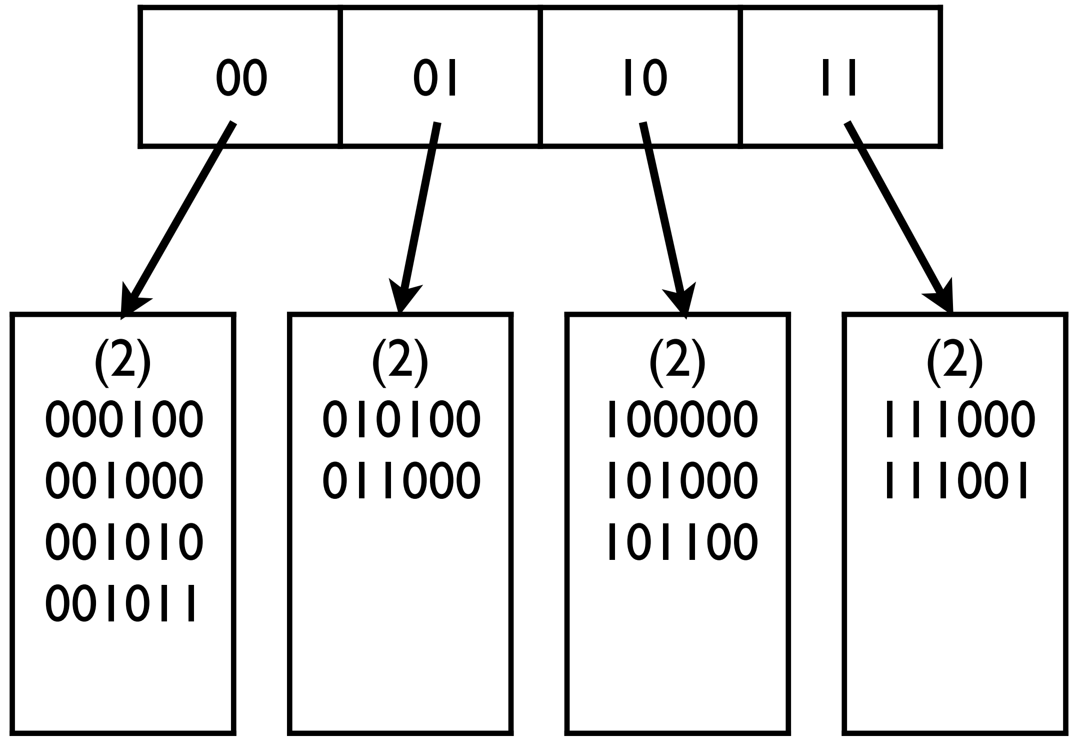
D: the number of bits used by the root

$D = 2$

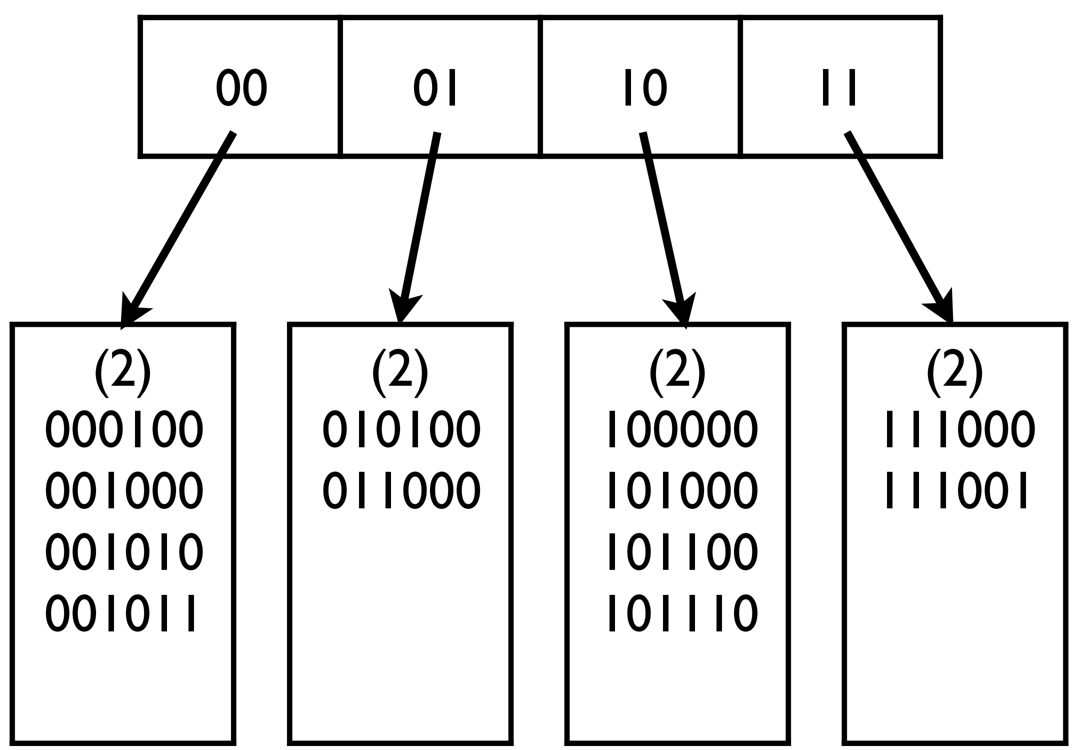
d_L: the number of leading bits that all elements of some leaf L have in common

$d_L = 2$

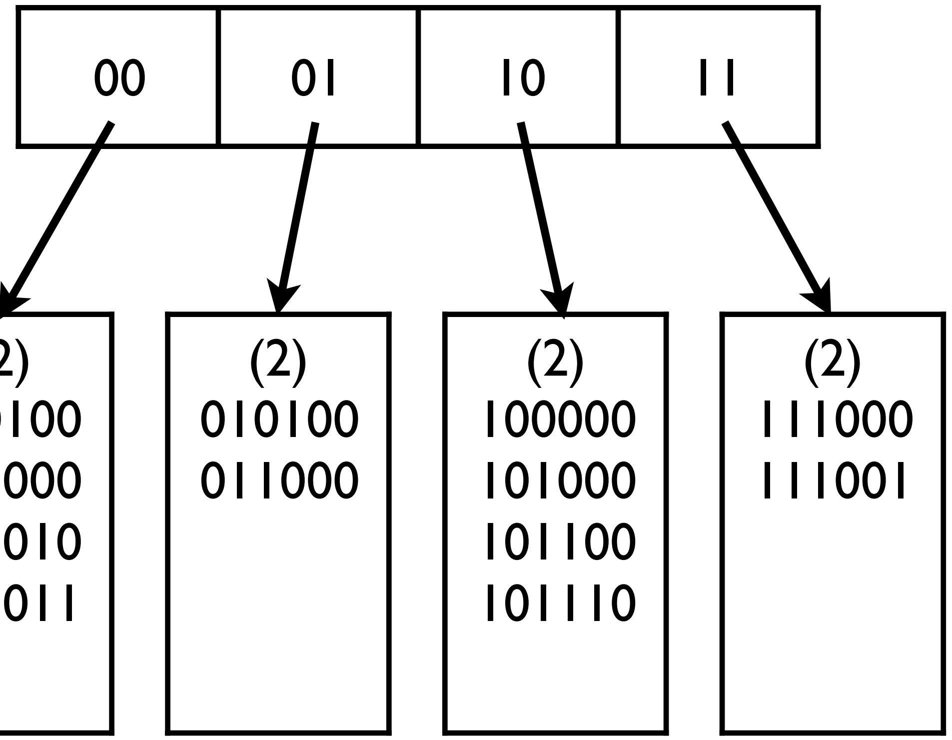
extendible hashing



insert 101110



extendible hashing



insert 100100