Hashing

Hash : Chop and mix

The symbol #

Hash Store : Older form of supermarkets

Hashing

Hash Based Indexing

- Hash indexing is based on using a specific function h(k) that returns an address value . The hash value can be used as a pointer for the location of a record.
- The hash function *h(key) provides a key-to-address translation.*

h(key)→ address

In file processing, hashing can be used in two ways:

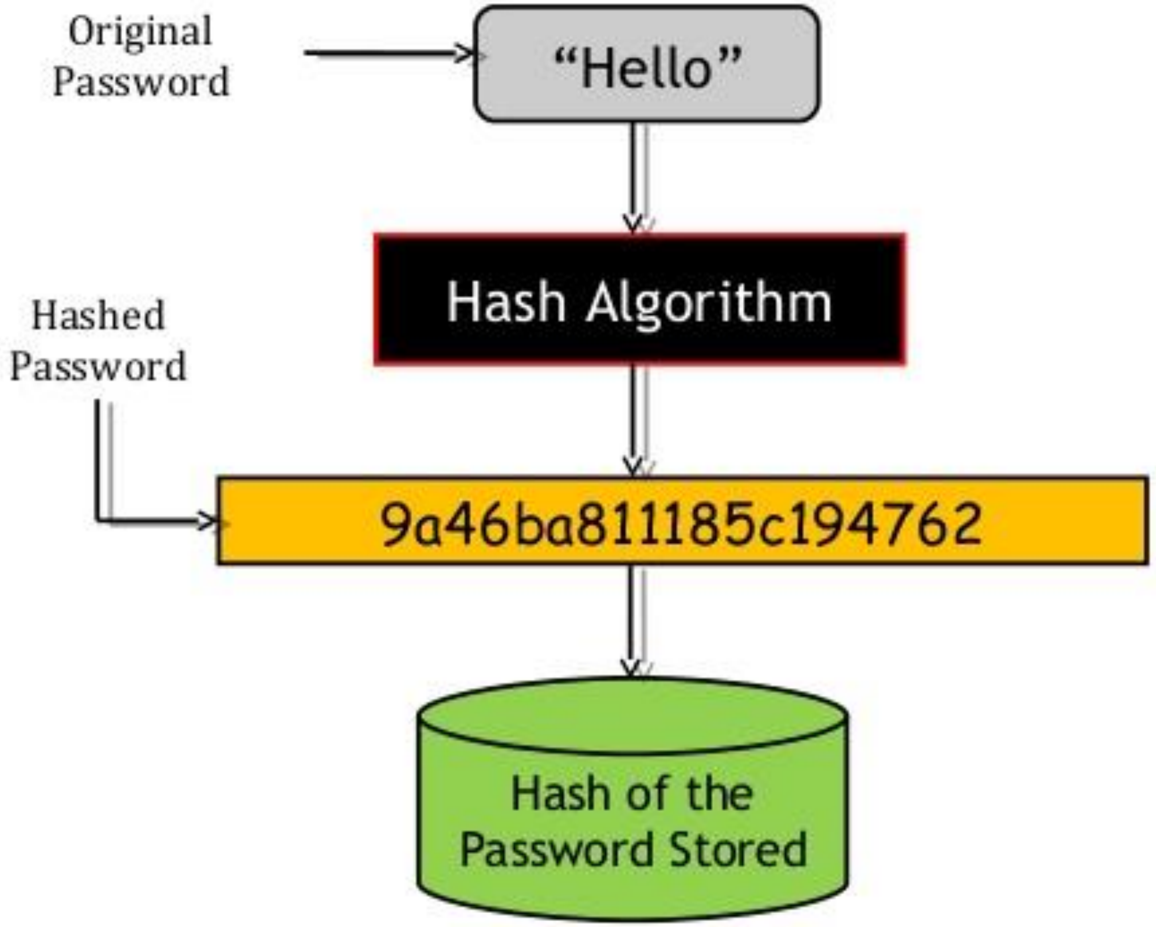
1.As a storage method→Record positions in a file are determined by the hash function.

2.As an indexing method→Hash function is used to access records whose key values are given.

Hashing in Cryptography

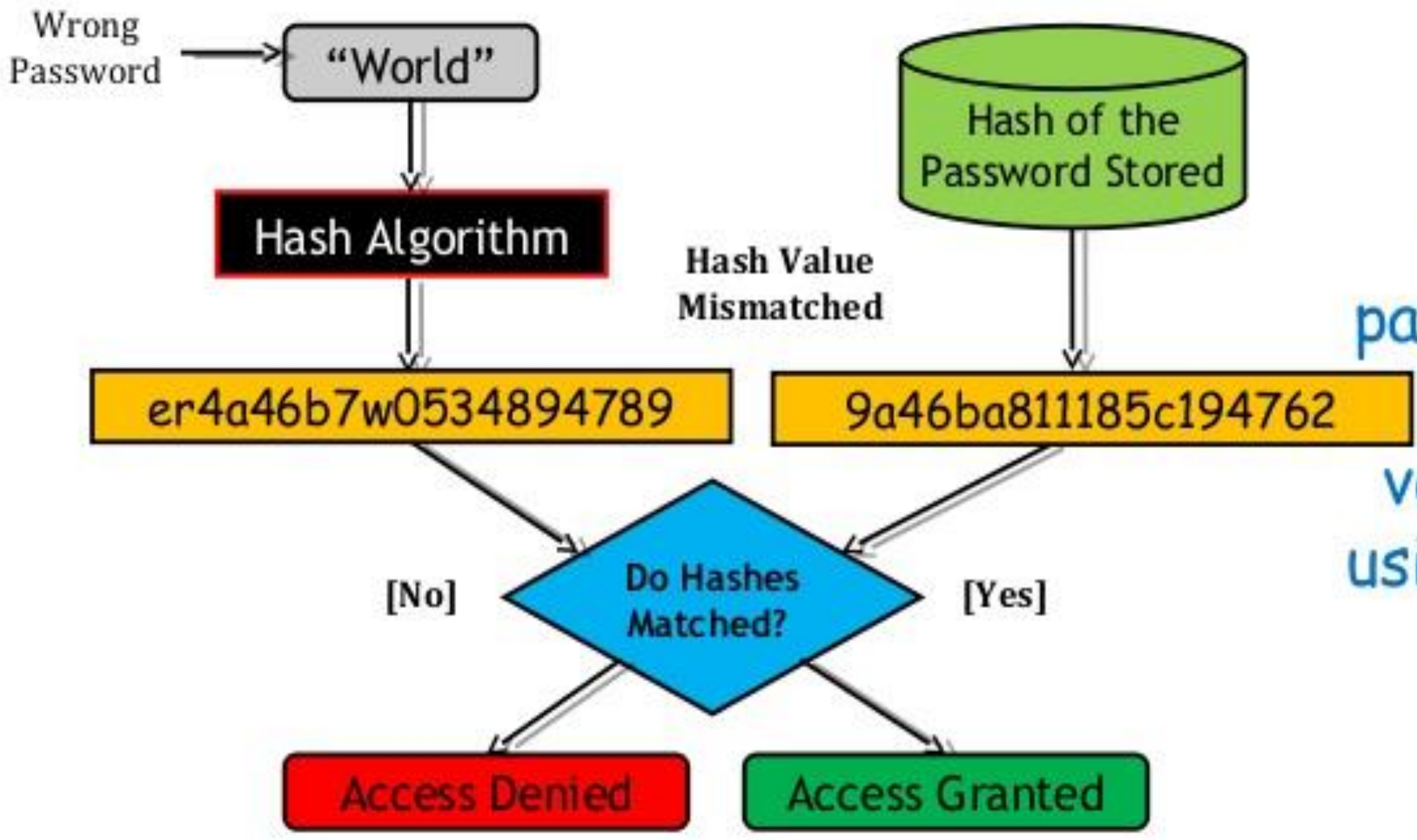
- Another area where hashing is used extensively is cryptography.
- Cryptographic Hashing is a method of cryptography that converts any form of data into a unique string of text.
- Any piece of data can be hashed, no matter its size or type.
- A hash is designed to act as a one-way function: you can put data into a hashing algorithm and get a unique string. But you cannot decipher the data it represents.
- A common application area is password protection.

Application: Hashing Password



How password is stored using hash

Applications of Hash



How password is verified using hash

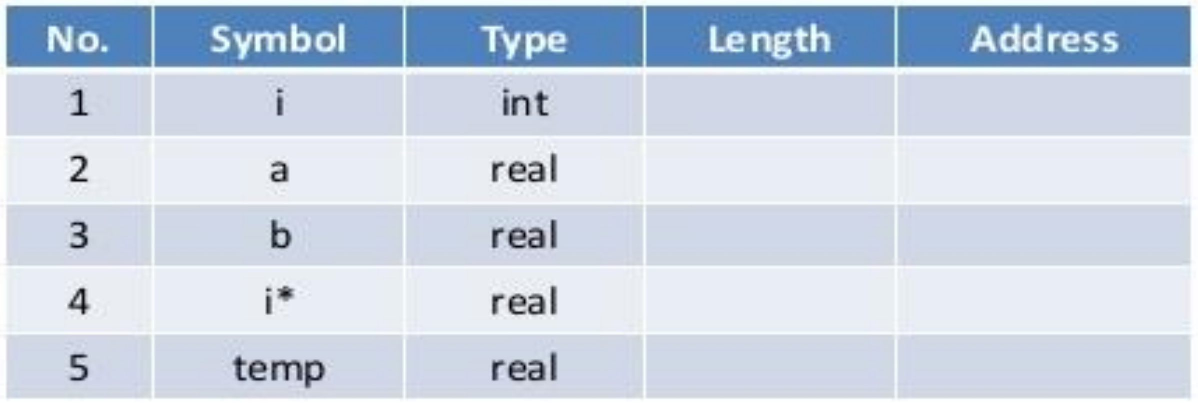
Hash Tables

- In file and database operations , hashing is used for performing insertions, deletions and searches in constant average time (O(1)).
- Hashing is useful as an indexing method and as a search method in symbol tables.
- Symbol table - repository of all information within a compiler. All parts of a compiler communicate thruough this table and access the data-symbols.
- Stores info about scope and binding information of various entities such as variable,label and function names, classes, objects, etc.

Symbol Table: Example

```
i:  integer;
a,b: real;
a := b+i;
```

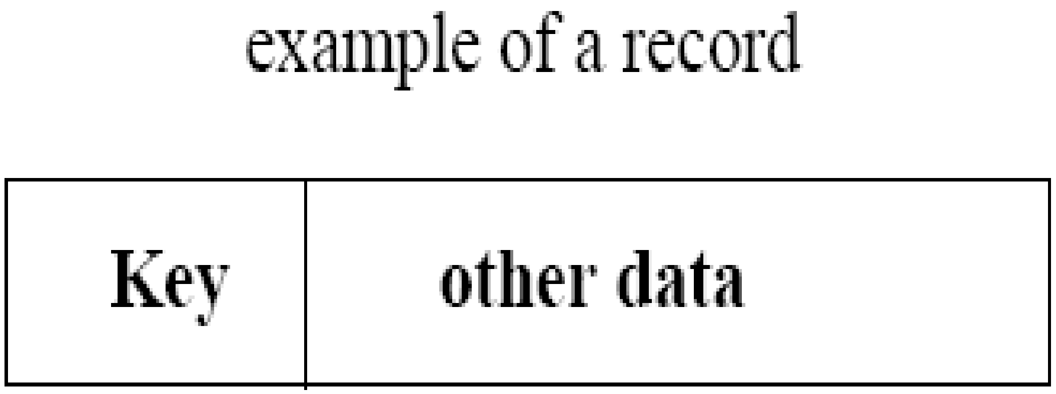
Symbol Table.



No.	Symbol	Type	Length	Address
1	i	int		
2	a	real		
3	b	real		
4	i*	real		
5	temp	real		

Searching Records in Files

- Find items with keys matching a given search key
  - Given a structure A, containing n keys, and a search key k, find the element i such as k=A[i]
  - As in the cases of sorting or searching, a key could be part of a large record.



Searching Records in Files

- Key is used in computing the location value for the item.
  - Key could be an *integer*, a *string*, etc
  - e.g. a name or Id that is a part of a large employee record
- The items that are stored in the hash table T are indexed by values from *0* to *TableSize – 1*.
- Each key is mapped into some number in the range 0 to *TableSize – 1.*
- Mapping is performed using a *hash function*.

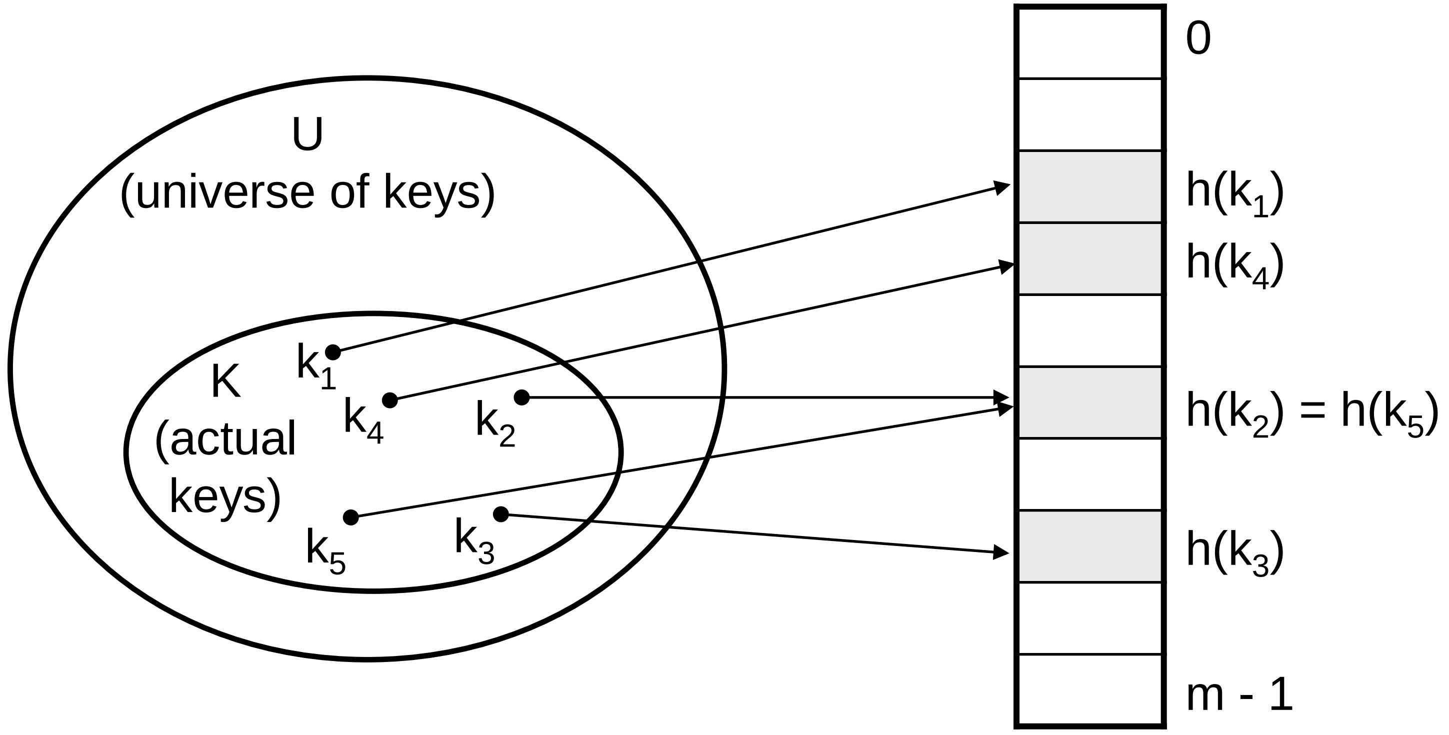
Forming Hash Tables

- Use a function h to compute the slot(Address) for each key k
- Store the element in slot h(k)
- A hash function h transforms a key into an index in a hash table T[0…m-1]:

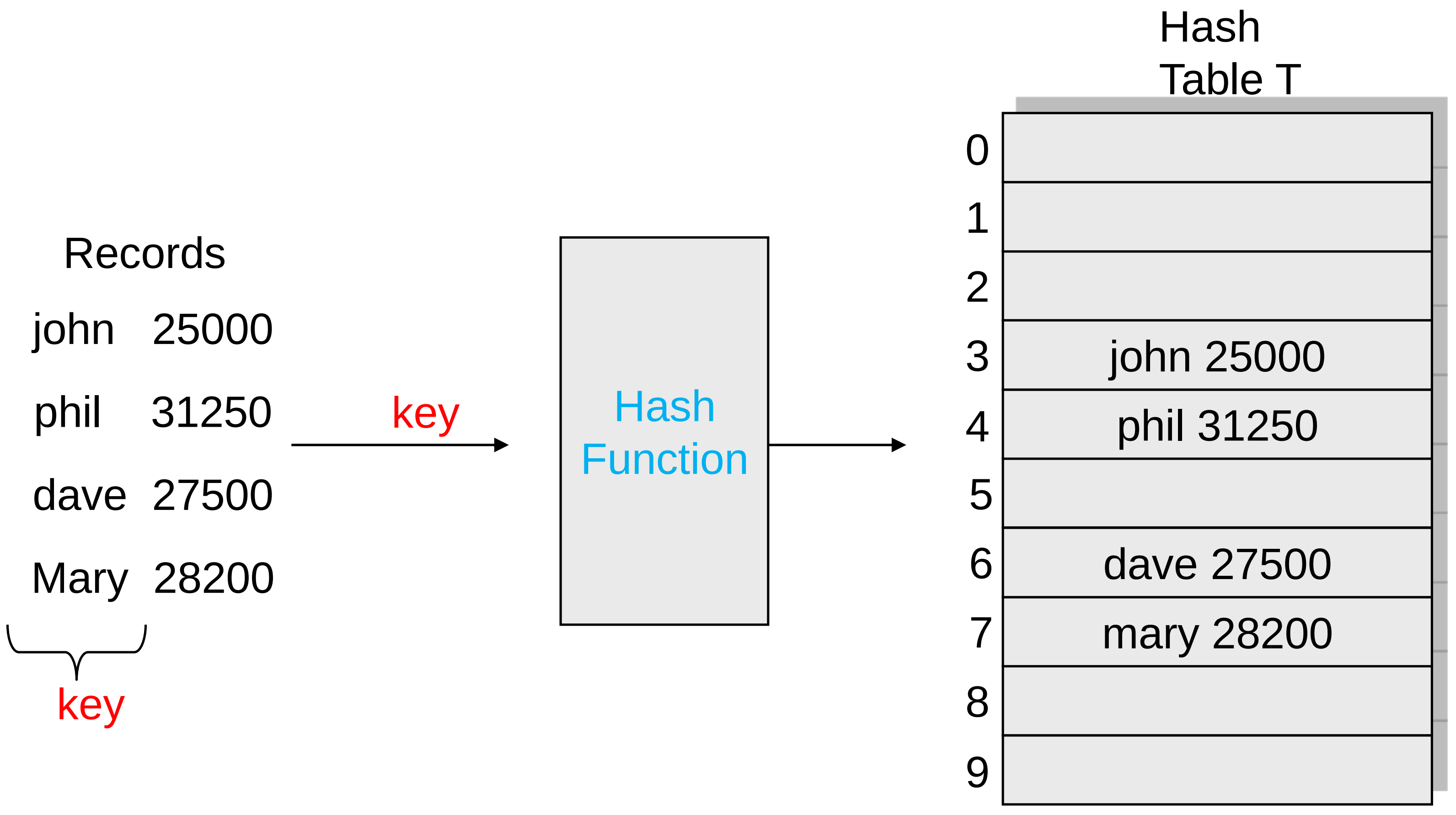
  $h : k \rightarrow \{0, 1, \ldots, m - 1\}$

- We say that h hashes key k to slot h(k)
- Advantages:
  - Reduce the range of array indices handled
  - Storage is also reduced

Example: HASH TABLES



Example Hash Function and Hash Table

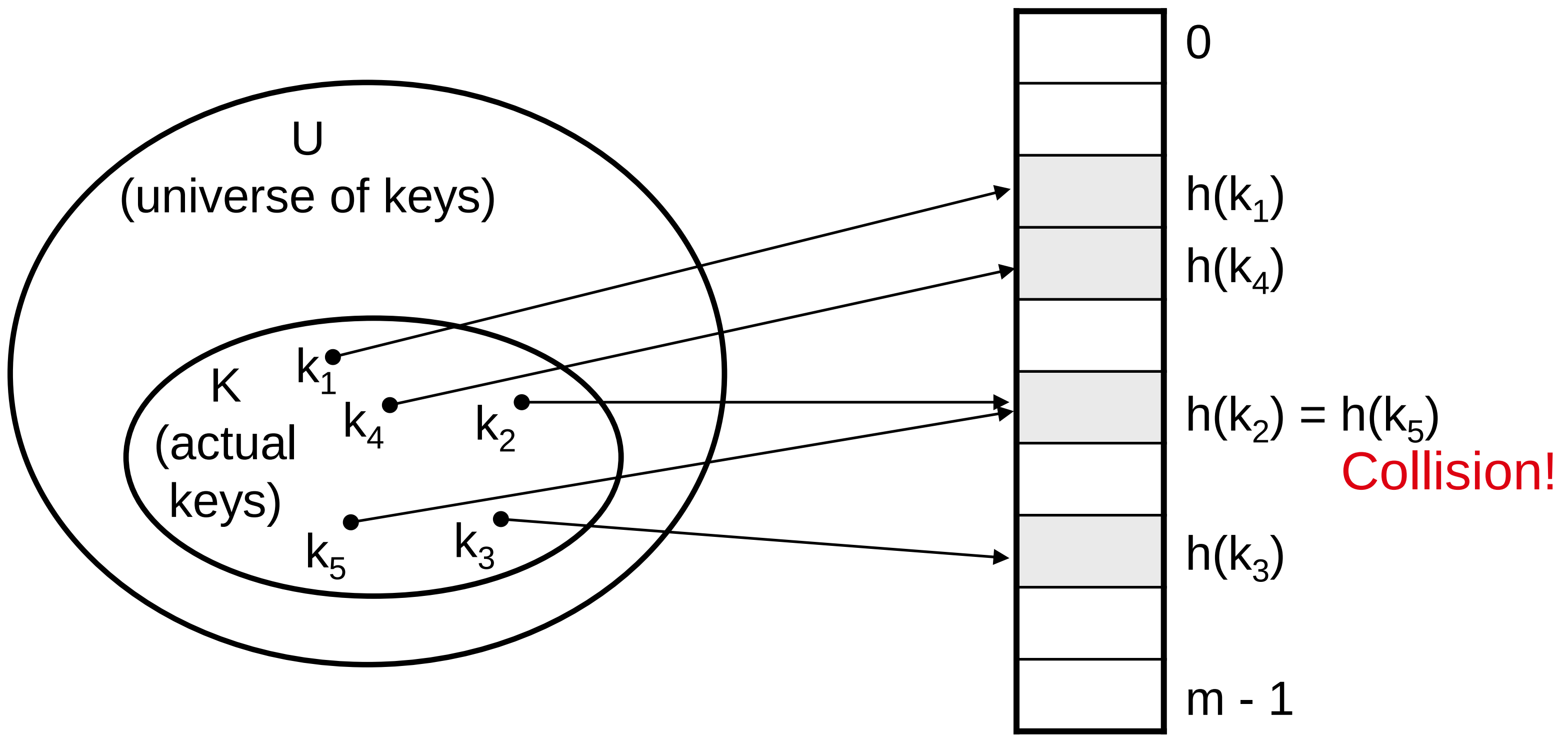


Hashing: Problems

Problems:

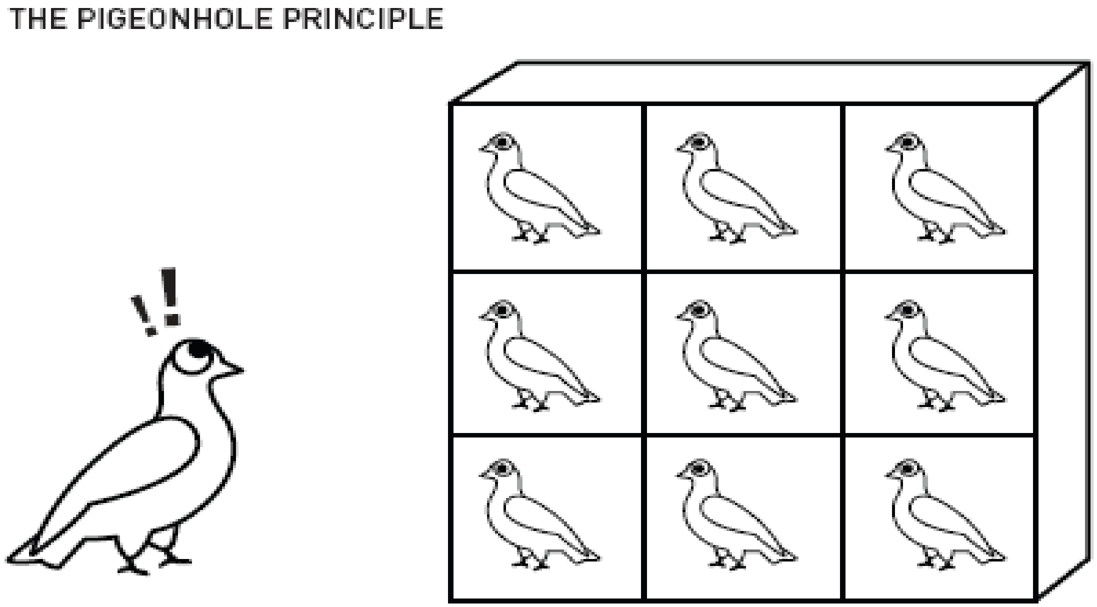
- Keys may not be numeric.
- Number of possible keys is much larger than the space available in the table.
- Different keys may map into the same location
  - Hash function is not one-to-one
  - In this case, the performance of hashing will suffer dramatically.

Do you see any problem ?



What is a Collision ?

- Two or more keys hash to the same slot : Collision
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function



Collisions

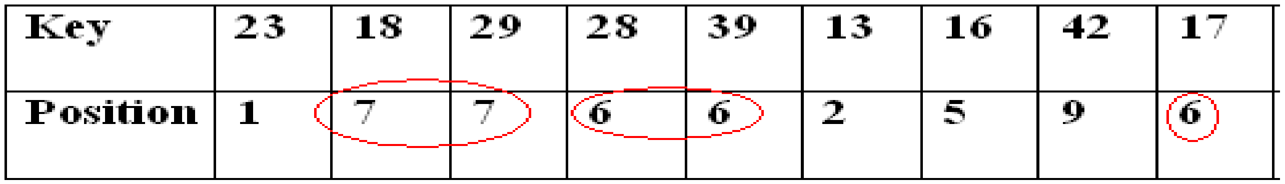
When the function assigns the same address to >1 records collisions occur.

Example:

- - Hash table size 11
- - Hash function: key mod hash size

23 mod 11 =1 , 18 mod 11= 7, 29 mod 11=7,…

So, the new positions in the hash table are:



Key	23	18	29	28	39	13	16	42	17
Position	1	7	7	6	6	2	5	9	6

collisions: positions 6, 7.

Hash Functions

- A hash function transforms a key into a table address
- What makes a good hash function?
  - (1) Easy to compute
  - (2) Approximates a random function: for every input, every output is equally likely :simple uniform hashing
- In practice, it is very hard to satisfy the simple uniform hashing property
  - – i.e., we don’t know in advance the probability distribution that keys are drawn from

Hash Functions

- If the input keys are integers then a general strategy is simply to use :

  Key mod *TableSize*

  - Unless key happens to have some undesirable properties. (e.g. all keys end in 0 and we use mod 10)
- If the keys are strings, hash function needs more care.
  - First convert the key into a numeric value, then apply hash function.

Some Hash Function Methods

- Division Method : Key mod m:
  - m is the size of the table, better if it is prime.
- Folding:
  - 123|456|789: add them and take mod.
- Truncation:
  - 123456789 map to a table of 1000 addresses by picking any 3 digits of the key.
- Mid-Square:
  - Square the key and then truncate to desired range

………………

- A good hash function is designed to distribute the keys roughly evenly into the available positions within the array (or hash table).

The Division Method

- Map a key k into one of the m slots by taking the remainder of k divided by m

  h(k) = k mod m

- This gives the indexes in the range 0 to m-1 so the hash table should be of size m

Example**:**Suppose we have a table of size 1000.

A good choice could be the nearest prime number *m*=1009.

If *k*=5127 then the corresponding hash value is

h(*k*)=5127 mod 1009=82   [5127= 5*1009 + 82]

Properties:

- Fast, requires only one operation but , certain values of m are bad. Ex : Powers of 2.

The Folding Method

- The key K is partitioned into a number of parts ,each of which has the same length as the required address with the possible exception of the last part .
- The parts are then added together , ignoring the final carry, to form an address.

Example: If key=356 942 781 is to be transformed into a three digit address.

P1=356, P2=942, P3=781

h(key)= (Last three digits (356+942+ 781))

=079.

The Mid- Square Method

- The key K is multiplied by itself and the address is obtained by selecting an appropriate number of digits from inside the square.
- The number of digits selected depends on the size of the table.
- Example: If key=123456 and m=1000:

$(123456)^2=1524\textcolor{red}{138}3936$

- Since a three-digit address is required, positions 5 to 7 could be chosen, giving the address 138.

Hashing a String Key

- Table size [0..99].Assign numerical values to the symbols that can be used to form a key :
  - A..Z ---> 1,2, ...26
  - 0..9 ----> 27,...36
- Key: CS1 → 3+19+28 (concat) = 31,928

  $(31{,}928)^2$ = 1019397184 - 10 digits
- Extract middle 2 digits (5th and 6th) as requested table size is 0..99.

  Get 39.

→ H(CS1) = 39.

Hashing a String Key

- Sum up ASCII (or Unicode) values for characters in string s
    - ASCII value for “A” =65,…,Z=90. sum will be in range 650-900 for 10 upper-case letters.

  EX: Hash code for string “abracadabra” is

    ASCII(“a”)+ ASCII(”b”)+…+ ASCII(”a”) = X

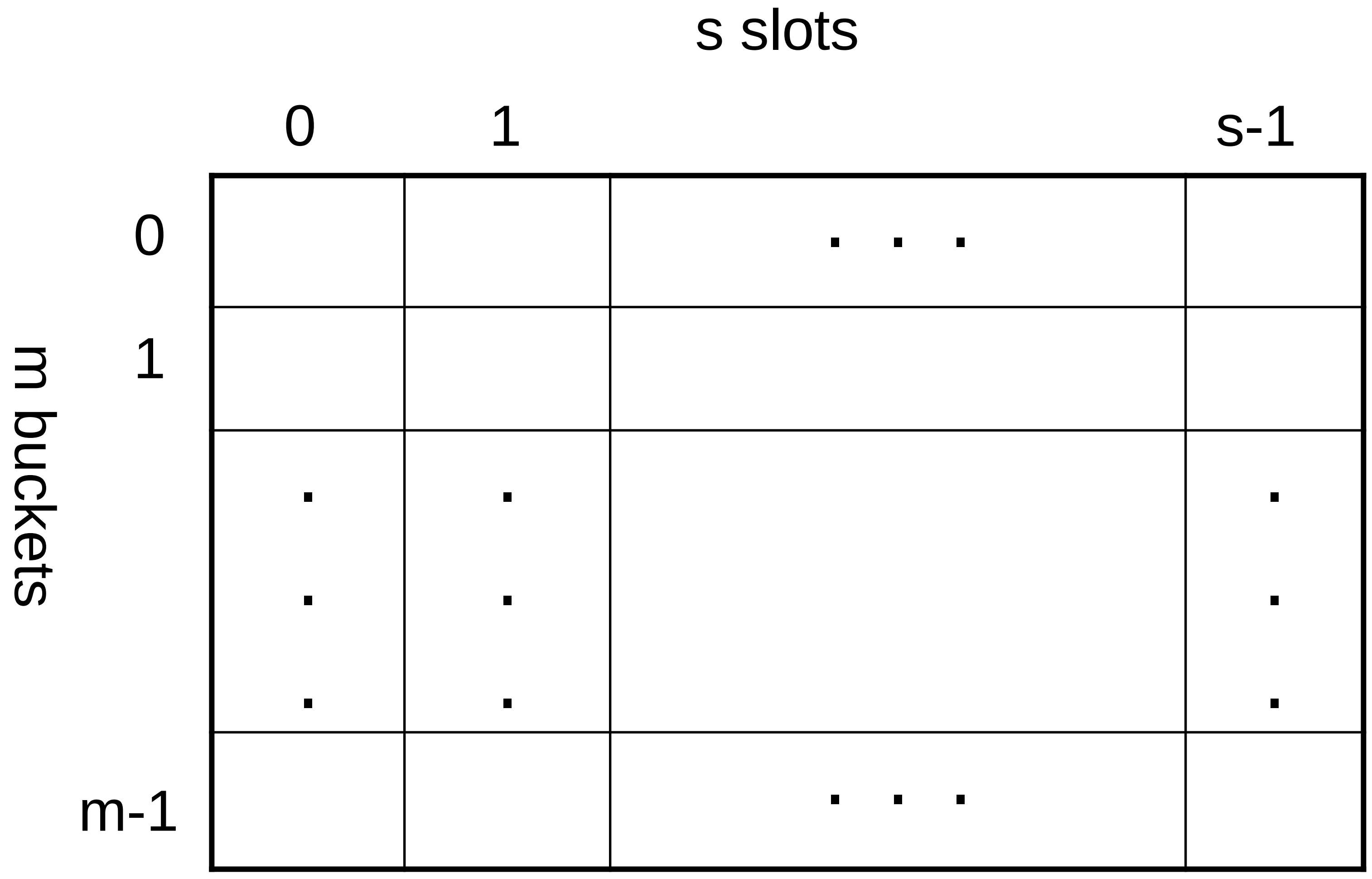
  →X can be used as a hash value or one of the previous methods can be used to convert X into the desired range.

Has the order of chars in string any effect ?

Static Hashing

- Key-value pairs are stored in a fixed size *hash table*.
  - A hash table T is partitioned into many buckets(Addresses)
  - Each bucket has many *slots*.
  - Each slot holds one record.
  - A hash function f(x) transforms the identifier (key) into an address in the hash table

The Structure of a Hash table



Static Hashing : Handling Collisions

- Collision: When a new record is hashed to an already full address or bucket,a collision occurs.

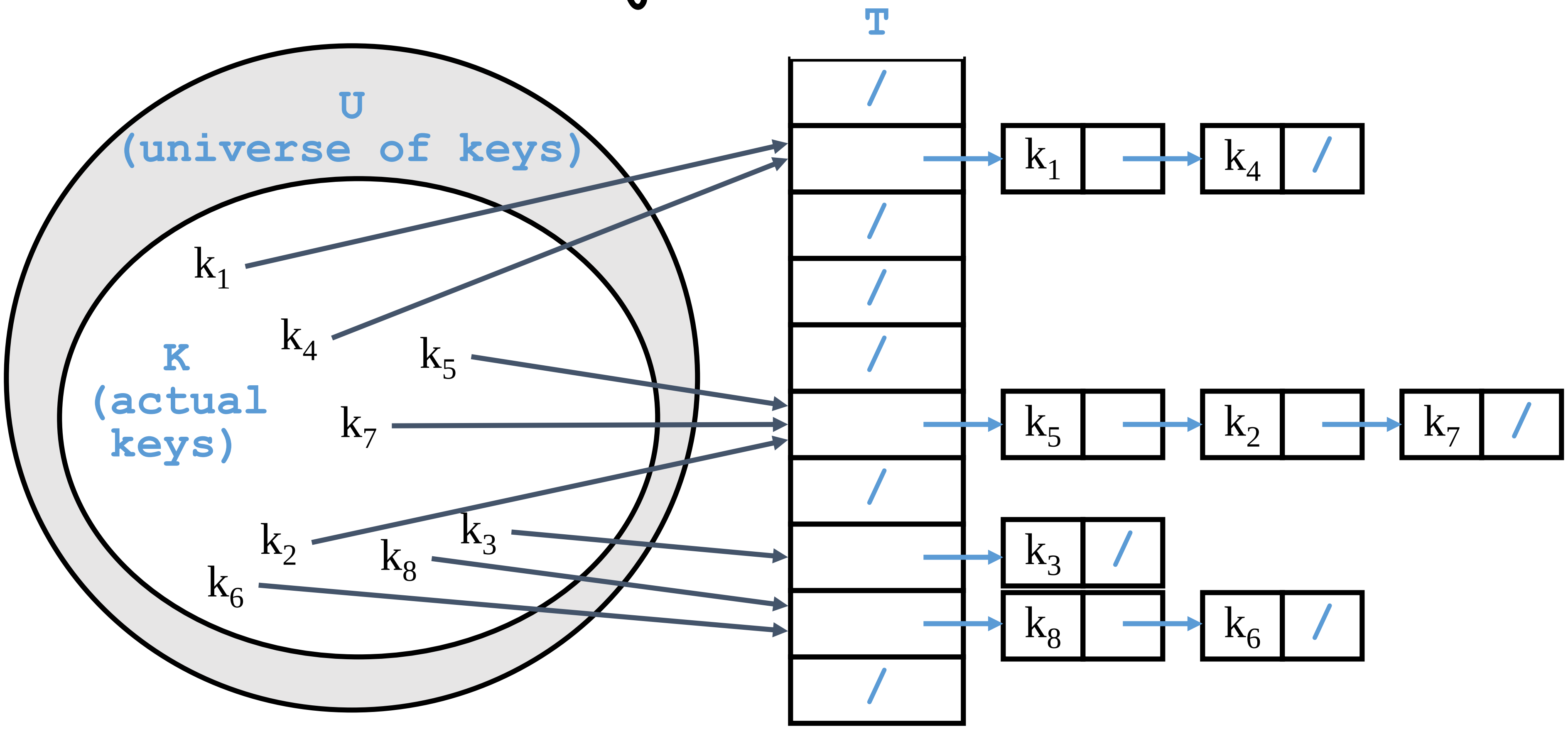
  →New record must be stored elsewhere.
- Finding the storage location for a colliding record is called collision resolution.
- Worst Case:The hash function maps all search-key values to the same bucket; this makes access time O(N).
- An ideal hash function should be *uniform*, i.e. each bucket is assigned ~same number of search-key values.

Handling Collisions

- There are several methods for dealing with collision problem:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Handling Collisions Using Chaining

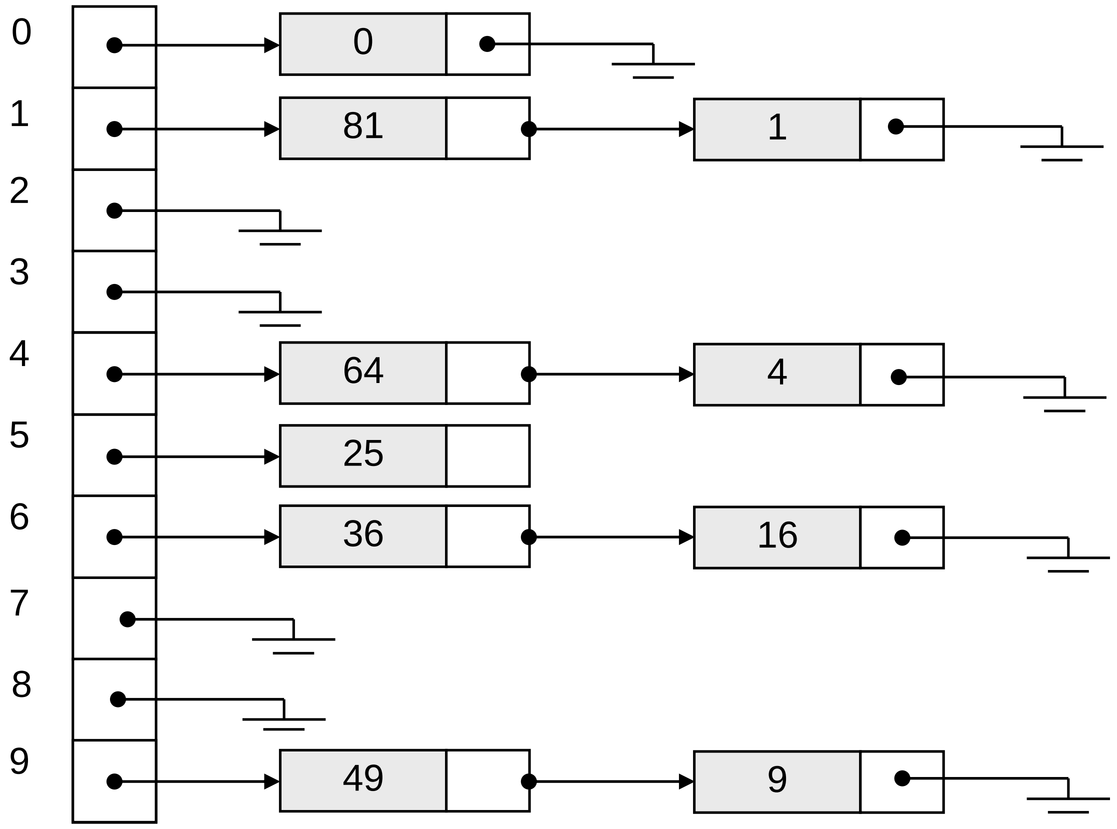
- Chaining puts elements that hash to the same slot in a linked list.
- Slot j contains a pointer to the head of the list of all elements that hash to j



Chaining : Example

Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81,m=10 addresses

hash(key) = key % 10.



Chaining : Table Properties

- Choosing the size of hash table:
  - Small enough not to waste space
  - Large enough such that collision lists remain short
- How should we keep the lists, ordered or not?
  - Not ordered, Why ?
    - Insertion is fast
    - Can easily delete the most recently inserted elements

Chaining: Hash Table Operations

- Initialization: All entries are set to NULL
- Insertion:
  - Locate the table cell(bucket), using hash function.
  - Insert the collided item to the front of the list.
- Find(Search):
  - locate the cell ,using hash function.
  - sequentially search the linked list for that cell.
- Deletion:
  - Locate the cell ,using hash function.
  - Delete the item from the linked list.

Chaining : Insertion in Hash Tables

T [ ] represents hash table array.

```
CHAINED-HASH-INSERT(T, x)
        //Insert x at the head of its list
          T[h(key[x])] ← x
```

- Worst-case complexity = O(1)
- Assumes that the element being inserted isn't already in the list
- It would take an additional search to check if it was already inserted

Chaining : Deletion in Hash Tables

```
CHAINED-HASH-DELETE(T, x)
        //Delete x from the head of its list
          T[h(key[x])] ← NIL
```

- Need to find the element to be deleted.
- Worst-case running time
    - Deletion time depends on searching the corresponding list

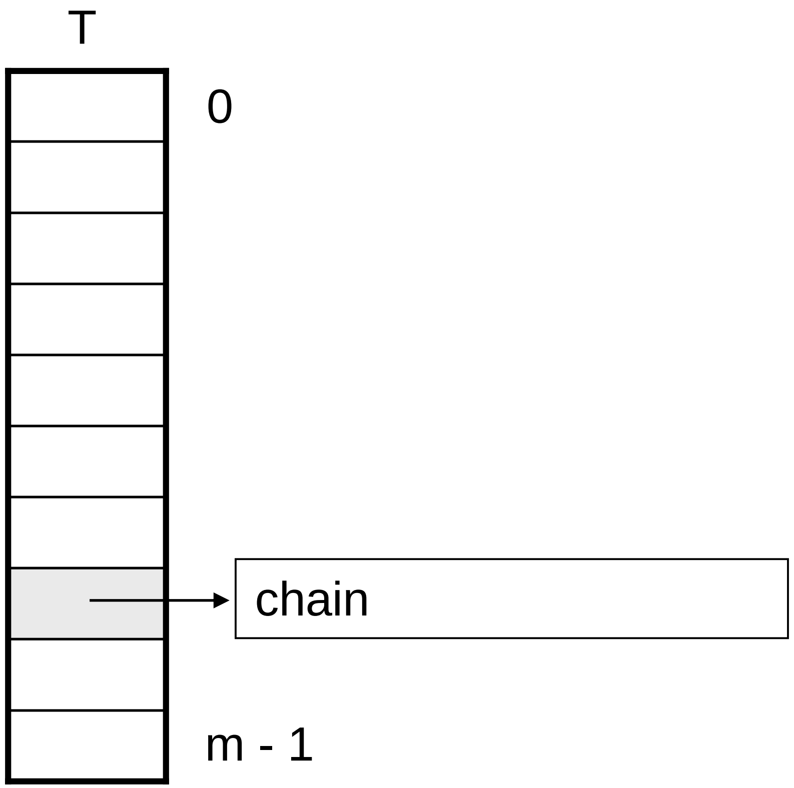
Chaining : Searching Hash Table

```
CHAINED-HASH-SEARCH(T, k)
    Find the key k in list T[h(k)]
  return the position
```

- Running time is proportional to the length of the list of elements in slot $h(k)$

Analysis of Hashing with Chaining:

- **Cost** :How long does it take to search for an element with a given key?
- Worst case:
  - All n keys hash to the same slot
  - Worst-case time to search is Θ(n), plus time to compute the hash function



Load Factor of a Hash Table

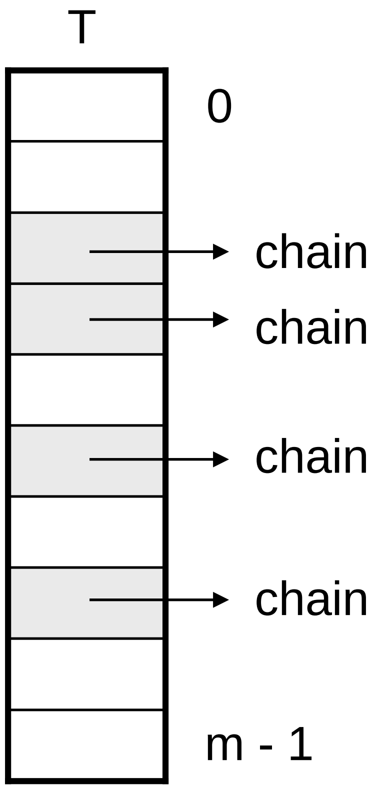
- Load factor of a hash table T:

  a = n/m

  - n = # of elements stored in the table
  - m = # of slots in the table

    = # of linked lists

- a:The average number of elements stored in a chain
- a can be <, =, > 1



Analysis of Chaining:Average Case

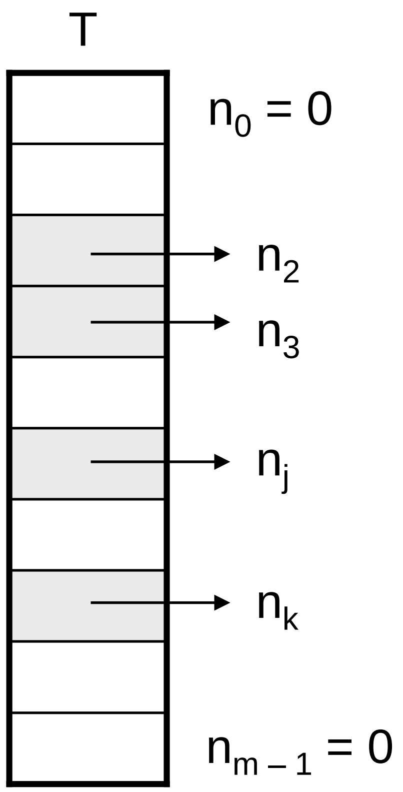
- Depends on how well the hash function distributes n keys among m slots
- Simple uniform hashing assumption:
  - Any given element is equally likely to hash into any of the m slots , i.e. probability of collision: Pr(h(x))=Pr(h(y))

    = 1/m
  - Length of a list:

    $T[j] = n_j, \quad j = 0, 1, \ldots, m-1$
  - Number of keys in the table:

    $n = n_0 + n_1 + \cdots + n_{m-1}$
  - Average (Expected)value of n_j:

    $E[n_j] = a = n/m$



Cost of searching

- Cost = Constant time to evaluate the hash function + time to traverse the list.

  Unsuccessful search:

  - We have to traverse the entire list, so we need to compare a nodes on the average:

    Worst case cost = 1+ a

    =O(a)

  Successful search:

  - List contains the one node that stores the searched item + 0 or more other nodes.
  - On the average, we need to check *half* of the *other nodes* while searching for a certain element

  →Average search cost = 1 + a/2

  = O(a)

Handling Collisions : Open Addressing

- Separate chaining has the disadvantage of using linked lists.
    - Requires the implementation of a second data structure.
- In an open addressing hashing system, all the data go inside the table.
    - Thus, a bigger table is needed.
    - If a collision occurs, alternative cells are tried until an empty cell is found.
- There are three common open addressing collision resolution strategies:
    - Linear Probing
    - Quadratic probing
    - Double hashing

Linear probing: Inserting a key

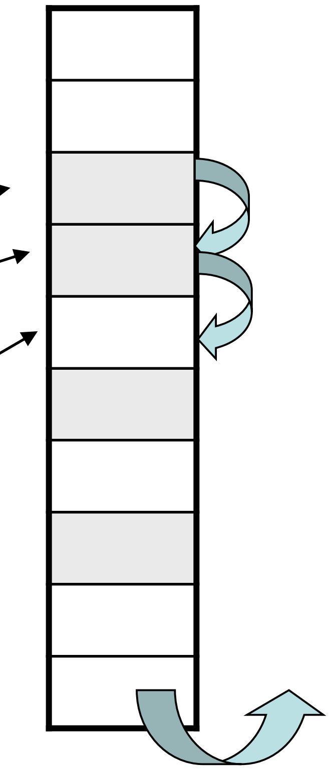
- Idea: when there is a collision, check the next available position in the table (i.e., probe ahead)

$$h(k,i) = (h_1(k) + i) \bmod m$$

$$i=0,1,2,\ldots$$

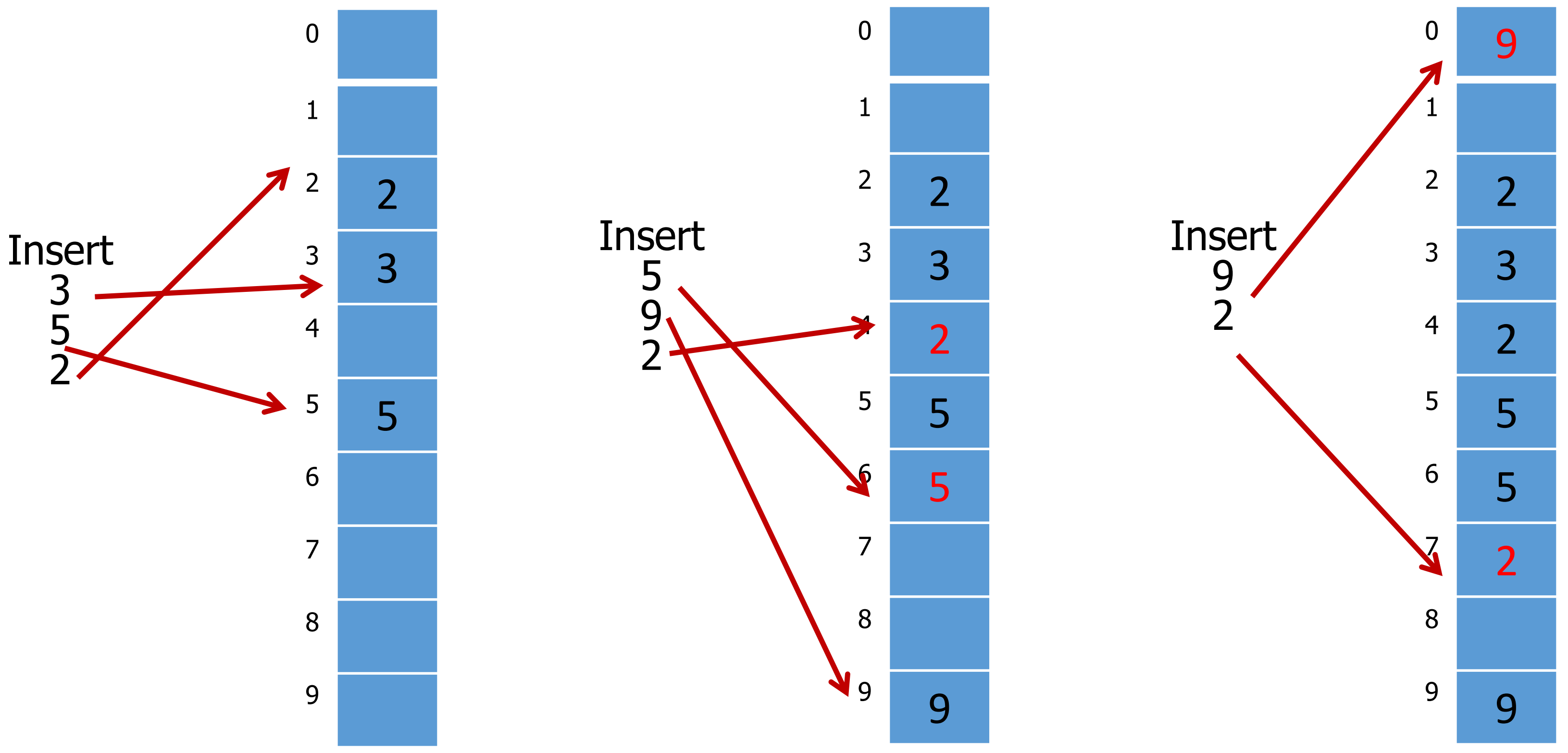
- First slot probed: $h_1(k)$
- Second slot probed: $h_1(k) + 1$
- Third slot probed: $h_1(k)+2$, and so on

probe sequence: [h1(k), h1(k)+1 , h1(k)+2 , ...]



Linear Probing: Example

Assume Table size m= 10. Hash function is hash(x) = x mod m.
Values to be inserted : 3,5,2,5,9,2,9,2



Collision resolution by linear probing

Linear Probing : Analysis

k: 20,30,35,13,25,24,10,9

- What is the average number of probes for a successful search for this hash table?
  - Hash Function: h(k) = k mod 11

Successful Search:

20: 9 , 30: 8 , 35 : 2 , 13: 2, 3 , 25:3,4

24: 2,3,4,5 , 10: 10 , 9: 9,10, 0

Average probe for Suscessful Search =

(1+1+1+2+2+4+1+3)/8=15/8

=1.88

0	9
1	
2	35
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

Linear Probing : Clustering Problem

- In linear probing we check the i-th position. If it is occupied, we check the i+1st position, next i+2nd, etc.
- Problem – Clustering occurs, that is, the used spaces tend to appear in groups which tends to grow and thus increase the search time to reach an open space.

(What may be the cause of clustering ?)

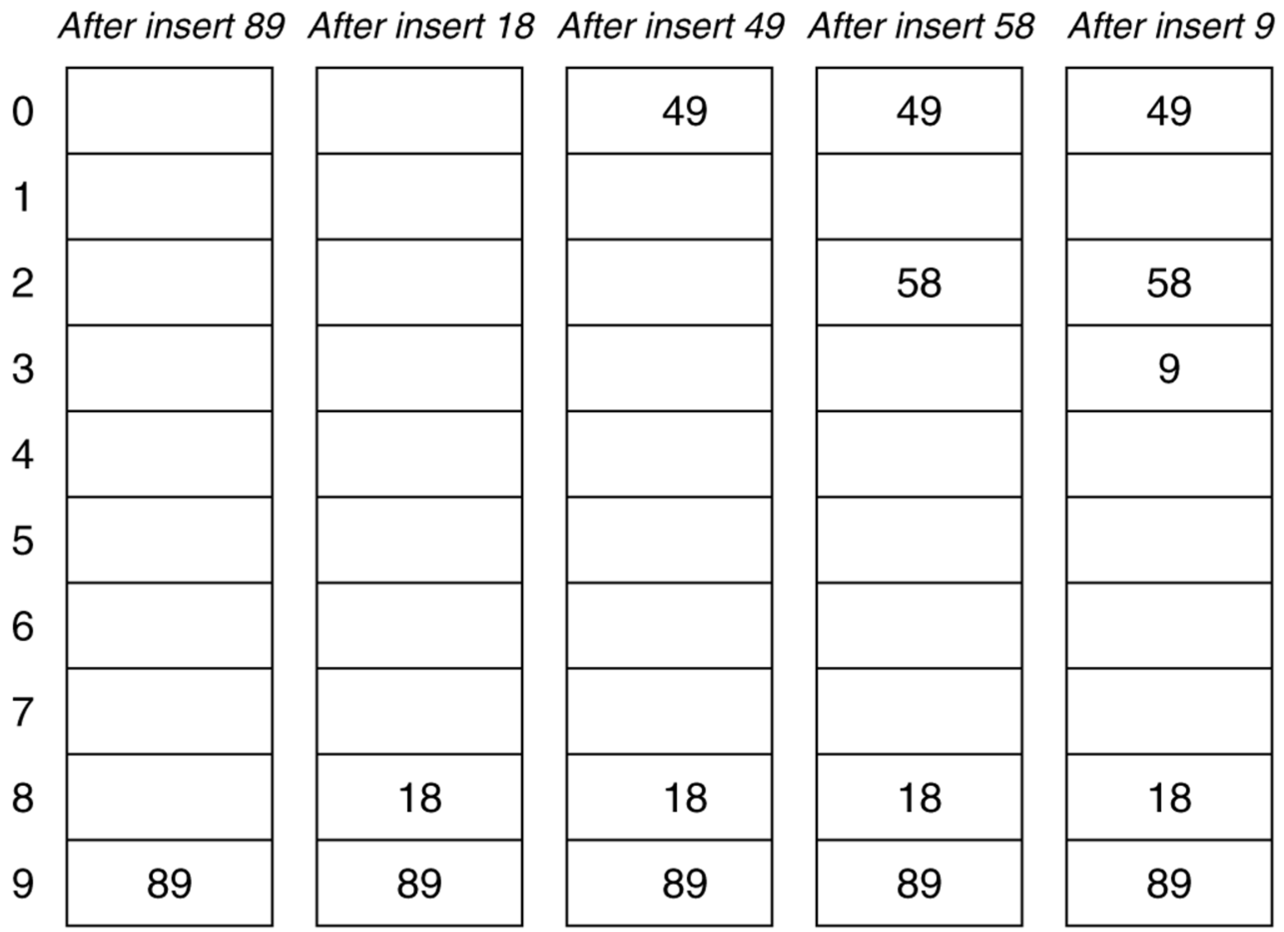
Quadratic Probing

- Quadratic Probing eliminates clustering problem of linear probing.
- Collision function is quadratic.
  - The popular choice is : $f(i) = i^2$.

  $\rightarrow$ i^{th} probe = $(h(k) + i^2) \bmod m$
- If the hash function evaluates to h and the cell h is occupied, we try cells $h + 1^2$, $h+2^2$, $h + 3^2$…
  - i.e. It examines cells 1,4,9 and so on away from the original probe.

A quadratic probing hash table after each insertion

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```



Problem of Quadratic Probing

- We may not be sure that we will probe all locations in the table.
- There is no guarantee to find an empty cell if table is more than half full.
- If the hash table size is not prime this problem will be much severe.

Handling Collisions with Rehashing

- Double Hashing is one of the best methods for dealing with collisions.
  - If the slot is full, then a second hash function is calculated and combined with the first hash function.
- Rehashing: If collision occurs; Try H_1, H_2, …, H_m in sequence. Here each H_i is a different hash function.

We are going to consider Double Hashing.

Double Hashing

Idea: Spread out the search for an empty slot by using a second hash function

1. Use one hash function to determine the first slot:

   Initial probe: $h_1(k)$

2. We apply a second hash function to k and probe at a distance: $h_2(k)$, $2*h_2(k)$, … and so on.

- Second probe is offset by $h_2(k)$ mod m, so on ...

  $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

  $i=0,1,\ldots$

- Disadvantage: harder to delete an element

Double Hashing Example

Insert keys : 76,93,40,47,10,55
Hash functions : $h_1(k)$ = k mod 7 , $h_2(k)$ = 5 – (k mod 5)

	76	93	40	47	10	55
0						
1				47	47	47
2		93	93	93	93	93
3					10	10
4						55
5			40	40	40	40
6	76	76	76	76	76	76
Probes	1	1	1	2	1	2

Problems of Collision Resolution

- Collision resolution is a costly process.With many collisions, hashing becomes inefficient.
- A good hash function should minimize collisions and provide a balanced distribution of records to addresses or buckets.
- But this is not guaranteed in many cases and frequent file reorganizations may be required.

Performance of Hashing

- A good hash function should be simple and should be computed quickly.Performance degrades if hash function is slow.
- If there is no collision, the time complexity is fixed:

  $T = O(1)$

  →Does not depend on the size!

  →Typically 1-2 attempts will be enough to locate the key value.

Static and Dynamic Hashing

- The hashing methods we have considered so far are "static" i.e. The maximum number of addresses or buckets are predefined.
- In static hashing, function *h* maps search-key values to this fixed set of bucket addresses.
  - If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If there are many deletions, space will be wasted.
- Dynamic hashing does not impose this restriction.The address space grows dynamically in response to increasing needs.

  (We shall not consider this topic here.)

Hashing Applications

- Compilers use hash tables to implement the *symbol table*.
- Game programs use hash tables to keep track of positions it has encountered (*transposition table*)
- A hash table can be used for on-line spelling checkers.If misspelling detection (rather than correction) is important, an entire dictionary can be hashed and words checked in constant time
- Hash functions can be used to quickly check for inequality.If two elements hash to different values they must be different.
- Criptography.
- File and database indexing and organization.