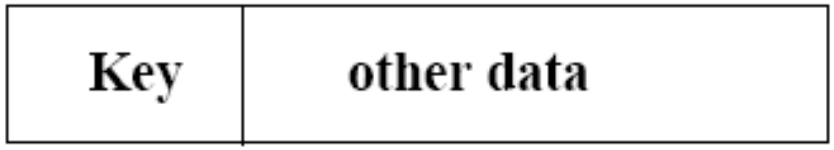
UNIT6

Hashing

The Search Problem

- Find items with keys matching a given search key
  - Given an array A, containing n keys, and a search key x, find the index i such as x=A[i]
  - As in the case of sorting, a key could be part of a large record.

example of a record



Searching Techniques-

In data structures, There are several searching techniques like linear search, binary search, search trees etc.

In these techniques, time taken to search any particular element depends on the total number of elements.

Example-

Linear Search takes O(n) time to perform the search in unsorted arrays consisting of n elements.

Binary Search takes O(logn) time to perform the search in sorted arrays consisting of n elements.

Drawback-

The main drawback of these techniques is-

As the number of elements increases, time taken to perform the search also increases.

This becomes problematic when total number of elements become too large.

Hashing in Data Structure-

In data structures,Hashing is a well-known technique to search any particular element among several elements.

It minimizes the number of comparisons while performing the search.

Advantage-

Unlike other searching techniques,

- Hashing is extremely efficient.
- The time taken by it to perform the search does not depend upon the total number of elements.
- It completes the search with constant time complexity O(1).

Hashing Mechanism-

In hashing, An array data structure called as **Hash table** is used to store the data items.

Based on the hash key value, data items are inserted into the hash table.

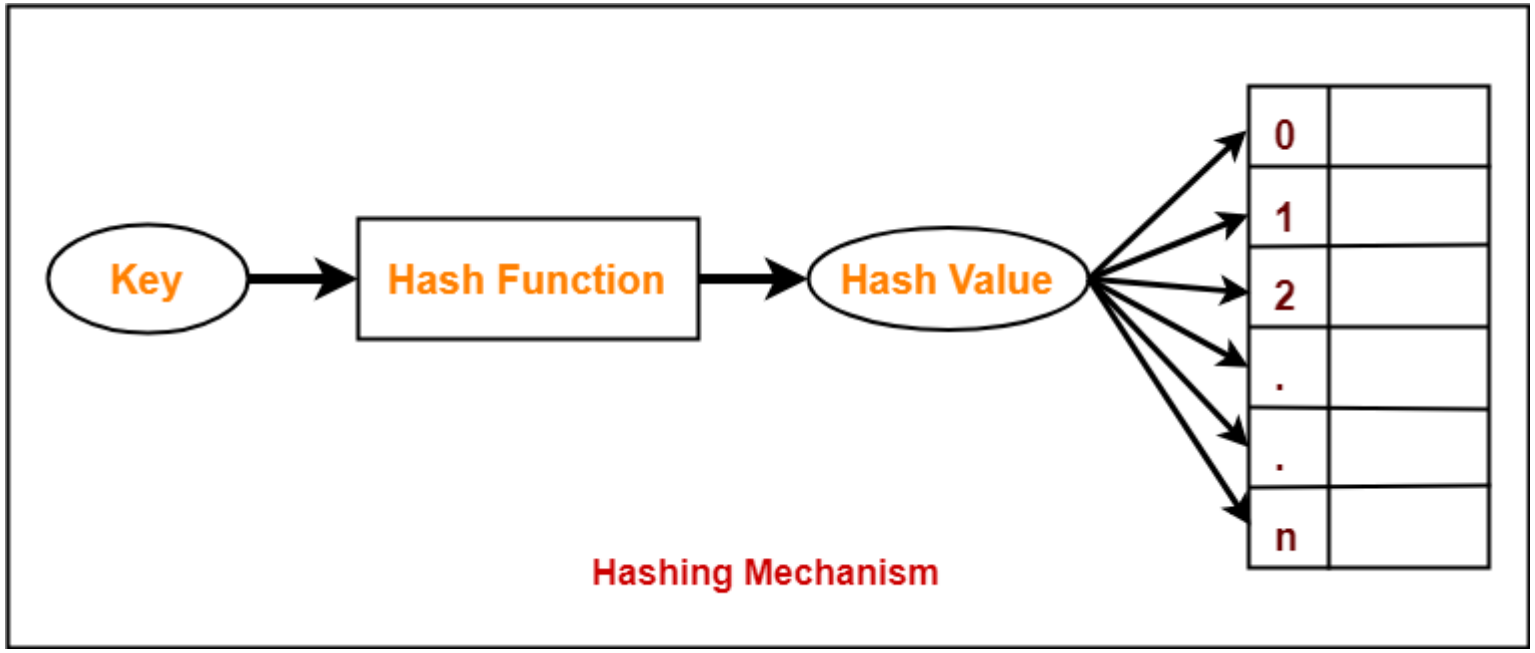
Hash Key Value-

Hash key value is a special value that serves as an index for a data item.

It indicates where the data item should be be stored in the hash table.

Hash key value is generated using a hash function.

HASHING MECHANISM



Hash Function-

Hash function takes the data item as an input and returns a small integer value as an output.
The small integer value is called as a hash value.
Hash value of the data item is then used as an index for storing it into the hash table.

Types of Hash Functions-

There are various types of hash functions available such as-

1. Division Hash Function
2. Mid Square Hash Function
3. Folding Hash Function
4. Multiplication method

It depends on the user which hash function he wants to use.

Properties of Hash Function-

It is efficiently computable.
It minimizes the number of collisions.
It distributes the keys uniformly over the table.

Hash function is a function that maps any big number or string to a small integer value.

Division Method

This is the easiest method to create a hash function. The hash function can be described as −

h(k) = k mod n

Here, h(k) is the hash value obtained by dividing the key value k by size of hash table n using the remainder.

It is best that n is a prime number as that makes sure the keys are distributed with more uniformity.

An example of the Division Method is as follows −

k=1276

n=10

h(1276) = 1276 mod 10

= 6

The hash value obtained is 6

A disadvantage of the division method is that consecutive keys map to consecutive hash values in the hash table. This leads to a poor performance.

Mid Square Method

The mid square method is a very good hash function. It involves squaring the value of the key and then extracting the middle r digits as the hash value. The value of r can be decided according to the size of the hash table.

An example of the Mid Square Method is as follows −

Suppose the hash table has 100 memory locations. So r=2 because two digits are required to map the key to memory location.

k = 50

k*k = 2500

h(50) = 50

The hash value obtained is 50

Folding method

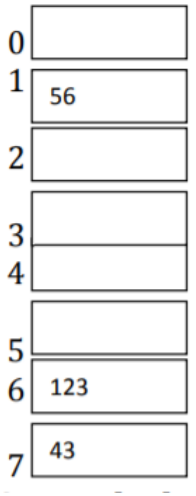
Folding Method : Partition the key K into number of parts, like K1,K2,.....Kn, then add the parts together and ignore the carry and use it as the hash value.

Example1:

H(123)= [1+2+3 =6]

H(43)= [4+3 = 7]

H(56)= [5+6 = 11]



Example2:

The task is to fold the key 123456789 into a Hash Table of ten spaces (0 through 9).

It is given that the key, say X is 123456789 and the table size (i.e., M = 10).

Since it can break X into three parts in any order. Let's divide it evenly.

Therefore, a = 123, b = 456, c = 789.

Now, H(x) = (a + b + c) mod M i.e., H(123456789) =(123 + 456 + 789) mod 10 = 1368 mod 10 = 8.

Hence, 123456789 is inserted into the table at address 8.

Multiplication Method

The hash function used for the multiplication method is −

$h(k) = floor(n (kA \bmod 1))$

Here, k is the key and A can be any constant value between 0 and 1. Both k and A are multiplied and their fractional part is separated. This is then multiplied with n to get the hash value.

An example of the Multiplication Method is as follows −

k=123

n=100

A=0.618033

h(123) = 100 (123 * 0.618033 mod 1)

= 100 (76.018059 mod 1)

= 100 (0.018059)

= 1

The hash value obtained is 1

An advantage of the multiplication method is that it can work with any value of A, although some values are believed to be better than others.

Hash Tables

A hash table is a data structure that maps keys to values. It uses a hash function to calculate the index for the data key and the key is stored in the index.

An example of a hash table is as follows −

The key sequence that needs to be stored in the hash table is −

35 50 11 79 76 85

The hash function h(k)

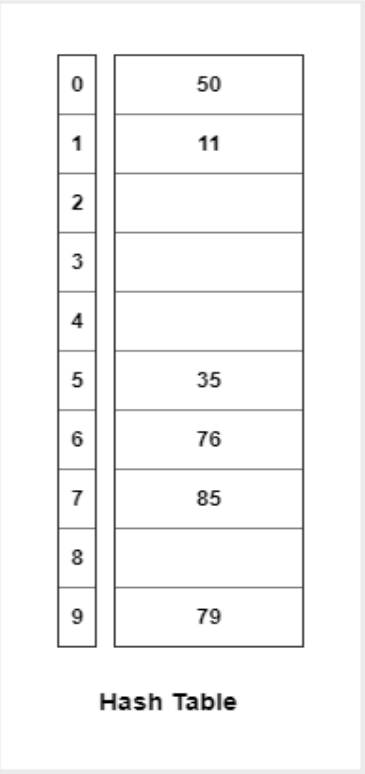
h(k) = k mod 10

Using linear probing,

the values are

stored in the

hash table as −



Applications

- Keeping track of customer account information at a bank
  - Search through records to check balances and perform transactions
- Keep track of reservations on flights
  - Search to find empty seats, cancel/modify reservations
- Search engine
  - Looks for all documents containing a given word

Special Case: Dictionaries

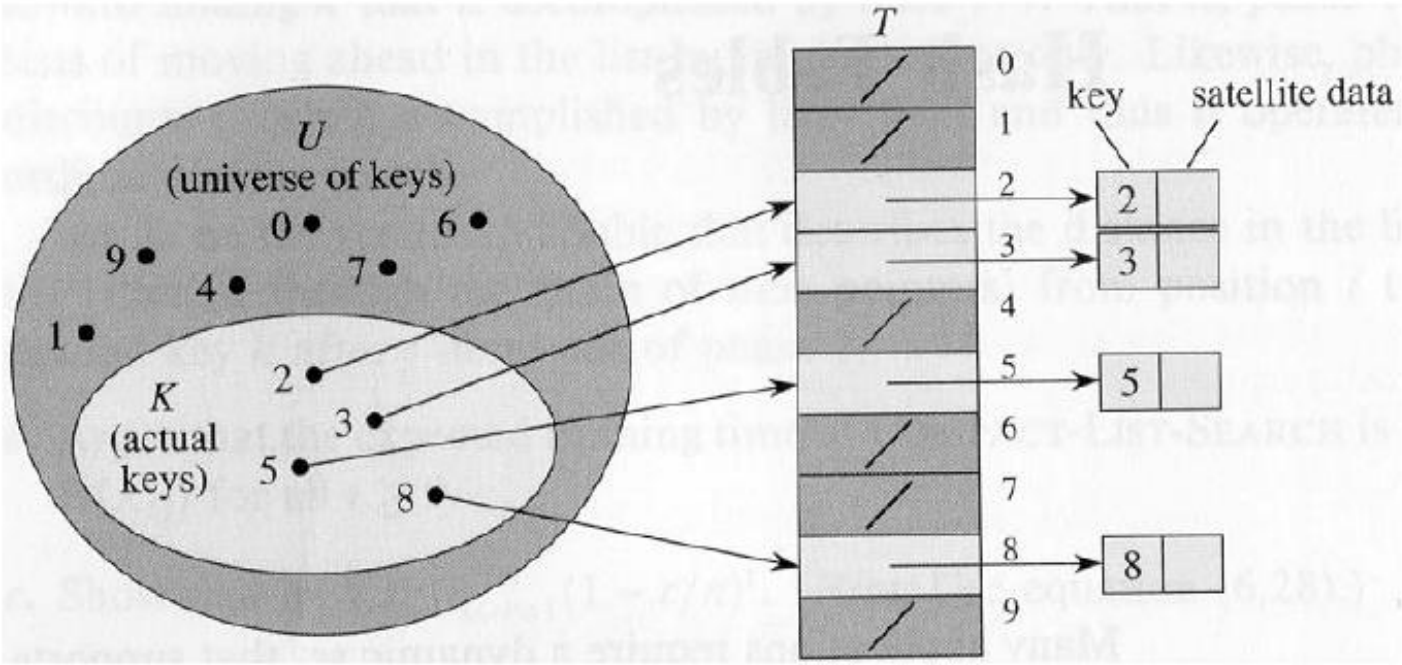
- **Dictionary** = data structure that supports mainly two basic operations: insert a new item and return an item with a given key
- Queries: return information about the set S:
  - Search (S, k)
  - Minimum (S), Maximum (S)
  - Successor (S, x), Predecessor (S, x)
- Modifying operations: change the set
  - Insert (S, k)
  - Delete (S, k) – not very often

Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe $U = \{0, 1, \ldots, m - 1\}$
- Idea:
  - Store the items in an array, indexed by keys

- **Direct-address table** representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented by NIL

Direct Addressing (cont'd)



(insert/delete in O(1) time)

Operations

Alg.: DIRECT-ADDRESS-SEARCH(T, k)

return T[k]

Alg.: DIRECT-ADDRESS-INSERT(T, x)

T[key[x]] ← x

Alg.: DIRECT-ADDRESS-DELETE(T, x)

T[key[x]] ← NIL

- Running time for these operations: $O(1)$

Comparing Different Implementations

- Implementing dictionaries using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered/unordered linked lists

	Insert	Search
direct addressing	O(1)	O(1)
ordered array	O(N)	O(lgN)
ordered list	O(N)	O(N)
unordered array	O(1)	O(N)
unordered list	O(1)	O(N)

Examples Using Direct Addressing

Example 1:

(i) Suppose that the keys are integers from 1 to 100 and that there are about 100 records

(ii) Create an array A of 100 items and store the record whose key is equal to i in $A[i]$

Example 2:

(i) Suppose that the keys are nine-digit social security numbers

(ii) We can use the same strategy as before but it very inefficient now: an array of 1 billion items is needed to store 100 records !!

- $|U|$ can be very large
- $|K|$ can be much smaller than $|U|$

Hash Tables

- When K is much smaller than U, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to $|K|$
  - Can still get $O(1)$ search time, but on the average case, not the worst case

Hash Tables

Idea:

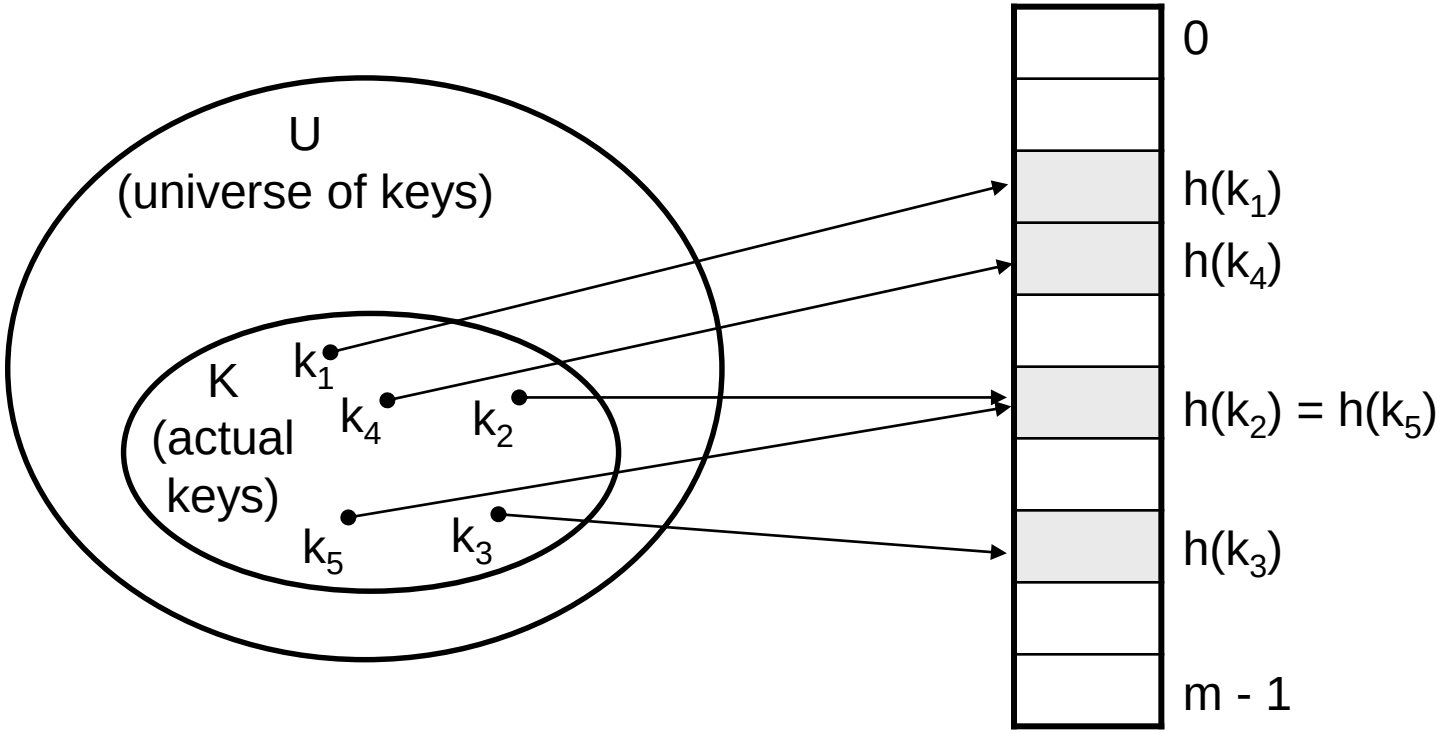
- Use a function h to compute the slot for each key
- Store the element in slot $h(k)$

- A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

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

- We say that k **hashes** to slot $h(k)$
- Advantages:
  - Reduce the range of array indices handled: m instead of $|U|$
  - Storage is also reduced

Example: HASH TABLES



Revisit Example 2

Suppose that the keys are nine-digit social security numbers

Possible hash function

$h(ssn) = sss \bmod 100$ (last 2 digits of ssn)

e.g., if $ssn = 10123411$ then $h(10123411) = 11$)

Hashing exercise example

Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. Hash table is of size 11.apply the division method and the mid-square method (with division)and find hash value associated with above set of integers.

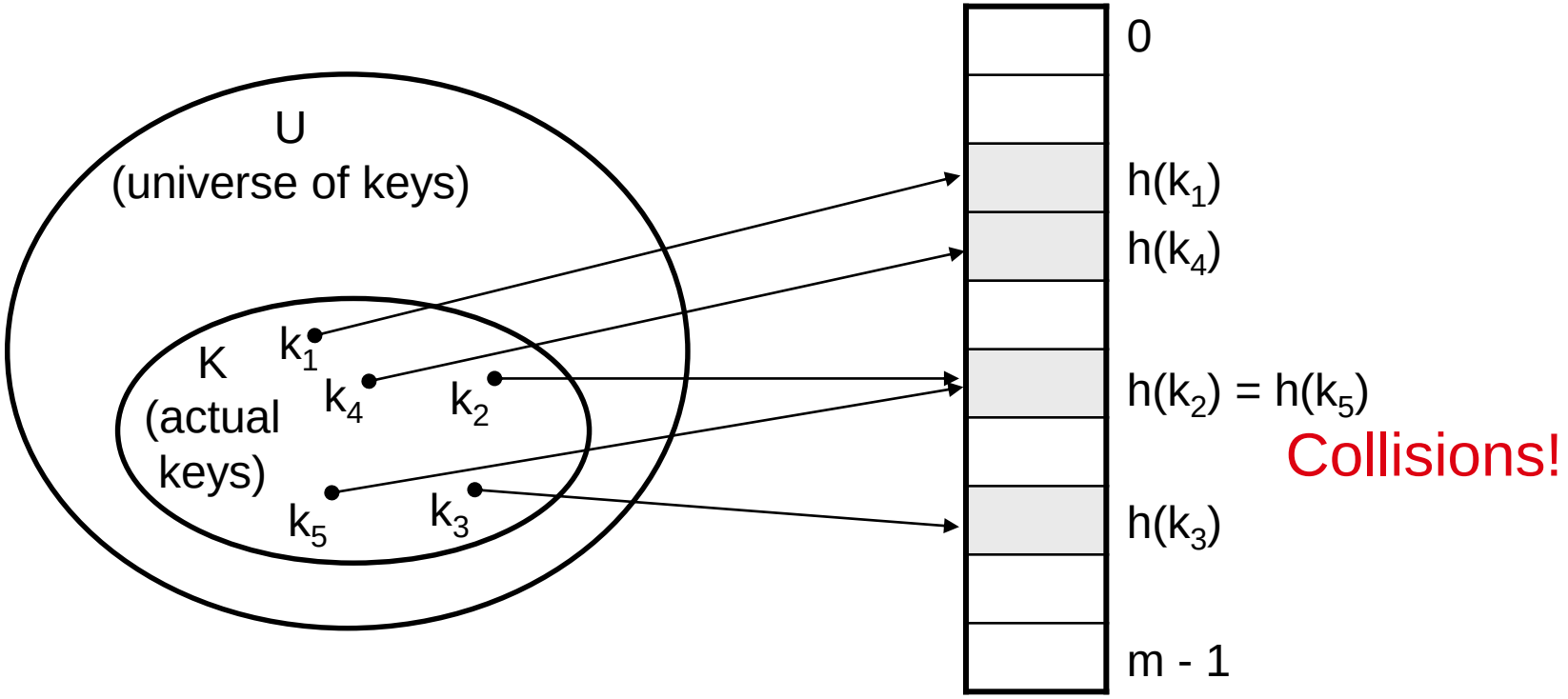
(In the mid-square method,first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute 442=1,936. By extracting the middle two digits, 93, and performing the remainder step, we get 5 (93 % 11))

Solution

Table 5: Comparison of Remainder and Mid-Square Methods

Item	Remainder	Mid-Square
54	10	3
26	4	7
93	5	9
17	6	8
77	0	4
31	9	6

Do you see any problems with this approach?



Collisions

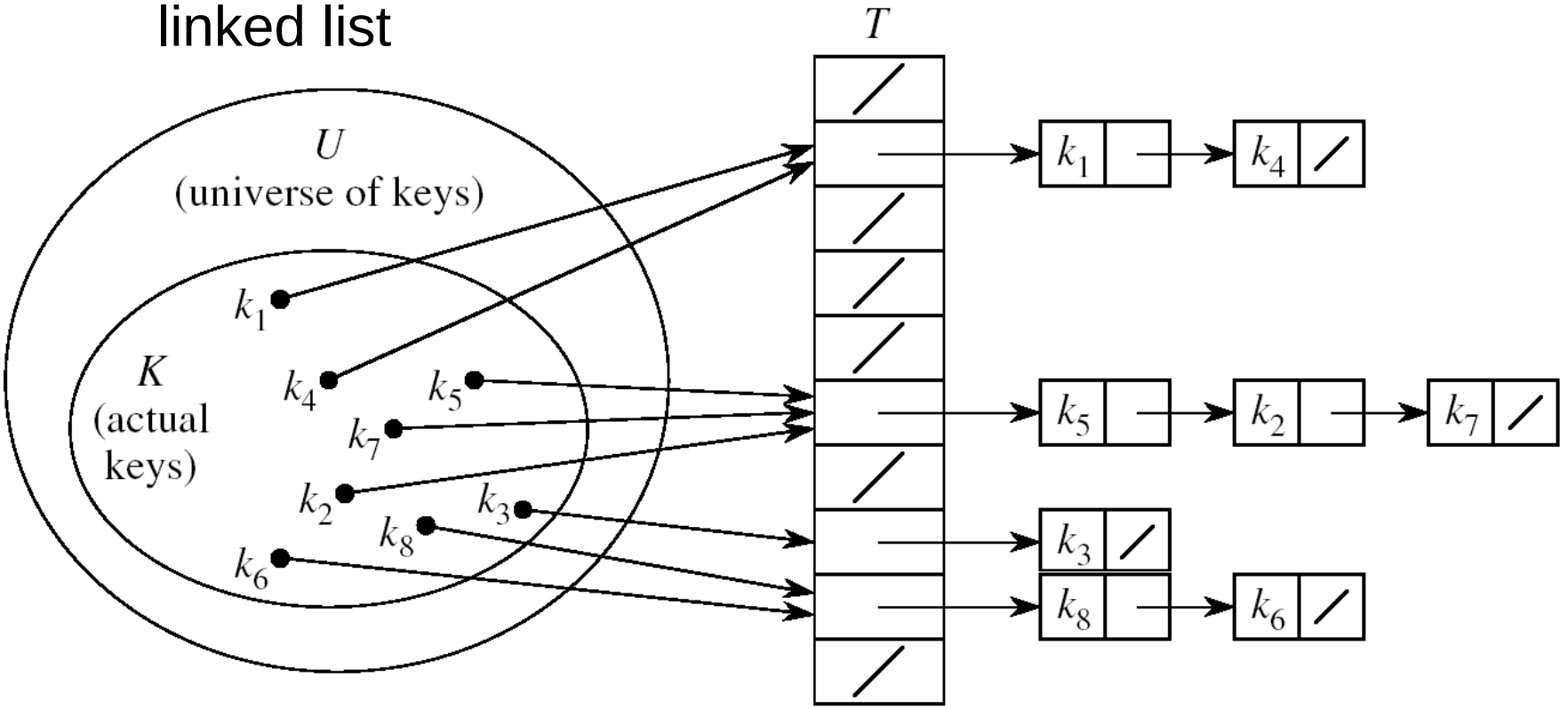
- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions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

Handling Collisions

- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Handling Collisions Using Chaining

- **Idea**:
  - Put all elements that hash to the same slot into a linked list



  - Slot j contains a pointer to the head of the list of all elements that hash to j

Collision with Chaining - Discussion

- Choosing the size of the table
  - Small enough not to waste space
  - Large enough such that lists remain short
  - Typically 1/5 or 1/10 of the total number of elements
- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Insertion in Hash Tables

Alg.: CHAINED-HASH-INSERT(T, x)

insert x at the head of list T[h(key[x])]

- Worst-case running time is $O(1)$
- 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

Deletion in Hash Tables

Alg.: CHAINED-HASH-DELETE(T, x)

delete x from the list T[h(key[x])]

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

Searching in Hash Tables

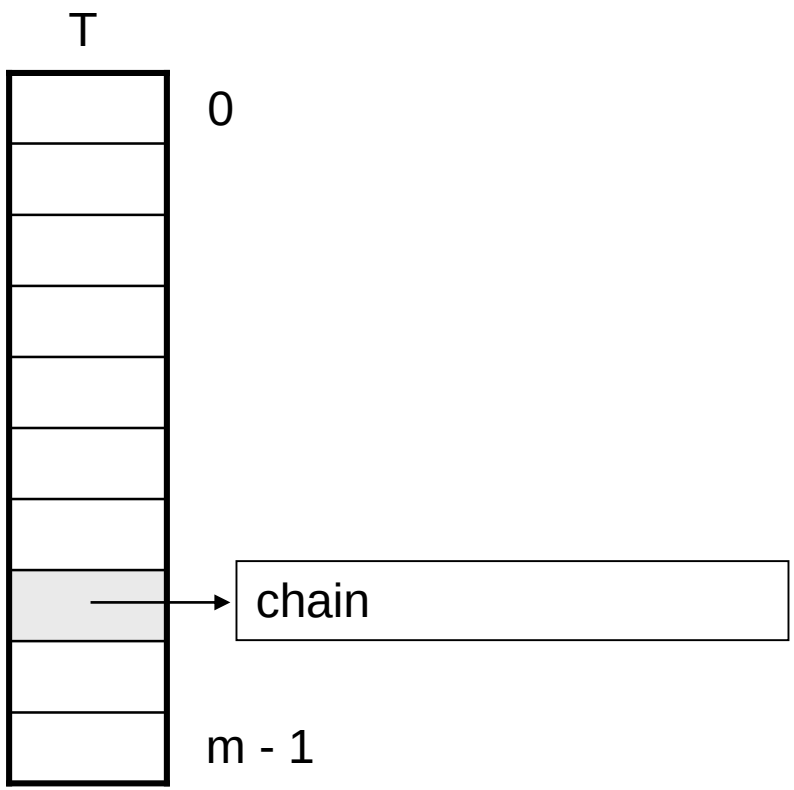
Alg.: CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list T[h(k)]

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

Analysis of Hashing with Chaining: Worst Case

- 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 $\Theta(n)$, plus time to compute the hash function



Analysis of Hashing with Chaining: Average Case

- Average case
  - depends on how well the hash function distributes the n keys among the 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)=h(y)), is 1/m)
- Length of a list:

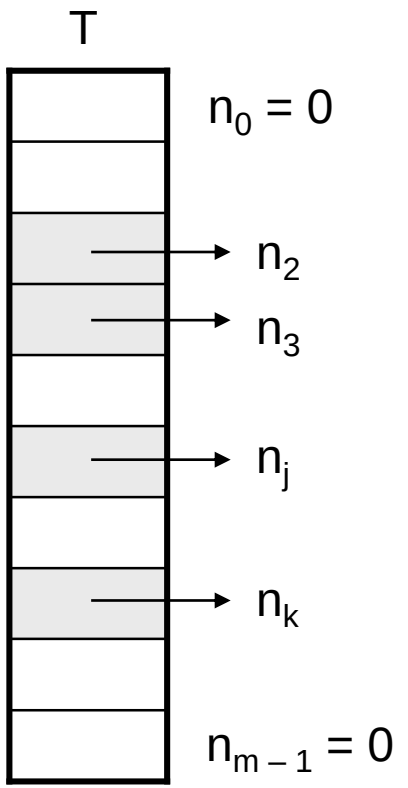
$$T[j] = n_j, \qquad j = 0, 1, \ldots, m - 1$$

- Number of keys in the table:

$$n = n_0 + n_1 + \cdots + n_{m-1}$$

- Average value of n_j:

$$E[n_j] = \alpha = n/m$$

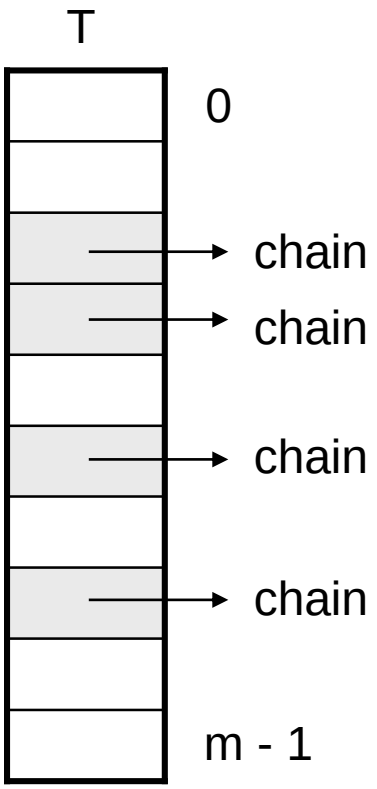


Load Factor of a Hash Table

- Load factor of a hash table T:

$$\alpha = n/m$$

  - n = # of elements stored in the table
  - m = # of slots in the table = # of linked lists
- α encodes the average number of elements stored in a chain
- α can be <, =, > 1



Case 1: Unsuccessful Search (i.e., item not stored in the table)

Theorem

An unsuccessful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing

(i.e., probability of collision Pr(h(x)=h(y)), is 1/m)

Proof

- Searching unsuccessfully for any key k
  - need to search to the end of the list T[h(k)]
- Expected length of the list:
  - $E[n_{h(k)}] = \alpha = n/m$
- Expected number of elements examined in an unsuccessful search is α
- Total time required is:
  - O(1) (for computing the hash function) + α → $\Theta(1+\alpha)$

Case 2: Successful Search

Successful search: $\Theta(1 + \frac{a}{2}) = \Theta(1 + a)$ time on the average

(search half of a list of length a plus $O(1)$ time to compute $h(k)$)

Analysis of Search in Hash Tables

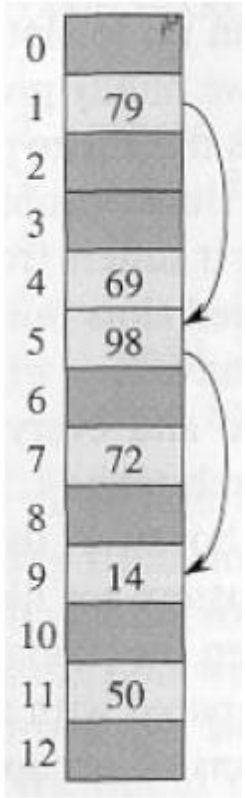
- If m (# of slots) is proportional to n (# of elements in the table):
- $n = O(m)$
- $\alpha = n/m = O(m)/m = O(1)$

$\Rightarrow$ Searching takes constant time on average

Open Addressing

- If we have enough contiguous memory to store all the keys (m > N) $\Rightarrow$ store the keys in the table itself
- No need to use linked lists anymore
- Basic idea:
  - Insertion: if a slot is full, try another one, until you find an empty one
  - Search: follow the same sequence of probes
  - Deletion: more difficult
- Search time depends on the length of the probe sequence!

e.g., insert 14



Generalize hash function notation:

- A hash function contains two arguments now:

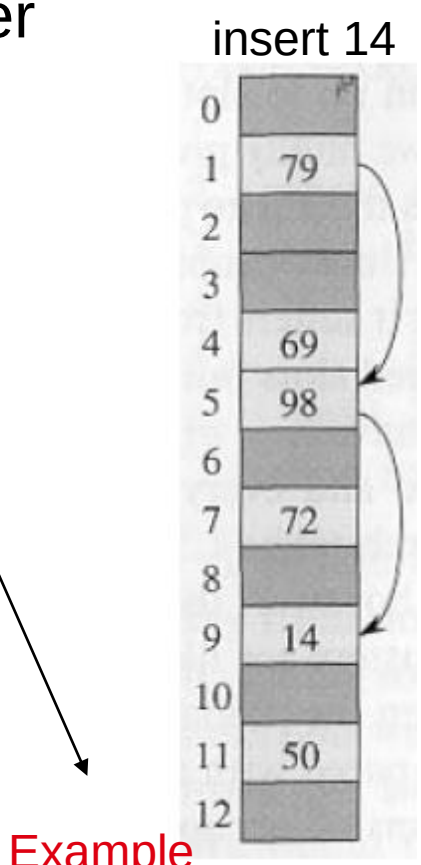
  (i) Key value, and (ii) Probe number

  $h(k,p), \quad p=0,1,\ldots,m-1$

- Probe sequences

  $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$

  - Must be a permutation of $\langle 0,1,\ldots,m-1 \rangle$
  - There are $m!$ possible permutations
  - Good hash functions should be able to produce all $m!$ probe sequences



Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

- Note: None of these methods can generate more than m^2 different probing sequences!

Linear probing: Inserting a key

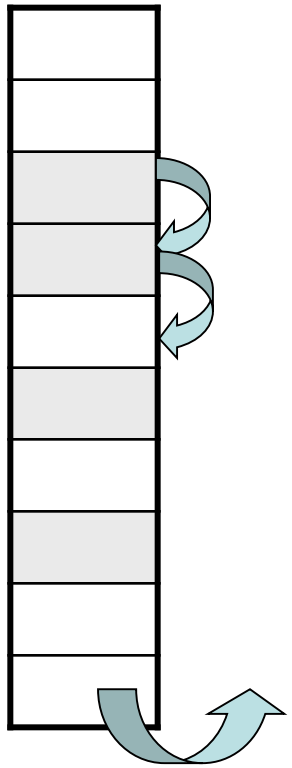
- Idea: when there is a collision, check the next available position in the table (i.e., probing)

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

$$i=0,1,2,...$$

- 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 , ...>

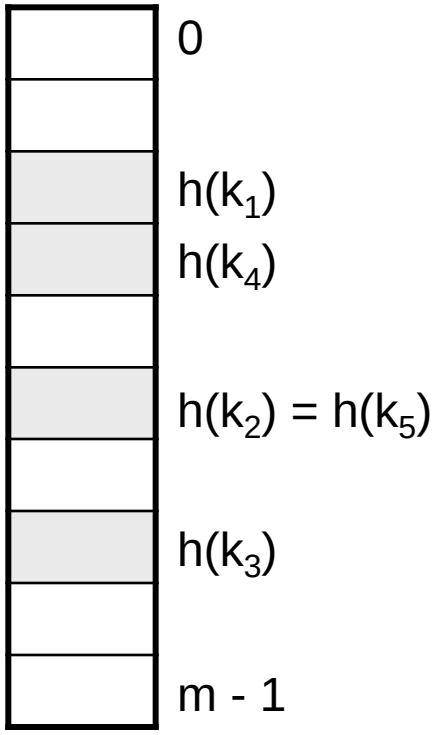


- Can generate m probe sequences maximum, why?

wrap around

Linear probing: Searching for a key

- Three cases:
  - (1) Position in table is occupied with an element of equal key
  - (2) Position in table is empty
  - (3) Position in table occupied with a different element
- Case 2: probe the next higher index until the element is found or an empty position is found
- The process wraps around to the beginning of the table

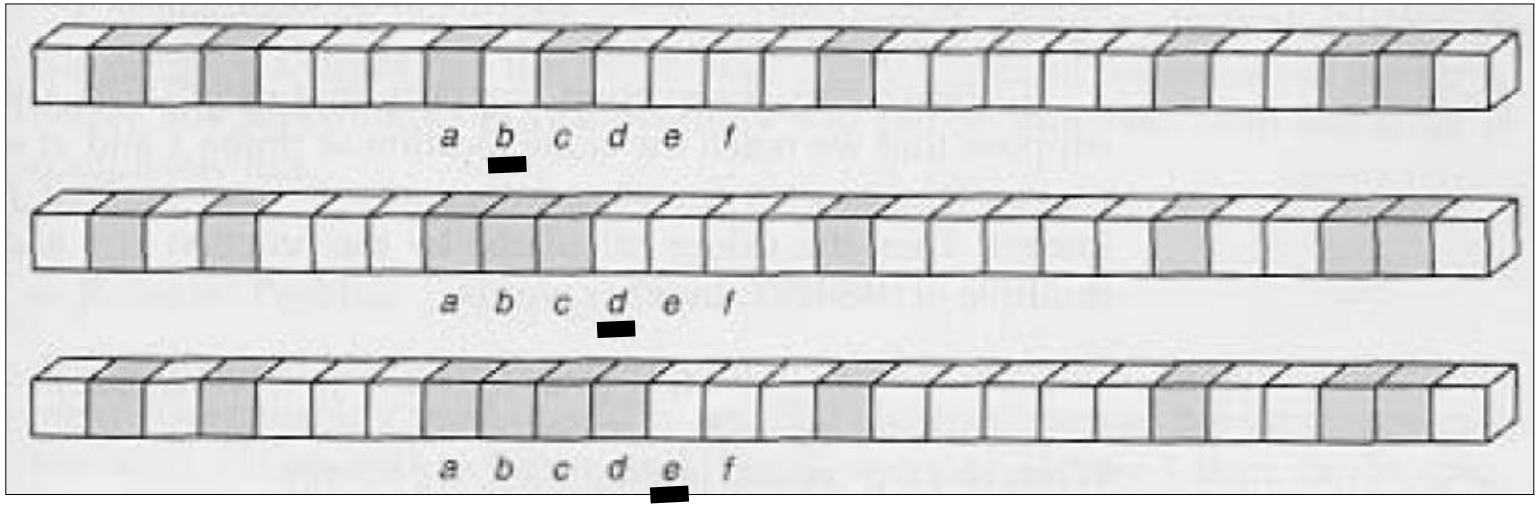


Primary Clustering Problem

- Some slots become more likely than others
- Long chunks of occupied slots are created

$\Rightarrow$ search time increases!!

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

Quadratic probing

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where $h' : U --> (0, 1, . . , m - 1)$

- Clustering problem is less serious but still an issue (*secondary clustering*)

- How many probe sequences quadratic probing generate ? m

(the initial probe position determines the probe sequence)

Suppose a record with key K has Hash Address H(k)=h, then instead of searching the location with addresses h, h+1, h+2, we linearly search locations with addresses h, h+1, h+4, h+9,h+16……

Double Hashing

(1) Use one hash function to determine the first slot

(2) Use a second hash function to determine the increment for the probe sequence

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,...$$

- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on ...
- Advantage: avoids clustering
- Disadvantage: harder to delete an element
- Can generate m^2 probe sequences maximum

Suppose a record with key K has Hash Address H(k)=h, and H'(k)= h'. then we linearly search locations with addresses h, h+h', h+2h', h+3h',h+4h'……. If m is a prime number, then above sequence will access all the locations in table T.

Double Hashing: Example

$h_1(k) = k \bmod 13$

$h_2(k) = 1 + (k \bmod 11)$

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

Insert key 14:

$h_1(14,0) = 14 \bmod 13 = 1$

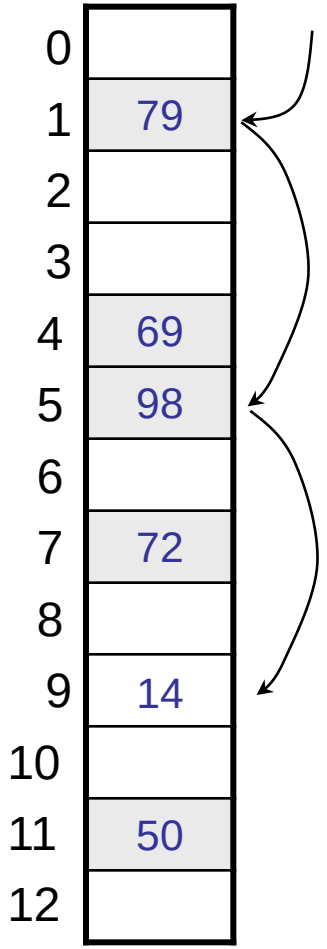
$h(14,1) = (h_1(14) + h_2(14))$

$= (1 + 4)$

$= 5$

$h(14,2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) = 9$



- Given a hash table with m=11 entries and the following hash function h1 and step function h2: h1(key) = key mod m h2(key) = {key mod (m-1)} + 1
- Insert the keys {22, 1, 13, 11, 24, 33, 18, 42, 31} in the given order (from left to right) to the hash table using each of the following hash methods:
- a. Chaining with h1 -> h(k) = h1(k)
- b. Linear-Probing with h1 -> h(k,i) = (h1(k)+i) mod m
- c. Double-Hashing with h1 as the hash function and h2 as the step function h(k,i) = (h1(k) + ih2(k)) mod m

	Chaining	Linear Probing	Double Hashing
0	33 → 11→ 22	22	22
1	1	1	1
2	24 → 13	13	13
3		11	
4		24	11
5		33	18
6			31
7	18	18	24
8			33
9	31→ 42	42	42
10		31	