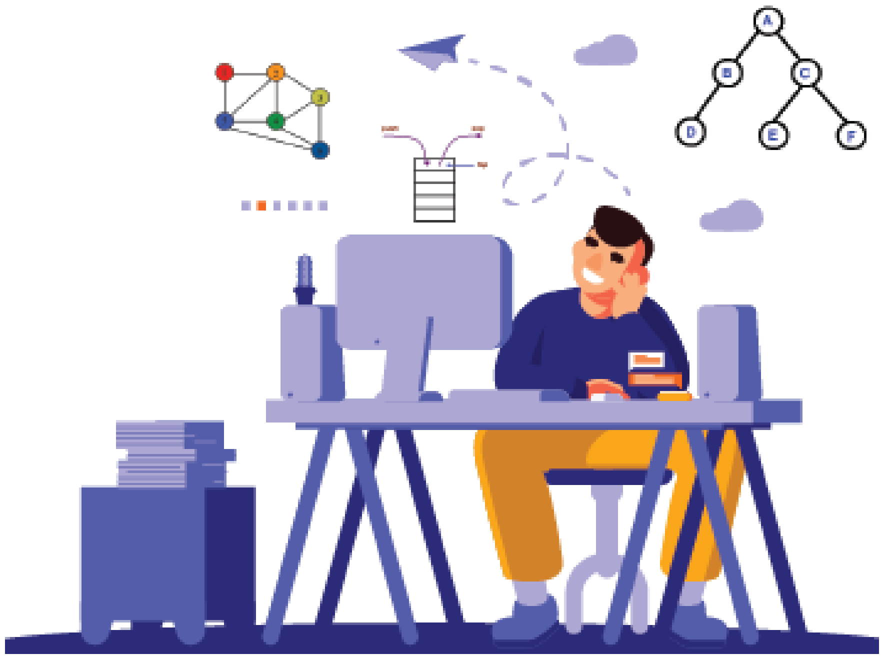
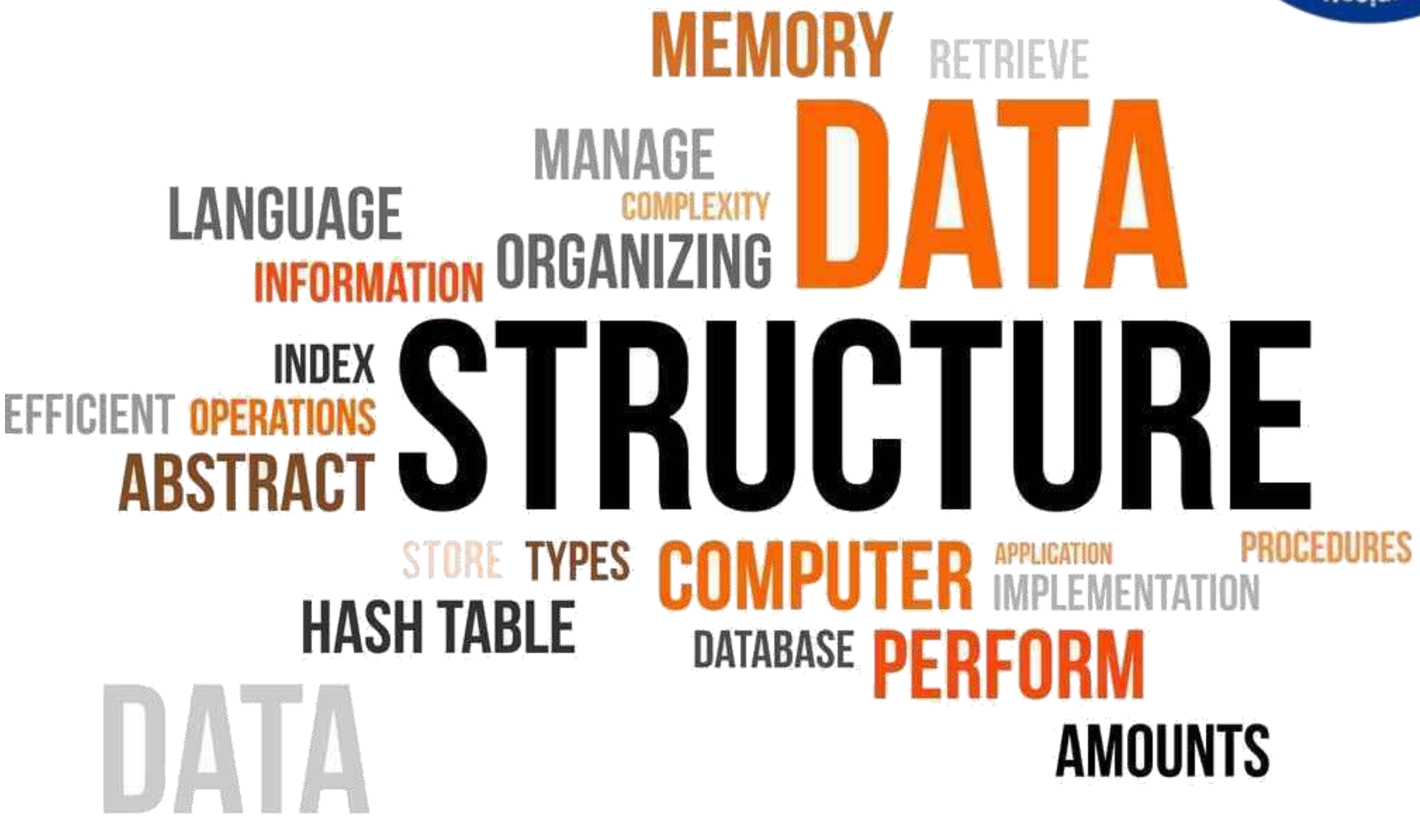


Data Structures

Course code: IT623



Dr. Rahul Mishra
Assistant Professor
DA-IICT, Gandhinagar

Hashing

Hashing is a process that involves converting data of any size or format into a fixed-size string of characters, which is typically a hexadecimal number.

- This output, known as a ***hash value or hash code***, is generated using a specific algorithm called a ***hash function***.

- The primary goals of hashing are to uniquely represent data, ensure data integrity, and provide quick data retrieval.

Some key characteristics of hashing:

1. Deterministic: A given input will always produce the same hash value. This property is essential for tasks like data verification.

2. Efficient: Hashing allows for quick processing and comparison of data. It is commonly used in data structures like hash tables for fast retrieval of information.

3. Collision Resistance: It should be unlikely for two different inputs to produce the same hash value. This property is crucial for data integrity and security.

Applications

1. Data Integrity: Hashes are used to verify that data has not been tampered with during transmission or storage. By comparing the hash of received data with the original hash, one can detect any alterations.

2. Password Storage: Instead of storing actual passwords, systems store the hash values of passwords. During login, the system hashes the entered password and checks it against the stored hash.

3. Cryptographic Applications: Hash functions play a vital role in various cryptographic algorithms, including digital signatures, message authentication codes (MACs), and cryptographic hash functions.

4. Data Retrieval: Hashing is used in data structures like hash tables for efficient storage and retrieval of information. This is commonly used in databases and caches.

5. Blockchain: Each block in a blockchain contains a hash of the previous block, creating a chain of blocks. This ensures the integrity and immutability of the data.

6. File Deduplication: Hashes are used to identify duplicate files. By comparing hash values, redundant copies can be identified and eliminated.

Direct-address tables

> Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

> Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m-1\}$, where m is not too large.

> Assumption: No two element have the same key.

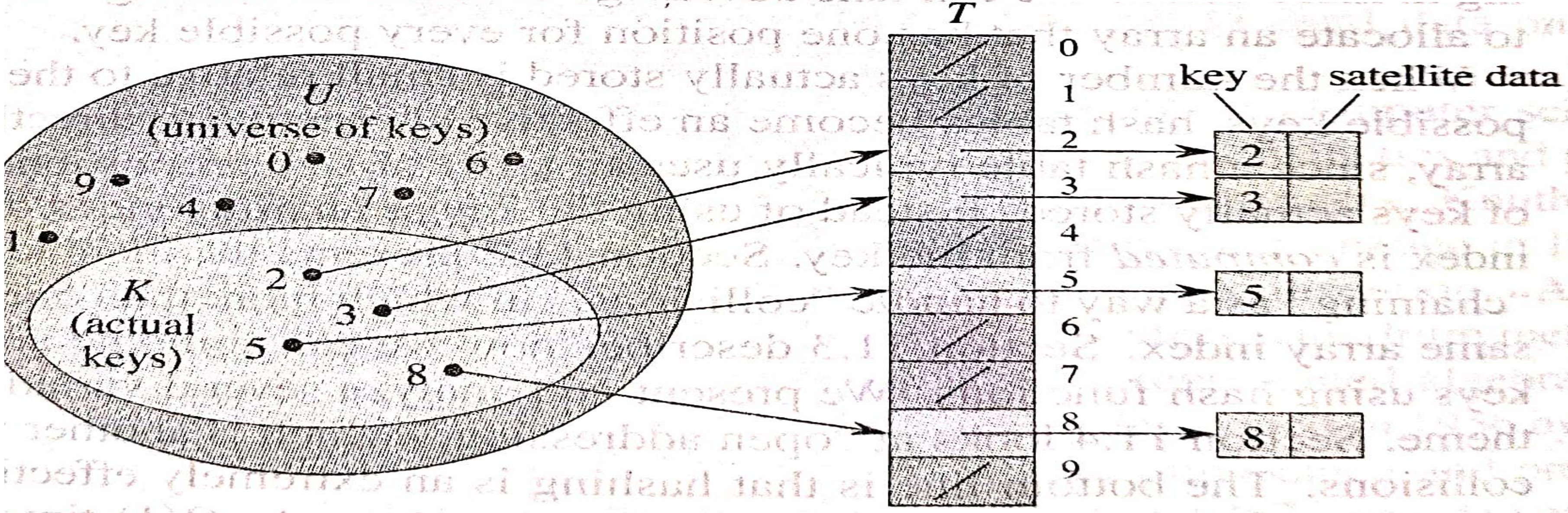
> To represent the dynamic set, we use an array, or direct-address table, denoted by $T[0..m-1]$, in which each position, or slot, corresponds to a key in the universe U.

DIRECT-ADDRESS-SEARCH(T, k)
    return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
    $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)
    $T[x.key] = \text{NIL}$

Each of these operations takes only $O(1)$ time.



> For some applications, the direct-address table itself can hold the elements in the dynamic set.

> Rather than storing an element's key and satellite data in an object external to the direct-address table, with a pointer from a slot in the table to the object.

> We would use a special key within an object to indicate an empty slot.

Hash tables :

Problems with direct addressing:

a) if the universe U is large, storing a table T of the size |U| may be impractical

b) 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.

* When the set of K keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table.

* We can reduce the storage requirement to $\Theta(|K|)$ while maintaining searching time $O(1)$.
→ (average-case) → direct address (worst)

* With hashing an element with key K is stored in slot $h(k)$, i.e; h is a hash function to compute the slot from the key k.

* $\boxed{h: U \rightarrow \{0, 1, \ldots m-1\}}$ the size m of the hash table is typically much less than $|U|$.

"h maps the universe U of keys into the slots of a hash table"

* An element with key k hashes to slot $h(k)$; we also say that $h(k)$ is the hash value of key k.

Example

Suppose that the keys are nine-digit social security numbers

<u>Possible hash function</u>

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

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

Class assignment (10 minutes)

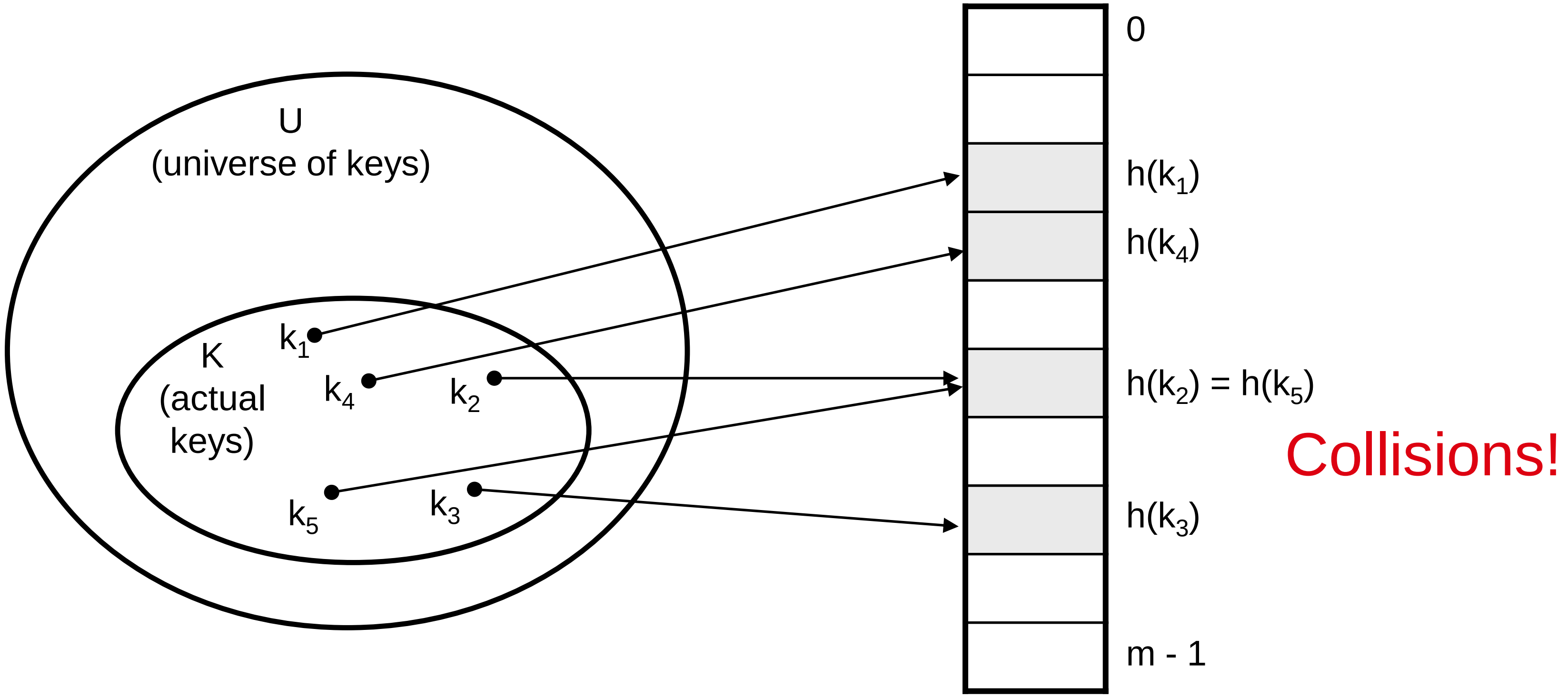
- Given the following input (4322, 1334, 1471, 9679, 1989, 6171, 6173, 4199) and the hash function (x mod 10). Calculate the hash codes.

- A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

- Find the order of key insertion…

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Do you see any problems with this approach?



Collisions

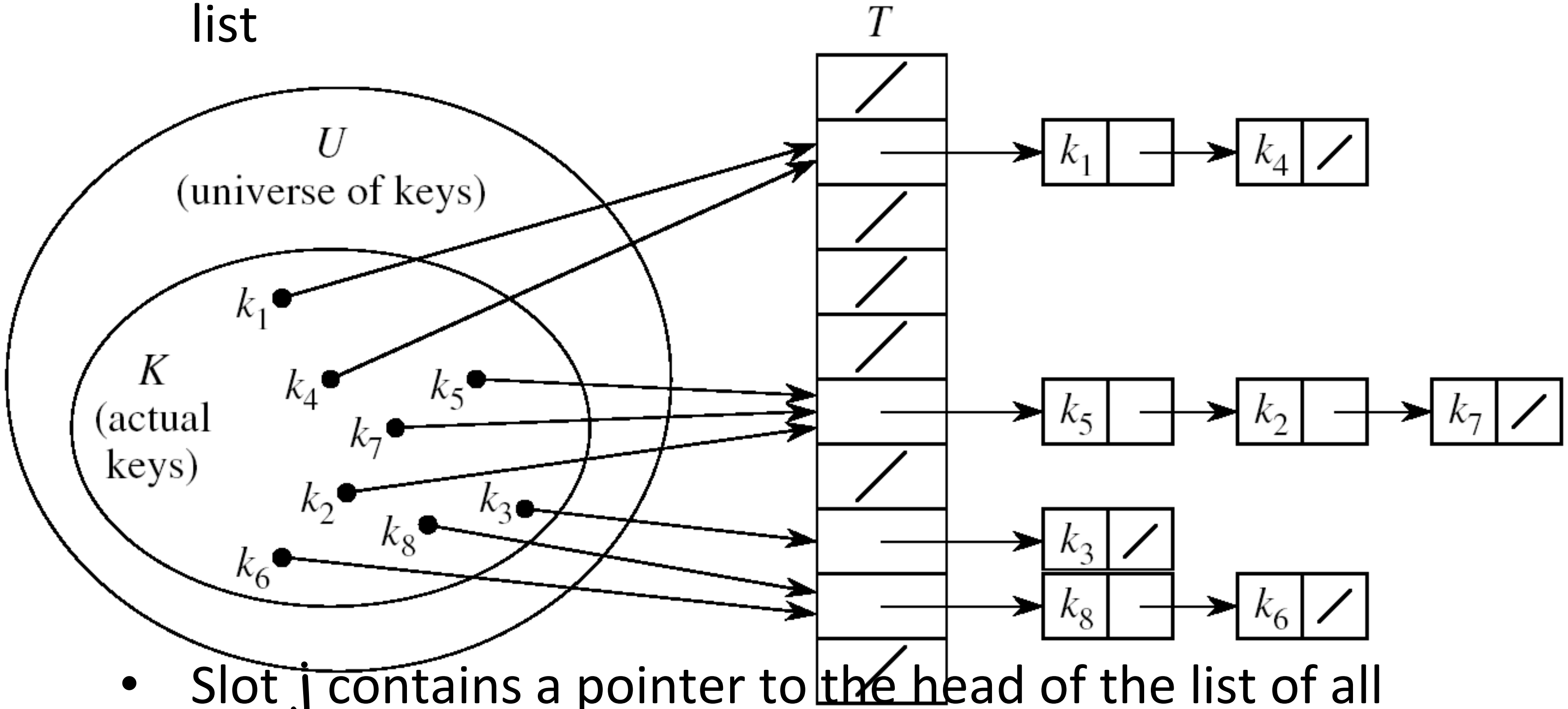
- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \le m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely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

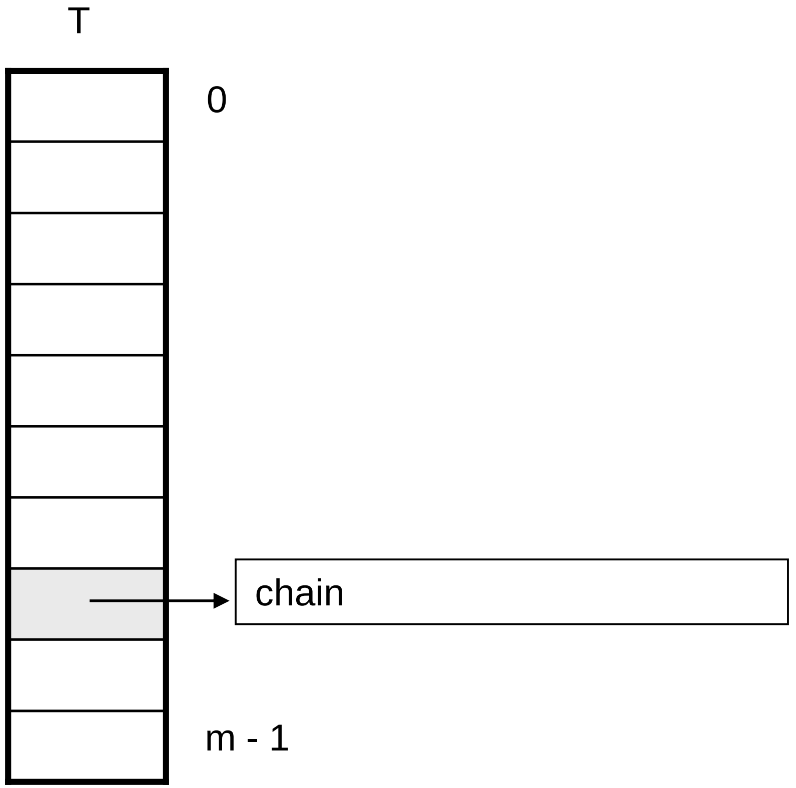
Alg.: CHAINED-HASH-SEARCH(T, k)

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:

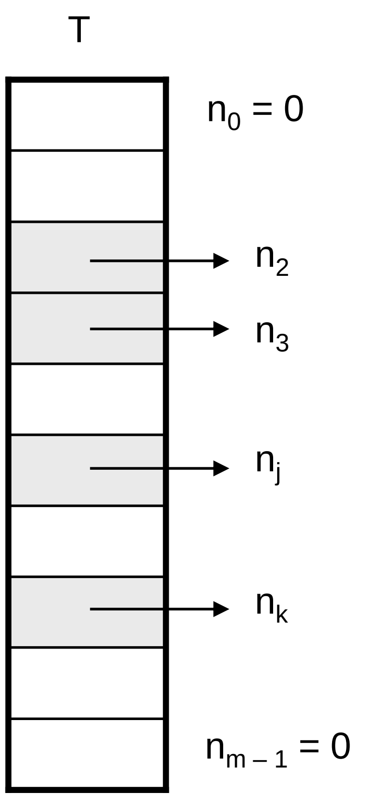
$$T[j] = n_j, \qquad j = 0, 1, \ldots, m - 1$$

- 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

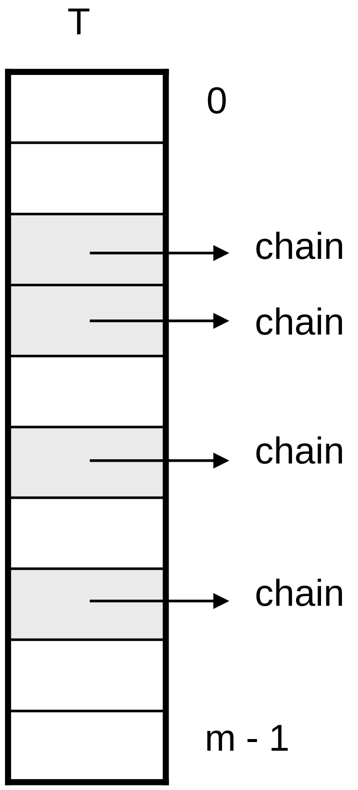
- Load factor of a hash table T:

$$\alpha = n/m$$

  - 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



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

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as pt and pts should hash to different slots
- **Derive a hash value that is independent from any patterns that may exist in the distribution of the keys**

The Division Method

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

$$h(k) = k \bmod m$$

- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

Example - The Division Method

- If $m = 2^p$, then $h(k)$ is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$

    $\Rightarrow h(k) = \{0, 1\}$, least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$

    $\Rightarrow h(k) = \{0, 1, 2, 3\}$, least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2

  k mod 97
  - Column 2: k mod 100
  - Column 3:

	m97	m100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

The Multiplication Method

Idea:

- Multiply key k by a constant A, where $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor = \lfloor m\,(k\,\underbrace{A \bmod 1)}_{\text{fractional part of } kA \,=\, kA - \lfloor kA \rfloor} \rfloor$$

- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical, e.g., typically 2^p

Example – Multiplication Method

- The value of m is not critical now (e.g., $m = 2^p$)

assume $m = 2^3$

```
       .101101 (A)
        110101 (k)
--------------
1001010.0110011 (kA)
```

discard: 1001010

shift .0110011 by 3 bits to the left

011.0011

take integer part: 011

thus, h(110101)=011

Definition of Universal Hash Functions

$H=\{h(k): U \rightarrow (0,1,..,m-1)\}$

H is said to be universal if

$$\text{for } x \neq y,\ |(\mathbf{h}() \in \mathbf{H}: \mathbf{h(x)=h(y)}| = |\mathbf{H}|/\mathbf{m}$$

(notation: |H|: number of elements in H - cardinality of H)

How is this property useful?

- What is the probability of collision in this case ?

It is equal to the probability of choosing a function $h \in U$ such that $x \neq y$ --> $h(x) = h(y)$ which is

Pr(h(x)=h(y))= $\dfrac{|H|/m}{|H|} = \dfrac{1}{m}$

Universal Hashing – Main Result

With universal hashing the chance of collision between distinct keys k and l is no more than the $1/m$ chance of collision if locations $h(k)$ and $h(l)$ were randomly and independently chosen from the set $\{0, 1, \ldots, m - 1\}$

Designing a Universal Class of Hash Functions

- Choose a prime number p large enough so that every possible key k is in the range $[0 \ldots p - 1]$

  $Z_p = \{0, 1, \ldots, p - 1\}$ and $Z_p^* = \{1, \ldots, p - 1\}$

- Define the following hash function

  $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m,$

  $\forall\ a \in Z_p^*$ and $b \in Z_p$

- The family of all such hash functions is

  $\mathcal{H}_{p,m} = \{h_{a,b}: a \in Z_p^* \text{ and } b \in Z_p\}$

- a , b: chosen randomly at the beginning of execution

The class $\mathcal{H}_{p,m}$ of hash functions is universal

Example: Universal Hash Functions

E.g.: $p = 17$, $m = 6$

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$$

$$h_{3,4}(8) = ((3 \cdot 8 + 4) \bmod 17) \bmod 6$$

$$= (28 \bmod 17) \bmod 6$$

$$= 11 \bmod 6$$

$$= 5$$

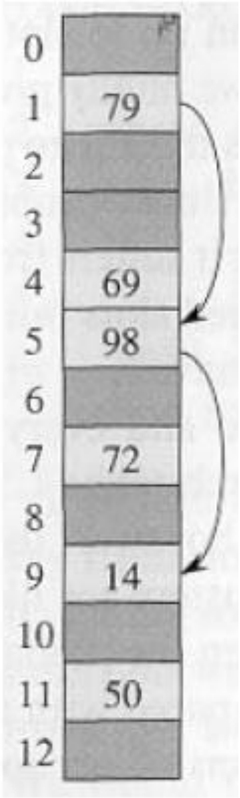
Advantages of Universal Hashing

- Universal hashing provides good results on average, independently of the keys to be stored
- Guarantees that no input will always elicit the worst-case behavior
- Poor performance occurs only when the random choice returns an inefficient hash function – this has small probability

Open Addressing

- If we have enough contiguous memory to store all the keys ($m > N$)
  $\Rightarrow$ store the keys in the table itself

e.g., insert 14



- 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 ... (we'll see why)
- Search time depends on the length of the
  probe sequence!

Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

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

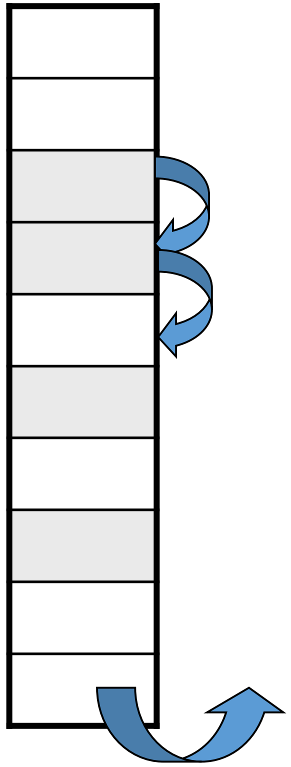
Linear probing: Inserting a key

- 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

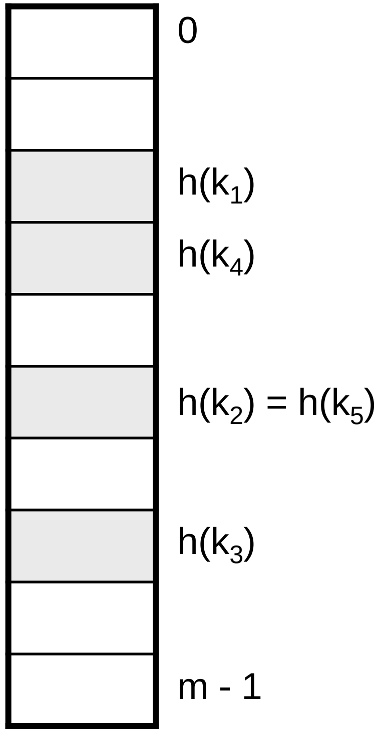


probe sequence: < h1(k), h1(k)+1 , h1(k)+2 , ...>

- Can generate m probe sequences maximum, why?

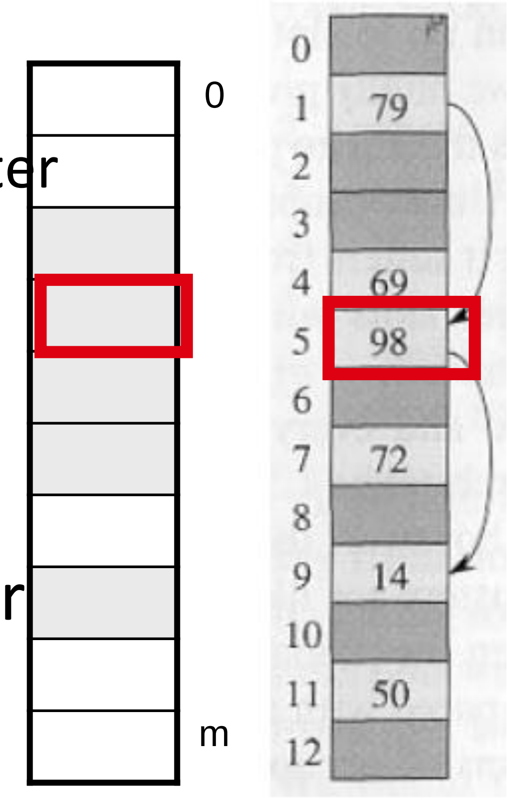
Linear probing: Searching for a key

- 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



Linear probing: Deleting a key

- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys

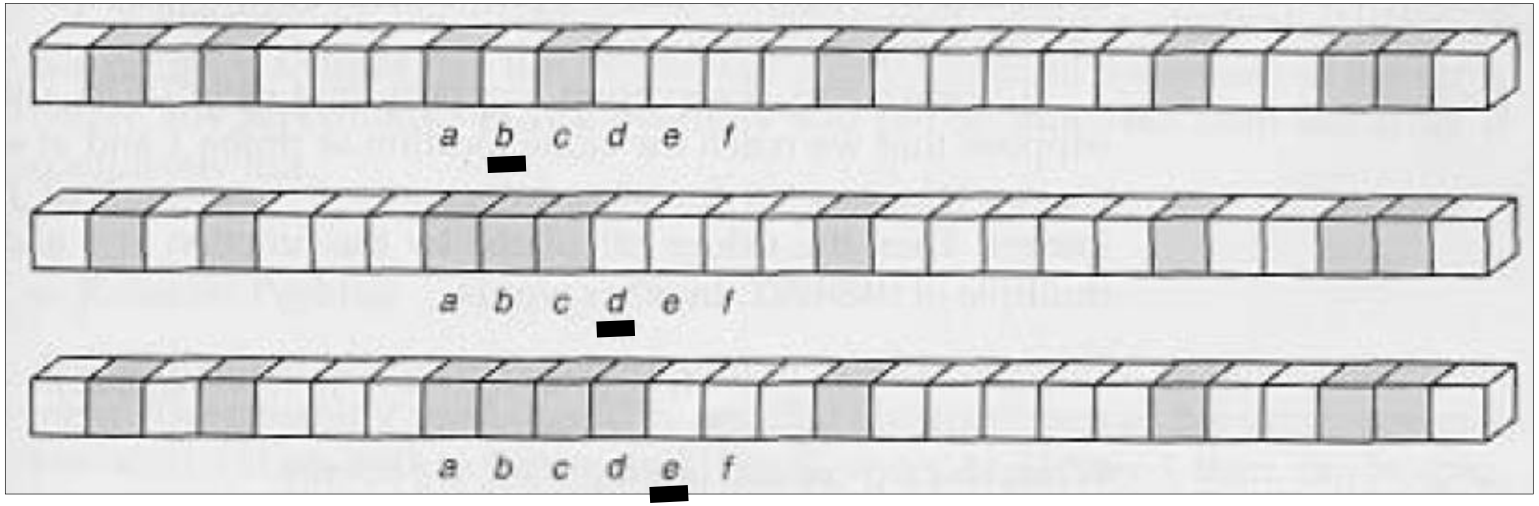


Primary Clustering Problem

- 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

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

i=0,1,2,...

- 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)

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

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

- 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

Double Hashing: Example

$h_1(k) = k \bmod 13$

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

$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

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

$h(14,1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

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

$= (1 + 8) \bmod 13 = 9$