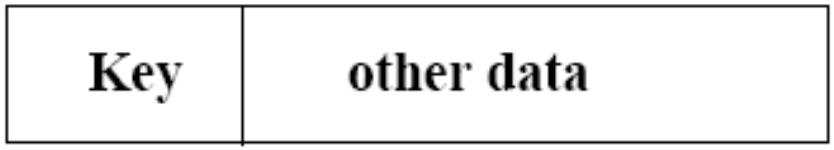
Hashing

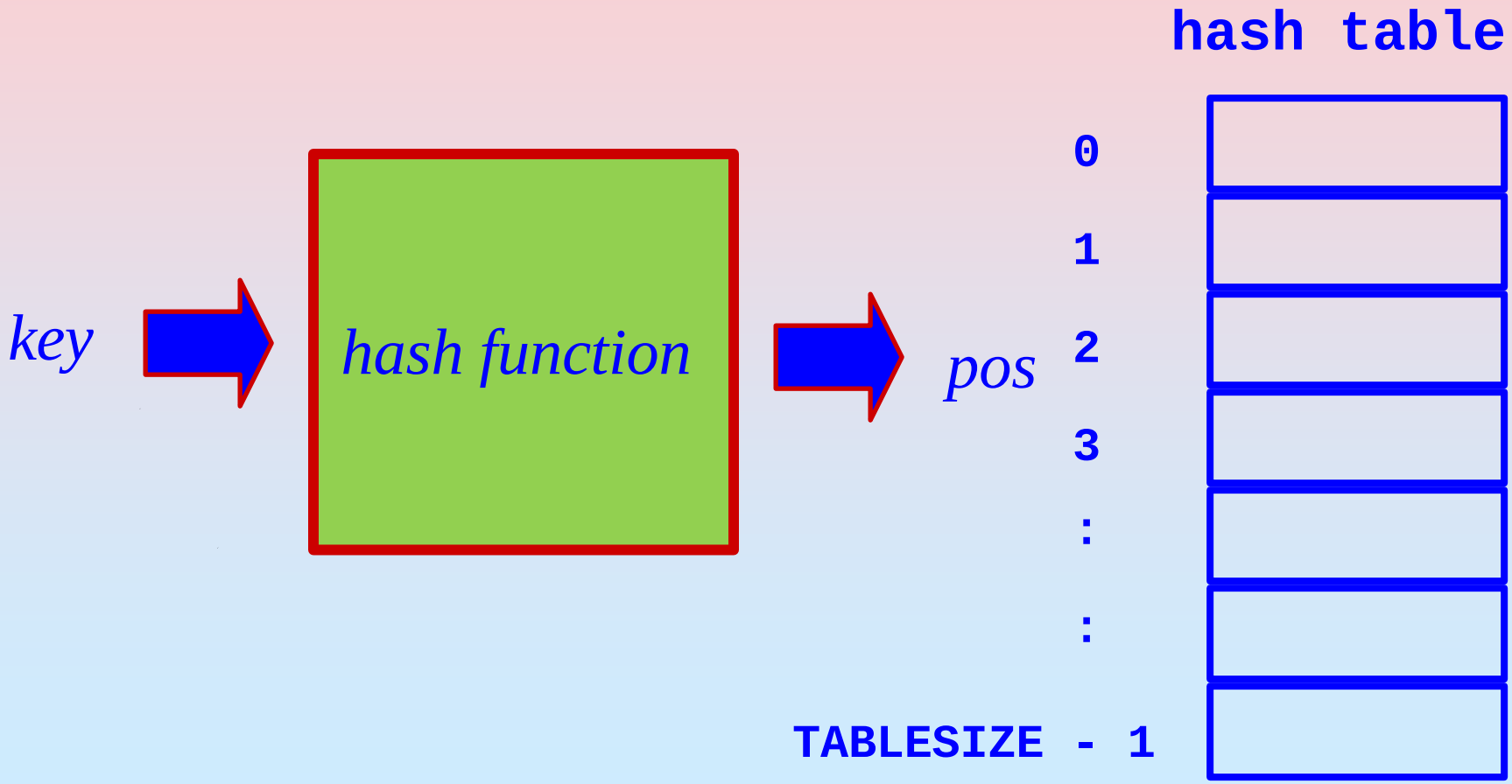
The Search Problem

- 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.

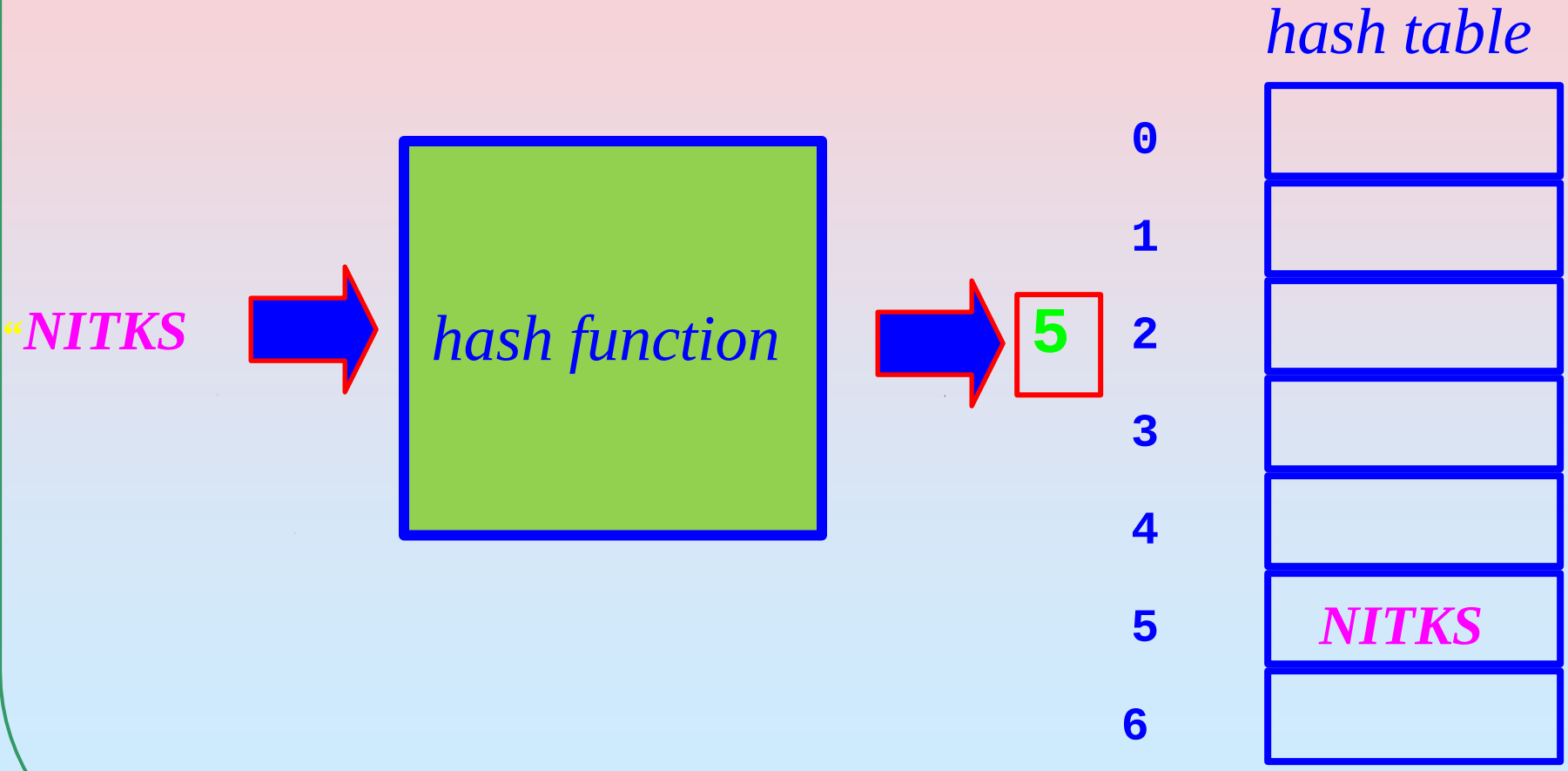
example of a record



Hashing



Example:



Hashing

- Each item has a **unique key**.
- Use a large array called a **Hash Table**.
- Use a **Hash Function**.

Operations

- **Initialize**
  - **all entries in Hash Table are empty**
- **Insert**
- **Search**
- **Delete**

Hash Function

- Maps keys to positions in the Hash Table.
- Be easy to calculate.
- Use all of the key.
- Spread the keys uniformly.

Applications of Hashing

Hashing's irreversibility and constant time access properties have found applications in a variety of domains.

Some examples of hashing applications are:

- **Security and Verification of Passwords:**
  - **Cryptographic hash functions deliver the hash value in encrypted format, using "one-way" or "irreversible" property of hash functions.**

Applications of Hashing

- **Security:**
  - **Website requires(first time) authentication via "Sign Up" to ensure that account does not fall into the wrong hands.**
  - **As a result, the password entered is stored in the database as a hash.**

Applications of Hashing

- **Verification:**
  - During login, the password hash is computed and submitted to the server, which verifies the password.
  - The hash value is sent so that there is no possibility of sniffing when data is sent from client to server.

Applications of Hashing

- **Data Structures: Hashing is extensively used in programming languages to define and develop data structures.**
  - **HashMap in Java,**
  - **Unordered map in C++, and so on.**

Applications of Hashing

- **Development of Compilers:**
  - **Programming language Compilers save all of the keywords (if, else, for, while etc.) that are used by that language in the form of a hash table.**
  - **It enables compiler to quickly access keywords in symbol table during compilation.**

Applications of Hashing

- **Digests of Messages:**
  - To ensure that files stored on cloud servers are not tampered by third-party cloud service providers, cryptographic hashing algorithms are used to compute the hash of the file/data.
  - Secure Hash Algorithm 256-bit(SHA256) is most widely used cryptographic algorithm.
  - These algorithms are also known as digest algorithms.

Applications of Hashing

- Other applications include the Rabin Karp algorithm, calculating the frequency of characters in a word, generating git commit codes with the git tool, and caching.

- Hashing is extensively used in Blockchain, Machine learning for feature hashing, storing objects or files in AWS S3(Simple Storage Service) Buckets, indexing on databases for faster retrieval, and many other applications.

Applications of Hashing

- Keeping track of customer account information in 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

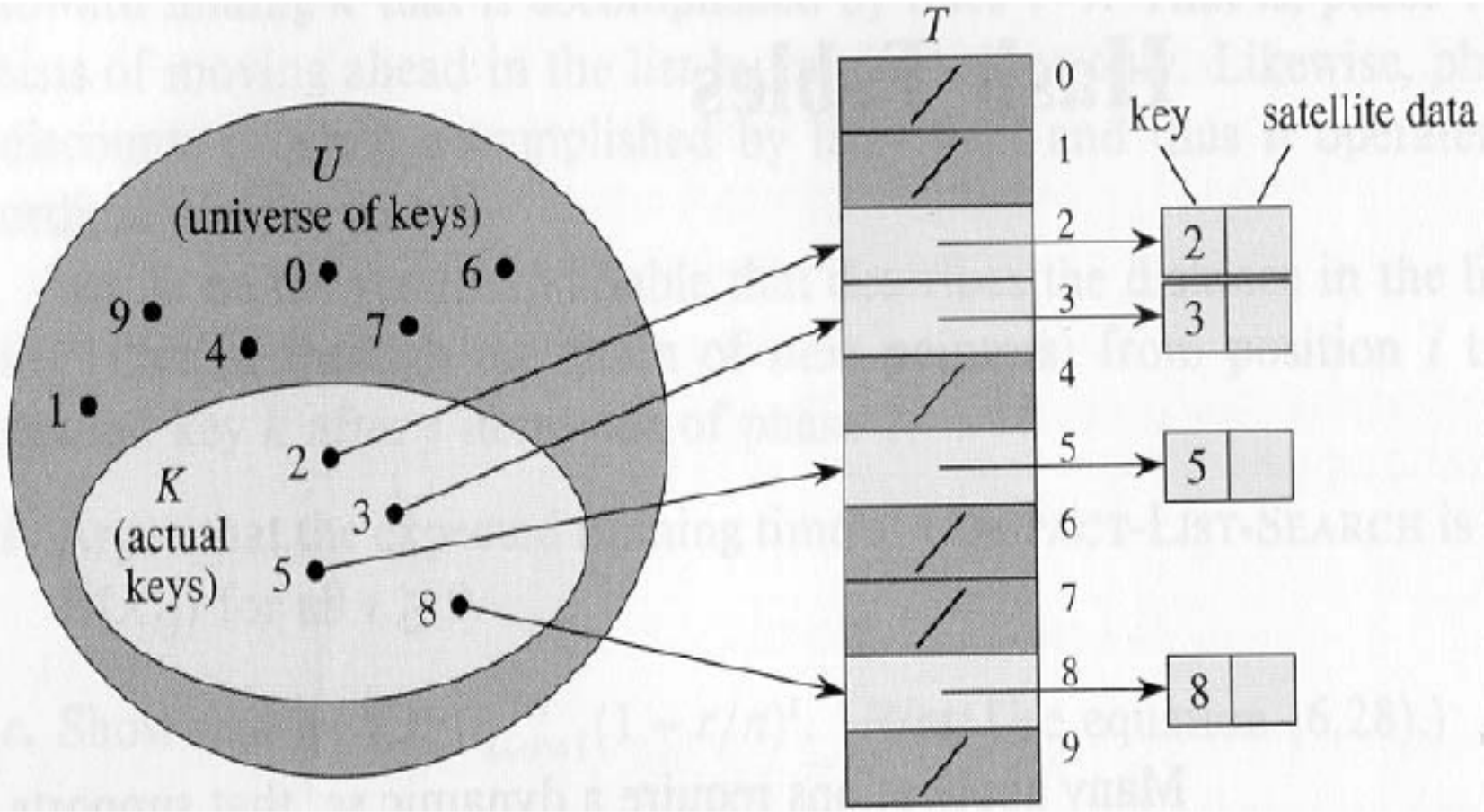
Special Case: Dictionaries

- 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, ..., 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 NULL**

Direct Addressing (cont'd)



(insert/delete in O(1) time)

Comparing Different Implementations

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

Complexity?

	Insert	Search
direct addressing	O(1)	O(1)
ordered array	O(N)	O(logN)
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.

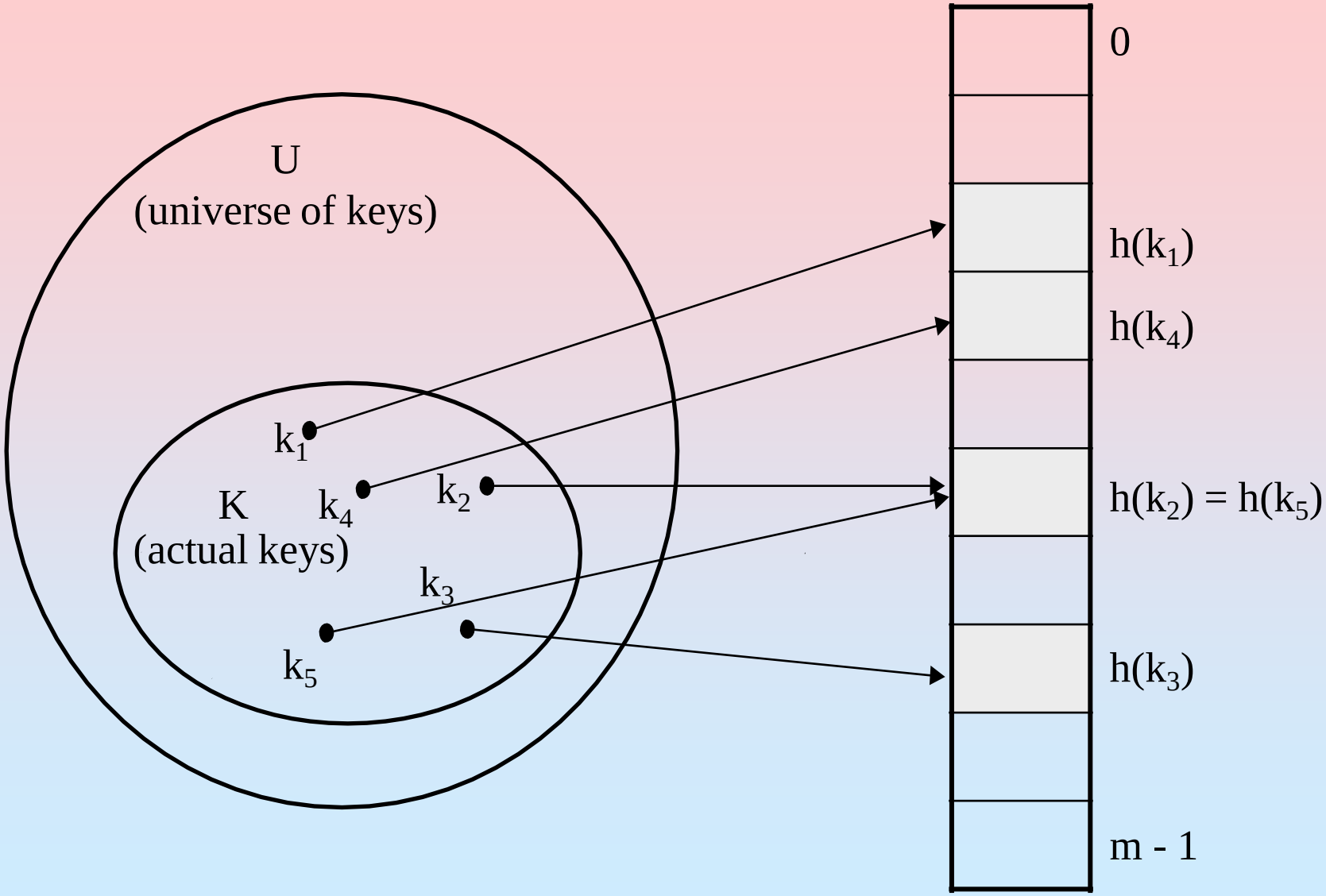
Hash Tables

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:
  - Reduced the range of array indices handled:
    - m instead of |U|
  - Storage is also reduced

Example: HASH TABLES



Revisit Example 2

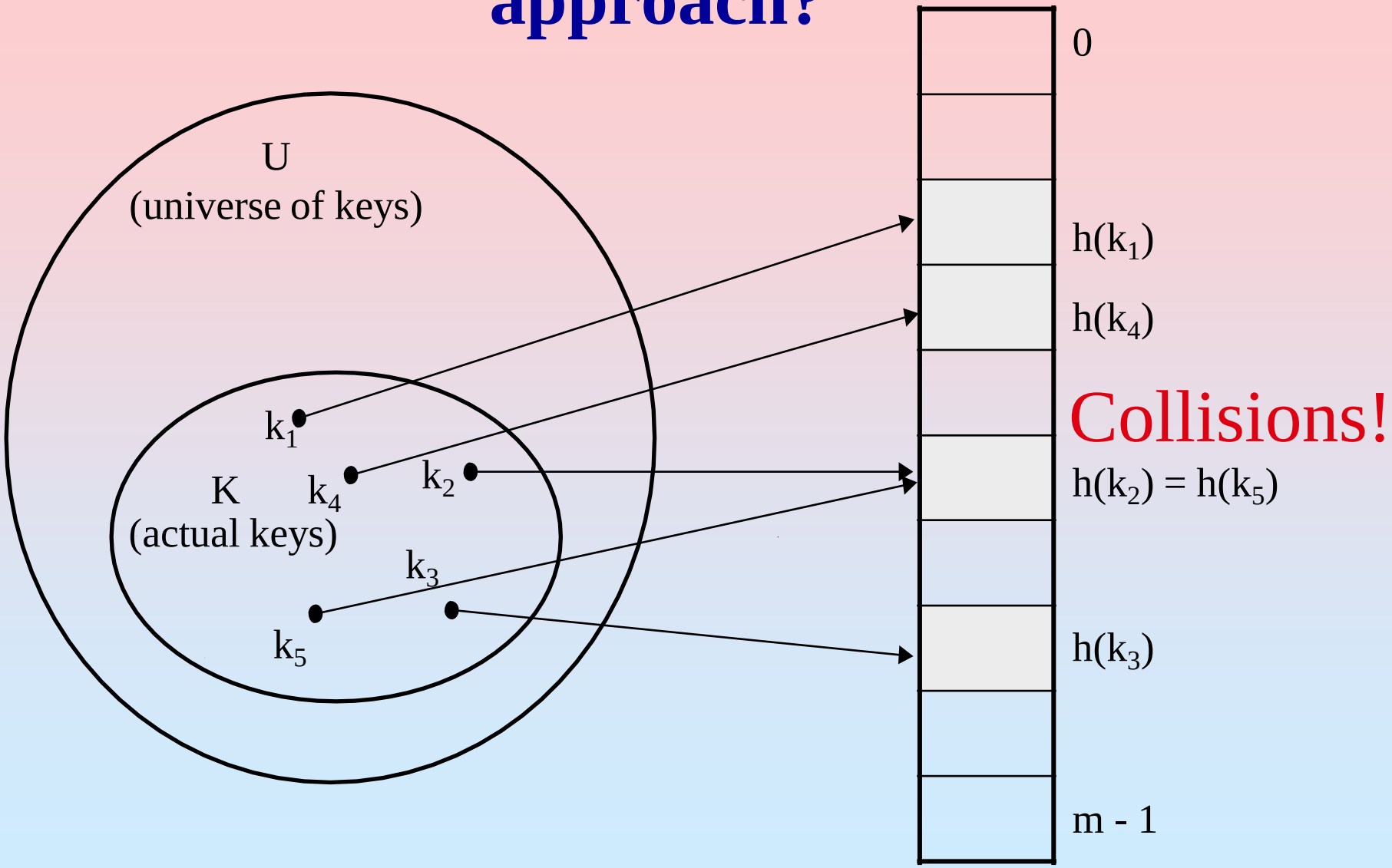
Suppose that the keys are nine-digit social security numbers

Possible hash function

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

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

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

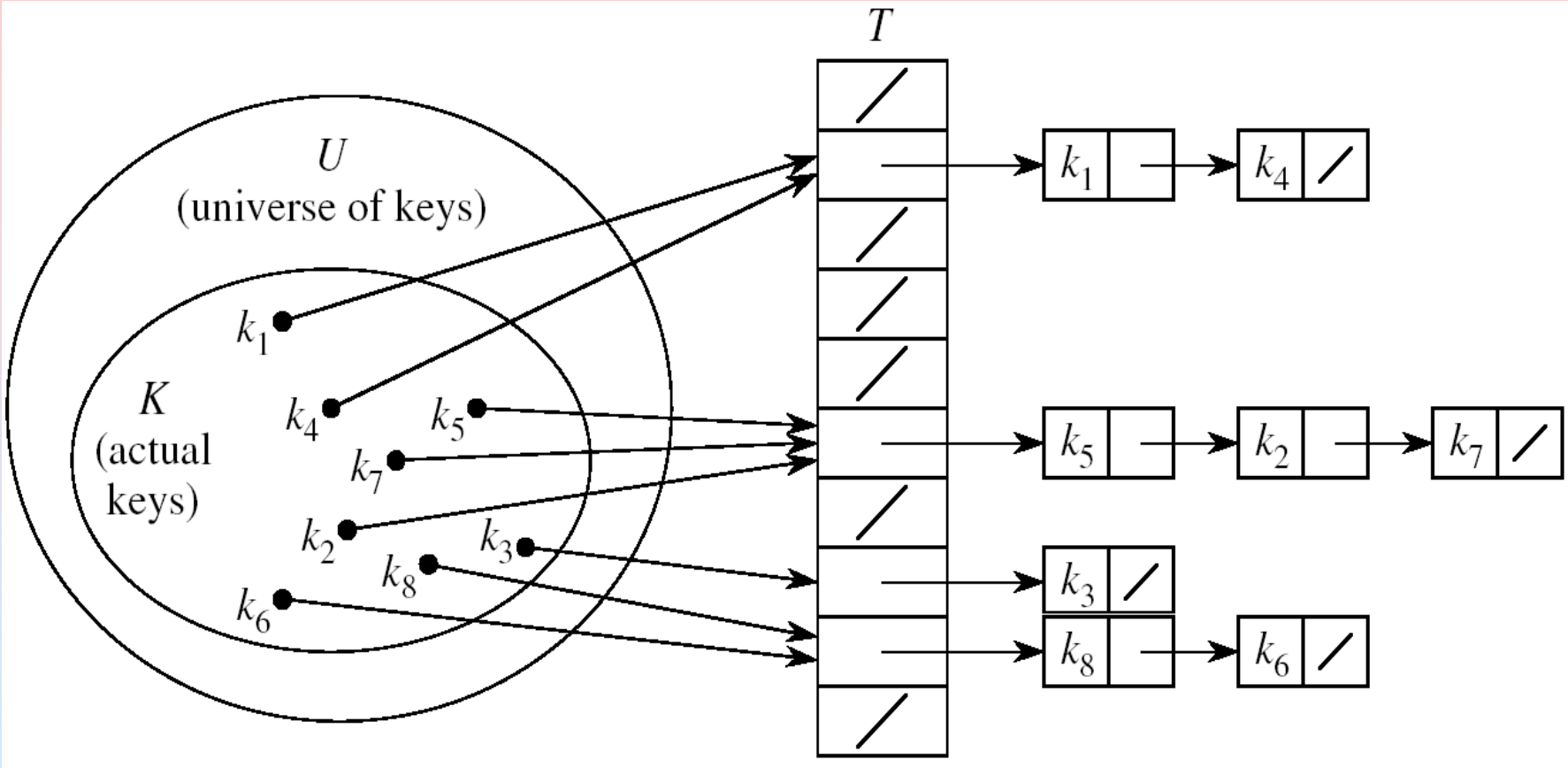
- Some of the methods to handle collisions are:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Handling Collisions using Chaining

- **Idea:**

Handling Collisions using Chaining

- **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**

Chaining : considerations

- **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 unordered?**
  - **Not ordered!**
    - **Insert is fast**
    - **Can easily remove the most recently inserted elements**

Insertion in Hash Tables

- **CHAINED-HASH-INSERT(T, x)**
  - **insert x at the head of the 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

- **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

- **CHAINED-HASH-SEARCH(T, k)**
  - **search for an element with key k in the list $T[h(k)]$**

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

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 of 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

- Choose m to be a prime, not close to a power of 2
- If $m = 2^p$, then h(k) is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$
    - h(k) = {0, 1}, least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$
    - 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
  - **Column 2:** k mod 97
  - **Column 3:** k mod 100

	m 97	m 100
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

Common Open Addressing Methods

- Linear probing
- Quadratic probing
- Double hashing

Linear probing: Inserting a key

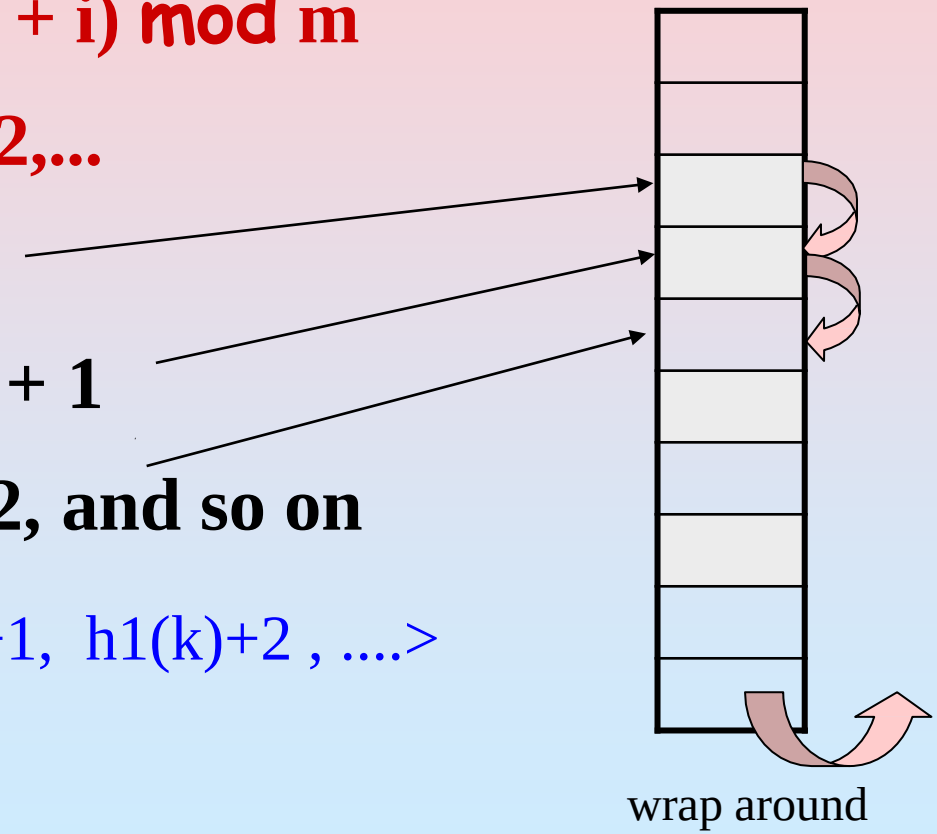
- **Idea: when there is a collision, check the next available position in the table (i.e., probing)**

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

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

- **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 , ...>



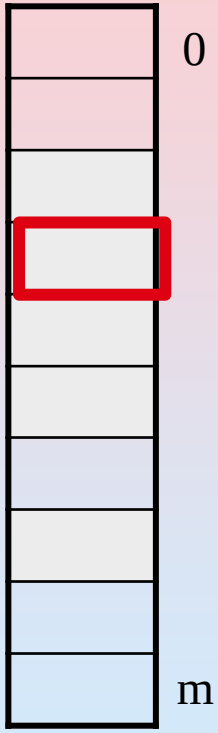
- **Can generate m probe sequences maximum.**

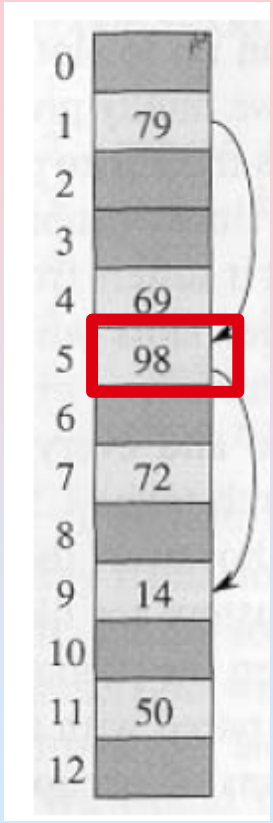
Linear probing: Searching for a key

- 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
- 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**



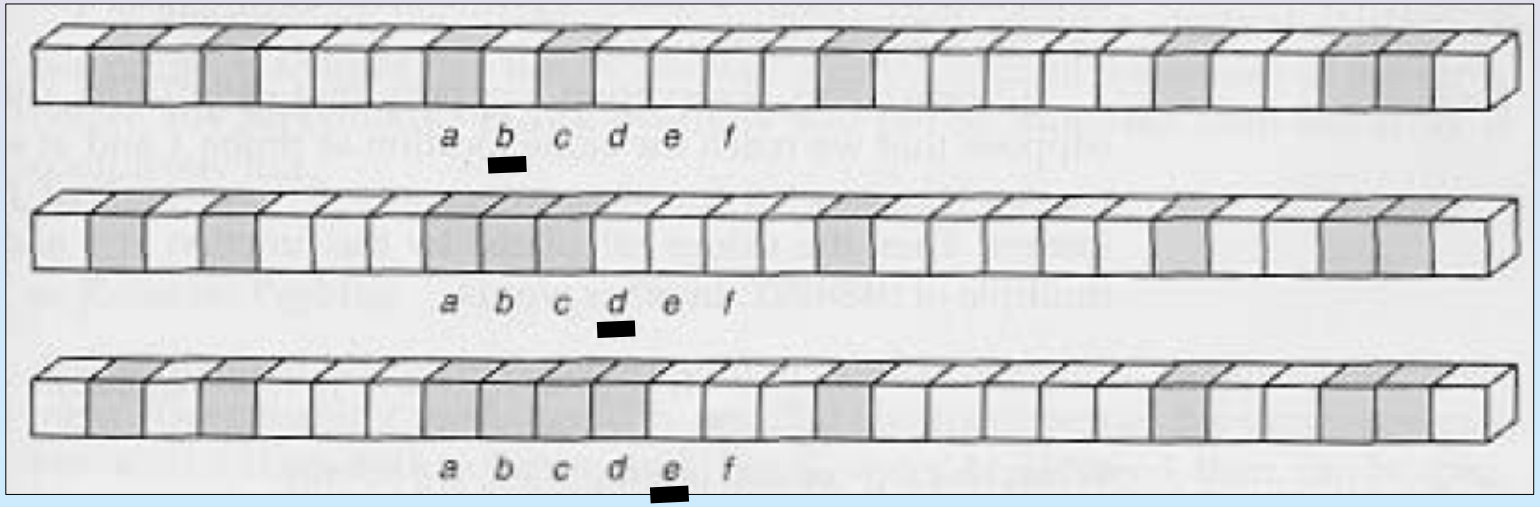


Primary Clustering Problem

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

⇒ search time increases!!

initially, all slots have probability 1/m



Slot b: 2/m

Slot d: 4/m

Slot e: 5/m

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

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:

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

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

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

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

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