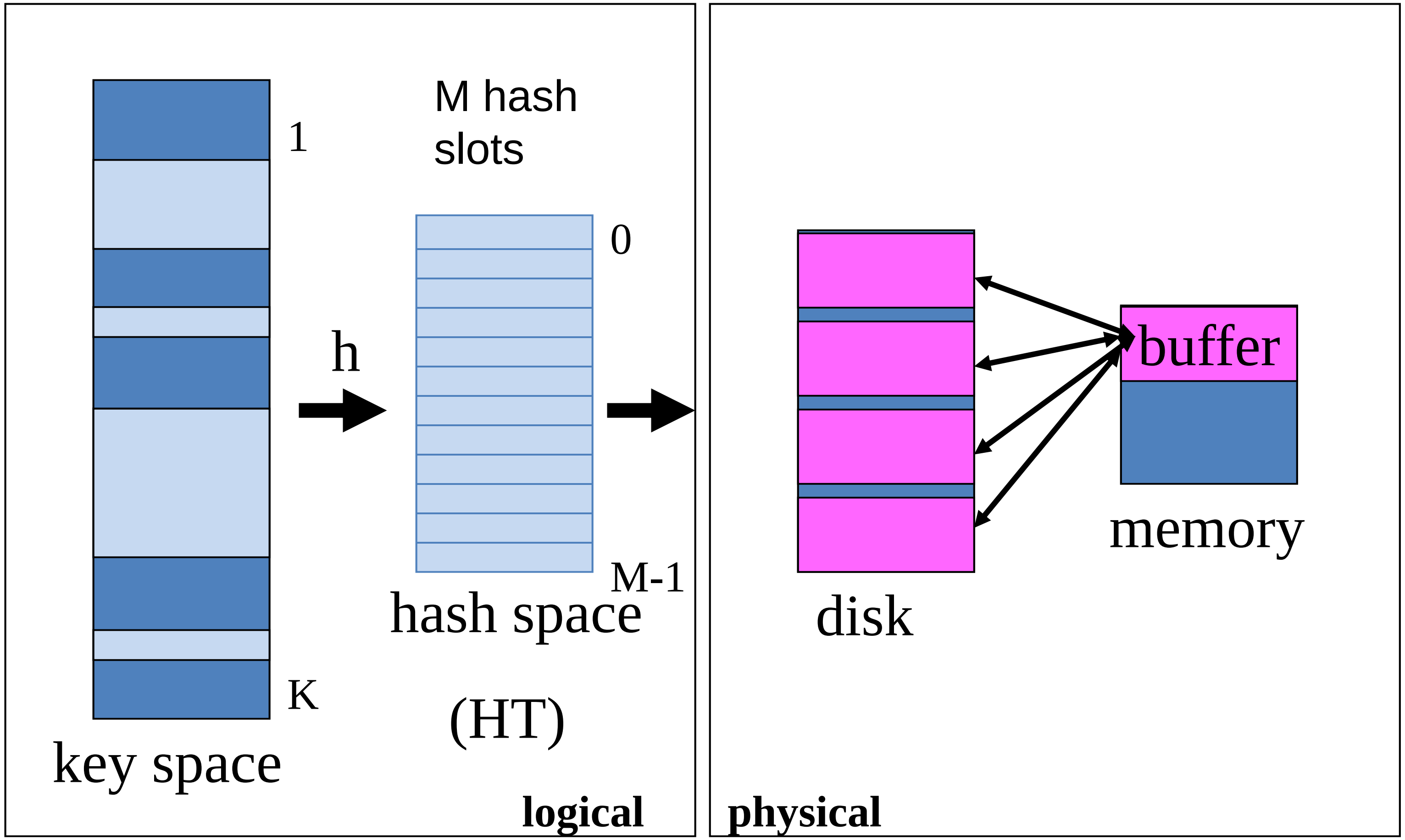
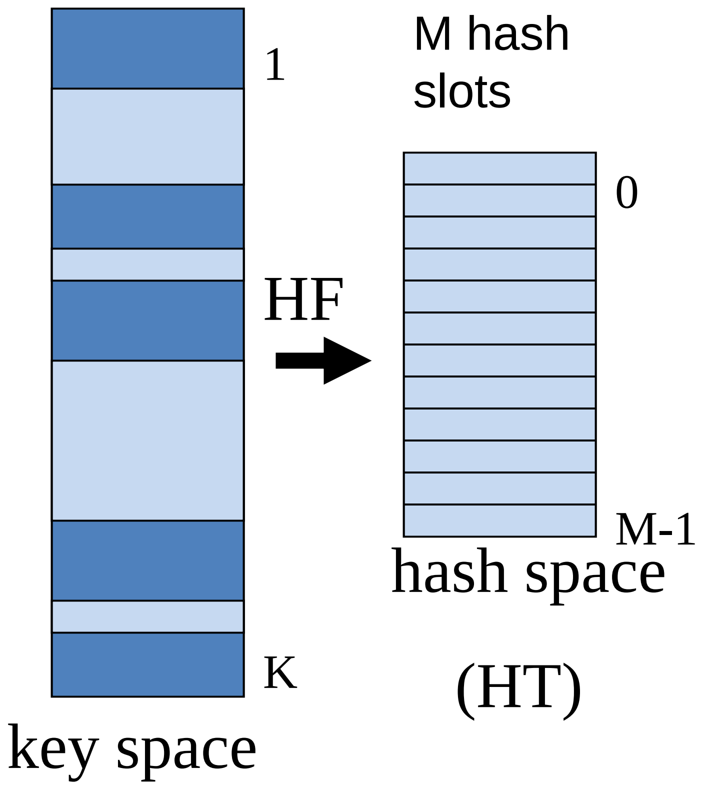
# Hashing

- Introduction
  - hash table (**HT**)
  - collection: **hash slots**
    - **element =**
      **key + data**
  - hashing function
    **h(key) → ref** **(index)**
  - operations
    - **search** (find)
    - **add** (insert)
    - **remove** (delete)
- **Advantage**
  - **search, add, remove - O(1) (constant time)**
- Disadvantages
  - HT not sorted
  - find min-, max-value not efficient
- Use
  - symbol tables / indexes

# Hashing Model



# Hashing: Logical Model




- Hashing function, h

  **0 <= h(key) <= M-1**

- Collision

  $h(key_i) = h(key_j)$ **where i != j**

- Collision resolution
  - **Chaining from slot**
  - **h(key) + f(i)**
    - **i is the i-th collision**
    - **f(i) = i, i*i, i*$h_2$(key)**

# Hashing Functions 1

- Ideal requirement
  - simple uniform hashing
    - each key equally likely to hash to any particular slot in HT
    - implies: must know the probability distribution of the data (keys)
    - in general - not known

- Hash function
  - maps to an integer **(0..M-1)**
  - key value may be
    - integer
    - character / string
    - other ???
  - easy to compute (time)
  - implementation issues
    - require care in choice of h

# Hashing Functions 2

- division method

  **h(k) = k mod M**

  (often M is prime)
- multiplication method

  h(k) =

  **floor( M (kA - floor(kA)))**

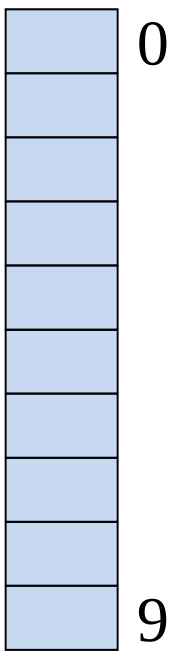
  where 0< A <1

- (other methods)

- Implementation issues
  - **overflow** in the calculation of h(k)
  - how to handle strings (non-numeric data)
  - probability distribution for the key
  - words in English are not uniformly distributed
  - Personal names (**Zipf**)

# Simple example

- Hash table HT




- Assumptions
  - h = mod 10
    - i.e. Hash space is 0..9
  - key values are simple integers
  - element = key + data

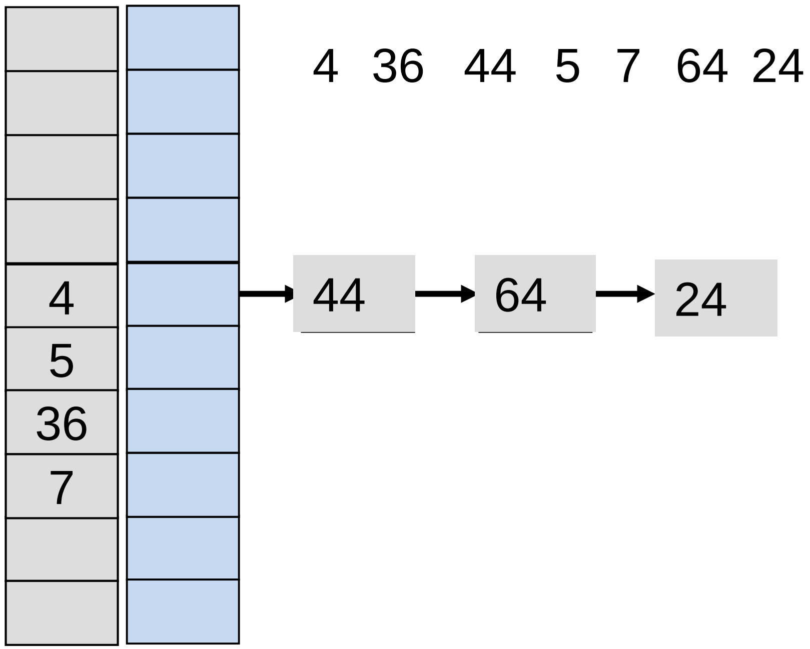
    **(only the key values are shown)**
  - HT entries may be

    key + ref

    (in some methods)

# Collision resolution 1

- E.g. **4 36 44 5 7 64 24**




- **Separate chaining**
  **(Open hashing)**
  - collisions to same slot placed in a linked list
  - may act like a stack or a queue
  - find traverses the list
  - -ve: requires pointers

# Collision resolution 2

E.g. 4 36 44 5 7 64 24

| | |
|---|---|
| 24 | |
| | |
| | |
| | |
| 4 | |
| 44 | |
| 36 | |
| 5 | |
| 7 | |
| 64 | |

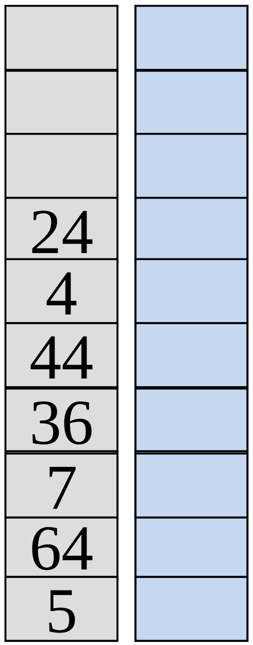
4 36 44 5 7 64 24

- **Open addressing**
  **(closed hashing)**
  - collisions resolved by searching forward in the hash space to find a free slot (circular search)
  - **h(key) + f(i) where f(i) = i**
  - find becomes a linear search O(n)
  - -ve: **primary clustering**

    solution: quadratic probing

# Collision resolution 3

E.g. 4 36 44 5 7 64 24




4 36 44 5 7 64 24

- **Quadratic probing**
  - **h(key) + f(i)**
    - **where f(i) = $i^2$**
  - i.e try slot k+1, k+4,…
  - solves primary clustering
  - **if size(HT) prime then if load < 50%, a new element can always be inserted**
  - -ve:

    **secondary clustering**

# Collision resolution 4

E.g. 4 36 44 5 7 64 24

| | |
|---|---|
| 64 | |
| | |
| | |
| | |
| 4 | |
| 5 | |
| 36 | |
| 7 | |
| 24 | |
| 44 | |

4 36 44 5 7 64 24

**7 – (24 mod 7) = 7 – 3 = 4*1 = 4**

**7 – (44 mod 7) = 7 – 2 = 5*1 = 5**

**7 – (64 mod 7) = 7 – 1 = 6*1 = 6**

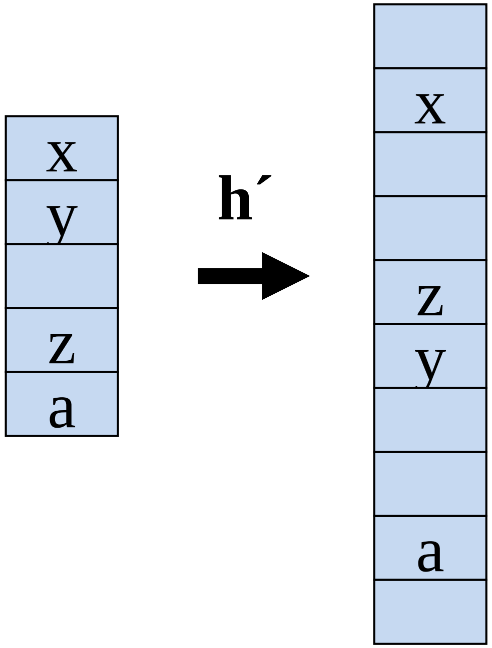
- Double hashing
  - **$h_1$(key) + f(i)**
  - **where f(i) = i * $h_2$(key)**
  - e.g. **$h_2$ = R - (key mod R)** where R is prime < size(HT)
  - Ex: **choose say R = 7**

    **$h_2$ = 7 - (key mod 7)**

    & probe is slot + f(i) **i = 1, 2,**
  - -ve: $h_2$ may add more time to the calculation

# High Load / Table full

- Rehash (expensive **O(n)** )



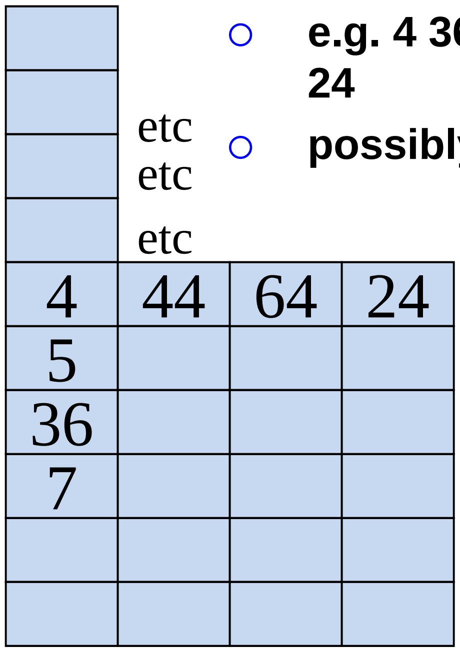

- New size(HT) =

**first prime > 2 * old size(HT)**

- quadratic probing
  - rehash when
    - **load > 50%**
    - **insert fails**
    - **load > x%**

# Hash Buckets / Overflow Slots

- Hash buckets
  - each slot may contain n entries
  - e.g. 4 36 44 5 7 64 24
  - possibly sort slot




- Overflow slots