Searching

Dr. Nguyen Hua Phung

HCMC University of Technology, Viet Nam

05, 2020

Outline

Definition

- $(k_1, l_1), (k_2, l_2), \ldots, (k_n, l_n)$
- Given K, locate (k_j, l_j) such that $K == k_j$

Terms

- Successfull search: $\exists j, K == k_j$
- Unsuccessful search: $\forall j, K \neq k_j$
- Exact-match query: key value exactly match searching key
- Range query: key value within a specific range

Sequential Searching

- Sequentially comparing K with k_j until K is equal to k_j or no more element.
- Best case: O(1)
- Worst case: O(n)

- Given sorted array A and some value j,
- Comparing K with every j'th element, i.e. A[j], A[2j], ... until K is
  - == A[kj] $\Rightarrow$ return found
  - $>$ A[kj] $\Rightarrow$ linear search from A[(k-1)j] to A[kj]
  - no more element $\Rightarrow$ linear search from A[(k-1)j] to A[n]
- If j is $\sqrt{n}$, worst case cost $\approx 2\sqrt{n}$.

Binary Search

- Given an array A in ascending order,
- Comparing K with the element at the middle A[n/2], if K is
  - == A[n/2] $\Rightarrow$ return found
  - < A[n/2] $\Rightarrow$ recursively search on the left half of A
  - > A[n/2] $\Rightarrow$ recursively search on the right half of A
- Worst case: $O(log_2 n)$

- Given a sorted array A,
- Assume the values in A are **evenly** distributed,
- Like binary search but comparing with the element at

$$p = \frac{\text{K - A[1]}}{\text{A[n] - A[1]}} \times n$$

Frequency of Access

- In many search applications, real access patterns follow the rule: *80/20*
- 80% access to 20% of data
- Arrange the array based on the frequency of access instead of the value of element
  - Sort the array based on the frequency
  - Move-to-front the element when it is found
  - Transpose: move the element one step toward the front when it is found
- Sequential search

Example of frequency Sorting

- Keep the array sorted by frequency
- Search F
- Search D
- Search D

0	0	0	0	0	0	0	0
A	B	C	D	E	F	G	H

Example of frequency Sorting

- Keep the array sorted by frequency
- Search F
- Search D
- Search D

0	0	0	0	0	1	0	0
A	B	C	D	E	F	G	H

Example of frequency Sorting

- Keep the array sorted by frequency
- Search F
- Search D
- Search D

1	0	0	0	0	0	0	0
F	A	B	C	D	E	G	H

Example of frequency Sorting

- Keep the array sorted by frequency
- Search F
- Search D
- Search D

1	0	0	0	1	0	0	0
F	A	B	C	D	E	G	H

Example of frequency Sorting

- Keep the array sorted by frequency
- Search F
- Search D
- Search D

1	1	0	0	0	0	0	0
F	D	A	B	C	E	G	H

Example of frequency Sorting

- Keep the array sorted by frequency
- Search F
- Search D
- Search D

1	2	0	0	0	0	0	0
F	D	A	B	C	E	G	H

Example of frequency Sorting

- Keep the array sorted by frequency
- Search F
- Search D
- Search D

2	1	0	0	0	0	0	0
D	F	A	B	C	E	G	H

Example of Move_to_front

- Apply move_to_front
- Search F
- Search D
- Search D

A	B	C	D	E	F	G	H

Example of Move_to_front

- Apply move_to_front
- Search F
- Search D
- Search D

A	B	C	D	E	F	G	H

Example of Move_to_front

- Apply move_to_front
- Search F
- Search D
- Search D

F	A	B	C	D	E	G	H

Example of Move_to_front

- Apply move_to_front
- Search F
- Search D
- Search D

F	A	B	C	D	E	G	H

- Apply move_to_front
- Search F
- Search D
- Search D

D	F	A	B	C	E	G	H

- Apply move_to_front
- Search F
- Search D
- Search D

D	F	A	B	C	E	G	H

Example of Transpose

- Apply transpose
- Search F
- Search D
- Search D

A	B	C	D	E	F	G	H

Example of Transpose

- Apply transpose
- Search F
- Search D
- Search D

A	B	C	D	E	F	G	H

- Apply transpose
- Search F
- Search D
- Search D

A	B	C	D	F	E	G	H

Example of Transpose

- Apply transpose
- Search F
- Search D
- Search D

A	B	C	D	F	E	G	H

Example of Transpose

- Apply transpose
- Search F
- Search D
- Search D

A	B	D	C	F	E	G	H

Example of Transpose

- Apply transpose
- Search F
- Search D
- Search D

A	B	D	C	F	E	G	H

Example of Transpose

- Apply transpose
- Search F
- Search D
- Search D

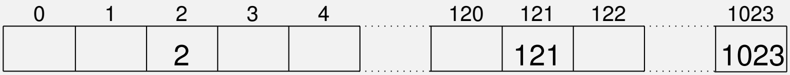
A	D	B	C	F	E	G	H

- Sequential Search: O(n)
- Binary Search: O(log n)

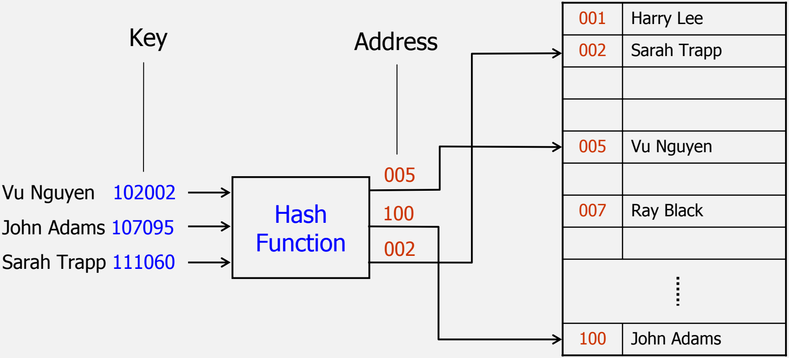
requiring several key comparisons before the target is found

- Is there a search algorithm whose complexity is O(1)?
  YES
  - Use key as the array index
  - Good on time complexity but bad on space complexity
  - Use a function to map from big space to smaller space: hashing function

Example



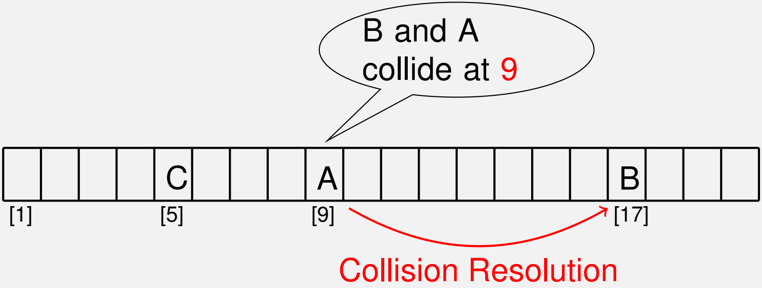
Hash Function Example



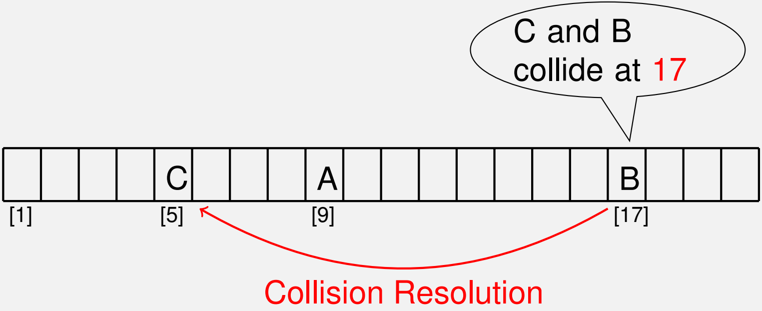
- Home address: address produced by a hash function
- Prime area: memory that contains all the home address
- Synonyms: a set of keys that hash to the same location
- Collision: the location of the data to be inserted is already occupied by other data.
- Probing: the technique to resolve the collisions.
- Ideal hashing:
  - No location collision
  - Compact address space

Basic Concepts

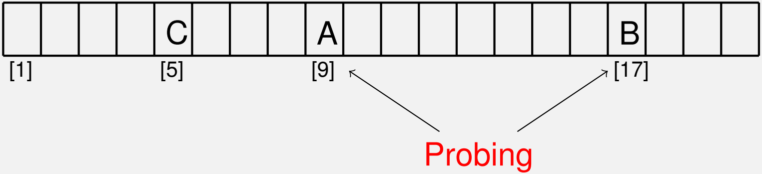
- Insert A, B, C
- hash(A) = 9
- hash(B) = 9
- hash(C) = 17



- Insert A, B, C
- hash(A) = 9
- hash(B) = 9
- hash(C) = 17



- Insert A, B, C
- hash(A) = 9
- hash(B) = 9
- hash(C) = 17



Hash Functions

- Direct hashing
- Modulo division
- Digit extraction
- Mid-square
- Folding
- Rotation
- Pseudo-random

Direct Hashing

- The address is the key itself
$$\text{hash}(\text{Key}) = \text{Key}$$
- Advantage: there is no collision.
- Disadvantage: the address space (storage size) is as large as the key space

Modulo Division

$$\text{Address} = \text{Key MOD listSize} + 1$$

- Fewer collisions if listSize is a prime number
- Example:
  - Numbering system to handle 1,000,000 employees
  - Data space to store up to 300 employees
  - hash(121267) = 121267 MOD 307 + 1 = 2 + 1 = 3

Digit Extraction

Address = selected digits from Key

- Example:

$$
\begin{array}{lcl}
379452 & \rightarrow & 394 \\
121267 & \rightarrow & 112 \\
378845 & \rightarrow & 388 \\
160252 & \rightarrow & 102 \\
045128 & \rightarrow & 051
\end{array}
$$

$$\text{Address} = \text{middle digits of Key}^2$$

- Example:

$$9452 * 9452 = 89340304 \rightarrow 3403$$

- Disadvantage: the size of the Key^2 is too large
- Variations: use only a portion of the key

379452	:	379 * 379	=	143641	$\rightarrow$	364
121267	:	121 * 121	=	014641	$\rightarrow$	464
045128	:	045 * 045	=	002025	$\rightarrow$	202

- The key is divided into parts whose size matches the address size
- Example:

  Key = 123456789
  Folding: 123|456|789
  Fold shift: 123 + 456 + 789 = 1368 $\Rightarrow$ 368
  or
  Fold boundary: 321 + 456 + 987 = 1764 $\Rightarrow$ 764

Rotation

- Hashing keys that are identical except for the last character may create synonyms.
- The key is rotated before hashing.
- Used in combination with fold shift
- Spreading the data more evenly across the address space

original key	rotated key	fold shift
600101	160010	16
600102	260010	26
600103	360010	36
600104	460010	46
600105	560010	56

Pseudorandom

Key → **Pseudorandom Number Generator** → Random Number → **Modulo Division** → Address

$$y = ax + c$$

- For maximum efficiency, a and c should be prime numbers
- Example:

  Key = 121267 a = 17 c = 7 listSize = 307

$$\begin{aligned}
\text{Address} &= ((17*121267 + 7) \text{ MOD } 307 + 1 \\
&= (2061539 + 7) \text{ MOD } 307 + 1 \\
&= 2061546 \text{ MOD } 307 + 1 \\
&= 41 + 1 \\
&= 42
\end{aligned}$$

- Except for the direct hashing, none of the others are one-to-one mapping
  $\Rightarrow$ Requiring collision resolution methods
- Each collision resolution method can be used independently with each hash function
- A rule of thumb: a hashed list should not be allowed to become more than 75% full.
- Load factor:

$$\alpha = (k/n) * 100$$

$$\begin{array}{lcl} n & = & \text{list size} \\ k & = & \text{number of filled elements} \end{array}$$

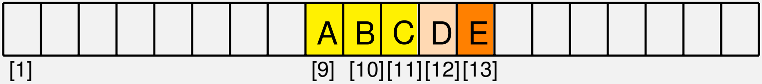
Collision resolution

- As data are added and collisions are resolved, hashing tends to cause data to group within the list
  $\Rightarrow$ Clustering: data are unevenly distributed across the list
- High degree of clustering increases the number of probes to locate an element
  $\Rightarrow$ Minimize clustering

Collision resolution

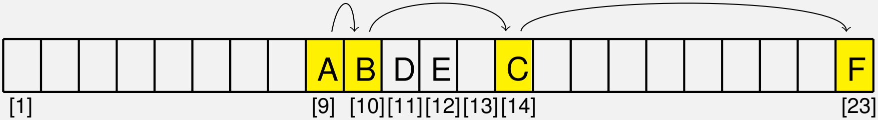
- Primary clustering: data become clustered around a home address.

  Insert A_9, B_9, C_9, D_{11}, E_{12}



- Secondary clustering: data become grouped along a collision path throughout a list.

Insert A_9, B_9, C_9, D_{11}, E_{12},F_9

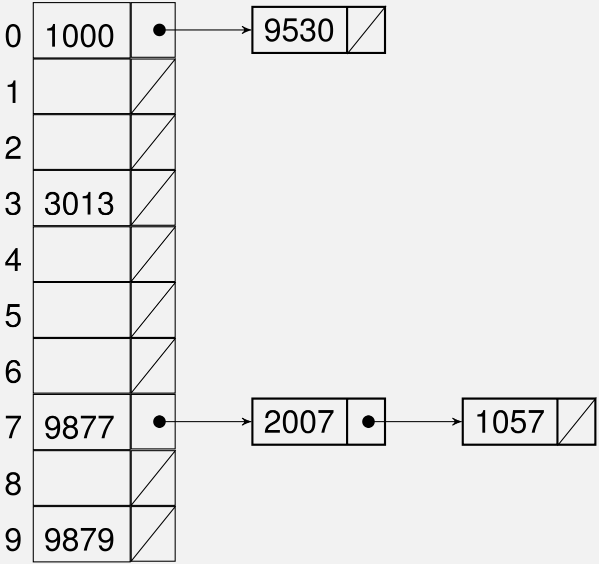


Collision Resolution Policies

- Open hashing
- Closed hashing
  - Bucket hashing
  - Open Addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Open Hashing

Use linked list to store synonyms

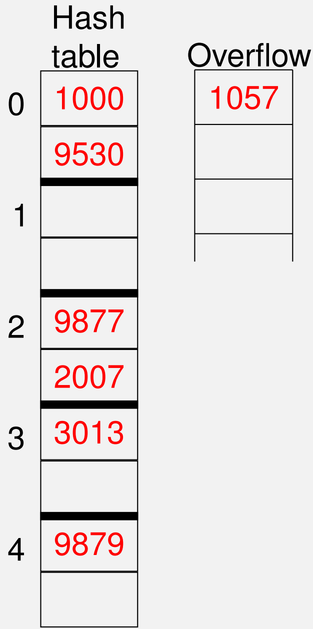


Bucket Hashing

- Hashing data to buckets that can hold multiple pieces of data.
- Each bucket has an address and collisions are postponed until the bucket is full.

Bucket Hashing Example

- Hash function: h(k) = k MOD 5
- Insert in the order: 9877, 2007, 1000, 9530,3013, 9879, and 1057



Open Addressing

- When a collision occurs, an unoccupied element is searched for placing the new element in.
- Normal hash function:

$$\underbrace{h: U}_{\text{set of keys}} \rightarrow \underbrace{\{0, \ldots, m\text{-}1\}}_{\text{addresses}}$$

- Open Addressing uses hash and probe function:

$$\underbrace{h: U}_{\text{set of keys}} \times \underbrace{\{0, 1, \ldots, m\text{-}1\}}_{\text{probe numbers}} \rightarrow \underbrace{\{0, \ldots, m\text{-}1\}}_{\text{addresses}}$$

Open Addressing

- There are different methods:
  - Linear probing
  - Quadratic probing
  - Double hashing

- When a home address is occupied, go to the next address (the current address + 1):

$$hp(k,i) = (h(k) + i) \text{ MOD } m$$

- Advantages:
  - quite simple to implement
  - data tend to remain near their home address (significant for disk addresses)
- Disadvantages:
  - produces primary clustering

Linear Probing Example

- Hash function: h(k) = k MOD 10
- hp(k,i) = (h(k) + i) MOD 10
- Insert 1001, 9050, 9877, 2037 and 1059

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	1059

Quadratic Probing

- The address increment is the collision probe number squared:

$$hp(k,i) = (h(k) + i^2) \text{ MOD } m$$

- Advantages:
  - works much better than linear probing
- Disadvantages:
  - time required to square numbers
  - produces secondary clustering

Quadratic Probing Example

- Hash function: A_9, B_9, C_9, D_{11}, E_{12},F_9
- hp(k,i) = (h(k) + i^2) MOD 23 + 1
- Insert A, B, C, D, E, F

									A	B	D	E	C					F				
[1]									[10]	[11]	[12]	[13]	[14]	[15]				[19]				[23]

- Using two hash functions:

$$hp(k,i) = (h_1(k) + i * h_2(k)) \text{ MOD } m$$

- Hash function 1: $h_1(k) = k \text{ MOD } 10$
- Hash function 2: $h_2(k) = 5 - (k \text{ MOD } 5)$
- $hp(k,i) = (h_1(k) + i * h_2(k)) \text{ MOD } 10$
- Insert 1001, 9050, 9877, 2037 and 1059
- $h_1(2037) = 7$, $h_2(2037) = 3$, $hp(k,1) = 0$, $hp(k,2) = 3$

0	9050
1	1001
2	
3	2037
4	
5	
6	
7	9877
8	
9	1059