Chapter 9: Hashing

- Basic concepts
- Hash functions
- Collision resolution
- Open addressing
- Linked list resolution
- Bucket hashing
Basic Concepts

• Sequential search: $O(n)$ Requiring several key comparisons
• Binary search: $O(\log_2 n)$ before the target is found
Basic Concepts

• Search complexity:

<table>
<thead>
<tr>
<th>Size</th>
<th>Binary</th>
<th>Sequential (Average)</th>
<th>Sequential (Worst Case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>4</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>50</td>
<td>6</td>
<td>25</td>
<td>50</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>1,000</td>
<td>10</td>
<td>500</td>
<td>1,000</td>
</tr>
<tr>
<td>10,000</td>
<td>14</td>
<td>5,000</td>
<td>10,000</td>
</tr>
<tr>
<td>100,000</td>
<td>17</td>
<td>50,000</td>
<td>100,000</td>
</tr>
<tr>
<td>1,000,000</td>
<td>20</td>
<td>500,000</td>
<td>1,000,000</td>
</tr>
</tbody>
</table>
Basic Concepts

• Is there a search algorithm whose complexity is \(O(1)\)?
Basic Concepts

• Is there a search algorithm whose complexity is $O(1)$?

YES.
Basic Concepts

Each key has only one address
Basic Concepts

Key

Vu Nguyen 102002
John Adams 107095
Sarah Trapp 111060

Hash Function

Address

005
100
002
001
007

001 Harry Lee
002 Sarah Trapp
005 Vu Nguyen
007 Ray Black
100 John Adams

Cao Hoang Tru
CSE Faculty - HCMUT

01 December 2008
Basic Concepts

- **Home address**: address produced by a hash function.

- **Prime area**: memory that contains all the home addresses.
Basic Concepts

• **Synonyms**: a set of keys that hash to the same location.

• **Collision**: the location of the data to be inserted is already occupied by the synonym data.
Basic Concepts

• **Ideal hashing:**
  – No location collision
  – Compact address space
Basic Concepts

Insert $A$, $B$, $C$

$\text{hash}(A) = 9$
$\text{hash}(B) = 9$
$\text{hash}(C) = 17$
Basic Concepts

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

Collision Resolution

B and A collide at 9
Basic Concepts

Insert A, B, C

hash(A) = 9
hash(B) = 9
hash(C) = 17

Collision Resolution

C and B collide at 9
B and A collide at 9
Basic Concepts

Search for $B$

$\text{hash}(A) = 9$

$\text{hash}(B) = 9$

$\text{hash}(C) = 17$
Hash Functions

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

• The address is the key itself:

\[
\text{hash}(\text{Key}) = \text{Key}
\]
Direct Hashing

- **Advantage**: there is no collision.
- **Disadvantage**: the address space (storage size) is as large as the key space.
Modulo Division

Address = 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:

379452 → 394
121267 → 112
378845 → 388
160252 → 102
045128 → 051
Mid-square

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

• Example: $\text{cuu duong than cong . com}$
  $9452 \times 9452 = 89340304 \rightarrow 3403$
Mid-square

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

\[
\begin{align*}
379452 & : 379 \times 379 = 143641 \rightarrow 364 \\
121267 & : 121 \times 121 = 014641 \rightarrow 464 \\
045128 & : 045 \times 045 = 002025 \rightarrow 202
\end{align*}
\]
Folding

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

Key = 123|456|789

```
fold shift
123 + 456 + 789 = 1368
⇒ 368
```
Folding

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

Key = 123|456|789

fold shift | fold boundary
123 + 456 + 789 = 1368 | 321 + 456 + 987 = 1764
⇒ 368 | ⇒ 764

Cao Hoang Tru
CSE Faculty - HCMUT
Rotation

- Hashing keys that are identical except for the last character may create synonyms.

- The key is rotated before hashing.

<table>
<thead>
<tr>
<th>original key</th>
<th>rotated key</th>
</tr>
</thead>
<tbody>
<tr>
<td>600101</td>
<td>160010</td>
</tr>
<tr>
<td>600102</td>
<td>260010</td>
</tr>
<tr>
<td>600103</td>
<td>360010</td>
</tr>
<tr>
<td>600104</td>
<td>460010</td>
</tr>
<tr>
<td>600105</td>
<td>560010</td>
</tr>
</tbody>
</table>
Rotation

- Used in combination with fold shift

<table>
<thead>
<tr>
<th>original key</th>
<th>rotated key</th>
</tr>
</thead>
<tbody>
<tr>
<td>600101 → 62</td>
<td>160010 → 26</td>
</tr>
<tr>
<td>600102 → 63</td>
<td>260010 → 36</td>
</tr>
<tr>
<td>600103 → 64</td>
<td>360010 → 46</td>
</tr>
<tr>
<td>600104 → 65</td>
<td>460010 → 56</td>
</tr>
<tr>
<td>600105 → 66</td>
<td>560010 → 66</td>
</tr>
</tbody>
</table>

Spreading the data more evenly across the address space
Pseudorandom

\[ y = ax + c \]

For maximum efficiency, \( a \) and \( c \) should be prime numbers.
Pseudorandom

• Example:

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

Address  = ((17*121267 + 7) MOD 307 + 1
= (2061539 + 7) MOD 307 + 1
= 2061546 MOD 307 + 1
= 41 + 1
= 42
Collision Resolution

• Except for the direct hashing, none of the others are one-to-one mapping
  ⇒ Requiring collision resolution methods

• Each collision resolution method can be used independently with each hash function
Collision Resolution

• A rule of thumb: a hashed list should not be allowed to become more than 75% full.

Load factor:

\[ \alpha = \left( \frac{k}{n} \right) \times 100 \]

- \( n \) = list size
- \( k \) = number of filled elements
Collision Resolution

• As data are added and collisions are resolved, hashing tends to cause data to group within the list.

⇒ **Clustering**: data are unevenly distributed across the list.

• High degree of clustering increases the number of probes to locate an element.

⇒ **Minimize** clustering.
Collision Resolution

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

Insert $A_9, B_9, C_9, D_{11}, E_{12}$
Collision Resolution

- **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

- Open addressing
- Linked list resolution
- Bucket hashing
Open Addressing

• When a collision occurs, an unoccupied element is searched for placing the new element in.
Open Addressing

- Hash function:

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

set of keys \quad \text{addresses}
Open Addressing

- Hash and probe function:

\[ hp: U \times \{0, \ldots, m - 1\} \rightarrow \{0, \ldots, m - 1\} \]

set of keys \hspace{1cm} probe numbers \hspace{1cm} addresses
**Open Addressing**

**Algorithm**  
`hashInsert (ref T <array>, val k <key>)`

Inserts key `k` into table `T`

1. `i = 0`
2. loop (i < m)
   1. `j = hp(k, i)`
   2. if (T[j] = nil)
      1. `T[j] = k`
      2. return j
   3. else
      1. `i = i + 1`
3. return error: "hash table overflow"

**End**  `hashInsert`
Open Addressing

**Algorithm**

```plaintext
hashSearch (val T <array>, val k <key>)

Searches for key k in table T

1   i = 0
2   loop (i < m)
    1   j = hp(k, i)
    2   if (T[j] = k)
        1       return j
    3   else if (T[j] = nil)
        1       return nil
    4   else
        1       i = i + 1
3   return nil
End hashSearch
```

Cao Hoang Tru
CSE Faculty - HCMUT

01 December 2008
Open Addressing

• There are different methods:
  – Linear probing
  – Quadratic probing
  – Double hashing
  – Key offset
Linear Probing

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

$$hp(k, i) = (h(k) + i) \ MOD \ m$$
Linear Probing

<table>
<thead>
<tr>
<th>Hash Function</th>
<th>002 Sarah Trapp (070918)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>003 Bryan Devaux (121267)</td>
</tr>
<tr>
<td></td>
<td>008 John Carver (378845)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>306 Tuan Ngo (160252)</td>
</tr>
<tr>
<td></td>
<td>307 Shouli Feldman (045128)</td>
</tr>
</tbody>
</table>

Cuo Duong Than Cong.com
https://fb.com/tailieudientucntt
Linear Probing

Hash Function

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Mary Dodd</td>
<td>(379452)</td>
</tr>
<tr>
<td>002</td>
<td>Sarah Trapp</td>
<td>(070918)</td>
</tr>
<tr>
<td>003</td>
<td>Bryan Devaux</td>
<td>(121267)</td>
</tr>
<tr>
<td>004</td>
<td>Harry Eagle</td>
<td>(166702)</td>
</tr>
<tr>
<td>008</td>
<td>John Carver</td>
<td>(378845)</td>
</tr>
<tr>
<td>306</td>
<td>Tuan Ngo</td>
<td>(160252)</td>
</tr>
<tr>
<td>307</td>
<td>Shouli Feldman</td>
<td>(045128)</td>
</tr>
</tbody>
</table>

Cao Hoang Tru
CSE Faculty - HCMUT

01 December 2008
Linear Probing

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

- **Disadvantages:**
  - produces primary clustering
Quadratic Probing

- The address increment is the collision probe number squared:

\[ hp(k, i) = (h(k) + i^2) \mod m \]
Quadratic Probing

- **Advantages:**
  - works much better than linear probing

- **Disadvantages:**
  - time required to square numbers
  - produces secondary clustering

\[ h(k_1) = h(k_2) \Rightarrow h_p(k_1, i) = h_p(k_2, i) \]
Double Hashing

• Using two hash functions:

\[ h_p(k, i) = (h_1(k) + ih_2(k)) \mod m \]
Key Offset

• The new address is a function of the collision address and the key.

\[
\text{offset} = \left[ \text{key} / \text{listSize} \right] \\
\text{newAddress} = (\text{collisionAddress} + \text{offset}) \mod \text{listSize}
\]
Key Offset

• The new address is a function of the collision address and the key.

\[
\text{offset} = \left[ \frac{\text{key}}{\text{listSize}} \right]
\]
\[
\text{newAddress} = (\text{collisionAddress} + \text{offset}) \mod \text{listSize}
\]

\[
\text{hp}(k, i) = (\text{hp}(k, i-1) + \left[ \frac{k}{m} \right]) \mod m
\]
Open Addressing

• Hash and probe function:

\[ hp: U \times \{0, \ldots, m - 1\} \rightarrow \{0, \ldots, m - 1\} \]

set of keys → probe numbers → addresses

\[ \langle hp(k,0), hp(k,1), \ldots, hp(k,m-1) \rangle \] is a permutation of \( \langle 0, 1, \ldots, m-1 \rangle \)
Linked List Resolution

- **Major disadvantage of Open Addressing**: each collision resolution increases the probability for future collisions.

  ⇒ use **linked lists** to store synonyms
Linked List Resolution

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Mary Dodd</td>
<td>379452</td>
</tr>
<tr>
<td>002</td>
<td>Sarah Trapp</td>
<td>070918</td>
</tr>
<tr>
<td>003</td>
<td>Bryan Devaux</td>
<td>121267</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>008</td>
<td>John Carver</td>
<td>378845</td>
</tr>
<tr>
<td>306</td>
<td>Tuan Ngo</td>
<td>160252</td>
</tr>
<tr>
<td>307</td>
<td>Shouli Feldman</td>
<td>045128</td>
</tr>
</tbody>
</table>

Harry Eagle (166702)
Chris Walljasper (572556)

overflow area

prime area
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

<table>
<thead>
<tr>
<th>Bucket</th>
<th>Name</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Mary Dodd</td>
<td>379452</td>
</tr>
<tr>
<td>002</td>
<td>Sarah Trapp</td>
<td>070918</td>
</tr>
<tr>
<td></td>
<td>Harry Eagle</td>
<td>166702</td>
</tr>
<tr>
<td></td>
<td>Ann Georgis</td>
<td>367173</td>
</tr>
<tr>
<td>003</td>
<td>Bryan Devaux</td>
<td>121267</td>
</tr>
<tr>
<td></td>
<td>Chris Walljasper</td>
<td>572556</td>
</tr>
<tr>
<td>307</td>
<td>Shouli Feldman</td>
<td>045128</td>
</tr>
</tbody>
</table>

**linear probing**
Indexing = Hashing