Searching and Hash structure

Data Structures and Algorithms

Dept. Computer Science
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology, VNU-HCM

Overview

SEARCHING ALGORITHMS

Searching algorithms

Definition

Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Based on the type of search operation, these algorithms are generally classified into two categories:

1. **Sequential Search**: In this, the list or array is traversed sequentially and every element is checked.
2. **Interval Search**: These algorithms are specifically designed for searching in sorted data-structures.

Linear Search

Approach

1. Start from the leftmost element of list and one by one compare `x` with each element of list.
2. If `x` matches with an element, return the index.
3. If `x` doesn't match with any of elements, return -1.

Linear Search

Approach

1. Start from the leftmost element of list and one by one compare `x` with each element of list.
2. If `x` matches with an element, return the index.
3. If `x` doesn't match with any of elements, return -1.

The **time complexity** of the above algorithm is $O(n)$.

Binary Search

Approach

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array.

1. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half, otherwise narrow it to the upper half.
2. Repeatedly check until the value is found or the interval is empty.

Binary Search

Approach

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array.

1. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half, otherwise narrow it to the upper half.
2. Repeatedly check until the value is found or the interval is empty.

Implemetation:

- Recursive
- Iterative

Time complexity: $O(\log n)$

Jump Search

Approach

Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

Jump Search

Approach

Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

Suppose that an array arr of size n divided to some blocks with fixed size m.

1. Find the block k such that the first element of block k is less than key and the first element of block $k+1$ is greater than or equals to key.
2. Perform the linear search on the block k.

Jump Search

Approach

Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

Suppose that an array arr of size n divided to some blocks with fixed size m.

1. Find the block k such that the first element of block k is less than key and the first element of block $k+1$ is greater than or equals to key.
2. Perform the linear search on the block k.

Time complexity: $\sqrt{n}$.

Jump Seach

Consider the following array:
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610].
Length of the array is 16, block size is 4.

1. Jump from index 0 to index 4.
2. Jump from index 4 to index 8.
3. Jump from index 8 to index 12.
4. Since the element at index 12 is greater than 55 we will jump back a step to come to index 8.
5. Do linear search from index 8 to get the element 55.

Interpolation Seach

Approach

Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed.

1. Calculate the value of pos using the probe position formula.

$$\texttt{pos} = \texttt{lo} + \frac{(x - \texttt{arr}[\texttt{lo}]) \times (\texttt{hi} - \texttt{lo})}{\texttt{arr}[\texttt{hi}] - \texttt{arr}[\texttt{lo}]}$$

   where

   - arr: array where elements need to be searched.
   - x: element to be searched.
   - lo: starting index in arr.
   - hi: ending index in arr.

Interpolation Seach

Approach

Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed.

1. Calculate the value of `pos` using the probe position formula.
2. If it is a match, return the index of the item, and exit.

Time complexity:

- If elements are uniformly distributed, then $O(\log \log n))$.
- In worst case, it can take up to $O(n)$.

Interpolation Seach

Approach

Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed.

1. Calculate the value of `pos` using the probe position formula.
2. If it is a match, return the index of the item, and exit.
3. If the item is less than `arr[pos]`, calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.

Time complexity:

- If elements are uniformly distributed, then $O(\log \log n))$.
- In worst case, it can take up to $O(n)$.

Interpolation Seach

Approach

Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed.

1. Calculate the value of `pos` using the probe position formula.
2. If it is a match, return the index of the item, and exit.
3. If the item is less than `arr[pos]`, calculate the probe position of the left sub-array. Otherwise calculate the same in the right sub-array.
4. Repeat until a match is found or the sub-array reduces to zero.

Time complexity:

- If elements are uniformly distributed, then $O(\log \log n))$.
- In worst case, it can take up to $O(n)$.

Basic concepts

Basic concepts

- Sequential search: $O(n)$
- Binary search: $O(\log_2 n)$

$\rightarrow$ Requiring several key comparisons before the target is found.

Basic concepts

Search complexity:

Size	Binary	Sequential (Average)	Sequential (Worst Case)
16	4	8	16
50	6	25	50
256	8	128	256
1,000	10	500	1,000
10,000	14	5,000	10,000
100,000	17	50,000	100,000
1,000,000	20	500,000	1,000,000

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

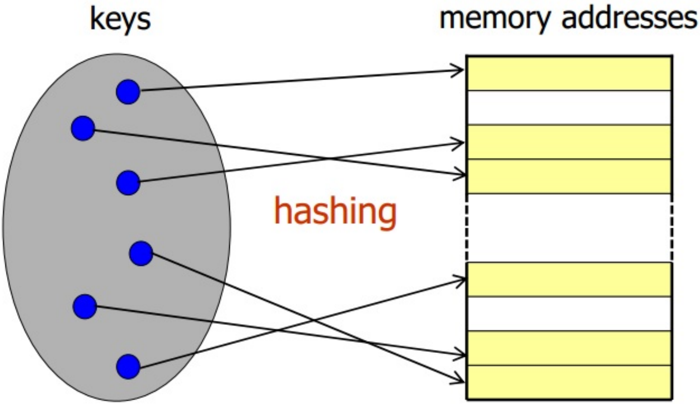
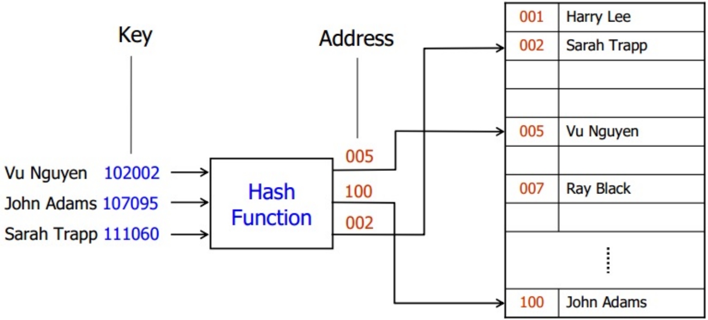


Figure: Each key has only one address

Basic concepts



Basic concepts

- Home address: address produced by a hash function.
- Prime area: memory that contains all the home addresses.

Basic concepts

- Home address: address produced by a hash function.
- Prime area: memory that contains all the home addresses.
- 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

- Home address: address produced by a hash function.
- Prime area: memory that contains all the home addresses.
- 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.
- Ideal hashing:
  - No location collision
  - Compact address space

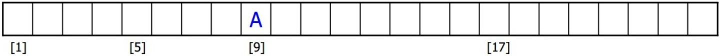
Basic concepts

Insert A, B, C

hash(A) = 9

hash(B) = 9

hash(C) = 17



Basic concepts

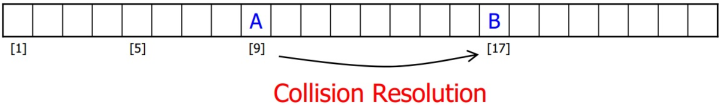
Insert A, B, C

hash(A) = 9

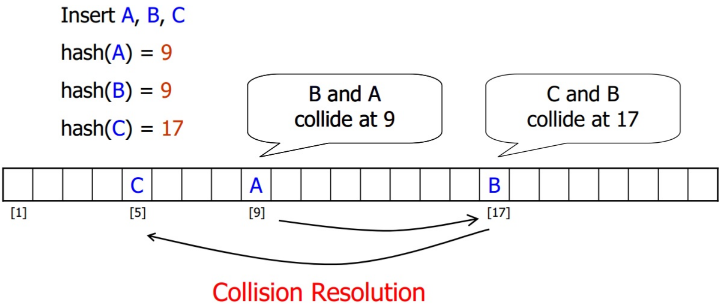
hash(B) = 9

hash(C) = 17

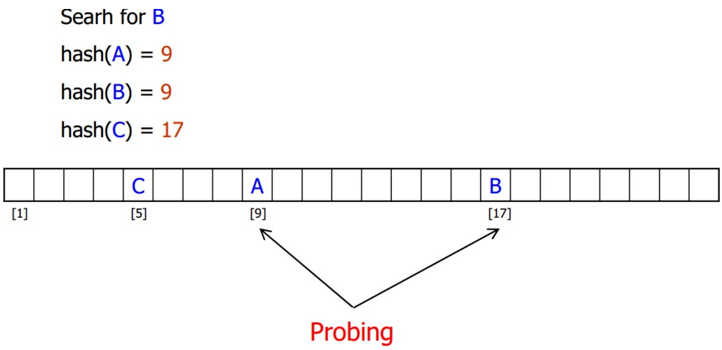
B and A
collide at 9



Basic concepts



Basic concepts



Hash functions

Hash functions

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

Direct Hashing

The address is the key itself:
$hash(Key) = Key$

Direct Hashing

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

Modulo division

$$Address = Key \ mod \ listSize$$

- 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
  $hash(121267) = 121267 \ mod \ 307 = 2$

Digit extraction

$$Address = selected\ digits\ from\ Key$$

Example:

379452→394

121267→112

378845→388

160252→102

045128→051

Mid-square

$$Address = middle\ digits\ of\ Key^2$$

Example:

9452 * 9452 = 89340304 → 3403

Mid-square

- Disadvantage: the size of the Key^2 is too large.
- Variations: use only a portion of the key.
  Example:
  379452: 379 * 379 = 143641 $\rightarrow$ 364
  121267: 121 * 121 = 014641 $\rightarrow$ 464
  045128: 045 * 045 = 002025 $\rightarrow$ 202

Folding

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

Example:

Key = 123—456—789

fold shift

$123 + 456 + 789 = 1368$

$\rightarrow 368$

Folding

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

Example:

Key = 123—456—789

fold shift

$123 + 456 + 789 = 1368$

$\rightarrow 368$

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.

original key	rotated key
600101	160010
600102	260010
600103	360010
600104	460010
600105	560010

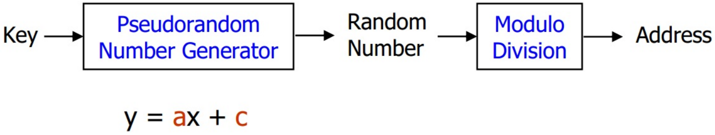
Rotation

- Used in combination with fold shift.

original key	rotated key
600101 → 62	160010 → 26
600102 → 63	260010 → 36
600103 → 64	360010 → 46
600104 → 65	460010 → 56
600105 → 66	560010 → 66

Spreading the data more evenly across the address space.

Pseudo-random



For maximum efficiency, a and c should be prime numbers.

Pseudo-random

Example:

Key = 121267

a = 17

c = 7

listSize = 307

Address = ((17*121267 + 7) mod 307

= (2061539 + 7) mod 307

= 2061546 mod 307

= 41

Collision resolution

Collision resolution

- 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

Collision resolution

- Closed Hashing
  - Open addressing
  - Bucket hashing
- Open Hashing
  - Linked list resolution

Open addressing

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

Open addressing

Hash function:

$$h : U \rightarrow \{0, 1, 2, ..., m - 1\}$$

set of keys addresses

Open addressing

Hash and probe function:

$hp : U \times \{0, 1, 2, ..., m-1\} \rightarrow \{0, 1, 2, ..., m-1\}$

set of keys probe numbers addresses

Open Addressing

```
Algorithm hashInsert(ref T ¡array¿, val k ¡key¿)
Inserts key k into table T.

i = 0
while i ¡ m do
    j = hp(k, i)
    if T[j] = nil then
        T[j] = k
        return j
    else
        i = i + 1
    end
end
return error: "hash table overflow"
End hashInsert
```

Open Addressing

```
Algorithm hashSearch(val T ¡array¿, val k ¡key¿)
Searches for key k in table T.

i = 0
while i ¡ m do
    j = hp(k, i)
    if T[j] = k then
        return j
    else if T[j] = nil then
        return nil
    else
        i = i + 1
    end
end
return nil
End hashSearch
```

Open Addressing

There are different methods:

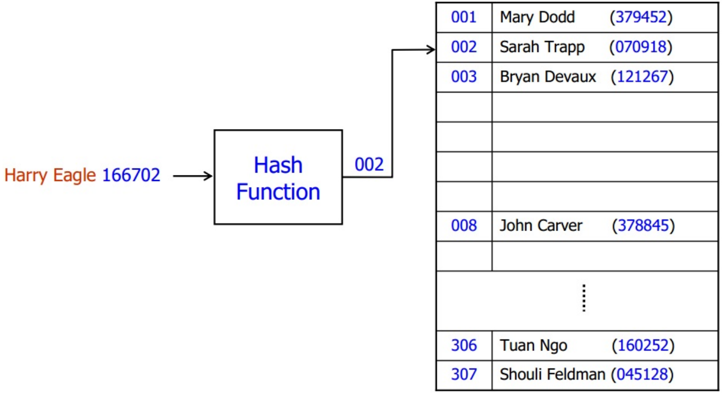
- Linear probing
- Quadratic probing
- Double hashing
- Key offset

Linear Probing

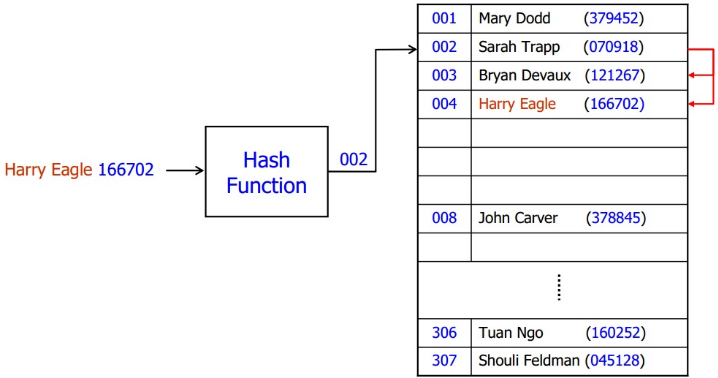
- When a home address is occupied, go to the next address (the current address + 1):
  $hp(k, i) = (h(k) + i)\ mod\ m$

Linear Probing

- 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



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
    $h(k_1) = h(k_2) \rightarrow hp(k_1, i) = hp(k_2, i)$

Double Hashing

- Using two hash functions:
  $hp(k, i) = (h_1(k) + ih_2(k)) \; mod \; m$

Key Offset

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

$$offset = [key/listSize]$$
$$newAddress = (collisionAddress + offset) \bmod listSize$$

Key Offset

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

$$offset = [key/listSize]$$
$$newAddress = (collisionAddress + offset) \ mod \ listSize$$

$$hp(k, i) = (hp(k, i - 1) + [k/m]) \ mod \ m$$

Open addressing

Hash and probe function:
$hp : U \times \{0, 1, 2, ..., m-1\} \rightarrow \{0, 1, 2, ..., m-1\}$

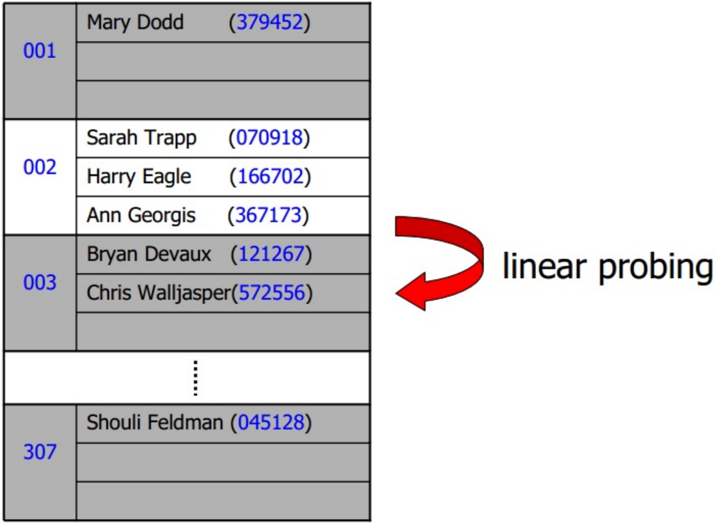
set of keys probe numbers addresses

$\{hp(k, 0), hp(k, 1), \ldots, hp(k, m-1)\}$ is a permutation of $\{0, 1, \ldots, m-1\}$

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.

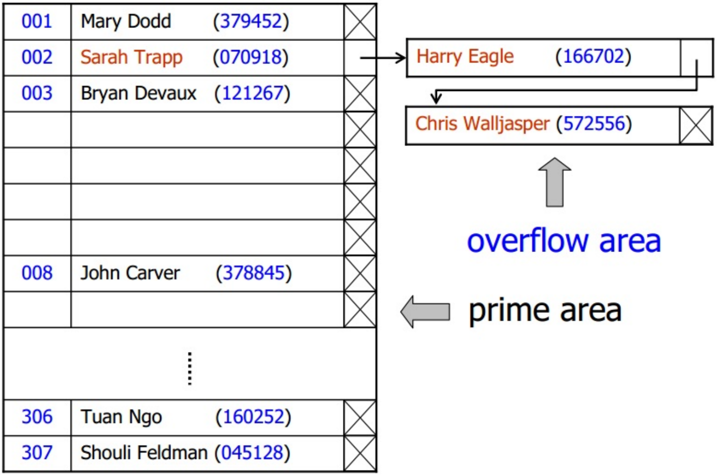
Bucket hashing



Linked List Resolution

- Major disadvantage of Open Addressing: each collision resolution increases the probability for future collisions.
  $\rightarrow$ use linked lists to store synonyms

Linked list resolution



THANK YOU.