Searching and Hash structure

Data Structures and Algorithms

Tran Ngoc Bao Duy
Faculty of Computer Science and Engineering
Ho Chi Minh University of Technology, VNU-HCM

Overview

1 Searching algorithms
- Sequential Search
- Interval Search

2 Hash table
- Basic concepts
- Direct-address tables
- Hash tables
- Hash functions
- Open addressing

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.

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

   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 $O(n)$.

Interpolation Search

Search the element 18 in array [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47].

1. Calculate the probe position:

$$\texttt{pos} = 0 + \frac{(18 - 10) \times (14 - 0)}{47 - 10} = 3$$

Interpolation Search

Search the element 18 in array [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47].

1. Calculate the probe position:

$$\texttt{pos} = 0 + \frac{(18 - 10) \times (14 - 0)}{47 - 10} = 3$$

2. arr[3] $= 16 > 18 = x \Rightarrow$ Search from $\texttt{index} = 4$ to the end $\texttt{index} = 14$.

Interpolation Search

Search the element 18 in array [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47].

1. Calculate the probe position:

$$\texttt{pos} = 0 + \frac{(18 - 10) \times (14 - 0)}{47 - 10} = 3$$

2. arr[3] $= 16 > 18 = x \Rightarrow$ Search from $\texttt{index} = 4$ to the end $\texttt{index} = 14$.
3. Calculate the probe position:

$$\texttt{pos} = 4 + \frac{(18 - 18) \times (14 - 4)}{47 - 18} = 4$$

Interpolation Search

Search the element 18 in array [10, 12, 13, 16, 18, 19, 20, 21, 22, 23, 24, 33, 35, 42, 47].

1. Calculate the probe position:

$$\texttt{pos} = 0 + \frac{(18-10)\times(14-0)}{47-10} = 3$$

2. arr[3] $= 16 > 18 = x \Rightarrow$ Search from $\texttt{index} = 4$ to the end $\texttt{index} = 14$.
3. Calculate the probe position:

$$\texttt{pos} = 4 + \frac{(18-18)\times(14-4)}{47-18} = 4$$

4. arr[4] $= 18 = x \Rightarrow$ End.

HASH TABLE

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

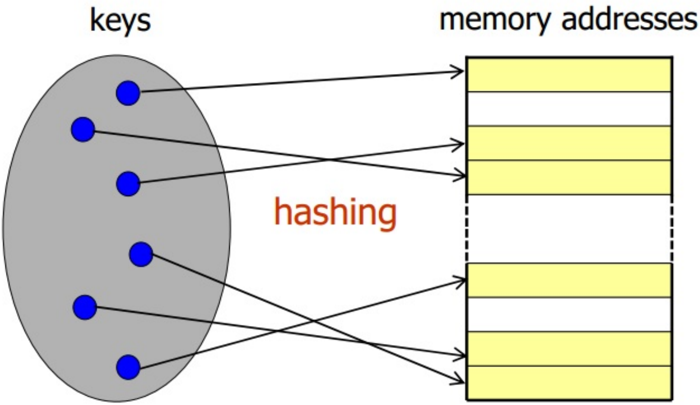
Is there a search algorithm whose complexity is $O(1)$?

Basic concepts

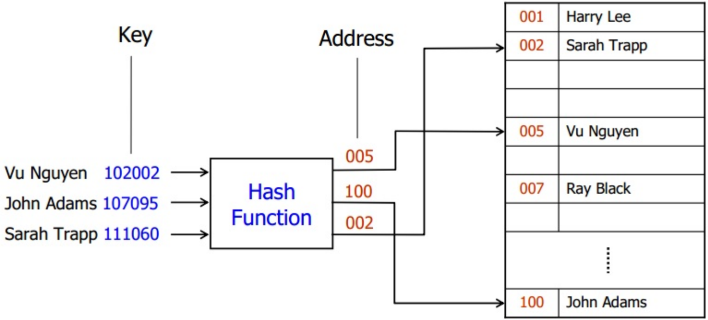
Is there a search algorithm whose complexity is $O(1)$?

YES

Basic concepts



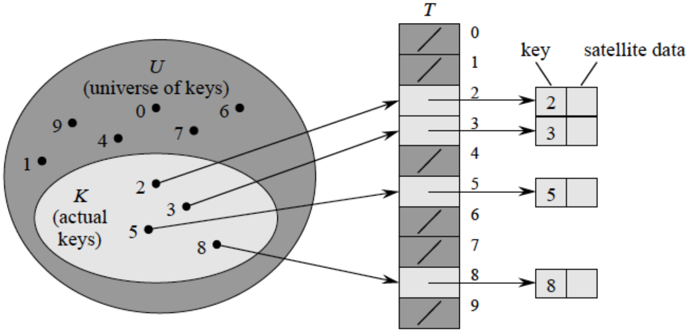
Basic concepts



Direct-address tables

Definition

Direct-address table is a structure using an array denoted by $T[0 \ldots m-1]$, in which each position, or **slot**, corresponds to a key in the universe U.



(Source: Introduction to algorithms - 3rd edition)

Direct-address tables

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] = \text{NIL}$

Direct-address tables

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] = \text{NIL}$

Each of these operations takes only $O(1)$ time.

Downside of direct addressing

- If the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible.
- The set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

Hash tables

Definition

Hash table is a dynamic set that supports only the dictionary operations `INSERT`, `SEARCH`, and `DELETE` with the average time to search for an element $O(1)$ but requires much less storage than the universe U of all possible keys.

Hash tables

Definition

Hash table is a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE with the average time to search for an element $O(1)$ but requires much less storage than the universe U of all possible keys.

- With direct addressing, an element with key k is stored in slot k.
- With hashing, this element is stored in slot $h(k)$. **Hash function** h computes the slot from the key k that maps the universe keys U into the slots of a hash table $T[0 \dots m-1]$:

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

where $m \ll |U|$.

Hash tables

Definition

Hash table is a dynamic set that supports only the dictionary operations `INSERT`, `SEARCH`, and `DELETE` with the average time to search for an element $O(1)$ but requires much less storage than the universe U of all possible keys.

- With direct addressing, an element with key k is stored in slot k.
- With hashing, this element is stored in slot $h(k)$. **Hash function** h computes the slot from the key k that maps the universe keys U into the slots of a hash table $T[0 \ldots m-1]$:

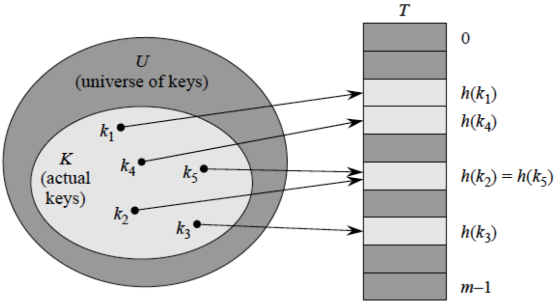
$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

where $m \ll |U|$.

Hash tables

Definition

Hash table is a dynamic set that supports only the dictionary operations `INSERT`, `SEARCH`, and `DELETE` with the average time to search for an element $O(1)$ but requires much less storage than the universe U of all possible keys.



(Source: Introduction to algorithms - 3rd edition)

Collision

There is one hitch: two keys may hash to the same slot. We call this situation a **collision**. Fortunately, we have effective techniques for resolving the conflict created by collisions.

Collision

There is one hitch: two keys may hash to the same slot. We call this situation a **collision**. Fortunately, we have effective techniques for resolving the conflict created by collisions.

- Ideal solution: avoid collisions altogether, try to achieve this goal by choosing a suitable hash function h.

Collision

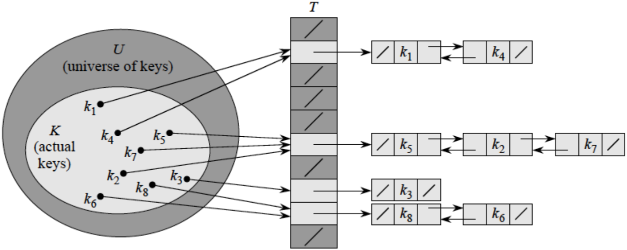
There is one hitch: two keys may hash to the same slot. We call this situation a **collision**. Fortunately, we have effective techniques for resolving the conflict created by collisions.

- Ideal solution: avoid collisions altogether, try to achieve this goal by choosing a suitable hash function h.
- Simplest solution: collision resolution by chaining.

Collision resolution by chaining

Definition

In **chaining**, we place all the elements that hash to the same slot into the same linked list.



(Source: Introduction to algorithms - 3rd edition)

Collision resolution by chaining

Definition

In **chaining**, we place all the elements that hash to the same slot into the same linked list.

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

Collision resolution by chaining

Definition

In **chaining**, we place all the elements that hash to the same slot into the same linked list.

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

The worst-case behavior of hashing with chaining is terrible: all n keys hash to the same slot, creating a list of length n.

HASH FUNCTIONS

Interpreting keys as natural numbers

- Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- If the keys are not natural numbers, we find a way to interpret them as natural numbers.

Interpreting keys as natural numbers

- Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- If the keys are not natural numbers, we find a way to interpret them as natural numbers.

Example

Interpret the key in string `pt` as natural number.

Interpreting keys as natural numbers

- Most hash functions assume that the universe of keys is the set $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- If the keys are not natural numbers, we find a way to interpret them as natural numbers.

Example

Interpret the key in string pt as natural number.
Since $\mathtt{p} = 112, \mathtt{t} = 116$, express it as a radix-128 integet. It becomes $(112 \times 128) + 116 = 14452$

A good hash function

A good hash function satisfies (approximately) the assumption of simple uniform hashing (SUHA): each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.

A good hash function

A good hash function satisfies (approximately) the assumption of simple uniform hashing (SUHA): each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.

- Division method
- Multiplication method
- Universal hashing
- Other methods

Division method

Definition

In the **division method for** creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \mod m$$

Division method

Definition

In the **division method for** creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \mod m$$

Example

If the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$

Division method

Definition

In the **division method for** creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \mod m$$

Example

If the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$

Value of m:

- m should not be a power of 2.
- m should be a prime not too close to an exact power of 2.

Multiplication method

Definition

The multiplication method for creating hash functions operates in two steps:

1. Multiply the key k by a constant A in the range $(0, 1)$ and extract the fractional part of kA.
2. Multiply this value by m and take the floor of the result.

In short, the hash function is

$$h(k) = \lfloor m\,(kA \mod 1) \rfloor$$

where $kA \mod 1$ means $kA - \lfloor kA \rfloor$.

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 $\rightarrow$ 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

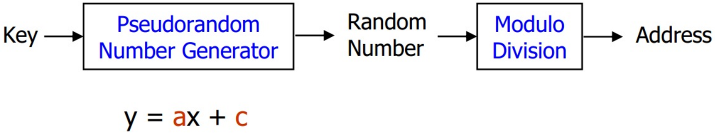
600103 → 64   360010 → 46

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

Open addressing

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

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

For every key k, the **probe sequence**:
$< h(k, 0), h(k, 1), \ldots, h(k, m-1) >$

Open addressing

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

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

For every key k, the **probe sequence**:
$< h(k, 0), h(k, 1), \ldots, h(k, m-1) >$
be a permutation of $< 0, 1, \ldots, m-1 >$

Open Addressing

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

i = 0
while i < m do
    j = h(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 = h(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
```

Linear probing

Given an ordinary hash function $h' : U \to \{0, 1, 2, ..., m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

$$h(k, i) = \left(h'(k) + ci\right) \mod m$$

where c is positive auxiliary constant.

Linear probing

Given an ordinary hash function $h' : U \rightarrow \{0, 1, 2, ..., m-1\}$, which we refer to as an **auxiliary hash function**, the method of **linear probing** uses the hash function

$$h(k, i) = \big(h'(k) + ci\big) \mod m$$

where c is positive auxiliary constant.

Example

$h'(k) = k \mod 11$
$h(k, i) = (h'(k) + i) \mod m$
Insert 10, 22, 31, 4, 15, 28, 17, 88, 59 into hash table.

Linear Probing

Insert 10.
$h(10, 0) = (h'(10) + 0) \mod 11 = 10.$

0	1	2	3	4	5	6	7	8	9	10
										10

Linear Probing

Insert 22.

$h(22, 0) = (h'(22) + 0) \mod 11 = 0.$

0	1	2	3	4	5	6	7	8	9	10
22										10

Linear Probing

Insert 31.

$h(31,0) = (h'(31) + 0) \mod 11 = 9.$

0	1	2	3	4	5	6	7	8	9	10
22									31	10

Linear Probing

Insert 4.
$h(4,0) = (h'(4) + 0) \mod 11 = 4.$

0	1	2	3	4	5	6	7	8	9	10
22				4					31	10

Linear Probing

Insert 15.

$h(15, 0) = (h'(15) + 0) \mod 11 = 4.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4					31	10

Linear Probing

Insert 15.
$h(15, 0) = (h'(15) + 0) \mod 11 = 4.$ Collission.
$h(15, 1) = (h'(15) + 1) \mod 11 = 5.$

0	1	2	3	4	5	6	7	8	9	10
22				4	15				31	10

Linear Probing

Insert 28.

$h(28, 0) = (h'(28) + 0) \mod 11 = 6.$

0	1	2	3	4	5	6	7	8	9	10
22				4	15	28			31	10

Linear Probing

Insert 17.

$h(17,0) = (h'(17) + 0) \mod 11 = 6$. Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4	15	28			31	10

Linear Probing

Insert 17.
$h(17, 0) = (h'(17) + 0) \mod 11 = 6$. Collission.
$h(17, 1) = (h'(17) + 1) \mod 11 = 7$.

0	1	2	3	4	5	6	7	8	9	10
22				4	15	28	17		31	10

Linear Probing

Insert 88.

$h(88, 0) = (h'(88) + 0) \mod 11 = 0.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4	15	28	17		31	10

Linear Probing

Insert 88.
$h(88, 0) = (h'(88) + 0) \mod 11 = 0$. Collission.
$h(88, 1) = (h'(88) + 1) \mod 11 = 1$.

0	1	2	3	4	5	6	7	8	9	10
22	88			4	15	28	17		31	10

Linear Probing

Insert 59.

$h(59, 0) = (h'(59) + 0) \mod 11 = 4.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22	88			4	15	28	17		31	10

Linear Probing

Insert 59.
$h(59,0) = (h'(59) + 0) \mod 11 = 4.$ Collission.
$h(59,1) = (h'(59) + 1) \mod 11 = 5.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22	88			4	15	28	17		31	10

Linear Probing

Insert 59.

$h(59, 0) = (h'(59) + 0) \mod 11 = 4$. Collission.

$h(59, 1) = (h'(59) + 1) \mod 11 = 5$. Collission.

$h(59, 2) = (h'(59) + 2) \mod 11 = 6$. Collission.

0	1	2	3	4	5	6	7	8	9	10
22	88			4	15	28	17		31	10

Linear Probing

Insert 59.

$h(59, 0) = (h'(59) + 0) \mod 11 = 4.$ Collission.
$h(59, 1) = (h'(59) + 1) \mod 11 = 5.$ Collission.
$h(59, 2) = (h'(59) + 2) \mod 11 = 6.$ Collission.
$h(59, 3) = (h'(59) + 3) \mod 11 = 7.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22	88			4	15	28	17		31	10

Linear Probing

Insert 59.

$h(59,0) = (h'(59) + 0) \mod 11 = 4.$ Collission.
$h(59,1) = (h'(59) + 1) \mod 11 = 5.$ Collission.
$h(59,2) = (h'(59) + 2) \mod 11 = 6.$ Collission.
$h(59,3) = (h'(59) + 3) \mod 11 = 7.$ Collission.
$h(59,4) = (h'(59) + 4) \mod 11 = 8.$

0	1	2	3	4	5	6	7	8	9	10
22	88			4	15	28	17	59	31	10

Linear probing: Evaluation

- Easy to implement.
- Suffers from a problem as **primary clustering**.

Quadratic probing

Quadratic probing uses a hash function of the form

$$h(k, i) = \left(h'(k) + c_1 i + c_2 i^2\right) \mod m$$

where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, and $i = 0, 1, \ldots, m-1$.

Quadratic probing

Quadratic probing uses a hash function of the form

$$h(k,i) = \left(h'(k) + c_1 i + c_2 i^2\right) \mod m$$

where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, and $i = 0, 1, \ldots, m-1$.

Example

$h'(k) = k \mod 11$
$c_1 = 1, c_2 = 3 \Rightarrow h(k,i) = (h'(k) + i + 3i^2) \mod 11$
Insert 10, 22, 31, 4, 15, 28, 17, 88, 59 into hash table.

Quadratic Probing

Insert 10.
$h(10, 0) = (h'(10) + 0) \mod 11 = 10.$

0	1	2	3	4	5	6	7	8	9	10
										10

Quadratic Probing

Insert 22.

$h(22, 0) = (h'(22) + 0) \mod 11 = 0.$

0	1	2	3	4	5	6	7	8	9	10
22										10

Quadratic Probing

Insert 31.

$h(31, 0) = (h'(31) + 0) \mod 11 = 9.$

0	1	2	3	4	5	6	7	8	9	10
22									31	10

Quadratic Probing

Insert 4.
$h(4,0) = (h'(4) + 0) \mod 11 = 4.$

0	1	2	3	4	5	6	7	8	9	10
22				4					31	10

Quadratic Probing

Insert 15.

$h(15, 0) = (h'(15) + 0) \mod 11 = 4.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4					31	10

Quadratic Probing

Insert 15.
$h(15, 0) = (h'(15) + 0) \mod 11 = 4$. Collission.
$h(15, 1) = (h'(15) + 1 + 3 \times 1^2) \mod 11 = 8$.

0	1	2	3	4	5	6	7	8	9	10
22				4				15	31	10

Quadratic Probing

Insert 28.

$h(28, 0) = (h'(28) + 0) \mod 11 = 6.$

0	1	2	3	4	5	6	7	8	9	10
22				4		28		15	31	10

Quadratic Probing

Insert 17.

$h(17, 0) = (h'(17) + 0) \mod 11 = 6.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4		28		15	31	10

Quadratic Probing

Insert 17.
$h(17, 0) = (h'(17) + 0) \mod 11 = 6$. Collission.
$h(17, 1) = (h'(17) + 1 + 3 \times 1^2) \mod 11 = 10$. Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4		28		15	31	10

Quadratic Probing

Insert 17.

$h(17, 0) = (h'(17) + 0) \mod 11 = 6.$ Collission.

$h(17, 1) = (h'(17) + 1 + 3 \times 1^2) \mod 11 = 10.$ Collission.

$h(17, 2) = (h'(17) + 2 + 3 \times 2^2) \mod 11 = 9.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4		28		15	31	10

Quadratic Probing

Insert 17.

$h(17,0) = (h'(17) + 0) \mod 11 = 6.$ Collission.

$h(17,1) = (h'(17) + 1 + 3 \times 1^2) \mod 11 = 10.$ Collission.

$h(17,2) = (h'(17) + 2 + 3 \times 2^2) \mod 11 = 9.$ Collission.

$h(17,2) = (h'(17) + 3 + 3 \times 3^2) \mod 11 = 3.$

0	1	2	3	4	5	6	7	8	9	10
22			17	4		28		15	31	10

Quadratic Probing

Insert 88.

$h(88, 0) = (h'(88) + 0) \mod 11 = 0$. Collission.

0	1	2	3	4	5	6	7	8	9	10
22			17	4		28		15	31	10

Quadratic Probing

Insert 88.

$h(88, 0) = (h'(88) + 0) \mod 11 = 0.$ Collission.

$h(88, 1) = (h'(88) + 1 + 3 \times 1^2) \mod 11 = 4.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22			17	4		28		15	31	10

Quadratic Probing

Insert 88.

$h(88,0) = (h'(88) + 0) \mod 11 = 0.$ Collission.

$h(88,1) = (h'(88) + 1 + 3 \times 1^2) \mod 11 = 4.$ Collission.

$h(88,2) = (h'(88) + 2 + 3 \times 2^2) \mod 11 = 3.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22			17	4		28		15	31	10

Quadratic Probing

Insert 88.

$h(88, 0) = (h'(88) + 0) \mod 11 = 0.$ Collission.

$h(88, 1) = (h'(88) + 1 + 3 \times 1^2) \mod 11 = 4.$ Collission.

$h(88, 2) = (h'(88) + 2 + 3 \times 2^2) \mod 11 = 3.$ Collission.

$h(88, 3) = (h'(88) + 3 + 3 \times 3^2) \mod 11 = 8.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22			17	4		28		15	31	10

Quadratic Probing

Insert 88.

$h(88, 0) = (h'(88) + 0) \mod 11 = 0.$ Collission.

$h(88, 1) = (h'(88) + 1 + 3 \times 1^2) \mod 11 = 4.$ Collission.

$h(88, 2) = (h'(88) + 2 + 3 \times 2^2) \mod 11 = 3.$ Collission.

$h(88, 3) = (h'(88) + 3 + 3 \times 3^2) \mod 11 = 8.$ Collission.

$h(88, 4) = (h'(88) + 4 + 3 \times 4^2) \mod 11 = 8.$ Collission.

.

.

.

0	1	2	3	4	5	6	7	8	9	10
22			17	4		28		15	31	10

Quadratic Probing

Insert 88.

$h(88, 0) = (h'(88) + 0) \mod 11 = 0.$ Collission.

$h(88, 1) = (h'(88) + 1 + 3 \times 1^2) \mod 11 = 4.$ Collission.

$h(88, 2) = (h'(88) + 2 + 3 \times 2^2) \mod 11 = 3.$ Collission.

$h(88, 3) = (h'(88) + 3 + 3 \times 3^2) \mod 11 = 8.$ Collission.

$h(88, 4) = (h'(88) + 4 + 3 \times 4^2) \mod 11 = 8.$ Collission.

.

.

.

$h(88, 8) = (h'(88) + 8 + 3 \times 8^2) \mod 11 = 2.$

0	1	2	3	4	5	6	7	8	9	10
22		88	17	4		28		15	31	10

Quadratic Probing

Insert 59.

⋮

$h(59, 2) = (h'(59) + 2 + 3 \times 2^2) \mod 11 = 7.$

0	1	2	3	4	5	6	7	8	9	10
22		88	17	4		28	59	15	31	10

Quadratic probing: Evaluation

- Easy to implement.
- Suffers from a problem as **secondary clustering**.

Double hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. **Double hashing** uses a hash function of the form

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m$$

where h_1 and h_2 are auxiliary hash functions.

Double hashing

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. **Double hashing** uses a hash function of the form

$$h(k, i) = (h_1(k) + ih_2(k)) \mod m$$

where h_1 and h_2 are auxiliary hash functions.

Example

$h_1(k) = k \mod 11$
$h_2(k) = 1 + (k \mod 10)$
$h(k, i) = (h_1(k) + ih_2(k)) \mod 11$
Insert 10, 22, 31, 4, 15, 28, 17, 88, 59 into hash table.

Double hashing

Insert 10.

$h(10, 0) = (h_1(10) + 0 \times h_2(10)) \mod 11 = 10.$

0	1	2	3	4	5	6	7	8	9	10
										10

Double hashing

Insert 22.

$h(22, 0) = (h_1(22) + 0 \times h_2(2)) \mod 11 = 0.$

0	1	2	3	4	5	6	7	8	9	10
22										10

Double hashing

Insert 31.
$h(31, 0) = (h_1(31) + 0 \times h_2(31)) \mod 11 = 9.$

0	1	2	3	4	5	6	7	8	9	10
22									31	10

Double hashing

Insert 4.

$$h(4, 0) = (h_1(4) + 0 \times h_2(4)) \mod 11 = 4.$$

0	1	2	3	4	5	6	7	8	9	10
22				4					31	10

Double hashing

Insert 15.
$h(15, 0) = (h_1(15) + 0 \times h_2(15)) \mod 11 = 4.$ Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4					31	10

Double hashing

Insert 15.

$h(15, 0) = (h_1(15) + 0 \times h_2(15)) \mod 11 = 4.$ Collission.

$h(15, 1) = (h_1(10) + 1 \times h_2(10)) \mod 11 = 10.$

Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4					31	10

Double hashing

Insert 15.

$h(15,0) = (h_1(15) + 0 \times h_2(15)) \mod 11 = 4.$ Collission.

$h(15,1) = (h_1(10) + 1 \times h_2(10)) \mod 11 = 10.$ Collission.

$h(15,2) = (h_1(10) + 2 \times h_2(10)) \mod 11 = 5.$

0	1	2	3	4	5	6	7	8	9	10
22				4	15				31	10

Double hashing

Insert 28.

$h(28, 0) = (h_1(28) + 0 \times h_2(28)) \mod 11 = 6.$

0	1	2	3	4	5	6	7	8	9	10
22				4	15	28			31	10

Double hashing

Insert 17.
$h(17, 0) = (h_1(17) + 0 \times h_2(17)) \mod 11 = 6.$
Collission.

0	1	2	3	4	5	6	7	8	9	10
22				4	15	28			31	10

Double hashing

Insert 17.

$h(17, 0) = (h_1(17) + 0 \times h_2(17)) \mod 11 = 6.$

Collission.

$h(17, 1) = (h_1(17) + 1 \times h_2(17)) \mod 11 = 2.$

0	1	2	3	4	5	6	7	8	9	10
22		17		4	15	28			31	10

Double hashing

Insert 88.
$h(88, 0) = (h_1(88) + 0 \times h_2(88)) \mod 11 = 0$.
Collission.

0	1	2	3	4	5	6	7	8	9	10
22		17		4	15	28			31	10

Double hashing

Insert 88.

$h(88, 0) = (h_1(88) + 0 \times h_2(88)) \mod 11 = 0.$

Collission.

$h(88, 1) = (h_1(88) + 1 \times h_2(88)) \mod 11 = 9.$

Collission.

0	1	2	3	4	5	6	7	8	9	10
22		17		4	15	28			31	10

Double hashing

Insert 88.

$h(88, 0) = (h_1(88) + 0 \times h_2(88)) \mod 11 = 0.$

Collission.

$h(88, 1) = (h_1(88) + 1 \times h_2(88)) \mod 11 = 9.$

Collission.

$h(88, 2) = (h_1(88) + 2 \times h_2(88)) \mod 11 = 7.$

0	1	2	3	4	5	6	7	8	9	10
22		17		4	15	28	88		31	10

Double hashing

Insert 59.
$h(59,0) = (h_1(59) + 0 \times h_2(59)) \mod 11 = 4.$
Collission.

0	1	2	3	4	5	6	7	8	9	10
22		17		4	15	28	88		31	10

Double hashing

Insert 59.

$h(59, 0) = (h_1(59) + 0 \times h_2(59)) \mod 11 = 4.$

Collission.

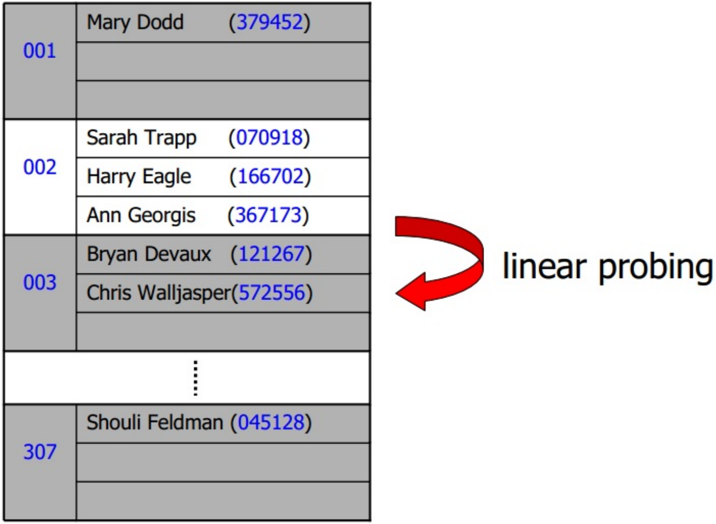
$h(59, 1) = (h_1(59) + 1 \times h_2(59)) \mod 11 = 3.$

0	1	2	3	4	5	6	7	8	9	10
22		17	59	4	15	28	88		31	10

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



001	Mary Dodd (379452)
002	Sarah Trapp (070918)
	Harry Eagle (166702)
	Ann Georgis (367173)
003	Bryan Devaux (121267)
	Chris Walljasper(572556)
⋮	
307	Shouli Feldman (045128)

THANK YOU.