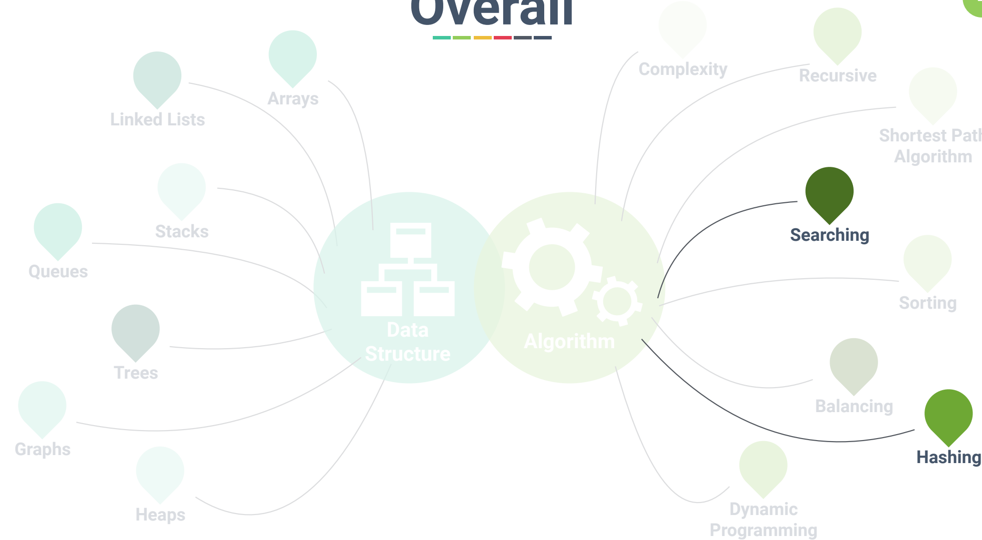
Chapter 9: Searching

Dr. Sirasit Lochanachit

Overall

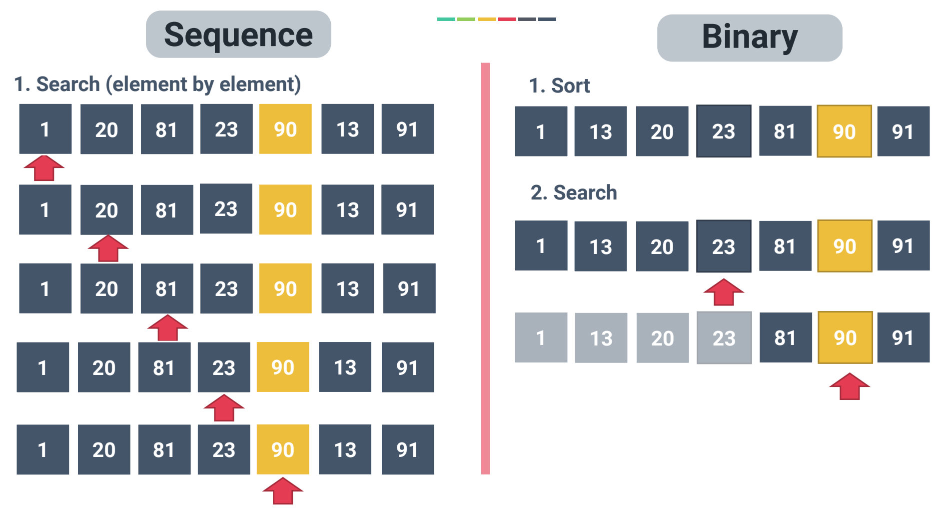


Outline

Searching:

- Sequential and binary search revisited
- Hashing:
  - Definition and method examples.
  - Collision resolutions

Searching in Array (Revisited)



Searching in Array (Revisited)

Sequence

1. Search (element by element)

Pseudocode: Sequential/linear search

```
linear_search (list, target_value)
  for each item in the list
    if item value == target_value
      return the item's location
    end if
  end for
  return 'no match'
END
```

Binary Search (Revisited)

- Binary Search
  - Locate a target value in a sequence of *n* elements that are sorted.
  - mid = (low + high) / 2
  - Initially, low = 0, high = n-1
- For instance, find number 5.

Data	1	5	7	9	10	11	20
Index	0	1	2	3	4	5	6

Binary Search (Revisited)

- Binary Search
  - If target value < data[mid], next interval is from low to mid-1.
  - If target value > data[mid], next interval is from mid + 1 to high.

	low			mid			high	
Data	1	5	7	9	10	11	20	mid = (0 + 6) / 2 = 3
Index	0	1	2	3	4	5	6	

	low	mid	high					
Data	1	5	7	9	10	11	20	mid = (0 + 2) / 2 = 1
Index	0	1	2	3	4	5	6	

Sequential and Binary Search

Sequential Search

- Easy to implement.
- Suitable for <u>unsorted sequence</u> or small array/list size.
- Search takes linear time - *O*(n)

Binary Search

- Suitable for <u>sorted sequence</u> with large array/list size.
- Search takes logarithmic time - *O*(log n)

What is Hashing?

Index	
0	(Data)
1	(Data)
2	(Data)
3	(Data)
4	(Data)
5	(Data)

(1) Array

Bucket	Hash Table	
0	(Key)	(Data)
1	(Key)	(Data)
2	(Key)	(Data)
3	(Key)	(Data)
4	(Key)	(Data)
5	(Key)	(Data)

(2) Hash table

Hashing is a technique that determines the index (bucket) using only a target search key, without the need to search.

Hashing Components

386 (Key) → Hash Function → 6 (Index)

Bucket	Hash Table	
0	(Keys)	(Data)
1	(Keys)	(Data)
2	(Keys)	(Data)
3	(Keys)	(Data)
4	(Keys)	(Data)
5	(Keys)	(Data)

There are two main components including

Hash Table or Bucket Array

- An array *A* of size *N*.
- Each cell of *A* is thought of as a "Bucket" that is, a collection of key-value pairs.
- Ideally, the keys are well distributed in the range [0, N-1] by a hash function.
- However, it is possible for two or more distinct keys to get mapped to the same index.

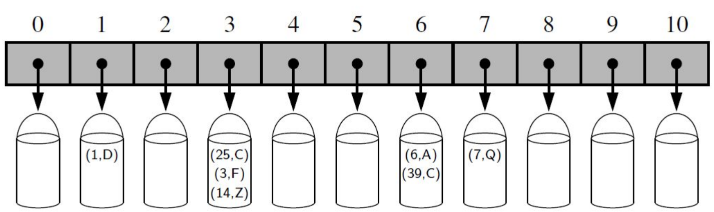
Hashing Components

386 (Key) → Hash Function → 6 (Index)

There are two main components including

Hash Table or Bucket Array

Bucket	Hash Table	
0	(Keys)	(Data)
1	(Keys)	(Data)
2	(Keys)	(Data)
3	(Keys)	(Data)
4	(Keys)	(Data)
5	(Keys)	(Data)



Hashing Components

386 (Key) → Hash Function → 6 (Index)

Bucket	Hash Table	
0	(Keys)	(Data)
1	(Keys)	(Data)
2	(Keys)	(Data)
3	(Keys)	(Data)
4	(Keys)	(Data)
5	(Keys)	(Data)

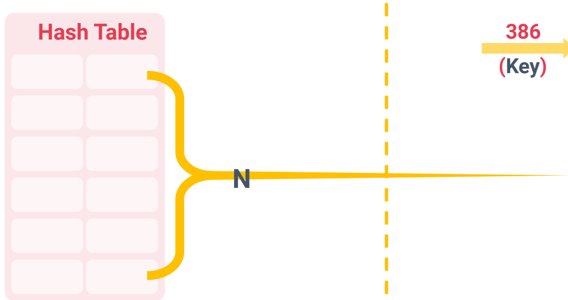
There are two main components including

Hash Function

- a function, *h*, that maps each key k to the corresponding integer index (bucket) in the hash table.

Hashing Process

Step 1: define the capacity of the hash table (*N*).



Step 2: define the hash function.

Basic Hash Function

Division method (or modulo arithmetic)

Hash value = k **mod** N

MAD method

Hash value = [(ak + b) **mod** p] **mod** N

Selection or mid-square method

Hash value = 2 digits of k^2

N = size of the bucket array
k = key of the item/element
p = prime number larger than N

a and b = random integers between [0, p - 1]

Division Method

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

Hash value = k **mod** N

Example:
Given a set of integer items {20, 19, 29, 10, 31, 3, 42, 14} and the capacity of the hash table is 11, calculate the hash values for each item.

Division Method

Index	Data
0	
1	42
2	
3	3, 14
4	
5	
6	
7	29
8	19
9	20, 31, 42
10	10

Hash value = k **mod** N

Example:
Given a set of integer items {20, 19, 29, 10, 31, 3, 42, 14} and the capacity of the hash table is 11, calculate the hash values for each item.

Division Method

Index	Data
0	
1	42
2	
3	3, 14
4	
5	
6	
7	29
8	19
9	20, 31, 42
10	10

Hash value = k **mod** N

To search for an item:

- Simply use the hash function to compute the bucket index for the item.
- Then, check the hash table to see if it exists.

It takes a constant time ***O*(1)** to compute the hash value and the index.

Highly likely to cause collision of hash values.

Collision

Index	Data
0	
1	42
2	
3	3, 14
4	
5	
6	
7	29
8	19
9	20, 31, 42
10	10

Hash value = k **mod** N

If two or more items produces the same hash value,

- They would need to be in the same bucket.
- This is known as **collision**.
- For example, there are three items in bucket #9.
- It makes the search operation to be more complex.

Division Method Exercise 1

Index	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	

Hash value = k **mod** N

Exercise 1:
Given a set of integer items {4, 1, 3, 10, 7, 5, 6} and the capacity of the hash table is 9, calculate the hash values for each item.

Division Method Exercise 2

Index	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	

Dec	Hex	Char	Dec	Hex	Char
64	40	@	96	60	`
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z
91	5B	[	123	7B	{
92	5C	\	124	7C	|
93	5D	]	125	7D	}
94	5E	^	126	7E	~
95	5F	_	127	7F	DEL

Hash value = k **mod** N

Exercise 2:
Given a set of character items {A, B, F, G, K, P, R, X, Y, Z} and the capacity of the hash table is set to 9, calculate the hash values for each item.

Collision Resolution

When two items hash to the same bucket or hash values are collided, a systematic method to find another spot for placing the second item in the hash table is called **collision resolution** or collision handling scheme.

Two fundamental method for resolving collisions are:

- First, use another available spot in the hash table.
- Second, change the structure of the hash table so that each array element can store more than one value.

Collision Resolution

- First, use another available spot in the hash table.

Linear Probing | Quadratic Probing | Double Hashing

Open addressing is the simple process to find the next empty bucket or address in the hash table.

It starts at the original hash value position and then move to the next hash value until the available bucket is found.

- Second, change the structure of the hash table so that each array element can store more than one value.

Open Addressing

When a collision occurs during the addition of an entry to a hash table, an **open addressing** scheme locates an alternate location in the hash table that is available. It can then be used to refer to the new entry.

Index	Data
0	20
1	
2	
3	
4	
5	34
6	19
7	
8	
9	

Index = 0

Data = 10

Index	Data
0	20
1	
2	
3	
4	
5	34
6	19
7	
8	
9	

10 ✖

Linear Probing

Linear probing method resolves a collision during hashing by looking at a sequential location, index by index, in the hash table that is available - starting at the original hash value. This search sequence is **probe sequence**.

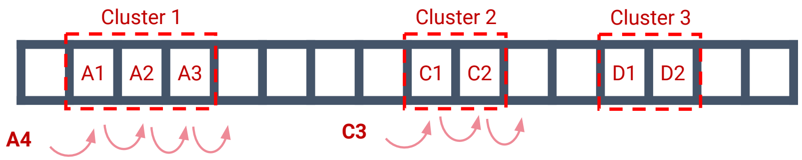
Index	Data
0	20
1	
2	
3	
4	
5	34
6	19
7	
8	
9	

Index = 0

Data = 10

Index	Data
0	20
1	10
2	
3	
4	
5	34
6	19
7	
8	
9	

Linear Probing



Although linear probing is a simple technique, it faces *a primary clustering problem*.

Collisions that are resolved with linear probing cause groups of consecutive locations (as so-called a cluster) in the hash table to be occupied.

Quadratic Probing

In order to avoid the primary clustering problem, the open addressing of the **quadratic probing** is used. The difference between linear and quadratic probing is that the new location is indicated by *hash+1* for the linear probing whereas it is indicated by *hash+j²* for the quadratic probing.

$$Rehash_{new} = (Hash_{old} + j^2) \bmod N$$

The number of the collision

Quadratic Probing

$$Rehash_{new} = (Hash_{old} + j^2) \bmod N$$

Index	Data
0	
1	
2	871
3	
4	
5	
6	
7	
8	415
9	
10	604

Index = 8
Data = 921

Index	Data
0	
1	
2	871
3	
4	
5	
6	
7	
8	415
9	921
10	604

Collision
$(8+1^2) = 9$

Index = 9
Data = 163

Index	Data
0	
1	
2	871
3	163
4	
5	
6	
7	
8	415
9	921
10	604

Collision #2
$(10+2^2) = 14$

Collision #1
$(9+1^2) = 10$

Quadratic Probing

Although the quadratic probing can avoid the primary clustering problem created by linear probing, it create its own kind of clustering, which is **secondary clustering**. Moreover, it adds more complexity for the running time. Specifically, it requires more time to compute the indices.

To avoid clustering with open addressing, we look into the other technique.

Collision Resolution

- First, use another available spot in the hash table.

Linear Probing | Quadratic Probing | Double Hashing

This choice sounds simple, but it can lead to several complications. Changing the structure of the hash table is not difficult and can be a better choice for resolving collisions than using an open addressing scheme.

- Second, change the structure of the hash table so that each array element can store more than one value.

Chaining

Chaining

Changing the structure of the hash table so that each array element can represent more than one value by using the **Linked List** structure.