Data structure and Programming (C++)

Hash Table
data structure



About array ....



- Assume that we have an array

```
int a[ ] = {4, 7, 10, 0, 9, 100, 50, 80, 25, 56, 90, 17};
```

We can use **for loop** to access data in this array.
Suppose that we want to search for a number and tell the user whether that number is exist or not.

A. Let's search for the number 90 in this array. How many times did you use the comparison operation?
B. Let's search for the number 10 in this array. How many times did you use the comparison operation?
C. Let's search for the number 99 in this array. How many times did you use the comparison operation?

→ By using **hash table**, we compare only 1 time in order to search.

Outline

- Time complexity
- What is Hashing? Hash table? Hash function?
- Collision
- Collision resolution techniques
  - Open hashing
  - Close hashing
- Examples

Time complexity

Time complexity is a time required for an algorithm to run.
It is denoted by a big-O notation: **O(n)**

- Best case
- Average case
- **Worst case**

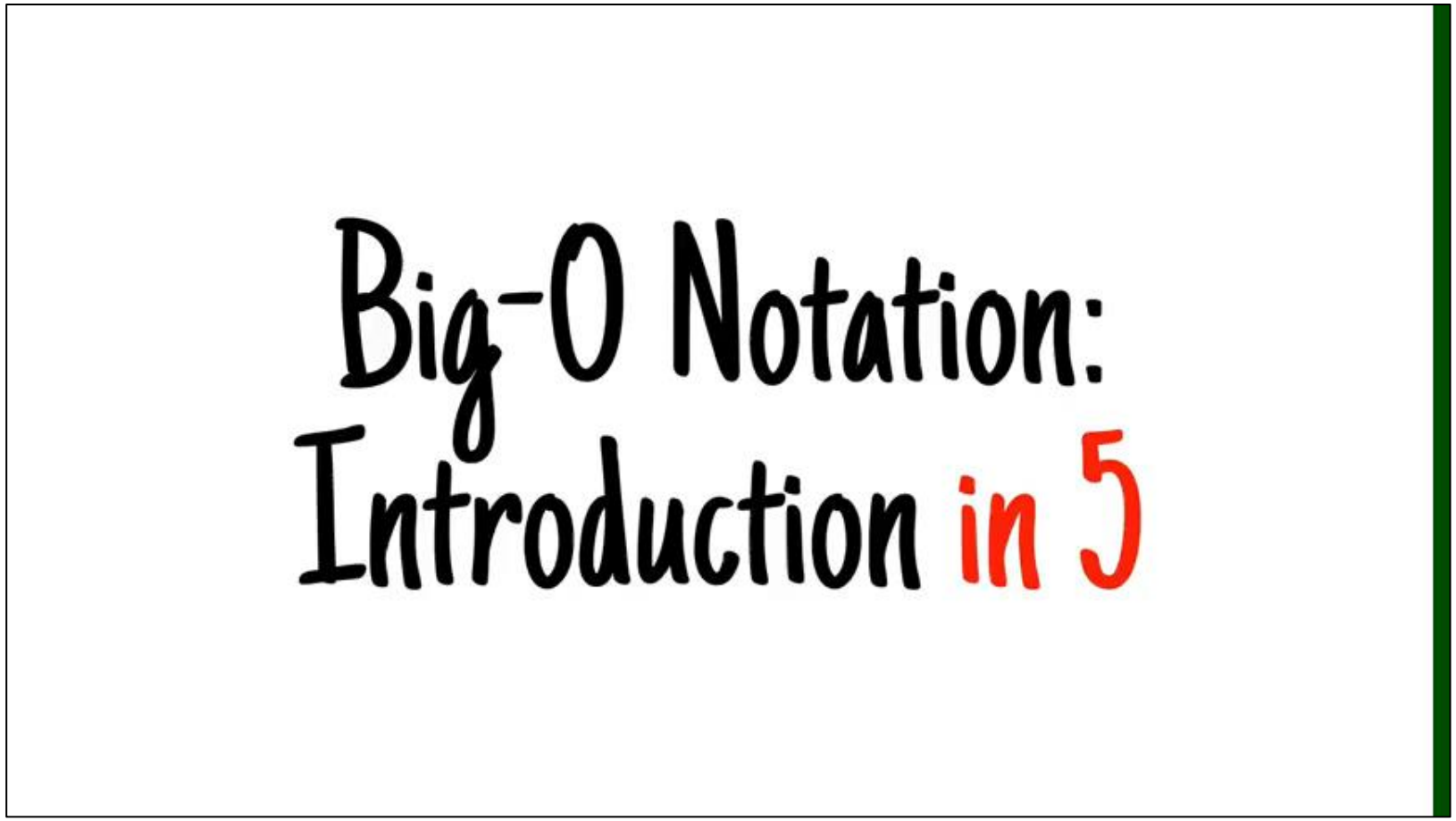
- Linked list : O(n)
- Array (unsorted) : O(n)
- Array (sorted) : O(log2 n)

Searching time complexity

Big-O Notation

❑ Introduction

- Introduction about Big-O Notation: bit.ly/312x9ou



Other Data structure

❑ Searching operation

- Linked list
- Array
- Stack
- Queue

What is the time required to search ?

O(n)

Hash Table

- Searching operation: Time Complexity

Hash Table : **O(1)**

General rules for time complexity

1. ignore constants

   $5n \rightarrow O(n)$

2. certain terms "dominate" others

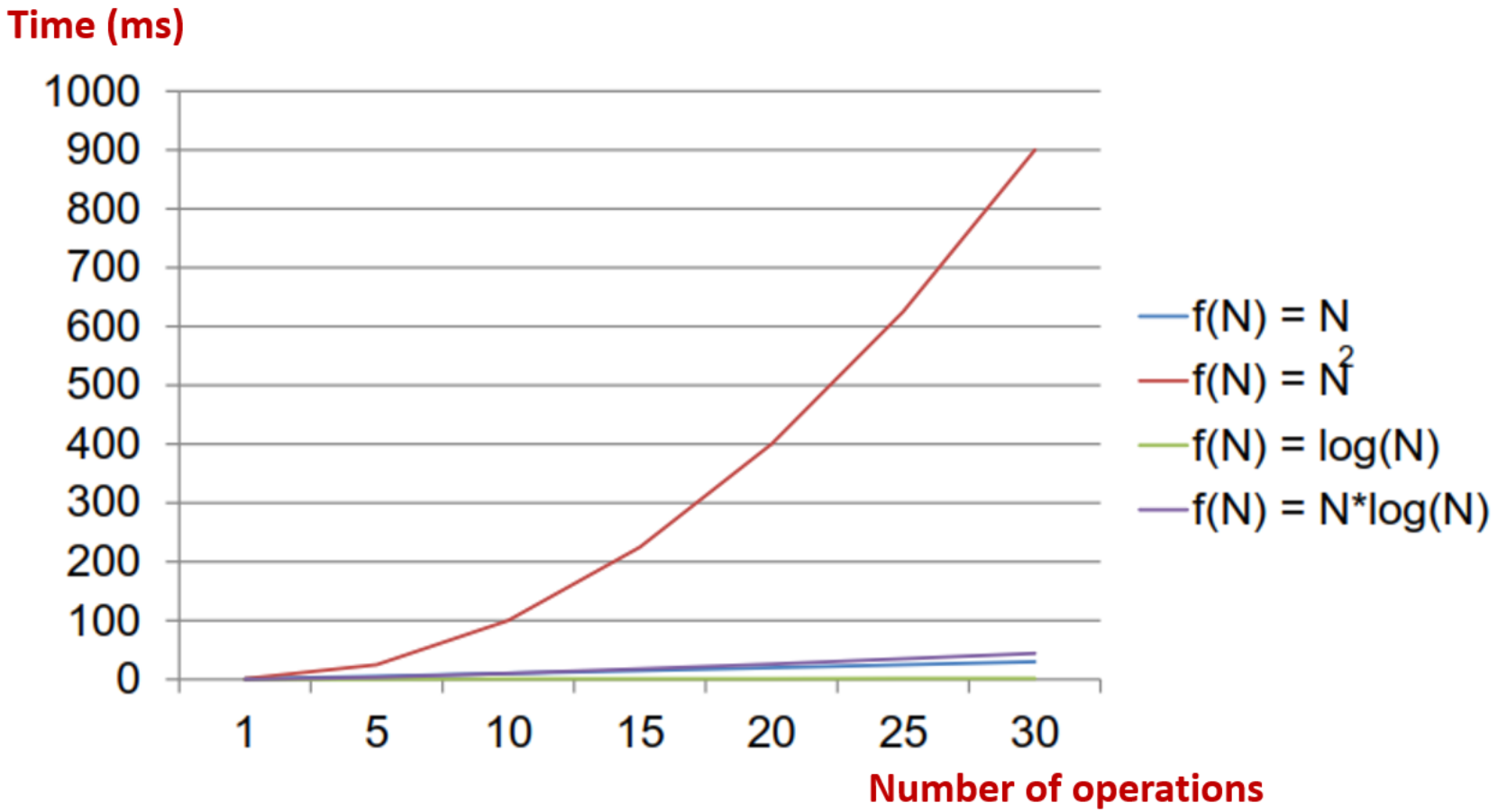
   $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(n!)$

   i.e., ignore low-order terms

Complexity of Algorithm

❑ Big-O Notation

- These time complexity can be treated as if it were a mathematical function



Algorithm	Time Complexity	Math function
Algorithm 1	O(N)	$f(N) = N$
Algorithm 2	$O(N^2)$	$f(N) = N^2$
Algorithm 3	O(log(N))	f(N) = log(N)
Algorithm 4	O(N*log(N))	f(N) = N*log(N)

Constant time complexity O(1)

constant time

O(1) "big oh of one"

```
x = 5 + (15 * 20);
```

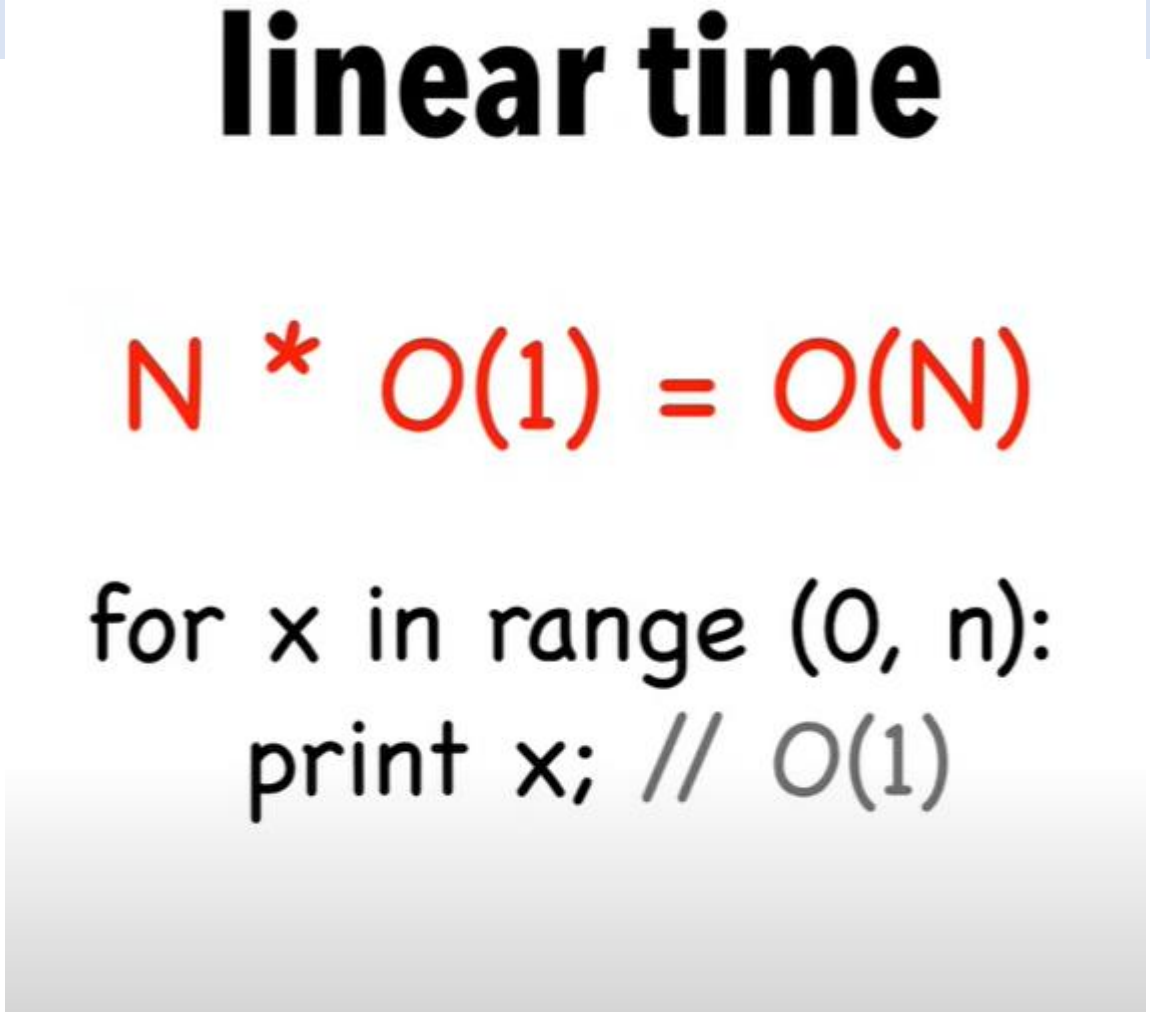
independent of input size, N

```
x = 5 + (15 * 20);
y = 15 - 2;
print x + y;
```

total time = O(1) + O(1) + O(1) = O(1)

3 * O(1)

Linear time complexity O(n)



Linear time complexity O(n)

```
y = 5 + (15 * 20);          O(1)
for x in range (0, n):  }   O(N)
    print x;
```

total time = O(1) + O(N) = O(N)

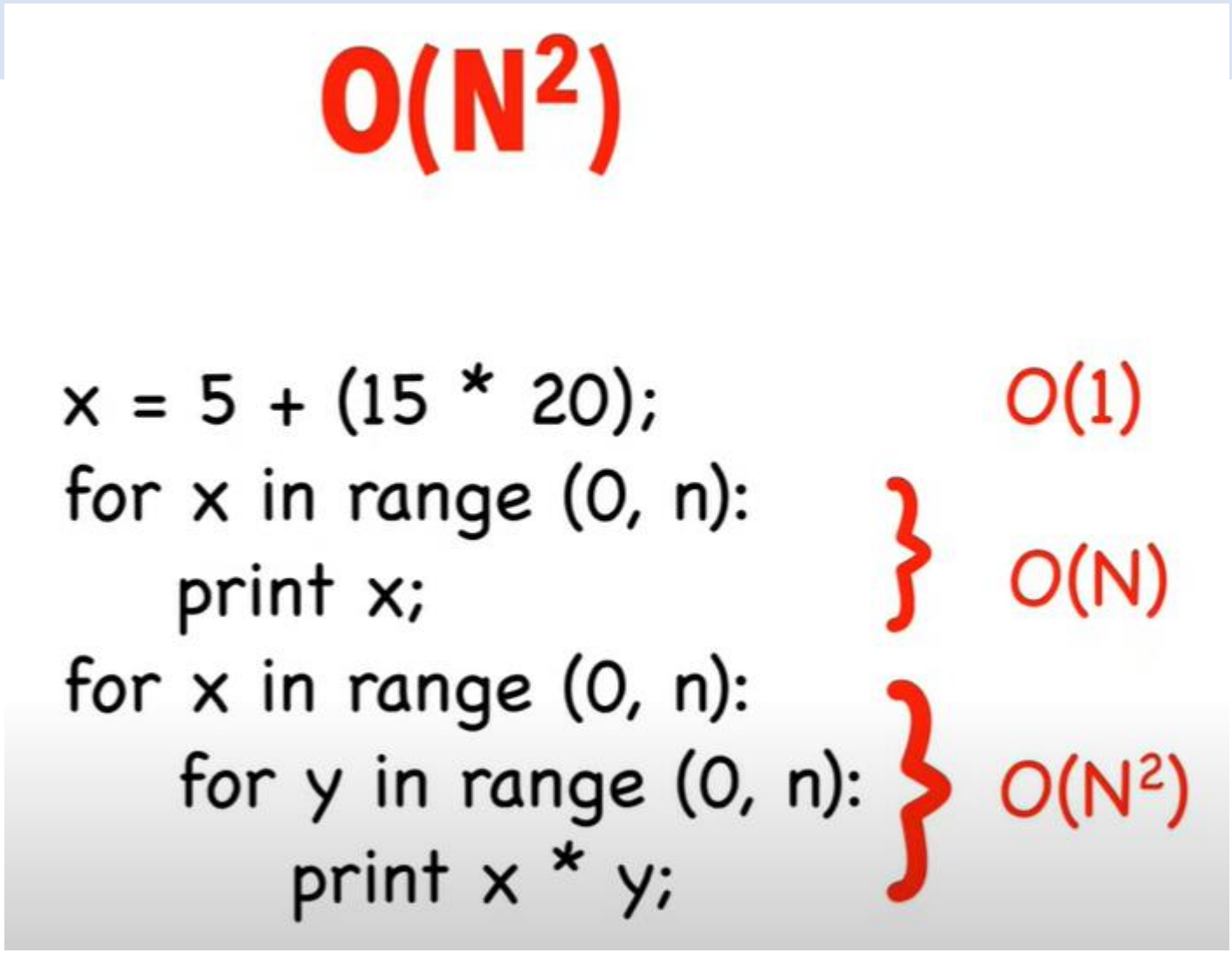
Quadratic time complexity $O(n^2)$

quadratic time

$O(N^2)$

```
for x in range (0, n):
    for y in range (0, n):
        print x * y; // O(1)
```

Quadratic time complexity $O(n^2)$



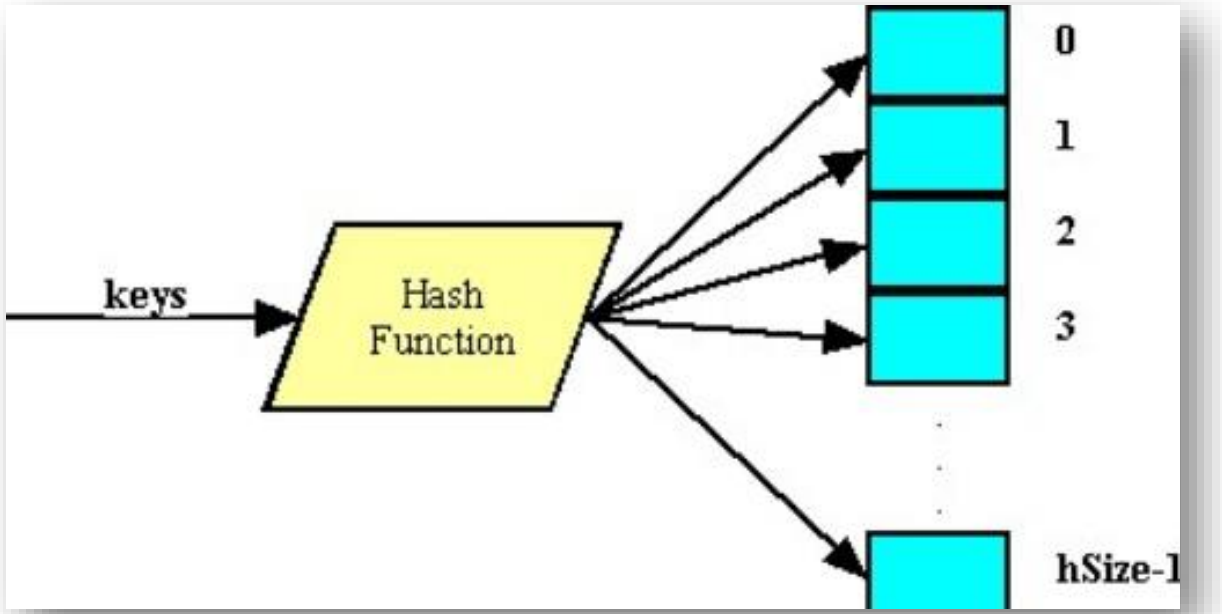
Terms

❑ Hash Table Vs. Hash Function Vs. Hashing

- Hash Table
  - Is a data structure which is used to store key-value pairs. It is implemented as array
- Hash Function
  - Is a function used by hash table **to compute an index of an array** in which an element will be inserted to or be searched at
- Hashing
  - In hashing, large keys are converted into smaller ones using hash function AND then the values are stored in hash table.

Hash Functions

- A hash function usually means a function that compresses.
  - The output is shorter than the input



An example of a hash function

- Definition

```
int  hash(int  key){
    return  key%7;
}
```

Size of array

Example: Implementation of hash table

```
#include<iostream>
using namespace std;

const int SIZE=7;
int ht[SIZE];

int hashFunction(int n){
        return   n%SIZE;
}
void insertData(int value){
        int index;
        index = hashFunction(value);
        ht[index] = value;
}
void displayHT(){
        for(int i=0; i<SIZE; i++){
                cout<<i<<"\t --> ";
                cout<<ht[i]<<"\n"'
        }
}
```

```
void initializeArray(){
    for(int i=0; i<SIZE; i++){
        ht[i] = -999
    }
}
```

1

```
main(){
        insertData(7);
        insertData(8);
        insertData(25);
        displayHT();
}
```

What is the output?

```
0        --> 7
1        --> 8
2        --> 0
3        --> 0
4        --> 25
5        --> 0
6        --> 0
```

2

```
main(){
        insertData(7);
        insertData(8);
        insertData(15);
        insertData(22);
        insertData(25);
        displayHT();
}
```

```
0        --> 7
1        --> 22
2        --> 0
3        --> 0
4        --> 25
5        --> 0
6        --> 0
```

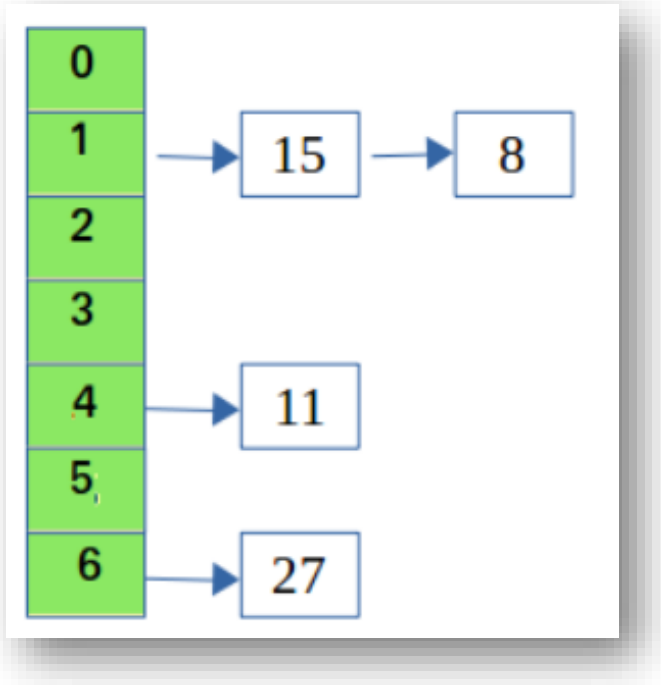
Ooop! Data is overridden?
This is known as collision!

Collision

❑ Definition

- A situation when two or more data hash to be stored in the same location in the table, is called a *hash collision*.

- Suppose we have a hash table with 7 elements
  - Hash function is to modulo with 7
  - How to insert these numbers?
    - 15, 11, 27, 8



Factors of a good hash function

- ✓ Easy to compute
- ✓ Minimize collision

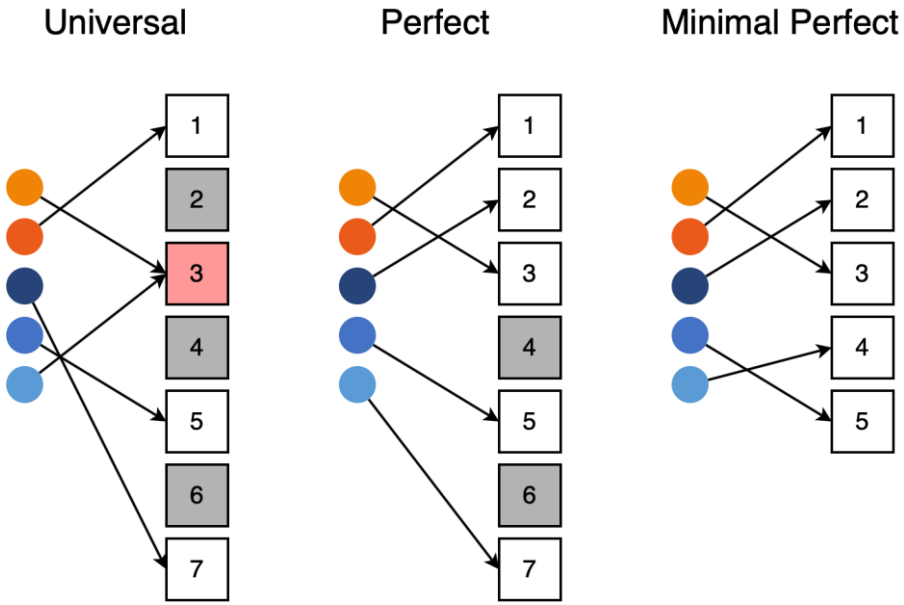


Perfect Hashing

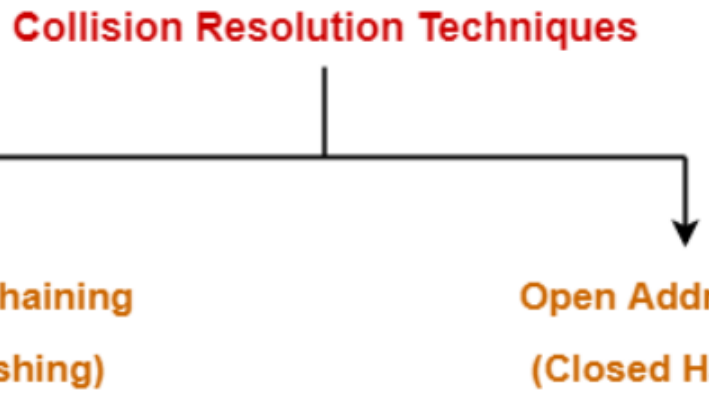
❑ Definition

- Perfect hashing maps each valid input to a different hash value

(no collision)

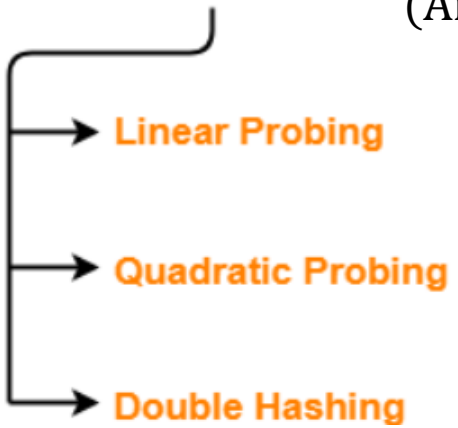


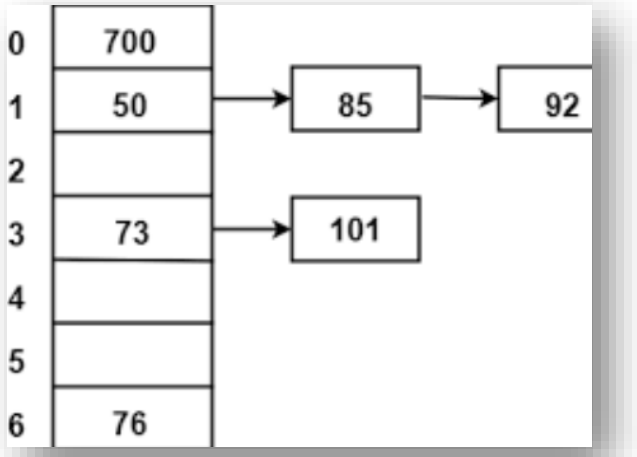
Collision Resolution Techniques



(An array of linked list implementation)

(Array-based implementation)





An example of using chaining

Solution for Hash Collision

❑ Open Hashing

- Open hashing defines **each slot in the hash table to be the head of a linked list**

- All records that hash to a particular slot (collision cases) are placed on that slot's **linked list**

Hash Table with Chaining resolution technique (An array of linked list)

1
```
struct Element{
   int value;
   Element *next;
};
struct List{
   int n;
   Element *head, *tail;
};
const int SIZE=7;
List  *ht[SIZE];
```

2
```
List* createAnEmptylist(){
      List *L1;
      L1 = new List();
      L1->n = 0;
      L1->head=NULL;
      L1->tail=NULL;
      return L1;
}
void createEmptyAllLists(){
     for(int i=0; i<SIZE; i++){
          ht[t] = createEmptyList()
    }
}
```

3
```
void addEnd(List *ls, int value){
   Element *e;
   e= new Element;
   e->next=NULL;
   e->value=value;

  if(ls->n==0){
        ls->head = e;
        ls->tail = e;
        ls->n = ls->n + 1;
  }else{
        ls->tail->next = e;
        ls->tail = e;
        ls->n = ls->n + 1;
   }
}
```

Display data in hash table

4
```
int hashFunction(int n){
        return n%SIZE;
}
```

5
```
void insertData(int value){
        int index;
        index = hashFunction(value);
        addEnd(ht[index], value);
}
```

6
```
void displayHT(){
   Element *e;
   for(int i=0; i<SIZE; i++){
      cout<<i<<"\t -->";
      if(ht[i]!=NULL){
         e = ht[i]->head;
         while(e!=0){
            cout<<e->value<<" ";
            e=e->next;
         }
      }
      cout<<endl;
   }
}
```

7
```
int main(){
   createEmptyAllLists();
   insertData(7);
   insertData(8);
   insertData(15);
   insertData(22);
   insertData(25);
   displayHT();
}
```

Output?

```
0        -->7
1        -->8 15 22
2        -->
3        -->
4        -->25
5        -->
6        -->
```

Solution for Hash Collision

❑ Closed Hashing

1. Linear probing
2. Quadratic probing
3. Double **hashing**

Case Study

Try these topics and explore how each method works!

Q & A

Hash Table

- Let’s store names of students into a hash table. This hash table has the size of 500.

How to compute the hash function?



ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]