哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

数据结构
Data Structures

Chapter 11 Efficient Searching

Prof. Yitian Shao
School of Computer Science and Technology

Efficient Searching

Course Overview

- Fast Search in ADTs
- Map
- C++ Unordered Map
- Hash Function
- Hashing Collision and Resolution
  - Probing
  - Separate Chaining
- Rehashing

Searching in an ADT: What we learned

- Unsorted Array (Vector)
- Linked List
- Stack
- Queue
- String
- Sorted Array (Vector)
- Binary Search Tree
- Graph

What are their searching efficiency?

Searching in an ADT: What we learned

- Unsorted Array (Vector)
- Linked List

$\}$ **O(N)**

- Stack
- Queue

$\}$ **Can only access the top/front/back**

- String **O(N)**
- Sorted Array (Vector)
- Binary Search Tree

$\}$ **O(logN)**

- Graph **O(E+N)** **BFS/DFS (E edges + N nodes)** **O(ElogN)** **Dijkstra's algorithm**

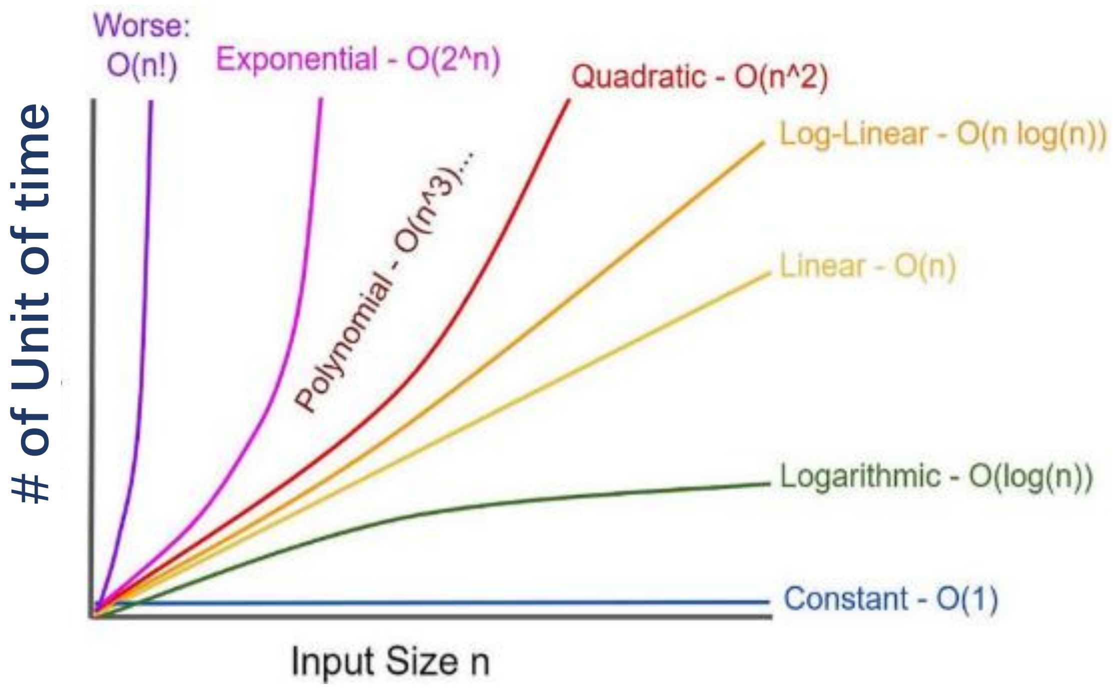
Can we search faster?

?

O(logN)

O(N)

O(NlogN)



Map

- Map a **key** to a **value**

Can find the corresponding value quickly

Example of Map: Encoding Characters

- Map a **key** to a **value**

Key (Code)	Value (Symbol)
65	A
66	B
⋮	⋮
90	Z

Code	Symbol	Code	Symbol	Code	Symbol	Code	Symbol
0	NUL (null)	32	(space)	64	@	96	`
1	SOH (start of header)	33	!	65	A	97	a
2	STX (start of text)	34	”	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	’	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	HT (horizontal tab)	41	)	73	I	105	i
10	LF (line feed/new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (form feed / new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (data control 1)	49	1	81	Q	113	q
18	DC2 (data control 2)	50	2	82	R	114	r
19	DC3 (data control 3)	51	3	83	S	115	s
20	DC4 (data control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of transmission block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL (delete)

Map

- Stores pairs of information
  - First half of the pair is called a **key**, and the second half is the associated **value**
  - Find a value by looking up its associated key
  - Keys must be **unique**
- Comparison with Vector
  - Vectors look up elements by index; **maps look them up by key**
  - Need to declare two types (for the key and the value)
  - Not ordered by index (Can be unordered or ordered by key)

Key (Code)	Value (Symbol)
65	A
66	B
⋮	⋮
90	Z

Example of Map: Encoding Characters

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

const char alphabetCodes[] = {'A','B','C','D','E','F','G','H','I','J',
                              'K','L','M','N','O','P','Q','R','S','T',
                              'U','V','W','X','Y','Z'};
char LookUp(int key){
    return alphabetCodes[key-65]; // User implementation of ASCII look up
                                  // table (of uppercase alphabets only)
}

int main(){
    for(int key = 65; key < 91; key += 2)
        cout << LookUp(key) << ' ';
}
```

```
A C E G I K M O Q S U W Y
```

Map of arbitrary size?

- We can use an array to realize a **map** with a given size (fixed)
- Each unique **key** can be converted to a unique array **index**, which then can be used to quickly find the corresponding **value** in O(1) time complexity
- Limitations
  - What if the number of key-value pairs is unknown?
  - What if map content changes during runtime?
  - What if the key is not an integer?

C++ Map (Unordered Map)

unordered_map functions	Time complexity	Usage
empty()	O(1)	checks whether the container is empty
insert()	O(1)	Inserts element into the container, if the container doesn't already contain an element with an equivalent key
erase()	O(1)	Removes specified elements from the container
find() **contains()** since c++20	O(1)	Finds an element with a specific key
size()	O(1)	returns the number of elements

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

unordered_map<key_type, value_type> name_of_the_map
```

Example of C++ Unordered Map: Insert()

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

int main()
{
    unordered_map<int, string> dict = {{1, "one"}, {5, "five"}};
    dict.insert({3, "three"});
    dict.insert(make_pair(4, "four"));
    dict.insert({{4, "another four"}, {2, "two"}});

    for (auto& p : dict) // Print out all key-value pairs
        cout << ' ' << p.first << " => " << p.second << '\n';
}
```

```
2 => two
4 => four
3 => three
5 => five
1 => one
```

Example of C++ Unordered Map: Erase()

```
six four two
```

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

int main(){
    unordered_map<int, string> c = {{1, "one"}, {2, "two"}, {3, "three"},
                                     {4, "four"}, {5, "five"}, {6, "six"}};
    c.erase(1);
    c.erase(3);
    c.erase(5);
    for (auto& p : c) // Print out all key-value pairs
        cout << p.second << ' ';
}
```

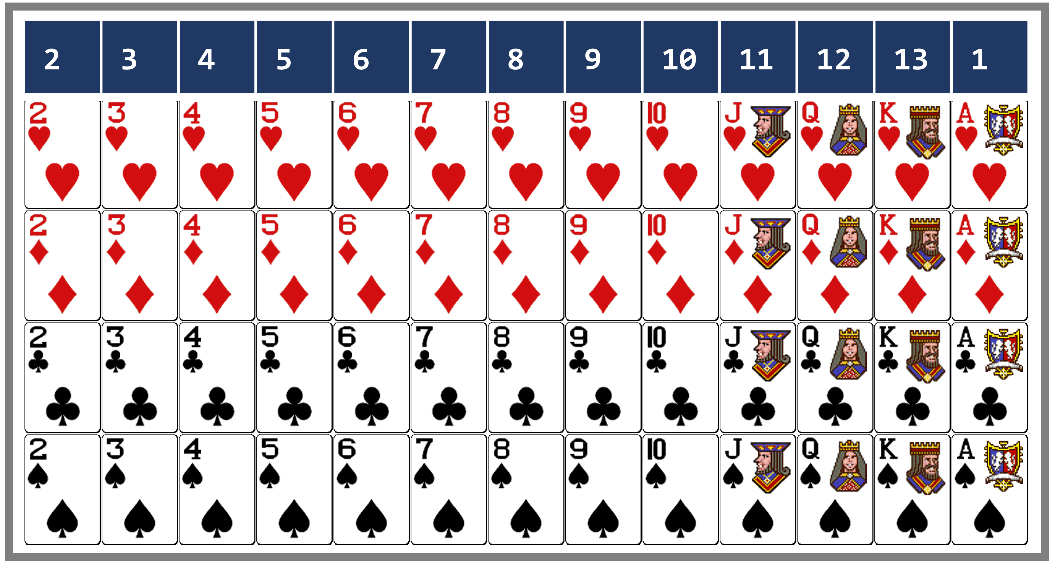
Example: Key Can Be Non-integer

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

int main(){
    unordered_map<string, int> phonebook = {{"Tyler", 5551234},
                                             {"Kate", 5559876}};

    for (auto& p : phonebook) // Print out all key-value pairs
        cout << p.first << " => " << p.second << endl;}
```

```
Kate => 5559876
Tyler => 5551234
```

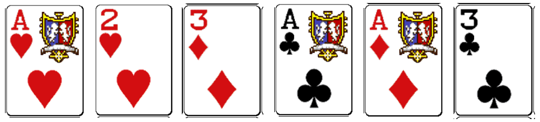
In-class Exercise: Paired Poker Cards

- Given an array of integers **`nums`**, return the total possible number of pairs, where a pair is defined as: **`nums[i] == nums[j]`** and **`i < j`**
- Only the number on the card is stored by **`nums`**



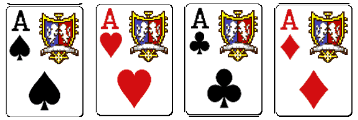
Example 1 contains 4 pairs

`nums = {1,2,3,1,1,3}`



Example 2 contains 6 pairs

`nums = {1,1,1,1}`



Solution: Paired Poker Cards

```
int CardPairs(vector<int>& nums) {
    unordered_map<int, int> pairs;

    int count = 0;

    for (int x: nums) {
        count += (pairs[x]++);
    }

    return count;
}
```

Solution: Paired Poker Cards

```
int CardPairs(vector<int>& nums) {
    vector<int> pairs(14); // This will also work, why?

    int count = 0;

    for (int x: nums) {
        count += (pairs[x]++);
    }

    return count;
}
```

Why O(1) Is Possible?

unordered_map functions	Time complexity
empty()	O(1)
insert()	O(1)
erase()	O(1)
find() contains() `since c++20`	O(1)
size()	O(1)

Hash Functions

- **Hash function**: function of the form

```
int hashFunc(Type arg)
```

- Must be deterministic (same input produces the same output)
- Should be well-distributed (the numbers produced are as spread out as possible)

- Idea: Store any given element value in the index given by the hash function (why hash functions must be consistent)

A Simple Hash Function for ASCII Alphabets

- A possible Hash function for finding ASCII alphabets using **indexing**

```
int hashFunc(int key)
    return key+65;
```

```
alphabetCodes[hashFunc(key)]
```

```
const char alphabetCodes[] =
{'A','B','C','D','E','F','G','H','I','J',
'K','L','M','N','O','P','Q','R','S','T',
'U','V','W','X','Y','Z'};
```

Key (Code)	Value (Symbol)
65	A
66	B
⋮	⋮
90	Z

Address (index)	Value (Symbol)
0	A
1	B
⋮	⋮
25	Z

Key → **Hash Function** → **Address to find the value**

Drawbacks of Linear Indexing Method

- Linearly map a **key** to **index** to access array values in O(1) time

  `HashFunc(key) → i → array[i] → value`

- The size changes dynamically in runtime?
- Too many values to be stored?
- Values stored in non-contiguous addresses?
- Drawbacks of array indexing
  - Array could be very sparse, mostly empty (memory waste)
  - Potentially requires an excessively large array (memory limitation)

Improving Space Efficiency

- If any number is equally possible, we'll need a huge array, even if we only have a couple of **buckets**
- Idea: use a hash function, but modify the result to be within a much smaller range (the size of the array)
- We can then think of the array as a sequence of **buckets** storing elements

```
HashFunc(key) → i → bucket_i → array[bucket_i] → value
```

Hash Code   Compression Map

Collision

- **Collision**: When a hash function maps ≥ 2 values to same index
- Collision example: we design hash function to improve space effciency

```
int CompressionMap(int key)
    return key % 10; // mod
```

- Add and store the following numbers: 11, 49, **24**, 37, **54**

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	54	0	0	37	0	49
size	5	*capacity*			10					

54 collides with 24!

- **Collision resolution**: An algorithm for fixing collisions
- A hash function should be **well-distributed** to minimize collisions

Collision Resolution: Probing

- **Probing**: Resolving a collision by moving to another index
  - **Linear probing**: Moves to the **next available index** (wraps if needed)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	54	0	37	0	49
size	5	*capacity*			10					

  - **Quadratic probing**: a variation that moves increasingly far away: index +1, +4, +9, …
- Drawbacks of probing? How does this change add, contains, etc.?

Clustering Problem

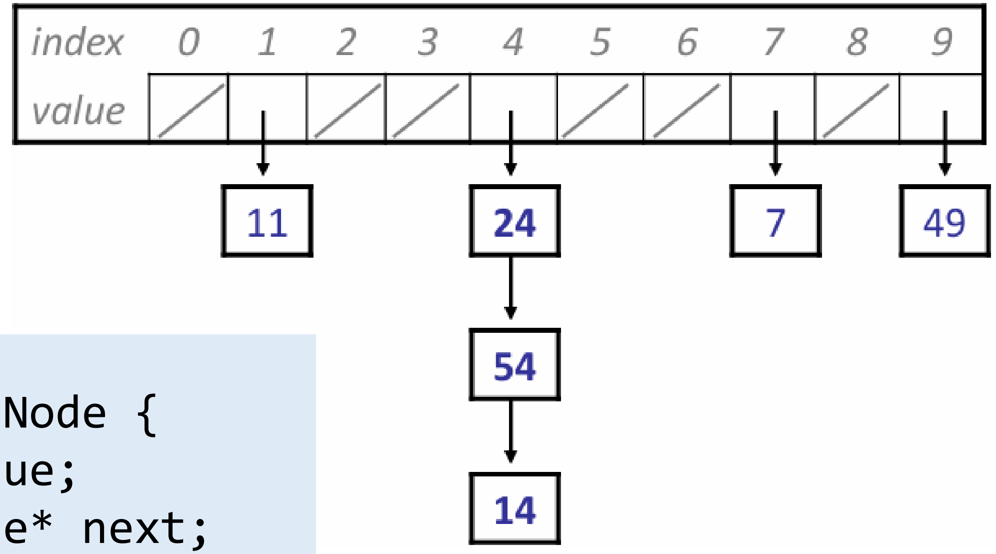
- **Clustering**: Clumps of elements at neighboring indexes
  - Slows down the hash table lookup; you must loop through them.
  - Add and store the following numbers: 11, 49, **24**, 37, **54, 14, 86**

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	54	14	37	86	49
size	5	capacity		10						

  - A lookup for 94 must look at 7 out of 10 total indexes
  - Must have a special value for removed elements (tombstones)

Separate Chaining

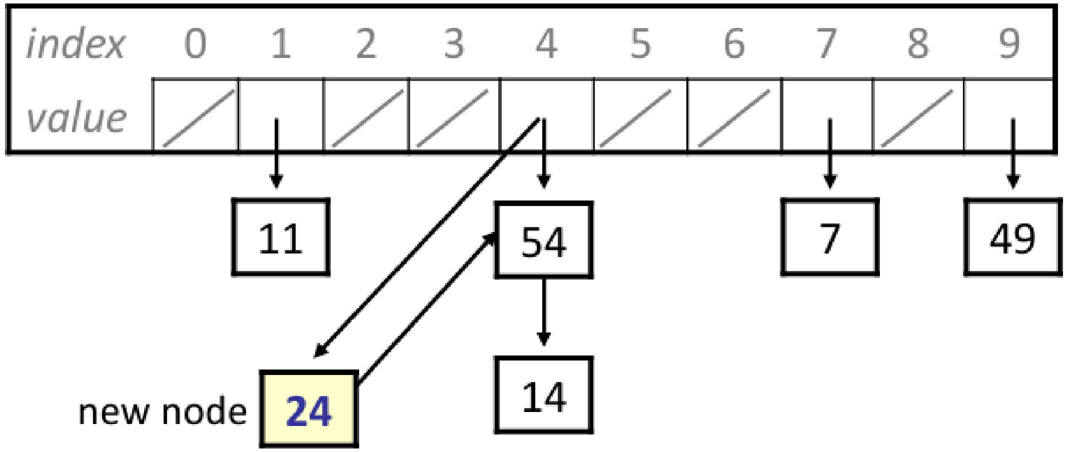
- **Separate chaining**: Solving collisions by storing a **list** at each index
- Add/search/remove must traverse lists, but the **lists are short**
- Impossible to "run out" of indexes, unlike with probing




```
struct HashNode {
    int value;
    HashNode* next;
};
```

The insert() Operation

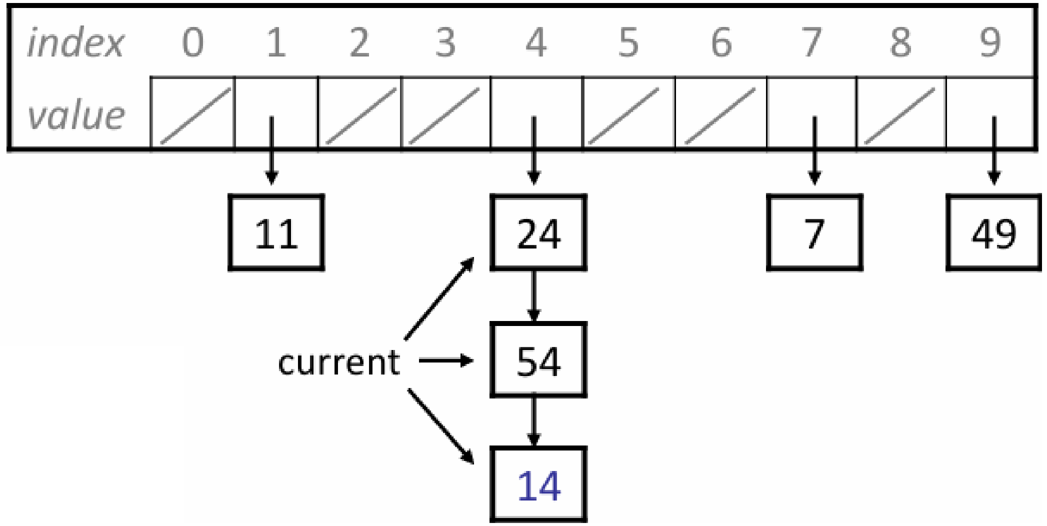
- How do we insert an element to the hash table?
  - Recall: To modify a linked list via changing the list's **front reference**
  - Where in the list should we add the new element?
  - Must make sure to avoid duplicates




```
void addToFront(int elem, HashNode* &front){
  HashNode* newNode = new HashNode(elem, front);
  front = newNode;
}
```

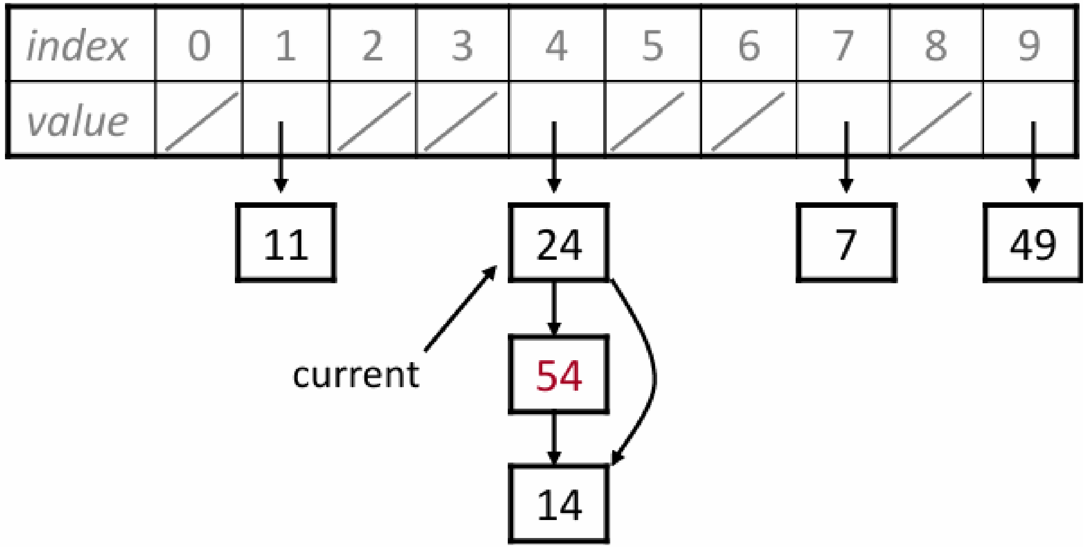
The contains() Operation

- How do we search for an element in the hash table?
  - Must loop through the linked list for the appropriate hash index, looking for the desired value
  - Recall: Traverse a linked list with a "current" node pointer



The erase() Operation

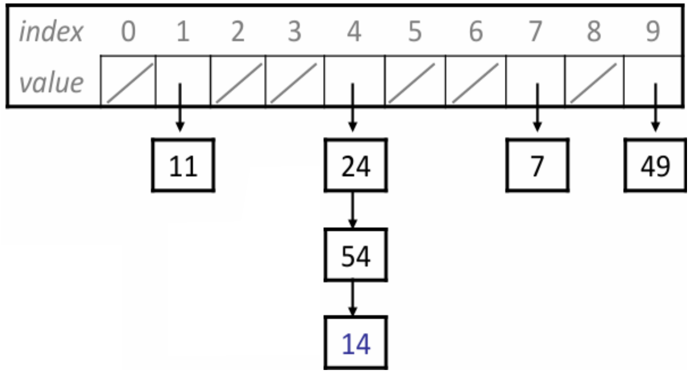
- How do we remove an element from the hash table?
  - Cases: front (24), non-front (14), not found in list (94), null (32)
  - To remove a node from a linked list, you must either change the list's front, or the next field of the previous node in the list



Rehashing

- **Rehash**: Growing to a larger array when the table is too full
- Cannot simply copy the old array to a new one

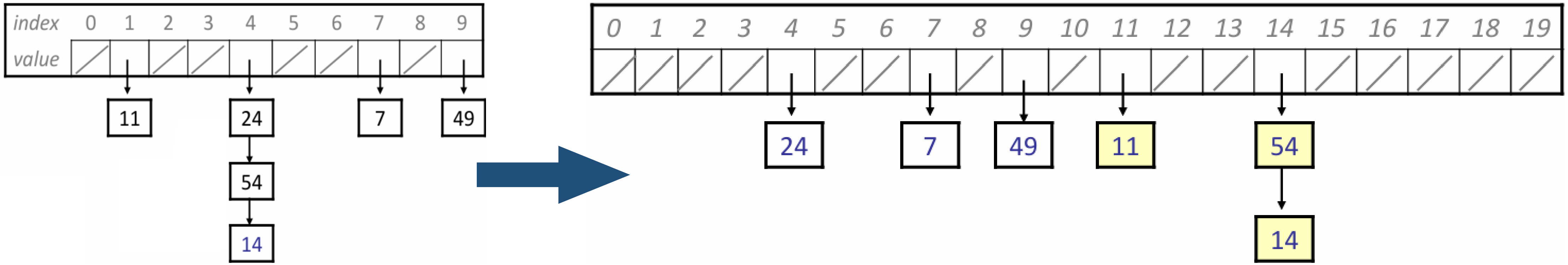
  (Why not?)



- Rehashing iterate through all elements and calculate their **new bucket positions** using the **new hash function** that corresponds to the new size of the hashmap
- This process can be **time-consuming** but it is necessary to maintain the efficiency of the hashmap

Rehashing

- **Rehash**: Growing to a larger array when the table is too full



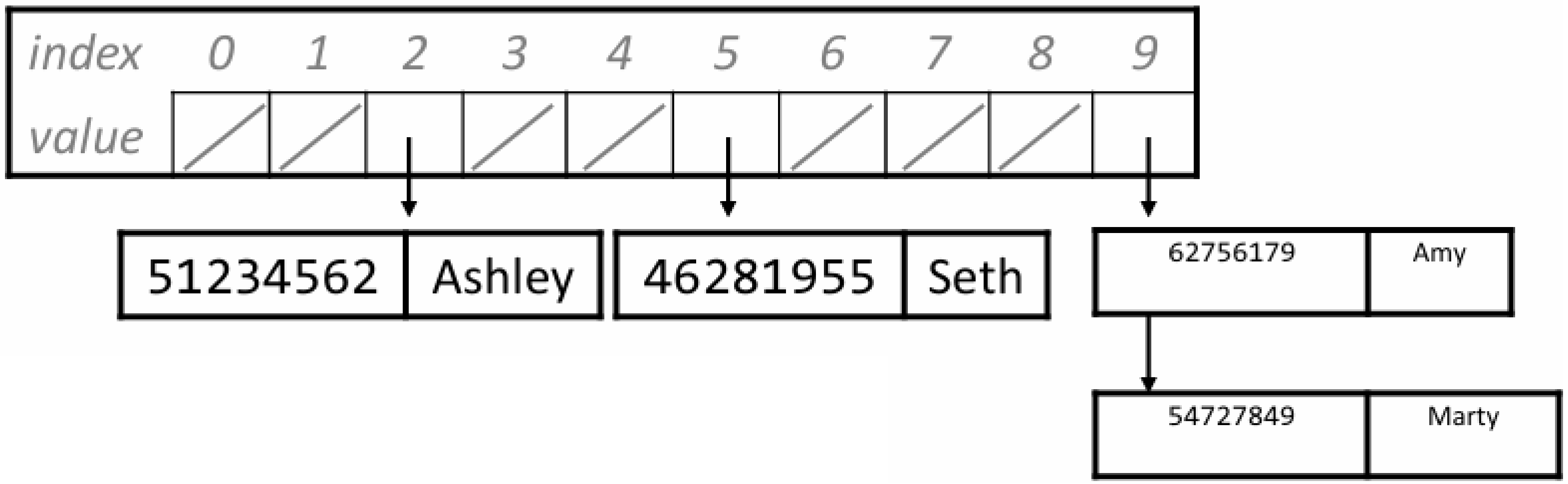
- **Load factor** = (number of elements) / (hash table length)
  - Many implementations rehash when load factor ≈ 0.75
  - This load factor needs to be kept low so that number of entries at one index is less and so is the complexity almost constant, i.e., O(1)

Rehashing Procedures

- For each addition of a new entry to the map, **check the load factor**
- If it's greater than its pre-defined value (0.75 by default), then **rehash**
- For **rehash**, make a new array of double the previous size and make it the new bucket array
- Then traverse to each element in the old bucket array and call the insert() for each to insert it into the new larger bucket array

Hash Map

- Hash map nodes store key-value pairs



- Note that the hashing is always done on the keys, not the values.

Extra Read: Hash Set

- Set Interface does not allow duplicate (key) values
- Hash set can be considered as Hash map but with nodes storing the keys only (instead of key-value pairs)

```
unordered_set<int> example_set{11, 2, 35, 4};
```

unordered_set functions	Time complexity	Usage
empty()	O(1)	checks whether the container is empty
insert()	O(1)	Inserts element into the container, if the container doesn't already contain an element with an equivalent key
erase()	O(1)	Removes specified elements from the container
find() **contains()** `since c++20`	O(1)	Finds an element with a specific key
size()	O(1)	returns the number of elements

Extra Read: Designing A Hash Function

- Need to choose a good **hash function**:
  - Quick to compute
  - Distributed keys uniformly throughout the bucket array, minimize the probability of collision
- Good hash functions are very rare
  - Birthday paradox: In a room of just 23 people there's a 50% chance of at least two people having the same birthday; in a room of 75 people, the chance increases to 99%!

Extra Read: Popular Hash Functions

- **Memory address**: Interpret the memory address of the key object as an integer
- **Integer cast**: Interpret the bits of the key as an integer
- **Component sum**: Partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and sum the components (ignoring overflows)
- **Polynomial accumulation**: Partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits) $a_0 a_1 \dots a_{n-1}$ and evaluate the polynomial $p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}$ at a fixed value z (ignoring overflows)

Summary: Searching in an ADT

- Unsorted Array (Vector)
- Linked List

O(N)

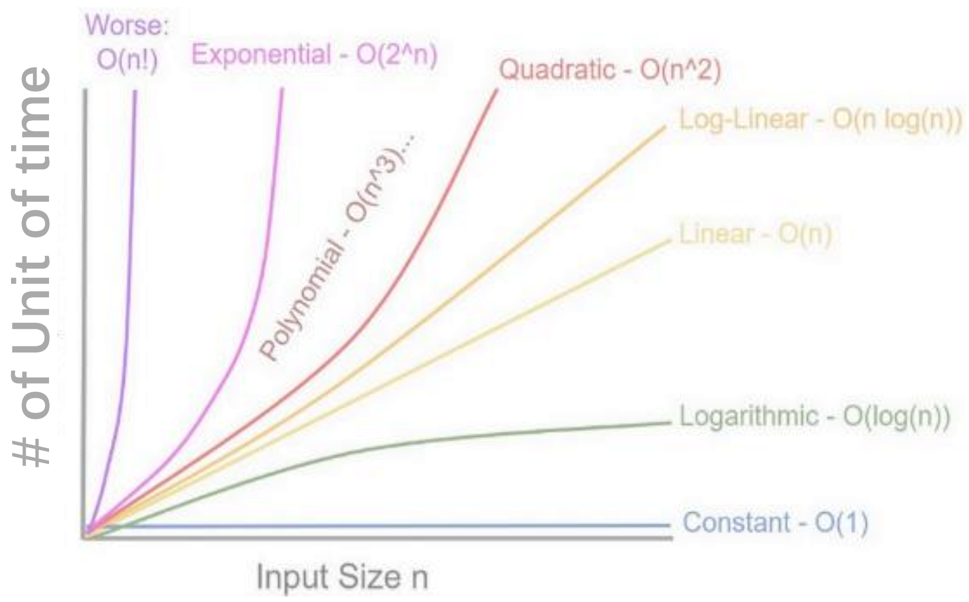
- Stack
- Queue

Can only access the top/front/back: O(1)

- String **O(N)**
- Sorted Array (Vector)
- Binary Search Tree

O(logN)

- Graph **O(E+N)** **BFS/DFS (E edges + N nodes)** **O(ElogN)** **Dijkstra's algorithm**
- Hash Map **O(1)**



Exercise 11.1

- Complete [LeetCode 1512]

1512. Number of Good Pairs

Easy | Topics | Companies | Hint

Given an array of integers `nums`, return *the number of* ***good pairs***.

A pair `(i, j)` is called *good* if `nums[i] == nums[j]` and `i` < `j`.

Example 1:

```
Input: nums = [1,2,3,1,1,3]
Output: 4
Explanation: There are 4 good pairs (0,3), (0,4), (3,4), (2,5) 0-indexed.
```

Example 2:

```
Input: nums = [1,1,1,1]
Output: 6
Explanation: Each pair in the array are good.
```

Exercise 11.2

- Complete [LeetCode 1832]

1832. Check if the Sentence Is Pangram

Easy | Topics | Companies | Hint

A **pangram** is a sentence where every letter of the English alphabet appears at least once.

Given a string `sentence` containing only lowercase English letters, return `true` *if* `sentence` *is a **pangram**, or* `false` *otherwise.*

Example 1:

```
Input: sentence = "thequickbrownfoxjumpsoverthelazydog"
Output: true
Explanation: sentence contains at least one of every letter of the English alphabet.
```

Example 2:

```
Input: sentence = "leetcode"
Output: false
```

Exercise 11.3

- Complete LeetCode 2965

2965. Find Missing and Repeated Values

Easy Topics Companies

You are given a **0-indexed** 2D integer matrix `grid` of size `n * n` with values in the range `[1, n²]`. Each integer appears **exactly once** except `a` which appears **twice** and `b` which is **missing**. The task is to find the repeating and missing numbers `a` and `b`.

Return *a **0-indexed** integer array* `ans` *of size* `2` *where* `ans[0]` *equals to* `a` *and* `ans[1]` *equals to* `b`.

Example 1:

```
Input: grid = [[1,3],[2,2]]
Output: [2,4]
Explanation: Number 2 is repeated and number 4 is missing so the answer is [2,4].
```

Exercise 11.4

- Complete [LeetCode 2744]

2744. Find Maximum Number of String Pairs

Easy | Topics | Companies | Hint

You are given a **0-indexed** array `words` consisting of **distinct** strings.

The string `words[i]` can be paired with the string `words[j]` if:

- The string `words[i]` is equal to the reversed string of `words[j]`.
- `0 <= i < j < words.length`.

Return *the **maximum** number of pairs that can be formed from the array* `words`.

Note that each string can belong in **at most one** pair.

Example 1:

```
Input: words = ["cd","ac","dc","ca","zz"]
Output: 2
Explanation: In this example, we can form 2 pair of strings in the following way:
- We pair the 0th string with the 2nd string, as the reversed string of word[0] is "dc" and is equal to words[2].
- We pair the 1st string with the 3rd string, as the reversed string of word[1] is "ca" and is equal to words[3].
It can be proven that 2 is the maximum number of pairs that can be formed.
```

Exercise 11.5

- Complete [LeetCode 1]

1. Two Sum

Easy Topics Companies Hint

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

You may assume that each input would have ***exactly* one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Exercise 11.6

- Complete [LeetCode 690]

690. Employee Importance

Medium | Topics | Companies

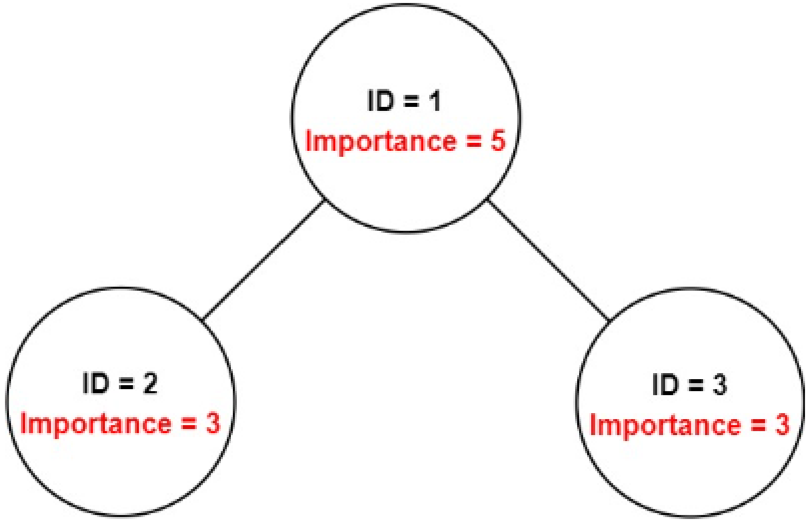
You have a data structure of employee information, including the employee's unique ID, importance value, and direct subordinates' IDs.

You are given an array of employees `employees` where:

- `employees[i].id` is the ID of the i^{th} employee.
- `employees[i].importance` is the importance value of the i^{th} employee.
- `employees[i].subordinates` is a list of the IDs of the direct subordinates of the i^{th} employee.

Given an integer `id` that represents an employee's ID, return *the **total** importance value of this employee and all their direct and indirect subordinates.*

Example 1:




```
Input: employees = [[1,5,[2,3]],[2,3,[]],[3,3,[]]], id = 1
Output: 11
Explanation: Employee 1 has an importance value of 5 and has two
direct subordinates: employee 2 and employee 3.
They both have an importance value of 3.
Thus, the total importance value of employee 1 is 5 + 3 + 3 = 11.
```

Hint: Create a map of employee info to allow fast checking, then perform DFS (or BFS)