Udemy - Notes

Complexity { time, space } ⇒ Big-O Notation → worst case

e.g. unorder list → sort (last element)

n = input size

- Constant time = $O(1)$
- Logarithmic time = $O(\log(n))$
- Linear time = $O(n)$
- Linearithmic time = $O(n \log n)$
- Quadric time = $O(n^2)$
- Cubic time = $O(n^3)$
- Exponential time = $O(b^n)$, $b>1$
- Factorial time = $O(n!)$

Big-O Properties

$O(c+n)$
$= O(cn) = O(n)$

e.g. $f(n) = 7\log n^3 + 15n^2 + 2n^3 + 8$
$= O(f(n)) = O(n^3)$

Big-O Examples

e.g.1 : $O(1)$ →
```
a = 1;
b = 2;
c = a + 5b;
```
→
```
i = 0;
while (i < 11) {
    i++;
}
```
X rely input

e.g.2 : $O(n)$ →
```
i = 0;
while (i < n) {
    i++;
}
```
$f(n) = n$

→
```
i = 0;
while (i < n) {
    i += 3;
}
```
$f(n) = n/3$

e.g.3 $O(n^2)$ → `for (int i = 0; i < n; i++)...`

$f(n) = (n)+(n-1)+(n-2)+ \dots +3+2+1 = n(n+1)/2$
$= (n^2+n)/2$
$\therefore O(f(n)) = O(n^2)$

e.g.4 $O(\log n)$ → Find in sorted array (Binary Search)
```
low = 0;
high = n - 1;
while (low <= high) {
    mid = (low + high) / 2;
}
if (array[mid] == value) { return mid; }
else if (array[mid] < value) { low = mid + 1; }
else if (array[mid] > value) { high = mid - 1; }
else { return null; }
```
$n(0.5)^k = 1$ k=times
$n = 2^k$
$\log_2 n = k$

e.g.5 $O(n^2)$ →
```
i = 0;
while (i < n) {
    j = 0;
    while (j < 3*n) {
        j += 1;
    }
    while (j < 2*n) {
        j += 1;
    }
    i++;
}
```
n, 3n, 2n

$f(n) = n(3n+2n) = 5n^2$
$\therefore O(f(n)) = O(n^2)$

classic examples:
- Finding all subsets of a set = $O(2^n)$
- Finding all permutations of string = $O(n!)$
- Sorting by mergesort = $O(n \log n)$
- Iterate in a matrix of size n by m = $O(nm)$

Arrays

A = [44, 12, -3, 7 ... 100] / [0, 1, 2, 3 ... 8]

Structure ← Static Array

- what? → a fixed length container containing n elements indexable from [0, n-1].
  - each index reference with no.
- where? → everywhere!
  - dynamic
  - sort
  - buffers
  - Look up
  - Temp. storage
- Complexity?
  - Access $O(1)$
  - Search $O(n)$
  - Insert n/a — $O(n)$ Dyn.
  - Append n/a — $O(1)$
  - Delete n/a — $O(n)$

Dynamic Array → grow & shrink as needed.
→ double everytime when not enough. (to keep $O(1)$)

Singly & Doubly Linked Lists

Terminology:
- Head
- Tail
- Pointer
- Node

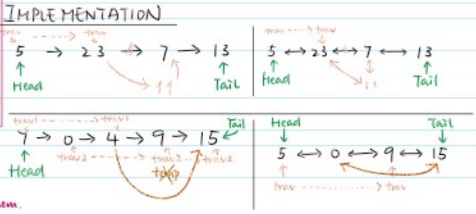
Linked List
- what? → sequential list of nodes
- where
  - list, queue, stack
  - circular list
  - real life example e.g. train
  - graph
  - seperate chain (HashTable)

Singly Linked List	vs	Doubly Linked List
- Hold reference to the next node from Head to the Tail.		- Hold reference both to the next & previous node from head to tail
✓ use less memory		✓ Traversed backwards
✓ simple implementation.		
X Cannot access previous elements easily		X Takes double memory

- PrintList
- Push_front
- Push_back
- Delete
- Clear
- Reverse

IMPLEMENTATION

① 1 ptr. = trav
② trav ⇒ [1]
③ new ele .next = 7
④ trav.next = 23



① Create 2 pointers
② Create 1 pointer
③ Clear up mem.

① Create 1 trav pointer
② Track prev. next
③ Remove.

Complexity

	singly linked	doubly linked
Search	$O(n)$	$O(n)$
Insert at Head	$O(1)$	$O(1)$
Insert at tail	$O(1)$	$O(1)$
Remove at Head	$O(1)$	$O(1)$
Remove at tail	$O(n)$	$O(1)$
Remove in middle	$O(n)$	$O(n)$

Stacks

- a one-ended linear data structure, having 2 primary operations: push, pop

Stacks
- WHAT?
- CHARACTERISTIC
  - LIFO
  - Last-in-First-Out
  - top pointer → push + pop here
- WHEN & WHERE?
  - text-editors
  - compiler (check for matching brackets, braces)
  - Real-world stack (books and plates)
  - Game (tower of H.)
  - use to support recursion
  - Depth First Search

小的必须在大的之上

Bracket Sequence e.g. [[{}]()]

→ current + reversed = removed
-(top == reversed)

```
if( stack != empty){
        return false;
}else {
        return true;
}
```

Complexity

Pushing	$O(1)$
Popping	$O(1)$
Peeking	$O(1)$
Searching	

Implementation with singly linked list

Instruction
```
PUSH(4)
PUSH(2)
POP()
POP()
```

1. Move Head
2. Null.

Pushing	O(1)
Popping	O(1)
Peeking	O(1)
Searching	O(n)
Size	O(1)

小的必须在大的之上

Queue

- linear data structure, with the 2 primary operations, enqueue & dequeue

Queue

WHAT?

WHERE & WHEN?

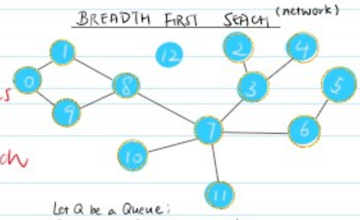
- a waiting line model
- keep track of the most recently added elements
- web server (FCFS)
- Breadth first Search graph traversal.

CHARACTERISTIC

- Remove queue front = dequeue
- Add from queue back = enqueue

Complexity	
enqueue	O(1)
dequeue	O(1)
peeking	O(1)
contains	O(n)
removal	O(n)
isEmpty	O(1)

BREADTH FIRST SEACH (network)



```
Let Q be a Queue;
Q.enqueue (starting_node);
starting_node.visited = true;

while (Q.isEmpty != true){
    node = Q.dequeue;

    for (neighbour in neighbours node)
        if (neighbour is not visit){
            neighbour.visited = true;
            Q.enquen (neighbour);
        }
}
```

Priority Queue & Heap

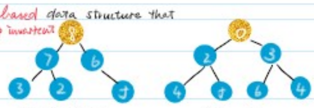
Priority Queue

WHAT?

- Similar to normal queue but each element has a certain priority (order)
- Only store comparable data
- Poll = remove

HEAP!

- a tree-based data structure that satisfy heap invariant



Max Heap. Min. Heap.

Complexity

Binary Heap construction	O(n)
Polling	O(logn)
Peeking	O(1)
Adding	O(logn)
Naive Removing	O(n)
Advanced Remove (hash)	O(logn)
Naive contain	O(n)
Contain check (hash)	O(1)

WHEN & WHERE?

- certain implementation of Dijkstra's Shortest Path algorithm
- Fetch "next best" & "next worst"
- Huffman coding (lossless coding compression)
- BFS (PQ → grap next most promising node.
- Minimum Spanning Tree

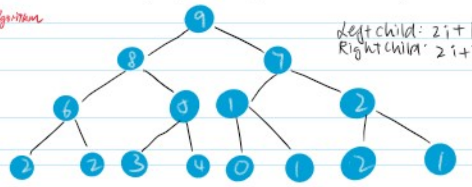
Turn min. PQ to max. PQ

- negate. → renegate

Let lex( ) as comparator → lex(s_1, s_2) = −1, $s_1 < s_2$
= 0, $s_1 = s_2$
= 1, $s_1 > s_2$
→ nlex(s_1, s_2) = 1, $s_1 < s_2$
= 0, $s_1 = s_2$
= −1, $s_1 > s_2$

ADDING ELEMENTS (BINARY HEAP)

- Heap * gives best possible time complexity



Left child: 2i+1
Right child: 2i+2

Binary Tree & BST

- a tree that at most 2 children
- BST = binary tree that satisfy BST invariant
  → left < right !

绕过 root → 左

Complexity

	average	worst
Insert	O(logn)	O(n)
Delete	O(logn)	O(n)
Remove	O(logn)	O(n)
Search	O(logn)	O(n)

INSERTION

* ORDER & COMPARABLE *
- Recurse down left (< case)
- Recurse down right (> case)
- Finding duplicate (= case)
- Create new node (null)

REMOVE

- find the element
- replace → 4 cases
  1. leaf node
  2. left sub
  3. right sub
  4. Both — choose successor either smallest in right subtree or largest in left

Traversal

- Preorder before recursive calls
- Inorder between recursive calls
- Postorder after recursive calls