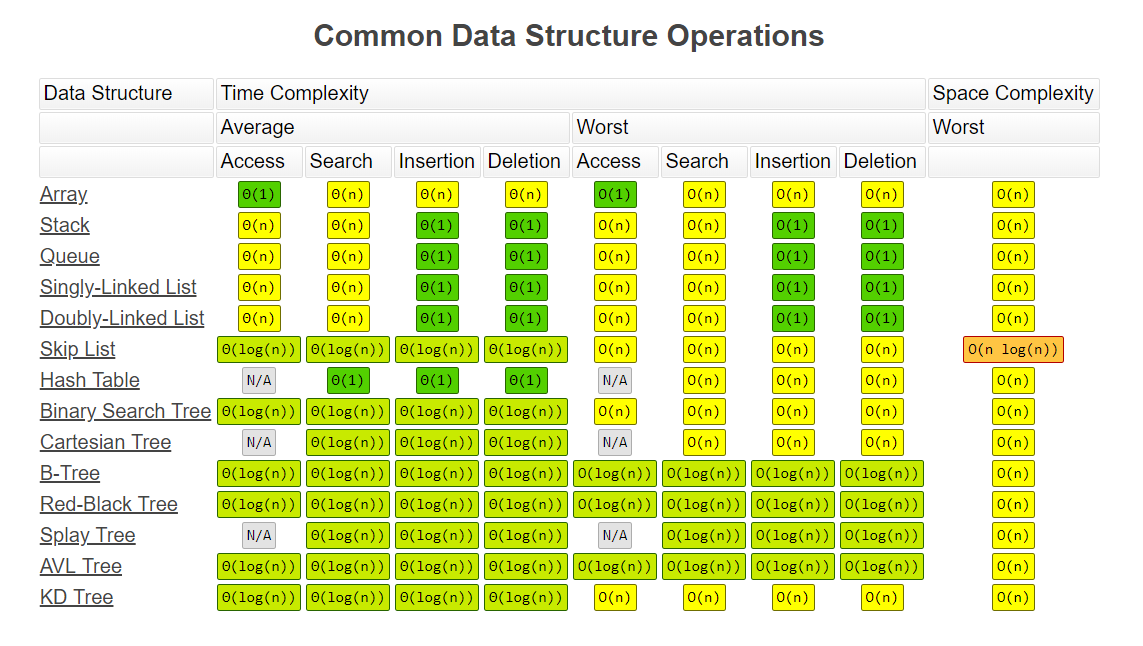
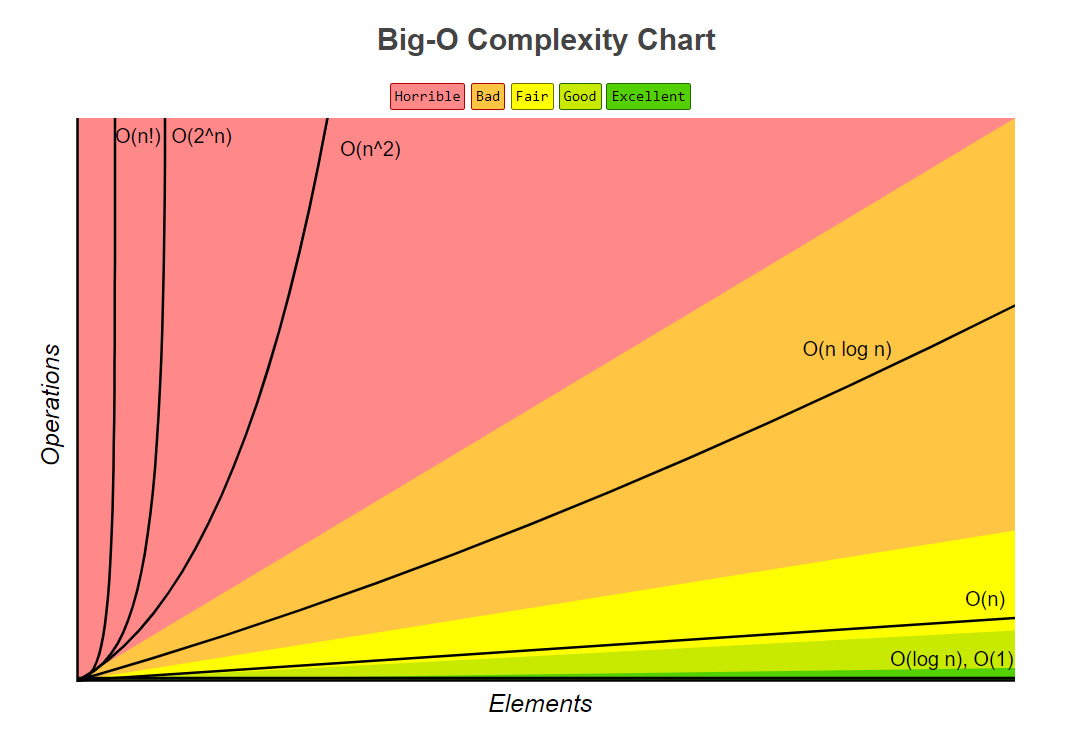
# 1、时间复杂度、空间复杂度

## <https://www.bigocheatsheet.com/>



# 1618631021(1)2、数据结构和算法

## 数据结构

• 一维：

• 基础：数组 array (string), 链表 linked list

• 高级：栈 stack, 队列 queue, 双端队列 deque, 集合 set, 映射 map (hash or map), etc

• 二维：

• 基础：树 tree, 图 graph

• 高级：二叉搜索树 binary search tree (red-black tree, AVL), 堆 heap, 并查集 disjoint set, 字典树 Trie, etc

• 特殊：

• 位运算 Bitwise, 布隆过滤器 BloomFilter

• LRU Cache

## 算法

• If-else, switch —> branch

• for, while loop —> Iteration

• 递归 Recursion (Divide & Conquer, Backtrace)

• 搜索 Search: 深度优先搜索 Depth first search, 广度优先搜索 Breadth first search, A\*, etc

• 动态规划 Dynamic Programming

• 二分查找 Binary Search

• 贪心 Greedy

• 数学 Math , 几何 Geometry

# 3.数组、链表、跳表

## 链表

### 时间复杂度

prepend O(1)

Append O(1)

Lookup O(n)

Insert O(1)

Delete O(1)

### 空间复杂度

O(n)

## 跳表

注意：只能用于元素有序的情况。

所以，跳表（skip list）对标的是平衡树（AVL Tree）和二分查找，

是一种 插入/删除/搜索 都是 O(log n) 的数据结构。1989 年出现。

它最大的优势是原理简单、容易实现、方便扩展、效率更高。因此

在一些热门的项目里用来替代平衡树，如 Redis、LevelDB 等

给有序的链表加速:升维——增加多级索引

### 时间复杂度

索引的高度：logn

在跳表中查询任意数据的时间复杂度就是 O(logn)

### 空间复杂度

O(n)

# 4.栈、队列、双端队列、优先队列

## Stack

先入后出；添加、删除皆为 O(1)

## Queue

先入先出；添加、删除皆为 O(1)

## Deque

1. 简单理解：两端可以进出的 Queue

Deque - double ended queue

1. 插入和删除都是 O(1) 操作

## Priority Queue

1.插入操作：O(1)

2.取出操作：O(logN) - 按照元素的优先级取出

3.底层具体实现的数据结构较为多样和复杂：heap、bst、treap

# 哈希表、映射、集合

## Hash table

哈希表（Hash table），也叫散列表，是根据关键码值（Key value

）而直接进行访问的数据结构。

它通过把关键码值映射到表中一个位置来访问记录，以加快查找的

速度。

这个映射函数叫作散列函数（Hash Function），存放记录的数组叫

作哈希表（或散列表）。

## Map

key-value对，key不重复

## Set

不重复元素的集合

# 6.树、二叉树、二叉搜索树

Linked List 是特殊化的 Tree

Tree 是特殊化的 Graph

## 二叉树遍历 Pre-order/In-order/Post-order

1.前序（Pre-order）：根-左-右

2.中序（In-order）：左-根-右

3.后序（Post-order）：左-右-根

示例代码

def preorder(self, root):

if root:

self.traverse\_path.append(root.val)

self.preorder(root.left)

self.preorder(root.right)

def inorder(self, root):

if root:

self.inorder(root.left)

self.traverse\_path.append(root.val)

self.inorder(root.right)

def postorder(self, root):

if root:

self.postorder(root.left)

self.postorder(root.right)

self.traverse\_path.append(root.val)

## 二叉搜索树 Binary Search Tree

二叉搜索树，也称二叉排序树、有序二叉树（Ordered Binary Tree）、排

序二叉树（Sorted Binary Tree），是指一棵空树或者具有下列性质的二叉

树：

1. 左子树上所有结点的值均小于它的根结点的值；

2. 右子树上所有结点的值均大于它的根结点的值；

3. 以此类推：左、右子树也分别为二叉查找树。 （这就是 重复性！）

中序遍历：升序排列

## 二叉搜索树常见操作

1. 查询

2. 插入新结点（创建）

3. 删除

平均时间复杂度：O(log(n)) 最坏：O(n)

# 7.堆和二叉堆

## Heap

可以迅速找到一堆树中最大或者最小值的数据结构。

将根节点最大的堆叫做大顶堆或大根堆，根节点最小的堆叫小顶堆或小根堆

常见的堆有二叉堆、斐波那契堆等

假设是大顶堆，则常见操作（API）

Find-max: O(1)

Delete-max:O(logN)

Insert(create):O(logN) or O(1)

## 二叉堆性质

通过完全二叉树来实现（注意：不是二叉搜索树）

二叉堆（大顶堆）满足下列性质：

性质一：是一颗完全树

性质二：树中任意节点的值总是>=其子节点的值

## 二叉堆实现细节

1. 二叉堆一般通过“数组”来实现
2. 假设“第一个元素”在数组中的索引为0，则父子节点的位置关系如下：
3. 索引为i的左孩子的索引为（2\*i+1）
4. 索引为i的右孩子的索引为（2\*i+2）
5. 索引为i的父节点的索引为floor((i-1)/2)

## 二叉堆

0.根节点（堆顶元素）为：a[0]

1.索引为i的左孩子的索引为（2\*i+1）

2.索引为i的右孩子的索引为（2\*i+2）

3.索引为i的父节点的索引为floor((i-1)/2)

### Insert插入操作

1. 新元素一律先插入到堆的尾部
2. 依次向上调整整个堆的结构（一直到根即可）

HeapifyUp O(logN)

### Delete Max删除堆顶操作

1. 将堆尾元素替换到顶部（即堆顶被替代删除掉）
2. 依次从根部向下调整整个堆的结构（一直到根即可）

HeapifyUp O(logN)

注意：二叉堆是堆（优先队列priority\_queue）的一种常见且简单的实现；但并不是最优的实现

# 8.泛型递归、树的递归

## 递归 Recursion

递归 - 循环

通过函数体来进行的循环

## 代码模版

public void recur(int level, int param) {

//1.terminator

If (level > MAX\_LEVEL) {

//process result

return;

}

//2.process current logic

process(level, param);

//3.drill down

recur( level: level + 1, newParam);

//4.restore current status

}

## 思维要点

1. 不要人肉进行递归（最大误区）

2. 找到最近最简方法，将其拆解成可重复解决的问题（重复子问题）

3. 数学归纳法思维