

数据结构与算法总览

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 Bloom Filter
 - LRU Cache

算法

- For/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

注意：在头脑中回忆上面每种算法的思想对应的模板

基础

数组array

实现：在内存中开辟一段连续的地址

特性：O(1) 查询很快

O(n) 插入、删除

java源码实现

```
public void add(int index, E e) {
    checkBoundsInclusive(index);
    sizeCount++;
    if (size == data.length)
        ensureCapacity(size + 1);
    if (index != size)
        System.arraycopy(data, index, data, index + 1, size - index);
    data[index] = e;
    size++;
}
```

链表linked list

双向链表：previous/next

循环链表：tail.next → head

特性：O(1) 插入、删除很快

O(n) 查找

应用：LRU Cache

当链表元素有序的时候，如何快速查找，这就引出了跳表

跳表skip list

查找如何优化：有序链表→增加维度→跳表

跳表查询的时间复杂度分析

$n/2, n/4, n/8, \dots$ 第 k 级索引结点的个数就是 $n/(2^k)$

假设索引有 h 级，最高级的索引有 2 个结点。 $n/(2^h) = 2$ ，从而求得 $h = \log_2(n)-1$

O(log n) 查找、插入、删除

现实中跳表的形态

跳表的空间复杂度分析

原始链表大小为 n，每 2 个结点抽 1 个，每层索引的结点数： $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \dots, 2, 1$

原始链表大小为 n，每 3 个结点抽 1 个，每层索引的结点数： $\frac{n}{3}, \frac{n}{9}, \frac{n}{27}, \dots, 1$

空间复杂度是 O(n)

O(n) 跳表的空间复杂度要比原始链表高，但是由于是收敛的，所以数量级还是在O(n)

应用：Redis

<https://redisbook.readthedocs.io/en/latest/internal-datastruct/skiplist.html>

<https://www.zhihu.com/question/20202931>

高级

栈stack

后进先出

O(1) 添加、删除

O(n) 查询

队列queue

先进先出

O(1) 添加、删除

O(n) 查询

双端队列deque

O(1) 添加、删除

O(n) 查询

优先队列priority queue

O(1) 添加

O(log n) 取出 -按元素的优先级取出 (特性)

复杂度分析

Common Data Structure Operations											
Data Structure	Time Complexity				Space Complexity						
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	Access	Search	Insertion
Array	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Stack	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Queue	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Deque	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Linked List	O(1)	O(n)	O(1)	O(1)	O(1)	O(n)	O(1)	O(1)	O(1)	O(n)	O(1)
Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Graph	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Hash Table	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Heap	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Priority Queue	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Binary Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
AVL Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
B+ Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
B* Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
B-tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
Red-Black Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
AVL Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
B-tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
B+ Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)
B* Tree	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

复杂度分析

时间复杂度

空间复杂度

复杂的情况：递归

数组长度

递归深度

主定理

-用来解决所有递归函数如何计算时间负责度问题