# 数据结构与算法

## 树

- 前序遍历
- 中序遍历
- 后序遍历
- 层序遍历

## 算法思想

### 泛型递归

```
public void recur(params){
    //terminate终止条件
    //current logic当前层逻辑
    //剪枝
    //drill down 下钻
    //reverse status 回溯
}
```

### 分治

找到重复性，然后将问题拆分为多个子问题，最后合并得到结果

### 回溯

回溯算法实际上是一个枚举尝试过程，当发现不满足求解条件时，就回溯返回，尝试别的路径

组合、排序问题

### 二分查找

```
public int binarySearch(int[] array, int target) {
    int left = 0, right = array.length - 1, mid;
    while (left <= right) {
        mid = (right + left) >> 2;
        if (array[mid] == target) {
            return mid;
        } else if (array[mid] > target) {
            right = mid - 1;
        } else {
             left = mid + 1;
         }
     }
     return -1;
}
```

二分查找的前提：1）单调性 2）存在上下界 3）能够通过索引访问

### 贪心

贪心是局部最优，且不能回退；动态规划会保存当前结果，全局最优，且能回退

### 动态规划

1.找重复性，将复杂问题转化为简单子问题
2.分支+记忆化搜索
3.顺推
4.多维dp

## 搜索

- 深度优先检索：尽可能深的搜索
- 广度优先检索：按层来搜索，用队列来实现
- 双向BFS
- 启发式搜索

## 排序算法

- O(n^2)
  - 冒泡排序：嵌套循环，每次查看相邻元素如果逆序，则交换
  - 选择排序：每次找到最小值，然后放到待排序数组的起始位置
  - 插入排序：从前到后逐步构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入
  - 希尔排序
- O(nlogn)
  - 快速排序：取标杆pivot，将小元素放在pivot左边，大元素放在pivot右边，然后依次对左右子数组继续快排，以达到整个数组有序
  - 归并排序：先拆分为多个子序列，然后对每个子序列采用归并排序，最后合并
  - 堆排序
- O(n)
  - 计数排序
  - 桶排序
  - 基数排序

| 排序方法 | 时间复杂度（平均） | 时间复杂度（最坏） | 时间复杂度（最好） | 空间复杂度 | 稳定性 |
|---|---|---|---|---|---|
| 插入排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 希尔排序 | $O(n^{1.3})$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 不稳定 |
| 选择排序 | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(1)$ | 不稳定 |
| 堆排序 | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(1)$ | 不稳定 |
| 冒泡排序 | $O(n^2)$ | $O(n^2)$ | $O(n)$ | $O(1)$ | 稳定 |
| 快速排序 | $O(nlog_2n)$ | $O(n^2)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | 不稳定 |
| 归并排序 | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(nlog_2n)$ | $O(n)$ | 稳定 |
| | | | | | |
| 计数排序 | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | $O(n+k)$ | 稳定 |
| 桶排序 | $O(n+k)$ | $O(n^2)$ | $O(n)$ | $O(n+k)$ | 稳定 |
| 基数排序 | $O(n*k)$ | $O(n*k)$ | $O(n*k)$ | $O(n+k)$ | 稳定 |

## 字符串匹配

- BF(brute force暴力法)
- RK：text子串的hash值与pat的hash值比较
- KMP算法
- BM(Boyer-Moore)

## 存储结构

- 数组
- 链表

## 逻辑结构

### 线性结构

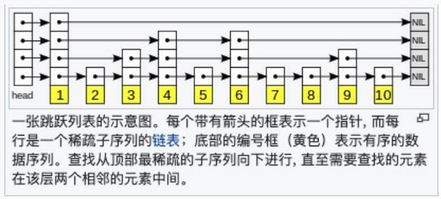
- 栈：先进后出 push()、pop()、peek()
- 队列：先进先出 offer()、poll()、peek()
- 优先队列：PriorityQueue，数组实现的小顶堆
- 双端队列：ArrayDeque、LinkedList

### 非线性结构

- 哈希表
- 集合
- 树
  - 二叉树
    - 完全二叉树
    - 满二叉树
    - 二叉搜索树：任一节点的左子树所有节点都小于该节点，右子树所有节点都大于该节点。二叉搜索树的中序遍历是升序的。
    - 平衡二叉搜索树
      - AVL树：任一节点的左右子树高度差不超过1
      - 红黑树：1.节点是红色或黑色 2.根是黑色 3.所有叶子都是黑色，NIL节点 4.每个红色节点必须有两个黑色的子节点 5.从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。
  - 多路搜索树
    - B树
    - B+树
    - 2-3树
- 图：可用邻接矩阵表示，分为有无权、有无向

### 其他结构

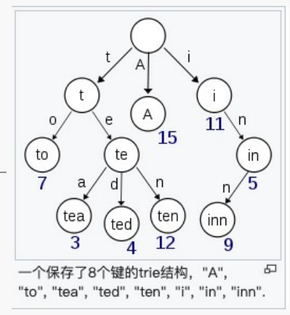
- 跳表




一张跳跃列表的示意图。每个带有箭头的框表示一个指针，而每行是一个稀疏子序列的链表；底部的编号框（黄色）表示有序的数据序列。查找从顶部最稀疏的子序列向下进行，直至需要查找的元素在该层两个相邻的元素中间。

**链表加多级索引的结构**

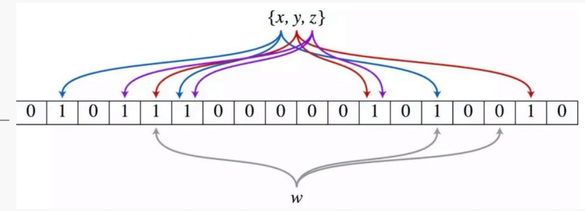
- 堆：任一节点>=(或<=)它的子节点，通常用数组实现
- 字典树Trie




一个保存了8个键的trie结构，"A", "to", "tea", "ted", "ten", "i", "in", "inn".

**又称前缀树，常用于搜索提示**

- 并查集：并查集主要处理两个不相交集合的查询和合并问题，场景：亲戚问题、朋友圈、武林帮派。
- 布隆过滤器




布隆过滤器可以快速的判断一个元素是否存在，有一定的误判率。应用场景：垃圾邮件、评论过滤、缓存击穿

- LRU Cache：Least Recently Use Cache，淘汰最近最少使用的缓存