算法&数据结构

1. 数据结构

1.1 堆

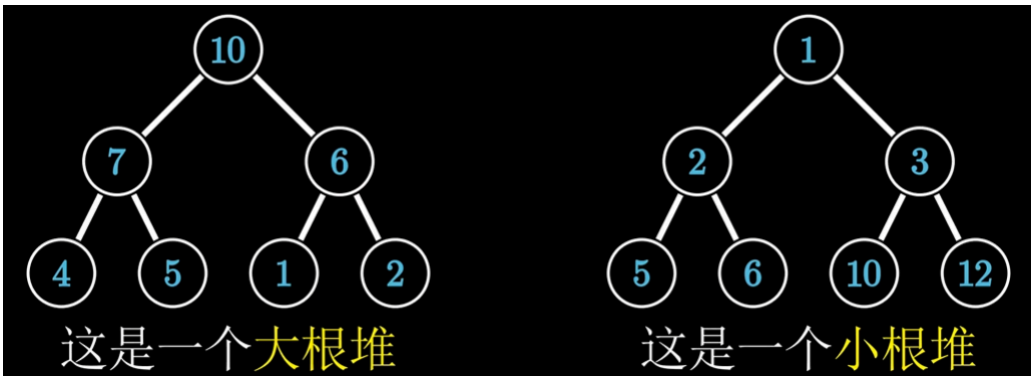
1.1.1 堆的定义

（这里指的是数据结构里的堆，不是操作系统的堆，二者不一样注意区分）

堆一般都使用**完全二叉树**，完全二叉树有这些性质：

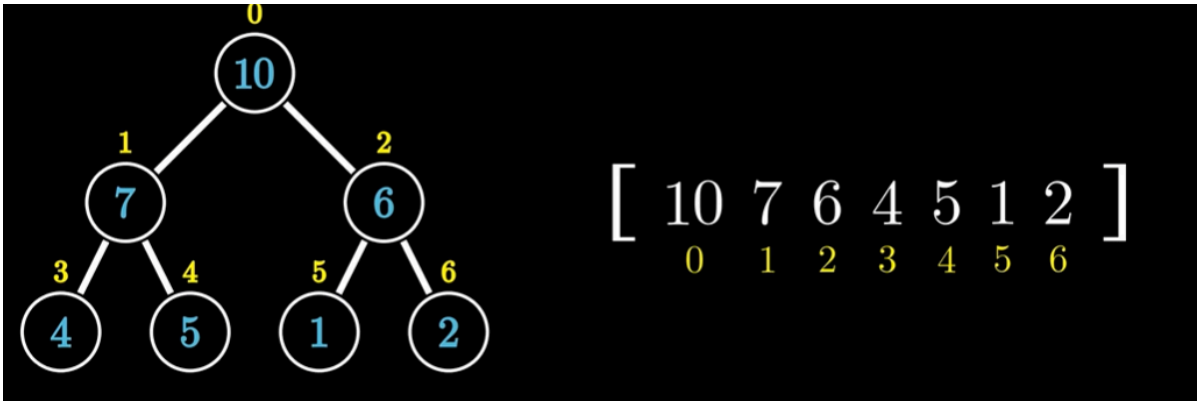
- 完全二叉树**只允许最后一行不为满**
- 最后一行必须从左到右排序
- 最后一行元素之间不可以有间隔

同时，堆还有自己独有的重要性质——**堆序性**：父节点要比左右子节点数值大（小），即大根堆（小根堆）



1.1.2 堆的储存

基于完全二叉树的性质，堆可以使用一个**一维数组**来存储，如下图所示：



节点间**规律**如下：

- 节点下标为 i;
- 左子节点下标为 $2i+1$;
- 右子节点下标为 $2i+2$;

1.1.3 堆的基本操作（上滤，下滤）

（以下说明**以大根堆为例子**，小根堆同理）

下滤 复杂度$O(logN)$

当父节点的值小于子节点时不满足大根堆的性质，此时需要调整该节点的位置，这个操作称为下滤，具体操作如下：

- 对比左右子节点找到最大的节点，将需要调整的节点与该节点进行交换，这一步的目的是保证调整后的父节点一定比子节点大
- 当调整后仍不满足大根堆的性质时，继续调整，直到满足大根堆性质或该节点已经是叶子节点

上滤 复杂度$O(logN)$

上滤的操作主要用于新元素插入到堆中时，当树的最后一个节点破坏了堆序性，即该节点大于父节点时需要进行上滤操作，具体操作如下：

- 将该节点与父节点进行比较，如果大于父节点则和父节点进行交换
- 当调整后仍不满足大根堆的性质时，继续调整，直到满足大根堆性质或该节点已经是根节点

1.1.4 堆的应用（优先队列）

优先队列有**两种操作**，一个是**插入元素**，另一个是**弹出**最大元素（大根堆）或最小元素（小根堆）

插入元素：复杂度$O(logN)$

将元素插入堆的末尾，执行**上滤操作**即可

弹出：

将根节点弹出并将堆的**最后一个元素移动到根节点**，执行**下滤操作**

1.1.5 堆的应用（堆排序）

简单来讲就是构建大根堆（小根堆）后将元素依次弹出，此时元素一定是有序的，具体请跳转**2.2.7**查看

1.2 二叉树

```
struct TreeNode{
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
}
```

1.2.1 前序遍历

中左右

- **递归法**

```

```

- **迭代法**

```

```

1.2.2 中序遍历

左中右

- **递归法**

- **迭代法**

1.2.3 后序遍历

左右中

- **递归法**

- **迭代法**

2. 算法

2.1 B树

问：为什么要保证m阶B树非根节点的关键字的数量要大于等于$ceil(m/2)-1$?

因为B树某个节点的分裂条件是关键字的数量大于m-1也就是等于m的时候，这时候会从中间分为两个节点，两个节点关键字的数量是ceil(m/2)-1,ceil(m/2)-1或者ceil(m/2), ceil(m/2)+1，基于这样一个原因必须保证删除后关键字的数量要大于等于$ceil(m/2)-1$，不然就破坏了B树的特性

2.2 排序算法

十大经典排序算法 - 冰狼爱魔 - 博客园 (cnblogs.com)

2.2.1 选择排序

性质：1、时间复杂度：$O(n^2)$ 2、空间复杂度：O(1) 3、非稳定排序 4、原地排序

不断找未排序区最小的元素，将其与未排序区第一个进行交换，同时减小未排序区的范围

2.2.2 插入排序

性质：1、时间复杂度：$O(n^2)$ 2、空间复杂度：O(1) 3、稳定排序 4、原地排序

假设第一个元素已经排好序，不断选取未排序区的元素将其插入到已排序区的对应位置，使其大于前一个小于后一个

初始数据有序程度越高或数据集越小，越高效（移动少）

2.2.3 冒泡排序

性质：1、时间复杂度：$O(n^2)$ 2、空间复杂度：O(1) 3、稳定排序 4、原地排序

适用于查找最大（小）的第n个元素

将第一个与第二个比较，如果大于第二个则交换位置，不断交换，最终排在最右边的一定就是未排序区最大的元素

- **优化：**

当遍历一次后没有发生交换则说明已经排好可以直接返回

2.2.4 希尔排序

性质：1、时间复杂度：$O(NlogN)$ 2、空间复杂度：O(1) 3、非稳定排序 4、原地排序

希尔排序是对插入排序的一种优化，用来减少移动次数，先让间隔为n/2的元素彼此有序，再让n/4的元素彼此有序，每排一次，数组都会比之前更加有序一些，这样当最后要进行将间隔为1的元素之间彼此有序的操作时（同插入排序），要移动的元素就会少很多

2.2.5 归并排序

性质：1、时间复杂度：$O(NlogN)$ 2、空间复杂度：O(n) 3、稳定排序 4、非原地排序

归并（递归，合并），基于分治思想的排序。通过递归的方式将大的数组一直分割，直到数组的大小为1，此时只有一个元素，那么该数组就是有序的了，之后再把两个数组大小为1的合并成一个大小为2的，再把两个大小为2的合并成4的 ..... 直到全部小的数组合并起来。

不考虑递归所消耗的空间，空间复杂度为O(1)的方法：[「归并排序：题目一」如何实现一个空间复杂度为 O(1) 的归并排序？-CSDN博客]

2.2.6 快速排序

性质：1、时间复杂度：$O(NlogN)$ 2、空间复杂度：O(logn) 3、非稳定排序 4、原地排序

[【漫画】不要再问我快速排序了 (qq.com)]

和归并排序一样采用了分治的思想，但是它不需要额外的数组，也不需要把额外的数组内容复制到原数组，但是因为它用到了递归，而递归本身要保存数据所以占用一定空间导致空间复杂度为O(logn)，

每次取一个主元调整它的位置，使其左边的小于等于它，右边的大于等于它

```
class Sort{
```

```
    static void QuickSort(vector<int>& v, int left, int right){
        if(left<right){
            int mid = partition(v, left, right);
            QuickSort(v, mid + 1, right);
            QuickSort(v, left, mid - 1);
        }
    }

    static int partition(vector<int>& v, int left, int right){
        int pivot = v[left];  //要调整的元素
        int i = left + 1;
        int j = right;
        while(true){
            while(i<=j && v[i] <= pivot) i++; //向右走找到第一个比pivot大的元素下标
            while(i<=j && v[j] >= pivot) j--; //向左走找到第一个比pivot小的元素下标

            if(i>=j) break;

            swap(v[i], v[j]);
        }
        swap(v[pivot], v[j]);

        return j;
    }
}
```

2.2.7 堆排序

【从堆的定义到优先队列、堆排序】 10分钟看懂必考的数据结构——堆*哔哩哔哩*bilibili

性质：1. 时间复杂度：$O(NlogN)$ 2、空间复杂度：O(1) 3、非稳定排序 4、原地排序

堆排序的整体过程是先将要排序的数组构造成大根堆（小根堆），再弹出根节点，但是要注意的是和优先队列的弹出不同，假设数组分为**未排序部分**和**已排序部分**，弹出的根节点时会把未排序部分的最后一个节点放入根节点并**把根节点放入最后一个节点的位置**，减少未排序部位的长度继续弹出，直到所有的都弹出此时就是一个有序的数组了，因为弹出的数会放在数组的后半部分，所以**使用大根堆排序的结果是正序的，使用小根堆排序的结果是倒序的**

```
class Sort{
    //大根堆为例
    void HeapSort(vector<int>& v){
        int n = v.size();
        //先构建大根堆
        for(int i = (n - 2)/2, i>=0; i--){
            DownAdjust(v, i, n-1);
        }
        //开始从堆顶一个个删除放入尾部，同时调整大根堆保证堆顶为最大值
        for(int i = n-1; i>0; i--){
            swap(v[0], v[i]);
            DownAdjust(v, 0, i-1);
        }
    }
```

```
    void DownAdjust(vector<int>& v, int parent, int n){
        //定位左孩子节点位置
        int child = parent*2 + 1;
        //开始下沉
        while(child <= n){
            if(child+1<=n && v[child+1] > v[child]){
                child++;
            }
            if(v[parent] >= v[child]) break; //如果父节点等于或大于最大子节点，则直接返
回
            swap(v[parent], v[child]);
            parent = child;
            child = parent*2 + 1;
        }
    }
}
```

2.2.8 计数排序

性质：1、时间复杂度：O(n+k) 2、空间复杂度：O(k) 3、稳定排序 4、非原地排序

适合于最大值和最小值的差值不是不是很大的排序

基本思想：就是把数组元素作为数组的下标，然后用一个临时数组统计该元素出现的次数，例如 temp[i] = m, 表示元素 i 一共出现了 m 次。最后再把临时数组统计的数据从小到大汇总起来，此时汇总起来是数据是有序的。

2.2.9 桶排序

2.2.10 基数排序

2.2.11 总结

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

2.3 单调栈

单调栈适合于给定数组，求某一位置元素左边或右边第一个比它大或小的元素位置，所谓单调栈就是指**栈里的元素是保持递增或递减的**，注意，**为了能够知道对应元素的位置所以单调栈中存放的是元素的下标**，但是**进行对比时对比的是对应下标所对应的值**

当单调栈中从顶部到底部单调递增则求的是第一个比它大的元素，如果单调递减则求的是第一个比它小的元素，单调栈的作用就是存放当前遍历过的元素下标，同时它们的位置根据对应下标数据的大小关系在插入时进行调整

单调栈的操作：

情况一：只找一边（以找右边第一个大的元素，从栈顶到栈底单调递增为例）

假设数组 v ，单调栈 s，存放右边比自己大的第一个元素下标的数组 cal ，$cal[i]$ 表示右边第一个比 $v[i]$ 大的元素下标为 $cal[i]$ ，时刻记住栈里存放的是元素的下标

- 当栈为空时，直接将下标 0 存入栈中，即执行 `s.push(0)` 操作；
- 当遍历到新的元素下标 i 时，将该元素 $v[i]$ 与栈顶元素 $v[s.top()]$ 进行对比：
  - 如果 $v[i] <= v[s.top()]$ ，直接执行 `s.push(i)` ；
  - 如果 $v[i] > v[s.top()]$ ，表示找到了第一个比 $v[s.top()]$ 大的元素，执行 `cal[s.top()] = i` 操作，因为此时已经找到了比 $v[s.top()]$ 大的元素了，所以没有必要存在于栈中，执行 `s.pop()` 操作，不断执行该步骤直到栈为空或栈顶元素大于 $v[i]$ ，此时将下标 i 放入栈顶，即 `s.push(i)`；

```
//以找右边第一个大的元素，从栈顶到栈底单调递增为例
vector<int> FindFirstMax(vector<int>& v){
    int n = v.size();

    stack<int> s;
```

```
    vector<int> cal(n, -1); //初始化为-1，当全部计算后仍为-1则表示右边没有比该元素大的
    for(int i = 0; i < n; i++){
        while(!s.empty() && v[i] > v[s.top()]){
            cal[v.top()] = i;
            v.pop();
        }

        v.push(i);
    }

    return cal;
}
```

情况二：两边都找（以找第一个大的元素，从栈顶到栈底单调递增为例）

假设数组 v ，单调栈 s，存放左边比自己大的第一个元素下标的数组 $lcal$ ，存放右边比自己大的第一个元素下标的数组 $rcal$

整体操作与只找一边类似，只是在将新的元素下标 i 放入栈前要注意：

- 当 i 可以放入栈顶时，此时的栈顶一定是 i 左边第一个比它大的，所以如果需要找两边比自己大的元素位置，在 `s.push()` 操作前要先 `lcal[i] = s.top()`

```
//以找第一个大的元素，从栈顶到栈底单调递增为例
vector<int> FindFirstMax(vector<int>& v){
    int n = v.size();

    stack<int> s;
    vector<int> lcal(n, -1), rcal(n, -1); //初始化为-1，当全部计算后仍为-1则表示没有比该元素大的
    for(int i = 0; i < n; i++){
        //找右边
        while(!s.empty() && v[i] > v[s.top()]){
            rcal[v.top()] = i;
            v.pop();
        }
        //找左边
        if(!s.empty()){
            lcal[i] = s.top();
        }
        v.push(i);
    }

    return cal;
}
```

相关例题：

739. 每日温度，42. 接雨水， 84. 柱状图中最大的矩形，

2.4 寻路算法

https://www.redblobgames.com/pathfinding/a-star/introduction.html

2.4.1 Dijkstra最短路径算法

2.4.2 A*搜索算法

A*搜索算法类似于广度优先算法，基于已走过的路径向未走过的路径慢慢扩充，但是相比BFS那种漫无目的没有方向性的查找不同，A*算法使用**启发式函数**给每个可以走的路径计算一个**代价值**，该代价值预估到达目标还需要多少步，基于这一信息就可以优先走代价值低的路径（优先队列）以此来减小搜索的时间开销。

其中代价值由两部分构成：当前代价（已走步数） + 预估代价（预计还有多少步） $f = g + f$

预估代价是一种估计，无视中间可能的障碍，计算它可以使用**启发式函数**如曼哈顿距离（$|x_1 - x_2| + |y_1 - y_2|$）或欧几里得距离（$(x_1 - x_2)^2 + (y_1 - y_2)^2$）以及其他方法

2.5 倍增

1483. 树节点的第 K 个祖先

2.6 KMP算法

查找B序列在A序列出现的位置，当A的 i 位置的元素与B的 j 位置元素匹配失败时，传递有两个信息:

- $A[i] != B[j]$;
- B下标 j 之前的元素匹配成功

基于这一信息，当A的 $[i-j, i-1]$ 子串的后缀集合与B的 $[0, j-1]$子串的前缀集合有交集时，找到交集里最长的那个元素，将B的指针指向该前缀的下一位 ，继续尝试与A的 i 位置元素进行比较。

这部分可以进一步优化，因为A的 $[i-j, i-1]$ 范围子串与B的 $[0, j-1]$子串一定一样，所以变成了 B的 $[0, j-1]$子串自己的前缀和后缀进行比较，基于这一信息，可以在B串与A串进行对比之前就求出当匹配不成功时，B的指针应该重新指向哪里，一般将存储这一信息的数组声明为**前缀表next**，$next[j]$表示 $[0, j-1]$ 范围的**最长公共前后缀**的长度，该长度的数值与 j 要回退到的下标数值一致，所以整体步骤为:

- A数组长度为n，B数组长度为m， i , j 初始化为0，`next[0] = -1`
- 如果`A[i] == B[j]`，`i++, j++` 继续匹配
- 如果`A[i] != B[j]`，回溯 j 到 $next[j]$ ，直到`A[i] == B[j]`，这里有一种情况B串需要移动到 i 后面:
    - j 回溯到0时也不能满足`A[i] == B[j]`，此时 j 已经不能后移，忽略 j，增加 i，直到`A[i] == B[j]`
- 当`j == m`时,输出位置，继续匹配可能还存在的相同序列

```
void CalNext(vector<int>& next, const string& s){
    next[0] = -1;
    int j = -1;
    int n = s.size();
```

```
    for(int i = 1; i < n; i++){
        while(j>=0 && s[i] != s[j+1]){
            j = next[j];
        }
        if(s[i] == s[j+1]){
            j++;
        }
        v[i] = j;
    }
}
```

2.7 回文串

查找回文串的方法就是从中心扩展，中心可能是一个字符或者2个字符

以leetcode中查找回文子串数量的题为例：**647. 回文子串**

```
class Solution {
public:
    int countSubstrings(string s) {
        int num = 0;
        int n = s.size();
        for(int i=0;i<n;i++)//遍历回文中心点
        {
            for(int j=0;j<=1;j++)//j=0,中心是一个点，j=1,中心是两个点
            {
                int l = i;
                int r = i+j;
                while(l>=0 && r<n && s[l--]==s[r++])num++;
            }
        }
        return num;
    }
};
```