算法与数据结构一直是计算机科学中特别重要的基础知识，Pascal之父Nicklaus Wirth凭借一句话

获得了图灵奖，“Algorithms + Data Structures = Programs”，他也写了这本比较经典的计算机理论书籍

《算法+数据结构=程序》，有兴趣的可以搜来翻阅。

在本科计算机系列课程中，相信老师也强调过其重要性，工作后，似乎没人强调它的重要性了，日常的

开发工作中，用到的也比较少的。感觉工作之余，是时候该好好整理一下这方面的知识了。用到的语言是C++，

代码放在github上，传送门<https://github.com/wuatai/dataStructure-Algorithm>

我会从排序算法来分享自己的数据结构与算法复习总结。首先先复习一下时间复杂度为O(n^2)的排序算法。

因为O(n^2)算法最基础，编码简单易于实现，而且通常，在尝试解决一个问题时，都会先用最简单的思路去试着

解决它，这个过程可能会加深我们对这个问题本身的理解。进而提出更优的解法。

选择排序基本思路（从小到大排序）

1.将数组分为2部分，一部分是有序的，一部分是无序的。

2.找到无序最小元素的位置，放在有序数组的最后一个位置（与无序的第一个元素交换位置）。

3.循环1和2，直到无序部分没有了元素。

8 6 2 3 1 5 7 4

1 6 2 3 8 5 7 4

1 2 6 3 8 5 7 4

1 2 3 6 8 5 7 4

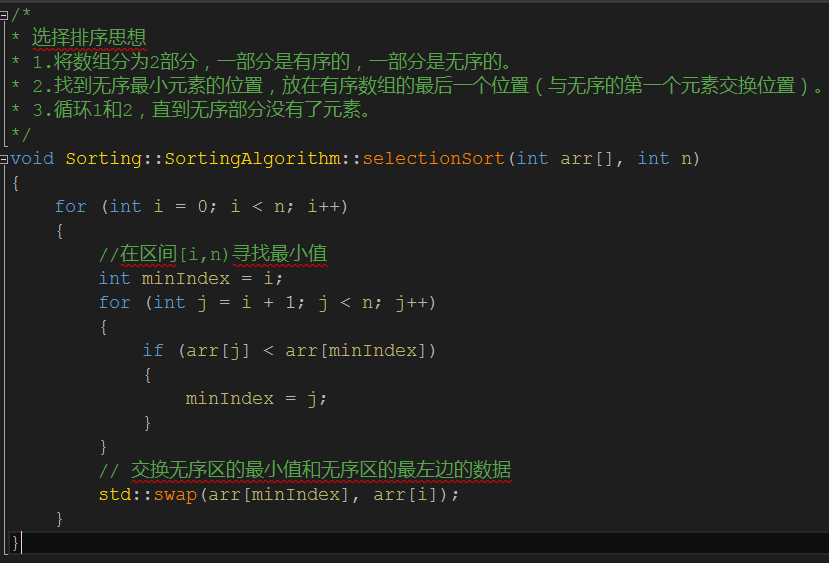
1 2 3 4 8 5 7 6

1 2 3 4 5 8 7 6

1 2 3 4 5 6 7 8

1 2 3 4 5 6 7 8

主要实现代码如下：



插入排序

插入排序的基本思想跟码扑克牌的思想类似。

1. 把数组分成两部分，无序和有序
2. 把无序数组的第一个元素，插入在有序数组的合适位置
3. 重复1和2

例子

8 6 2 3 1 5 7

6 8 2 3 1 5 7

2 6 8 3 1 5 7

2 3 6 8 1 5 7

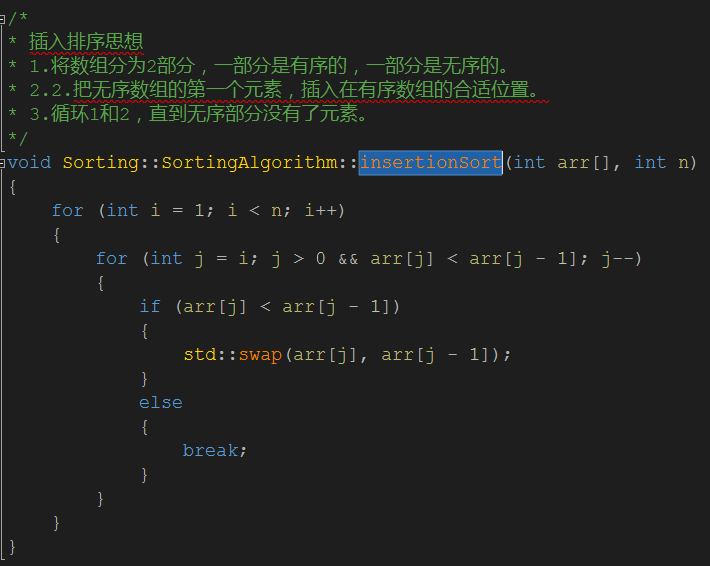
1 2 3 6 8 5 7

1 2 3 5 6 8 7

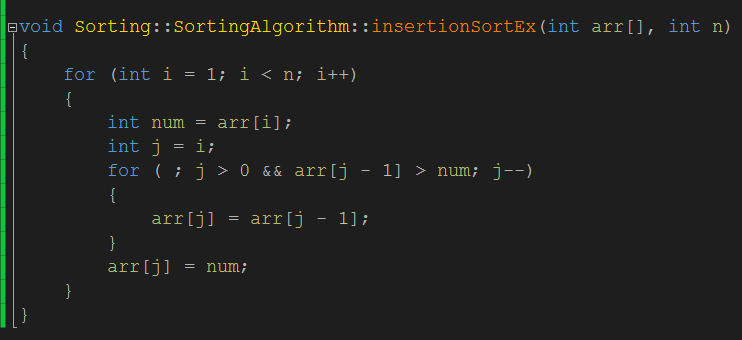
1 2 3 5 6 7 8

第二个步骤，让其与有序数组从右到左逐个比较，然后交换合适的位置。

代码如图



优化



比较：

我用了10000的数据量进行试验

对于近乎有序的数组



对于无序数组



选择排序：无论是近乎有序还是无序，选择排序都要走完2层循环

插入排序：对于近乎有序的数组，插入排序表现很好，对于无序数组，表现甚至不如选择排序，因为大量的交换和赋值操作。

两者最差的情况都是O(n^2)，但是对于近乎有序的数组，插入排序算法效率甚至比O(nlogn)表现还要好。

归并排序

O(nlogn)算法随着n规模的变大，比O(n^2)要快很多，这是学习O(nlogn)复杂度算法的原因。

首先，看下归并排序，基本思想：

1. 将待排序数组从中间一分为二，对左右两边进行递归分割操作，直到不可再分，得到n个有序子序列。
2. 对n个有序子序列递归执行合并操作，最终得到有序的序列。
3. 合并操作。利用一个额外数组，对下一层级的2个子数组进行排序赋值，因为子数组都是有序的，比较2数组最左边的元素，哪个小就把哪个放入额外数组，并且index++，存放完毕后，把额外数组复制到原来的位置上。

过程用一个例子表示：

8 6 2 3 1 5 7 4 //初始状态

8 6 2 3 1 5 7 4

8 6 2 3 1 5 7 4

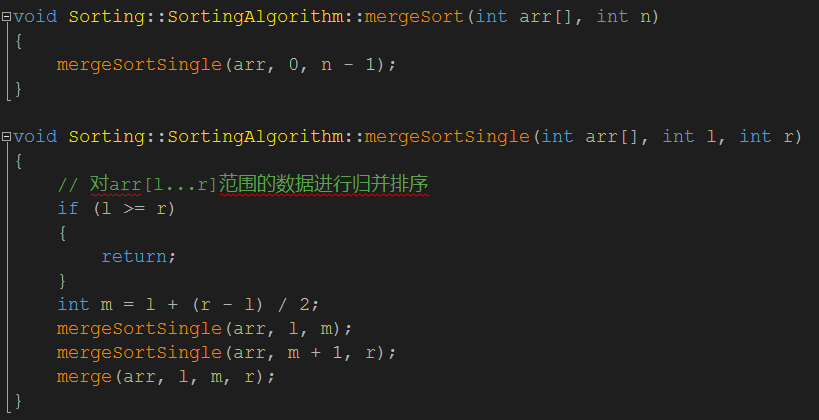
8 6 2 3 1 5 7 4 //完成分割

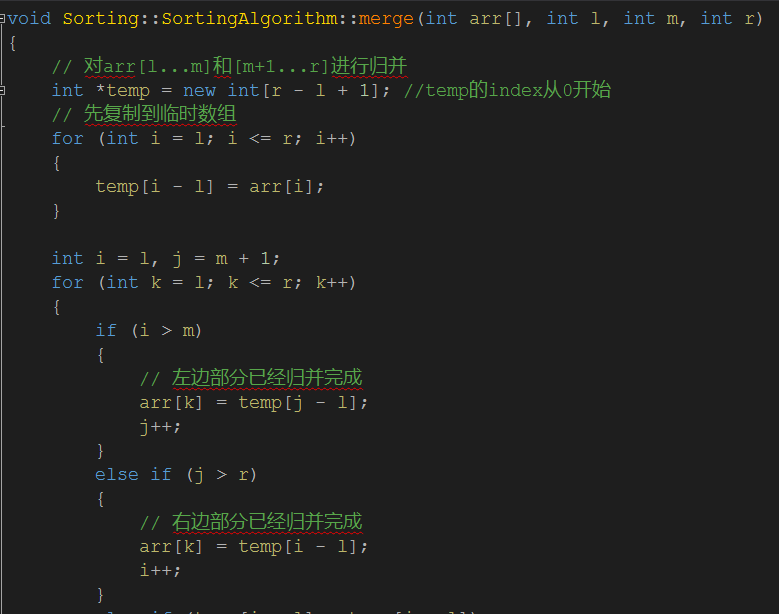
6 8 2 3 1 5 4 7

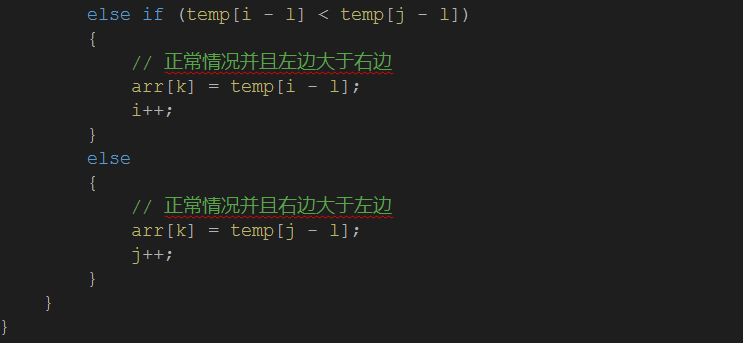
2 3 6 8 1 4 5 7

1 2 3 4 5 6 7 8 //完成归并

代码如下：



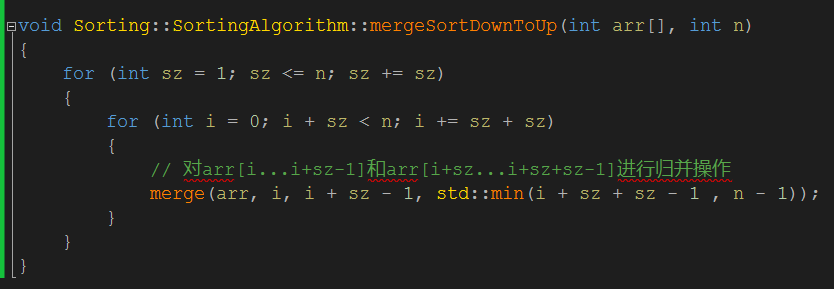




与选择和插入排序进行的比较，1w个数据，结果如下



自底向上法：



不用递归，改用自底向上会快一些，因为没有了大量的栈操作。

快速排序

快速排序的基本思想：

1. 选定一个基准数，通常是数组第一个元素。
2. 通过partition操作，把数组分成2部分，左边都小于等于基准数，右边都大于基准数。
3. 将左边和右边分别进行1,2操作，直到单个数组元素为1.

一次Partition操作案例如下，设j为左侧小于等于基准数的最后一个位置,l 为基准位置0

5 6 2 3 1 8 7 4 //初始状态，j = 0

5 6 2 3 1 8 7 4 // j = 0

5 2 6 3 1 8 7 4 // j = 1

5 2 3 6 1 8 7 4 // j = 2

5 2 3 1 6 8 7 4 // j = 3

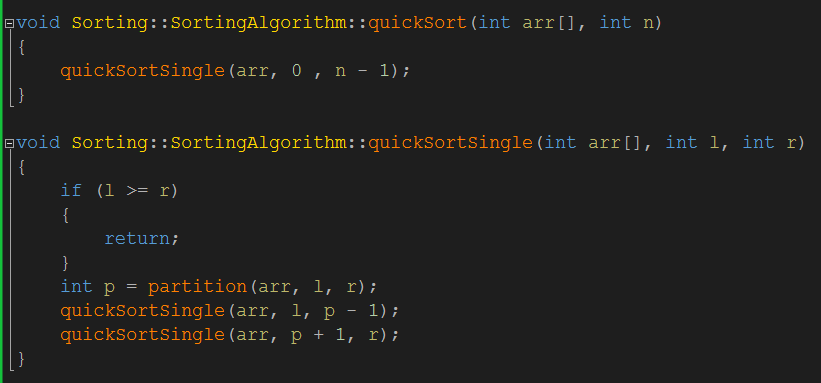
5 2 3 1 6 8 7 4 // j = 4

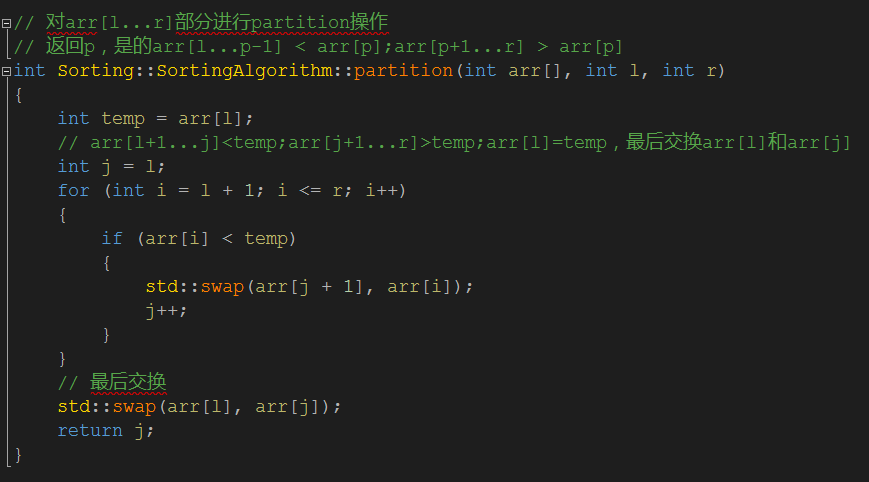
5 2 3 1 6 8 7 4 // j = 4

5 2 3 1 4 8 7 6 // j = 4

4 2 3 1 5 8 7 6 // 最后交换a[l]和a[j+1]

至此就完成了一次partition操作。完成后，把数组分为了2部分，之后分别对这2部分进行快速排序，下面用递归的方法实现：





这样会导致一个问题，对于有序的数组，每次partition之后，左边的数组个数始终会是1，从而快速排序退化成了O(n^2)的算法了，这是不能接受的。

下面来优化它。

优化1：随机标的（针对近乎有序的数组）

选取随机的数据作为标定元素，而不是每次都是第一个。

在partition中，Arr[rand()%(r-l+1)+l] 与 arr[l]交换。

优化2：双路快排（针对大量重复值的数组，如100w个1-10之间的数据）

基本思想：从左右两端出发，左边遇到大于标的值元素，停下来，右边遇到小于标的物的元素停下，交换这两个元素，左右两边再继续进行，知道左边的游标和右边游标相遇，停止。这样结束了一个partition操作。

一次Partition操作案例如下

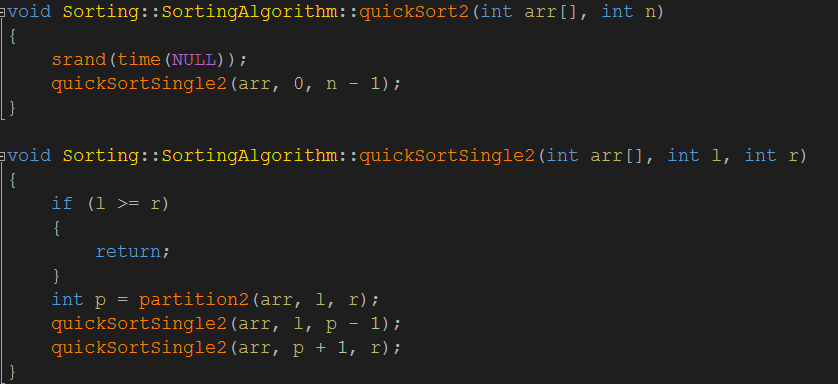
5 6 2 3 1 8 7 4 // 初始状态，i = 1， j = 7

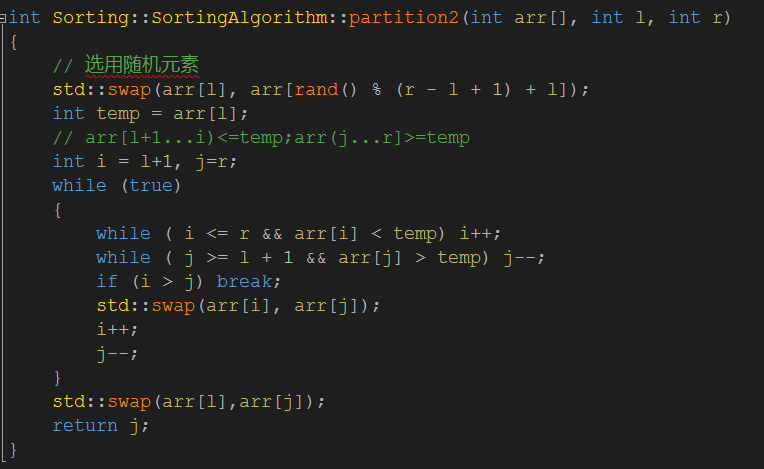
5 4 2 3 1 8 7 6 // 交换了一次 ,i = 2, j = 6

5 4 2 3 1 8 7 6 // 结束了，i = 5, j = 4

1 4 2 3 5 8 7 6 // 交换arr[l]和arr[j]

代码如图





归并排序和快速排序的衍生问题

Merge Sort和Quick Sort都是用了分治算法

题目1：求逆序对的个数

暴力解法：考察每一个数对，算法复杂度：O(n^2)

归并思想解法：

题目2：取数组中第n大的元素

排序解法：O(nlogn)

快速排序思想：O(n) = n + n/2 + n/4 + n/8 + ... + 1

数据结构堆

堆和优先队列Heap and Priority Queue

什么是优先队列？

普通队列：先进先出；后进后出

优先队列：出队顺序和入队顺序无关；和优先级相关

|  |  |  |
| --- | --- | --- |
|  | 入队 | 出队 |
| 普通数组 | O(1) | O(n) |
| 顺序数组 | O(n) | O(1) |
| 堆 | O(lgn) | O(lgn) |

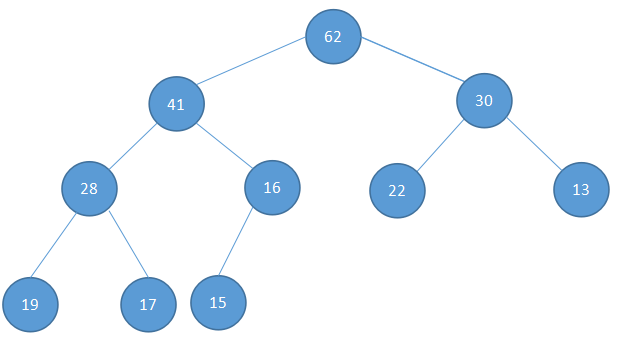
总共N个请求：

使用普通数组或者顺序数组，最差情况：O(n^2)

使用堆：O(nlogn)

堆的基本实现

二叉堆Binary Heap



二叉堆是一颗完全二叉树，堆中某个节点的值总是不大于父节点的值（最大堆）

用数组存储二叉堆：根节点从1开始

[-, 62, 41, 30, 28, 16, 22, 13, 19, 17, 15]

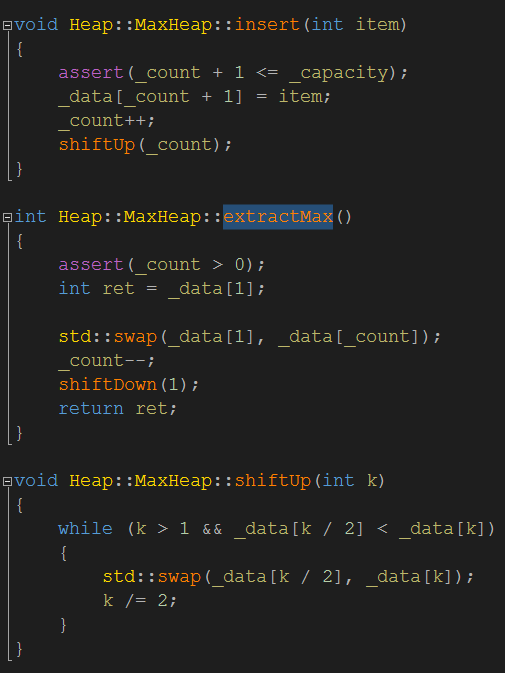
Parent(i) = i / 2

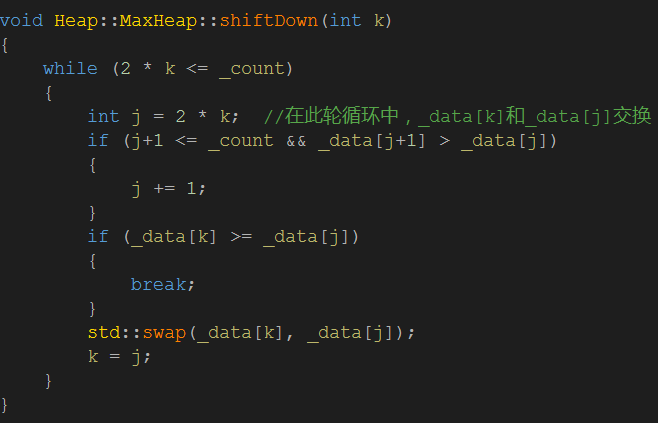
Left child(i) = 2 \* i

Right child(i) = 2 \* i + 1

代码如下



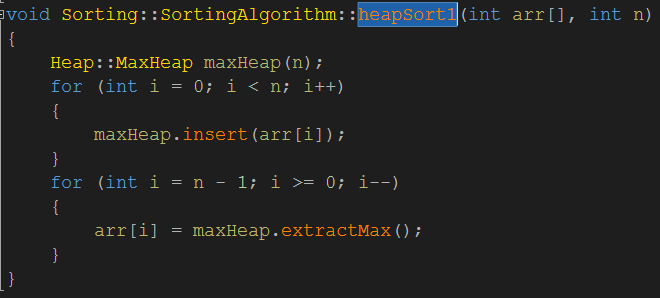




堆排序

方法1：

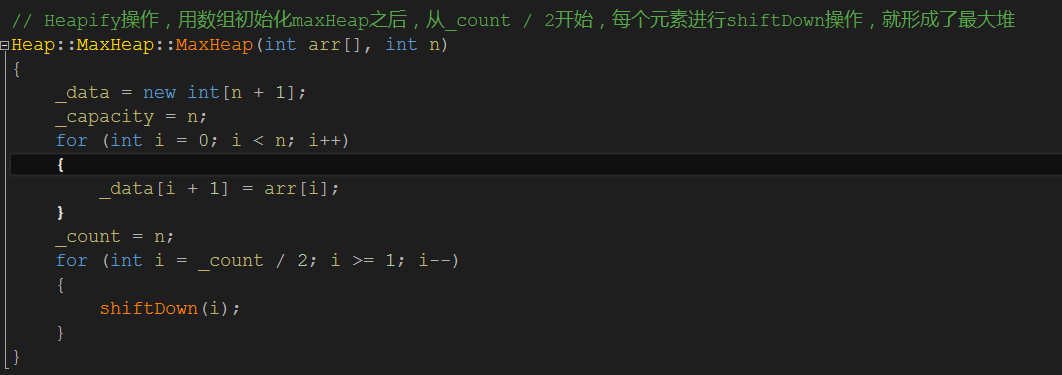
调用最大堆的insert函数，逐个插入数组的数据，之后再调用extractMax函数，把最大的元素找出来，代码如下



这样就完成了堆排序了，但是还是有优化空间的，用heapify替代逐个insert。优化前，和双路快排，10w个数据的比较如下：



Heapify操作，用数组作为MaxHeap的构造函数参数，从\_count / 2开始，每个元素进行shiftDown操作，就形成了最大堆了。比逐个insert效率要快。代码如下



与双路快排和insert操作的对排序相比较，heapify比insert操作出的堆排序还是要快一些的。

因为将n个元素逐个插入到一个空堆中，算法复杂度是O(nlogn)的

Heapify的过程，算法复杂度为O(n)，证明过程可以查一下网上的资料。



但是比快速排序还是要慢很多的，因此，堆排序还是比较少用到的，堆这种数据结构更多的是用在动态数据的维护上，

优化的堆排序(HeapSort)

之前的堆排序是先把数组放入到堆，再从堆放回到数组。其实完全可以利用堆排序的思想，直接在数组原地进行堆排序。

堆排序思想：注意这里堆从0开始索引

Parent(i) = (i-1)/2

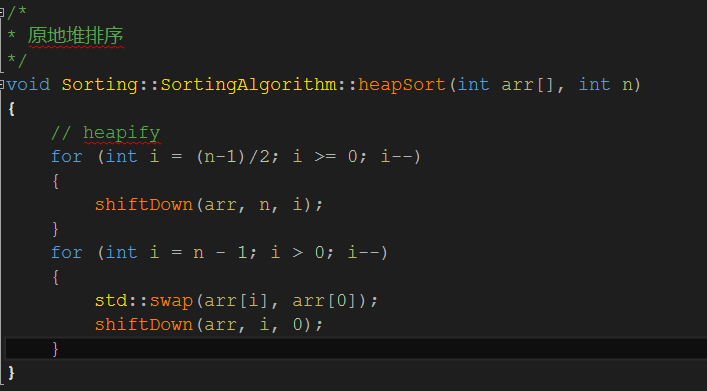
Left child(i) = 2\*i + 1

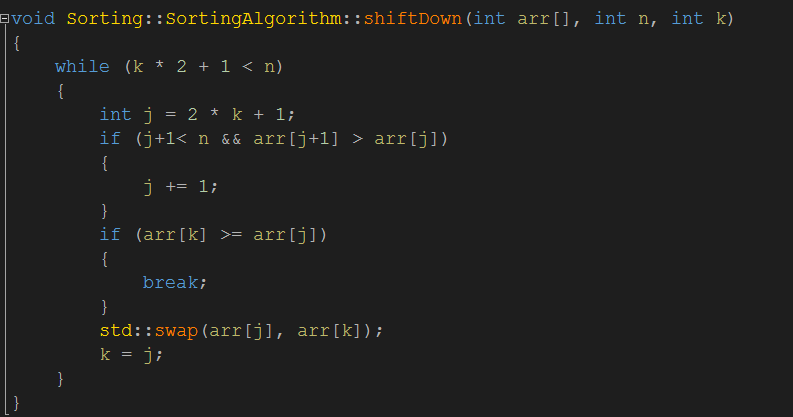
Right child(i) = 2\*i + 2

最后一个非叶子节点的索引是(count-1)/2

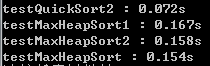
1. 对数组进行heapify操作，是的数组成为一个最大堆
2. 将数组第一个元素（最大值）与数组最后一个元素交换，则最后的元素是最大值。
3. 未排序数组大小减一，后面只对未排序的数组重复12操作，直到数组大小为1.

代码如下





与之前2种堆排序比较，发现消耗的时间又少了，但是任然是O(nlogn)的算法。



排序算法总结

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 平均时间复杂度 | 原地排序 | 额外空间 | 稳定 |
| 插入排序 | O(n^2) | 是 | O(1) | 是 |
| 归并排序 | O(nlogn) | 否 | O(n) | 是 |
| 快速排序 | O(nlogn) | 是 | O(logn) | 否 |
| 堆排序 | O(nlogn) | 是 | O(1) | 否 |

二叉搜索树

二分搜索树Binary Search Tree

查找问题是计算机中非常重要的基础问题

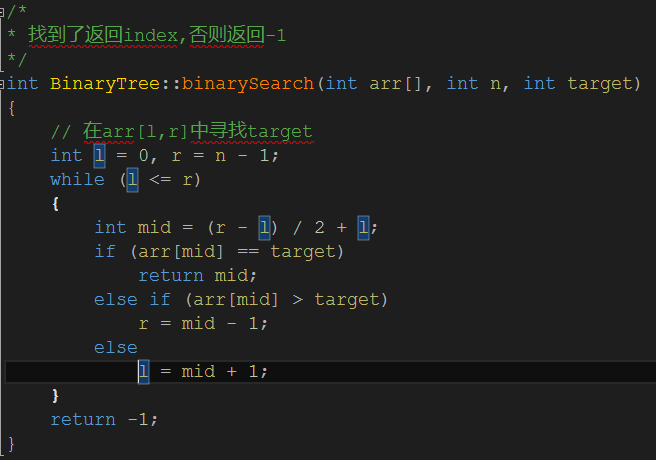
先介绍非常经典的二分查找法

二分查找法：Binary Search

对于有序的数组才能使用二分查找法（排序的作用）

二分查找法的思想在1946年提出，但是有意思的是，第一个没有bug的二分查找法在1962年才出现。

代码如下



二分搜索树的优势

|  |  |  |  |
| --- | --- | --- | --- |
|  | 查找元素 | 插入元素 | 删除元素 |
| 普通数组 | O(n) | O(n) | O(n) |
| 顺序数组 | O(logn) | O(n) | O(n) |
| 二分搜索树 | O(logn) | O(logn) | O(logn) |

高效：不仅可查找到数据；还可以高效地插入，删除数据，动态维护数据

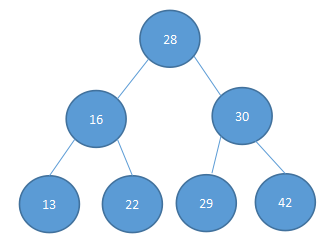
可以方便地回答很多数据之间的关系问题：min,max,floor,ceil,rank.select

二分搜索树定义：

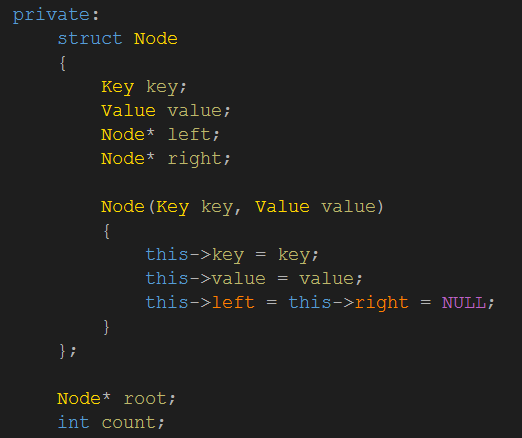
二叉树（不一定要是完全二叉树）

每个节点键值大于左孩子

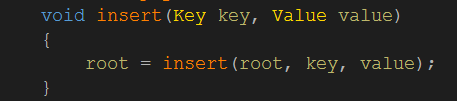
每个节点键值小于右孩子以左右孩子为根的子树仍为二分搜索树

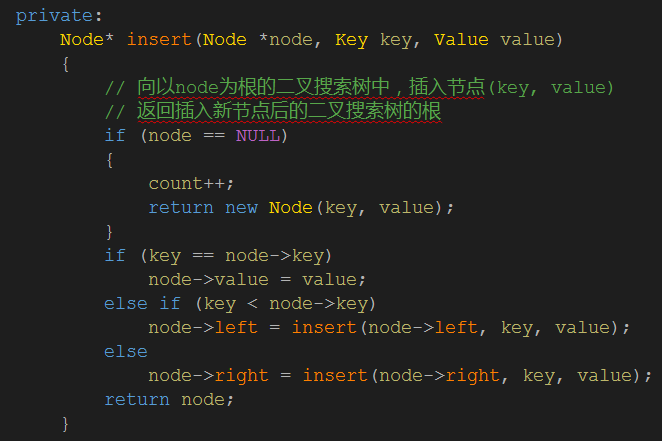


二叉搜索树节点

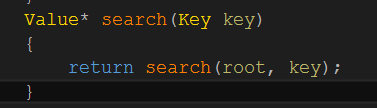


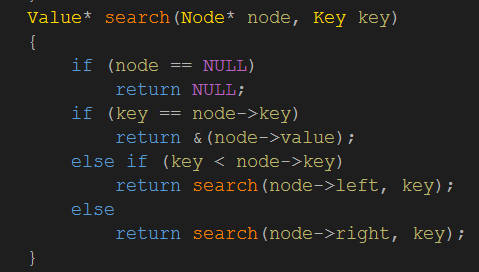
二分搜索树插入操作





二分搜索树查找操作





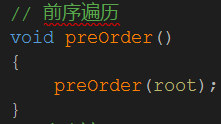
二分搜索树的深度优先遍历

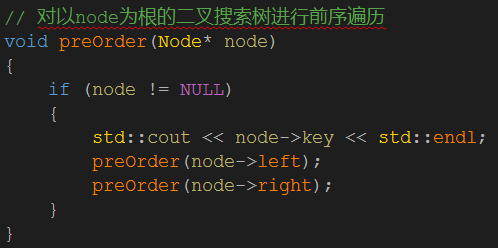
前序遍历：先访问当前节点，再一次访问左右子树

中序遍历：先递归访问左子树，再访问自身，再递归访问右子树（排序输出）

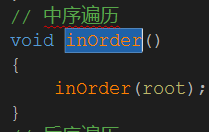
后序遍历：先递归访问左右子树，再访问自身节点（析构整个二叉搜索树）

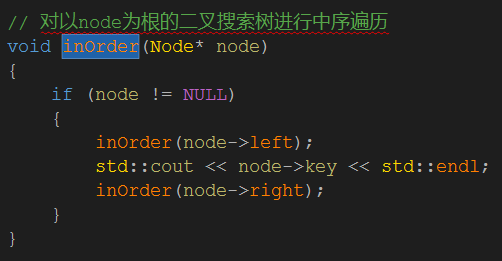
前序遍历



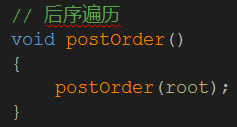


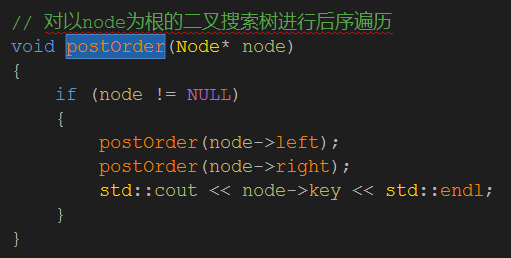
中序遍历



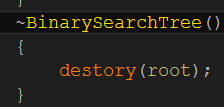


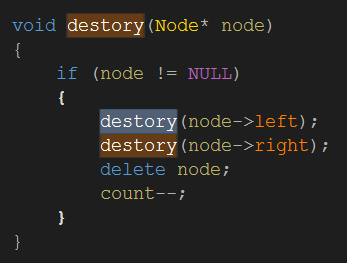
后序遍历





析构



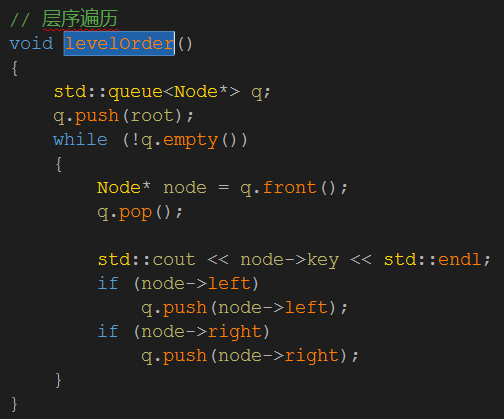


二分搜索树的广度（层次）优先遍历

利用队列

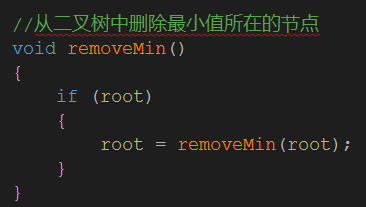
基本思想：将节点入队，如果队列不为空，则出队（打印，遍历），将左右子孩子入队

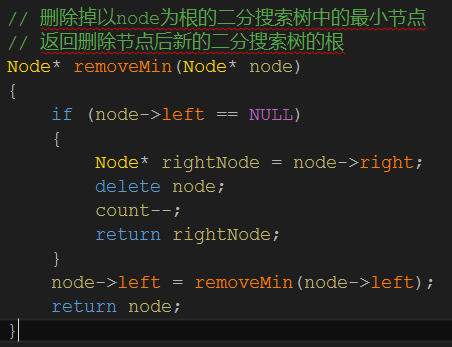
代码如下



二分搜索树 ------ 删除节点

删除最小值的节点





删除节点

Hubbard Deletion

第一种情况：无子节点

直接删除

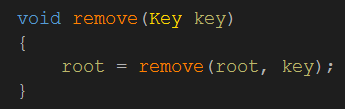
第二种情况：只有一个节点

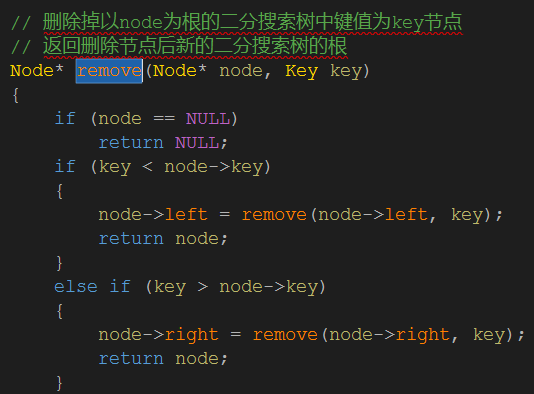
将其子节点替代本身的节点

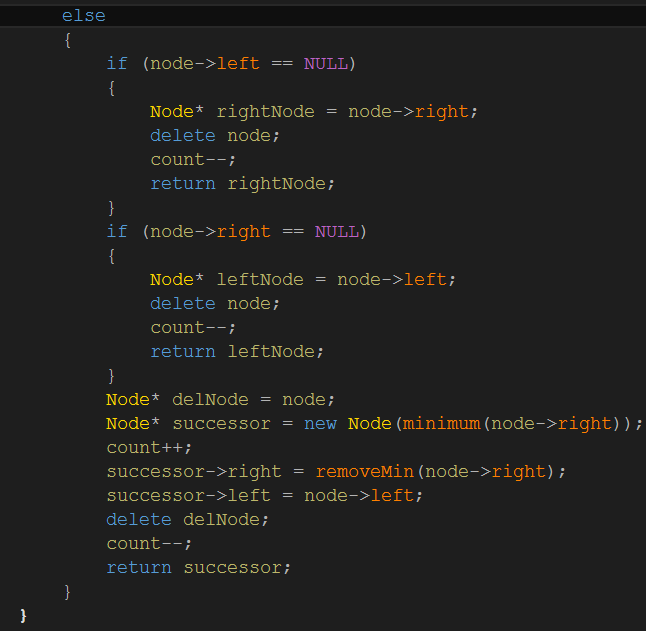
第三种情况：有两个节点，假设本节点为d

1. 找到节点s = min(d->right)，即找到后继节点（右子树中的最小值）
2. S->right = removeMin(d->right)
3. S->left = d->left;
4. Delete d，s是新的子树的根

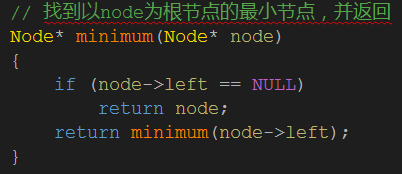
代码如下







其中minimum函数实现如下



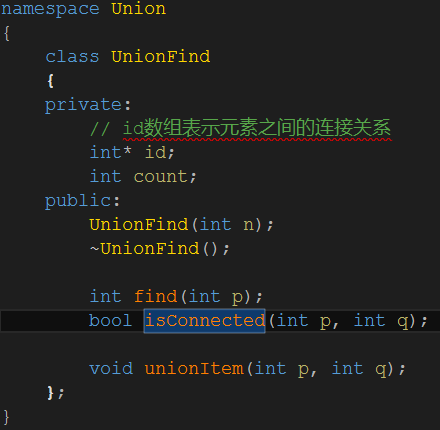
二分搜索树在极端情况下会退化成链表，而优化的数据结构是平衡二分搜索树，如红黑树，它保证左右子树的高度差不超过1.

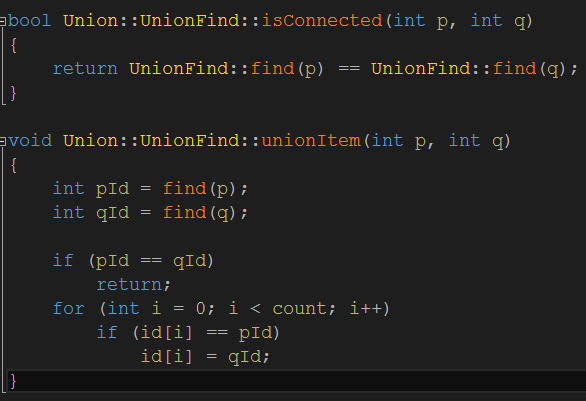
并查集Union Find

一种很不一样的树形结构

作用：解决连接问题Connectivity Problem，数学中的集合类问题

利用数组





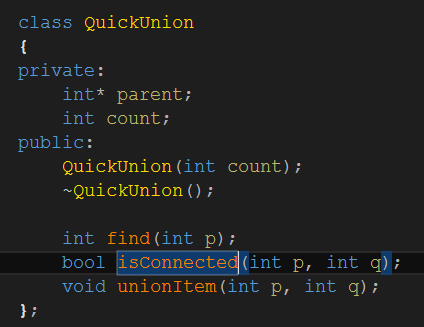
这种并查集的实现，并操作的时间复杂度是O(n)，查操作时间复杂度是O(1)，操作10w的数据时，测试结果如下：

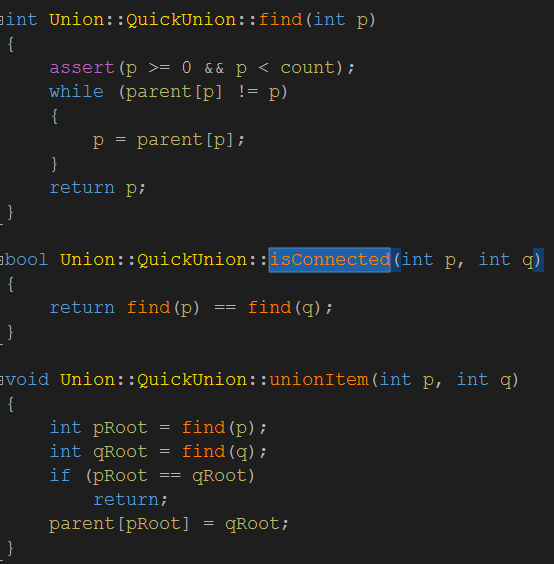


另一种并查集的实现

将每个元素看做是一个节点

同样用数组parent表示，parentp[i]表示，i节点的父亲节点是哪个。初始化时父亲元素都是自己。





与之前的比较1w的数据测试如下



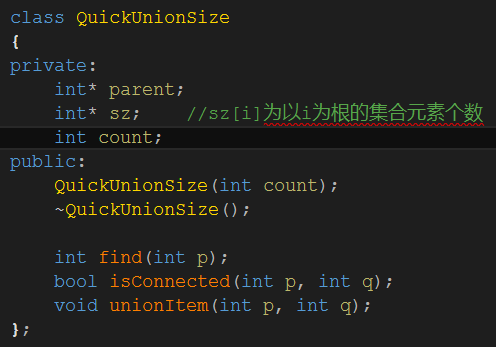
与之前的比较10w的数据测试如下

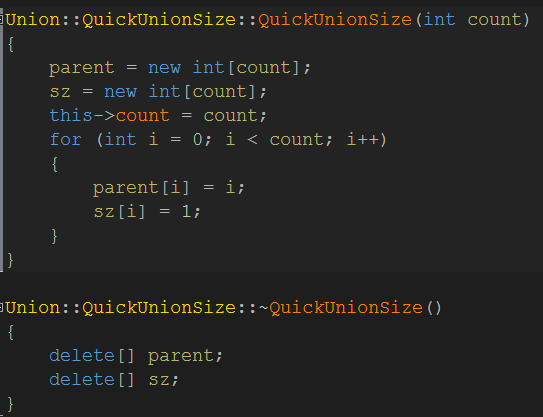


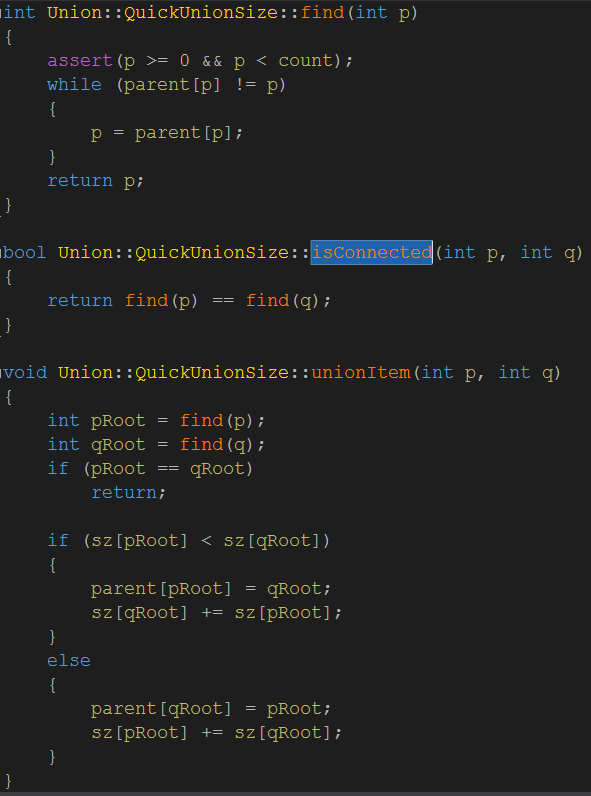
当数据量大了之后，find操作的量级就会很大，因为一直要找到它的根位置，极端的情况下是要寻找n次的，而UnionFind只需要一次读取操作，导致了QuickUnion甚至还不如UnionFind。因此需要优化

问题的关键在于，连接操作parent[pRoot] = qRoot，在进行并操作的时候，不能完全定死，需要根据一些特征来连接，使得并查集形成一个比较平衡的树。

第一个优化方案是基于size，每次连接操作都会记录根节点的集合元素个数，每次连接把size小的连接到size大的根节点。







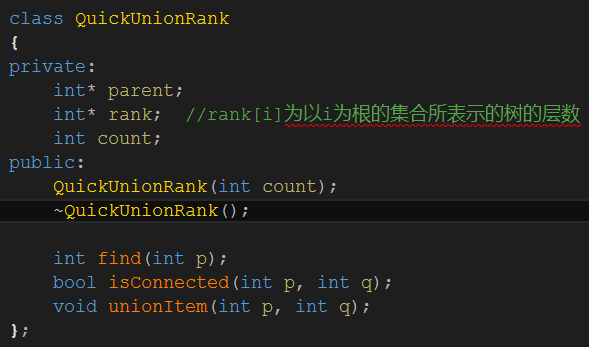
基于size优化之后的并查集与前面的2个比较结果如下



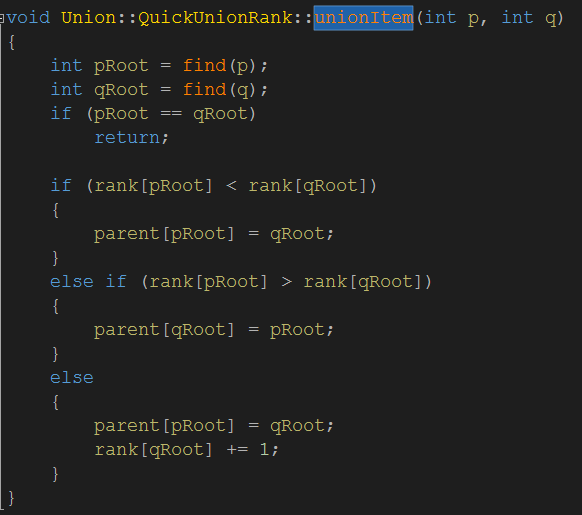
效果还是很明显的。

但是还有一些情况导致基于size的优化，效率变低。因此要基于rank的优化，rank[i]表示根节点第i个元素的高度。

代码如下



其他操作与基于size的没有什么变化，主要是union操作。



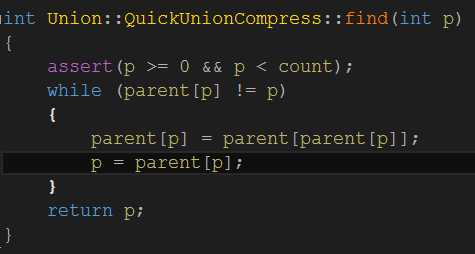
结果如下，10w的数据2个算法比较几乎没有变化，这是因为，比较难出现层数很高的情况，但是基于rank的并查集应用比size多。



基于路径压缩的并查集

在find过程中进行路径压缩，下次调用find函数时，就会快很多。

修改的地方在find中，parent[p] = parent[parent[p]];



测试结果

100w的数据，效果还是挺明显的。



图论Graph Theory

研究的是节点和边构成的一种数学模型

节点(Vertex)

边(Edge)

很多应用，如交通运输，社交网络，互联网，工作安排，脑区活动，程序状态执行（自动状态机）等

图的分类

无向图(Undirected Graph)

有向图(Directed Graph)

无权图(Unweighted Graph)

有权图(Weighted Graph)

图的连通性

简单图：没有自环边和平行边。（两个节点之间只有一条边，没有自己连向自己的边）

图的表示

邻接矩阵：A，Aij表示节点i和j是否相连。适合表示稠密图

邻接表：适合表示稀疏图

邻接矩阵的实现

如下

遍历邻边

遍历邻边是图算法中最常见的操作

相邻邻边迭代器：

可以隐藏私有数据，并且不用复制数据

图的遍历

1. 深度优先遍历

从一个点开始，不停往下试，直到试不下去。需要记录每个点是否被遍历过。

典型的应用：计算连通分量、寻路。

复杂度：

邻接表 : O(V+E) = O(V)

邻接矩阵 : O(V^2)

1. 广度优先遍历

从一个点开始，遍历相邻节点。再对这些相邻节点，遍历其相邻节点。

与广度优先遍历树类似，借助队列。

复杂度：

邻接表 : O(V+E) = O(V)

邻接矩阵 : O(V^2)

