树结构在程序设计中的运用

复旦大学附中　　周文超

树结构在程序设计中的运用

- 引言
- 一．并查集
- 二．线段树
- 三．树状数组
- 小结

引言

- 近年来，由于各种竞赛纷纷采用 free-pascal ，因此对于算法来说，空间效率上的要求降低了，而对时间效率却提出了更高的要求。这使得选手不仅要娴熟地掌握常规算法，而且要大胆创新，构造更高效的算法来解决问题。

- 在以往的程序设计中，链式结构采用得较多。的确链式结构有编程复杂度低、简单易懂等优点，但有一个致命的弱点：相邻的两个元素间的联系并不明显。而树结构却能很好的做到这一点。

返回

并查集

- 竞赛中会经常遇到这样的题目：给出各个元素之间的联系，要求将这些元素分成几个集合，每个集合中的元素直接或间接有联系。在这类问题中主要涉及的是对集合的合并和查找，因此将这种集合称为并查集。

- 链结构的并查集

- 树结构的并查集

链结构的并查集

- 链表被普通用来计算并查集：表中的每个元素设两个指针：一个指向同一集合中的下一个元素；另一个指向表首元素。采用链式存储结构，在进行集合的查找时的算法复杂度仅为O（1）；但合并集合时的算法复杂度却达到了O（n）。如果我们希望两种基本操作的时间效率都比较高的话，链式存储方式就“力不从心”了。

返回

树结构的并查集

- 采用树结构支持并查集的计算能够满足我们的要求。并查集与一般的树结构不同，每个顶点纪录的不是它的子结点，而是将它的父结点记录下来。下面我介绍一下树结构的并查集的两种运算方式

  - (1) 直接在树中查询
  - (2) 边查询边“路径压缩”

对应与前面的链式存储结构，树状结构的优势非常明显：编程复杂度低；时间效率高。

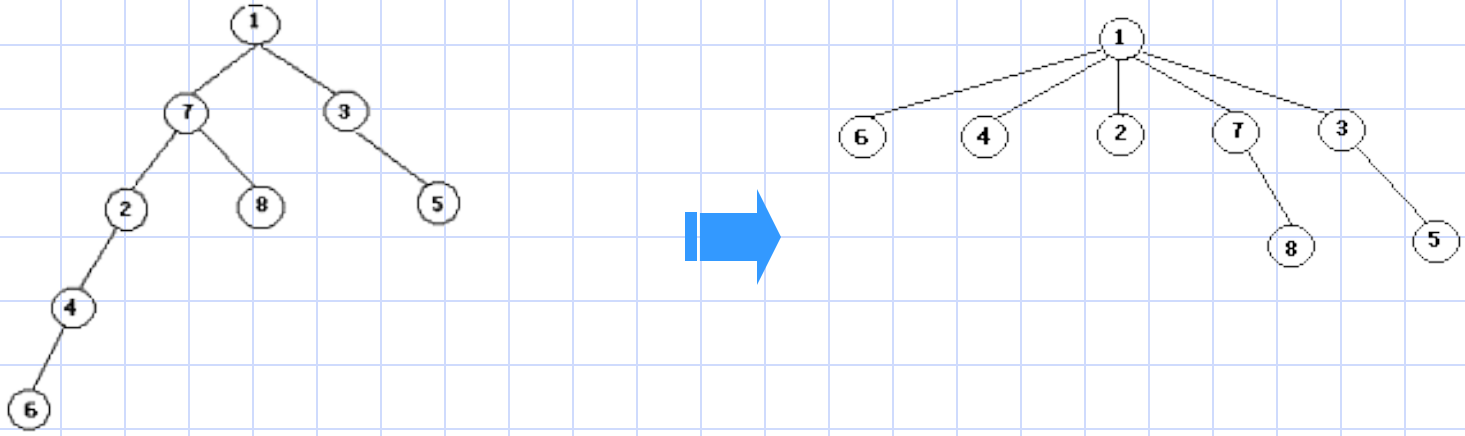
返回

直接在树中查询

集合的合并算法很简单，只要将两棵树的根结点相连即可，这步操作只要O（1）时间复杂度。所以算法的时间效率取决于集合查找的快慢。而集合的查找效率与树的深度呈线性关系。因此直接查询所需要的时间复杂度平均为O（logN）。但在最坏情况下，树退化成为一条链，使得每一次查询的算法复杂度为O（N）。

边查询边“路径压缩”

其实，我们还能将集合查找的算法复杂度进一步降低：采用“路径压缩”算法。它的想法很简单：在集合的查找过程中顺便将树的深度降低。采用路径压缩后，每一次查询所用的时间复杂度为增长极为缓慢的 ackerman 函数的反函数—— α （ x ）。对于可以想象到的 n， α （ n ）都是在 5 之内的。



返回

线段树

- 处理涉及到图形的面积、周长等问题的时候，并不需要依赖很深的数学知识，但要提高处理此类问题的效率却又十分困难。这就需要从根本上改变算法的基础——数据结构。这里要说的就是一种特殊的数据结构——线段树。
- 先看一道较基础的题目：给出区间上的 n 条线段，判断这些线段覆盖到的区间大小。通过这题我们来逐步认识线段树。

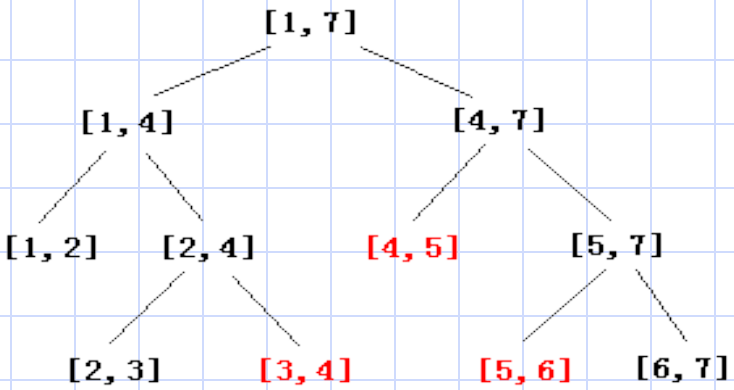
- 线段树的定义
- 在线段树中加入和删除线段
- 计算测度和连续线段数

返回

线段树的定义

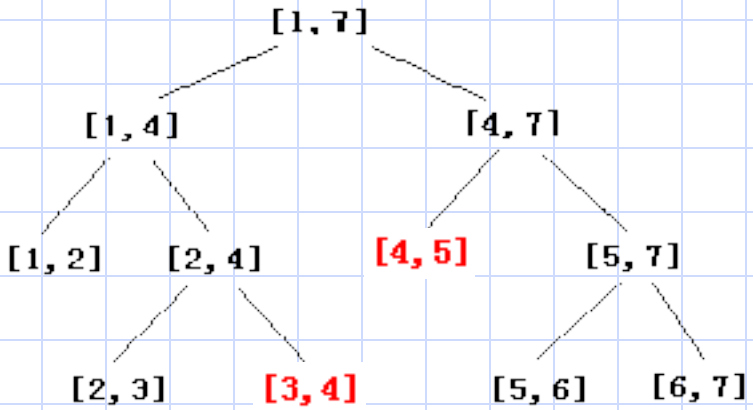
- 线段树是一棵二叉树，将一个区间划分为一个个 [i， i+1] 的单元区间，每个单元区间对应线段树中的一个叶子结点。每个结点用一个变量 count 来记录覆盖该结点的线段条数。

区间 [1， 7] 所对应的线段树如下图所示。区间上有一条线段 [3， 6]。



在线段树中插入和删除线段

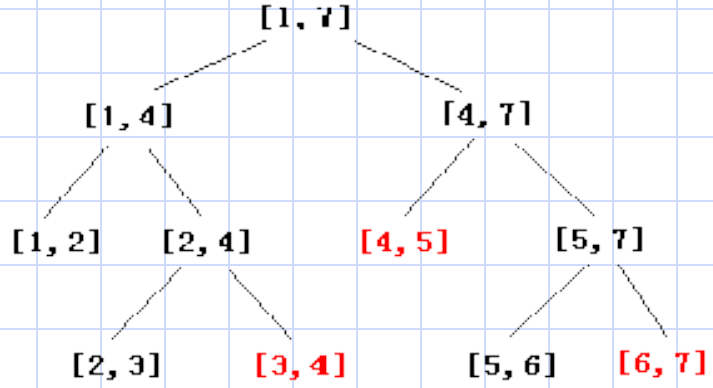
- 在线段树中插入和删除线段的方法类似，都采用递归逐层向两个子结点扫描，直到线段能够盖满结点表示的整个区间为止。



经过分析可以发现：在线段树中插入、删除线段的时间复杂度均为O（logN）。

计算测度和连续线段数

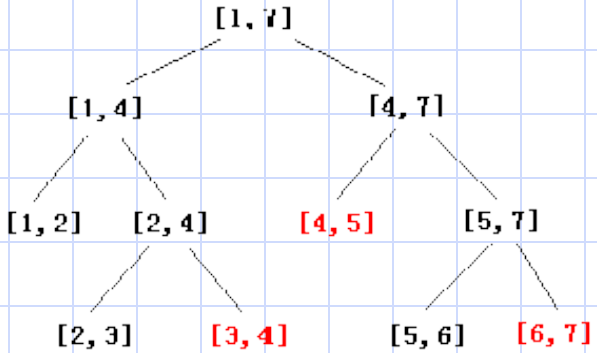
- 结点的测度 m 指的是结点所表示区间中线段覆盖过的长度。



$$m=\begin{cases} j - i & (count>0) \\ 0 & (count=0 \text{ 且结点为叶结点}) \\ lch.m + rch.m & (count=0 \text{ 且结点为内部结点}) \end{cases}$$

计算测度和连续线段数

- 连续线段数 line 指的是区间中互不相交的线段条数。

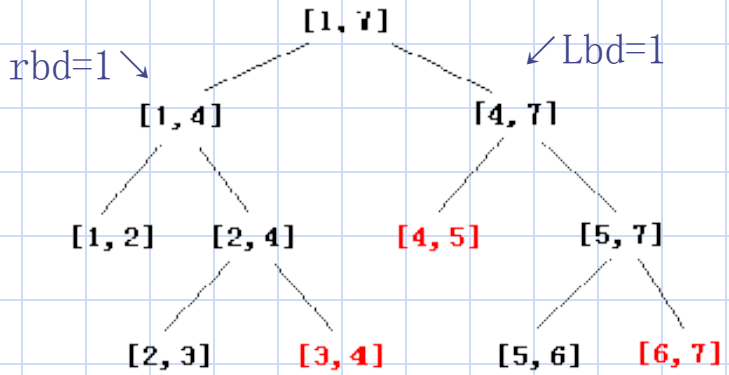


连续线段数并不能像测度那样将两个子结点中的连续线段数简单相加。于是我们引进了两个量 lbd， rbd，分别表示区间的左右两端是否被线段覆盖。

$$lbd = \begin{cases} 1 & \text{（左端点被线段覆盖到）} \\ 0 & \text{（左端点不被线段覆盖到）} \end{cases}$$

$$rbd = \begin{cases} 1 & \text{（右端点被线段覆盖到）} \\ 0 & \text{（右端点不被线段覆盖到）} \end{cases}$$

计算测度和连续线段数



line 可以根据 lbd， rbd 定义如下：

$$
Line=\begin{cases}
1 & (count > 0) \\
0 & (count=0 \text{ 且结点为叶结点}) \\
lch^{\wedge}.Line + rch^{\wedge}.Line - 1 & (count=0 \text{ 且结点为内部结点}\ lch^{\wedge}.rbd、rch^{\wedge}.lbd \text{ 都为 1}) \\
lch^{\wedge}.Line + rch^{\wedge}.Line & (count=0 \text{ 且结点为内部结点}\ lch^{\wedge}.rbd, rch^{\wedge}.lbd \text{ 不同时为 1})
\end{cases}
$$

返回

树状数组

- IOI2001 中的 MOBILE 难倒了很多选手。虽然该题的题意十分简单：在一个矩阵中，通过更新元素值修改矩阵状态，并计算某子矩阵的数字和，但难点在于数据的规模极大。下面，我来介绍一种新的数据结构—树状数组。

- 1、建立树状数组C
- 2、更新元素值
- 3、子序列求和

利用树状数组，编程的复杂度提高了，但程序的时间效率也大幅地提高。这正是利用了树结构能够减少搜索范围，将信息集中起来的优点，让更新数组和求和运算牵连尽量少的变量。

返回

建立树状数组 C

- 先将问题简化，考察一维子序列求和的算法。设该序列为a[1]，a[2]……a[n]

算法 1：直接在原序列中计算。显然更新元素值的时间复杂度为 O(1)；在最坏情况下，子序列求和的时间复杂度为 O（n）。

算法 2：增加数组 b，其中 b[i] ＝ a[1]+a[2]+……+a[i]。由于 a[i] 的更改影响 b[i]---b[n]，因此在最坏情况下更新元素值的算法复杂度为 O(n)；而子序列求和的算法复杂度仅为 O(1)。

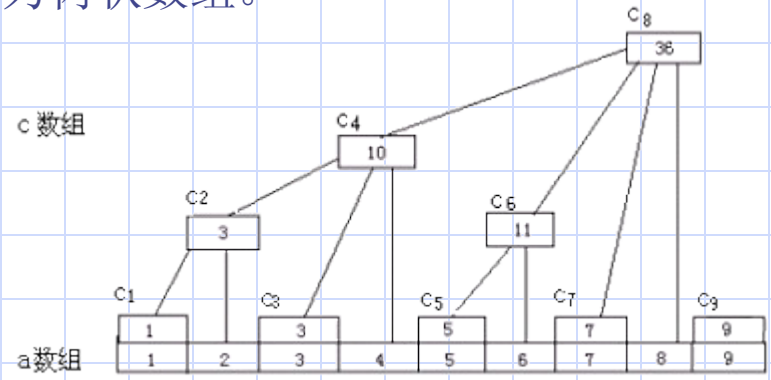
以上两种算法，要么在更新元素值上耗费时间过长（算法 1），要么在子序列求和上无法避免大量运算（算法 2）。有没有更好的方法呢？

算法三：增加数组 C，其中 C[i]=a[i-2^k+1]+……+a[i](k 为 i 在二进制形式下末尾 0 的个数）。由c 数组的定义可以得出：

```
c[1]=a[1]
c[2]=a[1]+a[2]=c[1]+a[2]
c[3]=a[3]
c[4]=a[1]+a[2]+a[3]+a[4]=c[2]+c[3]+a[4]
c[5]=a[5]
c[6]=a[5]+a[6]=c[5]+a[6]
……
```

c 数组的结构对应一棵树，因此称之为树状数组。

在统计更新元素值和子序列求和的算法复杂度后，会发现两种操作的时间复杂度均为 O（logN），大大提高了算法效率。



返回

更新元素值

- 定理

若 a[k] 所牵动的序列为 C[p_1]， C[p_2]……C[p_m]。
则 p_1=k，而 p_{i+1}=p_i+2^{1i}（1_i 为 p_i 在二进制中末尾 0 的个数）。

由此得出更改元素值的方法：若将 x 添加到 a[k]，则 c 数组中 c[p_1]、c[p_2]、…、c[p_m]（$p_m \leqslant n < p_{m+1}$）受其影响，亦应该添加 x。

例如 a[1]……a[9] 中， a[3] 添加 x；

p_1=k=3　　p_2=3+2^0=4

p3=4+2^2=8　　p_4=8+2^3=16>9

由此得出， c[3]、c[4]、c[8] 亦应该添加 x。

引理

若a[k]所牵动的序列为C[p_1]，C[p_2] …… C[p_m]，且$p_1 < p_2 < \cdots\cdots < p_m$，则有$l_1 < l_2 < \cdots\cdots < l_m$（$l_i$为$p_i$在二进制中末尾0的个数）。

证明：

若存在某个i有$l_i \geqslant l_{i+1}$，则$p_i - 2^{li} + 1 \leqslant k \leqslant p_i$，$p_{i+1} - 2^{li+1} + 1 \leqslant k \leqslant p_{i+1}$，$p_{i+1} - 2^{Li+1} + 1 \leqslant k \leqslant p_i$，即

$$P_{i+1} \leqslant P_i + 2^{Li+1} - 1 \qquad (1)$$

而由$L_i = L_{i+1}$、$P_i < P_{i+1}$可得

$$P_{i+1} \geqslant P_i + 2^{Li} \qquad (2)$$

(1) (2)矛盾，可知$l_1 < l_2 < \cdots\cdots < l_m$

定理

$p_1 = k$，而$p_{i+1} = p_i + 2^{li}$

证明：

因为$p_1 < p_2 < \cdots\cdots < p_m$且C[$p_1$]，C[$p_2$] …… C[$p_m$]中包含a[k]，因此$p_1 = k$。在p序列中，$p_{i+1} = p_i + 2^{li}$是$p_i$后最小的一个满足$l_{i+1} > 1i$的数（若出现$P_i + x$比$p_{i+1}$更小则$x < 2^{li}$，与x在二进制中的位数小于$l_i$相矛盾）。$P_{i+1} = p_i + 2^{li}$，$l_{i+1} \geqslant l_i + 1$。

由$p_i - 2^{li} + 1 \leqslant K \leqslant P_i$可知，$P_{i+1} - 2^{li+1} + 1 \leqslant P_i + 2^{li} - 2*2^{li} + 1 = P_i - 2^{li} + 1 \leqslant K \leqslant P_i \leqslant P_{i+1}$，故$P_i$与$p_{i+1}$之间的递推关系式为

$$P_{i+1} = P_i + 2^{li}$$

子序列求和

- 子序列求和可以转化为求由 a[1] 开始的序列 a[1]……a[k] 的和 S 。
- 而在树状数组中求 S 十分简单：

根据 $c[k]=a[k-2^{l}+1]+ \cdots +a[k]$ （l 为 k 在二进制数中末尾 0 的个数）

我们从 $k_1=k$ 出发，按照

$k_{i+1}=k_i-2^{lki}$ （ lki 为 k_i 在二进制数中末尾 0 的个数） 公式一

递推 k_2 ， k_3 ， …， k_m （$k_{m+1}=0$）。由此得出

$S=c[k_1]+c[k_2]+c[k_3]+ \cdots +c[k_m]$

例如，计算 a[1]+a[2]+a[3]+a[4]+a[5]+a[6]+a[7]

$k_1=7$

$k_2= k_1-2^{l1}=7-2^{0}=6$

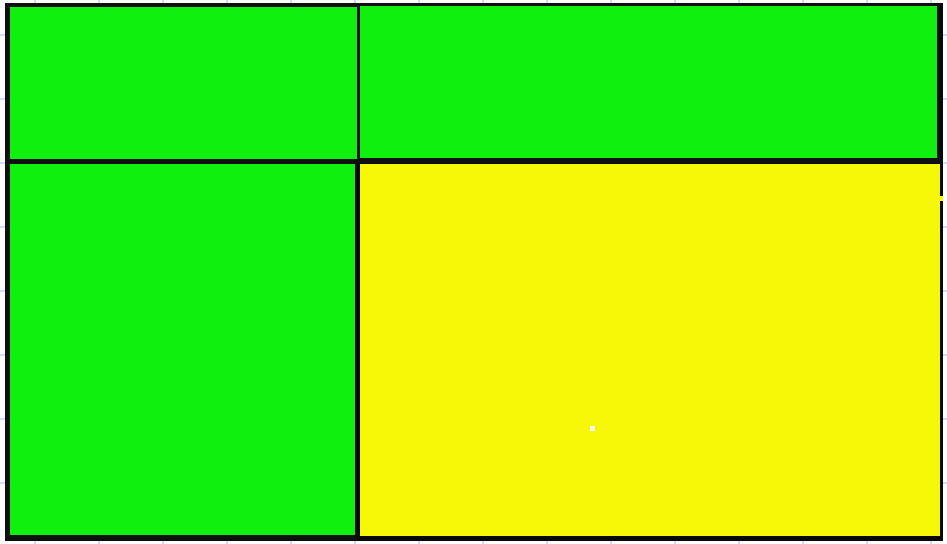
$k_3= k_2-2^{l2}=6-2^{1}=4$

$k_4= k_3-2^{l3}=4-2^{2}=0$

即 a[1]+a[2]+ a[3]+a[4]+ a[5]+a[6]+ a[7]=c[7]+c[6]+c[4]

子矩阵求和

推广到二维，同样也只需要计算由（1，1）开始的矩阵中的数字和，然后通过加减来计算子矩阵中的数字和。



返回

小结

在上面的例子中我们可以看出采用树结构处理问题时，往往能够缩小搜索范围。这样在每次的查找过程中都能正确地选择由根开始的一条路径，不会出现无用的搜索。

由于通常都是选取一条由根开始的路径，所以遍历树的快慢很大程度上取决于树的深度。如何降低树的高度就成了算法的关键。

相对于传统意义上的树，树状数组就其形式并不能称为“树”，但其核心思想却与树结构如出一辙。