《线段树》讲稿

安徽师范大学附属中学　杨弋

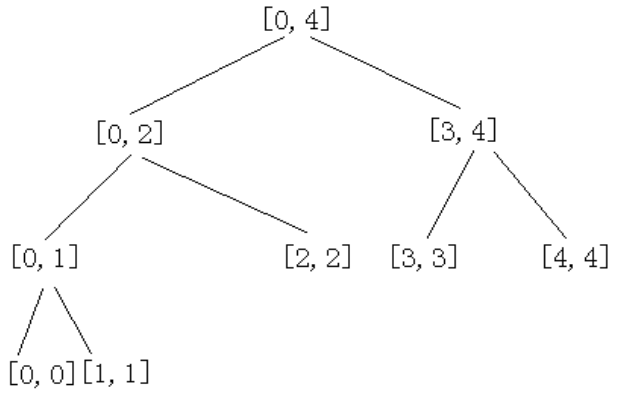
前言

在谈论到种种算法知识与数据结构的时候，线段树无疑总是与“简单”和“平常”联系起来的。而这些特征意味着，线段树作为一种常用的数据结构，有常用性，基础性和易用性等诸多特点。因此，今天我来讲一讲关于线段树的话题。

定义

首先，线段树是一棵“树”，而且是一棵完全二叉树。同时，“线段”两字反映出线段树的另一个特点：每个节点表示的是一个“线段”，或者说是一个区间。事实上，一棵线段树的根节点表示的是“整体”的区间，而它的左右子树也是一棵线段树，分别表示的是这个区间的左半边和右半边。

在此我们可以举一个例子来说明线段树通常的构造方法，以RMQ问题为例：
有N个数排成一排，每次询问某一段中的最小数。
构造的时候，让根节点表示区间[0,N-1]，即所有N个数所组成的一个区间，然后，把区间分成两半，分别由左右子树表示。不难证明，这样的线段树的节点数只有2N-1个，是O(N)级别的，如图：



对于每个节点，不但要知道它所表示的区间，以及它的儿子节点的情况，也记录一些别的值，不然，一棵孤零零的树能有什么用？在这个例子里，由于要查询的东西是最小值，不妨在每个节点内记录下它所表示区间中的最小值。这样，根据一个线性表构造出线段树的方法也就简单明白了：

```
function  构造以v为根的子树
    if v所表示的区间内只有一个元素
        v区间的最小值就是这个元素, 构造过程结束
```

```
    end if
    把v所属的区间一分为二，用w和x两个节点表示。
    标记v的左儿子是w，右儿子是x
    分别构造以w和以x为根的子树（递归）
    v区间的最小值 <- min(w区间的最小值,x区间的最小值)
end function
```

这样，一棵线段树就建立好了。不难证明构造过程是O(n)的，而线段树的高度是O(log n)的，准确地说，就是|log2 (n - 1)| + 1。

由构造过程可以发现，修改单个元素的操作异常简单：

```
function modify(v, i, newvalue) // 把i的值修改为newvalue，当前处理的子树根节点是v
    if v所表示的区间内只有一个元素 // 这个元素必定表示的就是[i, i]
        v区间的最小值 <- newvalue
        退出函数
    end if
    if (i属于v的左儿子的区间)
         modify(v的左儿子,i,newvalue)
    else
         modify(v的右儿子,i,newvalue)
    end if
    v区间的最小值 <- min(w区间的最小值,x区间的最小值) //更新数据
end function
```

由于每个节点都至多递归一次，它的时间复杂度 = O(树深度) = O(log n)。

区间查询操作

继续上面的例子，由于RMQ的目的是在区间内查询最小值，现在讨论如何利用刚刚构造出来的线段树高效回答这一提问：

比如刚才图中所示的树，如果询问区间是[0,2]，或者询问的区间是[3,3]，不难直接找到对应的节点回答这一问题。但并不是所有的提问都这么容易回答，比如[0,3]，就没有哪一个节点记录了这个区间的最小值。当然，解决方法也不难找到：把[0,2]和[3,3]两个区间（它们在整数意义上是相连的两个区间）的最小值“合并”起来，也就是求这两个最小值的最小值，就能求出[0,3]范围的最小值。同理，对于其他询问的区间，也都可以找到若干个相连的区间，合并后可以得到询问的区间。

幸运的是，很容易证明对于任何询问，这样的区间的个数不会超过O(log N)个。通常用来寻找这样一个区间的简单办法是，从根节点开始执行以下步骤：

```
function  在节点v查询区间[l,r]
    if v所表示的区间和[l,r]交集不为空集
        if v所表示的区间完全属于[l,r]
            选取v
        else
            在节点v的左右儿子分别查询区间[l,r]
        end if
    end if
```

end function

首先可见，这样的过程一定选出了尽量少的区间，它们相连后正好涵盖了整个[l,r]，没有重复也没有遗漏。同时，考虑到线段树上每层的节点最多会被选取2个，一共选取的节点数也是O(log n)的。类似地，同一层被访问的节点不会超过4个，因此查询的时间复杂度也是O(log n)。

换个问题，比如说我们的问题是：
对于一个长度为N的数字串S（从S[0]到S[N-1]），定义

```
        /   S[(N - 1) / 2]                                  如果N是奇数
f(S) = {
        \   f(S[0]..S[N / 2 - 1]) + f(S[N / 2]..S[N - 1])   如果N是偶数
```

给定数字串，要求能够回答它的某些子串的f值。

在N是2的整数次幂时，很容易按照题目条件构造出这棵线段树。对于修改某一个数字这样的操作，也是非常容易的。但是问题来了，真当我们想去用这棵线段树来回答题目中的提问，才会发现，我们选出了O(log n)个线段，它们相连，正好涵盖了我们所询问的区间。而且这些线段的f值都是已知的。可惜的是，这些信息对回答询问毫无帮助。

那么，什么情况下，线段树可以发挥它应有的功能，回答我们的区间查询呢？
当相邻的区间的信息，可以被合并成两个区间的并区间的信息时，就可以回答区间查询。
第一例中，两个相邻区间的最小值中的较小值，就是并区间的最小值。
第二例中，同等长度的数字串的f函数可以合并，但是不同长度就不可以。因此，虽然可以构造出这棵线段树，却不能回答我们所期望回答的区间询问。

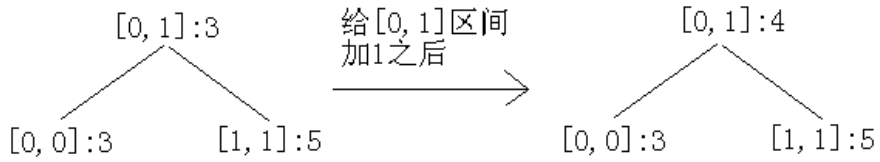
区间修改操作

仍然以RMQ问题为例，如果现在的修改操作一次可以修改的范围也是一个区间，比如给一个区间内所有数同时加上或减去某数，如果按照前面的定义，改动的节点数必定会远远超过O(log n)个。

原因是，我们这样的操作并没有很好利用修改操作也是对一个区间进行操作的性质。

既然要想把区间修改操作也控制在O(log n)的时间内，只修改O(log n)个节点的信息就成为必要。直观上，我们可以直接用前面给出的方法把操作区间变成O(log n)个相连的节点所表示的小区间。对每个区间进行一次修改操作，即可覆盖整个操作区间。正如查询操作如果查询的就是一个节点所表示的区间，就不用查找它的儿子节点的信息，区间修改时如果修改了一个节点所表示的区间，也不用去修改它的儿子节点。然而，对于被修改节点的祖先节点，也必须更新它所记录的值，否则查询操作就肯定会出问题（正如修改单个节点的情况一样）。

这些选出的节点的祖先节点直接更新值即可，而选出的节点却显然不能这么简单地处理：每个节点的值必须能由两个儿子节点的值得到，如这幅图中的例子：



这里，节点[0,1]的值应该是4，但是两个儿子的值又分别是3和5。如果查询[0,0]区间的RMQ，算出来的结果会是3，而正确答案显然是4。

问题显然在于，尽管修改了一个节点以后，不用修改它的儿子节点，但是它的儿子节点的信息事实上已经被改变了。这就需要我们在节点里增设一个域：**标记**。把对节点的修改情况储存在标记里面，这样，当我们自上而下地访问某节点时，就能把一路上所遇到的所有标记都考虑进去。

但是，在一个节点带上标记时，会给更新这个节点的值带来一些麻烦。继续上面的例子，如果我把位置0的数字从4改成了3，区间[0,0]的值应该变回3，但实际上，由于区间[0,1]有一个“添加了1”的标记，如果直接把值修改为3，则查询区间[0,0]的时候我们会得到3+1=4这个错误结果。但是，把这个3改成2，虽然正确，却并不直观，更不利于推广（参见下面的一个例子）。因此，我们引入“标记向下传”的概念：在访问一个节点的时候，“顺便”把它的标记传递给它的儿子节点。如本题中，则是这样实现：

```
v.lch.mark <- v.lch.mark + v.mark;
v.lch.value <- v.lch.value + v.mark;
v.rch.mark <- v.rch.mark + v.mark;
v.rch.value <- v.rch.value + v.mark;
v.mark = 0;
```

这样，就可以保证在访问一个节点的时候，不用真正去面对所谓的标记，而同时，仍然保持了各种操作的O(log n)的低复杂度。

下面再看一个例子：有一条直线，每次把其中一段染成某种颜色，要求输出最终的染色情况。

首先，可以把所有操作所涉及到的坐标离散化，从而可以用整数区间来表示直线上的线段。然后，由于现在的问题不涉及区间的查询，只涉及区间的修改，我们不需要考虑“如何合并两条线段的信息”这一问题，而只需考虑“如何在树上作标记以快速维护每个位置的颜色”。不难发现，这个题目中，每个节点不需要记录前面意义上的“值”，而只需要记录前面所说的“标记”。我们定义，一个节点上如果作了标记，就说明它所表示的区间里只有同一种颜色，而这种颜色是由标记所表示的；如果没有作上标记，就说明需要看它的儿子节点，才能确定它的颜色。

对于一开始完全没有染色的情况，直接把整个区间拆成O(log n)个被操作区间，然后分别做上标记即可。但是如果遇到这样的情况，如节点[0,3]已经被标记了，但是我们这次修改了[3,5]，那么[0,3]的标记就不再有效，但是如果直接删除的话，那么[0,2]这个范围内的颜色

又记错了。这就需要我们再次使用“标记向下传”的办法，这次的传的方法比较简单，直接把标记覆盖传给两个儿子节点，然后清空父亲节点的标记即可。回到刚才的第一句话，“对于一开始完全没有染色的情况”，事实上甚至我们都不需要完全的初始化，只需要把根节点做上一个“初始颜色”的标记，就行了。

在本题中加入随时询问某个位置的颜色，不需要对树进行任何更改即可完成。但是如果加入了随时询问某一段是否同色，标记就不能这样随便传了：我们需要在一个节点所代表的一段内是同色的时候，确保它作了标记。这样，在每次修改过程结束后，我们还需要再检查：是不是这个节点其实可以打上标记？伪代码如下所示：

```
function change(v, l, r, color) //  把[l,r]范围内修改成color

    if [l,r]和[v.l,v.r]没有交集
        退出
    end if

    if l = v.l AND r = v.r
        v.mark <- color                  // 直接作上标记
        退出
    end if

    if NOT v.mark =  未标记
        v.lch.mark <- v.mark
        v.rch.mark <- v.mark
        v.mark <-  未标记                 // 标记往下传
    end if

    change(v.lch,l,r,color)
    change(v.rch,l,r,color)

    if v.lch.mark = v.rch.mark AND NOT v.lch.mark =  未标记
        v.mark <- v.lch.mark          // 更新v的标记状态
    end if

end function
```

从这个例子可以看出来，“一个区间的颜色”这个性质，既符合“相邻区间的信息能合并”，又符合“标记可以向下传”。因此，没有绝对的“值”与“标记”的区分。所有的东西都是存在节点里的信息，通过不同的方式得到维护。前面的例子把两者分开，只是着重强调这两个性质分别的意义而已。

其他操作

有了以上的思想，可以解决很多常见的问题，如，“查询某区间内最靠右的不超过k的数”。为了回答这个询问，我们需要一棵存储RMQ的线段树。现在考虑，把询问的区间分成了O(log n)个节点所表示的区间之后，通过比较区间最小值与k的大小关系，立即可以判定某区间内是否有满足条件的数。在这些区间内选择一个最靠右的，然后如果它的右儿子存在满足条件的数，则在右儿子中查找，否则在左儿子中查找。时间复杂度当然是O(log n)。实际实现也不用像上面说的一样分为两步，一步就够了：

```
function find_rightmost (v, l, r, k)

    if [l,r]和[v.l,v.r]没有交集
        退出
    end if

    if (v.value > k)
        返回此区间内无解
    end if

    if (v是叶子)
        返回v
    end if

    if (find_rightmost(v.rch, l, r, k) 返回的不是无解)
        返回 find_rightmost(v.rch, l, r, k)
    else
        返回 find_rightmost(v.lch, l, r, k)
    end if

end function
```

某些时候，在一些题目中我们需要涉及到变化的“区间”：一开始数轴上有一些相连的线段，你每次可以查询某个点位于哪个线段，以及合并两个相邻线段，或者把某个线段一分为二。对于这样的问题，我们可以看成是线段的端点处都是1，别的地方都是0。则“查找点位于哪个线段”转化为“查找某点左侧区间的最右不小于1的数”以及“查找某点右侧区间的最左不小于1的数”；而线段合并和分离分别就是查找到某个位置，然后把1改成0，或0改成1。查找某个位置是第几条线段，可以统计某位置左侧区间的数字和。

可见，通过模型的转化，有相当多的问题都可以用线段树来解决。

实现

首先，如同几乎所有的树形数据结构一样，线段树可以使用指针表示。即如同前面的伪代码示例，使用指向儿子的指针来把节点串起来。这样的好处是，对于很大的区间以及很少的操作数量，可以建立一棵“虚”的树，逻辑上存在很多节点，但是实际上用不到的节点不予储存，直到访问的时候再去产生节点。这可以理解成存在一个“未建立节点”标记，而把“未建立节点”标记向下传的手段就是建立一个新节点。这样，就可以处理一些动态数据（因此

不能离散化）的问题，代价是，时空复杂度从O(n log n)上升到了O(n log L)，L是根节点所表示的区间大小。

此外，还存在一些不依赖指针，只用数组的存储方式。一种是堆式结构，即根节点占据地址0，然后节点x的左儿子为2x+1，右儿子为2x+2。这样，自上而下访问时可以顺便算出节点位置。

另一种是，对于区间[l,r]，用(l+r)来作为存储地址。但是缺点是，这样n必须取2的整数次幂。修正后的方法是，用(l+r) bitwise_or diff(l,r)作为地址。式中diff(l,r)返回1若l不等于r，否则返回0，bitwise_or是按位or。如果有兴趣，可以尝试证明一下，这个方法不重复也不遗漏地使用了[0,2N-2]的所有元素。

除了前面提到的自上而下的访问方式以外，对于某些题目，也可以不遍历树，直接在O(log n)时间内把一个操作区间划分为O(log n)个节点所表示的区间。

线段树也能扩展到二维形式，一种是“树套树”，即先处理一维，再处理第二维；另一种是“四叉树”。四叉树不能保证O(log^2 n)的单次操作时间；而前者看似结构麻烦，实际上往往只需把前面介绍的数组式实现扩展到二维（使用一个二维数组）即可。

例题

1. GSS2
yz对于分值相同的题目，只选择1题来做。于是给你一个题目序列，每个操作都是给你序列的一个子序列（本题中子序列的定义是连续的），让你选择这个子序列的一个子序列，让他取得尽量多的分数(题目有可能有负分数)。

2. AHOI2005 LANE
有一棵带权树，要求你回答两个点之间距离，同时随时可以修改边上的权值。

3. ZJOI2007 HIDE
有一棵树，树上的点有的是黑色，有的是白色。要求程序能回答两种操作：1. 输出树上的最远黑点对之间的距离。 2. 把一个点变色（黑变白，白变黑）。