浅谈线段树在信息学竞赛中的应用

岳云涛

yyt @ cominde

Wuhan University

【摘要】

本文介绍了一种高效的基于分治思想的数据结构——线段树，具体讲解了线段树的建树，查找，删除，统计等基本操作。并结合了一些例题深入研究了线段树的基本性质和线段树的应用方法。文章最后给出了一些线段树的练习题目。

【关键词】数据结构 二叉树 线段树

【目录】

【正文】

引言

在信息学竞赛中，我们经常会碰到一些跟区间有关的问题，比如给一些区间线段求并区间的长度，或者并区间的个数等等。这些问题的描述都非常简单，但是通常情况下数据范围会非常大，而朴素方法的时间复杂度过高，导致不能在规定时间内得到问题的解。这时，我们需要一种高效的数据结构来处理这样的问题，在本文中，我们介绍一种基于分治思想的数据结构－－线段树。

1 线段树

1.1 线段树的基本结构

线段树是一种二叉树形结构，属于平衡树的一种。它将线段区间组织成树形的结构，并用每个节点来表示一条线段。一棵[1,10)的线段树的结构如图 1.1 所示：

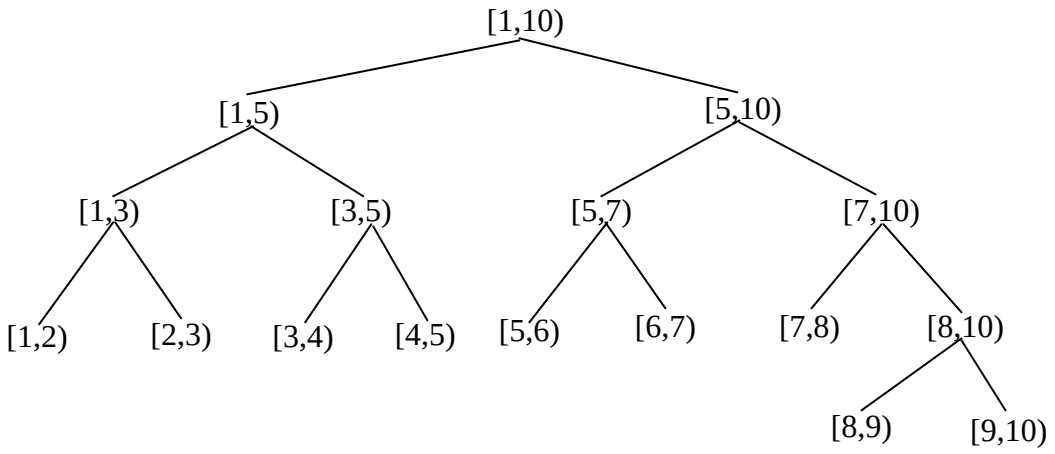

图 1.1 线段树的结构

可以看到，线段树的每个节点表示了一条前闭后开，即[a,b)的线段，这样表示的好处在于，有时候处理的区间并不是连续区间，可能是离散的点，如数组等。这样就可以用[a,a＋1)的节点来表示一个数组的元素，做到连续与离散的统一。

由图 1.1 可见，线段树的根节点表示了所要处理的最大线段区间，而叶节点则表示了形如[a,a+1)的单位区间。对于每个非叶节点（包括根节点）所表示的区间[s,t)，令 m = (s + t) / 2，则其左儿子节点表示区间[s,m)，右儿子节点表示区间[m,t)。

这样建树的好处在于，对于每条要处理的线段，可以类似“二分”的进入线段树中处理，使得时间复杂度在 O(logN)量级，这也是线段树之所以高效的原因。

1.2 线段树的性质

线段树具有如下一些性质：

- 线段树是一个平衡树，树的高度为 logN。
- 线段树把区间上的任意一条长度为 L 的线段都分成不超过 2log*L* 条线段的并
- 任两个结点要么是包含关系要么没有公共部分, 不可能部分重叠
- 给定一个叶子 p, 从根到 p 路径上所有结点代表的区间都包含点 p, 且其他结点代表的区间都不包含点 p

1.3 线段树的基本存储结构和操作

(1) 线段树的基本存储结构

线段树的一个节点的最基本的存储数据结构如下：

```
struct node{
    int left ,right, mid;
};
```

其中，left 和 right 分别代表该节点所表示线段的左右端点，即当前节点所表示的线段为[left, right).而 mid = (left + right) / 2，为当前线段的中点，储存这个点的好处是在每次在当前节点需要对线段分解的时候不需要再计算中点。这只是线段树节点的基本结构，在实际解决问题时，我们需要对应各种题目，在节点中添加各种数据域来保存有用的信息，比如添加 cover 域来保存当前节点所表示的线段是否被覆盖等等。根据题目来添加适当的数据域以及

对其进行维护是线段树解题的精髓所在，我们在平时的做题中需要注意积累用法，并进行推广扩展，做到举一反三。

(2) 线段树的基本操作

线段树的建立操作

在对线段树进行操作前，我们需要建立起线段树的结构。我们使用结构体数组来保存线段树，这样对于非叶节点，若它在数组中编号为 num，则其左右子节点的编号为 2 * num，2 * num + 1。由于线段树是二分的树型结构，我们可以用递归的方法，从根节点开始来建立一棵线段树。代码如下所示：

```
node seg_tree[3 * MAXN];
//由线段树的性质可知，建树所需要的空间大概是所需处理最长线段长度的 2 倍多，所以需要开 3 倍大小的数组
void make(int l, int r, int num){
//l,r 分别为当前节点的左右端点，num 为节点在数组中的编号
 seg_tree[num].left = l;
 seg_tree[num].right = r;
 seg_tree[num].mid = (l + r) / 2;
 if (l + 1 != r){
//若不为叶子节点，则递归的建立左右子树
 make(l, seg_tree[num].mid, 2 * num);
 make(seg_tree[num].mid, r, 2 * num + 1);
}
```

对应不同的题目，我们会在线段树节点中添加另外的数据域，并随着线段的插入或删除进行维护，要注意在建树过程中将这些数据域初始化。

线段树的插入操作

为了在插入线段后，我们能知道哪些节点上的线段被插入（覆盖）过。我们需要在节点中添加一个 cover 域，来记录当前节点所表示的线段是否被覆盖。这样，在建树过程中，我们需要把每个节点的 cover 域置 0；

如图 1.2 所示，在线段的插入过程中，我们从根节点开始插入，同样采取递归的方法。如果插入的线段完全覆盖了当前节点所代表的线段，则将当前节点的 cover 域置 1 并返回。否则，将线段递归进入当前节点的左右子节点进行插入。代码如下:

```
void insert(int l, int r, int num){
    //l,r 分别为插入当前节点线段的左右端点，num 为节点在数组中的编号
      if (seg_tree[num].left == l && seg_tree[num].right == r){
    //若插入的线段完全覆盖当前节点所表示的线段
         seg_tree[num].cover = 1;
          return;
      }
      if (r <= seg_tree[num].mid)
    //当前节点的左子节点所代表的线段包含插入的线段
           insert(l, r, 2 * num);
     else if (l >= seg_tree[num].mid)
    //当前节点的右子节点所代表的线段包含插入的线段
           insert(l, r, 2 * num +1);
     else {
    //插入的线段跨越了当前节点所代表线段的中点
         insert(l, seg_tree[num].mid, 2 * num);
         insert(seg_tree[num].mid, r, 2 * num + 1);
      }
}
```

要注意，这样插入线段时，有可能出现以下这种情况，即先插入线段[1,3)，再插入线段[1,5)，如图 1.3 所示。这样，代表线段[1,3)的节点以及代表线段[1,5)的节点的 cover 值均为 1，但是在统计时，遇到这种情况，我们可以只统计更靠近根节点的节点，因为这个节点所代表的线段包含了其子树上所有节点

所代表的线段。

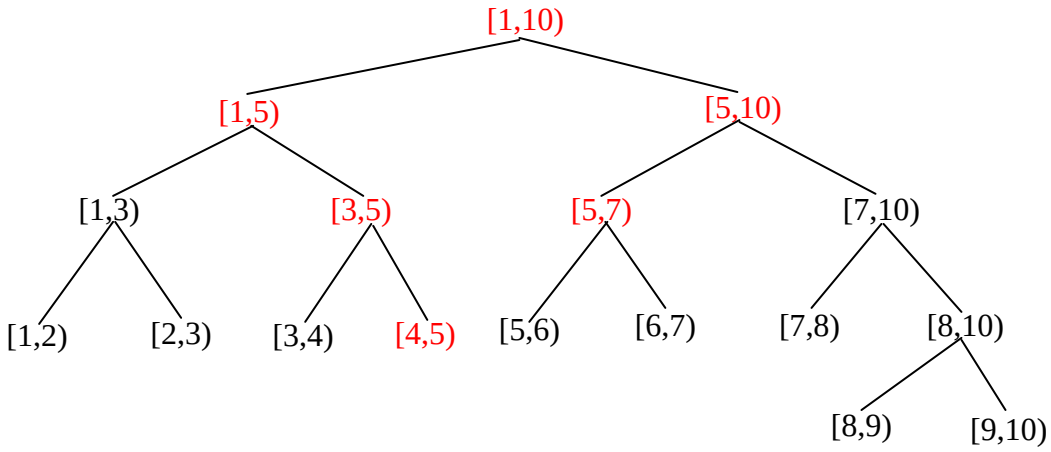

图 1.2 插入一条线段[4,7)

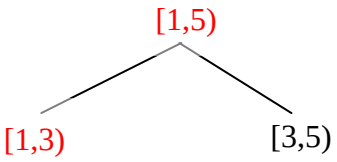

图 1.3 线段插入的特殊情况

线段树的删除操作

线段树的删除操作跟插入操作不大相同，因为一条线段只有被插入过才能被删除。比如插入一条线段[3,10)，则只能删除线段[4,6)，不能删除线段[7,12)。当删除未插入的线段时，操作返回 false 值。

我们一样采用递归的方法对线段进行删除，如果当前节点所代表的线段未被覆盖，即 cover 值为 0，则递归进入此节点的左右子节点进行删除。而如果当前节点所代表的线段已被覆盖，即 cover 值为 1，则要考虑两种情况。一是删除的线段完全覆盖当前节点所代表的线段，则将当前节点的 cover 值置 0.由于我们在插入线段的时候会出现图 1.3 所示的情况， 所以我们应该递归的在当前节点的子树上所有节点删除线段。另一种情况是删除的线段未完全覆盖当前节点所代表的线段，比如当前节点代表的线段为[1,10)，而要删除的线段为[4,7)，则删除后剩下线段[1,4)和[7,10)，我们采用的方法是，将当前节点的 cover 置 0，并将其左右子节点的 cover 置 1，然后递归的进入左右子节点进行删除。

删除操作的代码如下：

```
bool del(int l, int r, int num){
        if (seg_tree[num].left + 1 == seg_tree[num].right){
        //删除到叶节点的情况
            int f = seg_tree[num].f;
            seg_tree[num].f = 0;
            return f;
        }
        if (seg_tree[num].f == 1){
        //当前节点不为叶节点且被覆盖
            seg_tree[num].f = 0;
            seg_tree[2 * num].f = 1;
            seg_tree[2 * num + 1].f = 1;
         }
         if (r <= seg_tree[num].mid)
            return del(l, r, 2 * num);
        else if (l >= seg_tree[num].mid)
            return del(l, r, 2 * num + 1);
        else
            return del(l, seg_tree[num].mid, 2 * num) &&
                    del(seg_tree[num].mid, r, 2 * num + 1);
}
```

相对插入操作，删除操作比较复杂，需要考虑的情况很多，稍有不慎就会出错，在比赛中写删除操作时务必联系插入操作的实现过程，仔细思考，才能避免错误。

线段树的统计操作

对应不同的问题，线段树会统计不同的数据，比如线段覆盖的长度，线段

覆盖连续区间的个数等等。其实现思路不尽相同，我们以下以统计线段覆盖长度为例，简要介绍线段树统计信息的过程。文章之后的章节会讲解一些要用到线段树的题目，并会详细介绍线段树的用法，以及各种信息的统计过程。

对于统计线段覆盖长度的问题，可以采用以下的思路来统计信息，即从根节点开始搜索整棵线段树，如果当前节点所代表的线段已被覆盖，则将统计长度加上当前线段长度。否则，递归进入当前节点的左右子节点进行统计。实现代码如下：

```
int cal(int num){
        if (seg_tree[num].f)
            return seg_tree[num].right – seg_tree[num].left + 1;
        if (seg_tree[num].left + 1 == seg_tree[num].right)
       //当遍历到叶节点时返回
            return 0;
         return cal(2 * num) + cal(2 * num + 1);
}
```

小结：线段树作为一种数据结构只是解决问题的一个工具，具体的使用方法则非常灵活。以上介绍的仅仅为线段树的基础，在实际应用中，需要针对待解的题目设计节点存储信息，以及修改维护操作等等。下面将由浅及深的介绍线段树的一些应用，其中的一些线段树使用方法值得思考和借鉴。

2 线段树的初级应用

2.1 例题：City Horizon

链接：http://acm.whu.edu.cn/oak/problem/problem.jsp?problem_id=1224

题目大意：

如图 2.1 所示，在一条水平线上有 N 个建筑物，建筑物都是长方形的，且可以互相遮盖。给出每个建筑物的左右坐标值 Ai,Bi 以及每个建筑物的高度 Hi，需

要计算出这些建筑物总共覆盖的面积。

题目数据范围：

建筑物个数 N：1 <= N <= 40000

建筑物左右坐标值 Ai, Bi：1 <= Ai,Bi <= 10^9

建筑物的高度 Hi：1 <= Hi <= 10^9

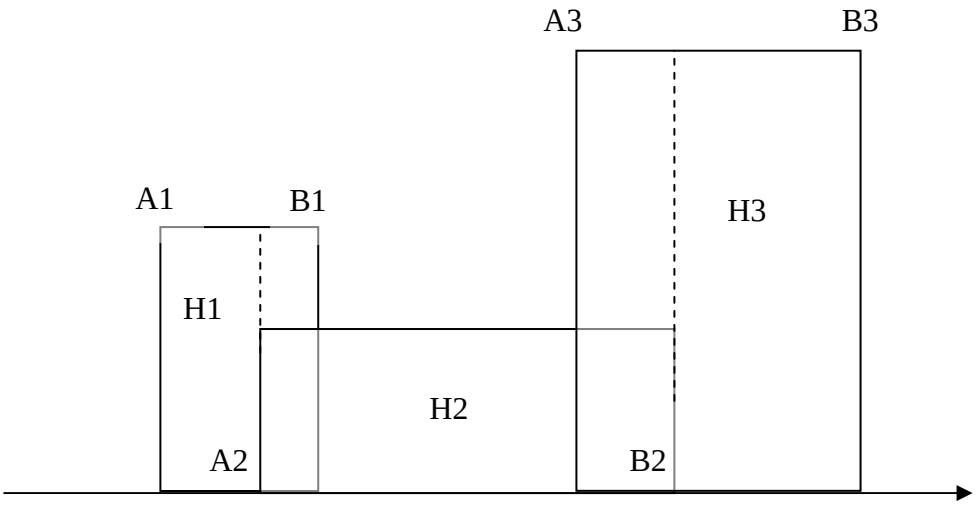

图 2.1 City Horizon

题目分析：

由题意可以知道，这道题需要求的即是这些矩形的面积并。考虑到题目中一个特殊的条件，所有的矩形的一边在一条直线上，我们可以好好利用这个条件：由于所有的矩形在这条直线上的投影均与矩形的一个边长相等。所以，我们可以把矩形“压缩”成直线上的线段，且每条线段都有一个权值，这个权值就是矩形的高度 Hi。那么，我们就可以利用线段树进行处理，计算面积并就相当于计算带权的线段并，即 S = H * (B – A)。当某条线段被多次覆盖时（比如图 2.1 中的线段 A2B1），只取 H 值最大的进行计算。如图 2.1 中的矩形面积并为：

S = H1*(B1 – A1) + H2 * (A3 – B2) + H3 * (B3 – A3)

基本思路清楚了，我们现在来考虑具体实现。由于题目中矩形的左右坐标的范围非常大(1 <= Ai,Bi <= 10^9)，如果建立大小为[1, 10^9)的线段树则会占用大量的空间。我们采用一种离散化的思想来处理这个问题，这种思路在线段树的题目中也是经常会用到的。考虑到一共只有 N <= 40000 个矩形，那么，这些矩形一共

也只有 2 * 40000 = 80000 个左右坐标值。我们首先将这 80000 个坐标值按大小排序，对排序后的坐标依次赋予一个新坐标值 k（1 <= k <= 80000），这样我们就把长度为[1, 10^9)的线段离散化成[1,80000)的线段了，而最后计算结果时，只需要按照新坐标值找回原始坐标值并代入计算即可。

举一个简单的例子，假设现在有三条线段[20,60),[10,50),[5,55)。我们将这三条线段的左右端点进行排序，其结果为 5,10,20,50,55,60。我们将它们依次赋上新值 1,2,3,4,5,6。这样原始的三条线段被离散化为[3,6),[2,4),[1,5)，我们就可以在[1,6)的空间内对其进行处理了。这就是离散化的威力。

回到原问题上来，当矩形所投影的线段被离散化以后，我们就可以建立线段树了。与之前讲过的初始化略有不同，现在每个线段树的节点不只是记录其所代表的线段是否被覆盖，而且要记录被覆盖的线段的权值。每次加入一个矩形就是在线段树上插入一条带权的线段，插入的实现过程与之前的也有不同。如果当前线段完全覆盖了节点所代表的线段，那么比较这两个线段的权值大小。如果节点所代表的线段的权值小或者在之前根本未被覆盖，则将其权值更新为当前线段的权值。实现代码如下：

```
void insert(int l, int r, int h, int num){
    if (seg_tree[num].left == l && seg_tree[num].right == r){
        if (seg_tree[num].h < h || !seg_tree[num].h)
            seg_tree[num].h = h;
        return;
    }
    if (r <= seg_tree[num].mid)
        insert(l, r, h, 2 * num);
    else if (l >= seg_tree[num].mid)
        insert(l, r, h, 2 * num + 1);
    else {
        insert(l, seg_tree[num].mid, h, 2 * num);
        insert(seg_tree[num].mid, r, h, 2 * num + 1);
    }
    return;
}
```

而最后计算面积并时，需要遍历整个线段树，因为这样才能确定每个从根节点到叶节点的路径上，即每个元线段上（形如[a,a+1)的线段），最大的高度是多少。统计过程从根向叶遍历，遍历过程中统计高度的最大值，并在叶节点上进行计算，非叶节点的计算结果是其左右子节点的计算结果之和。实现的代码如下(因为计算结果的数据超过了 int 的范围，所以使用 long long 数据类型):

```
long long cal(int h, int num){
    if (h > seg_tree[num].h)
        seg_tree[num].h = h;
    if (seg_tree[num].left + 1 == seg_tree[num].right){
        return (long long)seg_tree[num].h *
            (hash[seg_tree[num].right] – hash[seg_tree[num].left]);
    }
    return cal(seg_tree[num].h, 2 * num) + cal(seg_tree[num].h, 2 * num + 1);
}
```

小结：在本题的解决过程中，我们利用了题目的特殊条件，将矩形当成带权的线段来处理。同时使用了离散化的技巧，建立了线段树，并用线段树解决了统计问题。注意，在离散化的时候，如果有删除操作，那么值相同的点必须离散化成同一点，否则对线段操作时会出错。举个例子，线段[30,50),[20,30)必须离散化为[2,3),[1,2)。

2.2 例题 Joseph Problem

链接：http://acm.whu.edu.cn/oak/problem/problem.jsp?problem_id=1071

题目大意：

约瑟夫问题是一个非常经典的问题，它的问题描述是：有 n 个人围成一圈，从第 1 个人开始，每次按顺时针方向向后选择第 m 个人，并将这个人出列。问出列的顺序是怎样的。

题目数据范围：

1 <= n , m <= 30000

题目分析：

在数据结构课中，曾经讲过用循环链表模拟约瑟夫的出列过程，但是在本题中，n,m 的值都非常大，直接模拟的时间复杂度过高，为 O(nm)的，所以我们必须要另辟蹊径。首先，如图 2.2 所示，我们将圈拆开，变成从 1 到 n 排的 1 列，然后在这个列上进行处理。

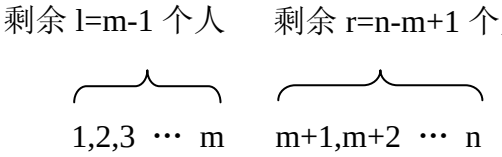

图 2.2 约瑟夫问题

图 2.2 表示的是第一次出列的情况，第一次出列的是第 m 个人，我们可以计算[1,m)和[m,n+1)这两个区间里剩余的人数 l 和 r。那么根据 m，我们可以计算出下一个出列的人，如果 m<r，他将在[m,n+1)中，否则，他将在[1,m)中。而且可以算出他是区间中从左向右数的第几个。

经过这样的分析，我们已经有了解决问题的大致思路。在具体实现中，我们仍然不能通过开数组模拟，那样的话时间复杂度没有降低。但是由于是在一个区间中处理问题，我们的线段树就可以发挥作用了。首先我们建立一个线段树，要注意在本题中，线段树的节点存储的不再是线段，而是一个包含离散点的区间，比如[1,4)表示的就是点 1，2，3。而线段树的叶节点[a,a+1)则表示了点 a。

建立线段树时，每个树中的节点需要保存一个新信息，即当前区间中，剩余点的个数。初始化的时候，每个节点[a,b)的剩余点个数为 b-a。要注意，根节点的区间是[1,n+1)而不是[1,n)。那么，每次出列一个人的时候，需要从叶节点到根节点维护这个信息。

在查找下一个人的时候，我们从根节点开始找，由于左右子节点区间中剩余的点数已经确定，我们可以确定要查找的点在哪个子节点中，且是子节点区间中的第几个人。通过这样递归的查找，直到找到叶节点为止，这样，叶节点所代表的点即是要出列的。

维护和查找的代码如下：

```
int update(int l, int r, int num){
    if (seg_tree[num].left == l && seg_tree[num].right == r){
        return seg_tree[num].count;
    }
    if (r <= seg_tree[num].mid){
        return update(l, r, 2 * num);
    }
    else if (l >= seg_tree[num].mid ){
        return update(l, r, 2 * num + 1);
    }
    else {
        return update(l, seg_tree[num].mid, 2 * num) +
               update(seg_tree[num].mid, r, 2 * num + 1);
    }
}
```

```
int search(int k, int num){
    if (seg_tree[num].left + 1 == seg_tree[num].right){
        seg_tree[num].count--;
        return seg_tree[num].left;
    }
    int t = seg_tree[2 * num].count;
    if (k <= t){
        seg_tree[num].count--;
        return search(k, 2 * num);
    }
    else {
        seg_tree[num].count--;
        return search(k - t, 2 * num + 1);
    }
}
```

小结：在本题中，我们采用了拆圈为线的方法，将问题转化到区间上，使得我们可以利用线段树进行求解。有了这种思想，对付其他变形的约瑟夫问题也可以如法炮制。在写程序时要注意区间线段树和一般线段树节点的含义的区别。

3 线段树的进阶应用

3.1 例题 Frequent Values

链接：http://acm.whu.edu.cn/oak/problem/problem.jsp?problem_id=1281

题目描述：

给 n 个数，已经按从大到小顺序排列好，一共有 q 个询问，每次询问一个区间，问这个区间中出现次数最多的数是什么。

题目数据范围：

数的个数，1 <= n <= 100000

询问次数，1 <= q <= 100000

每个数的大小，-100000 <= ai <= 100000

题目分析：

对于这种有大量区间内询问的问题，利用线段树一般来说可以比较好的解决。在本题中，我们很容易的想到建立线段树，并且在线段树的每个节点中保存其所代表区间内出现次数最多的数，以及这个数在区间中出现的次数。但是有两个问题需要解决，第一个是，建立线段树的时候，如何将两个子节点的信息合并到父节点。也就是说，知道了两个子结点各自出现次数最多的数，如何计算出父节点中出现次数最多的数。第二个是，在查询时，如果一条线段被拆成了两段，如何取值。

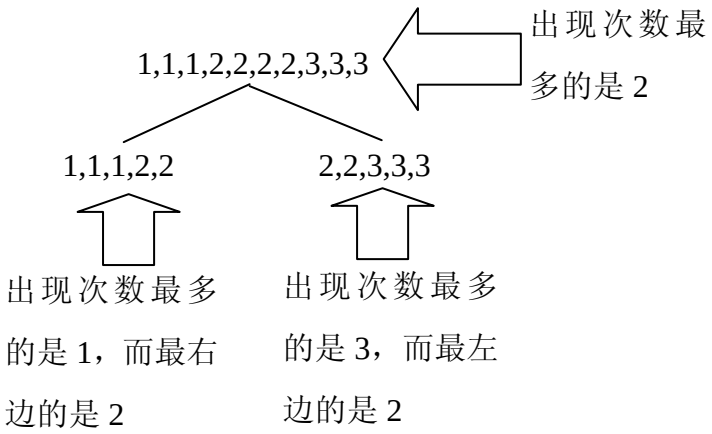

图 3.1 子结点合并成父节点

我们来考虑第一个问题，很显然，子结点中出现次数最多的数不一定就是父节点中出现次数最多的数，有可能一个数在两个子结点中的出现次数都不是最多，但是子结点合并成父节点后，这个数的出现次数就最多了。我们考虑到题目中一个非常重要的条件，即区间内的数是有序的。那么，父节点中出现最多的数，除了可能是子节点中出现次数最多的数外，还只有一种可能，即当左子节点区间最右的数与右子区间最左的数相等时，这个相等的数的出现次数最多（如图 3.1 所示）。那么我们在节点中还需要记录下几个信息，节点代表区间最左边和最右边的数，以及这两个数各自出现的次数。

那么在合并时，我们判断会不会出现图 3.1 中出现的情况，如果有，则计算中间的数出现的次数，并与左右子结点中出现次数最多的数比较，取次数最多的作为父节点中出现次数最多的数。若没有，则只需比较左右子结点中出现次数最多的数即可。在比较完后，非常重要的一步就是把父节点区间最左边的数和最右边的数以及它们的出现次数更新。

显然父节点区间最左边的数为左子节点区间最左边的数，最右边的数为右子结点区间最右边的数。但是它们的出现次数有可能会出现图 3.2 中的情况，所以在更新时必须检查子结点区间是否全部为相同的数，以及两个子结点区间相邻的两个数是否相同，若两个条件均满足，则需要在出现次数上加上这个数在另一个子节点中相应边上出现的次数。

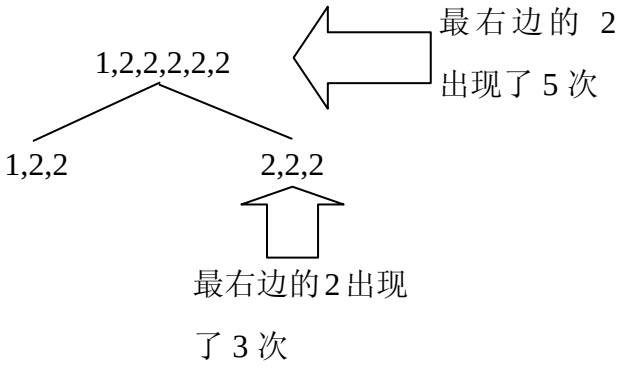

图 3.2 更新父节点

在线段树建立好以后，我们就可以开始查询了，查询的过程与建树的过程类似。当一个区间在线段树中被分为两个区间查询时，我们同样要考虑这两个区间

中各自出现次数最多的数，以及这两个区间相邻的数是否相等。从而得到所求的解。

建树的时间复杂度是 O(NlogN)的，而每次查询的时间复杂度是 O(logN)的。小结：在利用线段树解题时，一定要注意从子节点合并成父节点时，以及大区间分成两个小区间时，节点保存的信息应该如何更新。只有把细节想清楚，才能顺利的解决问题。

4 线段树的一些推广应用

4.1 多维线段树[1]

线段树处理的是线性统计问题，而我们往往会遇到一些平面统计问题和空间统计问题。因此我们需要推广一维线段树到多维中。比如将一维线段树改成二维线段树，有两种方法。一种是让原一维线段树中的每个节点保存一棵新的线段树，如图 4.1 所示。

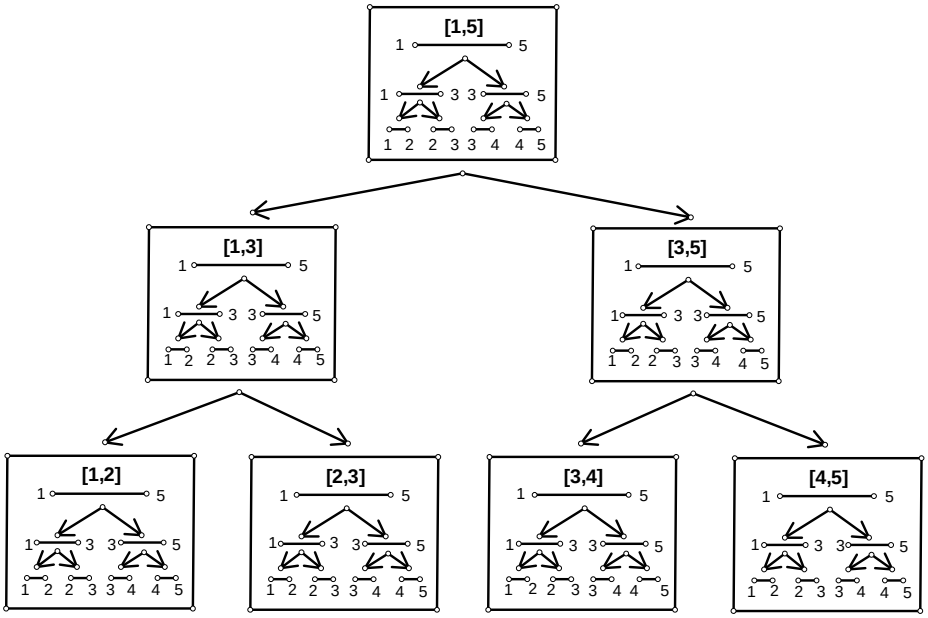

图 4.1 二维线段树[1]

例如在主线段树的结点[1,2]中，线段[3,4]表示的就是矩形(1,2,3,4),其中(1,2),(3,4)为矩形的左上端点和右下端点。

另一种方式是直接将原来线段树结点中的线段变成矩形。即每个结点代表

例如图 4.2 就是一棵以矩形(1,1,4,3)为根的矩形树：

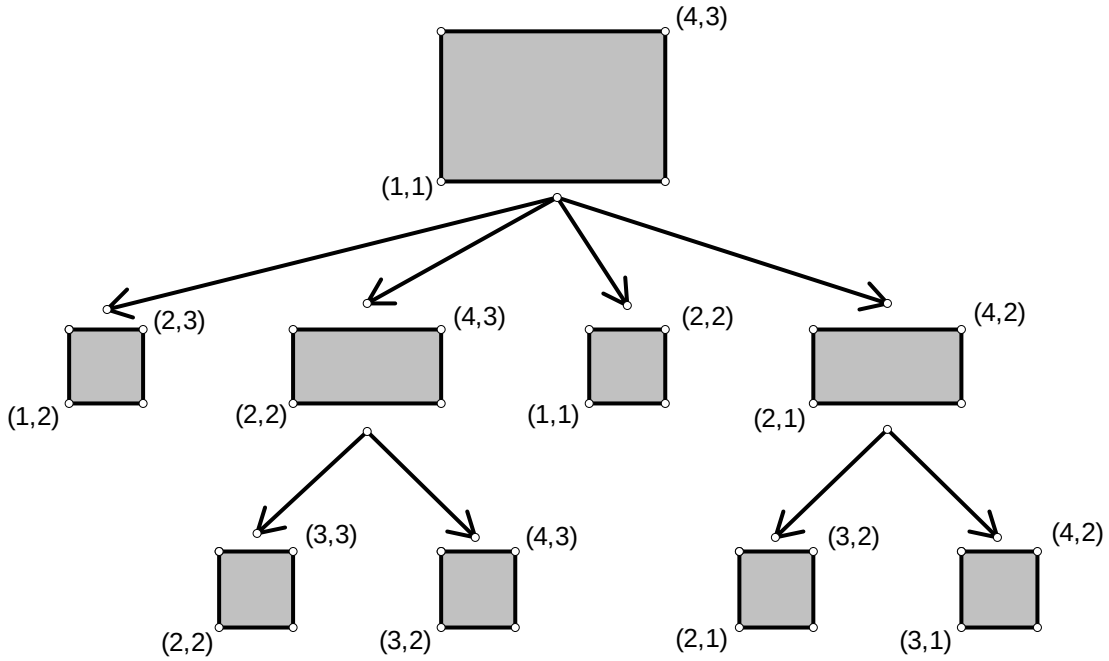

图 4.2 矩形树[1]

4.2 线段树与其他数据结构的组合

我们用一道例题来说明线段树与其他数据结构组合的情况。

例题： 百度之星 2008 初赛 第一天第三题 钉子与木板

题目描述：

墙上有 n 个钉子，编号为 1, 2, …, n。其中钉子 i 的横坐标为 i，纵坐标初始为 xi。可以进行两种操作：

0 k v：竖直移动钉子 k，坐标变为(k, v)。

1 s t v：若在高度为 v 处放一块横坐标范围是[s,t]的水平木板，它将下落到什么高度？换句话说，求出钉子 s, s+1, s+2, …, t 的纵坐标中，不超过 v 的最大值。如果这些钉子的高度全部大于 v，则木板将落到地上，高度为 0。

题目分析：

题目中的每个询问要求一个区间中，不超过 v 的最大值。由于每次询问的 v 值不同，我们无法在线段树的节点中记录不超过 v 的最大值。这时，我们可以在每个节点中另外使用一个新的数据结构——平衡树，平衡树可以在 O(logn)的时

间内查找一个具体数值，并计算树中比这个数值小的数的个数。具体思路如下，将每个节点的区间中的所有点建立平衡树，当有修改操作的时候，对根节点到叶节点路径上的所有节点中的平衡树进行修改。查询时，只需按一般方法递归查询即可。

这道题说明了线段树的节点中不仅可以存放单一的信息，还可以存放其他数据结构，做到线段树与其他数据结构的组合，使得功能更为强大，解题也更加灵活。

5 线段树应用总结

线段树是一种高效的数据结构。它的思想就是分治，非常基本也非常强大。在学习线段树的过程中，要时刻记住，线段树只是一种解题的工具，学习线段树只是学习一种解题的思维，就像图论中的 BFS 一样。至于在具体的题目中如何去运用这个工具，则需要开动脑筋，灵活的去建树并维护相关信息。这也需要在平时进行积累，做题时才能应用熟练。多做练习，勤思考，相信线段树能在你手中发挥无穷的威力！

6 线段树练习题目推荐

以下题目是各 OJ 上比较有名的线段树题目

Whu OnlineJudge

http://acm.whu.edu.cn/oak

题目号： 1071 1224 1361 1344

Pku OnlineJudge

http://acm.pku.edu.cn/JudgeOnline

题目号 3225 2482 1177 1029 2182 2750 2104 2528 2828 2777 2886 2761

Zju OnlineJudge

http://acm.zju.edu.cn

题目号 2301 1128 1659 2112

【参考资料】

[1] 《解决动态统计问题的两把利刃》 薛矛

[2]《线段树的应用》 林涛

[3] 《数据结构的选择与算法效率》 陈宏

[4] 《二分法与统计问题》 李睿