菜鸟都能理解的线段树入门经典

线段树的定义

首先，线段树既是线段也是树，并且是一棵二叉树，每个结点是一条线段,每条线段的左右儿子线段分别是该线段的左半和右半区间，递归定义之后就是一棵线段树，图示如下

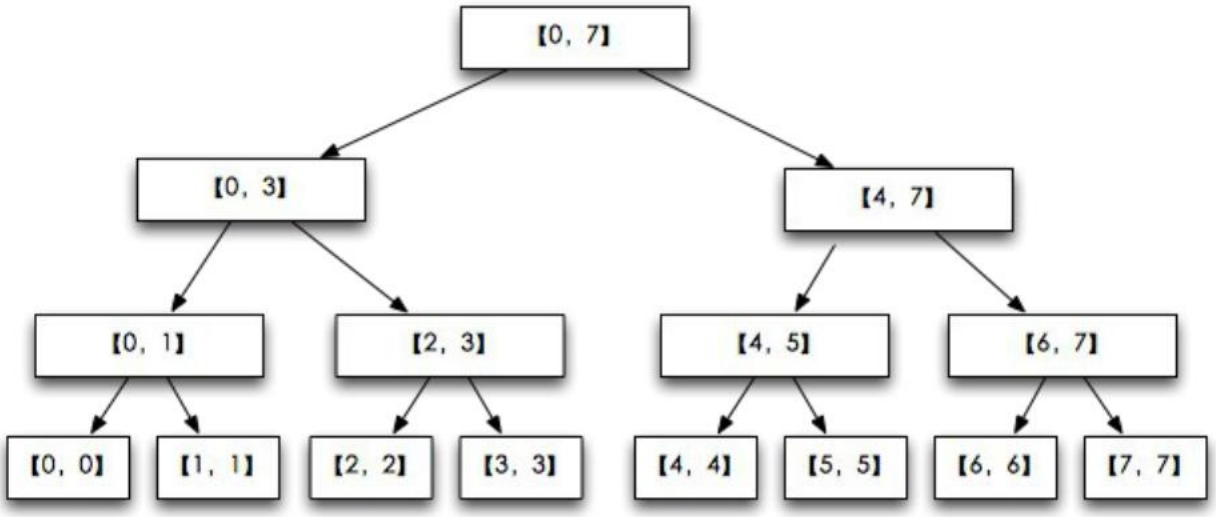


图 1.线段树示意图

定义线段树的数据结构

```
struct Line{
   int left, right, count;
   Line *leftChild, *rightChild;
   Line(int l, int r): left(l), right(r) {}
};
```

PS:其中的 count 字段表示该条线段有几条

明白了线段树的定义之后，我们来举一个例子来说明线段树的应用

例题：给定 N 条线段，{[2, 5], [4, 6], [0, 7]}, M 个点{2, 4, 7},判断每个点分别在几条线段出现过

看到题目，有的人第一感觉想出来的算法便是对于每一个点，逐个遍历每一条线段，很轻松地判断出来每个点在几条线段出现过，小学生都会的算法，时间复杂度为 O(M*N)

如国 N 非常大，比如 2^32-1, M 也非常大 M = 2^32 - 1, O(M*N)的算法将是无法忍受的，这个时候，线段树便隆重登场了

线段树的解法：

1.首先，我们找出一个最大的区间能够覆盖所有的线段，遍历所有的线段，找线段的最值(左端点的最小值，右端点的最大值)便可以确定这个区间，对于{[2, 5], [4, 6], [0, 7]}，这个区间为[0, 7]，时间复杂度为 O(N)

2.然后，根据此区间建一棵线段树(见图 1)，时间复杂度为 O(log(MAX-MIN))

3.对于每一条线段 A，从根节点开始遍历这棵线段树，对于每一个当前遍历的结点 NODE(其实线段树中每一个结点就是一条线段)，考虑三种情况

a)如果线段 A 包含在线段 NODE 的左半区间，那么从 NODE 的左儿子(其实就是 NODE 的左半区间)开始遍历这棵树

b)如果线段 A 包含在线段 NODE 的右半区间，那么从 NODE 的右儿子(其实就是 NODE 的右半区间)开始遍历这棵树

c)如果线段 A 刚好和线段 NODE 重合，停止遍历，并将 NODE 中的 count 字段加 1

d)除了以上的情况，就将线段 A 的左半部分在 NODE 的左儿子处遍历，将 A 的右半部分在 NODE 的右儿子处遍历

补充说明：对于以上的步骤，所做的工作其实就是不断地分割每一条线段，使得分割后的每一条小线段刚好能够落在线段树上，举个例子，比如要分割[2, 5],首先将[2, 5]和[0, 7]比较，符合情况 d，将 A 分成[2, 3]与[4, 5]

I)对于[2, 3]从[0, 7]的左半区间[0, 3]开始遍历

将[2, 3]与[0, 3]比较，满足情况 b,那么从[0, 3]的右半区间[2, 3]开始遍历,发现刚好重合，便将结点[2, 3]count 字段加 1

II)对于[4, 5]从[0, 7]的右半区间[4, 7]开始遍历

将[4, 5]与[4, 7]比较，满足情况 b，从[4, 7]的左半区间[4, 5]开始遍历，发现刚好重合，便将结点[4, 5]count 字段加 1

于是对于[2, 5]分割之后线段树的情况为图 2

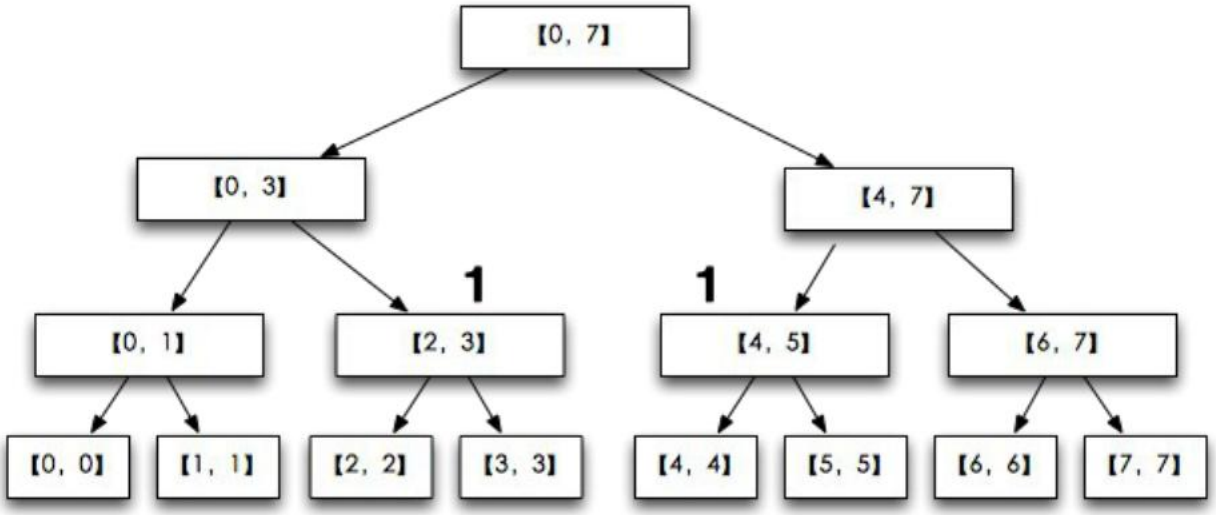
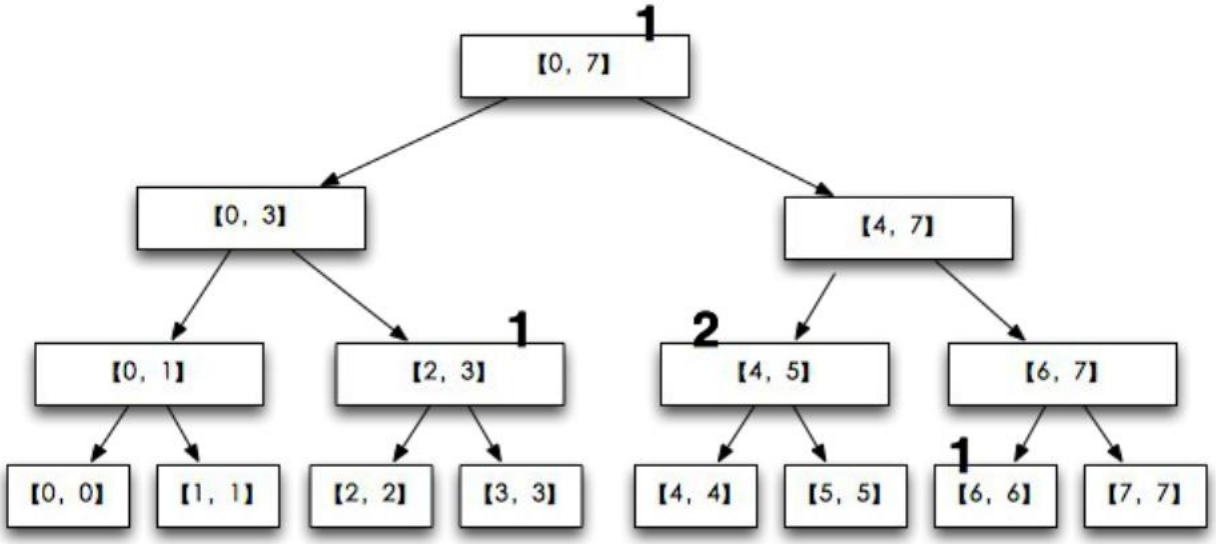


图 2.分割[2,5]之后线段树的情况

显然，我们看到，上述的遍历操作起始就是将[2, 5]按照线段树中的线段来分割，分割后的[2, 3]与[4, 5]其实是与[2, 5]完全等效的

最后，我们将剩下的两条线段按照同样的步骤进行分割之后，线段树的情况如下图 3



这一步的时间复杂度为 O(N*log(MAX-MIN))
4.最后，对于每一个值我们就可以开始遍历这一颗线段树，加上对于结点的 count 字段便是在线段中出现的次数
比如对于 4，首先遍历[0, 7]，次数 = 0+1=1；4 在右半区间，遍历[4, 7],次数 = 1+0=0;4 在[4, 7]左半区间，次数 = 1+2=3;4 在[4, 5]左半区间,次数 = 3+0 = 4，遍历结束，次数 = 3 说明 4 在三条线段中出现过,同理可求其他的值，这一步的时间复杂度为 O(M*log(MAX-MIN))
最后，总的时间复杂度为 O(N)+O(log(MAX-MIN))+O(N*log(MAX-MIN))+(M*log(MAX-MIN)) = O((M+N)*log(MAX-MIN))
由于 log(MAX-MIX)<=64 所以最后的时间复杂度为 O(M+N)

最后，放出源码
[cpp] view plaincopy

```
#include <iostream>
using namespace std;
struct Line{
   int left, right, count;
   Line *leftChild, *rightChild;
   Line(int l, int r): left(l), right(r) {}
};

//建立一棵空线段树
void createTree(Line *root) {
   int left = root->left;
   int right = root->right;
   if (left < right) {
      int mid = (left + right) / 2;
      Line *lc =    new Line(left, mid);
      Line *rc =    new Line(mid + 1, right);
      root->leftChild = lc;
      root->rightChild = rc;
      createTree(lc);
      createTree(rc);
   }
}

//将线段[l, r]分割
void insertLine(Line *root, int l, int r) {
   cout << l << " " << r << endl;
   cout << root->left << " " << root->right << endl << endl;
   if (l == root->left && r == root->right) {
      root->count += 1;
   } else if (l <= r) {
      int rmid = (root->left + root->right) / 2;
```

```
        if (r <= rmid) {
            insertLine(root->leftChild, l, r);
        } else if (l >= rmid + 1) {
            insertLine(root->rightChild, l, r);
        } else {
            int mid = (l + r) / 2;
            insertLine(root->leftChild, l, mid);
            insertLine(root->rightChild, mid + 1, r);
        }
    }
}
//树的中序遍历（测试用）
void inOrder(Line* root) {
    if (root != NULL) {
        inOrder(root->leftChild);
        printf("[%d, %d], %d\n", root->left, root->right, root->count);
        inOrder(root->rightChild);
    }
}

//获取值 n 在线段上出现的次数
int getCount(Line* root, int n) {
    int c = 0;
    if (root->left <= n&&n <= root->right)
        c += root->count;
    if (root->left == root->right)
        return c;
    int mid = (root->left + root->right) / 2;
    if (n <= mid)
        c += getCount(root->leftChild, n);
    else
        c += getCount(root->rightChild, n);
    return c;
}
int main() {
    int l[3] = {2, 4, 0};
    int r[3] = {5, 6, 7};
    int MIN = l[0];
    int MAX = r[0];
    for (int i = 1; i < 3; ++i) {
        if (MIN > l[i]) MIN = l[i];
        if (MAX < r[i]) MAX = r[i];
    }
    Line *root = new Line(MIN, MAX);
```

```
    createTree(root);
    for (int i = 0; i < 3; ++i) {
        insertLine(root, l[i], r[i]);
    }
    inOrder(root);
    int N;
    while (cin >> N) {
        cout << getCount(root, N) << endl;
    }
    return 0;
}
```