树状数组上机报告

刘浚哲

北京大学物理学院 *1500011370*

October 25, 2017

1. 树状数组简要介绍

平常我们会遇到一些对数组进行维护查询的操作. 比如修改某点的值、求某个区间的和等等. 当数据规模不大的时候, 对于修改某点的值是非常容易的, 复杂度是 $O(1)$. 但是对于求一个区间的和就要扫一遍了, 复杂度是 $O(N)$, 如果实时的对数组进行 M 次修改或求和, 最坏的情况下复杂度是 $O(M \cdot N)$. 当规模增大后这是划不来的, 而树状数组干同样的事复杂度却是 $O(M \cdot \lg N)$. 树状数组的示意图如下:

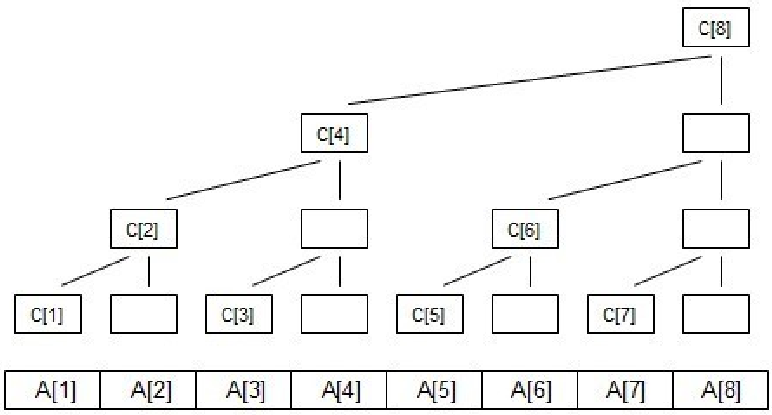

图 1: 树状数组示意

如图所示, 对于原数组 $A[i]$, 对应的树状数组 $C[i]$ 的值为:

$$C[i] = A[i - 2^k + 1] + A[i - 2^k + 2] + \cdots + A[i] \tag{1}$$

其中 k 为 i 的的二进制中从最低位到高位连续 0 的长度. 例如 $i = 8 = 01000$, 则 $k = 3$. 我们定义:2^k =lowbit(i).lowbit(i) 实际上就是取出 i 的最低位 1, 代码表示为:

```
int lowbit(int x) //树状数组 输出 x 的最低的为1的位
{
    return x&(-x);
}
```

其中用到了位运算中的" 按位与" 操作. 我们来证明它: 假设二进制表达为: $x = a1b$, 其中 b 全为 0 或者不存在, a 为任意前缀. 取反后, 其反码为按位取反并加 1, 有:

$$-x = (\sim a)0(\sim b) + 1 = (\sim a)1(b) \tag{2}$$

于是有:

$$x\&(-x) = (a)1(b)\&(\sim a)1(b) = (00\cdots0)1(00\cdots0) \tag{3}$$

上述推导利用了 b 全为 0 或者不存在的信息. 因此 $x\&(-x)$ 返回的是低位为 0 的长度, 也即 2^k.

从树状数组的定义可知, 它的值为一些原数组的区间之和, 因此利用树状数组可以很方便地利用二进制的性质求出数组的和. 例如 $i = 7 = 0111$, 有:

$$sum[7] = A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7] = C[7] + C[6] + C[4]$$

写成二进制有:

$$sum[0111] = C[0111] + C[0110] + C[0100]$$

序号每次都减少 lowbit(x), 于是我们可以写出求前 i 项和的算法:

```
int Sum(int i) //求前 i 个节点之和
{
    int ans=0;
    for (; i>0; i-=lowbit(i)) ans+=C[i];
    return ans;
}
```

当我们修改 $A[]$ 数组中的某一个值时, 我们同时需要更新树状数组 $C[]$. 但由于树状数组中的值是原数组中一个区间的和, 因此需要从某一节点开始向上回溯直到根节点, 修改途径的所有节点. 分析得知树状数组中下标为 i 的元素的父节点下标为 i+lowbit(i). 于是修改算法为:

```
void Add(int x, int addnum) //从下至上加入一个值 addnum, 父节点为 i+lowbit(i)
{
    for (; x<=n; x+=lowbit(x)) C[x]+=addnum;  //n 为数组元素个数
}
```

可以发现修改的过程是求和过程的逆过程. 由于其树形结构, 因此操作的时间复杂度为 $O(\lg n)$.

2. 题解:Apple Tree

题目描述:

给定一棵节点个数为 N 的苹果树. 树的根节点下标为 1. 每个节点用数字表示, 最多只能有一个苹果. 初始状态下每个节点都有一个苹果. 有两个操作, C x 表示如果 x 节点的权值为 1 则修正为 0, 否则修正为 1. Q x 表示询问以 x 为根节点的子树的权值.

题目分析:

首先构建原数组, 对于 N 个节点, 建立一个数组 apple[N] 用于保存各节点是否有苹果. 接下来, 建立树状数组, 建立一个 treearray[N] 用于对后续修改及求和操作进行快速处理.

接下来就是要考虑这个树的形态了. apple[N] 只能保存各个节点是否有苹果, 树状数组是用来计算区间之和的, 它们都不能反映节点之间的相连结构, 因此需要用一个结构来保存树形结构.

注意到由于题目只确定了根节点下标为 1, 并且对后续树枝的输入格式并未做进一步说明, 因此并不能确定是”先父后子”, 还是”先子后父”. 因此我们需要建立一个二维数组 tree[i][j], 行下标 i 为节点的下标 i, 对于一个横向量 tree[i][ ], 后续保存的是与 i 节点相连的节点下标. 若 <i,j> 相连, 则不光要在 tree[i][ ] 中加入 j, 还需要在 tree[j][ ] 加入 i, 具体父子关系则需要在遍历中进一步判断.

由于我们对一个相连关系进行了双向保存, 所以我们实际上创造的是一个无向图. 在遍历的时候, 我们需要判断当前节点是否已经遍历过, 如果已经访问过了则不再访问, 访问与 i 节点相连接的下一个节点. 于是我们需要添加一个记录访问结果的数组 visit[N] 用于保存当前节点是否已经被访问过. 初始每个节点都没有被访问, 全都设为 true, 之后在遍历时每访问一个节点就将它记为 false.

另外一个需要注意的是, 题目仅仅规定了根节点为 1, 其余点的标号全都没有进行规定. 也就是说: 除根节点外的所有节点可以是任意标号的, 第二个点的标号可以是 2, 也可以是 10, 或者是任意数. 而且也没有规定子节点的标号就会比父节点的标号大. 因此我们需要在建树之后, 将编号混乱的树形结构重新进行编号, 使之成为顺序编号的树. 这个过程就是 NFS 编号.

利用 NFS [1]编号, 将节点 i 所保存的左区间保存在数组 Left[i], 右区间保存在数组 Right[i] 中. 我们可以知道，对于某一节点i而言，我们是先通过这个点搜完所有他的后代编号才结束的，所以这个点的右值，包含了当前点所有的后代与祖先，后代必然是所有编号大于本节点的点，那么祖先呢，那必然是编号小于这个节点的点了. 所以我们通过 sum(Right[x])-sum(Left[x]-1) 就能得到 Q 查询的答案. 至于更新, 只需要利用 add 函数更新对应点即可.

数据结构与算法:

由上文分析, 数据结构如下:

```
#include <iostream>
#include <vector>
#include <string.h>
#define maxsize 100010
using namespace std;

vector<int>tree[maxsize]; //用一个二维数组 tree 保存节点之间的关系 tree[x]记录 x 节点的父节点及所有子节点
bool visit[maxsize],apple[maxsize]; //visit 数组记录这个节点是否被访问过. apple 数组用于记录该节点是否有苹果
int Left[maxsize], Right[maxsize]; //DFS 序.记录树中任意节点所代表的左区间和右区间 right-(left-1)为区间长度
int treearray[maxsize]; //树状数组 treearray.
int n=0; //用于 DFS 序遍历的整形变量
```

树状数组的三个操作已在上文给出.DFS 编号算法为:

```
void DFS(int x) //以 DFS 序遍历一个树(或者说是图), x节点的区间左界保存在 left[],右界保存在 right[]
{
    if (visit[x]==true) //如果该点没有访问过
    {
        Left[x]=++n; //区间左界
        visit[x]=false; //x节点点改为已访问
        for (int i=0; i<tree[x].size(); i++) DFS(tree[x][i]); //tree[x][i]保存所有与 x 相连的节点下标 i .要判
            断是否为 x 父节点
        Right[x]=n; //区间右界
    }
    else return;
}
```

[1]见文末附录

利用二维数组建树:

```
void buildtree(int N)
{
    for (int i=1; i<=N; i++)
    {
        if (i<N)
        {
            int x=0,y=0;
            cin>>x>>y;
            tree[x].push_back(y); //(x,y)连成一条线,因此 x 的数组后边要加入 y
            tree[y].push_back(x); //看成无向图, y 的数组后边要加入 x
        }
        visit[i]=true; //初始化 visit 数组,所有节点都未被访问
        apple[i]=true; //初始化 apple 数组,每个节点都有一个苹果
        treearray[i]=lowbit(i); //初始化树状数组,每个节点初始值应该是 lowbit
    }
}
```

操作函数:

```
void operate(int M)
{
    for (int i=1; i<=M; i++)
    {
        int x;
        char c;
        cin>>c>>x;
        if (c=='C') //摘/长苹果操作,有则摘,没有则长一个
        {
            if (apple[x]) Add(Left[x], -1); //有苹果就摘掉,并在树状数组 x 位置下减掉一个苹果,从下向上修改
            else Add(Left[x], 1); //没有苹果就长一个,然后在树状数组 x 位置下增加一个苹果,从下向上修改
            apple[x]=!apple[x]; //状态取反
        } //不是改x而是改Left[x].因为在DFS序中x不一定等于 Left[x]
        else cout<<Sum(Right[x])-Sum(Left[x]-1)<<'\n'; // Q查询
    }
}
```

这里需要注意的是, 进行修改操作时返回的下标应该是 Left[x], 而不是 x. 因为我们已经重新对数的节点下标进行了排列, 此时节点在 DFS 序下的下标保存在 Left[x]. 因此对于树状数组的操作下标应返回 Left[x].

主函数:

```
int main()
{
    int N,M;
    cin>>N;
    buildtree(N);
    DFS(1); //题目:1永远是树的根,所以从标号为1的节点向下遍历树
    cin>>M;
    operate(M);
    return 0;
}
```

运行结果:

```
10
1 10
10 2
5 10
8 5
7 10
6 2
9 3
3 7
7 4
5
Q 10
9
C 7
Q 1
9
Q 7
3
Q 1
9
Program ended with exit code: 0
```

图 2：Apple Tree 运行结果示意

经验与体会

本题的思路很巧妙, 树状数组用于最后对 apple 数组进行求和, 或者修改操作. 而真正要保存一个树, 我们需要新开一个二维数组, 并且利用无序图遍历的方法去读. 最后将树修改成 DFS 序, 就可以利用树状数组进行操作了.

附录：DFS 序

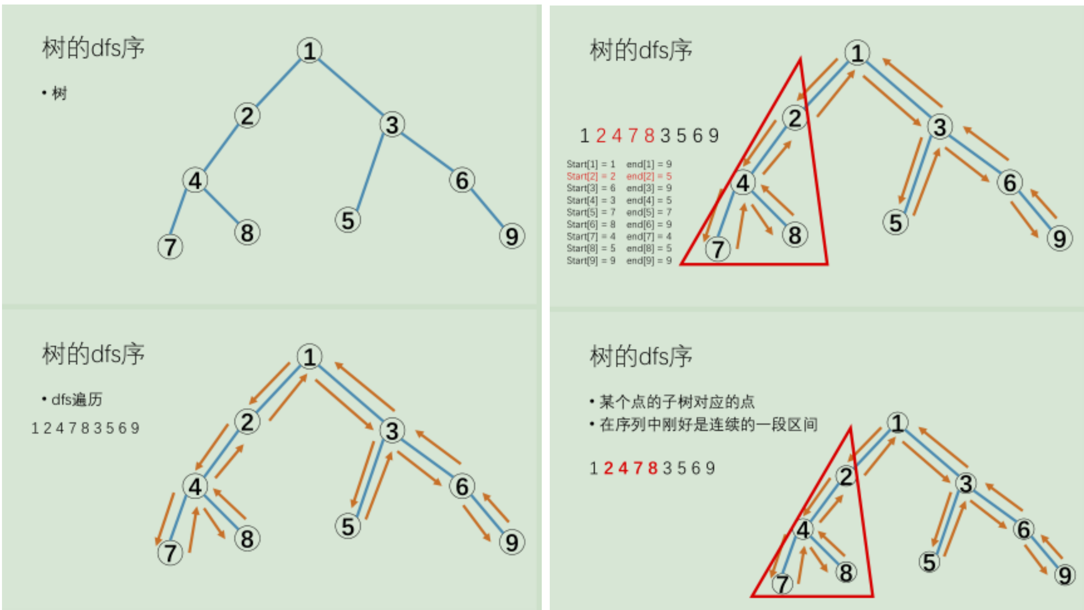

图 3: 树的 DFS 序示意图

树的 DFS 序就是用来维护一系列树上的问题的, 这类问题主要是解决一棵树上的所有子节点信息的更改和父节点有关, 主要先通过 DFS 来记录一个树的每一个顶点的出入时间戳 key, 来控制它子树上的所有结点的状态的更新. 只有向下搜索子节点时才有 key++, 而回溯到父节点时 key 是不会增加的, 因此每一次 key 增加都是找到了一个子节点, 最后 key 就等于树中的节点数. 用 Left[],Right[] 来记录这个父节点控制后代结点的区间.DFS 序的特点就是：一棵子树的 DFS 序就变成了一个区间.

DFS 序可以辅以各种数据结构（ST 表、树状数组、线段树）进行运算, 将子树求和转化为区间求和.