第 5 章　高级数据结构

☑ 并查集
☑ 二叉树
☑ Treap 树
☑ Splay 树
☑ 线段树
☑ 树状数组

竞赛题是对输入的数据进行运算，然后输出结果。因此，编写程序的一个基本问题就是数据处理，包括如何存储输入的数据、如何组织程序中的中间数据等。这个技术就是数据结构。学习数据结构，建立计算思维的基础，是成为合格程序员的基本功。本章讲解一些常用的高级数据结构。

数据结构的作用是分析数据、组织数据、存储数据。基本的数据类型有字符和数字，这些数据需要存储在空间中，然后程序按规则读取和处理它们。

数据结构和算法不同，它并不直接解决问题，但是数据结构是算法不可分割的一部分。首先，数据结构把杂乱无章的数据有序地组织起来，逻辑清晰，易于编程处理；其次，数据结构便于算法高效地访问和处理数据，大大减少空间和时间复杂度。

(1) 存储的空间效率。例如一个围棋程序，需要存储棋盘和棋盘上棋子的位置。棋盘可以简单地用一个 19×19 的二维数组(矩阵)表示。每个棋子是一个坐标，例如 $W[5][6]$ 表示位于第 5 行第 6 列的白棋。这种二维数组是数据结构“图”的一种描述，图是描述点和点之间连接关系的数据结构。棋盘只是一种简单的图，更复杂的图例如地图。地图上有两种元素，即点、点之间直连的道路。地图比棋盘复杂，棋盘的每个点只有上、下、左、右 4 个相邻的点，而地图上的一个点可能有很多相邻的点。那么如何存储一个地图？可以简单地用一个二维数组，例如有 n 个点，用一个 $n\times n$ 的二维矩阵表示地图，矩阵上的交叉点 (i,j) 表示第 i 点和第 j 点的连接关系，例如 1 表示相邻，0 表示不相邻。二维矩阵这种数据结构虽然简单、访问速度快，但是用它来存储地图非常浪费空间，因为这是一个稀疏矩阵，其中的交叉点绝大多数等于 0，这些等于 0 的交叉点并不需要存储。一个有 10 万个点的地图，存储它的二维矩阵大小是 100 000×100 000＝10GB。所以，在程序中使用二维矩阵来存储地图是不行的，例如手机上的导航软件，常常有几十万个地点，手机存储卡根本放不下。因此，大地图的存储需要用到更有效率的数据结构，这就是邻接表。

(2) 访问的效率。例如输入一大串个数为 n 的无序数字，如果直接存储到一个一维数组里面，那么要查找到某个数据，只能一个个试，需要的时间是 $O(n)$。如果先按大小排序然后再查询，处理起来就很有效率。在 n 个有序的数中找某个数，用折半查找的方法，可以在

$O(\log_2 n)$的时间里找到。

用数据结构存储和处理数据，可以使程序的逻辑更加清晰。

数据结构有以下 3 个要素[①]。

(1) 数据的逻辑结构：线性结构(数组、栈、队列、链表)、非线性结构、集合、图等。

(2) 数据的存储结构：顺序存储(数组)、链式存储、索引存储、散列存储等。

(3) 数据的运算：初始化、判空、统计、查找、遍历、插入、删除、更新等。

常见的数据结构有数组、链表、栈、队列、树、二叉树、集合、哈希、堆与优先队列、并查集、图、线段树、树状数组等。

在第 3 章中已经介绍了基本的数据结构[②]——栈、队列、链表，本章继续讲解一些常用的高级数据结构，包括并查集、二叉树、线段树、树状数组。

5.1 并 查 集

并查集(Disjoint Set)是一种非常精巧而且实用的数据结构，它主要用于处理一些不相交集合的合并问题。经典的例子有连通子图、最小生成树 Kruskal 算法[③]和最近公共祖先(Lowest Common Ancestors，LCA)等。

通常用“帮派”的例子来说明并查集的应用背景。在一个城市中有 n 个人，他们分成不同的帮派；给出一些人的关系，例如 1 号、2 号是朋友，1 号、3 号也是朋友，那么他们都属于一个帮派；在分析完所有的朋友关系之后，问有多少帮派，每人属于哪个帮派。给出的 n 可能是 10^6 的。

读者可以先思考暴力的方法以及复杂度。如果用并查集实现，不仅代码很简单，而且复杂度可以达到 $O(\log_2 n)$。

并查集：将编号分别为 $1\sim n$ 的 n 个对象划分为不相交集合，在每个集合中，选择其中某个元素代表所在集合。在这个集合中，并查集的操作有初始化、合并、查找。

下面先给出并查集操作的简单实现。在这个基础上，后文再进行优化。


视频讲解

1. 并查集操作的简单实现

(1) 初始化。定义数组 int $s[]$是以结点 i 为元素的并查集，在开始的时候还没有处理点与点之间的朋友关系，所以每个点属于独立的集，并且以元素 i 的值表示它的集 $s[i]$，例如元素 1 的集 $s[1]=1$。

图 5.1 所示为图解，左边给出了元素与集合的值，右边画出了逻辑关系。为了便于讲解，左边区分了结点 i 和集 s(把集的编号加上了下画线)；右边用圆圈表示集，方块表示元素。

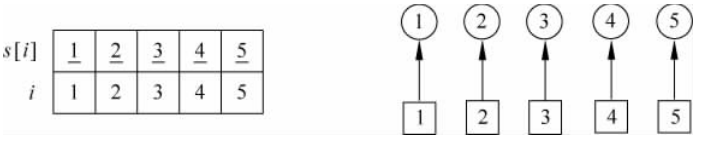


图 5.1 并查集的初始化

① 《数据结构与算法分析新视角》，作者周幸妮等，电子工业出版社。

② 《数据结构(STL 框架)》，作者王晓东，清华大学出版社。

③ 参考本书的“10.10.2 kruskal 算法”。

(2) 合并，例如加入第 1 个朋友关系(1,2)，如图 5.2 所示。在并查集 s 中，把结点 1 合并到结点 2，也就是把结点 1 的集1 改成结点 2 的集2。

图 5.2　合并(1,2)

(3) 合并，加入第 2 个朋友关系(1,3)，如图 5.3 所示。查找结点 1 的集是2，再递归查找元素 2 的集是2，然后把元素 2 的集2 合并到结点 3 的集3。此时，结点 1、2、3 属于一个集。在右图中，为了简化图示，把元素 2 和集2 画在了一起。

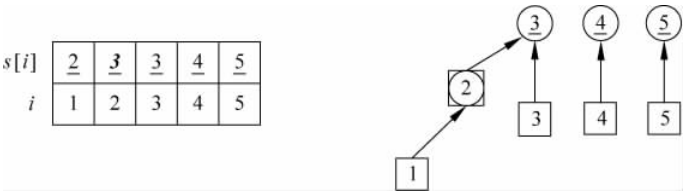


图 5.3　合并(1,3)

(4) 合并，加入第 3 个朋友关系(2,4)，如图 5.4 所示。结果如下，请读者自己分析。

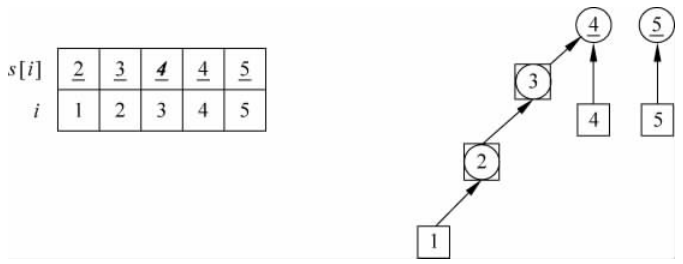


图 5.4　合并(2,4)

(5) 查找。在上面步骤中已经有查找操作。查找元素的集是一个递归的过程，直到元素的值和它的集相等就找到了根结点的集。从上面的图中可以看到，这棵搜索树的高度可能很大，复杂度是 $O(n)$的，变成了一个链表，出现了树的“退化”现象。

(6) 统计有多少个集。如果 $s[i]=i$，这是一个根结点，是它所在的集的代表；统计根结点的数量，就是集的数量。

下面以 hdu 1213 为例实现上述操作。

hdu 1213 “How Many Tables”

有 n 个人一起吃饭，有些人互相认识。认识的人想坐在一起，不想跟陌生人坐。例如 A 认识 B，B 认识 C，那么 A、B、C 会坐在一张桌子上。

给出认识的人，问需要多少张桌子。

一张桌子是一个集，合并朋友关系，然后统计集的数量即可。下面的代码是并查集操作

的具体实现。

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 1050;
int s[maxn];
void init_set(){                                    //初始化
   for(int i = 1; i<=maxn; i++)
        s[i] = i;
}
int find_set(int x){                                //查找
    return x==s[x]? x:find_set(s[x]);
}
void union_set(int x, int y){                       //合并
    x = find_set(x);
    y = find_set(y);
    if(x != y) s[x] = s[y];
}
int main(){
    int t, n, m, x, y;
    cin>>t;
    while(t--){
        cin>>n>>m;
        init_set();
        for(int i = 1; i<=m; i++){
            cin>>x>>y;
            union_set(x, y);
        }
        int ans = 0;
        for(int i = 1; i<=n; i++)                   //统计有多少个集
            if(s[i] == i)
                ans++;
        cout<<ans<<endl;
    }
    return 0;
}
```

复杂度：在上述程序中，查找 find_set()、合并 union_set()的搜索深度是树的长度，复杂度都是 $O(n)$，性能比较差。下面介绍合并和查询的优化方法，优化之后，查找和合并的复杂度都小于 $O(\log_2 n)$。

2. 合并的优化

在合并元素 x 和 y 时先搜到它们的根结点，然后再合并这两个根结点，即把一个根结点的集改成另一个根结点。这两个根结点的高度不同，如果把高度较小的集合并到较大的集上，能减少树的高度。下面是优化后的代码，在初始化时用 height[i]定义元素 i 的高度，在合并时更改。

```
int height[maxn];
void init_set(){
   for(int i = 1; i<=maxn; i++){
```

```
            s[i] = i;
            height[i] = 0;                          //树的高度
        }
    }
    void union_set(int x, int y){                   //优化合并操作
        x = find_set(x);
        y = find_set(y);
        if (height[x] == height[y]) {
            height[x] = height[x] + 1;              //合并,树的高度加一
            s[y] = x;
        }
        else{                                       //把矮树并到高树上,高树的高度保持不变
            if (height[x] < height[y])  s[x] = y;
            else   s[y] = x;
        }
    }
```

3. 查询的优化——路径压缩

在上面的查询程序 find_set()中,查询元素 i 所属的集需要搜索路径找到根结点,返回的结果是根结点。这条搜索路径可能很长。如果在返回的时候顺便把 i 所属的集改成根结点,如图 5.5 所示,那么下次再搜的时候就能在 $O(1)$的时间内得到结果。

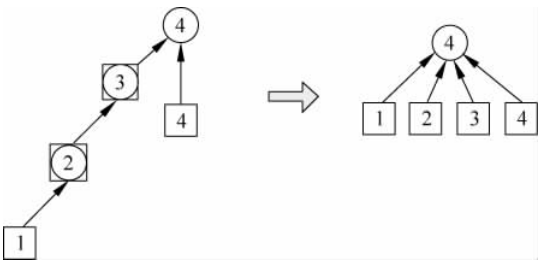


图 5.5　路径压缩

程序如下:

```
int find_set(int x){
    if(x != s[x]) s[x] = find_set(s[x]);        //路径压缩
    return s[x];
}
```

这个方法称为路径压缩,因为在递归过程中,整个搜索路径上的元素(从元素 i 到根结点的所有元素)所属的集都被改为根结点。路径压缩不仅优化了下次查询,而且优化了合并,因为在合并时也用到了查询。

上面的代码用递归实现,如果数据规模太大,担心爆栈,可以用下面的非递归代码:

```
int find_set(int x){
    int r = x;
    while (s[r] != r) r = s[r];                 //找到根结点
    int i = x, j;
    while(i != r){
```

```
            j = s[i];                          //用临时变量 j 记录
            s[i] = r;                          //把路径上元素的集改为根结点
            i = j;
        }
        return r;
    }
```

【习题】

poj 2524 “Ubiquitous Religions”,并查集简单题。
poj 1611 “The Suspects”,简单题。
poj 1703 “Find them, Catch them”。
poj 2236 “Wireless Network”。
poj 2492 “A Bug's Life”。
poj 1988 “Cube Stacking”。
poj 1182 “食物链”,经典题。
hdu 3635 “Dragon Balls”。
hdu 1856 “More is better”。
hdu 1272 “小希的迷宫”。
hdu 1325 “Is It A Tree”。
hdu 1198 “Farm Irrigation”。
hdu 2586 “How far away”,最近公共祖先,并查集+深搜。
hdu 6109 “数据分割”,并查集+启发式合并。

5.2 二 叉 树

树是非线性数据结构,它能很好地描述数据的层次关系。树形结构的现实场景很常见,例如,文件目录、书本的目录就是典型的树形结构。

二叉树是最常用的树形结构,特别适合程序设计,常常将一般的树转换成二叉树来处理。本节讲解二叉树的定义、遍历问题,以及二叉搜索树。

5.2.1 二叉树的存储

1. 二叉树的性质

二叉树的每个结点最多有两个子结点,分别是左孩子、右孩子,以它们为根的子树称为左子树、右子树。

二叉树的第 i 层最多有 2^{i-1} 个结点。如果每一层的结点数都是满的,称它为满二叉树。一个 n 层的满二叉树,结点数量一共有 2^n-1 个,可以依次编号为 $1,2,3,\cdots,2^n-1$。如果满二叉树只在最后一层有缺失,并且缺失的编号都在最后,那么称为完全二叉树。满二叉树和完全二叉树图示如图 5.6 所示。

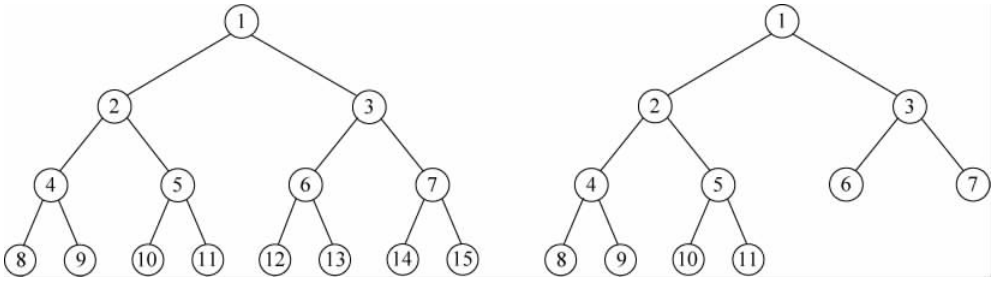


图 5.6　满二叉树和完全二叉树

完全二叉树非常容易操作。一棵结点数量为 k 的完全二叉树，设 1 号点为根结点，有以下性质：

(1) $i>1$ 的结点，其父结点是 $i/2$；

(2) 如果 $2i>k$，那么 i 没有孩子；如果 $2i+1>k$，那么 i 没有右孩子；

(3) 如果结点 i 有孩子，那么它的左孩子是 $2i$，右孩子是 $2i+1$。

2. 二叉树的存储结构

二叉树一般使用指针来实现，并指向左、右子结点。

```
struct node{
    int value;                          //结点的值
    node *l, *r;                        //指向左、右子结点
};
```

在新建一个 node 时，用 new 运算符动态申请内存。使用完毕后，应该用 delete 释放它，否则会内存泄漏。

二叉树也可以用数组来实现。特别是完全二叉树，用数组来表示父结点和子结点的关系非常简便。

5.2.2　二叉树的遍历

视频讲解

1. 宽度优先遍历

有时需要按层次一层层地遍历二叉树。例如在图 5.7 中需要按 $EBGADFICH$ 的顺序访问，那么用宽度优先搜索是最合适的。用队列实现搜索的过程见本书 4.3 节。

2. 深度优先遍历

用深度优先搜索遍历二叉树，代码极其简单。

按深度搜索的顺序访问二叉树，对根(父)结点、左儿子、右儿子进行组合，有先(根)序遍历、中(根)序遍历、后(根)序遍历这 3 种访问顺序，这里默认左儿子在右儿子的前面。

(1) 先序遍历。即按父结点、左儿子、右儿子的顺序访问。在图 5.7 中，访问返回的顺序是 $EBADCGFIH$。

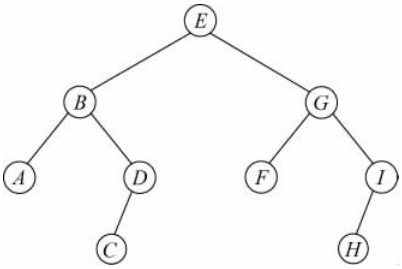


图 5.7　二叉树的遍历

先序遍历的第 1 个结点是根。先序遍历的伪代码如下：

```
void preorder(node * root){
        cout << root -> value;                  //输出
        preorder(root -> l);                    //递归左子树
        preorder(root -> r);                    //递归右子树
}
```

(2) 中序遍历。按左儿子、父结点、右儿子的顺序访问。在图 5.7 中，访问返回的顺序是 *ABCDEFGHI*。读者可能注意到“*ABCDEFGHI*”刚好是字典序，这不是巧合，是因为图示的是一个二叉搜索树。在二叉搜索树中，中序遍历实现了排序功能，返回的结果是一个有序排列。中序遍历还有一个特征：如果已知根结点，那么在中序遍历的结果中，排在根结点左边的点都在左子树上，排在根结点右边的点都在右子树上。例如，*E* 是根，*E* 左边的“*ABCD*”在它的左子树上；再如，在子树“*ABCD*”上，*B* 是子树的根，那么“*A*”在它的左子树上，“*CD*”在它的右子树上。

(3) 后序遍历。按左儿子、右儿子、父结点的顺序访问。在图 5.7 中，访问返回的顺序是 *ACDBFHIGE*。后序遍历的最后一个结点是根。

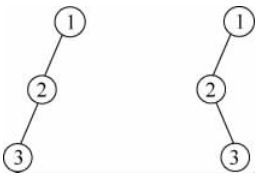


图 5.8 “先序遍历＋后序遍历”不能确定一棵树

如果已知某棵二叉树的 3 种遍历，可以把这棵树构造出来，即“中序遍历＋先序遍历”或者“中序遍历＋后序遍历”，都能确定一棵树。

但是，如果不知道中序遍历，只有“先序遍历＋后序遍历”，不能确定一棵树。例如图 5.8 中两棵不同的二叉树，它们的先序遍历都是“1 2 3”，后序遍历都是“3 2 1”。

上述几种 DFS 遍历的实现见下面例题给出的代码。

hdu 1710 “Binary Tree Traversals”

输入二叉树的先序和中序遍历序列，求后序遍历。

(1) 输入样例。

先序：1 2 4 7 3 5 8 9 6

中序：4 7 2 1 8 5 9 3 6

(2) 输出样例。

后序：7 4 2 8 9 5 6 3 1

建树的过程如下：

(1) 先序遍历的第 1 个数是整棵树的根，例如样例中的“1”。知道了“1”是根，对照中序遍历，“1”左边的“4 7 2”都在根的左子树上，右边的“8 5 9 3 6”都在根的右子树上。

(2) 递归上述过程。例如，上面步骤得到的中序遍历的“4 7 2”，对照先序的第 2 个数是“2”，那么“2”是左子树的根，在中序遍历的“4 7 2”中，“2”左边的“4 7”都在以“2”为根的左子树上，等等。

图 5.9 所示为示意图，画线的数字是读取先序遍历逐一处理的当前步骤的根，方框内是中序遍历的部分数字。

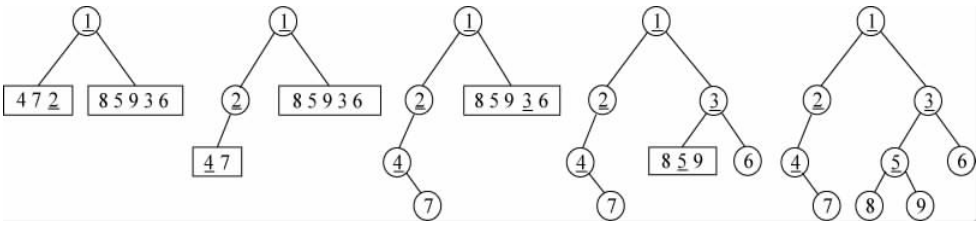


图 5.9　用先序遍历和中序遍历建二叉树

下面是 hdu 1710 的代码，其中 preorder()、inorder()、postorder()分别是先序遍历、中序遍历和后序遍历。可以看到，用 DFS 实现的二叉树遍历代码非常简单。

```
#include <bits/stdc++.h>
using namespace std;
const int N = 1010;
int pre[N], in[N], post[N];                    //先序、中序、后序
int k;
struct node{
    int value;
    node *l, *r;
    node(int value = 0, node *l = NULL, node *r = NULL):value(value), l(l), r(r){}
};
void buildtree(int l, int r, int &t, node* &root) {     //建树
    int flag = -1;
    for(int i = l; i <= r; i++)                 //先序的第1个数是根，找到对应的中序的位置
        if(in[i] == pre[t]){
            flag = i; break;
        }
    if(flag == -1) return;                      //结束
    root = new node(in[flag]);                  //新建结点
    t++;
    if(flag > l)  buildtree(l, flag - 1, t, root->l);
    if(flag < r)  buildtree(flag + 1, r, t, root->r);
}
void preorder(node *root){                      //求先序序列
    if(root != NULL){
        post[k++] = root->value;                //输出
        preorder(root->l);
        preorder(root->r);
    }
}
void inorder(node *root){                       //求中序序列
    if(root != NULL){
        inorder (root->l);
        post[k++] = root->value;                //输出
        inorder(root->r);
    }
}
void postorder(node *root){                     //求后序序列
```

```
    if(root != NULL){
        postorder(root -> l);
        postorder(root -> r);
        post[k++] = root -> value;              //输出
    }
}
void remove_tree(node * root){                  //释放空间
    if(root == NULL) return;
    remove_tree(root -> l);
    remove_tree(root -> r);
    delete root;
}
int main(){
    int n;
    while(~scanf("%d", &n)){
        for(int i = 1;i <= n;i++) scanf("%d", &pre[i]);
        for(int j = 1;j <= n;j++) scanf("%d", &in[j]);
        node * root;
        int t = 1;
        buildtree(1, n, t, root);
        k = 0;                                  //记录结点个数
        postorder(root);
        for(int i = 0;i < k;i++) printf("%d%c",post[i],i == k - 1?'\n':' ');
          //作为验证,这里可以用 preorder()和 inorder()检查先序和中序遍历
        remove_tree(root);
    }
    return 0;
}
```

代码中的 remove_tree()释放申请的空间，如果不释放，会内存泄漏，造成内存浪费。释放空间是标准的、正确的操作。不过，竞赛题目的代码很少，即使不释放空间，也不会出错；而且程序终止后，它申请的空间也会被系统收回。

5.2.5 节给出了用数组实现二叉树的例子。

5.2.3 二叉搜索树

BST(Binary Search Tree，二叉搜索树)是非常有用的数据结构，它的结构精巧、访问高效。BST 的特征如下：

(1) 每个元素有唯一的键值，这些键值能比较大小。通常把键值存放在 BST 的结点上。

(2) 任意一个结点的键值，比它左子树的所有结点的键值大，比它右子树的所有结点的键值小。也就是说，在 BST 上，以任意结点为根结点的一棵子树仍然是 BST。BST 是一棵有序的二叉树。可以推出，键值最大的结点没有右儿子，键值最小的结点没有左儿子。

图 5.10 是一棵二叉搜索树，用中序遍历可以得到它的有序排列。右图的虚线把每个结点隔开，很容易看出，结点正好按从小到大的顺序被虚线隔开了。有虚线的帮助，很容易理解后文介绍 Treap 树和 Splay 树时提到的“旋转”技术。

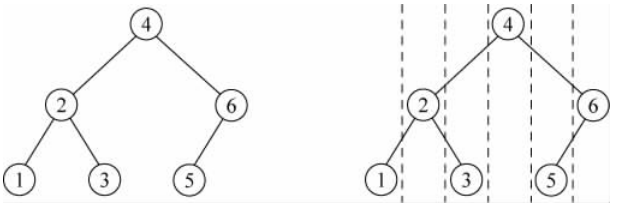


图 5.10　二叉搜索树

数据的基本操作是插入、查询、删除。给定一个数据序列，如何实现 BST？下面给出一种朴素的实现方法。

(1) 建树和插入。以第 1 个数据 x 为根结点，逐个插入其他所有数据。插入过程从根结点开始，如果数据 y 比根结点 x 小，就往 x 的左子树上插，否则就往右子树上插；如果子树为空，就直接放到这个空位，如果非空，就与子树的值进行比较，再进入子树的下一层，直到找到一个空位置。新插入的数据肯定位于一个最底层的叶子结点，而不是插到中间某个结点上替代原来的数据。

从建树的过程可知，如果按给定序列的顺序进行插入，最后建成的 BST 是唯一的。形成的 BST 可能很好，也可能很坏。在最坏的情况下，例如一列有序整数{1，2，3，4，5，6，7}，按顺序插入，会全部插到右子树上；BST 退化成一个只包含右子树的链表，从根结点到最底层的叶子，深度是 n，导致访问一个结点的复杂度是 $O(n)$。在最好的情况下，例如序列{4，2，1，3，6，5，7}，得到的 BST 左、右子树是完全平衡的，深度是 $\log_2 n$，访问复杂度是 $O(\log_2 n)$。退化的 BST 和平衡 BST 如图 5.11 所示。

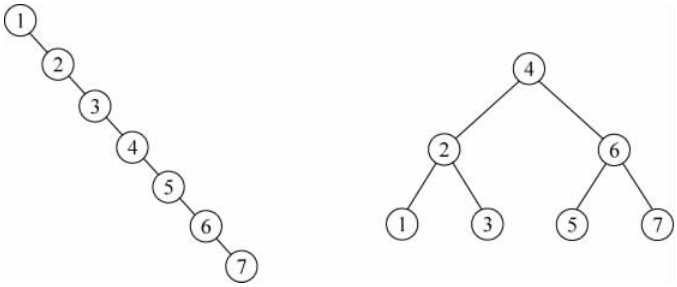


图 5.11　退化的 BST 和平衡 BST

(2) 查询。建树过程实际上也是一个查询过程，所以查询仍然是从根结点开始的递归过程。访问的复杂度取决于 BST 的形态。

(3) 删除。在删除一个结点 x 后，剩下的部分应该仍然是一个 BST。首先找到被删结点 x，如果 x 是最底层的叶子结点，直接删除；如果 x 只有左子树 L 或者只有右子树 R，直接删除 x，原位置由 L 或 R 代替。如果 x 左、右子树都有，情况就复杂了，此时，原来以 x 为根结点的子树需要重新建树。一种做法是，搜索 x 左子树中的最大元素 y，移动到 x 的位置，这相当于原来以 y 为根结点的子树，删除了 y，然后继续对 y 的左子树进行类似的操作，这也是一个递归的过程。删除操作的复杂度也取决于 BST 的形态。

(4) 遍历。在 5.2.2 节中提到用中序遍历 BST，返回的是一个从小到大的排序。

根据上述过程可知，BST 的优劣取决于它是否为一个平衡的二叉树。所以，BST 有

关算法的主要功能是努力使它保持平衡。那么如何实现一个平衡的 BST？由于无法提前安排元素的顺序(如果能一次读入所有元素，也能调整顺序，但是会大费周章，没有必要)，所以只能在建树之后通过动态调整使它变得平衡。BST 算法的区别就在于用什么办法调整。

BST 算法有 AVL 树、红黑树、Splay 树、Treap 树、SBT 树等。其中容易编程的有 Splay 树、Treap 树等，也是算法竞赛中容易出的题目，本节后续讲解 Treap 树和 Splay 树。

BST 是一个动态维护的有序数据集，用 DFS 对它进行中序遍历可以高效地输出字典序、查找第 k 大的数等。

STL 与 BST。STL 中的 set 和 map 是用二叉搜索树(红黑树)实现的，检索和更新的复杂度是 $O(\log_2 n)$。如果一个题目需要快速访问集合中的数据，可以用 set 或 map 实现，内容见本书第 3 章。

【习题】

hdu 3999 “The order of a Tree”，模拟 BST 的建树和访问。

hdu 3791 “二叉搜索树”，模拟 BST。

poj 2418 “Hardwood Species”，用 map 快速处理字符串。

5.2.4 Treap 树

首先研究一种比较简单的平衡二叉搜索树——Treap 树。

Treap 是一个合成词，把 Tree 和 Heap 各取一半组合而成。Treap 是树和堆的结合，可以翻译成树堆。

二叉搜索树的每个结点有一个键值，除此之外，Treap 树为每个结点人为添加了一个被称为优先级的权值。对于键值来说，这棵树是排序二叉树；对于优先级来说，这棵树是一个堆。堆的特征是：在这棵树的任意子树上，根结点的优先级最大。

1. Treap 树的唯一性

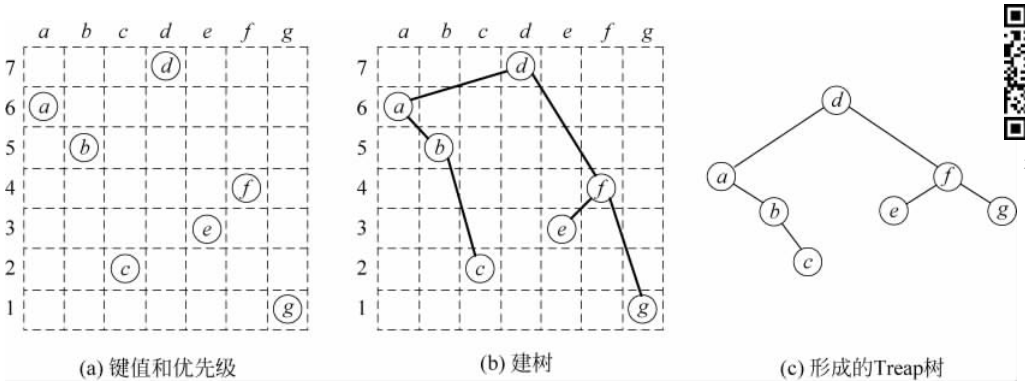
Treap 树的重要性质：令每个结点的优先级互不相等，那么整棵树的形态是唯一的，和元素的插入顺序没有关系。

下面用 7 个结点举例说明建树过程，其键值分别是 $\{a,b,c,d,e,f,g\}$，优先级分别是 $\{6,5,2,7,3,4,1\}$。图 5.12(a)的纵向是优先级，横向是结点的键值；图 5.12(b)按二叉搜索树的规则建了一棵树；图 5.12(c)是结果。从这个图中可以看出 Treap 树的形态是唯一的。

2. Treap 树的平衡问题

从图 5.12 可知，树的形态依赖于结点的优先级。那么如何配置每个结点的优先级，才能避免二叉树的形态退化成链表？最简单的方法是把每个结点的优先级进行随机赋值，那么生成的 Treap 树的形态也是随机的。这虽然不能保证每次生成的 Treap 树一定是平衡的，但是期望[①]的插入、删除、查找的时间复杂度都是 $O(\log_2 n)$的。

① 关于期望的概念，见本书中的“8.4 概率和数学期望”。




视频讲解

图 5.12　Treap 树的形态

了解了 Treap 树的概念，读者可以尝试自己完成建树的过程。在阅读下面的内容之前，不妨自己先试一试。

3. Treap 树的插入

如果预先知道所有结点的优先级，那么建树很简单，先按优先级排序，然后按优先级从高到低的顺序插入即可。例如在图 5.12 中，最高优先级的 d 第 1 个插入，是树根；第 2 优先级的 a 比 d 小，插到 d 的左子树上；第 3 优先级的 b 比 a 大，插到 a 的右子树……

不过，其实并不需要这么做。更简单的做法是每读入一个新结点，为它分配一个随机的优先级，插入到树中，在插入时动态调整树的结构，使它仍然是一棵 Treap 树。

把新结点 node 插入到 Treap 树的过程有以下两步：

(1) 用朴素的插入方法把 node 按键值大小插入到合适的子树上。

(2) 给 node 随机分配一个优先级，如果 node 的优先级违反了堆的性质，即它的优先级比父结点高，那么让 node 往上走，替代父结点，最后得到一个新的 Treap 树。

步骤(2)中的调整过程用到了一种技巧——旋转，包括左旋和右旋，如图 5.13 所示。

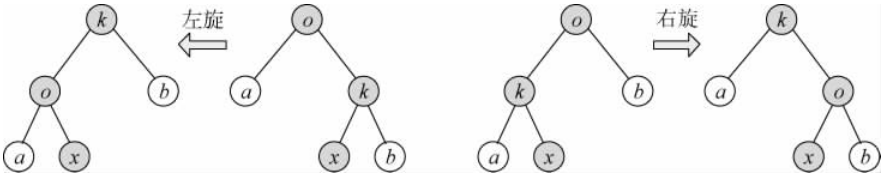


图 5.13　Treap 树的旋转(把 k 旋转到根)

旋转的代码如下，其中 son[0]是左儿子，son[1]是右儿子，代码中定义的结点名称和图 5.13 中的结点名称对应。

```
void rotate(Node * &o,int d){                    //d = 0,左旋转; d = 1,右旋
        Node * k = o -> son[d ^ 1];              //d ^ 1 与 1 - d 等价,但是更快
        o -> son[d ^ 1] = k -> son[d];           //图中的 x
        k -> son[d] = o;
        o = k;                                   //返回新的根
    }
```

这里仍然以键值为{a,b,c,d,e,f,g}、优先级为{6,5,2,7,3,4,1}的 Treap 树为例，调整过程如下：图 5.14(a)是初始 Treap 树；图 5.14(b)插入 d 点，按朴素的插入方法插入到底部；图 5.14(c)中 d 的优先级比父结点 c 高，左旋，上升；图 5.14(d)中 d 的优先级比新的父结点 b 高，继续左旋，上升；图 5.14(e)中，d 再次左旋，上升，完成了新的 Treap 树。

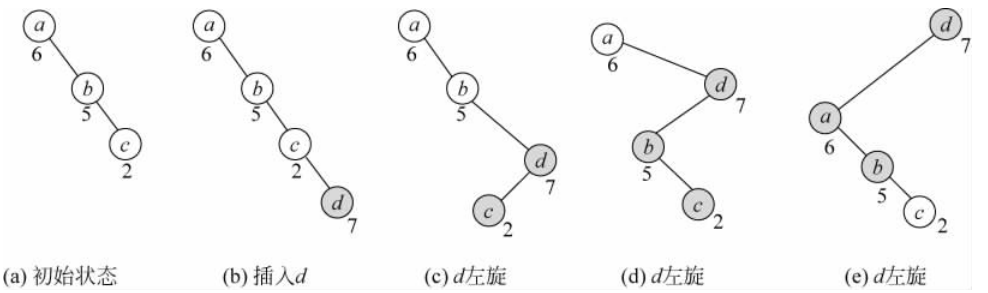


图 5.14　Treap 树的插入和调整

4. Treap 树的删除

如果待删除的结点 x 是叶子结点，直接删除。

如果待删除的结点 x 有两个子结点，那么找到优先级最大的子结点，把 x 向相反的方向旋转，也就是把 x 向树的下层调整，直到 x 被旋转到叶子结点，然后直接删除。

5. 分裂与合并问题

有时需要把一棵树分裂成两棵树，或者把两棵树合并成一棵树。Treap 树做这样的操作是比较烦琐的。读者可以用上面的例子尝试一下分裂和合并，例如在图 5.12(c)中，先把树分成{a,b}和{c,d,e,f,g}两棵树，然后再合并。注意在分裂和合并时仍然需要符合 Treap 树的规则。

5.2.5 节提到的 Splay 树做分裂和合并的操作非常简便。

6. Treap 与名次树问题

竞赛中与 Treap 有关的题目很多涉及名次树，例如：

hdu 4585 “Shaolin”

少林寺的第 1 个和尚是方丈，作为功夫大师，他规定每个加入少林寺的年轻和尚要选一个老和尚来一场功夫战斗。每个和尚有一个独立的 id 和独立的战斗等级，新和尚可以选择跟他的战斗等级最接近的老和尚战斗。

方丈的 id 是 1，战斗等级是 10^9。他丢失了战斗记录，不过他记得和尚们加入少林寺的早晚顺序。请帮他恢复战斗记录。

输入：第 1 行是一个整数 n，$0<n\leqslant 100\,000$，和尚的人数，但不包括方丈本人。下面有 n 行，每行有两个整数 k、g，表示一个和尚的 id 和战斗等级，$0\leqslant k,g\leqslant 5\,000\,000$。和尚以升序排序，即按加入少林寺的时间排序。最后一行用 0 表示结束。

输出：按时间顺序给出战斗，打印出每场战斗中新和尚和老和尚的 id。

输入样例：

```
3
2 1
3 3
4 2
0
```

输出样例：

```
2 1
3 2
4 2
```

题意很简单，先对老和尚的等级排序，在加入一个新和尚时，找到等级最接近的老和尚，输出老和尚的 id。由于题目给的 n 比较大，因此总复杂度需要是 $O(n\log_2 n)$的。

此题有多种解法，这里给出两种解法——STL map、Treap 树。

1）STL map 代码

STL 的 map 和 set 都是用二叉搜索树实现的。这一题可以用 map 来做。

```
#include<bits/stdc++.h>
using namespace std;
map<int, int> mp;                                    //it->first 是等级,it->second 是 id
int main(){
    int n;
    while (~scanf("%d",&n) && n){
        mp.clear();
        mp[1000000000] = 1;                          //方丈 1,等级是 1 000 000 000
        while(n--){
            int id,g;
            scanf("%d%d",&id,&g);                    //新和尚 id,等级是 g
            mp[g] = id;                              //新和尚进队
            int ans;
            map<int,int>::iterator it = mp.find(g);     //找到排好序的位置
            if (it == mp.begin())  ans = (++it)->second;
                else{
                map<int,int> :: iterator it2 = it;
                it2--; it++;                         //等级接近的前后两个老和尚
                if (g-it2->first <= it->first-g)
                    ans = it2->second;
                else ans = it->second;
            }
            printf("%d %d\n",id,ans);
        }
    }
    return 0;
}
```

2）Treap 树代码

下面的 Treap 程序[①]给出了 Treap 树的常用操作：定义结点 struct Node、旋转 rotate()、插入 insert()、找第 k 大的数 kth()、查询某个数 find()。

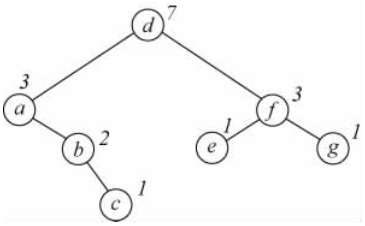


图 5.15　名次树

注意其中的 kth()和 find()，它与名次树问题有关。名次树有两个功能：①找到第 k 大的元素；②查询元素 x 的名次，即 x 排名第几。这两个功能的实现借助于给结点增加的一个 size 值。一个结点的 size 值是以它为根的子树的结点总数量，例如图 5.15 所示的名次树。图中结点上标注的数字就是这个结点的 size。

下面的代码中给出了找第 k 大数的函数 kth()以及查询元素名次的函数 find()，它们的复杂度都是 $O(\log_2 n)$的。

hdu 4585 的 Treap 代码（名次树）

```
#include <bits/stdc++.h>
using namespace std;
int id[5000000 + 5];
struct Node{
    int size;                                   //以这个结点为根的子树的结点总数量,用于名次树
    int rank;                                   //优先级
    int key;                                    //键值
    Node * son[2];                              //son[0]是左儿子, son[1]是右儿子
    bool operator < (const Node &a)const{return rank < a.rank;}
    int cmp(int x)const{
        if(x == key) return -1;
        return x < key?0:1;
    }
    void update(){                              //更新 size
        size = 1;
        if(son[0]!= NULL) size += son[0]->size;
        if(son[1]!= NULL) size += son[1]->size;
        }
};
void rotate(Node * &o,int d){                   //d = 0,左旋; d = 1,右旋
        Node * k = o->son[d^1];                 //d^1 与 1-d 等价,但是更快
        o->son[d^1] = k->son[d];
        k->son[d] = o;
        o->update();
        k->update();
        o = k;
    }
void insert(Node * &o,int x){                   //把 x 插入到树中
        if(o == NULL){
            o = new Node();
            o->son[0] = o->son[1] = NULL;
            o->rank = rand();
            o->key = x;
```

① 部分代码改编自《算法竞赛入门经典训练指南》，作者刘汝佳、陈锋，清华大学出版社，3.5.2 节，231 页。

```
            o->size = 1;
        }
        else {
            int d = o->cmp(x);
            insert(o->son[d],x);
            o->update();
            if(o<o->son[d])
                rotate(o,d^1);
        }
    }
int kth(Node* o,int k){                          //返回第 k 大的数
        if(o==NULL||k<=0||k>o->size)
            return -1;
        int s = o->son[1]==NULL?0:o->son[1]->size;
        if(k==s+1) return o->key;
        else if(k<=s) return kth(o->son[1],k);
        else return kth(o->son[0],k-s-1);
    }
int find(Node* o,int k){  //返回元素 k 的名次
        if(o==NULL)
            return -1;
        int d = o->cmp(k);
        if(d==-1)
            return o->son[1]==NULL? 1: o->son[1]->size+1;
        else if(d==1) return find(o->son[d],k);
        else{
            int tmp = find(o->son[d],k);
            if(tmp==-1) return -1;
            else
                return o->son[1]==NULL? tmp+1 : tmp+1+o->son[1]->size;
        }
    }
int main(){
    int n;
    while(~scanf("%d",&n)&&n){
        srand(time(NULL));
        int k,g;
        scanf("%d%d",&k,&g);
        Node *root = new Node();
        root->son[0] = root->son[1] = NULL;
        root->rank = rand(); root->key = g; root->size = 1;
        id[g] = k;
        printf("%d %d\n",k,1);
        for(int i=2;i<=n;i++){
            scanf("%d%d",&k,&g);
            id[g] = k;
            insert(root,g);
            int t = find(root,g);                //返回新和尚的名次
            int ans1,ans2,ans;
            ans1 = kth(root,t-1);                //前一名的老和尚
            ans2 = kth(root,t+1);                //后一名的老和尚
            if(ans1!=-1&&ans2!=-1)
                ans = ans1-g>=g-ans2 ? ans2:ans1;
            else if(ans1==-1) ans = ans2;
```

```
                else ans = ans1;
                printf("%d %d\n",k,id[ans]);
            }
        }
        return 0;
    }
```

【习题】

poj 1442，名次树问题。

hdu 3726 “Graph and Queries”，离线算法＋Treap 维护名次树。该题非常经典，是必做题。

5.2.5 Splay 树

Splay 树是一种 BST 树，它的查找、插入、删除、分割、合并等操作，复杂度都是 $O(\log_2 n)$ 的。它最大的特点是可以把某个结点往上旋转到指定位置，特别是可以旋转到根的位置，成为新的根结点。它有这样一种应用背景：如果需要经常查询和使用一个数，那么把它旋转到根结点，这样下次访问它，只需要查一次就找到了。

Splay 树有 Treap 树不具备的特点：①Splay 树允许把任意结点旋转到根，而 Treap 树不能，因为它的形态是固定的；②当需要分裂和合并时，Splay 树的操作非常简便。

下面介绍 Splay 操作，其中提根操作是核心。

1. 把结点旋转到根（提根）

Splay 树比 Treap 树的旋转操作的情况更多。

那么如何把一个结点 x 自底向上旋转到根？根据 x 的位置，有以下 3 种情况。

（1）x 的父结点就是根，只需要旋转一次。图 5.16 给出了 x 是根 c 的左儿子的情况，右儿子的情况与之类似。注意观察图中的中序遍历，即二叉搜索树的顺序“$a\ x\ b\ c\ d$”，保持不变。

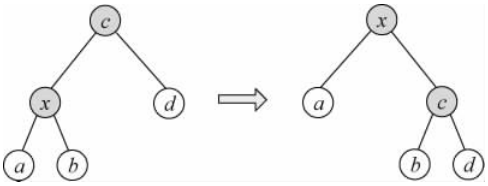


图 5.16　Splay 旋转情况 1

（2）x 的父结点不是根，x、x 的父结点、x 的父结点的父结点，三点共线。此时可以做两次单旋，即先旋转 x 的父结点，再旋转 x，如图 5.17 所示。

（3）x、x 的父结点、x 的父结点的父结点，三点不共线。把 x 按不同方向旋转两次，如图 5.18 所示。

按上述方法可以把任何深度的结点 x 旋转到根。

旋转一次的时间是个常数，那么把 x 从所在的深度提到根，总复杂度是多少？如果是平衡二叉树，最深的结点深度是 $O(\log_2 n)$，那么总复杂度就是 $O(\log_2 n)$。当然二叉树不一

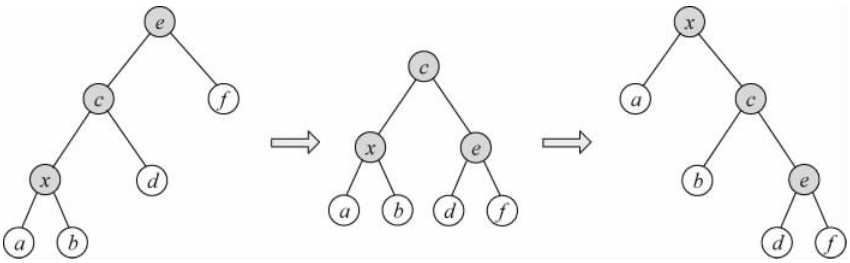


图 5.17　Splay 旋转情况 2

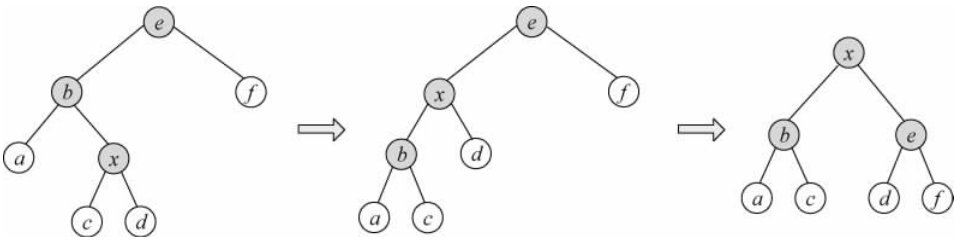


图 5.18　Splay 旋转情况 3

定是平衡的，不过在均摊意义上，可以把 Splay 提根操作的复杂度看成是 $O(\log_2 n)$的。这就是二叉树这种数据结构带来的优势。

下面的插入、分裂、合并，复杂度和提根的复杂度类似。

2. 插入

插入和普通二叉搜索树的方法一样。在插入之后，可以根据需要对新插入的结点做 Splay 操作。

3. 分裂

以第 k 小的数为界，把树分成两部分。先把第 k 小的元素旋转到树根，然后把它与右子树断开，就得到了两棵树。

4. 合并

可合并的两棵树，其中一棵树(设为 left)的所有元素应该小于另一棵树(设为 right)的所有元素。合并过程是先把 left 的最大元素 x 伸展到树根，此时树根 x 没有右子树，把 x 的右子树接到 right 的根，就完成了合并。

5. 删除

把待删除的结点旋转到根，删除它，然后合并左、右子树。

下面的例题给出了 Splay 树的编程细节。

hdu 1890 "Robotic Sort"

有 n 个数字，$1 \leqslant n \leqslant 100\,000$，用一个机械臂帮忙排序，其方法如图 5.19 所示。在左图中(图中的高度是数字大小)，用机械臂夹住第 1 个数和最小的数，翻转，变成中图的样子，最小的数就处于第 1 个位置。然后对中图用同样的方法找第 2 小的数。继续这个过程直到结束。

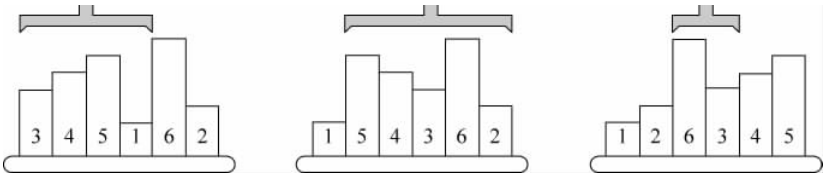


图 5.19　排序方法

输入一些数字，输出第 i 次翻转之前第 i 大的数的位置。

输入样例：3 4 5 1 6 2

输出样例：4 6 4 5 6 6

题目的基本操作是找到第 i 大的数，翻转它左边的数（不包括已经处理过的比它小的数），右边的数保持不变。如果用模拟法编程，复杂度约为 $O(n^3)$，会 TLE。本题需要 $O(n\log_2 n)$的方法。

注意，翻转有两种方法，这里以第 1 次翻转 3 4 5 1 为例：方法 1，直接翻转 3 4 5 1；方法 2，先把 1 挪到最左边，然后翻转 3 4 5。这两种方法的结果一样，在下面的 Splay 程序中适合用第 2 种方法。

本题的操作可以用 Splay 来模拟，利用了 Splay 树能把结点旋转到根的功能。

下面以第 1 个数的处理为例来说明过程。

（1）建树。把这个序列按初始位置建一个二叉搜索树。图 5.20(a)是建树的结果，圆圈内是初始位置，圆圈旁边的数字是题目给出的序列。根据中序遍历，它是题目的样例3 4 5 1 6 2。建树的代码是 buildtree()。

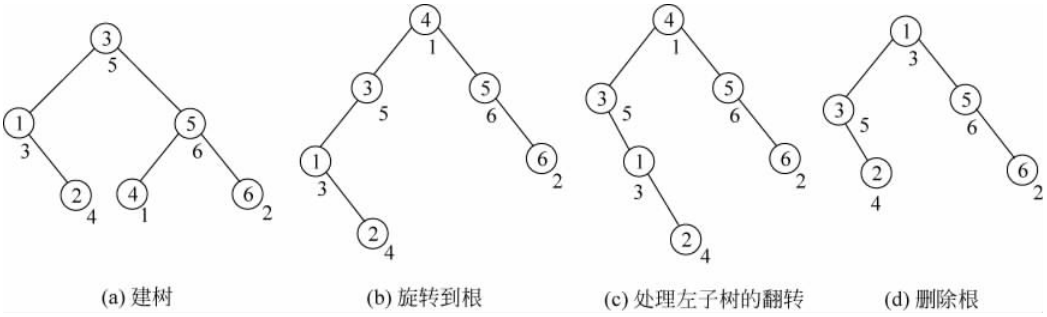


图 5.20　hdu 1890 题

（2）用 Splay 旋转到根。找到最小的数，用 Splay 把它旋转到根。其左子树的大小就是数列中排在它左边的数的个数，也就是题目的输出。其右边的数的顺序保持不变，左边的数需要模拟机械臂的翻转。旋转的代码是 splay()。

（3）翻转左子树。模拟题目中的机械臂翻转，但是，如果每次都完全翻转左子树，时间必然超时。这里从线段树[①]得到启发，用标记的方式记录翻转情况，减少直接操作的次数，

① 类似线段树的 lazy 操作，见本书中的“5.3.4 区间修改”。

等 Splay 操作的时候再处理。图(c)中只翻转了结点 3,对结点 3 做标记,而它的子树 1、2 保持不变。标记的代码是 update_rev()。注意,翻转会改变 BST 树的有序结构,所以本题并不是 Splay 树的裸题,只是用到了 Splay 树的旋转功能。在下面给出的代码中,如果去掉 update_rev(),就是纯粹的 Splay 代码。

(4) 删除根,即在树上删除最小数。在删除过程中,根据标记进行子树的翻转。最后的结果见图(d),这是去掉了最小数的树,第 1 次处理结束。删除根的代码是 del_root()。

下面是 hdu 1890 的代码。该代码中去掉 update_rev()、pushup()、pushdown(),就是纯的 Splay 代码。

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=100010;
int root;                                    //根
int rev[maxn],pre[maxn],size[maxn];
    //rev[i],标记 i 被翻转;pre[i],i 的父结点;size[i],i 的子树上结点的个数
int tree[maxn][2]; //记录树:tree[i][0],i 的左儿子;tree[i][1],i 的右儿子
struct node{
    int val,id;
    bool operator<(const node &A)const {   //用于 sort()排序
        if(val==A.val)return id<A.id;
        return val<A.val;
    }
}nodes[maxn];
void pushup(int x){                          //计算以 x 为根的子树包含多少子结点
    size[x]=size[tree[x][0]]+size[tree[x][1]]+1;
}
void update_rev(int x){
    if(!x)return;
    swap(tree[x][0],tree[x][1]);             //翻转 x:交换左右儿子
    rev[x]^=1;                               //标记 x 被翻转
}
void pushdown(int x){                        //在做 Splay 时,根据本题的需要,处理机械臂翻转
    if(rev[x]){
        update_rev(tree[x][0]);
        update_rev(tree[x][1]);
        rev[x]=0;
    }
}
void Rotate(int x,int c){                    //旋转,c=0 为左旋,c=1 为右旋
    int y=pre[x];
    pushdown(y);
    pushdown(x);
    tree[y][!c]=tree[x][c];
    pre[tree[x][c]]=y;
    if(pre[y])
        tree[pre[y]][tree[pre[y]][1]==y]=x;
    pre[x]=pre[y];
    tree[x][c]=y;
```

```
    pre[y] = x;
    pushup(y);
}
void splay(int x,int goal){
        //把结点 x 旋转为 goal 的儿子,如果 goal 是 0,则旋转到根
    pushdown(x);
    while(pre[x]!= goal){                    //一直旋转,直到 x 成为 goal 的儿子
        if(pre[pre[x]] == goal){             //情况(1):x 的父结点是根,单旋一次即可
            pushdown(pre[x]); pushdown(x);
            Rotate(x,tree[pre[x]][0] == x);
        }
        else{                                //x 的父结点不是根
            pushdown(pre[pre[x]]); pushdown(pre[x]); pushdown(x);
            int y = pre[x];
            int c = (tree[pre[y]][0] == y);
            if(tree[y][c] == x){             //情况(2):x、x 的父、x 父的父,不共线
                Rotate(x,!c);
                Rotate(x,c);
            }
            else{                            //情况(3):x、x 的父、x 父的父,共线
                Rotate(y,c);
                Rotate(x,c);
            }
        }
    }
    pushup(x);
    if(goal == 0) root = x;                  //如果 goal 是 0,则将根结点更新为 x
}
int get_max(int x){
    pushdown(x);
    while(tree[x][1]){
        x = tree[x][1];
        pushdown(x);
    }
    return x;
}
void del_root(){                             //删除根结点
    if(tree[root][0] == 0){
        root = tree[root][1];
        pre[root] = 0;
    }
    else{
        int m = get_max(tree[root][0]);
        splay(m,root);
        tree[m][1] = tree[root][1];
        pre[tree[root][1]] = m;
        root = m;
        pre[root] = 0;
        pushup(root);
    }
}
```

```
void newnode(int &x,int fa,int val){
    x = val;
    pre[x] = fa;
    size[x] = 1;
    rev[x] = 0;
    tree[x][0] = tree[x][1] = 0;
}
void buildtree(int &x,int l,int r,int fa){ //建树
    if(l > r) return;
    int mid = (l + r)>> 1;
    newnode(x,fa,mid);
    buildtree(tree[x][0],l,mid - 1,x);
    buildtree(tree[x][1],mid + 1,r,x);
    pushup(x);
}
void init(int n){
    root == 0;
    tree[root][0] = tree[root][1] = pre[root] = size[root] = 0;
    buildtree(root,1,n,0);
}
int main(){
    int n;
    while(~scanf("%d",&n) && n){
        init(n);
        for(int i = 1;i <= n;i++){
            scanf("%d",&nodes[i].val);  nodes[i].id = i;
        }
        sort(nodes + 1,nodes + n + 1);
        for(int i = 1;i < n;i++){
            splay(nodes[i].id,0);          //第 i 次翻转:把第 i 大的数旋到根
            update_rev(tree[root][0]);   //左子树需要翻转
            printf("%d ",i + size[tree[root][0]]);
              //i:第 i 次翻转;size:第 i 个被翻转数的左边的个数,就是它左子树的个数
            del_root();                    //删除第 i 次翻转的数,准备下一次翻转
        }
        printf("%d\n",n);
    }
    return 0;
}
```

读者可以在上面代码的基础上写出 Splay 树常见操作的代码,例如:

(1) 查找 x。执行 splay(x, 0),即把 x 旋转到根结点。

(2) 删除 x。先执行 splay(x, 0),把 x 旋转到根,然后用 del_root()删除它。

(3) 查找最大、最小、第 k 大的数。用中序遍历进行查找,查找后可以用 splay()把它旋转到根。

【习题】

hdu 1622,建二叉树;

hdu 3999,二叉树遍历;

hdu 3791，BST；

hdu 4453，Splay 基本题；

hdu 3726，离线处理＋Splay，经典题。该题用 Treap 树也能做。

5.3 线 段 树

有这样一类 RMQ(Range Minimum/Maximum Query)问题，求区间最大值或最小值。设有长度为 n 的数列$\{a_1,a_2,\cdots,a_n\}$，需要进行以下操作。

(1) 求最值：给定 $i,j\leqslant n$，求$\{a_i,\cdots,a_j\}$区间内的最值。

(2) 修改元素：给定 k 和 x，把 a_k 改成 x。

如果用普通数组存储数列，上面两个操作中，求最值的复杂度是 $O(n)$，修改是 $O(1)$。如果有 m 次“修改元素＋查询最值”，那么总复杂度是 $O(mn)$。如果 m 和 n 比较大，例如 100 000 以上，那么整个程序的复杂度是 10^{10} 的数量级。这个复杂度在竞赛中是不可承受的。

除了 RMQ 问题以外，类似的还有求区间和问题。对于数列$\{a_1,a_2,\cdots,a_n\}$，先更改某些数的值，然后给定 $i,j\leqslant n$，求 sum$=a_i+\cdots+a_j$ 的区间和。对于单个更改或者求和，很容易写出 $O(n)$ 的算法；如果更改和询问的操作总次数是 m，那么整个程序的复杂度是 $O(mn)$。和 RMQ 一样，这样的复杂度也是不行的。

对于这类问题，有一种神奇的数据结构，能在 $O(m\log_2 n)$的时间内解决，这就是线段树。

5.3.1 线段树的概念

视频讲解

线段树是一种用于区间处理的数据结构，用二叉树来构造。

线段树是建立在线段(或者区间)基础上的树，树的每个结点代表一条线段[L,R]。图 5.21 所示为是线段[1,5]的线段树。

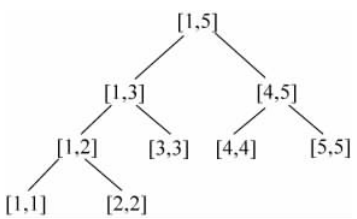


图 5.21 线段[1, 5]的线段树结构

考查每个线段$[L,R]$，L 是左子结点，R 是右子结点。

(1) $L=R$，说明这个结点只有一个点，它就是一个叶子结点。

(2) $L<R$，说明这个结点代表的不止一个点，它有两个儿子，左儿子代表的区间是$[L, M]$，右儿子代表的区间是$[M+1, R]$，其中 $M=(L+R)/2$。

线段树是二叉树，一个区间每次被折一半往下分，所以最多分 $\log_2 n$ 次就到达最低层。当需要查找一个点或者区间的时候，顺着结点往下找，最多 $\log_2 n$ 次就能找到。这就是线段树效率高的原因，使用了二叉树折半查找的方法。

回到 RMQ 问题，如果用线段树，“修改元素＋查询最值”这两个操作分别可以在 $O(\log_2 n)$的时间内完成。如图 5.22 所示，查询$\{1,2,5,8,6,4,3\}$的最小值，其中每个结点上的数字是这棵子树的最小值。

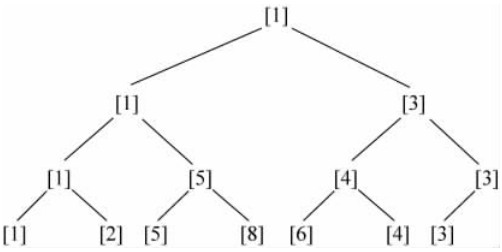


图 5.22　RMQ 问题(查询最小值)

如需修改元素,直接修改叶子结点上元素的值后从底往上更新线段树,操作次数也是 $O(\log_2 n)$。

m 次"修改+查询"的总复杂度是 $O(m\log_2 n\log_2 n)$。实际上,修改和查询可以同时做,所以总复杂度是 $O(m\log_2 n)$。这对规模 100 万的问题也能轻松解决。

5.3.2　点修改

首先讨论在线段树中每次只修改一个点的问题。

线段树如何构造?如何更新?如何查询?下面以 poj 2182 为例,引导出线段树的应用和编程细节。

poj 2182 "Lost Cows"

题目描述:有编号是 1～n 的 n 个数字,$2 \leqslant n \leqslant 8000$,乱序排列,顺序是未知的。对于每个位置的数字,知道排在它前面比它小的数字有多少个。求这个乱序数列的顺序。

例如有 5 个数,已知每个数字前面比它小的数的个数,分别是:

pre[]:　0 1 2 1 0

可以求得这个乱序排列是:

ans[]:　2 4 5 3 1

本题是"简单题",用线段树或者树状数组实现。

在讲解后续内容之前,这里先用暴力法实现,思路是从后往前处理 pre[]:

(1) pre[5]=0,表示 ans[5]前面比它小的数有 0 个,即 ans[5]是最小的,在 1～5 这几个编号中 1 最小,所以 ans[5]=1。ans[]的前 4 个编号不再包括 1,剩下 2～5 这几个编号。

(2) pre[4]=1,在剩下的 2～5 这几个编号中,编号 3 是第 2 大的,所以 ans[4]=3。

(3) pre[3]=2,在剩下的 2、4、5 这几个编号中,编号 5 是第 3 大的,所以 ans[3]=5。

(4) pre[2]=1,在剩下的 2、4 这两个编号中,编号 4 是第 2 大的,所以 ans[2]=4。

(5) pre[1]=0,剩下的编号 2,ans[1]=2。

概括以上步骤,在每一步,剩下的编号中第 pre[n]+1 大的编号就是 ans[n]。

用暴力的方法,从 pre[]末尾往前计算,每处理一头牛后,需要把剩下的牛重新排名,重新排名的计算时间是 $O(n)$;在重新排名时,可以顺便做下一次的查找,所以不需要另外算查找的时间。一共有 n 头牛,总复杂度是 $O(n^2)$。本题的数据规模 n 不大,只有 8000,用暴力的方法也能通过。

在下面的代码中，pre[]是输入的 1～n 个数字，例如{0 1 2 1 0}；ans[]是答案，例如{2 4 5 3 1}；num[]记录被处理过的数字，被处理后的数字置为－1。例如 num[]的初始值是{1 2 3 4 5}，得到 ans[5]＝1 后，num[]更新为{－1 2 3 4 5}，这里用－1 表示 1 这个数字已经用过了。

解题的关键是，在剩下的编号中，第 pre[n]＋1 个数字就是 ans[n]。

下面是代码。

poj 2182 的暴力法代码

```
#include <stdio.h>
const int Max = 8005;
int main(){
    int n, i, j, k;
    int pre[Max], ans[Max], num[Max];       //数组的第 0 个都不用,从第 1 个开始用
    scanf("%d", &n);
    pre[1] = 0;
    for(i=1; i<=n; i++)        num[i] = i;
    for(i = 2; i<=n; i++)     scanf("%d", &pre[i]);
    for(i = n; i>=1; i--) {             //从后往前处理数列
        k=0;
        for(j=1; j<=n; j++)             //查找 num[]中未处理的第 pre[i]+1 大的数
            if(num[j] != -1) {
                k++;
                if(k == pre[i]+1) {     //找到了
                      ans[i] = num[j];  //num[]中剩下的第 pre[i]+1 个数就是 ans[i]
                    num[j] = -1;
                    break;
                }
            }
    }
    for(i = 1; i<=n; i++)   printf("%d\n", ans[i]);
    return 0;
}
```

当 n 更大时，$O(n^2)$会 TLE，必须用更优的算法。问题的关键是，如何高效地对剩下的牛重新排名。

这里引入高级数据结构“线段树”，可以在 $O(\log_2 n)$的时间内完成一次重新排名。下面说明其要点。

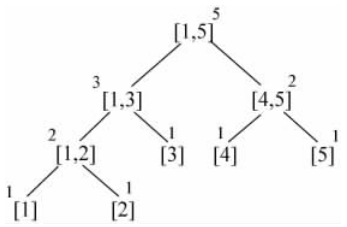


图 5.23　初始线段树

1. 用二叉树建立线段树

用二叉树的方法，把牛分成不同的组。在图 5.23 中，叶子结点内的数字是牛的编号，其他结点是牛的编号范围。例如根结点，包含 5 头牛，它的左子结点有 3 头牛，右子结点有两头牛。

2. 存储空间

如果牛有 n 头，这个二叉树的结点总数在编程时为

$4n$。请读者自己思考为什么是 $4n$(在后面的程序注释中有答案)。

3. 查询和更新

(1) 第 1 次处理 pre[5]=0,即找对应的第 1 头牛,如图 5.24(a)所示。步骤是从根结点开始,逐步找到左下角,即编号为 1 的结点,得到 ans[5]=1。在这个过程中,更新经过的每个结点,即把这个结点剩下的牛的数量减一。一共需要更新 4 个结点。左下角结点已经减到 0,表示后面的计算需要排除它。

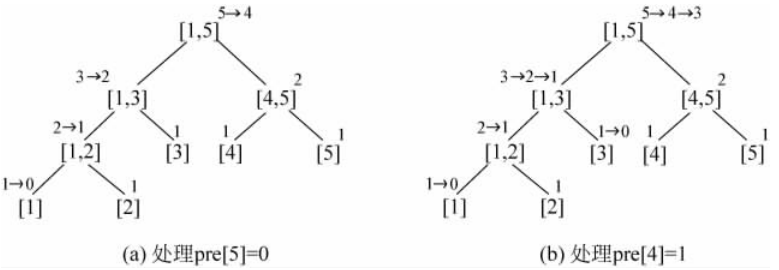


图 5.24　线段树的查询和更新

(2) 第 2 次处理 pre[4]=1,即找剩下的第 2 头牛,如图 5.24(b)所示。步骤是从根结点开始,逐步找到左边第 3 个结点,得到 ans[5]=3。更新经过的每个结点,一共更新 3 个结点。

(3) 依次处理,直到结束。

4. 复杂度

每次处理,从二叉树的根结点开始到最下一层,最多需要更新 $\log_2 4n$ 个结点,复杂度是 $O(\log_2 n)$;一共有 n 头牛需要处理,总复杂度是 $O(n\log_2 n)$。在暴力法中,每次需要查询和更新 n 个序列中的每个数,复杂度为 $O(n)$。线段树把 n 个数按二叉树进行分组,每次更新有关的结点时,这个结点下面的所有子结点都隐含被更新了,从而大大地减少了处理次数。

下面给出 poj 2182 的线段树代码。

poj 2182“用结构体实现线段树”

```
#include <stdio.h>
using namespace std;
const int Max = 10000;
struct{
    int l, r, len;                    //用 len 存储这个区间的数字个数,即这个结点下牛的数量
}tree[4 * Max];                       //这里是 4 倍,因为线段树的空间需要
int pre[Max], ans[Max];
void BuildTree(int left, int right, int u){ //建树
    tree[u].l = left;
    tree[u].r = right;
    tree[u].len = right - left + 1;       //更新结点 u 的值
     if(left == right)
         return;
    BuildTree(left, (left + right)>> 1, u << 1);              //递归左子树
    BuildTree(((left + right)>> 1) + 1, right, (u << 1) + 1);  //递归右子树
```

```
}
int query(int u, int num){                    //查询+维护,所求值为当前区间中左起第 num 个元素
    tree[u].len --;                           //对访问到的区间维护 len,即把这个结点上牛的数量减一
    if(tree[u].l == tree[u].r)
        return tree[u].l;
//情况 1:左子区间内牛的个数不够,则查询右子区间中左起第 num - tree[u<<1].len 个元素
    if(tree[u<<1].len < num)
        return query((u<<1)+1, num - tree[u<<1].len);
//情况 2:左子区间内牛的个数足够,依旧查询左子区间中左起第 num 个元素
    if(tree[u<<1].len >= num)
        return query(u<<1, num);
}
int main(){
    int n, i;
    scanf("%d", &n);
    pre[1] = 0;
    for(i = 2; i <= n; i++)
        scanf("%d", &pre[i]);
    BuildTree(1, n, 1);
    for(i = n; i >= 1; i --)                  //从后往前推断出每次最后一个数字
        ans[i] = query(1, pre[i]+1);
    for(i = 1; i <= n; i++)
        printf("%d\n", ans[i]);
    return 0;
}
```

5. 用完全二叉树实现线段树

在上面的例子中,线段树是一棵普通的二叉树,操作起来比较麻烦。其实可以用完全二叉树的结构来实现,编程更加简单,如图 5.25 所示。

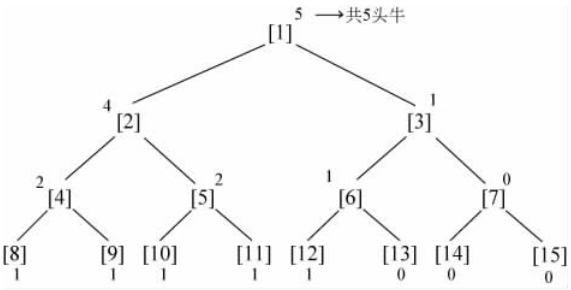


图 5.25 用完全二叉树建线段树

该图中的最后一行是牛的编号,例如[8]对应 1 号牛,[9]对应 2 号牛,等等。一共有 5 头牛。

在使用完全二叉树时,最后一层会存在“空叶子”。同样给空叶子按顺序编号,在遍历线段树时根据判断条件跳过这些“空叶子”就好了。用完全二叉树的方式存储线段树能提高插入线段和搜索时的效率。父结点 p 的左、右子结点分别是 $p*2$、$p*2+1$,用这样的索引方式检索 p 的左、右子树比用指针快。

poj 2182“用完全二叉树实现线段树”

```
#include <stdio.h>
#include <math.h>
const int Max = 10000;
int pre[Max] = {0}, tree[4 * Max] = {0},ans[Max] = {0};
//tree 是用数组实现的满二叉树.从图 5.25 可以知道,需要 4 倍大的空间
void BuildTree(int n,int last_left){        //用完全二叉树建一个线段树
    int i;
    for(i = last_left;i < last_left + n;i++)
                    //给二叉树的最后一行赋值,左边 n 个结点是 n 头牛
        tree[i] = 1;
    while(last_left != 1) {      //从二叉树的最后一行倒推到根结点,根结点的值是牛的总数
        for(i = last_left/2; i < last_left; i++)
            tree[i] = tree[i * 2] + tree[i * 2 + 1];
        last_left = last_left/2;
    }
}
int query(int u, int num,int last_left){
    //查询 + 维护,关键的一点是所求值为当前区间中左起第 num 个元素
    tree[u] -- ;                            //对访问到的区间维护剩下的牛的个数
    if(tree[u] == 0 && u >= last_left)
        return u;
//情况 1:左子区间的数字个数不够,则查询右子区间中左起第 num - tree[u<<1]个元素
    if(tree[u<<1] < num)
        return query((u<<1) + 1, num - tree[u<<1], last_left);
//情况 2:左子区间的数字个数足够,依旧查询左子区间中左起第 num 个元素
    if(tree[u<<1] >= num)
        return query(u<<1, num,last_left);
}
int main(){
    int n, last_left,i;
    scanf("%d", &n);
    pre[1] = 0;
    last_left = 1<<(int(log(n)/log(2)) + 1);
//二叉树最后一行的最左边一个.计算方法是找离 n 最近的 2 的指数,例如 3->4, 4->4, 5->8
    for(i = 2; i <= n; i ++)
        scanf("%d", &pre[i]);
    BuildTree(n,last_left);
    for(i = n; i >= 1; i -- )               //从后往前推断出每次最后一个数字
        ans[i] = query(1, pre[i] + 1,last_left) - last_left + 1;
    for(i = 1; i <= n; i ++)
        printf("%d\n", ans[i]);
    return 0;
}
```

5.3.3　离散化

建二叉树是线段树的基本操作,但是二叉树的大小并不是无限制的,例如规模 10 000 000

以上的二叉树会超过允许的存储空间。在竞赛中如果出现结点规模这样大的题目，当然不能在程序中建这么大的二叉树，此时需要用“离散化”这种小技巧来解决。

离散化就是把原有的大二叉树压缩成小二叉树，但是压缩前后子区间的关系不变。

例如一块宣传栏，横向长度的刻度标记为 1 到 10，贴 4 张不同颜色的海报，它们的宽度和宣传栏等宽，长度分别是[1,3]、[2,5]、[3,8]、[3,10]，并且用后者覆盖前者，问最后能看见几种颜色的海报。

离散化步骤如下：

(1) 提取这 4 张海报的 8 个端点：1 3 2 5 3 8 3 10

(2) 排序并且删除相同的端点，得到：1 2 3 5 8 10

(3) 把原线段的 8 个端点映射到新的线段上：

1	2	3	5	8	10
↓	↓	↓	↓	↓	↓
1	2	3	4	5	6

新的 4 个海报为[1,3]、[2,4]、[3,5]、[3,6]，覆盖关系没有改变。新的宣传栏长度是 1 到 6，即宣传栏的长度从 10 压缩到 6。

离散化的压缩比是很可观的。例如原线段树的区间长度是 10 000 000，而其中真正用到的子区间是 100 000，那么子区间的端点最多有 $2\times100\,000$ 个。经过离散化压缩后，新的线段树区间是 200 000，压缩率是 $200\,000/10\,000\,000=2\%$。

【习题】

poj 2528，题目中宣传栏的长度是 10 000 000。

5.3.4 区间修改

上面的例子都是只修改线段树上的某个点。区间修改是更复杂的问题。给定 n 个元素 $\{a_1,a_2,\cdots,a_n\}$，进行以下操作：

加：给定 $i,j\leqslant n$，把 $\{a_i,\cdots,a_j\}$ 区间内的值全部加 v。

查询：给定 $L,R\leqslant n$，计算 $\{a_L,\cdots,a_R\}$ 的区间和。

下面以 poj 3468 为例来讲解区间修改问题。

poj 3468 “A Simple Problem with Integers”

给出 N 个数，进行 Q 个操作，$1\leqslant N,Q\leqslant100\,000$。有两种操作：

“C a b c”，对区间[a,b]的每个数字加 c；

“Q a b”，查询区间[a,b]的数字和。

输入：N，Q，以及 N 个数字，Q 个操作；

输出：对每个查询操作，输出结果。

如果用暴力方法，直接对这 n 个数进行操作，那么每个 C 操作和 Q 操作都是 $O(n)$ 的，一共有 Q 次操作，总复杂度是 $O(n^2)$。

如果用前面的修改线段树点的方法，在做 C 操作时，对区间里的数一个一个进行修改，

一个数的修改是 $O(\log_2 n)$ 的，区间修改合起来是 $O(n\log_2 n)$，Q 次操作的总复杂度是 $O(n^2\log_2 n)$，比暴力法还要差。

lazy-tag 方法。此时可以采用一种“懒惰(lazy)”的做法。当修改的是一个整块区间时，只对这个线段区间进行整体上的修改，其内部每个元素的内容先不做修改，只有当这部分线段的一致性被破坏时才把变化值传递给子区间。那么，每次区间修改的复杂度是 $O(\log_2 n)$，一共有 Q 次操作，总复杂度是 $O(n\log_2 n)$。做 lazy 操作的子区间，需要记录状态(tag)，在下面的代码中用 add[]实现。


视频讲解

下面描述具体步骤。

(1) 初始化时建树。以区间[1，10]为例建树，图 5.26 所示为结果。在最后的叶子上是 1～10 这 10 个数字。图中最底层有很多叶子是空的。每个结点右上角的数字是以它为根结点的这棵子树的区间和。

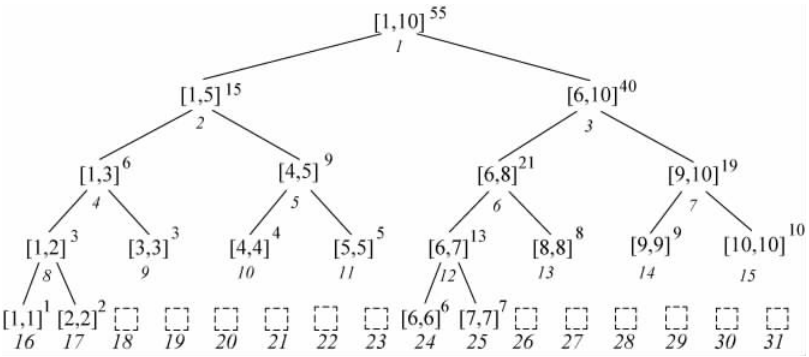


图 5.26　初始化建树

(2) “C a b c”操作。例如“C *3 6 3*”，在[3，6]区间内，把每个元素加 3。从根结点开始，用递归在子树中找区间[3，6]，有两种情况：[3，6]与子区间交错、[3，6]包含子区间。例如子区间[1，5]和[6，10]都与[3，6]交错，需要继续深入更底层子区间。在子区间[4，5]，它被[3，6]包含，那么根据 lazy 原理，把这个子区间进行整体修改，不继续深入，它下一层的[4，4]和[5，5]的区间和不用修改。图 5.27 所示为结果。部分结点的区间和发生了改变，见右上角。

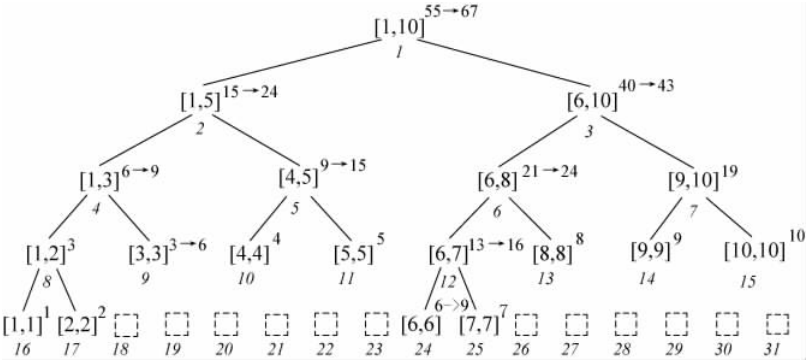


图 5.27　区间求和

(3)“Q a b”。同样可以利用 lazy 原理,当某个子区间包含在被查询的区间内时,直接返回这个子区间的区间和,不用继续深入。

下面是 poj 3468 的程序。build()函数建树,建树的结果见图 5.27;update()函数完成“C a b c”操作,query()函数完成“Q a b”操作。

sum[i]记录结点 i 的区间和,在图 5.27 中是结点右上角的数字。

add[i]是 tag,它记录结点 i 是否用到 lazy 原理,其值是“C a b c”中的 c;如果做了多次 lazy,add[i]可以累加。一旦结点 i 在某次“C a b c”中被深入,破坏了 lazy,就把 add[i]归零,push_down()函数完成这一任务。

```
#include <stdio.h>
using namespace std;
const int MAXN = 1e5 + 10;
long long sum[MAXN << 2], add[MAXN << 2];        //4 倍空间
void push_up(int rt){                            //向上更新,通过当前结点 rt 把值递归到父结点
    sum[rt] = sum[rt << 1] + sum[rt << 1 | 1];
}
void push_down(int rt, int m){                   //更新 rt 的子结点
    if(add[rt]){
        add[rt << 1] += add[rt];
        add[rt << 1 | 1] += add[rt];
        sum[rt << 1] += (m - (m >> 1)) * add[rt];
        sum[rt << 1 | 1] += (m >> 1) * add[rt];
        add[rt] = 0;                             //取消本层标记
    }
}
#define lson l, mid, rt << 1
#define rson mid + 1, r, rt << 1 | 1
void build(int l, int r, int rt){                //用满二叉树建树
    add[rt] = 0;
    if(l == r){                                  //叶子结点,赋值
        scanf("%lld", &sum[rt]);
        return;
    }
    int mid = (l + r) >> 1;
    build(lson);
    build(rson);
    push_up(rt);                                 //向上更新区间和
}
void update(int a, int b, long long c, int l, int r, int rt){     //区间更新
    if(a <= l && b >= r){
        sum[rt] += (r - l + 1) * c;
        add[rt] += c;
        return;
    }
    push_down(rt, r - l + 1);                    //向下更新
    int mid = (l + r) >> 1;
    if(a <= mid) update(a, b, c, lson);          //分成两半,继续深入
    if(b > mid) update(a, b, c, rson);
    push_up(rt);                                 //向上更新
}
```

```
long long query(int a, int b, int l, int r, int rt){    //区间求和
    if(a <= l && b >= r) return sum[rt];                //满足 lazy,直接返回值
    push_down(rt, r - l + 1);                           //向下更新
    int mid = (l + r) >> 1;
    long long ans = 0;
    if(a <= mid) ans += query(a, b, lson);
    if(b > mid) ans += query(a, b, rson);
    return ans;
}
int main(void){
    int n, m;
    scanf("%d%d", &n, &m);
    build(1, n, 1);
    while(m--){
        char str[2];
        int a, b; long long c;
        scanf("%s", str);
        if(str[0] == 'C'){
            scanf("%d%d%lld", &a, &b, &c);
            update(a, b, c, 1, n, 1);
        }else{
            scanf("%d%d", &a, &b);
            printf("%lld\n", query(a, b, 1, n, 1));
        }
    }
}
```

5.3.5　线段树习题

简单题：hdu 1166/1394/1698/1754/2795；

poj 1195/2182/2299/2828/2352/2750/2886/2777/3264/3468。

中等题：hdu 1540/1823/4027/5869；

poj 2155/2528/2823/3225。

综合题：hdu 1255/1542/3642/3974/4578/4614/4718/5756/4441。

5.4　树状数组

树状数组(Binary Indexed Tree,BIT)是一种利用数的二进制特征进行检索的树状结构。树状数组是一种奇妙的数据结构,不仅非常高效,而且代码极其简洁。

1. 树状数组的概念

从下面这个例子引导出树状数组的概念。

长度为 n 的数列$\{a_1,a_2,\cdots,a_n\}$,进行以下操作。

(1) 修改元素 add(k,x)：把 a_k 加上 x。

(2) 求和 sum(x)：$x \leqslant n$,sum$=a_1+a_2+\cdots+a_x$。那么,区间和 $a_i+\cdots+a_j=$sum(j)$-$sum($i-1$)。

这个程序很好写，用循环加或者前缀和，复杂度是 $O(n)$。然而，如果 n 很大，这样做的效率会非常低。读者可以用前面讲的线段树来实现高效的算法。其实有一种更好的数据结构，即树状数组，不仅效率和线段树一样高，只有 $O(\log_2 n)$，而且代码短得不可思议。先看一看代码：

```
#define lowbit(x)  ((x) & - (x))
void add(int x, int d) {                    //更新数组 tree[]。ax = ax + d,修改和 ax 有关的 tree[]
    while(x <= n) {
        tree[x] += d;
        x += lowbit(x);
    }
}
int sum(int x) {                            //求和:sum = a1 + a2 + … + ax
    int sum = 0;
    while(x > 0){
        sum += tree[x];
        x -= lowbit(x);
    }
    return sum;
}
```

add()和 sum()的复杂度都是 $O(\log_2 n)$。

上述代码的使用方法如下：

(1) 初始化，add()。先清空数组 tree[]，然后读取 $a_1, a_2, \cdots, a_n$，用 add()逐一处理这 n 个数，得到 tree[]数组。在程序中并不需要定义数组 a[]，因为它隐含在 tree[]中。

(2) 求和，sum()。计算 sum $= a_1 + a_2 + \cdots + a_x$，即执行 sum()。求和是基于数组 tree[]的。

(3) 如果需要修改元素，执行 add()，即修改数组 tree[]。

下面详细说明上述操作的原理。

2. lowbit()操作


视频讲解

从代码中可以看出，其核心是一个神奇的 lowbit(x)操作。lowbit(x) $= x$ & $-x$，功能是找到 ***x* 的二进制数的最后一个 1**。其原理是利用负数的补码表示，补码是原码取反加一。例如 $x = 6 = 00000110_2$，$-x = x_{补} = 11111010_2$，那么 lowbit(x) $= x$ & $-x = 10_2 = 2$。

1～9 的 lowbit()结果如表 5.1 所示。

表 5.1　1～9 的 lowbit()结果

x	1	2	3	4	5	6	7	8	9
x 的二进制	1	10	11	100	101	110	111	1000	1001
lowbit(x)	1	2	1	4	1	2	1	8	1
tree[x]数组	tree[1] $= a_1$	tree[2] $= a_1 + a_2$	tree[3] $= a_3$	tree[4] $= a_1 + a_2 + a_3 + a_4$	tree[5] $= a_5$	tree[6] $= a_5 + a_6$	tree[7] $= a_7$	tree[8] $= a_1 + a_2 + \cdots + a_8$	tree[9] $= a_9$

lowbit(x)有什么用呢？从 lowbit(x)引出一个 tree[]数组，所有的计算都围绕 tree[]进行。

令 m=lowbit(x)，定义 tree[x]的值，是把 a_x 和它前面共 m 个数相加的结果，如表 5.1 所示。例如 lowbit(6)=2，tree[6]=a_5+ a_6。

图 5.28 中的横线重新描述了这个关系，横线中的黑色表示 tree[x]，它等于横线上元素相加的和。

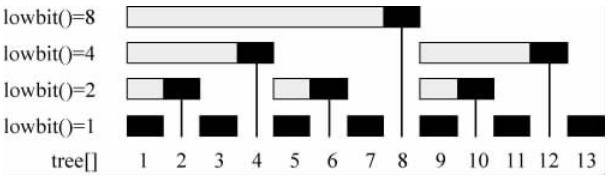


图 5.28　lowbit()计算

求和计算以及 tree[]数组的更新都可以通过 lowbit()完成。

1）求和计算 sum=a_1+a_2+…+a_x

可以借助 tree[]数组求 sum，例如：

sum(8)=tree[8]

sum[7]=tree[7]+tree[6]+tree[4]

sum[9]=tree[9]+tree[8]

然而，如何得到上面的关系呢？

很容易观察到，在计算 sum 时，对 tree[]的查找可以通过 lowbit(x)实现。例如 sum[7]=tree[7]+tree[6]+tree[4]。

首先从 7 开始，加上 tree[7]；

然后 7－lowbit(7)=6，加上 tree[6]；

接着 6－lowbit(6)=4，加上 tree[4]；

最后 4－lowbit(4)=0，结束。

编程细节见前面的求和函数 sum()，复杂度是 $O(\log_2 n)$。

2）tree[]数组的更新

更改 a_x，那么和它相关的 tree[]都会变化。例如改变 a_3，那么 tree[3]、tree[4]、tree[8]等都会改变。同样，这个计算也利用了 lowbit(x)。

首先更改 tree[3]；

然后 3+lowbit(3)=4，更改 tree[4]；

接着 4+lowbit(4)=8，更改 tree[8]；

继续，直到最后的 tree[n]。

编程细节见函数 add()，复杂度也是 $O(\log_2 n)$。add()函数也用于 tree[]的初始化过程：tree[]初始化为 0，然后用 add()逐一处理 a_1，a_2，…，a_n。

3. 例题

这里仍然以 poj 2182 为例，用树状数组实现。该题用树状数组更容易理解。

其中的关键点如下：

(1) 在 n 个位置上,每个位置有一头牛,即 $a_1=a_2=\cdots=a_n=1$。不过,在程序中并不需要直接定义和使用数组 $a[]$。

(2) tree[]数组的初始化。这个题目比较特殊,不需要用 add()初始化,因为 lowbit(i)就是 tree[i]。

(3) 程序所做的,就是对每个 pre[i]+1,用 findpos()找出 sum(x)=pre[i]+1 所对应的 x,就是第 x 头牛。在找到第 x 头牛之后,令 $a_x=0$,方法是用 add()更新数组 tree[],即执行 add(x, −1)。

下面的程序完全套用了上面提到的树状数组的模板。

poj 2182“树状数组”

```
#include <stdio.h>
#include <string.h>
const int Max = 10000;
int tree[Max], pre[Max], ans[Max];
int n;
#define lowbit(x)  ((x) & - (x))
void add(int x, int d){
    while(x <= n) {
        tree[x] += d;
          x += lowbit(x);
    }
}
int sum(int x){
    int sum = 0;
    while(x > 0) {
        sum += tree[x];
          x -= lowbit(x);
    }
    return sum;
}
int findpos(int x){                    //寻找 sum(x) = pre[i] + 1 所对应的 x,就是第 x 头牛
    int l = 1, r = n;
    while(l < r) {
        int mid = (l + r) >> 1;
        if(sum(mid) < x)
            l = mid + 1;
        else
            r = mid;
    }
    return l;
}
int main(){
    scanf("%d",&n);
    pre[1] = 0;
    for(int i = 2; i <= n; i++)
        scanf("%d",&pre[i]);
    for(int i = 1; i <= n; i++)        //初始化 tree[]数组
          //注意这个题目比较特殊,不需要用 add()初始化,因为 lowbit(i)就是 tree[i]
```

```
        tree[i] = lowbit(i);
    for(int i = n; i > 0; i -- )  {
        int x = findpos(pre[i] + 1);
        add(x, -1);                    //更新 tree[]数组
        ans[i] = x;
    }
    for(int i = 1; i <= n; i++)
        printf("%d\n", ans[i]);
    return 0;
}
```

4. 线段树和树状数组的对比

两者的复杂度同级，但是树状数组的常数明显优于线段树，编程复杂度也远远小于线段树。

线段树的适用范围大于树状数组，凡是可以使用树状数组解决的问题，使用线段树一定可以解决。树状数组的优点是编程非常简洁，使用 lowbit()可以在很短的几步操作中完成核心操作，代码效率远远高于线段树。

【习题】

简单题：poj 2299/2352/1195/2481/2029。

中等题：poj 2155/3321/1990；

hdu 3015/2430/2852。

难题：poj 2464，uva 11610。

5.5 小　　结

本章介绍了几个竞赛中常用的数据结构，限于篇幅，还有一些常用的数据结构没讲，例如堆、Hash、动态树 LCT 等。关于字符串的数据结构，在第 9 章中讲解；关于图的数据结构，在第 10 章中讲解。

高级数据结构是算法竞赛中比较难的内容，不仅本身的概念难以掌握，而且在具体的题目中需要根据情况灵活修改，以至于逻辑复杂、代码冗长。