后缀平衡树

算法简介&算法实现

前置芝士：平衡树 这里用 $Treap$

$Treap=BST+Heap$

$Heap$ 的结构是一定的 BST 按照同一种规则建的树的中序遍历也是一定的。

val 遵循 BST rnd 遵循 $Heap$ 所以只要确定根节点 $Treap$ 的结构就是确定的，根据 rnd 取最小作为根节点，这样左右不管怎么分所有点关键值都大于这个值，再把所有 val 小于根节点的点选到左侧，其余点选到右侧，再找到这些点 rnd 最小的值作为根节点的左右儿子，这样找下去这棵树的结构唯一确定。

当权值 val 不可控时，若关键值 rnd 为随机数，则这棵树的期望深度为 log_{n} 一颗平衡树中某一个点及其子树组成的树仍是平衡树。

好 复习完毕 开始后缀平衡树~

后缀平衡树可以维护一个字符串的所有后缀，仿照平衡树，后缀平衡树对一个字符串的所有后缀进行排序。

支持动态插入或删除一个后缀（其实是在字符串前插入或删除一个字符，因为在后面的话不同后缀在末尾加同一个字符之后排序会改变）

于是我们开始魔改平衡树

我们把一个节点的编号设为后缀开始的位置，比如结点 1 代表从 1 开始的后缀 val 的问题我们可以先用一个比较函数 cmp 代替（俗称耍赖）。

```
inline void zig(int &p,long long l,long long r) {//顺时针
    int q=trp[p].lson;
    trp[p].lson=trp[q].rson;
    trp[q].rson=p;
    pushup(p);
    pushup(q);
    get_val(q,l,r);
    p=q;
}
inline void zag(int &p,long long l,long long r) {//逆时针
    int q=trp[p].rson;
    trp[p].rson=trp[q].lson;
    trp[q].lson=p;
    pushup(p);
    pushup(q);
    get_val(q,l,r);
    p=q;
}

inline void ins(int &now,int pos,long long l,long long r) {
    if(!now) {
```

```
        trp[pos].lson=trp[pos].rson=0;
        trp[pos].val=l+r>>1;
        trp[pos].key=rand();
        trp[pos].size=1;
        trp[pos].cnt=1;
        now=pos;
        return;
    }
    int tmp=comp(now,pos);
    if(tmp>0) {
        ins(trp[now].lson,pos,l,trp[now].val-1);
        pushup(now);
        if(trp[trp[now].lson].key<trp[now].key) zig(now,l,r);
    } else if(tmp<0) {
        ins(trp[now].rson,pos,trp[now].val+1,r);
        pushup(now);
        if(trp[trp[now].rson].key<trp[now].key) zag(now,l,r);
    } else {
        trp[now].cnt++;
        pushup(now);
    }
}
```

那么第二个问题来了，这个坑得填上 cmp 函数到底应该怎么写？注意到我们对于两个后缀，除去第一个字母的两个新后缀是比较过大小的（先假定是这样 类似于归纳法？），我们先比较两个首字符的大小，如果分出胜负，后面的自然不用比了，如果两个字符一样，那么直接比较后面的新后缀就好了，这样我们就可以比较出大小了。至于用什么作为比较的依据，我们采用类似于线段树的做法，传两个参数 l 和 r 点的权值就是 $(l+r)/2$ 左儿子就是 l 和 $(l+r)/2-1$ 右儿子就是 $(l+r)/2+1$ 和 r 每次旋转之后再重新取这个值，注意将一个子树全部重新取完的时间是 $O(logn)$ 的，而插入之类的操作本身也是这个复杂度，所以省去常数复杂度不变。这里必要时刻为了不爆精度，最好使用 $double$ 类型。

```
inline int comp(int x,int y) {
    if(s[x]>s[y]||s[x]==s[y]&&trp[x+1].val>trp[y+1].val) return 1;
    else if(s[x]==s[y]&&trp[x+1].val==trp[y+1].val) return 0;
    else return -1;
}

inline void get_val(int now,long long l,long long r) {
    trp[now].val=l+r>>1;
    if(trp[now].lson) get_val(trp[now].lson,l,trp[now].val-1);
    if(trp[now].rson) get_val(trp[now].rson,trp[now].val+1,r);
}
```

至于删除操作，我们有可能会把一个子树的根节点给删掉，这时我们可以借用 $FHQ Treap$ 的合并做法，树高期望为 $O(logn)$ 不变，所以之后重构就可以了。

```
inline void pushup(int pos) {
    int lson=trp[pos].lson,rson=trp[pos].rson;
    trp[pos].size=trp[lson].size+trp[rson].size+trp[pos].cnt;
}
```

```
int merge(int x,int y){
    if(!x||!y) return x+y;
    if(trp[x].key<trp[y].key){
        trp[x].rson=merge(trp[x].rson,y);
        pushup(x);
        return x;
    }
    else{
        trp[y].lson=merge(x,trp[y].lson);
        pushup(y);
        return y;
    }
}
inline void del(int &now,int pos,long long l,long long r){
    if(!now) return;
    int tmp = comp(now,pos);
    if(!tmp){
        if(trp[now].cnt>1) trp[now].cnt--;
        else{
            now=merge(trp[now].lson,trp[now].rson);
            if(now) get_val(now,l,r);
        }
        return;
    }
    else if(tmp>0) del(trp[now].lson,pos,l,trp[now].val-1);
    else del(trp[now].rson,pos,trp[now].val+1,r);
    pushup(now);
}
```

至此后缀平衡树的插入操作可以实现。

后缀平衡树其实是可以实现和后缀数组有关的东西，但是可能是我人傻常数大，写的比后缀数组要慢亿点点。比如求 $sa, rk, height$ 这三个数组。sa 表示排名为 i 的后缀起点。rk 表示后缀七点为 i 的排名。$height$ 表示排名为 $i-1$ 和排名为 i 的后缀的 LCP 长度。dfs 一下解决前两个数组。第三个数组因为本身就是可以通过 sa 和 rk 求出来的。所以就搞定啦~

```
int cnt,sa[N],rk[N],height[N];
inline void dfs(int now){
    if(!now) return;
    dfs(trp[now].lson);
    sa[++cnt]=now;
    rk[now]=cnt;
    dfs(trp[now].rson);
}
inline void get_height() {
    int k=0;
    for (int i=1; i<=len; ++i) {
        if (rk[i]==1) continue;
        if (k) --k;
        int j=sa[rk[i]-1];
        while (j+k<=n && i+k<=n && s[i+k]==s[j+k]) ++k;
```

```
            height[rk[i]]=k;
        }
    }
```

后缀平衡树单次插入需要 $O(logn)$ 的时间，有 n 个后缀，所以时间复杂度是 $O(nlogn)$ 空间复杂度因为结点是 n 个，故为 $O(n)$

相对于后缀数组有额外的添加字符的操作，但是代码长时间慢。

相对于后缀自动机有额外的删除字符的功能。后缀自动机需要考虑字符集大小，后缀平衡树不需要。

代码练习

1.https://www.luogu.com.cn/problem/P5353

给定一个以 n 个节点的树，保证对于 2 到 n 的每个节点，其父亲的编号均小于自己的编号。

每个节点上有一个字符，一个点代表的字符串是从这个点到根的简单路径上经过的所有的字符连起来组成的字符串。

需要我们将这些字符串按照字典序排序，对于一样的字符串，优先比较他们父亲代表的字符串的大小，还是一样就按照他们的编号大小进行排序。

输入包含节点数 n 以及从 2 到 n 每个结点的父亲编号，最后输入一个字符串，第 i 个字符代表编号为 i节点上的字符。

输出：按照字符串排名从小到大输出节点编号

注意到我们只是说后缀平衡树可以满足字符的插入和删除，但是并不是只能解决一个字符串从后向前的插入问题，对于这种树上排序，后缀平衡树大有用武之地。

注意到因为父亲编号必小于自己的编号，我们这次需要正向插入，因为儿子节点是一个字符串的首字符相当于刚才讲的插入操作的不会改变已有排序顺序的字符，所以要正向插入。我们还需要改变比较规则，首字母一样时比较父亲的 val 值，再一样返回下标的比较结果。

```
inline int comp(int x,int y) {
    if(s[x]>s[y]||s[x]==s[y]&&trp[fa[x]].val>trp[fa[y]].val) return 1;
    else if(s[x]==s[y]&&trp[fa[x]].val==trp[fa[x]].val&&x>y) return 1;
    //else if(s[x]==s[y]&&trp[fa[x]].val==trp[fa[x]].val&&x==y) return 0;
    else return -1;
}

int main() {
    scanf("%d",&n);
    for(int i=2; i<=n; i++) scanf("%d",&fa[i]);
    scanf("%s",s+1);
    for(int i=1; i<=n; i++) ins(root,i,1,INF);
    dfs(root);
    for(int i=1; i<=n; i++) printf("%d ",sa[i]);
    return 0;
}
```