

左偏树

算法简介

一种可并堆，支持 $O(\log n)$ 的 push 、 pop 、 merge 操作。

算法思想

定义外节点为左儿子或右儿子为空的节点， dist_i 表示节点 i 到其子树中最近外节点的距离。

规定空节点的 dist 为 -1 。左偏树的定义为对每个节点 u 均满足 $\text{dist}_{\text{lson}} \geq \text{dist}_{\text{rson}}$ 的堆。

左偏树有如下性质

1. $\text{dist}_u = \text{dist}_{\text{rson}} + 1$
2. $\text{dist}_u \sim O(\log \text{sz}_u)$

关于性质一，有

$\text{dist}_u = \max(\text{dist}_{\text{lson}}, \text{dist}_{\text{rson}}) + 1 = \text{dist}_{\text{rson}} + 1$ 证毕。

关于性质二，有 dist_u 表示节点 u 到其子树中最近外节点的距离，所以有节点 u 的子树的前 dist_u 层均为非外节点。

所以可将前 dist_u 层视为满二叉树，第 $\text{dist}_u + 1$ 层至少有一个外节点，所以有 $\text{sz}_u \geq 2^{\text{dist}_u}$ 证毕。

考虑左偏树的合并操作，类似 fhq treap 的 merge 操作，根据优先级(这里是点权)不断合并两棵树，遇到空结点返回。

不同之处在于合并两棵左偏树时对跳左节点还是右节点没有限制，所以强制每次跳右节点，使得 dist_u 单调递减。

这样，时间复杂度为 $O(\text{dist}_u) = O(\log n)$

左偏树的另一个核心操作为寻根，事实上寻根操作可以利用路径压缩的并查集优化时间复杂度到 $O(\log n)$ 但需要注意一些细节，详细见代码。

代码模板

[洛谷p3377](#)

左偏树的 pop 操作类似 fhq treap 直接合并被删除节点的左右儿子，但是删除祖先节点会导致并查集出错。

具体解决方案为 $\text{root}[p] = \text{root}[\text{node}[p].\text{ch}[0]] = \text{root}[\text{node}[p].\text{ch}[1]] = \text{rt}$ ，即把原堆顶元素及其儿子节点的祖先节点指向新根节点。

```
const int MAXN=1e5+5;
struct Left_Heap{
    int root[MAXN];
    bool del[MAXN];
    struct Node{
        int id,val,dis,ch[2];
        bool operator < (const Node &b)const{return
val<b.val||(val==b.val&&id<b.id);}
    }node[MAXN];
    void build(int *a,int n){
        node[0].dis=-1;
        _rep(i,1,n)
        node[i].id=root[i]=i,node[i].val=a[i];
    }
    int find_root(int x){return x==root[x]?x:root[x]=find_root(root[x]);}
    void merge(int &k,int k1,int k2){
        if(!k1||!k2)return k=k1|k2,void();
        if(node[k2]<node[k1])swap(k1,k2);
        merge(node[k=k1].ch[1],node[k1].ch[1],k2);
        if(node[node[k].ch[0]].dis<node[node[k].ch[1]].dis)
        swap(node[k].ch[0],node[k].ch[1]);
        node[k].dis=node[node[k].ch[1]].dis+1;
    }
    void merge(int x,int y){
        if(del[x]||del[y])return;
        int rt,p1=find_root(x),p2=find_root(y);
        if(p1!=p2){
            merge(rt,p1,p2);
            root[p1]=root[p2]=rt;
        }
    }
    int top(int x){
        if(del[x])return -1;
        return node[find_root(x)].val;
    }
    void pop(int x){
        if(del[x])return;
        int rt,p=find_root(x);
        merge(rt,node[p].ch[0],node[p].ch[1]);
        root[p]=root[node[p].ch[0]]=root[node[p].ch[1]]=rt;
        del[p]=true;
    }
}heap;
int a[MAXN];
int main()
{
    int n=read_int(),m=read_int(),opt,x;
    _rep(i,1,n)
    a[i]=read_int();
```

```

    heap.build(a,n);
    while(m--){
        opt=read_int(),x=read_int();
        switch(opt){
            case 1:
                heap.merge(x,read_int());
                break;
            case 2:
                enter(heap.top(x));
                heap.pop(x);
        }
    }
    return 0;
}

```

算法习题

习题一

[洛谷p1456](#)

题意

维护 n 个堆，每个堆一开始只有一个元素 m 个操作。

每次选取两个堆，将堆顶元素减半，然后合并。

题解

修改堆顶元素方法为先合并左右儿子，然后修改堆顶，再次加入。同样要注意并查集的维护。

```

const int MAXN=1e5+5;
struct Left_Heap{
    int root[MAXN];
    struct Node{
        int id,val,dis,ch[2];
        bool operator < (const Node &b) const{return
val>b.val||(val==b.val&&id<b.id);}
    }node[MAXN];
    void build(int *a,int n){
        node[0].dis=-1;
        _rep(i,1,n)
node[i].id=root[i]=i,node[i].val=a[i],node[i].ch[0]=node[i].ch[1]=node[i].d
is=0;
    }
    int find_root(int x){return x==root[x]?x:root[x]=find_root(root[x]);}
}

```

```
void merge(int &k,int k1,int k2){
    if(!k1||!k2)return k=k1|k2,void();
    if(node[k2]<node[k1])swap(k1,k2);
    merge(node[k=k1].ch[1],node[k1].ch[1],k2);
    if(node[node[k].ch[0]].dis<node[node[k].ch[1]].dis)
        swap(node[k].ch[0],node[k].ch[1]);
    node[k].dis=node[node[k].ch[1]].dis+1;
}
int merge(int x,int y){
    int rt,p1=find_root(x),p2=find_root(y);
    if(p1==p2)return -1;
    update(p1);update(p2);
    p1=find_root(x),p2=find_root(y);
    merge(rt,p1,p2);
    root[p1]=root[p2]=rt;
    return node[rt].val;
}
void update(int rt){
    int temp,Root;
    merge(temp,node[rt].ch[0],node[rt].ch[1]);
    node[rt].val>>=1;node[rt].ch[0]=node[rt].ch[1]=node[rt].dis=0;
    merge(Root,temp,rt);
    root[Root]=root[rt]=Root;
}
}heap;
int a[MAXN];
int main()
{
    int n,m,x,y;
    while(~scanf("%d",&n)){
        _rep(i,1,n)
            a[i]=read_int();
        heap.build(a,n);
        m=read_int();
        _rep(i,1,m){
            x=read_int(),y=read_int();
            enter(heap.merge(x,y));
        }
    }
    return 0;
}
```

习题二

[洛谷p4331](#)

题意

给定一个序列 A 要求构造一个严格单增序列 B 满足 $\sum_{i=1}^n |a_i - b_i|$ 取最小值。

题解

首先将 a_i 变为 $a_i - i$ 于是问题转化为构造一个不减序列 B

单独考虑区间 $[x, x]$ 显然答案为 $b_x = a_x$ 对区间 $C_1[x, y], C_2[y+1, z]$ 假设区间 C_1 答案为 c_1 区间 C_2 答案为 c_2

若 $c_1 \leq c_2$ 显然不需要修改, 若 $c_1 > c_2$ 发现取区间 $[x, z]$ 的中位数作为区间 $[x, z]$ 的答案显然最优。

考虑单调栈维护中位数单调不减的区间, 同时左偏堆 (大根堆) 维护中位数。

对左偏堆弹出一半的数即可得到中位数, 可以证明在该算法下该操作不会导致答案错误, 时间复杂度 $O(n \log n)$

```
const int MAXN=1e6+5;
namespace Left_Heap{
    struct Node{
        int val,dis,ch[2];
        bool operator < (const Node &b){return val<b.val;}
    }node[MAXN];
    void init(){node[0].dis=-1;}
    void merge(int &k,int k1,int k2){
        if(!k1||!k2)return k=k1|k2,void();
        if(node[k1]<node[k2])swap(k1,k2);
        merge(node[k=k1].ch[1],node[k1].ch[1],k2);
        if(node[node[k].ch[0]].dis<node[node[k].ch[1]].dis)
            swap(node[k].ch[0],node[k].ch[1]);
        node[k].dis=node[node[k].ch[1]].dis+1;
    }
};
struct Heap{
    int tot,sz,rt;
    int top(){return Left_Heap::node[rt].val;}
    void pop(){
        Left_Heap::merge(rt,Left_Heap::node[rt].ch[0],Left_Heap::node[rt].ch[1]);
        sz--;
    }
    void merge(Heap y){
        Left_Heap::merge(rt,rt,y.rt);
        tot+=y.tot;
        sz+=y.sz;
    }
}heap[MAXN];
int a[MAXN];
```

```
int main()
{
    Left_Heap::init();
    int n=read_int(),top=0;
    _rep(i,1,n){
        Left_Heap::node[i].val=a[i]=read_int()-i;
        heap[++top]=Heap{1,1,i};
        while(top>1&&heap[top-1].top()>heap[top].top()){
            top--;
            heap[top].merge(heap[top+1]);
            while(heap[top].sz>(heap[top].tot+1)/2)
                heap[top].pop();
        }
    }
    int pos=1;
    LL ans=0;
    _rep(i,1,top){
        _for(j,0,heap[i].tot){
            ans+=abs(a[pos]-heap[i].top());
            pos++;
        }
    }
    enter(ans);
    pos=1;
    _rep(i,1,top){
        _for(j,0,heap[i].tot){
            space(heap[i].top()+pos);
            pos++;
        }
    }
    return 0;
}
```

习题三

洛谷p3261

题意

给定一棵以 1 为根的树，每个点拥有属性 (h_i, a_i, v_i) 同时给定 m 个士兵，每个士兵初始时在 s_i 号点，且拥有 c_i 点生命。

当士兵 i 处在点 j 时，如果 $c_i \leq h_j$ 则该士兵在该点死亡。

否则该士兵生命产生变化。具体得，如果 $a_i = 0$ 则士兵生命增加 v_i 如果 $a_i = 1$ 则士兵生命乘以 v_i

如果士兵未死亡，则士兵通过当前结点且继续前往该结点的父节点。

询问每个结点死亡的士兵数和每个士兵最终通过的结点数。数据保证士兵任意时刻生命值不爆
 long long 且如果 $a_i=1$ 则 $v_i>0$

题解

考虑 dfs 遍历城市，左偏树合并子节点士兵信息，每次弹出堆顶所有 h 的结点，同时更新答案。

对于修改操作，类似线段树懒标记处理即可。由于 $a_i=1$ 时保证 $v_i>0$ 所有结点的相对大小不变，所有懒标记处理可以保证正确性。

合并操作总复杂度 $(n \log m)$ 弹出操作总复杂度 $O(m \log m)$ 修改操作总时间复杂度 $O(n)$ 于是总时间复杂度 $O((n+m) \log m)$

```
const int MAXN=3e5+5;
namespace Left_Heap{
    struct Node{
        LL val,lazy[2];
        int idx,dis,ch[2];
        void set(int id,LL v){
            idx=id,val=v;
            lazy[0]=1,lazy[1]=0;
        }
        bool operator < (const Node &b)const{return
val<b.val||(val==b.val&&idx<b.idx);}
    }node[MAXN];
    void init(){node[0].dis=-1;}
    void push_tag(int k,LL tag1,LL tag2){
        if(!k)return;
        node[k].val=node[k].val*tag1+tag2;
        node[k].lazy[0]=node[k].lazy[0]*tag1;
        node[k].lazy[1]=node[k].lazy[1]*tag1+tag2;
    }
    void push_down(int k){
        push_tag(node[k].ch[0],node[k].lazy[0],node[k].lazy[1]);
        push_tag(node[k].ch[1],node[k].lazy[0],node[k].lazy[1]);
        node[k].lazy[0]=1,node[k].lazy[1]=0;
    }
    void merge(int &k,int k1,int k2){
        if(!k1||!k2)return k=k1|k2,void();
        push_down(k1);push_down(k2);
        if(node[k2]<node[k1])swap(k1,k2);
        merge(node[k=k1].ch[1],node[k1].ch[1],k2);
        if(node[node[k].ch[0]].dis<node[node[k].ch[1]].dis)
            swap(node[k].ch[0],node[k].ch[1]);
        node[k].dis=node[node[k].ch[1]].dis+1;
    }
};
struct Heap{
    int sz,rt;
```

```
pair<int,LL> top(){return
make_pair(Left_Heap::node[rt].idx,Left_Heap::node[rt].val);}
void pop(){
    Left_Heap::push_down(rt);
    Left_Heap::merge(rt,Left_Heap::node[rt].ch[0],Left_Heap::node[rt].ch[1]);
    sz--;
}
void push(int idx,LL v){
    Left_Heap::node[idx].set(idx,v);
    Left_Heap::merge(rt,rt,idx);
    sz++;
}
void merge(Heap y){
    Left_Heap::merge(rt,rt,y.rt);
    sz+=y.sz;
}
Heap(){sz=rt=0;}
}heap[MAXN];
struct Edge{
    int to,next;
}edge[MAXN];
int head[MAXN],edge_cnt;
void Insert(int u,int v){
    edge[++edge_cnt]=Edge{v,head[u]};
    head[u]=edge_cnt;
}
int a[MAXN],s[MAXN],d[MAXN],ans1[MAXN],ans2[MAXN];
LL h[MAXN],tag[MAXN];
void dfs(int u){
    for(int i=head[u];i;i=edge[i].next){
        int v=edge[i].to;
        d[v]=d[u]+1;
        dfs(v);
        heap[u].merge(heap[v]);
    }
    while(heap[u].sz&&heap[u].top().second<h[u]){
        ans1[u]++;
        ans2[heap[u].top().first]=d[u];
        heap[u].pop();
    }
    if(heap[u].sz){
        if(a[u])
            Left_Heap::push_tag(heap[u].rt,tag[u],0);
        else
            Left_Heap::push_tag(heap[u].rt,1,tag[u]);
    }
}
int main()
{
    Left_Heap::init();
```

```
int n=read_int(),m=read_int();
_rep(i,1,n)
h[i]=read_LL();
_rep(i,2,n){
    Insert(read_int(),i);
    a[i]=read_int(),tag[i]=read_LL();
}
_rep(i,1,m){
    LL c=read_LL();
    s[i]=read_int();
    heap[s[i]].push(i,c);
}
d[1]=1;
dfs(1);
_rep(i,1,n)
enter(ans1[i]);
_rep(i,1,m)
enter(d[s[i]]-ans2[i]);
return 0;
}
```

From:

<https://wiki.cvbbacm.com/> - CVBB ACM Team

Permanent link:

https://wiki.cvbbacm.com/doku.php?id=2020-2021:teams:legal_string:jxm2001:%E5%B7%A6%E5%81%8F%E6%A0%91

Last update: 2020/10/21 22:53