可以觀察到，薪水≥K的人的分佈會從根慢慢往下「擴散」並「分岔」  
  
把這個過程倒回來，就變成了薪水<K的人慢慢往上「蔓延」並在交叉點「合併」，不會往回跑  
  
假如我們能快速找出每次修改會影響到蔓延體的哪個頂端，並快速算出這個蔓延體頂端的薪水，就能判斷要不要繼續往上蔓延了  
  
可以的！  
可以利用並查集維護每個點沿著蔓延體往上會走到哪裡  
利用懶惰標記可以在蔓延上去時把懶惰標記往上丟，並直接算出該節點的值  
  
每蔓延一格，就把答案減一  
請記得答案是倒著算出來的  
  
時間複雜度：O(Qα(N)) (實際上，我的並查集沒有特別優化到O(α(N))，複雜度為O(logN))  
  
code:

#include<cstdio>

#include<cassert>

#include<utility>

#include<vector>

#include<queue>

using namespace std;

typedef long long LL;

int N,Q,PARENT[300000];

vector<int>ET[300000],TOPU;

LL K,COST[300001],TAG[300001];

int ANS[300000];

pair<int,LL>QUERY[300000];

int DJ[300001];

int Find(const int a){return DJ[a]==a?a:(DJ[a]=Find(DJ[a]));}

int main()

{

freopen("in.txt","r",stdin);

freopen("out.txt","w",stdout);

// freopen("s1.in","r",stdin);

while(scanf("%d%d%lld",&N,&Q,&K)==3)

{

for(int i=0;i<N;i++)ET[i].clear();

for(int i=1,v;i<N;i++)scanf("%d",&v),ET[PARENT[i]=--v].push\_back(i);

PARENT[0]=N;

for(int i=0;i<N;i++)COST[i]=TAG[i]=0LL;

for(int i=0;i<Q;i++)

{

auto &q=QUERY[i];

scanf("%d%lld",&q.first,&q.second),COST[--q.first]+=q.second;

}

if(true)

{

TOPU.clear();

queue<int>q;q.push(0);

while(!q.empty())

{

const int u=q.front();q.pop();

TOPU.push\_back(u);

for(const int nxt:ET[u])q.push(nxt);

}

assert((int)TOPU.size()==N);

}

for(int i=N-1;i>=0;i--)

{

const int u=TOPU[i];

for(const int nxt:ET[u])COST[u]+=COST[nxt];

}

for(int i=0;i<=N;i++)DJ[i]=i;

int ans=N;

for(int i=N-1;i>=0;i--)

{

const int u=TOPU[i];

if(COST[u]<K)DJ[u]=PARENT[u],ans--;

}

for(int i=Q-1;i>=0;i--)

{

ANS[i]=ans;

const auto &q=QUERY[i];

int u=Find(q.first);

TAG[u]+=q.second;

COST[u]-=q.second;

while(u!=N&&COST[u]<K)

{

const int nxt=Find(PARENT[u]);

TAG[nxt]+=TAG[u];

COST[nxt]-=TAG[u];

u=DJ[u]=nxt,ans--;

}

}

for(int i=0;i<Q;i++)printf("%d**\n**",ANS[i]);

}

return 0;

}

可以觀察到，一個人的薪水是他所有下屬的薪水總和  
也就是子樹的總和  
  
因此，我們可以把樹壓平  
簡單來說就是把每個節點都給一個編號，讓每個節點的編號>以它為根的子樹中所有點的編號  
編號1∼N的節點組成了一個序列  
這樣任何的子樹都可以用一段區間來表示了，區間右界即是子樹根節點的編號  
  
如果我們用線段樹來維護這個序列的區間和，那麼我們就可以隨時O(logN)得出任一個人目前的薪水是多少了！  
  
可是總不可能每次加薪完就檢查所有N個人吧  
  
可以發現，每個人的薪水只會增加，不會減少  
所以我們可以二分搜每個人的薪水甚麼時候會≥K  
這樣每一時刻有幾個人的薪水≥K就可以很容易的算出來了  
  
可是要判斷某個人在某個時間點的薪水是否≥K時，需要這個時間點的線段樹  
總不可能N個人讓線段樹重建N次吧  
  
  
  
可以觀察到，很多人會共用同一個時間點的判斷，那就先將這個時間點的線段樹弄出來，然後大家一次判斷完就好啦～  
  
這就是整體二分的概念  
  
時間複雜度：O((N+Q)logQlogN)  
  
code:

#include<cstdio>

#include<cassert>

#include<vector>

#include<utility>

using namespace std;

typedef long long LL;

struct SegTree

{

int N;

LL SUM[300000\*4];

void Build(const int id,const int l,const int r)

{

SUM[id]=0LL;

if(l<r)

{

const int mid=(l+r)/2;

Build(id\*2,l,mid),Build(id\*2+1,mid+1,r);

}

}

void Build(const int \_N){N=\_N;Build(1,0,N-1);}

void Modify(const int id,const int l,const int r,const int loc,const LL &val)

{

SUM[id]+=val;

if(l<r)

{

const int mid=(l+r)/2;

if(loc<=mid)Modify(id\*2,l,mid,loc,val);

else Modify(id\*2+1,mid+1,r,loc,val);

}

}

void Modify(const int loc,const LL &val){Modify(1,0,N-1,loc,val);}

LL Query(const int id,const int l,const int r,const int bl,const int br)

{

if(r<bl||br<l)return 0LL;

if(bl<=l&&r<=br)return SUM[id];

const int mid=(l+r)/2;

return Query(id\*2,l,mid,bl,br)+Query(id\*2+1,mid+1,r,bl,br);

}

LL Query(const int bl,const int br){return Query(1,0,N-1,bl,br);}

}SEG\_TREE;

struct QueryType

{

int u;

LL x;

};

vector<QueryType>QUERY;

pair<int,int>SEG[300000];//not related to SEG\_TREE

vector<int>ET[300000];

void BuildSEG(const int u,int &clock)

{

SEG[u].first=clock;

for(const int nxt:ET[u])BuildSEG(nxt,clock);

SEG[u].second=clock++;

}

int ANS[300001];

LL K;

void Solve(const int l,const int r,const vector<int>&staffs)

{

if(staffs.empty())return;

if(l==r){ANS[r]+=staffs.size();return;}

const int mid=(l+r)/2;

for(int i=l;i<=mid;i++)

{

const QueryType &q=QUERY[i];

SEG\_TREE.Modify(SEG[q.u].second,q.x);

}

vector<int>left\_staffs,right\_staffs;

for(const int s:staffs)

{

if(SEG\_TREE.Query(SEG[s].first,SEG[s].second)<K)right\_staffs.push\_back(s);

else left\_staffs.push\_back(s);

}

Solve(mid+1,r,right\_staffs);

for(int i=mid;i>=l;i--)

{

const QueryType &q=QUERY[i];

SEG\_TREE.Modify(SEG[q.u].second,-q.x);

}

Solve(l,mid,left\_staffs);

vector<int>().swap(left\_staffs);

vector<int>().swap(right\_staffs);

}

int N,Q;

int main()

{

freopen("in.txt","r",stdin);

freopen("out.txt","w",stdout);

// freopen("s2.in","r",stdin);

for(;scanf("%d%d%lld",&N,&Q,&K)==3;)

{

for(int i=0;i<N;i++)ET[i].clear();

for(int i=1,parent;i<N;i++)scanf("%d",&parent),ET[--parent].push\_back(i);

if(true){int clock=0;BuildSEG(0,clock);assert(clock==N);}

QUERY.clear();

for(int i=0;i<Q;i++)

{

static QueryType q;

scanf("%d%lld",&q.u,&q.x),q.u--;

QUERY.push\_back(q);

}

vector<int>staffs;

for(int i=0;i<N;i++)staffs.push\_back(i);

for(int i=0;i<Q;i++)ANS[i]=0;

SEG\_TREE.Build(N);

Solve(0,Q,staffs);

for(int i=1;i<Q;i++)ANS[i]+=ANS[i-1];

for(int i=0;i<Q;i++)printf("%d**\n**",ANS[i]);

}

return 0;

}

可以觀察到，一個人的薪水是他所有下屬的薪水總和  
也就是子樹的總和  
  
因此，我們可以把樹壓平  
簡單來說就是把每個節點都給一個編號，讓每個節點的編號>以它為根的子樹中所有點的編號  
編號1∼N的節點組成了一個序列  
這樣任何的子樹都可以用一段區間來表示了，區間右界即是子樹根節點的編號  
  
如果我們用線段樹來維護這個序列的區間和，那麼我們就可以隨時O(logN)得出任一個人目前的薪水是多少了！  
  
可是總不可能每次加薪完就檢查所有N個人吧  
  
可以發現，每個人的薪水只會增加，不會減少  
所以我們可以二分搜每個人的薪水甚麼時候會≥K  
這樣每一時刻有幾個人的薪水≥K就可以很容易的算出來了  
  
可是要判斷某個人在某個時間點的薪水是否≥K時，需要這個時間點的線段樹  
總不可能N個人讓線段樹重建N次吧  
  
其實，利用持久化線段樹，我們可以用O(NlogQ)的時間和O(NlogQ)的記憶體，同時擁有每一個時間點的Q棵線段樹  
  
時間複雜度：O(NlogQlogN)  
  
code:

#include<cstdio>

#include<cassert>

#include<vector>

#include<utility>

using namespace std;

typedef long long LL;

struct Node

{

Node \*ch[2];

LL sum;

}BUFFER[12000000],\*NEW;

Node \*NewNode(const LL &\_sum)

{

assert(NEW<BUFFER+12000000);

NEW->ch[0]=NEW->ch[1]=NULL;

NEW->sum=\_sum;

return NEW++;

}

Node \*Build(const int l,const int r)

{

Node \*ans=NewNode(0LL);

if(l<r)

{

const int mid=(l+r)/2;

ans->ch[0]=Build(l,mid),ans->ch[1]=Build(mid+1,r);

}

return ans;

}

Node \*Modify(Node \*o,const int l,const int r,const int loc,const LL &val)

{

Node \*ans=NewNode((o->sum)+val);

if(l<r)

{

const int mid=(l+r)/2;

if(loc<=mid)ans->ch[0]=Modify(o->ch[0],l,mid,loc,val),ans->ch[1]=o->ch[1];

else ans->ch[0]=o->ch[0],ans->ch[1]=Modify(o->ch[1],mid+1,r,loc,val);

}

return ans;

}

LL Query(Node \*o,const int l,const int r,const int bl,const int br)

{

if(r<bl||br<l)return 0LL;

if(bl<=l&&r<=br)return o->sum;

const int mid=(l+r)/2;

return Query(o->ch[0],l,mid,bl,br)+Query(o->ch[1],mid+1,r,bl,br);

}

int N,Q;

LL K;

pair<int,int>SEG[300000];

vector<int>ET[300000];

void BuildSEG(const int u,int &clock)

{

SEG[u].first=clock;

for(const int nxt:ET[u])BuildSEG(nxt,clock);

SEG[u].second=clock++;

}

Node \*TREE[300001];

int ANS[300002];

int main()

{

freopen("in.txt","r",stdin);

freopen("out.txt","w",stdout);

// freopen("s2.in","r",stdin);

for(;scanf("%d%d%lld",&N,&Q,&K)==3;)

{

for(int i=0;i<N;i++)ET[i].clear();

NEW=BUFFER;

for(int i=1,parent;i<N;i++)scanf("%d",&parent),ET[--parent].push\_back(i);

if(true){int clock=0;BuildSEG(0,clock);assert(clock==N);}

TREE[0]=Build(0,N-1);

for(int i=1,u;i<=Q;i++)

{

static LL x;

scanf("%d%lld",&u,&x),u--;

TREE[i]=Modify(TREE[i-1],0,N-1,SEG[u].second,x);

}

for(int i=0;i<=Q+1;i++)ANS[i]=0;

for(int i=0;i<N;i++)

{

int l=0,r=Q+1;

while(l<r)

{

const int mid=(l+r)/2;

if(Query(TREE[mid],0,N-1,SEG[i].first,SEG[i].second)<K)l=mid+1;

else r=mid;

}

ANS[r]++;

}

for(int i=1;i<=Q;i++)ANS[i]+=ANS[i-1];

for(int i=1;i<=Q;i++)printf("%d**\n**",ANS[i]);

}

return 0;

}

很明顯，這題可以用樹鏈剖分來做  
  
將樹輕重鏈剖分之後，每條鏈都用一顆線段樹來維護  
當要把u到根之間的路徑全部加上x時，就把路徑上的每棵線段樹做對應的區間修改  
可以證明，在輕重鏈剖分下，遇到的線段樹數量頂多O(logN)  
當要修改任一棵線段樹時，將答案加上「修改前線段樹中≥K的數量−修改後線段樹中≥K的數量」即可  
  
線段樹中≥K的數量怎麼求呢？  
可以發現，在本題，一條鏈中的值越接近根會越大  
因此，只要盡量往線段樹遠離根的方向走，同時保持最大值≥K，走到底的位置就可以轉換成答案  
  
時間複雜度：O(Qlog2N)  
  
code:

#include<cstdio>

#include<vector>

#include<map>

#include<cassert>

using namespace std;

typedef long long LL;

const LL INF=2147483647;

//void assert(bool valid){if(valid)return;for(;;);putchar('E');}

struct SegTree

{

vector<LL>mx,mn,tag;

map<LL,LL>mloc;

LL n,light;

void Clear(vector<LL>&nodes)

{

n=nodes.size();

assert(n>=1);

light=nodes[0];

mloc.clear();

for(LL i=0;i<n;i++)mloc[nodes[i]]=i;

Build(1,0,n-1);

}

void Build(const LL id,const LL l,const LL r)

{

while((int)mx.size()<=id)mx.push\_back(0),mn.push\_back(0),tag.push\_back(0);

mx[id]=mn[id]=tag[id]=0;

if(l==r)return;

const LL mid=(l+r)/2;

Build(id\*2,l,mid),Build(id\*2+1,mid+1,r);

}

void Add(const LL node,const LL v)

{

auto it=mloc.find(node);

assert(it!=mloc.end());

const LL loc=it->second;

Add(1,0,n-1,loc,v);

}

void PutDown(const LL id)

{

if(tag[id])

{

mx[id\*2]+=tag[id],mx[id\*2+1]+=tag[id];

mn[id\*2]+=tag[id],mn[id\*2+1]+=tag[id];

tag[id\*2]+=tag[id],tag[id\*2+1]+=tag[id];

tag[id]=0;

}

}

void Pull(const LL id)

{

mx[id]=max(mx[id\*2],mx[id\*2+1]);

mn[id]=min(mn[id\*2],mn[id\*2+1]);

}

void Add(const LL id,const LL l,const LL r,const LL loc,const LL v)

{

if(l>loc)return;

if(r<=loc){mx[id]+=v,mn[id]+=v,tag[id]+=v;return;}

const LL mid=(l+r)/2;

PutDown(id);

Add(id\*2,l,mid,loc,v),Add(id\*2+1,mid+1,r,loc,v);

Pull(id);

}

LL Query(const LL bound){return Query(1,0,n-1,bound);}

LL Query(const LL id,const LL l,const LL r,const LL bound)

{

if(mx[id]<bound)return 0;

if(mn[id]>=bound)return r-l+1;

if(l==r)return 1;

const LL mid=(l+r)/2;

PutDown(id);

return Query(id\*2,l,mid,bound)+Query(id\*2+1,mid+1,r,bound);

}

};

vector<SegTree>SEGTREE;

vector<LL>ET[300000];

LL FA[300000],SZ[300000];

struct RCmpLL{bool operator()(const LL a,const LL b){return a>b;}};

void BuildSEGTREE(const LL u,vector<LL>&rec)

{

rec.push\_back(u);

if(ET[u].empty())

{

// for(const auto v:rec)printf(" %lld",v);puts("");

SegTree seg;

seg.Clear(rec);

SEGTREE.push\_back(seg);

rec.clear();

return;

}

multimap<LL,LL,RCmpLL>sortch;

for(const auto nxt:ET[u])sortch.insert(make\_pair(SZ[nxt],nxt));

for(const auto &it:sortch)BuildSEGTREE(it.second,rec);

}

LL N,Q,K;

LL BuildSZ(const LL u)

{

LL &ans=SZ[u]=1;

for(const auto nxt:ET[u])ans+=BuildSZ(nxt);

return ans;

}

LL TREEIDX[300000];

LL ANS;

void AddSalary(LL u,const LL money)

{

for(;;)

{

LL ti=TREEIDX[u];

const LL minus=SEGTREE[ti].Query(K);

ANS-=minus;

// printf("minus %lld\n",minus);

SEGTREE[ti].Add(u,money);

const LL plus=SEGTREE[ti].Query(K);

ANS+=plus;

// printf("plus %lld\n",plus);

if(SEGTREE[ti].light==0)break;

u=FA[SEGTREE[ti].light];

}

}

int main()

{

// freopen("B.txt","r",stdin);

while(scanf("%lld%lld%lld",&N,&Q,&K)==3)

{

for(LL i=0;i<N;i++)ET[i].clear();

for(LL i=1;i<N;i++)scanf("%lld",&FA[i]),ET[--FA[i]].push\_back(i);

SEGTREE.clear();

BuildSZ(0);

vector<LL>rec;

BuildSEGTREE(0,rec);

assert(rec.empty());

for(LL i=0;i<N;i++)TREEIDX[i]=-1;

for(LL i=0;i<(int)SEGTREE.size();i++)for(const auto &it:SEGTREE[i].mloc)assert(TREEIDX[it.first]==-1),TREEIDX[it.first]=i;

for(LL i=0;i<N;i++)assert(TREEIDX[i]!=-1);

ANS=0;

while(Q--)

{

static LL a,x;scanf("%lld%lld",&a,&x);

a--;

AddSalary(a,x);

printf("%lld**\n**",ANS);

}

SEGTREE.clear();

break;

}

return 0;

}