**1.3 相亲大会**输出一行为最大连续子串和

#include<stdio.h>

int main()

{ int p=1,n=1,m=1,i,temp,sum,Msum;

scanf("%d",&temp);

Msum=sum=temp;

for(i=2;i<=7;i++)

{ scanf("%d",&temp);

sum+=temp;

if(sum>Msum) {Msum=sum;n=p;m=i;}

if(sum<=0) {sum=0;p=i+1;}

}

if(Msum<=0) printf("won't buy!");

else printf("%d %d %d",Msum,n,m);

return 0;

}

**2.3 击鼓传花**1 号同学出局，将花沿顺时针顺序传递m次，得到花的人出局

#include <stdio.h>

int main()

{

int i,a[500]={0},n,m,k=0,out=1;

scanf("%d %d", &n, &m);

for(i=2;i<=n;i++) a[i]=i;

for(i=1;out<n-1;i++)

{ if(a[i]) k++;

if(k==m) {a[i]=0;k=0;out++;}

if(i==n) i=1;

}

for(i=1;a[i]==0;i++) ;

printf("%d\n",a[i]);

return 0;

}

**5快速排序**

#include<stdio.h>

int a[101],n;

void quicksort(int left,int right)

{ int i,j,t,temp;

if(left>right) return;//递归结束标志

temp=a[left];//temp存基准数

i=left;//调换基准数

j=right;

while(i!=j)

{ while(a[j]>=temp && i<j) j--;

while(a[i]<=temp && i<j) i++;

if(i<j)

{ t=a[j];

a[j]=a[i];

a[i]=t;

}

}

a[left]=a[i];//基准数归位

a[i]=temp;

quicksort(left,i-1);

quicksort(i+1,right);

}

int main()

{ int i,j,t;

scanf("%d",&n);

for(i=1;i<=n;i++) scanf("%d",&a[i]);

quicksort(1,n);

for(i=1;i<=n;i++) printf("%d ",a[i]);

getchar();

return 0;

}

**5合并排序**

#include <iostream>

using namespace std;

int A[100005],B[100005],count=0;

void merge(int low, int mid, int top)

{ int i=low, j=mid+1, k=low;

while(i<=mid && j<=top)

{ if(A[i]>A[j])

{ B[k++]=A[j++];

//count+=mid-i+1; /\*即i~mid这几个数移动\*/

}

else

B[k++]=A[i++];

}

while(i<=mid) B[k++]=A[i++];

while(j<=top) B[k++]=A[j++];

for(i=low;i<=top;i++) A[i]=B[i];

}

void mergesort(int low,int top)

{ if(low<top)

{ int mid=(low+top)/2;

mergesort(low,mid);

mergesort(mid+1,top);

merge(low,mid,top);

}

}

int main()

{ int n,i;

cin>>n;

for(i=1;i<=n;i++) cin>>A[i];

mergesort(1,n);

cout<<count<<endl;

return 0;

}

**算法6.3sights(spfa算法 + 邻接表)**

#include<iostream>

#include<cstdio>

#include<cmath>

#include<queue>

using namespace std;

const int MAXN=100010;

const int mod=707063423;

const int MAX=99999999;

struct Edge

{

int to,w;

int next;

}edge[MAXN\*2];

\_\_int64 dis[MAXN];

int place=1;

int head[MAXN]={0};

void creat(int a,int b,int c)

{

edge[place].to=b;edge[place].w=c;

edge[place].next=head[a];

head[a]=place;

place++;

}

void spfa(int n,int s)

{

queue<int>q;

bool vis[MAXN]={0}; //vis[i]用于判断i是否在队列q中

int a,i;

for(i=0;i<=n; i++) dis[i]=MAX;

dis[s]=0;

q.push(s);

vis[s]=1;

while(!q.empty())

{

a=q.front();

q.pop();

vis[i]=0;

for(i=head[a]; i!=0; i=edge[i].next)

{

if(dis[edge[i].to]>dis[a]+edge[i].w)

{

dis[edge[i].to]=dis[a]+edge[i].w;

if(!vis[edge[i].to])

{

q.push(edge[i].to);

vis[edge[i].to]=1;

}

}

}

}

}

int main()

{

int n,a,b,c,i;

scanf("%d",&n);

for(i=1;i<=n-1;i++)

{

scanf("%d%d%d",&a,&b,&c);

creat(a,b,c);

creat(b,a,c);

}

spfa(n,1);

scanf("%d",&n);

while(n--)

{

scanf("%d%d%d",&a,&c,&b);

printf("%I64d\n",(c-dis[a]+dis[b]+mod)%mod );

}

return 0;

}

**到源点最短路径：**

**7dijkstar算法的实现**

#include <cstdio>

#include <iostream>

#include <cstring>

using namespace std;

const int N=101;

const int Max=1000000000;

int map[N][N];

int dis[N];

bool visit[N];

void mapinit(int n)//map[]初始化

{

int i,j;

for(i=1;i<=n;++i) //map[]初始化

{

map[i][i]=0;

for(j=1;j<i;++j)

map[i][j]=map[j][i]=Max;

}

}

void dijkstar(int n,int m,int s)

{

int k, min, i, j;

visit[s] = 1;

for(i=1; i<=n; i++) //dis[]初始化

{

dis[i] = map[s][i];

}

for(i=1;i<=n;++i)

{

for(j=1,k=s,min=Max; j<=n; ++j) //选出最优路，k并入集合

{

if(!visit[j]&&min>dis[j])

{

min=dis[j];

k=j;

}

}

visit[k]=1;

for(j=1; j<=n; ++j) //k并入集合后,更新数据

{

if(!visit[j]&&dis[j] > dis[k]+map[k][j])

dis[j]=dis[k]+map[k][j];

}

}

}

int main( )

{

int n, m, s, e;

int u, v, w, i;

scanf("%d%d%d%d",&n,&m,&s,&e);

mapinit(n);//map[]初始化

for(i=1;i<=m;++i) //map[]输入建立

{

scanf("%d%d%d",&u,&v,&w);

if(w<map[u][v]) //可能有多条路，只记录最短的

map[u][v]=map[v][u]=w;

}

dijkstar(n,m,s);

printf("%d\n", dis[e]);

return 0;

}

**7 Bellman-Ford算法实现**#include<iostream>

#include<cstdio>

using namespace std;

#define MAX 0x3f3f3f3f

#define N 1010

struct Edge //边

{

int u, v;

int cost;

}edge[N];

int nodenum; //顶点数

int edgenum; //边数

int original; //起点

int dis[N]; //其他点到源点最短距离

int pre[N];

bool Bellman\_Ford()

{

int i,j;

for(i=1; i<=nodenum; ++i) //初始化距离为最大值

dis[i]=(i==original? 0:MAX);

for(i=1; i<=nodenum-1; ++i) //最坏情况下每次循环至少一个点找到最优路径,除去源点要循环nodenum-1次

for(j=1; j<=edgenum; ++j) //要尝试每条边,

if(dis[edge[j].v] > dis[edge[j].u]+edge[j].cost) //其他点到源点距离一步步的收敛

{

dis[edge[j].v]=dis[edge[j].u]+edge[j].cost;

pre[edge[j].v]=edge[j].u;

}

bool flag = 1; //判断是否含有负权回路

for(i=1; i<=edgenum; ++i)

if(dis[edge[i].v] > dis[edge[i].u]+edge[i].cost)

{

flag = 0;

break;

}

return flag;

}

void print\_path(int root) //打印最短路的路径（反向）

{

while(root != pre[root]) //前驱

{

printf("%d-->", root);

root = pre[root];

}

if(root == pre[root])

printf("%d\n", root);

}

int main()

{

scanf("%d%d%d", &nodenum, &edgenum, &original);

pre[original] = original;

for(int i = 1; i <= edgenum; ++i)

scanf("%d%d%d", &edge[i].u, &edge[i].v, &edge[i].cost);

if(Bellman\_Ford())

for(int i = 1; i <= nodenum; ++i) //每个点最短路

{

printf("%d\n", dis[i]);

printf("Path:");

print\_path(i);

}

else

printf("have negative circle\n");

return 0;

}

**最小支撑树;**

**7prim算法（m>n^2时最优）**

#include<iostream>

#include<fstream>

using namespace std;

#define N 100

#define MAX 0x7fffffff

int map[N][N];

int closest[N];//closet[a]值是与a最近的点

int lowcost[N];//lowcost[i]是k到i距离

bool visit[N];

void mapinit(int n)//map[]初始化

{

int i,j;

for(i=1;i<=n;++i) //map[]初始化

{

map[i][i]=0;

for(j=1;j<i;++j)

map[i][j]=map[j][i]=MAX;

}

}

int prim(int n)

{

int i, j, min, k, sum=0;

for(i=2; i<=n; i++)

{

lowcost[i]=map[1][i];

closest[i]=1;

visit[i]=0;

}

closest[1] = 0;

visit[1]=1;

for(i=2; i<=n; i++)

{

for(j=2, k=0, min=MAX; j<=n; j++)

{

if(lowcost[j]<min && !visit[j])

{

min=lowcost[j];

k=j;

}

}

visit[k]=1;

sum += min;

cout << "V" << closest[k] << "-V" <<k<< "=" << min << endl;

for(j=2; j<=n; j++)

{

if(!visit[j] && map[k][j]<lowcost[j])

{

lowcost[j]=map[k][j];

closest[j]=k;

}

}

}

return sum;

}

int main()

{

int i, m, n, u, v, w;

scanf("%d%d",&n,&m);

mapinit(n);//map[]初始化

for(i=1;i<=m;++i) //map[]输入建立

{

scanf("%d%d%d",&u,&v,&w);

if(w<map[u][v]) //可能有多条路，只记录最短的

map[u][v]=map[v][u]=w;

}

int cost=prim(n);

cout << "最小权值和=" << cost << endl;

return 0;

}

**7.4玩游戏的亚索 (kruskal算法) （m<n^2时优先）**

#include<cstdio>

#include<algorithm>

using namespace std;

const int MAX=500001;

struct Rode

{

int a,b,d;

}R[MAX];

int f[MAX]; //用于并查集，若以用最短路径连接归并到一个集合内

int root(int x)

{

if(x!=f[x]) f[x]=root(f[x]);//压缩路径

return f[x];

}

int cmp(Rode x,Rode y)

{

return x.d<y.d;

}

int main( )

{

int n,m,i;

scanf("%d%d",&n, &m);

for(i=0; i<=n; i++) f[i]=i; //初始化，n个点分成n个集合

for(i=0; i<m; i++)

scanf("%d%d%d",&R[i].a, &R[i].b, &R[i].d);

sort(R, R+m, cmp); //按距离，从小到大排序

int aroot,broot,sum=0,time=0;

for(i=0; i<m; i++)

{

aroot=root(R[i].a); //含a点集合的首领

broot=root(R[i].b);

if(aroot==broot) continue; //a b在一个集合说明已经有最短路径相连或间接相连

sum+=R[i].d; time++; //a b不在一个集合，

f[aroot]=broot;

}

printf("%d\n",sum);

return 0;

}

**算法7.3被Gank的亚索**

#include <cstdio>

#include <iostream>

#include <cstring>

using namespace std;

const int N=101;

const int Max=1000000000;

int map[N][N];

int edgenum[N][N]={0};

int dis[N];

int pathnum[N]; //起点s到该点最优路径条数

bool visit[N];

int main( )

{

int n, m, s, e;

int u, v, w, i, j;

scanf("%d%d%d%d",&n,&m,&s,&e);

for(i=1;i<=n;++i) //map[]初始化

{

map[i][i]=0;

for(j=1;j<i;++j)

map[i][j]=map[j][i]=Max;

}

for(i=1;i<=m;++i) //输入

{

scanf("%d%d%d",&u,&v,&w);

if(w<map[u][v]) //可能有多条路，只记录最短的

{

map[u][v]=map[v][u]=w;

edgenum[u][v]=edgenum[v][u]=1;

}

else if(w==map[u][v])

{

edgenum[u][v]++;

edgenum[v][u]++;

}

}

int k, min;

visit[s] = 1;

pathnum[s] = 1;

for(i=1; i<=n; i++)

{

dis[i] = map[s][i];

if(map[s][i] != Max && i!=s) //有道路，且不重回该点

pathnum[i] = edgenum[s][i];

}

for(i=1;i<=n;i++)

{

for(j=1,k=s,min=Max; j<=n; ++j) //选出最优路，k并入集合

{

if(!visit[j]&&min>dis[j])

{

min=dis[j];

k=j;

}

}

if(k==e) break;

visit[k]=1;

for(j=1; j<=n; ++j) //k并入集合后,更新数据

{

if(!visit[j])

{

if(dis[j] > dis[k]+map[k][j])

{

dis[j]=dis[k]+map[k][j];

pathnum[j] = pathnum[k]\*edgenum[k][j];

}

else if(dis[j] == dis[k]+map[k][j] ) //有多条最优路径

{

pathnum[j] += pathnum[k]\*edgenum[k][j];

}

}

}

}

printf("%d %d\n", dis[e], pathnum[e]);

return 0;

}

L2-001. 紧急救援

#include <cstdio>

#include <iostream>

#include <cstring>

#include <stack>

using namespace std;

const int N=501;

const int Max=1000000000;

int map[N][N];

int edgenum[N][N]={0};

int dis[N];

int pre[505]; //记录路径

int car[505], totcar[505]; //car是各点救援人数， totcar是起点s到该点救援人数

int pathnum[505]; //起点s到该点最优路径条数

bool visit[505];

int main( )

{

int n, m, s, e, u, v, w, i, j;

scanf("%d%d%d%d",&n,&m,&s,&e);

for(i=0; i<n; i++) scanf("%d",&car[i]);

for(i=0;i<n;++i) //map[]初始化

{

map[i][i]=0;

for(j=0;j<i;++j)

map[i][j]=map[j][i]=Max;

}

for(i=1;i<=m;++i) //输入

{

scanf("%d%d%d",&u,&v,&w);

if(w<map[u][v]) //可能有多条路，只记录最短的

{

map[u][v]=map[v][u]=w;

edgenum[u][v]=edgenum[v][u]=1;

}

else if(w==map[u][v])

{

edgenum[u][v]++;

edgenum[v][u]++;

}

}

int k, min;

visit[s] = 1;

totcar[s] = car[s];

pathnum[s] = 1;

for(i=0; i<n; i++)

{

dis[i] = map[s][i];

if(map[s][i] != Max && i!=s) //有道路，且不重回该点

{

pre[i] = s;

totcar[i] = car[s] + car[i];

pathnum[i] = edgenum[s][i];

}

}

for(i=1;i<=n;i++)

{

for(j=0,k=s,min=Max; j<n; ++j)

{

if(!visit[j]&&min>dis[j])

{

min=dis[j];

k=j;

}

}

if(k==e) break;

visit[k]=1;

for(j=0; j<n; ++j)

{

if(!visit[j])

{

if(dis[j] > dis[k]+map[k][j])

{

dis[j]=dis[k]+map[k][j];

pathnum[j] = pathnum[k]\*edgenum[k][j];

pre[j]=k; //把(未走过 )j点上级改为K

totcar[j] = totcar[k]+car[j];

}

else if(dis[j] == dis[k]+map[k][j] ) //多条 最优路径

{

pathnum[j] += (pathnum[k]\*edgenum[k][j]);

if(totcar[j] < totcar[k]+car[j]) //路径同样最优下比较救援人数

{

totcar[j] = totcar[k]+car[j];

pre[j] = k;

}

}

}

}

}

printf("%d %d\n", pathnum[e], totcar[e]);

stack<int>q;

for(i=e; i!=s; i=pre[i])

{

q.push(i);

}

q.push(s);

while(q.size()>1)

{

printf("%d ", q.top());

q.pop();

}

printf("%d\n",q.top());

return 0;

}

**邻接表变为双向**

#include<iostream>

#include<cstdio>

using namespace std;

const int MAX=99999999;

const int N=100003;

struct Edge //边

{

int u, v;

int cost;

}edge[N];

int nodenum; //顶点数

int aedgenum; //边数

int bedgenum;

int original; //起点

int dis[N]; //其他点到源点最短距离

int pre[N];

bool Bellman\_Ford()

{

int i,j;

for(i=1; i<=nodenum; ++i) //初始化距离为最大值

dis[i]=(i==original? 0:MAX);

for(i=1; i<=nodenum-1; ++i) //最坏情况下每次循环至少一个点找到最优路径,除去源点要循环nodenum-1次

for(j=1; j<=2\*aedgenum+bedgenum; ++j) //要尝试每条边,

if(dis[edge[j].v] > dis[edge[j].u]+edge[j].cost) //其他点到源点距离一步步的收敛

{

dis[edge[j].v]=dis[edge[j].u]+edge[j].cost;

pre[edge[j].v]=edge[j].u;

}

bool flag = 1; //判断是否含有负权回路

for(i=1; i<=2\*aedgenum+bedgenum; ++i)

if(dis[edge[i].v] > dis[edge[i].u]+edge[i].cost)

{

flag = 0;

break;

}

return flag;

}

/\*void print\_path(int root) //打印最短路的路径（反向）

{

while(root != pre[root]) //前驱

{

printf("%d-->", root);

root = pre[root];

}

if(root == pre[root])

printf("%d\n", root);

}\*/

int main()

{

int i,a,b,c;

scanf("%d%d%d%d", &nodenum, &aedgenum,&bedgenum, &original);

pre[original] = original;

for(i = 1; i <=2\*aedgenum; i+=2)

{

scanf("%d%d%d", &a, &b, &c);

edge[i].u=a,edge[i].v=b,edge[i].cost=c;

edge[i+1].u=b,edge[i+1].v=a,edge[i+1].cost=c;

}

for(; i <=(2\*aedgenum+bedgenum); i++)

{

scanf("%d%d%d", &a, &b, &c);

edge[i].u=a,edge[i].v=b,edge[i].cost=c;

}

if(Bellman\_Ford())

for(int i = 1; i <= nodenum; ++i) //每个点最短路

{

if(dis[i]>=MAX)

printf("NO PATH\n");

else

printf("%d\n", dis[i]);

//print\_path(i);

}

return 0;

}

**算法10 AVL模板**

#include<iostream>

#include<queue>

#include<stack>

#define Max(a,b) (a>b?a:b)

using namespace std;

//AVL树节点信息

typedef int T;

class TreeNode

{

public:

T data;

TreeNode\* lson;

TreeNode\* rson;

int hgt;//以此节点为根的树的高度

unsigned int freq;//频率

TreeNode():lson(NULL),rson(NULL),hgt(0),freq(1){}

TreeNode(T a):data(a),lson(NULL),rson(NULL),freq(1){}

};

//AVL树类的属性和方法声明

class AVLTree

{

private:

TreeNode\* root;//根节点

TreeNode\* insertpri(TreeNode\* node,T x);//插入

TreeNode\* Deletepri(TreeNode\* node,T x);//删除

void pretree(TreeNode\* node); //前序

void pretree2(TreeNode\* node);

void intree(TreeNode\* node);//中序遍历

void intree2(TreeNode\* node);

void posttree(TreeNode\* node); //后序

void posttree2(TreeNode\* node);

void leveltree(TreeNode\* node);//层次遍历

int height(TreeNode\* node);//求树的高度

TreeNode\* SingRotateLeft(TreeNode\* k2);//左左情况下的旋转

TreeNode\* SingRotateRight(TreeNode\* k2);//右右情况下的旋转

TreeNode\* DoubleRotateLR(TreeNode\* k3);//左右情况下的旋转

TreeNode\* DoubleRotateRL(TreeNode\* k3);//右左情况下的旋转

public:

AVLTree():root(NULL){}

void insert(T x);//插入接口

void Delete(T x);//删除接口

void preorder(); //前序遍历接口

void preorder2();//非递归前序遍历接口

void inorder(); //中序遍历接口

void inorder2(); //非递归中序遍历接口

void postorder(); //后序遍历接口

void postorder2(); //非递归后序遍历接口

void levelorder(); //层次遍历接口

};

//计算以节点为根的树的高度

int AVLTree::height(TreeNode\* node)

{

if(node!=NULL) return node->hgt;

return -1;

}

//左左情况下的旋转

TreeNode\* AVLTree::SingRotateLeft(TreeNode\* k2)

{

TreeNode\* k1=k2->lson;

k2->lson=k1->rson;

k1->rson=k2;

k2->hgt=Max(height(k2->lson),height(k2->rson))+1;

k1->hgt=Max(height(k1->lson),k2->hgt)+1;

return k1;

}

//右右情况下的旋转

TreeNode\* AVLTree::SingRotateRight(TreeNode\* k2)

{

TreeNode \*k1=k2->rson;

k2->rson=k1->lson;

k1->lson=k2;

k2->hgt=Max(height(k2->lson),height(k2->rson))+1;

k1->hgt=Max(height(k1->rson),k2->hgt)+1;

return k1;

}

//左右情况的旋转

TreeNode\* AVLTree::DoubleRotateLR(TreeNode\* k3)

{

k3->lson=SingRotateRight(k3->lson);

return SingRotateLeft(k3);

}

//右左情况的旋转

TreeNode\* AVLTree::DoubleRotateRL(TreeNode\* k3)

{

k3->rson=SingRotateLeft(k3->rson);

return SingRotateRight(k3);

}

//插入

TreeNode\* AVLTree::insertpri(TreeNode\* node,T x)

{

if(node==NULL)//如果节点为空,就在此节点处加入x信息

{

node=new TreeNode(x);

}

else if(node->data > x)//如果x小于节点的值,就继续在节点的左子树中插入x

{

node->lson= insertpri(node->lson,x);

if(2==height(node->lson)-height(node->rson))

if(node->lson->data>x)

node=SingRotateLeft(node);

else

node=DoubleRotateLR(node);

}

else if(x>node->data)//如果x大于节点的值,就继续在节点的右子树中插入x

{

node->rson= insertpri(node->rson,x);

if(2==height(node->rson)-height(node->lson))//如果高度之差为2的话就失去了平衡,需要旋转

if(x>node->rson->data)

node=SingRotateRight(node);

else

node=DoubleRotateRL(node);

}

else ++(node->freq);//如果相等,就把频率加1

node->hgt=Max(height(node->lson),height(node->rson))+1;

return node;

}

//插入接口

void AVLTree::insert(T x)

{

root=insertpri(root,x);

}

//删除

TreeNode\* AVLTree::Deletepri(TreeNode\* node,T x)

{

if(node==NULL) return NULL;//没有找到值是x的节点

if(x < node->data)

{

node->lson= Deletepri(node->lson,x);//如果x小于节点的值,就继续在节点的左子树中删除x

if(2==height(node->rson)-height(node->lson))

if(node->rson->lson!=NULL&&(height(node->rson->lson)>height(node->rson->rson)) )

node=DoubleRotateRL(node);

else

node=SingRotateRight(node);

}

else if(x > node->data)

{

node->rson= Deletepri(node->rson,x);//如果x大于节点的值,就继续在节点的右子树中删除x

if(2==height(node->lson)-height(node->rson))

if(node->lson->rson!=NULL&& (height(node->lson->rson)>height(node->lson->lson) ))

node=DoubleRotateLR(node);

else

node=SingRotateLeft(node);

}

else//如果相等,此节点就是要删除的节点

{

if(node->lson&&node->rson)//此节点有两个儿子

{

if( height(node->lson)>height(node->rson) )//左子树比右子树高；

{

//(01)找出tree的左子树中的最大节点

//(02)将该最大节点的值赋值给tree。

//(03)删除该最大节点。

//这类似于用"tree的左子树中最大节点"做"tree"的替身；

//采用这种方式的好处是：删除"tree的左子树中最大节点"之后，AVL树仍然是平衡的。

TreeNode\* temp=node->lson;

while(temp&&temp->rson) temp=temp->rson;

node->data=temp->data;

node->freq=temp->freq;

node->lson=Deletepri(node->lson,temp->data);

}

else//它们相等，或右子树比左子树高

{

//则(01)找出tree的右子树中的最小节点

//(02)将该最小节点的值赋值给tree。

//(03)删除该最小节点。

//这类似于用"tree的右子树中最小节点"做"tree"的替身；

//采用这种方式的好处是：删除"tree的右子树中最小节点"之后，AVL树仍然是平衡的。

TreeNode\* temp=node->rson;

while(temp&&temp->lson) temp=temp->lson;

node->rson=Deletepri(node->rson, temp->data);

}

}

else//此节点有1个或0个儿子

{

TreeNode\* temp=node;

node=(node->lson!=NULL) ? node->lson : node->rson;

delete(temp);

}

}

if(node) node->hgt=Max(height(node->lson),height(node->rson))+1;

return node;

}

//删除接口

void AVLTree::Delete(T x)

{

root=Deletepri(root,x);

}

//递归前序遍历

void AVLTree::pretree(TreeNode\* node)

{

if(node==NULL) return;

cout<<node->data<<" ";

pretree(node->lson);

pretree(node->rson);

}

void AVLTree::preorder()

{

pretree(root);

cout<<endl;

}

//非递归前序遍历

void AVLTree::pretree2(TreeNode \*node)

{

stack<TreeNode\*>s;

while(node!=NULL||!s.empty())

{

while(node!=NULL)

{

cout<<node->data<<" ";

s.push(node);

node=node->lson;

}

if(!s.empty())

{

node=s.top();

s.pop();

if(node) node=node->rson;

}

}

}

void AVLTree::preorder2()

{

pretree2(root);

cout<<endl;

}

//中序遍历函数

void AVLTree::intree(TreeNode\* node)

{

if(node==NULL) return;

intree(node->lson);

cout<<node->data<<" ";

intree(node->rson);

}

void AVLTree::inorder()

{

intree(root);

cout<<endl;

}

//非递归中序遍历

void AVLTree::intree2(TreeNode\* node)

{

stack<TreeNode\*> s;

while(node!=NULL||!s.empty())

{

while(node!=NULL)

{

s.push(node);

node=node->lson;

}

if(!s.empty())

{

node=s.top();

s.pop();

cout<<node->data<<" ";

node=node->rson;

}

}

}

void AVLTree::inorder2()

{

intree2(root);

cout<<endl;

}

//后序遍历

void AVLTree::posttree(TreeNode\* node)

{

if(node==NULL) return;

posttree(node->lson);

posttree(node->rson);

cout<<node->data<<" ";

}

void AVLTree::postorder()

{

posttree(root);

cout<<endl;

}

//非递归后序遍历

void AVLTree::posttree2(TreeNode\* node)

{

stack<TreeNode\*> s;

TreeNode \*pre=NULL; //前一次访问的结点

s.push(node);

while(!s.empty())

{

node=s.top();

if((node->lson==NULL&&node->rson==NULL)||

(pre!=NULL&&(pre==node->lson||pre==node->rson)))

{

cout<<node->data<<" "; //如果当前结点没有孩子结点或者孩子节点都已被访问过

s.pop();

pre=node;

}

else

{ //压入栈，先右子后左子

if(node->rson!=NULL)

s.push(node->rson);

if(node->lson!=NULL)

s.push(node->lson);

}

}

}

void AVLTree::postorder2()

{

posttree2(root);

cout<<endl;

}

//层次遍历函数

void AVLTree::leveltree(TreeNode\* node)

{

queue<TreeNode\*>q;

if(node) q.push(node);

while(!q.empty())

{

node=q.front();

q.pop();

cout<<node->data<<" ";

if(node->lson) q.push(node->lson);

if(node->rson) q.push(node->rson);

}

}

void AVLTree::levelorder()

{

leveltree(root);

cout<<endl;

}

int main()

{

AVLTree tree;

int i,n,a;

cin>>n;

for(i=0;i<n;i++)

{

cin>>a;

tree.insert(a);

}

tree.preorder2();

tree.inorder2();

tree.postorder2();

tree.levelorder();

return 0;

}

**算法11优先队列(堆的实现)**

#include<iostream>

#include<cstdio>

using namespace std;

struct Node

{

int p,d;

};

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

typedef Node SetItem;

#define key(a) a.p

#define less(a,b) key(a)<key(b) //或者a.p<b.p

#define lessequ(a,b) key(a)<=key(b)

//结构体的复制不用重载

typedef struct minheap\* Heap;

typedef struct minheap //堆的大小，上限，放元素数组的地址

{

int last,maxsize;

SetItem \*heap;

}Minheap;

Heap MinHeapInit(int HeapSize) //创建上限HeapSiae的空堆(开空间)

{

Heap H=new struct minheap;

H->maxsize=HeapSize;

H->heap=new SetItem[H->maxsize]; //开maxsize大小的连续的空间，作为数组

H->last=0;

return H;

}

int HeapInsert(SetItem x, Heap H) //在堆H中插入元素x

{

if(H->last==H->maxsize) exit(0); //堆已满

H->last++; //堆大小加 1

int p=H->last; //从堆底往前搜索

while(p!=1 && less(x,H->heap[p/2]))

{

H->heap[p]=H->heap[p/2];

p/=2;

}

H->heap[p]=x;

return p;

}

SetItem DeleteMin(Heap H) //删除堆H的堆顶

{

if(H->last==0) exit(0); //堆已空

SetItem x=H->heap[1], y=H->heap[H->last]; //用堆底y填补删除堆顶x出现的空缺

H->last--; //堆大小减 1

int ps=2;

while(ps<=H->last) //从上往下寻找插入y的合适位置p(要求y比p的两个儿子都小或p没儿子)

{

if(ps<H->last && less(H->heap[ps+1],H->heap[ps])) ps+=1; //选出p最小儿子;

if(less(y,H->heap[ps])) break;

H->heap[ps/2]=H->heap[ps];

ps\*=2;

}

H->heap[ps/2]=y;//p位置元素会覆盖到其父亲上，不会使p位置元素丧失

return x;

}

Heap BuildHeap(SetItem a[], int size, int arraysize) //将上限arraysize，大小size的数组a[]调整成堆

{

Heap H=MinHeapInit(arraysize);

H->heap=a; //用数组对堆赋值

H->last=size; //赋值后堆大小是数组大小size

int p,ps;

SetItem y;

for(p=H->last/2; p>=1; p--) //从堆倒数第二层往前(先把最小分支调成堆，一步步扩大分支)

{

y=H->heap[p];

ps=2\*p;

while(ps<=H->last)

{

if(ps<H->last && less(H->heap[ps+1],H->heap[ps])) ps+=1; //选出p最小儿子;

if(less(y,H->heap[ps])) break;

H->heap[ps/2]=H->heap[ps];

ps\*=2;

}

H->heap[ps/2]=y;

}

return H;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main()//排序输出

{

Heap H;

Node a[10];

int i,n;

scanf("%d",&n);

for(i=1; i<=n; i++)

scanf("%d %d",&a[i].p,&a[i].d);

BuildHeap(a,n,10);

for(i=1; i<=n; i++)

printf("%d %d\n",a[i].p,a[i].d);

return 0;

}

**12并查集的删除**

#include<iostream>

#include<cstdio>

#include<cstring>

using namespace std;

const int N=100000;

int parent[2\*N];

int shell[N];

int place;

void init(int n) //初始化

{

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

parent[i]=shell[i]=i;

place=n+1;

}

int root(int x) //找首领

{

if(x!= parent[x])

parent[x]=root(parent[x]);

return parent[x];

}

int main()

{

int n,m,a,b;

char c;

scanf("%d%d",&n,&m);

init(n);

while(m--)

{

scanf("%c",&c);

if(c=='M')

{

scanf("%d%d",&a,&b);

if( root( shell[a] )!=root( shell[b] ) )

parent[root(shell[a])]= root(shell[b]);

}

else if(c=='I')

{

scanf("%d",&a);

shell[a]=place;

parent[place]=place;

place++;

}

else if(c=='Q')

{

scanf("%d%d",&a,&b);

if(root(shell[a])==root(shell[b]))

printf("Yes\n");

else

printf("No\n");

}

}

return 0;

}

**12山海经**

#include<iostream>

#include<cstdio>

using namespace std;

int f[50001],d[50001];

int M,N,K;

int root(int x)

{

if(x!=f[x])

{

int temp=f[x];

f[x]=root(f[x]);

d[x]=(d[x]+d[temp])%M;

}

return f[x];

}

int main()

{

int i,D,A,B,count=0;

scanf("%d%d%d",&M,&N,&K);

//scanf("%d%d",&N,&K);M=3;

for(i=1; i<=N; i++)

{

f[i]=i;

d[i]=0;

}//初始化

for(i=1; i<=K; i++)

{

scanf("%d%d%d",&D,&A,&B);

if( D>=M || A>N || B>N || (A==B&&D!=0))

{

count++;

continue;

}

if(root(A)==root(B))

{

if( d[A]!=(d[B]+D)%M ) count++;

}

else

{

d[f[A]]=(d[B]-d[A]+D+M)%M;

f[f[A]]=f[B];

}

}

cout<<count<<endl;

return 0;

}

**12箱子里的木块**#include<iostream>  
using namespace std;  
const int N=30001;  
int f[N],d[N];  
  
int findroot(int a)  
{  
if(f[a]>0)  
{  
int temp=f[a];  
f[a]=findroot(f[a]);  
d[a]=d[a]+d[temp];  
}  
if(f[a]<0)  return a;  
return f[a];  
}  
  
int main()  
{  
int n,T,i,a,b,alow,blow,atop,btop;  
char opt;  
cin>>n>>T;  
for(i=1; i<=n; i++)  
{  
f[i]=-i;  
d[i]=0;  
}  
  
while(T--)  
{  
cin>>opt;  
if(opt=='M')  
{  
cin>>a>>b;  
atop=findroot(a);  
btop=findroot(b);  
alow=-f[atop];  
blow=-f[btop];  
  
findroot(blow); //保证blow的d正确  
f[atop]=btop;  
d[atop]=d[blow]+1;  
f[btop]=-alow;  
}  
if(opt=='C')  
{  
cin>>a;  
findroot(a); //保证a的d值正确  
cout<<d[a]<<endl;  
}  
}  
return 0;  
}