**4.1.1 并查集的基本操作**

|  |
| --- |
| 例4-1. How Many Tables hdu 1213 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include <bits/stdc++.h>  using namespace std;  const int N = 1050;  int s[N];  void init\_set(){ //初始化  for(int i = 1; i <= N; i++) s[i] = i;  }  int find\_set(int x){ //查找  return x==s[x]? x:find\_set(s[x]);  }  void merge\_set(int x, int y){ //合并  x = find\_set(x); y = find\_set(y);  if(x != y) s[x] = s[y]; //把x合并到y上，y的根成为x的根  }  int main (){  int t, n, m, x, y; cin >> t; //t个测试  while(t--){  cin >> n >> m;  init\_set();  for(int i = 1; i <= m; i++){  cin >> x >> y;  merge\_set(x, y); //合并x和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;  } |

**4.1.2 合并的优化**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | int height[N];  void init\_set(){  for(int i = 1; i <= N; i++){  s[i] = i;  height[i]=0; //树的高度  }  }  void merge\_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;  }  } |

**4.1.4 带权并查集**

|  |
| --- |
| 例4-2. How Many Answers Are Wrong hdu 3038 |

。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #include <bits/stdc++.h>  using namespace std;  const int N = 200010;  int s[N]; //集  int d[N]; //权值：记录当前结点到根结点的距离  int ans;  void init\_set(){ //初始化  for(int i = 0; i <= N; i++) { s[i] = i; d[i] = 0; }  }  int find\_set(int x){ //带权值的路径压缩  if(x != s[x]) {  int t = s[x]; //记录父结点  s[x] = find\_set(s[x]); //路径压缩。递归最后返回的是根结点  d[x] += d[t]; //权值更新为x到根结点的权值  }  return s[x];  }  void merge\_set(int a, int b,int v){ //合并  int roota = find\_set(a), rootb = find\_set(b);  if(roota == rootb){  if(d[a] - d[b] != v) ans++;  }  else{  s[roota] = rootb; //合并  d[roota] = d[b]- d[a] + v;  }  }  int main(){  int n,m;  while(scanf("%d%d",&n,&m) != EOF){  init\_set();  ans = 0;  while(m--){  int a,b,v; scanf("%d%d%d",&a,&b,&v);  a--;  merge\_set(a, b, v);  }  printf("%d\n",ans);  }  return 0;  } |

|  |
| --- |
| 例4-3. 食物链 poj 1182 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | #include <iostream>  #include <stdio.h>  using namespace std;  const int N = 50005;  int s[N]; //集  int d[N]; // 0：同类； 1：吃； 2：被吃  int ans;  void init\_set(){ //初始化  for(int i = 0; i <= N; i++) { s[i] = i; d[i] = 0; }  }  int find\_set(int x){ //带权值的路径压缩  if(x != s[x]) {  int t = s[x]; //记录父结点  s[x] = find\_set(s[x]); //路径压缩。递归最后返回的是根结点  d[x] = (d[x] + d[t]) % 3; //权值更新为x到根结点的权值  }  return s[x];  }  void merge\_set(int x, int y, int relation){ //合并  int rootx = find\_set(x); int rooty = find\_set(y);  if (rootx == rooty){  if ((relation - 1) != ((d[x] - d[y] + 3) % 3)) //判断矛盾  ans++;  }  else {  s[rootx] = rooty; //合并  d[rootx] = (d[y] - d[x] + relation - 1) % 3; //更新权值  }  }  int main(){  int n, k; cin >> n >> k;  init\_set();  ans = 0;  while (k--){  int relation, x, y; scanf("%d%d%d",&relation,&x,&y);  if( x>n || y>n || (relation==2 && x==y) ) ans++;  else merge\_set(x,y,relation);  }  cout << ans;  return 0;  } |

**4.2.1 树状数组概念和基本编码**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | #include <bits/stdc++.h>  using namespace std;  const int N = 1000;  #define lowbit(x) ((x) & - (x))  int tree[N]={0};  void update(int x, int d) { //单点修改：修改元素a[x], a[x] = a[x] + d  while(x <= N) {  tree[x] += d;  x += lowbit(x);  }  }  int sum(int x) { //查询前缀和：返回前缀和sum = a[1] + a[2] +... + a[x]  int ans = 0;  while(x > 0){  ans += tree[x];  x -= lowbit(x);  }  return ans;  }  //以上是树状数组相关  int a[11]={0,4,5,6,7,8,9,10,11,12,13}; //注意：a[0]不用  int main (){  for(int i=1;i<=10;i++) update(i,a[i]); //初始化计算tree[]数组  cout << "old: [5,8]="<<sum(8)-sum(4)<<endl; //查询区间和，例如查询[5,8]=38  update(5,100); //模拟一次修改： a[5] = a[5]+100  cout << "new: [5,8]="<<sum(8)-sum(4); //重新查询区间和[5,8]=138  return 0;  } |

**4.2.2 树状数组基本应用**

|  |  |  |
| --- | --- | --- |
| 例4-4. Color the ball hdu 1556 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | //hdu 1556代码  //tree[N]，lowbit(x)，update()，sum()的代码在前面已给出  const int N = 100010;  int main(){  int n;  while(~scanf("%d",&n)) {  memset(tree,0,sizeof(tree)); //只需要一个tree[]数组  for(int i=1;i<=n;i++) { //区间修改  int L, R; scanf("%d%d",&L,&R);  update(L,1); //本题的d = 1  update(R+1,-1);  }  for(int i=1;i<=n;i++){ //单点查询  if(i!=n) printf("%d ",sum(i)); //把sum(i)看成a[i]  else printf("%d\n",sum(i));  }  }  return 0;  } |
| 例4-5. 线段树1 洛谷P3372 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | #include<bits/stdc++.h>  using namespace std;  #define ll long long  const int N = 100010;  #define lowbit(x) ((x) & - (x))  ll tree1[N],tree2[N]; //2个树状数组  void update1(ll x,ll d){while(x<=N) {tree1[x]+=d; x+=lowbit(x);}}  void update2(ll x,ll d){while(x<=N) {tree2[x]+=d; x+=lowbit(x);}}  ll sum1(ll x){ll ans = 0;while(x>0) {ans+=tree1[x];x-=lowbit(x);}return ans;}  ll sum2(ll x){ll ans = 0;while(x>0) {ans+=tree2[x];x-=lowbit(x);}return ans;}  int main(){  ll n, m; scanf("%lld%lld",&n,&m);  ll old = 0, a;  for (int i=1;i<=n;i++) {  scanf("%lld",&a); //输入每个初始值  update1(i, a-old); //差分数组原理，初始化  update2(i,(i-1)\*(a-old));  old = a;  }  while (m--){ //m个操作  ll q, L, R, d; scanf("%lld",&q);  if (q==1){ //区间修改  scanf("%lld%lld%lld",&L, &R, &d);  update1(L,d); //第1个树状数组  update1(R+1,-d);  update2(L,d\*(L-1)); //第2个树状数组  update2(R+1,-d\*R); //d\*R = d\*(R+1-1)  }  else { //区间询问  scanf("%lld%lld",&L,&R);  printf("%lld\n",R\*sum1(R)-sum2(R) - (L-1)\*sum1(L-1)+sum2(L-1));  }  }  return 0;  } |

**4.2.3 树状数组扩展应用**

|  |
| --- |
| 例4-6. 上帝造题的七分钟 洛谷 P4514 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | #include<bits/stdc++.h>  using namespace std;  const int N = 2050;  int t1[N][N],t2[N][N],t3[N][N],t4[N][N];  #define lowbit(x) ((x) & - (x))  int n,m;  void update(int x,int y,int d){  for(int i=x;i<=n;i+=lowbit(i))  for(int j=y;j<=m;j+=lowbit(j)){  t1[i][j] += d; t2[i][j] += x\*d;  t3[i][j] += y\*d; t4[i][j] += x\*y\*d;  }  }  int sum(int x,int y){  int ans = 0;  for(int i=x;i>0;i-=lowbit(i))  for(int j=y;j>0;j-=lowbit(j))  ans += (x+1)\*(y+1)\*t1[i][j] - (y+1)\*t2[i][j] - (x+1)\*t3[i][j] + t4[i][j];  return ans;  }  int main(){  char ch[2]; scanf("%s",ch);  scanf("%d%d",&n,&m);  while(scanf("%s",ch)!=EOF){  int a,b,c,d,delta; scanf("%d%d%d%d",&a,&b,&c,&d);  if(ch[0]=='L'){  scanf("%d",&delta);  update(a, b, delta); update(c+1,d+1, delta);  update(a, d+1,-delta); update(c+1,b, -delta);  }  else printf("%d\n",sum(c,d)+sum(a-1,b-1)-sum(a-1,d)-sum(c,b-1));  }  return 0;  } |

|  |
| --- |
| 例4-7. 逆序对 洛谷 P1908 |

作为对比，下面先给出用归并排序求解逆序对的代码，解释见第2章的“2.9分治法”。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | //用归并排序求解洛谷P1908  #include<bits/stdc++.h>  const int N = 5e5+5;  int a[N],tmp[N]; //a[]是原数组，b[]用于合并两半部分  long long ans=0; //记录逆序对数量  void Merge(int L, int mid, int R){ //合并  int i=L,j=mid+1,t=L;  while(i<=mid && j<=R) {  if(a[i]>a[j]){  ans += mid-i+1; //记录逆序对数量。去掉这一行，就是纯的归并排序  tmp[t++]=a[j++];  }  else tmp[t++]=a[i++];  }  //其中一半已经处理完。另一半还没有，它剩下的都是有序的，直接copy  while(i<=mid) tmp[t++]=a[i++];  while(j<=R) tmp[t++]=a[j++];  for(i=L;i<=R;i++) a[i]=tmp[i]; //把排好序的b[]复制回去  }  void Mergesort(int L,int R) { //分治  if(L>=R) return;  int mid= L + (R-L)/2; //比这样写好：int mid=(L+R)/2; 因为L+R可能溢出  Mergesort(L,mid); //左半  Mergesort(mid+1,R); //右半  Merge(L,mid,R); //合并  }  int main() {  int n; scanf("%d",&n);  for(int i=1;i<=n;i++) scanf("%d",&a[i]);  Mergesort(1,n); //归并排序，并统计逆序对  printf("%lld\n",ans);  return 0;  } |

下面是洛谷P1908的代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28 | //lowbit(x)，update()，sum()的代码前面已给出  const int N = 500010;  int tree[N],rank[N],n; //注：rank是C++的保留字，如果加了using namespace std，编译通不过  //在需要用c++定义的函数和变量时，例如sort，这样写：std::sort()  struct point{ int num,val;}a[N];  bool cmp(point x,point y){  if(x.val == y.val) return x.num < y.num; //如果相等，让先出现的更小  return x.val < y.val;  }  int main(){  scanf("%d",&n);  for(int i=1;i<=n;i++) {  scanf("%d",&a[i].val);  a[i].num = i; //记录顺序，用于离散化  }  sort(a+1,a+1+n,cmp); //排序  for(int i=1;i<=n;i++) rank[a[i].num]=i; //离散化，得到新的数字序列rank[]  long long ans=0;  /\*for(int i=1;i<=n;i++){ //正序处理  update(rank[i],1);  ans += i-sum(rank[i]); }\*/  for(int i=n;i>0;--i){ //倒序处理  update(rank[i],1);  ans += sum(rank[i]-1);  }  printf("%lld",ans);  return 0;  } |
| 例4-8. I Hate It hdu 1754 | | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #include<bits/stdc++.h>  using namespace std;  const int N = 2e5+10;  int n,m,a[N],tree[N];  int lowbit(int x){return x&(-x);}  void update1(int x,int value){;} //代码在前面  int query1(int L,int R){;} //代码在前面  int main(){  while(~scanf("%d%d",&n,&m)) {  memset(tree,0,sizeof(tree));  for(int i=1; i<=n; i++){ scanf("%d",&a[i]); update1(i,a[i]); }  while(m--){  char s[5];int A,B; scanf("%s%d%d",s,&A,&B);  if(s[0]=='Q') printf("%d\n",query1(A,B));  else{ a[A]=B; update1(A,B);}  }  }  return 0;  } | |

4.3.2 区间查询

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72 | //洛谷 P3372，线段树，区间修改 + 区间查询  #include<bits/stdc++.h>  using namespace std;  #define ll long long  const int N = 1e5 + 10;  ll a[N]; //记录数列的元素，从a[1]开始  ll tree[N<<2]; //tree[i]：第i个结点的值，表示一个线段区间的值，例如最值、区间和  ll tag[N<<2]; //tag[i]：第i个结点的lazy-tag，统一记录这个区间的修改  ll ls(ll p){ return p<<1; } //定位左儿子：p\*2  ll rs(ll p){ return p<<1|1;} //定位右儿子：p\*2 + 1  void push\_up(ll p){ //从下往上传递区间值  tree[p] = tree[ls(p)] + tree[rs(p)];  //本题是区间和。如果求最小值，改为：tree[p] = min(tree[ls(p)], tree[rs(p)]);  }  void build(ll p,ll pl,ll pr){ //建树。p是结点编号，它指向区间[pl, pr]  tag[p] = 0; //lazy-tag标记  if(pl==pr){tree[p]=a[pl]; return;} //最底层的叶子，赋值  ll mid = (pl+pr) >> 1; //分治：折半  build(ls(p),pl,mid); //左儿子  build(rs(p),mid+1,pr); //右儿子  push\_up(p); //从下往上传递区间值  }  void addtag(ll p,ll pl,ll pr,ll d){ //给结点p打tag标记，并更新tree  tag[p] += d; //打上tag标记  tree[p] += d\*(pr-pl+1); //计算新的tree  }  void push\_down(ll p,ll pl,ll pr){ //不能覆盖时，把tag传给子树  if(tag[p]){ //有tag标记，这是以前做区间修改时留下的  ll mid = (pl+pr)>>1;  addtag(ls(p),pl,mid,tag[p]); //把tag标记传给左子树  addtag(rs(p),mid+1,pr,tag[p]); //把tag标记传给右子树  tag[p]=0; //p自己的tag被传走了，归0  }  }  void update(ll L,ll R,ll p,ll pl,ll pr,ll d){ //区间修改：把[L, R]内每个元素加上d  if(L<=pl && pr<=R){ //完全覆盖，直接返回这个结点，它的子树不用再深入了  addtag(p, pl, pr,d); //给结点p打tag标记，下一次区间修改到p时会用到  return;  }  push\_down(p,pl,pr); //如果不能覆盖，把tag传给子树  ll mid=(pl+pr)>>1;  if(L<=mid) update(L,R,ls(p),pl,mid,d); //递归左子树  if(R>mid) update(L,R,rs(p),mid+1,pr,d); //递归右子树  push\_up(p); //更新  }  ll query(ll L,ll R,ll p,ll pl,ll pr){  //查询区间[L,R]；p是当前结点（线段）的编号，[pl,pr]是结点p表示的线段区间  if(pl>=L && R >= pr) return tree[p]; //完全覆盖，直接返回  push\_down(p,pl,pr); //不能覆盖，递归子树  ll res=0;  ll mid = (pl+pr)>>1;  if(L<=mid) res+=query(L,R,ls(p),pl,mid); //左子结点有重叠  if(R>mid) res+=query(L,R,rs(p),mid+1,pr); //右子结点有重叠  return res;  }  int main(){  ll n, m; scanf("%lld%lld",&n,&m);  for(ll i=1;i<=n;i++) scanf("%lld",&a[i]);  build(1,1,n); //建树  while(m--){  ll q,L,R,d; scanf("%lld",&q);  if (q==1){ //区间修改：把[L,R]的每个元素加上d  scanf("%lld%lld%lld",&L,&R,&d);  update(L,R,1,1,n,d);  }  else { //区间询问：[L,R]的区间和  scanf("%lld%lld",&L,&R);  printf("%lld\n",query(L,R,1,1,n));  }  }  return 0;  } |

4.3.5 区间最值和区间历史最值

|  |  |  |
| --- | --- | --- |
| 例4-14. Gorgeous Sequence hdu 5306 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | //代码改写自：https://blog.csdn.net/nbl97/article/details/76696784  #include<bits/stdc++.h>  using namespace std;  #define ll long long  const int N = 1e6 + 10;  ll sum[N<<2], ma[N<<2], se[N<<2], num[N<<2]; //num:区间的最大值个数  ll ls(ll p){ return p<<1; }  ll rs(ll p){ return p<<1|1;}  void pushup(int p) { //从下往上传递  sum[p] = sum[ls(p)] + sum[rs(p)]; //传递区间和  ma[p] = max(ma[ls(p)], ma[rs(p)]); //传递区间最大值  if (ma[ls(p)] == ma[rs(p)]) {  se[p] = max(se[ls(p)], se[rs(p)]);  num[p] = num[ls(p)] + num[rs(p)];  }  else {  se[p] = max(se[ls(p)],se[rs(p)]);  se[p] = max(se[p],min(ma[ls(p)], ma[rs(p)]));  num[p] = ma[ls(p)] > ma[rs(p)] ? num[ls(p)] : num[rs(p)];  }  }  void build(int p, int pl, int pr) {  if (pl == pr) { //叶子  scanf("%lld", &sum[p]);  ma[p] = sum[p]; se[p] = -1; num[p] = 1;  return;  }  ll mid = (pl+pr) >> 1;  build(ls(p),pl,mid);  build(rs(p),mid+1,pr);  pushup(p);  }  void addtag(int p, int x) {  if (x >= ma[p]) return;  sum[p] -= num[p] \* (ma[p] - x);  ma[p] = x;  }  void pushdown(int p) {  addtag(ls(p), ma[p]); //把标记传给左子树  addtag(rs(p), ma[p]); //把标记传给左子树  }  void update(int L, int R, int p, int pl, int pr, int x) {  if (x >= ma[p]) return; //情况（1）  if (L<=pl && pr<=R && se[p] < x) { addtag(p, x); return;} //情况（2）  pushdown(p); //情况（3）  ll mid = (pl+pr) >> 1;  if (L<=mid) update(L, R, ls(p), pl, mid, x);  if (R>mid) update(L, R, rs(p), mid+1, pr, x);  pushup(p);  }  int queryMax(int L, int R, int p, int pl, int pr) {  if (pl>=L && R >= pr) return ma[p];  pushdown(p);  int res = 0;  ll mid = (pl+pr) >> 1;  if (L<=mid) res = queryMax(L, R, ls(p), pl, mid);  if (R>mid) res = max(res, queryMax(L, R, rs(p), mid+1, pr));  return res;  }  ll querySum(int L, int R, int p, int pl, int pr) {  if (L <= pl && R >= pr) return sum[p];  pushdown(p);  ll res = 0;  ll mid = (pl+pr) >> 1;  if (L<=mid) res += querySum(L, R, ls(p), pl, mid);  if (R>mid) res += querySum(L, R, rs(p), mid+1, pr);  return res;  }  int main(){  int T; scanf("%d", &T);  while (T--) {  int n,m; scanf("%d%d", &n, &m);  build(1, 1, n);  while (m--) {  int q, L, R, x; scanf("%d%d%d", &q, &L, &R);  if (q == 0){ scanf("%d", &x); update(L, R, 1, 1, n, x);}  if (q == 1) printf("%d\n", queryMax(L, R, 1, 1, n));  if (q == 2) printf("%lld\n", querySum(L, R, 1, 1, n));  }  }  return 0;  } |

4.3.6 区间合并

|  |  |  |
| --- | --- | --- |
| 例4-16. Tunnel Warfare hdu 1540 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | #include<bits/stdc++.h>  using namespace std;  const int N = 50010;  int ls(int p){ return p<<1; }  int rs(int p){ return p<<1|1;}  int tree[N<<2], pre[N<<2], suf[N<<2]; //tree:记录元素的值；pre:前缀1的个数；suf：后缀1的个数  int history[N]; //记录村庄被毁的历史  void push\_up(int p,int len){ //len是结点p的长度  pre[p]=pre[ls(p)]; //父结点接收子结点的前缀信息  suf[p]=suf[rs(p)];  if(pre[ls(p)]==(len-(len>>1))) pre[p]=pre[ls(p)]+pre[rs(p)]; //左儿子都是1  if(suf[rs(p)]==(len>>1)) suf[p]=suf[ls(p)]+suf[rs(p)]; //右儿子都是1  }  void build(int p, int pl,int pr){  if(pl==pr){tree[p]=pre[p]=suf[p]=1; return;}  int mid = (pl+pr) >> 1;  build(ls(p),pl,mid);  build(rs(p),mid+1,pr);  push\_up(p,pr-pl+1);  }  void update(int x, int c, int p, int pl, int pr){  if(pl==pr){ tree[p]=suf[p]=pre[p]=c; return; } //更新叶子结点信息  int mid=(pl+pr)>>1;  if(x<=mid) update(x,c,ls(p),pl,mid);  else update(x,c,rs(p),mid+1,pr);  push\_up(p,pr-pl+1);  }  int query(int x,int p,int pl,int pr){  if(pl==pr) return tree[p]; //返回叶子的值  int mid=(pl+pr)>>1;  if(x<=mid){ //左子树  if(x + suf[ls(p)] > mid) return suf[ls(p)] + pre[rs(p)];  else return query(x,ls(p),pl,mid);  }  else{ //右子树  if(mid + pre[rs(p)] >= x) return pre[rs(p)] + suf[ls(p)];  else return query(x,rs(p),mid+1,pr);  }  }  int main(){  int n,m,x,tot;  while(scanf("%d%d",&n,&m)>0) {  build(1, 1,n);  tot = 0;  while(m--){  char op[10]; scanf("%s",op);  if(op[0]=='Q'){scanf("%d",&x); printf("%d\n",query(x,1,1,n));}  else if(op[0]=='D'){  scanf("%d",&x);  history[++tot]=x; //记录毁灭的历史  update(x,0,1,1,n);  }  else {x=history[tot--]; update(x,1,1,1,n); } //重建  }  }  } |

4.3.7 扫描线

|  |  |  |
| --- | --- | --- |
| 例4-19. Atlantis hdu 1542 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65 | //代码改写自：https://blog.csdn.net/narcissus2\_/article/details/88418870  #include<bits/stdc++.h>  using namespace std;  int ls(int p){ return p<<1; }  int rs(int p){ return p<<1|1;}  const int N = 20005;  int Tag[N]; //标志：线段是否有效，能否用于计算宽度  double length[N]; //存放区间i的总宽度  double xx[N]; //存放x坐标值，下标用lower\_bound查找  struct ScanLine{ //定义扫描线  double y; //边的y坐标  double right\_x,left\_x; //边的x坐标:右、左  int inout; //入边为1，出边为-1  ScanLine(){}  ScanLine(double y,double x2,double x1,int io):  y(y),right\_x(x2),left\_x(x1),inout(io){}  }line[N];  bool cmp(ScanLine &a,ScanLine &b) { return a.y<b.y; } //y坐标排序  void pushup(int p,int pl,int pr){ //从下往上传递区间值  if(Tag[p]) length[p] = xx[pr]-xx[pl];  //结点的Tag为正，这个线段对计算宽度有效。计算宽度  else if(pl+1 == pr) length[p] = 0; //叶子结点没有宽度  else length[p] = length[ls(p)] + length[rs(p)];  }  void update(int L,int R,int io,int p,int pl,int pr){  if(L<=pl && pr<=R){ //完全覆盖  Tag[p] += io; //结点的标志，用来判断能否用来计算宽度  pushup(p,pl,pr);  return;  }  if(pl+1 == pr) return; //叶子结点  int mid = (pl+pr) >> 1;  if(L<=mid) update(L,R,io,ls(p),pl,mid);  if(R>mid) update(L,R,io,rs(p),mid,pr); //注意不是mid+1  pushup(p,pl,pr);  }  int main(){  int n, t = 0;  while(scanf("%d",&n),n){  int cnt = 0; //边的数量，包括入边和出边  while(n--){  double x1,x2,y1,y2; scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);//输入一个矩形  line[++cnt] = ScanLine(y1,x2,x1,1); //给入边赋值  xx[cnt] = x1; //记录x坐标  line[++cnt] = ScanLine(y2,x2,x1,-1); //给出边赋值  xx[cnt] = x2; //记录x坐标  }  sort(xx+1,xx+cnt+1); //对所有边的x坐标排序  sort(line+1,line+cnt+1,cmp); //对扫描线按y轴方向从低到高排序  int num = unique(xx+1,xx+cnt+1)-(xx+1); //离散化：用unique去重，返回个数  memset(Tag,0,sizeof(Tag));  memset(length,0,sizeof(length));  double ans = 0;  for(int i=1;i<=cnt;++i) { //扫描所有入边和出边  int L,R;  ans += length[1]\*(line[i].y-line[i-1].y);//累加当前扫描线的面积=宽\*高  L = lower\_bound(xx+1,xx+num+1,line[i].left\_x)-xx;  //x坐标离散化：用相对位置代替坐标值  R = lower\_bound(xx+1,xx+num+1,line[i].right\_x)-xx;  update(L,R,line[i].inout,1,1,num);  }  printf("Test case #%d\nTotal explored area: %.2f\n\n",++t,ans);  }  return 0;  } |

下面是“矩形周长并”的模板题。

|  |  |  |
| --- | --- | --- |
| 例4-20. Picture hdu1828 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67 | //代码改写自：https://blog.csdn.net/qq3434569/article/details/78220821  #include<bits/stdc++.h>  using namespace std;  int ls(int p){ return p<<1; }  int rs(int p){ return p<<1|1;}  const int N = 200005;  struct ScanLine {  int l, r, h, inout; //inout=1 下边, inout=-1 上边  ScanLine() {}  ScanLine(int a, int b, int c, int d) :l(a), r(b), h(c), inout(d) {}  }line[N];  bool cmp(ScanLine &a, ScanLine &b) { return a.h<b.h; } //y坐标排序  bool lbd[N<<2], rbd[N<<2]; //标记这个结点的左右两个端点是否被覆盖（0表示没有，1表示有）  int num[N << 2]; //这个区间有多少条独立的边  int Tag[N << 2]; //标记这个结点是否有效  int length[N << 2]; //这个区间的有效宽度  void pushup(int p, int pl, int pr) {  if (Tag[p]) { //结点的Tag为正，这个线段对计算宽度有效  lbd[p] = rbd[p] = 1;  length[p] = pr - pl + 1;  num[p] = 1; //每条边有两个端点  }  else if (pl == pr) length[p]=num[p]=lbd[p]=rbd[p]=0;//叶子结点  else {  lbd[p] = lbd[ls(p)]; //和左儿子共左端点  rbd[p] = rbd[rs(p)]; //和右儿子共右端点  length[p] = length[ls(p)] + length[rs(p)];  num[p] = num[ls(p)] + num[rs(p)];  if (lbd[rs(p)] && rbd[ls(p)]) num[p] -= 1; //合并边  }  }  void update(int L, int R, int io, int p, int pl, int pr) {  if(L<=pl && pr<=R){ //完全覆盖  Tag[p] += io;  pushup(p, pl, pr);  return;  }  int mid = (pl + pr) >> 1;  if (L<= mid) update(L, R, io, ls(p), pl, mid);  if (mid < R) update(L, R, io, rs(p), mid+1, pr);  pushup(p, pl, pr);  }  int main() {  int n;  while (~scanf("%d", &n)) {  int cnt = 0;  int lbd = 10000, rbd = -10000;  for (int i = 0; i < n; i++) {  int x1,y1,x2,y2; scanf("%d%d%d%d", &x1,&y1,&x2,&y2); //输入矩形  lbd = min(lbd, x1); //横线最小x坐标  rbd = max(rbd, x2); //横线最大x坐标  line[++cnt] = ScanLine(x1, x2, y1, 1); //给入边赋值  line[++cnt] = ScanLine(x1, x2, y2, -1); //给出边赋值  }  sort(line+1, line + cnt+1, cmp); //排序。数据小，不用离散化  int ans = 0, last = 0; //last：上一次总区间被覆盖长度  for (int i = 1; i <= cnt ; i++){ //扫描所有入边和出边  if (line[i].l < line[i].r)  update(line[i].l, line[i].r-1, line[i].inout, 1, lbd, rbd-1);  ans += num[1]\*2 \* (line[i + 1].h - line[i].h); //竖线  ans += abs(length[1] - last); //横线  last = length[1];  }  printf("%d\n", ans);  }  return 0;  } |

4.3.8 二维线段树（树套树）

|  |  |  |
| --- | --- | --- |
| 例4-21. Luck and Love hdu 1823 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81 | //代码改写自https://www.cnblogs.com/ftae/p/7739512.html  #include<bits/stdc++.h>  using namespace std;  int ls(int p){ return p<<1; }  int rs(int p){ return p<<1|1;}  int n=1000, s[1005][4005]; //s[i][j]：身高区间i，活泼区间j的最大缘分  void subBuild(int xp, int p, int pl, int pr) { //建立第二维线段树：活泼度线段树  s[xp][p] = -1;  if(pl == pr) return;  int mid=(pl+pr)>>1;  subBuild(xp, ls(p), pl, mid);  subBuild(xp, rs(p), mid + 1, pr);  }  void build(int p,int pl, int pr) { //建立第一维线段树：身高线段树  subBuild(p, 1, 0, n);  if(pl == pr) return;  int mid=(pl+pr)>>1;  build(ls(p),pl, mid);  build(rs(p),mid + 1, pr);  }  void subUpdate(int xp, int y, int c, int p, int pl, int pr) {//更新第二维线段树  if(pl == pr && pl == y) s[xp][p] = max(s[xp][p], c);  else {  int mid=(pl+pr)>>1;  if(y <= mid) subUpdate(xp, y, c, ls(p), pl, mid);  else subUpdate(xp, y, c, rs(p), mid + 1, pr);  s[xp][p] = max(s[xp][ls(p)], s[xp][rs(p)]);  }  }  void update(int x, int y, int c, int p, int pl, int pr){ //更新第一维线段树：身高x  subUpdate(p, y, c, 1, 0, n); //更新第二维线段树：活泼度y  if(pl != pr) {  int mid=(pl+pr)>>1;  if(x <= mid) update(x, y, c, ls(p), pl, mid);  else update(x, y, c, rs(p), mid + 1, pr);  }  }  int subQuery(int xp, int yL, int yR, int p, int pl, int pr) { //查询第二维线段树  if(yL <= pl && pr <= yR) return s[xp][p];  else {  int mid=(pl+pr)>>1;  int res = -1;  if(yL <= mid) res = subQuery(xp, yL, yR, ls(p), pl, mid);  if(yR > mid) res = max(res, subQuery(xp, yL, yR, rs(p), mid + 1, pr));  return res;  }  }  int query(int xL, int xR, int yL, int yR, int p, int pl, int pr) {//查询第一维线段树  if(xL <= pl && pr <= xR) return subQuery(p, yL, yR, 1, 0, n);  //满足身高区间时，查询活泼度区间  else { //当前结点不满足身高区间  int mid = (pl+pr)>>1;  int res = -1;  if(xL <= mid) res = query(xL, xR, yL, yR, ls(p), pl, mid);  if(xR > mid) res = max(res, query(xL, xR, yL, yR, rs(p), mid + 1, pr));  return res;  }  }  int main(){  int t;  while(scanf("%d", &t) && t) {  build(1,100, 200);  while(t--) {  char ch[2]; scanf("%s", ch);  if(ch[0] == 'I') {  int h;double c, d; scanf("%d%lf%lf", &h, &c, &d);  update(h, c \* 10, d \* 10, 1, 100, 200);  } else {  int xL, xR, yL, yR; double c,d;  scanf("%d%d%lf%lf", &xL, &xR, &c, &d);  yL = c \* 10, yR = d \* 10; //转整数  if(xL > xR) swap(xL, xR);  if(yL > yR) swap(yL, yR);  int ans = query(xL, xR, yL, yR, 1,100, 200); //x：身高，y：活泼度  if(ans == -1) printf("-1\n");  else printf("%.1f\n", ans / 10.0);  }  }  }  return 0;  } |

4.4.2 区间第k大/小问题

|  |  |
| --- | --- |
| 例4-22. 主席树 洛谷 P3834 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | //洛谷 P3834代码，改写自：https://www.luogu.com.cn/problem/solution/P3834  #include <bits/stdc++.h>  using namespace std ;  const int N = 200010;  int cnt = 0; //用cnt标记可以使用的新结点  int a[N], b[N], root[N]; //a[]是原数组,b[]是排序后数组，root[i]记录第i棵线段树的根结点编号  struct{ //定义结点  int L, R, sum; //L左儿子, R右儿子，sum[i]是结点i的权值（即图中圆圈内的数字）  }tree[N<<5]; // <<4是乘16倍，不够用；<<5差不多够用  int build(int pl, int pr){ //初始化一棵空树，实际上无必要  int rt = ++ cnt; //cnt为当前结点编号  tree[rt].sum = 0;  int mid=(pl+pr)>>1;  if (pl < pr){  tree[rt].L = build(pl, mid);  tree[rt].R = build(mid+1, pr);  }  return rt; //返回当前结点的编号  }  int update(int pre, int pl, int pr, int x){ //建一棵只有logn个结点的新线段树  int rt = ++cnt; //新的结点，下面动态开点  tree[rt].L = tree[pre].L; //该结点的左右儿子初始化为前一棵树相同位置结点的左右儿子  tree[rt].R = tree[pre].R;  tree[rt].sum = tree[pre].sum + 1; //插了1个数，在前一棵树的相同结点加1  int mid = (pl+pr)>>1;  if (pl < pr){ //从根结点往下建logn个结点  if (x <= mid) //x出现在左子树，修改左子树  tree[rt].L = update(tree[pre].L, pl, mid, x);  else //x出现在右子树，修改右子树  tree[rt].R = update(tree[pre].R, mid+1, pr, x);  }  return rt; //返回当前分配使用的新结点的编号  }  int query(int u, int v, int pl, int pr, int k){ //查询区间[u,v]第k小  if (pl == pr) return pl; //到达叶子结点，找到第k小，pl是结点编号，答案是b[pl]  int x = tree[tree[v].L].sum - tree[tree[u].L].sum; //线段树相减  int mid = (pl+pr)>>1;  if (x >= k) //左儿子数字大于等于k时，说明第k小的数字在左子树  return query(tree[u].L, tree[v].L, pl, mid, k);  else //否则在右子树找第k-x小的数字  return query(tree[u].R, tree[v].R, mid+1, pr, k-x);  }  int main(){  int n, m; scanf("%d%d", &n, &m);  for (int i=1; i<=n; i++){ scanf("%d", &a[i]); b[i]=a[i]; }  sort(b+1, b+1+n); //对b排序  int size = unique(b+1, b+1+n)-b-1; //size等于b数组中不重复的数字的个数  //root[0] = build(1, size); //初始化一棵包含size个元素的空树，实际上无必要  for (int i = 1; i <= n; i ++){ //建n棵线段树  int x = lower\_bound(b+1, b+1+size, a[i]) - b;  //找等于a[i]的b[x]。x是离散化后a[i]对应的值  root[i] = update(root[i-1], 1, size, x);  //建第i棵线段树，root[i]是第i棵线段树的根结点  }  while (m--){  int x, y, k; scanf("%d%d%d", &x, &y, &k);  int t = query(root[x-1], root[y], 1, size, k);  //第y棵线段树减第x-1棵线段树，就是区间[x,y]的线段树  printf("%d\n", b[t]);  }  return 0;  } | |

4.5.2 基础莫队算法

|  |  |  |
| --- | --- | --- |
| 例4-29. HH项链 洛谷 1972 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53 | #include<bits/stdc++.h>  using namespace std;  const int N = 1e6;  struct node{ //离线记录查询操作  int L, R, k; //k：查询操作的原始顺序  }q[N];  int pos[N];  int ans[N];  int cnt[N]; //cnt[i]: 统计数字i出现了多少次  int a[N];  bool cmp(node a, node b){  //按块排序，就是莫队算法：  if(pos[a.L] != pos[b.L]) //按L所在的块排序，如果块相等，再按R排序  return pos[a.L] < pos[b.L];  if(pos[a.L] & 1) return a.R > b.R; //奇偶性优化，如果删除这一句，性能差一点  return a.R < b.R;  /\*如果不按块排序，而是直接L、R排序，就是普通暴力法：  if(a.L==b.L) return a.R < b.R;  return a.L < b.L; \*/  }  int ANS = 0;  void add(int x){ //扩大区间时（L左移或R右移），增加数x出现的次数  cnt[a[x]]++;  if(cnt[a[x]]==1) ANS++; //这个元素第1次出现  }  void del(int x){ //缩小区间时（L右移或R左移），减少数x出现的次数  cnt[a[x]]--;  if(cnt[a[x]]==0) ANS--; //这个元素消失了  }  int main(){  int n; scanf("%d",&n);  int block = sqrt(n); //每块的大小  for(int i=1;i<=n;i++){  scanf("%d",&a[i]); //读第i个元素  pos[i]=(i-1)/block + 1; //第i个元素所在的块  }  int m; scanf("%d",&m);  for(int i=1;i<=m;i++){ //读取所有m个查询，离线处理  scanf("%d%d",&q[i].L, &q[i].R);  q[i].k = i; //记录查询的原始顺序  }  sort(q+1, q+1+m, cmp); //对所有查询排序  int L=1, R=0; //左右指针的初始值。思考为什么？  for(int i=1;i<=m;i++){  while(L<q[i].L) del(L++); //{del(L); L++;} //缩小区间：L右移  while(R>q[i].R) del(R--); //{del(R); R--;} //缩小区间：R左移  while(L>q[i].L) add(--L); //{L--; add(L);} //扩大区间：L左移  while(R<q[i].R) add(++R); //{R++; add(R);} //扩大区间：R右移  ans[q[i].k] = ANS;  }  for(int i=1;i<=m;i++) printf("%d\n",ans[i]); //按原顺序打印结果  return 0;  } |

4.6 块状链表

|  |  |  |
| --- | --- | --- |
| 例4-32. 文本编辑器 洛谷 P4008 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | //改写自：https://www.luogu.com.cn/blog/EndSaH/solution-p4008  #include<bits/stdc++.h>  using namespace std;  int block = 2500; //一个块的标准大小 = sqrt(n)  list<vector<char> > List; //整体是链表，链表的每个元素是一个块  typedef list<vector<char> >::iterator it;  it Find(int &pos) { //返回块，并更新x为这个块中的位置  for (it i = List.begin(); ;i++) { //逐个找链表上的每个块  if(i == List.end() || pos <= i->size()) return i;  pos -= i->size(); //每经过一个块，就更新x  }  }  void Output(int L, int R) { // [L, R)  it L\_block = Find(L), R\_block = Find(R);  for (it it1 = L\_block; ; it1++){ //打印每个块  int a; it1 == L\_block ? a=L : a=0; //一个块的起点  int b; it1 == R\_block ? b=R : b=it1->size(); //块的终点  for (int i = a; i < b; i++) putchar(it1->at(i));  if(it1 == R\_block) break; //迭代器it不能用 <= ，只有 == 和 !=  }  putchar('\n');  }  it Next(it x){return ++x; } //返回下一个块  void Merge(it x) { //合并块x和块x+1  x->insert(x->end(), Next(x)->begin(), Next(x)->end());  List.erase(Next(x));  }  void Split(it x, int pos){ //把第x个块在这个块的pos处分成2块  if (pos == x->size()) return; //pos在这个块的末尾  List.insert(Next(x), vector<char>(x->begin() + pos, x->end()));  //把pos后面的部分划给下一个块  x->erase(x->begin() + pos, x->end()); //删除划出的部分  }  void Update(){ //把每个块重新划成等长的块  for (it i = List.begin(); i != List.end(); i++){  while (i->size() >= (block << 1)) //如果块大于2个block，分开  Split(i, i->size() - block);  while (Next(i) != List.end() && i->size() + Next(i)->size() <= block)  Merge(i); //如果块+下一个块小于block，合并  while (Next(i) != List.end() && Next(i)->empty()) //删除最后的空块  List.erase(Next(i));  }  }  void Insert(int pos, const vector<char>& ch){  it curr = Find(pos);  if (!List.empty()) Split(curr, pos); //把一个块拆为两个  List.insert(Next(curr), ch); //把字符串插到两个块中间  Update();  }  void Delete(int L, int R) { // [L, R)  it L\_block, R\_block;  L\_block = Find(L); Split(L\_block, L);  R\_block = Find(R); Split(R\_block, R);  R\_block++;  while(Next(L\_block) != R\_block) List.erase(Next(L\_block));  Update();  }  int main(){  vector<char> ch; int len, pos, n;  cin >> n;  while (n--) {  char opt[7]; cin >> opt;  if(opt[0]=='M') cin >> pos;  if(opt[0]=='I'){  ch.clear(); cin >> len; ch.resize(len);  for (int i = 0; i < len; i++){  ch[i] = getchar();  while(ch[i]<32||ch[i]>126) ch[i]=getchar();  } //读一个合法字符  Insert(pos, ch); //把字符串插入到链表中  }  if(opt[0]=='D'){ cin >> len; Delete(pos, pos + len); }  if(opt[0]=='G'){ cin >> len; Output(pos, pos + len); }  if(opt[0]=='P') pos--;  if(opt[0]=='N') pos++;  }  return 0;  } |

4.7.1 树的重心

|  |  |  |
| --- | --- | --- |
| 例4-33. Godfather poj 3107 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52 | // poj 3107的代码（链式前向星存树）  #include<cstdio>  #include<algorithm>  using namespace std;  const int N = 50005; //最大结点数  struct Edge{ int to, next;} edge[N<<1]; //两倍：u-v, v-u  int head[N], cnt = 0;  void init(){ //链式前向星：初始化  for(int i=0; i<N; ++i){  edge[i].next = -1;  head[i] = -1;  }  cnt = 0;  }  void addedge(int u,int v){ //链式前向星：加边u-v  edge[cnt].to = v;  edge[cnt].next = head[u];  head[u] = cnt++;  }  int n;  int d[N], ans[N], num=0, maxnum=1e9; //d[u]: 以u为根的子树的结点数量  void dfs(int u,int fa){  d[u] = 1; //递归到最底层时，结点数加1  int tmp = 0;  for(int i=head[u]; ~i; i=edge[i].next){ //遍历u的子结点。~i也可以写成i!=-1  int v = edge[i].to; //v是一个子结点  if(v == fa) continue; //不递归父亲  dfs(v,u); //递归子结点，计算v这个子树的结点数量  d[u] += d[v]; //计算以u为根的结点数量  tmp = max(tmp,d[v]); //记录u的最大子树的结点数量  }  tmp = max(tmp, n-d[u]); //tmp = u的最大连通块的结点数  //以上计算出了u的最大连通块  //下面统计疑似教父。如果一个结点的最大连通块比其他结点的都小，它是疑似教父  if(tmp < maxnum){ //一个疑似教父  maxnum = tmp; //更新“最小的”最大连通块  num = 0;  ans[++num] = u; //把教父记录在第1个位置  }  else if(tmp == maxnum) ans[++num] = u; //疑似教父有多个，记录在后面  }  int main(){  scanf("%d",&n);  init();  for(int i=1; i<n; i++){  int u, v; scanf("%d %d", &u, &v);  addedge(u,v); addedge(v,u);  }  dfs(1,0);  sort(ans+1, ans+1+num);  for(int i=1;i<=num;i++) printf("%d ",ans[i]);  } |

4.7.2 树的直径

|  |
| --- |
| 例4-34. 树的直径 |

**1. 做两次DFS（或BFS）**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | #include<bits/stdc++.h>  using namespace std;  const int N=1e5+10;  struct edge{ int to,w;}; //to: 边的终点 w:权值  vector<edge> e[N]; //用邻接表存边  int dist[N]; //记录距离  void dfs(int u,int father,int d){ //用dfs计算从u到每个子结点的距离  dist[u]=d;  for(int i=0;i<e[u].size();i++)  if(e[u][i].to != father) //很关键，这一句保证不回头搜父结点  dfs(e[u][i].to, u, d + e[u][i].w);  }  int main(void){  int n; cin>>n;  for(int i=0;i<n-1;i++){  int a,b,w; cin>>a>>b>>w;  e[a].push\_back({b,w}); //a的邻居是b，边长w  e[b].push\_back({a,w}); //b的邻居是a  }  dfs(1,-1,0); //计算从任意点（这里用1号点）到树上每个结点的距离  int s = 1;  for(int i=1;i<=n;i++) //找最远的结点s, s是直径的一个端点  if(dist[i]>dist[s]) s = i;  dfs(s,-1,0); //从s出发，计算以s为起点，到树上每个结点的距离  int t = 1;  for(int i=1;i<=n;i++) //找直径的另一个端点t  if(dist[i]>dist[t]) t = i;  cout << dist[t]<<endl; //打印树的直径的长度  return 0;  } |

**2. 树形DP**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | #include<bits/stdc++.h>  using namespace std;  const int N=1e5+10;  struct edge{int to,w; }; //to: 边的终点 w:权值  vector<edge> e[N];  int dp[N];  int maxlen = 0;  bool vis[N];  void dfs(int u){  vis[u] = true;  for(int i = 0; i < e[u].size(); ++ i){  int v = e[u][i].to, edge = e[u][i].w;  if(vis[v]) continue; //v已经算过  dfs(v);  maxlen = max(maxlen, dp[u]+ dp[v]+ edge);  //计算max{f[u]}。注意此时dp[u]不包括v这棵子树，下一行才包括  dp[u] = max(dp[u], dp[v] + edge); //计算dp[u]，此时包括了v这棵子树  }  return ;  }  int main(){  int n; cin >> n;  for(int i = 0; i < n-1; i++){  int a, b, w; cin >> a >> b >> w;  e[a].push\_back({b,w}); //a的邻居是b，路的长度w  e[b].push\_back({a,w}); //b的邻居是a  }  dfs(1); //从点1开始DFS  cout << maxlen << endl;  return 0;  } |

4.8.1 倍增法求LCA

|  |
| --- |
| 例4-35. 最近公共祖先 洛谷P3379 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | //洛谷P3379 的倍增代码  #include <bits/stdc++.h>  using namespace std;  const int N=500005;  struct Edge{int to, next;}edge[2\*N]; //链式前向星  int head[2\*N], cnt;  void init(){ //链式前向星：初始化  for(int i=0;i<2\*N;++i){ edge[i].next = -1; head[i] = -1; }  cnt = 0;  }  void addedge(int u,int v){ //链式前向星：加边  edge[cnt].to = v; edge[cnt].next = head[u]; head[u] = cnt++;  } //以上是链式前向星  int fa[N][20], deep[N];  void dfs(int x,int father){ //求x的深度deep[x]和fa[x][]。father是x的父结点。  deep[x] = deep[father]+1; //深度：比父结点深度多1  fa[x][0] = father; //记录父结点  for(int i=1; (1<<i) <= deep[x]; i++) //求fa[][]数组，它最多到根结点  fa[x][i] = fa[fa[x][i-1]][i-1];  for(int i=head[x]; ~i; i=edge[i].next) //遍历结点i的所有孩子。~i可写为i!=-1  if(edge[i].to != father) //邻居：除了父亲，都是孩子  dfs(edge[i].to, x);  }  int LCA(int x,int y){  if(deep[x]<deep[y]) swap(x,y); //让x位于更底层，即x的深度值更大  //（1）把x和y提到相同的深度  for(int i=19;i>=0;i--) //x最多跳19次：2^19 > 500005  if(deep[x]-(1<<i)>=deep[y]) //如果x跳过头了就换个小的i重跳  x = fa[x][i]; //如果x还没跳到y的层，就更新x继续跳  if(x==y) return x; //y就是x的祖先  //（2）x和y同步往上跳，找到LCA  for(int i=19;i>=0;i--) //如果祖先相等，说明跳过头了，换个小的i重跳  if(fa[x][i]!=fa[y][i]){ //如果祖先不等，就更新x、y继续跳  x = fa[x][i]; y = fa[y][i];  }  return fa[x][0]; //最后x位于LCA的下一层，父结点fa[x][0]就是LCA  }  int main(){  init(); //初始化链式前向星  int n,m,root; scanf("%d%d%d",&n,&m,&root);  for(int i=1;i<n;i++){ //读一棵树，用链式前向星存储  int u,v; scanf("%d%d",&u,&v);  addedge(u,v); addedge(v,u);  }  dfs(root,0); //计算每个结点的深度并预处理fa[][]数组  while(m--){  int a,b; scanf("%d%d",&a,&b);  printf("%d\n", LCA(a,b));  }  return 0;  } |

4.8.2 Tarjan求LCA

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | //洛谷P3379的tarjan代码，改写自 blog.csdn.net/Harington/article/details/105901338  #include <bits/stdc++.h>  using namespace std;  const int N=500005;  int fa[N], head[N], cnt, head\_query[N], cnt\_query, ans[N];  bool vis[N];  struct Edge{ int to, next, num;}edge[2\*N], query[2\*N]; //链式前向星  void init(){ //链式前向星：初始化  for(int i=0;i<2\*N;++i){  edge[i].next = -1; head[i] = -1;  query[i].next = -1; head\_query[i] = -1;  }  cnt = 0; cnt\_query = 0;  }  void addedge(int u,int v){ //链式前向星：加边  edge[cnt].to = v;  edge[cnt].next = head[u];  head[u] = cnt++;  }  void add\_query(int x, int y, int num) { //num 第几个查询  query[cnt\_query].to = y;  query[cnt\_query].num = num; //第几个查询  query[cnt\_query].next = head\_query[x];  head\_query[x] = cnt\_query++;  }  int find\_set(int x) { //并查集查询  return fa[x] == x ? x : find\_set(fa[x]);  }  void tarjan(int x){ //tarjan是一个DFS  vis[x] = true;  for(int i=head[x]; ~i; i=edge[i].next){ // ~i可以写为i!=-1  int y = edge[i].to;  if( !vis[y] ) { //遍历子结点  tarjan(y);  fa[y] = x; //合并并查集：把子结点y合并到父结点x上  }  }  for(int i = head\_query[x]; ~i; i = query[i].next){ //查询所有和x有询问关系的y  int y = query[i].to;  if( vis[y]) //如果to被访问过  ans[query[i].num] = find\_set(y); //LCA就是find(y)  }  }  int main () {  init();  memset(vis, 0, sizeof(vis));  int n,m,root; scanf("%d%d%d",&n,&m,&root);  for(int i=1;i<n;i++){ //读n个结点  fa[i] = i; //并查集初始化  int u,v; scanf("%d%d",&u,&v);  addedge(u,v); addedge(v,u); //存边  }  fa[n] = n; //并查集的结点n  for(int i = 1; i <= m; ++i) { //读m个询问  int a, b; scanf("%d%d",&a,&b);  add\_query(a, b, i); add\_query(b, a, i); //存查询  }  tarjan(root);  for(int i = 1; i <= m; ++i) printf("%d\n",ans[i]);  } |

4.8.3 LCA应用

|  |  |
| --- | --- |
| 例4-36. Max Flow P 洛谷P3128 | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | //洛谷P3128，LCA + 树上差分  #include <bits/stdc++.h>  using namespace std;  #define N 50010  struct Edge{int to,next;}edge[2\*N]; //链式前向星  int head[2\*N],D[N],deep[N],fa[N][20],ans,cnt;  void init();  void addedge(int u,int v);  void dfs1(int x,int father);  int LCA(int x,int y); //以上4个函数和“树上的倍增”中洛谷P3379的倍增代码完全一样  void dfs2(int u,int fath){  for (int i=head[u];~i;i=edge[i].next){ //遍历结点i的所有孩子。~i可以写为i!=-1  int e=edge[i].to;  if (e==fath) continue;  dfs2(e,u);  D[u]+=D[e];  }  Ans = max(ans,D[u]);  }  int main(){  init(); //链式前向星初始化  int n,m; scanf("%d%d",&n,&m);  for (int i=1;i<n;++i){  int u,v; scanf("%d%d",&u,&v);  addedge(u,v); addedge(v,u);  }  dfs1(1,0); //计算每个结点的深度并预处理fa[][]数组  for (int i=1; i<=m; ++i){  int a,b; scanf("%d%d",&a,&b);  int lca = LCA(a,b);  D[a]++; D[b]++; D[lca]--; D[fa[lca][0]]--; //树上差分  }  dfs2(1,0); //用差分数组求每个结点的权值  printf("%d\n",ans);  return 0;  } |

4.10.1 树链剖分的概念与LCA

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | //洛谷P3379 的树链剖分代码  #include<bits/stdc++.h>  using namespace std;  const int N=500005;  struct Edge{int to, next;}edge[2\*N]; //链式前向星  int head[2\*N], cnt;  void init(){ //链式前向星：初始化  for(int i=0;i<2\*N;++i){ edge[i].next = -1; head[i] = -1; }  cnt = 0;  }  void addedge(int u,int v){ //链式前向星：加边  edge[cnt].to = v; edge[cnt].next = head[u]; head[u] = cnt++;  }//以上是链式前向星  int deep[N],siz[N],son[N],top[N],fa[N];  void dfs1(int x, int father){  deep[x]=deep[father]+1; //深度：比父结点深度多1  fa[x]=father; //标记x的父亲  siz[x]=1; //标记每个结点的子树大小（包括自己）  for(int i=head[x];~i;i=edge[i].next){  int y=edge[i].to;  if(y!=father){ //邻居：除了父亲，都是孩子  fa[y]=x;  dfs1(y,x);  siz[x] += siz[y]; //回溯后，把x的儿子数加到x身上  if(!son[x] || siz[son[x]]<siz[y]) //标记每个非叶子结点的重儿子  son[x]=y; //x的重儿子是y  }  }  }  void dfs2(int x,int topx){  //id[x] = ++num; //对每个结点新编号，在下一小节用到  top[x] = topx; //x所在链的链头  if(!son[x]) return; //x是叶子，没有儿子，返回  dfs2(son[x],topx); //**先dfs重儿子**，所有重儿子的链头都是topx  for(int i=head[x];~i;i=edge[i].next){ //**再dfs轻儿子**  int y=edge[i].to;  if(y!=fa[x] && y!=son[x])  dfs2(y,y); //每一个轻儿子都有一条以它为链头的重链  }  }  int LCA(int x, int y){  while(top[x]!=top[y]){ //持续往上跳，直到若x和y属于同一条重链  if(deep[top[x]] < deep[top[y]]) swap(x,y); //让x是链头更深的重链  x = fa[top[x]]; //x穿过轻边，跳到上一条重链  }  return deep[x]<deep[y] ? x : y;}  int main(){  init();  int n,m,root; scanf("%d%d%d",&n,&m,&root);  for(int i=1;i<n;i++){  int u,v; scanf("%d%d",&u,&v);  addedge(u,v); addedge(v,u);  }  dfs1(root,0);  dfs2(root,root);  while(m--){  int a,b; scanf("%d%d",&a,&b);  printf("%d\n", LCA(a,b));  }  } |

4.10.2 树链剖分的典型应用

|  |  |  |
| --- | --- | --- |
| 例4-40. 轻重链剖分 洛谷 P3384 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117 | #include<bits/stdc++.h>  using namespace std;  const int N=100000+10;  int n,m,r,mod;  //以下是链式前向星  struct Edge{int to, next;}edge[2\*N];  int head[2\*N], cnt;  void init(); //与前一小节“洛谷P3379树链剖分”的init()一样  void addedge(int u,int v); //与前一小节“洛谷P3379树链剖分”的addedge()一样  //以下是线段树  int ls(int x){ return x<<1; } //定位左儿子：x\*2  int rs(int x){ return x<<1|1;} //定位右儿子：x\*2 + 1  int w[N],w\_new[N]; //w[]、w\_new[]初始点权  int tree[N<<2], tag[N<<2]; //线段树数组、lazy-tag操作  void addtag(int p,int pl,int pr,int d){ //给结点p打tag标记，并更新tree  tag[p] += d; //打上tag标记  tree[p] += d\*(pr-pl+1); tree[p] %= mod; //计算新的tree  }  void push\_up(int p){ //从下往上传递区间值  tree[p] = tree[ls(p)] + tree[rs(p)]; tree[p] %= mod;  }  void push\_down(int p,int pl, int pr){  if(tag[p]){  int mid = (pl+pr)>>1;  addtag(ls(p),pl,mid,tag[p]); //把tag标记传给左子树  addtag(rs(p),mid+1,pr,tag[p]); //把tag标记传给右子树  tag[p] = 0;  }  }  void build(int p,int pl,int pr){ //建线段树  tag[p] = 0;  if(pl==pr){  tree[p] = w\_new[pl]; tree[p] %= mod;  return;  }  int mid = (pl+pr) >> 1;  build(ls(p),pl,mid);  build(rs(p),mid+1,pr);  push\_up(p);  }  void update(int L,int R,int p,int pl,int pr,int d){  if(L<=pl && pr<=R){ addtag(p, pl, pr,d); return; }  push\_down(p,pl,pr);  int mid = (pl+pr) >> 1;  if(L<=mid) update(L,R,ls(p),pl,mid,d);  if(R> mid) update(L,R,rs(p),mid+1,pr,d);  push\_up(p);  }  int query(int L,int R,int p,int pl,int pr){  if(pl>=L && R >= pr) return tree[p] %= mod;  push\_down(p,pl,pr);  int res =0;  int mid = (pl+pr) >> 1;  if(L<=mid) res += query(L,R,ls(p),pl,mid);  if(R> mid) res += query(L,R,rs(p),mid+1,pr);  return res;  }  //以下是树链剖分  int son[N],id[N],fa[N],deep[N],siz[N],top[N];  void dfs1(int x, int father); //与前一小节“洛谷P3379树链剖分”dfs1()一样  int num = 0;  void dfs2(int x,int topx){ //x当前结点，topx当前链的最顶端的结点  id[x] = ++num; //对每个结点新编号  w\_new[num] = w[x]; //把每个点的初始值赋给新编号  top[x]=topx; //记录x的链头  if(!son[x]) return; //x是叶子，没有儿子，返回  dfs2(son[x],topx); //先dfs重儿子  for(int i=head[x];~i;i=edge[i].next){ //再dfs轻儿子  int y=edge[i].to;  if(y!=fa[x] && y!=son[x]) dfs2(y,y);//每个轻儿子都有一条从它自己开始的链  }  }  void update\_range(int x,int y,int z){ //和求LCA(x, y)的过程差不多  while(top[x]!=top[y]){  if(deep[top[x]]<deep[top[y]]) swap(x,y);  update(id[top[x]],id[x],1,1,n,z); //修改一条重链的内部  x = fa[top[x]];  }  if(deep[x]>deep[y]) swap(x,y);  update(id[x],id[y],1,1,n,z); //修改一条重链的内部  }  int query\_range(int x,int y){ //和求LCA(x,y)的过程差不多  int ans=0;  while(top[x]!=top[y]){ //持续往上跳，直到若x和y属于同一条重链  if(deep[top[x]]<deep[top[y]]) swap(x,y); //让x是链头更深的重链  ans += query(id[top[x]],id[x],1,1,n); //加上x到x的链头这一段区间  ans %= mod;  x = fa[top[x]]; //x穿过轻边，跳到上一条重链  }  if(deep[x]>deep[y]) swap(x,y);  //若LCA(x, y) = y，交换x,y,让x更浅，使得id[x] <= id[y]  ans += query(id[x],id[y],1,1,n); //再加上x, y的区间和  return ans % mod;  }  void update\_tree(int x,int k){ update(id[x],id[x]+siz[x]-1,1,1,n,k); }  int query\_tree(int x){ return query(id[x],id[x]+siz[x]-1,1,1,n) % mod; }  int main(){  init(); //链式前向星初始化  scanf("%d%d%d%d",&n,&m,&r,&mod);  for(int i=1;i<=n;i++) scanf("%d",&w[i]);  for(int i=1;i<n;i++){  int u,v; scanf("%d%d",&u,&v);  addedge(u,v); addedge(v,u);  }  dfs1(r,0);  dfs2(r,r);  build(1,1,n); //建线段树  while(m--){  int k,x,y,z; scanf("%d",&k);  switch(k){  case 1:scanf("%d%d%d",&x,&y,&z);update\_range(x,y,z); break;  case 2:scanf("%d%d",&x,&y); printf("%d\n",query\_range(x,y));break;  case 3: scanf("%d%d",&x,&y); update\_tree(x,y); break;  case 4: scanf("%d",&x); printf("%d\n",query\_tree(x)); break;  }  }  } |

4.12.3 例题

|  |
| --- |
| 例4-41. 普通平衡树 洛谷P3369 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135 | //洛谷P3369 替罪羊树  #include<bits/stdc++.h>  using namespace std;  const int N = 1e6+10;  const double alpha = 0.75; //不平衡率。一般用alpha来表示  struct Node{  int ls,rs; //左右儿子  int val; //结点存的数字  int tot; //当前子树占用的空间数量，包括实际存储的结点和被标记删去的点  int size; //子树上实际存储数字的数量  int del; //=1表示这个结点存有数字，=0表示这个点存的数字被删了  }t[N];  int order[N],cnt; //order[]记录拍扁后的结果，即那些存有数字的结点。cnt是数量  int tree\_stack[N],top = 0; //用一个栈来回收和分配可用的结点  int root = 0; //根结点，注意重建过程中根结点会变化  void inorder(int u){ //中序遍历，“拍平”摧毁这棵子树  if(!u) return; //已经到达叶子，退出  inorder(t[u].ls); //先遍历左子树  if(t[u].del) order[++cnt] = u; //如果该结点存有数字，读取它  else tree\_stack[++top] = u; //回收该结点，等待重新分配使用  inorder(t[u].rs); //再遍历右子树  }  void Initnode(int u){ //重置结点的参数  t[u].ls = t[u].rs = 0;  t[u].size = t[u].tot = t[u].del = 1;  }  void Update(int u){  t[u].size = t[t[u].ls].size + t[t[u].rs].size + 1;  t[u].tot = t[t[u].ls].tot + t[t[u].rs].tot + 1;  }  //int rebuild\_num=0; //测试：统计重建次数  void build(int l,int r,int &u){ //把拍扁的子树拎起来，重建  // rebuild\_num++; //测试：统计重建次数  int mid = (l + r) >> 1; //新的根设为中点，使重构出的树尽量平衡  u = order[mid];  if(l == r){Initnode(u); return;} //如果是叶子，重置后返回  if(l < mid) build(l,mid - 1,t[u].ls);//重构左子树  if(l == mid) t[u].ls = 0; //注意这里，不要漏了  build(mid + 1,r,t[u].rs); //重构右子树  Update(u); //更新  }  void rebuild(int &u){ //重建。注意是&u  cnt = 0;  inorder(u); //先拍平摧毁  if(cnt) build(1,cnt,u); //再拎起，重建树  else u = 0; //特判该子树为空的情况  }  bool notbalance(int u){ //判断子树u是否平衡  if((double)t[u].size\*alpha <=(double)max(t[t[u].ls].size,t[t[u].rs].size))  return true; //不平衡了  return false; //还是平衡的  }  void Insert(int &u,int x){ //插入数字x。注意是&u，传回了新的u  if(!u){ //如果结点u为空，直接将x插到这里  u = tree\_stack[top--]; //从栈顶拿出可用的空结点  t[u].val = x; //结点赋值  Initnode(u); //其他参数初始化  return;  }  t[u].size++;  t[u].tot++;  if(t[u].val >= x) Insert(t[u].ls,x); //插到右子树  else Insert(t[u].rs,x); //插到左子树  if(notbalance(u)) rebuild(u); //如果不平衡了，重建这棵子树  }  int Rank(int u,int x){ //排名，x是第几名  if(u==0) return 0;  if(x>t[u].val) return t[t[u].ls].size+ t[u].del + Rank(t[u].rs, x);  return Rank(t[u].ls,x);  }  int kth(int k){ //第k大数是几?  int u = root;  while(u){  if(t[u].del && t[t[u].ls].size + 1 == k) return t[u].val;  else if(t[t[u].ls].size >= k) u = t[u].ls;  else{  k -= t[t[u].ls].size + t[u].del;  u = t[u].rs;  }  }  return t[u].val;  }  void Del\_k(int &u,int k){ //删除排名为k的数  t[u].size--; //要删除的数肯定在这棵子树中，size减1  if(t[u].del && t[t[u].ls].size + 1 == k){  t[u].del = 0; //del=0表示这个点u被删除了，但是还保留位置  return;  }  if(t[t[u].ls].size + t[u].del >= k) Del\_k(t[u].ls,k); //在左子树上  else Del\_k(t[u].rs,k - t[t[u].ls].size - t[u].del); //在右子树上  }  void Del(int x){ //删除值为k的数  Del\_k(root,Rank(root,x)+1); //先找x的排名，然后用Del\_k()删除  if(t[root].tot \* alpha >= t[root].size)  rebuild(root); //如果子树上被删除的结点太多，就重构  }  /\*  void print\_tree(int u){ //测试：打印二叉树，观察  if(u){  cout<<"v="<<t[u].val<<",l="<<t[u].ls<<",r="<<t[u].rs<<endl;  print\_tree(t[u].ls);  print\_tree(t[u].rs);  }  }  int tree\_deep[N]={0},deep\_timer=0,max\_deep=0; //测试  void cnt\_deep(int u){ //测试：计算二叉树的深度  if(u){  tree\_deep[u]=++deep\_timer; //结点u的深度  max\_deep = max(max\_deep,tree\_deep[u]); //记录曾经的最大深度  cnt\_deep(t[u].ls);  cnt\_deep(t[u].rs);  deep\_timer--;  }  } \*/  int main(){  for(int i=N-1;i>=1;i--) tree\_stack[++top] = i; //把所有可用的t[]记录在这个栈里面  int q; cin>>q;  // rebuild\_num = 0;deep\_timer=0;max\_deep=0; //测试  while(q--){  int opt,x; cin >> opt >> x;  switch (opt){  case 1: Insert(root,x); break;  case 2: Del(x); break;  case 3: printf("%d\n",Rank(root,x)+1); break;  case 4: printf("%d\n",kth(x)); break;  case 5: printf("%d\n",kth(Rank(root,x))); break;  case 6: printf("%d\n",kth(Rank(root,x+1)+1)); break;  }  // cout<<">>"<<endl;print\_tree(root);cout<<endl<<"<<"<<endl; //测试：打印二叉树  // cnt\_deep(root); //测试：计算曾经的最大深度  }  // cout<<"rebuild num="<<rebuild\_num<<endl; //测试：打印重建次数  // cout<<"deep="<<max\_deep<<endl; //测试：打印替罪羊树的最大深度  return 0;  } |

4.13.2 基于旋转法的Treap操作

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 | //洛谷P3369，用旋转法实现Treap树  #include<bits/stdc++.h>  using namespace std;  const int M=1e6+10;  int cnt = 0; //t[cnt]: 最新结点的存储位置  struct Node{  int ls,rs; //左右儿子  int key,pri; // key：键值；pri：随机的优先级  int size; //当前结点为根的子树的结点数量，用于求第k大和rank  }t[M]; //tree[]，存树  void newNode(int x){ //初始化一个新结点  cnt++; //从t[1]开始存储结点，t[0]被放弃。若子结点是0，表示没有子结点  t[cnt].size = 1;  t[cnt].ls = t[cnt].rs = 0; //0表示没有子结点  t[cnt].key = x; //key: 键值  t[cnt].pri = rand(); //pri：随机产生，优先级  }  void Update(int u){ //更新以u为根的子树的size  t[u].size = t[t[u].ls].size + t[t[u].rs].size+1;  }  void rotate(int &o,int d){ //旋转，参考图示理解。 d=0右旋，d=1左旋  int k;  if(d==1) { //左旋，把o的右儿子k旋到根部  k=t[o].rs;  t[o].rs=t[k].ls;//图中的x  t[k].ls=o;  }  else { //右旋，把o的左儿子k旋到根部  k=t[o].ls;  t[o].ls=t[k].rs; //图中的x  t[k].rs=o;  }  t[k].size=t[o].size;  Update(o);  o=k; //新的根是k  }  void Insert(int &u,int x){  if(u==0){newNode(x);u=cnt;return;} //递归到了一个空叶子，新建结点  t[u].size++;  if(x>=t[u].key) Insert(t[u].rs,x); //递归右子树找空叶子，直到插入  else Insert(t[u].ls,x); //递归左子树找空叶子，直到插入  if(t[u].ls!=0 && t[u].pri>t[t[u].ls].pri) rotate(u,0);  if(t[u].rs!=0 && t[u].pri>t[t[u].rs].pri) rotate(u,1);  Update(u);  }  void Del(int &u,int x){  t[u].size--;  if(t[u].key==x){  if(t[u].ls==0&&t[u].rs==0){u=0; return;}  if(t[u].ls==0||t[u].rs==0){u=t[u].ls+t[u].rs; return;}  if(t[t[u].ls].pri < t[t[u].rs].pri)  { rotate(u,0); Del(t[u].rs, x); return;}  else{ rotate(u,1); Del(t[u].ls, x); return;}  }  if(t[u].key>=x) Del(t[u].ls,x);  else Del(t[u].rs,x);  Update(u);  }  int Rank(int u,int x){ //排名，x是第几名  if(u==0) return 0;  if(x>t[u].key) return t[t[u].ls].size+1+Rank(t[u].rs, x);  return Rank(t[u].ls,x);  }  int kth(int u,int k){ //第k大数是几?  if(k==t[t[u].ls].size+1) return t[u].key; //这个数为根  if(k> t[t[u].ls].size+1) return kth(t[u].rs,k-t[t[u].ls].size-1);//右子树  if(k<=t[t[u].ls].size) kth(t[u].ls,k); //左子树  }  int Precursor(int u,int x){  if(u==0) return 0;  if(t[u].key>=x) return Precursor(t[u].ls,x);  int tmp = Precursor(t[u].rs,x);  if(tmp==0) return t[u].key;  return tmp;  }  int Successor(int u,int x){  if(u==0) return 0;  if(t[u].key<=x) return Successor(t[u].rs,x);  int tmp = Successor(t[u].ls,x);  if(tmp==0) return t[u].key;  return tmp;  }  int main(){  srand(time(NULL));  int root = 0; //root是整棵树的树根，0表示初始为空  int n; cin>>n;  while(n--){  int opt,x; cin >> opt >> x;  switch (opt){  case 1: Insert(root,x); break;  case 2: Del(root,x); break;  case 3: printf("%d\n",Rank(root,x)+1); break;  case 4: printf("%d\n",kth(root,x)); break;  case 5: printf("%d\n",Precursor(root,x)); break;  case 6: printf("%d\n",Successor(root,x)); break;  }  }  return 0;  } |

4.14.1 FHQ的基本操作

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98 | //洛谷P3369，FHQ Treap  #include<bits/stdc++.h>  using namespace std;  const int M=1e6+10;  int cnt=0, root=0; //t[cnt]: 最新结点的存储位置；root:整棵树的根，用于访问树  struct Node{  int ls,rs; //左儿子lson，右儿子rson  int key,pri; //key：键值，pri：随机的优先级  int size; //当前结点为根的子树的结点数量，用于求第k大和rank  }t[M]; //tree[],存树  void newNode(int x){ //建立只有一个点的树  cnt++; //从t[1]开始存储结点，t[0]被放弃  t[cnt].size = 1;  t[cnt].ls = t[cnt].rs = 0; //0表示没有子结点  t[cnt].key = x; //key: 键值  t[cnt].pri = rand(); //pri：随机产生，优先级  }  void Update(int u){ //更新以u为根的子树的size  t[u].size = t[t[u].ls].size + t[t[u].rs].size+1;  }  void Split(int u,int x,int &L,int &R) { //权值分裂。返回以L和R为根的2棵树  if(u == 0){L = R = 0; return;} //到达叶子，递归返回  if(t[u].key <= x){ //本结点比x小，那么到右子树上找x  L = u; //左树的根是本结点  Split(t[u].rs, x, t[u].rs, R); //通过rs传回新的子结点  }  else{ //本结点比x大，继续到左子树找x  R = u; //右数的根是本结点  Split(t[u].ls,x,L,t[u].ls);  }  Update(u);//更新当前结点的size  }  int Merge(int L,int R){ //合并以L和R为根的两棵树，返回一棵树的根  if(L==0 || R==0) return L+R; //到达叶子。若L==0，就是返回L+R=R  if(t[L].pri > t[R].pri){ //左树L优先级大于右树R，则L节点是父结点  t[L].rs = Merge(t[L].rs,R); //合并R和L的右儿子,并更新L的右儿子  Update(L);  return L; //合并后的根是L  }  else{ //合并后R是父结点  t[R].ls = Merge(L,t[R].ls); //合并L和R的左儿子,并更新R的左儿子  Update(R);  return R; //合并后的根是R  }  }  int Insert(int x){ //插入数字x  int L,R;  Split(root,x,L,R);  newNode(x); //新建一棵只有一个点的树t[cnt]  int aa = Merge(L,cnt);  root = Merge(aa,R);  }  int Del(int x){ //删除数字x。请对比后面例题洛谷P5055的排名分裂的删除操作  int L,R,p;  Split(root,x,L,R); //<=x的树和>x的树  Split(L,x-1,L,p); //<x的树和==x的树  p = Merge(t[p].ls,t[p].rs); //合并x=p的左右子树，也就是删除了x  root = Merge(Merge(L,p),R);  }  void Rank(int x){ //查询数x的排名  int L,R;  Split(root,x-1,L,R); // <x的树和>=x的树  printf("%d\n",t[L].size+1);  root = Merge(L,R); //恢复  }  int kth(int u,int k){ //求排名第k的数  if(k==t[t[u].ls].size+1) return u; //这个数为根  if(k<=t[t[u].ls].size) return kth(t[u].ls,k); //在左子树  if(k>t[t[u].ls].size) return kth(t[u].rs,k-t[t[u].ls].size-1); //在右子树  }  void Precursor(int x){ //求x的前驱  int L,R;  Split(root,x-1,L,R);  printf("%d\n",t[kth(L,t[L].size)].key);  root = Merge(L,R); //恢复  }  void Successor(int x){ //求x的后继  int L,R;  Split(root,x,L,R); //<=x的树和>x的树  printf("%d\n",t[kth(R,1)].key);  root = Merge(L,R); //恢复  }  int main(){  srand(time(NULL));  int n; cin >> n;  while(n--){  int opt,x; cin >> opt >> x;  switch (opt){  case 1: Insert(x); break;  case 2: Del(x); break;  case 3: Rank(x); break;  case 4: printf("%d\n",t[kth(root,x)].key); break; //排名x的结点  case 5: Precursor(x); break;  case 6: Successor(x); break;  }  }  return 0;  } |

4.14.2 FHQ Treap应用

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60 | //洛谷P4008的FHQ treap代码  #include<bits/stdc++.h>  using namespace std;  const int M = 2e6+10;  int root = 0,cnt = 0;  struct Node{int ls,rs; char val; int pri; int size;}t[M]; //tree[]存树  void Update(int u){t[u].size = t[t[u].ls].size + t[t[u].rs].size+1;} //用于排名分裂  int newNode(char x){ //建立只有一个点的树  cnt++;  t[cnt].size = 1; t[cnt].pri = rand(); t[cnt].ls = t[cnt].rs = 0;  t[cnt].val = x; //一个字符  return cnt;  }  void Split(int u,int x,int &L,int &R){ //排名分裂，不是权值分裂  if(u == 0){L = R = 0; return ;}  if(t[t[u].ls].size+1 <= x){ //第x个数在u的右子树上  L = u; Split(t[u].rs, x-t[t[u].ls].size-1, t[u].rs, R);  }  else{R = u; Split(t[u].ls,x,L,t[u].ls); } //第x个数在左子树上  Update(u);  }  int Merge(int L,int R){ //合并  if(L==0 || R==0) return L+R;  if(t[L].pri > t[R].pri){ t[L].rs = Merge(t[L].rs,R); Update(L); return L; }  else{ t[R].ls = Merge(L,t[R].ls); Update(R); return R;}  }  void inorder(int u){ //中序遍历，打印结果  if(u == 0) return;  inorder(t[u].ls); cout << t[u].val; inorder(t[u].rs);  }  int main(){  srand(time(NULL));  int len, L, p, R, pos = 0; //pos是光标的当前位置  int n; cin>>n;  while(n--){  char opt[10]; cin>>opt;  if(opt[0]=='M') cin>>pos; //移动光标  if(opt[0]=='I'){ //插入len个字符  cin>>len;  Split(root,pos,L,R);  for(int i=1;i<=len;i++) { //逐个读入字符  char ch=getchar(); while(ch<32||ch>126) ch=getchar();  L = Merge(L,newNode(ch)); //把字符加到树中  }  root = Merge(L,R);  }  if(opt[0]=='D'){ //删除光标后len个字符  cin>>len; Split(root,pos+len,L,R); Split(L,pos,L,p);  root = Merge(L,R);  }  if(opt[0]=='G'){ //打印len个字符  cin>>len; Split(root,pos+len,L,R); Split(L,pos,L,p);  inorder(p); cout<<"\n"; //打印  root = Merge(Merge(L,p),R);  }  if(opt[0]=='P') pos--;  if(opt[0]=='N') pos++;  }  return 0;  } |

**2. 区间翻转**

|  |
| --- |
| 例4-42. 文艺平衡树 洛谷P3391 |

|  |  |  |  |
| --- | --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | #include<bits/stdc++.h>  using namespace std;  const int M = 1e5+10;  int cnt = 0, root = 0;  struct Node{int ls,rs,num,pri,size,lazy;}t[M]; //lazy懒标记，参考线段树  void newNode(int x){  cnt++;  t[cnt].size = 1; t[cnt].ls = t[cnt].rs = 0; t[cnt].pri = rand();  t[cnt].num = x; //本题不是按num分裂，是按排名分裂  }  void Update(int u){t[u].size = t[t[u].ls].size + t[t[u].rs].size + 1;}  void pushdown(int u){ //下传lazy标记。如果用splay实现，也是这样做  if(t[u].lazy) {  swap(t[u].ls, t[u].rs); //翻转u的左右部分，翻转不会破坏优先级pri  t[t[u].ls].lazy ^= 1; t[t[u].rs].lazy ^= 1; //向左右子树下传lazy  t[u].lazy = 0; //清除标记  }  }  void Split(int u,int x,int &L,int &R){ //排名分裂。返回：L，前x个数；R，其他数  if(u == 0){L = R = 0; return ;} //到达叶子，递归返回  pushdown(u); //处理lazy标记  if(t[t[u].ls].size+1 <= x){ //第x个数在u的右子树上  L = u; Split(t[u].rs, x-t[t[u].ls].size-1, t[u].rs, R);  }  else{ R = u; Split(t[u].ls,x,L,t[u].ls);} //第x个数在左子树上  Update(u);  }  int Merge(int L,int R){ //合并  if(L==0 || R==0) return L+R;  if(t[L].pri > t[R].pri){ //左树L优先级大于右树R，则L节点是父结点  pushdown(L); //处理lazy标记  t[L].rs = Merge(t[L].rs,R); Update(L); return L;  }  else{ //合并后R是父结点  pushdown(R);  t[R].ls = Merge(L,t[R].ls); Update(R); return R;  }  }  void inorder(int u){ //中序遍历，打印结果  if(u == 0) return;  pushdown(u); //处理lazy  inorder(t[u].ls); cout<<t[u].num<<" "; inorder(t[u].rs); //中序遍历  }  int main(){  srand(time(NULL));  int n,m; cin >> n >>m;  for(int i=1;i<=n;i++){newNode(i); root = Merge(root,cnt);} //建树，本题比较简单  while(m--){  int x,y; cin >> x >> y;  int L,R,p; //三棵树，分别指向三棵树的根结点  Split(root,y,L,R); //分裂成L,p,R三棵树，p是[x,y]区间  Split(L,x-1,L,p);  t[p].lazy ^= 1; //对p用lazy记录翻转  root = Merge(Merge(L,p),R); //合并，还原成一棵树。  }  inorder (root); //输出  return 0;  } | | |
| 例4-43. 可持久化文艺平衡树 洛谷P5055 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | //改写自：https://www.luogu.com.cn/blog/105496/solution-p5055  #include<bits/stdc++.h>  using namespace std;  #define ll long long  const int N=2e5+10;  struct node{int ls,rs,val,pri,size,lazy; ll sum;}t[(N<<7)];  int cnt = 0;  int new\_node(int x){  cnt++;  t[cnt].size = 1; t[cnt].ls = t[cnt].rs = 0;  t[cnt].val = x; t[cnt].sum = x; t[cnt].pri = rand();  t[cnt].lazy = 0;  return cnt;  }  int clone(int u){ //克隆树u。不需要复制整棵树，只复制根就行  int ret = new\_node(0);  t[ret] = t[u];  return ret;  }  void Update(int u){  t[u].size = t[t[u].ls].size + t[t[u].rs].size + 1;  t[u].sum = t[t[u].ls].sum + t[t[u].rs].sum + t[u].val;  }  void push\_down(int u){  if(!t[u].lazy) return;  if(t[u].ls != 0) t[u].ls = clone(t[u].ls);  if(t[u].rs != 0) t[u].rs = clone(t[u].rs);  swap(t[u].ls,t[u].rs);  t[t[u].ls].lazy^=1; t[t[u].rs].lazy^=1;  t[u].lazy = 0;  }  void Split(int u,int x,int &L,int &R){ //排名分裂  if(u == 0){L = R = 0; return ;}  push\_down(u);  if(t[t[u].ls].size+1 <=x){ //第x个数在u的右子树上  L = clone(u); //这个时间点的L，它是这个时间点的u的克隆  Split(t[L].rs, x-t[t[u].ls].size-1, t[L].rs,R);  Update(L);  }  else{  R = clone(u); //这个时间点的R，它是这个时间点的u的克隆  Split(t[R].ls,x,L,t[R].ls);  Update(R);  }  }  int Merge(int L,int R){  if(L==0 || R==0) return L+R;  push\_down(L); push\_down(R);  if(t[L].pri > t[R].pri){ t[L].rs = Merge(t[L].rs,R); Update(L); return L;}  else { t[R].ls = Merge(L,t[R].ls); Update(R); return R;}  }  int root[N];  int main(){  srand(time(NULL));  int version=0; int L,p,R; ll x,y;  for(int i=0;i<N;i++) root[i] = 0;  ll lastans=0;  int n; cin >> n;  while(n--){  int v,op; cin >> v >> op;  if(op==1){ //在第x个数后插入y  cin >>x >>y; x^=lastans;y^=lastans;  Split(root[v],x,L,p);  root[++version]=Merge(Merge(L,new\_node(y)),p); //记录在新的时间点上  }  if(op==2){ //删除第x个数  cin >>x; x^=lastans;  Split(root[v],x,L,R); Split(L,x-1,L,p);  root[++version] = Merge(L,R); //记录在新的时间点上  }  if(op==3){ //翻转区间[x,y]  cin >>x >>y; x^=lastans; y^=lastans;  Split(root[v],y,L,R); Split(L,x-1,L,p);  t[p].lazy ^= 1;  root[++version] = Merge(Merge(L,p),R); //记录在新的时间点上  }  if(op==4){ //查询区间和[x, y]  cin >> x >> y; x^=lastans; y^=lastans;  Split(root[v],y,L,R); Split(L,x-1,L,p); //p树是区间[x,y]  lastans = t[p].sum;  cout << lastans << endl;  root[++version]=Merge(Merge(L,p),R); //记录在新的时间点上  }  }  return 0;  } |

4.15.2 用单调栈建笛卡尔树

|  |
| --- |
| 例4-45. Binary Search Heap Construction Poj 1785 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61 | //改写自：https://blog.csdn.net/qq\_40679299/article/details/80395824  #include<cstdio>  #include<algorithm>  #include<cstring>  #include <stack>  using namespace std;  const int N = 50005;  const int INF = 0x7fffffff;  struct Node{  char s[100]; int ls,rs,fa,pri;  friend bool operator<(const Node& a,const Node& b){  return strcmp(a.s,b.s)<0;}  }t[N];  void buildtree(int n){ //不用栈，直接查最右链  for(int i=1;i<=n;i++){  int pos = i-1; //从前一个结点开始比较，前一个结点在最右链的末端  while(t[pos].pri < t[i].pri)  pos = t[pos].fa; //沿着最右链一直找，直到pos的优先级比i大  t[i].ls = t[pos].rs; //图(4)：i是19，pos是24, 15调整为19的左儿子  t[t[i].ls].fa = i; //图(4)：15的父亲是19  t[pos].rs = i; //图(4)：24的右儿子是19  t[i].fa = pos; //图(4)：19的父亲是24  }  }  void buildtree2(int n){ //用栈来辅助建树  stack <int> ST; ST.push(0); //t[0]进栈，它一直在栈底  for(int i=1;i<=n;i++){  int pos = ST.top();  while(!ST.empty() && t[pos].pri < t[i].pri){  pos = t[ST.top()].fa;  ST.pop(); //把比i优先级小的弹出栈  }  t[i].ls = t[pos].rs;  t[t[i].ls].fa = i;  t[pos].rs = i;  t[i].fa = pos;  ST.push(i); //每个结点都一定要进栈  }  }  void inorder(int x){ //中序遍历打印  if(x==0) return;  printf("(");  inorder(t[x].ls); printf("%s/%d",t[x].s,t[x].pri); inorder(t[x].rs);  printf(")");  }  int main(){  int n;  while(scanf("%d",&n),n){  for(int i=1;i<=n;i++){  t[i].ls = t[i].rs = t[i].fa = 0; //有多组测试，每次要清零  scanf(" %[^/]/%d", t[i].s, &t[i].pri); //注意输入的写法  }  t[0].ls = t[0].rs = t[0].fa = 0; //t[0]不用，从t[1]开始插结点  t[0].pri = INF; //t[0]的优先级无穷大  sort(t+1,t+1+n); //对标签先排序，非常关键。这样建树时就只需要考虑优先级pri  buildtree(n); //buildtree2(n); //两种建树方法  inorder(t[0].rs); //t[0]在树的最左端，第一个点是t[0].rs  printf("\n");  }  return 0;  } |

4.16.3 Splay的常用操作和代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101 | //改写自 https://www.luogu.com.cn/blog/dedicatus545/solution-p4008  #include<bits/stdc++.h>  using namespace std;  const int M = 2e6+10;  int cnt, root;  struct Node{int fa,ls,rs,size; char val;}t[M]; //tree[]存树;  void Update(int u){t[u].size = t[t[u].ls].size + t[t[u].rs].size+1;} //用于排名  char str[M]={0}; //一次输入的字符串  int build(int L,int R,int f){ //把字符串str[]建成平衡树  if(L>R) return 0;  int mid = (L+R)>>1;  int cur = ++cnt;  t[cur].fa = f;  t[cur].val= str[mid];  t[cur].ls = build(L,mid-1,cur);  t[cur].rs = build(mid+1,R,cur);  Update(cur);  return cur; //返回新树的根  }  int get(int x){return t[t[x].fa].rs == x;} //如果u是右儿子，返回1；左儿子返回0  void rotate(int x){ //单旋一次  int f=t[x].fa, g=t[f].fa, son=get(x); //f:父亲；g：祖父  if(son==1) { //x是右儿子，左旋zag  t[f].rs = t[x].ls; //图“单旋”中的结点b  if(t[f].rs) t[t[f].rs].fa = f;  }  else { //x是左儿子，右旋zig  t[f].ls = t[x].rs;  if(t[f].ls) t[t[f].ls].fa = f;  }  t[f].fa=x; //x旋为f的父结点  if(son == 1) t[x].ls=f; //左旋，f变为x的左儿子  else t[x].rs=f; //右旋，f变为x的右儿子  t[x].fa = g; //x现在是祖父的儿子  if(g){ //更新祖父的儿子  if(t[g].rs==f) t[g].rs = x;  else t[g].ls = x;  }  Update(f); Update(x);  }  void splay(int x,int goal){ //goal=0,新的根是x；goal!=0,把x旋为goal的儿子  if(goal == 0) root=x;  while(1){  int f = t[x].fa, g=t[f].fa; //一次处理x,f,g三个点  if(f == goal) break;  if(g != goal) { //有祖父，分一字旋和之字旋两种情况  if(get(x)==get(f)) rotate(f); //一字旋，先旋f、g  else rotate(x);} //之字旋，直接旋x  rotate(x);  }  Update(x);  }  int kth(int k,int u){ //第k大数的位置  if( k == t[t[u].ls].size + 1) return u;  if( k <= t[t[u].ls].size ) return kth(k,t[u].ls);  if( k >= t[t[u].ls].size + 1) return kth(k-t[t[u].ls].size-1,t[u].rs);  }  void Insert(int L,int len){ //插入一段区间  int x = kth(L,root), y = kth(L+1,root); //x:第L个数的位置，y:第L+1个数的位置  splay(x,0); splay(y,x); //分裂  //先把x旋到根，然后把y旋到x的儿子，此时y是x的右儿子，且y的左儿子为空  t[y].ls = build(1,len,y); //合并：建一棵树，挂到y的左儿子上  Update(y); Update(x);  }  void del(int L,int R){ //删除区间[L+1,R]  int x=kth(L,root), y=kth(R+1,root);  splay(x,0); splay(y,x); //y是x的右儿子，y的左儿子是待删除的区间  t[y].ls=0; //剪断左子树，等于直接删除。这里为了简单，没有释放空间  Update(y); Update(x);  }  void inorder(int u){ //中序遍历  if(u==0) return;  inorder(t[u].ls);cout<<t[u].val;inorder(t[u].rs);  }  int main(){  t[1].size=2; t[1].ls=2; //小技巧：虚拟祖父，防止旋转时越界而出错  t[2].size=1; t[2].fa=1; //小技巧：虚拟父亲  root=1; cnt=2; //在操作过程中，root将指向字符串的根  int pos=1; //光标位置  int n; cin>>n;  while(n--){  int len; char opt[10]; cin>>opt;  if(opt[0]=='I'){  cin>>len;  for(int i=1;i<=len;i++){  char ch=getchar(); while(ch<32||ch>126) ch=getchar();  str[i] = ch;  }  Insert(pos,len);  }  if(opt[0]=='D'){ cin>>len; del(pos,pos+len);} //删除区间[pos+1,pos+len]  if(opt[0]=='G'){  cin>>len; int x=kth(pos,root), y=kth(pos+len+1,root);  splay(x,0); splay(y,x);  inorder(t[y].ls); cout<<"\n";  }  if(opt[0]=='M'){ cin>>len; pos=len+1;}  if(opt[0]=='P') pos--;  if(opt[0]=='N') pos++;  }  } |

4.17.3 寻找最近点

|  |
| --- |
| 例4-46. In case of failure hdu2966 |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57 | #include<bits/stdc++.h>  using namespace std;  const int N = 100005;  const int K = 2;  #define ll long long  struct Point{int dim[K];}; //K维数据。本题是二维坐标x=dim[0],y=dim[1]  Point q[N]; //记录输入的n个坐标点  Point t[N]; //存二叉树，用最简单的方法：数组存二叉树  int now; //当前处于第几维，用于cmp()函数的比较  bool cmp(Point a,Point b){return a.dim[now] < b.dim[now];} //第now维数的比较  ll square(int x){return (ll)x\*x;}  ll dis(Point a,Point b){ //点a、b距离的平方  return square(a.dim[0]-b.dim[0])+square(a.dim[1]-b.dim[1]);  }  void build(int L,int R,int dep){ //建树  if(L>=R) return;  int d = dep % K; //轮转法。dep是当前层的深度，d是当前层的维度  int mid = (L+R) >> 1;  now = d;  nth\_element(t+L,t+mid,t+R,cmp); //找中值  build(L,mid,dep+1); //继续建二叉树的下一层  build(mid+1,R,dep+1);  }  ll ans; //答案：到最近点距离的平方值  void query(int L,int R,int dep,Point p){ //查找点p的最近点  if(L >= R)return;  int mid=(L+R)>>1;  int d = dep % K; //轮转法  ll mindis = dis(t[mid],p); //这棵子树的根到p的最小距离  if(ans == 0) ans = mindis; //赋初值  if(mindis!=0 && ans > mindis) ans = mindis; //需要特判t[mid]和p重合的情况  if(p.dim[d] > t[mid].dim[d]) { //在这个维度，p大于子树的根，接下来查右子树  query(mid+1,R,dep+1,p);  if(ans > square(t[mid].dim[d]-p.dim[d])) query(L,mid,dep+1,p);  //如果以ans为半径的圆与左子树相交，那么左子树也要查  }  else{  query(L,mid,dep+1,p); //在这个维度，p小于子树的根，接下来查左子树  if(ans > square(t[mid].dim[d]-p.dim[d])) //右子树也要查  query(mid+1,R,dep+1,p);  }  }  int main(){  int T; scanf("%d",&T);  while(T--){  int n; scanf("%d",&n);  for(int i=0;i<n;i++)  scanf("%d%d",&(q[i].dim[0]),&(q[i].dim[1])), t[i] = q[i];  build(0,n,0); //建树  for(int i=0;i<n;i++){  ans=0;  query(0,n,0,q[i]);  printf("%lld\n",ans);  }  }  return 0;  } |

4.17.4 区间查询

|  |  |  |
| --- | --- | --- |
| 例4-47. 简单题 洛谷P4148 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113 | #include<bits/stdc++.h>  using namespace std;  const int N = 200010;  const double alpha = 0.75; //替罪羊树的不平衡率  #define lc t[u].ls  #define rc t[u].rs  struct Point{  int dim[2],val; //dim[0]即x，dim[1]即y  Point(){};  Point(int x,int y,int vall){dim[0]=x,dim[1]=y,val=vall;}  };  Point order[N]; int cnt; //替罪羊树：用于拍平后存数据  struct kd\_tree{  int ls,rs;  int mi[2],ma[2]; //mi[i]: 第i维上区间的下界; ma[i]:第i维上区间的上界  int sum; //以该点为根的子树权值之和  int size;  Point p;  }t[N];  int tot,root;  int top,tree\_stack[N]; //替罪羊树：回收  int now;  bool cmp(Point a,Point b){return a.dim[now]<b.dim[now];}  void update(int u){  for(int i=0;i<2;i++){  t[u].mi[i] = t[u].ma[i] = t[u].p.dim[i];  if(lc){  t[u].mi[i] = min(t[u].mi[i],t[lc].mi[i]);  t[u].ma[i] = max(t[u].ma[i],t[lc].ma[i]);  }  if(rc){  t[u].mi[i] = min(t[u].mi[i],t[rc].mi[i]);  t[u].ma[i] = max(t[u].ma[i],t[rc].ma[i]);  }  }  t[u].sum = t[lc].sum + t[u].p.val+t[rc].sum;  t[u].size = t[lc].size + t[rc].size+1;  }  void slap(int u) { //替罪羊树：拍平  if(!u) return;  slap(lc); //这里用中序遍历。其实先序、后序也行  order[++cnt] = t[u].p;  tree\_stack[++top] = u; //回收结点  slap(rc);  }  int build(int l,int r,int d) { //替罪羊树：建树  if(l>r) return 0;  int u;  if(top) u = tree\_stack[top--];  else u = ++tot;  int mid=(l+r)>>1;  now = d;  nth\_element(order+l, order+mid, order+r+1, cmp);  t[u].p = order[mid];  lc = build(l,mid-1,d^1); //奇偶轮转法。没有用例题hdu2966的一般轮转法  rc = build(mid+1,r,d^1);  update(u);  return u;  }  bool notbalance(int u){ //替罪羊树：判断子树u是否平衡  if(t[lc].size>alpha\*t[u].size || t[rc].size>alpha\*t[u].size)  return true; //不平衡了  return false; //还是平衡的  }  void Insert(int &u,Point now,int d){  if(!u) {  if(top) u=tree\_stack[top--];  else u = ++tot;  lc = rc = 0,t[u].p = now;  update(u);  return;  }  if(now.dim[d] <= t[u].p.dim[d]) Insert(lc,now,d^1);//按第d维的坐标比较  else Insert(rc,now,d^1);  update(u);  if(notbalance(u)){ //不平衡  cnt = 0;  slap(u); //拍平  u = build(1,t[u].size,d); //重建  }  }  int query(int u,int x1,int y1,int x2,int y2){  if(!u) return 0;  int X1=t[u].mi[0], Y1=t[u].mi[1], X2=t[u].ma[0], Y2=t[u].ma[1];  if(x1<=X1 && x2>=X2 && y1<=Y1 && y2>=Y2) return t[u].sum;  //子树表示的矩形完全在询问矩形范围内  if(x1>X2 || x2<X1 || y1>Y2 || y2<Y1) return 0;  //子树表示的矩形完全在询问矩形范围外  int ans=0;  X1=t[u].p.dim[0], Y1=t[u].p.dim[1], X2=t[u].p.dim[0], Y2=t[u].p.dim[1];  if(x1<=X1 && x2>=X2 && y1<=Y1 && y2>=Y2) ans+=t[u].p.val;//根在询问矩形内  ans += query(lc,x1,y1,x2,y2) + query(rc,x1,y1,x2,y2); //递归左右子树  return ans;  }  int main(){  int n; cin >> n;  int ans=0;  while(1){  int opt;scanf("%d",&opt);  if(opt==1){  int x,y,val; scanf("%d%d%d",&x,&y,&val);  x^=ans,y^=ans,val^=ans;  Insert(root,Point(x,y,val),0);  }  if(opt==2){  int x1,y1,x2,y2; scanf("%d%d%d%d",&x1,&y1,&x2,&y2);  x1^=ans,y1^=ans,x2^=ans,y2^=ans;  ans = query(root,x1,y1,x2,y2);  printf("%d\n",ans);  }  if(opt==3) break;  }  } |

4.18.5 模板题

|  |  |  |
| --- | --- | --- |
| 例4-48. 动态树 洛谷 P3690 | | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96 | //代码改写自：https://www.luogu.com.cn/blog/ecnerwaIa/dai-ma-jiang-xie  #include<bits/stdc++.h>  using namespace std;  const int N=3e5+5;  struct node{ int fa,ch[2],sum,val,lazy; }t[N]; //lazy用来标记reverse()的左右翻转  #define lc t[x].ch[0] //左儿子  #define rc t[x].ch[1] //右儿子  bool isRoot(int x){ //判断是否是splay根节点  int g=t[x].fa;  return t[g].ch[0]!=x && t[g].ch[1]!=x;//若为根,则父结点不应该有这个儿子  }  void pushup(int x){ //本题的求路径异或和。上传信息  t[x].sum=t[x].val^t[lc].sum^t[rc].sum;  }  void reverse(int x){  if(!x)return;  swap(lc,rc); //翻转x的左右儿子  t[x].lazy^=1; //懒惰标记，先不翻转儿子的后代，后面再翻转  }  void pushdown(int x){ //递归翻转x的儿子的后代，并释放懒标记。  if(t[x].lazy){  reverse(lc);  reverse(rc);  t[x].lazy=0;  }  }  void push(int x){  if(!isRoot(x)) push(t[x].fa); //从根到x全部pushdown  pushdown(x);  }  void rotate(int x){  int y=t[x].fa;  int z=t[y].fa;  int k=t[y].ch[1]==x;  if(!isRoot(y)) t[z].ch[t[z].ch[1]==y]=x;  t[x].fa=z;  t[y].ch[k]=t[x].ch[k^1];  if(t[x].ch[k^1])t[t[x].ch[k^1]].fa=y;  t[y].fa=x;  t[x].ch[k^1]=y;  pushup(y);  }  void splay(int x){ //提根：把x旋转为它所在的Splay树的根  int y,z;  push(x); //先pushdown处理x的所有子孙的lazy标记  while(!isRoot(x)){  y=t[x].fa,z=t[y].fa;  if(!isRoot(y))  (t[z].ch[0]==y)^(t[y].ch[0]==x)?rotate(x):rotate(y);  rotate(x);  }  pushup(x);  }  void access(int x){ //在原树上建一条实链，起点是根，终点是x  for(int child=0; x; child=x, x=t[x].fa){ //从x往上走，沿着虚边走到根  splay(x);  rc = child; //右孩子是child，建立了一条实边  pushup(x);  }  }  void makeroot(int x){ //把x在原树上旋转到根的位置  access(x); splay(x); reverse(x);  }  void split(int x,int y){ //把原树上以x为起点、y为终点的路径，生成一条实链  makeroot(x);  access(y);  splay(y);  }  void link(int x,int y){ //在结点x和y之间连接一条边  makeroot(x); t[x].fa=y;  }  void cut(int x,int y){ //将x,y的边切断  split(x,y);  if(t[y].ch[0]!=x||rc) return;  t[x].fa=t[y].ch[0]=0;  pushup(x);  }  int findroot(int x){ //查找x在原树上的根  access(x); splay(x);  while(lc) pushdown(x),x=lc; //找Splay树最左端的结点  return x;  }  int main(){  int n,m; scanf("%d%d",&n,&m);  for(int i=1;i<=n;++i){ scanf("%d",&t[i].val); t[i].sum = t[i].val; }  while(m--){  int opt,a,b; scanf("%d%d%d",&opt,&a,&b);  switch(opt){  case 0: split(a,b); printf("%d\n",t[b].sum); break;  case 1: if(findroot(a) != findroot(b))link(a,b); break;  case 2: cut(a,b); break;  case 3: splay(a); t[a].val=b; break;  }  }  return 0;  } |

4.18.6 LCT的基本应用

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | //洛谷P3379的部分代码  const int N=500000+5;  int query\_lca(int x, int y) { //求LCA(x, y)  access(x);  int ans;  for (int child = 0; y; child = y, y = t[y].fa) { //模拟access(), y==0时退出  splay(y); //若y在从根出发的路径p上，splay(y)后y是Splay的根，y没有父结点，t[y].fa=0  t[y].ch[1] = child;  ans = y;  }  return ans;  }  int main() {  int n,m,rt; scanf("%d%d%d",&n,&m,&rt);  for (int i=1; i<n; ++i) { int x,y; scanf("%d%d",&x,&y); link(x, y);}  makeroot(rt); //定义根结点是rt  for (int i=1; i<=m; ++i){ int x,y; scanf("%d%d",&x,&y); printf("%d\n", query\_lca(x, y));}  return 0;  } |