

可持久化线段树

主席树

主席树全称是可持久化权值线段树，参见 [知乎讨论](#)。

⚠ 关于函数式线段树



函数式线段树 是指使用函数式编程思想的线段树。在函数式编程思想中，将计算机运算视为数学函数，并避免可改变的状态或变量。不难发现，函数式线段树是 [完全可持久化](#) 的。

引入

先引入一道题目：给定 n 个整数构成的序列 a ，将对于指定的闭区间 $[l, r]$ 查询其区间内的第 k 小值。

你该如何解决？

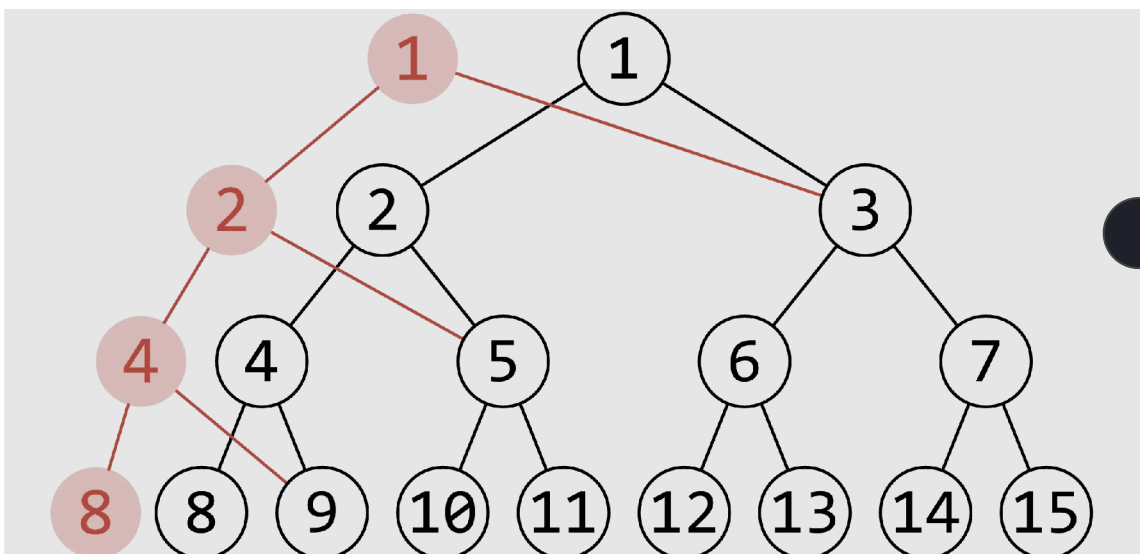
一种可行的方案是：使用主席树。主席树的主要思想就是：保存每次插入操作时的历史版本，以便查询区间第 k 小。

怎么保存呢？简单暴力一点，每次开一棵线段树呗。
那空间还不爆掉？

解释

我们分析一下，发现每次修改操作修改的点的个数是一样的。

（例如下图，修改了 $[1, 8]$ 中对应权值为 1 的结点，红色的点即为更改的点）



只更改了 $O(\log n)$ 个结点，形成一条链，也就是说每次更改的结点数 = 树的高度。

注意主席树不能使用堆式存储法，就是说不能用 $x \times 2$, $x \times 2 + 1$ 来表示左右儿子，而是应该动态开点，并保存每个节点的左右儿子编号。

所以我们只要在记录左右儿子的基础上，保存插入每个数的时候的根节点就可以实现持久化了。

我们把问题简化一下：每次求 $[1, r]$ 区间内的 k 小值。

怎么做呢？只需要找到插入 r 时的根节点版本，然后用普通权值线段树（有的叫键值线段树/值域线段树）做就行了。

这个相信大家都能理解，回到原问题——求 $[l, r]$ 区间 k 小值。

这里我们再联系另外一个知识：**前缀和**。

这个小东西巧妙运用了区间减法的性质，通过预处理从而达到 $O(1)$ 回答每个询问。

我们可以发现，主席树统计的信息也满足这个性质。

所以.....如果需要得到 $[l, r]$ 的统计信息，只需要用 $[1, r]$ 的信息减去 $[1, l - 1]$ 的信息就行了。

至此，该问题解决！

关于空间问题，我们分析一下：由于我们是动态开点的，所以一棵线段树只会出现 $2n - 1$ 个结点。

然后，有 n 次修改，每次至多增加 $\lceil \log_2 n \rceil + 1$ 个结点。因此，最坏情况下 n 次修改后的结点总数会达到 $2n - 1 + n(\lceil \log_2 n \rceil + 1)$ 。此题的 $n \leq 10^5$ ，单次修改至多增加 $\lceil \log_2 10^5 \rceil + 1 = 18$ 个结点，故 n 次修改后的结点总数为 $2 \times 10^5 - 1 + 18 \times 10^5$ ，忽略掉 -1 ，大概就是 20×10^5 。

最后给一个忠告：千万不要吝啬空间（大多数题目中空间限制都较为宽松，因此一般不用担心空间超限的问题）！大胆一点，直接上个 $2^5 \times 10^5$ ，接近原空间的两倍（即 $n \ll 5$ ）。

实现

```
1 #include <algorithm>
2 #include <cstdio>
```

```

3  #include <cstring>
4  using namespace std;
5  constexpr int MAXN = 1e5; // 数据范围
6  int tot, n, m;
7  int sum[(MAXN << 5) + 10], rt[MAXN + 10], ls[(MAXN << 5) + 10],
8      rs[(MAXN << 5) + 10];
9  int a[MAXN + 10], ind[MAXN + 10], len;
10
11 int getid(const int &val) { // 离散化
12     return lower_bound(ind + 1, ind + len + 1, val) - ind;
13 }
14
15 int build(int l, int r) { // 建树
16     int root = ++tot;
17     if (l == r) return root;
18     int mid = l + r >> 1;
19     ls[root] = build(l, mid);
20     rs[root] = build(mid + 1, r);
21     return root; // 返回该子树的根节点
22 }
23
24 int update(int k, int l, int r, int root) { // 插入操作
25     int dir = ++tot;
26     ls[dir] = ls[root], rs[dir] = rs[root], sum[dir] = sum[root] + 1;
27     if (l == r) return dir;
28     int mid = l + r >> 1;
29     if (k <= mid)
30         ls[dir] = update(k, l, mid, ls[dir]);
31     else
32         rs[dir] = update(k, mid + 1, r, rs[dir]);
33     return dir;
34 }
35
36 int query(int u, int v, int l, int r, int k) { // 查询操作
37     int mid = l + r >> 1,
38         x = sum[ls[v]] - sum[ls[u]]; // 通过区间减法得到左儿子中所存储的数值个数
39     if (l == r) return l;
40     if (k <= x) // 若 k 小于等于 x，则说明第 k 小的数字存储在左儿子中
41         return query(ls[u], ls[v], l, mid, k);
42     else // 否则说明在右儿子中
43         return query(rs[u], rs[v], mid + 1, r, k - x);
44 }
45
46 void init() {
47     scanf("%d%d", &n, &m);
48     for (int i = 1; i <= n; ++i) scanf("%d", a + i);
49     memcpy(ind, a, sizeof ind);
50     sort(ind + 1, ind + n + 1);
51     len = unique(ind + 1, ind + n + 1) - ind - 1;
52     rt[0] = build(1, len);
53     for (int i = 1; i <= m; ++i) rt[i] = update(getid(a[i]), 1, len, rt[i -
54 1]);
55 }
56
57 int l, r, k;
58
59 void work() {

```

```

60     while (m--) {
61         scanf("%d%d%d", &l, &r, &k);
62         printf("%d\n", ind[query(rt[l - 1], rt[r], 1, len, k)]); // 回答问题
63     }
64 }
65
66 int main() {
67     init();
68     work();
69     return 0;
70 }

```

拓展：基于主席树的可持久化并查集

主席树是实现可持久化并查集的便捷方式，在此也提供一个基于主席树的可持久化并查集实现示例。

```

1  #include <algorithm>
2  #include <iostream>
3  using namespace std;
4
5  struct SegmentTree {
6      int lc, rc, val, rnk;
7  };
8
9  constexpr int MAXN = 100000 + 5;
10 constexpr int MAXM = 200000 + 5;
11
12 SegmentTree
13     t[MAXN * 2 +
14         MAXM * 40]; // 每次操作1会修改两次，一次修改父节点，一次修改父节点的秩
15 int rt[MAXM];
16 int n, m, tot;
17
18 int build(int l, int r) {
19     int p = ++tot;
20     if (l == r) {
21         t[p].val = l;
22         t[p].rnk = 1;
23         return p;
24     }
25     int mid = (l + r) / 2;
26     t[p].lc = build(l, mid);
27     t[p].rc = build(mid + 1, r);
28     return p;
29 }
30
31 int getRnk(int p, int l, int r, int pos) { // 查询秩
32     if (l == r) {
33         return t[p].rnk;
34     }
35     int mid = (l + r) / 2;
36     if (pos <= mid) {
37         return getRnk(t[p].lc, l, mid, pos);
38     }
39     return getRnk(t[p].rc, mid + 1, r, pos);
40 }

```

```

38     } else {
39         return getRnk(t[p].rc, mid + 1, r, pos);
40     }
41 }
42
43 int modifyRnk(int now, int l, int r, int pos, int val) { // 修改秩 (高度)
44     int p = ++tot;
45     t[p] = t[now];
46     if (l == r) {
47         t[p].rnk = max(t[p].rnk, val);
48         return p;
49     }
50     int mid = (l + r) / 2;
51     if (pos <= mid) {
52         t[p].lc = modifyRnk(t[now].lc, l, mid, pos, val);
53     } else {
54         t[p].rc = modifyRnk(t[now].rc, mid + 1, r, pos, val);
55     }
56     return p;
57 }
58
59 int query(int p, int l, int r, int pos) { // 查询父节点 (序列中的值)
60     if (l == r) {
61         return t[p].val;
62     }
63     int mid = (l + r) / 2;
64     if (pos <= mid) {
65         return query(t[p].lc, l, mid, pos);
66     } else {
67         return query(t[p].rc, mid + 1, r, pos);
68     }
69 }
70
71 int findRoot(int p, int pos) { // 查询根节点
72     int f = query(p, 1, n, pos);
73     if (pos == f) {
74         return pos;
75     }
76     return findRoot(p, f);
77 }
78
79 int modify(int now, int l, int r, int pos, int fa) { // 修改父节点 (合并)
80     int p = ++tot;
81     t[p] = t[now];
82     if (l == r) {
83         t[p].val = fa;
84         return p;
85     }
86     int mid = (l + r) / 2;
87     if (pos <= mid) {
88         t[p].lc = modify(t[now].lc, l, mid, pos, fa);
89     } else {
90         t[p].rc = modify(t[now].rc, mid + 1, r, pos, fa);
91     }
92     return p;
93 }
94

```

```

95 int main() {
96     cin.tie(nullptr)->sync_with_stdio(false);
97     cin >> n >> m;
98     rt[0] = build(1, n);
99     for (int i = 1; i <= m; i++) {
100         int op, a, b;
101
102         cin >> op;
103         if (op == 1) {
104             cin >> a >> b;
105             int fa = findRoot(rt[i - 1], a), fb = findRoot(rt[i - 1], b);
106             if (fa != fb) {
107                 if (getRnk(rt[i - 1], 1, n, fa) >
108                     getRnk(rt[i - 1], 1, n, fb)) { // 按秩合并
109                     swap(fa, fb);
110                 }
111                 int tmp = modify(rt[i - 1], 1, n, fa, fb);
112                 rt[i] = modifyRnk(tmp, 1, n, fb, getRnk(rt[i - 1], 1, n, fa) +
113 1);
114             } else {
115                 rt[i] = rt[i - 1];
116             }
117         } else if (op == 2) {
118             cin >> a;
119             rt[i] = rt[a];
120         } else {
121             cin >> a >> b;
122             rt[i] = rt[i - 1];
123             cout << (findRoot(rt[i], a) == findRoot(rt[i], b)) << '\n';
124         }
125     }
126
127     return 0;
}

```

参考

https://en.wikipedia.org/wiki/Persistent_data_structure

<https://www.cnblogs.com/zinthos/p/3899565.html>

🔧 本页面最近更新：2024/12/11 21:43:19，[更新历史](#)

🔧 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[Ir1d](#), [H-J-Granger](#), [StudyingFather](#), [EndlessCheng](#), [Enter-tainer](#), [countercurrent-time](#), [NachtgeistW](#), [cjsoft](#), [abc1763613206](#), [Alpha1022](#), [AngelKitty](#), [CCXXI](#), [diauweb](#), [Early0v0](#), [ezoixx130](#), [GekkaSaori](#), [Konano](#), [LovelyBuggies](#), [Makkiy](#), [mgt](#), [minghu6](#), [ouuan](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [sshwy](#), [Suyun514](#), [Tiphereth-A](#), [weiyong1024](#), [billchenchina](#), [ChungZH](#), [FinParker](#), [GavinZhengOI](#), [Gesrua](#), [Honeta](#), [hsfzLZH1](#), [iamtwz](#), [ksyx](#), [kxccc](#), [lychees](#), [Marcythm](#), [Peanut-Tang](#), [renbaoshuo](#), [SukkaW](#), [william-song-](#)



shy

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用

