可持久化线段树

河南省实验中学信息技术组

2026 年 04 月 16 日

可持久化

- 数据结构一般维护的都是数据集的最新状态，但是如何获取数据集在任意时间的历史状态 (即如果有 m 个操作，执行完第 i 项操作后数据集的状态)。
- 一种朴素做法是在第 i 项操作后，把整个数据结构复制一边存储在 $history[i]$ 中，但是这样会使用大量空间。
- 可持久化提供了一种思想，在每项操作结束后，仅创建数据结构中发生改变的部分的副本。这样，维护数据结构的时间复杂度没有增加，空间复杂度仅增长为与操作同级的规模。
- 基于以上思想，可以很容易设计出多种数据结构的可持久化版本——前提是数据结构的内部结构在操作过程中不发生变化。

【例】可持久化线段树

给定一个长度为 n 的整数序列 a，为第 0 版，然后执行 m 次操作：

- `0 x y`，表示产生一个新版本，然后将 $a[x]$ 增加 y。
- `1 k l r`，表示查询第 k 个版本中 $a[l,r]$ 的和。

数据范围：$1 \le n, m \le 10^5, |a[i]|, |y| \le 10^9$，保证 $1 \le k \le$ 已经存在的版本。

【例】可持久化线段树

【输入格式】
第一行一个正整数 n，表示序列 a 的大小，一个正整数 m，表示 m 次操作。
第二行包含 n 个整数，表示序列 a。
接下来 m 行，每行一个操作，具体格式参考【题目描述】。
【输出格式】
对于每个查询操作，输出一行一个整数表示答案。

【样例输入】

```
5 8
4 3 6 2 4
0 2 1
0 5 7
1 2 1 4
0 3 4
1 1 2 4
1 3 4 4
0 4 6
1 4 2 4
```

【样例输出】

```
16
12
2
22
```

可持久化线段树

- 对于可持久化线段树，在进行单点修改的过程中，只会使 $\log N$ 个区间对应的结点发生更新，所以可以新建 $\log N$ 个新结点用于维护变化的信息即可。

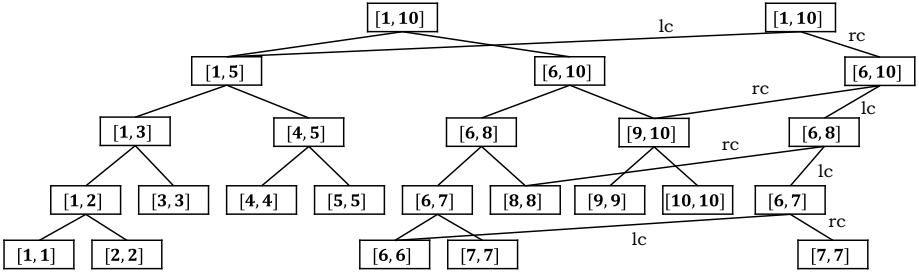


图: 单点修改 7

可持久化线段树

- 单点更新算法
- 在线段树中进行自上而下的更新时，设当前被更新的结点为 p，则我们创建该结点的副本 q。
- 只要 p 不是叶子结点，那么 p 的左右子树之一必然发生了更新。
- 不妨设被更新的是 p 的左子树 $lc[p]$，那么需要新建 q 的左子树 $lc[q]$，那么 p 递归到 $lc[p]$、q 递归到 $lc[q]$。
- 递归返回时在 q 上更新区间的和即可。

实现

- 可持久化线段树不再具有完全二叉树的性质，所以不能用堆的方式来存储，只能使用普通二叉树来实现。

```
// 初始时需要 2 * n 的空间，对于每个操作需要建立 log n 个结点
const int N = 2e6 + 10;
int a[N];
int sum[N], lc[N], rc[N], rt[N], tot = 0;

void build(int &p, int L, int R)
{
    p = ++tot;
    if(L == R) { sum[p] = a[L]; return; }
    build(lc[p], L, M), build(rc[p], M + 1, R);
    sum[p] = sum[lc[p]] + sum[rc[p]]
}
// 主函数中
build(rt[0], 1, n);
```

实现

- 对于第 s 次修改，以可持久化线段树的第 $s-1$ 个版本为基础。

```
// 单点修改 注意 q 传入的是引用
void update(int p, int &q, int L, int R, int x, int y)
{
    q = ++tot;
    lc[q] = lc[p], rc[q] = rc[p], sum[q] = sum[p];
    if(L == R) { sum[q] += y; return; }
    if(x <= M) update(lc[p], lc[q], L, M, x, y);
    else update(rc[p], rc[q], M + 1, R, x, y);
    sum[q] = sum[lc[q]] + sum[rc[q]];
}

// 主函数中
update(rt[s - 1], rt[s], x, val);
```

可持久化线段树

- 可持久化线段树维护了每次操作之后的线段树的历史形态。
- 从 $tt[i]$ 出发向下能访问到的结点构成了执行前 i 次单点修改后的线段树。
- 因为每次修改都会创建 $\log N$ 个结点，所以可持久化线段树的空间复杂度为 $\mathrm{O}(N + M \log N)$。
- 和普通线段树类似，因为线段树结点代表的区间一般是不变的，所以可以不在结点中存储区间边界，只需要在处理时将区间边界作为函数参数即可。

【例】可持久化线段树

给定一个长度为 n 的整数序列 a，为第 0 版，然后执行 m 次操作：

- `0 l r x`，表示产生一个新版本，然后将 $a[l,r]$ 增加 x。
- `1 k l r`，表示查询第 k 个版本中 $a[l,r]$ 的和。

数据范围：$1 \le n, m \le 10^5, |a[i], |x| \le 10^6$，保证 $1 \le k \le$ 已经存在的版本。

【例】可持久化线段树

【输入格式】
第一行一个正整数 n，表示序列 a 的大小，一个正整数 m，表示 m 次操作。
第二行包含 n 个整数，表示序列 a。
接下来 m 行，每行一个操作，具体格式参考【题目描述】。
【输出格式】
对于每个查询操作，输出一行一个整数表示答案。

【样例输入】

```
5 8
1 9 1 10 3
0 2 5 4
0 3 4 1
1 2 1 2
1 2 3 4
1 1 3 4
0 2 4 5
0 4 5 4
1 4 1 5
```

【样例输出】

```
14
21
19
65
```

标记永久化

- 可持久化线段树难以支持大部分的区间修改操作。
- 可持久化线段树建立的结点的孩子结点可能会指向历史上的结点，而当结点下传延迟标记时，它的左右子结点 lc, rc 的将会发生变化，那么历史上结点信息就有可能发生变化，这与事实不符。
- 那么我们也可以在下传延迟标记时，新建左右子节点 lc', rc'，此时可能需要重新计算某些信息，这是不可接受的。
- 故而标记是不可以下传的，那么标记只能永久保存在结点中，因此也称为标记永久化。
- 那么查询时，除了计算结点本身的信息值，还需要计算标记产生的影响，故而标记永久化解决区间修改问题的应用是有很大局限性的。

标记永久化

- 执行操作：区间 $[3,6]$ 上每个数字加 2。
- 在 rt_1 上查询区间 $[4,5]$ 的和，结果为区间 $[3,4]$ 的标记 $2\times$ 区间 $[4,5]$ 与区间 $[3,4]$ 重合长度 $1+$ 区间 $[5,6]$ 的标记 $2\times$ 区间 $[4,5]$ 与区间 $[5,6]$ 重合长度 $1+$ 区间 $[4,4]$ 的值 $+$ 区间 $[5,5]$ 的值 $=2\times1+2\times1+0+0=4$。

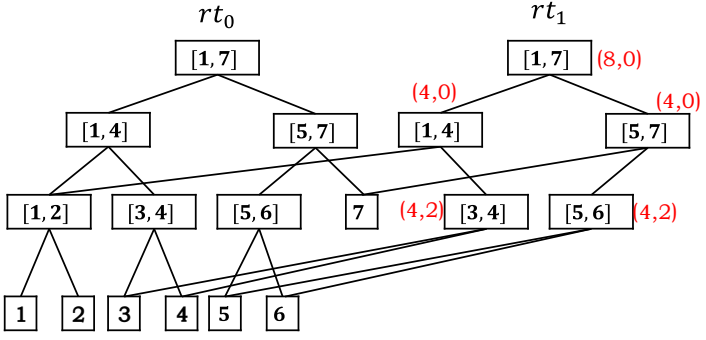


图: 括号内第一个数为区间和，第二个数为延迟标记

标记永久化

- 执行操作：区间 $[5,7]$ 上每个数字加 3。
- 在 rt_2 上查询区间 $[3,6]$ 的和，结果为区间 $[3,4]$ 的值 + 区间 $[5,6]$ 的值 + 区间 $[5,7]$ 的标记 $3\times$ 区间 $[3,6]$ 与区间 $[5,7]$ 的重合长度 $2 = 4+4+3\times 2 = 14$。

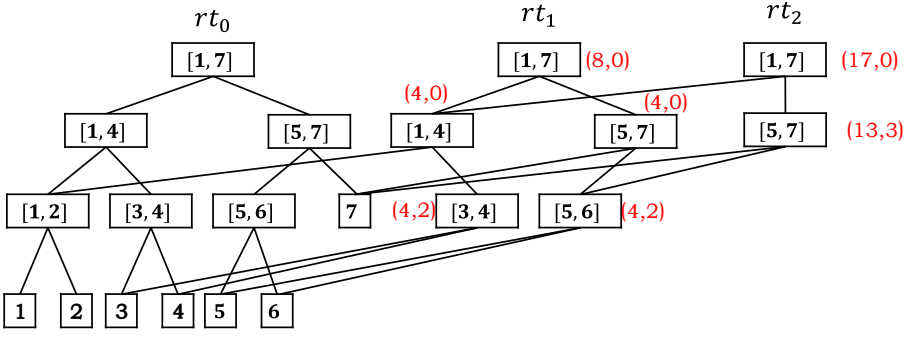


图: 括号内第一个数为区间和，第二个数为延迟标记

标记永久化

```
// 区间查询
// 因为延迟标记产生的影响与查询的区间有关 所以传入的区间端点一定要准确
long long query(int p, int L, int R, int l, int r)
{
    if(l == L && R == r) return sum[p];
    long long ans = 0;
    if(r <= M) ans = query(lc[p], L, M, l, r);
    else if(M < l) ans = query(rc[p], M + 1, R, l, r);
    else ans = query(lc[p], L, M, l, M) + query(rc[p], M + 1, R, M + 1, r);
    ans += add[p] * (r - l + 1); // 计算延迟标记产生的影响
    return ans;
}
// 主函数
long long ans = query(rt[k], 1, n, l, r);
```

标记永久化

```
// 区间修改
// 注意 q 传入的是引用
void update(int p, int &q, int L, int R, int l, int r, int x)
{
    q = ++tot;
    lc[q] = lc[p], rc[q] = rc[p], sum[q] = sum[p], add[q] = add[p];
    if(l <= L && R <= r)
    {
        sum[q] += (long long)(R - L + 1) * x;
        add[q] += x;
        return;
    }
    if(l <= M) update(lc[p], lc[q], L, M, l, r, x);
    if(M < r) update(rc[p], rc[q], M + 1, R, l, r, x);
    sum[q] = sum[lc[q]] + sum[rc[q]];
    // 重新计算延迟标记的影响 因为延迟标记没有下传 所以从子结点计算出的结果不准确
    sum[q] += (long long)(R - L + 1) * add[q];
}
//主函数
update(rt[s - 1], rt[s], 1, r, x);
```

【例】第 k 小的数

【题目描述】
给定一个长度为 n 的整数序列 a，执行 m 次操作，每次操作给出三个整数 l, r, k，求 $a_l, a_{l+1}, \cdots, a_r$ 中第 k 小的数是多少。
【输入格式】
第一行一个正整数 n，表示序列 a 的大小，一个正整数 m，表示 m 次询问。
第二行包含 n 个不同的整数，表示序列 a。
接下来 m 行，每行包含三个整数 l, r, k，表示一次操作。
【输出格式】
共有 m 行，每行一个整数，表示查询的第 k 小的数。
【样例输入】

```
7 3
1 5 2 6 3 7 4
2 5 3
4 4 1
1 7 3
```

【样例输出】

```
5
6
3
```

【数据范围】
$1 \leq n, m \leq 10^5, 1 \leq l, r \leq n, 1 \leq k \leq r - l + 1, |a_i| \leq 10^9$。

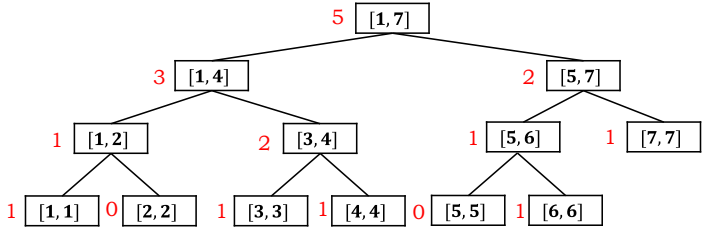
【例】第 k 小的数

- 如果不考虑查询区间 $[l,r]$，只求序列 a 的第 k 小的数，有很多方法解决[1]。
- 其中第 2 种方法在分治的过程中，记录分治的左侧区间有多少个数，然后与 k 比较从而确定结果在左侧区间还是右侧区间。
- 定义权值线段树维护序列 a 有多少个数落在 $[L,R]$ 之间 (记为 $cnt_{L,R}$)，那么只需要比较 $cnt_{L,M}$ 与 k 的大小关系，即可以确定 a 的第 k 小的数在 $[L,M]$ 还是在 $[M+1,R]$ 之间，从而在线段树中继续向下寻找，最终找到答案。

[1] 1. 排序 2. 快速排序 3. 线性时间选择

【例】第 k 小的数

- 例如，有 5 个数 4,1,3,6,7，在其中找第 3 大的数。
- 首先，在根结点 $[1,7]$，查询左孩子的 $cnt_{1,4}$ 结果为 3，说明第 3 大的数应该是左孩子 $[1,4]$ 中的第 3 大的数。
- 进入结点 $[1,4]$，查询左孩子的 $cnt_{1,2}$ 结果为 1，说明第 3 大的数应该在右孩子 $[3,4]$ 中，且为右孩子 $[3,4]$ 的第 2 大的数。
- 进入结点 $[3,4]$，查询左孩子的 $cnt_{3,3}$ 结果为 1，说明第 2 大的数应该在右孩子 $[4,4]$ 中，且为右孩子 $[4,4]$ 的第 1 大的数。
- 进入结点 $[4,4]$，因为区间长度为 1，所以答案就是 4。

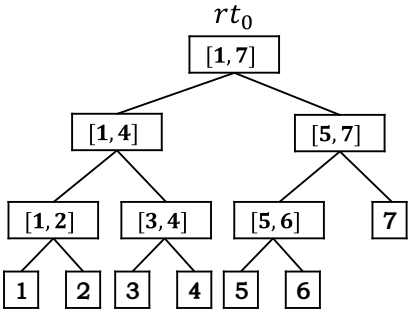


【例】第 k 小的数

- 当然序列 a 数值范围较大，可以先离散化，使得 a 的数值范围落在 $[1,s]$。
- 有查询区间 $[l,r]$ 的限制情况下，因为线段树中存储的是每个数出现的次数，已经丢失了数据原来的位置信息，此时已经不知道 $a[l:r]$ 中的数在哪里。
- 可以考虑使用可持久化线段树来维护，对于每个数都定义一棵范围在 $[1,s]$ 的权值线段树，其中第 i 棵树维护 $a[1:i]$ 各个数值出现的次数 cnt。
- 那么以 $rt[i]$ 为根的线段树的信息减去以 $rt[i-1]$ 为根的线段树的信息就是 a_i 的值。
- 对于以 $rt[i]$ 为根的线段树，查询区间 $[L,R]$ 的区间和，就是 $a[1:i]$ 在 $[L,R]$ 中出现的数的个数。

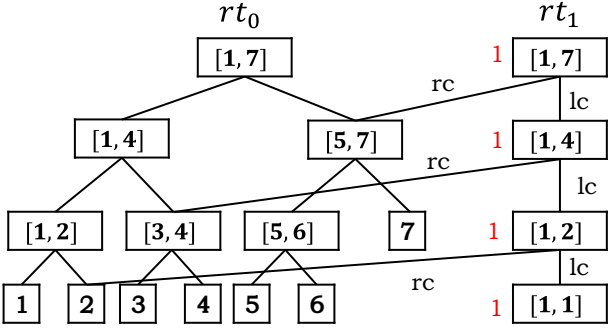
【例】第 k 小的数

- 插入的数据为 1 5 2 6 3 7 4，离散化后的数据和原来一样。
- 初始情况下，先建立一棵完整的线段树，线段树结点中保存的 cnt 值为 0。



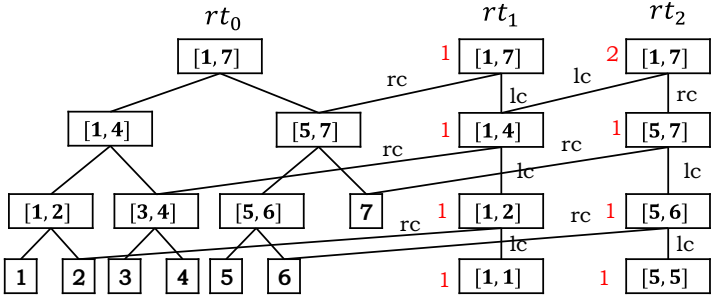
【例】第 k 小的数

- 插入第 1 个数 1：



【例】第 k 小的数

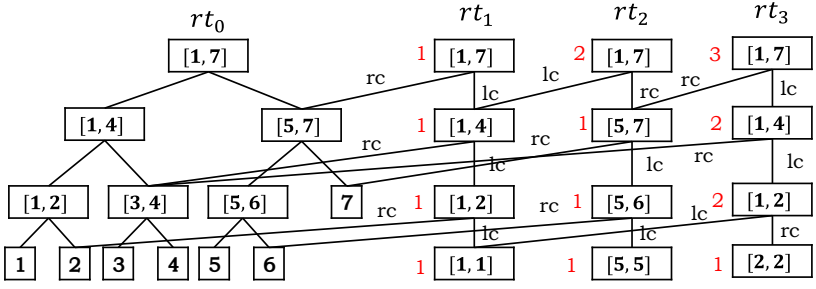
- 插入第 2 个数 5：



- 此时在以 $root_2$ 为根的线段树上进行区间 $[2,5]$ 的查询结果为 1，表示前 2 个数落在区间 $[2,5]$ 的数共有 1 个 (数字 5)。

【例】第 k 小的数

- 插入第 3 个数 2：



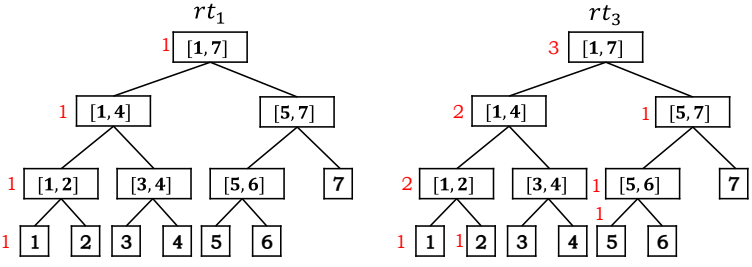
- 此时在以 $root_3$ 为根的线段树上进行区间 $[2, 5]$ 的查询结果为 2，表示前 2 个数落在区间 $[2, 5]$ 的数共有 2 个 (数字 2、5)。

【例】第 k 小的数


```
1  void update(int p, int &q, int L, int R, int x)
2  {
3      q = ++tot;
4      lc[q] = lc[p], rc[q] = rc[p], cnt[q] = cnt[p];
5      if(L == R) { ++cnt[q]; return; }
6      if(x <= M) update(lc[p], lc[q], L, M, x);
7      else update(rc[p], rc[q], M + 1, R, x);
8      cnt[q] = cnt[lc[q]] + cnt[rc[q]];
9  }
10 // 主函数
11 for(int i = 1; i <= n; ++i) update(rt[i - 1], rt[i], 1, s, a[i]);
```

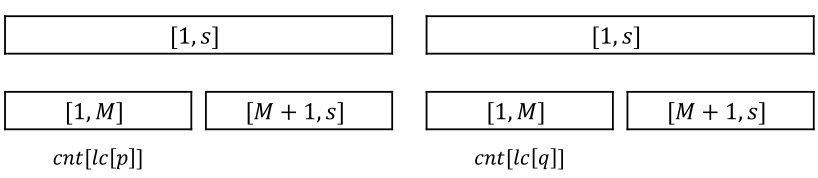
【例】第 k 小的数

- 对于询问 l, r, k，以 $rt[r]$ 为根的线段树中区间 $[L, R]$ 查询结果减去以 $rt[l-1]$ 为根的线段树中区间 $[L, R]$ 的查询结果就等于 $a[l:r]$ 中有多少个落在区间 $[L, R]$ 内。
- 例如，以 rt_3 为根的线段中区间 $[1, 7]$ 查询结果为 3，以 rt_1 为根的线段中区间 $[1, 7]$ 查询结果为 1，二者的差值为 2 表示 $a[2:3]$ 中落在 $[1, 7]$ 中有 2 个。

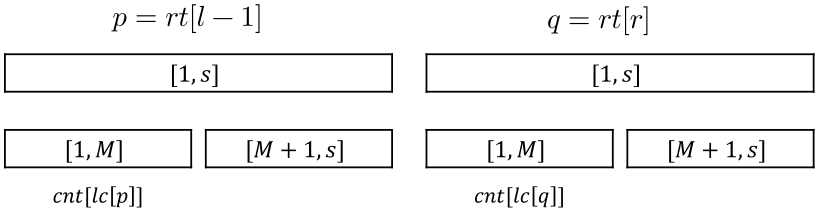


【例】第 k 小的数

- 于是分别查询 $q = rt[r]$ 的左孩子的值 $cnt[lc[q]]$ 和 $p = rt[l-1]$ 的左孩子的值 $cnt[lc[p]]$，二者之差为 $a[l:r]$ 落在 $[1, M]$ 的数的个数，而且这些数是 $a[l:r]$ 中较小的一部分。
- 设二者的差值为 $lcnt$，如果 $k \le lcnt$，说明第 k 小的数在 $[1, M]$ 中，否则 $k > lcnt$，说明第 k 小的数在 $[M+1, s]$ 中，且为其中第 $k - lcnt$ 小的数。



【例】第 k 小的数




```
int query(int p, int q, int L, int R, int k)
{
    if(L == R) return L;
    int lcnt = cnt[lc[q]] - cnt[lc[p]];
    if(k <= lcnt) return query(lc[p], lc[q], L, M, k);
    else return query(rc[p], rc[q], M + 1, R, k - lcnt);
}
// 主函数
int x = query(rt[l - 1], rt[r], 1, s, k);
printf("%d\n", b[x]);
```

练习

- 可持久化线段树 1(COGS 3311)
- 可持久化线段树 (COGS 2554)
- 第 k 小的数 (COGS 3537)
- 第 k 小的数 (COGS 930)
- 可持久化线段树 2(COGS 3322)
- 任务查询系统 [CQOI 2015](COGS 1936)
- 神秘数 [FJOI 2016](COGS 3327)
- 美味 [SCOI 2016](COGS 3338)
- 中值 [国家集训队 2012](COGS 1763)
- 混合果汁 [CTSC 2018](COGS 3372)