浅谈可追溯化数据结构

kczno1,King_George,diamond_duke

2019.1.

Introduction

现在有一个数据结构，支持一些修改操作和询问操作。

Introduction

现在有一个数据结构，支持一些修改操作和询问操作。
我们在进行了一些修改操作之后，发现某一次修改操作输入错了，希望修改这个修改操作，然后继续。

Introduction

现在有一个数据结构，支持一些修改操作和询问操作。
我们在进行了一些修改操作之后，发现某一次修改操作输入错了，希望修改这个修改操作，然后继续。
因此就有了可追溯化数据结构。

Definition

对于一个数据结构，维护一个操作序列，支持在某个位置插入一个操作，或删除一个操作，以及对按顺序执行这些操作后的状态进行一些询问。

Definition

对于一个数据结构，维护一个操作序列，支持在某个位置插入一个操作，或删除一个操作，以及对按顺序执行这些操作后的状态进行一些询问。
实现这些功能，就称为原数据结构的**部分可追溯化(Partial Retroactivity)**。

Definition

对于一个数据结构，维护一个操作序列，支持在某个位置插入一个操作，或删除一个操作，以及对按顺序执行这些操作后的状态进行一些询问。
实现这些功能，就称为原数据结构的**部分可追溯化(Partial Retroactivity)**。
如果还支持对操作序列的任意一个前缀进行询问，就称为原数据结构的**完全可追溯化(Full Retroactivity)**。

Union-Find

下面先以并查集的可追溯化为例。

Union-Find

下面先以并查集的可追溯化为例。
维护一个并查集的操作序列，支持以下操作：

Union-Find

下面先以并查集的可追溯化为例。
维护一个并查集的操作序列，支持以下操作：

1. Insert(t, union(x,y)) 在第 t 次操作后插入合并 x, y 所属集合的操作。

Union-Find

下面先以并查集的可追溯化为例。
维护一个并查集的操作序列，支持以下操作：

1. Insert(t, union(x,y)) 在第 t 次操作后插入合并 x, y 所属集合的操作。

为了简化问题，这里不考虑删除操作。
(有删除操作时的部分可追溯化实际上就是动态图连通性问题)

Partially Retroactive Union-Find

部分可追溯化并查集要求回答以下询问:

1. Query_sameset(x,y) 询问当前时刻 x 和 y 是否属于同一个集合。

这里认为所有操作发生的时刻互不相同且按操作序列上的顺序递增，当前时刻为 $+\infty$ 。

Partially Retroactive Union-Find

部分可追溯化并查集要求回答以下询问：

1. Query_sameset(x,y) 询问当前时刻 x 和 y 是否属于同一个集合。

这里认为所有操作发生的时刻互不相同且按操作序列上的顺序递增，当前时刻为 $+\infty$ 。
union 操作具有交换律，因此直接上并查集就好了。

Fully Retroactive Union-Find

完全可追溯化并查集要求回答以下询问：

1. Query_sameset(t,x,y) 询问在 t 时刻 x 和 y 是否属于同一个集合。

Fully Retroactive Union-Find

完全可追溯化并查集要求回答以下询问：

1. Query_sameset(t,x,y) 询问在 t 时刻 x 和 y 是否属于同一个集合。

lct 维护最小生成树即可。

Queue

维护一个队列的操作序列，支持以下操作：

Queue

维护一个队列的操作序列，支持以下操作：

1. Insert(t, enqueue(x)) 在第 t 次操作后插入将元素 x 入队操作。
2. Insert(t, dequeue()) 在第 t 次操作后插入出队操作。
3. Delete(t) 删除第 t 次操作。

Partially Retroactive Queue

部分可追溯化队列要能回答以下询问:

Partially Retroactive Queue

部分可追溯化队列要能回答以下询问：

1. Query(front()) 询问当前时刻队列的下一个将被弹出的元素。
2. Query(back()) 询问当前时刻队列中最后一个被加入的元素。

Partially Retroactive Queue

不难发现队列的操作都是可以合并的，无需考虑一个 enqueue(x) 操作和一个 dequeue() 操作的顺序。可以先将所有的 enqueue(x) 都做一遍，再做所有的 dequeue() 操作，所得到的队列仍然一样。

Partially Retroactive Queue

不难发现队列的操作都是可以合并的，无需考虑一个 enqueue(x) 操作和一个 dequeue() 操作的顺序。可以先将所有的 enqueue(x) 都做一遍，再做所有的 dequeue() 操作，所得到的队列仍然一样。
那么就只用维护 enqueue(x) 操作的先后顺序，即每次高效地在第 t 个数后插入一个数 x 就行了。

Partially Retroactive Queue

不难发现队列的操作都是可以合并的，无需考虑一个 enqueue(x) 操作和一个 dequeue() 操作的顺序。可以先将所有的 enqueue(x) 都做一遍，再做所有的 dequeue() 操作，所得到的队列仍然一样。

那么就只用维护 enqueue(x) 操作的先后顺序，即每次高效地在第 t 个数后插入一个数 x 就行了。

平衡树？

Partially Retroactive Queue

不难发现队列的操作都是可以合并的，无需考虑一个 enqueue(x) 操作和一个 dequeue() 操作的顺序。可以先将所有的 enqueue(x) 都做一遍，再做所有的 dequeue() 操作，所得到的队列仍然一样。

那么就只用维护 enqueue(x) 操作的先后顺序，即每次高效地在第 t 个数后插入一个数 x 就行了。

平衡树？

太难写、复杂度太高。

Partially Retroactive Queue

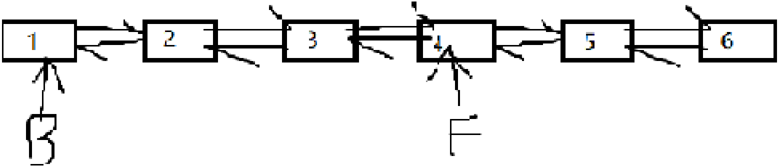
不难发现队列的操作都是可以合并的，无需考虑一个 enqueue(x) 操作和一个 dequeue() 操作的顺序。可以先将所有的 enqueue(x) 都做一遍，再做所有的 dequeue() 操作，所得到的队列仍然一样。
那么就只用维护 enqueue(x) 操作的先后顺序，即每次高效地在第 t 个数后插入一个数 x 就行了。
平衡树？
太难写、复杂度太高。
其实链表就行了。

Partially Retroactive Queue

将 enqueue(x) 操作按照时间顺序存在链表里。并维护两个指针 F 和 B 分别表示队头元素和队尾元素，这样回答 Query(front()) 和 Query(back()) 只用分别返回 F 和 B 的值就行了。

Partially Retroactive Queue

将 enqueue(x) 操作按照时间顺序存在链表里。并维护两个指针 F 和 B 分别表示队头元素和队尾元素，这样回答 Query(front()) 和 Query(back()) 只用分别返回 F 和 B 的值就行了。
enqueue(6) enqueue(5) enqueue(4) enqueue(3) enqueue(2) enqueue(1) dequeue() dequeue() 的例子如下：



Partially Retroactive Queue

一次 Insert(t, enqueue(x)) 对链表的修改：

Partially Retroactive Queue

一次 Insert(t, enqueue(x)) 对链表的修改：

1. 插入元素 x。
2. 若其成为了新的队尾则更新 B。
3. 如果在 F 后插入元素，则将 F 指向 F 的后继，否则不变。

Partially Retroactive Queue

一次 Delete(t) 操作对链表的修改，t 为 enqueue(x) 操作：

Partially Retroactive Queue

一次 Delete(t) 操作对链表的修改，t 为 enqueue(x) 操作：

1. 删除元素 x。
2. 若其原来为队尾则更新 B。
3. 如果在 F 后删除元素，则将 F 指向 F 的前驱，否则不变。

Partially Retroactive Queue

一次 Insert(t, dequeue()) 操作对链表的修改：

Partially Retroactive Queue

一次 Insert(t, dequeue()) 操作对链表的修改：
将 F 指向 F 的前驱。

Partially Retroactive Queue

一次 Delete(t) 操作对链表的修改，t 为 dequeue() 操作：

Partially Retroactive Queue

一次 Delete(t) 操作对链表的修改，t 为 dequeue() 操作：
将 F 指向 F 的后继。

Partially Retroactive Queue

一次 Delete(t) 操作对链表的修改，t 为 dequeue() 操作：
将 F 指向 F 的后继。
一次操作复杂度是 $O(1)$ 的。

Fully Retroactive Queue

完全可追溯化队列要能回答以下询问：

Fully Retroactive Queue

完全可追溯化队列要能回答以下询问：

1. Query(t, front()) 询问 t 时刻后队列的下一个将被弹出的元素。
2. Query(t, back()) 询问 t 时刻后队列中最后一个被加入的元素。

Fully Retroactive Queue

由于 enqueue(x) 操作和 dequeue() 操作是独立的，所以维护两棵平衡树 T_e 和 T_d 将两种操作分开处理。

Fully Retroactive Queue

由于 enqueue(x) 操作和 dequeue() 操作是独立的，所以维护两棵平衡树 T_e 和 T_d 将两种操作分开处理。
T_e 按时间顺序存下所有 enqueue(x) 操作。
T_d 按时间顺序存下所有 dequeue() 操作。

Fully Retroactive Queue

由于 enqueue(x) 操作和 dequeue() 操作是独立的，所以维护两棵平衡树 T_e 和 T_d 将两种操作分开处理。
T_e 按时间顺序存下所有 enqueue(x) 操作。
T_d 按时间顺序存下所有 dequeue() 操作。
修改操作直接在对应的平衡树上进行对应的操作即可，复杂度 $O(log\ m)$。

Fully Retroactive Queue

回答 Query(t, back()) 询问：

Fully Retroactive Queue

回答 Query(t, back()) 询问：
在 T_e 中找到 t 之前的最后一个操作。

Fully Retroactive Queue

回答 Query(t, front()) 询问:

Fully Retroactive Queue

回答 Query(t, front()) 询问：
先在 T_d 中找到 t 之前有多少个 dequeue() 操作，设次数为 k。
在 T_e 中找到第 $k+1$ 次插入的值，可以用平衡树上二分。

Fully Retroactive Queue

回答 Query(t, front()) 询问：
先在 T_d 中找到 t 之前有多少个 dequeue() 操作，设次数为 k。
在 T_e 中找到第 $k+1$ 次插入的值，可以用平衡树上二分。
一次询问复杂度是 $O(log\ m)$ 的。

Stack

维护一个栈的操作序列，支持以下操作：

Stack

维护一个栈的操作序列，支持以下操作：

1. Insert(t, push(x)) 在第 t 次操作后插入将元素 x 入栈操作。
2. Insert(t, pop()) 在第 t 次操作后插入出栈操作。
3. Delete(t) 删除第 t 次操作。

Stack

维护一个栈的操作序列，支持以下操作：

1. Insert(t, push(x)) 在第 t 次操作后插入将元素 x 入栈操作。
2. Insert(t, pop()) 在第 t 次操作后插入出栈操作。
3. Delete(t) 删除第 t 次操作。

由于部分可追溯化和完全可追溯化的单次操作复杂度都是 $O(\log\ m)$ 的，所以只交流一下完全可追溯化栈。

Fully Retroactive Stack

完全可追溯化栈要能回答以下询问:

Fully Retroactive Stack

完全可追溯化栈要能回答以下询问：

1. Query(t, top()) 询问 t 时刻后的栈顶元素。

Fully Retroactive Stack

考虑什么样的元素才能是最后的 top()。

Fully Retroactive Stack

考虑什么样的元素才能是最后的 top()。
可以发现一个操作 Insert(t', push(x)) 要是最后的 top() 元素当且仅当 $(t', t]$ 中的 push 操作数量与 pop 操作数量一样。

Fully Retroactive Stack

考虑什么样的元素才能是最后的 top()。
可以发现一个操作 Insert(t', push(x)) 要是最后的 top() 元素当且仅当 $(t', t]$ 中的 push 操作数量与 pop 操作数量一样。
所以问题就变成了如何快速找到这样一个操作。

Fully Retroactive Stack

用平衡树来解决。用一个平衡树按时间顺序维护所有操作，push(x) 操作记为 $+1$，pop() 操作记为 -1。

Fully Retroactive Stack

用平衡树来解决。用一个平衡树按时间顺序维护所有操作，
push(x) 操作记为 $+1$，pop() 操作记为 -1。
这样问题就变成了求最后一个后缀和为 1 的点。

Fully Retroactive Stack

用平衡树来解决。用一个平衡树按时间顺序维护所有操作，push(x) 操作记为 $+1$，pop() 操作记为 -1。
这样问题就变成了求最后一个后缀和为 1 的点。
由于数值只有 ±1，故区间 $[l_t, r_t]$ 内的后缀和肯定是一段连续的值 $[l_x, r_x]$，所以对平衡树每个节点记录 *min* 和 *max* 表示子树内后缀和的最小值、最大值，以及子树内权值和 *sum*，查找后缀和为 1 的点就可以在平衡树上二分了。

Fully Retroactive Stack

用平衡树来解决。用一个平衡树按时间顺序维护所有操作，push(x) 操作记为 $+1$，pop() 操作记为 -1。
这样问题就变成了求最后一个后缀和为 1 的点。
由于数值只有 ± 1，故区间 $[l_t, r_t]$ 内的后缀和肯定是一段连续的值 $[l_x, r_x]$，所以对平衡树每个节点记录 min 和 max 表示子树内后缀和的最小值、最大值，以及子树内权值和 sum，查找后缀和为 1 的点就可以在平衡树上二分了。
单次修改和询问的复杂度是 $O(logm)$ 的。

Deque

维护一个双端队列的操作序列，支持以下操作：

Deque

维护一个双端队列的操作序列，支持以下操作：

1. Insert(t, pushL(x)) 在第 t 次操作后插入将元素 x 从左端加入双端队列操作。
2. Insert(t, pushR(x)) 在第 t 次操作后插入将元素 x 从右端加入双端队列操作。
3. Insert(t, popL()) 在第 t 次操作后插入弹出左端第一个数操作。
4. Insert(t, popR()) 在第 t 次操作后插入弹出右端第一个数操作。
5. Delete(t) 删除第 t 次操作。

Deque

维护一个双端队列的操作序列，支持以下操作：

1. Insert(t, pushL(x)) 在第 t 次操作后插入将元素 x 从左端加入双端队列操作。
2. Insert(t, pushR(x)) 在第 t 次操作后插入将元素 x 从右端加入双端队列操作。
3. Insert(t, popL()) 在第 t 次操作后插入弹出左端第一个数操作。
4. Insert(t, popR()) 在第 t 次操作后插入弹出右端第一个数操作。
5. Delete(t) 删除第 t 次操作。

由于部分可追溯化和完全可追溯化的单次操作复杂度都是 $O(\log\ m)$ 的，所以只交流一下完全可追溯化双端队列。

Fully Retroactive Deque

完全可追溯化双端队列要能回答以下询问:

Fully Retroactive Deque

完全可追溯化双端队列要能回答以下询问：

1. Query(t, left()) 询问 t 时刻后双端队列中最左端元素的值。
2. Query(t, right()) 询问 t 时刻后双端队列中最右端元素的值。

Fully Retroactive Deque

既然有了完全可追溯化栈，能否将双端队列拆成两半，对 pushL(x) 和 popL() 操作维护一个左端栈，对 pushR(x) 和 popR() 操作维护一个右端栈？

Fully Retroactive Deque

既然有了完全可追溯化栈，能否将双端队列拆成两半，对 pushL(x) 和 popL() 操作维护一个左端栈，对 pushR(x) 和 popR() 操作维护一个右端栈？

由于双端队列一侧的操作如果不合法了会影响到另一侧，所以最后一个后缀和为 1 的点不一定就是答案，譬如说下面这个例子：

Fully Retroactive Deque

既然有了完全可追溯化栈，能否将双端队列拆成两半，对 pushL(x) 和 popL() 操作维护一个左端栈，对 pushR(x) 和 popR() 操作维护一个右端栈？
由于双端队列一侧的操作如果不合法了会影响到另一侧，所以最后一个后缀和为 1 的点不一定就是答案，譬如说下面这个例子：
pushR(1) popL() pushL(2) Query(t, right())

Fully Retroactive Deque

既然有了完全可追溯化栈，能否将双端队列拆成两半，对 pushL(x) 和 popL() 操作维护一个左端栈，对 pushR(x) 和 popR() 操作维护一个右端栈？

由于双端队列一侧的操作如果不合法了会影响到另一侧，所以最后一个后缀和为 1 的点不一定就是答案，譬如说下面这个例子：

pushR(1) popL() pushL(2) Query(t, right())

当然，这个错误算法也给了我们一些启发。一个值肯定是被一次 pushL(x) 操作或者一次 pushR(x) 操作加入双端队列中的，如果能得到 pushL(x) 操作中最有可能是答案的和 pushR(x) 操作中最有可能是答案的再判断一下就行了。

Fully Retroactive Deque

考虑一种手写双端队列的方法。建一个数组 a，初始时 $L = 1$，$R = 0$。

1. pushL(x) 操作对应 a[- -L] = x
2. pushR(x) 操作对应 a[++R] = x
3. popL() 操作对应 ++L
4. popR() 操作对应 - -R

Fully Retroactive Deque

考虑一种手写双端队列的方法。建一个数组 a，初始时 $L = 1$，$R = 0$。

1. pushL(x) 操作对应 a[- -L] = x
2. pushR(x) 操作对应 a[++R] = x
3. popL() 操作对应 ++L
4. popR() 操作对应 - -R

这样一次询问只要知道 i 和 a[i] 就行了。

Fully Retroactive Deque

维护两棵平衡树 T_l 和 T_r，T_l 按时间存下 pushL(x) 操作和 popL() 操作，权值分别为 $+1$ 和 -1，T_r 一样。求 t 时刻的 i 的值只用在 T_l 或 T_r 中求一下前缀和就行了。

Fully Retroactive Deque

维护两棵平衡树 T_l 和 T_r，T_l 按时间存下 pushL(x) 操作和 popL() 操作，权值分别为 $+1$ 和 -1，T_r 一样。求 t 时刻的 i 的值只用在 T_l 或 T_r 中求一下前缀和就行了。
a[i] 的值肯定是最后一次 pushL(x) 后 $L = i$ 的操作或者最后一次 pushR(x) 后 $R = i$ 的操作。这两个都可以在平衡树上二分出后缀和为 k 的点，取两者较晚的一次操作即可。

Fully Retroactive Deque

维护两棵平衡树 T_l 和 T_r， T_l 按时间存下 pushL(x) 操作和 popL() 操作，权值分别为 $+1$ 和 -1， T_r 一样。求 t 时刻的 i 的值只用在 T_l 或 T_r 中求一下前缀和就行了。
a[i] 的值肯定是最后一次 pushL(x) 后 $L = i$ 的操作或者最后一次 pushR(x) 后 $R = i$ 的操作。这两个都可以在平衡树上二分出后缀和为 k 的点，取两者较晚的一次操作即可。
单次修改和询问的复杂度是 $O(logm)$ 的。

Fully Retroactive Deque

维护两棵平衡树 T_l 和 T_r，T_l 按时间存下 pushL(x) 操作和 popL() 操作，权值分别为 $+1$ 和 -1，T_r 一样。求 t 时刻的 i 的值只用在 T_l 或 T_r 中求一下前缀和就行了。
a[i] 的值肯定是最后一次 pushL(x) 后 $L = i$ 的操作或者最后一次 pushR(x) 后 $R = i$ 的操作。这两个都可以在平衡树上二分出后缀和为 k 的点，取两者较晚的一次操作即可。
单次修改和询问的复杂度是 $O(logm)$ 的。
可以发现回答询问时并没有用到最左端点和最右端点这个限制，所以回答双端队列中任意一个元素的值也是可以的。

Priority Queue

维护一个优先队列（堆）的操作序列，支持以下操作：

Priority Queue

维护一个优先队列（堆）的操作序列，支持以下操作：

1. Insert(t, insert(k)) 在第 t 次操作后插入将元素 k 入堆操作。
2. Insert(t, delete_min()) 在第 t 次操作后插入弹出最小元素的操作。
3. Delete(t) 删除第 t 次操作。

Partially Retroactive Priority Queue

部分可追溯化优先队列要能回答以下询问：

Partially Retroactive Priority Queue

部分可追溯化优先队列要能回答以下询问：

1. Query() 询问当前时刻队列的最小元素。

Partially Retroactive Priority Queue

记 Q_t 表示 t 时刻队列内的元素，我们只要能够询问 Q_{now} 即可。

Partially Retroactive Priority Queue

记 Q_t 表示 t 时刻队列内的元素，我们只要能够询问 Q_{now} 即可。
考虑 Insert(t, insert(k)) 对 Q_{now} 的影响。可以发现他相当于是在 Q_{now} 中插入了

$$\max\{k, k' \mid k' \text{ deleted at time} \geq t\}$$

Partially Retroactive Priority Queue

记 Q_t 表示 t 时刻队列内的元素，我们只要能够询问 Q_{now} 即可。
考虑 Insert(t, insert(k)) 对 Q_{now} 的影响。可以发现他相当于是在 Q_{now} 中插入了

$$\max\{k, k' \mid k' \text{ deleted at time} \geq t\}$$

问题在于删除了哪些元素并不好进行维护。

Partially Retroactive Priority Queue

我们称一个时刻 t 是一个桥(**bridge**)，当且仅当 $Q_t \subseteq Q_{now}$。

Partially Retroactive Priority Queue

我们称一个时刻 t 是一个桥**(bridge)**，当且仅当 $Q_t \subseteq Q_{now}$。
若 t' 是 t 时刻前的第一个桥，则：

$$\begin{aligned} &\max\{k' \mid k' \text{ deleted at time} \geq t\} \\ =& \max\{k' \notin Q_{now} \mid k' \text{ inserted at time} \geq t'\} \end{aligned} \tag{1}$$

Partially Retroactive Priority Queue

我们称一个时刻 t 是一个桥**(bridge)**，当且仅当 $Q_t \subseteq Q_{now}$。若 t' 是 t 时刻前的第一个桥，则：

$$\begin{aligned} &\max\{k' \mid k' \text{ deleted at time} \geq t\} \\ = &\max\{k' \notin Q_{now} \mid k' \text{ inserted at time} \geq t'\} \end{aligned} \quad (1)$$

类似地，我们考虑 Insert(t, delete_min()) 的影响。若 t' 是 t 后面的第一个桥，则影响即为删除

$$\min\{k' \in Q_{now} \mid k' \text{ inserted at time} \leq t'\} \quad (2)$$

Partially Retroactive Priority Queue

而对于撤销一个 insert(k) 操作，如果 k 在 Q_{now} 中，那么影响就是删除 k，否则就是删除

$$\min\{k' \in Q_{now} \mid k' \text{ inserted at time} \leq t'\}$$

Partially Retroactive Priority Queue

而对于撤销一个 insert(k) 操作，如果 k 在 Q_{now} 中，那么影响就是删除 k，否则就是删除

$$\min\{k' \in Q_{now} \mid k' \text{ inserted at time} \leq t'\}$$

类似地，撤掉一个 delete_min() 的操作的影响即为

$$\max\{k' \notin Q_{now} \mid k' \text{ inserted at time} \geq t'\}$$

Partially Retroactive Priority Queue

而对于撤销一个 insert(k) 操作，如果 k 在 Q_{now} 中，那么影响就是删除 k，否则就是删除

$$\min\{k' \in Q_{now} \mid k' \text{ inserted at time} \leq t'\}$$

类似地，撤掉一个 delete_min() 的操作的影响即为

$$\max\{k' \notin Q_{now} \mid k' \text{ inserted at time} \geq t'\}$$

于是我们可以进行 Q_{now} 的维护。

Partially Retroactive Priority Queue

考虑使用平衡树维护 Q_{now}。为了处理修改，我们需要另外的平衡树。

Partially Retroactive Priority Queue

考虑使用平衡树维护 Q_{now}。为了处理修改，我们需要另外的平衡树。
我们用另外一棵平衡树维护插入。对于每个节点 u，我们存下

$$\max\{k' \notin Q_{now} \mid k' \text{ inserted in } u\text{'s subtree}\}$$

以及

$$\min\{k' \in Q_{now} \mid k' \text{ inserted in } u\text{'s subtree}\}$$

Partially Retroactive Priority Queue

我们还需要一个平衡树用来求出桥。我们存储 Insert 的操作，并如下赋权：

1. 对于操作 insert(k)，且 $k \in Q_{now}$，我们赋权 0；
2. 对于操作 insert(k)，且 $k \notin Q_{now}$，我们赋权 $+1$；
3. 对于操作 delete_min()，我们赋权 -1。

我们在每个节点上存储子树和，前缀和的最小值，以及前缀和的最大值。

Partially Retroactive Priority Queue

因为桥等价于前缀和等于 0，所以我们对于每次修改，可以通过树上二分在 $\mathcal{O}(\log_2 m)$ 的时间内找到 t 前面的一个桥。

Partially Retroactive Priority Queue

因为桥等价于前缀和等于 0，所以我们对于每次修改，可以通过树上二分在 $\mathcal{O}(\log_2 m)$ 的时间内找到 t 前面的一个桥。
在求出桥之后，我们可以通过 1 式和 2 式来求出这次操作对 Q_{now} 的影响，从而实时维护 Q_{now}。

Partially Retroactive Priority Queue

因为桥等价于前缀和等于 0，所以我们对于每次修改，可以通过树上二分在 $\mathcal{O}(\log_2 m)$ 的时间内找到 t 前面的一个桥。
在求出桥之后，我们可以通过 1 式和 2 式来求出这次操作对 Q_{now} 的影响，从而实时维护 Q_{now}。
求出对 Q_{now} 的影响后，可以同时维护另外两棵平衡树的信息。

Partially Retroactive Priority Queue

因为桥等价于前缀和等于 0，所以我们对于每次修改，可以通过树上二分在 $\mathcal{O}(\log_2 m)$ 的时间内找到 t 前面的一个桥。
在求出桥之后，我们可以通过 1 式和 2 式来求出这次操作对 Q_{now} 的影响，从而实时维护 Q_{now}。
求出对 Q_{now} 的影响后，可以同时维护另外两棵平衡树的信息。
时间复杂度：单次修改、询问均为 $\mathcal{O}(\log_2 m)$。

Fully Retroactive Priority Queue

Fully Retroactive Priority Queue

堆的完全可追溯化比较困难，因此使用的方法都是比较通用的，适用范围比较广的方法。

Fully Retroactive Priority Queue

有一个通用的方法可以将 full 的复杂度做到 partial 的复杂度乘上 $O(\sqrt{m})$ 。

Fully Retroactive Priority Queue

有一个通用的方法可以将 full 的复杂度做到 partial 的复杂度乘上 $O(\sqrt{m})$ 。
对操作序列分块，每 $O(\sqrt{m})$ 个操作分成一块。

Fully Retroactive Priority Queue

有一个通用的方法可以将 full 的复杂度做到 partial 的复杂度乘上 $O(\sqrt{m})$ 。
对操作序列分块，每 $O(\sqrt{m})$ 个操作分成一块。
对于每个块，将这个块之前的所有操作组成的操作序列用 partial 的方法进行维护。

Fully Retroactive Priority Queue

有一个通用的方法可以将 full 的复杂度做到 partial 的复杂度乘上 $O(\sqrt{m})$ 。
对操作序列分块，每 $O(\sqrt{m})$ 个操作分成一块。
对于每个块，将这个块之前的所有操作组成的操作序列用 partial 的方法进行维护。
插入，删除和询问都可以非常暴力的做。

Fully Retroactive Priority Queue

通过一个叫 hierarchical checkpointing 的方法，可以做到单次插入删除 $O(\log^2 m)$ ，询问 $O(\log^2 m \log\log m)$ 。

Fully Retroactive Priority Queue

用一棵替罪羊树维护整个操作序列。

Fully Retroactive Priority Queue

用一棵替罪羊树维护整个操作序列。
对于每个结点，对这个结点代表的子树组成的操作序列，维护部分可追溯化堆，将维护的结果记作两棵平衡树 Q_{now} 和 Q_{del} ，分别包含没被删除的元素和被删除的元素(如果某次删除的元素的不存在，则认为是无穷大)。

Fully Retroactive Priority Queue

用一棵替罪羊树维护整个操作序列。
对于每个结点，对这个结点代表的子树组成的操作序列，维护部分可追溯化堆，将维护的结果记作两棵平衡树 Q_{now} 和 Q_{del} ，分别包含没被删除的元素和被删除的元素(如果某次删除的元素的不存在，则认为是无穷大)。
插入结点时对每个祖先都执行一次插入。
如果导致一个子树不平衡则需要进行重构(重构的实现后面会讲)。

Fully Retroactive Priority Queue

用一棵替罪羊树维护整个操作序列。
对于每个结点，对这个结点代表的子树组成的操作序列，维护部分可追溯化堆，将维护的结果记作两棵平衡树 Q_{now} 和 Q_{del} ，分别包含没被删除的元素和被删除的元素(如果某次删除的元素的不存在，则认为是无穷大)。
插入结点时对每个祖先都执行一次插入。
如果导致一个子树不平衡则需要进行重构(重构的实现后面会讲)。
删除是类似的。
为了实现删除，可以打删除标记，也可以只在叶节点存储信息，就是类似线段树的结构。

Fully Retroactive Priority Queue

询问时，需要合并 $O(\log m)$ 个部分可追溯化堆，然后对合并的结果进行查询。

Fully Retroactive Priority Queue

询问时，需要合并 $O(\log m)$ 个部分可追溯化堆，然后对合并的结果进行查询。
引理：将两个部分可追溯化堆 Q_1, Q_2 合并的结果记作 Q_3 ，则有

$$Q_{3,now} = Q_{2,now} \bigcup max\text{-}A\{Q_{1,now} \cup Q_{2,del}\}$$

$$Q_{3,del} = Q_{1,del} \bigcup min\text{-}D\{Q_{1,now} \cup Q_{2,del}\}$$

这里 $A = |Q_{1,now}|$ ，$D = |Q_{2,del}|$ ，$max\text{-}C\{S\}$ 表示集合 S 中前 C 大的元素。

Fully Retroactive Priority Queue

为了快速合并，我们用多棵平衡树的并来表示 Q_{now} 和 Q_{del} 。

Fully Retroactive Priority Queue

为了快速合并，我们用多棵平衡树的并来表示 Q_{now} 和 Q_{del} 。定义可并的部分可追溯化堆 Q^k ，表示 Q^k_{now} 和 Q^k_{del} 都是用不超过 k 棵平衡树表示的。

Fully Retroactive Priority Queue

为了快速合并，我们用多棵平衡树的并来表示 Q_{now} 和 Q_{del} 。定义可并的部分可追溯化堆 Q^k ，表示 Q^k_{now} 和 Q^k_{del} 都是用不超过 k 棵平衡树表示的。
那么如果现在要合并 Q^k_1 和 Q^k_2 ，

Fully Retroactive Priority Queue

为了快速合并，我们用多棵平衡树的并来表示 Q_{now} 和 Q_{del} 。
定义可并的部分可追溯化堆 Q^k ，表示 Q^k_{now} 和 Q^k_{del} 都是用不超过 k 棵平衡树表示的。
那么如果现在要合并 Q^k_1 和 Q^k_2 ，
令 $T = Q_{1,now} \bigcup Q_{2,del}$ ，记 $T_i (1 \leq i \leq 2 \times k)$ 为 T 中第 i 棵平衡树，

Fully Retroactive Priority Queue

为了快速合并，我们用多棵平衡树的并来表示 Q_{now} 和 Q_{del} 。
定义可并的部分可追溯化堆 Q^k ，表示 Q^k_{now} 和 Q^k_{del} 都是用不超过 k 棵平衡树表示的。
那么如果现在要合并 Q^k_1 和 Q^k_2 ，
令 $T = Q_{1,now} \bigcup Q_{2,del}$ ，记 $T_i(1 \leq i \leq 2 \times k)$ 为 T 中第 i 棵平衡树，
令 $x = max\text{-}A\{T\}$ ，

Fully Retroactive Priority Queue

为了快速合并，我们用多棵平衡树的并来表示 Q_{now} 和 Q_{del} 。定义可并的部分可追溯化堆 Q^k ，表示 Q^k_{now} 和 Q^k_{del} 都是用不超过 k 棵平衡树表示的。

那么如果现在要合并 Q^k_1 和 Q^k_2 ,

令 $T = Q_{1,now} \bigcup Q_{2,del}$ ，记 $T_i(1 \leq i \leq 2 \times k)$ 为 T 中第 i 棵平衡树，

令 $x = max\text{-}A\{T\}$ ，

将 T_i 根据 x 分裂成两棵平衡树 $T_{i,<}$ 和 $T_{i,>}$

Fully Retroactive Priority Queue

为了快速合并，我们用多棵平衡树的并来表示 Q_{now} 和 Q_{del} 。
定义可并的部分可追溯化堆 Q^k ，表示 Q^k_{now} 和 Q^k_{del} 都是用不超过 k 棵平衡树表示的。
那么如果现在要合并 Q^k_1 和 Q^k_2 ，
令 $T = Q_{1,now} \bigcup Q_{2,del}$ ，记 $T_i(1 \leq i \leq 2 \times k)$ 为 T 中第 i 棵平衡树，
令 $x = max\text{-}A\{T\}$ ，
将 T_i 根据 x 分裂成两棵平衡树 $T_{i,<}$ 和 $T_{i,>}$
则 $Q_{3,now} = Q_{2,now} \bigcup T_{1,>}, \cdots, T_{2k,>}$
$Q_{3,del} = Q_{1,del} \bigcup T_{1,<}, \cdots, T_{2k,<}$

Fully Retroactive Priority Queue

为了快速合并，我们用多棵平衡树的并来表示 Q_{now} 和 Q_{del} 。
定义可并的部分可追溯化堆 Q^k ，表示 Q^k_{now} 和 Q^k_{del} 都是用不超过 k 棵平衡树表示的。
那么如果现在要合并 Q_1^k 和 Q_2^k ,
令 $T = Q_{1,now} \bigcup Q_{2,del}$ ，记 $T_i (1 \le i \le 2 \times k)$ 为 T 中第 i 棵平衡树，
令 $x = max\text{-}A\{T\}$ ，
将 T_i 根据 x 分裂成两棵平衡树 $T_{i,<}$ 和 $T_{i,>}$
则 $Q_{3,now} = Q_{2,now} \bigcup T_{1,>}, \cdots, T_{2k,>}$
$Q_{3,del} = Q_{1,del} \bigcup T_{1,<}, \cdots, T_{2k,<}$
这样我们就由 Q_1^k 和 Q_2^k 得到了 Q_3^{3k} 。
时间复杂度 $O(k \log m)$

Fully Retroactive Priority Queue

求解 $max\text{-}A\{T\}$ 有点麻烦，这里不再赘述，有兴趣的同学可以自己去看论文。

Fully Retroactive Priority Queue

假设现在要合并 k 个部分可追溯化堆 $Q_{1..k}$ ，记合并的结果为 Q_* 。

Fully Retroactive Priority Queue

假设现在要合并 k 个部分可追溯化堆 $Q_{1..k}$ ，记合并的结果为 Q_* 。
如果直接分治，使用上面的方法合并，那么 $Q_{*,now}$ 和 $Q_{*,now}$ 会用 $O(3^{\log_2 k}) = O(k^{\log_2 3})$ 棵平衡树来表示。

Fully Retroactive Priority Queue

假设现在要合并 k 个部分可追溯化堆 $Q_{1..k}$ ，记合并的结果为 Q_* 。
如果直接分治，使用上面的方法合并，那么 $Q_{*,now}$ 和 $Q_{*,now}$ 会用 $O(3^{\log_2 k}) = O(k^{\log_2 3})$ 棵平衡树来表示。
引理：存在 a_i 和 a'_i ，使得

$$Q_{*,now} = \bigcup_i max\text{-}a_i Q_{i,now} \bigcup_i max\text{-}a'_i Q_{i,del}$$

Fully Retroactive Priority Queue

假设现在要合并 k 个部分可追溯化堆 $Q_{1..k}$ ，记合并的结果为 Q_* 。
如果直接分治，使用上面的方法合并，那么 $Q_{*,now}$ 和 $Q_{*,now}$ 会用 $O(3^{\log_2 k}) = O(k^{\log_2 3})$ 棵平衡树来表示。
引理：存在 a_i 和 a'_i ，使得

$$Q_{*,now} = \bigcup_i max\text{-}a_i Q_{i,now} \bigcup_i max\text{-}a'_i Q_{i,del}$$

因此 $Q*, now$ 和 $Q*, del$ 都可以用 $2k$ 棵平衡树来表示。

Fully Retroactive Priority Queue

假设现在要合并 k 个部分可追溯化堆 $Q_{1..k}$ ，记合并的结果为 Q_* 。
如果直接分治，使用上面的方法合并，那么 $Q_{*,now}$ 和 $Q_{*,now}$ 会用 $O(3^{\log_2 k}) = O(k^{\log_2 3})$ 棵平衡树来表示。
引理：存在 a_i 和 a'_i ，使得

$$Q_{*,now} = \bigcup_i max\text{-}a_i Q_{i,now} \bigcup_i max\text{-}a'_i Q_{i,del}$$

因此 $Q*, now$ 和 $Q*, del$ 都可以用 $2k$ 棵平衡树来表示。
因此合并 k 个部分可追溯化堆的时间为 $O(k \log k \log m)$ 。
询问时的复杂度为 $O(\log^2 m \log \log m)$ 。

Fully Retroactive Priority Queue

关于子树重构，只需要使用归并操作来合并 $Q_{1,now}$ 和 $Q_{2,del}$ 。

Fully Retroactive Priority Queue

关于子树重构，只需要使用归并操作来合并 $Q_{1,now}$ 和 $Q_{2,del}$ 。
对于大小的 m 的子树重构的复杂度为 $O(m \log m)$ 。

Fully Retroactive Priority Queue

关于子树重构，只需要使用归并操作来合并 $Q_{1,now}$ 和 $Q_{2,del}$ 。
对于大小的 m 的子树重构的复杂度为 $O(m\log m)$ 。
所以插入删除的复杂度为均摊 $O(\log^2 m)$ 。

End

谢谢聆听。祝大家在 WC 取得好成绩!

Referrences

ERIK D. DEMAINE, JOHN IACONO and STEFAN LANGERMAN. Retroactive Data Structures.

Erik D. Demaine , Tim Kaler, Quanquan Liu, Aaron Sidford, and Adam Yedidia. Polylogarithmic Fully Retroactive Priority Queues via Hierarchical Checkpointing.