Treap

前置知识：朴素二叉搜索树，堆基础。

简介

Treap（树堆）是一种 **弱平衡** 的 **二叉搜索树**。

Treap 的结点除了被维护的 **权值**（val）之外，还附加了一个随机的 **优先级**（$priority$）。其中，权值满足二叉搜索树性质，优先级满足堆性质（小根堆或大根堆）。

其中，二叉搜索树的性质是指：

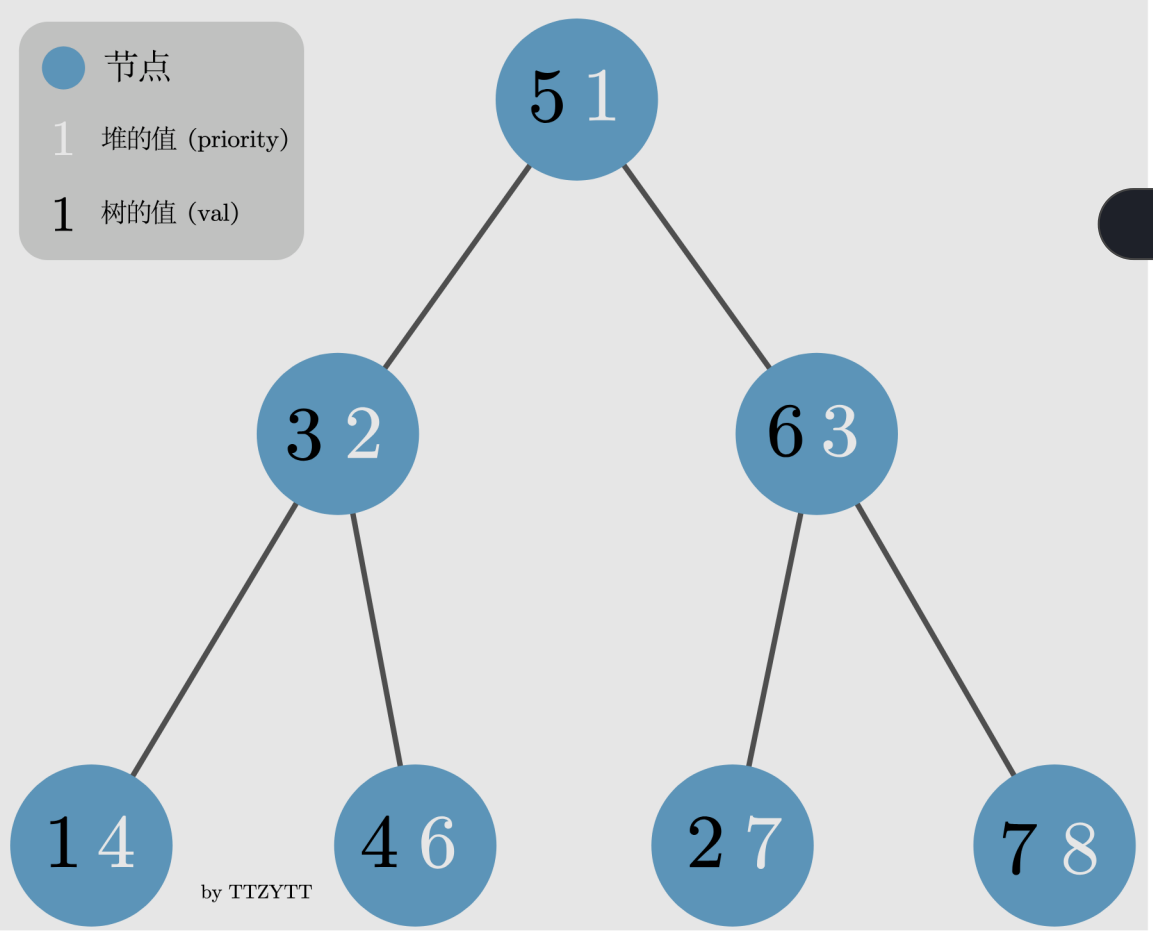
- 左子节点的权值（val）比父节点小。
- 右子节点的权值（val）比父节点大。

堆的性质是：

- 子节点优先级（$priority$）比父节点大或小（取决于是小根堆还是大根堆）。

不难看出，如果用的是同一个值，那么这两种数据结构在组合后会变成一条链，所以我们再在搜索树的基础上，引入一个给堆的值 $priority$。对于 val 值，我们维护搜索树的性质，对于 $priority$ 值，我们维护堆的性质。其中 $priority$ 这个值是随机给出的。

下图就是一个 Treap 的例子（这里使用的是小根堆，即根节点的优先级最小）。



那我们为什么需要大费周章的去让这个数据结构符合树和堆的性质，并且随机给出堆的值呢？

要理解这个，首先需要理解朴素二叉搜索树的问题。在给朴素搜索树插入一个新节点时，我们需要从这个搜索树的根节点开始递归，如果新节点比当前节点小，那就向左递归，反之亦然。

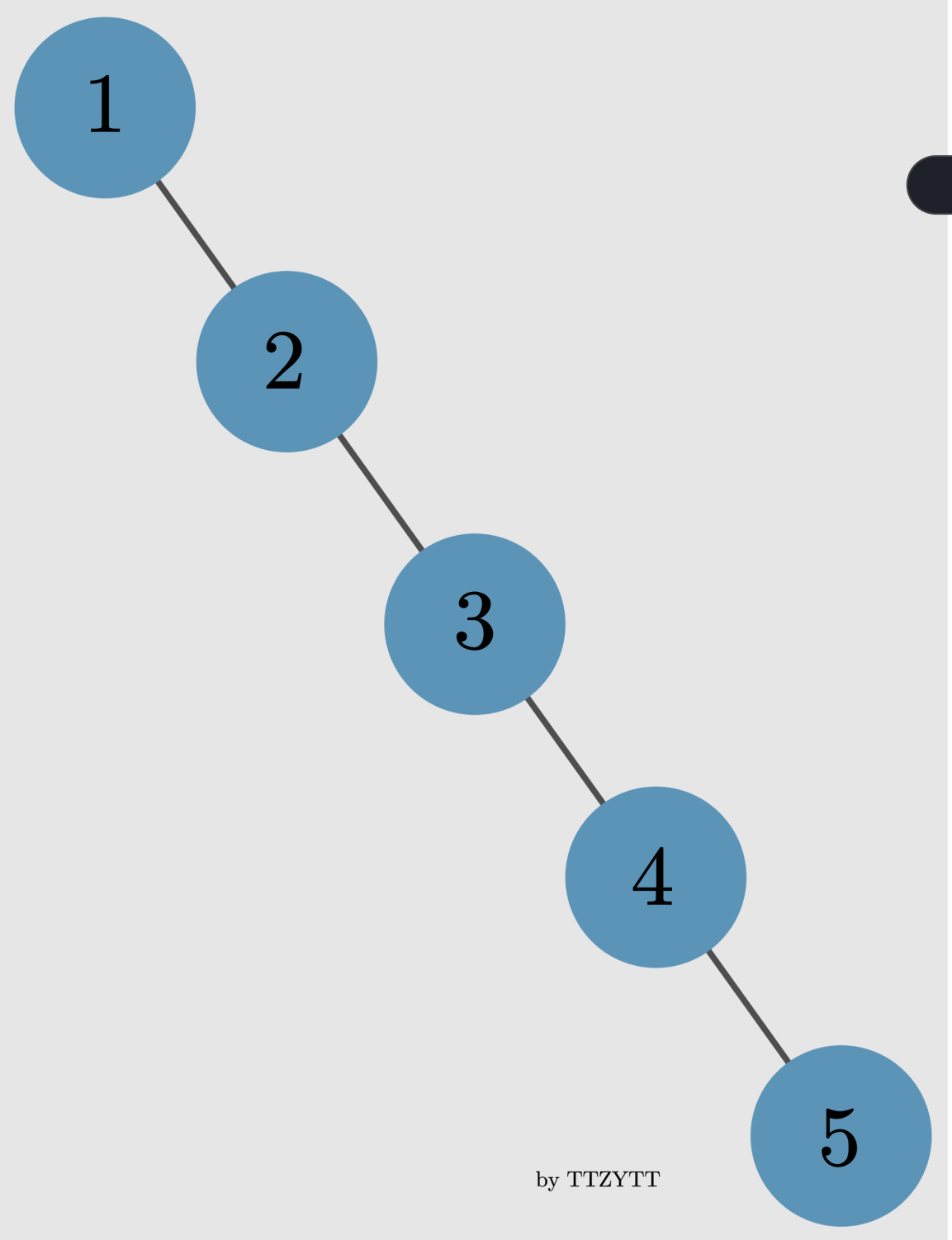
最后当发现当前节点没有子节点时，就根据新节点的值的大小，让新节点成为当前节点的左或右子节点。

如果插入结点的权值是随机的（换言之，是随机插入的），那这个朴素搜索树的高度较小（接近 $\log n$，其中 n 为结点数），而每一层的节点数较多，即它的形状会非常的「胖」。上图的 Treap 就是一个例子。因此此时的任意操作复杂度都将会是 $O(\log n)$ 左右。

不过，这只是在随机情况下的复杂度，如果我们按照下面这个非常有序的顺序给一个朴素的搜索树插入节点：

```
1 2 3 4 5
```

那么这棵树将会退化成链，即变得非常「瘦长」（每次插入的节点都比前面的大，所以都被安排到右子节点了）：



不难看出，查询的复杂度也从 $O(\log n)$ 变成了 $O(n)$.

而 treap 为了解决这个问题、达到一个较为「平衡」的状态，通过维护随机的优先级满足堆性质，「打乱」了节点的插入顺序，从而让二叉搜索树达到了理想的复杂度，避免了退化成链的问题。

Treap 复杂度的证明

由于 treap 各种操作的复杂度都和所操作的结点的深度有关，我们首先证明，所有结点的期望高度都是 $O(\log n)$.

记号约定

为了方便表述，我们约定：

- n 是节点个数。
- Treap 结点中满足二叉搜索树性质的称为 **权值**，满足堆性质的（也就是随机的）称为 **优先级**。不妨设优先级满足小根堆性质。
- x_k 表示权值第 k 小的结点。
- $X_{i,j}$ 表示集合 $\{x_i, x_{i+1}, \cdots, x_{j-1}, x_j\}$，即按权值升序排列后第 i 个到第 j 个的结点构成的集合。
- $\mathrm{dep}(x)$ 表示结点 x 的深度。规定根节点的深度是 0.
- $Y_{i,j}$ 是一个指示器随机变量，当 x_i 是 x_j 的祖先时值为 1，否则为 0. 特别地，$Y_{i,i} = 0$.
- $\Pr(A)$ 表示事件 A 发生的概率。

树高的证明

由于结点 x_i 的深度等于它祖先的个数，因此有

$$\mathrm{dep}(x_i) = \sum_{k=1}^{n} Y_{k,i}$$

那么根据期望的线性性，有

$$E(\mathrm{dep}(x_i)) = E\left(\sum_{k=1}^{n} Y_{k,i}\right) = \sum_{k=1}^{n} E(Y_{k,i})$$

由于 $Y_{k,i}$ 是指示器随机变量，它的期望就等于它为 1 的概率，因此

$$E(\mathrm{dep}(x_i)) = \sum_{k=1}^{n} \Pr(Y_{k,i} = 1)$$

我们先证明引理：$Y_{i,j} = 1$ 当且仅当 x_i 的优先级是 $X_{i,j}$ 中最小的。

引理的证明

证明：考虑分类讨论 x_i 和 x_j 的情况。

1. 若 x_i 是根节点：由于优先级满足小根堆性质，x_i 的优先级最小，并且对于任意的 x_j，x_i 都是 x_j 的祖先。
2. 若 x_j 是根节点：同理，x_j 优先级最小，因此 x_i 不是 $X_{i,j}$ 中优先级最小的；同时 x_i 也不是 x_j 的祖先。
3. 若 x_i 和 x_j 在根节点的两个子树中（一左一右），那么根节点 $r \in X_{i,j}$. 因此 x_i 的优先级不可能是 $X_{i,j}$ 中最小的（因为根节点的比它小）。同时，由于 x_i 和 x_j 分属两个子树，x_i 也不是 x_j 的祖先。
4. 若 x_i 和 x_j 在根节点的同一个子树中，此时可以将这个子树单独拿出来作为一棵新的 treap，递归进行上面的证明即可。□

那么根据引理，深度的期望可以转化成

$$E(\operatorname{dep}(x_i)) = \sum_{k=1}^{n} \Pr(x_k = \min X_{i,k} \wedge k \neq i)$$

又因为结点的优先级是随机的，我们假定集合 $X_{i,j}$ 中任何一个结点的优先级最小的概率都相同，那么

$$\begin{aligned}
E(\operatorname{dep}(x_i)) &= \sum_{k=1}^{n} \Pr(x_k = \min X_{i,k} \wedge k \neq i) \\
&= \sum_{k=1}^{n} \Pr(x_k = \min X_{i,k}) - 1 \\
&= \sum_{k=1}^{n} \frac{1}{|i-k|+1} - 1 \\
&= \sum_{k=1}^{i-1} \frac{1}{i-k+1} + \sum_{k=i+1}^{n} \frac{1}{k-i+1} \\
&= \sum_{j=2}^{i} \frac{1}{j} + \sum_{j=2}^{n-i+1} \frac{1}{j} \\
&\leq 2\sum_{j=2}^{n} \frac{1}{j} < 2\sum_{j=2}^{n} \int_{j-1}^{j} \frac{1}{x}\mathrm{d}x \\
&= 2\int_{1}^{n} \frac{1}{x}\mathrm{d}x = 2\ln n = O(\log n)
\end{aligned}$$

因此每个结点的期望高度都是 $O(\log n)$.

而朴素的二叉搜索树的操作的复杂度均是 $O(h)$，同时 treap 维护堆性质的复杂度也是 $O(h)$ 的，因此 treap 各种操作的期望复杂度都是 $O(\log n)$.

期望复杂度的感性理解

首先，我们需要认识到一个节点的 $priority$ 属性是和它所在的层数有直接关联的。再回忆堆的性质：

- 子节点值（$priority$）比父节点大或小（取决于是小根堆还是大根堆）

我们发现层数低的节点，比如整个树的根节点，它的 $priority$ 属性也会更小（在小根堆中）。并且，在朴素的搜索树中，先被插入的节点，也更有可能会有比较小的层数。我们可以把这个 $priority$ 属性和被插入的顺序关联起来理解，这样，也就理解了为什么 treap 可以把节点插入的顺序通过 $priority$ 打乱。

给 treap 插入新节点时，需要同时维护树和堆的性质。其中，搜索树的性质可以在插入时维护，而堆性质的维护则有两种处理方法，分别是旋转和分裂、合并。使用这两种方法的 treap 被分别称为 **旋转 treap** 和 **无旋 treap**。

旋转 treap

旋转 treap 维护平衡的方式为旋转，和 AVL 树的旋转操作类似，分为 **左旋** 和 **右旋**。即在满足二叉搜索树的条件下根据堆的优先级对 treap 进行平衡操作。

旋转 treap 在做普通平衡树题的时候，是所有平衡树中常数较小的。

下面的讲解中的代码用指针实现了旋转 treap，文末附有数组形式的完整实现。

Info

代码中的 `rank` 代表前面讲的优先级（$priority$ 属性），该属性满足的是小根堆性质。

节点结构

```cpp
struct Node {
  Node *ch[2];  // 两个子节点的地址
  int val, rank;
  int rep_cnt;  // 当前这个值（val）重复出现的次数
  int siz;      // 以当前节点为根的子树大小

  Node(int val) : val(val), rep_cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    rank = rand();
    // 注意初始化的时候，rank 是随机给出的
  }

  void upd_siz() {
    // 用于旋转和删除过后，重新计算 siz 的值
    siz = rep_cnt;
```

```
16      if (ch[0] != nullptr) siz += ch[0]->siz;
17      if (ch[1] != nullptr) siz += ch[1]->siz;
18    }
19  };
```

旋转

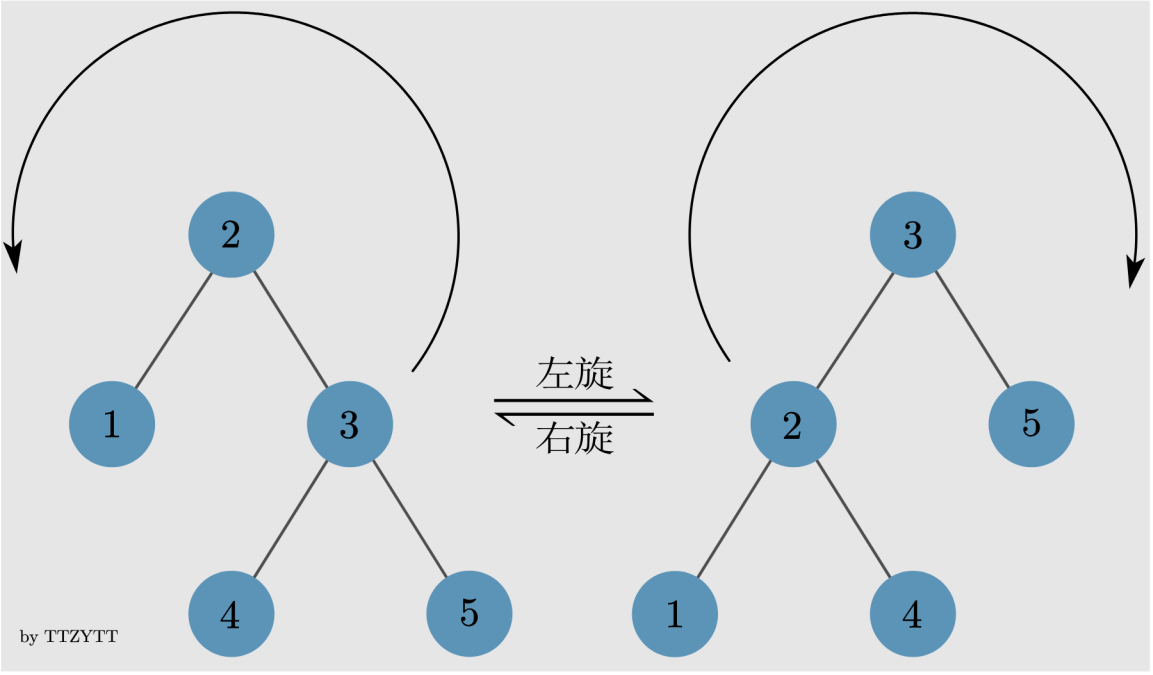
旋转操作是 treap 的一个非常重要的操作，主要用来在保持 treap 树性质的同时，调整不同节点的层数，以达到维护堆性质的作用。

旋转操作的左旋和右旋可能不是特别容易区分，以下是两个较为明显的特点：

旋转操作的含义：

- 在不影响搜索树性质的前提下，把和旋转方向相反的子树变成根节点（如左旋，就是把右子树变成根节点）
- 不影响性质，并且在旋转过后，跟旋转方向相同的子节点变成了原来的根节点（如左旋，旋转完之后的左子节点是旋转前的根节点）

左旋和右旋操作是相互的，如下图。




```
1  enum rot_type { LF = 1, RT = 0 };
2
3  void _rotate(Node *&cur,
4               rot_type dir) {  // dir参数代表旋转的方向 0为右旋，1为左旋
5    // 注意传进来的 cur 是指针的引用，也就是改了这个
6    // cur，变量是跟着一起改的，如果这个 cur 是别的 树的子节点，根据 ch
7    // 找过来的时候，也是会找到这里的
8
9    // 以下的代码解释的均是左旋时的情况
```

```
    Node *tmp = cur->ch[dir];  // 让 C 变成根节点，
                               // 这里的 tmp
                               // 是一个临时的节点指针，指向成为新的根节点的节点

    /* 左旋：也就是让右子节点变成根节点
     *         A                 C
     *        / \               / \
     *       B  C    ---->     A   E
     *         / \            / \
     *        D   E          B   D
     */
    cur->ch[dir] = tmp->ch[!dir];    // 让 A 的右子节点变成 D
    tmp->ch[!dir] = cur;             // 让 C 的左子节点变成 A
    cur->upd_siz(), tmp->upd_siz();  // 更新大小信息
    cur = tmp;  // 最后把临时储存 C 树的变量赋值到当前根节点上（注意 cur 是引用）
  }
```

插入

类似普通二叉搜索树的插入，但是需要在插入的过程中通过旋转来维护优先级的堆性质。

```
void _insert(Node *&cur, int val) {
  if (cur == nullptr) {
    // 没这个节点直接新建
    cur = new Node(val);
    return;
  } else if (val == cur->val) {
    // 如果有这个值相同的节点，就把重复数量加一
    cur->rep_cnt++;
    cur->siz++;
  } else if (val < cur->val) {
    // 维护搜索树性质，val 比当前节点小就插到左边，反之亦然
    _insert(cur->ch[0], val);
    if (cur->ch[0]->rank < cur->rank) {
      // 小根堆中，上面节点的优先级一定更小
      // 因为新插的左子节点比父节点小，现在需要让左子节点变成父节点
      _rotate(cur, RT);  // 注意前面的旋转性质，要把左子节点转上来，需要右旋
    }
    cur->upd_siz();  // 插入之后大小会变化，需要更新
  } else {
    _insert(cur->ch[1], val);
    if (cur->ch[1]->rank < cur->rank) {
      _rotate(cur, LF);
    }
    cur->upd_siz();
  }
}
```

删除

主要就是分类讨论，不同的情况有不同的处理方法，删完了树的大小会有变化，要注意更新。并且如果要删的节点有左子树和右子树，就要考虑删除之后让谁来当父节点（维护 rank 小的节点

在上面)。

```cpp
void _del(Node *&cur, int val) {
  if (val > cur->val) {
    _del(cur->ch[1], val);
    // 值更大就在右子树，反之亦然
    cur->upd_siz();
  } else if (val < cur->val) {
    _del(cur->ch[0], val);
    cur->upd_siz();
  } else {
    if (cur->rep_cnt > 1) {
      // 如果要删除的节点是重复的，可以直接把重复值减小
      cur->rep_cnt--, cur->siz--;
      return;
    }
    uint8_t state = 0;
    state |= (cur->ch[0] != nullptr);
    state |= ((cur->ch[1] != nullptr) << 1);
    // 00都无，01有左无右，10，无左有右，11都有
    Node *tmp = cur;
    switch (state) {
      case 0:
        delete cur;
        cur = nullptr;
        // 没有任何子节点，就直接把这个节点删了
        break;
      case 1:  // 有左无右
        cur = tmp->ch[0];
        // 把根变成左儿子，然后把原来的根节删了，注意这里的 tmp 是从 cur
        // 复制的，而 cur 是引用
        delete tmp;
        break;
      case 2:  // 有右无左
        cur = tmp->ch[1];
        delete tmp;
        break;
      case 3:
        rot_type dir = cur->ch[0]->rank < cur->ch[1]->rank
                           ? RT
                           : LF;  // dir 是 rank 更小的那个儿子
        _rotate(cur, dir);  // 这里的旋转可以把优先级更小的儿子转上去，rt 是 0，
                            // 而 lf 是 1，刚好跟实际的子树下标反过来
        _del(
            cur->ch[!dir],
            val);  // 旋转完成后原来的根节点就在旋方向那边，所以需要
                   // 继续把这个原来的根节点删掉
                   // 如果说要删的这个节点是在整个树的「上层的」，那我们会一直通过这
                   // 这里的旋转操作，把它转到没有子树了（或者只有一个），再删掉它。
        cur->upd_siz();
        // 删除会造成大小改变
        break;
    }
  }
}
```

根据值查询排名

操作含义：查询以 cur 为根节点的子树中，val 这个值的大小的排名（该子树中小于 val 的节点的个数 + 1）

```
int _query_rank(Node *cur, int val) {
  int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
  // 这个树中小于 val 的节点的数量
  if (val == cur->val)
    // 如果这个节点就是要查的节点
    return less_siz + 1;
  else if (val < cur->val) {
    if (cur->ch[0] != nullptr)
      return _query_rank(cur->ch[0], val);
    else
      return 1;  // 如果左子树是空的，说比最小的节点还要小，那这个数字就是最小的
  } else {
    if (cur->ch[1] != nullptr)
      // 如果要查的值比这个节点大，那这个节点的左子树以及这个节点自身肯定都比要查的值小
      // 所以要加上这两个值，再加上往右边找的结果
      // （以右子树为根的子树中，val 这个值的大小的排名）
      return less_siz + cur->rep_cnt + _query_rank(cur->ch[1], val);
    else
      return cur->siz + 1;
    // 没有右子树的话直接整个树 + 1 相当于 less_siz + cur->rep_cnt + 1
  }
}
```

根据排名查询值

要根据排名查询值，我们首先要知道如何判断要查的节点在树的哪个部分：

以下是一个判断方法的表：

左子树	根节点/当前节点	右子树
排名 ≤ 左子树的大小	排名 > 左子树的大小，并且 ≤ 左子树的大小 + 根节点的重复次数	排名 > 左子树的大小 + 根节点的重复次数

注意如果在右子树，递归的时候需要对原来的 `rank` 进行处理。递归的时候就相当去查，在右子树中为这个排名的值，为了把排名转换成基于右子树的，需要把原来的 `rank` 减去左子树的大小和根节点的重复次数。

可以把所有节点想象成一个排好序的数组，或者数轴（如下），

```
1 -> |左子树的节点|根节点|右子树的节点| -> n
                          ^
                          要查的排名
```

```
                    ⬇转换成基于右子树的排名
1 -> |右子树的节点| -> n
       ^
       要查的排名
```

这里的转换方法就是直接把排名减去左子树的大小和根节点的重复数量。

```cpp
int _query_val(Node *cur, int rank) {
  // 查询树中第 rank 大的节点的值
  int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
  // less siz 是左子树的大小
  if (rank <= less_siz)
    return _query_val(cur->ch[0], rank);
  else if (rank <= less_siz + cur->rep_cnt)
    return cur->val;
  else
    return _query_val(cur->ch[1], rank - less_siz - cur->rep_cnt);  // 见前文
}
```

查询第一个比 val 小的节点

注意这里使用了一个类中的全局变量，`q_prev_tmp`。

这个值是只有在 val 比当前节点值大的时候才会被更改的，所以返回这个变量就是返回 val 最后一次比当前节点的值大，之后就是更小了。

```cpp
int _query_prev(Node *cur, int val) {
  if (val <= cur->val) {
    // 还是比 val 大，所以往左子树找
    if (cur->ch[0] != nullptr) return _query_prev(cur->ch[0], val);
  } else {
    // 只有能进到这个 else 里，才会更新 q_prev_tmp 的值
    q_prev_tmp = cur->val;
    // 当前节点已经比 val，小了，但是不确定是否是最大的，所以要到右子树继续找
    if (cur->ch[1] != nullptr) _query_prev(cur->ch[1], val);
    // 接下来的递归可能不会更改 q_prev_tmp
    // 了，那就直接返回这个值，总之返回的就是最后一次进到 这个 else 中的
    // cur->val
    return q_prev_tmp;
  }
  return NIL;
}
```

查询第一个比 val 大的节点

跟前一个很相似，只是大于小于号换了一下。

```cpp
int _query_nex(Node *cur, int val) {
  if (val >= cur->val) {
    if (cur->ch[1] != nullptr) return _query_nex(cur->ch[1], val);
```

```
    } else {
      q_nex_tmp = cur->val;
      if (cur->ch[0] != nullptr) _query_nex(cur->ch[0], val);
      return q_nex_tmp;
    }
    return NIL;
  }
```

无旋 treap

无旋 treap 的操作方式使得它天生支持维护序列、可持久化等特性。

无旋 treap 又称分裂合并 treap。它仅有两种核心操作，即为 **分裂** 与 **合并**。通过这两种操作，在很多情况下可以比旋转 treap 更方便的实现别的操作。下面逐一介绍这两种操作。

> **注释**
>
> 讲解无旋 treap 应当提到 **FHQ-Treap**（by 范浩强）。即可持久化，支持区间操作的无旋 Treap。更多内容请参照《范浩强谈数据结构》ppt。

分裂（split）

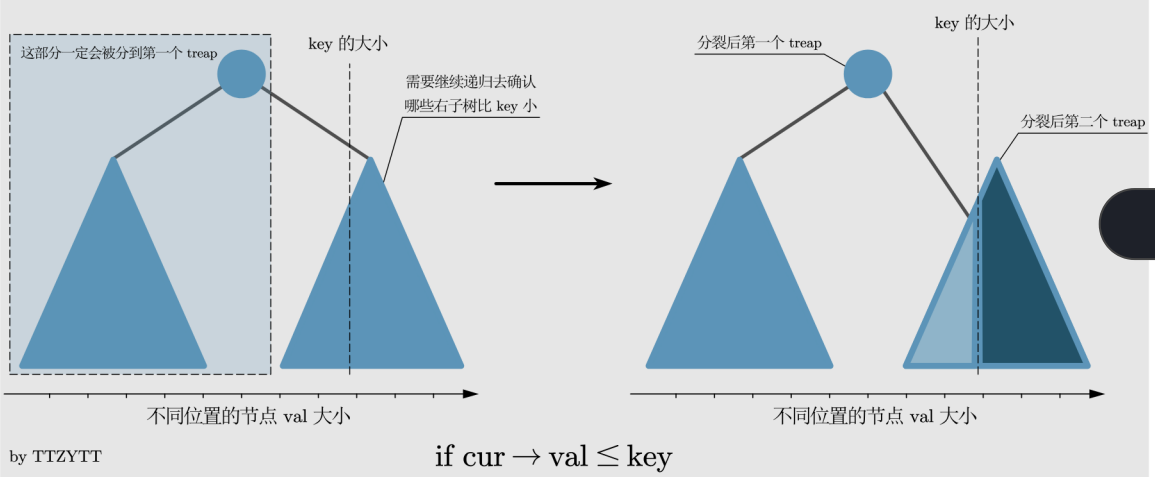
按值分裂

分裂过程接受两个参数：根指针 cur、关键值 key。结果为将根指针指向的 treap 分裂为两个 treap，第一个 treap 所有结点的值（val）小于等于 key，第二个 treap 所有结点的值大于 key。

该过程首先判断 key 是否小于 cur 的值，若小于，则说明 cur 及其右子树全部大于 key，属于第二个 treap。当然，也可能有一部分的左子树的值大于 key，所以还需要继续向左子树递归地分裂。对于大于 key 的那部分左子树，我们把它作为 cur 的左子树，这样，整个 cur 上的节点都是大于 key 的。

相应的，如果 key 大于等于 cur 的值，说明 cur 的整个左子树以及其自身都小于等于 key，属于分裂后的第一个 treap。并且，cur 的部分右子树也可能有部分小于等于 key，因此我们需要继续递归地分裂右子树。把小于等于 key 的那部分作为 cur 的右子树，这样，整个 cur 上的节点都小于等于 key。

下图展示了 cur 的值小于等于 key 时按值分裂的情况。[1]




```cpp
pair<Node *, Node *> split(Node *cur, int key) {
  if (cur == nullptr) return {nullptr, nullptr};
  if (cur->val <= key) {
    // cur 以及它的左子树一定属于分裂后的第一个树
    auto temp = split(cur->ch[1], key);
    // 但是它可能有部分右子树也比 key 小
    cur->ch[1] = temp.first;
    // 我们把小于 key 的那部分拿出来，作为 cur 的右子树，这样整个 cur 都是小于
    // key 的 剩下的那部分右子树成为分裂后的第二个 treap
    cur->upd_siz();
    // 分裂过后树的大小会变化，需要更新
    return {cur, temp.second};
  } else {
    // 同上
    auto temp = split(cur->ch[0], key);
    cur->ch[0] = temp.second;
    cur->upd_siz();
    return {temp.first, cur};
  }
}
```

按排名分裂

比起按值分裂，这个操作更像是旋转 treap 中的根据排名（某个节点的排名是树中所有小于此节点值的节点的数量 $+1$）查询值：

此函数接受两个参数，节点指针 cur 和排名 rk，返回分裂后的三个 treap。

其中，第一个 treap 中每个节点的排名都小于 rk，第二个的排名等于 rk，并且第二个 treap 只有一个节点（不可能有多个等于的，如果有的话会增加 `Node` 结构体中的 `cnt`），第三个则是大于。

此操作的重点在于判断排名和 cur 相等的节点在树的哪个部分，这也是旋转 treap 根据排名查询值操作时的重要部分，在前文有非常详细的解释，这里不过多讲解。

并且，此操作的递归部分和按值分裂也非常相似，这里不赘述。

```cpp
tuple<Node *, Node *, Node *> split_by_rk(Node *cur, int rk) {
  if (cur == nullptr) return {nullptr, nullptr, nullptr};
  int ls_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
  if (rk <= ls_siz) {
    // 排名和 cur 相等的节点在左子树
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur->ch[0], rk);
    cur->ch[0] = r;  // 返回的第三个 treap 中的排名都大于 rk
    // cur 的左子树被设成 r 后，整个 cur 中节点的排名都大于 rk
    cur->upd_siz();
    return {l, mid, cur};
  } else if (rk <= ls_siz + cur->cnt) {
    // 和 cur 相等的就是当前节点
    Node *lt = cur->ch[0];
    Node *rt = cur->ch[1];
    cur->ch[0] = cur->ch[1] = nullptr;
    // 分裂后第二个 treap 只有一个节点，所有要把它的子树设置为空
    return {lt, cur, rt};
  } else {
    // 排名和 cur 相等的节点在右子树
    // 递归过程同上
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur->ch[1], rk - ls_siz - cur->cnt);
    cur->ch[1] = l;
    cur->upd_siz();
    return {cur, mid, r};
  }
}
```

合并（merge）

合并过程接受两个参数：左 treap 的根指针 u、右 treap 的根指针 v。必须满足 u 中所有结点的值小于等于 v 中所有结点的值。一般来说，我们合并的两个 treap 都是原来从一个 treap 中分裂出去的，所以不难满足 u 中所有节点的值都小于 v

在旋转 treap 中，我们借助旋转操作来维护 $priority$ 符合堆的性质，同时旋转时还不能改变树的性质。在无旋 treap 中，我们用合并达到相同的效果。

因为两个 treap 已经有序，所以我们在合并的时候只需要考虑把哪个树「放在上面」，把哪个「放在下面」，也就是是需要判断将哪个一个树作为子树。显然，根据堆的性质，我们需要把 $priority$ 小的放在上面（这里采用小根堆）。

同时，我们还需要满足搜索树的性质，所以若 u 的根结点的 $priority$ 小于 v 的，那么 u 即为新根结点，并且 v 因为值比 u 更大，应与 u 的右子树合并；反之，则 v 作为新根结点，然后因为 u 的值比 v 小，与 v 的左子树合并。

```cpp
Node *merge(Node *u, Node *v) {
  // 传进来的两个树的内部已经符合搜索树的性质了
  // 并且 u 内所有节点的值 < v 内所有节点的值
  // 所以在合并的时候需要维护堆的性质
  // 这里用的是小根堆
```

```
    if (u == nullptr && v == nullptr) return nullptr;
    if (u != nullptr && v == nullptr) return u;
    if (v != nullptr && u == nullptr) return v;

    if (u->prio < v->prio) {
      // u 的 prio 比较小，u应该作为父节点
      u->ch[1] = merge(u->ch[1], v);
      // 因为 v 比 u 大，所以把 v 作为 u 的右子树
      u->upd_siz();
      return u;
    } else {
      // v 比较小，v应该作为父节点
      v->ch[0] = merge(u, v->ch[0]);
      // u 比 v 小，所以递归时的参数是这样的
      v->upd_siz();
      return v;
    }
  }
```

插入

在无旋 treap 中，插入，删除，根据值查询排名等基础操作既可以用普通二叉查找树的方法实现，也可以用分裂和合并来实现。通常来说，使用分裂和合并来实现更加简洁，但是速度会慢一点[2]。为了帮助更好的理解无旋 treap，下面的操作全部使用分裂和合并实现。

在实现插入操作时，我们利用了分裂操作的一些性质。也就是值小于等于 val 的节点会被分到第一个 treap。

所以，假设我们根据 val 分裂当前这个 treap。会有下面两棵树，并符合以下条件：

$$T_1 \leq val$$
$$T_2 > val$$

其中 T_1 表示分裂后所有被分到第一个 treap 的节点的集合，T_2 则是第二个。

如果我们再按照 $val-1$ 继续分裂 T_1，那么会产生下面两棵树，并符合以下条件：

$$T_{1\,\text{left}} \leq val-1$$
$$T_{1\,\text{right}} > val-1 \;\&\; T_{1\,\text{right}} \leq val$$

其中 $T_{1\,\text{left}}$ 表示 T_1 分裂后所有被分到第一个 treap 的节点的集合，$T_{1\,\text{right}}$ 则是第二个。并且上面的式子中，后半部分的 $\& \; T_{1\,\text{right}} \leq val$ 来自于 T_1 所符合的条件 $T_1 \leq val$。

不难发现，只要 val 和节点的值是一个整数（大多数使用场景下会使用整数）那么符合 $T_{1\,\text{right}}$ 条件的节点只有一个，也就是值等于 val 的节点。

在插入时，如果我们发现符合 $T_{1\,\text{right}}$ 的节点存在，那就可以直接增加重复次数，否则，就新开一个节点。

注意把树分裂好了还需要用合并操作把它「粘」回去，这样下次还能继续使用。并且，还需要注意合并操作的参数顺序是有要求的，第一个树的所有节点的值都需要小于第二个。

```cpp
void insert(int val) {
  auto temp = split(root, val);
  // 根据 val 的值把整个树分成两个
  // 注意 split 的实现，等于 val 的子树是在左子树的
  auto l_tr = split(temp.first, val - 1);
  // l_tr 的左子树 <= val - 1，如果有 = val 的节点，那一定在右子树
  Node *new_node;
  if (l_tr.second == nullptr) {
    // 没有这个节点就新开，否则直接增加重复次数。
    new_node = new Node(val);
  } else {
    l_tr.second->cnt++;
    l_tr.second->upd_siz();
  }
  Node *l_tr_combined =
      merge(l_tr.first, l_tr.second == nullptr ? new_node : l_tr.second);
  // 合并 T_1 left 和 T_1 right
  root = merge(l_tr_combined, temp.second);
  // 合并 T_1 和 T_2
}
```

删除

删除操作也使用和插入操作相似的方法，找到值和 val 相等的节点，并且删除它。

```cpp
void del(int val) {
  auto temp = split(root, val);
  auto l_tr = split(temp.first, val - 1);
  if (l_tr.second->cnt > 1) {
    // 如果这个节点的重复次数大于 1，减小即可
    l_tr.second->cnt--;
    l_tr.second->upd_siz();
    l_tr.first = merge(l_tr.first, l_tr.second);
  } else {
    if (temp.first == l_tr.second) {
      // 有可能整个 T_1 只有这个节点，所以也需要把这个点设成 null 来标注已经删除
      temp.first = nullptr;
    }
    delete l_tr.second;
    l_tr.second = nullptr;
  }
  root = merge(l_tr.first, temp.second);
}
```

根据值查询排名

排名是比这个值小的节点的数量 $+1$，所以我们根据 $val-1$ 分裂当前树，那么分裂后的第一个树就符合：

$$T_1 \leq val - 1$$

如果树的值和 val 为整数，那么 T_1 就包含了所有值小于 val 的节点。

```cpp
int qrank_by_val(Node* cur, int val) {
  auto temp = split(cur, val - 1);
  int ret = (temp.first == nullptr ? 0 : temp.first->siz) + 1;  // 根据定义
+ 1
  root = merge(temp.first, temp.second);  // 拆好了再粘回去
  return ret;
}
```

根据排名查询值

调用 `split_by_rk()` 函数后，会返回分裂好的三个 treap，其中第二个只包含一个节点，它的排名等于 rk，所以我们直接返回这个节点的 val。

```cpp
int qval_by_rank(Node *cur, int rk) {
  Node *l, *mid, *r;
  tie(l, mid, r) = split_by_rk(cur, rk);
  int ret = mid->val;
  root = merge(merge(l, mid), r);
  return ret;
}
```

查询第一个比 val 小的节点

可以把这个问题转化为，在比 val 小的所有节点中，找出排名最大的。我们根据 val 来分裂这个 treap，返回的第一个 treap 中的节点的值就全部小于 val，然后我们调用 `qval_by_rank()` 找出这个树中值最大的节点。

```cpp
int qprev(int val) {
  auto temp = split(root, val - 1);
  // temp.first 就是值小于 val 的子树
  int ret = qval_by_rank(temp.first, temp.first->siz);
  // 这里查询的是，所有小于 val 的节点里面，最大的那个的值
  root = merge(temp.first, temp.second);
  return ret;
}
```

查询第一个比 val 大的节点

和上个操作类似，可以把这个问题转化为，在比 val 大的所有节点中，找出排名最小的。那么根据 val 分裂后，返回的第二个 treap 中的所有节点的值就大于 val。

然后我们去查询这个树中排名为 1 的节点（也就是值最小的节点）的值，就可以成功查到第一个比 val 大的节点。

```cpp
int qnex(int val) {
  auto temp = split(root, val);
  int ret = qval_by_rank(temp.second, 1);
  // 查询所有大于 val 的子树里面，值最小的那个
```

```
    root = merge(temp.first, temp.second);
    return ret;
  }
```

建树（build）

将一个有 n 个节点的序列 $\{a_n\}$ 转化为一棵 treap。

可以依次暴力插入这 n 个节点，每次插入一个权值为 v 的节点时，将整棵 treap 按照权值分裂成权值小于等于 v 的和权值大于 v 的两部分，然后新建一个权值为 v 的节点，将两部分和新节点按从小到大的顺序依次合并，单次插入时间复杂度 $O(\log n)$，总时间复杂度 $O(n \log n)$。

在某些题目内，可能会有多次插入一段有序序列的操作，这是就需要在 $O(n)$ 的时间复杂度内完成建树操作。

方法一：在递归建树的过程中，每次选取当前区间的中点作为该区间的树根，并对每个节点钦定合适的优先值，使得新树满足堆的性质。这样能保证树高为 $O(\log n)$。

方法二：在递归建树的过程中，每次选取当前区间的中点作为该区间的树根，然后给每个节点一个随机优先级。这样能保证树高为 $O(\log n)$，但不保证其满足堆的性质。这样也是正确的，因为无旋式 treap 的优先级是用来使 `merge` 操作更加随机一点，而不是用来保证树高的。

方法三：观察到 treap 是笛卡尔树，利用笛卡尔树的 $O(n)$ 建树方法即可，用单调栈维护右链即可。

无旋 treap 的区间操作

建树

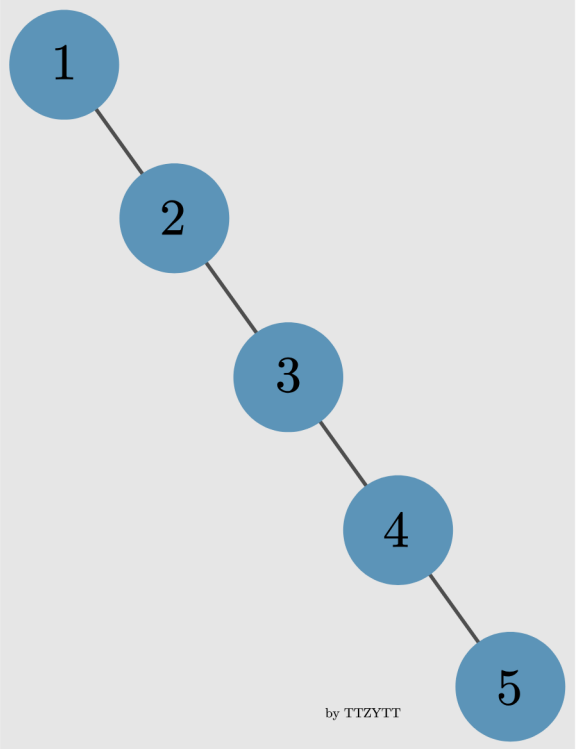
无旋 treap 相比旋转 treap 的一大好处就是可以实现各种区间操作，下面我们以文艺平衡树的 模板题 为例，介绍 treap 的区间操作。

> 您需要写一种数据结构（可参考题目标题），来维护一个有序数列。
>
> 其中需要提供以下操作：翻转一个区间，例如原有序序列是 $5\ 4\ 3\ 2\ 1$，翻转区间是 $[2,4]$ 的话，结果是 $5\ 2\ 3\ 4\ 1$。 对于 100% 的数据，$1 \leq n$（初始区间长度）m（翻转次数）$\leq 10^5$

在这道题目中，我们需要实现的是区间翻转，那么我们首先需要考虑如何建树，建出来的树需要是初始的区间。

我们只需要把区间的下标依次插入 treap 中，这样在中序遍历（先遍历左子树，然后当前节点，最后右子树）时，就可以得到这个区间[3]。

我们知道在朴素的二叉查找树中按照递增的顺序插入节点，建出来的树是一个长链，按照中序遍历，自然可以得到这个区间。



如上图，按照 $1\ 2\ 3\ 4\ 5$ 的顺序给朴素搜索树插入节点，中序遍历时，得到的也是 $1\ 2\ 3\ 4\ 5$。

但是在 treap 中，按增序插入节点后，在合并操作时还会根据 $priority$ 调整树的结构，在这样的情况下，如何确保中序遍历一定能正确的输出呢?

可以参考 笛卡尔树的单调栈建树方法 来理解这个问题。

设新插入的节点为 u。

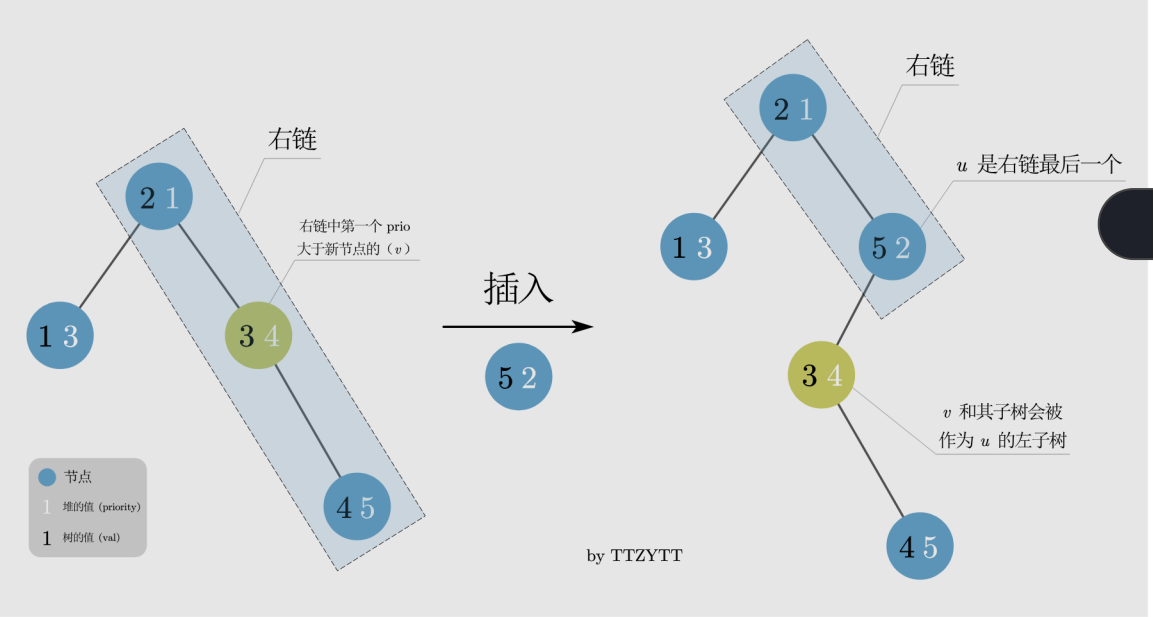
首先，因为是递增地插入节点，每一个新插入的节点肯定会被连接到 treap 的右链（即从根结点一直往右子树走，经过的结点形成的链）上。

从根节点开始，右链上的节点的 $priority$ 是递增的（小根堆）。那我们可以找到右链上第一个 $priority$ 大于 u 的节点，我们叫这个节点 v，并把这个节点换成 u。

因为 u 一定大于这个树上其他的全部节点，我们需要把 v 以及它的子树作为 u 的左子树。并且此时 u 没有右子树。

可以发现，中序遍历时 u 一定是最后一个被遍历到的（因为 u 是右链中的最后一个，而中序遍历中，右子树是最后被遍历到的）。

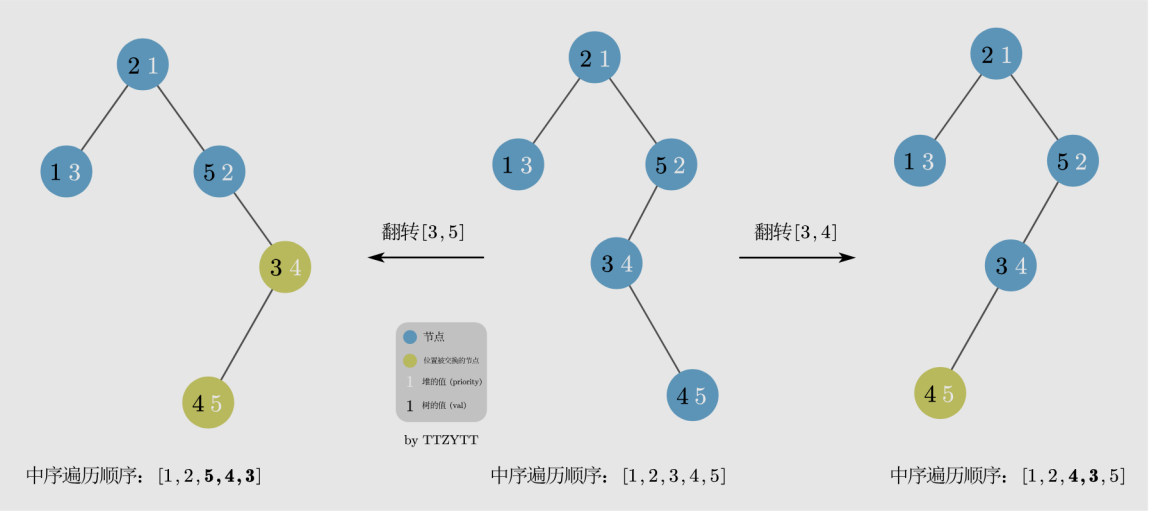
下图是一个 treap 根据递增顺序插入 $1 \sim 5$ 号节点时，插入 5 号节点时的变化，可以用这张图更好的理解按照增序插入的过程。



区间翻转

翻转 $[l,r]$ 这个区间时，基本思路是将树分裂成 $[1,l-1]$，$[l,r]$，$[r+1,n]$ 三个区间，再对中间的 $[l,r]$ 进行翻转[3]。

翻转的具体操作是把区间内的子树的每一个左，右子节点交换位置。如下图就展示了翻转上图中 treap 的 $[3,4]$ 和 $[3,5]$ 区间后的 treap。



注意如果按照这个方法翻转，那么每次翻转 $[l,r]$ 区间时，就会有 $r-l$ 个节点会被交换位置，这样频繁的操作显然不能满足 10^5 的数据范围，其 $O(n\times\log_2 n)$ 的单次翻转复杂度甚至不如暴力（因为我们除了需要花线性时间交换节点外，还需要在树中花费 $O(\log_2 n)$ 的时间找到需要交换的节点）。

再观察题目要求，可以发现因为只需要最后输出操作完的区间，所以并不需要每次都真的去交换。如此一来，便可以使用线段树中常用的懒标记（lazy tag）来优化复杂度。交换时，只需要在父节点打上标记，代表这个子树下的每个左右子节点都需要交换就行了。

在线段树中，我们一般在更新和查询时下传懒标记。这是因为，在更新和查询时，我们想要更新/查询的范围不一定和懒标记代表的范围重合，所以要先下传标记，确保查到和更新后的值是正确的。

在无旋 treap 中也是一样。具体操作时我们会把 treap 分裂成前文讲到的三个树，然后给中间的树打上懒标记后合并这三棵树。因为我们想要翻转的区间和懒标记代表的区间不一定重合，所要在分裂时下传标记。并且，分裂和合并操作会造成每个节点及其懒标记所代表的节点发生变动，所以也需要在合并前下传懒标记。

换句话说，是当树的结构发生改变的时候，当我们进行分裂或合并操作时需要改变某一个点的左右儿子信息时之前，应该下放标记，而非之后，因为懒标记是需要下传给儿子节点的，但更改左右儿子信息之后若懒标记还未下放，则懒标记就丢失了下放的对象。[4]

以下为代码讲解，代码参考了[3]。

因为区间操作中大部分操作都和普通的无旋 treap 相同，所以这里只讲解和普通无旋 treap 不同的地方。

下传标记

需要注意这里的懒标记代表需要把这个树中的每一个子节点交换位置。所以如果当前节点的子节点也有懒标记，那两次翻转就抵消了。如果子节点不需要翻转，那么这个懒标记就需要继续被下传到子节点上。

```
// 这里这个 pushdown 是 Node 类的成员函数，其中 to_rev 是懒标记
void pushdown() {
  swap(ch[0], ch[1]);
  if (ch[0] != nullptr) ch[0]->to_rev ^= 1;
  if (ch[1] != nullptr) ch[1]->to_rev ^= 1;
  to_rev = false;
}

void check_tag() {
  if (to_rev) pushdown();
}
```

分裂

注意在这个题目中，因为翻转操作，treap 中的 val 会不符合二叉搜索树的性质（见区间翻转部分的图），所以我们不能根据 val 来判断应该往左子树还是右子树递归。

所以这里的分裂跟普通无旋 treap 中的按排名分裂更相似，是根据当前树的大小判断往左还是右子树递归的，换言之，我们是按照开始时这个节点在树中的位置来判断的。

返回的第一个 treap 中节点的排名全部小于等于 sz，而第二个 treap 中节点的排名则全部大于 sz。

```
#define siz(_) (_ == nullptr ? 0 : _->siz)

```

```
pair<Node*, Node*> split(Node* cur, int sz) {
  // 按照树的大小判断
  if (cur == nullptr) return {nullptr, nullptr};
  cur->check_tag();
  // 分裂前先下传
  if (sz <= siz(cur->ch[0])) {
    auto temp = split(cur->ch[0], sz);
    cur->ch[0] = temp.second;
    cur->upd_siz();
    return {temp.first, cur};
  } else {
    auto temp =
        split(cur->ch[1],
              sz - siz(cur->ch[0]) -
                  1);  // 这里的转换在有旋 treap 的 「根据排名查询值有讲」
    cur->ch[1] = temp.first;
    cur->upd_siz();
    return {cur, temp.second};
  }
}
```

合并

唯一需要注意的是在合并前下传懒标记

```
Node *merge(Node *sm, Node *bg) {
  // small, big
  if (sm == nullptr && bg == nullptr) return nullptr;
  if (sm != nullptr && bg == nullptr) return sm;
  if (sm == nullptr && bg != nullptr) return bg;
  sm->check_tag(), bg->check_tag();
  if (sm->prio < bg->prio) {
    sm->ch[1] = merge(sm->ch[1], bg);
    sm->upd_siz();
    return sm;
  } else {
    bg->ch[0] = merge(sm, bg->ch[0]);
    bg->upd_siz();
    return bg;
  }
}
```

区间翻转

和前面介绍的一样，分裂出 $[1,l-1]$, $[l,r]$, $[r+1,n]$ 三个区间，然后对中间的区间打上标记后再合并。

```
void seg_rev(int l, int r) {
  // 这里的 less 和 more 是相对于 l 的
  auto less = split(root, l - 1);
  // 所有小于等于 l - 1 的会在 less 的左子树
  auto more = split(less.second, r - l + 1);
  // 从 l 开始的前 r - l + 1 个元素的区间
  more.first->to_rev = true;
```

```
    root = merge(less.first, merge(more.first, more.second));
  }
```

中序遍历打印

要注意在打印时要下传标记。

```
  void print(Node* cur) {
    if (cur == nullptr) return;
    cur->check_tag();
    // 中序遍历 -> 先左子树，再自己，最后右子树
    print(cur->ch[0]);
    cout << cur->val << " ";
    print(cur->ch[1]);
  }
```

完整代码

旋转 treap

指针实现

完整代码

以下是前文讲解的代码的完整版本，是普通平衡树的模板代码。

```cpp
// author: (ttzytt)[ttzytt.com]
#include <cstdint>
#include <cstdio>
#include <cstdlib>
using namespace std;

struct Node {
  Node *ch[2];
  int val, rank;
  int rep_cnt;
  int siz;

  Node(int val) : val(val), rep_cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    rank = rand();
  }

  void upd_siz() {
    siz = rep_cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
  }
};

class Treap {
 private:
  Node *root;

  constexpr static int NIL = -1;  // 用于表示查询的值不存在

  enum rot_type { LF = 1, RT = 0 };

  int q_prev_tmp = 0, q_nex_tmp = 0;

  void _rotate(Node *&cur, rot_type dir) {  // 0为右旋，1为左旋
    Node *tmp = cur->ch[dir];
    cur->ch[dir] = tmp->ch[!dir];
    tmp->ch[!dir] = cur;
    cur->upd_siz(), tmp->upd_siz();
    cur = tmp;
  }

  void _insert(Node *&cur, int val) {
    if (cur == nullptr) {
      cur = new Node(val);
      return;
```

```
48      } else if (val == cur->val) {
49        cur->rep_cnt++;
50        cur->siz++;
51      } else if (val < cur->val) {
52        _insert(cur->ch[0], val);
53        if (cur->ch[0]->rank < cur->rank) {
54          _rotate(cur, RT);
55        }
56        cur->upd_siz();
57      } else {
58        _insert(cur->ch[1], val);
59        if (cur->ch[1]->rank < cur->rank) {
60          _rotate(cur, LF);
61        }
62        cur->upd_siz();
63      }
64    }
65
66    void _del(Node *&cur, int val) {
67      if (val > cur->val) {
68        _del(cur->ch[1], val);
69        cur->upd_siz();
70      } else if (val < cur->val) {
71        _del(cur->ch[0], val);
72        cur->upd_siz();
73      } else {
74        if (cur->rep_cnt > 1) {
75          cur->rep_cnt--, cur->siz--;
76          return;
77        }
78        uint8_t state = 0;
79        state |= (cur->ch[0] != nullptr);
80        state |= ((cur->ch[1] != nullptr) << 1);
81        // 00都无，01有左无右，10，无左有右，11都有
82        Node *tmp = cur;
83        switch (state) {
84          case 0:
85            delete cur;
86            cur = nullptr;
87            break;
88          case 1:  // 有左无右
89            cur = tmp->ch[0];
90            delete tmp;
91            break;
92          case 2:  // 有右无左
93            cur = tmp->ch[1];
94            delete tmp;
95            break;
96          case 3:
97            rot_type dir = cur->ch[0]->rank < cur->ch[1]->rank
98  ? RT : LF;
99            _rotate(cur, dir);
```

```
100             _del(cur->ch[!dir], val);
101             cur->upd_siz();
102             break;
103           }
104       }
105     }
106
107     int _query_rank(Node *cur, int val) {
108       int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]-
109 >siz;
110       if (val == cur->val)
111         return less_siz + 1;
112       else if (val < cur->val) {
113         if (cur->ch[0] != nullptr)
114           return _query_rank(cur->ch[0], val);
115         else
116           return 1;
117       } else {
118         if (cur->ch[1] != nullptr)
119           return less_siz + cur->rep_cnt + _query_rank(cur-
120 >ch[1], val);
121         else
122           return cur->siz + 1;
123       }
124     }
125
126     int _query_val(Node *cur, int rank) {
127       int less_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]-
128 >siz;
129       if (rank <= less_siz)
130         return _query_val(cur->ch[0], rank);
131       else if (rank <= less_siz + cur->rep_cnt)
132         return cur->val;
133       else
134         return _query_val(cur->ch[1], rank - less_siz - cur-
135 >rep_cnt);
136     }
137
138     int _query_prev(Node *cur, int val) {
139       if (val <= cur->val) {
140         if (cur->ch[0] != nullptr) return _query_prev(cur-
141 >ch[0], val);
142       } else {
143         q_prev_tmp = cur->val;
144         if (cur->ch[1] != nullptr) _query_prev(cur->ch[1],
145 val);
146         return q_prev_tmp;
147       }
148       return NIL;
149     }
150
151     int _query_nex(Node *cur, int val) {
```

```
    if (val >= cur->val) {
      if (cur->ch[1] != nullptr) return _query_nex(cur->ch[1], val);
    } else {
      q_nex_tmp = cur->val;
      if (cur->ch[0] != nullptr) _query_nex(cur->ch[0], val);
      return q_nex_tmp;
    }
    return NIL;
  }

 public:
  void insert(int val) { _insert(root, val); }

  void del(int val) { _del(root, val); }

  int query_rank(int val) { return _query_rank(root, val); }

  int query_val(int rank) { return _query_val(root, rank); }

  int query_prev(int val) { return _query_prev(root, val); }

  int query_nex(int val) { return _query_nex(root, val); }
};

Treap tr;

int main() {
  srand(0);
  int t;
  scanf("%d", &t);
  while (t--) {
    int mode;
    int num;
    scanf("%d%d", &mode, &num);
    switch (mode) {
      case 1:
        tr.insert(num);
        break;
      case 2:
        tr.del(num);
        break;
      case 3:
        printf("%d\n", tr.query_rank(num));
        break;
      case 4:
        printf("%d\n", tr.query_val(num));
        break;
      case 5:
        printf("%d\n", tr.query_prev(num));
        break;
      case 6:
```

```
                printf("%d\n", tr.query_nex(num));
                break;
            }
        }
    }
```

数组实现

以下是 bzoj 普通平衡树模板代码，使用数组实现。

完整代码

```cpp
#include <iostream>

constexpr int MAXN = 100005;
constexpr int INF = 1 << 30;

int n;

struct treap {  // 直接维护成数据结构，可以直接用
  int l[MAXN], r[MAXN], val[MAXN], rnd[MAXN], size_[MAXN],
w[MAXN];
  int sz, ans, rt;

  void pushup(int x) { size_[x] = size_[l[x]] + size_[r[x]] +
w[x]; }

  void lrotate(int &k) {
    int t = r[k];
    r[k] = l[t];
    l[t] = k;
    size_[t] = size_[k];
    pushup(k);
    k = t;
  }

  void rrotate(int &k) {
    int t = l[k];
    l[k] = r[t];
    r[t] = k;
    size_[t] = size_[k];
    pushup(k);
    k = t;
  }

  void insert(int &k, int x) {  // 插入
    if (!k) {
      sz++;
      k = sz;
      size_[k] = 1;
      w[k] = 1;
      val[k] = x;
      rnd[k] = rand();
      return;
    }
    size_[k]++;
    if (val[k] == x) {
      w[k]++;
    } else if (val[k] < x) {
      insert(r[k], x);
      if (rnd[r[k]] < rnd[k]) lrotate(k);
```

```
50      } else {
51        insert(l[k], x);
52        if (rnd[l[k]] < rnd[k]) rrotate(k);
53      }
54    }
55
56    bool del(int &k, int x) {  // 删除节点
57      if (!k) return false;
58      if (val[k] == x) {
59        if (w[k] > 1) {
60          w[k]--;
61          size_[k]--;
62          return true;
63        }
64        if (l[k] == 0 || r[k] == 0) {
65          k = l[k] + r[k];
66          return true;
67        } else if (rnd[l[k]] < rnd[r[k]]) {
68          rrotate(k);
69          return del(k, x);
70        } else {
71          lrotate(k);
72          return del(k, x);
73        }
74      } else if (val[k] < x) {
75        bool succ = del(r[k], x);
76        if (succ) size_[k]--;
77        return succ;
78      } else {
79        bool succ = del(l[k], x);
80        if (succ) size_[k]--;
81        return succ;
82      }
83    }
84
85    int queryrank(int k, int x) {
86      if (!k) return 0;
87      if (val[k] == x)
88        return size_[l[k]] + 1;
89      else if (x > val[k]) {
90        return size_[l[k]] + w[k] + queryrank(r[k], x);
91      } else
92        return queryrank(l[k], x);
93    }
94
95    int querynum(int k, int x) {
96      if (!k) return 0;
97      if (x <= size_[l[k]])
98        return querynum(l[k], x);
99      else if (x > size_[l[k]] + w[k])
100       return querynum(r[k], x - size_[l[k]] - w[k]);
101     else
```

```
102         return val[k];
103       }
104
105       void querypre(int k, int x) {
106         if (!k) return;
107         if (val[k] < x)
108           ans = k, querypre(r[k], x);
109         else
110           querypre(l[k], x);
111       }
112
113       void querysub(int k, int x) {
114         if (!k) return;
115         if (val[k] > x)
116           ans = k, querysub(l[k], x);
117         else
118           querysub(r[k], x);
119       }
120     } T;
121
122     using std::cin;
123     using std::cout;
124
125     int main() {
126       cin.tie(nullptr)->sync_with_stdio(false);
127       srand(123);
128       cin >> n;
129       int opt, x;
130       for (int i = 1; i <= n; i++) {
131         cin >> opt >> x;
132         if (opt == 1)
133           T.insert(T.rt, x);
134         else if (opt == 2)
135           T.del(T.rt, x);
136         else if (opt == 3) {
137           cout << T.queryrank(T.rt, x) << '\n';
138         } else if (opt == 4) {
139           cout << T.querynum(T.rt, x) << '\n';
140         } else if (opt == 5) {
141           T.ans = 0;
142           T.querypre(T.rt, x);
143           cout << T.val[T.ans] << '\n';
144         } else if (opt == 6) {
145           T.ans = 0;
146           T.querysub(T.rt, x);
147           cout << T.val[T.ans] << '\n';
148         }
149       }
       return 0;
     }
```

无旋 treap

指针实现

完整代码

以下是前文讲解的代码的完整版本，是普通平衡树的模板代码。

```cpp
// author: (ttzytt)[ttzytt.com]
#include <cstdio>
#include <cstdlib>
#include <ctime>
#include <tuple>
using namespace std;

struct Node {
  Node *ch[2];
  int val, prio;
  int cnt;
  int siz;

  Node(int _val) : val(_val), cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    prio = rand();
  }

  Node(Node *_node) {
    val = _node->val, prio = _node->prio, cnt = _node->cnt,
siz = _node->siz;
  }

  void upd_siz() {
    siz = cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
  }
};

struct none_rot_treap {
#define _3 second.second
#define _2 second.first
  Node *root;

  pair<Node *, Node *> split(Node *cur, int key) {
    if (cur == nullptr) return {nullptr, nullptr};
    if (cur->val <= key) {
      auto temp = split(cur->ch[1], key);
      cur->ch[1] = temp.first;
      cur->upd_siz();
      return {cur, temp.second};
    } else {
      auto temp = split(cur->ch[0], key);
      cur->ch[0] = temp.second;
      cur->upd_siz();
      return {temp.first, cur};
```

```
    }
  }

  tuple<Node *, Node *, Node *> split_by_rk(Node *cur, int
rk) {
    if (cur == nullptr) return {nullptr, nullptr, nullptr};
    int ls_siz = cur->ch[0] == nullptr ? 0 : cur->ch[0]->siz;
    if (rk <= ls_siz) {
      Node *l, *mid, *r;
      tie(l, mid, r) = split_by_rk(cur->ch[0], rk);
      cur->ch[0] = r;
      cur->upd_siz();
      return {l, mid, cur};
    } else if (rk <= ls_siz + cur->cnt) {
      Node *lt = cur->ch[0];
      Node *rt = cur->ch[1];
      cur->ch[0] = cur->ch[1] = nullptr;
      return {lt, cur, rt};
    } else {
      Node *l, *mid, *r;
      tie(l, mid, r) = split_by_rk(cur->ch[1], rk - ls_siz -
cur->cnt);
      cur->ch[1] = l;
      cur->upd_siz();
      return {cur, mid, r};
    }
  }

  Node *merge(Node *u, Node *v) {
    if (u == nullptr && v == nullptr) return nullptr;
    if (u != nullptr && v == nullptr) return u;
    if (v != nullptr && u == nullptr) return v;
    if (u->prio < v->prio) {
      u->ch[1] = merge(u->ch[1], v);
      u->upd_siz();
      return u;
    } else {
      v->ch[0] = merge(u, v->ch[0]);
      v->upd_siz();
      return v;
    }
  }

  void insert(int val) {
    auto temp = split(root, val);
    auto l_tr = split(temp.first, val - 1);
    Node *new_node;
    if (l_tr.second == nullptr) {
      new_node = new Node(val);
    } else {
      l_tr.second->cnt++;
      l_tr.second->upd_siz();
```

```
    }
    Node *l_tr_combined =
        merge(l_tr.first, l_tr.second == nullptr ? new_node :
l_tr.second);
    root = merge(l_tr_combined, temp.second);
  }

  void del(int val) {
    auto temp = split(root, val);
    auto l_tr = split(temp.first, val - 1);
    if (l_tr.second->cnt > 1) {
      l_tr.second->cnt--;
      l_tr.second->upd_siz();
      l_tr.first = merge(l_tr.first, l_tr.second);
    } else {
      if (temp.first == l_tr.second) {
        temp.first = nullptr;
      }
      delete l_tr.second;
      l_tr.second = nullptr;
    }
    root = merge(l_tr.first, temp.second);
  }

  int qrank_by_val(Node *cur, int val) {
    auto temp = split(cur, val - 1);
    int ret = (temp.first == nullptr ? 0 : temp.first->siz) +
1;
    root = merge(temp.first, temp.second);
    return ret;
  }

  int qval_by_rank(Node *cur, int rk) {
    Node *l, *mid, *r;
    tie(l, mid, r) = split_by_rk(cur, rk);
    int ret = mid->val;
    root = merge(merge(l, mid), r);
    return ret;
  }

  int qprev(int val) {
    auto temp = split(root, val - 1);
    int ret = qval_by_rank(temp.first, temp.first->siz);
    root = merge(temp.first, temp.second);
    return ret;
  }

  int qnex(int val) {
    auto temp = split(root, val);
    int ret = qval_by_rank(temp.second, 1);
    root = merge(temp.first, temp.second);
    return ret;
```

```
    }
};

none_rot_treap tr;

int main() {
    srand(time(nullptr));
    int t;
    scanf("%d", &t);
    while (t--) {
        int mode;
        int num;
        scanf("%d%d", &mode, &num);
        switch (mode) {
            case 1:
                tr.insert(num);
                break;
            case 2:
                tr.del(num);
                break;
            case 3:
                printf("%d\n", tr.qrank_by_val(tr.root, num));
                break;
            case 4:
                printf("%d\n", tr.qval_by_rank(tr.root, num));
                break;
            case 5:
                printf("%d\n", tr.qprev(num));
                break;
            case 6:
                printf("%d\n", tr.qnex(num));
                break;
        }
    }
}
```

无旋 treap 的区间操作

指针实现

完整代码

以下是前文讲解的代码的完整版本，是文艺平衡树题目的模板代码。

```cpp
// author: (ttzytt)[ttzytt.com]
#include <cstdlib>
#include <ctime>
#include <iostream>
using namespace std;

// 参考：https://www.cnblogs.com/Equinox-Flower/p/10785292.html
struct Node {
  Node* ch[2];
  int val, prio;
  int cnt;
  int siz;
  bool to_rev = false;  // 需要把这个子树下的每一个节点都翻转过来

  Node(int _val) : val(_val), cnt(1), siz(1) {
    ch[0] = ch[1] = nullptr;
    prio = rand();
  }

  int upd_siz() {
    siz = cnt;
    if (ch[0] != nullptr) siz += ch[0]->siz;
    if (ch[1] != nullptr) siz += ch[1]->siz;
    return siz;
  }

  void pushdown() {
    swap(ch[0], ch[1]);
    if (ch[0] != nullptr) ch[0]->to_rev ^= 1;
    // 如果原来子节点也要翻转，那两次翻转就抵消了，如果子节点不翻转，那这个
    //  tag 就需要继续被 push 到子节点上
    if (ch[1] != nullptr) ch[1]->to_rev ^= 1;
    to_rev = false;
  }

  void check_tag() {
    if (to_rev) pushdown();
  }
};

struct Seg_treap {
  Node* root;
#define siz(_) (_ == nullptr ? 0 : _->siz)

```

```cpp
  pair<Node*, Node*> split(Node* cur, int sz) {
    // 按照树的大小划分
    if (cur == nullptr) return {nullptr, nullptr};
    cur->check_tag();
    if (sz <= siz(cur->ch[0])) {
      // 左边的子树就够了
      auto temp = split(cur->ch[0], sz);
      // 左边的子树不一定全部需要，temp.second 是不需要的
      cur->ch[0] = temp.second;
      cur->upd_siz();
      return {temp.first, cur};
    } else {
      // 左边的加上右边的一部分（当然也包括这个节点本身）
      auto temp = split(cur->ch[1], sz - siz(cur->ch[0]) -
1);
      cur->ch[1] = temp.first;
      cur->upd_siz();
      return {cur, temp.second};
    }
  }

  Node* merge(Node* sm, Node* bg) {
    // small, big
    if (sm == nullptr && bg == nullptr) return nullptr;
    if (sm != nullptr && bg == nullptr) return sm;
    if (sm == nullptr && bg != nullptr) return bg;
    sm->check_tag(), bg->check_tag();
    if (sm->prio < bg->prio) {
      sm->ch[1] = merge(sm->ch[1], bg);
      sm->upd_siz();
      return sm;
    } else {
      bg->ch[0] = merge(sm, bg->ch[0]);
      bg->upd_siz();
      return bg;
    }
  }

  void insert(int val) {
    auto temp = split(root, val);
    auto l_tr = split(temp.first, val - 1);
    Node* new_node;
    if (l_tr.second == nullptr) new_node = new Node(val);
    Node* l_tr_combined =
        merge(l_tr.first, l_tr.second == nullptr ? new_node :
l_tr.second);
    root = merge(l_tr_combined, temp.second);
  }

  void seg_rev(int l, int r) {
    // 这里的 less 和 more 是相对于 l 的
    auto less = split(root, l - 1);
```

```
      // 所有小于等于 l - 1 的会在 less 的左边
      auto more = split(less.second, r - l + 1);
      // 拿出从 l 开始的前 r - l + 1 个
      more.first->to_rev = true;
      root = merge(less.first, merge(more.first, more.second));
    }

    void print(Node* cur) {
      if (cur == nullptr) return;
      cur->check_tag();
      print(cur->ch[0]);
      cout << cur->val << " ";
      print(cur->ch[1]);
    }
  };

  Seg_treap tr;

  int main() {
    srand(time(nullptr));
    int n, m;
    cin >> n >> m;
    for (int i = 1; i <= n; i++) tr.insert(i);
    while (m--) {
      int l, r;
      cin >> l >> r;
      tr.seg_rev(l, r);
    }
    tr.print(tr.root);
  }
```

例题

普通平衡树

文艺平衡树（Splay）

「ZJOI2006」书架

「NOI2005」维护数列

CF 702F T-Shirts

参考资料与注释

1. 本图的设计参考了 维基百科 treap 词条的配图 ↩

2. https://charleswu.site/archives/1051 ↩

3. https://www.cnblogs.com/Equinox-Flower/p/10785292.html ↩ ↩ ↩

4. https://www.luogu.com.cn/blog/85514/fhq-treap-xue-xi-bi-ji ↩
