Splay 树

本页面将简要介绍如何用 Splay 维护二叉查找树。

定义

Splay 树，或 **伸展树**，是一种平衡二叉查找树，它通过 **伸展（splay）操作** 不断将某个节点旋转到根节点，使得整棵树仍然满足二叉查找树的性质，能够在均摊 $O(\log N)$ 时间内完成插入、查找和删除操作，并且保持平衡而不至于退化为链。

Splay 树由 Daniel Sleator 和 Robert Tarjan 于 1985 年发明。

基本结构与操作

本节讨论 Splay 树的基本结构和它的核心操作，其中最为重要的是伸展操作。

Splay 树是一棵二叉查找树，查找某个值时满足性质：左子树任意节点的值 $<$ 根节点的值 $<$ 右子树任意节点的值。

维护信息

本文使用数组模拟指针来实现 Splay 树，需要维护如下信息：

rt	id	fa[i]	ch[i][0/1]	val[i]	cnt[i]	sz[i]
根节点编号	已使用节点个数	父亲	左右儿子编号	节点权值	权值出现次数	子树大小

初始化时，所有信息都置零即可。

辅助操作

首先是一些简单的辅助操作：

- `dir(x)`：判断节点 x 是父亲节点的左儿子还是右儿子；
- `push_up(x)`：在改变节点位置后，根据子节点信息更新节点 x 的信息。

实现

```
bool dir(int x) { return x == ch[fa[x]][1]; }

void push_up(int x) { sz[x] = cnt[x] + sz[ch[x][0]] + sz[ch[x][1]]; }
```

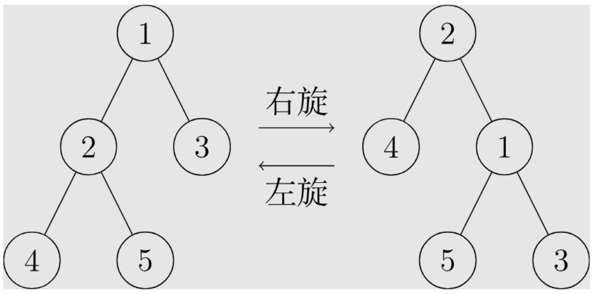
旋转操作

为了使 Splay 保持平衡，需要进行旋转操作。旋转的作用是将某个节点上移一个位置。

旋转需要保证：

- 整棵 Splay 的中序遍历不变（不能破坏二叉查找树的性质）；
- 受影响的节点维护的信息依然正确有效；
- `rt` 必须指向旋转后的根节点。

在 Splay 中旋转分为两种：左旋和右旋。



观察图示可知，如果要通过旋转将节点 x（左旋时的 1 和右旋时的 2）上移，则旋转的方向由该节点是其父节点的左节点还是右节点唯一确定。因此，实现旋转操作时，只需要将要上移的节点 x 传入即可。

具体分析旋转步骤：（假设需要上移的节点为 x，以右旋为例）

1. 首先，记录节点 x 的父节点 y，以及 y 的父节点 z（可能为空），并记录 x 是 y 的左子节点还是右子节点；
2. 按照旋转后的树中自下向上的顺序，依次更新 y 的左子节点为 x 的右子节点，x 的右子节点为 y，以及若 z 非空，z 的子节点为 x；
3. 按照同样的顺序，依次更新当前 y 的左子节点（若存在）的父节点为 y，y 的父节点为 x，以及 x 的父节点为 z；
4. 自下而上维护节点信息。

实现

```
void rotate(int x) {
  int y = fa[x], z = fa[y];
  bool r = dir(x);
  ch[y][r] = ch[x][!r];
  ch[x][!r] = y;
  if (z) ch[z][dir(y)] = x;
  if (ch[y][r]) fa[ch[y][r]] = y;
  fa[y] = x;
  fa[x] = z;
  push_up(y);
  push_up(x);
}
```

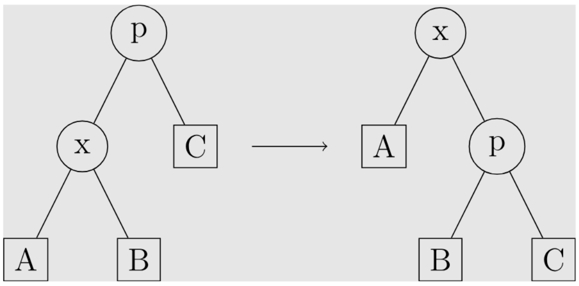
在所有函数的实现时，都应注意不要修改节点 0 的信息。

伸展操作

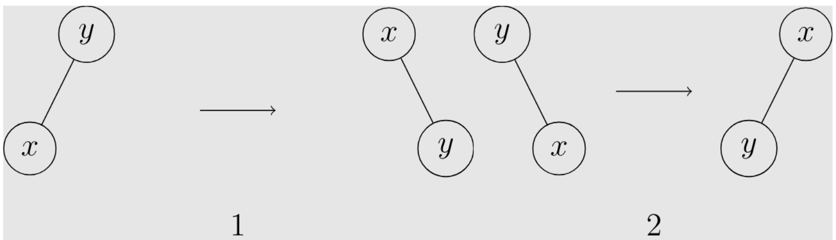
Splay 树要求每访问一个节点 x 后都要强制将其旋转到根节点。该操作也称为伸展操作。

设刚访问的节点为 x。要做伸展操作，就是要对 x 做一系列的 **伸展步骤**。每次对 x 做一次伸展步骤，x 到根节点的距离都会更近。定义 p 为 x 的父节点。伸展步骤有三种：

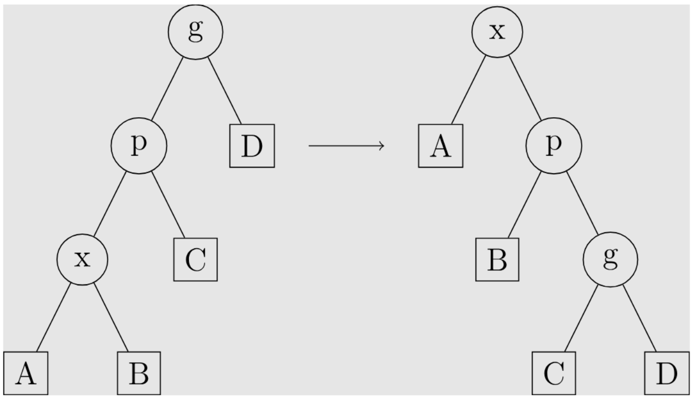
1. **zig**: 在 p 是根节点时操作。Splay 树会根据 x 和 p 间的边旋转。**zig** 存在是用于处理奇偶校验问题，仅当 x 在伸展操作开始时具有奇数深度时作为伸展操作的最后一步执行。

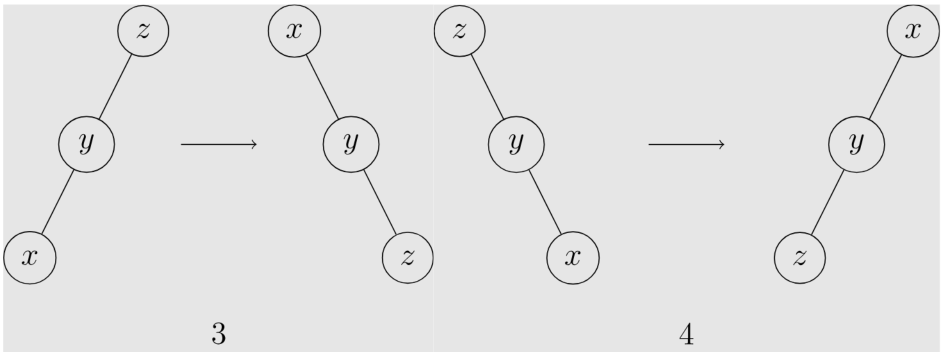
   即直接将 x 右旋或左旋（图 1, 2）。

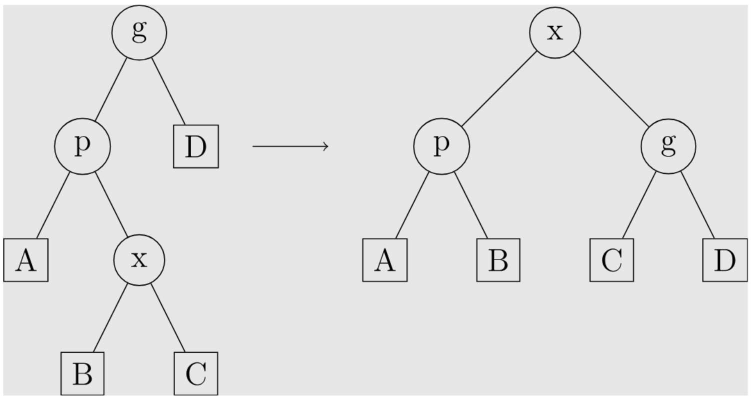
2. **zig-zig**: 在 p 不是根节点且 x 和 p 都是右侧子节点或都是左侧子节点时操作。下方例图显示了 x 和 p 都是左侧子节点时的情况。Splay 树首先按照连接 p 与其父节点 g 边旋转，然后按照连接 x 和 p 的边旋转。



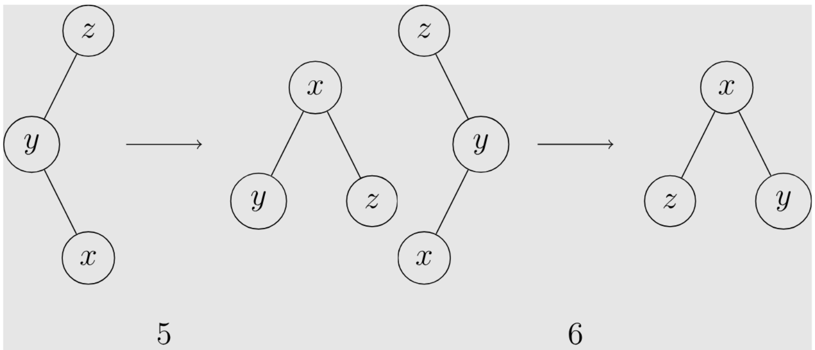
即首先将 p 右旋或左旋，然后将 x 右旋或左旋（图 3, 4）。



3. **zig-zag**: 在 p 不是根节点且 x 和 p 一个是右侧子节点一个是左侧子节点时操作。Splay 树首先按 p 和 x 之间的边旋转，然后按 x 和 g 新生成的结果边旋转。



即将 x 先左旋再右旋或先右旋再左旋（图 5, 6）。



> **Tip**
>
> 请读者尝试自行模拟 6 种旋转情况，以理解伸展操作的基本思想。

比较三种伸展步骤可知，要区分此时应使用哪种操作，关键是要判断 x 是否是根节点的子节点，以及 x 和它父节点是否在各自的父节点同侧。

此处提供的实现，可以指定任意根节点 z，并将它的子树内任意节点 x 上移至 z 处：

1. 首先记录根节点 z 的父节点 w，从而可以利用 `fa[x] == w` 判断 x 已经位于根结点处；
2. 记录 x 当前的父节点 y，如果 y 和 w 相同，说明 x 已经到达根节点；
3. 否则，利用 `fa[y] == w` 判断 y 是否是根节点。如果是，直接做 zig 操作将 x 旋转；如果不是，利用 `dir(x) == dir(y)` 判断使用 zig-zig 还是 zig-zag，前者先旋转 y 再旋转 x，后者直接旋转两次 x。

> **实现**

```cpp
void splay(int& z, int x) {
  int w = fa[z];
  for (int y; (y = fa[x]) != w; rotate(x)) {
    if (fa[y] != w) rotate(dir(x) == dir(y) ? y : x);
  }
  z = x;
}
```

伸展操作是 Splay 树的核心操作，也是它的时间复杂度能够得到保证的关键步骤。请务必保证每次向下访问节点后，都进行一次伸展操作。

另外，伸展操作会将当前节点 x 到根节点 z 的路径上的所有节点信息自下而上地更新一遍。正是因为这一点，才可以修改非根节点，再通过伸展操作将它上移至根来完成整个树的信息更新。

时间复杂度

对大小为 n 的 Splay 树做 m 次伸展操作的复杂度是 $O((n+m)\log n)$ 的，单次均摊复杂度是 $O(\log n)$ 的。

基于势能分析的复杂度证明

为此只需分析 **zig**、**zig-zig** 和 **zig-zag** 三种操作的复杂度。为此，我们采用 **势能分析法**，通过研究势能的变化来推导操作的均摊复杂度。假设对一棵包含 n 个节点的 Splay 树进行了 m 次伸展操作，可以通过如下方式进行分析：

定义：

1. **单个节点的势能**：$w(x) = \log(\text{size}(x))$，其中 $\text{size}(x)$ 表示以节点 x 为根的子树大小。
2. **整棵树的势能**：$\varphi = \Sigma\ w(x)$，即树中所有节点势能的总和，初始势能满足 $\varphi_0 \le n \log n$。
3. **第 i 次操作的均摊成本**：$c_i = t_i + \varphi_i - \varphi_{i-1}$，其中 t_i 为实际操作代价，φ_i 和 φ_{i-1} 分别为操作后和操作前的势能。

性质：

1. 如果 p 是 x 的父节点，则有 $w(p) \ge w(x)$，即父节点的势能不小于子节点的势能。
2. 由于根节点的子树大小在操作前后保持不变，因此根节点的势能在操作过程中不变。
3. 如果 $\text{size}(p) \ge \text{size}(x) + \text{size}(y)$，那么有 $2w(p) - w(x) - w(y) \ge 2$。

性质 3 的证明

根据均值不等式可知

$$\begin{aligned}2w(p) - w(x) - w(y) &= \log \frac{\text{size}(p)^2}{\text{size}(x)\cdot\text{size}(y)}\\ &> \log \frac{(\text{size}(x)+\text{size}(y))^2}{\text{size}(x)\cdot\text{size}(y)}\\ &\ge \log 4\\ &= 2.\end{aligned}$$

接下来，分别对 **zig**、**zig-zig** 和 **zig-zag** 操作进行势能分析。设操作前后的节点 x 的势能分别是 $w(x)$ 和 $w'(x)$。节点的记号与 上文 一致。

zig：根据性质 1 和 2，有 $w(p) = w'(x)$，且 $w'(x) \ge w'(p)$。由此，均摊成本为

$$\begin{aligned}c_i &= 1 + w'(x) + w'(p) - w(x) - w(p)\\ &= 1 + w'(p) - w(x)\\ &\le 1 + w'(x) - w(x).\end{aligned}$$

zig-zig：根据性质 1 和 2，有 $w(g) = w'(x)$，且 $w'(x) \ge w'(p)$，$w(x) \le w(p)$。因为

$$\begin{aligned}\text{size}'(x) &= 3 + \text{size}(A) + \text{size}(B) + \text{size}(C) + \text{size}(D)\\ &> (1 + \text{size}(A) + \text{size}(B)) + (1 + \text{size}(C) + \text{size}(D))\\ &= \text{size}(x) + \text{size}'(g),\end{aligned}$$

根据性质 3 可得

$$2w'(x) - w(x) - w'(g) \ge 2.$$

由此，均摊成本为

$$\begin{aligned}
c_i &= 2 + w'(x) + w'(p) + w'(g) - w(x) - w(p) - w(g) \\
&= 2 + w'(p) + w'(g) - w(x) - w(p) \\
&\le (2w'(x) - w(x) - w'(g)) + w'(p) + w'(g) - w(x) - w(p) \\
&= 2(w'(x) - w(x)) + w'(p) - w(p) \\
&\le 3(w'(x) - w(x)).
\end{aligned}$$

zig-zag：根据性质 1 和 2，有 $w(g) = w'(x)$，且 $w(p) \ge w(x)$。因为 $\text{size}'(x) > \text{size}'(p) + \text{size}'(g)$，根据性质 3，可得

$$2 \cdot w'(x) - w'(g) - w'(p) \ge 2.$$

由此，均摊成本为

$$\begin{aligned}
c_i &= 2 + w'(x) + w'(p) + w'(g) - w(x) - w(p) - w(g) \\
&= 2 + w'(p) + w'(g) - w(x) - w(p) \\
&\le (2w'(x) - w'(g) - w'(p)) + w'(p) + w'(g) - w(x) - w(p) \\
&= 2w'(x) - w(x) - w(p) \\
&\le 2(w'(x) - w(x)).
\end{aligned}$$

单次伸展操作：

令 $w^{(n)}(x) = (w^{(n-1)})'(x)$ 且 $w^{(0)}(x) = w(x)$。假设一次伸展操作依次访问了 $x_1, x_2, \cdots, x_n$ 等节点，最终 x_1 成为根节点。这必然经过若干次 **zig-zig** 和 **zig-zag** 操作和至多一次 **zig** 操作，前两种操作的均摊成本均不超过 $3(w'(x) - w(x))$，而最后一次操作的均摊成本不超过 $3(w'(x) - w(x)) + 1$，所以总的均摊成本不超过

$$3(w^{(n)}(x_1) - w^{(0)}(x_1)) + 1 \le 3\log n + 1.$$

因此，一次伸展操作的均摊复杂度是 $O(\log n)$ 的。从而，基于伸展的插入、查询、删除等操作的时间复杂度也为均摊 $O(\log n)$。

结论：

在进行 m 次伸展操作之后，实际成本

$$\begin{aligned}
\sum_{i=1}^{m} t_i &= \sum_{i=1}^{m} (c_i + \varphi_{i-1} - \varphi_i) \\
&= \sum_{i=1}^{m} c_i + \varphi_0 - \varphi_m \\
&\le m(3\log n + 1) + n\log n.
\end{aligned}$$

因此，m 次伸展操作的实际时间复杂度为 $O((m+n)\log n)$。

为什么 Splay 树的再平衡操作可以获得 $O(\log n)$ 的均摊复杂度？

朴素的再平衡思路就是对节点反复进行旋转操作使其上升，直到它成为根节点。这种朴素思路的问题在于，对于所有子节点都是左（右）节点的链状树来说，它相当于反复进行 **zig** 操作，因而 **zig** 操作的均摊复杂度中的常数项 1 会不断累积，造成最终的均摊复杂度达到 $O(\log n + n)$ 级别。Splay 树的再平衡操作的设计，避免了连续 **zig** 的情形中的常数累积，使得一次完整的伸展操作中，至多进行一次单独的 **zig** 操作，从而优化了时间复杂度。

平衡树操作

本节讨论基于 Splay 树实现平衡树的常见操作的方法。其中，较为重要的是按照值或排名查找元素，它们可以将某个特定的元素找到，并上移至根节点处，以便后续处理。

作为例子，本节将讨论模板题目 普通平衡树 的实现。

按照值查找

作为二叉查找树，可以通过值 v 查找到相应的节点，只需要将待查找的值 v 和当前节点的值比较即可，找到后将该元素上移至根部即可。

应注意，经常存在树中不存在相应的节点的情形。对于这种情形，要记录最后一个访问的节点（即实现中的 y），并将 y 上移至根部。此时，节点 y 存储的值必然要么是所有小于 v 的元素中最大的（即 v 的前驱），要么是所有大于 v 的元素中最小的（即 v 的后继）。这是因为查找过程保证，左子树总是存储小于 v 的值，而右子树总是存储大于 v 的值。

实现

```
void find(int& z, int v) {
  int x = z, y = fa[x];
  for (; x && val[x] != v; x = ch[y = x][v > val[x]]);
  splay(z, x ? x : y);
}
```

该实现允许指定任何节点 z 作为根节点，并在它的子树内按值查找。

按照排名访问

因为记录了子树大小信息，所以 Splay 树还可以通过排名访问元素，即查找树中第 k 小的元素。

设 k 为剩余排名，具体步骤如下：

- 如果左子树非空且剩余排名 k 不大于左子树的大小，那么向左子树查找；
- 否则，如果 k 不大于左子树加上根的大小，那么根节点就是要寻找的；
- 否则，将 k 减去左子树的和根的大小，继续向右子树查找；
- 将最终找到的元素上移至根部。

实现

```cpp
void loc(int& z, int k) {
  int x = z;
  for (;;) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
    } else if (sz[ch[x][0]] + cnt[x] >= k) {
      break;
    } else {
      k -= sz[ch[x][0]] + cnt[x];
      x = ch[x][1];
    }
  }
  splay(z, x);
}
```

该实现需要保证排名 k 不超过根 z 处的树大小。

模板题目中操作 4 要求按照排名返回值，直接调用该方法，并返回值即可。

实现

```cpp
int find_kth(int k) {
  if (k > sz[rt]) return -1;
  loc(rt, k);
  return val[rt];
}
```

合并操作

有些时候需要合并两棵 Splay 树。

设两棵树的根节点分别为 x 和 y，那么为了保证结果仍是二叉查找树，需要要求 x 树中的最大值小于 y 树中的最小值。这条件通常都可以满足，因为两棵树往往是从更大的子树中分裂出的。

合并操作如下：

- 如果 x 和 y 其中之一或两者都为空树，直接返回不为空的那一棵树的根节点或空树；
- 否则，通过 `loc(y, 1)` 将 y 树中的最小值上移至根 y 处，再将它的左节点（此时必然为空）设置为 x，并更新节点信息，返回节点 y。

实现

```cpp
int merge(int x, int y) {
  if (!x || !y) return x | y;
  loc(y, 1);
  ch[y][0] = x;
  fa[x] = y;
  push_up(y);
  return y;
}
```

分裂操作类似。因而，Splay 树可以模拟 无旋 treap 的思路做各种操作，包括区间操作。后文 会介绍更具有 Splay 树风格的区间操作处理方法。

插入操作

插入操作是一个比较复杂的过程。具体步骤如下：（假设插入的值为 v）

- 类似按值查找的过程，根据 v 向下查找到存储 v 的节点或者空节点，过程中记录父节点 y；
- 如果存在存储 v 的节点 x，直接更新信息，否则就新建节点 x；
- 做伸展操作，将最后一个节点 x 上移至根部。

实现

```cpp
void insert(int v) {
  int x = rt, y = 0;
  for (; x && val[x] != v; x = ch[y = x][v > val[x]]);
  if (x) {
    ++cnt[x];
    ++sz[x];
  } else {
    x = ++id;
    val[x] = v;
    cnt[x] = sz[x] = 1;
    fa[x] = y;
    if (y) ch[y][v > val[y]] = x;
  }
  splay(rt, x);
}
```

该实现允许直接向空树内插入值。若不想处理空树，可以在树中提前插入哑节点。

删除操作

删除操作也是一个比较复杂的操作。具体步骤如下：（假设删除的值为 v）

- 首先按照值 v 查找存储它的节点，并上移至根部；
- 如果不存在存储它的节点，直接返回；（上一步已经做了伸展操作）
- 否则，更新节点信息；
- 如果得到的根节点为空节点，就合并左右子树作为新的根节点，注意合并前需要更新两个子树的根的父节点为空。

实现

```cpp
bool remove(int v) {
  find(rt, v);
  if (!rt || val[rt] != v) return false;
  --cnt[rt];
  --sz[rt];
  if (!cnt[rt]) {
    int x = ch[rt][0];
    int y = ch[rt][1];
    fa[x] = fa[y] = 0;
    rt = merge(x, y);
  }
  return true;
}
```

查询排名

直接按照值 v 访问节点（并上移至根），然后返回相应的值即可。

注意，当 v 不存在时，方法 `find(rt, v)` 返回的根和 v 的大小关系无法确定，需要单独讨论。

实现

```cpp
int find_rank(int v) {
  find(rt, v);
  return sz[ch[rt][0]] + (val[rt] < v ? cnt[rt] : 0) + 1;
}
```

查询前驱

前驱定义为小于 v 的最大的数。具体步骤如下：

- 按照值 v 访问节点（并上移至根部）；
- 如果根部的值小于 v，那么它必然是最大的那个，直接返回；
- 否则，在左子树中找到最大值，并上移至根部。

> 最后一步相当于直接调用 `loc(ch[rt][0], sz[ch[rt][0]])`，只是省去了不必要的判断。

实现

```
int find_prev(int v) {
  find(rt, v);
  if (rt && val[rt] < v) return val[rt];
  int x = ch[rt][0];
  if (!x) return -1;
  for (; ch[x][1]; x = ch[x][1]);
  splay(rt, x);
  return val[rt];
}
```

该实现允许前驱不存在，此时返回 -1。

查询后继

后继定义为大于 x 的最小的数。查询方法和前驱类似，只是将左子树的最大值换成了右子树的最小值，即调用 `loc(ch[rt][1], 1)`。

实现

```
int find_next(int v) {
  find(rt, v);
  if (rt && val[rt] > v) return val[rt];
  int x = ch[rt][1];
  if (!x) return -1;
  for (; ch[x][0]; x = ch[x][0]);
  splay(rt, x);
  return val[rt];
}
```

参考实现

本节的最后，给出模板题目 普通平衡树 的参考实现。

参考实现

```cpp
#include <iostream>

constexpr int N = 2e6;
int id, rt;
int fa[N], val[N], cnt[N], sz[N], ch[N][2];

bool dir(int x) { return x == ch[fa[x]][1]; }

void push_up(int x) { sz[x] = cnt[x] + sz[ch[x][0]] + sz[ch[x]
[1]]; }

void rotate(int x) {
  int y = fa[x], z = fa[y];
  bool r = dir(x);
  ch[y][r] = ch[x][!r];
  ch[x][!r] = y;
  if (z) ch[z][dir(y)] = x;
  if (ch[y][r]) fa[ch[y][r]] = y;
  fa[y] = x;
  fa[x] = z;
  push_up(y);
  push_up(x);
}

void splay(int& z, int x) {
  int w = fa[z];
  for (int y; (y = fa[x]) != w; rotate(x)) {
    if (fa[y] != w) rotate(dir(x) == dir(y) ? y : x);
  }
  z = x;
}

void find(int& z, int v) {
  int x = z, y = fa[x];
  for (; x && val[x] != v; x = ch[y = x][v > val[x]]);
  splay(z, x ? x : y);
}

void loc(int& z, int k) {
  int x = z;
  for (;;) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
    } else if (sz[ch[x][0]] + cnt[x] >= k) {
      break;
    } else {
      k -= sz[ch[x][0]] + cnt[x];
      x = ch[x][1];
    }
```

```cpp
  }
  splay(z, x);
}

int merge(int x, int y) {
  if (!x || !y) return x | y;
  loc(y, 1);
  ch[y][0] = x;
  fa[x] = y;
  push_up(y);
  return y;
}

void insert(int v) {
  int x = rt, y = 0;
  for (; x && val[x] != v; x = ch[y = x][v > val[x]]);
  if (x) {
    ++cnt[x];
    ++sz[x];
  } else {
    x = ++id;
    val[x] = v;
    cnt[x] = sz[x] = 1;
    fa[x] = y;
    if (y) ch[y][v > val[y]] = x;
  }
  splay(rt, x);
}

bool remove(int v) {
  find(rt, v);
  if (!rt || val[rt] != v) return false;
  --cnt[rt];
  --sz[rt];
  if (!cnt[rt]) {
    int x = ch[rt][0];
    int y = ch[rt][1];
    fa[x] = fa[y] = 0;
    rt = merge(x, y);
  }
  return true;
}

int find_rank(int v) {
  find(rt, v);
  return sz[ch[rt][0]] + (val[rt] < v ? cnt[rt] : 0) + 1;
}

int find_kth(int k) {
  if (k > sz[rt]) return -1;
  loc(rt, k);
  return val[rt];
```

```
}

int find_prev(int v) {
  find(rt, v);
  if (rt && val[rt] < v) return val[rt];
  int x = ch[rt][0];
  if (!x) return -1;
  for (; ch[x][1]; x = ch[x][1]);
  splay(rt, x);
  return val[rt];
}

int find_next(int v) {
  find(rt, v);
  if (rt && val[rt] > v) return val[rt];
  int x = ch[rt][1];
  if (!x) return -1;
  for (; ch[x][0]; x = ch[x][0]);
  splay(rt, x);
  return val[rt];
}

int main() {
  int n;
  std::cin >> n;
  for (; n; --n) {
    int op, x;
    std::cin >> op >> x;
    switch (op) {
      case 1:
        insert(x);
        break;
      case 2:
        remove(x);
        break;
      case 3:
        std::cout << find_rank(x) << '\n';
        break;
      case 4:
        std::cout << find_kth(x) << '\n';
        break;
      case 5:
        std::cout << find_prev(x) << '\n';
        break;
      case 6:
        std::cout << find_next(x) << '\n';
        break;
    }
  }
  return 0;
}
```

序列操作

Splay 树也可以运用在序列上，用于维护区间信息。与线段树对比，Splay 树常数较大，但是支持更复杂的序列操作，如区间翻转等。上文提到 Splay 树同样支持分裂和合并操作，因而可以模拟 无旋 treap 进行区间操作，在此不再过多讨论。本节主要讨论基于伸展操作的区间操作实现方法。

将序列建成的 Splay 树有如下性质：

- Splay 树的中序遍历相当于原序列从左到右的遍历；
- Splay 树上的一个节点代表原序列的一个元素；
- Splay 树上的一颗子树，代表原序列的一段区间。

因为有伸展操作，可以快速提取出代表某个区间的 Splay 子树。

作为例子，本节将讨论模板题目 文艺平衡树 的实现。

根据序列建树

在操作之前，需要根据所给的序列先把 Splay 树建出来。根据 Splay 树的特性，直接建出一颗只有左儿子的链即可。时间复杂度是 $O(n)$ 的。

参考实现

```
void build(int n) {
  for (int i = 1; i <= n + 2; ++i) {
    ++id;
    ch[id][0] = rt;
    if (rt) fa[rt] = id;
    rt = id;
    val[id] = i - 1;
  }
  splay(rt, 1);
}
```

最后的伸展操作自下而上地更新了节点信息。为了后文区间操作方便，序列左右两侧添加了两个哨兵节点。

区间翻转

以区间翻转为例，可以理解区间操作的方法：（设区间为 $[L, R]$）

- 首先将节点 $L-1$ 上移到根节点，再在其右子树中，将节点 $R+1$ 上移到右子树的根节点；
- 此时，设 x 为根节点的右子节点的左子节点，则以 x 为根的子树就对应着区间 $[L, R]$；

- 在 x 处对区间 $[L, R]$ 做操作，并打上懒标记；
- 在 x 处将标记下传一次，然后利用伸展操作将 x 上移到根。

第一步需要的操作就是前文平衡树操作中的「按照排名访问」，因为元素的标号就是它的排名。因为涉及懒标记的管理，它的实现与上文略有不同。

参考实现

```cpp
void reverse(int l, int r) {
  loc(rt, l);
  loc(ch[rt][1], r - l + 2);
  int x = ch[ch[rt][1]][0];
  lazy_reverse(x);
  push_down(x);
  splay(rt, x);
}
```

最后一步的伸展操作并非为了保证复杂度正确，而是为了更新节点信息。因为伸展操作涉及到节点 x 的左右子节点，所以之前需要将节点 x 处的标记先下传一次。当然，仅对于区间翻转操作而言，子区间的翻转不会对祖先节点产生影响，所以省去这一步骤也是正确的。此处实现保留这两行，是为了说明一般的情形下的操作方法。

懒标记管理

首先，需要辅助函数 `lazy_reverse(x)` 和 `push_down(x)`。前者交换左右节点，并更新懒标记；后者将标记下传。

参考实现

```cpp
void lazy_reverse(int x) {
  std::swap(ch[x][0], ch[x][1]);
  lz[x] ^= 1;
}

void push_down(int x) {
  if (lz[x]) {
    if (ch[x][0]) lazy_reverse(ch[x][0]);
    if (ch[x][1]) lazy_reverse(ch[x][1]);
    lz[x] = 0;
  }
}
```

然后，只需要在向下经过节点时下传标记即可。模板题要求的操作比较简单，只有按照排名寻找的操作（即 `loc`）涉及向下访问节点。注意，需要在函数每次访问一个新的节点 **前** 下传标记。

参考实现

```
void loc(int& z, int k) {
  int x = z;
  for (push_down(x); sz[ch[x][0]] != k - 1; push_down(x)) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
    } else {
      k -= sz[ch[x][0]] + 1;
      x = ch[x][1];
    }
  }
  splay(z, x);
}
```

因为向下访问节点时已经移除了经过的路径的所有懒标记，所以利用伸展操作上移节点时不再需要处理懒标记。但是，对于区间操作的那一个节点要谨慎处理：因为它同样位于伸展操作的路径上，但是刚刚操作完，可能存在尚未下传的标记，需要首先下传再做伸展操作，正如同上文所做的那样。

参考实现

本节的最后，给出模板题目 文艺平衡树 的参考实现。

参考实现

```cpp
#include <iostream>

constexpr int N = 2e6;
int id, rt;
int fa[N], val[N], sz[N], lz[N], ch[N][2];

bool dir(int x) { return x == ch[fa[x]][1]; }

void push_up(int x) { sz[x] = 1 + sz[ch[x][0]] + sz[ch[x][1]]; }

void lazy_reverse(int x) {
  std::swap(ch[x][0], ch[x][1]);
  lz[x] ^= 1;
}

void push_down(int x) {
  if (lz[x]) {
    if (ch[x][0]) lazy_reverse(ch[x][0]);
    if (ch[x][1]) lazy_reverse(ch[x][1]);
    lz[x] = 0;
  }
}

void rotate(int x) {
  int y = fa[x], z = fa[y];
  bool r = dir(x);
  ch[y][r] = ch[x][!r];
  ch[x][!r] = y;
  if (z) ch[z][dir(y)] = x;
  if (ch[y][r]) fa[ch[y][r]] = y;
  fa[y] = x;
  fa[x] = z;
  push_up(y);
  push_up(x);
}

void splay(int& z, int x) {
  int w = fa[z];
  for (int y; (y = fa[x]) != w; rotate(x)) {
    if (fa[y] != w) rotate(dir(x) == dir(y) ? y : x);
  }
  z = x;
}

void loc(int& z, int k) {
  int x = z;
  for (push_down(x); sz[ch[x][0]] != k - 1; push_down(x)) {
    if (sz[ch[x][0]] >= k) {
      x = ch[x][0];
```

```cpp
    } else {
      k -= sz[ch[x][0]] + 1;
      x = ch[x][1];
    }
  }
  splay(z, x);
}

void build(int n) {
  for (int i = 1; i <= n + 2; ++i) {
    ++id;
    ch[id][0] = rt;
    if (rt) fa[rt] = id;
    rt = id;
    val[id] = i - 1;
  }
  splay(rt, 1);
}

void reverse(int l, int r) {
  loc(rt, l);
  loc(ch[rt][1], r - l + 2);
  int x = ch[ch[rt][1]][0];
  lazy_reverse(x);
  push_down(x);
  splay(rt, x);
}

void print(int x) {
  if (!x) return;
  push_down(x);
  print(ch[x][0]);
  std::cout << val[x] << ' ';
  print(ch[x][1]);
}

void print() {
  loc(rt, 1);
  loc(ch[rt][1], sz[rt] - 1);
  print(ch[ch[rt][1]][0]);
}

int main() {
  int n, m;
  std::cin >> n >> m;
  build(n);
  for (; m; --m) {
    int l, r;
    std::cin >> l >> r;
    reverse(l, r);
  }
  print();
```

```
    return 0;
}
```

习题

这些题目都是裸的 Splay 树维护二叉查找树：

- 【模板】普通平衡树
- 【模板】文艺平衡树
- 「HNOI2002」营业额统计
- 「HNOI2004」宠物收养所

Splay 树还出现在更复杂的应用场景中：

- 「Cerc2007」robotic sort 机械排序
- 「HNOI2011」括号修复 / 「JSOI2011」括号序列
- 二逼平衡树（树套树）
- BZOJ 2827 千山鸟飞绝
- 「Lydsy1706 月赛」K 小值查询
- POJ3580 SuperMemo

参考资料与注释

本文部分内容引用于 algocode 算法博客，特别鸣谢！

本页面最近更新：2025/7/2 22:02:43，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, StudyingFather, c-forrest, H-J-Granger, countercurrent-time, NachtgeistW, Tiphereth-A, sshwy, Xeonacid, Early0v0, Enter-tainer, ezoixx130, yyyu-star, AngelKitty, CCXXXI, cjsoft, diauweb, GavinZhengOI, GekkaSaori, Gesrua, Henry-ZHR, Konano, LovelyBuggies, Makkiy, mgt, minghu6, P-Y-Y, PotassiumWings, SamZhangQingChuan, shuzhouliu, Siyuan, Suyun514, weiyong1024, zzxLLLL, abc1763613206, Alpacabla, aofall, Catreap, CoelacanthusHex, GrapeLemonade, Great-designer, HeRaNO, hly1204, hsfzLZH1, iamtwz, isdanni, ksyx, kxccc, longlongzhu123, lychees, Macesuted, Marcythm, mcendu, Molmin, ouuan, partychicken, Peanut-Tang, Persdre, saigonoinorio, SukkaW, xiaofu-15191, yuhuoji, zcz0263, 代建杉