# 常用数据结构

NiroBC

Aug 22, 2021

## 1 堆

堆是这样的一个容器，支持以下操作：

1. 给定一个数 $x$，将 $x$ 加入该容器，若有重复元素就都保留。
2. 若该容器非空，输出该容器中最大的元素。
3. 若该容器非空，删去一个该容器中最大的元素。

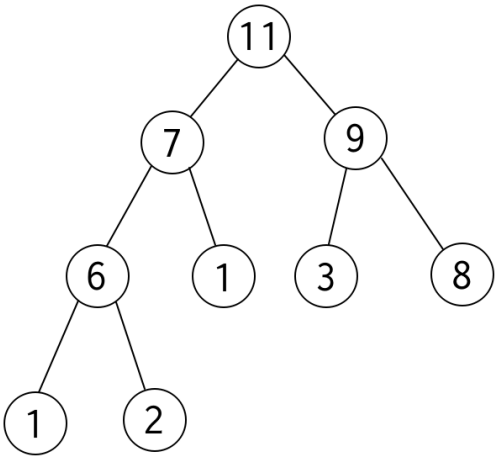
### 1.1 传统堆

传统堆，当然就是简简单单的一个堆啦。

如何实现这样的堆呢?

考虑一棵完全二叉树，它的每个节点的权值代表容器中的一个元素。对于每一个非根节点，都满足它的权值小于等于它的父亲的权值。这样，这棵二叉树中最大的元素就是根节点。

下图就是存有元素 $\{1, 1, 2, 3, 6, 7, 8, 9, 11\}$ 的可能的二叉树：




如何对其进行操作?

对于加入操作，我们在完全二叉树的最底下一层的第一个空的位置插入该元素。为了维持堆的性质，我们将这个元素不断和其父亲比较，如果大小关系不满足堆的性质，就和其父亲交换，直到向上移动到一个合适的位置。

对于删除操作，我们把根节点的元素删去之后，把完全二叉树最底下一层的最右边一个元素填到根节点的位置。为了维持堆的性质，我们将这个元素不断

和其儿子比较，如果大小关系不满足堆的性质，就和较大的那个儿子交换，直到向下移动到一个合适的位置。

## 1.2 可并堆-左偏树

可并堆，就是维护一系列堆的集合，除了传统的堆的操作外，还可以将两个堆合并，得到一个新的堆。

可并堆，我们也可以用二叉树来维护。(不必是完全二叉树哦)

我们先来看最重要的合并操作吧。假如我们要将根节点分别为 $x$ 和 $y$ 的堆合并，那么可以写出如下代码:

```
int merge(int x, int y)
{
    if (x == 0 || y == 0)
        return x + y;
    if (w[x] < w[y])
        swap(x, y);
    rs[x] = merge(rs[x], y);
    return x;
}
```

假设节点 $x$ 的权值大于等于节点 $y$ 的权值，那么，新的堆将以节点 $x$ 为根，新的堆中，根的左子树依然是 $x$ 原来的左子树，根的右儿子将是 $x$ 的右子树和整个 $y$ 合并的结果。

这时，有了一个复杂度的问题，这个合并函数的复杂度正比于二叉树的右链深度，即从根开始不断往右儿子走，能走几步。而右链深度在最坏情况下是 $O(n)$ 的。

如果我们可以强制，对于每一个节点，它的左儿子的右链深度大于等于右儿子的右链深度 (方便起见，空节点的右链深度记作 $-1$)，那么，根节点的右链深度将是 $O(\log n)$ 级别的。

如何维护这个性质? 在合并之后，如果发现右儿子的右链深度比左儿子大，交换两个孩子就好啦。

于是有了这样的代码:

```
int merge(int x, int y)
{
    if (x == 0 || y == 0)
        return x + y;
    if (w[x] < w[y])
        swap(x, y);
    rs[x] = merge(rs[x], y);
    if (rdep[ls[x]] < rdep[rs[x]])
        swap(ls[x], rs[x]);
    rdep[x] = rdep[rs[x]] + 1;
    return x;
}
```

说完了合并操作，我们来讲讲插入操作和删除操作。

插入操作就是将一个堆和一个只有一个元素的堆合并。

删除操作就是将一个堆变成其左子树和右子树的合并结果，这样根节点就没有了。

这样，插入和删除都可以调用 `merge` 函数来实现。

# 2 并查集

并查集解决的是这样的问题：维护一张一开始没有边的无向图，支持以下两个操作：

1. 在两个点之间加上一条边。
2. 询问两个点是否联通。

有一个很暴力的想法：我们维护一个有根树的森林，两个点在同一个联通块，当且仅当它们所在的有根树的根节点相同。连接两个点的时候，如果它们不在同一个联通块，就找出他们各自所在有根树的根，将其中一个根的父亲变成另一个根。

```
int find_root(int x)
{
    while (father[x] != 0)
        x = father[x];
    return x;
}
void add(int u, int v)
{
    u = find_root(u);
    v = find_root(v);
    father[u] = v;
}
bool check(int u, int v)
{
    return find_root(u) == find_root(v);
}
```

很遗憾，有根树的深度可能是 $O(n)$，所以单次操作的时间复杂度是 $O(n)$。

## 2.1 按秩合并

**秩**是什么？**秩**就是对于一个有根树，记下其深度最深的点的深度。在进行连边操作时，总是让秩较大的根成为秩较小的根的父亲。这样，如果两个有根树的秩相同，新有根树的秩将是 $\text{原秩}+1$，如果秩不同，新有根树的秩将依旧是原来较大的秩。

这么做有什么好处？我们发现秩永远是 $O(\log n)$ 级别的！于是，并查集的复杂度被优化到了 $O(\log n)$。

## 2.2 启发式合并

和按秩合并类似，启发式合并需要记下每个有根树的点数，总是让点数较大的有根树的根成为点数较小的有根树的父亲，这么做的时间复杂度也是 $O(\log n)$。

## 2.3 路径压缩

加一个小小的优化：每当执行完一个 `find_root` 操作，就将 $x$ 到根路径上所有的点的父亲都改成根节点，这样，它们的深度都被压缩成了 1。

这个优化的时间复杂度证明十分高深。

当这个优化和按秩合并一起用时，均摊复杂度是 $O(\alpha(n))$，可以视作常数。

# 3 分块

## 3.1 单点修改，区间查询

来看一个经典的问题，维护一个序列 $a_1, a_2, \ldots, a_n$，支持以下操作：

1. 给定 $x, y$，使得 $a_x$ 的值加上 $y$。

2. 给定 $l, r$，求 $a_l + a_{l+1} + \ldots + a_r$。

显然有一个修改 $O(1)$，询问 $O(n)$ 的算法。
也可以通过记录前缀和的方式得到一个修改 $O(n)$，询问 $O(1)$ 的算法。

```
void add(int x, int y)
{
    for (int i = x; i <= n; i++)
        sum[i] += y;
}
int query(int l, int r)
{
    return sum[r] - sum[l - 1];
}
```

如何优化？方便起见，令 $n = k^2$，我们将整个序列分成 $k$ 块，每块 $k$ 个。

| | | | |
|---|---|---|---|
| $a_1, a_2, \ldots, a_k$ | $a_{k+1}, a_{k+2}, \ldots, a_{2k}$ | $\ldots$ | $a_{k(k-1)+1}, a_{k(k-1)+2}, \ldots, a_{k^2}$ |

除了单独记下 $a_1, a_2, \ldots, a_n$ 的值，我们还要记下每块的和。每当修改一个数，我们需要修改这个数本身，以及它所在块的和，这样，修改依然是 $O(1)$ 的。

对于询问，我们对于区间内部完整的块，直接取出块的和，对于零散部分，一个一个加入答案。例如，当 $n = 16, k = 4$ 时，我们询问 $[3, 13]$ 的和，计算示意如下：

$$a_3 + a_4 + (a_5 + a_6 + a_7 + a_8) + (a_9 + a_{10} + a_{11} + a_{12}) + a_{13}$$

完整的块最多 $O(\sqrt{n})$ 个，零散部分最多 $O(\sqrt{n})$ 个，询问的时间复杂度 $O(\sqrt{n})$。

有了修改 $O(1)$，询问 $O(\sqrt{n})$ 的算法，我们再来想想修改 $O(\sqrt{n})$，询问 $O(1)$ 的算法。

模仿修改 $O(n)$，询问 $O(1)$ 的算法，我们需要记录一些类似前缀和的东西。

对于每个 $1 \le i \le k$，我们记录前 $i$ 块所有数的和，即

$$\sum_{j=1}^{ik} a_j$$

。

对于每个块内部，我们记录这个块内部的前缀和。

每当修改一个数，受到影响的数据有 $O(\sqrt{n})$ 个，修改的复杂度是 $O(\sqrt{n})$。

询问时，完整块的部分何以用 前 $i$ 块所有数的和 计算得到，零散部分可以通过块内前缀和计算得到，询问复杂度 $O(1)$。

### 3.2 数颜色

给定一个长度为 $n$ 的颜色序列 $c_1, c_2, \ldots, c_n$，多次询问，每次询问给定 $l, r$，求有多少 $l \le i < j \le r$ 满足 $c_i = c_j$，即区间内部相同颜色的对数。

同样地，将序列进行分块，用 $O(n\sqrt{n})$ 的时间预处理出所有左右区间端点都在块的分界处的答案，以及对于每个 $i, j$，求出颜色 $i$ 在前 $j$ 个块中出现的次数 $appear_{i,j}$。

询问时，先拿出内部完整的块的答案。再将零碎部分一个一个加进去。每加进去一个数，答案的变化有两部分：

- 新加进去的数和已加进去的零碎部分之间产生的贡献，这个可以开一个数组记录下已加进去的零碎部分中各个颜色分别出现了几次。
- 新加进去的数和一开始的完整的块的部分产生的贡献，这个可以通过预处理出的 $appear_{i,j}$ 计算得到。

这样，我们就得到了一个预处理 $O(n\sqrt{n})$，内存复杂度 $O(n\sqrt{n})$，单次询问 $O(\sqrt{n})$ 的算法。

## 4 线段树

### 4.1 单点修改，区间查询

还是来看这个经典的问题，维护一个序列 $a_1, a_2, \ldots, a_n$，支持以下操作：

1. 给定 $x, y$，使得 $a_x$ 的值加上 $y$。
2. 给定 $l, r$，求 $a_l + a_{l+1} + \ldots + a_r$。

考虑在分块的基础上进行优化。将分块的层数增加，我们可以想到一个修改 $O(1)$，询问 $O(n^{\frac{1}{3}})$ 的算法。

将整个序列分成 $n^{\frac{1}{3}}$ 个大块，每个大块 $n^{\frac{2}{3}}$ 个元素。

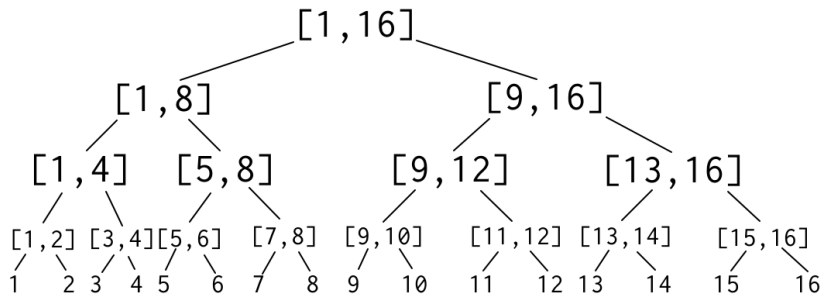
再将每个大块继续分成 $n^{\frac{1}{3}}$ 个小块，每个小块 $n^{\frac{1}{3}}$ 个元素。

实时记下每个元素单独的值，每个小块的和，以及每个大块的和。修改时，只会改变 3 个数。询问时，只需要拿出 $O(n^{\frac{1}{3}})$ 个数相加。

再继续推广这个算法，假设分块的层数为 $d$，我们可以得到修改 $O(d)$，询问 $O(dn^{\frac{1}{d}})$ 的算法。当然，使用记录前缀和的方式，还能得到修改 $O(dn^{\frac{1}{d}})$，询问 $O(d)$ 的算法。

取 $d = \log_2 n$，每一级的块都往下分成 2 个小块，形成一个二叉树的结构，就是经典的线段树算法。线段树算法的查询和修改都是 $O(\log n)$。

例如 $n = 16$，形成如下的二叉树结构，每个节点都代表一个区间，我们为每个区间记录下这个区间的和。修改时，只会影响 $O(\log n)$ 个数据。询问时，只要把 $O(\log n)$ 个数据取出来相加：



例如，询问 $[6,13]$ 时，我们需要取出 $6,[7,8],[9,12],13$ 这几个数据。

## 4.2 线段树上二分

维护一个可重集，一开始为空，支持以下操作：

1. 加入一个数。
2. 删除一个数。
3. 询问可重集中第 $k$ 小的值。

其中所有加入的数都在 $10^5$ 范围内。

建立出下标范围为 $10^5$ 的线段树，线段树上每个区间表示这个可重集中值位于该区间内的数有几个。

修改较为简单。

询问时，我们可以从线段树的根节点出发。设当前我们位于线段树节点 $[l,r]$，想要找到该区间中第 $k$ 小的值。

如果左子树的值 $\geq k$，就说明值位于 $[l,mid]$ 的数 $\geq k$ 个，也就是第 $k$ 小的值 $\leq mid$，那就往左子树走。

否则就说明第 $k$ 小的值 $> mid$，那么就让 $k = k - \text{Tree}[l,mid]$ 后往右子树走。

直到走到最底层，就能确定答案。

这相当于是利用线段树上的数据进行了二分答案。

## 4.3 Lazy Tag

考虑这样一道题：

维护一个长度为 $N$ 的序列 $a_1,a_2,\ldots,a_N$，依次进行 $Q$ 个操作：

1. 给定 $l,r,k$，将 $a_l,\ldots,a_r$ 全都改成 $k$。
2. 给定 $l,r,k$，将 $a_l,\ldots,a_r$ 全都加上 $k$。
3. 给定 $l,r$，求 $\sum_{i=l}^{r} a_i$。

显然，每个节点上需要存下对应区间的 $a_i$ 的和。

然而，我们遇到了一个问题：对于 1. 与 2. 操作，我们将待修改区间划分成 $O(\log n)$ 个区间后，需要修改信息的不是 $O(\log n)$ 个节点，而是 $O(\log n)$ 个子树！

那怎么办？我们当然不能老老实实去改。我们只在待修改的子树的根部打上一个“即将要去改”的标记。

那就会有问题，除了这棵子树的根节点，其他下方的节点上存的数据都是过时的。

不过没关系，在线段树上，我们是从上到下访问的。只有在需要访问这个子树中下方的元素时，才把根节点处的一个标记分成两个标记，放到两个儿子处，并在两个儿子处真正地修改。

这样，下放标记不会带来额外的复杂度，并且我们依然能时刻访问到每个节点准确的数据，整个线段树算法的复杂度依然是 $O(\log n)$。

# 5 平衡树

维护一个数的可重集，支持如下操作：

1. 插入一个数 $x$。
2. 删除一个数 $x$。
3. 给定一个数 $x$，求出集合中最小的大于等于 $x$ 的数。
4. 给定一个数 $x$，求出集合中最大的小于等于 $x$ 的数。
5. 给定一个数 $x$，求小于等于 $x$ 的数的数量。
6. 给定一个数 $x$，求大于等于 $x$ 的数的数量。

## 5.1 二叉查找树

我们可以用一棵二叉查找树来进行以上操作。

这棵二叉树有这样的性质：中序遍历在前的点的权值一定小于等于中序遍历在后的点的权值，也就是说，对于任何一个点，它的左子树中所有点的权值都小于等于它，它的右子树中所有点的权值都大于等于它。

对于插入操作，我们调用函数 `insert(root, x)`，伪代码如下：

```
insert(p, x)
{
    if (p 为空)
    {
        在 p 处插入权值为 x 的节点;
        return;
    }
    if (x <= p 的权值)
        insert(p 的左儿子, x);
    else
        insert(p 的右儿子, x);
}
```

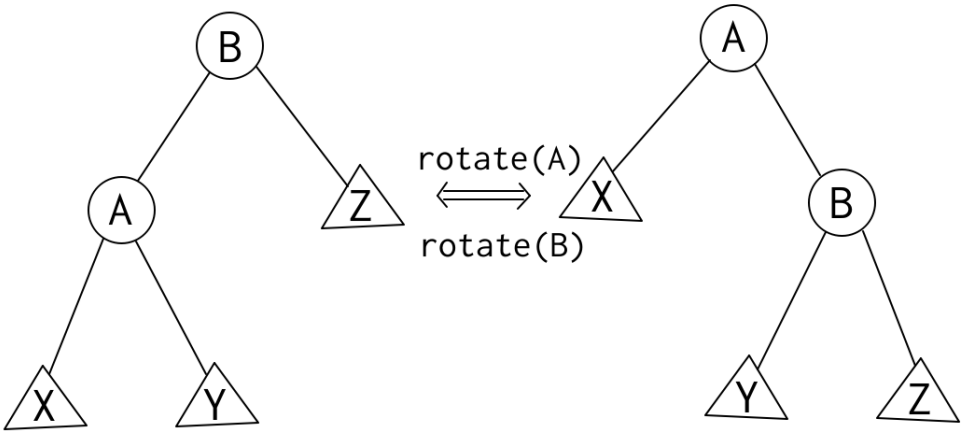
对于删除操作，我们首先要用以上方法找到待删节点，然后：

1. 如果待删节点是叶子节点，直接删去即可。
2. 如果待删节点只有左儿子，那么将其删去后，将其左儿子提上来。只有右儿子的情况同理。
3. 如果待删节点两个儿子都有，那么我们需要找出左子树中最右下方的节点，将其提到原来待删节点的位置。原左子树中最右下方的节点处相当于删去了一个只有左儿子的点或叶子节点，按照 1. 或 2. 处理即可。

其他操作也可以用类似如上的从根开始往下寻找的方式实现，时间复杂度和树的高度成正比，是 $O(n)$ 的。

## 5.2 旋转

由于树的高度可能到达 $O(n)$，时间复杂度十分不妙，我们需要通过一些方式来控制树的高度，同时需要保证树的中序遍历不变。




可以发现，树的形态发生了变化，但中序遍历没有变化，一直都是 `XAYBZ`。

我们可以通过旋转，将树的高度维持在一个可以接受的水平，以降低时间复杂度。

## 5.3 AVL

我们定义一个节点 $p$ 的高度 $\text{height}_p$ 为其子树中最深的节点与节点 $p$ 的距离，空节点的深度 $\text{height}_\emptyset$ 定义为 $-1$。对于一个节点，如果它的两个儿子的 height 的差的绝对值不超过 1，我们就说它是平衡的。在任一时刻，我们需要保证树上的每个节点都是平衡的。

插入或删除节点后，可能有一些节点不再平衡，我们需要通过适当的旋转操作使得整棵树重新平衡。

接下来我们开始分类讨论。

### 5.3.1 插入

如果没有点因此次插入操作变得不平衡，当然什么都不用做。否则，我们设进行插入操作后整棵树上最深的不平衡的点为 `A`，它是在 `C` 子树中进行了插入操作后才变得不平衡的。以下按 `C` 是 `A` 的左儿子的左儿子或左儿子的右儿子进行讨论：

第一种：

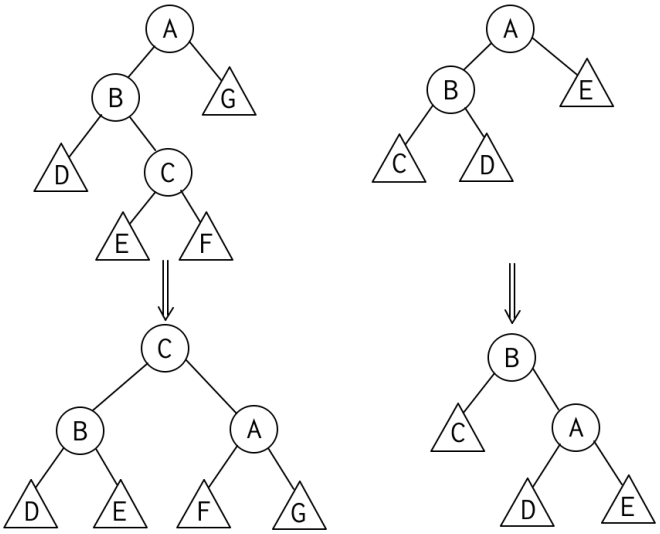
$\text{height}_{\texttt{B}} = h+2, \text{height}_{\texttt{G}} = h, \text{height}_{\texttt{D}} = h, \text{height}_{\texttt{C}} = h+1, h-1 \le \text{height}_{\texttt{E}}, \text{height}_{\texttt{F}} \le h$

第一种调整后：$\text{height}_{\texttt{B}} = \max\{\text{height}_{\texttt{D}}, \text{height}_{\texttt{E}}\} + 1 = h+1, \text{height}_{\texttt{A}} = \max\{\text{height}_{\texttt{F}}, \text{height}_{\texttt{G}}\} = h+1$，重新达成了平衡

第二种：

$\text{height}_{\texttt{B}} = h+2, \text{height}_{\texttt{E}} = h, \text{height}_{\texttt{C}} = h+1, \text{height}_{\texttt{D}} = h$

第二种调整后：$\text{height}_{\texttt{C}} = h+1, \text{height}_{\texttt{A}} = \max\{\text{height}_{\texttt{D}}, \text{height}_{\texttt{E}}\} = \max\{h, h\}+1 = h+1$，重新达成了平衡




可以发现，调整完之后，所有点都重新平衡，并且整棵子树的深度较插入前没有变化。

既然整棵子树的深度较插入前没有变化，那么 A 的所有祖先的平衡性都没有被破坏，只需调整最多一次即可结束。

### 5.3.2 删除

按照二叉查找树删除节点的方法删去待删节点后，如果没有不平衡的点，当然什么都不用做，否则，我们设进行删除操作后整棵树上最深的不平衡的点为 A。我们在 A 的左子树 B 中进行删除操作后，$\text{height}_{\texttt{B}}$ 较原先减小了 1，才使得 A 不平衡。

分类讨论如下：

第一种：

$\text{height}_{\texttt{B}} = h, \text{height}_{\texttt{C}} = h+2, h \le \text{height}_{\texttt{D}} \le h+1, \text{height}_{\texttt{E}} = h+1$

第一种调整后：

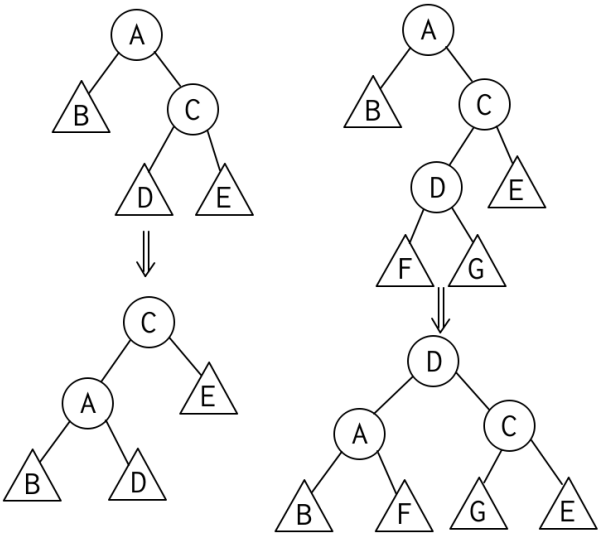
$h+1 \le \text{height}_{\texttt{A}} \le h+2, \text{height}_{\texttt{E}} = h+1$

第二种：

$\text{height}_{\texttt{B}} = h, \text{height}_{\texttt{C}} = h+2, \text{height}_{\texttt{D}} = h+1, \text{height}_{\texttt{E}} = h, h-1 \le \text{height}_{\texttt{F,G}} \le h$

第二种调整后：

$\text{height}_{\texttt{A}} = \max\{\text{height}_{\texttt{B}}, \text{height}_{\texttt{F}}\} = h+1, \text{height}_{\texttt{C}} = \max\{\text{height}_{\texttt{G}}, \text{height}_{\texttt{E}}\} = h+1$




调整后，整棵子树的高度相较于删除前可能不变，也可能减小 1。如果减小了 1，要继续向上找不平衡的节点。这样，进行的旋转次数不超过整棵树的深度。

### 5.3.3 时间复杂度

我们设 $F_h$ 表示深度为 $h$ 的 AVL 树至少有几个点，显然有 $F_h = F_{h-1}+F_{h-2}$，也就是说，$F_h \sim \mathrm{fib}_h \sim \left(\frac{1+\sqrt{5}}{2}\right)^h$。

那么，点数为 $n$ 的 AVL 的深度是 $O(\log n)$ 的。

## 5.4 Splay

Splay 是 Tarjan 发明的一种非常奇妙的平衡树，这种平衡树不需要满足任何性质，仅仅通过一个 `splay` 操作保持其均摊复杂度为 $O(\log n)$，而单次操作复杂度仍有可能达到 $O(n)$。

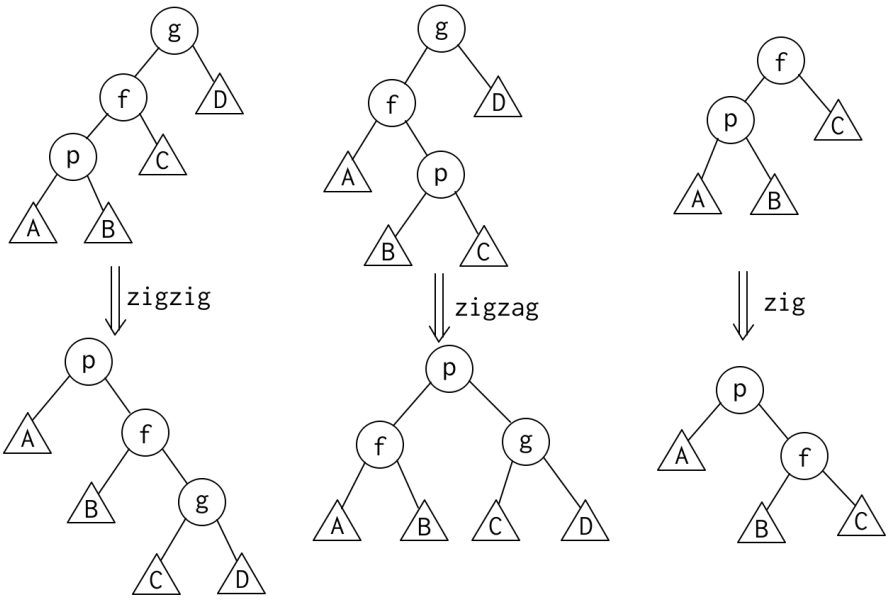
### 5.4.1 `splay` 操作

`splay` 操作是通过不断的旋转将树上某个节点从任意位置提至根节点的操作。

伪代码如下：

```
splay(p)
{
    while (p 不是根)
    {
        if (p 没有爷爷)
            rotate(p); // zig
        else if ((p 的爷爷与 p 的父亲的关系) == (p 的父亲与 p 的关系))
        {
            rotate(p 的父亲);
            rotate(p);
```

```
        } // zigzig
        else
        {
            rotate(p);
            rotate(p);
        } // zigzag
    }
}
```




### 5.4.2 插入

按照传统二叉查找树的方式插入节点后，将这个新的节点 `splay` 至根部。这个 `splay` 操作保证了均摊复杂度为 $O(\log n)$。

### 5.4.3 删除

找出待删节点的前驱和后继。将前驱 `splay` 至根部，再将后继 `splay` 至前驱的右儿子处。此时，根节点的右儿子的左子树（即后继的左子树）中只有一个节点，即待删节点。简单将其删去即可。

### 5.4.4 询问

为保证时间复杂度，在询问完毕后，我们要把询问时找到的节点 `splay` 至根部。

### 5.4.5 时间复杂度分析

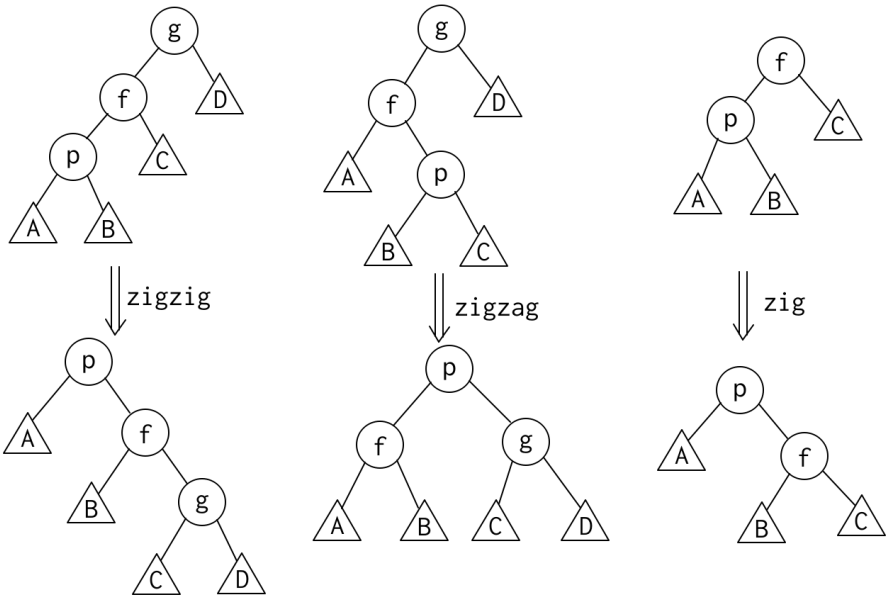
对于一个节点 $p$，我们设 $\text{size}(p)$ 为其子树中节点个数（包括 $p$ 本身），并规定 $\text{rank}(p) = \log_2(\text{size}(p))$。

对于整棵树 $T$，我们设

$$\Phi(T) = \sum_{p \in T} \mathrm{rank}(p)$$

表示势能函数。

我们依次来分析 `zig,zigzig,zigzag` 的势能函数变化。




对于一次 `zigzig`，势能函数的变化为

$$\begin{aligned}
\Delta\Phi(T) &= \mathrm{rank}'(\mathtt{p}) + \mathrm{rank}'(\mathtt{f}) + \mathrm{rank}'(\mathtt{g}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) - \mathrm{rank}(\mathtt{g}) \\
&= \mathrm{rank}'(\mathtt{f}) + \mathrm{rank}'(\mathtt{g}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) \\
&\le \mathrm{rank}'(\mathtt{f}) + \mathrm{rank}'(\mathtt{g}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) + 2\mathrm{rank}'(\mathtt{p}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}'(\mathtt{g}) - 2 \\
&(\forall a, b > 0 \Rightarrow \log_2(a) + \log_2(b) + 2 \le 2\log_2(a+b)) \\
&= \mathrm{rank}'(\mathtt{f}) - 2\mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) + 2\mathrm{rank}'(\mathtt{p}) - 2 \\
&\le \mathrm{rank}'(\mathtt{p}) - 2\mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{p}) + 2\mathrm{rank}'(\mathtt{p}) - 2 \\
&= 3(\mathrm{rank}'(\mathtt{p}) - \mathrm{rank}(\mathtt{p})) - 2
\end{aligned}$$

对于一次 `zigzag`，势能函数的变化为

$$\begin{aligned}
\Delta\Phi(T) &= \mathrm{rank}'(\mathtt{p}) + \mathrm{rank}'(\mathtt{f}) + \mathrm{rank}'(\mathtt{g}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) - \mathrm{rank}(\mathtt{g}) \\
&= \mathrm{rank}'(\mathtt{f}) + \mathrm{rank}'(\mathtt{g}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) \\
&\le \mathrm{rank}'(\mathtt{f}) + \mathrm{rank}'(\mathtt{g}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) + 2\mathrm{rank}'(\mathtt{p}) - \mathrm{rank}'(\mathtt{f}) - \mathrm{rank}'(\mathtt{g}) - 2 \\
&(\forall a, b > 0 \Rightarrow \log_2(a) + \log_2(b) + 2 \le 2\log_2(a+b)) \\
&= 2\mathrm{rank}'(\mathtt{p}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) - 2 \\
&\le 2(\mathrm{rank}'(\mathtt{p}) - \mathrm{rank}(\mathtt{p})) - 2 \\
&\le 3(\mathrm{rank}'(\mathtt{p}) - \mathrm{rank}(\mathtt{p})) - 2
\end{aligned}$$

对于一次 `zig`，势能函数的变化为

$$\begin{aligned}
\Delta\Phi(T) &= \mathrm{rank}'(\mathtt{p}) + \mathrm{rank}'(\mathtt{f}) - \mathrm{rank}(\mathtt{p}) - \mathrm{rank}(\mathtt{f}) \\
&= \mathrm{rank}'(\mathtt{f}) - \mathrm{rank}(\mathtt{p}) \\
&\le \mathrm{rank}'(\mathtt{p}) - \mathrm{rank}(\mathtt{p}) \\
&\le 3(\mathrm{rank}'(\mathtt{p}) - \mathrm{rank}(\mathtt{p}))
\end{aligned}$$

所以，对于将节点 `p` `splay` 至根的过程，势能函数的总变化为

$$\Delta\Phi(T) \leq 3(\text{rank}(\texttt{root}) - \text{rank}(\texttt{p})) - 2(\texttt{zigzig,zigzag}\ \text{的总次数})$$

其中，$3(\text{rank}(\texttt{root}) - \text{rank}(\texttt{p})) = O(\text{rank}'(\texttt{root})) = O(\log n)$，而每多 `zigzig` 或 `zigzag` 一次，势能函数都会减小 2。所以，`zigzig` 与 `zigzag` 的总次数不会超过 $O(q \log n)$，而每次 `splay`，`zig` 最多调用一次，可忽略不计。

那么，`splay` 单次操作的复杂度在均摊意义下为 $O(\log n)$。

### 5.4.6 Splay 维护序列操作：BZOJ 1500 NOI 2005 维修数列

**题意**

给定一个长度为 $n$ 的序列 $a_1, a_2, \cdots, a_n$，进行 $m$ 次操作，每次操作为以下类型之一：

- `INSERT`: 给定 $posi, tot, c_1, c_2, \cdots, c_{tot}$，在当前数列的第 $posi$ 个数字后插入 $c_1, c_2, \cdots, c_{tot}$ 这 $tot$ 个数；
- `DELETE`: 给定 $posi, tot$，将当前序列的第 $posi, posi+1, \cdots, posi+tot-1$ 个数删除；
- `MAKE-SAME`: 给定 $posi, tot, c$，将当前序列的第 $posi, posi+1, \cdots, posi+tot-1$ 个数全部修改成 $c$；
- `REVERSE`: 给定 $posi, tot$，将当前序列的第 $posi, posi+1, \cdots, posi+tot-1$ 个数形成的子区间翻转（即顺序颠倒）；
- `GET-SUM`: 给定 $posi, tot$，求当前序列第 $posi, posi+1, \cdots, posi+tot-1$ 个数之和；
- `MAX-SUM`: 给定 $posi, tot$，求当前序列第 $posi, posi+1, \cdots, posi+tot-1$ 个数中，和最大的子区间的和。

**做法**

用 Splay 树来维护该序列，Splay 树上的中序遍历从左往右存储该序列。

修改、询问操作可以用和线段树类似的方式来实现。

插入、删除操作可以用 Splay 的插入、删除操作来实现。

翻转操作的实现方式是这样的：每当我们要让一个子树内的中序遍历整个翻转，就相当于是要让子数内所有节点的左右孩子互相交换。那么，我们可以在子树的根节点处打上一个“翻转”Tag，每当从上往下遇到这个 Tag，就交换两个孩子，并向下传递 Tag.

# 6 数据结构套数据结构

数据结构套数据结构，顾名思义就是在外层数据结构的每一个节点上面都存放一个内层数据结构。

## 6.1 UOJ218 [UNR #1] 火车管理

有 $n$ 个栈，分别为 $S_1, S_2, \ldots, S_n$，做 $m$ 次以下操作：

- 读入 `1 l r`，求出 $S_l, S_{l+1}, \ldots, S_r$ 这些栈的栈顶元素之和，这之中的空栈不计入答案。
- 读入 `2 l`，将 $S_l$ 的栈顶元素删去。
- 读入 `3 l r x`，在 $S_l, S_{l+1}, \ldots, S_r$ 这些栈的栈顶各插入一个元素 $x$。

数据范围：$1 \le n, m \le 5 \times 10^5$,$1 \le x \le 1000$，强制在线。

**做法一**

我们可以尝试维护一个序列 $a_1, a_2, \cdots, a_n$，每一个时刻，$a_i$ 的值等于 $S_i$ 的栈顶元素的值，若 $S_i$ 为空则 $a_i$ 为零。

那么，这三种操作能给 $a$ 序列带来怎样的变化呢？

- `1 l r`：输出 $a_l + a_{l+1} + \cdots + a_r$。
- `2 l`：将 $S_l$ 的栈顶元素删去后，将 $a_l$ 修改成新的栈顶元素，即删去前 $S_l$ 中第二个元素。
- `3 l r x`，将 $a_l, a_{l+1}, \cdots, a_r$ 全部修改成 $x$。

这样，只要我们能够在每一个 `2` 操作时知道 $S_l$ 的新的栈顶元素是什么，这个问题就是简单的线段树维护区间和问题。

我们可以建立一棵线段树，线段树上的每个节点都维护一个堆，堆维护了堆内**加入时间最晚**的元素。

当执行 `3` 操作时，我们将区间 $[l, r]$ 拆成 $O(\log n)$ 个线段树上的节点，在这 $O(\log n)$ 个节点的堆中都加入元素 $x$（以加入时间排序）。

当执行 `2` 操作时，我们找出线段树上所有包含了 $l$ 的 $O(\log n)$ 个节点（是一个叶子到根的路径），在这 $O(\log n)$ 个节点的堆的 $O(\log n)$ 个堆顶元素中找出插入时间最晚的（假设位于节点 $[L, R]$）。这个就是**删除堆顶前**的堆顶 $a_l$。

然后，我们把元素 $a_l$ 从节点 $[L, R]$ 所在的堆里删去，再把区间 $[L, l-1]$ 和区间 $[l+1, R]$ 拆成 $O(\log n)$ 个线段树上的节点，再把 $a_l$ 加入这 $O(\log n)$ 个节点的堆里。

执行完这些操作以后，我们再找一遍所有包含了 $l$ 的 $O(\log n)$ 个节点的堆的 $O(\log n)$ 个堆顶元素中插入时间最晚的，这就是新的堆顶 $a'_l$。

总时间复杂度 $O(m \log^2 n)$。

## 6.2 矩阵数点

有一个可重点集 $S$，一开始为空，接下来依次进行 $Q$ 个操作：

1. 往 $S$ 中加入一个给定的点 $(x, y)$。
2. 给定一个矩形 $([x_{min}, x_{max}], [y_{min}, y_{max}])$，问 $S$ 中有多少点位于这个矩形内部（边界、角上也算）。

数据范围：$1 \le Q \le 10^5$,$1 \le x, y \le 10^9$

**做法**

外层是一棵区间范围 $[1, 10^9]$ 的线段树，外层线段树的每个节点上都是一棵区间范围 $[1, 10^9]$ 的内层线段树。

注意，对于子树内所有元素都为 0 的节点，我们不要把它们建出来，不要让它们白白占用内存，这样内存复杂度和时间复杂度就都降到了 $O(Q \log^2(10^9))$。

# 7 可持久化数据结构

可持久化数据结构，即在支持传统的操作的基础上，额外增加一个操作：回到之前的第 $i$ 次操作后的状态。

如果原数据结构的时间复杂度依赖均摊分析，可持久化后均摊分析会失效。因为我们可以不断重复做一个最慢的操作，导致时间复杂度变大。

## 7.1 可持久化栈

维护一个栈，支持以下操作：

1. 在栈顶加入元素 $x$。
2. 删去并输出栈顶元素。
3. 将整个栈回到第 $i$ 次操作后的状态。

我们可以维护一棵有根树和一个指针。指针所在的位置是栈顶，指针到根路径上的各个元素是当前栈中所有元素。

插入时，我们给指针所在的点新建一个儿子，并将指针移到这个儿子上。

删除时，将指针向父亲移动。

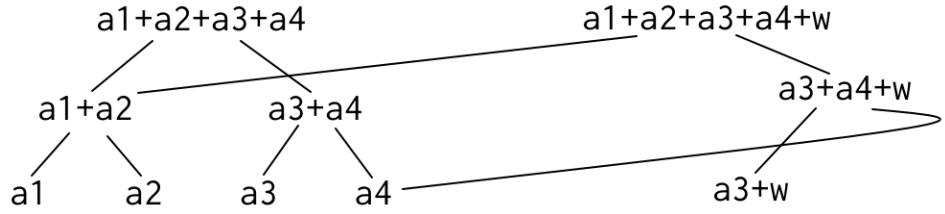
回到第 $i$ 次操作后的状态时，将指针移到第 $i$ 个时刻所在的位置。

## 7.2 可持久化线段树

可持久化数据结构，最重要的问题就是内存问题。在可持久化栈中，每次修改只会让一个节点发生变化，所以我们可以很轻松地将内存从 $O(n^2)$ 压缩至 $O(n)$。

如果我们要将线段树可持久化，最重要的问题依然是内存问题。线段树上的每次修改都会让 $O(\log n)$ 个节点发生变化，我们可不可以每次只将这 $O(\log n)$ 个节点存下来，以达到 $O(q \log n)$ 的内存复杂度?

例如，$n = 4$，我们要在 $i = 3$ 处作单点 $+w$ 的操作：



我们发现，每次修改只需要新建 $O(\log n)$ 个节点，其余节点只需要把孩子指针指过去，不必整个复制。

把每一时刻的根节点全部记下来，查询时，从对应的根节点开始，像一般的线段树一样向下查询即可。

时间复杂度和内存复杂度都是 $O(q \log n)$。

## 7.3 区间计数

有一个长度为 $n$ 的序列 $a_1, a_2, \ldots, a_n$，有 $m$ 次询问，每次给定 $l, r, min, max$，求 $a_l, a_{l+1}, \ldots, a_r$ 中有多少数介于 $[min, max]$。

数据范围：$1 \le n \le 10^5, 1 \le a_i \le 10^9$

## 7.4 POJ2104 区间第 k 大

有一个长度为 $n$ 的序列 $a_1, a_2, \ldots, a_n$，有 $m$ 次询问，每次给定 $l, r, k$，求 $a_l, a_{l+1}, \ldots, a_r$ 中第 $k$ 大的数。

数据范围：$1 \le n \le 10^5, 1 \le a_i \le 10^9$

### 做法

假如对于所有询问都有 $l = 1, r = n$，显然可以在由所有数构成的线段树上二分。

现在我们从左到右依次在线段树的 $a_1, a_2, \ldots, a_n$ 位置做单点修改。询问时，在 $r$ 时刻版本和 $l-1$ 时刻版本的线段树的差上做二分。

（线段树的差：对应位置的节点上的值相减）

## 7.5 UOJ218 [UNR #1] 火车管理

有 $n$ 个栈，分别为 $S_1, S_2, \ldots, S_n$，做 $m$ 次以下操作：

- 读入 `1 l r`，求出 $S_l, S_{l+1}, \ldots, S_r$ 这些栈的栈顶元素之和，这之中的空栈不计入答案。
- 读入 `2 l`，将 $S_l$ 的栈顶元素删去。
- 读入 `3 l r x`，在 $S_l, S_{l+1}, \ldots, S_r$ 这些栈的栈顶各插入一个元素 $x$。

数据范围：$1 \le n, m \le 5 \times 10^5$,$1 \le x \le 1000$，强制在线。

### 做法二

和之前的做法一一样，关键都是如何求出，将一个栈顶元素弹出以后，这个栈的下一个栈顶元素是多少。我们现在要用可持久化线段树来做这个事情。

在 $i$ 时刻，设 $S_l$ 的栈顶元素是 $t(a_l)$ 时刻加入的 $a_l$，那么，当我们把 $a_l$ 删去后，$S_l$ 中的下一个栈顶元素是什么？

是 $t(a_l) - 1$ 时刻 $S_l$ 的栈顶。

所以，我们只要把维护答案数组 $a_1, a_2, \cdots, a_n$ 的线段树可持久化，即可求出任一时刻的任何一个栈的栈顶元素。

时间复杂度 $O(m \log n)$。