分块　线段树　树状数组

July 17, 2018, riteme

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

操作有以下四种:

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

操作有以下四种:

1. 给定 k 和整数 v，将 $A[k]$ 加上 v。注意 v 允许为负数。——**单点修改**

7	1	1	1	8	6	2	5
+5							

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

操作有以下四种：

1. 给定 k 和整数 v，将 $A[k]$ 加上 v。注意 v 允许为负数。——单点修改

7	1	1	1	8	6	2	5

+5

2. 给定区间 $[l, r]$ 和整数 v，将 $A[l..r]$ 都加上 v。——区间修改

7	1	3	3	10	6	2	5

+2

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

操作有以下四种:

1. 给定 k 和整数 v，将 $A[k]$ 加上 v。注意 v 允许为负数。——**单点修改**

7	1	1	1	8	6	2	5

+5

2. 给定区间 $[l, r]$ 和整数 v，将 $A[l..r]$ 都加上 v。——**区间修改**

7	1	3	3	10	6	2	5

+2

3. 给定 k，输出 $A[k]$。——**单点查询**

7	1	3	3	10	6	2	5

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

操作有以下四种:

1. 给定 k 和整数 v，将 $A[k]$ 加上 v。注意 v 允许为负数。——单点修改

7	1	1	1	8	6	2	5

+5

2. 给定区间 $[l, r]$ 和整数 v，将 $A[l..r]$ 都加上 v。——区间修改

7	1	3	3	10	6	2	5

+2

3. 给定 k，输出 $A[k]$。——单点查询

7	1	3	3	10	6	2	5

4. 给定区间 $[l, r]$，输出 $A[l..r]$ 内所有元素的和。——区间查询

7	1	3	3	10	6	2	5

Total: 21

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

操作有以下四种:

1. 给定 k 和整数 v，将 $A[k]$ 加上 v。注意 v 允许为负数。——**单点修改**

7	1	1	1	8	6	2	5

+5

2. 给定区间 $[l, r]$ 和整数 v，将 $A[l..r]$ 都加上 v。——**区间修改**

7	1	3	3	10	6	2	5

+2

3. 给定 k，输出 $A[k]$。——**单点查询**

7	1	3	3	10	6	2	5

4. 给定区间 $[l, r]$，输出 $A[l..r]$ 内所有元素的和。——**区间查询**

7	1	3	3	10	6	2	5

Total: 21

单点操作是区间操作的特例。

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

操作有以下四种：

1. 给定 k 和整数 v，将 $A[k]$ 加上 v。注意 v 允许为负数。——**单点修改**

7	1	1	1	8	6	2	5

+5

2. 给定区间 $[l, r]$ 和整数 v，将 $A[l..r]$ 都加上 v。——**区间修改**

7	1	3	3	10	6	2	5

+2

3. 给定 k，输出 $A[k]$。——**单点查询**

7	1	3	3	10	6	2	5

4. 给定区间 $[l, r]$，输出 $A[l..r]$ 内所有元素的和。——**区间查询**

7	1	3	3	10	6	2	5

Total: 21

单点操作是区间操作的特例。

一般情况下，$n, q \leqslant 10^5$。

一个经典数据结构题

初始给定一个长度为 n 的整数序列 $A[1..n]$，执行 q 次操作。

2	1	1	1	8	6	2	5

操作有以下四种:

1. 给定 k 和整数 v，将 $A[k]$ 加上 v。注意 v 允许为负数。——**单点修改**

7	1	1	1	8	6	2	5

+5

2. 给定区间 $[l, r]$ 和整数 v，将 $A[l..r]$ 都加上 v。——**区间修改**

7	1	3	3	10	6	2	5

+2

3. 给定 k，输出 $A[k]$。——**单点查询**

7	1	3	3	10	6	2	5

4. 给定区间 $[l, r]$，输出 $A[l..r]$ 内所有元素的和。——**区间查询**

7	1	3	3	10	6	2	5

Total: 21

单点操作是区间操作的特例。

一般情况下，$n, q \leqslant 10^5$。

模板题：【LG P3372】

朴素算法

朴素算法主要慢在所有的区间操作上。区间越长，朴素模拟就越慢。

朴素算法

朴素算法主要慢在所有的区间操作上。区间越长，朴素模拟就越慢。
将大区间 “缩小” 是优化的主要方向。

分块

将序列切成一块一块的，更形式化地讲，每 S 个元素分在一起，最后不足 S 个的也分在一起。

分块

将序列切成一块一块的，更形式化地讲，每 S 个元素分在一起，最后不足 S 个的也分在一起。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

分块

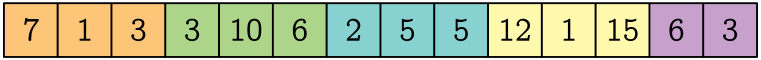
将序列切成一块一块的，更形式化地讲，每 S 个元素分在一起，最后不足 S 个的也分在一起。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

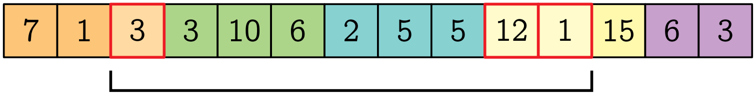
针对区间修改，当 S 比较小时，区间很长的操作就可以被分成一块一块的。一整块同时加一个数字可以不用一个一个元素地修改，而是为这一块记录一个 `offest` 值，表示整体的增加量。单点查询时加上 `offest` 的影响即可。

分块

将序列切成一块一块的，更形式化地讲，每 S 个元素分在一起，最后不足 S 个的也分在一起。



针对区间修改，当 S 比较小时，区间很长的操作就可以被分成一块一块的。一整块同时加一个数字可以不用一个一个元素地修改，而是为这一块记录一个 `offest` 值，表示整体的增加量。单点查询时加上 `offest` 的影响即可。



分块

将序列切成一块一块的，更形式化地讲，每 S 个元素分在一起，最后不足 S 个的也分在一起。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

针对区间修改，当 S 比较小时，区间很长的操作就可以被分成一块一块的。一整块同时加一个数字可以不用一个一个元素地修改，而是为这一块记录一个 `offest` 值，表示整体的增加量。单点查询时加上 `offest` 的影响即可。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

当然，之前还忽略了一些东西。区间的两端不一定就在块与块的边界上。

分块

将序列切成一块一块的，更形式化地讲，每 S 个元素分在一起，最后不足 S 个的也分在一起。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

针对区间修改，当 S 比较小时，区间很长的操作就可以被分成一块一块的。一整块同时加一个数字可以不用一个一个元素地修改，而是为这一块记录一个 `offest` 值，表示整体的增加量。单点查询时加上 `offest` 的影响即可。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

当然，之前还忽略了一些东西。区间的两端不一定就在块与块的边界上。
这时候没有什么太多的办法，只能单个修改。单个修改的时候 `offest` 可以不改动，保持其整体增量的意义。或者是加到原序列上，然后清空 `offest`。

分块

将序列切成一块一块的，更形式化地讲，每 S 个元素分在一起，最后不足 S 个的也分在一起。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

针对区间修改，当 S 比较小时，区间很长的操作就可以被分成一块一块的。一整块同时加一个数字可以不用一个一个元素地修改，而是为这一块记录一个 `offest` 值，表示整体的增加量。单点查询时加上 `offest` 的影响即可。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

当然，之前还忽略了一些东西。区间的两端不一定就在块与块的边界上。
这时候没有什么太多的办法，只能单个修改。单个修改的时候 `offest` 可以不改动，保持其整体增量的意义。或者是加到原序列上，然后清空 `offest`。
现在我们顺利实现了单点查询和区间修改。区间查询和区间修改类似，为了能够整块查询，我们需要为每个块再记录一个 `sum` 表示这个块内的总和。如果块的 `offest` 不为 0 ，则实际的总和需要加上 `num * offest`，这里 `num` 是块中元素个数。

分块

将序列切成一块一块的，更形式化地讲，每 S 个元素分在一起，最后不足 S 个的也分在一起。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

针对区间修改，当 S 比较小时，区间很长的操作就可以被分成一块一块的。一整块同时加一个数字可以不用一个一个元素地修改，而是为这一块记录一个 `offest` 值，表示整体的增加量。单点查询时加上 `offest` 的影响即可。

7	1	3	3	10	6	2	5	5	12	1	15	6	3

当然，之前还忽略了一些东西。区间的两端不一定就在块与块的边界上。

这时候没有什么太多的办法，只能单个修改。单个修改的时候 `offest` 可以不改动，保持其整体增量的意义。或者是加到原序列上，然后清空 `offest`。

现在我们顺利实现了单点查询和区间修改。区间查询和区间修改类似，为了能够整块查询，我们需要为每个块再记录一个 `sum` 表示这个块内的总和。如果块的 `offest` 不为 0，则实际的总和需要加上 `num * offest`，这里 `num` 是块中元素个数。

注意查询的区间在单独的一个块内的特殊情况。

分块：参考实现

```
BLOCK_ID(k): (k - 1) / S   // 块的编号，从 0 开始
L(k): S * k + 1            // 块的左边界
R(k): min(S * (k + 1), n)  // 块的右边界
NUM(k): R(k) - L(k) + 1    // 块中元素个数。除了最后一块其余都是 S
int offest[], sum[]
function initialize():  // 初始化
    for i in [1..n]:
        sum[BLOCK_ID(i)] += A[i]
function modify(int l, int r, int v):  // 区间 [l, r] 加上 v
    for i in [BLOCK_ID(l), BLOCK_ID(r)]:
        if l <= L(i) and R(i) <= r:  // 如果被整块包含
            offest[i] += v
        else:
            for j in [max(l, L(i)), min(r, R(i))]:
                sum[i], A[j] += v
function query(int l, int r):  // 查询区间 [l, r]
    int ret = 0
    for i in [BLOCK_ID(l), BLOCK_ID(r)]:
        if l <= L(i) and R(i) <= r:  // 如果被整块包含
            ret += sum[i] + NUM(i) * offest[i]
        else:
            for j in [max(l, L(i)), min(r, R(i))]:
                ret += A[j] + offest[i]
    return ret
```

分块：时间复杂度

每次最多有两个块是块内单独修改，这样的修改最多进行 $2S$ 次。

分块：时间复杂度

每次最多有两个块是块内单独修改，这样的修改最多进行 $2S$ 次。

序列被分为 $\lceil n/S \rceil$ 块，这也是整块操作的最大次数。

分块：时间复杂度

每次最多有两个块是块内单独修改，这样的修改最多进行 $2S$ 次。

序列被分为 $\lceil n/S \rceil$ 块，这也是整块操作的最大次数。

S 完全由我们决定，注意到随着 S 增大，一个代价在上升，而另一个在下降。

分块：时间复杂度

每次最多有两个块是块内单独修改，这样的修改最多进行 $2S$ 次。

序列被分为 $\lceil n/S \rceil$ 块，这也是整块操作的最大次数。

S 完全由我们决定，注意到随着 S 增大，一个代价在上升，而另一个在下降。

考虑总代价 $2S + n/S$，最小化该函数从而达到最优性能。

分块：时间复杂度

每次最多有两个块是块内单独修改，这样的修改最多进行 $2S$ 次。

序列被分为 $\lceil n/S \rceil$ 块，这也是整块操作的最大次数。

S 完全由我们决定，注意到随着 S 增大，一个代价在上升，而另一个在下降。

考虑总代价 $2S + n/S$，最小化该函数从而达到最优性能。

根据均值不等式:

$$2S + n/S \geqslant \sqrt{2n} \quad \text{iff. } S = \sqrt{n/2}$$

分块：时间复杂度

每次最多有两个块是块内单独修改，这样的修改最多进行 $2S$ 次。

序列被分为 $\lceil n/S \rceil$ 块，这也是整块操作的最大次数。

S 完全由我们决定，注意到随着 S 增大，一个代价在上升，而另一个在下降。

考虑总代价 $2S + n/S$，最小化该函数从而达到最优性能。

根据均值不等式:

$$2S + n/S \geqslant \sqrt{2n} \quad \text{iff. } S = \sqrt{n/2}$$

所以 S 大约取 $\sqrt{n/2}$ 附近的值即可达到单次操作 $O(\sqrt{n})$ 的时间复杂度。

分块：时间复杂度

每次最多有两个块是块内单独修改，这样的修改最多进行 $2S$ 次。

序列被分为 $\lceil n/S \rceil$ 块，这也是整块操作的最大次数。

S 完全由我们决定，注意到随着 S 增大，一个代价在上升，而另一个在下降。

考虑总代价 $2S + n/S$，最小化该函数从而达到最优性能。

根据均值不等式：

$$2S + n/S \geqslant \sqrt{2n} \quad \text{iff. } S = \sqrt{n/2}$$

所以 S 大约取 $\sqrt{n/2}$ 附近的值即可达到单次操作 $O(\sqrt{n})$ 的时间复杂度。

当然直接从渐进上讲会简单的多：总复杂度为 $O(S + n/S)$，关键在于最小化 $\max\{S, n/S\}$，这个东西只有在 $S = n/S$ 即 $S = \sqrt{n}$ 时取最小值。

分块：时间复杂度

每次最多有两个块是块内单独修改，这样的修改最多进行 $2S$ 次。

序列被分为 $\lceil n/S \rceil$ 块，这也是整块操作的最大次数。

S 完全由我们决定，注意到随着 S 增大，一个代价在上升，而另一个在下降。

考虑总代价 $2S + n/S$，最小化该函数从而达到最优性能。

根据均值不等式:

$$2S + n/S \geqslant \sqrt{2n} \quad \text{iff. } S = \sqrt{n/2}$$

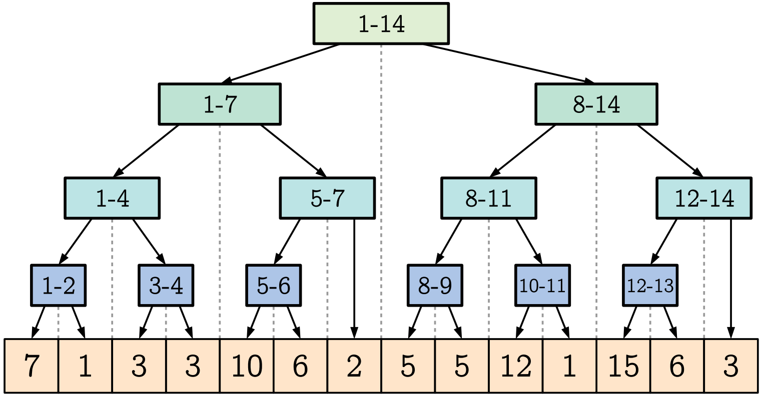
所以 S 大约取 $\sqrt{n/2}$ 附近的值即可达到单次操作 $O(\sqrt{n})$ 的时间复杂度。

当然直接从渐进上讲会简单的多：总复杂度为 $O(S + n/S)$，关键在于最小化 $\max\{S, n/S\}$，这个东西只有在 $S = n/S$ 即 $S = \sqrt{n}$ 时取最小值。

无论哪种分析都只是理论上的最优值，它们没有考虑实际常数的影响，只具有指导意义。想真正让程序最优化，还需要自己动手试一试不同的 S 值。

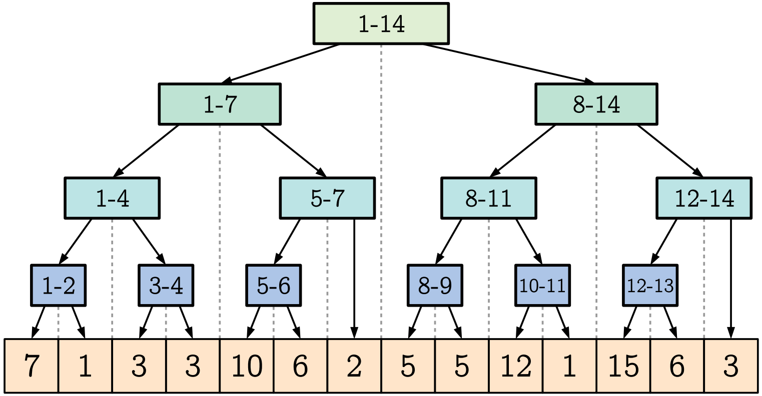
线段树

线段树



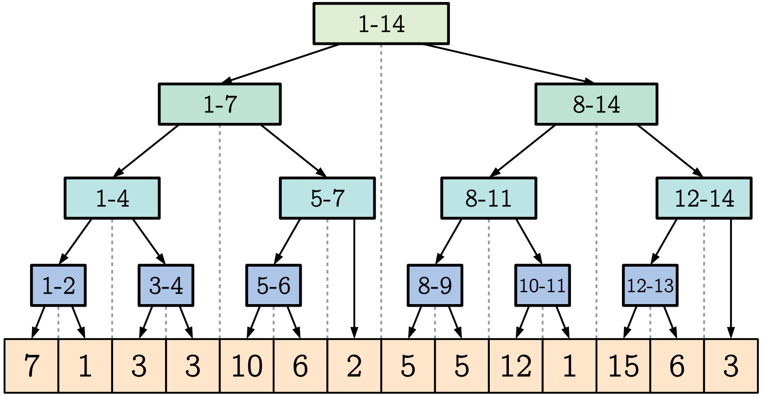
1-14
1-7
8-14
1-4
5-7
8-11
12-14
1-2
3-4
5-6
8-9
10-11
12-13
7
1
3
3
10
6
2
5
5
12
1
15
6
3

线段树



线段树实际上是分治的产物：将当前序列切为两半，分别递归下去处理。

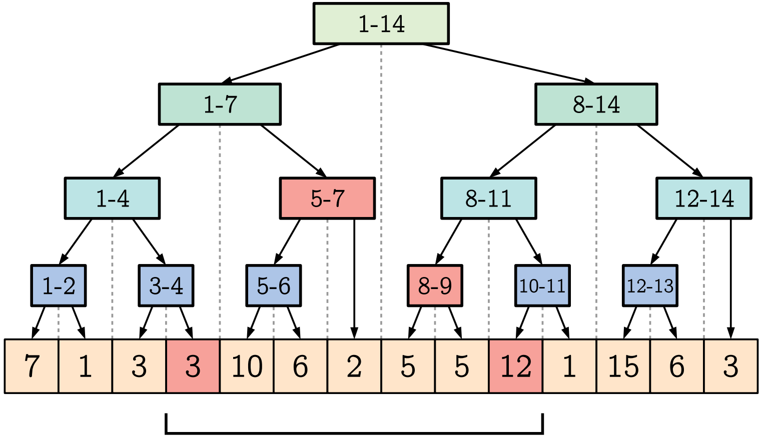
线段树



线段树实际上是分治的产物：将当前序列切为两半，分别递归下去处理。

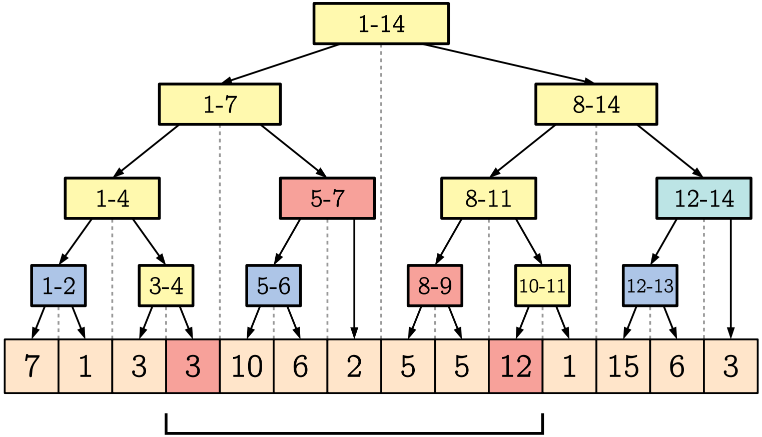
除了叶节点，线段树上其余节点都表示一个区间。像之前在分块中那样，这样的节点可以储存对整体的操作。

线段树



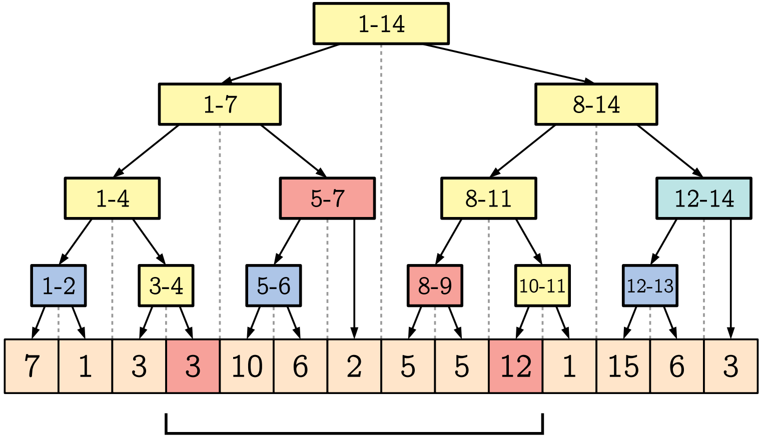
对于任意一个区间，都可以被拆分成线段树上一些节点。（上图红色节点）

线段树



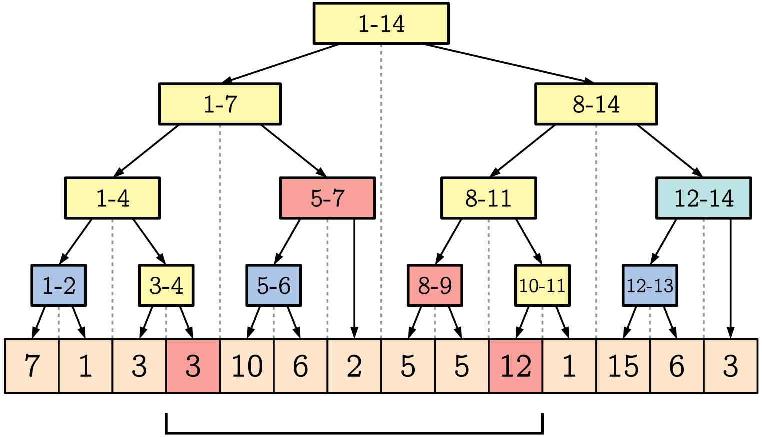
此外，我们的处理是从根节点开始的，因此还要多遍历一些其它的节点，才能找到我们想要的红色节点。（上图黄色节点）

线段树：区间操作



在每个节点处存储对应区间内所有元素的和 sum，如果是区间查询，将上图中所有红色节点的 sum 加起来就是答案。

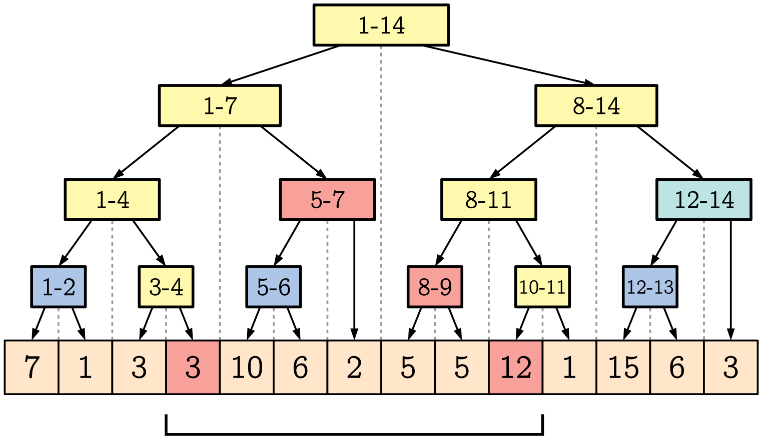
线段树：区间操作



在每个节点处存储对应区间内所有元素的和 sum，如果是区间查询，将上图中所有红色节点的 sum 加起来就是答案。

对于区间修改，像在分块中的处理方法类似，每个节点记录一个整体增量 offest，表示对应区间内的元素全部增加了 offest。

线段树：区间操作



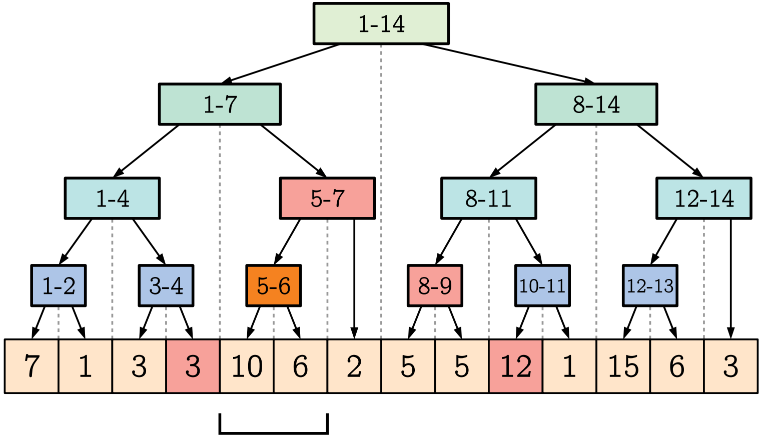
在每个节点处存储对应区间内所有元素的和 sum，如果是区间查询，将上图中所有红色节点的 sum 加起来就是答案。

对于区间修改，像在分块中的处理方法类似，每个节点记录一个整体增量 offest，表示对应区间内的元素全部增加了 offest。

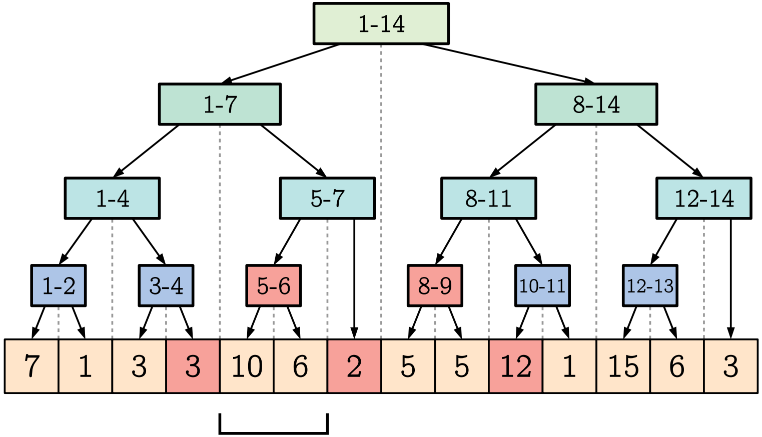
特殊的是，除了要更新红色节点的 sum 之外，不难发现黄色节点的 sum 也需要更新，因为它们所对应的区间包含了某些红色节点。

线段树：区间操作

线段树：区间操作

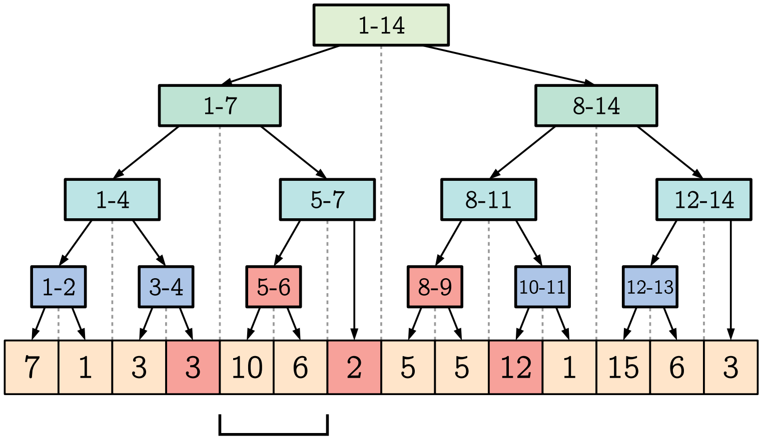


然而还有一个问题，如上图所示，经过之前的区间修改之后，如果查询 $[5, 6]$，则还需要将其到父节点到根节点上所有的 `offest` 值都计入。



此外，我们其实可以将 `offest` 视为一种标记。当需要进入更深的节点时，将标记下传。如上图所示，即将 $[5, 7]$ 的 `offest` 分别加到 $[5, 6]$ 和 7 的 `offest` 上，然后将 $[5, 7]$ 的 `offest` 清空。

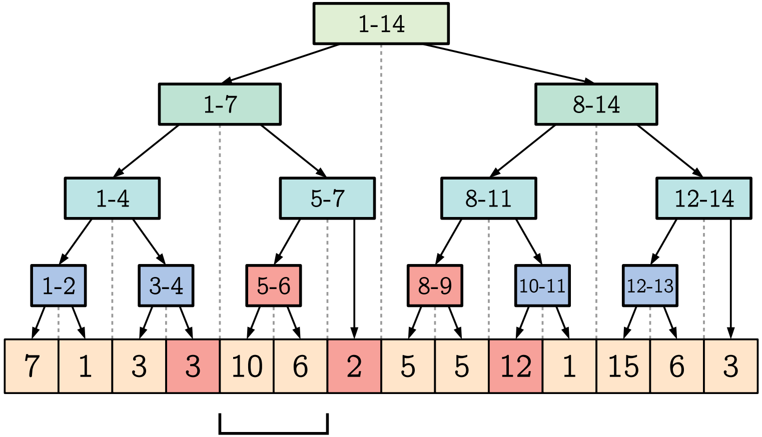
线段树：区间操作



此外，我们其实可以将 `offest` 视为一种标记。当需要进入更深的节点时，将标记下传。如上图所示，即将 $[5, 7]$ 的 `offest` 分别加到 $[5, 6]$ 和 7 的 `offest` 上，然后将 $[5, 7]$ 的 `offest` 清空。

这种做法也被称作 “Lazy 标记”。

线段树：区间操作



此外，我们其实可以将 `offest` 视为一种标记。当需要进入更深的节点时，将标记下传。如上图所示，即将 $[5, 7]$ 的 `offest` 分别加到 $[5, 6]$ 和 7 的 `offest` 上，然后将 $[5, 7]$ 的 `offest` 清空。

这种做法也被称作 “Lazy 标记”。

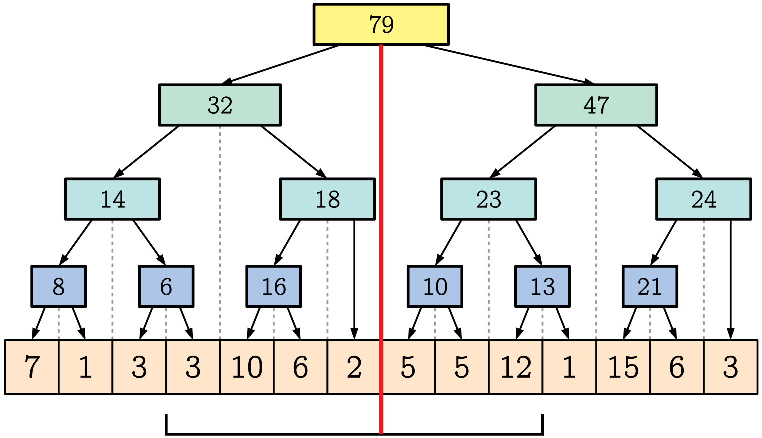
总而言之，必须时刻保持每个节点的 `sum` 值真实有效。

线段树：示例

区间查询操作：（询问区间为 $[4, 10]$）

线段树：示例

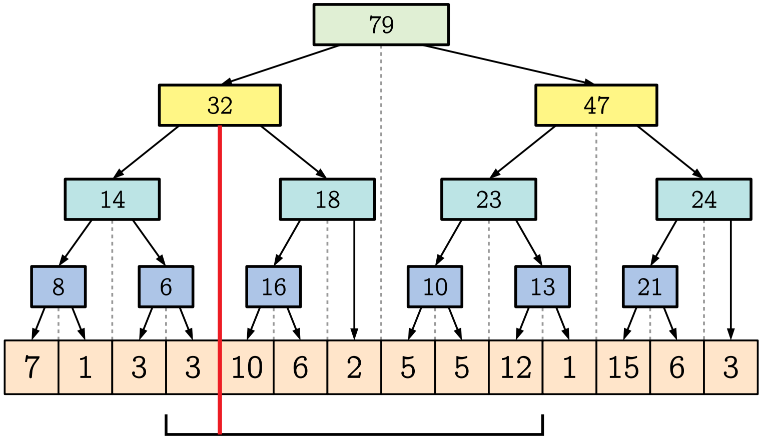
区间查询操作：（询问区间为 $[4, 10]$）



初始在根节点，询问区间横跨中线，说明中线左右都要有更小的节点来划分询问区间，因此将先后递归至左右儿子处。

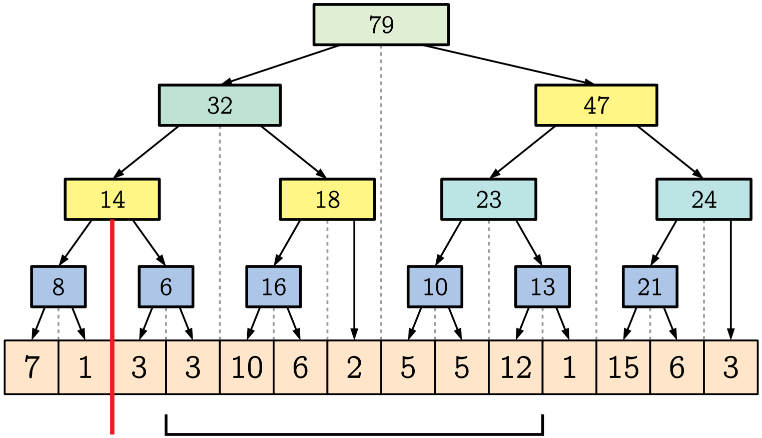
线段树：示例

区间查询操作：（询问区间为 $[4, 10]$）



同理，继续递归至左右儿子处。

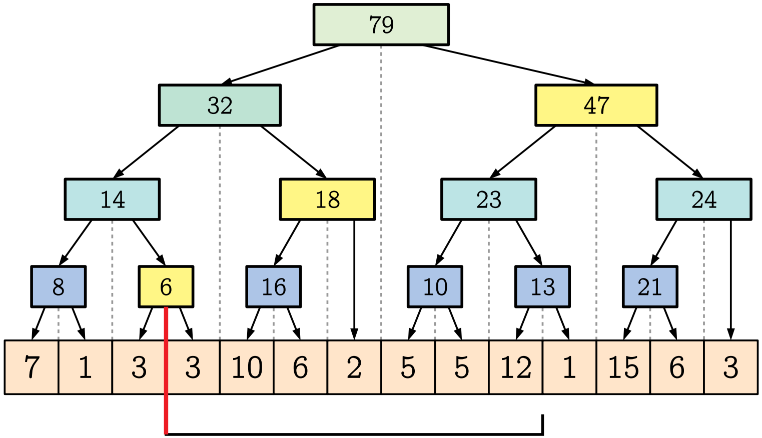
区间查询操作：（询问区间为 $[4,\ 10]$）



询问区间在中线右边，说明与中线左边的节点无关，因此只递归至右儿子处。

线段树：示例

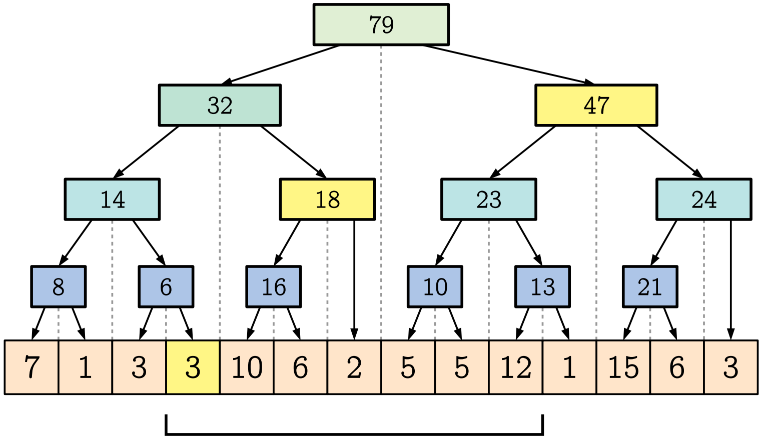
区间查询操作：（询问区间为 $[4, 10]$）



同上。

线段树：示例

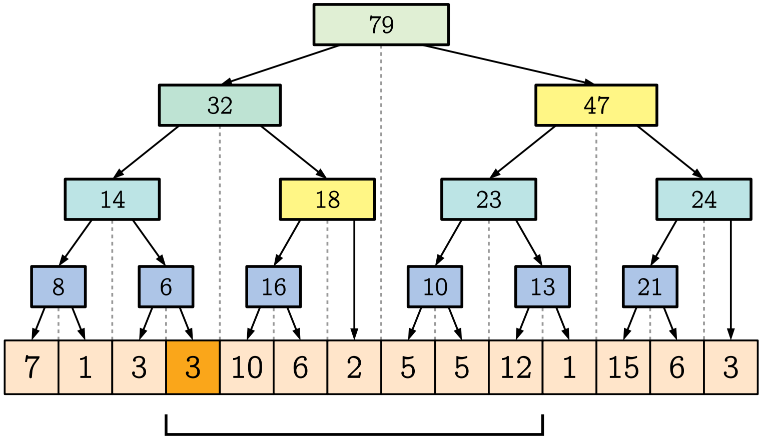
区间查询操作：（询问区间为 $[4, 10]$）



终于出现了被完全包含在询问区间内的节点。

线段树：示例

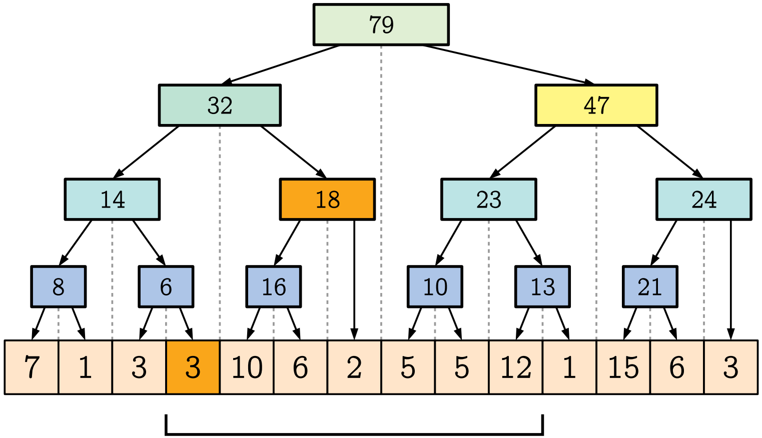
区间查询操作：（询问区间为 $[4, 10]$）



终于出现了被完全包含在询问区间内的节点。

线段树：示例

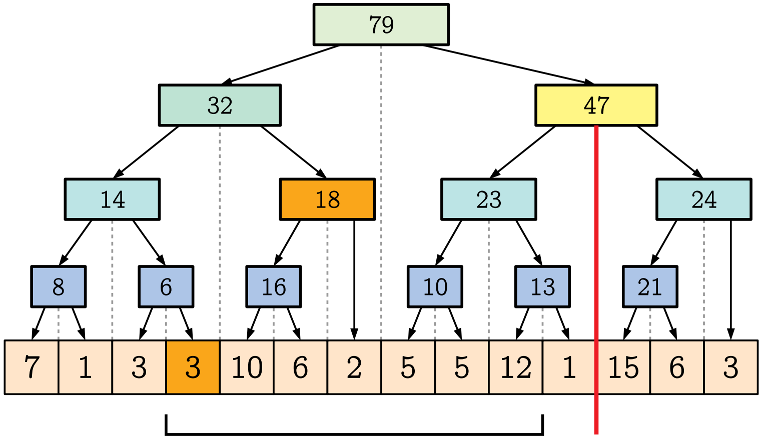
区间查询操作：（询问区间为 $[4, 10]$）



回溯后发现另一个被完全包含的节点。

线段树：示例

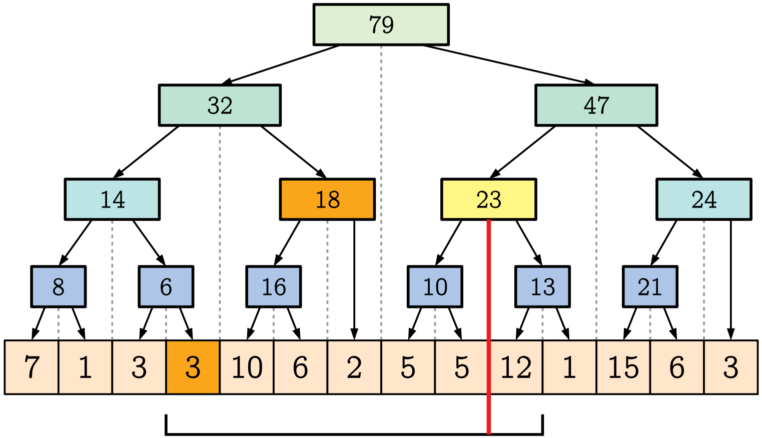
区间查询操作：（询问区间为 $[4, 10]$）



回溯至根节点后递归至右儿子。

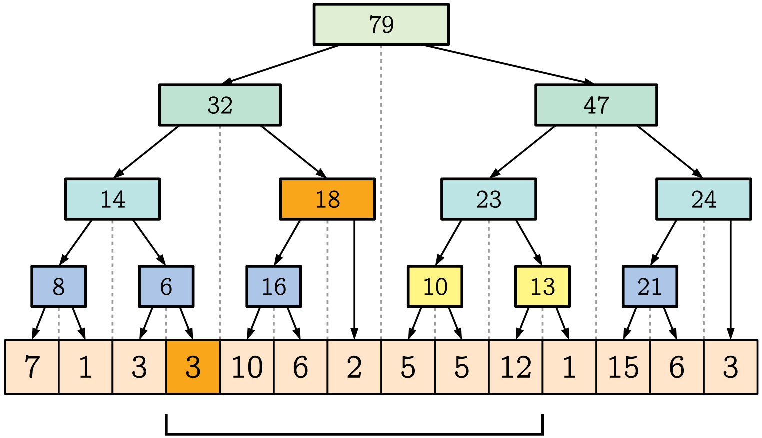
线段树：示例

区间查询操作：（询问区间为 $[4, 10]$）



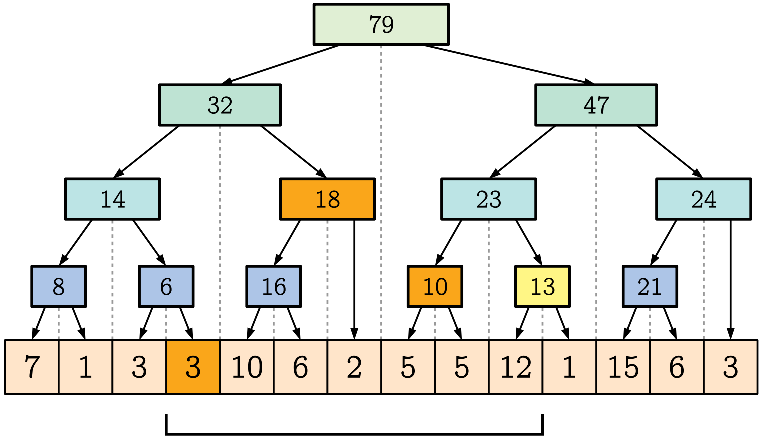
线段树：示例

区间查询操作：（询问区间为 $[4, 10]$）



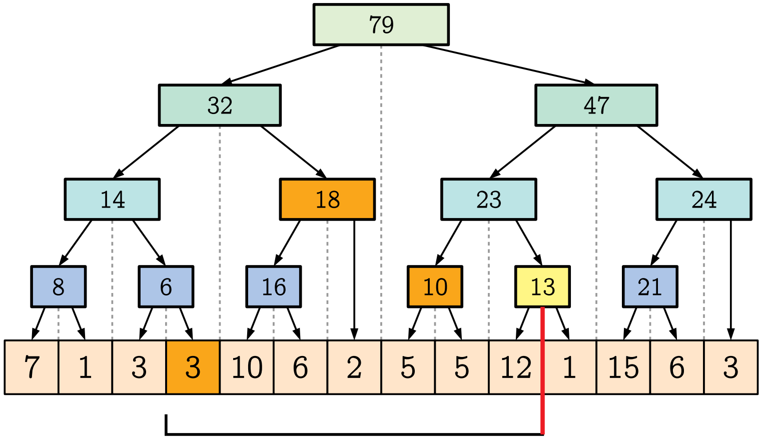
线段树：示例

区间查询操作：（询问区间为 $[4, 10]$）



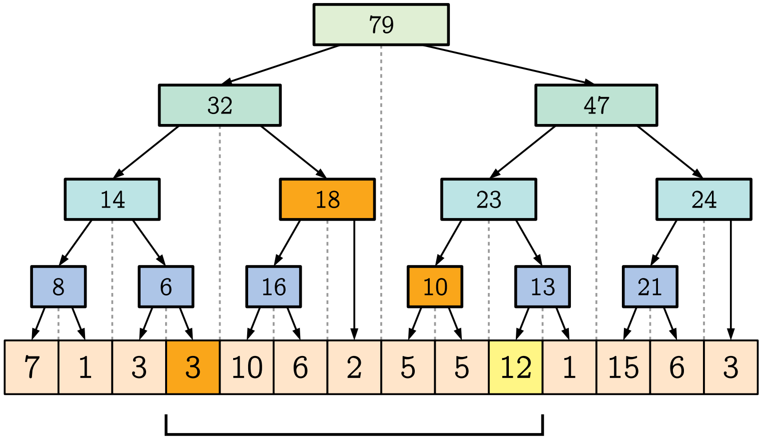
线段树：示例

区间查询操作：（询问区间为 $[4,\ 10]$）



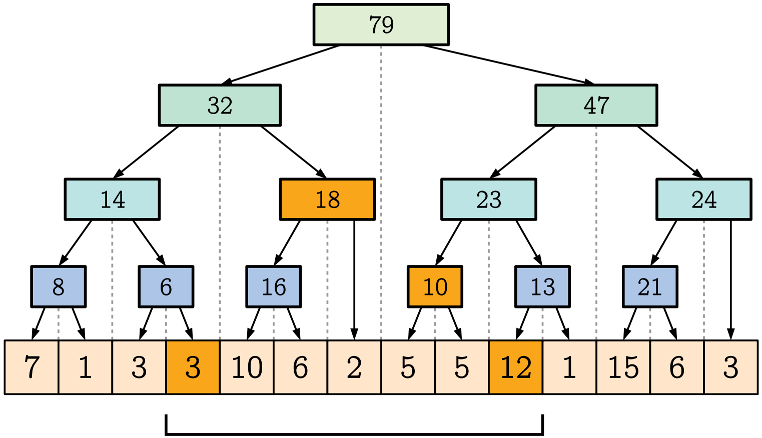
线段树：示例

区间查询操作：（询问区间为 $[4, 10]$）



线段树：示例

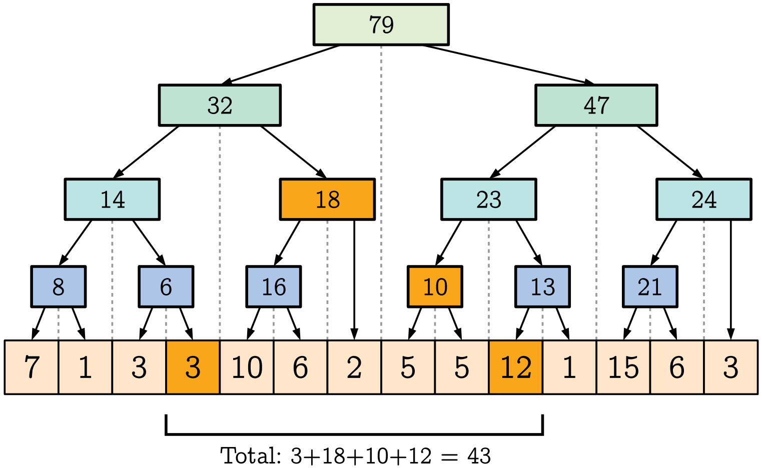
区间查询操作：（询问区间为 $[4, 10]$）



所有节点均被找到。

线段树：示例

区间查询操作：（询问区间为 $[4, 10]$）



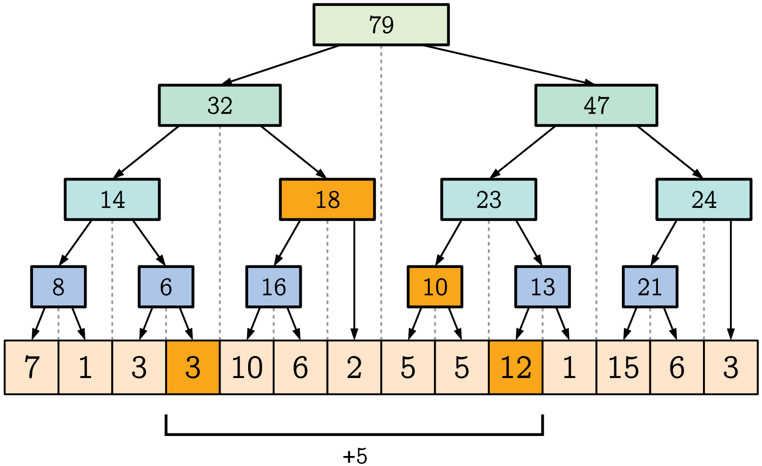
（最后的总和实际上是回溯的过程中处理的）

线段树：示例

区间修改操作：

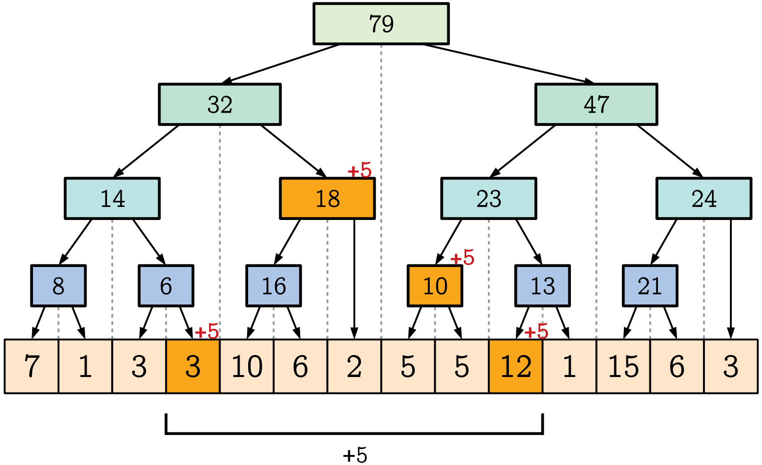
线段树：示例

区间修改操作：



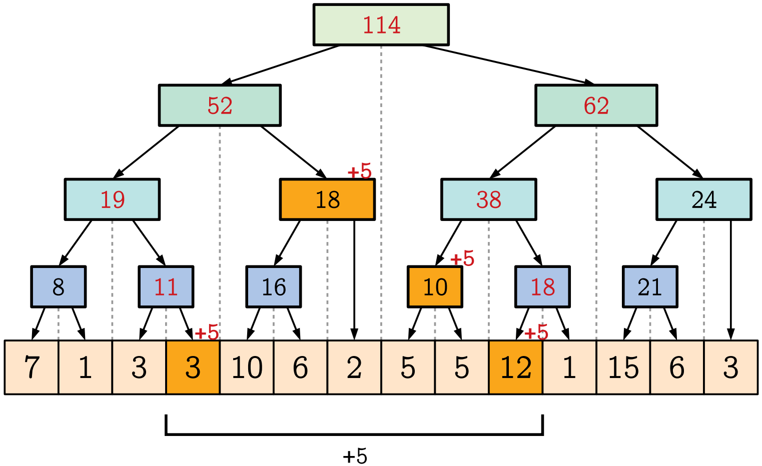
线段树：示例

区间修改操作：



线段树：示例

区间修改操作:



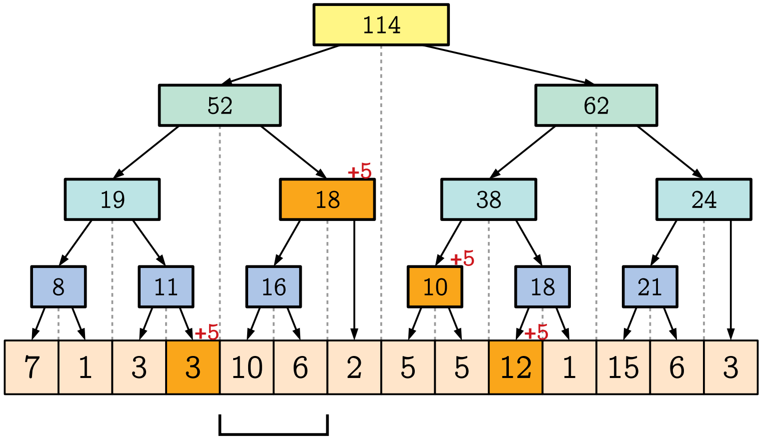
在回溯的时候更新节点的 `sum`（将左右儿子区间内总和加起来）。

线段树：示例

标记下传：

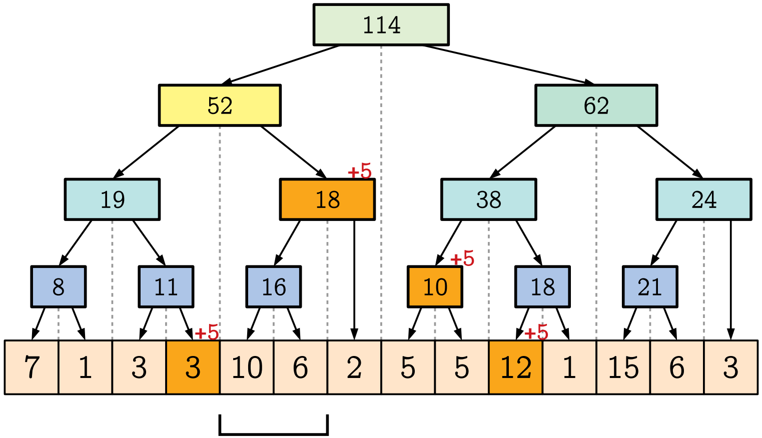
线段树：示例

标记下传:



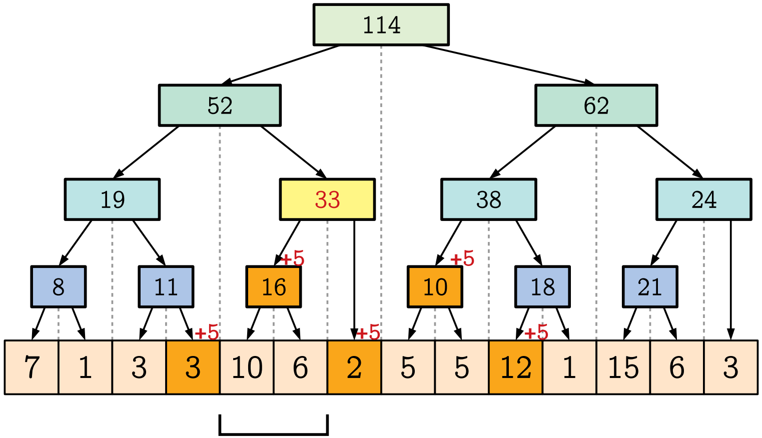
线段树：示例

标记下传:



线段树：示例

标记下传:



记得标记下传完毕后要更新 `sum`。

线段树：实现

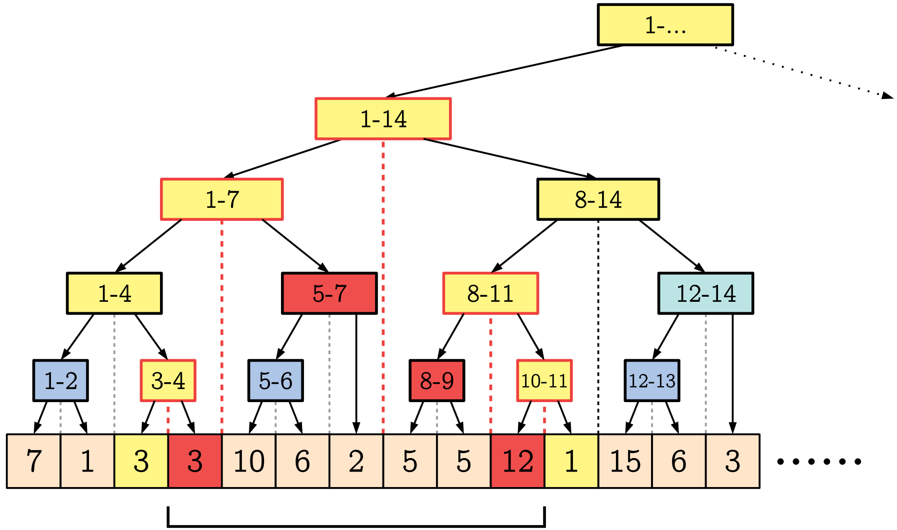
```
struct Node:
    int l, r, sum, offest  // 所管辖的区间、区间内总和、offest 标记
    Node *lch, *rch        // 左右儿子
    real_sum: sum + (r - l + 1) * offest  // 经过 offest 修正后的总和
function build(l, r):  // 建树
    if l == r: return Node(sum = A[l])  // 叶子节点
    int m = (l + r) / 2  // 中线
    x = Node(l, r)
    x->lch = build(l, m)
    x->rch = build(m + 1, r)
    x->sum = x->lch->sum + x->rch->sum
    return x
function modify(x, l, r, v):
    if l <= x->l and x->r <= r:  // 节点被操作区间覆盖
        x->offest += v
    else:
        x->lch->offest, x->rch->offest += x->offest  // 标记下传
        x->offest = 0
        int m = (x->l + x->r) / 2  // 节点 x 的中线
        if l <= m:  // 中线左边有操作区间的一部分
            modify(x->lch, l, r, v)
        if r > m:
            modify(x->rch, l, r, v)
        x->sum = x->lch->real_sum + x->rch->real_sum  // 更新 sum
```

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类:

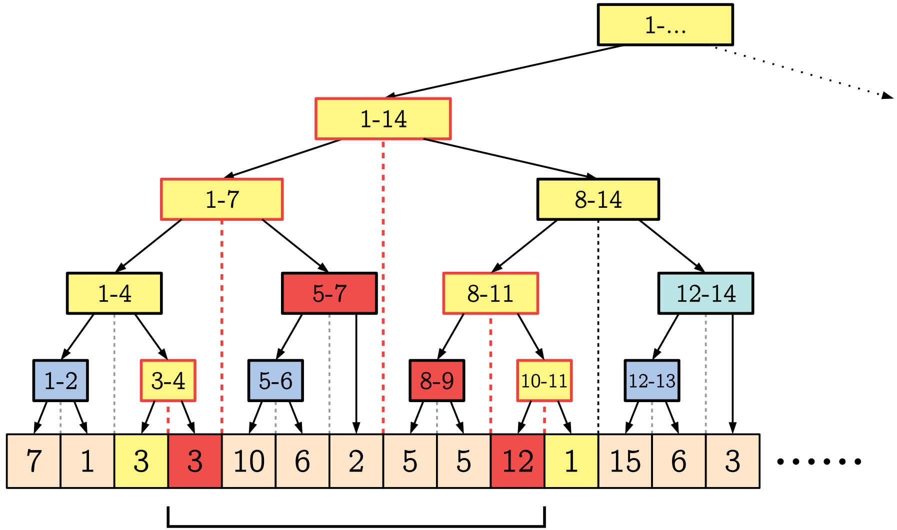
线段树：时间复杂度

每次区间操作访问过的节点主要分为两类:



线段树：时间复杂度

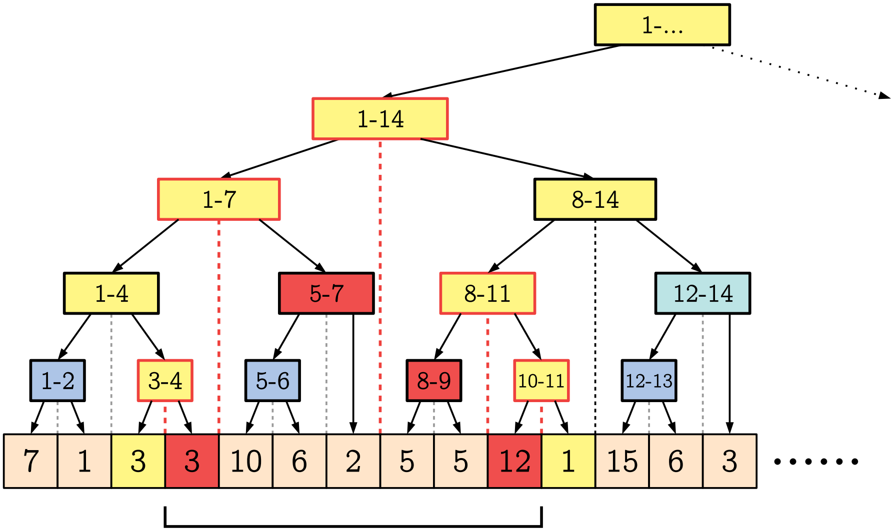
每次区间操作访问过的节点主要分为两类：



1. 被操作区间完全覆盖的节点（上图红色节点）。

线段树：时间复杂度

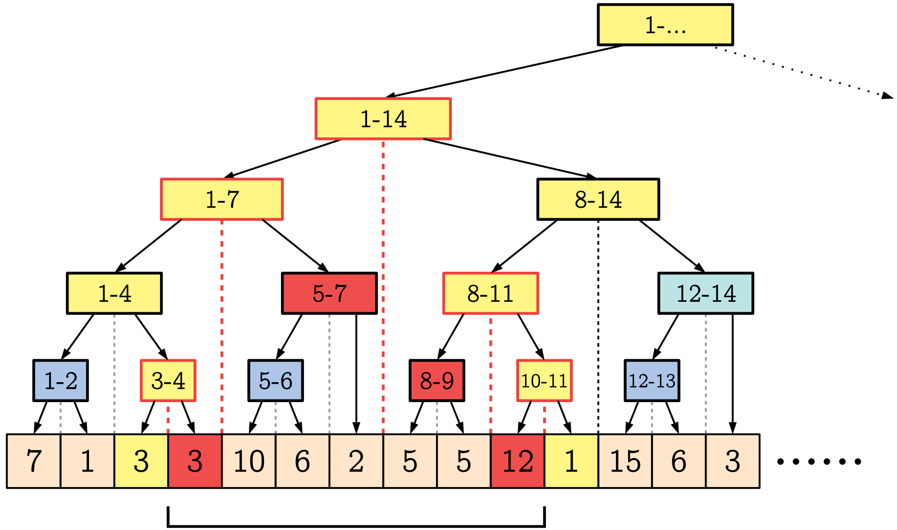
每次区间操作访问过的节点主要分为两类:



1. 被操作区间完全覆盖的节点（上图红色节点）。
2. 递归过程中会访问到的其余节点（上图黄色节点）。

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：

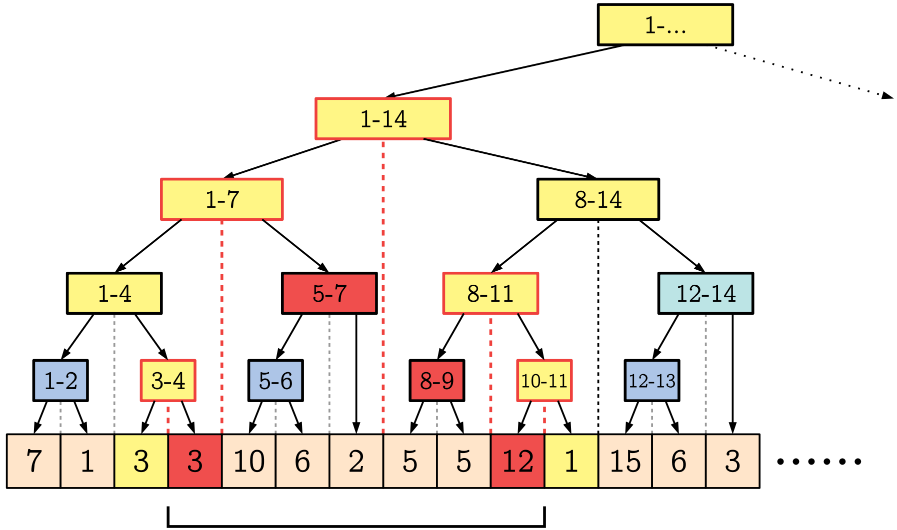


1. 被操作区间完全覆盖的节点（上图红色节点）。
2. 递归过程中会访问到的其余节点（上图黄色节点）。

黄色节点中一类比较特殊的是**分岔节点**（上图红色边框节点），它们的中线穿过了操作区间，因此两个儿子都会被访问到。其余未被访问到的节点都不在考虑范围之内。

线段树：时间复杂度

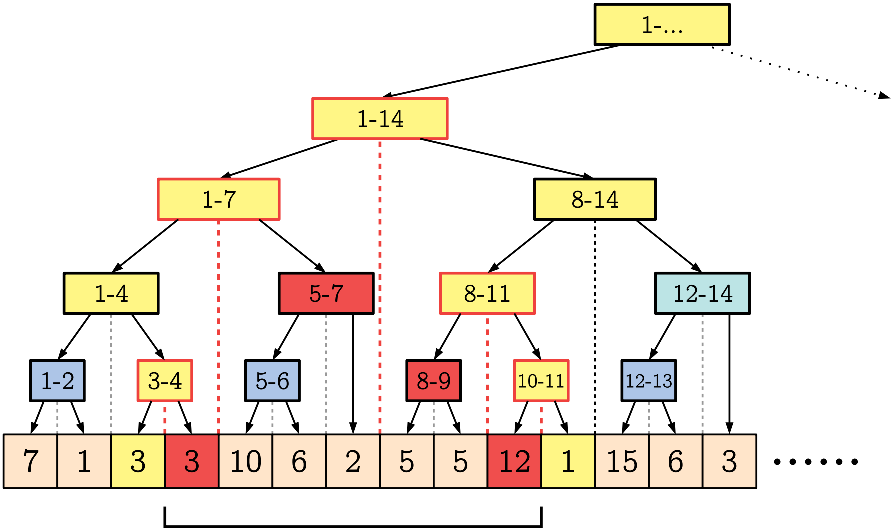
每次区间操作访问过的节点主要分为两类：



在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：

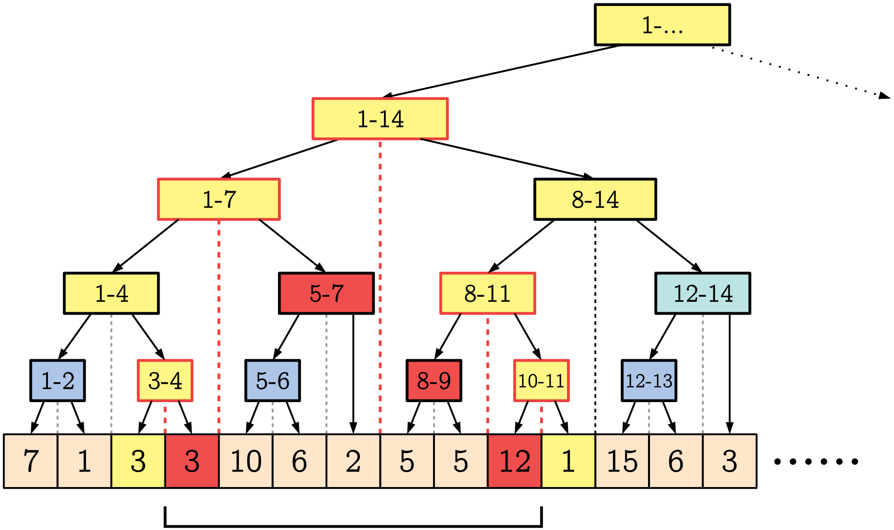


在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

性质 1　一个黄色节点的两个儿子不可能都是红色节点。

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类:



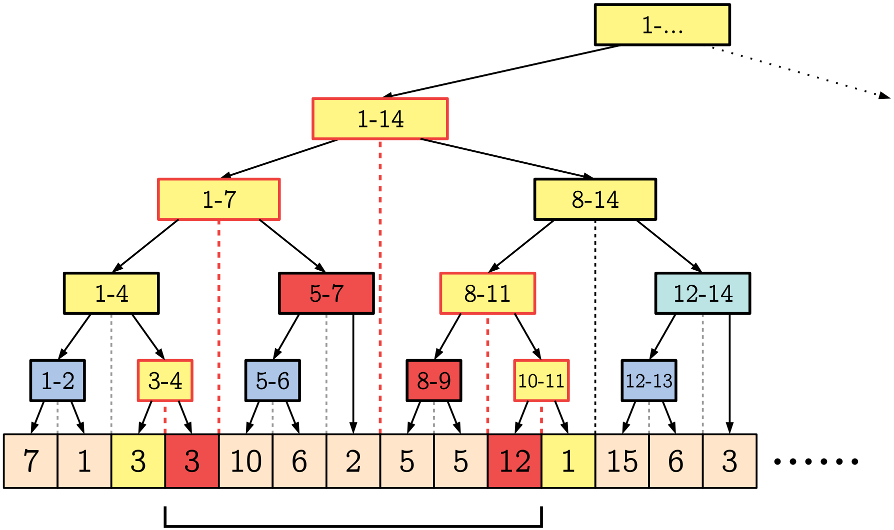
在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点:

性质 1　一个黄色节点的两个儿子不可能都是红色节点。

证明　如果这样，则自己也会被操作区间所覆盖。

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：

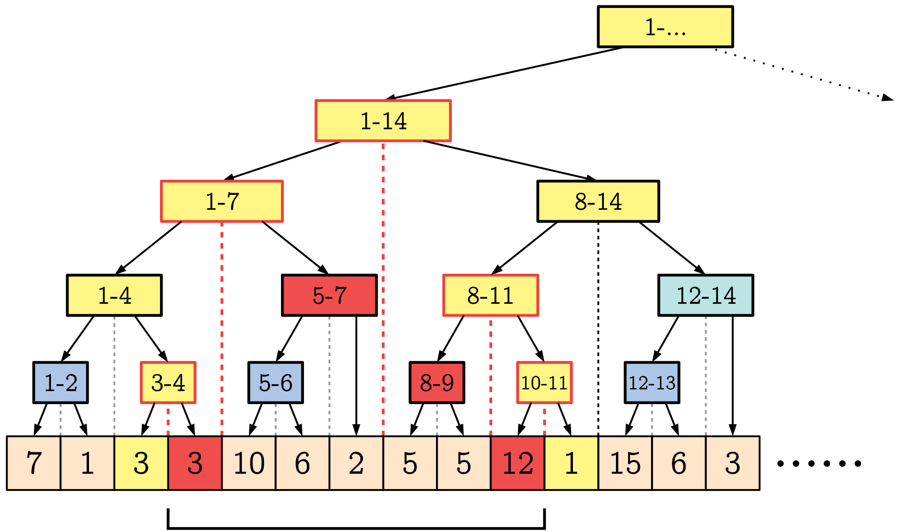


在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

性质 2　最多只有一个分岔节点的两个儿子都是黄色节点。

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：



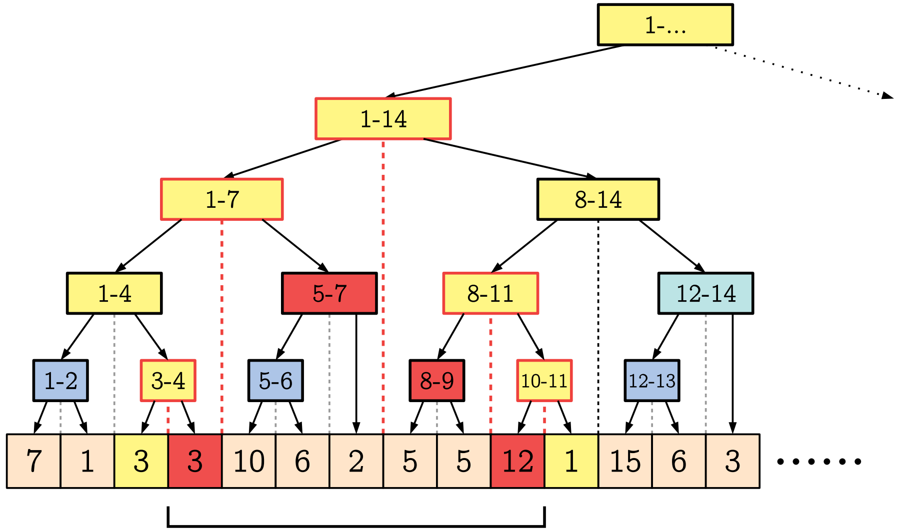
在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

性质 2　最多只有一个分岔节点的两个儿子都是黄色节点。

证明　上图中节点 $[1, 14]$ 就是这样的分岔节点。假设现在存在一个，那么继续递归过程中，如果当前节点是分岔节点，则意味着它的左儿子或者右儿子将被某两条分岔节点的中线所夹。而分岔节点的中线穿过操作区间，所以这个儿子必须是红色节点。

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：

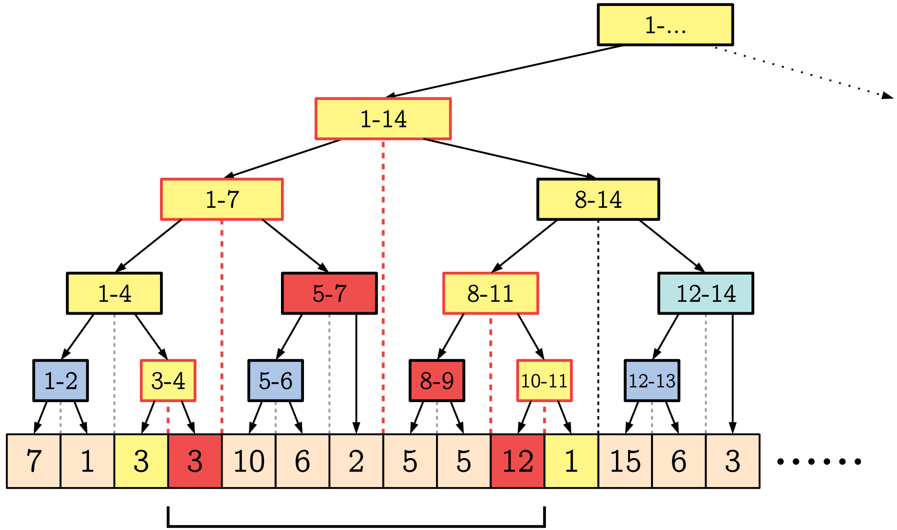


在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

性质 3　红色节点和非根黄色节点的父亲均为黄色节点。

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：



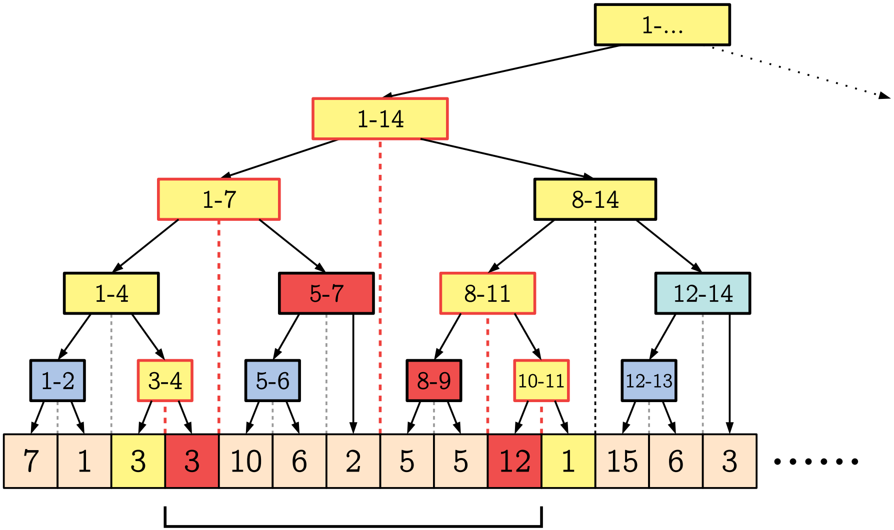
在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

性质 3 红色节点和非根黄色节点的父亲均为黄色节点。

证明 ~~显然~~

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：

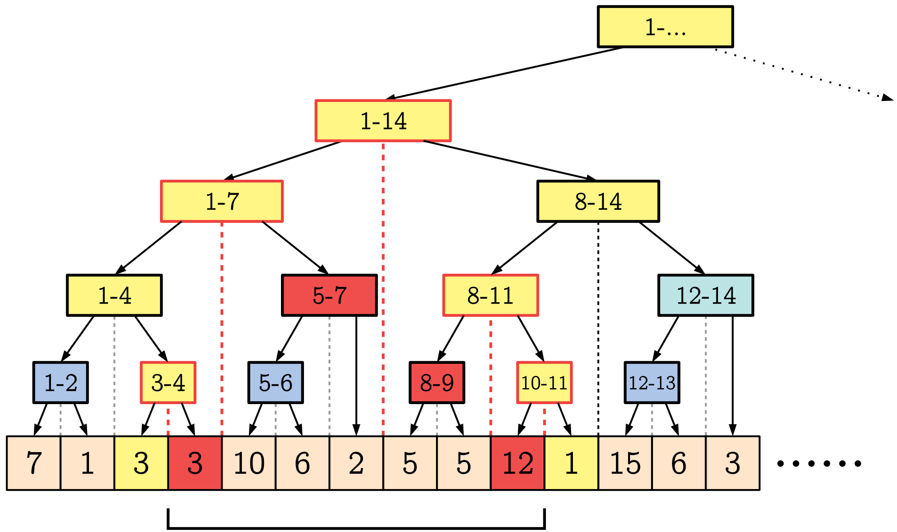


在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

性质 4　线段树的树高为 $\lceil \log n \rceil$。

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：



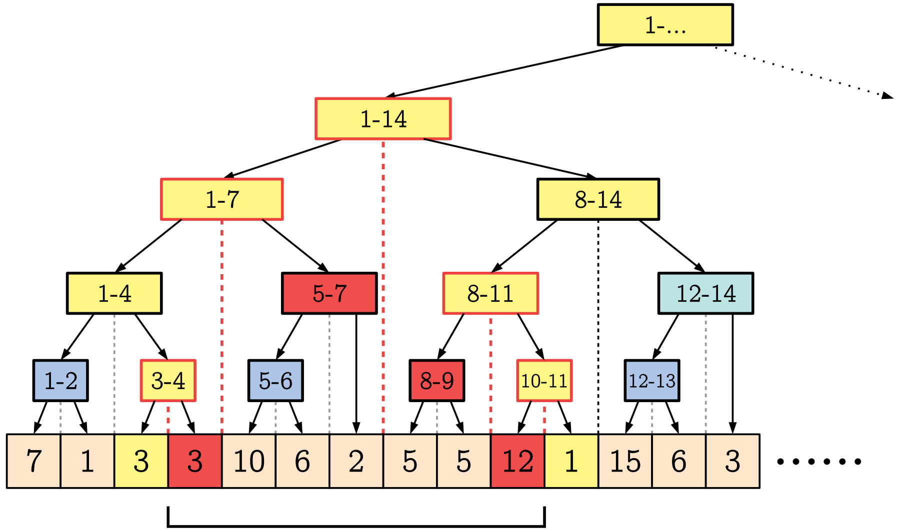
在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

性质 4 线段树的树高为 $\lceil \log n \rceil$。

证明 回忆线段树的构造过程：从中线划开。所以左右儿子所管辖的区间长度至少减半。~~想必你们应该很熟悉了~~

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：

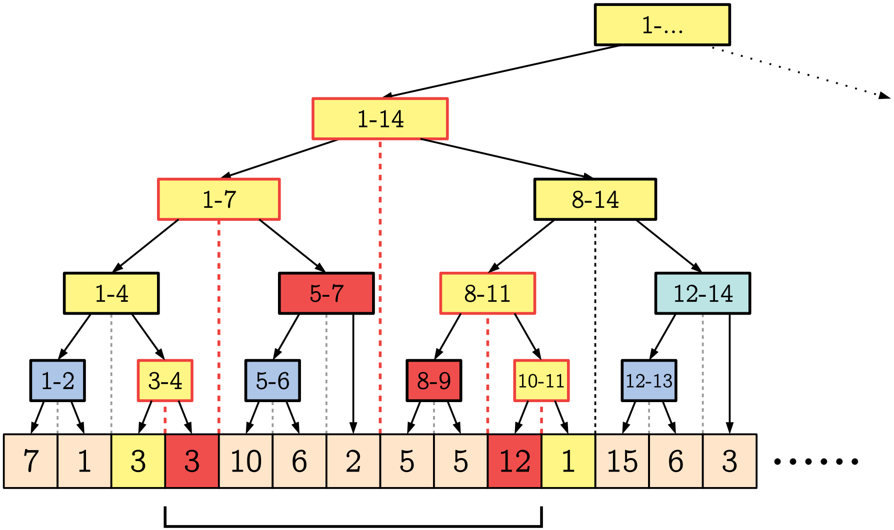


在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

综上，黄色节点至多分岔成两条链，每条链长不能超过树高 $\lceil \log n \rceil$，每个黄色节点最多携带一个红色节点。故单次区间操作访问的节点总数不超过 $4\lceil \log n \rceil$，即线段树区间操作的时间复杂度为 $O(\log n)$。

线段树：时间复杂度

每次区间操作访问过的节点主要分为两类：



在计算线段树区间操作的时间复杂度之前，先来观察一下区间操作的特点：

综上，黄色节点至多分岔成两条链，每条链长不能超过树高 $\lceil \log n \rceil$，每个黄色节点最多携带一个红色节点。故单次区间操作访问的节点总数不超过 $4\lceil \log n \rceil$，即线段树区间操作的时间复杂度为 $O(\log n)$。

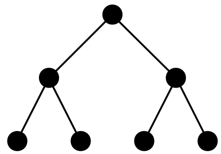
虽然之前给的示例里面线段树处理一个长度为 7 的区间访问了很多节点，但是最后给出的复杂度~~简直完爆分块有木有！~~

满二叉树实现

满二叉树：除叶子节点外，其余节点均有两个儿子的二叉树。

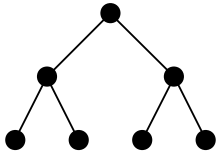
满二叉树实现

满二叉树：除叶子节点外，其余节点均有两个儿子的二叉树。



满二叉树实现

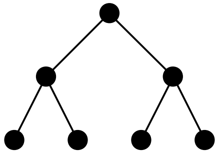
满二叉树：除叶子节点外，其余节点均有两个儿子的二叉树。



高度为 h 的满二叉树上有 $2^h - 1$ 个节点，刚好可以从二进制 $1_{(2)}$ 编号到 $111...1_{(2)}$（h 个 1）。

满二叉树实现

满二叉树：除叶子节点外，其余节点均有两个儿子的二叉树。

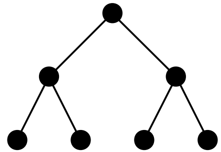


高度为 h 的满二叉树上有 $2^h - 1$ 个节点，刚好可以从二进制 $1_{(2)}$ 编号到 $111...1_{(2)}$（h 个 1）。

编号：从上至下，从左至右依次从 1 开始编号。

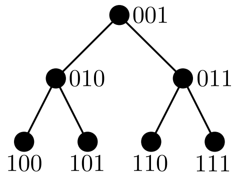
满二叉树实现

满二叉树：除叶子节点外，其余节点均有两个儿子的二叉树。



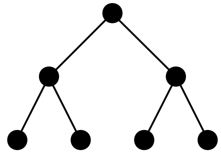
高度为 h 的满二叉树上有 $2^h - 1$ 个节点，刚好可以从二进制 $1_{(2)}$ 编号到 $111...1_{(2)}$（h 个 1）。

编号：从上至下，从左至右依次从 1 开始编号。



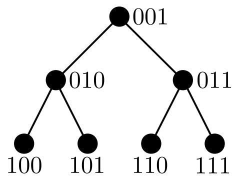
满二叉树实现

满二叉树： 除叶子节点外，其余节点均有两个儿子的二叉树。



高度为 h 的满二叉树上有 $2^h - 1$ 个节点，刚好可以从二进制 $1_{(2)}$ 编号到 $111...1_{(2)}$（h 个 1）。

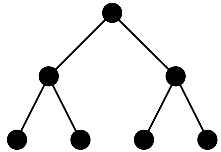
编号： 从上至下，从左至右依次从 1 开始编号。



性质　编号为 `x` 的父亲为 `x >> 1`，其两个儿子分别为 `x << 1` 和 `x << 1 | 1`。

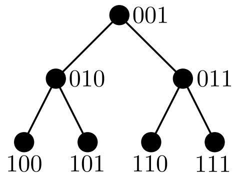
满二叉树实现

满二叉树： 除叶子节点外，其余节点均有两个儿子的二叉树。



高度为 h 的满二叉树上有 2^h-1 个节点，刚好可以从二进制 $1_{(2)}$ 编号到 $111...1_{(2)}$（h 个 1）。

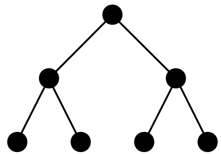
编号： 从上至下，从左至右依次从 1 开始编号。



性质 编号为 `x` 的父亲为 `x >> 1`，其两个儿子分别为 `x << 1` 和 `x << 1 | 1`。
关键在于利用了位运算，不仅简化代码，同时优化了常数。

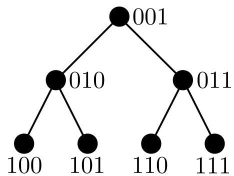
满二叉树实现

满二叉树： 除叶子节点外，其余节点均有两个儿子的二叉树。



高度为 h 的满二叉树上有 2^h-1 个节点，刚好可以从二进制 $1_{(2)}$ 编号到 $111...1_{(2)}$（h 个 1）。

编号： 从上至下，从左至右依次从 1 开始编号。



性质　编号为 `x` 的父亲为 `x >> 1`，其两个儿子分别为 `x << 1` 和 `x << 1 | 1`。
关键在于利用了位运算，不仅简化代码，同时优化了常数。
将序列长度扩充为 2 的某个幂次，然后使用满二叉树的方式存储。

原码 / 反码 / 补码

对于一个有符号整型（如 `int`），其**原码**就是原始二进制表示:

$$13 \to 00001101_{(2)}$$

原码 / 反码 / 补码

对于一个有符号整型（如 `int`），其**原码**就是原始二进制表示:

$$13 \to 00001101_{(2)}$$

其**反码**是将除**符号位**（即最高位）之外的所有位取反得到的二进制:

$$00001101 \to 01110010$$

原码 / 反码 / 补码

对于一个有符号整型（如 `int`），其**原码**就是原始二进制表示:

$$13 \to 00001101_{(2)}$$

其**反码**是将除**符号位**（即最高位）之外的所有位取反得到的二进制:

$$00001101 \to 01110010$$

对于负数，其符号位为1:

$$01110010 \to 11110010$$

原码 / 反码 / 补码

对于一个有符号整型（如 `int`），其**原码**就是原始二进制表示:

$$13 \to 00001101_{(2)}$$

其**反码**是将除**符号位**（即最高位）之外的所有位取反得到的二进制:

$$00001101 \to 01110010$$

对于负数，其符号位为1:

$$01110010 \to 11110010$$

二进制数的**补码**就是将其反码加上 1，对于溢出的位直接丢弃，符号位也参与计算:

$$-13 \to 11110011$$

$$-0 \to 11111111 + 1 \to 1\,00000000 \to 00000000$$

原码 / 反码 / 补码

对于一个有符号整型（如 `int`），其**原码**就是原始二进制表示:

$$13 \to 00001101_{(2)}$$

其**反码**是将除**符号位**（即最高位）之外的所有位取反得到的二进制:

$$00001101 \to 01110010$$

对于负数，其符号位为1:

$$01110010 \to 11110010$$

二进制数的**补码**就是将其反码加上 1，对于溢出的位直接丢弃，符号位也参与计算:

$$-13 \to 11110011$$

$$-0 \to 11111111 + 1 \to 1\,00000000 \to 00000000$$

计算机采用补码存储负数，原码存储自然数，这样减法可以变为加法:

$$1 - 13 = 1 + (-13) \to 00000001 + 11110011 = 11110100 \to -12$$

lowbit

“lowbit” 即最低位，是整型二进制表示中最后一个 1 的位置。

00001101

01101000

lowbit

“lowbit” 即最低位，是整型二进制表示中最后一个 1 的位置。

00001101

01101000

GCC 内置函数 `__builtin_ctz` 可以数出整型末尾 0 的个数（`ctz` 是 Count Tailing Zeros 的缩写）。但是 CCF 明令禁止使用双下划线开头的函数……

lowbit

“lowbit” 即最低位，是整型二进制表示中最后一个 1 的位置。

00001101

01101000

GCC 内置函数 `__builtin_ctz` 可以数出整型末尾 0 的个数（`ctz` 是 Count Tailing Zeros 的缩写）。但是 CCF 明令禁止使用双下划线开头的函数……

另外一种获得 lowbit 的方法是 `x & -x`。因为负数存的是补码，而补码是反码加 1。因此 `x` 末尾的 0 变成 1，lowbit 上的 1 变成 0，然后加 1，末尾的 1 进位后在 lowbit 上填 1，而 lowbit 之前的位置 `x` 和 `-x` 恰好相反，所以这种方法可以得到 lowbit（实际上是 2^{lowbit}）。

lowbit

“lowbit” 即最低位，是整型二进制表示中最后一个 1 的位置。

$$00001101$$

$$01101000$$

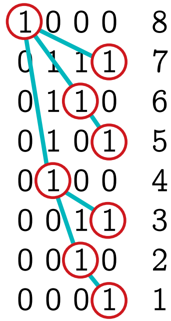
GCC 内置函数 `__builtin_ctz` 可以数出整型末尾 0 的个数（`ctz` 是 Count Tailing Zeros 的缩写）。但是 CCF 明令禁止使用双下划线开头的函数……

另外一种获得 lowbit 的方法是 `x & -x`。因为负数存的是补码，而补码是反码加 1。因此 `x` 末尾的 0 变成 1，lowbit 上的 1 变成 0，然后加 1，末尾的 1 进位后在 lowbit 上填 1，而 lowbit 之前的位置 `x` 和 `-x` 恰好相反，所以这种方法可以得到 lowbit（实际上是 2^{lowbit}）。

$$\begin{aligned} &01101000 \\ &10010111+1 \\ \&\ &10011000 \\ =\ &00001000 \end{aligned}$$

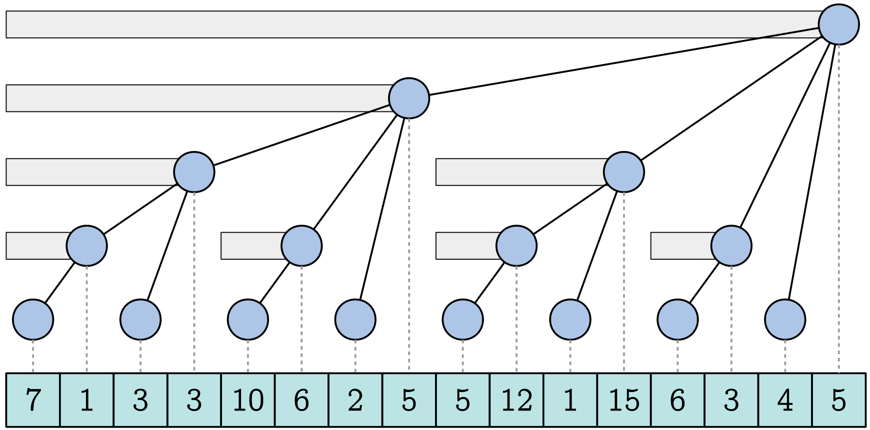
lowbit 树

lowbit 树

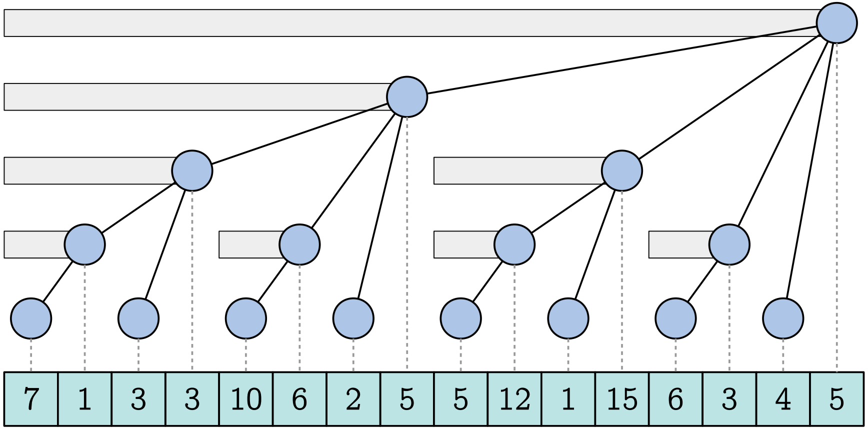


大小为 n 的 lowbit 树可以由上图方法构造：从 1 到 n 对每个整数 x，根据 lowbit 后面 0 的数量，建立不同等级的节点（等级从 0 开始），同时该节点的编号也为 x。对于等级 k 的节点，向在它之前 0 至 $k-1$ 各级最后一个节点连边。

lowbit 树

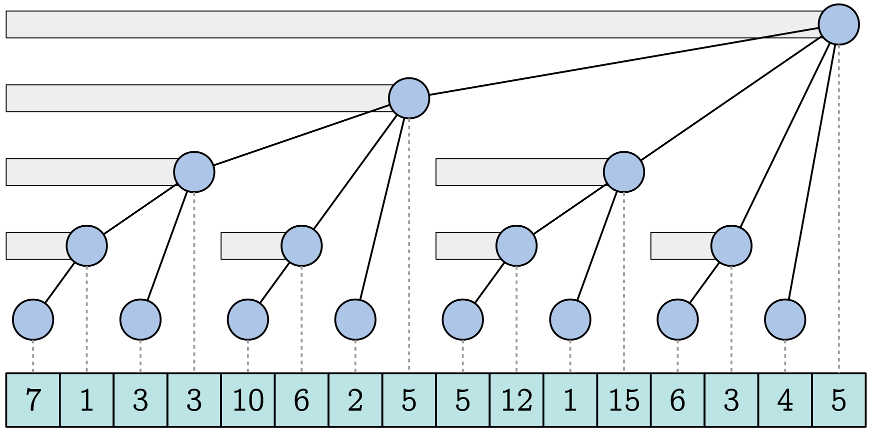


lowbit 树



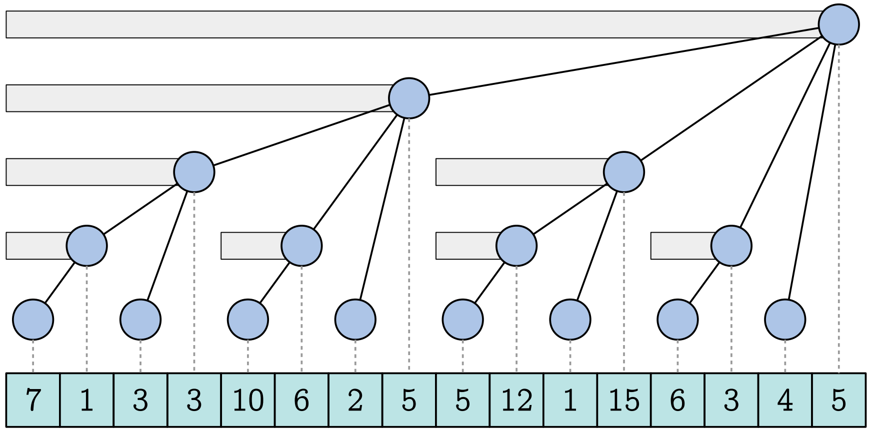
类似线段树，我们可以利用 lowbit 树来处理序列问题。如上图所示，每个节点管辖一个区间，这个区间是其儿子所管辖的区间的并集再加上自己本身所在的元素，并记录所管辖的区间内元素总和 `sum`。现在我们来观察一下 lowbit 树有什么特点。

lowbit 树



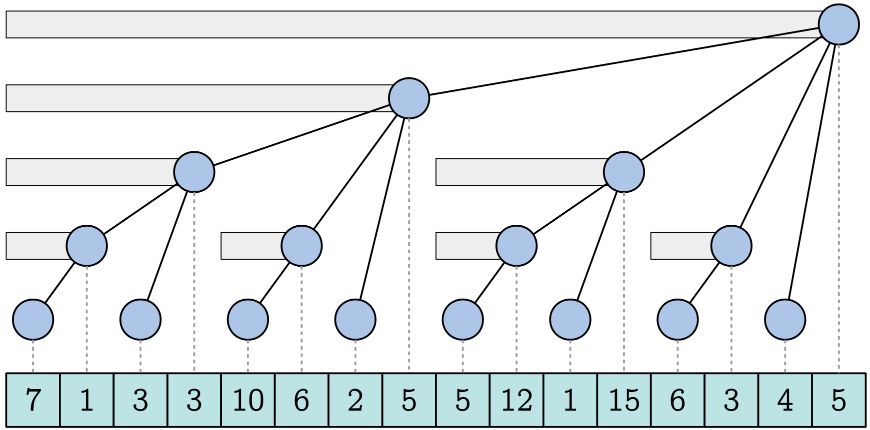
性质 1 等级为 k 的节点管辖的区间长度为 2^k，并且任意两个由等级 k 管辖的区间之间的距离也是 2^k。即在 lowbit 树的第 k 层上以 2^k 为单位长度交替出现。

lowbit 树



性质 1 等级为 k 的节点管辖的区间长度为 2^k，并且任意两个由等级 k 管辖的区间之间的距离也是 2^k。即在 lowbit 树的第 k 层上以 2^k 为单位长度交替出现。

证明 实际上是二进制进位的规律。

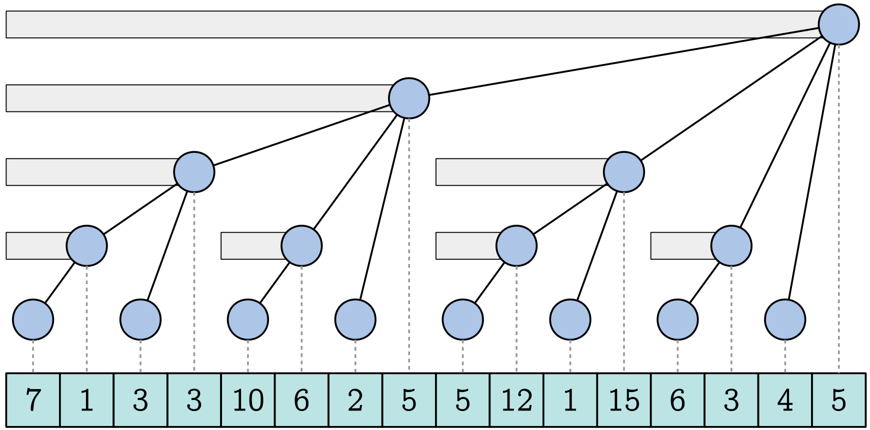


性质 1 等级为 k 的节点管辖的区间长度为 2^k，并且任意两个由等级 k 管辖的区间之间的距离也是 2^k。即在 lowbit 树的第 k 层上以 2^k 为单位长度交替出现。

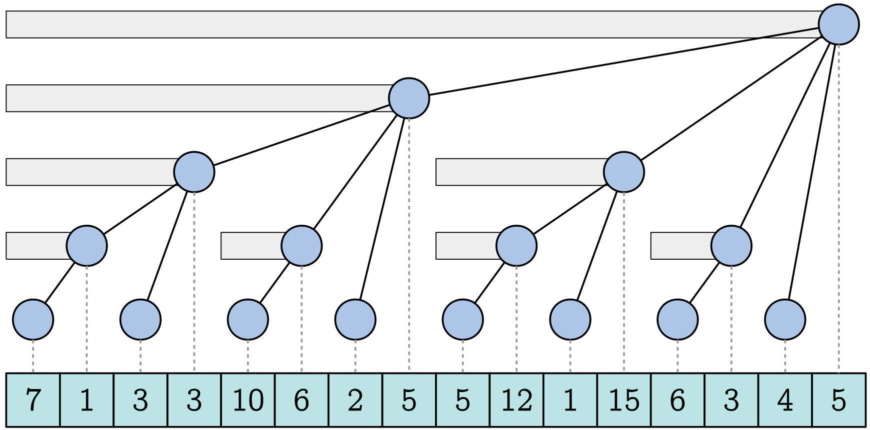
证明 实际上是二进制进位的规律。

先考虑第 0 层，这一层上 0 级节点按奇偶交替分布：只有奇数上方才有 0 级节点。然后将所有奇数删去，二进制中最后一位也就被删去了，此时 1 级节点也是“按奇偶分布”的。重复上述过程可以得到规律。

lowbit 树



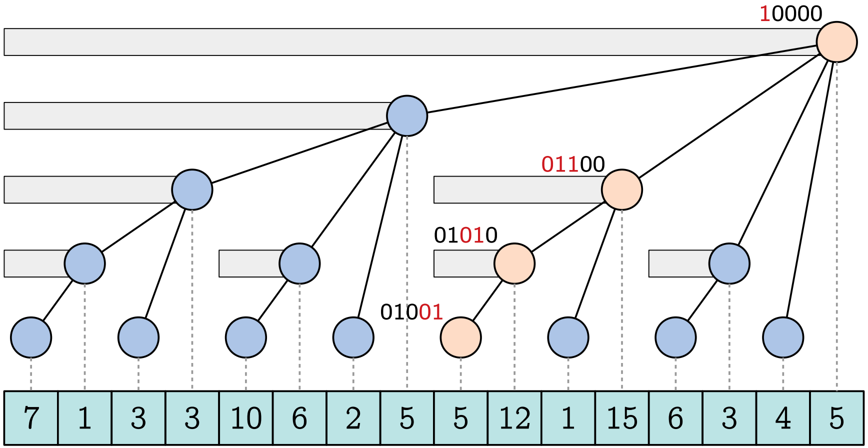
性质 2　靠在节点 x 所管辖的区间左边的节点等级比 x 高。



性质 2 靠在节点 x 所管辖的区间左边的节点等级比 x 高。

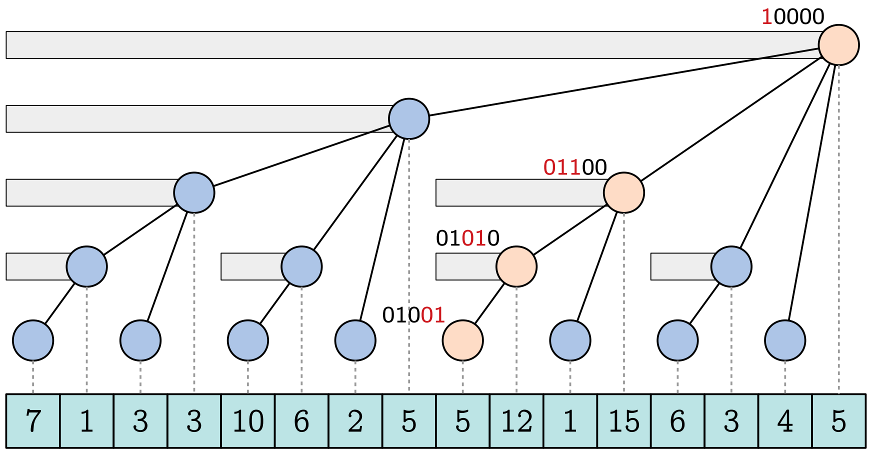
因为这个节点是 $x - 2^{\text{lowbit}}$（即 `x - (x & -x)`），所以其 lowbit 位置更高。

lowbit 树



性质 3　节点 x 的父亲是 $x + 2^{\text{lowbit}}$，即 `x + (x & -x)`。

lowbit 树



性质 3　节点 x 的父亲是 $x + 2^{\text{lowbit}}$，即 `x + (x & -x)`。

因为要成为 x 的父亲，首先必须要 lowbit 比 x 高。而要得到下一个 lowbit 更高的节点，需要在当前 lowbit 位上进位，否则当前 lowbit 下面总有 1，因此在 x 的 lowbit 位上加 1 就是父亲节点。

树状数组

上一页我们发现 lowbit 树上移动非常方便，只需要简单地加或减 `x & -x` 就可以了。因此我们没有必要真正将 lowbit 树的完整结构像线段树那样建出来，只用直接在原序列上操作即可。

树状数组

上一页我们发现 lowbit 树上移动非常方便，只需要简单地加或减 `x & -x` 就可以了。因此我们没有必要真正将 lowbit 树的完整结构像线段树那样建出来，只用直接在原序列上操作即可。

这样的数组被称作**树状数组**。

树状数组

上一页我们发现 lowbit 树上移动非常方便，只需要简单地加或减 `x & -x` 就可以了。因此我们没有必要真正将 lowbit 树的完整结构像线段树那样建出来，只用直接在原序列上操作即可。

这样的数组被称作**树状数组**。

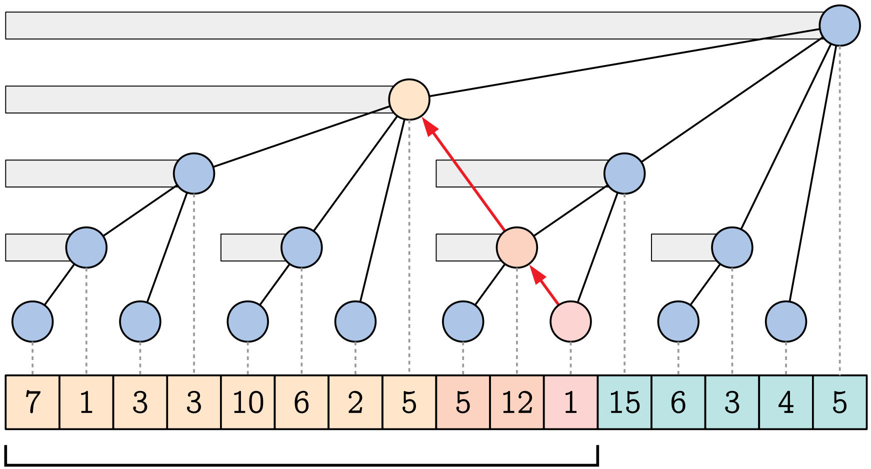
利用性质 1，可以划分一个前缀区间（左端点为 1 的区间）：

树状数组

上一页我们发现 lowbit 树上移动非常方便，只需要简单地加或减 `x & -x` 就可以了。因此我们没有必要真正将 lowbit 树的完整结构像线段树那样建出来，只用直接在原序列上操作即可。

这样的数组被称作**树状数组**。

利用性质 1，可以划分一个前缀区间（左端点为 1 的区间）：



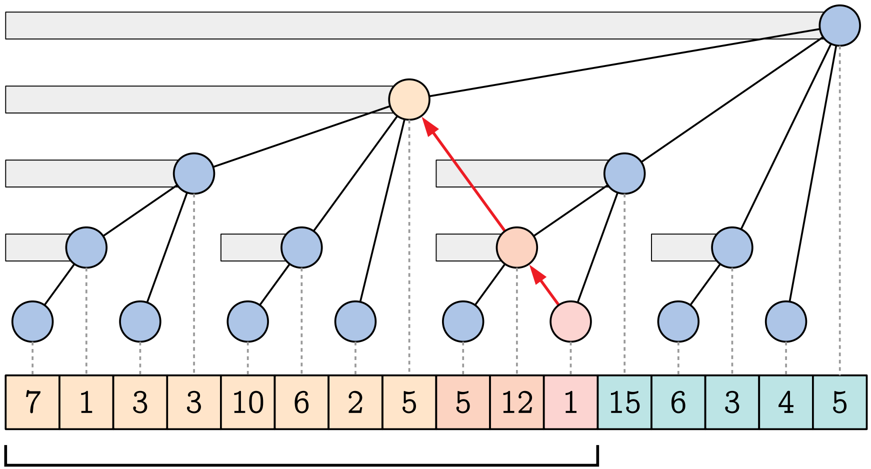
将自己的编号减去自己所管辖的区间长度就可得到下一个划分节点。

树状数组

上一页我们发现 lowbit 树上移动非常方便，只需要简单地加或减 `x & -x` 就可以了。因此我们没有必要真正将 lowbit 树的完整结构像线段树那样建出来，只用直接在原序列上操作即可。

这样的数组被称作**树状数组**。

利用性质 1，可以划分一个前缀区间（左端点为 1 的区间）：



将自己的编号减去自己所管辖的区间长度就可得到下一个划分节点。

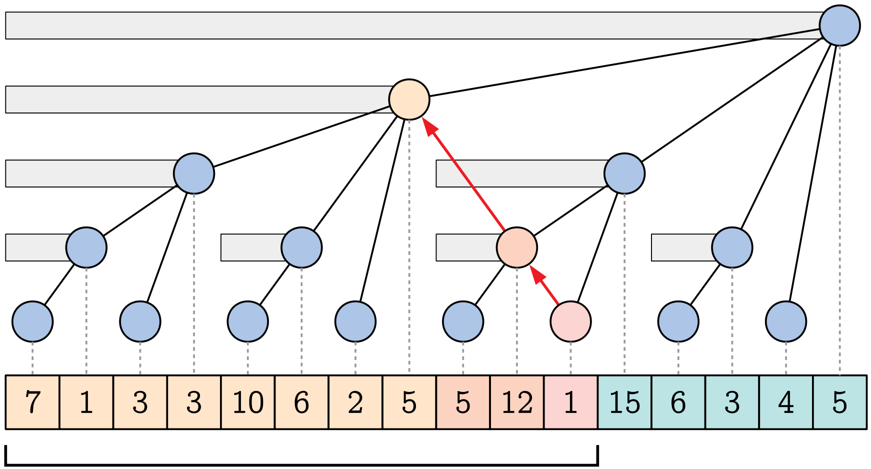
在每个节点处记录了 `sum` 值，这样就可以计算一个前缀区间内元素总和。

树状数组

上一页我们发现 lowbit 树上移动非常方便，只需要简单地加或减 `x & -x` 就可以了。因此我们没有必要真正将 lowbit 树的完整结构像线段树那样建出来，只用直接在原序列上操作即可。

这样的数组被称作**树状数组**。

利用性质 1，可以划分一个前缀区间（左端点为 1 的区间）：



将自己的编号减去自己所管辖的区间长度就可得到下一个划分节点。

在每个节点处记录了 `sum` 值，这样就可以计算一个前缀区间内元素总和。

此外还能支持单点修改。像线段树一样，单点修改还需要更新所有父亲的 `sum` 值。

总而言之，树状数组支持两条路线：

树状数组

总而言之，树状数组支持两条路线:

第一条是向前的路线，之前被用来查询前缀和。

```
function query(int r):  // 区间查询 [1, r]
    ret = 0
    while r > 0:
        ret += sum[r]
        r -= r & -r
    return ret
```

树状数组

总而言之，树状数组支持两条路线:

第一条是向前的路线，之前被用来查询前缀和。

```
function query(int r):  // 区间查询 [1, r]
    ret = 0
    while r > 0:
        ret += sum[r]
        r -= r & -r
    return ret
```

另一条是向后的路线，用于更新树状数组上的信息。

```
function modify(int x, int v):  // 单点修改 A[x] += v
    while x <= n:
        sum[x] += v
        x += x & -x
```

树状数组

总而言之，树状数组支持两条路线:

第一条是向前的路线，之前被用来查询前缀和。

```
function query(int r):  // 区间查询 [1, r]
    ret = 0
    while r > 0:
        ret += sum[r]
        r -= r & -r
    return ret
```

另一条是向后的路线，用于更新树状数组上的信息。

```
function modify(int x, int v):  // 单点修改 A[x] += v
    while x <= n:
        sum[x] += v
        x += x & -x
```

这两个操作还可以反转一下，向前的用于前缀区间修改，向后的用于单点查询。此时的修改相当于打了一个整体增加的标记。

树状数组

总而言之，树状数组支持两条路线:

第一条是向前的路线，之前被用来查询前缀和。

```
function query(int r):  // 区间查询 [1, r]
    ret = 0
    while r > 0:
        ret += sum[r]
        r -= r & -r
    return ret
```

另一条是向后的路线，用于更新树状数组上的信息。

```
function modify(int x, int v):  // 单点修改 A[x] += v
    while x <= n:
        sum[x] += v
        x += x & -x
```

这两个操作还可以反转一下，向前的用于前缀区间修改，向后的用于单点查询。此时的修改相当于打了一个整体增加的标记。

支持查询前缀和就支持查询区间和，因为 $[l, r]$ 的和可以用 $[1, r]$ 的和减去 $[1, l-1]$ 的和得到。

总而言之，树状数组支持两条路线：
第一条是向前的路线，之前被用来查询前缀和。

```
function query(int r):  // 区间查询 [1, r]
    ret = 0
    while r > 0:
        ret += sum[r]
        r -= r & -r
    return ret
```

另一条是向后的路线，用于更新树状数组上的信息。

```
function modify(int x, int v):  // 单点修改 A[x] += v
    while x <= n:
        sum[x] += v
        x += x & -x
```

这两个操作还可以反转一下，向前的用于前缀区间修改，向后的用于单点查询。此时的修改相当于打了一个整体增加的标记。
支持查询前缀和就支持查询区间和，因为 $[l, r]$ 的和可以用 $[1, r]$ 的和减去 $[1, l-1]$ 的和得到。
同理，支持修改前缀就可以支持区间修改：将区间 $[l, r]$ 都加上 v 可以视为将 $[1, r]$ 加上 v 后又使 $[1, l-1]$ 减去 v / 加上 $-v$。

树状数组：时间复杂度

只需要分析向前和向后两条路线的长度就好了。

树状数组：时间复杂度

只需要分析向前和向后两条路线的长度就好了。

无论是哪一条路线，节点的等级一直是严格单增的，而最高等级是 $\lfloor \log n \rfloor$，所以时间复杂度都为 $O(\log n)$。

树状数组

现在有个尴尬的地方：树状数组可以支持区间查询或者区间修改，但两者不能兼得。

树状数组

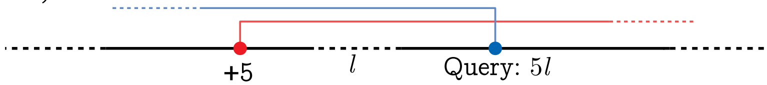
现在有个尴尬的地方：树状数组可以支持区间查询或者区间修改，但两者不能兼得。
需要一点小技巧，树状数组也可以同时支持两种操作。

前缀和 & 后缀和

目标：支持修改后缀和，查询前缀和。

前缀和 & 后缀和

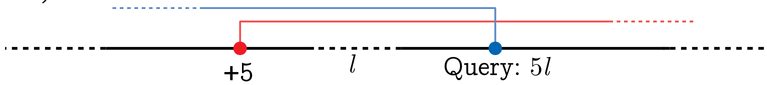
目标：支持修改后缀和，查询前缀和。



如图，红色是后缀加 5，蓝色是前缀查询，两者之间有 l 个元素（包括红蓝两点），修改操作对查询操作的贡献是 $5l$。

前缀和 & 后缀和

目标：支持修改后缀和，查询前缀和。

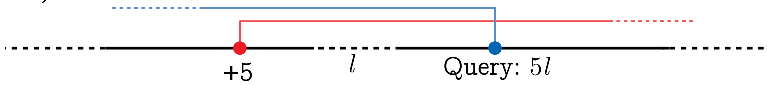


如图，红色是后缀加 5，蓝色是前缀查询，两者之间有 l 个元素（包括红蓝两点），修改操作对查询操作的贡献是 $5l$。

在数据结构中，我们使用节点上的数据来加速查询操作，此时这 $5l$ 的总贡献可能需要分开考虑：

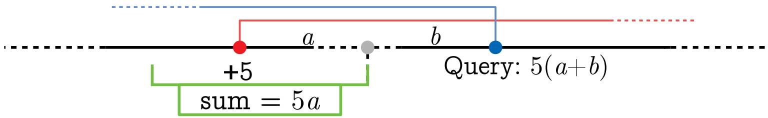
前缀和 & 后缀和

目标：支持修改后缀和，查询前缀和。



如图，红色是后缀加 5，蓝色是前缀查询，两者之间有 l 个元素（包括红蓝两点），修改操作对查询操作的贡献是 $5l$。

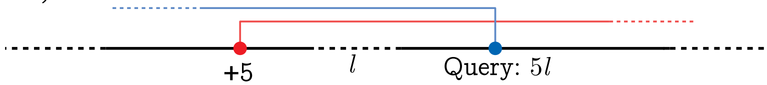
在数据结构中，我们使用节点上的数据来加速查询操作，此时这 $5l$ 的总贡献可能需要分开考虑：



绿色方框代表树状数组中的节点，sum 存储了它所管辖范围内元素总和，即 $5a$。注意 a 是与查询操作无关的，与其有关的是另一个长度 b。

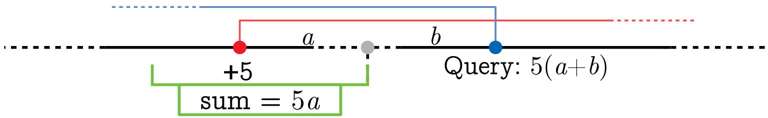
前缀和 & 后缀和

目标：支持修改后缀和，查询前缀和。



如图，红色是后缀加 5，蓝色是前缀查询，两者之间有 l 个元素（包括红蓝两点），修改操作对查询操作的贡献是 $5l$。

在数据结构中，我们使用节点上的数据来加速查询操作，此时这 $5l$ 的总贡献可能需要分开考虑：

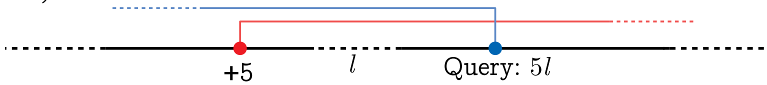


绿色方框代表树状数组中的节点，`sum` 存储了它所管辖范围内元素总和，即 $5a$。注意 a 是与查询操作无关的，与其有关的是另一个长度 b。

实际的总贡献是 $5(a+b)$，而 $5b$ 只有在知道查询操作时才能确定，因此除了 `sum` 之外，还需要记录一个 `tag` 值，表示所有修改操作的参数之和。这样对查询的贡献为 $\text{sum} + \text{tag} \cdot b$。

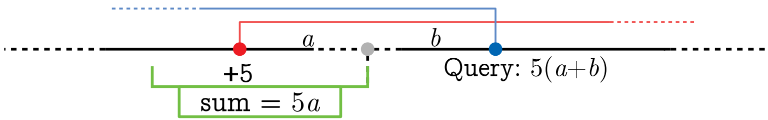
前缀和 & 后缀和

目标：支持修改后缀和，查询前缀和。



如图，红色是后缀加 5，蓝色是前缀查询，两者之间有 l 个元素（包括红蓝两点），修改操作对查询操作的贡献是 $5l$。

在数据结构中，我们使用节点上的数据来加速查询操作，此时这 $5l$ 的总贡献可能需要分开考虑：



绿色方框代表树状数组中的节点，sum 存储了它所管辖范围内元素总和，即 $5a$。注意 a 是与查询操作无关的，与其有关的是另一个长度 b。

实际的总贡献是 $5(a+b)$，而 $5b$ 只有在知道查询操作时才能确定，因此除了 sum 之外，还需要记录一个 tag 值，表示所有修改操作的参数之和。这样对查询的贡献为 $\text{sum} + \text{tag} \cdot b$。

相当于使用了两个树状数组。

树状数组：完整实现

```
int sum[], tag[]
function modify(int l, int v):
    int x = l
    while x <= n:
        sum[x] += (x - l + 1) * v
        tag[x] += v
        x += x & -x
function query(int r):
    int ret = 0
    int x = r
    while x > 0:
        ret += sum[x] + (r - x) * tag[x]
        x -= x & -x
    return ret
function real_modify(int l, int r, int v):
    modify(l, v)
    modify(r + 1, -v)
function real_query(int l, int r):
    return query(r) - query(l - 1)
```

树状数组：完整实现

```
int sum[], tag[]
function modify(int l, int v):
    int x = l
    while x <= n:
        sum[x] += (x - l + 1) * v
        tag[x] += v
        x += x & -x
function query(int r):
    int ret = 0
    int x = r
    while x > 0:
        ret += sum[x] + (r - x) * tag[x]
        x -= x & -x
    return ret
function real_modify(int l, int r, int v):
    modify(l, v)
    modify(r + 1, -v)
function real_query(int l, int r):
    return query(r) - query(l - 1)
```

树状数组常数很小，代码也相对好写。

总结

到目前为止，一共口胡了三个东西：

总结

到目前为止，一共口胡了三个东西：

1. 分块：时间复杂度 $O(\sqrt{n})$。

总结

到目前为止，一共口胡了三个东西：

1. 分块：时间复杂度 $O(\sqrt{n})$。
2. 线段树：区间操作最多访问 $4\lceil\log n\rceil$ 个节点。

总结

到目前为止，一共口胡了三个东西:

1. 分块：时间复杂度 $O(\sqrt{n})$。
2. 线段树：区间操作最多访问 $4\lceil \log n \rceil$ 个节点。
3. 树状数组：区间操作最多访问 $2(\lfloor \log n \rfloor + 1)$ 个节点。

总结

到目前为止，一共口胡了三个东西：

1. 分块：时间复杂度 $O(\sqrt{n})$。
2. 线段树：区间操作最多访问 $4\lceil\log n\rceil$ 个节点。
3. 树状数组：区间操作最多访问 $2(\lfloor\log n\rfloor+1)$ 个节点。

现在是时候来练练手了！实践出真知！

【LG P3373】线段树模板题

初始一个长度为 n 的序列，之后有 q 次操作，操作是以下三种之一：

1. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素加 v。
2. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素乘以 v。
3. 给定区间 $[l, r]$ 和 P，求序列 $A[l..r]$ 中所有元素的和对 P 取模后的值。

$n, q \leqslant 10^5$

【LG P3373】线段树模板题

初始一个长度为 n 的序列，之后有 q 次操作，操作是以下三种之一：

1. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素加 v。
2. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素**乘以** v。
3. 给定区间 $[l, r]$ 和 P，求序列 $A[l..r]$ 中所有元素的和对 P 取模后的值。

$n, q \leqslant 10^5$

如果不知道取模可以先不用管……假设不会爆 `long long`。

【LG P3373】线段树模板题

初始一个长度为 n 的序列，之后有 q 次操作，操作是以下三种之一：

1. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素加 v。
2. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素**乘以** v。
3. 给定区间 $[l, r]$ 和 P，求序列 $A[l..r]$ 中所有元素的和对 P 取模后的值。

$n, q \leqslant 10^5$

如果不知道取模可以先不用管......假设不会爆 `long long`。

除了区间加和区间查询之外，还加入了区间乘操作。

【LG P3373】线段树模板题

初始一个长度为 n 的序列，之后有 q 次操作，操作是以下三种之一：

1. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素加 v。
2. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素乘以 v。
3. 给定区间 $[l, r]$ 和 P，求序列 $A[l..r]$ 中所有元素的和对 P 取模后的值。

$n, q \leqslant 10^5$

如果不知道取模可以先不用管……假设不会爆 `long long`。

除了区间加和区间查询之外，还加入了区间乘操作。

打两个标记，分别是加法的 `add` 标记和乘法的 `mul` 标记，然后使用线段树就好了。

【LG P3373】线段树模板题

初始一个长度为 n 的序列，之后有 q 次操作，操作是以下三种之一：

1. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素加 v。
2. 给定区间 $[l, r]$ 和 v，将序列 $A[l..r]$ 中每个元素乘以 v。
3. 给定区间 $[l, r]$ 和 P，求序列 $A[l..r]$ 中所有元素的和对 P 取模后的值。

$n, q \leqslant 10^5$

如果不知道取模可以先不用管……假设不会爆 `long long`。

除了区间加和区间查询之外，还加入了区间乘操作。

打两个标记，分别是加法的 `add` 标记和乘法的 `mul` 标记，然后使用线段树就好了。

此外需要仔细考虑两个标记共存的情况，需要你自己定义到底是先乘法还是先加法。

【POJ 2777】 Count Colors

现在有 30 种颜色，和一个长度为 L 的板子，每单位长度上可以画上某种颜色。有 q 个操作，操作有以下两种:

1. 将区间 $[l, r]$ 内的板子画上颜色 C。
2. 数出区间 $[l, r]$ 有多少种不同的颜色。

$L, q \leqslant 10^5$

【POJ 2777】 Count Colors

现在有 30 种颜色，和一个长度为 L 的板子，每单位长度上可以画上某种颜色。有 q 个操作，操作有以下两种:

1. 将区间 $[l, r]$ 内的板子画上颜色 C。
2. 数出区间 $[l, r]$ 有多少种不同的颜色。

$L, q \leqslant 10^5$

一段区间内是否有某种颜色 C 可以用一个长度为 30 的 `bitset` 来表示，`int` 也可以。

【POJ 2777】 Count Colors

现在有 30 种颜色，和一个长度为 L 的板子，每单位长度上可以画上某种颜色。有 q 个操作，操作有以下两种:

1. 将区间 $[l, r]$ 内的板子画上颜色 C。
2. 数出区间 $[l, r]$ 有多少种不同的颜色。

$L, q \leqslant 10^5$

一段区间内是否有某种颜色 C 可以用一个长度为 30 的 `bitset` 来表示，`int` 也可以。
使用线段树，记录这样的 `bitset`，再加入一个区间修改的标记即可。

【SHOI 2007 / LG P2163】园丁的烦恼

在一个平面上有 n 棵树，每棵树有一个正整数点坐标 (x_k, y_k)。现在有 q 个询问，每个询问包含一个四边与坐标轴平行的矩形，要求输出位于该矩形内的树的个数。

$n, q \leqslant 5 \times 10^5$

【SHOI 2007 / LG P2163】园丁的烦恼

在一个平面上有 n 棵树，每棵树有一个正整数点坐标 (x_k, y_k)。现在有 q 个询问，每个询问包含一个四边与坐标轴平行的矩形，要求输出位于该矩形内的树的个数。

$n, q \leqslant 5 \times 10^5$

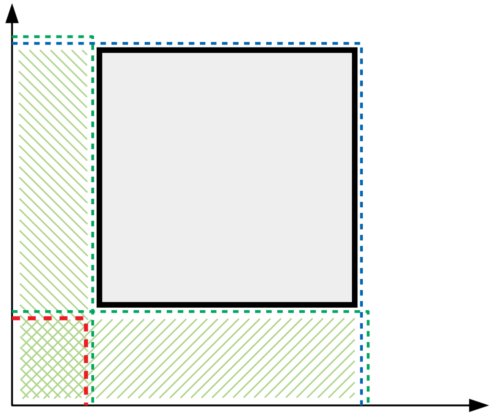
~~二维线段树二维树状数组~~

【SHOI 2007 / LG P2163】园丁的烦恼

在一个平面上有 n 棵树，每棵树有一个正整数点坐标 (x_k, y_k)。现在有 q 个询问，每个询问包含一个四边与坐标轴平行的矩形，要求输出位于该矩形内的树的个数。

$n, q \leqslant 5 \times 10^5$

将矩形询问拆分成四个前缀询问：

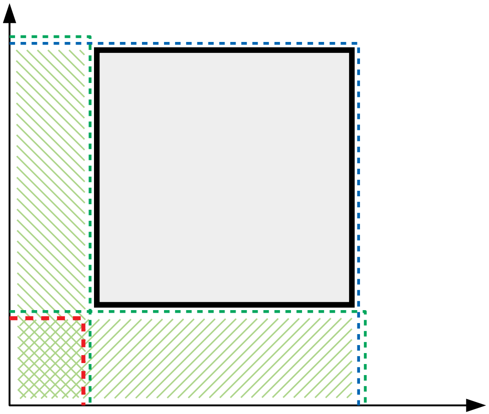


【SHOI 2007 / LG P2163】园丁的烦恼

在一个平面上有 n 棵树，每棵树有一个正整数点坐标 (x_k, y_k)。现在有 q 个询问，每个询问包含一个四边与坐标轴平行的矩形，要求输出位于该矩形内的树的个数。

$n, q \leqslant 5 \times 10^5$

将矩形询问拆分成四个前缀询问：



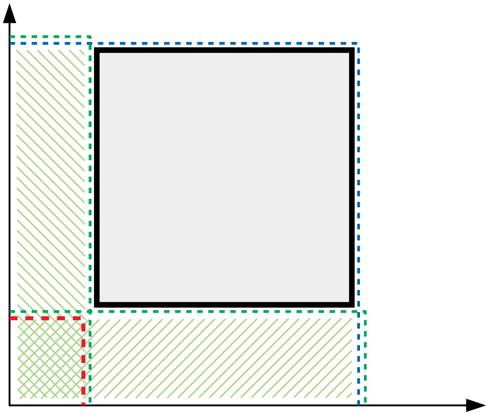
使用一根扫描线，从 $y = 0$ 开始逐渐增加，遍历所有的树。这样就可以处理 y 方向上的前缀和。

【SHOI 2007 / LG P2163】园丁的烦恼

在一个平面上有 n 棵树，每棵树有一个正整数点坐标 (x_k, y_k)。现在有 q 个询问，每个询问包含一个四边与坐标轴平行的矩形，要求输出位于该矩形内的树的个数。

$n, q \leqslant 5 \times 10^5$

将矩形询问拆分成四个前缀询问：



使用一根扫描线，从 $y = 0$ 开始逐渐增加，遍历所有的树。这样就可以处理 y 方向上的前缀和。

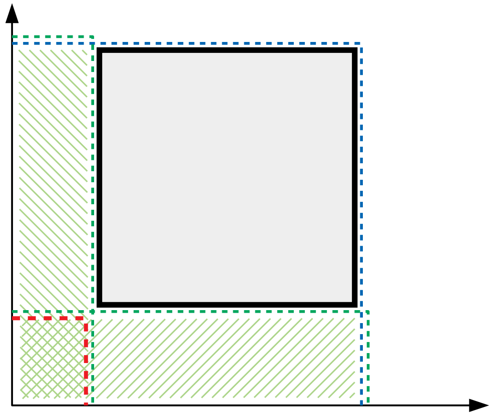
还剩下 x 一维的前缀和，这里就可以使用树状数组啦～

【SHOI 2007 / LG P2163】园丁的烦恼

在一个平面上有 n 棵树，每棵树有一个正整数点坐标 (x_k, y_k)。现在有 q 个询问，每个询问包含一个四边与坐标轴平行的矩形，要求输出位于该矩形内的树的个数。

$n, q \leqslant 5 \times 10^5$

将矩形询问拆分成四个前缀询问：



使用一根扫描线，从 $y = 0$ 开始逐渐增加，遍历所有的树。这样就可以处理 y 方向上的前缀和。

还剩下 x 一维的前缀和，这里就可以使用树状数组啦～

具体而言，将所有树和拆分后询问（都是平面上的点）按照 y 优先，x 其次的顺序排序，然后按顺序遍历。如果当前遍历到树，则在树状数组 x 对应位置上加 1；如果是询问，则在树状数组中 x 对应位置处查询前缀和。最后综合一下每个询问的结果输出。

选做题 #1

【LG P1382】：楼房

选做题 #1

【LG P1382】：楼房

Notice：不一定要用分块 / 线段树 / 树状数组。其他的方法也是可以做的。

选做题 #2

【HNOI 2010 / LG P3203】：弹飞绵羊

选做题 #2

【HNOI 2010 / LG P3203】：弹飞绵羊

Hint：分块。

选做题 #2

【HNOI 2010 / LG P3203】：弹飞绵羊

Hint：分块。

Final Hint：对每一块维护从块中某个点进入这个块后将用多少步才能跳出，以及跳出后会落在的地方。