# 线段树

## Segment Tree

河南省实验中学信息技术组

2026 年 04 月 23 日






# 回顾

- 分治思想
- 堆的性质

# 【引例】数列操作

【题目描述】
给定一个长度为 $n$ 的整数序列 $a[1:n]$，然后对此序列进行 $m$ 次操作，每次我们可以进行如下操作：

1. `c x val`：表示将 $a[x]$ 增加 $z$。
2. `a l r val`：表示将区间 $[l,r]$ 所有数值都增加 $val$。
3. `q l r`：表示查询区间 $[l,r]$ 的所有数值和。

【输入格式】
第一行一个一个整数 $n(n \leq 10^5)$，表示序列的长度。
接下来一行 $n$ 个整数，表示原始序列。
接下列一行一个整数 $m(m \leq 10^5)$，表示操作的次数。
接下来 $m$ 行，每行一个操作，具体操作见题面描述。
【输出格式】
对于每个查询操作，输出一行一个整数表示查询结果。

# 【引例】数列操作

【样例输入】

```
10
0 5 3 0 4 3 2 3 2 1
5
c 6 2
q 3 8
a 1 5 4
a 4 7 2
q 3 8
```

【样例输出】

```
17
37
```

# 【引例】数列操作

- 用数组存储整个序列。
- 对于操作❶和❷，修改对应位置元素即可。
- 对于操作❸，遍历求和即可。
- 当然区间修改可以利用差分来加快速度，但是由于修改和查询操作随机进行，差分无法对问题进行优化。
- 每次操作的时间复杂度为 $\mathrm{O}(N)$，总体时间复杂度为 $\mathrm{O}(MN)$。
- 当 $n, m$ 过大时，这种算法效率很低。其低效的原因主要是每一次操作都是针对每个元素进行维护的，而要求的操作都是针对区间的。
- 假如设计一种数据结构，能够直接维护所需处理的区间，那么就能更加有效地解决这个问题，这就是线段树(区间树)。

# 线段树

线段树 (区间树，Segment Tree) 是一种基于分治思想的二叉树结构，用于在区间上进行信息统计。它的具体定义如下：

1. 线段树的每个结点都代表一个区间。
2. 线段树具有唯一的根结点，代表的区间是整个统计范围，例如 $[1, n]$。
3. 线段树的每个叶子结点都代表一个长度为 1 的元区间 $[x, x]$。
4. 对于每个内部结点 $[l, r]$，它的左孩子结点是 $[l, m]$，右孩子结点是 $[m + 1, r]$，其中 $m = \lfloor \frac{l+r}{2} \rfloor$。

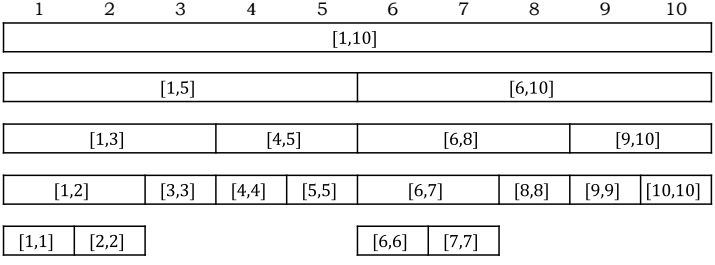


图: 区间视角

# 线段树

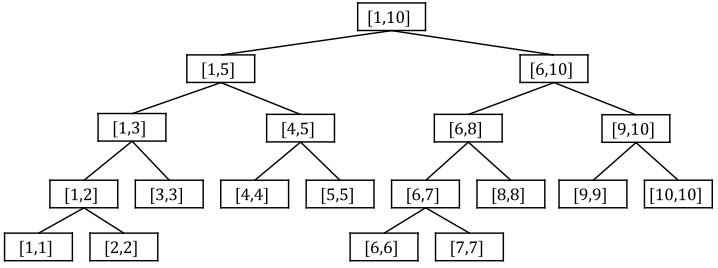


图: 二叉树视角

线段树性质:

1. 树的深度不超过 $\lfloor \log_2(n-1) \rfloor + 1$。
2. 结点个数为 $2n - 1$[1]。
3. 任意一个区间 $[l, r]$ 都分成不超过 $2\log_2(r-l+1)$ 个区间。

[1]对于任意一棵二叉树，如果其叶子结点有 $n_0$ 个，度为 2 的结点有 $n_2$ 个，则 $n_0 = n_2 + 1$。

# 存储方式

- 按照二叉树的孩子表示法进行存储。
- 每个结点需要存储区间和、左右孩子下标、区间起点终点等。
- 根据线段树的性质可知，线段树的结点数不超过区间长度的 2 倍。

```
// 线段树结点数不超过区间长度的 2 倍
int sum[2 * N];    // 区间和
int lc[2 * N], rc[2 * N]; // 左孩子 右孩子
int L[2 * N], R[2 * N];   // 区间 [L,R]
int rt = 0, tot = 0;    // 根结点 结点数目计数器
```

# 单点修改

- `c x val`：表示将 $a[x]$ 增加 $val$。
- 从根结点出发，递归找到代表区间 $[x,x]$ 的叶结点，然后从下向上更新 $[x,x]$ 以及它的所有祖先结点上保存的信息。

```
// 单点修改
void update(int c, int x, int val) // c 表示当前结点
{
    if(L[c] == R[c]) { sum[c] += val; return; }  // 叶子结点
    int M = (L[c] + R[c]) / 2;       // 区间分界点
    if(x <= M) update(lc[c], x, val);      // x 在左半区间
    else       update(rc[c], x, val);      // x 在右半区间
    sum[c] = sum[lc[c]] + sum[rc[c]];
}
update(rt, x, val);  // 调用方法
```

- 时间复杂度 $\mathrm{O}(\log N)$。

# 单点修改

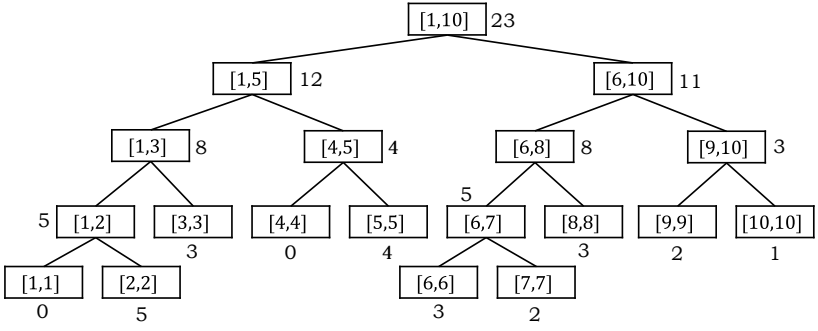
[1,10] 23
[1,5] 12
[6,10] 11
[1,3] 8
[4,5] 4
[6,8] 8
[9,10] 3
5
[1,2]
[3,3]
3
[4,4]
0
[5,5]
4
5
[6,7]
[8,8]
3
[9,9]
2
[10,10]
1
[1,1]
0
[2,2]
5
[6,6]
3
[7,7]
2

图: 单点修改前

# 单点修改

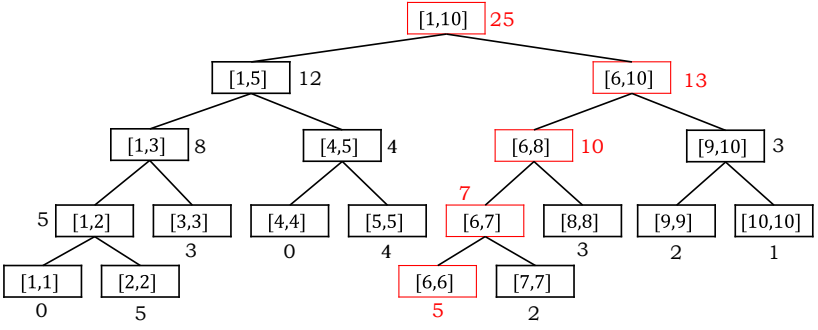
[1,10] 25
[1,5] 12
[6,10] 13
[1,3] 8
[4,5] 4
[6,8] 10
[9,10] 3
5
[1,2]
[3,3] 3
[4,4] 0
[5,5] 4
7
[6,7]
[8,8] 3
[9,9] 2
[10,10] 1
[1,1] 0
[2,2] 5
[6,6] 5
[7,7] 2

图: 将 $a[6]$ 加 2

# 区间查询

- `q l r`：表示查询区间 $[l, r]$ 的所有数值和。
- 将待查询区间分割成若干个线段树结点 (区间)：
  - 若 $[l, r]$ 完全覆盖了当前结点代表的区间，则立即返回，该结点的 $sum$ 的值即为答案的组成部分。
  - 否则，若左子结点与 $[l, r]$ 有重叠部分，则递归访问左子结点。
  - 若右子结点与 $[l, r]$ 有重叠部分，则递归访问右子结点。

```
1  // 区间查询
2  int query(int c, int l, int r) // c 为当前结点
3  {
4      if(l <= L[c] && R[c] <= r) return sum[c];   // 区间全覆盖
5      int M = (L[c] + R[c]) / 2;       // 区间分界点
6      int sum = 0;
7      if(l <= M) sum += query(lc[c], l, r); // 左半段区间和
8      if(M < r)  sum += query(rc[c], l, r); // 右半段区间和
9      return sum;
10 }
11 int ans = query(rt, l, r);        // 调用方法
```

- 时间复杂度 $\mathrm{O}(\log N)$。

# 区间查询



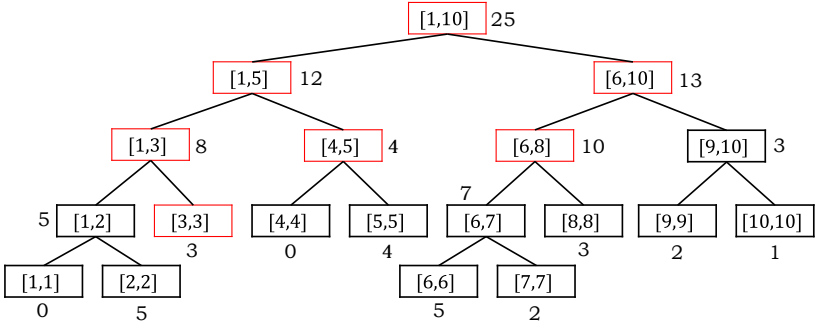


**图: 查询区间** $[3,8]$

# 建树

- 建树：利用原始序列 $a[1:n]$ 建立线段树。

```
1  // 建树
2  int build(int l, int r)
3  {
4      int c = ++tot;
5      L[c] = l, R[c] = r;
6      if(l == r) { sum[c] = a[l]; return c; } // 叶子结点
7      int m = (l + r) / 2;
8      lc[c] = build(l, m);      // 递归建左子树
9      rc[c] = build(m + 1, r); // 递归建右子树
10     sum[c] = sum[lc[c]] + sum[rc[c]];   // 从下向上更新区间信息
11     return c;
12 }
13
14 rt = build(1, n); // 调用方法
```

- 时间复杂度 $\mathrm{O}(N)$。

# 区间修改

- 区间修改需要在线段树结点中增加延迟标记。
- 延迟标记的作用就是暂时标记某一结点 (区间) 值修改的情况，从而无需修改它的子孙结点的值。
- 如果需要查询或再修改带延迟标记结点 (区间) 的某一部分，才将延迟标记下传给子孙结点。
- 以区间求和为例，我们需要在结点中增加属性 $add$ 用来保存区间各数值的增量。

```
1 // 线段树结点数不超过区间长度的 2 倍
2 int sum[2 * N];    // 区间和
3 int add[2 * N];    // 区间各数值的增量延迟标记
4 int lc[2 * N], rc[2 * N]; // 左孩子 右孩子
5 int L[2 * N], R[2 * N];   // 区间 [L,R]
6 int rt = 0, tot = 0;    // 根结点 结点数目计数器
```

# 延迟标记下传

- 如果需要查询或再修改带延迟标记结点 (区间) 的某一部分，需要将延迟标记下传给子孙结点。

```
// 延迟标记下传
void pushdown(int c)
{
    if(add[c] == 0) return;
    sum[lc[c]] += add[c] * (R[lc[c]] - L[lc[c]] + 1);  // 更新左孩子信息
    sum[rc[c]] += add[c] * (R[rc[c]] - L[rc[c]] + 1);  // 更新右孩子信息
    add[lc[c]] += add[c];    // 给左孩子打延迟标记
    add[rc[c]] += add[c];    // 给右孩子打延迟标记
    add[c] = 0;              // 延迟标记已下传 清除之
}
```

# 区间修改

- `a l r val`：表示将区间 $[l, r]$ 所有数值都增加 $val$。
- 如果修改区间完全覆盖某一结点区间，只对该区间进行标记即可。

```
1  // 区间修改
2  void update(int c, int l, int r, int val)  // c 为当前结点
3  {
4      if(l <= L[c] && R[c] <= r)      // 区间完全覆盖
5      {
6          sum[c] += val * (R[c] - L[c] + 1);   // 更新节点和信息到最新
7          add[c] += val;           // 给结点打延迟标记
8          return;
9      }
10     pushdown(c);  // 修改区间和结点区间部分重合，需要下传标记
11     int M = (L[c] + R[c]) / 2;       // 区间分界点
12     if(l <= M) update(lc[c], l, r, val);
13     if(M < r)  update(rc[c], l, r, val);
14     sum[c] = sum[lc[c]] + sum[rc[c]]; // 从下向上更新信息
15 }
```

- 时间复杂度 $\mathrm{O}(\log N)$。

# 区间修改

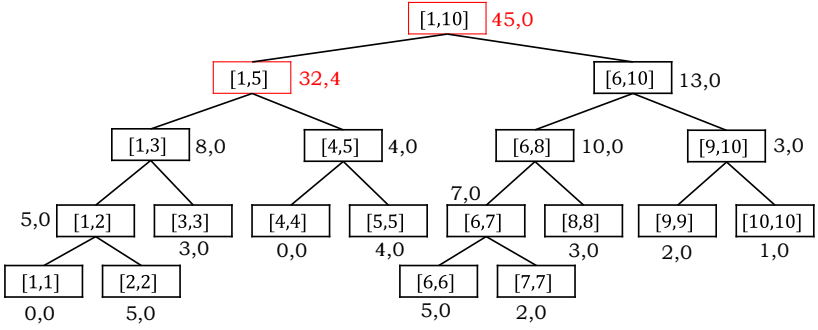


图: 将区间 $[1,5]$ 的元素增加 4

# 区间修改

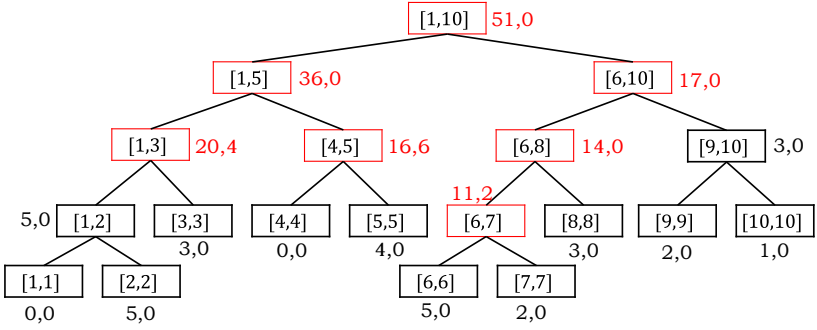


图: 将区间 $[4,7]$ 的元素增加 2

# 带延迟标记的区间查询

- q l r：表示查询区间 $[l, r]$ 的所有数值和。
- 此时查询区间可能信息未更新，因为它的祖先结点有延迟标记。

```
1  int query(int c, int l, int r) // c 为当前结点
2  {
3      if(l <= L[c] && R[c] <= r) return sum[c];   // 区间全覆盖
4      pushdown(c);    // 查询区间和结点区间部分重合，需要下传标记
5      int M = (L[c] + R[c]) / 2;       // 区间分界点
6      int sum = 0;
7      if(l <= M) sum += query(lc[c], l, r); // 左半段区间和
8      if(M < r)  sum += query(rc[c], l, r); // 右半段区间和
9      return sum;
10 }
11 int ans = query(rt, l, r);        // 调用方法
```

- 时间复杂度 $O(\log N)$。

# 带延迟标记的区间查询



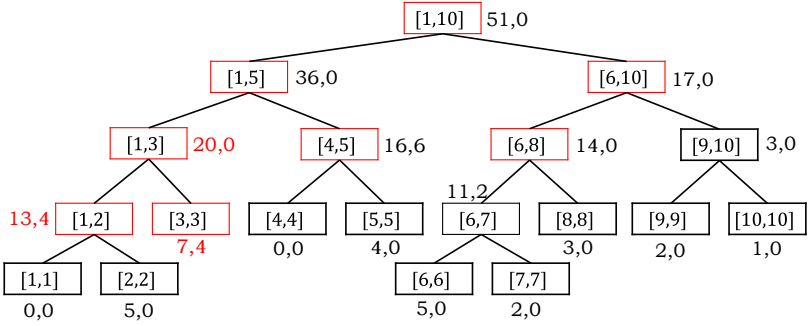


图: 查询区间 $[3,8]$

# 带延迟标记的单点修改

- `c x val`：表示将 $a[x]$ 增加 $val$。
- 单点修改会影响从根结点到叶子结点 $[x, x]$ 中间的所有结点区间，这些区间需要下传标记。

```
1 void update(int c, int x, int val) // c 表示当前结点
2 {
3     if(L[c] == R[c]) { sum[c] += val; return; }  // 叶子结点
4     pushdown(c);    // 单点会影响所有经过的区间，需要下传标记
5     int M = (L[c] + R[c]) / 2;       // 区间分界点
6     if(x <= M) update(lc[c], x, val);      // x 在左半区间
7     else       update(rc[c], x, val);      // x 在右半区间
8     sum[c] = sum[lc[c]] + sum[rc[c]];
9 }
```

- 单点修改可以看作是对区间 $[x, x]$ 的修改，可以不实现。
- 时间复杂度 $\mathrm{O}(\log N)$。

# 逆序对

【逆序对】
设 $a$ 为一个有 $n$ 个数字的有序集 $(n > 1)$，其中所有数字各不相同。如果存在正整数 $i, j$ 使得 $1 \leq i < j \leq n$ 而且 $a[i] > a[j]$，则 $(a[i], a[j])$ 这个对称为 $a$ 的一个逆序对。
例如：数组 $\{3, 1, 4, 5, 2\}$ 的逆序对有 $(3, 1), (3, 2), (4, 2), (5, 2)$，共 4 个。
那么，对于一个数组，如何求出这个数组的逆序对数呢?
【输入格式】
第一行一个整数 $n(n \leq 10^5)$。
接下来一行 $n$ 个整数，表示序列。输入保证 $1 \leq a[i] \leq 10^6$。
【输出格式】
一个整数，表示序列的逆序对个数。
【样例输入】

```
10
9 1 4 2 6 7 5 8 3 10
```

【样例输出】

```
16
```

# 逆序对

- 在数据范围 $[1, 10^6 + 5]$ 内建立线段树统计每个数出现的次数。
- 遍历数组中的数 $a[i]$，对于 $a[i]$ 它的逆序对是在它前面比它大的数据个数，可以通过查询 $[a[i] + 1, 10^6 + 5]$ 中的数出现的次数来统计，累加该结果即可。
- 这种存储数据出现次数或其他统计信息的线段树被称为权值线段树。

# 逆序对

```
1 const int INF = 1e6 + 5;
2 long long ans = 0;
3 for(int i = 1; i <= n; ++i)
4 {
5     ans += query(rt, a[i] + 1, INF);
6     update(rt, a[i], 1);  // a[i] 出现次数加 1
7 }
8 cout << ans;
```

- 时间复杂度：$O(N \log M)$，$M$ 为数值范围大小。
- 当数值范围较大时，时间复杂度较高，该算法效率降低。
- 当然可以先进行离散化，但是离散化必须要使用排序算法，那就不如直接用归并排序来求解逆序对。
- 线段树一般主要针对全排列或者数据范围小的数据，一般需要维护逆序对变化。


# 小结



- 线段树是一种比较通用的处理区间修改、区间查询的数据结构。
- 使用线段树时要求结点中存储的信息能按照区间进行划分与合并(又称满足区间可加性)。
- 相比于树状数组，线段树理解较简单，但实现较复杂；而树状数组理解困难，但实现简单。
- 一般能用树状数组解决的问题也能用线段树解决，反之则不然。






# 练习

- 数列操作 A(COGS 264)
- 数列操作 B(COGS 1316)
- 数列操作 C(COGS 1317)
- 售票系统 (COGS 247)
- Balanced Lineup(COGS 3704)
- 广告牌 (COGS 2372)
- 快速矩阵操作 (COGS 3962)
- 开关 [TJOI 2009](洛谷 P3870)
- 无聊的数列 (洛谷 P1438)
- 扶苏的问题 (洛谷 P1253)
- 简单的线段树 (COGS 2961)

# 基于堆结构的实现

- 通过观察可以发现，除去线段树的最后一层，整棵树是一棵完全二叉树，故而可以用类似二叉堆的方式存储。
- 根据二叉堆的性质，父子结点位置可以相互推导，因此结点中不需要存储左右孩子位置。
- 在堆的存储模式下，最底层有大量空余，所以线段树结点空间不小于区间长度的 4 倍。

```
1 // 线段树结点数不小于区间长度的 4 倍
2 int sum[4 * N];    // 区间和
3 int add[4 * N];    // 区间各数值的增量延迟标记
4 int L[4 * N], R[4 * N];   // 区间 [L,R]
```

# 基于堆结构的实现

- 建树

```
// 建树 根结点一定为 1
void build(int c, int l, int r)
{
    L[c] = l, R[c] = r;
    if(l == r) { sum[c] = a[l]; return; } // 叶子结点
    int m = (l + r) / 2;
    int lc = c * 2, rc = c * 2 + 1;     // 左右孩子
    build(lc, l, m);     // 递归建左子树
    build(rc, m + 1, r); // 递归建右子树
    sum[c] = sum[lc] + sum[rc];   // 从下向上更新区间信息
}
```

# 基于堆结构的实现

- 标记下传

```
// 延迟标记下传
void pushdown(int c)
{
    if(add[c] == 0) return;
    int lc = c * 2, rc = c * 2 + 1;     // 左右孩子
    sum[lc] += add[c] * (R[lc] - L[lc] + 1);  // 更新左孩子信息
    sum[rc] += add[c] * (R[rc] - L[rc] + 1);  // 更新右孩子信息
    add[lc] += add[c];    // 给左孩子打延迟标记
    add[rc] += add[c];    // 给右孩子打延迟标记
    add[c] = 0;             // 延迟标记已下传 清除之
}
```

# 基于堆结构的实现

- 区间修改

```
1  void update(int c, int l, int r, int val)  // c 为当前结点
2  {
3      if(l <= L[c] && R[c] <= r)       // 区间完全覆盖
4      {
5          sum[c] += val * (R[c] - L[c] + 1);   // 更新节点和信息到最新
6          add[c] += val;           // 给结点打延迟标记
7          return;
8      }
9      pushdown(c);  // 修改区间和结点区间部分重合，需要下传标记
10     int M = (L[c] + R[c]) / 2;       // 区间分界点
11     int lc = c * 2, rc = c * 2 + 1;      // 左右孩子
12     if (l <= M) update(lc, l, r, val);
13     if (M < r)  update(rc, l, r, val);
14     sum[c] = sum[lc] + sum[rc];   // 从下向上更新信息
15 }
```

# 基于堆结构的实现

- **区间查询**

```
int query(int c, int l, int r) // c 为当前结点
{
    if(l <= L[c] && R[c] <= r) return sum[c];   // 区间全覆盖
    pushdown(c);    // 查询区间和结点区间部分重合，需要下传标记
    int M = (L[c] + R[c]) / 2;       // 区间分界点
    int lc = c * 2, rc = c * 2 + 1;     // 左右孩子
    int sum = 0;
    if (l <= M) sum += query(lc, l, r); // 左半段区间和
    if (M < r)  sum += query(rc, l, r); // 右半段区间和
    return sum;
}
```

# 不存储区间

- 通过观察可以发现，线段树结点所代表的区间也是不变的，因此可以不在结点中存储区间边界，只需要在处理时将区间边界作为函数参数即可。

```
// 线段树结点数不小于区间长度的 4 倍
int sum[4 * N];    // 区间和
int add[4 * N];    // 区间各数值的增量延迟标记
// 注意后续用 L,R 表示结点的区间 (大写暗示其是确定不变的)
```

# 不存储区间

- 建树

```
1  // 建树 根结点一定为 1
2  void build(int c, int L, int R)
3  {
4      if(L == R) { sum[c] = a[L]; return; } // 叶子结点
5      int M = (L + R) / 2;
6      int lc = c * 2, rc = c * 2 + 1;      // 左右孩子
7      build(lc, L,     M);
8      build(rc, M + 1, R);
9      sum[c] = sum[lc] + sum[rc];  // 从下向上更新区间信息
10 }
```

# 不存储区间

- 标记下传

```
1  void pushdown(int c, int L, int R)
2  {
3      if(add[c] == 0) return;
4      int lc = c * 2, rc = c * 2 + 1;      // 左右孩子
5      int M = (L + R) / 2;                 // 区间分界点
6      sum[lc] += add[c] * (M - L + 1);  // 更新左孩子信息
7      sum[rc] += add[c] * (R - M);      // 更新右孩子信息
8      add[lc] += add[c];                // 给左孩子打延迟标记
9      add[rc] += add[c];                // 给右孩子打延迟标记
10     add[c] = 0;              // 延迟标记已下传 清除之
11 }
```

# 不存储区间

- 区间修改

```
1  void update(int c, int L, int R, int l, int r, int val)  // c 为当前结点
2  {
3      if(l <= L && R <= r)       // 区间完全覆盖
4      {
5          sum[c] += val * (R - L  + 1); // 更新节点和信息到最新
6          add[c] += val;          // 给结点打延迟标记
7          return;
8      }
9      pushdown(c, L, R);  // 修改区间和结点区间部分重合，需要下传标记
10     int M = (L + R) / 2;             // 区间分界点
11     int lc = c * 2, rc = c * 2 + 1;     // 左右孩子
12     if(l <= M) update(lc, L,     M, l, r, val);
13     if(M < r)  update(rc, M + 1, R, l, r, val);
14     sum[c] = sum[lc] + sum[rc];  // 从下向上更新区间信息
15 }
```

# 不存储区间

- **区间查询**

```
int query(int c, int L, int R, int l, int r) // c 为当前结点
{
    if(l <= L && R <= r) return sum[c];   // 区间全覆盖
    pushdown(c, L, R);    // 查询区间和结点区间部分重合，需要下传标记
    int M = (L + R) / 2;             // 区间分界点
    int lc = c * 2, rc = c * 2 + 1; // 左右孩子
    int sum = 0;
    if(l <= M) sum += query(lc, L,     M, l, r); // 左半段区间和
    if(M < r)  sum += query(rc, M + 1, R, l, r); // 右半段区间和
    return sum;
}
```

# 动态开点

- 在线段树的普通二叉树的实现中，可以在最初值建立一个根结点代表整个区间，当需要访问线段树的某棵子树 (子区间) 时，再建立代表这个子区间的结点。
- 根据线段树定义可知，线段树最多有 $2n-1$ 个结点。

```
// 线段树结点数不超过区间长度的 2 倍
int sum[2 * N];    // 区间和
int add[2 * N];    // 区间各数值的增量延迟标记
int lc[2 * N], rc[2 * N]; // 左孩子 右孩子
int L[2 * N], R[2 * N];   // 区间 [L,R]
int rt = 0, tot = 0;    // 根结点 结点数目计数器
```

# 动态开点

- 建树

```
1  int build(int l, int r)
2  {
3      int c = ++tot;
4      L[c] = l, R[c] = r;
5      sum[c] = s[r] - s[l - 1];
6      add[c] = lc[c] = rc[c] = 0;
7      return c;
8  }
9
10 // 建树
11 rt = build(1, n);
```

# 动态开点

- 标记下传
- 在区间修改和查询时需要提前进行标记下传，可以在标记下传时动态开点，其他部分不需要作修改。

```
1  void pushdown(int c)
2  {
3      if(add[c] == 0) return;
4      int M = (L[c] + R[c]) / 2;
5      if(!lc[c]) lc[c] = build(L[c], M);       // 左孩子动态开点
6      if(!rc[c]) rc[c] = build(M + 1, R[c]);  // 右孩子动态开点
7      sum[lc[c]] += add[c] * (R[lc[c]] - L[lc[c]] + 1);  // 更新左孩子信息
8      sum[rc[c]] += add[c] * (R[rc[c]] - L[rc[c]] + 1);  // 更新右孩子信息
9      add[lc[c]] += add[c];    // 给左孩子打延迟标记
10     add[rc[c]] += add[c];    // 给右孩子打延迟标记
11     add[c] = 0;              // 延迟标记已下传 清除之
12 }
```

# 线段树合并

- 对于若干棵线段树，它们维护相同的区间，不同的操作在不同的线段树上进行，那么最终结果可以通过合并这些线段树得到。
- 对于这些线段树，我们可以依次合并它们，对于两棵以 $p, q$ 为根的线段树：
  - 若 $p, q$ 之一为空，则以非空的那个作为合并后的结点。
  - 若已经到达叶子结点，则直接把两个结点的 $sum$ 值相加即可。
  - 若 $p, q$ 均不为空，则递归合并两棵左子树和两棵右子树，然后删除结点 $q$，以 $p$ 为合并后的结点。

```
1  int merge(int p, int q)
2  {
3      if(!p) return q;
4      if(!q) return p;
5      if(L[p] == R[p]) { sum[p] += sum[q]; return p; }
6      lc[p] = merge(lc[p], lc[q]);
7      rc[p] = merge(rc[p], rc[q]);
8      sum[p] += sum[q];
9      return p;
10 }
```