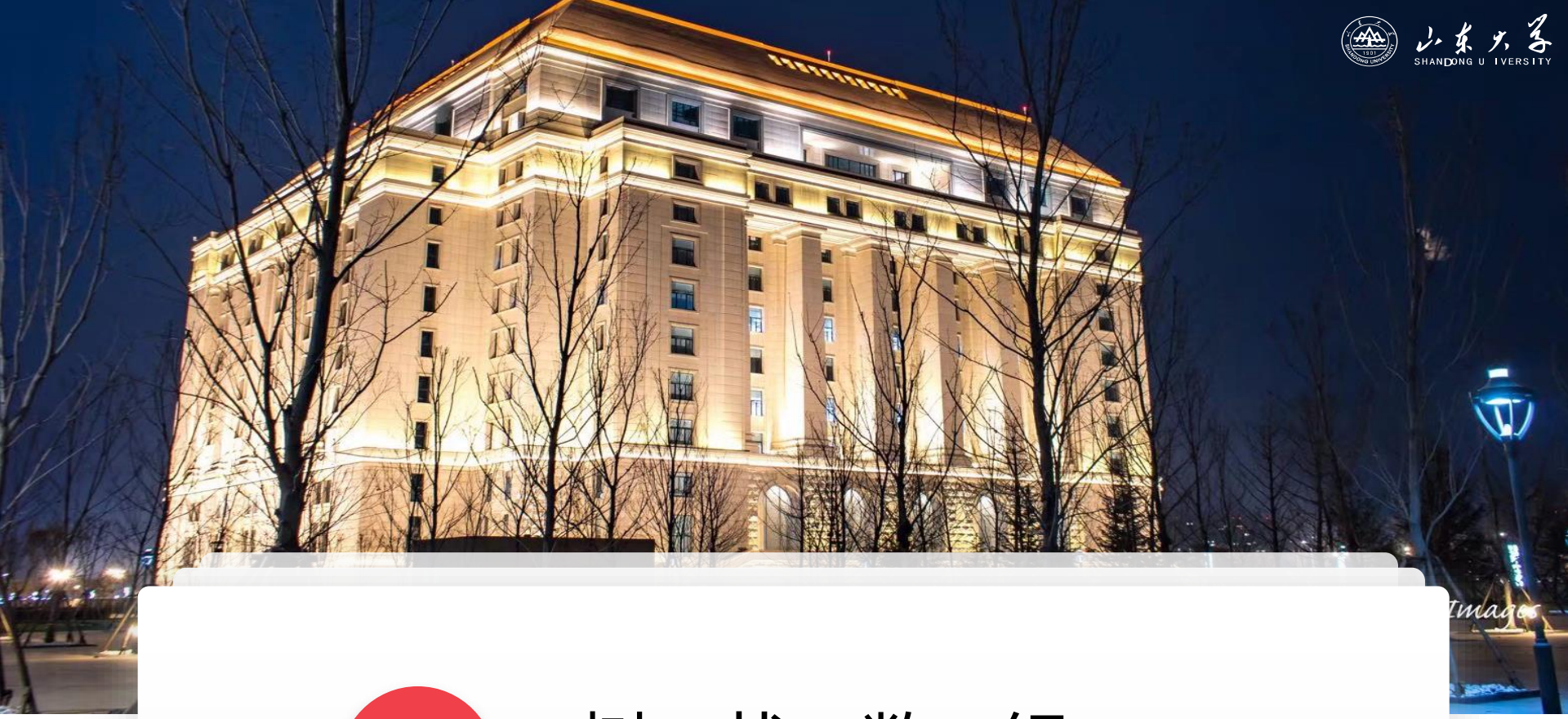
程序设计思维与实践

Thinking and Practice in Programming

树形数据结构及其应用 | 内容负责：叶苏伟



1 树 状 数 组

Binary Indexed Tree

引入

- 问题：求取指定连续区间的和。
- 给定一个数组 A[N],求 A[L:R]的和是多少。

- 解：
- 枚举法：A[L],A[L+1],…,A[R],加起来。时间复杂度：O(n)
- 前缀和差分法：Presum[R]-Presum[L-1]。时间复杂度：O(1)

引入

- 问题：求取求指定连续区间的和。
- 解：
- 通过利用提前维护好的前缀和(O(n))数组，在数组中做一次差分运算(O(1))得到区间和。
- 例如：
  - 对于数组　　d[1:6]　　1, 2, 5, 4, 1, 3
  - 其前缀和数组 s[0:6]　　0, 1, 3, 8, 12, 13, 16

  - 求取原数组 [2, 4] 的区间和时，只需要计算 s[4] - s[1] 即可，即：12 - 1 = 11 = 2 + 5 + 4

引入

- 问题拓展：
- 动态连续区间查询问题：
  - 已知一个数列，你需要进行下面两种操作：
  - 1.求出某区间每一个数的和
  - 2.将某一个数加上x

- 解法：枚举法，差分法。
- 对于前缀和，需要修改时，则需要付出 O(n) 的代价。
- 例如：
  - 将上述的 d[2] 改为 3，那么相对的，s[2:6] 都需要进行修正。
  - d'[1:6]　　1,　3,　5,　4,　1,　3
  - s'[0:6]　　0,　1,　4,　9,　13,　14,　17

- 程序时间复杂度O(n^2)。

树状数组的定义

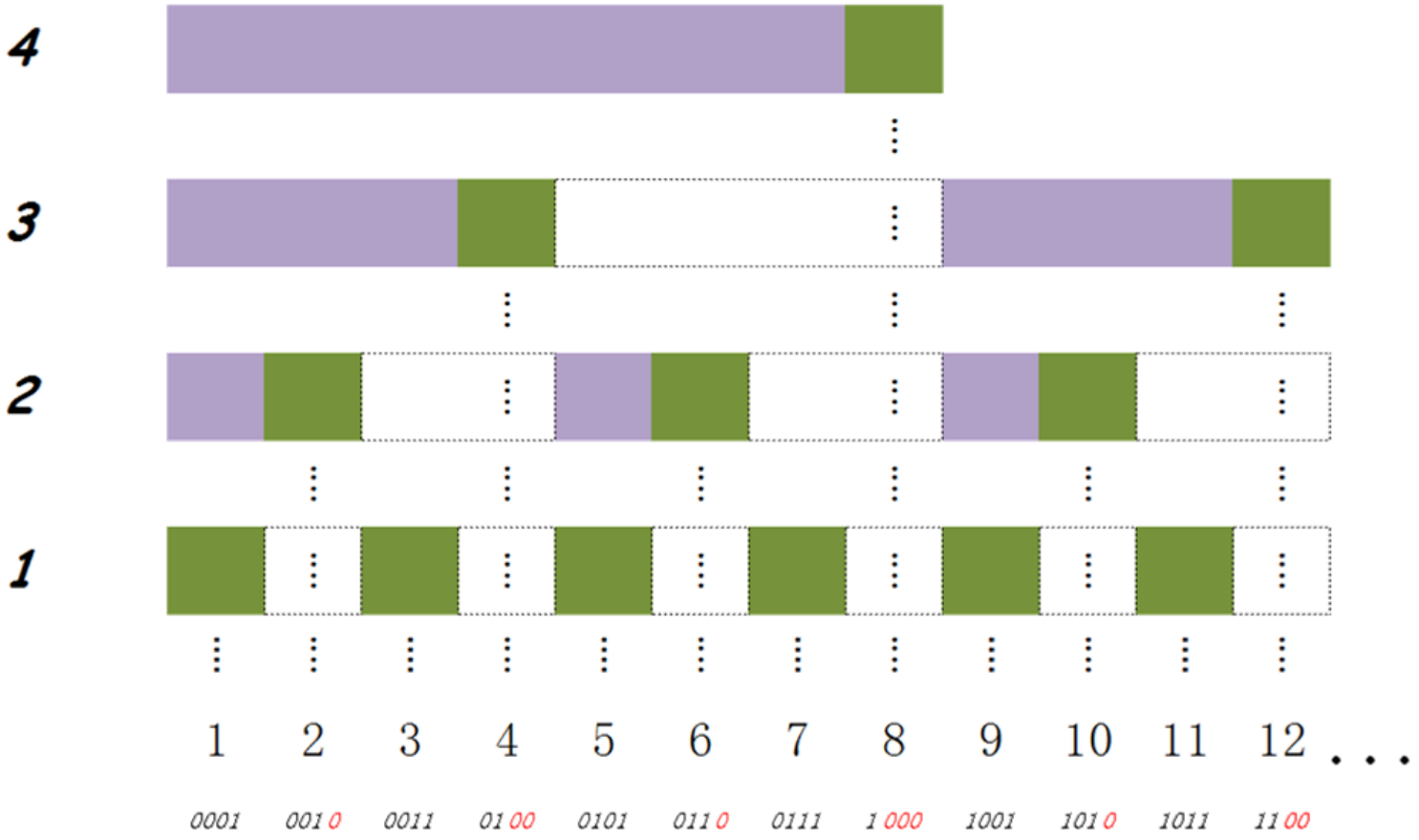
- 树状数组是一种维护前缀和的数据结构，可以实现 O(log n) 查询一个前缀的和，O(log n) 对原数列的一个位置进行修改。

- 查询: O(1)→O(log n)
- 修改: O(n)→O(log n)

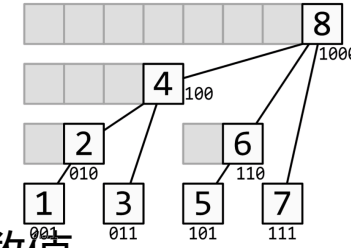
- 总体：O(n^2) → O (nlogn)

树状数组的定义

- S[N]


4
3
2
1
1 2 3 4 5 6 7 8 9 10 11 12 . . .
0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100

lowbit(i)



- lowbit(i) 函数即为 i 的二进制表示下的最低位1和后面的0构成的数值。

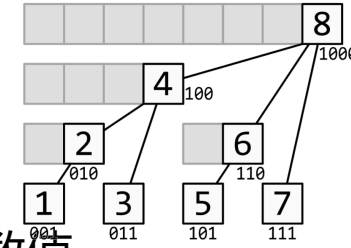
例如：

$lowbit(9) = lowbit(1001_2) = 0001_2 = 1$

$lowbit(6) = lowbit(0010_2) = 0010_2 = 2$

$lowbit(24) = lowbit(11000_2) = 01000_2 = 8$

lowbit(i)



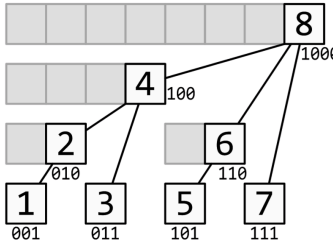
- lowbit(i) 函数即为 i 的二进制表示下的最低位1和后面的0构成的数值。

例如：

$$lowbit(9) = lowbit(1001_2) = 0001_2 = 1$$
$$lowbit(6) = lowbit(0010_2) = 0010_2 = 2$$
$$lowbit(24) = lowbit(11000_2) = 01000_2 = 8$$

- 根据位运算的规则，当我们对一个数取反(~)时，其 2 进制的 01 将被翻转。原数末尾的 0 经过翻转全部变为 1，其最低位的 1 经过翻转变成了 0。
- 此时若在取反的数上 +1，那么末尾的 1 将全部发生进位变成 0，第一个 0 会因为之前的进位变成 1。其余都保持不变。
  - 原数：　101011000
  - 取反：　010100111
  - 取反+1：　010101000
- 这时我们发现，原数与取反+1之后的数进行按位与操作，即可实现lowbit函数的功能，即：lowbit(i) = i & ((~i) + 1)
- 在计算机中，补码也是对原码进行了取反+1的操作，所以：
  - (~i) + 1 = -i
- 最终，lowbit(i) = i & (-i)

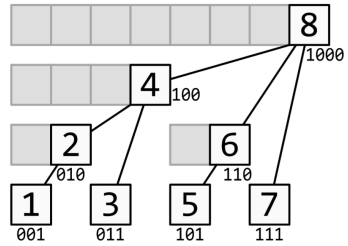
lowbit(i)



- 练习
- 求出下列函数值

- 1.lowbit(48)=
  A. 32　　B. 2　　C. 24　　D.16
- 2. lowbit(2048)=
  A. 1024　　B. 48　　C. 16　　D.2048
- 3. lowbit(22)=
  A. 22　　B. 16　　C. 4　　D.2

树状数组的定义



- 与前缀和相同的是，树状数组使用一个数组即可维护。
- 与前缀和不同的是，树状数组的一个 i 位置存储的并不是前 i 个元素的和，而是**从 i 开始，（包括 i）向前 i 的 lb_i 个元素的和。**
- 其中 **lb_i 是**的二进制表示下的最低位1和后面的0构成的数值。
  - 例如：
    - $i=9_{(1001)}$ 时，$lb_i=1_{(0001)}$；　s[9] = d[9]
    - $i=6_{(0110)}$ 时，$lb_i=2_{(0010)}$；　s[6] = d[5] + d[6]
    - $i=24_{(11000)}$ 时，$lb_i=8_{(0100)}$。　s[24] = d[17] + d[18] + … + d[24]

树状数组形式化定义

- 若 d[i] 为原数列，树状数组用来记录信息的数组为 s[1...n]，那么

$$s[x] = \sum_{i \in (x - lb_x,\ x]} d[i]$$

树状数组形式化定义

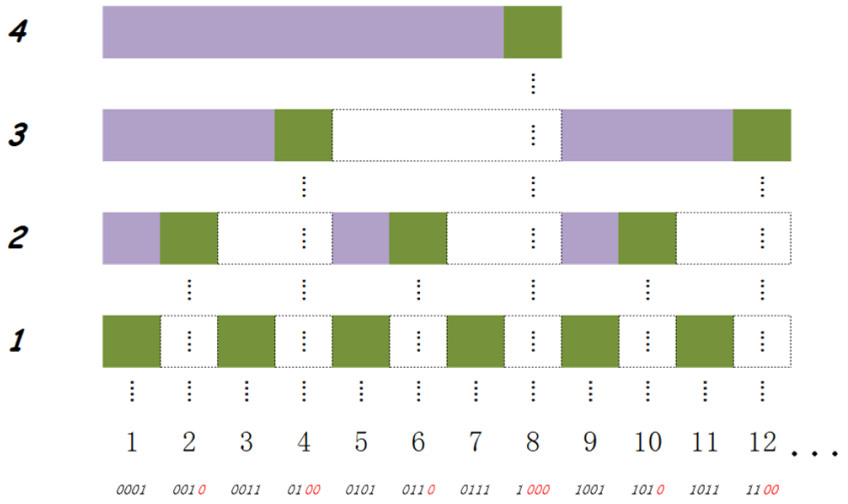
- 若 d[i] 为原数列，树状数组用来记录信息的数列为 s[1...n]，那么

$$s[x] = \sum_{i \in (x - lb_x, x]} d[i]$$

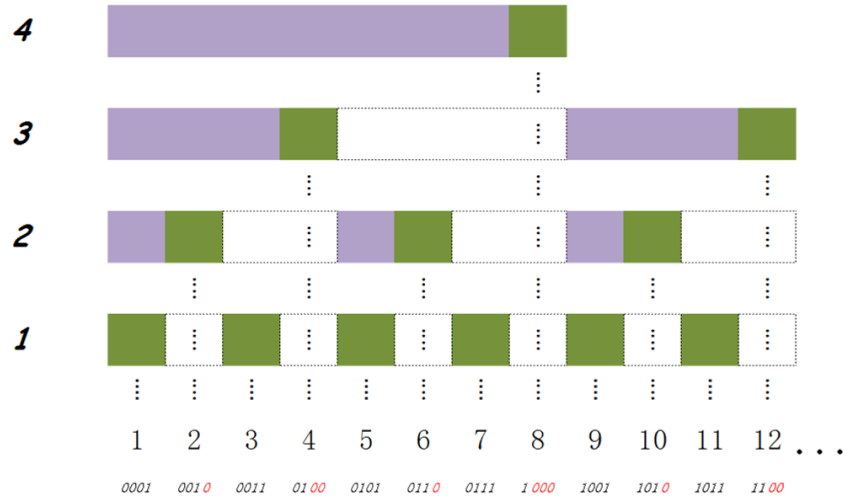
- 从 2 进制的角度考虑，若 x 的 2 进制形式表示为
  - X...X10...0
- 其中 X 表示 0 或 1 ，那么 $(x\text{-}lb_x, x]$ 的区间可以表示为
  - (X...X00...0 , X...X10...0]
- 例如 x=(10101000)，那么对应的 $(x\text{-}lb_x, x]$ 可以表示为
  - (10100000 , 10101000]

树状数组的查询

- 对于指定位置 x，要查询区间 [1, x] 的前缀和。
- 考虑到树状数组的定义，可以发现区间 [1, x] 可以使用一些 s 中现有的区间进行表示。



树状数组的查询



- 例如：
  - 对于区间 [1, 11] 可以使用 s[11] + s[10] + s[8] 进行表示
  - 写成二进制非别为：(1011)、(1010) 、(1000)
  - 其中表示的 d 的区间分别为：
    - (0000, 1000],(1000, 1010],(1010, 1011]
    - 合在一起就的(0000,1011]

    - 11 → 10 → 8
    - 11 – lowbit(11) == 10
    - 10 – lowbit(10) == 8
    - 8 – lowbit(8) == 0

树状数组的查询

- 对于确定的 x，由于 $s[x]=\sum_{i\in(x-t_x,x]} d[i]$ 若令 y = x - t_x 那么紧挨着区间 (x-t_x, x] 的上一个区间就可以使用 (y-t_y, y] 来表示。这样不断的进行迭代就能完整的表示 [1, x] 区间。
- 上述过程可以形式化的表示为：

$$\sum_{i=1}^{x} d[x]=\sum_{i=1}^{p} s[a_i]$$

- 其中，p 是 x 的 2 进制中 1 的数量，a_i 是一个下标数列，其定义如下：

$$a_i=\begin{cases} x & i=1 \\ a_{i-1}-t_{a_{i-1}} & i\geq 2 \end{cases}$$

- 由于每次执行类似于 x - tx 的操作都将使现在数字 2 进制中的 1 的数量减少 1。如此，最坏情况下需要执行的操作此时为：$\log_2 x$ 。所以此操作的时间复杂度为 O(log n)。

树状数组的查询

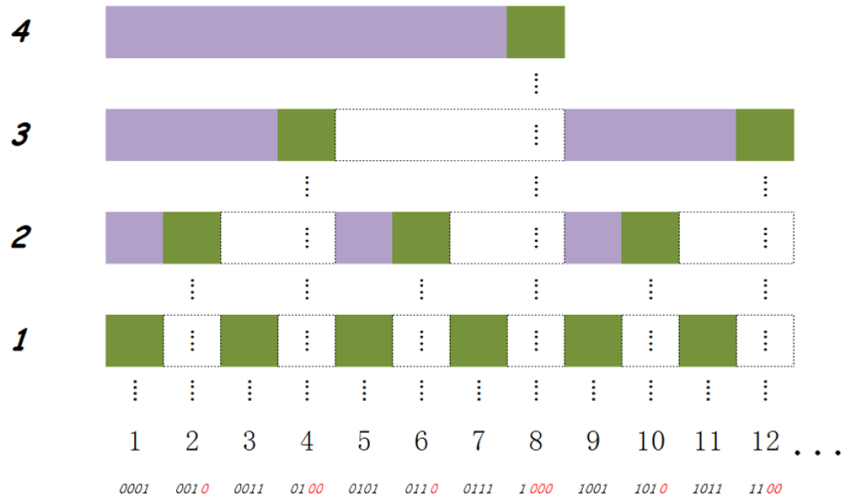
- 使用代码实现也是十分简单的：

$$a_i = \begin{cases} x & i = 1 \\ a_{i-1} - t_{a_{i-1}} & i \geq 2 \end{cases}$$

```
int s[MAXN],n;
#define lb(x) ( x & -x )
//查询
int ask(int p){
    int res=0;
    for(int i = p; i>=1 ; i -= lb(i))
        res += s[i];
    return res;
}
```

- 若是需要查询一段区间 [l, r] 的和，使用 ask(r) - ask(l - 1) 即可。

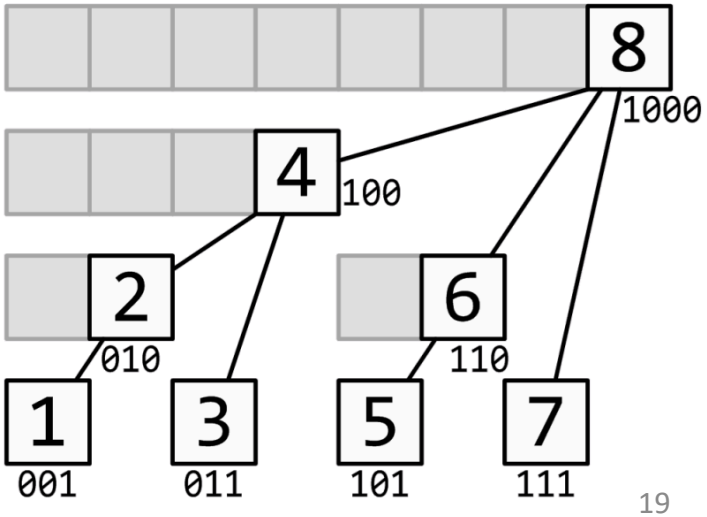
树状数组的修改



- 对于指定位置 x，要将 d[x] 的值增加 v 。（若操作为指定 d[x] 的值，计算差值即可转化为上述操作）
- 此时只需关心有哪些树状数组中位置包含 d[x] 的值，依次进行修改即可。
- 例如：
  - 把 d[5] 的值增加 v，则受到影响的点为 s[5], s[6], s[8], ...
  - 写成二进制非别为：(0101)、(0110) 、(1000)、...
- 如何找到这些点呢?
- 这就要参考树状数组的一些性质了。

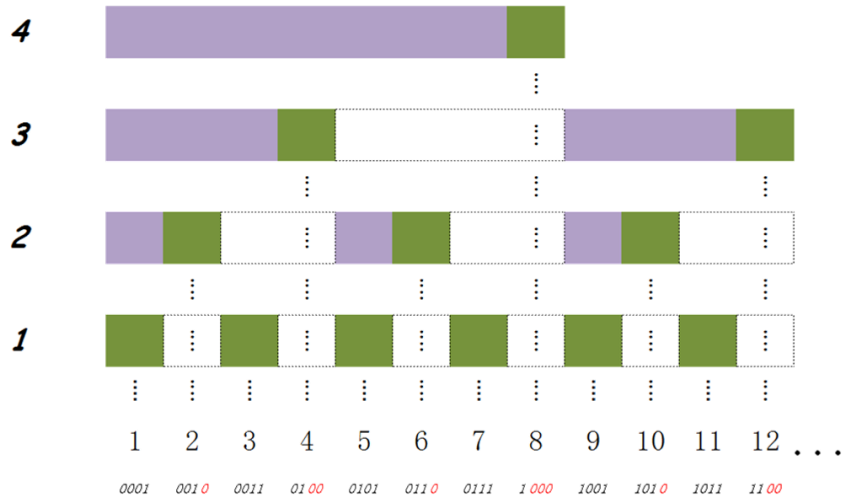
树状数组的性质

- 性质1：若当前节点为 x，且令 x + lb_x 为其父节点，则树状数组将形成一个树形结构。
- 性质2：节点 x 记录区间 (x - lb, x] 的信息，其子节点所记录的区间是 (x - lb_x, x] 的子集，且不会相互覆盖。
- 性质3：节点 x 记录的区间为节点 y 记录区间的子集，当且仅当节点 y 是节点 x 的祖先节点。



注：证明这些性质过程较为复杂，不是本课程的重点，有兴趣的同学可以参阅相关资料（https://zhuanlan.zhihu.com/p/297885717）或阅读提出者原论文 （Fenwick P M . A new data structure for cumulative frequency tables[J]. Soft.pract.exp, 2010, 24(3):327-336.）

树状数组的修改



- 对于指定位置 x，要将 d[x] 的值增加 v 。（若操作为指定 d[x] 的值，计算差值即可转化为上述操作）
- 此时只需关心有哪些树状数组中位置包含 d[x] 的值，依次进行修改即可。
- 如何找到这些点呢?
- 根据性质 3 可知，只需要访问 x 所有祖先节点即可。若这些节点的编号序列为 b_i，则 b_i 可形式化的表示为：

$$b_i = \begin{cases} x & i = 1 \\ b_{i-1} + t_{b_{i-1}} & i \geq 2 \end{cases}$$

- 当 b_i 大于原数列 d 的长度时将没有意义，且每次迭代都增加了 lb_x，则 x 的末尾每次都会至少多出一个 0，当末尾 0 的数量大于 $\log_2 n$ 时，操作将失去意义，所以此操作的时间复杂度为 O(log n)。

树状数组的修改

- 使用代码实现也是十分简单的：

$$b_i = \begin{cases} x & i = 1 \\ b_{i-1} + t_{b_{i-1}} & i \geq 2 \end{cases}$$

```
int s[MAXN],n;
#define lb(x) ( x & -x )
// 修改
void upd(int p,int v){
    for(int i=p;i<=n;i+= lb(i))
        s[i]+=v;
}
```

树状数组的应用

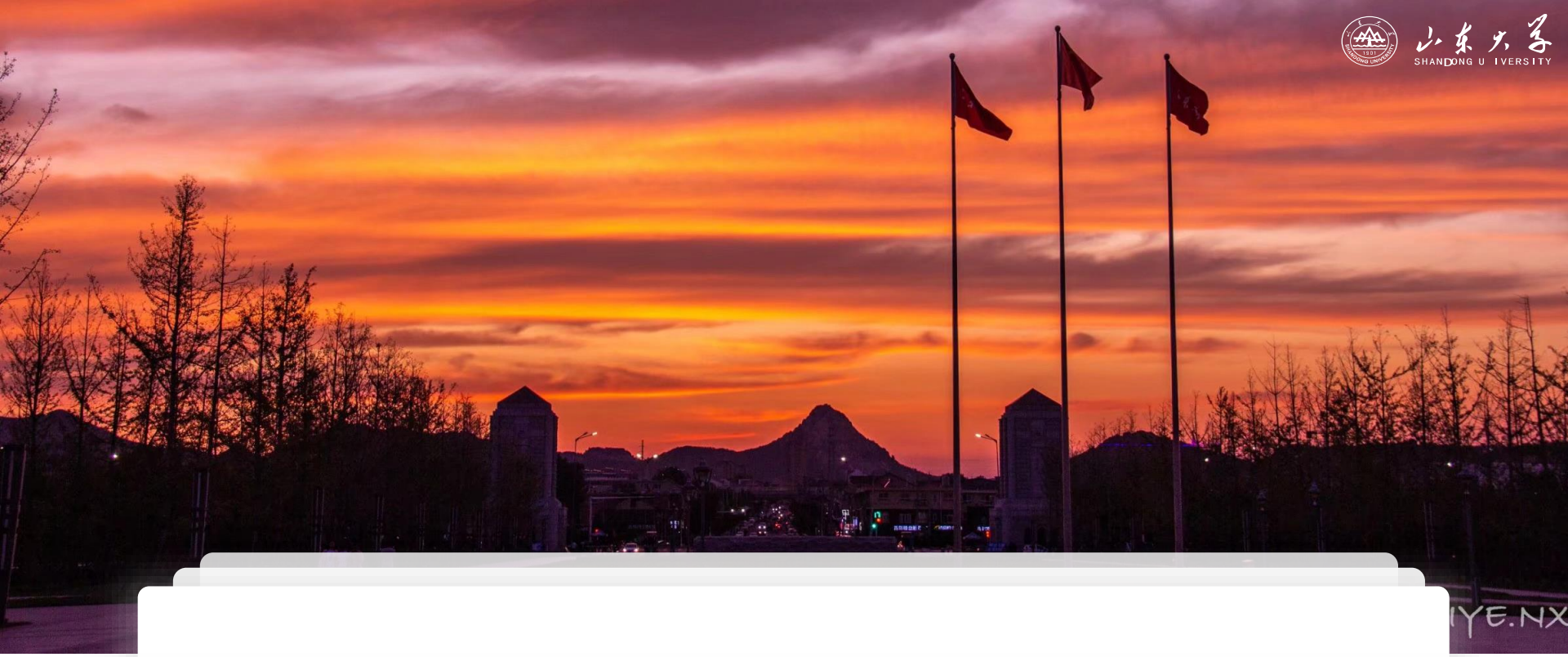
- 求逆序对（二维偏序问题）
- 给定一个序列 $a_1, a_2, a_3, \ldots, a_n$，如果存在 $i < j$ 且 $a_i > a_j$，那么我们称之为逆序对，求给定序列中逆序对的数目。其中 $1 \le a_i, n \le 10^5$。
- 样例：输入：[4 1 3 2 5] 输出：4
- 解：　(4,1) (4,3),(4,2) (3,2)
- 对于任意一个 a_i，统计在其之前且大于 a_i 的数的数量，即为以 a_i 结尾的逆序对的数量。
- 只要枚举 $i \in [1, n]$ 分别计算，再求和，即为所求答案。
- 使用树状数组维护桶，记录每个数字出现的数量。从 a_1 到 a_n 依次插入，并在每个数字插入前，查询大于当前数字的数量之和。

- 下面是 live coding 环节。

树状数组的应用

[4 1 3 2 5]

D[1]	D[2]	D[3]	D[4]	D[5]	...



2 线 段 树

Segment Tree

引入

- 树状数组已经可以做到在含有单点修改的前提下，实现区间和的维护，其实现思路是利用两次查询的**前缀和相减**得到指定区间的和。

- 但是有些信息却是无法相减的，例如最小值。
  - 例如：无法通过 区间 $[1, l-1]$ 的最小值与区间 $[1, r]$ 的最小值求得区间 $[l, r]$ 的最小值。

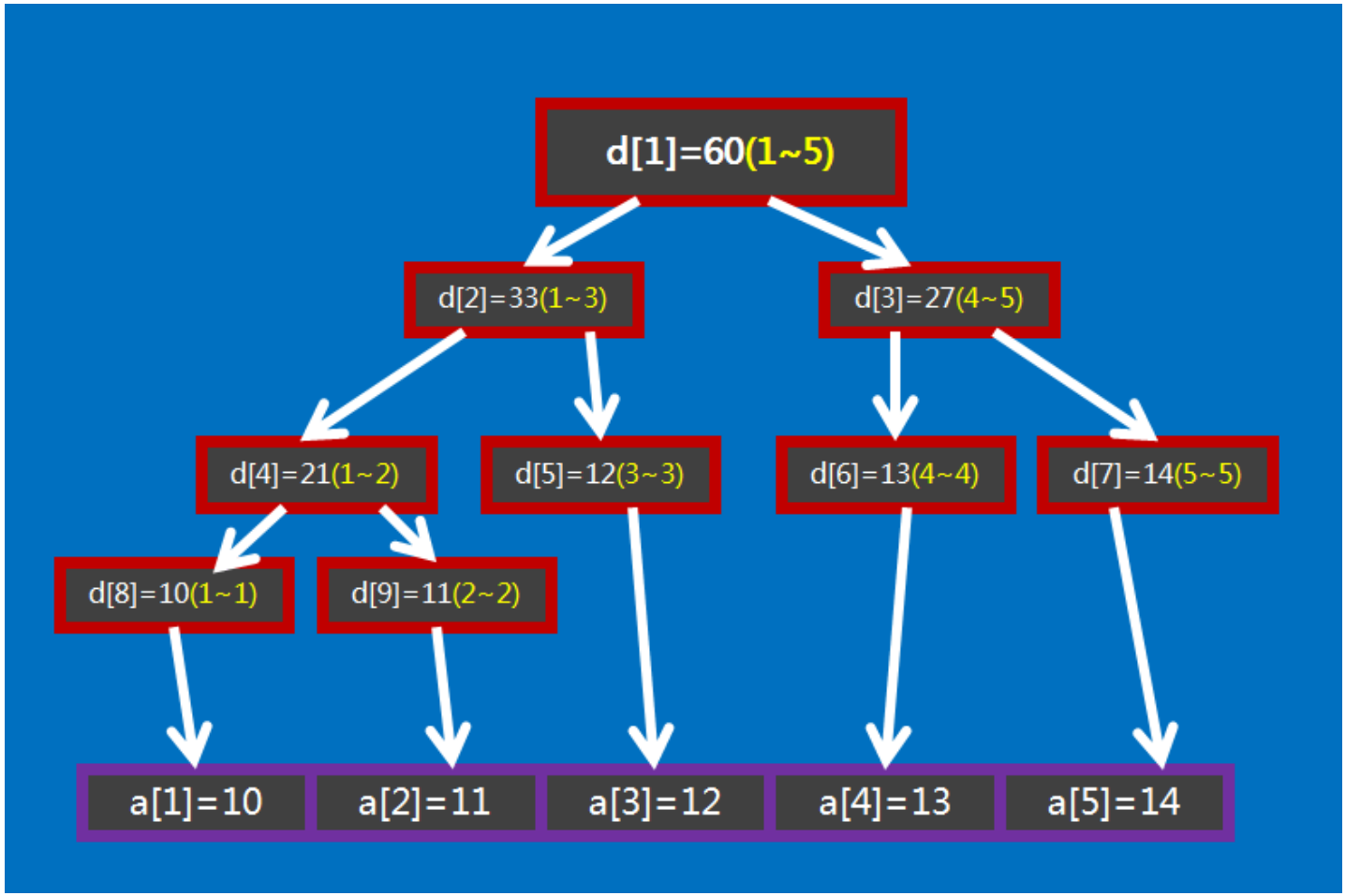
- 这时就需要使用线段树对相关的信息进行维护了。

线段树的简介

- 线段树的功能非常强大，且拓展性也很强，通过各种操作的组合可以实现在 O(log n) 的时间内**维护区间信息**。
- 支持单点修改以及一些特殊的区间整体修改。
  - 具体来说支持单点修改、区间修改、区间查询（区间求和，求区间最大值，求区间最小值）等操作。
- 线段树维护的信息，需要满足**可加性**，即能以可以接受的速度合并信息和修改信息。
  - 例如：若已知区间 [A, B] 是由区间 [A, C] 与区间 [C, B] 组合而来，此时当分别已知 [A, C] 与 [C, B] 的信息时，可以计算出 [A, B] 的信息。显然，区间和/积，区间最值都满足这个条件；但是区间的中位数就不满足这个条件。

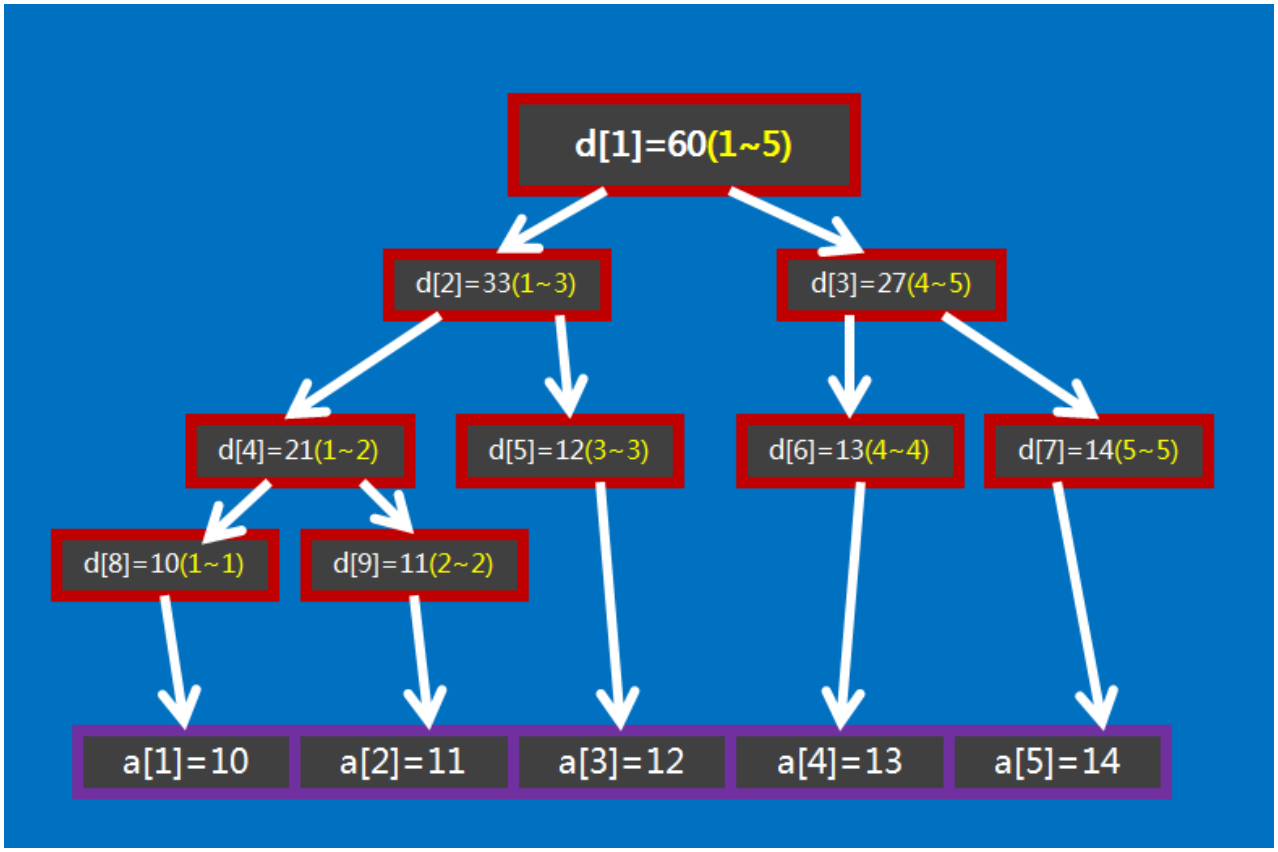
线段树的基本结构

- 线段树将每个长度不为 1 的区间划分成左右两个区间递归求解，把整个线段划分为一个树形结构，通过合并左右两区间信息来求得该区间的信息。这种数据结构可以方便的进行大部分的区间操作。

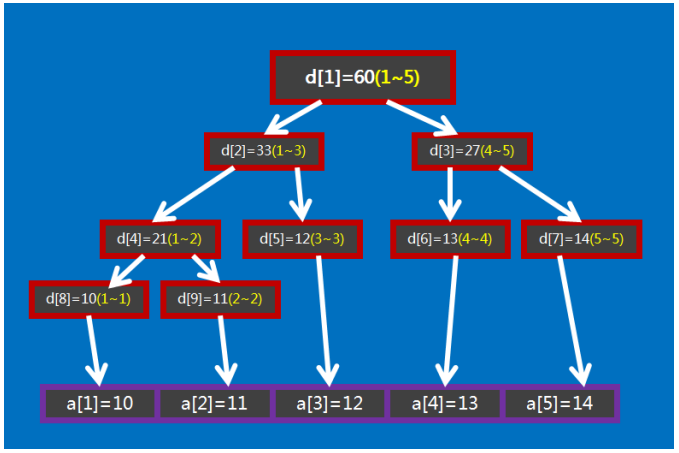


线段树的基本结构

- 图中 d_1 表示根节点，紫色方框是数组 a，红色方框是数组 d，红色方框中的括号中的黄色数字表示它所在的那个红色方框表示的线段树节点所表示的区间.
  - 如 d_1 所表示的区间就是 [1,5]（a_1,a_2, … ,a_5）
  - 即 d_1 所保存的值是 $a_1+a_2+ … +a_5$
  - $d_1=60$ 表示的是 $a_1+a_2+ … +a_5=60$。

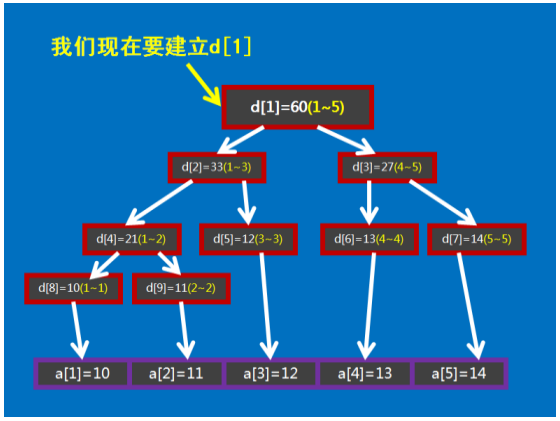


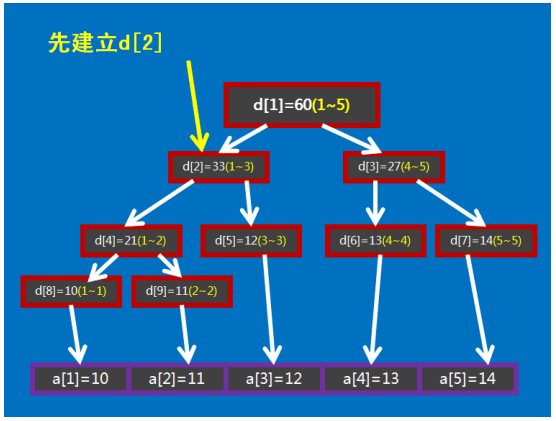
线段树的基本结构

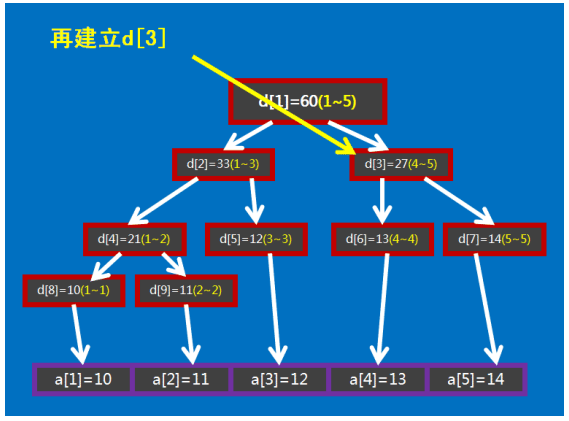


- 线段树的根节点维护 [1, n] 区间的信息
- 若当前节点维护的区间为 [l, r]，定义 $m = \left\lfloor \frac{l+r}{2} \right\rfloor$
  - 其左孩子维护的区间为[l, m]
  - 其右孩子维护的区间为[m+1, r]
- 若当前节点维护的区间长度为 1，则其值即为线段树的初值

线段树的建立

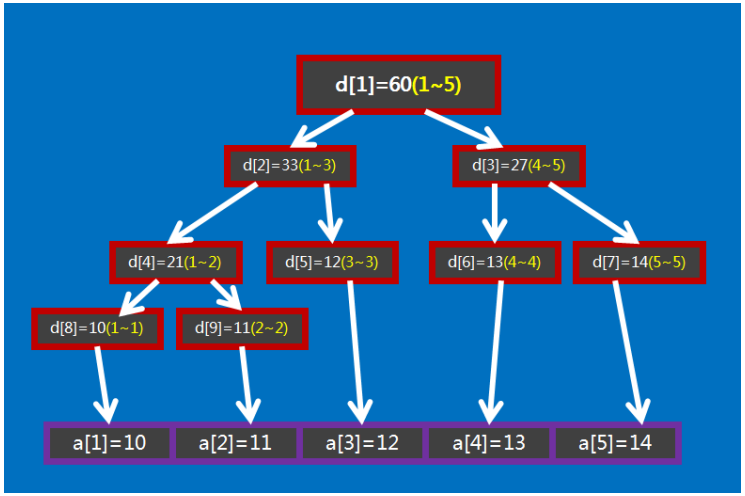






- 线段树使用递归的方法建立，由于一个区间的值取决于其左右孩子的值，所以先递归建立左右孩子，通过合并左右区间的答案得到当前区间的答案。
- 可以使用数组模拟建立线段树，其数组的使用方式与完全二叉树相同。
  - 若一个节点的标号为 x：
    - 其左孩子的标号为 x * 2；
    - 其右孩子的标号为 x * 2 + 1。
- 当区间长度为 1 时，到达递归的边界，此时采用初值更新区间的值并返回。

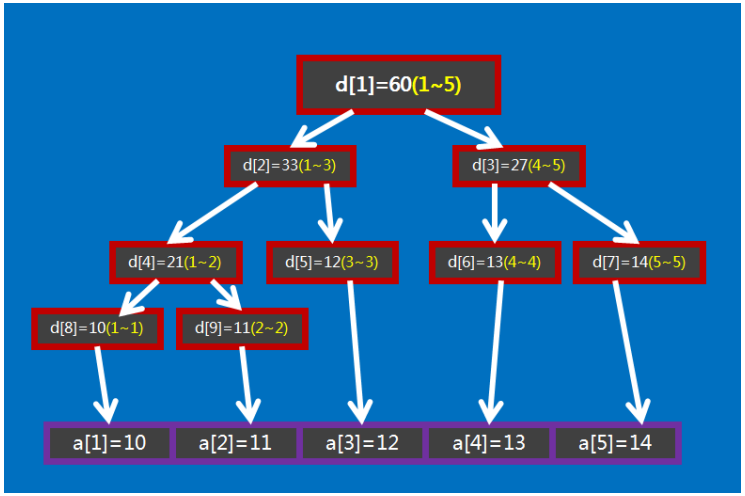
线段树的建立



- 其代码实现如下：

```
const int MAXN = 100010;
// 维护区间和
int d[MAXN << 2], a[MAXN];
void build(int x, int l, int r){
    if(l == r){
        d[x] = a[l];
        return;
    }
    int m = (l + r) / 2;
    build( x: x * 2, l, m);
    build( x: x * 2 + 1, l: m + 1, r);
    d[x] = d[x * 2] + d[x * 2 + 1];
}
```

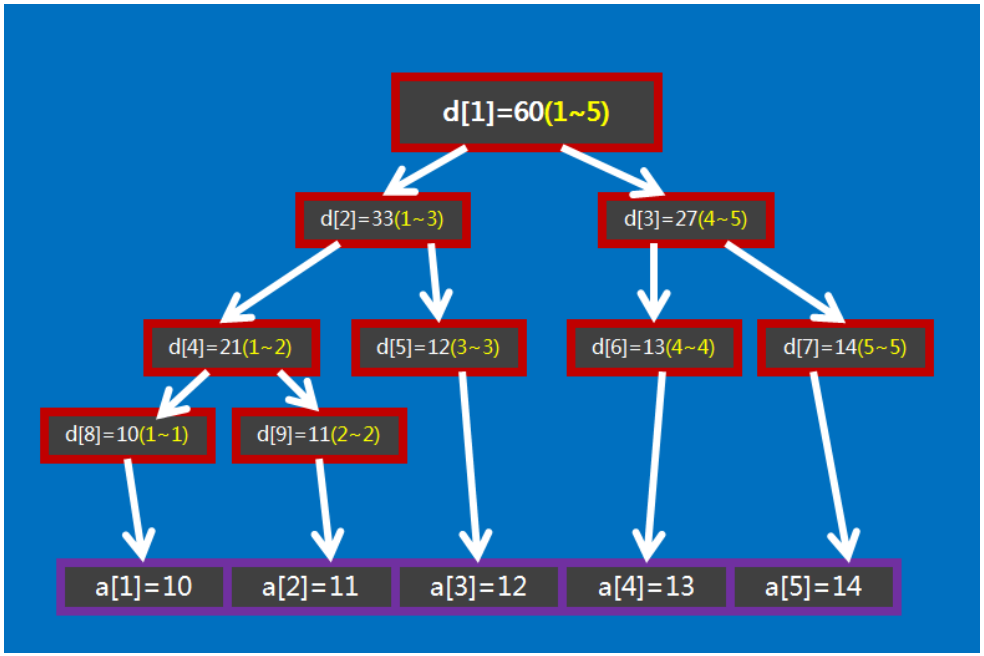
线段树的单点修改



- 找到相应的叶子节点，更新其值，在返回时维护经过的节点值即可。
- 代码如下：

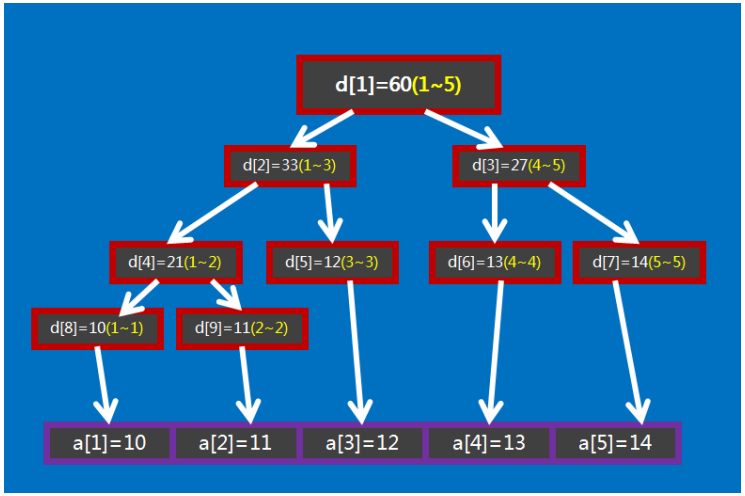
```
// 将 p 位置的值增加 v
int p, v;
void upd(int x, int l, int r) {
    if (l == r) {
        d[x] += v;
        return;
    }
    int m = (l + r) / 2;
    if (p <= m) upd( x: x * 2, l, m);
    else upd( x: x * 2 + 1,  l: m + 1, r);
    // 由于孩子被更新，当前节点也要更新
    d[x] = d[x * 2] + d[x * 2 + 1];
}
```

线段树的区间查询



- 以上面这张图为例，如果要查询区间 [1,5] 的和，那直接获取 d_1 的值（60）即可。
- 如果要查询的区间为 [3,5]，此时就不能直接获取区间的值，但是 [3,5] 可以拆成 [3,3] 和 [4,5]，可以通过合并这两个区间的答案来求得这个区间的答案。
- 一般地，如果要查询的区间将其拆成最多为 O(log n) 个**极大**的区间，合并这些区间即可求出 [l,r] 的答案。

线段树的区间查询



- 代码如下：

```
// 查询 [p1, p2] 的答案
int ask(int x, int l, int r, int p1, int p2){
    // 要查的区间刚好是当前区间，直接返回
    if(l == p1 && r == p2) return d[x];
    int m = (l + r) / 2;
    // 情况1 要查询的区间完全在左边的区间
    if(p2 <= m) return ask( x: x * 2, l, m, p1, p2);
    // 情况2 要查询的区间完全在右边的区间
    else if(p1 > m) return ask( x: x * 2 + 1, l: m + 1, r, p1, p2);
    // 情况3 要查询的区间分布在左右两边，分别计算，合并后作为答案
    else{
        int lch_val = ask( x: x * 2, l, m, p1, m);
        int rch_val = ask( x: x * 2 + 1, l: m + 1, r, p1: m + 1, p2);
        return lch_val + rch_val;
    }
}
```

线段树的应用

- 这一部分当中需要重点掌握的应用是使用线段树维护区间的最小值并支持单点修改，这将在本课程后续的“动态规划优化”中将得到应用。

- 下面是 live coding 环节。

3

*带有标记的线段树

Segment Tree with lazy-tag

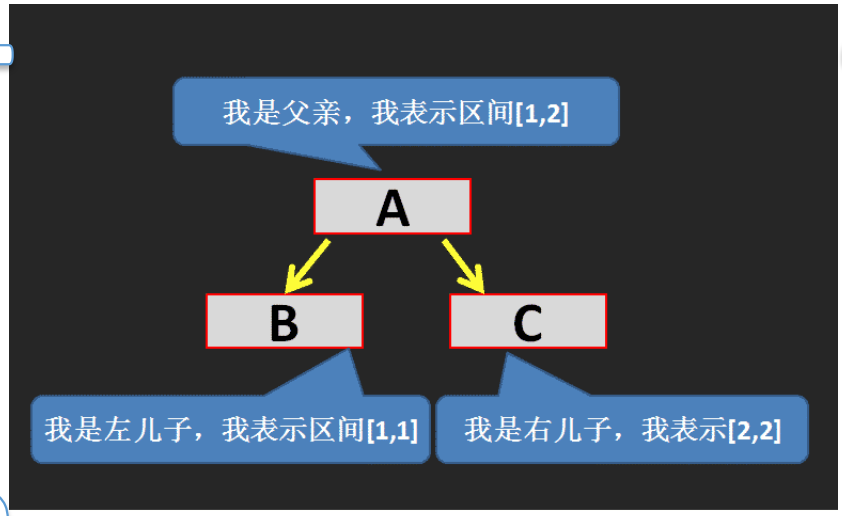
关于带有标记线段树

- 线段树曾经出现在 CSP 的第五题（202012-5）中，但是要想在此题中使用线段树，则需要使用线段树的区间修改操作，这需要在维护答案的同时顺带维护区间修改的标记。
- 根据此部分的掌握情况可以在 202012-5 分别得到 40,60,80 分。
  - 40 分为掌握区间加法标记
  - 60 分为掌握区间加法与乘法混合标记
  - 80 分为掌握区间加法与乘法混合标记以及技巧性的其他标记
- 此题拿到 100 分还需要掌握使用动态分配内存的方法生成线段树。

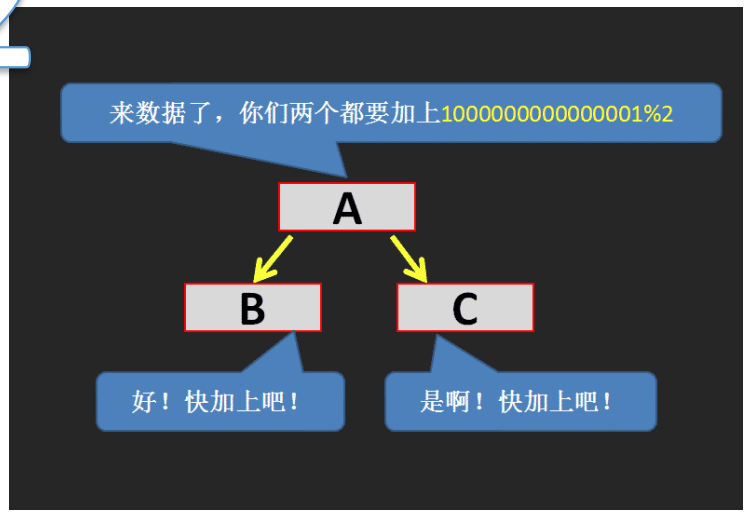
- 因为这一部分难度较大，所以设定为**选讲**，同学们可以根据自己的情况，选择是否掌握本部分。期末考试以及当堂测试**不会涉及该部分内容**。

感性理解区间懒标记

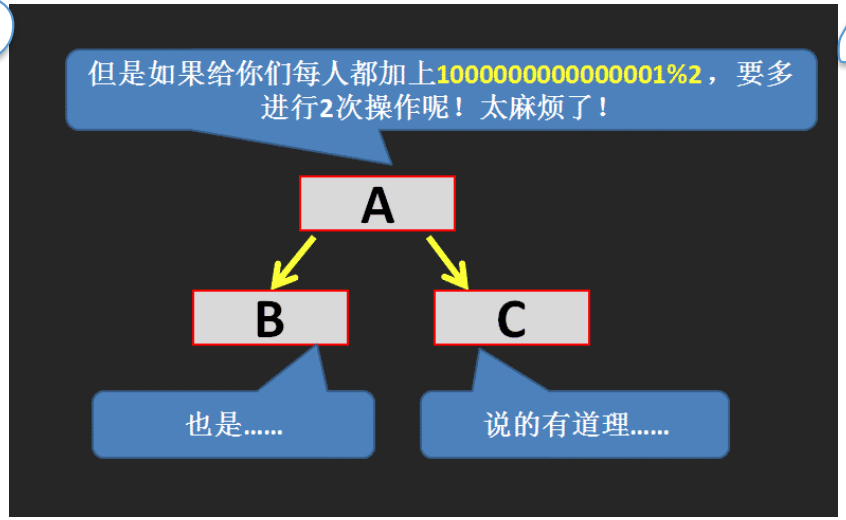
1



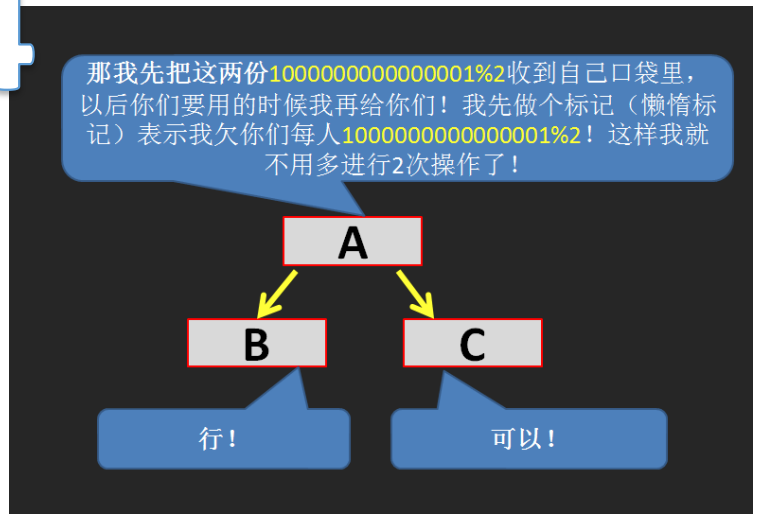
2



3

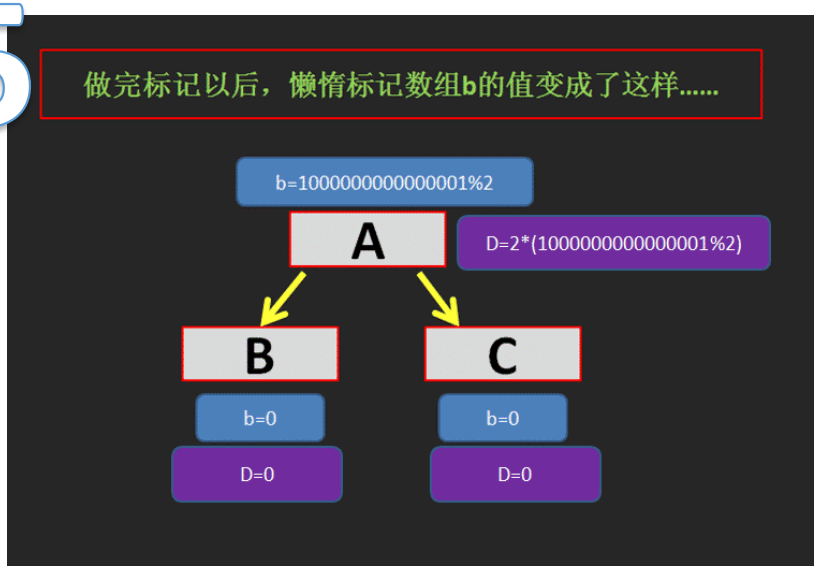


4

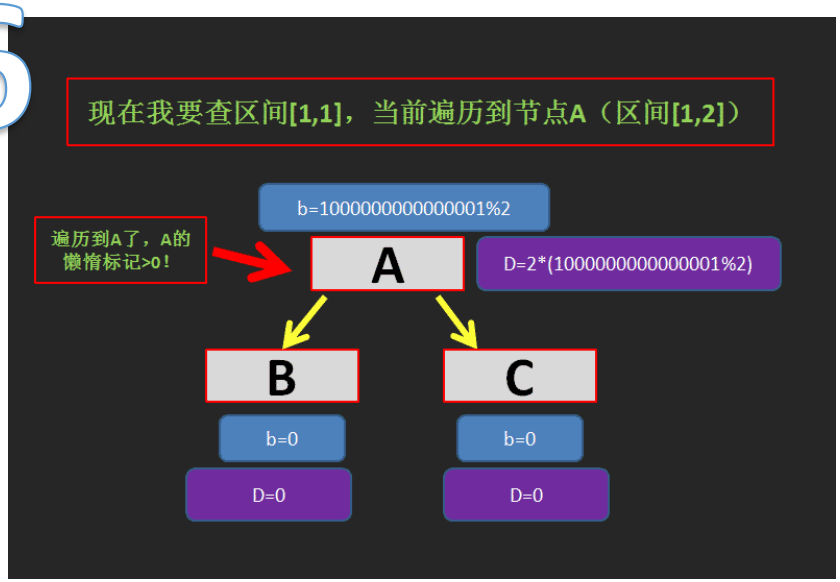


感性理解区间懒标记

5

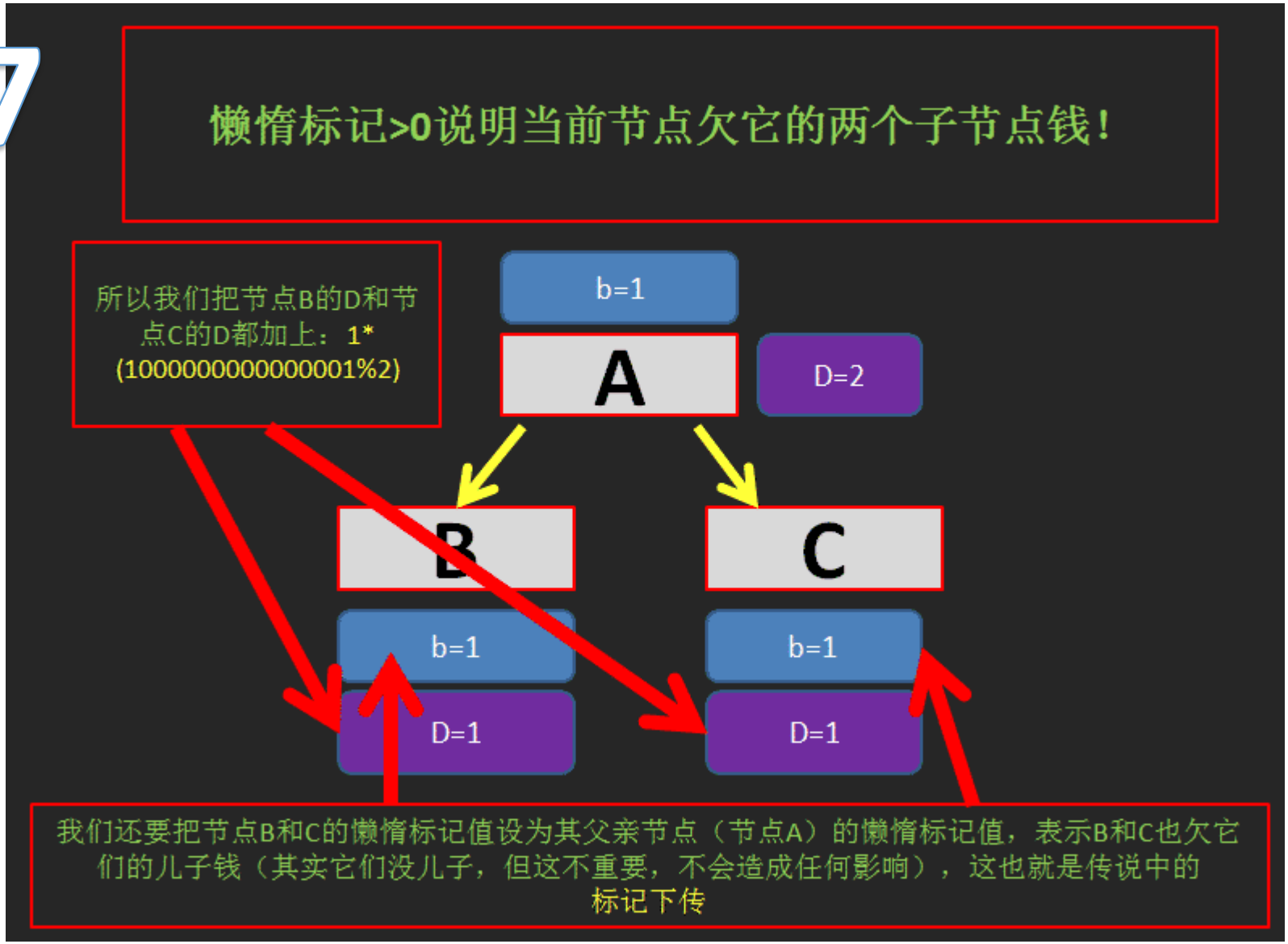


6



感性理解区间懒标记

7

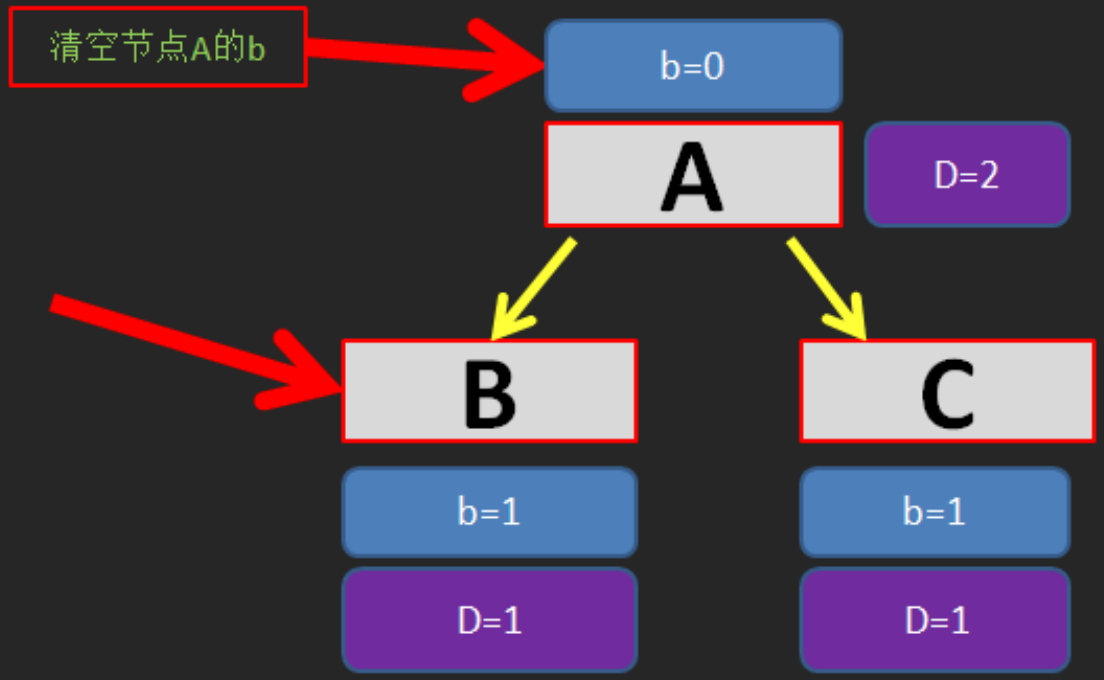

懒惰标记>0说明当前节点欠它的两个子节点钱！
所以我们把节点B的D和节点C的D都加上：1*(1000000000000001%2)
b=1
A
D=2
B
C
b=1
b=1
D=1
D=1
我们还要把节点B和C的懒惰标记值设为其父亲节点（节点A）的懒惰标记值，表示B和C也欠它们的儿子钱（其实它们没儿子，但这不重要，不会造成任何影响），这也就是传说中的标记下传

感性理解区间懒标记

8

继续遍历，向下找。我们发现节点B表示的区间[1,1]包含在我们要查询的区间[1,1]中，所以返回值加上B的值（B的D，即1）。而节点C与查询的区间[1,1]并没有相交，所以不去管节点C。返回值（就是我们要找的答案）

=0+1=1



理性认识区间懒标记

- 要想在一个区间打上一个标记，需要满足以下条件：
  - **标记可以合并**：在已有标记的基础上，再进行操作还能继续维护标记。
  - **可以维护当前区间的答案**：在完成标记之后，可以快速的计算出更新之后当前区间的答案。
    - 以区间加法为例，若对 [1, 5] 区间每一个数增加 2，当前维护的是区间和，可知区间长度为 5，所以整体的区间和应增加 2 * 5 = 10。

区间加法标记

```cpp
const int MAXN = 100010;
// 维护区间和
int d[MAXN << 2], a[MAXN], n;
// 区间和标记
int sum_tag[MAXN << 2];

// 使用宏定义简化代码书写
#define lch (x << 1)
#define rch (x << 1 | 1)
#define mid ((l + r) >> 1)
#define tl lch, l, mid
#define tr rch, mid+1, r

// 初始化标记
void init_tag(int x){
    sum_tag[x] = 0;
}

// 初始化函数
void build(int x, int l, int r) {
    init_tag(x);
    if (l == r) {
        d[x] = a[l];
        return;
    }
    build(tl), build(tr);
    d[x] = d[lch] + d[rch];
}
```

```cpp
// 更新当前 x 节点
void add(int x, int l, int r, int v){
    d[x] += v * (r - l + 1);
    sum_tag[x] += v;
}

// 标记下放
void down(int x, int l, int r){
    add(tl, sum_tag[x]), add(tr, sum_tag[x]);
    init_tag(x);
}

// 修改函数
int v;
void upd(int x, int l, int r, int p1, int p2){
    if(l == p1 && r == p2){
        add(x, l, r, v);
        return;
    }
    down(x, l, r);
    if(p2 <= mid)  upd(tl, p1, p2);
    else if(p1 > mid) upd(tr, p1, p2);
    else upd(tl, p1, p2: mid), upd(tr, p1: mid+1, p2);
    d[x] = d[lch] + d[rch];
}
```

加法与乘法标记混合

- 若对区间的操作处理支持加法外，还要支持对区间内的数同时乘以固定值，这时需要注意：
  - 乘法标记对加法标记的影响
  - 加法标记对乘法标记的影响
  - 下放标记时，顺序对结果的影响

- 乘法标记对加法标记的影响
  - 若当前已有加法标记，由于乘法在后，所以加法标记也应该乘上乘数
- 加法标记对乘法标记的影响
  - 若当前已有乘法标记，在乘完之后，可以视为正常在其后追加，不会对乘法标记造成任何影响。
- 下放标记时，顺序对结果的影响
  - 若是先下放加法标记，再下放乘法标记，将导致标记中的数进行了两次乘法，这显然是错的
  - 所以应该先下放乘法标记，再下放加法标记。

加法与乘法标记混合

```
const int MAXN = 100010;
// 维护区间和
int d[MAXN << 2], a[MAXN], n;
// 区间和标记
int sum_tag[MAXN << 2], mul_tag[MAXN << 2];

// 使用宏定义简化代码书写
#define lch (x << 1)
#define rch (x << 1 | 1)
#define mid ((l + r) >> 1)
#define tl lch, l, mid
#define tr rch, mid+1, r

// 初始化标记
void init_tag(int x){
    sum_tag[x] = 0;
    mul_tag[x] = 1;
}

// 初始化函数
void build(int x, int l, int r) {
    init_tag(x);
    if (l == r) {
        d[x] = a[l];
        return;
    }
    build(tl), build(tr);
    d[x] = d[lch] + d[rch];
}
```

与只有加法相比，只有红框部分不同

```
// 更新当前 x 节点（加法）
void add(int x, int l, int r, int v){
    d[x] += v * (r - l + 1);
    sum_tag[x] += v;
}

// 更新当前 x 节点（乘法）
void mul(int x, int v){
    d[x] *= v, mul_tag[x] *= v;
    sum_tag[x] *= v;
}

// 标记下放
void down(int x, int l, int r){
    mul(lch, mul_tag[x]), mul(rch, mul_tag[x]);
    add(tl, sum_tag[x]), add(tr, sum_tag[x]);
    init_tag(x);
}

// 修改函数
int v;
void upd(int x, int l, int r, int p1, int p2){
    if(l == p1 && r == p2){
        add(x, l, r, v);
        return;
    }
    down(x, l, r);
    if(p2 <= mid)  upd(tl, p1, p2);
    else if(p1 > mid) upd(tr, p1, p2);
    else upd(tl, p1, mid), upd(tr, mid+1, p2);
    d[x] = d[lch] + d[rch];
}
```

更多的操作

- 更多的标记基本都与上述的两种相似。在此不再做过多的讨论。

- 更多关于 202012-5 的方法，可以自学动态开点线段树，或参考下面的代码。
  - https://paste.ubuntu.com/p/9X8fPbg2sS/

- 其实关于线段树的操作很有很多很多，例如：（以下**都不要求掌握，感兴趣可以自学**）
  - zkw 线段树 https://www.luogu.com.cn/blog/82152/Introduction-of-zkwSegmentTree
  - 权值线段树与主席树 https://www.luogu.org/blog/your-alpha1022/WeightSegmentTree-ChairmanTree
  - 线段树合并 https://www.luogu.org/blog/styx-ferryman/xian-duan-shu-ge-bing-zong-ru-men-dao-fang-qi
  - 线段树分治 https://www.luogu.org/blog/foreverlasting/xian-duan-shu-fen-zhi-zong-jie
  - 李超线段树 https://www.luogu.com.cn/blog/infinity-dimension/Li-Chao-Tree

总结


树形数据结构与应用
树状数组
修改操作
查询操作
应用
线段树
构造
修改操作
区间查询操作
应用
*带有标记的线段树（选讲）

感谢收听

Thank You For Your Listening