基础数据结构专题

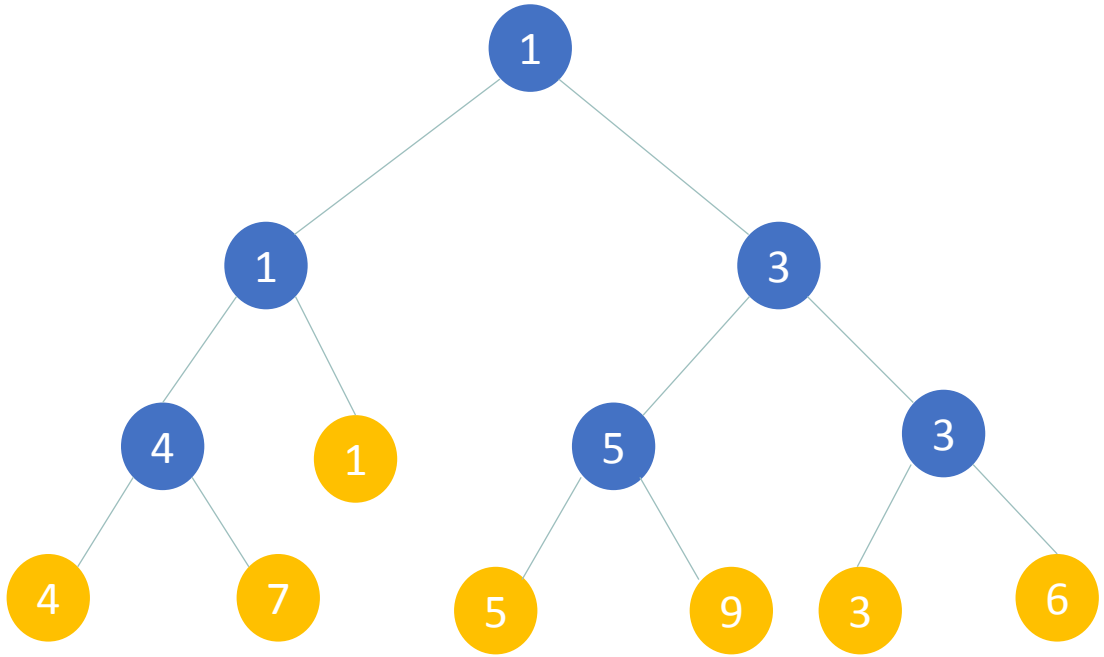
湖南师大附中 许力

数据结构在OI中处于什么位置？

- 问题：从数组a[n]中询问某个区间内的最值。数组中的元素可能不断被更新

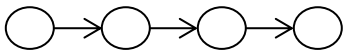
  n≤10^6

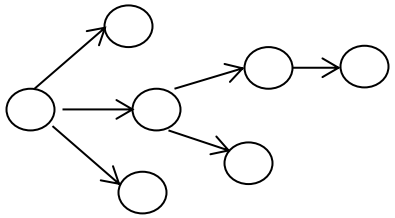
- 样例 4 7 1 5 9 3 6

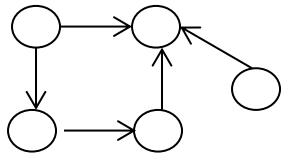


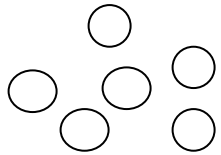
数据的逻辑结构

- 线性结构
- 树
- 图
- 集合









数据的存储结构

- 顺序存储：逻辑上相邻的元素，物理上仍然相邻
- 我们最早接触，应用最为广泛的数据结构——数组就是典型，数组不仅可以顺序访问，还可以通过下标实现各种复杂访问

- 链式存储：逻辑上相邻的元素物理上不一定相邻，相互关系通过地址确定。链式存储更多通过指针实现

栈

- 后进先出（LIFO）的线性数据结构

- 其模型类似于摆放在桌上的一叠书，后放上的书处于最上方，可被直接取走；先放的书被压在下方，需要先拿走放于其上的书，才能被取走

数组模拟实现栈

- 记录 top 表示栈顶位置

```
int stk[N], top = 0;    //栈顶初始 0

stk[++top] = x;     //元素 x入栈

return stk[top];  //返回栈顶元素

stk[--top];    //弹出栈顶元素

while (top > 0)     //栈不空
```

stack实现栈

```
stack<int> s;     //定义栈
s.push(x);    //元素 x入栈
s.top();    //返回栈顶元素
s.pop();    //弹出栈顶元素
s.size();   //返回当前栈长度
while (!s.empty())    //栈不空
```

栈的应用

- 可以利用栈来检验括号的匹配，也可以用来检查回文

计算表达式

- 我们熟知的算术表达式：

8 - (3 + 2 * 6) / 5 + 4

其实是中缀表达式，因为运算符在两个运算数的中间

- 但是对于程序而言，处理后缀表达式形态的算术式效率更高：

8326 * + 5 / - 4 +

因为借助于栈，可以从左至右遍历表达式，发现是数字就入栈，发现是运算符就从栈顶连续取出两个数字进行运算，运算结果再入栈

- 后缀表达式无须考虑运算优先级，自然也无须括号

中缀表达式转后缀表达式

1. 遇到数字直接输出
2. 遇到运算符和左括号压入符号栈
3. 遇到右括号则一直弹栈输出直到遇到左括号
4. 压入运算符时，如果栈顶符号不为括号且运算符优先级不小于当前运算符，则弹出

栈顶运算符并输出。直到栈空或者遇到左括号或优先级低的运算符时停止弹栈，压入当前的运算符

5. 读入结束后弹出栈内所有运算符

中缀表达式转后缀表达式

- 试一试：

(8 + (7 - 6) + 5) + 4 * 3 / 2 * (1 + 9)

8 7 6 - + 5 + 4 3 * 2 / 1 9 + * +

回文串

- 一段长度未知的字符串（最长10w），判断其是否为回文串，若是则输出“Yes”，否则输出“No”

sample1	sample2
Input Ahaha Output No	Input To be eb oT Output Yes

队列

- 先进先出（FIFO）的线性数据结构
- 其模型类似于食堂里买饭的队伍，先来的同学先买饭并离开；后来的同学只能排在队尾并依序前进
- 当然在具体代码实现的时候，队伍（数组）是不动的，那样复杂度太高；动的是首尾指针

数组模拟实现队列

- 头指针 head 指向队首元素，尾指针 tail 指向队尾元素

```
int que[N], head = 0, tail = 0;    //队列初始化

que[tail++] = x;    //元素 x入队

return que[head];    //返回队首元素

que[head++];    //队首元素出队

while (head < tail)    //队列不空
```

数组模拟实现队列

- 不过和栈不一样的是：这样处理，长度为 n 的数组很有可能无法模拟长度为 n 的队列

- 解决办法是首尾循环利用数组：

```
que[tail++] = x;     //元素 x入队
if (tail == N) tail = 0;

que[head++];         //队首元素出队
if (head == N) head = 0;
```

- 当然这样有可能 head 出现在 tail 的右边，队列判空的条件得改一改

数组模拟实现队列

- 判断队列不空的依据是 head 和 tail 的位置不重叠

```
while (head != tail)     //队列不空
```

数组模拟实现队列

- 还有一种做法是定义数组元素为结构体：包含数据域本身及首尾指针

```
struct que
{
    int data[N];
    int head;
    int tail;
 } q;              //定义队列

q.head = 0; q.tail = 0;    //队列初始 0

q.data[q.tail++] = x;     //元素 x入队

q.data[q.head];     //取队首元素

q.head++;          //队首元素出队

while (q.head < q.tail)     //队列不空
```

queue实现队列

- 不过需要注意 queue 的常数偏大

```
queue<int> q;      //定义队列
q.push(x);      //元素 x入队
q.front();      //返回队首元素
q.pop();        //弹出队首元素
q.size();       //返回队列当前长度
while (!q.empty())      //队列不空
```

优先队列

- 每个元素附带一个优先权值
- 根据这个优先权值来决定进出队的先后顺序
- 优先队列中的元素始终保持有序（默认从大到小），即便有元素插入或者删除也如此

priority_queue实现优先队列

```
priority_queue<int> q;    //定义，默认从大到小

priority_queue<int,vector<int>,greater<int> > q;    //定义，从小到大

q.push(x);    //变量x进队

q.top();    //取队首元素

q.pop();    //队首元素出队

q.size();    //队列长度

!q.empty();    //队列不空
```

单调队列

- 队列中元素值单调不增/不减的子序列即为单调队列，可以两头出队
- 单调队列不光要保存元素值，还要记录元素的索引位置
- 以（3，4，2，6，5，9）为例

–2，5，9

单调队列

- 给定一个长度为 n （n≤2*10^6）的序列，求出每一个长度为 m 的区间内的最小值

sample

Input

```
2    //m
7 8 1 4 3 2
```

Output

```
7 1 1 3 2
```

分析

- 假设现在我们已经处理了一个长度为 m 的区间 [L，R]，在右边加入一个元素，然后从左边删除一个数字就得到了下一个长度为 m 的区间 [L+1，R+1]

- 加入一个数字求最小值很简单，删除一个数字呢？

分析

- 我们只会从左边删除，考虑在区间内维护一个从左至右递增的队列，相当于是最小值的候选队，队首就是当前的最小值。如果最小值被删除，它后面一位就是能够顶替它的新的最小值

- 右边加入元素可以像之前单调栈那样维护队列的单调性

链表（linked list）

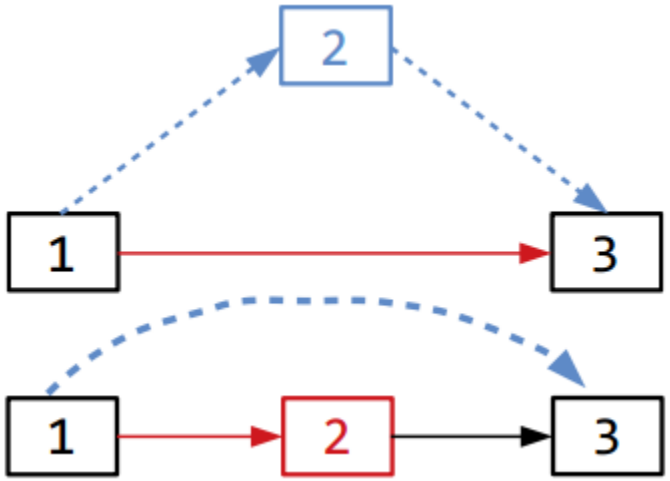
- 通常会开结构体，节点(node)包含 data 和 next 两个域

- 如果 next 为 NULL，表示是链表的最后一个元素

```
struct node
{
    int data;
    node *next;
};
```

链表（linked list）

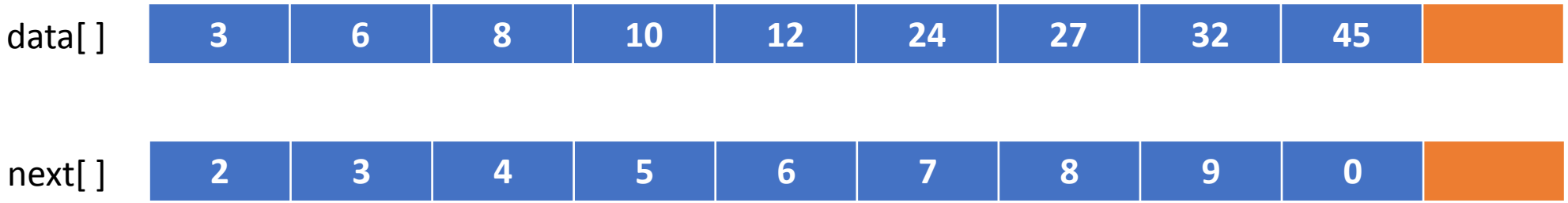
- 链表支持O(1)复杂度的插入和删除操作



- 数组的缺点刚好是链表的优点，链表的缺点刚好是数组的优点

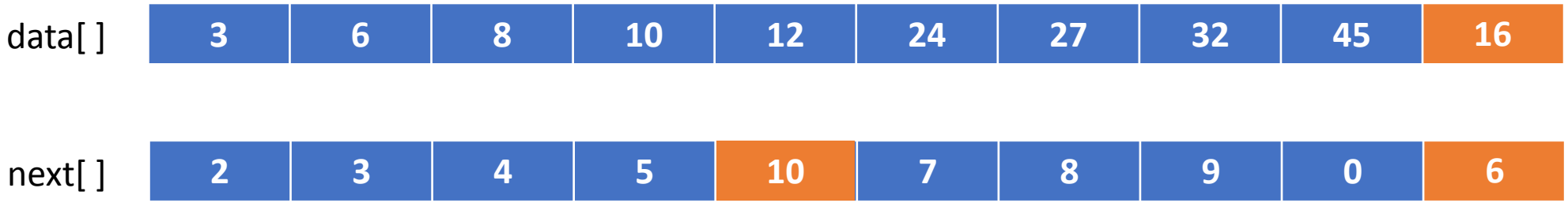
数组模拟实现链表

- 一般没有这种做法，纯为理解链表而设



数组模拟实现链表

- 一般没有这种做法，纯为理解链表而设

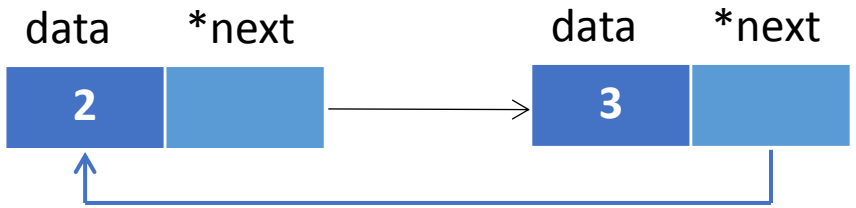


双向链表

- 每个节点(node)添加一个前驱指针 pre，使得可以在链表中双向移动

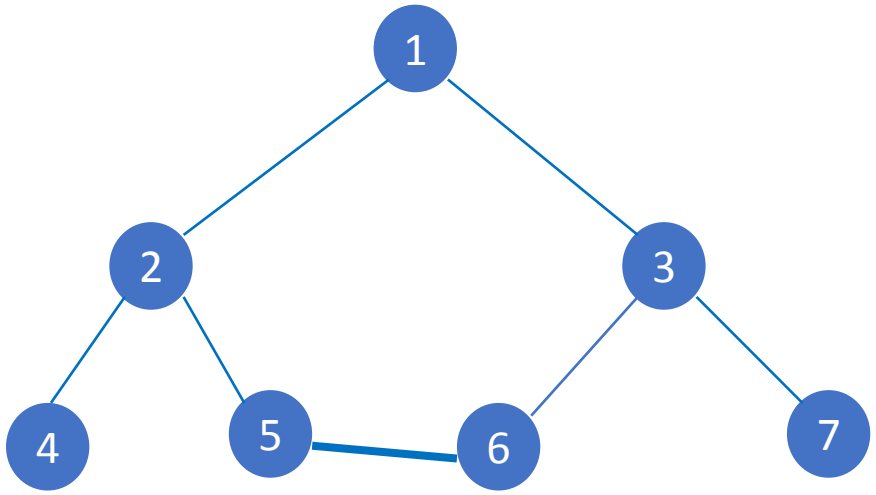
循环链表

- 链尾节点的 next 不指向 NULL，而是指向链首节点，使得可以从链表中任意一个元素开始遍历整个链表



树

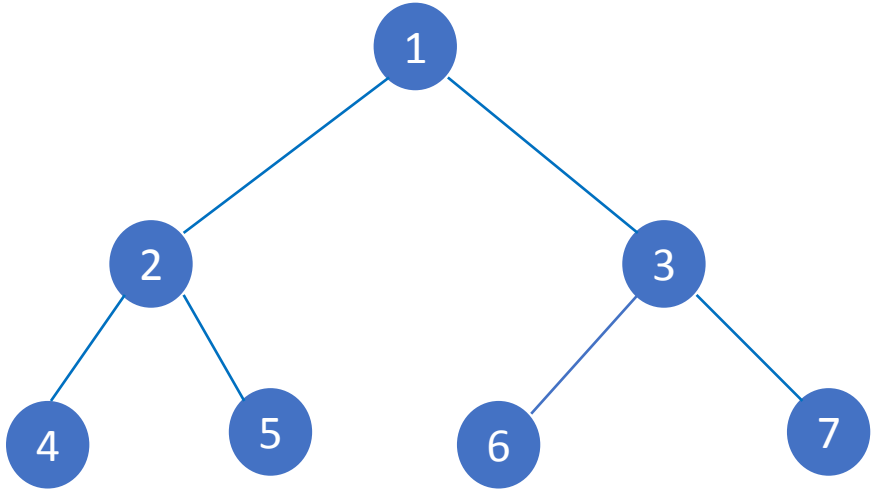
- 无向无环连通图
- n个节点，n-1条边
- 任意两个节点仅有一条路径连通



- 根节点
- 叶节点
- 父亲节点、儿子节点、兄弟节点、祖先节点

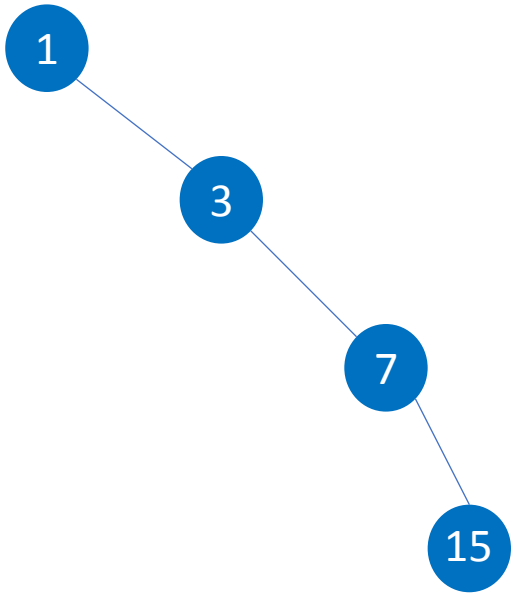
二叉树

- 左子树、右子树
- 满二叉树、完全二叉树



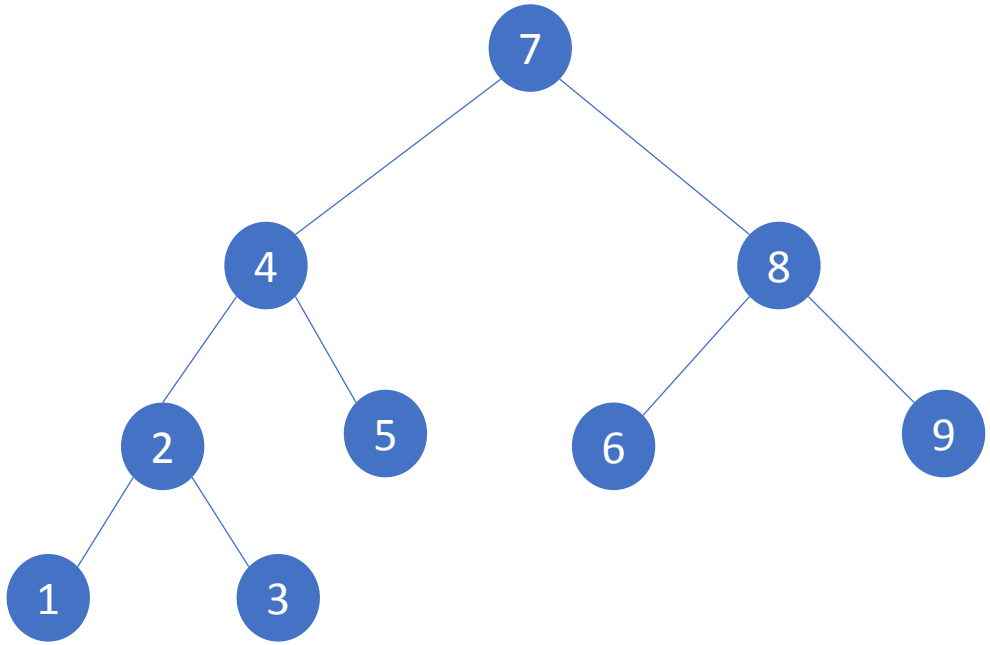
二叉树的表示

- 如果是完全二叉树，那么可以用k表示父节点，2k表示左儿子，2k+1表示右儿子，反之亦然
- 如果二叉树完全退化了呢？
- 父亲表示，左右儿子表示等等



二叉排序树/查找树

- 左子树若不空则不大于根节点，右子树若不空则不小于根节点
- 左右子树均符合递归定义

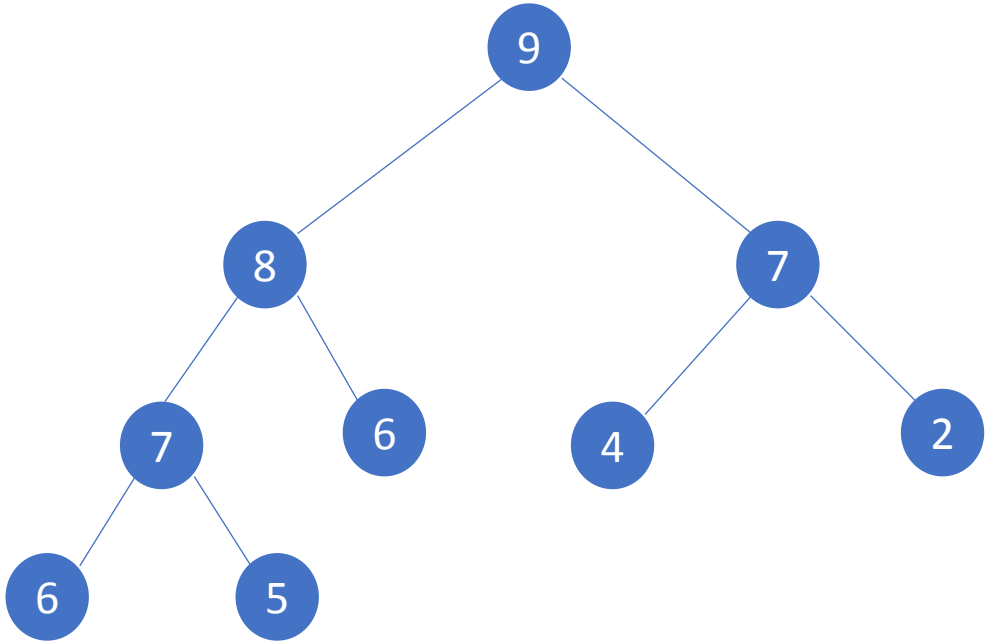


提问

- 以 {45，53，12，37，24，100，3，61，90，78} 为例，手工构造二叉排序树

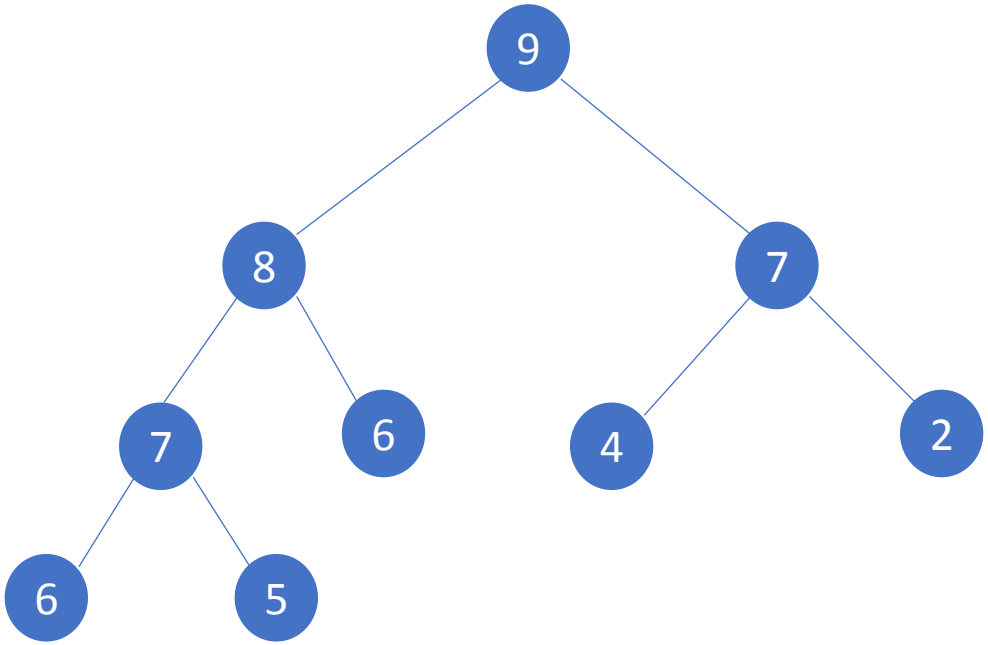
堆

- 满足特殊性质的完全二叉树
- 若左/右子树不空，则左/右子树不大于根节点（大顶堆）
- 左/右子树符合递归定义



堆

- 大顶堆/小顶堆



- 插入
- 删除
- 更新

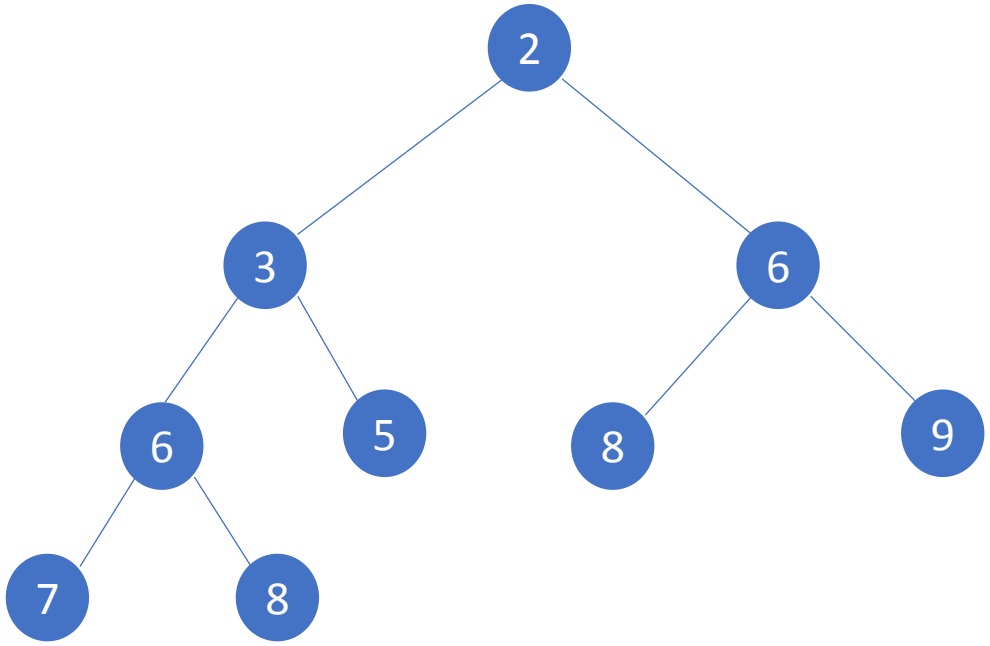
- 堆的性质适宜于查询并维护区间最值

heap

```
make_heap(a, a + n);  //将a数组0~n-1号元素建堆，默认大顶堆

push_heap(a, a + n + 1);  //a数组n号元素插入堆

pop_heap(a, a + n);  //删除堆顶，其实是放到堆尾

sort_heap(a, a + n);  //堆排序,不过排序之后就不是堆了
```

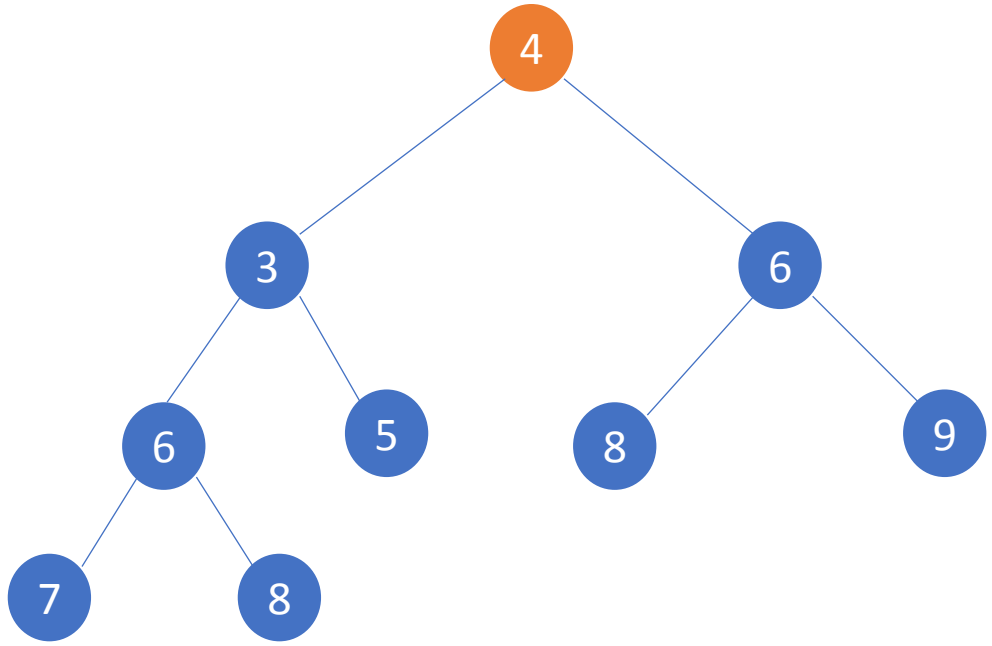
区间最值查询（RMQ）

- 2 3 6 6 5 8 9 7 8
- 询问最小值



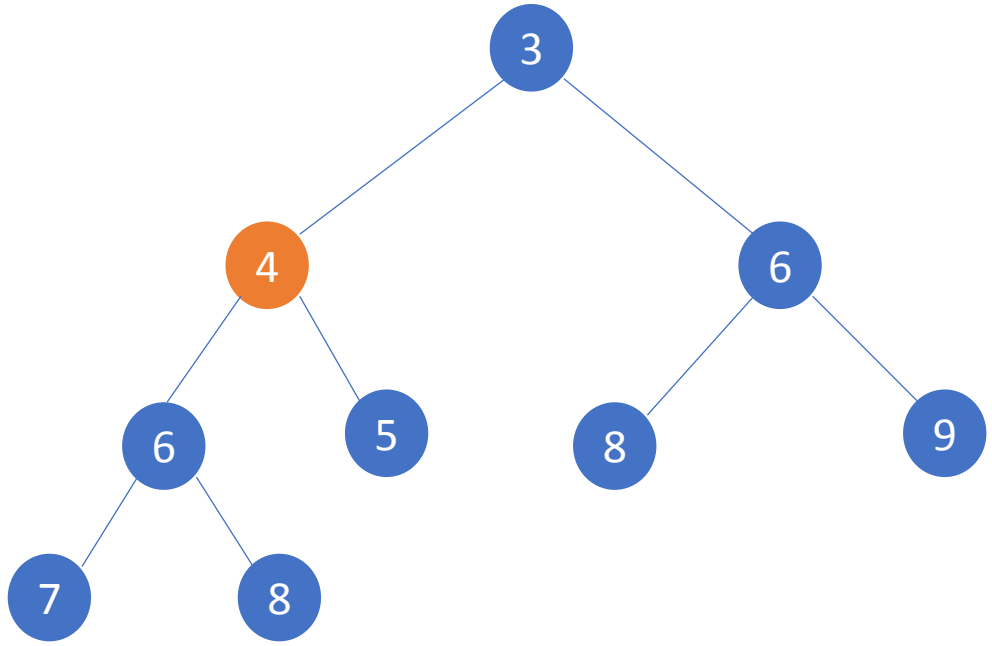
区间最值查询（RMQ）

- 2 3 6 6 5 8 9 7 8
- 将最小值删除，然后增加一个值4，再询问最小值



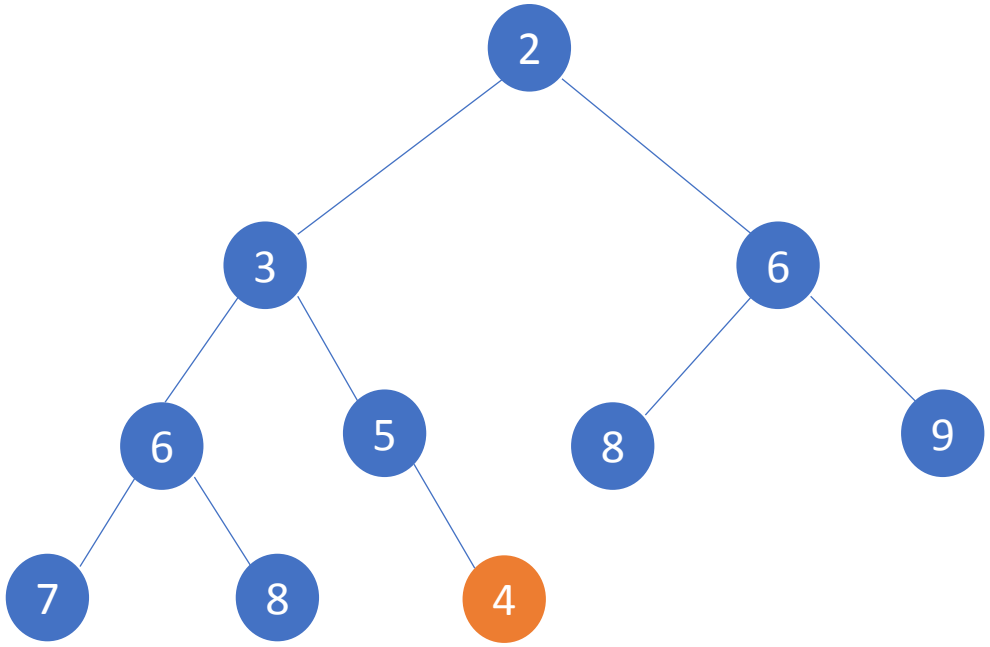
区间最值查询（RMQ）

- 2 3 6 6 5 8 9 7 8
- 将最小值删除，然后增加一个值4，再询问最小值



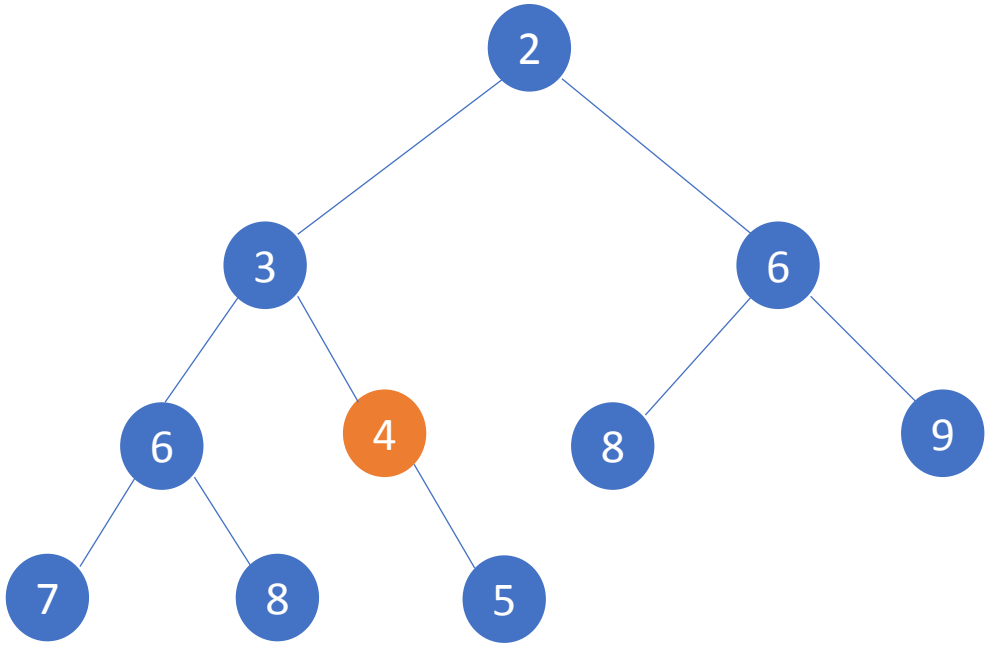
区间最值查询（RMQ）

- 2 3 6 6 5 8 9 7 8
- 不删除，直接增加一个值4，再询问最小值



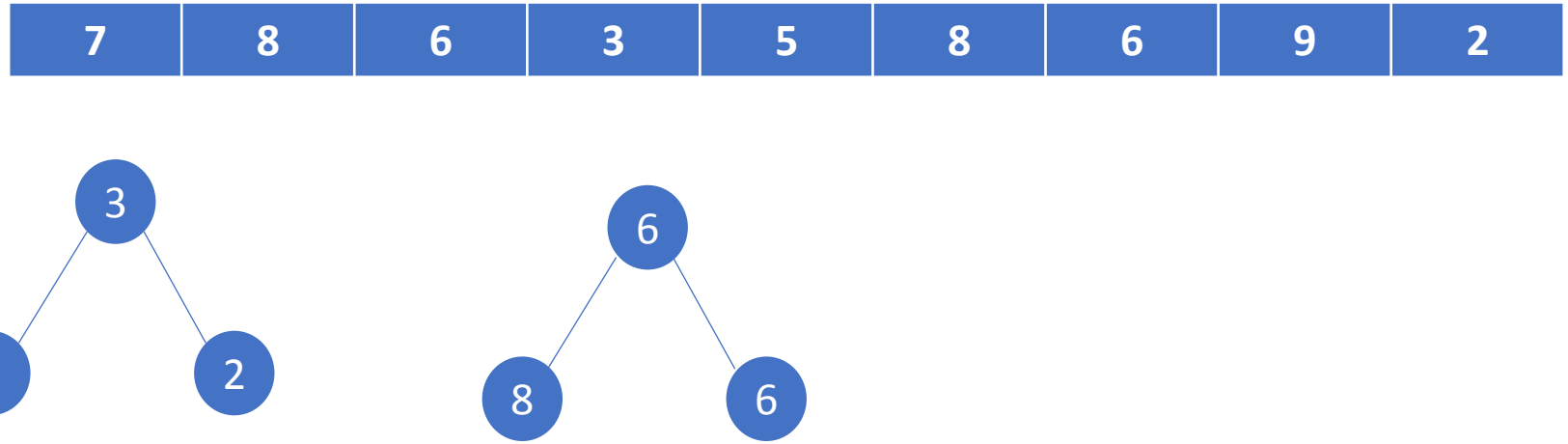
区间最值查询（RMQ）

- 2 3 6 6 5 8 9 7 8
- 不删除，直接增加一个值4，再询问最小值



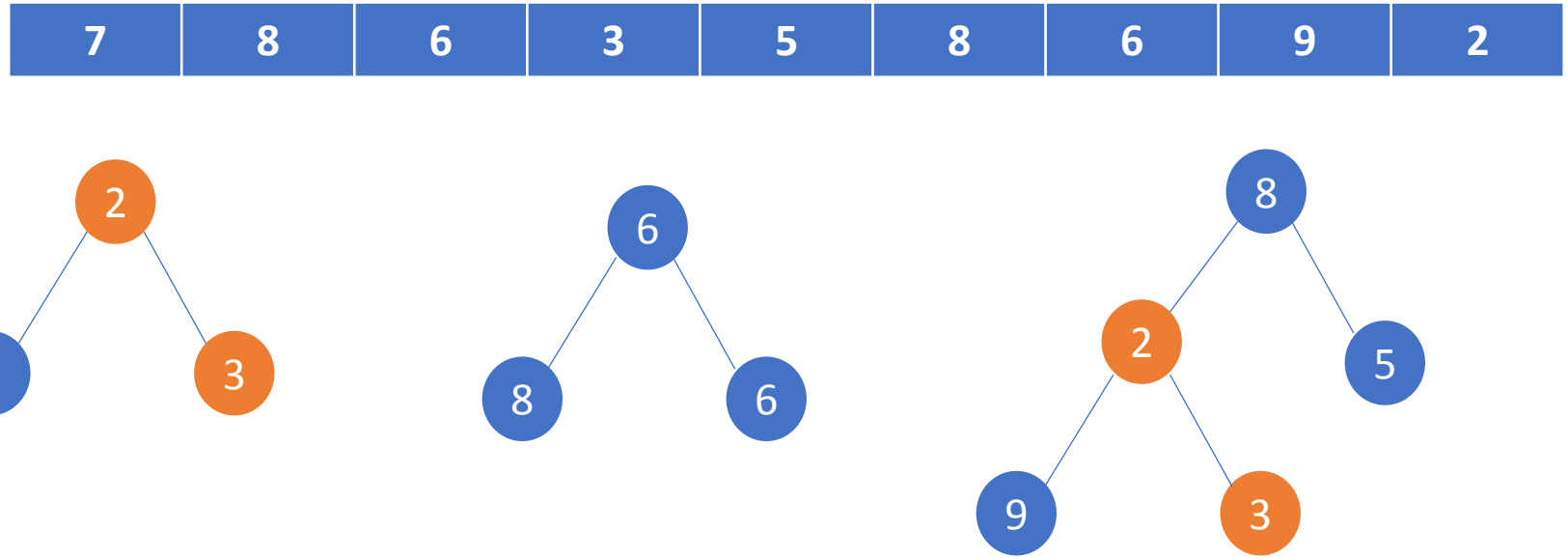
建堆

- 从第一个元素开始依次做插入操作直到全部加入，复杂度O(n)
- 还有更快的方法吗？



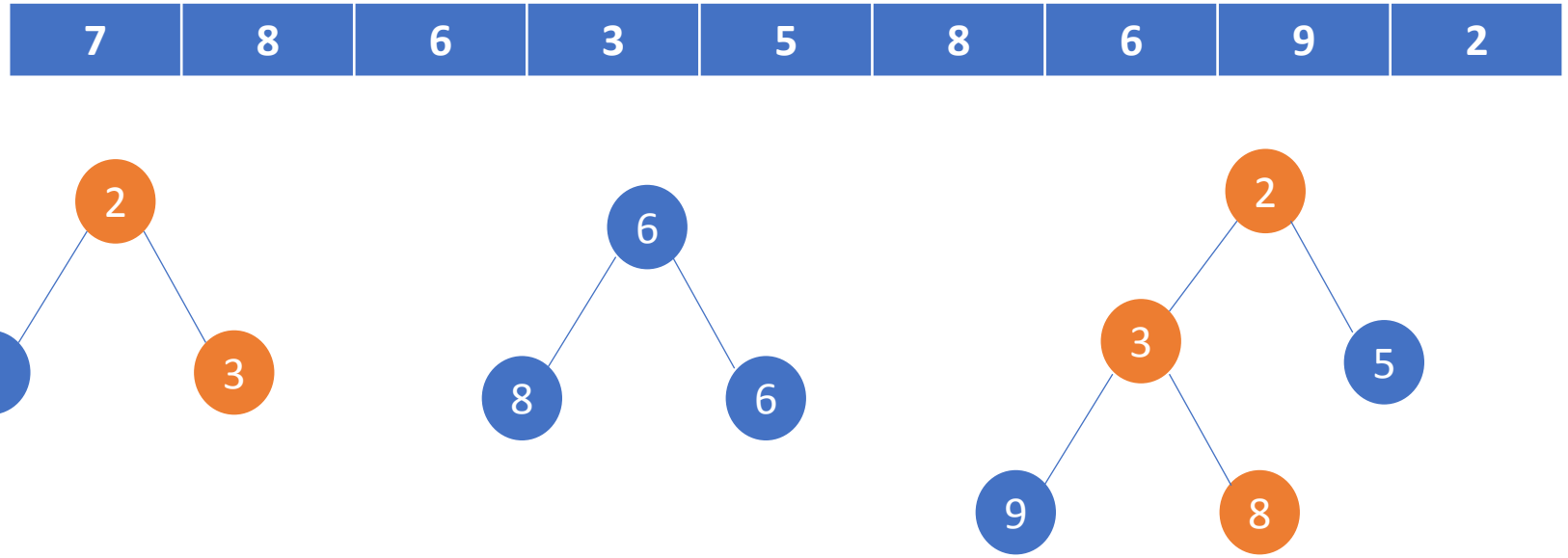
建堆

- 从第一个元素开始依次做插入操作直到全部加入，复杂度O(n)
- 还有更快的方法吗？

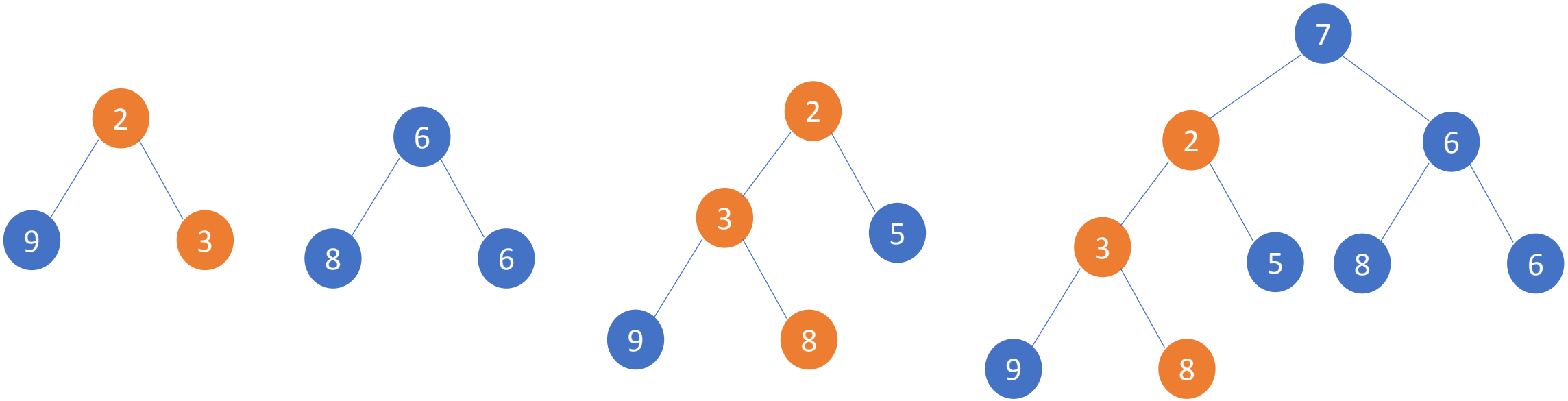


建堆

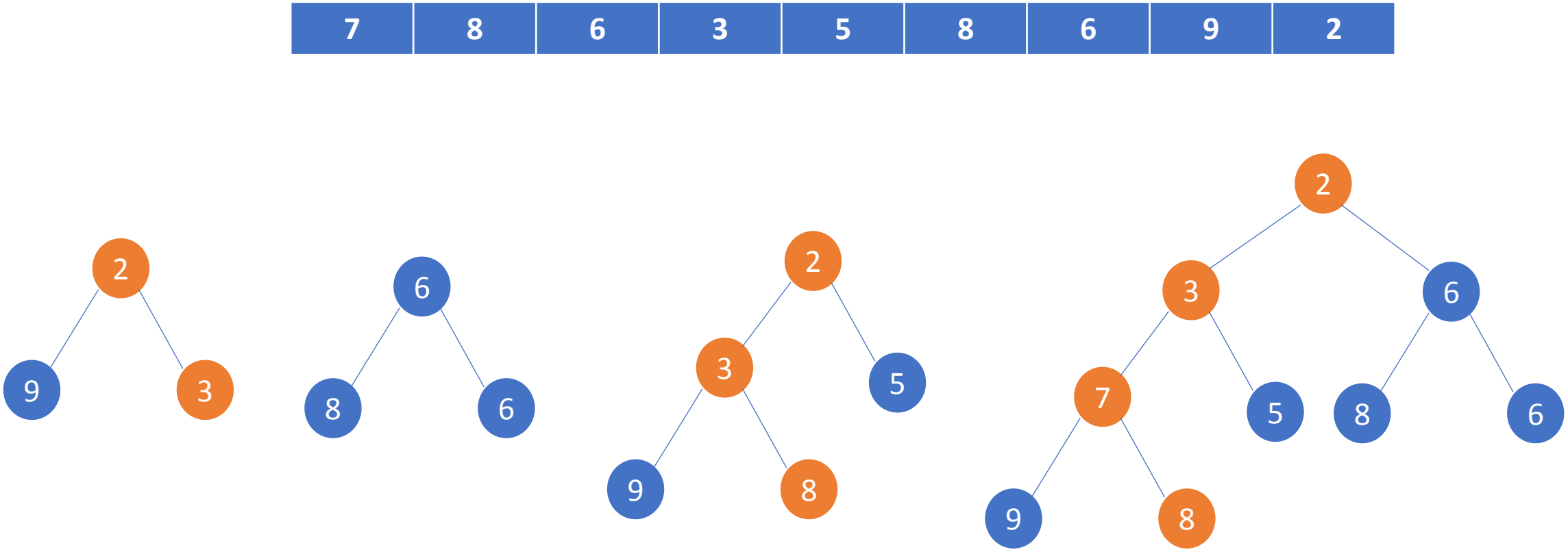
- 从第一个元素开始依次做插入操作直到全部加入，复杂度O(n)
- 还有更快的方法吗？



建堆



建堆



建堆

- make_heap也是采用的这种建堆方式

```
for (int i = n / 2; i >= 1; -- i)
    down(i);
```

提问

- 询问区间第 k 大

– 先排序，找到第 k 大，复杂度O(nlogn*q)　　// q为询问次数，下同

提问

- 询问区间第 k 大

– 扫描 k 次，每次把当前的最大值删除，复杂度O(nk*q)

提问

- 询问区间第 k 大

– 利用快排的思想，找到前面有 k-1 个数时的 key 即为答案（并不要求前 k-1 个数有序），复杂度 O(nlogk*q)

提问

- 询问区间第 k 大

  - 利用堆排的思想，建大小为 k 的小顶堆，后续元素逐个加入并与堆顶比较大小，维持堆大小为 k，最后堆顶即为答案，建堆复杂度 O(k)，维护堆复杂度 O((n-k)logk)，综合复杂度 O(nlogk*q)

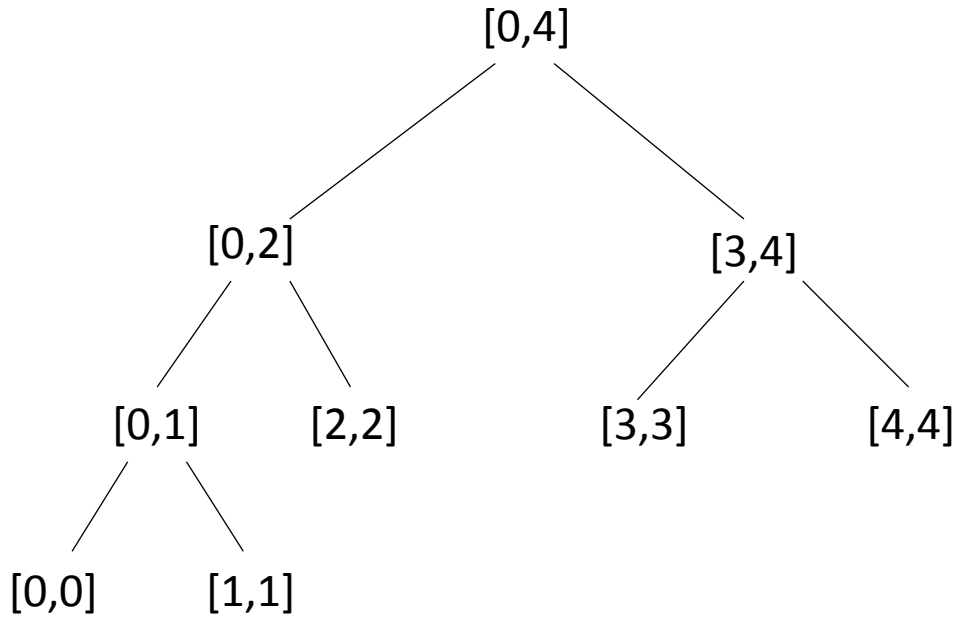
线段树

- 从树的角度来管理一段较大的连续线性区间，使得区间维护的复杂度从O(n)降到O(logn)
- 例如线段树可以在10的数量级管理1000数量级的线性区间，在20的数量级管理1,000,000数量级的线性区间
- 因此线段树主要用于维护线性区间的插入、修改、查询等操作

线段树

- 建树：根节点表示区间[0,n-1]，然后把区间分成两半，分别由左右子树表示并递归下去
- 叶节点可以是单元区间，也可以是单点（点树）
- 线段树是平衡二叉树（AVL），线段树把任意区间[a,b]最多分成log(b-a)份

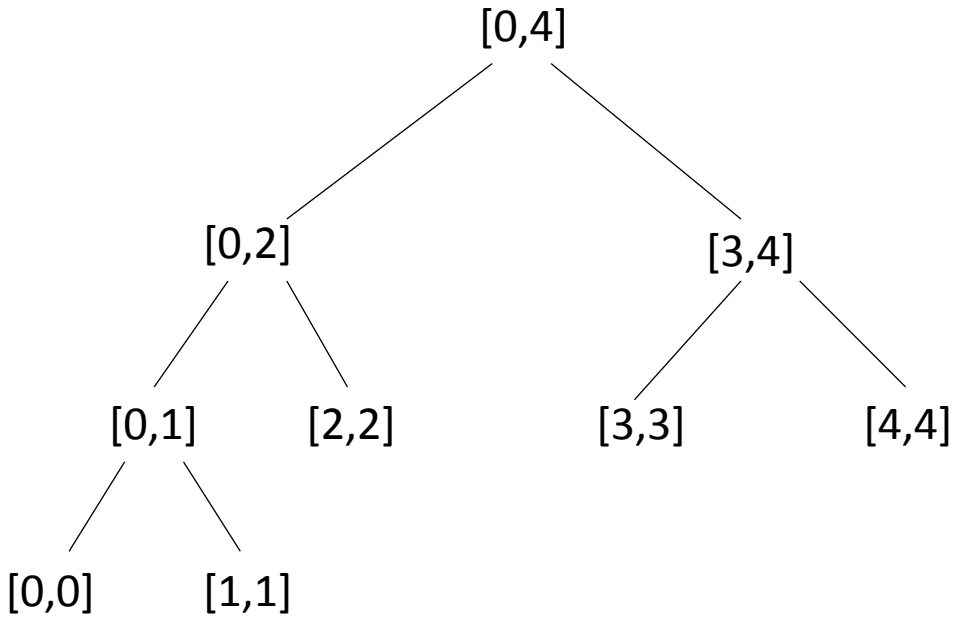
线段树



- 用一个struct存储一个节点，包括：
  - 所辖区间左、右端点 left、right
  - 左右儿子 lc、rc
  - 该节点存储的数值

- 这样一个一维struct数组就可以表示一棵线段树

线段树



- 对每个节点x，有：

fa(x) = $\lfloor x/2 \rfloor$

lc(x) = 2*x

rc(x) = 2*x+1

- 数组需要开多大？

满树的节点数为底层节点的两倍-1

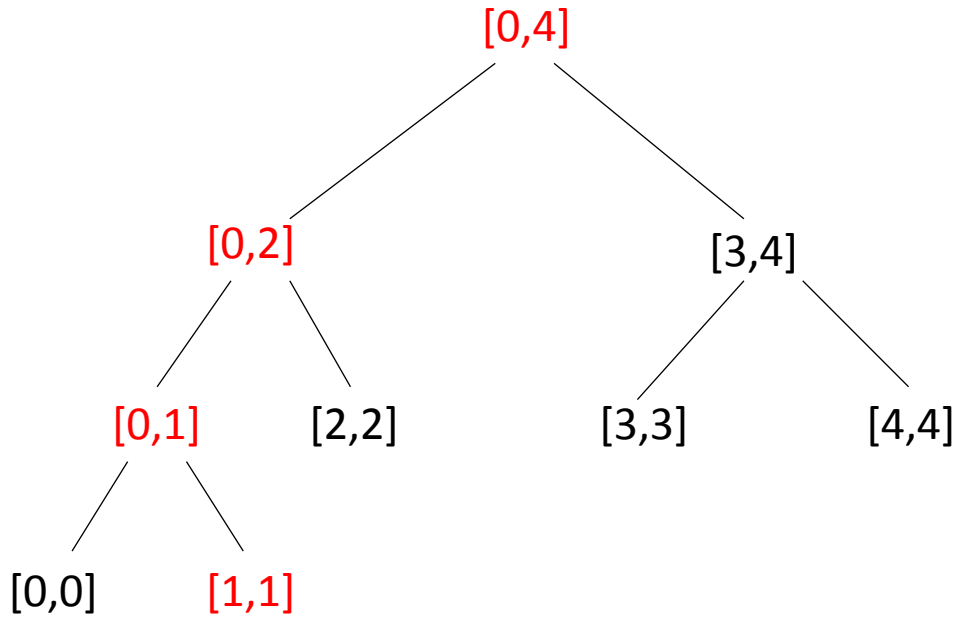
底层节点最多为2*n

故数组大小一般开4*n

线段树的点修改

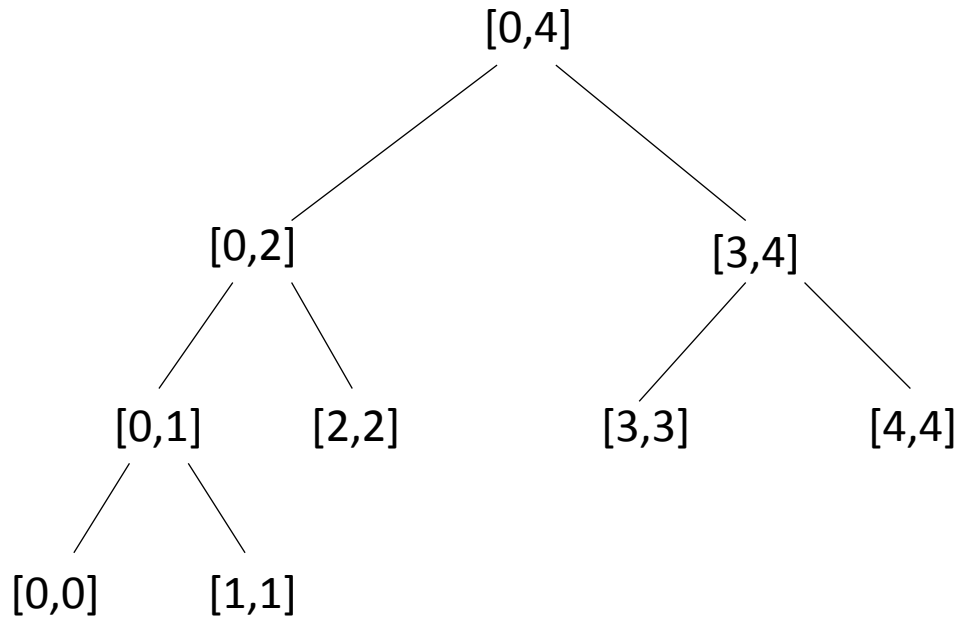
- 利用二分找到叶节点，可以点修改

- 修改一个点，需要修改从根节点到该节点的一条链



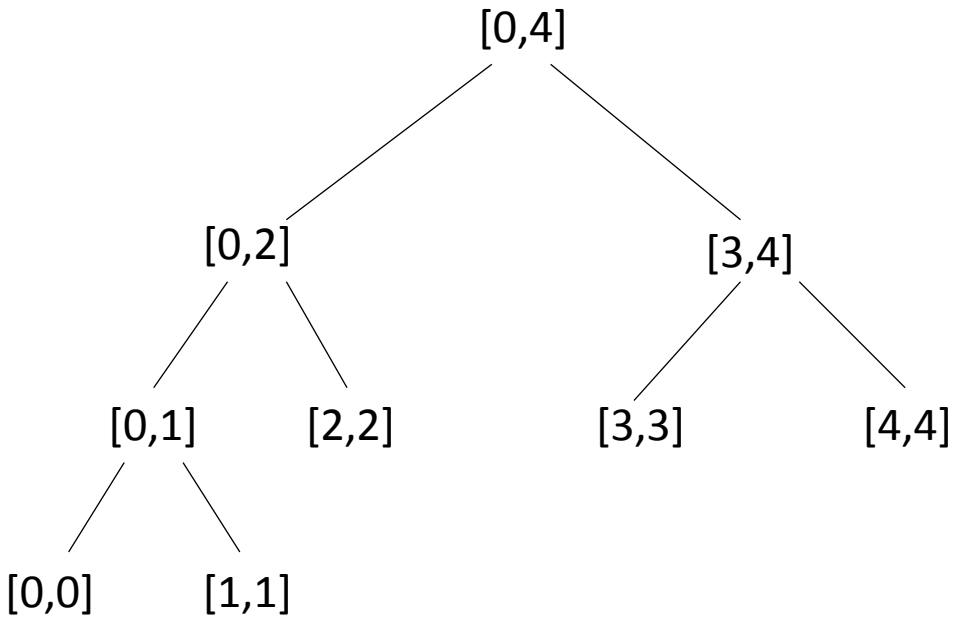
线段树的区间查询

- RMQ[0,2]
- RMQ[0,3]



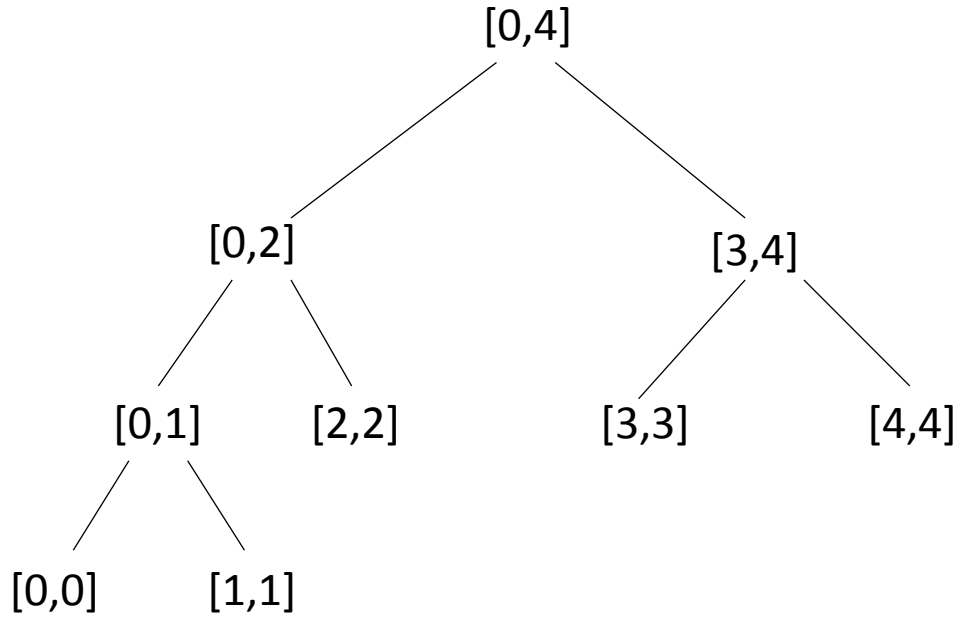
线段树的区间修改

- 区间修改是指修改某个区间内所有叶节点的值
- 这样的修改必然会影响其对应的祖先节点、父节点
- 所以为每个节点新增一个标记，记录该节点是否进行了修改



线段树的区间修改

- 例如：区间修改[0,2]+2
- 找到非叶节点[0,2]，修改其值，并标记2，修改完毕
- 若要查询区间[0,1]的最小值，查询到[0,2]时发现其还要向下查询且其标记为2，就要把标记向下传递
- 修改[0,1]、[2,2]的值+2，并标记2，同时消除[0,2]的标记



再举例

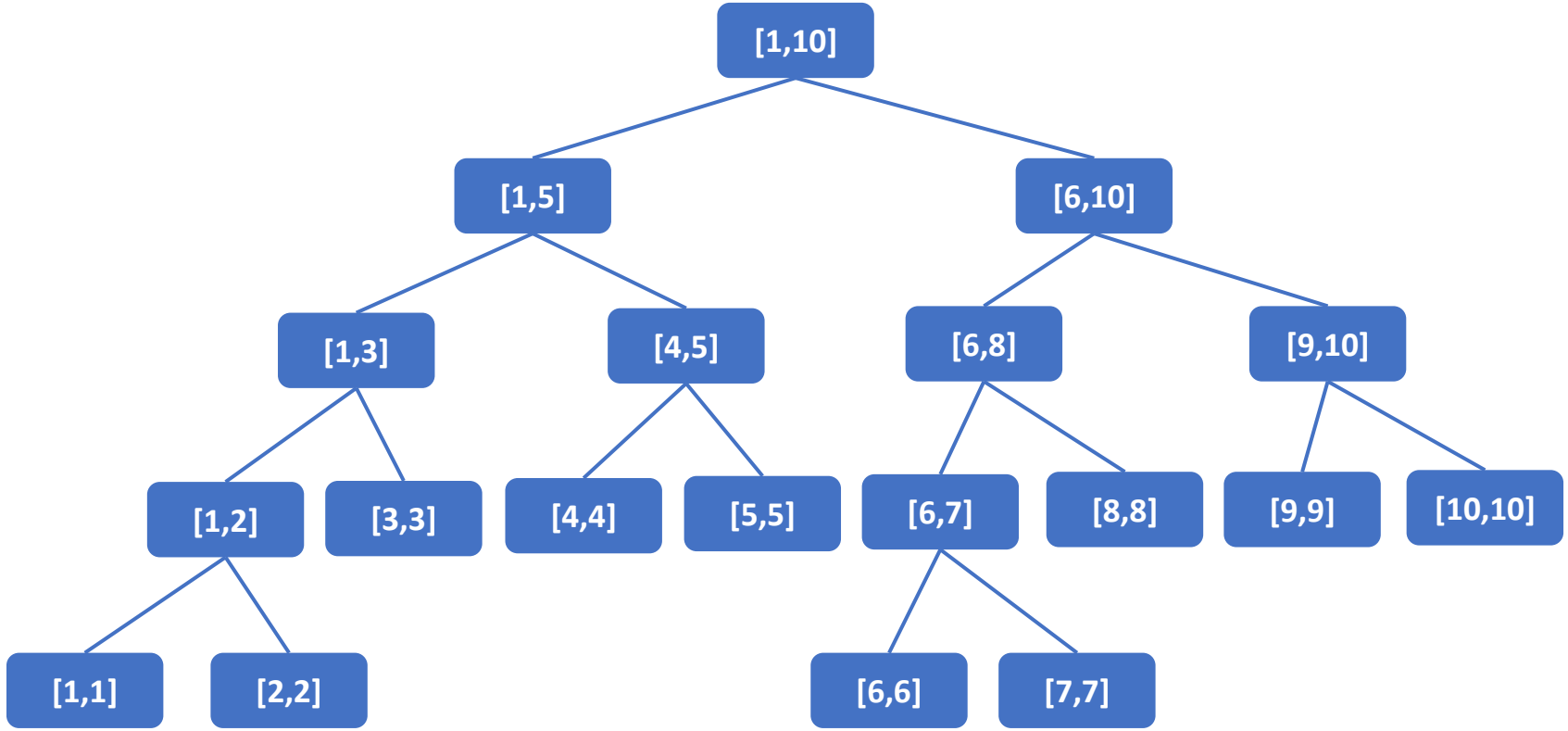
- 假如需要在表示区间[1,10]的线段树中做两个操作：先修改[1,5]的值，再询问[1,3]的值。
- 在修改[1,5]时，递归到区间[1,5]时就直接退出了
- 然而在查找[1,3]时，并不知道原来[1,5]修改过

延迟标记

- 对于一个修改操作，我们在线段树中找到对应的节点时，我们可以将其打上标记，而不是直接往下继续修改
- 在询问或修改一个节点 i 时，若 i 被标记过，则修改 i 并将标记传给 i 的左右儿子，同时清空 i 的标记
- 对于某些修改，比如说对一段区间同时加上某个数，只需在询问时自底向上更新值即可

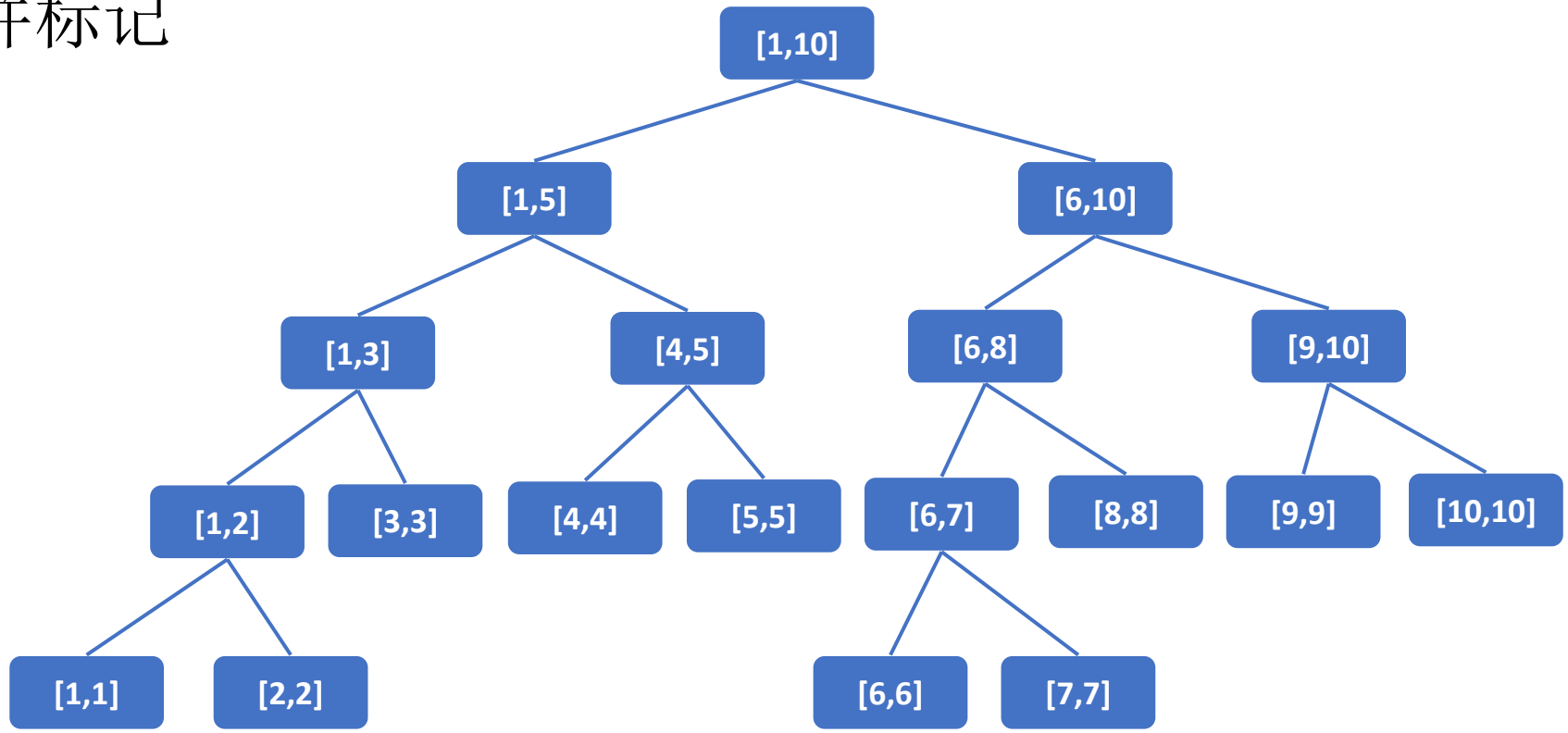
区间修改演示

- 比如有一棵[1,10]的线段树



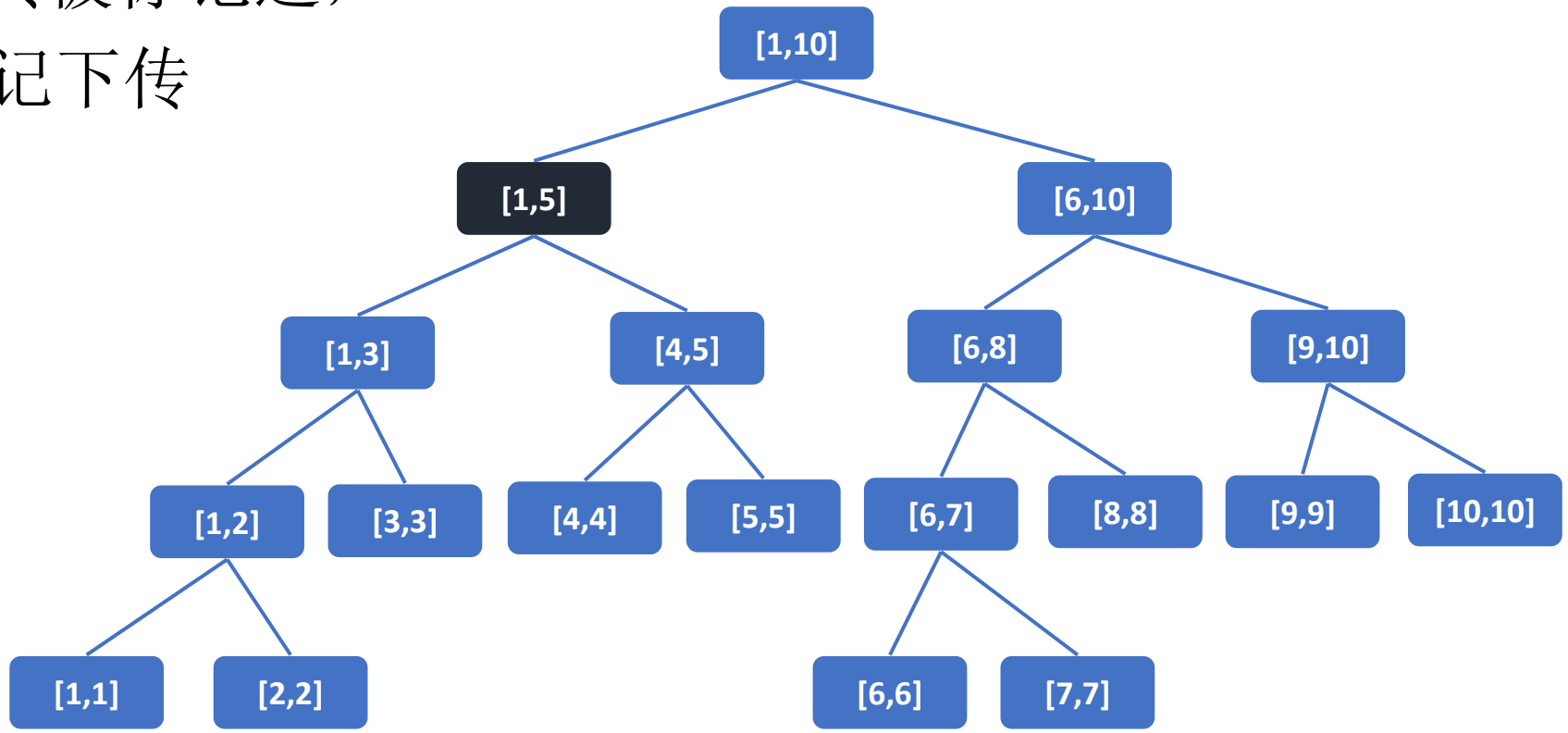
区间修改演示

- 现在要修改区间[3,5]
- 找到[1,5]并标记



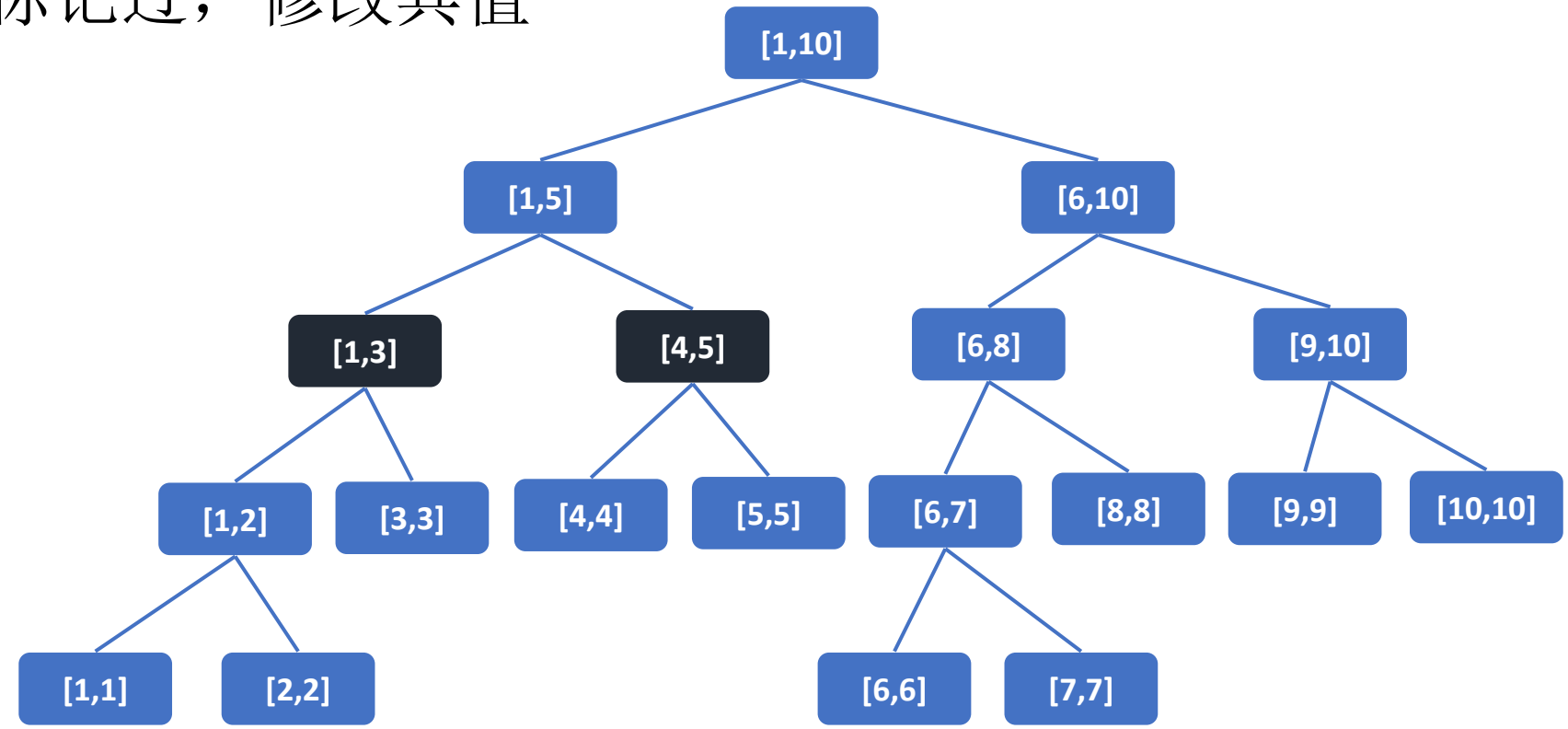
区间修改演示

- 然后要询问3，从根节点开始向下递归询问
- 找到[1,5]，发现其被标记过，
- 于是修改并将标记下传



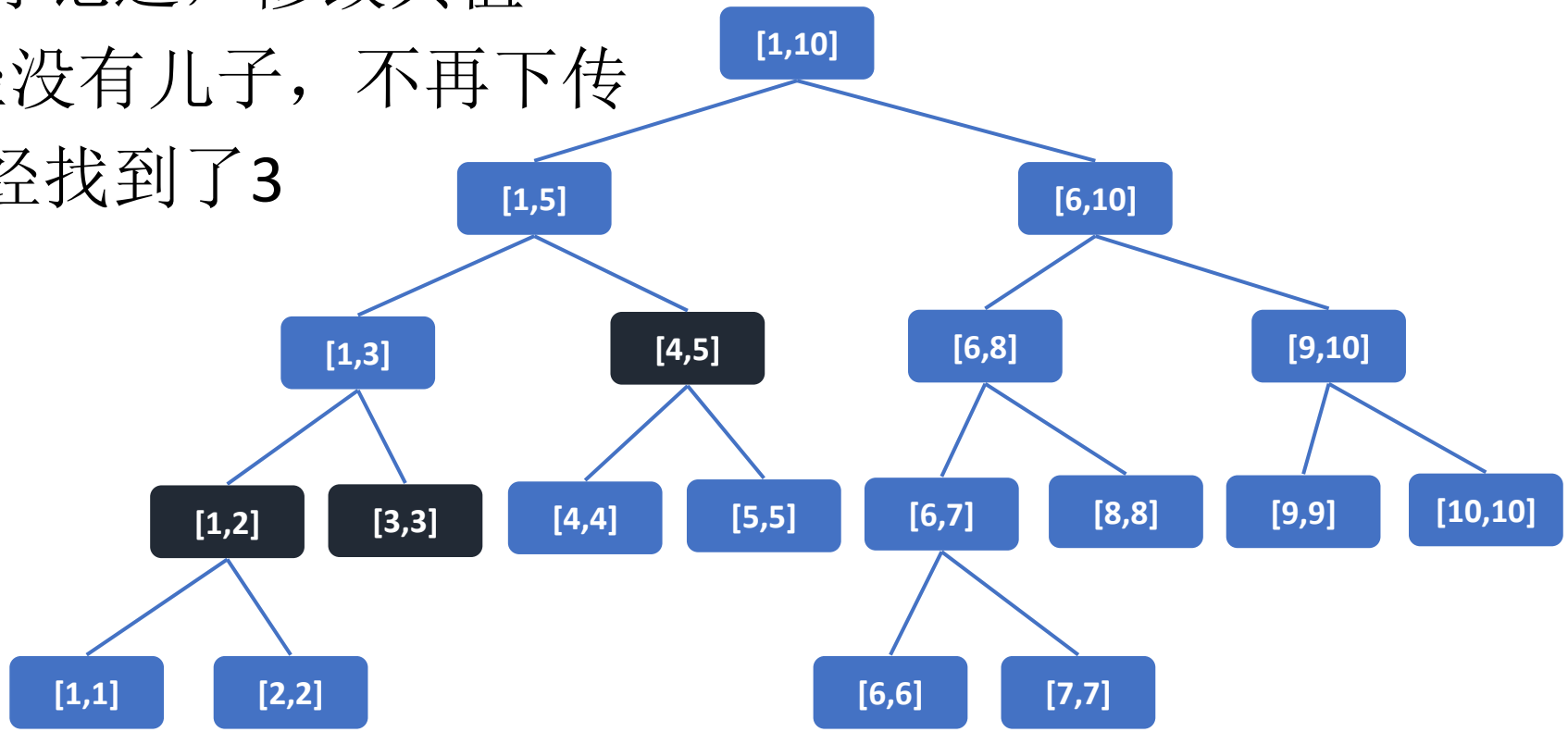
区间修改演示

- 继续向下递归，找到[1,3]
- 发现其被标记过，修改其值并向下传



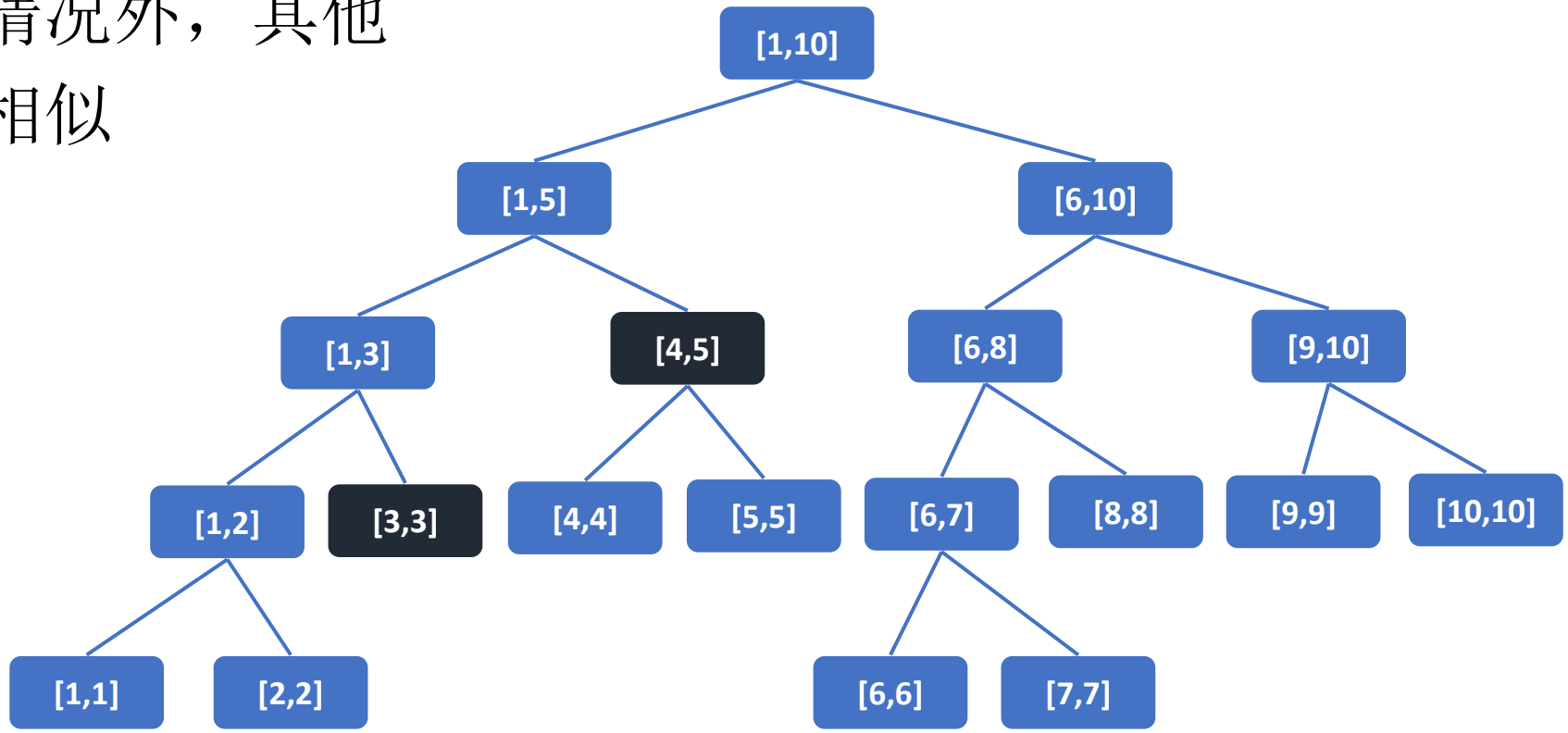
区间修改演示

- 继续向下递归，找到3
- 发现其被标记过，修改其值
- 由于3已经没有儿子，不再下传
- 此时，已经找到了3



区间修改演示

- 具体实现除了要再记录节点的标记情况外，其他和区间查询相似



ZKW版线段树

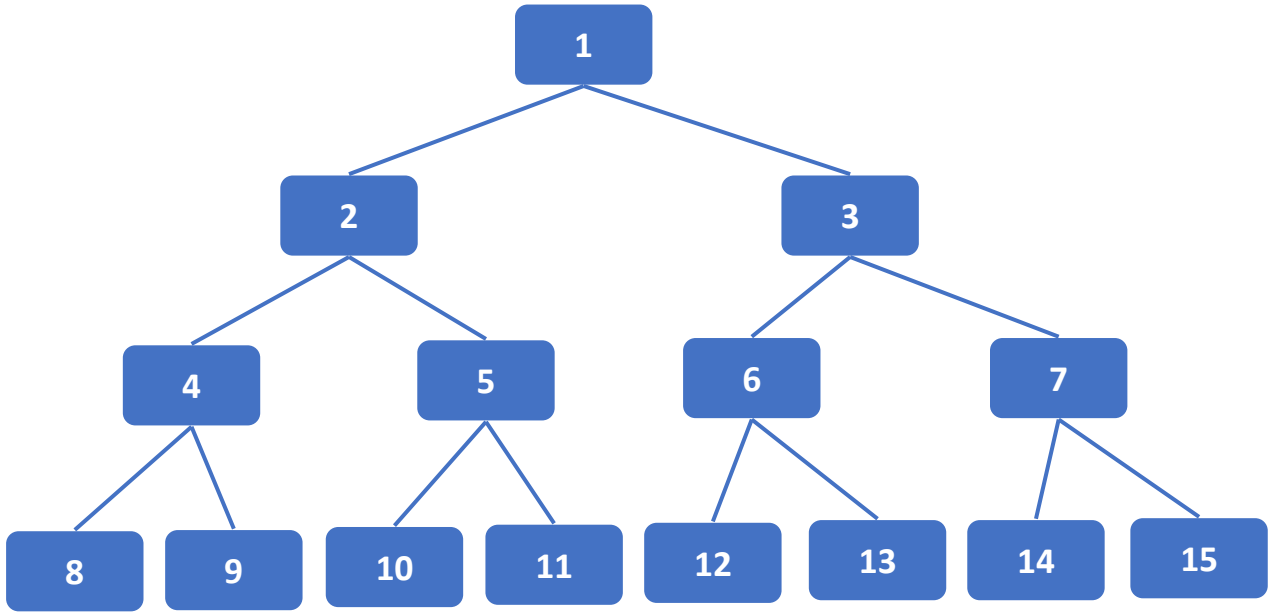
- ZKW版线段树的本质就是强行将一棵线段树填满，此时我们所关心的叶节点在线段树数组中的位置就能直接计算出来，从而实现自底向上实现， 达到优化常数的目的

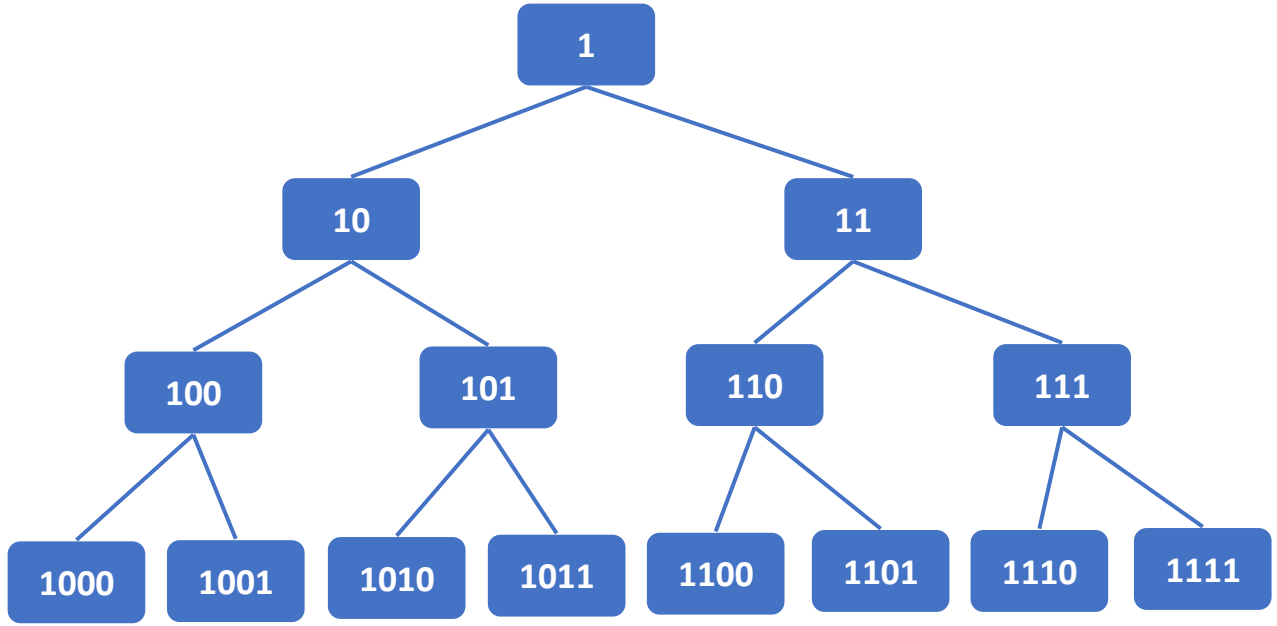
ZKW版线段树

- [1,8]的线段树刚好是一棵满二叉树
- [1,7]、[1,10]呢？
- 都建成满二叉树

ZKW版线段树

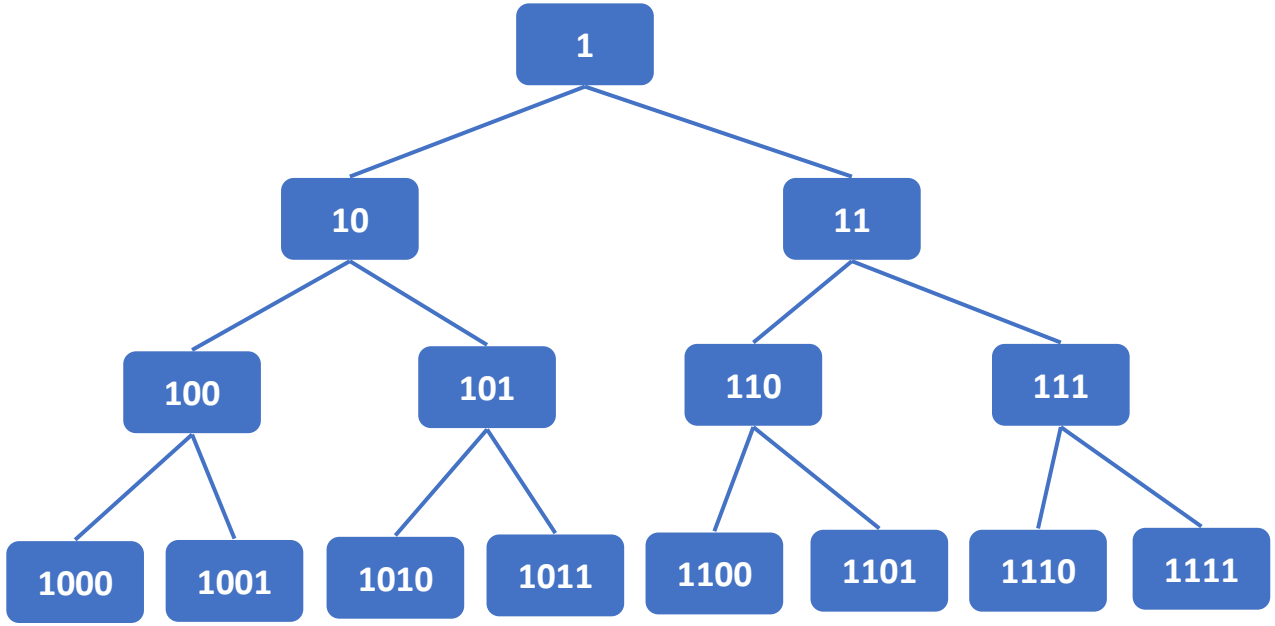
- 建成满二叉树后，以二进制形式存储点树形态线段树





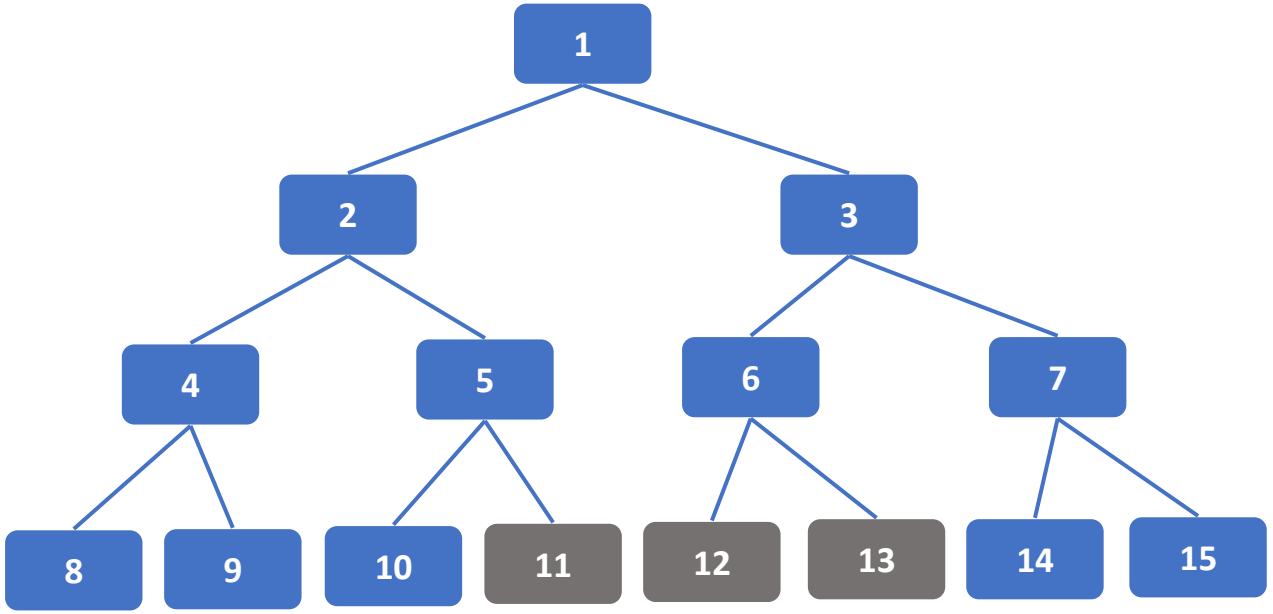
ZKW版线段树

- 这样做的好处是每个节点存放的是以这个节点为前缀的区间和



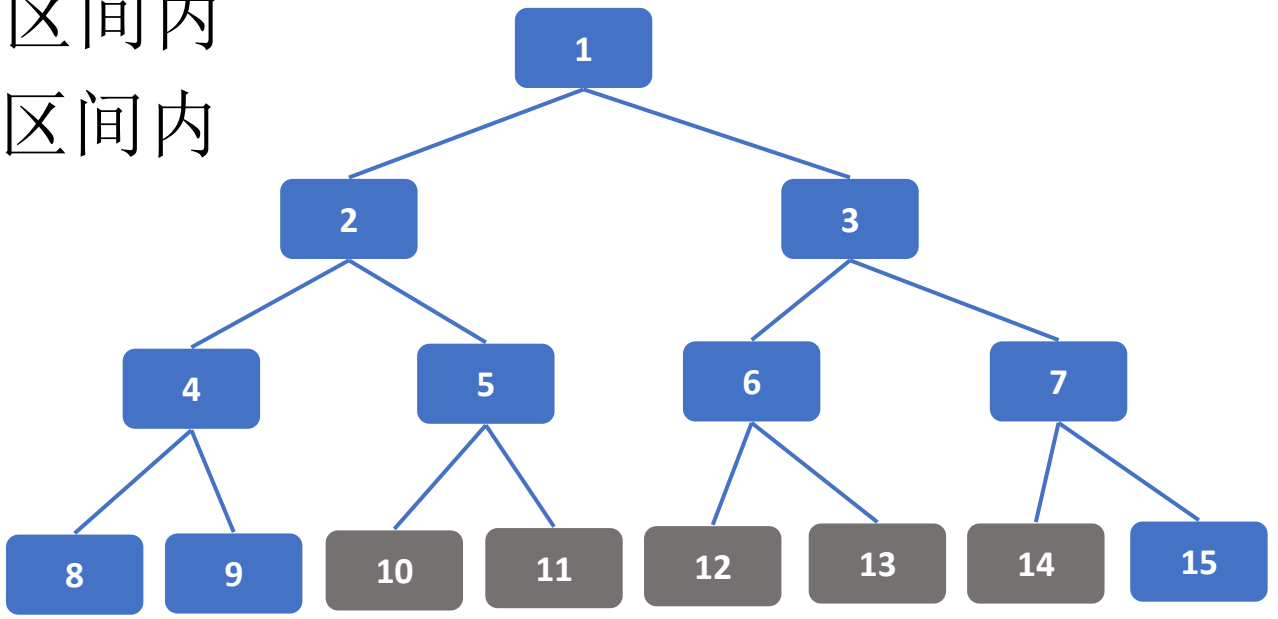
ZKW版线段树的区间查询

- 查询闭区间[3,5]
- 注意建树的时候第一个和最后一个节点为空



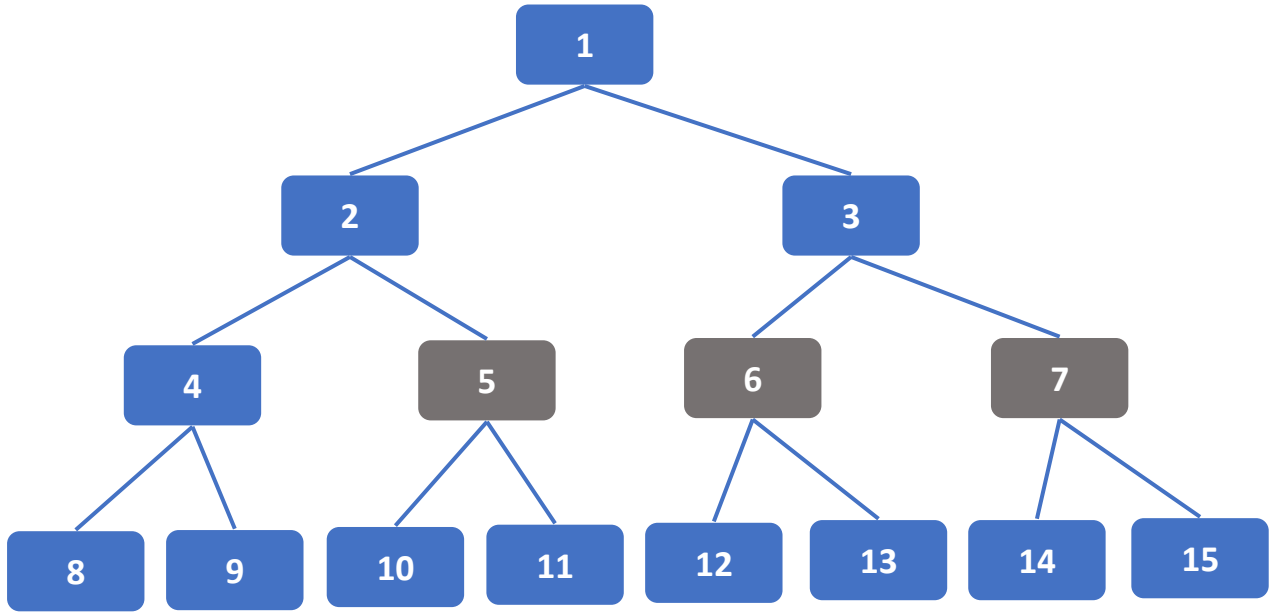
ZKW版线段树的区间查询

- 改成开区间的形式
- 先找到区间[s,t]的叶节点
- 若s是左子树，则其右兄弟必在区间内
- 若t是右子树，则其左兄弟必在区间内



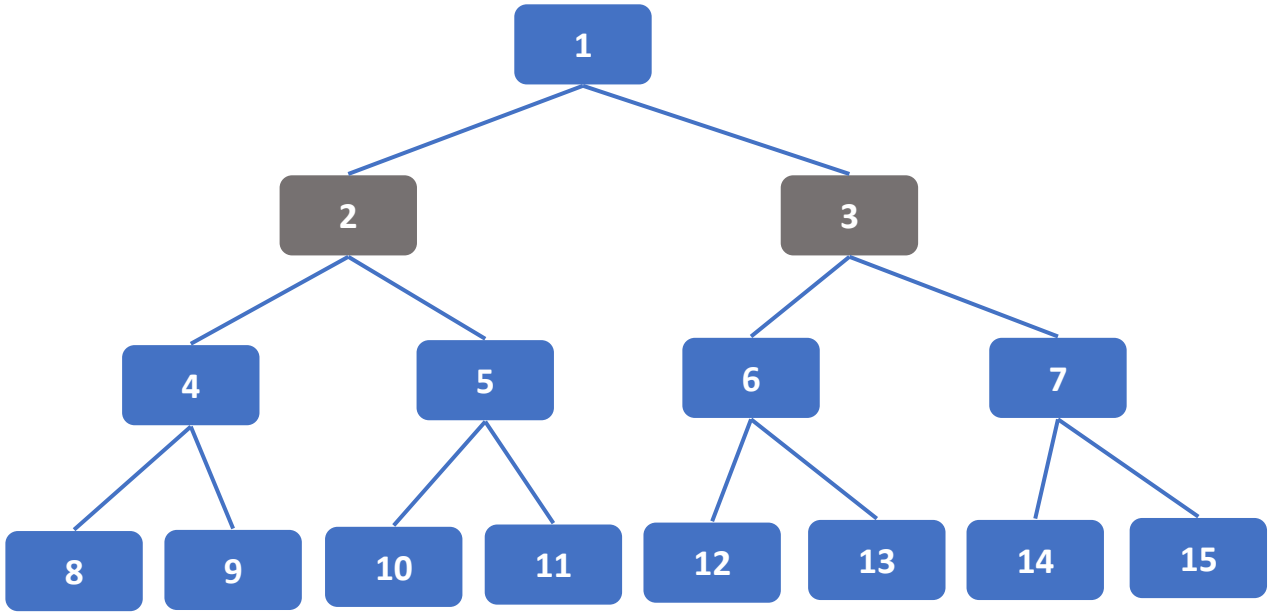
ZKW版线段树

- 上移一层
- 为什么要改成开区间？
- 为什么建树的时候要保持第一个和最后一个节点空？



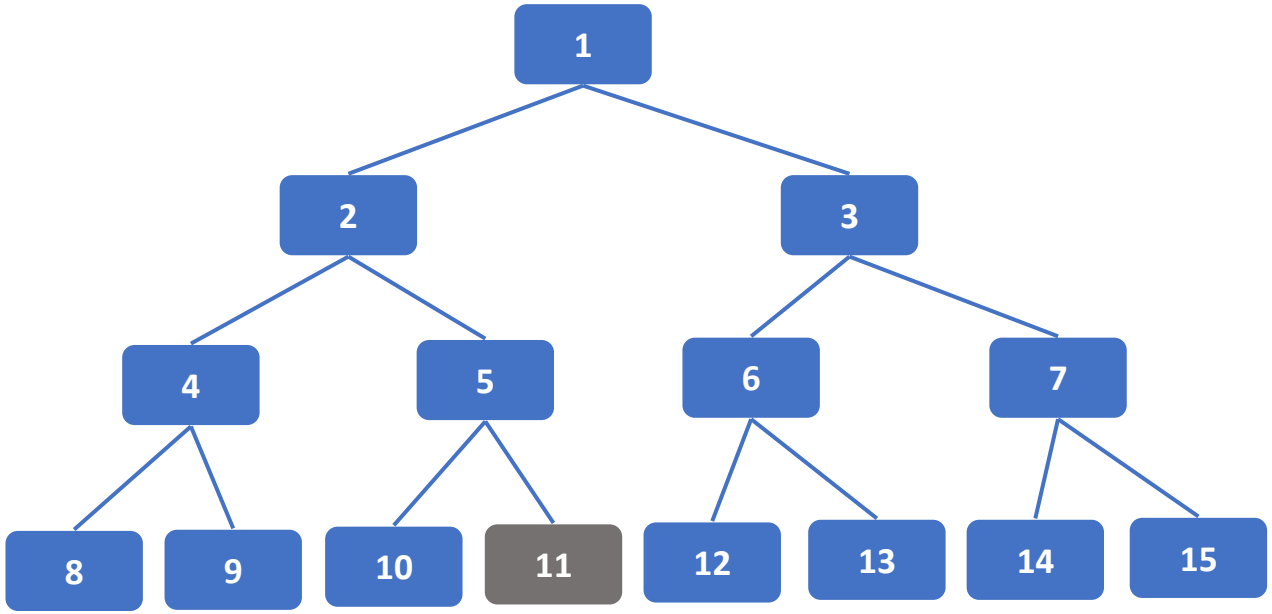
ZKW版线段树的区间查询

- 继续上移
- 当左右节点编号相邻，查询结束



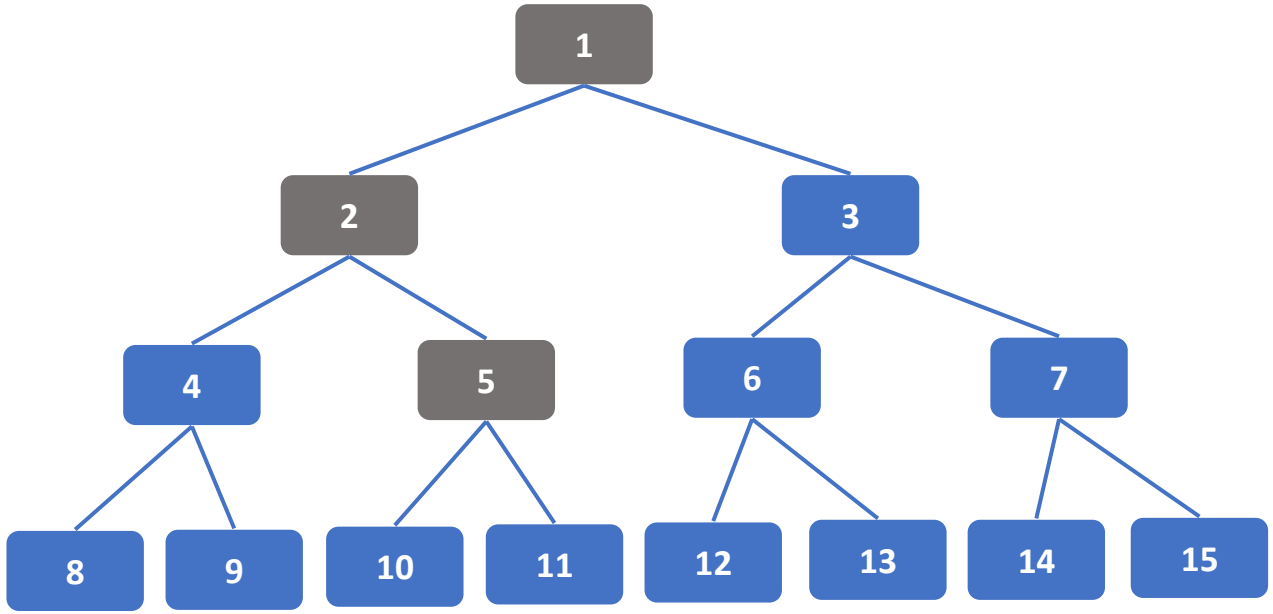
ZKW版线段树的点修改

- 修改点3的值



ZKW版线段树的点修改

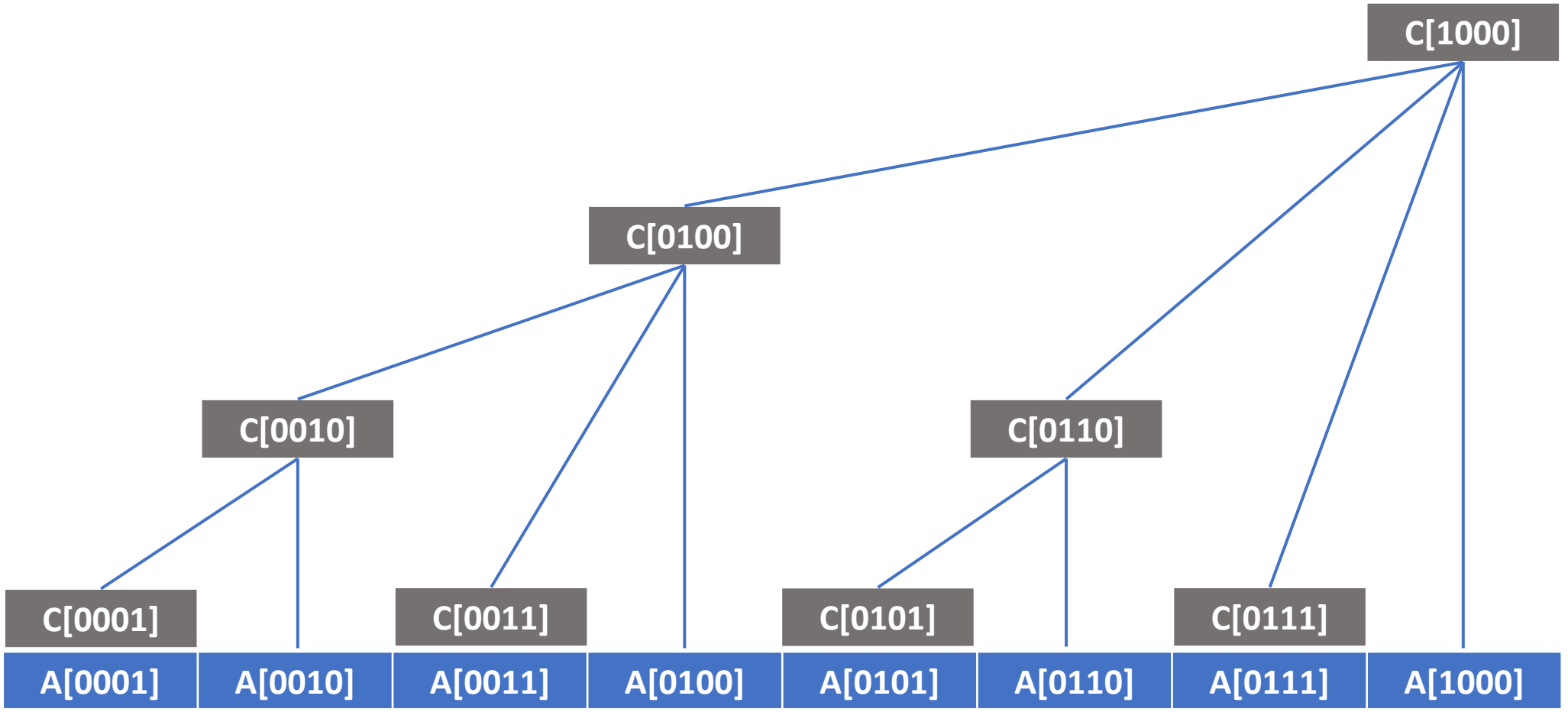
- 影响到点3的有（图中编号）5号、2号、1号三个点



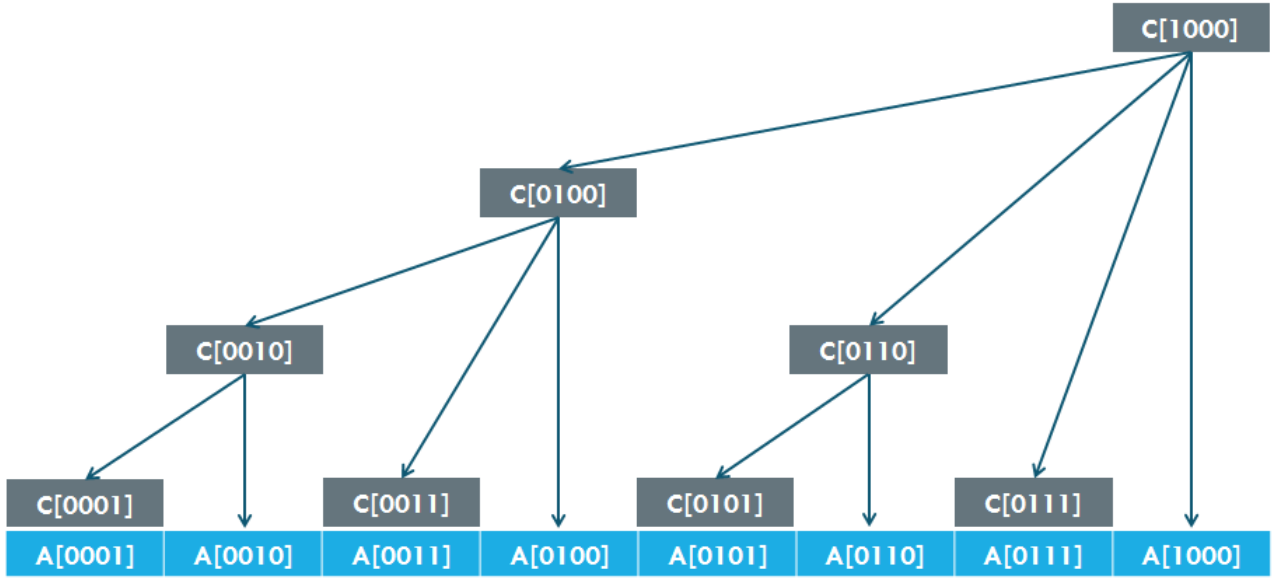
树状数组

- 树状数组是前缀和思想在树上的延伸
- 前缀和在维护静态数组的区间和问题上是一个有力工具。但如果是动态数组，我们很容易发现，修改任意一个数组元素 a[i]，都不得不修改前缀和数组中的 s[i]、s[i+1]、……、s[n]，如果修改操作比较频繁，将直接导致前缀和失去应有的作用
- 由此我们需要新的辅助数组来应对维护动态数组的区间和一类问题

换成二进制


C[1000]
C[0100]
C[0010]
C[0110]
C[0001]
C[0011]
C[0101]
C[0111]
A[0001]
A[0010]
A[0011]
A[0100]
A[0101]
A[0110]
A[0111]
A[1000]

再看有什么性质

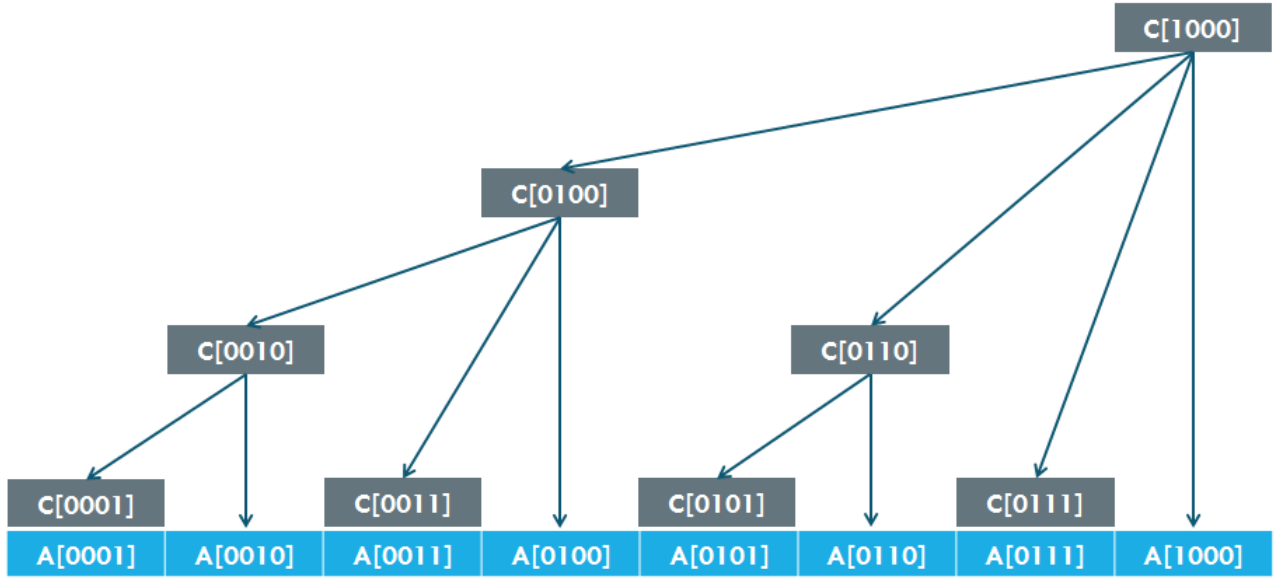


- C[0001]=A[0001]
- C[0010]=A[0001]+A[0010]
- C[0011]=A[0011]
- C[0100]=A[0001] + A[0010] + A[0011] + A[0100]
- C[0101]=A[0101]
- C[0110]=A[0101] + A[0110]
- C[1000]=A[0001]+A[0010]+A[0011]+A[0100]+A[0101]+A[0110]+A[0111]+A[1000]

再看有什么性质

- 设节点编号为X，K为X的二进制表示中末尾“0”的个数。那么：

1. C[X]=A[X-2^K+1]+……+A[X]
2. 节点X所管辖的区间共有2^K个元素



树状数组

- 树状数组的核心思想是通过维护某些段的和来减少一次修改和询问影响到的量

- 在这一过程中巧妙利用了二进制及位运算

树状数组有什么用

- 看这样一类典型问题：对一个长度为N的序列进行M次操作，操作有两种类型：

1. 修改X位置元素的值
2. 查询[1..Y]区间元素的和

- 以上两种操作分别是点修改和区间查询

朴素做法

- 点修改：直接修改A[X]，复杂度O(1)
- 区间查询：直接统计A[1]、A[2]、……、A[Y]的和，复杂度O(n)

前缀和做法

- 增加前缀和数组B[X]，令B[X]=A[1]+A[2]+……+A[X]
- 区间查询：直接统计B[Y]，复杂度O(1)
- 点修改：当A[X]增加值s，B[X]、B[X+1]、……都需要增加s，复杂度O(n)

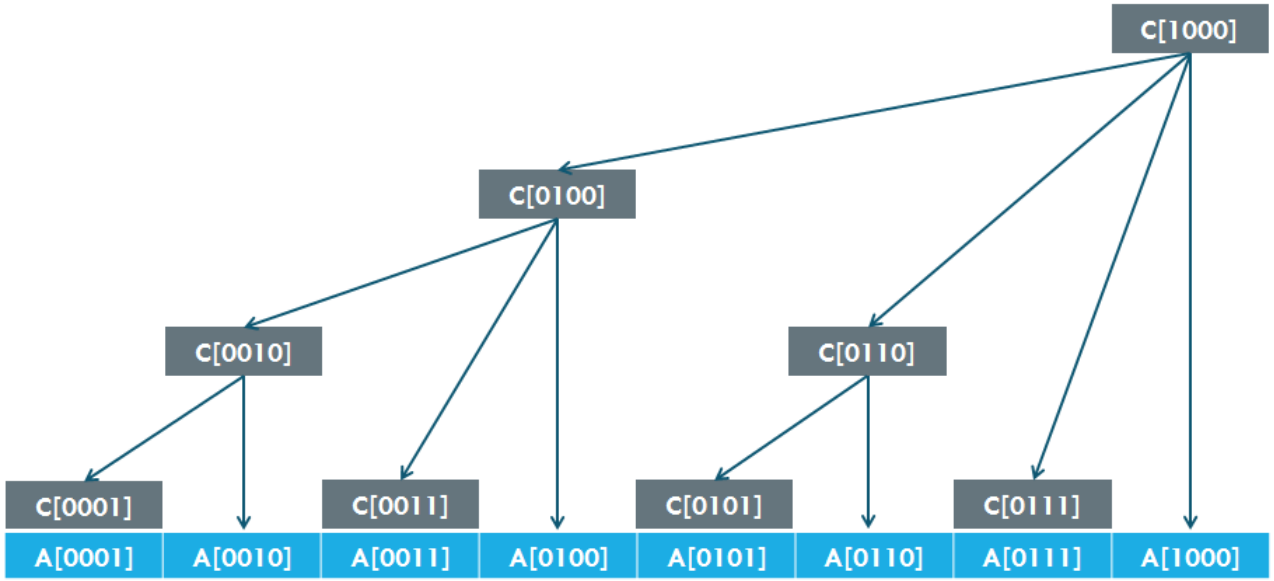
树状数组的点修改

- 观察下图：当修改A[X]时，影响到的点是：

C[X], C[X+=2^{k1}], C[X+= 2^{k2}],…

ki代表第i次X末尾0的数量

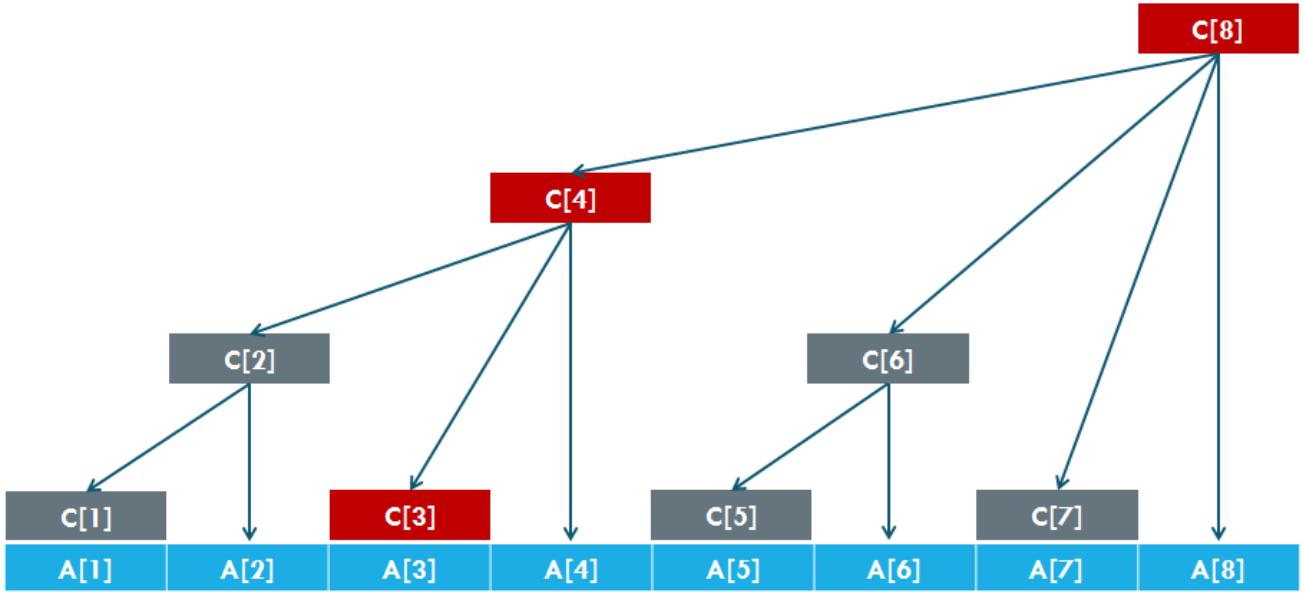
- 被影响到的C数组数量至多为logn个



树状数组的点修改

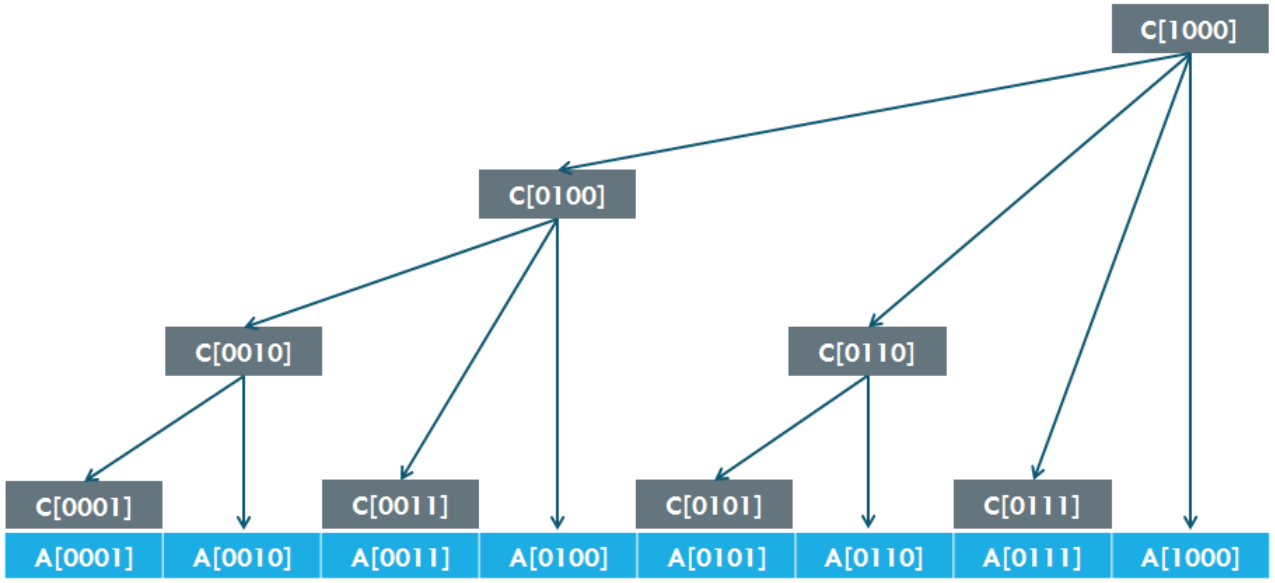
- 例如要修改A[3]：
- 影响到的点是C[3]、C[3+2^0]、C[4+2^2]

- 例如要修改A[6]：
- 影响到的点是C[6]、C[6+2^1]



树状数组的区间查询

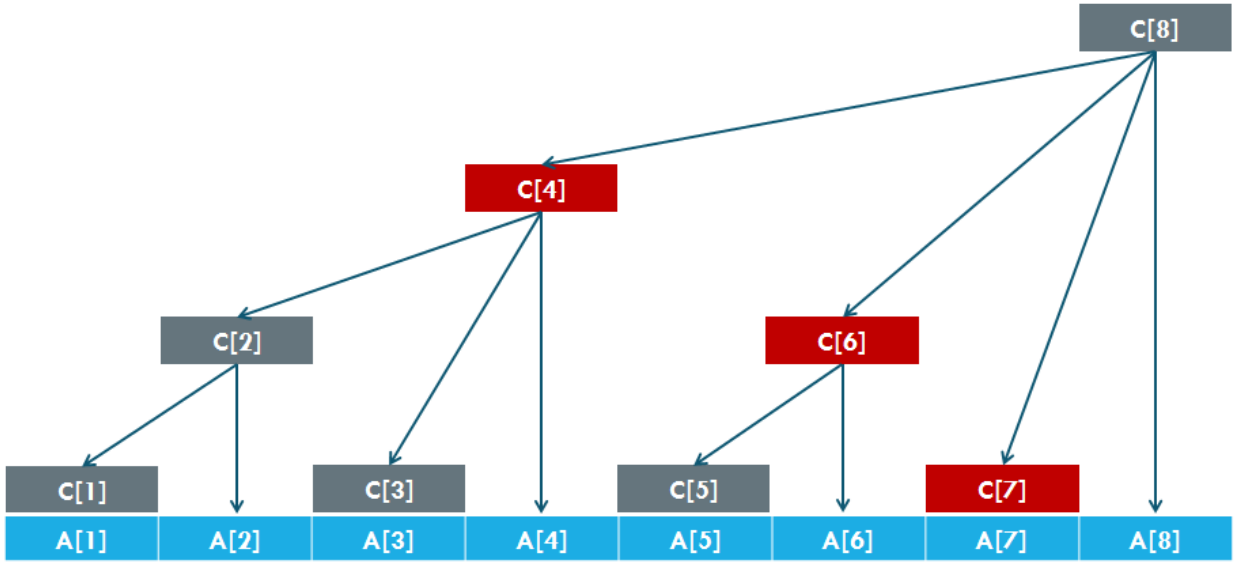
- 我们可以将A[1]~A[Y]划分成若干个C[i]的和。它们分别是：
- C[Y], C[Y-= 2^{k1}], C[Y-= 2^{k2}],…
- ki代表第i次X末尾0的数量
- 被影响到的C数组数量也至多为logn个



树状数组的区间查询

- 例如要查询区间A[1]…A[7]的和：
- 划分的区间为C[7]+C[7-2^0]+C[6-2^1]

- 例如要查询区间A[1]…A[4]的和：
- 划分的区间为C[4]



怎样快速求X的2^k

lowbit = x & (-x)

- lowbit 即最低位，是整型数的二进制表示中最后一位1的位置
- 比如：00001101、01101000

- 那么lowbit为什么是 x & (-x) 呢？

怎样快速求X的2^k

- 因为计算机中的负数是用补码表示的。而补码是反码加1，如：
- x = 1，表示为 0001，-1 表示为 1110 + 1，即1111
  - 1 & (-1)：0001 & 1111 = 0001 = 2^0
- x = 4，表示为 0100，-4 表示为 1011 + 1，即1100
  - 4 & (-4)：0100 & 1100 = 0100 = 2^2
- x = 6，表示为 0110，-6 表示为 1001 + 1，即1010
  - 6 & (-6)：0110 & 1010 = 0010 = 2^1
- x = 8，表示为 01000，-8 表示为 10111 + 1，即11000
  - 8 & (-8)：01000 & 11000 = 01000 = 2^3

怎样快速求X的2^k

lowbit = x & (-x)

- 实际上得到的是2^{lowbit}

一点优化

- 树状数组C实际上相当于原数组A的一个辅助数组
- 如果在线操作的话，还可以再节省一个数组的空间开销

树状数组小结

- 树状数组是由前缀和的思想在树上的应用，可以用来求[1，R]的区间和，能求[L，R]的区间和吗？
- 树状数组可以用来求前 i 个元素的最值，能求区间最值吗？
- 树状数组可以向二维扩展（询问一个矩阵的权值和）

线段树和树状数组

线段树	树状数组
可以维护任意区间 维护的值必须满足“区间加法” 常数比树状数组大 不容易扩展到多维情况 一维情况下，树状数组能胜任的，线段树一般都能解决	只能维护前缀和及其转化 若要进行区间询问，维护的值必须满足“区间减法” 代码简短，常数较小 较容易扩展到多维情况

并查集

- 并查集即是描述不相交集合的数据结构，并支持相关操作

- 将编号分别为 1…N 的 N 个对象划分为不相交集合，在每个集合中，选择其中某个元素代表所在集合

- 常见两种操作：

1. 合并两个集合
2. 查找某元素属于哪个集合 / 查询某两个元素是否同属一个集合

并查集

- 用编号最小的元素标记所在集合
- 定义一个数组 fa[n]，其中 fa[i] 表示元素 i 所在的集合

i	1	2	3	4	5	6	7	8	9	10
fa[i]	1	2	1	4	2	6	1	6	2	2

- 不相交的集合：{1,3,7}、{4}、{2,5,9,10}、{6,8}

并查集

- 这样，并查集只需要支持两种操作：

1. **join**：将两个集合合并，将集合内的代表元素（红色）作为参数传入
2. **find**：查找一个元素所在集合的代表元素（红色）

- 不相交的集合：{1,3,7}、{4}、{2,5,9,10}、{6,8}

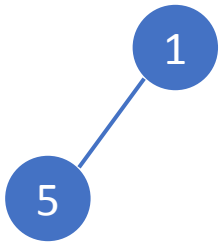
存在的问题

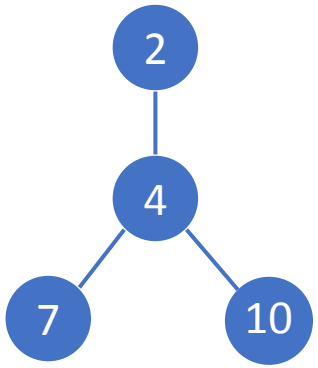
- 如果是线性数据结构，对于合并操作，必须全部扫一遍
- 碰到类似的状况，一般的想法是能否借助于树

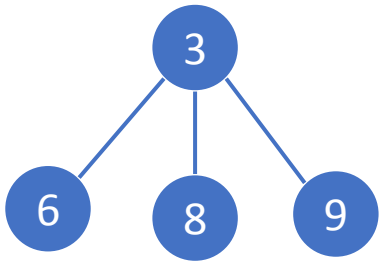
改进

- 每个集合用一棵树表示
- 定义一个数组 fa[ ]：其中 fa[i] = i，表示元素 i 代表本集合，并且是这棵树的根；
- fa[i] = j，j != i，则 j 是 i 的父节点

i	1	2	3	4	5	6	7	8	9	10
fa[i]	1	2	3	2	1	3	4	3	3	4



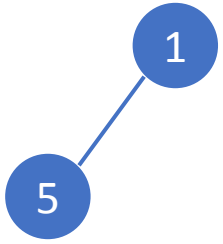


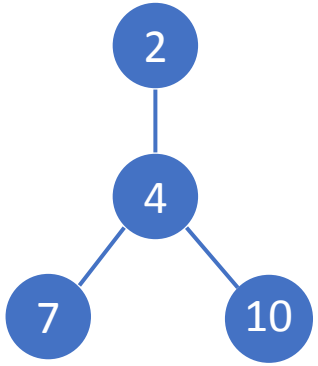


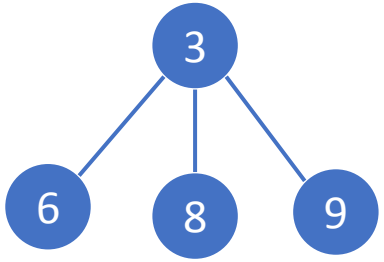
并查集

- fa[i] = j，j != i，则 j 是 i 的父节点
- 这样，每个节点我们只需要记录其父亲节点即可，如果某节点的父亲节点为它自己，说明该节点是根节点

i	1	2	3	4	5	6	7	8	9	10
fa[i]	1	2	3	2	1	3	4	3	3	4



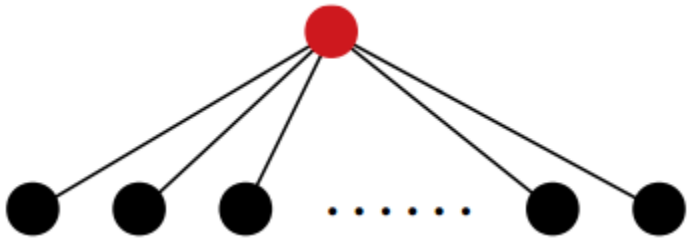




路径压缩

- 路径压缩是针对 find 操作的优化

- 由于并查集并没有删除操作，因此同一集合中的元素只增不减，所以一棵树的形态其实可以很“畸形”也没有任何影响。最理想的情况是如下的：



- 就是扁平化管理，没有中间商。。。

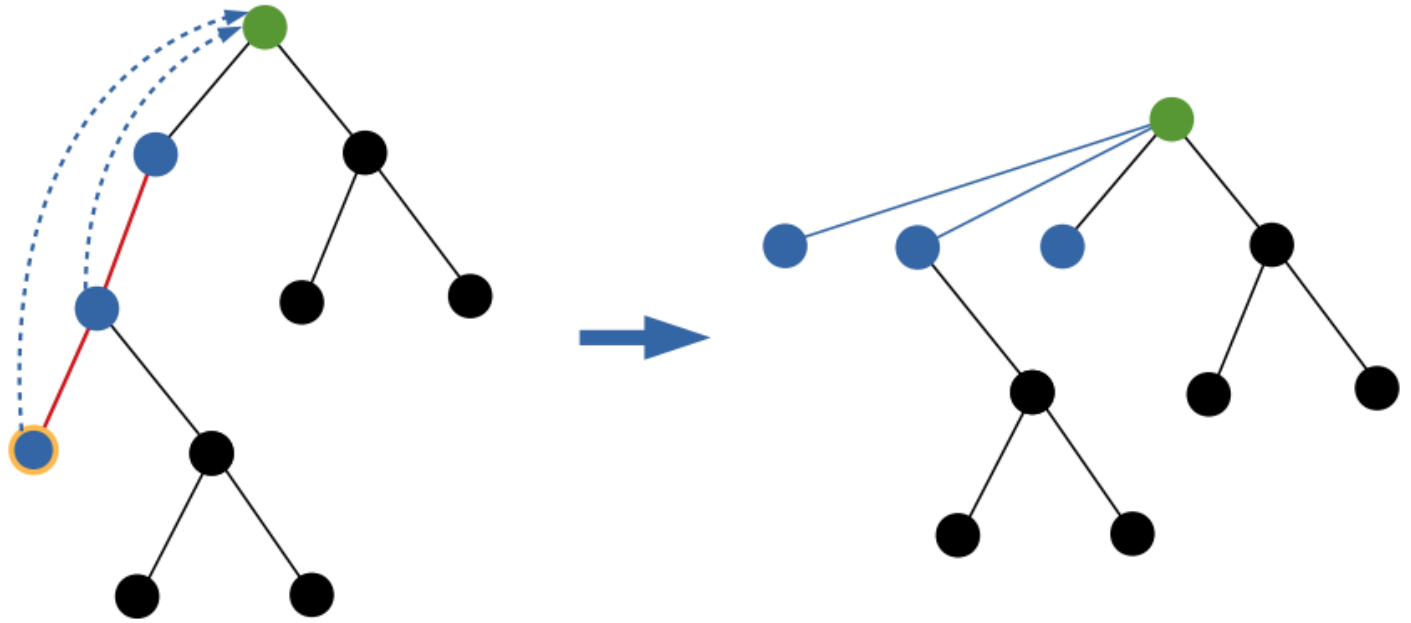
路径压缩

- 路径压缩的做法是：join 的时候什么都不做，而在每次 find 的时候，将所有访问到的元素的父亲节点全改为根节点，以便下次 find 时可以一步访问到根节点

- 步骤：

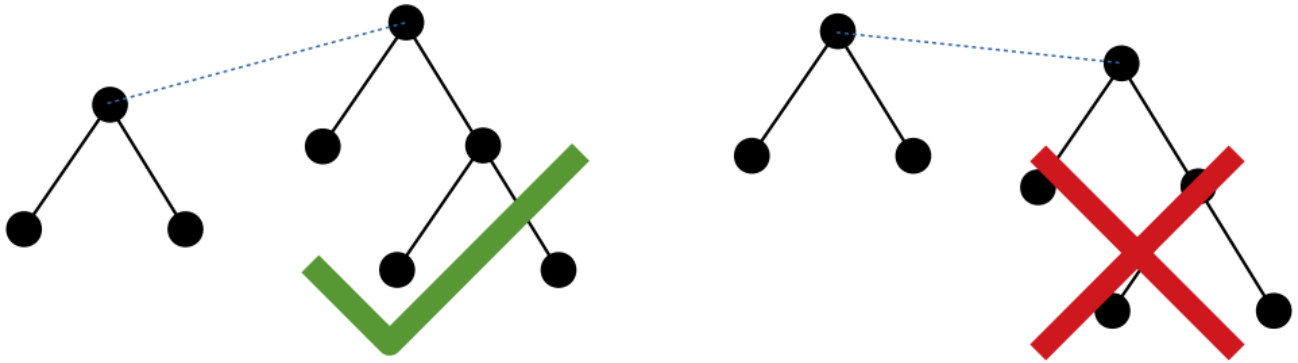
1. 找到根节点
2. 修改查找路径上的所有节点，使它们都指向根节点

路径压缩示意图



按秩合并

- 按秩合并是针对 join 操作的优化
- 记录每棵树的高度，将高度小的树合并到高度大的树



按秩合并

- 假设两棵树的高度分别是 h1 和 h2，则合并后树的高度 h 是：

max(h1，h2)　　// if h1 != h2

h1 + 1　　// if h1 == h2

- 任意顺序的合并后，包括 k 个节点的树最大高度不超过 ⌊logk⌋

按秩合并

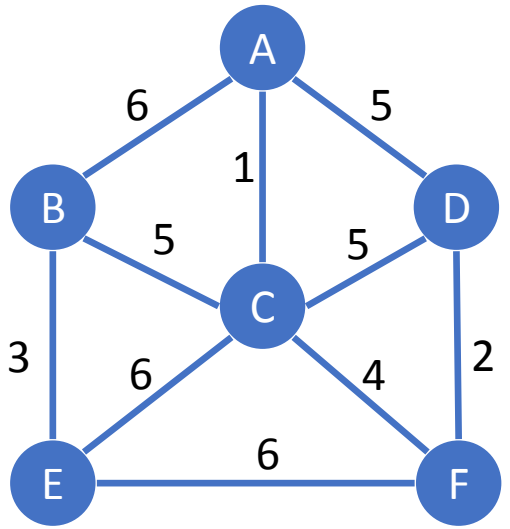
- 怎么记录每棵树的高度 h 呢？
- 我们可以为每个节点再开一个高度域，记录 h 值
  - h 值：所有点到根节点的路径上，经过的边的数量
- 但其实只有根节点的 h 值是有价值的，因为 h 的最大值（树的高度）只可能记录在根节点上

按秩合并

- 这个优化常被称为按秩合并，秩即高度
- 通俗来说就是：一般会使高度较小的树成为高度较大的树的子树，以便降低整棵树的高度

Kruskal借助于并查集的例子

顶点编号	A	B	C	D	E	F
所属集合	1	2	3	4	5	6
所属集合	1	2	1	4	5	6
所属集合	1	2	1	4	5	4
所属集合	1	2	1	4	2	4
所属集合	1	2	1	1	2	1
所属集合	1	1	1	1	1	1



集合

- 给定 A、B和 P，初始时为所有 [A，B] 范围内的整数 n 建立一个集合。之后，如果存在两个处在不同集合的数字 x 和 y，它们之间有不小于 P 的公共质因子，则将 x 和 y 所在的集合合并。问最后会剩下几个集合

  A，B ≤ 10^5，2 ≤ P ≤ B

分析

- 枚举 [P，B]中的所有质数 p，将 p 在范围 [A，B]内的所有倍数用并查集连起来，最后统计不同的集合个数

团伙

- 有 n 个强盗和 m 条信息，每条信息会表明某两个强盗之间是朋友关系还是敌人关系
- 并且他们坚信：
  - 朋友的朋友是朋友
  - 敌人的敌人也是朋友
- 朋友之间会构成团伙。现在根据 m 条信息推出 n 个强盗之间会构成多少个团伙

  n，m ≤ 10^5

分析

- 如果关系是朋友，就直接连起来
- 注意到一个人的所有敌人之间都是朋友关系，所以如果关系是敌人，先存起来，最后每个人的所有敌人之间都连起来即可

食物链

- 动物王国中有三类动物 A、B、C，构成了循环的食物链：A 吃 B，B 吃 C，C 吃 A
- 现在有 n 个动物（但我们不知道这些动物具体是哪一类的），和 m 句话。这 m 句话依次说出来，并且是以下两种说法之一：

1. 动物 x 和动物 y 是同类
2. 动物 x 吃动物 y

- 如果一句话自相矛盾或者与之前说过的真话有冲突，则为假话，否则为真话。需要数出m 句话中假话的数量

分析

- 由于不清楚动物的具体信息，所以尝试用假设的方法来处理。如果动物 x 捕食动物 y，那么当 x 为 A 时 y 就为 B，x 为 B 时 y 就为 C，……相当于是两者不同状态之间的等价关系
- 因此为每个动物建 3 个点，分别表示其为 A、B 和 C 时的状态。使用并查集来处理这些等价关系，同时可以查询“x 为 A，y 为 B”是否可以从之前的真话推出来

分析

- 上面是针对真话的处理方法，在这之前还需要判断这一句话是不是真话。我们可以查询与这句话相矛盾的情况
- 在并查集中查询。如果并查集中没有查出，则为真话

程序自动分析

- 有 10^9 个变量，分别为 x_1 至 x_{10^9}，此外还有 n 个已知条件。每个已知条件会指定两个数字 i 和 j，要么是 $x_i = x_j$，要么是 $x_i \neq x_j$。需要判断这 n 个条件是否有冲突

  $n \leq 10^5$

分析

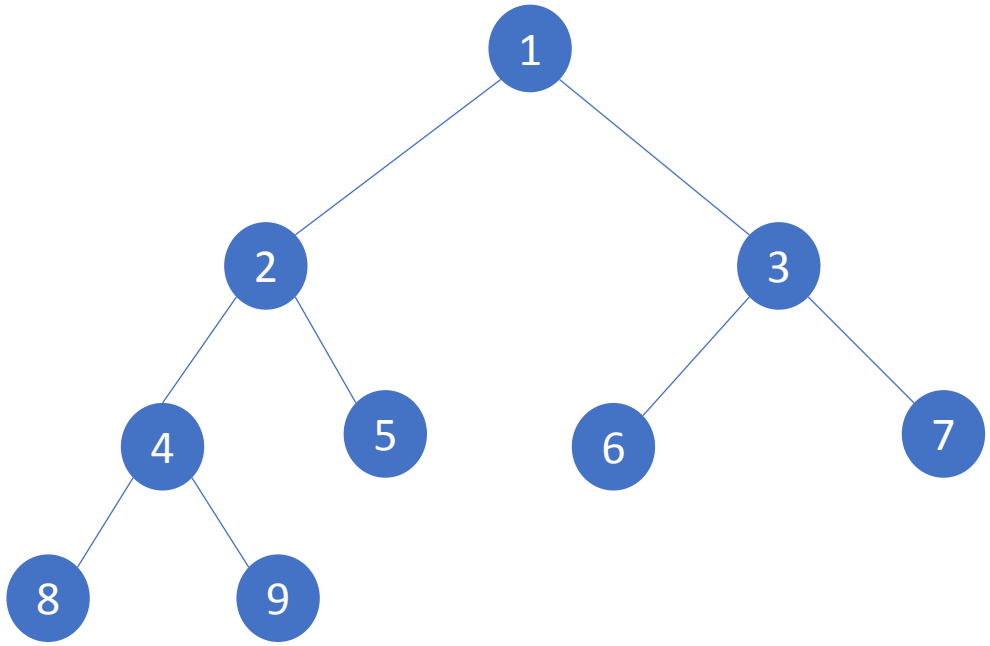
- 等式具有传递性，将相等的变量放入一个集合内，用并查集进行维护。处理完所有等式之后就可以知道哪些变量是相等的了

分析

- 然后考虑所有的不等式是否能满足要求。如果不等式两边的变量在同一个集合内，就说明发生了冲突

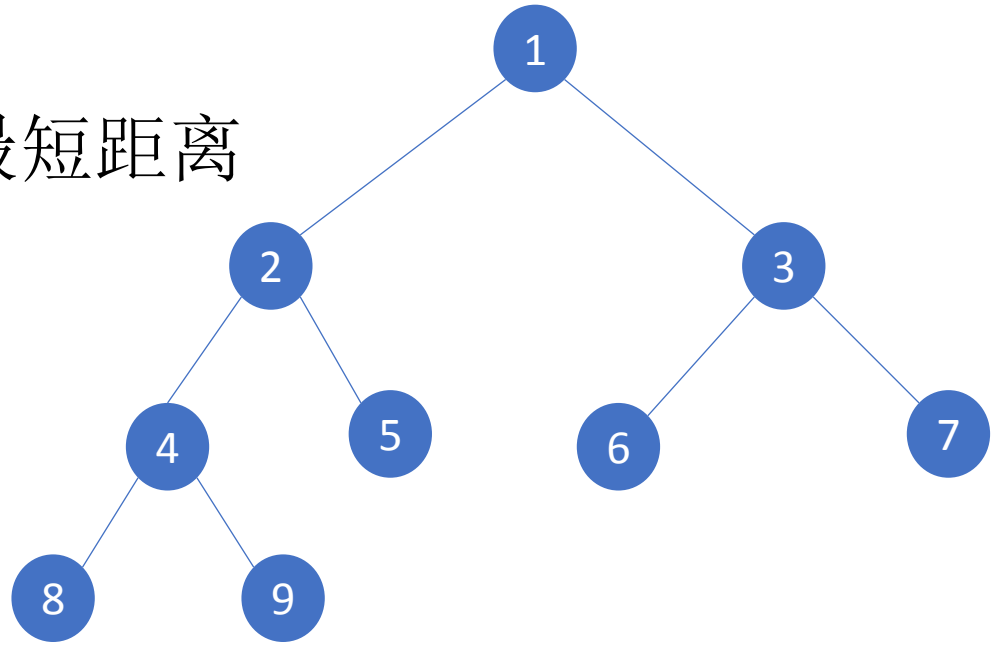
最近公共祖先（LCA）

- LCA(8,9)=4
- LCA(8,5)=2
- LCA(5,7)=1



最近公共祖先（LCA）

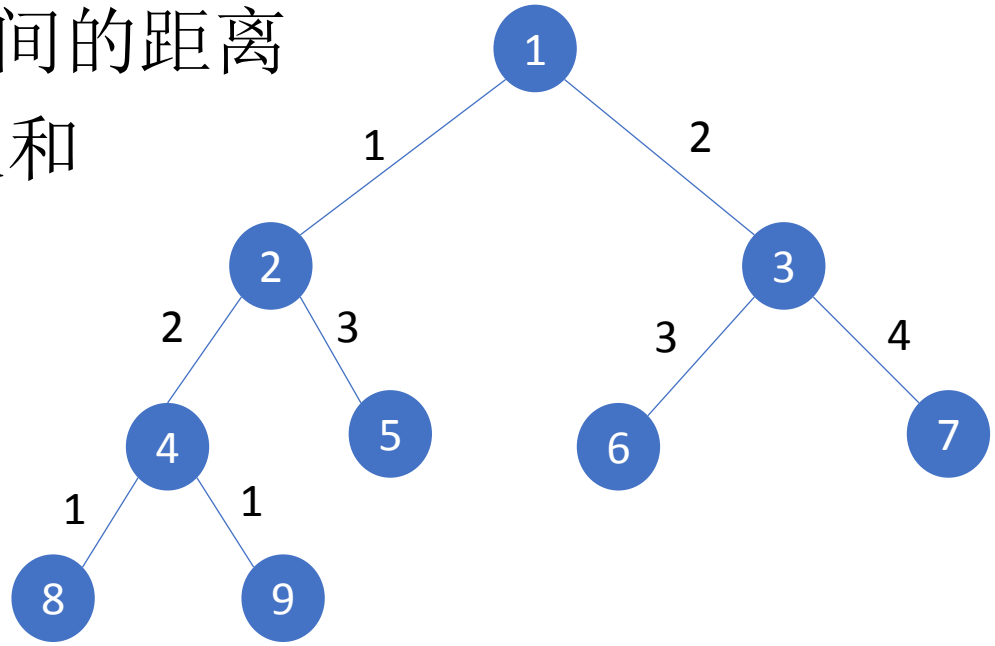
- 代码上怎么判断？
- 先定义深度函数d(u)：u到根节点的最短距离
  - d(1)=0，d(5)=2，d(9)=3



- 那么LCA(u,v)可以定义为：u、v的所有祖先节点中d最大的节点

LCA的作用

- 可以在O(1)时间内求出树上任意两点间的距离
- 如果是带边权，那就是两点间的边权和

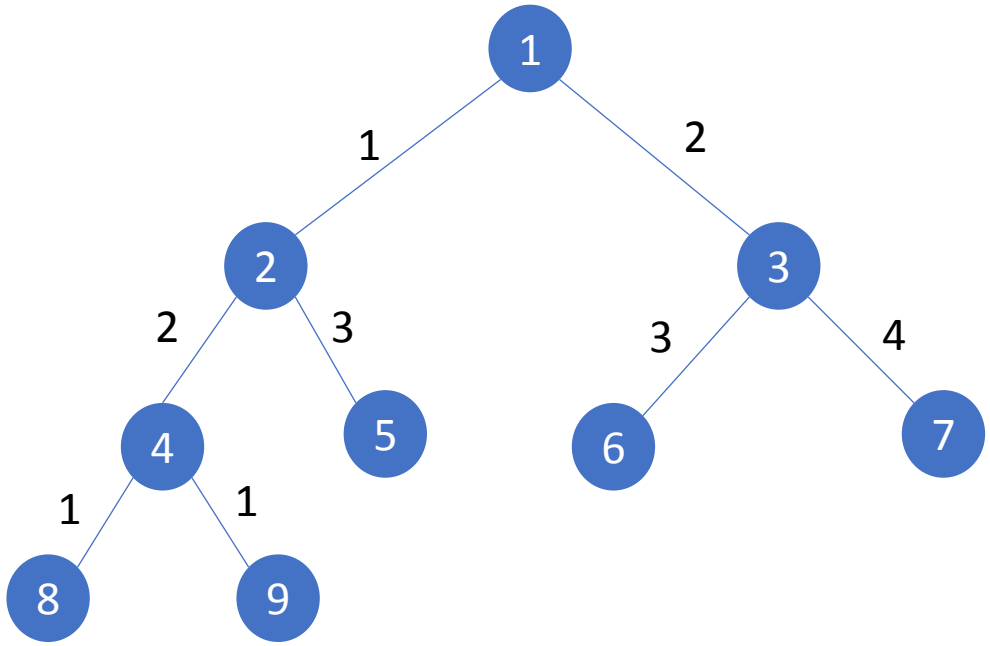


LCA的作用

- 怎么计算？
- dis[v]可以根据下面的公式BFS预处理出来：

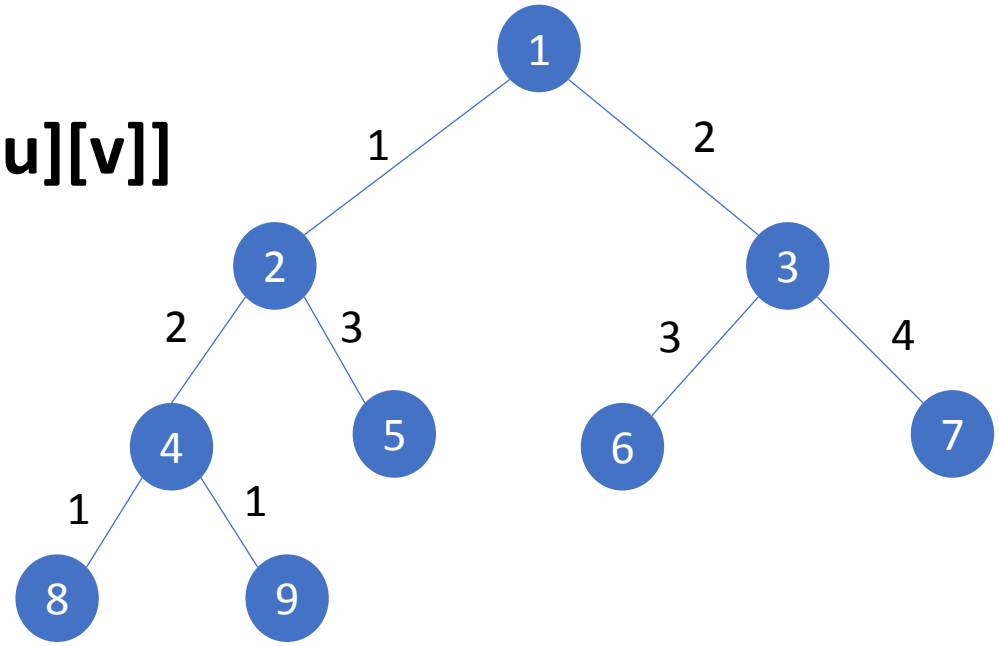
dis[v] = dis[u] + w[u][v]

– dis[2] = 1，dis[4] = 3，dis[5] = 4，dis[9] = 4

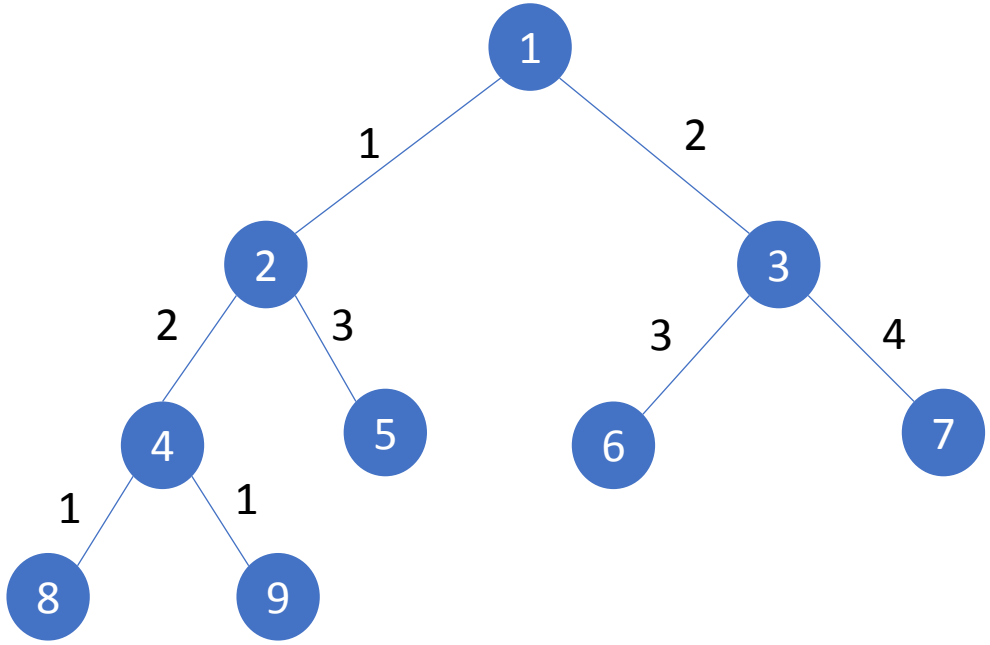


LCA的作用

- 怎么计算？
- **dis[u][v] = dis[u] + dis[v] – 2 * dis[LCA[u][v]]**
  - dis[4][5] = 3+4-2*1 = 5
  - dis[9][6] = 4+5-2*0 = 9



求LCA的Tarjan算法



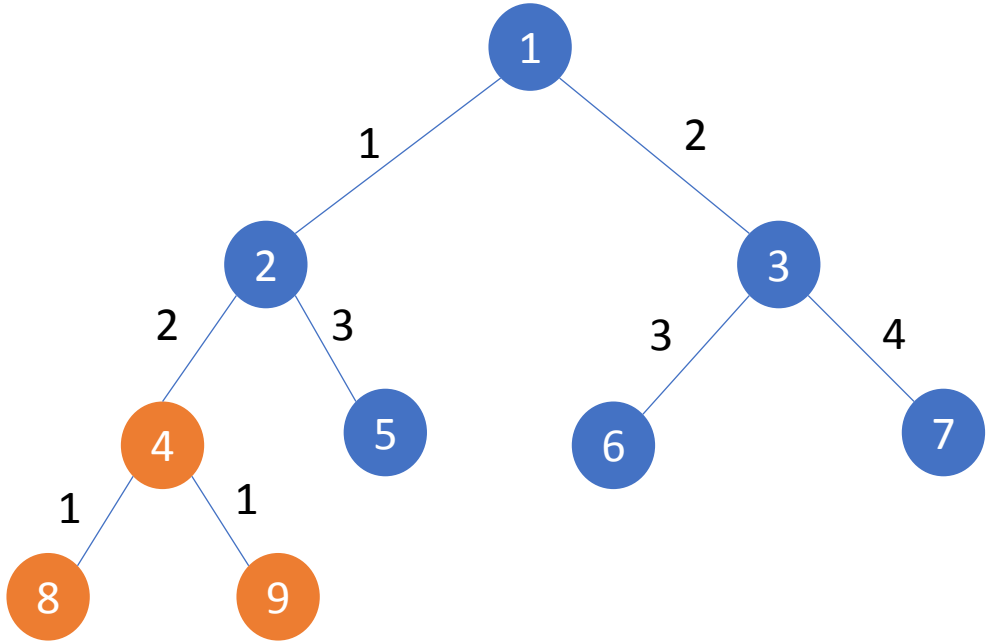
- 类似于DFS的递归调用过程

1. 首先遍历到8号节点
2. 没有节点被处理，返回4号节点
3. 此时[8,4]构成一个集合，4是这个集合的LCA
4. DFS到9号节点
5. 假设我们询问了LCA(8,9)，那么在处理9时，会发现8被标记过了，8所在的集合的LCA是4

而9是从4递归下来的，因此9和8的公共祖先中会有4。又因为DFS的特点会尽可能处理深度较大的节点，因此LCA(8,9) = 4

求LCA的Tarjan算法

- Tarjan算法运用了并查集的思想
- 时间复杂度O(n+q) q为询问次数
- 空间复杂度O(n^2)

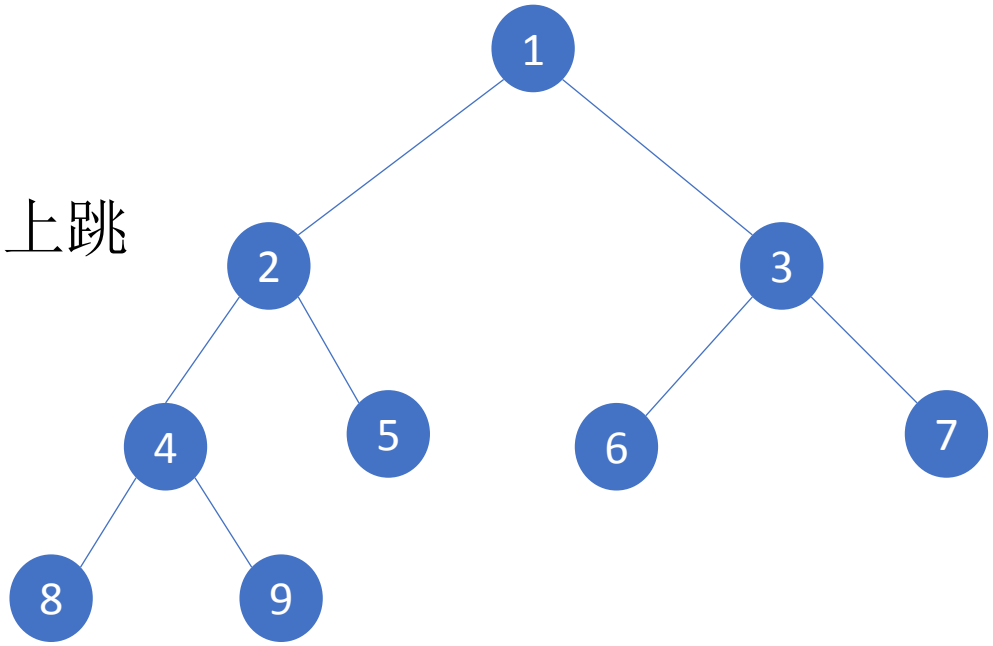


- Tarjan算法求LCA需要离线操作，而且内存开销也比较大

求LCA的倍增算法

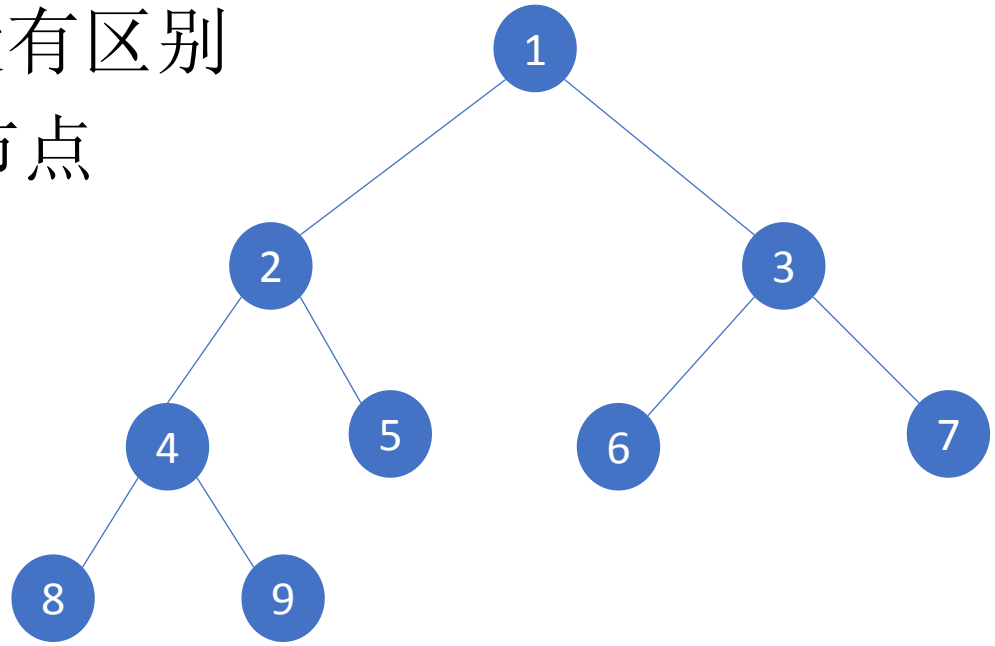
- 倍增算法的基本思路是：

1. 首先将 u,v 跳到同一深度
2. 如果此时 u,v 不是同一个点，则继续同时上跳
3. 直到 u,v 是同一个点为止



求LCA的倍增算法

- 但是如果一步一步跳，那就和暴力没有区别
- 设 f[i][j] 数组记录离节点 i 相距 2^j 的节点
- 即：dis[i][f[i][j]] = 2^j
  - f[8][0] = 4，f[9][1] = 2



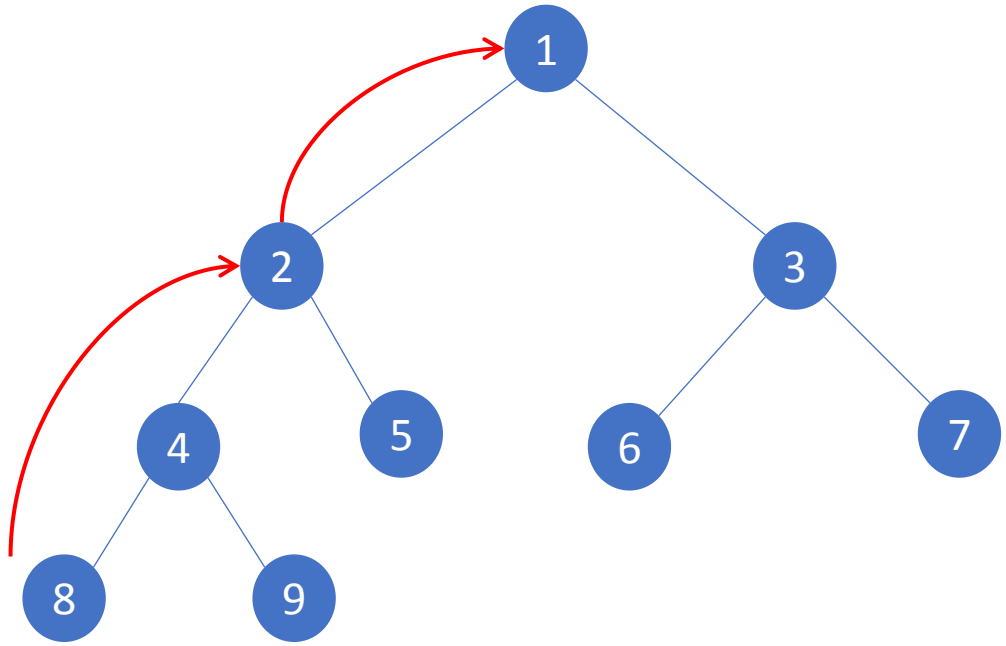
状态转移方程如下：

f[i][j] = f[f[i][j-1]][j-1]

求LCA的倍增算法

- 得到了 f 数组，那么节点可以每次 2^j 的距离上跳

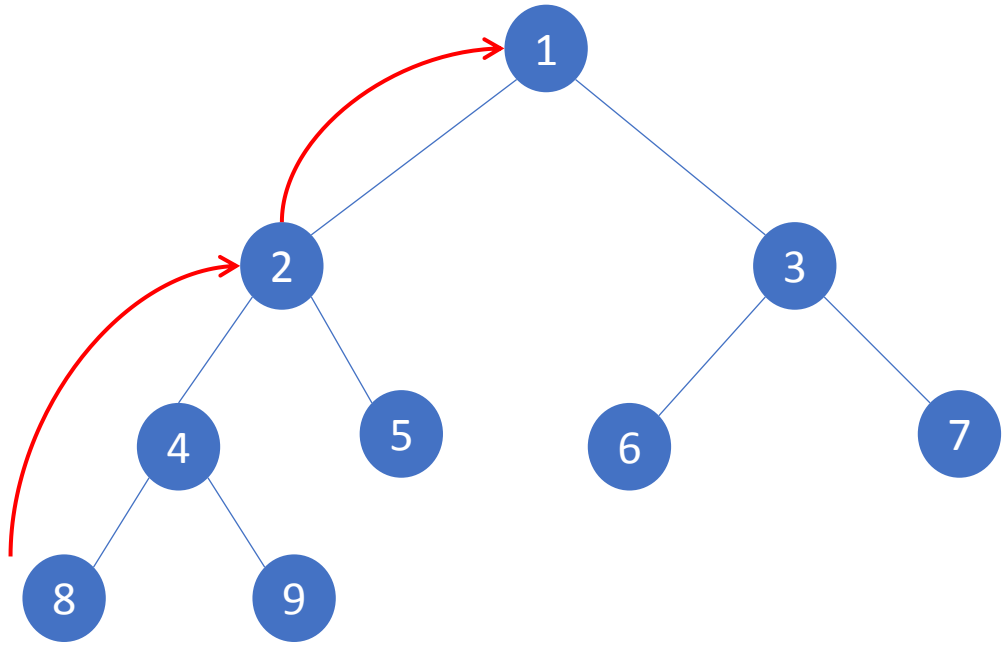
- 两个节点到LCA的深度差可以被分解成多个2的幂之和，因此我们逐步将它们上移至LCA的位置



求LCA的倍增算法

- 得到了 f 数组，那么节点可以每次 2^j 的距离上跳

- 但是为了方便判断是否跳的太远，我们不让它们最后跳到LCA的位置，而是跳到LCA的儿子节点处
- 此时其父节点就是LCA



推荐题单

- 海港（队列模拟）
- 表达式求值（栈模拟）
- 时间复杂度（栈模拟）
- 奶酪（并查集）
- 蚯蚓（优先队列）
- 列队（平衡树/线段树/树状数组）