21级实验室暑假第五讲

目录

- 基础线段树
- 树状数组
- 离散化

基础线段树

引入:Problems with Intervals（区间问题）

- ✓ 区间查询

  询问某段区间的某些性质（极值(最大值,最小值)，求和等等)
- ✓ 区间更新

  某些操作影响了某段区间（统一加一个值,统一乘一个值等等)
- ✓ 三个问题
  - ➢ 更新点，查询区间
  - ➢ 更新区间，查询点
  - ➢ 更新区间，查询区间

  其中**只更新和查询点**的操作可直接利用**数组** 进行O(1)更新和O(1)的查询，关键如何解决前面三种问题

基础线段树

引入:Problems with Intervals（区间问题）

- 一个长度为N的一维数组(a[1]~a[N])
- 我们每次对该数组有一些操作:
- 1、修改数组中某个元素的值
  - 【1,**5**,4,1,6】---(a[2]=3)---> 【1,**3**,4,1,6】
- 2、询问数组中某段区间的最大值
  - 【**1,5,4,1**,6】---(max(1,4)=?)---> 5
- 3、询问数组中某段区间的和
  - 【1,5,**4,1,6**】---(sum(3,5)=?)---> 11

基础线段树

引入:Problems with Intervals（区间问题）

如果只有一次询问?

- 枚举相应区间内的元素，输出答案　O(N)

更多的询问?

- Q次询问，O(NQ) That's too SLOW!

主要问题在于询问是针对区间的，而我们维护的信息是针对每个元素的!

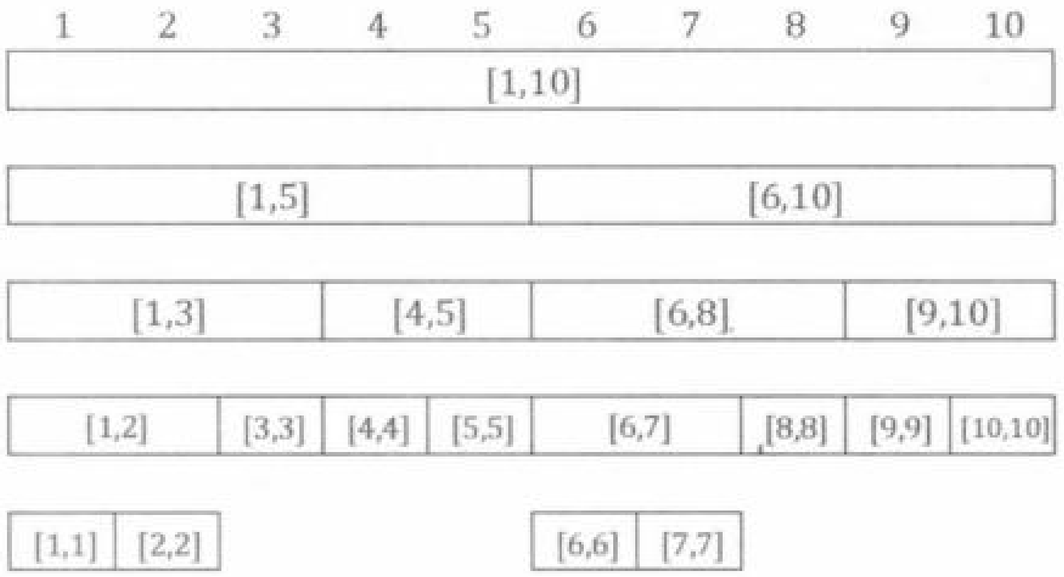
线段树——在O(logN)的时间内完成每次操作 O(QlogN)

- Less than 1 seconds when N=Q=100000

基础线段树

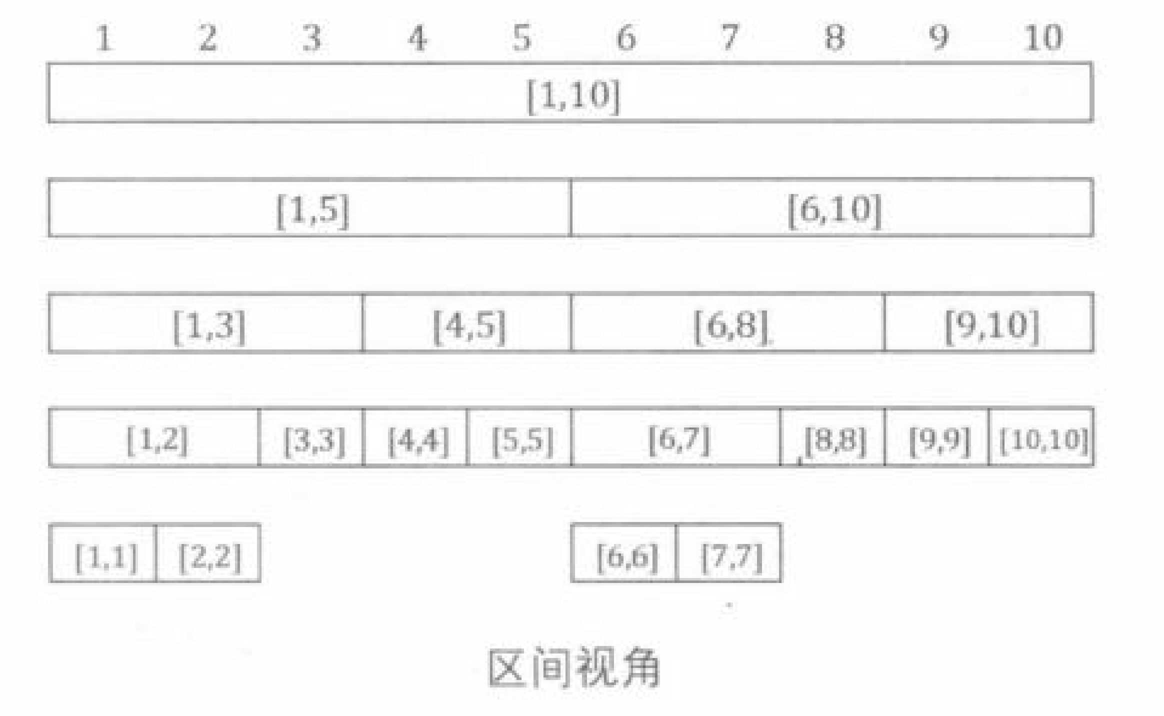
线段树(Segment Tree) 是一种基于分治思想的二叉树结构，用于在**区间上进行信息统计**。与按照二进制为(2的次幂)进行区间划分的树状数组(后面讲)相比，线段树是一种更加通用的结构：

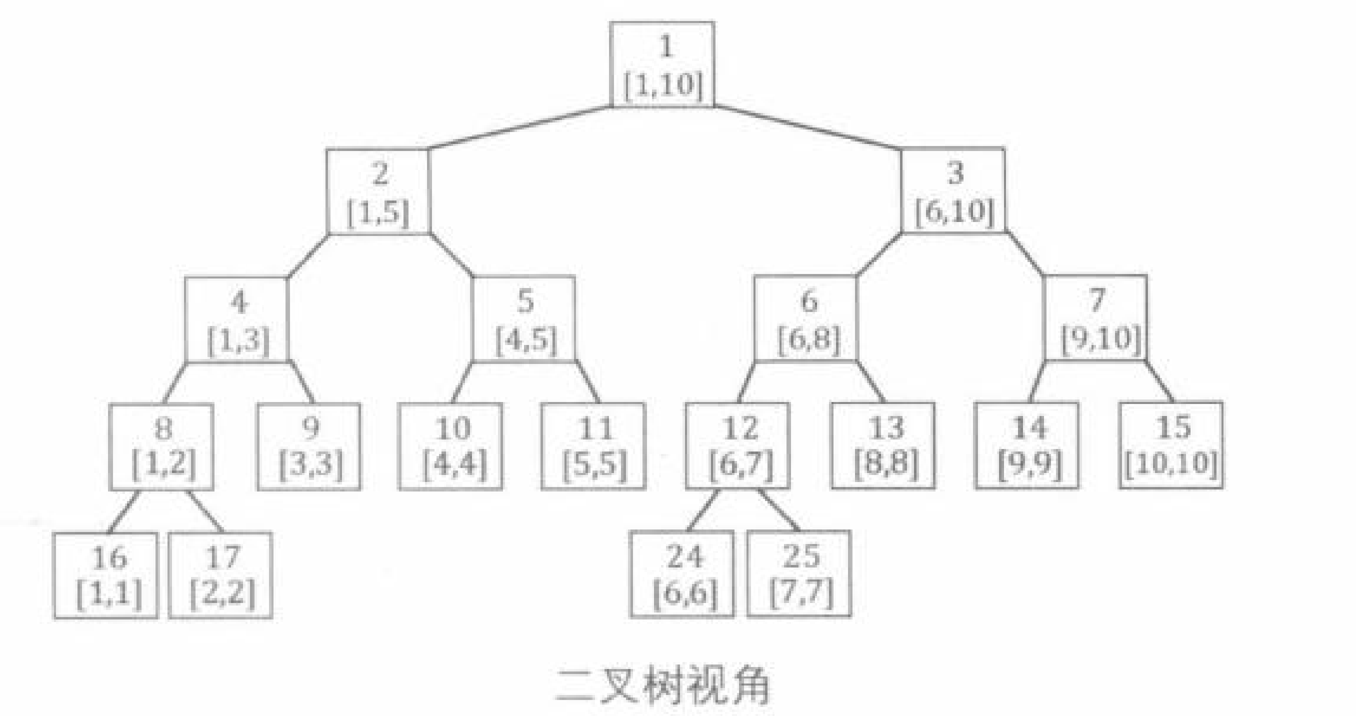
- 线段树的每个结点都代表一个区间；
- 线段树具有唯一的根结点，代表的区间是整个统计范围，如[1,N]；
- 线段树的每个叶子结点都代表一个长度为1的元区间[x,x]；
- 对于每个内部结点[l,r]，它的左子结点是[l,mid],右子结点是[mid+1,r],其中 $mid = \left\lfloor \frac{(l+r)}{2} \right\rfloor$



区间视角

基础线段树





上图展示了一棵线段树。可以发现，除去树的最后一层，整棵线段树一定是一棵完全二叉树，树的深度为O(logN)。因此，可以按照与二叉堆类似的“父子2倍”结点编号方法：

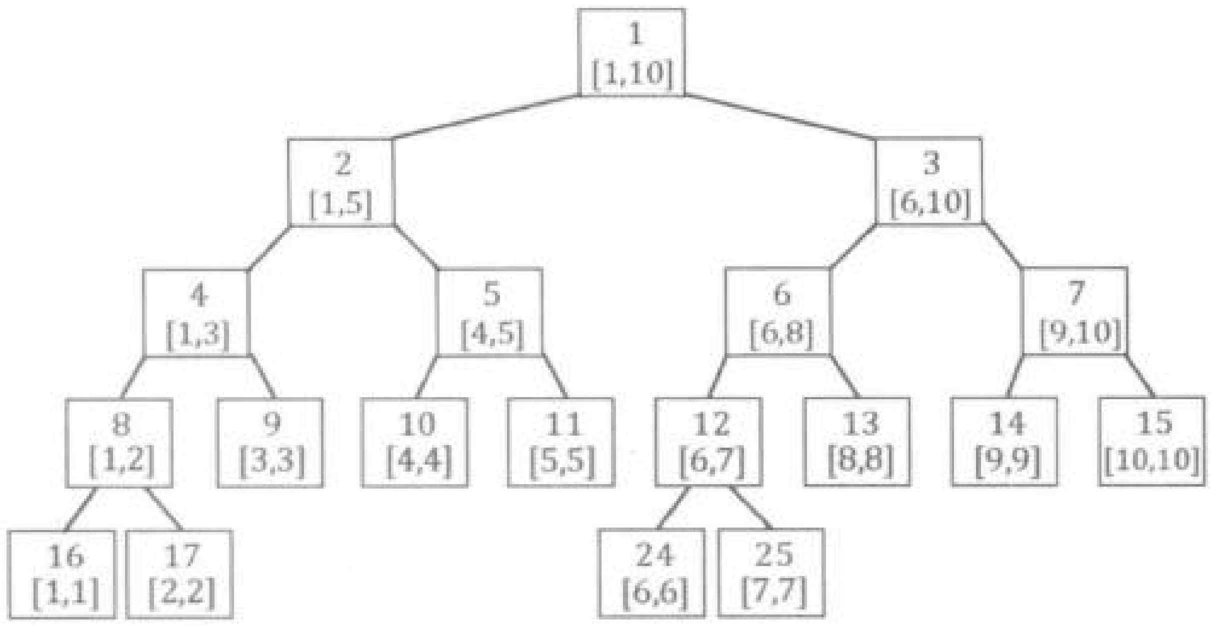
1. 根结点编号为1；
2. 编号为x的结点的左子结点编号为x*2,右子结点编号为2*x+1。

基础线段树

证明(错位相减法)：

$$N+\frac{N}{2}+\frac{N}{4}+\cdots+2+1=2\times(N+\frac{N}{2}+\frac{N}{4}+\cdots+2+1)-(N+\frac{N}{2}+\frac{N}{4}+\cdots+2+1)$$

$$=(2N+\boxed{N+\frac{N}{2}+\cdots+2})-(\boxed{N+\frac{N}{2}+\frac{N}{4}+\cdots+2}+1)=2N-1$$



二叉树视角

这样一来，就能简单地使用一个结构体struct 数组来保存线段树。当然，树的最后一层结点在数组保存的位置不是连续的，直接空出来数组中的多余位置即可。在理想状态下，N个叶结点的满二叉树有

$$N+\frac{N}{2}+\frac{N}{4}+\ldots+2+1=2N-1$$

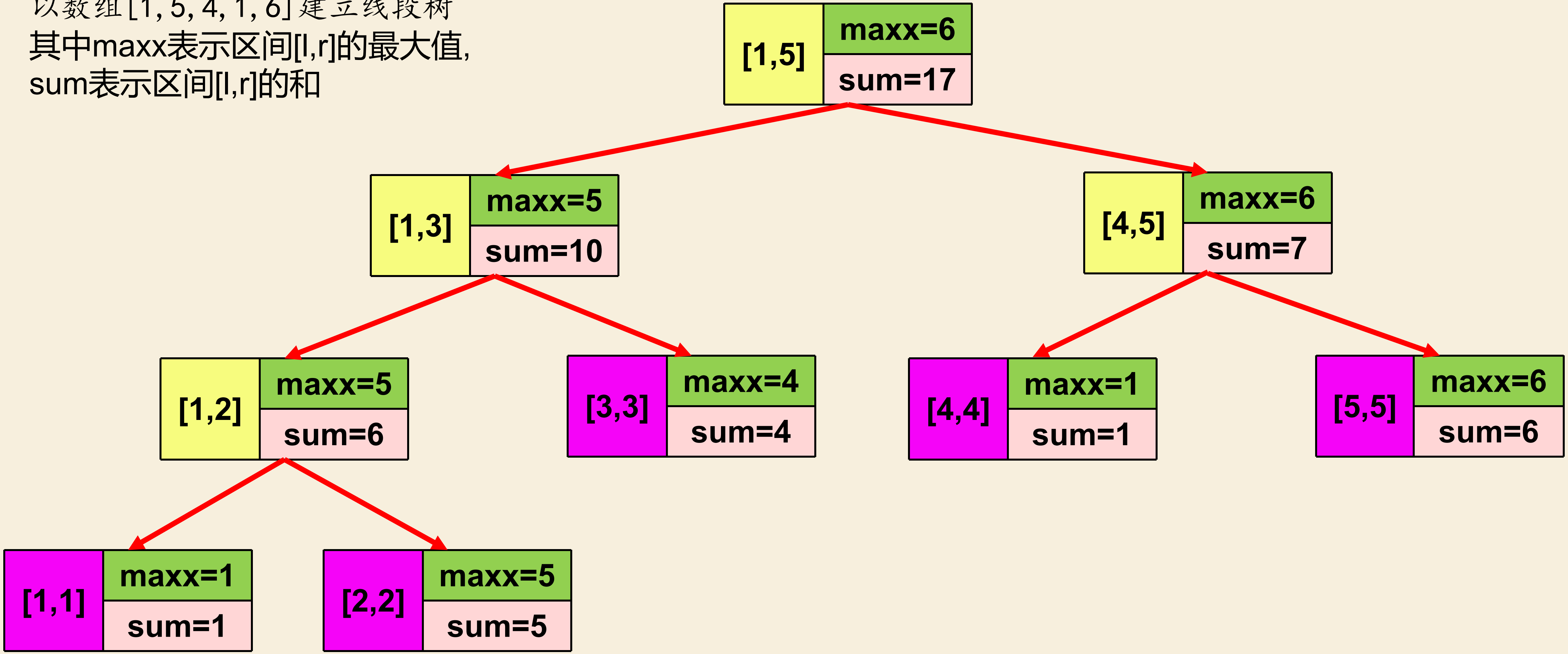
个结点。因此在上述存储方式下，最后还有一层产生了空余，所以**保存线段树的数组长度要不小于4N**才能保证不越界。

基础线段树

线段树空间应开为原数组长度的4倍

基础线段树

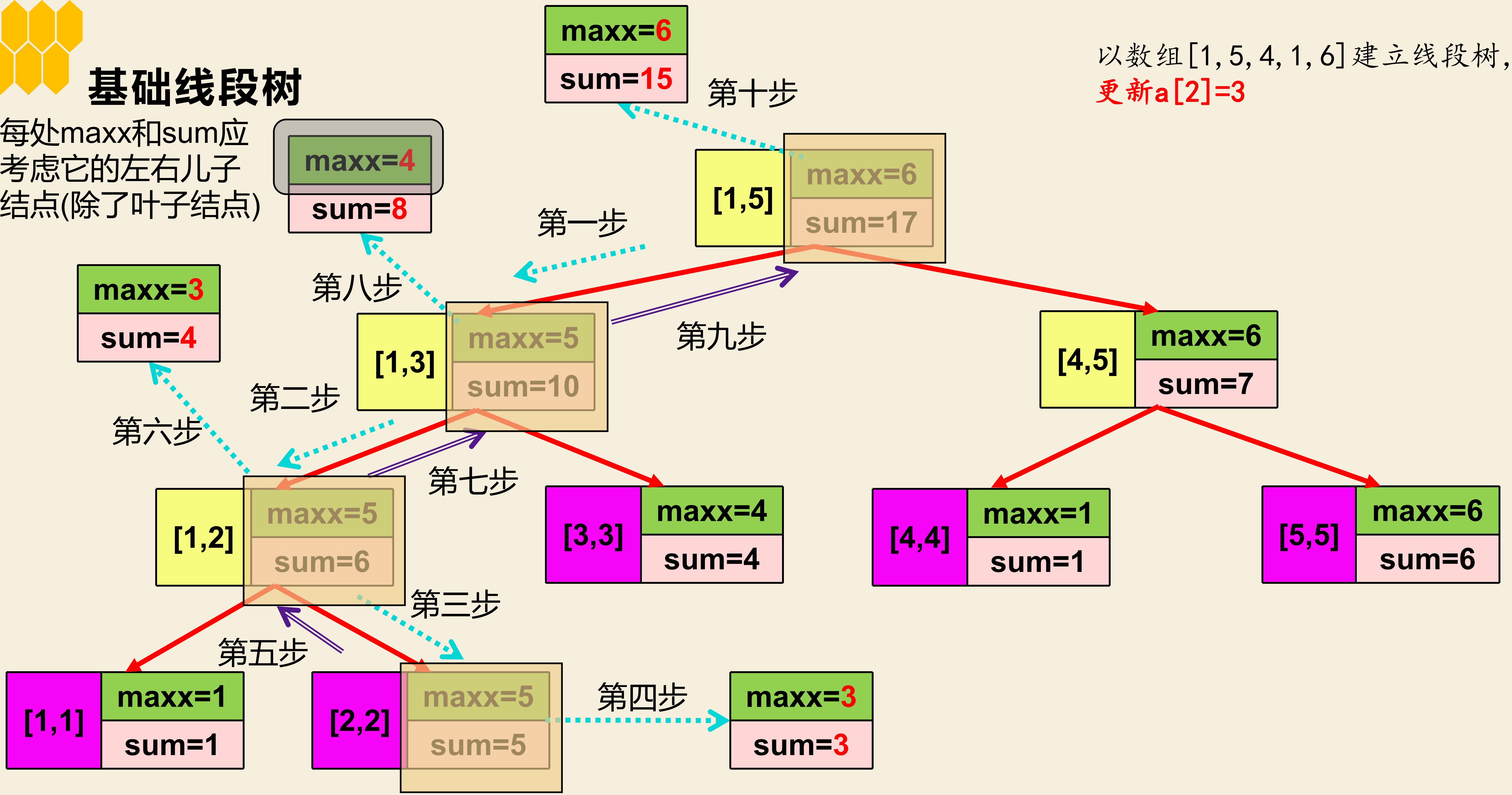
以数组[1, 5, 4, 1, 6]建立线段树
其中maxx表示区间[l,r]的最大值,
sum表示区间[l,r]的和



基础线段树

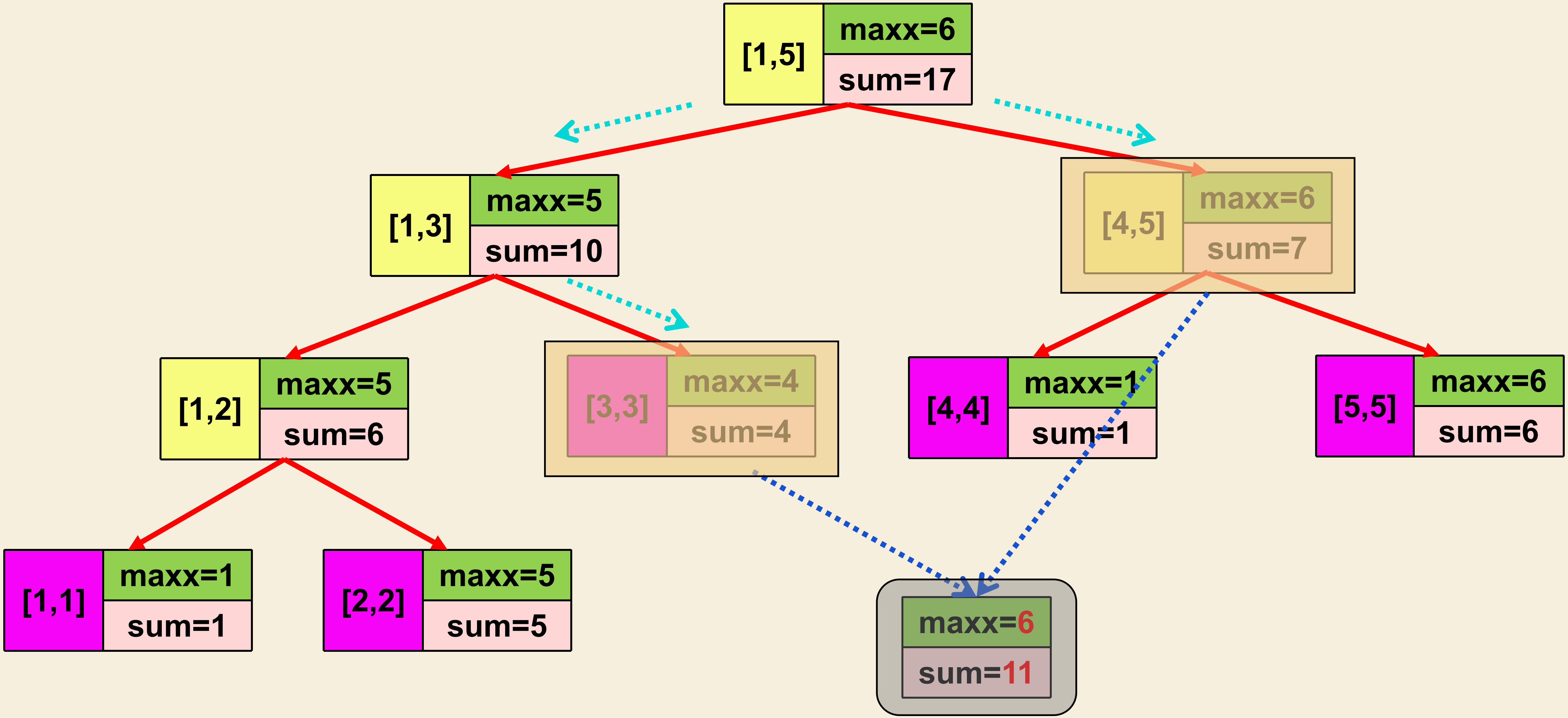
每处maxx和sum应考虑它的左右儿子结点(除了叶子结点)

以数组[1, 5, 4, 1, 6]建立线段树，
更新a[2]=3



基础线段树

以数组[1,5,4,1,6]建立线段树，
查询区间[3,5]的最大值以及和



基础线段树

每一个结点记录的结构体信息

```
const MAXN 100010

struct Tree{
    int l;//该结点对应区间的左端点
    int r;//该结点对应区间的右端点
    int dat;//维护的信息,视具体题目而定
    //例如: int maxx,sum,add,mul;最大值,求和,加数懒标记,乘数懒标记
}tree[4*MAXN];//注意此处空间开4倍
```

已知当前结点的编号是root，如何记录左儿子结点和右儿子结点的编号？mid怎么计算？

```
//左儿子结点编号 2*root      root << 1
//右儿子结点编号 2*root+1    root << 1 | 1
//mid计算方式:
mid=(tree[root].l+tree[root].r)/2;
mid= tree[root].l+tree[root].r >> 1;
```

本课件主要以前一种方式进行表示(易于明白)，网上大部分基本上是以第二种方式表示左右结点编号和mid

基础线段树

线段树常写的5个函数

1. pushup()　上传操作

2. pushdown()　下沿操作

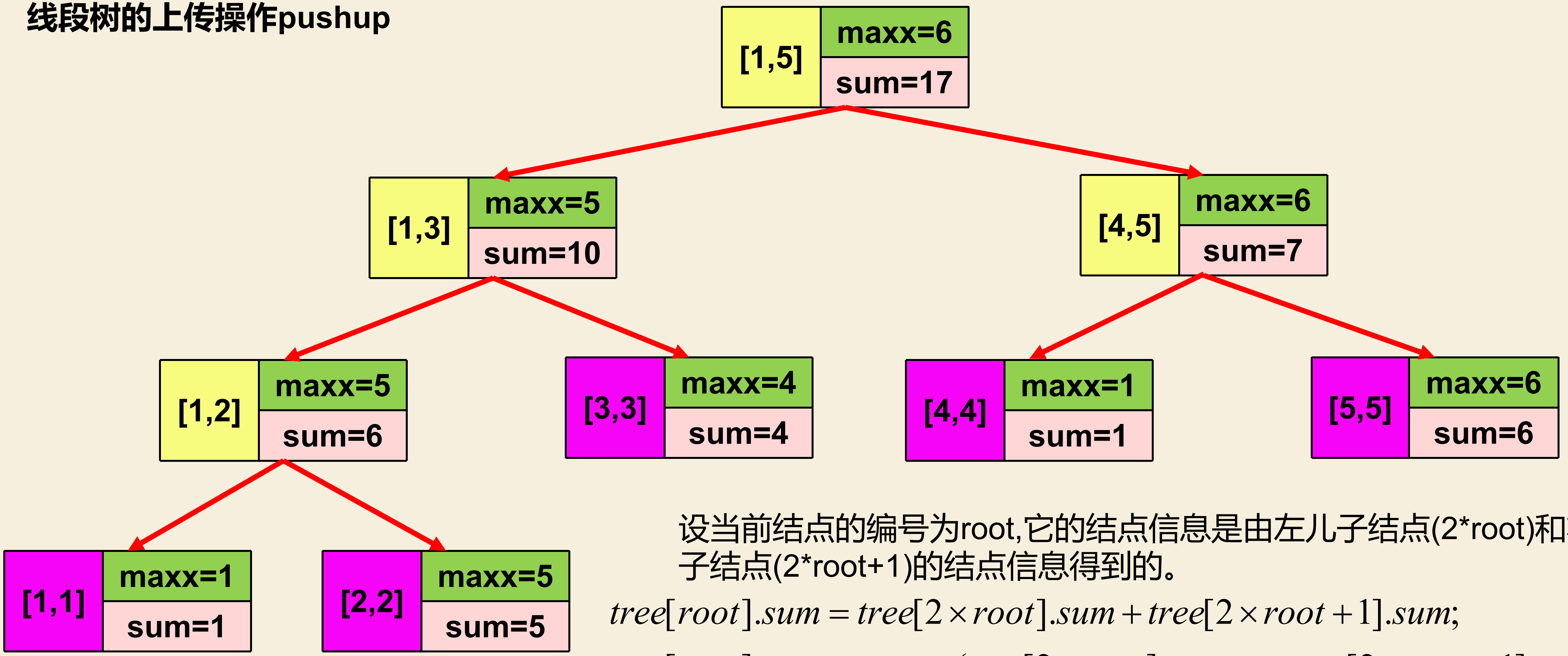
3. build()　建树操作——建树操作有pushup操作

4. modify()　修改操作(包括单点修改和区间修改)
区间修改含有pushdown操作和pushup操作,单点修改不含pushdown操作

5. query()　查询操作(包括单点查询和区间查询)——查询操作无pushup操作
单点查询可能含有pushdown操作，区间查询可能含有pushdown操作(如果是只有**单点修改**时没有pushdown操作，如果是有**区间修改操作**时有pushdown操作)

基础线段树

线段树的上传操作pushup



设当前结点的编号为root,它的结点信息是由左儿子结点(2*root)和右儿子结点(2*root+1)的结点信息得到的。

$$tree[root].sum = tree[2\times root].sum + tree[2\times root+1].sum;$$

$$tree[root].\max x = \max(tree[2\times root].\max x + tree[2\times root+1].\max x);$$

基础线段树

线段树的上传操作pushup

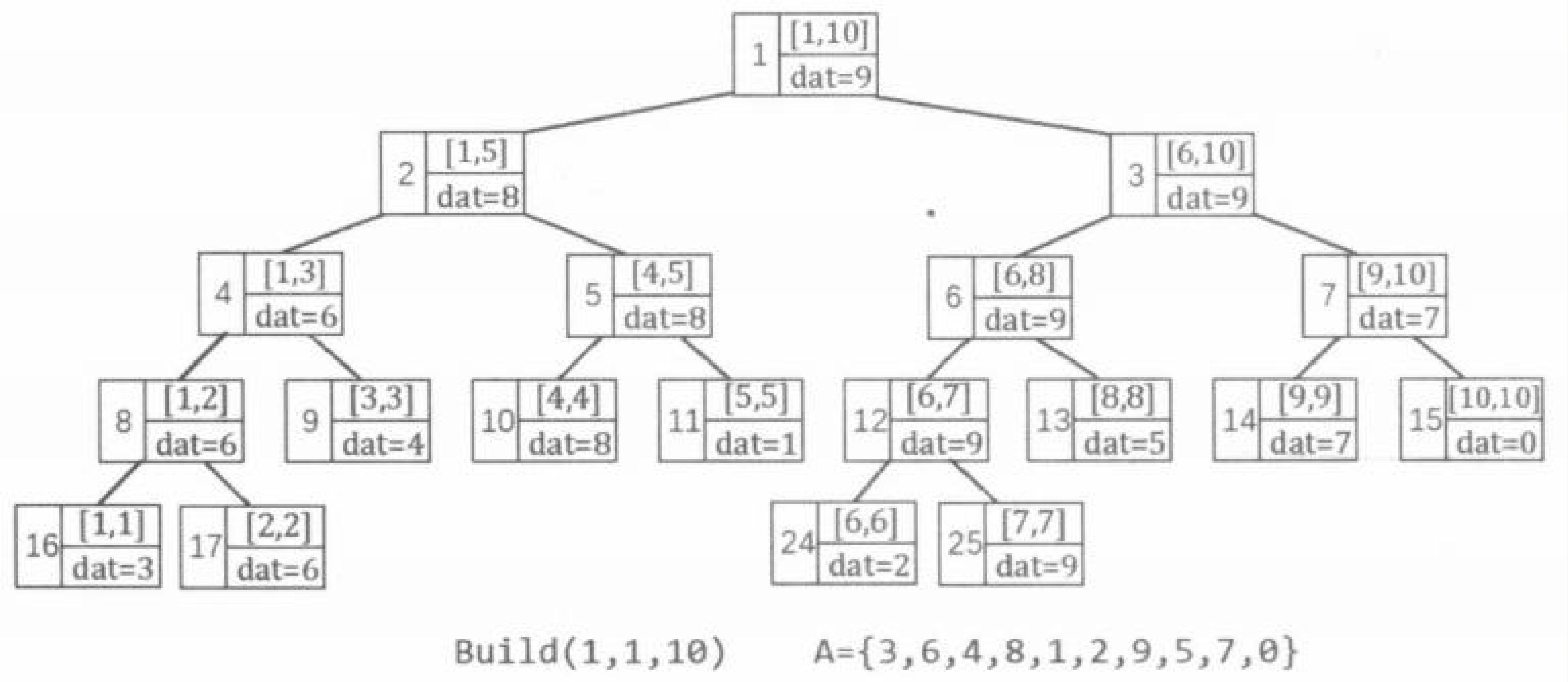
```
void pushup(int root)//上传操作,函数内部视具体题目而定
{
    tree[root].dat=tree[2*root].dat+tree[2*root+1].dat;
    //tree[root].sum=tree[2*root].sum+tree[2*root+1].sum;
    //tree[root].maxx=max(tree[2*root].maxx,tree[2*root+1].maxx);
}
```

基础线段树

线段树的建树build

线段树的基本用途是对序列进行维护，支持**查询与修改**指令。给定一个长度为N的序列A,我们可以在区间[1,N]上建立一棵线段树，每个叶子结点[i,i]保存A[i]的值。线段树的二叉树结构可以很方便地从下往上传递信息。以区间最大值为例，记dat(l,r)等于 $\max_{l\le i\le r}\{A[i]\}$,显然

$$dat(l,r)=\max(dat(l,mid),dat(mid+1,r)),\text{其中}mid=\frac{(l+r)}{2}$$



基础线段树

线段树的建树build

```
void build(int root,int l,int r){//建树操作
    tree[root].l=l,tree[root].r=r;//结点root的代表区间是[l,r]
        //一定用初始化,不然会段错误
        //线段树的题一般很容易就段错误,每个地方都要注意
    if(l==r){//到达叶子结点
        tree[root].dat=a[l];//保存序列a[l]的值
        return ;//此处写了return ;后面可不用else
    }
    int mid=(l+r)/2;//折半,此处可以写成(tree[root].l+tree[root].r)/2
    build(2*root,l,mid);//建立左儿子结点[l,mid],编号为2*root
    build(2*root+1,mid+1,r);//建立右儿子结点[mid+1,r],编号为2*root+1
    pushup(root);//从下往上传递信息,上传操作,函数内部视具体题目而定
}

build(1,1,n);//调用入口,建立以1为根结点,且总区间是[1,n]的线段树
```

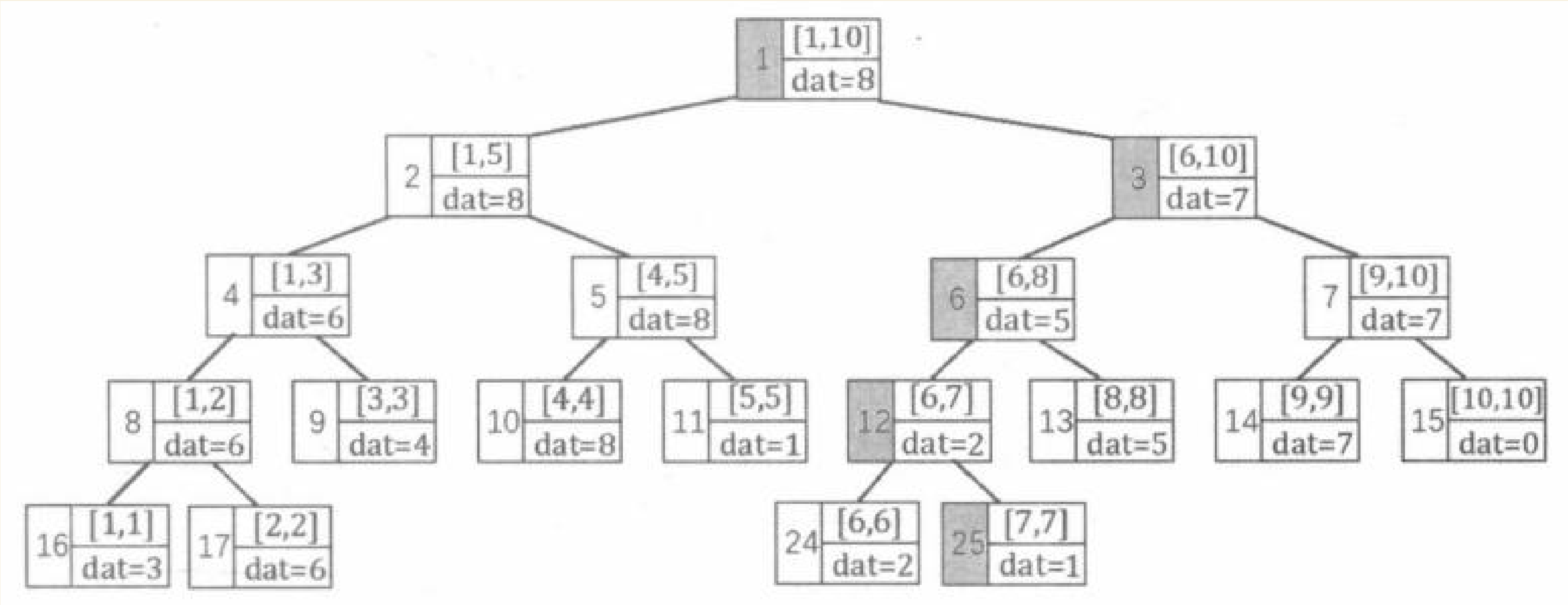
基础线段树

线段树的单点修改modify

单点修改时一条形如"C x v"的指令，表示把A[x]的值修改为v。

在线段树中，根结点(编号为1的结点)是执行各种指令的入口。我们需要从根结点出发递归找到代表[x,x]的叶子结点。然后从下往上更新[x,x]以及它的所有祖先结点的上保存的信息，时间复杂度为O(logN)。

将A[7]更新为1，如下图所示。



基础线段树

线段树的单点修改modify

```
void modify(int root,int pos,int val){//单点修改操作
    if(tree[root].l==pos&&tree[root].r==pos)//到达叶子结点[pos,pos]
        tree[root].dat=val;//修改叶子结点A[pos]的值为val
    else{//上一个if执行完可写return ;此时可不写else
        int mid=(tree[root].l+tree[root].r)/2;//折半
        if(pos<=mid)//pos在左半区间
            modify(2*root,pos,val);//只修改编号信息,其它值不变
        else modify(2*root+1,pos,val);//pos在右半区间
        pushup(root);//从下往上传递信息,上传操作
    }
}
```

基础线段树

线段树的单点查询query

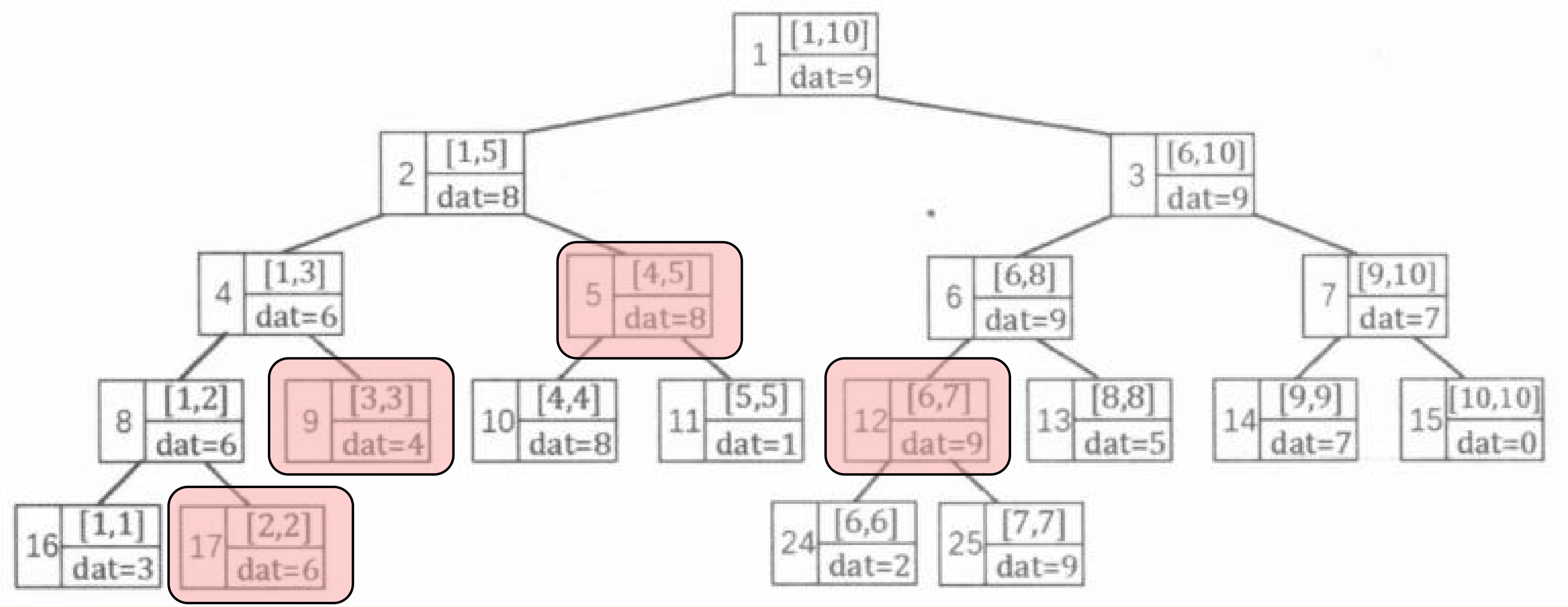
单点查询的思想和单点修改的思路类似，<u>从根结点出发递归找到代表[x,x]的叶子结点</u>返回结果。需注意的是如果还包含区间修改操作时，单点查询函数中应具有pushdown操作，它不需要pushup操作。

```
int query(int root,int pos){//单点查询操作
    if(tree[root].l==pos&&tree[root].r==pos)//到达叶子结点[pos,pos]
        return tree[root].dat;//返回叶子结点的值
    //pushdown(root);如果该题含有区间修改时一般添加该句
    int mid=(tree[root].l+tree[root].r)/2;
      //注意是该结点区间[tree[root].l,tree[root].r]的mid
    if(pos<=mid)//pos在左半区间
        return query(2*root,pos);
    else return query(2*root+1,pos);//pos在右半区间
}
```

基础线段树

线段树的区间查询query

查询区间[2,7]的最大值
结果为红色区域的dat最大值9



区间查询是一条形如“Q l r”的指令,例如查询序列A在区间[l,r]上的最大值，即 $\max_{l\le i\le r}\{A[i]\}$。我们只需要从根结点开始，递归执行一下过程:

1. 若[l,r]完全覆盖了当前结点代表的区间，则立即回溯，并且该结点的dat值为候选答案；
2. 若r<=当前结点代表区间的mid时，即区间[l,r]只在左区间，只递归访问左子结点；
3. 若l> 当前结点代表区间的mid时，即区间[l,r]只在右区间，只递归访问右子结点；
4. 若区间[l,r]横跨左右子结点,都要进行递归。

2-4部分整理亦可理解为：

2. 若左子结点与[l,r]有重叠部分，则递归访问左子结点；
3. 若右子结点与[l,r]有重叠部分，则递归访问右子结点；

基础线段树

含4种情况的query函数

线段树的区间查询query

```
int query(int root,int l,int r){//区间查询操作
    if(l<=tree[root].l&&r>=tree[root].r)//第1种情况,完全包含
        return tree[root].dat;
    else{
        //pushdown(root);如果该题含有区间修改时一般添加该句
        int mid=(tree[root].l+tree[root].r)/2;
       //注意是该结点区间[tree[root].l,tree[root].r]的mid,不是区间[l,r]的mid
        if(r<=mid)//第2种情况,只递归左子结点
            return query(2*root,l,r);//只修改编号信息,其它值不变
        else if(l>mid)//第3种情况,只递归右子结点
            return query(2*root+1,l,r);//只修改编号信息,其它值不变
        else{//第4种情况,左右子结点都递归
            int res=0;
            res=max(query(2*root,l,r),query(2*root+1,l,r));//两种递归取最大值
            //res=query(2*root,l,r)+query(2*root+1,l,r);//两种递归求和
            return res;
        }
    }
}
```

基础线段树

浓缩为3种情况的query函数

线段树的区间查询query

```
int query(int root,int l,int r){//区间查询操作
    if(l<=tree[root].l&&r>=tree[root].r)//第1种情况,完全包含
        return tree[root].dat;
    else{
       //pushdown(root);如果该题含有区间修改时一般添加该句
        int mid=(tree[root].l+tree[root].r)/2;
       //注意是该结点区间[tree[root].l,tree[root].r]的mid,不是区间[l,r]的mid
       int res=0;
        if(l<=mid)//第2种情况,左子结点与区间部分覆盖
            res=max(res,query(2*root,l,r));//求区间最大值
            //res+=query(2*root,l,r);//求区间和
        if(r>mid)//第3种情况,右子结点与区间部分覆盖
            res=max(res,query(2*root+1,l,r);//求区间最大值
            //res+=query(2*root+1,l,r);//求区间和
        return res;
    }
}
```

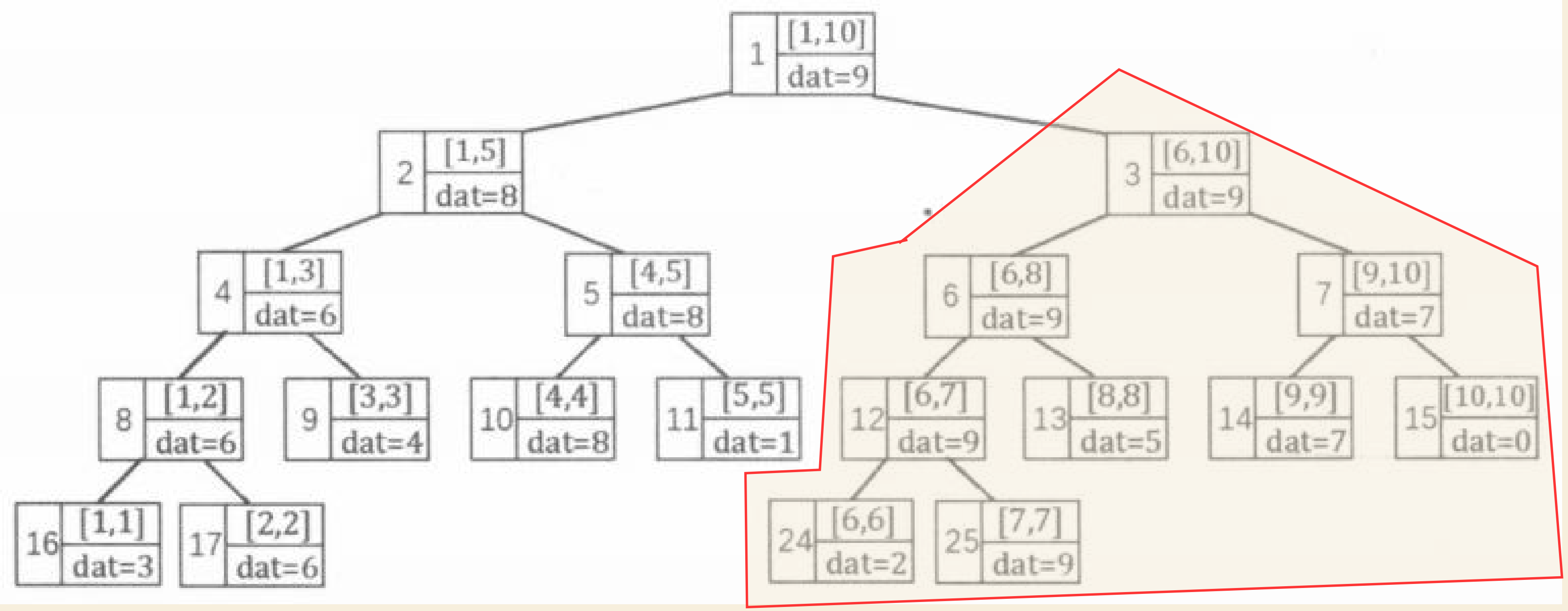
基础线段树

线段树的下沿操作pushdown及区间修改modify

在线段树的“区间查询”指令中，每当遇到被询问区间[l,r]完全覆盖的结点时，可以立即把该结点上存储的信息作为候选答案返回。被询问区间[l,r]在线段树上被分为O(logN)个小区间(结点)，从而在O(logN)的时间内求出答案。

不过，在“区间修改”指令中，如果某个区间被修改区间[l,r]完全覆盖，那么**以该结点为根的整棵子树中的所有结点存储的信息都会发生变化，若逐一进行更新，将使得一次区间修改指令的时间复杂度增加到O(N)**，这是我们不能接受的。

修改区间[6,10]加上5



基础线段树

线段树的下沿操作pushdown及区间修改modify

试想，如果我们在一次修改中发现结点root代表的区间[tree[root].l,tree[root].r]被修改区间[l,r]完全覆盖，并且逐一更新了子树root的所有结点，但是在**之后的查询指令中却根本没有用到[l,r]的子区间作为候选答案**，那么更新整棵子树就是徒劳的。

换言之，在执行修改指令时，同时可以在 $l \le tree[root].l \le tree[root].r \le r$ 的情况下立即返回，只不过在回溯之前向结点root添加一个**懒标记**，标识为“**该结点曾经被修改过，但其子结点尚未更新**”。

如果在后续的指令中，需要从结点root向下递归，我们检查root是否具有懒标记。若有懒标记，就**根据懒标记信息更新root的两个子结点，同时为root的两个子结点增加懒标记，然后清除p的懒标记**。

上一段话实际就是**下沿pushdown的大致操作流程**。

也就是说，除了在修改指令中直接划分成的O(logN)个结点之外，对**任意结点的修改都延迟到“在后续操作中递归进入它的父结点时”**再执行。这样一来，每条查询或修改指令的时间复杂度都降低到了O(logN)。这些懒标记也可称为”延迟标记”。**延迟标记提供了线段树中从上往下传递信息的方式**。这种“延迟”也是设计算法与解决问题的一个重要思路。

基础线段树

线段树的下沿操作pushdown及区间修改modify

如果在后续的指令中，需要从结点root向下递归，我们检查root是否具有懒标记。若有懒标记，就**根据懒标记信息更新root的两个子结点，同时为root的两个子结点增加懒标记，然后清除p的懒标记。**

```
void pushdown(int root){//下沿操作
    if(tree[root].add){//结点root有懒标记
        tree[2*root].add+=tree[root].add;//给左子结点打延迟标记
        tree[2*root+1].add+=tree[root].add;//给右子结点打延迟标记
        tree[2*root].dat+=tree[root].add*(tree[2*root].r-tree[2*root].l+1);
        //更新左子结点区间和
        tree[2*root+1].dat+=tree[root].add*(tree[2*root+1].r-tree[2*root+1].l+1);
        //更新右子结点区间和
        tree[root].add=0;//清除root的懒标记
    }
}
```

基础线段树

线段树的区间修改modify

区间修改的思想和区间查询类似（**4种情况**和3种情况都行）

```
void modify(int root,int l,int r,int val){//区间修改
    if(l<=tree[root].l&&r>=tree[root].r){//第1种情况,完全包含
        tree[root].add+=val;//给该结点区间加上懒标记
        tree[root].dat+=val*(tree[root].r-tree[root].l+1);//更新该结点的信息
    }
    else{
        pushdown(root);//一定在修改之前pushdown
        int mid=(tree[root].l+tree[root].r)/2;
        //注意是该结点区间[tree[root].l,tree[root].r]的mid,不是区间[l,r]的mid
        if(r<=mid)//第2种情况,只递归左子结点
            modify(2*root,l,r,val);
        else if(l>mid)//第3种情况,只递归右子结点
            modify(2*root+1,l,r,val);
        else{//第4种情况,左右子结点都递归
            modify(2*root,l,r,val);
            modify(2*root+1,l,r,val);
        }
        pushup(root);//修改之后一定pushup
    }
}
```

基础线段树

线段树的区间修改modify

区间修改的思想和区间查询类似（4种情况和**3种情况**都行）

```
void modify(int root,int l,int r,int val){//区间修改
    if(l<=tree[root].l&&r>=tree[root].r){//第1种情况,完全包含
        tree[root].add+=val;//给该结点区间加上懒标记
        tree[root].dat+=val*(tree[root].r-tree[root].l+1);//更新该结点的信息
    }
    else{
        pushdown(root);//一定在修改之前pushdown
        int mid=(tree[root].l+tree[root].r)/2;
         //注意是该结点区间[tree[root].l,tree[root].r]的mid,不是区间[l,r]的mid
        if(l<=mid)//第2种情况,左子结点与区间部分覆盖
            modify(2*root,l,r,val);
        if(r>mid)//第3种情况,右子结点与区间部分覆盖
            modify(2*root+1,l,r,val);
        pushup(root);//修改之后一定pushup
    }
}
```

基础线段树

线段树的修改和查询一般以下面三种情况进行组合:

此处以均以加一个数(单点加数,区间加数)为例的代码

- ✓ **单点修改，区间查询　例题　代码**

至少需含有pushup(root),build(root,l,r),modify(root,pos,val),query(root,l,r)四个函数

- ✓ **区间修改，单点查询　例题　代码**

至少需含有pushup(root),pushdown(root),build(root,l,r),modify(root,l,r,val),query(root,pos)五个函数

- ✓ **区间修改，区间查询　例题　代码**

至少需含有pushup(root),pushdown(root),build(root,l,r),modify(root,l,r,val),query(root,l,r)五个函数

讲完后可以用线段树和树状数组做一遍

基础线段树

1、线段树可以做很多很多与区间有关的事情
2、空间复杂度O(N*4)，每次更新和查询操作的复杂度都是O(logN)。

3、在更新和查询区间[l,r]的时候，为了保证**复杂度是严格的O(logN)**，必须在达到被[l,r]覆盖的区间的结点时就立即返回。而为了保证这样做的正确性，需要在这两个过程中做一些相关的“懒”操作。

“懒标记”在更新区间的有关问题上至关重要

4、pushup函数一般写在修改完后，pushdown函数一般写在修改和查询之前
5、线段树的代码一般不太好调试，容易发生**段错误**，代码难写较长

树状数组

树状数组，也称作“二叉索引树”（Binary Indexed Tree）或 Fenwick 树。 它可以高效地实现如下两个操作：1、**数组前缀和的查询**；2、**单点更新**。下面具体解释这两个操作。

- 数组前缀和的查询

已知**数组**sum是原数组a(下标从1开始)的前缀和，故 $sum[i]=sum[i-1]+a[i]$

如果求区间[l,r]和，可在O(1)的时间复杂度内求出数组前缀和,即 $res=sum[r]-sum[l-1]$，但是如果还要执行“单点更新”，就得重新更新前缀和数组sum(计算一次前缀和)，此时单点修改的时间复杂度为O(N)。

- 单点修改

如果我们只使用**原数组**a进行操作，当执行单点修改时，只用修改其对应下标的值即可，此时单点修改的时间复杂度为O(1)，若执行数组前缀和的查询，需要扫描对应区间的每一个值进行求和，此时前缀和的查询的时间复杂度为O(N)

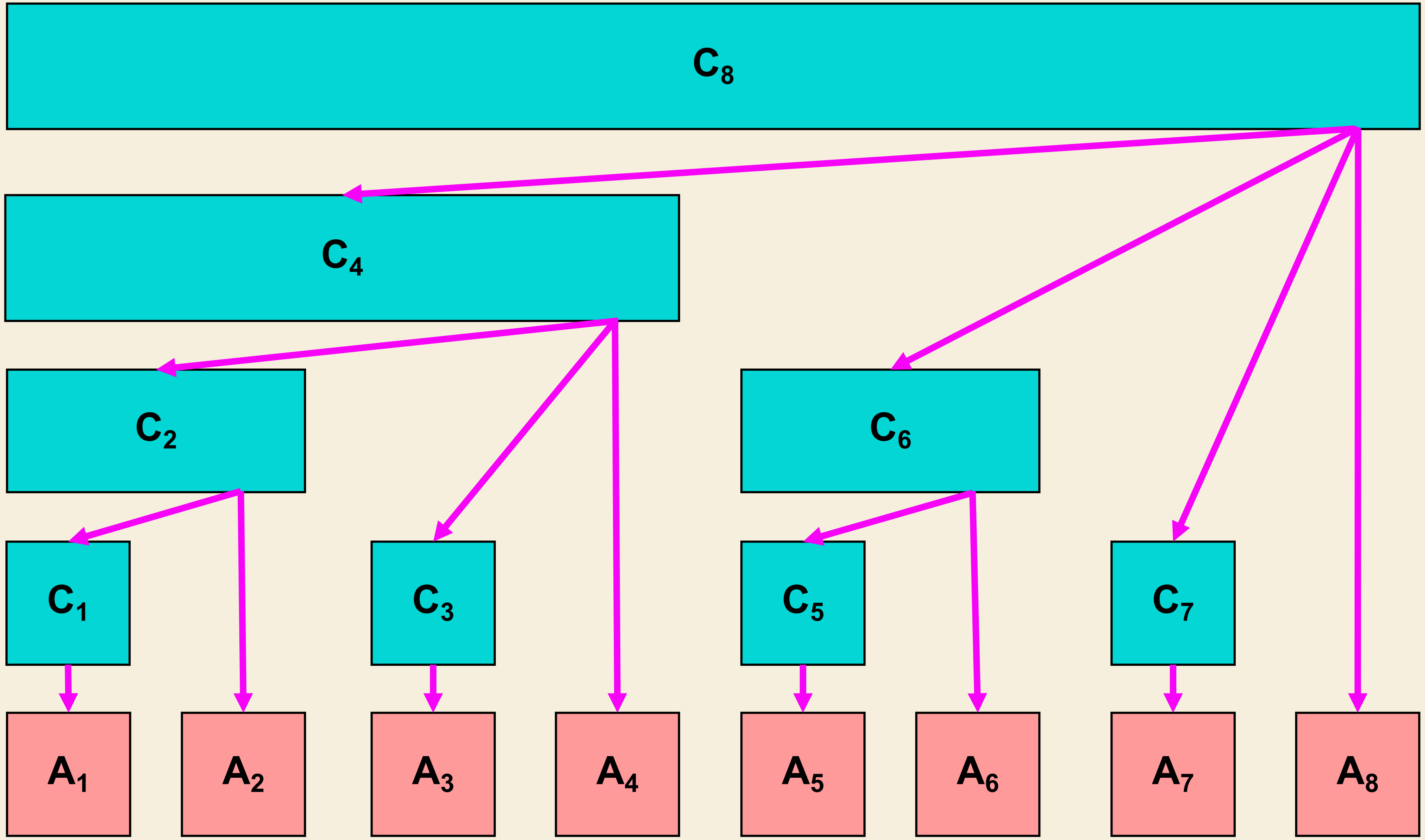
树状数组

那如果我在一次业务场景中“前缀和计算”和“单点更新”的次数都很多，前缀和数组就不高效了。而 Fenwick 树(树状数组)就是“高效的”实现“前缀和”和“单点更新”这两个操作的数据结构，能**在O(logN)的时间复杂度解决两个问题**。

我们首先先看看树状数组长什么样，有一个直观的认识。

树状数组

我们以一个有 8 个元素的数组 A 为例（如上图），在数组 A 之上建立一个数组 C，使得数组 C 的形成如上的一个多叉树形状，数组 C 就是一个树状数组。



如何解释“前缀和查询”、“单点更新”？

例如我们要查询“前缀和(4)”，本来应该问 A_1、A_2、A_3、A_4，有了数组 C 之后，我们只要问 C_4 即可。

再如，我们要更新结点 A_1 的值，只要自底向上更新 C_1、C_2、C_4、C_8 的值即可。

树状数组

理解C数组
树状数组的**下标从 1 开始计数**，这一点我们看到后面就会很清晰了。我们先了解如下的定义，记住这些记号所代表的含义：

- ✓ 数组 C 是一个对原始数组 A 的预处理数组。
- ✓ 为了方便说明，避免后面行文啰嗦，我们将固定使用记号 i 、j ，它们的定义如下：
  - 记号 i ： 表示**预处理数组 C** 的索引（十进制表示）。
  - 记号 j ： 表示**原始数组 A** 的索引（十进制表示）。

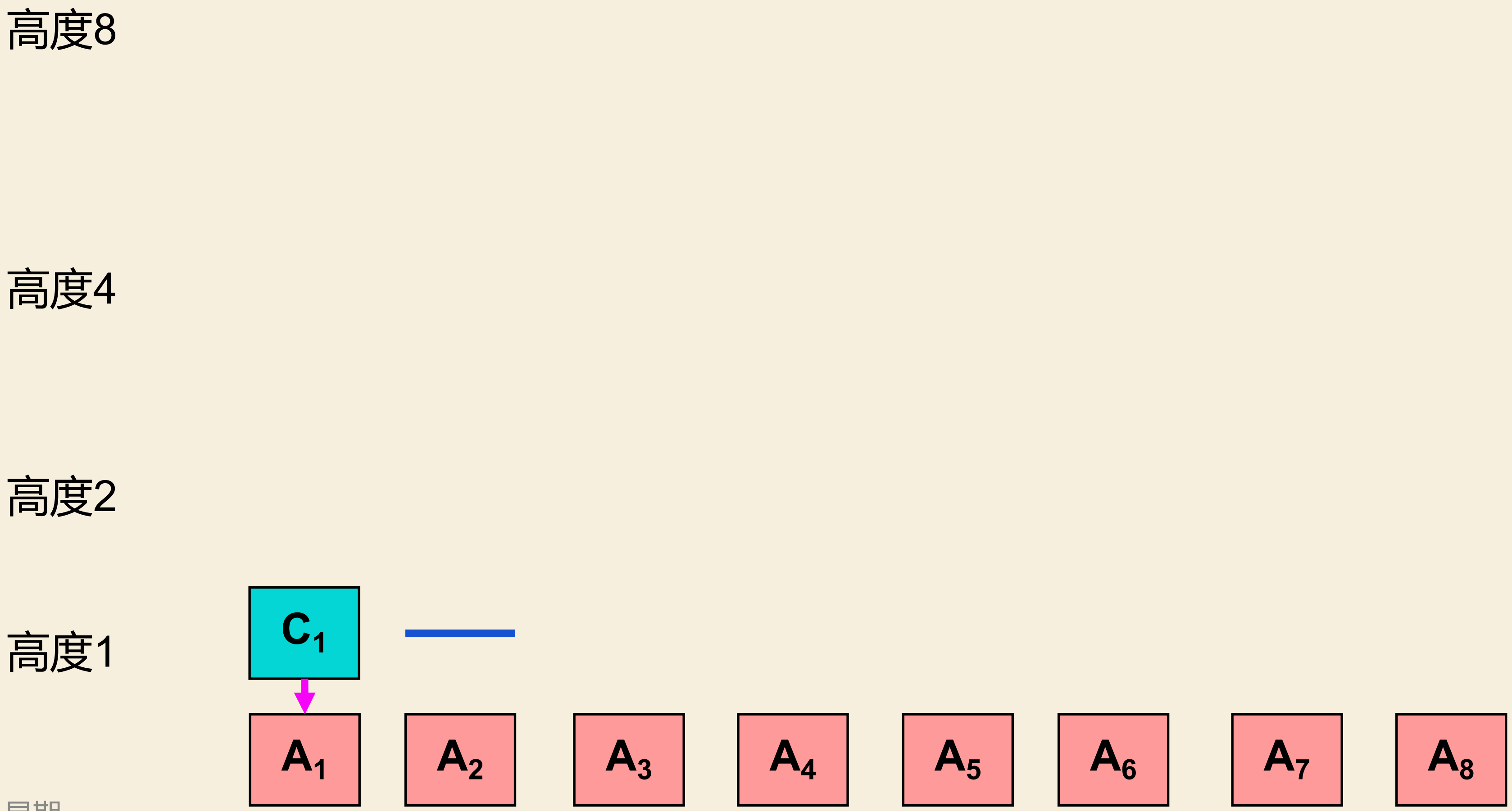
树状数组

那么C_1、C_2、C_3、C_4、C_5、C_6、C_7、C_8 分别是如何定义的?

用容易理解的解释:

前人栽树,后人乘凉，前人搭梯子，后人跳更高，**后人不忘恩**。

C_1放置的梯子，梯子的规则是：**梯子的宽度和高度相当，梯子能够帮助后人登上的高度也和梯子的高度相当**。此时C_1的高度是1，因此C_1的梯子宽度只能让A_2够得着。C_1的梯子能够让别人跳上的高度也和梯子的高度相当，因此爬上这个梯子的人(只有C_2）可以直接到高度2的位置。



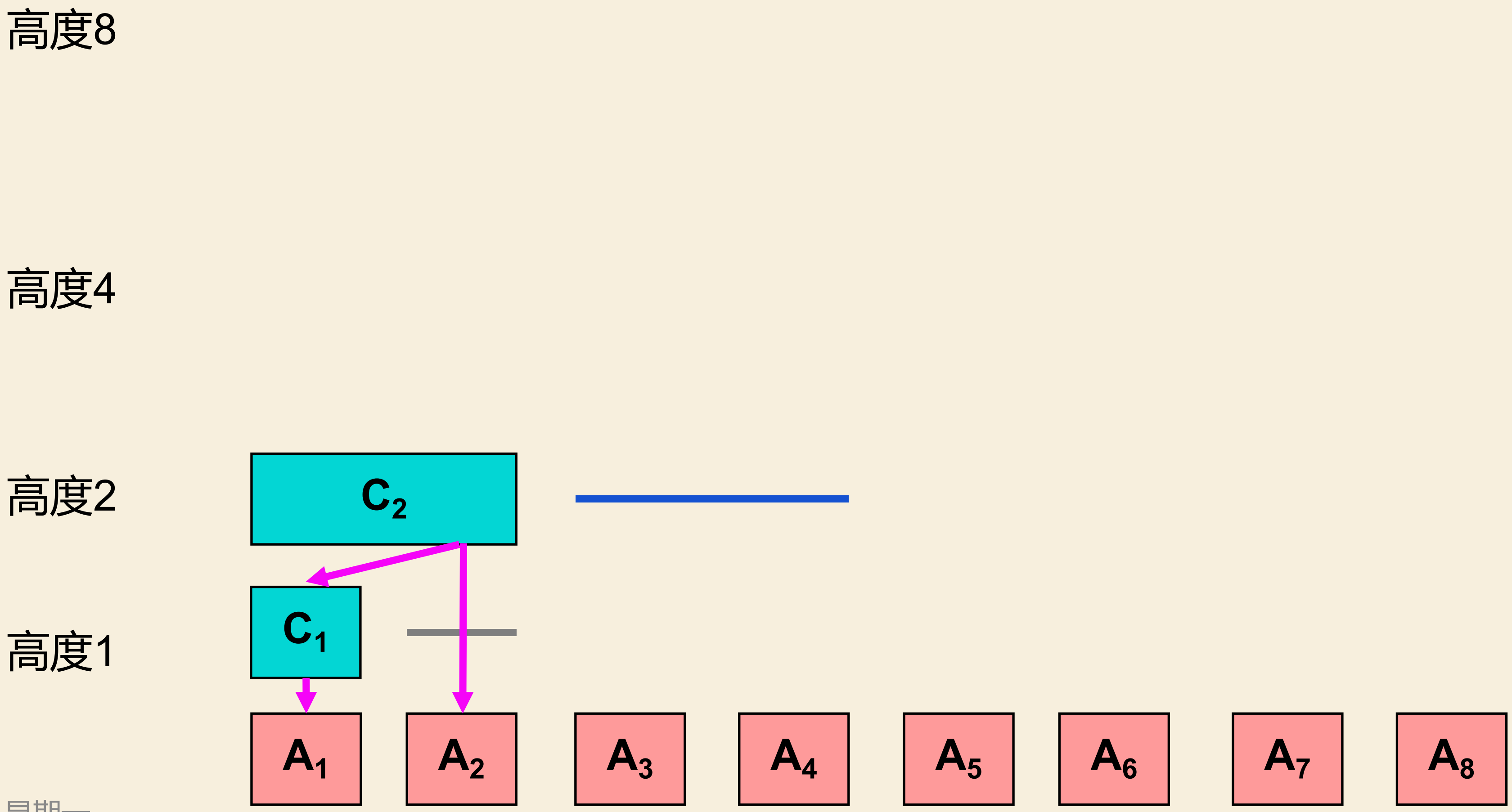
树状数组

那么C_1、C_2、C_3、C_4、C_5、C_6、C_7、C_8 分别是如何定义的?

用容易理解的解释:

接下来C_2来了，**C_2顺着C_1的梯子跳到了高度为2的地方**，在这里放了个梯子，供后面的人使用。此时C_2的高度为2，C_2既然用了C_1的梯子，就得“关照”C_1，因此，它连接了C_1。

然后C_2也要放梯子，它的梯子的宽度只能是2。梯子能够帮助后来人到达的高度等于它当前的高度，因此爬上它的人能直接到高度4。

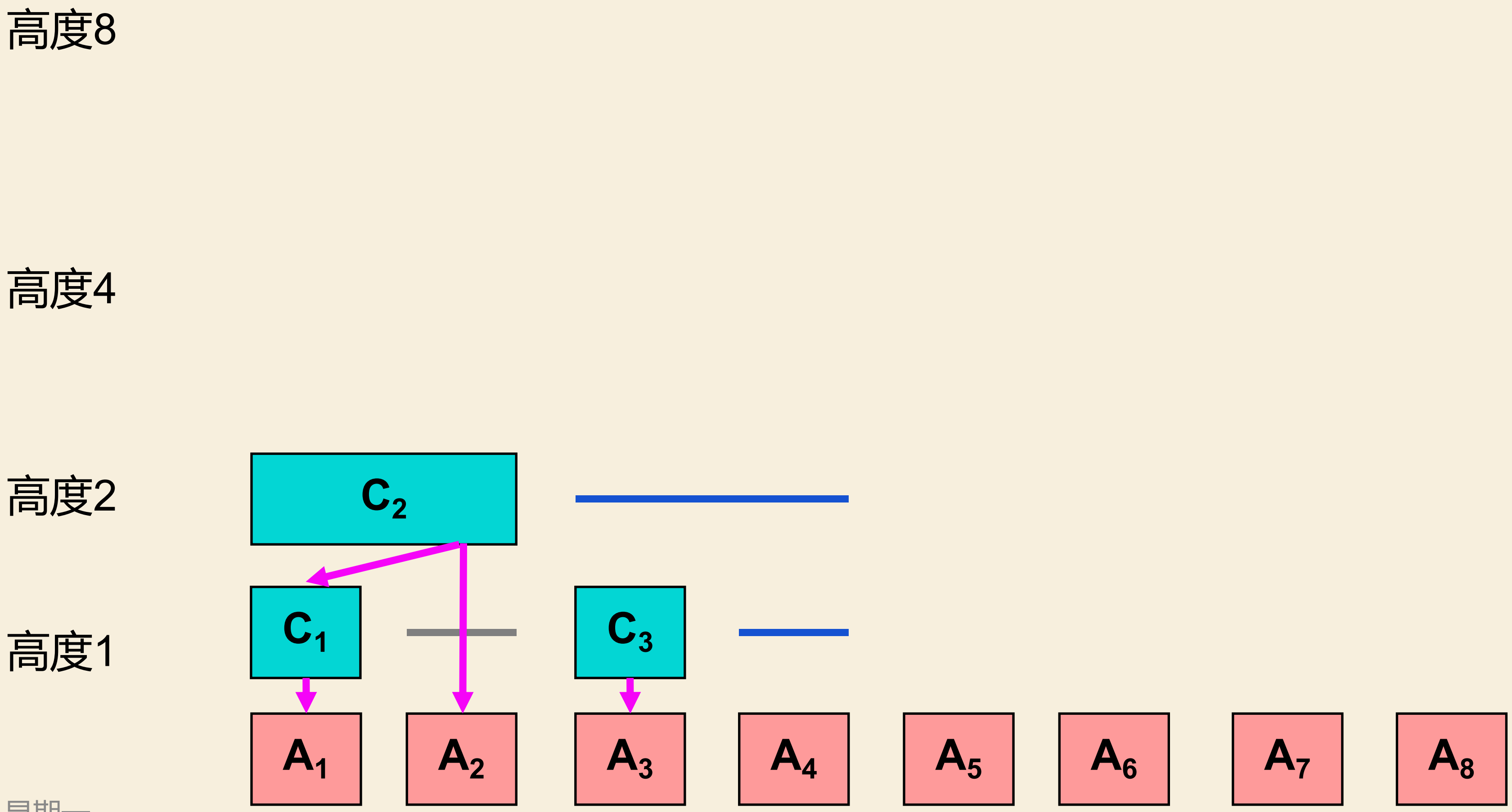


树状数组

用容易理解的解释:

C_3重复C_1的动作，放了一个宽度为1能帮助后来人跳到高度2的梯子。

那么C_1、C_2、C_3、C_4、C_5、C_6、C_7、C_8 分别是如何定义的?



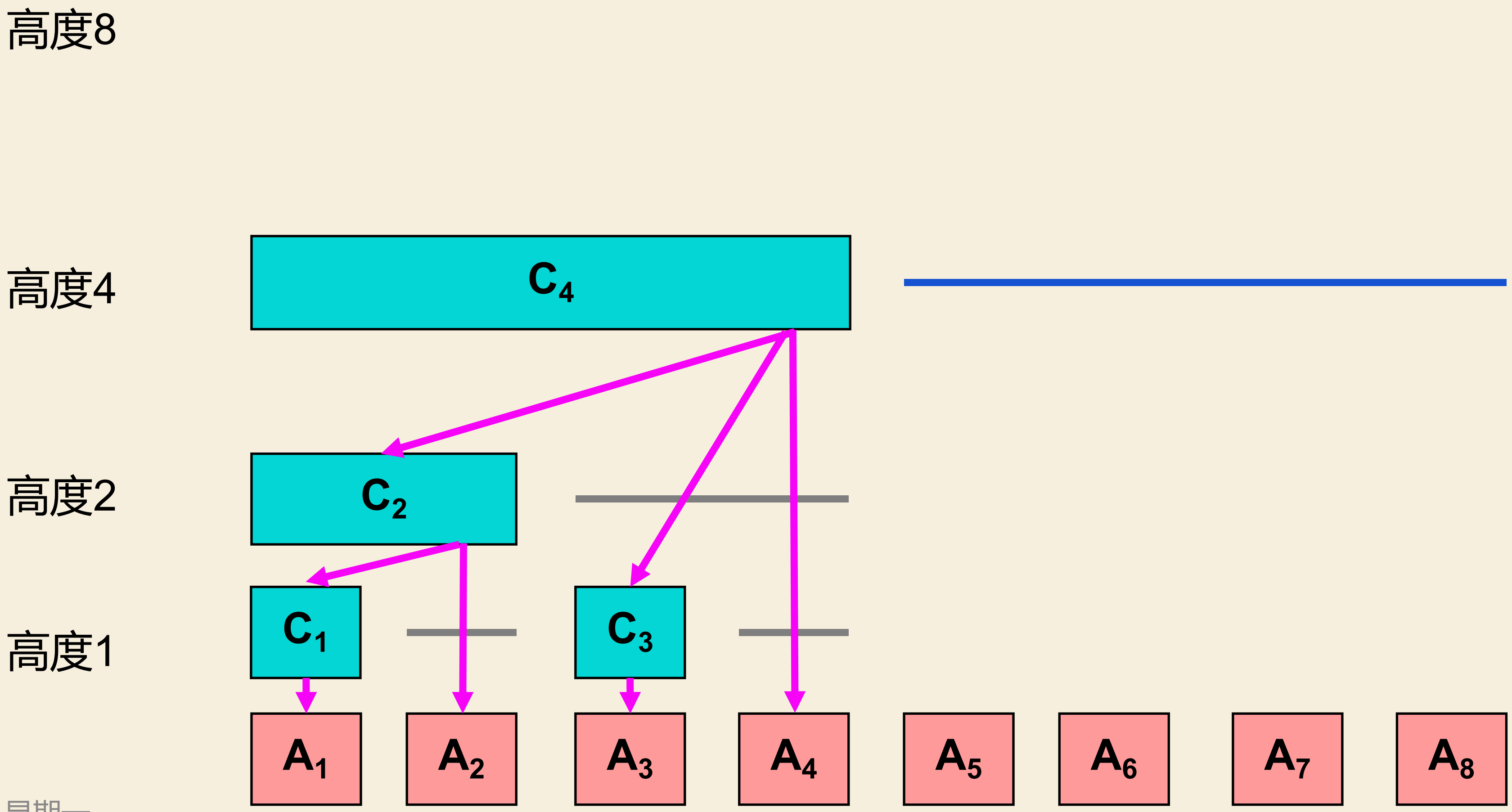
树状数组

那么C_1、C_2、C_3、C_4、C_5、C_6、C_7、C_8 分别是如何定义的?

用容易理解的解释:

C_4来了，它顺着C_3的梯子来到了高度为2的梯子。

在高度为2的地方发现了C_2的梯子，于是跳到了高度为4的地方。C_4使用了C_3、C_2的梯子，所以要“关照”它们，于是连接C_3和C_2。最后**C_4在高度为4**的地方放置了高度为4能帮助后来人跳到8的梯子。

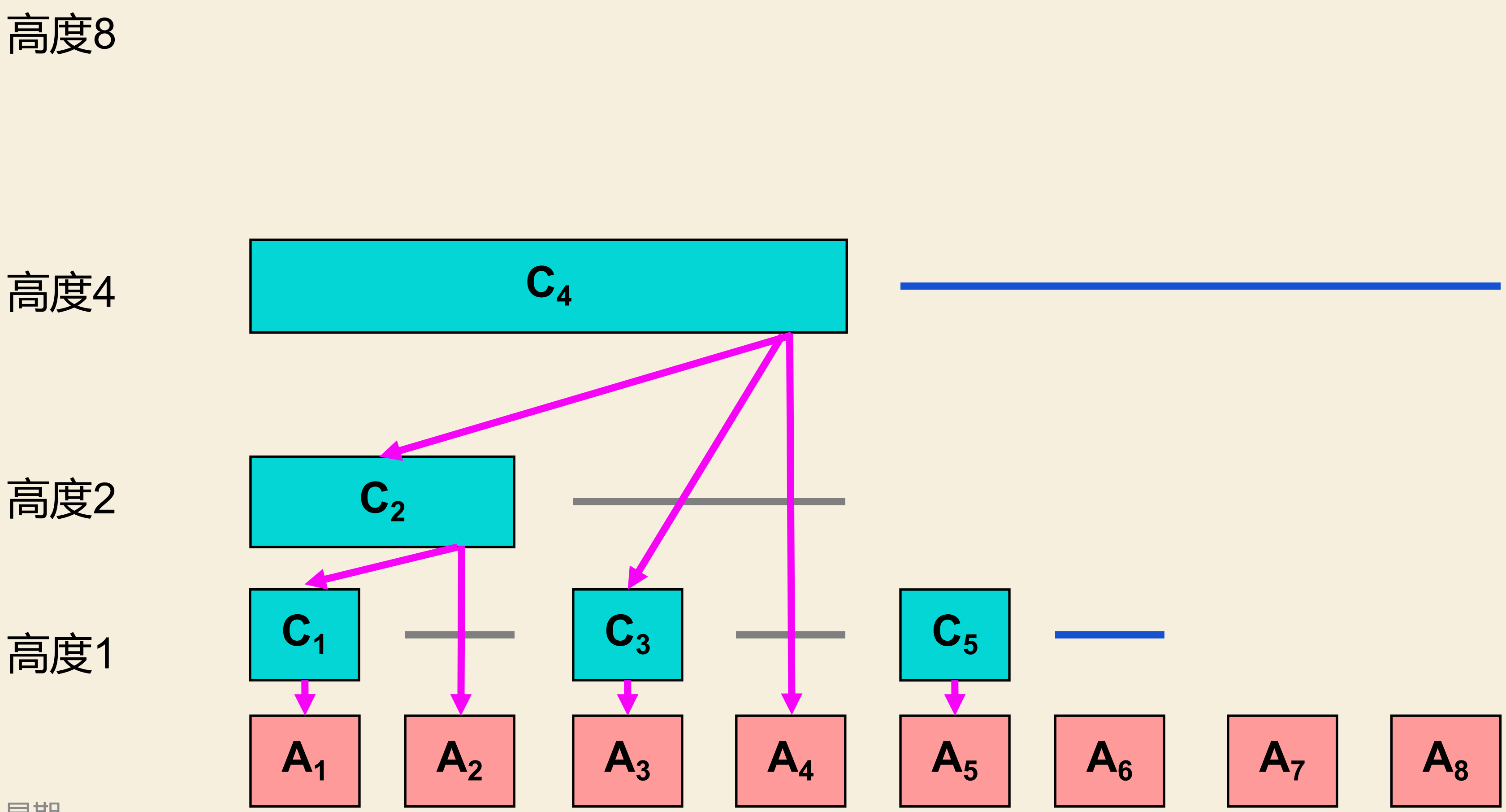


树状数组

那么C_1、C_2、C_3、C_4、C_5、C_6、C_7、C_8 分别是如何定义的?

用容易理解的解释:

C_5重复C_1和C_3的动作，放了一个高度为1能帮助后来人跳到2的梯子。



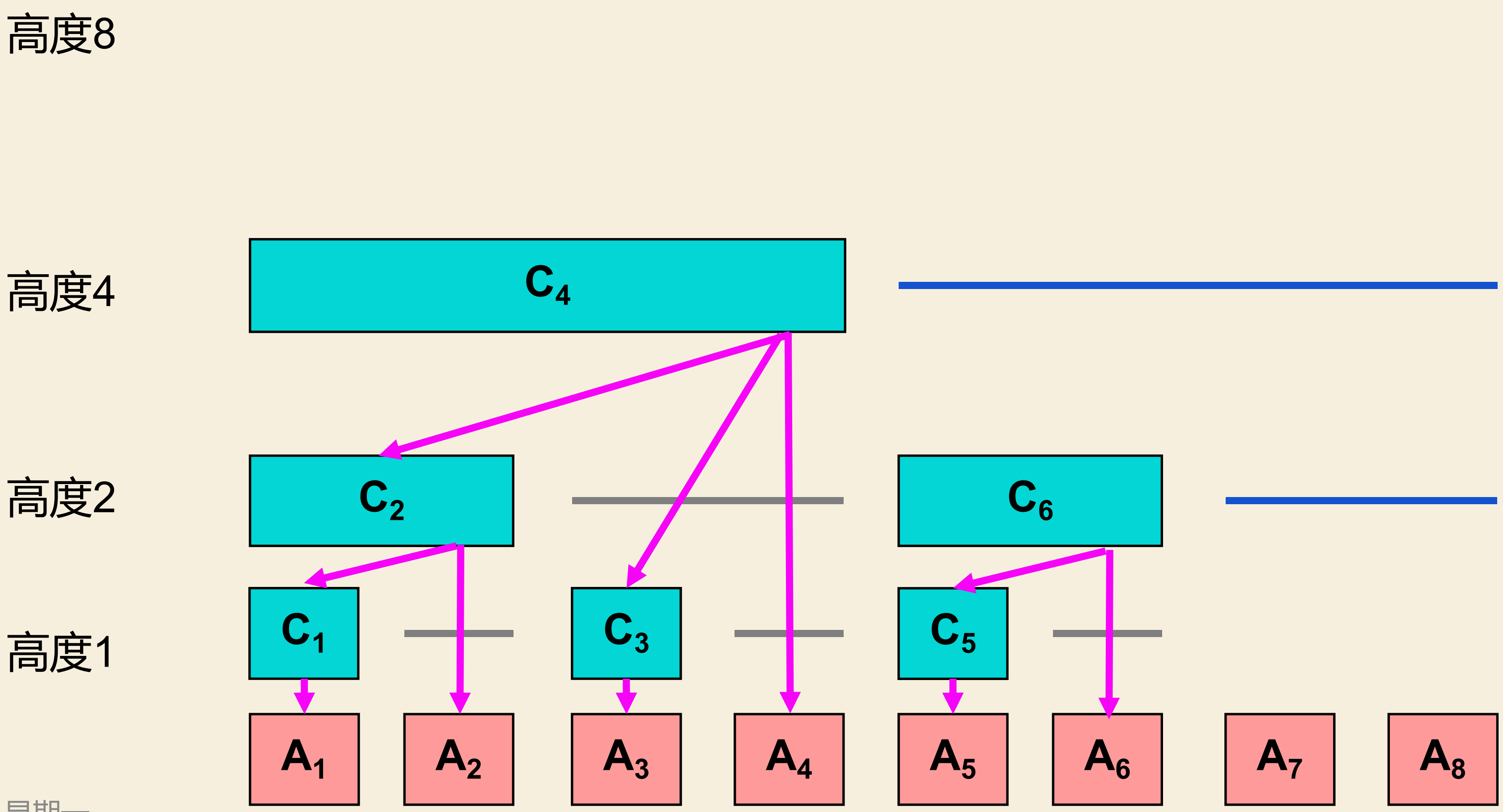
树状数组

那么C_1、C_2、C_3、C_4、C_5、C_6、C_7、C_8 分别是如何定义的?

用容易理解的解释:

　　C_6重复C_2的动作，它登上了C_5的梯子，来到了高度2。因为使用了C_5的梯子，因此要和C_5产生连接。

　　与此同时C_6放置了宽度为2，能帮助后来人跳到高度4的梯子。



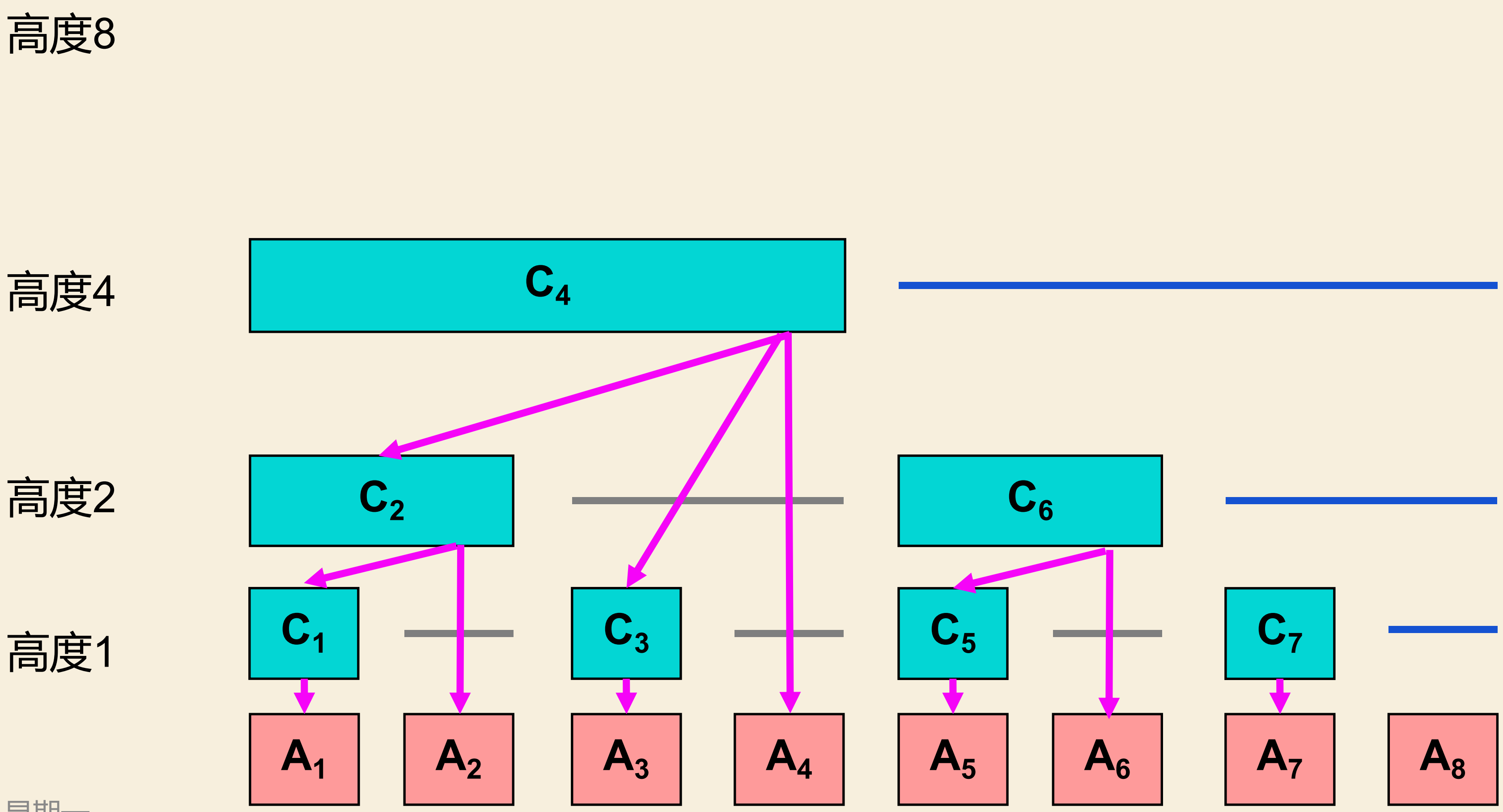
树状数组

用容易理解的解释:

C_7重复C_1，C_3和C_5的动作，

放了一个高度为1能帮助后来人跳到2的梯子。

那么C_1、C_2、C_3、C_4、C_5、C_6、C_7、C_8 分别是如何定义的?



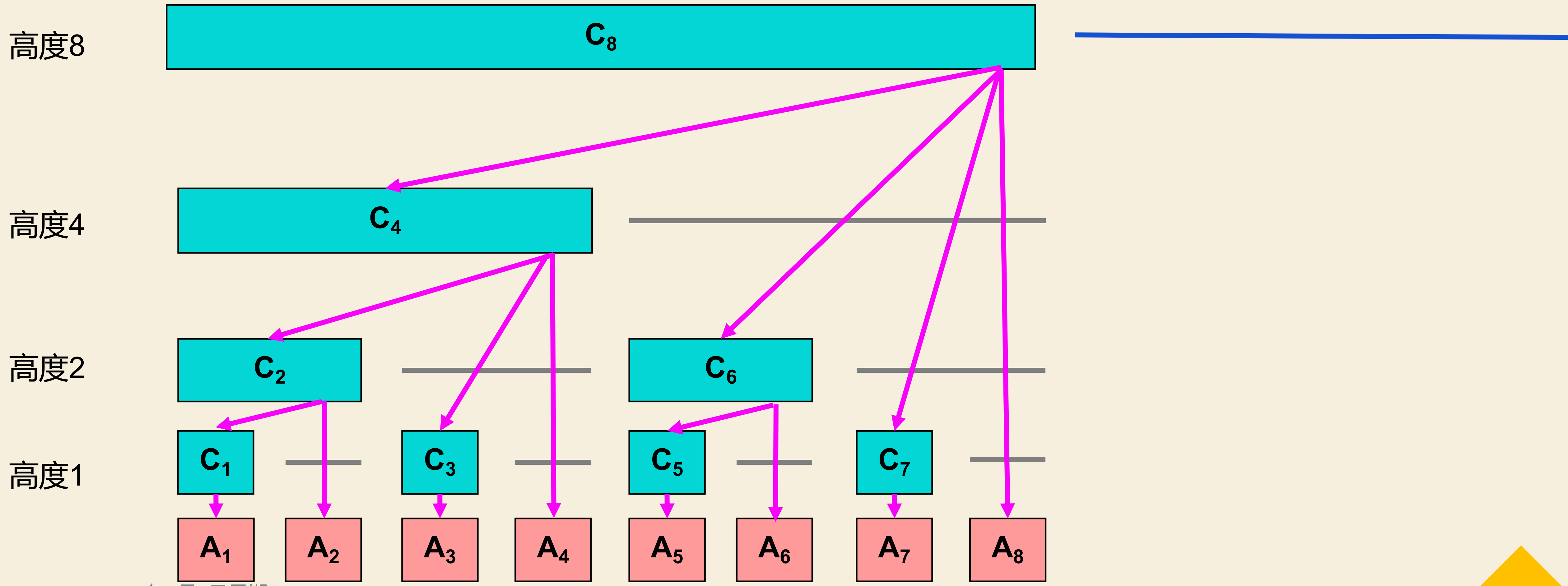
树状数组

那么C_1、C_2、C_3、C_4、C_5、C_6、C_7、C_8 分别是如何定义的?

用容易理解的解释:

C_8登上了C_7放的梯子,来到了高度2，然后登上了C_6的梯子，来到了高度4，然后登上了C_4的梯子，来到了高度8,C_8使用了C_7、C_6、C_4的梯子，因此就要“关照”它们，C_8与C_7、C_6、C_4产生连接。

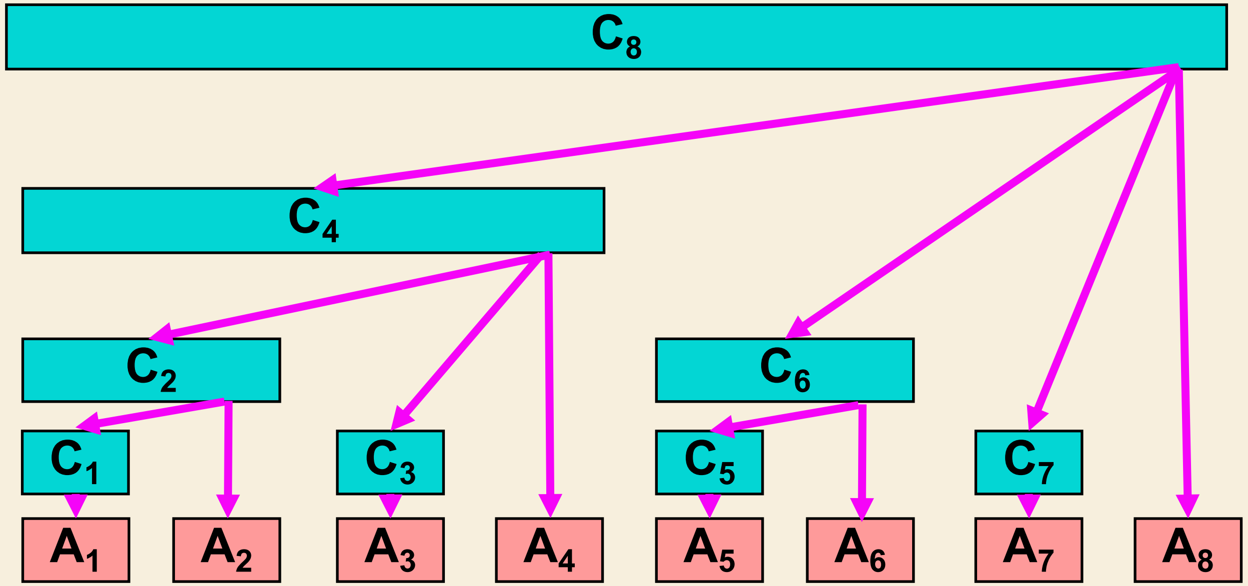
最后,C_8放置了宽度为8能够帮助后来人跳到16的梯子。



树状数组

用右表来表示。

数组C的索引i	数组C的和定义由数组A的哪些元素而来	数组C中的元素来自数组A的个数
1	$C_1=A_1$	1
2	$C2=A_1+A_2$	2
3	$C3=A_3$	1
4	$C4=A_1+A_2+A_3+A_4$	4
5	$C5=A_5$	1
6	$C6=A_5+A_6$	2
7	$C7=A_7$	1
8	$C8=A_1+A_2+A_3+A_4+A_5+A_6+A_7+A_8$	8



树状数组

数组C的索引i	数组C的和定义由数组A的哪些元素而来	数组C中的元素来自数组A的个数
1	$C_1=A_1$	1
2	$C2=A_1+A_2$	2
3	$C3=A_3$	1
4	$C4=A_1+A_2+A_3+A_4$	4
5	$C5=A_5$	1
6	$C6=A_5+A_6$	2
7	$C7=A_7$	1
8	$C8=A_1+A_2+A_3+A_4+A_5+A_6+A_7+A_8$	8

数组 C的索引与数组 A 的索引的关系

伟大的计算机科学家注意到上表中标注了“数组 C 中的元素来自数组 A 的元素个数”，它们的规律如下：将数组 C 的索引 i 表示成二进制，从右向左数，遇到 1 则停止，数出 0 的个数记为 k，则**计算 2^k就是“数组 C 中的元素来自数组 A 的个数”**，并且可以具体得到来自数组 A 的表示，即**从当前索引 i 开始，从右向前数出 2^k 个数组 A 中的元素的和，即组成了 C[i]**。下面具体说明。

树状数组

记号 k：将 i 的二进制表示**从右向左数出的 0的个数，遇到 1 则停止，记为 k**。 我们只对数组 C 的索引 i 进行这个计算，数组 A 的索引 j 不进行相应的计算。理解 k 是如何得到的是关键。下面我们通过两个例子进行说明。

当 i=5 时，计算 k
因为 5 的二进制表示是 0000 0101，从右边向左边数，第 1 个是 1 ，因此 0 的个数是 0，此时 k=0
当 i=8 时，计算 k
因为 8 的二进制表示是 0000 1000，从右边向左边数遇到 1 之前，遇到了 3 个 0，此时 k = 3

计算出 k 以后，2^k 立马得到，故得到下述表格:

树状数组

索引i	i的二进制表示	k	2^k	数组C的和定义由数组A的哪些元素而来
1	0000 0001	0	**1**	$C_1=A_1$
2	0000 001**0**	1	**2**	$C2=A_1+A_2$
3	0000 0011	0	**1**	$C3=A_3$
4	0000 01**00**	2	**4**	$C4=A_1+A_2+A_3+A_4$
5	0000 0101	0	**1**	$C5=A_5$
6	0000 011**0**	1	**2**	$C6=A_5+A_6$
7	0000 0111	0	**1**	$C7=A_7$
8	0000 1**000**	3	**8**	$C8=A_1+A_2+A_3+A_4+A_5+A_6+A_7+A_8$

我们看到 2^k 是我们最终想要的。下面我们介绍一种很酷的操作，叫做 **lowbit** ， 它可以高效地计算 2^k，即我们要证明：

$$lowbit(i)=2^k$$

其中 k 是将 i 表示成二进制以后，从右向左数，遇到 1 则停止时，数出的 0 的个数。

树状数组

通过 lowbit 高效计算 2^k

$$lowbit(i) = 2^k$$

```
int lowbit(int x){
    return x & -x;
}
```

举例:以6$(0000\ 0110)_2$为例计算lowbit(6)
-7的补码: 1111 0001
而~6的补码: 1111 0001,即-7=~6
~6+1=-6,其补码为: 1111 0010
6&-6
6 0000 0110
&-6 1111 0010
结果为: 0000 0010 lowbit(6)=2

大致证明过程:

设$x>0$,x的第k位是1,第$0\sim k-1$都是0

为了实现$lowbit$运算,先把x取反,此时第k位变成0,第$0\sim k-1$变为1,

再令$x=x+1$,此时因为进位,第k位变为1,第$0\sim k-1$变为0

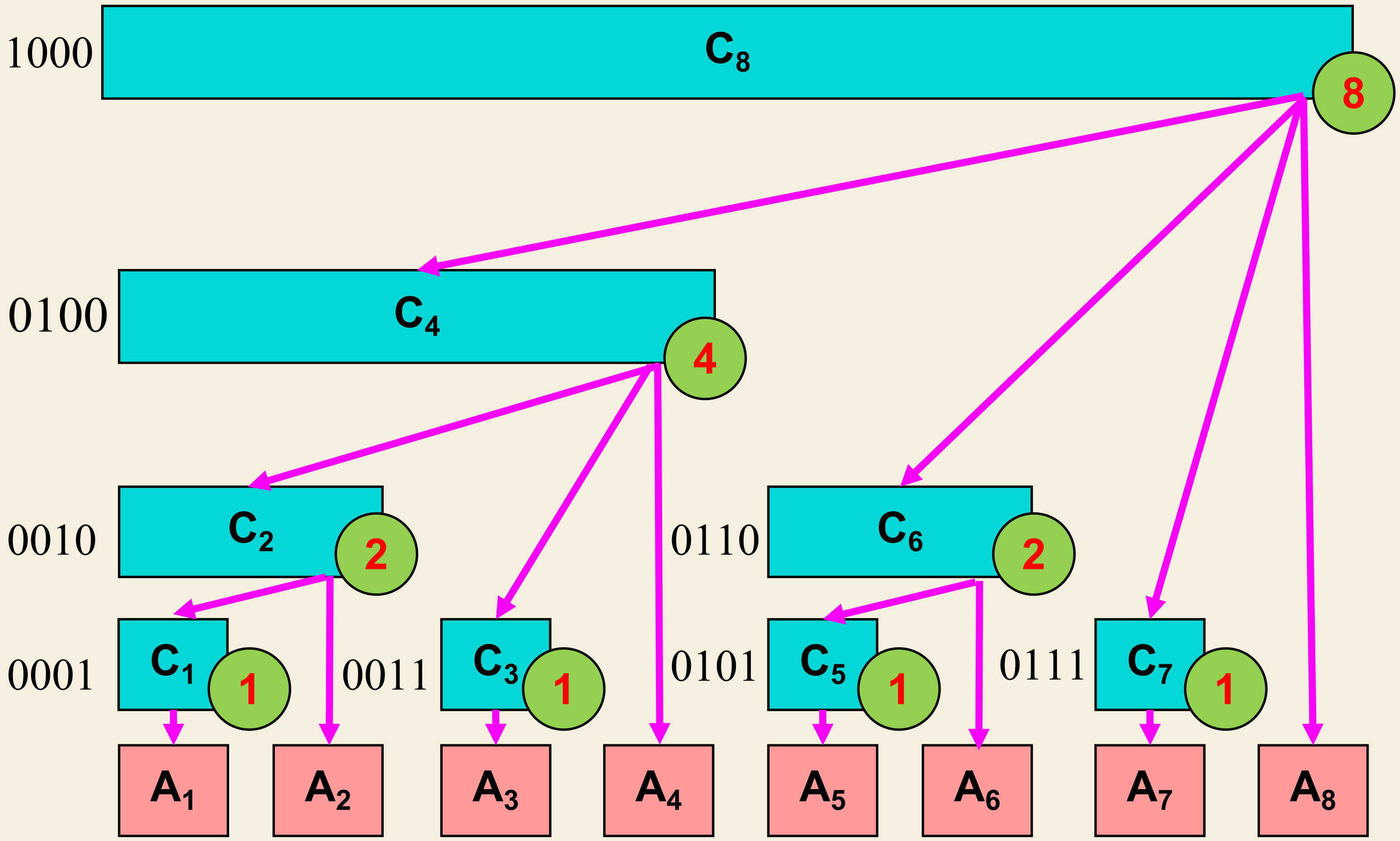
在上面的取反加1操作后,x的第$k+1$位到最高位恰好刚好与原来相反,所以

$x\,\&\,(\sim x+1)$仅有第k位为1,其余都是0,而在补码表示下,$\sim x=-x-1$,因此

$$lowbit(x) = x\,\&\,(\sim x+1) = x\,\&\,(-x-1+1) = x\,\&\,(-x)$$

树状数组

"单点更新"操作："从子结点到父结点"



修改 A[3]， 分析对数组 C 产生的变化。

从图中我们可以看出 A[3]的父结点以及祖先结点依次是 **C[3]、C[4]、C[8]**，所以修改了 A[3] 以后 C[3]、C[4]、C[8] 的值也要修改。

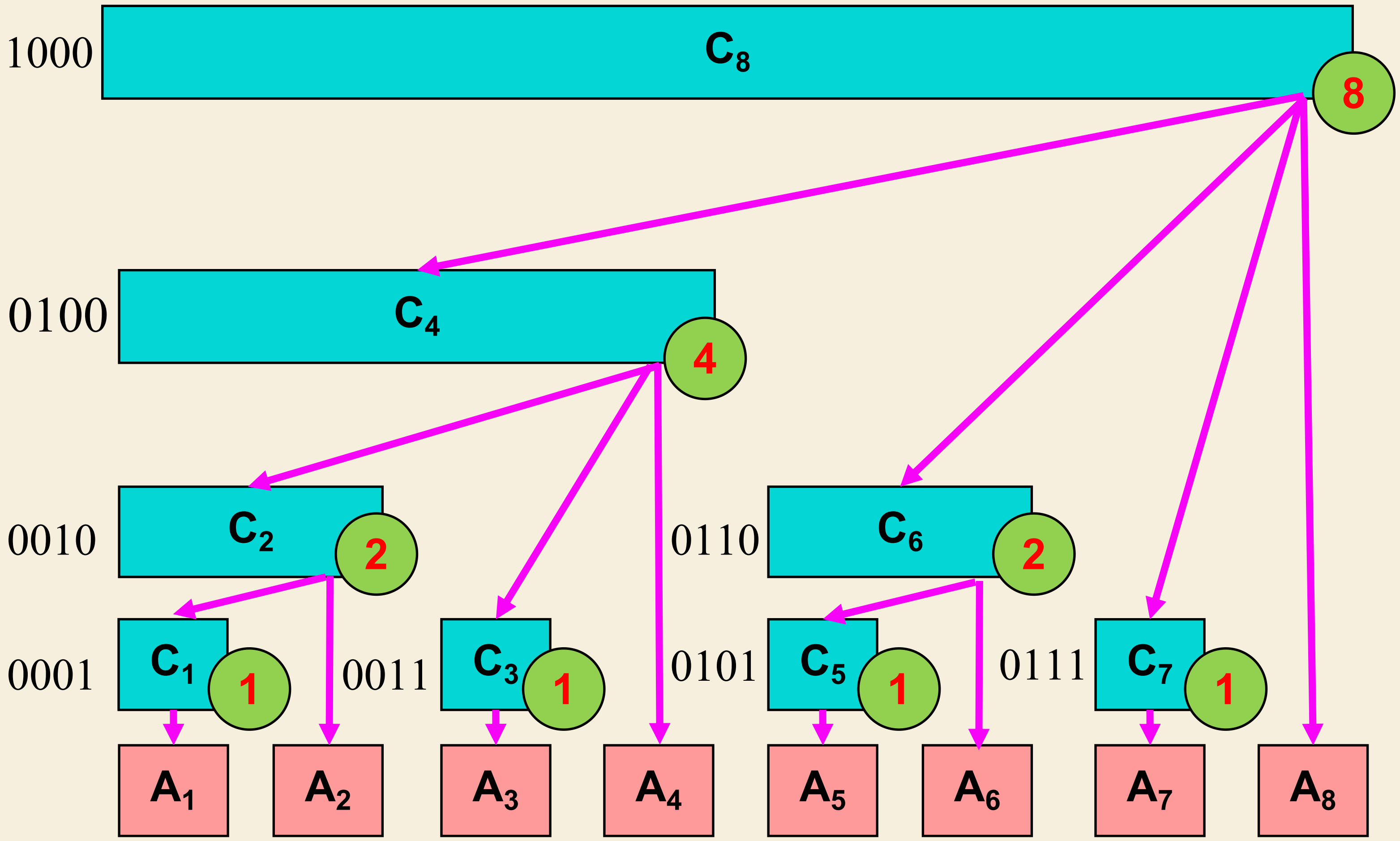
先看 C[3]，lowbit(3)=1， 3+lowbit(3)=4 就是 C[3]的父亲结点 C[4] 的索引值。

再看 C[4]，lowbit(4)=4， 4+lowbit(4)=8 就是 C[4] 的父亲结点 C[8] 的索引值。

从图中，也可以验证： **"蓝色结点的索引值 + 右下角绿色圆形结点的值 = 蓝色结点的双亲结点的索引值"**。

树状数组

“单点更新”操作：“从子结点到父结点”



修改 A[3]， 分析对数组 C 产生的变化。

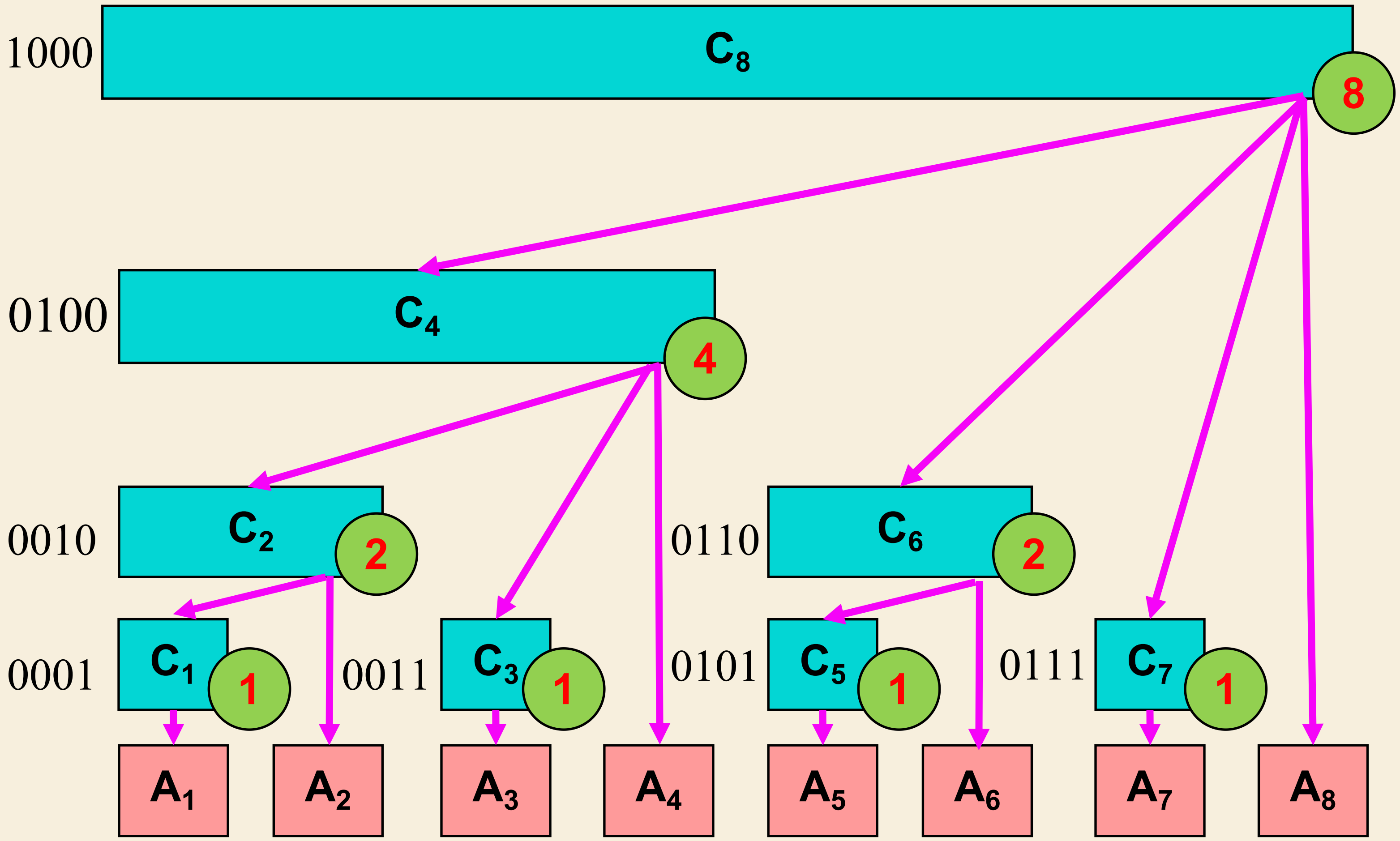
3即 0011，从右向左，遇到 0 放过，遇到 1 为止，给这个数位加1，这个操作就相当于加上了一个 2^k 的二进制数，即一个lowbit 值，有意思的事情就发生在此时，马上就发发生了进位，得到 0100，即 4 的二进制表示。

接下来处理 0100，从右向左，从右向左，遇到 0 放过，遇到 1 为止，给这个数位加 1，同样地，这个操作就相当于加上了一个 2^k 的二进制数，即一个 lowbit值，可以看到，马上就发发生了进位，得到 1000，即 8 的二进制表示。

树状数组

“单点更新”操作：“从子结点到父结点”

修改 A[3]，分析对数组 C 产生的变化。

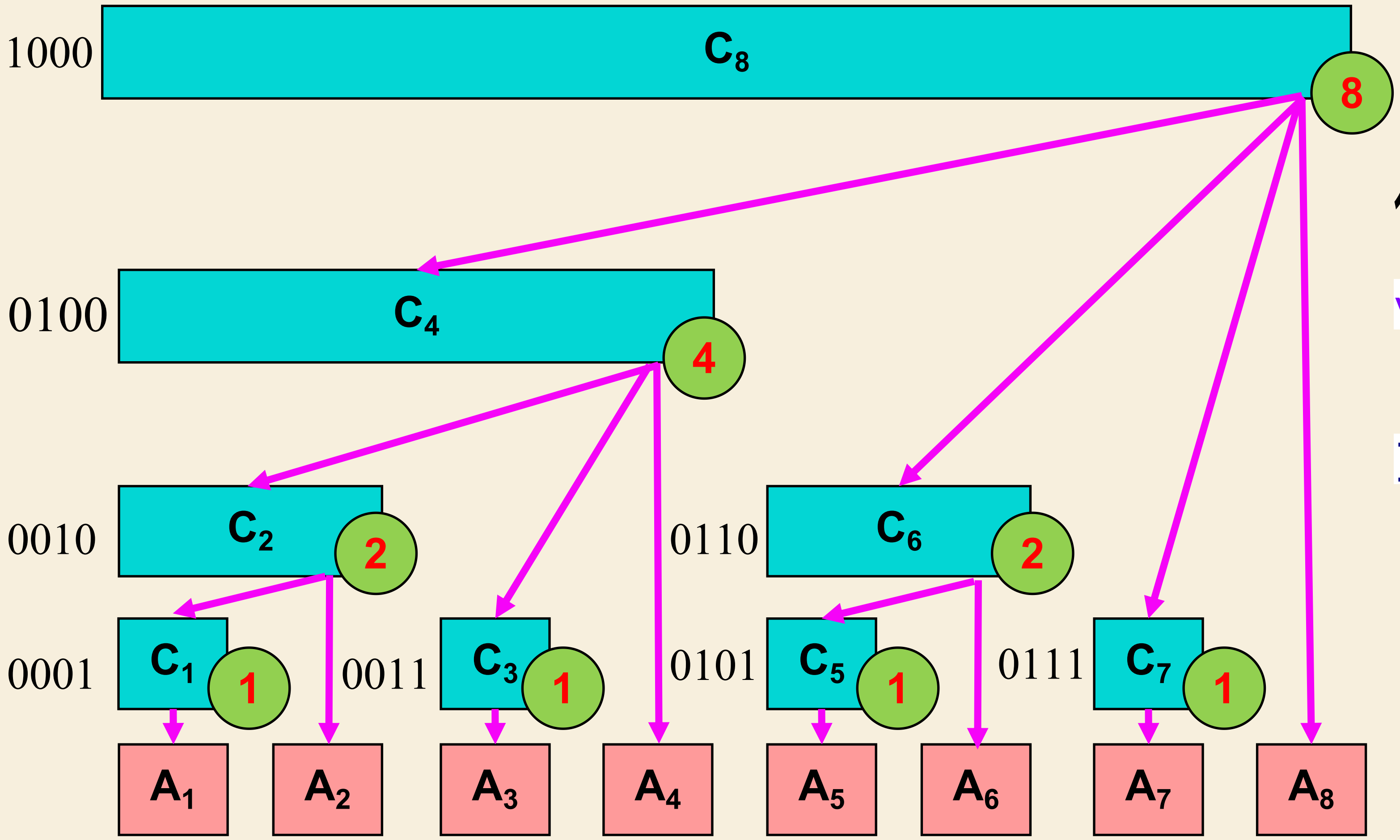


从上面的描述中，可以发现，又在做“从右边到左边数，遇到 1 之前数出 0 的个数”这件事情了，由此我们可以总结出规律：从已知子结点的索引 i，则结点 i 的父结点的索引 parent的计算公式为：

$$parent(i) = i + lowbit(i)$$

树状数组

“单点更新”操作：“从子结点到父结点”

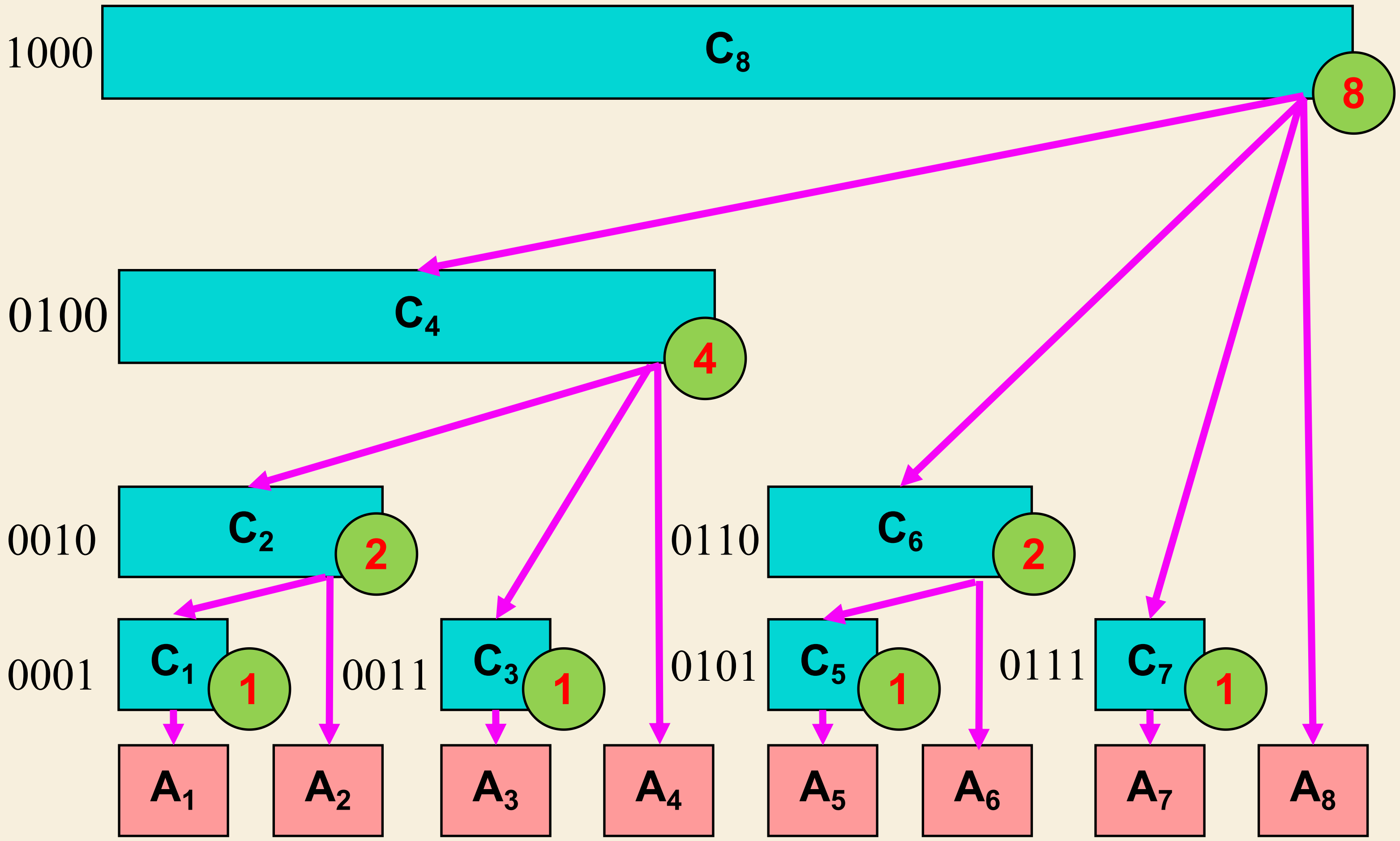


代码实现：

```
void add(int x,int val){
    for(int i=x;i<=n;i+=lowbit(i))
        C[i]+=val;
}
```

树状数组

“前缀和查询操作”：计算前缀和由预处理数组的那些元素表示”



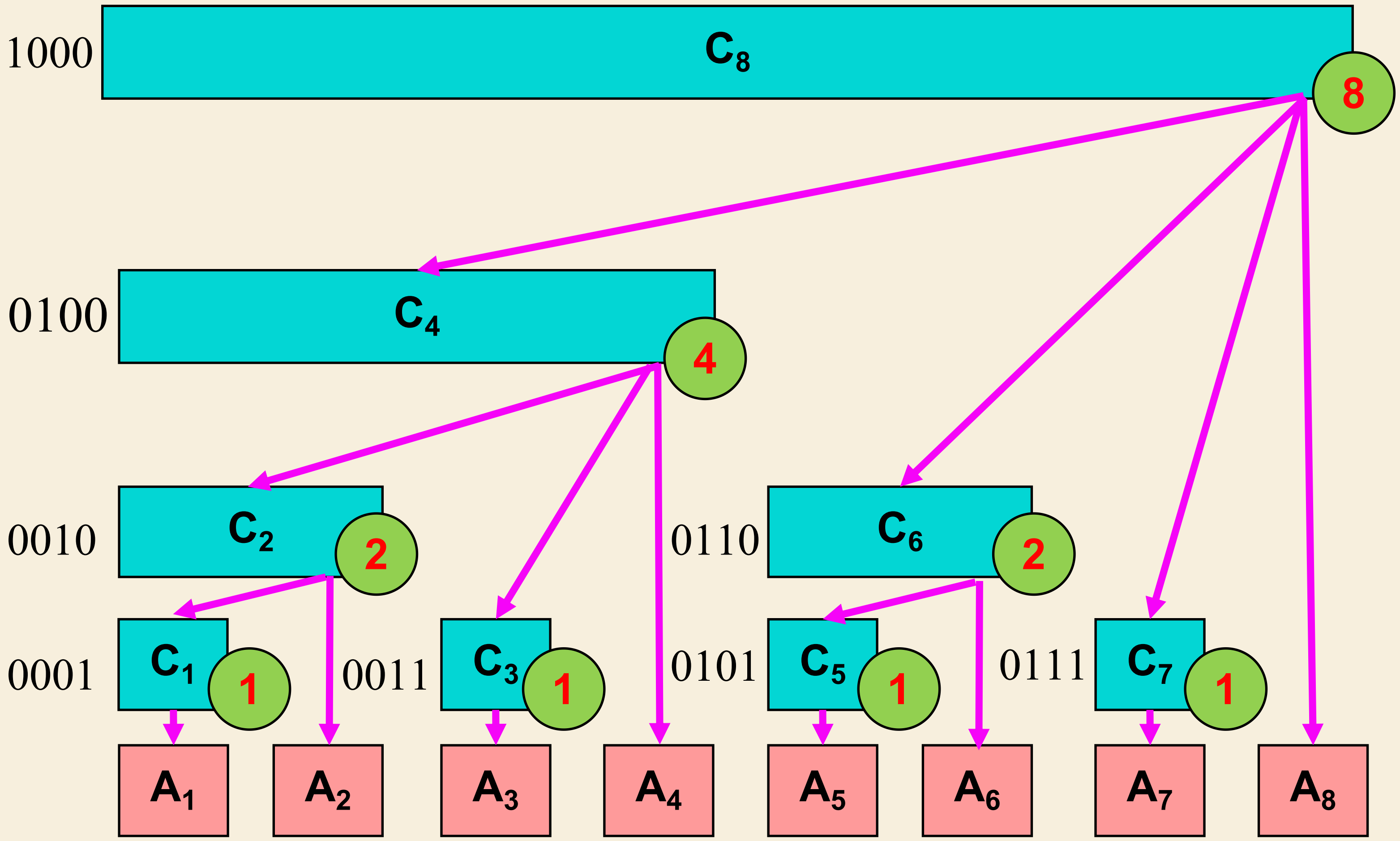
求出“前缀和(6)”。

由图可以看出“前缀和(6) ” = C[6]+ C[4]。先看 C[6]，lowbit(6)=2， 6−lowbit(6)=4 正好是 C[6] 的上一个非叶子结点 C[4] 的索引值。

求出“前缀和(5)”。
再看 C[5]， lowbit(5)=1， 5−lowbit(5)=4 正好是 C[5]的上一个非叶子结点 C[4] 的索引值，故“前缀和(5)” = C[5] + C[4]。

树状数组

“前缀和查询操作”：计算前缀和由预处理数组的那些元素表示”



求出 “前缀和(7)”

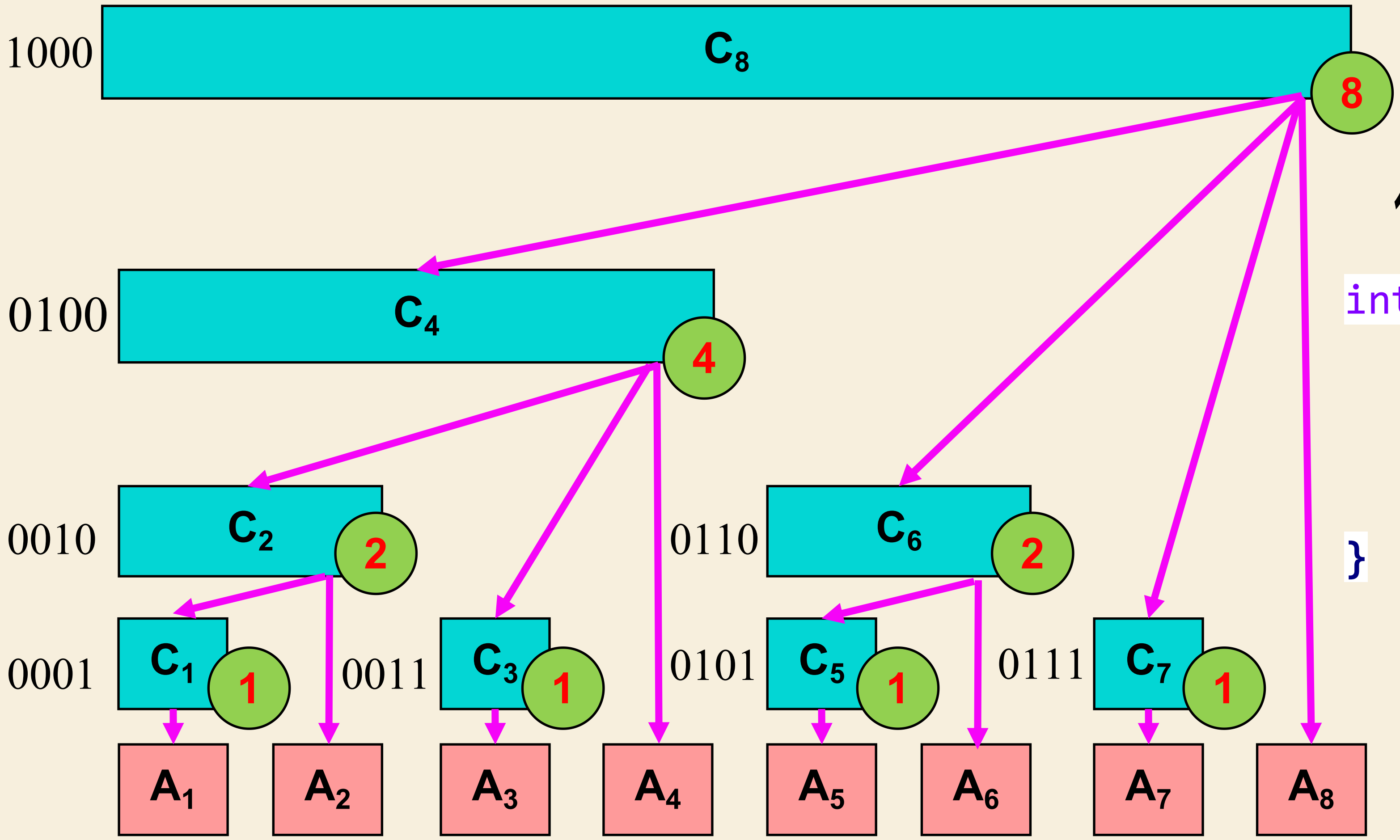
再看 C[7]，lowbit(7)=1， 7−lowbit(7)=6 正好是 C[7] 的上一个非叶子结点 C[6] 的索引值，再由前缀和(6) 的分析， “前缀和(7)” =C[7] + C[6] + C[4]。

求出 “前缀和(8)”

再看 C[8]，lowbit(8)=8， 8−lowbit(8)=0，0表示没有，故 “前缀和(8)” = C[8]

树状数组

“前缀和查询操作”：计算前缀和由预处理数组的那些元素表示”



代码实现:

```
int sum(int x){
    int res=0;
    for(int i=x;i>=1;i-=lowbit(i))
        res+=C[i];
    return res;
}
```

树状数组

树状数组的三个核心函数:

```
int lowbit(int x){//计算最低位的1及其后面所有的0表示的数值
        return x & -x;
}

void add(int x,int val){//单点修改
        for(int i=x;i<=n;i+=lowbit(i))
                C[i]+=val;
}

int sum(int x){//查询前缀和[1,x]
        int res=0;
        for(int i=x;i>=1;i-=lowbit(i))
                res+=C[i];
        return res;
}
```

注：树状数组一般用于解决求和(异或)问题等，不适用于求**最值问题(因为最值在区间不存在前缀和的关系)**

树状数组

故求任意区间[l,r]的和可通过树状数组的sum函数得到: $res = sum(r) - sum(l-1)$

[树状数组单点修改和区间查询完整代码链接]

故树状数组实现了**单点修改和区间查询**的操作

对于**C数组的初始化问题**:

一般用**第一种方法(时间复杂度为O(nlogn))就行**，除非极其卡树状数组的时间就采用第二种方法(时间复杂度为O(n))

✓ 方法一: 一开始C[i]所有元素都是0，然后从1-n调用add(i,a[i])函数即可实现对C数组的初始化。

```
void init(){
    for(int i=1;i<=n;i++)
        add(i,a[i]);
}
```

✓ 方法二: 使用**C[i]维护的区间范围是[i-lowbit(i)+1,i]**，借助前缀和pre解决初始化问题**pre[i]-pre[i-lowbit(i)]**

```
void init(){
    for(int i=1;i<=n;i++){
        pre[i]=pre[i-1]+a[i];//前缀和pre数组
        C[i]=pre[i]-pre[i-lowbit(i)];
    }
}
```

树状数组

扩展:

既然树状数组实现了**单点修改和区间查询**的操作，那么可以像线段树那样实现**区间修改和单点查询**，**区间修改和区间查询**的操作呢?

当然是可以的，但是维护的信息不同（需用到**差分和前缀和**）

树状数组

扩展:

区间修改和单点查询

给定数列 $a[1], a[2], \ldots, a[n]$，你需要依次进行 q 个操作，操作有两类：

- `1 l r x`：给定 l, r, x，对于所有 $i \in [l, r]$，将 $a[i]$ 加上 x（换言之，将 $a[l], a[l+1], \ldots, a[r]$ 分别加上 x）；
- `2 i`：给定 i，求 $a[i]$ 的值。

由于树状数组只支持“单点修改”和“区间查询”，不支持“区间修改”，故需要将问题转换一下

设数组b为原数组a的差分数组，则 $$b[i] = \begin{cases} a_i - a_{i-1}, i \in [2, n] \\ a_1, i = 1 \end{cases}$$

差分序列的前缀和等于原数组的a[i] $$a_i = \sum_{j=1}^{i} b_j$$

根据差分序列的性质，我们**用树状数组维护差分序列**

树状数组区间修改和单点查询完整代码链接

- 初始化树状数组，C[i]=a[i]−a[i−1]
  add(i,a_i−a_{i-1})
- 区间 [l,r]+x（区间内所有的数都加 x）
  add(l,+x),add(r+1,-x)
- 询问修改后下标为 i 的值，执行
  sum(i)

树状数组

扩展:

给定数列 $a[1], a[2], \dots, a[n]$，你需要依次进行 q 个操作，操作有两类：

- `1 l r x`：给定 l, r, x，对于所有 $i \in [l, r]$，将 $a[i]$ 加上 x（换言之，将 $a[l], a[l+1], \dots, a[r]$ 分别加上 x）；
- `2 l r`：给定 l, r，求 $\sum_{i=l}^{r} a[i]$ 的值（换言之，求 $a[l] + a[l+1] + \cdots + a[r]$ 的值）。

区间修改和区间查询

由区间修改和单点查询的思想，同理借用差分数组实现。

数组C维护原数组a的差分数组b，可得

$$
\begin{aligned}
a_1 &= b_1 \\
a_2 &= b_1 + b_2 \\
a_3 &= b_1 + b_2 + b_3 \\
&\dots \\
a_n &= b_1 + b_2 + b_3 + \cdots\cdots + b_n
\end{aligned}
$$

$$
\begin{aligned}
&\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots a_n \\
&= b_1 + (b_1 + b_2) + (b_1 + b_2 + b_3) + \cdots + (b_1 + b_2 + b_3 + \cdots + b_n) \\
&= \sum_{i=1}^{n} \sum_{j=1}^{i} b_j
\end{aligned}
$$

树状数组

扩展:

给定数列 $a[1], a[2], \ldots, a[n]$，你需要依次进行 q 个操作，操作有两类：

- 1 l r x：给定 l, r, x，对于所有 $i \in [l, r]$，将 $a[i]$ 加上 x（换言之，将 $a[l], a[l+1], \ldots, a[r]$ 分别加上 x）；
- 2 l r：给定 l, r，求 $\sum_{i=l}^{r} a[i]$ 的值（换言之，求 $a[l] + a[l+1] + \cdots + a[r]$ 的值）。

区间修改和区间查询

这个公式横着看看不出来撒，此时我们竖着看，其中有n个b_1,n-1个b_2,n-2个b_3,...,1个b_n，即这些数求和

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots a_n =$$

$$\begin{aligned}
& b_1 + \\
& b_1 + b_2 + \\
& b_1 + b_2 + b_3 + \\
& \cdots + \\
& b_1 + b_2 + b_3 + \cdots + b_n
\end{aligned}$$

$$nb_1 + (n-1)b_2 + \cdots + b_n = \sum_{i=1}^{n} (n+1-i) \times b_i \text{(将里面的和拆成两个和)}$$

$$= \sum_{i=1}^{n} (n+1) \times b_i + \sum_{i=1}^{n} (-i) \times b_i = (n+1)\sum_{i=1}^{n} b_i - \sum_{i=1}^{n} i \times b_i$$

$$= \sum_{i=1}^{n} \sum_{j=1}^{i} b_j$$

树状数组

给定数列 $a[1], a[2], \ldots, a[n]$，你需要依次进行 q 个操作，操作有两类：

- `1 l r x`：给定 l, r, x，对于所有 $i \in [l, r]$，将 $a[i]$ 加上 x（换言之，将 $a[l], a[l+1], \ldots, a[r]$ 分别加上 x）；
- `2 l r`：给定 l, r，求 $\sum_{i=l}^{r} a[i]$ 的值（换言之，求 $a[l] + a[l+1] + \cdots + a[r]$ 的值）。

扩展:

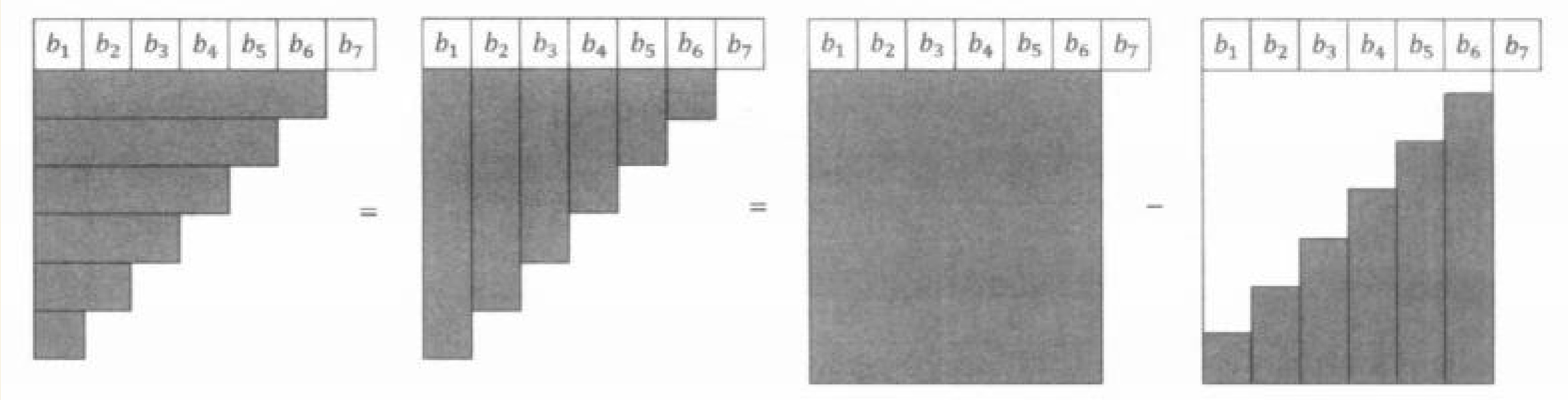
区间修改和区间查询

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n}\sum_{j=1}^{i} b_j$$

这个推导也可以用图形来直观描绘:

$$= \sum_{i=1}^{n}(n+1)\times b_i + \sum_{i=1}^{n}(-i)\times b_i = (n+1)\sum_{i=1}^{n} b_i - \sum_{i=1}^{n} i\times b_i$$

最终求的是n个b_1,n-1个b_2,n-2个b_3,...,1个b_n,此处可理解为将**每个b_i补全成(n+1)个**，再**减去i个b_i的和**。



树状数组

扩展:

给定数列 $a[1], a[2], \dots, a[n]$，你需要依次进行 q 个操作，操作有两类：

- 1 l r x：给定 l, r, x，对于所有 $i \in [l, r]$，将 $a[i]$ 加上 x（换言之，将 $a[l], a[l+1], \dots, a[r]$ 分别加上 x）；
- 2 l r：给定 l, r，求 $\sum_{i=l}^{r} a[i]$ 的值（换言之，求 $a[l] + a[l+1] + \cdots + a[r]$ 的值）。

区间修改和区间查询

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n}\sum_{j=1}^{i} b_j$$

$$= \sum_{i=1}^{n}(n+1)\times b_i + \sum_{i=1}^{n}(-i)\times b_i = (n+1)\sum_{i=1}^{n} b_i - \sum_{i=1}^{n} i\times b_i$$

因此只需维护两个树状数组即可

一个是**差分数组b[i]的树状数组C1[i]**，还有一个是**i*b[i]的树状数组C2[i]**

- 初始化树状数组，C1[i]=a[i]−a[i−1]　C2[i]=i*(a[i]-a[i-1])

add(C1,i,a_i−a_{i-1}),　**add(C2,i,i*(a_i−a_{i-1}))**

- 区间 [l,r]+x（区间内所有的数都加 x）

add(C1,l,+x),add(C1,r+1,-x),　**add(C2,l,l*x),add(C2,r+1,-(r+1)*d)**

- 询问区间[l,r]的和，执行　已知前缀和**pre(x)=(x+1)*sum(C1,x)-sum(C2,x)**

pre[r]-pre[l-1]

树状数组区间修改和区间查询完整代码链接

树状数组

1、树状数组可以单点修改区间查询(至于区间修改和单点查询，区间修改和区间查询一般可由差分前缀和的思想实现)。
2、单操作时间复杂度O(logN),空间复杂度O(N).
3、代码简洁通用。

线段树和树状数组比较

1、线段树可以做到的，树状数组不一定能，树状数组可以做到的，线段树一定能。

2、树状数组的常数明显小于线段树

3、线段树的代码量高于树状数组，但能解决的问题类型也多了很多。

离散化

离散化，把**无限空间中有限的个体映射到有限的空间**中去，以此提高算法的时空效率。

通俗的说，离散化是在不改变数据相对大小的条件下，对数据进行相应的缩小。例如：

原数据：1,1000000,100,30000,999999999;

处理后：1,4,2,3,5;

离散化

适用范围：

有些数据本身很大， 自身无法**作为数组的下标保存对应的属性。** 如果这时只是需要这堆数据的相对属性， 那么可以对其进行离散化处理。

当数据**只与它们之间的相对大小有关**，而与具体是多少无关时，可以进行离散化。

- **利用STL离散化——重复元素离散化后的数字相同**
- **利用结构体排序——重复元素离散化后的数字不同**

实际情况处理**重复元素离散化后的数字相同**较多

离散化

利用STL离散化——重复元素离散化后的数字相同

思路是：先排序，再**删除重复元素**，最后就是索引元素离散化后对应的值。
假定待离散化的序列为a[n]，b[n]是序列a[n]的副本,c[n]可为去重后的数组

删除重复元素

方法一：

```
int k=unique(b+1,b+n+1)-(b+1);//去重,并获得去重后的长度k
```

方法二：

```
int k=0;
for(int i=1;i<=n;i++){//其中副本b数组排好序
    if(i==1||b[i]!=b[i-1])
        c[++k]=b[i];
}
```

离散化

利用STL离散化——重复元素离散化后的数字相同

```
int n,a[MAXN],b[MAXN];
int res[MAXN];
//以下标1为序列的起点，一般情况下从0开始也可以
for(int i=1;i<=n;i++){
    scanf("%d",&a[i]);
    b[i]=a[i];//b是一个临时数组，用来得到离散化的映射关系
}
//下面使用了STL中的sort(排序)，unique(去重)，lower_bound(查找)函数
sort(b+1,b+n+1);//排序
int k=unique(b+1,b+n+1)-(b+1);//去重,并获得去重后的长度k
//注:实际上[k+1,n]仍存在,保存着重复元素
for(int i=1;i<=n;i++){
    //通过二分查找lower_bound，快速地把元素和映射对应起来(返回是0~k-1)
    res[i]=lower_bound(b+1,b+k+1,a[i])-(b+1);
    printf("%d ",res[i]);
}
```

例如:
6
1 23424 242 65466 242 0

输出:

1 3 2 4 2 0

离散化

利用结构体排序——重复元素离散化后的数字不同

排序之后，枚举着放回原数组

用一个结构体存下原数和位置，按照原数排序，最后离散化后数在res数组里面

例如:
6
1 23424 242 65466 242 0

输出:

2 5 3 6 4 1

```
int res[MAXN];
struct node{
    int dat;
    int id;
}s[MAXN];

int cmp(node x,node y){
    return x.dat<y.dat;
}

for(int i=1;i<=n;i++){
    scanf("%d",&s[i].dat);
    s[i].id=i;
}
sort(s+1,s+n+1,cmp);
for(int i=1;i<=n;i++)
    res[s[i].id]=i;//存储的是1~n
for(int i=1;i<=n;i++)
    printf("%d ",res[i]);
```