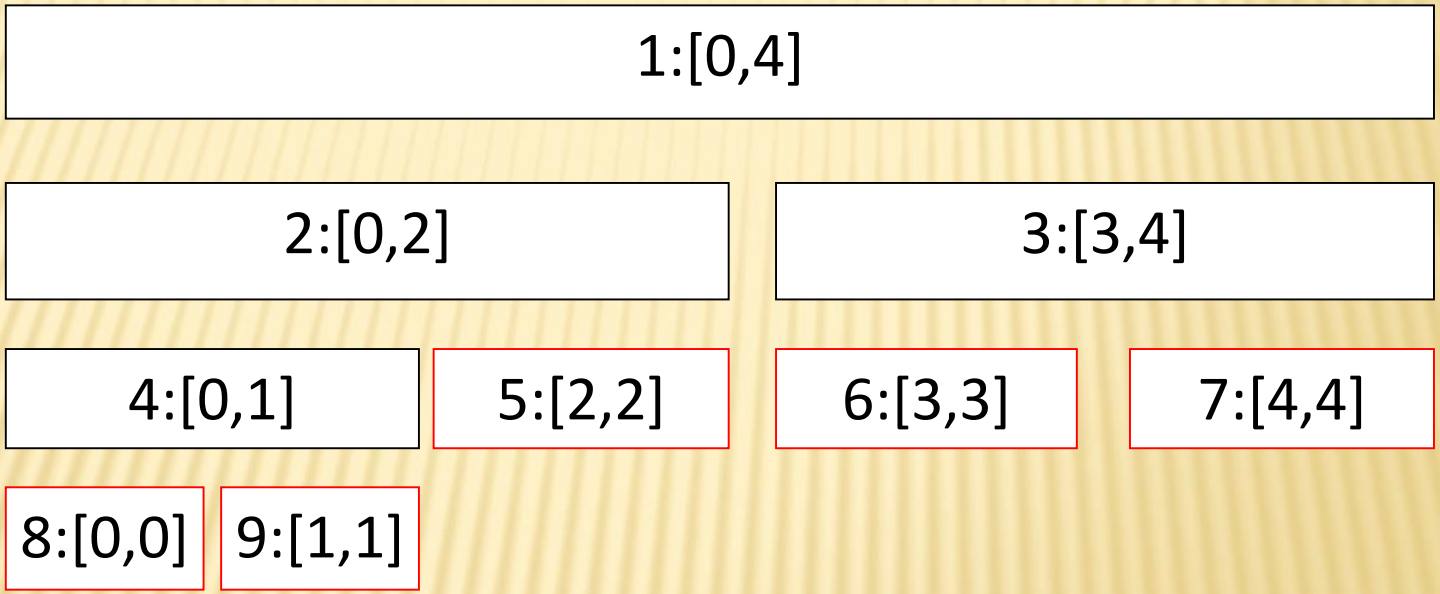
线段树的一些高级技巧v2.0

By 1120132001

张昆玮枚举区间法

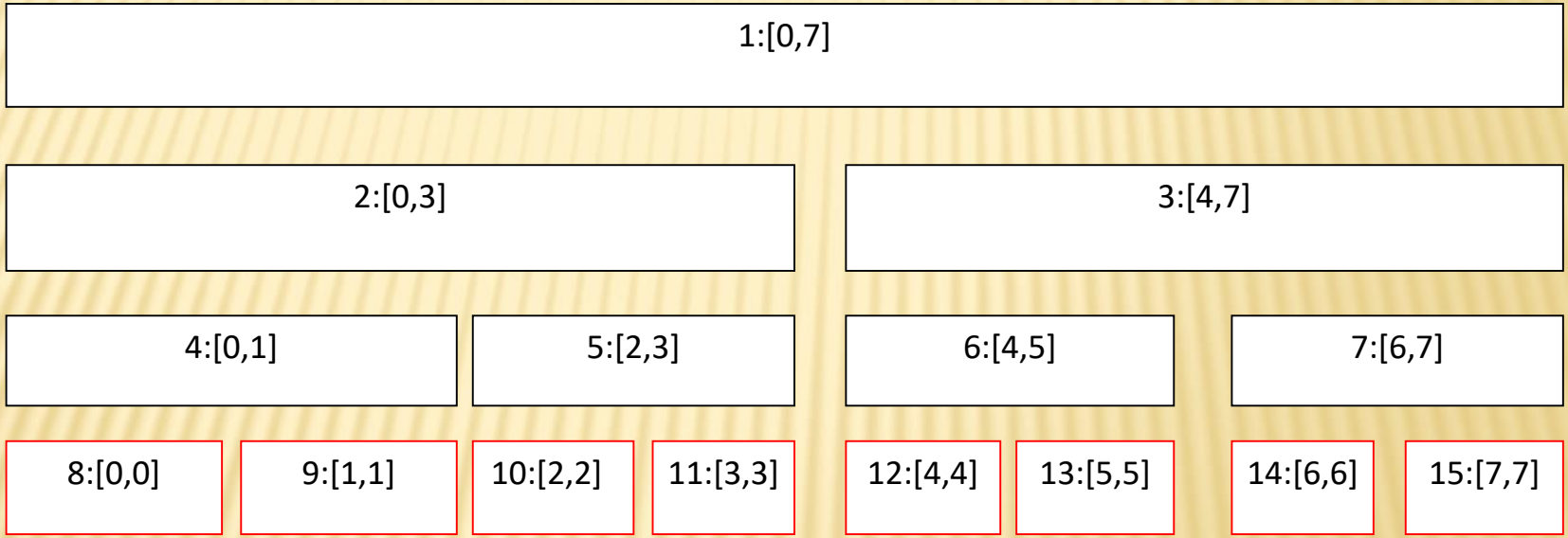
- 线段树的数组下标和线段本来是没有联系的.....



- 但是如果我们强行把它变成满二叉树的话.....

张昆玮枚举区间法

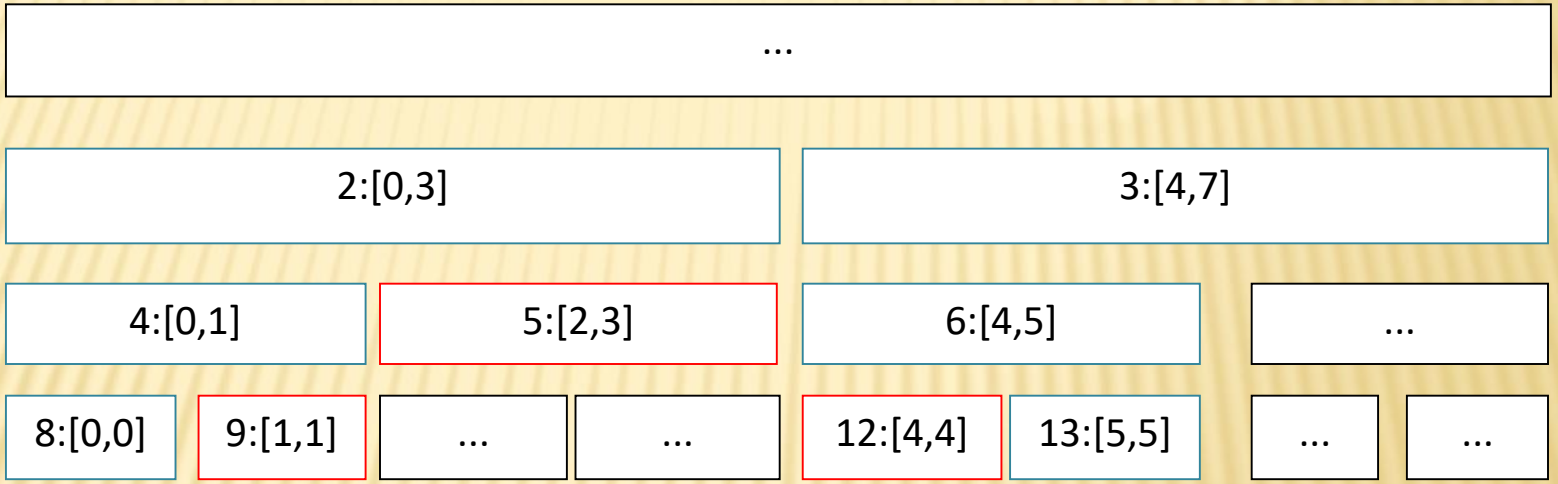
- 假设线段树建立在[0,N-1]上，
- 那么[i,i]就是node[i+N]



- 有些叶子没有用，不过这不重要。
- 继续，我们假定所有的查询都是[1,N-2]范围内的

张昆玮枚举区间法

- 可以直接找到叶子，然后向上走，以[1,4]为例：



- 红色是我们需要合并的区间，注意看蓝色的部分：
- 左边的二进制码是1000->100->10
- 右边的是1101->110->11

张昆玮枚举区间法

- 天然的非递归方法：

```
int query(int l, int r){  //查询[l,r]
    int lret = 0, rret = 0;
    //转化成开区间，每次卡取内部
    for(l+=N-1, r+=N+1; l^r^1;l>>=1,r>>=1){
        if(~l & 1)  lret = lret + node[l ^ 1];
        if( r & 1)  rret = node[r ^ 1] + rret;
    }
    return lret + rret;
    //注意这里并不要求加法是有序的，可以换成其他任何操作
}
```

- 非递归枚举区间就是这么简单！
- 注意我们已经假定l>0,r<N-1了，所以不会溢出
- 如果不得不查询N-1,那么提前把N翻一倍就好。

张昆玮枚举区间法

- 修改其实也很简单：

```
void change(int pos, int val){
    pos += N;
    node[pos] = val;
    for(pos >>= 1; pos; pos >>= 1){
        node[pos] = node[pos<<1] + node[(pos<<1)+1];
    }
}
```

- 从叶子直接到根，该维护啥维护啥。

张昆玮枚举区间法

- 建树更简单

```
void build(int N){
    int i;
    //假设N是2的幂次
    for(i=N-1;i>0;i--)
        node[i+N] = arr[i];
    for(i=N-1;i>0;i--)
        node[i] = node[i<<1] + node[(i<<1)+1];
}
```

张昆玮枚举区间法

- 但是也有一些问题:
- 1. 区间染色、区间修改等等操作需要合理处理标记。（详见zkw《统计的力量》）
- 2. 队友可能看不懂你的写法。
- 3. 操作[1,7]的时候不得不倍增成N=16

- 建议:
- 常数虽小，不要滥用。适度骗分不会TLE。
- 有时不得不先枚举区间然后深入查找某个点的，
- 拿过来用会有奇效，相当简练。

离散化

- 问题：一段长为2^64的墙，每次放一张海报盖住其中的一部分，一张海报如果被其他海报完全盖住就看不见了。放了N张海报之后，问能看见几张？
- 分析：相当于对一段染色，最后统计剩余的颜色数。如果就2^64的坐标建线段树，显然内存不够。

离散化

- 如果输入的是[12,18],[6,11],[10,15]
- 其实和[4,6],[1,3],[2,5]没什么差别
- 所以预先读入所有的输入坐标，排序去重后，把它们直接转化成大小排名就好了。
- 假设一共10^5个线段，于是依然只需要对2*10^5建树即可。

离线+离散化

- 题目：先输入二维平面上若干个点，然后Q次询问，每次给出(a,b)查询x∈ [0,a],y∈ [0,b]的矩形内覆盖了多少个点。
- 分析：输入x，y坐标分别离散化，然后呢？
- 二维线段树？MLE.....
- 其实不需要。
- 只需要把所有的点按x坐标排序，所有的查询也按x坐标排序，然后就可以算了。
- 线段树维护的是某y坐标范围内点的个数。

离线+离散化

- i=0;线段树[]={0};
- For(j=1;j<=Q;j++)
- 　if(第j次查询的x坐标大于点i的x坐标)
- 　　线段树[第i个点的y坐标] += 1;
- 　查询区间(0,第j次查询的y坐标)的和

- 最后重新排序查询，按顺序输出即可。
- 其实在线也是可以的.....

持久化

- 先想想原来我们是怎么建树的：（不是zkw线段树哈）

```
struct NODE{
    int l, r;
    int val;
}node[100];
void build(int id, int l, int r){
    int mid = (l + r) >> 1;
    node[id].l = l;
    node[id].r = r;
    if(l == r){
        node[id].val = arr[mid];
    }else{
        build(id << 1, l, mid);
        build((id << 1) + 1, mid + 1, r);
        pushup(id);
    }
}
```

持久化

- 其实id和id*2和id*2+1的位置关系不是必要的，也就是说我们可以动态管理：

```
struct NODE{
    int l, r;
    NODE *lp, *rp;
    int val;
}node[MAXN*4];
int cnt;
NODE *build(int l, int r){
    NODE *ret = &node[cnt++];
    int mid = (l + r) >> 1;
    ret->l = l;
    ret->r = r;
    if(l == r){
        ret->val = arr[mid];
    }else{
        ret->lp = build(l, mid);
        ret->rp = build(mid + 1, r);
        pushup(ret);
    }
    return ret;
}
```

持久化

- 于是乎节点cur的孩子是cur->lp和cur->rp
- 不一定是node[id*2].....
- 下标不一定比id大.....
- 甚至不一定是**这棵**线段树的node.....

- 节点的子树可以是历史上任何一棵线段树的子树.....
- 这意味着什么呢?

持久化

- 还回到之前那道题，如果我们可以对所有离散化的x，各建一颗线段树维护[0,x]区间内每个y坐标范围内点的个数，那么就不必离线了。
- 原始的方法，空间是O(N*N)的，MLE.....
- 但是每次更新一个x坐标不过是加入若干个点.....
- 每次加入一个点不过是更新一条路径.....
- 每次更新一条路径不过是修改log N个node.....

- 既然每次就修改log N个node，剩下的**用旧的**不就好了么？
- O(N log N)的空间，这总不会MLE了.....

持久化

- 原来的更新方法：

```
void update(int id, int y){
    if(node[id].l == node[id].r){
        node[id].val ++;
    }else{
        int mid=(node[id].l+node[id].r)>>1;
        if(y <= mid)
            update(id<<1, y);
        else
            update((id<<1)+1, y);
    }
}
```

```
void update(int id, int y){
    while(node[id].l!=node[id].r){
        int mid=(node[id].l+node[id].r)>>1;
        if(y <= mid)
            id = (id << 1);
        else
            id = (id << 1) + 1;
    }
    node[id].val ++;
}
```

- 左边是递归版，右边是循环版。
- 我们令cur=&node[id]，然后.....

持久化

- 每次更新，不是修改旧的，而是构造新的：

```
NODE *update(NODE *cur, int y){
    NODE *ret = &node[cnt++];
    (*ret) = (*cur);
    if(ret->l == ret->r){
        ret->val ++;
    }else{
        int mid=(ret->l+ret->r)>>1;
        if(y <= mid)
            ret->lp = update(ret->lp, y);
        else
            ret->rp = update(ret->rp, y);
    }
    return ret;
}
```

- 虽然只有很微小的差异，但是旧的node被我们保留下来了！

持久化

- 每次加入新node的时候，我们这么干：

```
NODE *root[MAXN];
int rcnt;
int Insert(int y){
    // 每次更新，构造一个新的根，相当于一棵新的线段树
    root[rcnt] = update(root[rcnt-1], y);
    // 然后返回新构造的线段树
    return (rcnt ++);
}
```

- 我们当然知道x坐标变化的时候对应的Insert的返回值是啥
- 也就是说随便给我们一个x，我们知道对应的线段树的根是啥，也知道对应的线段树是啥，这棵线段树维护的是y坐标区间上的点的个数
- 也就是说随便给我们一个a，和一个b的区间，我们知道[0,x]X[0,y]矩形内点的个数

持久化

- 这不就解决了么？
- 如果还MLE，我们还有能省内存的地方.....
- 这是原来的查询（注意已经换了持久化线段树了）：

```
int Query(NODE *root, int yl, int yr){
    if(root->l==yl&&root->r==yr){
        return root->val;
    }else{
        int mid=(root->l+root->r)>>1;
        //注意这里如果有lazy下传的话就别复制节点了，直接改没问题
        if(yr<=mid){
            return Query(root->lp,yl,yr);
        }else if(yl>mid){
            return Query(root->rp,yl,yr);
        }else{
            return Query(root->lp,yl,mid)
                +  Query(root->rp,mid+1,yr);
        }
    }
}
```

此处表述有误，见29页

- 修改也差不多。思考：root->l,r有必要存么？

持久化

```
struct NODE{
    //int l, r;
    //已经不需要存了
    NODE *lp, *rp;
    int val;
}node[MAXN*30];
//这个时候MAXN要乘大点
int cnt;
```

```
//放在修改的内部
NODE *update(NODE *cur, int y, int L, int R){
    NODE *ret = &node[cnt++];
    (*ret) = (*cur);
    if(L == R){
        ret->val ++;
    }else{
        int mid=(L+R)>>1;
        if(y <= mid)
            ret->lp = update(ret->lp,y,L,mid);
        else
            ret->rp = update(ret->rp,y,mid+1,R);
    }
    return ret;
}
int Insert(int y){
    root[rcnt] = update(root[rcnt-1],y,1,N);
    return (rcnt ++);
}
```

持久化

```
//还有查询的
int Query(NODE *root, int yl, int yr, int L, int R){
    if(L==yl&&R==yr){
        return root->val;
    }else{
        int mid=(L+R)>>1;
        if(yr<=mid){
            return Query(root->lp,yl,yr,L,mid);
        }else if(yl>mid){
            return Query(root->rp,yl,yr,mid+1,R);
        }else{
            return Query(root->lp,yl,mid,L,mid)
                +  Query(root->rp,mid+1,yr,mid+1,R);
        }
    }
}
```

- 看，sizeof(NODE)不是又变回12了？
- 但是MAXN*30就是没办法的事情了.....

全域化

- 其实内存依然有很多多余的地方.....
- 还记得建树时候我们做的事么？因为所有点的x坐标大于0，而我们恰好是对x=0建的树，于是所有的arr[i]=0，所有的node[i].val=0.....
- 然后我们每次Insert都建立了新的node，旧的N*2-1个node全部都还保留着，于是.....

- 我们要那么多val=0的node干啥啊？！

全域化

```
NODE *pending;
//表示：这个节点本来应该是有的，但是还没来得及建
Segtree(){
    pending = &node[0];
    cnt = 1;  //已经用了一个节点了
    //左右孩子也都没来得及建
    pending->lp=pending->rp=pending;
    //此区间内没有点
    pending->val = 0;
}
NODE *build(){
    NODE *ret = &node[cnt++];
    ret->val = 0;
    ret->lp = ret->rp = pending;
    return ret;
}
```

```
//Update完全不用变，查询这里改一下
int Query(NODE *root, int yl, int yr, int L, int R){
    if(root==pending || (L==yl&&R==yr)){
        return root->val;
    }else{
        int mid=(L+R)>>1;
        if(yr<=mid){
            return Query(root->lp,yl,yr,L,mid);
        }else if(yl>mid){
            return Query(root->rp,yl,yr,mid+1,R);
        }else{
            return Query(root->lp,yl,mid,L,mid)
                +  Query(root->rp,mid+1,yr,mid+1,R);
        }
    }
}
```

- 于是乎我们只有1个0节点了。
- （代码没调试过不保证，思路肯定没错就是）

全域化

- 但是还不仅仅是这样.....
- 还记得原来的空间复杂度么？假设有Q次插入，线段树维护[0,N-1]的区间，要求保留全部历史记录。
- 二维线段树，或者每次复制一份，复杂度O(N*N)
- 持久化线段树，复杂度O(2*N+Q*logN)
- 去掉最开始的一堆0后复杂度O(Q*logN)
- 已经跟N没多大关系了.....

- 那么我们还离散化y干啥?
- 令N=2^32完事。

全域化

- 于是这时候：
- root=>[0,2^32-1]
- root->lp=>[0,2^31-1]，最高位是0的unsigned long
- root->rp=>[2^31,2^32-1]，最高位是1的unsigned long
- root->lp->lp=>00xxxx
- root->lp->rp=>01xxxx
- root->rp->lp=>10xxxx
- root->rp->rp=>11xxxx
- ……
- 这不就是二进制的**字典树（Trie树）**么？

总结

- 回顾整个ppt:
  - 最开始的时候我们讲了ZKW线段树:

    树状数组 = 线段树
  - 然后我们讲了离散化，讲了离线处理;
  - 然后又想办法用空间换取在线的机会，用pending代替离散化

    字典树 = 线段树
  - 等到那一天给Splay或者非旋转Treap打上lazy标记的时候你会发现:

    平衡树 = 线段树
  - ACM数据结构题=>万有线段树定律.....

综合题目：区间K大

- 给定一串数字，每次查询其中一段的第k大的数是多少。
  - 我们用**可持久化线段树**，按顺序插入数并维护历史，每次查询**二分答案**然后通过两棵线段树的差解决这个问题。
- 如果加上随时修改某点呢?
  - 我们先**离散化**所有的值，在前一问的基础上用范围为离散化的值域的**树状数组**套上一个可持久化线段树，构成一棵**主席树**.....
- 如果再加上随时插入和删除某个点呢?
  - 我们用**替罪羊树**啥的高大上的玩意套上一个可持久化线段树.....
- 如果再加上.....?
  - 我们用**块状链表**直接暴力出答案.....

那些东西都已经不那么重要了，如果想知道，自己百度去吧!

V2.0更正和补充内容

- 好吧我又来了，之所以要改这一发交上了是因为看到之前我写说可持久化线段树“lazy标记下传不用创建新节点”，然后发现这里其实是误导，因为不用创建新节点仅仅是query函数里不用创建新的而已，我把创建和复制节点的工作放在pushdown里了.....
- 这里必须复制的原因是显然的，因为我们并不知道root,lp,rp都是新树的节点还是旧树的，如果不复制，万一其中一部分是新树的而另一部分是旧树的，直接崩溃。当然要是学zkw直接标记永久化就没事了不过显然有些标记是没法永久化的。
- 然后就是考完试发现之前ppt有一些错误又做出了修改。
- 然后就是关于函数式编程的一些基本概念，想想还是普及一下的好。虽然不是常考知识点，但是知道了总没坏处不是么。
- 然后就没有然后了。NB的学弟们尽管喷我吧.....

副作用

- 什么是可变的?
- 比如说Java里Person类维护一个Date类型的成员表示生日，有一个函数可以供朋友查询这个Date。然后其他人查询到这个人的生日之后修改了一下，然后这个人的生日就自动改变了，因为Java里所有的类作为参数传递的时候都是指针传递。
- 于是我们的Person类只能每次有人需要这个Date的时候就复制一个新的出来，防止别人去修改它 =》 **读时复制**
- 可是既然没有改变我们为什么还要复制呢？让修改它的人复制不更好吗？（**写时复制**）
- 于是我们有了final.....

副作用

- 副作用：函数的行为受到全局状态的影响。（简言：“有变量”）
- 无副作用的好处：结果缓存，引用透明。

- 最直接意义在于就是，任何时候可以以引用（或者指针）的形式把数据传递给任何人，而不用担心他们修改它。
- 而在比赛中，我们关注的更多不是安全性的考虑，而是我们可以在任何时候保存一个无副作用的对象当前的状态，而不需要复制它。
- 就像之前的可持久化线段树一样。

写时复制 => 灵活自由的版本控制

- 相应的，比赛时，我们也仅在需要**版本控制**的时候使用无副作用。

引用透明

- 引用透明：对一个对象的引用和上下文无关。

- [北京]在北半球。【真】
- [中国首都]在北半球。【真】
- [(任何代指北京的东西)]在北半球。【真】

- [北京]有两个汉字。【真】
- [中国首都]有两个汉字。【假】
- [(任何代指北京的东西)]有两个汉字。【？？？】

引用透明

- 引用透明：对象的行为只依赖于输入。相同的输入一定有相同的输出。

- int f(int x){return x&(-x);} ——引用透明
- int g; int f(int x){return g&(-x);} ——引用不透明

- 普通线段树的Query ——引用不透明
- 持久化线段树的Query ——引用透明

- 已知引用透明的函数bool g(bool x)
- 功能：一定是x；!x；true；false四类中的一个。
- 它不会做：读取文件，查询数据库，读取键盘输入.....

惰性求值

- 惰性计算：某个值真正被计算，仅在它第一次被需要的时候。
- 例如：

```
int judge(bool cond, int val1, int val2){
    if(cond){
        return val1;
    }else{
        return val2;
    }
}
```

- 然而我们并不需要同时计算val1和val2.....
- 效率不高！

惰性求值

```
int judge(bool cond, int (*val1)(), int (*val2)()){
    if(cond){
        return val1();
    }else{
        return val2();
    }
}
```

- 虽然延迟了计算，但是每次需要val1的时候都算一遍，仍然不好.....

惰性求值

```
class DelayInt{
public:
    bool calced;
    union{
        int val;
        int (*func)();
    };
    DelayInt(int(*func)()):
        func(func),calced(false){}
    int Force(){
        if(!calced){
            val = func();
            calced = true;
        }
        return val;
    }
};
```

- 并不是无副作用的，但是引用透明。
- 基本解决！

惰性求值

- 结果是：
- 如果a[1]二选一的依赖于a[2]和a[3]，a[2]二选一的依赖于a[4]和a[5]，a[n]二选一的依赖于a[2n]和a[2n+1]，而我们需要N次a[1]，那么我们会沿着某一条路径计算到底，然后返回一个a[1]的值，不会计算不需要的路径上的点的值。计算量为对数级别。

- 如果a[2]和a[3]依赖于a[1]，a[4]和a[5]依赖于a[2]，a[2n]和a[2n+1]依赖于a[n]，而我们需要N次某个底层的a[x]，那么我们会沿着a[x]向上的路径，计算它的所有父亲一次，而不会管其他的。计算量为对数级别。

引用计数

- 何时删除一个指针指向的对象？当它不被需要的时候。
- 什么时候一个指针不被需要？.....

- 我们并不能查找到一个指针的所有引用，因为是引用透明的.....
- 可是我们也并不需要找到指针的所有引用才知道它有没有被引用。

- 加一个标记记录一下它被引用了几次就好了！
  =>并非无副作用，然而引用透明

引用计数

```
class OBJ{
private:
    int ref;
public:
    //创建时有一次引用
    OBJ():ref(1){};
    //每次让一个指针指向它的时候增加引用
    void AddRef(){
        ref ++;
    }
    //每次让一个指向它的指针指向别的东西的时候削减引用
    void Release(){
        ref --;
        if(ref == 0)
            delete this;
    }
};
```

```
typedef OBJ *POBJ;
//用DoRef(A, B)来替代A=B（new除外）
//DoRef(A, nullptr)完成释放
void DoRef(POBJ &ptr, const POBJ val){
    if(!ptr)
        ptr->Release();
    ptr = val;
    if(ptr)
        ptr->AddRef();
}
```

指针的释放是自动的！

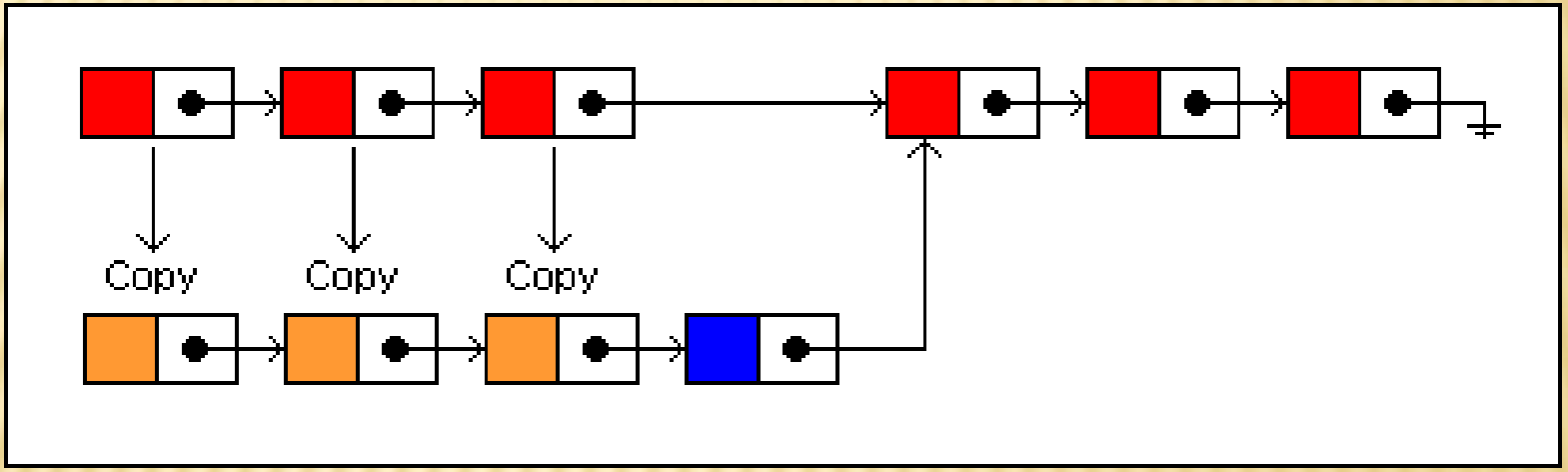
函数式编程概念在C++中的应用

- STL的string：写时复制，引用计数（引用为1的时候不复制直接改）
- COM+：引用计数（实现半自动的垃圾回收）
- G++的rope（持久化红黑树）：写时复制，引用透明
- 段加段改的线段树，Splay：惰性求值（懒标记）
- 可持久化线段树：写时复制（update和pushdown里面的复制），引用透明（root数组），引用计数（这个可以轻微减少节点复制次数和减少内存消耗，但是考虑代码量问题一般不写），惰性求值（pending）
- .....

- 接下来我们看一些经典的可持久化数据结构。

可持久化单向链表

- 如图，实现可以反悔的**栈**很方便，此外基本不用。
- 但在函数式编程语言中是基本的数据结构。



可持久化队列

- 直接用链表实现是不可取的。采用均摊的思想：
- 把队列从中间劈开，拆成队头段F和队尾段T。队头顺序用单项链表存，队尾逆向用单项链表存。
- 定义两种惰性操作：反转和按顺序合并。
- 针对单个链表的反转操作不取出元素时不计算，取出第一个元素时先递归求值然后整体反转。
- 针对两个链表的按顺序合并操作求值是先计算前面的链表并取出一个元素，后面的链表不计算；当前面的链表为空时，去掉合并标记并计算后面的链表。如果是反转的计算过程中遇到合并标记会因为不断地从中取出节点而强迫计算到底。
- 每次队尾插入元素直接插入到T的开头；队头取出元素从F开头取出。
- 不同的是，每次插入删除后比较S和T的大小，如果链表T比S长，那么强制把T打上惰性反转标记后惰性链接到F的后面。
- 均摊复杂度O(1)，但常数很大。

可持久化数组

- 直接搞是不可能的，因为可持久化数据结构根本不支持基于下标的访问，除非每次复制整个数组，但是那样效率还不如链表.....
- 替代性方案1（常用）：用可持久化线段树实现，除了叶子节点外不再存储任何实际数据。但是复杂度会变成O(log n)
- 替代性方案2：用可持久化平衡树，空间开销貌似是减少了，但是时间略有增加（平衡树的维护代价高）。依然是O(log n)
- 替代性方案3：用可持久化块状链表。块状链表每一块的地址存在一个数组里，每次修改暴力复制一份，被修改的块内数据暴力复制一份，剩下的块用旧的。复杂度O(sqrt(n))

可持久化平衡树

- 原始的方法：一般的平衡树（Splay例外，不可持久化），我们每次总是操作O(log n)个数的节点，所以操作哪个节点就复制哪个节点就好。旋转操作会为时间和空间都带来额外的开销。
- G++的rope是可持久化红黑树。

- 非旋转的方法：采用B树或者非旋转Treap，仅限于节点被劈分和合并的时候才复制，不需要考虑旋转带来的问题。特别是非旋转Treap可以打标记和分离区间，即使不持久化也可以当作Splay和可合并堆（左偏树、二叉堆）等的替代品，常数稍大但用途很多。
- 这里不再深入讨论。

- 值得注意的是，可持久化平衡树不能存储父亲节点。（因为同一个孩子在不同时刻有不同的父亲）

可持久化并查集——数组实现

- 并查集其实就是一个数组，每个数组元素的内部写着它的父节点标号、大小、秩等信息。因此可以采用G++的rope，或者线段树模拟持久化数组来实现。
- 仅仅是编写出代码来说并不难。这里不打算多讲。记得union（merge）的时候不光要改孩子的父亲指针，还要改父亲的节点大小和秩。
- 于是我们来谈谈效率问题。

- 人们常说函数式编程的效率依赖于奇怪的地方.....

可持久化并查集——数组实现

- 1. 并查集应该使用编号（数组下标）还是指针

- 普通的并查集我们使用哪一个并没有太大差异，不如说指针的反而更快一些。
- 然而遗憾的是，持久化并查集并不能使用指针。因为线段树的节点是不允许有向父亲节点的指针的。如果使用指针，修改节点的时候很难找到到底改沿哪条路径更新。

可持久化并查集——数组实现

- 2. 是否使用路径压缩？

- 普通的并查集，采用路径压缩技术可以让我们把时间效率从O(log n)压缩到接近O(1)。
- 然而在持久化并查集中不能使用路径压缩。理由如下：
- 1）持久化会在不同版本之间跳转，因而摊还分析失效。基于均摊的策略来降低时间复杂度的做法此时不可取。Splay不能持久化正是这个原因。（好吧其实我不懂我在说什么，大神是这么写的）
- 2）路径压缩会导致每次查询根的操作都有可能进行多次修改。而且这种修改必须复制新节点。不然，先a->b，然后b->c，查询a，如果直接改成改a->c的话，回退一步再查询a，就挂了。这种复制操作会带来相当巨大的内存开销。

可持久化并查集——数组实现

- 3. 是否使用按秩合并?

- 普通并查集我们并不使用这个，因为有了O(α)的路径压缩，我们没有必要使用最坏O(log n)的按秩合并。
- 但是在没有路径压缩的现在按秩合并是必要的。秩的树形结构并不会收到可持久化的影响，时间效率维持最坏O(log n)，随机O(1)。

可持久化并查集——数组实现

- 4. 是否可以用pending来优化?
- 可以。
- 5. 总的时间复杂度是多少?
- 最坏情况其实是O(NlogNlogN)，有时认为是O(NlogN)的（随机数据），但是一般认为是O(NlogNlogN)。
- 6. CDQ分治+可持久化并查集log被乘了3次幂TLE了，怎么办?
- 看下页，其实只是时间轴分治的话我们并不需要可持久化并查集。

可持久化并查集——暴力实现

- 实际上只有需要复杂的版本控制的时候，我们才需要可持久化并查集。如果只有回退这一种操作，不会产生复杂的树形版本结构的话，我们可以用最朴素的方法。

- 注意到如果不采用路径压缩，那么并查集的所有操作都是可以撤销的。于是我们只需要在每次合并操作后用一个堆栈记录3个值：当前合并操作合并的父节点和子节点，以及被并入的子集大小。
- 然后每次需要回退的时候暴力退栈，然后还原合并时的修改操作即可。
- 总时间复杂度O(NlogN)

最后的总结

- 把握基本思想，不要死记模板
- 通过离散化进行值域建树
- 灵活运用位操作和惰性求值提高速度
- 离线和在线、离散化和全域的相互转化思想
- 通过引用透明性和引用计数设计版本控制
- 合理控制内存使用，对算法效率进行估算和规划
- 了解什么情况下不能用上述数据结构
- 通过合理设计，把未知的数据结构用已知的结构来实现

謹　　此