# 线段树

**问题：**

区间查询求和问题:给定一个含有n个整数序列的数组A，查询任意区间最大值与区间和的复杂度为O(n)，若进行m次查询，则总的复杂度为O(mn)。

设计一个数据结构(线段树)，优化区间查询与求和操作，使其时间复杂度为O(logn)。举例描述说明:

1该数据结构的节点与结构;

2建立与更新（数组A中某一个元素值更新时）数据结构的具体步骤;

3任意区间查询与求和操作的具体步骤。

提示:更新数据结构时，采用延迟更新策略，即，更新时，不立即进行更新操作，只标记这个点需要更新，真正使用时（如查询时）才进行更新。

**举例说明：（代码在最后一起展示）**

1. **该数据结构的节点与结构;**

**1结点：**

typedef long long ll;

struct TREE

{

int l,r;//代表节点维护的区间范围;代表了a[l]->a[r]

ll SUM; ////代表了a[l]->a[r]的值的sum总和

ll lazy; //涉及lazy标记的东西;

ll Max;//代表了a[l]->a[r]的值的最大值

}t[MAXN << 2];

L,r代表的是区间Data[l]->Data[r]

SUM代表的是这个区间的所有值的总和

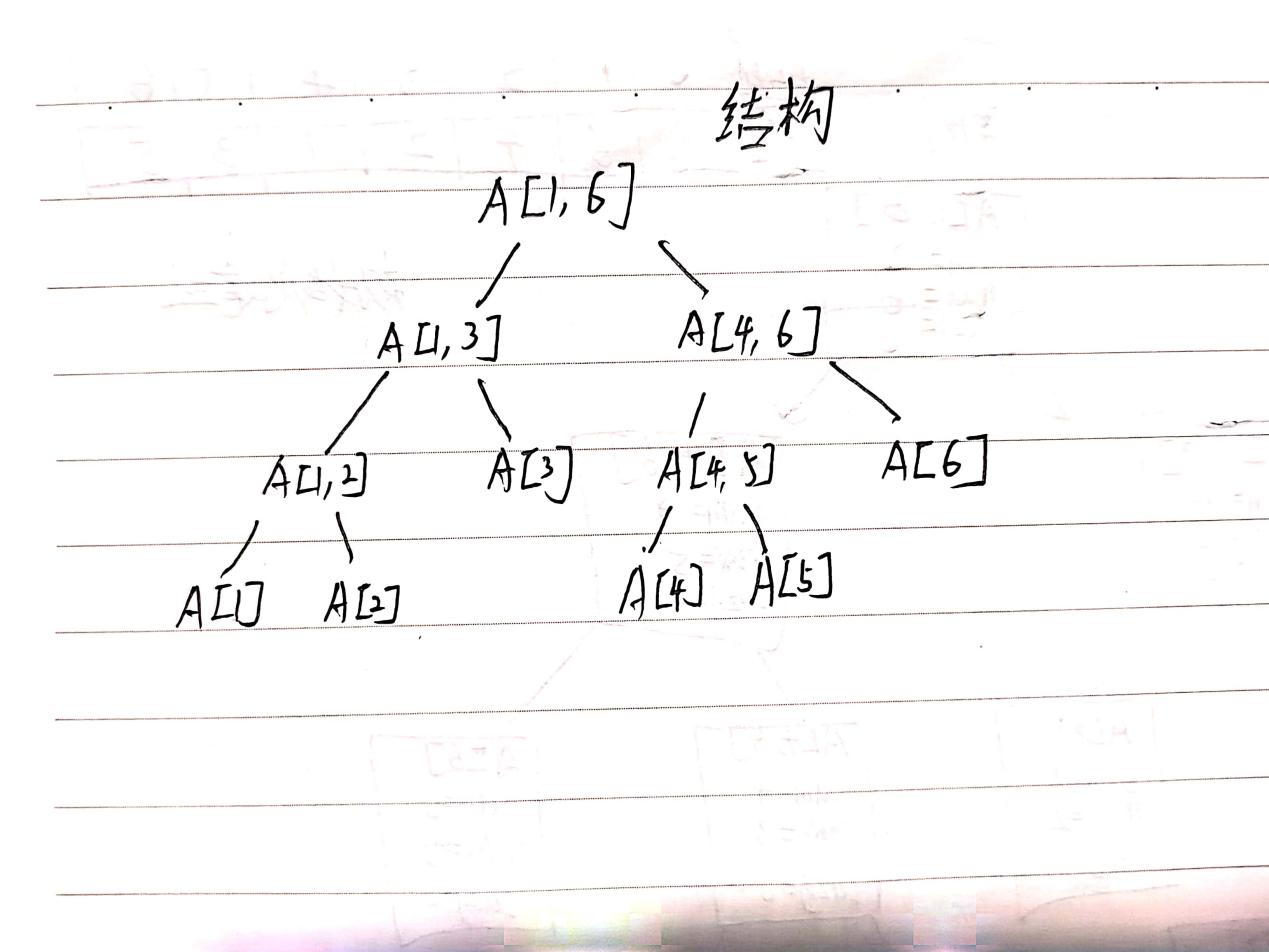
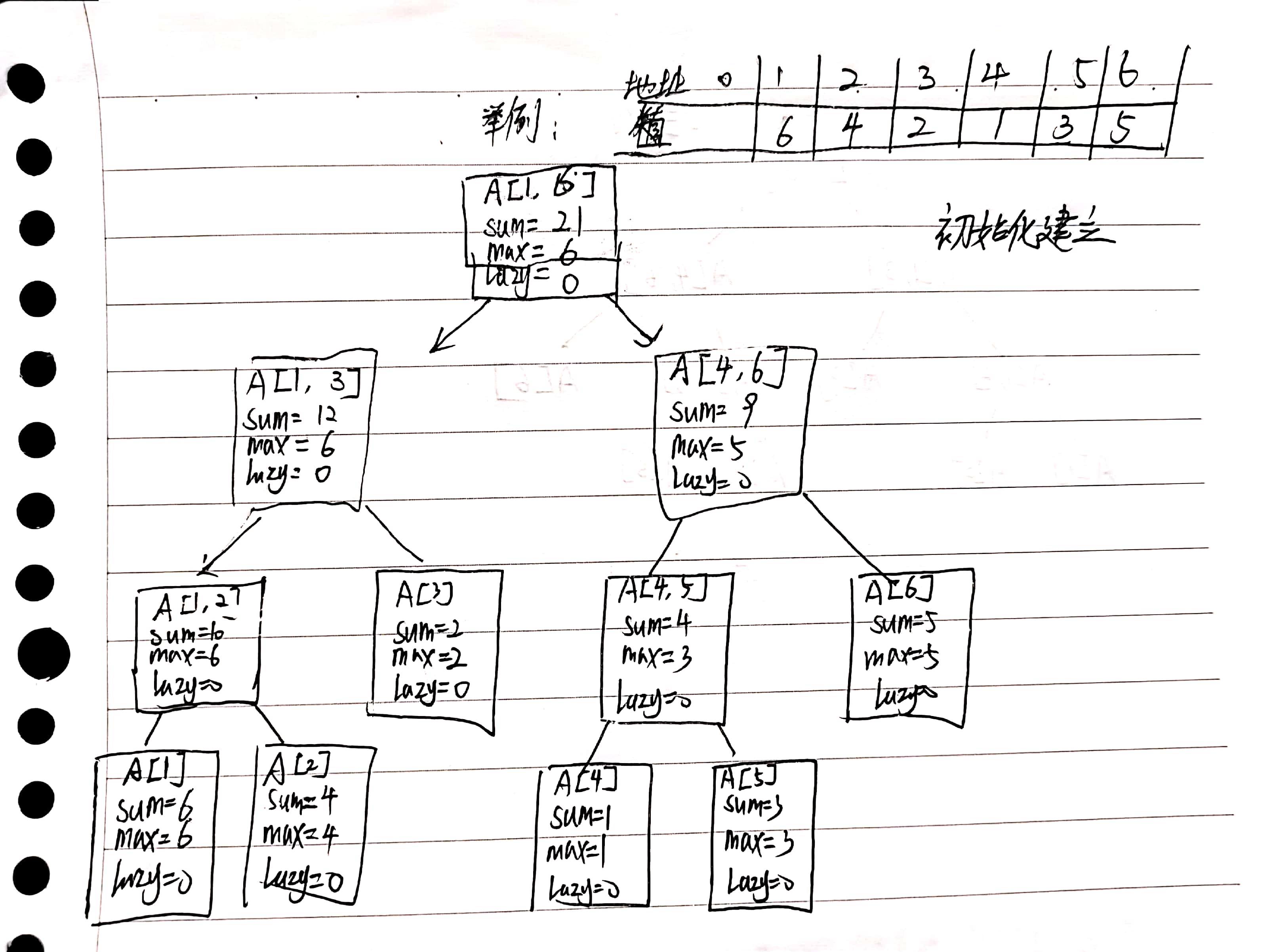
Lazy标记用于采用延迟更新策略

Max代表的是这个区间的所有值的最大值

**2结构：**

采用线段树来优化区间查询，每一个结点储存着Data[L]->Data[R]的信息，其中叶节点的L=R，不断地将大区间划分为小区间，利用递归建立线段树，

**举例A【1】到A【6】的线段树**

****

1. **建立与更新（数组A中某一个元素值更新时）数据结构的具体步骤;**

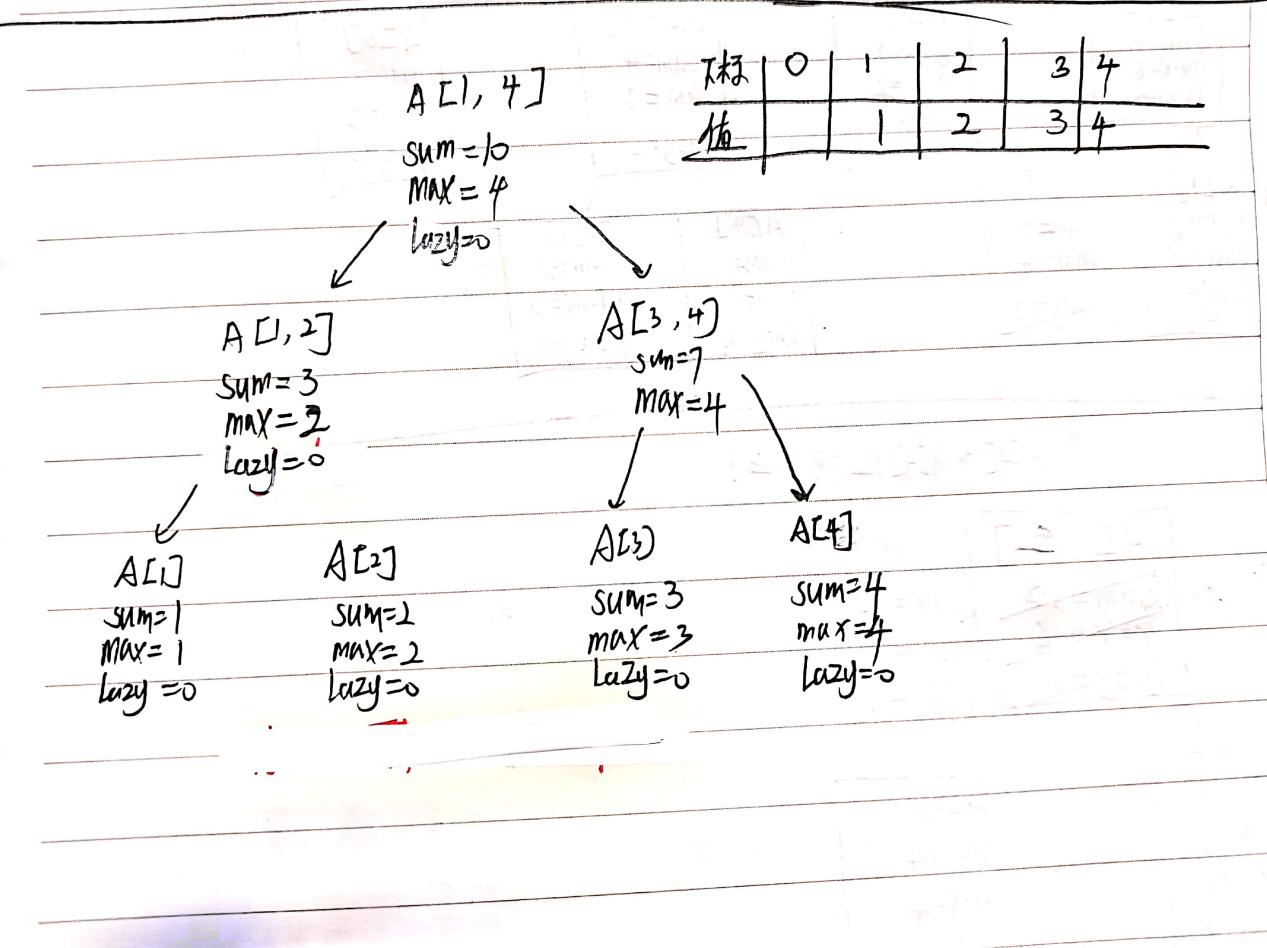
**建立：**

若为叶子结点，直接赋值；否则根据构造左右子树得到该结点的值

### 建立线段树

1. 如果l == r情况则直接将SUM和Max 赋值为a[l]
2. 如果l！=r情况，则递归建立线段树
   1. 定义Mid = t[p].l + t[p].r >> 1;
   2. 递归调用build(lson(p),l,mid);处理左子树
   3. 递归调用build(rson(p),mid + 1,r);处理右子树
   4. 回溯后将子节点信息保存下来t[p].SUM = t[lson(p)].SUM + t[rson(p)].SUM；t[p].Max = max(t[lson(p)].Max,t[rson(p)].Max);

**举例A[1]=1,A[2]=2,A[3]=3,A[4]=4建立线段树**

****

**更新：**

* 1. 单个数值修改线段树（数组A中某一个元素值更新时），可以借助于已有的区间修改，不如想要将令A[L]的值变成y，利用区间更新函数等价于将从A[L]到A[L]的值都加入value =y-A[L]

而区间更新具体方法：

1、如果当前区间被完全覆盖在目标区间里，修改sum和max

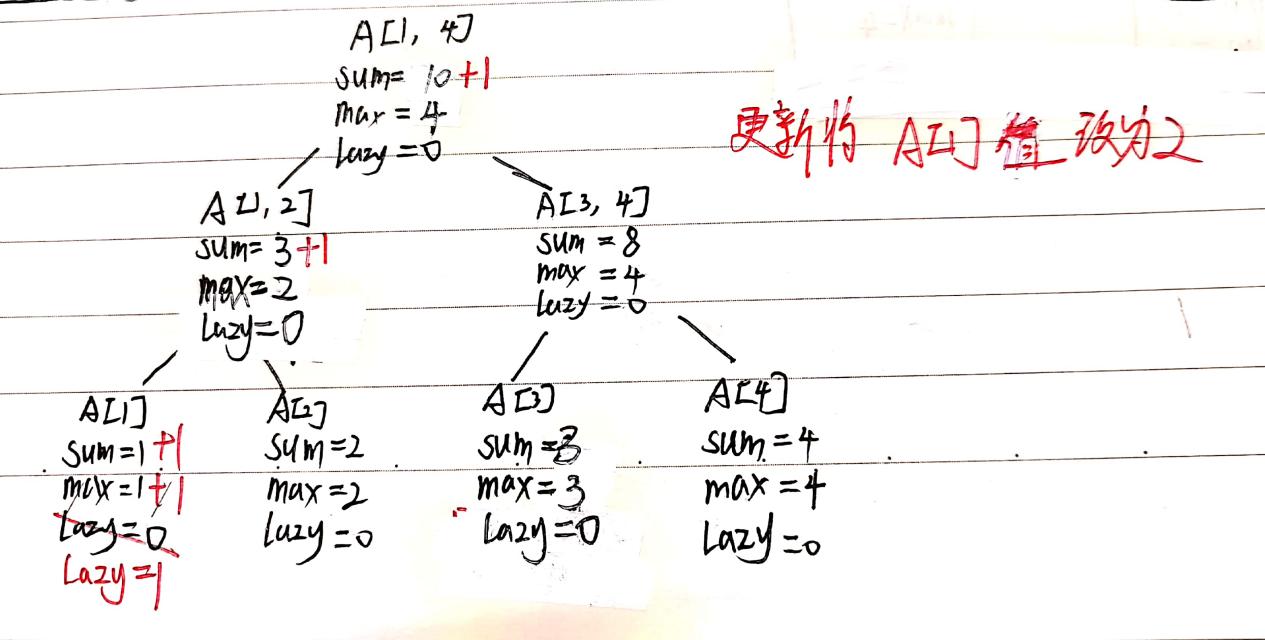
2.如果没有完全覆盖，则先下传lazy

3、如果这个区间的左儿子和目标区间有交集，那么搜索左儿子

4、如果这个区间的右儿子和目标区间有交集，那么搜索右儿子

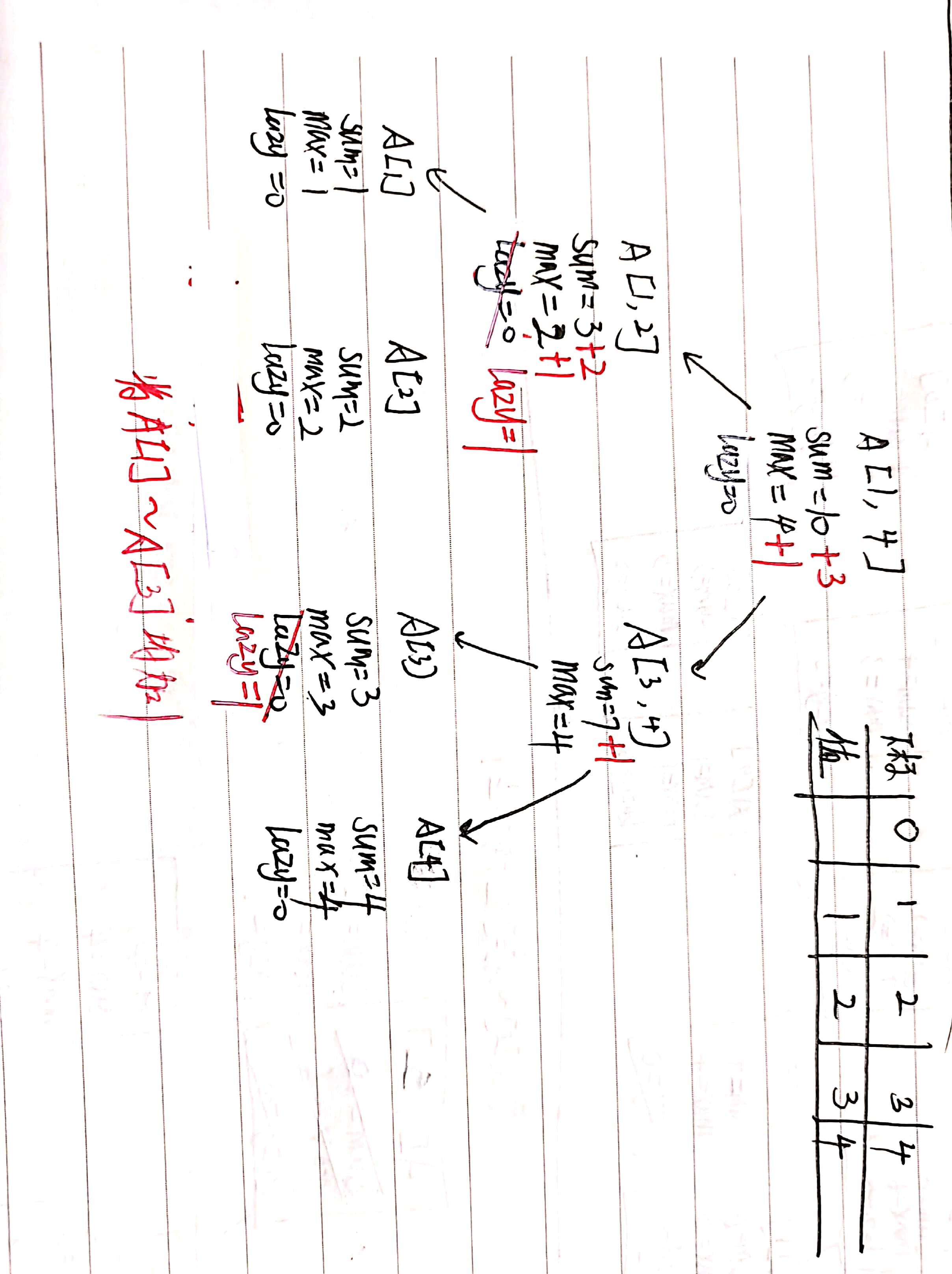
1. 区间更新线段树update函数
   1. 如果区间被覆盖的情况
      1. 直接进行修改
      2. t[p].SUM += value \* (t[p].r - t[p].l + 1);
      3. t[p].Max += value;
      4. t[p].lazy += value;
      5. Return结束
   2. 如果区间没有被覆盖的情况
      1. 执行push\_down（p）函数，执行向下更新左右儿子结点的数据
         1. 当lazy不为零的时候，将左右儿子的值利用lazy修改左右儿子的SUM和Max
         2. 将当前p的lazy下传给左右儿子的lazy
         3. 当前p的lazy清零
      2. 定义mid = t[p].l + t[p].r >> 1便于后续判断
         1. 如果覆盖了左儿子就修改左儿子update(lson(p),l,r,value)
         2. 如果覆盖了右儿子就修改右儿子update(rson(p),l,r,value)
      3. 更新当前p结点的Sum和Max数据
2. 单个数值修改线段树（数组A中某一个元素值更新时）
   1. 令Data[L]的值变成y，利用区间更新函数等价于将Data[L]->Data[L]的值都加入value =y-Data[L]
   2. update(1,l,l,value);

**举例：**

**更新A【1】修改为2（假定A[1]=1,A[2]=2,A[3]=3,A[4]=4建立线段树）**

**举例：**

**更新A【1】到A【3】的值都加入1（假定A[1]=1,A[2]=2,A[3]=3,A[4]=4建立线段树）**

****

1. **任意区间查询与求和操作的具体步骤。**
2. 如果当前区间被完全覆盖在目标区间里，直接return
3. 如果没有完全覆盖，则先pushdown执行向下更新左右儿子结点的数据

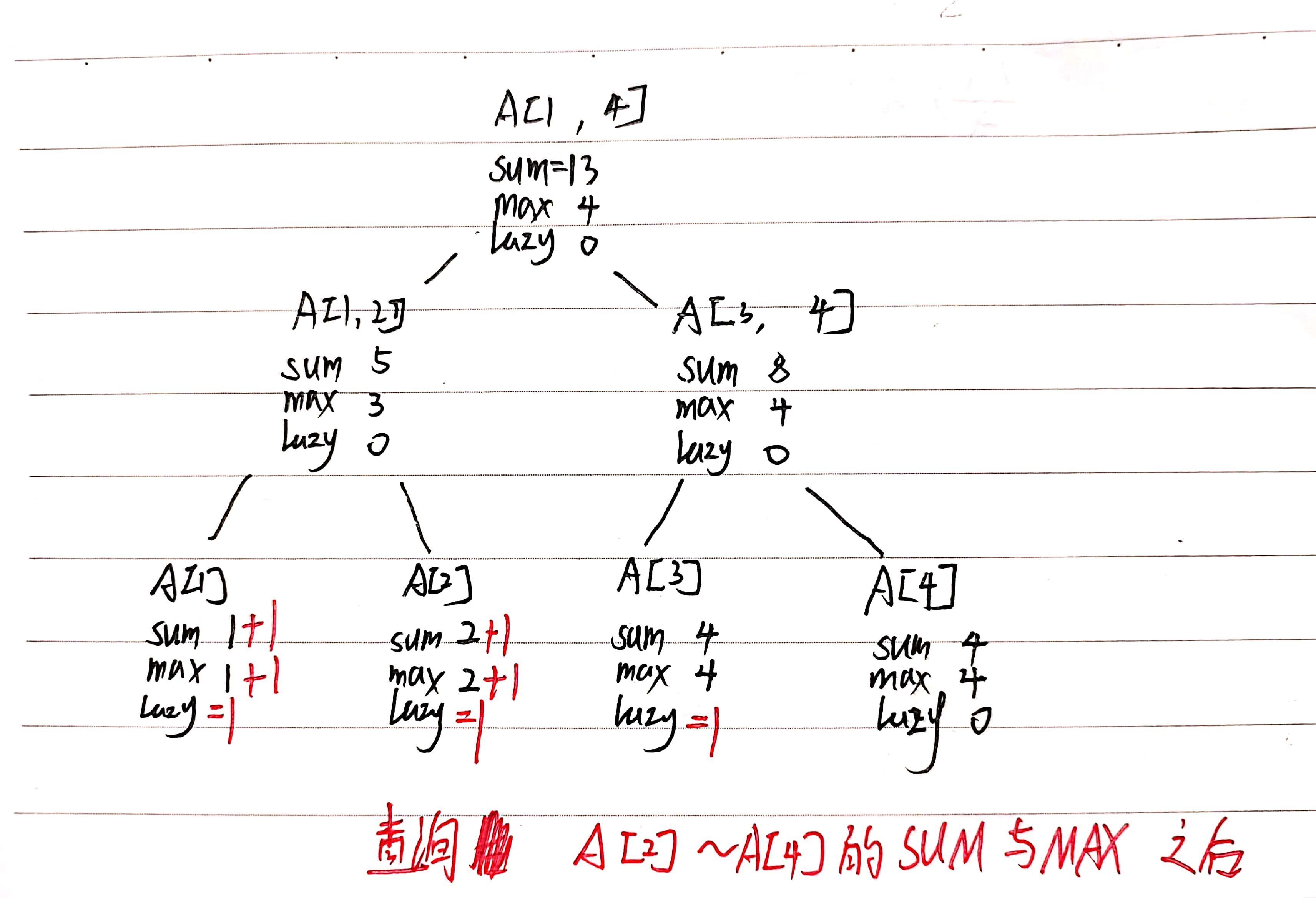
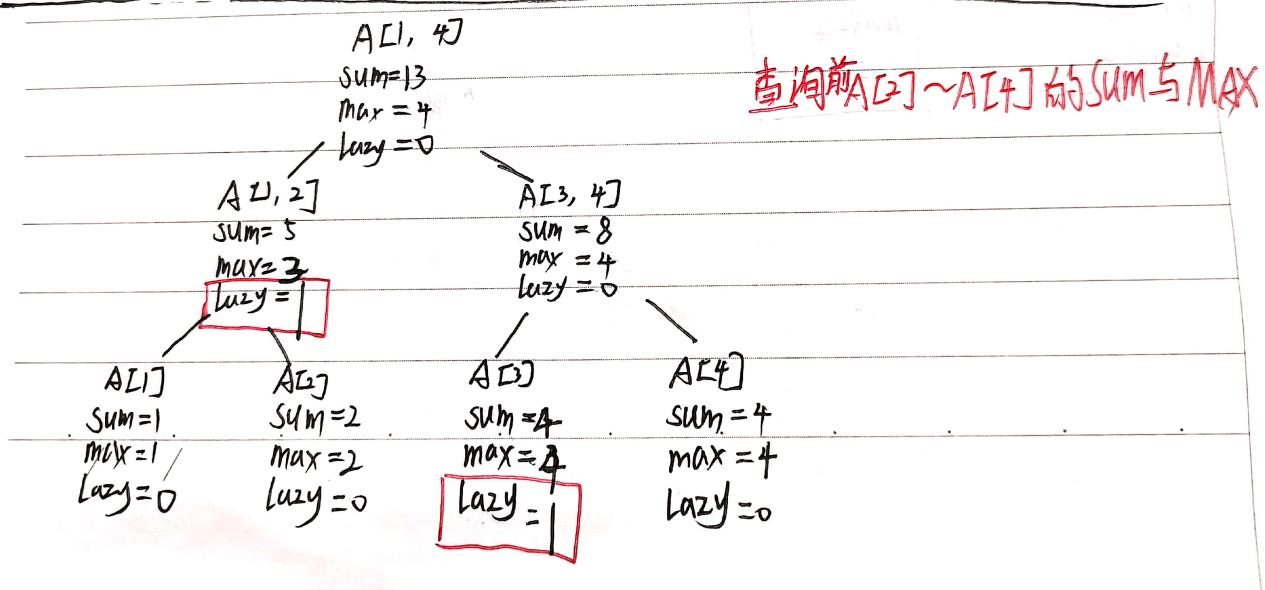
3、如果这个区间的左儿子和目标区间有交集，那么处理左儿子

4、如果这个区间的右儿子和目标区间有交集，那么处理右儿子

1. 查询区间和querySum函数
   1. 如果区间被覆盖
      1. 直接返回区间的数据Returnt [p].SUM
   2. 如果区间没有被覆盖
      1. 执行push\_down（P），执行向下更新左右儿子结点的数据
         1. 当lazy不为零的时候，将左右儿子的值利用lazy修改左右儿子的SUM和Max
         2. 将当前p的lazy下传给左右儿子的lazy
         3. 当前p的lazy清零
      2. 定义mid = t[p].l + t[p].r >> 1;方便后续判断
      3. 如果(l <= mid)需要查询左儿子，就加入整理左儿子的数据sum += querySum(lson(p),l,r)
      4. 如果(r > mid)，如要查询右儿子，就加入整理右儿子的数据sum += querySum(rson(p),l,r);
      5. 返回return sum
2. 查询区间最大值query\_Max函数
   1. 如果区间被覆盖
      1. 直接return t[p].Max
   2. 如果区间没有被覆盖
      1. 定义mid = t[p].l + t[p].r >> 1;方便后续判定
      2. 定义maxL = -inf,maxR = -inf;
      3. 如果(l <= mid)需要查询左儿子，则查左儿子的最大值maxL = max(maxL,query\_Max(lson(p),l,r));
      4. 如果(r > mid)需要查询右儿子，则查询右儿子最大值maxR = max(maxR,query\_Max(rson(p),l,r));
   3. 返回 左右子树的最大值

**举例：**

**查询A【2】到A【4】的区间和，最大值（假定，A[1]=1,A[2]=2,A[3]=3,A[4]=4建立线段树，并且已经更新A【1】到A【3】的值都加入1）**

****

**得到查询结果**

## 算法思想： 采用线段树来优化区间查询，每一个结点储存着Data[L]->Data[R]的信息，其中叶节点的L=R，不断地将大区间划分为小区间，利用递归建立线段树，在利用这些区间进行修改和查询。而在进行更新数据时候利用延迟标记lazy标记，如果当前区间需要被修改的区间被完全覆盖就用lazy标记，在下一次查询或者更新时候才调用lazy下传和清零修改

**算法步骤**

1. 结点：

struct tree

{

int l,r;//代表节点维护的区间范围;代表了a[l]->a[r]

ll SUM; //代表该节点维护的值;//!!!!!!!!!sum总和

ll lazy; //涉及lazy标记的东西;

ll Max;//最大值

}t[MAXN << 2];

L,r代表的是区间Data[l]->Data[r]

SUM代表的是这个区间的所有值的总和

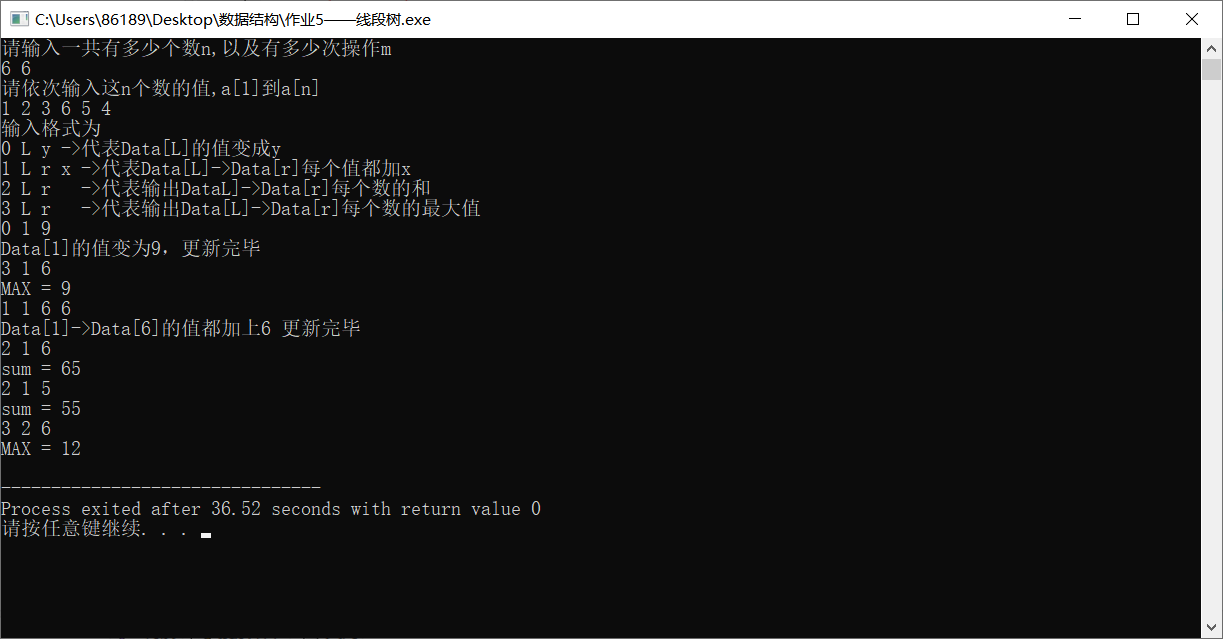
Lazy标记用于采用延迟更新策略

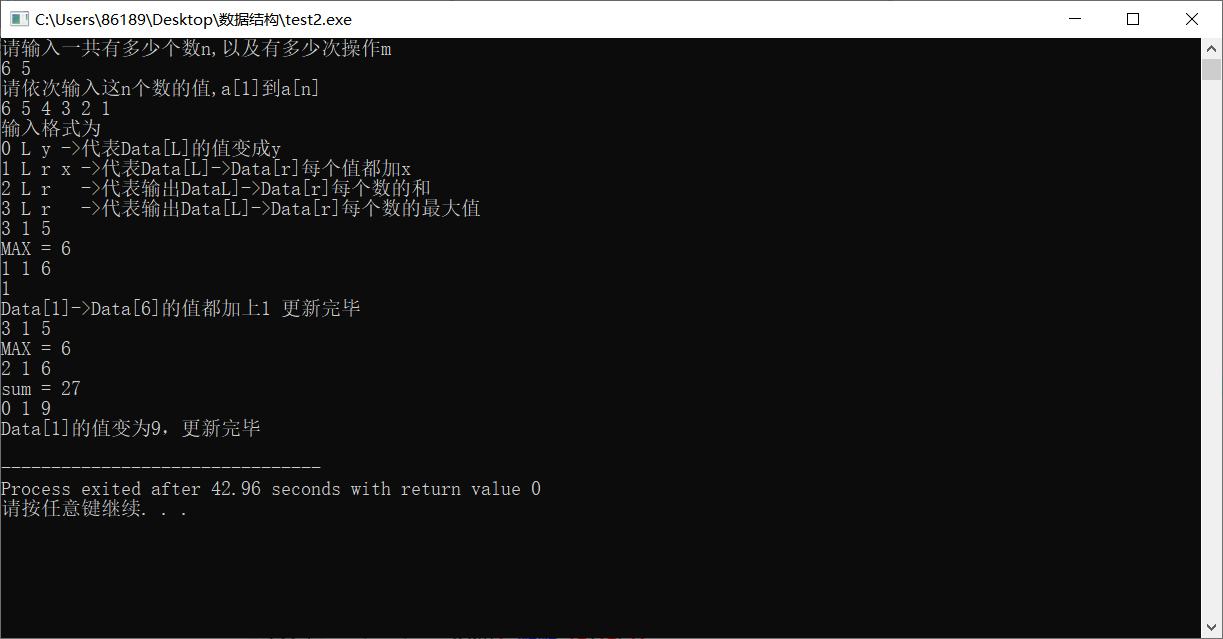
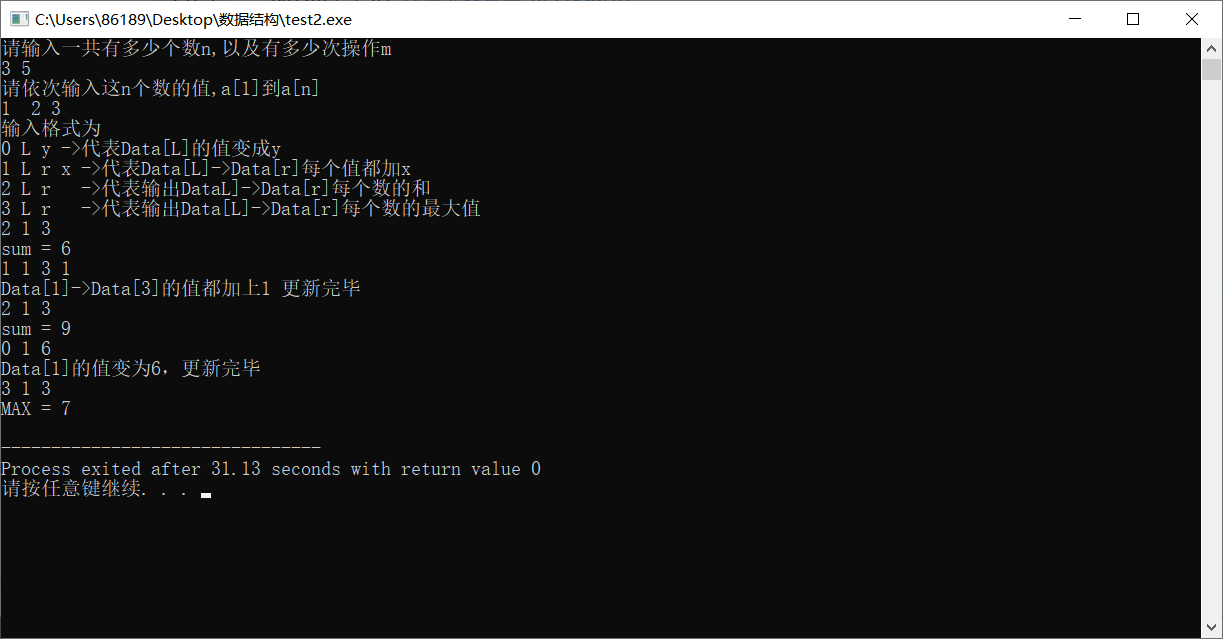
Max代表的是这个区间的所有值的最大值

ll Data[MAXN];，数组Data代表这每一个区间的一块地点存储的值

### 建立线段树

1. 如果l == r情况则直接将SUM和Max 赋值为a[l]
2. 如果l！=r情况，则递归建立线段树
   1. 定义Mid = t[p].l + t[p].r >> 1;
   2. 递归调用build(lson(p),l,mid);处理左子树
   3. 递归调用build(rson(p),mid + 1,r);处理右子树
   4. 回溯后将子节点信息保存下来t[p].SUM = t[lson(p)].SUM + t[rson(p)].SUM；t[p].Max = max(t[lson(p)].Max,t[rson(p)].Max);
3. 区间更新线段树update函数
   1. 如果区间被覆盖的情况
      1. 直接进行修改
      2. t[p].SUM += value \* (t[p].r - t[p].l + 1);
      3. t[p].Max += value;
      4. t[p].lazy += value;
      5. Return结束
   2. 如果区间没有被覆盖的情况
      1. 执行push\_down（p）函数，执行向下更新左右儿子结点的数据
         1. 当lazy不为零的时候，将左右儿子的值利用lazy修改左右儿子的SUM和Max
         2. 将当前p的lazy下传给左右儿子的lazy
         3. 当前p的lazy清零
      2. 定义mid = t[p].l + t[p].r >> 1便于后续判断
         1. 如果覆盖了左儿子就修改左儿子update(lson(p),l,r,value)
         2. 如果覆盖了右儿子就修改右儿子update(rson(p),l,r,value)
      3. 更新当前p结点的Sum和Max数据
4. 单个数值修改线段树（数组A中某一个元素值更新时）
   1. 令Data[L]的值变成y，利用区间更新函数等价于将Data[L]->Data[L]的值都加入value =y-Data[L]
   2. update(1,l,l,value);
5. 查询区间和querySum函数
   1. 如果区间被覆盖
      1. 直接返回区间的数据Returnt [p].SUM
   2. 如果区间没有被覆盖
      1. 执行push\_down（P），执行向下更新左右儿子结点的数据
         1. 当lazy不为零的时候，将左右儿子的值利用lazy修改左右儿子的SUM和Max
         2. 将当前p的lazy下传给左右儿子的lazy
         3. 当前p的lazy清零
      2. 定义mid = t[p].l + t[p].r >> 1;方便后续判断
      3. 如果(l <= mid)需要查询左儿子，就加入整理左儿子的数据sum += querySum(lson(p),l,r)
      4. 如果(r > mid)，如要查询右儿子，就加入整理右儿子的数据sum += querySum(rson(p),l,r);
      5. 返回return sum
6. 查询区间最大值query\_Max函数
   1. 如果区间被覆盖
      1. 直接return t[p].Max
   2. 如果区间没有被覆盖
      1. 定义mid = t[p].l + t[p].r >> 1;方便后续判定
      2. 定义maxL = -inf,maxR = -inf;
      3. 如果(l <= mid)需要查询左儿子，则查左儿子的最大值maxL = max(maxL,query\_Max(lson(p),l,r));
      4. 如果(r > mid)需要查询右儿子，则查询右儿子最大值maxR = max(maxR,query\_Max(rson(p),l,r));
   3. 返回 左右子树的最大值

测试样例：



# 具体代码

#include<bits/stdc++.h>

using namespace std;

typedef long long ll;

#define mem(a,x) memset(a,x,sizeof(a))

#define IOS ios\_base::sync\_with\_stdio(false);cin.tie(NULL);cout.tie(NULL);

const double PI = acos(-1.0);

const ll MAXN = 2e5 + 10;

const ll mod = 998244353;

const ll inf = 1e18;

const ll mo = 1e9+7;

int n,m,mx;

ll Data[MAXN];//代表这每一个区间的一块地点存储的值

struct TREE

{

int l,r;//代表节点维护的区间范围;代表了a[l]->a[r]

ll SUM; ////!!!!!!!!!sum总和

ll lazy; //涉及lazy标记的东西;

ll Max;//最大值

}t[MAXN << 2];

inline int lson(int p){return p << 1;}//左儿子;

inline int rson(int p){return p << 1 | 1;}//右儿子;

void build(int p,int l,int r)//建线段树

{

t[p].l = l,t[p].r = r;

if(l == r)//叶子节点

{

t[p].SUM = Data[l];

t[p].Max = Data[l];//

return;

}

int mid = t[p].l + t[p].r >> 1;

build(lson(p),l,mid);

build(rson(p),mid + 1,r);

t[p].SUM = t[lson(p)].SUM + t[rson(p)].SUM;

t[p].Max = max(t[lson(p)].Max,t[rson(p)].Max);//递归建树!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

}

void push\_down(int p)

{

if(t[p].lazy)

{

//如果lazy标记不为0

t[lson(p)].SUM += t[p].lazy \* (t[lson(p)].r - t[lson(p)].l + 1);

t[rson(p)].SUM += t[p].lazy \* (t[rson(p)].r - t[rson(p)].l + 1);

t[lson(p)].Max += t[p].lazy;//!!!!

t[rson(p)].Max += t[p].lazy;//!!!!

//下传;

t[lson(p)].lazy += t[p].lazy;

t[rson(p)].lazy += t[p].lazy;

t[p].lazy = 0;//下传完成,更新lazy为0;

}

}

void update(int p,int l,int r,ll value)//将区间 [x, y][x,y] 内每个数加上 k

{

if(l <= t[p].l && r >= t[p].r)//1.区间被覆盖,就修改;

{

t[p].SUM += value \* (t[p].r - t[p].l + 1);

t[p].Max += value;

t[p].lazy += value;

return;

}

push\_down(p);

int mid = t[p].l + t[p].r >> 1;

if(l <= mid)

update(lson(p),l,r,value);//覆盖了左儿子就修改左儿子;

if(r > mid)

update(rson(p),l,r,value);

t[p].SUM = t[lson(p)].SUM + t[rson(p)].SUM;

t[p].Max = max(t[lson(p)].Max,t[rson(p)].Max);//!!!!!!!!!!!!!!

}

ll querySum(int p,int l,int r)//输出区间 [x, y][x,y] 内每个数的和

{

if(l <= t[p].l && r >= t[p].r)

return t[p].SUM;//覆盖了该区间就直接返回整个数据;

push\_down(p);

ll sum = 0;

int mid = t[p].l + t[p].r >> 1;

if(l <= mid)

sum += querySum(lson(p),l,r);

if(r > mid)

sum += querySum(rson(p),l,r);

return sum;

}

ll query\_Max(int p,int l,int r)

{

if(l <= t[p].l && r >= t[p].r)

return t[p].Max;

int mid = t[p].l + t[p].r >> 1;

ll maxL = -inf,maxR = -inf;

if(l <= mid)maxL = max(maxL,query\_Max(lson(p),l,r));

if(r > mid)maxR = max(maxR,query\_Max(rson(p),l,r));

return max(maxL,maxR);

}

void update\_point(int l,ll y)//l y ->代表a[l]的值变成y\n

{//等将于 将区间[l,l]内每个数加上 value = y-a[l]

ll value = y- Data[l];

update(1,l,l,value);

}

int main()

{

printf("请输入一共有多少个数n,以及有多少次操作m\n");

scanf("%d%d",&n,&m);

printf("请依次输入这n个数的值,a[1]到a[n]\n");

for(int i = 1;i <= n;i ++)

scanf("%lld",&Data[i]);

build(1,1,n);//建立线段树 a[1]->a[n]

//执行操作

printf("输入格式为\n");

printf("0 L y ->代表Data[L]的值变成y\n");

printf("1 L r x ->代表Data[L]->Data[r]每个值都加x\n");

printf("2 L r ->代表输出DataL]->Data[r]每个数的和\n");

printf("3 L r ->代表输出Data[L]->Data[r]每个数的最大值\n");

int l,r;

ll y;

ll value;

for(int i = 1;i <= m;i ++)

{

int oporate;

scanf("%d",&oporate);

if(oporate == 0)

{

scanf("%d",&l);

scanf("%lld",&y);

value = y-Data[l];

update(1,l,l,value);

printf("Data[%d]的值变为%lld，更新完毕\n",l,y);

}

else if(oporate == 1)

{

scanf("%d%d",&l,&r);

scanf("%lld",&value);

update(1,l,r,value);

printf("Data[%d]->Data[%d]的值都加上%lld 更新完毕\n",l,r,value);

}

else if(oporate ==2)

{

scanf("%d%d",&l,&r);

printf("sum = %lld\n",querySum(1,l,r));

}

else

{

scanf("%d%d",&l,&r);

mx = query\_Max(1,l,r);//输入l,r，找出a[l]到a[r]的最大值

printf("MAX = %lld\n",mx);//出最大值

}

}

}