

数据结构--线段树 (segment)



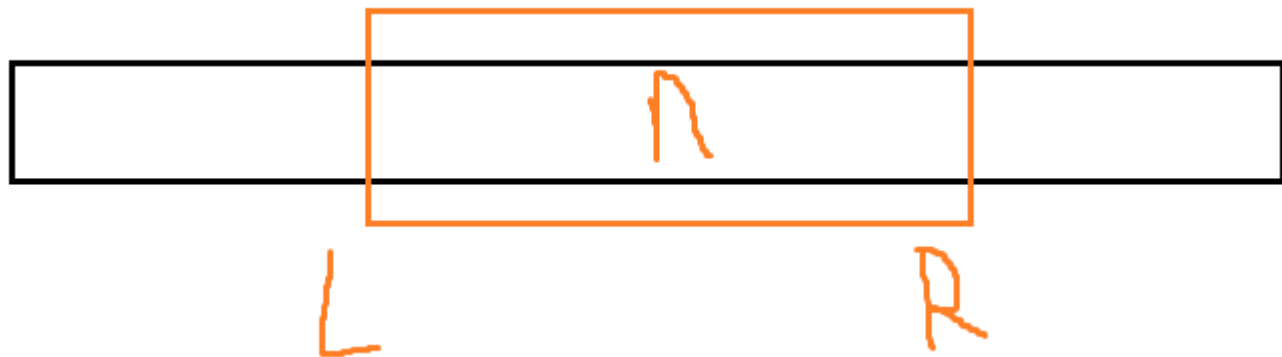
引入

给定一个数组，数组长度可能非常大。现在我们需要对数组里面的数据反反复复进行两个操作

- 求出某一个区间里面所有元素之和，(query操作)
- 修改某个元素的值，(update操作)

暴力解决

对区间 $[L,R]$ （长度为 n ）取和，并且更新一个元素 i 的值，采用暴力解决方法



可得

- $\text{query}(L,R)$ 时间复杂度为 $O(n)$
- $\text{update}(i)$ 时间复杂度为 $O(1)$

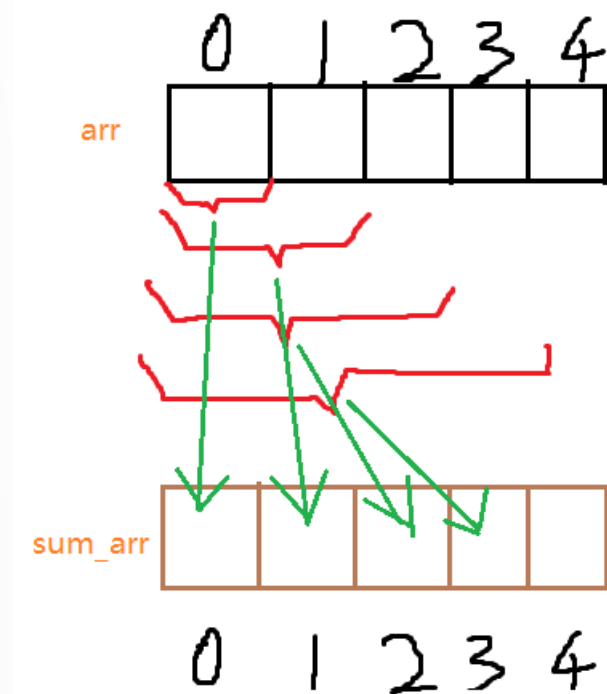


如果区间范围很大，再加上多次操作，暴力取和明显会超时，可以采用前缀和方式优化查询

$\text{sum_arr}[0] = \text{arr}[0]$

$\text{sum_arr}[1] = \text{arr}[0] + \text{arr}[1]$

$\text{sum_arr}[2] = \text{arr}[0] + \text{arr}[1] + \text{arr}[2]$



这样，假如我们想得到区间[2,4]的和，我们可以用 $\text{sum_arr}[4] - \text{sum_arr}[1]$ 计算到

- $\text{query}(L,R)$ 时间复杂度减小为 $O(1)$
- 因为改变一个值后，要同时更新后面的`sum_arr`数组，所以 $\text{update}(i)$ 时间复杂度增大为 $O(n)$

如果用线段树的话，我们可以将查询和更新的时间复杂度都变为 $O(\log n)$

| 方法 | query | update |
|-----|-------------|-------------|
| 暴力 | $O(n)$ | $O(1)$ |
| 前缀和 | $O(1)$ | $O(n)$ |
| 线段树 | $O(\log n)$ | $O(\log n)$ |



线段树简介

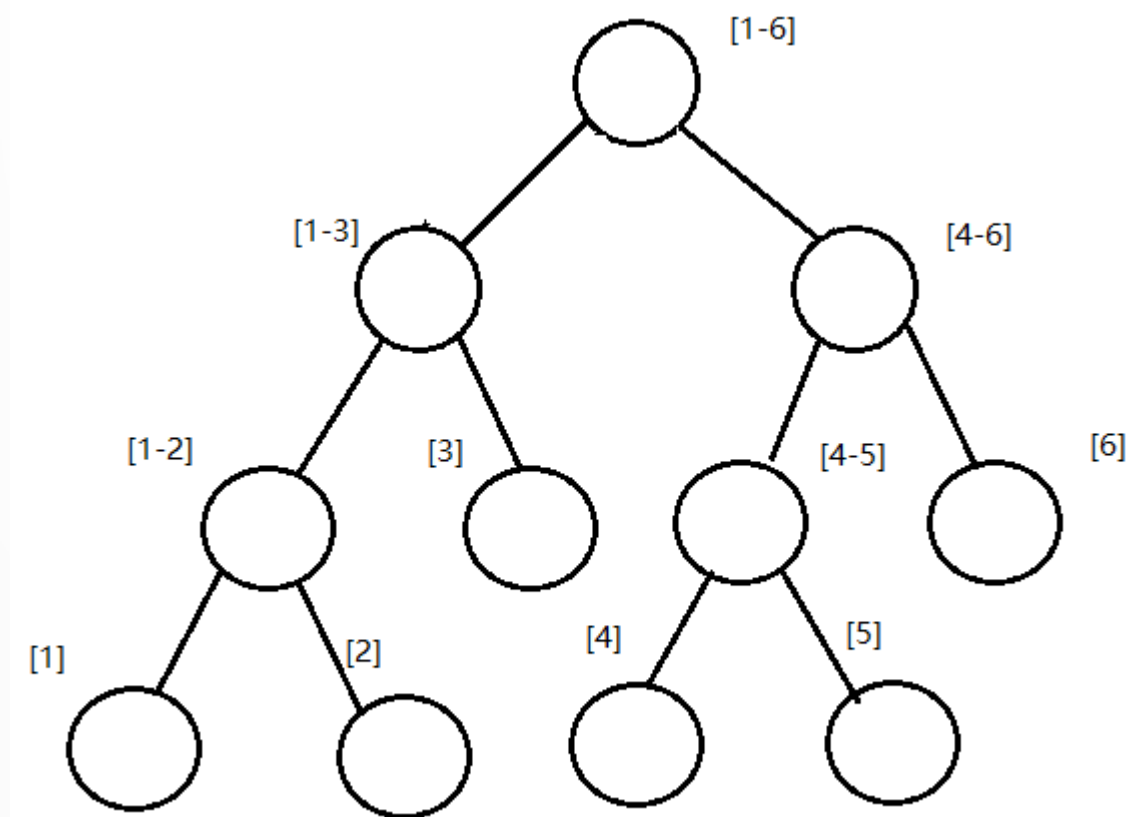
线段树是一种二叉搜索树，它将一个区间划分成一些单元区间，每个单元区间对应线段树中的一个叶结点。使用线段树可以快速的查找某一个节点在若干条线段中出现的次数，时间复杂度为 $O(\log n)$ 。是一种可以在很短的时间内对某个区间进行操作的数据结构。

可用于：

单点修改、区间修改、区间查询（如：区间求和，求区间最大值，求区间最小值.....）



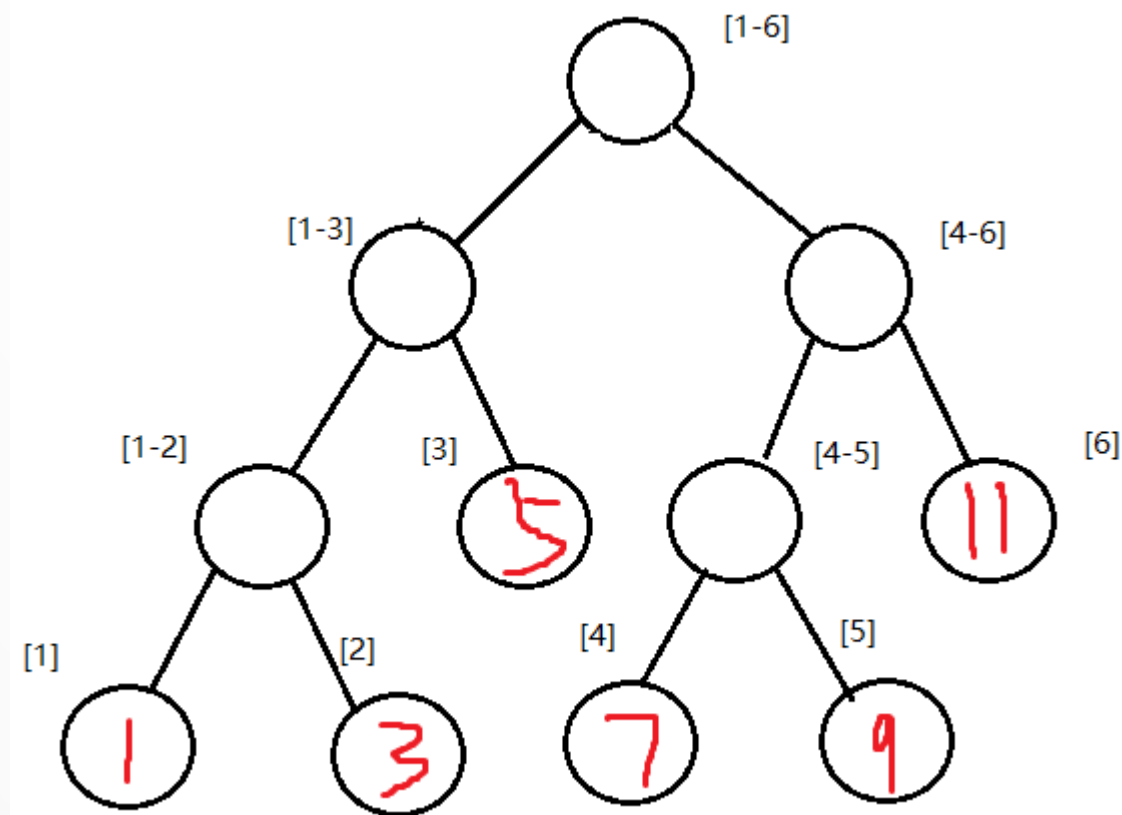
线段树构建



线段树构建

然后给每个叶子结点赋值

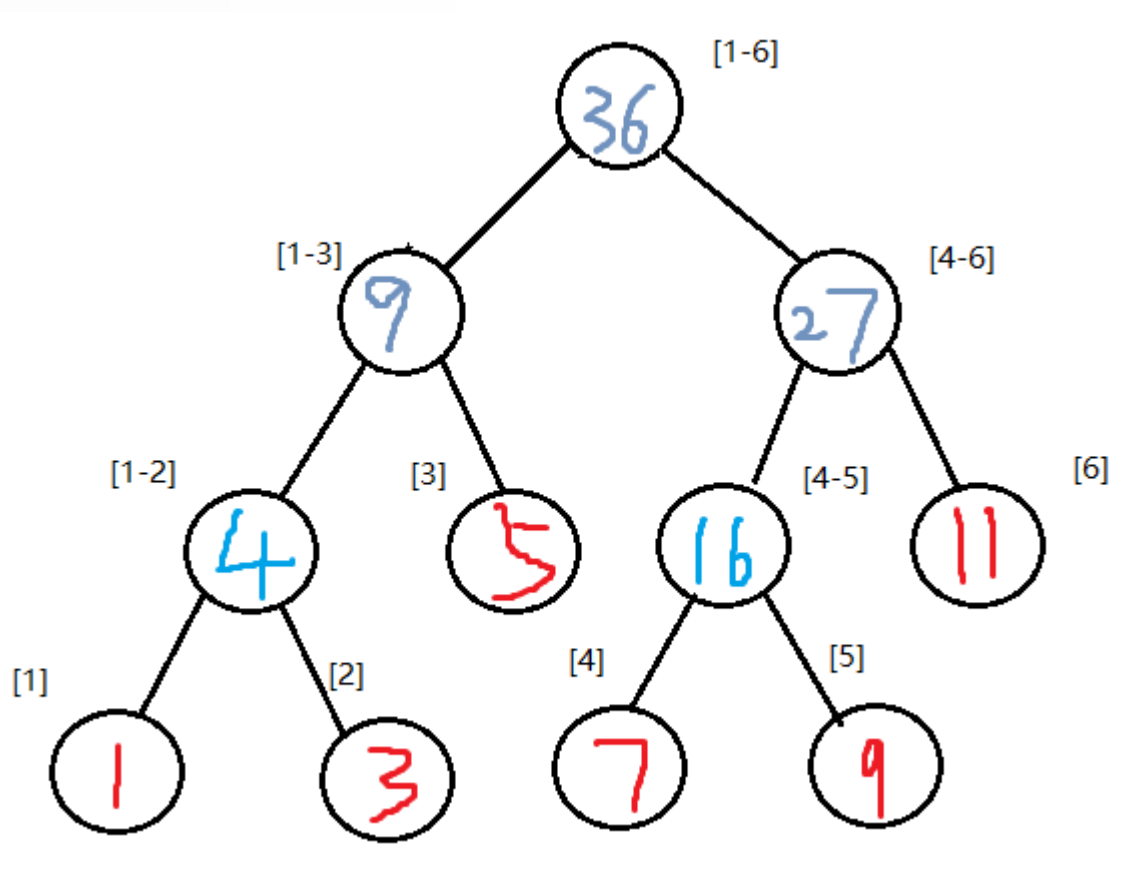
| 数组下标 | 1 | 2 | 3 | 4 | 5 | 6 |
|------|---|---|---|---|---|----|
| 数组元素 | 1 | 3 | 5 | 7 | 9 | 11 |



线段树构建

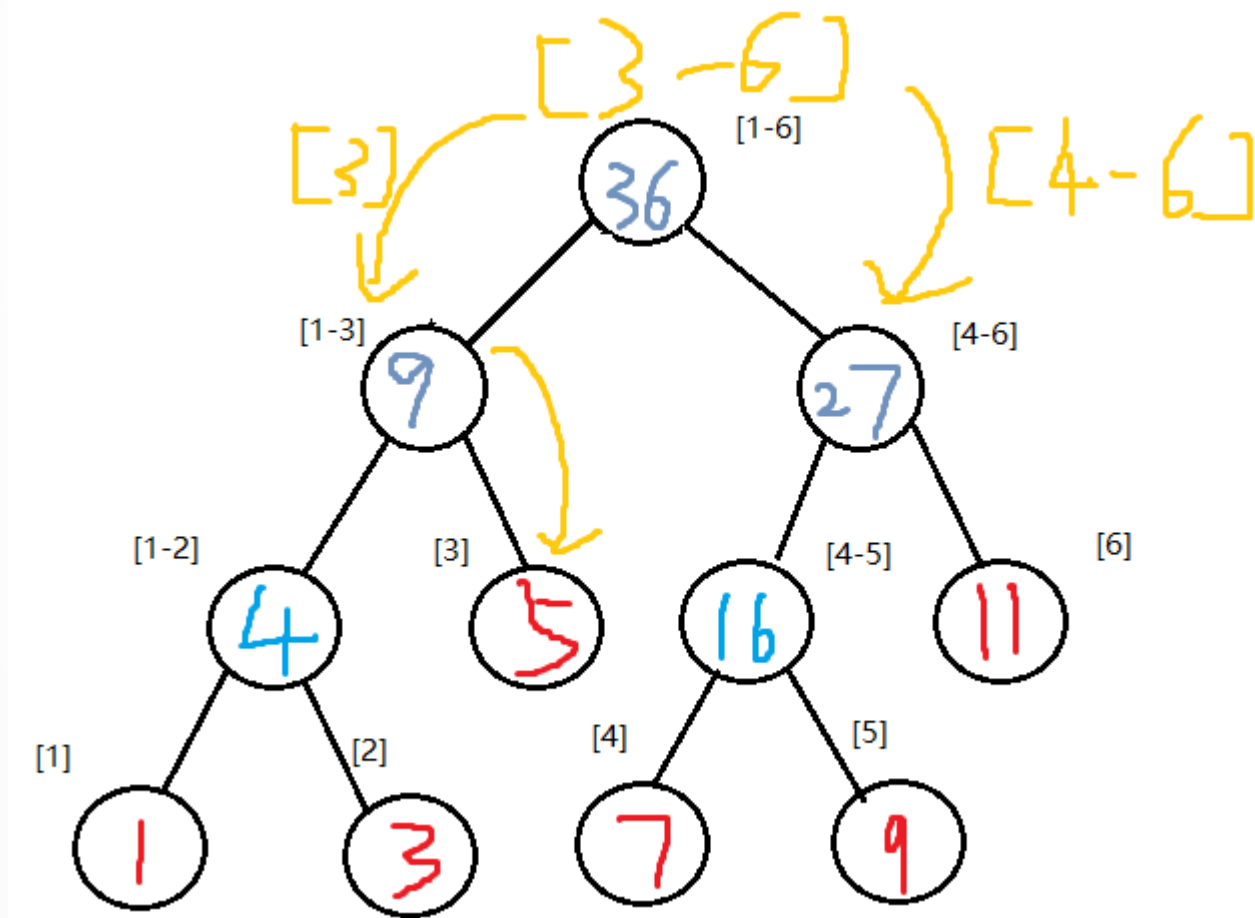
接着由孩子结点构成双亲结点

| | | | | | | |
|------|---|---|---|---|---|----|
| 数组下标 | 1 | 2 | 3 | 4 | 5 | 6 |
| 数组元素 | 1 | 3 | 5 | 7 | 9 | 11 |



回到原题

如何找到[2-5]这个区间的和

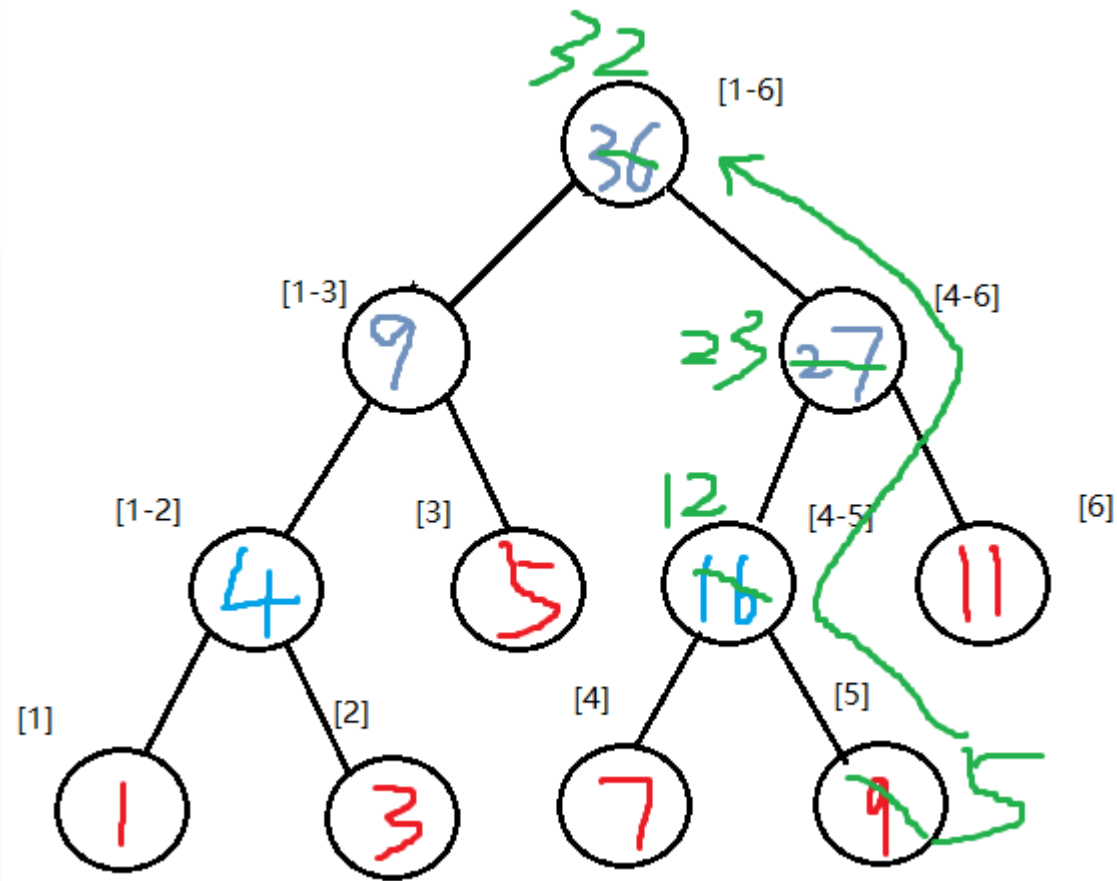


根节点记录的是[1-6]的和，可以把[3-6]分成两半，左边找的是[3],右边是[4-6]，右边可以直接得到[4-6]的和为27，而[3]可以通过[1-3]得到和为5，最终结果为 $27+5=32$ 。

这样子可以省掉很多搜索的时间，最坏情况是把整棵树都搜索一遍，时间复杂度为 $O(\log n)$

回到原题

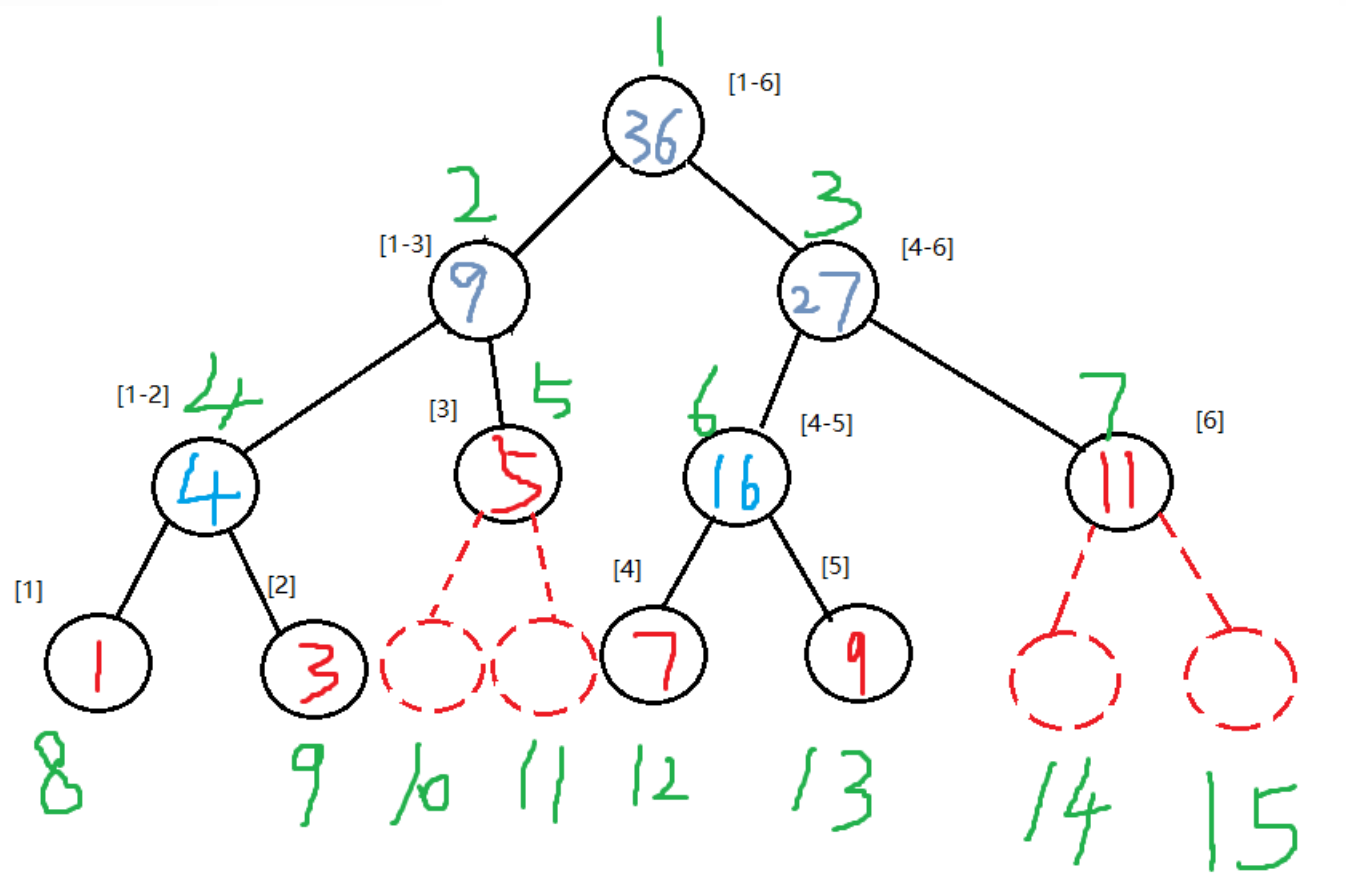
那么如何更新呢，假如我们想把第5个元素由9变成6



我们先找到9那个结点，然后把其值变为6，接着顺着一条路从下往上一直更新。
同样，更新的时间复杂度也是 $O(\log n)$

线段树的实现

由于线段树是用二叉树结构储存的，而且是近乎完全二叉树的，所以我使用了数组tree来存储
数组下标从1开始，同时添加虚结点使其变成完全二叉树



| 数组下标 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|------|----|---|----|---|---|----|----|---|---|----|----|----|----|----|----|
| 数组元素 | 36 | 9 | 27 | 4 | 5 | 16 | 11 | 1 | 3 | X | X | 7 | 9 | X | X |



线段树的实现

树的结点的定义

```
struct node
{
    int value; // 节点对应区间的权值（不唯一，也可以代表区间最大值等）
    int left, right; // 区间[left, right]
};
struct node tree[1000];
int father[100]; // 记录某个点的序号，方便查找对应的数组下标
```

建树

```
Build_tree(1, 1, 6);  
//为区间[left, right]建立一个以top为祖先的线段树, top为根节点下标  
void Build_tree(int top, int left, int right)  
{  
    tree[top].left = left; //写入第index个结点的左区间  
    tree[top].right = right; //写入第index个结点的右区间  
    tree[top].value = 0; //每个区间的值初始化为0  
  
    if(left == right) //区间长度为0时, 赋值并且结束递归  
    {  
        tree[top].value = arr[left];  
        father[left] = top;  
        return;  
    }  
    int mid = (right + left) / 2; //取区间中点  
    int left_node = top * 2; //左孩子下标  
    int right_node = top * 2 + 1; //右孩子下标  
    Build_tree(left_node, left, mid); //往左孩子方向继续建立线段树  
    Build_tree(right_node, mid + 1, right); //往右孩子方向继续建立线段树  
    //更新结点值为左右孩子的和  
    tree[top].value = tree[left_node].value + tree[right_node].value;  
}
```

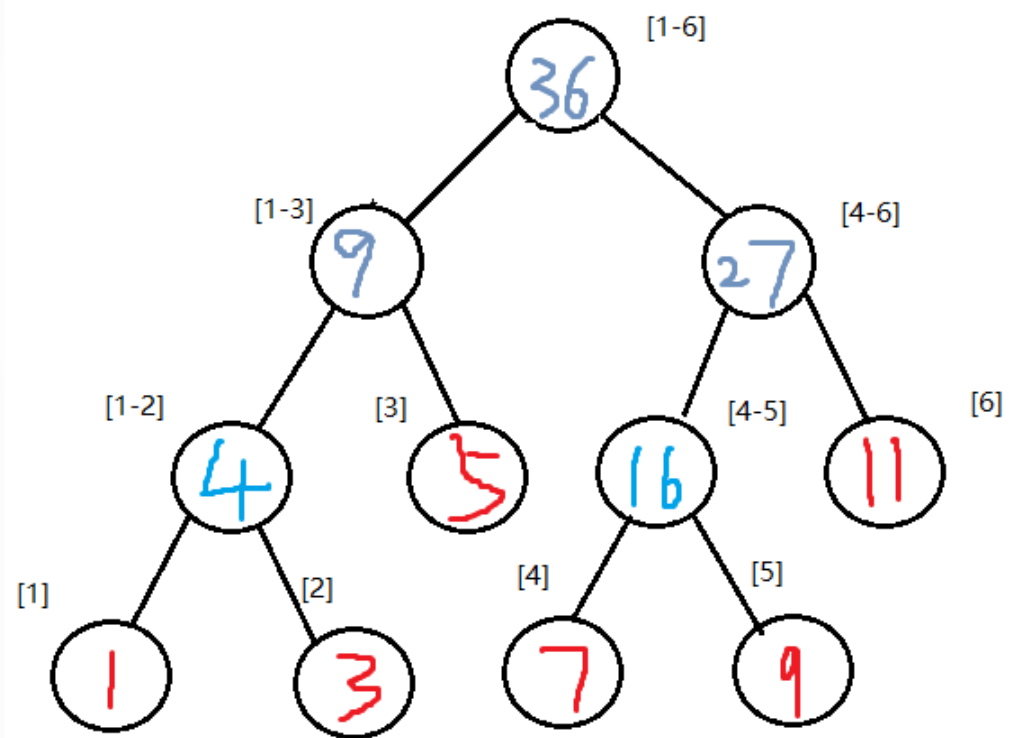
更新

更新数组的第5个元素的值为6，直接在树里面更新该结点的值后，然后从父结点往上更新，直到更新到了根结点。

```
int main()
{
    tree[father[5]].value = 6;
    Update(father[5]);
}

void Update(int index) //index为要修改那个点的数组下标
{
    int father_node = index / 2; //父结点下标
    int left_node = father_node * 2; //左孩子下标
    int right_node = father_node * 2 + 1; //右孩子下标
    tree[father_node].value = tree[left_node].value + tree[right_node].value; //更新值
    if(father_node == 1) //找到树的根结点，终止退出
        return;
    Update(father_node); //递归更新，由父结点往上找
}
```

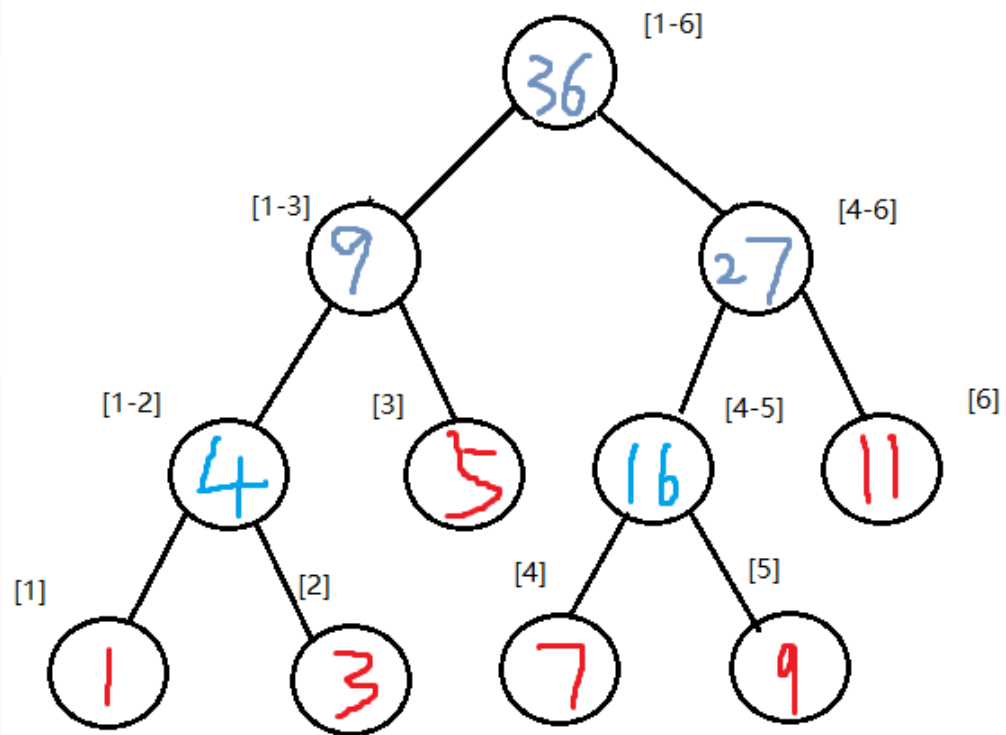

查询[L,R]区间



先查询左子树：

1. 如果满足条件 $3 \geq L$, 则要查询的区间有涉及左子树, 例如查询 $[1,2]$, $[2,4]$
 - 如果满足 $3 \geq R$, 则要查询的区间完全在左子树, 例如 $[1,2]$, 这时候要查询的区间不变
 - 如果不满足 $3 \geq R$, 则要查询的区间不完全在左子树, 例如 $[2,4]$, 这时候就需要查询在左子树那边的区间, 所以要查询 $[2,3]$

查询[L,R]区间



同理再查询左子树：

1. 如果满足条件 $R \geq 4$, 则要查询的区间有涉及右子树, 例如查询 $[4,6]$, $[2,5]$
 - 如果满足 $L \geq 4$, 则要查询的区间完全在右子树, 例如 $[4,6]$, 这时候要查询的区间不变
 - 如果不满足 $L \geq 4$, 则要查询的区间不完全在右子树, 例如 $[2,5]$, 这时候就需要查询在右子树那边的区间, 所以要查询 $[4,5]$

查询

```
//从index开始查询, 所以index一般为树的根结点, 查询的区间是[L,R], 结果保留在ans里面
void Query(int index, int L, int R, int& ans){
    if(tree[index].left == L && tree[index].right == R){ //找到了一个完全重合的区间
        ans += tree[index].value;
        return;
    }
    int left_node = index*2;
    if(L <= tree[left_node].right){ //左区间有涉及
        if(R <= tree[left_node].right) //全包含于左区间, 查询区间不变
            Query(left_node, L, R, ans);
        else //半包含于左区间, 则查询区间拆分, 左端点不变, 右端点变为左孩子的右区间端点
            Query(left_node, L, tree[left_node].right, ans);
    }
    int right_node = left_node + 1;
    if(R >= tree[right_node].left){ //右区间有涉及
        if(L >= tree[right_node].left) //全包含于右区间, 查询区间不变
            Query(right_node, L, R, ans);
        else //半包含于左区间, 则查询区间拆分, 与上同理
            Query(right_node, tree[right_node].left, R, ans);
    }
}
```

```

1 #include <iostream>
2 using namespace std;
3
4 struct node{
5     int value; //节点对应区间的权值 (不唯一, 也可以代表区间最大值等)
6     int left, right; //区间[left, right]
7 };
8 struct node tree[1000];
9 int father[100]; //记录某个点的序号, 方便查找对应的数组下标
10 int arr[] = {-1, 1, 3, 5, 7, 9, 11};
11
12 //为区间[left, right]建立一个以top为祖先的线段树, top为根节点下标
13 void Build_tree(int top, int left, int right){
14
15     tree[top].left = left; //写入第index个结点的左区间
16     tree[top].right = right; //写入第index个结点的右区间
17     tree[top].value = 0; //每个区间的值初始化为0
18
19     if(left == right){ //区间长度为0时, 赋值并且结束递归
20
21         tree[top].value = arr[left];
22         father[left] = top;
23         return;
24     }
25     int mid = (right + left) / 2; //取区间中点
26     int left_node = top * 2; //左孩子下标
27     int right_node = top * 2 + 1; //右孩子下标
28     Build_tree(left_node, left, mid); //往左孩子方向继续建立线段树
29     Build_tree(right_node, mid + 1, right); //往右孩子方向继续建立线段树
30     tree[top].value = tree[left_node].value + tree[right_node].value; //更新结点值为左右
31 }
32
33 void Update(int index){ //index为要修改那个点的数组下标
34     int father_node = index / 2; //父结点下标
35     int left_node = father_node * 2; //左孩子下标
36     int right_node = father_node * 2 + 1; //右孩子下标
37     tree[father_node].value = tree[left_node].value + tree[right_node].value; //更新值
38     if(father_node == 1) //找到树的根结点, 终止退出
39         return;
40     Update(father_node); //递归更新, 由父结点往上找
41 }

```

```

1 void Query(int index, int L, int R, int& ans){
2     if(tree[index].left == L && tree[index].right == R){ //找到了一个完全重合的区间
3         ans += tree[index].value;
4         return;
5     }
6     int left_node = index * 2;
7     if(L <= tree[left_node].right){ //左区间有涉及
8
9         if(R <= tree[left_node].right) //全包含于左区间, 查询区间不变
10             Query(left_node, L, R, ans);
11         else //半包含于左区间, 则查询区间拆分, 左端点不变, 右端点变为左孩子的右区间端点
12             Query(left_node, L, tree[left_node].right, ans);
13     }
14     int right_node = left_node + 1;
15     if(R >= tree[right_node].left){ //右区间有涉及
16         if(L >= tree[right_node].left) //全包含于右区间, 查询区间不变
17             Query(right_node, L, R, ans);
18         else //半包含于右区间, 则查询区间拆分, 与上同理
19             Query(right_node, tree[right_node].left, R, ans);
20     }
21 }
22 int main(){
23     Build_tree(1, 1, 6); //建树
24     tree[father[5]].value = 6; //修改第5个元素的值为6
25     Update(father[5]);
26     int ans = 0;
27     Query(1, 3, 6, ans); //查询区间[3-6]的值
28     cout << ans << endl;
29     return 0;
30 }

```

模板题HDU1754

I Hate It

Time Limit: 9000/3000 MS (Java/Others) Memory Limit: 32768/32768 K (Java/Others)
Total Submission(s): 114462 Accepted Submission(s): 42612

Problem Description

很多学校流行一种比较的习惯。老师们很喜欢询问，从某某到某某当中，分数最高的是多少。
这让很多学生很反感。

不管你喜不喜欢，现在需要你做的是，就是按照老师的要求，写一个程序，模拟老师的询问。当然，老师有时候需要更新某位同学的成绩。

Input

本题目包含多组测试，请处理到文件结束。

在每个测试的第一行，有两个正整数 N 和 M ($0 < N \leq 200000, 0 < M \leq 5000$)，分别代表学生的数目和操作的数目。

学生ID编号分别从1编到 N 。

第二行包含 N 个整数，代表这 N 个学生的初始成绩，其中第 i 个数代表ID为 i 的学生的成绩。

接下来有 M 行。每一行有一个字符 C (只取'Q'或'U')，和两个正整数 A, B 。

当 C 为'Q'的时候，表示这是一条询问操作，它询问ID从 A 到 B (包括 A, B)的学生当中，成绩最高的是多少。

当 C 为'U'的时候，表示这是一条更新操作，要求把ID为 A 的学生的成绩更改为 B 。

Output

对于每一次询问操作，在一行里面输出最高成绩。

Sample Input

```
5 6
1 2 3 4 5
Q 1 5
U 3 6
Q 3 4
Q 4 5
U 2 9
Q 1 5
```

Sample Output

```
5
6
5
9
```

Hint

Huge input,the C function scanf() will work better than cin

936MS 8336K（注意，线段树需要空间比较大，数组开小可能会wa或者TLE，一般为 $4n$ ）

```
1  const int MAXNODE = 2e5*4;
2  struct node{
3      int value;
4      int left, right;
5  }tree[MAXNODE];
6  int father[MAXNODE];
7
8  void Build_tree(int i, int left, int right){
9      tree[i].left = left;
10     tree[i].right = right;
11     tree[i].value = 0;
12     if(left == right){
13         father[left] = i;
14         return;
15     }
16     Build_tree(i*2, left, (left+right)/2);
17     Build_tree(i*2+1, (left+right)/2+1, right);
18 }
19
20 int MAX;
21 void Query(int i, int L, int R){
22     if(tree[i].left == L && tree[i].right == R){
23         MAX = max(MAX, tree[i].value);
24         return;
25     }
26     i *= 2;
27     if(L <= tree[i].right){
28         if(R <= tree[i].right)
29             Query(i, L, R);
30         else
31             Query(i, L, tree[i].right);
32     }
33     i++;
34     if(R >= tree[i].left){
35         if(L >= tree[i].left)
36             Query(i, L, R);
37         else
38             Query(i, tree[i].left, R);
39     }
40 }
```

```
1  void Update(int ch){
2      if(ch == 1) return;
3      int fa = ch / 2;
4      int num1 = tree[fa * 2].value;
5      int num2 = tree[fa * 2 + 1].value;
6      tree[fa].value = max(num1, num2);
7      Update(fa);
8  }
9  int main(){
10     int n, m, num, a, b;
11     char op;
12     while(scanf("%d %d", &n, &m) != EOF){
13         Build_tree(1, 1, n);
14         for(int i = 1; i <= n; i++){
15             scanf("%d", &num);
16             tree[father[i]].value = num;
17             Update(father[i]);
18         }
19         while(m--){
20             getchar();
21             scanf("%c %d %d", &op, &a, &b);
22             if(op == 'Q'){
23                 MAX = 0;
24                 Query(1, a, b);
25                 printf("%d\n", MAX);
26             }
27             else{
28                 tree[father[a]].value = b;
29                 Update(father[a]);
30             }
31         }
32     }
33     return 0;
34 }
```

线段树部分优化

1. $a*2$ 可以用 $a \ll 1$ 代替, $a/2$ 可以用 $a \gg 1$ 代替。(位运算其实就是直接对在内存中的二进制数据进行操作, 因此处理数据的速度非常快)
2. 因为下标为 a 的节点的左儿子下标为 $a*2$, 右儿子下标为 $a*2+1$, 所以可以

```
//加入一些编译预处理指令可以提高编程效率, 加快编译速度
#define LS(a) (a << 1)
// a<<1 等同于 a*2
#define RS(a) (a << 1 | 1)
// a<<1|1 等同于 a*2+1
```

线段树进阶

1. Lazy标记--用于同时更新一段区间的值: [poj3468 A Simple Problem with Integers](#)
2. 线段树离散化（节约空间）--[poj2528 Mayor's posters](#)
3. 线段树应用：扫描线问题（求多个矩形互相覆盖后的面积）：[poj1151 Atlantis](#)
4. 可持久化（保留整个操作的历史）--主席树