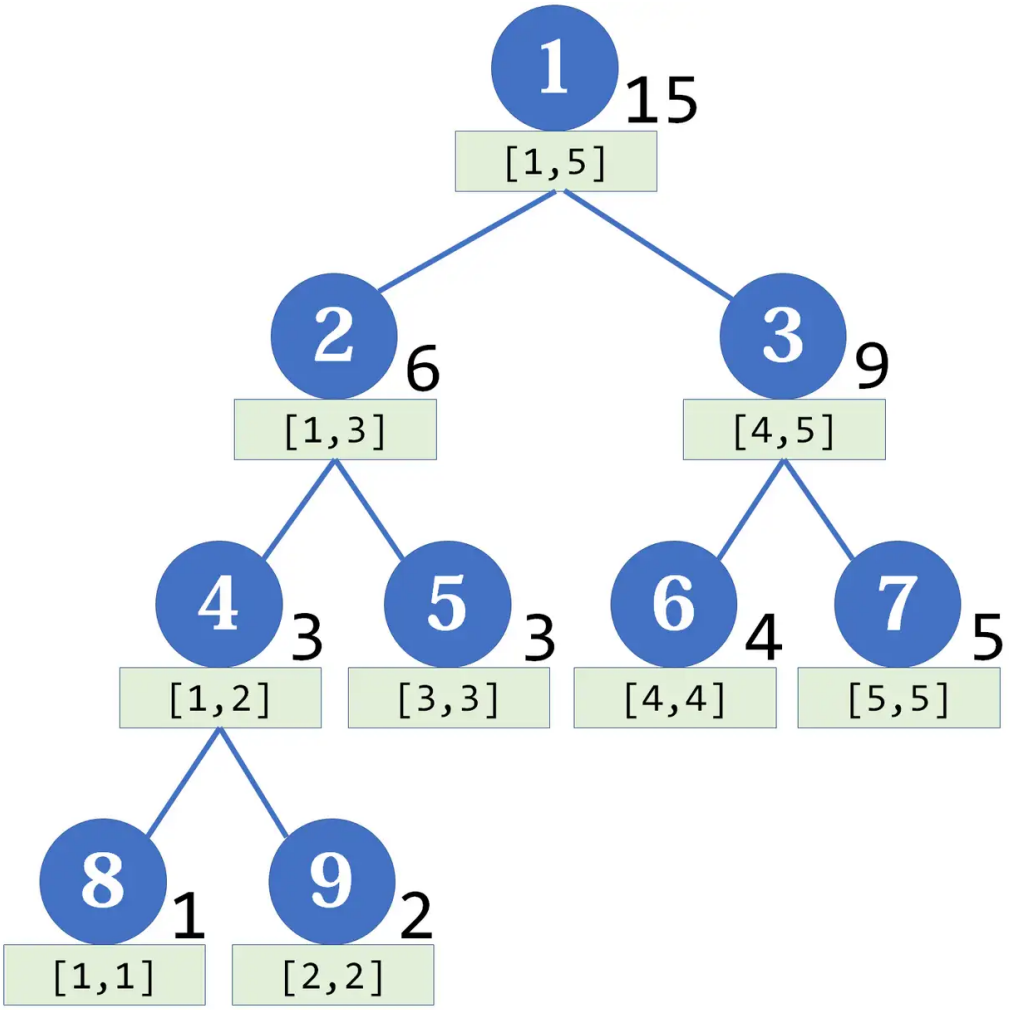
线段树

线段树（Segment Tree）几乎是算法竞赛最常用的数据结构了，它主要用于维护**区间信息**（要求满足结合律）。与树状数组相比，它可以实现 $O(\log n)$ 的**区间修改**，还可以同时支持**多种操作**（加、乘），更具通用性。

线段树的建立

线段树是一棵**平衡二叉树**。母结点代表整个区间的和，越往下区间越小。注意，线段树的每个**节点**都对应一条**线段（区间）**，但并不保证所有的线段（区间）都是线段树的节点，这两者应当区分开。

如果有一个数组 $[1, 2, 3, 4, 5]$，那么它对应的线段树大概长这个样子：



每个节点 p 的左右子节点的编号分别为 $2p$ 和 $2p+1$，假如节点 p 储存区间 $[a, b]$ 的和，设 $mid = \lfloor \frac{l+r}{2} \rfloor$，那么两个子节点分别储存 $[l, mid]$ 和 $[mid+1, r]$ 的和。可以发现，左节点对应的区间长度，与右节点**相同**或者比之**恰好多** 1。

如何从数组建立一棵线段树？我们可以考虑**递归**地进行。

```
// 根节点
constexpr int rt = 1;

struct Node {
    T sum;
```

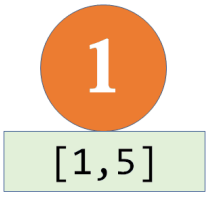
```
};

// 合并两棵子树信息的函数
Node merge(const Node& a, const Node& b) {
    Node ret {a.sum + b.sum};
    return ret;
}

// 叶子节点如何初始化
Node init(T x) {
    return {x};
}

// lc - 左子节点，rc - 右子节点，p - 当前节点
// cl，cr - 当前节点对应区间左、右端点
void build(const vector<T>& a, int p, int cl = 0, int cr = n - 1) {
    if (cl == cr) {t[p] = init(a[cl]); return;}
    auto lc = p << 1, rc =lc + 1, mid = (cl + cr) >> 1;
    build(a, lc, cl, mid); build(a, rc, mid + 1, cr);
    t[p] = merge(t[lc], t[rc]);
}
```

下面的动图生动形象地说明了我们的线段树是怎么工作的：



单点修改

单点修改比较容易实现，从**根节点**沿着我们的线段树一路往下，如果当前的线段**不包含**我们待修改的点，则直接退出，否则进行相应的修改

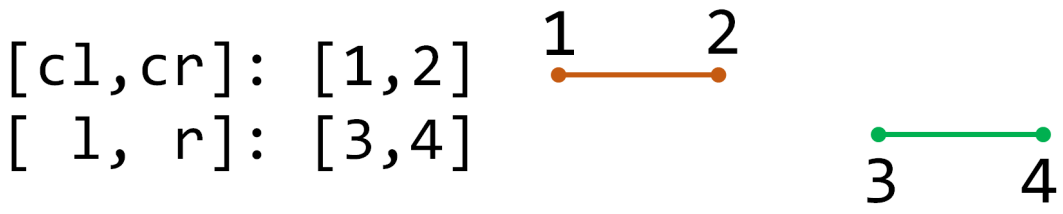
以**单点重新赋值**为例：

```
void modify(int pos, T v, int p = rt, int cl = 0, int cr = n - 1) {
    // 不含待修改的区间
    if (pos > cr || pos < cl) return;
    // 就是待修改的点
    if (cl == cr && pos == cl) {t[p] = init(v); return;}
    auto lc = p << 1, rc = lc + 1, mid = (cl + cr) >> 1;
    // 尝试修改左右子树
    modify(pos, v, lc, cl, mid); modify(pos, v, rc, mid + 1, cr);
    // 合并左右子树
    t[p] = merge(t[lc], t[rc]);
}
```

区间查询

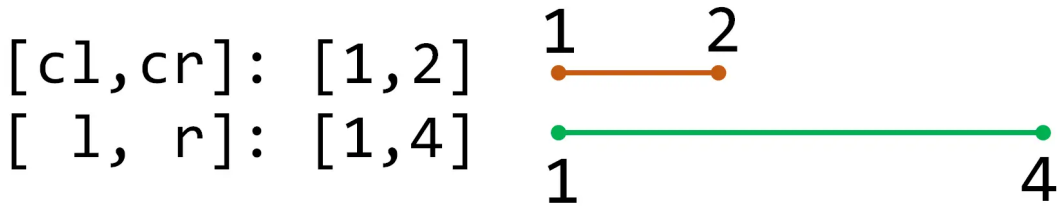
对于**查询区间**和线段树的节点所**代表的区间**，一共有以下三种情况：

1. 当前区间与目标区间**没有交集**：



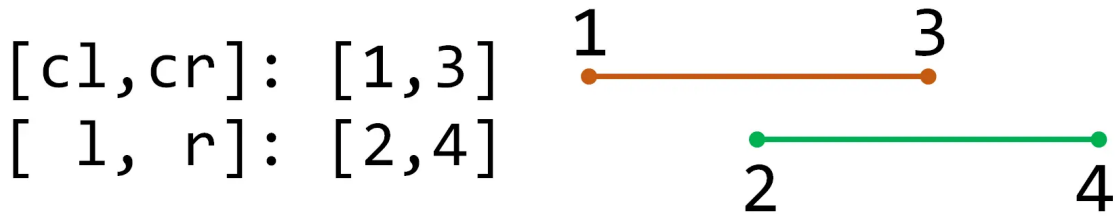
   此时直接结束递归

2. 当前区间被**包含**在目标区间里



这时可以**直接使用**该区间的信息

3. 当前区间与目标区间**相交，但不包含**于其中



这时把当前区间**一分为二**，**分别**进行处理

```
Node qry(int l, int r, int p = rt, int cl = 0, int cr = n - 1) {
    // 类型1
    if (cr < l || cl > r) return {0};
    // 类型2
    if (cl >= l && cr <= r) return t[p];
    // 类型3
    auto lc = p << 1, rc = lc + 1, mid = (cl + cr) >> 1;
    return merge(qry(l, r, lc, cl, mid), qry(l, r, rc, mid + 1, cr));
}
```

现在给出**单点修改，区间查询**的完整代码

```
namespace sgt {
using T = ll;
int n;
static constexpr int rt = 1;

struct Node {
    T sum;
};
```

```
vector<Node> t;

Node merge(const Node& a, const Node& b) {
    Node ret {a.sum + b.sum};
    return ret;
}

Node init(T x) {
    return {x};
}

void build(const vector<T>& a, int p, int cl = 0, int cr = n - 1) {
    if (cl == cr) {t[p] = init(a[cl]); return;}
    auto lc = p << 1, rc =lc + 1, mid = (cl + cr) >> 1;
    build(a, lc, cl, mid); build(a, rc, mid + 1, cr);
    t[p] = merge(t[lc], t[rc]);
}

void init(const vector<T>& a) {
    n = a.size();
    t.resize(n << 2);
    build(a, rt);
}

Node qry(int l, int r, int p = rt, int cl = 0, int cr = n - 1) {
    if (cr < l || cl > r) return {0};
    if (cl >= l && cr <= r) return t[p];
    auto lc = p << 1, rc = lc + 1, mid = (cl + cr) >> 1;
    return merge(qry(l, r, lc, cl, mid), qry(l, r, rc, mid + 1, cr));
}

void modify(int pos, T v, int p = rt, int cl = 0, int cr = n - 1) {
    if (pos > cr || pos < cl) return;
    if (cl == cr && pos == cl) {t[p] = init(v); return;}
    auto lc = p << 1, rc = lc + 1, mid = (cl + cr) >> 1;
    modify(pos, v, lc, cl, mid); modify(pos, v, rc, mid + 1, cr);
    t[p] = merge(t[lc], t[rc]);
}

}
```