线段树（Segment Tree）是一种用于处理区间查询与更新问题的数据结构。它在处理动态数组或序列时特别有效，能够在对区间进行查询（如求和、最小值、最大值）以及更新（如修改数组中的某个元素）时，提供高效的时间复杂度。

线段树的特点

1. **时间复杂度**:
   - **构建**: 构建线段树的时间复杂度是 $O(n)$，其中 n 是数组的长度。
   - **查询**: 线段树可以在 $O(\log n)$ 时间内完成区间查询操作。
   - **更新**: 单点更新也可以在 $O(\log n)$ 时间内完成。
2. **存储结构**:
   - 线段树通常使用一个基于数组的完全二叉树来存储，每个节点表示一个区间的某种属性（如和、最大值、最小值等）。
   - 如果原数组的大小是 n，则线段树的大小约为 $4n$，但实际应用中，大部分实现都使用 $2n$ 的大小。

线段树的构建

假设我们有一个数组 `arr` ，它的长度为 `n` ，我们需要构建一棵线段树来支持区间查询和更新。

基本思想

- 线段树的每个叶子节点表示数组中的一个元素。
- 非叶子节点表示对应区间中的某个属性（如区间和、区间最小值等）。

构建过程

1. **递归分治**:
   - 将区间 `[l, r]` 分成两个子区间 `[l, mid]` 和 `[mid+1, r]` 。
   - 递归构建左右子树，直到区间长度为 1（即叶子节点）。
2. **合并**:
   - 在返回的过程中，将左右子区间的值合并为当前区间的值。

示例

假设我们要构建一个表示区间和的线段树，对于数组 `arr = [1, 3, 5, 7, 9, 11]` ：

1. 构建区间 `[0, 5]` 的线段树，将其分为 `[0, 2]` 和 `[3, 5]` 两部分。

2. 继续递归分治，直到处理单个元素。
3. 然后，从底向上合并，构建完整的线段树。

线段树的查询与更新

查询

线段树的查询过程也是基于递归的。对于查询区间 `[l, r]` ：

1. 如果当前区间与查询区间没有交集，返回一个默认值（如 0 对于求和）。
2. 如果当前区间完全包含在查询区间中，返回当前节点的值。
3. 否则，将当前区间分成两个子区间，递归查询，并将结果合并。

更新

更新过程类似：

1. 找到对应叶子节点，更新值。
2. 向上递归更新父节点，直到根节点。

总结

线段树是处理区间操作的强大工具，特别适用于需要频繁查询和更新的大规模数据场景。它的高效性来自于其树形结构，使得每次查询和更新都只需要访问 $O(\log n)$ 个节点。

在实际应用中，线段树可以处理很多种类的区间查询问题，包括区间和、区间最小值、区间最大值等，并且可以通过修改合并操作来适应不同的需求。

下面是一段使用C++实现线段树来处理区间最小值查询问题的代码。我们将逐行解释代码的原理和操作。

```cpp
#include <iostream>
#include <vector>
#include <algorithm>  // 用于 std::min 函数

using namespace std;

class SegmentTree {
private:
    vector<int> tree;
    int n;

    // 构建线段树，初始化区间 [start, end] 的节点 idx
    void buildTree(const vector<int>& arr, int start, int end, int idx) {
        if (start == end) {
            // 叶子节点表示单个元素
            tree[idx] = arr[start];
        } else {
            int mid = start + (end - start) / 2;
            // 递归构建左右子树
            buildTree(arr, start, mid, 2 * idx + 1);
            buildTree(arr, mid + 1, end, 2 * idx + 2);
            // 当前节点存储左右子树的最小值
            tree[idx] = min(tree[2 * idx + 1], tree[2 * idx + 2]);
        }
    }

    // 递归查询区间 [l, r] 的最小值
    int queryMin(int start, int end, int l, int r, int idx) {
        if (l <= start && r >= end) {
            // 完全覆盖，直接返回当前区间的最小值
            return tree[idx];
        }
        if (end < l || start > r) {
            // 没有交集，返回一个无穷大的值
            return INT_MAX;
        }
        // 部分覆盖，查询左右子区间
        int mid = start + (end - start) / 2;
        return min(queryMin(start, mid, l, r, 2 * idx + 1),
                   queryMin(mid + 1, end, l, r, 2 * idx + 2));
    }

    // 递归更新节点，修改位置 pos 的值为 newVal
```

```
    void updateValue(int start, int end, int pos, int newVal, int idx) {
        if (start == end) {
            // 更新叶子节点
            tree[idx] = newVal;
        } else {
            int mid = start + (end - start) / 2;
            if (pos <= mid) {
                // 更新左子树
                updateValue(start, mid, pos, newVal, 2 * idx + 1);
            } else {
                // 更新右子树
                updateValue(mid + 1, end, pos, newVal, 2 * idx + 2);
            }
            // 更新当前节点为左右子树的最小值
            tree[idx] = min(tree[2 * idx + 1], tree[2 * idx + 2]);
        }
    }

public:
    // 构造函数
    SegmentTree(const vector<int>& arr) {
        n = arr.size();
        tree.resize(4 * n);  // 预留足够的空间
        buildTree(arr, 0, n - 1, 0);
    }

    // 对外暴露的查询接口，查询区间 [l, r] 的最小值
    int queryMin(int l, int r) {
        return queryMin(0, n - 1, l, r, 0);
    }

    // 对外暴露的更新接口，将位置 pos 的值更新为 newVal
    void updateValue(int pos, int newVal) {
        updateValue(0, n - 1, pos, newVal, 0);
    }
};

int main() {
    vector<int> arr = {2, 5, 1, 4, 9, 3};
    SegmentTree segTree(arr);

    cout << "Initial minimum in range [1, 4]: " << segTree.queryMin(1, 4) << endl; // 输
```

```
    segTree.updateValue(3, 0);  // 更新 arr[3] 为 0
    cout << "Minimum in range [1, 4] after update: " << segTree.queryMin(1, 4) << endl;

    return 0;
}
```

代码解释

1. 导入库与定义

```
#include <iostream>
#include <vector>
#include <algorithm>  // 用于 std::min 函数
```

- 导入标准库，用于输入输出、容器操作以及求最小值函数 `std::min` 。

2. 线段树类定义

```
class SegmentTree {
private:
    vector<int> tree;
    int n;
```

- 定义一个 `SegmentTree` 类，私有成员包括存储线段树的 `tree` 数组和原始数组大小 `n` 。

3. 线段树的构建

```
void buildTree(const vector<int>& arr, int start, int end, int idx) {
    if (start == end) {
        tree[idx] = arr[start];
    } else {
        int mid = start + (end - start) / 2;
        buildTree(arr, start, mid, 2 * idx + 1);
        buildTree(arr, mid + 1, end, 2 * idx + 2);
        tree[idx] = min(tree[2 * idx + 1], tree[2 * idx + 2]);
    }
}
```

- `buildTree` 函数递归地构建线段树。
- 如果区间 `[start, end]` 只包含一个元素（`start == end`），则将该元素存入树的对应位置。
- 如果区间大于1，则将区间分为两部分，递归构建左、右子树，并将两个子区间的最小值存入当前节点。

4. 区间最小值查询

```
int queryMin(int start, int end, int l, int r, int idx) {
    if (l <= start && r >= end) {
        return tree[idx];
    }
    if (end < l || start > r) {
        return INT_MAX;
    }
    int mid = start + (end - start) / 2;
    return min(queryMin(start, mid, l, r, 2 * idx + 1),
               queryMin(mid + 1, end, l, r, 2 * idx + 2));
}
```

- `queryMin` 函数在 `[l, r]` 区间查询最小值。
- 如果当前区间完全在查询区间内，直接返回当前节点的值。
- 如果没有交集，返回 `INT_MAX` 作为无效值。
- 如果部分重叠，递归查询左、右子区间，合并结果。

5. 更新操作

```
void updateValue(int start, int end, int pos, int newVal, int idx) {
    if (start == end) {
        tree[idx] = newVal;
    } else {
        int mid = start + (end - start) / 2;
        if (pos <= mid) {
            updateValue(start, mid, pos, newVal, 2 * idx + 1);
        } else {
            updateValue(mid + 1, end, pos, newVal, 2 * idx + 2);
        }
        tree[idx] = min(tree[2 * idx + 1], tree[2 * idx + 2]);
    }
}
```

- `updateValue` 函数递归更新某个位置的值。
- 通过递归找到需要更新的叶子节点并修改其值。
- 更新完成后，向上递归更新其父节点，确保父节点值的正确性。

6. 构造函数

```
SegmentTree(const vector<int>& arr) {
    n = arr.size();
    tree.resize(4 * n);
    buildTree(arr, 0, n - 1, 0);
}
```

- 构造函数初始化线段树。
- `n` 为数组大小，`tree` 的大小设置为 `4 * n` 以确保足够的空间构建完全二叉树。
- 调用 `buildTree` 函数从原始数组构建线段树。

7. 对外接口

```
int queryMin(int l, int r) {
    return queryMin(0, n - 1, l, r, 0);
}

void updateValue(int pos, int newVal) {
    updateValue(0, n - 1, pos, newVal, 0);
}
```

- `queryMin` 和 `updateValue` 是对外暴露的接口，用于查询最小值和更新数组值。

8. 测试代码

```
int main() {
    vector<int> arr = {2, 5, 1, 4, 9, 3};
    SegmentTree segTree(arr);

    cout << "Initial minimum in range [1, 4]: " << segTree.queryMin(1, 4) << endl; // 输

    segTree.updateValue(3, 0);  // 更新 arr[3] 为 0
    cout << "Minimum in range [1,

4] after update: " << segTree.queryMin(1, 4) << endl; // 输出 0

    return 0;
}
```

- 创建一个数组，并构建相应的线段树。
- 查询区间 `[1, 4]` 的最小值，初始查询得到 1。
- 更新数组中的一个值（将 `arr[3]` 从 4 更新为 0），再次查询该区间，结果变为 0。
  在这段代码中， `idx` 是一个参数，用于表示线段树中某个节点的索引。它指向线段树中某个节点的位置，用来追踪当前递归所处理的节点在 `tree` 数组中的位置。

`idx` 的作用

- **在构建线段树时**：
  - `idx` 用于标识当前节点在 `tree` 数组中的位置。
  - 在递归过程中，通过计算左子节点和右子节点的索引（`2 * idx + 1` 和 `2 * idx + 2`），来构建整棵树。
- **在查询和更新操作中**：
  - `idx` 也是用来指示当前节点的位置。通过递归传递 `idx` 参数，程序能够定位到需要操作的具体节点。
  - 对于查询操作， `idx` 表示当前递归所在的节点；对于更新操作， `idx` 表示需要更新的节点。

具体示例

假设有一个包含 6 个元素的数组 `arr = {2, 5, 1, 4, 9, 3}` ，在构建线段树时：

- `idx = 0` 表示根节点，负责管理整个数组 `[0, 5]` 。
- 递归过程中， `idx = 1` 可能对应左子树，管理数组的前半部分 `[0, 2]` ，而 `idx = 2` 可能对应右子树，管理数组的后半部分 `[3, 5]` 。

- 继续递归， `idx = 3` 可能表示 `[0, 1]` ， `idx = 4` 表示 `[2, 2]` 等等。

因此， `idx` 就是线段树中节点的唯一标识符，通过它可以在 `tree` 数组中找到相应节点的位置。

为了更详细地解释如何构建线段树，我将使用从1开始的数组编号，并逐行解释代码的每一步。假设我们有一个数组 `arr = {2, 5, 1, 4, 9, 3}` ，它的索引从1开始。

线段树的构建

我们将重点放在 `buildTree` 函数上，这个函数的作用是递归地构建线段树。下面是该函数的代码，以及每行代码的详细解释。

```
void buildTree(const vector<int>& arr, int start, int end, int idx) {
    if (start == end) {
        tree[idx] = arr[start];
    } else {
        int mid = start + (end - start) / 2;
        buildTree(arr, start, mid, 2 * idx + 1);
        buildTree(arr, mid + 1, end, 2 * idx + 2);
        tree[idx] = min(tree[2 * idx + 1], tree[2 * idx + 2]);
    }
}
```

参数说明

- `arr` ：原始数组，包含要构建线段树的数据。
- `start` ：当前处理的区间的起始索引（从1开始）。
- `end` ：当前处理的区间的结束索引（从1开始）。
- `idx` ：当前节点在线段树数组 `tree` 中的位置。

例子

假设我们有数组 `arr = {2, 5, 1, 4, 9, 3}` ，我们将构建一个线段树来处理区间最小值查询。

1. **初始调用**： `buildTree(arr, 1, 6, 0)` 对应处理区间 `[1, 6]` ，即整个数组。
   这意味着根节点 `idx = 0` 管理整个数组 `[1, 6]` 的最小值。
2. **叶子节点处理**：

   ```
   if (start == end) {
       tree[idx] = arr[start];
   }
   ```

- 如果 `start` 和 `end` 相等，表示当前区间只有一个元素，这是一个叶子节点。
- 直接将该元素存入 `tree` 数组的 `idx` 位置。

示例：当 `start = 3` ， `end = 3` 时，调用 `buildTree(arr, 3, 3, 4)` ，此时 `idx = 4` ，这意味着我们处理的是区间 `[3, 3]` ，即数组 `arr[3]` 。我们将 `tree[4]` 设置为 `arr[3]` 的值 `1` 。

3. **递归构建左子树和右子树**：

```
int mid = start + (end - start) / 2;
buildTree(arr, start, mid, 2 * idx + 1);
buildTree(arr, mid + 1, end, 2 * idx + 2);
```

- 我们首先计算当前区间的中点 `mid` ，将区间分成左右两部分。
- `mid` 的计算方式是 `mid = start + (end - start) / 2` 。例如，对于 `[1, 6]` ， `mid = 1 + (6 - 1) / 2 = 3` ，所以区间分为 `[1, 3]` 和 `[4, 6]` 。
- 递归调用 `buildTree` 函数来构建左子树和右子树。
- 左子树的索引为 `2 * idx + 1` ，右子树的索引为 `2 * idx + 2` 。

示例：对于根节点（ `idx = 0` ），区间 `[1, 6]` 被分为 `[1, 3]` 和 `[4, 6]` ，接着递归构建左子树 `buildTree(arr, 1, 3, 1)` 和右子树 `buildTree(arr, 4, 6, 2)` 。

4. **合并结果**：

```
tree[idx] = min(tree[2 * idx + 1], tree[2 * idx + 2]);
```

- 当左子树和右子树构建完成后，将它们的最小值合并为当前节点的值。
- `tree[idx]` 保存的是左右子树中最小的值。

示例：假设 `buildTree(arr, 1, 3, 1)` 处理区间 `[1, 3]` 返回的左子树最小值为 `1` ， `buildTree(arr, 4, 6, 2)` 处理区间 `[4, 6]` 返回的右子树最小值为 `3` ，那么根节点 `tree[0]` 将被设为 `min(1, 3) = 1` 。

完整树的结构

通过递归调用，最终我们将得到一棵线段树。假设 `arr = {2, 5, 1, 4, 9, 3}` ，用1开始的索引来表示：

- 根节点 `tree[0]` 管理整个数组 `[1, 6]` ，存储最小值 `1` 。
- 左子树 `tree[1]` 管理 `[1, 3]` ，存储最小值 `1` 。
- 右子树 `tree[2]` 管理 `[4, 6]` ，存储最小值 `3` 。

逐渐递归下去，直到所有叶子节点都填满为止。

总结

每个节点的 `idx` 表示线段树中节点在数组 `tree` 中的位置。通过递归分治法，我们将整个区间分成若干子区间，逐步构建线段树的每个节点。最终，树的根节点（ `tree[0]` ）存储了整个数组的最小值，树的其他节点存储了各自子区间的最小值。

在线段树中，搜索过程通常指的是查询某个区间内的最小值、最大值或其他聚合信息。为了更好地理解搜索过程，我们将详细讲解如何使用线段树来查询一个给定区间内的最小值，并通过逐步解析代码的方式说明每一步的具体操作。

问题描述

假设我们有一个从1开始编号的数组 `arr = {2, 5, 1, 4, 9, 3}` ，我们希望使用线段树来查询任意给定区间 `[l, r]` 内的最小值。

查询函数： `queryMin`

我们使用如下函数来查询区间 `[l, r]` 内的最小值：

```
int queryMin(int start, int end, int l, int r, int idx) {
    if (l <= start && r >= end) {
        return tree[idx];
    }
    if (end < l || start > r) {
        return INT_MAX;
    }
    int mid = start + (end - start) / 2;
    return min(queryMin(start, mid, l, r, 2 * idx + 1),
               queryMin(mid + 1, end, l, r, 2 * idx + 2));
}
```

参数说明

- `start` ：当前节点管理的区间起始索引。
- `end` ：当前节点管理的区间结束索引。
- `l` 和 `r` ：查询的目标区间 `[l, r]` 。
- `idx` ：当前节点在 `tree` 数组中的位置。

查询过程的详细解释

假设我们要查询区间 `[2, 5]` 的最小值，初始调用 `queryMin(1, 6, 2, 5, 0)` ，即从根节点开始搜索。

1. **判断当前区间是否完全覆盖查询区间**：

   ```
   if (l <= start && r >= end) {
       return tree[idx];
   }
   ```

   - 如果当前节点管理的区间 `[start, end]` 完全包含在查询区间 `[l, r]` 内（即 `l <= start` 且 `r >= end` ），那么当前节点存储的值就是该区间的最小值，直接返回 `tree[idx]` 。

   示例：如果我们查询区间 `[2, 5]` ，而当前节点管理的是 `[4, 6]` ，此时 `start = 4, end = 6, l = 2, r = 5` ，不完全覆盖，所以不返回。
2. **判断当前区间与查询区间是否没有交集**：

   ```
   if (end < l || start > r) {
       return INT_MAX;
   }
   ```

   - 如果当前区间与查询区间没有任何交集（即 `end < l` 或 `start > r` ），这意味着当前节点不需要参与最小值的计算，返回一个极大值 `INT_MAX` （表示无效值）。

   示例：如果当前节点管理的区间 `[1, 1]` ，此时 `start = 1, end = 1, l = 2, r = 5` ，显然没有交集，返回 `INT_MAX` 。
3. **部分覆盖：递归查询左右子区间**：

   ```
   int mid = start + (end - start) / 2;
   return min(queryMin(start, mid, l, r, 2 * idx + 1),
              queryMin(mid + 1, end, l, r, 2 * idx + 2));
   ```

   - 如果当前区间 `[start, end]` 和查询区间 `[l, r]` 部分重叠，继续递归查询左右子树。
   - 首先计算当前区间的中点 `mid` ，将当前区间 `[start, end]` 分为 `[start, mid]` 和 `[mid + 1, end]` 两个子区间。
   - 递归查询左子树和右子树的最小值，然后返回这两个子区间的最小值作为结果。

   示例：查询区间 `[2, 5]` ，当前节点管理区间 `[1, 6]` ，将其分为 `[1, 3]` 和 `[4, 6]` ，然后递归查询左右子区间，最后返回两个子区间的最小值。

完整查询过程示例

假设我们要查询 `arr` 中 `[2, 5]` 的最小值，调用 `queryMin(1, 6, 2, 5, 0)`。以下是递归查询的详细步骤：

1. **根节点**：
   - `start = 1, end = 6, l = 2, r = 5, idx = 0`。
   - 当前区间 `[1, 6]` 与查询区间 `[2, 5]` 部分重叠，继续递归查询。
   - 计算 `mid = 1 + (6 - 1) / 2 = 3`。
   - 查询左子树 `[1, 3]` 和右子树 `[4, 6]`。
2. **左子树 `[1, 3]`**：
   - `start = 1, end = 3, l = 2, r = 5, idx = 1`。
   - 当前区间 `[1, 3]` 与查询区间 `[2, 5]` 部分重叠，继续递归查询。
   - 计算 `mid = 1 + (3 - 1) / 2 = 2`。
   - 查询左子树 `[1, 2]` 和右子树 `[3, 3]`。
3. **左子树的左子树 `[1, 2]`**：
   - `start = 1, end = 2, l = 2, r = 5, idx = 3`。
   - 当前区间 `[1, 2]` 与查询区间 `[2, 5]` 部分重叠，继续递归查询。
   - 计算 `mid = 1 + (2 - 1) / 2 = 1`。
   - 查询左子树 `[1, 1]` 和右子树 `[2, 2]`。
4. **到达叶子节点 `[1, 1]`**：
   - `start = 1, end = 1, l = 2, r = 5, idx = 7`。
   - 当前区间 `[1, 1]` 与查询区间 `[2, 5]` 无交集，返回 `INT_MAX`。
5. **叶子节点 `[2, 2]`**：
   - `start = 2, end = 2, l = 2, r = 5, idx = 8`。
   - 当前区间 `[2, 2]` 完全包含在查询区间 `[2, 5]` 内，返回 `tree[8] = 5`。
6. **回到节点 `[1, 2]`**：
   - 左子树返回 `INT_MAX`，右子树返回 `5`，取 `min(INT_MAX, 5) = 5`。
7. **右子树 `[3, 3]`**：
   - `start = 3, end = 3, l = 2, r = 5, idx = 4`。
   - 当前区间 `[3, 3]` 完全包含在查询区间 `[2, 5]` 内，返回 `tree[4] = 1`。
8. **回到节点 `[1, 3]`**：
   - 左子树返回 `5`，右子树返回 `1`，取 `min(5, 1) = 1`。
9. **右子树 `[4, 6]`**：
   - `start = 4, end = 6, l = 2, r = 5, idx = 2`。
   - 当前区间 `[4, 6]` 与查询区间 `[2, 5]` 部分重叠，继续递归查询。
   - 计算 `mid = 4 + (6 - 4) / 2 = 5`。
   - 查询左子树 `[4, 5]` 和右子树 `[6, 6]`。

10. **左子树 `[4, 5]`** ：
    - `start = 4, end = 5, l = 2, r = 5, idx = 5` 。
    - 当前区间 `[4, 5]` 完全包含在查询区间 `[2, 5]` 内，返回 `tree[5] = 4` 。
11. **右子树 `[6, 6]`** ：
    - `start = 6, end = 6, l = 2, r = 5, idx = 6` 。
    - 当前区间 `[6, 6]` 与

查询区间 `[2, 5]` 无交集，返回 `INT_MAX` 。

12. **回到节点 `[4, 6]`** ：
    - 左子树返回 `4` ，右子树返回 `INT_MAX` ，取 `min(4, INT_MAX) = 4` 。
13. **回到根节点 `[1, 6]`** ：
    - 左子树返回 `1` ，右子树返回 `4` ，最终结果为 `min(1, 4) = 1` 。

总结

通过递归地将查询区间 `[l, r]` 与线段树节点管理的区间 `[start, end]` 比较，我们可以有效地找到查询区间内的最小值。这个过程利用了线段树的分治思想和区间重叠的特性，确保每次查询的时间复杂度为 `O(log n)` 。