Segment Tree là gì?

Segment Tree (Cây phân đoạn) hay segtree, về cơ bản là một cây nhị phân được sử dụng để lưu trữ các khoảng hoặc phân đoạn. Mỗi nút trên cây segment tree quản lí cho một đoạn. Xét mảng **A** có kích thước **N** và một cây phân đoạn **T** tương ứng:

1. Gốc của **T** sẽ đại diện cho toàn bộ mảng **A[0: N – 1]**
2. Mỗi nút lá trên cây sẽ quản lí một phần tử **A[i]** duy nhất sao cho **0 <= i <= N**
3. Các nút trung gian của cây **T** quản lí cho các phần tử **A[i: j]** trong đó **0 <= i < j < N**.

Gốc của "cây phân đoạn" sẽ đại diện cho toàn bộ mảng **A [0: N-1]**. Sau đó, chúng ta sẽ chia đoạn này thành 2 nửa và hai con của gốc sẽ quản lí các phần tử từ **A [0: (N-1) / 2]** và **A[(N-1) / 2 + 1: (N-1)]**. Tóm lại, trong mỗi bước chúng ta sẽ chia một đoạn thành 2 nửa và hai con sẽ đại diện cho hai nửa đó. Vì vậy, chiều cao của cây phân đoạn sẽ là log2(N). Có N nút lá biểu diễn N phần tử của mảng. Số nút trung gian là N-1. Vậy tổng số nút là 2 * N - 1.

Khi xây dựng một cây phân đoạn, chúng ta không thể thay đổi cấu trúc của nó, tức là nó có cấu trúc tĩnh. Chúng ta có thể cập nhật giá trị của các nút nhưng không thể thay đổi cấu trúc của nó. Cây phân đoạn có bản chất là đệ quy. Do tính chất đệ quy của nó, cây phân đoạn rất dễ thực hiện. Cây phân đoạn có thể thực hiện 2 thao tác:

1. Cập nhật: Trong thao tác này, chúng ta có thể cập nhật một phần tử trong Mảng và phản ánh sự thay đổi tương ứng trong cây Phân đoạn.
2. Truy vấn: Trong thao tác này, chúng ta có thể truy vấn trên một khoảng hoặc phân đoạn và trả lại kết quả cho câu hỏi trên khoảng đó.

Thực hiện:

Vì cây phân đoạn là cây nhị phân, chúng ta có thể sử dụng một mảng tuyến tính đơn giản để biểu diễn cây phân đoạn. Trong hầu hết mọi vấn đề của cây segtree, chúng ta cần suy nghĩ về những gì chúng ta cần lưu trữ trong cây phân đoạn?.

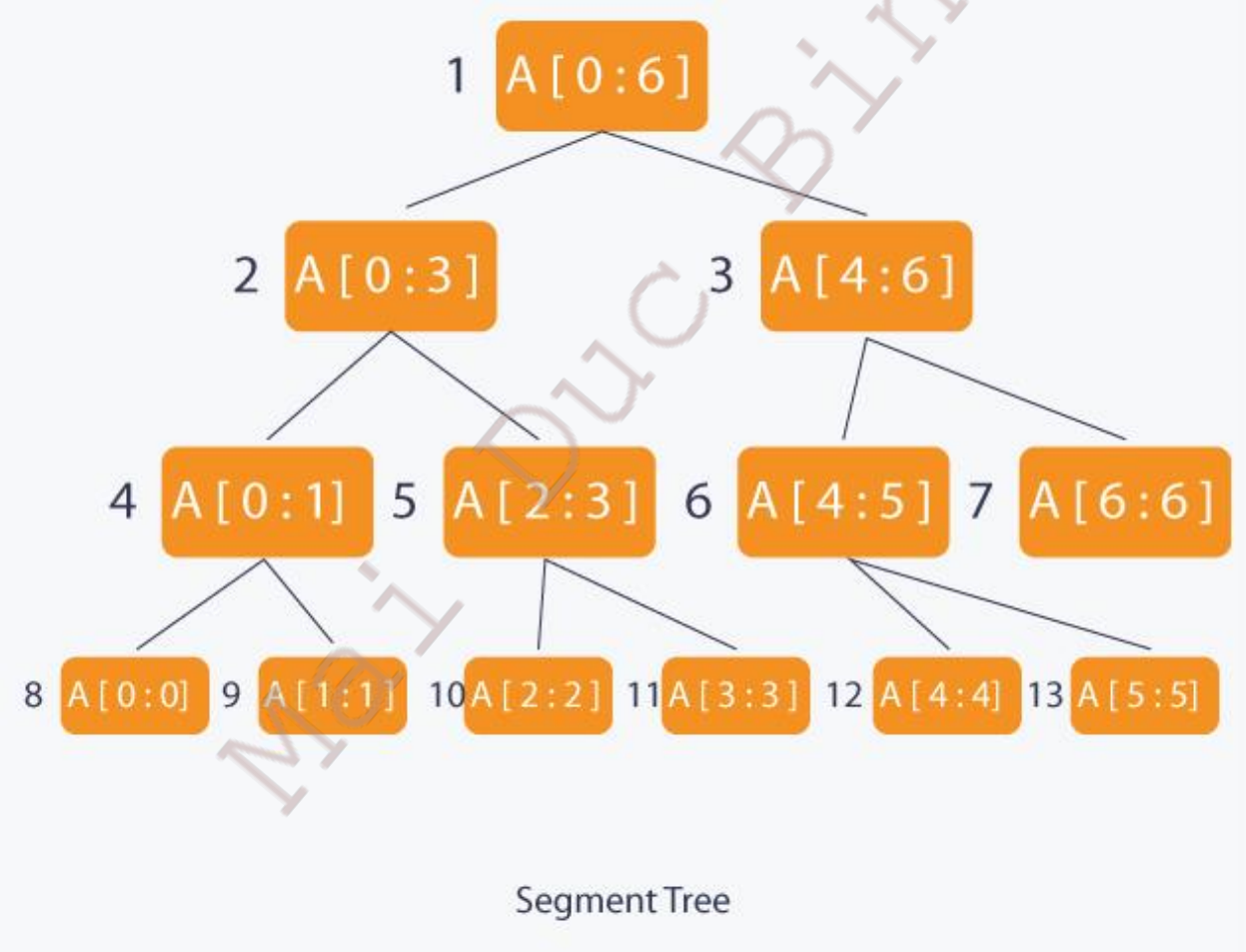
Ví dụ, nếu chúng ta muốn tìm tổng của tất cả các phần tử trong một mảng từ trái sang phải, thì tại mỗi nút (trừ các nút lá), chúng ta sẽ lưu trữ tổng các nút con của nó. Nếu chúng ta muốn tìm giá trị min của tất cả các phần tử trong một mảng từ trái sang phải, thì tại mỗi nút (trừ các nút lá), chúng ta sẽ lưu trữ giá trị min của các nút con.

Khi biết cần lưu trữ những gì trên cây phân đoạn, chúng ta có thể xây dựng cây bằng cách sử dụng đệ quy (xây dựng từ dưới lên). Chúng ta sẽ bắt đầu với các lá và đi lên gốc và cập nhật các thay đổi tương ứng trong các nút nằm trên đường đi từ lá đến gốc. Lá đại diện cho một phần tử duy nhất. Trong mỗi bước, chúng ta sẽ hợp nhất hai nút con để tạo thành một nút cha trung gian. Mỗi nút trung gian sẽ đại diện cho một tập hợp các phần tử

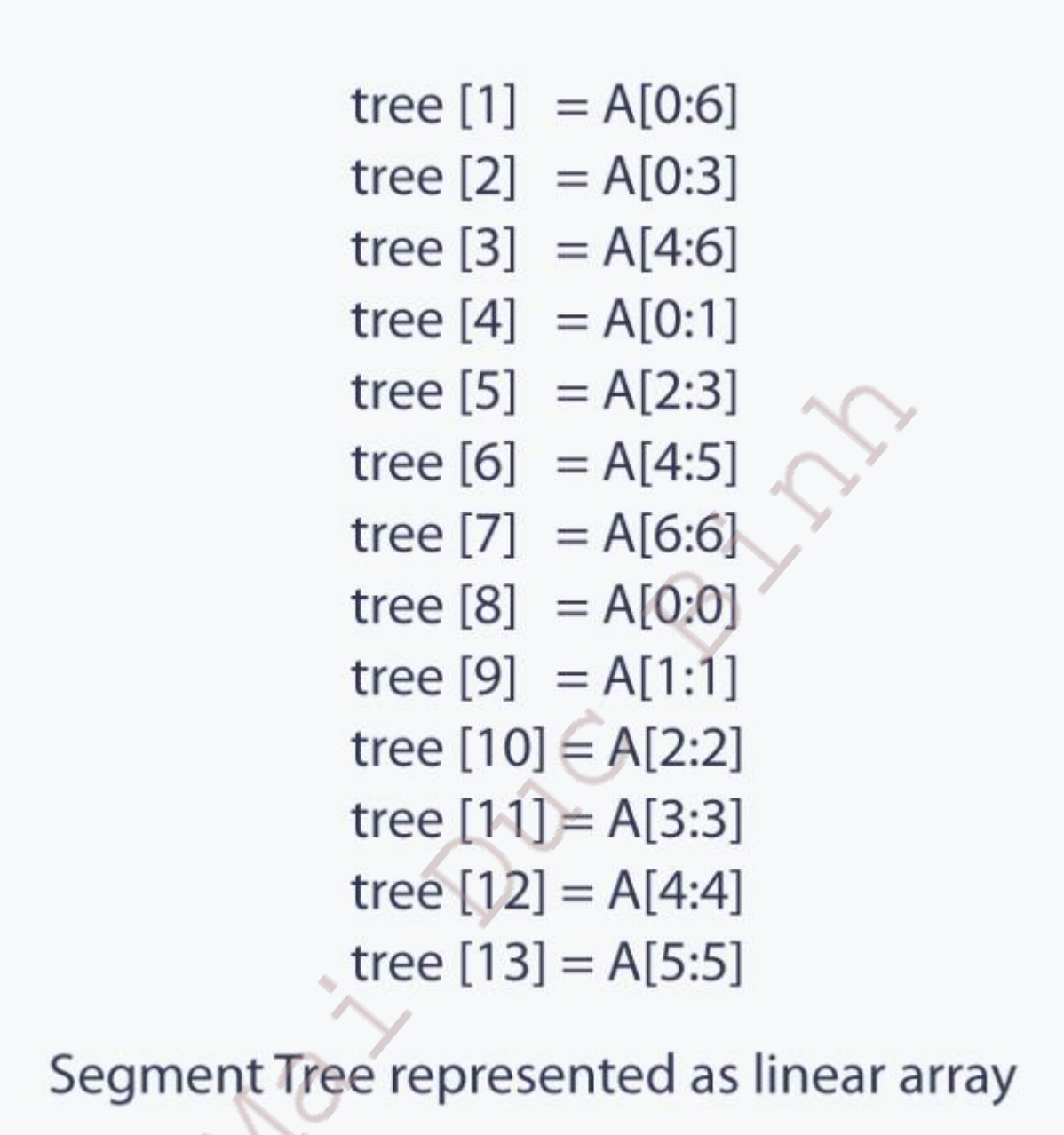
nằm trong đoạn con của nó. Việc hợp nhất có thể khác nhau đối với các yêu cầu khác nhau. Vì vậy, đệ quy sẽ kết thúc ở gốc, nơi sẽ đại diện cho toàn bộ mảng.

Để cập nhật, chúng ta chỉ cần tìm kiếm nút lá chứa phần tử cần cập nhật. Điều này có thể được thực hiện bằng cách đi đến con bên trái hoặc con bên phải tùy thuộc vào đoạn có chứa phần tử. Sau khi tìm thấy nút lá, chúng ta sẽ cập nhật nó và một lần nữa sử dụng phương pháp từ dưới lên để cập nhật sự thay đổi tương ứng trên đường đi từ lá đến gốc.

Để thực hiện một truy vấn trên cây phân đoạn từ l đến r, chúng ta sẽ đệ quy trên cây bắt đầu từ gốc và kiểm tra xem đoạn được quản lí bởi nút hiện tại có nằm hoàn toàn trong khoảng từ l đến r hay không. Nếu đoạn được quản lí bởi một nút nằm hoàn toàn trong khoảng từ l đến r, chúng tôi sẽ trả về giá trị của nút đó. Cây phân đoạn của mảng A có kích thước 7 sẽ như sau:



Segment Tree



Tất cả điều này chúng ta sẽ làm rõ bằng cách lấy một ví dụ. Cho một mảng A có kích thước N và một số truy vấn. Có hai loại truy vấn:

1. Cập nhật: tăng giá trị của **A[idx]** lên **val** đơn vị.
2. Truy vấn: trả về giá trị của **A[l] + A[l + 1] + A[l + 2] + ... + A[r-1] + A[r]** sao cho **0 <= l <= r < N**

Giải thuật đơn giản:

Cách tiếp cận cơ bản nhất: Đối với mỗi truy vấn, hãy chạy một vòng lặp từ l đến r và tính tổng của tất cả các phần tử. Vì vậy, truy vấn sẽ lấy **O(N)**. **A[idx] += val** sẽ cập nhật giá

trị của phần tử. Cập nhật sẽ tốn **O(1).** Thuật toán này ổn nếu số lượng thao tác cập nhật rất lớn và rất ít thao tác truy vấn.

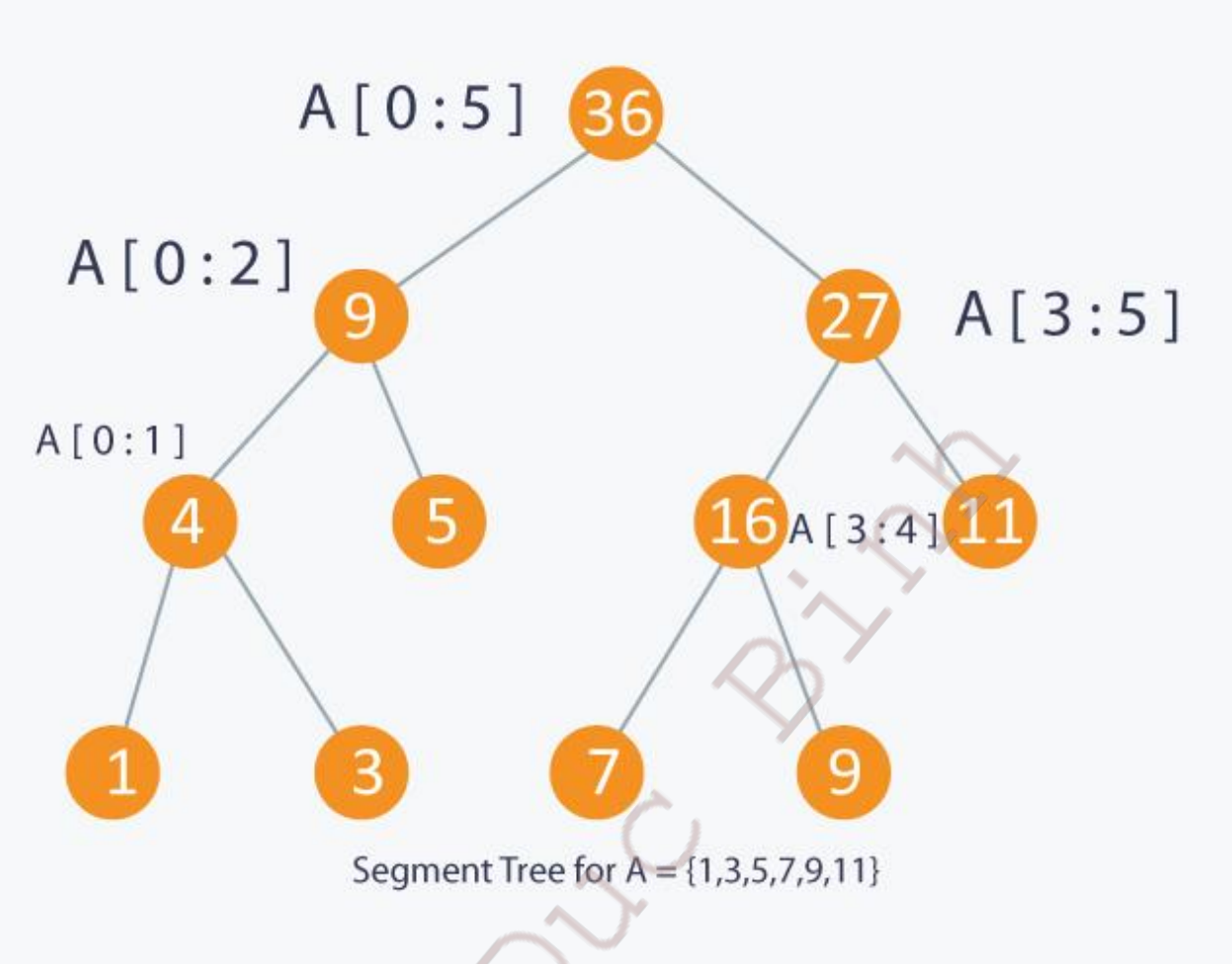
Hoặc: chúng ta sẽ xử lý trước và lưu trữ tổng tích lũy của các phần tử của mảng trong một mảng tổng. Đối với mỗi truy vấn chỉ cần trả về (**sum[r] - sum[l-1]**). Bây giờ hoạt động truy vấn sẽ tốn **O(1).** Nhưng để cập nhật, chúng ta cần chạy một vòng lặp và thay đổi giá trị của tất cả các **sum[i]** sao cho **l <= i <= r**. Vì vậy thao tác cập nhật sẽ tốn **O(N)**. Thuật toán này ổn nếu số lượng thao tác truy vấn rất lớn và rất ít thao tác cập nhật.

Sử dụng Segment Tree:

Hãy xem cách sử dụng cây phân đoạn và những gì chúng ta sẽ lưu trữ trong cây phân đoạn trong bài toán này. Như đã biết, mỗi nút trung gian của segtree sẽ đại diện cho một khoảng hoặc một đoạn. Trong bài toán này, chúng ta cần tìm tổng của tất cả các phần tử trong dãy đã cho. Vì vậy, trong mỗi nút, chúng ta sẽ lưu trữ tổng của tất cả các phần tử của khoảng được quản lí bởi nút đó. Bằng cách nào? Chúng ta sẽ xây dựng một cây phân đoạn bằng cách sử dụng đệ quy (phương pháp tiếp cận từ dưới lên) như đã giải thích ở trên. Mỗi lá quản lí một phần tử duy nhất. Tất cả các nút trung gian là tổng của cả hai nút con của nó.

```
void build(int node, int start, int end)
{
    if(start == end)
    {
        // Leaf node will have a single element
        tree[node] = A[start];
    }
    else
    {
        int mid = (start + end) / 2;
        // Recurse on the left child
        build(2*node, start, mid);
        // Recurse on the right child
        build(2*node+1, mid+1, end);
        // Internal node will have the sum of both of its children
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
```

Trong đoạn code trên, chúng ta sẽ bắt đầu từ nút gốc **root** và đệ quy ở con bên trái và con bên phải cho đến khi gặp các lá. Từ các lá, chúng ta sẽ quay trở lại gốc và cập nhật tất cả các nút trên đường đi. Vì cây phân đoạn là cây nhị phân, **2*node** sẽ đại diện cho nút bên trái và **2*node + 1** đại diện cho nút bên phải. **start** và **end** đại diện cho hai đầu được quản lí bởi nút. Độ phức tạp của hàm build() là **O(N).**



Để cập nhật một phần tử, cần xét đoạn quản lí phần tử đó và đệ quy tương ứng ở con bên trái hoặc con bên phải.

```
void update(int node, int start, int end, int idx, int val)
{
    if(start == end)
    {
        // Leaf node
        A[idx] += val;
        tree[node] += val;
    }
    else
    {
        int mid = (start + end) / 2;
        if(start <= idx and idx <= mid)
        {
            // If idx is in the left child, recurse on the left child
            update(2*node, start, mid, idx, val);
        }
        else
        {
            // if idx is in the right child, recurse on the right child
```

```
            update(2*node+1, mid+1, end, idx, val);
        }
        // Internal node will have the sum of both of its children
        tree[node] = tree[2*node] + tree[2*node+1];
    }
}
```

Độ phức tạp của hàm update() sẽ là **O(logN).**

Để truy vấn trên một đoạn, chúng ta cần kiểm tra 3 điều kiện:

1. Đoạn được quản lí bởi nút hiện tại nằm trong đoạn cần xét.
2. Đoạn được quản lí bởi nút hiện tại nằm ngoài đoạn cần xét.
3. Hai đoạn này giao nhau.

Nếu đoạn được quản lí bởi nút hiện tại nằm hoàn toàn bên ngoài đoạn cần xét, chúng ta sẽ chỉ trả về 0. Nếu đoạn được quản lí bởi nút hiện tại nằm hoàn toàn bên trong đoạn cần xét, chúng ta sẽ trả về giá trị của nút là tổng của tất cả các phần tử trong đoạn được quản lí bởi nút đó. Và nếu đoạn được quản lí bởi nút có một phần nằm bên trong và một phần bên ngoài đoạn cần xét, chúng ta sẽ trả về tổng của phần con bên trái hoặc phần con bên phải. Độ phức tạp của truy vấn sẽ là **O(logN).**

```
int query(int node, int start, int end, int l, int r)
{
    if(r < start or end < l)
    {
        // range represented by a node is completely outside the given range
        return 0;
    }
    if(l <= start and end <= r)
    {
        // range represented by a node is completely inside the given range
        return tree[node];
    }
    // range represented by a node is partially inside and partially outside
the given range
    int mid = (start + end) / 2;
    int p1 = query(2*node, start, mid, l, r);
    int p2 = query(2*node+1, mid+1, end, l, r);
    return (p1 + p2);
}
```


Cập nhật đoạn (Lazy)

Đôi khi các truy vấn sẽ yêu cầu bạn cập nhật một khoảng từ **l** đến **r**, thay vì một phần tử duy nhất. Một giải pháp là cập nhật từng phần tử một. Độ phức tạp của phương pháp này sẽ là **O(N)** cho mỗi thao tác vì N phần tử ở đâu đó trong mảng và việc cập nhật một phần tử đơn lẻ sẽ mất **O(logN)**.

Để tránh nhiều lần gọi hàm, chúng ta có thể sửa đổi hàm update() có thể thực hiện trên một đoạn.

```
void updateRange(int node, int start, int end, int l, int r, int val)
{
    // out of range
    if (start > end or start > r or end < l)
        return;

    // Current node is a leaf node
    if (start == end)
    {
        // Add the difference to current node
        tree[node] += val;
        return;
    }

    // If not a leaf node, recur for children.
    int mid = (start + end) / 2;
    updateRange(node*2, start, mid, l, r, val);
    updateRange(node*2 + 1, mid + 1, end, l, r, val);

    // Use the result of children calls to update this node
    tree[node] = tree[node*2] + tree[node*2+1];
}
```

Khi cần cập nhật một đoạn, chúng ta sẽ cập nhật một nút và đánh dấu nút con của nó rằng nó cần được cập nhật và cập nhật nó khi cần thiết. Đối với điều này, chúng ta cần một mảng **lazy[]** có cùng kích thước với cây phân đoạn. Ban đầu tất cả các phần tử của mảng **lazy[]** sẽ là 0 thể hiện rằng không có cập nhật nào đang chờ xử lý. Nếu có phần tử **lazy[k]** khác 0, cần cập nhật nút **k** trên cây phân đoạn trước khi thực hiện bất kỳ thao tác truy vấn nào.

Để cập nhật đoạn, cần lưu ý 3 điều.

1. Nếu nút hiện tại đang chờ cập nhật, thì trước tiên hãy cập nhật vào nút hiện tại.
2. Nếu đoạn được quản lí bởi nút hiện tại nằm hoàn toàn trong đoạn cần cập nhật, thì hãy cập nhật nút hiện tại và cập nhật mảng **lazy[]** cho các nút con.
3. Nếu đoạn được quản lí bởi nút hiện tại giao với đoạn cần cập nhật, thì hãy cập nhật các nút con.

Vì đã thay đổi hàm update() để hoãn hoạt động cập nhật, chúng ta cũng phải thay đổi hàm truy vấn. Thay đổi duy nhất mà chúng ta cần thực hiện là kiểm tra xem có bất kỳ hoạt động nào đang chờ cập nhật trên nút đó hay không. Nếu có một hoạt động cập nhật đang chờ xử lý, trước tiên hãy cập nhật nút và sau đó thực hiện giống như hàm truy vấn trước đó.

```
void updateRange(int node, int start, int end, int l, int r, int val)
{
    if(lazy[node] != 0)
    {
        // This node needs to be updated
        tree[node] += (end - start + 1) * lazy[node]; // Update it
        if(start != end)
        {
            lazy[node*2] += lazy[node];               // Mark child as lazy
            lazy[node*2+1] += lazy[node];             // Mark child as lazy
        }
        lazy[node] = 0;                               // Reset it
    }
    if(start > end or start > r or end < l)           // Current segment is
not within range [l, r]
        return;
    if(start >= l and end <= r)
    {
        // Segment is fully within range
        tree[node] += (end - start + 1) * val;
        if(start != end)
        {
            // Not leaf node
            lazy[node*2] += val;
            lazy[node*2+1] += val;
        }
        return;
    }
    int mid = (start + end) / 2;
    updateRange(node*2, start, mid, l, r, val);       // Updating left child
    updateRange(node*2 + 1, mid + 1, end, l, r, val); // Updating right child
    tree[node] = tree[node*2] + tree[node*2+1];       // Updating root
}
int queryRange(int node, int start, int end, int l, int r)
{
    if(start > end or start > r or end < l)
        return 0;         // Out of range
    if(lazy[node] != 0)
    {
        // This node needs to be updated
        tree[node] += (end - start + 1) * lazy[node];            // Update it
        if(start != end)
        {
            lazy[node*2] += lazy[node];      // Mark child as lazy
            lazy[node*2+1] += lazy[node];    // Mark child as lazy
        }
        lazy[node] = 0;                  // Reset it
    }
    if(start >= l and end <= r)
        return tree[node];
    int mid = (start + end) / 2;
    int p1 = queryRange(node*2, start, mid, l, r);       // Query left child
    int p2 = queryRange(node*2 + 1, mid + 1, end, l, r); // Query right child
    return (p1 + p2);
}
```