Tất tần tật về Cây Phân Đoạn (Segment Tree)

Table of Contents

LƯU Ý:

- Khi Segment Tree mới được du nhập vào Việt Nam, một số tài liệu gọi là Interval Tree. Đây là cách gọi không chính xác, bởi Interval Tree là một CTDL khác.

- Tất cả hàm trong bài đều đánh số từ 1. Các nút của cây phân đoạn sẽ quản lý đoạn $[l, r]$
- Segment Tree còn có một cách cài đặt khác sử dụng ít bộ nhớ hơn (tối đa $2 * N$ phần tử), cài đặt ngắn hơn và chạy nhanh hơn. Tuy nhiên theo cá nhân mình không dễ hiểu bằng cách cài đặt trong bài viết này.

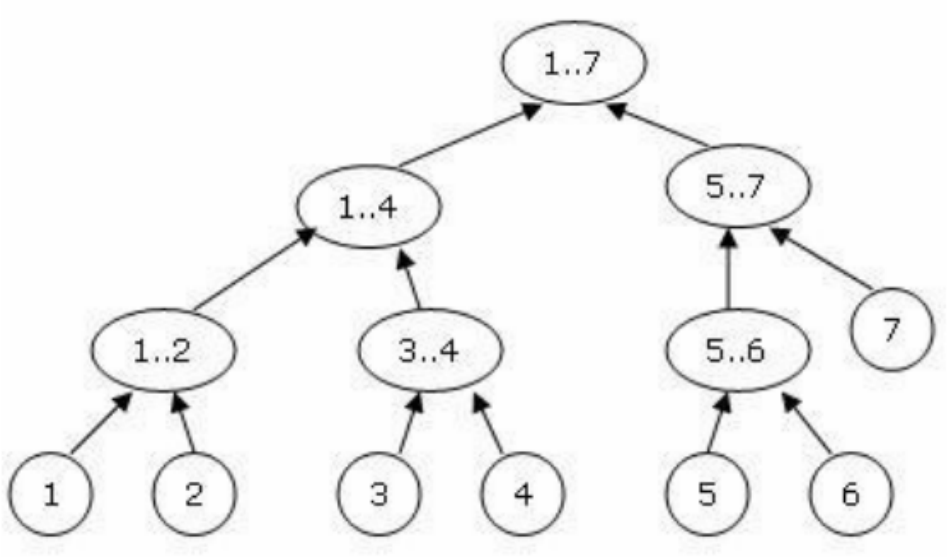
0. Giới thiệu

Segment Tree là một cấu trúc dữ liệu được sử dụng rất nhiều trong các kỳ thi, đặc biệt là trong những bài toán xử lý trên dãy số.

Segment Tree là một cây. Cụ thể hơn, nó là một cây nhị phân đầy đủ (mỗi nút là lá hoặc có đúng 2 nút con), với mỗi nút quản lý một đoạn trên dãy số. Với một dãy số gồm N phần tử, nút gốc sẽ lưu thông tin về đoạn $[1, N]$, nút con trái của nó sẽ lưu thông tin về đoạn $[1, \lfloor N/2 \rfloor]$ và nút con phải sẽ lưu thông tin về đoạn $[\lfloor N/2 \rfloor + 1, N]$. Tổng quát hơn: nếu nút A lưu thông tin đoạn $[i, j]$, thì 2 con của nó: $A1$ và $A2$ sẽ lưu thông tin của các đoạn $[i, \lfloor (i + j)/2 \rfloor]$ và đoạn $[\lfloor (i + j)/2 \rfloor + 1, j]$.

Ví dụ

Xét một dãy gồm 7 phần tử, Segment Tree sẽ quản lý các đoạn như sau:



Cài đặt

Để cài đặt, ta có thể dùng một mảng 1 chiều, phần tử thứ nhất của mảng thể hiện nút gốc. Phần tử thứ id sẽ có 2 con là $2 * id$ (con trái) và $2 * id + 1$ (con phải). Với cách cài đặt này, người ta đã chứng minh được bộ nhớ cần dùng cho ST không quá $4 * N$ phần tử.

Áp dụng

Để dễ hình dung, ta lấy 1 ví dụ cụ thể:

- Cho dãy N phần tử $(N \leq 10^5)$. Ban đầu mỗi phần tử có giá trị 0.
- Có Q truy vấn $(Q \leq 10^5)$. Mỗi truy vấn có 1 trong 2 loại:
  i. Gán giá trị v cho phần tử ở vị trí i.
  ii. Tìm giá trị lớn nhất cho đoạn $[i, j]$.

Cách đơn giản nhất là dùng 1 mảng A duy trì giá trị các phần tử. Với thao tác 1 thì ta gán $A[i] = v$. Với thao tác 2 thì ta dùng 1 vòng lặp từ i đến j để tìm giá trị lớn nhất. Rõ ràng cách này có độ phức tạp là $O(N * Q)$ và không thể chạy trong thời gian cho phép.

Cách dùng Segment Tree như sau:

- Với truy vấn loại 1, ta sẽ cập nhật thông tin của các nút trên cây ST mà đoạn nó quản lý chứa phần tử i.
- Với truy vấn loại 2, ta sẽ tìm tất cả các nút trên cây ST mà đoạn nó quản lý nằm trong $[i, j]$, rồi lấy max của các nút này.

Cài đặt như sau:

```
// Truy vấn: A(i) = v
// Hàm cập nhật trên cây ST, cập nhật cây con gốc id quản lý đoạn [l, r]
void update(int id, int l, int r, int i, int v) {
    if (i < l || r < i) {
        // i nằm ngoài đoạn [l, r], ta bỏ qua nút i
        return ;
    }
    if (l == r) {
        // Đoạn chỉ gồm 1 phần tử, không có nút con
        ST[id] = v;
        return ;
    }

    // Gọi đệ quy để xử lý các nút con của nút id
    int mid = (l + r) / 2;
    update(id*2, l, mid, i, v);
    update(id*2 + 1, mid+1, r, i, v);

    // Cập nhật lại giá trị max của đoạn [l, r] theo 2 nút con:
    ST[id] = max(ST[id*2], ST[id*2 + 1]);
}

// Truy vấn: tìm max đoạn [u, v]
// Hàm tìm max các phần tử trên cây ST nằm trong cây con gốc id - quản lý đoạn [l, r]
int get(int id, int l, int r, int u, int v) {
    if (v < l || r < u) {
        // Đoạn [u, v] không giao với đoạn [l, r], ta bỏ qua đoạn này
        return -INFINITY;
    }
    if (u <= l && r <= v) {
        // Đoạn [l, r] nằm hoàn toàn trong đoạn [u, v] mà ta đang truy vấn, ta trả lại
        // thông tin lưu ở nút id
        return ST[id];
    }
    int mid = (l + r) / 2;
    // Gọi đệ quy với các con của nút id
    return max(get(id*2, l, mid, u, v), get(id*2 + 1, mid+1, r, u, v));
}
```

Phân tích thời gian chạy

Mỗi thao tác truy vấn trên cây ST có độ phức tạp $O(\log N)$. Để chứng minh điều này, ta xét 2 loại thao tác trên cây ST:

1. Truy vấn 1 phần tử trên ST (giống thao tác `update` ở trên)
2. Truy vấn nhiều phần tử trên ST (giống thao tác `get` ở trên)

Đầu tiên ta có thể chứng minh được:

- Độ cao của cây ST không quá $O(\log N)$.
- Tại mỗi độ sâu của cây, không có phần tử nào nằm trong 2 nút khác nhau của cây.

Thao tác loại 1

Với thao tác này, ở mỗi độ sâu của cây, ta chỉ gọi đệ quy các con của không quá 1 nút. Phân tích đoạn code trên, ta xét các trường hợp:

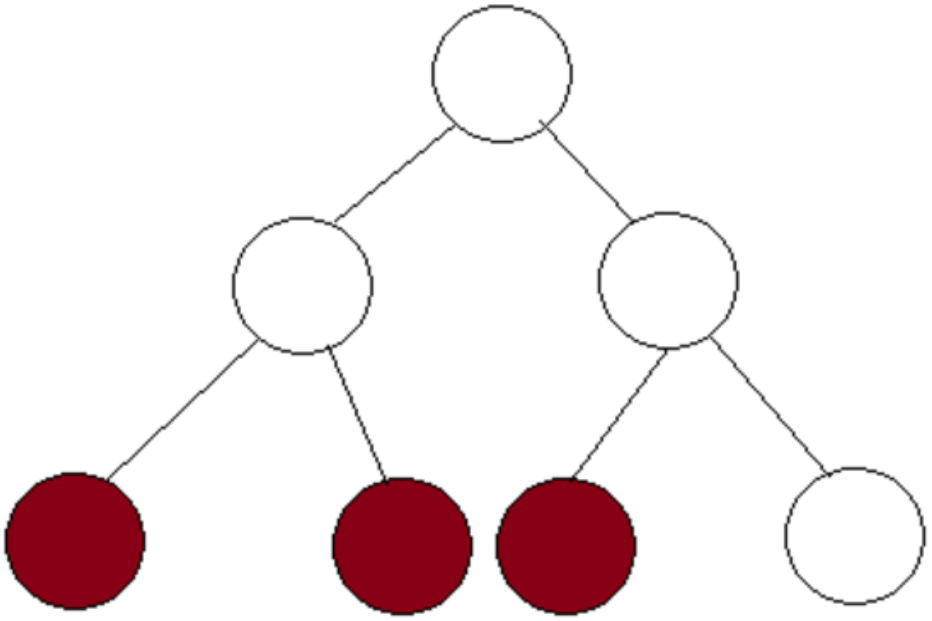
- Phần tử cần xét không nằm trong đoạn $[l, r]$ do nút id quản lý. Trường hợp này ta dừng lại, không xét tiếp.
- Phần tử cần xét nằm trong đoạn $[l, r]$ do nút id quản lý. Ta xét các con của nút `id`. Tuy nhiên chỉ có 1 con của nút `id` chứa phần tử cần xét và ta sẽ phải xét tiếp các con của nút này. Với con còn lại, ta sẽ dừng ngay mà không xét các con của nó nữa.

Do đó độ phức tạp của thao tác này không quá $O(\log N)$.

Thao tác loại 2

Với thao này, ta cũng chứng minh tương tự, nhưng ở mỗi độ sâu của cây, ta chỉ gọi hàm đệ quy với các con của không quá 2 nút.

Ta chứng minh bằng phản chứng, giả sử ta gọi đệ quy với 3 nút khác nhau của cây ST (đánh dấu màu đỏ):



Trong trường hợp này, rõ ràng toàn bộ đoạn của nút ở giữa quản lý nằm trong đoạn đang truy vấn. Do đó ta không cần phải gọi đệ quy các con của nút ở giữa. Từ đó suy ra vô lý, nghĩa là ở mỗi độ sâu ta chỉ gọi đệ quy với không quá 2 nút.

Phân tích bộ nhớ

Ta xét 2 trường hợp:

- $N = 2^k$: Cây ST đầy đủ, ở độ sâu cuối cùng có đúng 2^k lá, và các độ sâu thấp hơn không có nút lá nào (và các nút này đều có đúng 2 con). Như vậy:
  - Tầng k: có 2^k nút

    - Tầng $k-1$: có 2^{k-1} nút
    - ... Tổng số nút không quá 2^{k+1}.
- Với $N > 2^k$ và $N < 2^{k+1}$. Số nút của cây ST không quá số nút của cây ST với $N = 2^{k+1}$.

Do đó, số nút của cây cho dãy N phần tử, với $N \leq 2^k$ là không quá 2^{k+1}, giá trị này xấp xỉ $4 * N$. Bằng thực nghiệm, ta thấy dùng $4 * N$ là đủ.

1. Segment Tree cổ điển

Tại sao lại gọi là cổ điển? Đây là dạng ST đơn giản nhất, chúng ta chỉ giải quyết truy vấn update một phần tử và truy vấn đoạn, mỗi nút lưu một loại dữ liệu cơ bản như số nguyên, boolean, ...

Ví dụ 1

Bài toán: 380C-Codeforces

Tóm tắt đề

Cho một dãy ngoặc độ dài N $(N \leq 10^6)$, cho M truy vấn có dạng $l_i, r_i (1 \leq l_i \leq r_i \leq N)$. Yêu cầu của bài toán là với mỗi truy vấn tìm một chuỗi con (không cần liên tiếp) của chuỗi từ l_i đến r_i dài nhất mà tạo thành dãy ngoặc đúng.

Lời giải

Với mỗi nút(ví dụ như nút id, quản lý đoạn $[l, r]$) chúng ta lưu ba biến nguyên:

- `optimal`: Là kết quả tối ưu trong đoạn $[l, r]$.
- `open`: Số lượng dấu `(` sau khi đã xóa hết các phần tử thuộc dãy ngoặc đúng độ dài `optimal` trong đoạn.
- `close`: Số lượng dấu `)` sau khi đã xóa hết các phần tử thuộc dãy ngoặc đúng độ dài `optimal` trong đoạn.

Ta tạo 1 kiểu dữ liệu cho 1 nút của cây ST như sau:

```
struct Node {
    int optimal;
    int open;
    int close;

    Node(int opt, int o, int c) { // Khởi tạo struct Node
        optimal = opt;
        open = o;
        close = c;
    }
};
```

Và ta khai báo cây ST như sau:

```
Node st[MAXN * 4];
```

Định lý

Để tính thông tin ở nút id quản lý đoạn $[l, r]$, dựa trên 2 nút con $2 * id$ và $2 * id + 1$, ta định nghĩa 1 thao tác kết hợp 2 nút của cây ST:

```
Node operator + (const Node& left, const Node& right) {
    Node res;
    // min(số dấu "(" thừa ra ở cây con trái, và số dấu ")" thừa ra ở cây con phải)
    int tmp = min(left.open, right.close);

    // Để xây dựng kết quả tối ưu ở nút id, ta nối dãy ngoặc tối ưu ở 2 con, rồi thêm
    // min(số "(" thừa ra ở con trái, số ")" thừa ra ở con phải).
    res.optimal = left.optimal + right.optimal + tmp;

    res.open = left.open + right.open - tmp;
    res.close = left.close + right.close - tmp;

    return res;
}
```

Ban đầu ta có thể khởi tạo cây như sau:

```
void build(int id, int l, int r) {
    if (l == r) {
        // Đoạn [l, r] chỉ có 1 phần tử.
        if (s[l] == '(') st[id] = Node(0, 1, 0);
        else st[id] = Node(0, 0, 1);
        return ;
    }
    int mid = (l + r) / 2;
    build(id * 2, l, mid);
    build(id * 2 + 1, mid+1, r);

    st[id] = st[id * 2] + st[id * 2 + 1];
}
```

Để trả lời truy vấn, ta cũng làm tương tự như trong bài toán cơ bản:

```
Node query(int id, int l, int r, int u, int v) {
    if (v < l || r < u) {
        // Trường hợp không giao nhau
        return Node(0, 0, 0);
    }
    if (u <= l && r <= v) {
        // Trường hợp [l, r] nằm hoàn toàn trong [u, v]
        return st[id];
    }

    int mid = (l + r) / 2;
    return query(id * 2, l, mid, u, v) + query(id * 2 + 1, mid+1, r, u, v);
}
```

Ví dụ 2

Bài toán: SPOJ-KQUERY

Tóm đề:

- Cho một dãy số $a_i (1 \leq a_i \leq 10^9)$ có $N (1 \leq N \leq 30,000)$ phần tử
- Cho $Q (1 \leq Q \leq 200,000)$ truy vấn có dạng 3 số nguyên là l_i, r_i, k_i $(1 \leq l_i \leq r_i \leq N, 1 \leq k \leq 10^9)$. Yêu cầu của bài toán là đếm số lượng số $a_j (l_i \leq j \leq r_i)$ mà $a_j \geq k$.

Giả sử chúng ta có một mảng b với $b_i = 1$ nếu $a_i > k$ và bằng 0 nếu ngược lại. Thì chúng ta có thể dễ dàng trả lời truy vấn (i, j, k) bằng cách lấy tổng từ i đến j.

Cách làm của bài này là xử lý các truy vấn theo một thứ tự khác, để ta có thể dễ dàng tính được mảng b. Kĩ năng này được gọi là **xử lý offline** (tương tự nếu ta trả lời các truy vấn theo đúng thứ tự trong input, thì được gọi là **xử lý online**):

- Sắp xếp các truy vấn theo thứ tự tăng dần của k.
- Lúc đầu mảng b gồm toàn bộ các số 1.
- Với mỗi truy vấn, ta xem trong mảng a có những phần tử nào lớn hơn giá trị k của truy vấn trước, và nhỏ hơn giá trị k của truy vấn hiện tại, rồi đánh dấu các vị trí đó trên mảng b thành 0. Để làm được việc này một cách hiệu quả, ta cũng cần sắp xếp lại mảng a theo thứ tự tăng dần.

Ta tạo kiểu dữ liệu cho truy vấn:

```
struct Query {
    int k;
    int l, r;
};

// so sánh 2 truy vấn để dùng vào việc sort.
bool operator < (const Query& a, const Query &b) {
    return a.k < b.k;
}
```

Phần xử lý chính sẽ như sau:

```
vector< Query > queries; // các truy vấn
// Đọc vào các truy vấn
readInput();

// Sắp xếp các truy vấn
sort(queries.begin(), queries.end());

// Khởi tạo mảng id sao cho:
// a[id[1]], a[id[2]], a[id[3]] là mảng a đã sắp xếp tăng dần.

// Khởi tạo Segment Tree

for(Query q : queries) {
    while (a[id[i]] <= q.k) {
        b[id[i]] = 0;
        // Cập nhật cây Segment Tree.
        ++i;
    }
}
```

Vậy ta có thể viết hàm xây dựng cây như sau:

```
void build(int id, int l, int r) {
    if (l == r) {
        // Nút id chỉ gồm 1 phần tử
        st[id] = 1;
        return ;
    }
    int mid = (l + r) / 2;
    build(id * 2, l, mid);
    build(id * 2, mid+1, r);

    st[id] = st[id*2] + st[id*2+1];
}
```

Một hàm cập nhật khi ta muốn gán lại một vị trí bằng 0:

```
void update(int id, int l, int r, int u) {
    if (u < l || r < u) {
        // u nằm ngoài đoạn [l, r]
        return ;
    }
    if (l == r) {
        st[id] = 0;
        return ;
    }
    int mid = (l + r) / 2;
    update(id*2, l, mid, u);
    update(id*2 + 1, mid+1, r, u);

    st[id] = st[id*2] + st[id*2+1];
}
```

Và cuối cùng là thực hiện truy vấn lấy tổng một đoạn:

```
int get(int id, int l, int r, int u, int v) {
    if (v < l || r < u) {
        // Đoạn [l, r] nằm ngoài đoạn [u, v]
        return 0;
    }
    if (u <= l && r <= v) {
        // Đoạn [l, r] nằm hoàn toàn trong đoạn [u, v]
        return st[id];
    }
    int mid = (l + r) / 2;
    return get(id*2, l, mid, u, v)
        + get(id*2+1, mid+1, r, u, v);
}
```

2. Lazy Propagation

Đây là kĩ thuật được sử dụng trong ST để giảm độ phức tạp của ST với các truy vấn cập nhật đoạn.

Tư tưởng

Giả sử ta cần cập nhật đoạn $[u, v]$. Dễ thấy ta không thể nào cập nhật tất cả các nút trên Segment Tree (do tổng số nút nằm trong đoạn $[u, v]$ có thể lên đến $O(N)$). Do đó, trong quá trình cập nhật, ta chỉ thay đổi giá trị ở các nút quản lý các đoạn to nhất nằm trong $[u, v]$. Ví dụ với $N = 7$, cây Segment tree như hình minh hoạ ở đầu bài. Giả sử bạn cần cập nhật $[1, 6]$:

- Bạn chỉ cập nhật giá trị ở các nút quản lý các đoạn $[1, 4]$ và $[5, 6]$.
- Giá trị của các nút quản lý các đoạn $[1, 2]$, $[3, 4]$, $[1, 1]$, $[2, 2]$, $[5, 5]$, ... sẽ không đúng. Ta sẽ chỉ cập nhật lại giá trị của các nút này khi thật sự cần thiết (Do đó kĩ thuật này được gọi là lazy - lười biếng).

Cụ thể, chúng ta cùng xem bài toán sau:

Bài Toán

VNOJ - QMAX2

Tóm tắt đề

Cho dãy số A với N phần tử $(N \leq 50,000)$. Bạn cần thực hiện 2 loại truy vấn:

1. Cộng tất cả các số trong đoạn $[l, r]$ lên giá trị val.
2. In ra giá trị lớn nhất của các số trong đoạn $[l, r]$.

Phân tích

Thao tác 2 là thao tác cơ bản trên Segment Tree, đã được ta phân tích ở bài toán đầu tiên.

Với thao tác 1, truy vấn đoạn $[u, v]$. Giả sử ta gọi $F(id)$ là giá trị lớn nhất trong đoạn mà nút id quản lý. Trong lúc cập nhật, muốn hàm này thực hiện với độ phức tạp không quá $O(\log N)$, thì khi đến 1 nút id quản lý đoạn $[l, r]$ với đoạn $[l, r]$ nằm hoàn toàn trong đoạn $[u, v]$, thì ta không được đi vào các nút con của nó nữa (nếu không độ phức tạp sẽ là $O(N)$ do ta đi vào tất cả các nút nằm trong đoạn $[u, v]$). Để giải quyết, ta dùng kĩ thuật Lazy Propagation như sau:

- Lưu $T(id)$ với ý nghĩa, tất cả các phần tử trong đoạn $[l, r]$ mà nút id quản lý đều được cộng thêm $T(id)$.
- Trước khi ta cập nhật hoặc lấy 1 giá trị của 1 nút id' nào đó, ta phải đảm bảo ta đã "đẩy" giá trị của mảng T ở tất cả các nút tổ tiên của id' xuống id'. Để làm được điều này, ở các hàm `get` và `update`, trước khi gọi đệ quy xuống các con $2 * id$ và $2 * id + 1$, ta phải gán:
  - `T[id*2] += T[id]`
  - `T[id*2+1] += T[id]`
  - `T[id] = 0` chú ý ta cần phải thực hiện thao tác này, nếu không mỗi phần tử của dãy sẽ bị cộng nhiều lần, do ta đẩy xuống nhiều lần.

Chú ý: Bài QMAX2 này có cách cài đặt khác không sử dụng Lazy Propagation, tuy nhiên sẽ không được trình bày ở đây.

Cài đặt

Ta có kiểu dữ liệu cho 1 nút của ST như sau:

```
struct Node {
    int lazy; // giá trị T trong phân tích trên
    int val; // giá trị lớn nhất.
} nodes[MAXN * 4];
```

Hàm "đẩy" giá trị T xuống các con:

```
void down(int id) {
    int t = nodes[id].lazy;
    nodes[id*2].lazy += t;
    nodes[id*2].val += t;

    nodes[id*2+1].lazy += t;
    nodes[id*2+1].val += t;

    nodes[id].lazy = 0;
}
```

Hàm cập nhật:

```
void update(int id, int l, int r, int u, int v, int val) {
    if (v < l || r < u) {
```

```
        return ;
    }
    if (u <= l && r <= v) {
        // Khi cài đặt, ta LUÔN ĐẢM BẢO giá trị của nút được cập nhật ĐỒNG THỜI với
        // giá trị lazy propagation. Như vậy sẽ tránh sai sót.
        nodes[id].val += val;
        nodes[id].lazy += val;
        return ;
    }
    int mid = (l + r) / 2;

    down(id); // đẩy giá trị lazy propagation xuống các con

    update(id*2, l, mid, u, v, val);
    update(id*2+1, mid+1, r, u, v, val);

    nodes[id].val = max(nodes[id*2].val, nodes[id*2+1].val);
}
```

Hàm lấy giá trị lớn nhất:

```
int get(int id, int l, int r, int u, int v) {
    if (v < l || r < u) {
        return -INFINITY;
    }
    if (u <= l && r <= v) {
        return nodes[id].val;
    }
    int mid = (l + r) / 2;
    down(id); // đẩy giá trị lazy propagation xuống các con

    return max(get(id*2, l, mid, u, v),
        get(id*2+1, mid+1, r, u, v));
    // Trong các bài toán tổng quát, giá trị ở nút id có thể bị thay đổi (do ta đẩy lazy propagation
    // xuống các con). Khi đó, ta cần cập nhật lại thông tin của nút id dựa trên thông tin của các con.
}
```

Đến đây các bạn đã nắm được kiến thức cơ bản về Segment Tree. Những phần tiếp theo nói về các kiến thức nâng cao - các mở rộng của ST. Bạn nên làm nhiều bài luyện tập (tham khảo ở cuối bài) trước khi nghiên cứu tiếp.

3. Ứng dụng với cấu trúc mảng động

Trong loại bài toán này với mỗi nút của cây ta lưu lại một `vector` và một số biến khác.

Ví dụ

Cách làm online cho bài KQUERY.

Tóm tắt đề

- Cho dãy A với N phần tử. Cần trả lời Q truy vấn.
- Truy vấn: đếm số phần tử lớn hơn k trong đoạn $[l, r]$.
- Giới hạn:
  - $N \leq 30,000$
  - $A_i \leq 10^9$
  - $Q \leq 200,000$

Phân tích

- Có $\log N$ nút mà ta cần xét khi trả lời truy vấn của đoạn $[u, v]$.
- Nếu trên mỗi nút chúng ta có thể lưu lại danh sách các phần tử đó theo thứ tự tăng dần, ta có thể tìm ra kết quả ở mỗi nút bằng tìm kiếm nhị phân.

Vì thế với mỗi nút ta lưu lại một `vector` chứa các phần tử từ l đến r theo thứ tự tăng dần. Điều này có thể được thực hiện với bộ phức tạp bộ nhớ là $\mathcal{O}(N \log N)$ do mỗi phần tử có thể ở tối đa $\mathcal{O}(\log N)$ nút (độ sâu của cây không quá $\mathcal{O}(\log N)$). Ở mỗi nút cha có ta có thể gộp hai nút con vào nút cha bằng phương pháp giống như **Merge Sort** (lưu lại hai biến chạy và so sánh lần lượt từng phần tử ở hai mảng) để có thể xây dựng cây trong $\mathcal{O}(N \log N)$.

Hàm xây cây có thể được như sau:

```
void build(int id, int l, int r) {
    if (l == r) {
        // Đoạn gồm 1 phần tử. Ta dễ dàng khởi tạo nút trên ST.
        st[id].push_back(a[l]);
        return ;
    }
    int mid = (l + r) / 2;
    build(id*2, l, mid);
    build(id*2+1, mid+1, r);

    merge(st[id*2].begin(), st[id*2].end(), st[id*2+1].begin(), st[id*2+1].end(), st[id].begin());
}
```

Và hàm truy vấn có thể cài đặt như sau:

```
int get(int id, int l, int r, int u, int v, int k) { // Trả lời truy vấn (x, y, k)
    if (v < l || r < u) {
        return 0;
    }
    if (u <= l && r <= v) {
        // Đếm số phần tử > K bằng chặt nhị phân
        return st[id].size() - (upper_bound(st[id].begin(), st[id].end(), k) - st[id].begin());
    }
    int mid = (l + r) / 2;
    return get(id*2, l, mid, u, v, k) + get(id*2+1, mid+1, r, u, v, k);
}
```

Một ví dụ khác là bài Component Tree

4. Ứng dụng với cấu trúc set

Ở cấu trúc này mỗi nút chúng ta lưu một `set`, `multiset`, `hashmap`, hoặc `unodered map` và một số biến khác.

Đây là một bài toán ví dụ: Cho n vector $a_1, a_2, a_3, \ldots, a_n$ rỗng ban đầu. Chúng ta có thể thực hiện m truy vấn trên những vector này:

1. Truy vấn $A\ p\ k$ là thêm số k vào cuối vector a_p.
2. Truy vấn $C\ l\ r\ k$ là xuất ra $\sum_{i=l}^{r} count(a_i, k)$, với $count(a_i, k)$ là số lần xuất hiện của số k trong vector a_i.

Bài toán này chúng ta lưu lại mỗi nút của cây là một `multiset` s, với mỗi nút lưu số k đúng $\sum_{i=l}^{r} count(a_i, k)$ lần với độ phức tạp bộ nhớ chỉ $\mathcal{O}(q \log n)$.

Với mỗi truy vấn $C\ x\ y\ k$ chúng ta sẽ in ra tổng của tất cả dùng cây phân đoạn và truy vấn trên set trong mỗi đoạn thuộc đoạn x đến y như truy trên truy vấn cây phân đoạn bình thường.

Chúng ta sẽ không có hàm xây cây do các vector ban đầu đang là rỗng, nhưng chúng ta sẽ có thêm hàm cộng phần tử vào như sau:

```
void add(int id, int l, int r, int p, int k) {  // Thực hiện truy vấn A p k
    s[id].insert(k);
    if (l == r) return ;

    int mid = (l + r) / 2;
    if (p <= mid) add(id*2, l, mid, p, k);
    else add(id*2 + 1, mid+1, r, p, k);
}
```

Và một hàm cho truy vấn 2:

```
int ask(int id, int l, int r, int x, int y, int k) { // Trả lời C x y k
    if (y < l || r < x) return 0;
    if (x <= l && r <= y) {
        return s[id].count(k);
    }
    int mid = (l + r) / 2;
    return ask(id*2, l, mid, x, y, k) + ask(id*2+1, mid+1, r, x, y, k);
}
```

5. Ứng dụng với các cấu trúc dữ liệu khác

Cây phân đoạn còn có thể có thể sử dụng một cách linh hoạt với các cấu trúc dữ liệu khác như ở trên. Sử dụng một cây phân đoạn khác trên từng nút có thể giúp chúng ta truy vấn dễ dàng hơn trên mảng hai chiều. Trên đây cũng có thể là các loại cây như **Cây tiền tố(Trie)** hoặc cũng có thể là cấu trúc **Disjoint Set**. Sau đây mình xin giới thiệu một loại cây khác cũng sử dụng nhiều trong cây phân đoạn đó chính là **Cây Fenwick (Binary Indexed Tree)**:

Như trên mỗi nút của cây sẽ là một cây **Fenwick** và có thể một số biến khác. Dưới đây là một bài toán ví dụ:

Cho n vectors $a_1, a_2, a_3, \ldots, a_n$ rỗng ban đầu. Chúng ta cần thực hiện hai loại truy vấn:

1. Truy vấn $A\ p\ k$ là thêm số k vào đằng sau vector a_p.
2. Truy vấn $C\ l\ r\ k$ là xuất ra $\sum_{i=l}^{r} count(a_i, j)$ với $j \leq k$ với $count(a_i, j)$ là số lần xuất hiện k trong a_i.

Với bài toán này, ta cũng lưu lại ở một nút là một `vector` v chứa số k khi và chỉ khi $\sum_{i=l}^{r} count(a_i, j) \neq 0$ (độ phức tạp bộ nhớ sẽ là $\mathcal{O}(q \log n)$) (các số theo theo thứ tự tăng dần)

Đầu tiên, đọc và lưu các truy vấn lại với mỗi truy vấn loại 1 ta sẽ thêm v vào tất cả vector có chứa phần tử p. Sau đó ta tiến hành sắp xếp các truy vấn theo phương pháp **Merge Sort** đã nói ở trên và dùng hàm `unique` để loại các phần tử trùng.

Sau đó chúng ta sẽ xây dựng ở mỗi nút một cây Fenwick có độ lớn bằng độ dài vector. Sau đây là hàm thêm giá trị:

```
void insert(int id, int l, int r, int p, int k) { // Thực hiện A p k
    if (l == r) {
        v[id].push_back(k);
        return ;
    }
    int mid = (l+r) / 2;
    if (p < mid)
        insert(id*2, l, mid, p, k);
```

```
    else
        insert(id*2+1, mid+1, r, p, k);
}
```

Hàm sắp xếp sau khi đã đọc hết các truy vấn:

```
void sort_(int id, int l, int r) {
    if (l == r) return ;
    int mid = (l + r) / 2;
    sort_(id*2, l, mid);
    sort_(id*2+1, mid+1, r);

    merge(v[2 * id].begin(), v[2 * id].end(), v[2 * id + 1].begin(), v[2 * id +1].end(), v[id].begin());
}
```

Với mỗi truy vấn loại 1 ta làm như sau với mỗi nút x:

```
for(int i = a + 1; i < fen[x].size(); i += i & -i)
    fen[x][i] ++;
```

Với tất cả $v[x][a] = k$:

```
void update(int id, int l, int r, int p, int k) {
    int a = lower_bound(v[id].begin(), v[id].end(), k) - v[id].begin();
    for(int i = a + 1; i < fen[id].size(); i += i & -i)
        fen[id][i]++;

    if (l == r) return ;

    int mid = (l + r) / 2;
    if (p < mid)
        update(id*2, l, mid, p, k);
    else
        update(id*2+1, mid+1, r, p, k);
}
```

Còn lại việc tính toán truy vấn loại 2 trở nên dễ dàng hơn:

```
int ask(int id, int l, int r, int x, int y, int k) { // Trả lời C x y-1 k
    if (y < l || r < x) return 0;
    if (x <= l && r <= y) {
        int a = lower_bound(v[id].begin(), v[id].end(), k) - v[id].begin();
        int ans = 0;
        for(int i = a + 1; i > 0; i -= i & -i)
            ans += fen[id][i];
        return ans;
    }
    int mid = (l + r) / 2;
    return ask(id*2, l, mid, x, y, k)
        + ask(id*2+1, mid+1, r, x, y, k);
}
```

6. Ứng dụng trong cây có gốc

Ta có thể thấy cây phân đoạn là một ứng dụng trong mảng, vì lí do đó nếu chúng ta có thể đổi cây thành các mảng, ta có thể dễ dàng xử lý các truy vấn trên cây. Đây là tư tưởng của Heavy Light Decomposition.

Bài tập ví dụ: 396C - On Changing Tree

Gọi h_v là độ cao tương ứng của nút v.

Ta có với mỗi nút u trong cây con gốc v sau truy vấn một giá trị của nó sẽ tăng một lượng là $x + (h_u - h_v) * -k = x + k * h_v - k * h$. Kết quả của truy vấn 2 sẽ là $\sum_{i \in s}(k_i * h_{v_i} + x_i) - h_u * \sum_{i \in s} k_i$. Vì vậy ta chỉ cần tính hai giá trị là $\sum_{i \in s}(k_i * h_{v_i} + x_i)$ và $\sum_{i \in s} k_i$. Vậy với mỗi nút ta có thể lưu lại hai giá trị là $hkx = \sum x + h * k$ và $sk = \sum k$ (không cần lazy propagation do chúng ta chỉ update nút đầu tiên thỏa việc nằm trong đoạn.

Với truy vấn cập nhật:

```
void update(int id, int l, int r, int x, int k, int v) {
    if (s[v] >= r || l >= f[v]) return ;
    if (s[v] <= l && r <= f[v]) {
        hkx[id] = (hkx[id] + x) % mod;
                        int a = (1LL * h[v] * k) % mod;
                        hkx[id] = (hkx[id] + a) % mod;
                        sk[id] = (sk[id] + k) % mod;
                        return ;
    }
    int mid = (l+r) / 2;
    update(id*2, l, mid, x, k, v);
    update(id*2+1, mid+1, r, x, k, v);
}
```

Và truy vấn:

```
int ask(int id, int l, int r, int v) {
    int a = (1LL * h[v] * sk[id]) % mod;
    int ans = (hkx[id] + mod - a) % mod;
    if (l == r) return ans;
    int mid = (l+r) / 2;
    if(s[v] < mid)
        return (ans + ask(2 * id, l, mid, v)) % mod;
    return (ans + ask(2*id + 1, mid, r, v)) % mod;
}
```

7. Persistent Segment Trees

Persistent Data Structures là những cấu trúc dữ liệu được dùng khi chúng ta cần có **toàn bộ lịch sử** của các thay đổi trên 1 cấu trúc dữ liệu (CTDL).

Các bạn có thể đọc thêm ở: Persistent Data Structures

8. IT đoạn thẳng

Bài toán

Cho một tập hợp chứa các đường thẳng có dạng $ax + b$, mỗi đường thẳng được biểu diễn bằng một cặp số (a, b). Cần thực hiện hai truy vấn:

1. Thêm một đường thẳng vào tập hợp.
2. Trả lời xem tại hoành độ q, điểm nào thuộc ít nhất một đường thẳng trong tập có tung độ lớn nhất. Nói cách khác, đường thẳng (a, b) nào có $aq + b$ lớn nhất.

Để giải bài toán này, hai cách phổ biến là ứng dụng bao lồi và sử dụng cây Interval Tree lưu đoạn thẳng

9. Chặt nhị phân trên Segment tree

Nguồn: Binary Search on Segment Tree

Đây là một thao tác khá thường gặp khi dùng Segment tree, nó có tên gọi là **chặt nhị phân trên Segment tree**, tên tiếng anh là "*Binary search over/on Segment tree*", hoặc là "*Walk on Segment tree*".

Trước hết, ta cần phải nắm được kiến thức cơ bản về Segment tree và chặt nhị phân. Bạn có thể tìm hiểu thuật toán **chặt nhị phân** ở đây.

Bài toán 1

Cho một mảng các số nguyên a có n phần tử. Có q truy vấn có dạng:

- k : tìm i nhỏ nhất sao cho $a[i] \leq k$.

Cách giải

Ta nhận thấy do $a[i] \leq k$ và i nhỏ nhất, cho nên $a[j] > k$ với mọi $1 \leq j < i$.

Do đó, $min(a[1], a[2], \ldots, a[i]) = a[i]$.

Đặt $f[i] = min(a[1], a[2], \ldots, a[i])$.

Nhận xét 1: Việc tìm i nhỏ nhất sao cho $a[i] \leq k$ cũng tương ứng với việc tìm i nhỏ nhất sao cho $f[i] \leq k$.

Nhận xét 2: $f[i-1] \geq f[i]$. Nói cách khác, f là mảng không tăng.

Vậy bài toán có thể phát biểu lại như sau:

Cho một mảng các số nguyên f đã "sắp xếp" giảm dần, có q truy vấn có dạng:

- k : tìm i nhỏ nhất sao cho $f[i] \leq k$.

Rõ ràng bài toán này chỉ là bài toán chặt nhị phân cơ bản, vì mảng f đã được "sắp xếp". Tới đây ta có thể trả lời các truy vấn trong độ phức tạp $O(\log n)$. Code thì nó sẽ giống giống thế này:

```
int query(int k) {
    int l = 1, r = n, pos = -1;
    while (l <= r) {
        int mid = (l + r) / 2;
        if (f[mid] <= k)
            pos = mid, r = mid - 1;
        else
            l = mid + 1;
    }
    return pos;
}
```

Bài toán 2

Cho một mảng các số nguyên a có n phần tử. Có q truy vấn có dạng:

- $i\ x$: gán $a[i] = x$.
- k : tìm i nhỏ nhất sao cho $a[i] \leq k$

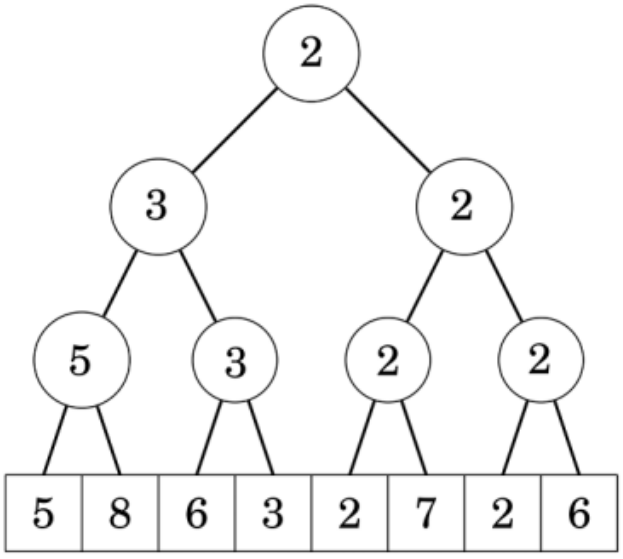
Bài toán này giống **bài toán 1**, nhưng có thêm truy vấn cập nhật phần tử, điều này làm cho mảng f bị thay đổi. Ta có thể sửa lại yêu cầu bài toán một chút, là có 3 loại truy vấn:

- $i\ x$: gán $a[i] = x$.
- k : tìm i nhỏ nhất sao cho $a[i] \leq k$
- i : tính $min(a[1], a[2], \ldots, a[i])$.

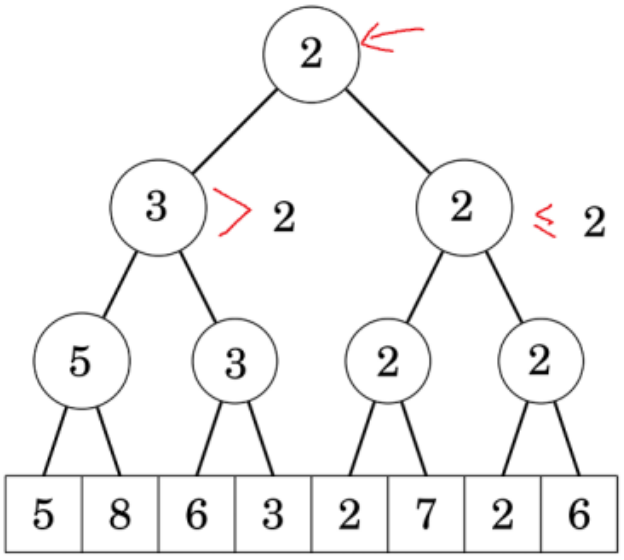
Rõ ràng truy vấn 1 và 3 có thể thực hiện bằng Segment tree với độ phức tạp $O(\log n)$, vậy thì tới đây bài toán quay về **bài toán 1**, chỉ có điều khi ta cần tính $f[i]$ thì ta phải gọi hàm trên Segment tree để lấy *min*, độ phức tạp cho việc trả lời truy vấn 2 là $O(\log^2 n)$:

```
int query(int k) {
    int l = 1, r = n, pos = -1;
    while (l <= r) {
        int mid = (l + r) / 2;
        if (getMin(1, mid) <= k)
            pos = mid, r = mid - 1;
        else
            l = mid + 1;
    }
    return pos;
}
```

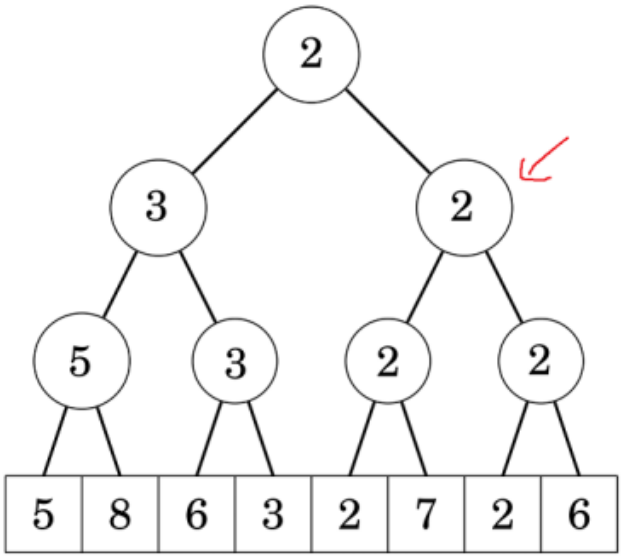
Nhưng nếu chỉ dừng ở đây thì đã không cần phải nhắc đến trong bài viết này rồi <(") . Ta nhìn một chút vào cấu trúc cây Segment tree (quản lý *min*) dưới dây:



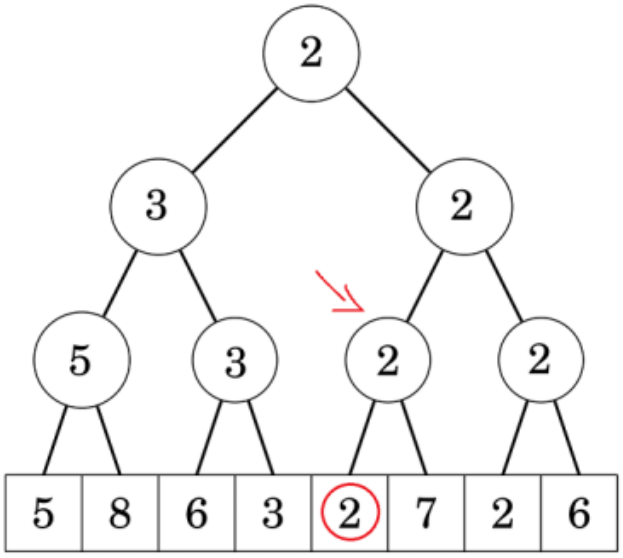
Giả sử ta cần tìm vị trí đầu tiên có giá trị không vượt quá 2. Ta đứng từ gốc, xét 2 con trái phải lần lượt có giá trị là 3 và 2:



Do ta đang cần tìm giá trị không vượt quá 2, nên ta chắc chắn kết quả không nằm trong cây con bên trái (vì *min* của cây con này là 3, suy ra mọi phần tử được quản lý bởi cây con này đều lớn hơn 2). Và do cây con phải có giá trị là 2, suy ra kết quả chắc chắn nằm cây con này, ta đệ quy xuống cây con bên trái:



Tương tự, cây con này có 2 cây con trái và phải, cả 2 đều có giá trị là 2, nghĩa là luôn tồn tại ít nhất một số có giá trị bằng 2 trong cả 2 cây con này, từ đó suy ra cả 2 cây con đều có thể chứa kết quả ta cần tìm. Nhưng do ta muốn tìm vị trí có i bé nhất, nên ta sẽ ưu tiên đi vào cây con bên trái (cây con này quản lý các vị trí nhỏ hơn các vị trí của cây con phải).



Lập luận tương tự thì ta sẽ biết được kết quả nằm ở cây con trái, lúc này cây chỉ quản lý duy nhất một phần tử nên ta có thể kết luận luôn vị trí cần tìm.

Đoạn code mẫu cho việc tìm vị trí đầu tiên không vượt quá số k có thể code như sau, lưu ý, trong code này mình xem mảng st là mảng lưu giá trị của Segment tree, 3 tham số $root, l, r$ thể hiện cho việc nút $root$ quản lý một đoạn từ $[l, r]$:

```
int query(int root, int l, int r, int k) {
    if (st[root] > k) return -1; //nếu cả đoạn [l, r] đều lớn hơn k thì không thỏa mãn
    if (l == r) return l; //khi đoạn có 1 phần tử thì đó là kết quả
    int mid = (l + r) / 2;
    if (st[root * 2] <= k) //nếu min cây con trái không vượt quá k
        return query(root * 2, l, mid, k);
    //ngược lại thì kết quả nằm ở bên cây con phải
    return query(root * 2 + 1, mid + 1, r, k)
}
//cout << query(1, 1, n, k);
```

Hàm trên có độ phức tạp là $O(\log n)$, bởi vì mỗi lần đệ quy chỉ gọi ra một hàm khác (từ một nút chỉ đi qua một nút khác), và số lần gọi đệ quy chính bằng độ cao của Segment tree. Tới đây ta đã xong **bài toán 2**.

Lưu ý là, với các bài toán mà truy vấn cập nhật là một đoạn (thay vì một phần tử như **bài toán 2**), thì việc cài đặt hàm $query$ ở trên vẫn không đổi, chỉ có thêm vào *lazy* trước khi xét 2 cây con trái phải, mình xin giành cho bạn đọc vậy.

Bài toán 3:

Cho một mảng các số nguyên a có n phần tử. Có q truy vấn có dạng:

- $i\ x$: gán $a[i] = x$.
- $L\ k$: tìm i nhỏ nhất sao cho $L \leq i$ và $a[i] \leq k$

Bài toán này khó hơn **bài toán 2** một chút, đó là có thêm một cận dưới của i (thay vì tìm i bé nhất, thì ta cần tìm i bé nhất nhưng lớn hơn một số nào đó), ta có thể thay đổi code một tí như sau:

```
int query(int root, int l, int r, int lowerbound, int k) {
    if (st[root] > k) return -1; //nếu cả đoạn [l, r] đều lớn hơn k thì không thỏa mãn
    if (r < lowerbound) return -1; //ta chỉ xét những vị trí không nhỏ hơn lowerbound
    if (l == r) return l; //khi đoạn có 1 phần tử thì đó là kết quả
    int mid = (l + r) / 2;
    int res = -1;
```

```
    if (st[root * 2] <= k) //nếu min cây con trái không vượt quá k
        res = query(root * 2, l, mid, lowerbound, k);
    //nếu cây con trái không tìm được kết quả <=> min nằm ngoài lowerbound
    //thì ta sẽ tìm kết quả ở cây con phải
    if (res == -1)
        res = query(root * 2, mid + 1, r, lowerbound, k);
    return res;
}
//cout << query(1, 1, n, l, k);
```

Code này có một chút lạ, khác so với code ở **bài toán 2** một chút, ở **bài toán 2**, thì mỗi lần đệ quy chỉ thăm duy nhất một con trái hoặc phải, nhưng ở code mới này thì một lần đệ quy có thể phải thăm cả 2 con, lý do là vì có thể một cây con nó có *min* không vượt quá k, nhưng vị trí đạt *min* nó có thể nhỏ hơn $lowerbound$, vì thế ta phải tìm ở cây con khác.

Để đánh giá độ phức tạp code trên thì hơi rườm rà một chút, nhưng nó vẫn là $O(\log n)$. Đại ý là ta có thể chứng minh số lần mà $r < lowerbound$ sẽ không quá $O(\log n)$.

Bài tập áp dụng:

- VNOJ - Educational Segment Tree Contest
- VNOJ - QMAX
- VNOJ - NKLINEUP
- VNOJ - GSS
- VNOJ - LITES
- VNOJ - DQUERY
- VNOJ - KQUERY
- FREQUENT
- VNOJ - KQUERY2
- GSS2
- GSS3
- MULTQ3
- POSTERS
- PATULJCI
- New Year Domino
- Copying Data
- DZY Loves Fibonacci Numbers
- FRBSUM

Đọc thêm:

- Cấu trúc dữ liệu đặc biệt - Đoàn Mạnh Hùng
- Cấu trúc dữ liệu đặc biệt - Nguyễn Minh Hiếu

Các nguồn tham khảo:

- Codeforces

- Một số vấn đề đáng chú ý trong môn Tin học

Like 6 Share Save to Facebook

9 Comments Sort by Oldest

Add a comment...

Minh Tuan Nguyen

Mình có chút góp ý:
- Ở ví dụ bài QMAX đoạn update cần phải sửa:
Ko có điều kiện dừng khi l = r, và khi l = r thì ST[id] = v rồi sau đó mới cập nhật ST[id] = max(ST[id * 2], ST[id * 2 + 1]), lấy cập nhật như vậy giá trị cũ của a[i] vẫn còn trong cây
- Ở ví dụ 380C cf thì cần sửa chỗ: res.optimal = left.optimal + right.optimal + 2 * tmp;

Like · Reply · 5y

Trung Nguyen

cảm ơn bạn, mình đã sửa lại bài viết

Like · Reply · 1 · 5y

Toàn Đồng Xuân

Em chào thầy!

Like · Reply · 1y

Quang Nguyen

Cho mình hỏi là trong bài KQUERY2, thao tác cập nhật được thực hiện như thế nào?

Like · Reply · 5y

Chi Dung Nguyen Duy

update mất log^2 thì phải, mình đoán là đến nút chứa đoạn l==r và l= vị trí cần update, mình đổi giá trị phần tử duy nhất của vector là số cần cập nhật, rồi lúc đi lên dùng merge để cập nhật lại vector cho các nút cha

Like · Reply · 5y

Quang Nguyen

Chi Dung Nguyen Duy Nếu thực hiện cập nhật như vậy thì độ phức tạp thao tác cập nhật sẽ là O(N) (Do phải cập nhật vector quản lý toàn bộ cây).

Like · Reply · 5y

Chi Dung Nguyen Duy

ừ nhỉ, mình xl ^^

Like · Reply · 5y

Show 2 more replies in this thread

Tung Trong

bài KQUERY2 update sao vậy bạn ?

Like · Reply · 5y

Võ Minh Thiên Long

bài QMAX đề hình như là tăng đoạn u, v lên k chứ đâu phải gán đoạn u, v bằng k đâu ạ

Like · Reply · 5y

Thành Đô Nguyễn

Cho em hỏi không biết ST có đúng là IT không hay là 2 cái hoàn toàn khác nhau ạ ?

Em có đọc trên:

https://en.wikipedia.org/wiki/Interval_tree
http://www.geeksforgeeks.org/interval-tree/

Với cả trong quyển tài liệu chuyên tin quyển 2 cũng có nhắc đến IT, nhưng hình cấu trúc cây nó hoàn toàn khác với Segmented Tree ở trên (thay vì việc chia đôi khoảng l, r thành 2 bên l, mid và mid+1, r) thì nó lại là một BST (trong đấy cũng có nhắc đến việc IT là ứng dụng của BST)

Like · Reply · 4y

Trung Nguyen

Ngày xưa lúc Segment Tree du nhập vào VN, thì được gọi tên là Interval Tree, do đó trong cộng đồng người VN thì thường Interval Tree == Segment Tree.

Còn thuật ngữ chuẩn thì Interval Tree là cái trong link wiki của bạn.

Like · Reply · 3y

Load 4 more comments