**HỘI THẢO KHOA HỌC**

**CÁC TRƯỜNG THPT CHUYÊN**

**KHU VỰC DUYÊN HẢI VÀ ĐỒNG BẰNG BẮC BỘ**

**NĂM 2021**

**Môn: Tin học**

**CHUYÊN ĐỀ**

**CÂY PHÂN ĐOẠN**

**Segment Tree**

**Tháng 9/2021**

**Mục lục**

[1. Mở đầu 4](#_Toc79072397)

[**2. Dạng đơn giản nhất của cây phân đoạn** 4](#_Toc79072398)

[3. Cấu trúc cây phân đoạn 5](#_Toc79072399)

[**4. Xây dựng** 6](#_Toc79072400)

[**5. Truy vấn** 7](#_Toc79072401)

[**6. Cập nhật phần tử** 8](#_Toc79072402)

[**7. Cài đặt** 9](#_Toc79072403)

[**8. Bộ nhớ được sử dụng để cài đặt** 11](#_Toc79072404)

[**9. Các bài toán của cây phân đoạn** 12](#_Toc79072405)

[**Tìm số lớn nhất** 12](#_Toc79072406)

[**Tìm số lớn nhất và số lần xuất hiện của nó** 12](#_Toc79072407)

[**Tìm USCLN và BSCNN** 14](#_Toc79072408)

[**Đếm số lượng số 0, tìm kiếm vị trí số 0 thứ k** 15](#_Toc79072409)

[**Tìm kiếm tổng tiền tố lớn hơn hoặc bằng một số nhất định** 16](#_Toc79072410)

[**Tìm kiếm phần tử đầu tiên lớn hơn một lượng cho trước** 16](#_Toc79072411)

[**Tìm các phân đoạn có tổng lớn nhất** 17](#_Toc79072412)

[**10. Lưu trữ mảng trong đỉnh** 21](#_Toc79072413)

[**Tìm phần tử nhỏ nhất, phần tử lớn hơn một số cho trước (Không có truy vấn sửa đổi)** 22](#_Toc79072414)

[**Tìm phần tử lớn hơn hoặc bằng một giá trị nào đó (Có truy vấn sửa đổi)** 24](#_Toc79072415)

[**Tìm phần tử lớn hơn hoặc bằng một giá trị nào đó. Tăng tốc với kỹ thuật “Xếp chồng phân đoạn - fractional cascading”.** 25](#_Toc79072416)

[**11. Các biến thể của cây phân đoạn** 29](#_Toc79072417)

[**12. Cập nhật phân đoạn (Kỹ thuật lan truyền lười nhác)** 29](#_Toc79072418)

[**Tăng thêm giá trị** 29](#_Toc79072419)

[**Gán giá trị của một phân đoạn** 31](#_Toc79072420)

[**Tăng thêm trong phân đoạn, tìm số lớn nhất** 34](#_Toc79072421)

[**13. Cây phân đoạn nâng cao** 36](#_Toc79072422)

[**Cây phân đoạn 2D đơn giản** 36](#_Toc79072423)

[**Nén cây 2D** 41](#_Toc79072424)

[**14. Ghi lại lịch sử sửa đổi cây phân đoạn (Cây phân đoạn liên tục - Persistent Segment Tree)** 42](#_Toc79072425)

[**Tìm kiếm k-số nhỏ nhất trong một đoạn** 45](#_Toc79072426)

[**15. Cây phân đoạn ngầm định** 48](#_Toc79072427)

[**16. Bài tập áp dụng** 51](#_Toc79072428)

[17. Tài liệu tham khảo 52](#_Toc79072429)

# 1. Mở đầu

Cây phân đoạn là cấu trúc dữ liệu cho phép thực hiện các truy vấn đoạn trên mảng một cách hiệu quả, trong khi vẫn cho phép sửa đổi mảng. Các bài toán có thể giải quyết được có thể bao gồm: Tính tổng các phần tử liên tiếp trong một đoạn [l,r]. Hoặc tìm phần tử lớn nhất, nhỏ nhất trong một đoạn như vậy với độ phức tạp O(logn). Giữa lúc trả lời các câu hỏi như vậy, cây phân đoạn cũng cho phép thay đổi mảng với các thao tác như thay đổi giá trị của một phần tử hoặc thậm chí thay đổi giá trị của cả một đoạn (Ví dụ như gán một giá trị của đoạn [l,r], hoặc cộng thêm vào các phần tử trong đoạn [l,r] một giá trị nào đó).

Nói chung, cây phân đoạn là một cấu trúc dữ liệu linh động và có thể giải quyết được rất nhiều bài toán. Ngoài ra, cũng có thể áp dụng cấu trúc cây phân đoạn để trả lời được các câu hỏi phức tạp hơn là việc tìm tổng, tìm giá trị lớn nhất, nhỏ nhất trong một đoạn. Đặc biệt, cây phân đoạn cũng có thể được tổng quát hóa thành cây phân đoạn có cấu trúc phức tạp hơn với dữ liệu lớn hơn. Ví dụ, cây phân đoạn hai chiều, ta có thể giải các bài toán tính tổng hoặc tìm min, max trên một phần con của một mảng hai chiều nhất định chỉ với chi phí về mặt thời gian là O(log2n).

Một tính chất quan trọng của cây phân đoạn đó là chỉ cần bộ nhớ tuyến tính, nó chỉ cần 4n đỉnh để thể hiện được mảng có kích thước n.

**2. Dạng đơn giản nhất của cây phân đoạn**

Để bắt đầu, chúng ta hãy xét dạng đơn giản nhất của cây phân đoạn, ta cần trả lời các truy vấn tính tổng một cách hiệu quả. Bài toán gốc là: *Cho một mảng a[0…n−1]. Hãy trả lời m truy vấn tính tổng trên các đoạn [l,r]. Trong khi thực hiện tính tổng, có thể thay đổi các giá trị của một số phần tử rồi mới thực hiện truy vấn tính tổng*. Cây phân đoạn cho phép ta thực hiện các truy vấn nói trên trong thời gian chỉ O(logn).

# 3. Cấu trúc cây phân đoạn

Vậy, cấu trúc cây phân đoạn là gì?

Chúng ta bắt đầu với việc tính toán tổng các giá trị các phần tử trong toàn mảng  a[0…n - 1] và lưu trữ nó lại sau đó chia nó thành hai, một nửa sẽ tính tổng trong đoạn a[0…n / 2] và nửa kia sẽ tính tổng các phần tử trong đoạn còn lại a[n / 2 + 1… n] và lưu trữ lại. Đến lượt hai nửa này, chúng cũng lại được chia thành hai, tổng của chúng cũng được lưu trữ lại. Quá trình này lặp lại cho đến khi mảng phân đoạn có kích thước chỉ còn một phần tử. Nói cách khác, chúng ta bắt đầu với mảng a[0… n - 1], chia đôi đoạn này ra (Đến lúc đoạn này chỉ có 1 phần tử) rồi gọi lại thủ tục nói trên cho hai nửa, kết quả tính toán sẽ được lưu trữ lại.

Chúng ta có thể nói rằng, cây phân đoạn là một cây nhị phân có gốc tính tổng các giá trị trong đoạn a[0 … n - 1], mỗi đỉnh (trừ đỉnh lá) có đúng hai đỉnh con. Đây là lí do tại sao người ta gọi đây là cây phân đoạn, mặc dù trong hầu hết các cách triển khai, cây phân đoạn không được xây dựng một cách rõ ràng.

Đây là hình ảnh đại diện trực quan cho cây phân đoạn như vậy trong mảng a = [1 , 3 , - 2 , 8 , - 7]:

|  |
| --- |
| "Cây phân đoạn tổng" |
| Hình 1: Cây phân đoạn đơn giản |

Từ mô tả ngắn gọn như trên, ta có thể thấy rằng cây phân đoạn chỉ cần yêu cầu số lượng đỉnh trong đoạn tuyến tính. Mức đầu tiên của cây chứa một nút duy nhất là nút gốc, mức thứ hai là hai đỉnh, mức thứ ba là 4 đỉnh, … cho đến khi số đỉnh của cây đủ lớn để thể hiện hết n phần tử. Do đó, số lượng các đỉnh cần thiết là: 1+2+4+⋯+2⌈log2n⌉=2⌈log2n⌉+1<4n.

Điều đáng chú ý là: Bất cứ khi nào, n không phải là lũy thừa của 2 thì cây phân đoạn được triển khai có thể không được lấp đầy. Chúng ta có thể thấy rõ qua hình ảnh. Hiện tại, chúng ta có thể quên thực tế này, nhưng nó sẽ là một lưu ý quan trọng sau này trong phần cài đặt.

Chiều cao của cây phân đoạn là O(logn) bởi vì khi đi từ nút gốc lên nút lá, đoạn của các đoạn giảm đi một nửa.

**4. Xây dựng**

Trước khi triển khai cây phân đoạn chúng ta cần khẳng định hai ý:

+ Các nút của cây phân đoạn sẽ lưu trữ tổng các phần tử của mảng trong đoạn [l , r].

+ Hai nút anh em trên một cây phân đoạn có thể được hợp nhất thành nút thấp hơn. Ví dụ: trong một cây phân đoạn tổng, hai nút tương ứng với các đoạn [l1…r1] và [l2…r2] có thể được hợp nhất thành một nút tương ứng với đoạn một [l1…r2] bằng cách cộng các giá trị của hai nút.

Chú ý rằng, trong một cây phân đoạn, một nút lá tương ứng với một phần tử trong mảng ban đầu. Nó xuất hiện tại mức cao nhất của cây phân đoạn.

Bây giờ, để xây dựng cây phân đoạn, ta bắt đầu từ đỉnh ở mức cao nhất (các đỉnh lá) và gán cho chúng các giá trị tương ứng. Trên cơ sở các giá trị này, ta có thể tính được giá trị của các nút thấp hơn, đến lượt chúng ta lại có thể tính toán giá trị của các nút trước đó, quá trình này lặp lại cho đến khi ta đạt đến nút gốc.

Hàm đệ quy sẽ được sử dụng. Một đỉnh từ gốc đến lá có thể được xây dựng như sau:

+ Xây dựng các hàm đệ quy tính giá trị của hai đỉnh con

+ Hợp nhất các giá trị tính toán được từ hai đỉnh con này.

Chúng ta có giá trị của nút gốc và do đó có thể tính toán giá trị của toàn bộ cây phân đoạn.

Độ phức tạp của giải thật là O(n), nếu coi thời gian hợp hai giá trị của các nút con không đáng kể thì độ phức tap của bài toán chính là số nút của cây phân đoạn.

**5. Truy vấn**

Bây giờ chúng ta bắt đầu phải trả lời các truy vấn tính tổng, với cây phân đoạn, ta có thể trả lời các câu truy vấn dạng này với độ phức tạp O(logn).

Để trả lời các truy vấn này, ta duyệt qua các cây phân đoạn. Giả sử ta cần trả lời truy vấn tính tổng trên đoạn a[tl…tr]. Các trường hợp có thể sảy ra:

+ Trường hợp đơn giản nhất là đoạn cần truy vấn chính là một nút trên cây phân đoạn. (tức là a[l … r] = a[tl …tr]). Trong trường hợp này, ta chỉ cần in ra giá trị đang được lưu trữ ở nút tương ứng.

+ Trường hợp thứ hai là truy vấn có thể rơi vào miền con trái hoặc miền con phải của nút đang xét. Trong trường hợp này, chúng ta cũng sẽ áp dụng giải thuật đệ quy với các nút con.

+ Và sau cùng là trường hợp phức tạp nhất, miền truy vấn sẽ không hoàn toàn nằm trong miền con trái và (hoặc) không nằm hoàn toàn trong miền con phải của một nút. Trong trường hợp này, không còn cách nào khác là ta phải thực hiện hai công việc đệ quy cho mỗi con, sau đó đưa ra câu trả lời cuối cùng là kết quả của truy vấn.

Vì vậy, khi xử lí các truy vấn các hàm đệ quy sẽ được gọi, trong trường hợp các đoạn truy vấn không rơi hoàn toàn vào đoạn quản lí của nút hiện tại thì nó sẽ gọi đệ quy đến các nút con.

Quá trình duyệt sẽ bắt đầu từ nút gốc.

Chúng ta sẽ minh họa, với mảng a = [1 , 3 , - 2 , 8 , - 7] và đây chúng ta sẽ tính tổng S =  . Các đỉnh được gọi sẽ được tô màu xanh. Kết quả được tính là −2+1=−1.

|  |
| --- |
| "Sum Segment Tree Query" |
| Hình 2: Truy vấn trên cây phân đoạn |

Vậy tại sao độ phức tạp của của các truy vấn là O(logn). Chúng ta xem xét số lượng nút phải truy cập ứng với từng truy vấn. Hóa ra, số nút được truy cập đối với một truy vấn sẽ không vượt quá 4n và chiều cao của cây thì không vượt quá O(logn).

**6. Cập nhật phần tử**

Bây giờ chúng ta muốn sửa đổi một phần tử nào đó trong mảng, giả sử như việc gán một giá trị a[i] = x và kéo theo việc sửa đổi cây phân đoạn theo mảng mới đã sửa đổi.

Truy vấn kiểu này dễ hơn truy vấn kiểu tính tổng, Mỗi cấp của cây phân đoạn tạo thành một phân vùng của mảng. Do đó mỗi phần tử a[i] chỉ tham gia vào một phân vùng riêng biệt trên cùng một chiều cao, chính vì vậy việc cập nhật giá trị của một phần tử cũng chỉ cần thực hiện trong thời gian O(logn).

Dễ thấy, để cập nhật giá trị của phần tử có thể sử dụng hàm đệ quy. Hàm đệ quy này tính toán với giá trị của nút hiện tại, và như vậy nó gọi đệ quy đến một trong hai nút con của nó. Có thể mô tả trong hình như sau. Ở đây ta thực hiện cập nhật a [2] = 3. Các đỉnh màu xanh lá cây là các đỉnh mà ta thăm và cập nhật.

|  |
| --- |
| "Cập nhật cây phân đoạn tổng" |
| Hình 3: Thao tác cập nhật giá trị |

**7. Cài đặt**

Ta sẽ sử dụng một mảng có kích thước 4n để xây dựng một cây phân đoạn biểu diễn một mảng có kích thước n.

Khai báo có thể như sau:

**int n, t[4\*MAXN];**

Quy trình xây dựng cây phân đoạn có thể được mô tả như sau: Từ một mảng a[n] như đã nói, xây dựng một hàm đệ quy bulid() với các tham số được mô tả như sau: v là chỉ số của đỉnh hiện tại trên cây, tl và tl là hai giới hạn trái và giới hạn phải của đoạn [l,r] cần tính tổng. Trong khi kết quả cuối cùng để tính tổng trong mảng a[0,n-1] là giá trị trên cây tại v =0 và tl = 0 và tr = n-1. (Code xem trong bài SUM)

**void build(int a[], int v, int tl, int tr) {**

**if (tl == tr) {**

**t[v] = a[tl];**

**} else {**

**int tm = (tl + tr) / 2;**

**build(a, v\*2, tl, tm);**

**build(a, v\*2+1, tm+1, tr);**

**t[v] = t[v\*2] + t[v\*2+1];**

**}**

**}**

Để trả lời các truy vấn tính tổng, ta cũng sẽ sử dụng một hàm đệ quy, hàm này sẽ thực hiện công việc tính tổng các giá trị trong đoạn [tl,tr] của cây nhị phân gốc v quản lí, hàm sẽ kết thúc nếu như tl>tr.

**int sum(int v, int tl, int tr, int l, int r) {**

**if (l > r)**

**return 0;**

**if (l == tl && r == tr) {**

**return t[v];**

**}**

**int tm = (tl + tr) / 2;**

**return sum(v\*2, tl, tm, l, min(r, tm))**

**+ sum(v\*2+1, tm+1, tr, max(l, tm+1), r);**

**}**

Cuối cùng là thao tác cập nhật, Hàm sẽ nhận các thông tin về đoạn được cây phân đoạn gốc v quản lí, đồng thời về vị trí pos trên mảng a[0,n-1] cũng như giá trị new\_val mà sẽ để gán cho phần tử trong mảng.

**void update(int v, int tl, int tr, int pos, int new\_val) {**

**if (tl == tr) {**

**t[v] = new\_val;**

**} else {**

**int tm = (tl + tr) / 2;**

**if (pos <= tm)**

**update(v\*2, tl, tm, pos, new\_val);**

**else**

**update(v\*2+1, tm+1, tr, pos, new\_val);**

**t[v] = t[v\*2] + t[v\*2+1];**

**}**

**}**

**8. Bộ nhớ được sử dụng để cài đặt**

Hều hết mọi người đều sử dụng kích thước mảng tĩnh để triển khai. Nếu nhìn vào mảng t bạn sẽ thấy các nút của nó tuân theo cách đánh số các nút theo một trình tự của phép duyệt BFS; Một nút con của cây tại đỉnh v sẽ có tương ứng hai nút con ở tại đỉnh thức 2v và 2v+1. Điều này chỉ đúng với các cây có số đỉnh là lũy thừa của 2. Với các cây còn lại, sẽ có một số đỉnh được sử dụng, các đỉnh còn lại thì để trống.

Để giảm bớt được số lượng đỉnh phải sử dụng, chúng ta có một cách triển khai khác. Ta đánh lại các đỉnh của cây theo thứ tự của một đường đi ngang qua Euler (theo thứ tự trước) và ta có thể viết tất cả các đỉnh này bên cạnh nhau.

Hãy nhìn vào một đỉnh tại vị trí v, nó lưu giá trị về phân đoạn [l , r], và ta có mid=(l + r) /2. Rõ ràng là con bên trái sẽ có chỉ số v + 1. Con trái chịu trách nhiệm phân đoạn[l , mid], tức là tổng cộng sẽ có 2 \* (m i d- l + 1) – 1 đỉnh trong cây con bên trái của con. Do đó, ta có thể tính toán chỉ mục của con bên phải của v. Chỉ số sẽ là v + 2 ∗ (mid- l + 1). Bằng cách đánh số này, chúng ta sẽ giảm bộ nhớ cần thiết xuống 2n.

**9. Các bài toán của cây phân đoạn**

Cây phân đoạn là một cấu trúc linh hoạt, nó cho phép giải một số bài toán khác trên cơ sở bài toán tính tổng nói trên.

**Tìm số lớn nhất**

Hãy để ý một chút, nếu ta thay đổi một chút chương trình ở trên ta có thể thu được cây phân đoạn quản lý các giá trị tối đa trên mảng a[0,n-1].

Và tất nhiên chương trình cũng có thể được thay đổi để tính giá trị nhỏ nhất.

**Tìm số lớn nhất và số lần xuất hiện của nó**

Bài toán lần này cũng tương tự như bài toán lần trước, bên cạnh việc tìm giá trị lớn nhất ta sẽ phai tìm thêm số lần xuất hiện của nó nữa.

Để giải được bài toán này, ta lưu trữ một cặp số ở mỗi đỉnh của cây phân đoạn đó là giá trị lớn nhất và số lần xuất hiện của nó trong đoạn mà đỉnh đó quản lí.

Ta có thể thấy, việc này cũng có thể thực hiện với cây phân đoạn và các thao tác chỉ cần cải tiến một chút từ bài toán tìm giá trị lớn nhất (Code có thể tham khảo trong bài MAX AND NUMBER OF occurrences)

**pair<int, int> t[4\*MAXN];**

**pair<int, int> combine(pair<int, int> a, pair<int, int> b) {**

**if (a.first > b.first)**

**return a;**

**if (b.first > a.first)**

**return b;**

**return make\_pair(a.first, a.second + b.second);**

**}**

**void build(int a[], int v, int tl, int tr) {**

**if (tl == tr) {**

**t[v] = make\_pair(a[tl], 1);**

**} else {**

**int tm = (tl + tr) / 2;**

**build(a, v\*2, tl, tm);**

**build(a, v\*2+1, tm+1, tr);**

**t[v] = combine(t[v\*2], t[v\*2+1]);**

**}**

**}**

**pair<int, int> get\_max(int v, int tl, int tr, int l, int r) {**

**if (l > r)**

**return make\_pair(-INF, 0);**

**if (l == tl && r == tr)**

**return t[v];**

**int tm = (tl + tr) / 2;**

**return combine(get\_max(v\*2, tl, tm, l, min(r, tm)), get\_max(v\*2+1, tm+1, tr, max(l, tm+1), r));**

**}**

**void update(int v, int tl, int tr, int pos, int new\_val) {**

**if (tl == tr) {**

**t[v] = make\_pair(new\_val, 1);**

**} else {**

**int tm = (tl + tr) / 2;**

**if (pos <= tm)**

**update(v\*2, tl, tm, pos, new\_val);**

**else**

**update(v\*2+1, tm+1, tr, pos, new\_val);**

**t[v] = combine(t[v\*2], t[v\*2+1]);**

**}**

**}**

**Tìm USCLN và BSCNN**

Trong bài toán tiếp theo, ta muốn tìm USCLN và BSCNN trong một đoạn nào đó của mảng được quản lí.

Ta có thể thấy rằng USCLN có thể làm gần như là bài toán tính tổng, cho nên việc tìm USCLN là hoàn toàn có thể dựa trên bài toán này, Thêm nữa BSCNN thì có thể dựa trên công thức BSCNN(x,y) = (x\*y)/USCLN(x,y). Hoặc dựa trên bài toán nói trên, ta lưu một cặp số vào mỗi đỉnh.

**Đếm số lượng số 0, tìm kiếm vị trí số 0 thứ k**

Trong phần này, chúng ta muốn tìm số lượng số 0 trong một đoạn nhất định và tìm thêm thứ tự của phần tử có giá trị bằng 0 một cách nhanh chóng

Lần này ta có thể thay đổi ý nghĩa của cây t một chút. Thay vì tính lưu trữ tổng ta có thể lấy ý nghĩa của đỉnh s trên cây t là lưu trữ số lượng số 0 trong đoạn [l,r]. Ta có thể thấy các thao tác build và update và queryZero cũng khá giống như là trên cây phân đoạn tính tổng. Như vậy, ta có thể giải quyết được phần đầu tiên của bài toán.

Bây giờ chúng ta tập trung vào giải quyết phần, tìm vị trí xuất hiện của số 0 thứ k nào đó. Ta có thể thấy rằng, trên cây phân đoạn đã được xây dựng, nếu bắt đầu từ nút gốc, với mỗi nút lưu số chữ số 0 trong đoạn mà nút đó quản lí, vậy ta có thể tìm đệ quy trên nút con trái và con phải.

Ở đây kết quả sẽ cho ra -1 nếu như không tìm được theo yêu cầu đầu vào. (Code có thể tham khảo trong bài NUMBEROFZERO)

**int find\_kth(int v, int tl, int tr, int k) {**

**if (k > t[v])**

**return -1;**

**if (tl == tr)**

**return tl;**

**int tm = (tl + tr) / 2;**

**if (t[v\*2] >= k)**

**return find\_kth(v\*2, tl, tm, k);**

**else**

**return find\_kth(v\*2+1, tm+1, tr, k - t[v\*2]);**

**}**

**Tìm kiếm tổng tiền tố lớn hơn hoặc bằng một số nhất định**

Bài toán được cho như sau: Cho một số nguyên x, chúng ta cần tìm ra chỉ số i sao cho tổng i số nguyên đầu tiên trong mảng a lớn hơn hoặc bằng x (với giả sử ràng mảng a chỉ chứa các số nguyên không âm)

Bài toán này có thể giải bằng phương pháp tìm kiếm nhị phân, Tuy nhiên ở đây ta có thể có một phương pháp khác. Ta có thể giải quyết bài toán nói trên theo ý tưởng về ứng dụng cây phân đoạn tính tổng, sau đó tìm kiếm trên cây phân đoạn này, mỗi lần di chuyển sang phải hay trái thì ta sẽ di chuyển phụ thuộc vào tổng con bên trái. Với cách làm như thế này thì ta chỉ mất thời gian là  O(logn) (Code có thể tham khảo trong bài PREFIX).

**Tìm kiếm phần tử đầu tiên lớn hơn một lượng cho trước**

Bài toán được cho như sau: Cho một số nguyên x, và đoạn [l…r]. Tìm ra chỉ số i nhỏ nhất trong đoạn [l…r] nói trên sao cho a[i] lớn hơn hoặc bằng x.

Với cách giải thông thường, ta có thể giải bài toán này bằng tìm kiếm tuần tự.

Tuy vậy, ta có thể ứng dụng cây phân đoạn ở đây, ta cũng di chuyển giữa các nút tương tự như mục ở trên. Độ phức tạp của thuật toán sẽ là O(logn). (Code xem trong bài AMOUNT)

**int get\_first(int v, int lv, int rv, int l, int r, int x) {**

**if(lv > r || rv < l) return -1;**

**if(l <= lv && rv <= r) {**

**if(t[v] <= x) return -1;**

**while(lv != rv) {**

**int mid = lv + (rv-lv)/2;**

**if(t[2\*v] > x) {**

**v = 2\*v;**

**rv = mid;**

**}else {**

**v = 2\*v+1;**

**lv = mid+1;**

**}**

**}**

**return lv;**

**}**

**int mid = lv + (rv-lv)/2;**

**int rs = get\_first(2\*v, lv, mid, l, r, x);**

**if(rs != -1) return rs;**

**return get\_first(2\*v+1, mid+1, rv, l ,r, x);**

**}**

**Tìm các phân đoạn có tổng lớn nhất**

Ở đây, một lần nữa chúng ta được cho một đoạn [l … r] với mỗi truy vấn. Lần này chúng ta cần tìm đoạn [l’ … r’] chứa trong đoạn [l … r] nói trên và tổng các phần tử đạt cực đại. Cũng như trước đây, có thể cập nhật một giá trị của một phần tử nào đó trong mảng và các phần tử trong mảng a có thể âm cũng như phân đoạn tối ưu có thể không xác định được (Ví dụ như trường hợp, tất cả các phần tử trong mảng đều là số âm)

Bài toán này có thể giải bằng cây phân đoạn, tuy nhiên cây phân đoạn trong trường hợp này không thông thường. Lần này, ta sẽ lưu bốn giá trị tại mỗi đỉnh: Tổng của đoạn, tổng tiền tố lớn nhất, tổng hậu tố lớn nhất, và tổng của phân đoạn lớn nhất trong đó. Nói cách khác, đối với mỗi phân đoạn, câu trả lời sẽ được lưu sẵn, việc lấy kết quả chỉ phụ thuộc vào việc chọn biên.

Làm thế nào để xây dựng một cây phân đoạn như vậy? Một lần nữa, chúng ta có thể triển khai theo hướng đệ quy. Trước tiên, chúng ta có thể tính toán tất cả bốn giá trị của các nút con trái và phải của nút hiện tại rồi kết hợp lại chúng ta sẽ có các giá trị của nút hiện tại. Lưu ý rằng, các giá trị đang được lưu trữ ở đỉnh hiện tại là:

+ Trường hợp giá trị lớn nhất rơi vào nút con trái: Phân đoạn có tổng lớn nhất nằm hoàn toàn trong phân đoạn bên trái.

+ Trường hợp giá trị lớn nhất rơi vào nút con phải: Phân đoạn có tổng lớn nhất nằm hoàn toàn trong phân đoạn bên phải.

+ Trong trường hợp còn lại nghĩa là phân đoạn tối đa nằm trên phần quản lí của cả hai nút con thì ta có thể thấy nó chính là giá trị lớn nhất của một trong ba giá trị: Tổng tiền tố lớn nhất trên nút con trái, tổng hậu tố lớn nhất trên nút con phải và tổng hậu tố lớn nhất trên nút con trái cộng tổng tiền tố lớn nhất trên nút con phải.

Do đó, giá trị tối đa cần phải tìm là giá trị lớn nhất là giá trị lớn nhất trong ba giá trị nói trên (sum, pref, suff). Chú ý là ở đây tổng tiền tố và hậu tố lớn nhất là của đoạn chứ không phải là của mảng và việc tính toán các giá trị đó có thể tổng hợp các giá trị từ các nút con.

Cấu trúc dữ liệu để thể hiện một đỉnh có thể mô tả như sau:

**struct data {**

**int sum, pref, suff, ans;**

**};**

**data combine(data l, data r) {**

**data res;**

**res.sum = l.sum + r.sum;**

**res.pref = max(l.pref, l.sum + r.pref);**

**res.suff = max(r.suff, r.sum + l.suff);**

**res.ans = max(max(l.ans, r.ans), l.suff + r.pref);**

**return res;**

**}**

Chúng ta có thể triển khai cây phân đoạn giống hệt như trong các bài trước, ở đây chúng ta thêm hàm make\_data() chứa thông tin về một nút lá.

**data make\_data(int val) {**

**data res;**

**res.sum = val;**

**res.pref = res.suff = res.ans = max(0, val);**

**return res;**

**}**

**void build(int a[], int v, int tl, int tr) {**

**if (tl == tr) {**

**t[v] = make\_data(a[tl]);**

**} else {**

**int tm = (tl + tr) / 2;**

**build(a, v\*2, tl, tm);**

**build(a, v\*2+1, tm+1, tr);**

**t[v] = combine(t[v\*2], t[v\*2+1]);**

**}**

**}**

**void update(int v, int tl, int tr, int pos, int new\_val) {**

**if (tl == tr) {**

**t[v] = make\_data(new\_val);**

**} else {**

**int tm = (tl + tr) / 2;**

**if (pos <= tm)**

**update(v\*2, tl, tm, pos, new\_val);**

**else**

**update(v\*2+1, tm+1, tr, pos, new\_val);**

**t[v] = combine(t[v\*2], t[v\*2+1]);**

**}**

**}**

Vấn đề còn lại là tính toán câu trả lời cho mỗi truy vấn được đặt ra. Ở đây, một lần nữa hàm đệ quy lại được sử dụng như các truy vấn ở trước.

**data query(int v, int tl, int tr, int l, int r) {**

**if (l > r)**

**return make\_data(0);**

**if (l == tl && r == tr)**

**return t[v];**

**int tm = (tl + tr) / 2;**

**return combine(query(v\*2, tl, tm, l, min(r, tm)),**

**query(v\*2+1, tm+1, tr, max(l, tm+1), r));**

**}**

(Code có thể xem trong bài MAXSUM)

**10. Lưu trữ mảng trong đỉnh**

Đây là một hướng triển khai khác của cây phân đoạn. Tại mỗi nút đỉnh của cây phân đoạn, chúng ta lưu trữ các thông tin về phân đoạn tương ứng dưới dạng nén (tổng, tối thiểu, tối đa,...) nhưng lại lưu trữ tất cả các phần tử của phân đoạn. Do đó gốc của cây phân đoạn sẽ lưu trữ tất cả cá giá trị của mảng, nút con phải của nó sẽ lưu trữ các phần tử nửa đầu, nút con trái sẽ lưu trữ các phần tử nửa sau.

Trong ưng dụng đơn giản nhất của kỹ thuật này, ta có thể lưu trữ các phần tử dưới dạng được sắp xếp. Trong các phiên bản phức tạp hơn của kỹ thuật này tại mỗi nút của cây phân đoạn, ta có thể sử dụng các cấu trúc phức tạp (set, map). Những cách lưu trữ này có cùng một đặc điểm đó là sử dụng bộ nhớ tuyến tính có số phần tử bằng số phần tử của đoạn mà nó lưu trữ.

Câu hỏi tự nhiên sẽ nảy sinh đó là, số bộ nhớ mà nó sử dụng? Theo trực quan có thể cảm thấy nó sử dụng lượng bộ nhớ cỡO(n2) nhưng trên thực tế, số lượng bộ nhớ mà cây sử dụng chỉ khoảng O(nlogn). Điều này là do độ cao của cây và số lượng phần tử giảm dần theo các nút từ gốc đến lá.

Như vậy, trong các bài toán mà việc sử dụng các phần tử yêu cầu ít nghiêm ngặt về bộ nhớ, việc triển khai cây phân đoạn theo kiểu này cũng là một gợi ý. Việc triển khai nó chỉ yêu cầu về bộ nhớ tốn kém đôi chút so với cây phân đoạn thông thường.

Đây là cách triển khai cây phân đoạn có một số ứng dụng trên thực tế nó gần như là một kiểu dữ liệu 2D. Tuy nhiên vẫn một phần là chưa hoàn toàn. Sau đây là một vài ứng dụng dựa theo cách triển khai này.

**Tìm phần tử nhỏ nhất, phần tử lớn hơn một số cho trước (Không có truy vấn sửa đổi)**

Chúng ta cần trả lời truy vấn có dạng như sau: cho ba số (l , r , x) , cần phải tìm số nhỏ nhất và số lớn hơn hoặc bằng x trong đoạn [l … r].

Chúng ta có thê xây dựng cây phân đoạn, tại mỗi đỉnh ta lưu trữ danh sách được sắp xếp của các phần tử xuất hiện trong phân đoạn tương ứng. Giống như các phần trên, một câu hỏi là: “Làm thế nào để xây dựng một cây phân đoạn như vậy một cách hiệu quả nhất có thể?”. Cũng giống như mọi khi, ta có thể tiếp cận vấn đề này bằng cách đệ quy, nút bên trái sẽ lưu một mảng và nút bên phải cũng lưu một mảng, nút gốc sẽ là tổng hợp của hai nút con. Chú ý ở đây là đệ quy sẽ bắt đầu gọi từ nút bé nhất. Từ quan điểm này, ta có thể sắp xếp hai mảng thành một mảng. Ở đây có thể sử dụng hai con trỏ trong STL của C++ để triển khai thuật toán.

Bởi vì cấu trúc của cây phân đoạn trong trường hợp này giống như thuật toán sắp xếp hợp nhất nên trong một số trường hợp nó còn được gọi là “Cây sắp xếp hợp nhất”.

**vector<int> t[4\*MAXN];**

**void build(int a[], int v, int tl, int tr) {**

**if (tl == tr) {**

**t[v] = vector<int>(1, a[tl]);**

**} else {**

**int tm = (tl + tr) / 2;**

**build(a, v\*2, tl, tm);**

**build(a, v\*2+1, tm+1, tr);**

**merge(t[v\*2].begin(), t[v\*2].end(), t[v\*2+1].begin(), t[v\*2+1].end(),**

**back\_inserter(t[v]));**

**}**

**}**

Chúng ta đã biết rằng, cây phân đoạn được xây dựng theo kiểu này yêu cầu O(n logn) bộ nhớ. Và dựa theo cách triển khai này nó cũng mất O(n logn) thời gian cho xây dựng sau khi mỗi nút được xây dựng theo thời gian tuyến tính là kích thước của nút.

Bây giờ ta hãy xem xét các truy vấn, Chúng ta sẽ đi xuống trong cây giống như trong cây phân đoạn thông thường.

Để trả lời câu hỏi tìm phần tử bé nhất, ta nhớ lại rằng, với các nút đã được sắp xếp thì giá trị bé nhất đó là giá trị bé nhất của một trong hai nút con, còn đối với truy vấn “tìm phần tử lớn hơn hoặc bằng một giá trị x” thì ta phải tìm kiếm nhị phân trên cả hai nút.

Do đó kết hợp với các phần đã biết ở trên về cây nhị phân được triển khai theo phương pháp này thì tổng thời gian thực hiện cho câu hỏi số hai là : O(log2n).

**int query(int v, int tl, int tr, int l, int r, int x) {**

**if (l > r)**

**return INF;**

**if (l == tl && r == tr) {**

**vector<int>::iterator pos = lower\_bound(t[v].begin(), t[v].end(), x);**

**if (pos != t[v].end())**

**return \*pos;**

**return INF;**

**}**

**int tm = (tl + tr) / 2;**

**return min(query(v\*2, tl, tm, l, min(r, tm), x),**

**query(v\*2+1, tm+1, tr, max(l, tm+1), r, x));**

**}**

Ở đây, hằng INF là đại diện cho câu trả lời, không tìm ra phần tử nào thỏa mãn.

Do cách triển khai chương trình trong phần này khá đơn giản, nên ta không đưa ra code hoàn chỉnh.

**Tìm phần tử lớn hơn hoặc bằng một giá trị nào đó (Có truy vấn sửa đổi)**

Bài toán lần này giống như bài toán lần trước nhưng có phần phức tạp hơn một chút đó là sẽ có thêm các thao tác cập nhật phần tử trong đoạn. Việc triển khai cũng sẽ không tránh khỏi các nhược điểm của cách triển khai truyền thống theo hướng này.

Giải pháp được triển khai tương tự như trong hướng triển khai lần trước nhưng trong mỗi đỉnh ta sẽ lựa chọn một cấu trúc dữ liệu thuận lợi cho việc thêm, sửa, xóa, hoặc tìm kiếm phần tử, vì có thể trong danh sách các phần tử được lưu trữ tại một nút có thể có các phần tử bằng nhau.

Việc xây dựng cây phân đoạn cũng khá giống với bài trước. Ở bài này, ta sử dụng cấu trúc dữ liệu **multiset** và không phải là danh sách được sắp xếp. Điều này có thể dẫn đến thời gian thực hiện có thể là O(nlog2n). Và nói chung thời gian hợp nhất hai nút là tuyến tính.

Việc truy vấn cũng hoàn toàn đơn giản. Ở đây, nhờ cấu trúc dữ liệu **multiset** được áp dụng nên ta có thể sử dụng các hàm lower\_bound để truy cập phần tử có giá trị thỏa mãn truy vấn. Thời gian thực hiện truy vấn là : O(logn) với một truy vấn bất kỳ.

Cuối cùng là đến thao tác sửa đổi, vì, áp dụng cấu trúc **multiset** nên ta có thể chèn thêm một phần tử mới và đảm bảo rằng giá trị của phần tử đó chỉ xuất hiện một lần.

**void update(int v, int tl, int tr, int pos, int new\_val) {**

**t[v].erase(t[v].find(a[pos]));**

**t[v].insert(new\_val);**

**if (tl != tr) {**

**int tm = (tl + tr) / 2;**

**if (pos <= tm)**

**update(v\*2, tl, tm, pos, new\_val);**

**else**

**update(v\*2+1, tm+1, tr, pos, new\_val);**

**} else {**

**a[pos] = new\_val;**

**}**

**}**

Tổng thời gian xử lí cho bài toán vẫn là O(log2n)

**Tìm phần tử lớn hơn hoặc bằng một giá trị nào đó. Tăng tốc với kỹ thuật “Xếp chồng phân đoạn - fractional cascading”.**

Chúng ta vẫn tiếp tục công việc giải bài toán, tìm phần tử nhỏ nhất nhưng lớn hơn hoặc bằng một giá trị nào đó, nhưng lần này chúng ta sẽ tăng tốc nó bằng kỹ thuật “Xếp chồng phân đoạn”.

Xếp chồng phân đoạn là một kỹ thuật cho phép bạn tăng tốc của nhiều tìm kiếm nhị phân được thưc hiện cùng một lúc. Cách tiếp cận trước đây của chúng ta là với mỗi truy vấn, ta chia nhiệm vụ thành nhiều nhiệm vụ con, mỗi nhiệm vụ con là một tìm kiếm nhị phân trên đoạn tương ứng. Xếp tầng phân đoạn cho phép bạn có thể thay thế tất cả các tìm kiếm nhị phân thành một tìm kiếm nhị phân duy nhất.

Ví dụ đơn giản nhất cho kỹ thuật xếp chồng phân đoạn là như sau: Cho tập k các danh sách đã được sắp xếp, ta phải tìm trong k danh sách đó số đầu tiên lớn hơn hoặc bằng một phần tử x đã cho.

Thay vì tìm kiếm nhị phân trong từng danh sách chúng ta có thể tập hợp tất cả các danh sách đó thành một danh sách lớn được sắp xếp. Thêm vào đó, đối với mỗi phần tử **y** ta có thể lưu trữ một danh sách là kết quả tìm kiếm phần tử đầu tiên có giá trị lớn hơn hoặc bằng **y** trong **k** danh sách. Theo đó, Nếu muốn tìm kiếm phần tử nhỏ nhất lớn hơn hoặc bằng x chúng ta chỉ cần một phép tìm kiếm nhị phân. Tuy nhiên bộ nhớ lại yêu cầu khá cao, chính vì vậy, cách tiếp cận này có thể kém hiệu quả.

Xếp tầng phân đoạn có thể làm giảm độ phức tạp tính toán và không gian lưu trữ mảng: Độ phức tạpO(log n + k) và bộ nhớ O(n). Với k danh sách đã có ta tạo ra k danh sách mới. Giả sử các danh sách đã có là L1 đến Lk và ta sẽ tạo ra các danh sách từ M1 đến Mk, thì Mk có các phần tử của Lk , các danh sách Li còn lại sẽ là hợp nhất của hai danh sách Li và các phần tử tại vị trí chẵn của Mi+1. Thêm vào đó với mỗi phần tử y trong danh sách mới được tạo sẽ thêm vào đó hai chỉ số là chỉ số phần tử đầu tiên lớn hơn hoặc bằng y trong Li và chỉ số hần tử đầu tiên lớn hơn hoặc bằng y trong Mi+1.

*Ví dụ như sau:*

Danh sách đã có:

L1 = 24, 64, 65, 80, 93

L2 = 23, 25, 26

L3 = 13, 44, 62, 66

L4 = 11, 35, 46, 79, 81

Các danh sách được tạo:

M1 = 24[0, 1], 25[1, 1], 35[1, 3], 64[1, 5], 65[2, 5], 79[3, 5], 80[3, 6], 93[4, 6]

M2 = 23[0, 1], 25[1, 1], 26[2, 1], 35[3, 1], 62[3, 3], 79[3, 5]

M3 = 13[0, 1], 35[1, 1], 44[1, 2], 62[2, 3], 66[3, 3], 79[4, 3]

M4 = 11[0, 0], 35[1, 0], 46[2, 0], 79[3, 0], 81[4, 0]

Giả sử chúng ta muốn thực hiện một truy vấn trong cấu trúc này, cho q = 50. Đầu tiên chúng ta thực hiện tìm kiếm nhị phân tiêu chuẩn cho q trong M1 , tìm giá trị 64 [1,5]. "1" trong 64 [1,5], cho chúng ta biết rằng tìm kiếm q trong L1 sẽ trả về L 1 [1] = 64. "5" trong 64 [1,5] cho chúng ta biết rằng vị trí gần đúng của q trong M2 là vị trí 5. Chính xác hơn, tìm kiếm nhị phân cho q trong M2 sẽ trả về giá trị 79 [3,5] ở vị trí 5 hoặc giá trị 62 [3,3] một vị trí trước đó. Bằng cách so sánhq đến 62, và quan sát thấy nó nhỏ hơn, chúng ta xác định rằng kết quả tìm kiếm đúng trong M2 là 62 [3,3]. Số "3" đầu tiên trong 62 [3,3] cho chúng ta biết rằng tìm kiếm q trong L2 sẽ trả về L2 [3], một giá trị cờ nghĩa là q nằm ở cuối danh sách L2. Số "3" thứ hai trong 62 [3,3] cho chúng ta biết rằng vị trí gần đúng của q trong M3 là vị trí 3. Chính xác hơn, tìm kiếm nhị phân cho q trong M3 sẽ trả về giá trị 62 [2,3] tại vị trí 3, hoặc giá trị 44 [1,2] trước đó một vị trí. Một so sánh của q với giá trị nhỏ hơn 44 cho chúng ta thấy rằng kết quả tìm kiếm đúng trong M3 là 62 [2,3]. "2" trong 62 [2,3] cho chúng ta biết rằng tìm kiếm q trong L3 sẽ trả về L3 [2] = 62 và "3" trong 62 [2,3] cho chúng ta biết rằng kết quả tìm kiếm với q trong M4 là M4 [3] = 79 [3,0] hoặc M 4 [2] = 46 [2,0]; so sánh q với 46 cho thấy kết quả đúng là 79 [3,0] và kết quả tìm kiếm q trong L 4 là L 4[3] = 79. Như vậy, chúng ta đã tìm thấy q trong mỗi danh sách trong số bốn danh sách của chúng ta, bằng cách thực hiện tìm kiếm nhị phân trong danh sách M 1, sau đó là một phép so sánh duy nhất trong mỗi danh sách kế tiếp.

(Xem thêm trong Wikipedia: Fractional cascading)

Quay trở lại vấn đề chính, Chúng ta sẽ ứng dụng tư tưởng của xếp chồng phân đoạn. Trong cây phân đoạn của chúng ta, một đỉnh sẽ chứa danh sách đã sắp xếp của các phần tử xuất hiện tại nút con trái và nút con phải của đỉnh đó (Như trong cây sắp xếp hợp nhất). Ngoài ra trong danh sách đã sắp xếp này, với mỗi phần tử y chúng ta lưu trữ thêm hai chỉ mục nhỏ: i là vị trí đầu tiên của các phần tử trong nút con trái có giá trị lớn hơn hoặc bằng y và j là vị trí đầu tiên trong nút con phải có giá trị lớn hơn hoặc bằng y. Hai giá trị này có thể tính toán song song trong quá trình xây dựng cây.

Làm thế nào để tăng tốc truy vấn

Chúng ta nhớ lại rằng, trong thuật toán trước đây chúng ta cần tìm kiếm trong từng phân đoạn, nhưng với cách cải tiến mới thì ta chỉ cần tìm kiếm trong một phân đoạn.

Để trả lời một truy vấn, ta chỉ cần tìm kiếm trong nút gốc. Giả sử tìm được giá trị y≥x trong danh sách tại nút gốc, nó sẽ cho ra hai giá trị là vị trí nhỏ nhất của phần tử lớn hơn hoặc bằng x trong nút con trái và vị trí nhỏ nhất của phần tử lớn hơn hoặc bằng x trong nút con phải. Trong mảng thông thường có thể ta cần tìm kiếm nhị phân còn trong trường hợp danh sách được lưu dưới dạng cấu trúc dữ liệu nào đó thì có thể chỉ mất thời gian O(1) và chúng ta sẽ lặp lại truy vấn đến khi nào gặp nút chứa phân đoạn như yêu cầu.

Tóm lại, với mỗi truy vấn, ta mất thời gian là O(logN). Với nút gốc ta thực hiện tìm kiếm nhị phân và đệ quy với các nút khác.

Nhưng lưu ý ở đây với dạng cây này là sử dụng nhiều bộ nhớ hơn cây phân đoạn truyền thống vốn đã sử dụng nhiều bộ nhớ (O (n logn)).

Thật dễ dàng để áp dụng kỹ thuật này cho một bài toán mà không yêu cầu nhiều sửa đổi. Hai vị trí chỉ là số nguyên và có thể dễ dàng được tính bằng cách đếm khi hợp nhất hai chuỗi đã sắp xếp.

Nó vẫn có thể cho phép các truy vấn sửa đổi, nhưng điều đó làm phức tạp toàn bộ mã. Thay vì các số nguyên, bạn cần lưu trữ mảng được sắp xếp dưới dạng multiset và thay vì các chỉ số, bạn cần lưu trữ các trình vòng lặp. Và bạn cần phải làm việc rất cẩn thận, để bạn tăng hoặc giảm các trình vòng lặp chính xác trong một truy vấn sửa đổi.

**11. Các biến thể của cây phân đoạn**

Với cây phân đoạn truyền thống, ta có thể triển khai một loạt các cải tiến của nó, chẳng hạn như trong một đỉnh có thể áp dụng các cấu trúc dữ liệu theo đó một loạt các biến thể của cây phân đoạn được ra đời như: Fenwick Trees, Cartesian trees, v v.

**12. Cập nhật phân đoạn (Kỹ thuật lan truyền lười nhác)**

Trong các vấn đề được đề cập ở trên thì việc cập nhật phần tử chỉ được thực hiện được với một phần tử. Cây phân đoạn còn cho phép thực hiện sửa đổi với nhiều phần tử liên tiếp trong cùng một thời điểm với độ phức tạp O(logn).

**Tăng thêm giá trị**

Chúng ta bắt đầu với truy vấn đơn giản nhất: tăng thêm một lượng x cho cho các phần tử trong đoạn [l … r] , và truy vấn thứ hai là đưa ra giá trị của một phần tử a[i].

Truy vấn tăng thêm giá trị rất đơn giản, giả sử ta muốn tăng thêm một lượng x vào đoạn [l..r] thi việc đầu tiên là lần theo các phân đoạn của nó rồi tìm đến các phần tử cơ sở, sau đó từ các phần tử cơ sở ta cộng thêm vào kết quả cuối cùng.

Truy vấn tìm giá trị của một phần tử cũng chỉ cần căn cứ vào chỉ mục rồi cộng tất cả các giá trị trên đường đi là đủ.

(Xem code trong Lazy Propagation AddLazy Propagation Add)

**void build(int a[], int v, int tl, int tr) {**

**if (tl == tr) {**

**t[v] = a[tl];**

**} else {**

**int tm = (tl + tr) / 2;**

**build(a, v\*2, tl, tm);**

**build(a, v\*2+1, tm+1, tr);**

**t[v] = 0;**

**}**

**}**

**void update(int v, int tl, int tr, int l, int r, int add) {**

**if (l > r)**

**return;**

**if (l == tl && r == tr) {**

**t[v] += add;**

**} else {**

**int tm = (tl + tr) / 2;**

**update(v\*2, tl, tm, l, min(r, tm), add);**

**update(v\*2+1, tm+1, tr, max(l, tm+1), r, add);**

**}**

**}**

**int get(int v, int tl, int tr, int pos) {**

**if (tl == tr)**

**return t[v];**

**int tm = (tl + tr) / 2;**

**if (pos <= tm)**

**return t[v] + get(v\*2, tl, tm, pos);**

**else**

**return t[v] + get(v\*2+1, tm+1, tr, pos);**

**}**

**Gán giá trị của một phân đoạn**

Giả sử bây giờ ta muốn gán giá trị của tất cả các phần tử a[i] trong đoạn [l … r] bằng một giá trị p nào đó và truy vấn thứ hai là tìm giá trị của phần tử a[i].

Để thực hiện truy vấn này trên phân đoạn, chúng ta sẽ xem phân đoạn tương ứng có chung một giá trị hay không? Điều này cho phép chúng ta thực hiện sự “lan truyền lười nhác”: Thay vì phải thay đổi tất cả các phân đoạn khác trong cây phân đoạn, chúng ta chỉ cần thực hiện thao tác với phân đoạn được truy vấn, các phân đoạn khác thì giữ nguyên. Một đỉnh được đánh dấu có nghĩa là tất cả các phần tử trong đỉnh đó sẽ được gán cùng một giá trị, và thực tế thì cây con của nó cũng được gán cùng một giá trị này. Theo một ý nghĩa nào đó, chúng ta “lười nhác” việc ghi giá trị cho tất cả các đỉnh đó, công việc tẻ nhạt này sẽ được thực hiện sau, nếu thấy cần thiết.

Vì vậy, sau khi sửa đổi được thực thi, một số phần tử của cây trở nên không liên quan – một số sửa đổi vẫn chưa được thực hiện trong đó.

Ví dụ: Nếu một truy vấn sửa đổi "Gán giá trị cho toàn bộ mảng a[0…n-1]"được thực thi, trong Cây phân đoạn chỉ có một thay đổi duy nhất được thực hiện - số được đặt trong gốc của cây và đỉnh này được đánh dấu. Các đoạn còn lại không thay đổi, mặc dù trên thực tế, số lượng phải được đặt trong toàn bộ cây.

Giả sử bây thực hiện truy vấn sửa đổi thứ hai là nửa đầu của mảng a[0 … n/2]được gán một giá trị. Để xử lý truy vấn này, chúng ta phải gán từng phần tử trong toàn bộ phần con bên trái của đỉnh gốc với số đó. Nhưng trước khi chúng ta làm điều này, trước tiên chúng ta phải đi qua đỉnh gốc trước. Điều tinh tế ở đây là nửa bên phải của mảng vẫn nên được gán cho giá trị của truy vấn đầu tiên và thông tin của nửa bên phải không được lưu trữ.

Cách giải quyết là đẩy thông tin của gốc sang con của nó, tức là nếu gốc của cây được gán với một số bất kỳ, thì ta gán các đỉnh con bên trái và bên phải với số này và xóa dấu của gốc. Sau đó, chúng ta có thể gán giá trị mới cho con bên trái mà không làm mất bất kỳ thông tin cần thiết nào.

Tóm lại, chúng ta được: Đối với bất kỳ truy vấn nào (truy vấn sửa đổi hoặc đọc) trong quá trình đi xuống dọc theo cây, chúng ta phải luôn đẩy thông tin từ đỉnh hiện tại vào cả hai đỉnh con của nó. Chúng ta có thể hiểu điều này theo cách như vậy, rằng khi chúng ta hạ gốc cây, chúng ta áp dụng các sửa đổi lười biềng, nhưng chính xác ở mức cần thiết (để không làm giảm độ phức tạp củaO(log n)

Để thực hiện, chúng ta cần thực hiện hàm **push**, sẽ nhận đỉnh hiện tại, và nó sẽ đẩy thông tin về đỉnh của nó cho cả hai con của nó. Chúng ta sẽ gọi hàm này ở đầu các hàm truy vấn (nhưng chúng ta sẽ không gọi nó từ các lá, vì không cần phải đẩy thông tin từ chúng thêm nữa).

(Code xem tại Lazy Propagation Assign)

**void push(int v) {**

**if (marked[v]) {**

**t[v\*2] = t[v\*2+1] = t[v];**

**marked[v\*2] = marked[v\*2+1] = true;**

**marked[v] = false;**

**}**

**}**

**void update(int v, int tl, int tr, int l, int r, int new\_val) {**

**if (l > r)**

**return;**

**if (l == tl && tr == r) {**

**t[v] = new\_val;**

**marked[v] = true;**

**} else {**

**push(v);**

**int tm = (tl + tr) / 2;**

**update(v\*2, tl, tm, l, min(r, tm), new\_val);**

**update(v\*2+1, tm+1, tr, max(l, tm+1), r, new\_val);**

**}**

**}**

**int get(int v, int tl, int tr, int pos) {**

**if (tl == tr) {**

**return t[v];**

**}**

**push(v);**

**int tm = (tl + tr) / 2;**

**if (pos <= tm)**

**return get(v\*2, tl, tm, pos);**

**else**

**return get(v\*2+1, tm+1, tr, pos);**

**}**

Chú ý: hàm truy vấn cũng có thể được triển khai theo cách khác: không thực hiện cập nhật lười nhác mà trả về giá trị ngay lập tức t [v] nếu marked[v] = 1;

**Tăng thêm trong phân đoạn, tìm số lớn nhất**

Bây giờ truy vấn sửa đổi là thêm một số vào tất cả các phần tử trong một phân đoạn và truy vấn là để tìm giá trị lớn nhất trong một phân đoạn.

Vì vậy, đối với mỗi đỉnh của Cây phân đoạn, chúng ta phải lưu trữ giá trị của phần tử tối đa của phân đoạn tương ứng. Phần thú vị là làm thế nào để tính toán lại các giá trị này trong một truy vấn sửa đổi.

Để làm công việc này, chúng ta lưu trữ một giá trị bổ sung cho mỗi đỉnh. Với giá trị này, chúng ta thêm biến addends là giá trị sẽ được cộng vào. Chúng ta cũng gọi hàm push để truyền giá trị cho cả hai con và ta phải làm điều này trong cả hai hàm cập nhật và truy vấn.

(Code xem trong Code xem tại Lazy Propagation ADD2)

**void push(int v) {**

**t[v\*2] += lazy[v];**

**lazy[v\*2] += lazy[v];**

**t[v\*2+1] += lazy[v];**

**lazy[v\*2+1] += lazy[v];**

**lazy[v] = 0;**

**}**

**void update(int v, int tl, int tr, int l, int r, int addend) {**

**if (l > r)**

**return;**

**if (l == tl && tr == r) {**

**t[v] += addend;**

**lazy[v] += addend;**

**} else {**

**push(v);**

**int tm = (tl + tr) / 2;**

**update(v\*2, tl, tm, l, min(r, tm), addend);**

**update(v\*2+1, tm+1, tr, max(l, tm+1), r, addend);**

**t[v] = max(t[v\*2], t[v\*2+1]);**

**}**

**}**

**int query(int v, int tl, int tr, int l, int r) {**

**if (l > r)**

**return -INF;**

**if (l <= tl && tr <= r)**

**return t[v];**

**push(v);**

**int tm = (tl + tr) / 2;**

**return max(query(v\*2, tl, tm, l, min(r, tm)),**

**query(v\*2+1, tm+1, tr, max(l, tm+1), r));**

**}**

**13. Cây phân đoạn nâng cao**

Một cách tự nhiên, cây phân đoạn có thể có nhiều chiều. Nếu trong trường hợp một chiều, chúng ta chia các chỉ số mảng thành các phân đoạn được quản lí trên cây thì trong trường hợp hai chiều với chỉ số đầu tiên chúng ta lưu trữ cây phân đoạn và triển khai cây phân đoạn bình thường với chiều thứ hai.

**Cây phân đoạn 2D đơn giản**

Một mảng a [0 … n - 1 , 0 … m-1] là dữ liệu đầu vào và chúng ta phải tìm tổng (hoặc tối thiểu/tối đa) trên một số ma trận con một [x1…x2,y1…y2], cũng như thực hiện sửa đổi các phần tử ma trận riêng lẻ (tức là các truy vấn có dạng a [x] [y] = p).

Vì vậy, chúng ta cần xây dựng Cây phân đoạn 2D: đầu tiên là Cây phân đoạn bằng cách sử dụng tọa độ đầu tiên (x), sau đó là thứ hai (y).

Để làm cho quá trình xây dựng dễ hiểu hơn, bạn có thể quên mảng là hai chiều và chỉ để lại chỉ số đầu tiên. Chúng ta sẽ xây dựng Cây phân đoạn một chiều thông thường chỉ sử dụng chỉ số đầu tiên. Nhưng thay vì lưu trữ một số một phân đoạn, hãy lưu trữ toàn bộ Cây phân đoạn: tức là tại thời điểm này, chúng ta nhớ rằng chúng ta cũng có chỉ số thứ hai; nhưng bởi vì tại thời điểm này, chỉ số đầu tiên đã được cố định trong một khoảng nào đó[…r], chúng ta sẽ thực hiện trên chiều của chỉ số kia a [l … r , 0 … m - 1] và như thế chúng ta xây dựng Cây phân đoạn.

Đây là phần thực hiện xây dựng Cây phân đoạn 2D. Nó gồm hai công việc riêng biệt: Công việc xây dựng Cây phân đoạn theo x (buildx), và y (buildy). Đối với các nút lá trong xây buildy chúng ta phải phân tách hai trường hợp: khi phân đoạn hiện tại của đoạn đầu tiên [tlx … trx]có độ dài 1 và khi nó có độ dài lớn hơn một. Trong trường hợp đầu tiên, chúng ta chỉ lấy giá trị tương ứng từ ma trận và trong trường hợp thứ hai, chúng ta có thể kết hợp các giá trị của hai Cây phân đoạn từ bên trái và con phải trong chỉ số x.

**void build\_y(int vx, int lx, int rx, int vy, int ly, int ry) {**

**if (ly == ry) {**

**if (lx == rx)**

**t[vx][vy] = a[lx][ly];**

**else**

**t[vx][vy] = t[vx\*2][vy] + t[vx\*2+1][vy];**

**} else {**

**int my = (ly + ry) / 2;**

**build\_y(vx, lx, rx, vy\*2, ly, my);**

**build\_y(vx, lx, rx, vy\*2+1, my+1, ry);**

**t[vx][vy] = t[vx][vy\*2] + t[vx][vy\*2+1];**

**}**

**}**

**void build\_x(int vx, int lx, int rx) {**

**if (lx != rx) {**

**int mx = (lx + rx) / 2;**

**build\_x(vx\*2, lx, mx);**

**build\_x(vx\*2+1, mx+1, rx);**

**}**

**build\_y(vx, lx, rx, 1, 0, m-1);**

**}**

Cây Phân đoạn như vậy vẫn sử dụng một lượng bộ nhớ tuyến tính, nhưng lớn hơn: 16 n\*m. Rõ ràng hàm buildx cũng hoạt động trong thời gian tuyến tính.

Bây giờ chúng ta chuyển sang xử lý các truy vấn. Chúng ta sẽ trả lời cho truy vấn hai chiều bằng cách sử dụng cùng một nguyên tắc: đầu tiên cần truy vấn trên chỉ số thứ nhất, và sau đó đối với mọi đỉnh đạt tới tương ứng, chúng ta gọi Cây phân đoạn tương ứng với chỉ số thứ hai.

**int sum\_y(int vx, int vy, int tly, int try\_, int ly, int ry) {**

**if (ly > ry)**

**return 0;**

**if (ly == tly && try\_ == ry)**

**return t[vx][vy];**

**int tmy = (tly + try\_) / 2;**

**return sum\_y(vx, vy\*2, tly, tmy, ly, min(ry, tmy))**

**+ sum\_y(vx, vy\*2+1, tmy+1, try\_, max(ly, tmy+1), ry);**

**}**

**int sum\_x(int vx, int tlx, int trx, int lx, int rx, int ly, int ry) {**

**if (lx > rx)**

**return 0;**

**if (lx == tlx && trx == rx)**

**return sum\_y(vx, 1, 0, m-1, ly, ry);**

**int tmx = (tlx + trx) / 2;**

**return sum\_x(vx\*2, tlx, tmx, lx, min(rx, tmx), ly, ry)**

**+ sum\_x(vx\*2+1, tmx+1, trx, max(lx, tmx+1), rx, ly, ry);**

**}**

Hàm này hoạt động trong thời gian O(logn\*logm), vì nó lần đầu tiên đi xuống trong chỉ số đầu tiên của mảng sau đó đi xuống theo cây phân đoạn ở chỉ số thứ hai.

Cuối cùng, chúng ta xem xét truy vấn cập nhật. Chúng ta muốn tìm hiểu cách sử dụng Cây phân đoạn để cập nhật[x] [y] = p. Rõ ràng là các thay đổi sẽ chỉ xảy ra trong các đỉnh tương ứng quản lí vị trí x (và như vậy mất thời gian là O(logn)), và tương tự với cây phân đoạn tiếp theo sẽ mất thời gian là O(logm). Do đó, việc triển khai sẽ không khác lắm so với trường hợp một chiều.

**void update\_y(int vx, int lx, int rx, int vy, int ly, int ry, int x, int y, int new\_val) {**

**if (ly == ry) {**

**if (lx == rx)**

**t[vx][vy] = new\_val;**

**else**

**t[vx][vy] = t[vx\*2][vy] + t[vx\*2+1][vy];**

**} else {**

**int my = (ly + ry) / 2;**

**if (y <= my)**

**update\_y(vx, lx, rx, vy\*2, ly, my, x, y, new\_val);**

**else**

**update\_y(vx, lx, rx, vy\*2+1, my+1, ry, x, y, new\_val);**

**t[vx][vy] = t[vx][vy\*2] + t[vx][vy\*2+1];**

**}**

**}**

**void update\_x(int vx, int lx, int rx, int x, int y, int new\_val) {**

**if (lx != rx) {**

**int mx = (lx + rx) / 2;**

**if (x <= mx)**

**update\_x(vx\*2, lx, mx, x, y, new\_val);**

**else**

**update\_x(vx\*2+1, mx+1, rx, x, y, new\_val);**

**}**

**update\_y(vx, lx, rx, 1, 0, m-1, x, y, new\_val);**

**}**

(Code xem trong SEGMENT2D)

**Nén cây 2D**

Chẳng hạn ta có bài toán như sau: Có n các điểm trên mặt phẳng được cho bởi tọa độ của chúng (xi,yi) và thường xuyên phải trả lời các truy vấn dạng "đếm số điểm nằm trong hình chữ nhật ((x1,y1) , (x2,y2))". Rõ ràng là trong trường hợp này, nếu xây dựng Cây phân đoạn hai chiều vớiO(n2) để mô tả số điểm nằm trong đoạn [(x1,y1) , (x2,y2)] với n2 phần tử thì hầu hết bộ nhớ này sẽ bị lãng phí, và với mỗi truy vấn ta có thể mất O(logn) để tìm theo tọa độ thứ nhất và cũng mất O(nlogn) để tìm đến tọa độ thứ hai.

Ta có thể tiến hành cải tiến như sau: Tại mỗi đỉnh của Cây phân đoạn đối với tọa độ đầu tiên, chúng ta lưu trữ một Cây phân đoạn được xây dựng bởi các tọa độ thứ hai xảy ra trong đoạn hiện tại của tọa độ đầu tiên. Nói cách khác, khi xây dựng Cây phân đoạn bên trong một số đỉnh có chỉ mục vx và ranh giới tlx và trx, chúng ta chỉ xem xét những điểm rơi vào đoạn x ∈ [tlx , trx]và xây dựng Cây phân đoạn bằng cách sử dụng chúng.

Vì vậy, chúng ta sẽ biết được rằng mỗi Cây phân đoạn trên tọa độ thứ hai sẽ chiếm chính xác nhiều bộ nhớ. Do đó, tổng dung lượng bộ nhớ sẽ giảm xuống O(nlogn). Chúng ta vẫn có thể trả lời các câu hỏi trong thời gianO(log2n), và chúng ta chỉ cần thực hiện một tìm kiếm nhị phân trên tọa độ thứ hai, nhưng điều này sẽ không làm giảm độ phức tạp.

Nhưng các truy vấn sửa đổi sẽ là không thể với cấu trúc này: trên thực tế, nếu một điểm mới xuất hiện, chúng ta phải thêm một phần tử mới vào giữa một số Cây phân đoạn dọc theo tọa độ thứ hai, điều này không thể được thực hiện một cách hiệu quả.

Kết luận, chúng ta lưu ý rằng Cây phân đoạn hai chiều được hợp đồng theo cách được mô tả trở nên thực tế tương đương với việc sửa đổi Cây phân đoạn một chiều (*xem Lưu toàn bộ mảng con trong mỗi đỉnh*). Cụ thể Cây phân đoạn hai chiều chỉ là một trường hợp đặc biệt của việc lưu trữ một mảng con trong mỗi đỉnh của cây. Sau đó, nếu bạn đã từ bỏ Cây phân đoạn hai chiều do không thể thực hiện truy vấn, bạn nên cố gắng thay thế Cây phân đoạn lồng nhau bằng một số cấu trúc dữ liệu mạnh mẽ hơn, ví dụ như cây Đề-các.

**14. Ghi lại lịch sử sửa đổi cây phân đoạn (Cây phân đoạn liên tục - Persistent Segment Tree)**

Cấu trúc dữ liệu liên tục là cấu trúc dữ liệu ghi nhớ trạng thái trước đó cho mỗi lần sửa đổi. Điều này cho phép truy cập bất kỳ phiên bản nào của cấu trúc dữ liệu này mà chúng ta quan tâm và thực hiện truy vấn trên đó. *(xem Wikipedia Persistent\_data)*

Cây phân đoạn là một cấu trúc dữ liệu có thể được biến thành một cấu trúc dữ liệu liên tục một cách hiệu quả (cả về thời gian và mức tiêu thụ bộ nhớ). Chúng ta muốn tránh sao chép cây hoàn chỉnh trước mỗi lần sửa đổi và chúng ta không muốn mấtO(log n) thời gian để trả lời các truy vấn đoạn.

Trên thực tế, bất kỳ yêu cầu sửa đổi nào trong Cây phân đoạn đều dẫn đến thay đổi dữ liệu củaO(logn) đỉnh dọc theo đường đi bắt đầu từ gốc. Vì vậy, nếu chúng ta lưu trữ Cây phân đoạn bằng cách sử dụng các con trỏ (tức là một đỉnh lưu trữ các con trỏ tới các đỉnh con bên trái và bên phải), thì khi thực hiện truy vấn sửa đổi, chúng ta chỉ cần tạo các đỉnh mới thay vì thay đổi các đỉnh có sẵn. Các đỉnh không bị ảnh hưởng bởi truy vấn sửa đổi vẫn có thể được sử dụng bằng cách trỏ con trỏ đến các đỉnh cũ. Vì vậy, đối với một truy vấn sửa đổi O(logn) các đỉnh mới sẽ được tạo, bao gồm một đỉnh gốc mới của Cây phân đoạn và toàn bộ phiên bản trước của cây được bắt nguồn từ đỉnh gốc cũ sẽ không thay đổi.

Hãy đưa ra một ví dụ triển khai cho Cây phân đoạn đơn giản nhất: khi chỉ có một truy vấn yêu cầu tổng và các truy vấn sửa đổi của các phần tử đơn lẻ.

**struct Vertex {**

**Vertex \*l, \*r;**

**int sum;**

**Vertex(int val) : l(nullptr), r(nullptr), sum(val) {}**

**Vertex(Vertex \*l, Vertex \*r) : l(l), r(r), sum(0) {**

**if (l) sum += l->sum;**

**if (r) sum += r->sum;**

**}**

**};**

**Vertex\* build(int a[], int tl, int tr) {**

**if (tl == tr)**

**return new Vertex(a[tl]);**

**int tm = (tl + tr) / 2;**

**return new Vertex(build(a, tl, tm), build(a, tm+1, tr));**

**}**

**int get\_sum(Vertex\* v, int tl, int tr, int l, int r) {**

**if (l > r)**

**return 0;**

**if (l == tl && tr == r)**

**return v->sum;**

**int tm = (tl + tr) / 2;**

**return get\_sum(v->l, tl, tm, l, min(r, tm))**

**+ get\_sum(v->r, tm+1, tr, max(l, tm+1), r);**

**}**

**Vertex\* update(Vertex\* v, int tl, int tr, int pos, int new\_val) {**

**if (tl == tr)**

**return new Vertex(new\_val);**

**int tm = (tl + tr) / 2;**

**if (pos <= tm)**

**return new Vertex(update(v->l, tl, tm, pos, new\_val), v->r);**

**else**

**return new Vertex(v->l, update(v->r, tm+1, tr, pos, new\_val));**

**}**

Đối với mỗi sửa đổi của Cây phân đoạn, chúng ta sẽ nhận được một đỉnh gốc mới. Để chuyển nhanh giữa hai phiên bản khác nhau của Cây phân đoạn, chúng ta cần lưu trữ gốc này trong một mảng. Để sử dụng một phiên bản cụ thể của Cây phân đoạn, chúng ta chỉ cần gọi truy vấn bằng cách sử dụng đỉnh gốc thích hợp.

Với cách tiếp cận được mô tả ở trên, hầu hết mọi Cây phân đoạn đều có thể được biến thành một cấu trúc dữ liệu liên tục.

**Tìm kiếm k-số nhỏ nhất trong một đoạn**

Lần này chúng ta phải trả lời các truy vấn dạng "Tìm k-phần tử nhỏ nhất trong đoạn [l … r]. Truy vấn này có thể được trả lời bằng cách sử dụng tìm kiếm nhị phân và Cây sắp xếp hợp nhất, nhưng độ phức tạp về thời gian cho một truy vấn duy nhất sẽ làO (log3n). Chúng ta sẽ giải bài toán bằng cách sử dụng Cây phân đoạn liên tục trong thời gianO(logn).

Trước tiên, chúng ta sẽ nói về giải pháp cho bài toán đơn giản hơn: Chúng ta sẽ chỉ xem xét các mảng trong đó các phần tử bị ràng buộc bởi 0 ≤ a [i] < n. Và chúng ta chỉ muốn tìm phần tử nhỏ nhất thứ k trong một số tiền tố của mảng a. Sẽ rất dễ dàng để mở rộng các ý tưởng được phát triển sau này cho các mảng không bị giới hạn và các truy vấn đoạn không bị hạn chế. Lưu ý rằng chúng ta sẽ sử dụng chỉ mục dựa trên a.

Chúng ta sẽ sử dụng Cây phân đoạn đếm tất cả các số xuất hiện, tức là trong Cây phân đoạn, chúng ta sẽ lưu trữ tần suất của mảng. Vì vậy, các đỉnh lá sẽ lưu trữ tần suất các giá trị 0, 1, …, n–1 sẽ xuất hiện trong mảng và các đỉnh khác lưu trữ bao nhiêu số trong một số đoạn trong mảng. Nói cách khác, chúng ta tạo một Cây phân đoạn thông thường với các truy vấn tổng trên tần suất của mảng. Nhưng thay vì tạo ra tất cả n Cây phân đoạn cho mỗi tiền tố có thể có, chúng ta sẽ tạo một cây liên tục, chứa cùng một thông tin. Chúng ta sẽ bắt đầu với Cây phân đoạn trống (tất cả số lượng sẽ là 0) được trỏ đến bởi root0 và thêm các phần tử a[1], a [2], …, a [n] lân lượt từng cái một. Đối với mỗi sửa đổi, chúng ta sẽ nhận được một đỉnh gốc mới, Gọi rooti là gốc của Cây phân đoạn sau khi chèn i phần tử đầu tiên trong các phần tử của mảng a. Cây phân đoạn có gốc tại rooti sẽ chứa tần suất của tiền tố a [1 … i]. Sử dụng Cây phân đoạn này, chúng ta có thể mất O(logn) thời gian để tìm vị trí của phần tử lớn thứ k và sử dụng kỹ thuật tương tự đã nói để tìm vị trí của số 0 thứ k.

Bây giờ đến phiên bản động của bài toán

Đầu tiên cho hạn chế đối với các truy vấn: Thay vì chỉ thực hiện các truy vấn này trên tiền tố của a, chúng ta muốn sử dụng bất kỳ phân đoạn tùy ý nào a [l … r]. Ở đây, chúng ta cần một Cây phân đoạn đại diện cho tần suất của các phần tử trong đoạn a [l … r]. Dễ dàng nhận thấy rằng Cây phân đoạn như vậy chỉ là sự khác biệt giữa Cây phân đoạn được bắt nguồn từ rootr và Cây phân đoạn bắt nguồn từ rootl - 1, tức là mọi đỉnh trong [l … r] Cây phân đoạn có thể được tính với đỉnh của rootr cây trừ đi đỉnh của rootl - 1 trong cây.

Trong việc thực hiện hàm find\_kth có thể được xử lý bằng cách chuyển hai con trỏ đỉnh và tính toán số / tổng của đoạn hiện tại dưới dạng hiệu số của hai số / tổng của các đỉnh.

Đây là những sửa hàm liên quan.

**Vertex\* build(int tl, int tr) {**

**if (tl == tr)**

**return new Vertex(0);**

**int tm = (tl + tr) / 2;**

**return new Vertex(build(tl, tm), build(tm+1, tr));**

**}**

**Vertex\* update(Vertex\* v, int tl, int tr, int pos) {**

**if (tl == tr)**

**return new Vertex(v->sum+1);**

**int tm = (tl + tr) / 2;**

**if (pos <= tm)**

**return new Vertex(update(v->l, tl, tm, pos), v->r);**

**else**

**return new Vertex(v->l, update(v->r, tm+1, tr, pos));**

**}**

**int find\_kth(Vertex\* vl, Vertex \*vr, int tl, int tr, int k) {**

**if (tl == tr)**

**return tl;**

**int tm = (tl + tr) / 2, left\_count = vr->l->sum - vl->l->sum;**

**if (left\_count >= k)**

**return find\_kth(vl->l, vr->l, tl, tm, k);**

**return find\_kth(vl->r, vr->r, tm+1, tr, k-left\_count);**

**}**

Như đã viết ở trên, chúng ta cần lưu trữ gốc của Cây phân đoạn ban đầu và cũng như tất cả các gốc sau mỗi lần cập nhật. Đây là code để xây dựng Cây phân đoạn liên tục trên một vectơ a có các phần tử trong đoạn [0, MAX\_VALUE].

**int tl = 0, tr = MAX\_VALUE + 1;**

**std::vector<Vertex\*> roots;**

**roots.push\_back(build(tl, tr));**

**for (int i = 0; i < a.size(); i++) {**

**roots.push\_back(update(roots.back(), tl, tr, a[i]));**

**}**

**// find the 5th smallest number from the subarray [a[2], a[3], ..., a[19]]**

**int result = find\_kth(roots[2], roots[20], tl, tr, 5);**

Bây giờ đến các hạn chế: Chúng ta có thể chuyển đổi bất kỳ mảng nào thành một mảng như vậy bằng cách nén chỉ mục. Phần tử nhỏ nhất trong mảng sẽ được gán giá trị 0, phần tử nhỏ thứ hai có giá trị 1, v.v. Dễ dàng tạo các bảng tra cứu (ví dụ: map), chuyển đổi một giá trị thành chỉ mục của nó và ngược lại thì mấtO(logn) thời gian.

**15. Cây phân đoạn ngầm định**

Trước đây, chúng ta đã xem xét các trường hợp, chúng ta đã xây dựng cây phân đoạn với kích thước bộ nhớ cố định. Nhưng phải làm gì nếu kích thước ban đầu được lấp đầy bởi một số phần tử mặc định, nhưng kích thước thực của nó không cho phép bạn hoàn toàn xây dựng trước?

Chúng ta có thể giải quyết vấn đề này bằng cách không tạo cây phân đoạn một cách rõ ràng. Ban đầu, chúng ta sẽ chỉ tạo gốc, và chúng ta sẽ tạo các đỉnh khác chỉ khi chúng ta cần chúng. Trong trường hợp này, chúng ta sẽ sử dụng việc triển khai trên các con trỏ (trước khi đi đến các đỉnh con, hãy kiểm tra xem chúng có được tạo hay không, và nếu không, hãy tạo chúng). Mỗi truy vấn vẫn chỉ có độ phức tạpO (logn), đủ nhỏ cho hầu hết các trường hợp sử dụng (Chẳng hạn. log2109≈30).

Trong triển khai này, chúng ta có hai truy vấn, thêm giá trị vào một vị trí (ban đầu tất cả các giá trị là 0), và tính toán tổng của tất cả các giá trị trong một đoạn. Vertex(0, n) sẽ là đỉnh gốc của cây ngầm định.

**struct Vertex {**

**int left, right;**

**int sum = 0;**

**Vertex \*left\_child = nullptr, \*right\_child = nullptr;**

**Vertex(int lb, int rb) {**

**left = lb;**

**right = rb;**

**}**

**void extend() {**

**if (!left\_child && left + 1 < right) {**

**int t = (left + right) / 2;**

**left\_child = new Vertex(left, t);**

**right\_child = new Vertex(t, right);**

**}**

**}**

**void add(int k, int x) {**

**extend();**

**sum += x;**

**if (left\_child) {**

**if (k < left\_child->right)**

**left\_child->add(k, x);**

**else**

**right\_child->add(k, x);**

**}**

**}**

**int get\_sum(int lq, int rq) {**

**if (lq <= left && right <= rq)**

**return sum;**

**if (max(left, lq) >= min(right, rq))**

**return 0;**

**extend();**

**return left\_child->get\_sum(lq, rq) + right\_child->get\_sum(lq, rq);**

**}**

**};**

Rõ ràng ý tưởng này có thể được mở rộng theo nhiều cách khác nhau. Ví dụ: bằng cách thêm hỗ trợ cho các cập nhật đoạn thông qua lan truyền lười nhác.

**16. Bài tập áp dụng**

* [SPOJ - KQUERY](http://www.spoj.com/problems/KQUERY/) [Persistent segment tree / Merge sort tree]
* [Codeforces - Xenia and Bit Operations](https://codeforces.com/problemset/problem/339/D)
* [UVA 11402 - Ahoy, Pirates!](https://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=2397)
* [SPOJ - GSS3](http://www.spoj.com/problems/GSS3/)
* [Codeforces - Distinct Characters Queries](https://codeforces.com/problemset/problem/1234/D)
* [Codeforces - Knight Tournament](https://codeforces.com/contest/356/problem/A) [For beginners]
* [Codeforces - Ant colony](https://codeforces.com/contest/474/problem/F)
* [Codeforces - Drazil and Park](https://codeforces.com/contest/515/problem/E)
* [Codeforces - Circular RMQ](https://codeforces.com/problemset/problem/52/C)
* [Codeforces - Lucky Array](https://codeforces.com/contest/121/problem/E)
* [Codeforces - The Child and Sequence](https://codeforces.com/contest/438/problem/D)
* [Codeforces - DZY Loves Fibonacci Numbers](https://codeforces.com/contest/446/problem/C) [Lazy propagation]
* [Codeforces - Alphabet Permutations](https://codeforces.com/problemset/problem/610/E)
* [Codeforces - Eyes Closed](https://codeforces.com/problemset/problem/895/E)
* [Codeforces - Kefa and Watch](https://codeforces.com/problemset/problem/580/E)
* [Codeforces - A Simple Task](https://codeforces.com/problemset/problem/558/E)
* [Codeforces - SUM and REPLACE](https://codeforces.com/problemset/problem/920/F)
* [COCI - Deda](https://oj.uz/problem/view/COCI17_deda) [Last element smaller or equal to x / Binary search]
* [Codeforces - The Untended Antiquity](https://codeforces.com/problemset/problem/869/E) [2D]
* [CSES - Hotel Queries](https://cses.fi/problemset/task/1143)
* [CSES - Polynomial Queries](https://cses.fi/problemset/task/1736)
* [CSES - Range Updates and Sums](https://cses.fi/problemset/task/1735)

# 17. Tài liệu tham khảo

[1]. Hồ Sĩ Đàm (2016),Tài liệu giáo khoa chuyên Tin học quyển 1,Nhà xuất bản giáo dục Việt Nam.

[2]. <https://vnoi.info/wiki/Home>

[3]. <https://codeforces.com>

[4]. <https://www.codechef.com>

[5]. <https://vn.spoj.com>