

VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING



DATA STRUCTURES AND ALGORITHMS (CO200B)

Assignment

“Segment Tree”

Instructor(s): Nguyễn Đức Dũng

Students: Nguyễn Xuân Huy Hoàng - 2311070 (*Group TN01 - Team 126, **Leader***)

HO CHI MINH CITY, DECEMBER 2024

Contents

List of Symbols	2
List of Acronyms	2
List of Figures	4
List of Tables	4
1 Giới thiệu về Segment tree	4
1.1 Giới thiệu	4
1.2 Lịch sử	4
1.3 Một số bài toán về Range Query	4
2 Thao tác trên Segment Tree trong các bài toán Range Query	5
2.1 Xây dựng	5
2.2 Cập nhật	6
2.3 Truy vấn	8
2.4 Một số kỹ thuật trên cây phân đoạn	9
2.4.1 Lazy Propagation	9
2.4.2 Persistent Segment Tree	9
2.4.3 Mergeable Segment Tree	10
3 Ứng dụng vào quản lý điểm số của sinh viên trong một hệ thống học tập	11
4 Mã nguồn cho hệ thống quản lý điểm số của sinh viên	13
4.1 Cấu Trúc Dữ Liệu Sinh Viên	15
4.2 Các Hàm Phép Toán	15
4.3 Cây Phân Đoạn	15
4.3.1 Cấu Trúc Cây Phân Đoạn	16
4.3.2 Phương Thức Chính	16
4.4 Chương Trình Chính	16
4.5 Kết Luận	16
4.6 Kiểm thử	16
4.6.1 Testcase 1	16
4.6.2 Testcase 2	17
5 Kết luận	19
6 References	20



List of Symbols

\mathbb{N} Set of natural numbers

\mathbb{R} Set of real numbers

\mathbb{R}^+ Set of positive real numbers

List of Acronyms

ODE (First-Order) Ordinary Differential Equation

IVP Initial-Value Problem

LTE Local Truncation Error

DS Dynamical System

Fig. Figure

Tab. Table

Sys. System of Equations

Eq. Equation

e.g. For Example

i.e. That Is



List of Figures

List of Tables

1 Giới thiệu về Segment tree

1.1 Giới thiệu

Segment Tree (Cây phân đoạn) là một cấu trúc dữ liệu mạnh mẽ, được thiết kế để xử lý hiệu quả các bài toán truy vấn trên mảng như tính tổng, tìm giá trị lớn nhất/nhỏ nhất trong một đoạn, và cập nhật giá trị của các phần tử. Đây là một giải pháp lý tưởng cho các bài toán mà mảng đầu vào thay đổi thường xuyên và yêu cầu truy vấn nhanh chóng.

1.2 Lịch sử

Segment Tree được giới thiệu trong lĩnh vực khoa học máy tính vào khoảng những năm 1970 bởi *Jon Bentley*. Nó được phát triển nhằm giải quyết các bài toán liên quan đến xử lý tín hiệu số và quản lý bộ nhớ. Segment Tree nổi lên như một phương pháp tối ưu để giải quyết các vấn đề liên quan đến các phép toán trên đoạn con (range queries) trong thời gian ngắn.

Cấu trúc của Segment Tree có khả năng phân chia bài toán lớn thành các bài toán con nhỏ hơn, dựa trên nguyên lý chia để trị (divide and conquer). Điều này giúp nó trở thành một công cụ hữu ích trong các lĩnh vực như xử lý hình ảnh, thống kê và cơ sở dữ liệu.

1.3 Một số bài toán về Range Query

- *Tính tổng đoạn (Range Sum Query):*

Được sử dụng để tính tổng các phần tử trong một đoạn mảng bất kỳ. Khi mảng được cập nhật, cần phải cập nhật giá trị nhanh chóng mà không cần tính lại toàn bộ mảng.

- *Truy vấn giá trị lớn nhất/nhỏ nhất (Range Minimum/Maximum Query):*

Tìm giá trị nhỏ nhất hoặc lớn nhất trong một đoạn con của mảng. Điều này hữu ích trong các bài toán như tìm khoảng thời gian dài nhất mà hiệu suất không giảm hoặc tìm điểm cao nhất trong một đoạn thời gian.

- *Cập nhật phần tử nhanh chóng (Point Update):*

Khi một phần tử trong mảng thay đổi, cần phải cập nhật giá trị này với độ phức tạp tối đa là $O(\log n)$, phải nhanh hơn so với cập nhật tuyến tính.

- *Truy vấn với điều kiện (Range Count Query):*

Đếm số lượng phần tử trong một đoạn mảng thỏa mãn một điều kiện nào đó, ví dụ như đếm số sinh viên có điểm lớn hơn một mức cụ thể.

- *Bài toán bài toán liên quan đến khoảng thời gian (Interval Problems):*

Những bài toán về quản lý thời gian biểu, đặt phòng hoặc kiểm tra xung đột trong khoảng thời gian.

2 Thao tác trên Segment Tree trong các bài toán Range Query

2.1 Xây dựng

Cây phân đoạn có thể xây dựng bằng hai cách:

□ Bottom - up

○ Quy trình

1. Khởi tạo các nút lá tương ứng với các phần tử trong mảng.
2. Di chuyển từ các nút lá lên gốc cây, tính giá trị cho từng nút cha dựa trên hai nút con của nó.

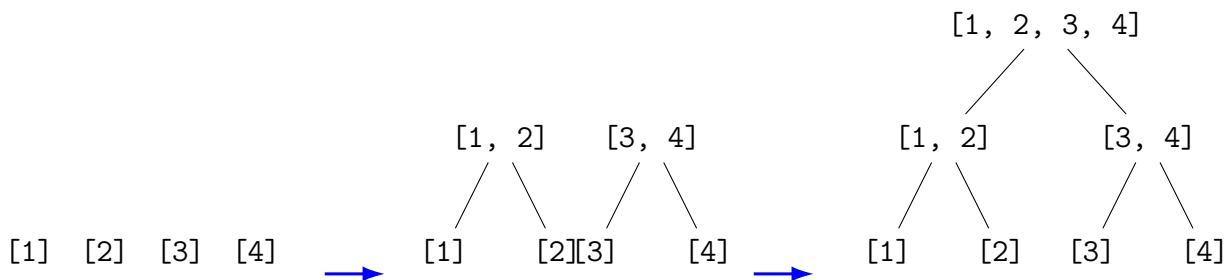
○ Độ phức tạp: $O(n)$ (với n là số phần tử có trong mảng)

○ Ví dụ:

- Nút lá lưu giá trị từ mảng ban đầu.
- Nút cha lưu mối quan hệ từ hai nút con.

$$tree[i] = func(tree[2 \cdot i], tree[2 \cdot i + 1]) \text{ với } i \text{ là chỉ số tính từ } 1$$

- $func$ là hàm tính tổng đối với bài toán tìm tổng hoặc là hàm tìm giá trị lớn nhất/nhỏ nhất đối với bài toán tìm giá trị tương ứng.
- Cách ghi $[a, b, c]$ thể hiện mối liên hệ giữa a, b, c .



□ Top - down

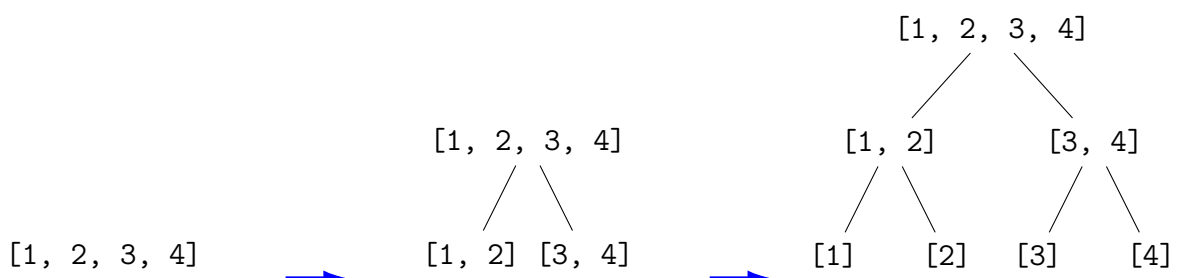
○ Quy trình

1. Sử dụng phương pháp đệ quy, bắt đầu từ gốc.
2. Chia dãy ban đầu thành hai nửa, xây dựng cây cho từng nửa và gộp kết quả vào nút cha.

○ Độ phức tạp: $O(n)$ (với n là số phần tử có trong mảng)

○ Ví dụ:

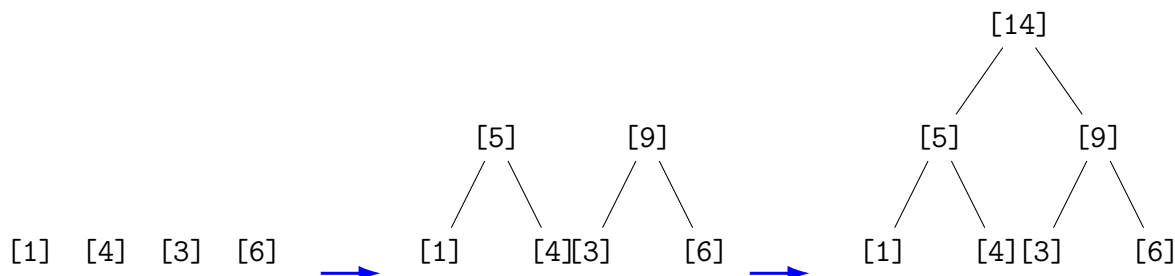
- Nếu mảng $[1, 2, 3, 4]$, ta chia thành hai nửa $[1, 2]$ và $[3, 4]$, tiếp tục chia nhỏ cho đến khi chỉ còn 1 phần tử.



2.2 Cập nhật

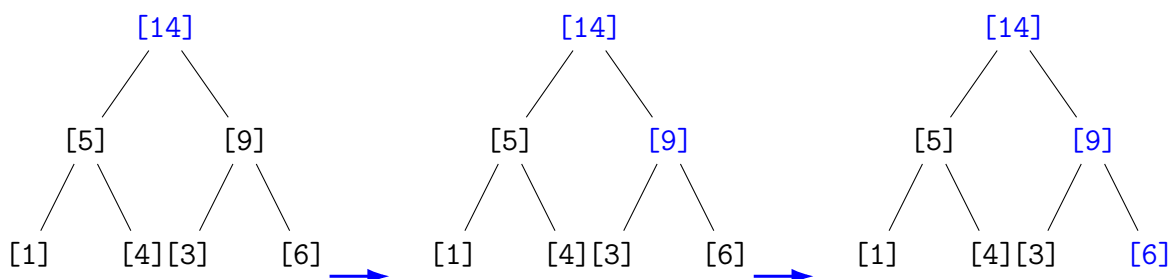
Cập nhật giá trị trong mảng đồng thời điều chỉnh các nút liên quan trong cây.

□ **Ví dụ:** Đối với bài toán **truy vấn tổng**. Giả sử có mảng gồm 4 phần tử $\{1, 4, 3, 6\}$, sau khi xây dựng bằng phương pháp *top - down* được cây như sau:

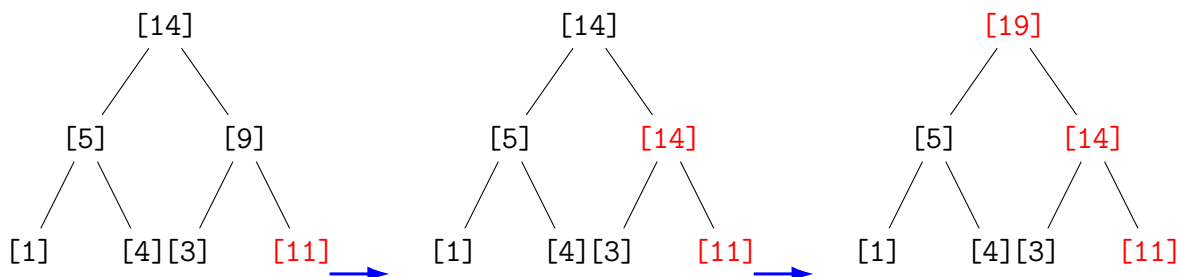


□ Cập nhật điểm (Point Update)

- Mục tiêu: Thay đổi một phần tử tại vị trí i trong mảng.
- Quy trình
 1. Tìm nút lá tương ứng với vị trí i .
 2. Cập nhật giá trị tại nút lá.
 3. Di chuyển từ nút lá lên gốc, cập nhật các nút cha dựa trên các nút con.
- Độ phức tạp: $O(\log n)$ (với n là số phần tử có trong mảng)
- Nếu muốn cập nhật phần tử thứ 4 lên 11, đầu tiên phải đệ quy để tìm phần tử thứ 4 trước.



- Sau khi đã tìm được thì tiến hành cập nhật phần tử thứ 4 từ 6 lên 11.

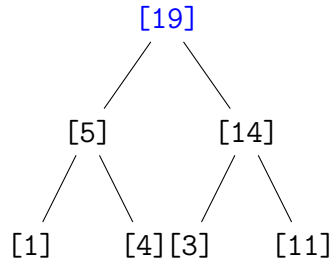


□ Cập nhật đoạn (Range Update)

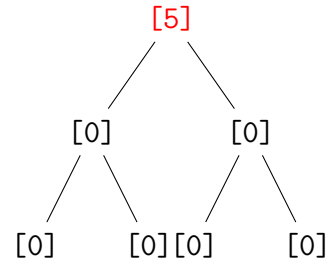
- Mục tiêu: Cập nhật giá trị cho một đoạn $[l, r]$.
- Phương pháp phổ biến: *Lazy Propagation* hay cập nhật lười, sử dụng mảng phụ để trì hoãn việc cập nhật khi cần thiết. Nói cách khác, ta tạo thêm một cây phân đoạn mới chỉ để chứa giá trị mới cập nhật nhằm giảm độ phức tạp xuống.

- Độ phức tạp: $O(\log n)$ (với n là số phần tử có trong mảng)
- Nếu cần cập nhật các phần tử thứ 2, 3, 4 lên 5 giá trị, đầu tiên phải truy tới node và dùng cập nhật lười thay vì cập nhật hết các nút lá.

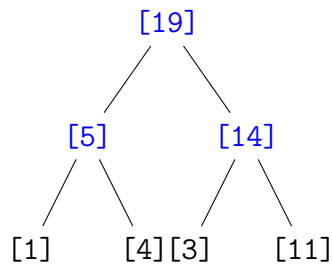
Segment Tree



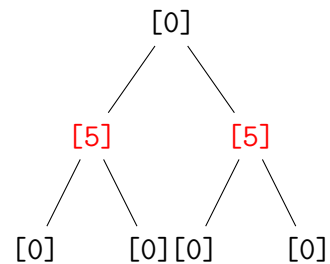
Lazy Propagation Tree



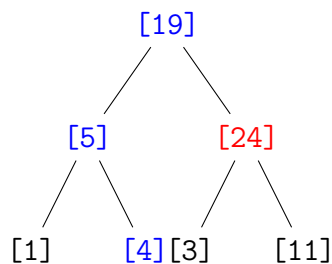
Segment Tree



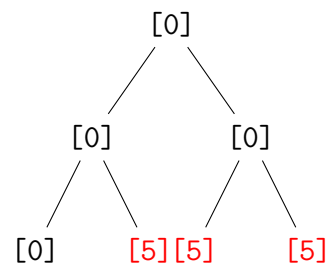
Lazy Propagation Tree



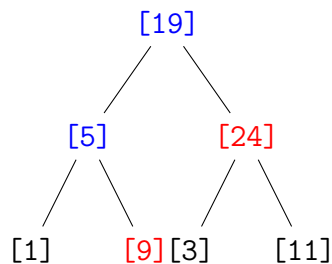
Segment Tree



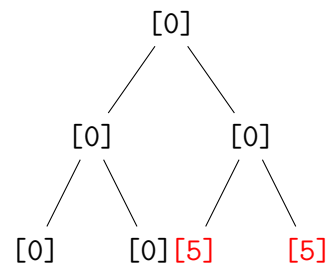
Lazy Propagation Tree

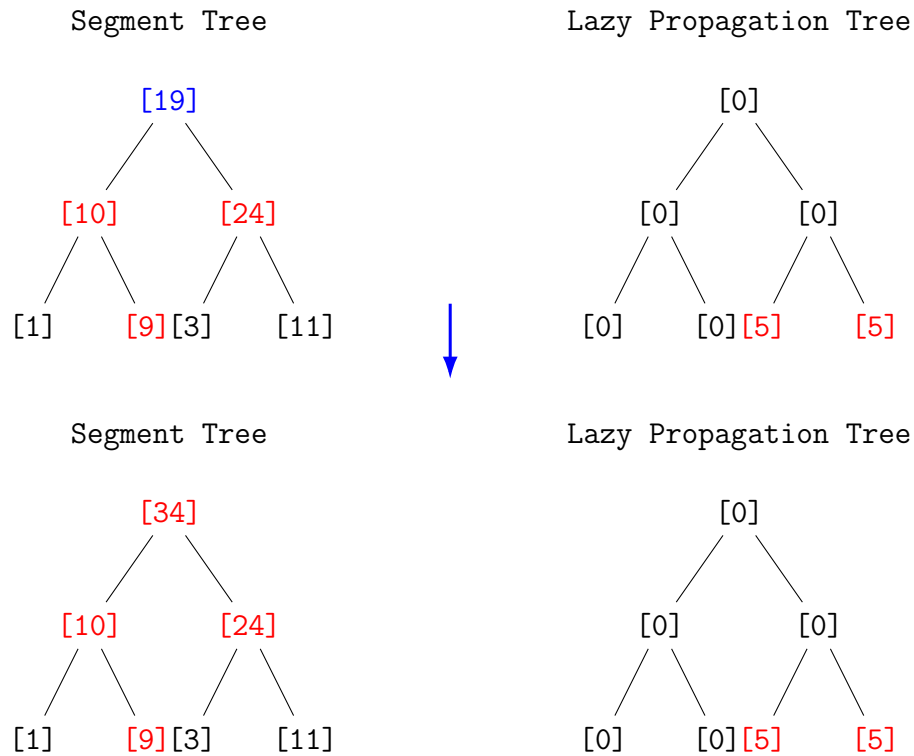


Segment Tree



Lazy Propagation Tree



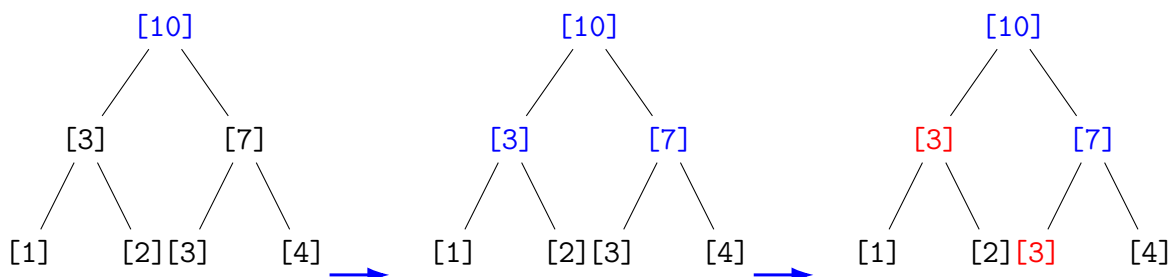


- *Lazy Propagation* đóng vai trò làm trì hoãn việc cập nhật, giúp giảm thiểu độ phức tạp thời gian xuống.

2.3 Truy vấn

□ Truy vấn đoạn (Range Query)

- Tính toán giá trị (như tổng, max, min, gcd) trên một đoạn $[l, r]$.
- Quy trình
 1. Bắt đầu từ gốc, kiểm tra xem đoạn $[l, r]$ giao với đoạn của mỗi nút thế nào.
 2. Nếu đoạn của nút hoàn toàn nằm trong $[l, r]$, sử dụng giá trị của nút đó.
 3. Nếu không, đệ quy xuống hai nút con.
 4. Kết hợp kết quả từ hai nút con để trả về kết quả cho đoạn cha.
- Độ phức tạp: $O(\log n)$ (với n là số phần tử có trong mảng)
- Ví dụ: Truy vấn đoạn $[1, 3]$ trong mảng $[1, 2, 3, 4]$.
 - Đệ quy để tìm các đoạn phù hợp.
 - Chính là $[1, 2]$ tương ứng với giá trị 3 và $[3, 3]$ tương ứng với giá trị 3 rồi cộng lại trả về 6.



Hoạt động	Cách thực hiện	Độ phức tạp
Xây dựng (Bottom-up)	Từ lá lên gốc	$O(n)$
Xây dựng (Top-down)	Đệ quy từ gốc	$O(n)$
Cập nhật điểm	Điều chỉnh từ lá lên gốc	$O(\log n)$
Cập nhật đoạn	Lazy Propagation	$O(\log n)$
Truy vấn đoạn	Đệ quy, kết hợp giá trị hai nút con	$O(\log n)$

2.4 Một số kỹ thuật trên cây phân đoạn

Cây phân đoạn (Segment Tree) có thể được tối ưu và mở rộng với nhiều kỹ thuật khác nhau để xử lý các bài toán phức tạp hơn. Dưới đây là mô tả chi tiết một số kỹ thuật nổi bật:

2.4.1 Lazy Propagation

- Mục tiêu: Lazy Propagation là một kỹ thuật tối ưu để xử lý **cập nhật đoạn** $[l, r]$ và **truy vấn đoạn** hiệu quả mà không cần cập nhật toàn bộ cây ngay lập tức.
- Ý tưởng chính:
 - **Trì hoãn việc cập nhật:** Lưu thông tin cập nhật tại nút hiện tại thay vì cập nhật ngay trên các nút con. Việc cập nhật chỉ thực hiện khi thực sự cần (khi truy vấn hoặc tính toán trên đoạn đó).
 - Sử dụng mảng phụ `lazy[]` để lưu các cập nhật bị trì hoãn.
- Độ phức tạp:
 - Cập nhật đoạn: $O(\log n)$.
 - Truy vấn đoạn: $O(\log n)$.
- Cách hoạt động:
 1. **Cập nhật đoạn $[l, r]$:**
 - Lưu giá trị cần cập nhật vào `lazy[]` tại nút hiện tại.
 - Không thực hiện cập nhật ngay trên các nút con, giảm công việc đệ quy.
 2. **Khi truy vấn hoặc cần cập nhật thực tế:**
 - Kiểm tra và "đẩy" các giá trị trong `lazy[]` xuống các nút con.
 - Cập nhật giá trị tại các nút liên quan.

2.4.2 Persistent Segment Tree

- Mục tiêu: Persistent Segment Tree cho phép lưu trạng thái của cây tại mỗi thời điểm (phiên bản), hỗ trợ truy vấn trên các phiên bản trước đó.
- Ý tưởng chính:
 - Khi cập nhật, thay vì sửa trực tiếp, tạo ra một phiên bản mới bằng cách sao chép và sửa đổi các nút bị ảnh hưởng.
 - Các nút không bị thay đổi được tái sử dụng để tiết kiệm bộ nhớ.
- Độ phức tạp:
 - Cập nhật: $O(\log n)$.
 - Truy vấn: $O(\log n)$.
 - Bộ nhớ: $O(n \log n)$.
- Cách hoạt động:
 1. Khi cập nhật hoặc thêm phiên bản mới:
 - Tạo nút mới cho các nút bị ảnh hưởng.
 - Sao chép các nút không bị thay đổi từ phiên bản trước.
 2. Mỗi phiên bản lưu trữ gốc (root) của cây mới, cho phép truy cập lại trạng thái cũ.

2.4.3 Mergeable Segment Tree

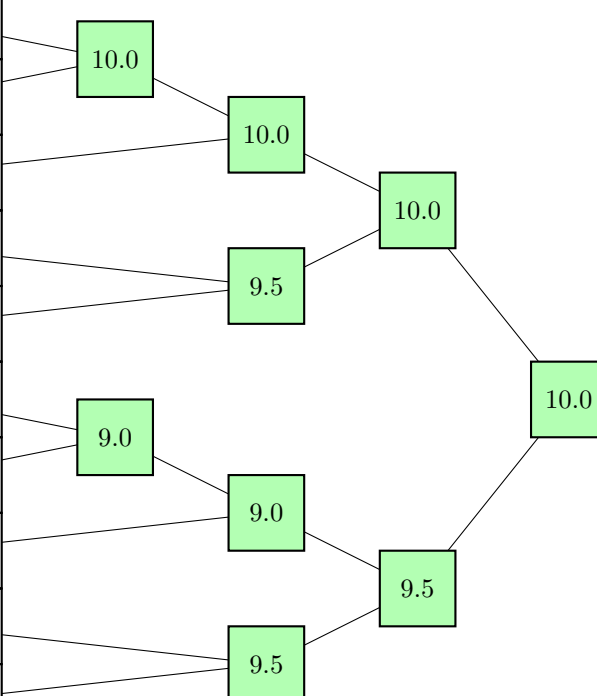
- Mục tiêu: Hỗ trợ hợp nhất hai cây phân đoạn (ứng với hai đoạn con) thành một cây duy nhất, thường áp dụng trong bài toán trên đồ thị hoặc xử lý nhiều tập hợp dữ liệu.
- Ý tưởng chính:
 - Sử dụng phép hợp (*merge*) để kết hợp thông tin từ hai cây con thành một cây mới.
 - Các phép toán phổ biến: tổng, max, min, gcd, ...
- Độ phức tạp:
 - Hợp nhất hai cây: $O(\log n)$.

Tóm tắt

Kỹ thuật	Ứng dụng	Độ phức tạp (Cập nhật / Truy vấn)
Lazy Propagation	Cập nhật đoạn hiệu quả	$O(\log n)$
Persistent Segment Tree	Lưu nhiều trạng thái	$O(\log n)$
Mergeable Segment Tree	Hợp nhất hai cây	$O(\log n)$

3 Ứng dụng vào quản lý điểm số của sinh viên trong một hệ thống học tập

STT	Tên sinh viên	Điểm
1	Nguyễn Xuân Huy Hoàng 1	8.0
2	Nguyễn Xuân Huy Hoàng 2	10.0
3	Nguyễn Xuân Huy Hoàng 3	8.0
4	Nguyễn Xuân Huy Hoàng 4	9.5
5	Nguyễn Xuân Huy Hoàng 5	6.5
6	Nguyễn Xuân Huy Hoàng 6	7.5
7	Nguyễn Xuân Huy Hoàng 7	9.0
8	Nguyễn Xuân Huy Hoàng 8	4.0
9	Nguyễn Xuân Huy Hoàng 9	7.5
10	Nguyễn Xuân Huy Hoàng 10	9.5



1. Truy vấn tổng điểm nhanh chóng

Bài toán: Tính tổng điểm của một nhóm sinh viên trong khoảng $[L, R]$.

Lợi ích:

- Thời gian truy vấn là $O(\log n)$, nhanh hơn rất nhiều so với duyệt tuyến tính $O(n)$.
- Hệ thống có thể xử lý hàng nghìn hoặc hàng triệu sinh viên mà vẫn đảm bảo hiệu năng cao.

2. Cập nhật điểm linh hoạt và hiệu quả

Bài toán: Khi sinh viên được cập nhật điểm, cần cập nhật lại tổng điểm của nhóm sinh viên.

Lợi ích:

- Segment Tree hỗ trợ **cập nhật một phần tử hoặc cả đoạn** trong $O(\log n)$.
- Giúp hệ thống dễ dàng cập nhật khi điểm số thay đổi, không cần tính toán lại toàn bộ.

3. Hỗ trợ đa dạng loại truy vấn

Các bài toán:

- Tìm **điểm cao nhất (max)**, **thấp nhất (min)**, hoặc **điểm trung bình** trong một đoạn.
- Truy vấn **tổng điểm** hoặc **đếm số sinh viên** có điểm trong khoảng $[L, R]$.

Lợi ích:

- Chỉ cần điều chỉnh Segment Tree, có thể giải quyết nhiều dạng bài toán khác nhau.

4. Quản lý dữ liệu thời gian thực (Real-time)

Khi có nhiều yêu cầu truy vấn và cập nhật liên tục, Segment Tree đảm bảo **xử lý nhanh và trả về kết quả ngay lập tức**.

Hệ thống sẽ hoạt động hiệu quả, tránh chậm trễ khi truy cập dữ liệu lớn.

5. Hỗ trợ cập nhật đoạn (Lazy Propagation)

Bài toán: Cập nhật điểm cho **toàn bộ sinh viên trong khoảng** $[L, R]$.

Lợi ích:

- Lazy Propagation cho phép cập nhật nhanh toàn bộ đoạn mà không cần duyệt từng sinh viên.
- Giảm thiểu thời gian cập nhật từ $O(n)$ xuống $O(\log n)$.

6. Tiết kiệm bộ nhớ và dễ triển khai

Lợi ích:

- Segment Tree sử dụng bộ nhớ khoảng $4n$, phù hợp để triển khai trên các hệ thống lớn.
- Cấu trúc đơn giản, dễ mở rộng và bảo trì.

Ví dụ cụ thể

Bài toán:

- Truy vấn tổng điểm của sinh viên từ thứ 200 đến thứ 500.
- Cập nhật điểm số của sinh viên thứ 300.
- Cộng thêm 5 điểm cho toàn bộ sinh viên từ thứ 100 đến thứ 400.

Giải pháp với Segment Tree:

- Truy vấn tổng điểm: $O(\log n)$.
- Cập nhật điểm cá nhân hoặc cả đoạn: $O(\log n)$.
- Hệ thống chạy nhanh ngay cả khi số sinh viên tăng lên hàng triệu.

4 Mã nguồn cho hệ thống quản lý điểm số của sinh viên

```
1 // nxhoang - the dreamer
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 struct Student {
6     string name;
7     double score;
8     Student(string _name, double _score) : name(_name), score(_score) {}
9
10    bool operator<(const Student& other) const { return score < other.score; }
11    bool operator>(const Student& other) const { return score > other.score; }
12    bool operator==(const Student& other) const { return score == other.score && name
13    ↪ == other.name; }
14    bool operator<=(const Student& other) const { return *this < other || *this ==
15    ↪ other; }
16    bool operator>=(const Student& other) const { return *this > other || *this ==
17    ↪ other; }
18    Student operator+(const Student& other) const { return Student(name, score +
19    ↪ other.score); }
20 };
21 static double cur = 0;
22
23 Student getMax(const Student& a, const Student& b) {cur = 0; return (a.score >
24 ↪ b.score) ? a : b; }
25
26 Student getMin(const Student& a, const Student& b) {cur = 1000; return (a.score <
27 ↪ b.score) ? a : b; }
28
29 Student getSum(const Student& a, const Student& b) { return a + b; }
30
31 class SegmentTree {
32 private:
33     vector<Student> tree;
34     vector<Student> lazy;
35     function<Student(const Student&, const Student&)> Function;
36
37     void build(int idx, int l, int r) {
38         if (l == r) {
39             string name;
40             double score;
41             cin >> name >> score;
42             tree[idx] = Student(name, score);
43             return;
44         }
45
46         int mid = (l + r) / 2;
47         build(2 * idx + 1, l, mid);
48         build(2 * idx + 2, mid + 1, r);
49
50         tree[idx] = Function(tree[2 * idx + 1], tree[2 * idx + 2]);
51     }
52
53 public:
54     SegmentTree(int n, function<Student(const Student&, const Student&)> func) :
55         Function(func) {
```

```
49     tree.resize(4 * n, Student("", cur));
50     lazy.resize(4 * n, Student("", cur));
51     build(0, 0, n - 1);
52 }
53
54 void pushLazy(int idx, int l, int r) {
55     if (lazy[idx].score != 0) {
56         tree[idx].score = lazy[idx].score;
57         if (l < r) {
58             lazy[2 * idx + 1].score = lazy[idx].score;
59             lazy[2 * idx + 2].score = lazy[idx].score;
60         }
61         lazy[idx].score = 0;
62     }
63 }
64
65 void update(int idx, int l, int r, int x, int y, double val) {
66     pushLazy(idx, l, r);
67     if (y < l || r < x) return;
68
69     if (l >= x && r <= y) {
70         lazy[idx].score = val;
71         pushLazy(idx, l, r);
72         return;
73     }
74
75     int mid = (l + r) / 2;
76     update(2 * idx + 1, l, mid, x, y, val);
77     update(2 * idx + 2, mid + 1, r, x, y, val);
78     tree[idx] = Function(tree[2 * idx + 1], tree[2 * idx + 2]);
79 }
80
81 Student query(int idx, int l, int r, int x, int y) {
82     pushLazy(idx, l, r);
83     if (r < x || y < l) return Student("", cur);
84
85     if (l >= x && r <= y) return tree[idx];
86
87     int mid = (l + r) / 2;
88     return Function(query(2 * idx + 1, l, mid, x, y), query(2 * idx + 2, mid + 1,
89         ↪ r, x, y));
90 }
91
92
93 int main() {
94     int n, q;
95     cin >> n >> q;
96
97     unordered_map<string, function<Student(const Student&, const Student&)>>
98     ↪ operations = {
99         {"max", getMax},
100         {"min", getMin},
101         {"sum", getSum}
102     };
103
104     string operationType;
```

```
104     cin >> operationType;
105
106     SegmentTree st(n, operations[operationType]);
107
108     while(q--){
109         int typE, left, right;
110         cin >> typE >> left >> right;
111         if (typE == 1) {
112             double x;
113             cin >> x;
114             st.update(0, 0, n - 1, left - 1, right - 1, x);
115         } else {
116             Student stu = st.query(0, 0, n - 1, left - 1, right - 1);
117             cout << std::left << setw(10) << stu.name << std::right << setw(5) <<
118                 stu.score << endl;
119         }
120     }
121     return 0;
122 }
```

Hệ thống này quản lý điểm số của sinh viên thông qua một cấu trúc dữ liệu cây phân đoạn (Segment Tree). Các sinh viên được lưu trữ dưới dạng đối tượng **Student** bao gồm tên và điểm số. Các phép toán trên cây phân đoạn như tìm giá trị lớn nhất, nhỏ nhất hoặc tổng điểm trong một khoảng được thực hiện thông qua các hàm khác nhau.

4.1 Cấu Trúc Dữ Liệu Sinh Viên

struct Student đại diện cho một sinh viên, với các thuộc tính là **name** (tên) và **score** (điểm số). Các toán tử được định nghĩa cho lớp **Student** như sau:

- **operator<** so sánh điểm số giữa hai sinh viên.
- **operator>** so sánh ngược lại điểm số giữa hai sinh viên.
- **operator==** kiểm tra xem hai sinh viên có điểm số và tên giống nhau hay không.
- **operator<=** và **operator>=** sử dụng **operator<** và **operator==** để xác định mối quan hệ giữa hai sinh viên.
- **operator+** cộng điểm số của hai sinh viên và trả về một sinh viên mới với tên của một sinh viên và điểm số là tổng của điểm số.

4.2 Các Hàm Phép Toán

- **getMax**: Trả về sinh viên có điểm số cao hơn giữa hai sinh viên.
- **getMin**: Trả về sinh viên có điểm số thấp hơn giữa hai sinh viên.
- **getSum**: Trả về sinh viên có điểm số là tổng điểm số của hai sinh viên.

4.3 Cây Phân Đoạn

class SegmentTree là lớp đại diện cho cây phân đoạn, dùng để xử lý các phép toán trên một dãy sinh viên. Cây phân đoạn lưu trữ các sinh viên ở các nút của nó và hỗ trợ các phép toán như tìm giá trị lớn nhất, nhỏ nhất hoặc tổng điểm số trong một khoảng.

4.3.1 Cấu Trúc Cây Phân Đoạn

Cây phân đoạn lưu trữ:

- **tree**: Mảng lưu trữ các sinh viên ở các nút của cây.
- **lazy**: Mảng lưu trữ các giá trị "lười" (lazy propagation) để tối ưu hóa các thao tác cập nhật.
- **Function**: Hàm phép toán được áp dụng lên các sinh viên trong cây phân đoạn (max, min, sum).

4.3.2 Phương Thức Chính

- **build**: Xây dựng cây phân đoạn từ dữ liệu đầu vào. Đối với mỗi phạm vi, cây phân đoạn lưu trữ thông tin về sinh viên tại đó.
- **pushLazy**: Xử lý các phép toán "lười" để đảm bảo rằng mọi cập nhật đều được áp dụng một cách chính xác.
- **update**: Cập nhật giá trị trong một phạm vi của cây phân đoạn. Nếu có phép toán "lười", nó sẽ được đẩy xuống các nút con của cây.
- **query**: Truy vấn giá trị trong một phạm vi của cây phân đoạn.

4.4 Chương Trình Chính

Chương trình chính thực hiện các bước sau:

- Nhập số lượng sinh viên n và số lượng truy vấn q .
- Chọn loại phép toán (max, min, sum) để áp dụng trong cây phân đoạn.
- Xây dựng cây phân đoạn với dữ liệu sinh viên và phép toán đã chọn.
- Xử lý các truy vấn:
 - Nếu là truy vấn cập nhật ($type = 1$), cập nhật điểm cho sinh viên trong phạm vi từ **left** đến **right**.
 - Nếu là truy vấn truy xuất ($type = 2$), tìm và in ra sinh viên có tên và điểm số trong phạm vi từ **left** đến **right**.

4.5 Kết Luận

Hệ thống quản lý điểm số của sinh viên sử dụng cây phân đoạn để xử lý hiệu quả các phép toán trên dãy điểm số của sinh viên, đặc biệt là các phép toán yêu cầu truy vấn và cập nhật trong phạm vi. Các phép toán như tìm giá trị lớn nhất, nhỏ nhất hoặc tổng điểm trong một phạm vi được thực hiện thông qua các hàm hàm số học trên các đối tượng **Student**.

4.6 Kiểm thử

4.6.1 Testcase 1

Input

```
1 10 5
2 max
3 Hoang 7.6
4 Huyen 9.1
5 Hien 8.3
6 Duc 4.2
7 Linh 3.9
8 Nhin 4.5
9 Luc 6.5
10 Nhuc 5.6
```



```
11 Dung 10.0
12 Lam 9.0
13 2 1 10
14 2 1 5
15 1 1 3 9.3
16 2 2 4
17 1 3 6 4.0
```

Output

```
1 Dung      10
2 Huyen     9.1
3 Hien      9.3
```

4.6.2 Testcase 2

Input

```
1 20 10
2 min
3 Thang 8.1
4 Kien 6.9
5 Hoa 7.4
6 Minh 9.2
7 Lan 8.5
8 Anh 7.1
9 Duy 6.4
10 Tuan 7.8
11 Hoa 8.0
12 Mai 7.9
13 Quang 9.0
14 Chien 7.6
15 Hieu 8.3
16 Le 6.7
17 Thao 8.9
18 Tam 9.3
19 Hien 7.2
20 Khang 8.7
21 Lam 9.1
22 Son 7.0
23 2 1 5
24 1 3 8 8.5
25 2 1 10
26 1 2 6 9.0
27 2 5 10
28 1 8 15 6.3
29 2 3 12
30 1 6 9 7.7
31 2 1 6
32 2 4 7
```

Output

```
1 Kien      6.9
2 Kien      6.9
3 Mai       7.9
```



4	Chien	6.3
5	Anh	7.7
6	Duy	7.7

5 Kết luận

Segment Tree là một cấu trúc dữ liệu mạnh mẽ và linh hoạt, đóng vai trò quan trọng trong việc giải quyết các bài toán truy vấn và cập nhật trên đoạn một cách hiệu quả. Với khả năng xử lý các phép tính trên đoạn như tổng, giá trị lớn nhất, nhỏ nhất hoặc đếm phần tử, Segment Tree giúp giảm thiểu thời gian truy vấn và cập nhật từ $O(n)$ xuống $O(\log n)$.

Trong các hệ thống lớn như quản lý điểm số sinh viên, quản lý dữ liệu thời gian thực hay các bài toán cần cập nhật liên tục, việc sử dụng Segment Tree không chỉ đảm bảo hiệu năng cao mà còn tối ưu bộ nhớ và dễ triển khai. Tính năng mở rộng như Lazy Propagation giúp giải quyết các bài toán cập nhật đoạn nhanh chóng, làm cho Segment Tree trở thành công cụ không thể thiếu trong các ứng dụng thực tế.

Việc nắm vững và triển khai Segment Tree một cách chính xác sẽ mở ra nhiều hướng giải quyết cho các bài toán phức tạp, từ đó cải thiện đáng kể hiệu suất hệ thống và tiết kiệm tài nguyên.

6 References

Trích dẫn *et. al.* [1, 2, 3]

References

- [1] Nguyễn Châu Khanh, “Segment tree cơ bản.” <https://wiki.vnoi.info/algo/data-structures/segment-tree-basic.md>. VNU University of Engineering and Technology (VNU-UET).
- [2] CP-Algorithms Team, “Segment tree.” https://cp-algorithms.com/data_structures/segment_tree.html, 2023.
- [3] GeeksforGeeks Team, “Segment tree data structure.” <https://www.geeksforgeeks.org/segment-tree-data-structure/>, 2024.