Data Structures and algorithms
Instructor : **Dr.Amin Eskandari**
Head-Ta's : Hamid Namjoo manesh , Amir Hossein Hemati , Ali Mojahed
Ta's : Amir Mohammad Asadjoo , Padina razmjooi , Armin Janfaza , Alireza Heydari , Mahdi Torabi ,
Maryam Ghorbani , Fatemeh Keshavarz , MohammadReza Fasli , Abolfazl Tadrisi

Week 09 Answers

پاسخنامه تکالیف هفته ۰۹

پاسخ سوال اول :

۱. تحلیل اولیه مسئله و چالش‌ها

ما با یک آرایه به طول nسروکار داریم و دو نوع عملیات از ما خواسته شده است:

۱. اضافه کردن مقدار vبه تمام عناصر بازه $[l, r]$

۲. محاسبه مجموع عناصر در بازه $[l, r]$

چرا روش‌های ساده (Naive) کار نمی‌کنند؟

اگر بخواهیم با یک حلقه ساده (Loop) روی آرایه حرکت کنیم، برای اضافه کردن مقادیر یا محاسبه مجموع، در بدترین حالت باید کل آرایه را پیمایش کنیم. زمان این کار $O(n)$خواهد بود. با توجه به اینکه تعداد درخواست‌ها (q) و اندازه آرایه (n) تا $200{,}000$می‌رسد، زمان کل برنامه در بدترین حالت $O(n \times q)$یعنی حدود 4×10^{10}عملیات خواهد شد که قطعاً باعث خطای محدودیت زمان (Time Limit Exceeded) می‌شود. ما به روشی نیاز داریم که هر دو عملیات را در زمان $O(\log n)$انجام دهد.

---

۲. استراتژی حل: تبدیل مجموع بازه‌ای به مجموع پیشوندی (Prefix Sum)

می‌دانیم که مجموع عناصر در بازه $[l, r]$را می‌توان با استفاده از تفاضل دو مجموع پیشوندی محاسبه کرد. اگر تابع $S(x)$را برابر با مجموع عناصر از اندیس 1تا اندیس xتعریف کنیم، آنگاه:

$$Sum(l, r) = S(r) - S(l - 1)$$

پس مسئله اصلی ما به این تبدیل می‌شود: **چگونه تابع $S(x)$(مجموع پیشوندی تا اندیس x) را در زمان $O(\log n)$محاسبه کنیم، در حالی که آرایه مدام در حال تغییر است؟**

---

۳. قلب تپنده راه‌حل: کشف فرمول ریاضی با آرایه تفاضلی

برای اینکه بتوانیم روی یک بازه $[l, r]$به سرعت مقدار vرا اضافه کنیم، از تکنیک «آرایه تفاضلی» (Difference Array) استفاده می‌کنیم. فرض کنید آرایه‌ای به نام Dداریم.

اگر بخواهیم به بازه $[l, r]$مقدار vرا اضافه کنیم، در آرایه تفاضلی فقط دو تغییر ایجاد می‌کنیم:

1. $D[l] = D[l] + v$
2. $D[r+1] = D[r+1] - v$

ارتباط آرایه اصلی (A) با آرایه تفاضلی (D) به این شکل است که هر عنصر $A[x]$برابر است با مجموع عناصر Dتا اندیس x:

$$A[x] = \sum_{i=1}^{x} D[i]$$

حالا می‌خواهیم مجموع پیشوندی یعنی $S(x)$را بر اساس آرایه Dمحاسبه کنیم. مجموع پیشوندی برابر است با مجموع تمام $A[i]$ها تا اندیس x:

$$S(x) = \sum_{i=1}^{x} A[i] = \sum_{i=1}^{x} \sum_{j=1}^{i} D[j]$$

اگر این مجموع دوگانه را روی کاغذ باز کنیم، الگوی بسیار زیبایی پدیدار می‌شود:

عنصر $D[1]$در محاسبه $A[1]$تا $A[x]$حضور دارد (یعنی xبار تکرار می‌شود).

عنصر $D[2]$در محاسبه $A[2]$تا $A[x]$حضور دارد (یعنی $x-1$بار تکرار می‌شود).

به طور کلی، عنصر $D[i]$دقیقاً $(x - i + 1)$بار تکرار می‌شود.

پس می‌توانیم فرمول را به شکل زیر ساده کنیم:

$$S(x) = \sum_{i=1}^{x} D[i] \times (x - i + 1)$$

حالا برای اینکه بتوانیم این را با درخت فن‌ویک محاسبه کنیم، فرمول را پخش (Expand) می‌کنیم:

$$S(x) = \sum_{i=1}^{x} (D[i] \times x - D[i] \times (i - 1))$$

$$S(x) = x \times \left(\sum_{i=1}^{x} D\,[i] \right) - \left(\sum_{i=1}^{x} D\,[i] \times (i - 1) \right)$$

این فرمول نهایی است! این معادله به ما فریاد می‌زند که برای محاسبه مجموع پیشوندی، به **دو درخت فن‌ویک مستقل** نیاز داریم.

---

۴. نقش هر درخت فن‌ویک در کد شما

با توجه به فرمول بالا، شما در کدتان دو شیء از کلاس Fenwick ساخته‌اید: bit1 و bit2.

درخت اول (bit1):

وظیفه این درخت نگهداری قسمت اول فرمول، یعنی $\sum_{i=1}^{x} D[i]$ است.

وقتی دستور ADD l r v می‌آید، شما در تابع range_add به این شکل عمل می‌کنید:

- به اندیس lمقدار vاضافه می‌کنید.
- به اندیس $r + 1$مقدار $-v$اضافه می‌کنید.

درخت دوم (bit2):

وظیفه این درخت نگهداری قسمت دوم فرمول، یعنی $\sum_{i=1}^{x} D[i] \times (i-1)$ است.

وقتی دستور ADD l r v می‌آید، طبق این الگو، باید مقدار تغییرات را ضربدر $(i-1)$کنیم:

- برای اندیس l(که اندیس شروع تغییر است، یعنی $i = l$)، باید مقدار $v \times (l-1)$را اضافه کنید.
- برای اندیس $r+1$(که اندیس پایان تغییر است، یعنی $i = r+1$)، باید مقدار $-v \times ((r+1)-1) = -v \times r$را اضافه کنید.

کد شما در تابع range_add دقیقاً همین دو مرحله را بی‌نقص اجرا کرده است.

---

۵. پیاده‌سازی کوئری‌ها (پرس‌وجو)

با داشتن این دو درخت، تابع prefix_sum(x) به راحتی همان فرمول ریاضی را ترجمه می‌کند:

```
return x * bit1.query(x) - bit2.query(x)
```

و تابع range_sum(l, r) نیز تفاضل دو مجموع پیشوندی را برمی‌گرداند:

```
return prefix_sum(r) - prefix_sum(l - 1)
```

---

۶. مقدار دهی اولیه (Initialization)

آرایه در ابتدا خالی نیست و مقادیری در Aوجود دارد. به جای اینکه برای مقداردهی اولیه یک منطق پیچیده بنویسید، شما بسیار هوشمندانه عمل کرده‌اید:

یک حلقه for نوشته‌اید و فرض کرده‌اید که هر عنصر اولیه $A[i]$، در واقع یک عملیات ADD i i A[i] است (یعنی اضافه کردن مقدار به بازه‌ای به طول ۱). با این کار، مقادیر اولیه دقیقاً وارد هر دو درخت فن‌ویک می‌شوند و سیستم آماده پاسخگویی به درخواست‌های بعدی می‌گردد.

---

۷. پیچیدگی زمانی و فضایی (Complexity Analysis)

- **پیچیدگی زمانی:**

عملیات حرکت روی گره‌های درخت فن‌ویک با استفاده از بیت‌آرایی idx -= idx & -idx انجام می‌شود که پرش‌هایی روی توان‌های عدد ۲ است. بنابراین ارتفاع درخت همیشه لگاریتمی است.

زمان هر دستور ADD: $O(\log n)$

زمان هر دستور SUM: $O(\log n)$

زمان ساخت اولیه آرایه: $O(n \log n)$

زمان کل برنامه برای qکوئری: $O((n+q)\log n)$که برای $n, q = 200{,}000$فوق‌العاده سریع است و شرط سوال را پاس می‌کند.

- **پیچیدگی فضایی (Memory):**

ما دو آرایه bit1 و bit2 از اندازه $n+1$ساخته‌ایم. پس حافظه مصرفی از مرتبه $O(n)$است که بسیار بهینه است.

نتیجه‌گیری نهایی

این راه‌حل یکی از کلاسیک‌ترین و در عین حال قدرتمندترین روش‌ها در برنامه‌نویسی رقابتی برای حل مسائل Range Update / Range Query است. نسبت به درخت سگمنت (Segment Tree)، پیاده‌سازی این روش با درخت فن‌ویک بسیار کوتاه‌تر، حافظه مصرفی آن کمتر و ثابت زمانی (Time Constant) آن به شدت پایین‌تر است. کد نوشته شده از نظر ریاضی و منطق کاملاً بی‌نقص است.

autorenewthumb_upthumb_down

پاسخ سوال دوم :

۱. تحلیل مشکل روش‌های ساده (Naive Approach)

اگر بخواهیم این مسئله را با یک آرایه‌ی دوبعدی ساده به ابعاد $n \times m$حل کنیم:

- عملیات ADD x y v: در زمان $O(1)$انجام می‌شود.
- عملیات SUM x1 y1 x2 y2: نیازمند دو حلقه‌ی تودرتو برای پیمایش کل مستطیل است. در بدترین حالت (گزارش‌گیری از کل دشت)، این عملیات به زمان $O(n \times m)$نیاز دارد.

با توجه به اینکه ابعاد زمین حداکثر 1000×1000است، هر کوئری ممکن است 1,000,000عملیات بطلبد. برای $q = 200{,}000$درخواست، تعداد کل عملیات‌ها به 2×10^{11}می‌رسد که اجرای آن ساعت‌ها طول می‌کشد و باعث خطای Time Limit Exceeded (TLE) می‌شود.

۲. استراتژی حل: درخت فن‌ویک دوبعدی (D Fenwick Tree2)

برای فرار از غضب پادشاه، باید ساختاری داشته باشیم که هم آپدیت و هم کوئری را سریع انجام دهد. درخت فن‌ویک (Binary Indexed Tree) که در یک بُعد زمان $O(\log n)$را ارائه می‌دهد، به راحتی قابل تعمیم به دو بُعد است. در این حالت، ما یک آرایه‌ی دوبعدی bit[n+1][m+1] می‌سازیم.

هر خانه‌ی این درخت، مجموع محصولات یک زیرمستطیل خاص از دشت را در خود ذخیره می‌کند.

۳. کالبدشکافی عملیات‌ها در D Fenwick Tree2

الف) به‌روزرسانی (Update - کاشت بذر جادویی):

وقتی می‌خواهیم مقدار vرا به نقطه‌ی (x, y)اضافه کنیم، باید تمام مستطیل‌هایی در درخت فن‌ویک که این نقطه را پوشش می‌دهند، آپدیت کنیم.

- برای پیمایش سطرها از xشروع می‌کنیم و با استفاده از فرمول جادوییِ $i \leftarrow i + (i \ \& \ -i)$به سمت بالا می‌رویم.
- به ازای هر سطر i، برای ستون‌ها از yشروع کرده و با فرمول $j \leftarrow j + (j \ \& \ -j)$در آن بُعد حرکت می‌کنیم و مقدار vرا به bit[i][j] اضافه می‌نماییم.

این کار در زمان $O(\log n \times \log m)$انجام می‌شود.

ب) محاسبه مجموع پیشوندی (Prefix Sum Query):

تابع query(x, y) مجموع تمام خانه‌های مستطیلی که از گوشه‌ی $(1, 1)$تا گوشه‌ی (x, y)امتداد دارد را برمی‌گرداند.

- برای این کار دقیقاً برعکس آپدیت عمل می‌کنیم و با کم کردن کم‌ارزش‌ترین بیت روشن یعنی $i \leftarrow i - (i \ \& \ -i)$مسیر را به سمت مبدأ می‌پیماییم.

ج) محاسبه مستطیل دلخواه (اصل شمول و عدم شمول):

پادشاه مجموع خانه‌های (x_1, y_1)تا (x_2, y_2)را می‌خواهد. ما تابع query را داریم که فقط از مبدأ $(1, 1)$حساب می‌کند. در اینجا از یک اصل زیبای ریاضی (Inclusion-Exclusion Principle) استفاده می‌کنیم:

$$S = \text{query}(x_2, y_2) - \text{query}(x_1 - 1, y_2) - \text{query}(x_2, y_1 - 1) + \text{query}(x_1 - 1, y_1 - 1)$$

- ابتدا کل مستطیل از مبدأ تا (x_2, y_2)را می‌گیریم.
- سپس مساحت اضافه‌ی سمت چپ $(x_1 - 1, y_2)$را کم می‌کنیم.
- سپس مساحت اضافه‌ی بالای مستطیل $(x_2, y_1 - 1)$را کم می‌کنیم.
- چون تقاطع این دو بخشِ کم‌شده (یعنی مستطیل مبدأ تا $(x_1 - 1, y_1 - 1)$) دو بار کم شده است، آن را یک بار اضافه می‌کنیم تا تعادل برقرار شود.

۴. تحلیل پیچیدگی (Complexity Analysis)

- **پیچیدگی زمانی (Time Complexity):**
- هر آپدیت: $O(\log n \times \log m)$
- هر کوئری (چون ۴ بار تابع پایه صدا زده می‌شود): $O(4 \times \log n \times \log m) = O(\log n \times \log m)$
- **زمان کل:** برای qعملیات برابر است با $O(q \times \log n \times \log m)$. با توجه به اعداد مسئله ($200{,}000 \times 10 \times 10 \approx 2 \times 10^7$ عملیات)، کد در کسری از ثانیه اجرا می‌شود.
- **پیچیدگی فضایی (Space Complexity):**
- ما تنها به یک آرایه دوبعدی bit نیاز داریم که اندازه آن $(n + 1) \times (m + 1)$است.
- پیچیدگی حافظه $O(n \times m)$خواهد بود که برای محدودیت‌های فعلی (حدود یک میلیون خانه صحیح) بسیار بهینه است.

پاسخ سوال سوم :

مسئله از ما می‌خواهد تا دو عملیات را روی یک آرایه با طول nمدیریت کنیم: تغییر مقدار یک عنصر در یک اندیس مشخص (Update) و شمارش تعداد جفت‌های وارونگی در یک بازهٔ مشخص $[l, r]$(Query).

چرا روش‌های کلاسیک کار نمی‌کنند؟

- **روش $O(n^2)$:** بررسی تمام جفت‌های ممکن در یک بازه برای هر کوئری، با توجه به محدودیت $n, q \leq 200000$منجر به خطای محدودیت زمان (Time Limit Exceeded - TLE) می‌شود.
- **روش استفاده از فقط یک Fenwick Tree:** در حالت عادی می‌توان با استفاده از یک Fenwick Tree و مرتب‌سازی، تعداد وارونگی‌های کل آرایه را در زمان $O(n\log\ n)$پیدا کرد. اما وقتی نیاز به محاسبه وارونگی در **زیربازه‌های دلخواه** و اعمال **بروزرسانی‌های نقطه‌ای** داریم، ساخت مجدد یا کپی کردن درخت برای هر کوئری زمان $O(n\log\ n)$می‌برد که در مجموع برای qکوئری به $O(q \times n\log\ n)$می‌رسد و مجدداً باعث TLE می‌شود.

به همین دلیل، ما به یک ساختار دادهٔ دو-بُعدی یا ترکیبی نیاز داریم که بتواند هم اطلاعات بازه‌های مکانی (اندیس‌ها) و هم اطلاعات مقادیر (اعداد درون آرایه) را به صورت همزمان با پیچیدگی لگاریتمی مدیریت کند.

۲. پیش‌نیاز حیاتی: فشرده‌سازی مختصات (Coordinate Compression)

مقادیر عناصر آرایه (و همچنین مقادیری که در عملیات Update داده می‌شوند) می‌توانند تا 10^9و یا حتی منفی باشند ($-10^9 \leq A[i] \leq 10^9$). ساختار Fenwick Tree بر اساس مقادیر اعداد به عنوان اندیس کار می‌کند و نمی‌توانیم آرایه‌ای به طول 2×10^9در حافظه داشته باشیم (خطای Memory Limit Exceeded).

بنابراین، پیش از شروع ساخت درخت‌ها، تمامی مقادیر اولیه آرایه و تمام مقادیر جدیدی که در طول کوئری‌های UPDATE قرار است وارد شوند را در یک مجموعه (Set) جمع‌آوری می‌کنیم تا مقادیر تکراری حذف شوند. سپس آن‌ها را مرتب کرده و به هر مقدار واقعی، یک "رتبه" یا "شناسه فشرده‌سازی شده" از 1تا M(که Mتعداد کل مقادیر یکتای ممکن است) اختصاص می‌دهیم. از این پس، تمام ساختارهای داده ما با این رتبه‌های کوچک (که نهایتاً در حدود $n + q$یعنی 400000هستند) کار می‌کنند.

۳. معماری ساختار داده ترکیبی

هسته اصلی راه حل، استفاده از یک **Segment Tree** روی بازه‌های اندیس‌های آرایه است که در **هر نودِ آن، یک Fenwick Tree** قرار دارد.

- **نقش Segment Tree (بُعد اول - اندیس‌ها):** بازهٔ کل آرایه (از 1تا n) را به دو نیمه تقسیم می‌کند و این کار را تا رسیدن به تک‌عنصرها (برگ‌ها) ادامه می‌دهد. هر نود در این درخت نماینده یک بازه مانند $[L, R]$از اندیس‌های آرایه است.
- **نقش Fenwick Tree در هر نود (بُعد دوم - مقادیر):** درون هر نودِ Segment Tree، یک Fenwick Tree مجزا وجود دارد. وظیفه این درخت، نگهداری فراوانی (Frequency) اعدادی است که در بازه $[L, R]$وجود دارند.

نگهداری اطلاعات وارونگی:

علاوه بر Fenwick Tree، هر نودِ Segment Tree باید یک متغیر عددی برای ذخیره **«تعداد وارونگی‌های داخلی»** بازهٔ خود داشته باشد.

برای محاسبه وارونگی داخلی یک نود (که فرزند چپ و راست دارد)، از این فرمول استفاده می‌کنیم:

$$\text{Internal Inversions} = \text{Inversions(Left Child)} + \text{Inversions(Right Child)} + \text{Cross Inversions}$$

وارونگی متقاطع (Cross Inversions) یعنی جفت‌هایی که عنصر بزرگتر در فرزند چپ و عنصر کوچکتر در فرزند راست قرار دارد.

۴. پردازش عملیات به‌روزرسانی (UPDATE i x)

وقتی می‌خواهیم مقدار اندیس iرا از مقدار قدیمی (v_{old}) به مقدار جدید (v_{new}) تغییر دهیم، مراحل زیر از ریشه به سمت پایین (Top-Down) و سپس بازگشت به بالا (Bottom-Up) انجام می‌شود:

1. **پیمایش رو به پایین:** در Segment Tree مسیری را از ریشه تا برگی که نماینده اندیس iاست طی می‌کنیم.
2. **به‌روزرسانی Fenwick Tree ها:** در هر نودی که در این مسیر می‌بینیم (که همگی بازه‌هایی هستند که اندیس iرا شامل می‌شوند)، در Fenwick Tree اختصاصیِ همان نود، فراوانی مقدار v_{old}را یکی کم می‌کنیم و فراوانی مقدار v_{new}را یکی اضافه می‌کنیم.

3. **به‌روزرسانی وارونگی‌های داخلی (در مسیر بازگشت):** وقتی تغییرات در برگ اعمال شد، در مسیر بازگشت به سمت ریشه، باید وارونگی داخلیِ هر نود را آپدیت کنیم.

- تغییر مقدار در اندیس iباعث می‌شود تعدادی وارونگی که با v_{old}ساخته شده بودند از بین بروند و تعدادی وارونگی جدید با v_{new}ساخته شوند.
- برای محاسبه دقیق این تغییرات در یک نود (که اندیس iمثلاً در فرزند چپ آن است)، از Fenwick Tree فرزند راست استفاده می‌کنیم تا بفهمیم چند عنصر از v_{old}کوچکتر/بزرگتر بوده‌اند و چند عنصر از v_{new}کوچکتر/بزرگتر هستند و تفاوت آن‌ها را به وارونگی کل نود اعمال می‌کنیم.

۵. پردازش عملیات پرس‌وجو (QUERY l r)

وقتی می‌خواهیم تعداد کل وارونگی‌ها را در بازه دلخواه $[l, r]$پیدا کنیم، مستقیماً نمی‌توانیم فقط وارونگی‌های یک نود را بخوانیم، زیرا بازه $[l, r]$ممکن است ترکیبی از چندین نود استاندارد در Segment Tree باشد.

1. **تجزیه بازه:** بازه $[l, r]$را در Segment Tree تجزیه می‌کنیم که در بدترین حالت به $O(\log n)$نود استاندارد (از چپ به راست) شکسته می‌شود.
2. **محاسبه مجموع:** مجموع وارونگی‌های کل در این بازه برابر است با:

- مجموع **وارونگی‌های داخلیِ** تمام این $O(\log n)$نود (که از قبل در نودها محاسبه و ذخیره شده است).
- به‌علاوهٔ **وارونگی‌های متقاطعِ بین این نودها**.

3. **محاسبه وارونگی‌های متقاطع بین نودها:** از چپ به راست روی این $O(\log n)$نود حرکت می‌کنیم. برای هر نود، باید بدانیم در نودهایی که سمت چپ آن قرار داشته‌اند، چند عدد وجود دارد که از اعداد داخل نود فعلی به شکل جفتی بزرگتر هستند. این کار با استفاده از کوئری زدن روی Fenwick Tree های مربوط به آن نودها انجام می‌شود.

۶. پیچیدگی زمانی و مکانی (Complexity)

- **زمان فشرده‌سازی:** $O((n+q)\log(n+q))$که فقط یک بار در ابتدا انجام می‌شود.

- **عملیات UPDATE:** درخت Segment Tree دارای عمق $O(\log n)$است. در هر نود از مسیر، یک عملیات روی Fenwick Tree انجام می‌دهیم که آن هم $O(\log M)$زمان می‌برد. بنابراین زمان هر آپدیت $O(\log n \times \log M)$است.
- **عملیات QUERY:** تجزیه به $O(\log n)$نود و محاسبه وارونگی‌های متقاطع با استفاده از Fenwick Treeها، زمان اجرایی برابر با $O(\log^2 n)$دارد.
- **حافظه (Space):** در هر سطح از Segment Tree، عناصر آرایه در Fenwick Treeها پخش شده‌اند. چون عمق درخت $\log n$است و مجموع عناصر در هر سطح برابر nاست، پیچیدگی حافظه دقیقاً $O(n \log n)$خواهد بود که برای $n = 200000$کاملاً در محدوده مجاز و بهینه است.

پاسخ سوال چهارم :

۱. تحلیل صورت مسئله و چالش‌های ذاتی الگوریتم

ما در این مسئله با یک ساختار داده پویا (Dynamic Data Structure) روبرو هستیم که دو وظیفه کلیدی و متضاد بر عهده دارد:

الف) حفظ داده‌ها به صورت مرتب و متعادل در زمان پردازشی $O(\log N)$.

ب) پاسخگویی به این سوال که آیا دو زیردرخت به ریشه‌های Uو Vدقیقاً هم‌شکل و هم‌رنگ (ایزومورف) هستند یا خیر.

روش‌های سنتی برای بررسی ایزومورفیسم دو درخت، مبتنی بر پیمایش همزمان (Simultaneous Traversal) هر دو زیردرخت است. در این روش، اگر سایز زیردرخت‌ها برابر با Kباشد، پیچیدگی زمانی مقایسه $O(K)$خواهد بود. در بدترین حالت که کوئری‌ها روی زیردرخت‌های بزرگ اعمال شوند، پیچیدگی کل به $O(Q \times N)$می‌رسد. این زمان برای تعداد کوئری‌های بالا (مثلاً 10^5کوئری) قطعاً منجر به خطای Time Limit Exceeded (TLE) خواهد شد.

بنابراین، نیازمند مکانیزمی هستیم که فارغ از سایز زیردرخت، در زمان ثابت $O(1)$به کوئری‌ها پاسخ دهد. راهکار قطعی این چالش، استفاده از مفهوم پیشرفته **هشینگ درخت (Tree Hashing)** است.

۲. طراحی ریاضی تابع هش (Hash Function Design)

برای مقایسه دو ساختار پیچیده در زمان $O(1)$، باید به هر زیردرخت یک «اثر انگشت ریاضی» (Hash Value) یکتا اختصاص دهیم. در صورتی که اثر انگشت دو زیردرخت کاملاً یکسان باشد، با تقریب بسیار نزدیک به یقین، آن دو زیردرخت از نظر هندسی و مقادیر یکسان هستند.

تابع هش ما در این درخت باید چهار ویژگی حیاتی گره را در خود جای دهد:

- رنگ گره (C_v): تأثیرگذاری مستقیم رنگ‌های قرمز یا سیاه.
- اندازه زیردرخت (S_v): تعداد کل گره‌های موجود در آن زیردرخت.
- هش فرزند چپ (H_{left}): کدگذاری ساختار زیردرخت سمت چپ.
- هش فرزند راست (H_{right}): کدگذاری ساختار زیردرخت سمت راست.

به منظور ترکیب این پارامترها و به حداقل رساندن احتمال برخورد (Collision)، از اعداد اول بزرگ (Prime Numbers) استفاده شده است. فرمول ریاضی پیاده‌سازی شده در هسته الگوریتم به شرح زیر است:

$$H(v) = ((C_v \times P_{color} + S_v \times P_{size}) + (H_{left} \times P_{left}) + (H_{right} \times P_{right}))(\mathrm{mod} M)$$

نکته بسیار مهم در این فرمول، تفاوت مقادیر P_{left} و P_{right}است (در پیاده‌سازی ما $P_{left} = 313$و $P_{right} = 317$). اگر این دو مقدار یکسان بودند، درختی که فقط یک فرزند چپ دارد، با درختی که دقیقاً همان فرزند را در سمت راست دارد، هش یکسانی تولید می‌کردند؛ در حالی که از نظر هندسی نامتقارن و متفاوت هستند. پیمانه Mنیز برابر با $10^9 + 7$تعیین شده است تا محاسبات از محدوده اعداد صحیح فراتر نرفته و از سرریز حافظه (Overflow) جلوگیری شود.

۳. مهندسی به‌روزرسانی هش‌ها در درخت قرمز-سیاه پویا

تولید هش برای یک درخت ایستا کار ساده‌ای است، اما چالش اصلی زمانی آغاز می‌شود که درخت در حال چرخش و تغییر رنگ است و ما باید هش‌ها را همواره معتبر و به‌روز نگه داریم. درخت قرمز-سیاه (RBT) برای حفظ ارتفاع خود در محدوده $2\log N$، پس از هر درج از دو عملیات استفاده می‌کند:

الف) مدیریت چرخش‌ها (Rotations):

زمانی که چرخش به چپ روی گره Xانجام می‌شود، Xبه سطح پایین‌تری می‌رود و فرزند راست آن، یعنی Y، جایگاه او را می‌گیرد. در این حالت، زیردرخت‌های Xو Yدستخوش تغییرات بنیادین می‌شوند. در پیاده‌سازی متدهای چرخش، همواره ابتدا تابع آپدیت برای گره X(که اکنون فرزند شده) و سپس برای گره Y(که اکنون پدر شده) فراخوانی می‌شود. این رعایت تقدم و تأخر، از اصول بنیادین برنامه‌نویسی پویا روی درخت (Tree DP) است.

ب) مدیریت تغییر رنگ‌ها (Color Flips):

در طول عملیات ترمیم (Fixup)، هر زمان که قانون توالی رنگ‌های قرمز نقض شود، سیستم مجبور به تغییر رنگ گره‌ها و عموهای آن‌ها می‌شود. پس از هر تغییر متغیر رنگ (تغییر از ۰ به ۱ یا بالعکس)، بلافاصله تابع آپدیت روی آن گره فراخوانی می‌شود تا این تغییر بلافاصله در اثر انگشت گره لحاظ شود.

ج) انتشار تغییرات به سمت ریشه (Bottom-up Maintenance):

اضافه شدن یک گره جدید به عنوان برگ، هش تمامی اجداد (Ancestors) آن گره تا ریشه درخت را نامعتبر می‌سازد. به همین دلیل، در پایان عملیات درج، حلقه‌ای تعبیه شده که از گره جدید آغاز شده و در زمان $O(\log N)$، مقدار هش و سایز تمامی گره‌ها را تا رسیدن به ریشه، پله‌پله به‌روزرسانی می‌کند.

۴. معماری گره نگهبان (Sentinel Node / TNULL)

در پیاده‌سازی‌های استاندارد، برگ‌های خالی در درخت‌های دودویی با اشاره‌گر پوچ (None یا Null) نشان داده می‌شوند. با این حال، در مفاهیم درخت قرمز-سیاه، برگ‌های خالی به عنوان گره‌های سیاه در نظر گرفته می‌شوند و در مسئله ما، باید در محاسبات فرمول هش نیز شرکت داده شوند.

برای جلوگیری از کدنویسی‌های پیچیده و بررسی‌های مکرر خالی بودن گره‌ها، از یک گره مصنوعی و یکتا به نام $TNULL$استفاده شده است. این گره دارای رنگ سیاه $(C = 0)$، سایز صفر $(S = 0)$ و هش صفر $(H = 0)$ است. تمامی اشاره‌گرهای چپ و راستِ خالی، به جای اتصال به None، به این گره متصل می‌شوند. این الگوی طراحی، کد را در برابر خطاهای زمان اجرا مقاوم کرده و یکپارچگی محاسبات ریاضی را تضمین می‌کند.

۵. مکانیزم پاسخگویی آنی به کوئری‌ها (Query Handling)

با توجه به اینکه هش تمامی گره‌ها در طول زمان به صورت بلادرنگ محاسبه و ذخیره شده‌اند، عملیات بررسی ایزومورفیسم (CLONE_CHECK) با بالاترین سرعت ممکن انجام می‌شود.

ما در پس‌زمینه از یک جدول درهم‌سازی (Dictionary یا Hash Map) استفاده می‌کنیم که مقادیر اعدادی ورودی را مستقیماً به آدرس حافظه آن گره‌ها در درخت مرتبط می‌کند. هنگام درخواست کاربر برای مقایسه گره‌های U و V:

۱. ابتدا آدرس دو گره از دیکشنری در زمان $O(1)$ استخراج می‌شود.

۲. سپس مقادیر هش آن‌ها ($H(U)$ و $H(V)$) با یکدیگر مقایسه می‌شوند.

۳. به عنوان لایه دوم امنیتی برای خنثی کردن احتمال بسیار ناچیز برخورد هش‌ها (Hash Collisions)، سایز دو زیردرخت نیز با هم مقایسه می‌گردند ($S_U == S_V$).

در صورت برقراری هر دو شرط، سیستم با قطعیت پاسخ YES می‌دهد. کل این فرآیند استنتاج، در زمان ثابت $O(1)$ صورت می‌پذیرد.

۶. تحلیل پیچیدگی (Complexity Analysis)

- **پیچیدگی زمانی:**

عملیات درج (INSERT) نیازمند یافتن جایگاه، انجام حداکثر دو چرخش، چندین تغییر رنگ و پیمایش رو به بالا برای آپدیت هش‌هاست که تمام این مسیرها متناسب با ارتفاع درخت هستند. بنابراین زمان درج اکیداً $O(\log N)$ است. عملیات پرس‌وجو (CLONE_CHECK) نیز شامل خواندن از دیکشنری و یک مقایسه منطقی ساده است که در زمان مطلق $O(1)$ اجرا می‌شود. در نهایت، پردازش Q کوئری با پیچیدگی $O(Q \log N)$ انجام خواهد شد که از نظر عملکردی بسیار بهینه است.

- **پیچیدگی مکانی (حافظه):**

به ازای هر داده ورودی، یک شیء کامل (گره) شامل اشاره‌گرها و صفات در حافظه تخصیص می‌یابد. دیکشنری مرجع نیز به همان اندازه فضا اشغال می‌کند. در نتیجه پیچیدگی مکانی برابر با خطی $O(N)$ می‌باشد که برای محدودیت‌های متداول مسابقات الگوریتمی کاملاً استاندارد و کارآمد است.

پاسخ سوال پنجم :

مسئلهٔ «کوره جادویی و سندان‌های سرخ‌وسیاه» نیازمند معماری چندلایه‌ای از ساختمان داده‌هاست تا بتواند به همهٔ کوئری‌ها در بدترین حالت (Worst-case) در زمان $O(\log N)$ یا $O(\log Q)$ پاسخ دهد.

استفاده از ساختارهای پیش‌فرض زبان‌ها (مثل dict در پایتون یا unordered_map در C++) به دلیل استفاده از جداول هش با تغییر اندازهٔ پویا (Dynamic Resizing) و خطر تصادم‌های عمدی (Hash Collision Attacks) می‌تواند پیچیدگی زمانی را به $O(Q)$تنزل دهد. بنابراین، این پاسخ‌نامه بر اساس پیاده‌سازی کاملاً دستی (Manual) تدوین شده است.

معماری سه‌لایهٔ سیستم

سیستم ما از سه بخش کاملاً مجزا اما متصل به هم تشکیل شده است:

۱. **لایهٔ فیزیکی (مدیریت سندان‌ها):** آرایه‌ای از پشته‌ها.

۲. **لایهٔ کوئری‌های بازه‌ای:** درخت سگمنت (Segment Tree).

۳. **لایهٔ ردیابی و جستجوی نام (هستهٔ سخت):** ترکیب هشینگ چندجمله‌ای و درخت سرخ‌سیاه (RBT).

---

۱. لایهٔ فیزیکی: آرایهٔ پشته‌ها (Array of Stacks)

از آنجا که شمشیرها روی هم قرار می‌گیرند و در عملیات MELT همواره بالاترین شمشیر برداشته می‌شود، ماهیت هر سندان دقیقاً منطبق بر یک پشته (LIFO) است.

- **ساختار:** یک آرایه دو بعدی پویا (یا آرایه‌ای از std::vector) به طول $N = 10^5$.
- **عملیات:**
  - در FORGE، تاپل $(power، name)$در زمان $O(1)$به انتهای پشتهٔ مربوط به سندان posاضافه (Push) می‌شود.
  - در MELT، در زمان $O(1)$آخرین عنصر پشته حذف (Pop) می‌شود.

---

۲. لایهٔ کوئری: درخت سگمنت (Segment Tree)

برای پاسخ به کوئری‌های MAX_TOP L R به صورت آنلاین، ما فقط به مقدار قدرتِ **سرِ پشتهٔ هر سندان** نیاز داریم، نه تمام شمشیرهای درون آن.

- **ساختار:** یک آرایه به طول $4 \times N$ که هر گره در آن، بیشینهٔ (Maximum) بازهٔ متناظر خود را نگه می‌دارد.
- **تابع Update:** با پیچیدگی $O(\log N)$.
  - وقتی FORGE رخ می‌دهد، برگ متناظر با سندان pos به مقدار $power$ شمشیر جدید به‌روزرسانی می‌شود.
  - وقتی MELT رخ می‌دهد، مقدار برگ باید به قدرت شمشیرِ زیرین تغییر کند. اگر سندان کاملاً خالی شد، مقدار برگ صفر (0) می‌شود. این مقدار به سمت ریشه (Root) به بالا حرکت کرده و مقدار max پدرها را اصلاح می‌کند.
- **تابع Query:** با پیچیدگی $O(\log N)$.
  - بازهٔ $[L, R]$ به مجموعه‌ای از زیربازه‌های استاندارد درخت تجزیه شده و ماکزیمم آن‌ها برگردانده می‌شود.

---

۳. لایهٔ جستجو: درخت سرخ-سیاه (Red-Black Tree) و هشینگ دستی

چالش اصلی پاسخ‌گویی به دستور INSPECT name است. ما باید یک رشته را جستجو کنیم.

برای تضمین پیچیدگی زمانی قطعیِ $O(\log K)$ (که K تعداد شمشیرهای موجود است)، از یک BST بالانس‌شده (درخت سرخ‌سیاه) استفاده می‌کنیم. اما مقایسهٔ رشته‌ها در هر گره، سربار محاسباتی دارد. بنابراین رشته را هش می‌کنیم:

الف) هشینگ چندجمله‌ای (Polynomial Rolling Hash)

ما نام شمشیر (رشته S) را به یک کلید عددی (Key) تبدیل می‌کنیم. فرمول استفاده شده:

$$H(S) = \left(\sum_{i=0}^{|S|-1} (S[i] - 'a' + 1) \times P^i \right) (\mathrm{mod} M)$$

- مقادیر ثابت: $P = 31$ و $M = 10^9 + 9$.

- این تابع در زمان $O(|S|)$(که با توجه به محدودیت ۲۰ کاراکتری، معادل $O(1)$است) یک کلید یکتا تولید می‌کند.

ب) پیاده‌سازی دستی درخت سرخ-سیاه (Manual RBT)

کلیدهای تولید شده به همراه نام رشته‌ایِ اصلی، قدرت و شماره سندان، در یک گره RBT ذخیره می‌شوند. ذخیرهٔ نام اصلی برای این است که در صورت بروز تصادم هش (Collision)، بتوانیم چک کنیم آیا رشته‌ها دقیقاً یکسان هستند یا خیر.

- **عملیات Insert و Insert-Fixup:**گره جدید به عنوان یک گره قرمز (RED) اضافه می‌شود. اگر پدر آن نیز قرمز باشد، نقض قانون RBT رخ داده است. در اینجا با بررسی رنگ "عمو" (Uncle) گره، یکی از سه حالتِ تغییر رنگ (Color Flip)، چرخش به چپ (Left Rotate) یا چرخش به راست (Right Rotate) در زمان $O(\log K)$انجام می‌شود تا درخت دوباره متوازن گردد.
- **عملیات Delete و Delete-Fixup:**هنگام عملیات MELT، باید شمشیر حذف شده را در RBT پیدا کرده و پاک کنیم. اگر گره حذف شده سیاه (BLACK) باشد، ارتفاع سیاه درخت (Black-Height) در یک مسیر به هم می‌ریزد. با اجرای delete_fixup و بررسی گره برادر (Sibling) و فرزندان آن، با حداکثر ۳ چرخش و تعدادی تغییر رنگ، تعادل درخت در زمان $O(\log K)$بازمی‌گردد.

تحلیل روال اجرای دستورات (Execution Flow)

۱. **دستور FORGE name power pos:**

- ابتدا $Hash(name)$محاسبه می‌شود ($O(1)$).
- گره جدید در RBT با موفقیت درج و بالانس می‌شود ($O(\log Q)$).
- شمشیر وارد پشتهٔ سندان posمی‌شود ($O(1)$).
- برگ posدر درخت سگمنت به مقدار $power$آپدیت می‌شود ($O(\log N)$).

۲. **دستور MELT pos:**

- در زمان $O(1)$سر پشته چک می‌شود. نام شمشیر ذوب‌شده به دست می‌آید.
- کلید $Hash(name)$محاسبه شده و این گره از RBT به صورت دستی Delete و درخت Fixup می‌شود ($O(\log\ Q)$).
- شمشیر از پشته Pop می‌شود ($O(1)$).
- درخت سگمنت برای سندان posبا مقدار سر پشتهٔ جدید آپدیت می‌شود ($O(\log\ N)$).

۳. **دستور INSPECT name:**

- محاسبهٔ هش $O(1)$و جستجوی باینری روی کلیدها درون RBT انجام می‌شود ($O(\log\ Q)$). اگر گره پیدا شد، مقادیر $power$و posچاپ می‌شود.

۴. **دستور MAX_TOP L R:**

- به سادگی یک تابع بازگشتی روی درخت سگمنت اجرا شده و خروجی در زمان $O(\log\ N)$تولید می‌شود.

پیچیدگی نهایی

- **زمان بدترین حالت (Worst-case Time Complexity):** به ازای هر درخواست حداکثر $O(\log\ N + \log\ Q)$عملیات پایه‌ای انجام می‌شود. برای کل Qدرخواست، زمان اجرا $O(Q\log\ (\max\ (N, Q)))$است که کاملاً مطلوب است.
- **فضای مصرفی (Space Complexity):** درخت سگمنت به اندازهٔ $O(N)$، پشته‌ها در مجموع $O(Q)$و گره‌های RBT در بدترین حالت $O(Q)$حافظه می‌گیرند. فضای کل برابر است با $O(N + Q)$.

پاسخ سوال ششم :

حل این مسئله نیازمند طراحی یک سیستم ترکیبی (Hybrid System) است که در آن چهار ساختمان دادهٔ پیشرفته به صورت همزمان و هماهنگ کار می‌کنند. هیچ‌یک از این ساختارها به تنهایی نمی‌توانند

تمام نیازهای مسئله را برآورده کنند، بنابراین ما باید آن‌ها را به یکدیگر متصل کنیم. در ادامه، ریزبه‌ریز و با جزئیات کامل بررسی می‌کنیم که هر ساختار چرا استفاده شده و چگونه پیاده‌سازی و متصل می‌شود.

---

۱. معماری و اجزای سیستم (کالبدشکافی ساختمان‌های داده)

برای حل این مسئله در زمان بهینه O(log N) برای هر عملیات، ما به ۴ قلب تپنده نیاز داریم:

الف) درخت سرخ-سیاه (Red-Black Tree) - هستهٔ مرکزی مدیریت شناسه‌ها

- **دلیل استفاده:** شناسه‌ها (id) می‌توانند تا 9^10 بزرگ باشند و ما نمی‌توانیم از یک آرایه ساده برای ذخیره آن‌ها استفاده کنیم. از طرفی برای عملیات REMOVE نیاز داریم در زمان O(log N) دقیقاً بدانیم عنصر با فلان شناسه چه ارزشی دارد، در کدام گروه است و مکانش در لیست پیوندی کجاست.
- **نحوه عملکرد:** هر گره (Node) در این درخت نه تنها کلید (id) را ذخیره می‌کند، بلکه مقادیر val، group و **یک اشاره‌گر (Pointer) به گره متناظرش در لیست پیوندی** را نیز نگه می‌دارد. این درخت با قوانین تغییر رنگ و چرخش‌ها (Rotations) همواره متوازن می‌ماند و از بدترین حالت O(N) جلوگیری می‌کند.

ب) درخت سگمنت پویا (Dynamic Segment Tree) - نگهبان بازه‌ها

- **دلیل استفاده:** برای پاسخ به کوئری MAX_VAL L R، باید بیشترین ارزش را در یک بازه از شناسه‌ها پیدا کنیم. چون دامنه شناسه‌ها 1 9^10 dots است، یک درخت سگمنت معمولی باعث خطای کمبود حافظه (Memory Limit) می‌شود.
- **نحوه عملکرد:** در سگمنت تری پویا، ما آرایه‌ای به طول ۴ میلیارد نمی‌سازیم! بلکه فقط ریشه را داریم و هر زمان که به یک بازهٔ جدید نیاز شد، فرزند چپ و راست آن را در همان لحظه (On-the-fly) می‌سازیم (Dynamic Node Allocation). این ساختار برای هر گره، بیشترین val در آن بازه را ذخیره می‌کند.

ج) درخت فنویک (Fenwick Tree / BIT) - حسابدار گروه‌ها

- **دلیل استفاده:** برای کوئری GROUP_SUM G باید مجموع ارزش‌های فعال در گروه‌های 1 تا G را محاسبه کنیم.
- **نحوه عملکرد:** چون تعداد گروه‌ها معقول است (حداکثر 5^10)، یک درخت فنویک استاندارد به طول 5^10 + 1 می‌سازیم. این درخت اجازه می‌دهد در زمان O(log M) (که M تعداد گروه‌هاست) مقدار یک گروه را آپدیت کنیم و مجموع پیشوندی (Prefix Sum) بگیریم.

د) لیست پیوندی دوطرفه (Doubly Linked List) - حافظهٔ زمانی گروه‌ها

- **دلیل استفاده:** عملیات RECENT_IN_GROUP آخرین عنصری را می‌خواهد که وارد شده و هنوز حذف نشده است. صف‌ها یا پشته‌های معمولی با عملیات حذف از وسط (به خاطر دستور REMOVE) مشکل دارند و زمان O(N) می‌گیرند.
- **نحوه عملکرد:** برای هر گروه، دو اشاره‌گر به نام‌های head (ابتدا) و tail (انتها) داریم. وقتی عنصری اضافه می‌شود، به عنوان یک گره جدید به tail آن گروه متصل می‌شود. چون لیست دوطرفه است، اگر عنصری بخواهد حذف شود (حتی از وسط لیست)، با داشتن اشاره‌گر آن (که از درخت سرخ-سیاه می‌گیریم) می‌توانیم در زمان O(1) پیوندهای قبل و بعد آن را به هم وصل کرده و آن را حذف کنیم.

---

۲. تشریح جزءبه‌جزء عملیات‌ها (چگونه چرخ‌دنده‌ها با هم کار می‌کنند؟)

عملیات اول: ADD id val group string

وقتی این دستور صادر می‌شود، سیستم باید عنصر را در هر ۴ ساختار ثبت کند:

1. **لیست پیوندی:** یک گره جدید از نوع ListNode با اطلاعات id و val ساخته می‌شود. سپس به انتهای (Tail) لیست مربوط به شماره group متصل می‌شود.
2. **درخت سرخ-سیاه:** گره جدیدی با کلید id ساخته می‌شود که علاوه بر val و group، ارجاعی به گرهِ ساخته شده در مرحلهٔ قبل (در لیست پیوندی) دارد. این گره با حفظ قوانین سرخ-سیاه درج می‌شود (زمان O(log N)).

3. **درخت سگمنت پویا:** متد آپدیت صدا زده می‌شود. از ریشه تا برگِ مربوط به شاخص id پایین می‌رویم و در صورت نبودنِ مسیر، گره‌ها را می‌سازیم. در برگ، مقدار بیشینه را برابر val قرار می‌دهیم و در مسیر برگشت به بالا، مقادیر max_val تمام گره‌های پدر را به‌روزرسانی می‌کنیم (زمان O(log N)). در این مرحله، رشتهٔ طلسم نیز هش (Hash) شده و در گره ذخیره می‌شود.
4. **درخت فنویک:** متد add(group, val) فراخوانی می‌شود تا مقدار val به ایندکس group اضافه شود و تمام بازه‌های درگیر در درخت فنویک آپدیت شوند (زمان O(log M)).

عملیات دوم: REMOVE id

این حساس‌ترین بخش سیستم است، زیرا حذف باید کاملاً تمیز انجام شود:

1. **یافتن اطلاعات:** ابتدا در درخت سرخ-سیاه به دنبال id می‌گردیم. اگر وجود داشت، val، group و اشاره‌گرِ لیست پیوندیِ آن را استخراج می‌کنیم.
2. **درخت فنویک:** با استفاده از اطلاعات بالا، متد add(group, -val) را صدا می‌زنیم تا دقیقاً همان مقدار از مجموع گروه کم شود.
3. **درخت سگمنت پویا:** متد آپدیت را برای شاخص id با یک فلگ (پرچم) is_delete=True صدا می‌زنیم. این کار باعث می‌شود مقدار آن در برگ برابر با منفی بی‌نهایت (-infty) شود و با برگشت به بالا، ماکزیمم‌های بازه‌ها اصلاح شوند.
4. **لیست پیوندی:** با استفاده از اشاره‌گری که از درخت سرخ-سیاه برداشتیم، مستقیماً به سراغ گرهِ مربوطه در لیست پیوندی می‌رویم. اگر گره در وسط است، prev و next آن را به هم وصل می‌کنیم. اگر tail است، tail گروه را به گره قبلی منتقل می‌کنیم. این کار در زمان O(1) انجام می‌شود!
5. **درخت سرخ-سیاه:** در نهایت، گره را از درخت سرخ-سیاه حذف کرده و قوانین تعادل و تغییر رنگ (Fixup) را اعمال می‌کنیم تا درخت به هم نریزد.

عملیات سوم: MAX_VAL L R

این عملیات فقط با **درخت سگمنت پویا** سر و کار دارد:

سیستم از ریشهٔ درخت سگمنت شروع می‌کند. اگر بازهٔ گرهِ فعلی کاملاً درون [L, R] باشد، مقدار max_val آن برگردانده می‌شود. اگر خارج از بازه باشد، منفی بی‌نهایت برمی‌گردد. اگر همپوشانی جزئی داشته باشد، سیستم به فرزندان چپ و راست می‌رود و بیشترین مقدار بین آن دو را برمی‌گرداند. این کار در زمان $O(\log \text{MAX\_ID})$ انجام می‌شود.

عملیات چهارم: GROUP_SUM G

این عملیات منحصراً به عهدهٔ **درخت فنویک** است:

سیستم کوئری استاندارد فنویک برای شاخص G را اجرا می‌کند. از G شروع کرده و با حذف کم‌ارزش‌ترین بیت (Least Significant Bit)، مقادیر ذخیره شده را جمع می‌کند تا به صفر برسد. زمان اجرا: $O(\log \text{MAX\_GROUP})$.

عملیات پنجم: RECENT_IN_GROUP G

این کوئری زیباییِ ترکیب درخت سرخ-سیاه و **لیست پیوندی** را نشان می‌دهد:

سیستم مستقیماً به آرایه/دیکشنری tails نگاه می‌کند و اشاره‌گر tail مربوط به گروه G را بررسی می‌کند.

- اگر tail وجود داشته باشد (یعنی لیست خالی نیست)، مقدار val ذخیره شده در آن را برمی‌گرداند.
- اگر tail خالی (None) باشد، پیغام EMPTY را چاپ می‌کند.
- چون در عملیات REMOVE همواره tail را به درستی آپدیت می‌کنیم، اینجا هیچ نیازی به جستجو نیست و جواب در زمان مطلق $O(1)$ به دست می‌آید.

خلاصه تحلیل پیچیدگی (Complexity Analysis)

- **زمان:**
- ADD: $O(\log N + \log M + \log \text{MAX\_ID})$
- REMOVE: $O(\log N + \log M + \log \text{MAX\_ID})$

- MAX_VAL: O(log (text{MAX_ID}))
- GROUP_SUM: O(log M)
- RECENT_IN_GROUP: O(1)
- **حافظه (Space):** به لطف استفاده از سگمنت تری پویا و نودهای مستقل، حافظهٔ مصرفی کاملاً متناسب با تعداد دستورات Q است و در بدترین حالت (O(Q log (text{MAX_ID})) حافظه مصرف می‌شود که کاملاً در محدودهٔ مجاز مسابقات (معمولاً ۲۵۶ مگابایت) قرار دارد.

پاسخ سوال هفتم :

۱ .تحلیل منطق الگوریتم

برای حل این سوال، ما نیاز به پیمایش درخت (DFS) داریم تا ۴ قانون اصلی را چک کنیم:

- **قانون ریشه :**قبل از شروع پیمایش، چک می‌کنیم که colorریشه Bباشد.
- **قانون قرمز-قرمز :**در هر گام از پیمایش بازگشتی، اگر گره فعلی Rباشد، چک می‌کنیم که فرزندانش حتماً B(یا نیل) باشند.
- **قانون ارتفاع سیاه :**این پیچیده‌ترین بخش است. تابع بازگشتی ما برای هر گره، "ارتفاع سیاه" (تعداد گره‌های سیاه تا برگ) را برمی‌گرداند.
  - اگر زیردرخت چپ و راست ارتفاع سیاه متفاوتی داشته باشند، تابع مقدار 1-(به معنی خطا) برمی‌گرداند.
  - اگر یکی از زیردرخت‌ها خطای 1-برگرداند، گره فعلی نیز 1-را به بالا می‌فرستد (انتشار خطا).

۲ .پیچیدگی زمانی و فضایی

- **پیچیدگی زمانی**$O(N)$ **:**
  - ما هر گره را دقیقاً یک بار ملاقات می‌کنیم.

- **پیچیدگی فضایی** $O(H)$ **:یا** $O(\log N)$
  - فضای مصرفی مربوط به پشته (Stack) بازگشتی است که متناسب با ارتفاع درخت می‌باشد.

پاسخ سوال هشتم :

۱ .تحلیل منطق الگوریتم

در این سوال، ما یک کلاس RedBlackTree کامل پیاده‌سازی می‌کنیم. الگوریتم درج شامل مراحل زیر است:

1. **درج باینری (BST Insert) : عدد را در محل مناسب درج کرده و رنگ آن را REDمی‌کنیم.**
2. **اصلاح (Fix-up): اگر والد گره جدید نیز REDباشد، قانون قرمز-قرمز نقض می‌شود. سه حالت پیش می‌آید:**
   - **حالت ۱ (عمو قرمز) :رنگ والد و عمو را سیاه، و رنگ پدربزرگ را قرمز می‌کنیم. نشانگر zرا روی پدربزرگ می‌گذاریم.**
   - **حالت ۲ (عمو سیاه - مثلثی) :یک چرخش انجام می‌دهیم تا به حالت خطی تبدیل شود.**
   - **حالت ۳ (عمو سیاه - خطی) :رنگ والد را سیاه و پدربزرگ را قرمز کرده، سپس چرخش روی پدربزرگ انجام می‌دهیم.**

۲ .پیچیدگی زمانی

- **درج :پیدا کردن محل مناسب** $O(\log N)$ **زمان می‌برد.**
- **اصلاح :(Fix-up) در بدترین حالت، تغییر رنگ تا ریشه بالا می‌رود (**$O(\log N)$**) اما حداکثر ۲ چرخش انجام می‌شود.(**$O(1)$**)**
- **عدد Qبرای کل عملیات: O(Q log Q)**