

Интервално дърво (Segment Tree)

Весела Николова

1. Увод

Интервалното дърво е много ефективно за решаване на проблеми като RSQ/RMQ (Range Sum Query/Range Minimum Query), където се налага обработване на заявки за дадени интервали. Ако разглеждаме сумите на такива сегменти в масив, бихме се справили, използвайки динамично програмиране или по-конкретно – парциални суми. Но ако търсим минимален/максимален елемент в интервали или имаме заявки за промяна стойността на елементи, предпочитан подход е решението с помощта на интервално дърво.

2. Теоретични бележки

Интервалното дърво представлява дървовидна структура от данни, която съдържа информация за интервали в даден масив. Могат да се осъществяват следните операции:

- строене на дървото
- отговаряне на заявки за интервали
- промяна на стойността на единичен елемент
- промяна на стойността на всички елементи в даден интервал (lazy propagation)

Интервалното дърво е двоично кореново дърво, като коренът отговаря за целия масив, с който се работи. Ако даден връх няма наследници (листо), то той отговаря за един елемент от масива. Ако има – то те са точно два на брой. Разделим ли интервала на този връх на две половини, единият наследник ще отговаря за лявата половина от елементите, а другият – за дясната. Тъй като винаги намаляваме големините на интервалите двойно, докато не станат равни на 1, височината на дървото е $\log(N)$. Уточнихме, че един връх в това дърво има или 0, или 2 наследника, никога 1 – в такова двоично дърво общият брой на върховете е $2*N-1$ (N листа и $N-1$ вътрешни върхове).

За да можем да работим с интервално дърво трябва да сме дефинирали:

- 1) Какво пазим във всеки връх от дървото?
- 2) Как сливаме левия и десния интервал в едно цяло?

3. Реализация

3.1. Избор на подходящи инструменти

В компютърната памет не представяме интервалното дърво като дърво, то е фиктивно, служи само за онагледяване начина на работа на сегментното дърво. Вместо това използваме линейна структура. За масив с N елемента използваме такъв с $4*N$ за изобразяване на самото дърво.

Нивата в дървото са $\log(N)$ и започваме от един връх в началото, а броят за всяко следващо ниво се удвоява спрямо предходното. Следователно върховете са максимално:

$$1 + 2 + 4 + \dots + 2^{\log(N)} = 2^{\log(N)+1} - 1 < 4*N$$

Тоест $4*N$ елемента са ни достатъчни за съхраняване информацията за цялото дърво.

3.2. Същинска реализация

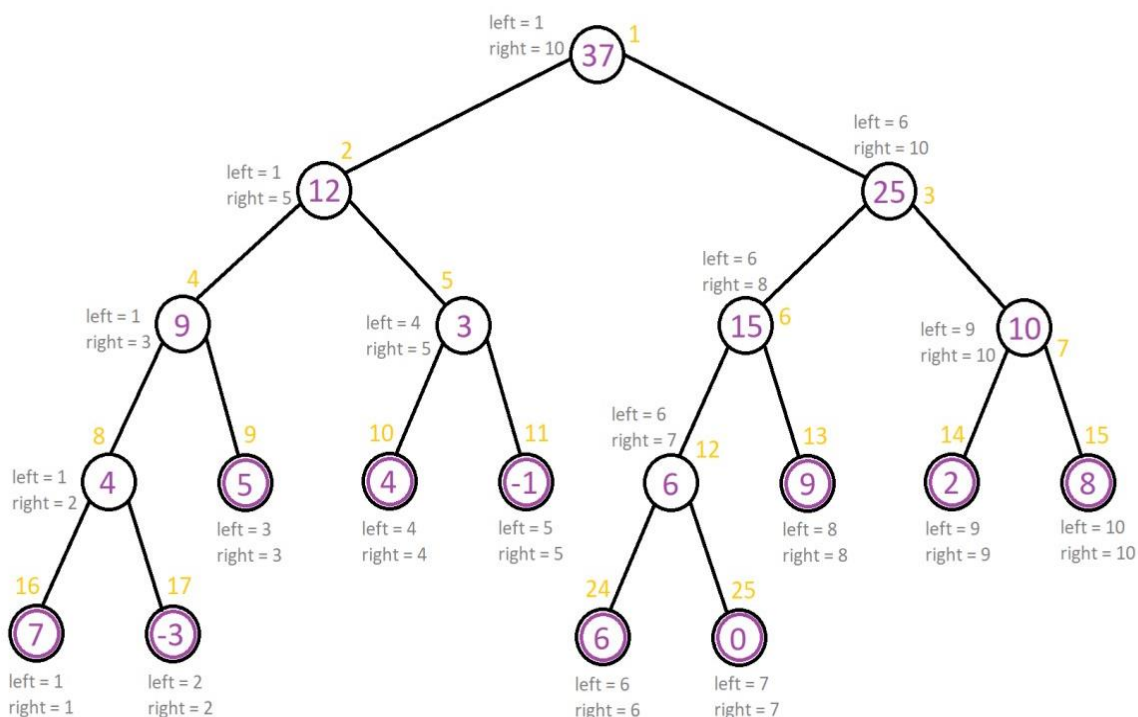
Ще работим с примера от фигурата:

arr[]

7	-3	5	4	-1	6	0	9	2	8
1	2	3	4	5	6	7	8	9	10

tree[]

37	12	25	9	3	15	10	4	5	4	-1	6	9	2	8	7	-3	X	X	X	X	X	X	6	0	X	X	X	X	X	...
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31



За реализация на операциите ще използваме рекурсия, защото е по-интуитивно, но могат да бъдат написани и итеративно.

*Ако текущият връх има индекс i , то лявото му дете е с индекс $2*i$, а дясното – $2*i+1$. Тъй като на първо ниво върхът е с индекс 1, на второ ниво първият връх е с индекс 2, на трето – с 4 и т.н. Тоест ако сме на първия връх от текущото ниво и то има индекс i , то лявото му дете ще е с индекс $2*i$, а дясното с едно повече. Следващият връх ще бъде детето на връх $i+1$ и ще е с индекс $2*(i+1)+1 = 2*i+2 = 2*(i+1)$.

3.2.1. Построяване на дървото

- Ако сме в листо, присвояваме стойност от оригиналния масив. (Например `tree[17] = arr[2]`)
- В противен случай разделяме интервала на две половини и пускаме функцията да обработи новите интервали, след което сливаме резултатите от наследниците на текущия връх. (Например интервалът от 6 до 8 го разделяме на два, съответно от 6 до 7 и от 8 до 8, като след това събираме получените стойности:
`tree[6] = tree[12] + tree[13]`) Примерна

реализация:

```
void make_tree(int index, int left, int right)
{
    if (left == right)
    {
        tree[index] = arr[left];
        return;
    }
    int middle = (left + right) / 2;
    make_tree(2*index, left, middle);
    make_tree(2*index+1, middle+1, right);
    tree[index] = tree[2*index] + tree[2*index+1];
}
```

Имайки предвид, че при строенето на дървото посещаваме всеки връх по веднъж, а те са 2^N-1 на брой, това означава, че сложността по време е $O(2^N-1)$, игнорираме константите и получаваме сложност $O(N)$.

3.2.2. Отговаряне на заявки за дадени интервали

- Проверяваме дали текущият интервал няма никакво пресичане с този, който ни интересува, ако да – няма нужда да работим с този сегмент и спираме действието на функцията. (Например, ако търсим отговор за сумата на елементите от 4 до 9 от фигурата по-горе, но сме стигнали до интервала от 1 до 3, то няма нужда да го разглеждаме)
- Ако текущият интервал изцяло попада в този, който ни интересува, връщаме отговора, съдържащ се във върха, на

който сме от дървото, и отново прекратяваме действието на функцията, защото няма смисъл да разделяме интервала на по-малки сегменти. (Например, ако търсим отговор за сумата на елементите от 4 до 9 и сме стигнали до интервала от 6 до 8 – директно връщаме отговор за него, без да слизаме до 6-7 и 8-8)

- В случай че в момента интервалът ни не е изцяло вън или изцяло вътре в сегмента, който ни интересува, го разделяме на две и връщаме отговор за двете половини.

Примерна реализация:

```
int query_sum(int index, int left, int right, int&query_left, int&query_right)
{
    if (right < query_left || left > query_right)
        return 0;
    if (left >= query_left && right <= query_right)
        return tree[index];
    int middle = (left + right) / 2;
    return query_sum(2*index, left, middle, query_left, query_right) +
           query_sum(2*index+1, middle+1, right, query_left, query_right);
}
```

За всяко ниво посещаваме най-много 4 върха. Щом сме стигнали до тези 4 върха, значи от горното ниво е имало два интервала, които се пресичат частично със заявката, която ни интересува, но не изцяло. Няма как да са повече от два такива интервала, защото ако има трети – той ще е вляво от тези (тоест извън заявката), вдясно (отново извън заявката) или между тях (което ще значи, че изцяло се застъпва с търсения сегмент). Нивата са $\log(N)$ на брой, следователно сложността по време е $O(4 \cdot \log(N))$, като игнорираме константите получаваме $O(\log(N))$.

3.2.3. Промяна на стойността на единичен елемент

- Ако сме в листо, то отговаря за стойността, която искаме да променим, и директно я присвояваме.
- В противен случай продължаваме да обработваме лявата или дясната половина, в зависимост от това коя е позицията, на която ще променяме стойност. Накрая отново сливаме резултатите от наследниците на текущия връх, за да се обновят интервалите, в които е участвал елементът с променена стойност. (Например, ако сме сменили седмия елемент, трябва да обновим отговорите за интервалите от 6 до 7, от 6 до 8, от 6 до 10 и от 1 до 10)

Примерна реализация:

```
void point_update(int index, int left, int right, int&position, int&value)
{
    if (left == right)
    {
        tree[index] = value;
        return;
    }
    int middle = (left + right) / 2;
    if (position <= middle) point_update(2*index, left, middle, position, value);
    else point_update(2*index+1, middle+1, right, position, value);
    tree[index] = tree[2*index] + tree[2*index+1];
}
```

При промяна стойността на елемент от върха на дървото стигаме до съответното листо, което се изпълнява за сложност равна на височината на дървото – $O(\log(N))$.

3.2.4. Промяна на стойностите на всички елементи в даден интервал (lazy propagation)

С цел да променяме бързо стойностите в даден интервал, вместо поединично да стигаме до съответните листа и да работим с тях, си въвеждаме помощен масив `lazy[]`, където ще пазим каква стойност трябва да се добави към съответния връх от дървото (към интервала). Обновяването на стойностите се извършва по подобен начин на отговарянето на заявките. Разглеждаме само интервали, които се пресичат със сегментът, който представлява интерес. Ако изцяло се препокрива със заявката, добавяме `lazy[i]` към текущия връх, ако не – разделяме интервалът на две и работим отделно с двете половини. След което обновяваме стойностите в дървото.

Въвеждаме функцията `push_lazy`, която ще променя стойностите в даден връх на дървото и ще предава `lazy`-то на децата му (ако има такива). Тази функция извикваме в началото на тази за промяна стойностите на интервала, за да сме сигурни, че не сме стигнали даден връх, а над него има непренесено `lazy`, тоест работим с некоректни стойности. Също така във всеки момент, в който добавяме `lazy` в даден връх, го предаваме нататък поради същите съображения. И при функцията за отговаряне на заявка ще предаваме `lazy`-то надолу по нивата, в които се движим.

Примерна реализация:

```
void push_lazy(int&index, int&left, int&right)
{
    if (lazy[index])
    {
        tree[index] += (right-left+1) * lazy[index];
        if (left != right)
        {
            lazy[2*index] += lazy[index];
            lazy[2*index+1] += lazy[index];
        }
        lazy[index] = 0;
    }
}

void update_range(int index, int left, int right, int&query_left, int&query_right, int&add)
{
    push_lazy(index, left, right);
    if (right < query_left || left > query_right)
        return;
    if (left >= query_left && right <= query_right)
    {
        lazy[index] += add;
        push_lazy(index, left, right);
        return;
    }
    int middle = (left + right) / 2;
    update_range(2*index, left, middle, query_left, query_right, add);
    update_range(2*index+1, middle+1, right, query_left, query_right, add);
    tree[index] = tree[2*index] + tree[2*index+1];
}
```

Промяната във функцията за отговор на заявка е отразена тук:

```
int query_sum(int index, int left, int right, int&query_left, int&query_right)
{
    push_lazy(index, left, right);
    if (right < query_left || left > query_right)
        return 0;
    if (left >= query_left && right <= query_right)
        return tree[index];
    int middle = (left + right) / 2;
    return query_sum(2*index, left, middle, query_left, query_right) +
           query_sum(2*index+1, middle+1, right, query_left, query_right);
}
```

Сложността на операциите не се променя, защото функцията push_lazy работи за константно време.

4. Приложение

Приложени са задачи за упражнение:

<https://cses.fi/problemset/>

Static Range Minimum Queries

Dynamic Range Minimum Queries

Range Xor Queries

Range Update Queries

Prefix Sum Queries

<https://informatics.msk.ru/>

Задача №3312

Задача №3318

<https://arena.olimpiici.com/>

Контролни за момичешки и младши отбор, 2018, Контролно 4:
numbers

Пролетен турнир, В, 2018: *tv*

Национална олимпиада, Областен кръг, А, 2019: *beatle*

За въпроси от всякакво естество: veselanikolova333@gmail.com