Projektni zadatak iz predmeta Algoritmi i Strukture Podataka

Segment Tree (Delete)

Ivana Đorđević, IT59/2019
Inženjerstvo informacionih
sistema
Godina 2024.

Uvodni opis i motivacija za izradu projekta

Struktura podataka Segment Tree već decenijama se koristi u računarstvu, posebno za rešavanje problema sa intervalnim upitima i efikasnim ažuriranjima podataka. Iako nije poznat jedan konkretan autor koji je osmislio ovu strukturu, ona se smatra standardnim konceptom u literaturi o algoritmima i strukturama podataka.

Segment tree je binarno stablo koje se koristi za čuvanje intervala ili segmenata. Svaki čvor u segmentnom stablu predstavlja jedan **interval**.

Prednosti Segment Tree-a u poređenju sa drugim strukturama podataka:

1. **Efikasnos**t:
    - Upiti: Kompleksnost je O(log n), dok je kod jednostavnog niza O(n).
    - Ažuriranja: Kompleksnost je takođe O(log n), dok kod jednostavnih metoda može biti O(n).

2. **Efikasnost u memoriji**:
    - Za razliku od struktura kao što je Fenwick Tree (BIN – Binary Indexed Tree), Segment Tree omogućava rad sa nekomutativnim funkcijama, čineći ga univerzalnijim.

3. **Fleksibilnost**:
    - Podržava više tipova operacija kao što su zbir, minimum, maksimum, i prilagođene funkcije (npr. bitwise OR, XOR).

4. **Podrška za dinamičke promene na opsezima**:
    - Uz tehniku poput lenjog propagiranja (*lazy propagation*), Segment Tree može efikasno obraditi velike promene na intervalima.

5. **Generičnost**:
    - Može se koristiti za različite tipove podataka (npr. cele brojeve, realne brojeve ili čak objekte sa prilagođenim operacijama).

Struktura podataka **Segment Tree** predstavlja balans između fleksibilnosti i efikasnosti, što je čini idealnim izborom za rešavanje problema koji uključuju **intervalne upite** i **dinamička ažuriranja**. Zbog toga se Segment Tree široko koristi u oblastima kao što su **konkurentno programiranje**, **baze podataka**, i drugi sistemi koji zahtevaju rad sa intervalima.

Potrebno je naglasiti da segmentno stablo **nema direktnu operaciju brisanja čvora**. Razlog tome je što segmentno stablo nije lista elemenata, već je izgrađeno na osnovu segmenta niza, a svaki čvor predstavlja neki opseg.

Detaljno objašnjenje strukture

Osnovne karakteristike:

1. Koreni čvor predstavlja ceo niz.
2. Listovi stabla predstavljaju pojedinačne elemente niza.
3. Unutrašnji čvorovi predstavljaju kombinaciju rezultata operacija svojih podstabala.

1.Kreiranje segmentnog stabla

1) Počinjemo sa celim nizom A[0:N-1].
2) Delimo niz na dva podintervala sve dok ne dođemo do pojedinačnih elemenata.
3) Vrednost unutrašnjih čvorova izračunava se kombinovanjem vrednosti njihovih levog i desnog podstabla, u zavisnosti od operacije.

```
Funkcija BUILD(node, start, end):
    Ako start == end:  // Ako je list
        tree[node] = A[start]
    Inače:
        mid = (start + end) / 2
        BUILD(2 * node + 1, start, mid)  // Kreiraj levo podstablo
        BUILD(2 * node + 2, mid + 1, end)  // Kreiraj desno podstablo
        tree[node] = KOMBINUJ(tree[2 * node + 1], tree[2 * node + 2])
```

2.Preuzimanje informacije iz intervala (Query)

Ako interval čvora u potpunosti odgovara traženom intervalu, vraćamo njegovu vrednost. Međutim, ako se interval čvora delimično preklapa sa traženim intervalom, rekurzivno ispitujemo levo i desno podstablo.
Rezultati iz levog i desnog podstabla se kombinuju u zavisnosti od operacije (npr. sabiranje, minimum).

```
Funkcija QUERY(node, start, end, L, R):
    Ako interval čvora [start, end] ne pripada [L, R]:
        Vratiti neutralnu vrednost
    Ako interval čvora [start, end] pripada [L, R]:
        Vratiti tree[node]
    Inače:
        mid = (start + end) / 2
        levo = QUERY(2 * node + 1, start, mid, L, R)
        desno = QUERY(2 * node + 2, mid + 1, end, L, R)
        Vratiti KOMBINUJ(levo, desno)
```

3.Ažuriranje vrednosti (Update)

Ako je indeks elementa koji ažuriramo u opsegu trenutnog čvora, menjamo njegovu vrednost. Promene propagiramo ka roditeljskim čvorovima kako bi se očuvala konzistentnost strukture.

```
Funkcija UPDATE(node, start, end, idx, value):
    Ako start == end:  // Ako je list
        tree[node] = value
    Inače:
        mid = (start + end) / 2
        Ako idx <= mid:
            UPDATE(2 * node + 1, start, mid, idx, value)
        Inače:
            UPDATE(2 * node + 2, mid + 1, end, idx, value)
        tree[node] = KOMBINUJ(tree[2 * node + 1], tree[2 * node + 2])
```

4.Lazy Propagation

Kod intervalnih ažuriranja, kada želimo da izmenimo veliki broj elemenata, možemo koristiti tehniku lenje propagacije. Ova tehnika omogućava odlaganje ažuriranja dok nije apsolutno neophodno. Prednosti ove tehnike jesu mogućnost brže operacije ažuriranja na velikim intervalima kao I to što održava kompleksnost po upitu ili ažuriranju.

```
Funkcija UPDATE_RANGE(node, start, end, L, R, value):
    Ako postoji lenji zahtev za trenutni čvor:
        Ažuriraj trenutni čvor
        Propagiraj zahtev na potomke

    Ako trenutni interval [start, end] ne pripada [L, R]:
        Povratak

    Ako trenutni interval [start, end] u potpunosti pripada [L, R]:
        Ažuriraj vrednost trenutnog čvora
        Ako nije list:
            Zabeleži lenje ažuriranje za potomke
        Povratak

    Inače:
        mid = (start + end) / 2
        UPDATE_RANGE(2 * node + 1, start, mid, L, R, value)
        UPDATE_RANGE(2 * node + 2, mid + 1, end, L, R, value)
        tree[node] = KOMBINUJ(tree[2 * node + 1], tree[2 * node + 2])
```

5.Brisanje vrednosti (Delete)

Kada implementiramo ovu operaciju, cilj nam je da uklonimo određeni element iz segmentnog stabla, ažuriramo njegovu vrednost na neutralan element (npr. 0 za zbir), i propagiramo promene kroz stablo. Ova operacija ima kompleksnost **O(log n)** jer se promene prenose kroz visinu stabla.

```
Funkcija DELETE(index, value):
    1. Ažuriraj list na poziciji 'index' sa vrednošću '0'.
    2. Rekurzivno se pomeraj ka gore kroz stablo.
       Za svaki roditeljski čvor:
       Ažuriraj vrednost kao zbir vrednosti svojih sinova.
    3. Stop kada dođeš do korena stabla.
```

Analiza kompleksnosti

Osnovne operacije Segment Tree-a

Segment Tree podržava tri osnovne operacije:

1. **Kreiranje stabla (Build)** – Jednokratna inicijalizacija na osnovu početnih podataka.
2. **Upit (Query)** – Dohvatanje informacije sa određenog intervala (npr. zbir, minimum, maksimum).
3. **Ažuriranje (Update)** – Promena vrednosti jednog elementa ili intervala.

Kompleksnost operacija

1. **Kreiranje stabla**:

   - Proces kreiranja stabla zahteva rekurzivno deljenje početnog niza na podintervale i spajanje rezultata.
   - Na svakom nivou stabla prolazi se kroz sve elemente, a visina stabla je **O(log n)**.
   - Kompleksnost ove operacije je **O(n)** jer se svaki element obrađuje samo jednom.

2. **Upit**:

   - Upit traži informaciju na određenom intervalu (npr. [L, R]).
   - Proces je efikasan jer se interval deli na manje delove i preklapajuće čvorove, pri čemu se samo relevantni čvorovi obrađuju.
   - Broj čvorova koji se obrađuju na svakom nivou stabla je ograničen, što vodi do kompleksnosti **O(log n)**.

3. **Ažuriranje**:

    - Ažuriranje jednog elementa propagira promene kroz sve roditeljske čvorove na putu ka korenu stabla.
    - Slično kao kod upita, kompleksnost je **O(log n)** jer je potrebno samo ažurirati čvorove na visini stabla.

4. **Lazy Propagation:**

    - Kada se ažurira interval umesto jednog elementa, koristi se tehnika "lenje propagacije" da se promene odlože do trenutka kada su neophodne.
    - Ovo dodatno smanjuje broj ažuriranja na **O(log n)** za pojedinačni interval, dok je ukupna efikasnost i dalje linearna za sva ažuriranja.

5. **Delete**:
    - Svaki list se nalazi na dnu stabla, a svako ažuriranje utiče samo na roditeljske čvorove duž jednog puta do korena.
    - Visina stabla određuje broj ažuriranih čvorova, što dovodi do logaritamske kompleksnosti.
    - Dodatni prostor nije potreban, jer se sve promene dešavaju **in-place** u postojećem stablu.

Analiza vremenske kompleksnosti po segmentima:

Operacija	Kompleksnost	Objašnjenje
Kreiranje stabla	O(n)	Svaki element početnog niza se obrađuje samo jednom tokom konstrukcije stabla.
Upit	O(log n)	Upit se deli na manje delove intervala, pri čemu se koristi visina stabla (logaritam broja elemenata).
Ažuriranje	O(log n)	Promene se propagiraju od lista do korena, takođe zavisno od visine stabla.
Lenja propagacija	O(log n)	Intervalne promene se odlažu i izvršavaju samo kada je potrebno.
Brisanje	O(log n)	Prolazimo samo kroz visinu stabla

Segmenti koji mogu uticati na kompleksnost:

1. **Velik broj elemenata** (n): Ukupna efikasnost zavisi od broja elemenata jer se veličina stabla povećava proporcionalno broju elemenata (oko 4 * n).

2. **Tip operacije**: Jednostavni upiti kao što su zbir ili minimum zahtevaju minimalno procesiranje. Ažuriranja intervala zahtevaju dodatnu logiku (lenja propagacija), ali i dalje imaju kompleksnost O(log n).

3. **Balansirano stablo**: Segment Tree je uvek balansirano stablo, što garantuje da visina ne prelazi log n.

4. **Dodatna memorija**: Memorijski trošak je O(4 * n) jer Segment Tree koristi dodatni niz za čuvanje čvorova.

Implementacija u C#

Nazivi svih metoda koje su korišćene u projektu:

1. **Konstruktor**

Kreira strukturu stabla i priprema dodatni niz za lazy propagation.

```
public SegmentTree(int[] arr, int neutralValue = 0)
{
    n = arr.Length;
    this.neutralValue = neutralValue;
    tree = new int[4 * n];
    lazy = new int[4 * n];
    Build(0, 0, n - 1, arr);
}
```

2. **Build Tree**

Rekurzivno gradi segmentno stablo od ulaznog niza.

```
private void Build(int node, int start, int end, int[] arr)
{
    if (start == end)
    {
        tree[node] = arr[start];
    }
    else
    {
        int mid = (start + end) / 2;
        Build(2 * node + 1, start, mid, arr);
        Build(2 * node + 2, mid + 1, end, arr);
        tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
    }
}
```

3. Lazy Propagation

```
private void Propagate(int node, int start, int end)
{
    if (lazy[node] != 0)
    {
        tree[node] += (end - start + 1) * lazy[node];
        if (start != end)
        {
            lazy[2 * node + 1] += lazy[node];
            lazy[2 * node + 2] += lazy[node];
        }
        lazy[node] = 0;
    }
}
```

4. Query for range sum

Rekurzivno pretražuje stablo kako bi se izračunao zbir u datom opsegu.

```
private int Query(int node, int start, int end, int l, int r)
{
    Propagate(node, start, end);

    if (r < start || l > end)
        return 0;

    if (l <= start && end <= r)
        return tree[node];

    int mid = (start + end) / 2;
    int left = Query(2 * node + 1, start, mid, l, r);
    int right = Query(2 * node + 2, mid + 1, end, l, r);
    return left + right;
}
```

5. Range update with lazy propagation

Ažurira sve elemente u datom opsegu dodavanjem određene vrednosti.

```
public void RangeUpdate(int l, int r, int val)
{
    RangeUpdate(0, 0, n - 1, l, r, val);
}
```

6. Delete

Simulira brisanje tako što postavlja vrednost na neutralni element (0 za sumu).

```
public void Delete(int index)
{
    if (index < 0 || index >= n)
        {
            throw new ArgumentOutOfRangeException(nameof(index), "Element je van opsega.");
        }

    Update(index, neutralValue);
}
```

7. Update

```
public void Update(int index, int newValue)
{
    Update(0, 0, n - 1, index, newValue);
}
```

Ažurira vrednost jednog elementa na određenom indeksu.

```
private void Update(int node, int start, int end, int index, int newValue)
{
    Propagate(node, start, end);

    if (start == end)
    {
        tree[node] = newValue;
        return;
    }

    int mid = (start + end) / 2;
    if (index <= mid)
        Update(2 * node + 1, start, mid, index, newValue);
    else
        Update(2 * node + 2, mid + 1, end, index, newValue);

    tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
}
```

Rekurzivno ažurira određeni element u stablu