Programare cu premeditare

Cătălin Frâncu

Cuprins

Partea I

Structuri de date pe vectori

Următoarele capitole tratează structuri de date care pot procesa anumite operații pe vectori în timp mai bun decît $\mathcal{O}(N)$. Ocazional aceste structuri se aplică și matricilor.

Subiectele de ONI / baraj ONI / lot din anii trecuți abundă în probleme rezolvabile cu astfel de structuri:

- 3dist (baraj ONI 2022)
- 6 de Pentagrame (lot 2024)
- Babel (baraj ONI 2025)
- Balama (baraj ONI 2024)
- Bisortare (ONI 2021)
- Circuit (lot 2025)
- Emacs (baraj ONI 2021)
- Erinaceida (lot 2022)
- Guguștiuc (baraj ONI 2022)
- Împiedicat (baraj ONI 2023)
- Lupușor (ONI 2022)
- Medwalk (lot 2025)
- Perm (baraj ONI 2024)
- Piezișă (baraj ONI 2022)
- Subiectul III (lot 2024)
- Șirbun (baraj ONI 2023)
- Trapez (lot 2025)

Pare o idee bună să le învățăm și să le stăpînim bine. 🙂 Concret, vom studia trei structuri:

1. arbori de intervale;
2. arbori indexați binar;
3. descompunere în radical.

Vom exemplifica structurile și vom face benchmarks pe două probleme didactice. Apoi vom vedea, prin probleme, cum putem extinde aceleași structuri pentru nevoi mai complicate.

Varianta 1 (actualizări punctuale, interogări pe interval): Se dă un vector de N elemente întregi și Q operații de două tipuri:

1. $\langle 1, x, val \rangle$: Adaugă val pe poziția x a vectorului.
2. $\langle 2, x, y \rangle$: Calculează suma pozițiilor de la x la y inclusiv.

Varianta 2 (actualizări pe interval, interogări pe interval): Similar, dar operația 1 este pe interval:

1. $\langle 1, x, y, val \rangle$: Adaugă val pe pozițiile de la x la y inclusiv.
2. $\langle 2, x, y \rangle$: Calculează suma pozițiilor de la x la y inclusiv.

Vom menționa ocazional și **Varianta 3 (actualizări pe interval, interogări punctuale)**:

1. $\langle 1, x, y, val \rangle$: Adaugă val pe pozițiile de la x la y inclusiv.
2. $\langle 2, x \rangle$: Returnează valoarea poziției x.

Toate implementările mele sînt disponibile pe GitHub.

Capitolul 1

Arbori de intervale

Arborii de intervale[1] (AINT) sînt o structură foarte puternică și flexibilă. Ușurința implementării depinde de natura operațiilor pe care dorim să le admitem.

1.1 Reprezentare

Ca multe alte structuri (heap-uri, AIB, păduri disjuncte), arborii de intervale se reprezintă pe un simplu vector. Ei sînt arbori doar la nivel logic, în sensul că fiecare poziție din vector are o altă poziție drept părinte.

Pentru început, să presupunem că vectorul dat are $n = 2^k$ elemente. Atunci vectorul necesar S are $2n$ elemente, în care cele n elemente date sînt stocate începînd cu poziția n. Apoi,

- Cele $n/2$ elemente anterioare stochează valori agregate (sume, minime, xor etc.) pentru perechi de valori din vectorul dat.
- Cele $n/4$ elemente anterioare stochează valori agregate pentru grupe de 4 valori din vectorul dat.
- ...
- Elementul $S[1]$ stochează valoarea agregată a întregului vector.
- Valoarea $S[0]$ rămîne nefolosită.

Iată un exemplu pentru $n = 16$. Datele de la intrare se regăsesc pe pozițiile 16-31.

[1] Există o inversiune între nomenclatura internațională și cea românească. Internațional, structura pe care o învățăm astăzi se numește segment tree, iar interval tree este o structură diferită, care stochează colecții de intervale. Cîțiva ani am înotat împotriva curentului și am fost (posibil) singurul român care se referea la această structură ca „arbori de segmente". În acest curs am adoptat și eu denumirea încetățenită.

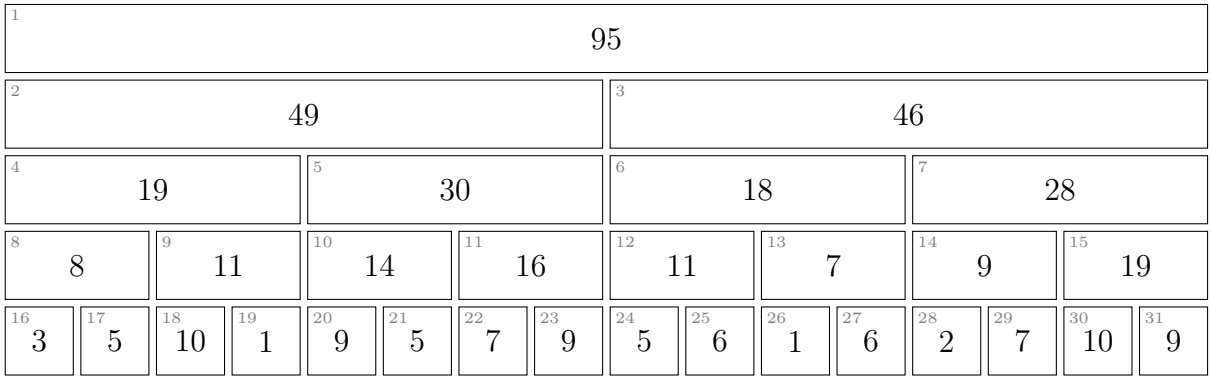

Figura 1.1: Un arbore de intervale cu 16 frunze și 15 noduri interne. Valorile din fiecare celulă reprezintă suma din frunzele subîntinse de acea celulă. Cu cifre mici este notat indicele fiecărei celule.

Facem cîteva observații preliminare:

- Fiii unui nod i sînt $2i$ și $2i + 1$.
- Părintele lui i este $\lfloor i/2 \rfloor$.
- Toți fiii stîngi au numere pare și toți fiii drepți au numere impare.

De exemplu, fiii lui 6 sînt 12 și 13, iar fiii acestora sînt respectiv 24-25 și 26-27. Aceasta corespunde cu intenția noastră ca 6 stocheze informații agregate (suma) despre nodurile 24-27.

După cum vom vedea în secțiunea următoare, arborii de intervale obțin timpi logaritmici pentru operații, deoarece numărul de niveluri este $\log n$.

1.1.1 Memoria necesară

În această formă, structura necesită $2n$ memorie pentru n elemente dacă n este putere a lui 2 sau foarte aproape. De exemplu, pentru $n = 1024$, sînt necesare 2048 de celule. Dar, dacă n depășește cu puțin o putere a lui 2, atunci el trebuie rotunjit în sus. Pentru $n = 1025$, baza arborelui necesită 2048 de celule, iar arborele în întregime necesită 4096 de celule. De aceea spunem că, în cel mai rău caz, arborele poate ajunge la $4n$ celule ocupate în cel mai rău caz.

În realitate, necesarul este doar de $3n$ cu puțină atenție la alocare. Pentru $n = 1025$, alocăm 2048 de celule pentru nivelurile superioare ale arborelui, dar putem aloca fix 1025 pentru bază (nu 2048). Totalul este circa $3n$.

Pentru a calcula următoare putere a lui 2, putem folosi bucla naivă:

```
int p = 1;
while (p < n) {
  p *= 2;
}
n = p;
```

Sau o buclă care folosește *bit hacks*:

```
while (n & (n - 1)) {
  n += n & -n;
}
```

Mai concis, putem folosi funcția `__builtin_clz(x)`, care ne spune cu cîte zerouri începe numărul `x`:

```
n = 1 << (32 - __builtin_clz(n - 1));
```

1.1.2 Reprezentări alternative

Există și reprezentări mai compacte, care ocupă exact $2n-1$ noduri, adică strictul necesar teoretic. Iată un exemplu pentru un vector cu 6 noduri.

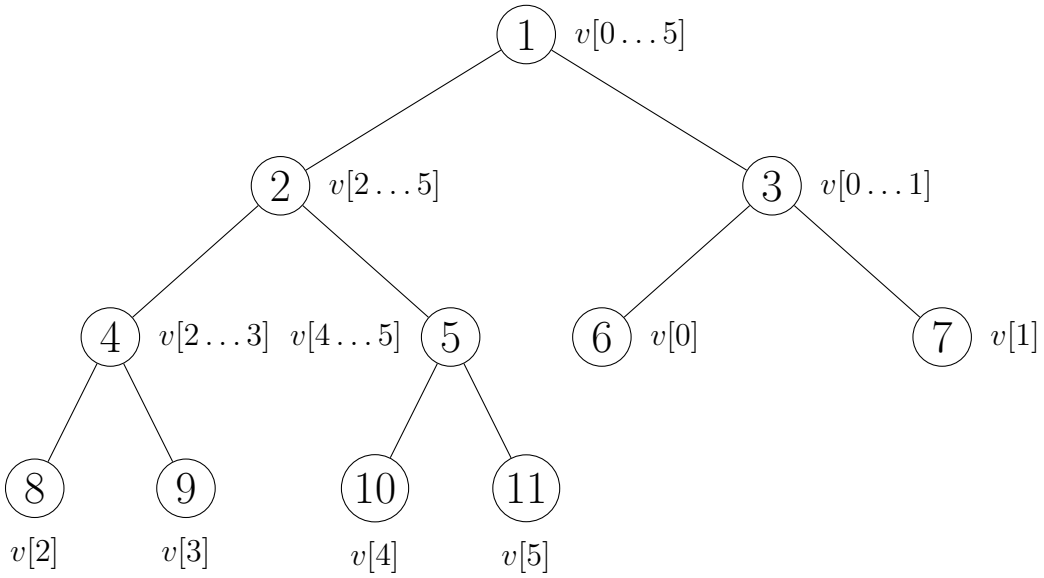

Figura 1.2: Reprezentarea arborilor de intervale cu exact $2n-1$ noduri.

Vedem că frunzele (adică vectorul dat, $v[0] \dots v[5]$) se află pe pozițiile consecutive 6-11. În schimb, această reprezentare pare mai greu de vizualizat și încalcă o abstracție importantă: frunzele nu mai sînt la același nivel. Structura se pretează la operațiile de modificare și interogare, dar nu sînt sigur că se pretează și la restul operațiilor pe care le discutăm în secțiunile următoare. De aceea prefer să folosesc și să predau structura rotunjită la 2^k noduri.

1.2 Operații elementare

1.2.1 Actualizarea punctuală

Nu uitați că poziția i din datele de intrare este stocată efectiv în $s[n+i]$. Apoi, cînd elementul aflat pe poziția i primește valoarea val, toate nodurile care acoperă poziția i trebuie recalculate:

```
void set(int pos, int val) {
  pos += n;
  s[pos] = val;
  for (pos /= 2; pos; pos /= 2) {
    s[pos] = s[2 * pos] + s[2 * pos + 1];
  }
}
```

Dacă nu ni se dă noua valoare absolută, ci variația δ față de valoarea anterioară, atunci codul este chiar mai simplu, căci toți strămoșii poziției se modifică tot cu δ:

```
void add(int pos, int delta) {
  for (pos += n; pos; pos /= 2) {
    s[pos] += delta;
  }
}
```

Apropo de *clean code*: Remarcați că am denumit funcțiile `set` și `add`, nu le-am denumit pe ambele `update`. Astfel am evidențiat diferența dintre ele.

1.2.2 Construcția în $\mathcal{O}(n \log n)$

O variantă de construcție este să invocăm funcția set de mai sus pentru fiecare valoare de la intrare. Complexitatea va fi $\mathcal{O}(n \log n)$.

1.2.3 Construcția în $\mathcal{O}(n)$

Putem reduce timpul de construcție dacă doar inserăm valorile frunzelor, fără a le propaga la strămoși. La final calculăm foarte simplu nodurile interne, în ordine descrescătoare.

```
void build() {
  for (int i = n - 1; i >= 1; i--) {
    s[i] = s[2 * i] + s[2 * i + 1];
  }
}
```

1.2.4 Calculul sumei pe interval

Să calculăm suma pe intervalul original [2, 12], care corespunde intervalului [18, 28] din reprezentarea internă. Ideea este să descompunem acest interval într-un număr logaritmic de segmente, mai exact [18,19], [20,23], [24,27] și [28,28]. Avantajul descompunerii este că avem deja calculate sumele acestor intervale, respectiv în nodurile 9, 5, 6 și 28.

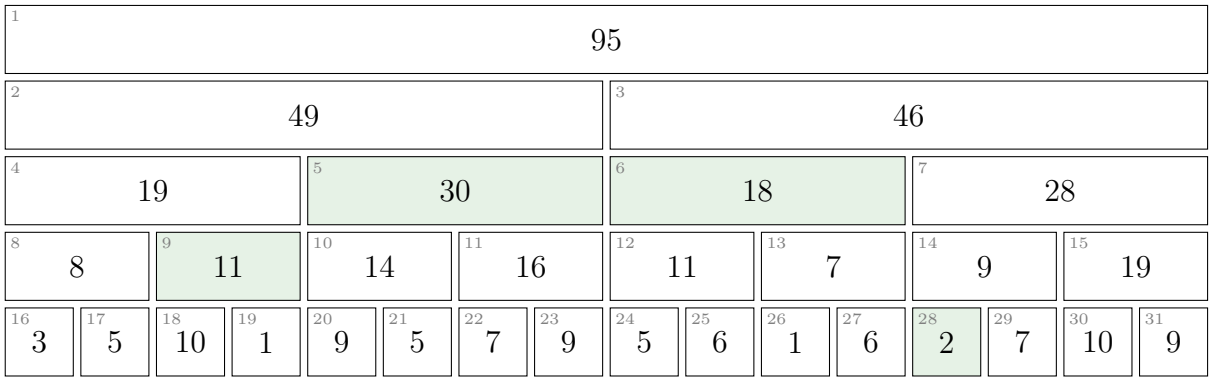

Figura 1.3: Suma intervalului [18, 28] este egală cu suma valorilor nodurilor 9, 5, 6 și 28.

Pornim cu doi pointeri l și r din capetele interogării date. Apoi procedăm astfel:

- Dacă l este fiu stîng, putem aștepta ca să includem un strămoș al său, care va include și alte poziții utile. În schimb, dacă l este fiu drept, trebuie să îl includem în sumă, căci orice strămoș al său va include și elemente inutile din stînga lui l. Apoi avansăm l spre dreapta.
- Printr-un raționament similar, dacă r este fiu stîng, includem valoarea sa în sumă și avansăm r spre stînga.
- Urcăm pe nivelul următor prin înjumătățirea lui l și r.
- Continuăm cît timp $l \leq r$.

Astfel, vom selecta cel mult două intervale de pe fiecare nivel al arborelui și vom restrînge corespunzător intervalul dat, pînă cînd îl reducem la zero. De aici rezultă complexitatea logaritmică.

```
long long query(int l, int r) { // [l, r] închis
  long long sum = 0;

  l += n;
  r += n;
```

```
  while (l <= r)  {
    if (l & 1) {
      sum += s[l++];
    }
    l >>= 1;

    if (!(r & 1)) {
      sum += s[r--];
    }
    r >>= 1;
  }

  return sum;
}
```

Clarificare: la ultimul nivel, dacă $l = r$, atunci $s[l]$ va fi selectat exact o dată, fie datorită lui l, fie datorită lui r, după cum poziția este impară sau pară.

1.2.5 Căutarea unei sume parțiale

Ca și la AIB-uri, dacă toate valorile sînt pozitive are sens întrebarea: pe ce poziție suma parțială atinge valoarea P? Pentru simplitate, recomand să adăugați o santinelă de valoarea infinită pe poziția n. Aceasta garantează că suma parțială se atinge întotdeauna, iar dacă răspunsul este n, atunci de fapt suma parțială nu există în vectorul fără santinelă.

```
int search(int sum) {
  int pos = 1;

  while (pos < n) {
    pos *= 2;
    if (sum > s[pos]) {
      sum -= s[pos++];
    }
  }

  return pos - n;
}
```

1.2.6 Căutarea într-un arbore de maxime

Dat fiind un vector v cu n elemente, ni se cere să răspundem la interogări de tipul $\langle pos, val \rangle$ cu semnificația: găsiți cea mai mică poziție $i > pos$ pe care se află o valoare $v[i] > val$. În secțiunea următoare vom vedea problemele Points și Împiedicat care au această nevoie.

Pentru rezolvare, să construim peste acest vector un arbore de intervale de maxime. Fiecare nod stochează maximul dintre cei doi fii ai săi. Ca urmare, fiecare nod stochează maximul dintre frunzele pe care le subîntinde. Atunci soluția constă din doi pași:

- Mergi la dreapta și în sus, similar pointerului l din operația de sumă pe interval prezentată anterior. Oprește-te cînd ajungi la un nod cu o valoare $> val$. Știm că acest nod subîntinde cel puțin o frunză de valoare $> val$.
- Din acest nod, coboară în fiul care are la rîndul său o valoare $> val$. Dacă ambii fii au această proprietate, coboară în fiul stîng. Oprește-te cînd ajungi la o frunză.

Pentru a simplifica codul, putem adăuga o santinelă infinită la finalul vectorului, ca să ne asigurăm că problema are soluție.

```
int find_first_after(int pos, int val) {
  pos += n + 1;
```

```
  while (v[pos] <= val) {
    if (pos & 1) {
      pos++;
    } else {
      pos >>= 1;
    }
  }

  while (pos < n) {
    pos = (v[2 * pos] > val) ? (2 * pos) : (2 * pos + 1);
  }

  return pos - n;
}
```

Am inclus acest algoritm, deși este rar întîlnit în pratică, pentru a ilustra flexibilitatea uriașă a arborilor de intervale.

1.2.7 Adaptarea la alte tipuri de operații

Aceeași structură de date poate răspunde la multe alte feluri de actualizări și interogări. Nu detaliem aici, vom studia probleme. Ce este important este să ne dăm seama ce stocăm în fiecare nod și cum combină părintele informațiile din cei doi fii.

1.2.8 Implementarea recursivă

Există și o implementare recursivă, pe care nu o vom discuta acum (o menționez doar ca să o fac de rîs). O vom discuta mai tîrziu în acest capitol. Este păcat că mulți elevi învață și stăpînesc doar acea implementare, pe care o aplică și cînd nu este nevoie de ea, deși implementarea iterativă de mai sus este de 2-3 ori mai rapidă. Implementarea iterativă ar trebui să fie implementarea voastră de referință oricînd este suficientă.

Exemplu: din implementarea iterativă rezultă imediat că:

1. Complexitatea este $\mathcal{O}(\log n)$, întrucît l și r urcă exact un nivel la fiecare iterație.
2. De pe fiecare nivel selectăm cel mult două intervale.

Vă urez succes să demonstrați aceste lucruri în implementarea recursivă. 😈

1.3 Probleme

Bibliografie

[1] CS Academy, *Segment Trees*, URL: https://csacademy.com/lesson/segment_trees.
[2] CP Algorithms, *Segment Tree*, URL: https://cp-algorithms.com/data_structures/segment_tree.html.