Programare cu premeditare

Cătălin Frâncu

Prefață

Aici voi spune ceva introductiv.

- Problemele sînt ordonate după dificultate.
- Pentru Codeforces și Kilonova aveți nevoie de un cont pentru a vedea sursele.

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.

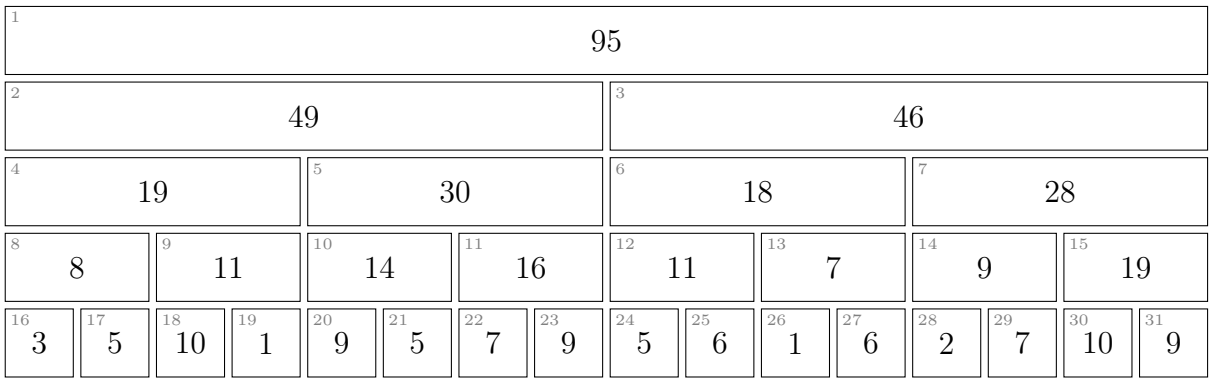

Figura 1.1: Un arbore de intervale cu 16 frunze și 15 noduri interne. Valorile din fiecare celulă reprezintă suma din frunzele subîntinse de acea celulă. Cu cifre mici este notat indicele fiecărei celule.

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.

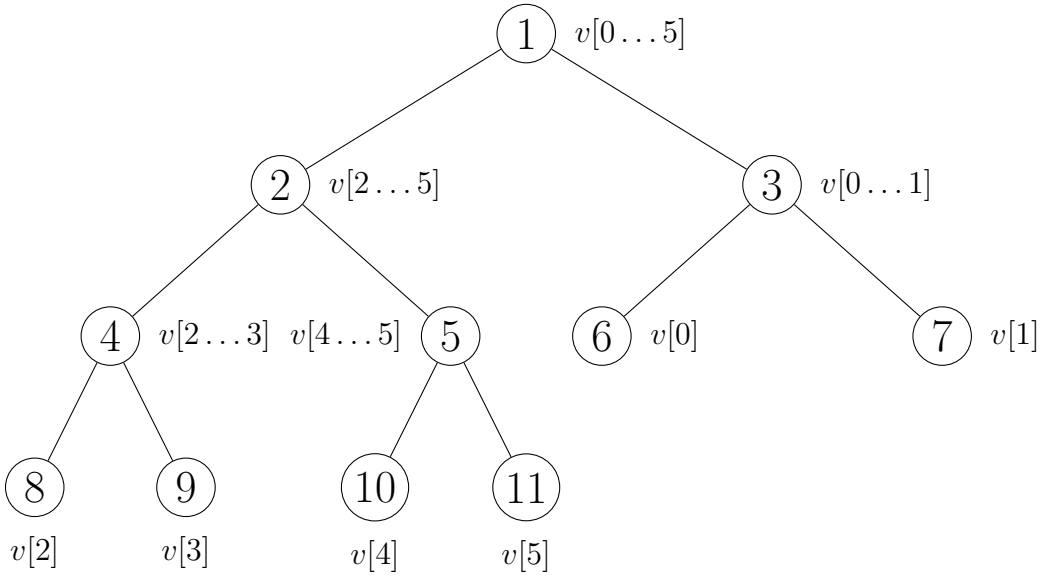

Figura 1.2: Reprezentarea arborilor de intervale cu exact $2n-1$ noduri.

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.

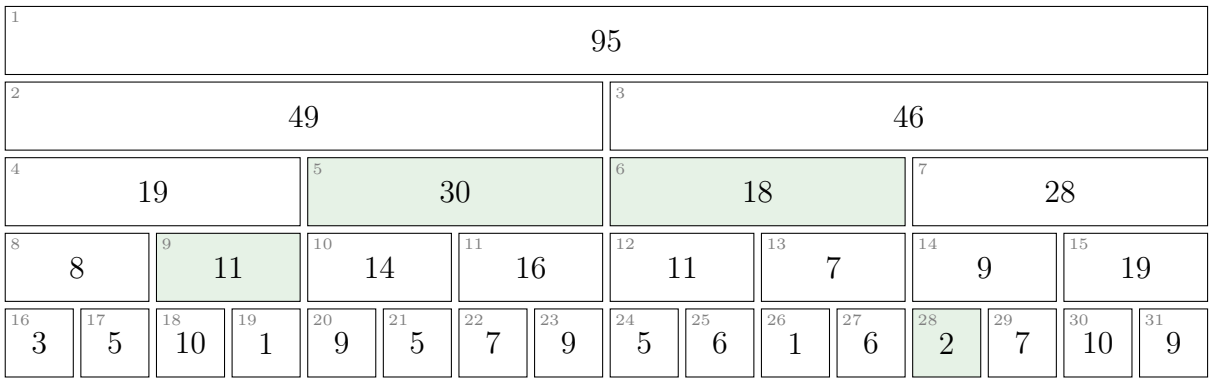

Figura 1.3: Suma intervalului [18, 28] este egală cu suma valorilor nodurilor 9, 5, 6 și 28.

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

1.3.1 Problema Xenia and Bit Operations (Codeforces)

enunț • sursă

Problema este simplisimă. O includ doar ca exemplu de arbore care face operații diferite pe niveluri diferite.

1.3.2 Problema Distinct Characters Queries (Codeforces)

enunț • sursă

Există diverse abordări pentru această problemă. Una este să construim un AIB sau un AINT pentru fiecare caracter, cu memorie totală $\mathcal{O}(\Sigma n)$ (tradițional Σ denotă mărimea alfabetului). Fiecare structură reține pozițiile pe care apare un caracter. Modificările sînt simple: debifăm poziția în AIB-ul corespunzător vechiului caracter și o marcăm în AIB-ul noului caracter. Pentru interogări, verificăm pentru fiecare din cele 26 de caractere dacă suma pe intervalul dat este non-zero. Rezultă o complexitate de $\mathcal{O}(\Sigma q \log n)$.

Dar iată și o soluție mai elegantă, care reduce complexitatea la $\mathcal{O}(q \log n)$, folosind paralelismul nativ pe 32 de biți al procesorului. Vom folosi 26 de biți din fiecare întreg, cîte unul pentru fiecare caracter. Într-o frunză care stochează litera 'f' vom seta pe 1 doar cel de-al șaselea bit, așadar valoarea întreagă va fi `000...000100000`. Apoi, un nod intern va stoca OR-ul pe biți al frunzelor din intervalul acoperit. Acest gen de informație se numește **mască de biți** (engl. *bitmask*).

Ce semnifică acest OR pe biți? Fiecare dintre biți va fi 1 dacă și numai dacă litera corespunzătoare apare cel puțin o dată în intervalul acoperit. Să observăm că bitul 6 va fi 1 indiferent dacă intervalul conține un caracter 'f' sau multiple caractere 'f'. Rezultă că fiecare mască va avea atîția biți setați (biți 1) cîte caractere distincte există în interval.

Facem modificări în acest arbore înlocuind masca din frunză și propagînd valoarea spre strămoși cu operația OR. Pentru a răspunde la interogări,

- colectăm cele $\mathcal{O}(\log n)$ măști care compun interogarea;
- le combinăm cu OR;
- numărăm biții din rezultat, de exemplu cu funcția `__builtin_popcount`.

1.3.3 Problema K-query (SPOJ)

enunț • sursă

Problema fiind offline, este destul de natural să ordonăm interogările. Sper să vă obișnuiți și voi să luați în calcul această posibilitate.

Ordonarea după capătul stîng sau drept nu pare să ducă nicăieri. Exemplu: ordonăm interogările după capătul drept dr. Atunci, după ce adăugăm elementul $a[dr]$ la structura noastră (oricare ar fi ea), trebuie să răspundem la interogări de tipul: cîte numere $> k$ există începînd cu poziția st? Eu nu am reușit să găsesc o structură echilibrată care să răspundă la întrebări. Poate voi reușiți?

În schimb, ordonarea descrescătoare după valoare duce la o soluție relativ directă. Pentru o interogare (st, dr, k), marcăm (cu 1) într-o structură de date toate pozițiile elementelor mai mari decît k. Apoi numărăm valorile 1 din intervalul $[st, dr]$.

Pentru a găsi rapid toate elementele mai mari decît k (care nu au fost deja inserate în structură), rezultă că trebuie să sortăm și vectorul în ordine descrescătoare, reținînd și poziția originală a

fiecărei valori.

În fapt, putem implementa această soluție chiar și cu un AIB. Iată o sursă, identică cu cea bază pe arbori de intervale cu excepția `struct`-ului. În acest caz, timpii de rulare sînt aproape egali, dar în general vă recomand să folosiți AIB unde se poate.

1.3.4 Problema Sereja and Brackets (Codeforces)

enunț • sursă

Iată și o problemă pentru a cărei rezolvare este mai puțin clar că ne ajunge un arbore de intervale. Vom construi un arbore în care nodurile stochează valori mai complexe care se combină după reguli speciale.

Să considerăm o subsecvență contiguă. Din ce constă ea? Dintr-un subșir (pe sărite) care este bine format, plus niște paranteze deschise neîmperecheate, plus niște paranteze închise neîmperecheate. De exemplu, în subșirul `))) ( () ( () ( () (` am evidențiat cu bold cele 6 caractere bine formate. Rămîn 4 paranteze deschise și 3 închise. Să notăm aceste cantități cu f (lungimea subșirului bine format), d (surplusul de paranteze deschise) și i (surplusul de paranteze închise).

Cum combinăm două subsecvențe adiacente (f_1, d_1, i_1) și (f_2, d_2, i_2)? Clar putem concatena porțiunile bine formate. Dar mai mult, putem prelua și $\min(d_1, i_2)$ perechi dintre surplusurile de paranteze deschise, respectiv închise. Șirul rezultat va fi bine format. Ne putem convinge de asta eliminînd porțiunile bine formate f_1 și f_2, ca și cînd ele nu ar exista. Dacă nu sînteți convinși, puteți apela la o definiție echivalentă pentru un șir de paranteze bine format: pentru orice prefix, diferența dintre numărul de paranteze deschise și închise este pozitivă.

Rezultă că intervalul concatenat va avea parametrii:

- $f = f_1 + f_2 + 2\min(d_1, i_2)$
- $d = d_1 + d_2 - \min(d_1, i_2)$
- $i = i_1 + i_2 - \min(d_1, i_2)$

Construcția arborelui se face ca de obicei, combinînd fiii doi cîte doi. La interogare este nevoie de puțină atenție pentru a colecta și combina intervalele în ordinea corectă (de la stînga la dreapta). Ne bazăm pe observația că operația de compunere nu este comutativă, dar este asociativă.

1.3.5 Problema Copying Data (Codeforces)

enunț • sursă

Aici întîlnim o formă complementară a arborilor de intervale: actualizări pe interval și interogări punctuale (*range update, point query*). Mecanismul necesar folosește o reprezentare puțin diferită. O problemă foarte similară este Range Update Queries (CSES).

(Cei dintre voi care stăpînesc arborii de intervale cu propagare *lazy* vor fi tentați să se repeadă la aceia: Pe fiecare nod ținem informația *lazy* că segmentul din b a fost suprascris cu un segment

din a începînd de la o poziție p (sau cu o deplasare $\pm p$, cum preferați). La actualizări, propagăm informația la fii după nevoie. La interogare, propagăm informația pînă în frunza cerută, pentru a afla de unde provine. Dar nu este nevoie de aceste complicații.)

Să pornim de la observația de bun simț: Dacă o copiere acoperă o poziție, atunci la descompunerea sa în intervale, unul dintre acele intervale va fi strămoș al poziției poziția (*duh!*).

Ne vom folosi și de numerele de ordine ale interogărilor, care vor funcționa ca niște momente de timp între 1 și q. Acum, să construim un arbore de intervale care, pentru o operație de copiere (x, y, k):

- Descompune intervalul $[y, y + k - 1]$ prin metoda obișnuită.
- Notează pe fiecare interval momentul t și diferența $x - y$.

Dacă ulterior o altă copiere va acoperi unul dintre aceste intervale, vom nota acolo momentul t' și diferența $x' - y'$. Atunci ultimul moment (și, implicit, ultima proveniență) a suprascrierii unei poziții este dată de cel mai mare moment de timp **dintre toți strămoșii poziției**.

1.3.6 Problema Points (Codeforces)

enunț • sursă

Problema are rating de 2800 pentru că se compune din multe blocuri, dar niciunul nu este de speriat, căci sîntem deja versați în arbori de intervale. 😎 Aș zice că problema ar fi grea la un baraj ONI sau ușoară la lot.

Ca să putem construi un arbore de intervale, în primul rînd normalizăm coordonatele x. Păstrăm și o tabelă cu valorile originale, căci pe acelea trebuie să le afișăm.

Am putea reformula întrebarea pentru operația `find x y`: dintre toate punctele cu $x' > x$, există vreunul cu $y' > y$? Ne gîndim că am putea folosi un AINT de maxime, indexat după x, cu valori din y, cu interogarea: „Caută maximul pe intervalul $[x + 1, n)$ și spune-mi dacă este mai mare decît y”.

Dar astfel aflăm doar dacă există un punct. Ca să-l găsim, întrebarea corectă este: „dă-mi cea mai din stînga poziție după x pe care maximul depășește y”. Din fericire, putem face asta cu același AINT maxime, așa cum am explicat în secțiunea de teorie:

1. Pornind de la prima poziție validă (în cazul nostru, $x + 1$), mergem în sus și spre dreapta, spre intervale tot mai mari, pînă cînd găsim o poziție de valoare $> y$. Ca să evităm cazurile particulare, adăugăm la finalul vectorului o santinelă de valoare infinită.
2. De la această poziție, coborîm în timp ce menținem în vizor valoarea $> y$. Dacă putem coborî în orice direcție, preferăm stînga.

Astfel putem gestiona operațiile de adăugare (cînd maximul pentru un x fixat poate doar să crească). Următoarea întrebare este cum gestionăm ștergerile. Cea mai directă soluție este să menținem cîte un `set` STL pentru fiecare coordonată x. Suma mărimilor acestor seturi nu va depăși n. Cu metoda `rbegin()` putem afla noul maxim după inserări și ștergeri.

Ultima întrebare, odată ce stabilim că răspunsul pentru `find x y` este la abscisa x', este: care dintre punctele cu această abscisă este răspunsul? Folosim același set și metoda `upper_bound()` pentru a afla cel mai mic y' strict mai mare decît y.

Complexitatea soluției este $\mathcal{O}(n \log n)$, atît pentru normalizarea inițială cît și pentru procesarea operațiilor. Fiecare operație necesită o căutare în set și o căutare sau modificare în AINT.

1.3.7 Problema Medwalk (Lot 2025)

enunț • sursă

Problema admite și o soluție diferită, mult mai rapidă, bazată pe AIB-uri 2D, dar iată o soluție care folosește doar arbori de intervale.

Din enunț putem defini forma drumului: el va merge pe linia de sus a unor coloane, apoi va folosi ambele linii de pe o coloană c pentru a coborî, apoi va merge pe linia de jos a coloanelor rămase. Acum, să presupunem că avem un oracol care, pentru orice interogare, ne spune coloana c. Atunci vom muta restul coloanelor fie în stînga, fie în dreapta lui c, pentru a folosi valoarea de sus sau de jos, oricare este mai mică.

Cu alte cuvinte, mulțimea de valori de pe drumul care minimizează medianul constă din

- minimele de pe toate coloanele;
- minimul dintre maximele de pe coloane.

Răspunsul la fiecare interogare este elementul median al acestei mulțimi. Logica pentru a afla a k-a valoare este relativ simplă și implică trei valori: al k-lea minim, al $k-1$-lea minim și minimul maximelor. De aici înainte, putem abstractiza matricea ca doi vectori, unul cu minimele perechilor și altul cu maximele. De exemplu, cînd o coloană se modifică din (3, 6) în (3, 2), atunci minimul se modifică din 3 în 2, iar maximul din 6 în 3.

De aceea, avem nevoie de două structuri independente:

- O structură pentru maxime, care să admită actualizări punctuale și interogare de minim pe interval.
- O structură pentru minime, care să admită actualizări punctuale și interogări de al k-lea element pe interval.

Pentru prima structură, ochiul nostru de-acum experimentat ne spune că putem folosi un simplu AINT. Dar pentru a doua? Am găsit pe StackOverflow o idee bine explicată, pe care o reiau.

Vom folosi un arbore de intervale **pe valori**. Așadar, nu indexăm pozițiile conform cu pozițiile din vector, ci cu valorile existente în vector. Fiecare frunză din aint, corespunzătoare unei valori v, reține o colecție ordonată (un `set`, în esență) cu pozițiile pe care apare valoarea v. Fiecare nod intern reține reuniunea colecțiilor fiilor săi. Cu alte cuvinte, dacă un nod subîntinde valorile $[l, r]$, colecția sa va enumera toate pozițiile pe care apar valori între l și r.

Remarcăm că memoria necesară este $\mathcal{O}(n \log V_{max})$, deoarece aint-ul conține V_{max} valori, deci are înălțime $\log V_{max}$, iar fiecare poziție din vectorul original va fi enumerată în $\log V_{max}$ colecții.

Pentru actualizare, trebuie să ștergem poziția modificată din lista vechii valori minime și din listele tuturor strămoșilor. Apoi inserăm poziția în listele noii valori minime. De exemplu, dacă minimul coloanei 100 se modifică din 30 în 20, atunci de la poziția 30 din aint și din toți strămoșii eliminăm elementul 100 din colecție. Apoi la poziția 20 în aint și în toți strămoșii inserăm elementul 100.

Rămîne să descriem interogările. Pentru a afla al k-lea minim dintr-un interval de coloane $[l, r]$, pornim din rădăcina arborelui de intervale (luînd așadar în calcul toate valorile de la 0 la V_{max}). Consultăm fiul stîng (valorile $1 \ldots V_{max}/2$) și ne întrebăm: cîte apariții au aceste valori pe poziții din $[l, r]$? Putem răspunde la această întrebare printr-o diferență, reducînd întrebarea la forma: cîte apariții au aceste valori pe pozițiile $0 \ldots r$? Așadar, trebuie numărate elementele mai mici sau egale cu r din setul rădăcinii. Set-ul simplu din STL nu poate gestiona această întrebare, dar putem folosi un set extins din PB/DS. Nu detaliem acum, dar ne vom reîntîlni cu acest tip de date.

Dacă numărul de valori între 0 și $V_{max}/2$ care apar pe poziții între l și r este $\geq k$, atunci acolo se va afla și al k-lea element, deci coborîm în fiul stîng. Altfel coborîm în fiul drept.

Complexitatea algoritmului este $\mathcal{O}((n+q) \log n \log V_{max})$. De exemplu, fiecare interogare coboară $\log V_{max}$ niveluri, iar la fiecare nivel face o căutare într-un set de $\mathcal{O}(n)$ elemente în timp $\mathcal{O}(\log n)$.

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.

Capitolul 2

Arbori de intervale cu propagare *lazy*

2.1 Operații pe interval

Să reluăm exemplul din capitolul trecut și să spunem acum că dorim să adăugăm 100 pe intervalul [2, 12], corespunzător nodurilor [12, 28] din arbore.

Ca să nu facem efort $\mathcal{O}(n)$, vom descompune intervalul ca mai înainte și vom nota informația „+100" în nodurile 9, 5, 6 și 28, cu semnificația că valoarea reală a tuturor frunzelor de sub aceste noduri a crescut cu 100.

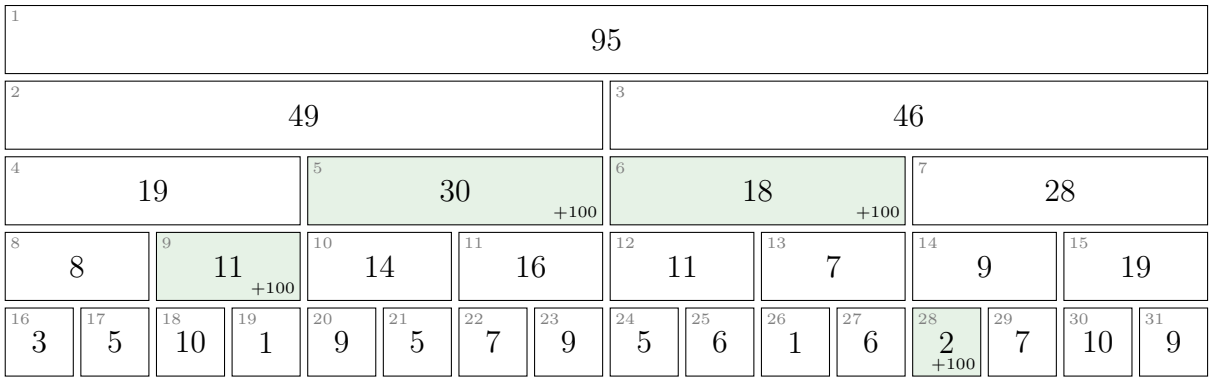

Figura 2.1: Pentru a adăuga 100 pe intervalul [18, 28], notăm valoarea *lazy* 100 pe nodurile 9, 5, 6 și 28.

Aceasta este o **informație *lazy***: o informație care stă într-un nod intern și care trebuie propagată tuturor frunzelor subîntinse de acel nod. Totuși, amînăm efortul acestei propagări pînă cînd el devine strict necesar; tocmai de aceea se numește **propagare *lazy***. (Mulți elevi denumesc întreaga structură „AINT cu *lazy*", dar asta este... lene.)

Evaluarea *lazy* este un concept des întîlnit:

- Memoizarea unor valori într-un vector / matrice, cu speranța că nu va fi nevoie să calculăm tabelul complet.

- Amînarea evaluării lui `y` în expresia booleană `x || y`, cu speranța că `x` va fi evaluat ca adevărat, iar `y` va deveni irelevant.
- Inițializarea unei componente costisitoare dintr-un program doar cînd devine necesară (o conexiune la baza de date, o zonă a hărții dintr-un joc).

Așadar, definim un al doilea vector numit `lazy` și executăm `lazy[x] += 100` pe pozițiile 9, 5, 6 și 28.

Motivul pentru care treaba se complică este următorul. Dacă acum primim o interogare de sumă pe intervalul [25,29]? Nu putem să însumăm, ca de obicei, pozițiile 25, 13 și 14, căci pierdem din vedere că unele dintre noduri au (cîte) +100. Sigur, putem lua asta în calcul, dar trebuie să clarificăm operațiile, altfel efortul poate deveni $\mathcal{O}(n)$.

În primul rînd, introducem două funcții noi (le puteți include în alte funcții, dar pentru claritate le puteți declara de sine stătătoare):

- `push()`, care propagă informația *lazy* de la un nod la fiii săi;
- `pull()`, care combină în părinte informația din cei doi fii după o actualizare.

```
void push(int node, int size) {
  s[node] += lazy[node] * size;
  lazy[2 * node] += lazy[node];
  lazy[2 * node + 1] += lazy[node];
  lazy[node] = 0;
}

void pull(int node, int size) {
  s[node] =
    s[2 * node] + lazy[2 * node] * size / 2 +
    s[2 * node + 1] + lazy[2 * node + 1] * size / 2;
}
```

Pentru problema dată (dar nu pentru toate problemele), codul are nevoie să știe numărul de frunze subîntinse (`size`).

Să presupunem acum că dorim să calculăm suma intervalului [2, 12] și că este posibil să avem niște sume *lazy* în multe alte noduri. Știm că codul descompune interogările în intervale mai scurte și nu urcă mai sus de acestea. Dacă există valori *lazy* mai sus (să zicem în rădăcină), codul nu va afla de ele. De aceea, în pregătirea interogării, trebuie să vizităm toți strămoșii intervalului și să propagăm în jos (*push*) informația *lazy*. Dar, dacă ne gîndim, lista completă a acestor strămoși constă doar din strămoșii capetelor de interval! Pentru intervalul [18,28], este nevoie să propagăm în jos informația *lazy* din strămoșii lui 18 (adică 1, 2, 4 și 9) și ai lui 28 (adică 1, 3, 7 și 14).

Dacă apelăm push din acești strămoși, de sus în jos, avem garanția că informația pe intervalele dorite este la zi. Vă rămîne vouă ca experiment de gîndire să demonstrați că, după operațiile *push*, nu va mai exista informație *lazy* în niciun strămoș al niciunei poziții din interogare.

Astfel obținem o funcție foarte similară cu cea din capitolul trecut

```
void push_path(int node, int size) {
  if (node) {
    push_path(node / 2, size * 2);
    push(node, size);
  }
}

long long query(int l, int r) {
  long long sum = 0;
  int size = 1;

  l += n;
  r += n;
  push_path(l / 2, 2); // pornim din părinte
  push_path(r / 2, 2);

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

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

  return sum;
}
```

Dacă viteza este crucială, putem scrie și o funcție `push_path` cu circa 10% mai rapidă, iterativă, folosind operații pe biți. Să considerăm nodul $22 = 10110_{(2)}$. Strămoșii lui sînt 1, 2, 5 și 11 care au respectiv reprezentările binare 1, 10, 101 și 1011, care sînt fix prefixele lui 10110! Deci îl vom deplasa pe 10110 la dreapta cu 4, 3, 2 și respectiv 1 bit pentru a-i obține strămoșii.

```
void push_path(int node) {
  int bits = 31 - __builtin_clz(n);
  for (int b = bits, size = n; b; b--, size >>= 1) {
    int x = node >> b;
    push(x, size);
  }
}

// Acum primul apel este chiar din frunză:
```

```
...
push_path(l);
push_path(r);
...
```

Actualizările sînt foarte similare. Apelăm `pull()` după terminarea actualizărilor, deoarece trebuie să lăsăm arborele într-o stare coerentă și trebuie să preluăm orice modificare de la fii.

```
void pull_path(int node) {
  for (int x = node / 2, size = 2; x; x /= 2, size <<= 1) {
    pull(x, size);
  }
}

void update(int l, int r, int delta) {
  l += n;
  r += n;
  int orig_l = l, orig_r = r;
  push_path(l / 2, 2);
  push_path(r / 2, 2);

  while (l <= r)  {
    if (l & 1) {
      lazy[l++] += delta;
    }
    l >>= 1;

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

  pull_path(orig_l);
  pull_path(orig_r);
}
```

Iată și o altă implementare care combină bucla `while` principală cu funcția `pull_path`.

Notă: În această implementare, valoarea *lazy* se aplică întregului subarbore, inclusiv nodului însuși. În implementarea de pe CP Algorithms, valoarea *lazy* se aplică subarborelui fără nodul însuși. Oricare dintre formulări este acceptabilă, cîtă vreme o folosiți consecvent.

Notă: În practică, cîmpurile `lazy` și `s` merită încapsulate într-un `struct`. Datorită localității acceselor la memorie, diferența de viteză este notabilă (circa 25%). Aici le-am lăsat separate pentru concizie.

2.2 Implementare recursivă (actualizări punctuale)

Lecția trecută am spus că există și o implementare recursivă. Să o examinăm acum (mulți o știți deja).

```
void update(int node, int pl, int pr, int pos, int delta) {
  if (pr - pl == 1) {
    s[node] += delta;
  } else {
    int mid = (pl + pr) >> 1;
    if (pos < mid) {
      update(2 * node, pl, mid, pos, delta);
    } else {
      update(2 * node + 1, mid, pr, pos, delta);
    }
    s[node] = s[2 * node] + s[2 * node + 1];
  }
}
```

Metoda recursivă cară după ea 5 parametri:

- `node`: nodul curent din arbore (aka poziția în vector);
- `pl`, `pr`: intervalul din vectorul inițial acoperit de `node`. Eu am optat pentru implementarea cu `pl` inclusiv și `pr` exclusiv. Dacă preferați intervale închise, este OK.
- `pos`, `delta`: poziția de modificat și valoarea de adăugat/scăzut.

Vedem că funcția coboară recursiv în fiul stîng sau fiul drept, după caz. Un exemplu de apel ar fi:

```
update(1, 0, n, some_pos, some_val);
```

Mai interesant, iată și implementarea funcției de interogare (sumă pe interval):

```
long long query(int node, int pl, int pr, int l, int r) {
  if (l >= r) {
    return 0;
  } else if ((l == pl) && (r == pr)) {
    return s[node];
  } else {
    int mid = (pl + pr) >> 1;

    return
      query(node * 2, pl, mid, l, min(r, mid)) +
      query(node * 2 + 1, mid, pr, max(l, mid), r);
  }
}
```

Regăsim trei din aceiași parametri, `node`, `pl` și `pr`. În plus,

- `l`, `r`: Intervalul [închis, deschis) pe care dorim să calculăm suma.

Funcția se reapelează pe cei doi fii, restrîngînd corespunzător intervalul $[l, r)$. Iată o imagine care arată arborele de apeluri pentru calculul sumei pe intervalul [3, 10):

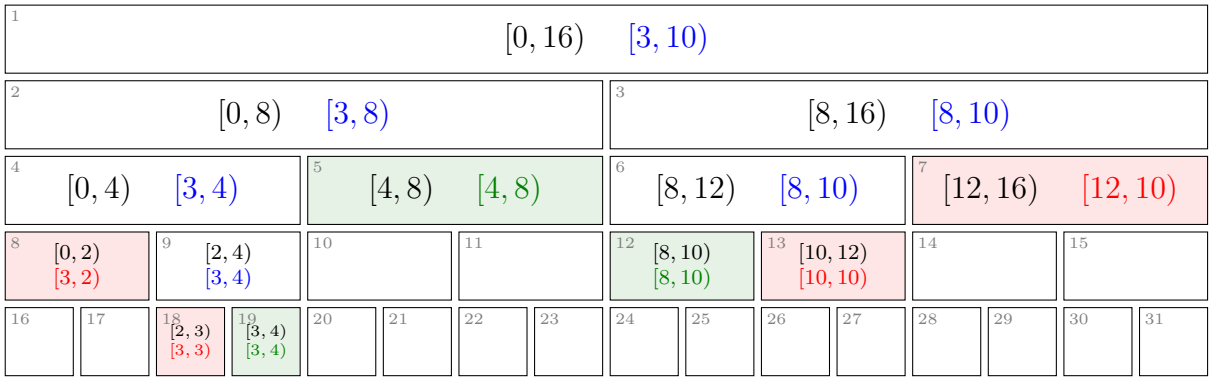

Figura 2.2: Arborele de apeluri în funcția de actualizare recursivă pe intervalul $[3, 10)$.

Am ilustrat cu albastru nodurile care au nevoie să-și apeleze descendenții, cu verde nodurile selectate integral, iar cu roșu nodurile eliminate.

Complexitatea rămîne $\mathcal{O}(\log n)$, deși funcția se reapelează pentru ambii fii. De ce?

Discutăm implementarea recursivă pentru că ea plutește prin supa culturală și vreau să puteți citi cod scris astfel. Dar ea este un exemplu de dopaj, de implementare repetată *mot à mot* indiferent de nevoile problemei. Implementarea recursivă este de 2-3 ori mai lentă decît cea iterativă pentru actualizări punctuale. Presupun că există două motive:

Implementarea recursivă cară după ea 5-6 parametri la fiecare apel, care trebuie copiați, puși/scoși de pe stivă etc. Implementarea iterativă folosește doar 3 variabile.

Implementarea recursivă este nevoită să pornească din rădăcină, să coboare pînă la frunze, apoi să revină din recursivitate. Implementarea iterativă se oprește imediat ce termină de descompus intervalul $[l, r]$.

La varianta cu propagare *lazy* diferența de timp aproape dispare, pentru că ambele implementări trebuie să urce pînă la rădăcină.

Nu vă năpustiți la implementarea recursivă dacă nu este nevoie. Rezistați tentației de a fi leneși, de a învăța o singură structură de date, pe care să o pictați indiferent de situație! Trebuie să aspirați la mai mult de atît, dacă este să vă meritați locul în lot.

2.3 Implementare recursivă (actualizări pe interval)

În această implementare, observăm cum:

- apelăm `push` înainte de reapelarea recursivă, pentru a-i garanta fiecărui nod că deasupra sa nu mai există informații lazy;

- apelăm `pull` după revenirea din recursivitate, ca să lăsăm arborele într-o stare coerentă.

```
long long query(int node, int pl, int pr, int l, int r) {
  if (l >= r) {
    return 0;
  } else if ((l == pl) && (r == pr)) {
    return s[node] + lazy[node] * (r - l);
  } else {
    push(node, pr - pl);
    int mid = (pl + pr) >> 1;
    return
      query(node * 2, pl, mid, l, min(r, mid)) +
      query(node * 2 + 1, mid, pr, max(l, mid), r);
  }
}

void update(int node, int pl, int pr, int l, int r, int delta) {
  if (l >= r) {
    return;
  } else if ((l == pl) && (r == pr)) {
    lazy[node] += delta;
  } else {

    push(node, pr - pl);
    int mid = (pl + pr) >> 1, child_size = (pr - pl) >> 1;
    update(2 * node, pl, mid, l, min(r, mid), delta);
    update(2 * node + 1, mid, pr, max(l, mid), r, delta);
    pull(node, child_size);

  }
}

void process_ops() {
  for (int i = 0; i < num_queries; i++) {
    if (q[i].t == OP_UPDATE) {
      update(1, 0, n, q[i].l - 1, q[i].r, q[i].val);
    } else {
      answer[num_answers++] = query(1, 0, n, q[i].l - 1, q[i].r);
    }
  }
}
```

2.4 Probleme

2.4.1 Problema Polynomial Queries (CSES)

enunț • sursă cu AINT iterativ • sursă cu AINT recursiv

Problema seamănă mult cu cea discutată la teorie, dar pe intervale nu mai adăugăm constante, ci progresii aritmetice. Așadar, pare natural să reținem exact această informație *lazy*: în fiecare nod reținem că în fiecare frunză acoperită de acel nod trebuie să adăugăm cîte un termen al unei progresii cu un anumit prim element și pasul (deocamdată) 1. De exemplu, dacă în figura 2.1 facem o actualizare pe intervalul [18, 28], atunci în nodul 5 notăm progresia cu primul termen 3 și pasul 1. Informația *lazy* este o pereche $\langle 3, 1 \rangle$.

Trebuie tratate atent diversele cazuri care iau naștere. Dacă două progresii acoperă același interval, vor lua naștere progresii cu pas mai mare decît 1. Să luăm un exemplu:

- Progresia cu primul termen 5 și pasul 3, așadar 5, 8, 11, 14, ...
- Progresia cu primul termen 2 și pasul 7, așadar 2, 9, 16, 23, ...
- După însumare dorim să avem termenii 7, 17, 27, 37, ...
- Rezultă că suma este și ea o progresie cu primul termen 7 și pasul 10. Cu alte cuvinte, informațiile *lazy* se pot compune ușor: $\langle 5, 3 \rangle + \langle 2, 7 \rangle = \langle 7, 10 \rangle$.

La propagarea în jos a informației *lazy*, în cei doi fii vom adăuga progresii cu același pas. În fiul drept, primul termen trebuie calculat, dar este ușor. Dacă într-un nod care acoperă 16 elemente avem o progresie cu primul element 3 și pasul 5, atunci fiul drept va începe cu al nouălea termen al progresiei:

$$3 + 5 \cdot (16/2) = 43$$

La operațiile de adăugare, pe toate intervalele din descompunere vom aduna progresii cu pasul 1, dar primul element diferă pentru fiecare interval (la fel, nu este greu de calculat).

Contractul pe care l-am ales pentru implementarea iterativă este:

- `first` și `step` înseamnă că pe nodurile din intervalul acoperit trebuie adăugate valorile `first`, `first + step`, `first + 2 * step`, ...
- Valoarea `s` din fiecare nod **nu** include și suma progresiei dată de `<first, step>` din acel nod.