UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA

DE

MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licență

Variații și aplicații ale arborilor de intervale în programarea competitivă

Absolvent

Enache Alexandru

Coordonator științific

Lect. dr. Dumitran Adrian Marius

București, iunie 2023

Variații și aplicații ale arborilor de intervale în programarea competitivă

Enache Alexandru

Rezumat

Lucrarea de față este dedicată atât pasionaților de structuri de date și programare competitivă, cât și elevilor și studenților care se pregătesc pentru participarea la olimpiadă sau concursuri precum ACM-ICPC. În această lucrare este prezentată una dintre cele mai versatile structuri de date, arborele de intervale, precum și diverse variații și aplicații ale acesteia. Fiecare capitol va fi acompaniat de probleme culese de la competiții naționale și internaționale. Acestea au rolul de a complementa conceptele discutate și a sublinia utilitatea acestora practică, pornind de la o soluție naivă și ajungând la o soluție optimă, accentul fiind pus pe complexitatea timp.

Abstract

This paper is dedicated to both enthusiasts of data structures and competitive programming, as well as to pupils and students who are preparing to participate in the Olympiad or competitions such as ACM-ICPC. This paper presents one of the most versatile data structures, the Segment tree, as well as its numerous variations and applications. Each chapter will be accompanied by problems collected from national and international competitions. These have the role of complementing the discussed concepts and underlining their practical utility, starting from a naive solution and reaching an optimal solution, the emphasis being on time complexity.

Cuprins

I Introducere

Arborii de intervale sunt structuri de date ce permit interogări asupra subsecvențelor unui șir memorând în noduri, începând de la rădăcina, date despre intervale din șir, urmând o regulă logaritmică într-o paradigmă de tipul Divide et Impera. Datorită modului în care este reținută informația, arborii de intervale acceptă totodată și actualizarea elementelor șirului în timp logaritmic.

În 1977, Victor Klee a formulat și rezolvat problema reuniunii unui șir de intervale [7] în complexitate timp $O(N \cdot log_2 N)$. În același an, Jon Bentley a ridicat generalizarea naturală în 2 dimensiuni a acestei probleme. În timp ce studia soluții pentru această problemă, Jon Bentley a introdus pentru prima dată conceptul de arbore de intervale în rezolvarea sa propusă [3]. Astfel, acesta a realizat un algoritm de complexitate $O(N \cdot log_2 N)$ folosind o tehnică de baleiere care menține o structură de arbore de intervale în spate.

De obicei, arborii de intervale sunt cunoscuți pentru operații simple precum aflarea sumei unei subsecvențe sau a elementului cu valoare maximă/minimă din aceasta. În realitate, datele reținute în nodurile arborelui pot avea o natură diversă, permițând interogări complexe asupra subsecvențelor unui șir. În plus, există variații ale arborilor de intervale, precum arborele de intervale persistent sau arborele de intervale 2D, care fac posibile noi operații.

În programarea competitivă, structura flexibilă a arborilor de intervale îi face să fie o unealtă utilă în vederea rezolvării problemelor într-o complexitate timp optimă. Din acest motiv, se pot găsi diverse resurse pe site-uri și forum-uri specializate în programare competitivă precum Infoarena, Codeforces si CP-Algorithms. În același timp, există un număr limitat de materiale didactice pe acest subiect, mai ales în limba română, unul dintre puținele astfel de materiale accesibile în limba română fiind un articol succint scris de Dana Lica [13]. Acest fapt motivează atât structura lucrării, cât și modul de expunere a informațiilor, aceasta urmând a fi integrată, alături de alte lucrări cu scop similar, ca parte a unei lucrări de sinteză dedicată structurilor de date.

În lucrarea de față, am abordat diverse variații ale acestei structuri de date, punând accentul pe tipurile de operații acceptate și complexitatea timp a acestora. Pe parcursul capitolelor, teoria este complementată de probleme reprezentative ale căror soluții corespund cu conceptele prezentate în capitolul respectiv.

În primul capitol este prezentată forma de bază a arborelui de intervale, care permite operațiile de interogare a unei subsecvențe din șir, actualizarea unui element din șir și căutarea unui prefix al șirului. În următoarele trei capitole sunt prezentate variații ale acestuia precum Lazy propagation, care introduce actualizarea unei întregi subsecvențe din șir, arborele de intervale

persistent, care poate să rețină mai multe stări ale arborelui, și arborele de intervale cu mai multe dimensiuni.

În capitolul **VI** am arătat aplicabilitatea arborilor de intervale în alți algoritmi, fiind prezentat algoritmul Heavy path decomposition, precum și optimizarea soluției de programare dinamică în aflarea unui subșir crescător de lungime maximală.

În ultimul capitolul este prezentat arborele indexat binar, o alternativă a arborelui de intervale mai puțin flexibilă, fiind folosit în special pentru aflarea sumei elementelor aparținând subsecvențelor unui șir. Acesta păstrează complexitatea teoretică logaritmică pe operații, însă este mai ușor de implementat și se comportă mai bine decât arborele de intervale în practică.

II Arbori de intervale

II.1 Concept

Definiție. Arborele de intervale este structura de date reprezentată de un arbore binar în care fiecare nod are asociat un interval $[left, right]$ cu $left \leq right$. Acest interval va delimita o subsecvență a unui șir notat s, nodul reținând date despre aceasta. Notăm lungimea șirului s cu N. Astfel, rădăcina arborelui va avea asociat intervalul întregului șir $[1, N]$.

Frunzele arborelui corespund intervalelor elementare de lungime 1, iar restul nodurilor vor avea ca fiu stâng nodul asociat intervalului $[left, \lfloor \frac{left+right}{2} \rfloor]$, iar ca fiu drept nodul asociat intervalului $[\lfloor \frac{left+right}{2} \rfloor + 1, right]$.

Deoarece fiecare nod care nu este frunză are exact doi fii, iar intervalul fiilor este jumătate din cel al părintelui acestora, înălțimea arborelui este $O(log_2 N)$.

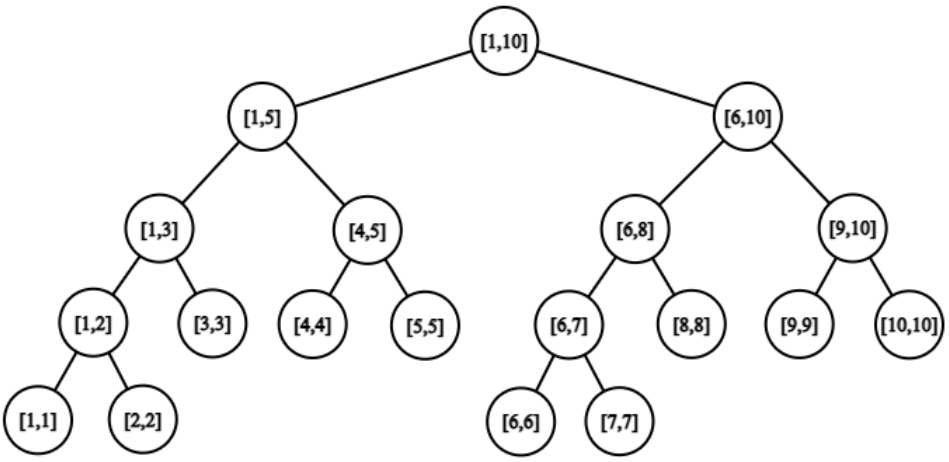

Figura II.1: Arbore de intervale asociat unui șir de 10 elemente

II.2 Operații

Algoritm	Complexitate Timp
Inițializare	$O(N)$
Actualizarea unui element	$O(log_2 N)$
Interogarea unei subsecvențe	$O(log_2 N)$
Căutarea unui prefix	$O(log_2 N)$

Scop. Scopul principal al arborelui de intervale este reprezentat de operațiile de actualizare și interogare realizate în $O(log_2 N)$ complexitate timp. În continuare vom analiza operațiile

de inițializare, actualizarea unui element și interogarea unei subsecvențe, precum și căutarea unui prefix. De menționat că este posibilă și actualizarea unui subsecvențe tot în $O(log_2 N)$ complexitate timp, acest lucru fiind discutat în următorul capitol.

Inițializarea unui arbore de intervale

Concept. Avem un vector v indexat de la 1 la N și vrem să inițializăm valorile din arborele de intervale pe baza acestuia. Practic, un nod asociat intervalului $[left, right]$ să corespundă valorilor $v[left], v[left + 1], ..., v[right]$. Vom reține arborele ca un vector notat $tree$ unde $tree[1]$ este rădăcina, iar $tree[node \cdot 2]$ și $tree[node \cdot 2 + 1]$ reprezintă fiul stâng, respectiv fiul drept al nodului $tree[node]$ pentru orice nod $node$ care nu este frunză.

Vectorul $tree$ va avea nevoie de cel mult $4 \cdot N$ elemente, deoarece în cel mai rău caz numărul de noduri din arbore poate fi estimat ca $1 + 2 + 4 + \cdots + 2^{\lceil \log_2 N \rceil} < 2^{\lceil \log_2 N \rceil + 1} < 4 \cdot N$. [12]

Algorithm 1 Inițializarea unui arbore de intervale [12]

```
1:  function INIT(node, left, right)
2:      if left = right then                          ▷ Am ajuns la o frunză
3:          tree[node] ← v[left]                      ▷ Frunza primește valoarea din v
4:          return
5:      end if
6:      mid ← (left + right)/2                        ▷ Calculăm mijlocul intervalului
7:      INIT(node · 2, left, mid)                     ▷ Continuăm pe fiul stâng
8:      INIT(node · 2 + 1, mid + 1, right)            ▷ Continuăm pe fiul drept
9:
10:     COMBINE(node, node · 2, node · 2 + 1)  ▷ Combinăm valorile din fiul stâng și drept în
    nodul curent
11: end function
12:
13: INIT(1, 1, N)          ▷ Apelăm funcția cu nodul rădăcina 1 și întregul interval [1, N]
```

Complexitate. Vom trece prin fiecare nod din arborele de intervale o singură dată. Cum numărul maxim de noduri este $4 \cdot N$, complexitatea timp este $O(N)$.

Actualizarea unui element

Concept. Vrem să actualizăm valoarea unui element din șir de pe poziția notată pos. Pentru a realiza acest lucru, va trebui să modificăm frunza din arborele de intervale corespunzătoare intervalului $[pos, pos]$ împreună cu drumul de la rădăcina arborelui până la aceasta.

Actualizarea poate să fie și de tip absolut, unde avem o valoare nouă ce înlocuiește valoarea trecută, dar și de tip relativ, când se realizează o modificare a valorii existente (de exemplu se

adaugă o valoare x la ea). Vom reprezenta mai jos actualizarea de tip absolut, cealaltă realizându-se într-un mod analog.

Algorithm 2 Actualizarea unui element [12]

```
1: function UPDATE(node, left, right, pos, val)
2:     if left = right then                          ▷ Am ajuns la frunza corespunzătoare poziției
3:         tree[node] ← val                          ▷ Frunza primește valoarea nouă
4:         return
5:     end if
6:     mid ← (left + right)/2                        ▷ Calculăm mijlocul intervalului
7:     if position ≤ mid then
8:         UPDATE(node · 2, left, mid, pos, val)     ▷ Continuăm pe fiul stâng
9:     else
10:        UPDATE(node · 2 + 1, mid + 1, right, pos, val)   ▷ Continuăm pe fiul drept
11:    end if
12:
13:    COMBINE(node, node · 2, node · 2 + 1)  ▷ Combinăm valorile din fiul stâng și drept în
       nodul curent
14: end function
15:
16: UPDATE(1, 1, N, pos, val)                        ▷ Apelăm funcția
```

Complexitate. Vom coborî pe drumul de la rădăcina arborelui spre frunza asociată poziției. Deoarece la fiecare pas, intervalul curent se înjumătățește, avem $O(log_2 N)$ complexitate timp.

Interogarea unei subsecvențe

Concept. Vrem să aflăm o informație asociată unei subsecvențe $[a, b]$ din șir. Această informație poate fi de mai multe tipuri, depinzând de modul cum generăm arborele, acest lucru fiind detaliat în secțiunea următoare. Un exemplu simplu este suma elementelor din subsecvență.

Algorithm 3 Interogarea unei subsecvențe [12]

```
1:  function QUERY(node, left, right, a, b)
2:      if a ≤ left and right ≤ b then          ▷ Am ajuns la un interval cuprins în [a, b]
3:          return tree[node]                   ▷ Returnăm valoarea din arbore
4:      end if
5:      mid ← (left + right)/2                  ▷ Calculăm mijlocul intervalului
6:      if a ≤ mid then
7:          QUERY(node · 2, left, mid, a, b)    ▷ Continuăm pe fiul stâng
8:      end if
9:      if mid + 1 ≤ b then
10:         QUERY(node · 2 + 1, mid + 1, right, a, b)   ▷ Continuăm pe fiul drept
11:     end if
12:
13:     return ...   ▷ Dacă se continuă pe ambii fii, combinăm valorile și returnăm. Altfel,
        returnăm valoarea fiului pe care am continuat.
14: end function
15:
16: QUERY(1, 1, N, a, b)                        ▷ Apelăm funcția
```

Complexitate. Dacă nu se întâmplă să între vreodată pe ambele ramuri de continuare, complexitatea este analog actualizării. În cazul în care se întâmplă o spargere înseamnă că mijlocul intervalului curent mij se află între capetele subsecvenței căutate $[a, b]$. Astfel, vom continua pe ambii fii, și vom avea ori un prefix al intervalului reprezentat de nod (în cazul fiului drept) ori un sufix al acestuia (în cazul fiului stâng) complet cuprins între a și b. Din acest motiv, orice spargere viitoare va avea cel puțin o ramură a acesteia rezolvată printr-o returnare imediată. Astfel, pe fiecare nivel al arborelui pot să fie vizitate cel mult 4 noduri [12]. Cum arborele are $O(log_2 N)$ nivele, complexitatea timp este $O(log_2 N)$.

De exemplu, interogarea subsecvenței $[5, 9]$ a unui șir de 10 elemente va parcurge următoarele noduri în arborele de intervale:

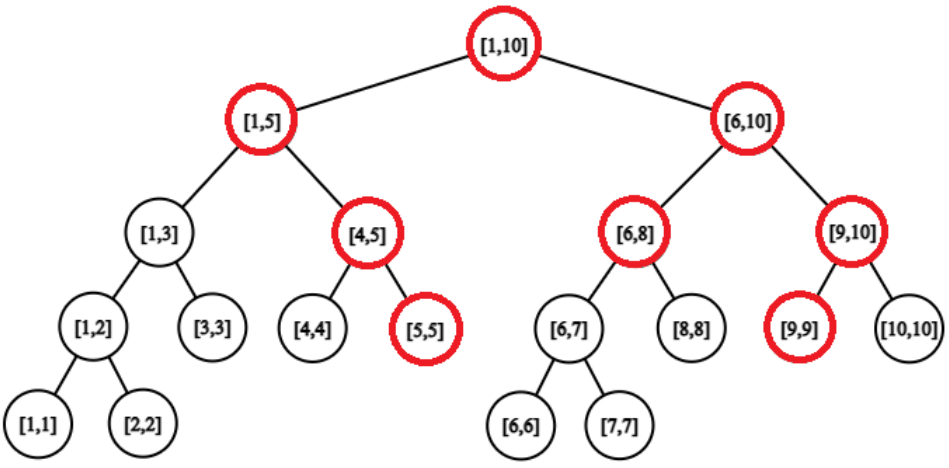

Figura II.2: Interogarea subsecvenței $[5, 9]$

Astfel, prima spargere se va întâmpla chiar în cadrul rădăcinii, continuând pe ambii fii. În acest moment, intervalul $[5, 9]$ poate fi reprezentat ca un sufix al fiului stâng $[1, 5]$ și un prefix al fiului drept $[6, 10]$. Spargerile viitoare au cel puțin una dintre ramuri rezolvate printr-o returnare imediată, acest lucru putând fi observat în cadrul nodului $[6, 8]$, care este complet inclus în intervalul $[5, 9]$.

Căutarea unui prefix în arborele de intervale

Concept. Căutarea unui prefix în arborele de intervale este o operație care poate exista doar dacă sunt respectate anumite condiții. Mai precis, prefixele trebuie să fie monotone în raport cu proprietatea căutată. Vom discuta despre cea mai comună situație în care este folosită această operație:

- Avem un șir de N numere naturale.
- Avem actualizări ale elementelor în șir.
- Avem interogări de tipul "Se dă un X, care este cea mai mică poziție pos astfel încât suma numerelor de la pozițiile 1 până la pos este mai mare ca X?".

Primul pas în rezolvarea acestei probleme este de a ne crea un arbore de intervale în care frunzele acestuia reține elementele șirului, iar restul nodurilor reține suma valorilor din fii acestora. Astfel, permitem actualizarea elementelor din șir în complexitate $O(log_2 N)$, dar putem rezolva și interogările într-un timp optim.

Folosind operația de interogare a unei subsecvențe, putem răspunde în complexitate timp $O((log_2 N)^2)$ pentru fiecare interogare din cerință prin căutarea binară a poziției pos. La fiecare pas în căutarea binară, este folosită operația de interogare a unei subsecvențe pentru aflarea valorii sumei elementelor din subsecvența $[1, pos]$.

Complexitatea poate fi îmbunătățită la $O(log_2 N)$ printr-o coborâre eficientă pe nodurile arborelui de intervale. La fiecare pas, dacă fiul stâng reține o valoare mai mare ca X vom continua pe acesta, altfel vom scădea valoarea fiului stâng din X și vom continua pe fiul drept. Astfel, frunza pe care ajungem va corespunde chiar poziției căutate. Această operație va fi numită căutarea unui prefix în arborele de intervale.

II.3 Tipuri de informații reținute de noduri

Concept. Condiția pentru a putea reține o structură algebrică în arborele de intervale, este ca aceasta să fie semigrup, adică legea de compoziție (combinarea nodurilor) să fie asociativă. În multe cazuri vom lucra chiar cu un monoid, adică având element neutru, dar acest lucru nu este necesar, deoarece mereu putem să ne marcăm printr-un simbol un pseudo-element neutru. În continuare vom prezenta câteva exemple de tipuri de informații reținute în noduri.

Suma elementelor unei subsecvențe

În cazul acesta, arborele de intervale va reține în frunze elementele unui șir (de exemplu numere naturale sau întregi), iar în fiecare nod care nu este frunză se va afla suma valorilor fiilor acestuia.

Algorithm 4 Suma elementelor unei subsecvențe

1: **function** COMBINE($node, left\_child, right\_child$)
2: $\quad tree[node] \leftarrow tree[left\_child] + tree[right\_child]$
3: **end function**

Elementul de valoare minimă/maximă dintr-o subsecvență

Algorithm 5 Elementul de valoare minimă dintr-o subsecvență

1: **function** COMBINE($node, left\_child, right\_child$)
2: $\quad tree[node] \leftarrow min(tree[left\_child], tree[right\_child])$
3: **end function**

Algorithm 6 Elementul de valoare maximă dintr-o subsecvență

1: **function** COMBINE($node, left\_child, right\_child$)
2: $\quad tree[node] \leftarrow max(tree[left\_child], tree[right\_child])$
3: **end function**

Subsecvența de sumă maximă cuprinsă într-un interval de poziții din șir

Fie un șir de N numere întregi, se dorește aflarea subsecvenței de sumă maximă cuprinsă între două poziții a și b.

Pentru a putea afla subsecvența de sumă maximă, fiecare nod va conține 4 valori:

- sum - suma elementelor din interval
- max_pref - cea mai mare sumă a unui prefix din interval
- max_suf - cea mai mare sumă a unui sufix din interval
- $max\_subsecv$ - cea mai mare sumă a unei subsecvențe din interval

Algorithm 7 Subsecvența de sumă maximă cuprinsă într-un interval de poziții din șir

1: **function** COMBINE($node, left\_child, right\_child$)
2: $tree[node].sum \leftarrow tree[left\_child].sum + tree[right\_child].sum$
3: $tree[node].max\_pref \leftarrow max(tree[left\_child].max\_pref,$
$tree[left\_child].sum + tree[right\_child].max\_pref)$
4: $tree[node].max\_suf \leftarrow max(tree[right\_child].max\_suf,$
$tree[right\_child].sum + tree[left\_child].max\_suf)$
5: $tree[node].max\_subsecv \leftarrow max(tree[left\_child].max\_subsecv,$
$tree[right\_child].max\_subsecv,$
$tree[left\_child].max\_suf + tree[right\_child].max\_pref,$
$tree[node].max\_pref,$
$tree[node].max\_suf,$
$tree[node].sum)$
6: **end function**

O aplicație bună pentru acest mod de construcție a arborelui de intervale este reprezentată de problema Maxq [18] din cadrul olimpiadei naționale de informatică, anul 2007.

Compunerea permutărilor aparținând unei subsecvențe

Fiecare element din șir va fi o permutare de lungime K. Astfel, se dorește aflarea compunerii permutărilor asociate unei subsecvențe a șirului.

Deoarece compunerea de permutări este asociativă, aceasta poate fi reținută într-un arbore de intervale. Mai precis, fiecare nod va reprezenta un vector care reține o permutare, iar compunerea nodurilor este chiar compunerea permutărilor.

Algorithm 8 Compunerea permutărilor aparținând unei subsecvențe

1: **function** COMBINE($node, left\_child, right\_child$)
2: $tree[node] \leftarrow$ COMPOSE($tree[left\_child], tree[right\_child]$)
3: **end function**

II.4 Probleme rezolvate

Sereja and Brackets [17]

Enunț. Fie N un număr natural nenul și s un șir de N paranteze deschise și închise. Se dau M interogări de forma $[a_i, b_i]$, delimitând o subsecvență din s. Pentru fiecare astfel de subsecvența, trebuie să aflăm lungimea celui mai lung subșir al acesteia care reprezintă o parantezare corectă.

Definim un subșir al lui t de lungime X ca un șir $x = t_{k_1}t_{k_2}...t_{k_X}$ unde $k_1 < k_2 < ... < k_X$ [17].

Definim o parantezare corectă un șir care poate fi transformat într-o expresie aritmetică adăugând doar '1' și '+'. De exemplu "()()" și "(())" sunt corecte deoarece se pot transforma în "$(1)+(1)$" și "$((1+1)+1)$", dar ")(" nu este corectă [17].

Soluție naivă. Pentru fiecare interogare parcurgem șirul s de la poziția a_i până la poziția b_i. Când găsim o paranteză deschisă o adaugăm într-o stivă, iar când găsim o paranteză închisă, dacă stiva nu este goală, scoatem o paranteză deschisă din stivă și incrementăm răspunsul cu 2. Pentru fiecare interogare parcurgem în cel mai rău caz tot șirul s, deci algoritmul are complexitate timp $O(M \cdot N)$.

Soluție optimă. Pentru a ajunge la soluția optimă se va crea un arbore de intervale in care fiecare nod va reține următoarele valori:

- max_len - lungimea celui mai lung subșir care reprezintă o parantezare corectă din intervalul corespunzător nodului.
- cnt_open - numărul de paranteze deschise care nu fac parte din cel mai lung subșir care reprezintă o parantezare corectă din intervalul corespunzător nodului.
- cnt_closed - numărul de paranteze închise care nu fac parte din cel mai lung subșir care reprezintă o parantezare corectă din intervalul corespunzător nodului.

Nodurile vor fi combinate în următorul mod:

Algorithm 9 Cel mai lung subșir care reprezintă o parantezare corectă

1: **function** COMBINE($node, left\_child, right\_child$)
2: $new\_pairs \leftarrow min(tree[left\_child].cnt\_open, tree[right\_child].cnt\_closed)$
3: $tree[node].max\_len \leftarrow tree[left\_child].max\_len$
$+ tree[right\_child].max\_len$
$+ 2 \cdot new\_pairs$
4: $tree[node].cnt\_open \leftarrow tree[left\_child].cnt\_open$
$+ tree[right\_child].cnt\_open$
$- new\_pairs$
5: $tree[node].cnt\_closed \leftarrow tree[left\_child].cnt\_closed$
$+ tree[right\_child].cnt\_closed$
$- new\_pairs$
6: **end function**

În variabila new_pairs se reține numărul de perechi care se formează prin unirea nodurilor, acestea fiind generate de parantezele deschise nefolosite din fiul stâng și parantezele închise nefolosite din fiul drept. Aceste perechi vor face acum parte din cel mai lung subșir, deci se va adăuga la max_len (fiecare pereche are 2 paranteze, deci se înmulțește cu 2), și se va scădea din cnt_open și cnt_closed.

Arborele de intervale va fi inițializat pe baza șirului s. Frunzele arborelui care corespund unei paranteze deschise din s vor avea cnt_open egal cu 1 și cnt_closed egal cu 0. Celelalte frunze vor corespunde parantezelor deschise din s, acestea având cnt_open egal cu 0 și cnt_closed egal cu 1. Toate frunzele vor avea max_len egal cu 0. Inițializarea arborelui va avea complexitatea $O(N)$.

Astfel, fiecare interogare din cerință poate fi rezolvată printr-o operație de interogare a subsecvenței $[a_i, b_i]$ în arborele de intervale creat în complexitate timp $O(log_2 N)$. Fiind M astfel de interogări, complexitatea timp finală va fi $O(N + M \cdot log_2 N)$.

Calafat [16]

Enunț. Fie N un număr natural nenul și v un șir de N numere naturale cu valorile cuprinse între 1 și N. Se dau M interogări de forma $[a_i, b_i]$, delimitând o subsecvență din v. Pentru fiecare interogare, trebuie să aflăm suma distanţelor corespunzătoare tuturor valorilor distincte din subsecvență.

Distanţa corespunzătoare unei valori se definește ca diferența dintre indicii ultimei și primei apariții a acesteia în subsecvență.

Soluție naivă. Pentru fiecare interogare parcurgem șirul v de la poziția a_i până la poziția b_i. Astfel, putem reține indicele primei apariții și a ultimei apariții pentru fiecare valoare. După ce știm acești indici, doar trebuie să facem suma diferențelor. Pentru fiecare interogare parcurgem în cel mai rău caz tot șirul v, deci algoritmul are complexitate timp $O(M \cdot N)$.

Soluție optimă. Deoarece toate cele M interogări sunt știute de la început, iar șirul v este constant (nu suferă nicio modificare), putem alege ordinea în care dorim să efectuăm rezolvarea acestora, acest mod de rezolvare a interogărilor fiind găsit des în practică sub numele de rezolvare ”offline”. Pentru fiecare poziție pos de la 1 la N ne vom reține toate interogările care au capătul de dreapta b_i egal cu pos.

Interogările vor fi sortate după capătul dreapta al acestora (b_i), fiind rezolvate în această ordine.

Pentru aceasta soluție vom crea un arbore de intervale cu N frunze, acestea având inițial valoarea 0, iar în fiecare nod care nu este frunză se va afla suma valorilor fiilor acestuia.

Parcurgem elementele vectorului v de la stânga la dreapta. Notăm poziția curentă cu pos. Vom reține ultima apariție a fiecărei valori până la poziția curentă într-un vector notat cu $last$. Pentru fiecare poziție curentă vom face următoarele operații:

- Dacă valoarea $v[pos]$ a mai apărut măcar o dată înainte, actualizăm în arbore frunza corespunzătoare poziției $last[v[pos]]$, schimbându-i valoarea în $pos - last[pos]$ și recalculând drumul de la rădăcina până la aceasta.
- Rezolvăm toate interogările cu capătul dreapta b_i egal cu pos. Valoarea fiecărei astfel de interogări va fi egală cu operația de interogare a subsecvenței $[a_i, b_i]$ din arborele de intervale.
- $last[v[pos]]$ ia valoarea pos.

Algoritmul se bazează pe următoarea idee: dacă indicii unde apare o valoare într-o subsecvență sunt $t_1, t_2, ..., t_k$ atunci $t_k - t_1 = \sum_{i=1}^{k-1} t_{i+1} - t_i$. Astfel, se reţin în frunzele corespunzătoare pozițiilor $t_1, t_2, ..., t_{k-1}$ valorile $t_2 - t_1, t_3 - t_2, ...t_k - t_{k-1}$, suma acestor elemente fiind chiar $t_k - t_1$.

Deoarece se vor efectua N actualizări de element în arbore și M interogări de subsecvența, complexitatea timp a soluției este $O((N + M) \cdot log_2 N)$.

III Lazy propagation

III.1 Concept

Definiție. În capitolul anterior am discutat doar actualizarea unui element din șirul s pe baza căruia este construit arborele de intervale, dar acesta permite și actualizarea unei întregi subsecvențe din șirul s, această tehnică fiind numită "Lazy propagation".

Astfel, setul actualizat de operații permise de un arbore de intervale este:

Algoritm	Complexitate Timp
Inițializare	$O(N)$
Actualizarea unui element	$O(log_2 N)$
Actualizarea unei subsecvențe	$O(log_2 N)$
Interogarea unei subsecvențe	$O(log_2 N)$
Căutarea unui prefix	$O(log_2 N)$

Se poate observa faptul că nu am avea cum să facem actualizarea unei subsecvențe în complexitate timp $O(log_2 N)$ ajungând în fiecare frunză din interval deoarece numărul acestora este în cel mai rău caz chiar N. Din acest motiv va trebui să marcăm doar anumite noduri și să propagăm mai jos de acestea doar în cazul în care vom procesa ulterior altă operație (actualizare sau interogare) care ar necesita acest lucru.

Pentru a realiza actualizarea unei subsecvențe $[a, b]$, va trebui să actualizăm un set de noduri din arborele de intervale astfel încât reuniunea intervalelor corespunzătoare acestora sa fie chiar $[a, b]$. Am demonstrat în capitolul anterior, în cadrul prezentării interogării unei subsecvențe, faptul că orice interval poate fi format din $O(log_2 N)$ noduri, deci acestea vor fi alese și pentru actualizarea unei subsecvențe. Așadar, complexitatea timp este $O(log_2 N)$.

Algorithm 10 Propagare [12]

```
1:  function PROPAGATE(node, left, right)
2:      if tree[node].lazy then                          ▷ Dacă nodul este marcat
3:          if left ≠ right then                         ▷ Dacă nodul nu este frunza
4:              tree[node · 2].lazy ← tree[node].lazy    ▷ Marcăm fiul stâng
5:              tree[node · 2 + 1].lazy ← tree[node].lazy ▷ Marcăm fiul drept
6:          end if
7:          Valoarea nodului este actualizată în funcție de tree[node].lazy
8:          tree[node].lazy ← 0                          ▷ Demarcăm nodul
9:      end if
10: end function
```

Algorithm 11 Actualizarea unei subsecvențe [12]

```
1: function UPDATE(node, left, right, a, b, value)
2:     PROPAGATE(node, left, right)        ▷ Propagăm valoare din nod dacă acesta este marcat
3:     if a ≤ left and right ≤ b then      ▷ Am ajuns la un interval cuprins în [a, b]
4:         tree[node].lazy ← value         ▷ Marcăm nodul
5:         return
6:     end if
7:     mid ← (left + right)/2              ▷ Calculăm mijlocul intervalului
8:     if a ≤ mid then
9:         UPDATE(node · 2, left, mid, a, b, value)        ▷ Continuăm pe fiul stâng
10:    end if
11:    if mid + 1 ≤ b then
12:        UPDATE(node · 2 + 1, mid + 1, right, a, b, value)   ▷ Continuăm pe fiul drept
13:    end if
14:    PROPAGATE(node · 2, left, mid)      ▷ Propagăm valoare din fiul stâng
15:    PROPAGATE(node · 2 + 1, mid + 1, right)      ▷ Propagăm valoare din fiul
16:    COMBINE(node, node · 2, node · 2 + 1)   ▷ Combinăm valorile din fiul stâng și drept
17: end function
18: UPDATE(1, 1, N, a, b, value)            ▷ Apelăm funcția
```

Operațiile de actualizarea unui element, interogarea unei subsecvențe și căutare unui prefix se implementează exact ca în capitolul anterior, dar necesită apelarea funcției PROPAGATE la fiecare intrare într-un nod.

III.2 Exemple

Adunarea unei valori elementelor dintr-un interval + Interogări de sumă

Fiecare nod al arborelui de intervale va reține următoarele valori:

- sum - suma elementelor din interval.
- $lazy$ - valoarea care se adună nodurilor din interval sau 0 dacă nu se adună nimic.

Algorithm 12 Propagare

```
1: function PROPAGATE(node, left, right)
2:     if tree[node].lazy then                                   ▷ Dacă nodul este marcat
3:         if left ≠ right then                                  ▷ Dacă nodul nu este frunza
4:             tree[node · 2].lazy += tree[node].lazy            ▷ Marcăm fiul stâng
5:             tree[node · 2 + 1].lazy += tree[node].lazy        ▷ Marcăm fiul drept
6:         end if
7:         tree[node].sum += tree[node].lazy · (right − left + 1)   ▷ Adaugăm la sumă
   valoarea adăugata la fiecare nod din interval
8:         tree[node].lazy ← 0                                   ▷ Demarcăm nodul
9:     end if
10: end function
```

Schimbarea elementelor dintr-un interval într-o valoare strict pozitivă + Interogări de maxim

Fiecare nod al arborelui de intervale va reține următoarele valori:

- max - elementul maxim din interval.
- $lazy$ - valoarea în care se schimbă nodurile din interval sau 0 dacă nu se schimbă.

Algorithm 13 Propagare

```
1: function PROPAGATE(node, left, right)
2:     if tree[node].lazy then                                   ▷ Dacă nodul este marcat
3:         if left ≠ right then                                  ▷ Dacă nodul nu este frunza
4:             tree[node · 2].lazy ← tree[node].lazy             ▷ Marcăm fiul stâng
5:             tree[node · 2 + 1].lazy ← tree[node].lazy         ▷ Marcăm fiul drept
6:         end if
7:         tree[node].max ← tree[node].lazy   ▷ Valoarea maximă este acum cea în care se
   schimbă tot intervalul
8:         tree[node].lazy ← 0                                   ▷ Demarcăm nodul
9:     end if
10: end function
```

III.3 Probleme rezolvate

Hotel [1]

Enunț. Fie un hotel cu N camere, inițial toate fiind goale. Se dau M instrucțiuni de 3 tipuri:

1. Camerele din intervalul $[a_i, b_i]$ se ocupă. Se garantează că intervalul primit va conține doar camere goale.
2. Camerele din intervalul $[a_i, b_i]$ se golesc. Se garantează că intervalul primit va conține doar camere ocupate.
3. Afișează lungimea maximă a unui interval format doar din camere goale.

Soluție naivă. Se rețin camerele ca un vector notat v unde $v[i] = 0$ dacă camera i este goala, iar $v[i] = 1$ dacă aceasta este ocupată. Inițial vectorul v are doar valori de 0, deoarece toate camerele sunt goale.

Astfel, pentru instrucțiuni de tipul 1 și 2 trecem prin tot intervalul și schimbăm valoarea reținută în vector pe acele poziții. Deoarece în cel mai rău caz se parcurg toate camerele, complexitatea timp pentru instrucțiunile de tipul 1 sau 2 este $O(N)$.

Pentru instrucțiunile de tipul 3, inițializăm două variabile notate cu max_count și $count$ cu valoarea 0 și se parcurge tot vectorul v de la poziția 1 la poziția N. La fiecare poziție i, dacă $v[i] = 0$ atunci se adună 1 la $count$, altfel $count$ devine 0. După schimbarea variabilei $count$, variabila max_count se actualizează cu valoarea maximă dintre valoarea ei și valoarea lui $count$. La finalul parcurgerii, lungimea maximă a unui interval format doar din camere goale se va afla în variabila max_count. Complexitatea timp pentru instrucțiunile de tipul 3 este $O(N)$.

Cum orice instrucțiune are complexitate timp $O(N)$, atunci complexitatea timp finală va fi $O(M \cdot N)$.

Soluție optimă. Pentru o soluție eficientă, vom folosi un arbore de intervale. Fiecare nod al arborelui de intervale va reține următoarele valori:

- max_len - lungimea maximă a unui interval format doar din camere goale cuprins în intervalul nodului.
- $pref\_len$ - lungimea celui mai mare prefix format doar din camere goale.
- suf_len - lungimea celui mai mare sufix format doar din camere goale.
- $lazy$ - egal cu 0 dacă nodul nu este marcat,
  egal cu 1 dacă camerele din întregul interval se ocupă,
  egal cu -1 dacă camerele din întregul interval se golesc.

Arborele de intervale va avea N frunze, fiecare corespunzând câte unei camere de hotel. Aceste sunt inițializate cu $max\_len = 1, pref\_len = 1, suf\_len = 1, lazy = 0$ deoarece inițial toate camerele sunt goale.

Deoarece instrucțiunea de tipul 3 se raportează la întregul interval $[1, N]$, va trebui implementată doar operația de actualizare a unei subsecvențe, răspunsul pentru instrucțiunea de tipul 3 aflându-se chiar în rădăcina arborelui, fiind valoarea max_len corespunzătoare acesteia. Greutatea problemei reprezintă implementarea funcțiilor COMBINE și PROPAGATE:

Algorithm 14 Combinare

function COMBINE($node, left\_child, right\_child, left, right, mid$)
    if $tree[left\_child].max\_len = mid - left + 1$ **then** ▷ Fiul stâng complet gol
        $tree[node].pref\_len \leftarrow (mid - left + 1) + tree[right\_child].pref\_len$
    else
        $tree[node].pref\_len \leftarrow tree[left\_child].pref\_len$
    end if

    if $tree[right\_child].max\_len = right - (mid + 1) + 1$ **then** ▷ Fiul drept complet gol
        $tree[node].suf\_len \leftarrow (right - (mid + 1) + 1) + tree[leftt\_child].suf\_len$
    else
        $tree[node].suf\_len \leftarrow tree[right\_child].suf\_len$
    end if

    $tree[node].max\_len \leftarrow max(tree[left\_child].max\_len,$
                        $tree[right\_child].max\_len,$
                        $tree[left\_child].suf\_len + tree[right\_child].pref\_len)$
end function

Se poate observa că în cazul acesta, funcția de combinare ia ca parametri în plus și $left$, $right$ și mid pentru a putea determina dacă fiul stâng sau fiul drept conțin doar camere goale.

Algorithm 15 Propagare

```
1:  function PROPAGATE(node, left, right)
2:      if tree[node].lazy then                          ▷ Dacă nodul este marcat
3:          if left ≠ right then                         ▷ Dacă nodul nu este frunza
4:              tree[node · 2].lazy ← tree[node].lazy    ▷ Marcăm fiul stâng
5:              tree[node · 2 + 1].lazy ← tree[node].lazy ▷ Marcăm fiul drept
6:          end if
7:          if tree[node].lazy = 1 then                  ▷ Toate camerele din intervalul se ocupă
8:              tree[node].max_len ← 0
9:              tree[node].pref_len ← 0
10:             tree[node].suf_len ← 0
11:         end if
12:         if tree[node].lazy = −1 then                 ▷ Toate camerele din intervalul se golesc
13:             tree[node].max_len ← right − left + 1
14:             tree[node].pref_len ← right − left + 1
15:             tree[node].suf_len ← right − left + 1
16:         end if
17:         tree[node].lazy ← 0                          ▷ Demarcăm nodul
18:     end if
19: end function
```

Pentru instrucțiunile de tipul 1 și 2 vom actualiza subsecvența $[a_i, b_i]$, marcând nodurile cu valoarea 1 în cazul tipului 1, respectiv cu valoarea -1 în cazul tipului 2. Complexitate timp $O(log_2 N)$.

Pentru instrucțiunile de tipul 3, răspunsul se va afla chiar în $tree[1].max\_len$. Complexitate timp $O(1)$.

Complexitatea timp a inițializării arborelui este $O(N)$, iar fiecare dintre cele M instrucțiuni se realizează în $O(log_2 N)$. Deci complexitatea timp finală este $O(N + M \cdot log_2 N)$.

A Simple Task [15]

Enunț. Fie N un număr natural nenul și s un șir de N litere mici ale alfabetului latin (a, b, ..., z). Se dau M instrucțiuni de 2 tipuri:

1. Literele din intervalul $[a_i, b_i]$ se sortează crescător.
2. Literele din intervalul $[a_i, b_i]$ se sortează descrescător.

La finalul procesării instrucțiunilor, se afișează șirul s.

În explicația soluțiilor și a complexității, vom nota alfabetul latin ca σ și la numărul de litere mici din alfabetul latin ca $|\sigma|$ (egal cu 26).

Soluție naivă. Pentru fiecare instrucțiune facem sortarea. Deoarece sunt puține litere diferite, putem face sortarea prin numărare (Counting Sort). Astfel, avem $O(N + |\sigma|)$ complexitatea fiecărei instrucțiuni, deci $O(M \cdot (N + |\sigma|))$ complexitate timp finală.

Soluție optimă. Pentru soluția optimă, vom crea $|\sigma|$ arbori de intervale, câte unul pentru fiecare literă din alfabet. În continuare, vom nota arborele de intervale corespunzător unei litere j ca arborele j.

Toți arborii de intervale vor urma aceeași structură. Fiecare nod al arborelui j va reține următoarele valori:

- $count$ - numărul de apariții a literei j în intervalul corespunzător nodului.
- $lazy$ - egal cu 0 dacă nodul nu este marcat,
  egal cu 1 dacă intervalul corespunzător nodului acum conține doar litera j,
  egal cu -1 dacă intervalul corespunzător nodului acum nu conține nicio apariție a literei j.

Cei $|\sigma|$ arbori de intervale sunt construiți pe baza șirului s, fiecare frunză va corespunde unui element aflat pe poziția pos în șirul s. Astfel, pentru un arbore j, frunzele acestuia se vor inițializa cu $count = 0$ dacă $s[pos] \neq j$, respectiv $count = 1$ dacă $s[pos] = j$. Inițial $lazy$ este egal cu 0 pentru toate nodurile arborilor de intervale. Inițializarea va avea complexitate timp $O(|\sigma| \cdot N)$.

Fiecare arbore j va utiliza următoarele funcții de COMBINE și PROPAGATE:

Algorithm 16 Combinare

```
1: function COMBINE(node, left_child, right_child)
2:     tree[node].count ← tree[left_child].count + tree[right_child].count
3: end function
```

Algorithm 17 Propagare

```
1: function PROPAGATE(node, left, right)
2:     if tree[node].lazy then                                   ▷ Dacă nodul este marcat
3:         if left ≠ right then                                  ▷ Dacă nodul nu este frunza
4:             tree[node · 2].lazy ← tree[node].lazy             ▷ Marcăm fiul stâng
5:             tree[node · 2 + 1].lazy ← tree[node].lazy         ▷ Marcăm fiul drept
6:         end if
7:         if tree[node].lazy = 1 then    ▷ Acum intervalul corespunzător nodului va conține
   doar litera asociată arborelui
8:             tree[node].count ← right − left + 1
9:         end if
10:        if tree[node].lazy = −1 then   ▷ Acum intervalul corespunzător nodului nu va
   conține nicio apariție a literei asociată arborelui
11:            tree[node].count ← 0
12:        end if
13:        tree[node].lazy ← 0                                   ▷ Demarcăm nodul
14:    end if
15: end function
```

Logica din spatele soluție este să aflăm frecvența fiecărei litere în subsecvența $[a_i, b_i]$ asociată unei instrucțiuni prin câte o interogare de subsecvență în arborele de intervale asociat literei. Având aceste frecvențe, putem să aplicăm sortarea prin numărare (crescător sau descrescător) și să generăm maxim $|\sigma|$ intervale compacte care vor conține doar o anumită literă repetată. De exemplu, după efectuarea unei instrucțiuni de tipul 1, în subsecvența generată va fi mai întâi un interval doar cu litere de a, după unul doar cu litere de b și tot așa.

Astfel, fiecare instrucțiune va fi rezolvată în următorul mod:

- Se găsește frecvența fiecărei litere în subsecvența $[a_i, b_i]$.
- Vom actualiza subsecvența $[a_i, b_i]$ în toți cei $|\sigma|$ arbori, marcand nodurile cu valoarea -1 pentru a goli subsecvența de orice literă.
- Calculăm intervalele compacte care vor conține doar o anumită literă.
- Pentru fiecare astfel de interval, în arborele asociat acestei litere, se va actualiza subsecvența asociată intervalului, marcând nodurile cu valoarea 1.

După efectuarea instrucțiunilor, vom propaga nodurile tuturor arborilor până la frunze. Într-un arbore j, șirul s va avea litera j pe pozițiile asociate frunzelor acestui arbore care au valoare $count$ egală cu 1.

Fiecare instrucțiune are complexitate timp $O(|\sigma| \cdot log_2 N)$ deoarece, pentru fiecare dintre cei $|\sigma|$ arbori de intervale, se efectuează o operație de interogare a unei subsecvențe și două operații de actualizare a unei subsecvențe. Astfel, complexitatea timp finală este $O(|\sigma| \cdot N + M \cdot |\sigma| \cdot log_2 N)$.

IV Persistență

IV.1 Concept

Definiție. Persistența structurilor de date a fost formalizată pentru prima dată de către Driscoll et al. în lucrarea ”Making data structures persistent” [4]. O structură de date persistentă reprezintă o structură de date care poate să își rețină o copie a stării acesteia după fiecare actualizare. Astfel, aceasta permite următoarele operații:

- Interogarea unei stări (după o anumită actualizare).
- Operația de ”undo” (anulăm ultima actualizare).
- Operația de ”rebase” (ne întoarcem la o anumită actualizare, anulând toate actualizările după aceasta).

O abordare trivială pentru a face un arbore de intervale persistent ar fi ca, după fiecare actualizare, să reținem în memorie o copie a întregului arbore. Dar această abordare ar fi complet ineficientă, deoarece ar fi inutil să avem o actualizare în complexitate timp $O(log_2 N)$ dacă după fiecare actualizare oricum se copiază arborele în complexitate timp $O(N)$. În plus, trebuie luată în calcul și complexitatea spațiu, deoarece pentru a reține toate copiile în acest mod ar fi necesară $O(M \cdot N)$ memorie, unde M reprezintă numărul de actualizări.

Din acest motiv, este necesară o abordare mai bună atât ca timp cât și ca memorie. Deoarece actualizarea unei element afectează doar lanțul până la frunza corespunzătoare poziției acestuia, sunt $O(log_2 N)$ noduri schimbate. Astfel, se poate reține în memorie doar câte o copie a acestor noduri, restul rămânând neschimbate. Această abordare este optimă atât din punct de vedere al timpului ($O(log_2 N)$ operații de copiere), cât și din punct de vedere al memoriei (fiecare actualizare acum aduce doar $O(log_2 N)$ noduri noi). Același lucru se poate aplica și în cazul actualizării de interval, fiind demonstrat în capitolul anterior că se vor schimba $O(log_2 N)$ noduri.

Pentru a permite copierea nodurilor în memorie, va fi necesară schimbarea modului în care este construit arborele de intervale. Până acum am reținut arborele într-un vector unde, pentru un nod aflat la indicele $node$, fiul acestuia stâng era la indicele $node \cdot 2$, iar fiul drept la indicele $node \cdot 2 + 1$. Acum va trebui să implementăm totul dinamic, adică fiecare nod va reține doi pointeri către fiul stâng, respectiv fiul drept.

Implementare nod. Fiecare nod va reține atât informația lui asupra intervalului, cat și doi pointeri către fiul stâng și fiul drept, notați cu $left\_child$, respectiv $right\_child$. Astfel, fiecare nod va fi o instanță a tipului de date notat cu $Node$. Crearea unui nou nod va fi notată cu $new\ Node()$.

Implementare rădăcini. Pentru a putea interoga o stare anterioară, vom reține un vector de pointeri către fiecare rădăcină a arborelui după o actualizare, notat cu $root$. Practic, $root[i]$ reprezintă rădăcina arborelui după actualizarea i, iar $root[0]$ reprezintă rădăcina arborelui înainte de orice actualizare (adică exact după inițializare).

IV.2 Operații

Arborele de date persistent permite atât operațiile de baza ale unui arbore de intervale, cat și operațiile unei structuri de date persistente.

Algoritm	Complexitate Timp	Complexitate Spațiu
Inițializare	$O(N)$	$O(N)$
Actualizarea unui element	$O(log_2 N)$	$O(log_2 N)$
Actualizarea unei subsecvențe	$O(log_2 N)$	$O(log_2 N)$
Interogarea unei subsecvențe într-o stare anterioară	$O(log_2 N)$	$O(1)$
UNDO (anulăm ultima actualizare)	$O(1)$	$O(1)$
REBASE (ne întoarcem la o stare a arborelui)	$O(1)$	$O(1)$

Inițializare

Înainte de toate va trebui inițializat arborele. Primul pas este să inițializăm $root[0]$, după inițializăm recursiv restul arborelui. Funcția INIT va primi ca parametri:

- $node$ - pointer către nodul curent.
- $left$ - capătul stâng al intervalului.
- $right$ - capătul drept al intervalului.

În pseudocodul de mai jos, este prezentată inițializarea dintr-un vector v, dar acesta se poate inițializa și cu un element neutru dacă nu există o stare de început (de exemplu totul cu 0 pentru a reține suma elementelor dintr-un interval).

Algorithm 18 Inițializare arbore de intervale persistent [12]

```
1: function INIT(node, left, right)
2:     if left = right then                                   ▷ Am ajuns la o frunză
3:         node → val ← v[left]                               ▷ Frunza primește valoarea din v
4:         return
5:     end if
6:     mid ← (left + right)/2                                 ▷ Calculăm mijlocul intervalului
7:     node → left_child ← new Node()                         ▷ Inițializăm fiul stâng
8:     node → right_child ← new Node()                        ▷ Inițializăm fiul drept
9:     INIT(node → left_child, left, mid)                     ▷ Continuăm pe fiul stâng
10:    INIT(node → right_child, mid + 1, right)               ▷ Continuăm pe fiul drept
11:
12:    COMBINE(node, node → left_child, node → right_child)   ▷ Combinăm valorile din
                                                              fiul stâng și drept în nodul curent
13: end function
14:
15: root[0] ← new Node()                                      ▷ Inițializăm rădăcina originala
16: INIT(root[0], 1, N)   ▷ Apelăm funcția cu nodul rădăcină root[0] și întregul interval [1, N]
```

Actualizarea unui element

Pentru a face actualizarea unui element, mai întâi se copiază rădăcina din care dorim sa pornim. Notăm $curr\_root$ indicele rădăcinii curente la care ne aflăm, acesta fiind egal cu numărul de actualizări efectuate până în acest moment. Actualizarea se va face la fel ca în cazul arborelui de intervale normal, dar acum drumul către elementul actualizat este copiat pentru a nu afecta starea trecută a arborelui. Funcția UPDATE va primi ca parametri:

- $node$ - pointer către nodul curent.
- $left$ - capătul stâng al intervalului.
- $right$ - capătul drept al intervalului.
- pos - poziția care se actualizează.
- val - valoarea cu care se actualizează.

Algorithm 19 Actualizarea unui element în arborele de intervale persistent [12]

```
1: function UPDATE(node, left, right, pos, val)
2:     if left = right then                          ▷ Am ajuns la frunza corespunzătoare poziției
3:         node → val ← val                          ▷ Frunza primește valoarea nouă
4:         return
5:     end if
6:     mid ← (left + right)/2                        ▷ Calculăm mijlocul intervalului
7:     if pos ≤ mid then
8:         node → left_child ← copy(node → left_child)   ▷ Copiem datele din fiul stâng
9:         UPDATE(node → left_child, left, mid, pos, val)   ▷ Continuăm pe fiul stâng
10:    else
11:        node → right_child ← copy(node → right_child)  ▷ Copiem datele din fiul drept
12:        UPDATE(node → right_child, mid + 1, right, pos, val)   ▷ Continuăm pe fiul drept
13:    end if
14:
15:    COMBINE(node, node → left_child, node → right_child)   ▷ Combinăm valorile din fiul stâng și drept în nodul curent
16: end function
17:
18: curr_root ← curr_root + 1                        ▷ Rădăcina curentă acum este curr_root + 1
19: root[curr_root] ← copy(root[curr_root − 1])      ▷ Copiem datele din rădăcina anterioară
20: UPDATE(root[curr_root], 1, N, pos, val)          ▷ Apelăm funcția
```

Actualizarea unei subsecvențe

La fel ca în cazul actualizării unei poziții, mai întâi se copiază rădăcina anterioara. De asemenea, toate nodurile în care vom ajunge pentru actualizarea intervalului vor fi copiate pentru a nu afecta starea trecută a arborelui. În plus, faptul că trebuie să și propagăm valori sporește dificultatea implementării operației. Dacă ajungem la un nod care conține o valoare de $lazy$ care trebuie propagată mai jos, înseamna că acesta a fost schimbat într-o actualizare anterioară, când acesta a fost copiat, dar fii acestuia nu au fost. Din acest motiv, nu putem să facem o propagare normală, fii acestuia făcând parte dintr-o stare anterioară a arborelui. Deci va trebui să copiăm nodul și să copiem și fii acestuia înainte de a face propagarea.

Pentru a clarifica acest aspect vom oferi un exemplu. Să presupunem că aceasta este forma inițiala a arborelui de intervale:

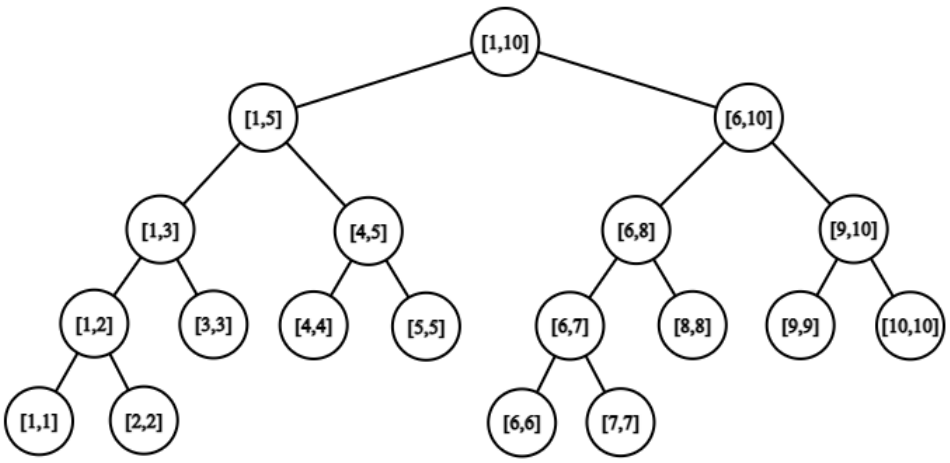

Figura IV.1: Actualizare interval în arbore persistent - Forma inițială

După o actualizare a subsecvenței $[4, 8]$, nodurile marcate în figura de mai jos se vor copia, iar nodurile corespunzătoare intervalelor $[4, 5]$ și $[6, 8]$ acum vor conține valori de $lazy$.

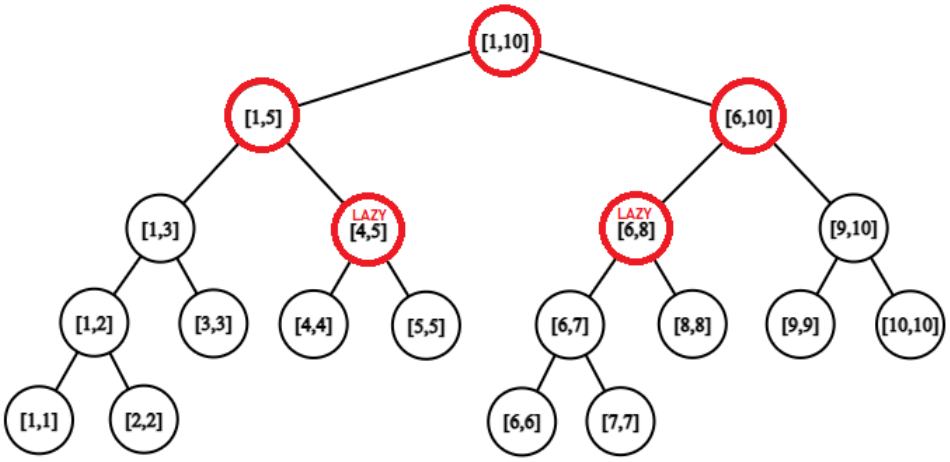

Figura IV.2: Actualizare interval în arbore persistent - Actualizare $[4, 8]$

În acest moment, nodurile corespunzătoare intervalelor $[4, 4]$ și $[5, 5]$ nu au fost copiate după actualizare, deci nodul $[4, 5]$ acum copiat are ca fii 2 noduri din starea inițială a arborelui. Din acest motiv nu putem sa propagăm valoarea din nod, pentru că am modifica starea inițială.

Dacă următoarea actualizare corespunde subsecvenței $[3, 4]$, următoarele noduri vor fi copiate:

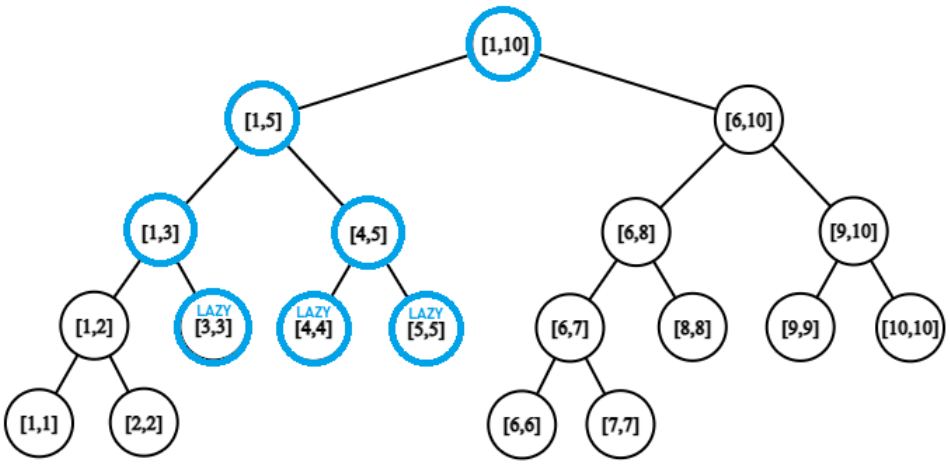

Figura IV.3: Actualizare interval în arbore persistent - Actualizare $[3, 4]$

După cum se observă din imagine, acum și nodul corespunzător intervalului $[5, 5]$ a fost copiat. Dar acest nod nu face parte din drumul până la nodurile $[3, 3]$ și $[4, 4]$ care formează intervalul actualizat $[3, 4]$. Deoarece la actualizarea anterioară am ajuns să avem o valoare de $lazy$ în nodul corespunzător intervalului $[4, 5]$, la actualizarea curentă a trebuit să ii copiem amândoi fii, $[4, 4]$ și $[5, 5]$.

În pseudocodul de mai jos este prezentată actualizarea unui subsecvențe în arborele de intervale persistent. Funcția UPDATE va primi ca parametri:

- $node$ - pointer către nodul curent.
- $left$ - capătul stâng al intervalului corespunzător nodului.
- $right$ - capătul drept al intervalului corespunzător nodului.
- a - capătul de stânga al intervalului actualizat.
- b - capătul de dreapta al intervalului actualizat.
- val - valoarea cu care se actualizează.

Algorithm 20 Actualizare unei subsecvențe în arborele de intervale persistent

```
1: function UPDATE(node, left, right, a, b, val)
2:     already_copied ← false
3:     if node → lazy then
4:         already_copied ← true
5:         node → left_child ← copy(node → left_child)      ▷ Copiem datele din fiul stâng
6:         node → right_child ← copy(node → right_child)    ▷ Copiem datele din fiul drept
7:         PROPAGATE(node, left, right)          ▷ Propagăm valoare din nod dacă acesta este
   marcat
8:     end if
9:
10:    if a ≤ left and right ≤ b then              ▷ Am ajuns la un interval cuprins în [a, b]
11:        node → lazy ← val                                              ▷ Marcăm nodul
12:        return
13:    end if
14:    mid ← (left + right)/2                          ▷ Calculăm mijlocul intervalului
15:    if a ≤ mid then
16:        if !already_copied then
17:            node → left_child ← copy(node → left_child)
18:        end if
19:        UPDATE(node → left_child, left, mid, a, b, val)                ▷ Continuăm
                                                                          pe fiul stâng
20:    end if
21:
22:    if mid + 1 ≤ b then
23:        if !already_copied then
24:            node → right_child ← copy(node → right_child)
25:        end if
26:        UPDATE(node → right_child, mid + 1, right, a, b, val)          ▷ Continuăm
                                                                          pe fiul drept
27:    end if
28:
29:    COMBINE(node, node → left_child, node → right_child)  ▷ Combinăm valorile din
                                                          fiul stâng și drept în nodul curent
30: end function
31:
32: curr_root ← curr_root + 1                  ▷ Rădăcina curentă acum este curr_root + 1
33: root[curr_root] ← copy(root[curr_root − 1])    ▷ Copiem datele din rădăcina anterioară
34: UPDATE(root[curr_root], 1, N, a, b, val)                              ▷ Apelăm funcția
```

Interogarea unei subsecvențe într-o stare anterioară .

Deoarece este reținut vectorul $root$, va trebui doar să facem interogarea subsecvenței pornind de la rădăcina corespunzătoare stării dorite. Interogarea se implementează similar cu interogarea într-un arbore de intervale normal. Singurul lucru care trebuie tratat special este cazul în care arborele persistent permite și actualizarea unei subsecvențe. În cazul acesta nu este permisa

propagarea directă în interogare, valorile de $lazy$ întâlnite fiind reținute separat.

UNDO (anulăm ultima actualizare).

Pentru a anula ultima actualizare va trebui doar să ignorăm ultima rădăcină din vectorul $root$.

REBASE (ne întoarcem la o stare a arborelui).

Pentru a ne întoarce la o stare a arborelui asociată unei actualizări anterioare, va trebui doar să ignorăm toate rădăcinile din vectorul $root$ care vin după actualizarea dorită.

IV.3 Probleme rezolvate

Ants [10]

Enunț. Fie N un număr natural nenul și v un șir de N numere naturale. Aceasta este configurație inițiala a lui v, notată ca configurația 0. Se dau M instrucțiuni de tipul P, X, Y, V, L, R unde:

1. Se creează o nouă configurația a lui v ca o copie a configurației P.
2. Pe această nouă configurație se adaugă valoarea V tuturor numerelor de pe pozițiile din intervalul $[X, Y]$.
3. După adăugarea valorii V, se cere afișarea sumei numerelor de pe pozițiile din intervalul $[L, R]$.
4. Instrucțiunile sunt date astfel încât să fie necesară rezolvarea acestora ”online”. Adică trebuie rezolvată instrucțiunea curentă înainte de a putea citi următoarea instrucțiune.

Soluție naivă. O soluție naivă ar fi să reținem în memorie o copie a șirului v după fiecare instrucțiune și să parcurgem șirul copiat pentru a adăuga valoarea V și a calcula suma. Aceasta rezolvare nu doar va avea complexitatea timp egală cu $O(N \cdot M)$, dar și complexitatea spațiu este la fel $O(N \cdot M)$.

Soluție optimă. Pentru a realiza instrucțiunile într-un mod atât eficient ca timp, cat și ca spațiu, vom utiliza un arbore de intervale persistent. Fiecare nod al arborelui de intervale persistent va reține următoarele valori:

- $left\_child$ - pointer către fiul stâng al acestuia.
- $right\_child$ - pointer către fiul drept al acestuia.
- sum - suma elementelor din interval.
- $lazy$ - valoarea care se adună nodurilor din interval sau 0 dacă nu se adună nimic.

Arborele va fi inițializat pe baza configurației 0 a șirului v, astfel fiecare frunză a acestuia corespunzătoare poziției pos va avea valoarea sum egală cu $v[pos]$.

Combinarea nodurilor se realizează în următorul mod:

Algorithm 21 Combinare

```
1: function COMBINE(node, left_child, right_child)
2:     node → sum ← left_child → sum + right_child → sum
3: end function
```

Propagarea valorii de $lazy$ va urma următoarea regulă:

Algorithm 22 Propagare

1: **function** PROPAGATE($node, left, right$)
2: **if** $node \to lazy$ **then** ▷ Dacă nodul este marcat
3: **if** $left \neq right$ **then** ▷ Dacă nodul nu este frunza
4: $node \to left\_child \to lazy \mathrel{+}= node \to lazy$ ▷ Marcăm fiul stâng
5: $node \to right\_child \to lazy \mathrel{+}= node \to lazy$ ▷ Marcăm fiul drept
6: **end if**
7: $node \to sum \mathrel{+}= node \to lazy \cdot (right - left + 1)$ ▷ Adaugăm la sumă valoarea adăugata la fiecare nod din interval
8: $node \to lazy \leftarrow 0$ ▷ Demarcăm nodul
9: **end if**
10: **end function**

Astfel, pentru fiecare instrucțiune se va genera o nouă stare a arborelui de intervale persistent pornind de la starea P.

Algorithm 23 Generarea unei stări noi

1: $curr\_root \leftarrow curr\_root + 1$ ▷ Rădăcina curentă acum este $curr\_root + 1$
2: $root[curr\_root] \leftarrow copy(root[P])$ ▷ Copiem datele din rădăcina P

Se va actualiza subsecvența $[X, Y]$ în starea curentă, adăugându-se V tuturor valorilor de $lazy$ reținute în nodurile ale căror intervale formează subsecvența, nodurile parcurse în actualizare fiind copiate. După realizarea operației de actualizare, se va interoga suma elementelor din subsecvența $[L, R]$ pentru starea curentă, acesta fiind răspunsul pentru instrucțiunea curentă.

Fiecare dintre cele M instrucțiuni se va realiza în $O(log_2 N)$ complexitate timp și $O(log_2 N)$ spațiu adițional. Luând în calcul și inițializarea arborelui, complexitatea timp finală va fi $O(N + M \cdot log_2 N)$ și complexitatea spațiu finală va fi $O(N + M \cdot log_2 N)$.

V Arbori de intervale cu mai multe dimensiuni (2D)

V.1 Concept

Definiție. Arborele de intervale poate fi extins la mai multe dimensiuni, cel mai comun fiind cel în 2 dimensiuni (2D). În cazul acesta, în loc să se rețină intervalele unui șir, acum în noduri se vor reține informații referitoare la submatricile unei matrici.

Problema de bază. Fie N și M două numere naturale nenule și mat o matrice de N linii și M coloane ce conține numere naturale. Se dau instrucțiuni de 2 tipuri:

- Actualizarea elementului de la poziția $mat[i][j]$ cu o nouă valoare val.
- Interogarea sumei elementelor din sumbatricea $mat[x_1...x_2][y_1...y_2]$.

O submatrice poate fi vizualizată ca un interval de coloane dintr-un interval de linii. Deci putem face un arbore de intervale ale cărui noduri rețin un interval de linii, iar în fiecare astfel de nod se află cate un arbore de intervale care reține intervale de coloane. [19]

V.2 Operații

Algoritm	Complexitate Timp
Inițializare	$O(N \cdot M)$
Actualizarea unui element	$O(log_2 N \cdot log_2 M)$
Interogarea unei submatrice	$O(log_2 N \cdot log_2 M)$

Inițializarea unui arbore de intervale 2D pe baza unei matrice

Concept. Avem un o matrice mat cu liniile indexate de la 1 la N și coloanele indexate de la 1 la M. Vrem să inițializăm valorile din arborele de intervale 2D pe baza acesteia.

Vom reține arborele ca o matrice notată $tree$ de $4 \cdot N$ linii și $4 \cdot M$ coloane.

Algorithm 24 Inițializarea unui arbore de intervale 2D [12]

```
1: function INIT_Y(node_x, left_x, right_x, node_y, left_y, right_y)
2:     if left_y = right_y then                          ▷ Am ajuns la o frunză
3:         if left_x = right_x then
4:             tree[node_x][node_y] ← mat[left_x][left_y]   ▷ Frunza primește valoarea din mat
5:         else
6:             tree[node_x][node_y] ← tree[node_x·2][node_y]+tree[node_x·2+1][node_y]
7:         end if
8:         return
9:     end if
10:    mid ← (left_y + right_y)/2                       ▷ Calculăm mijlocul intervalului
11:    INIT_Y(node_x, left_x, right_x, node_y · 2, left_y, mid)   ▷ Continuăm pe fiul stâng
12:    INIT_Y(node_x, left_x, right_x, node_y · 2 + 1, mid + 1, right_y)   ▷ Continuăm pe fiul drept
13:
14:    tree[node_x][node_y] ← tree[node_x][node_y · 2] + tree[node_x][node_y · 2 + 1]   ▷ Combinăm valorile din fiul stâng și drept în nodul curent
15: end function
16:
17: function INIT_X(node_x, left_x, right_x)
18:    if left_x = right_x then                         ▷ Am ajuns la o frunză
19:        INIT_Y(node_x, left_x, right_x, 1, 1, M)
20:        return
21:    end if
22:    mid ← (left_y + right_y)/2                       ▷ Calculăm mijlocul intervalului
23:    INIT_X(node_x · 2, left_x, mid)                  ▷ Continuăm pe fiul stâng
24:    INIT_X(node_x · 2 + 1, mid + 1, right_x)         ▷ Continuăm pe fiul drept
25:    INIT_Y(node_x, left_x, right_x, 1, 1, M)
26: end function
27:
28: INIT_X(1, 1, N)          ▷ Apelăm funcția cu nodul rădăcina 1 și întregul interval [1, N]
```

Complexitate. Vom trece prin fiecare nod al arborelui de intervale, deci $O(N \cdot M)$ complexitate timp.

Actualizarea unui element în arborele de intervale 2D

Concept. În cazul arborelui de intervale 2D, poziția unui element va fi caracterizată de linia și coloana acesteia, notate cu pos_x, respectiv pos_y. Pseudocodul actualizării va arata similar cu cel al inițializării, doar că acum la fiecare pas mergem doar pe un singur fiu. Din fiecare nod aflat pe drumul spre frunza corespunzătoare lui pos_x, vom apela UPDATE_Y pentru a recalcula valorile intervalelor care îl cuprind pe pos_y.

Algorithm 25 Actualizarea unui element în arborele de intervale 2D [12]

```
1: function UPDATE_Y(node_x, left_x, right_x, node_y, left_y, right_y, pos_y, val)
2:     if left_y = right_y then                          ▷ Am ajuns la o frunză
3:         if left_x = right_x then
4:             tree[node_x][node_y] ← val                ▷ Frunza primește valoarea val
5:         else
6:             tree[node_x][node_y] ← tree[node_x·2][node_y] + tree[node_x·2+1][node_y]
7:         end if
8:         return
9:     end if
10:    mid ← (left_y + right_y)/2                        ▷ Calculăm mijlocul intervalului
11:    if pos_x ≤ mid then
12:        UPDATE_Y(node_x, left_x, right_x,
                    node_y · 2, left_y, mid, pos_y, val)
13:    else
14:        UPDATE_Y(node_x, left_x, right_x,
                    node_y · 2 + 1, mid + 1, right_y, pos_y, val)
15:    end if
16:    tree[node_x][node_y] ← tree[node_x][node_y · 2] + tree[node_x][node_y · 2 + 1]
17: end function
18:
19: function UPDATE_X(node_x, left_x, right_x, pos_x, pos_y, val)
20:    if left_x = right_x then                          ▷ Am ajuns la o frunză
21:        INIT_Y(node_x, left_x, right_x, 1, 1, M, pos_y, val)
22:        return
23:    end if
24:    mid ← (left_y + right_y)/2                        ▷ Calculăm mijlocul intervalului
25:    if pos_x ≤ mid then
26:        UPDATE_X(node_x · 2, left_x, mid, pos_x, pos_y, val)
27:    else
28:        UPDATE_X(node_x · 2 + 1, mid + 1, right_x, pos_x, pos_y, val)
29:    end if
30:    UPDATE_Y(node_x, left_x, right_x, 1, 1, M, pos_y, val)
31: end function
32:
33: UPDATE_X(1, 1, N, pos_x, pos_y, val)                ▷ Apelăm funcția cu nodul rădăcină 1
```

Complexitate. Vom coborî pe drumul de la rădăcina arborelui spre frunza asociată poziției pos_x. Deoarece la fiecare pas, intervalul curent se înjumătățește, avem $O(log_2 N)$ apelări a funcției UPDATE_Y. La fiecare apelare, funcția UPDATE_Y va coborî până la frunza asociată poziției pos_y în același mod. Astfel, complexitatea timp este $O(log_2 N \cdot log_2 M)$.

Interogarea unei submatrici în arborele de intervale 2D

Concept. O submatrice va fi caracterizata de (x_1, y_1) și (x_2, y_2), marcând poziția colțului stânga sus a submatricei, respectiv poziția colțului dreapta jos a acesteia.

Algorithm 26 Interogarea unei submatrici în arborele de intervale 2D [12]

```
1: function QUERY_Y(node_x, node_y, left, right, y1, y2)
2:     if y1 ≤ left and right ≤ y2 then          ▷ Am ajuns la un interval cuprins în [y1, y2]
3:         return tree[node_x][node_y]
4:     end if
5:     mid ← (left + right)/2                    ▷ Calculăm mijlocul intervalului
6:     ret ← 0
7:     if y1 ≤ mid then
8:         ret += QUERY_Y(node_x, node_y · 2, left, mid, y1, y2)
9:     end if
10:    if mid + 1 ≤ y2 then
11:        ret += QUERY_Y(node_x, node_y · 2 + 1, mid + 1, right, y1, y2)
12:    end if
13:    return ret
14: end function
15:
16: function QUERY_X(node_x, left, right, x1, x2, y1, y2)
17:    if x1 ≤ left and right ≤ x2 then          ▷ Am ajuns la un interval cuprins în [x1, x2]
18:        return QUERY_Y(node_x, 1, 1, M, y1, y2)
19:    end if
20:    mid ← (left + right)/2                    ▷ Calculăm mijlocul intervalului
21:    ret ← 0
22:    if x1 ≤ mid then
23:        ret += QUERY_X(node_x · 2, left, mid, x1, x2, y1, y2)
24:    end if
25:    if mid + 1 ≤ x2 then
26:        ret += QUERY_X(node_x · 2 + 1, mid + 1, right, x1, x2, y1, y2)
27:    end if
28:    return ret
29: end function
30:
31: QUERY_X(1, 1, N, x1, x2, y1, y2)             ▷ Apelăm funcția
```

Interogarea unei submatrici în arborele de intervale 2D va începe prin a găsi nodurile care formează intervalul $[x_1, x_2]$. În arborii de intervale din aceste noduri vom găsi nodurile care formează intervalul $[y_1, y_2]$. Rezultatul va fi suma tuturor valorilor din acestea.

Complexitate. Intervalul $[x_1, x_2]$ va fi format din $O(log_2 N)$ noduri (acest lucru fiind demonstrat în primul capitol), iar, analog, intervalul $[y_1, y_2]$ va fi format din $O(log_2 M)$ noduri. Complexitatea timp va fi $O(log_2 N \cdot log_2 M)$.

VI Aplicații a arborilor de intervale în alți algoritmi

VI.1 Heavy path decomposition

Concept. Heavy path decomposition (numit și Heavy-light decomposition) este un algoritm care descompune un arbore înrădăcinat în mai multe lanțuri. Aceste lanțuri se formează după următoarea regulă: fiecare nod care nu este frunză va face parte din același lanț ca fiul sau cu cel mai greu subarbore (subarborele cu cele mai multe noduri). Astfel, se poate demonstra faptul că orice drum de la radacină la un nod va trece prin $O(log_2 N)$ lanțuri din descompunere. Conceptul a fost introdus de către Daniel D. Sleator și Robert Endre Tarjan în lucrarea "A data structure for dynamic trees" [20], unde au și demonstrat acest lucru.

Problema de bază. [6] Fie un arbore cu N noduri unde fiecare nod are asociată o valoare v_i. Se dau M instrucțiuni de 2 tipuri:

1. Valoare v_x asociată nodului X devine val.
2. Se dau 2 noduri X și Y, să se afișeze valoare maximă de pe lanțul elementar cu extremitățile în aceste noduri.

Soluție. Pentru a rezolva eficient această problemă, mai întâi vom înrădăcina arborele într-un nod oarecare (pentru simplitate putem alege chiar nodul 1). Astfel, putem descompune arborele în lanțuri folosind Heavy path decomposition. În acest moment, orice drum de la rădăcină la un nod X va trece prin $O(log_2 N)$ lanțuri din descompunere. Cum orice lanț elementar este format din drumul de la fiecare extremitate până la cel mai apropiat strămoș comun al acestora, iar cel mai apropiat strămoș comun se află chiar pe drumul de la rădăcină la fiecare extremitate, reiese că orice lanț elementar va parcurge $O(log_2 N)$ lanțuri din descompunere.

Singurul lucru rămas este să putem răspunde eficient la interogări de tipul maximul de pe un interval dintr-un lanț din descompunere și să putem face operații de actualizare de element în aceste lanțuri. Arborii de intervale sunt perfecți pentru acest lucru, astfel, fiecare lanț din descompunere se va retine ca un arbore de intervale a nodurilor din acesta. Deci fiecare actualizare și interogare într-un lanț se poate realiza în complexitate timp $O(log_2 N)$.

Complexitate. Fiecare actualizare se va realiza în complexitate timp $O(log_2 N)$. Pentru fiecare interogare se va trece prin $O(log_2 N)$ lanțuri din descompunere, iar pentru fiecare astfel de lanț interogarea unei subsecvente se realizează în complexitate timp $O(log_2 N)$ datorită arborelui de intervale asociat acestuia. Deci complexitatea timp a unei interogări este $O((log_2 N)^2)$. Astfel, complexitatea timp finală este $O(M \cdot (log_2 N)^2)$.

VI.2 Optimizări de programare dinamică

Concept. În programarea dinamică, arborii de intervale sunt foarte utili pentru recurențe în care starea curentă este definită pe baza minimului/maximului/suma a unui interval de stări din trecut.

Cel mai lung subșir crescător. [5] Determinarea celui mai lung subșir crescător este o problema clasică de programare dinamică în care, dându-se un șir v de N numere naturale, trebuie sa se determine lungimea celui mai lung subșir al lui v în care elementele acestuia se află într-o ordine crescătoare.

Această problemă are 2 rezolvări în complexitate timp optimă $O(N \cdot log_2 N)$, una dintre aceste soluții fiind bazată pe arbori de intervale. [14]

Soluție. Primul pas al soluției este să normalizăm numerele din șirul v. Spre exemplu, dacă numerele din șir ar fi $[15, 22, 22, 40]$ acestea ar deveni $[1, 2, 2, 3]$. Astfel, valorile din șir acum vor fie cuprinse între 1 și N.

Logica algoritmului va fi ca, pentru fiecare element v_i, definim dp_i ca lungimea celui mai lung subșir crescător care se termină cu acest element. Pentru a realiza acest lucru ne vom reține un arbore de intervale unde fiecare frunză va corespunde unei valori din șir (acestea fiind cuprinse acum între 1 și N). Frunza i va reține lungimea celui mai lung subșir crescător de până acum care se termină în valoarea i. Arborele va permite interogări de aflarea elementului maxim dintr-o subsecvență și se va inițializa cu valoarea 0 în toate nodurile.

Se parcurg elementele șirului v de la stânga la dreapta. Pentru un element v_i vom interoga arborele de intervale pe subsecvența $[1, v_i]$, notând acest rezultat cu Q. Q va reprezenta lungimea celui mai lung subșir crescător de până acum care se termină într-o valoare mai mică sau egală cu elementul curent. Astfel, înseamna ca putem extinde acest subșir cu elementul curent. Deci $dp_i = Q + 1$.

După aflarea valorii lui dp_i, se va actualiza frunza arborelui de intervale corespunzătoare poziției v_i cu valoarea dp_i, recalculându-se drumul de la rădăcina până la aceasta.

Rezultatul final va fi valoarea maximă din arborele de intervale după ce s-au parcurs toate elementele șirului v.

Algorithm 27 Cel mai lung subșir crescător

Se normalizează valorile elementelor șirului v.

for $i \leftarrow 1$ to N **do**
    $dp_i \leftarrow QUERY(1, 1, N, 1, v_i) + 1$ ▷ Se află maximul din intervalul $[1, v_i]$
    $UPDATE(1, 1, N, v_i, dp_i)$ ▷ Se actualizează frunza corespunzătoare poziției v_i cu valoarea dp_i
end for

Se afișează $QUERY(1, 1, N, 1, N)$ ▷ Elementul maxim din arbore

Complexitate. Deoarece se realizează câte o interogare și o actualizare în arborele de intervale pentru fiecare element din șirul v, complexitatea finală este $O(N \cdot log_2 N)$.

VII Arbori indexati binar

VII.1 Concept

Definiție. Nu putem vorbi despre arbori de intervale fără a preciza și arborii indexați binar, numiți și Fenwick Tree, fiind descriși prima oară în lucrarea "A New Data Structure for Cumulative Frequency Tables" [8] scrisă de Peter M. Fenwick. Aceștia au un subset din ficționalitățile unui arbore de intervale, având aceeași complexitate teoretică logaritmică pe operații. În practică, arborii indexați binar se comportă mai bine decât arborii de intervale deoarece prezintă o constantă mică la complexitate.

Fie v un șir de N elemente și o funcție f. Formal, arborele indexat binar este definit ca structura de date care [11]:

- Permite calcularea valorii funcției f peste elementele din șirul v dintr-o subsecvență în complexitate $O(log_2 N)$.
- Permite actualizarea elementelor din v în complexitate $O(log_2 N)$.
- Folosește $O(N)$ spațiu.
- Nodul i din arborele indexat binar va reține valoarea funcției f peste elementele din intervalul de la i până la nodul părinte al lui i. Vom descrie în continuare cum se realizează legăturile dintre nodurile unui arbore indexat binar.

Părintele nodului i va fi reprezentat de nodul $i - LSB(i)$ unde LSB reprezintă cel mai puțin semnificativ bit în reprezentarea sa binară. De exemplu, $LSB(12) = LSB(1100_2) = 100_2 = 4$. Deci părintele nodului 12 este nodul $12 - 4 = 8$, iar nodul 12 va reține valoarea funcției f peste elementele din intervalul $[9, 12]$. $LSB(i)$ se poate calcula ușor cu formula $LSB(i) = i\&(-i)$, unde $\&$ reprezintă operatorul bitwise AND. [8]

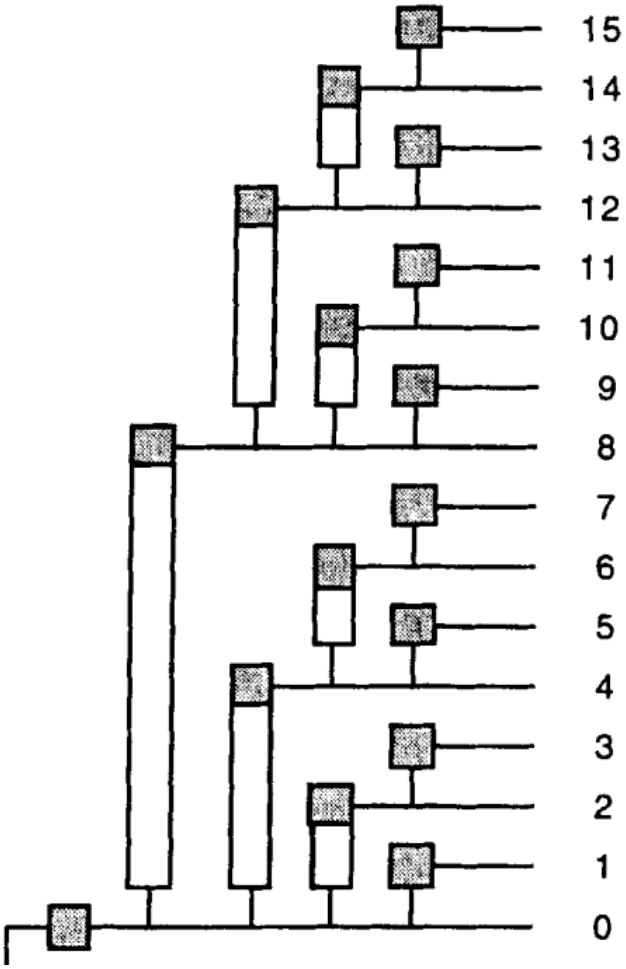

Figura VII.1: Intervalele reprezentate de nodurile din arborele indexat binar [8]

Funcționalitate de bază. Arborele indexat binar are ca scop principal calcularea sumei elementelor subsecvențelor unui șir, permițând și actualizarea elementelor. Astfel, voi descrie operațiile arborelui indexat binar unde funcția f reprezintă funcția de sumă a elementelor dintr-un interval.

VII.2 Operații

Algoritm	Complexitate Timp
Inițializare	$O(N)$
Actualizarea unui element	$O(log_2 N)$
Interogarea unei subsecvențe	$O(log_2 N)$

Inițializare

Concept. Avem un vector v indexat de la 1 la N și vrem să inițializăm valorile din arborele indexat binar pe baza acestuia.

Deoarece fiecare nod i va reține suma elementelor din intervalul $[i - LSB(i) + 1, i]$, se poate demonstra relația:

$$aib[i] = v[i] + \sum_{j+LSB(j)=i} aib[j]$$

.

Demonstrație. Orice nod i poate fi reprezentat în baza 2 ca $...10^k$ unde 0^k reprezintă un număr natural (posibil nul) k de zerouri la finalul numărului în baza 2. Astfel, $LSB(i) = 2^k$ și nodul i va reține suma pe intervalul $[i - 2^k + 1, i]$.

Orice nod j cu proprietatea că $j + LSB(j) = i$ poate fi reprezentat în baza 2 ca $...01^x0^{k-x}$. Acest lucru se datorează faptului că, la adăugarea $LSB(j)$, toate cele x de 1 din reprezentarea binară se vor transforma în 0 iar primul 0 dinainte de acestea în 1. Astfel, rezultând numărul $...10^k = i$.

Sunt k astfel de numere, fiecare dintre acestea reprezentând un interval $[i-2^{k-x+1}+1, i-2^{k-x}]$. Aceste intervale sunt disjuncte și reuniunea acestora este $[i - 2^k + 1, i - 1]$. Așadar, mai trebuie adăugat elementul $v[i]$ pentru a crea intervalul $[i-2^k+1, i]$, fiind intervalul reprezentat de nodul i.

Algorithm 28 Inițializare arbore indexat binar [8]

```
1: function INIT
2:     for i ← 1 to N do
3:         aib[i] += v[i]                          ▷ Adaugăm elementul v[i] la aib[i]
4:         if i + LSB(i) ≤ N then
5:             aib[i + LSB(i)] += aib[i]      ▷ Adaugăm valoarea aib[i] la nodul i + LSB(i)
6:         end if
7:     end for
8: end function
```

Complexitate. Algoritmul de inițializare este liniar, având complexitate timp $O(N)$.

Actualizarea unui element

Concept. Se dorește adăugarea valorii val elementului aflat la poziția pos în șirul v. Pentru a realiza acest lucru în arborele indexat binar, va trebui să actualizăm atât nodul pos, cât și toate nodurile care conțin elementul $v[pos]$ în intervalele acestora. Fiind demonstrat anterior faptul că orice nod j va fi cuprins în nodul $j + LSB(j)$, rezultă faptul că elementul $v[pos]$ va fi conținut în nodurile $pos, pos + LSB(pos), pos + LSB(pos) + LSB(pos + LSB(pos)), ...$ Deci vor trebui actualizate toate aceste noduri.

Algorithm 29 Actualizare poziție arbore indexat binar [8]

```
1: function UPDATE(pos, val)
2:     while pos ≤ N do
3:         aib[pos] += val                    ▷ Adăugăm val la aib[pos]
4:         pos += LSB(pos)                    ▷ Trecem la următorul nod de actualizat
5:     end while
6: end function
```

Complexitate. Reprezentăm poziția pos în baza 2 ca $...10^k$. Astfel, $LSB(pos) = 2^k$. Prin adăugarea lui $LSB(pos)$ la pos, se va genera un număr cu cel puțin $k + 1$ zerouri la final, deoarece ultimul 1 din reprezentarea binară a lui pos va deveni 0. Acest eveniment poate fi generalizat prin inecuația $LSB(pos + LSB(pos)) \geq 2 \cdot LSB(pos)$.

Cum la fiecare pas se va adăuga un număr de cel puțin 2 ori mai mare ca la pasul anteriori, se vor parcurge $O(log_2 N)$ noduri până se va depăși valoare N. Astfel, complexitatea timp este $O(log_2 N)$.

Interogarea unei subsecvențe

Concept. Arborele indexat binar permite interogarea eficientă a sumei unui prefix al șirului, interogarea unei subsecvențe $[a, b]$ fiind diferența dintre suma prefixului $[1, b]$ și cea a prefixului $[1, a - 1]$.

Pentru a interoga un prefix $[1, pos]$, se pornește din nodul aflat la poziția pos și se vor parcurge nodurile arborelui mergând mereu în părintele nodului curent. Adunând valorile din aceste noduri se va genera suma întregului prefix.

Algorithm 30 Interogare suma interval arbore indexat binar [8]

```
1: function QUERYPREFIX(pos)
2:     sum ← 0
3:     while pos > 0 do
4:         sum += aib[pos]
5:         pos −= LSB(pos)
6:     end while
7:     return sum
8: end function
9:
10: function QUERYINTERVAL(a, b)
11:     return QUERYPREFIX(b) − QUERYPREFIX(a − 1)
12: end function
```

Complexitate. La fiecare pas în interogarea unui prefix, se va scădea $LSB(pos)$ din poziția curentă pos. Astfel, la fiecare pas poziția curentă va avea cu exact un bit de 1 mai puțin în reprezentarea acestuia binară. Deoarece există $O(log_2 N)$ biți de 1 în reprezentarea binară a lui pos, se vor realiza $O(log_2 N)$ pași. Deci complexitatea timp a unei interogări este $O(log_2 N)$.

VIII Concluzii

În lucrarea de față au fost prezentați arborii de intervale împreună cu diversele variații ale acestora. Teoria a fost complementă de probleme reprezentative, subliniind importanța acestora practică. Accentul a fost pus pe operațiile permise de fiecare structură de date și complexitatea acestora. Din acest motiv, le vom reaminti în concluzie.

Un arbore de interval simplu permite următoarele operații:

Algoritm	Complexitate Timp
Inițializare	$O(N)$
Actualizarea unui element	$O(log_2 N)$
Interogarea unei subsecvențe	$O(log_2 N)$
Căutarea unui prefix	$O(log_2 N)$

”Lazy propagation” introduce actualizarea unei subsecvențe în complexitate timp $O(log_2 N)$. Astfel, un arbore de intervale va avea asociate următoarele operații:

Algoritm	Complexitate Timp
Inițializare	$O(N)$
Actualizarea unui element	$O(log_2 N)$
Actualizarea unei subsecvențe	$O(log_2 N)$
Interogarea unei subsecvențe	$O(log_2 N)$
Căutarea unui prefix	$O(log_2 N)$

Arborele de intervale persistent o să introducă conceptul de stare anterioară a structurii, acesta permițând atât operațiile specifice unui arbore de intervale, cât și cele specifice unei structuri de date persistente:

Algoritm	Complexitate Timp	Complexitate Spațiu
Inițializare	$O(N)$	$O(N)$
Actualizarea unui element	$O(log_2 N)$	$O(log_2 N)$
Actualizarea unei subsecvențe	$O(log_2 N)$	$O(log_2 N)$
Interogarea unei subsecvențe într-o stare anterioară	$O(log_2 N)$	$O(1)$
UNDO (anulăm ultima actualizare)	$O(1)$	$O(1)$
REBASE (ne întoarcem la o stare a arborelui)	$O(1)$	$O(1)$

Arborele de intervale 2D va permite extinderea operațiilor la două dimensiuni, astfel arborele acum se va baza pe o matrice în loc de un șir de elemente:

Algoritm	Complexitate Timp
Inițializare	$O(N \cdot M)$
Actualizarea unui element	$O(log_2 N \cdot log_2 M)$
Interogarea unei submatrice	$O(log_2 N \cdot log_2 M)$

Arborii de intervale au un rol semnificativ în rezolvarea a multor probleme algoritmice într-o complexitate timp optimă. În plus, aceștia fac parte din algoritmi consacrați precum Heavy path decomposition [6] sau problema determinării celui mai lung subșir crescător al unui șir [14].

Direcții de dezvoltare

Extinderea lucrării poate fi realizată prin abordarea structurilor de date similare cu arborii de intervale precum K-d Tree [2] si Quadtree [9], acestea fiind importante în domeniul geometriei computaționale.

Lucrarea de față va fi integrată atât ca parte a unui volum dedicat structurilor de date, cât și în cadrul programei unui curs opțional al Facultății de Matematică și Informatică ce va aborda structuri de date avansate.

În vederea acestui aspect, noi probleme rezolvate vor fi adăugate în cadrul capitolelor pentru a oferi studenților mai multe exemple practice și oportunități de aplicare a teoriei predate.

Bibliografie

[1] Mugurel Ionut Andreica. Hotel. *Infoarena*, 2002. Accesat 17 Iunie 2023 3:23 PM EEST https://infoarena.ro/problema/hotel.

[2] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, sep 1975. doi:10.1145/361002.361007.

[3] Jon Louis Bentley. Solutions to klee's rectangle problems. *Technical report*, 1977.

[4] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989. doi:https://doi.org/10.1016/0022-0000(89)90034-2.

[5] Arhiva Educationala. Subsir crescator maximal. *Infoarena*, 2008. Accesat 17 Iunie 2023 4:10 PM EEST https://www.infoarena.ro/problema/scmax.

[6] Arhiva Educationala. Heavy path decomposition. *Infoarena*, 2013. Accesat 17 Iunie 2023 3:40 PM EEST https://www.infoarena.ro/problema/heavypath.

[7] Jeff Erickson. Klee's measure problem. Accesat 17 Iunie 2023 3:55 PM EEST https://jeffe.cs.illinois.edu/open/klee.html.

[8] Peter M Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and experience*, 24(3):327–336, 1994. doi:https://doi.org/10.1002/spe.4380240306.

[9] Raphael Finkel and Jon Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 03 1974. doi:10.1007/BF00288933.

[10] Yordan Chaparov Georgi Georgiev. Ants. *Varena*, 2014. Accesat 17 Iunie 2023 3:34 PM EEST https://www.varena.ro/problema/ants.

[11] Jakob Kogler. Fenwick tree. *cp-algorithms*, 2014. Accesat 17 Iunie 2023 3:50 PM EEST https://cp-algorithms.com/data_structures/fenwick.html.

[12] Jakob Kogler. Segment tree. *cp-algorithms*, 2014. Accesat 17 Iunie 2023 3:46 PM EEST https://cp-algorithms.com/data_structures/segment_tree.html.

[13] Dana Lica. Arbori de intervale si aplicatii in geometria computationala. *Infoarena*, 2010. Accesat 17 Iunie 2023 3:12 PM EEST https://www.infoarena.ro/arbori-de-intervale.

[14] Alexandru Luchianov. [tutorial]using segment trees to solve dynamic programming problems. *Codeforces*, 2022. Accesat 17 Iunie 2023 4:01 PM EEST https://codeforces.com/blog/entry/101210.

[15] Amr Mahmoud. A simple task. *Codeforces*, 2015. Accesat 17 Iunie 2023 3:25 PM EEST https://codeforces.com/contest/558/problem/E.

[16] Denis-Gabriel Mita. Calafat. *Infoarena*, 2016. Accesat 17 Iunie 2023 3:20 PM EEST https://www.infoarena.ro/problema/calafat.

[17] Serj Nagin. Sereja and brackets. *Codeforces*, 2014. Accesat 17 Iunie 2023 3:17 PM EEST https://codeforces.com/contest/380/problem/C.

[18] Daniel Pasaila. Maxq. *Infoarena*, 2007. Accesat 17 Iunie 2023 3:14 PM EEST https://www.infoarena.ro/problema/maxq.

[19] Pushkar Mishra. A new algorithm for updating and querying sub-arrays of multidimensional arrays, 2015. doi:10.13140/RG.2.1.2394.2485.

[20] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:https://doi.org/10.1016/0022-0000(83)90006-5.