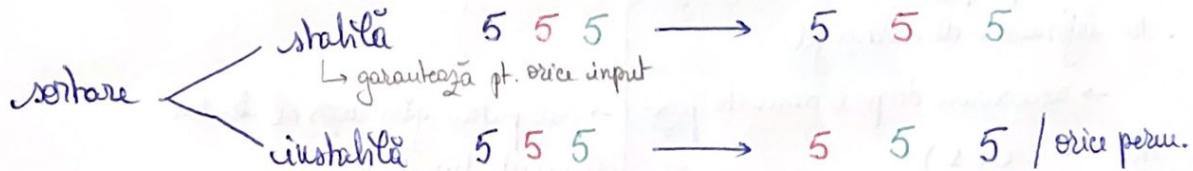


Burs 1-2 → SORTĂRI

Algoritmi sortare

lăsătare lineară - $O(n \log n)$

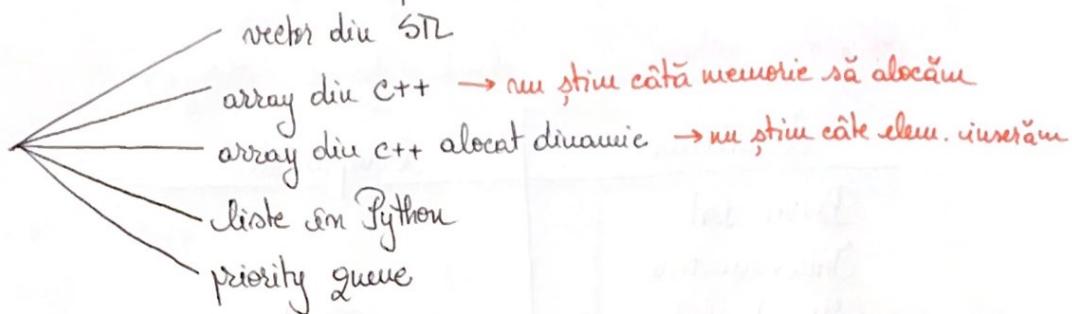
Denumire	Complexitate
Bubble Sort	$O(n^2)$
Interschimbare	
Gravity Sort	
Selection Sort	
Insert Sort	
Merge Sort	$O(n \log n) \rightarrow O(n^2)$
Quick Sort	
Heap Sort	$O(n \log n)$
Intro Sort	
Tim Sort	
Radix Sort	$O(n \log \text{max})$
Bucket Sort	$O(n+k) \rightarrow O(n^2)$
Count Sort	$O(n + \text{max})$
Shell Sort	$O(n \sqrt{n}) \sim O(n \cdot n!)$
Bogo Sort	



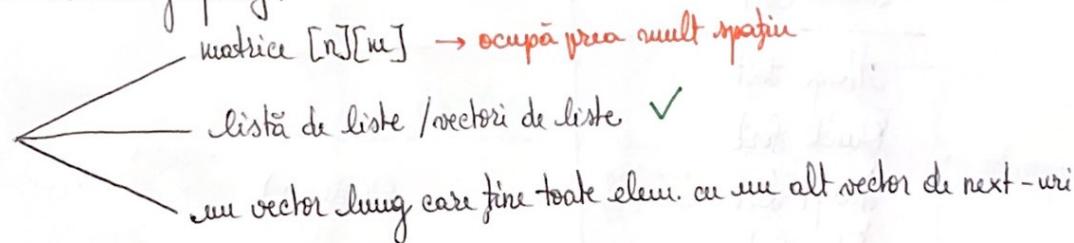
(Th) Orice alg. de sortare care se bazează pe comparații face cel puțin $O(n \log n)$ comparații.

Lecție 3-4 → LISTE, VECTORI, STIVE, COZI, DEQUE

ex: Se citesc nr. de la tastatură până se citește 0. Sortați acele nr.



ex: Se citesc $n \leq 10^6$ nr. care fac parte din unul din cele $m \leq 10^3$ grupuri. Care este al ck-lea nr. din grupul j?



array	Liste
<ul style="list-style-type: none"> → nuai proprietăți → pet căea o risipă de memorie; nu știm antideamă câtă memorie tehnice alocate → putem avea probleme să alo căm o secvență lungă continuă sau să o extindem → ocupă poziții consecutive și reține informații de același fel → accesarea de pe o anumită poziție - $O(1)$ 	<ul style="list-style-type: none"> → permite alocarea memoriei când avem nevoie de ea → nu putem găsi index al k-lea element din listă → inserare / ștergere - $O(1)$ dacă avem pointer-ul de care avem nevoie → alo căm / șterge memoria

COMPLEXITATE

	dliste	array
Încărcare oriunde	$O(1)$	$O(n)$
Redimensionare	$O(1)$	$O(1)$
Încărcare / stergere capăt	$O(1)$	$O(1)$
Adăugare al k-lea element	$O(k)$	$O(1)$
Sortare	$O(n \log n)$	$O(n \log n)$
Înlăturare din struc. sortată	$O(n)$	$O(\log n)$

Array	Vector
<ul style="list-style-type: none"> → facem risipa → trebuie să reținem căte elemente folosim → rapizi → folosesc memoria eficient 	<ul style="list-style-type: none"> → alocăm înainte memorie la început => putem aloca din start și nr. de elem. → viteză mai mică → redimensionabil → array declarat dinamic → putem rezerva locuri / să adăugăm la final

→ liste înălțuite

↳ elem. de tipuri diferite

↳ simple înălțuite → → → ...

↳ duble înălțuite ⇌ ⇌ ⇌ ...

↳ circulare ↗ / ↘

↳ container (list)

↳ alocate de memorie

→ Stive ← implement. ca vector
implement. ca listă

↳ LIFO

→ avem acces numai la elem. din vîrf

→ operări

* push = adaugă un elem. în vf.

* pop = eliminarea elem. din vf.

* size() = nr. elem.

* isEmpty() = returnază true dacă nr. de elem. e 0

* peek() = spune valoarea din vf. fără să o extragă

→ Logi (queue) ← implement. ca vector
implement. ca listă

↳ FIFO

→ avem acces la primul și ultimul element

→ operări

* push = adaugă un element la coadă

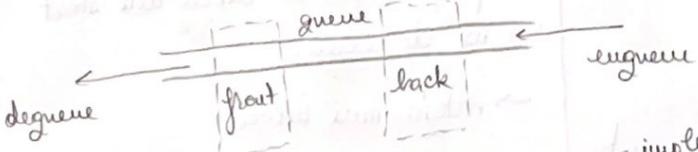
* pop = eliminarea unui elem.

* size() = nr. elem.

* isEmpty() = returnază true dacă nr. de elem. e 0

* first() = spune valoarea de la început fără extragere

* last() = spune valoarea de la sfârșit fără extragere



→ Locată cu două capete (dequeue) ← implement. ca listă
implement. ca array

→ operări

* push ← front
back

* pop ← front
back

* size()

* front()

* back()

* isEmpty()

ex: Se dă un vector cu n elem. și apoi m operații de genul:

1) $i \in j$ — care este min din intervalul $[i, j]$

2) $i \in x$ — modificarea elem. de pe poz. i din x

$\text{sgrt}(n, m, v)$

Smerul lui Batoz: împărțim vectorul în zone de lungime L și calculăm minimul pe fiecare zonă în parte

→ complexitate 1 (query)

- împărțim vectorul în n/L zone de lungime L

- putem itera aproape complet 2 zone (început și/sau final)

$$O(2L) \Rightarrow O(n/L + 2 \times L)$$

* pt. complexitate minimă: $L = \text{sgrt}(n)$

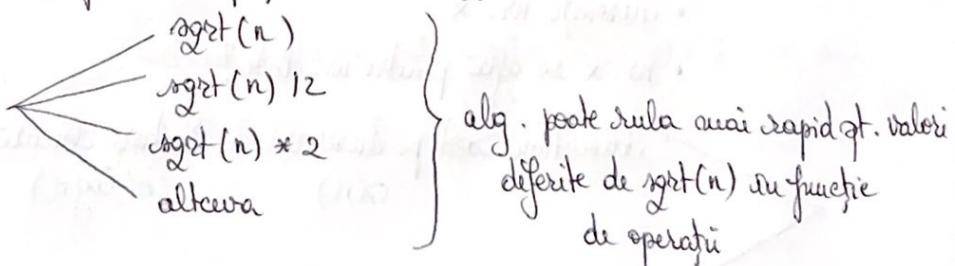
$$O(\text{sgrt}(n))$$

→ complexitate 2 (update)

- modificare elem. de pe poz. i

- trebuie să facem update pe zona respectivă (să recalculem min) $\Rightarrow O(L) = O(\text{sgrt}(n))$

≈ pt. ceea ce putem împărti vectorul în zone:



Hours 5-6 → HASH-URI

COMPLEXITATE

	Inserare	Stergere min	Stergere cu pointer	Stergere fără pointer	afisare min	Căutare	Făcător succesor	afisare sortat
Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n \log n)$
Arbore de căutare echilibrat	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Vector	$O(1)$	$O(n)$	$O(1) / O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log n)$
Listă înláțuită	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log n)$
Hash-uri	$O(1)$		$O(1)$	$O(1)$		$O(1)$		

Ex: Se dau 2 tipuri de operații pe nr de la 1 la N. Se dă pâna la $M \leq 10^6$ operații

- inserati nr. x
- nr. x se află printre ur. date?

inserație sort pe inserare + căutare lineară → inefficient

$$O(n) + O(n \log n)$$

Puteam avea un vector linear: $a[i] = 1$ dacă elem. s-a dat

$$\left\{ \begin{array}{l} \\ 0 \end{array} \right. \quad \text{sau} \quad a[i] = 0 \text{ dacă elem. nu s-a dat}$$

$$O(1) \text{ inserare} + O(1) \text{ căutare}$$

Ex: Se dau 2 tipuri de operații pe nr. de la 1 la N. ($N \leq 10^6$)

- inserati nr. x
- nr. x se află printre ur. date?
- stergere: elimină nr. x din ur. măle

— analog. ex anterior $O(1) \text{ inserare} + O(1) \text{ căutare} + O(1) \text{ stergere}$

→ Funcție de dispersie

$$\hookrightarrow h(x) = x \% p, \quad p \text{ nr. prime} \quad \{ \text{prin } g(x)(n) \} \quad O(\log(n)) \quad O(n/p)$$

ex: Vrem să calculăm toate aparițiile unui sir mai mic sau mai mare

(pattern matching)

Algoritmul Rabin Karp

- ① calculăm hash-ul pt. sirul mai mic
- ② calculăm hash-ul pt. toate sirurile de aceeași lungime din sirul mai mare

↪ dacă două siruri au hash-uri egale \Rightarrow facem 2 hash-uri și vedem dacă anchete sunt egale

↪ dacă da \Rightarrow OK

↪ reprobarea coliziunilor: vomține o listă căutată

③ Dacă h e aleasă dintr-o colecție universală de funcții de dispersie și este folosită pt. a dispersa n chei într-o tabelă de dimensiune m , $n \leq m$, nr. median de coliziuni în care este implicată o cheie particulară x este mai mic decât 1.

↪ 2 metode de calculare a poziției cheii. în tabelul de dispersie de mărime

m testare liniară: $h(x, i) = (h(x, 0) + i) \% m$

hash dublu: $h(x, i) = (h_1(x) + i * h_2(x)) \% m$

Curs 7 - 8

→ HEAP-URI

→ graf $G = (V, E)$

V - noduri
 E - muchii

→ arbore

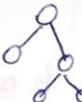
- graf conex aciclic, cu n noduri, cu $n-1$ muchii
- graf aciclic maximal, cu $n-1$ muchii
- nu are cicluri, dreapta unică spre +2 noduri
- dacă are $n \geq 2$ nf ⇒ conține minim 2 frunze
 - ↳ nod de gradul 2 (poate fi inclusiv rădăcina)
- un frunză de rădăcină ⇒ nf diferit

→ Arboare binară

= arbore cu rădăcină în care fiecare nod are cel mult 2 copii

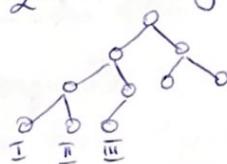
stâng (left, L) drept (right, R)

plin → fiecare nod are 0/2 fiu



balansat → diferența între fiul stâng și drept este max 1, pt. fiecare nod

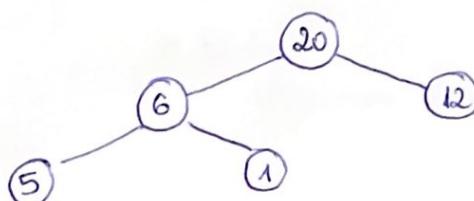
complet → toate nivelurile sunt complete, cu excepția ultimului care se completează de la stânga la dreapta.



→ un arbore binar cu adâncimea h are cel mult $2^{h+1} - 1$

→ Heaps

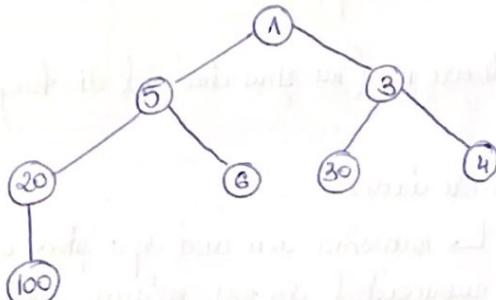
* **Heap de maxim** = arbore binar complet cu proprietatea că fiecare nod este mai mare decât fiu său



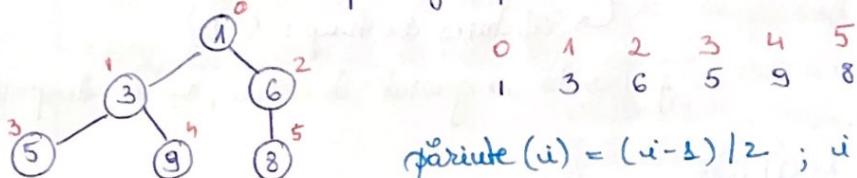
* **heap de minim** = arbore binar complet cu proprietatea că fiecare nod este mai mic decât fișorii săi

↳ muchiile pot fi mai mici decât nepoții (5 < 4)

↳ ordonare pe nivele; dor către descendenți



→ un arbore binar poate fi reprezentat ca vector



$$! h = \log n$$

→ repararea heap-urilor (prin curățare): $O(\log n)$

→ inserare în heap-uri: $O(\log n)$

↳ elementul e pus pe ultima poziție și se repară heap-ul curățare

→ repararea heap-urilor (prin colorare): $O(\log n)$

→ eliminarea din heap-uri: $O(\log n)$

↳ elementul de pe ultima poziție este pus în locul rădăcinii și se repară heap-ul prin colorare

→ construire heap (Heapify): $O(n \log n)$

* inserare n elemente: $O(n \log n)$

* linear

→ colorăm fiecare element începând de jos în sus

ex : Se dau multe operații de genul :

* inserare $O(\log n)$

* afisare minim $O(1)$

* eliminare maxim

Bună poate folosi un heap?

PROBLEMA : Eliminare nr (nu știm indexul din heap \Rightarrow nu putem elibera în $\log n$)

Eliminare indirectă

→ marcam un nod spate stergere, dar el sterge și alia
în momentul în care ajunge să fie

→ mai multă memorie

→ căutarea din heap : $O(n)$

~ va fi folositor în general la arbori, nu la heap-uri

COMPLEXITATE

Operatie	Timp mediu	Worst case
Inserare min	$O(1)$	$O(1)$
Inserare	$O(1)$	$O(\log n)$
Stergere	$O(\log n)$	$O(\log n)$
Ispatiu	$O(n)$	$O(n)$
Căutare	$O(n)$	$O(n)$
Construcție n elem.	$O(n)$	$O(n)$
Minime (2 heap-uri de n elem)	$O(n)$	$O(n)$

→ Heap-uri Binomiale și Heap-uri Fibonacci

↳ reunirea e înceată și alte operații pot fi implementate

	Băutare min	Așterge min	Întoarcere	Update	Reuniune
Heap Binar	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$
Heap Binomial	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$ (amortizat)	$\Theta(\log n)$	$\Theta(\log n)$
Heap Fibonacci	$\Theta(1)$	$\Theta(\log n)$ (amortizat)	$\Theta(1)$	$\Theta(1)$ (amortizat)	$\Theta(1)$

→ Arborescențe binomiale

- un arbor binomial de ordin 0 are un singur nod
- un arbor binomial de ordin K = reunirea a 2 arbori binomiali de mărime K-1, fiecare pe rând din el fiind singur al celuilalt
- proprietăți

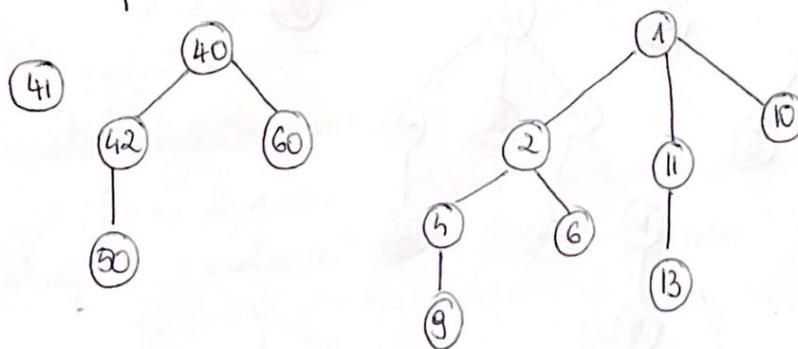
- * are exact 2^k noduri
- * are înalțimea k
- * sunt exact C_k^i noduri de înalțime i, ie $0 \leq i \leq k$
- * rădăcina are gradul k și copiii săi - arbori binomiali de tip $k-1, k-2, \dots, 0$

→ Heap-uri binomiale

= colecție de arbori binomiali, fiecare având proprietatea de heap minim

Obs. J este singura structură de heap binomial pt. orice mărime

ex: heap binomial cu 13 noduri



Băutare minim

- minimul - se află în rădăcina unui arbore binomial
- ne uităm la rădăcina lor și refinem minimul; $O(\log n)$
- totuști, putem fiind multe valoare când facem orice fel de operatie și să răspundem $O(1)$

Extragerea minimului

- eliminăm minimul
- apoi facem remeniu

Inserare

→ adăugăm un arbore binomial de mărime 1, apoi apelăm remeniuarea

Reuniune

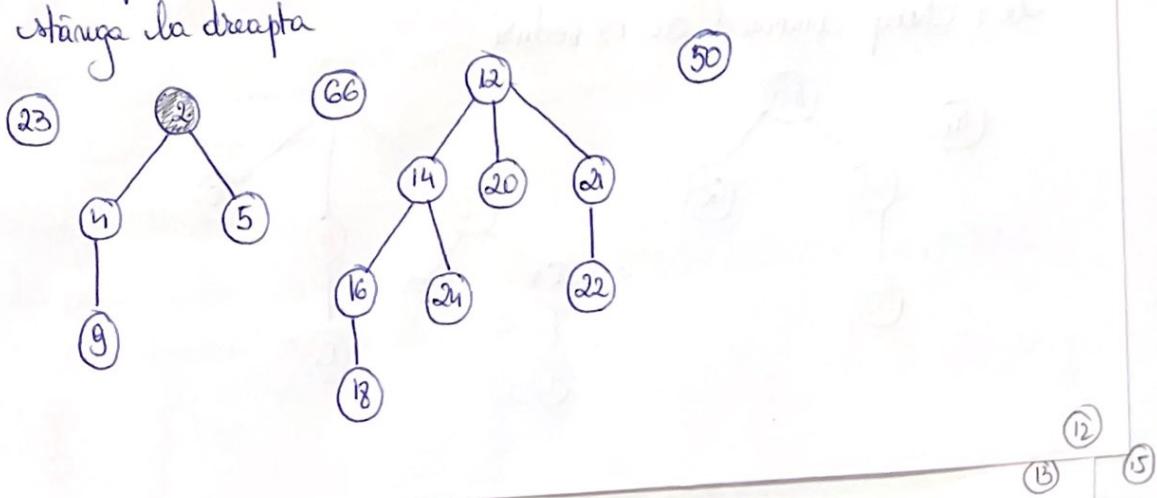
- $O(\log n)$
- reunirea a doi arbori; $O(1)$

→ **Heaps-wi Fibonaci**
= colecție de arbori care au proprietatea de ordonare de heap (arbore nu trebuie să fie binomiali)

→ arborii sunt ordonați

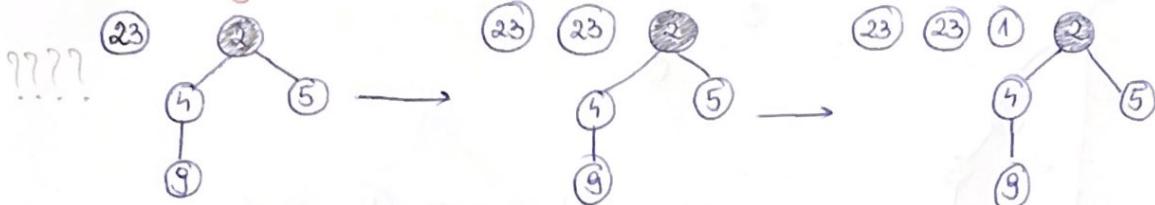
→ arborele din componentă are mărimea puteri ale lui 2

→ fizici - arbori de mărime $1, \dots, k-1$, dar nu neapărat sortați de la stânga la dreapta



Jusserare ned

- creare un arbore cu un singur element
 - il plasam în stânga rădăcinii
 - nu facem remenire $\rightarrow O(1)$

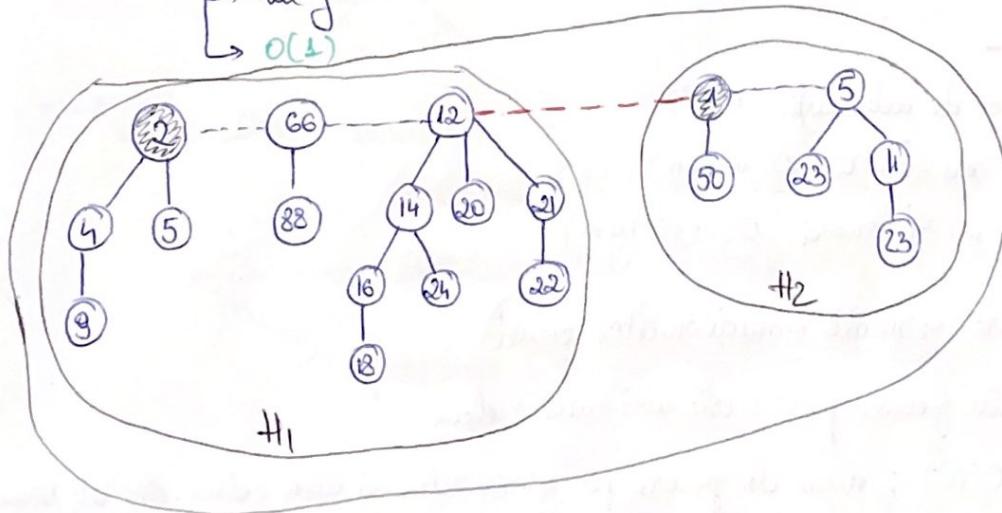


Bäubare minne

- la flèche pas finement pointer vers minima
 (ou)

Review

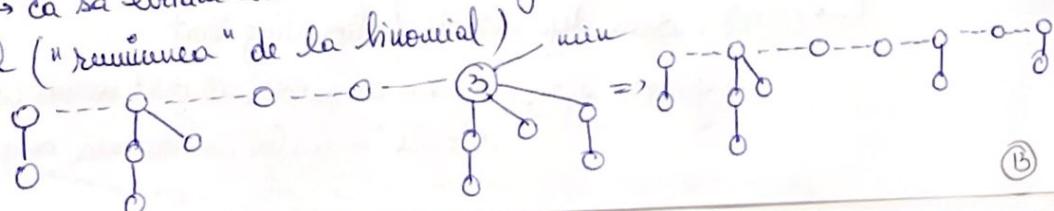
- concatenăm rădăcinele lui t_2 cu cele ale lui t_1
 - trebuie să păstrăm lista dublu enășterită
 - avem grija să păstrăm min
 - nu facem consolidarea

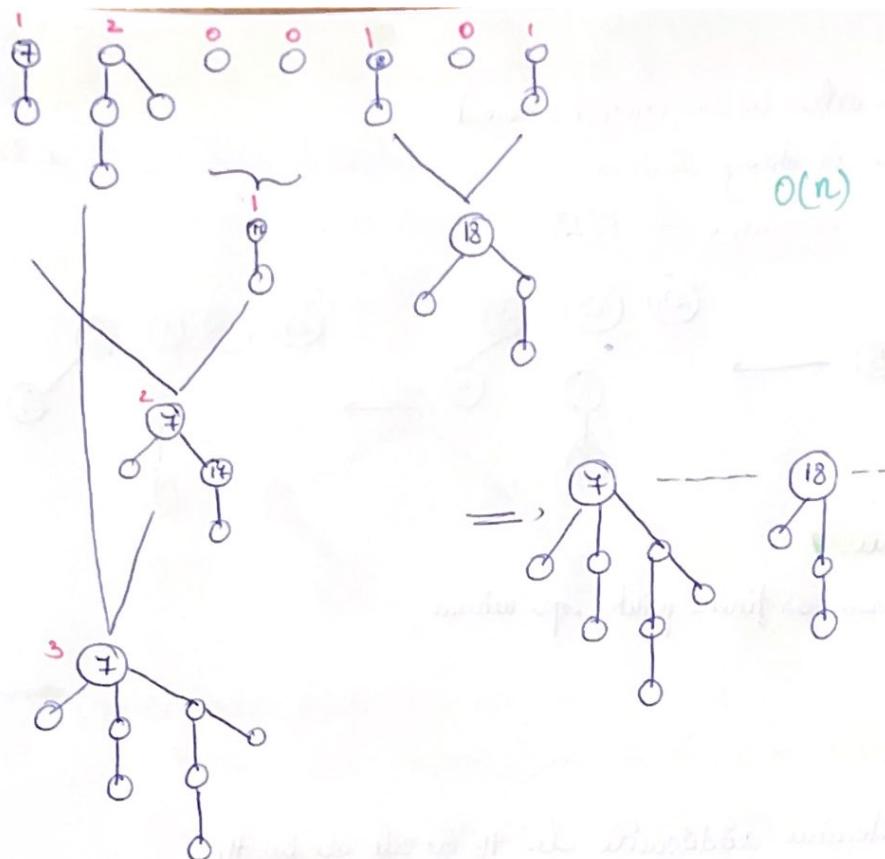


Extragerre minima

Extragere minim

↳ ca să evităm costul mare de extragere a minimului \Rightarrow consolidarea cheap-ului ("remânește" de la binomial) 





$\rightarrow O(\log n)$ pt. urmărirea
 $\rightarrow O(\log n)$ amortizat

Dijkstra

- matrice de adiacență $O(n^2)$
- heap-uri binare $O(m \log n)$
- heap-uri fibonacci $O(m + \log n)$

ex: Interclasare optimă a mai multor zăvoiuri

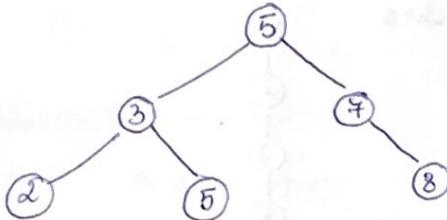
- la fiecare pas : cele mai mici 2 elem
- $O(n^2)$: dacă la fiecare pas găsim cele mai mici 2 elem. din cele lăsate
- $O(n \log n)$: heap-uri din care reținem toate valoările (inclusiv cele din urmă)
- $O(n)$: elem deja sortate / Counting Sort
 - * folosim 2 cozi - una cu valoare initial sortată; a doua cu valoare sunetele în ordinea care vin

Lecție 9

→ ARBORI BINARII DE CĂUTARE

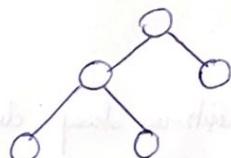
1.0 → Arbori binari de căutare

= arbore binar care satisfac proprietatea:
y nod din subarborele stâng \Rightarrow cheie[y] \leq cheie[x]
pt. un nod x depth \Rightarrow cheie[x] \leq cheie[y]



→ Arbori binari stricti

= arbore binar în care fiecare nod fie nu are niciun fiu, fie are exact doi
noduri doi copii: noduri interne
 fără copii: noduri externe / frunze

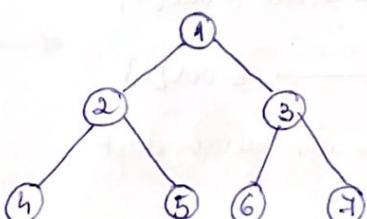


→ Arbori binari - parcurgeri

inordine (SRD, stânga rădăcină dreapta)

preordine (PSD, rădăcină stânga dreapta)

postordine (SDR, stânga dreapta rădăcină)



inordine: 4 2 5 1 6 3 7

preordine: 1 2 4 5 3 6 7

postordine: 4 5 2 6 7 3 1

1.1 → Arbori binari de căutare

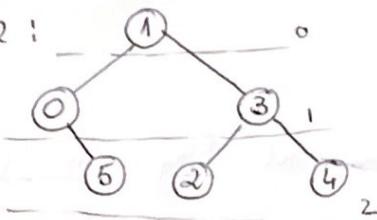
min: arb. binar complet: $\log n$

h max: lățaj (eleme. inserate cresc./descresc.): n

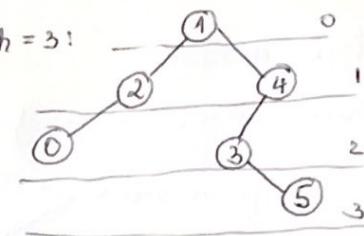
15

ex: Desenati arbori binari de inaltime 2, 3, 4, 5 pt. valoare {1, 2, 3, 4, 5}

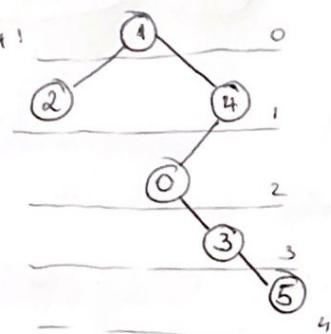
$h=2:$



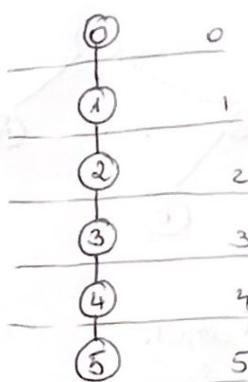
$h=3:$



$h=4:$



$h=5:$



Minim și Maximum

$O(h)$

cel mai din
stanga nod

cel mai din
dreapta nod

Ieșire

$O(h)$

→ min și max se găsesc mai greu decât într-un heap, dar căutarea unui nod din arbore este mai eficientă

→ începe din rădăcina și dacă valoarea din nodul curent este :

mai mică \Rightarrow mergem în stânga

mai mare \Rightarrow mergem în dreapta

Succesor și predecessor

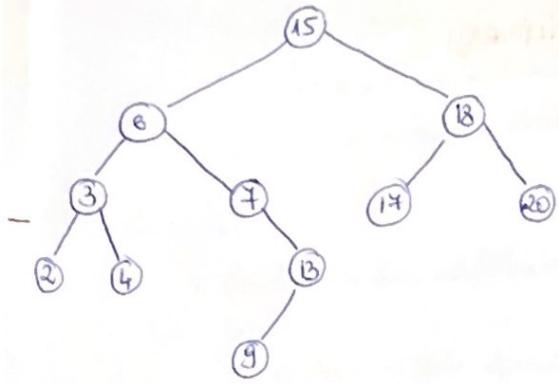
$O(h)$

→ succesor : care este cea mai mică valoare din arbore $\geq \text{val}[x]$

→ predecessor : $\text{--- more ---} \leq \text{val}[x]$

au fiu drept \Rightarrow cel mai mic elem. va fi elem. din subarb. drept

NU au fiu drept \Rightarrow primul următor în care se mută în subarb. său



necesar de 3 : 4
 → 6 : 7
 → 15 : 17
 → 13 : 15
 → 4 : 6

Inserare $O(h)$

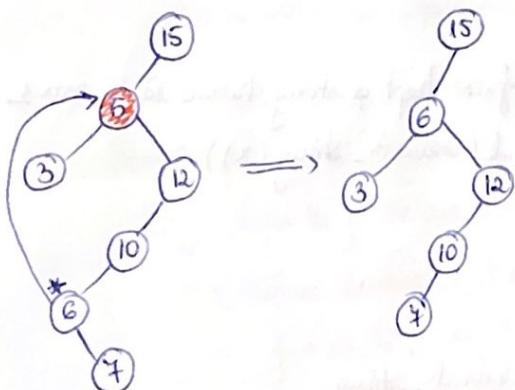
→ similar cu căutarea

Stergere $O(h)$

nodul nu are fiu \Rightarrow îl stergem

nodul are un fiu \Rightarrow îl stergem și creăm una tată și nou fiu

nodul are ambi fiu \Rightarrow găsim succesorul său, prenem de locul său și înlocuim legătura acestui nod cu viitorul fiu
(dacă există)



Astori binari de căutare cu chei egale

→ în caz de egalitate, alegem tot timpul stânga / dreapta și inserăm în aceeași direcție

→ înțelegem o listă cu toate elem. egale sub-un singur nod

Cours 10-11 → ARBORI BINARI ECHILIBRATI



Th) Înaltimea medie a unui arbore binar de căutare construit aleator cu u elor distinție este $O(\lg n)$.

→ Red Black Trees

- fiecare nod
- rădăcina: mereu neagră
- nu putem avea două noduri adiacente roșii
- același nr. de noduri negre
- orice drum de la un nod la descendant - null

→ AVL

- pt. fiecare nod, diferența numărului fililor drept și stâng trebuie să fie max 1
- $BF = h(\text{subarb. drept}(x)) - h(\text{subarb. stâng}(x))$

Rotatii

1) rotatie stanga - stanga

- ↳ un nod este inserat în stanga subarb. stang
- ↳ se realizează o rotatie la dreapta

2) rotatie dreapta - dreapta

- ↳ un nod este inserat în dreapta subarb. drept
- ↳ se realizează o rotacie la stanga

3) rotatie dreapta - stanga

- ↳ un nod este inserat la dreapta subarb. stang
- ↳ se realizează două rotatii

4) rotatie stanga - dreapta

- când un nod este inserat la stanga subarb. drept
- se realizează două rotații

→ skip lists

→ structuri de date achiziționate

alte struc. de date eficiente
 $\leq \log n$

hash tables - nu sunt sortate

heap-uri - nu putem căuta în ei

AVL, Red Black Trees

→ căutare rapidă

→ elementele sunt sortate

→ implementate ca liste anumite

→ costul căutării : $\log n \cdot \sqrt[n]{n}$

$O(\log n)$

Căutare $O(\log n)$

- incepe căutarea cu primul nivel (cel mai de sus)
- avansăm în dreapta, până când, dacă am avansat, am merge prea departe (elem. următor este prea mare)
- ne mutăm din următoarea listă (mergem în jos)
- ii)

Inserare

$O(\log n)$

Obs. dista de jos trebuie să contină toate elem.

→ x trebuie inserat în lista cea mai de jos

→ căutăm locul lui x în lista de jos $\Rightarrow \text{search}(x)$

→ adăugăm x în locul găsit în lista cea mai de jos

→ cum alegem lista în care trebuie să fie adăugat? - probabilistică



q să fie inserat și la mij. elem.

$$p = \frac{1}{2}$$

Stergere

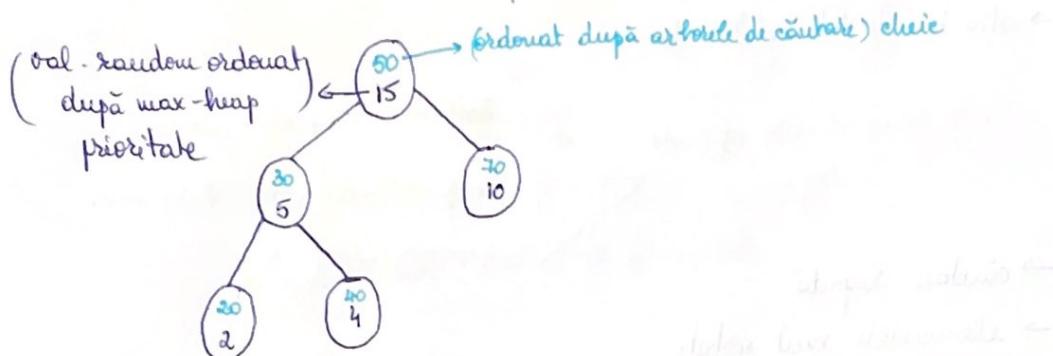
$O(\log n)$

→ stergem x din toate listele ce-l conțin

Heap-uri

= structură de date arborescentă care menține simultan proprietatea de:
arbori binari de căutare

max-heap

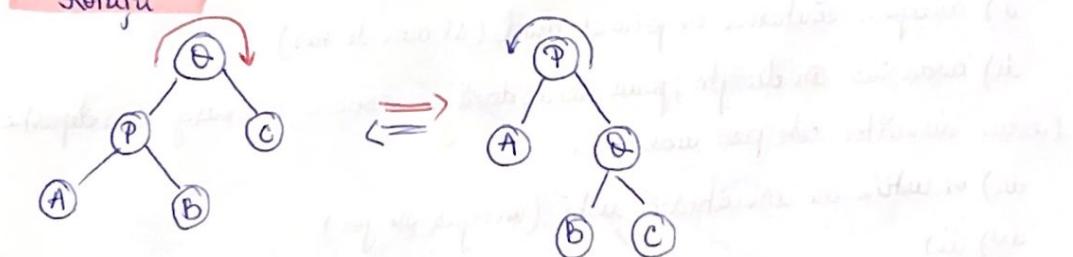


→ ușor de implementat

→ mai rapid decât arborii regii - negri și skip list - urile

→ cu modificări pot permite abordarea query-urilor și update-urilor

Rotări



Inserare

→ inserare recursiv ca la un arbori binari de căutare (TBC)

→ odată inserat, la fiecare pas înapoi în recursivitate se hindează în sens invers subarb. cu rădăcina un nodul curent din frunze de tipul fiului

Înlăturare

→ analog TBC

Stergere

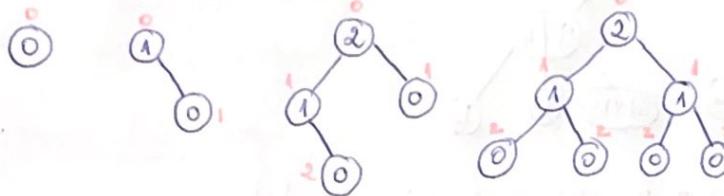
are frunze \Rightarrow stergere

are numai un fiu \Rightarrow fiul în locul nodului respectiv

are doi fiu \Rightarrow se hindează în locul rădăcinii subarb. făcut din nodul curent fiul cu prop. cea mai mare. Repetăm până la (1), (2)

B-arbore

→ arbore echilibrat = arbore în care, în nod, diferența dintre adâncimile subarborelor cîntăgării de la drept este maxim. 1.



= arbore echilibrat, destinat căutării eficiente

→ poate avea mai mult de 2 fiu pt. un nod

→ înălțimea: $O(\log n)$

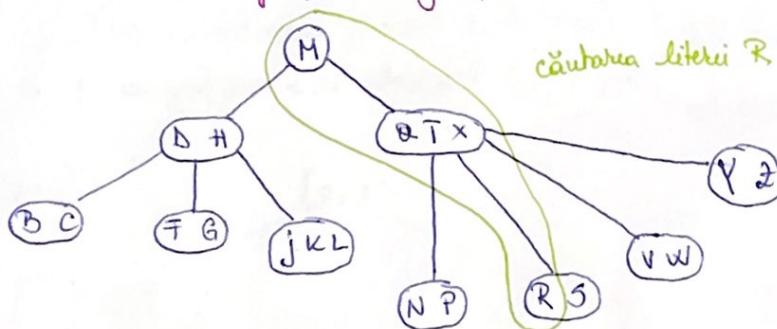
→ proprietăți

* un nod poate să contină mai mult de o cheie

* nr. de chei ale unui nod $x = n[x]$ - nr. chei memorate în nodul x

* un nod $x = n[x] + \Delta$ fiu

* toate frunzele se află pe același nivel



Cheiă₁[x] ≤ cheie₂[x] ≤ ... ≤ cheie_{n[x]}[x]

→ x nod intern $\Rightarrow n[x]+1$ poziții către fiu săi

→ cheie_i - cheie_{i+1} cîtarecare între-un subarbore cu rădăcină fiu; [x]

$k_1 \leq \text{cheie}_1[x] \leq k_2 \leq \text{cheie}_2[x] \leq \dots \leq \text{cheie}_{n[x]}[x] \leq k_{n[x]+1}$

→ gradul unui B-arbore

↳ spăcă nod, fără rădăcină - cel puțin $t-1$ chei



spăcă nod intern - cel puțin t fiu

→ dacă arborele nu este \Rightarrow rădăcină - cel puțin t chei

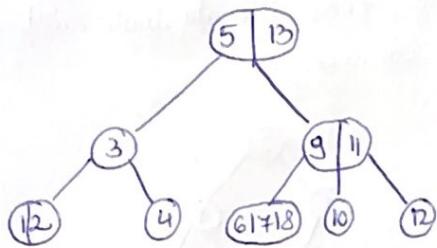
↳ spăcă nod - cel mult $2t-1$ chei

↳ spăcă nod intern - cel mult $2t$ fiu

(21)

(25)

\hookrightarrow um nod cee $2t-1$ elui = nod plin



(Th) Dacă $n \geq 2 \Rightarrow$ și B-arbore eu n chei și grad + ≥ 2.

$h \leq \log_2 \frac{n+1}{2}$ - înălțimea arborelui

Curs 13

→ ARBORI DE INTERVALE

ex: Se dă un vector cu n nr. și operații de genul:
 • adăugare la poz. i val. x ($x < 0$)
 • cercer maximul pe intervalul $[i, j]$

Principiu lui Patog

ex: Se dă un vector cu n nr. Sortați-l cu principiu lui Patog!

- ① găsește minimul
 - ② găsește minimul și-l transformă într-o val. mai mare
 - ③ calculează nouă minim pe aceea zonă
 - ④ de la primul val. de sol
până la ultimul val. de sol
- până la val. minim din zonă în vector

→ Arboare de intervale

$O(n)$ memorie

= arbori cu rădăcină fixată de la $[0, n]$

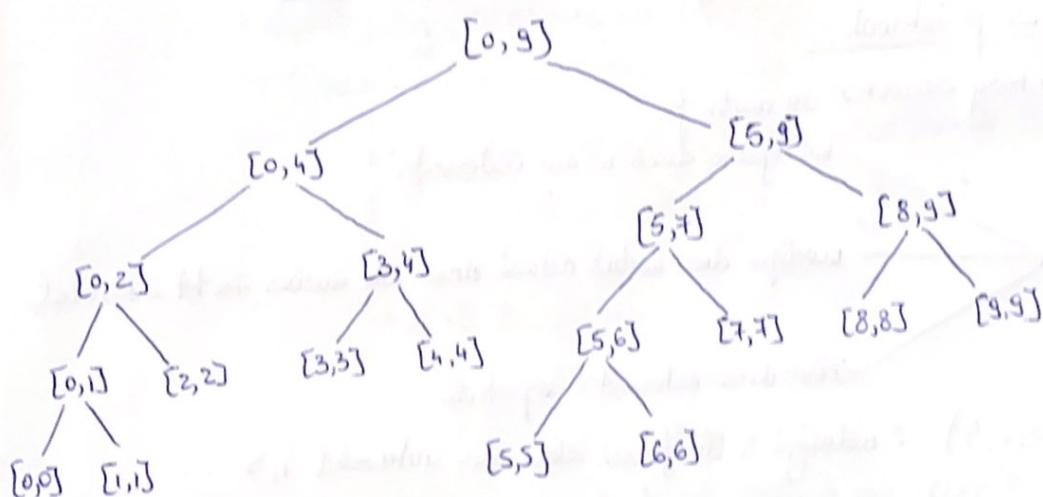
→ pt. un nod care e un interval $[L, R]$

ori patr. $j \times 2$ în vector

fiul stâng: $[L, (L+R)/2]$

cel drept: $[(L+R)/2, R]$

ori patr. $j \times 2 + 1$ în vector



→ operații

- * query pe index
- * query pe interval
- * modificare element
- * modificare interval

(23)

(23)

Query pe index

avem pointeri spre frunze \Rightarrow căsp. direct

permite top down \uparrow

Query pe interval

\rightarrow permitem din rădăcina și mergem recursiv și L și R

\rightarrow dacă intervalul nodului nu se intersectează \Rightarrow STOP

\rightarrow dacă intervalul e inclus complet \rightarrow luăm cîteva și STOP

\rightarrow parcugem $4 * \log n$ noduri

Modificare element

\rightarrow suntem - top-down

\rightarrow suntem

top-down up

\rightarrow coborâm prin rădăcina pînă găsim frunza pe care o modif.

\rightarrow la urcare, update date = (nou valoare)

bottom-up

\rightarrow analog cuas, dar avem indexul referent

Modificare pe interval

\rightarrow mergem recursiv în anumită fuziune

nu opresc dacă nu-are intersecție

\rightarrow modifică doar nodul actual dacă este inclus în tot intervalul

coboră dacă intersecția e parțială

add (3, 1, 3) - adaugă 3 la fiecare elem. din intervalul 1, 3

