- Sortari:

- Countsort:

  Numere intregi mici

  Timp: O(n+max)

  Spatiu: O(max)

- Radixsort:

  In special pentru ordonarea sirurilor de caractere

  Cum sunt utilizate bucket-urile?

  - Elementele sunt sortate după fiecare cifră, pe rând
  - Bucket-urile sunt cifrele numerelor
  - Fiecare bucket `b[i]` conține, la un pas, elementele care au cifra curentă = i

  Timp: O(nlog(max))

  Spatiu: O(n+b)

  LSD = Leas Significant Digit: iterativ rapid

  MSD = Most Signifcant Digit: recursiv

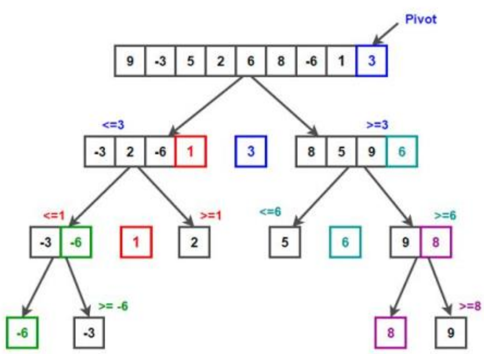
- Quicksort:

  Divide: se împarte vectorul în doi subvectori în funcție de un **pivot x**, astfel încât elementele din subvectorul din stânga sunt **≤ x ≤** elementele din subvectorul din dreapta

  Impera: se sortează recursiv cei doi subvectori

Quick Sort - exemplu

- Pivot ales la coadă
- Contraexemplu ?
  - 1 2 3 4 5 6 7 8 9
- Pivotul în centru
  - 4 3 2 1 **9** 8 7 6 5
  - *4 3 2 1 8 7 6 5* **9**
  - **1** *4 3 2 8 7 6 5* **9**



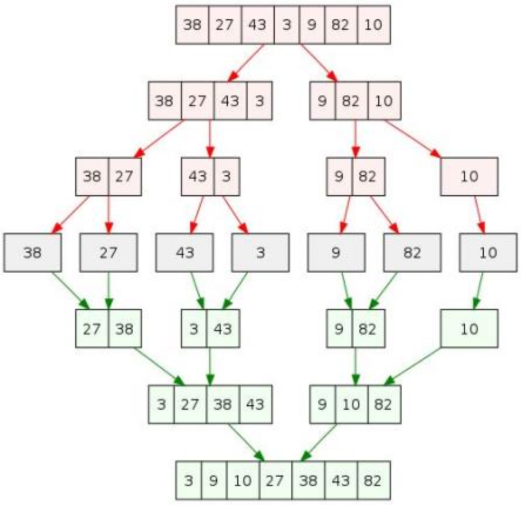
Cum alegem pivotul?

- Primul element
- Elementul din mijloc
- Ultimul element
- Un element random
- Mediana din 3
- Mediana din 5, 7 (**atenție** când vectorul devine mic, facem mult calcul pentru puțin)
- Mediana medianelor

Timp: O(nlogn)

Spatiu: O(logn) -> din cauza recursivitatii

- Mergesort:



Poate fi folosit pentru a afla numarul de inversiuni dintr-un vector
Are nevoie de vector suplimentar si face multe mutari. Quicksort e in place
Time: O(nlogn)
Space: O(n)

! Orice algoritm de sortare care se bazeaza pe comparatii face putin O(nlogn) comparatii.

Heapsort = creare heap, si scoatere elemente unul cate unul

Arbore de intervale sort = Scoatere minim, inlocuirea lui cu numarul maxim

sau

Nodurile vor fi vectori cu elementele sortate (mergesort)

Arbore de cautare binara sort = creare, parcurge in inordine, SRD

Skiplist sort = creare, parcurgere pe nivel de baza

- Liste, Vectori, Stive, Cozi

- Liste:

  Alocare dinamica
  O(1) inserare/stergere oriunde, daca avem pointerul de care avem nevoie
  Nu putem gasi usor al k-lea element
  Grija cu alocare/stergere de memorie
  Simplu/dublu inlantuite
  Circulare

- Array:

  Alocare statica

  Sunt mai rapizi decat listele

  ## Array vs Liste

  Complexitate:

	Liste	Array
Inserare oriunde	În caz bun, O(1)	O(n)
Inserare/ștergere la capăt	O(1)	O(1)
Afişarea celui de-al k-lea element	O(k)	O(1)
Sortare	O(n logn)	O(n logn)
Căutare în structura sortată	O(n)	O(logn)
Redimensionare	O(1)	O(n)

- Vectori:

  Alocare dinamica

  Alocam niste memorie la inceput, redimensionam

- Stive:

  Last In First Out

  Avem acces doar la top

  Operatii de baza:

    Push – adauga element in varf

    Pop – eliminare element din varf

  Operatii suplimentare:

    Size – nr. de elemente

    isEmpty – true daca e, atlfel false

    Peek/top – returneaza valorea din varf

- Cozi:

  First In First Out

  Avem acces la primul si ultimul element

  Operatii de baza:

    Push – adauga element in varf

    Pop – eliminare element din varf

  Operatii suplimentare:

    Size – nr. de elemente

    isEmpty – true daca e, atlfel false

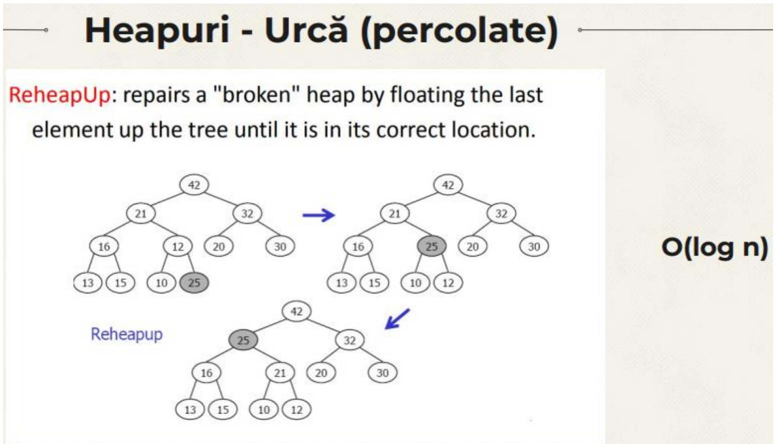
    front – valorea de la inceput, fara sa o stearga

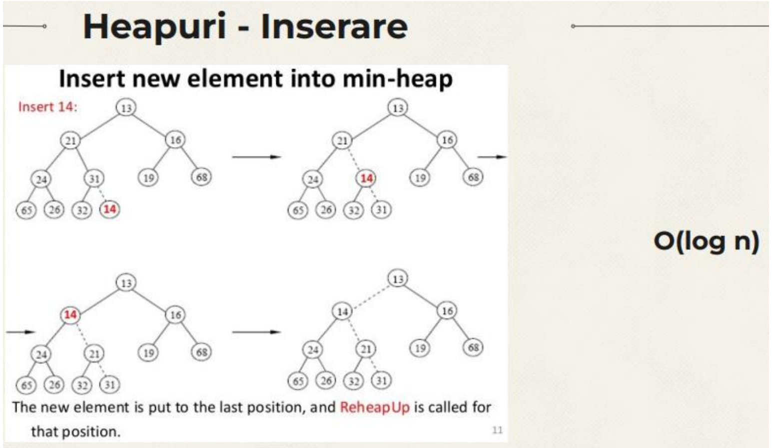
    back – valoarea de la final fara sa o stearga

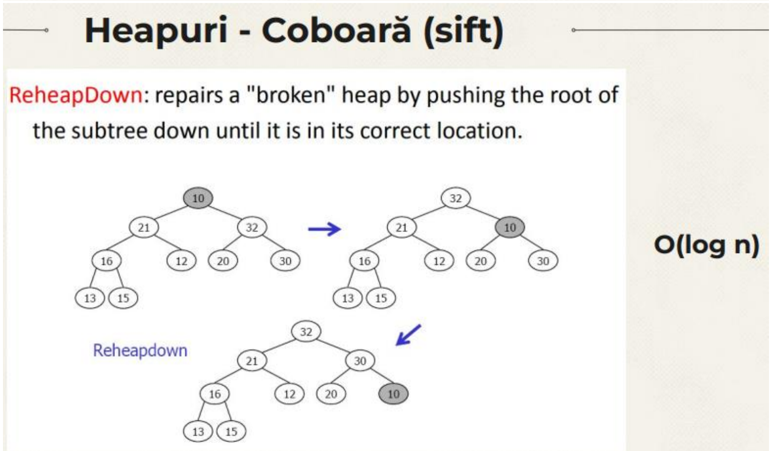
- Deque:
    - Double ended queue (coadă cu două capete)
    - Operații de bază:
        - Push Front
        - Push Back
        - Pop Front
        - Pop Back
    - Operații suplimentare
        - Size()
        - Front()
        - Back()
        - isEmpty()

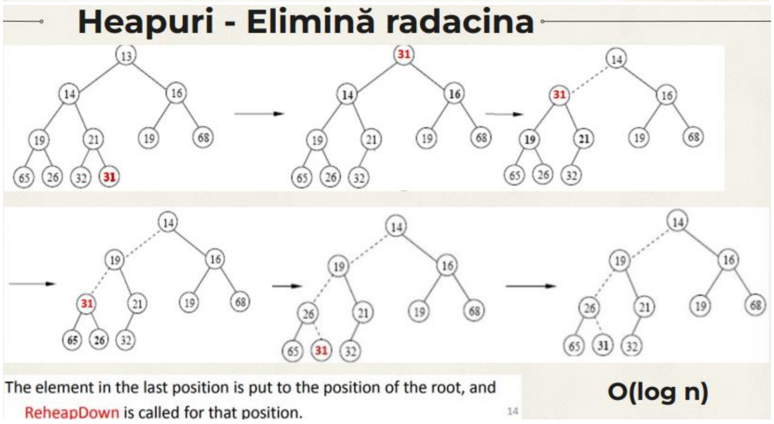
- Heap

- Arbore binar = fiecare nod are cel mult 2 copii
  Arbore binar plin = daca fieare nod are fie 0 fie 2 copii
  Arbore binar complet = daca toate nivelurile sunt complete, mai putin ultimul (completat stanga -> dreapta)
- Abore binar balansat/echilibrat = pentru orice nod diferenta dintre fiul stang si cel drept e maxim 1
- Nr. noduri arbore binar cu inaltime h intre: h si $2^{h+1} - 1$
- ! Un heap e un arbore binar complet
- Parinte = (i-1) / 2
  CopilStanga = 2*i + 1
  CopilDreapta = 2*i + 2







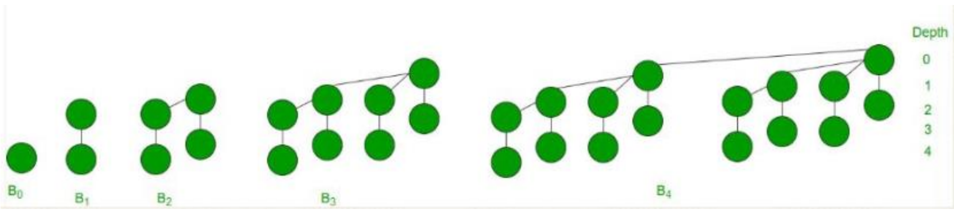


- Heapify:
  1) Inseram n elemente – O(nlogn)
  2) Plecam de la primul element care nu e frunza si facem siftare – O(n)
- Lazy Deletion:
  Marcam nod spre stergere, dar nu il stergem decat cand ajunge in varf
  Cautarea devine O(n)

Operație	Timp Mediu	Cel mai rău caz
Spațiu	O(n)	O(n)
Căutare	O(n)	O(n)
Inserare	O(1) n/2 * 0 + n/4 * 1 + n/8 * 2 …~= 1	O(log n)
Ștergere minim	O(log n)	O(log n)
Căutare minim	O(1)	O(1)
Construcție n elemente	O(n)	O(n)
Uniune (2 heapuri de n elemente)	O(n)	O(n)

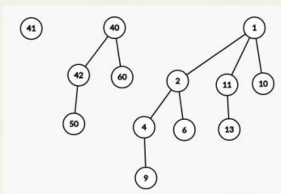
	Căutare Min	Ștergere Min	Inserare	Update	Reuniune
Heap Binar	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Heap Binomial	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$ (amortizat)	$\Theta(\log n)$	$O(\log n)$
Heap Fibonacci	$\Theta(1)$	$O(\log n)$ (amortizat)	$\Theta(1)$	$\Theta(1)$ (amortizat)	$\Theta(1)$

- Arbore binomial:
  - Are exact 2^k noduri
  - Are inaltime k
  - Sunt exact C_i^k noduri de inaltime i
  - Radacina are gradul k

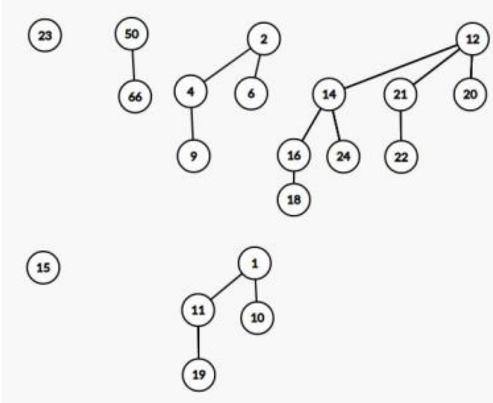


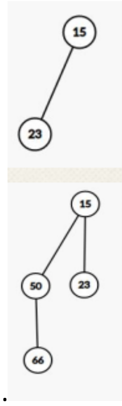
- Heap binomial:
  - Colectie de arbori binomiali, fiecare cu proprietate de heap minim
  - OBS: Exista o singura structura de heap binomial pentru orice marime

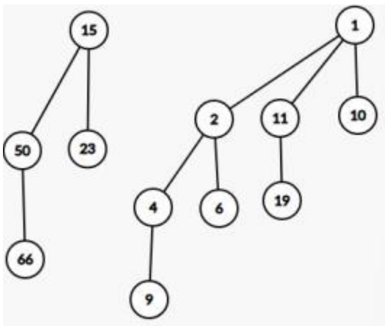
1) Minimul se află în rădăcina unui arbore binomial. Putem parcurge toți arborii binomiali, să ne uităm la rădăcina lor și să reținem minimul → **O(log n)**
2) Totuși, putem ține minte valoarea când facem orice fel de operație și să răspundem în **O(1)**

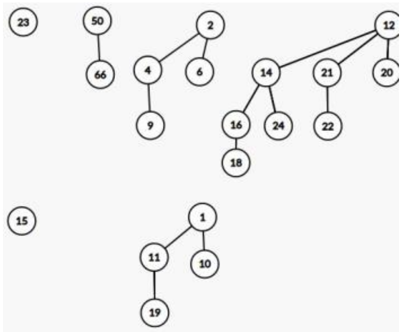


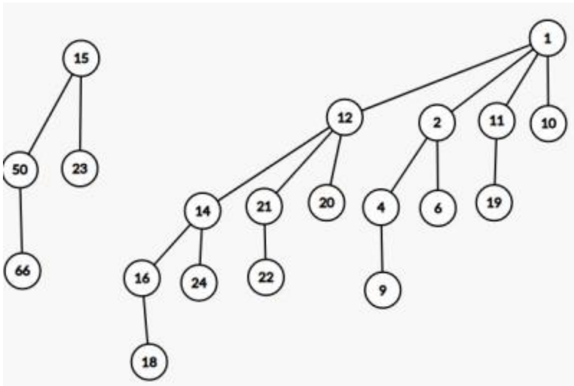
Reuniune:


1.


2.


3.


4.

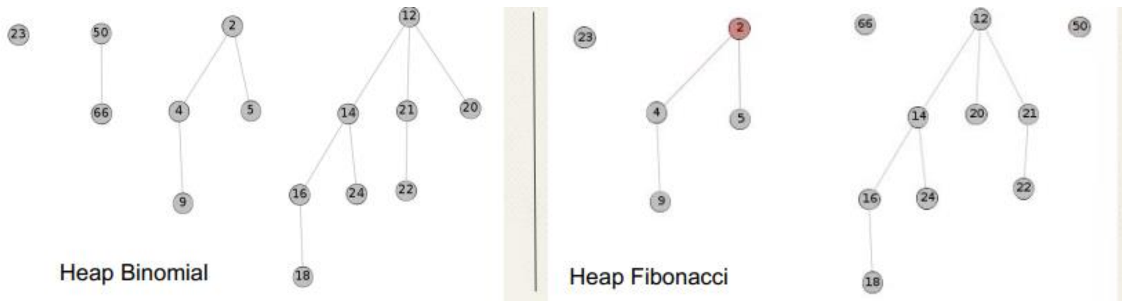

5.

Timp: O(logn)
Reuniunea a doi arbori se face in O(1)

Extragere minim: eliminam minimul, apoi facem reuniune

Inserare: adaugam arbore binomial de marime 1, apoi facem reuniune

- Heap Fibonacci:
  Colectie de arbori care au proprietatea de heap (nu trebuie sa fie binomiali)
  Arborii nu sunt ordonati
  Arborii din componenta au marimi puteri ale lui 2



Inserare:

Cream arbore cu un singur element

Il plasam in stanga radacinii

Nu facem reuniune

Timp: O(1)

Cauta minim:

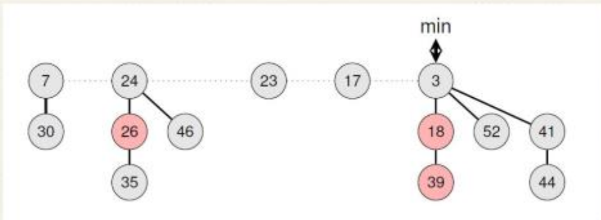
Tinem la fiecare pas pointer spre minim

Timp: O(1)

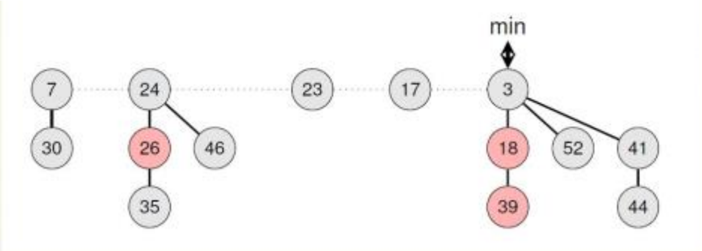
Extragere minim:

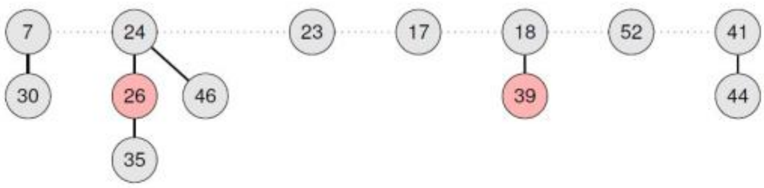
Extragem minim, fiii sai devin arbori liberi

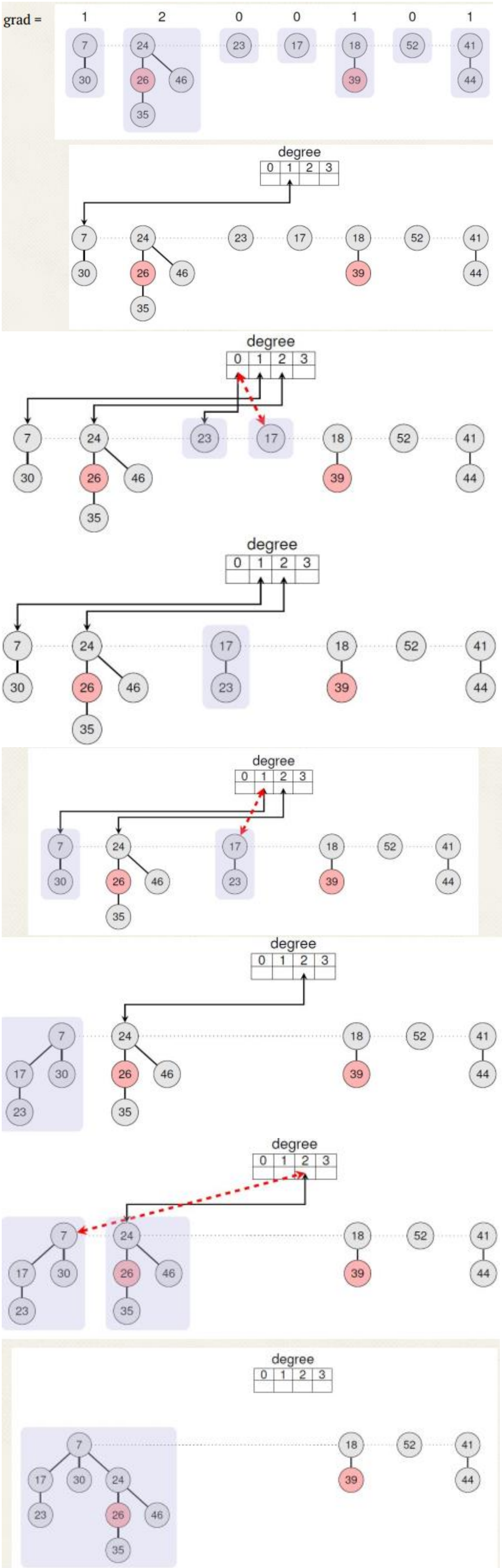
Ca să evităm să avem de mai multe ori cost mare pentru extragerea minimului, vom consolida heapul ("reuniunea" de la heapul binomial).



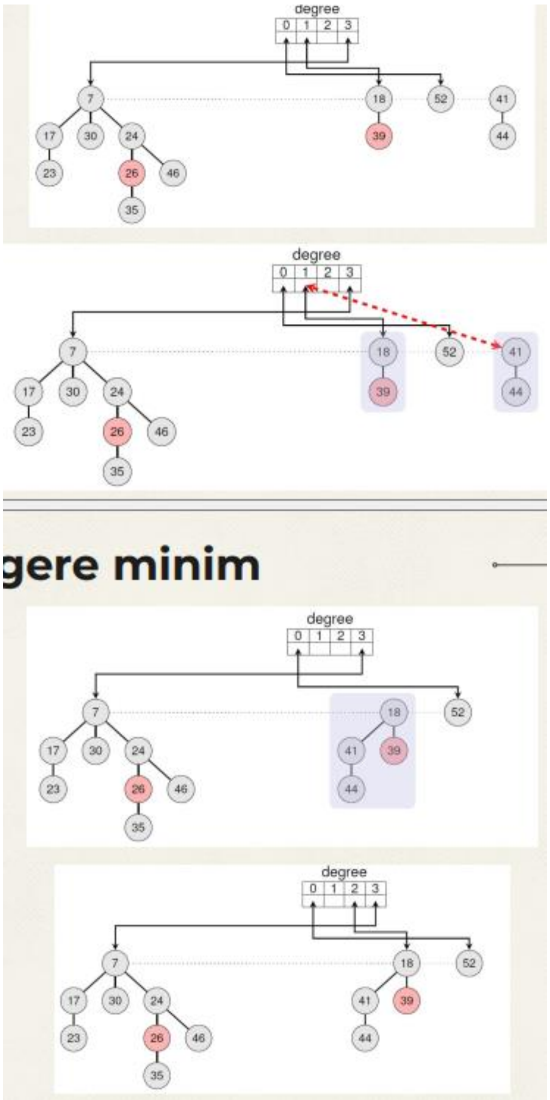
Eliminăm minimul, se creeaza multe "rădăcini"







grad = 1 2 0 0 1 0 1
degree
0 1 2 3
degree
0 1 2 3
degree
0 1 2 3
degree
0 1 2 3
degree
0 1 2 3
degree
0 1 2 3
degree
0 1 2 3



Timp:

O(n) – pentru prima

O(logn) – pentru urmatoarele, daca nu facem alte operatii

O(logn) – amortizat

- Arbori de intervale, RMQ, LCA, LA:

Şmenul lui Batog

Se dă un vector cu n elemente și apoi n operații de genul:

- `1 i j` → care este minimul din intervalul `[i,j]`
- `2 i x` → modificați elementul de pe poziția `i` în `x`

Idee:
Împărțim vectorul în zone de lungime L și calculăm minimul pe fiecare zonă în parte.

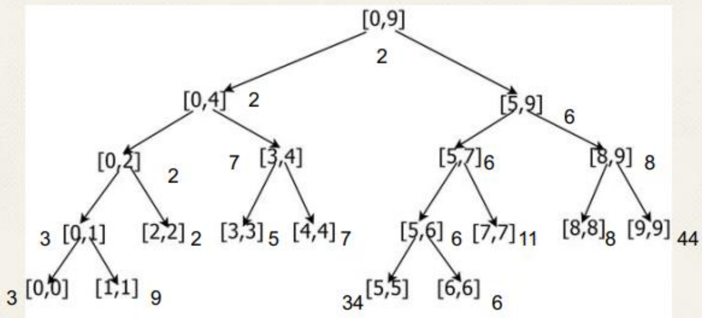
0	1	2	3	4	5	6	7	8
3	9	2	5	7	34	6	11	8
	2			5			6	

Lungime optima = sqrt(n)
Timp: O(sqrt(n))
Pentru update:
Modificam elementul de pe pozitia i
Recalculam proprietatea pe zona respectiva
Sortare: O(n sqrt(n))
Extragem minim, il inlocuim cu un numar maxim
Recalculam zona respectiva
Repetam

- Adaugam la pozitia i valoarea x
- Cerem minim pe interval i,j

0	1	2	3	4	5	6	7	8	9
3	9	2	**5**	**7**	**34**	**6**	11	8	44

minimul!



Reprezentare similară cu heapul:

- Rădăcina (1 de multe ori) are intervalul [0,n) [L,R)
  - Fiul stâng are [L, (L+R)/2]; el are poziția în vector i*2
  - Fiul drept are [(L+R)/2 + 1, R]; el are poziția în vector i*2+1
  - Vectorul poate avea niște elemente lipsă pe ultimul rând (vezi 2 slide-uri mai sus).

În total vectorul are 2*n noduri "active", dar avem nevoie de mai mult de 2*n memorie. 4*n e safe

O(n) memorie.

- Operatii:
  query pe index
  query pe interval
  modificare element
  modificare interval

- Query pe interval
  - Evident, nu luăm toate valorile; ar putea fi liniar
  - Q(1,5) min
  - Pornim din rădăcina și mergem recursiv și L și R
  - Caz I — Dacă intervalul nodului nu se intersectează, oprim
  - Caz II — **Dacă intervalul e inclus complet, luăm info & ne oprim**
  - Câte noduri putem parcurge?
    - Doar 4*log n
    - Coborâm pe o ramură până facem un split
      - După split, în fiecare parte, unul dintre fii va fi ori cazul I, ori cazul II, deci se va coborî pe maxim 2 drumuri până jos.

Modificare element
- Dacă țin suma, pot face top-down
- Dacă țin minim, pot face ori:
  - Top down up
    - coborâm din rădăcină până găsim frunza pe care o modificăm
    - La urcare, facem update tata = min(cei 2 fii)
  - Bottom up
    - Exact ca mai sus, dar avem deja indexul ținut

Modificare pe interval
- Similar cu query pe interval
- Merg recursiv în ambii fii
  - Mă opresc dacă nu am intersecție
  - Modific doar nodul actual dacă este inclus de tot în interval
    - Aici trebuie să ținem în nod o informație suplimentară (toate nodurile cresc cu o anumită valoare)
  - Cobor dacă e intersectie parțială

- LA:
  O(h) – parcurg din tata in tata
  O(1) – pentru fiecare nod retin D[i][j] = stramosul de nivel j al lui i
  dar memorie si preprocesare O(n*h)
  O(sqrt(n)) query, O(n) mem – tin tatal de ordin radical din n
  O(logn) query, O(nlogn) mem – pentru fiecare nod tin tatii de inaltime 1,2,4,8,16,...
  Complexitate: O(nlogn) preprocesare
  O(nlogn) mem
  O(logn) query
- RMQ:

Ținem pentru fiecare element puterile lui 2 și răspundem similar LA în log n.

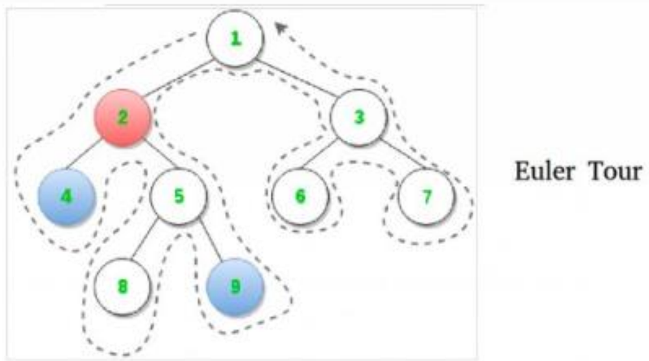
	0	1	2	3	4	5	6	7	8	9
min	3	9	2	8	5	3	8	7	6	11
min2	3	2	2	5	3	3	7	6	6	11
min4	2	2	2	3	3	3	6	6	6	11
min8	2	2	2	3	3	3	6	6	6	11

Caz de baz: RMQ[0][j] = V[j]
In general: RMQ[i][j] = MIN(RMQ[i-1][j], RMQ[i-1][j+2^i - 1]
Query(x,y): MIN(RMQ[p][x], RMQ[p][y-2^p+1]
    unde p e maxim astfel incat 2^p <= j – i +1
Preprocesare si memorie: O(nlogn)
Query: O(1)
OBS: Inafara de min,max,sum pentru arbori de intervale si RMQ merge si CMMDC pentru ca este idempotent

- LCA:

LCA → RMQ

Începem o parcurgere RSD din rădăcină și scriem fiecare nod **de fiecare dată când trecem prin el.**
Pentru fiecare nod, reținem și distanța de la el la rădăcină
Pentru fiecare nod, mai reținem și prima sa apariție în parcurgerea Euler...
De exemplu, pentru 4 e poziția 2, pentru 9 este 7



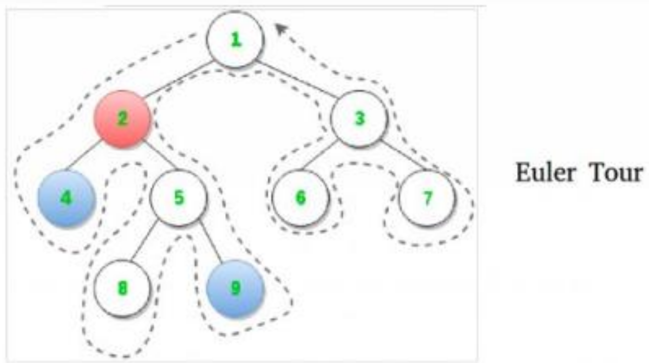
An euler tour of the tree starting from node 1 will yield:

1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1

The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0

LCA(i,j) este RMQ(first[i], first[j])...
LCA(4,9) va fi RMQ pe parcurgerea Euler între primele apariții ale lui 4 și 9
Deci RMQ(2,7)...
RMQ se va face pe vectorul de distanțe, până la rădăcină (2, 7), prin urmare obținem distanța 1 către rădăcina care corespunde nodului 2.
Orice drum între 4 și 9 trece prin 2, dar nu mai sus de 2!



An euler tour of the tree starting from node 1 will yield:

1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1

The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0

- Arbori binari de cautare (BST):

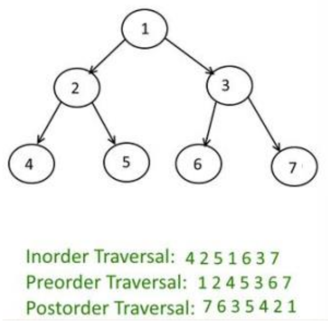
  Arbore binar strict = fiecare nod fie nu are niciun fiu, fie are exact 2

  Parcurgeri:

  Inordine (SRD, stanga radacina dreapta)

  Preordine (RSD, radacina stanga dreapta)

  Postordine (SDR, stanga dreapta radacina)



In acest desen e gresita postordinea (4,5,2,6,7,3,1)

Inaltime:

  Minim = logn, arbore binar complet

  Maxim = n, lant, elementele sunt inserate crescator sau descrescator

Minimul se afla in cel mai din stanga nod

Maximul in cel mai din dreapta

Predecesor:

1) are fiu stanga -> stanga si dupa dreapta full
2) nu are fiu stanga -> primul tata mai mic

Succesor:

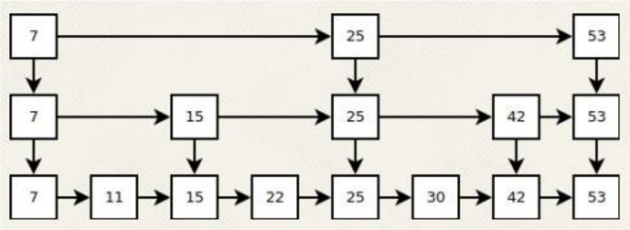
1) are fiu dreapta -> dreapta si stanga full
2) nu are fiu dreapta -> primul tata mai mare

Stergere:

1) nodul e frunza -> sterg
2) are un singur fiu -> il sterg si unesc tatal de fii
3) are 2 fii -> iau succesorul il sterg si il pun in locul elementului

Operație	Complexitate
Căutare	O(h)
Găsire Minim	O(h)
Inserare	O(h)
Succesor / Predecesor	O(h)
Ștergere	O(h)

- Skiplist:



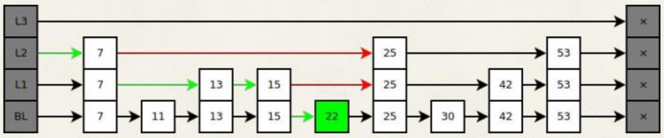
- Daca am avea doar 2 nivele:
  ar fi bine elemente egal departate
  iar distantata dintre elemente = sqrt(n)

Skip Lists - Căutare

1) Începem căutarea cu primul nivel (cel mai de sus)
2) Avansăm în dreapta, până când, dacă am mai avansa, am merge prea departe (adică elementul următor este prea mare)
3) Ne mutăm în următoarea listă (mergem în jos)
4) Reluăm algoritmul de la pasul 2)

Exemplu: search(22)

Complexitate: O(logn)



Skip Lists - Inserare

- Vrem să inserăm elementul x
- x trebuie să fie inserat cu siguranță în nivelul cel mai de jos
- Cum alegem în ce altă listă să fie adăugat?
  - Alegem metoda probabilistică:
    - aruncăm o monedă
      - dacă pică Stema - o adăugăm în lista următoare și aruncăm din nou moneda
      - dacă pică Banul - ne oprim
    - probabilitatea să fie inserat și la nivelul următor: ½
- În medie:
  - ½ elemente nepromovate
  - ¼ elemente promovate 1 nivel
  - ⅛ elemente promovate 2 nivele
  - etc.
- Complexitate: O(logn)

Skip Lists - Ștergere

- Ștergem elementul x din toate listele care îl conțin
- Complexitate: O(logn)

Complexitate: O(logn) pentru opeartii, worst-case O(n)

- Hash-uri:

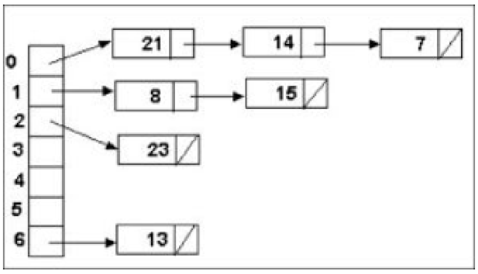
	Inserare	Ştergere min	Ştergere cu pointer	Ştergere fără pointer	Afişare minim	Căuare	Succesor	Afişare sortat
Heap	O(log n)	O(log n)	O(log n)	O(n)	O(1)	**O(n) :(**	**O(n) :(**	O(n logn)
Arbori de căutare echilibraţi	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(n)
Vector	O(1)	O(n)	O(?) O(1) sau O(n)	O(n)	O(n)	O(n)	O(n)	O(n log n)
Listă înlănţuită	O(1)	O(n)	O(1)	O(n)	O(n)	O(n)	O(n)	O(n log n)

Functii de hash:

1) metoda diviziunii: f(k) = k%p, p e un nr. prim
   bun, simplu, ineficient
2) metoda multiplicarii:
   inmultesc cu o constanta mica, si apoi inmultesc cu o putere a lui 2 si iau partea intreaga
   $f(k) = [2^p(k*c)]$

Rezolvare coliziuni:

1) Inlantuire



2) Adresare directa
   Pun pe prima pozitie livera incepand cu f(k)
   Caut: pana il gasesc sau dau de primul loc liber
   Consider vectorul circular
   Conditie necesara: mai mult spatiu decat elemente
   La stergere: trebuie sa tinem un placeholder ca altfel nu gasim elementele

Trade-off: cu cat e mai rapida functia de hash cu atat folosim mai multa memorie

- Pattern Matching cu Rolling Hash, algoritmul Rabin Karp:

  Cautam sir A in sir B, |A| < |B|

  1. Calculăm hash-ul pentru șirul mai mic
  2. Calculăm hash-ul pentru toate șirurile de aceeași lungime din șirul mai mare