* Sortari:
* Countsort:

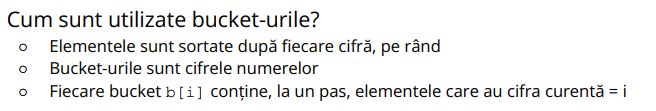
Numere intregi mici

Timp: O(n+max)

Spatiu: O(max)

* Radixsort:

In special pentru ordonarea sirurilor de caractere



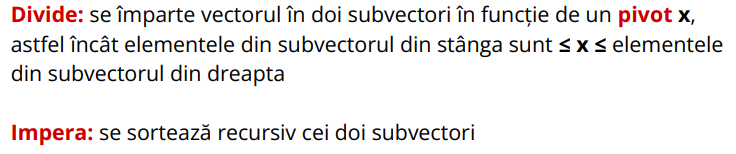
Timp: O(nlog(max))

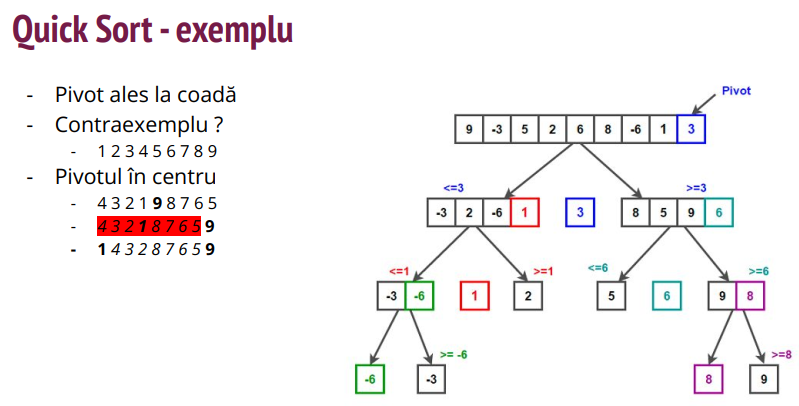
Spatiu: O(n+b)

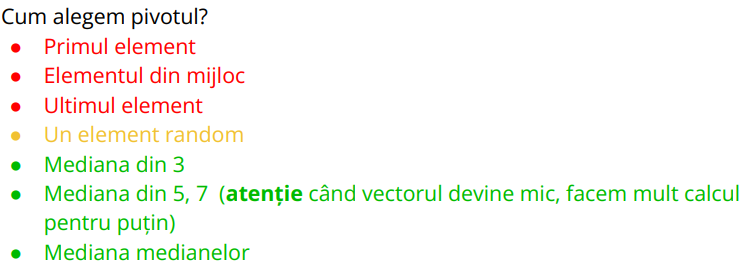
LSD = Leas Significant Digit: iterativ rapid

MSD = Most Signifcant Digit: recursiv

* Quicksort:



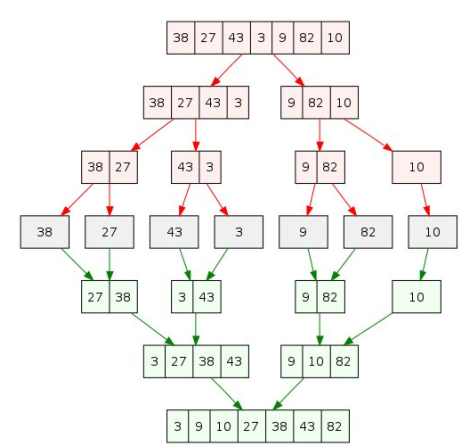




Timp: O(nlogn)

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

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



* 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:

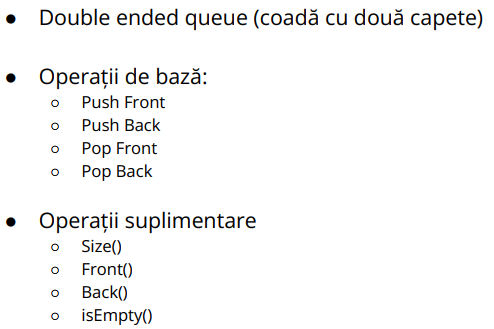
Size – nr. de elemente

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:



* Heap
* Arbore binar = fiecare nod are cel mult 2 copii

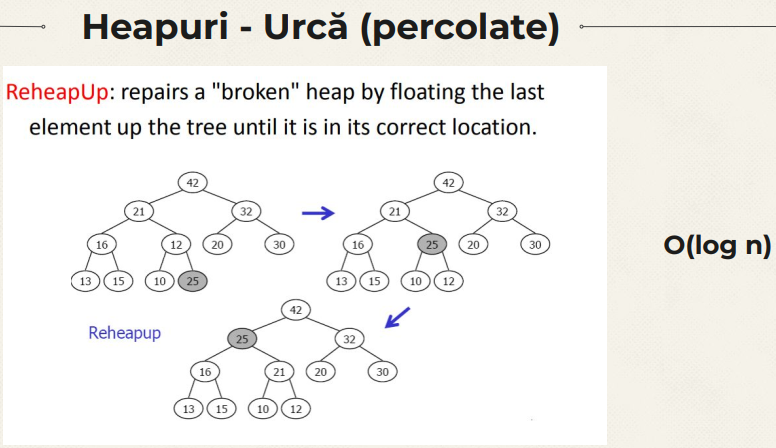
Arbore binar plin = daca fieare nod are fie 0 fie 2 copii

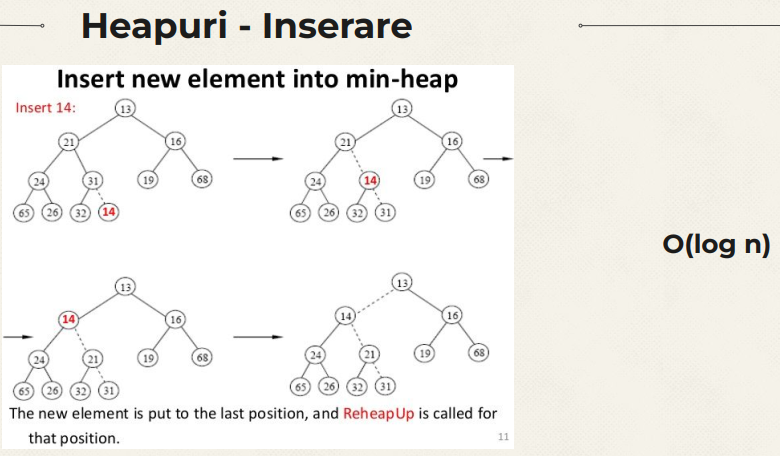
Arbore binar complet = daca toate nivelurile sunt complete, mai putin ultimul (completat stanga -> dreapta)

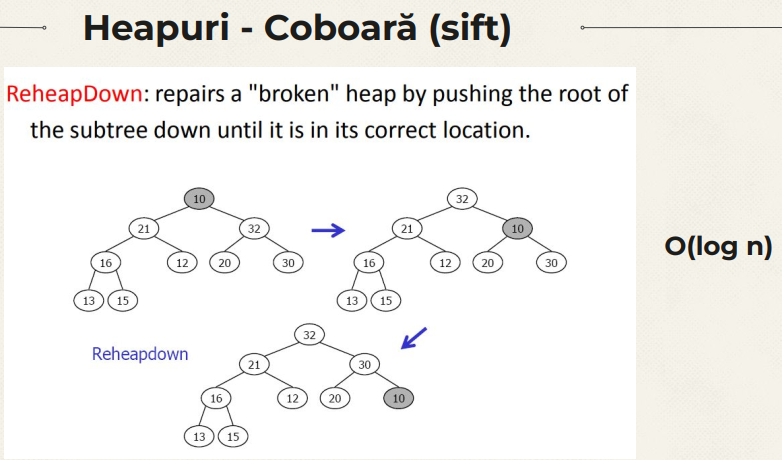
* Abore binar balansat/echilibrat = pentru orice nod diferenta dintre fiul stang si cel drept e maxim 1
* Nr. noduri arbore binar cu inaltime h intre: h si 2h+1 – 1
* ! Un heap e un arbore binar complet
* Parinte = (i-1) / 2

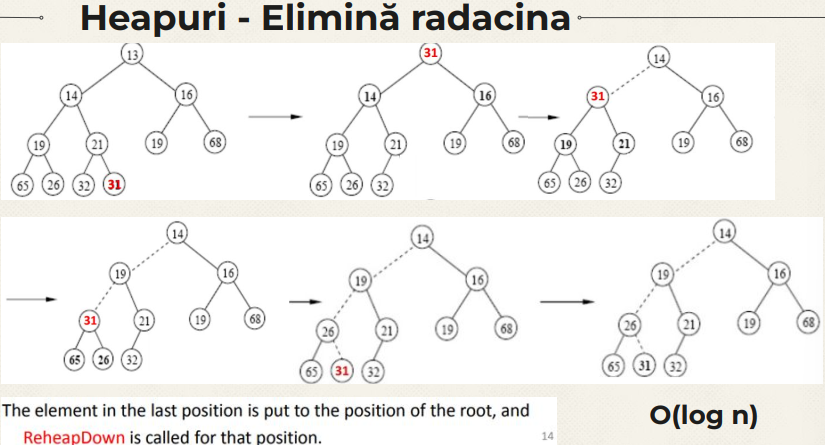
CopilStanga = 2\*i + 1

CopilDreapta = 2\*i + 2









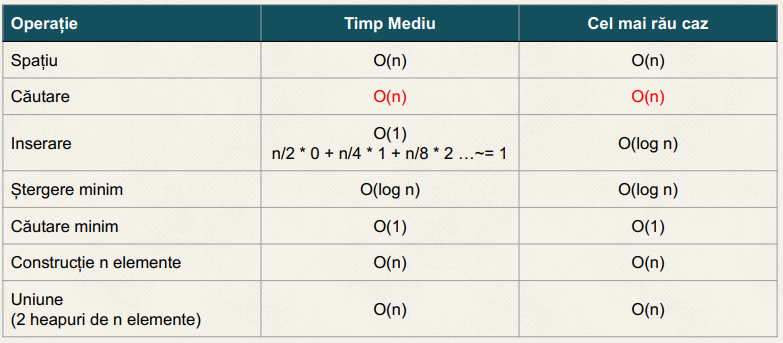
* Heapify:

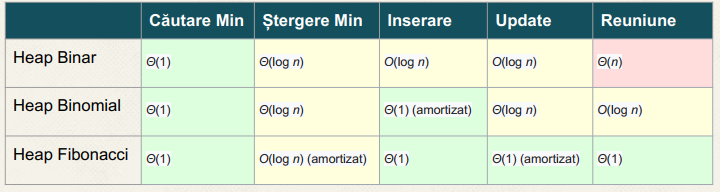
1. Inseram n elemente – O(nlogn)
2. Plecam de la primul element care nu e frunza si facem siftare – O(n)

* Lazy Deletion:

Marcam nod spre stergere, dar nu il stergem decat cand ajunge in varf

Cautarea devine O(n)





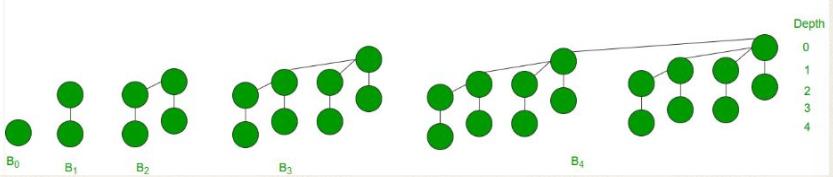
* Arbore binomial:

Are exact 2k noduri

Are inaltime k

Sunt exact Cik noduri de inaltime i

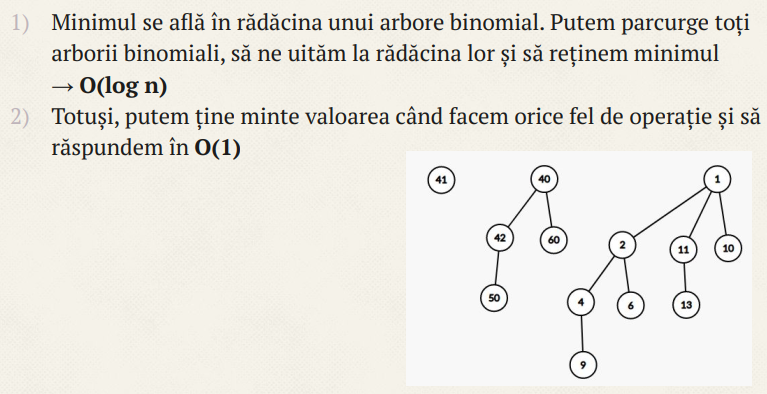
Radacina are gradul k



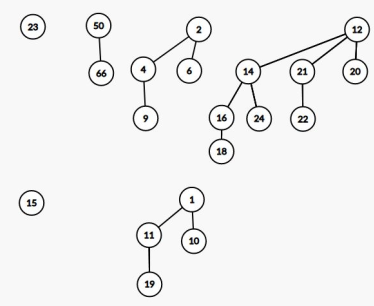
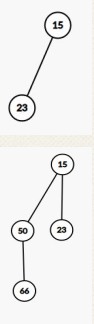
* Heap binomial:

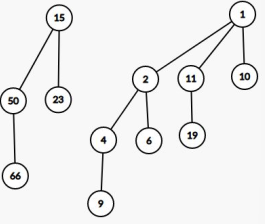
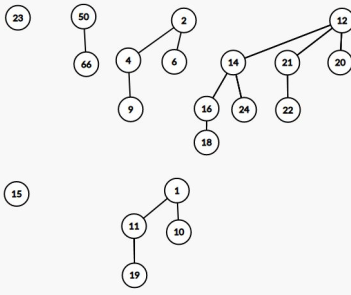
Colectie de arbori binomiali, fiecare cu proprietate de heap minim

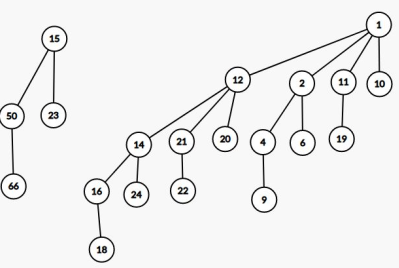
OBS: Exista o singura structura de heap binomial pentru orice marime



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

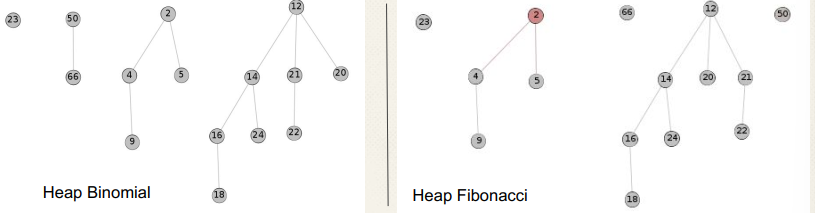
Inserare: adaugam arbore binomial de marime 1, apoi facem reuniune

* 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)

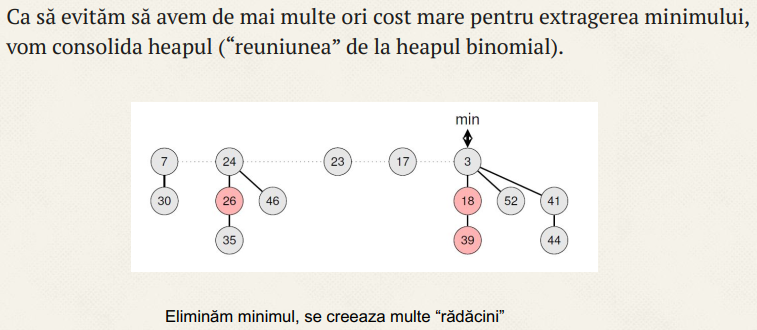
Cauta minim:

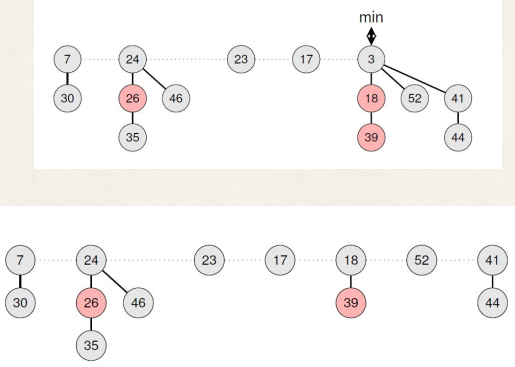
Tinem la fiecare pas pointer spre minim

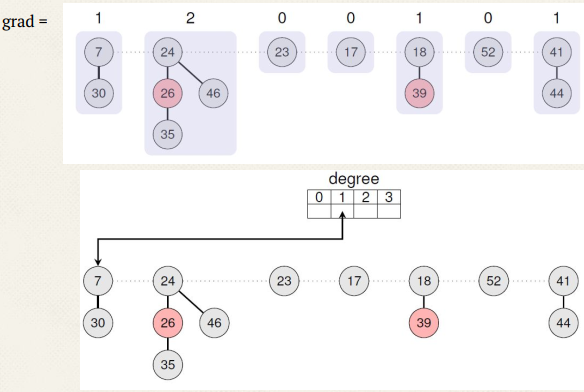
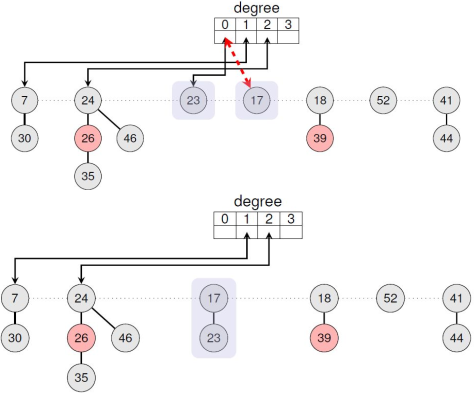
Timp: O(1)

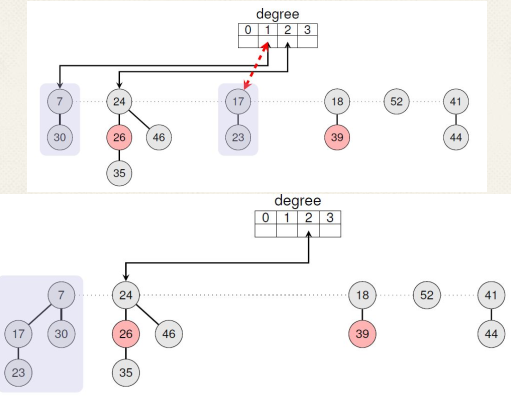
Extragere minim:

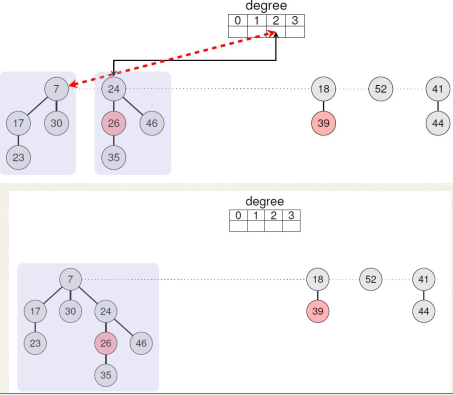
Extragem minim, fiii sai devin arbori liberi

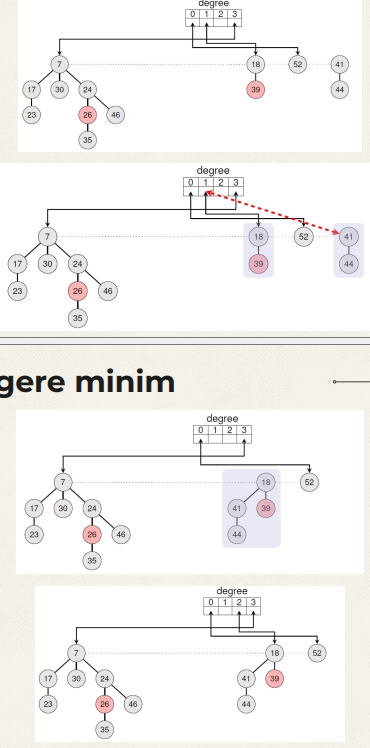










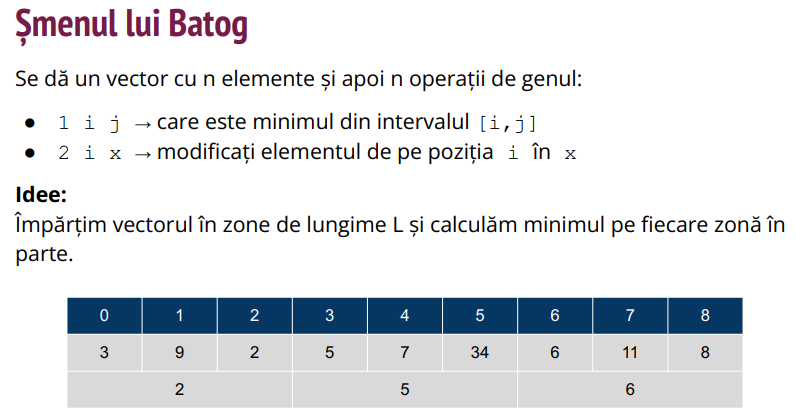
Timp:

O(n) – pentru prima

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

O(logn) – amortizat

* Arbori de intervale, RMQ, LCA, LA:



Lungime optima = sqrt(n)

Timp: O(sqrt(n))

Pentru update:

Modificam elementul de pe pozitia i

Recalculam proprietatea pe zona respectiva

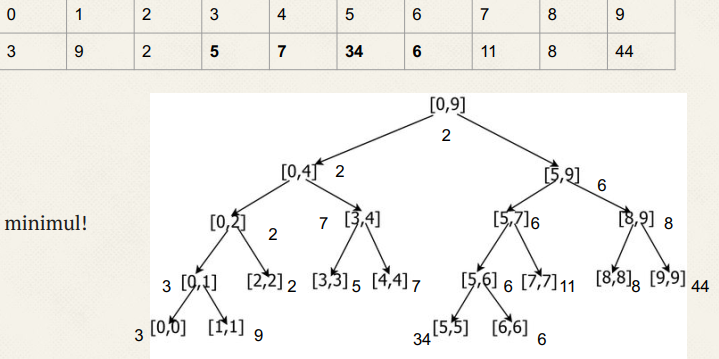
Sortare: O(n sqrt(n))

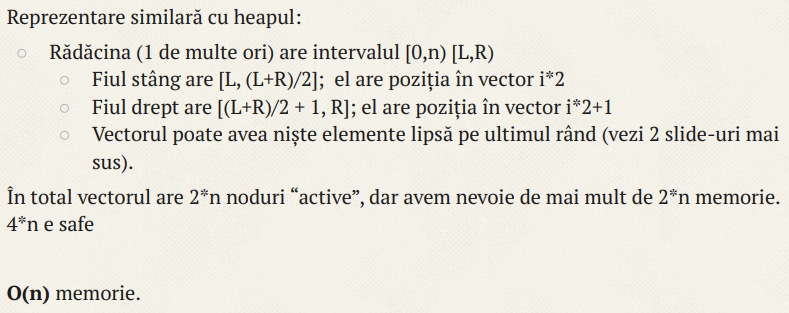
Extragem minim, il inlocuim cu un numar maxim

Recalculam zona respectiva

Repetam

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





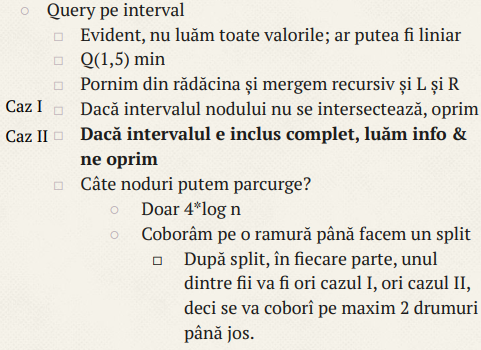
* Operatii:

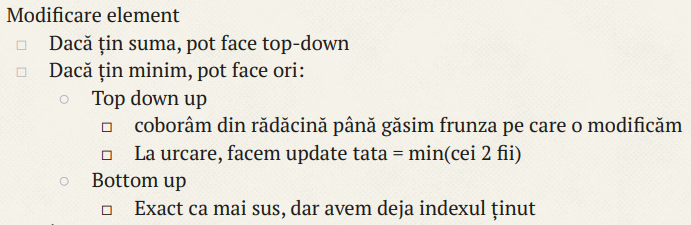
query pe index

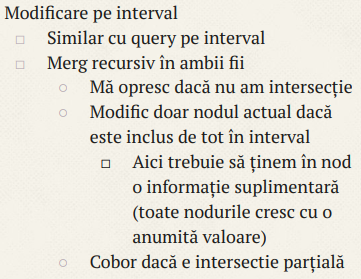
query pe interval

modificare element

modificare interval







* 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

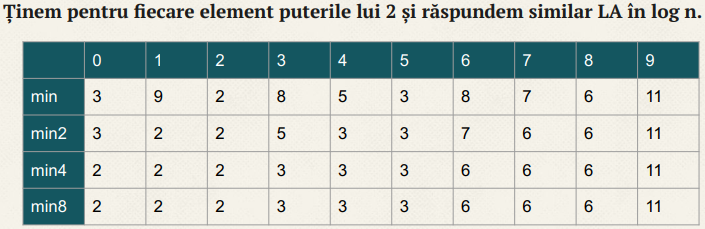
O(logn) query, O(nlogn) mem – pentru fiecare nod tin tatii de inaltime 1,2,4,8,16,...

Complexitate: O(nlogn) preprocesare

O(nlogn) mem

O(logn) query

* RMQ:



Caz de baz: RMQ[0][j] = V[j]

In general: RMQ[i][j] = MIN(RMQ[i-1][j], RMQ[i-1][j+2i - 1]

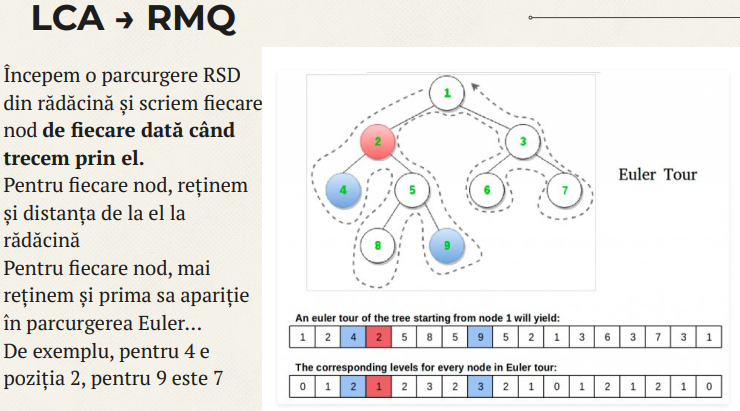
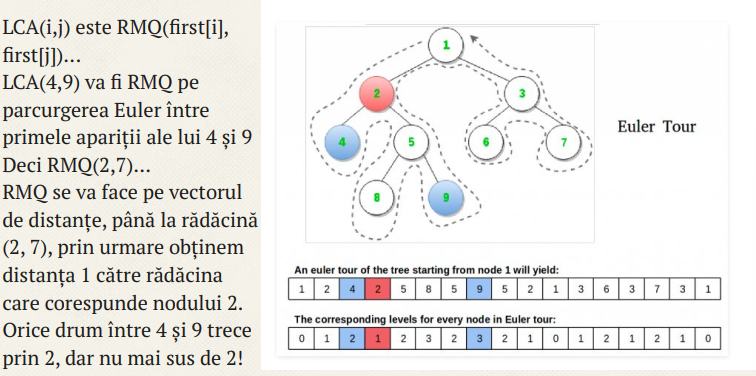
Query(x,y): MIN(RMQ[p][x], RMQ[p][y-2p+1]

unde p e maxim astfel incat 2p <= 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:  
   
* Arbori binari de cautare (BST):