Algoritmi si Structuri de Date

- seminar III -

Square Root Decomposition.

Este o tehnica de programare folosita de obicei cand avem o multime de elemente care se pot modifica si apoi trebuie sa raspundem unor intrebari legate de intreaga multime.

Problema: Se da un sir care contine N numere intregi si M operatii de doua tipuri:
- operatie de tipul 1 a b: afisati care este suma elementelor aflate intre pozitiile a si b;
- operatie de tipul 2 a b: valoarea elementului de pe pozitia a va deveni b.

Input:	Output:
12 7	66
2 9 0 5 1 8 7 3 10 4 11 6	30
1 0 11	43
2 2 5	68
1 0 5	
1 5 10	
2 6 4	
1 4 7	
1 0 11	

Daca primim doar operatii de tipul 1, putem gasi un algoritm de complexitate O(N + M) folosind un vector de sume partiale.

Rezolvare: Vom imparti sirul initial in parti de $\sqrt{N}$ si pentru fiecare parte vom memora care este suma din acea parte. Cand vom modifica un element, vom modificam si suma care se afla in partea respectiva.

Vom indexa vectorii de la zero.

Observatie! Daca folosim vectori, pentru a gasi “partea” care trebuie modificata, va trebui sa impartim indexul elementului la $\sqrt{N}$.

2	9	0	5	1	8	7	3	10	4	11	6
11			14			20			21		

Cand primim o operatie de tip 1 (trebuie sa afisam suma elementelor intre a si b), vom parcurge vectorul de jos si vom afisa suma elementelor din acel vector.
Daca primim de exemplu operatia 1 **5** **10**:

- vom afla care este partea din care face parte primul element (**5** / 3 = 1)
- vom merge insuma liniar elementele pana cand ajungem la o parte noua (S = 8)
- ajungem la noua parte (6 % $\sqrt{N}$ == 0), verificam daca putem insuma direct noua parte (trecand peste $\sqrt{N}$ elemente): (6 + $\sqrt{N}$ - 1= 8 < **10**),S = 8 + 20 = 28
- verificam din nou daca putem sari elemente: 9 + $\sqrt{N}$ = 12 > **10**, nu putem, asadar insumam liniar restul elementelor S = 28 + 4 + 11 = 43.

Cand primim o operatie de tip 2 (trebuie sa modificam un element), vom inlocui elementul respectiv si vom actualiza vectorul de jos prin diferenta.
Exemplu: Operatia 2 2 5

2	9	0	5	1	8	7	3	10	4	11	6
11			14			20			21		

Pe pozitia 2 se afla elementul 0. Partea corespunzatoare elementului de pe pozitia 2 este 2 / $\sqrt{N}$ = 2 / 3 = 0 (prima parte). Vom inlocui vechiul element **0** cu noul element **5**:

2	9	**5**	5	1	8	7	3	10	4	11	6
11			14			20			21		

Si apoi vom actualiza prima parte (vom parcurge elementele primei parti, vom calcula suma lor si vom actualiza cel de-al doilea vector). Diferenta este **5** - **0**, asadar vom avea:

2	9	5	5	1	8	7	3	10	4	11	6
16			14			20			21		

Complexitate operatii.

Cand construim al doilea vector vom parcurge fiecare parte si vom salva suma. Constructia celui de-al doilea vector: O(n).

Cand primim o instructiune de tip 1, parcurgem doar cel de-al doilea vector, care contine maxim N / $\sqrt{N}$ = $\sqrt{N}$ elemente. Complexitate operatie 1: O($\sqrt{N}$).

Cand primim o instructiune de tip 2, modificam un element (O(1) prin diferenta sau O($\sqrt{N}$) prin recalculare) si actualizam o parte. Complexitate operatie 2: O(1).

Complexitate finala: O (N + M * $\sqrt{N}$)

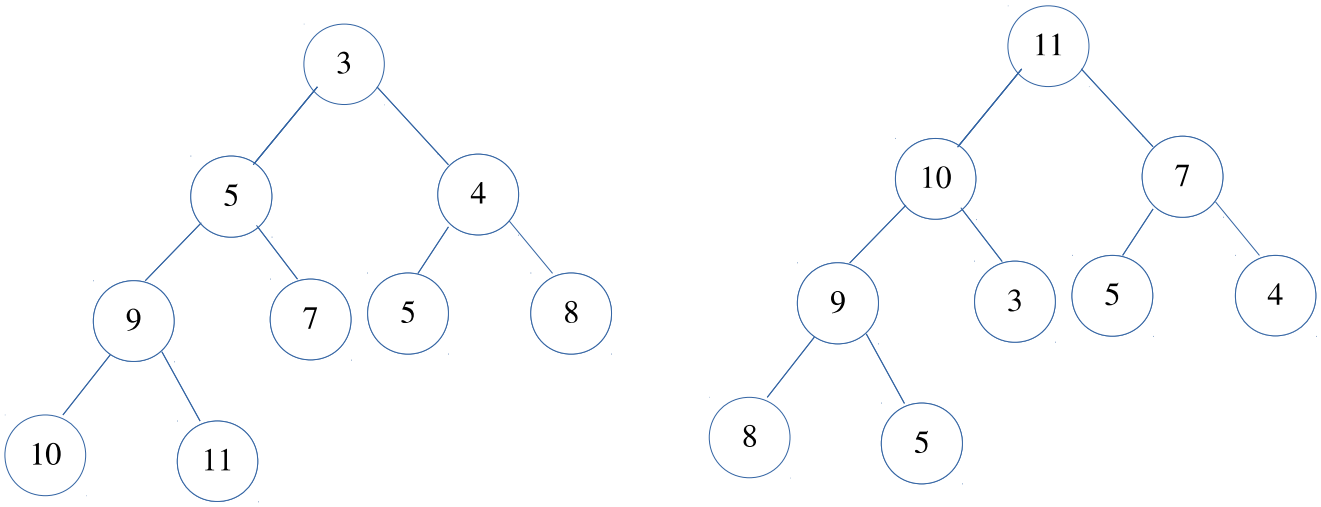
Heap.

Heap-ul (sau in romana ansamblu) este o structura de date abstracta cu ajutorul careia puteti raspunde in timp constant la intrebari de tipul “care este elementul minim/maxim din multime”.

Un heap este de obicei implementat ca un arbore binar complet (toate nivelurile sunt contin un numar maxim de noduri, mai putin ultimul nivel care se completeaza de la stanga la dreapta) in care valoarea fiecare fiu al unui nod este mai mare/mica decat valoarea din nodul respectiv.

Pentru ca aceasta relatie (<= / >=) este o relatie tranzitiva, valoarea din radacina arborelui este cel mai mic/mare element din multime.

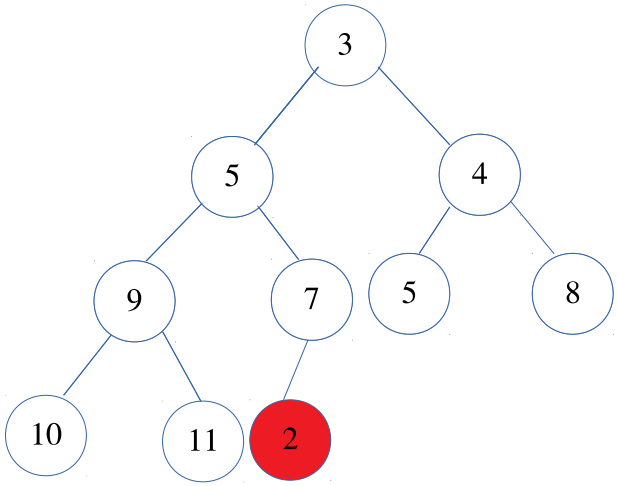
Exemplu: Min-heap Max-heap



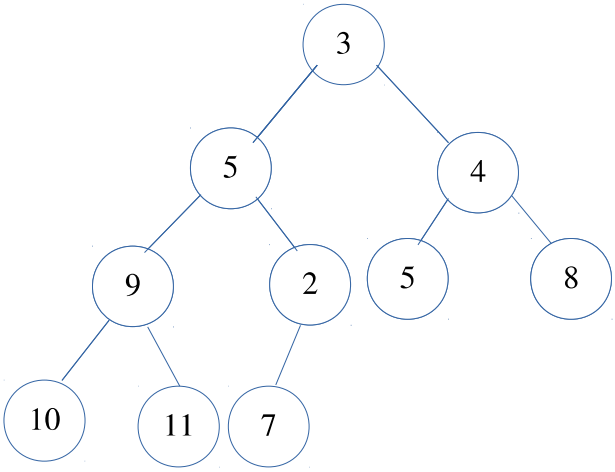
Operatii (min-heap).

- Afisare element minim. Afisam valoarea radacinii.
- Adaugare element in heap. Cand introducem un element in heap, il vom insera ca ultimul element al arborelui. Dupa aceea, vom folosi o functie SiftUp, care va compara valoarea acestuia cu valoarea tatalui. Daca tatal are o valoare mai mare, vom interschimba elementele si vom apela recursiv functia pentru tata.

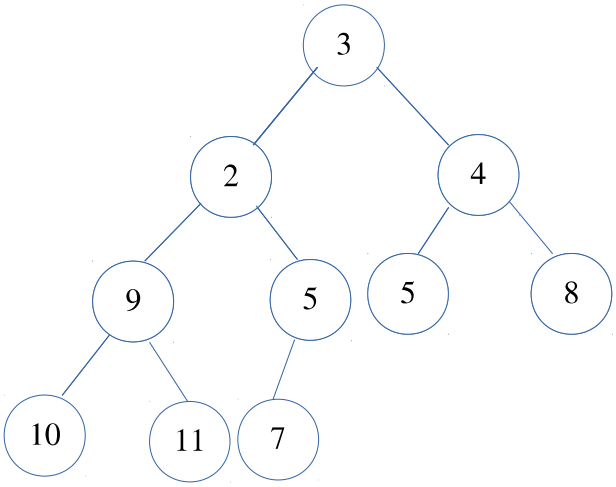
  Ca exemplu, vom insera elementul 2 in min-heap-ul de mai sus.



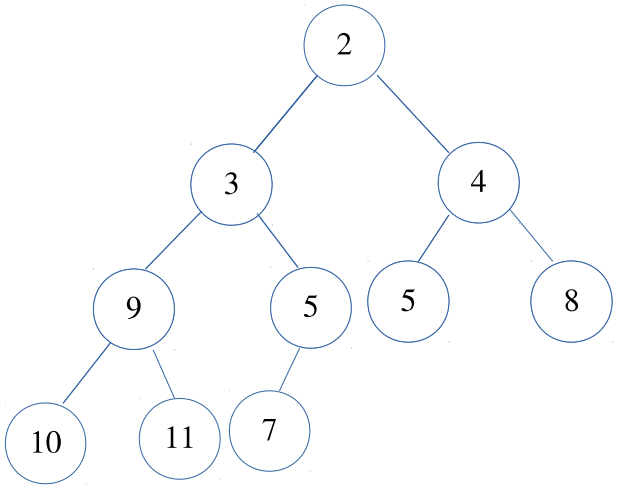
2 < 7, interschimbam valorile



2 < 5, interschimbam valorile

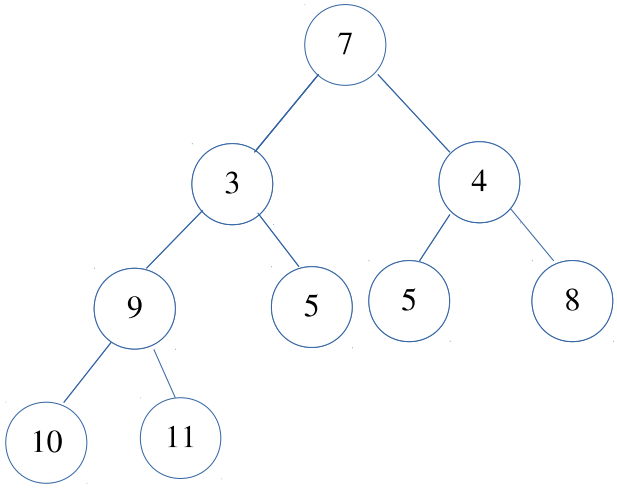


2 < 3, interschimbam valorile

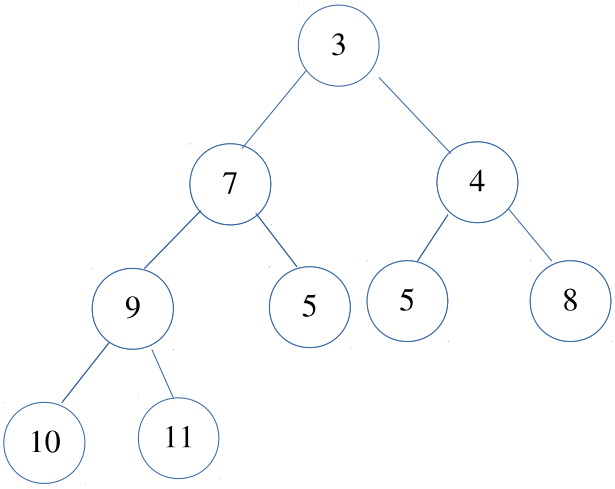


- Stergere element minim din heap (decapitare). Pentru a sterge cel mai mic element din heap, in primul rand vom interschimba valoarea din radacina cu valoarea ultimului element din heap si apoi vom apela functia SiftDown pe radacina, care are rolul de a “impinge” valoarea respectiva cat mai jos.

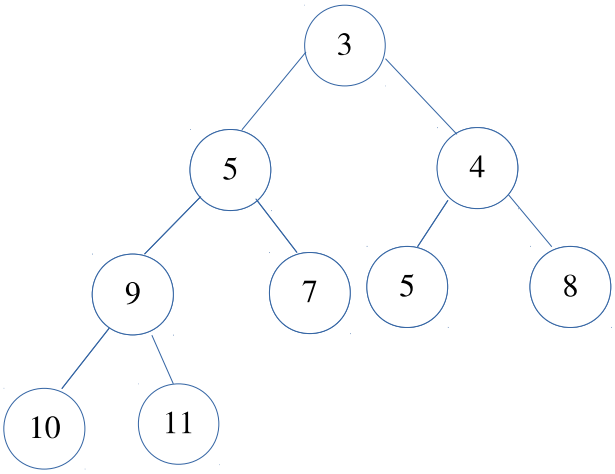
  Exemplu: Vom decapita heap-ul de mai sus.



- se compara 7 cu 3 si cu 4, si se va inlocui cu 3 pentru a pastra proprietatea de heap -



- se compara 7 cu 9 si cu 5 si se va inlocui cu 5 -



Implementare.

Pentru a facilita implementarea algoritmilor SiftUp si SiftDown vom salva heap-ul intr-un vector indexat de la 0, in care pe prima pozitie se afla valoarea din radacina, pe pozitiile 1 si 2 se afla cei doi fii ai radacinii, pe pozitiile 3 si 4 se vor afla fiii nodului de pe pozitia 1, pe 5 si 6 fiii nodului aflat pe pozitia 2, etc.

Min-heap-ul initial poate fi salvat astfel:

0	1	2	3	4	5	6	7	8
3	5	4	9	7	5	8	10	11

Putem observa ca, daca primim un index, putem afla pe ce pozitie se afla tatal sau ((idx - 1) / 2) si de asemenea pe ce pozitii se afla fiii sai (2 * idx + 1, respectiv 2 * idx + 2).

Exemplu. Nodul cu valoarea 9 se afla pe pozitia 3.

Tata(3) = (3 - 1) / 2 = 1
FiuStang(3) = 3 * 2 + 1 = 7
FiuDrept(3) = 3 * 2 + 2 = 8
Site interactiv: https://www.cs.usfca.edu/~galles/visualization/Heap.html

Algoritm in pseudocod:

```
def sift_up(heap, idx):
    if idx == 0:          # daca am ajuns la radacina
        return            # terminam apelul (nu putem urca mai sus)

    father = (idx - 1) / 2              # calculam indexul tatalui
    if heap[father] > heap[idx]:        # daca tatal e mai mare decat fiul
        heap[father], heap[idx] = heap[idx], heap[father]   # interschimbam
        sift_up(heap, father)                       # apelam recursiv pentru tata

def sift_down(heap, idx):
    n = len(heap)

    swap_son = idx              # variabila in care salvam idx pt interschimbare
    left_son = 2 * idx + 1
    right_son = 2 * idx + 2

    if left_son < n and heap[left_son] < heap[swap_son]:      # daca exista fiu st
        swap_son = left_son       # si este mai mic ca tatal, updatam variabila
    if right_son < n and heap[right_son] < heap[swap_son]:  # daca exista fiu dr
        swap_son = right_son      # si este mai mica ca tatal + fiul st, updatam

    heap[idx], heap[swap_son] = heap[swap_son], heap[idx] # interschimbare
    if swap_son != idx:                         # daca a avut loc o interschimbare
        sift_down (heap, swap_son)      # apelam recursiv pentru fiul respectiv
```

Avand cele doua functii ajutatoare, adaugarea si stergerea din heap se fac foarte usor:

```
def find_min(heap):
    return heap[0]              # returnam valoarea radacinii

def insert(heap, x):
    heap.append(x)              # adaugam elementul x la finalul vectorului
    sift_up(heap, len(heap) - 1)    # urcam elementul pe pozitia corecta

def delete_min(heap):
    heap[0], heap[-1] = heap[-1], heap[0] # interschimbam val rad. cu ultimul el
    heap.pop()                          # stergem ultimul element
    sift_down(heap, 0)                  # coboram elementul in heap

def delete_at_idx(heap, idx):
    heap[idx], heap[-1] = heap[-1], heap[idx]
    heap.pop()
    sift_up(idx)
    sift_down(idx)
```

Complexitate operatii.

Daca gasim complexitatile functilor sift_up si sift_down putem calcula usor si complexitatile celorlalte functii. Functia sift_up porneste de la un element aflat pe ultimul nivel in arbore si urca in cel mai rau caz pana la radacina. Inaltimea unui arbore binar complet cu n elemente este maxim $\log_2(n)$ (nivelul k are 2^{k-1} noduri, daca arborele ar fi complet, ar avea $1 + 2 + 4 + \ldots + 2^{h-1} = 2^h$ noduri; $n = 2^h \Rightarrow h = \log_2(n)$).
Complexitate sift_up: O(log n)
Complexitate sift_down: O(log n)

Exercitii.

1. Construiti un min-heap folosind valorile: 15 7 10 11 9 5 8 1; Decapitati heap-ul.
2. Care din urmatoare elemente poate fi considerat ansamblu si de ce fel?
   - 10 4 9 1 3 8 7 0 5
   - 1 4 5 10 14 6 7 11
   - 13 2 10 1 0 9 8