

# SDA - Seminar - TAD Colecție

---

- **Cuprins:**

- Convenții
- TAD Colecție - definire și specificare
- TAD Iterator - specificare
- Exemplificare reprezentare
- Implementare Python

## Convenții

- Algoritmii vor fi descriși în **Pseudocod**
- Elementele din containere vor fi elemente generale/abstracte
  - **TElement**
    - Operație de atribuire:  $e_1 \leftarrow e_2$
    - Verificarea egalității:  $e_1 = e_2$
  - **TComparabil**
    - Pentru containere ordonate
    - Permite inclusiv aplicarea unor relații de ordine:  $<$ ,  $>$ , etc

## TAD Colecție

### Definire

- Spre deosebire de cazul mulțimii, **elementele nu sunt neapărat distincte**, putându-se, deci, repeta
- Precum și în cazul mulțimii, **ordinea elementelor este irelevantă**
  - Nu există poziții într-o colecție. Așadar, operațiile colecției nu primesc poziții ca parametri și nu returnează poziții.
  - Elementele adăugate nu sunt stocate în ordinea adăugării (cel puțin, nu avem această garanție)

De exemplu, dacă adăugăm într-o colecție elementele 1, 3, 2, 6, 2, 5, 2, la afișarea conținutului colecției i, orice ordine a elementelor e posibilă:

- 1, 2, 2, 2, 3, 6, 5
- 1, 3, 2, 6, 2, 5, 2
- 1, 5, 6, 2, 3, 2, 2
- etc...

## Specificare

### 1) Definirea domeniului:

$\mathcal{C} = \{c \mid c \text{ este o colecție cu elemente de tip TElement}\}$

### 2) Specificarea interfeței:

#### creeaza(c)

pre : adevărat  
post:  $c \in \mathcal{C}$ ,  $c$  este colecția vidă ( sau  $\dim(c) = 0$  )

#### distruge(c)

pre:  $c \in \mathcal{C}$   
post: colecția  $c$  a fost distrusă

#### adauga(c, e)

pre:  $c \in \mathcal{C}$ ,  $e$ : TElement post:  $c' \in \mathcal{C}$ ,  $c' = c \cup \{e\}$  ( sau  $c' = c \oplus \{e\}$  )

#### sterge(c, e)

pre:  $c \in \mathcal{C}$ ,  $e$ : TElement  
post:  $c' \in \mathcal{C}$ ,  $c' = c \setminus \{e\}$  (**OBS:** se sterge o singură apariție a elementului  $e$ )

#### cauta(c, e)

pre:  $c \in \mathcal{C}$ ,  $e$ : TElement  
post:  $cauta = \begin{cases} \text{adevărat, dacă } e \in c \\ \text{fals, altfel} \end{cases}$

#### dim(c)

pre:  $c \in \mathcal{C}$   
post: dim = numărul total de elemente din  $c$

#### iterator(c, i)

pre:  $c \in \mathcal{C}$   
post:  $i \in I$ ,  $i$  este iterator pe  $c$  și  $i$  referă un prim element din  $c$

## TAD Iterator

## Specificare

### 1) Definirea domeniului:

$I = \{i \mid i \text{ este iterator pe } c \in \mathcal{C}\}$

## 2) Specificarea interfeței:

creeaza(i, c)

pre:  $c \in \mathcal{C}$

post:  $i \in l$ ,  $i$  este iterator pe colecția  $c$  și referă un prim element din  $c$

valid(i)

pre:  $i \in l$

post:  $valid = \begin{cases} \text{adevarat, dacă elementul curent referit de } i \text{ este valid} \\ \text{fals, altfel} \end{cases}$

urmator(i)

pre:  $i \in l$

post:  $i' \in l$ ,  $i'$  referă următorul element din colecția iterată față de cel referit de  $i$

element(i, e)

pre:  $i \in l$

post:  $e$ : Telement,  $e$  este elementul curent referit de iterator

## Exemplificare reprezentare:

### O primă variantă de reprezentare (R1):

- Ca tablou unidimensional (vector) de elemente care se pot repeta
- În acest caz, iteratorul conține indexul elementului curent

Exemplu:

1	3	2	6	2	5	2
---	---	---	---	---	---	---

### O a doua variantă de reprezentare (R2):

- Ca vector de perechi (element, frecvența), elementele din perechi fiind distincte
- În acest caz, nu mai este suficient ca în reprezentarea iteratorului să avem doar indexul curent. Este necesară și frecvența curentă pentru elementul de la indexul curent.

Exemplu:

(1,1)	(3,1)	(2,3)	(5,1)	(6,1)
-------	-------	-------	-------	-------

## Implementare Python

- Reprezentare R1:

```
1.  class Colectie:
2.      def __init__(self):
3.          self.__elemente = []
4.      def adauga(self, e):
5.          self.__elemente.append(e)
6.      def cauta(self, e):
7.          return e in self.__elemente
8.      def sterge(self, e):
9.          return self.__elemente.remove(e)
10.     def dim(self):
11.         return len(self.__elemente)
12.     def iterator(self):
13.         return Iterator(self)
14.
15. class Iterator:
16.     def __init__(self,c):
17.         self.__c = c
18.         self.__current = 0
19.     def valid(self):
20.         return self.__current < self.__c.dim()
21.     def element(self):
22.         return self.__c._Colectie__elemente[self.__current]
23.     def urmator(self):
24.         self.__current = self.__current+1
25.
26. def populeazaColectieIntregi(c):
27.     c.adauga(1)
28.     c.adauga(2)
29.     c.adauga(3)
30.     c.adauga(2)
31.
32. def tipareste(c):
33.     it = c.iterator()
34.     while it.valid():
35.         print(it.element())
36.         it.urmator()
37.
38. def main():
39.     c = Colectie()
40.     populeazaColectieIntregi(c)
41.     tipareste(c)
```

- Reprezentare R2:

```
1.  class PerecheElementFrecventa:
```



```

49.         else:
50.             self.__perechi.remove(p)
51.
52.     def dim(self):
53.         d = 0
54.         for p in self.__perechi:
55.             d = d + p.getFrecventa()
56.         return d
57.
58.     def iterator(self):
59.         return IteratorColectieFrecventa(self)
60.
61.
62. class IteratorColectieFrecventa:
63.     def __init__(self, c):
64.         self.__col = c
65.         self.__current = 0
66.         self.__f = 1
67.
68.     def valid(self):
69.         return self.__current < len(self.__col._ColectieElementFrecventa__perechi)
70.
71.     def element(self):
72.         return self.__col._ColectieElementFrecventa__perechi[self.__current].getElement()
73.
74.     def urmator(self):
75.         #Distingem doua cazuri: frecventa curenta a elementului curent este (1) strict mai mica decat sau (2) egala cu frecventa elementului curent in colectia iterata
76.         # Daca frecventa curenta a elementului curent este strict mai mica decat
77.         # frecventa elementului curent in colectia iterata, incrementam frecventa curenta
78.         if self.__f < self.__col._ColectieElementFrecventa__perechi[self.__current]
79.             .getFrecventa():
80.                 self.__f = self.__f+1
81.             self.__current = self.__current+1
82.             self.__f=1
83.
84.     def populeazaColectieIntregi(c):
85.         c.adauga(1)
86.         c.adauga(2)
87.         c.adauga(3)
88.         c.adauga(2)
89.         c.sterge(2)
90.         c.adauga(2)
91.

```

```
92. def tipareste(c):
93.     it = c.iterator()
94.     while it.valid():
95.         print(it.element())
96.         it.urmator()
97.
98. def main():
99.     c = ColectieElementFrecventa()
100.    populeazaColectieIntregi(c)
101.    tipareste(c)
```

# SDA – Seminar 2 – Complexități

---

- **Cuprins:**

- Introducere
- Notații asymptotice de complexitate
- Exerciții

## Introducere

Cum definim eficiența unui algoritm? Eficiența unui algoritm este determinată de cantitatea de resurse pe care le consumă, în termeni de *timp* și *memorie*.

Ca modalități de măsurare a complexității timp, avem:

- Analiza **empirică** (sau **experimentală**), care constă în măsurarea timpului efectiv de execuție (aceasta reprezintă un avantaj al metodei), pentru un subset al datelor de intrare posibile, fără a putea prezice performanța pentru toate datele de intrare posibile (aceasta reprezintă un dezavantaj al metodei). Timpul de execuție va fi exprimat numeric, ca număr efectiv de secunde necesare procesării.
- Analiza **asimptotică** (sau **matematică**) este analiza în care surprindem nu timpii exacti de execuție (aceasta ar putea reprezenta un dezavantaj al metodei), ci ordinul de creștere al timpului de execuție, pentru toate datele de intrare posibile (aceasta reprezintă un avantaj al metodei).

Complexitatea unui algoritm poate să depindă și de valorile datelor de intrare, nu doar de dimensiunea lor. Prin urmare, distingem următoarele tipuri de analiză de complexitate:

- **Analiza complexității în cazul defavorabil**, adică pentru date de intrare defavorabile (care implică număr maxim de operații). Această analiză este importantă deoarece oferă o garanție, aceea că algoritmul nu se va purta mai prost, indiferent de valorile datelor de intrare.
- **Analiza complexității în cazul favorabil**, adică pentru date de intrare favorabile (care implică număr minim de operații), potrivită, mai degrabă, pentru probleme în care datele de intrare tind să fie favorabile.
- **Analiza complexității în cazul mediu**, adică pentru date de intrare **aleatorii**. O astfel de analiză este în mod particular utilă, însă este, totodată, mai dificil de realizat. De ce? Deoarece necesită cunoașterea (altfel, presupunerea) unei distribuții statistice a valorilor datelor de intrare pentru a calcula media ponderată cu probabilitățile lor de apariție.

## Notății asimptotice de complexitate

Pentru clase de complexitate avem următoarele notății asimptotice: O, Ω, Θ.

### O

#### Definiții:

- $T(n) \in O(f(n)) \Leftrightarrow \exists$  constantele  $c \in \mathbf{R}_+, c > 0$ , și  $n_0 \in \mathbf{N}$  a.i.  $0 \leq T(n) \leq c \times f(n)$ , pentru orice  $n \geq n_0$
- $T(n) \in O(f(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = 0$  sau constantă nenulă, dar nu  $\infty$

**Semnificație:** pentru valori (suficient de) mari ale dimensiunii intrării,  $c \times f(n)$  este o limită superioară pentru  $T(n)$ , algoritmul purtându-se mereu mai bine decât această limită.

### Ω

#### Definiții:

- $T(n) \in \Omega(f(n)) \Leftrightarrow \exists$  constantele  $c \in \mathbf{R}_+, c > 0$ , și  $n_0 \in \mathbf{N}$  a.i.  $0 \leq c \times f(n) \leq T(n)$ , pentru orice  $n \geq n_0$
- $T(n) \in \Omega(f(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = \text{constantă nenulă}$  sau  $\infty$

**Semnificație:** pentru valori (suficient de) mari ale dimensiunii intrării,  $c \times f(n)$  este o limită inferioară pentru  $T(n)$ , algoritmul purtându-se mereu mai prost decât această limită.

### Θ

#### Definiții:

- $T(n) \in \Theta(f(n)) \Leftrightarrow \exists$  constantele  $c_1, c_2 \in \mathbf{R}_+, c_1 > 0, c_2 > 0$ , și  $n_0 \in \mathbf{N}$  a.i.  $0 \leq c_1 \times f(n) \leq T(n) \leq c_2 \times f(n)$ , pentru orice  $n \geq n_0$
- $T(n) \in \Theta(f(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{T(n)}{f(n)} = \text{constantă nenulă, finită}$

**Semnificație:** pentru valori (suficient de) mari ale dimensiunii intrării,  $c_1 \times f(n)$  este o limită inferioară pentru  $T(n)$ , iar  $c_2 \times f(n)$  este o limită superioară

#### Observație:

Să ne amintim complexitățile algoritmilor cunoscuți de căutare și sortare...

Algoritm	Complexitate timp				Complexitate spațiu
	CF	CD	CM	Total	
Căutare secvențială	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Căutare binară	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$O(\log_2 n)$	$\Theta(1)$
Sortare prin selecție	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1) - \text{in place}^*$
Sortare prin inserție	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1) - \text{in place}$
Sortare prin metoda bulelor	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1) - \text{in place}$
Sortare rapidă ( <i>QuickSort</i> )	$\Theta(n \log_2 n)$	$\Theta(n^2)$	$\Theta(n \log_2 n)$	$O(n^2)$	$\Theta(1) - \text{in place}$
Sortare prin interclasare ( <i>Merge Sort</i> )	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n) - \text{out of place}$

\*Algoritmii de sortare *in place* sortează sirul fără a folosi structuri de date suplimentare, ci doar spațiu de memorie adițional constant, pentru variabilele auxiliare. De exemplu, un algoritm de sortare care efectuează ordonarea doar prin interschimbări de elemente este *in place*. Un algoritm care nu este *in place* este *out of place*.

### Exerciții

1. Adevărat sau fals?
  - a.  $n^2 \in O(n^3)$
  - b.  $n^3 \in O(n^2)$
  - c.  $2^{n+1} \in \Theta(2^n)$
  - d.  $n^2 \in \Theta(n^3)$
  - e.  $2^n \in O(n!)$
  - f.  $\log_{10} n \in \Theta(\log_2 n)$
  - g.  $O(n) + \Theta(n^2) = \Theta(n^2)$
  - h.  $\Theta(n) + O(n^2) = O(n^2)$
  - i.  $O(n) + O(n^2) = O(n^2)$
  - j.  $O(f) + O(g) = O(\max\{f, g\})$
  - k.  $O(n) + \Theta(n) = O(n)$

### Soluții:

- a.  $n^2 \in O(n^3)$  - Adevărat
- b.  $n^3 \in O(n^2)$  – Fals
- c.  $2^{n+1} \in \Theta(2^n)$  – Adevărat
- d.  $n^2 \in \Theta(n^3)$  – Fals
- e.  $2^n \in O(n!)$  - Adevărat
- f.  $\log_{10} n \in \Theta(\log_2 n)$  - Adevărat

- g.  $O(n) + \Theta(n^2) = \Theta(n^2)$  - Adevărat
- h.  $\Theta(n) + O(n^2) = O(n^2)$  - Adevărat
- i.  $O(n) + O(n^2) = O(n^2)$  - Adevărat
- j.  $O(f) + O(g) = O(\max\{f,g\})$  - Adevărat
- k.  $O(n) + \Theta(n) = O(n)$  - Adevărat, dar preferăm  $O(n) + \Theta(n) = \Theta(n)$ , întrucât este mai exact

2. Construiți un algoritm cu timpul  $\Theta(n \log_2 n)$

**Exemplu de soluție :**

Pentru  $i = 1, n$  execută

```

j ← n
Câttimp j ≠ 0 execută
    j = ⌈ j / 2 ⌉
sf_câttimp
sf_pentru
```

3. Calculați complexitatea pentru următorii 2 algoritmi:

a)

```

gasit ← fals
Pentru i ← 1, n executa
    Dacă  $x_i = a$  atunci
        gasit ← adevarat
    sf_daca
sf_pentru
```

**Soluție:**

A se observa că ciclul se execută independent de valoarea variabilei *găsit*.

$$\begin{aligned} CF: \theta(n) \\ CD: \theta(n) \end{aligned} \Rightarrow \theta(n)$$

b)

```

gasit ← fals
i ← 1
câttimp gasit = fals și i < n execută
    Dacă  $x_i = a$  atunci
        gasit ← adevarat
    sf_daca
    i ← i + 1
sf_câttimp
```

CF:  $\Theta(1)$

CD:  $\Theta(n)$

CM: sunt  $n + 1$  cazuri posibile (elementul de găsește pe oricare dintre cele  $n$  poziții sau pe niciuna)

$$T(n) = \sum_{I \in D} P(I) * E(I) = \frac{1}{n+1} + \frac{2}{n+1} + \dots + \frac{n+1}{n+1} = \frac{(n+1)*(n+2)}{2*(n+1)} = \frac{n+2}{2} \in \Theta(n)$$

Complexitatea algoritmului este, prin urmare,  $O(n)$ .

4. Fie  $X$  este un sir de  $n$  numere naturale, fiecare element fiind  $\leq n$  și următorul algoritm.

```
k ← 0
Pentru i = 1, n execută
    Pentru j = 1,  $x_i$  execută
        * k ← k +  $x_j$ 
    sf_pentru
sf_pentru
```

Observând că operația \* se efectuează de un număr de ori egal cu suma elementelor sirului, este corect să determinăm și să exprimăm complexitatea timp precum mai jos ?

$$T(n) = \sum_{i=1}^n \sum_{j=1}^{x_i} 1 = \sum_{i=1}^n x_i = s \text{ (suma elementelor)} \in \Theta(s)$$

**Soluție:**

Exemplificăm un sir care respectă proprietatea enunțată, fiind definit astfel:

Fie  $x_i = \begin{cases} 1, & \text{dacă } i \text{ este patrat perfect} \\ 0, & \text{altfel} \end{cases}$

Deducem că suma elementelor sirului este  $s = \lceil \sqrt{n} \rceil$ , deci, conform calculului de mai sus, complexitatea este  $\Theta(\sqrt{n})$ , deci este sub-liniară. Dar ciclul exterior efectuează  $n$  pași. Survine, astfel, o contradicție.

Prin urmare, exprimarea corectă a complexității este:

$$T(n) \in \Theta(\max\{n, s\})$$

**Observații:**

- Dacă elementele sirului ar fi fost numere naturale nenele, exprimarea inițială a complexității ar fi fost corectă
- Altfel, dacă ne gândim la cazul favorabil, acesta apare atunci când, de pildă, toate elementele sunt nule, în acest caz efectuându-se  $n$  operații, deci complexitatea fiind  $T(n) \in \Theta(\max\{n, 0\}) = \Theta(n)$ , iar cazul defavorabil corespunde cazului în care toate elementele sirului au valoarea  $n$ , în acest caz complexitatea fiind  $T(n) \in \Theta(\max\{n, n^2\}) = \Theta(n^2)$ .

5. Cum putem determina dacă un sir arbitrar de numere  $x_1 \dots x_n$  conține cel puțin două elemente egale în  $\Theta(n \log_2 n)$ ?

**Soluție:**

Se va aplica MergeSort pe sir, verificarea rezumându-se ulterior la a parcurge sirului și a verifica existența a două elemente egale, pe poziții consecutive (aceasta efectuându-se în  $O(n)$ ).

6. Calculați complexitatea:

```

Pentru i = 1, n execută
    @op elementară
    sf_pentru
    i ← 1
    k ← adevarat
    Câttimp i <= n - 1 și k execută
        j ← i
        k1 ← adevărat
        Câttimp j <= n și k1 execută
            @ op elementară (k1 poate fi modificat)
            j ← j + 1
        sf_câttimp
        i ← i + 1
        @op elementara (k poate fi modificat)
    sf_câttimp

```

**Soluție:**

CF: k, k<sub>1</sub> poate deveni *false* după primul pas  $\Rightarrow \Theta(n)$

CD: k, k<sub>1</sub> nu devin *false* niciodată

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n-1} \sum_{j=i}^n 1 = \sum_{i=1}^{n-1} n - i + 1 = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1} 1 = \\
&= n * (n - 1) - \frac{n * (n - 1)}{2} + n - 1 \in \Theta(n^2)
\end{aligned}$$

CM: pentru un i fixat, k<sub>1</sub> poate deveni *false* după 1, 2, ..., n-i+1 execuții.

Prob:  $\frac{1}{n-i+1}$

$$\frac{1}{n-i+1} + \frac{2}{n-i+1} + \dots + \frac{n-i+1}{n-i+1} = \frac{(n-i+1) * (n-i+2)}{2(n-i+1)} = \frac{(n-i+2)}{2}$$

Pentru ciclul *Câtimp* exterior, k poate deveni *fals* după 1, 2, ..., n-1 iterații.

$$\begin{aligned} T(n) &= \frac{1}{n-1} * \frac{n-1+2}{2} + \frac{2}{n-1} * \frac{n-2+2}{2} + \dots + \frac{n-1}{n-1} * \frac{n-(n-1)+2}{2} = \\ &\quad \frac{1}{2 * (n-1)} * \sum_{i=1}^{n-1} i * (n-i+2) = \dots \\ &= \frac{1}{2 * (n-1)} * \left( \frac{n * (n-1) * n}{2} - \frac{(n-1) * n * (2n-1)}{6} + 2 * \frac{(n-1) * n}{2} \right) \\ &= \frac{1}{2} * \left( \frac{n^2}{2} - \frac{2 * n^2 - n}{6} + n \right) = \frac{1}{2} * \left( \frac{3n^2 - 2n^2 + 5n}{6} \right) \in \theta(n^2) \end{aligned}$$

Complexitatea totală:  $O(n^2)$

7. Calculați complexitatea:

```

Subalg p(x,s,d) este:
Daca s < d atunci
    m ← [(s+d)/2]
    Pentru i = s, d-1, execută
        @op. Elementare
    sf_pentru
    Pentru i = 1,2 execută
        p(x, s, m)
    sf_pentru
sf_daca
sf_subalg

apel: p(x, 1, n)

```

**Soluție:**

$$T(n) = \begin{cases} 2 * T\left(\frac{n}{2}\right) + n, & \text{daca } n > 1 \\ 0, & \text{altfel} \end{cases}$$

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n & n &= 2^k \\ T(2^k) &= 2 * T(2^{k-1}) + 2^k \\ 2 * T(2^{k-1}) &= 2^2 * T(2^{k-2}) + 2^k \\ 2^2 * T(2^{k-2}) &= 2^3 * T(2^{k-3}) + 2^k \\ &\quad \ddots \\ 2^{k-1} * T(2) &= 2^k * T(1) + 2^k \end{aligned}$$

$$T(2^k) = 2^k * T(1) + k * 2^k = k * 2^k = n * \log_2 n \rightarrow T(n) \in \theta(n \log_2 n)$$

8. Calculați complexitatea:

```

s ← 0
Pentru i = 1, n2 execută
    j ← i
    Câtimp j ≠ 0 execută
        s ← s + j
        j ← j - 1
    sf_câtimp
sf_pentru

```

**Soluție:**

$$T(n) = \sum_{i=1}^{n^2} \sum_{j=1}^i 1 = \sum_{i=1}^{n^2} i = \frac{n^2 * (n^2 + 1)}{2} \in \theta(n^4)$$

9. Calculați complexitatea:

```

s ← 0
pentru i = 1, n2 execută
    j ← n
    Câtimp j ≠ 0 execută
        s ← s + j - 10 * [j/10]
        j ← [j/10]
    sf_câtimp
sf_pentru

```

**Soluție:**

- Ciclul *Câtimp* se execută de  $\log_{10} n$  ori.
- $T(n) \in \Theta(n^2 \log_{10} n) \Rightarrow T(n) \in \Theta(n^2 \log_2 n)$

10. Calculați complexitatea:

```

Subalg operație(n, i) este
Daca n > 1 atunci
    i ← 2 * i
    m ← [n/2]
    operație (m, i-2)
    operație (m, i-1)
    operație (m, i+2)
    operație (m, i+1)
altfel
    scrie i
sf_daca

```

### sf\_subalg

$$T(n) = \begin{cases} 4 * T\left(\frac{n}{2}\right) + 2, & n > 1 \\ 1, & \text{altfel} \end{cases}$$

Soluție:

$$\begin{aligned} T(n) &= 4 * T\left(\frac{n}{2}\right) + 2 \\ T(2^k) &= 4 * T(2^{k-1}) + 2 \\ 4 * T(2^{k-1}) &= 4^2 * T(2^{k-2}) + 4 * 2 \\ 4^2 * T(2^{k-2}) &= 4^3 * T(2^{k-3}) + 4^2 * 2 \\ &\quad \cdots \\ 4^{k-1} * T(2) &= 4^k * T(1) + 4^{k-1} * 2 \\ T(n) &= 4^k + 2 * (4 + 4^2 + \dots + 4^{k-1}) = 4^k + 2 * \frac{4^k - 2}{3} = n^2 + 2 * \frac{n^2 - 2}{3} \in \theta(n^2) \end{aligned}$$

# Seminar 3 - Multidictionar ordonat (MDO)

---

- **Cuprins:**

- Definire MDO
- Reprezentare MDO
- Reprezentare și implementare Iterator MDO
- Implementare MDO

## Definire MDO

Ce este un Dicționar?

- Un Dicționar este un container care conține perechi <cheie, valoare>, cheile fiind distințe și fiecare cheie având o singură valoare asociată.

Ce este un **Multidicționar**? Prin ce diferă el de un Dicționar?

- Un **Multidicționar** este un container care conține asocieri <cheie, valoare>, dar în care o cheie poate avea mai multe valori asociate, deci nu una singură.

Ce este un Multidicționar **Ordonat**? Prin ce diferă el de un **Multidicționar** oarecare / neordonat?

- Într-un Multidicționar **Ordonat** cheile sunt memorate într-o anumită ordine, dată de o relație de ordine.

**Problemă:** Să se implementeze a) TAD MDO (Multidictionar Ordonat) reprezentat pe listă simplu înălțuită (LSI) cu alocare dinamică și b) iteratorul aferent.

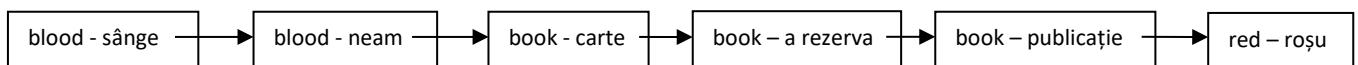
## Reprezentare MDO

Să considerăm ca exemplu un multidicționar conținând traducerile unor cuvinte din engleză în română:

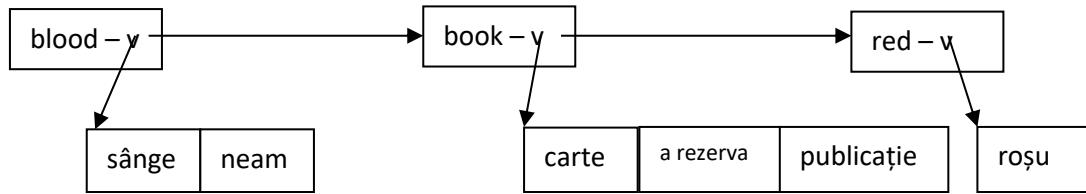
- book – carte, a rezerva, publicație
- red – roșu
- blood – sânge, neam

Cum am putea reprezenta în memorie, folosind LSI ca structură de date, MDO-ul exemplificat anterior?

**Reprezentare 1:** Listă înălțuită conținând (în noduri, ca informație utilă) **perechi <cheie, valoare>**, putând fi mai multe noduri cu o aceeași cheie. Acestea vor fi consecutive.



**Reprezentare 2:** Listă înlănțuită conținând **perechi <cheie, listă de valori>**. Cheile sunt unice și ordonate.



Pentru implementare, vom considera reprezentarea 2. Aceasta presupune că MDO-ul este reprezentat printr-o LSI compusă din noduri în care avem ca informație utilă perechi <cheie, (referință la) listă de valori asociate>.

Ce vom avea, aşadar în reprezentarea MDO-ului?

#### TElement:

c: TCheie  
l: TListă

#### NodT:

e: TElement  
urm: ↑NodT

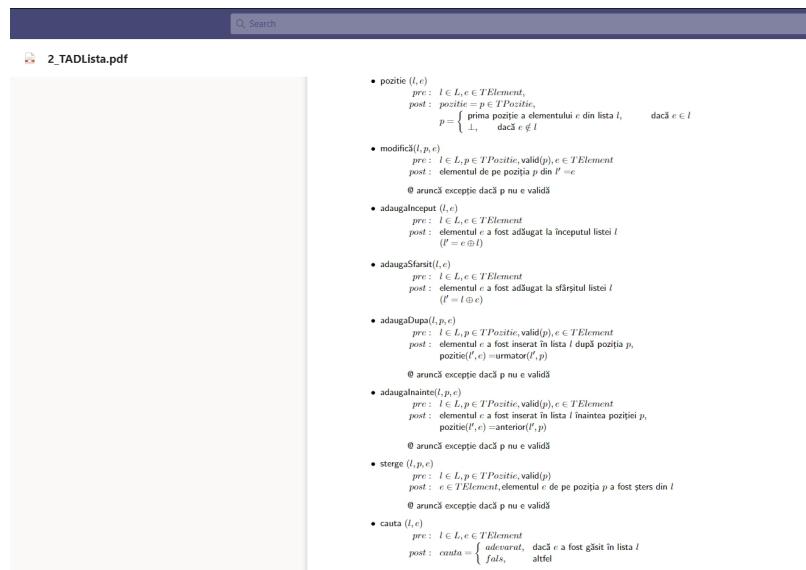
#### MDO:

prim: ↑NodT  
R: Relație

$$R(c_1, c_2) = \begin{cases} \text{adevărat, dacă } c_1 \leq c_2 \text{ (} c_1 \text{ vine înaintea lui } c_2 \text{)} \\ \text{fals, altfel} \end{cases}$$

#### **Explicații:**

- **TElement** este tipul informației utile din nodurile listei simplu înlănțuite (LSI). Aceasta se compune din cheie (unică în LSI, de tipul TCheie) și o listă (TListă) de valori. **TListă** este un TAD conținând elemente de tipul valorilor din MDO, adică TValoare. Pentru a crea, accesa și manipula liste de valori, ne vom folosi de operațiile specifice TAD Listă (pe acestea le găsiți în materialul 2\_TADLista aferent cursului nr. 5).



- Nodurile din LSI sunt de tipul **NodT**. Un nod conține informația utilă, de tipul TElement, detaliat anterior, și un pointer la nodul următor ( $\uparrow$  este notația Pseudocod pentru "pointer la").

- În reprezentarea **MDO**, vom reține pointer la primul nod al LSI (acesta ne va permite să accesăm întreaga LSI, datorită câmpului *urm* din noduri) și relația de ordine în raport cu care sunt ordonate cheile din MDO.

### **Reprezentare și implementare Iterator MDO**

#### **Reprezentare:**

##### **IteratorMDO:**

mdo: MDO  
current:  $\uparrow$ NodT  
itL: IteratorListă

#### **Explicații:**

Pentru reprezentarea Iteratorului pe MDO, avem nevoie de:

- o **referință la MDO-ul iterat** (în general, iteratorul reține o referință la containerul iterat)
- un **pointer la nodul current** din LSI cu ajutorul căreia implementăm MDO-ul (acesta ne va permite să accesăm cheia curentă, precum și lista valorilor asociate ei)
- un **iterator pe lista de valori** asociată cheii curente, din nodul current (acesta ne va permite să accesăm valoarea curentă asociată cheii curente). Fiind un TAD, lista valorilor ne va oferi un iterator ca mecanism de parcurgere, noi nefiind preoccupați de reprezentarea acesteia.

#### **Operațiile din interfață:**

##### **- creeaza**

- Constructor: creeaza un iterator pe un MDO care referă prima pereche <cheie, valoare> din MDO. Cheia din această pereche va fi cea mai mică cheie în raport cu relația de ordine  $R$ , adică aceea care este în relație cu toate celelalte.

##### **- element**

- Returnează perechea curentă, de tipul <cheie, **valoare**>, din MDO (Așadar, iteratorul va returna perechi individuale <cheie, valoare> și nu perechi <cheie, listă de valori>; Observăm că, dacă există mai multe valori asociate unei chei  $c$ , atunci perechile având cheia  $c$  vor fi returnate consecutiv de iterator).

##### **- valid**

- Funcție booleană care verifică validitatea iteratorului, adică dacă mai sunt perechi <cheie, valoare> de iterat

##### **- următor**

- Operație prin care îi solicităm iteratorului să refere următoarea pereche <cheie, valoare> din MDO, pentru a putea continua parcurgerea.

Afișarea elementelor dintr-un MDO folosind iteratorul:

```
Subalg tipărire(mdo)// subalgoritmul primește ca argument MDO-ul al cărui conținut dorim să-l iterăm
    iterator(mdo, i) // îi cerem MDO-ului un iterator pe conținutul lui; nil returnează în i; i
        referă prima pereche <cheie, valoare> din MDO-ul mdo
    câtimp valid(i) execută: //cat timp iteratorul este valid, deci mai sunt perechi <cheie,
        valoare> de parcurs,
        element(i, <c,v>) //îi cerem iteratorului perechea <cheie, valoare> curentă; ne-o
        returnează în <c,v>
        @tipărește c și v //o tipărim
        următor(i) //și îi cerem iteratorului să "mearga mai departe" / sa refere următoarea
        pereche <cheie, valoare> din MDO
    sf_câtimp
sf_subalg
```

### Specificarea domeniului

I = {i, i este un iterator pe un MDO cu chei de tipul TCheie și valori de tipul TValoare}

### Implementarea operațiilor iteratorului

```
//pre: mdo:MDO
//post: it:I, it referă prima pereche <cheie, valoare> din MDO-ul mdo (daca aceasta
exista; altfel, daca mdo este vid, valid(it)=fals)
Subalg creează (it, mdo):
    it.mdo ← mdo //initializam referinta la MDO-ul iterator
    it.current ← mdo.prim //cursorul iteratorului (de tip pointer) va referi primul nod al LSI
    dacă it.current ≠ NIL atunci: //daca acest nod nu este NIL, adica exista cel putin un nod
    in LSI => exista cel putin o pereche <cheie, valoare> in MDO, deci mdo nu e vid
        iterator([it.mdo.prim].e.l, it.itL) //creăm un interator de lista valorilor din
    primul nod al LSI, adica pe lista valorilor asociate primei chei
    sf_dacă
sf_subalg
Complexitate: Θ(1) (se efectueaza un numar constant de operatii elementare)
```

*Observație: [pointer\_la\_nod] este notația Pseudocod folosită pentru a accesa nodul.*  
În exemplul de mai sus, *it.mdo.prim* este de tip  $\uparrow$ NodT (conform reprezentării), iar *[it.mdo.prim]* este de tip NodT.

---

```
//pre: it:I, valid(it)
//post: e = <c,v>, e fiind perechea <cheie, valoare> curentă din parcurgere, c:
TCheie, v:TValoare
Subalg element(it, e):
    c ← [it.current].e.c //cheia curentă este cea conținută în informația utilă a nodului current
    element(it.itL, v) //, iar valoarea curentă este cea pe care iteratorul pe lista de valori o
    referă
    e ← <c,v> //perechea curentă returnată fiind compusă din cheia curentă și valoarea curentă
sf_subalg
```

Complexitate:  $\Theta(1)$  (se efectueaza un numar constant de operatii elementare)

---

```

//pre: it:I
//post: valid(it) = adevarat, daca it refera o pereche <cheie, valoare> din MDO-ul
iterat / mai sunt perechi <cheie, valoare> de parcurs; altfel, valid(it) = fals
Funcția valid(it):
    Dacă it.current ≠ NIL atunci //verificăm dacă cursorul iteratorului referă un nod valid din
LSI, adică dacă nu am parcurs deja, în întregime, LSI.
        valid ← adevărat
    altfel
        valid ← fals
sf_aceasta

```

Complexitate:  $\Theta(1)$  (se efectuează un număr constant de operații elementare)

---

```

//pre: it:I, valid(it)
//post: it':I, it' referă urmatoarea pereche <cheie, valoare> din MDO-ul iterat față
de cea referită de it
Subalgoritm următor(it):
    următor(it.itL) //mutam iteratorul pe lista de valori asociate cheii curente a.i. sa refere
urmatoarea valoare
    dacă l valid(it.itL) atunci //, iar daca acesta devine nevalid, adica nu mai sunt valori
neiterante asociate cheii curente
        it.current ← [it.current].urm //atunci ne deplasam la urmatoarea cheie, prin a face
cursorul iteratorului sa refere următorul nod din LSI, dat fiind ca urmatoarea cheie se află în următorul
nod
        dacă it.current ≠ NIL atunci // daca următorul nod intr-adevar există, adică nu am
epuizat deja toate cheile (ceea ce ar însemna că am finalizat iterarea), creăm un iterator pe lista
valorilor asociate noii chei curente
            iterator ([it.current].e.l, it.itL)
            sf_dacă
        sf_dacă
sf_subalgoritm

```

Complexitate:  $\Theta(1)$  (se efectuează un număr constant de operații elementare)

---

## Implementare MDO

### Definirea domeniului TAD-ului MDO:

MDO = {mdo, mdo este un MDO cu chei de tipul TCheie și valori de tipul TValoare, cheile fiind ordonate în raport cu o relație de ordine R:TCheie x TCheie → {adevarat, fals}}

Pentru a exprima complexitățile timp ale operațiilor vom folosi următoarele notații:

n – nr de chei distincte

mdo – nr total de elemente (nr total de perechi <cheie, valoare>)

```

//pre: R:Relatie: R:Tcheie x Tcheie -> {adevarat, fals}
//post: mdo: MDO, mdo este un MDO vid, ale cărui chei vor fi ordonate în raport cu
//relația de ordine R
subalg creează(mdo, R):
    mdo.R ← R //initializăm relația în raport cu care se vor ordona cheile
    mdo.prim ← NIL //cursorul iteratorului de tip ↑NodT (adică pointer la nod) va referi NIL,
    întrucât nu există încă noduri în LSI, dat fiind faptul că MDO-ul este vid
sf_subalg

```

Complexitate:  $\Theta(1)$  (se efectueaza un numar constant de operatii elementare)

---

```

//pre: mdo: MDO
//post: mdo a fost distrus / memoria ocupată de mdo a fost eliberată
subalg distrugă(mdo):
    cât timp mdo.prim ≠ NIL execută //cât timp mai există noduri nedeallocate în LSI
        p ← mdo.prim //reținem referința (pointerul) la primul nod al LSI
        mdo.prim ← [mdo.prim].urm //ne deplasăm la următorul nod din LSI
        distrugă([p].e.l) //distrugem lista de valori asociate cheii din primul nod al LSI,
    folosind destrutorul TAD-ului TListă
        dealocă(p) //deallocăm primul nod al LSI, pointerul la acesta reținându-l în p, în
    prealabil
        sf_câtimp
sf_subalg

```

Complexitate:

Ne amintim că pentru a exprima complexitățile timp ale operațiilor, am convenit folosirea următoarelor notații:

$n$  – nr de chei distincte

$mdo$  – nr total de elemente (nr total de perechi <cheie, valoare>)

$\Theta(mdo)$  - dacă distrugerea unei liste de valori l se face în  $\Theta(\text{lungimea listei})$  (de exemplu, dacă acestea sunt reprezentate folosind LSI alocate dinamic). În acest caz,  $mdo$  = numarul de perechi din MDO = suma lungimilor listelor de valori.

sau

$\Theta(n)$  - dacă listele de valori pot fi dealocate în  $\Theta(1)$  (de exemplu, dacă acestea sunt reprezentate pe vector)

---

- Propun să implementăm o operație ajutătoare, neexpusă în interfață, pe care o vom folosi pentru a implementa operațiile *caută*, *adaugă* și *șterge* din interfața MDO-ului. Aceasta va avea următoarele specificații.

```

//pre: mdo: MDO, c:Tcheie
//post: prec, nod: ↑TNod, nod este pointer la nodul din mdo continand cheia c, iar
prec este pointer la nodul precedent acestuia. Daca cheia nu exista, nod=NIL, iar
prec va fi nodul dupa care ar trebui inserat un nod continand cheia c a.i. mdo sa
ramana ordonat dupa inserare

```

//**OBSERVATIE:** Aceasta metoda este una privată, adică nu este expusă în interfața TAD-ului MDO.

**Subalgoritm cautaNod(mdo, c, nod, prec) este:**

```

aux ← mdo.prim //aux va fi pointer la nodul curent, îl inițializăm a.î. să refere primul nod
din LSI
precedent ← NIL //precedent va fi pointer la nodul precedent nodului referit de aux, îl
inițializăm cu NIL, întrucât aux referă primul nod, acesta neavând nod precedent
gasit ← fals //deocamdată nu am găsit nodul conținând cheia c
Cat timp aux ≠ NIL și mdo.R([aux].e.c, c) și not gasit execută
    //parcugem LSI cât timp mai sunt noduri, nu am găsit încă cheia, iar cheia curentă este
    în relație cu cheia căutată (în momentul în care relația între acestea nu mai are loc, putem opri căutarea,
    întrucât nu mai sunt șanse să găsim cheia căutată, dat fiind faptul că MD-ul este ordonat)
    Daca [aux].e.c = c atunci //verificam daca cheia curentă este cheia cautată
        gasit ← adevarat //în caz afirmativ, marcăm găsirea ei, prin actualizarea
    valorii variabilei găsit
    Altfel //în felică, continuăm parcugerea LSI
        precedent ← aux //nodul precedent devenind nodul curent
        aux ← [aux].urm //, iar nodul curent, urmatorul nod din LSI
SfarsitDaca
SfarsitCatTimp
Daca gasit atunci //daca am găsit cheia, inițializăm corespunzător datele de ieșire
    nod ← aux //nodul continând cheia fiind aux
    prec ← precedent //, iar cel precedent lui precedent
Altfel //în felică
    nod ← NIL //nu există nod continând cheia cautată
    prec ← precedent //, iar un nod continând cheia cautată ar trebui adăugat după
precedent
SfarsitDaca
SfarsitSubalgoritm

```

Complexitate:  $O(n)$  (în cel mai defavorabil caz, se parcurge întreaga LSI, aceasta având  $n$  noduri)

Căutarea în MDO se efectuează după cheie, iar rezultatul este de tip **TListă**, returnându-se lista de valori asociate cheii date.

```

//Date intrare: mdo:MDO, c:TCheie
//Date ieșire: lista: TLista
//pre: mdo: MDO, c:TCheie
//post: lista:TLista, lista fiind lista de valori asociate cheii c în
multidictionarul ordonat mdo (daca cheia c nu există în mdo, se va returna o lista
vida/goala)
Subalgoritm cauta(mdo, c, lista) este:
    cautaNod (mdo, c, nod, prec)
    Daca nod = NIL atunci //cheia c nu există în mdo
        creeaza(lista) //se initializează lista (ca lista vida/goala), folosind constructorul
    altfel
        lista ← [nod].e.l //în felică, lista valorilor asociate cheii c a găsit în nod, ca parte din
informația utilă
    SfarsitDaca
SfarsitSubalgoritm

```

**Complexitate:**  $O(n)$  (complexitatea operatiei `cautaNod` fiind  $O(n)$ , iar restul operatiilor necesitand timp constant)

---

```
//pre: mdo: MDO, c:TCheie, v:TValoare
//post: mdo':MDO (starea multidictionarului se schimba), perechea <c,v> este adaugata
in multidictionar, mdo' = mdo (+) <c,v>
```

```
//valoarea v este adaugata in lista valorilor asociate cheii c; Daca cheia c nu a
fost adaugata in prealabil in mdo, o vom adauga, asociindu-i lista de valori formata
din unicul element v)
```

**Subalgoritm adauga(mdo, c, v)** este:

```
    cautaNod (mdo, c, nod, prec)
    Daca nod = NIL atunci //cheia c nu exista in mdo
        adaugaCheieNoua(mdo, c, v, prec) //o adaugam, deci, ca si cheie noua
    Altfel //cheia c exista deja in mdo
        adaugaFinal([nod].e.l, v) //adaugam, prin urmare, valoarea in lista de
                                    valori asociata ei
```

**SfarsitDaca**

**SfarsitSubalgoritm**

**Complexitate:**

`cautaNod` -  $O(n)$

`adaugaCheieNoua` -  $\Theta(1)$  (datorita utilizarii `prec`, a nodului dupa care efectuam adaugarea)

In locul operatiei `adaugaFinal` se poate folosi o alta operatie de adaugare (de pilda, la inceput) a.i. sa obtinem complexitate constanta ( $\Theta(1)$ ). Aceasta este posibil deoarece relatia de ordine se impune strict cheilor, deci nu si valorilor asociate unei chei.

Daca `adaugaFinal`(sau o operatie de adaugare alternativa) are complexitatea  $\Theta(1) \Rightarrow O(n)$  ca si complexitate pentru `adauga`

Daca `adaugaFinal`(sau o operatie de adaugare alternativa) are complexitatea  $\Theta(\text{lungimea listei}) \Rightarrow O(mdo)$  ca si complexitate pentru `adauga`, `mdo` fiind numarul total de elemente (perechi `<cheie, valoare>`)

---

```
//Operatie auxiliara, nefacand parte din interfata MDO
//pre: mdo: MDO, c: TCheie, v:TValoare, prec este un ↑TNod (fiind nodul dupa care
noul nod trebuie inserat)
//post: mdo':MDO (starea multidictionarului se schimba), un nou nod avand cheia c si
valoarea asociata v (lista de valori asociata formata din unicul element v) este
adaugat multidictionaruluiordonat. Ordinea cheilor va respecta relatia de ordine
impusa.
```

**Subalgoritm adaugaCheieNoua(mdo, c, v, prec)** este:

```
    aloca(nou) //alocam un nou nod
    [nou].e.c ← c //il completam cu informatia utila, adica cu cheia data
    creeaza([nou].e.l) //si o lista care va contine doar valoarea v
    adaugaFinal([nou].e.l, v)
    Daca prec = NIL atunci //noul nod trebuie adaugat ca prim nod
        [nou].urm ← mdo.prim
```

```

    mdo.prim← nou
Altfel //noul nod se va insera dupa nodul prec
    [nou].urm ← [prec].urm
    [prec].urm ← nou
SfarsitDaca
SfarsitSubalgorithm

```

**Complexitate:**  $\Theta(1)$  // adaugaFinal are complexitatea  $\Theta(1)$  intrucat adaugarea se va face intr-o lista vida

---

```

//pre: mdo: MDO, c:TCheie, v:TValoare
//post: mdo':MDO (starea multidictionarului se schimba), perechea <c,v> este
eliminata din multidictionar, mdo' = mdo (-) <c,v>

Subalgoritm sterge(mdo, c, v) este:
    cautaNod (mdo, c, nod, prec)
    Daca nod ≠ NIL atunci // daca exista in mdo cheia c
        pozitie ← pozitie([nod].e.l, v) //determin pozitia valorii v in lista
        valorilor asociate cheii c
        Daca valid([nod].e.l, pozitie) atunci //daca pozitia e valida, deci v exista
            sterge([nod].e.l, pozitie, e) //,sterg v din lista valorilor
        SfarsitDaca
        Daca esteVida([nod].e.l) atunci //daca am sters singura valoare, deci lista asociata cheii
            c este, ulterior stergerii valorii v, vida
            stergeCheie(mdo, prec) //, sterg cheia, prin intermediul unei opreactii pe
            care o descriem in cele ce urmeaza
        SfarsitDaca
    SfarsitDaca
SfarsitSubalgorithm

```

**Complexitate:**  $O(m)$

---

```

//Operatie auxiliara, nefacand parte din interfata MDO

//pre: mdo:MDO, c: TCheie, prec: ↑TNod, mdo contine un nod avand cheia c dupa nodul
prec (daca prec este NIL atunci primul nod al mdo contine cheia c). Lista de valori
din nodul avand cheia c este vida.

//post: nodul continand cheia c este eliminat din mdo

```

```

Subalgoritm stergeCheie(mdo, prec) este:
    Daca prec = NIL atunci //daca nodul de sters este primul (cel referit de mdo.prim), stergem primul
    nod din LSI
        sters ← mdo.prim
        mdo.prim ← [mdo.prim].urm
        distrug([sters].e.l)
        dealoca(sters)
    sltfel
    //altfel, stergem nodul urmator nodului referit de prec
        sters ← [prec].urm

```

```
[prec].urm← [[prec].urm].urm  
distruge([sters].e.l)  
dealloca(sters)
```

**SfarsitDaca**

**SfarsitSubalgorithm**

**Complexitate:**  $\Theta(1)$  (*distruge* va distrugere o lista vida)

# SDA – Seminar 4

---

- **Cuprins:**

- Algoritmi de sortare
  - ❖ BucketSort
  - ❖ Sortarea Lexicografică
  - ❖ Radix Sort
- Interclasarea a două liste simplu înlăntuite, alocate dinamic

## Algoritmi de sortare

Să ne amintim, din seminarul 2, complexitățile algoritmilor de sortare cunoscuți.

Algoritm	Complexitate timp				Complexitate spațiu
	CF	CD	CM	Total	
Căutare secvențială	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Căutare binară	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$O(\log_2 n)$	$\Theta(1)$
Sortare prin selecție	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1) - \text{in place}^*$
Sortare prin inserție	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1) - \text{in place}$
Sortare prin metoda bulelor	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1) - \text{in place}$
Sortare rapidă <i>(QuickSort)</i>	$\Theta(n \log_2 n)$	$\Theta(n^2)$	$\Theta(n \log_2 n)$	$O(n^2)$	$\Theta(1) - \text{in place}$
Sortare prin interclasare <i>(Merge Sort)</i>	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n) - \text{out of place}$

Toți algoritmii de sortare din tabelul anterior sunt **algoritmi de sortare prin comparare**. De ce? Deoarece sortarea unui sir se efectuează pe bază de comparații între elementele sirului. Se poate arăta că orice algoritm de sortare prin comparare necesită  $\Omega(n \log_2 n)$  comparații, în cel mai defavorabil caz.

În seminarul curent, se vor prezenta algoritmi de sortare având **complexitate liniară**, dar care sunt aplicabili doar dacă datele de intrare verifică anumite constrângeri. Așadar, aceștia **nu** vor fi algoritmi de sortare prin comparare.

## A. BucketSort

### Enunțul problemei

- Se dă un sir S de  $n$  perechi (cheie, valoare),  $cheie_i \in \{0, 1, \dots, N-1\}$ ,  $i=1, n$
- Se cere să se sorteze S după chei

### Exemplu

Se dă un sir S: (7, d) (1, c) (3, b) (7, g) (3, a) (7, e)

n = 6

N = 9 (sau 8, sau 10 etc.)

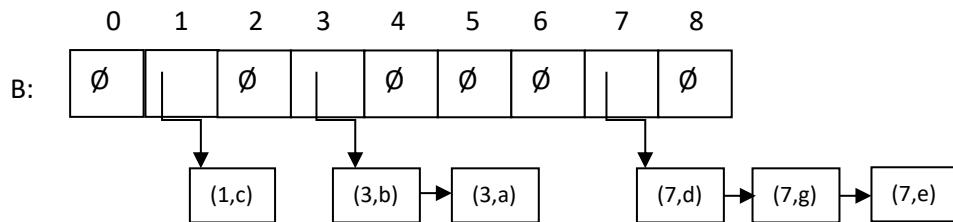
=> (1, c) (3, b) (3, a) (7, d) (7, g) (7, e)

Observăm că se cere ca sortarea să se efectueze strict după chei. Prin urmare, perechile având aceeași cheie fi consecutive în sirul sortat, însă ordinea lor este irelevantă. Dacă există mai multe perechi cu o aceeași cheie, vor exista mai multe soluții. Alegerea uneia dintre soluții se poate face după criteriul păstrării ordinii initiale a perechilor având aceeași cheie.

### Ideea algoritmului:

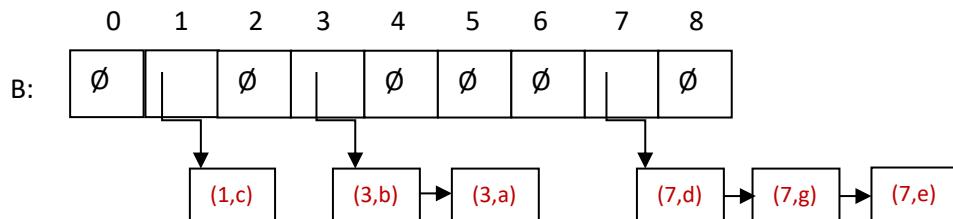
- Se folosește un sir (tablou unidimensional = vector) de siruri (sau liste, numite în contextul algoritmului *bucket-uri*) auxiliar, B, de dimensiune N => B[0...N-1]
- Într-un prim pas, fiecare pereche din S se mută în B în lista de pe indexul corespunzător (egal cu cheia), adică în lista (B[c]), adăugându-se la finalul acesteia.

B la finalul primului pas, pentru exemplul dat:



- În cel de-al doilea pas, se parcurge B (de la 0 la N-1) și se mută, în ordine, perechile din fiecare subșir (adică element sir) al lui B la finalul secvenței S (care a fost golită la pasul anterior).

Observăm că de-al doilea pas conduce la obținerea sirului ordonat: (1, c) (3, b) (3, a) (7, d) (7, g) (7, e) .



Pentru a descrie în Pseudocod algoritmul, presupunem următoarele operații pe elementele de tip listă ale tabloului B și pe secvența S.

- vidă (Listă)

- prim (Listă): TPozitie
- şterge (Listă, TPozitie)
- adaugăSfărşit(Listă, (c,v))

Cu alte cuvinte, considerăm că elementele tabloului B și secvența S sunt de tip generic, de tipul (TAD) Listă.

Descrierea Pseudocod a algoritmului BucketSort, conform ideii descrise mai sus:

Algoritm BucketSort(S, N) este:

```
//presupunem că sirul B există
//Primul pas al algoritmului (conform ideii algoritmului):
Câttimp ~ vîda (S) execută: //cât timp mai există perechi în secvență / lista S
    p ← prim (S)           //obținem poziția primeia dintre ele
    (c, v) ← şterge(S, p)   //o ștergem din secvență
    adaugaSfărşit(B[c], (c,v)) //și o adăugăm la sfârșitul listei B[c], adică a
listei de pe indexul egal cu cheia
    sf_câttimp
//Al doilea pas al algoritmului (conform ideii algoritmului):
Pentru i ← 0, N-1, execută: //pentru fiecare listă din B, pe rând,
    Câttimp ~ vîda (B[i]) execută: //cât timp mai există perechi în lista
currentă
    p ← prim (B[i])           //obținem poziția primeia dintre ele
    (c, v) ← şterge (B[i], p) //o ștergem din listă
    adaugaSfărşit (S, (c,v)) //și o adăugăm la sfârșitul secvenței /
listei S
    sf_câttimp
sf_pentru
sf_algoritm
```

Complexitate:  $\Theta(N + n)$

#### **Justificare:**

- Prima structură repetitivă, *Câttimp*, efectuează  $n$  pași,  $n$  fiind lungimea / numărul de elemente ale listei  $S$

- Operațiile *vîdă*, *prim*, *sterge* și *adaugă* se pot efectua în timp constant. De exemplu, dacă  $S$  și listele din  $B$  sunt reprezentate ca LDI alocate dinamic (sau ca LSI alocate dinamic, dar reținându-se și pointerul la ultimul nod), atunci atât ștergerea de la început, cât și adăugarea la sfârșit se vor efectua în timp constant.

=> Complexitatea primului pas la algoritmului este  $\Theta(n)$

- A doua structură repetitivă, *Pentru*, efectuează  $N$  pași,  $N-1$  fiind limita superioară pentru valorile întregi ale cheilor, iar  $N$ , lungimea tabloului  $B$

- Structura repetitivă interioară, *Câttimp*, efectuează la fiecare pas, pentru fiecare  $i$ , un număr de pași egal cu lungimea listei  $B[i]$  => În total, cele 3 operații din *Câttimp* se efectuează de un număr de ori egal cu suma lungimilor listelor din  $B$ , adică  $B[0], B[1], \dots, B[N-1]$ . Dar, având în vedere faptul că în aceste liste, inițial vide, au fost distribuite cele  $n$  perechi din  $S$ , în primul pas al algoritmului, suma lungimilor lor este  $n$ . Revenind, cele 3 operații din *Câttimp*, care se pot efectua în timp constant, se efectuează, în total de  $n$  ori

=> Complexitatea celui de-al doilea pas al algoritmului este  $\Theta(N+n) \Leftrightarrow \Theta(\max\{N,n\})$  (a se observa similaritatea cu problema 4 din seminarul 2)

=> Complexitatea algoritmului BucketSort este  $\Theta(N+n) \Leftrightarrow \Theta(\max\{N,n\})$

Dacă  $N \in O(n)$  => Complexitatea algoritmului BucketSort este  $\Theta(n)$ , deci **liniară**.

#### Observații:

- ◆ Pentru a aplica întocmai algoritmul prezentat, este necesar ca cheile să fie numere naturale cel mult egale cu  $N-1$ , acesta pentru a avea o corespondență directă între chei și indecsi în sirul  $B$  (adică a adăuga, în primul pas al algoritmului, o pereche  $(c,v)$  în lista  $B[c]$ )



Dar dacă am avea chei din mulțimea  $\{a, a+1, \dots, b\}$ , a și b fiind numere naturale? Cum am putea adapta algoritmul (astfel încât să gestionăm eficient spațiul de memorare)? Care este numărul minim de indecsi de care avem nevoie în  $B$ ? În lista de pe care dintre indecsii din  $B$  am adăuga o pereche  $(c,v)$  ?

- ✓ Numărul minim de indecsi de care avem nevoie în  $B$  este egal cu numărul de valori posibile pentru  $c$ , adică  $b-a+1$ . Astfel, vom avea în  $B$  listele  $B[0], B[1], \dots, B[b-a]$ .
- ✓ În primul pas al algoritmului, o pereche  $(c,v)$  se va adăuga în lista  $B[c-a]$ . Observăm că o pereche având cheia  $a$  va fi adăugată în lista  $B[0]$ , iar o pereche având cheia  $b$  va fi adăugată în lista  $B[b-a]$ .



Dar dacă am avea chei din mulțimea  $\{-a, -a+1, \dots, a-1, a\}$ , a fiind un număr natural? Care este numărul minim de indecsi de care avem nevoie în  $B$ ? În lista de pe care dintre indecsii din  $B$  am adăuga o pereche  $(c,v)$  ?

- ✓ Numărul minim de indecsi de care avem nevoie în  $B$  este egal cu numărul de valori posibile pentru  $c$ , adică  $2*a+1$ . Astfel, vom avea în  $B$  listele  $B[0], B[1], \dots, B[2*a]$ .
- ✓ În primul pas al algoritmului, o pereche  $(c,v)$  se va adăuga în lista  $B[c+a]$ . Observăm că o pereche având cheia  $-a$  va fi adăugată în lista  $B[0]$ , iar o pereche având cheia  $a$  va fi adăugată în lista  $B[2*a]$ .



Dar dacă am avea chei din mulțimea  $\{'A', 'B', \dots, 'Z'\}$ ? Care este numărul minim de indecsi de care avem nevoie în  $B$ ? În lista de pe care dintre indecsii din  $B$  am adăuga o pereche  $(c,v)$  ?

- ✓ Numărul minim de indecsi de care avem nevoie în  $B$  este egal cu numărul de valori posibile pentru  $c$ , adică cu numărul de litere din alfabet. Considerăm alfabetul latin modern, care conține 26 de litere. Astfel, vom avea în  $B$  listele  $B[0], B[1], \dots, B[25]$ .
- ✓ În primul pas al algoritmului, o pereche  $(c,v)$  se va adăuga în lista  $B[\text{ASCII}(c)-\text{ASCII}('A')]$ . Observăm că o pereche având cheia 'A' va fi adăugată în lista  $B[0]$ , iar o pereche având cheia 'Z' va fi adăugată în lista  $B[25]$ .

- ◆ O a doua observație este faptul că algoritmul păstrează ordinea inițială a perechilor având aceeași cheie, ceea ce înseamnă că, în versiunea prezentată, BucketSort este un algoritm de sortare **stabil**.

În general, un algoritm de sortare se numește **stabil** dacă păstrează ordinea inițială a elementelor egale, în raport cu relația de ordine după care se efectuează sortarea.



Cum am putea adapta algoritmul prezentat pentru a-l putea aplica pentru a ordona un sir de numere reale din intervalul  $[0, 1]$ ? **Indicație:** se alege prima cifră de după virgulă pe post de cheie (fictivă). Care este numărul de indecși de care avem nevoie în  $B$ ? În lista de pe care dintre indecșii din  $B$  am adăuga o pereche  $(c, v)$ ?

- ✓ Vom avea 10 liste (sau *bucket-uri*) în  $B$ :  $B[0], B[1], \dots, B[9]$ , una pentru fiecare valoare posibilă pentru prima cifră de după virgulă.
- ✓ O valoare  $v$  se va **insera**, în primul pas al algoritmului, în lista **ordonată**  $B[[v*10]]$  (indexul este egal cu partea întreagă a produsului  $v*10$ ). Adăugarea la final poate fi substituită de inserare într-o listă ordonată, astfel încât să obținem liste (*bucket-uri*) ordonate. O alternativă ar fi să adăugam în continuare la sfârșit, dar să sortăm *bucket-urile* ulterior primului pas al algoritmului.

În această variantă, algoritmul BucketSort **nu** este unul stabil (conform definiției de mai sus).

Observații:

- Dacă datele de intrare urmează o distribuție uniformă în intervalul  $[0, 1]$ , se obține un timp de execuție liniar.
- Se poate generaliza, alegând ca  $B$  să aibă dimensiune  $N$  (care poate fi chiar  $n$ , adică numărul de elemente din lista de ordonat). În acest caz, BucketSort divizează intervalul  $[0,1]$  în  $N$  (eventual  $n$ ) subintervale egale și apoi distribuie cele  $n$  elemente în aceste subintervale.



În acest caz, dacă  $B$  are dimensiune  $N$ , deci indici între 0 și  $N-1$ , în lista de pe care dintre indici va fi distribuită o valoare  $v$  din  $[0,1]$ ?

- ✓  $v \rightarrow B[[v*N]]$  (indexul este egal cu partea întreagă a produsului  $v*N$ )

Puteți vizualiza aplicarea algoritmului pe un exemplu aici:

<https://www.youtube.com/watch?v=VuXbEb5ywrU>

În acest exemplu, elementele se adaugă în *bucket-uri* la sfârșit, urmând ca la finalul primului pas al algoritmului fiecare *bucket* să fie ordonat individual., folosind sortarea prin inserție. După cum am mai precizat, o alternativă ar fi fost să se insereze elementele în *bucket-uri* astfel încât acestea să rămână ordonate.

## B. Sortare Lexicografică

### Enunțul problemei

Se dă o secvență  $S$  de  $n$   $d$ -tpluri, adică tupluri cu  $d$  componente / de dimensiune  $d$ , de forma  $(x_1, x_2, \dots, x_d)$ .

Se cere să se sorteze  $S$  în ordine lexicografică.

Observație:  $(x_1, x_2, \dots, x_d) < (\text{lexicografic}) (y_1, y_2, \dots, y_d) \Leftrightarrow x_1 < y_1 \vee (x_1 = y_1 \wedge ((x_2, \dots, x_d) < (y_2, \dots, y_d)))$

- Se face compararea după prima dimensiune, după aceea după a 2-a, și așa mai departe.

### Exemplu

Se dă secvența  $S$ :  $(7, 4, 6), (5, 1, 5), (2, 4, 6), (2, 1, 4), (3, 2, 4)$

$n = 5$

$d = 3$  (componentele secvenței  $S$  sunt triplete)

$\Rightarrow (2, 1, 4), (2, 4, 6), (3, 2, 4), (5, 1, 5), (7, 4, 6)$

### Ideea algoritmului

Vom folosi:

- $C_i$  – obiect comparator (relație de ordine) care compară 2 tupluri după dimensiunea  $i$ ; avem câte un astfel de obiect comparator pentru fiecare dintre cele  $d$  dimensiuni
- $stableSort(S, c)$  – algoritm de sortare **stabilă** care folosește un comparator  $c$

- ✓ Sortarea Lexicografică apelează algoritmul  $stableSort$  de  $d$  ori, o dată pentru fiecare dimensiune



Cu care dintre dimensiuni începem sortarea? În ce ordine considerăm cele  $d$  dimensiuni în apelurile algoritmului  $stableSort$ ?

- ✓ În ordine **înversă**, adică de la cea mai puțin semnificativă (dimensiunea / componenta  $d$ ) la cea mai semnificativă (prima dimensiune / componentă).

**Observație:** Dacă am începe ordonarea cu prima dimensiune și finaliza-o cu ultima dimensiune, pentru exemplul considerat, s-ar obține:

- Inițial:  $S: (7, 4, 6), (5, 1, 5), (2, 4, 6), (2, 1, 4), (3, 2, 4)$
- Ulterior ordonării după **prima** componentă:  $S: (2, 4, 6), (2, 1, 4), (3, 2, 4), (5, 1, 5), (7, 4, 6)$
- Ulterior ordonării după **a doua** componentă:  $S: (2, 1, 4), (5, 1, 5), (3, 2, 4), (2, 4, 6), (7, 4, 6)$
- Ulterior ordonării după **a treia** componentă:  $S: (2, 1, 4), (3, 2, 4), (5, 1, 5), (2, 4, 6), (7, 4, 6)$

Observăm că ordinea obținută **nu** este cea lexicografică.

Dacă, însă, începem cu ultima dimensiune și finalizăm cu prima, obținem:

- Inițial: S: (7, 4, 6), (5, 1, 5), (2, 4, 6), (2, 1, 4), (3, 2, 4)
- Ulterior ordonării după **a treia** componentă: S: (2, 1, 4), (3, 2, 4), (5, 1, 5), (7, 4, 6), (2, 4, 6)
- Ulterior ordonării după **a doua** componentă: S: (2, 1, 4), (5, 1, 5), (3, 2, 4), (7, 4, 6), (2, 4, 6)
- Ulterior ordonării după **prima** componentă: S: (2, 1, 4), (2, 4, 6), (3, 2, 4), (5, 1, 5), (7, 4, 6)

Observăm că ordinea obținută este cea lexicografică.



Este necesar ca algoritmul de sortare care se aplică (pentru a sorta după o anumită dimensiune) să fie un algoritm de sortare stabil?

- ✓ Da.

Altfel, în cazul exemplului considerat, ulterior sortării după prima componentă s-ar putea obține:

- Ulterior ordonării după **a doua** componentă: S: (2, 1, 4), (5, 1, 5), (3, 2, 4), (7, 4, 6), (2, 4, 6)
- Ulterior ordonării după **prima** componentă: S: (2, 4, 6), (2, 1, 4), (3, 2, 4), (5, 1, 5), (7, 4, 6)

Observăm că ordinea obținută **nu** este cea lexicografică.

Intuiție: Dacă algoritmul de sortare folosit nu este stabil, atunci la sortarea după dimensiunea  $i$  nu asigură păstrarea ordinii obținute prin sortarea după dimensiunea  $i+1$ .

Descrierea Pseudocod a algoritmului se Sortare Lexicografică:

Algoritm LexicographicSort(S) :

```
Pentru i ← d, 1 exec: //pentru fiecare dimensiune i a tuplurilor din S, începând cu  
ultima și finalizând cu prima  
    stableSort(S, ci) //sortăm tuplurile din S după dimensiunea i  
    sf_pentru  
sf_subalgoritm
```

### Exemplu

- Inițial: S: (7, 4, 6), (5, 1, 5), (2, 4, 6), (2, 1, 4), (3, 2, 4)  $\Rightarrow d = 3$
- Ulterior ordonării după  $d=3$  componentă: S: (2, 1, 4), (3, 2, 4), (5, 1, 5), (7, 4, 6), (2, 4, 6)
- Ulterior ordonării după  $d=2$  componentă: S: (2, 1, 4), (5, 1, 5), (3, 2, 4), (7, 4, 6), (2, 4, 6)
- Ulterior ordonării după  $d=1$  componentă: S: (2, 1, 4), (2, 4, 6), (3, 2, 4), (5, 1, 5), (7, 4, 6)

Complexitatea timp a algoritmului de Sortare Lexicografică (în funcție de complexitatea timp a algoritmului de sortare stabilă folosit):

- ✓  $\Theta(d * T(n))$ , unde  $T(n)$  – complexitatea algoritmului *stableSort*

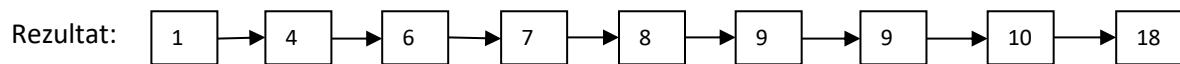
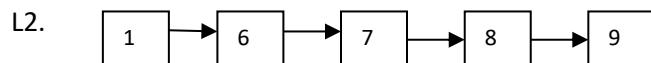
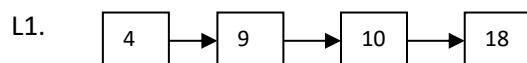
### C. Radix Sort

- *RadixSort* este o specializare a sortării lexicografice, în care algoritmul de sortare stabilă folosit este BucketSort => toate elementele din tupluri trebuie să fie numere naturale din mulțimea {0, 1,..., N-1} (altfel, este necesară adaptarea asocierilor din chei și indecșii).
- Complexitatea *RadixSort* va fi, prin urmare  $\Theta(d * (n + N))$

### Interclasarea a două liste simplu înlăntuite, alocate dinamic

Enunț Scrieți o procedură care interclasează două liste ordonate (LO) reprezentate pe liste simplu înlăntuite (LSI) alocate dinamic. Analizați complexitatea operației.

#### Exemplu



Reprezentarea TAD LO (Listă Ordonată) folosind LSI alocată dinamic:

#### NodT:

e: TComparabil  
urm:  $\uparrow$ NodT

#### LO:

prim:  $\uparrow$ NodT  
R: Relație: TComparabil x TComparabil  $\rightarrow \{A, F\}$

Descrierea în Pseudocod a algoritmului de interclasare a două liste ordonate L1 și L2, reprezentate la LSI alocate dinamic, astfel încât interclasarea să nu afecteze L1 și L2, ci să construiască o nouă listă rezultat, LR:

Subalgoritm interclasare (L1, L2, LR):

```
currentL1 ← L1.prim      //începem parcurgerea listei L1 cu primul ei nod
currentL2 ← L2.prim      //începem parcurgerea listei L2 cu primul ei nod
primLR ← NIL              //initial, lista rezultat este vidă
ultimLR ← NIL              //reținem și pointerul la ultimul nod al listei rezultat, pentru a
eficientiza adăugarea la finalul ei
    când currentL1 ≠ NIL și currentL2 ≠ NIL execută //până când mai avem noduri de
parcurs în ambele liste
```

```

aloca(nou) //alocăm un nou nod
[nou].urm ← NIL //pe care îl vom adăuga la finalul listei rezultat, după ce îl
completăm cu informația utilă
dacă L1.R([currentL1].e,[currentL2].e) atunci //verificăm care dintre cele 2
elemente curente din L1 și L2 este mai mic în raport cu relația de ordine R
[nou].e ← [currentL1].e //dacă acesta este elementul curent din L1, atunci
el va fi completat ca informație utilă în noul nod
currentL1 ← [currentL1].urm //și avansăm în L1
altfel
[nou].e ← [currentL2].e //elementul curent din L2 va fi completat ca
informație utilă în noul nod
currentL2 ← [currentL2].urm //și avansăm în L2
sf_dacă
dacă primLR = NIL atunci //verificăm dacă noul nod pe care îl adăugăm îl LR este
și primul
primLR ← nou //în caz afirmativ, inițializăm primLR (cu pointerul la primul nod
al listei rezultat)
altfel
[ultimoLR].urm ← nou //altfel, facem legătura de la ultimul nod actual din LR
la acest nod
sf_dacă
ultimoLR ← nou //noul nod devine ultimul nod din lista rezultat
sf_câttimp
//După ce am epuizat cel puțin una dintre liste, ne rămâne să parcurgem lista în continuare
lista rămasă (dacă au rămas noduri neparcuse între-una dintre liste) și să preluăm elementele din
nodurile rămase în noi noduri pe care să le adăugăm la sfârșitul listei rezultat
current ← currentL1 //presupunem că mai avem noduri de parcurs în L1
dacă currentL1 = NIL atunci //în caz contrar
    current ← currentL2 //presupunem că mai avem noduri de parcurs în L2
sf_dacă
câttimp current ≠ NIL execută //cât timp mai avem noduri de parcurs în lista ne-
epuizată (OBS: dacă am parcurs integral ambele LSI, acest ciclu nu va efectua niciun pas)
alocă (nou) //alocăm un nou nod
[nou].urm ← NIL //pe care îl vom adăuga la finalul listei rezultat, după ce îl
completăm cu informația utilă
[nou].e ← [current].e //elementul din nodul curent din LSI încă neparcursă integral
va fi completat ca informație utilă în noul nod
current ← [current].urm //și avansăm în LSI de intrare
dacă primLR = NIL atunci //verificăm dacă noul nod pe care îl adăugăm îl LR este
și primul (<= dacă una dintre L1 și L2 a fost vidă)
primLR ← nou //în caz afirmativ, inițializăm primLR (cu pointerul la primul nod
al listei rezultat)
altfel
[ultimoLR].urm ← nou //altfel, facem legătura de la ultimul nod actual din LR
la acest nod
sf_dacă
ultimoLR ← nou //noul nod devine ultimul nod din lista rezultat
sf_câttimp
LR.prim ← primLR //inițializăm câmpul prim al listei rezultat cu pointerul la primul ei nod
//OBS (considerăm că s-a inițializat câmpul R în prealabil, în constructor)
Sf_subalgoritm

```

**Complexitatea algoritmului de interclasare:**

- ✓  $\Theta(n + m)$ , unde  $n$  este lungimea listei L1, iar  $m$  este lungimea listei L2.

# SDA - Seminar 6 - TD

---

## Conținut:

- Iterator DO reprezentat sub formă de TD, cu rezolvare coliziuni prin liste independente
- Dicționar reprezentat sub formă de TD, cu rezolvare coliziuni prin liste întrepătrunse

### 1. Iterator pe Dicționar Ordonat (DO) reprezentat sub formă de tabelă de dispersie (TD), cu rezolvare coliziuni prin liste independente.

- Presupunem că:
  - Memorăm doar cheile
  - Avem chei numere naturale
  - Ordinea impusă cheilor este “<” (cheile fiind distincte)

#### Exemplu:

- Presupunem că dorim să adăugăm în dicționar următoarele 9 chei: 5, 28, 19, 15, 20, 33, 12, 17, 10
- , iar dicționarul este reprezentat pe o TD cu
  - $m = 9$  locații
  - și optăm pentru
  - dispersia prin diviziune



În cazul dispersiei prin diviziune, cum este definită funcția de dispersie  $d(d(c)=?)$  ?

✓  $d(c) = c \text{ mod } m$



Care sunt valorile de dispersie pentru cele 9 chei care se doresc a fi adăugate în DO?

c	5	28	19	15	20	33	12	17	10
d(c)									

c	5	28	19	15	20	33	12	17	10
d(c)	5	1	1	6	2	6	3	8	1

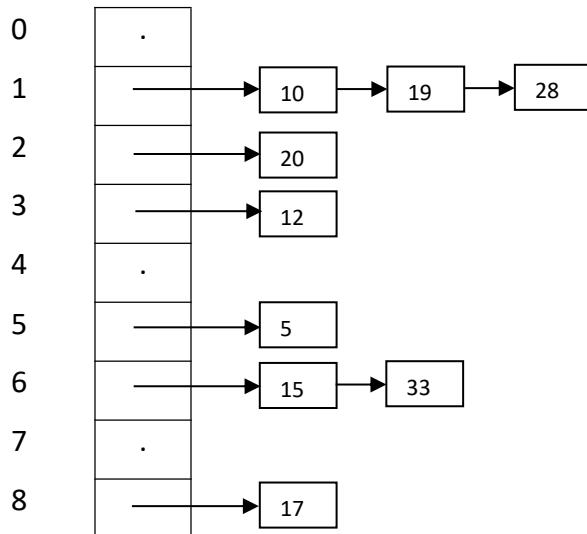
- **Observație:** Secvența d(c) formată de valorile de dispersie conține valori care se repetă. Acestea corespund coliziunilor. Având în vedere că discutăm în contextul unei TD cu rezolvare coliziuni prin liste independente, cheile aparținând unei coliziuni vor fi memorate în aceeași listă (independent de cheile având alte valori de dispersie).



Cum arată TD cu ajutorul căreia se reprezintă Dicționarul Ordonat, ulterior inserării cheilor exemplificate ( 5, 28, 19, 15, 20, 33, 12, 17, 10 )?

0	()
1	()
2	()
3	()
4	()
5	()
6	()
7	()
8	()

**TD:**



**Observație:** Cheile se inserează în listele independente astfel încât să se respecte relația impusă Dicționarului Ordonat. Cu alte cuvinte, toate listele independente vor fi (independent) ordonate.



Care este ordinea de parcurgere a cheilor din DO care se așteaptă a fi oferită de Iterator?

- ✓ Elementele ar trebui să fie parcuse de Iterator în această ordine: 5, 10, 12, 15, 17, 19, 20, 28, 33, adică în ordine crescătoare.



Cum se obține această secvență ordonată a cheilor pornind de la TD ilustrată anterior?

- ✓ Se observă că lista ordonată a cheilor se obține prin interclasarea tuturor listelor independente (cu ajutorul cărora se memorează cheile care intră în coliziune).  
Așadar, **implementarea Iteratorului pe DO se poate reduce la interclasarea prealabilă** (în constructor) a listelor independente, într-o nouă listă, ordonată, urmată de parcurgerea ei .



Considerând că reprezentăm listele independente ca Liste Simplu Înlănțuite (LSI) alocate dinamic, care este reprezentarea Dicționarului Ordonat?

#### Reprezentare DO:

##### NodT:

c: TCheie (*presupunând că memorăm doar cheile; altfel, adăugăm v:TValoare*)  
urm:  $\uparrow$ NodT

##### DicționarOrdonat:

m: Intreg (*dimensiunea tabelei*)  
l :  $(\uparrow$ NodT) $^[]$  (*vectorul de liste independente*)  
d: TFuncție: TCheie  $\rightarrow \{0,1,\dots,m-1\}$  (*funcția de dispersie*)  
R: Relație: TCheie  $\times$  TCheie  $\rightarrow \{A, F\}$  (*relația de ordine impusă cheilor*)



Cum putem reprezenta Iteratorul pe Dicționarul Ordonat?

##### IteratorDicționar:

d: DicționarOrdonat

I:  $\uparrow \text{NodT}$  (*pointer la primul nod al listei ordonate obținută prin interclasare*) (sau, alternativ, TListă)  
currentNod:  $\uparrow \text{NodT}$  (*relația de ordine impusă cheilor*)



Cum implementăm constructorul Iteratorului având această reprezentare?

```
subalgoritm creeaza(it, d) :  
    it.d ← d  
    interclaseazaListe (d, it.l)  
    it.currentNod ← it.l (sau it.l.prim, depinzând de tipul ales,  
    al rezultatului operației interclasează)  
sf_subalgoritm
```



Care vor fi complexitățile timp ale operațiilor *valid*, *următor*, *element*?

✓ Operațiile *valid*, *următor*, *element* vor avea complexitate constantă:  $\Theta(1)$ .



Care sunt posibilitățile de interclasare ale listelor independente (de implementare a operației *interclaseazaListe*)?

✓ *interclaseazăListe* poate interclasă listele independente ordonate :

1. pe rând, adică interclasând prima listă cu a 2-a, după care lista rezultată din interclasarea primelor două liste cu cea de-a 3-a și.m.d.
2. Concomitant, adică interclasând toate listele deodată folosind un ansamblu



Care va fi complexitatea interclasării (pentru fiecare dintre cele două variante menționate anterior)?

✓ Complexitatea interclasării:

$$\left. \begin{array}{l} \text{TD cu } m \text{ poziții} \\ \text{Dicționar cu } n \text{ elemente} \end{array} \right\} \Rightarrow \text{nr mediu de elemente într-o listă: } \frac{n}{m} = \alpha \text{ (factor de încărcare)}$$

1. Interclasare prima lista cu a 2-a, etc.:

- $\text{lista1} + \text{lista2} \Rightarrow \text{lista12} \rightarrow \alpha + \alpha = 2\alpha$  (operații elementare)
- $\text{lista12} + \text{lista3} \Rightarrow \text{lista123} \rightarrow 2\alpha + \alpha = 3\alpha$

- $\text{lista123} + \text{lista4} \Rightarrow \text{lista1234} \rightarrow 3\alpha + \alpha = 4\alpha$
- ...

$$\Rightarrow \text{Se obțin, în total : } 2\alpha + 3\alpha + \dots + m\alpha \approx \left. \frac{\frac{m*(m+1)}{2}\alpha}{\alpha = \frac{n}{m}} \right\} \rightarrow \frac{m(m+1)}{2} \cdot \frac{n}{m} \Rightarrow \in \theta(n * m) \approx \theta(n)$$

(presupunând m – constant)

2. Dicționar reprezentat pe TD, în varianta de rezolvare a coliziunilor prin liste întrepătrunse

- Presupunem, din nou, că:
    - Memorăm doar cheile (doar pentru ilustrarea exemplului)
    - Avem chei numere naturale

## **Exemplu:**

- Presupunem că dorim să adăugăm în dicționar următoarele 7 chei: 5, 18, 16, 15, 13, 31, 26
  - , iar dicționarul este reprezentat pe o TD cu
    - m = 13 locații
    - și optăm pentru
    - dispersia prin diviziune



Care sunt valorile de dispersie pentru cele 7 chei care se doresc a fi adăugate în Dictionar?

<b>c</b>	5	18	16	15	13	31	26
<b>d(c)</b>							

<b>c</b>	5	18	16	15	13	31	26
<b>d(c)</b>	5	5	3	2	0	5	0



Cum arată TD cu ajutorul căreia se reprezintă Dicționarul, ulterior adăugării cheii 5?

➤ Adăugăm cheia 5

- Valoarea de dispersie pentru 5 este 5
  - Slotul având indexul 5 este liber
  - Adăugăm cheia 5 în slotul având indexul 5
  - primLiber = 0 (rămâne neschimbat)



Cum arată TD cu ajutorul căreia se reprezintă Dicționarul, ulterior adăugării cheii 18?

➤ Adăugăm cheia 18

- Valoarea de dispersie pentru 18 este 5
- Slotul având indexul 5 este ocupat
- Prin urmare, adăugăm 18 pe slotul având indexul primLiber = 0
- primLiber = 01 (actualizăm valoarea primului slot liber)
- Având în vedere că 18 se dispersează pe indexul 5, începând de la indexul 5 trebuie să ajungem, urmând legăturile, la cheia 8 (deci la indexul 0). Cu alte cuvinte, adăugăm cheia 18 (de pe indexul 0) la finalul înlănțuirii începând de pe indexul 5 (egal cu valoarea ei de dispersie). Aceasta se rezumă la urm[5]=0.

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18					5							
urm	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1



Cum arată TD cu ajutorul căreia se reprezintă Dicționarul, ulterior adăugării cheii 16?

➤ Adăugăm cheia 16

- Valoarea de dispersie pentru 16 este 3
- Slotul având indexul 3 este liber
- Adăugăm cheia 16 în slotul având indexul 3
- primLiber = 01 (rămâne neschimbat)

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18			16		5							
urm	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1



Cum arată TD cu ajutorul căreia se reprezintă Dicționarul, ulterior adăugării cheii 15?

➤ Adăugăm cheia 15

- Valoarea de dispersie pentru 15 este 2
- Slotul având indexul 2 este liber

- Adăugăm cheia 15 în slotul având indexul 2
- primLiber = 01 (rămâne neschimbat)

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18		15	16		5							
urm	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1



Cum arată TD cu ajutorul căreia se reprezintă Dicționarul, ulterior adăugării cheii 13?

➤ Adăugăm cheia 13

- Valoarea de dispersie pentru 13 este 0
- Slotul având indexul 0 este ocupat
- Prin urmare, adăugăm 13 pe slotul având indexul primLiber = 1
- primLiber = 014 (cel mai mic index mai mare decât 1, al unui slot liber)
- Având în vedere că 13 se dispersează pe indexul 0, începând de la indexul 0 trebuie să ajungem, urmând legăturile, la cheia 13 (deci la indexul 1). Cu alte cuvinte, adăugăm cheia 13 (de pe indexul 1) la finalul înlățuirii începând de pe indexul 0 (egal cu valoarea ei de dispersie). Aceasta se rezumă la urm[0]=1.

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18	13	15	16		5							
urm	-1	-1	-1	-1	-1	0	-1	-1	-1	-1	-1	-1	-1



Cum arată TD cu ajutorul căreia se reprezintă Dicționarul, ulterior adăugării cheii 31?

➤ Adăugăm cheia 31

- Valoarea de dispersie pentru 31 este 5
- Slotul având indexul 5 este ocupat
- Prin urmare, adăugăm 31 pe slotul având indexul primLiber = 4
- primLiber = 0146 (cel mai mic index mai mare decât 4, al unui slot liber)
- Având în vedere că 31 se dispersează pe indexul 5, începând de la indexul 5 trebuie să ajungem, urmând legăturile, la cheia 31 (deci la indexul 4). Cu alte cuvinte, adăugăm cheia 31 (de pe indexul 4) la finalul înlățuirii începând de pe indexul 5 (egal cu valoarea ei de dispersie). Având în vedere că urm[5]=0, urm[0] = 1 și abia urm[1]=-1, aceasta se rezumă la urm[1]=4.

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18	13	15	16	31	5							

<b>urm</b>	-4 1	-4 4	-1	-1	0	-1	-1	-1	-1	-1	-1	-1
------------	------	------	----	----	---	----	----	----	----	----	----	----



Cum arată TD cu ajutorul căreia se reprezintă Dicționarul, ulterior adăugării cheii 26?

➤ Adăugăm cheia 26

- Valoarea de dispersie pentru 26 este 0
- Slotul având indexul 0 este ocupat
- Prin urmare, adăugăm 26 pe slotul având indexul primLiber = 6
- primLiber = 01467 (cel mai mic index mai mare decât 6, al unui slot liber)
- Având în vedere că 26 se dispersează pe indexul 0, începând de la indexul 0 trebuie să ajungem, urmând legăturile, la cheia 26 (deci la indexul 6). Cu alte cuvinte, adăugăm cheia 26 (de pe indexul 6) la finalul înlănțuirii începând de pe indexul 0 (egal cu valoarea ei de dispersie). Având în vedere că urm[0]=1, urm[1] = 4 și abia urm[4]=-1, aceasta se rezumă la urm[4]=6.

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18	13	15	16	31	5	26						
urm	-4 1	-4 4	-1	-1	-4 6	-4 0	-1	-1	-1	-1	-1	-1	-1

**Observații:**

- primLiber se actualizează considerând pozițiile libere de la stânga la dreapta, ceea ce înseamnă că spațiul liber nu este înlănțuit, ci se gestionează secvențial
- Într-o înlănțuire putem avea elemente care aparțin unor coliziuni diferite. De exemplu, coliziunea care începe pe poziția 5: 5 (valoare dispersie: 5) → 18 (valoare dispersie: 5) → 13 (valoare dispersie: 0) → 31 (valoare dispersie: 5) → 26 (valoare dispersie: 0) conține chei care se dispersează atât pe poziția 5, cât și pe poziția 0.



Care este reprezentarea TAD Dicționar, considerând TD în varianta de rezolvare a coliziunilor prin liste întrepătrunse?

✓ **Reprezentare Dicționar:**

**TElement:**

c: TCheie

v: TValoare

**Dicționar:**

m: Întreg (dimensiunea tabelei)

e: (TElement)[]

urm:{0,..., m-1}[]

primLiber: Întreg {0,...,m}

d: TFuncție: TCheie -> {0,..., m-1}

- **Observație:** În stabilirea domeniului valorilor pentru primLiber, s-a presupus că primLiber=m indică faptul că TD este plină.



Cum implementăm constructorul Dicționarului având această reprezentare?

**subalgoritm** creează (d) :

```

@ se inițiază funcție de dispersie d
@ se inițiază m
pentru i ← 0, m-1 execută
    d.e[i] ← NULL_TElement
    d.urm[i] ← -1
sf_pentru
    d.primLiber ← 0
sf_subalgoritm
```



Care este complexitatea timp a operației *creează*?

- ✓ Complexitate:  $\Theta(m)$ , m reprezentând dimensiunea tabelei



Care este descrierea Pseudocod a operației de căutare din interfața Dicționarului?

**Funcția** caută(d, c) :

```

    i ← d.d(c) //Calculăm valoarea de dispersie pentru cheia c
    câttimp (i ≠ -1 și d.e[i].c ≠ c) execută //Cât timp mai sunt
    elemente în înlățuirea începând de la indexul egal cu valoarea de dispersie
    și nu am găsit cheia căutată
        i ← d.urm[i] //ne deplasăm la următorul element din înlățuire,
    urmând legăturile
        sf_câttimp
        dacă i = -1 atunci //în cazul în care cheia nu a fost găsită
            caută ← NULL_TValoare //semnalăm aceasta returnând
        NULL_TValoare
        altfel
            caută ← d.e[i].v //altfel, returnăm valoarea asociată ei
sf_funcție
```



Care este complexitatea operației *caută*?

- ✓ Complexitate:  $O(m)$ , dar în medie (în ipoteza dispersiei uniforme simple)  $\Theta(1)$

**Observație:** Implementarea operației de adăugare este descrisă în materialul aferent cursului 10, prin urmare continuăm cu operația de **ștergere**.

Exemplificăm operația de ștergere pentru cheia 5.

Pornim de la TD în această stare:

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18	13	15	16	31	5	26						
urm	1	4	-1	-1	6	0	-1	-1	-1	-1	-1	-1	-1

, indexul primului slot liber fiind `primLiber=7`.

**Observație:**

- Nu putem elibera slotul 5 marcând, totodată, `urm[5]` cu -1 deoarece am pierde legăturile înspre 18 și 31, nemaiputându-le găsi prin operația *caută*.

**Idea ștergerii** (pentru exemplul considerat; descrierea generică este prezentată în materialul aferent cursului 10):

- Căutăm elemente, în înlănțuirea începând de la poziția de pe care ștergem (în cazul nostru, 5) care se dispersează pe această poziție (adică pe poziția 5).
  - Dacă nu există astfel de elemente, ștergem elementul precum am șterge un nod dintr-o listă simplu înlănțuită, refăcând legăturile
  - Dacă există un astfel de element, atunci mutăm elementul pe poziția de unde dorim să ștergem și repetăm procesul de ștergere pentru poziția de unde am mutat elementul.

Așadar:

- Dorim să ștergem cheia 5, care este pe poziția 5

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18	13	15	16	31	5	26						
urm	1	4	-1	-1	6	0	-1	-1	-1	-1	-1	-1	-1

- Căutăm primul element care se dispersează pe poziția 5. Din 5, urmând legătura (`urm[5]=0`), ajungem la cheia 18, care se dispersează întocmai pe poziția 5.
- Așadar, mutăm (copiem) 18 pe poziția 5

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18	13	15	16	31	5	18	26					

<b>urm</b>	1	4	-1	-1	6	0	-1	-1	-1	-1	-1	-1	-1
------------	---	---	----	----	---	---	----	----	----	----	----	----	----

- În continuare, dorim să ștergem cheia 18 de pe poziția 0 (de unde am identificat-o, aducând-o pe poziția 5)
- Căutăm primul element care se dispersează pe poziția 0. Din 0, urmând legătura (`urm[0]=1`), ajungem la cheia 13, care se dispersează întocmai pe poziția 0.
- Așadar, mutăm (copiem) 13 pe poziția 0

	0	1	2	3	4	5	6	7	8	9	10	11	12
<b>c</b>	18 13	13	15	16	31	5 18	26						
<b>urm</b>	1	4	-1	-1	6	0	-1	-1	-1	-1	-1	-1	-1

- În continuare, dorim să ștergem cheia 13 de pe poziția 1 (de unde am identificat-o, aducând-o pe poziția 0)
- Căutăm primul element care se dispersează pe poziția 1 (adică pe poziția de unde dorim să ștergem). Din 1, urmând legătura (`urm[1]=4`), ajungem la cheia 31 (de pe poziția 4), care nu se dispersează pe poziția 1. Din 4, urmând legătura (`urm[4]=6`), ajungem la cheia 26 (de pe poziția 6), care, din nou, nu se dispersează pe poziția 1. Cheia 26 este, însă, ultima cheie din înlățuire, având în vedere că `urm[6]=-1`. Prin urmare, deducem că nu există o cheie care se dispersează pe poziția 1. Așadar, ștergem poziția 1 din înlățuire (precum am șterge dintr-o LSI), modificând legăturile (`urm[0]<-urm[1] = 4`)

	0	1	2	3	4	5	6	7	8	9	10	11	12
<b>c</b>	18 13		15	16	31	5 18	26						
<b>urm</b>	1 4	4 -1	-1	-1	6	0	-1	-1	-1	-1	-1	-1	-1

Având în vedere că am eliberat un index mai mic decât `primLiber=7`, `primLiber` devine 1 (`primLiber=71`).



Care este descrierea Pseudocod a operației *sterge*, conform ideii prezentate anterior?

**subalgoritm** *sterge(d, c)* **este**

```
i ← d.d(c) //calculăm valoarea de dispersie pentru cheia c
j ← -1 //rețin precedentul lui i, întrucât, când ștergem, ne este
        necesar "nodul" anterior, pentru refacerea legăturii
        //în următoarea structură repetitivă, parcurgem tabela pentru a verifica
        dacă poziția i are vreun anterior
k ← 0
```

```

câttimp (k < d.m și j = -1) execută
  dacă d.urm[k] = i atunci
    j ← k
  altfel
    k ← k+1
sf_câttimp
  //în următoarea structură repetitivă, localizăm cheia care trebuie
  //ștearsă, actualizând și precedentul
  câttimp i ≠ -1 și d.e[i].c ≠ c execută
    j ← i
    i ← d.urm[i]
sf_câttimp
  //verificăm existența cheii
  dacă i = -1 atunci
    @cheia nu există
  altfel //în caz afirmativ,
    //căutăm, în înlănțuirea începând de pe poziția i o primă altă cheie
    //care se dispersează în i (poziția de șters)
    gata ← fals //folosim o variabilă booleană care devine adevărat
    când nu se dispersează nimic în i (poziția de șters)
  repetă
    p ← d.urm[i] //prima poziție verificată
    pp ← i //anteriorul poziției p
    câttimp p ≠ -1 și d.d(d.e[p].c) ≠ i execută //cât timp
    mai avem chei în înlănțuire și nu am găsit o cheie care se dispersează pe
    poziția i, avansăm în înlănțuire, urmând legăturile, cu poziția curentă și cu
    cea anterioară ei
      pp ← p
      p ← d.urm[p]
sf_câttimp
  dacă p = -1 atunci
    gata ← adevarat //marcăm situația în care nu există o
    //cheie care se dispersează în i (poziția de șters)
  altfel //altfel
    d.e[i] ← d.e[p] //aducem cheia (perechea) de pe poziția
    p pe poziția i
    j ← pp //poziția anterioară noii poziții de șters devine
    pp
    i ← p //iar noua poziție de șters devine p (poziția de pe
    care am adus cheia pe fosta poziție de șters)
  sf_dacă
până_când gata
  //nemaifiind chei care se dispersează pe poziția de șters, ștergem
  //această poziție din înlănțuire
  dacă j ≠ -1 atunci //în cazul în care are un precedent, refacem
  //legătura de la poziția precedentă (în înlănțuire) la următoarea poziție (în
  //înlănțuire)
    d.urm[j] ← d.urm[i]
sf_dacă
  //și marcam faptul că poziția i a fost eliberată

```

```
d.e[i] ← NULL_TElement
d.urm[i] ← -1
//în cazul în care am eliberat o poziție mai mică decât primLiber,
actualizăm valoarea primei poziții libere, dat fiind că gestionăm spațiul
liber de la stânga la dreapta.
dacă d.primLiber > i atunci
    d.primLiber ← i
sf_dacă
sf_dacă
sf_subalgoritm
```

## SDA - Seminar 7 - AB

- **Conținut:** Materialul curent exemplifică rezolvarea mai multor probleme cu **Arborei Binari** (AB).

1. Să se construiască arborele binar asociat unei expresii aritmetice conținând operatorii aritmetici +, -, \*, /, pornind de la forma poloneză post-fixată.



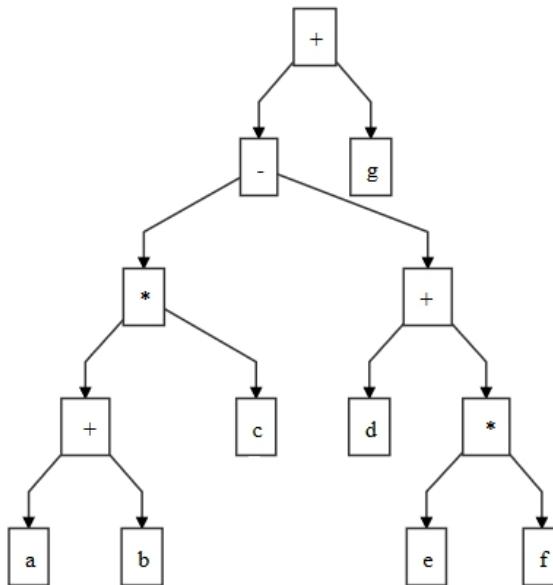
Considerând expresia aritmetică  $(a + b) * c - (d + e * f) + g$ , care este forma ei poloneză postfixată?

✓ Forma poloneză postfixată (FPP) a expresiei este: a b + c \* d e f \* + - g +

**Observație:** În forma postfixată, operatorii apar **după** operanzi.



Care este arborele binar asociat expresiei exemplificate?



### Justificare:

Considerând expresia  $(a + b) * c - (d + e * f) + g$ , observăm că:

- Se efectuează o sumă ( $\Rightarrow +$  în rădăcină, pe nivelul 0) între o diferență ( $\Rightarrow -$  ca fiu stâng pentru rădăcină) și  $g$  ( $\Rightarrow g$  ca fiu drept pentru rădăcină)
- Diferența se efectuează între un produs ( $\Rightarrow *$  ca fiu stâng pentru  $-$ ) și o sumă ( $\Rightarrow +$  ca fiu drept pentru  $-$ )

- Produsul se efectuează între o sumă ( $\Rightarrow +$  ca fiu stâng pentru \* de pe nivelul 2) și c ( $\Rightarrow \mathbf{c}$  ca fiu drept pentru \* de pe nivelul 2), iar suma se efectuează între d ( $\Rightarrow \mathbf{d}$  ca fiu stâng pentru + de pe nivelul 2) și un produs ( $\Rightarrow *$  ca fiu drept pentru + de pe nivelul 2)
- Suma se efectuează între a ( $\Rightarrow \mathbf{a}$  ca fiu stâng pentru + de pe nivelul 3) și b ( $\Rightarrow \mathbf{b}$  ca fiu drept pentru + de pe nivelul 3), iar produsul se efectuează între e ( $\Rightarrow \mathbf{e}$  ca fiu stâng pentru \* de pe nivelul 3) și f ( $\Rightarrow \mathbf{f}$  ca fiu drept pentru \* de pe nivelul 3)

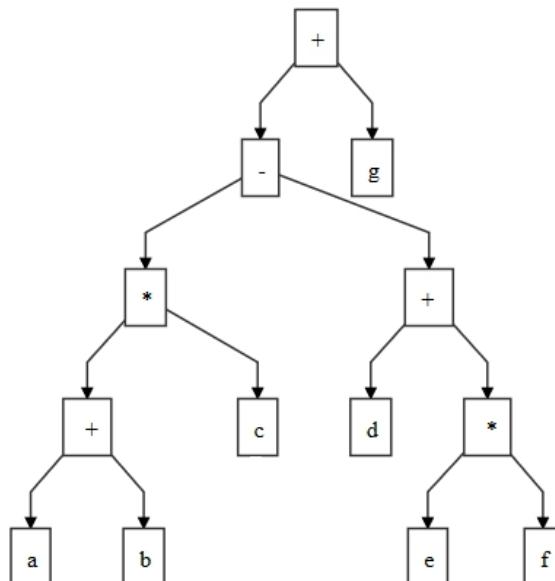
Observații: S-a considerat rădăcina ca fiind pe nivelul 0.

Dacă s-ar fi considerat operatorul - în rădăcină ar fi fost necesară transcrierea expresiei ca  $(a + b) * c - [(d + e * f) - g]$ .



Ce secvență se obține prin parcurgerea în postordine a arborelui obținut?

**Observație:** Parcurgerea în postordine presupune parcurgerea rădăcinii **după** parcurgerea celor (cel mult) doi fii (stâng și drept).



- ✓ Parcurgând arborele în postordine, se obține secvența: a b + c \* d e f \* + - g +, deci întocmai forma postfixată a expresiei.

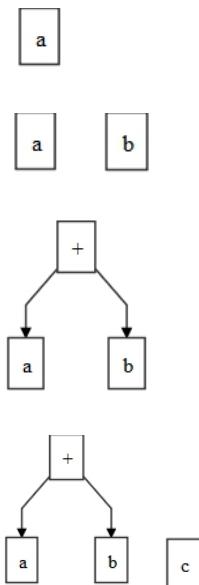


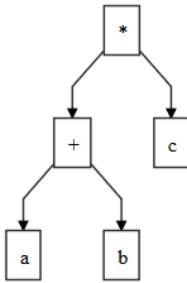
**Ideea algoritmului:**

- Folosind o **stivă** care conține adresele nodurilor din arbore, construim arborele de jos în sus, astfel:
  - Se parcurge expresia în FPP
    - Dacă întâlnim un operand, îl adăugăm în stivă
    - Dacă întâlnim un operator
      - Scoatem ultimul element din stivă, care va deveni fiul drept
      - Scoatem penultimul element din stivă, care va deveni fiul stâng
      - Creăm un nod având ca informație utilă operatorul curent, iar ca descendenți elementele obținute la pașii i. și ii.
      - Adăugăm nodul creat în stivă
  - (Pointerul la) rădăcina arborelui va fi ultimul (și singurul) element rămas în stivă după parcurgerea expresiei în FPP

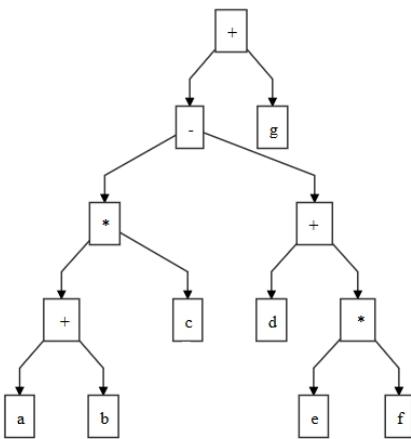


Cum va “evoluă” conținutul stivei odată cu parcurgerea expresiei în FPP și aplicarea algoritmului descris anterior?





.....



Presupunând o reprezentare înlanțuită cu alocare dinamică pentru AB, cum descriem reprezentarea în memorie a tipului de date AB?

#### Nod:

e:TElement  
st, dr: ↑Nod

AB:  
răd: ↑Nod

- ❖ Stiva folosită va avea elemente de tip  $\uparrow$ Nod și vom folosi operațiile stivei:
  - creează
  - adaugă
  - Șterge



Care este descrierea Pseudocod a algoritmului *creează* care, pornind de la expresia *Epost* în FPP creează un AB *arb* asociat expresiei, conform ideii prezentate anterior?

Subalgoritm creează (Epost, arb):

```
creează (s) //creăm o stivă, folosindu-ne de constructorul din
interfața TAD Stivă
pentru fiecare e din Epost execută //parcurgem expresia și
    alocă (nou) //pentru fiecare element, creăm un nou nod
    [nou].e ← e //avându-l ca informație utilă
    dacă e este operand atunci //dacă este operand, înseamnă că
va fi frunză în AB, deci nu are fii
        [nou].st ← NIL
        [nou].dr ← NIL
    altfel //altfel îi extragem fiile din stivă
        șterge(s, p1)
        șterge(s, p2)
        [nou].st ← p2 //stabilim legăturile cu acesteia
        [nou].dr ← p1
    sf_dacă
    adaugă (s, nou) //și îl adăugăm în stivă
sf_pentru
    șterge(s, p) //în final, dacă expresia este corectă, stiva
    arb.răd ← p //va conține doar pointerul la nodul rădăcină
sf_subalgoritm
```

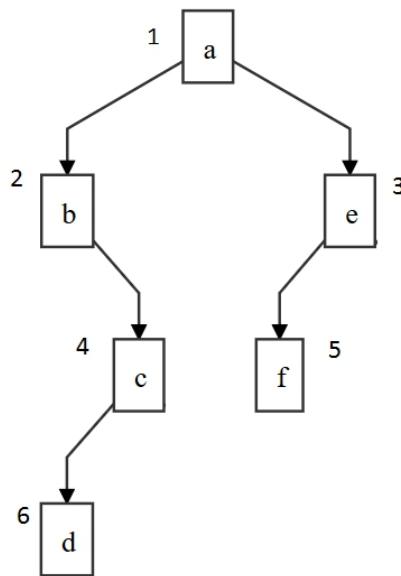
- **Observație:** Dacă dorim să folosim interfața AB în implementare, stiva va conține elemente de tipul AB și se vor folosi operații din interfața AB, algoritmul rescriindu-se astfel:

Subalgoritm creează (Epost, arb):

```
creează (s)
pentru fiecare e din Epost execută
    dacă e este operand atunci
        creeazăFrunză(ab, e)
    altfel
        șterge(s, p1)
        șterge(s, p2)
        creeazăArbore(ab, p2, e, p1)
    sf_dacă
        adaugă(s, ab)
sf_pentru
    șterge(s, arb)
sf_subalgoritm
```

- **Observație:** Specificațiile operațiilor din interfața AB se găsesc în cursul 11.

2. Să se genereze tabelul corespunzător arborelui, numerotându-se nodurile pe nivele, de la stânga la dreapta, precum în exemplul următor.



	1 Informație utilă	2 Index fiu stâng	3 Index fiu drept
1	a	2	3
2	b	0	4
3	e	5	0
4	c	6	0
5	f	0	0
6	d	0	0

- **Observație:** Inexistența unui fiu stâng / drept se marchează cu 0.

- Împărțim soluția în 2 funcții: *numerotare, parcursere*
- Pentru numerotare, presupunem că tipul Nod are un câmp nr: Întreg, în care vom reține numărul asociat nodului. Astfel, reprezentarea devine:

Nod:

e:TElement

st, dr: ↑Nod

nr: Întreg

AB:

răd: ↑Nod



Cum putem parcurge AB pe nivele?

- ✓ Pentru a parcurge AB pe nivele (pe lățime) vom folosi o coadă în care vom reține nodurile



Care este descrierea Pseudocod a algoritmului *numerotare* care, dat fiind un AB *arb*, numerotează nodurile arborelui în ordinea dată de parcurgerea lui pe nivele, completând câmpul *nr* al nodurilor, și returnează numărul *k* al nodurilor din arbore (necesar pentru construirea ulterioră a tabelului)?

Subalgoritm numerotare (*arb, k*):

```
k ← 0 //initializăm numărul nodurilor din arbore
creează(c) //creăm o coadă
dacă arb.răd ≠ NIL atunci //în cazul în care arborele este nevid
    adaugă(c, arb.răd) //începem prin a îi adăuga rădăcina în
    coadă
    k ← 1 //incrementăm k
    [arb.răd].nr ← k //și numerotăm nodul rădăcină
sf_dacă
cât timp (¬ vidă (c)) execută //cât timp mai avem elemente în
coadă
    șterge (c, p) //ștergem elementul cel mai devreme adăugat în
    coadă
    dacă ([p].st ≠ NIL) atunci //verificăm dacă are fiu stâng
        k ← k + 1 //în caz afirmativ,
        [[p].st].nr ← k //îl numerotăm
        adauga(c, [p].st) //și îl adăugăm în coadă
    sf_dacă
    dacă ([p].dr ≠ NIL) atunci //procedăm similar
    k ← k + 1 //pentru fiul drept
        [[p].dr].nr ← k
        adauga(c, [p].dr)
    sf_dacă
sf_cât timp
sf_subalgoritm
```



Care este descrierea Pseudocod a unui algoritm (recursiv) *parcuregere* care, dat fiind un pointer *p* la rădăcina unui arbore având nodurile numerotate și un tablou bidimensional *T* deja alocat corespunzător, reprezentând tabelul de completat, completează *T* conform enunțului?

subalgoritm parcurgere(*p, T*):

```
dacă (p ≠ NIL) atunci //dacă arborele referit de p este nevid
    T[[p].nr, 1] ← [p].e //completăm prima coloană a rândului
    corespunzător nodului rădăcină cu informația utilă din acesta
    dacă ([p].st ≠ NIL) atunci //în cazul în care nodul
    rădăcină are fiu stâng
```

```

T[[p].nr, 2] ← [[p].st].nr //completăm cea de-a două
coloană a rândului corespunzător cu numărul asociat fiului stâng
altfel
    T[[p].nr, 2] ← 0 //altfel, completăm cea de-a două
coloană a rândului corespunzător cu 0 indicând, astfel, inexistența
fiului stâng
    sf_dacă
    dacă ([p].dr ≠ NIL) atunci //în cazul în care nodul rădăcină are
fiu drept
        T[[p].nr, 3] ← [[p].dr].nr //completăm cea de-a treia
coloană a rândului corespunzător cu numărul asociat fiului drept
        altfel
            T[[p].nr, 3] ← 0 //altfel, completăm cea de-a treia
coloană a rândului corespunzător cu 0 indicând, astfel, inexistența
fiului drept
            sf_dacă
            parcurgere([p].st, T)//apelăm, recursiv, subalgoritmul
pentru parcurgerea fiului stâng (OBS: dacă nu există, nu va fi
respectată condiția de continuitate verificată la intrarea în
subalgoritmul recursiv)
            parcurgere([p].dr, T) //apelăm, recursiv, subalgoritmul
pentru parcurgerea fiului drept
            sf_dacă
sf_subalgoritm

```

- **Observație:** În cele ce urmează, descriem în Pseudocod subalgoritmul principal care ulterior numerotării, creează un tabel de dimensiune corespunzătoare și apelează subalgoritmul recursiv de parcursere.

subalgoritm principal(arb, T, k) este:  
numerotare(arb, k)  
@creăm T cu k linii și 3 coloane  
parcurgere(arb.răd, T)  
sf\_subalgoritm



Cum putem adapta soluția în cazul în care nu ne dorim existența unui câmp nr în reprezentarea nodurilor AB?

- ✓ Dacă dorim rezolvarea problemei printr-un singur subalgoritm (de numerotare + parcurgere), putem pune în coadă perechi <nod:↑Nod, nr:Întreg (număr asociat nodului)>.
- ✓ Dacă dorim descompunerea soluției în 2 subalgoritmi, subalgoritmul *numerotare* poate să creeze un Dictionar cu elemente <nod:↑Nod, nr:Întreg> sau <arb:AB, nr:Întreg>, care să fie transmis subalgoritmului *parcurgere*.



Cum putem adapta soluția în cazul în care nu ne dorim ca subalgoritmul *parcurgerea* să fie recursiv?

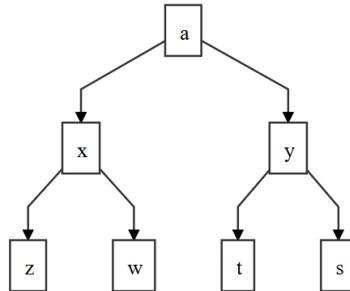
- ✓ Putem folosi o coadă sau o stivă pentru parcurgere, având în vedere că, odată numerotate nodurile conform ordinii de parcurgere în lățime / pe nivele, ordinea de parcurgere pentru completarea tabelului devine irelevantă.

**3. Se dă arborele genealogic al unei persoane, incluzând strămoșii până la generația a  $n$ -a, arborescența stângă reprezentând linia maternă, iar cea dreaptă linia paternă. Presupunem că rădăcina este de gen feminin.**

- a. Să se afișeze toate persoanele de sex feminin (presupunând că rădăcina este de gen feminin)



Care este răspunsul așteptat pentru arborele genealogic exemplificat ulterior?



- ✓ a, x, z, t

**Semnificație:**

- Mama domnișoarei *a* este doamna *x*, iar tatăl ei este domnul *y*
- Bunica domnișoarei *a*, din partea mamei, este doamna *z*, iar bunicul din partea mamei este domnul *w*
- Bunica domnișoarei *a*, din partea tatălui, este doamna *t*, iar bunicul din partea tatălui este domnul *s*

- b. Să se afișeze toți strămoșii de gradul  $k$  (se consideră gradul rădăcinii ca fiind 0)



Care este răspunsul așteptat pentru arborele genealogic exemplificat anterior, în cazul în care  $k=2$ ?

- ✓ **z, w, t, s**



Cum putem rezolva cerința a)?

- ✓ Se parcurge arborele, folosind o coadă (sau o stivă) și se tipăresc, pe lângă rădăcină, doar fii stângi. De pildă, subalgoritmul poate fi descris astfel:

Subalgoritm genFeminin(arb) :  
creeaza(c) //creăm o coadă  
dacă arb.răd ≠ NIL atunci //în cazul în care arborele este nevid  
    adaugă (c, arb.răd) //începem parcurgerea prin a îi  
    adăuga rădăcina în coadă  
        scrie arb.răd//și tipărim informația utilă din rădăcină  
        sf\_dacă  
        cât timp ¬vidă(c) execută //cât timp mai avem noduri de  
        parcurs  
            șterge(c, p) //îl ștergem pe cel introdus cel mai  
            devreme în coadă  
            dacă ([p].st ≠ NIL) atunci //iar dacă are fiu stâng,  
            deci de gen feminin,  
                adaugă(c, [p].st)//îl adăugăm în coadă și  
                scrie [[p].st].e //îi tipărim informația utilă  
                sf\_dacă  
                dacă ([p].dr ≠ NIL) atunci //dacă are fiu drept, deci  
                de gen masculin,  
                    adaugă(c, [p].dr) //doar îl adăugăm în coadă  
                    (deoarece printre strămoșii acestuia putem găsi persoane de gen  
                    feminin)  
                    sf\_dacă  
                sf\_câtimp  
sf\_subalgoritm



Cum putem rezolva cerința b)?

- a. În cele ce urmează, descriem în Pseudocod o varianta recursivă a subalgoritmului  $nivelk$  care returnează într-un vector  $v$  strămoșii de grad  $k$  dintr-un arbore genealogic  $arb$ , folosind doar interfața AB, deci fără a intra în detaliile de reprezentare.

```

Subalgoritm nivelk(arb, k, v) este
    dacă ¬vid(arb) atunci //dacă (sub)arborele curent este nevid,
        dacă k = 0 atunci //și dacă rădăcina lui este pe nivelul k
            în arborele initial, deci, prin decrementare, k a ajuns 0
                adaugă(v, element(arb)) //adăugam elementul / 
                informația utilă din aceasta în vector
            altfel //altfel, înseamnă că nu am ajuns încă la nivelul k,
            deci este nevoie să mai "coborâm" în arbore
                dacă ¬vid(stâng (arb)) atunci //așadar, dacă avem
                subarbore stâng, reaplicăm, recursiv, subalgoritmul pe acesta,
                decrementând nivelul
                    nivel(stâng (arb), k-1, v)
                    sf_dacă
                    dacă ¬vid(drept (arb)) atunci //analog pentru
                    subarborele drept
                        nivel(drept (arb), k-1, v)
                        sf_dacă
                        sf_dacă
                        sf_dacă
                    sf_subalgoritm

```

- **Observație:** Dacă ne dorim tipărirea elementelor din vector (desigur, s-ar fi putut face în algoritmul anterior), descriem, în cele ce urmează, subalgoritmul principal care, ulterior inițializării vectorului rezultat, apelează subalgoritmul recursiv anterior, iar apoi afișează conținutul vectorului rezultat.

```

Subalgoritm strămoși (arb, k, v) este:
    creeaza (v) // inițializăm un vector vid
    nivelk (arb, k, v)
    pentru i ← 1, dim(v) execută //tipărim conținutul
    vectorului rezultat
        scrie element(v, i)
    sf_pentru
sf_subalgoritm

```



Cum putem rezolva cerința b) printr-un algoritm nerecursiv?

- ✓ Folosindu-ne de o coadă (sau stivă) în care să adăugăm perechi <p:↑Nod, nivel:Întreg> sau <arb:AB, nivel:Întreg>.

**4. Să se creeze un arbore pe nivele. Se dă informațiile astfel:**

Rădăcina: 1

Descendenții lui 1: 2, 5

Descendenții lui 2: 0, 3

Descendenții lui 5: 6, 0

Descendenții lui 3: 4, 0

Descendenții lui 6: 0, 0

Descendenții lui 4: 0, 0

Presupunem că marcăm cu 0 inexistența unui nod.



Cum am putea aborda această problemă? Ar fi potrivită folosirea unei stive sau a unei cozi?

- ✓ Folosindu-ne de o **coadă**, descriem în continuare un subalgoritm creeazăNivele, reprezentând o soluție pentru această problemă. Aceasta folosește următoarea funcție auxiliară:

Funcția creeazăNod():

```
@returnează un pointer la un Nod care conține e ca
informație utilă și având cei doi fii NIL
sf_functie
```

Subalgoritm creeazăNivele(arb):

```
scrie "Rădăcina:"
citește e //citim informația din rădăcina,
dacă e ≠ 0 atunci //iar dacă aceasta există, deci arb. e nevid
    arb.răd ← creeazăNod(e)//o înglobăm într-un nod
    creează(c) //și creăm o coadă
    adaugă(c, arb.răd) //în care o adăugăm
    cât timp ¬ vidă(c) execută //cât timp mai avem noduri pentru
    care să interogăm descendenții
        șterge(c, p) //eliminăm nodul cel mai devreme introdus
        în coadă
        scrie "Descendenții lui " [p].e //cerem descendenții
        lui
        citește e1, e2 //îi citim (ca informații)
        dacă e1 ≠ 0 atunci //dacă există fiu stâng
            p1 ← creeazăNod (e1) //îl înglobăm într-un nod
            [p].st ← p1 //și stabilim legătura de la părinte
            la acesta
            adaugă(c, p1) //urmând să îl adăugăm în coadă
sf_dacă
```

```

dacă e2 ≠ 0 atunci //similar pentru descendantul drept
    p2 ← creeazăNod (e2)
    [p].dr ← p2
    adaugă (c, p2)
    sf_dacă
sf_câtimp
altfel
    arb.răd ← NIL //altfel, deci dacă arborele este vid, marcăm
(pointerul la) nodul rădăcină cu NIL
    sf_dacă
sf_subalgoritm

```

# SDA – Seminar 5 - Ansamblu

---

- **Conținut:** Materialul curent exemplifică aplicabilitatea structurii de date **Ansamblu (Heap)** pentru rezolvarea unei probleme, prezentându-se și comparându-se din perspectiva eficienței timp soluției multiple.

## Enunțul problemei

Determinați suma celor mai mari  $k$  elemente dintr-un vector conținând  $n$  elemente numerice distincte.

## Exemplu

De exemplu, dacă vectorul conține următoarele 10 elemente: [6, 12, 91, 9, 3, 5, 25, 81, 11, 23] și  $k = 3$ , rezultatul așteptat este:  $91 + 81 + 25 = 197$ .

## Soluții posibile

### I. Determinând maximul de $k$ ori

#### Observații:

- Dacă, pentru exemplul considerat, aplicăm funcția de determinare a maximului de 3 ori, aceasta va returna 91 de fiecare dată. Așadar, este necesară impunerea unei limite superioare impuse maximului calculat, căutându-se astfel elementul maxim care este mai mic decât limita superioară impusă.
- Generalizând, începând cu cel de-al doilea pas, limita superioară impusă este maximul determinat la pasul anterior. La primul pas se poate alege ca limită o valoare mai mare decât marginea superioară a domeniului valorilor posibile.
- Pentru exemplul considerat:
  - Inițial, maximul este 91.
  - Prin cel de-al doilea apel al funcției de determinare a maximului, dorim determinarea maximului dintre elementele strict mai mici decât 91. Vom obține 81.
  - La cel de-al treilea apel, dorim maximul strict mai mic decât 81, obținând 25.



Care este complexitatea timp a acestei soluții?

- Complexitatea acestei soluții este:  $\Theta(k*n)$  – găsirea maximului are loc în  $\Theta(n)$  și se repetă de  $k$  ori.

## II. Sortarea șirului în ordine descrescătoare și însumarea primelor $k$ elemente.



Care este complexitatea timp a acestei soluții?

- Sortarea poate fi efectuată în  $\Theta(n*\log_2 n)$  (folosind sortarea prin interclasare sau *HeapSort*) (OBS: dacă domeniul valorilor șirului ar respecta condițiile necesare aplicării *BucketSort*, s-ar putea efectua în timp liniar, însă neavând această asigurare, nu o vom presupune)
- Calcului sumei primelor  $k$  elemente din șirul ordonat descrescător în prealabil se efectuează în  $\Theta(k)$
- Obținem, astfel, complexitatea totală a soluției ca fiind  $\Theta(n*\log_2 n) + \Theta(k) \in \Theta(n*\log_2 n)$

## III. Prin adăugarea tuturor celor $n$ elemente într-un ansamblu binar maximal (*binary max-heap*), urmată de eliminarea și însumarea primelor $k$ (celor mai mari) elemente.



Care este complexitatea adăugării unui element într-un ansamblu cu  $m$  elemente?

- Complexitatea adăugării unui element într-un ansamblu cu  $m$  elemente este  **$O(\log_2 m)$** .



Care este complexitatea ștergerii unui element dintr-un ansamblu cu  $m$  elemente?

- Complexitatea eliminării unui element dintr-un ansamblu cu  $m$  elemente este  **$O(\log_2 m)$** .



Care este numărul maxim de elemente conținut de ansamblul în care adăugăm / din care ștergem elemente?

- Ansamblul în care adăugăm și din care ștergem elemente conține cel mult n elemente (anterior primei ștergeri conține exact n elemente).



Câte adăugări se efectuează? Dar ștergeri?

- Se efectuează n adăugări (fiecare dintre cele n elemente fiind adăugat în ansamblu) și k ștergeri (sau, eventual, k-1 ștergeri + accesarea maximului dintre elementele rămase în ansamblu și adăugarea lui sumei).



Care este complexitatea totală obținută?

- Obținem complexitatea totală:  $O(n \cdot \log_2 n) + O(k \cdot \log_2 n)$ . Având în vedere că  $n \geq k$ , se obține  $O(n \cdot \log_2 n)$ .

**Functie** sumaCelorMaiMariK(sir, n, k):

```
//sir este un vector (tablou unidimensional) de numere
întregi distințe
//n este dimensiunea vectorului sir (adică numărul de
elemente pe care acesta le conține)
//k este numărul de elemente pe care dorim să le însumăm,
verificându-se inegalitatea k <= n
```

```
init(a, "≥") //presupunând că avem o structură de date Ansamblu
implementată, a fiind o instanță a ei, inițializăm relația impusă
ansamblului cu "≥", obținând un ansamblu maximal (max-
heap)
Pentru i ← 1, n execută //fiecare dintre cele n elemente din
sir
    adaugă(a, sir[i]) //este adăugat în ansamblu
SfPentru
suma ← 0 //inițializăm suma
Pentru i ← 1, k execută //de k ori
```

```

        elem ← şterge(a) //eliminăm maximul curent din
ansamblu
        suma ← suma + elem //și îl adăugăm sumei
SfPentru
        sumaCelorMaiMariK ← suma //stabilim rezultatul algoritmului
(functiei)
SfFunctie

```



Putem reduce complexitatea soluției anterioare,  $O(n * \log_2 n)$ , printr-o soluție mai eficientă ?

IV. Observăm nu avem nevoie de toate cele  $n$  elemente în ansamblu, întrucât ne interesează doar cele mai mari.

- Dacă reconsiderăm exemplul (șirul este [6, 12, 91, 9, 3, 5, 25, 81, 11, 23] și  $k = 3$ ), **putem păstra în ansamblu, la fiecare pas, doar cele mai mari  $k$  elemente** întâlnite până la momentul curent, procedând după cum urmează:
- Inițial adăugăm primele  $k$  elemente în ansamblu. Deci, pentru exemplul considerat, în momentul în care ajungem cu parcurgerea șirului la elementul 9, ansamblul conține 6, 12 și 91.
- Când întâlnim 9, putem renunța la 6, știind sigur că nu va face parte din cele mai mari 3 numere, întrucât avem deja 3 numere mai mari decât acesta. Așadar, păstrăm 12, 91 și 9.
- Când întâlnim 3, știm că 3 nu va fi printre cele mai mari 3 numere, întrucât avem deja 3 elemente mai mari decât 3 în ansamblu. Conținutul ansamblului rămâne, în continuare: 12, 91 și 9. Similar pentru elementul 5.
- Când întâlnim 25, putem renunța la 9, continuând cu 12, 91 și 25.



Ce se întâmplă la următorii 3 pași și care vor fi cele 3 elemente conținute în final de ansamblu?

- Când întâlnim 81, putem renunța la 12, continuând cu 91, 25 și 81.
- Când întâlnim 11, continuăm cu 91, 25 și 81.
- Când întâlnim 23, păstrăm în ansamblu 91, 25 și 81.
- Astfel, ansamblul conține la finalul parcurgerii șirului elementele 91, 25 și 81, care sunt cele mai mari 3 elemente din întreg șirul.



La ce se reduce rezolvarea problemei ulterior parcurgerii întregului sir și actualizării corespunzătoare a conținutul ansamblului?

- Ulterior parcurgerii sirului și construirii ansamblului format din cele mai mari k elemente, rezolvarea problemei se reduce la însumarea elementelor componente ale ansamblului.



Ansamblul în care păstrăm cele mai mari k elemente din sir ar trebui să fie un ansamblu maximal (*max-heap*) sau unul minimal (*min-heap*)?

- Având în ansamblu, la un moment dat, cele mai mari k elemente anterioare elementului curent, suntem interesați de elementul minim dintre acestea, astfel încât să îl comparăm cu elementul curent. Dacă elementul curent este mai mare decât minimul elementelor din ansamblu, atunci îl va înlocui. Altfel, conținutul ansamblului rămâne nemodificat (știind că toate elementele din ansamblu sunt mai mari decât elementul curent). Prin urmare, avem nevoie de un **ansamblu minimal (*min-heap*)**.

**Functie** sumaCelorMaiMariK\_2(sir, n, k):

```
//sir este un vector (tablou unidimensional) de numere întregi  
distincte  
//n este dimensiunea vectorului sir (adică numărul de elemente  
pe care acesta le conține)  
//k este numărul de elemente pe care dorim să le însumăm,  
verificându-se inegalitatea k <= n
```

```
init(a, " $\leq$ ") //presupunând că avem o structură de date Ansamblu  
implementată, a fiind o instanță a ei, inițializăm relația impusă ansamblului  
cu " $\leq$ ", obținând un ansamblu minimal (min-heap)
```

```
Pentru i ← 1, k execută //primele k elemente ale sirului sunt  
adăugate în ansamblu în mod implicit  
    adaugă(a, sir[i])
```

**SfPentru**

```
Pentru i ← k+1, n execută //fiecare element rămas în sir va fi  
comparat cu elementul minim din ansamblu
```

```
    Dacă sir[i] > prim(a) atunci //dacă este mai mare decât  
elementul minim din ansamblu, returnat de funcția prim, atunci  
        șterge(a) //, elementul minim din ansamblu este eliminat  
        adaugă(a, sir[i]) //, iar elementul curent este adăugat  
    în ansamblu
```

### **SfDacă**

```
//calculăm suma celor k elemente conținute de ansamblu la finalul
parcurgerii sirului
```

### **SfPentru**

```
Pentru i ← 1, k execută //de k ori
    elem ← șterge(a) //eliminăm minimul curent din
    ansamblu
    suma ← suma + elem //și îl adăugăm sumei
```

### **SfPentru**

```
sumaCelorMaiMariK_2 ← suma
```

### **SfSFunctie**



Care este complexitatea timp a acestei soluții?



Care este numărul maxim de elemente conținut de ansamblul în care adăugăm / din care ștergem elemente?

- Ansamblul în care adăugăm și din care ștergem elemente conține cel mult **k** elemente.



Câte sunt complexitățile operațiilor de adăugare și ștergere ?

- Adăugările și ștergerile se efectuează în **O(log<sub>2</sub>k)**.



Câte adăugări și câte ștergeri se efectuează?

- Adăugăm și ștergem de cel mult **n** ori (de pildă, dacă sirul este ordonat crescător).
  - Toate cele n elemente ale sirului vor fi adăugate în ansamblu
  - Primele n-k elemente vor fi eliminate din ansamblu până la construirea ansamblului final, odată cu parcurgerea sirului, restul de k elemente fiind eliminate pentru calculul sumei



Care este complexitatea totală obținută?

- Obținem complexitatea totală:  **$O(n * \log_2 k)$** .

### Observații:

Dacă am lucra direct pe reprezentare, în ultima structură repetitivă *Pentru* în care eliminăm elemente din ansamblu și le însumăm, am putea doar să însumăm elementele din ansamblu. În acest caz, clasa de complexitate nu se schimbă, dar numărul efectiv de operații elementare efectuate se reduce.

- V. Ne amintim (din Cursul 7) că putem să transformăm sirul **într-un ansamblu** în  $O(n)$ . Prin urmare, putem proceda similar soluției III, dar în loc să adăugăm elementele unul câte unul în ansamblu, putem să transformăm sirul **într-un ansamblu maximal și apoi să eliminăm k elemente din el**.



Care este complexitatea acestei soluții?

- Complexitatea acestei soluții este:  **$O(n + k * \log_2 n)$** .