# 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 distincte și fiecare cheie având o singură valoare asociată.

Ce este un **Multi**dicționar? Prin ce diferă el de un Dicționar?

- Un **Multi**dicționar 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 **Multi**dicționar oarecare / neordonat?

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

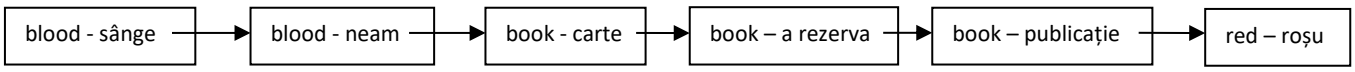
**Problemă:** Să se implementeze a) TAD MDO (Multidictionar Ordonat) reprezentat pe listă simplu înlănțuită (LSI) cu alocare dinamică și b) iteratorul aferent.

**Reprezentare MDO**

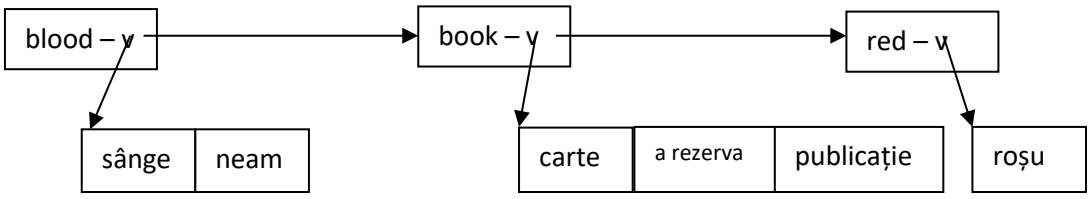
Să considerăm ca exemplu un multidicționar conținând traducerile unor cuvinte din engleză în română:
- book – carte, a rezerva, publicație
- red – roșu
- blood – sânge, neam

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

**Reprezentare 1**: Listă înlănțuită 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, așadar în reprezentarea MDO-ului?

**TElement:**
c: TCheie
l: TListă

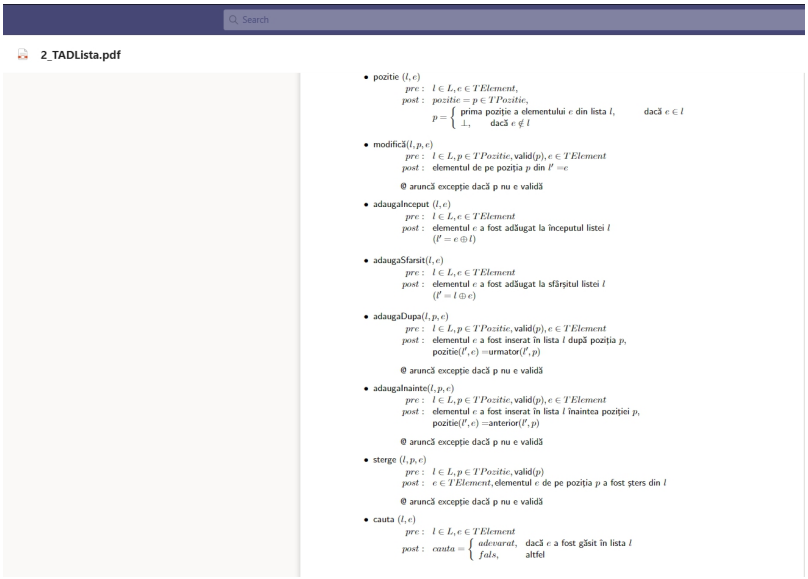
**NodT:**
e: TElement
urm: ↑NodT

**MDO:**
prim: ↑NodT
R: Relație

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

**Explicații:**

- **TElement** este tipul informației utile din nodurile listei simplu inlănțuite (LSI). Aceasta se compune din cheie (unică în LSI, de tipul TCheie) si o listă (TListă) de valori. **TListă** este un TAD conținând elemente de tipul valorilor din MDO, adică TValoare. Pentru a crea, accesa și manipula listele 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 (↑ 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
curent: ↑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 curent 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 curent (acesta ne va permite să accesăm valoare curentă asociată cheii curente). Fiind un TAD, lista valorilor ne va oferi un iterator ca mecanism de parcurgere, noi nefiind preocupați 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> si 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; ni-l returnează în i; i
       refera prima pereche <cheie, valoare> din MDO-ul mdo
       câttimp 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
       returneaza in <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_câttimp
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 iterat
       it.curent ← mdo.prim //cursorul iteratorului (de tip pointer) va referi primul nod al LSI
       dacă it.curent ≠ 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 ↑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.curent].e.c //cheia curentă este cea conținută în informația utilă a nodului curent
       element(it.itL, v) //, iar valoarea curenta este cea pe care iteratorul pe lista de valori o
refera
       e ← <c,v> //perechea curenta returnata fiind compusa din cheia curenta si valoarea curenta
sf_subalg

Complexitate: θ(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.curent ≠ NIL atunci //verificăm dacă cursorul iteratorului refera un nod valid din LSI, adică dacă nu am parcurs deja, în întregime, LSI.
		valid ← adevărat
	altfel
		valid ← fals
sf_funcție

Complexitate: θ(1) (se efectueaza un numar constant de operatii elementare)
```

---

```
//pre: it:I, valid(it)
//post: it':I, it' refera urmatoarea pereche <cheie, valoare> din MDO-ul iterat față de cea referită de it
Subalg următor(it):

	urmator(it.itL) //mutam iteratorul pe lista de valori asociate cheii curente a.i. sa refere urmatoarea valoare
	dacă ¬ valid(it.itL) atunci  //, iar daca acesta devine nevalid, adica nu mai sunt valori neiterate asociate cheii curente
		it.curent ← [it.curent].urm //atunci ne deplasam la urmatoarea cheie, prin a face cursorul iteratorului sa refere urmatorul nod din LSI, dat fiind ca urmatoarea cheie se afla in următorul nod
		dacă it.curent ≠ NIL atunci // daca urmatoarul nod intr-adevar exista, adica nu am epuizat deja toate cheile (ceea ce ar insemna ca am finalizat iterarea), creăm un iterator pe lista valorilor aociate noii chei curente
			iterator ([it.curent].e.l, it.itL)
		sf_dacă
	sf_dacă
sf_subalg

Complexitate: θ(1) (se efectueaza un numar constant de operatii 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 in raport cu o relatie 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 //inițializăm 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: θ(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 distruge(mdo):
	câttimp mdo.prim ≠ NIL execută //cât timp mai există noduri nedealocate 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
		distruge([p].e.l) //distrugem lista de valori asociate cheii din primul nod al LSI,
folosind destructorul TAD-ului TListă
		dealocă(p) //dealocăm primul nod al LSI, pointerul la acesta reținându-l în p, în
prealabil
	sf_câttimp
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>)

θ(mdo) - dacă distrugerea unei liste de valori l se face in θ(lungimea listei) (de exemplu, dacă acestea sunt reprezentate folosind LSI alocate dinamic). In acest caz, mdo = numarul de perechi din MDO = suma lungimilor listelor de valori.

sau

θ(n) – dacă listele de valori pot fi dealocate in θ(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 privata, adică nu este expusa in interfata 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 si mdo.R([aux].e.c, c) si not gasit executa
		//parcurgem 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 curenta este cheia cautata
			gasit ← adevarat //in caz afirmativ, marcăm găsirea ei, prin actualizarea valorii variabilei găsit
		Altfel //alfel, continuăm parcurgerea 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 gasit cheia, initializam corespunzator datele de iesire
		nod ← aux //nodul continand cheia fiind aux
		prec ← precedent //, iar cel precedent lui precendet
	Altfel //altfel
		nod ← NIL //nu exista nod continand cheia cautata
		prec ← precedent //, iar un nod continand cheia cautata ar trebui adaugat dupa precedent
	SfarsitDaca
SfarsitSubalgoritm
```

Complexitate: O(n) (in cel mai defavorabil caz, se parcurge intreaga LSI, aceasta avand 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 iesire: lista: TLista
//pre: mdo: MDO, c:TCheie
//post: lista:TLista, lista fiind lista de valori asociate cheii c in multidictionarul ordonat mdo (daca cheia c nu exista in 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 exista in mdo
		creeaza(lista) //se initializeaza lista (ca lista vida/goala), folosind constructorul
	altfel
	lista ← [nod].e.l //altfel, lista valorilor asociate cheii o gasim in nod, ca parte din informatia utila
	SfarsitDaca
SfarsitSubalgorithm
```

**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
SfarsitSubalgorithm
```

**Complexitate**:

*cautaNod* - O(n)
*adaugaCheieNoua* - Θ(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 (Θ(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 Θ(1) => O(n) ca si complexitate pentru *adauga*
Daca *adaugaFinal*(sau o operatie de adaugare alternativa) are complexitatea Θ(lungimea listei) => 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 multidictionarului ordonat. 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**: 0 (1) // *adaugaFinal* are complexitatea 0(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 opreații pe care o descriem în cele ce urmează
            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 chei 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
            distruge([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)
            dealoca(sters)
      SfarsitDaca
SfarsitSubalgorithm
```

**Complexitate**: 0(1) *(distruge* va distruge o lista vida)