

Lista ordonată (sortată)

SORTED LIST

- Se poate impune o *ordine* între elementele unei liste sub forma unei relații de ordine între elementele acesteia.
 - elementele din listă sunt de **TComparabil**
- Din perspectiva unei ierarhii de containere
 - **Lista ordonată** este o un **Container ordonat/sortat**
 - **Lista ordonată** este o **Listă**
- Interfața tipului abstract de date **Lista ordonată**
 - modifică interfața **TAD Lista** astfel:
 - * constructorul va primi ca parametru relația de ordine între elemente
 - $\text{creeaza}(lo, \mathcal{R})$
{creează o listă ordonată vidă}
 - $pre : \mathcal{R}$ e o relație de ordine definită pe $TElement \times TElement$
 - $post : lo$ e lista vidă, relația de ordine devine \mathcal{R}
 - * operațiile de adăugare din interfața **TAD Lista** ($adaugaSfarsit$, $adaugaInceput$, $adaugaInainte$, $adaugaDupa$) se înlocuiesc cu o singură operație de **adăugare** (numită și *inserare*)
 - adaugă un element în listă astfel încât să se păstreze relația de ordine dintre elementele listei.
 - $\text{adauga}(lo, e)$
{adaugă un element în listă ordonată a.î să se păstreze relația de ordine între elemente}
 - $pre : lo$ e o listă ordonată , $e \in TElement$
 - $post : e$ e inserat în lo , lo' rămâne ordonată
 - * operația *modifică* (setarea unui element pe o anumită poziție în listă) este eliminată
 - prin modificarea unui element de pe o anumită poziție nu se poate asigura faptul că lista va rămâne ordonată.
 - pe lângă operațiile din interfața minimală a Listei, putem adăuga și alte operații (moștenite de la containerul **Colecție**), spre exemplu:
 - $\text{sterge}(l, e)$
 - $pre : l \in L, e \in TElement$
 - $post : \text{prima apariție a elementului } e \text{ a fost ștersă din } l$

- ca și la TAD-ul **Lista**, tipul $TPozitie$ expus în interfață poate fi particularizat (i.e. un indice sau un iterator), rezultând astfel
 - **Lista ordonată cu poziție indice (indexată)** - *poziția* este văzută ca un indice $\Rightarrow TPozitie = Intreg$.
 - **Lista ordonată cu poziție iterator** - *poziția* este dată de un *iterator* pe listă $\Rightarrow TPozitie = Iterator$.

Modalități de implementare a unei liste ordonate

Sunt aceleași modalități de implementare ca pentru o listă neordonată

- memorând elementele sale **secvențial** într-un vector (dinamic)
 - vectorul se va memora ordonat/sortat în raport cu relația de ordine
 - * 9 5 3 2 dacă relația de ordine $\mathcal{R} = \geq$
 - * 2 3 5 9 dacă relația de ordine $\mathcal{R} = \leq$
 - operația **cauta** (căutarea unui element) se realizează în $O(\log_2 n)$ (folosind căutare binară)
 - operația **adauga** va avea complexitatea timp $O(n)$
- memorând elementele sale **înlănțuit** într-o listă înlănțuită
 - lista înlănțuită va fi ordonată - memorează elementele în ordine în raport cu relația de ordine (a se consulta **Cursul 4**).
 - lista înlănțuită poate fi
 - * simplu înlănțuită (LSI)
 - * dublu înlănțuită (LDI)

Exemplu

Considerăm reprezentarea Listei ordonate cu poziție iterator, folosind o LDI alocată dinamic.

- elementele sunt de tip **TComparabil** (**TElement**=**TComparabil**)
- ordinea între elementele listei o vom memora sub forma unei relații de ordine $\mathcal{R} \subseteq \mathbf{TComparabil} \times \mathbf{TComparabil}$, al cărei tip îl vom nota **Relație**. Reamintim faptul că relația va fi implementată în C++ sub forma unui pointer spre o funcție (a se consulta **Cursul 4**, pentru detalii).

Reprezentarea listei și a iteratorului pe listă sunt date mai jos

Nod

e: TElement //informația utilă nodului

urm: \uparrow Nod //adresa la care e memorat următorul nod

prec: \uparrow Nod //adresa la care e memorat nodul anterior

Lista

prim: \uparrow Nod//adresa primului nod din listă

ultim: \uparrow Nod//adresa ultimului nod din listă

\mathcal{R} : Relație//relația de ordine între elemente

IteratorLista

l: Lista//referință către listă

curent: \uparrow Nod//adresa nodului curent din listă

Descriem mai jos, în Pseudocod, operațiile **creeaza** (constructorul listei ordonate) și **cauta** (localizarea unui element în listă). Reamintim specificația acestei operații

cauta (lo, e)

pre : lo listă ordonată în raport cu o relație de ordine $\mathcal{R}, e \in TElement$

post : returnează un iterator: prima poziție pe care apare elementul sau iterator invalid

Subalgoritm **creeaza**(lo, rel)

{*pre*: rel : Relatie}

{*post*: lo e ListaOrdonata, lo e lista vidă, ordinea între elementele listei e rel }

{lista e vidă}

$lo.prim \leftarrow NIL$

$lo.ultim \leftarrow NIL$

{setăm relația}

$lo.\mathcal{R} \leftarrow rel$

SfSubalgoritm

- Complexitate: $\theta(1)$

La căutarea primei poziții pe care apare un element în lista ordonată, trebuie să ținem cont de faptul că lista e ordonată. De exemplu, dacă lista e 4 7 9 11 ($\mathcal{R} = \leq$)

1. dacă vrem să căutăm **3** ($3 < 4$) sau **12** ($12 > 11$), suntem siguri de eșuarea căutării.
2. dacă vrem să căutăm elementul **e=6**, în momentul în care ajungem la **7** nu are rost să continuăm căutarea - am găsit un element mai mare decât **e** (în cazul general, am găsit un element care nu e în relația \mathcal{R} cu **e**).

Funcția **cauta**(lo, e)

{*pre*: lo : ListaOrdonata}

{*post*: returnează un iterator: prima poziție pe care apare elementul sau iterator invalid}

{se creează un iterator invalid pe lista lo }

{apelăm constructorul iteratorului}

creeaza(it, lo)

```

{iteratorul e invalid}
it.curent ← NIL
{dacă e sigur că  $e$  nu apare în listă (caz 1 de mai sus)}
Daca ( $lo.prim = NIL$ )  $\vee (\neg ([lo.prim].e \mathcal{R} e)) \vee (\neg (e \mathcal{R} [lo.ultim].e))$ 
atunci
    {returnăm iterator invalid)}
    cauta ← it
altfel
    {căutăm până găsim elementul, sau e sigur că nu apare în listă (cazul 2
    de mai sus)}
     $p \leftarrow lo.prim$ 
    CatTimp ( $p \neq NIL$ )  $\wedge ([p].e \neq e) \wedge (\neg (e \mathcal{R} [p].e))$  executa
         $p \leftarrow [p].urm$ 
    SfCatTimp
    {dacă am găsit elementul, setăm iteratorul pe nodul găsit)}
    Daca ( $p \neq NIL$ )  $\wedge ([p].e = e)$  atunci
         $it.curent \leftarrow p$ 
    SfDaca
    {returnăm iteratorul)}
    cauta ← it
SfDaca
SfFunctia

```

- Complexitate: $O(n)$, n fiind numărul de elemente din listă

În directorul TAD Lista Ordonata (pe pagina cursului, **Curs 5**) găsiți implementarea parțială în limbajul C++ a containerului **Lista ordonată** cu poziție iterator (reprezentarea este sub forma unei LDI, folosind alocare dinamică pentru reprezentarea înlănțuirilor). **Atenție:** operația de adăugare nu e completă (e tratat doar cazul în care elementul trebuie adăugat la începutul listei).