

# Notiuni introductive

- informații → date → → **structuri de date** (SD).
- sistemele de calcul ocupă mult timp cu
  - stocarea datelor (S)
  - accesarea datelor (A)
  - manipularea datelor (M)
- în toate domeniile (sisteme de operare, baze de date, compilatoare, grafică, inteligență artificială, etc) se pune problema (S)+(A)+(M)
- reprezentarea obiectelor din lumea reală în aplicații software necesită
  1. obiecte din lumea reală  $\xrightarrow{\text{MODELARE}}$  entități matematice
  2. mulțimea de operații definite pe entitățile de la 1.
  3. maniera în care entitățile de la 1. sunt **STOCATE** și **ARANJATE** în memoria calculatorului
  4. algoritmii utilizați pentru a efectua operațiile de la 2.  
(1)+(2) → **TAD**  
(3)+(4) → **SD**

## Tipuri Abstracte de Date

- Un Tip Abstract de Date (TAD) este un tip de date cu următoarele proprietăți:
  1. *specificarea obiectelor* din domeniu este independentă de reprezentarea lor;
  2. *specificarea operațiilor* este independentă de implementarea lor.
- Mulțimea operațiilor Tipului Abstract de Date definesc interfața (contractul) său.
- un TAD este o entitate specificată matematic, care definește o mulțime – a instanțelor sale, având o *interfață* specifică – o colecție de *signaturi* ale operațiilor care pot fi aplicate pe o instanță a tipului de date, pentru fiecare operație asigurându-se o specificație clară.

- Domeniul unui tip (abstract) de date poate fi definit fie prin enumerarea elementelor sale, în cazul în care este finit, fie printr-o regulă care descrie elementele sale.
- După definirea domeniului unui tip (abstract) de date, este necesară *specificarea operațiilor* (date, rezultate, precondiții, postcondiții).
- “*De ce abstractizare?*”
  - Deoarece se impune specificarea operațiilor structurilor de date și amânarea detaliilor de implementare.
  - Există multe Tipuri Abstracte de Date, ca urmare decizia de a alege TAD-ul potrivit este un pas foarte important în proiectare.
  - Limbajele de nivel înalt furnizează deseori implementări pentru TAD-uri predefinite (biblioteca STL din C++, bibliotecile standard din Python, Java).
  - TAD de tip *container*: *vector* (**array**), *colecție* (**collection**), *multime* (**set**), *listă* (**list**), *dicționar* (**dictionary**, **map**), etc.
    - \* “*De ce să încercăm să creem propriile noastre implementări pentru diverse TAD dacă unele limbaje ne oferă implementari ale TAD pe care ni-l dorim?*”
      1. pentru a învăța modul de utilizare și creare de implementări de Tipuri Abstracte de Date.
      2. pentru a învăța comportamentul unora dintre cele mai utilizate Tipuri Abstracte de Date.

## Interfața unui TAD

Tipurile de operații din interfața unui TAD sunt următoarele:

- - operații care să permită crearea elementelor de acel tip (*constructori*);
- - operație care să permită distrugerea elementelor de acel tip (*destructor*);
- - operații de accesare a componentelor instanțelor;
- - operații specifice de manipulare a instanțelor;
- - operații de verificare ale unor proprietăți ale instanțelor;
- - operații specifice tipului de date.

## Folosirea abstractizării și încapsulării datelor în proiectarea programelor

1. **Încapsularea datelor** (ascunderea informației) este definită ca fiind ascunderea detaliilor de implementare ale unui obiect față de lumea exterioară (aplicațiile care îl folosesc).

2. **Abstractizarea** datelor este definită ca fiind separarea dintre *specificarea* unui obiect și *implementarea* lui.

În procesul de proiectare a programelor, folosirea abstractizării și încapsulării datelor sunt deosebit de importante, din următoarele motive:

- **Simplificarea procesului de dezvoltare** a programelor - dacă în faza de proiectare a programului am identificat că avem nevoie de tipurile de date A, B, C, ... cu acestea se poate lucra independent, deoarece nu interesează decât specificațiile lor.
- **Testarea și depanarea** se poate face mai ușor, deoarece fiecare dintre tipurile A, B, C, ... pot fi testate și verificate separat.
- **Reutilizarea** - Extragerea unei structuri de date dintr-o aplicație și folosirea acesteia în altă aplicație va fi deosebit de facilă în cazul în care pentru tipul respectiv de date am folosit încapsularea datelor.
- **Schimbarea reprezentării** unui tip de date - se poate schimba reprezentarea unui tip de date fără a fi afectate programele care folosesc acel tip de date cu condiția ca operațiile tipului să nu fie modificate (interfața să rămână aceeași).

Una din cerințele esențiale ale programării prin abstractizarea și încapsularea datelor este să nu se expună în exterior (într-o aplicație) reprezentarea unui Tip Abstract de Date.

# Structuri de Date

- Domeniul *Structurilor de Date* (SD) se ocupă cu stocarea și accesarea datelor.
- În rezolvarea de probleme se manipulează informații. În organizarea datelor, conform algoritmului de rezolvare, informațiile de diferite tipuri se grupează în structuri, numite *structuri de date*.
- O SD putem să o considerăm din două puncte de vedere:
  1. **Logic**, ca și definiție (elementele ei constitutive și legăturile dintre ele).
  2. **Fizic**, ca mod de memorare (stocare).
    - (a) Structuri de date **static**e. O structură **statică** ocupă în memoria calculatorului o zonă de dimensiune constantă, în care elementele ocupă tot timpul execuției același loc. Exemple de structuri de date statice sunt: articolele, tablourile.
    - (b) Structuri de date **semistatic**e. O structură **semistatică** ocupă în memoria calculatorului o zonă de dimensiune constantă, dar elementele ocupă loc variabil în timpul execuției programului. Exemple de structuri de date semistatice sunt tabele de dispersie.
    - (c) Structuri de date **dinamic**e. O structură **dinamică** ocupă în memoria calculatorului o zonă care se aloca dinamic, în timpul execuției programului, pe măsura nevoilor de prelucrare, fără a avea o dimensiune constantă. Exemple de structuri de date dinamice sunt: liste înlănuite, arbori.
- O structură de date logică ar putea fi implementată atât ca structură statică, semistatică sau dinamică. De exemplu, lista înlănuită, care din punct de vedere conceptual este o structură dinamică (se modifică în timp), ar putea fi implementată ca o structură semistatică.
- Pentru un TAD vom discuta mai multe structuri de date folosite pentru implementarea acestuia, fiecare aducând beneficii specifice, dar și eventuale dezavantaje
  - În unele cazuri nu se poate spune foarte clar care ar fi cea mai “bună” structură de date pentru implementarea unui TAD.
  - Alegerea structurii de date corespunzătoare rezolvării unei probleme este extrem de importantă.
    - \* alegerea unor tehnici de structuri de date corespunzătoare poate avea ca efect reduceri masive de timp - analiza complexității operațiilor este importantă.
  - Cum putem proiecta SD noi?

TAD/Container	SD
<b>Vector Dinamic</b> <b>Matrice</b> <b>Colecție</b> <b>Mulțime</b> <b>Dicționare</b> Dicționar, Dicționar ordonat, Multidicționar, Multidicționar ordonat <b>Lista</b> <b>Lista ordonată</b> <b>Coadă</b> <b>Stiva</b> <b>Coadă cu priorități</b> <b>Arbore (binar)</b>	<b>tablou</b> <b>lista înlănțuită</b> - simplu - dublu - alocare dinamică - înlănțuiri pe tablou <b>tabela de dispersie</b> - liste independente - liste întrepătrunse - adresare deschisă <b>ansamblu</b> <b>arbore binar de căutare</b> - arbore echilibrat (AVL)

Urmărim o prezentare generală a structurilor de date, independentă de un limbaj de programare sau altul - pentru descrierea *algoritmilor* vom folosi limbajul *Pseudocod*.

# Algoritmi

- Domeniile SD și al algoritmilor (de manipulare a acestor structuri) se interconectează.
- Aspecte de bază legate de algoritmi - eficiența algoritmilor
  - cantitatea de resurse utilizate
    - \* timp
    - \* spațiu
  - măsurarea eficienței:
    - \* analiză asimptotică (complexitate timp și spațiu)
    - \* analiza empirică

# Limbajul Pseudocod

- Vom folosi două tipuri de propoziții pseudocod:
  1. propoziții standard, fiecare având sintaxa și semantica sa;
  2. propoziții nestandard (texte care descriu părți ale algoritmului încă incomplet elaborate). Aceste propoziții convenim să înceapă cu semnul '@'.
- Comentariile vor fi cuprinse între accolade.
- Citirea datelor se face folosind propoziția standard:  
**citeste** *lista*
- Tipărirea rezultatelor se face folosind propoziția standard:  
**tipărește** *lista*
- Atribuirea se va simboliza prin  $\leftarrow$ .
- Instrucțiunea alternativă va avea forma:

```

Daca expresie logica atunci
    instructiuni
altfel
    instructiuni
SfDaca

```

unde secțiunea **altfel** poate lipsi.

- Structura repetitivă cu număr cunoscut de pași:

```

Pentru contor = li, lf, pas executa
    instructiuni
SfPentru

```

unde contorul ia valori de la valoarea inițială *li*, la valoarea finală *lf*, la fiecare pas adăugându-se valoarea *pas*. Pasul poate lipsi fiind implicit egal cu 1.

- Structura repetitivă cu număr necunoscut de pași condiționată anterior (test inițial):

```

CatTimp expresie_logica executa
    instructiuni
SfCatTimp

```

- Structura repetitivă cu număr necunoscut de pași condiționată posterior (test final):

```

Repete
    instructiuni
PanaCand expresie logica

```

- Definirea unui subalgoritm se va face folosind propoziția standard:

```

Subalgoritm nume(...)
    instructiuni
SfSubalgoritm

```

- Definirea unei funcții se va face folosind propoziția standard:

```

Functia nume(...)
    instructiuni
SfFunctia

```

- Pentru a specifica rezultatul întors de o funcție vom folosi numele funcției.

*Exemplu:*

```

Functia minim(a, b)
    min ← a
    Daca a < b atunci
        min ← b
    SfDaca
    minim ← min
SfFunctia

```

- Apelul unei proceduri se face folosind:

```

nume(< lista_parametri_actuali >)

```

- apelul unei funcții se face scriind într-o expresie numele funcției urmat de lista parametrilor actuali (ex:  $m \leftarrow \minim(2, 3)$ ).

## Extensii și convenții

- Dacă vrem să declarăm o variabilă  $i$  de tip *Intreg*, atunci vom folosi notația  $i : \text{Intreg}$ .
- În cazul în care dorim să declarăm un tablou unidimensional  $t$  cu elemente de tip *TElement* vom folosi notația  $t : TElement[]$ .
  - Dacă se dorește precizarea exactă a limitelor de variație a unui indice, vom folosi notația care se bazează pe tipul subdomeniu:  $TElement[MIN..MAX]$
- O înregistrare (un vector având lungimea  $n$  și elementele de tip *TElement*) o vom reprezenta sub forma

```

Vector
    n:Intreg
    e:TE[]

```

- **Accesul** la elementele unei înregistrări îl vom face folosind caracterul “.”
- \* Dacă ne referim la o variabilă  $v$  de tip *Vector*, atunci prin:
  - $v.n$  - ne vom referi la numărul de elemente ale vectorului;
  - $v.e[i]$  - ne vom referi la al  $i$ -lea element din vector.
- Pentru a indica pointeri (adrese ale unor zone de memorie), vom folosi caracterul  $\uparrow$ , cu alte cuvinte dacă vrem să declarăm un pointer  $p$  care referă un număr întreg, acest lucru îl vom scrie în următoarea manieră:

$p : \uparrow \text{Intreg}$

Conținutul locației referite de pointerul  $p$  îl vom nota  $[p]$ .

- Pointerul nul (care nu referă nimic) îl vom nota NIL.
- Operațiile de alocare, respectiv dealocare a pointerilor le vom nota:
  - $\text{aloca}(p)$
  - $\text{dealoca}(p)$

## Convenții folosite în specificații.

- Specificarea unei operații se va face prin:

- 1 **pre:** - date și precondiții
- 2 **post:** - rezultate și postcondiții

[3 ] @ - (optional) exceptii aruncate

- În specificarea operațiilor prin precondiții și postcondiții, când folosim numele unei variabile ne referim la valoarea acesteia.
- Având o variabilă  $i$  de tip  $Tip$  ( $i : Tip$ ), notația  $i \in Tip$  (exemplu:  $i \in Intreg$ ), va fi folosită pentru a evidenția faptul că valoarea variabilei aparține domeniului de definiție a tipului  $Tip$  ( $Intreg$ )).
- Datorită faptului că valorile variabilelor pot fi modificate în urma executării unei operații, este necesară delimitarea dintre valoarea variabilei înainte de efectuarea operației și cea de după execuția ei. Vom conveni să folosim caracterul ' (apostrof) pentru a specifica valoarea variabilei după aplicarea operației.
  - De exemplu, având o operație **dec** care decrementează valoarea unei variabile  $x$  ( $x : Intreg$ ), specificația operației va fi:

**dec**( $x$ )

*pre* :  $x \in Integer$   
*post* :  $x' = x - 1$

## Tip de date generic

Pentru generalitate, vom considera că elementele unui TAD sunt de un tip de date generic **TElement** cu o interfață minimală formată din următoarele operații:

- atribuire

**atribuie**( $x, y$ ) - notație  $x \leftarrow y$

*pre* :  $x, y \in TElement$   
*post* :  $x = y$

- testarea egalității

**egal**( $x, y$ ) - notație  $x = y$

*pre* :  $x, y \in TElement$   
*post* :  $egal = \begin{cases} adevarat, & x = y \\ fals, & x \neq y \end{cases}$

Pentru simplitate, vom folosi notațiile

- “ $x=y$ ” în locul apelului funcției “ $egal(x,y)$ ” pentru a ilustra egalitatea a două elemente de tip **TElement**.
- “ $x \leftarrow y$ ” în locul apelului subalgoritmului “ $atribuie(x,y)$ ” pentru a ilustra operația de atribuire.

Dacă pe domeniul valorilor unui tip de date se poate defini o relație de ordine ( $\leq$ ), vom defini și tipul generic ***TComparabil***, care derivă din tipul *TElement*; pe lângă interfața acestuia, *TComparabil* admite și următoarea operații:

- compararea a două elemente

**compară**(*x, y*)

*pre* :  $x, y \in TComparabil$

$$\text{post : } \text{compara} = \begin{cases} -1, & \text{dacă } x < y \\ 0, & \text{dacă } x = y \\ 1, & \text{dacă } y > x \end{cases}$$

Pentru simplitate, vom folosi notările “ $x < y$ ”, “ $x \leq y$ ”, “ $x > y$ ”, “ $x \geq y$ ” pentru a ilustra relațiile corespunzătoare între elemente de tip ***TComparabil***.

# Containere și iteratori

- Un *container* este o grupare de date în care se pot adăuga (insera) și din care se pot șterge (extragă) obiecte.
- Un container poate fi definit ca fiind o colecție de date care suportă cel puțin următoarele operații:
  - *adăugarea* unui element în container;
  - *ștergerea* unui element din container;
  - *returnarea numărului de elemente* din container (dimensiunea containerului);
  - *căutarea* unui obiect în container.
  - furnizare *acces* la obiectele stocate (de obicei folosind iteratori) - *căutarea* unui obiect în container.
- TAD
- Ce container de date este potrivit într-o anumită aplicație?

## Iteratori

- Iteratorii pot fi văzuți ca o generalizare a referințelor, și anume ca obiecte care referă alte obiecte. Iteratorii sunt des utilizati pentru a parcurge un container de obiecte.
- Sunt importanți în programarea generică: un container trebuie doar să furnizeze un mecanism de accesare a elementelor sale folosind iteratori.
- Iteratorul va conține (Figura 1)

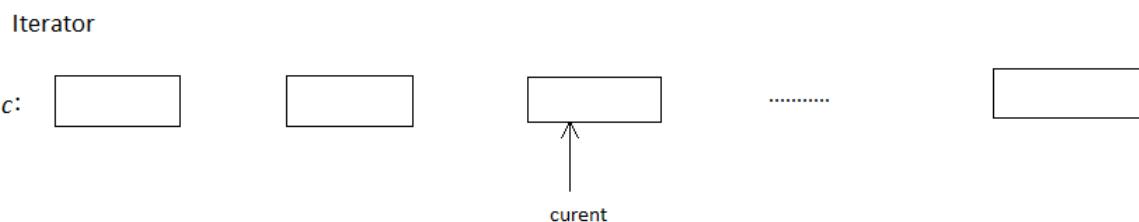


Figura 1: Iterator pe un container *c*.

- o referință spre containerul pe care-l iterează
- o referință spre elementul *current* din iterație, referință numită în general *current* (cursor).

- Iterarea elementelor containerului se va face mutând referința “current” (în funcție de tipul iteratorului) în container atât timp cât referința este validă (adică mai sunt elemente de iterat în container).
- Există mai multe categorii de iteratori, în funcție de maniera de iterare a containerului:
  - iteratori unidirecționali (cu control într-o direcție);
  - iteratori bidirecționali (cu control în două direcții);
  - iteratori cu acces aleator ;
  - read-write* (permite ștergere și inserare de elemente în container)

Vom prezenta specificația TAD Iterator cu o interfață minimală (numărul minimal de operații) pentru un iterator unidirecțional.

### TAD Iterator domeniu

$$\mathcal{I} = \{i \mid i \text{ este un iterator pe un container având elemente de tip } TElement\}$$

**operații** (interfața TAD-ului Iterator)

- creeaza( $i, c$ )

*pre* :  $c$  este un container  
*post* :  $i \in \mathcal{I}$ , s-a creat iteratorul  $i$  pe containerul  $c$

- prim( $i$ )

*pre* :  $i \in \mathcal{I}$   
*post* : *current* referă ‘primul’ element din container

- valid( $i$ )

*pre* :  $i \in \mathcal{I}$   
*post* :  $valid = \begin{cases} adeu, & \text{dacă } current \text{ referă o poziție validă} \\ & \text{din container} \\ fals, & \text{contrar} \end{cases}$

- element( $i, e$ )

*pre* :  $i \in \mathcal{I}$ , *current* este valid (referă un element din container)  
*post* :  $e \in TElement$ ,  $e$  este elementul *current* din iterare  
           (elementul din container referit de *current*)

- următor( $i$ )

*pre* :  $i \in \mathcal{I}$ , *current* este valid  
*post* : *current'* referă ‘următorul’ element din container  
           față de cel referit de *current*

## Observații

- pentru simplitate, operațiile **creeaza** și **prim** se pot combina în operația **creeaza** (constructor) cu specificația de mai jos
- **creeaza(*i, c*)**
  - pre* : *c* este un container
  - post* : *i* ∈  $\mathcal{I}$ , s-a creat iteratorul *i* pe containerul *c* (elementul *current* din iterator referă ‘primul’ element din container)
- vom considera în cele ce urmează varianta simplificată de mai sus

- Orice *container* va avea în interfață să o operație

– **iterator(*c, i*)**

*pre* : *c* container

*post* : *i* ∈  $\mathcal{I}$ , *i* este un iterator pe containerul *c*

- Operația **iterator** din interfața containerului apelează, în general, constructorul iteratorului

**Subalgoritm iterator(*c, i*)**

*pre:* *c* este un container de date

*post:* *i* este un iterator pe containerul *c*

{se creeaza iteratorul *i* pe containerul *c*}

**creeaza(*i, c*)**

**SfSubalgoritm**

- Folosind iteratori putem crește foarte mult gradul de genericitate a algoritmilor care lucrează pe containere.

Tipărirea elementelor din containerul *c* se va face în felul următor:

**Subalgoritm Tiparire(*c*)**

*pre:* *c* este un container de date

*post:* elementele containerului *c* au fost tipărite

**iterator(*c, i*)**

{containerul își obține iteratorul}

**CatTimp valid(*i*) executa**

{cât timp iteratorul e valid}

**element(*i, e*)**

{se obține elementul current din operație}

**tipareste(*e*)**

{se tipărește elementul current}

**următor(*i*)**

{se deplasează iteratorul}

**SfCatTimp**

**SfSubalgoritm**

# Articol și tablou

## Articolul (înregistrarea)

- Structură de date statică.
- Un articol reprezintă reuniunea unui număr fix de componente care pot avea tipuri diferite (caracter *neomogen*), numite *câmpuri*, care constituie logic o unitate de prelucrare.

$$\text{Persoana} = \{\text{nume}, \text{data\_nasterii}, \text{adresa}, \text{ID}\}$$

- Folosite pentru descrierea unor obiecte care constau din mai multe componente.
- Operații specifice: **creare, selecție și modificare** componente.

## Tabloul

- Structură de date statică.
- Notații
  - $[n] = \{1, 2, \dots, n\}, n \in \mathbb{N}^*$
  - colecție  $C$  de elemente având tipul generic  $TElement$ .

Aplicația  $f : [n_1] \times [n_2] \times \dots \times [n_k] \rightarrow C$ , care atașează fiecărui  $k$ -uplu de indici  $(i_1, i_2, \dots, i_k)$  unde  $i_j \in [n_j]$ , un element  $c_{i_1, i_2, \dots, i_k}$  definește un tablou  $k$ -dimensional  $T(n_1, n_2, \dots, n_k)$ .

- În cazul particular  $k = 1$  se obține un tablou unidimensional numit *vector*,  $f : [n] \rightarrow C$ , având elementele  $c_1 = f(1), c_2 = f(2), \dots, c_n = f(n)$ .
  - În memoria internă elementele unui vector vor ocupa în ordine locații succesive de memorie ⇒ **reprezentare secvențială**.
    - \* **Detaliu de implementare** - spațiul de memorie necesar stocării elementelor vectorului se poate aloca dinamic, în timpul execuției programului.
    - Identificarea elementelor se face cu ajutorul *indicilor*.
- Un tablou este static: nu pot fi inserate sau șterse elemente(celule).
- Tablourile sunt foarte mult folosite în programare și pentru reprezentarea altor structuri de date ⇒ pentru tablourile statice limbajele de programare includ notații specifice.
- Memorare a tablourilor
  - secvențial (ordine lexicografică a indicilor)
  - înălțuit (cazul matricilor rare)

# Vector Dinamic

## DYNAMIC ARRAY

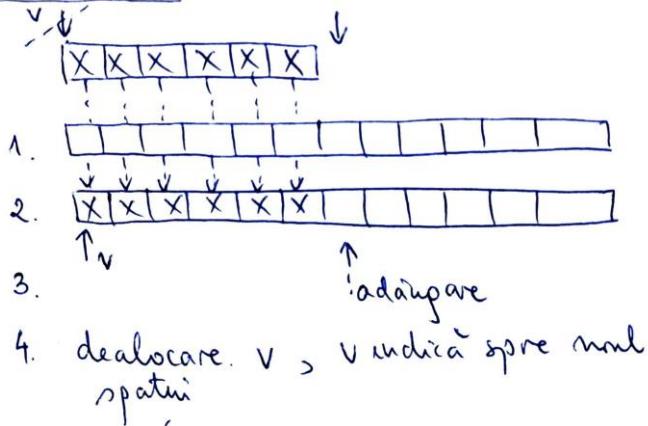
### Observații

1. Un tablou este static: nu pot fi inserate sau șterse celule.
2. Vector - tablou unidimensional
3. Reprezentarea vectorilor este **sevențială**, adică elementele sunt memorate în locații succesive de memorie
  - Detaliu de implementare: spațiul de memorare poate fi alocat *static* sau *dinamic* (în timpul execuției programului).
  - Accesul la elemente este **direct** (prin intermediul indicilor)  $\Rightarrow$  complexitate-timp  $\theta(1)$ .
4. Există așa numitele **tablouri dinamice - vectori dinamici** ("dynamic arrays"), care sunt tablouri unidimensionale cu caracter dinamic, a căror lungime se modifică în timp și în care se pot inseră, respectiv șterge elemente.
  - spațiul de memorare alocat tabloului crește automat în momentul în care nu mai există spațiu suficient pentru inserarea unui nou element  $\Rightarrow$  alocare dinamică
6. Alegerea unei reprezentări bazate pe **alocarea dinamică** a memoriei prezintă avantajul amânării cunoașterii mărimi efective a vectorului până la momentul execuției. Un tablou alocat *static* impune specificarea mărimi tabloului ca o constantă.

### Caracteristici

1. Vectorul dinamic - reprezentare:
  - a. capacitatea sa ( $cp$ ) – numărul de locații alocate vectorului;
  - b. dimensiunea sa ( $n$ ) – lungimea efectivă a vectorului;
  - c. elementele vectorului ( $e = e_1, e_2, \dots, e_n$ ).
    - în C/C++,  $e$  este adresa la care se memorează primul element al vectorului
2. Dacă la adăugarea unui element în vector se depășește capacitatea vectorului, atunci se **mărește** capacitatea acestuia (de obicei se dublează, pentru generalitate se poate considera un anumit raport de creștere a capacitatii față de dimensiunea vectorului).
  - Ca detaliu de implementare, în cazul în care spațiul de memorie necesar stocării elementelor vectorului se alocă dinamic, atunci se va face o **reallocare** a acestui spațiu (la noua capacitate), se copiază elementele din vechiul spațiu în noul spațiu, se adaugă/inserează elementul, după care se dealocă vechiul spațiu.

## Redimensionare



- Cu toate că această operație de redimensionare/reallocare este costisitoare ( $\theta(n)$ ), operația de adăugare a unui element la sfârșitul vectorului are, totuși, *complexitatea-timp amortizată*  $\theta(1)$

### 3. Implementări în bibliotecile existente în diferite limbaje

- Java – clasa **Vector**
  - implementată încât să funcționeze cu acces concurrent
  - dacă nu se dorește acces concurrent  $\Rightarrow$  **ArrayList** (lista reprezentată secvențial pe tablou)
- STL din C++ - clasa **Vector**
  - Considerat un container de tip *secvență* (acces pozitie)
  - implementat ca un *vector dinamic*

### 4. **VectorDinamic** - reprezentare secvențială a containerului **Lista**.

### 5. **VectorDinamic** se consideră potrivit pentru:

- Accesare element de pe o anumită pozitie  $\theta(1)$  (complexitate-timp).
- Iterare elemente în orice ordine – timp liniar  $\theta(n)$
- Adăugare element la sfârșit – complexitate-timp **amortizată**  $\theta(1)$
- Ștergere element de la sfârșit - timp constant  $\theta(1)$

În continuare, pentru un număr natural  $n$  folosim notația  $[n] = \{1, 2, \dots, n\}$ . De asemenea, vom considera următoarele:

- adăugarea și ștergerea elementelor se poate face la atât la sfârșitul vectorului, cât și pe orice pozitie în vector.
- la redimensionarea capacitatei vectorului, pp. că aceasta se dublează.

**Deși este unanim acceptat faptul că reprezentarea tablourilor unidimensionale este secvențială, se poate abstractiza tipul de dată VectorDinamic.**

## TAD VectorDinamic

### domeniu

$$\mathbf{V} = \{ \mathbf{v} \mid v = (cp, n, e_1 e_2 \dots e_n), cp, n \in N, n \leq cp, e_i \text{ sunt de tip } \mathbf{TElement} \}$$

## operații (interfață)

*creează*( $v, cp$ )

{constructor - se creează un vector cu lungime 0, având capacitatea  $cp$ }

**pre:**  $cp \in \mathbf{Natural}$

**post:**  $v \in \mathbf{V}, v.n=0, v.cp=cp$

@ aruncă excepție dacă  $cp$  e negativ

*dim*( $v$ )

**pre:**  $v \in \mathbf{V}$

**post:**  $\mathbf{dim} = \text{lungimea vectorului } v \text{ (numărul de elemente)} \in \mathbf{Natural}$

*element*( $v, i, e$ )

**pre:**  $v \in \mathbf{V}, i \in \mathbf{Natural}, i \in [v.n]$

**post:**  $e \in \mathbf{TElement}, e=v.e_i$  (elementul de pe poziția  $i$  din vectorul  $v$ )

@ aruncă excepție dacă  $i$  e în afara intervalului  $[v.n]$

*modifică*( $v, i, e$ )

**pre:**  $v \in \mathbf{V}, i \in \mathbf{Natural}, i \in [v.n], e \in \mathbf{TElement}$

**post:**  $v' \in \mathbf{V}, v'.e_i=e$  (al  $i$ -lea element din  $v'$  devine  $e$ )

@ aruncă excepție dacă  $i$  e în afara intervalului  $[v.n]$

*adaugaSfarsit*( $v, e$ )

{se adaugă la sfârșitul vectorului elementul  $e$ ; dacă  $v.n=v.cp$  atunci crește capacitatea}

**pre:**  $v \in \mathbf{V}, e \in \mathbf{TElement}$

**post:**  $v' \in \mathbf{V}, v'.n = v.n + 1$

$(v.cp = v.n) \Rightarrow (v'.cp = v.cp * 2, v'.e[v'.n] = e)$

*adaugaPozitie*( $v, i, e$ )

{se adaugă pe poziția  $i$  elementul  $e$ ; dacă  $v.n=v.cp$  atunci crește capacitatea}

**pre:**  $v \in \mathbf{V}, i \in \mathbf{Natural}, i \in [v.n] + 1, e \in \mathbf{TElement}$

**post:**  $v' \in \mathbf{V}, v'.n = v.n + 1$

$(v.cp = v.n) \Rightarrow (v'.cp = v.cp * 2, v'.e[j] = v.e[j-1] \forall j = v'.n, v'.n - 1, \dots, i + 1, v'.e[i] = e)$

@ aruncă excepție dacă  $i$  e în afara intervalului  $[v.n]$

*stergeSfarsit*( $v, e$ )

{se șterge elementul de la sfârșitul vectorului }

**pre:**  $v \in \mathbf{V}, v.n > 0$

**post:**  $e \in \mathbf{TElement}, e = v.e[v.n], v' \in \mathbf{V}, v'.n = v.n - 1$

*stergePoziție(v, i, e)*

{se șterge pe poziția  $i$  a vectorului }

**pre:**  $v \in \mathbf{V}$ ,  $v.n > 0$ ,  $i \in \mathbf{Natural}$ ,  $i \in [v.n]$

**post:**  $e \in \mathbf{TElement}$ ,  $e = v.e[i]$ ,  $v' \in \mathbf{V}$ ,  $v'.n = v.n - 1$ ,  $v'.e[j] = v.e[j + 1] \forall j = i, i + 1, \dots, v'.n$   
@ aruncă excepție dacă  $i$  e în afara intervalului  $[v.n]$

*iterator(v, i)*

{se creează un iterator pe vectorul  $v$ }

**pre:**  $v \in \mathbf{V}$

**post:**  $i \in \mathbf{I}$ ,  $i$  este iterator pe vectorul  $v$

*distruge(v)*

{destructor}

**pre:**  $v \in \mathbf{V}$

**post:** vectorul  $v$  a fost 'distrus' (spațiul de memorie alocat a fost eliberat)

....alte operații....

Elementele unui vector dinamic pot fi tipărite în două moduri:

1. Prin iterator, ca orice container.
2. Folosind accesul la elemente prin indici, datorită reprezentării secvențiale.

Ca urmare, tipărirea se poate face

1. **subalgoritm** *tipărire(v)* este {complexitate-timp  $\theta(n)$  }

{pre:  $v$  este un vector dinamic}

{post: se tipăresc elementele vectorului}

**iterator(v,i)** {vectorul își construiește iteratorul}

**CâtTimp valid(i) execută** {cât timp iteratorul e valid}

**element(i, e)** {se obține elementul curent din iterație}

**@ tipărește e** {se tipărește elementul curent}

**următor(i)** {se deplasează iteratorul}

**SfCâtTimp**

**sfTipărire**

2. **subalgoritm** *tipărire(v)* este {complexitate-timp  $\theta(n)$  }

{pre:  $v$  este un vector dinamic}

{post: se tipăresc elementele vectorului}

**pentru  $i \leftarrow 1, \dim(v)$  execută**

**element(v, i, e)** {se obține elementul de poziția}

**@ tipărește e** {se tipărește elementul curent}

**sPentru**

**sfTipărire**

## Operatia adaugăSfârșit

### Reprezentare

#### VectorDinamic

cp: Intreg {capacitatea maximă de memorare}

n: Intreg {dimensiunea efectivă a vectorului – număr elemente memorate}

e: TElement[1..cp] {elementele memorate}

subalgoritm adaugăSfârșit. ( $v, e$ ) este  $\{\Theta(n)\}$

daca  $v.n = v.cp$  atunci

{vectorul e plin}

@ redimensionare (reallocare, copiere, ...)

sf daca

{dacă e goal, vectorul n-a redimensionat}

$v, e [v.n+1] \leftarrow e$

$v.n \leftarrow v.n + 1$

### Complexitate amortizată

- timp de execuție mediu pentru o secvență de operații pentru care se consideră evaluarea în cel mai defavorabil caz
- diferă de evaluarea pentru cazul mediu, nu sunt folosite probabilități și se evaluatează secvențe de operații, pentru care se face media

De ex. analiza amortizată pentru operația **adaugăSfârșit**

- dacă tabloul e plin, se doublează spațiul (reallocare, copiere, eliberarea vechiului spațiu)
- dacă se consideră **n** operații de adăugare la sfârșit, n fiind dimensiunea tabloului  $\Rightarrow$  redimensionarea se va face cel mult o dată  $\Rightarrow$

$$\frac{n \text{ operatii} \quad \text{adaugăSfârșit}}{\underbrace{\Theta(1) + \Theta(1) + \dots + \Theta(1) + \Theta(n)}_{n-1 \text{ ori}}} = \frac{\Theta(n)}{n} = \Theta(1)$$

#### IteratorVectorDinamic

v: VectorDinamic {ca implementare, va fi o referință către container}

current: Intreg {poziția unui element din vector}

**subalgoritm** creează( $i, v$ ) este  $\{\Theta(1)\}$

{se creează iteratorul  $i$  pe vectorul dynamic  $v$ }

$i.v \leftarrow v$

$i.current \leftarrow 1$

**sfCreează**

**funcția**  $valid(i)$  **este** {  $\theta(1)$  }

{verifică dacă iteratorul este valid}

$valid \leftarrow (i.current \leq i.v.n)$

**sfValid**

**subalgoritm**  $element(i, e)$  **este** {  $\theta(1)$  }

{ $e$  este elementul curent referit de iterator}

$e \leftarrow i.v[i.current]$

**sfElement**

**subalgoritm**  $urmator(i)$  **este** {  $\theta(1)$  }

{se deplasează referința curent în container}

$i.current \leftarrow i.current + 1$

**sfUrmator**

## Observații

1. Complexitatea operațiilor unui TAD poate fi determinată după ce s-au luat decizii legate de reprezentarea și modul de implementare a TAD-ului. Considerând reprezentarea secvențială a unui vector dinamic, operațiile de bază din interfața **TAD VectorDinamic** au complexitățiile:

- **dim** complexitate-timp  $\theta(1)$
- **element** complexitate-timp  $\theta(1)$
- **modifică** complexitate-timp  $\theta(1)$
- **adaugaSfarsit** complexitate-timp amortizată  $\theta(1)$
- **adaugaPozitie** complexitate-timp  $O(n)$
- **stergeSfarsit** complexitate-timp  $\theta(1)$   
!!! dacă se folosește redimensionare, atunci complexitatea-timp amortizată e  $\theta(1)$
- **stergePozitie** complexitate-timp  $O(n)$
- **iterator** complexitate-timp  $\theta(1)$

2. Deoarece se permite modificarea capacitatei vectorului, se impune pentru implementare folosirea **alocării dinamice a memoriei**.
3. După cum menționam anterior, este posibil ca la redimensionarea capacitatei vectorului să se folosească un raport de creștere  $RC$  (a capacitatei față de numărul de elemente din vector) - în specificația anterioară am folosit  $RC = 2$ .
4. Pentru o gestionare mai eficientă a spațiului de memorie alocat vectorului, pentru a evita situații în care numărul de elemente efectiv memorate în vector este mult mai mic decât capacitatea de memorare a

acestuia, se poate folosi un raport maxim  $RM$ , care asigură faptul că nu se ajunge la o creștere a capacitatei prea mare față de numărul de elemente. Ceea ce înseamnă că la operația de ștergere, în cazul în care  $\frac{v.cp}{v.n} > RM$ , atunci se va micșora capacitatea vectorului (ceea ce va implica realocare/copiere elemente...)

5. Spre deosebire de operația **adaugaSfarsit**, a cărei complexitate-timp defavorabilă este  $\theta(n)$ , dar totuși complexitatea-timp amortizată este  $\theta(1)$ , operația **adaugaPozitie** are complexitatea-timp amortizată  $O(n)$  (ca și cea defavorabilă).
6. În interfața TAD VectorDinamic pot fi adăugate și alte operații, spre exemplu: verificarea dacă vectorul este sau nu vid (fără elemente), căutarea unui element în vector și returnarea poziției pe care apare, transformarea în string (*toString*), transformarea în vector (*toArray*) etc.

### Reprezentare secvențială – caracteristici

- inserări, ștergeri  $O(n)$
- gestionare ineficientă a spațiului de memorare
- accesul la elemente este **direct**  $\theta(1)$
- adăugare la sfârșit  $\theta(1)$  amortizat
- ștergere la sfârșit  $\theta(1)$  (amortizat, dacă se face redimensionare)

# TAD Coadă (QUEUE)

## Observații:

1. În limbajul uzual cuvântul “coadă” se referă la o înșiruire de oameni, mașini, etc., aranjați în ordinea sosirii și care așteaptă un eveniment sau serviciu.
  - Noii sosiți se poziționează la sfârșitul cozii.
  - Pe măsură ce se face servirea, oamenii se mută către o poziție înainte, până când ajung în față și sunt serviti, asigurându-se astfel respectarea principiului “primul venit, primul servit”.
  - Exemple de cozi sunt multiple: coada de la benzinării, coada pentru cumpărarea unui produs, etc. Tipul de date **Coadă** permite implementarea în aplicații a acestor situații din lumea reală.
2. O *coadă* este o structură liniară de tip listă care restricționează adăugările la un capăt și ștergerile la celălalt capăt (lista FIFO - *First In First Out*).
3. **Accesul** într-o coadă este *prespecificat* (se poate accesa doar elementul cel mai devreme introdus în coadă), nu se permite accesul la elemente pe baza pozitiei. Dintr-o coadă se poate **șterge** elementul CEL MAI DEVREME introdus (primul).
4. Se poate considera și o capacitate inițială a cozii (număr maxim de elemente pe care le poate include), caz în care dacă numărul efectiv de elemente atinge capacitatea maximă, spunem că avem o *coadă plină*.
  - adăugarea în coada plină se numește **depășire superioară**.
5. O coadă fără elemente o vom numi *coadă vidă* și o notăm  $\Phi$ .
  - ștergerea din coada vidă se numește **depășire inferioară**.
6. O coadă în general nu se iterează.
7. Cozile sunt frecvent utilizate în programare - crearea unei cozii de așteptare a task-urilor într-un sistem de operare.
  - dacă task-urile nu au asociată o prioritate, ele sunt procesate în ordinea în care intră în sistem → **Coadă**.
  - dacă task-urile au asociate o prioritate și trebuie procesate în ordinea priorității lor → **Coadă cu priorități**.

## Tipul Abstract de Date COADA:

**domeniu:**  $\mathcal{C} = \{c \mid c \text{ este o coadă cu elemente de tip } TElement\}$

## operații:

- **creeaza( $c$ )**  
 $\{\text{creează o coadă vidă}\}$

*pre : true*  
*post :  $c \in \mathcal{C}, c = \Phi(\text{coada vidă})$*

- **adauga( $c, e$ )**  
 $\{\text{se adaugă un element la sfârșitul cozii}\}$

*pre :  $c \in \mathcal{C}, e \in TElement, c \text{ nu e plină}$*   
*post :  $c' \in \mathcal{C}, c' = c \oplus e, e \text{ va fi cel mai recent element introdus în coadă}$*

© aruncă excepție dacă coada e plină

- **sterge( $c, e$ )**  
 $\{\text{se sterge primul element introdus în coadă}\}$

*pre :  $c \in \mathcal{C}, c \neq \Phi$*   
*post :  $e \in TElement, e \text{ este elementul cel mai devreme introdus în coadă}, c' \in \mathcal{C}, c' = c \ominus e$*

© aruncă excepție dacă coada e vidă

- **element( $c, e$ )**  
 $\{\text{se accesează primul element introdus în coadă}\}$

*pre :  $c \in \mathcal{C}, c \neq \Phi$*   
*post :  $c' = c, e \in TElement, e \text{ este elementul cel mai devreme introdus în coadă}$*

© aruncă excepție dacă coada e vidă

- **vida ( $c$ )**

*pre :  $c \in \mathcal{C}$*   
*post :  $\text{vida} = \begin{cases} \text{adev}, & \text{dacă } c = \Phi \\ \text{fals}, & \text{dacă } c \neq \Phi \end{cases}$*

- **plina ( $c$ )**

*pre :  $c \in \mathcal{C}$*   
*post :  $\text{plina} = \begin{cases} \text{adev}, & \text{dacă } c \text{ e plină} \\ \text{fals}, & \text{contrar} \end{cases}$*

- **distruge( $c$ )**  
 $\{\text{destructor}\}$

*pre :  $c \in \mathcal{C}$*   
*post :  $c \text{ a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)}$*

## Observații

- Coada nu este potrivită pentru aplicațiile care necesită traversarea ei (nu avem acces direct la elementele din interiorul cozii).
- Afisarea conținutului cozii poate fi realizată folosind o coadă auxiliară (scoatem valorile din coadă punându-le într-o coadă auxiliară, după care se reintroduc în coada inițială). Complexitatea timp a subalgoritmului **tiparire** (descriș mai jos) este  $\theta(n)$ ,  $n$  fiind numărul de elemente din coadă.

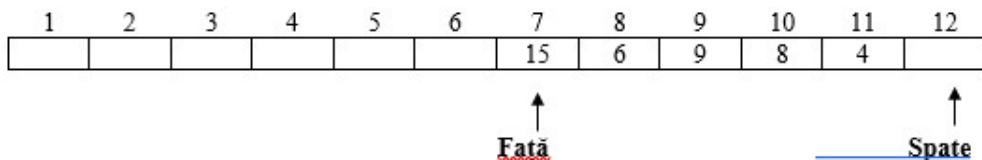
```
Subalgoritm tiparire(c)
{pre: c este o Coadă}
{post: se tipăresc elementele din Coadă}
creeaza(cAux) {se creează o coadă auxiliară vidă}
{se sterg elementele din c și se adaugă în cAux}
CatTimp ⊢ vida(c) execută
    sterge(c, e)
    @ tipărește e
    adauga(cAux, e)
SfCatTimp
{se sterg elementele din cAux și se refac c}
CatTimp ⊢ vida(cAux) execută
    sterge(cAux, e)
    adauga(c, e)
SfCatTimp
SfSubalgoritm
```

## Implementări ale cozilor folosind

- tablouri - vectori (dinamici) - reprezentare circulară (Figura 1, Figura 2).
- liste înlántuite.

## Generalizare a cozilor

- **Coada completă** (*Double ended queue - DEQUEUE*) - adăugări, ștergeri se pot face la ambele capete ale cozii.



$n = 12$  (capacitatea tabloului)

**Fată** = 7 - indicele unde e memorat primul element din coadă

**Spate** = 12 – primul indice liber din spatele cozii

Pp. că în vector memorăm elementele de la poziția 1

- Se adaugă în Spate, se șterge din Fată
- Elementele cozii se află între indicii Fată, Fată+1,...,Spate-1
- Initializarea cozii: **Fată=Spate=1** (dacă vectorul se memorează de la poziția 0, atunci **Fată=Spate=0**)
- Depășire inferioară (coada vidă): **Fată=Spate**
- Depășire superioară (coada plină): a) **Fată=1 și Spate=n** sau b) **Fată=Spate+1**

a) **Fată=1 și Spate=12**

1	2	3	4	5	6	7	8	9	10	11	12
2	8	9	3	5	7	15	6	9	8	4	

a) **Fată=7 și Spate=6**

+	1	2	3	4	5	6	7	8	9	10	11	12
	2	8	9	3	5		15	6	9	8	4	5

Figura 1: Reprezentare circulară pe tablou

### Reprezentare

#### Coada

cp: Intreg {capacitatea maximă de memorare}  
 Fată, Spate: Intreg {indică Fată, Spate}  
 e: TElement[1..cp] {elementele memorate}

**subalgoritm** creează(*c, cp*) este {  $\theta(1)$  }

```

c.cp  $\leftarrow$  cp
c.Fată  $\leftarrow$  1
c.Spate  $\leftarrow$  1

```

**sfCreează**

**subalgoritm** adaugă(*c, e*) este {  $\theta(1)$  }

{nu se verifică depășirea superioară}

```

c.e[c.Spate]  $\leftarrow$  e
dacă c.Spate = c.cp atunci
  c.Spate  $\leftarrow$  1
altfel
  c.Spate  $\leftarrow$  c.Spate + 1

```

**sfdacă**

**sfAdaugă**

**subalgoritm** șterge(*c, e*) este {  $\theta(1)$  }

{nu se verifică depășirea inferioară}

```

e  $\leftarrow$  c.e[c.Fată]
dacă c.Fată = c.cp atunci
  c.Fată  $\leftarrow$  1
altfel
  c.Fată  $\leftarrow$  c.Fată + 1

```

**sfdacă**

**sfAdaugă**

Figura 2: Operații pe coada reprezentată circular pe tablou

# Colecția

## COLLECTION, BAG, MULTI-SET

### Observatii

- **Colecție** ("bag") este un container, o grupare finită de elemente.
- Într-o colecție elementele nu sunt distințe (nu există o singură instanță a unui element).
  - Din această cauză colecția mai este cunoscută sub numele de **multi-mulțime** ("multi-set").
  - Operațiile specifice pe o colecție sunt: adăugarea, ștergerea, căutarea unui element într-o colecție.
    - Ca urmare tipul elementelor din colecție, **TElement** ar trebui să suporte cel puțin operațiile de: atribuire ( $\leftarrow$ ) și testarea egalității ( $=$ ).
- Caracteristică a colecției - nu conținează ordinea elementelor.
  - Spre exemplu, o colecție de numere întregi ar putea fi:  $c=\{1, 2, 3, 1, 3, 2, 4, 2, 2\}$ .

În continuare, vom prezenta specificația Tipul Abstract de Date **Colecție** (interfață minimală).

### domeniu

$Col = \{col \mid col \text{ este o colecție cu elemente de tip } TElement\}$

### operații (interfață TAD-ului Colecție)

*creează(c)*

**pre:** -

**post:**  $c \in Col$ ,  $c$  este colecția vidă (fără elemente)

*adaugă(c, e)*

**pre:**  $c \in Col$ ,  $e \in TElement$

**post:**  $c' \in Col$ ,  $c' = c + \{e\}$

{s-a adăugat elementul în colecție}

*sterge(c, e)*

**pre:**  $c \in Col$ ,  $e \in TElement$

**post:**  $c' \in Col$ ,  $c' = c - \{e\}$

{s-a eliminat o apariție a elementului din colecție}

{se poate returna adevărat dacă elementul a fost șters}

*caută(c, e)*

**pre:**  $c \in Col$ ,  $e \in TElement$

**post:**  $caută = \text{adevărat} \quad \text{dacă } e \in c$

fals în caz contrar

*dim(c)*

pre:  $c \in Col$

**post:**  $\dim =$  dimensiunea colecției  $c$  (numărul de elemente)  $\in \mathbb{N}$

vidă(c)

pre:  $c \in Col$

<b>post:</b> <i>vidă</i> = adevărat	în cazul în care c e colecția vidă
fals	în caz contrar

*iterator(c, i)*

pre:  $c \in Col$

**post:**  $i \in I$ ,  $i$  este un iterator pe colecția  $c$

*destroye(c)*

**pre:**  $c \in Col$

**post:** colecția c-a fost 'distrusă' (spațiul de memorie alocat a fost eliberat)

Menționăm că pot fi definite în interfața Tipului Abstract de Date Colecție și operații specifice cum ar fi: reunirea, intersecția, diferența a două colecții.

Deoarece colecția are o operație care furnizează un iterator pe elementele sale, subalgoritmul care va tipări elementele unei colecții *c* poate fi descris sub forma:

**Subalgoritmul tipărire(c) este**

{pre: c este o colecție}

{post: se tipăresc elementele colecției}

**iterator(c,i)** {colectia își construiește iteratorul}

**CâtTimp** valid(i) execută {cât timp iteratorul e valid}

**element(i, e)** {se obține elementul curent din iterare}

**@ tipărește e** {se tipărește elementul curent}

următor(i) {se deplasează iteratorul}

{se deplasează iteratorul}

SfCâtTimp

SfTipărire

Complexitatea timpului de lucru al algoritmului de tipărire este  $\Theta(|c|)$ , unde prin  $|c|$  am notat dimensiunea colecției  $c$ .

Ca și modalități de reprezentarea ale unei colecții, avem cel puțin următoarele posibilități:

- se reprezintă toate elementele colecției : 1, 2, 1, 4, 3, 1, 4, 2, 5;
  - se reprezintă colecția sub forma unor perechi de forma  $(e_1, f_1), (e_2, f_2), \dots, (e_n, f_n)$ , unde  $e_1, e_2, \dots, e_n$  reprezintă elementele distincte din colecție, iar  $f_1, f_2, \dots, f_n$  reprezintă frecvențele de apariție (numărul

de apariții în colecție) a elementelor corespunzătoare: spre exemplu colecția anterioară s-ar reprezenta sub forma perechilor  $(1, 3), (2, 2), (4, 2), (3, 1), (5, 1)$ .

**Observație.** În cazul în care elementele din colecție sunt de tip **TComparabil**, elementele pot fi memorate în ordine (în raport cu o anumită relație de ordine, de ex.  $\leq$ ), pentru a reduce complexitatea timp a unor operații.

Modalități de implementare ale colecțiilor ar putea fi folosind:

- tablouri (dinamice);
- liste înlănuite;
- tabele de dispersie;
- arbori binari (de căutare echilibrați).

În directorul TAD Colecție (curs 3) găsiți implementarea parțială, în limbajul C++, a containerului **Colecție**, reprezentare secvențială pe vector dinamic (se memorează toate elementele colecției în vector).

# Dicționar (MAP)

## Observații

- Elementele din dicționar sunt perechi de forma **(cheie, valoare)**. Dicționarele păstrează elemente în aşa fel încât ele să poată fi ușor localizate folosind **chei**.
- Spre exemplu, un dicționar poate păstra conturi bancare: fiecare cont este un obiect identificat printr-un număr de cont (considerat **cheia** elementului) și informații adiționale (numele și adresa deținătorului contului, informații despre depozite, etc). Informațiile adiționale vor fi considerate ca fiind **valoarea** elementului.
- Implementarea unui dicționar (SD aleasă pentru implementare) trebuie să ofere un mecanism eficient de regăsire a valorilor pe baza cheilor.
- Într-un dicționar cheile sunt **unice**.
- În general, o **cheie** are o unică **valoare** asociată. Dacă o cheie poate avea mai multe valori asociate => Multi-dicționar (**MultiMap**)

Dăm în continuare specificația Tipului Abstract de Date **Dicționar**.

## domeniu

$\mathcal{D} = \{d \mid d \text{ este un dicționar cu elemente } e = (c, v), c \text{ de tip } T\text{Cheie}, v \text{ de tip } T\text{Valoare}\}$

## operații (interfața TAD-ului Dicționar)

*creează(d)*

**pre:** true

**post:**  $d \in \mathcal{D}$ , d este dicționarul vid (fără elemente)

*adaugă(d, c, v)*

**pre:**  $d \in \mathcal{D}, c \in T\text{Cheie}, v \in T\text{Valoare}$ ,

**post:**  $d' \in \mathcal{D}, d' = d + (c, v)$  (se adaugă în dicționar perechea  $(c, v)$ )

{dacă există deja cheia în dicționar, înlocuiește valoarea asociată cheii și se poate returna vechea valoare. Dacă nu există cheia, adaugă perechea și se poate returna  $0_{T\text{Valoare}}$ }

*caută(d, c, v)*

**pre:**  $d \in \mathcal{D}, c \in T\text{Cheie}$

**post:** *caută*= adevărat  
fals

dacă  $(c, v) \in d$ , caz în care  $v \in T\text{Valoare}$  este valoarea asociată cheii c  
în caz contrar, caz în care  $v = 0_{T\text{Valoare}}$

*șterge*(d, c, v)

**pre:**  $d \in \mathcal{D}$ ,  $c \in T\text{Cheie}$

**post:**  $v \in T\text{Valoare}$

perechea  $(c, v)$  este ștearsă din dicționar, dacă  $c \in d$   
 $v = 0_{T\text{Valoare}}$  în caz contrar

*dim*(d)

**pre:**  $d \in \mathcal{D}$

**post:**  $\text{dim} = \text{dimensiunea dicționarului } d \text{ (numărul de elemente)} \in \mathcal{N}^*$

*vid*(d)

**pre:**  $d \in \mathcal{D}$

**post:**  $\text{vid} = \begin{cases} \text{adevărat} & \text{în cazul în care } d \text{ e dicționarul vid} \\ \text{fals} & \text{în caz contrar} \end{cases}$

*chei*(d, m)

**pre:**  $d \in \mathcal{D}$

**post:**  $m \in \mathcal{M}$ , m este mulțimea cheilor din dicționarul d

*valori*(d, c)

**pre:**  $d \in \mathcal{D}$

**post:**  $c \in \mathcal{Cof}$ , c este colecția valorilor din dicționarul d

*perechi*(d, m)

**pre:**  $d \in \mathcal{D}$

**post:**  $m \in \mathcal{M}$ , m este mulțimea perechilor (cheie, valoare) din dicționarul d

*iterator*(d, i)

{se creează un iterator pe dicționarul d}

**pre:**  $d \in \mathcal{D}$

**post:**  $i \in I$ , i este iterator pe dicționarul d

*distruge*(d)

**pre:**  $d \in \mathcal{D}$

**post:** dicționarul d a fost 'distrus' (spațiul de memorie alocat a fost eliberat)

Modalități de implementare ale dicționarelor:

- tablouri (dinamice);
- liste înlántuite;
- tabele de dispersie;
- arbori binari.

## Observații

### 1. Multi-dicționar (**MultiMap**)

- O cheie are o listă de valori asociate
- Operație din interfața TAD Dicționar a cărei specificație se modifică
  - **sterge(d, c, v)**

pre:  $d \in \mathcal{D}$ ,  $c \in T\text{cheie}$ ,  $v \in T\text{Element}$

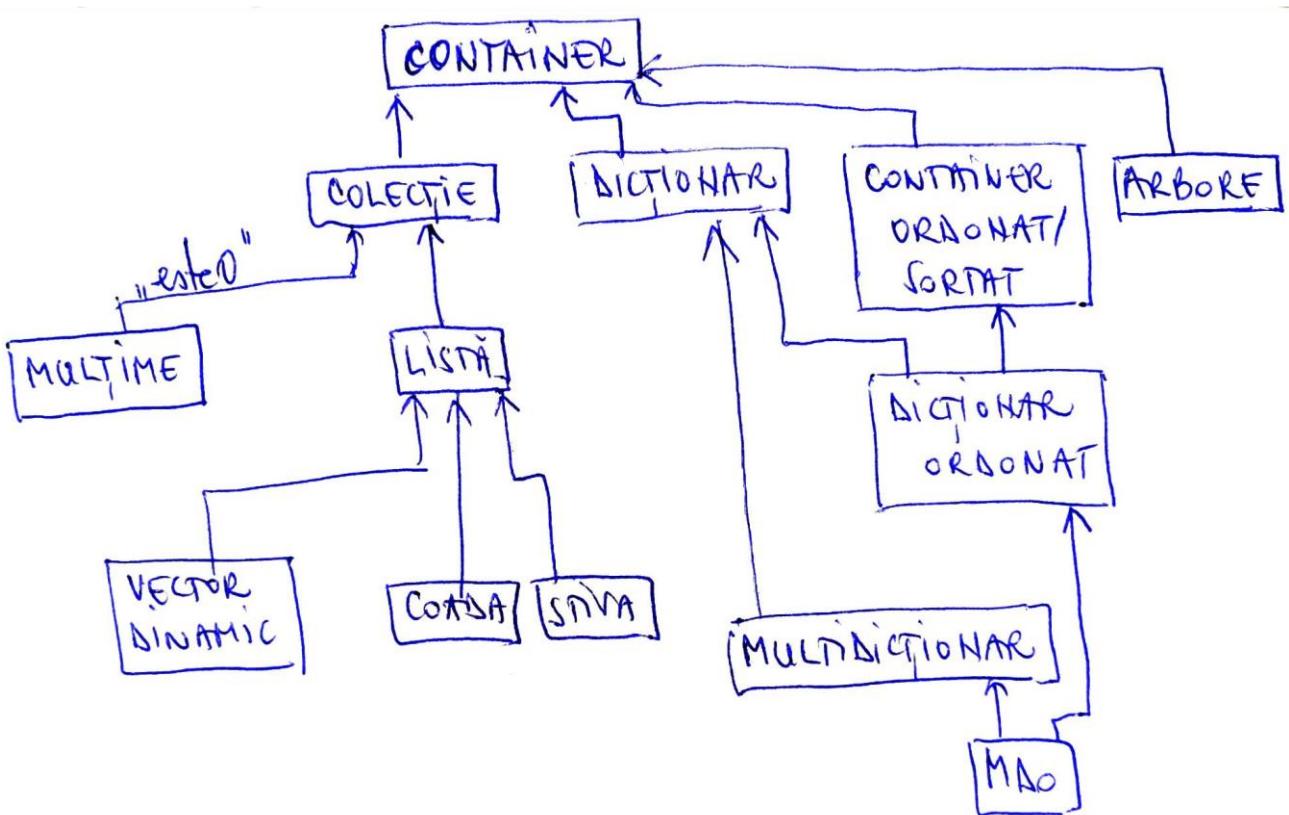
post:  $d' \in \mathcal{D}$

perechea  $(c, v)$  este ștearsă din dicționar, dacă  $c \in d$

### 2. Dicționar ordonat/sortat (**SortedMap**)

- **TCheie=TComparabil**
- Este definită o relație de ordine între chei  $\mathfrak{R} \subseteq T\text{cheie} \times T\text{cheie}$
- Nu se modifică interfața
- **Cerintă** – operațiile **iterator** și **perechi** returnează elementele în ordine în raport cu relația  $\mathfrak{R}$

### 3. Multi-dicționar ordonat/sortat (**Sorted MultiMap**)



# Matrice

## MATRIX - BIDIMENSIONAL ARRAY

Matricea este un tablou bidimensional static. Fără a da specificația completă a TAD Matrice, putem enumera un număr minim de operații în interfață sa:

- **creează**( $m, nrLin, nrCol$ ) {constructor - creează matricea nulă  $m$  având  $nrLin$  linii și  $nrCol$  coloane}
- **nrLinii**( $m$ ) {returnează numărul de linii}
- **nrColoane**( $m$ ) {returnează numărul de coloane}
- **element**( $m, i, j, e$ ) {accesare element de pe linia  $i$  și coloana  $j$  –  $e$  este elementul accesat}
- **modifică**( $m, i, j, e$ ) {înlocuirea cu  $e$  a elementului de pe linia  $i$  și coloana  $j$ }  
    {în cazul în care elementul nu există, îl adaugă}

Alte operații posibile: căutare element și returnarea (liniei, coloanei) pe care a fost găsit, iterator pentru accesare elemente în ordinea liniilor, iterator pentru accesare elemente în ordinea coloanelor, etc.

## Observații

1. În general, tablourile bidimensionale se memorează **secvențial**
2. Generalizare - tablouri bidimensionale **dinamice**.
3. În cazul în care multe elemente ale matricei sunt nule ( $0_{TElement}$ ) – *matrice rară* -, nu este eficientă memorarea tuturor elementelor matricei, ci doar a elementelor nenule.

În continuare vom discuta posibilități de reprezentare/implementare a matricelor rare. Vom lua un exemplu concret, matricea de mai jos

$$\begin{pmatrix} 0 & -2 & 0 & -7 & 0 \\ -6 & 0 & 0 & 0 & 0 \\ 0 & -9 & -8 & 0 & -5 \\ 0 & 0 & 0 & 0 & -2 \end{pmatrix}$$

A. *Reprezentare prin triplete <Linie, Coloana, Valoare>* (Valoare  $\neq 0_{TElement}$ ), ordonate lexicografic crescător în raport cu <Linie, Coloana>. Pentru exemplul de mai sus se vor memora următoarele triplete:

Linie	1	1	2	3	3	3	4
Coloana	2	4	1	2	3	5	5

Valoare	-2	-7	-6	-9	-8	-5	-2
---------	----	----	----	----	----	----	----

Tripletele se pot memora folosind

- un vector (dinamic) ordonat - reprezentare secvențială
- o listă înlănțuită ordonată (reprezentare înlănțuită)
  - înlănțuirile reprezentate folosind alocare dinamică
  - înlănțuirile reprezentate folosind alocare statică pe tablou
- un arbore binar de căutare/arbore AVL

**B.** *Reprezentare condensată pe coloane.* Se folosesc 3 vectori: Linie, Coloana, Valoare cu următoarea semnificație: elementele de pe coloana **j** ( $j=1,2,\dots,nrCol$ ) se află pe liniile

Linie[Coloana[j]], Linie[Coloana[j]+1],..., Linie[Coloana[j+1]-1]

și au valorile

Valoare[Coloana[j]], Valoare[Coloana[j]+1],..., Valoare[Coloana[j+1]-1]

Pentru exemplul de mai sus se vor memora următorii vectori:

- Coloana – având  $nrCol+1$  elemente
- Linie, Valoare – a căror dimensiune= $nr.$  de elemente nenule din matrice

indice	1	2	3	4	5	6
Coloana	1	2	4	5	6	8

indice	1	2	3	4	5	6	7
Linie	2	1	3	3	1	3	4
Valoare	-6	-2	-9	-8	-7	-5	-2

**C.** *Reprezentare condensată pe linii.* Se folosesc 3 vectori: Coloana, Linie, Valoare cu următoarea semnificație: elementele de pe linia **i** ( $i=1,2,\dots,nrLin$ ) se află pe liniile

Coloana[Linie[i]], Coloana[Linie[i]+1],..., Coloana[Linie[i+1]-1]

și au valorile

Valoare[Linie[i]], Valoare[Linie[i]+1],..., Valoare[Linie[i+1]-1]

Pentru exemplul de mai sus se vor memora următorii vectori:

- Linie – având  $nrLin+1$  elemente
- Coloana, Valoare – a căror dimensiune= $nr.$  de elemente nenule din matrice

indice	1	2	3	4	5
Linie	1	3	4	7	8

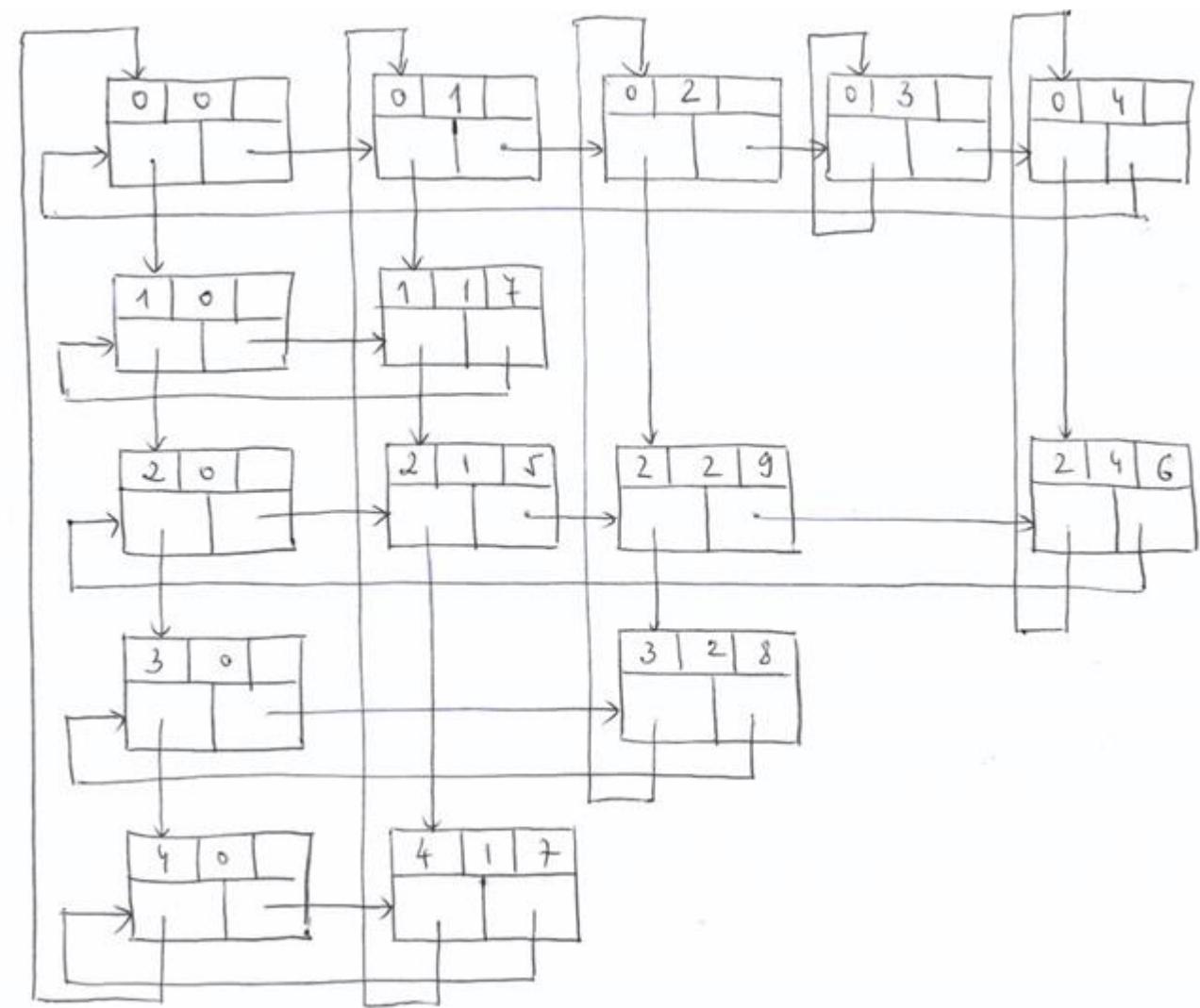
indice	1	2	3	4	5	6	7
Coloana	2	4	1	2	3	5	5
Valoare	-2	-7	-6	-9	-8	-5	-2

**D. Reprezentare înlățuită folosind liste circulare**

- Colecție de liste circulare interconectate

*Exemplu*

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 5 & 9 & 0 & 6 \\ 0 & 8 & 0 & 0 \\ 7 & 0 & 0 & 0 \end{pmatrix}$$



## Mulțimea (SET)

O **mulțime** ("set") este un container cu ajutorul căruia se poate reprezenta o colecție finită de elemente distincte. Altfel spus, într-o mulțime elementele nu se pot repeta, există o singură instanță a unui element. O altă caracteristică a mulțimii este faptul că într-o mulțime nu contează ordinea elementelor.

**Mulțimea** are toate operațiile specifice **Colecției**, cu observația că operația adăugare într-o mulțime are specificație diferită față de operația de adăugare într-o colecție (într-o mulțime elementele trebuie să fie distincte).

Tipul elementelor din mulțime, **TElement**, ca și într-o colecție, suportă cel puțin operațiile de: atribuire ( $\leftarrow$ ) și testarea egalității ( $=$ ).

Spre exemplu, o mulțime de numere întregi ar putea fi:  $m=\{1, 2, 3, 5, 4\}$ .

Caracterul finit al unei mulțimi ne permite (totuși) indexarea elementelor sale, ceea ce face ca la nivelul reprezentării interne, mulțimea **M** să poată fi asimilată cu un vector  $m_1, m_2, \dots, m_n$  (chiar dacă ordinea elementelor dintr-o mulțime nu este esențială).

Pentru a putea preciza modul în care se vor efectua operațiile pe mulțimi, vom defini structura de **submulțime**. Aceasta se poate realiza cu ajutorul unui vector format din valorile funcției caracteristice asociate submulțimii.

Dacă **M** este o mulțime, atunci submulțimea  $S \subseteq M$  va avea asociat vectorul  $V_S = (s_1, s_2, \dots, s_n)$  unde

$$s_i = \begin{cases} 1, & \text{daca } m_i \in S \\ 0, & \text{daca } m_i \notin S \end{cases}$$

De exemplu, dacă  $M = \{110, 200, 318, 400\}$ , atunci submulțimea  $S = \{200, 400\}$  a lui **M** se va reprezenta sub forma vectorului **(0, 1, 0, 1)** sau **(false, true, false, true)**.

Operațiile pe submulțimi pot fi acum definite prin intermediul operațiilor pe vectorii caracteristici asociați.

Fie  $S1, S2 \subseteq M$  două submulțimi ale mulțimii **M**. Atunci:

- a)  **$S1 \cup S2$  (reuniunea celor două submulțimi)** va fi caracterizată de vectorul **V** obținut din  $V_{S1}$  și  $V_{S2}$  efectuând operația logică " $\vee$ " (sau) element cu element

$\vee$	0	1
0	0	1
1	1	1

- b)  **$S1 \cap S2$  (intersecția celor două submulțimi)** va fi caracterizată de vectorul **V** obținut din  $V_{S1}$  și  $V_{S2}$  efectuând operația logică " $\wedge$ " (sau) element cu element

$\wedge$	0	1
0	0	0
1	0	1

Ca urmare, orice alte operații cu submulțimile unei mulțimi pot fi imaginate ca operații logice asupra vectorilor atașați.

În continuare, vom prezenta specificația Tipul Abstract de Date **Mulțime**.

## domeniu

$\mathcal{M} = \{m \mid m \text{ este o mulțime cu elemente de tip } \mathbf{TElement}\}$

## operații (interfață TAD-ului Mulțime)

*creează*(m)

**pre:** -

**post:**  $m \in \mathcal{M}$ , m este mulțimea vidă (fără elemente)

*adăugă*(m, e)

**pre:**  $m \in \mathcal{M}$ ,  $e \in \mathbf{TElement}$

**post:**  $m' \in \mathcal{M}$ ,  $m' = m \cup \{e\}$

{e se "reunește" la mulțime, adică se va adăuga numai dacă e nu mai apare în mulțime.}  
{se poate returna adevărat dacă elementul a fost adăugat}

*sterge*(m, e)

**pre:**  $m \in \mathcal{M}$ ,  $e \in \mathbf{TElement}$

**post:**  $m' \in \mathcal{M}$ ,  $m' = m - \{e\}$

{se sterge e din m}

{se poate returna adevărat dacă elementul a fost sters}

*caută*(m, e)

**pre:**  $m \in \mathcal{M}$ ,  $e \in \mathbf{TElement}$

**post:**  $cauta =$  adevărat      dacă  $e \in m$   
                      fals                în caz contrar

*dim*(m)

**pre:**  $m \in \mathcal{M}$

**post:**  $dim =$  dimensiunea mulțimii m (numărul de elemente)  $\in \mathcal{N}$

*vidă*(m)

**pre:**  $m \in \mathcal{M}$

**post:**  $vida =$  adevărat      în cazul în care m e mulțimea vidă  
                      fals                în caz contrar

*iterator*(m, i)

**pre:**  $m \in \mathcal{M}$

**post:**  $i \in \mathcal{I}$ , i este un iterator pe mulțimea m

*distruge*(m)

**pre:**  $m \in \mathcal{M}$

**post:** mulțimea  $m$  a fost 'distrusă' (spațiul de memorie alocat a fost eliberat)

Accesarea elementelor mulțimii se va face în aceeași manieră ca la colecție, folosind iteratorul pe care-l oferă mulțimea.

**Observație.** În cazul în care elementele din mulțime sunt de tip **TComparabil**, elementele pot fi memorate în ordine (în raport cu o anumită relație de ordine, de ex.  $\leq$ ), pentru a reduce complexitatea timp a unor operații.

Modalități de implementare ale mulțimilor:

- tablouri (dinamice);
- vectori booleeni (de biți);
- liste înlățuite;
- tabele de dispersie;
- arbori binari (de căutare echilibrați).

# Listă înlățuită

- Structură de date *dinamică* - una dintre cele mai simple și des folosite structuri fundamentale de date pentru reprezentarea TAD-urilor (*Colecție*, *Mulțime*, *Dicționar*, *Listă*, *Stivă*, *Coadă*, etc).
- Structură de date liniară.
- Listele înlățuite sunt folosite pentru reprezentarea *înlățuită* a containerelor de date.
- *Lista înlățuită* - colecție de elemente stocate în locații numite *noduri*, a căror **ordine** este determinată de o *legătură* (referință) conținută în fiecare nod.
  - ordine între *pozițiile* elementelor în cadrul listei
- Fiecare nod al listei conține:
  - elementul propriu-zis (informația utilă) din nodul listei (poate fi văzut ca o cheie a nodului); și
  - legături (referințe):
    - \* spre **următorul** element din listă; SI/SAU
    - \* spre **precedentul** elementul din listă.
- Nodurile unei liste înlățuite nu sunt memorate în locații succesive în memorie (așa cum se întâmplă la reprezentarea secvențială)
  - de aceea e necesar să memorăm nodul **următor/precedent**
- Caracteristica reprezentării înlățuite - **accesul secvențial**: un element va putea fi accesat pornind de la primul element al listei (numit și capul listei) și urmând legăturile până când elementul este găsit (operația are complexitate-timp *liniară*).
- Operații (depind de specificul containerului care se reprezintă folosind lista înlățuită):
  - inserare/ștergere elemente pe orice *pozitie* în listă
  - căutarea unei valori în listă
  - determinarea succesorului (predecesorului) unui element aflat pe o anumită *pozitie*
  - accesarea unui element pe baza *pozitioniei* sale în listă.

Există următoarele tipuri de reprezentări înlántuite:

### 1. Reprezentare simplu înlántuită. (Singly linked list)

- În fiecare nod este memorată legătura spre următorul element din listă.
- Lista va fi identificată printr-o referință la primul său element
- se poate memora și referință către ultimul element pentru a eficientiza anumite operații.

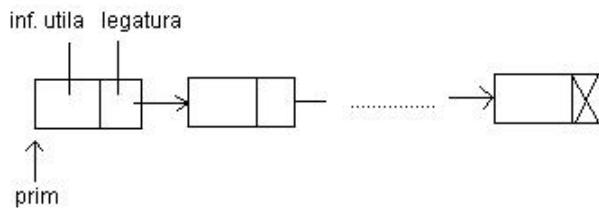


Figura 1: Reprezentarea simplu înlántuită a listelor.

- Dacă legăturile între noduri ar fi memorate sub formă de adrese în memorie (pointeri), atunci ultimul element va conține în câmpul de legătură pointerul nul (NIL).
  - În acest caz, dacă **prim** este NIL, atunci lista este vidă.

### 2. Reprezentare dublu înlántuită. (Double linked list)

- În fiecare nod sunt memorate legături atât spre următorul element din listă, cât și spre precedentul element.
- Lista va fi identificată prin căte o referință la primul și la ultimul său element.

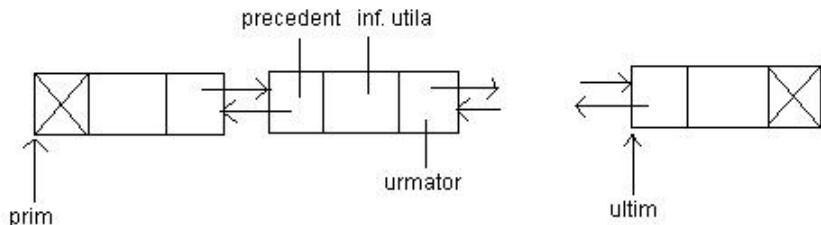


Figura 2: Reprezentarea dublu înlántuită a listelor.

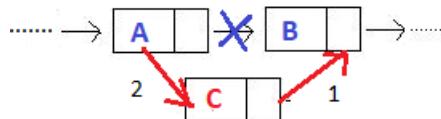
- Dacă legăturile între noduri ar fi memorate sub formă de adrese (pointeri), atunci ultimul element va conține în câmpul de legătură **urmator** pointerul nul (NIL), iar primul element va conține în câmpul de legătură **precedent** pointerul nul.
  - În acest caz, dacă **prim** este NIL și **ultim** este NIL, atunci lista este vidă.

### 3. Reprezentare înlănită circulară (Circular linked list) – reprezentare înlănită (simplu sau dublu) în care

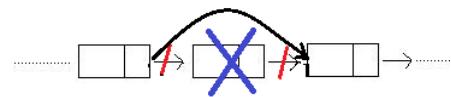
- ultimul element are o legătură spre primul element al listei.
- Lista va păstra o referință către ultimul element al listei - **ultim**
- referința către primul nod se obține din legătura spre următorul element al nodului ultim.

#### Avantaje ale reprezentării înlănită

- noi elemente pot fi adăugate sau șterse oriunde în listă, gestionând legăturile între elemente, fără costuri mari (complexitate-timp *constante*);
  - în Figura 3(a) este ilustrată o adăugare între două noduri într-o *listă simplu înlănită* -  $\theta(1)$ 
    - \* se creează nodul conținând informația utilă dorită (**C**)
    - \* se adaugă legăturile în ordinea 1, 2 (marcate cu **roșu**)
  - în Figura 3(b) este ilustrată ștergerea unui nod dintr-o *listă simplu înlănită* -  $\theta(1)$ 
    - \* ștergerea nodului marcat cu albstru presupune adădugarea legăturii evidențiate pe figură



(a) Adăugare element între două noduri.



(b) Ștergere nod.

Figura 3: Exemplu adăugare (a) și ștergere (b).

- în cazul în care se folosește alocarea dinamică, nu există limitare a capacitatii listei (număr de elemente).

#### Dezavantaje ale reprezentării înlănită

- spațiu suplimentar de memorie pentru memorarea legăturilor.
- accesul la elementul de pe poziția  $k$  este dificil (complexitate timp *liniară*).

#### Modalități de reprezentare a înlănuirilor

1. folosind alocare dinamică a memoriei (*poziția* în cadrul listei va fi adresa de memorare a unui nod al listei - *pointer*).
2. folosind alocare statică a memoriei (tablou) (*poziția* în cadrul listei va fi indicele din tablou unde se memorează un nod al listei - *întreg*).

## **Alte tipuri de liste înlățuite**

A se consulta documentul din Class Materials/Curs 4.

- **Liste dublu înlățuite de tip XOR** (*XOR double linked list*)
  - pentru a reduce spațiul de memorare pentru a stoca legăturile în nodurile unei liste dublu înlățuite
- **Skip lists**
  - structură eficientă de date pentru memorarea unui dicționar ordonat

# Operații pe liste înlănuite

## Reprezentarea înlănuirilor folosind alocare dinamică

- Vom prezenta, în cele ce urmează, în Pseudocod, implementarea unor operații pe liste înlănuite, folosind alocare dinamică pentru reprezentarea înlănuirilor
- Vom prezenta reprezentarea și câteva operații specifice pe următoarele structuri de date:
  - lista simplu înlănuită
  - lista dublu înlănuită
  - lista simplu înlănuită sortată
- Reamintim convențiile de notații Pseudocod pentru pointeri
  - Pentru a indica pointeri (adrese ale unor zone de memorie), vom folosi caracterul  $\uparrow$ , cu alte cuvinte dacă vrem să declarăm un pointer  $p$  care referă un număr întreg, acest lucru îl vom scrie în următoarea manieră:

$p : \uparrow Intreg$

Conținutul locației referite de pointerul  $p$  îl vom nota  $[p]$ .

- Pointerul nul (care nu referă nimic) îl vom nota prin NIL.
- Operațiile de alocare, respectiv dealocare a pointerilor le vom nota:
  - \*  $\text{aloca}(p)$
  - \*  $\text{dealoca}(p)$

### Lista simplu înlănuită (LSI)

Pentru reprezentarea LSI avem nevoie de două structuri: una pentru un **nod** și o structură pentru listă.

#### NodLSI

e: TElement //informația utilă a nodului

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

#### LSI

prim:  $\uparrow$  NodLSI //adresa primului nod din listă

{ultim:  $\uparrow$  NodLSI} //eventual adresa ultimului nod din listă

- În general, pentru o LSI se memorează doar adresa primului element din listă (**prim**). Se poate memora și adresa ultimului element din listă (**ultim**), caz în care operația de adăugare la sfârșit va avea complexitate timp constantă  $\theta(1)$ .

Pentru operațiile de adăugare, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă dată.

Figura 1 ilustrează crearea unui nod cu o informație utilă  $e$ .

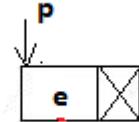


Figura 1: Creare nod cu informație utilă  $e$ .

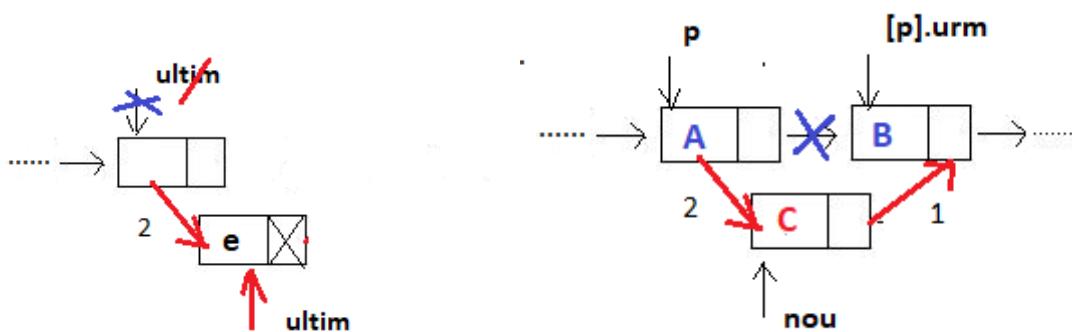
```

Functia creeazaNodLSI(lsi, e)
{pre: lsi: LSI, e: TElement}
{post: se returneză un  $\uparrow$  NodLSI conținând  $e$  ca informație utilă}
{se alocă un spațiu de memorare pentru un NodLSI }
{p:  $\uparrow$  NodLSI}
aloca(p)
[p].e  $\leftarrow$  e
[p].urm  $\leftarrow$  NIL
{rezultatul returnat de funcție}
creeazaNodLSI  $\leftarrow$  p
SfFunctia
    
```

- Complexitate:  $\theta(1)$

Subalgoritmul pentru adăugarea unui element la finalul listei este descris mai jos. Considerăm că lista memorează și adresa ultimului nod din listă (**ultim**).

Figura 2 ilustrează adăugarea la sfârșit și adăugarea după un anumit nod  $p$ .



(a) Adăugare element la sfârșit.

(b) Adăugare element după nod  $p$  diferit de *ultim*.

Figura 2

```

Subalgoritm adaugaSfarsit(lsi, e)
{pre: lsi: LSI, e: TElement}
{post: se adaugă  $e$  la finalul lsi}
nou  $\leftarrow$  creeazaNodLSI(lsi, e)
    
```

```

{dacă lista nu e vidă }
Daca lsi.ultim ≠ NIL atunci
    {se adaugă după ultim}
    [lsi.ultim].urm ← nou
altfel
    {nodul adăugat este și primul}
    lsi.prim ← nou
SfDaca
{se actualizează ultim}
lsi.ultim ← nou
SfSubalgoritm

```

- Complexitate:  $\theta(1)$

Subalgoritmul pentru adăugarea unui element după un nod din listă (indicat prin adresa sa - pointer). Considerăm că lista memorează și adresa ultimului nod din listă (**ultim**). Sunt două cazuri care trebuie tratate

- adăugare după *ultim* (dacă  $p = \text{ultim}$ ) (Figura 2(a))
- Adăugare element după nod  $p$  diferit de *ultim* (Figura 2(b))

```

Subalgoritm adaugaDupa(lsi, p, e)
{pre: lsi: LSI, p :↑ NodLSI, p ≠ NIL este adresa unui nod din lsi, e: TElement}
{post: se adaugă e după nodul indicat de p}
nou ← creeazaNodLSI(lsi, e)
{dacă se adaugă după ultimul nod }
Daca p = lsi.ultim atunci
    {se adaugă după ultim care e diferit de NIL, din precondiție}
    [lsi.ultim].urm ← nou
    {se actualizează ultim}
    lsi.ultim ← nou
altfel
    {se adaugă între p și [p].urm}
    [nou].urm ← [p].urm
    [p].urm ← nou
SfDaca
SfSubalgoritm

```

- Complexitate:  $\theta(1)$

Subalgoritmul pentru ștergerea unui nod din listă (indicat prin adresa sa - pointer). Considerăm că lista memorează și adresa ultimului nod din listă (**ultim**). Sunt 3 cazuri la ștergere:

- se șterge *prim* (Figura 3 (a));
- se șterge *ultim* (Figura 3 (b));
- se șterge un nod  $p$  diferit de *prim* și *ultim* (Figura 4).

```

Subalgoritm sterge(lsi, p, e)
{pre: lsi: LSI, p :↑ NodLSI, p ≠ NIL este adresa unui nod din listă}

```



(a) Ștergere *prim*.

(b) Ștergere *ultim*.

Figura 3

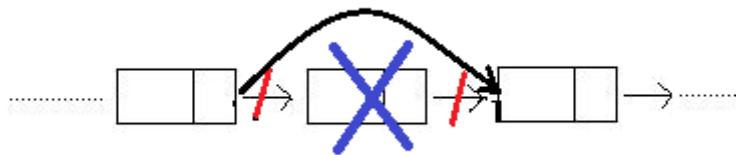


Figura 4: Ștergere nod *p* diferit de *prim* și *ultim*.

```

{post: se șterge din listă nodul indicat de p, e: TElement este elementul
șters}
{elementul șters}
e  $\leftarrow [p].e$ 
{dacă se șterge primul element al listei}
Daca p = lsi.prim atunci
    {se modifică prim}
    lsi.prim  $\leftarrow [p].urm$ 
    {dacă noul prim e NIL, atunci lista e vidă}
    Daca lsi.prim = NIL atunci
        lsi.ultim  $\leftarrow$  NIL
    SfDaca
    altfel
        {se parcurge pâna la nodul p}
        q  $\leftarrow lsi.prim$ 
        {sigur p este în listă, prin precondiție}
        CatTimp [q].urm  $\neq p$  executa
            q  $\leftarrow [q].urm$ 
        SfCatTimp
        {q este nodul care precede p }
        {dacă se șterge ultimul element al listei }
        Daca p = lsi.ultim atunci
            lsi.ultim  $\leftarrow q$ 
            {lista nu poate fi vidă, s-a tratat cazul la început}
        altfel
            {se șterge nodul p }
            [q].urm  $\leftarrow [p].urm$ 
        SfDaca
    SfDaca

```

```

{deallocăm spațiul de memorare pentru p }
dealloca(p)
SfSubalgoritm

```

- Complexitate:  $O(n)$ ,  $n$  fiind numărul de elemente din listă
  - cazul favorabil  $\theta(1)$  - șterg la început/sfârșit
  - cazul defavorabil  $\theta(n)$  - șterg penultimul element al listei

### **Iterator pe un container reprezentat folosind o LSI**

Presupunem că avem un **Container** oarecare (de ex. Colecție) reprezentat sub forma unei LSI, după cum urmează.

#### NodLSI

e: TElement //informația utilă nodului  
 urm:  $\uparrow$  NodLSI //adresa la care e memorat următorul nod

#### Container

prim:  $\uparrow$  NodLSI //adresa primului nod din listă

În acest caz, iteratorul pe Container ar trebui să conțină:

- o referință către container
- adresa unui nod din lista simplu înlanțuită folosită pentru reprezentarea containerului (*current*)

#### IteratorContainer

c : Container //containerul pe care îl iterează  
 current:  $\uparrow$  NodLSI //adresa unui nod current al LSI

Operațiile specifice ale iteratorului (creează, valid, element, următor) le vom descrie, mai jos, în Pseudocod. Toate operațiile au complexitate timp  $\theta(1)$ .

```

Subalgoritm creeaza(i, c)
  {pre: c este un container}
  {post: se creează iteratorul i pe containerul c}
  {
    elementul current al iteratorului referă primul element din c
    {se setează containerul în iterator}
    i.c ← c
    {se setează elementul current al iteratorului}
    i.current ← c.prim
  }
SfSubalgoritm

```

```

Functia valid(i)
  {pre: i este un iterator}
  {post: se verifică dacă elementul current este valid}
  {iteratorul este valid dacă elementul current este diferit de NIL }

```

```

    valid ← i.curent ≠ NIL
SfFunctia

Subalgoritm element(i,e)
{pre: i este un iterator, i este valid}
{post: e este elementul indicat de current}
e ← [i.curent].e
SfSubalgoritm

```

```

Subalgoritm urmator(i)
{pre: i este un iterator, i este valid}
{post: se deplasează referința current a iteratorului}
i.curent ← [i.curent].urm
SfSubalgoritm

```

În directorul asociat Cursului 4 găsiți implementarea parțială, în limbajul C++, a containerului **Colecție** (repräsentarea este sub forma unei LSI care memorează toate elementele colecției, folosind alocare dinamică pentru reprezentarea înlățuirilor).

**TEMĂ.** Scrieți în Pseudocod/implementați restul operațiilor specifice pe LSI și deduceți complexitățile acestora:

- adăugare element la începutul listei (adaugaInceput(lsi, e))
- adăugare element înaintea unui nod dat (adaugaInainte(lsi, p, e))
- căutarea unui element dat în listă (cauta(lsi, e))

## **Lista dublu înlățuită (LDI)**

Pentru reprezentarea LDI avem nevoie de două structuri: una pentru un **nod** și o structură pentru listă.

### NodLDI

e: TElement //informația utilă nodului  
 urm:  $\uparrow$  NodLDI //adresa la care e memorat următorul nod  
 prec:  $\uparrow$  NodLDI //adresa la care e memorat nodul precedent

### LDI

prim:  $\uparrow$  NodLDI //adresa primului nod din listă  
 ultim:  $\uparrow$  NodLDI //adresa ultimului nod din listă

Pentru operațiile de adăugare, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă.

```

Functia creeazaNodLDI(ldi, e)
{pre: ldi: LDI, e: TElement}
{post: se returneză un  $\uparrow$  NodLDI conținând e ca informație utilă}
{se alocă un spațiu de memorare pentru un NodLDI }

```

```

{p:   $\uparrow$  NodLDI}
aloca(p)
[p].e  $\leftarrow$  e
[p].urm  $\leftarrow$  NIL
[p].prec  $\leftarrow$  NIL
{rezultatul returnat de funcție}
creeazaNodLDI  $\leftarrow$  p
SfFunctia

```

- Complexitate:  $\theta(1)$

Subalgoritmul pentru adăugarea unui element înaintea unui nod din listă (indicat prin adresa sa - pointer). Sunt două cazuri care trebuie tratate

- adăugare înainte de *prim* (dacă  $p = \text{prim}$ ) (Figura 5(a))
- adăugare înaintea unui nod  $p$  diferit de *prim* (Figura 5(b))

Figura 5 ilustrează adăugarea la început și adăugarea înaintea unui nod  $p$  diferit de *prim*.

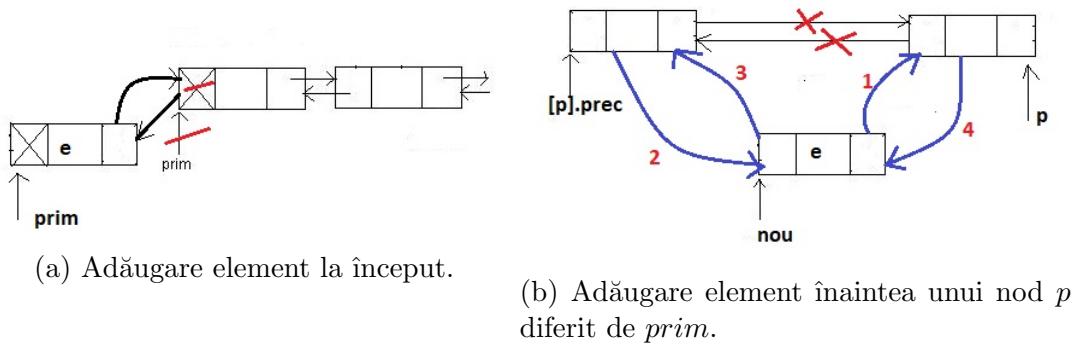


Figura 5

Subalgoritm adaugaInainte( $ldi, p, e$ )

```

{pre: ldi: LDI, p: $\uparrow$  NodLDI, p  $\neq$  NIL este adresa unui nod din ldi, e: TElement}
{post: se adaugă e înaintea nodului indicat de p}
nou  $\leftarrow$  creeazaNodLDI(ldi, e)
{dacă se adaugă înaintea primului nod }
Daca  $p = ldi.\text{prim}$  atunci
    {se adaugă înainte de prim}
    [nou].urm  $\leftarrow$  ldi.\text{prim}
    {p este diferit de NIL, prin precondiție}
    [ldi.\text{prim}].prec  $\leftarrow$  nou
    {se actualizează prim}
    ldi.\text{prim}  $\leftarrow$  nou
altfel
    {se adaugă între [p].prec și p}
    [nou].urm  $\leftarrow$  p
    [[p].prec].urm  $\leftarrow$  nou
    [nou].prec  $\leftarrow$  [p].prec

```

```


[p].prec← nou



SfDaca



SfSubalgoritm


```

- Complexitate:  $\theta(1)$

## TEMĂ

- Scrieți în Pseudocod/implementați restul operațiilor specifice pe LDI și deduceți complexitățile acestora:
  - adăugare element la începutul și sfârșitul listei
  - adăugare element după un nod dat
  - căutarea unui element dat în listă
  - ștergerea unui nod din listă
- Similar cu ce s-a prezentat pentru LSI, scrieți în Pseudocod operațiile pe iteratorul unui **Container** oarecare (de ex. Colecție) reprezentat sub forma unei LDI, folosind alocare dinamică pentru reprezentarea înlățuirilor.

## **Lista simplu înlățuită sortată/ordonată (LSIO)**

Într-o LSIO, elementele sunt de **TComparabil** (**TElement=TComparabil**) și sunt memorate în ordine în raport cu o anumită relație de ordine  $\mathcal{R} \subseteq \text{TComparabil} \times \text{TComparabil}$  (reflexivă, tranzitivă și antisimetrică)

- de exemplu, dacă  $\mathcal{R} = \leq$ , atunci elementele vor fi stocate în ordine crescătoare:  
x. 4 7 9 11
- ca implementare, relația va fi o funcție (pointer spre funcție în C++)�

De exemplu,

```

typedef int TComparabil;
typedef TComparabil TElement;
typedef bool(*Relatie)(TElement, TElement);

//relatia <=
bool relatia1(TElement e1, TElement e2) {
    if (e1 <= e2) return true;
    else return false;
}

```

Pentru reprezentarea LSIO avem nevoie de două structuri: una pentru un **nod** și o structură pentru listă.

### NodLSIO

e: TElement //informația utilă nodului

urm:  $\uparrow \text{NodLSIO} // adresa la care e memorat următorul nod$

### LSIO

prim:  $\uparrow \text{NodLSI} // adresa primului nod din listă$

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

Pe LSIO va fi definită o singură operație de **adăugare**, numită, de obicei, *inserare*, care va inseră un element în LSIO astfel încât să se păstreze, după inserare, relația de ordine între elemente. Pentru operația de *inserare*, vom folosi funcția auxiliară **creeazaNodLSI** definită anterior pentru LSI.

- notăm prin  $a\mathcal{R}b$  faptul că  $a$  e în relația  $\mathcal{R}$  cu  $b$  (de ex.  $a \leq b$ ).

De exemplu, dacă vrem să inserăm un element  $e$  într-o LSIO în care  $\mathcal{R} = \leq$ ,  $3 \ 7 \ 9 \ 12 \ 18$ , identificăm 2 cazuri:

- adăugăm înaintea primului element: dacă  $e = 1$ , atunci lista devine  $1 \ 3 \ 7 \ 9 \ 12 \ 18$ 
  - dacă se memora și ultim, se tratează și cazul adăugării după ultimul element
- adăugăm undeva în interiorul listei: dacă  $e = 10$ , atunci lista devine  $3 \ 7 \ 9 \ 10 \ 12 \ 18$

**Subalgoritm insereaza(lsio, e)**

```

{pre: lsio: LSIO, e:TElement }
{post: se adaugă în LSIO elementul e, păstrând relația de ordine între elemente}
nou ← creeazaNod(lsio,e)
{dacă se adaugă la începutul listei }
Daca (lsio.prim = NIL) ∨ (e R [lsio.prim].e) atunci
    [nou].urm← lsio.prim
    lsio.prim ← nou
altfel
    {se parcurge pâna la nodul q după care trebuie adăugat nou}
    {dacă lista e 1 2 5, relația e ≤ și vrem să adăugăm 3, ne oprim pe nodul
     q cu informația 2}
    q ← lsio.prim
    CatTimp ([q].urm ≠ NIL) ∧ ¬ (e R [[q].urm].e) executa
        q ← [q].urm
    SfCatTimp
    {q este nodul după care se adaugă nou }
    [nou].urm ← [q].urm
    [q].urm ← nou
SfDaca
SfSubalgoritm

```

- Complexitate:  $O(n)$ ,  $n$  fiind numărul de elemente din listă
  - cazul favorabil  $\theta(1)$  - inserez la început
  - cazul defavorabil  $\theta(n)$  - adaug la finalul listei

Operația de ștergere din LSIO este similară cu cea definită anterior pe LSI. De asemenea, iteratorul pe un container reprezentat folosind o LSIO este similar cu cel definit anterior pe LSI.

**TEMĂ.** Analizați cazul listei dublu înlăncuite sortate/ordonate - similar cu ce s-a discutat în Secțiunea anterioară.

# Liste dublu înlăntuite de tip XOR

O structură de date pentru a reduce spațiul de memorare pentru a stoca legăturile din nodurile unei liste dublu înlăntuite.

- Pentru memorarea legăturilor în listă, se folosește alocare dinamică a memoriei (pointeri).
- Un nod al listei nu memorează legăturile **următor** (spre următorul nod al listei) și **precedent** (spre precedentul nod al listei), ci o singură legătură **link**.

## Nod

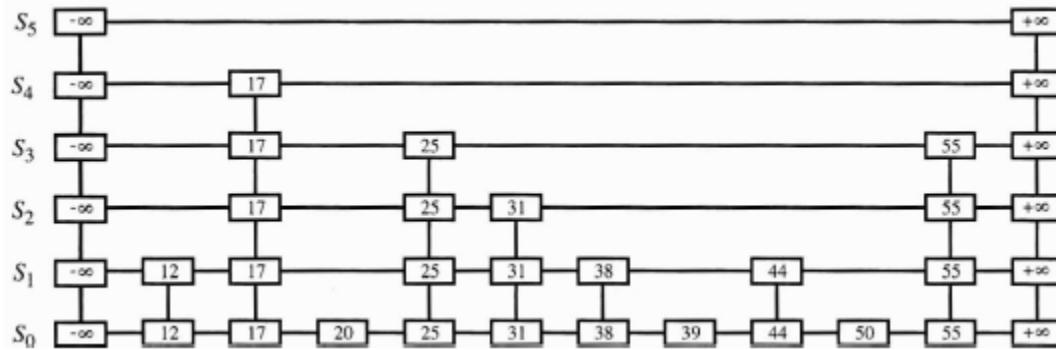
e: TElement {informația utilă a nodului}  
**link**:  $\uparrow$ Nod {pointer spre Nod}

- De exemplu, dacă lista este A→B→C.....atunci
  - **link(B)** =  $addr(A)$  XOR  $addr(C)$
  - $addr$  este funcția care indică adresa de memorare a unui anumit nod
  - XOR e operația **SAU exclusiv**
- Această convenție de memorare permite determinarea adresei de memorare a nodului următor unui anumit nod
  - De exemplu, dacă parcugem lista de la stânga la dreapta și suntem la nodul B (deci cunoaștem adresa de memorare a nodului A), putem determina adresa  $addr(C)$  la care e memorat nodul C în felul următor
    - $addr(C) = \text{link}(B)$  XOR  $addr(A)$   
 $( = addr(A) \text{ XOR } addr(C) \text{ XOR } addr(A))$   
 $= addr(A) \text{ XOR } addr(A) \text{ XOR } addr(C)$   
 $= 0 \text{ XOR } addr(C)$   
 $= addr(C)$   
)

# Skip Lists

O structură aleatorie (randomizată – *randomized data structure*) de stocare a datelor, eficientă pentru memorarea unui **dicționar ordonat** este **Skip List**

Ex: cheile 20, 17, 50, 44, 55, 12, 44, 31, 39,25



- Operațiile specifice (adăugare, căutare, modificare) necesită  $O(\log_2 n)$  cu o probabilitate mare (în medie) -  $O(n)$  caz defavorabil, dar puțin probabil să apară
- În Java există implementarea *ConcurrentSkipListMap*
- Intrările din  $S_{i+1}$  sunt alese aleator din intrările din  $S_i$ , alegând ca fiecare intrare din  $S_i$  să fie și în  $S_{i+1}$  cu probabilitatea 0.5.
- O poziție are 4 legături (*următor*, *precedent*, *sus*, *jos*)
- Căutare
  - Cu succes **39**:  $-\infty, 17, 17, 25, 25, 31, 31, 38, 38, 39$
  - Fără succes **41**:  $-\infty, 17, 17, 25, 25, 31, 31, 38, 38, 39$
- <https://www.ics.uci.edu/~pattis/ICS-23/lectures/notes/Skip%20Lists.pdf>
- <http://web.eecs.utk.edu/~bvz/cs302/notes/skip-lists.html>

# Liste

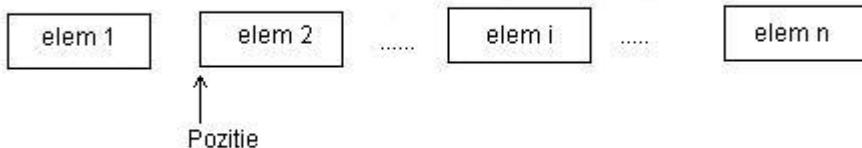
- In limbajul uzual cuvântul “listă” referă o “înşirare, într-o anumită *ordine*, a unor nume de persoane sau de obiecte, a unor date etc.”
- Exemple de liste sunt multiple: listă de cumpărături, listă de prețuri, listă de studenți, etc.
  - *Ordinea* în listă poate fi interpretată
    - ca un fel de „legătură” între elementele listei: după prima cumpărătură urmează a doua cumpărătură, după a doua cumpărătură urmează a treia cumpărătură, etc)
    - sau
    - poate fi văzută ca fiind dată de numărul de ordine al elementului în listă (1-a cumpărătură, a 2-a cumpărătură, etc).
  - Tipul de date Listă care va fi definit în continuare permite implementarea în aplicații a acestor situații din lumea reală.
- O *listă* o putem vedea ca pe o secvență de elemente  $\langle l_1, l_2, \dots, l_n \rangle$  de un același tip (TElement), fiecare element având o *poziție* bine determinată în cadrul listei, existând o ordine între pozițiile elementelor în cadrul listei
  - Lista poate fi văzută ca o colecție dinamică de elemente în care este esențială ordinea elementelor.
  - Numărul  $n$  de elemente din listă se numește *lungimea* listei.
  - O listă de lungime 0 se va numi lista *vidă*.
  - Caracterul de dinamicitate al listei este dat de faptul că lista își poate modifica în timp lungimea prin adăugări și ștergeri de elemente în/din listă.
- *Pozitia* elementelor în cadrul listei este esențială, astfel accesul, ștergerea și adăugarea se pot face pe orice *poziție* în listă.
- **Lista liniară**
  - o structură care fie este vidă (nu are nici un element), fie
    - are un unic prim element;
    - are un unic ultim element;
    - fiecare element din listă (cu excepția ultimului element) are un singur succesor;
    - fiecare element din listă (cu excepția primului element) are un singur predecesor.
  - Într-o listă liniară se pot insera elemente, șterge elemente, se poate determina succesorul (predecesorul) unui element aflat pe o anumită *poziție*, se poate accesa un element pe baza *poziției* sale în listă.

- Fiecare element al unei liste liniare are o *poziție* bine determinată în cadrul listei.
  - este importantă prima *poziție* în cadrul listei
  - *poziția* unui element este relativă la listă
  - *poziția* unui element din listă
    - identifică elementul din listă
    - poziția elementului predecesor și poziția elementului successor în listă (dacă acestea există)
  - ordine între pozițiile elementelor în cadrul listei.

**Poziția** unui element în cadrul listei poate fi văzută în diferite moduri:

1. ca fiind dată de **rangul** (numărul de ordine al) elementului în cadrul listei.
  - similitudine cu tablourile, *poziția* unui element în listă fiind *indexul* acestuia în cadrul listei.
  - Într-o astfel de abordare, lista este văzută ca un tablou dinamic în care se pot accesa/adăuga/șterge elemente pe orice poziție în listă.
2. ca fiind dată de o **referință** la locația unde se stochează elementul listei (ex: pointer spre locația unde se memorează elementul).

Pentru a asigura generalitatea, vom abstractiza noțiunea de *poziție* a unui element în listă și vom presupune că elementele listei sunt accesate prin intermediul unei *poziții* generice.



- **O Poziție**  $p$  într-o listă o considerăm **validă** dacă este poziția unui element al listei.
  - dacă  $p$  ar fi un pointer spre locația unde se memorează un element al listei, atunci  $p$  este **valid** dacă este diferit de pointerul nul sau de orice altă adresă care nu reprezintă adresa de memorare a unui element al listei.
  - dacă  $p$  ar fi rangul (numărul de ordine al) elementului în listă, atunci  $p$  este **valid** dacă e cuprins între 1 și numărul de elemente din listă.

# TAD Lista (LIST)

## Observații:

1. Tipul (abstract) de date *TPozitie* abstractizează noțiunea de poziție a unui element în listă (pentru a se asigura generalitatea).
2. O poziție  $p \in TPozitie$  din lista  $l$  o numim *poziție validă* dacă este poziția unui element din lista  $l$ .
3. În domeniul de valori a *TPozitie*, notată cu  $\perp$  o valoare specială p car o vom numi poziție nedefinită. Poziția nedefinită  $\perp$  nu este o poziție *validă* (conform celor menționate anterior).
4. Lista vidă o notăm cu  $\Phi$ .

## Tipul Abstract de Date LISTA:

### domeniu:

$$\mathcal{L} = \{l \mid l \text{ este o listă cu elemente de tip } TElement, \text{ fiecare element având o poziție unică în } l \text{ de tip } TPozitie\}$$

### operări:

- **creeaza( $l$ )**  
  {creează o listă vidă}  
    *pre* : true  
    *post* :  $l \in L, l = \Phi$
- **prim( $l$ )**  
    *pre* :  $l \in L$   
    *post* :  $prim = p \in TPozitie,$   
          
$$p = \begin{cases} \text{poziția primului element din lista } l, & \text{dacă } l \neq \Phi \\ \perp, & \text{dacă } l = \Phi \end{cases}$$
- **ultim ( $l$ )**  
    *pre* :  $l \in L$   
    *post* :  $ultim = p \in TPozitie,$   
          
$$p = \begin{cases} \text{poziția ultimului element din lista } l, & \text{dacă } l \neq \Phi \\ \perp, & \text{dacă } l = \Phi \end{cases}$$
- **valid( $l, p$ )**  
    *pre* :  $l \in L, p \in TPozitie$   
    *post* :  $valid = \begin{cases} \text{true}, & \text{dacă } p \text{ este o poziție a unui element din lista } l \\ \text{false}, & \text{altfel} \end{cases}$
- **următor( $l, p$ )**  
    *pre* :  $l \in L, p \in TPozitie, p \text{ poziție validă}$   
    *post* :  $urmator = q \in TPozitie,$   
          
$$q = \begin{cases} \text{poziția următoare poziției } p \text{ din lista } l, & \text{dacă } p \text{ nu e poziția ultimului element din lista } l \\ \perp, & \text{dacă } p \text{ e poziția ultimului element din lista } l \end{cases}$$

© aruncă excepție dacă  $p$  nu e validă

- **anterior( $l, p$ )**

*pre :*  $l \in L, p \in TPozitie, p$  poziție validă  
*post :*  $anterior = q \in TPozitie,$

$$q = \begin{cases} \text{poziția precedentă poziției } p \text{ din lista } l, \\ \quad \text{dacă } p \text{ nu e poziția primului element din lista } l \\ \perp, \quad \text{dacă } p \text{ e poziția primului element din lista } l \end{cases}$$

© aruncă excepție dacă  $p$  nu e validă
- **element( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p)$   
*post :*  $e \in TElement, e = \text{elementul de pe poziția } p \text{ din } l$

© aruncă excepție dacă  $p$  nu e validă
- **pozitie ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$ ,  
*post :*  $pozitie = p \in TPozitie,$

$$p = \begin{cases} \text{prima poziție a elementului } e \text{ din lista } l, & \text{dacă } e \in l \\ \perp, & \text{dacă } e \notin l \end{cases}$$
- **modifică( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul de pe poziția  $p$  din  $l' = e$

© aruncă excepție dacă  $p$  nu e validă
- **adaugalnceput ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :* elementul  $e$  a fost adăugat la începutul listei  $l$   
 $(l' = e \oplus l)$
- **adaugaSfarsit( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :* elementul  $e$  a fost adăugat la sfârșitul listei  $l$   
 $(l' = l \oplus e)$
- **adaugaDupa( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul  $e$  a fost inserat în lista  $l$  după poziția  $p$ ,  
 $\text{pozitie}(l', e) = \text{urmator}(l', p)$

© aruncă excepție dacă  $p$  nu e validă
- **adaugalnainte( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul  $e$  a fost inserat în lista  $l$  înaintea poziției  $p$ ,  
 $\text{pozitie}(l', e) = \text{anterior}(l', p)$

© aruncă excepție dacă  $p$  nu e validă
- **sterge ( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p)$   
*post :*  $e \in TElement, \text{elementul } e \text{ de pe poziția } p \text{ a fost sters din } l$

© aruncă excepție dacă  $p$  nu e validă
- **cauta ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :*  $cauta = \begin{cases} \text{adevarat}, & \text{dacă } e \text{ a fost găsit în lista } l \\ fals, & \text{altfel} \end{cases}$

- **vida** ( $l$ )

*pre* :  $l \in L$

*post* :  $\text{vida} = \begin{cases} \text{true}, & \text{dacă } l = \Phi \\ \text{false}, & \text{dacă } l \neq \Phi \end{cases}$

- **dim**( $l$ )

*pre* :  $l \in L$

*post* :  $\text{dim} = n \in \text{Natural}$ ,

$n = \text{numărul de elemente ale listei } l$

- **distruge**( $l$ )

{destructor}

*pre* :  $l \in L$

*post* :  $l$  a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)

- **iterator**( $l, i$ )

*pre* :  $l \in L$

*post* :  $i \in \mathcal{I}, i$  este un iterator pe lista  $l$

## Observații

- Operația **cauta** poate fi specificată mai general

- returnează prima *poziție* pe care apare un element în listă, dacă elementul e găsit în listă
- returnează *poziție invalidă* dacă elementul nu e găsit în listă

- Din perspectiva unei ierarhii de containere

- **Lista** este o **Colecție**

- **Vector Dinamic** este o **Listă**

- \* **Vectorul Dinamic** poate fi văzut ca o **Listă** reprezentată *secvențial*

- Există anumite dezavantaje induse de folosirea unui parametru de tip *TPozitie* în interfața listei:

1. Tipurile de referințe concrete folosite diferă în funcție de reprezentarea listei.
2. Interfața listei este destul de greoaie și nesigură prin faptul că expune în exterior pozițiile (referințele la locațiile din listă).
  - acesta este motivul pentru care bibliotecile existente particularizează tipul *TPozitie* expus în interfața containerului Listă (după cum se va vedea în continuare)

Implementări ale containerului **Lista** în biblioteci existente:

### 1. STL - list

- *poziția* este dată de un *iterator* pe listă  $\Rightarrow \text{TPozitie} = \text{Iterator}$ .
- în STL, *list* e văzut ca și un container de tip *secvențial*: elementele sunt aranjate într-o ordine (liniară) strictă.
- reprezentarea este dublu *înlănțuită*
  - dacă se dorește reprezentare simplu *înlănțuită*, se va folosi **forward\_list**.

- dacă se dorește reprezentare *secvențială*, se va folosi **vector**.

## 2. Java - List

- *poziția* este văzută ca un indice  $\Rightarrow TPozitie = Intreg$ .
  - permite accesarea elementelor din listă prin intermediul indicilor (ca la reprezentarea secvențială - **Vector Dinamic**)
- dacă se dorește reprezentare *înlănțuită* a listei, se va folosi **Linked List**.

## Modalități de implementare a unei liste

- memorând elementele sale **secvențial** într-un tablou/vector (dinamic)
  - accesul la elementele listei este *direct* ( $\theta(1)$ )
- memorând elementele sale **înlănțuit** într-o listă înlănțuită
  - accesul la elementele listei este *secvențial* ( $O(n)$ )
  - lista înlănțuită poate fi
    - \* simplu înlănțuită (LSI)
    - \* dublu înlănțuită (LDI)

## Analiza complexității timp a celor mai importante operații ale containerului Lista în funcție de implementarea acesteia

În Tabelul 1 vom considera, comparativ

- reprezentare secvențială folosind un vector dinamic (poziția este indice);
- reprezentare simplu înlănțuită (LSI) cu alocare dinamică (poziția este adresa de memorare a unui nod);
- reprezentare dublu înlănțuită (LDI) cu alocare dinamică (poziția este adresa de memorare a unui nod);

Notăm cu  $n$  numărul de elemente din listă. Observăm faptul că reprezentarea dublu înlănțuită este cea mai eficientă ca și timp, dar ocupă spațiu de memorare suplimentar pentru legături (pentru a reduce spațiul de memorare se pot folosi liste de tip XOR - a se vedea cursul 4).

Vom particulariza, în cele ce urmează, TAD-ul generic **Lista**, atfel încât să regăsim cele două specificații ale containerului **Lista** descrise anterior(STL/Java).

## Lista - cu poziție indice (indexată)

- corespunde modului în care este specificată lista în Java.
- *poziția* este văzută ca un indice  $\Rightarrow TPozitie = Intreg$ .
  - permite accesarea elementelor prin intermediul indicilor
- Accesul la elemente se face pe baza rangului, se permit inserări și ștergeri la orice poziție (poziția unui element reprezintă indicele acestuia în cadrul listei).

Operatie	Reprezentare secvențială	Reprezentare folosind o LSI alocată dinamic	Reprezentare folosind o LDI alocată dinamic
creeaza	$\theta(1)$	$\theta(1)$	$\theta(1)$
prim	$\theta(1)$	$\theta(1)$	$\theta(1)$
ultim	$\theta(1)$	$\theta(1)$ - dacă memorăm ultim $O(n)$ - fără a memora ultim	$\theta(1)$
următor	$\theta(1)$	$\theta(1)$	$\theta(1)$
anterior	$\theta(1)$	$O(n)$	$\theta(1)$
adaugaInceput	$\theta(n)$	$\theta(1)$	$\theta(1)$
adaugaSfarsit	$\theta(1)$ amortizat	$\theta(1)$ - dacă memorăm ultim $\theta(n)$ - fără a memora ultim	$\theta(1)$
adaugaDupa	$O(n)$	$\theta(1)$	$\theta(1)$
adaugaInainte	$O(n)$	$O(n)$	$\theta(1)$
sterge	$O(n)$	$O(n)$	$\theta(1)$

Tabela 1: Complexități timp ale operațiilor.

- O poziție  $i$  în cadrul listei  $l$  este *validă* dacă  $1 \leq i \leq \text{lungime}(l)$ .
- Se simplifică interfața
  - interfața este aceeași cu a unui **Vector Dinamic**

Specificația Listei indexate este dată mai jos

**domeniu:**

$$L = \{l \mid l = [e_1, e_2, \dots, e_n], e_i \in TElement \ \forall i = 1, 2, \dots, n\}$$

**operații:**

- **creeaza** ( $l$ )

*pre* : true  
*post* :  $l \in L, l = \Phi$  lista vidă

- **adaugaSfarsit**( $l, e$ )

*pre* :  $l \in L, e \in TElement$   
*post* : elementul  $e$  a fost adăugat la sfârșitul listei  $l$   
 $(l' = l \oplus e)$

- **adauga** ( $l, i, e$ )

*pre* :  $l \in L, e \in TElement, i \in \text{Intreg},$   
 $i$  poziție validă în  $l \vee i = \text{lungime}(l) + 1$   
*post* :  $l' = (e_1, \dots, e_{i-1}, e, e_i, e_{i+1}, \dots, e_n)$   
 $(\text{pozitie}(l', e) = i)$

© aruncă excepție dacă  $i$  nu e valid

- **sterge** ( $l, i, e$ )

*pre* :  $l \in L, l = (e_1, \dots, e_{i-1}, e_i, e_{i+1}, \dots, e_n), i \in \text{Intreg}, i$  poziție validă  
*post* :  $e \in TElement, e =$  elementul de pe poziția  $i$  din  $l$   
 $l' = (e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n)$   
 $(\text{pozitie}(l', e) = i)$

© aruncă excepție dacă  $i$  nu e valid

- cauta ( $l, e$ )
 

*pre* :  $l \in L, e \in TElement$

*post* :  $cauta = \begin{cases} i, & \text{dacă } i \text{ e prima pozitie pe care } e \text{ a fost găsit în lista } l \\ -1, & e \notin L \end{cases}$
- element ( $l, i, e$ )
 

*pre* :  $l \in L, i \in Intreg, i$  poziție validă

*post* :  $e \in TElement, e = \text{elementul de pe poziția } i \text{ din } l$

© aruncă excepție dacă  $i$  nu e valid
- modifica ( $l, i, e$ )
 

*pre* :  $l \in L, i \in Intreg, i$  poziție validă,  $e \in TElement$

*post* : elementul de pe poziția  $i$  din  $l' = e$

© aruncă excepție dacă  $i$  nu e valid
- vida ( $l$ )
 

*pre* :  $l \in L$

*post* :  $vida = \begin{cases} true, & \text{dacă } l = \Phi \\ false, & \text{altfel} \end{cases}$
- dim ( $l$ )
 

*pre* :  $l \in L$

*post* :  $dim = n \in Intreg,$   
 $n = \text{numărul de elemente din lista } l$
- iterator( $l, i$ )
 

*pre* :  $l \in L$

*post* :  $i \in \mathcal{I}, i$  este un iterator pe lista  $l$
- distrugе( $l$ )
 

*pre* :  $l \in L$

*post* :  $l$  a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)

## Exemplu

Considerăm reprezentarea Listei indexate folosind o LSI alocată dinamic. Descriem mai jos, în Pseudocod, operația **element**.

Reprezentarea listei este

### Nod

e: TElement //informația utilă nodului  
 urm:  $\uparrow$  Nod //adresa la care e memorat următorul nod

### Lista

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

Subalgoritm element( $l, i, e$ )

```
{pre: l: Lista, i:Intreg, 1 ≤ i ≤ lungime(l), e:TElement }
{post: e este al i-lea element al listei }
{se parcurge până la al i-lea element }
p ← l.prim
{se parcurg i-1 legături }
Pentru  $i = 1, i - 1$  executa
  p ← [p].urm
```

```

SfPentru
{p este al i-lea nod }
  e  $\leftarrow$  [p].e
SfSubalgoritm

```

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

Să considerăm subalgoritmul **tiparire** care tipărește elementele unei liste indexate reprezentate folosind o LSI alocată dinamic. Tipărirea trebuie realizată folosind iteratorul, în caz contrar, tipărirea se va realiza în timp pătratic în raport cu numărul de elemente din listă.

1. folosind un iterator: complexitate timp  $\theta(n)$ , *n* fiind numărul de elemente ale listei

```

Subalgoritm tiparire(l)
  {pre: l: Lista}
  {post: se tipăresc elementele listei}
  iterator(l, i)
    CatTimp valid(i) executa
      element(i, e)
      @tipărește e
      urmator(i)
    SfCatTimp
SfSubalgoritm

```

2. folosind accesul la elemente prin indici: complexitate timp  $\theta(n^2)$ , *n* fiind numărul de elemente ale listei

```

Subalgoritm tiparire(l)
  {pre: l: Lista}
  {post: se tipăresc elementele listei}
  Pentru i = 1, dim(l) executa
    element(l, i, e)
    @tipărește e
  SfPentru
SfSubalgoritm

```

## **Lista - cu poziție iterator**

- corespunde modului în care este specificată lista în STL.
- *poziția* este dată de un *iterator* pe listă  $\Rightarrow TPozitie = Iterator$ .
- se simplifică interfața
  - operațiile *următor*, *anterior*, *valid* și *element* sunt operațiile pe *iterator*

Enumerăm, mai jos, operațiile din interfața Listei în care accesul e pe baza unei poziții date de un iterator, fără a mai da specificația completă a operațiilor (specificațiile sunt cele indicate la containerul generic **Lista**, dar cu  $TPozitie = IteratorLista$ ).

### **Operații din interfață:**

- creeaza (*l* : Lista)
- vida (*l* : Lista)
- dim (*l* : Lista)
- IteratorLista prim(*l* : Lista)

- $TElement$  element( $l$  :Lista,  $poz$ :IteratorLista)
- $TElement$  modifica( $l$  :Listă,  $poz$ :IteratorLista,  $e$  :  $TElement$ )
- adaugaInceput( $l$  :Listă,  $e$  :  $TElement$ )
- adaugaSfarsit( $l$  :Listă,  $e$  :  $TElement$ )
- adauga ( $l$  :Listă,  $poz$ :IteratorListă,  $e$  :  $TElement$ )
- $TElement$  sterge( $l$  :Lista,  $poz$ :IteratorLista)
- IteratorLista cauta( $l$  :Lista,  $e$  :  $TElement$ )
- distruge ( $l$  : Lista)

## Exemplu

Considerăm reprezentarea Listei cu poziție iterator, folosind o LDI alocată dinamic. Descriem mai jos, în Pseudocod, operația **adaugaDupa**.

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

### IteratorLista

l: Lista//referință către listă  
 curent: $\uparrow$  Nod//adresa nodului curent din listă

Pentru operația de adăugare, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă.

```
Functia creeazaNod( $l, e$ )
{pre:  $l$ : Lista,  $e$ : TElement}
{post: se returneză un  $\uparrow$  Nod conținând  $e$  ca informație utilă}
{se alocă un spațiu de memorare pentru un Nod }
{ $p$ :  $\uparrow$  Nod}
aloca( $p$ )
[ $p$ ].e  $\leftarrow e$ 
[ $p$ ].urm  $\leftarrow$  NIL
[ $p$ ].prec  $\leftarrow$  NIL
{rezultatul returnat de funcție}
creeazaNod  $\leftarrow p$ 
```

### SfFunctia

- Complexitate:  $\theta(1)$

### Subalgoritm adaugaDupa( $l, i, e$ )

```
{pre:  $l$ : Lista,  $i$ : IteratorLista,  $i$  este valid,  $e$ : TElement}
{post: se adaugă  $e$  după nodul curent al lui  $i$ }
nou  $\leftarrow$  creeazaNod( $l, e$ )
 $p \leftarrow i.curent$ 
{se va adaugă după  $p$  }
```

```

{dacă p este ultimul nod al listei }
Daca p = l.ultim atunci
  {p este diferit de NIL, din precondiție }
    [l.ultim].urm  $\leftarrow$  nou
    [nou].prec  $\leftarrow$  l.ultim
    {se actualizează l.ultim}
    l.ultim  $\leftarrow$  nou
  altfel
    {se adaugă între p și [p].urm}
    [nou].urm  $\leftarrow$  [p].urm
    [[p].urm].prec  $\leftarrow$  nou
    [p].urm  $\leftarrow$  nou
    [nou].prec  $\leftarrow$  p
SfDaca
SfSubalgoritm

```

- Complexitate:  $\theta(1)$

În directorul TAD Lista (Curs 5) găsiți implementarea parțială, în limbajul C++, a containerului **Lista cu poziție iterator** (repräsentarea este sub forma unei LDI, folosind alocare dinamică pentru reprezentarea înlăntuirilor).

# Concluzii - Liste

- Memorarea elementelor listei **secvențial** într-un tablou unidimensional (vector).
  - eficientă pentru acele liste în care se fac multe operații de adăugare la sfârșit, accesare și mai puține inserări.
  - dacă se folosește un tablou static, deficiența este dată de gestionarea inefficientă a spațiului de memorare (este deseori necesar să se supraestimeze spațiul necesar memorării elementelor).
  - tabloul dinamic exclude dezavantajul tablourilor statice de stabilire statică a capacitatei maxime a unei liste, dar totuși rămâne dezavantajul dat de ineficiența operațiilor de inserare și ștergere a elementelor din interiorul listei. Inserările și ștergerile, într-o astfel de listă, se fac dificil deoarece necesită deplasări ale elementelor.
- Reprezentarea **înlănțuită**.
  - spațiu adițional pentru memorarea legăturilor - ceea ce conduce la creșterea complexității-spațiu
  - gestionarea memoriei se face mai eficient
  - operațiile de inserare și ștergere se pot face mult mai eficient.
- Decizia asupra alegerii modului de implementare a unei liste depinde de gradul de dinamicitate al listei și de tipul aplicațiilor în care urmează a fi folosită:
  - Dacă actualizările (inserări, ștergeri) sunt rare, este preferată reprezentarea folosind tablouri.
  - Dacă actualizările sunt dese, este preferată reprezentarea înlănțuită.
- În funcție de restricțiile de acces și actualizare a elementelor unei liste, există diferite specializări ale listelor: *stive*, *cozi*, *cozi complete*, liste liniare generalizate.

## PROBLEME - LISTE

- Într-o bibliotecă sunt mai multe teancuri de cărți. Bibliotecarul vrea să rețină doar cărțile având anul de apariție mai mare decât 1970, dar în ordine alfabetică a titlurilor. Să se afișeze cărțile în ordinea în care trebuie reținute pe raft. **Indicație:** Stivă, CoadăCuPriorități.
- Jocul Gâsca Roșie.** 2 jucători primesc inițial  $\frac{n}{2}$  cărți de joc (fiecare carte poate avea culoarea roșie sau neagră). Jucătorii pun alternativ câte o carte pe masă (din vârful teancului lor de cărți), până se pune o carte roșie (caz în care teancul de pe masă va fi luat de către jucătorul care nu a pus cartea roșie și adăugat sub teancul său de cărți). Pierde jucătorul care nu mai are cărți. Simulați jocul. **Indicație:** Stivă, Coadă.
- Fie un labirint (rețea dreptunghiulară) cu celule ocupate (X) și libere (\*). Fie R un robot în acest labirint. Robotul se poate deplasa în 4 direcții: N, S, E, V.

*	*	X	*	*
X	*	*	*	X
X	*	R	*	*
X	*	X	X	*
*	*	*	*	*

- Testați dacă R poate ieși din labirint (poate ajunge la margine).
- Determinați un drum pentru ieșire (dacă există).

### Indicatie

Fie  $T$  mulțimea pozițiilor în care robotul poate ajunge pornind de la poziția inițială. Notăm cu  $S$  mulțimea pozițiilor în care robotul a ajuns până la un moment dat și din care s-ar putea deplasa. Un algoritm pentru determinarea mulțimilor  $T$  și  $S$  ar putea fi:

$$T \leftarrow \{\text{poziția inițială}\}$$

$$S \leftarrow \{\text{poziția inițială}\}$$

Cât timp  $S \neq \emptyset$  execută

Fie  $p$  un element din  $S$

$$S \leftarrow S \setminus \{p\}$$

Pentru fiecare poziție  $q$  alăturată poziției  $p$ ,  $q \neq 'X'$  și  $q \notin T$  execută

$$S \leftarrow S \cup \{q\}$$

$$T \leftarrow T \cup \{q\}$$

SfPentru

SfCatTimp

### Observații

- Pentru a răspunde la punctul a), algoritmul s-ar putea termina dacă poziția  $q$  care satisfac condițiile este pe frontieră labirintului

- Structura  $S$  poate fi o Stivă sau o Coadă (pentru a răspunde la a) și b)), Coadă (pentru a răspunde la c))
  - Mulțimea  $T$  poate fi memorată printr-o matrice asociată labirintului (ex: 0 pentru pozițiile neatinsene încă, respectiv 1 pentru pozițiile în care robotul a ajuns).
  - Pentru a răspunde la punctul b), ne putem gândi la un algoritm care pornind de la o poziție de pe frontieră (în cazul în care răspunsul alături punctul a) a fost pozitiv) merge din aproape în aproape pe pozițiile marcate cu 1 (atinse) spre poziția inițială.
4. Se dă un text care conține caractere incluzând paranteze rotunde, paranteze drepte și accolade. Se cere să se verifice dacă în text parantezele se închid corect. De exemplu, în textul `{a= (2 + b[3])*5;}` parantezele se închid corect; în textul `{ a = (b[0] . 1]; }` parantezele nu se închid corect. **Indicație.** Se va folosi **Stivă**.
  5. Scrieți 4 proceduri cu timpul de execuție  $\Theta(1)$  pentru inserare elemente și ștergere de elemente la ambele capete ale unei cozi duble (complete).
  6. Arătați cum se poate implementa o coadă prin 2 stive. Scrieți în Pseudocod operațiile cozii folosind doar operațiile din interfața Stivei. Analizați timpul de execuție al operațiilor cozii.
  7. Arătați cum se poate implementa o stivă prin 2 cozii. Scrieți în Pseudocod operațiile cozii folosind doar operațiile din interfața Cozii. Analizați timpul de execuție al operațiilor stivei.

# TAD CoadaCuPrioritati (PRIORITY QUEUE)

## Observații:

1. O coadă cu priorități este un container în care fiecare element are asociat o anumită prioritate.
2. Prințipiu de gestionare a unei cozi cu priorități este “HPF - Highest Priority First”.
3. **Accesul** într-o coadă cu priorități este *prespecificat* (se poate accesa doar elementul cel mai prioritar - prioritate maximă/minimă). Pentru generalitate, se poate considera o relație de ordine  $\mathcal{R}$  între priorități ( $\mathcal{R} : TPrioritate \times TPrioritate$ ,  $\mathcal{R}$  relație de ordine). În cazul general, vom considera că se accesează elementul ‘cel mai prioritar’ în raport cu relația de ordine  $\mathcal{R}$  (de ex. dacă se dorește accesul la elementul de prioritate maximă, atunci  $\mathcal{R} = “\geq”$ ). Într-o coadă cu priorități se **sterge** elementul ‘cel mai prioritar’ (în raport cu relația de ordine  $\mathcal{R}$ ). **Inserarea** unui element se va face astfel încât să se păstreze relația de ordine între priorități.
4. Se poate considera și o capacitate inițială a cozii cu priorități (număr maxim de elemente pe care le poate include), caz în care dacă numărul efectiv de elemente atinge capacitatea maximă, spunem că avem o *coadă cu priorități plină*.
5. Cozile cu priorități sunt mai simple decât dicționarele (se restricționează accesul).
6. O coadă cu priorități în general nu se iterează.
7. Aplicații
  - simularea unor sisteme complexe (aeroport) - sunt mai multe evenimente care au loc la un anumit moment de timp  $t$ . Simularea se va face printr-o coadă cu priorități ( $t$  este prioritatea) - accesul este la elementul având prioritate *minimă*.
  - probleme de concurență a proceselor.

## Tipul Abstract de Date COADA CU PRIORITATI:

## domeniu:

$\mathcal{CP} = \{cp \mid cp \text{ este o coadă cu priorități cu elemente } (e, p) \text{ de tip } TElement \times TPrioritate\}$

## operări:

- **creeaza**( $cp, \mathcal{R}$ )
 

{creează o coadă cu priorități vidă}

*pre :*   $\mathcal{R} : TPrioritate \times TPrioritate$ ,  $\mathcal{R}$  relație de ordine  
*post :*   $cp \in \mathcal{CP}, cp = \Phi$
- **adauga**( $cp, e, p$ )
 

{se adaugă un element în coada cu priorități}

*pre :*   $cp \in \mathcal{CP}, e \in TElement, p \in TPrioritate$   
*post :*   $cp' \in \mathcal{CP}, cp' = cp \oplus (e, p)$
- **sterge**( $cp, e, p$ )
 

{se sterge elementul 'cel mai prioritar'}

*pre :*   $cp \in \mathcal{CP}, cp \neq \Phi$   
*post :*   $e \in TElement, p \in TPrioritate$ ,  $e$  este 'cel mai prioritar' element din  $cp$   
 $p$  e prioritatea sa,  $cp' \in \mathcal{CP}, cp' = cp \ominus (e, p)$

© aruncă excepție dacă coada cu priorități e vidă
- **element**( $cp, e, p$ )
 

{se accesează elementul 'cel mai prioritar'}

*pre :*   $cp \in \mathcal{CP}, cp \neq \Phi$   
*post :*   $cp' = cp, e \in TElement, p \in TPrioritate$   
 $e$  este 'cel mai prioritar' element din  $cp$ ,  $p$  e prioritatea sa

© aruncă excepție dacă coada cu priorități e vidă
- **vida** ( $cp$ )
 

*pre :*   $cp \in \mathcal{CP}$   
*post :*   $vida = \begin{cases} adev, & \text{dacă } cp = \Phi \\ fals, & \text{dacă } cp \neq \Phi \end{cases}$
- **plina** ( $cp$ )
 

*pre :*   $cp \in \mathcal{CP}$   
*post :*   $plina = \begin{cases} adev, & \text{dacă } cp \text{ e plină} \\ fals, & \text{contrar} \end{cases}$
- **distruge**( $cp$ )
 

{destructor}

*pre :*   $cp \in \mathcal{CP}$   
*post :*   $cp$  a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)

## Observații

- CP nu este potrivită pentru aplicațiile care necesită traversarea ei (nu avem acces direct la elementele din interiorul cozii).
- Afisarea conținutului CP poate fi realizată (ca și la o **coadă** simplă) folosind o CP auxiliară. Complexitatea timp a subalgoritmului **tiparire** (descriș mai jos) este  $\theta(n)$ ,  $n$  fiind numărul de elemente din CP.

**Subalgoritm tiparire**( $cp$ )  
 {*pre: cp este o CP*}  
 {*post: se tipăresc elementele din cp*}

```

creeaza(cpAux) {se creează o CP auxiliară vidă}
{se sterg elementele din cp și se adaugă în cpAux}
CatTimp ⊢ vida(cp) executa
    sterge(cp, e)
    @ tipărește e
    adauga(cpAux, e)
SfCatTimp
{se sterg elementele din cpAux și se refac cp}
CatTimp ⊢ vida(cpAux) executa
    sterge(cpAux, e)
    adauga(cp, e)
SfCatTimp
SfSubalgoritm

```

### Implementări ale cozilor cu priorități folosind

- o listă sortată (ordonată) în raport cu prioritatea elementelor
  - reprezentare secvențială pe tablou (vector dinamic) .
  - reprezentare înlănțuită (simplu/dublu, reprezentare înlănțuiri folosind alocare dinamică/allocare statică).
- **ansamblu (heap)** (va fi discutat în Cursul 7).

### Observație:

- *Ansamblul* este cea mai potrivită SD pentru implementarea unei CP, deoarece oferă o complexitate  $O(\log_2 n)$  pentru operațiile specifice (adăugare, stergere) și  $\theta(1)$  pentru operația de accesare.

### Aplicație

**Acordarea burselor pentru studenți.** Să se simuleze procesul de acordare a burselor pentru studenți. Sunt  $n$  studenți, se acordă  $m$  burse. Bursele se acordă în ordine descrescătoare a mediilor.

# Liste

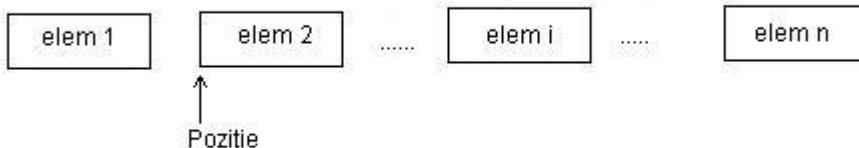
- In limbajul uzual cuvântul “listă” referă o “înşirare, într-o anumită *ordine*, a unor nume de persoane sau de obiecte, a unor date etc.”
- Exemple de liste sunt multiple: listă de cumpărături, listă de prețuri, listă de studenți, etc.
  - *Ordinea* în listă poate fi interpretată
    - ca un fel de „legătură” între elementele listei: după prima cumpărătură urmează a doua cumpărătură, după a doua cumpărătură urmează a treia cumpărătură, etc)
    - sau
    - poate fi văzută ca fiind dată de numărul de ordine al elementului în listă (1-a cumpărătură, a 2-a cumpărătură, etc).
  - Tipul de date Listă care va fi definit în continuare permite implementarea în aplicații a acestor situații din lumea reală.
- O *listă* o putem vedea ca pe o secvență de elemente  $\langle l_1, l_2, \dots, l_n \rangle$  de un același tip (TElement), fiecare element având o *poziție* bine determinată în cadrul listei, existând o ordine între pozițiile elementelor în cadrul listei
  - Lista poate fi văzută ca o colecție dinamică de elemente în care este esențială ordinea elementelor.
  - Numărul  $n$  de elemente din listă se numește *lungimea* listei.
  - O listă de lungime 0 se va numi lista *vidă*.
  - Caracterul de dinamicitate al listei este dat de faptul că lista își poate modifica în timp lungimea prin adăugări și ștergeri de elemente în/din listă.
- *Pozitia* elementelor în cadrul listei este esențială, astfel accesul, ștergerea și adăugarea se pot face pe orice *poziție* în listă.
- **Lista liniară**
  - o structură care fie este vidă (nu are nici un element), fie
    - are un unic prim element;
    - are un unic ultim element;
    - fiecare element din listă (cu excepția ultimului element) are un singur succesor;
    - fiecare element din listă (cu excepția primului element) are un singur predecesor.
  - Într-o listă liniară se pot insera elemente, șterge elemente, se poate determina succesorul (predecesorul) unui element aflat pe o anumită *poziție*, se poate accesa un element pe baza *poziției* sale în listă.

- Fiecare element al unei liste liniare are o *poziție* bine determinată în cadrul listei.
  - este importantă prima *poziție* în cadrul listei
  - *poziția* unui element este relativă la listă
  - *poziția* unui element din listă
    - identifică elementul din listă
    - poziția elementului predecesor și poziția elementului successor în listă (dacă acestea există)
  - ordine între pozițiile elementelor în cadrul listei.

**Poziția** unui element în cadrul listei poate fi văzută în diferite moduri:

1. ca fiind dată de **rangul** (numărul de ordine al) elementului în cadrul listei.
  - similitudine cu tablourile, *poziția* unui element în listă fiind *indexul* acestuia în cadrul listei.
  - Într-o astfel de abordare, lista este văzută ca un tablou dinamic în care se pot accesa/adăuga/șterge elemente pe orice poziție în listă.
2. ca fiind dată de o **referință** la locația unde se stochează elementul listei (ex: pointer spre locația unde se memorează elementul).

Pentru a asigura generalitatea, vom abstractiza noțiunea de *poziție* a unui element în listă și vom presupune că elementele listei sunt accesate prin intermediul unei *poziții* generice.



- **O Poziție** p într-o listă o considerăm **validă** dacă este poziția unui element al listei.
  - dacă p ar fi un pointer spre locația unde se memorează un element al listei, atunci p este **valid** dacă este diferit de pointerul nul sau de orice altă adresă care nu reprezintă adresa de memorare a unui element al listei.
  - dacă p ar fi rangul (numărul de ordine al) elementului în listă, atunci p este **valid** dacă e cuprins între 1 și numărul de elemente din listă.

# TAD Lista (LIST)

## Observații:

1. Tipul (abstract) de date *TPozitie* abstractizează noțiunea de poziție a unui element în listă (pentru a se asigura generalitatea).
2. O poziție  $p \in TPozitie$  din lista  $l$  o numim *poziție validă* dacă este poziția unui element din lista  $l$ .
3. În domeniul de valori a *TPozitie*, notată cu  $\perp$  o valoare specială p car o vom numi poziție nedefinită. Poziția nedefinită  $\perp$  nu este o poziție *validă* (conform celor menționate anterior).
4. Lista vidă o notăm cu  $\Phi$ .

## Tipul Abstract de Date LISTA:

### domeniu:

$$\mathcal{L} = \{l \mid l \text{ este o listă cu elemente de tip } TElement, \text{ fiecare element având o poziție unică în } l \text{ de tip } TPozitie\}$$

### operări:

- **creeaza( $l$ )**  
  {creează o listă vidă}  
    *pre* : true  
    *post* :  $l \in L, l = \Phi$
- **prim( $l$ )**  
    *pre* :  $l \in L$   
    *post* :  $prim = p \in TPozitie,$   
          
$$p = \begin{cases} \text{poziția primului element din lista } l, & \text{dacă } l \neq \Phi \\ \perp, & \text{dacă } l = \Phi \end{cases}$$
- **ultim ( $l$ )**  
    *pre* :  $l \in L$   
    *post* :  $ultim = p \in TPozitie,$   
          
$$p = \begin{cases} \text{poziția ultimului element din lista } l, & \text{dacă } l \neq \Phi \\ \perp, & \text{dacă } l = \Phi \end{cases}$$
- **valid( $l, p$ )**  
    *pre* :  $l \in L, p \in TPozitie$   
    *post* :  $valid = \begin{cases} \text{true}, & \text{dacă } p \text{ este o poziție a unui element din lista } l \\ \text{false}, & \text{altfel} \end{cases}$
- **următor( $l, p$ )**  
    *pre* :  $l \in L, p \in TPozitie, p \text{ poziție validă}$   
    *post* :  $urmator = q \in TPozitie,$   
          
$$q = \begin{cases} \text{poziția următoare poziției } p \text{ din lista } l, & \text{dacă } p \text{ nu e poziția ultimului element din lista } l \\ \perp, & \text{dacă } p \text{ e poziția ultimului element din lista } l \end{cases}$$

© aruncă excepție dacă  $p$  nu e validă

- **anterior( $l, p$ )**

*pre :*  $l \in L, p \in TPozitie, p$  poziție validă  
*post :*  $anterior = q \in TPozitie,$

$$q = \begin{cases} \text{poziția precedentă poziției } p \text{ din lista } l, \\ \quad \text{dacă } p \text{ nu e poziția primului element din lista } l \\ \perp, \quad \text{dacă } p \text{ e poziția primului element din lista } l \end{cases}$$

© aruncă excepție dacă  $p$  nu e validă
- **element( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p)$   
*post :*  $e \in TElement, e = \text{elementul de pe poziția } p \text{ din } l$

© aruncă excepție dacă  $p$  nu e validă
- **pozitie ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$ ,  
*post :*  $pozitie = p \in TPozitie,$

$$p = \begin{cases} \text{prima poziție a elementului } e \text{ din lista } l, & \text{dacă } e \in l \\ \perp, & \text{dacă } e \notin l \end{cases}$$
- **modifică( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul de pe poziția  $p$  din  $l' = e$

© aruncă excepție dacă  $p$  nu e validă
- **adaugalnceput ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :* elementul  $e$  a fost adăugat la începutul listei  $l$   
 $(l' = e \oplus l)$
- **adaugaSfarsit( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :* elementul  $e$  a fost adăugat la sfârșitul listei  $l$   
 $(l' = l \oplus e)$
- **adaugaDupa( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul  $e$  a fost inserat în lista  $l$  după poziția  $p$ ,  
 $\text{pozitie}(l', e) = \text{urmator}(l', p)$

© aruncă excepție dacă  $p$  nu e validă
- **adaugalnainte( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul  $e$  a fost inserat în lista  $l$  înaintea poziției  $p$ ,  
 $\text{pozitie}(l', e) = \text{anterior}(l', p)$

© aruncă excepție dacă  $p$  nu e validă
- **sterge ( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p)$   
*post :*  $e \in TElement, \text{elementul } e \text{ de pe poziția } p \text{ a fost șters din } l$

© aruncă excepție dacă  $p$  nu e validă
- **cauta ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :*  $cauta = \begin{cases} \text{adevarat}, & \text{dacă } e \text{ a fost găsit în lista } l \\ fals, & \text{altfel} \end{cases}$

- **vida** ( $l$ )

*pre* :  $l \in L$

*post* :  $\text{vida} = \begin{cases} \text{true}, & \text{dacă } l = \Phi \\ \text{false}, & \text{dacă } l \neq \Phi \end{cases}$

- **dim**( $l$ )

*pre* :  $l \in L$

*post* :  $\text{dim} = n \in \text{Natural},$

$n = \text{numărul de elemente ale listei } l$

- **distruge**( $l$ )

{destructor}

*pre* :  $l \in L$

*post* :  $l$  a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)

- **iterator**( $l, i$ )

*pre* :  $l \in L$

*post* :  $i \in \mathcal{I}, i$  este un iterator pe lista  $l$

## Observații

- Operația **cauta** poate fi specificată mai general

- returnează prima *poziție* pe care apare un element în listă, dacă elementul e găsit în listă
- returnează *poziție invalidă* dacă elementul nu e găsit în listă

- Din perspectiva unei ierarhii de containere

- **Lista** este o **Colecție**

- **Vector Dinamic** este o **Listă**

- \* **Vectorul Dinamic** poate fi văzut ca o **Listă** reprezentată *secvențial*

- Există anumite dezavantaje induse de folosirea unui parametru de tip *TPozitie* în interfața listei:

1. Tipurile de referințe concrete folosite diferă în funcție de reprezentarea listei.
2. Interfața listei este destul de greoaie și nesigură prin faptul că expune în exterior pozițiile (referințele la locațiile din listă).
  - acesta este motivul pentru care bibliotecile existente particularizează tipul *TPozitie* expus în interfața containerului Listă (după cum se va vedea în continuare)

## Implementări ale containerului **Lista** în biblioteci existente:

### 1. **STL - list**

- *poziția* este dată de un *iterator* pe listă  $\Rightarrow \text{TPozitie} = \text{Iterator}$ .
- în STL, *list* e văzut ca și un container de tip *secvențial*: elementele sunt aranjate într-o ordine (liniară) strictă.
- reprezentarea este dublu *înlănțuită*
  - dacă se dorește reprezentare simplu *înlănțuită*, se va folosi **forward\_list**.

- dacă se dorește reprezentare *secvențială*, se va folosi **vector**.

## 2. Java - List

- *poziția* este văzută ca un indice  $\Rightarrow TPozitie = Intreg$ .
  - permite accesarea elementelor din listă prin intermediul indicilor (ca la reprezentarea secvențială - **Vector Dinamic**)
- dacă se dorește reprezentare *înlănțuită* a listei, se va folosi **Linked List**.

## Modalități de implementare a unei liste

- memorând elementele sale **secvențial** într-un tablou/vector (dinamic)
  - accesul la elementele listei este *direct* ( $\theta(1)$ )
- memorând elementele sale **înlănțuit** într-o listă înlănțuită
  - accesul la elementele listei este *secvențial* ( $O(n)$ )
  - lista înlănțuită poate fi
    - \* simplu înlănțuită (LSI)
    - \* dublu înlănțuită (LDI)

## Analiza complexității timp a celor mai importante operații ale containerului Lista în funcție de implementarea acesteia

În Tabelul 1 vom considera, comparativ

- reprezentare secvențială folosind un vector dinamic (poziția este indice);
- reprezentare simplu înlănțuită (LSI) cu alocare dinamică (poziția este adresa de memorare a unui nod);
- reprezentare dublu înlănțuită (LDI) cu alocare dinamică (poziția este adresa de memorare a unui nod);

Notăm cu  $n$  numărul de elemente din listă. Observăm faptul că reprezentarea dublu înlănțuită este cea mai eficientă ca și timp, dar ocupă spațiu de memorare suplimentar pentru legături (pentru a reduce spațiul de memorare se pot folosi liste de tip XOR - a se vedea cursul 4).

Vom particulariza, în cele ce urmează, TAD-ul generic **Lista**, atfel încât să regăsim cele două specificații ale containerului **Lista** descrise anterior(STL/Java).

## Lista - cu poziție indice (indexată)

- corespunde modului în care este specificată lista în Java.
- *poziția* este văzută ca un indice  $\Rightarrow TPozitie = Intreg$ .
  - permite accesarea elementelor prin intermediul indicilor
- Accesul la elemente se face pe baza rangului, se permit inserări și ștergeri la orice poziție (poziția unui element reprezintă indicele acestuia în cadrul listei).

Operatie	Reprezentare sevențială	Reprezentare folosind o LSI alocată dinamic	Reprezentare folosind o LDI alocată dinamic
creeaza	$\theta(1)$	$\theta(1)$	$\theta(1)$
prim	$\theta(1)$	$\theta(1)$	$\theta(1)$
ultim	$\theta(1)$	$\theta(1)$ - dacă memorăm ultim $O(n)$ - fără a memora ultim	$\theta(1)$
următor	$\theta(1)$	$\theta(1)$	$\theta(1)$
anterior	$\theta(1)$	$O(n)$	$\theta(1)$
adaugaInceput	$\theta(n)$	$\theta(1)$	$\theta(1)$
adaugaSfarsit	$\theta(1)$ amortizat	$\theta(1)$ - dacă memorăm ultim $\theta(n)$ - fără a memora ultim	$\theta(1)$
adaugaDupa	$O(n)$	$\theta(1)$	$\theta(1)$
adaugaInainte	$O(n)$	$O(n)$	$\theta(1)$
sterge	$O(n)$	$O(n)$	$\theta(1)$

Tabela 1: Complexități timp ale operațiilor.

- O poziție  $i$  în cadrul listei  $l$  este *validă* dacă  $1 \leq i \leq \text{lungime}(l)$ .
- Se simplifică interfața
  - interfața este aceeași cu a unui **Vector Dinamic**

Specificația Listei indexate este dată mai jos **domeniu**:

$$L = \{l \mid l = [e_1, e_2, \dots, e_n], e_i \in TElement \quad \forall i = 1, 2, \dots, n\}$$

**operații:**

- **creeaza** ( $l$ )

*pre* : true  
*post* :  $l \in L, l = \Phi$  lista vidă

- **adaugaSfarsit**( $l, e$ )

*pre* :  $l \in L, e \in TElement$   
*post* : elementul  $e$  a fost adăugat la sfârșitul listei  $l$   
 $(l' = l \oplus e)$

- **adauga** ( $l, i, e$ )

*pre* :  $l \in L, e \in TElement, i \in \text{Intreg},$   
 $i$  poziție validă în  $l \vee i = \text{lungime}(l) + 1$   
*post* :  $l' = (e_1, \dots, e_{i-1}, e, e_i, e_{i+1}, \dots, e_n)$   
 $(\text{pozitie}(l', e) = i)$

© aruncă excepție dacă  $i$  nu e valid

- **sterge** ( $l, i, e$ )

*pre* :  $l \in L, l = (e_1, \dots, e_{i-1}, e_i, e_{i+1}, \dots, e_n), i \in \text{Intreg}, i$  poziție validă  
*post* :  $e \in TElement, e =$  elementul de pe poziția  $i$  din  $l$   
 $l' = (e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n)$   
 $(\text{pozitie}(l', e) = i)$

© aruncă excepție dacă  $i$  nu e valid

- cauta ( $l, e$ )
 

*pre* :  $l \in L, e \in TElement$

*post* :  $cauta = \begin{cases} i, & \text{dacă } i \text{ e prima pozitie pe care } e \text{ a fost găsit în lista } l \\ -1, & e \notin L \end{cases}$
- element ( $l, i, e$ )
 

*pre* :  $l \in L, i \in Intreg, i$  poziție validă

*post* :  $e \in TElement, e = \text{elementul de pe poziția } i \text{ din } l$

© aruncă excepție dacă  $i$  nu e valid
- modifica ( $l, i, e$ )
 

*pre* :  $l \in L, i \in Intreg, i$  poziție validă,  $e \in TElement$

*post* : elementul de pe poziția  $i$  din  $l' = e$

© aruncă excepție dacă  $i$  nu e valid
- vida ( $l$ )
 

*pre* :  $l \in L$

*post* :  $vida = \begin{cases} true, & \text{dacă } l = \Phi \\ false, & \text{altfel} \end{cases}$
- dim ( $l$ )
 

*pre* :  $l \in L$

*post* :  $dim = n \in Intreg,$   
 $n = \text{numărul de elemente din lista } l$
- iterator( $l, i$ )
 

*pre* :  $l \in L$

*post* :  $i \in \mathcal{I}, i$  este un iterator pe lista  $l$
- distrugе( $l$ )
 

*pre* :  $l \in L$

*post* :  $l$  a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)

## Exemplu

Considerăm reprezentarea Listei indexate folosind o LSI alocată dinamic. Descriem mai jos, în Pseudocod, operația **element**.

Reprezentarea listei este

### Nod

e: TElement //informația utilă nodului  
 urm:  $\uparrow$  Nod //adresa la care e memorat următorul nod

### Lista

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

Subalgoritm element( $l, i, e$ )

```
{pre: l: Lista, i:Intreg, 1 ≤ i ≤ lungime(l), e:TElement }
{post: e este al i-lea element al listei }
{se parcurge până la al i-lea element }
p ← l.prim
{se parcurg i-1 legături }
Pentru i = 1, i - 1 executa
  p ← [p].urm
```

```

SfPentru
{p este al i-lea nod }
  e  $\leftarrow$  [p].e
SfSubalgoritm

```

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

Să considerăm subalgoritmul **tiparire** care tipărește elementele unei liste indexate reprezentate folosind o LSI alocată dinamic. Tipărirea trebuie realizată folosind iteratorul, în caz contrar, tipărirea se va realiza în timp pătratic în raport cu numărul de elemente din listă.

1. folosind un iterator: complexitate timp  $\theta(n)$ , *n* fiind numărul de elemente ale listei

```

Subalgoritm tiparire(l)
  {pre: l: Lista}
  {post: se tipăresc elementele listei}
  iterator(l, i)
  CatTimp valid(i) executa
    element(i, e)
    @tipărește e
    urmator(i)
  SfCatTimp
SfSubalgoritm

```

2. folosind accesul la elemente prin indici: complexitate timp  $\theta(n^2)$ , *n* fiind numărul de elemente ale listei

```

Subalgoritm tiparire(l)
  {pre: l: Lista}
  {post: se tipăresc elementele listei}
  Pentru i = 1, dim(l) executa
    element(l, i, e)
    @tipărește e
  SfPentru
SfSubalgoritm

```

## Lista - cu poziție iterator

- corespunde modului în care este specificată lista în STL.
- *poziția* este dată de un *iterator* pe listă  $\Rightarrow TPozitie = Iterator$ .
- se simplifică interfața
  - operațiile *următor*, *anterior*, *valid* și *element* sunt operațiile pe *iterator*

Enumerăm, mai jos, operațiile din interfața Listei în care accesul e pe baza unei poziții date de un iterator, fără a mai da specificația completă a operațiilor (specificațiile sunt cele indicate la containerul generic **Lista**, dar cu  $TPozitie = IteratorLista$ ).

### Operații din interfață:

- creeaza (*l* : Lista)
- vida (*l* : Lista)
- dim (*l* : Lista)
- IteratorLista prim(*l* : Lista)

- $TElement$  element( $l$  :Lista,  $poz$ :IteratorLista)
- $TElement$  modifica( $l$  :Listă,  $poz$ :IteratorLista,  $e$  :  $TElement$ )
- adaugaInceput( $l$  :Listă,  $e$  :  $TElement$ )
- adaugaSfarsit( $l$  :Listă,  $e$  :  $TElement$ )
- adauga ( $l$  :Listă,  $poz$ :IteratorListă,  $e$  :  $TElement$ )
- $TElement$  sterge( $l$  :Lista,  $poz$ :IteratorLista)
- IteratorLista cauta( $l$  :Lista,  $e$  :  $TElement$ )
- distrugе ( $l$  : Lista)

## Exemplu

Considerăm reprezentarea Listei cu poziție iterator, folosind o LDI alocată dinamic. Descriem mai jos, în Pseudocod, operația **adaugaDupa**.

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

### IteratorLista

l: Lista//referință către listă  
 curent: $\uparrow$  Nod//adresa nodului curent din listă

Pentru operația de adăugare, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă.

```
Functia creeazaNod( $l, e$ )
{pre:  $l$ : Lista,  $e$ : TElement}
{post: se returneză un  $\uparrow$  Nod conținând  $e$  ca informație utilă}
{se alocă un spațiu de memorare pentru un Nod }
{ $p$ :  $\uparrow$  Nod}
aloca( $p$ )
[ $p$ ].e  $\leftarrow e$ 
[ $p$ ].urm  $\leftarrow$  NIL
[ $p$ ].prec  $\leftarrow$  NIL
{rezultatul returnat de funcție}
creeazaNod  $\leftarrow p$ 
```

### SfFunctia

- Complexitate:  $\theta(1)$

### Subalgoritm adaugaDupa( $l, i, e$ )

```
{pre:  $l$ : Lista,  $i$ : IteratorLista,  $i$  este valid,  $e$ : TElement}
{post: se adaugă  $e$  după nodul curent al lui  $i$ }
nou  $\leftarrow$  creeazaNod( $l, e$ )
 $p \leftarrow i.curent$ 
{se va adaugă după  $p$  }
```

```

{dacă p este ultimul nod al listei }
Daca p = l.ultim atunci
  {p este diferit de NIL, din precondiție }
    [l.ultim].urm  $\leftarrow$  nou
    [nou].prec  $\leftarrow$  l.ultim
    {se actualizează l.ultim}
    l.ultim  $\leftarrow$  nou
  altfel
    {se adaugă între p și [p].urm}
    [nou].urm  $\leftarrow$  [p].urm
    [[p].urm].prec  $\leftarrow$  nou
    [p].urm  $\leftarrow$  nou
    [nou].prec  $\leftarrow$  p
SfDaca
SfSubalgoritm

```

- Complexitate:  $\theta(1)$

În directorul TAD Lista (pe pagina cursului, curs 5) găsiți implementarea parțială, în limbajul C++, a containerului **Lista** cu poziție iterator (repräsentarea este sub forma unei LDI, folosind alocare dinamică pentru reprezentarea înlănuirilor).

# Concluzii - Liste

- Memorarea elementelor listei **secvențial** într-un tablou unidimensional (vector).
  - eficientă pentru acele liste în care se fac multe operații de adăugare la sfârșit, accesare și mai puține inserări.
  - dacă se folosește un tablou static, deficiența este dată de gestionarea inefficientă a spațiului de memorare (este deseori necesar să se supraestimeze spațiul necesar memorării elementelor).
  - tabloul dinamic exclude dezavantajul tablourilor statice de stabilire statică a capacitatei maxime a unei liste, dar totuși rămâne dezavantajul dat de ineficiența operațiilor de inserare și ștergere a elementelor din interiorul listei. Inserările și ștergerile, într-o astfel de listă, se fac dificil deoarece necesită deplasări ale elementelor.
- Reprezentarea **înlănțuită**.
  - spațiu adițional pentru memorarea legăturilor - ceea ce conduce la creșterea complexității-spațiu
  - gestionarea memoriei se face mai eficient
  - operațiile de inserare și ștergere se pot face mult mai eficient.
- Decizia asupra alegerii modului de implementare a unei liste depinde de gradul de dinamicitate al listei și de tipul aplicațiilor în care urmează a fi folosită:
  - Dacă actualizările (inserări, ștergeri) sunt rare, este preferată reprezentarea folosind tablouri.
  - Dacă actualizările sunt dese, este preferată reprezentarea înlănțuită.
- În funcție de restricțiile de acces și actualizare a elementelor unei liste, există diferite specializări ale listelor: *stive*, *cozi*, *cozi complete*, liste liniare generalizate.

# Liste

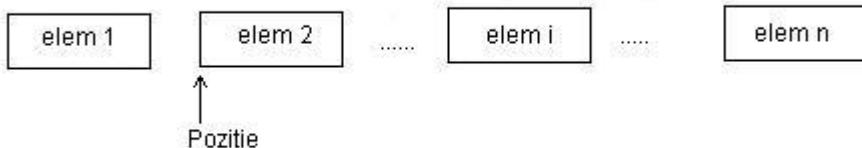
- In limbajul uzual cuvântul “listă” referă o “înşirare, într-o anumită *ordine*, a unor nume de persoane sau de obiecte, a unor date etc.”
- Exemple de liste sunt multiple: listă de cumpărături, listă de prețuri, listă de studenți, etc.
  - *Ordinea* în listă poate fi interpretată
    - ca un fel de „legătură” între elementele listei: după prima cumpărătură urmează a doua cumpărătură, după a doua cumpărătură urmează a treia cumpărătură, etc)
    - sau
    - poate fi văzută ca fiind dată de numărul de ordine al elementului în listă (1-a cumpărătură, a 2-a cumpărătură, etc).
  - Tipul de date Listă care va fi definit în continuare permite implementarea în aplicații a acestor situații din lumea reală.
- O *listă* o putem vedea ca pe o secvență de elemente  $\langle l_1, l_2, \dots, l_n \rangle$  de un același tip (TElement), fiecare element având o *poziție* bine determinată în cadrul listei, existând o ordine între pozițiile elementelor în cadrul listei
  - Lista poate fi văzută ca o colecție dinamică de elemente în care este esențială ordinea elementelor.
  - Numărul  $n$  de elemente din listă se numește *lungimea* listei.
  - O listă de lungime 0 se va numi lista *vidă*.
  - Caracterul de dinamicitate al listei este dat de faptul că lista își poate modifica în timp lungimea prin adăugări și ștergeri de elemente în/din listă.
- *Pozitia* elementelor în cadrul listei este esențială, astfel accesul, ștergerea și adăugarea se pot face pe orice *poziție* în listă.
- **Lista liniară**
  - o structură care fie este vidă (nu are nici un element), fie
    - are un unic prim element;
    - are un unic ultim element;
    - fiecare element din listă (cu excepția ultimului element) are un singur succesor;
    - fiecare element din listă (cu excepția primului element) are un singur predecesor.
  - Într-o listă liniară se pot insera elemente, șterge elemente, se poate determina succesorul (predecesorul) unui element aflat pe o anumită *poziție*, se poate accesa un element pe baza *poziției* sale în listă.

- Fiecare element al unei listei liniare are o *poziție* bine determinată în cadrul listei.
  - este importantă prima *poziție* în cadrul listei
  - *poziția* unui element este relativă la listă
  - *poziția* unui element din listă
    - identifică elementul din listă
    - poziția elementului predecesor și poziția elementului successor în listă (dacă acestea există)
  - ordine între pozițiile elementelor în cadrul listei.

**Poziția** unui element în cadrul listei poate fi văzută în diferite moduri:

1. ca fiind dată de **rangul** (numărul de ordine al) elementului în cadrul listei.
  - similitudine cu tablourile, *poziția* unui element în listă fiind *indexul* acestuia în cadrul listei.
  - Într-o astfel de abordare, lista este văzută ca un tablou dinamic în care se pot accesa/adăuga/șterge elemente pe orice poziție în listă.
2. ca fiind dată de o **referință** la locația unde se stochează elementul listei (ex: pointer spre locația unde se memorează elementul).

Pentru a asigura generalitatea, vom abstractiza noțiunea de *poziție* a unui element în listă și vom presupune că elementele listei sunt accesate prin intermediul unei *poziții* generice.



- **O Poziție**  $p$  într-o listă o considerăm **validă** dacă este poziția unui element al listei.
  - dacă  $p$  ar fi un pointer spre locația unde se memorează un element al listei, atunci  $p$  este **valid** dacă este diferit de pointerul nul sau de orice altă adresă care nu reprezintă adresa de memorare a unui element al listei.
  - dacă  $p$  ar fi rangul (numărul de ordine al) elementului în listă, atunci  $p$  este **valid** dacă e cuprins între 1 și numărul de elemente din listă.

# TAD Lista (LIST)

## Observații:

1. Tipul (abstract) de date *TPozitie* abstractizează noțiunea de poziție a unui element în listă (pentru a se asigura generalitatea).
2. O poziție  $p \in TPozitie$  din lista  $l$  o numim *poziție validă* dacă este poziția unui element din lista  $l$ .
3. În domeniul de valori a *TPozitie*, notată cu  $\perp$  o valoare specială p car o vom numi poziție nedefinită. Poziția nedefinită  $\perp$  nu este o poziție *validă* (conform celor menționate anterior).
4. Lista vidă o notăm cu  $\Phi$ .

## Tipul Abstract de Date LISTA:

### domeniu:

$$\mathcal{L} = \{l \mid l \text{ este o listă cu elemente de tip } TElement, \text{ fiecare element având o poziție unică în } l \text{ de tip } TPozitie\}$$

### operări:

- **creeaza( $l$ )**  
  {creează o listă vidă}  
    *pre* : true  
    *post* :  $l \in L, l = \Phi$
- **prim( $l$ )**  
    *pre* :  $l \in L$   
    *post* :  $prim = p \in TPozitie,$   
          
$$p = \begin{cases} \text{poziția primului element din lista } l, & \text{dacă } l \neq \Phi \\ \perp, & \text{dacă } l = \Phi \end{cases}$$
- **ultim ( $l$ )**  
    *pre* :  $l \in L$   
    *post* :  $ultim = p \in TPozitie,$   
          
$$p = \begin{cases} \text{poziția ultimului element din lista } l, & \text{dacă } l \neq \Phi \\ \perp, & \text{dacă } l = \Phi \end{cases}$$
- **valid( $l, p$ )**  
    *pre* :  $l \in L, p \in TPozitie$   
    *post* :  $valid = \begin{cases} \text{true}, & \text{dacă } p \text{ este o poziție a unui element din lista } l \\ \text{false}, & \text{altfel} \end{cases}$
- **următor( $l, p$ )**  
    *pre* :  $l \in L, p \in TPozitie, p \text{ poziție validă}$   
    *post* :  $urmator = q \in TPozitie,$   
          
$$q = \begin{cases} \text{poziția următoare poziției } p \text{ din lista } l, & \text{dacă } p \text{ nu e poziția ultimului element din lista } l \\ \perp, & \text{dacă } p \text{ e poziția ultimului element din lista } l \end{cases}$$

© aruncă excepție dacă  $p$  nu e validă

- **anterior( $l, p$ )**

*pre :*  $l \in L, p \in TPozitie, p$  poziție validă  
*post :*  $anterior = q \in TPozitie,$

$$q = \begin{cases} \text{poziția precedentă poziției } p \text{ din lista } l, \\ \quad \text{dacă } p \text{ nu e poziția primului element din lista } l \\ \perp, \quad \text{dacă } p \text{ e poziția primului element din lista } l \end{cases}$$

© aruncă excepție dacă  $p$  nu e validă
- **element( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p)$   
*post :*  $e \in TElement, e = \text{elementul de pe poziția } p \text{ din } l$

© aruncă excepție dacă  $p$  nu e validă
- **pozitie ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$ ,  
*post :*  $pozitie = p \in TPozitie,$

$$p = \begin{cases} \text{prima poziție a elementului } e \text{ din lista } l, & \text{dacă } e \in l \\ \perp, & \text{dacă } e \notin l \end{cases}$$
- **modifică( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul de pe poziția  $p$  din  $l' = e$

© aruncă excepție dacă  $p$  nu e validă
- **adaugalnceput ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :* elementul  $e$  a fost adăugat la începutul listei  $l$   
 $(l' = e \oplus l)$
- **adaugaSfarsit( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :* elementul  $e$  a fost adăugat la sfârșitul listei  $l$   
 $(l' = l \oplus e)$
- **adaugaDupa( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul  $e$  a fost inserat în lista  $l$  după poziția  $p$ ,  
 $\text{pozitie}(l', e) = \text{urmator}(l', p)$

© aruncă excepție dacă  $p$  nu e validă
- **adaugalnainte( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p), e \in TElement$   
*post :* elementul  $e$  a fost inserat în lista  $l$  înaintea poziției  $p$ ,  
 $\text{pozitie}(l', e) = \text{anterior}(l', p)$

© aruncă excepție dacă  $p$  nu e validă
- **sterge ( $l, p, e$ )**

*pre :*  $l \in L, p \in TPozitie, valid(p)$   
*post :*  $e \in TElement, \text{elementul } e \text{ de pe poziția } p \text{ a fost sters din } l$

© aruncă excepție dacă  $p$  nu e validă
- **cauta ( $l, e$ )**

*pre :*  $l \in L, e \in TElement$   
*post :*  $cauta = \begin{cases} \text{adevarat}, & \text{dacă } e \text{ a fost găsit în lista } l \\ fals, & \text{altfel} \end{cases}$

- **vida** ( $l$ )

*pre* :  $l \in L$

*post* :  $\text{vida} = \begin{cases} \text{true}, & \text{dacă } l = \Phi \\ \text{false}, & \text{dacă } l \neq \Phi \end{cases}$

- **dim**( $l$ )

*pre* :  $l \in L$

*post* :  $\text{dim} = n \in \text{Natural}$ ,

$n = \text{numărul de elemente ale listei } l$

- **distruge**( $l$ )

{destructor}

*pre* :  $l \in L$

*post* :  $l$  a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)

- **iterator**( $l, i$ )

*pre* :  $l \in L$

*post* :  $i \in \mathcal{I}, i$  este un iterator pe lista  $l$

## Observații

- Operația **cauta** poate fi specificată mai general

- returnează prima *poziție* pe care apare un element în listă, dacă elementul e găsit în listă
- returnează *poziție invalidă* dacă elementul nu e găsit în listă

- Din perspectiva unei ierarhii de containere

- **Lista** este o **Colecție**

- **Vector Dinamic** este o **Listă**

- \* **Vectorul Dinamic** poate fi văzut ca o **Listă** reprezentată *secvențial*

- Există anumite dezavantaje induse de folosirea unui parametru de tip *TPozitie* în interfața listei:

1. Tipurile de referințe concrete folosite diferă în funcție de reprezentarea listei.
2. Interfața listei este destul de greoaie și nesigură prin faptul că expune în exterior pozițiile (referințele la locațiile din listă).
  - acesta este motivul pentru care bibliotecile existente particularizează tipul *TPozitie* expus în interfața containerului Listă (după cum se va vedea în continuare)

Implementări ale containerului **Lista** în biblioteci existente:

### 1. STL - list

- *poziția* este dată de un *iterator* pe listă  $\Rightarrow \text{TPozitie} = \text{Iterator}$ .
- în STL, *list* e văzut ca și un container de tip *secvențial*: elementele sunt aranjate într-o ordine (liniară) strictă.
- reprezentarea este dublu *înlănțuită*
  - dacă se dorește reprezentare simplu *înlănțuită*, se va folosi **forward\_list**.

- dacă se dorește reprezentare *secvențială*, se va folosi **vector**.

## 2. Java - List

- *poziția* este văzută ca un indice  $\Rightarrow TPozitie = Intreg$ .
  - permite accesarea elementelor din listă prin intermediul indicilor (ca la reprezentarea secvențială - **Vector Dinamic**)
- dacă se dorește reprezentare *înlănțuită* a listei, se va folosi **Linked List**.

## Modalități de implementare a unei liste

- memorând elementele sale **secvențial** într-un tablou/vector (dinamic)
  - accesul la elementele listei este *direct* ( $\theta(1)$ )
- memorând elementele sale **înlănțuit** într-o listă înlănțuită
  - accesul la elementele listei este *secvențial* ( $O(n)$ )
  - lista înlănțuită poate fi
    - \* simplu înlănțuită (LSI)
    - \* dublu înlănțuită (LDI)

## Analiza complexității timp a celor mai importante operații ale containerului Lista în funcție de implementarea acesteia

În Tabelul 1 vom considera, comparativ

- reprezentare secvențială folosind un vector dinamic (poziția este indice);
- reprezentare simplu înlănțuită (LSI) cu alocare dinamică (poziția este adresa de memorare a unui nod);
- reprezentare dublu înlănțuită (LDI) cu alocare dinamică (poziția este adresa de memorare a unui nod);

Notăm cu  $n$  numărul de elemente din listă. Observăm faptul că reprezentarea dublu înlănțuită este cea mai eficientă ca și timp, dar ocupă spațiu de memorare suplimentar pentru legături (pentru a reduce spațiul de memorare se pot folosi liste de tip XOR - a se vedea cursul 4).

Vom particulariza, în cele ce urmează, TAD-ul generic **Lista**, atfel încât să regăsim cele două specificații ale containerului **Lista** descrise anterior(STL/Java).

## Lista - cu poziție indice (indexată)

- corespunde modului în care este specificată lista în Java.
- *poziția* este văzută ca un indice  $\Rightarrow TPozitie = Intreg$ .
  - permite accesarea elementelor prin intermediul indicilor
- Accesul la elemente se face pe baza rangului, se permit inserări și ștergeri la orice poziție (poziția unui element reprezintă indicele acestuia în cadrul listei).

Operatie	Reprezentare secvențială	Reprezentare folosind o LSI alocată dinamic	Reprezentare folosind o LDI alocată dinamic
creeaza	$\theta(1)$	$\theta(1)$	$\theta(1)$
prim	$\theta(1)$	$\theta(1)$	$\theta(1)$
ultim	$\theta(1)$	$\theta(1)$ - dacă memorăm ultim $O(n)$ - fără a memora ultim	$\theta(1)$
următor	$\theta(1)$	$\theta(1)$	$\theta(1)$
anterior	$\theta(1)$	$O(n)$	$\theta(1)$
adaugaInceput	$\theta(n)$	$\theta(1)$	$\theta(1)$
adaugaSfarsit	$\theta(1)$ amortizat	$\theta(1)$ - dacă memorăm ultim $\theta(n)$ - fără a memora ultim	$\theta(1)$
adaugaDupa	$O(n)$	$\theta(1)$	$\theta(1)$
adaugaInainte	$O(n)$	$O(n)$	$\theta(1)$
sterge	$O(n)$	$O(n)$	$\theta(1)$

Tabela 1: Complexități timp ale operațiilor.

- O poziție  $i$  în cadrul listei  $l$  este *validă* dacă  $1 \leq i \leq \text{lungime}(l)$ .
- Se simplifică interfața
  - interfața este aceeași cu a unui **Vector Dinamic**

Specificația Listei indexate este dată mai jos

**domeniu:**

$$L = \{l \mid l = [e_1, e_2, \dots, e_n], e_i \in TElement \ \forall i = 1, 2, \dots, n\}$$

**operații:**

- **creeaza** ( $l$ )

*pre* : true  
*post* :  $l \in L, l = \Phi$  lista vidă

- **adaugaSfarsit**( $l, e$ )

*pre* :  $l \in L, e \in TElement$   
*post* : elementul  $e$  a fost adăugat la sfârșitul listei  $l$   
 $(l' = l \oplus e)$

- **adauga** ( $l, i, e$ )

*pre* :  $l \in L, e \in TElement, i \in \text{Intreg},$   
 $i$  poziție validă în  $l \vee i = \text{lungime}(l) + 1$   
*post* :  $l' = (e_1, \dots, e_{i-1}, e, e_i, e_{i+1}, \dots, e_n)$   
 $(\text{pozitie}(l', e) = i)$

© aruncă excepție dacă  $i$  nu e valid

- **sterge** ( $l, i, e$ )

*pre* :  $l \in L, l = (e_1, \dots, e_{i-1}, e_i, e_{i+1}, \dots, e_n), i \in \text{Intreg}, i$  poziție validă  
*post* :  $e \in TElement, e =$  elementul de pe poziția  $i$  din  $l$   
 $l' = (e_1, \dots, e_{i-1}, e_{i+1}, \dots, e_n)$   
 $(\text{pozitie}(l', e) = i)$

© aruncă excepție dacă  $i$  nu e valid

- cauta ( $l, e$ )
 

*pre* :  $l \in L, e \in TElement$

*post* :  $cauta = \begin{cases} i, & \text{dacă } i \text{ e prima pozitie pe care } e \text{ a fost găsit în lista } l \\ -1, & e \notin L \end{cases}$
- element ( $l, i, e$ )
 

*pre* :  $l \in L, i \in Intreg, i$  poziție validă

*post* :  $e \in TElement, e = \text{elementul de pe poziția } i \text{ din } l$

© aruncă excepție dacă  $i$  nu e valid
- modifica ( $l, i, e$ )
 

*pre* :  $l \in L, i \in Intreg, i$  poziție validă,  $e \in TElement$

*post* : elementul de pe poziția  $i$  din  $l' = e$

© aruncă excepție dacă  $i$  nu e valid
- vida ( $l$ )
 

*pre* :  $l \in L$

*post* :  $vida = \begin{cases} true, & \text{dacă } l = \Phi \\ false, & \text{altfel} \end{cases}$
- dim ( $l$ )
 

*pre* :  $l \in L$

*post* :  $dim = n \in Intreg,$   
 $n = \text{numărul de elemente din lista } l$
- iterator( $l, i$ )
 

*pre* :  $l \in L$

*post* :  $i \in \mathcal{I}, i$  este un iterator pe lista  $l$
- distrugе( $l$ )
 

*pre* :  $l \in L$

*post* :  $l$  a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)

## Exemplu

Considerăm reprezentarea Listei indexate folosind o LSI alocată dinamic. Descriem mai jos, în Pseudocod, operația **element**.

Reprezentarea listei este

### Nod

e: TElement //informația utilă nodului  
 urm:  $\uparrow$  Nod //adresa la care e memorat următorul nod

### Lista

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

Subalgoritm element( $l, i, e$ )

```
{pre: l: Lista, i:Intreg, 1 ≤ i ≤ lungime(l), e:TElement }
{post: e este al i-lea element al listei }
{se parcurge până la al i-lea element }
p ← l.prim
{se parcurg i-1 legături }
Pentru  $i = 1, i - 1$  executa
  p ← [p].urm
```

```

SfPentru
{p este al i-lea nod }
  e  $\leftarrow$  [p].e
SfSubalgoritm

```

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

Să considerăm subalgoritmul **tiparire** care tipărește elementele unei liste indexate reprezentate folosind o LSI alocată dinamic. Tipărirea trebuie realizată folosind iteratorul, în caz contrar, tipărirea se va realiza în timp pătratic în raport cu numărul de elemente din listă.

1. folosind un iterator: complexitate timp  $\theta(n)$ , *n* fiind numărul de elemente ale listei

```

Subalgoritm tiparire(l)
  {pre: l: Lista}
  {post: se tipăresc elementele listei}
  iterator(l, i)
    CatTimp valid(i) executa
      element(i, e)
      @tipărește e
      urmator(i)
    SfCatTimp
SfSubalgoritm

```

2. folosind accesul la elemente prin indici: complexitate timp  $\theta(n^2)$ , *n* fiind numărul de elemente ale listei

```

Subalgoritm tiparire(l)
  {pre: l: Lista}
  {post: se tipăresc elementele listei}
  Pentru i = 1, dim(l) executa
    element(l, i, e)
    @tipărește e
  SfPentru
SfSubalgoritm

```

## **Lista - cu poziție iterator**

- corespunde modului în care este specificată lista în STL.
- *poziția* este dată de un *iterator* pe listă  $\Rightarrow TPozitie = Iterator$ .
- se simplifică interfața
  - operațiile *următor*, *anterior*, *valid* și *element* sunt operațiile pe *iterator*

Enumerăm, mai jos, operațiile din interfața Listei în care accesul e pe baza unei poziții date de un iterator, fără a mai da specificația completă a operațiilor (specificațiile sunt cele indicate la containerul generic **Lista**, dar cu  $TPozitie = IteratorLista$ ).

### **Operații din interfață:**

- creeaza (*l* : Lista)
- vida (*l* : Lista)
- dim (*l* : Lista)
- IteratorLista prim(*l* : Lista)

- $TElement$  element( $l$  :Lista,  $poz$ :IteratorLista)
- $TElement$  modifica( $l$  :Listă,  $poz$ :IteratorLista,  $e$  :  $TElement$ )
- adaugaInceput( $l$  :Listă,  $e$  :  $TElement$ )
- adaugaSfarsit( $l$  :Listă,  $e$  :  $TElement$ )
- adauga ( $l$  :Listă,  $poz$ :IteratorListă,  $e$  :  $TElement$ )
- $TElement$  sterge( $l$  :Lista,  $poz$ :IteratorLista)
- IteratorLista cauta( $l$  :Lista,  $e$  :  $TElement$ )
- distruge ( $l$  : Lista)

## Exemplu

Considerăm reprezentarea Listei cu poziție iterator, folosind o LDI alocată dinamic. Descriem mai jos, în Pseudocod, operația **adaugaDupa**.

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

### IteratorLista

l: Lista//referință către listă  
 curent: $\uparrow$  Nod//adresa nodului curent din listă

Pentru operația de adăugare, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă.

```
Functia creeazaNod( $l, e$ )
{pre:  $l$ : Lista,  $e$ : TElement}
{post: se returneză un  $\uparrow$  Nod conținând  $e$  ca informație utilă}
{se alocă un spațiu de memorare pentru un Nod }
{ $p$ :  $\uparrow$  Nod}
aloca( $p$ )
[ $p$ ].e  $\leftarrow e$ 
[ $p$ ].urm  $\leftarrow$  NIL
[ $p$ ].prec  $\leftarrow$  NIL
{rezultatul returnat de funcție}
creeazaNod  $\leftarrow p$ 
```

### SfFunctia

- Complexitate:  $\theta(1)$

### Subalgoritm adaugaDupa( $l, i, e$ )

```
{pre:  $l$ : Lista,  $i$ : IteratorLista,  $i$  este valid,  $e$ : TElement}
{post: se adaugă  $e$  după nodul curent al lui  $i$ }
nou  $\leftarrow$  creeazaNod( $l, e$ )
 $p \leftarrow i.curent$ 
{se va adaugă după  $p$  }
```

```

{dacă p este ultimul nod al listei }
Daca p = l.ultim atunci
  {p este diferit de NIL, din precondiție }
    [l.ultim].urm  $\leftarrow$  nou
    [nou].prec  $\leftarrow$  l.ultim
    {se actualizează l.ultim}
    l.ultim  $\leftarrow$  nou
  altfel
    {se adaugă între p și [p].urm}
    [nou].urm  $\leftarrow$  [p].urm
    [[p].urm].prec  $\leftarrow$  nou
    [p].urm  $\leftarrow$  nou
    [nou].prec  $\leftarrow$  p
SfDaca
SfSubalgoritm

```

- Complexitate:  $\theta(1)$

În directorul TAD Lista (Curs 5) găsiți implementarea parțială, în limbajul C++, a containerului **Lista cu poziție iterator** (repräsentarea este sub forma unei LDI, folosind alocare dinamică pentru reprezentarea înlăntuirilor).

# Concluzii - Liste

- Memorarea elementelor listei **secvențial** într-un tablou unidimensional (vector).
  - eficientă pentru acele liste în care se fac multe operații de adăugare la sfârșit, accesare și mai puține inserări.
  - dacă se folosește un tablou static, deficiența este dată de gestionarea inefficientă a spațiului de memorare (este deseori necesar să se supraestimeze spațiul necesar memorării elementelor).
  - tabloul dinamic exclude dezavantajul tablourilor statice de stabilire statică a capacitatei maxime a unei liste, dar totuși rămâne dezavantajul dat de ineficiența operațiilor de inserare și ștergere a elementelor din interiorul listei. Inserările și ștergerile, într-o astfel de listă, se fac dificil deoarece necesită deplasări ale elementelor.
- Reprezentarea **înlănțuită**.
  - spațiu adițional pentru memorarea legăturilor - ceea ce conduce la creșterea complexității-spațiu
  - gestionarea memoriei se face mai eficient
  - operațiile de inserare și ștergere se pot face mult mai eficient.
- Decizia asupra alegerii modului de implementare a unei liste depinde de gradul de dinamicitate al listei și de tipul aplicațiilor în care urmează a fi folosită:
  - Dacă actualizările (inserări, ștergeri) sunt rare, este preferată reprezentarea folosind tablouri.
  - Dacă actualizările sunt dese, este preferată reprezentarea înlănțuită.
- În funcție de restricțiile de acces și actualizare a elementelor unei liste, există diferite specializări ale listelor: *stive*, *cozi*, *cozi complete*, liste liniare generalizate.

# 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
      - **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 atfel încât să se păstreze relația de ordine dintre elementele listei.
        - adauga**(*lo*, *e*)  
{adăugă 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:
      - sterge** (*l*, *e*)
        - pre* : *l*  $\in$  *L*, *e*  $\in$  *TElement*
        - post* : prima apariție a elementului *e* a fost ștearsă 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  $\mathcal{R} \subseteq TComparabil \times TComparabil$ , 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:  $\uparrow \text{Nod} // \text{adresa la care e memorat nodul anterior}$

### Lista

prim:  $\uparrow \text{Nod} // \text{adresa primului nod din listă}$

ultim:  $\uparrow \text{Nod} // \text{adresa ultimului nod din listă}$

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

### IteratorLista

l: Lista//referință către listă

current:  $\uparrow \text{Nod} // \text{adresa nodului current din listă}$

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}

$lo.\text{prim} \leftarrow \text{NIL}$

$lo.\text{ultim} \leftarrow \text{NIL}$

{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 eşuarea 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**).

Functia 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.current ← NIL
{dacă e sigur că e nu apare în listă (caz 1 de mai sus)}
Daca (lo.prim = NIL) ∨ (¬ ([lo.prim].e  $\mathcal{R}$  e)) ∨ (¬ (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 ← lo.prim
    CatTimp (p ≠ NIL)  $\wedge$  ([p].e ≠ e)  $\wedge$  (¬ (e  $\mathcal{R}$  [p].e)) executa
        p ← [p].urm
    SfCatTimp
    {dacă am găsit elementul, setăm iteratorul pe nodul găsit)}
    Daca (p ≠ NIL)  $\wedge$  ([p].e = e) atunci
        it.current ← p
    SfDaca
    {returnăm iteratorul)}
    cauta ← it
SfDaca
SfFunctia

```

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

În directorul TAD Lista Ordonata (**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 înlățuirilor). **Atenție:** operația de adăugare nu e completă (e tratat doar cazul în care elementul trebuie adăugat la începutul listei).

# TAD Stiva (STACK)

## Observații:

1. În limbajul uzual cuvântul “stivă” referă o “grămadă în care elementele constitutive sunt așezate ordonat unele peste altele”.
  - Un element nou se adaugă în stivă deasupra elementului cel mai recent adăugat în stivă.
  - Din stivă se poate accesa și extrage doar elementul cel mai recent introdus.
  - Exemple de stive sunt multiple: stivă de farfurii, stivă de lemne, etc. Tipul de date **Stivă** permite implementarea în aplicații a acestor situații din lumea reală.
2. O *stivă* este o structură liniară de tip listă care restricționează adăugările și ștergerile la un singur capăt (listă LIFO - *Last In First Out*).
3. **Accesul** într-o stivă este *prespecificat* (se poate accesa doar elementul cel mai recent introdus în stivă - din vârful stivei), nu se permite accesul la elemente pe baza poziției. Dintr-o stivă se poate **șterge** elementul CEL MAI RECENT introdus în stivă - cel din vârful stivei.
4. Se poate considera și o capacitate inițială a stivei (număr maxim de elemente pe care le poate include), caz în care dacă numărul efectiv de elemente atinge capacitatea maximă, spunem că avem o *stivă plină*.
  - adăugarea în stiva plină se numește **depășire superioară**.
5. O stivă fără elemente o vom numi *stivă vidă* și o notăm  $\Phi$ .
  - ștergerea din stiva vidă se numește **depășire inferioară**.
6. O stivă în general nu se iterează.
7. Stivele sunt frecvent utilizate în programare - recursivitate, backtracking iterativ.

## Tipul Abstract de Date STIVA:

### domeniu:

$$\mathcal{S} = \{s \mid s \text{ este o stivă cu elemente de tip } TElement\}$$

### operații (interfață):

- **creeaza( $s$ )**  
    {creează o stivă vidă}  
        *pre : true*  
        *post :  $s \in \mathcal{S}, s = \Phi(\text{stiva vidă})$*

- **adauga**( $s, e$ )  
 {se adaugă un element în vârful stivei}
 

*pre :*   $s \in \mathcal{S}, e \in TElement, s$  nu e plină  
*post :*   $s' \in \mathcal{S}, s' = s \oplus e, e$  va fi cel mai recent element introdus în stivă

© aruncă excepție dacă stiva e plină
- **sterge**( $s, e$ )  
 {se scoate un element din vârful stivei}
 

*pre :*   $s \in \mathcal{S}, s \neq \Phi$   
*post :*   $e \in TElement, e$  este cel mai recent element introdus în stivă,  $s' \in \mathcal{S}, s' = s \ominus e$

© aruncă excepție dacă stiva e vidă
- **element**( $s, e$ )  
 {se accesează elementul din vârful stivei}
 

*pre :*   $s \in \mathcal{S}, s \neq \Phi$   
*post :*   $s' = s, e \in TElement, e$  este cel mai recent element introdus în stivă

© aruncă excepție dacă stiva e vidă
- **vida** ( $s$ )
 

*pre :*   $s \in \mathcal{S}$   
*post :*   $vida = \begin{cases} adev, & \text{dacă } s = \Phi \\ fals, & \text{dacă } s \neq \Phi \end{cases}$
- **plina** ( $s$ )
 

*pre :*   $s \in \mathcal{S}$   
*post :*   $plina = \begin{cases} adev, & \text{dacă } s \text{ e plină} \\ fals, & \text{contrar} \end{cases}$
- **distruge**( $s$ )
 

{destructor}

*pre :*   $s \in \mathcal{S}$   
*post :*   $s$  a fost 'distrusa' (spațiul de memorie alocat a fost eliberat)

## Observații

- Stiva nu este potrivită pentru aplicațiile care necesită traversarea ei (nu avem acces direct la elementele din interiorul stivei).
- Afisarea conținutului stivei poate fi realizată folosind o stivă auxiliară (scoatem valorile din stivă punându-le pe o stivă auxiliară, după care se repun pe stiva inițială). Complexitatea timp a subalgoritmului **tiparire** (descriș mai jos) este  $\theta(n)$ ,  $n$  fiind numărul de elemente din stivă.

Subalgoritm **tiparire**( $s$ )

{*pre: s este o Stivă*}

{*post: se tipăresc elementele din Stivă*}

**creeaza**( $sAux$ ) {*se creează o stivă auxiliară vidă*}

{*se șterg elementele din s și se adaugă în sAux*}

```

CatTimp ⊨ vida(s) executa
    sterge(s, e)
    adauga(sAux, e)
SfCatTimp
{se șterg elementele din sAux și se refac s}
CatTimp ⊨ vida(sAux) executa
    sterge(sAux, e)
    @ tipărește e
    adauga(s, e)
SfCatTimp
SfSubalgoritm

```

### **Implementări ale stivelor folosind**

- tablouri - vectori (dinamici)
- liste înláńtuite.

# Lista înlățuită cu reprezentarea înlățuirilor pe tablou

După cum am arătat în **Cursul 4**, există două modalități de a reprezenta înlățuirile pentru o listă înlățuită:

- folosind **alocare dinamică** (pointeri)
  - această modalitate de reprezentare a înlățuirilor a fost discutată în **Cursul 4**.
- folosind **tablouri** statice/dinamice
  - această modalitate de reprezentare a înlățuirilor o vom discuta în continuare.

## **1. Lista simplu înlățuită (LSI) - reprezentarea înlățuirilor pe tablou**

---

Într-o astfel de reprezentare, atât elementele, cât și legăturile se memorează folosind două tablouri. Presupunem ca *Max* este capacitatea maximă de memorare a tabloului. Locațiile se consideră a fi numerotate de la 1 la *Max*.

- **elementele** (informațiile utile din nodurile listei) se memorează în tabloul **e[1..Max]** (indexarea poate fi 0..*Max*-1 la implementarea în C/C++)
- **legăturile** dintre nodurile listei se memorează în tabloul **urm[1..Max]** (indexarea poate fi 0..*Max*-1 la implementarea în C/C++)
  - **urm[i]** este indicele din tabloul **e** unde se memorează nodul care urmează în listă după nodul *i*
- echivalentul legăturii **NIL** (folosită în cazul reprezentării înlățuirilor folosind **alocare dinamică**) este **urm[i]=0**
  - în cazul în care vectorii **e** și **urm** sunt indexați de la 0, echivalentul legăturii **NIL** va fi **urm[i]=-1**
- valoarea unui element din vectorul **urm** este un indice între 1 și *Max*
- corespondența dintre implementarea listei înlățuite prin adrese (ex. nod a cărui adresă de memorare este *p*) și prin “indici” (nod indicat prin indicele *i*) este dată în Tabela 1.
- capacitatea vectorilor **e** și **urm** poate fi mărită dinamic, dacă este necesar - numărul de elemente din listă depășește numărul de locații alocat inițial (a se vedea **vectorul dinamic**).
  - după redimensionare și copierea elementelor, e necesară reinitializarea listei înlățuite a spațiului liber (folosind locațiile nou adăugate în vector)
  - în cazul redimensionării, considerând lista reprezentată simplu înlățuit, operația *adaugaInceput* va avea complexitatea timp **amortizată**  $\theta(1)$ .

	Alocare dinamică	Înăntuiri pe tablou
<b>nod din listă</b>	$p$	$i$
<b>informația utilă a unui nod</b>	$[p].e$	$e[i]$
<b>legătura unui nod</b>	$[p].urm$	$urm[i]$

Tabela 1: Corespondența dintre implementarea listei înlăntuite prin adrese și prin “indici”

- LSI va memora  $prim$  (indicele la care este memorat pimul nod al listei), eventual  $ultim$ 
  - lista este **vidă** dacă  $prim = 0$  (sau -1, în cazul în care indexarea vectorilor începe de la 0).

### Exemplu

Să considerăm lista  $l=(a, b, c, d, e, f)$  reprezentată simplu înlăntuit, cu înăntuirile reprezentate folosind un tablou având capacitatea maximă (initială) de 10 locații ( $Max = 10$ ).

Reprezentarea este dată în Tabela 2.

Indice	1	2	3	4	5	6	7	8	9	10
<b>e</b>	c	-	e	a	-	b	-	-	d	f
<b>urm</b>	9	5	10	6	8	1	2	0	3	0

Tabela 2:  $prim=4$

În cazul memorării elementelor listei și a înănturilor pe tablou, avem nevoie de un mecanism pentru gestionarea spațiului liber rămas în tablouri pentru memorarea altor elemente.

- $\Rightarrow$  lista înlăntuită a spațiului liber, a cărui prim element îl numim **primLiber**
- pentru exemplul din Tabela 2, lista înlăntuită a spațiului liber poate fi **7**  $\rightarrow$  **2**  
 $\rightarrow$  **5**  $\rightarrow$  **8**
  - în cazul de mai sus  $primLiber=7$ , iar ultimul spațiu liber este 8.
  - semnificația elementelor din lista spațiului liber este următoarea: locațiile din tabloul  $e$  (în care sunt memorate elementele) având indicii 7, 8, 5 și 2 sunt **libere** (nu au memorate elemente ale listei)

În plus, trebuie furnizate 2 operații **rapide** ( $\theta(1)$ ) pentru *alocarea* și *delocarea* unui spațiu liber

- similar cu ceea ce oferă compilatoarele pentru alocarea/deallocarea pointerilor (**new/delete** în C++)

- operațiile **aloca** / **dealoca** pe care le folosim în Pseudocod, conform convențiilor noastre

### Cum se face alocarea?

- se șterge prima valoare din lista înlățuită a spațiului liber (**primLiber**)
  - complexitatea timp a operației e  $\theta(1)$

### Cum se face dealocarea?

- se adaugă (în față) o valoare în lista înlățuită a spațiului liber (înainte de **primLiber**)
  - complexitatea timp a operației e  $\theta(1)$

Pe lângă cele două operații anterior menționate, vom avea nevoie de inițializarea spațiului liber din listă (la început, în constructorul listei).

### Cum se face inițializareaa listei spațiului liber?

- toate locațiile (1..*Max*) trebuie adăugate în lista înlățuită a spațiului liber și **primLiber** setat corespunzător
- de exemplu, se poate inițializa lista spațiului liber astfel: **1** → **2** → **3** → ... **Max**
  - ulterior, datorită dinamicității listei (adăugări, ștergeri) lista spațiului liber nu va mai conține locațiile în ordine
  - complexitatea timp a operației e  $\theta(\text{Max})$  (operția va fi descrisă în Pseudocod, mai jos)
- se poate alege orice modalitate de înlățuire a locațiilor în lista spațiului liber (ex. *Max* → *Max-1* → ... → 1, sau se pot înlățui locațiile în ordine aleatoare)

#### 1.1. Exemplu LSI - reprezentarea înlățuirilor pe tablou

Pentru reprezentarea LSI pe tablou, avem nevoie de următoarea structură

##### LSI

cp: Intreg //capacitatea de memorare a celor doi vectori  
 prim, primLiber: 0..cp //numere întregi, a căror valoare e în intervalul [0, cp]  
 e: TElement[] //vectorul de elemente  
 urm: Intreg[] //vectorul de legături - valorile sunt în intervalul [0, cp]

Vom începe prin a descrie operațiile de:

- *alocare* spațiu liber în listă
  - pentru similitudine cu alocarea dinamică, vom numi această operație **aloca**

- *dealocare* spațiu (care a fost liber)
  - pentru similitudine cu alocarea dinamică, vom numi această operație **dealoca**
- inițializarea listei spațiului liber
  - operația o vom numi **initSpatiuLiber**

De menționat că operațiile **aloca**, **dealoca**, **initSpatiuLiber** NU vor fi în interfața containerelor implementate folosind o LSI cu înlănțuirii pe tablou, ci doar în partea de implementare.

- în clasa care implementează containerul, operațiile vor fi în secțiunea **private**, nu **public**

**Subalgoritm alocă(*l, i*)**

```
{furnizează un spațiu liber de indice i }
{se șterge primul nod din lista spațiului liber}
i ← l.primLiber
l.primLiber ← l.urm[l.primLiber]
```

SfSubalgoritm

- Complexitate:  $\theta(1)$

Trebuie menționat faptul că spațiul liber *i* rezultat în urma apelului **alocă(*l, i*)** poate fi 0, caz în care lista nu mai are spațiu liber, fiind necesară redimensionarea.

**Subalgoritm dealoca(*l, i*)**

```
{trece poziția i în lista spațiului liber }
{se adaugă i lînceputul listei spațiului liber}
l.urm[i] ← l.primLiber
l.primLiber ← i
```

SfSubalgoritm

- Complexitate:  $\theta(1)$

**Subalgoritm initSpatiuLiber(*l, cp*)**

```
{creează o LSI de lungime cp - toate pozițiile din tablou sunt libere }
```

Pentru  $i = 1, cp - 1$  executa

```
  l.urm[i] ← i+1
```

SfPentru

```
  l.urm[cp] ← 0
```

```
  l.primLiber ← 1
```

SfSubalgoritm

- Complexitate:  $\theta(cp)$

Indice	1	2	3	...	$cp-1$	$cp$
e	-	-	-	-	-	-
urm	2	3	4	...	$cp$	0

Tabela 3: Operația de inițializare a spațiului liber (**initSpatiuLiber**).  $primLiber=1$

Descriem, în continuare, constructorul listei, care inițializează lista cu lista vidă.

```

Subalgoritm creeaza( $l, cp$ )
  {creează lista vidă}
  {se inițializează lista spațiului liber}
  initSpatiuLiber( $l, cp$ )
   $l.\text{prim} \leftarrow 0$ 
SfSubalgoritm

```

- Complexitate:  $\theta(cp)$

Vom descrie, în continuare, operația **adaugaInceput**. Similar cu cazul listei înlăntuite cu alocare dinamică, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă. Specificația funcției a fost descrisă în **Cursul 4**.

Crearea unui nod în lista simplu înlăntută cu înlăntuiri pe tablou ( $i$  este o poziție liberă,  $e$  este informația utilă care trebuie memorată) este ilustrată mai jos.

indice	$i$
element	$e$
următor	$0$

Tabela 4: Crearea unui nod în lista simplu înlăntută cu înlăntuiri pe tablou ( $i$  este o poziție liberă,  $e$  este informația utilă care trebuie memorată) este ilustrată mai jos

```

Functia creeazaNod( $l, e$ )
  {dacă nu mai există spațiu de memorare în listă}
  Daca  $l.\text{primLiber} = 0$  atunci
    @realocare vectori
    @reinitializare spațiu liber
  SfDaca
  aloca( $l, i$ )
  {sigur  $i$  e diferit de 0}
   $l.e[i] \leftarrow e$ 
   $l.urm[i] \leftarrow 0$ 
  creeazaNod  $\leftarrow i$ 
SfFunctia

```

- Complexitate:  $\theta(1)$ ) amortizat, dacă se folosesc vectori dinamici (redimensionare)

```

Subalgoritm adaugaInceput( $l, e$ )
  {adaugă elementul  $e$  la începutul listei}
  nou  $\leftarrow$  creeazaNod( $l, e$ )
  {adaugăm elementul  $e$  înainte de prim}
   $l.urm[nou] \leftarrow l.\text{prim}$ 

```

```

l.prim ← nou
SfSubalgoritm

```

- Complexitate:  $\theta(1)$ ) amortizat, dacă se folosesc vectori dinamici (redimensionare)

Subalgoritmul pentru ștergerea unui element de pe o poziție  $p$  din listă (indicată prin indicele din tablou). . Sunt 2 cazuri la ștergere:

- se șterge  $prim$  (Figura 1 (a));
- se șterge un nod  $p$  diferit de  $prim$  (Figura 1 (b)).

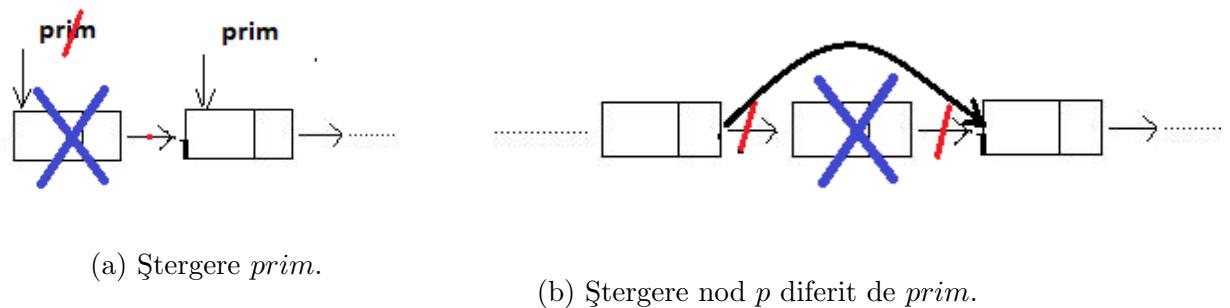


Figura 1

```

Subalgoritm sterge(l, p, e)
{se șterge nodul p, p ≠ 0, e e valoarea ștearsă}
{e e elementul șters }
e ← l.e[p]
Daca p = l.prim atunci
    {se șterge prim}
    l.prim ← l.urm[p]
altfel
    {se parcurge pâna la nodul p}
    q ← l.prim
    {sigur p este în listă, prin precondiție}
    CatTimp l.urm[q] ≠ p executa
        q ← l.urm[q]
    SfCatTimp
    {q este nodul care precede p }
    {se șterge nodul p }
    l.urm[q] ← l.urm[p]
SfDaca
{adăugăm locația p în lista spațiului liber }
dealoca(l, p)
SfSubalgoritm

```

- Complexitate:  $O(n)$ ,  $n$  fiind numărul de elemente din listă
  - cazul favorabil  $\theta(1)$  - șterg la început/sfârșit
  - cazul defavorabil  $\theta(n)$  - șterg penultimul element al listei

## 1.2. Exemplu implementare iterator pe un container reprezentat sub forma unei LSI cu înlăntuiri pe tablou

Presupunem că avem un **Container** oarecare (de ex. Colecție) reprezentat sub forma unei LSI cu înlăntuiri pe tablou, după cum urmează.

### Container

cp: Intreg //capacitatea de memorare a celor doi vectori  
prim, primLiber: 0..cp //numere întregi, a căror valoare e în intervalul [0, cp]  
e: TElement[] //vectorul de elemente  
urm: Intreg[] //vectorul de legături - valorile sunt în intervalul [0, cp]

În acest caz, iteratorul pe Container ar trebui să conțină:

- o referință către container
- adresa unui nod din lista simplu înlăntuită folosită pentru reprezentarea containerului (*current*)

### IteratorContainer

c : Container //containerul pe care îl iterează  
current: Intreg //poziția (indicele) nodului current al LSI

Operațiile specifice ale iteratorului (creează, valid, element, următor) le vom descrie, mai jos, în Pseudocod. Toate operațiile au complexitate timp  $\theta(1)$ .

Subalgoritm **creeaza**(*i, c*)  
{*pre: c este un container*}  
{*post: se creează iteratorul i pe containerul c*}  
{*elementul current al iteratorului referă primul element din c*}  
{*se setează containerul în iterator*}  
*i.c*  $\leftarrow c$   
{*se setează elementul current al iteratorului*}  
*i.current*  $\leftarrow c.\text{prim}$   
SfSubalgoritm

Functia **valid**(*i*)  
{*pre: i este un iterator*}  
{*post: se verifică dacă elementul current este valid*}  
{*iteratorul este valid dacă elementul current este diferit de 0*}  
*valid*  $\leftarrow i.\text{current} \neq 0$   
SfFunctia

Subalgoritm **element**(*i, e*)  
{*pre: i este un iterator, i este valid*}  
{*post: e este elementul indicat de current*}  
*e*  $\leftarrow l.e[i.\text{current}]$   
SfSubalgoritm

Subalgoritm **urmator**(*i*)

```

{pre: i este un iterator, i este valid}
{post: se deplasează referința curent a iteratorului}
i.current ← l.urm[i.current]
SfSubalgoritm

```

În directorul asociat Cursului 6 găsiți implementarea parțială, în limbajul C++, a containerului **Colecție** (repräsentarea este sub forma unei LSI care memorează toate elementele colecției, folosind reprezentarea înlățuirilor pe vector STATIC).

**TEMĂ.** Scrieți în Pseudocod/implementați operațiile specifice pe LSI ordonată cu înlățuirile reprezentate pe tablou (LSIO) și deduceți complexitățile acestora.

## 2. Lista dublu înlățuită (LDI) - reprezentarea înlățuirilor pe tablou

Convențiile de memorare sunt aceleași ca la LSI cu înlățuiriri pe tablou, doar se adaugă un treilea vector **prec** pentru a memora legăturile spre nodurile precedente din listă.

Să considerăm lista  $l=(a, b, c)$  reprezentată dublu înlățuit, cu înlățuirile reprezentate folosind un tablou având capacitatea maximă de 8 locații ( $Max = 10$ ). Locațiile se consideră a fi numerotate de la 1 la  $Max$ .

Reprezentarea este dată în Tabela 3.

Indice	1	2	3	4	5	6	7	8
e	-	a	-	-	b	-	-	c
urm	3	5	4	6	8	7	0	0
prec		0			2			5

Tabela 5:  $prim=2$ ,  $ultim=8$ ,  $primLiber=1$

### Observații

- Reprezentarea listei va fi

#### LDI

cp: Intreg //capacitatea de memorare a celor doi vectori  
 prim, ultim, primLiber: 0..cp //numere întregi, a căror valoare e în [0, cp]  
 e: TElement[] //vectorul de elemente  
 urm, prec: Intreg[] //vectorii de legături - valorile sunt în intervalul [0, cp]

- lista înlățuită a spațiului liber e suficient să fie simplu înlățuită.

- pentru exemplul din Tabela 3, lista înlățuită a spațiului liber este **1** → **3** → **4** → **6** → **7**
- operațiile **aloca**, **dealoca**, **initSpatiuLiber** sunt ca și la LSI, fiind valabil tot ce s-a discutat în Secțiunea 1.
- capacitatea vectorilor **e**, **urm** și **prec** poate fi mărită dinamic, dacă este necesar - numărul de elemente din listă depășește numărul de locații alocat inițial (a se vedea **vectorul dinamic**).

- \* după dimensionare și copierea elementelor, e nevoie de reinitializarea listei înlăntuite a spațiului liber (folosind locațiile nou adăugate în vector)

Vom descrie, în continuare, operația **adaugaInainte**. Similar cu cazul listei simplu înlăntuite cu înlăntuiri pe tablou, vom folosi o funcție auxiliară care creează un nod având o anumită informație utilă. Specificația funcției a fost descrisă în **Cursul 4**.

Crearea unui nod în lista dublu înlăntută cu înlăntuiri pe tablou ( $i$  este o poziție liberă,  $e$  este informația utilă care trebuie memorată) este ilustrată mai jos.

indice	$i$
element	$e$
următor	$0$
precedent	$0$

Tabela 6: Crearea unui nod în lista dublu înlăntută cu înlăntuiri pe tablou ( $i$  este o poziție liberă,  $e$  este informația utilă care trebuie memorată) este ilustrată mai jos

```

Functia creeazaNod( $l, e$ )
{dacă nu mai există spațiu de memorare în listă}
Daca  $l.\text{primLiber} = 0$  atunci
    @realocare vectori
    @reinițializare spațiu liber
SfDaca
    aloca( $l, i$ )
     $l.e[i] \leftarrow e$ 
     $l.urm[i] \leftarrow 0$ 
     $l.pec[i] \leftarrow 0$ 
    creeazaNod  $\leftarrow i$ 
SfFunctia

```

- Complexitate:  $\theta(1)$ ) amortizat, dacă se folosesc vectori dinamici (redimensionare)

Subalgoritmul pentru adăugarea unui element înaintea unui nod din listă (indicat prin indicele la care este memorat în tablou). Sunt două cazuri care trebuie tratate

- adăugare înainte de  $prim$  (dacă  $p = prim$ ) (Figura 2(a))
- adăugare înainte de un nod  $p$  diferit de  $prim$  (Figura 2(b))

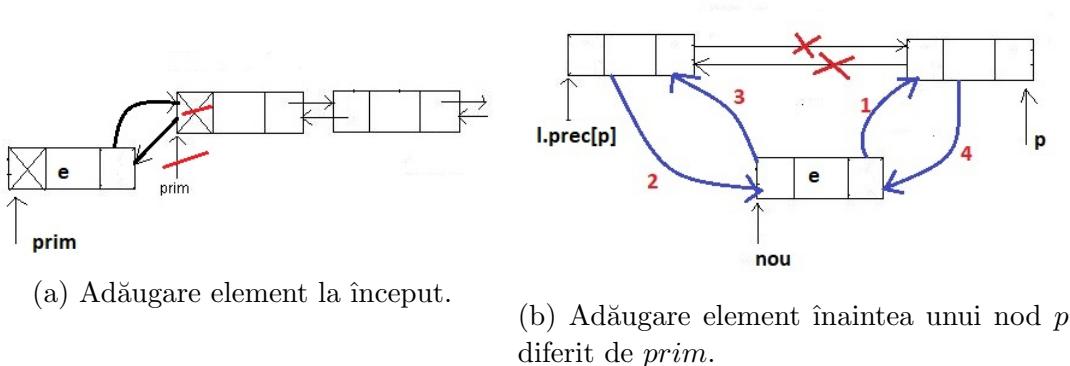


Figura 2

Subalgoritm adaugaInainte( $l, p, e$ )

```

{pre: l: LDI, p,  $p \neq 0$  este poziția unui nod (indice) în l, e: TElement}
{post: se adaugă e înaintea nodului p}
nou ← creeazaNod(l, e)
{dacă se adaugă înaintea primului nod }
Daca  $p = l.prim$  atunci
    {se adaugă înainte de prim}
    l.urm[nou] ← l.prim
    {p este diferit de 0, prin precondiție}
    l.prec[l.prim] ← nou
    {se actualizează prim}
    l.prim ← nou
altfel
    {se adaugă între precedentul lui p și p}
    l.urm[nou] ← p
    l.urm[l.prec[p]] ← nou
    l.prec[nou] ← l.prec[p]
    l.prec[p] ← nou
SfDaca
SfSubalgoritm

```

- Complexitate:  $\theta(1)$ ) amortizat, dacă se folosesc vectori dinamici (redimensionare)

### Observație

- Implementarea iteratorului pe un container oarecare (de ex. Colecție) reprezentat sub forma unei LDI cu înlățuiriri pe tablou este similar cu cel descris în Secțiunea 1.2. Spre deosebire de cazul LSI, iteratorul poate fi bidirectional.

**TEMĂ.** Scrieți în Pseudocod operațiile specifice pe LDI/LDIO (ordonată) cu înlățuirile reprezentate pe tablou și deduceți complexitățile acestora.

# Concluzii - liste

- Memorarea elementelor listei secvențial într-un tablou unidimensional (vector).
  - eficientă pentru acele liste în care se fac multe operații de adăugare la sfârșit, accesare și mai puține inserări.
  - dacă se folosește un tablou static, deficiența este dată de gestionarea inefficientă a spațiului de memorare (este deseori necesar să se supraestimeze spațiul necesar memorării elementelor).
  - tabloul dinamic exclude dezavantajul tablourilor statice de stabilire statică a capacitatei maxime a unei liste, dar totuși rămâne dezavantajul dat de ineficiența operațiilor de inserare și ștergere a elementelor din interiorul listei. Inserările și ștergerile, într-o astfel de listă, se fac dificil deoarece necesită deplasări ale elementelor.
- Reprezentarea înlănțuită.
  - spațiu adițional pentru memorarea legăturilor - ceea ce conduce la creșterea complexității-spațiu
  - gestionarea memoriei se face mai eficient
  - operațiile de inserare și ștergere se pot face mult mai eficient.
- Decizia asupra alegerii modului de implementare a unei liste depinde de gradul de dinamicitate al listei și de tipul aplicațiilor în care urmează a fi folosită:
  - Dacă actualizările (inserări, ștergeri) sunt rare, este preferată reprezentarea folosind tablouri.
  - Dacă actualizările sunt dese, este preferată reprezentarea înlănțuită.
- În funcție de restricțiile de acces și actualizare a elementelor unei liste, există diferite specializări ale listelor: stive, cozi, cozi complete, liste liniare generalizate.

# ANSAMBLU (HEAP)

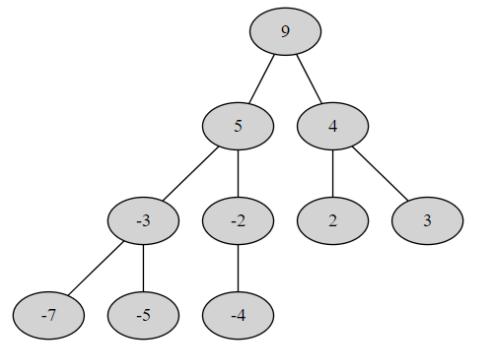
- Este o **structură de date** eficientă pentru memorarea *cozilor cu priorități*
- Tipuri de ansamblu: **binar**, binomial, Fibonacci, *leftist heaps*, *skew heaps*, etc.
- Vom prezenta, în cele ce urmează, structura de **ansamblu binar**.
  - În directorul [Curs7/docs](#) găsiți documentații despre celelalte tipuri de ansambluri.

**DEFINITIE.** Un *ansamblu binar* ( $a_1, a_2, \dots, a_n$ ) este un vector (ale căruia elemente sunt de tip comparabil, **TElement=TComparabil**) care poate fi vizualizat sub forma unui arbore binar având **structură de ansamblu** și care verifică **proprietatea de ansamblu**.

**Structură de ansamblu** – arborele binar (sub forma căruia poate fi vizualizat ansamblul) este *aproape plin* (dacă toate nivelurile acestuia sunt complete, exceptând ultimul nivel care este plin de la stânga la dreapta).

**Exemplu** Vectorul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** poate fi vizualizat sub forma arborelui binar din Figura 1. Acesta are structură de *ansamblu*, fiind *aproape plin*.

- elementele vectorului sunt dispuse pe niveluri, în ordine, începând cu primul element.



**Figura 1.** Vectorul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** vizualizat sub forma unui arbore cu structură de ansamblu

**Observații:** Pe baza vizualizării vectorului  $a_1, a_2, \dots, a_n$  sub forma unui arbore binar aproape plin (ca în Figura 1) deducem următoarele

- $a_1$  este elementul din rădăcina arborelui
- $a_i$  are fiul stâng  $a_{2i}$  dacă  $2 \cdot i \leq n$  și fiul drept  $a_{2i+1}$  dacă  $2 \cdot i + 1 \leq n$
- $a_i$  are părintele  $a_{[i/2]}$

**Proprietatea de ansamblu** constă în verificarea următoarelor condiții (impuse între valorile nodurilor din arborele asociat și valorile din descendenți – stâng și drept)

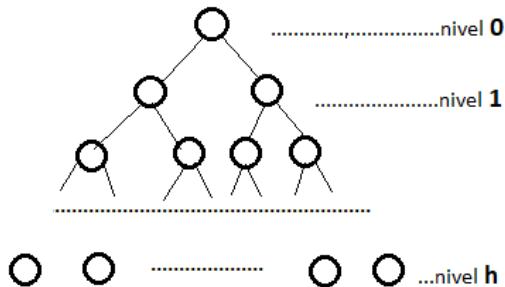
- 1)  $a_i \geq a_{2i} \quad \forall i$ , dacă  $2 \cdot i \leq n$
- 2)  $a_i \geq a_{2i+1} \quad \forall i$  dacă  $2 \cdot i + 1 \leq n$

## Observații

- Dacă relația de ordine din inegalitățile 1) și 2) (de mai sus) este „ $\geq$ ”, atunci *heap*-ul se numește **max-heap**;
  - o Vectorul indicat în **Exemplu** verifică proprietatea de ansamblu și este un **max-heap** (la fiecare nod din arbore, valoarea elementului este mai mare sau egală cu cea a descendenților).
- Dacă relația de ordine din inegalitățile 1) și 2) (de mai sus) este „ $\leq$ ”, atunci *heap*-ul se numește **min-heap**;
- Relația „ $\geq$ ” poate fi generalizată la o relație de ordine  $\mathfrak{R}$  oarecare
- Ansambul binar este, în general, memorat **secvențial** folosind un vector (dinamic), fără a fi necesară memorarea înlănțuită - legături între elemente (ex. pointeri).

Datorită **proprietății de ansamblu** (relațiile pe care le satisfac elementele acestuia), următoarele afirmații sunt adevărate într-un ansamblu  $a_1, a_2, \dots, a_n$ .

- $a_1$  este cel mai **mare** element din ansamblu dacă  $\mathfrak{R} = " \geq "$
- Dacă  $\mathfrak{R} = " \geq "$ , atunci pe orice drum de la rădăcină la un nod, elementele sunt ordonate descrescător.
- **Înălțimea** unui heap cu  $n$  elemente este  $\theta(\log_2 n)$ . Ca urmare, timpul de execuție a operațiilor specifice va fi  $O(\log_2 n)$ .
  - o **Înălțimea** este definită ca lungimea drumului de la rădăcină la o frunză (nod care nu mai are descendenți – toate frunzele, într-un ansamblu, sunt pe ultimul nivel). În exemplul din Figura 1, înălțimea este 3.
  - o În cazul în care arborele binar asociat ansamblului ar fi plin (toate nivelurile ar fi pline), ca în figura de mai jos, iar  $h$  este înălțimea ansamblului, observăm următoarele:



$$\begin{aligned}
 &\text{pe nivelul } i \text{ în arbore sunt } 2^i \text{ noduri} \Rightarrow n = 1 + 2 + \dots + 2^h \\
 &\Rightarrow n = 2^{h+1} - 1 \\
 &\Rightarrow h = \log_2(n+1) - 1 \in \theta(\log_2 n)
 \end{aligned}$$

Sunt 2 operații specifice pe ansamblu:

- o **adăugare** element (astfel încât să se păstreze proprietatea de ansamblu)
  - o **ștergere** element (se șterge elementul maxim dacă  $\mathfrak{R} = " \geq "$ , cel din vârful ansamblului).
- Pp. în continuare  $\mathfrak{R} = " \geq "$ .
  - Pp. în cele ce urmează că elementele din vectorul care memorează ansamblul sunt indexate de la 1.

Pentru reprezentarea ansamblului vom folosi o structură care memorează vectorul corespunzător.

## Ansamblu

Max: Intreg {capacitatea maximă de memorare}

n: Intreg {nr.de elemente din ansamblu}

e: TElement[1..n] {elementele din ansamblu}

Vom discuta, în cele ce urmează, cele două operații.

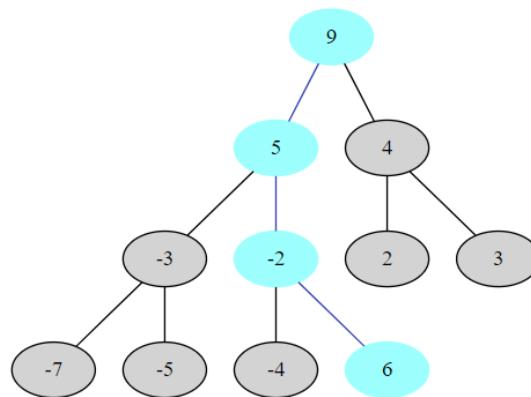
## Adăugare

Presupunem că în ansamblul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** (vizualizat în Figura 1) dorim să adăugăm valoarea **6**.

Adăugarea presupune următoarele

- Adăugăm valoarea **6** la finalul ansamblului (vectorului)
- Restabilim proprietatea de ansamblu, posibil alterată în urma adăugării elementului.

În exemplul nostru, prin adăugarea lui **6** la finalul ansamblului obținem



Observăm că proprietatea de ansamblu este alterată doar pe drumul de la elementul adăugat până la rădăcină (6, -2, 5, 9). Practic, va trebui să căutăm loc pentru **6** printre ascendenții săi (marcați pe figură), să îl **urcăm** în ansamblu, până când va fi verificată proprietatea de ansamblu. Modificările necesare pentru a reface proprietatea de ansamblu sunt următoarele:

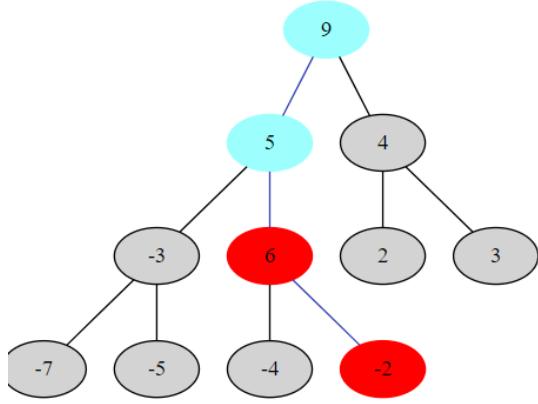
**Pas 1.** Comparăm 6 cu părintele său (-2). 6 este mai mare decât -2, înseamnă că îl coborâm pe -2 în locul lui 6 și continuăm să îl urcăm pe 6.

**Pas 2.** Comparăm 6 cu 5. 6 este mai mare decât 5, înseamnă că îl coborâm pe 5 în locul lui 6 și continuăm să îl urcăm pe 6.

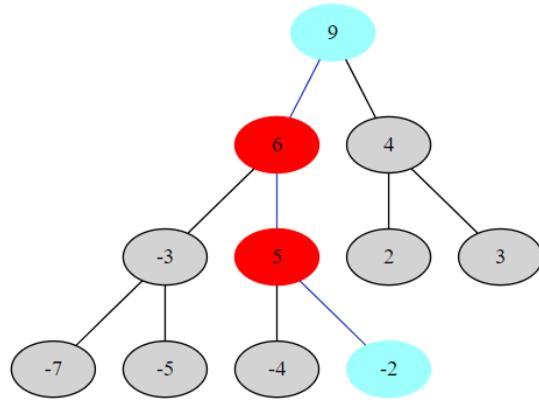
**Pas 3.** Comparăm pe 6 cu 9.  $6 \leq 9$  (părintele), înseamnă că oprim procesul iterativ, am găsit loc pentru 6.

**STOP**

## Pas 1



## Pas 2



Prin urmare, ansamblul rezultat în urma adăugării lui **6** în ansamblul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** este **9, 6, 4, -3, 5, 2, 3, -7, -5, -4, -2** (corespunzător arborelui din dreapta).

Din procesul descris anterior, observăm faptul că numărul maxim de pași ai structurii iterative pentru urcarea lui **6** în ansamblu este  $h$  (înălțimea arborelui). Rezultă faptul că operația de **adăugare** are complexitatea timp  $O(\log_2 n)$ .

Subalgoritmul de adăugare este descris în Pseudocod mai jos.

**Subalgoritm ADAUGĂ** ( $a, e$ ) este {complexitate timp  $O(\log_2 n)$  }

{pre:  $a$ : Ansamblu,  $a$  nu e plin,  $e$ :TElement }

{post:  $a$  rămâne ansamblu după adăugare}

$a.n \leftarrow a.n + 1$

$a.e[a.n] \leftarrow e$

**URCĂ**( $a, a.n$ ) {restabilește proprietatea de ansamblu posibil alterată}

sfADAUGĂ

**Obs.** În subalgoritmul anterior, nu s-a verificat la adăugare dacă ansamblul e plin. La implementare se poate redimensiona vectorul dacă se observă că se depășește capacitatea maximă alocată.

**Subalgoritm URCĂ** ( $a, i$ ) este { complexitate timp  $O(\log_2 n)$  }

{urcă elementul de pe poziția  $i$  spre rădăcină până va fi satisfăcută proprietatea de ansamblu}

{pre:  $a$  ansamblu nevid, elem. de pe poziția  $i$  a fost actualizat}

{post:  $a$  este ansamblu}

$e \leftarrow a.e[i]$  {elementul de urcat}

$k \leftarrow i$  {poziția unde va fi pus elementul  $e$ }

$p \leftarrow [k/2]$  {părintele lui  $k$ }

{căutăm o poziție pentru  $e$  printre strămoșii lui}

Cât timp ( $p \geq 1$ ) și ( $a.e[p] < e$ ) execută

$a.e[k] \leftarrow a.e[p]$  {strămoșii mai mici decât  $e$  sunt coborâți}

$k \leftarrow p$

$p \leftarrow [p/2]$

sfCâtTimp

{s-a gasit poziția  $k$  pe care poate fi adăugat  $e$ }

$a.e[k] \leftarrow e$

sfURCĂ

Ilustrăm, în tabelul de mai jos, execuția pas cu pas a algoritmului **URCA**, în cazul adăugării valorii 6 (exemplul anterior).

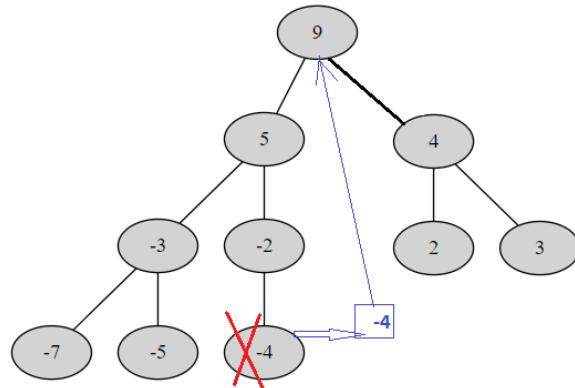
$e$	$k$	$p$	$a.e[p]$	$a.e[p] < e$	Modificări
6	11	5	-2	$-2 < 6$ , DA	$a.e[11] \leftarrow -2$
	5	2	5	$5 < 6$ , DA	$a.e[5] \leftarrow 5$
	2	1	9	$9 < 6$ , NU	$a.e[2] \leftarrow 6$

## Ștergere

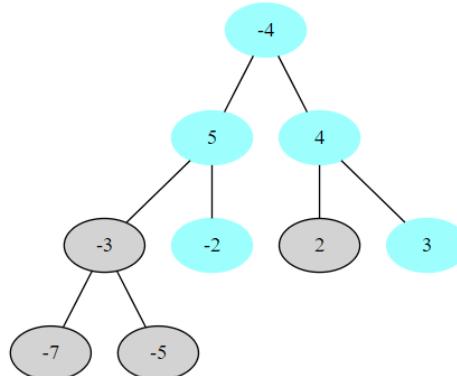
Presupunem că dorim să ștergem elementul maxim (**9**) din ansamblul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** (vizualizat în Figura 1). După cum menționam anterior, ștergerea din ansamblu este prespecificată, se șterge doar primul element din ansamblu (memorat în rădăcina arborelui asociat).

Ștergerea presupune următoarele

- Ultimul element din ansamblu (cel de finalul vectorului, **-4** în exemplul nostru), îl mutăm în locul rădăcinii.
- Restabilim proprietatea de ansamblu, posibil alterată în urma modificării elementului din vârful ansamblului.



În exemplul nostru, prin mutarea lui **-4** în rădăcină, obținem

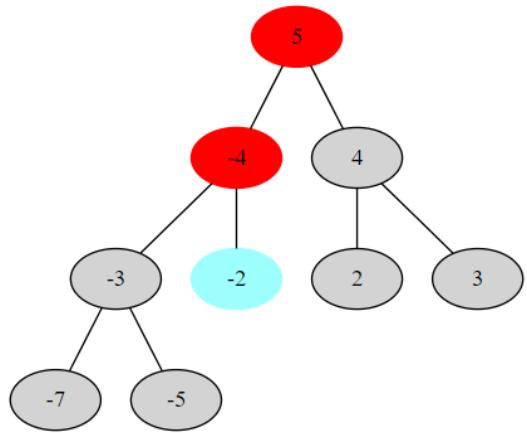


Observăm că proprietatea de ansamblu este alterată pe două drumuri  $(-4, 5, -2)$  și  $(-4, 4, -3)$ . Practic, va trebui să căutăm loc pentru **-4** printre descendenții săi (marcați pe figură), să îl **coborâm** în ansamblu până când va fi verificată proprietatea de ansamblu. De asemenea, observăm că este suficient să refacem proprietatea de ansamblu pe direcția descendantului maxim. Modificările necesare pentru a reface proprietatea de ansamblu sunt următoarele:

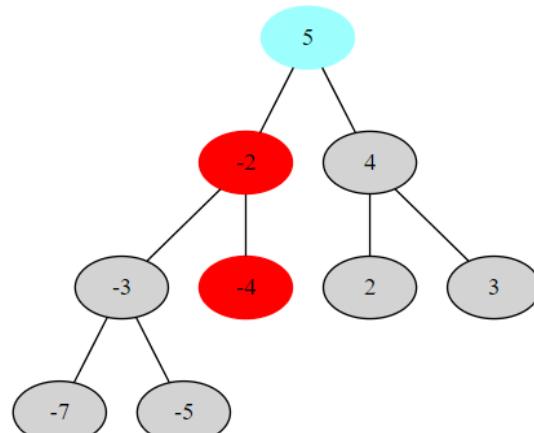
**Pas 1.** Comparăm pe  $-4$  cu descendantul său maxim  $(5)$ . Părintele  $(-4)$  este mai mic decât descendantul, înseamnă că îl urcăm pe  $5$  în locul lui  $-4$  și continuăm coborârea lui  $-4$ .

**Pas 2.** Comparăm  $-4$  cu  $-2$ .  $-4$  este mai mic decât  $-2$ , înseamnă că îl urcăm pe  $-2$  în locul lui  $-4$ . Nu mai avem unde să coborâm, îl vom pune pe  $-4$  în locul lui  $-2$ . **STOP**

**Pas 1**



**Pas2**



Prin urmare, ansamblul rezultat în urma ștergerii (valorii **9**) din ansamblul **9, 5, 4, -3, -2, 2, 3, -7, -5, -4** este **5, -2, 4, -3, -4, 2, 3, -7, -5** (corespunzător arborelui din dreapta).

Din procesul descris anterior, observăm faptul că numărul maxim de pași ai structurii iterative pentru coborârea lui **9** în ansamblu este  $h$  (înălțimea arborelui). Rezultă faptul că operația de **ștergere** are complexitatea timp  $O(\log_2 n)$ .

Subalgoritmul de ștergere este descris în Pseudocod mai jos.

**Subalgoritmul STERGE** ( $a, e$ ) este {complexitate timp  $O(\log_2 n)$ }  
{pre:  $a$ :Ansamblu,  $a$  nu e vid}  
{post:  $e$ :TElement este elementul maxim și e șters,  $a$  rămâne ansamblu după ștergere}  
 $e \leftarrow a.e[1]$  {elementul maxim}  
 $a.e[1] \leftarrow a.e[a.n]$   
 $a.n \leftarrow a.n-1$   
**COBOARĂ**( $a, 1$ ) {restabilește proprietatea de ansamblu posibil alterată}  
**sfSTERGE**

**Subalgoritmul COBOARĂ** ( $a$ ,  $poz$ ) este { complexitate timp  $O(\log_2 n)$  }

{ coboară elementul de pe poziția  $poz$  printre descendenți până va fi satisfăcută proprietatea de ansamblu }

{ pre:  $a$  ansamblu nevid, elem. de pe poziția  $poz$  a fost actualizat }

{ post:  $a$  este ansamblu }

```

 $e \leftarrow a.e[poz]$  {elementul de mutat}
 $i \leftarrow poz$  {poziția unde va fi pus elementul  $e$ }
 $j \leftarrow 2 \cdot poz$  {fiul stâng al lui  $i$ }
    {căutăm o poziție pentru  $e$  printre descendenți. Descendenții mai mari decât  $e$  urcă un nivel în arbore }
    câtimp ( $j \leq a.n$ ) execută {  $i$  are fiu stâng}
        dacă ( $j < a.n$ ) atunci {  $i$  are și fiu drept? Dacă da, îl luăm pe cel mai mare dintre ei}
            dacă  $a.e[j] < a.e[j+1]$  atunci
                 $j \leftarrow j+1$ 
            sfdacă
        sfdacă
        dacă  $a.e[j] \leq e$  atunci {cel mai mare fiu este mai mic sau egal cu  $e$ , atunci STOP}
             $j \leftarrow a.n+1$ 
        altfel
             $a.e[i] \leftarrow a.e[j]$  {fiul  $j$  urcă}
             $i \leftarrow j$ 
             $j \leftarrow 2 \cdot i$ 
        Sfdacă
    Sfcâttimp
     $a.e[i] \leftarrow e$  {pun elementul înapoi în structură}
sfCOBOARĂ

```

Ilustrăm, în tabelul de mai jos, execuția pas cu pas a algoritmului **COBOARĂ**, în cazul ștergerii (valorii 9).

$e$	$i$	$j$	$a.e[j] < a.e[j+1]$	$a.e[j] \leq e$	Modificări
-4	1	2	$5 < 4$ , NU	$5 \leq -4$ , NU	$a.e[1] \leftarrow 5$
	2	4	$-3 < -2$ , DA	$-2 \leq -4$ , NU	$a.e[2] \leftarrow -2$
		5			
	5	10 ( $j > a.n=9$ , STOP)			$a.e[5] \leftarrow -4$

## Aplicații ale structurii de ansamblu

1. **Ansamblul** este cea mai potrivită structură de date pentru memorarea elementelor unei **Cozi cu Priorități** (CP): operațiile *adaugă*, *element* (accesare element), *șterge* au complexitate  $O(\log_2 n)$ .

Analizăm, comparativ, următoarele structuri de date pentru reprezentarea unei CP folosind

- **Vector dinamic** ordonat, în care elementul cel mai prioritar este ultimul.
- **Listă simplu înlăntuită** ordonată, în care elementul cel mai prioritar este primul.
- **Listă dublu înlăntuită** ordonată, în care elementul cel mai prioritar este primul sau ultimul (nu contează).
- **Ansamblu**, cu elementul cel mai prioritar primul (în rădăcină)

Structura de date	adăugare	ștergere	accesare
Vector dinamic ordonat	$O(n)$	$\Theta(1)$ amortizat (dacă e cu redimensionare)	$\Theta(1)$
LSIO	$O(n)$	$\Theta(1)$	$\Theta(1)$
LDIO	$O(n)$	$\Theta(1)$	$\Theta(1)$
Ansamblu	$O(\log_2 n)$	$O(\log_2 n)$	$\Theta(1)$

2. **HEAPSORT.** Sortarea unui vector cu  $n$  elemente folosind un ansamblu. Complexitate timp  $O(n \log_2 n)$   
- se poate *in place*, fără memorarea suplimentară a ansamblului.

- **Folosind un ansamblu auxiliar (*out of place*, spațiu suplimentar de memorare  $\Theta(n)$ )**, ideea este următoarea:

- Se iau, pe rând, elementele din vector și se adaugă într-un ansamblu  $\Rightarrow O(n \log_2 n)$ 
  - se poate arăta că timpul necesar pentru construcția unui heap cu  $n$  elemente este  $O(n)$  (a se vedea Observația 2)
- Se aplică de  $n$  ori ștergerea din ansamblul auxiliar și rezultă elementele în ordine  $\Rightarrow O(n \log_2 n)$

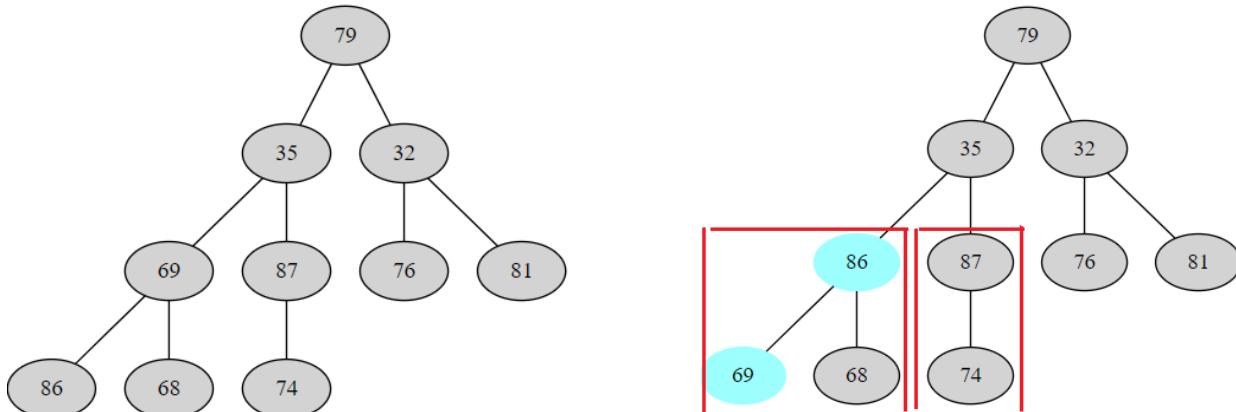
**Exemplu** Fie vectorul 1, 5, 3, 9, 7. Vrem să îl sortăm descrescător. Construim un **max-heap** cu elementele sale  $\Rightarrow$  ansamblul 9, 7, 3, 1, 5. Apoi scoatem toate elementele din heap și rezultă 9, 7, 5, 3, 1

- **Fără a folosi un ansamblu auxiliar (*in place*)**. Ideea: restructurăm vectorul încât să devină ansamblu (să fie satisfăcută proprietatea de ansamblu la orice nivel în arborele asociat).
  - se încearcă refacerea proprietății de ansamblu de jos în sus (de la frunze spre rădăcină), pornind de la nodurile de înălțime  $h-1$ , apoi  $h-2, \dots$  până se ajunge la înălțime 0 (întregul ansamblu).
    - această operație are complexitate timp  $O(n \log_2 n)$ .
  - Se aplică de  $n$  ori ștergerea din ansamblul auxiliar și rezultă elementele în ordine  $\Rightarrow O(n \log_2 n)$

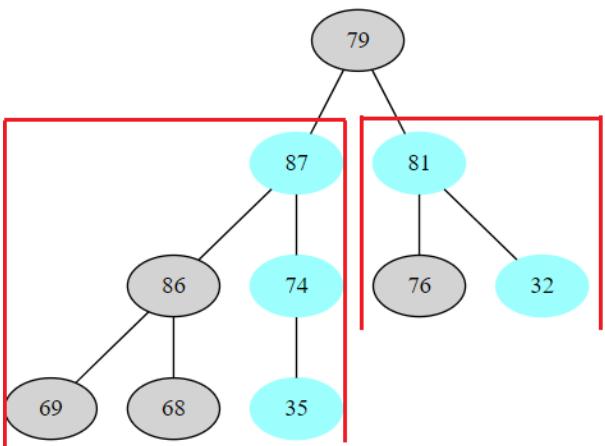
**Exemplu** Fie vectorul 79, 35, 32, 69, 87, 76, 81, 86, 68, 74. Ilustrăm, mai jos, modul în care este restructurat ansamblul

Inițial

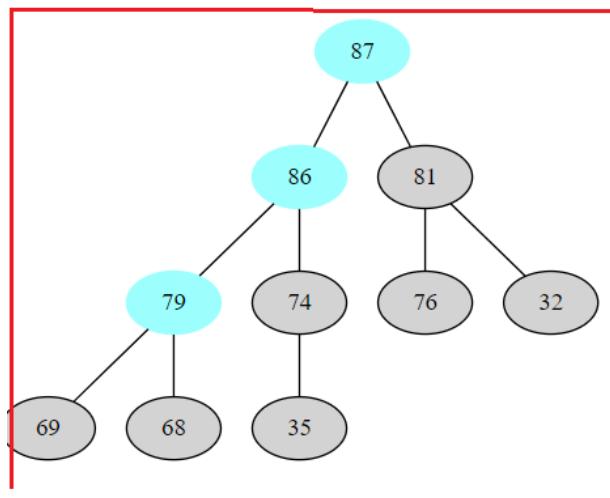
Pas 1



## Pas 2



## Pas 3



Se observă că ansamblul final (obținut după Pasul 3) **87, 86, 81, 79, 74, 76, 32, 69, 68, 35** reprezintă un **max-heap**

## Observații

1. Se poate demonstra (prin inducție) că un ansamblu binar cu  $n$  elemente are cel mult  $\lceil n/(2^{h+1}) \rceil$  noduri de înălțime  $h$ .
2. Se poate demonstra (pe baza 1) că un ansamblu binar se poate construi în  $O(n)$  dintr-un vector cu  $n$  elemente.
3. Reunirea (interclasarea) a două ansambluri binare cu  $n$  și  $m$  elemente se poate face în  $O(n+m)$ .

## PROBLEME

1. Fie ansamblul  $\langle 1, 2, 4, 2, 5 \rangle$ . Aplicați de două ori operația de ștergere.
2. Generalizați relația " $\geq$ " la o relație de ordine  $\mathcal{R}$  oarecare și implementați operațiile specifice.
3. Care este cel mai mic, respectiv cel mai mare număr de elemente dintr-un ansamblu având înălțimea  $h$ ?
4. Arătați că un ansamblu având  $n$  elemente are înălțimea  $\lceil \log_2 n \rceil$
5. Arătați că în orice subarbore al unui ansamblu rădăcina subarborelui conține cea mai mare valoare care aparține în acel arbore (dacă  $\mathcal{R} = " \geq "$ ).
6. Dacă  $\mathcal{R} = " \geq "$ , unde se poate afla cel mai mic element al unui ansamblu, presupunând că toate elementele sunt distințe?
7. Este vectorul în care elementele se succed în ordine descrescătoare un ansamblu?
8. Este secvența  $\langle 23, 17, 14, 6, 13, 10, 15, 7, 12 \rangle$  un ansamblu?
9. Să se generalizeze ansamblul binar la un ansamblu *ternar* sau *cuaternar* (în loc de 2 descendenți sunt 3, respectiv 4).
10. Să se implementeze, folosind un ansamblu binar, un container **CPk** similar cu **Coadă cu priorități**, exceptând faptul că vrem să accesăm și să ștergem **al k-lea cel mai prioritar element** în raport cu o relație

de ordine  $\mathfrak{R}$  între priorități (dacă  $\mathfrak{R} = \leq$ , atunci elementul cel mai prioritar este **minimul**). **Indicație:** pentru determinarea elementului cel mai prioritar dintre  $k$  elemente, se va folosi tot un ansamblu binar.

11. Găsiți un algoritm  $O(n \cdot \log_2 k)$  pentru a interclasă  $k$  liste ordonate, unde  $n$  este numărul total de elemente din listele de intrare. Reprezentarea listelor este ascunsă, acestea se parcurg folosind iteratori.

### Indicație

- generalizăm ideea de la interclasarea a două liste ordonate
- în fiecare dintre cele  $k$  liste, avem un element curent (indicat de un iterator)
  - a). pentru a extrage minimul/maximul dintre cele  $k$  liste, folosim un ansamblu  $\Rightarrow O(\log_2 k)$ 
    - în lista din care a fost găsit minimul/maxim, deplasăm iteratorul
    - elementul indicat de iterator, îl adăugăm în heap
  - b) repetăm pasul a) de  $n$  ori (pentru numărul de elemente din toate listele)  $\Rightarrow O(n \cdot \log_2 k)$

# TABELA DE DISPERSIE

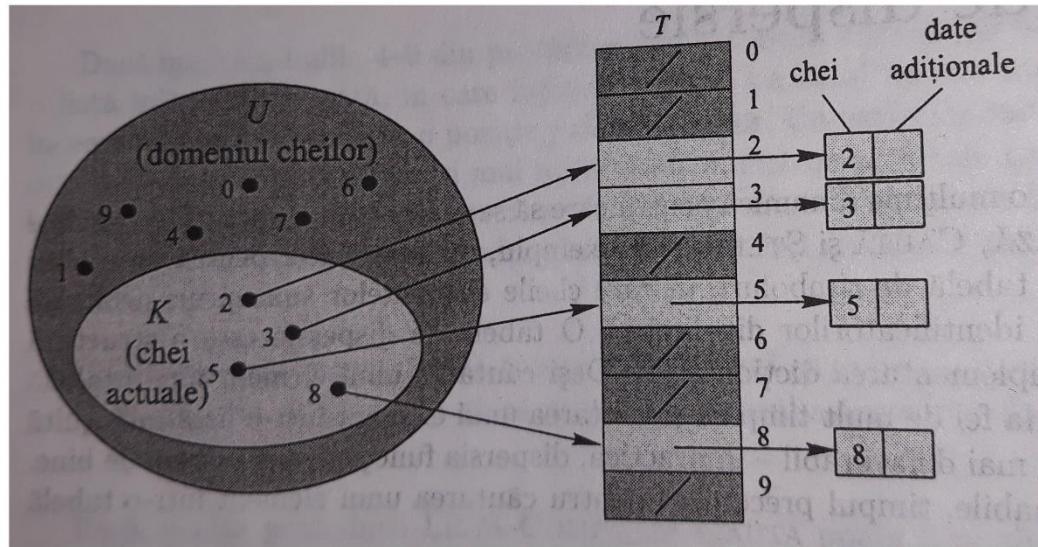
## Hash Table

- Este o structură de date **eficientă** pentru implementarea dicționarelor (și nu numai).
- Exemplu: un compilator păstrează o **tabelă de simboluri**, în care cheia este sirul de caractere corespunzător unui identificator
- TD poate fi folosită pentru implementarea containerelor pe care operațiile specifice sunt: **adăugare** element, **căutare** element, **ștergere** element. Ex: dicționare, colecții, multimi
  - JAVA
    - HashMap (dicționar reprezentat folosind o tabelă de dispersie)
    - HashSet (mulțime reprezentată folosind o tabelă de dispersie)
  - STL
    - unordered\_set (mulțime reprezentată folosind o tabelă de dispersie)
    - unordered\_map (dicționar reprezentat folosind o tabelă de dispersie).
- TD este o generalizare a noțiunii mai simple de **tabelă cu adresare directă**
- **Notății**
  - $n$  – numărul de elemente din container
  - un element  $e$  din container este o pereche cheie ( $c$ ) – valoare ( $v$ ) ( $\text{TElement} = \text{Tcheie} \times \text{TValoare}$ )
  - $U$  – **domeniu** (universul) cheilor
  - $K$  – domeniu actual al cheilor (mulțimea cheilor efectiv memorate în container)

### Tabelă cu adresare directă

- Notății și presupuneri
  - Presupunem chei numere naturale, chei distincte
  - Domeniul cheilor  $U = \{0, 1, 2, \dots, m-1\}$  -  $m$  relativ mic
  - $K$  – domeniu actual al cheilor (mulțimea cheilor efectiv memorate în container)
- Tabela cu adresare directă este memorată sub forma unui vector  $T[0..m-1]$ 
  - Locația  $T[c]$  va corespunde cheii  $c$  (la acea locație se memorează cheia și datele adiționale asociate acesteia)
  - Dacă o cheie  $c \notin K$ , atunci  $T[c]$  va conține NIL (sau o valoare specială care marchează locație goală)
  - $T[c]$  poate memora un pointer spre elementul având cheia  $c$  sau chiar elementul (cheia și valoarea asociată)

## Exemplu



Considerăm următoarea reprezentare:

**TElement**

$c$ :TCheie

$v$ :TValoare

**TabelaAdresareDirecta**

$m$ :Întreg

$e$ :TElement[0.. $m-1$ ]

Cele trei operații (**căutare**, **adăugare**, **ștergere**) pe o tabelă cu adresare directă sunt summarizate mai jos:

**CAUTĂ** ( $T, c$ )

//pre:  $T$  este o tabelă cu adresare directă,  $c$  este o cheie, de tip TCheie  
@ returnează  $T.e[c]$

**ADAUGĂ** ( $T, e$ )

//pre:  $T$  este o tabelă cu adresare directă,  $e$  este de tip TElement  
@  $T.e[e.c] \leftarrow e$

**ȘTERGE** ( $T, c$ )

//pre:  $T$  este o tabelă cu adresare directă,  $c$  este de tip TCheie  
@  $T.e[c] \leftarrow \text{NIL}$

- **Observații**

- o tabelă cu adresare directă funcționează bine dacă universul cheilor este mic
- complexitatea timp a operațiilor este  $\Theta(1)$
- spațiul de memorare este  $\Theta(|U|)$

- **Dezavantaje**

- dacă universul  $U$  este mare, memorarea tabelului  $T$  poate fi nepractică, sau chiar imposibilă, dată fiind memoria disponibilă.
- dacă mulțimea  $K$  este mică relativ la  $U$ , rămâne mult spațiu nefolosit  $\Rightarrow$  gestionare ineficientă a spațiului de memorare.

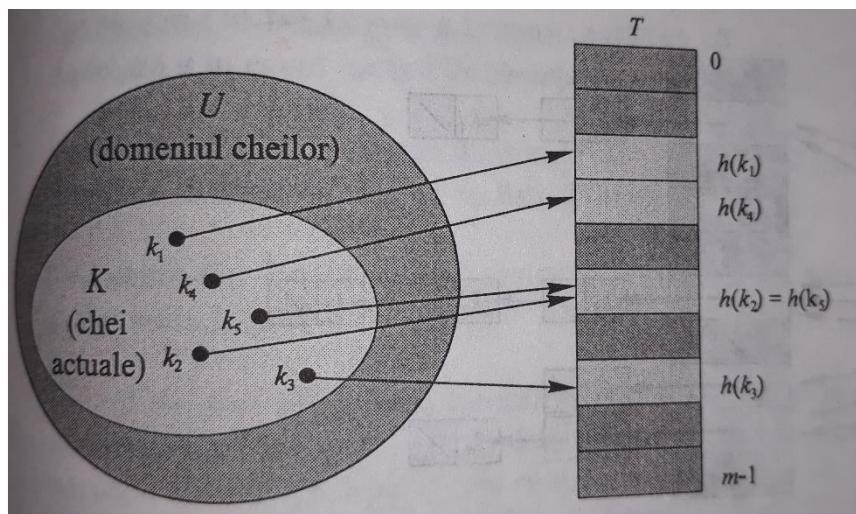
## PROBLEMĂ

Sugerați cum se poate implementa o tabelă cu adresare directă în care cheile elementelor memorate nu sunt neapărat distințe și elementele pot avea date adiționale.

## Tabelă de dispersie

- $T[0..m-1]$ 
  - $m$  – număr locații din tabelă
- reduce spațiul de memorare la  $\Theta(|K|)$  - eficientizare a spațiului de memorare (mai ales când  $K$  este mult mai mică decât  $m$ )
- complexitatea timp **medie** pentru toate operațiile pe TD (adăugare, căutare, ștergere) este  $\Theta(1)$ .
  - **căutarea** unui element într-o TD poate necesita  $\Theta(n)$  în caz **defavorabil** (ca și căutarea în liste)
    - în practică, dispersia funcționează foarte bine
    - timpul **mediu** preconizat pentru căutarea este  $\Theta(1)$
- se definește o **funcție de dispersie** (*hash function*)  $d: U \rightarrow \{0, 1, \dots, m-1\}$ 
  - $d(c)$  este **valoarea de dispersie** a cheii  $c$
  - vom spune **C** se **dispersează** în locația  $d(c)$
- dacă două chei  $c_1$  și  $c_2$  se dispersează în aceeași locație, adică  $d(c_1) = d(c_2)$ , spunem că avem o **coliziune**
  - evitarea totală a coliziunilor este imposibilă
    - deoarece  $|U| > m$ , sigur există două chei care să fie în coliziune
  - minimizare numărului de coliziuni
    - printr-o alegere potrivită a funcției de dispersie

**Exemplu** În figura de mai jos, cheile sunt notate cu  $k$  (**keys**), iar funcția de dispersie prin  $h$  (**hashing function**).



- **dispersia perfectă** (*perfect hashing, perfect hash function*)

- fără coliziuni
  - când se cunosc cheile (mulțimea de chei este statică – ex. compilatoare)
- vom discuta în cursul 10
- cum se face **adăugarea unui nou element**  $e=(c, v)$ ?
  - se calculează locația de dispersie a cheii  $c$ ,  $i = d(c)$
  - dacă locația  $i$  este liberă, atunci se adaugă elementul la locația  $i$
  - dacă la locația  $i$  mai e memorat un alt element  $\Rightarrow$  **rezolvare coliziune**
    - 2 tipuri de metode de dispersie
      - **dispersia deschisă** (*open hashing*)
        - cheile sunt stocate în liste înlănțuite atașate celulelor unei TD.
      - **dispersia închisă** (*closed hashing*)
        - cheile sunt stocate în interiorul TD fără a utiliza liste înlănțuite.
    - 3 metode de rezolvare a coliziunilor
      - **prin liste independente (înlănțuire)**
        - dispersie deschisă
      - **prin liste întrepătrunse**
        - dispersie deschisă
      - **prin adresare deschisă**
        - dispersie închisă
  - **funcție de dispersie bună**
    - este ușor de calculat (folosește operații aritmetice simple) -  $\Theta(1)$
    - produce cât mai puține coliziuni.

### Interpretarea cheilor ca numere naturale

- Majoritatea funcțiilor de dispersie presupun universul cheilor din mulțimea numerelor naturale
- În cazul în care cheile nu sunt numere naturale, trebuie găsită o modalitate de a le interpreta ca numere naturale – o funcție care asociază fiecărei chei un număr natural (implementare **hashCode: TCheie → {0, 1, 2...}**)
  - identificatorul **pt** poate fi interpretat ca un număr în baza **128** –  $(pt)_{128} = 112 \cdot 128 + 116 = 14452$ .
  - pentru un sir de caractere putem considera suma codurilor ASCII ale caracterelor.
  - ...
- În cazul în care în container elementele sunt de tip **TElement** (nu au asociată o cheie - ex. mulțime, colecție), **hashCode: TElement → {0, 1, 2...}**
- Pp. în cele ce urmează că avem chei naturale.

### Functii de dispersie

- O funcție de dispersie bună satisfacă (aproximativ) *ipoteza dispersiei uniforme simple* (**Simple Uniform Hashing - SUH**): fiecare cheie se poate dispersa cu aceeași probabilitate în oricare din cele  $m$  locații.
  - $P(d(c) = j) = \frac{1}{m}, \forall j = 0, \dots, m-1 \quad \forall c \in U$ 
    - $P(d(c_1) = d(c_2)) = \frac{1}{m}, \quad \forall c_1, c_2 \in U$

- dacă  $P(c)$  este probabilitatea de a alege cheia  $c$ , atunci  $\sum_{c:d(c)=j} P(c) = \frac{1}{m} \forall j = 0, 1, \dots, m-1$ 
  - în general, nu se poate verifica această condiție, deoarece nu se cunoaște distribuția de probabilitate  $P$
  - dacă această ipoteză ar fi verificată, atunci se minimizează numărul de coliziuni
  - în practică se pot folosi tehnici euristice pentru a crea funcții de dispersie care să se comporte bine.

## I. Metoda diviziunii

- Dispersia prin diviziune
- $d(c) = c \bmod m$
- Experimental: valori bune pentru  $m$  sunt numerele prime nu prea apropiate de puteri exacte ale lui 2 (ex: 13,...)
- $m=13$ 
  - $c=63 \Rightarrow d(c)=11$
  - $c=26 \Rightarrow d(c)=0$
- ex: pentru a reține  $n=2000$  șiruri de caractere (1 caracter = 8 biți)
  - 3 elemente, în medie, într-o coliziune
  - $\Rightarrow m=701$  (apropiat de  $2000/3$ , nu e apropiat de o putere a lui 2)
  - $\Rightarrow d(c) = c \bmod 701$

## II. Metoda înmulțirii

- $d(c) = [m \cdot (c \cdot A \bmod 1)]$  unde " $c \cdot A \bmod 1$ " reprezintă partea fracționară a lui  $c \cdot A$  ( $c \cdot A - [c \cdot A]$ )
- Valoarea lui  $m$  nu e critică (în general este o putere a lui 2)
- Knuth: valoarea optimă pentru  $A$  este  $\frac{\sqrt{5}-1}{2} \approx 0.6180339887$  (golden ratio-1)
  - $m = 13, A = 0.6180339887$  (Knuth)
  - $c=63 \Rightarrow d(c)=[13 * \text{frac}(63 * A)]=12$
  - $c=52 \Rightarrow d(c)=[13 * \text{frac}(52 * A)]=1$
  - $c=129 \Rightarrow d(c)=[13 * \text{frac}(129 * A)]=9$

## III. Dispersia universală

- $c = \langle c_1, c_2, \dots, c_k \rangle$
- $d(c) = (\sum_{i=1}^k c_i \cdot x_i) \bmod m$  unde  $\langle x_1, x_2, \dots, x_k \rangle$  este o secvență de numere aleatoare fixate (selectate de-a lungul inițializării funcției de dispersie)
- apropiată de ipoteza SUH

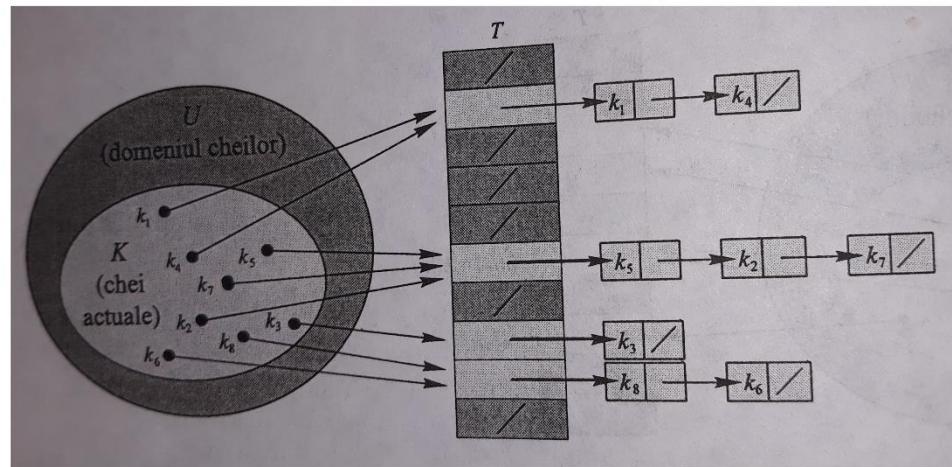
## Observație

- în cazul în care cheile nu sunt numere naturale, funcția de dispersie  $d$  (una din cele definite anterior) se definește nu pe cheia  $c$ , ci pe  $\text{hashCode}$ -ul acesteia
  - ex:  $d(c) = \text{hashCode}(c) \bmod m$

## A. Rezolvare coliziuni prin liste independente (înlăntuire) – SEPARATE CHAINING

- Elementele care se dispersează în aceeași locație (sunt într-o coliziune), vor fi puse într-o listă înlănțuită.
  - în general, alocare dinamică pentru memorarea înlănțuirilor
  - listele pot fi simplu sau dublu înlănțuite
- Locația  $j$  conține un pointer către capul listei înlănțuite a elementelor care se dispersează în locația  $j$  (dacă această listă e vidă, se memorează NIL).
- Operațiile sunt ușor de implementat.

**Exemplu** În figura de mai jos, cheile sunt notate cu  $k$  (**keys**), iar funcția de dispersie prin  $h$  (**hash function**).



Dacă  $m=10$ ,  $K=\{11, 21, 31, 5, 15, 7, 17, 27\}$ ,  $d(c)=c \bmod m$ , atunci

- $d(11)=d(21)=d(31)=1$
- $d(5)=d(15)=5$
- $d(7)=d(17)=d(27)=7$

și tabela tabelă va fi

0	
1	→ 11 → 21 → 31
2	
3	
4	
5	→ 5 → 15
6	
7	→ 7 → 17 → 27
8	
9	

## Reprezentare și operații

<b>TElement</b>	<b>Container</b>
$c: \text{TCheie}$	$m: \text{Întreg}$
$v: \text{Valoare}$	$l: \text{Listă}[0..m-1]$

- $d$  este funcția de dispersie,  $\mathbf{d}: \text{TCheie} \rightarrow \{\mathbf{0}, \mathbf{1} \dots \mathbf{m - 1}\}$
- pp. cheia are o singură valoare asociată
- Container poate fi, de ex., dicționar, mulțime, colecție.
  - în cazul mulțimii/colecției,  $\text{TCheie} = \text{TElement}$  și nu există valoare asociată cheii.

### **CAUTĂ** ( $C, ch$ )

// pre:  $C$  este un container reprezentat sub forma unei TD (coliziuni prin înlănțuire),  $ch$  este de tip //  $\text{TCheie}$   
 @ caută elementul cu cheia  $ch$  în lista  $C.l[d(ch)]$

### **ADAUGĂ** ( $C, e$ )

// pre:  $C$  este un container reprezentat sub forma unei TD (coliziuni prin înlănțuire),  $e$  este de tip //  $\text{TElement}$   
 @ se adaugă elementul  $e$  în capul listei înlănțuite  $C.l[d(e.c)]$

### **ȘTERGE** ( $T, ch$ )

// pre:  $C$  este un container reprezentat sub forma unei TD (coliziuni prin înlănțuire),  $ch$  este de tip //  $\text{TCheie}$   
 @ se șterge elementul cu cheia  $ch$  din lista înlănțuită  $C.l[d(ch)]$

## Observații

- Este posibil ca listele independente să fie memorate ordonat după cheie sau valoare
- Funcția de dispersie este considerată *bună* dacă listele au aproximativ aceeași lungime
- Dacă apar multe liste de vide sau liste prea lungi, se modifică  $m \Rightarrow$  redispersare (**rehashing**)

## Timp defavorabil pentru operații

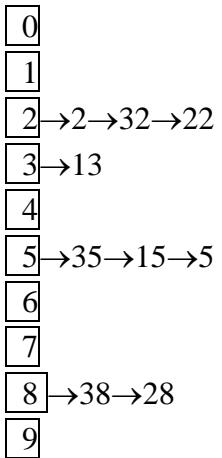
Pp  $n$  este numărul elementelor din container.

- **CAUTĂ –  $O(n)$** 
  - toate elementele se dispersează în aceeași locație ( $\Theta(n)$  – dacă elementul nu e găsit)
- **ADAUGĂ –  $\Theta(1)$** 
  - se poate adăuga la începutul listei înlănțuite
- **ȘTERGE – presupune**
  - (1) căutare nod în lista înlănțuită + (2) ștergere nod  $\Rightarrow O(n)$

## Exemplu

$m=10$ ,  $d(c)=c \bmod m$

c	5	15	13	22	28	35	38	32	2
d(c)	5	5	3	2	8	5	8	2	2



## Iterator

- dacă listele sunt simplu înlățuite cu alocare dinamică
- iteratorul va memora
  - o referință  $c$  către containerul reprezentat folosind o TD cu coliziuni prin liste independente
  - poziția curentă  $pozCurrent$  din tabelă (indică lista înlățuită iterată)
  - adresa unui nod (pointer)  $current$  din lista înlățuită de la poziția  $pozCurrent$

Telement	Nod	Container	IteratorContainer
$c:TCheie$	$e:TElement$	$m:\text{Întreg}$	$c:Container$ //referință (în implementare)
$v:TValoare$	$urm:\uparrow\text{Nod}$	$l:\uparrow\text{Nod} [0..m-1]$	$pozCurrent:\text{Intreg}$ $current:\uparrow\text{Nod}$

Operațiile pe iterator sunt descrise în Pseudocod, în continuare.

Pe lângă operațiile uzuale ale iteratorului (*creează, prim, valid, element, următor*), avem nevoie de o operație auxiliară **deplasare** care, dacă lista de la locația curentă  $pozCurrent$  a fost iterată până la final ( $current$  devine invalid), deplasează  $pozCurrent$  pe următoarea locație din tabelă care conține o listă nevidă și poziționează  $current$  pe primul nod din această listă.

- în exemplul anterior, dacă  $pozCurrent=3$  și s-a terminat de iterat lista de la poziția 3, mută  $pozCurrent$  pe 5, iar  $current$  va indica 35.
- această operație NU va fi în interfața iteratorului (secțiunea publică), ci în implementare (secțiunea privată)

## **Subalgoritm deplasare (i)** este

{pre:  $i: \text{IteratorContainer}$ }

{post: deplasează iteratorul pe prima listă nevidă care urmează după locația  $pozCurrent$ }

{incrementăm  $pozCurrent$  cât timp nu s-a epuizat tabela și lista de la poziția  $pozCurrent$  e vidă}

**CâtTimp** ( $i.pozCurent < i.c.m$ )  $\wedge$  ( $i.c.l[i.pozCurent] = \text{NIL}$ ) **execută**

$i.pozCurent = i.pozCurent + 1$

**SfCâtTimp**

{dacă nu s-a epuizat tabela}

**Dacă**  $i.pozCurent < i.c.m$  **atunci**

$i.curent \leftarrow i.c.l[i.pozCurent]$

**SfDacă**

**SfSubalgoritm**

**Subalgoritm creează** ( $i, c$ ) este

$i.c \leftarrow c$

$i.pozCurent \leftarrow 0$

{căutăm prima listă nevidă, pentru a poziționa iteratorul}

deplasare( $i$ )

**SfSubalgoritm**

**Subalgoritm prim** ( $i$ ) este

$i.pozCurent \leftarrow 0$

deplasare( $i$ )

**SfSubalgoritm**

**Funcția valid( $i$ )** este

{locația curent iterată nu depășește numărul de locații din tabelă și nodul curent este valid }

$\text{valid} \leftarrow (i.pozCurent < i.c.m) \wedge (i.curent \neq \text{NIL})$

**SfFuncție**

**Subalgoritm element** ( $i, e$ ) este

{pre:  $i$  este valid}

$e \leftarrow [i.curent].e$

**SfSubalgoritm**

**Subalgoritm urmator** ( $i$ ) este

{pre:  $i$  este valid}

$i.curent \leftarrow [i.curent].urm$

{dacă s-a terminat de iterat lista curentă, căutăm prima listă nevidă, pentru a repozitiona iteratorul}

**Dacă**  $i.curent = \text{NIL}$  **atunci**

$i.pozCurent = i.pozCurent + 1$

deplasare( $i$ )

**SfDacă**

**SfSubalgoritm**

### Observație

- complexitatea iterării unui container cu  $n$  elemente, reprezentat folosind o TD cu  $m$  locații și liste independente este  $\Theta(n + m)$

În directorul asociat cursului 8, găsiți implementarea parțială, în limbajul C++, a containerului **Colecție** (reprezentarea este sub forma unei TD în care coliziunile sunt reprezentate prin înlățuire).

## Analiza dispersiei cu înlățuire

### Notății și presupuneri

- $\alpha = \frac{n}{m}$  factorul de încărcare al tabelei (numărul mediu de elemente memorate într-o înlățuire)
- Pp. că timpul de calcul al funcției de dispersie este  $\Theta(1)$  (!! la timpul de căutare se adaugă și timpul de calcul al funcției de dispersie)
- La căutare apar 2 cazuri
  - Căutare **cu succes** (găsim elementul)
  - Căutare **fără succes** (nu găsim elementul)

**Teorema 1.** Într-o TD în care coliziunile sunt rezolvate prin înlățuire, în ipoteza dispersiei uniforme simple (SUH), o căutare **fără succes**, necesită, în **medie**, un timp  $\Theta(1 + \alpha)$ .

În ipoteza SUH, fiecare listă are aceeași lungime,  $\alpha$ , iar o cheie se poate dispersa, cu aceeași probabilitate, în orice locație (poate fi în oricare dintre liste)

- (1) căutarea fără succes necesită iterarea unei liste  $\Rightarrow \alpha$
- (2) calcul funcției de dispersie  $\Rightarrow 1$
- Din (1) și (2)  $\Rightarrow \Theta(1 + \alpha)$ .

**Teorema 2.** Într-o TD în care coliziunile sunt rezolvate prin înlățuire, în ipoteza dispersiei uniforme simple (SUH), o căutare **cu succes**, necesită, în **medie**, un timp  $\Theta(1 + \alpha)$ .

### Intuiție

- probabilitatea ca o cheie să se disperseze într-una din liste este  $\frac{1}{m}$
- în lista de pe poziția  $j$ , elementul poate fi găsit după 1, 2, ...,  $\alpha$  pași  $\Rightarrow$  timpul mediu este aproximativ

$$\frac{1}{m} \sum_{j=1}^m \sum_{i=1}^{\alpha} \frac{i}{\alpha} \in \Theta(1 + \alpha)$$

## **CONCLUZII**

- Dacă  $n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1) \Rightarrow$  căutarea necesită, în **medie**, timp constant  $\Theta(1)$
- Adăugarea necesită  $\Theta(1)$
- Dacă listele sunt dublu înlățuite atunci ștergerea unui nod se poate face în  $\Theta(1)$

$\Rightarrow$  TOATE OPERAȚIILE (adăugare, căutare, ștergere) POT FI EXECUTATE ÎN MEDIE ÎN  $\Theta(1)$

### Observații

- Pentru memorarea listelor independente se pot folosi și arbori echilibrați, ceea ce va reduce complexitatea timp în caz defavorabil la căutare de la  $\Theta(n)$  la  $\Theta(\log_2 n)$ .

- Rezolvarea colizunilor prin liste independente se mai numește și **dispersie deschisă** (*open hashing*) sau **adresare închisă** (*closed addressing*)
  - elemente sunt memorate în afara tabelei.

## PROBLEME

1. Presupunem că folosim o funcție de dispersie aleatoare  $d$  pentru a dispersa  $n$  chei distințe într-o tabelă  $T$  de dimensiune  $m$ . Care este numărul mediu de coliziuni? (cardinalul probabil al mulțimii  $\{(x, y) \in T_{Cheie} \times T_{Cheie} : d(x) = d(y)\}$ )
2. Presupunem că folosim o TD în care coliziunile sunt rezolvate prin înlănțuire (liste independente), dar fiecare listă este ordonată după cheie. Care va fi timpul de execuție pentru **căutare** (cu succes, fără succes), **adăugare** și **ștergere**?
3. Arătați că dacă  $|U| > n \cdot m$ , atunci există o submulțime a lui  $U$  de mărime  $n$  ce conține chei care se dispersează toate în aceeași locație, astfel încât timpul de căutare pentru dispersia cu înlănțuire, în cazul cel mai defavorabil, este  $\Theta(n)$ .

**Analysis of Open Addressing:** We'll look at the complexity of `INSERT` since, in open addressing, searching for a key  $k$  that is in the table takes exactly as long as it took to insert  $k$  in the first place. The time to search for an element  $k$  that does not appear in the table is the time it would take to insert that element in the table. You should check why these two statements are true.

It's not hard to come up with worst-case situations where the above types of open addressing require  $\Theta(n)$  time for `INSERT`. On average, however, it can be very difficult to analyze a particular type of probing. Therefore, we will consider the following situation: there is a hash table with  $m$  locations that contains  $n$  elements and we want to insert a new key  $k$ . We will consider a random probe sequence for  $k$ —that is, its probe sequence is equally likely to be any permutation of  $(0, 1, \dots, m-1)$ . This is a realistic situation since, ideally, each key's probe sequence is as unrelated as possible to the probe sequence of any other key.

Let  $T$  denote the number of probes performed in the `INSERT`. Let  $A_i$  denote the event that every location up until the  $i$ -th probe is occupied. Then,  $T \geq i$  iff  $A_1, A_2, \dots, A_{i-1}$  all occur, so

$$\begin{aligned}\Pr(T \geq i) &= \Pr(A_1 \cap A_2 \cap \dots \cap A_{i-1}) \\ &= \Pr(A_1) \Pr(A_2 | A_1) \Pr(A_3 | A_1 \cap A_2) \dots \Pr(A_{i-1} | A_1 \cap \dots \cap A_{i-2})\end{aligned}$$

For  $j \geq 1$ ,

$$\Pr(A_j | A_1 \cap \dots \cap A_{j-1}) = (n - j + 1)/(m - j + 1),$$

because there are  $n - j + 1$  elements that we haven't seen among the remaining  $m - j + 1$  slots that we haven't seen. Hence,

$$\Pr(T \geq i) = n/m \cdot (n - 1)/(m - 1) \cdots (n - i + 2)/(m - i + 2) \leq (n/m)^{i-1} = a^{i-1}. \quad (7)$$

Now we can calculate the expected value of  $T$ , or the average-case complexity of insert:

$$\begin{aligned}
E(T) &= \sum_{i=0}^{m-1} i \Pr(T = i) \\
&\leq \sum_{i=1}^{\infty} i \Pr(T = i) \\
&= \sum_{i=1}^{\infty} i(\Pr(T \geq i) - \Pr(T \geq i + 1)) \\
&= \sum_{i=1}^{\infty} \Pr(T \geq i) \quad \text{by telescoping} \\
&\leq \sum_{i=1}^{\infty} a^{i-1} \quad \text{by (7)} \\
&= \sum_{i=0}^{\infty} a^i \\
&= \frac{1}{1-a}
\end{aligned}$$

Remember that  $a < 1$  since  $n < m$ . The bigger the load factor, however, the longer it takes to insert something. This is what we expect, intuitively.

Suppose there are  $N$  keys in the table at the start of this program, and let

$$\alpha = N/M = \text{load factor of the table.} \quad (14)$$

Then the average value of  $A$  in an unsuccessful search is obviously  $\alpha$ , if the hash function is random; and exercise 39 proves that the average value of  $C$  in an unsuccessful search is

$$C'_N = 1 + \frac{1}{4} \left( \left( 1 + \frac{2}{M} \right)^N - 1 - \frac{2N}{M} \right) \approx 1 + \frac{e^{2\alpha} - 1 - 2\alpha}{4}. \quad (15)$$

Thus when the table is half full, the average number of probes made in an unsuccessful search is about  $\frac{1}{4}(e+2) \approx 1.18$ ; and even when the table gets completely full, the average number of probes made just before inserting the final item will be only about  $\frac{1}{4}(e^2+1) \approx 2.10$ . The standard deviation is also small, as shown in exercise 40. These statistics prove that *the lists stay short even though the algorithm occasionally allows them to coalesce*, when the hash function is random. Of course  $C$  can be as high as  $N$ , if the hash function is bad or if we are extremely unlucky.

In a successful search, we always have  $A = 1$ . The average number of probes during a successful search may be computed by summing the quantity  $C + A$  over the first  $N$  unsuccessful searches and dividing by  $N$ , if we assume that each key is equally likely. Thus we obtain

$$\begin{aligned} C_N &= \frac{1}{N} \sum_{0 \leq k < N} \left( C'_k + \frac{k}{M} \right) = 1 + \frac{1}{8} \frac{M}{N} \left( \left( 1 + \frac{2}{M} \right)^N - 1 - \frac{2N}{M} \right) + \frac{1}{4} \frac{N-1}{M} \\ &\approx 1 + \frac{e^{2\alpha} - 1 - 2\alpha}{8\alpha} + \frac{\alpha}{4} \end{aligned} \quad (16)$$

as the average number of probes in a random successful search. Even a full table will require only about 1.80 probes, on the average, to find an item! Similarly (see exercise 42), the average value of  $C_1$  is approximately 1.

# TABELA DE DISPERSIE

- continuare -

## C. Rezolvare coliziuni prin *adresare deschisă* – OPEN ADDRESSING

- Toate elementele sunt memorate în interiorul tabelei, nu există liste memorate în afara tabelei.
  - Se mai numește și **dispersie închisă** (*closed hashing*).
    - **rezolvarea coliziunilor prin liste întrepătrunse** (*closed hashing*) este o combinație între *open addressing* și *separate chaining* (*open hashing*)
  - Fiecare intrare în tabelă conține fie un element al containerului, fie un marcat pentru locație liberă (ex. NIL).
  - Nu se folosesc pointeri pentru înlățuiri.
  - Factorul de încărcare este subunitar  $\alpha < 1$ , altfel tabela este plină
  - Dezavantaj: tabela se poate umple ( $\alpha = 1$ ). Soluție: se crește  $m$ , ceea ce presupune redispersarea elementelor.
  - Avantaj: spațiul de memorie suplimentar (nu se memorează pointeri) oferă tabelei un număr mai mare de locații pentru același spațiu de memorie, putând rezulta coliziuni mai puține și acces rapid.
  - Secvența de locații care se examinează nu se determină folosind **pointeri**, ci se **calculează**
  - La **adăugare** în tabelă, se examinează succesiv locațiile, până se găsește o locație liberă în care să se adauge cheia (elementul). În loc să fie fixată ordinea de verificare a tabelei (ex: 0,1,2,...,m-1, ca la vectori) care ar necesita timp  $\Theta(m)$ , secvența de poziții examineate depinde de cheia (elementul) care se inserează.
- 
- Se extinde funcția de dispersie  $d: U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$ 
    - al doilea argument al funcției se numește **număr de verificare**
  - Pentru o cheie  $c \in U$  secvența  $\langle d(c, 0), d(c, 1), \dots, d(c, m-1) \rangle$  se numește **secvență de verificare** a cheii  $c$

### Cerintă

- secvența de verificare a oricărei chei  $c \in U \langle d(c, 0), d(c, 1), \dots, d(c, m-1) \rangle$  trebuie să fie o permutare a mulțimii  $\langle 0, 1, \dots, m-1 \rangle$ 
  - pentru ca la adăugarea oricărei chei să fie verificate toate locațiile din tabelă

### Ipoteza dispersiei uniforme simple (SUH)

- Pentru orice cheie  $c \in U$ , permutarea  $\langle d(c, 0), d(c, 1), \dots, d(c, m-1) \rangle$  poate să apară sub forma oricărei permutări a  $\langle 0, 1, \dots, m-1 \rangle$

Sunt 3 metode pentru a stabili secvența de verificare a unei chei  $c$ , a.î. să se asigure faptul că  $c \in U \langle d(c, 0), d(c, 1), \dots, d(c, m-1) \rangle$  este o permutare a mulțimii  $\langle 0, 1, \dots, m-1 \rangle$ . Acestea vor fi descrise, în continuare.

### C.1. Verificare liniară – LINEAR PROBING

$$d(c, i) = (d'(c) + i) \bmod m \quad \forall i = 0, 1, \dots, m - 1$$

- $d': U \rightarrow \{0, \dots, m - 1\}$  este o funcție de dispersie uzuală (ex:  $d'(c) = c \bmod m$ )

- Fiind dată o cheie  $c$ , secvența ei de verificare este  $\langle d'(c), d'(c) + 1, d'(c) + 2, \dots, m - 1, 0, 1, \dots, d'(c) - 1 \rangle$
- Problema: **grupare primară** – se formează șiruri lungi de locații ocupate, crescând timpul mediu de căutare

### C.2. Verificare pătratică – QUADRATIC PROBING

$$d(c, i) = (d'(c) + c_1 \cdot i + c_2 \cdot i^2) \bmod m \quad \forall i = 0, 1, \dots, m - 1$$

$d': U \rightarrow \{0, \dots, m - 1\}$  este o funcție de dispersie uzuală (ex:  $d'(c) = c \bmod m$ ),  $c_1 \neq 0$  și  $c_2 \neq 0$  sunt constante auxiliare fixate la inițializarea funcției de dispersie.

- Constantele  $c_1 \neq 0$  și  $c_2 \neq 0$  se pot determina euristic
  - Pentru a asigura faptul că secvența de verificare este o permutare  $\{0, \dots, m - 1\}$ 
    - $m$  se alege putere a lui 2 și  $c_1 = c_2 = 0.5$
    - $m$  se alege număr prim de forma  $4 \cdot k + 3$ ,  $d'(c) = c \bmod m$  și  $c_1 = 0$ ,  $c_2 = (-1)^i$
- Fiind dată o cheie  $c$ , prima poziție examinată este  $d'(c)$ , după care următoarele poziții examineate sunt decalate cu cantități ce depind într-o manieră pătratică de locația anterior examinată.
- Problema: **grupare secundară** – dacă 2 chei au aceeași poziție de start a verificării, atunci secvența lor de verificare coincide (dacă  $d(c', 0) = d(c'', 0) \Rightarrow d(c', i) = d(c'', i) \quad \forall i = 0, 1, \dots, m - 1$ )
- Experimental: funcționează **mai bine** decât **verificarea liniară**

### C.3. Dispersia dublă – DOUBLE HASHING

$$d(c, i) = (d_1(c) + i \cdot d_2(c)) \bmod m \quad \forall i = 0, 1, \dots, m - 1$$

$d_1$  și  $d_2$  sunt funcții de dispersie aleatoare.

- Este considerată una dintre cele mai bune metode disponibile pentru adresarea deschisă
- Fiind dată o cheie  $c$ , prima poziție examinată este  $d_1(c)$ , după care următoarele poziții examineate sunt decalate față de poziția anterioară cu  $d_2(c) \bmod m$ .
- $d_2(c)$  și  $m$  să fie prime între ele pentru a fi parcursă întreaga tabelă
- **Exemplu**
  - $m$  prim
  - $d_1(c) = c \bmod m \quad d_2(c) = (1 + c \bmod m')$
  - $m'$  se alege de obicei  $m-1$  sau  $m-2$
- Performanța dispersiei duble apare ca fiind foarte apropiată de performanța schemei ideale a dispersiei uniforme ( $\Theta(m^2)$ ) secvențe de verificare posibile pentru o cheie)
  - în cazurile C1 și C2, numărul secvențelor de verificare posibile pentru o cheie e doar  $\Theta(m)$

## Presupuneri și notări:

- Pp. că în container memorăm doar chei
- O locație din tabelă va conține:
  - NIL (constantă simbolică) – dacă locația e liberă (nu conține o cheie)
  - O cheie din container

Reprezentarea containerului folosind o TD cu adresare deschisă

### **Container**

m: Intreg {nr.de locații din tabelă}  
 ch: TCheie[0..m-1] {cheile din container}  
 d: TFunctie {funcția de dispersie asociată}

## ADĂUGARE

Dacă adăugăm o cheie  $c$ , determinăm locația la care ar trebui memorată în tabelă ( $i=d(c)$ ), după care vom avea două situații

- Locația  $i$  este liberă (NIL, în convenția noastră)  $\Rightarrow$  caz favorabil, memorăm cheia
- Locația  $i$  nu este liberă  $\Rightarrow$  avem coliziune
  - verificăm, pe rând, locațiile din tabelă, în ordinea dată de secvența de verificare  
 $< d(c, 0), d(c, 1), \dots, d(c, m - 1) >$
  - dacă găsim o locație liberă, adăugăm
  - dacă toate locațiile din secvența de verificare sunt ocupate  $\Rightarrow$  tabela este plină

## EXEMPLE

### 1. Adresare deschisă cu verificare liniară

- $m=10$
- $d(c, i) = (d'(c) + i) \bmod m \quad \forall i = 0, 1, \dots, m - 1$
- $d'(c)=c \bmod m$

c	5	15	13	22	20	35	30	32	2
$d'(c)$	5	5	3	2	0	5	0	2	2

### Pas 1. Inițializare

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	NIL									

### Pas 2. Adăugăm cheia 5

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	NIL	NIL	NIL	NIL	NIL	5	NIL	NIL	NIL	NIL

### Pas 3. Adăugăm cheia 15

- Secvența de verificare a cheii este  $<5, 6, 7, 8, 9, 0, 1, 2, 3, 4>$
- Prima locație liberă din secvență este locația 6, acolo se va adăuga

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	NIL	NIL	NIL	NIL	NIL	5	15	NIL	NIL	NIL

....

La final, tabela va fi

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	20	30	22	13	32	5	15	35	2	NIL

## 2. Adresare deschisă cu verificare pătratică

- $m=8$
- $d(c, i) = (d'(c) + c_1 \cdot i + c_2 \cdot i^2) \bmod m \quad \forall i = 0, 1, \dots, m - 1$
- $d'(c)=c \bmod m$
- $c_1 = c_2 = 0.5$

c	5	15	13	2	0	32
$d'(c)$	5	7	5	2	0	0

### Pas 1. Inițializare

Indice	0	1	2	3	4	5	6	7
Cheie	NIL							

### Pas 2, 3. Adăugăm cheile 5, 15

Indice	0	1	2	3	4	5	6	7
Cheie	NIL	NIL	NIL	NIL	NIL	5	13	15

### Pas 4 Adăugăm cheia 13

- Secvența de verificare a cheii este  $<5, 6, \dots>$
- Prima locație liberă din secvență este locația 6, acolo se va adăuga

.....

La final, tabela va fi

Indice	0	1	2	3	4	5	6	7
Cheie	0	32	2	NIL	NIL	5	13	15

### 3. Adresare deschisă cu dispersie dublă

- $m=13$
- $d(c, i) = (d_1(c) + i \cdot d_2(c)) \bmod m \quad \forall i = 0, 1, \dots, m-1$
- $d_1(c)=c \bmod m, d_2(c)=1 + c \bmod (m-2)$

<b>c</b>	79	69	96	14
<b>d<sub>1</sub>(c)</b>	1	4	5	1
<b>d<sub>2</sub>(c)</b>	3	4	9	4

Adaugăm cheile 79, 69, 96 și tabela devine

<b>Indice</b>	0	1	2	3	4	5	6	7	8	9	10	11	12
<b>Cheie</b>	NIL	79	NIL	NIL	69	96	NIL						

Pentru cheia 14, care e în coliziune, secvența de verificare e 1, 5, 9, ....

- prima poziție liberă e 9, adăugăm

Tabela finală e

<b>Indice</b>	0	1	2	3	4	5	6	7	8	9	10	11	12
<b>Cheie</b>	NIL	79	NIL	NIL	69	96	NIL	NIL	NIL	14	NIL	NIL	NIL

#### Subalgoritmul ADAUGĂ ( $c, ch$ ) este

{ $c$ :Container,  $ch$ :TCheie}

$i \leftarrow 0$  {numărul de verificare}

$gasit \leftarrow$  fals {nu am găsit poziția de adăugare}

repetă

$j \leftarrow c.d(ch, i)$  {locația de verificat}

(\*)            dacă  $c.ch[j] = \text{NIL}$  atunci

$c.ch[j] \leftarrow ch$  {memorez cheia}

$gasit \leftarrow$  adev {am găsit poziția unde putem adăuga}

altfel

$i \leftarrow i+1$  {căutăm mai departe pe secvența de verificare}

sfdacă

până\_când ( $i=c.m$ ) sau ( $gasit$ )

dacă  $i=c.m$  atunci {tabela e plină}

@ depășire tabelă

Sfdacă

#### sfADAUGĂ

### CĂUTARE

- pp. că vrem să căutăm cheia  $c$

- o căutăm în ordinea dată de secvență de verificare a cheii  $\langle d(c, 0), d(c, 1), \dots, d(c, m - 1) \rangle$  până să dăm de o locație liberă (marcată cu NIL)
- dacă găsim cheia undeava pe secvență de verificare  $\Rightarrow$  căutare cu succes, altfel căutare fără succes
- în exemplul 1
  - o căutăm **35 (cu succes)**: căutăm în ordinea 5, 6, 7
    - o găsim pe poziția 7
  - o căutăm **45 (fără succes)**: căutăm în ordinea 5, 6, 7, 8, NIL
    - nu găsim cheia pe secvență de verificare

**Funcția CAUTĂ** ( $c, ch$ ) este

{ $c$ :Container,  $ch$ :TCheie}

$i \leftarrow 0$  {numărul de verificare}

$gasit \leftarrow \text{fals}$  {nu am găsit cheia}

repetă

$j \leftarrow c.d(ch, i)$  {locația de verificat}

dacă  $c.ch[j] = ch$  atunci {am găsit cheia}

$gasit \leftarrow \text{adev}$

altfel

$i \leftarrow i + 1$  {căutăm mai departe pe secvență de verificare}

sfdacă

până\_când ( $c.ch[j] = \text{NIL}$ ) sau ( $i = c.m$ ) sau ( $gasit$ )

**CAUTĂ**  $\leftarrow gasit$

**sfCAUTĂ**

## **STERGERE**

Ștergerea unei chei  $c$

- identificăm locația  $i$  la care este memorată cheia (ca și la căutare)
- nu putem marca locația  $i$  cu NIL (ca și cum ar fi liberă), deoarece ar fi imposibil să mai accesăm orice cheie a cărei inserare a verificat locația  $i$  și a găsit-o liberă

Sunt două soluții la ștergere

1. O soluție este de a marca la locația  $i$  o valoare specială, ȘTERS (în loc de NIL)
  - vom modifica ADAUGĂ astfel încât să trateze locațiile marcate cu ȘTERS ca și cum ar fi libere
    - linia marcată cu (\*) în ADAUGĂ o vom înlocui cu
      - dacă ( $c.ch[j] = \text{NIL}$ ) sau ( $c.ch[j] = \text{ȘTERS}$ ) atunci
  - nu este necesar să modificăm CAUTĂ
2. Prin deplasări ale datelor, astfel încât să nu mai existe riscul de a nu regăsi o cheie

## **Ștergere prin deplasări de date**

Ilustrăm ideea ștergerii prin deplasări de date, presupunând verificarea liniară a tabelei.

- vrem să ștergem cheia  $c$
- o localizăm (căutând pe secvența de verificare)
- fie  $i$  locația pe care se află cheia  $c$  și care trebuie ștearsă

Va trebui să ne asigurăm că ștergerea locației  $i$  (marcarea acesteia cu NIL) nu afectează regăsirea cheilor care au fost memrate pornind de la locația  $i$  (la adăugare, au găsit locația ocupată).

**Pas 1.** Luăm, pe rând, locațiile în ordinea secvenței de verificare  $j = i+1, i+2, \dots, m-1, 0, \dots$  până la o zonă liberă

- Fie  $p$  locația unde trebuia memorată data (cheia, în cazul nostru), de la locația  $j$ 
  - $p = d'(ch[j])$
  - dacă am șterge cheia de la locația  $i$ , am putea regăsi cheia de la locația  $j$ , pornind de la  $p$  (iterând pe secvența de verificare, fără a da de o locație liberă)?
- Avem de tratat două cazuri, fiecare caz cu 3 subcazuri

### 1. $i < j$

1a)  $0 \leq p \leq i$

0 .....  $p$  .....  $i$  .....  $j$  .....  $m-1$

➤ (\*) se mută data (cheia, în cazul nostru) de la locația  $j$  la locația  $i$  și se continuă cu ștergerea locației  $j$

- $ch[i] \leftarrow ch[j]$
- $i \leftarrow j$

➤ Salt la **Pas 1.**

1b)  $i < p \leq j$

0 .....  $i$  .....  $p$  .....  $j$  .....  $m-1$

➤ nu e necesară mutare de date (ștergerea lui  $i$  nu afectează)

➤ Salt la **Pas 1.**

1c)  $j < p \leq m-1$

0 .....  $i$  .....  $j$  .....  $p$  .....  $m-1$

➤ se efectuează mutarea de date (\*) de la 1a)

➤ Salt la **Pas 1.**

### 2. $j < i$

2a)  $0 \leq p \leq j$

0 .....  $p$  .....  $j$  .....  $i$  .....  $m-1$

➤ nu e necesară mutare de date (ștergerea lui  $i$  nu afectează)

➤ Salt la **Pas 1.**

2b)  $j < p \leq i$

0 .....  $j$  .....  $p$  .....  $i$  .....  $m-1$

- se efectuează mutarea de date (\*) de la **1a**)
- Salt la **Pas 1**.

**2c)**  $i < p \leq m-1$

0 .....	$j$ .....	$i$ .....	$p$ .....	$m-1$
---------	-----------	-----------	-----------	-------

- nu e necesară mutare de date (ștergerea lui  $i$  nu afectează)
- Salt la **Pas 1**.

**Pas 2.** La încheierea structurii repetitive de la **Pas 1**, se poate șterge cheia de locația de la  $i$ .

**Exemplu** Considerăm exemplul 1 - **adresare deschisă cu verificare liniară**

- $m=10$
- $d(c, i) = (d'(c) + i) \bmod m \quad \forall i = 0, 1, \dots, m-1$
- $d'(c)=c \bmod m$

c	5	15	13	22	20	35	30	32	2
$d'(c)$	5	5	3	2	0	5	0	2	2

Vrem să ștergem cheia 5.

Tabela inițială este

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	20	30	22	13	32	5	15	35	2	NIL

i	j	Locația j liberă?	p	Caz
5	6	NU	5	1a) ⇒ mutare date
6	7	NU	5	1a) ⇒ mutare date
7	8	NU	2	1a) ⇒ mutare date
8	9	DA ⇒ STOP	-	<b>Pas 2</b> , se șterge cheia de la $i$

Tabela finală va fi

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	20	30	22	13	32	15	35	2	NIL	NIL

**Iteratorul** pe un container reprezentat folosind o TD cu adresare deschisă este simplu de implementat

- se iterează vectorul asociat, iterând doar pe pozițiile pe care e memorată o cheie diferită de NIL (în convenția noastră).
- $\Theta(m)$

**În directorul asociat cursului, găsiți implementarea parțială a containerului Colecție reprezentat folosind o TD cu adresare deschisă și verificare liniară.**

## Analiza dispersiei cu adresare deschisă

1. Teorema. Într-o TD cu adresare deschisă, în ipoteza dispersiei uniforme simple (SUH), cu factor de încărcare  $\alpha = \frac{n}{m} < 1$  numărul mediu de verificări este CEL MULT

**CORMEN, THOMAS H. - LEISERSON, CHARLES - RIVEST, RONALD R.: Introducere în algoritmi.**  
**Cluj-Napoca: Editura Computer Libris Agora, 2000.**

- $\frac{1}{1-\alpha}$  pentru adăugare și căutare fără succes
- $\frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}$  pentru căutare cu succes

Demonstrațiile sunt schițate și în directorul asociat cursului, subdirectorul **Demonstratii Complexitatii - TD Adresare deschisă**.

## **CONCLUZII**

- Dacă  $\alpha$  e constant  $\Rightarrow \Theta(1)$  în medie pentru operații
- Caz defavorabil  $O(n)$

## Observație

- în cazul în care se folosește vector dinamic pentru implementarea tabelei, analiza anterioară a complexităților este valabilă pentru cazul **amortizat**
- deși, din perspectivă teoretică, adresarea deschisă e mai performantă decât cea închisă, în bibliotecile existente (Java, STL), TD sunt implementate prin liste independente
  - Java – *HashTable*
    - se poate preciza, prin constructor, capacitatea inițială a tabelei și factorul de încărcare
    - implicit  $m=11$ , factorul de încărcare maxim e  $\alpha=0.75$
    - redispersare dacă se depășește factorul de încărcare implicit
  - STL - *unordered\_map/set*
    - implicit  $m=8$ , factorul de încărcare maxim e  $\alpha=1$
    - redispersare dacă se depășește factorul de încărcare implicit

## PROBLEME

1. Considerând o tabelă de dispersie cu adresare deschisă, scrieți un algoritm pentru operația de **ștergere** și modificați operația **adăugă** și **caută** pentru a încorpora valoarea specială **ȘTERS**.
2. Se consideră o tabelă de dispersie cu adresare deschisă, cu dispersie uniformă și factor de încărcare 0.5. Dați margini superioare pentru numărul mediu de verificări într-o căutare cu succes și o căutare fără succes.

## 2. DISPERSIA PERFECTĂ (PERFECT HASHING)

- Scop - să nu existe coliziuni.
  - Cât de mare să fie tabela încât să fim siguri că nu sunt coliziuni?
  - Dacă  $m=N^2$ , atunci tabela este fără coliziuni cu probabilitatea cel puțin 0.5.
    - $N$  – numărul de chei din container
  - Impractic.
- Soluție - *Dispersia perfectă (perfect hashing)*
  - Doar dacă avem o colecție STATICĂ de chei (nu se adaugă chei)
  - Se folosește o TD de dimensiune  $N$  (tabela **primară**)
  - În locul listelor independente se folosește o altă TD (tabela **secundară**)
  - Tabela secundară de la o locație  $i$  se va construi cu dimensiunea  $n_i^2$  unde  $n_i$  este numărul de elemente din acea tabelă (numărul de coliziuni de la locația  $i$ ).
  - Tabela secundară se va construi cu o altă funcție de dispersie și va fi reconstruită până nu va avea coliziuni.
  - Se poate demonstra că spațiul total de memorare pentru tabelele secundare este cel mult  $2*N \Rightarrow O(N)$ .
- Fie  $p$  numărul prim mai mare decât cea mai mare cheie.
- Funcțiile de dispersie se aleg dintr-o familie de *funcții de dispersie universală*.
  - $d_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m$
  - $1 \leq a \leq p-1, 0 \leq b \leq p-1$  ( $a$  și  $b$  selectate aleator la inițializarea funcției de dispersie)
  - $m = N$
- Performanța în caz **defavorabil** este  $\theta(1)$  (se caută cel mult 2 poziții – cea din TD principală și secundară)

### EXEMPLU

- 15 litere: I, N, S, X, E,....
- $N=m=15$
- Fiecărei litere îi asociem ca *hashCode* numărul de ordine al literei în alfabet
- Dacă  $a=3$  și  $b=2$  (alese aleator)
- $p=29$

Litera	I	N	S	X	E
<i>hashCode</i>	9	14	19	24	5
<i>d(hashCode)</i>	0	0	1	1	2

- **Coliziuni**
  - poziția 0 – I, N
  - poziția 1 – S, X

- poziția 2 – E
- ...
- Pentru pozițiile unde nu avem coliziuni (ex. poziția 2) avem o TD secundară cu un singur element și  $d(x)=0$
- Pentru pozițiile cu 2 elemente, vom avea o TD secundară cu 4 elemente și diferite funcții de dispersie, alese din același *univers*, cu diferite valori aleatoare pentru  $a$  și  $b$ .
- De ex., pentru poziția 0, putem defini  $a=4$  și  $b=11$  și vom avea
  - $d(I)=d(9)=2, d(N)=d(14)=1$
- Pentru poziția 1, să pp. că avem  $a=5$  și  $b=2$ .
  - $d(S)=d(19)=2, d(X)=d(24)=2 \Rightarrow$  coliziune
  - Alegem alte valori pentru  $a$  și  $b$  – de ex.  $a=2$  și  $b=13$ . Vom avea
    - $d(S)=d(9)=2, d(X)=d(14)=3$

### 3. ALTE VARIANTE DE DISPERSIE

#### 1. Dispersia Cuckoo (*Cuckoo hashing*)

- Se folosesc 2 TD cu două funcții de dispersie diferite
  - Fiecare tabelă e mai mult de jumătate *goală*
- Se poate garanta că un element va fi fie în prima, fie în a doua tabelă.
- **Căutarea și ștergerea** sunt simple (elementul se va localiza în una din cele 2 TD)
- **Inserarea unei chei  $c$** 
  - Se încearcă adăugarea în prima tabelă. Dacă e liber, se adaugă.
  - Dacă poziția în prima tabelă e ocupată de cheia  $c'$ , se scoate  $c'$  din prima tabelă, se adaugă noul element  $c$ . Elementul scos din prima tabelă  $c'$  se va adăuga în a doua. Dacă poziția în a doua tabelă e ocupată de  $c''$ , se va scoate acel element (în locul său se va adăuga elementul  $c'$  din prima tabelă) și  $c''$  se va adăuga în prima tabelă. Se va repeta procesul până se va obține o poziție liberă. Dacă se revine în aceeași poziție de start (există un ciclu) se face re-dispersare (**rehashing**)

#### 2. Liste independente interconectate (*Linked hashing*)

- JAVA – *LinkedHashMap*
  - *HashMap* din Java folosește rezolvare coliziuni prin liste independente (initial  $m=16$ )
  - Dacă  $\alpha > 0.75$ , se face redispersare (*rehashing*) –  $m$  se dublează
  - Java 8 – în locul listelor înlănuite se folosesc arbori binari de căutare echilibrați  $\Rightarrow \Theta(\log_2 n)$  caz defavorabil pentru căutare
- Combină ideea de TD și listă înlănuită.
  - Păstrează o listă dublu înlănuită cu toate elementele din TD într-o anumită ordine (implicit –în ordinea în care au fost adăugate în dicționar)
  - Fiecare intrare în tabelă (*Entry*) – nod care memorează perechea  $\langle c, v \rangle$  - are 2 pointeri adiționali spre perechea *anterioară* și cea *următoare* (adresele sunt ale nodurilor din lista dublu înlănuită)

- Datorită mecanismului de memorare, cheile vor fi returnate (la iterare) în ordinea în care au fost adăugate (varianta implicită – *insertion order* – se poate modifica la *access order*: de la cea mai recent accesată la cea mai devreme accesată).
- Aceeași performanță ca și *HashMap*
- Ca și implementarea *HashMap*, *LinkedHashMap* nu e sincronizată (nu funcționează cu acces concurrent)
- **Dezavantaj** – spațiu de memorare suplimentar pentru lista înlănțuită
- **Avantaj** - iterare în  $\Theta(n)$  (față de  $\Theta(n + m)$ )

## TABELA DE DISPERSIE

- continuare -

### B. Rezolvare coliziuni prin liste întrepătrunse (întrepătrunderea listelor) – COALESCED CHAINING

- Toate listele (care memorează coliziuni) se memorează în tabelă, nu sunt liste în afara tabelei (vezi lista înlățuită cu înlățuirile reprezentate pe tablou)
- Se mai numește și **dispersie închisă** (*closed hashing*)
  - considerată o tehnică de **adresare deschisă** (*open addressing*)
  - nu sunt elemente memorate în afara tabelei.
  - se consideră mai bună decât rezolvarea coliziunilor prin liste independente (elementele se memorează în tabelă, nu se memorează liste separate)
- Nu se folosesc pointeri pentru memorarea înlățuirilor
- Factorul de încărcare este subunitar  $\alpha < 1$ , altfel tabela este plină
- Gestiona spațiului liber în tabelă poate fi făcută ca la lista înlățuită cu înlățuirile reprezentate pe tablou (folosind o listă înlățuită a spațiului liber)
- Dezavantaj: tabela se poate umple ( $\alpha = 1$ ). Soluție: se crește  $m$ , ceea ce presupune redispersarea elementelor.
- Experimental: funcția de dispersie se consideră bună dacă spațiul de memorie e ocupat mai puțin de 75% ( $\alpha < 0.75$ )
- 

**Teoremă.** Într-o TD în care coliziunile sunt rezolvate prin liste întrepătrunse, în ipoteza dispersiei uniforme simple (SUH), o TOATE operațiile (**adăugare, căutare, ștergere**), necesită, în medie, un timp  $\theta(1)$ .

*Donald E. Knuth, The Art of Computer Programming, Second edition, University of Stanford, 1998*

- Timpul mediu pentru **căutare fără succes**  $T(\alpha) \approx 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$
- Timpul mediu pentru **căutare cu succes**  $T(\alpha) \approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$

Demonstrațiile sunt schițate și în directorul asociat cursului, subdirectorul **Demonstratii Complexitatii - TD Liste Intrepatrunse**.

#### Observație

- în cazul în care se folosește vector dinamic pentru implementarea tabelei, analiza anterioară a complexităților este valabilă pentru cazul **amortizat**.

#### Reprezentare

Ca și la lista simplu înlățuită în care înlățuirile sunt memorate pe tablou, table va memora doi vectori

- un vector care memorează elementele
- un vector care memorează legăturile între elemente (sub forma unor indici).

#### Presupuneri:

- Se memorează doar cheile.

- Chei distințe.
- Dacă o locație nu are legătură spre o altă locație din tabelă, se memorează **-1** în câmpul de legătură (*urm*).
- Chei naturale (se memorează **-1** dacă locația e liberă)
- Spațiul liber e gestionat secvențial (de la stânga la dreapta) – sau de la dreapta la stânga: *primLiber* indică prima poziție liberă din tabelă

## Container

*m*: Intreg //capacitatea tabelei

*ch*: TCheie[] //cheile

*urm*: Intreg[] //legaturile

*primLiber*: Intreg //prima locație libera

Funcția de dispersie este  $d: \text{TCheie} \rightarrow \{0, 1, \dots, m-1\}$

## EXEMPLU

$m=10$ ,  $d(c)=c \bmod m$

c	5	15	13	22	20	35	30	32	2
d(c)	5	5	3	2	0	5	0	2	2

## ADĂUGARE

Dacă adăugăm o cheie  $c$ , determinăm locația la care ar trebui memorată în tabelă ( $i=d(c)$ ), după care vom avea două situații

- Locația  $i$  este liberă (-1, în convenția noastră)  $\Rightarrow$  caz favorabil, memorăm cheia
  - dacă  $i$  e chiar *primLiber*, atunci se actualizează primul liber
- Locația  $i$  nu este liberă  $\Rightarrow$  avem coliziune
  - dacă *primLiber* = -1 (tabela este plină)  $\Rightarrow$  redimensionare: mărим  $m$ , ceea ce presupune redispersarea elementelor (*rehashing*)
  - memorăm cheia  $c$  la *primLiber*
  - ultimul nod din lista înlănțuită care începe de la locația  $i$  este legat de *primLiber*
  - se actualizează primul liber

## Pas 1. Inițializare

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
Următor	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- *primLiber* = 0

## Pas 2. Adăugăm cheia 5

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	-1	-1	-1	-1	-1	5	-1	-1	-1	-1
Următor	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- *primLiber* = 0

### Pas 3. Adăugăm cheia 15

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	15	-1	-1	-1	-1	5	-1	-1	-1	-1
Următor	-1	-1	-1	-1	-1	0	-1	-1	-1	-1

- $primLiber = 1$

### Pas 3. Adăugăm cheia 13

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	15	-1	-1	13	-1	5	-1	-1	-1	-1
Următor	-1	-1	-1	-1	-1	0	-1	-1	-1	-1

- $primLiber = 1$

### Pas 4. Adăugăm cheia 22

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	15	-1	22	13	-1	5	-1	-1	-1	-1
Următor	-1	-1	-1	-1	-1	0	-1	-1	-1	-1

- $primLiber = 1$

### Pas 5. Adăugăm cheia 20

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	15	20	22	13	-1	5	-1	-1	-1	-1
Următor	1	-1	-1	-1	-1	0	-1	-1	-1	-1

- $primLiber = 4$

.....

La final, tabela va fi

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	15	20	22	13	35	5	30	32	2	-1
Următor	1	4	7	-1	6	0	-1	8	-1	-1

**subalgoritm *actPrimLiber(c)* este**

//se actualizează *primLiber* după ce locația a fost ocupată

// operația nu este în interfața containerului

*c.primLiber*←*c.primLiber*+1

**câttimp** (*c.primLiber*≤ *c.m-1*) și (*c.ch[c.primLiber]*≠-1) **execută**

*c.primLiber*←-*c.primLiber*+1

**sfcâttimp**

**sfactPrimLiber**

- algoritmul de adăugare la sfârșitul listei înlănuite (în caz de coliziune) – **LISCH (Late Insertion Standard Coalesced Hashing)**

**subalgoritm *adaugă(c, ch)* este**

//pre: *c* e containerul, *ch* cheia care se adaugă

*i*←*c.d(ch)*

**dacă** *c.ch[i]*=-1 **atunci** //locația e liberă, memorăm

```

c.ch[i] ← ch
dăcă i=c.primLiber atunci
    actPrimLiber(c)
sfdacă

altfel
//adăugăm la finalul listei înlănțuite care este memorată de la locația i
// dacă mai găsim cheia, ne oprim
câttimp (i≠-1) și (c.ch[i] ≠ ch) execută
    j←i
    i←c.urm[i]

sfcâttimp
dăcă i≠-1 atunci //am mai găsit cheia
    @ cheie existentă
altfel
dăcă c.primLiber ≤ c.m-1 atunci //tabela nu este plină
    c.ch[c.primLiber] ← ch
    c.urm[j] ← c.primLiber
    actPrimLiber(c)

altfel
    @ depășire tabelă
sfdacă
sfdacă

sfadaugă

```

## CĂUTARE

- pp. că vrem să căutăm cheia  $ch$
- o căutăm în lista înlănțuită care pornește de la locația de dispersie a cheii  $ch$ , adică  $d(ch)$
- dacă găsim cheia în lista înlănțuită ⇒ **căutare cu succes**, altfel **căutare fără succes**
- exemplu
  - o căutăm **35 (cu succes)**: 5→15→20→**35**
  - o căutăm **45 (fără succes)**: 5→15→20→35→30→ -1

## STERGERE

- pp. că vrem să ștergem cheia  $ch$
- localizăm cheia
- exemplu
  - o  $ch=5$ , identificăm locația unde află cheia,  $i=5$
  - $(5,5) \rightarrow (0,15) \rightarrow (1,20) \rightarrow (4,35) \rightarrow (6,30)$

**Observație.** Într-o înlănțuire (de ex., înlănțuirea care pornește de la locația 5, nu avem doar chei care se află în aceeași coliziune). De ex. cheile 5, 15, 35 și 20, 30 se află în două coliziuni diferite.

Dacă am memora -1 pe poziția  $i$  (la care se află cheia  $ch$ ), nu s-ar mai regăsi cheile care au fost memorate pornind de la acea locație.

**Cum vom face ștergerea?** Prin deplasări ale cheilor, în lista înlănțuită care pornește de la poziția  $i$ .

- **Pas 1** pe înlănțuirea care pornește de la  $i$ , caut prima locație  $j$  care memorează o cheie  $c$  care ar fi trebuit memorată la  $i$  ( $d(c)=i$ )
- dacă găsesc locația  $j$ , atunci
  - mut cheia  $c$  la locația  $i$
  - continu cu ștergerea locației  $j$  ( $i \leftarrow j$ , salt la Pas 1)
- dacă nu găsesc locația  $j$ , înseamnă că pot șterge nodul de la locația  $i$ , deoarece ștergerea acestuia nu mai afectează regăsirea altor elemente (ștergerea acestui nod se reduce la ștergerea unui nod din lista simplu înlănțuită)
  - memorez -1 pe poziția cheii și legăturii de la  $i$
  - legătura precedentului lui  $i$  din înlănțuire o setez pe legătura lui  $i$

$$\Rightarrow (5,15) \rightarrow (0,20) \rightarrow (1,20) \rightarrow (4,35) \rightarrow (6,30)$$

- tabela rezultată în urma ștergerii

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	20	-1	22	13	35	15	30	32	2	
Următor	4	-1	7	-1	6	0	-1	8	0	-1

Iteratorul pe un container reprezentat folosind o TD cu liste întrepătrunse este simplu de implementat

- se iterează vectorul asociat, iterând doar pe pozițiile pe care e memorată o cheie diferită de -1 (în convenția noastră).
- $\Theta(m)$

**În directorul asociat cursului, găsiți implementarea parțială a containerului Colecție reprezentat folosind o TD cu liste întrepătrunse.**

**Variante pentru îmbunătățirea performanței (reducerea numărului de locații verificate în caz de coliziune)**

- organizarea tabelei - zonă separată (*primary area*) pentru elementele care nu sunt în coliziune și zonă separată pentru elementele care sunt în coliziuni (*overflow area*)
  - *Address Factor = |primary area| / dimensiunea tabelei*
  - o valoare de aprox. 0.86 pentru *Address Factor* conduce la o performanță aproape optimă pentru *căutare* pentru majoritatea valorilor lui  $a$ .
- modalitatea de înlănțuire a elementelor dintr-o coliziune
  - dublu înlănțuit
- modalitatea de selectare a spațiului liber (*primLiber*)
  - de la dreapta spre stânga (de la finalul tabelei)

- alegerea aleatoare a spațiului liber (doar 1% îmbunătățire) – alg. de adăugare **REISCH** (R - *random*)
- alternarea selecției spațiului liber între începutul și finalul tabelei - **BLISCH** (B – *bidirectional*)
- experimental: pentru  $\alpha > 0.2$  **LISCH** e mai performant

## **IMPLEMENTAREA OPERAȚIILOR DE CĂUTARE ȘI STERGERE LA SEMINARUL 6**

# ARBORELE (TREE)

- Arborii și variantele lor sunt printre cele mai comune și cele mai frecvent utilizate structuri de date, fiind utilizate într-o gamă foarte variată de aplicații cum ar fi teoria compilării, prelucrarea de imagini, etc., oferind o modalitate eficientă de memorare și manipulare a unei colecții de date.
- În teoria grafurilor, un arbore este un graf neorientat conex și fără cicluri.
- În informatică, *arborii cu rădăcină* sunt cei utilizati. De aceea, termenul *arbore* este asociat în informatică *arborelui cu rădăcină*.

**Definiție 1** Un arbore este o mulțime finită  $\mathcal{T}$  cu 0 sau mai multe elemente numite noduri, care are următoarele caracteristici:

- Dacă  $\mathcal{T}$  este vidă, atunci arborele este vid.
- Dacă  $\mathcal{T}$  este nevidă, atunci:
  - Există un nod special  $R$  numit rădăcină.
  - Celelalte noduri sunt partaționate în  $k$  ( $\geq 0$ ) arbori disjuncți,  $T_1, T_2, \dots, T_k$ , nodul  $R$  fiind legat de rădăcina  $R_i$  a fiecărui  $T_i$  ( $1 \leq i \leq k$ ) printr-o muchie. Arborii  $T_1, T_2, \dots, T_k$  se numesc subarbori ai lui  $R$  (nodurile  $R_i$  sunt fii/descendenți ai lui  $R$ ), iar  $R$  se numește părintele subarborilor  $T_i$  ( $1 \leq i \leq k$ ), respectiv a nodurilor  $R_i$ .

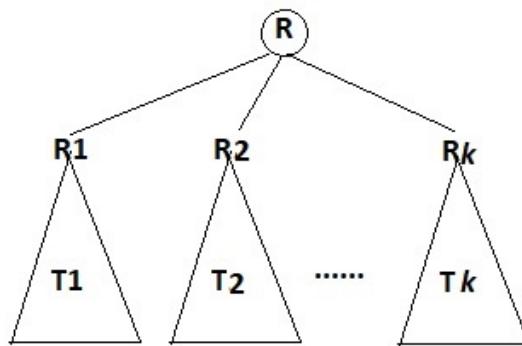


Figura 1: Arbore cu rădăcină.

**Arbore ordonat** - fiii fiecărui nod se consideră a forma o listă și nu doar o mulțime – adică ordinea fililor este bine definită și relevantă.

- *gradul* unui nod este definit ca fiind numărul de fii ai nodului. Nodurile având gradul 0 se numesc frunze.
- *Adâncimea (nivelul)* unui nod în arbore este definită ca fiind lungimea (numărul de muchii) drumului unic de la radacină la acel nod. Ca urmare, rădăcina arborelui este pe nivelul 0.

- Înălțimea unui nod în arbore este definită ca fiind lungimea (numărul de muchii) celui mai lung drum de la nod la un nod frunză.
- Înălțimea (adâncimea) unui arbore este definită ca fiind înălțimea rădăcinii arborelui, adică nivelul maxim al nodurilor din arbore.

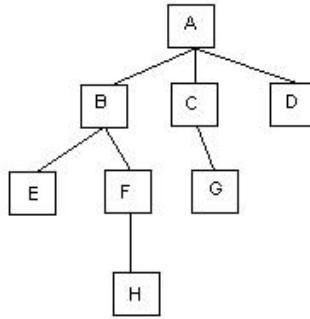


Figura 2: Arbore – exemplu.

Spre exemplu, arborele din Figura 2 are următoarele caracteristici:

- Rădăcina  $A$  este situată pe nivelul 0. Nodurile situate pe nivelul 1 sunt:  $B$ ,  $C$  și  $D$ . Nodurile situate pe nivelul 2 sunt:  $E$ ,  $F$  și  $G$ . Pe nivelul 3 există un singur nod, nodul  $H$ .
- Adâncimea (înălțimea) arborelui este 3.
- Nodul  $B$  are adâncimea 1 și înălțimea 2.

**Definiție 2 (Arbore binar)** Un arbore ordonat în care fiecare nod poate să aibă cel mult 2 subarbore se numește arbore binar. Mai exact, putem defini arborele binar ca având următoarele proprietăți:

- Un arbore binar poate fi vid.
- Într-un arbore binar nevid, fiecare nod poate avea cel mult 2 fi (subarbore). Subarboreii sunt identificați ca fiind subarborele stâng, respectiv drept. În Figura 3, nodul  $r$  are subarborele stâng  $A_1$  și subarborele drept  $A_2$ .

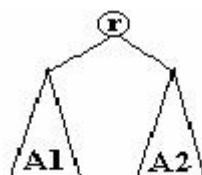


Figura 3: Arbore binar.

- Într-un arbore binar se face o distincție clară între subarborele drept și cel stâng.
- Dacă subarborele stâng este nevid, atunci rădăcina lui se numește fiul stâng al rădăcinii arborelui binar.
- Dacă subarborele drept este nevid, rădăcina lui se numește fiul drept al rădăcinii arborelui binar.
- Arborii binari din Figura 4 sunt distincți, deși conțin aceeași mulțime de noduri.

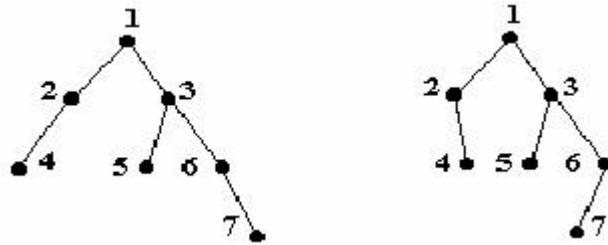


Figura 4: Arbori binari distincți.

Între arborii binari putem deosebi câteva categorii speciale:

- Un arbore binar pentru care fiecare nod interior are 2 fi (vezi Figura 5).
- Un arbore binar îl numim *plin* dacă fiecare nod interior are 2 fi și toate nodurile frunză au aceeași adâncime (vezi Figura 6).
- Un arbore binar are o structură de *ansamblu* (*heap*) dacă arborele este plin, exceptând ultimul nivel, care este plin de la stânga la dreapta doar până la un anumit loc (vezi Figura 7).

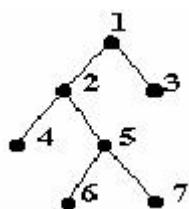


Figura 5: Arbore binar - nodurile interioare au 2 fi (*complet*).

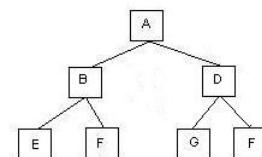


Figura 6: Arbore binar plin.

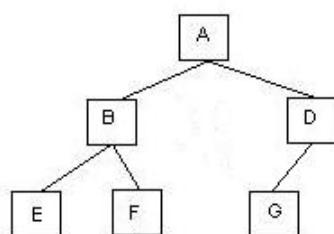


Figura 7: Arbore binar cu structură de ansamblu.



Figura 8: Arbori binari degenerați.

- Arborii binari se poate spune că au *formă*, forma lor fiind determinată de numărul nodurilor și de distanțele dintre noduri.
- Arborii binari din Figura 8 se numesc *degenerați*, deoarece au forma unui lanț de valori.
- Forma unui arbore influențează timpul necesar localizării unei valori în arbore.

**Arbore binar echilibrat** este un arbore binar cu proprietatea că înălțimea subarborelui său stâng nu diferă cu mai mult de  $\pm 1$  de înălțimea subarborelui său drept (și această proprietate este verificată pentru orice nod din arbore).

### Proprietăți ale AB:

1. Un arbore (nu neapărat binar) cu  $N$  vârfuri are  $N - 1$  muchii.
2. Numărul de noduri dintr-un arbore binar plin de înălțime  $N$  este  $2^{N+1} - 1$ .
3. Numărul maxim de noduri dintr-un arbore binar de înălțime  $N$  este  $2^{N+1} - 1$ .
4. Un arbore binar cu  $n$  vârfuri are înălțimea cel puțin  $\lceil \log_2 n \rceil$ .
5. Un arbore binar având o structură de ansamblu și  $n$  vârfuri are înălțimea  $\theta(\log_2 n)$ .

### Parcurgeri ale arborilor binari

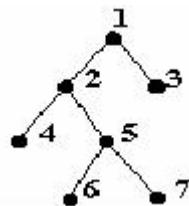


Figura 9: Arbore binar.

### Parcure în preordine

Pentru a parcurge în *preordine* un arbore binar, se vizitează rădăcina, apoi se parcurge în preordine subarborele stâng, apoi se parcurge în preordine subarborele drept (**RSD**).

- de exemplu, pentru arboarele binar din Figura 9 parcurgerea în preordine este: 1, 2, 4, 5, 6, 7, 3.

## Parcure în inordine (ordine simetrică)

Pentru a parcure în *inordine (ordine simetrică)* un arbore binar, se parcurează subarborele stâng, apoi se parcurează subarborele drept (**SRD**).

- de exemplu, pentru arborele binar din Figura 9 parcurea în inordine este: 4, 2, 6, 5, 7, 1, 3.

## Parcure în postordine

Pentru a parcure în *postordine* un arbore binar, se parcurează subarborele stâng, apoi se parcurează subarborele drept, după care se vizitează rădăcina (**SDR**).

- de exemplu, pentru arborele binar din Figura 9 parcurea în postordine este: 4, 6, 7, 5, 2, 3, 1.

## Parcure în lățime (pe niveluri)

Pentru a parcure în *lățime* un arbore binar, se vizitează nodurile pe niveluri, în ordine de la stânga la dreapta: nodurile de pe nivelul 0, apoi nodurile de pe nivelul 1, nodurile de pe nivelul 2, etc.

- de exemplu, pentru arborele binar din Figura 9 parcurea în lățime este: 1, 2, 3, 4, 5, 6, 7.
- traversarea BFS (*Breadth first search*)

## **TAD ArboreBinar (BINARY TREE)**

### Observații

- Sunt aplicații în care am avea nevoie de memorarea datelor sunt forma unui arbore binar.
  - de exemplu, memorarea informațiilor dintr-un arbore genealogic.
- În majoritatea bibliotecilor existente, containerul **Tree** apare pe partea de GUI (ex. Java - clasa **JTree**).
  - arborele binar (de căutare) este folosit ca structură de date pentru a implementa containere: ex. în Java - TreeSet, TreeMap, etc)
- Dăm în continuarea interfața minimală TAD ArboreBinar.

- Pe lângă operațiile de mai jos, pot fi adăugate operații care:

- să **șteargă** un subarbore;
- să **modifice** informația utilă a rădăcinii unui subarbore;
- să **caute** un element în arbore, etc.

## TAD ArboreBinar

**domeniu:**

$$\mathcal{AB} = \{ab \mid ab \text{ arbore binar cu noduri care conțin informații de tip } TElement\}$$

**operatii (interfața minimală):**

- creeaza( $ab$ )

*pre : adevarat*  
*post :  $ab \in \mathcal{AB}$ ,  $ab$  = arbore vid ( $a = \Phi$ )*

- creeazaFrunza( $ab, e$ )

*pre :  $e \in TElement$*   
*post :  $ab \in \mathcal{AB}$ ,  $ab$  arbore având un singur nod și informația din nodul rădăcină este egală cu  $e$*

- creeazaArb( $ab, st, e, dr$ )

*pre :  $st, dr \in \mathcal{AB}, e \in TElement$*   
*post :  $ab \in \mathcal{AB}$ ,  $ab$  arbore cu subarbore stâng =  $st$ , cu subarbore drept =  $dr$  și informația din nodul rădăcină este egală cu  $e$*

- adaugaSubStang( $ab, st$ )

*pre :  $ab, st \in \mathcal{AB}$*   
*post :  $ab' \in \mathcal{AB}$ ,  $ab'$  are subarborele stâng =  $st$*

- adaugaSubDrept( $ab, dr$ )

*pre :  $ab, dr \in \mathcal{AB}$*   
*post :  $ab' \in \mathcal{AB}$ ,  $ab'$  are subarborele drept =  $dr$*

- element( $ab$ )

*pre :  $ab \in \mathcal{AB}, ab \neq \Phi$*   
*post : element =  $e$ ,  $e \in TElement$ ,  $e$  este informația din nodul rădăcină*

- stang( $ab$ )

*pre :  $ab \in \mathcal{AB}, ab \neq \Phi$*   
*post : stang =  $st$ ,  $st \in \mathcal{AB}$ ,  $st$  este subarborele stâng al lui  $ab$*

- drept( $ab$ )

*pre :  $ab \in \mathcal{AB}, ab \neq \Phi$*   
*post : drept =  $dr$ ,  $dr \in \mathcal{AB}$ ,  $dr$  este subarborele stâng al lui  $ab$*

- $\text{vid}(ab)$

$$\begin{aligned} \text{pre : } ab &\in \mathcal{AB} \\ \text{post : } \text{vid} &\begin{cases} \text{true} & \text{dacă } ab = \Phi \\ \text{false}, & \text{altfel} \end{cases} \end{aligned}$$

- iterator  $(ab, ordine, i)$

$$\begin{aligned} \text{pre : } ab &\in \mathcal{AB}, \text{ordine este ordinea de traversare a arborelui} \\ \text{post : } i &\in \mathcal{I}, i \text{ este un iterator pe arborele } ab \text{ în ordinea} \\ &\text{precizată de } ordine \end{aligned}$$

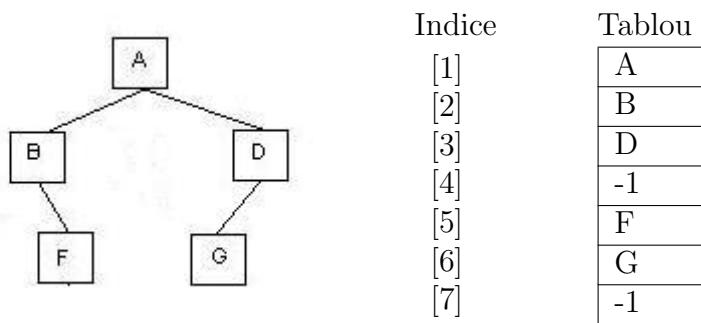
- distruge( $ab$ ) {destructor}

$$\begin{aligned} \text{pre : } ab &\in \mathcal{AB} \\ \text{post : } ab &\text{ a fost 'distrus' (spațiul de memorie alocat a fost eliberat)} \end{aligned}$$

## Reprezentări posibile pentru arbori binari

### A. Reprezentarea secvențială pe tablou

- Se folosește ca schemă de memorare un ansamblu ( $A[1..Max]$ ):
  - $A_1$  este elementul din nodul rădăcină.
  - fiul stâng al lui  $A_i$  este  $A_{2 \cdot i}$ .
  - fiul drept al lui  $A_i$  este  $A_{2 \cdot i + 1}$ .
  - părintele lui  $A_i$  este  $A_{[i/2]}$ .



### B. Reprezentarea înlănțuită

Într-o astfel de reprezentare, în fiecare nod reprezentat al arborelui sunt memorate:

- informația utilă a nodului din arbore;
- două referințe spre cei doi fii – stâng și drept.

Arborele va fi identificat prin referința spre nodul rădăcină.

## B.1 Reprezentarea înlățuirilor folosind alocarea dinamică a memoriei.

- Referințele sunt în acest caz pointeri (adrese de memorie).
- Pointerul NIL indică un arbore vid.

De exemplu, pentru un **Container** oarecare (de ex. Colecție) reprezentat sub forma unui AB:

### Nod

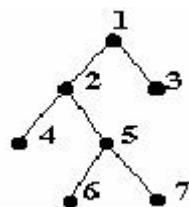
e: TElement //informația utilă nodului  
st:  $\uparrow$  Nod //adresa la care e memorat descendantul stâng  
dr:  $\uparrow$  Nod //adresa la care e memorat descendantul drept

### Container

rad:  $\uparrow$  Nod //adresa rădăcinii AB

## B.2. Reprezentarea înlățuirilor pe tablou.

Pentru arborele



reprezentarea (B.2) este

Indice	1	2	3	4	5	6	7	8	9	10
Element	3	-	5	1	-	2	6	-	4	7
Stanga	0	8	7	6	0	9	0	5	0	0
Dreapta	0		10	1		3	0		0	0

Tabela 1: radacina=4, primLiber=2

**Observație:**

1. Capacitatea vectorilor poate fi mărită dinamic, dacă este necesar - numărul de elemente din arbore depășește numărul de locații alocate inițial (vezi vectorul dinamic).

În directorul asociat Cursului 11 găsiți implementarea parțială, în limbajul C++, a containerului **ArboreBinar** (reprezentarea este înlățuită, cu alocare dinamică a nodurilor). Iteratorii nu sunt implementați.

## Parcurgeri recursive ale arborilor binari

- Variantele recursive de parcure sunt simplu de implementat (datorită definiției recursive a arborelui binar).
- Dezavantajul procedurilor recursive: supraîncărcarea stivei de execuție.

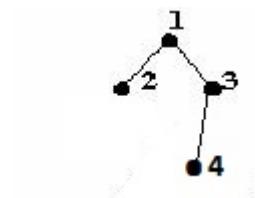
Considerând TAD ArboreBinar definit anterior, indiferent de reprezentarea acestuia, se pot tipări nodurile arborelui (ex. în PREORDINE), folosind procedura recursivă de mai jos:

```
Subalgoritm preordine(ab)
    {complexitate timp:  $\theta(n)$ }
pre:    ab este un arbore binar
post:   se afișează în preordine elementele din arbore
        {dacă arborele nu este vid}
        Daca  $\neg$  vid(ab) atunci
            {se prelucrează informația din rădăcina arborelui}
            element(ab, e)
            {prelucrează e}
            {se prelucrează recursiv subarborele stâng}
            preordine(stang(ab))
            {se prelucrează recursiv subarborele drept}
            preordine(drept(ab))
        SfDaca
SfSubalgoritm
```

## Parcurgeri nerecursive ale arborilor binari

Pentru a implementa iteratorii pe arbore, avem nevoie de o parcure iterativă.

Presupunem (în cele ce urmează) reprezentare înlănțuită cu alocare dinamică.  
Pentru a exemplifica parcurgerile, considerăm AB de mai jos



### PREORDINE

Se va folosi o **STIVĂ** auxiliară.

```
Subalgoritm preordine(ab)
    {complexitate timp:  $\theta(n)$ }
```

pre:  $ab$  este un arbore binar  
 post: se afișează în preordine elementele din arbore  
 {stiva va conține adrese de noduri}  
**creeaza( $S$ )**  
 Daca  $ab.rad \neq NIL$  atunci  
     {arborele nu e vid}  
     adauga( $S, ab.rad$ )  
     {se adauga radacina in stiva}  
**SfDaca**  
**CatTimp**  $\neg \text{vida}(S)$  executa  
     sterge( $S, p$ )  
     {se sterge nodul din varful stiva}  
     @tipareste[ $p$ ].e  
 Daca  $[p].dr \neq NIL$  atunci  
     {exista legatura dreapta}  
     adauga( $S, [p].dr$ )  
     {se adauga descendantul drept in stiva}  
**SfDaca**  
 Daca  $[p].st \neq NIL$  atunci  
     {exista legatura stanga}  
     adauga( $S, [p].st$ )  
     {se adauga descendantul stang in stiva}  
**SfDaca**  
**SfCatTimp**  
**SfSubalgoritm**

### Exemplificare

Stiva $S$	$p$	Ce se tipărește
①	-	-
$\emptyset$	①	1
---	---	---
②		
③	-	-
③	②	2
$\emptyset$	③	3
---	---	---
④	-	-
$\emptyset$	④	4
STOP		

### LĂTIME (parcuregere pe niveluri)

Se va folosi o **COADĂ** auxiliară.

**Subalgoritm niveluri( $ab$ )**  
     {complexitate timp:  $\theta(n)$ }  
 pre:  $ab$  este un arbore binar  
 post: se afișează în lățime elementele din arbore  
     {coada va conține adrese de noduri}

```

creeaza(C)
Daca ab.rad ≠ NIL  atunci
    {arborele nu e vid}
    adauga(C, ab.rad)
    {se adauga radacina in coada}
SfDaca
CatTimp ¬vida(C) executa
    sterge(C, p)
    {se sterge nodul din coada}
    @tipareste[p].e
    Daca [p].st ≠ NIL  atunci
        {exista legatura stanga}
        adauga(C, [p].st)
        {se adauga descendantul stang in coada}
    SfDaca
    Daca [p].dr ≠ NIL  atunci
        {exista legatura dreapta}
        adauga(C, [p].dr)
        {se adauga descendantul drept in coada}
    SfDaca
SfCatTimp
SfSubalgoritm

```

## INORDINE

Se va folosi o **STIVĂ** auxiliara.

```

Subalgoritm inordine(ab)
    {complexitate timp:  $\theta(n)$ }
pre:   ab este un arbore binar
post:  se afiseaza in inordine elementele din arbore
       {stiva va contine adrese de noduri}
       creeaza(S)
       p ← ab.rad
       {nodul curent}
CatTimp ¬ vida(S) ∨ (p ≠ NIL) executa
    CatTimp p ≠ NIL executa
        {se adauga in stiva ramura stanga a lui p}
        adauga(S, p)
        p ← [p].st
    SfCatTimp
    sterge(S, p)
    {se sterge nodul din varful stivei}
    @tipareste[p].e
    p ← [p].dr
SfCatTimp
SfSubalgoritm

```

## Iterator INORDINE

Nod

AB

IteratorInordine

<i>e</i> :TElement	<i>rad</i> : $\uparrow$ Nod	<i>a</i> :AB
<i>st, dr</i> : $\uparrow$ Nod		<i>current</i> : $\uparrow$ Nod
		<i>s</i> :Stivă

- se va folosi o **Stivă** care va conține  $\uparrow$ Nod și care are în interfață operațiile specifice: *creează(s)*, *vidă(s)*, *adaugă(s,e)*, *element(s,e)*, *sterge(s,e)*.

## Varianta 1

```

Subalgoritm creează(i, ab)
{constructorul iteratorului}
i.ab  $\leftarrow ab$ 
i.current  $\leftarrow ab.rad$ 
{constructorul stivei}
creează(i.s)
SfSubalgoritm

Functia valid(i)
{validitatea iteratorului}
valid  $\leftarrow (i.current \neq NIL) \vee \neg vida(i.s)$ 
SfFunctia

Subalgoritm element(i, e)
{elementul curent al iteratorului}
CatTimp i.current  $\neq NIL$  executa
{se adauga in stiva ramura stanga a elementului curent}
adauga(i.s, i.current)
i.current  $\leftarrow [i.current].st$ 
SfCatTimp
sterge(i.s, i.current)
{se sterge nodul din varful stivei}
e  $\leftarrow [i.current].e$ 
SfSubalgoritm

Subalgoritm urmator(i)
{deplasează iteratorul}
i.current  $\leftarrow [i.current].dr$ 
SfSubalgoritm

```

## Varianta 2

```

Subalgoritm creează(i, ab)
{constructorul iteratorului}
i.ab  $\leftarrow ab$ 
{constructorul stivei}
creează(i.s)
i.current  $\leftarrow ab.rad$ 
CatTimp i.current  $\neq NIL$  executa
{se adauga in stiva ramura stanga a elementului curent}
adauga(i.s, i.current)
i.current  $\leftarrow [i.current].st$ 

```

```

SfCatTimp
Daca  $\neg$  vida(i.s) atunci
    {se acceseaza nodul din varful stivei}
    element(i.s, i.current)
SfDaca
SfSubalgoritm

Functia valid(i)
    {validitatea iteratorului}
    valid  $\leftarrow$  (i.current  $\neq$  NIL)
SfFunctia

Subalgoritm element(i, e)
    {elementul curent al iteratorului}
    e  $\leftarrow$  [i.current].e
SfSubalgoritm

Subalgoritm urmator(i)
    {se sterge nodul din varful stivei}
    sterge(i.s, i.current)
    Daca [i.current].dr  $\neq$  NIL atunci
        {deplasează iteratorul}
        i.current  $\leftarrow$  [i.current].dr
        CatTimp i.current  $\neq$  NIL execută
            {se adauga în stiva ramura stanga a elementului curent}
            adauga(i.s, i.current)
            i.current  $\leftarrow$  [i.current].st
    SfCatTimp
    SfDaca
    Daca  $\neg$  vida(i.s) atunci
        {se acceseaza nodul din varful stivei}
        element(i.s, i.current)
    altfel
        i.current  $\leftarrow$  NIL
    SfDaca
SfSubalgoritm

```

## POSTORDINE

Se va folosi o **STIVĂ** auxiliară. Un element din stivă va fi de forma  $[p, k]$  unde:

- $p$  este adresa nodului;
- $k$  este 0 (dacă nu s-a trecut la partea dreaptă a lui  $p$ ) sau 1 (s-a trecut la parcerea subarborelui drept al lui  $p$ ).

```

Subalgoritm postordine(ab)
    {complexitate timp:  $\theta(n)$ }
    pre:   ab este un arbore binar
    post:  se afișează în postordine elementele din arbore

```

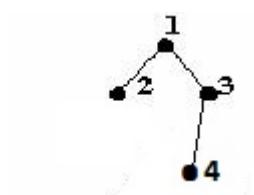
```

creeaza(S)
p  $\leftarrow$  ab.rad
{nodul curent}
CatTimp  $\neg$  vida(S)  $\vee$  (p  $\neq$  NIL) executa
  CatTimp p  $\neq$  NIL executa
    {se adauga in stiva ramura stanga a lui p}
    adauga(S, [p, 0])
    p  $\leftarrow$  [p].st
  SfCatTimp
  sterge(S, [p, k])
  {se sterge nodul din varful stivei}
  Daca k = 0 atunci
    {nu s-a traversat subarborele drept al lui p}
    adauga(S, [p, 1])
    p  $\leftarrow$  [p].dr
  altfel
    @tipareste[p].e
    p  $\leftarrow$  NIL
    {trebuie extras un nou nod din stiva - al doilea ciclu CatTimp nu trebuie sa se mai execute}
  SfDaca
SfCatTimp
SfSubalgoritm

```

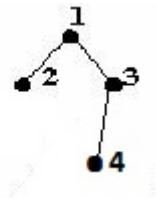
## Probleme

- Implementați iteratori cu parcurgere în preordine, postordine și lățime a nodurilor unui arbore binar.
- Descrieți în Pseudocod operația pentru căutarea într-un arbore binar a unei informații date. Se va folosi o implementare iterativă.
- Descrieți în Pseudocod operația pentru determinarea înălțimii unui arbore binar. Se va folosi o operație iterativă.
- Să se scrie o procedură iterativă pentru determinarea nivelului pe care apare într-un arbore binar o informație dată.
- Scrieți o operație nerecursivă care determină părintele unui nod *p* dintr-un AB.
- Cunoscând preordinea și inordinea nodurilor unui arbore binar, să se descrie o operatie care construiește arborele. (vezi SEMINAR 7)
  - de exemplu. dacă preordinea (RSD) este 1 2 3 4 și inordinea (SRD) este 2 1 4 3, atunci arborele este



7. Cunoscând postordinea și inordinea nodurilor unui arbore binar, să se descrie o operatie care construiește arborele. (vezi SEMINAR 7)

- de exemplu, dacă postordinea (SDR) este 2 4 3 **1** și inordinea (SRD) este 2 **1** 4 3, atunci arborele este



## Expresii aritmetice

O expresie aritmetică se poate reprezenta printr-un arbore binar ale cărui noduri terminale sunt asociate cu variabile sau constante și ale cărui noduri interne sunt asociate cu unul dintre operatorii: +, -, ×, și /. (Vezi Figura 11.)

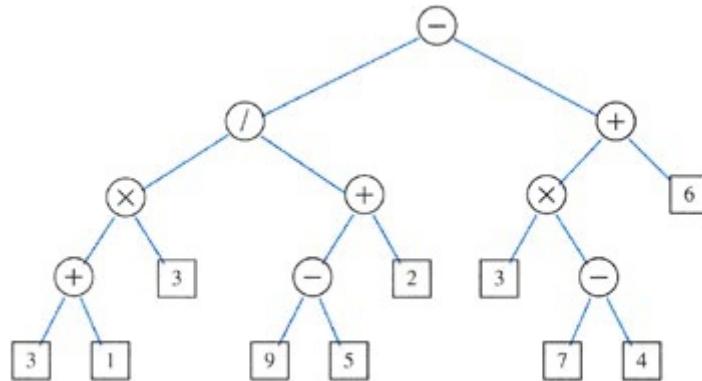


Figura 10: Arbore corespunzător expresiei  $((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6)$ .

Fiecare nod dintr-un asemenea arbore are o valoare asociată:

- Dacă nodul este terminal valoarea sa este cea a variabilei sau constantei asociate.
- Dacă nodul este neterminal valoarea sa este definită prin aplicarea operației asociate asupra filor lui.

1. **Evaluarea unei expresii aritmetice din forma postfixată.** Fie  $EPost$  o expresie aritmetică CORECTĂ în forma postfixată, conținând operatorii binari: +, -, \*, /, iar ca operanzi cifre 0-9 (ex:  $EPost = 1\ 2\ +\ 3\ *\ 4\ /$ ). Se cere să se determine valoarea expresiei (ex: valoarea este 2.25). **Indicație:** Se va folosi o stivă în care se vor adăuga operanzii. În final, stiva va conține valoarea expresiei.

Vom putea folosi următorul algoritm.

- Pentru fiecare  $e \in EPost$  (în ordine de la stânga la dreapta)
  - Dacă  $e$  este operand, atunci se adaugă în stivă.
  - Dacă  $e$  este operator, atunci se scoate din stivă doi operanzi ( $op_1$  și  $op_2$ ), se efectuează operația  $e$  între cei doi operanzi ( $v = op_2 e op_1$ ), după care se adaugă  $v$  în stivă. **Obs.** Aici s-ar putea depista dacă expresia nu ar fi corectă în forma postfixată (s-ar încerca extragerea unui element dintr-o stivă vidă).
- În presupunerea noastră că expresia în forma postfixată este validă, stiva va conține la final o singură valoare, care se va extrage din stivă. Această valoare reprezintă valoarea expresiei.

**Exemplu.** Fie expresia  $EPost$   $1\ 2\ 3\ *\ +\ 4\ 5\ *\ -$ . Arborele asociat ar fi

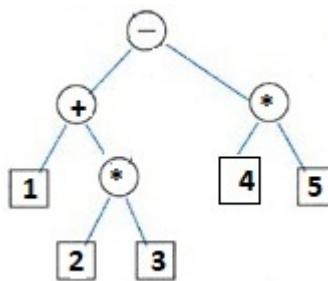


Figura 11: Arbore corespunzător expresiei în forma postfixată  $1\ 2\ 3\ *\ +\ 4\ 5\ *\ -$ .

Aplicarea algoritmului indicat mai sus este ilustrată în Tabelul de mai jos.

(3)	(2)	(5)		
1	6	4	20	
1	1	7	7	-13

Tabela 2: Stiva pe parcursul aplicării algoritmului

2. **Translatarea unei expresii aritmetice din forma infixată în forma postfixată.** Fie  $E$  o expresie aritmetică CORECTĂ în forma infixată, fără paranteze, conținând operatorii binari:  $+$ ,  $-$ ,  $*$ ,  $/$ , iar ca operanzi cifre 0-9 (ex:  $E = 1 + 2 * 3$ ). Se cere să se determine forma postfixată  $EPost$  a expresiei (ex:  $EPost = 1\ 2\ 3\ *\ +$ ). **Indicație:** Se va folosi o stivă în care se vor adăuga operatorii și o coadă  $EPost$  care va conține în final forma postfixată a expresiei.

Vom putea folosi următorul algoritm.

- Pentru fiecare  $e \in E$  (în ordine de la stânga la dreapta)
  - Dacă  $e$  este operand, atunci se adaugă în coada  $EPost$ .

(b) Dacă  $e$  este operator, atunci se scot din stivă operatorii având prioritate de evaluare mai mare sau egală decât a lui  $e$  și se adaugă în coada  $EPost$ , după care se adaugă  $e$  în stivă.

- Se scot din stivă operatorii rămași și se adaugă în coada  $EPost$ .
- În final,  $EPost$  va conține forma postfixată a expresiei.

**Exemplu.** Considerăm expresia  $E$  în forma infixată corespunzătoare exemplului din Figura 11:  $E = 1 + 2 * 3 - 4 * 5$ .

- $EPost$  ar trebui să fie  $1\ 2\ 3\ *\ +\ 4\ 5\ *\ -$

Aplicarea algoritmului indicat mai sus este ilustrată în Tabelul de mai jos.

$EPost$		
1	2	3
1	2	3
1	2	3
1	2	3   4
1	2	3   4

3. Translați o expresie aritmetică din forma infixată în forma postfixată. Expressia conține paranteze, care indică ordinea de evaluare. De exemplu, expresia  $(2+4)*6-(3+2)$  are forma postfixată  $2\ 4\ +\ 6\ *\ 3\ 2\ +\ -$ . **Indicație:** Ideea/algoritmul de bază este același ca și în cazul în care expresia nu ar conține paranteze. Paranteza deschisă "(" se va adăuga în stivă. Va trebui să identificați pașii care vor trebui efectuați la întâlnirea unei paranteze închise ")".
4. Translați o expresie aritmetică din forma infixată în forma prefixată/postfixată. Folosiți un AB intermediar.
5. Evaluați o expresie aritmetică în forma infixată folosind un AB intermedier.

# ARBORI BINARI DE CĂUTARE (BINARY SEARCH TREE)

- *Arborii binari de căutare* (ABC) sunt structuri de date des folosite în implementarea containerelor care conțin elemente de tip *TComparabil* (sau identificate prin chei de tip *TComparabil*) și care suportă următoarele operații:
  1. căutare;
  2. adăugare;
  3. ștergere;
  4. determinare maxim, minim, predecesor, succesor.
- ABC sunt SD care se folosesc pentru implementare:
  - dicționar, dictionar ordonat
    - \* **TreeMap** în Java (folosește ABC echilibrat - arbore roșu-negru)
    - \* **map** din STL folosește ABC echilibrat ca implementare.
      - C++ 11 - **unordered\_map** (tabelă de dispersie)
  - coadă cu priorități;
  - listă (**TreeList** în Java; folosește ABC echilibrat);
  - colecție, mulțime (**TreeSet** în Java; folosește ABC echilibrat - arbore roșu-negru);

**Observație:** Presupunem în cele ce urmează (fără a reduce generalitatea):

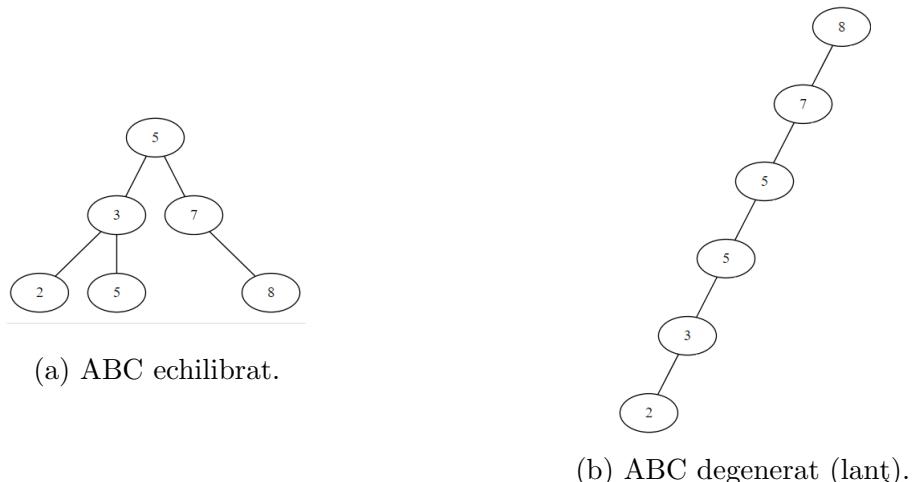
1. *Elementele* sunt identificate printr-o **cheie**.
2. *Cheia* elementului este de tip *TComparabil*.
3. Pp. relația “ $\leq$ ” între chei (se poate ușor generaliza la o relație de ordine oarecare).

**Definiție 1** Un ABC este un AB care satisface următoarea **proprietate** (proprietarya ABC):

- dacă  $x$  este un nod al ABC, atunci:
  - $\forall y$  un nod din subarborele stâng al lui  $x$ , are loc inegalitatea  $\text{cheie}(y) \leq \text{cheie}(x)$  ( $\text{cheie}(y) \mathcal{R} \text{cheie}(x)$ ).
  - $\forall y$  un nod din subarborele drept al lui  $x$ , are loc inegalitatea  $\text{cheie}(x) < \text{cheie}(y)$  ( $\neg(\text{cheie}(y) \mathcal{R} \text{cheie}(x))$ ).

**Exemplu.** Presupunem că în container avem cheile 2 3 5 5 7 8 și relația  $\mathcal{R} = \leq$ . În Figura 1 sunt indicați 2 ABC care conțin aceste chei.

- operațiile de bază pe ABC consumă timp  $O(\text{înălțimea arborelui})$ .



- dacă ABC este plin  $\Rightarrow$  înălțimea este  $\theta(\log_2 n)$  - caz (a);
- dacă ABC este degenerat (lanț liniar)  $\Rightarrow$  înălțimea este  $\theta(n)$  - caz (b).
- forma unui ABC este importantă
  - \* operațiile vor avea complexitate timp  $O(h)$ ,  $h$  fiind înălțimea arborelui

**Proprietate.** Traversarea în inordine a unui ABC furnizează cheile în ordine în raport cu relația de ordine (ordine crescătoare dacă  $R = \leq$ ).

- în exemplele (a) și (b) de mai sus, parcurgerea în inordine este 2 3 5 5 7 8

## Reprezentarea ABC

Un ABC (ca și un AB) poate fi reprezentat:

1. secvențial;
  2. înlántuit
    - reprezentarea înlántuirilor folosind alocare dinamică (pointeri);
    - reprezentarea înlántuirilor folosind alocare statică (tablouri).
- pentru implementarea operațiilor, pp. în cele ce urmează
    - reprezentare înlántuită folosind alocare dinamică.
    - $\mathcal{R} = \leq$
  - notăm cu  $h$  înălțimea arborelui.
  - notăm cu  $n$  numărul de noduri din arbore.

## REPREZENTARE

- într-un nod memorăm informația utilă și adresa celor doi descendenți (stâng și drept)

- putem memora (pentru a eficientiza anumite operații) și alte informații
  - adresa părintelui
  - adâncimea, înalțimea nodului

### Nod

e: TElement //informația utilă nodului  
 st:  $\uparrow$  Nod //adresa la care e memorat descendantul stâng  
 dr:  $\uparrow$  Nod //adresa la care e memorat descendantul drept

### Container

rad:  $\uparrow$  Nod //adresa rădăcinii ABC

## CĂUTARE - pp. chei distințe.

- returnează subarborele a cărui rădăcină conține cheia căutată.

### Varianta recursivă.

Modelul matematic recursiv

- arborele binar de căutare îl vom referi sub forma  $abc(r,s,d)$ 
  - $r$  - e informația din rădăcină
  - $s$  - e subarborele stâng
  - $d$  - e subarborele drept

$$cauta\_rec(abc(r,s,d), e) = \begin{cases} \emptyset & abc(r, s, d) = \emptyset \\ abc(r, s, d) & r.c = e.c \\ cauta\_rec(s, e) & e.c \leq r.c \\ cauta\_rec(d, e) & altfel \end{cases}$$

### Functia cauta( $abc, e$ )

{complexitate timp:  $O(h)$ }

pre:  $abc$  este un **container** reprezentat folosind un arbore binar de căutare

post: se returnează pointer spre nodul care conține un element având cheia egală cu cheia lui  $e$

$cauta \leftarrow cauta\_rec(abc.rad, e)$

### SfFunctia

### Functia cauta\_rec( $p, e$ )

{complexitate timp:  $O(h)$ }

pre:  $p$  este adresa unui nod;  $p : \uparrow Nod$  - rădăcina unui subarbore

post: se returnează pointer spre nodul care conține un element având cheia egală cu cheia lui  $e$  în subarborele de rădăcină  $p$

{dacă s-a ajuns la subarbore vid sau nodul este cel căutat}

Daca  $p = NIL \vee [p].e.c = e.c$  atunci

```

cauta_rec ← p
altfel
Daca e.c < [p].e.c atunci
{se caută în subarborele stâng}
cauta_rec ← cauta_rec([p].st, e)
altfel
{se caută în subarborele drept}
cauta_rec ← cauta_rec([p].dr, e)
SfDaca
SfDaca
SfFunctia

```

### Varianta iterativă.

Functia cauta( $abc, e$ )  
{complexitate timp:  $O(h)$ }

post: se returnează pointer spre nodul care conține un element având cheia egală cu cheia lui  $e$  în arborele  $abc$

```

p← abc.rad
CatTimp p ≠ NIL ∧ [p].e.c ≠ e.c executa
    Daca e.c < [p].e.c atunci
        {se caută în subarborele stâng}
        p← [p].st
    altfel
        {se caută în subarborele drept}
        p← [p].dr
    SfDaca
    SfCatTimp
    cauta← p
SfFunctia

```

## ADĂUGARE

- returnează arborele rezultat prin inserarea unui element într-un arbore

Modelul matematic recursiv

$$adauga\_rec(abc(r, s, d), e) = \begin{cases} abc(e, \emptyset, \emptyset) & abc(r, s, d) = \emptyset \\ abc(r, adauga\_rec(s, e), d) & e.c \leq r.c \\ abc(r, s, adauga\_rec(d, e)) & altfel \end{cases}$$

Functia creeazaNod( $e$ )  
{complexitate timp:  $\theta(1)$ }

pre:  $e$  este de tip  $TElement$

post: se returnează pointer spre un nod care conține elementul  $e$   
{se alocă un spațiu de memorare pentru un nod;  $p : \uparrow Nod$ }  
aloca( $p$ )  
{se completează componentele nodului}  
 $[p].e \leftarrow e$

```


$[p].st \leftarrow NIL$



$[p].dr \leftarrow NIL$



$\text{creeazaNod} \leftarrow p$



SfFunctia



Subalgoritm adauga( $abc, e$ )  

 $\{\text{complexitate timp: } O(h)\}$



pre:  $abc$  este un container reprezentat folosind un arbore binar de căutare



post:  $abc'$  este containerul în care a fost adăugat  $e$



$abc.rad \leftarrow \text{adauga\_rec}(abc.rad, e)$



SfSubalgoritm



Functia adauga_rec( $p, e$ )  

 $\{\text{complexitate timp: } O(h)\}$



pre:  $p$  este adresa unui nod;  $p : \uparrow \text{Nod}$  - rădăcina unui subarbore



post: se adaugă  $e$  în subarborele de rădăcină  $p$  și se returnează rădăcină nouui subarbore  

 $\{\text{dacă s-a ajuns la subarbore vid se adaugă}\}$



Daca  $p = NIL$  atunci



$p \leftarrow \text{creeazaNod}(e)$



altfel



Daca  $e.c \leq [p].e.c$  atunci  

 $\{\text{se adaugă în subarborele stâng}\}$



$[p].st \leftarrow \text{adauga\_rec}([p].st, e)$



altfel  

 $\{\text{se adaugă în subarborele drept}\}$



$[p].dr \leftarrow \text{adauga\_rec}([p].dr, e)$



SfDaca



SfDaca  

 $\{\text{se returnează rădăcina subarborelui}\}$



$\text{adauga\_rec} \leftarrow p$



SfFunctia


```

## MINIM

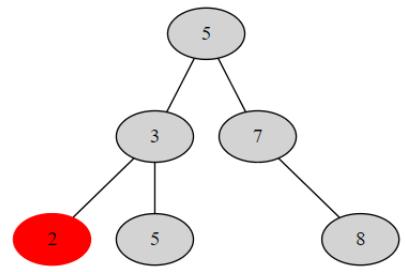


Figura 2: Minim.

**Functia minim( $p$ )**  
 $\{\text{complexitate timp: } O(h)\}$

*pre:*  $p$  este adresa unui nod;  $p : \uparrow \text{Nod}; p \neq NIL$

*post:* se returnează adresa nodului având cheia minimă din subarborele de rădăcină  $p$

CatTimp  $[p].st \neq NIL$  execută

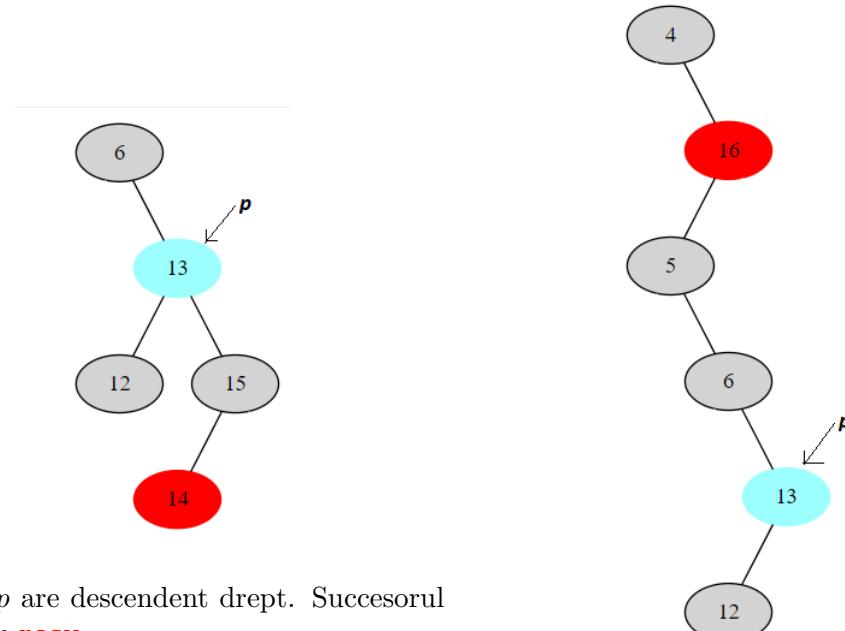
```

 $p \leftarrow [p].st$ 
SfCatTimp
minim  $\leftarrow p$ 
SfFunctia

```

## SUCCESOR

- cheia din container imediat mai mare (dacă  $\mathcal{R} = \leq$ ) decât o cheia dintr-un nod  $p$  dat
- de exemplu, dacă am avea cheile 2 6 3 4 2 3 1, atunci succesorul cheii 4 este 6.



(a) Nodul  $p$  are descendent drept. Succesorul e marcat cu **roșu**.

(b) Nodul  $p$  nu are descendent drept. Succesorul e marcat cu **roșu**.

```

Functia succesor( $p$ )
pre:    $p$  este adresa unui nod;  $p : \uparrow Nod$ ;  $p \neq NIL$ 
post:  se returnează adresa nodului având cheia imediat mai mare decât cheia din  $p$ 
Daca  $[p].dr \neq NIL$  atunci
    {există subarbore drept al lui  $p$ }
    succesor  $\leftarrow \text{minim}([p].dr)$ 
altfel
    prec  $\leftarrow \text{parinte}(p)$ 
    CatTimp prec  $\neq NIL \wedge p = [prec].dr$  executa
         $p \leftarrow prec$ 
        prec  $\leftarrow \text{parinte}(p)$ 
    SfCatTimp
    succesor  $\leftarrow prec$ 
SfDaca
SfFunctia

```

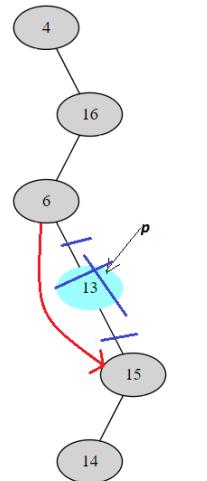
## Observație

- dacă un nod memorează adresa părintelui, atunci complexitatea operației este  $O(h)$ , în caz contrar este  $O(h^2)$ .

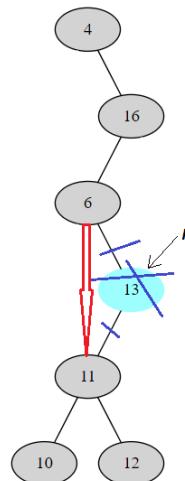
## ȘTERGERE

- pp. chei distincte
- se șterge nodul având cheia egală cu cea a unui element dat
- se returnează arborele rezultat în urma ștergerii

Sunt trei cazuri la ștergere, indicate mai jos.



(a) Nodul  $p$  nu are descendent stâng.



(b) Nodul  $p$  nu are descendent drept.

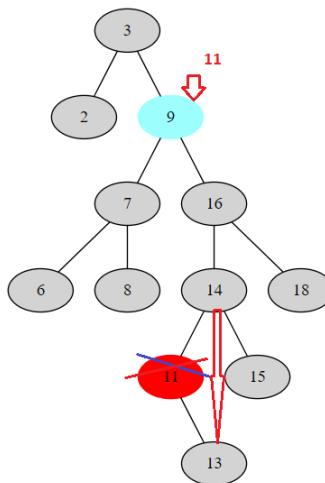


Figura 5: Nodul  $p$  are și descendent stâng și drept.

## Modelul matematic recursiv

$$sterge\_rec(abc(r, s, d), e) = \begin{cases} \emptyset & abc(r, s, d) = \emptyset \\ abc(r, sterge\_rec(s, e), d) & e.c < r.c \\ abc(r, s, sterge\_rec(d, e)) & e.c > r.c \\ d & s = \emptyset \\ s & d = \emptyset \\ abc(minim(d), s, sterge\_rec(d, minim(d))) & altfel \end{cases}$$

**Subalgoritm** **sterge**(*abc*, *e*)  
 {complexitate timp:  $O(h)$ }  
**pre:** *abc* este un **container** reprezentat folosind un arbore binar de căutare  
**post:** *abc'* este containerul din care a fost șters *e*  
 $abc.rad \leftarrow \text{sterge\_rec}(abc.rad, e)$   
**SfSubalgoritm**

**Functia** **sterge\_rec**(*p*, *e*)  
 {complexitate timp:  $O(h)$ }  
**pre:** *p* este adresa unui nod; *p* : $\uparrow$  Nod - rădăcina unui subarbore  
**post:** se șterge nodul având cheia egală cu cheia lui *e* din subarborele de rădăcină *p* și se returnează rădăcină nouui subarbore  
 {dacă s-a ajuns la subarbore vid}  
**Daca** *p*=NIL **atunci**  
 $\text{sterge\_rec} \leftarrow \text{NIL}$   
**altfel**  
**Daca** *e.c* < [*p*].*e.c* **atunci**  
 {se șterge din subarborele stâng}  
 $[p].st \leftarrow \text{sterge\_rec}([p].st, e)$   
 $\text{sterge\_rec} \leftarrow p$   
**altfel**  
**Daca** *e.c* > [*p*].*e.c* **atunci**  
 {se șterge din subarborele drept}  
 $[p].dr \leftarrow \text{sterge\_rec}([p].dr, e)$   
 $\text{sterge\_rec} \leftarrow p$   
**altfel**  
 {am ajuns la nodul care trebuie șters}  
**Daca**  $[p].st \neq \text{NIL} \wedge [p].dr \neq \text{NIL}$  **atunci**  
 {nodul are și subarbore stâng și subarbore drept}  
 $temp \leftarrow \text{minim}([p].dr)$   
 {se mută cheia minimă în *p*}  
 $[p].e \leftarrow [temp].e$   
 {se șterge nodul cu cheia minimă din subarborele drept}  
 $[p].dr \leftarrow \text{sterge\_rec}([p].dr, [p].e)$   
 $\text{sterge\_rec} \leftarrow p$   
**altfel**  
 $temp \leftarrow p$   
**Daca**  $[p].st = \text{NIL}$  **atunci**  
 {nu există subarbore stâng}  
 $repl \leftarrow [p].dr$   
**altfel**

```

{nu există subarbore drept, [p].dr= NIL}
repl ← [p].st
SfDaca
{deallocă spațiul de memorare pentru nodul care trebuie șters}
dealloca(temp)
sterge_rec ← repl
SfDaca
SfDaca
SfDaca
SfDaca
SfFunctia

```

În directorul asociat Cursului 12 găsiți implementarea parțială, în limbajul C++, a containerului **Colecție** cu elemente de tip comparabil, reprezentarea este sub forma unui ABC reprezentat înlănțuit, cu alocare dinamică a nodurilor). Iteratorul (în inordine) nu este implementat.

## Probleme

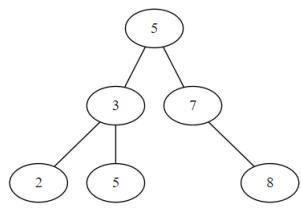
1. Scrieți o operație nerecursivă care determină părintele unui nod  $p$  dintr-un ABC.
2. Scrieți varianta iterativă pentru operația de adăugare într-un ABC.
3. Presupunând că dorim ca fiecare nod din arbore să memoreze următoarele: informația utilă, referință către subarborele stâng, referință către subarborele drept, referință către părinte. Folosind reprezentarea înlănțuită cu alocare dinamică a nodurilor, scrieți operația de adăugare în ABC (varianta iterativă, varianta recursivă).
4. Presupunând ca elementele sunt de forma (cheie, valoare) și relația de ordine între chei este “ $\leq$ ”, scrieți operația **MAXIM** care determină elementul din ABC având cea mai mare cheie.
5. Presupunând ca elementele sunt de forma (cheie, valoare), relația de ordine între chei este “ $\leq$ ”, și arborele este reprezentat înlănțuit cu alocare dinamică a nodurilor, scrieți operația **PREDECESOR** care pentru un nod  $p$  dintr-un ABC determină elementul având cea mai mare cheie mai mică decât cheia lui  $p$ .
6. Implementați operațiile pe ABC generalizând relația “ $\leq$ ” de la proprietatea unui ABC la o relație de ordine oarecare  $R$ .

# ARBORI DE CĂUTARE ECHILIBRATI

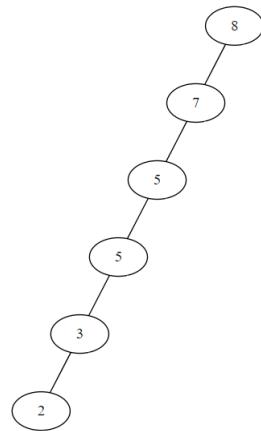
(BALANCED SEARCH TREES)

## Analiza arborilor binari de căutare

- operațiile specifice se execută în timp dependent de înălțimea arborelui (complexitate timp  $O(h)$ ).
- în cel mai rău caz pentru  $n$  elemente înălțimea este  $n - 1$  (arbore degenerat)  $\Rightarrow \theta(n)$  înălțime (complexitate în caz defavorabil pentru operații).
- cazul ideal: arbore echilibrat a cărui înălțime să fie  $O(\log_2 n)$ .



(a) ABC echilibrat.



(b) ABC degenerat (lanț).

- ideea: la fiecare nod să păstrăm *echilibrarea*.
- când un nod își pierde *echilibrul*  $\Rightarrow$  **reechilibrare** (prin rotații specifice).
- sunt mai multe moduri de definire a echilibrării  $\Rightarrow$  variante de arbori de căutare echilibrați.
  - **arbori AVL**, arbori splay, arbori roșu-negru, B-arbori, etc.
  - caracteristică comună: înălțimea arborelui este  $O(\log_2 n)$ .

## ARBORI AVL

**Definiție 1** Un **Arbore AVL** (Adelson Velski Landis) este un ABC care satisface următoarea proprietate (**invariant AVL**):

- dacă  $x$  este un nod al AVL, atunci:

- înălțimea subarborelui stâng al lui  $x$  diferă de înălțimea subarborelui drept al lui  $x$  cu 0, 1 sau -1 (0, 1 sau -1 se numește **factor de echilibrare**).

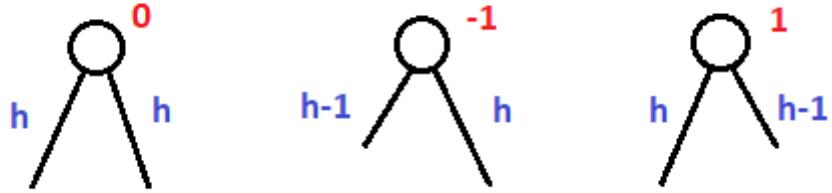


Figura 2: Factori de echilibrare posibili la orice nod al unui AVL.

- în AVL cheile (elementele) memorate în noduri sunt distincte.

**Exemplu** Presupunem că în container avem cheile  $4 \ 5 \ 6 \ 7 \ 8 \ 10$  și relația  $\mathcal{R} = \leq$ . În Figura ?? sunt indicați 2 arbori care conțin aceste chei. Cel din stânga nu este AVL, pe când cel din dreapta este AVL.

- la aborele din Figura ??(a) va fi necesară o rotație în subarborele marcat, pentru a-l echilibra.

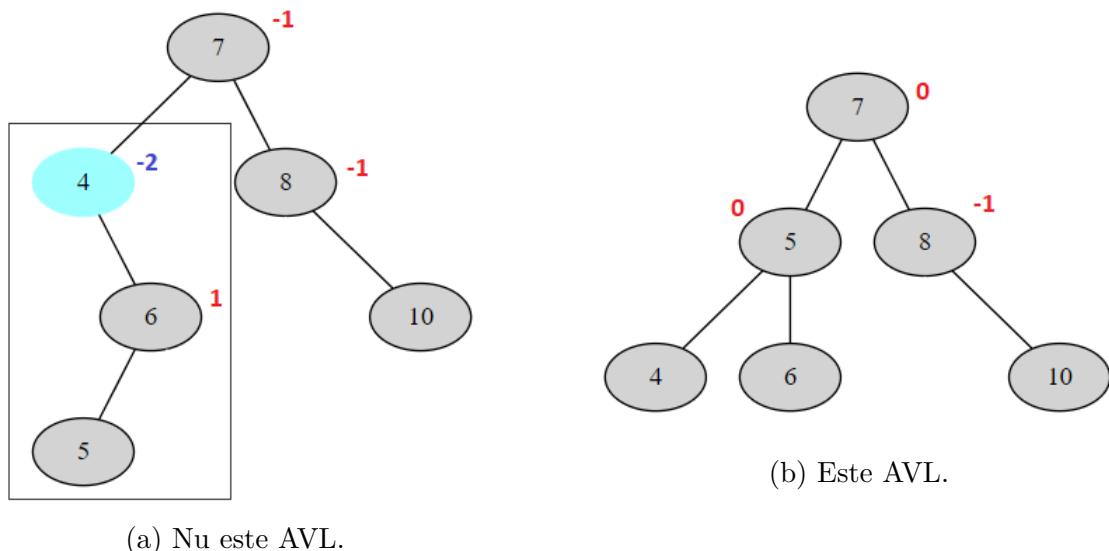


Figura 3: Arboare care conțin aceeași mulțime de chei: cel din stânga nu e echilibrat, cel din dreapta e echilibrat.

**Proprietate.** Înălțimea unui arbore AVL cu  $n$  noduri este  $\theta(\log_2 n)$ .

În cazul în care arborele are număr maxim de noduri ( $n$ ), acesta este **plin** (orice nod interior are factorul de echilibrare 0).  $\Rightarrow h = \theta(\log_2 n)$

- Notăm cu  $N(h)$  numărul minim de noduri ale unui arbore AVL de înălțime  $h$ .
  - toate nodurile interioare au factor de echilibrare -1 sau 1.
  - înălțimea unui arbore (a cărui rădăcină este  $p$ ) se poate defini recursiv ca fiind  $1 +$  maximul dintre înălțimea subarborelui stâng și înălțimea subarborelui drept  $h(p) = 1 + \max(h([p].st), h([p].dr))$ .

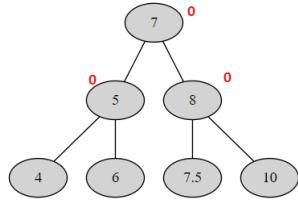


Figura 4: AVL cu număr maxim de noduri  $n$ .

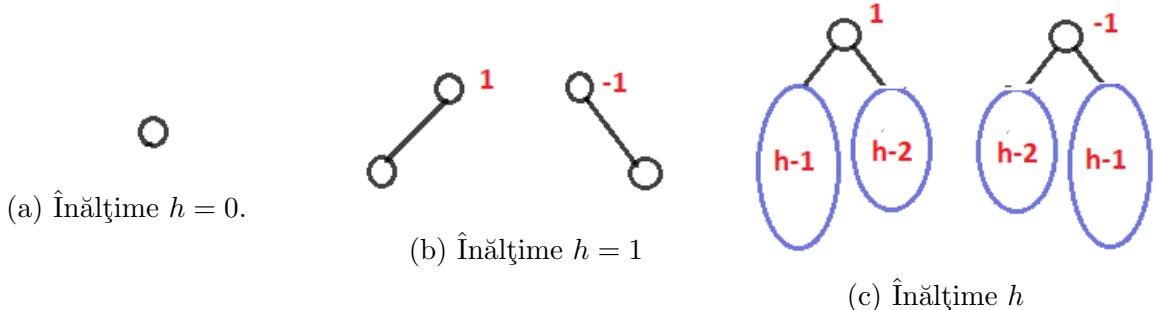


Figura 5: Număr minim de noduri în AVL

$$N(h) = \begin{cases} 1 & h = 0 \\ 2 & h = 1 \\ N(h-1) + N(h-2) + 1 & altfel \end{cases} \quad (1)$$

– se poate arăta că  $N(h) \approx \phi^h$ , unde  $\phi = \frac{1+\sqrt{5}}{2}$  este numărul de aur (*golden ratio*),  $\phi = 1.618\dots$

$$\Rightarrow h \approx \log_\phi n \in \theta(\log_2 n)$$

## Rotații și situații de reechilibrare în AVL

- 6 situații de reechilibrare (Knuth);

- în cazul **adăugării** unui element
- în cazul **ștergerii** unui element

- 4 tipuri de **rotații** pentru reechilibrare:

1. o singură rotație spre stânga (**SRS**);
2. dublă rotație spre stânga (**DRS**);
3. o singură rotație spre dreapta (**SRD**);
4. dublă rotație spre dreapta (**DRD**).

## Situatii de reechilibrare la adaugare

### Caz I - rotații spre stânga

Caz Ia) - e necesară o SRS (Figura 6)

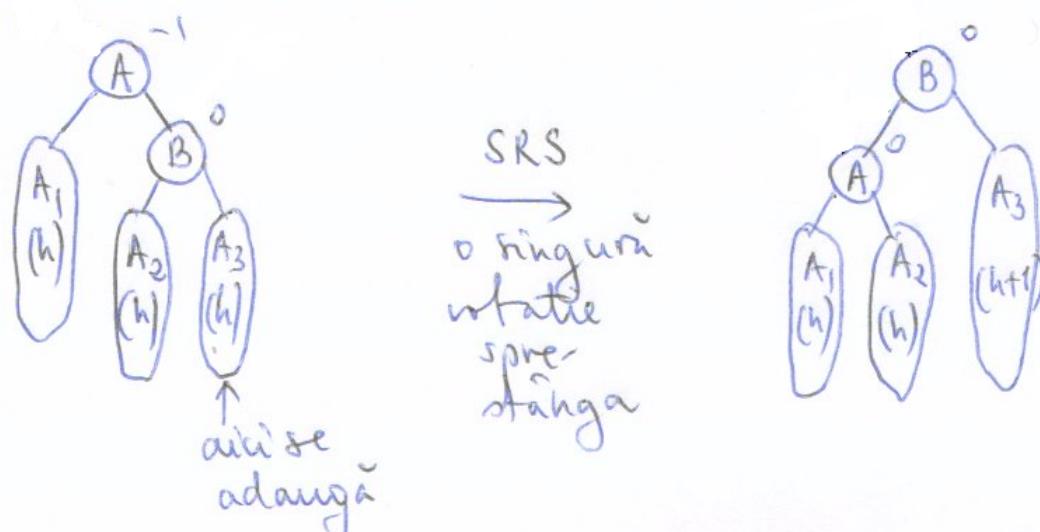


Figura 6: Caz Ia) la adăugare - e necesară o SRS pentru reechilibrare.

### Exemplu caz Ia)

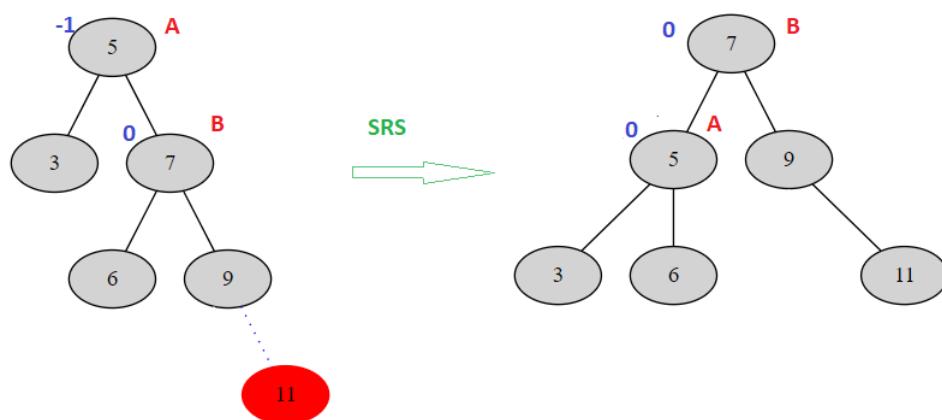


Figura 7: Exemplu caz Ia) la adăugare - e necesară o SRS pentru reechilibrare.

### !!! Atenție !!!

- la inserarea unui element, rotațiile se aplică în subarborele de înălțime minimă a cărui rădăcină și-a pierdut echilibrul (de jos în sus - de la frunze spre rădăcină)

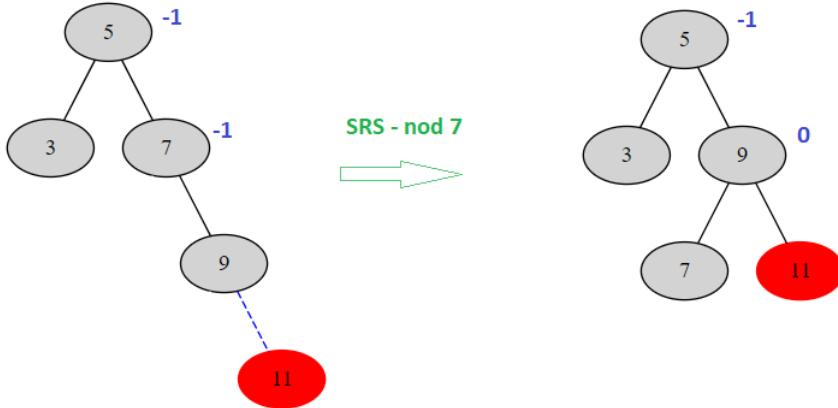


Figura 8: Echilibrul se strică la nodul 7. În subarborele de rădăcină 7 se aplică SRS pentru reechilibrare.

## Implementare SRS

- pentru implementarea operațiilor, pp. în cele ce urmează reprezentare înlățuită folosind alocare dinamică.
- pp. că fiecare nod memorează:
  - informația utilă ( $e$ );
  - adresa celor doi subarbori (stâng  $st$  și drept  $dr$ );
  - înălțimea nodului în arbore ( $h$ ).

### Nod

```

e: TElement //informația utilă nodului
st: ↑ Nod //adresa la care e memorat descendantul stâng
dr: ↑ Nod //adresa la care e memorat descendantul drept
h: ↑ Intreg //înălțimea nodului
  
```

### Functia $h(p)$

{complexitate timp:  $\theta(1)$ }

*pre:*  $p : \uparrow \text{Nod}$

*post:* se returnează înălțimea lui  $p$

{dacă e subarbore vid}

Daca  $p = \text{NIL}$  atunci

$h \leftarrow -1$

altfel

$h \leftarrow [p].h$

SfDaca

SfFunctia

### Functia $inaltime(p)$

{complexitate timp:  $\theta(1)$ }

*pre:*  $p : \uparrow \text{Nod}$

*post:* recalculează înălțimea lui  $p$  pe baza înălțimilor subarborilor lui  $p$

{dacă e subarbore vid}

Daca  $p = \text{NIL}$  atunci

$inaltime \leftarrow -1$

altfel

{se recalculează înălțimea lui  $p$  pe baza înălțimilor celor doi fiți  
 $\text{inaltime} \leftarrow \max(\text{h}([p].st), \text{h}([p].dr)) + 1$ }

SfDaca

SfFunctia

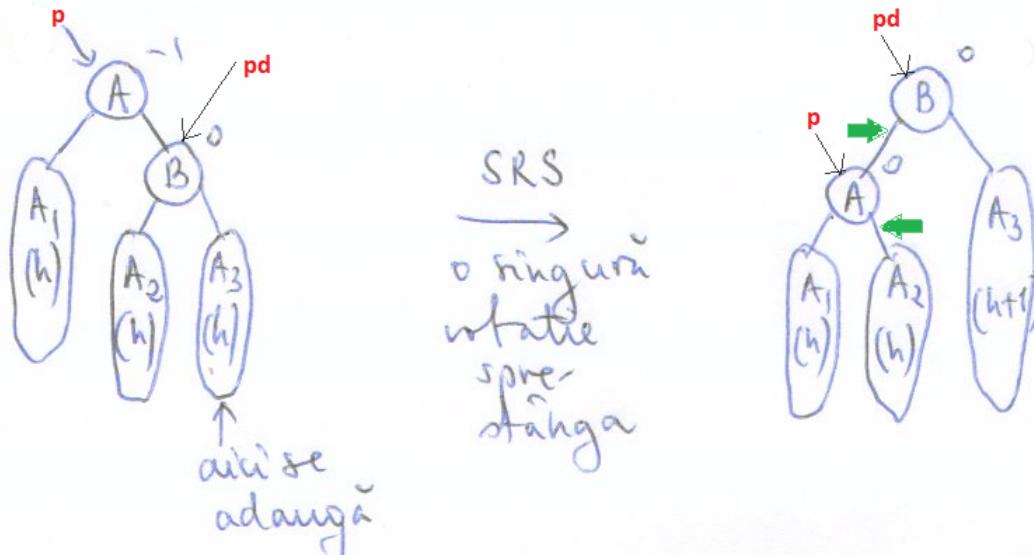


Figura 9: Situația de SRS pentru reechilibrare.

Functia  $\text{SRS}(p)$

{complexitate timp:  $\theta(1)$ }

pre:  $p$  este adresa unui nod;  $p : \uparrow \text{Nod}$  este rădăcina unui subarbore

post: se returnează rădăcina noului subarbore rezultat în urma unei SRS aplicate arborelui cu

rădăcina  $p$

{  $pd : \uparrow \text{Nod}$  e fiul drept }

$pd \leftarrow [p].dr$

{ se restabilesc legăturile între noduri conform SRS }

$[p].dr \leftarrow [pd].st$

$[pd].st \leftarrow p$

{se recalculează înălțimile conform SRS}

$[p].h \leftarrow \text{inaltime}(p)$

$[pd].h \leftarrow \text{inaltime}(pd)$

$\text{SRS} \leftarrow pd$

SfFunctia

**Caz Ib)** - e necesară o DRS (Figura 10)

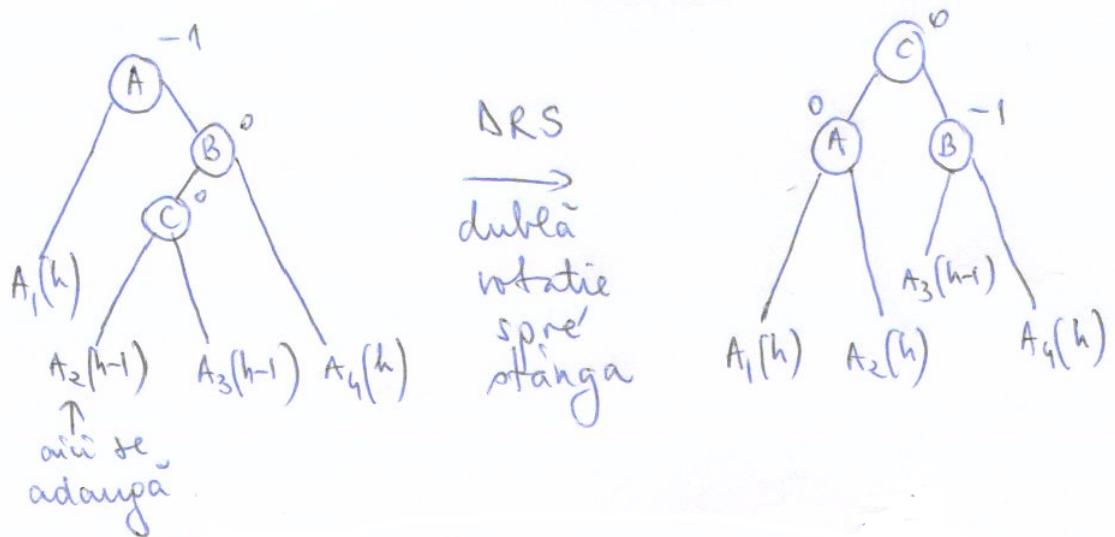


Figura 10: Caz Ib) la adăugare - e necesară o DRS pentru reechilibrare.

### Exemplu caz Ib)

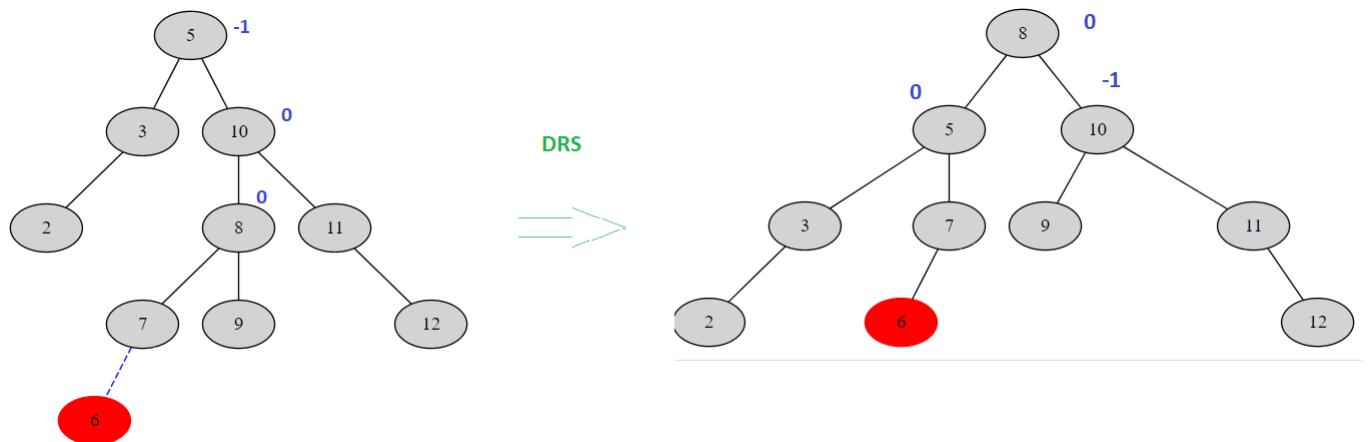


Figura 11: Exemplu caz Ib) la adăugare - e necesară o DRS pentru reechilibrare.

**Caz Ic)** - e necesară o DRS (Figura 12)

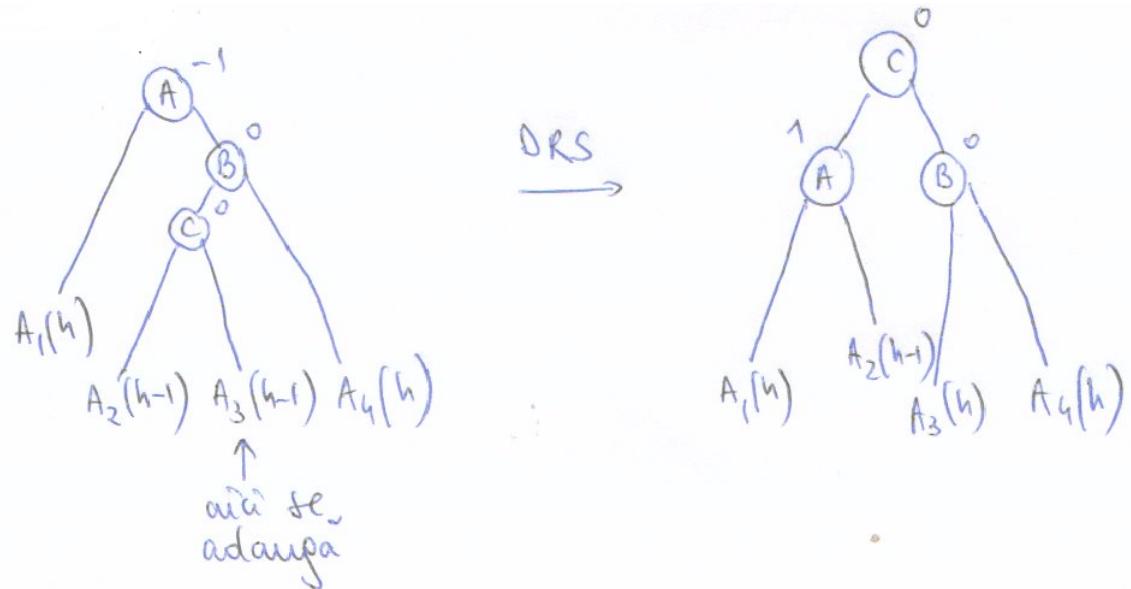


Figura 12: Caz Ic) la adăugare - e necesară o DRS pentru reechilibrare.

### Exemplu caz Ic)

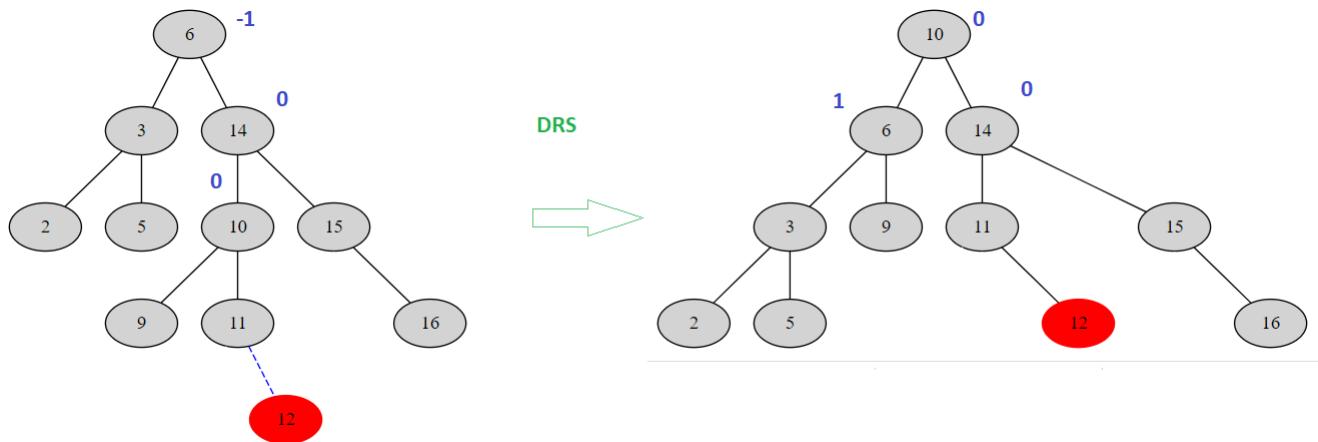


Figura 13: Exemplu caz Ic) la adăugare - e necesară o DRS pentru reechilibrare.

### Caz II - rotații spre dreapta

- simetric cu cele trei situații de la cazul I
- **Caz IIa)** - e necesară o SRD (caz simetric ca cel descris în Figura 6)
- **Caz IIb)** - e necesară o DRD (caz simetric ca cel descris în Figura 10)
- **Caz IIc)** - e necesară o DRD (caz simetric ca cel descris în Figura 12)

### Exemplu caz IIa)

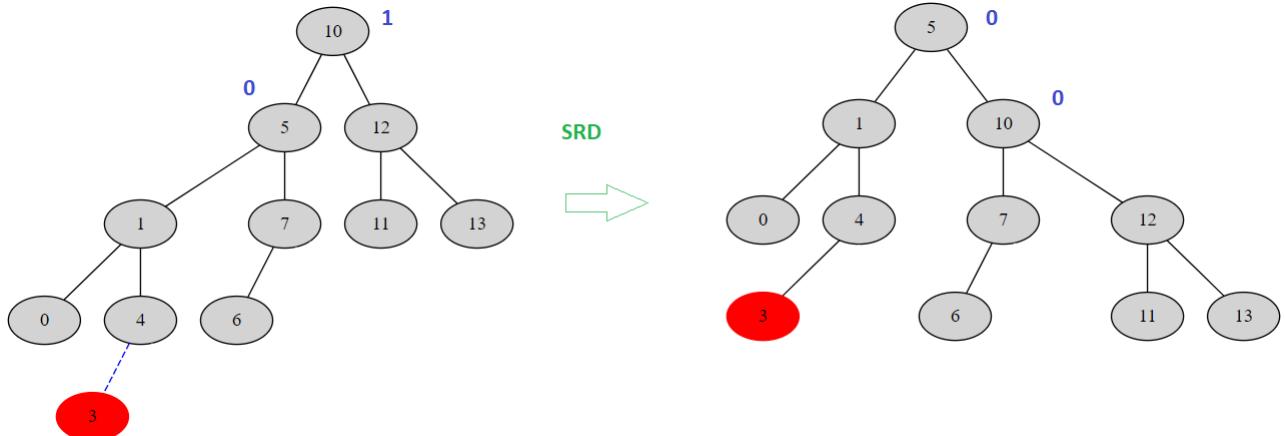


Figura 14: Exemplu caz IIa) la adăugare - e necesară o SRD pentru reechilibrare.

### Exemplu caz IIc)

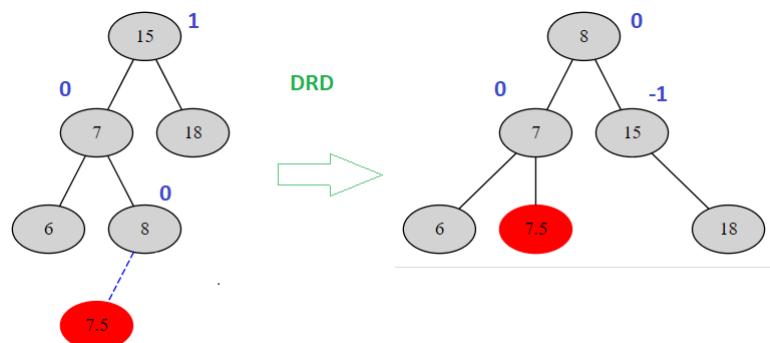


Figura 15: Exemplu caz IIc) la adăugare - e necesară o DRD pentru reechilibrare.

## Operația de adăugare în arbore AVL

PP. reprezentare înlățuită cu alocare dinamică, fiecare nod memorează și înălțimea sa.

- un element identificat de o *cheie*

```
|Functia creeazaNod(e) este { δ(l) }
{creeaza un nod avand informatia utila 'e' si cei doi descendenti NIL}
    alocă(p)      {p:↑Nod}
    [p].e←e
    [p].st←NIL
    [p].dr←NIL
    [p].h←0
    creeazaNod←p
sf creeazaNod
```

```

Functia adauga_rec(p, e) este { $O(\log_2 n)$ }
{se adauga informatia utila 'e' in subarborele de radacina 'p' si se returneaza noua
radacina a subarborelui }

Daca p=NIL atunci
    p← creeazaNod(e)
altfel
    daca e.c>[p].e.c atunci
        [p].dr←adauga_rec([p].dr,e)
        daca h([p].dr)-h([p].st)=2 atunci
            daca e.c>[[p].dr].e.c atunci {caz Ia}
                p←SRS (p)
            altfel {caz Ib, Ic}
                p←DRS (p)
            sfDaca
        altfel
            [p].h←inaltime(p)
        sfDaca
    altfel
        daca e.c<[p].e.c atunci
            @ simetric pe partea stanga – rotatii spre dreapta
        altfel
            @ cheie duplicat – nu e permisa in AVL
        sfDaca
    sfDaca
    adauga_rec←p
sf_adauga_rec

```

**Subalgoritm adauga(ab, e) este** { $O(\log_2 n)$ }

{se adauga informatia utila 'e' in arborele 'ab' si se returneaza arborele rezultat}

ab.rad←adauga\_rec(ab.rad, e)

**sf\_adauga**

## Observații

- alte reprezentări posibile pentru arborele AVL (ca și pentru un ABC)
  - reprezentare înlántuită cu reprezentare înlántuirii pe tablou.
  - reprezentare secvențială, folosind ca schemă de memorare un ansamblu.
- în locul înălțimii fiecărui nod, se poate memora *factorul de echilibrare* al acestuia

## Situări de reechilibrare la ștergere

### Caz I - rotații spre stânga

**Caz Ia) și Ib)** - dacă prin ștergere din A1, înălțimea devine  $h-1$ , e necesară o SRS (Figura 16)

- e posibil ca prin ștergere din A1, înălțimea să rămână  $h \Rightarrow$  nu e necesară rotație

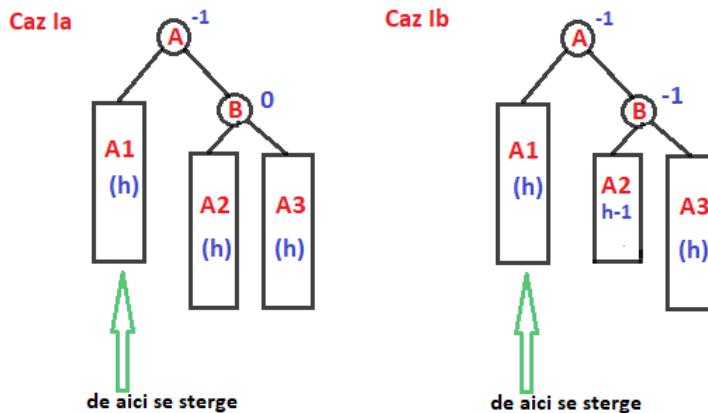


Figura 16: Caz Ia) și Ib) la ștergere - e necesară o SRS pentru reechilibrare.

### Exemple caz Ia) și caz Ib) în care sunt necesare rotații

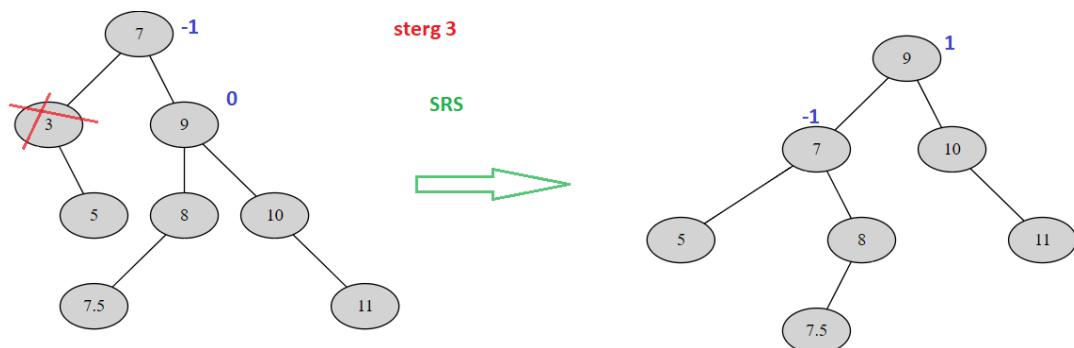


Figura 17: Exemplu caz Ia) la ștergere - e necesară o SRS pentru reechilibrare.

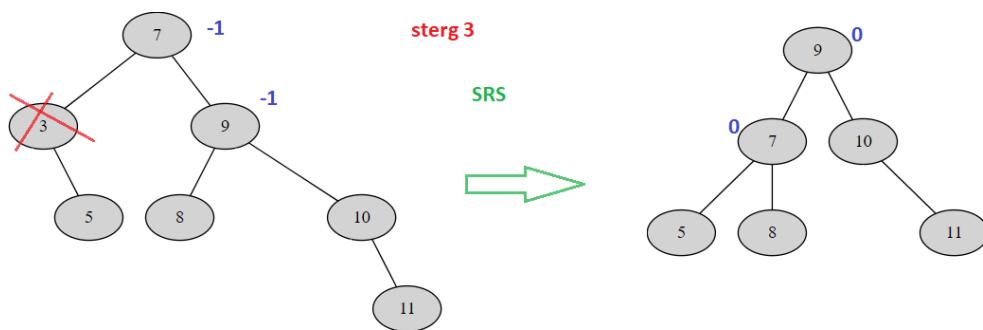


Figura 18: Exemplu caz Ib) la ștergere - e necesară o SRS pentru reechilibrare.

### Exemplu caz Ia) în care nu e necesară rotație

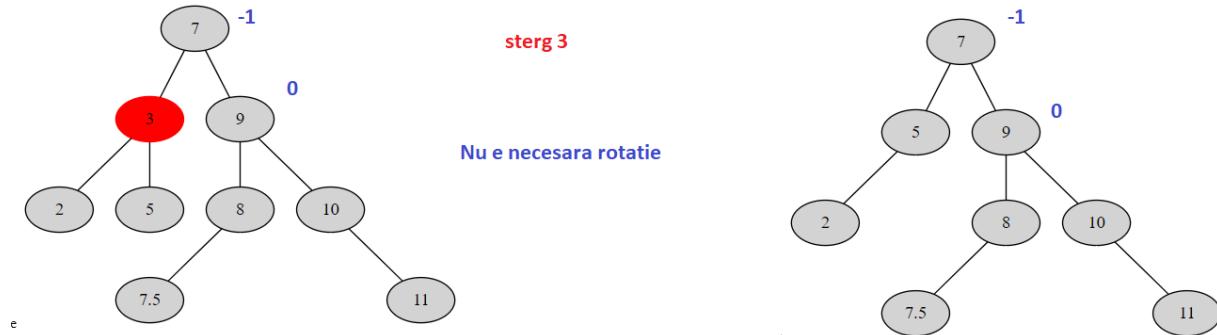


Figura 19: Exemplu caz Ia) la ștergere - nu e necesară rotație.

**Caz Ic)** - dacă prin ștergere din **A1**, înălțimea devine  $h-1$ , e necesară o DRS (Figura 20)

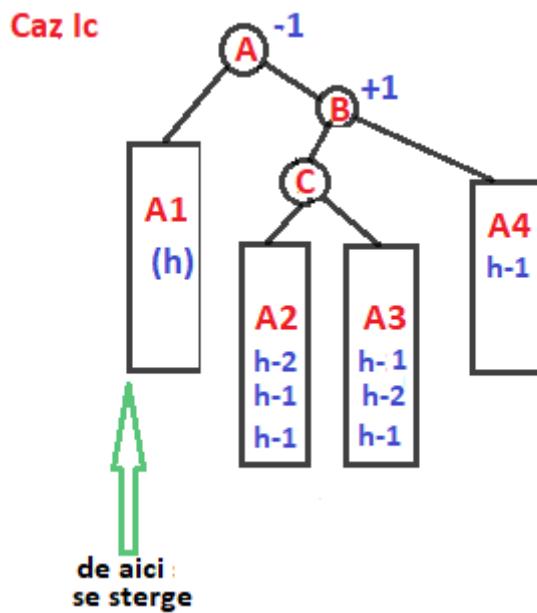


Figura 20: Caz Ic) la ștergere - e necesară o DRS pentru reechilibrare.

### Exemplu caz Ic)

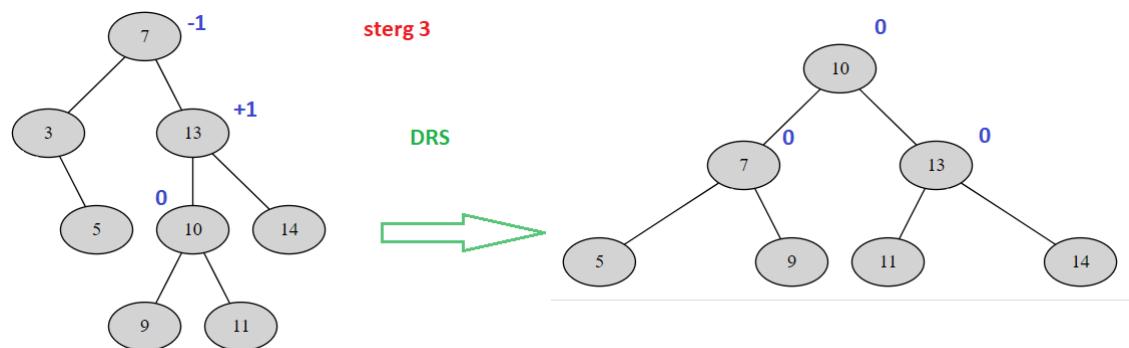


Figura 21: Exemplu caz Ic) la ştergere - e necesară o DRS pentru reechilibrare.

## Caz II - rotații spre dreapta

- simetric cu Ia)

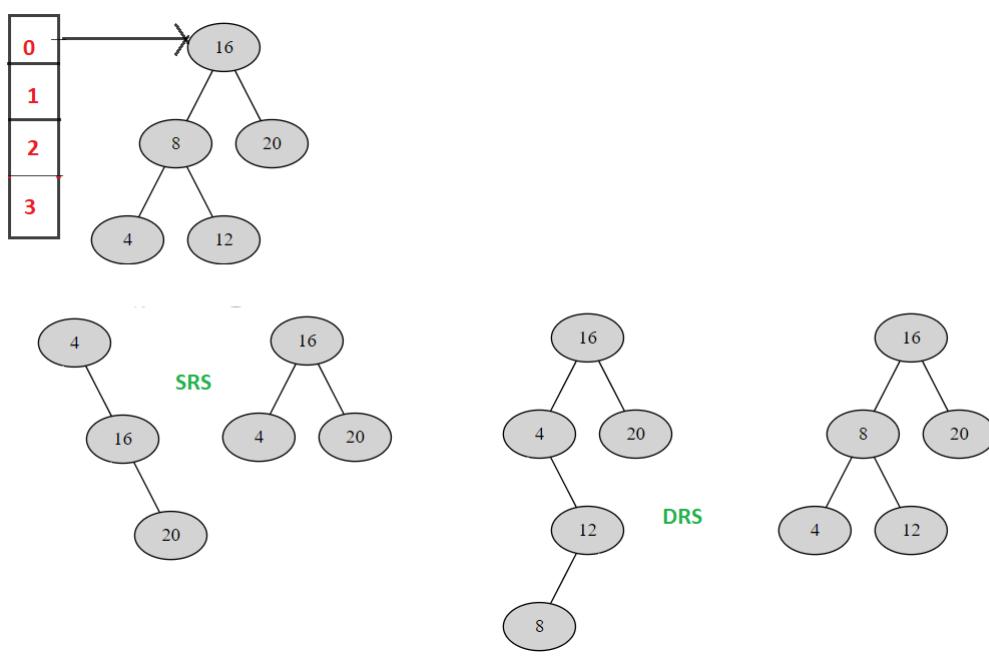
### Observație

Tabelele de dispersie cu rezolvare coliziuni prin liste independente își pot memora liste folosind arbori AVL.

- se reduce complexitatea timp defavorabil la căutare de la  $\theta(n)$  la  $\theta(\log_2 n)$ .

Fie  $m = 4$  și funcția de dispersie prin divizune.

$c(\text{heie})$	4	5	16	9	20	7	12	8
$c \bmod m$	0	1	0	1	0	3	0	0



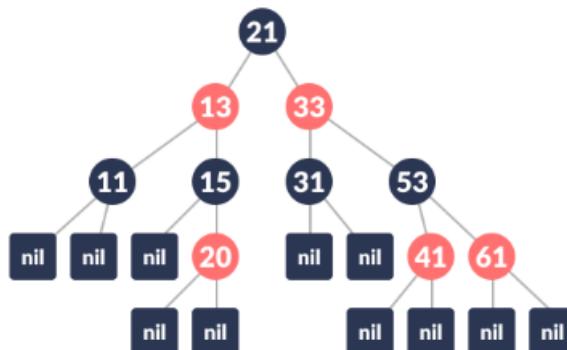
## Probleme

1. Descrieți în Pseudocod următoarele rotații: DRS, SRD, DRD.
2. Dați exemple concrete în care apare necesitatea următoarelor tipuri de rotații la adăugare/ștergere: SRS, SRD, DRS, DRD.
3. Implementați subalgoritmii discutați pe AB, ABC, AVL folosind următoarele reprezentări:
  - reprezentare înlanțuită cu reprezentare înlanțuiri pe tablou.
  - reprezentare secvențială, folosind ca schemă de memorare un ansamblu.

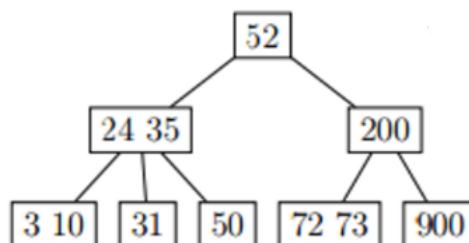
## Alte tipuri de ABC echilibrați

Înălțimea este  $O(\log_2 n)$

- arbori roșu-negru (*red-black trees*)
  - sunt ABC
  - nodurile au o culoare: *roșie* sau *neagră*
  - frunzele (**nil**) sunt negre
  - un nod roșu are cei doi fi de culoare neagră
  - pe orice drum de la rădăcină la o frunză, numărul nodurilor negre este același



- B-arbori



- generalizare ABC
- fiecare nod interior conține mai multe chei
- nodurile pot avea mai mult de 2 descendenți
  - \* dacă un nod conține 2 chei  $c_1$  și  $c_2$ , atunci are 3 descendenți
- folosiți în *baze de date* și *sisteme de fisiere*

## Examen

### Exemple grilă

1. Numărul de pași efectuat într-o căutare binară a unui element într-un vector ordonat cu  $n$  elemente este
  - a)  $O(\log_2 n)$
  - b)  $\theta(\log_2 n)$
  - c)  $\theta(n)$
  - d)  $O(n)$
  - e)  $O(\sqrt{n})$
2. Căutarea binară a unui element într-un vector ordonat cu  $n$  elemente se execută în  $O(k)$ . Cea mai mică valoare a lui  $k$  este
  - a)  $\log_2 n$
  - b)  $n$
  - c)  $\sqrt{n}$