# PROGRAMARE ȘI STRUCTURI DE DATE

## CURS 5

Lect. dr. Oneț - Marian Zsuzsanna

Facultatea de Matematică și Informatică UBB
în colaborare cu NTT Data

# În cursul 3 și 4:

- Tipuri definite de utilizator
- Subprograme
- Recursivitate
- Containere și iteratorul
  - TAD Colecție
  - TAD Mulțime

# Cuprins

# Containere I

- Un *container* este o grupare de date în care se pot adăuga (insera) elemente noi și din care se pot șterge elemente.
- Lista operațiilor pentru un container (împreună cu precondiții și postcondiții) reprezintă *interfața* containerului.
- Există mai multe containere, în funcție de diferite criterii:
  - Dacă elementele se pot repeta sau nu
  - Dacă elemente au o poziție sau nu
  - Dacă elementele stocate sunt elemente simple sau perechi
  - Dacă orice element poate fi accesat din container sau doar anumite elemente

# Containere II

- În cursul 4 am vorbit despre 2 containere:
  - Colecție
    - elementele se pot repeta
    - nu există poziții
  - Mulțime
    - elementele nu se pot repeta (sunt unice)
    - nu există poziții

# TAD Listă

- Să considerăm problema următoare: *vrem să implementăm o aplicație care să găsească cel mai scurt traseu dintr-un punct A într-un punct B. Traseul este o secvență de străzi, care ar trebui păstrate într-un container.*
- Care ar trebui să fie caracteristicile acestui container?

# TAD Listă

- Să considerăm problema următoare: *vrem să implementăm o aplicație care să găsească cel mai scurt traseu dintr-un punct A într-un punct B. Traseul este o secvență de străzi, care ar trebui păstrate într-un container.*
- Care ar trebui să fie caracteristicile acestui container?
  - Ordinea elementelor contează (trebuie să avem o primă stradă, a 2-a stradă, etc., elementele au poziții)
  - Elementele se pot repeta
- Containerul în care elementele au o poziție, se numește **Lista** (**List** în engleză).

# TAD Listă

- TAD Listă reprezintă o secvență de elemente, aflate într-o anumită ordine, fiecare element având o *poziție* bine determinată în cadrul listei.

# TAD Listă - Interfață I

- Ce operații ar trebui să aibă containerul *Listă*?

# TAD Listă - Interfață II

- creează ()
  - **descriere:** creează o listă nouă, vidă (constructorul listei)
  - **pre:** adevărat
  - **post:** *lista* a fost creată, *lista* este o Listă vidă

# TAD Listă - Interfață III

- lst.adaugaSfarsit (el)
  - **descriere:** adaugă un element nou la sfârșitul listei
  - **pre:** *lst* este o Listă, *el* este un TElement
  - **post:** *el* a fost adăugat în *lst*, *el* este ultimul element din *lst*

# TAD Listă - Interfață IV

- lst.adaugaPozitie (el, poz)
  - **descriere:** adaugă un element pe o poziție dată din listă
  - **pre:** *lst* este o Listă, *el* este un TElement, *poz* este un Întreg, 0 <= *poz* <= lungimea lui *lst* (adică *poz* este poziție validă)
  - **post:** *el* a fost adăugat în *lst* pe poziția *poz* (adică, după adăugare, elementul de pe poziția *poz* este *el*, elementele din listă începând de la poziția *poz* au fost mutate cu o poziție în spate)
  - **aruncă:** excepție dacă *poz* nu este o poziție validă

# TAD Listă - Interfață V

- lst.sterge (el)
  - **descriere:** șterge prima apariție a unui element din listă
  - **pre:** *lst* este o Listă, *el* este un TElement
  - **post:** prima apariție a lui *el* a fost ștearsă din *lst*. Elementele de după cel șters sunt mutate în față cu o poziție. Dacă *el* nu există în *lst*, *lst* nu este modificată.

# TAD Listă - Interfață VI

- lst.sterge (poz)
  - **descriere:** șterge un element de pe o poziție dată
  - **pre:** *lst* este o Listă, *poz* este un Întreg, 0 <= *poz* < lungimea lui *lst* (adică *poz* este poziție validă)
  - **post:** elementul de pe poziția *poz* a fost ștearsă din *lst* (elementele de după poziția *poz* au fost mutate cu o poziție în față).
  - **aruncă:** excepție dacă *poz* nu este o poziție validă

# TAD Listă - Interfață VII

- lst.element (poz)
  - **descriere:** returnează elementul de pe o poziție dată
  - **pre:** *lst* este o Listă, *poz* este un Întreg, 0 <= *poz* < lungimea lui *lst* (adică *poz* este poziție validă)
  - **post:** element returnează elementul de pe poziția *poz*
  - **aruncă:** excepție dacă *poz* nu este o poziție validă

# TAD Listă - Interfață VIII

- lst.modifica(poz, el)
  - **descriere:** modifică valoarea elementului de pe o poziție dată
  - **pre:** *lst* este o Listă, *poz* este un Întreg, 0 <= *poz* < lungimea lui *lst* (adică *poz* este poziție validă), *el* este un TElement
  - **post:** valoarea elementului de pe poziția *poz* devine *el*. modifica returnează valoarea veche de pe poziția *poz*.
  - **aruncă:** excepție dacă *poz* nu este o poziție validă

# TAD Listă - Interfață IX

- lst.pozitie (el)
  - **descriere:** returnează prima poziție pe care se găsește un element
  - **pre:** *lst* este o Listă, *el* este un TElement
  - **post:** poziție returnează prima poziție unde se găsește elementul *el*, sau -1 dacă nu se găsește în *lst*.

# TAD Listă - Interfață X

- lst.cauta (el)
  - **descriere:** verifică dacă un element se găsește în listă
  - **pre:** *lst* este o Listă, *el* este un TElement
  - **post:** cauta returnează Adevărat, dacă *el* apare în *lst*, sau Fals.

# TAD Listă - Interfață XI

- lst.vida ()
  - **descriere:** verifică dacă o listă este vidă
  - **pre:** *lst* este o Listă
  - **post:** vida returnează Adevărat, dacă *lst* nu conține niciun element, sau Fals.

# TAD Listă - Interfață XII

- lst.dim ()
  - **descriere:** returnează numărul de elemente din listă
  - **pre:** *lst* este o Listă
  - **post:** dim returnează numărul de elemente din *lst*

# TAD Listă - Interfață XIII

- lst.iterator ()
  - **descriere:** creează și returnează un iterator pentru o listă
  - **pre:** *lst* este o Listă
  - **post:** iterator returnează un iterator pentru *lst*

# Lista în Java

- Containerul Listă există în Java, se numește **List**
- Lista operațiilor pentru **List** se găsește la adresa: https://docs.oracle.com/javase/8/docs/api/java/util/List.html

# Lista - exemplu de utilizare

- Ca exemplu de utilizare la Colecție am folosit problema următoare:
  - *Să reprezentăm conținutul unui portmoneu. La început se citesc pe rând valorile bancnotelor din portmoneu (până se citește valoarea 0). Afișați suma totală pe care o avem.*

# Lista - exemplu de utilizare

- Ca exemplu de utilizare la Colecție am folosit problema următoare:
  - *Să reprezentăm conținutul unui portmoneu. La început se citesc pe rând valorile bancnotelor din portmoneu (până se citește valoarea 0). Afișați suma totală pe care o avem.*
- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind o listă?

# Lista - exemplu de utilizare

- Ca exemplu de utilizare la Colecție am folosit problema următoare:
  - *Să reprezentăm conținutul unui portmoneu. La început se citesc pe rând valorile bancnotelor din portmoneu (până se citește valoarea 0). Afișați suma totală pe care o avem.*
- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind o listă?
- Da, putem rezolva problema și cu o listă. Pentru această problemă nu contează ordinea elementelor, dar nu e o problemă nici dacă folosim un container în care există poziții.

# Lista - exemplu de utilizare

- Din nou vom împărți rezolvarea problemei în 3 subprograme:
  - Un subprogram care citește bancnotele
  - Un subprogram care calculează suma totală
  - Programul principal

# Lista - exemplu de utilizare - citire portmoneu

```
subalgoritm citesteBancnote(portmoneu: Lista) este:
    val: întreg
    val = 1
    câttimp val ≠ 0 execută
        scrie "Valoarea bancnotei:"
        citește val
        dacă val ≠ 0 atunci //să nu adăugăm ultimul 0
            portmoneu.adaugaSfarsit(val)
            //am putea folosi și alte funcții pentru adăugare
        sf_dacă
    sf_câttimp
sf_subalgoritm
```

# Lista - exemplu de utilizare

- Cum calculăm suma bancnotelor din portmomenu?

# Lista - exemplu de utilizare

- Cum calculăm suma bancnotelor din portmomenu?
- De data asta avem poziții, deci putem parcurge lista folosind pozițiile. Dar Lista are și iterator, deci am putea folosi și iteratorul.
- În general, când iteratorul poate fi folosit, preferăm să folosim iteratorul, dar acum vom face o parcurgere cu poziții (parcurgerea cu iterator este exact la fel ca la Colecție sau Mulțime).

# Lista - exemplu de utilizare - sumă

**funcție** sumaTotală(portmoneu: Lista) **este**:
  *//portmoneu este o Lista în care avem bancnote*
  suma, val, i: întreg
  suma = 0 *//aici vom calcula suma*
  **pentru** i = 0, portmoneu.dim(), 1, **execută**
    val = portmoneu.element(i)
    suma = suma + val
  **sf_pentru**
  **returnează** suma *//returnăm suma*
**sf_funcție**

# Lista - exemplu de utilizare - program principal

- Ne mai trebuie programul principal care să apeleze funcțiile făcute și să afișeze suma totală

**algoritm** BaniiMei **este:**
  portmoneu: Lista *//definesc o variabilă de tip Lista*
  citesteBancnote(portmoneu)
  suma: întreg
  suma = sumaTotală(portmoneu)
  **scrie** "Suma totală în portmoneu este: " + suma
**sf_algoritm**

# TAD Stivă

- Să ne imaginăm un restaurant în care farfuriile curate sunt stocate una peste alta (ca în figura din dreapta).
- Dacă ar trebui să luați o farfurie (să puneți mâncare în ea) pe care ați lua-o?
- Dacă ar trebui să adăugați o farfurie nouă (acum ați spălat-o), unde ați pune-o?

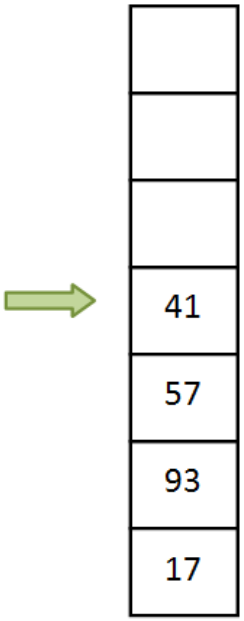
Figure: Sursă: http://www.alaintruong.com/archives/2013/10/04/28146084.html

# TAD Stivă

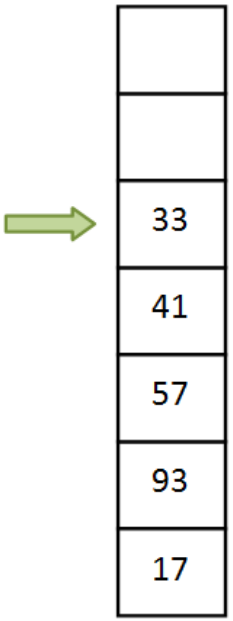
- **Stiva** (**Stack** în engleză) este un container unde accesul la elemente este restricționat la un singur capăt al containerului. Acest capăt se numește *vârful* stivei.
  - Dacă adaug un element nou, elementul este adăugat automat la *vârful* stivei.
  - Dacă șterg un element, se șterge automat elementul din vârful stivei.
  - Pot accesa doar elementul din vârful stivei.
- Din cauza restricției la accesarea elementelor, spunem că Stiva funcționează după un model **LIFO** (Last In First Out) - ultimul element adăugat va fi primul element șters.

# TAD Stivă - Exemplu

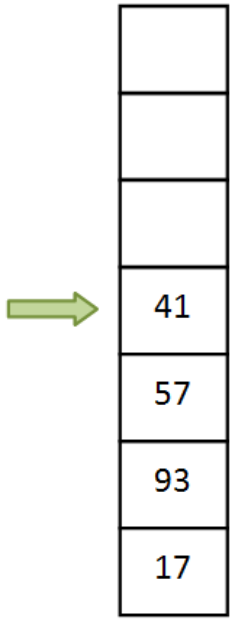
- Să presupunem că avem următoarea stivă (săgeata verde marchează vârful stivei):




- Adăugăm numărul 33:

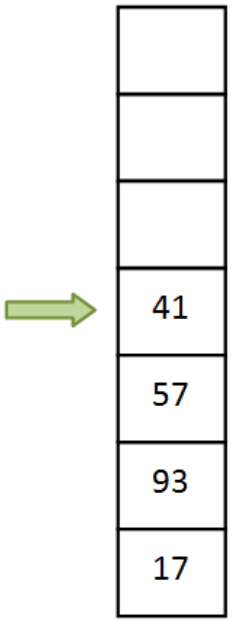



- Ștergem un element:

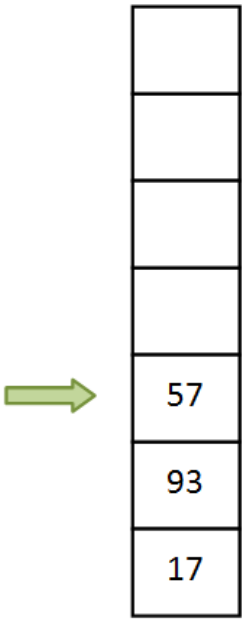


# TAD Stivă - Exemplu

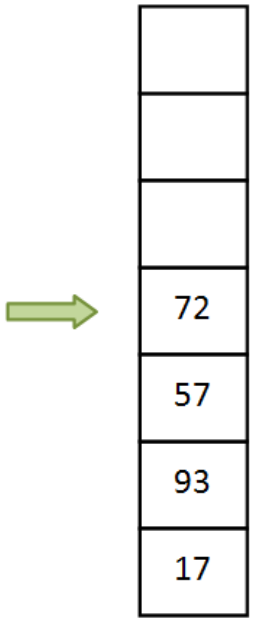
- Aceasta este stiva noastră:




- Ștergem încă un element:




- Adăugăm numărul 72:



# TAD Stivă - Interfață I

- Din cauza accesului restricționat, TAD-ul Stivă are puține operații.

# TAD Stivă - Interfață II

- creeaza()
  - **Descriere:** creează o stivă nouă, vidă
  - **Pre:** Adevărat
  - **Post:** o Stivă vidă este creată

# TAD Stivă - Interfață III

- s.adauga(e)
  - **Desciere:** adaugă un element la vârful stivei (în engleză operația în general se numește *push*)
  - **Pre:** *s* este o Stivă, *e* este TElement
  - **Post:** *e* este adăugat în *s*, *e* este cel mai recent adăugat element din *s*

# TAD Stivă - Interfață IV

- s.sterge()
  - **Descriere:** șterge și returnează elementul de la vârful stivei (în engleză operația în general se numește *pop*)
  - **Pre:** *s* este o Stivă, *s* nu este vidă
  - **Post:** din *s* se șterge elementul *e*, *e* este un TElement, *e* este cel mai recent adăugat element în *s*, sterge returnează *e*
  - **Aruncă:** excepție, dacă *s* este vidă.

# TAD Stivă - Interfață V

- s.element()
  - **Descriere:** returnează elementul de pe vârful stivei, dar nu îl șterge (în engleză operația se numește *peek* sau *top*)
  - **Pre:** *s* este o Stivă, *s* nu este vidă
  - **Post:** element returnează e, *e* este un TElement, *e* este cel mai recent element din *s*
  - **Aruncă:** excepție, dacă *s* este vidă

# TAD Stivă - Interfață VI

- s.vida()
  - **Descriere:** verifică dacă stiva este vidă
  - **Pre:** *s* este o Stivă
  - **Post:** vida returnează Adevărat, dacă *s* nu conține niciun element, Fals, altfel.

# TAD Stivă - Interfață VII

- **Observație:** Stiva nu poate fi iterată, nu are iterator! Nu avem metode de a *vedea* toate elementele Stivei, doar cel de pe vârf. (Dacă vrem să vedem restul elementelor trebuie să tot ștergem din stivă).

# Stiva în Java

- În Java există containerul Stivă, se numește **Stack**.
- Lista operațiilor pentru **Stack** se găsește la adresa: https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html

# Stiva - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind o stivă?

# Stiva - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind o stivă?
- La prima vedere se pare că nu putem rezolva problema folosind o stivă, în principiu pentru că nu avem metode de a itera elementele stivei (ca să calculăm suma).
- O opțiune, dacă totuși vrem să folosim stiva, este ca la calculul sumei, când ștergem elemente din stivă, să le adăugăm într-o altă stivă auxiliară, iar la final să le mutăm la loc.

# Stiva - exemplu de utilizare

- Din nou vom împărți rezolvarea problemei în 3 subprograme:
  - Un subprogram care citește bancnotele
  - Un subprogram care calculează suma totală
  - Programul principal

# Stiva - exemplu de utilizare - citire portmoneu

```
subalgoritm citesteBancnote(portmoneu: Stiva) este:
    val: întreg
    val = 1
    câttimp val ≠ 0 execută
        scrie "Valoarea bancnotei:"
        citește val
        dacă val ≠ 0 atunci //să nu adăugăm ultimul 0
            portmoneu.adauga(val)
        sf_dacă
    sf_câttimp
sf_subalgoritm
```

# Stiva - exemplu de utilizare - sumă

```
funcție sumaTotală(portmoneu: Stiva) este:
    suma, val: întreg
    aux: Stiva
    suma = 0 //aici vom calcula suma
    câttimp NOT portmoneu.vida() execută
        val = portmoneu.sterge()
        suma = suma + val
        aux.adauga(val)
    sf_câttimp
    //si acum mutăm înapoi elementele în portmoneu
    câttimp NOT aux.vida() execută
        val = aux.sterge()
        portmoneu.adauga(val)
    sf_câttimp
    returnează suma //returnăm suma
sf_funcție
```

# Stiva - exemplu de utilizare - sumă

- Din portmoneul (stiva) mutăm elementele într-o altă stivă, după care mutăm elementele înapoi în portmoneu. Ce se întâmplă cu ordinea elementelor pe parcursul mutărilor?

# Stiva - exemplu de utilizare - program principal

- Ne mai trebuie programul principal care să apeleze funcțiile făcute și să afișeze suma totală

**algoritm** BaniiMei **este:**
  portmoneu: Stiva *//definesc o variabilă de tip Stiva*
  citesteBancnote(portmoneu)
  suma: întreg
  suma = sumaTotală(portmoneu)
  **scrie** "Suma totală în portmoneu este: " + suma
**sf_algoritm**

# TAD Coadă


Figure: Sursă: www.123rf.com

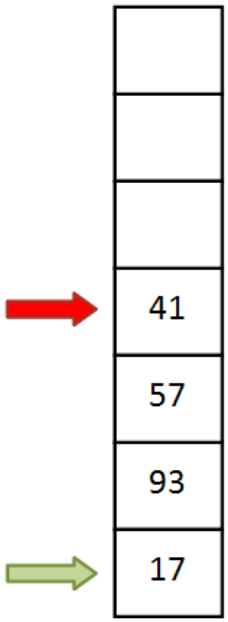
- Să ne imaginăm că stăm la coadă la biroul de check-in din aeroport.
- Dacă vine cineva care vrea să facă check-in, unde ar trebui să se pună?
- Dacă termină check-in-ul persoana care e acum la birou, cine va fi următoarea persoană care va merge la birou?

# TAD Coada

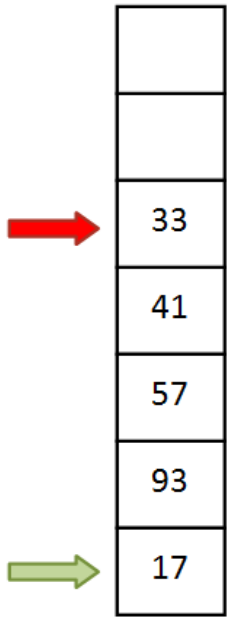
- **Coada** (**Queue** în engleză) este un container unde accesul la elemente este restricționat la cele 2 capete ale containerului. Aceste 2 capete se numesc *front* și *end*.
  - Dacă adaug un element nou, elementul este adăugat automat la un capăt al cozii (*end*).
  - Dacă șterg un element, se șterge automat din celălalt capăt (*front*).
  - Pot accesa doar elementul din *front*.
- Din cauza restricției la accesarea elementelor, spunem că Coada funcționează după un model **FIFO** (First In First Out) - primul element adăugat va fi primul element șters.

# TAD Coadă - Exemplu

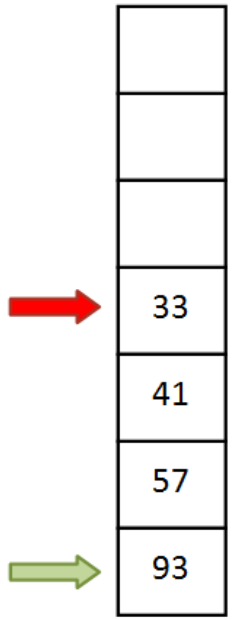
- Să presupunem că avem următoarea coada (săgeata verde e *front*-ul, cea roșie e *end*-ul):




- Adăugăm numărul 33:

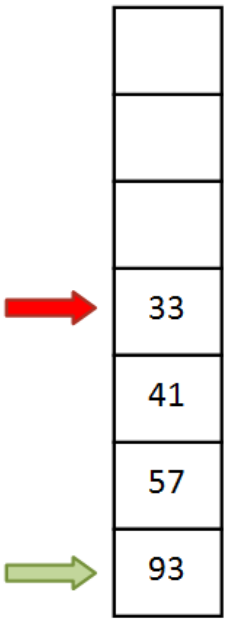



- Ștergem un element:

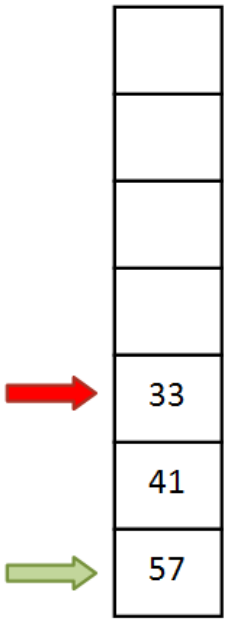


# TAD Coadă - Exemplu

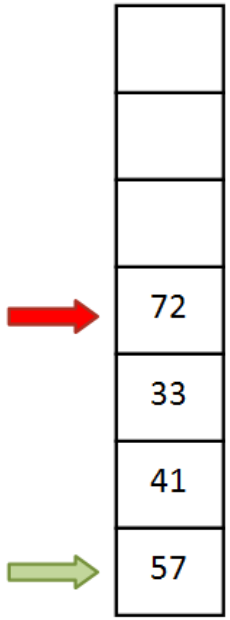
- Aceasta este coada noastră:




- Ștergem încă un element:




- Adăugăm numărul 72:



# TAD Coadă - Interfață I

- Din cauza accesului restricționat, TAD-ul Coadă are puține operații (denumirile operațiilor sunt la fel ca la Stivă).

# TAD Coadă - Interfață II

- creeaza()
  - **Descriere:** crează o coadă nouă, vidă
  - **Pre:** Adevărat
  - **Post:** o Coadă vidă este creată

# TAD Coadă - Interfață III

- c.adauga(e)
  - **Desciere:** adaugă un element la *end*-ul cozii (în engleză operația în general se numește *push*)
  - **Pre:** *c* este o Coadă, *e* este TElement
  - **Post:** *e* este adăugat în *c*, *e* este elementul de la *end*-ul lui *c*

# TAD Coadă - Interfață IV

- c.sterge()
  - **Descriere:** șterge și returnează elementul de la *front*-ul cozii (în engleză operația în general se numește *pop*)
  - **Pre:** *c* este o Coadă, *c* nu este vidă
  - **Post:** din *c* se șterge elementul *e*, *e* este un TElement, *e* este elementul de la *front*-ul cozii (deci elementul adăugat prima dată în *c*), sterge returnează *e*
  - **Aruncă:** excepție, dacă *c* este vidă.

# TAD Coadă - Interfață V

- c.element()
  - **Descriere:** returnează elementul din *front*-ul cozii, dar nu îl șterge (în engleză operația se numește *peek* sau *top*)
  - **Pre:** *c* este o Coadă, *c* nu este vidă
  - **Post:** returnează e, *e* este un TElement, *e* este primul element adăugat în *c*
  - **Aruncă:** excepție, dacă *c* este vidă

# TAD Coadă - Interfață VI

- c.vida()
  - **Descriere:** verifică dacă coada este vidă
  - **Pre:** *c* este o Coadă
  - **Post:** vida returnează Adevărat, dacă *c* nu conține niciun element, Fals, altfel.

# TAD Coadă - Interfață VII

- **Observație:** Coada nu poate fi iterată, nu are iterator! Nu avem metode de a *vedea* toate elementele Cozii, doar cel de pe front. (Dacă vrem să vedem restul elementelor trebuie să tot ștergem din coadă).

# Coada în Java

- În Java există coada implementată, se numește **Queue**.
- Lista operațiilor pentru **Queue** se găsește la adresa: https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

# Coadă - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind o coadă?

# Coadă - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind o coadă?
- Da, metoda folosită ca să rezolvăm problema cu o stivă, funcționează și dacă vrem să rezolvăm problema cu o coadă.

# Coadă - exemplu de utilizare

- Din moment ce la o coadă adăugarea și ștergerea se fac la două capete diferite, putem rezolva problema și fără o coadă auxiliară, adăugând elementele șterse înapoi în coada inițială.
- Problema este că așa nu mai știm când să ne oprim cu ciclul câttimp, pentru că niciodată nu ajungem la o coadă vidă. În mod normal vrem să ne oprim când primul element ajunge să fie egal cu primul element inițial, dar din cauza elementelor care se repetă nu știm când e primul element egal cu cel inițial.
- Putem adăuga în coadă un element *fictiv*, unul care să nu reprezinte un element valid, dar care să marcheze când ajungem înapoi la începutul listei.
- Vom reimplementa doar funcția *sumaTotala*

# Coada - exemplu de utilizare - sumă

**funcție** sumaTotală(portmoneu: Coadă) **este**:
    suma, val, val_speciala: întreg
    suma = 0 *//aici vom calcula suma*
    val_speciala = 99 *//daca se introduce bancnota 99, schimbăm doar aici codul*
    portmoneu.adauga(val_speciala)
    **câttimp** portmoneu.element() ≠ val_speciala **execută**
        val = portmoneu.sterge()
        suma = suma + val
        portmoneu.adauga(val)
    **sf_câttimp**
    *//portmoneu începe cu val_speciala. Trebuie ștearsă*
    portmoneu.sterge()
    **returnează** suma *//returnăm suma*
**sf_funcție**

# Lista I

- Dacă ne gândim la operațiile discutate până acum pentru TAD Colecție, Mulțime, Stivă și Coadă, toate acele operații există și pentru Listă (numai Lista mai are și alte operații în plus, cele cu poziții, Lista fiind singurul container care are poziții).
- După cum am văzut în exemplul cu portmoneu, putem folosi Lista și pentru rezolvarea unei probleme care nu necesită poziții.
- Deși nu am avut exemplu de acest gen, dar nu e greu de văzut că și problemele care necesită o Stivă sau o Coadă pot fi rezolvate folosind o Listă.
- Dacă Lista poate fi folosită pentru a rezolva și probleme unde nu sunt necesare poziții, de ce mai avem nevoie de TAD Colecție, Mulțime, Stivă și Coadă?

# Lista II

- Avantajul Colecției și Mulțimii se vede când discutăm despre cum se pot implementa aceste containere. Când nu avem nevoie de poziții, putem avea implementări mai eficiente, decât dacă trebuie să ținem cont de poziții.

- Un alt motiv pentru a folosi containerul "cel mai potrivit" este că aceste containere ne dau și un *vocabular*. Dacă în cod vedem la un moment dat că o mulțime a fost folosită, imediat știm că în containerul respectiv nu vor fi duplicate, ceea ce ajută la înțelegerea codului.

- Sau dacă se discută rezolvarea unei probleme, e mult mai ușor să spunem "vom folosi o Stivă" decât "vom folosi o listă dar tot timpul adăgăm și ștergem de la același capât".

# Dicționar I

- Să considerăm problema următoare: *vrem să facem o aplicație care "traduce" un text în echivalentul lui în codul Morse. De exemplu, pentru "SOS" vrem să avem "...- - -...". Pentru a face traducerea, trebuie să reținem undeva pentru fiecare literă codul corespunzător.*

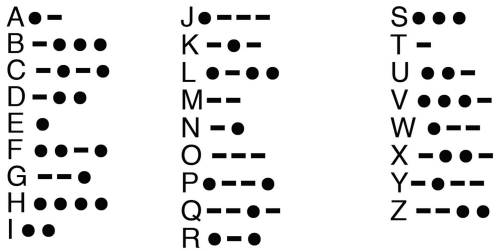



Figure: Sursă: https://electronicsforu.com/wp-contents/uploads/2016/05/morse-code.jpeg

- Ce caracteristici ar trebui să aibă containerul în care să reținem literele cu coduri morse?

# Dicționar II

- Elementele din container trebuie să fie perechi: ne trebuie și litera și codul corespunzător.
- Nu contează ordinea elementelor, dar trebuie să putem găsi codul asociat unei litere.
- Literele sunt unice (în problema noastră și codurile asociate sunt unice, dar, "traducerea" s-ar putea face și dacă codurile nu ar fi unice)

- Containerul care respectă aceste condiții, este **Dicționarul** (în engleză **Map**).

# TAD Dicționar III

- **Dicționarul** este un container în care reținem perechi de forma (cheie, valoarea).
- Cheile dintr-un dicționar trebuie să fie **unice** (nu pot exista 2 perechi cu aceeași cheie, chiar dacă valorile ar fi diferite).
- Nu există poziții într-un dicționar, accesarea elementelor se face pe baza cheilor.

# TAD Dicționar - Interfață I

- Ce operații credeți că ar trebui să existe pentru un dicționar?

# TAD Dicționar - Interfață II

- creeaza()
  - **Descriere:** creează un dicționar nou, vid
  - **Pre:** Adevărat
  - **Post:**un Dicționar vid a fost creat

# TAD Dicționar - Interfață III

- d.adauga(c, v)
  - **Descriere:** adaugă o pereche cheie - valoare în dicționar
  - **Pre:** *d* este un dicționar, *c* este un TElement, *v* este un TElement (*c* și *v* pot avea tipuri diferite)
  - **Post:** perechea *(c, v)* a fost adăugată în dicționarul *d*.

# TAD Dicționar - Interfață IV

- Cheile trebuie să fie unice într-un dicționar. Ce se întâmplă dacă apelez adaugă cu o cheie care există deja? Există mai multe opțiuni:
  - Nu se întâmplă nimic (dicționarul nu este modificat)
  - Se aruncă o excepție (și dicționarul nu este modificat)
  - Se schimbă valoarea existentă cu cea transmisă ca parametru.

# TAD Dicționar - Interfață V

- d.sterge(c)
  - **Descriere:** Șterge perechea cu o anumită cheie din dicționar (atenție, se transmite doar cheia ca parametru)
  - **Pre:** *d* este un Dicționar, c este un TElement
  - **Post:** din dicționarul *d* se șterge perechea cu cheia *c*. Presupunem că *v* este valoarea asociată cheii *c*. sterge returnează v (se șterge perechea și se returnează valoarea).

# TAD Dicționar - Interfață VI

- d.cauta(c)
  - **Descriere:** Caută dacă există vreo pereche cu o cheie dată în dicționar, dacă există, returnează valoarea asociată cheii.
  - **Pre:** *d* este Dicționar, *c* este un TElement
  - **Post:** cauta returnează *v*, dacă perechea (c, v) există în *d*, NIL (sau null) altfel.

# TAD Dicționar - Interfață VII

- d.dim ()
  - **Descriere:** Returnează numărul de perechi din dicționar
  - **Pre:** *d* este un Dicționar
  - **Post:** dim returnează numărul de perechi din *d*

# TAD Dicționar - Interfață VIII

- d.iterator()
  - **Descriere:** returnează un iterator pentru un dicționar
  - **Pre:** *d* este un Dicționar
  - **Post:** iterator returnează un iterator pentru *d*

- Iteratorul pentru un dicționar funcționează la fel ca iteratorul pentru alte containere. Operația *element* însă returnează o pereche (cheie, valoare).

# TAD Dicționar - Interfață IX

- Alte operații posibile pentru un Dicționar:

# TAD Dicționar - Interfață X

- d.chei ()
  - **Descriere:** returnează mulțimea cheilor pentru un dicționar
  - **Pre:** *d* este un Dicționar
  - **Post:** chei returnează *m*, *m* este o Mulțime, *m* conține toate cheile din *d*

# TAD Dicționar - Interfață XI

- d.valori ()
  - **Descriere:** returnează colecția valorilor pentru un dicționar
  - **Pre:** *d* este un Dicționar
  - **Post:** valori returnează *c*, *c* este o Colecție, *c* conține toate valorile din *d*

- De ce returnează operația *chei* o mulțime și *valori* o colecție?

# Dicționarul în Java

- În Java există dicționar implementat, se numește **Map**.
- Lista operațiilor pentru **Map** se găsește la adresa: https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

# Dicționar - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind un dicționar?

# Dicționar - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind un dicționar?

- La prima vedere se pare că dicționarul nu se potrivește pentru această problemă. Într-un dicționar trebuie să avem chei și valori, dar în problema noastră nu avem perechi. Și nici nu avem elemente unice.

- O opțiune, dacă totuși vrem să folosim dicționarul, este să avem valoarea bancnotelor ca cheie și numărul de apariții ca valoare.

# Dicționar - exemplu de utilizare

- Din nou vom împărți rezolvarea problemei în 3 subprograme:
  - Un subprogram care citește bancnotele
  - Un subprogram care calculează suma totală
  - Programul principal

# Dicționar - exemplu de utilizare - citire portmoneu

```
subalgoritm citesteBancnote(portmoneu: Dicționar) este:
    val, nrVechi: întreg
    val = 1
    câttimp val ≠ 0 execută
        scrie "Valoarea bancnotei:"
        citește val
        dacă val ≠ 0 atunci //să nu adăugăm ultimul 0
        //verificăm dacă avem deja în portmoneu valoarea
            dacă portmoneu.cauta(val) == NIL atunci
                portmoneu.adauga(val, 1)
            altfel
                nrVechi = portmoneu.sterge(val)
                portmoneu.adauga(val, nrVechi + 1)
            sf_dacă
        sf_dacă
    sf_câttimp
sf_subalgoritm
```

# Dicționar - exemplu de utilizare - sumă

**funcție** sumaTotală(portmoneu: Dicționar) **este**:
    suma, val, nr: întreg
    it: Iterator
    it = portmoneu.iterator()
    suma = 0 *//aici vom calcula suma*
    **câttimp** it.valid() **execută**
        <val, nr> = it.element() *//returnăm o pereche*
        suma = suma + val * nr
        it.următor()
    **sf_câttimp**
    **returnează** suma
*//returnăm suma*
**sf_funcție**

# Dicționar - exemplu de utilizare - program principal

- Ne mai trebuie programul principal care să apeleze funcțiile făcute și să afișeze suma totală

**algoritm** BaniiMei **este:**
  portmoneu: Dicționar *//definesc o variabilă de tip Dicționar*
  citesteBancnote(portmoneu)
  suma: întreg
  suma = sumaTotală(portmoneu)
  **scrie** "Suma totală în portmoneu este: " + suma
**sf_algoritm**

# TAD MultiDicționar

- Să considerăm problema următoare: pentru un string (șir de caractere), vrem pentru fiecare literă, să reținem poziția pe care litera respectivă se găsește.
- De exemplu, pentru textul "structuri de date" vrem să avem: s - 0, t - 1, r - 2, u - 3, c - 4, t - 5, u - 6, r - 7, etc.
- Ce caracteristici ar trebui să aibă containerul în care reținem datele?

# TAD MultiDicționar

- Să considerăm problema următoare: pentru un string (șir de caractere), vrem pentru fiecare literă, să reținem poziția pe care litera respectivă se găsește.
- De exemplu, pentru textul "structuri de date" vrem să avem: s - 0, t - 1, r - 2, u - 3, c - 4, t - 5, u - 6, r - 7, etc.
- Ce caracteristici ar trebui să aibă containerul în care reținem datele?
  - Ar trebui să avem perechi cheie - valoare (literă - poziție)
  - O cheie ar trebui să aibă mai multe valori asociate (mai multe poziții) => cheile nu mai sunt unice
  - Nu conteaza ordinea în care reținem elementele

# TAD MultiDicționar II

- Containerul care conține perechi cheie-valoare, dar unde o cheie poate să aibă mai multe valori asociate, se numește **MultiDicționar** (în engleză **MultiMap**).
- Ordinea elementelor într-un MultiDicționar nu contează (nu sunt poziții).

# TAD MultiDicționar - Interfață I

- MultiDicționarul are operații similare cu Dicționarul, dar anumite operații au parametrii/valori returnate diferite din cauza faptului că o cheie poate să aibă mai multe valori.

# TAD MultiDicționar - Interfață II

- creeaza()
  - **Descriere:** creează un multidicționar nou, vid
  - **Pre:** Adevărat
  - **Post:** un MultiDicționar vid este creat.

# TAD MultiDicționar - Interfață III

- md.adauga(c, v)
  - **Descriere:** adaugă o pereche cheie - valoare în multidicționar
  - **Pre:** *md* este un MultiDicționar, *c* este un TElement, *v* este un TElement
  - **Post:** perechea *(c, v)* a fost adăugată în multidicționarul *md* (indiferent dacă cheia mai avea alte valori).

# TAD MultiDicționar - Interfață IV

- md.sterge(c, v)
  - **Descriere:** Șterge o pereche cheie, valoare din multidicționar
  - **Pre:** *md* este un MultiDicționar, c este un TElement, *v* este un TElement
  - **Post:** din multidicționarul *md* se șterge perechea (c, v).

# TAD MultiDicționar - Interfață V

- md.cauta(c)
  - **Descriere:** Returnează o listă cu toate valorile asociate unei chei.
  - **Pre:** *md* este MultiDicționar, *c* este un TElement
  - **Post:** cauta returnează *l*, l este lista tuturor valorilor asociate cu cheia *c*. Dacă *c* nu apare în *md*, funcția returnează o listă vidă.

# TAD MultiDicționar - Interfață VI

- md.dim ()
  - **Descriere:** Returnează numărul de perechi din multidicționar
  - **Pre:** *md* este un MultiDicționar
  - **Post:** dim returnează numărul de perechi din *md*

# TAD MultiDicționar - Interfață VII

- md.iterator()
  - **Descriere:** returnează un iterator pentru un multidicționar
  - **Pre:** *md* este un MultiDicționar
  - **Post:** iterator returnează un iterator pentru *md*

- Iteratorul pentru un multidicționar funcționează la fel ca iteratorul pentru alte containere. Operația *element* însă returnează o pereche (cheie, valoare) - dacă există mai multe valori asociate cu o cheie, atunci mai multe apeluri consecutive la *element* vor returna perechi cu aceeași cheie (și valori diferite).

# TAD MultiDicționar - Interfață VIII

- Alte operații posibile pentru un MultiDicționar:

# TAD MultiDicționar - Interfață IX

- md.chei ()
  - **Descriere:** returnează mulțimea cheilor pentru un multidicționar
  - **Pre:** *md* este un MultiDicționar
  - **Post:** chei returnează *m*, *m* este o Mulțime, *m* conține toate cheile din *md*

# TAD MultiDicționar - Interfață X

- md.valori ()
  - **Descriere:** returnează colecția valorilor pentru un multidicționar
  - **Pre:** *md* este un MultiDicționar
  - **Post:** valori returnează *c*, *c* este o Colecție, *c* conține toate valorile din *md*

# MultiDicționar în Java

- În Java nu există containerul MultiDicționar implementat.
- Dacă avem de rezolvat o problemă unde avem nevoie de un MultiDicționar, putem folosi un Dicționar, în care valoarea asociată unei chei este o listă (de valori).

# MultiDicționar - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind un multiDicționar?

# MultiDicționar - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind un multiDicționar?

- Implementarea făcută pentru dicționar se potrivește și pentru multiDicționar (orice dicționar este și un multiDicționar, unul în care fiecare cheie are o singură valoare).

# MultiDicționar - exemplu de utilizare

- Oare am putea rezolva problema cu portmoneul și bancnote (varianta inițială, fără seria bancnotelor) folosind un multiDicționar?
- Implementarea făcută pentru dicționar se potrivește și pentru multiDicționar (orice dicționar este și un multiDicționar, unul în care fiecare cheie are o singură valoare).
- Dacă trecem la varianta cu păstrarea seriei bancnotelor, putem folosi un multiDicționar în care cheia este valoarea bancnotei, iar valoarea este seria bancnotei.