Noţiuni introductive

- **informaţii** → **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, inteligenţa artificială, etc) se pune problema (S)+(A)+(M)
- reprezentarea obiectelor din lumea reală în aplicaţii software necesită
  1. obiecte din lumea reală $\xrightarrow{\text{MODELARE}}$ entităţi matematice
  2. mulţimea de operaţii definite pe entităţile de la 1.
  3. maniera în care entităţile de la 1. sunt **reprezentate** şi **stocate** în memoria calculatorului
  4. algoritmii utilizaţi pentru a efectua operaţiile de la 2.

  (1)+(2) → **TAD**

  (3)+(4) → **SD**

Tipuri Abstracte de Date

"În dezvoltarea înţelegerii fenomenelor complexe, cel mai puternic mecanism disponibil omului este **abstractizarea**. Abstractizarea derivă din recunoaşterea similarităţilor dintre anumite obiecte, situaţii sau procese din lumea reală precum şi din decizia de a concentra aceste similarităţi şi de a ignora, pentru moment, diferenţele dintre ele."

- C.A.R. Hoare

- **Tip de date**
  - *domeniu*
  - *operaţii*
- Un Tip Abstract de Date (TAD) este un tip de date cu următoarele proprietăţi:

1. *specificarea obiectelor* din domeniu este independentă de reprezentarea lor;
2. *specificarea operaţiilor* este independentă de implementarea lor.

- Mulţimea operaţiilor Tipului Abstract de Date definesc interfaţa (contractul) său.
- un TAD este o entitate specificată matematic, care defineşte o mulţime – a instanţelor sale, având o *interfaţă* specifică – o colecţie de *signaturi* ale operaţiilor care pot fi aplicate pe o instanţă a tipului de dată, pentru fiecare operaţie asigurându-se o specificaţie clară.
- Domeniul unui tip (abstract) de dată 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* operaţiilor(date, rezultate, precondiţii, postcondiţii).
- *"De ce abstractizare?"*
  - Deoarece se impune specificarea operaţiilor structurilor de date şi 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 Pyhton, Java).
  - TAD de tip *container*: *vector* (**array**), *colecţie* (**collection**), *mulţime* (**set**), *listă* (**list**), *dicţionar* (**dictionary, map**), etc.
    - *"De ce să încercăm să creem propriile noastre implementări pentru diferite TAD dacă unele limbaje ne oferă implementari ale TAD pe care ni-l dorim?"*
      1. pentru a învăţa modul de utilizare şi creare de implementări de Tipuri Abstracte de Date.
      2. pentru a învăţa comportamentul unora dintre cele mai utilizate Tipuri Abstracte de Date.

Interfaţa unui TAD

Tipurile de operaţii din interfaţa unui TAD sunt următoarele:

- - operaţii care să permită crearea elementelor de acel tip (*constructori*);
- - operaţie care să permită distrugerea elementelor de acel tip (*destructor*);
- - operaţii de accesare a componentelor instanţelor;
- - operaţii specifice de manipulare a instanţelor;
- - operaţii de verificare ale unor proprietăţi ale instanţelor;
- - operaţii specifice tipului de date.

Folosirea abstractizării şi încapsulării datelor în proiectarea programelor

1. **Încapsularea datelor** (ascunderea informaţiei) este definită ca fiind ascunderea detaliilor de implementare ale unui obiect faţă de lumea exterioară (aplicaţiile care îl folosesc).

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

În procesul de proiectare a programelor, folosirea abstractizării şi 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 specificaţiile lor.

- **Testarea** şi **depanarea** se poate face mai uşor, deoarece fiecare dintre tipurile A, B, C, ... pot fi testate şi verificate separat.

- **Reutilizarea** - Extragerea unei structuri de date dintr-o aplicaţie şi folosirea acesteia în altă aplicaţie va fi deosebit de facilă în cazul în care pentru tipul respectiv de date am folosit încapsularea datelor.

- **Schimbarea reprezentării** unui tip de dată - se poate schimba reprezentarea unui tip de date fără a fi afectate programele care folosesc acel tip de date cu condiţia ca operaţiile tipului să nu fie modificate (interfaţa să rămână aceeaşi).

Una din cerinţele esenţiale ale programării prin abstractizarea şi încapsularea datelor este să nu se expună în exterior (într-o aplicaţie) reprezentarea unui Tip Abstract de Date.

Structuri de Date

- Domeniul *Structurilor de Date* (SD) se ocupă cu stocarea şi accesarea datelor.
- În rezolvarea de probleme se manipulează informaţii. În organizarea datelor, conform algoritmului de rezolvare, informaţiile 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 şi definiţie (elementele ei constitutive şi legăturile dintre ele).
  2. **Fizic**, ca mod de memorare (stocare).

     (a) Structuri de date **statice**. O structură **statică** ocupă în memoria calculatorului o zonă de dimensiune constantă, în care elementele ocupă tot timpul execuţiei acelaşi loc. Exemple de structuri de date statice sunt: articolele, tablourile.

     (b) Structuri de date **semistatice**. O structură **semistatică** ocupă în memoria calculatorului o zonă de dimensiune constantă, dar elementele ocupă loc variabil în timpul execuţiei programului. Exemple de structuri de date semistatice sunt tabele de dispersie.

     (c) Structuri de date **dinamice**. O structură **dinamică** ocupă în memoria calculatorului o zonă care se alocă dinamic, în timpul execuţiei programului, pe măsura nevoilor de prelucrare, fără a avea o dimensiune constantă. Exemple de structuri de date dinamice sunt: liste înlănţuite, arbori.

- O structură de date logică ar putea fi implementată atât ca structură statică, semistatică sau dinamică. De exemplu, lista înlănţuită, 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 şi 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 complexităţii operaţiilor este importantă.

  – Cum putem proiecta SD noi?

TAD/Container	SD
Vector Dinamic	**tablou**
Matrice	**lista înlănţuită**
Colecţie	- simplu
Mulţime	- dublu
Dicţionare	- alocare dinamică
Dicţionar, Dicţionar ordonat,	- înlănţuiri pe tablou
Multidicţionar, Multidicţionar ordonat	**tabela de dispersie**
Lista	- liste independente
Lista ordonată	- liste întrepătrunse
Coada	- adresare deschisă
Stiva	**ansamblu**
Coada cu priorităţi	**arbore binar de căutare**
Arbore (binar)	- arbore echilibrat (AVL)