1. Structuri de date fundamentale

1.1. Introducere

- Sistemele de calcul moderne sunt dispozitive care au fost concepute cu scopul de a facilita şi accelera **calcule complicate**, mari consumatoare de timp.
    - Din acest motiv, în majoritatea aplicaţiilor, **viteza, frecvenţa de lucru**, **capacitatea de a memora**, **posibilitatea** de **a avea acces la cantităţi mari de informaţii** şi **capabilităţile de conectivitate**, au un rol hotărâtor şi sunt considerate caracteristici primordiale ale calculatoarelor.
    - **Abilităţile de calcul**, în cele mai multe cazuri este irelevantă în manieră directă.
- Cantitatea mare de informaţii pe care o prelucrează un sistem de calcul reprezintă de regulă, o **abstractizare** a lumii înconjurătoare, respectiv a unei părţi a lumii reale.
    - Informaţia furnizată sistemului de calcul constă dintr-o mulţime de **date selectate** de către utilizator care **modelează** lumea reală.
    - **Datele selectate** se referă la **mulţimea de date** care este considerată cea mai reprezentativă pentru **problema tratată** şi despre care se presupune că din ea pot fi obţinute (deduse) **rezultatele dorite**.
- În acest context, **datele** reprezintă o **abstractizare a realităţii.**
- Ce este o **abstractizare**?
    - O **abstractizare** este de fapt o **simplificare** a faptelor.
    - În consecinţă, anumite **proprietăţi** şi **caracteristici** ale obiectelor reale, sunt **ignorate** întrucât pot fi considerate **periferice şi nerelevante** pentru problema particulară tratată.
    - Dacă nu s-ar tine cont de abstractizare şi s-ar lua în considerare toate caracteristicile şi proprietăţile problemei tratate, aceasta ar deveni extrem de complexă, practic inabordabilă.
- În rezolvarea unei probleme cu ajutorul unui sistem de calcul un rol esenţial revine alegerii unei **abstractizări convenabile** a realităţii.
    - Acest lucru se poate realiza în doi paşi:

        - (1) Definirea unui **set reprezentativ de date** care **modelează** (reprezintă) situaţia reală.

        - (2) Stabilirea **reprezentării abstractizării**.

    - De cele mai multe ori cei doi paşi se întrepătrund.

- **Alegerea abstractizării** este determinată de regulă de **natura problemei** de rezolvat.

    - Aceasta activitate este determinată la rândul ei, de **uneltele** care vor fi utilizate în rezolvarea problemei, spre exemplu de facilităţile oferite de un anume sistem de calcul.

- **Alegerea reprezentării** este de multe ori o activitate dificilă întrucât **nu** este determinată în mod unic de facilităţile disponibile.

    - Se cunoaşte că reprezentarea internă a informaţiei într-un sistem de calcul se realizează cu ajutorul **cifrelor binare** (biţi).

    - Această reprezentare care este foarte potrivită pentru **circuitele electronice**, **nu** este potrivită pentru **fiinţa umană** din cauza numărului prea mare de cifre implicate.

- Din acest motiv, alegerea reprezentării datelor se poate realiza la **diferite niveluri de abstractizare**, funcţie de obiectivul urmărit şi de limbajul de programare utilizat.

    - **Limbajul de programare** reprezintă de fapt **suportul** sau **mediul** care suportă abstractizarea.

- Astfel, un **limbaj de programare**:

    - Reprezintă un **calculator abstract,** capabil să înţeleagă **fraze** construite cu ajutorul **termenilor** definiţi în cadrul acestui limbaj de programare.

    - **Termenii** definiţi în cadrul limbajului de programare, pot să încorporeze un anumit **nivel de abstractizare** al entităţilor efectiv utilizate (definite) de către maşina reală.

- Utilizând un astfel de limbaj, programatorul va fi eliberat spre exemplu de chestiuni legate de reprezentarea numerelor dacă acestea constituie entităţi elementare în descrierea limbajului.

- Utilizarea unui **limbaj de programare** care oferă un **set fundamental de abstractizări**, valabil pentru marea majoritate a problemelor de prelucrare a datelor, se reflectă în special **în domeniul de reliabilitate** al programelor care rezultă.

    - Este mult mai uşor să se conceapă un program bazat pe obiecte familiare (numere, mulţimi, secvenţe) decât unul bazat pe structuri de biţi, cuvinte şi salturi.

- Desigur, oricare sistem de calcul reprezintă în ultimă instanţă datele ca şi un masiv de informaţii binare.

    - Acest fapt este însă **transparent** pentru programator.

- Programatorul poate opera cu **noţiuni abstracte**, mai uşor de manipulat întrucât dispune de **compilatoare** sau **translatoare specializate** care preiau sarcina **transformării** noţiunilor abstracte în termeni specifici sistemului de calcul ţintă.

- Cu cât **nivelul de abstractizare** este mai strâns legat de un anumit sistem de calcul:
    - (1) Cu atât este mai **simplu** pentru dezvoltatorul de aplicaţii să aleagă o **reprezentare** mai eficientă a datelor.
    - (2) Cu atât mai **redusă** este posibilitatea ca reprezentarea aleasă să satisfacă o **clasă mai largă** de aplicaţii convenabile.
- Aceste elemente precizează **limitele nivelului de abstractizare** abordat pentru un sistem de calcul real respectiv pentru un limbaj de programare.
- În categoria limbajelor de programare avute în vedere în acest curs se includ limbajele ***C***, respectiv ***C++*** .
    - Aceste limbaje acoperă în mod fericit o plajă largă situată între:
        - (1) **Limbajele orientate spre maşină** sau limbaje dependente de maşină care lasă deschise problemele reprezentărilor.
        - (2) **Limbajele cu un înalt nivel de abstractizare** bazate pe obiecte care rezolvă automat problemele de reprezentare.
    - În acest mod utilizatorul poate aborda nivelul de abstractizare convenabil din punctul de vedere al realizării unei anumite aplicaţii.
- Cu alte cuvinte, **procesul de abstractizare şi de reprezentare a datelor** utilizate în activitatea de programare este suportat de către **limbajele de programare**.
- Cum se realizează acest lucru?

1.2. Tipuri de date. Tipuri de date abstracte

1.2.1. Conceptul de tip de date

- În procesul de **prelucrare a datelor** se face o distincţie clară între:
    - (1) Numerele reale
    - (2) Numerele complexe
    - (3) Valorile logice
    - (4) Variabilele care reprezintă valori individuale, mulţimi de valori, mulţimi de mulţimi sau funcţii, mulţimi de funcţii, etc.
- În acest sens se statuează **principiul** conform căruia fiecare constantă, variabilă, expresie sau funcţie este încadrată într-un anumit **tip**.
- Un **tip** este în mod esenţial caracterizat prin:
    - (1) **Mulţimea valorilor** căreia îi aparţine o constantă a tipului în cauză, respectiv mulţimea valorilor pe care le poate asuma o variabilă, o expresie sau care pot fi generate de o funcţie încadrată în acel tip.

- (2) Un anumit **grad de structurare** (organizare) a informaţiei;

- (3) Un **set de operatori** specifici.

- În textele matematice tipul variabilelor este în general deductibil din maniera lor de **prezentare**, respectiv din forma tipografică a caracterelor utilizate (fonturi), fără a lua în considerare contextul.

    - Această modalitate este însă dificil de utilizat în textele sursă ale programelor pentru calculatoare unde în general se utilizează o gamă relativ redusă de caractere tipografice.

- Din acest motiv, în practica curentă, **tipul asociat** unei constante, variabile sau funcţii este precizat printr-o **declaraţie explicită de tip** referitoare la constanta, variabila sau funcţia respectivă, declaraţie care precede textual utilizarea constantei, variabilei sau funcţiei în cauză.

    - Această metodă simplifică activitatea compilatorului permiţând verificarea automată a legimităţii utilizării entităţilor asociate tipului respectiv.

- **Caracteristicile** conceptului de **tip de date** sunt următoarele:

    - (1) Un tip de date determină **mulţimea valorilor** căreia îi aparţine o constantă, sau pe care le poate asuma o variabilă sau o expresie, sau care pot fi generate de un operator sau o funcţie.

    - (2) **Tipul** unei valori precizate de o constantă, variabilă sau expresie poate fi **dedus** din **forma** sau din **declaraţia** sa, fără a fi necesară execuţia unor procese de calcul.

    - (3) Fiecare **operator** sau **funcţie** acceptă **argumente** de un tip precizat şi conduce la un **rezultat** de un tip precizat.

        - Dacă un operator admite argumente de diferite tipuri, atunci tipul rezultatului poate fi determinat din regulile specifice limbajului.

        - De exemplu - adunarea numerelor întregi cu numere reale, care conduce de regulă la valori reale.

    - (4) Un tip presupune un anumit **nivel de structurare** (organizare) a informaţiei.

- Drept urmare, respectând aceste reguli, un **compilator** poate verifica compatibilitatea şi legalitatea anumitor construcţii de limbaj, în **faza de compilare**, fără a fi necesară execuţia efectivă a programului.

    - Acest tip de **redundanţă controlată** a textului programului este extrem de folositor în dezvoltarea de programe şi este considerat ca un **mare avantaj** al **limbajelor de nivel superior** asupra **limbajelor de asamblare**.

- Din punctul de vedere al sistemului de calcul, memoria este o **masă omogenă de biţi** fără vreo structură aparentă.

    - Ori tocmai **structurile abstracte de date** sunt acelea care permit recunoaşterea, interpretarea şi prelucrarea configuraţiilor monotone de cifre binare prezente în memoria uui sistem de calcul.

1.2.2. Tipuri de date abstracte

- Definirea noţiunii de **tip de date abstract** (**TDA**):
    - Un **TDA** se defineşte ca un **model matematic** căruia i se asociază o **colecţie de operatori specifici**.
- Vom realiza o paralelă cu conceptul de **procedură**.
    - (1) Procedura **generalizează** noţiunea de **operator**.
        - De regulă, un programator care utilizează un limbaj de programare este constrâns la utilizarea exclusivă a operatorilor definiţi în cadrul limbajului de programare ("built-in" operators).
        - Folosind procedurile sau funcţiile, programatorul este liber să-şi **definească proprii săi operatori**, pe care ulterior să-i aplice asupra unor operanzi care nu e necesar să aparţină tipurilor de bază (primitive) ale limbajului utilizat.
        - Cu alte cuvinte, utilizând **funcţiile** sau **procedurile**, operatorul poate **generaliza** anumiţi **operatori specifici** aplicaţiilor proprii.
            - Un exemplu de funcţie utilizată în această manieră este rutina de înmulţire a două matrici.
    - (2) Procedurile **încapsulează** anumite părţi ale unui algoritm prin "**localizare**".
        - Aceasta înseamnă amplasarea într-o singură secţiune a programului a tuturor instrucţiunilor relevante.
- Într-o manieră similară:
    - (1) **Un TDA generalizează** tipurile de date primitive (întreg, real, etc.), după cum procedurile sunt generalizări ale operaţiilor primitive (+, -, etc.).
    - (2) **Un TDA încapsulează** conceptual un tip de date în sensul că din punct de vedere logic şi fizic, definirea tipului şi toate operaţiile referitoare la el sunt localizate într-o singură secţiune (zonă) a programului.
        - Dacă apare necesitatea **modificării implementării** TDA-ului:
            - (2.1) Programatorul ştie cu certitudine locul (unic) în care trebuiesc efectuate modificările.
            - (2.2) Poate fi sigur că modificarea secţiunii respective **nu** produce modificări nedorite în restul codului programului, dacă nu se modifică formatul operatorilor TDA-ului respectiv.
    - (3) În plus, în afara secţiunii în care sunt definiţi operatorii, **TDA**-ul în cauză poate fi tratat ca un **tip primitiv**.
        - În acest caz un utilizator tratează un TDA astfel definit, în termenii operatorilor asociaţi nefiind în nici un fel preocupat de implementarea acestora.

- O problemă care poate să apară este cea legată de faptul că anumite operaţii pot să se refere la mai multe TDA-uri, caz în care accesarea acestor operaţii trebuie realizată în mod specific în funcţie de definirea fiecărui TDA.

- **Dezavantajul** major al folosirii TDA-urilor rezidă în necesitatea respectării riguroase a **disciplinei de utilizare**.

  - Cu alte cuvinte **toate referirile** la datele încapsulate într-un TDA trebuiesc realizate prin **operatorii specifici**.

  - Această cerinţă **nu** este verificabilă la nivelul **compilatorului** şi trebuie acceptată ca şi o **constrângere impusă** a disciplinei de programare.

- **Exemplul 1.2.2.**
- Se consideră un tip de date abstract **Listă.**
- Nodurile listei aparţin unui tip precizat **TipNod**.
- Fie:
  - L o instanţă a TDA-ului (L: Lista),
  - v o variabilă nod (v: TipNod),
  - c o referinţă la un nod al listei (c: TipReferintaNodLista).
- Un posibil **set de operatori** pentru **TDA Listă** este următorul:

  1**. *ListăVidă****(L:Lista);**- operator care iniţializează pe L ca listă vidă;

  2. ***c:= Primul****(L:Lista);* - operator care returnează valoarea indicatorului la primul nod al listei, sau indicatorul vid în cazul unei liste goale;

  3. ***c:= Următor****(L:Lista);* - operator care returnează valoarea indicatorului următorului nod al listei, respectiv indicatorul vid dacă un astfel de nod nu există;

  4. ***Inserează****(v:TipNod, L:Lista);* - operator care inserează nodul v în lista L după nodul curent.

  5. ***Furnizează****(v:TipNod, L:Lista**);* - operator care furnizează conţinutul nodului curent al listei.

- Câteva **observaţii** referitoare la avantajele definirii unui astfel de TDA:

  - (1) Odată definit şi implementat TDA Listă, **toate prelucrările de liste** se pot realiza în termenii operatorilor definiţi asupra acestuia, declarând un număr corespunzător de instanţe ale TDA-ului, respectiv câte o instanţă pentru fiecare listă.

  - (2) Indiferent de maniera concretă de implementare a TDA Listă (tablouri, pointeri, cursori), din punctul de vedere al utilizatorului, **forma operatorilor** rămâne cea definită iniţial.

- Chiar în accepţiunea **modificării pe parcurs a implementării** spre exemplu din motive de eficienţă, programele utilizator rămân nemodificate, singurele modificări fiind suportate de secţiunea care include TDA-ul.

- (3) Nu există nici o **limitare** referitoare la numărul de operaţii care pot fi aplicate **instanţelor** unui model matematic precizat.

    - Se face însă sublinierea că fiecare set de operatori defineşte şi se referă la o instanţă distinctă a unui TDA.

- (4) Eficienţa, siguranţa şi corectitudinea utilizării TDA-ului Lista este garantată numai în situaţia în care utilizatorul realizează **accesele** la listele definite **numai** prin **operatorii asociaţi** tipului respectiv, adică respectă strict restricţiile impuse.

1.2.2.1. Definirea unui tip de date abstract (TDA)

- După cum s-a precizat, un tip de date abstract (TDA) este conceput ca şi **un model matematic** căruia i se asociază o **colecţie de operatori** definiţi pe acest model.

- În consecinţă, **definirea** unui tip de date presupune:

    - (1) Precizarea **modelului matematic**.

    - (2) Definirea **operatorilor** asociaţi.

- Limbajele de programare includ mai multe **metode de definire** a tipurilor de date.

- În toate situaţiile se porneşte de la un set de tipuri denumite **tipuri primitive.**

    - **Tipurile primitive** joacă rolul de **cărămizi** ale dezvoltării.

- În cadrul acestor **tipuri primitive** se disting:

        - (1) **Tipurile standard** sau **predefinite.**

        - (2) **Tipurile primitive definite de utilizator**.

- (1) **Tipurile standard** sau **predefinite** includ de regulă numerele şi valorile logice.

    - Ele se bazează pe **modele matematice consacrate** (mulţimile de numere întregi, reale, logice, etc.) şi implementează operaţiile specifice definite pe aceste categorii de numere.

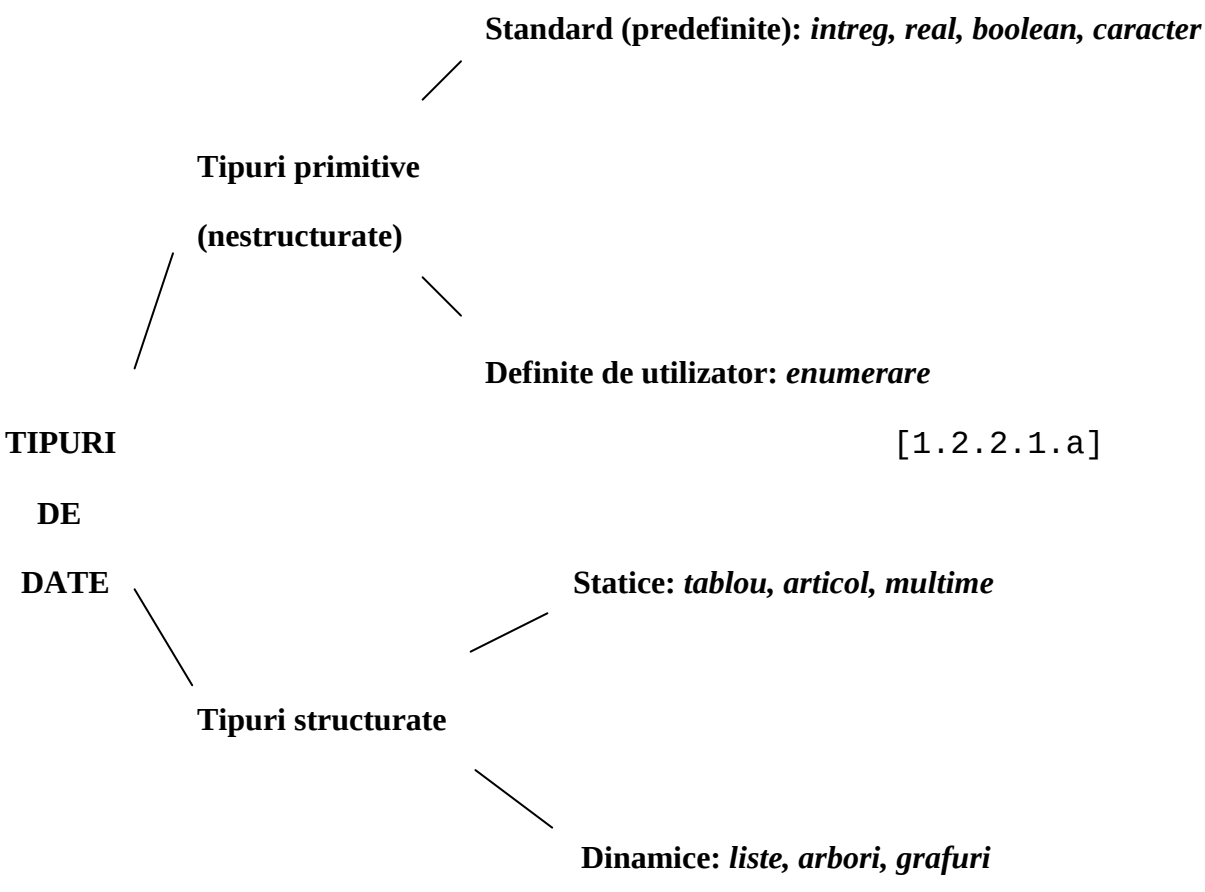
    - Dacă între valorile individuale ale tipului există o **relaţie de ordonare**, atunci tipul se numeşte **ordonat** sau **scalar**.

- (2) **Tipuri primitive definite de utilizator**.

    - O metodă larg utilizată de construcţie a unor tipuri primitive de către utilizatori este aceea a **enumerării** valorilor constitutive ale tipului.

    - Spre exemplu, într-un program care prelucrează figuri geometrice, poate fi introdus un tip primitiv numit *TipFigură* ale cărui valori pot fi precizate de identificatorii *dreptunghi, pătrat, elipsă, cerc*.

- În multe cazuri, **noile tipuri** se definesc în termenii unor **tipuri anterior definite**.
    - Valorile unor astfel de tipuri, sunt **conglomerate** de valori componente ale unuia sau mai multor tipuri constitutive definite anterior motiv pentru care se numesc **tipuri structurate**.
- **Tipurile structurate**. Sunt de regulă conglomerate de valori de componente. Dacă există **un singur** tip constitutiv, acesta se numeşte **tip de bază.**
    - Numărul valorilor distincte aparţinătoare unui tip T se numeşte **cardinalitatea** lui T.
- Pentru definirea tipurilor structurate, se utilizează **metode de structurare**.
    - Poate fi construită o întreagă ierarhie oricât de complexă de astfel de structuri.
- **Metodele de structurare** pot fi (secventa [12.2.1.a]):
    - **Statice**: tabloul, articolul, (mulţimea), secvenţa (fişierul).
    - **Dinamice**: listele, arborii, grafurile.



- Pentru a utiliza un tip de date definit, trebuiesc declarate **variabile** încadrate în tipul în cauză, proces care se numeşte **instanţiere**.

- Prelucrarea acestor variabile se realizează cu ajutorul operatorilor asociaţi tipului.

- Există mai multe **feluri de operatori**:

    - (1) **Omogeni** - pot avea drept operanzi numai instanţe ale TDA-ului în cauză.

    - (2) **Micşti** - pot avea drept operanzi instanţe ale altor tipuri, de regulă instanţe ale unor alte TDA-uri compatibile.

    - (3) **Interni** - rezultatul aplicării unui astfel de operator poate fi doar o altă instanţă a TDA-ului respectiv.

    - (4) **Externi** - rezultatul aplicării unui astfel de operator poate aparţine unui alt tip.

- Există mai multe **categorii de operatori**:

    - (1) **Primitivi** sau **operatori atomici** - aceşti operatori sunt de regulă implementaţi prin hardware şi reflectă operaţii specifice cablate ale sistemului de calcul.

    - (2) **Definiţi prin limbaj** - aceşti operatori includ de regulă metodele de structurare şi metodele de acces la componente.

    - (3) **Definiţi şi implementaţi de utilizator**.

- **Operatori de bază** sunt:

    - **Comparaţia** şi **atribuirea** - sunt operatori fundamentali definiţi pentru toate tipurile de date. Execuţia lor poate însă să presupună un volum cu atât mai sporit de calcul, cu cât gradul de structurare este mai înalt.

    - **Operatorii de transfer**- sunt operatorii care organizează (constituie) tipurile de date în alte tipuri (generează structuri). Din acest motiv ei sunt importanţi în conexiune cu tipurile structurate.

    - **Operatorii constructori** - generează valorile structurate pornind de la valorile lor componente.

    - **Operatorii selectori** - realizează extragerea valorilor componente dintr-o structură.

- **Constructorii şi selectorii** sunt de fapt operatori de transfer care organizează (constituie) tipurile constitutive în tipuri structurate şi invers.

    - Fiecare metodă de structurare dispune de perechea sa particulară de operatori constructor-selector care diferă în mod clar prin maniera de notare.

- **Operatorii de transfer** pot fi:

    - **Impliciţi** - caz în care sunt implementaţi direct în limbaj (spre exemplu accesul la un câmp al unui articol sau accesul la un element al unui tablou);

- **Expliciţi** - în cazul în care sunt sintetizaţi de programator prin elemente de limbaj (spre exemplu accesul la nodurile unei liste).

- În concluzie:
  - (1) Un tip este de fapt o manieră de structurare a informaţiei.
  - (2) Pentru tipurile nestructurate este mai relevantă mulţimea constantelor tipului şi mai puţin gradul de structurare.
  - (3) Pe măsură ce tipul devine mai complex, gradul de structurare devine tot mai relevant.

1.2.2.2. Implementarea unui TDA

- **Implementarea** unui TDA este de fapt o translatare, adică o exprimare a sa în termenii unui limbaj de programare concret.
- **Implementarea** unui TDA presupune:
  - (1) Precizarea structurii propriu-zise (**modelul matematic**) în termenii unui limbaj de programare.
  - (2) Definirea **procedurilor** care implementează operatorii specifici.
- Desigur, activitatea de implementare a unui tip se realizează **numai** în **cazul tipurilor de date definite de utilizator**, în restul cazurilor elementele specifice de implementare fiind de regulă incluse în limbajul de programare utilizat.
- Un **tip structurat** se construieşte pornind de la tipurile de date de bază care sunt definite în cadrul limbajului, utilizând **facilităţile de structurare** disponibile.
  - Definirea procedurilor care implementează operatorii este intim legată de maniera de implementare aleasă pentru materializarea TDA-ului.
- Există posibilitatea abordării **ierarhice** a implementării TDA-urilor, respectiv implementarea unui TDA în termenii altor TDA-uri deja existente
- La modul ideal, programatorii ar dori să-şi redacteze programele în limbaje ale căror tipuri primitive de date să fie cât mai apropiate modelele şi operaţiile TDA-urilor proprii.
  - Din acest punct de vedere, în multe sensuri limbajele obişnuite de programare **nu** sunt suficient de înzestrate în ceea ce priveşte facilităţile de a crea diverse TDA-uri şi nici de definire a operatorilor specifici.

1.2.3. Tip de date abstract. Tip de date. Structură de date

- Care este semnificaţia următorilor termeni care sunt adesea confundaţi?
  - (1) **Tip de date abstract** (TDA)

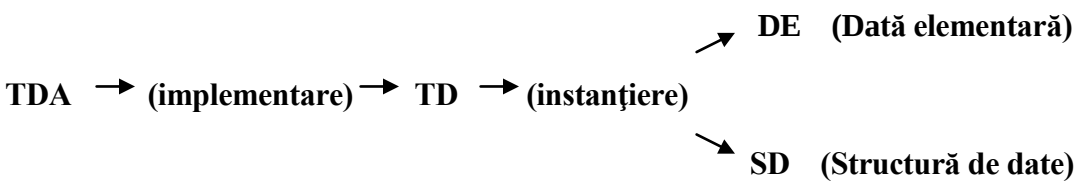
- (2) **Tip de date** (TD) sau simplu **tip**
- (3) **Structură de date** (SD)
- (4) **Dată elementară** (DE)

- (1) După cum s-a precizat, un **tip de date abstract** este un model matematic, împreună cu totalitatea operaţiilor definite pe acest model.
  - La **nivel conceptual**, algoritmii pot fi proiectaţi în termenii unor tipuri de date abstracte (TDA).
  - Pentru a **implementa** însă un astfel de algoritm într-un anumit limbaj de programare, este absolut necesar să se realizeze o **reprezentare** a acestor TDA-uri în termenii tipurilor de date şi ai operatorilor definiţi în limbajul de programare respectiv.

- (2) Un **tip de date** este o **implementare** a unui TDA într-un **limbaj de programare**.
  - Un **tip de date** (TD) **nu** poate fi utilizat ca atare.
  - În consecinţă în cadrul programului se declară **variabile** încadrate în tipul respectiv, variabile care pot fi efectiv prelucrate şi care iau valori în mulţimea valorilor tipului.
  - Acest proces se numeşte **instanţiere** şi el de fapt conduce la rezultate diferite:
    - Instanţierea unui tip **nestructurat** conduce la generarea unei **date elementare** (DE).
    - Instanţierea unui tip **structurat** conduce la generarea unei **structuri de date** (SD).

- (3) O **dată elementară** este o **instanţă** a unui **tip de date nestructurat**.

- (4) O **structură de date** este o **instanţă** a unui **tip de date structurat**.
  - Ca atare legătura dintre aceste noţiuni este materializată de următoarea formulă:



- Se face precizarea că formula de mai sus este valabilă pentru tipurile primitive definite de utilizator şi pentru cele structurate.
  - Prin ***implementare*** se înţelege **definirea tipului.**
  - Prin ***instanţiere,*** se înţelege **declararea unei variabile** asociate tipului.

- În cazul **tipurilor predefinite** faza de implementare lipseşte ea fiind inclusă intrinsec în limbajul de programare, ca atare pentru aceste tipuri este valabilă formula:

TD → (instanţiere) → DE

1.3. Tipuri nestructurate

1.3.1. Tipul enumerare

- În marea majoritate a programelor se utilizează numerele întregi, chiar atunci când nu sunt implicate valori numerice şi când întregii reprezintă de fapt abstractizări ale unor entităţi reale.

- Pentru a evita această situaţie limbajele de programare introduc un nou tip de date primitiv nestructurat precizat prin enumerarea tuturor valorilor sale posibile

- Un astfel de tip se numeşte **Tip enumerare** şi el este definit prin enumerarea tuturor valorilor sale posibile: $c_1, c_2, ..., c_n$.

```
-------------------------------------------------------------
// Definirea tipului enumerare - Variante

enum tipEnumerare {c1,c2,c3,...,cn} variabila;

typedef enum {c1,c2,c3,...,cn} numeTip;          [1.3.1.a]
-------------------------------------------------------------
```

- Tipul eunumerare este un tip **ordonat**, valorile sale considerându-se ordonate crescător în ordinea în care au fost definite în baza convenţiei:

$$(c_i < c_j) \Leftrightarrow (i < j)$$

- Cardinalitatea tipului este **card**(`tipEnumerare`) = n.

- Tipul enumerare este un **tip nestructurat definit de utilizator**

    - În consecinţă utilizarea sa presupune atât faza de **implementare** (declarare a tipului) cât şi faza de **instanţiere** (declarare de variabile aparţinătoare tipului).

```
-------------------------------------------------------------

// Exemplu de implementare a tipului enumerare

typedef enum {dreptunghi,patrat,cerc,elipsa} tipFigura;

typedef enum {alb, rosu, portocaliu, galben, verde, albastru,
maro, negru} tipCuloare;

typedef enum {adevarat,fals} tipBoolean;      /*1.3.1.b*/

typedef enum {luni,marti,miercuri,joi,vineri,sambata,duminica}
tipZiSaptamina;
```

```
typedef enum{soldat,caporal, sergent, sublocotenent,
locotenent, capitan, maior, colonel,general} tipGrad;

typedef enum {constanta,tip,variabila,procedura,functie}
tipObiect;

typedef enum {secventa,tablou,articol,multime} tipStructura;
```

- La **definirea** unui astfel de tip se introduce **nu** numai un nou **identificator de tip**, dar se definesc şi **identificatorii** care precizează valorile noului tip (de fapt se precizează explicit domeniul valorilor tipului).

- Aceşti identificatori pot fi utilizaţi ca şi constante în cadrul programului, sporind în acest fel considerabil gradul de lizibilitate şi de înţelegere al textului sursă.

```
/* Exemplu de instanţiere a tipului enumerare */

tipCuloare c;                                    /*1.3.1.c */
tipZiSaptamina z;
tipGrad g;
tipBoolean b;

c = alb;            /*c:= 0;*/
z = miercuri;       /*z:= 2;*/
g = capitan;        /*g:= 5;*/
b = fals;           /*b:= 1;*/
```

  - Astfel, referitor la secvenţa [1.3.1.b] se pot introduce spre exemplu variabilele `c,z,g,b` [1.3.1.c].

  - În mod evident, această reprezentare la nivel de program este mult mai intuitivă decât spre exemplu:

    ```
    c:= 0;  z:= 2;  g:= 5 ;  b:= 1;
    ```

  - Aceasta reprezentare se bazează pe presupunerea că `c,z,g,b` sunt de tip întreg şi că în calitate de constante le sunt asociate numere întregi în ordinea enumerării lor.

  - Drept urmare un compilator poate verifica uşor utilizarea unor operatori nespecifici pentru astfel de tipuri nenumerice, ca de exemplu:

    `c= c+1;` sau `g= white;` vor fi semnalate ca erori.

- Tipul enumerare include implicit ca şi operatori **comparaţia** şi **atribuirea**.

- Deoarece, tipul enumerare este un **tip ordonat**, în setul său de operatori pot fi definiţi operatorii care generează succesorul respectiv predecesorul unui argument precizat.

  - Aceştia sunt operatori : **`SUCC(x)`** şi **`PRED(x)`** sau variante ale lor.

1.3.2. Tipuri standard predefinite

- **Tipurile standard predefinite** sunt acele tipuri care sunt disponibile în marea majoritate a sistemelor de calcul ca şi caracteristici hardware.

- **Tipurile standard predefinite** includ **numerele întregi**, **valorile logice**, **caracterele** şi **numerele fracţionare** adică: INTEGER, BOOLEAN, CHAR, REAL.

- (1) **Tipul întreg** implementează o submulţime a numerelor întregi.

    - Dimensiunea întregilor (numărul maxim de cifre) variază de la un sistem de calcul la altul şi depinde de caracteristicile hardware ale sistemului.

    - Se presupune că toate operaţiile efectuate asupra acestui tip conduc exclusiv la valori exacte şi se conformează regulilor obişnuite ale aritmeticii.

    - Procesul de calcul se întrerupe în cazul obţinerii unui rezultat în afara setului reprezentabil (depăşirea capacităţii registrelor - DCR).

    - Pe mulţimea numerelor întregi în afara operatorilor clasici de **comparare** şi **atribuire**, se definesc şi **operatori standard**:

        - Operatori standard definiţi pe mulţimea întregilor: **adunare** (+), **scădere** (-), **înmulţire** (*), **împărţire întreagă** (**/**) şi **modulo** (%).

$$m - n < (m / n) * n <= m$$

$$(m / n) * n + (m \% n) = m$$

- În secvenţa [1.3.2.1.a] este prezentat în manieră sintetică **TDA Întreg**.

```
-------------------------------------------------------------
TDA Întreg

Modelul matematic: elemente scalare cu valori în mulţimea
    numerelor întregi: {..., -3, -2, -1, 0, 1, 2, 3,....}.

Notaţii:
    i,j,k - întregi;                                  [1.3.2.a]
    inz   - întreg nonzero;
    inn   - întreg nonnegativ;
    e     - valoare întreagă (constantă sau variabilă);
    b     - valoare booleană.

Operaţii:
     AtribuireIntregi(i,e)  - memorează valoarea lui e în
        variabila i (procedură);
     k= AdunareÎntregi(i,j)      (funcţie);
     k= ScădereÎntregi(i,j)      (funcţie);
     k= ÎnmulţireÎntregi(i,j)    (funcţie);
     k= ÎmpărţireÎntregi(i,inz)  (funcţie);
     inn= Modulo(i,inz)  - returnează întregul pozitiv care
        reprezintă restul împărţirii lui i cu inz (funcţie);
     b= EgalZero(i)              (funcţie);
     b= MaiMareCaZero(i)         (funcţie).
-------------------------------------------------------------
```

- (2) **Tipul real** implementează o submulţime reprezentabilă a numerelor reale.

- În timp ce aritmetica numerelor întregi conduce la rezultate exacte, aritmetica valorilor de tip real este aproximativă în limitele erorilor de rotunjire cauzate de efectuarea calculelor cu un număr finit de cifre zecimale.

- Numerele reale sunt reprezentate cu o anumită **precizie** exprimată în număr de cifre zecimale

- Din acest motiv tipurile întreg respectiv real în marea majoritate a limbajelor de programare se tratează separat.

- Operaţiile se notează cu simbolurile consacrate cu excepţia împărţirii numerelor reale care se notează cu (/).

- Operaţiile care conduc la valori care depăşesc domeniul de reprezentabilitate al implementării conduc la erori (DCR).

- Implementările curente ale limbajelor de programare conţin mai multe categorii de tipuri întregi (int, long int, short int, unsigned int, etc.) respectiv mai multe tipuri reale (float, double, long double, etc) care diferă de regulă prin numărul de cifre binare utilizate în reprezentare.

- (3) **Tipul Boolean** are două valori care sunt precizate de către identificatorii **TRUE** (adevărat) şi **FALSE** (fals).

  - În limbajul C aceste valori lipsesc fiind substituite de valori întregi:

    - `1` sau `<>0` semnifică **adevărat.**

    - `0` semnifică **fals**.

  - Operatorii specifici definiţi pentru acest tip sunt operatorii logici: conjuncţie, reuniune şi negaţie.

P	Q	P **&&** Q	P **||** Q	**!** P
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Tabelul 1.3.2.a. Operatori logici

- (4) **Tipul standard caracter** cuprinde o mulţime de caractere afişabile.

  - **Codificarea ASCII** *(American Standard Code for Information Interchage)*

```
-------------------------------------------------------------
Litere mari:    A    B    C   ...   X    Y    Z
```

```
 hexazecimal:  x'41'   x'42'   x'43'        x'58'   x'59'   x'5A'
 zecimal:       65     66     67            88      89      90

Cifre:          0      1      2    ...      9
 hexazecimal:  x'30'   x'31'  x'32'         x'39'
 zecimal:       48     49     50            57

Litere mici:    a      b      c    ...      z
 hexazecimal:  x'61'  x'62'  x'63'        x'7A'
 zecimal:       97     98     99           122

Caracterul blanc:  hexazecimal: x'20'      zecimal: 32
```

- De regulă limbajele de programare definesc tipul primitiv `char`.
- Stabilirea naturii unui caracter:

  `('A' <= x) && (x <= 'Z')` - `x` este literă mare.

  `('a' <= x) && (x <= 'z')` - `x` este literă mică.

  `('0' <= x) && (x <= '9')` - `x` este o cifră.

- În limbajul C tipurile `char` şi `int` sunt echivalente, deşi au interpretări diferite.
- Se pot defini două funcţii standard de transfer între tipurile `char` şi `integer`:
  - `f('5')=5` transformă un caracter în întregul corespunzător.
  - `g(8)='8'` transformă un întreg în caracterul corespunzător.
  - `f` şi `g` sunt funcţii inverse:

    `f(g(i))=i    (0 < i < 9)`

    `g(f(c))=c    ('0' < c < '9')`

```
// Implementarea funcţiilor de transfer  întreg-caracter

char c; int n;
    n=c-'0'       //f(c)-> i                        [1.3.2.b]
    c=n+'0';      //g(i)-> c
```

1.4. Tipuri structurate

1.4.1. Structura tablou. Tipul de date abstract tablou.

- Un tablou este o **structură omogenă** el constând dintr-o mulţime de componente de acelaşi tip numit **tip de bază**.

- Tabloul este o structură de date cu **acces direct** (*random-acces*), deoarece oricare dintre elementele sale sunt direct şi în mod egal accesibile.

- Pentru a preciza o componentă individuală, numelui întregii structuri i se asociază un **indice** care selectează poziţional componenta dorită.

- La definirea unui tablou în principiu se precizează în principiu:

    - (1) **Metoda de structurare**, care este implementată direct de limbaj.

    - (2) **Numele tipului** de date rezultat (`tipTablou`).

    - (3) **Tipul de bază** al tabloului (`tipElement`).

    - (4) **Tipul indice** al tabloului (`tipIndice`).

    - (5) **Dimensiunea tabloului** adică numărul maxim de elemente.

- În limbajul C lucrurile există unele particularităţi:

    - Nu se precizează explicit metoda de structurare. Ea este indicată de prezenţa parantezelor drepte în sintaxa declaraţiei.

    - Tipul indice este în mod obligatoriu tipul întreg (int).

    - Domeniul valorilor indicilor este restrâns la mulţimea întregilor pozitivi: `0, 1,2,3,..., n-1`, unde `n` este numărul de elemente al tabloului.

    - Între parantezele drepte se indică dimensiunea tabloului (numărul total de elemente).

```
-------------------------------------------------------------
/*Definirea unui tablou */
                                                /*1.4.1.a*/
tipElement numeTablou[nrElemente];
-------------------------------------------------------------
/* Definirea unui tip tablou */
                                                /*1.4.1.b*/
typedef tipElement tipTablou[nrElemente];

tipTablou numeTablou;
-------------------------------------------------------------
/* Definirea unui tablou – exemple */

     float tablou[10];                          /*1.4.1.c*/

     char sir[10];
-------------------------------------------------------------
/* Definirea unui tip tablou – exemplu */

typedef float tipTablou[100];

tipTablou a,b;
-------------------------------------------------------------
```

- **Tabloul** este o **structură de date statică** pentru care rezervarea de memorie se realizează în **faza de compilare** a programului.

- O valoare structurată (o instanţă) a unui tip de date `TipTablou` având componentele $c_1, c_2..., c_n$ poate fi iniţializată printr-un **constructor de tablou** şi o operaţie de atribuire:

```
tipElement numeTablou[nrElemente]={expr_const0,
expresie_const1,...};
```

```
-------------------------------------------------------------
/* Construcţia unui tablou */

int vect[5]={0,1,33,4,8};
int mat[2][3]={{11,12,13},{1,2,3}};              [1.4.1.d]
char str[9]="Turbo C++";
-------------------------------------------------------------
```

- Operatorul invers al unui constructor este **selectorul**. Acesta selectează o componentă individuală a unui tablou.

- Fiind dată o variabilă tablou `vector`, un **selector de tablou** se precizează adăugând numelui tabloului, indexul `i` al componentei selectate: `vector[i]`.

- În locul unui indice constant poate fi utilizată o **expresie index** (de indice).

    - Evaluarea acestei expresii în timpul execuţiei programului, conduce la determinarea componentei selectate.

- Dacă tipul de bază al unui tablou este **ordonat**, atunci în cadrul tipului tablou poate fi definită o relaţie de **ordonare naturală**.

- **Ordonarea naturală** a două tablouri de acelaşi tip este determinată de către acele componente corespunzătoare care sunt diferite şi care au cel mai mic indice:

```
(2,3,5,8,9)>(2,3,5,7,11)

    'CAPAC' < 'COPAC'
```

```
-------------------------------------------------------------
/* Ordonarea naturală a două tablouri */

typedef tipElement tipTablou[n];

tipTablou x,y;                                   /*1.4.1.e*/

  (x<y) <=> Ek:0≤k<n:((Ai:0≤i<k:x[i]=y[i])&(x[k]<y[k]))
-------------------------------------------------------------
```

- Un exemplu clasic în acest sens îl constituie **ordonarea lexicografică** a tablourilor de caractere.

- Deoarece toate componentele unui tip tablou `tipTablou` aparţin aceluiaşi tip de bază `tipElement` avem:

$$\text{Card}(\texttt{tipTablou}) = \text{Card}(\texttt{tipElement})^{\text{Card(tipIndice)}}.$$

- Tablourile de o singură dimensiune se mai numesc şi **tablouri liniare.**

    - **Accesul** la oricare element al unui astfel de tablou se realizează utilizând un singur indice în mecanismul de selecţie.

- `tipElement` precizat la definirea unui tip de date tablou poate fi la rândul său un tip tablou.
    - Prin acest mecanism se definesc **tablourile de mai multe dimensiuni**.
    - Astfel de tablouri se numesc de obicei **matrice.**
    - Accesul la un element al matricei se realizează utilizând un număr de indici egal cu numărul de dimensiuni ale matricei:

```
                mat[i,j,k,… ,m,n]
```

```
-----------------------------------------------------------------
/* Definirea unui tip matrice */

typedef tipElement tipTablou[nrElemente];
typedef tipTablou tipMatrice[nrElemente1];      /*1.4.1.f*/
-----------------------------------------------------------------
```

- O matrice poate fi definită direct ca şi tablou multidimensional:

```
-----------------------------------------------------------------
/* Definirea unei matrici */

tipElement numeMatrice[nrElemente][nrElemente1];/*1.4.1.g*/
-----------------------------------------------------------------
/* Definirea unui tip tablou multidimensional (matrice) */

typedef tipElement tipTablou[nrElemente1][nrElemente2]...
            [nrElementeN];
tipTablou matrice;                              /*1.4.1.h*/
-----------------------------------------------------------------
```

- Tipul de date abstract tablou poate fi prezentat ca şi în secvenţa [1.4.1.i].
- **Exemplu** de implementare a unui astfel de tip de date abstract [1.4.1.j].

```
-----------------------------------------------------------------
TDA Tablou

Model matematic: Secvenţă de elemente de acelaşi tip. Indicele
    asociat aparţine unui tip ordinal finit. Există o
    corespondenţă biunivocă între indici şi elementele
    tabloului.

Notaţii:
     tipElement -tipul elementelor tabloului;
     a - tablou unidimensional;
     i - index;                                    [1.4.1.i]
     e - obiect de tip tipElement.

Operaţii:
    DepuneInTablou(a,i,e) – operator procedură care depune
        valoarea lui e în cel de-al i-lea element al tablolui
        a;
    e= FurnizeazăDinTablou(a,i) – operator funcţie care
        returnează valoarea celui de-al i-lea element al
        tabloului a.
```

```
-------------------------------------------------------------
/* Exemplu de implementare                          /*1.4.1.j*/

#define numarMaxElem valoareIntreaga
typedef tipElement tipTablou[numarMaxElem];

  int i;
  tipTablou a;
  tipElement e;

  a[i]=e; /*DepuneInTablou(a,i,e)*/
  e=a[i]; /*FurnizeazaDinTablou(a,i)*/
-------------------------------------------------------------
```

1.4.2. Tehnici de căutare în tablouri

- **Formularea problemei**: se presupune că este dată o colecţie de date, în care se cere să se localizeze (fixeze) prin căutare un anumit element.

- Se consideră că mulţimea celor `n` elemente este organizată în forma unui un **tablou liniar**:

```
tipElement a[n];
```

- De regulă `tipElement` are o structură de **articol** cu un câmp specific pe post de **cheie**.

- Sarcina **căutării** este aceea de a găsi un element al tabloul `a`, a cărui cheie este identică cu o cheie dată `x` .

- Indexul `i` care rezultă din procesul de căutare satisface relaţia `a[i].cheie=x` şi el permite accesul şi la alte câmpuri ale elementului localizat

- Se presupune în continuare că `tipElement` constă numai din câmpul `cheie`, adică el este chiar cheia.

    - Cu alte cuvinte se va opera de fapt cu un **tablou de chei**.

1.4.2.1. Căutarea liniară

- Atunci când **nu** există nici un fel de informaţii referitoare la colecţia de date în care se face căutarea, tehnica evidentă este aceea de a parcurge în mod **secvenţial** colecţia prin incrementarea indexului de căutare pas cu pas.

- Această metodă se numeşte **căutare liniară** şi este ilustrată în secvenţa [1.4.2.1.a].

```
-------------------------------------------------------------
/*căutare liniară - varianta while */
                                                 /*1.4.2.1.a*/
    tipElement a[n];
    tipElement x;

    int i=0;
    while ((i<n-1) && (a[i]!=x))
     i++;
```

```
    if (a[i]!=x ){ /*în tablou nu există elementul căutat*/
    }
      else{ /*avem o coincidenţă la indicele i*/
    }
```

- Există două condiţii care finalizează căutarea:
  - (1) Elementul a fost găsit adică `a[i]=x` .
  - (2) Elementul nu a fost găsit după parcurgerea integrală a tabloului adică `i=n-1` şi `a[i]≠x`.
- **Invariantul** buclei, care de fapt este condiţia care trebuie îndeplinită înaintea fiecărei incrementări a indexului `i` , este:

  $(0 <= i < n-1) \& (A_k : 0 <= k < i : a[k] <> x)$

  - Acest invariant exprimă faptul că pentru valori ale lui `k<i` , nu există coincidenţă.
- **Condiţia de terminare** a căutării este:

  $((i = n-1) \mathbf{OR} (a[i] = x)) \& (A_k : 0 <= k < i : a[k] <> x)$

  - Această condiţie poate fi utilizată într-un algoritm de căutare bazat pe ciclul **DO** [1.4.2.1.b].

```
/*căutare liniară - varianta do*/
                                                    /*1.4.2.1.b*/
    tipElement a[n];
    tipElement x;

    int i=-1;
    do{
       i++;
    }while ((i<n-1) && (a[i]!=x))
    if (a[i]!=x ){ /*în tablou nu există elementul căutat*/
    }
      else{ /*avem o coincidenţă la indicele i*/
    }
```

- În legătură cu acest algoritm, indiferent de implementare, se precizează următoarele:
  - La părăsirea buclei mai trebuie realizată o comparaţie a lui `a[i]` cu `x` în vederea stabilirii existenţei sau nonexistenţei elementului căutat (dacă `i=n-1`).
  - Dacă elementul este găsit, el este elementul cu cel mai mic indice în situaţia în care există mai multe elemente identice.
- Desigur există şi varianta clasică a căutării liniare implementată cu instrucţiunea **for**

```
/*căutare liniară - varianta for */          /*1.4.2.1.c*/

/*returnează valoarea n pentru x negăsit */
/*returnează o valoare i<n pentru x găsit pe poziţia i */
```

```
int cautare_liniara(int x,int a[],int n)
{int i;
 for(i=0;i<n && a[i]-x;i++);
 return i;
}
```

- După cum se observă, fiecare pas al algoritmului necesită evaluarea expresiei booleene (care este formată din doi factori) urmată de incrementarea indexului.
    - Acest proces poate fi simplificat şi în consecinţă căutarea poate fi accelerată, prin simplificarea expresiei booleene.
    - O soluţie de a simplifica expresia este aceea de a găsi **un singur factor** care să-i implice pe cei doi existenţi.
    - Acest lucru este posibil dacă se garantează faptul că va fi găsită cel puţin o potrivire.
- În acest scop tabloul `a` se completează cu un e**lement adiţional** `a[n]` căruia i se atribuie iniţial valoarea lui `x` (elementul căutat). Tabloul devine:

```
tipElement a[n+1];
```

- Elementul `x` poziţionat pe ultima poziţie a tabloului se numeşte **fanion**, iar tehnica de căutare, **tehnica fanionului**.
    - Evident, dacă la părăsirea buclei de căutare `i = n`, rezultă că elementul căutat **nu** se află în tablou.
- Algoritmul de căutare astfel modificat apare în [1.4.2.1.d].

```
/*căutare liniară tehnica fanionului (while) */

    tipElement a[n+1];                          /*1.4.2.1.d*/
    tipElement x;

    int i=0;
    a[n]=x;
    while (a[i]!=x)
     i++;
    if (i==n){ /*în tablou nu există elementul căutat*/
    }
      else{  /*avem o coincidenţă la indicele i*/
    }
```

- **Condiţia de terminare** dedusă din acelaşi invariant al ciclului este:

$$(a[i]=x)\&(A_k:0<=k<i:a[k]<>x)$$

- În secvenţa [1.4.2.1.e] apare acelaşi algoritm implementat cu ajutorul ciclului **do**, iar în secvenţa [1.4.2.1.f] cu ajutorul ciclului **for.**

```
/*căutare liniară tehnica fanionului (do) */
```

```
                                                    /*1.4.2.1.e*/
    tipElement a[n+1];
    tipElement x;

    int i=-1;
    a[n]=x;
    do{
       i++;
    }while (a[i]!=x)
    if (i==n){ /*în tablou nu există elementul căutat*/
    }
      else{ /*avem o coincidenţă la indicele i*/
    }
-----------------------------------------------------------------
/*căutare liniară tehnica fanionului (for)*/
/*returnează valoarea n pentru x negăsit */
/*returnează o valoare i<n pentru x găsit  pe poziţia i */

int cautare_fanion(int x,int a[],int n)        /*1.4.1.f*/
{int i;
 a[n]=x;
 for(i=0;a[i]-x;i++);
 return i;
}
-----------------------------------------------------------------
```

- **Performanţa** căutării liniare este O(*n*/2), cu alte cuvinte, în medie un element este găsit după parcurgerea a jumătate din elementele tabloului.

1.4.2.2. Căutarea binară

- Procesul de căutare poate fi mult accelerat dacă se dispune de **informaţii suplimentare** referitoare la datele căutate.

- Este bine cunoscut faptul că o căutare se realizează mult mai rapid într-un **masiv de date ordonate** (spre exemplu într-o carte de telefoane sau într-un dicţionar).

- În continuare se prezintă un algoritm de căutare într-o **structură de date ordonată**, în cazul de faţă un **tablou** care satisface predicatul A_k:

```
-----------------------------------------------------------------
/* Definirea unui tablou ordonat*/

   tipElement a[n];

     Ak:0<k<=n-1:a[k-1]<= a[k]                  [1.4.2.2.a]
-----------------------------------------------------------------
```

- **Ideea de bază** a căutării binare**:**

  - Se inspectează un element aleator `a[m]` şi se compară cu elementul căutat `x`.

  - Dacă `a[m]` este egal cu `x` căutarea se termină.

- Dacă `a[m]` este mai mic decât `x`, se restrânge intervalul de căutare la elementele care au indici mai mari ca `m`.

- Dacă `a[m]` este mai mare decât `x`, se restrânge intervalul la elementele care au indicii mai mici ca `m`.

- În consecinţă rezultă algoritmul denumit **căutare binară**.

- Variabilele `stanga` şi `dreapta` sunt indici şi ele precizează limitele stânga respectiv dreapta ale intervalului în care elementul ar mai putea fi găsit.

```
-------------------------------------------------------------
/*cautare binară */
                                                  /*1.4.2.2.b*/
      tip_element a[n]; /*n>0*/
      tip_element x;
      tip_indice stanga,dreapta,m;
      int gasit;

      stanga=0; dreapta=n-1; gasit=0;
      while((stanga<=d)&&(!gasit))
      {
        m=(stanga+dreapta)/2;  /*sau orice valoare cuprinsă
                                  între stanga şi dreapta*/
        if(a[m]==x)
            gasit=1;
          else
            if(a[m]<x)
                stanga=m+1;
              else
                dreapta=m-1;
      }
      if(gasit) /*avem o coincidenţă la indicele i*/
-------------------------------------------------------------
```

- **Invariantul buclei**, adică condiţia ce trebuie îndeplinită înaintea fiecărui pas este:

$$(stanga <= dreapta)\&(A_k: 0 <= k < stanga: a[k] < x)\&$$

$$(A_k: dreapta < k <= n-1: a[k] > x)$$

- **Condiţia de terminare** este:

$$gasit\ \mathbf{OR}\ ((stanga > dreapta)\&(A_k: 0 <= k < stanga: a[k] < x)\&$$

$$(A_k: dreapta < k <= n-1: a[k] > x))$$

- Alegerea lui `m` este **arbitrară** în sensul că ea **nu** influenţează corectitudinea execuţiei algoritmului, în schimb influenţează eficienţa sa.

- **Soluţia optimă** în acest sens este alegerea elementului din **mijlocul intervalului**, întrucât ea elimină la fiecare pas jumătate din intervalul în care se face căutarea.

  - În consecinţă rezultă că **numărul maxim** de paşi de căutare va fi $O(log_2 n)$.

  - Această performanţă reprezintă o îmbunătăţire remarcabilă faţă de căutarea liniară unde numărul mediu de căutări este $n/2$.

- O variantă mai compactă a căutării binare apare în secvenţa următoare [1.4.2.2.c]

```
-------------------------------------------------------------
/* căutare binară */

int cautare_binara(int x,int a[],int n){
 int stanga=0,dreapta=n-1,m;
 do{
    (x>a[m=(stanga+dreapta)/2])?(stanga=m+1):(dreapta=m-1);
                                                  [1.4.2.2.c]
   }while(a[m]-x && stanga<=dreapta);
 return(a[m]-x)?n:m;
}
-------------------------------------------------------------
```

- Eficienţa căutării binare poate fi îmbunătăţită ca şi în cazul căutării liniare, prin **simplificarea condiţiei de terminare**.

  - Acest lucru se poate realiza simplu dacă **se renunţă** la ideea de a termina algoritmul de îndată ce s-a stabilit coincidenţa.

  - La prima vedere acest lucru **nu** pare prea înţelept.

  - La o examinare mai atentă, se poate observa că câştigul în eficienţă obţinut la fiecare pas este mai substanţial decât pierderea provocată de câteva comparaţii suplimentare de elemente.

  - Se reaminteşte faptul că numărul de paşi este $log_2 n$.

- Soluţia cea mai rapidă are la bază următorul **invariant**:

  $(A_k: 0<=k<stanga: a[k]<x) \& (A_k: dreapta<k<=n-1: a[k]>x)$

- Procesul de căutare continuă până când intervalul de căutare ajunge de dimensiune banală (0 sau 1 element). Această tehnică este ilustrată în [1.4.2.2.d].

```
-------------------------------------------------------------
/*căutare binară ameliorată */
                                              /*1.4.2.2.d*/
      tip_element a[n]; /*n>0*/
      tip_element x;
      tip_indice stanga,dreapta,m;

      stanga=0; dreapta=n;
      while(stanga<dreapta){
           m=(stanga+dreapta)/2;
        if(a[m]<x)
           stanga=m+1;
         else
           dreapta=m;
       }
       if(dreapta>n) /*nu există elementul căutat*/;
       if(dreapta<=n)
         if(a[dreapta]==x)
           /*elementul există*/;
           else /*elementul nu există*/;
-------------------------------------------------------------
```

- Condiţia de terminare în acest caz este `stanga =dreapta`.

- Cu toate acestea, egalitatea `stanga = dreapta` **nu** indică automat găsirea elementului căutat.

- Dacă `dreapta > n` **nu** există nici o coincidenţă. S-a căutat un element mai mare decât cel mai mare element al tabloului.

- Dacă `dreapta<=n` se observă că elementul `a[dreapta]` corespunzător ultimului indice `d` **nu** a fost comparat cu cheia.

    - În consecinţă este necesară efectuarea în afara ciclului a unui test `a[dreapta]=x` care de fapt stabileşte existenţa coincidenţei.

- Acest algoritm asemenea căutării liniare, găseşte elementul cu **cel mai mic indice**, lucru care **nu** este valabil pentru căutarea binară normală.

- O altă variantă de implementare a căutării binare ameliorate apare în secvenţa [1.4.2.2.e].

```
-------------------------------------------------------------
/* căutare binară ameliorata - varianta 2 */

int cautare_binara_ameliorata(int x,int a[],int n){
 int stanga=0,dreapta=n,m;
 do{
    (x>a[m=(stanga+dreapta)/2])?(stanga=m+1):(dreapta=m);
                                              /*1.4.2.2.e*/
   }while(stanga<dreapta);
 return
  ((dreapta>=n)||(dreapta<n && a[dreapta]-x))?n:dreapta;
}
-------------------------------------------------------------
```

1.4.2.3. Tehnica căutării prin interpolare

- O altă îmbunătăţire posibilă a metodei de căutare binară, constă în a determina cu **mai multă precizie** locul în care se face căutarea în intervalul curent, în loc de a selecta întotdeauna elementul de la mijloc.

    - Astfel când se caută într-o structură ordonată, spre exemplu cartea de telefon, un nume începând cu B este căutat la începutul cărţii, în schimb un nume care începe cu V este căutat spre sfârşitul acesteia.

- Această metodă, numită **căutare prin interpolare**, necesită o simplă modificare în raport cu căutarea binară.

- Astfel, în **căutarea binară**, noul loc în care se realizează accesul este întotdeauna mijlocul intervalului curent de căutare.

    - Acest loc este determinat cu ajutorul relaţiei de mai jos, unde `stanga` şi `dreapta` sunt **limitele intervalului de căutare**:

$$m = stânga + \frac{1}{2} * (dreapta - stânga)$$

- Următoarea evaluare a lui m, numită **căutare prin interpolare**, se consideră a fi mai performantă:

$$m = stânga + \frac{x - a[stânga]}{a[dreapta] - a[stânga]} * (dreapta - stânga)$$

  - Evaluarea ţine cont de valorile efective ale cheilor care mărginesc intervalul, în stabilirea indicelui la care se face următoarea căutare

  - Această estimare însă, presupune o **distribuţie uniformă** a valorilor cheilor pe domeniul vizat.

```
-----------------------------------------------------------------
/* căutare prin interpolare  */

int cautare_interpolare(element a[],int ultim,int x){
      long m,stanga,dreapta;
      stanga =0;
      dreapta=ultim-1;                              [1.4.2.1.m]
      do{
          m=stanga+((x-a[stanga].cheie)*(dreapta-stanga))/
            (a[dreapta].cheie-a[stanga].cheie);
          if (x>a[m].cheie) stanga=m+1;
             else dreapta=m-1;
      }
      while ((stanga<=dreapta)&&(a[m].cheie!=x)&&
             (a[dreapta].cheie!=a[stanga].cheie)&&
             (x>=a[stanga].cheie)&&(x<=a[dreapta].cheie));
      if (a[m].cheie==x) return m;
        else return ultim;
}
-----------------------------------------------------------------
```

- **Proprietate**. Căutarea prin interpolare utilizează atât în caz de reuşită cât şi în caz de nereuşită mai puţin de $lg\,(lg\,(n)+1)$ comparaţii [Se 88].

- Funcţia $lg\,(lg\,n)$ este o funcţie care creşte extrem de încet.

  - Astfel spre exemplu, pentru valori ale lui n de ordinul 10^9, $lg(lg\ n) < 5$.

1.4.3. Structura articol. Tipul de date abstract articol

- Metoda cea mai generală de a obţine a **tipuri structurate** este aceea de a **reuni** elemente ale mai multor tipuri, unele dintre ele fiind la rândul lor structurate, într-un tip **compus**.

- **Exemple** în acest sens sunt:

  - **Numerele complexe** din matematică care sunt compuse din două numere reale.

  - **Punctele de coordonate** compuse din două sau mai multe numere reale în funcţie de numărul de dimensiuni ale spaţiului la care se referă.

- În matematică un astfel de **tip compus** se numeşte **produsul cartezian** al **tipurilor constitutive.**

- **Domeniul de valori** al **tipului compus** constă din toate combinaţiile posibile ale valorilor tipurilor componente, selectând câte o singură valoare din fiecare.

- O astfel de combinaţie se numeşte **n-uplu**.

- Numărul total de combinaţii este egal cu produsul numerelor de elemente ale fiecărui tip constitutiv, adică **cardinalitatea tipului compus** este egală cu **produsul cardinalităţilor** tipurilor constitutive.

- Termenul care descrie o **dată compusă** de această natură în programare este cuvântul **record** (**articol**) respectiv **struct.**

- În secvenţa [1.4.3.a.] se prezintă modul generic de definire a unui **tip articol** în limbajul C.

```
-----------------------------------------------------------------
/*Definire structură */

struct nume_structura{
     tip1 lista_numeCâmp1;
     tip2 lista_numeCâmp2;                          [1.4.3.a]
          ...
     tipn lista_numeCâmpn;
} lista_var_structura;
-----------------------------------------------------------------
```

- Exemple de definire a unor tipuri de date articol [1.4.3.b].

```
-----------------------------------------------------------------
/* Exemple de definire structuri articol */

struct data {
     int zi, luna, an;
     } data_nasterii, data_inscrierii;          [1.4.3.b]

struct {
     int zi, luna, an;
     } data_nasterii, data_inscrierii;

struct data{
     int zi, luna, an;
     }

struct data data_nasterii, data_inscrierii;

typedef struct data {
     int zi, luna, an;
     }data_calendar;
data_calendar data_nasterii, data_inscrierii;

typedef struct structura_persoana {
         char[12] nume;
         char[12] prenume;
         data_calendar dataNastere;
         enum stare(necasatorit,casatorit,divortat,vaduv)
              sitSociala;
         enum sexul(barbatesc,femeiesc)sex;
       }persoana;
```

- **Identificatorii** numeCâmp$_1$, numeCâmp$_2$, ..., numeCâmp$_n$ introduşi la definirea tipului, sunt **nume** conferite de către programator componentelor individuale ale tipului.

- Identificatorii sunt utilizaţi ca şi **selectori de articol** care permit accesul la câmpurile unei variabile structurate de tip articol.

- Pentru o variabilă x de tip nume_structura, cea de-a i-a componentă, adică cel de-al i-lea câmp poate fi precizat prin notaţia "**dot**" sau "**punct**":

  x.numeCâmp$_i$

- O atribuire a respectivei componente poate fi precizata prin construcţia sintactică:

  x.numeCâmp$_i$ = x$_i$

  unde x$_i$ este o valoare (expresie) de tip tip$_i$.

- În secvenţa [1.4.3.c] apar câteva exemple de instanţiere a tipurilor articol anterior definite.

{Exemple de instanţiere a structurilor articol}

```
data_calendar d;
persoana p;                                    [1.4.3.c]

d.luna=9;
p.nume=”popescu”;
p.dataNastere.zi=24;
p.sitSociala=necasatorit;
```

- Exemplul care se referă la tipul persoana, pune în evidenţă faptul că constituenţii unui tip articol pot fi la rândul lor structuraţi, această situaţie conducând la **concatenarea** selectorilor.

- În mod evident, tipuri structurate diferite pot fi utilizate într-o manieră încuibată.

  - Astfel cea de-a i-a componentă a unui tablou a, care la rândul său este o componentă a unei variabile structurate x se precizează prin construcţia sintactică:

    x.a[i]

  - Componenta având numele de selector numeCâmp aparţinând celei de-a i-a componente de tip articol a unui tablou a este precizată prin construcţia:

    a[i].numeCâmp.

- Din punct de vedere formal tipul de date abstract articol este prezentat în secvenţa [1.4.3.d.].

TDA Articol [1.4.3.d]

Modelul matematic:

O colecţie finită de elemente numite câmpuri, care pot aparţine unor tipuri diferite. Exista o corespondenţă

biunivocă între lista identificatorilor de câmpuri şi colecţia de elemente.

Notatii:
a - obiect de tip articol;
id - identificator nume de câmp;
e - obiect de acelaşi tip cu câmpul *id* din articolul *a*.

Operatori:
***DepuneArticol**(a,id,e)* - procedură care memorează valoarea lui e în câmpul *id* al lui *a*;
*e= **FurnizeazăArticol**(a,id)* - funcţie care returnează valoarea câmpului *id* din *a.*

- Secvenţa [1.4.3.e.] prezintă câte un exemplu de implementare a tipului de date abstract articol

  - După cum se observă, fazele de definire şi de instanţiere pot fi separate sau se pot cumula în diferite moduri, limbajul C permiţând multă flexibilitate în acest sens

```
/* Exemplu de implementare */

struct punct {
  int x;
  int y;
}
struct dreptunghi {
  struct punct varf1;
  struct punct varf2;
}                                                    [1.4.3.e]
struct punct p1; double dist;
struct dreptunghi fereastra;
```

```
//Exemplu de utilizare

dist=sqrt((double)(p1.x*p1.x)+(double)(p1.y*p1.y));
fereastra.varf1.x=188;
```

1.4.4. Structura uniune

- Limbajul C defineşte structura **uniune.**

- O uniune este de fapt o variabilă care poate memora la momente diferite de timp obiecte de diferite tipuri şi dimensiuni. Compilatorul este acela care păstrează evidenţa şi aliniază în mod corespunzător datele memorate.

- Uniunile au fost concepute în ideea de a manipula tipuri diferite de date într-o singură locaţie (zonă) de memorie.

- Structura **uniune** este de fapt o variantă simplificată a articolului cu variante definit în PASCAL.

```
-------------------------------------------------------------
/* Definirea structurii uniune */

union indictor_uniune {
     int valoare_intreaga;
     float valoare_reala;
     char *sir_caractere;
} uniune
-------------------------------------------------------------
```

- În acest exemplu variabila `uniune` are o lungime dependentă de tipul variabilei care i-a fost atribuită. Acest lucru se realizează transparent pentru utilizator, care nu trebuie să ia nici o măsură de precauţie în acest sens.

    - Cu alte cuvinte, pentru orice utilizare corectă a tipului, valoarea memorată va aparţine ultimului tip memorat.

- Accesul la componentele structurii se realizeză utilizând notaţia dot.

    `uniune.membru` sau `pointer_uniune.membru`

1.4.5. Structura secvenţă. Tipul de date abstract secvenţă

- Caracteristica comună a structurilor de date prezentate până în prezent (tablouri, articole, mulţimi) este aceea că toate au **cardinalitatea finită**, respectiv cardinalitatea tipurilor lor componente este finită.

    - Din acest motiv, implementarea lor nu ridică probleme deosebite.

- Cele mai multe dintre aşa numitele structuri avansate: secvenţele, listele, arborii, grafurile, etc, sunt caracterizate prin **cardinalitate infinită**.

    - Această diferenţă faţă de structurile clasice este de importanţă profundă având consecinţe practice semnificative.

- Spre exemplu **structura secvenţă** având tipul de bază T_0 se defineşte după cum urmează [14.5.a]:

```
-------------------------------------------------------------
     S0=< >  (secvenţa vidă)
     Si=<Si-1,si>    unde 0 < i şi si∈ T0                [1.4.5.a]
-------------------------------------------------------------
```

- Cu alte cuvinte, o secvenţă cu tipul de bază T_0, este:

    - (1) Fie o secvenţă **vidă.**

    - (2) Fie o **concatenare** a unei secvenţe (cu tipul de bază T_0) cu o valoare (s_i) a tipului T_0.

- **Definirea recursivă** a unui tip secvenţă conduce la o **cardinalitate infinită**.

- Fiecare valoare a tipului secvenţă conţine în realitate un număr **finit** de componente de tip T_0.

- Acest număr este **nemărginit** deoarece, pornind de la orice secvenţă se poate construi o secvenţă mai lungă.

- Consecinţe importante ale acestui fapt:

  - (1) Volumul de memorie necesar reprezentării unei structuri avansate, **nu** poate fi cunoscut în momentul compilării.

  - (2) Este necesară aplicarea unor scheme de **alocare dinamică a memoriei**, în care memoria este alocată structurilor care "cresc" şi este eliberată de către structurile care "descresc".

- Pentru a implementa această cerinţă, este necesar ca limbajul superior utilizat în implementare să fie prevăzut cu acces la **funcţii sistem** care permit:

  - **Alocarea / eliberarea dinamică** a memoriei.

  - **Legarea / referirea dinamică** a componentelor.

- În aceste condiţii cu ajutorul instrucţiilor limbajului pot fi descrise şi utilizate structuri avansate de date.

- Tehnicile de generare şi de manipulare a unor astfel de structuri avansate sunt prezentate în capitolele următoare ale cursului.

- Structura **secvenţă** este din acest punct de vedere o structură **intermediară**:

  - Ea este o **structură avansată** din punctul de vedere al cardinalităţii care este infinită.

  - Este însă atât de curent utilizată încât includerea sa în mulţimea **structurilor fundamentale** este consacrată.

  - Această situaţie este influenţată şi de faptul că, alegerea unui **set potrivit de operatori** referitori la **structura secvenţă**, permite implementatorilor să adopte reprezentări potrivite şi eficiente ale acestei structuri.

  - Drept consecinţă, **mecanismele alocării dinamice** a memoriei devin suficient de simple pentru a permite o implementare eficientă, neafectată de detalii, la nivel de limbaj superior.

1.4.5.1. Tipul de date abstract secvenţă

- Includerea **structurii secvenţă** în rândul structurilor fundamentale, presupune **restrângerea setului de operatori** de o asemenea manieră încât se permite numai **accesul secvenţial** la componentele structurii.

  - Această structură este cunoscută şi sub denumirea de **fişier secvenţial** sau pe scurt **fişier**.

- **Structura secvenţă** este supusă **restricţiei** de a avea componente de acelaşi tip, cu alte cuvinte este o structură **omogenă.**

- Numărul componentelor, denumit şi **lungime** a secvenţei, se presupune a fi **necunoscut** atât în faza de compilare, cât şi în faza de execuţie a codului.

    - Mai mult chiar, acest număr **nu** se presupune constant el putându-se modifica în timpul execuţiei.

- Cardinalitatea tipului secvenţă este în consecinţă **infinită**.

- În dependenţă de maniera de implementare a structurilor secvenţă, acestea se înregistrează de regulă pe suporturi de memorie externă reutilizabile cum ar fi **benzile magnetice** sau **discurile în alocare secvenţială.**

    - Acest lucru face posibilă tratarea relativ simplă a structurilor secvenţă şi permite încadrarea lor în rândul **structurilor fundamentale**, deşi de drept ele aparţin **structurilor dinamice**.

- Secvenţa este o **structură ordonată.** Ordonarea elementelor structurii este stabilită de ordinea în timp a creării componentelor individuale.

- Datorită mecanismului de acces **secvenţial** preconizat, selectarea prin indici a componentelor individuale devine improprie.

- În consecinţă la definirea unui tip secvenţă se precizează numai **tipul de bază**.

```
TYPE TipSecvenţă = FILE OF TipDeBază;
```

- În cadrul unei secvenţe definite ca mai sus în orice moment este **accesibilă o singură componentă**, denumită **componentă curentă**, care este precizată printr-un **indicator** (pointer) asociat secvenţei.

    - Acest indicator avansează secvenţial, practic după execuţia oricărei operaţii asupra secvenţei.

- În plus oricărei secvenţe i se asociază un operator boolean standard **sfârşit de fişier** notat cu `Eof(f:TipSecventa)` care permite sesizarea sfârşitului secvenţei.

- Modul concret de acces la componente, precum şi posibilităţile de prelucrare efectivă, rezultă din semantica operatorilor definiţi pentru tipul de date abstract secvenţă [1.4.5.1.a].

- Implementările recente ale limbajelor Pascal şi C introduc în setul de instrucţiuni accesibile programatorului şi operatori pentru prelucrarea secvenţelor implementate ca şi fişiere secvenţiale.

- În secvenţa [1.4.5.1.b] apare o astfel de implementare definită în limbajul Pascal. Variante asemănătoare stau la dispoziţie şi în limbajul C.

TDA Secvenţă

Modelul matematic: secvenţă de elemente de acelaşi tip. Un indicator la secvenţă indică elementul următor la care se

poate realiza accesul. Accesul la elemente este strict secvenţial.

Notaţii:

TipElement - tipul unui element al secvenţei. Nu poate fi de tip secvenţă;
f - variabilă secvenţă;
e - variabilă de *TipElement*; [1.4.5.1.a]
b - valoare booleană;
numeFisierDisc - şir de caractere.

Operatori:

***Atribuie**(f,numeFisierDisc)* - atribuie variabilei secvenţă *f* numele unui fişier disc precizat;

***Rescrie**(f)* - procedură care mută indicatorul secvenţei la începutul lui *f* şi deschide fişierul *f* în regim de scriere. Dacă fişierul *f* nu există el este creat. Dacă *f* există, vechea sa variantă se pierde şi se creează un nou fişier *f* vid;

***ResetSecvenţă**(f)* - procedură care mută indicatorul la începutul secvenţei *f* şi deschide secvenţa în regim de consultare. Dacă *f* nu există se semnalează eroare de execuţie;

***DeschideSecvenţă**(f)* - procedură care în anumite implementări joacă rol de *rescrie* sau *reset* de secvenţă;

b:= ***Eof**(f)* - funcţie care returnează valoarea **true** dacă indicatorul secvenţei indică marcherul de sfârşit de fişier al lui *f*;

***FurnizeazăSecvenţă**(f,e)* - procedură care acţionează în regim de consultare. Atâta vreme cât *Eof(f)* este fals, furnizează în *e* elementul curent al secvenţei *f* şi avansează indicatorul acesteia;

***DepuneSecvenţă**(f,e)* - pentru secvenţa *f* deschisă în regim de scriere, procedura copiază valoarea lui *e* în elementul curent al secvenţei *f* şi avansează indicatorul acesteia;

***Adaugă**(f)* - procedură care deschide secvenţa *f* în regim de scriere, poziţionând indicatorul la sfârşitul acesteia, cu posibilitatea de a adăuga elemente noi numai la sfârşitul secvenţei.

***InchideSecvenţa**(f)* - închide secvenţa.

{Implementarea tipului secvenţă - Varianta Pascal}

```
TYPE TipElement = ... ;
     TipSecvenţă = TEXT;

VAR  f: TipSecvenţă;                                    [1.4.5.1.b]
```

```
     e: TipElement;
     numeFisierDisc: string;

{Atribuie(f,numeFisierDisc)}   assign(f,numeFisierDisc)
{Rescrie(f)}                   rewrite(f)
{DepuneSecvenţă(f,e)}          write(f,e) ; writeln(...)
{ResetSecvenţă(f)}             reset(f)
{Eof(f)}                       eof(f)
{FurnizeazăSecvenţă(f,e)}      read(f,e);  readln(...)
{Adaugă(f)}                    append(f)
{InchideSecvenţă(f)}           close(f)
-------------------------------------------------------------
```

1.4.5.2. Tipul de date abstract fişier cu acces direct

- Principalul dezavantaj al structurii secvenţă constă în faptul că accesul la componentele sale se realizează în manieră **strict secvenţială** prin intermediul unui pointer a cărui poziţionare este inaccesibilă programatorului în mod direct.

- Din acest motiv, pornind de la tipul de date abstract secvenţă, a fost dezvoltată o structură mai flexibilă numită **fişier cu acces direct**.

- Astfel, unui fişier cu acces direct i se asociază un **indicator** prin intermediul căruia pot fi accesate **oricare** din componentele fişierului.

- În acest scop fiecare **componentă** a fişierului este identificată printr-un **indice** care îi precizează **poziţia** în fişier exact ca şi la structura tablou.

    - Numerotarea componentelor începe cu `0,1,2,...` şi continuă până la `DimensiuneFişier-1`.

- Crearea şi modificarea structurii **fişier cu acces direct** se realizează însă tot în manieră **secvenţială**.

    - Cu alte cuvinte, se pot **adăuga** sau **modifica** înregistrări dar **nu** se pot **insera** altele noi între cele existente.

- Tipul de date abstract asociat acestei structuri apare în secvenţa [1.4.5.2.a] iar în secvenţa [1.4.5.2.b] un exemplu de implementare Pascal.

```
-------------------------------------------------------------
```

TDA Fişier cu acces direct

Modelul matematic: secvenţă de elemente de acelaşi tip. Un indicator asociat fişierului indică elementul la care se poate realiza accesul. Acesta poate fi oricare element al structurii. Fiecare componentă a structurii este identificată printr-un indice. Numărătoarea începe cu 0,1,2,... şi continuă până la *DimensiuneFisier-1*.

Notaţii:

TipElement - tipul unui element al fişierului. Nu poate fi de tip fişier;

f - fişier cu acces direct;

e - variabilă de *TipElement*; [1.4.5.2.a]
i - indice în fişier.

Operatori:

***Atribuie**(f,numeFisierDisc)* - atribuie variabilei fişier *f* numele unui fişier disc precizat;

***Rescrie**(f)* - procedură care mută indicatorul fişierului la începutul acestuia. Daca fişierul *f* nu există el este creat. Daca *f* există, vechea sa variantă se pierde şi se creează un nou fişier *f* vid. Indicatorul asociat fişierului ia valoarea 0;

***Reset**(f)* - procedură care mută indicatorul la începutul fişierului *f*. Indicatorul ia valoarea 0. Daca *f* nu există se semnalează eroare;

***DeschideFisier**(f)* - procedură care în anumite implementări joacă rol de *rescrie* sau *reset* de fişier;

i:= ***DimensiuneFisier**(f)* - funcţie care returnează numărul curent de componente al fişierului *f* (numărul de componente memorate în *f*);

i:= ***PoziţieFişier**(f)* - funcţie care returnează numărul înregistrării curente (cea precizată de indicatorul fişierului). La deschiderea unui fişier *PoziţieFişier* ia valoarea 0;

***Caută**(f,i)* - selectează înregistrarea precizată prin numărul *i* poziţionând indicatorul fişierului pe înregistrarea respectivă;

***FurnizeazăFişier**(f,e)* - procedură care furnizează în *e* elementul curent al fişierului *f* şi avansează indicatorul acestuia;

***Depune**(f,e)* - procedură care depune valoarea lui *e* în fişierul *f* în poziţia precizată de indicatorul fişierului şi avansează indicatorul la elementul următor;

***ÎnchideFisier**(f)* - închide fişierul *f*.

{Implementarea tipului secvenţă cu acces direct- Varianta Pascal}

```
TYPE TipElement = ... ;
     TipFisierAccesDirect = FILE OF TipElement;
     TipIndice = integer;

VAR  f: TipFisierAccesDirect;
     e: TipElement;                          [1.4.5.2.b]
     i: TipIndice;

   {Rescrie(f)}                rewrite(f)
   {Reset(f)}                  reset(f)
```

```
   {i:= DimensiuneFisier(f)}    i:= filesize(f)
   {i:= PoziţeFişier(f)}        i:= filepos(f)
   {Caută(f,i)}                 seek(f,i)
   {DepuneFişier(f,e)}          write(f,e); writeln(...);
   {FurnizeazăFişier(f,e)}      read(f,e); readln(...);
   {ÎnchideFişier(f)}           close(f)
-------------------------------------------------------------
```

- Variante similare de implementare ale **TDA secvenţă** există şi pentru limbajele C respectiv C++.

- Dacă la nivel principial conceptele prezentate rămân valabile, la nivel de implementare pot să apară diferenţieri importante.

1.5. Aplicaţii propuse

Aplicaţia 1.5.1

Se consideră o structură de date tablou de dimensiune `N`. Se cere să se realizeze un program interactiv care implementează următoarele operaţii:

- Creează în mod interactiv tabloul.
- Afişează conţinutul acestuia.
- Caută un element precizat prin următoarele tehnici:
  - (1) Căutare liniara.
  - (2) Tehnica fanionului.
  - (3) Căutare binară.
  - (4) Căutare binară performantă.
  - (5) Căutare prin interpolare.

Aplicaţia 1.5.2

Se consideră următoarele structuri de date:

```
-------------------------------------------------------------
typedef struct data {

     int zi, luna, an;

     }data_calendar;
```

```
data_calendar data_nasterii, data_inscrierii;


typedef struct structura_persoana {
         char[12] nume;
         char[12] prenume;
         data_calendar dataNastere;
         enum stare(necasatorit,casatorit,divortat,vaduv)
              sitSociala;
         enum sexul(barbatesc,femeiesc)sex;
       }persoana;


persoana angajati[10];
```

Se cere să se realizeze un program interactiv care implementează următoarele operaţii:

- Creează în mod interactiv tabloul angajati pentru un număr precizat de persoane.
- Caută în tablou o persoană după un nume precizat şi îi afişează caracteristicile.
- Adaugă o persoană în tablou.
- Şterge o persoană din tablou după următoarea tehnică: aduce ultima persoană în locul celei şterse şi decrementează contorul de persoane.

Aplicaţia 1.5.3

Se cere să se realizeze un program interactiv care efectuează operaţii cu mulţimi. Programul va accepta de la terminal elementele mulţimilor implicate şi operatorul specific, va efectua operaţia şi va vizualiza rezultatul. Se vor implementa următoarele operaţii:

- Reuniunea a două mulţimi.
- Diferenţa a două mulţimi.
- Intersecţia a două mulţimi.
- Testul de egalitate a două mulţimi.
- Comparaţie subset (inclus în) (<=).
- Comparaţie superset (include pe) (>=).
- Testul de apartenenţă la mulţime.

Operaţiile realizate se vizualizează în baza următorului model:

```
OPERATIA REUNIUNE
  ['a','b','c','d','e'] + ['c','e','f'] =
  ['a','b','c','d','e','f']


OPERATIA APARTENETA LA MULTIME

  'd' IN ['a','b','c','d','e'] = TRUE
```

Programul se va redacta în limbajul C++, iar TDA mulţime va fi implementat ca şi clasă, în două variante:

- Utilizând ca structură de date, o variabilă long unsigned.
- Utilizând ca structură de date un tablou boolean.

Aplicaţia 1.5.4

Se consideră o structură de date secvenţă ale cărei componente sunt articole cu următoarea structură:

```
-------------------------------------------------------------

    typedef struct nod {

        int grad;

        float sinus;

    };

-------------------------------------------------------------
```

Structura conţine valorile sinusurilor unghiurilor cuprinse între 1 grad si 90 grade cu pasul 1. Se cere:

- Să se creeze în mod dinamic structura de date.
- Să se recitească componentele secvenţei şi să se calculeze suma câmpurilor "Sinus".
- Să se creeze o altă secvenţă cu aceeaşi structură de noduri, în care valorile câmpurilor fiecărui element sunt mediile aritmetice ale câmpurilor aparţinând la două componente consecutive ale primei secvenţe.