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, **viteza, frecvenţa de lucru** şi **capacitatea lor de a memora** şi de **a avea acces** la cantităţi mari de informaţii au un rol hotărâtor şi sunt considerate caracteristici primordiale ale calculatoarelor
  - **Abilitatea 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** referitoare la 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**
  - În procesul de abstractizare anumite **proprietăţi** şi **caracteristici** ale obiectelor reale, pe care acestea le reprezintă, sunt **ignorate** întrucât pot fi considerate nerelevante pentru problema particulară tratată.
- O **abstractizare** este de fapt o **simplificare** a faptelor.
- În rezolvarea unei probleme cu ajutorul unui sistem de calcul un rol esenţial revine alegerii unei **abstractizări convenabile** a realităţii
  - **Alegerea abstractizării** 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:
  - (1) De **natura problemei** de rezolvat.
  - (2) 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 deosebit de dificilă întrucât **nu** este determinată în mod unic de facilităţile disponibile.
  - De regulă, alegerea reprezentării datelor se poate realiza la **diferite niveluri de abstractizare**, funcţie de obiectivul urmărit şi de limbajul de programare utilizat.
- În acest context, un **limbaj de programare** reprezintă:
  - Un **calculator abstract** capabil să înţeleagă **fraze** construite cu ajutorul **termenilor** definiţi în cadrul acestui limbaj de programare
  - **Termeni** 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**, 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 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.
- Limbajul de programare avut în vedere este limbajul ***C***
    - Acest limbaj acoperă în mod fericit o plajă largă situată între
        - (1) Limbajele orientate spre maşină sau limbaje dependente de maşină care lasă deschise problemele reprezentărilor şi
        - (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.

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) Valori 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 de 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 practica curentă, tipul asociat unei variabile este precizat printr-o **declaraţie explicită** de constantă, variabilă sau funcţie, declaraţie care precede textual utilizarea respectivei constante, variabile sau funcţii.
- **Caracteristicile** conceptului de tip 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.
    - (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.
- 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 de cifre binare prezente în memoria sistemului 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 **funcţie**.
    - (1) Funcţia **generalizează** noţiunea de operator.
        - În loc de a fi limitat la utilizarea exclusivă a operatorilor definiţi în cadrul limbajului de programare ("built-in" operators), folosind 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.
        - Un exemplu de procedură utilizată în această manieră este spre exemplu, funcţia de înmulţire a două matrici.
    - (2) Funcţiile **încapsulează** anumite părţi ale unui algoritm prin "localizare"
        - Aceasta înseamnă plasarea 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 funcţiile 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 a programului.
    - (3) Dacă apare necesitatea **modificării implementării** TDA-ului

- (3.1) Programatorul ştie cu certitudine locul în care trebuiesc efectuate modificările
- (3.2) Poate fi sigur că modificarea secţiunii respective **nu** produce modificări nedorite în restul codului programului, dacă nu se modifică formatul operatorilor TDA-ului respectiv

- (4) În plus, în afara secţiunii în care sunt definiţi operatorii, **TDA**-ul în cauză poate fi tratat ca un **tip primitiv**.

- **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** 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 set posibil 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.

- **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

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.

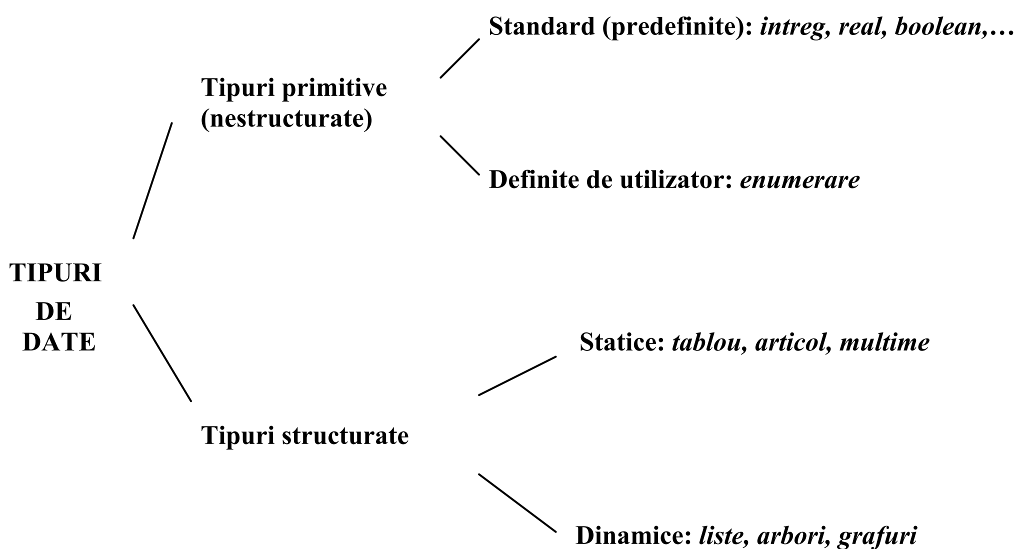
3) Nu există nici o **limitare** referitoare la numărul de operaţii care pot fi aplicate instanţelor unui model matematic precizat.

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.

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 situațiile 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 dezvoltate de utilizator**.
- **(1) Tipurile standard** sau **predefinite** includ de regulă numerele şi valorile logice.
    - Ele se bazează pe modele matematice consacrate (mulțimile de numere întregi, reale, logice, etc.) şi implementează operațiile specifice definite pe aceste categorii de numere.
    - Dacă între valorile individuale ale tipului există o relație de ordonare, atunci tipul se numeşte **ordonat** sau **scalar**.
- **(2) Tipuri primitive definite de utilizator**.
    - O metodă larg utilizată de construcție a unor astfel de tipuri este aceea a **enumerării** valorilor constitutive ale tipului.
    - Spre exemplu, într-un program care prelucrează figuri geometrice, poate fi introdus un tip primitiv numit *tip_figură* 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 de obicei **conglomerate** de valori componente ale unuia sau mai multor tipuri constitutive definite anterior motiv pentru care se numesc **tipuri structurate**.
    - **Tipuri structurate**. Sunt conglomerate de valori de componente. Dacă există un singur tip constitutiv, acesta se numeşte **tip de bază.**
        - Poate fi construită o întreagă **ierarhie** de astfel de structuri,
    - Pentru definirea tipurilor structurate, se utilizează **metode de structurare**. Metodele de structurare pot fi:
        - **Statice**: tabloul, articolul, mulțimea şi secventa (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 **instanțiere**
  - Prelucrarea acestor **variabile** se realizează cu ajutorul **operatorilor** asociați tipului.
- **În concluzie:**
  - Un **tip** este de fapt o manieră de **structurare** a informației.
  - (2) Pentru **tipurile nestructurate** este mai relevantă **mulțimea constantelor** tipului şi mai puțin gradul de structurare.
  - (3) Pe măsură ce tipul devine mai **complex**, **gradul de structurare** devine tot mai relevant.

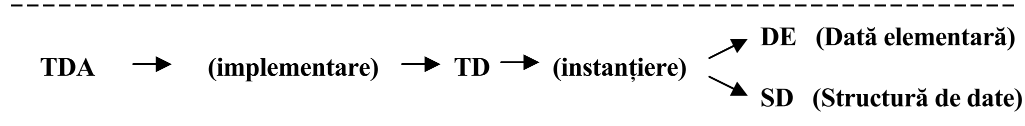
1.2.2.2. Implementarea unui TDA

- **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 funcțiilor care implementează operatorii specifici.
- 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 implementează pornind de la **tipurile de date de bază** care sunt definite în cadrul limbajului, utilizând **facilitățile de structurare** disponibile.
- În continuare definirea funcțiilor 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, spre exemplu aparținând unor biblioteci

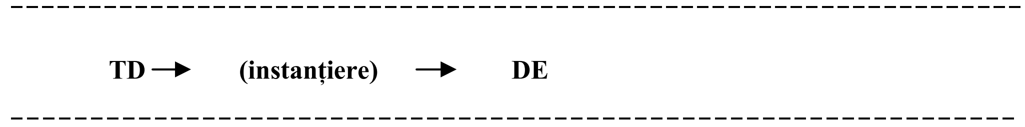
1.2.3. Tip de date abstract. Tip de date. Structură de date

- Se pune întrebarea: Care este semnificația termenilor de mai jos, care sunt adesea confundați?
  - **(1) Tip de date abstract** (TDA),
  - **(2) Tip de date** (TD) sau simplu **tip**
  - **(3) Structură de date** (SD)
  - (4) **Dată elementara** (DE)
- (1) După cum s-a precizat, un **tip de date abstract** este un model matematic, împreună cu totalitatea operațiilor definite pe acest model.
  - La nivel conceptual, algoritmii pot fi proiectați 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 definiți 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 consecință în cadrul programului se declară variabile încadrate în tipul respectiv, variabile care pot fi efectiv prelucrate şi care iau valori în mulțimea valorilor tipului.
  - Acest proces se numeşte **instanțiere** şi el de fapt conduce în cazul tipurilor nestructurate la generarea unei date elementare (**DE**) iar în cazul tipurilor structurate la generarea unei structuri de date (**SD**).
- (3) O **dată elementară** este o **instanță** a unui tip de date nestructurat
- (4) O **structură de date** este o **instanță** a unui tip de date structurat.
  - Ca atare legătura dintre aceste noțiuni 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 înțelege **definirea tipului** iar prin ***instanțiere***, **declararea unei variabile** asociate tipului.
- În cazul tipurilor predefinite faza de implementare lipseşte ea fiind înclusă intrinsec în limbajul de programare ca atare pentru aceste tipuri este valabilă formula:



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 entități reale.
- Pentru a evita această situație 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: $c_1, c_2, \ldots, c_n$.

```
------------------------------------------------------------
// Definirea tipului enumerare
enum TipEnumerare {c1,c2,c3,...,cn} variabila;
typedef enum {c1,c2,c3,...,cn} nume_tip;          [1.3.1.b]
------------------------------------------------------------
```

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

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

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

- În consecință utilizarea sa presupune atât faza de **implementare** (declarare a tipului) cât şi faza de **instanțiere** (declarare de variabile aparținătoare tipului).

```
------------------------------------------------------------------
/*Exemplu de implementare a tipului enumerare*/       /*1.3.1.c*/
typedef enum {dreptunghi,patrat,cerc,elipsa} TipFigura;

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

typedef enum {adevarat,fals} TipBoolean;

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

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

- 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 domeniul valorilor tipului)
- Aceşti identificatori pot fi utilizați ca şi constante în cadrul programului, sporind în acest fel considerabil gradul de înțelegere al textului sursă.

```
------------------------------------------------------------------
/*Exemplu de instanţiere a tipului enumerare*/       /*1.3.1.e */
TipCuloare 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 secvența [1.3.1.c] se pot introduce spre exemplu variabilele `c,z,g,b` [1.3.1.e].
  - În mod evident, această reprezentare la nivel de program este mult mai intuitivă decât spre exemplu:
    `c = 0;  z= 2;  g = 5 ;  b = 1`

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 fracționare** adică: INTEGER, BOOLEAN, CHAR, REAL.
- **(1) Tipul întreg** implementează o submulțime 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 însă că toate operațiile efectuate asupra acestui tip conduc la valori exacte şi se conformează regulilor obişnuite ale aritmeticii.

- Procesul de calcul se întrerupe în cazul obținerii unui rezultat în afara setului reprezentabil (depășirea capacității registrelor - DCR) de exemplu împărțirea cu zero.
- Pe mulțimea numerelor întregi în afara operatorilor clasici de **comparare** şi **atribuire**, se definesc şi operatori standard:
    - Operatori standard definiți pe mulțimea: adunare (+), scădere (-), înmulțire (*), împărțire întreagă (`/`) şi modulo (`%`)

```
m - n < (m / n)* n <= m
(m / n)* n + (m % n) = m
```

- **(2) Tipul real** implementează o submulțime 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.
    - Din acest motiv tipurile întreg respectiv real în marea majoritate a limbajelor de programare se tratează separat.
    - Operațiile se notează cu simbolurile consacrate cu excepția împărțirii numerelor reale care se notează cu (/).
    - Operațiile care conduc la valori care depăşesc domeniul de reprezentabilitate al implementării conduc la erori (DCR) - de exemplu împărțirea cu zero.
    - Implementările curente ale limbajelor de programare conțin mai multe categorii de tipuri întregi (simplu, fără semn, scurt, lung, etc.) respectiv mai multe tipuri reale (normal, dublu, lung) 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**, respectiv 0 semnifică **fals**.
    - Operatorii specifici definiți pentru acest tip sunt operatorii logici: **conjuncție**, **reuniune** şi **negație**

P	Q	P **AND** Q	P **OR** Q	**NOT** 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 mulțime 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
----------------------------------------------------------------
```

- În limbajul C tipurile `char` şi `int` sunt echivalente.
- Stabilirea naturii unui caracter:
  `('A' <= x)` **AND** `(x <= 'Z')` – x este literă mare;
  `('a' <= x)` **AND** `(x <= 'z')` – x este literă mică;
  `('0' <= x)` **AND** `(x <= '9')` – x este o cifră.
- Transformarea caracter-întreg se realizează în baza următoarelor relaţii [1.3.2.c]

```
----------------------------------------------------------------
// Corespondenţa întreg-caracter
char c; int n;
     n=c-'0      // transformare caracter-întreg      [1.3.2.c]
     c=n+'0';    // transformare întreg-caracter
------------------------------------------------------------
```

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 se precizează:
  - **(1) Metoda de structurare**, care este implementată direct de limbaj
  - **(2) Numele** tabloului de date rezultat
  - **(3) Tipul de bază** al tabloului
  - **(4) Tipul indice** al tabloului
  - **(5) Dimensiunea** sau dimensiunile tabloului

```
----------------------------------------------------------------
// Definirea unei structuri tablou -
//tip_element  nume_tablou[nr_elemente];
   float tablou[10];                             [1.4.1.b]
   char sir[10];
------------------------------------------------------------
```

  - **Metoda de structurare tablou** este indicată de prezenţa perechii de paranteze drepte []
  - `nume_tablou` precizează **numele tabloului** care se defineşte
  - `tip_element` precizează numele **tipului de bază**, adică tipul elementelor tabloului

- `tip_element` **nu** este supus nici unei restricții, adică el poate fi oricare alt tip (inclusiv un tip structurat)

  - Tipul indice este **în mod implicit** tipul întreg (`int`). Indicele ia valori în domeniul `[0, nr_elemente-1]`
  - Între parantezele dreapte se precizează în mod efectiv numărul de elemente ale tabloului (`nr_elemente`) adică **dimensiunea** acestuia

- Tabloul este o structură de date **statică** pentru care rezervarea de memorie se realizează în faza de compilare a programului.
- O structură **tablou** poate fi inițializată printr-un **constructor** de tablou şi o operație de atribuire:

```
tip_element nume_tablou[nr_elemente]={expr_const0,
expresie_const1,...};
```

```
-----------------------------------------------------------------
// Construcția unui tablou
int vect[5]={0,1,33,4,8};
int mat[2][3]={{11,12,13},{1,2,3}};
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 execuției programului, conduce la determinarea componentei selectate.
- Tablourile definite sunt tablouri de o singură dimensiune sau **tablouri liniare.**
  - **Accesul** la oricare element al unui astfel de tablou se realizează utilizând un singur indice în mecanismul de selecție.
- `tip_element` precizat la definirea unui tip de date tablou poate fi la rândul său un tip tablou.
  - Prin acest mecanism se pot defini **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 unei structuri tablou multidimensional
tip_element nume_tablou[nr_elemente1][nr_elemente2]...
             [nr_elementeN];
------------------------------------------------------------------
```

- Formal, TDA Tablou poate fi precizat cam şi în [1.4.1.e] iar o implementare a sa în limbajul C ca şi în [1.4.1.f]

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

<u>TDA Tablou</u>

Model matematic: Secvență de elemente de acelaşi tip. Indicele asociat aparține unui tip ordinal finit. Există o corespondență biunivocă între indici şi elementele tabloului.

```
Notații:
      TipElement -tipul elementelor tabloului;
      a - tablou unidimensional;
      i - index;                              [1.4.1.e]
      e - obiect de tip TipElement.
Operații:
     DepuneInTablou(a,i,e) - procedură care depune valoarea lui e
          în cel de-al i-lea element al tabloului a;
     e:= FurnizeazăDinTablou(a,i) - funcţie care returnează
          valoarea celui de-al i-lea element al tabloului a.
-----------------------------------------------------------------
/*Exemplu de implementare - Varianta C */ /*1.4.1.f*/
#define NumarMaxElem valoareIntreaga
typedef tip_element tip_tablou[NumarMaxElem];

  int i;
  tip_tablou a;
  tip_element 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 colecție de date, în care se cere să se localizeze (fixeze) prin căutare un anumit element.
- Se consideră că mulțimea celor `n` elemente este organizată în forma unui un **tablou liniar** `a`:

  `tip_element a[n];`
- De regulă `tip_element` poate fi o structură **struct** 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 relația `a[i].cheie=x` şi el permite accesul şi la alte câmpuri ale elementului localizat
- Se presupune în continuare că `tip_element` 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 informații referitoare la colecția de date în care se face căutarea, tehnica evidentă este aceea de a parcurge în mod **secvențial** colecția prin incrementarea indexului de căutare pas cu pas.
- Această metodă se numeşte **căutare liniară**
- În secvențele următoare sunt prezentate trei variante de căutare liniară

```
-----------------------------------------------------------
/*cautare liniara - varianta while */  /*1.4.2.1.a*/
    tip_element a[nr_elemente];
    tip_element x;
    int i=0;
    while ((i<n-1) && (a[i]!=x))
      i++;
```

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

- Există două condiţii care finalizează căutarea:
    - Elementul a fost găsit adică `a[i] = x` ;
    - Elementul **nu** a fost găsit după parcurgerea integrală a tabloului.

```
/*cautare liniara - varianta do */   /*1.4.2.1.b*/
    tip_element a[nr_elemente];
    tip_element x;
    int i=-1;
    do{
       i++;
    }while ((i<n-1) && (a[i]!=x))
    if (a[i]!=x ){
        /*în tablou nu exista elementul cautat*/
    }
    else{
        /*avem o coincidenta la indicele i*/
    }
```

- În legătură cu aceste variante de algoritm 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`)
    - Dacă elementul este găsit, el este elementul cu cel mai mic indice (dacă există mai multe elemente identice).

```
/*cautare liniara - varianta for */         /*1.4.2.1.e*/
/*returneaza n pentru negasit, respectiv i<n pentru gasit*/
/* pe pozitia 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) şi 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+1]` căruia i se atribuie iniţial valoarea lui `x` (elementul căutat). Tabloul devine:

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

- Elementul `x` poziționat pe ultima poziție a tabloului se numește **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.c, d și e].

```
----------------------------------------------------------------
/*cautare liniara tehnica fanionului (while)*/ /*1.4.2.1.c*/
    tip_element a[nr_elemente+1];
    tip_element x;
    int i=0;
    a[nr_elemente]=x;
    while (a[i]!=x)
      i++;
    if (i == nr_elemente){
        /*în tablou nu exista elementul cautat*/
    }
    else{
        /*avem o coincidenta la indicele i*/
    }
----------------------------------------------------------------
```

- În secvența [1.4.2.1.d] apare același algoritm implementat cu ajutorul ciclului **`(do)`**.

```
---------------------------------------------------------------
/*cautare liniara tehnica fanionului (do) */ /*1.4.2.1.d*/
    tip_element a[nr_elemente+1];
    tip_element x;
    int i=-1;
    a[nr_elemente]=x;
    do{
       i++;
    }while (a[i]!=x)

    if (i == NumarMaxElem){
        /*în tablou nu exista elementul cautat*/
    }
    else{
        /*avem o coincidenta la indicele i*/
    }
-----------------------------------------------------------
```

- **Performanța** căutării liniare este O*(n*/2*)*, cu alte cuvinte, în medie un element este găsit după parcurgerea a jumătate din elementele tabloului.
- În secvența următoare apare un alt **exemplu** de implementare în limbajul C a tehnicii de căutare discutate.

```
---------------------------------------------------------------
//cautare liniara tehnica fanionului
//returneaza n pentru negasit, respectiv i<n pentru gasit // pe
pozitia i

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

1.4.2.2. Căutarea binară

- Procesul de căutare poate fi mult accelerat dacă se dispune de informații 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 dicționar).
- În continuare se prezintă un algoritm de căutare într-o **structură de date ordonată**, adică care satisface condiția:

```
------------------------------------------------------------
    tip_element a[nr_elemente];
      Ak:1 < k <= n: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 este egal cu `x` căutarea se termină;
  - Dacă este mai mic decât `x`, se restrânge intervalul de căutare la elementele care au indici mai mari ca `m`;
  - Dacă este mai mare decât `x`, se restrânge intervalul la elementele care au indicii mai mici ca `m`.
- În consecință rezultă algoritmul denumit **căutare binară**.
- Variabilele `s` şi `d` sunt de tip indice şi ele precizează limitele stânga respectiv dreapta ale intervalului în care elementul ar mai putea fi găsit:

```
------------------------------------------------------------
/*cautare binara */                          /*1.4.2.2.b*/
      tip_element a[n];
      tip_element x;
      int s,d,m;
      int gasit;
      s=0; d=n-1; gasit=0;
      while((s<=d)&&(!gasit))
      {
        m=(s+d)/2;    /*sau orice valoare cuprinsa intre s si d*/
        if(a[m]==x)
            gasit=1;
          else
            if(a[m]<x)
                s=m+1;
              else
                d=m-1;
      }
    if(gasit) /*avem o coincidenta la indicele m*/
----------------------------------------------------------------
```

- **Soluția 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 consecință rezultă că **numărul maxim** de paşi de căutare va fi $O(log_2 n)$,
  - Această performanță reprezintă o îmbunătățire remarcabilă față de căutarea liniară unde numărul mediu de căutări este $n/2$.
- Eficiența căutării binare poate fi îmbunătățită dacă se interschimbă instrucțiunile `IF` între ele.
- O îmbunătățire a eficienței se poate obține - ca şi în cazul căutării liniare - prin **simplificarea condiției de terminare**.

    - Acest lucru se poate realiza 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, însă la o examinare mai atentă, se poate observa că câştigul în eficienţă la fiecare pas este mai substanţial decât pierderea provocată de câteva comparaţii suplimentare de elemente.
    - Se reaminteşte faptul că numărul de paşi este $log_2 n$.
- Cu alte cuvinte, 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.c].

```
------------------------------------------------------------
/*cautare binară ameliorată  */              /*1.4.2.2.c*/
      tip_element a[n];
      tip_element x;
      int s,d,m;
      s=0; d=n;
      while(s<d){
           m=(s+d)/2;
         if(a[m]<x)
           s=m+1;
         else
           d=m;
       }
       if(d>n) /*nu exista elementul cautat*/;
       if(d<=n)
         if(a[d]==x)
           /*elementul exista pe pozitia d*/;
         else
           /*elementul nu exista*/;
------------------------------------------------------------
```

- Ciclul se termină când `s = d`.
    - Cu toate acestea egalitatea `s = d` nu indică automat găsirea elementului căutat.
    - Dacă `d > n` **nu** există nici o coincidenţă. S-a căutat un element mai mare decât orice element al tabloului
    - Dacă `d<=n` se observă că elementul `a[d]` corespunzător ultimului indice `d` **nu** a fost comparat cu cheia,
    - În consecinţă este necesară efectuarea în afara ciclului a unui test `a[d]=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.
- Alte variante de implementare a celor două căutări binare în limbajul C apar în secvenţa [1.4.2.2.d].

```
------------------------------------------------------------
/*cautari binare - varianta C */    /*1.4.2.2.d*/
int cautare_binara(int x,int a[],int n){
 int s=0,d=n-1,m;
 do{
  (x>a[m=(s+d)/2])?(s=m+1):(d=m-1);
 }while(a[m]-x && s<=d);
 return(a[m]-x)?n:m;
}
```

```
int cautare_binara_ameliorata(int x,int a[],int n){
 int s=0,d=n,m;
 do{
  (x>a[m=(s+d)/2])?(s=m+1):(d=m);
 }while(s<d);
 return
 ((d>=n)||(d<n && a[d]-x))?n:d;
}
------------------------------------------------------------
```

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.**
- Setul 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**;
- Termenul care descrie o **dată compusă** de această natură este cuvântul **struct**
- În secvenţele [1.4.3.a.] şi [1.4.3.b.] se prezintă modul generic de definire a unuei structuri **struct**

```
----------------------------------------------------------------
//Definire structura
struct nume_structura{
      tip₁ lista_nume_câmp₁;
      tip₂ lista_nume_câmp₂;                    [1.4.3.b]
          ...
      tipₙ lista_nume_câmpₙ;
} lista_var_structura;
------------------------------------------------------------
```

- **Identificatorii** `nume_câmp`$_1$, `nume_câmp`$_2$, ... `nume_câmp`$_n$ introduşi la definirea structrii, sunt **nume** conferite componentelor individuale ale acesteia.
- Ei sunt utilizaţi în **selectorii de articol** care permit accesul la câmpurile unei variabile structurate de tip articol.
- Pentru o variabilă `struct nume_structura x`, cel de-al i-lea câmp poate fi precizat prin notaţia "**dot**" sau "**punct**":

  `x.nume_câmp`$_i$

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

  `x.nume_câmp`$_i$`=` x_i

  unde x_i este o valoare (expresie) de `tip`$_i$.
- Câteva exemple de definire a unor structuri articol.

```
-----------------------------------------------------------
/*Exemplu 1 de definire a unor structuri articol*/
struct data {
      int zi, luna, an;
      } data_nasterii, data_inscrierii
```

```
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
----------------------------------------------------------------
/*Exemplu 2 de definire a unor structuri articol*/

struct punct {
  int x;
  int y;
}
struct dreptunghi {
  struct punct varf1;
  struct punct varf2;
}                                               [1.4.3.g]
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 secvență. Tipul de date abstract secvență

- Caracteristica comună a structurilor de date prezentate până în prezent (tablouri, articole, mulțimi) 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: secvențele, listele, arborii, grafurile, etc, sunt caracterizate prin **cardinalitate infinită**.
    - Această diferență față de structurile clasice este de profundă importanță având consecințe practice semnificative.
- Spre exemplu **structura secvență** având tipul de bază T_0 se defineşte după cum urmează:

S_0=< >   (secvența vidă)

S_i=<S_{i-1},s_i>   unde $0<i$ şi $s_i \in T_0$

- Cu alte cuvinte, o secvență cu tipul de bază T_0, este:
    - Fie o secvență **vidă**

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

- Definirea **recursivă** a unui tip secvență conduce la o **cardinalitate infinită**.
  - Fiecare valoare a tipului secvență conține în realitate un număr **finit** de componente de tip T_0,
  - Acest număr este **nemărginit** deoarece, pornind de la orice secvență se poate construi o secvență mai lungă.
- O consecință importantă a acestui fapt este:
  - Volumul de memorie necesar reprezentării unei structuri avansate, **nu** poate fi cunoscut în momentul compilării,
  - Ca atare, 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 cerință, este necesar ca limbajul superior utilizat în implementare să fie prevăzut cu acces la funcții sistem care permit:
  - **Alocarea / eliberarea dinamică** a memoriei
  - **Legarea / referirea dinamică** a componentelor
- În aceste condiții cu ajutorul instrucțiilor 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 secvență** este din acest punct de vedere o structură **intermediară**;
  - (1) Ea este o **structură avansată** din punctul de vedere al cardinalității care este infinită
  - (2) Este însă atât de curent utilizată încât includerea sa în mulțimea **structurilor fundamentale** este consacrată.
  - (3) Această situație este influențată şi de faptul că alegerea unui **set potrivit de operatori** referitori la structura secvență, permite implementatorilor să adopte reprezentări potrivite şi eficiente ale acestei structuri.
  - (4) Drept consecință, **mecanismele alocării dinamice** a memoriei devin suficient de simple pentru a permite o implementare eficientă, neafectată de detalii, la nivel de limbaj superior.

1.4.4.1. Tipul de date abstract secvență

- Includerea structurii secvență în rândul structurilor fundamentale, presupune **constrângerea setului de operatori** de o asemenea manieră încât se permite numai **accesul secvențial** la componentele structurii.
- Această structură este cunoscută şi sub denumirea de **fişier secvențial** sau pe scurt **fişier**.
- **Structura secvență** este supusă **unicei restricții** de a avea componente de acelaşi tip, cu alte cuvinte este o structură **omogenă**.
- Numărul componentelor, denumit şi **lungime** a secvenței, se presupune a fi **necunoscut** atât în faza de compilare, cât şi în faza de execuție a codului.
  - Mai mult chiar, acest număr **nu** se presupune constant el putându-se modifica în timpul execuției.
- Cardinalitatea tipului secvență este în consecință **infinită**.

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

- Acest lucru face posibilă tratarea relativ simplă a structurilor de tip secvență şi permite încadrarea lor în rândul structurilor fundamentale, deşi de drept ele aparțin structurilor dinamice.
- Secvența este o structură **ordonată**.
    - Ordonarea elementelor structurii este stabilită de ordinea în timp a creării componentelor individuale.
- Datorită mecanismului de acces secvențial preconizat, selectarea prin indici a componentelor individuale devine improprie.
- În consecință la definirea unui tip de secvență se precizează numai **tipul de bază**.

```
        TYPE TipSecvență = FILE OF TipDeBază;
```

- În cadrul unei secvențe definite ca mai sus în orice moment este **accesibilă o singură componentă**, denumită **componentă curentă**, care este precizată printr-un **pointer** (indicator) asociat secvenței.
    - Acest indicator avansează secvențial, practic după execuția oricărei operații asupra secvenței.
- În plus oricărei secvențe i se asociază un operator boolean standard **sfârşit de fişier** notat cu `Eof(f:TipSecventa)` care permite sesizarea sfârşitului secvenței.
- Modul concret de acces la componente, precum şi posibilitățile de prelucrare efectivă, rezultă din semantica operatorilor definiți pentru tipul de date abstract secvență este prezentat sintetic în [1.4.5.1.a].
- Implementările recente ale limbajelor Pascal şi C introduc în setul de instrucțiuni accesibile programatorului şi operatori pentru prelucrarea secvențelor implementate ca şi fişiere secvențiale.
- În secvența [1.4.5.1.b] apare o astfel de implementare definită în limbajul Pascal. Variante asemănătoare stau la dispoziție şi în limbajul C.

```
------------------------------------------------------------
TDA Secvență

Modelul matematic: secvență de elemente de acelaşi tip. Un
     indicator la secvență indică elementul următor la care se
      poate realiza accesul. Accesul la elemente este strict
     secvențial.
Notații:
      TipElement - tipul unui element al secvenței. Nu poate
        fi de tip secvență;
      f - variabilă secvență;
      e - variabilă de TipElement;            [1.4.5.1.a]
      b - valoare booleană;
      numeFisierDisc - şir de caractere.
Operatori:
     Atribuie(f,numeFisierDisc) - atribuie variabilei secvență f
          numele unui fişier disc precizat;
     Rescrie(f) - procedură care mută indicatorul secvenței la
          începutul lui f şi  deschide fişierul f în regim de
          scriere. Dacă fişierul f nu există el este creat. Daca
          f există, vechea sa variantă se pierde şi se creează un
          nou fişier f vid;
```

```
ResetSecvență(f) - procedură care mută indicatorul la
     începutul secvenței f şi deschide secvența în regim de
     consultare. Dacă f nu există se semnalează eroare de
     execuție;
DeschideSecvență(f) - procedură care în anumite
     implementări joacă rol de rescrie sau reset de
     secvență;
b:= Eof(f) - funcție care returnează valoarea true dacă
     indicatorul secvenței indică marcherul de sfârşit de
     fişier al lui f;
FurnizeazăSecvență(f,e) - procedură care acționează în
     regim de consultare. Atâta vreme cât Eof(f) este fals,
     furnizează în e următorul element al secvenței f şi
     avansează indicatorul acesteia;
DepuneSecvență(f,e) -  pentru secvența f deschisă în regim
     de scriere, procedura copiază valoarea lui e în elementul
     următor al secvenței f şi avansează indicatorul
     acesteia;
 Adaugă(f) - procedură care deschide secvența f în regim de
   scriere, poziționând indicatorul la sfârşitul acesteia,
   cu posibilitatea de a adăuga elemente noi numai la
   sfârşitul secvenței.
 InchideSecvența(f) - închide secvența.
```

```
------------------------------------------------------------
{Implementarea tipului secvență - Varianta Pascal}

TYPE TipElement = ... ;
     TipSecvență = TEXT;

VAR  f: TipSecvență;                        [1.4.5.1.b]
     e: TipElement;
     numeFisierDisc: string;
{Atribuie(f,numeFisierDisc)}   assign(f,numeFisierDisc)
{Rescrie(f)}                   rewrite(f)
{DepuneSecvență(f,e)}          write(f,e) ; writeln(...)
{ResetSecvență(f)}             reset(f)
{Eof(f)}                       eof(f)
{FurnizeazăSecvență(f,e)}      read(f,e);  readln(...)
{Adaugă(f)}                    append(f)
{InchideSecvență(f)}           close(f)
------------------------------------------------------------
```

1.5 Rezumat

- Pentru utilizarea sistemelor de calcul se folosesc **datele**. Datele sunt o **abstractizare** a lumii reale. Drept suport al abstractizării se folosesc **limbajele de programare**.
- Datele se încadrează în **tipuri de date** care precizează domeniul valorilor, operațiile şi nivelul de structurare al entităților care sunt încadrate în tipul respectiv.
- Un **tip de date abstract** (TDA) este conceput ca un **model matematic** căruia i se asociază un **set de operatori** specifici. Pentru a fi utilizat un TDA trebuie **definit**, apoi **implementat** într-un limbaj de programare şi **instanțiat** în cadrul aplicației
- **Tipurile de date** se clasifica în
  - **Tipuri de date primitive** (nestructurate) care includ tipurile **standard** sau **predefinite** şi **tipurile primitive de date definite de utilizator**
  - **Tipuri de date structurate** care includ **tipurile de date statice** (tablou, articol, secvență) şi **tipurile de date dinamice** (listă, arbore, graf)

- **Tipurile structurate** se construiesc din cele **nestructurate** utilizînd **metode de structurare** definite în limbajele de programare.
- Un **TDA** definit, prin implementare devine **tip de date** (TD) care prin instanțiere devine **dată elementară** DE (dacă este nestructurat) respectiv o **structura de date** SD (dacă este un tip structurat)
- Tipurile de date primitive includ tipurile: **enumerare, întreg, real, boolean, caracter**
- Tipurile structurate statice includ tipurile: **tablou, articol şi secvență**
- Una dintre operațiile cel mai frecvente aplicate asupra structurii tablou este **căutarea**.
  - Pentru tablourile obişnite se utilizează **căutarea liniară** respectiv varianta de căutare numită **tehnica fanionului**
  - Pentru tablourile ordonate se foloseşte o metodă mult mai performantă: **căutarea binară**.

1.6 Exerciții

1. Definiți conceptele de *dată*, *abstractizare* şi *limbaj de programare*. Evidențiați legătura dintre ele.
2. Ce este un *tip de date*? Prin ce se caracterizează un tip de date? Care sunt *caracteristicile* unui tip de date?
3. Definiți conceptul de *tip de date abstract* (TDA). Care sunt *avantajele* utilizării TDA-urilor?
4. Cum se *defineşte* un TDA? Cum se *implementează* un TDA?
5. Cum se *clasifică* tipurile de date?
6. Care este diferența între următoarele entități: *tip de date abstract, tip de date, structuraă de date şi dată elementară*?
7. Descrieți *tipurile de date nestructurate* (enumerare, întreg, real, caracter). Ce au în comun aceste tipuri?
8. Ce este un *tablou*? Cum se defineşte *structura tablou*? Dați exemple de definire a unor *tablouri liniare* şi a unor *tablouri de mai multe dimensiuni*
9. Se cere să se redacteze două funcții: una care implementează *cautarea liniară*, celaltă *căutarea cu tehnica fanionului* într-un tablou liniar.
10. Se cere să se redacteze două funcții: una care implementează *cautarea binară*, celaltă *căutarea binară ameliorată* într-un tablou liniar ordonat.
11. Se cere să se redacteze un *program C* care:
    a) defineşte un tablou de numere întregi
    b) citeşte elementele tabloului de la tastatură
    c) citeşte o cheie x
    d) caută cheia x în tablou prin diferite tehnici de căutare, utilizând funcțiile din aplicațiile anterioare
    e) afişează rezultatul căutării
12. Ce este o *structură*? Cum se *defineşte* o structură? Cum se *accesează componentele* unei structuri? Dați exemple de structuri.
13. Se cere să redacteze un program C care implementează operațiile matematice cu numere complexe: adunarea, scăderea şi modulul. La definirea numerelor complexe se va utiliza un tip structurat. Se va defini de asemenea o funcție pentru citirea şi o funcție pentru afişarea unui număr complex. Programul principal va exemplifica operațiile implementate.
14. Ce este şi cum se defineşte *structura secvență*? Care sunt principalele *caracteristici* ale structurii secvență?