

1. Structuri de date fundamentale

1.1. Introducere

- Sistemele de calcul moderne sunt dispozitive care au fost concepute cu scopul de a facilita și accelera **calcule complicate**, mari consumatoare de timp.
 - Din acest motiv, în majoritatea aplicațiilor, **viteza, frecvența de lucru, capacitatea de a memora, posibilitatea de a avea acces la cantități mari de informații și capabilitățile de conectivitate**, au un rol hotărâtor și sunt considerate caracteristici primordiale ale calculatoarelor.
 - **Abilitățile de calcul**, în cele mai multe cazuri sunt irelevante în manieră directă.
- Cantitatea mare de informații pe care o prelucrează un sistem de calcul reprezintă de regulă, o **abstractizare** a lumii înconjurătoare, respectiv a unei părți a lumii reale.
 - Informația furnizată sistemului de calcul constă dintr-o mulțime de **date selectate** de către utilizator care **modelează** lumea reală.
 - **Datele selectate** se referă la **mulțimea de date** care este considerată cea mai reprezentativă pentru **problema tratată** și despre care se presupune că din ea pot fi obținute (deduse sau calculate) **rezultatele dorite**.
- În acest context, **datele** reprezintă o **abstractizare a realității**.
- Ce este o **abstractizare**?
 - O **abstractizare** este de fapt o **simplificare** a faptelor.
 - Anumite **proprietăți și caracteristici** ale obiectelor reale, sunt **ignore** întrucât pot fi considerate **periferice și nerelevante** pentru problema particulară tratată.
 - Dacă nu s-ar ține cont de abstractizare și s-ar lua în considerare toate caracteristicile și proprietățile problemei tratate, aceasta ar deveni extrem de complexă, practic inabordabilă.
- În **rezolvarea unei probleme** cu ajutorul unui **sistem de calcul** un rol esențial revine alegerii unei **abstractizări convenabile** a realității.
 - Acest lucru se poate realiza în doi pași:
 - (1) Definirea unui **set reprezentativ de date** care **modelează** (reprezintă) situația reală.

- (2) Stabilirea **reprezentării abstractizării**.
- De cele mai multe ori cei doi pași 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 facilitățile oferite de un anumit sistem de calcul.
- **Alegerea reprezentării** este de multe ori o activitate dificilă întrucât **nu** este determinată în mod unic de facilitățile disponibile.
 - Se cunoaște că reprezentarea internă a informației într-un sistem de calcul se realizează cu ajutorul **cifrelor binare** (biți).
 - Această reprezentare care este foarte potrivită pentru **circuitele electronice**, **nu** este potrivită pentru **ființa umană** din cauza numărului prea mare de cifre implicate.
- Datorită evoluției istorice, azi alegerea reprezentării datelor se poate realiza la **diferite niveluri de abstractizare**, funcție de obiectivul urmărit și 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 înțeleagă **fraze** construite cu ajutorul **termenilor** definiți în cadrul acestui limbaj de programare.
 - **Termenii** definiți în cadrul limbajului de programare, pot să încorporeze un anumit **nivel de abstractizare** al entităților efectiv utilizate (definite) de către mașina reală.
- Utilizând un astfel de limbaj, programatorul va fi eliberat spre exemplu de chestiuni legate de reprezentarea numerelor dacă acestea constituie entități 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 fiabilitate (fidelitate)** al programelor care rezultă.
 - Este mult mai ușor să se conceapă un program bazat pe obiecte familiare (numere, mulțimi, secvențe) decât unul bazat pe structuri de biți, cuvinte și salturi.
- Desigur, oricare sistem de calcul reprezintă în ultimă instanță datele ca și un masiv de informații binare.
 - Acest fapt este însă **transparent** pentru programator.
- Programatorul poate opera cu **noțiuni abstracte**, mai ușor de manipulat întrucât dispune de **compilatoare** sau **translatoare specializate** care preiau sarcina **transformării** noțiunilor abstracte în termeni specifici sistemului de calcul țintă.
- Cu cât **nivelul de abstractizare** este mai strâns legat de un anumit sistem de calcul:

- (1) Cu atât este mai **simplu** pentru dezvoltatorul de aplicații să aleagă o **reprezentare** mai eficientă a datelor.
- (2) Cu atât mai **redușă** este posibilitatea ca reprezentarea aleasă să satisfacă o **clasă mai largă** de aplicații 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 mașină** sau limbaje dependente de mașină 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 aplicații.
- Cu alte cuvinte, **procesul de abstractizare și 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 distincție clară între:
 - (1) Numerele reale;
 - (2) Numerele complexe;
 - (3) Valorile logice;
 - (4) Variabilele care reprezintă valori individuale, mulțimi de valori, mulțimi de mulțimi sau funcții, mulțimi de funcții, etc.
- În acest sens se statuează **principiul** conform căruia fiecare **constantă**, **variabilă**, **expresie** sau **funcție** este încadrată într-un anumit **tip**.
- Un **tip** este în mod esențial caracterizat prin:
 - (1) **Mulțimea valorilor** căreia îi aparține o constantă a tipului în cauză, respectiv mulțimea valorilor pe care le poate asuma o variabilă, o expresie sau care pot fi generate de o funcție încadrată în acel tip.
 - (2) Un anumit **grad de structurare** (organizare) a informației;

- (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 **funcții** este precizat printr-o **declarație explicită de tip** referitoare la constanta, variabila sau funcția respectivă, declarație care precede textual utilizarea constantei, variabilei sau funcției în cauză.
 - Această metodă simplifică activitatea compilatorului permițând verificarea automată a legitimității utilizării entităților asociate tipului respectiv.
- **Caracteristicile** conceptului de **tip de date** sunt următoarele:
 - (1) Un **tip de date** determină **mulțimea valorilor** căreia îi aparține o constantă, sau pe care le poate asuma o variabilă sau o expresie, sau care pot fi generate de un operator sau o funcție, care sunt încadrate în tipul respectiv.
 - (2) **Tipul** unei valori precizate de o constantă, variabilă sau expresie poate fi **dedus** din **forma** sau din **declarația** sa, fără a fi necesară execuția unor procese de calcul.
 - (3) Fiecare **operator** sau **funcție** asociată tipului, acceptă **argumente** de un tip precizat și 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 informației.
- Drept urmare, respectând aceste reguli, un **compilator** poate verifica compatibilitatea și legalitatea anumitor construcții de limbaj, în **faza de compilare**, fără a fi necesară execuția efectivă a programului.
 - Această **redundanță controlată** a textului programului este extrem de folositoare în dezvoltarea de programe și este considerată 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 biți** fără vreo structură aparentă.
 - Ori tocmai **structurile abstracte de date** sunt acelea care permit recunoașterea, interpretarea și prelucrarea configurațiilor monotone de cifre binare prezente în memoria unui sistem de calcul.

1.2.2. Tipuri de date abstracte

- Definirea noțiunii de **tip de date abstract (TDA)**:
 - Un **TDA** se definește ca un **model matematic** căruia i se asociază o **colecție de operatori specifici**.
- Vom realiza o paralelă cu conceptul de **funcție (procedură)**.
 - (1) Funcția (procedura) **generalizează** noțiunea de **operator**.
 - De regulă, un programator care utilizează un limbaj de programare este constrâns la utilizarea exclusivă a operatorilor definiți în cadrul limbajului de programare ("built-in" operators).
 - Folosind procedurile sau funcțiile, programatorul este liber să-și **definească proprii săi operatori**, pe care ulterior să-i aplice asupra unor operanzi care nu e necesar să aparțină tipurilor de bază (primitive) ale limbajului utilizat.
 - Cu alte cuvinte, utilizând **funcțiile** sau **procedurile**, operatorul poate **generaliza** anumiți **operatori specifici** aplicațiilor proprii.
 - Un exemplu de funcție utilizată în această manieră este rutina de înmulțire a două matrici.
 - (2) Funcțiile (procedurile) **încapsulează** anumite părți ale unui algoritm prin "**localizare**".
 - Aceasta înseamnă amplasarea într-o singură secțiune a programului a tuturor instrucțiunilor relevante.
- Într-o manieră similară:
 - (1) **Un TDA generalizează tipurile de date primitive** (întreg, real, etc.), după cum procedurile sunt generalizări ale operațiilor primitive (+, -, etc.).
 - (2) **Un TDA încapsulează** conceptual un tip de date în sensul că din punct de vedere logic și fizic, definirea tipului și toate operațiile referitoare la el sunt localizate într-o singură secțiune (zonă) a programului.
 - Dacă apare necesitatea **modificării implementării** TDA-ului:
 - (2.1) Programatorul știe cu certitudine locul (unic) în care trebuie efectuate modificările.
 - (2.2) Poate fi sigur că modificarea secțiunii respective **nu** produce modificări nedorite în restul codului programului, dacă nu se modifică formatul operatorilor TDA-ului respectiv.
 - (3) În plus, în afara secțiunii în care sunt definiți 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 asociați nefiind în nici un fel preocupat de implementarea acestora.

- O problemă care poate să apară este cea legată de faptul că anumite operații pot să se refere la mai multe TDA-uri, caz în care accesarea acestor operații trebuie realizată în mod specific în funcție 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 cerință **nu** este verificabilă la nivelul **compilerului** și trebuie acceptată ca și o **constrângere impusă** a disciplinei de programare.
-

• **Exemplul 1.2.2.**

- Se consideră un tip de date abstract **Listă**.
 - Nodurile listei aparțin unui tip precizat **TipNod**.
 - Fie:
 - L o instanță a TDA-ului - (*Lista L*),
 - v o variabilă nod - (*TipNod v*),
 - c o referință la un nod al listei - (*TipReferintaNodLista c*).
 - Un posibil **set de operatori** pentru **TDA Listă** este următorul:
 1. **ListăVidă**(*Lista L*); - operator care inițializează pe L ca listă vidă;
 2. **c = Primul**(*Lista L*); - operator care returnează valoarea indicatorului la primul nod al listei, sau indicatorul vid în cazul unei liste goale;
 3. **c = Următor**(*Lista L*); - operator care returnează valoarea indicatorului următorului nod al listei, respectiv indicatorul vid dacă un astfel de nod nu există;
 4. **Inserează**(*TipNod v, Lista L*); - operator care inserează nodul v în lista L după nodul curent.
 5. **Furnizează**(*TipNod v, Lista L*); - operator care furnizează conținutul nodului curent al listei.
-

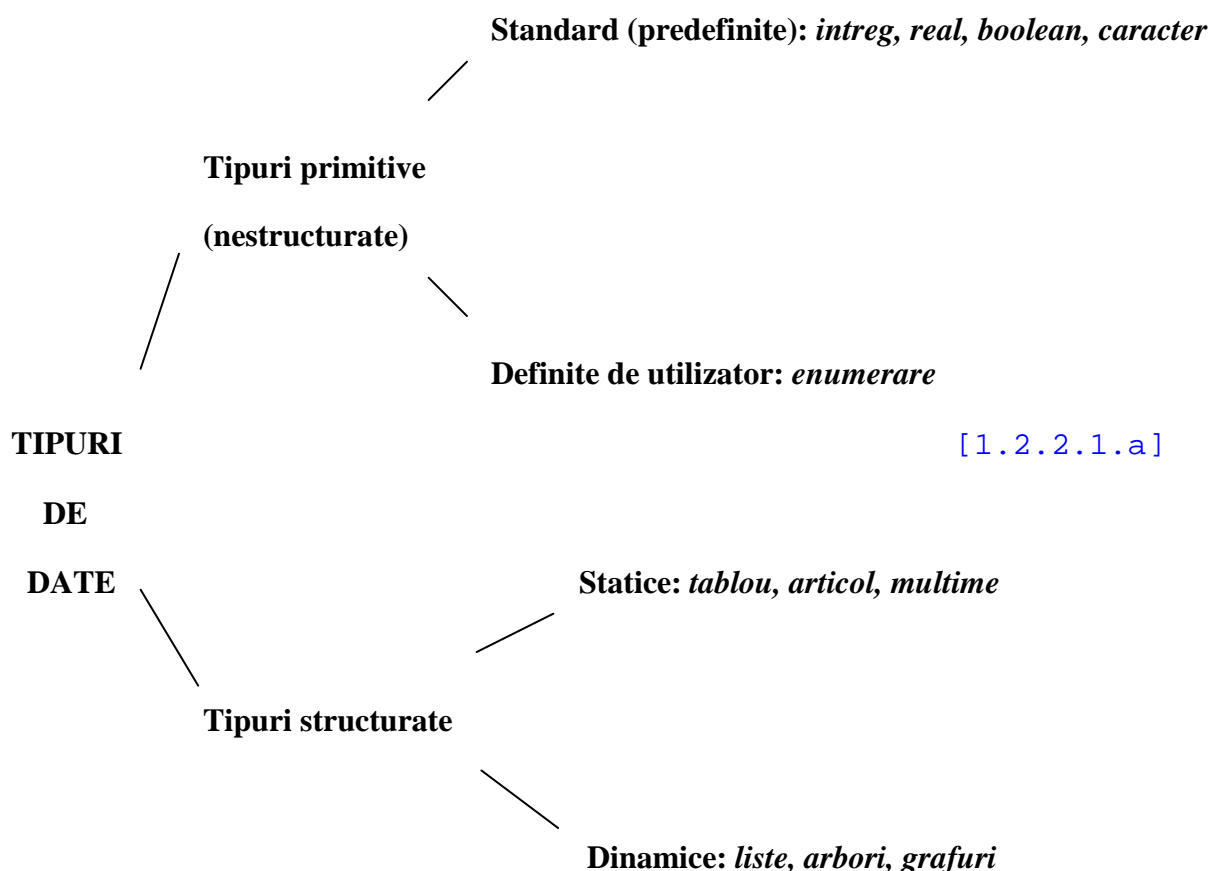
- Câteva **observații** referitoare la avantajele definirii unui astfel de TDA:
 - (1) Odată definit și implementat TDA Listă, **toate prelucrările de liste** se pot realiza în termenii operatorilor definiți asupra acestuia, declarând un număr corespunzător de instanțe ale TDA-ului, respectiv câte o instanță 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 inițial.

- Chiar în accepțiunea **modificării pe parcurs a implementării**, spre exemplu din motive de eficiență, programele utilizator rămân nemodificate, singurele modificări fiind suportate de secțiunea care definește TDA-ul.
- (3) Nu există nici o **limitare** referitoare la numărul de operații care pot fi aplicate **instanțelor** unui model matematic precizat.
 - Se face însă sublinierea că fiecare set de operatori definește și se referă la o instanță distinctă a unui TDA.
- (4) **Eficiența, siguranța și corectitudinea** utilizării TDA-ului. Lista **este garantată numai în situația** în care utilizatorul realizează **accesele** la listele definite **numai prin operatorii asociați** tipului respectiv, adică respectă strict restricțiile 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 și **un model matematic** căruia i se asociază o **colecție de operatori** definiți pe acest model.
- În consecință, **definirea** unui tip de date presupune:
 - (1) Precizarea **modelului matematic**.
 - (2) Definirea **operatorilor** asociați.
- Limbajele de programare includ mai multe **metode de definire** a tipurilor de date.
- În toate situațiile se pornește 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 și valorile logice.
 - Ele se bazează pe **modele matematice consacrate** (mulțimile de numere întregi, reale, logice, etc.) și implementează operațiile specifice definite pe aceste categorii de numere.
 - Dacă între valorile individuale ale tipului există o **relație de ordonare**, atunci tipul se numește **ordonat** sau **scalar**.
- (2) **Tipuri primitive definite de utilizator**.
 - O metodă larg utilizată de construcție 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 care aparțin unor tipuri constiutive. Dacă există **un singur** tip constitutiv, acesta se numește **tip de bază**.
 - Numărul valorilor distincte aparținătoare unui tip T se numește **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]):
 - **Stalice**: tabloul, articolul, (mulțimea), secvența (fișierul).
 - **Dinamice**: listele, arborii, grafurile.



-
- Pentru a **utiliza** un tip de date definit, trebuie să **declare** **variabile** încadrate în tipul în cauză, proces care se numește **instanțiere**.
 - Prelucrarea acestor **variabile** se realizează cu ajutorul **operatorilor** asociați tipului.
 - Din punctul de vedere al **naturii** lor, există mai multe **feluri de operatori**:
 - (1) **Omogeni** - pot avea drept operanzi numai instanțe ale TDA-ului în cauză.
 - (2) **Micști** - pot avea drept operanzi instanțe ale altor tipuri, de regulă instanțe ale unor alte TDA-uri compatibile.
 - (3) **Interni** - rezultatul aplicării unui astfel de operator poate fi doar o altă instanță a TDA-ului respectiv.
 - (4) **Externi** - rezultatul aplicării unui astfel de operator poate aparține unui alt tip.
 - Din punctul de vedere al **definirii**, există de asemenea mai multe **categorii de operatori**:
 - (1) **Primitivi** sau **operatori atomici** - acești operatori sunt de regulă implementați prin hardware și reflectă operații specifice cablate ca și instrucțiuni ale sistemului de calcul.
 - (2) **Definiți prin limbaj** - acești operatori includ de regulă metodele de structurare și metodele de acces la componente.
 - (3) **Definiți și implementați de utilizator**.
 - **Operatorii de bază** incluși de regulă în orice set de operatori sunt următorii:
 - **Comparația și atribuirea** - sunt operatori fundamentali definiți pentru toate tipurile de date. Execuția 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) sau selectează componente ale acestora. Din acest motiv ei sunt importanți în conexiune cu tipurile structurate. Se cunosc două categorii de operatori de transfer:
 - (1) **Operatorii constructori** - generează valorile structurate pornind de la valorile lor componente.
 - (2) **Operatorii selectori** - realizează extragerea valorilor componente dintr-o structură.
 - **Constructorii și selectorii** sunt de fapt operatori de transfer care organizează (constituie) tipurile constitutive în tipuri structurate și 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:

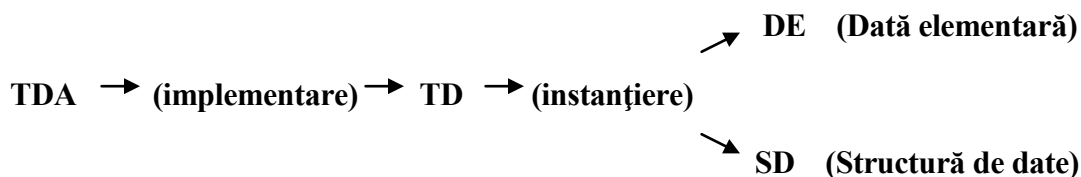
- **Impliciți** - caz în care sunt implementați direct în limbaj (spre exemplu accesul la un câmp al unui articol sau accesul la un element al unui tablou);
- **Expliciți** - în cazul în care sunt sintetizați de programator prin elemente de limbaj (spre exemplu accesul la nodurile unei liste).
- În concluzie:
 - (1) Un tip este de fapt o manieră de structurare a informației.
 - (2) Pentru tipurile nestructurate este mai relevantă mulțimea constantelor tipului și 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 traducere, 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 construiește pornind de la tipurile de date de bază care sunt definite în cadrul limbajului, utilizând **facilitățile 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-și redacteze programele în limbaje ale căror tipuri primitive de date să fie cât mai apropiate modelele și operațiile TDA-urilor proprii.
 - Din acest punct de vedere, în multe sensuri limbajele obișnuite de programare **nu** sunt suficient de înzestrate în ceea ce privește facilitățile de a crea diverse TDA-uri și nici de definire a operatorilor specifici.

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

- Care este semnificația următorilor termeni 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 elementară** (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 **implementare** a acestor TDA-uri în termenii tipurilor de date și 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 și care iau valori în mulțimea valorilor tipului.
 - Acest proces se numește **instanțiere** și el de fapt conduce la rezultate diferite:
 - Instanțierea unui tip **nestructurat** conduce la generarea unei **date elementare** (DE).
 - Instanțierea unui tip **structurat** conduce 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 și pentru cele structurate:
 - Prin *implementare* se înțelege **definirea tipului** în termenii unui limbaj de programare.
 - Prin *instanțiere*, se înțelege **declararea unei variabile** asociate tipului respectiv.
- În cazul **tipurilor predefinite** faza de implementare lipsește ea fiind inclusă intrinsec în limbajul de programare, ca atare pentru aceste tipuri este valabilă formula:

TD → (instanțiere) → 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 și 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 numește **Tip enumerare** și el este definit prin enumerarea tuturor valorilor sale posibile: c_1, c_2, \dots, c_n .

// Definirea tipului enumerare - Variante

```
enum tip_enumerare {c1,c2,c3,...,cn} variabila;
```

```
typedef enum {c1,c2,c3,...,cn} nume_tip;           /*1.3.1.a*/
```

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

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

- Cardinalitatea tipului este **card** (tip_enumerare) = n .
- Tipul enumerare este un **tip nestructurat definit de utilizator**
 - În consecință utilizarea sa presupune atât faza de **implementare** (declarare a tipului) cât și faza de **instanțiere** (declarare de variabile aparținătoare tipului).

// Exemplu de implementare a tipului enumerare

```
typedef enum {dreptunghi,patrat,cerc,elipsa} tip_figura;
```

```

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

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

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

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

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

typedef enum {secventa,tablou,articol,multime} tip_structura;

```

- La **definirea** unui astfel de tip se introduce **nu** numai un nou **identificator de tip**, dar se definesc și **identificatorii** care precizează valorile noului tip (de fapt se precizează explicit domeniul valorilor tipului).
- Acești identificatori pot fi utilizați ca și constante în cadrul programului, sporind în acest fel considerabil gradul de lizibilitate și de înțelegere al textului sursă.

```

/* Exemplu de instanțiere a tipului enumerare */

```

```

tip_culoare c;                                     /*1.3.1.c */
tip_zi_saptamina z;
tip_grad g;
tip_boolean b;

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

```

- Astfel, referitor la secvența [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 și că în calitate de constante le sunt asociate numere întregi în ordinea enumerării lor.
- Drept urmare un compilator poate verifica ușor utilizarea unor operatori nespecifici pentru astfel de tipuri nenumerice, ca de exemplu:

```

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

```

- Tipul enumerare include implicit ca și operatori **comparația** și **atribuirea**.

- Deoarece, tipul enumerare este un **tip ordonat**, în setul său de operatori pot fi definiți operatorii care generează succesorul respectiv predecesorul unui argument precizat.
 - Spre exemplu în limbajul PASCAL, operatorii definiți în acest scop sunt **SUCC(x)** și **PRED(x)**.

1.3.2. Tipuri standard predefinite

- **Tipurile standard predefinite** sunt acele tipuri care sunt disponibile în marea majoritate a sistemelor de calcul ca și caracteristici hardware.
- **Tipurile standard predefinite** includ **numerele întregi, valorile logice, caracterele și 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 și depinde de caracteristicile hardware ale sistemului.
 - Se presupune că toate operațiile efectuate asupra acestui tip conduc exclusiv la valori exacte și se conformează regulilor obișnuite ale aritmeticii.
 - Procesul de calcul se întrerupe în cazul obținerii unui rezultat în afara setului reprezentabil (depășirea capacității registrelor - DCR).
 - Pe mulțimea numerelor întregi în afara operatorilor clasici de **comparare** și **atribuire**, se definesc și **operatori standard**:
 - Operatori standard definiți pe mulțimea întregilor: **adunare (+)**, **scădere (-)**, **înmulțire (*)**, **împărțire întreagă (/)** și **modulo (%)**.

$$m - n < (m / n) * n \leq m$$

$$(m / n) * n + (m \% n) = m$$
 - Implementările curente ale limbajelor de programare conțin mai multe categorii de tipuri întregi (**int**, **long int**, **short int**, **unsigned int**, etc.) care diferă de regulă prin numărul de cifre binare utilizate în reprezentare.
- În secvența [1.3.2.1.a] este prezentat în manieră sintetică **TDA Întreg**.

TDA Întreg

Modelul matematic: elemente scalare cu valori în mulțimea numerelor întregi: $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$.

Notații:

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

Operații:

```

AtribuireÎntregi(i,e) - memorează valoarea lui e în
    variabila i (procedură);
k= AdunareÎntregi(i,j)      (funcție);
k= ScădereÎntregi(i,j)      (funcție);
k= ÎnmulțireÎntregi(i,j)     (funcție);
k= ÎmpărțireÎntregi(i,inz)  (funcție);
inn= Modulo(i,inz) - returnează întregul pozitiv care
    reprezintă restul împărțirii lui i cu inz (funcție);
b= EgalZero(i)              (funcție);
b= MaiMareCaZero(i)         (funcție).

```

- (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.
 - 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.
 - Operațiile se notează cu simbolurile consacrate cu excepția împărțirii numerelor reale care se notează cu (/).
 - Operațiile care conduc la valori care depășesc domeniul de reprezentabilitate al implementării conduc la erori (DCR).
 - Implementările curente ale limbajelor de programare conțin mai multe categorii de 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) și **FALSE** (fals).
 - În limbajul C aceste valori lipsesc fiind substituite de valori întregi:
 - 1 sau <>0 semnifică **adevărat**.
 - 0 semnifică **fals**.
 - Operatorii specifici definiți pentru acest tip sunt operatorii logici: conjuncție, reuniune și negație.

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 mulțime de caractere afișabile.

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

Litere mari:	A	B	C	...	X	Y	Z
binar :	01000001	01000010	01000011		01011000	01011001	01011010
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		
binar:	00110000	00110001	00110010		00111001		
hexazecimal:	x'30'	x'31'	x'32'		x'39'		
zecimal:	48	49	50		57		
Litere mici:	a	b	c	...	z		
binar :	01100001	01100010	01100011		01111010		
hexazecimal:	x'61'	x'62'	x'63'		x'7A'		
zecimal:	97	98	99		122		
Caracterul blanc:	binar: 00100000		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` și `int` sunt echivalente, deși au interpretări diferite.
- Se pot defini două **funcții standard de transfer** între tipurile `char` și `integer`:
 - `f('5')=5` transformă un caracter cifră în întregul corespunzător.
 - `g(8)='8'` transformă un întreg în caracterul cifră corespunzător.
 - `f` și `g` sunt funcții inverse:

$$f(g(i))=i \quad (0 < i < 9)$$

$$g(f(c))=c \quad ('0' < c < '9')$$

// Implementarea funcțiilor de transfer întreg-caracter

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



```

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

typedef float tip_tablou[100];

tip_tablou 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 instanță) a unui tip de date tip_tablou având componentele  $c_1, c_2, \dots, c_n$  poate fi inițializată printr-un constructor de tablou și 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} };           /*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 execuției programului, conduce la determinarea componentei selectate.
- Dacă tipul de bază al unui tablou este **ordonat**, atunci în cadrul tipului tablou poate fi definită o relație de **ordonare naturală**.
- **Ordonarea naturală** a două tablouri de același tip este determinată de către acele componente corespunzătoare care sunt diferite și care au cel mai mic indice:

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

$$'CAPAC' < 'COPAC'$$

```

-----
/* Ordonarea naturală a două tablouri (<, =, >) */

typedef tip_element tip_tablou[n];

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

```

```

(x<y)  <=>  Ek:0≤k<n: ((Ai:0≤i<k: x[i]=y[i])&(x[k]<y[k]))
(x=y)  <=>  Ek:0≤k<n: (Ai:0≤i<k: x[i]=y[i])
(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 `tip_tablou` aparțin aceluiași tip de bază `tip_element` avem:

$$\text{Card}(\text{tip_tablou}) = \text{Card}(\text{tip_element})^{\text{Card}(\text{tip_indice})}.$$

- Tablourile de o singură dimensiune se mai numesc și **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 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 tip_element tip_tablou[nr_elemente];`

`typedef tip_tablou tip_matrice[nr_elemente1];` */*1.4.1.f*/*

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

/ Definirea unei matrici */*

`tip_element nume_matrice[nr_elemente][nr_elemente1];` */*1.4.1.g*/*

/ Definirea unui tip tablou multidimensional (matrice) */*

`typedef tip_element tip_tablou[nr_elemente1][nr_elemente2]...
[nr_elementeN];`

`tip_tablou matrice;` */*1.4.1.h*/*

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

TDA Tablou

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

Notatii:

tip_element -tipul elementelor tabloului;
a - tablou unidimensional;
i - index; [1.4.1.i]
e - obiect de tip *tip_element*.

Operații:

DepuneInTablou(*a,i,e*) - operator procedură care depune valoarea lui *e* în cel de-al *i*-lea element al tabloului *a*;
e= **FurnizeazăDinTablou**(*a,i*) - operator funcție care returnează valoarea celui de-al *i*-lea element al tabloului *a*.

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

```
#define numar_max_elemente valoare_intreaga
typedef tip_element tip_tablou[numar_max_elemente];

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**:

tip_element 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 relația *a[i].cheie=x* și el permite accesul și 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 numește **căutare liniară** și este ilustrată în secvența [1.4.2.1.a].

```
/*căutare liniară - varianta while */
/*1.4.2.1.a*/

tip_element a[n]; /*tabloul de căutare*/
tip_element x;     /*elementul căutat*/

int i=0; /*indice de căutare*/
while ((i<n-1) && (a[i]!=x))
    i++;
if (a[i]!=x) { /*în tablou nu există elementul căutat*/ }
else { /*avem o coincidență la indicele i*/ }
```

- Există două condiții care finalizează căutarea:
 - (1) Elementul a fost găsit adică $a[i]=x$.
 - (2) Elementul nu a fost găsit după parcurgerea integrală a tabloului adică $i=n-1$ și $a[i] \neq x$.
- **Invariantul** buclei, care de fapt este condiția care trebuie îndeplinită înaintea fiecărei incrementări a indexului i , este:

$(0 \leq i < n-1) \wedge (\forall k: 0 \leq k < i: a[k] \neq x)$

- Acest invariant exprimă faptul că pentru valori ale lui $k < i$, nu există coincidență.
- **Condiția de terminare** a căutării este:
 $((i = n-1) \vee (a[i] = x)) \wedge (\forall k: 0 \leq k < i: a[k] \neq x)$
- Această condiție 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*/

tip_element a[n]; /*tabloul de căutare*/
tip_element x;     /*elementul căutat*/

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

- În legătură cu acest algoritm, indiferent de implementare, se precizează următoarele:

- La părăsirea buclei mai trebuie realizată o comparație a lui $a[i]$ cu x în vederea stabilirii existenței sau nonexistenței elementului căutat (dacă $i=n-1$).
- Dacă elementul este găsit, el este elementul cu cel mai mic indice în situația în care există mai multe elemente identice.
- Desigur există și varianta clasică a căutării liniare implementată cu instrucțiunea **for**

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

/*parametri de intrare: x- elementul căutat; a- tabloul de
căutare; n- dimensiunea tabloului*/
/*returnează valoarea n pentru x negăsit */
/*returnează valoarea i<n pentru x găsit pe poziția 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 și în consecință căutarea poate fi accelerată, prin simplificarea expresiei booleene.
 - O soluție de a simplifica expresia este aceea de a găsi **un singur factor** care să-i implice pe cei doi existenți.
 - Acest lucru este posibil dacă se garantează faptul că va fi găsită cel puțin o potrivire.
- În acest scop tabloul a se completează cu un **element adițional** $a[n]$ căruia i se atribuie inițial 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.d].

```
-----
/*căutare liniară tehnica fanionului (while) */
                                                    /*1.4.2.1.d*/

tip_element a[n]; /*tabloul de căutare*/
tip_element x;    /*elementul căutat*/

int i=0; /*indice de căutare*/
a[n]=x;  /*poziționare fanion*/
while (a[i]!=x)
  i++;
```

```

    if (i==n){ /*în tablou nu există elementul căutat*/}
    else {/*avem o coincidență la indicele i*/}

```

- **Condiția de terminare** dedusă din același invariant al ciclului este:

```

(a[i]=x)&(Ak:0 ≤ k < i:a[k]≠x)

```

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

```

/*căutare liniară tehnica fanionului (do) */

```

```

/*1.4.2.1.e*/

```

```

tip_element a[n]; /*tabloul de căutare*/
tip_element x; /*elementul căutat*/

int i=-1; /*indice de căutare*/
a[n]=x; /*poziționare fanion*/
do{
    i++;
}while (a[i]!=x);
if (i==n){ /*în tablou nu există elementul căutat*/}
else {/*avem o coincidență la indicele i*/};

```

```

/*căutare liniară tehnica fanionului (for)*/

```

```

/*parametri de intrare: x- elementul căutat; a- tabloul de
   căutare; n- dimensiunea tabloului (include fanionul)*/
/*returnează valoarea n pentru x negăsit */
/*returnează o valoare i<n pentru x găsit pe poziția i */

```

```

int cautare_fanion(int x,int a[],int n) /*1.4.1.f*/
{int i;
  a[n-1]=x; /*poziționare fanion*/
  for(i=0;a[i]-x;i++);
  return i;
}

```

- **Performanța** căutării liniare este $O(n)$, 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 **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**, în cazul de față un **tablou** care satisface predicatul A_k :

```

/* Definirea unui tablou ordonat*/

```

```
tip_element 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 $a[m]$ și 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 indici mai mici ca m .

- În consecință rezultă algoritmul denumit **căutare binară**.
 - Variabilele *stanga* și *dreapta* sunt indici și 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]; /*tabloul de căutare*/  
tip_element x; /*elementul de căutat*/  
int stanga, dreapta; /*limitele intervalului de căutare*/  
int m; /*indice element selectat*/  
int gasit;  
  
stanga=0; dreapta=n-1; gasit=0;  
while((stanga<=dreapta)&&(!gasit))  
{  
    m=(stanga+dreapta)/2; /*sau orice valoare cuprinsă  
                                între stanga și dreapta*/  
    if(a[m]==x)  
        gasit=1;  
    else  
        if(a[m]<x)  
            stanga=m+1;  
        else  
            dreapta=m-1;  
}  
if(gasit) { /*avem o coincidență la indicele i*/ }
```

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

```
(stanga<= dreapta)&(Ak: 0 <= k < stanga : a[k]<x)&
```

```
(Ak: dreapta< k <= n-1 : a[k] > x)
```

- **Condiția de terminare** este:

```
gasit OR ((stanga>dreapta)&(Ak: 0<=k<stanga : a[k]<x)&
```


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

- Alegerea lui m este **arbitrară** în sensul că ea **nu** influențează corectitudinea execuției algoritmului, în schimb influențează eficiența sa.
- **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 pași de căutare va fi $O(\log_2 n)$.
 - Această performanță reprezintă o îmbunătățire remarcabilă față de căutarea liniară unde numărul mediu de căutări este $n/2$.
- O variantă mai compactă a căutării binare apare în secvența 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;
}
```

- Eficiența căutării binare poate fi îmbunătățită ca și în cazul căutării liniare, prin **simplificarea condiției de terminare**.
 - Acest lucru se poate realiza simplu dacă **se renunță** la ideea de a termina algoritmul de îndată ce s-a stabilit coincidența.
 - La prima vedere acest lucru **nu** pare prea înțelept.
 - La o examinare mai atentă, se poate observa că câștigul în eficiență obținut la fiecare pas este mai substanțial decât pierderea provocată de câteva comparații suplimentare de elemente.
 - Se reamintește faptul că numărul de pași este $\log_2 n$.
- Soluția cea mai rapidă are la bază următorul **invariant**:

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

- 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]; /*tabloul de căutare*/
tip_element x; /*elementul de căutat*/
int stanga,dreapta; /*limitele intervalului de căutare*/
int m; /*îndice element selectat*/
```

```
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*/}

```

- Condiția de terminare în acest caz este $stanga = dreapta$.
- Cu toate acestea, egalitatea $stanga = dreapta$ **nu** indică automat găsirea elementului căutat.
- Dacă $dreapta \geq n$ **nu** există nici o coincidență. 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 consecință este necesară efectuarea în afara ciclului a unui test $a[dreapta] = x$ care de fapt stabilește existența coincidenței.
- Acest algoritm asemenea căutării liniare, găsește 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 secvența [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 îmbunătățire 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 cărții, în schimb un nume care începe cu V este căutat spre sfârșitul 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 relației de mai jos, unde stanga și dreapta sunt **limitele intervalului de căutare**:

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

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

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

- Evaluarea ține cont de valorile efective ale cheilor care mărginesc intervalul, în stabilirea indicelui la care se face următoarea căutare
- Această estimare însă, presupune o **distribuție uniformă** a valorilor cheilor pe domeniul vizat.

/ căutare prin interpolare */*

```
int cautare_interpolare(element a[],int ultim,int x){
    int 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 reușită cât și în caz de nereușită mai puțin de $lg(lg(n) + 1)$ comparații [Se 88].
- Funcția $lg(lg n)$ este o funcție care crește 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 obține 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 funcție de numărul de dimensiuni ale spațiului la care se referă.
- În matematică un astfel de **tip compus** se numește **produsul cartezian** al **tipurilor constitutive**.
 - **Domeniul de valori** al **tipului compus** constă din toate combinațiile posibile ale valorilor tipurilor componente, selectând câte o singură valoare din fiecare.
 - O astfel de combinație se numește **n-uplu**.
- Numărul total de combinații este egal cu produsul numerelor de elemente ale fiecărui tip constitutiv, adică **cardinalitatea tipului compus** este egală cu **produsul cardinalităților** tipurilor constitutive.
- Termenul care descrie o **dată compusă** de această natură în programare este cuvântul **record (articol)** respectiv **struct**.
- În secvența [1.4.3.a.] se prezintă modul generic de definire a unui **tip articol** în limbajul C.

```
/*Definire structură */
```

```
struct nume_structura{
    tip1 lista_nume_câmp1;
    tip2 lista_nume_câmp2;                                [1.4.3.a]
    ...
    tipn lista_nume_câmpn;
} 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 data_nastere;
    enum stare{necasatorit,casatorit,divortat,vaduv}
        situatie_sociala;
    enum sexul{barbatesc,femeiesc} sex;
} persoana;

```

-
- **Identificatorii** $\text{nume_c\acute{a}mp}_1, \text{nume_c\acute{a}mp}_2, \dots, \text{nume_c\acute{a}mp}_n$ introduși la definirea tipului, sunt **nume** conferite de către programator componentelor individuale ale tipului.
 - Identificatorii sunt utilizați ca și **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 notația "**dot**" sau "**punct**":

$x.\text{nume_c\acute{a}mp}_i$

- O atribuire a respectivei componente poate fi precizată prin construcția sintactică:

$x.\text{nume_c\acute{a}mp}_i = x_i$

unde x_i este o valoare (expresie) de tip tip_i .

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

{Exemple de instanțiere a structurilor articol}

```

data_calendar d;
persoana p;                                     /*1.4.3.c*/

d.luna=9;
p.nume="popescu";
p.data_nastere.zi=24;
p.situatie_sociala=necasatorit;

```

-
- Exemplul care se referă la tipul *persoana*, pune în evidență faptul că constituenții unui tip articol pot fi la rândul lor structurați, această situație 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 construcția sintactică:

x.a[i]

- Componenta având numele de selector `nume_câmp` aparținând celei de-a i-a componente de tip articol a unui tablou a este precizată prin construcția:

`a[i].nume_câmp.`

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

TDA Articol

[1.4.3.d]

Modelul matematic:

O colecție finită de elemente numite câmpuri, care pot aparține unor tipuri diferite. Exista o corespondență biunivocă între lista identificatorilor de câmpuri și colecția de elemente.

Notatii:

a - obiect de tip articol;
id - identificator nume de câmp;
e - obiect de același 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*) - funcție care returnează valoarea câmpului *id* din *a*.

- Secvența [1.4.3.e.] prezintă câte un exemplu de implementare a tipului de date abstract articol
 - După cum se observă, fazele de definire și de instanțiere pot fi separate sau se pot cumula în diferite moduri, limbajul C permițând multă flexibilitate în acest sens
-

/* Exemplu de implementare */

```
struct punct {
    int x;
    int y;
}
struct dreptunghi {
    struct punct varf1;
    struct punct varf2;
}
struct punct p1, p2;
double dist;
struct dreptunghi fereastr;
```

//Exemplu de utilizare

```
dist=sqrt((double)(p2.x-p1.x)*(p2.x-p1.x))+(double)(p2.y-
p1.y)*(p2.y-p1.y));
fereastr.varf1.x=188;
```

1.4.4. Structura uniune

- Limbajul C definește structura **uniune**.
- O uniune este de fapt o variabilă care poate memora la momente diferite de timp obiecte de diferite tipuri și dimensiuni. Compilatorul este acela care păstrează evidența și aliniază în mod corespunzător datele memorate.
- Uniunile au fost concepute în ideea de a manipula tipuri diferite de date într-o singură locație (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;           /*1.4.4.a*/  
    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 precauție în acest sens.
 - Cu alte cuvinte, pentru orice utilizare corectă a tipului, valoarea memorată va aparține ultimului tip memorat.
- Accesul la componentele structurii se realizează utilizând notația dot.

uniune.membru sau pointer_uniune.membru

1.4.5. 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 așa numitele structuri avansate: secvențele, listele, arborii, grafurile, etc, sunt caracterizate prin **cardinalitate infinită**.
 - Această diferență față de structurile clasice este de importanță profundă având consecințe practice semnificative.

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

$S_0 = \langle \rangle$ (secvența vidă)

$S_i = \langle S_{i-1}, s_i \rangle$ unde $0 < i$ și $s_i \in T_0$ [1.4.5.a]

- Cu alte cuvinte, o secvență cu tipul de bază T_0 , este:
 - (1) Fie o secvență **vidă**.
 - (2) 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.
- Consecințe 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" și 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 și utilizate structuri avansate de date.
- Tehnicile de generare și 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**:
 - Ea este o **structură avansată** din punctul de vedere al cardinalității care este infinită.
 - Este însă atât de curent utilizată încât includerea sa în mulțimea **structurilor fundamentale** este consacrată.

- Această situație este influențată și de faptul că, alegerea unui **set potrivit de operatori** referitori la **structura secvență**, permite implementatorilor să adopte reprezentări potrivite și eficiente ale acestei structuri.
- 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.

1.4.5.1. Tipul de date abstract secvență

- Includerea **structurii secvență** în rândul structurilor fundamentale, presupune **restrângerea setului de operatori** de o asemenea manieră încât se permite numai **accesul secvențial** la componentele structurii.
 - Această structură este cunoscută și sub denumirea de **fișier secvențial** sau pe scurt **fișier**.
- **Structura secvență** este supusă **restricției** de a avea componente de același tip, cu alte cuvinte este o structură **omogenă**.
- Numărul componentelor, denumit și **lungime** a secvenței, se presupune a fi **necunoscut** atât în faza de compilare, cât și 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 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 secvență și permite încadrarea lor în rândul **structurilor fundamentale**, deși 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 secvență se precizează numai **tipul de bază**.


```
TYPE tip_secvență = FILE OF tip_de_bază;
```
- În cadrul unei secvențe definite ca mai sus în orice moment este **accesibilă o singură componentă**, denumită **componentă curentă**, care este precizată printr-un **indicator** (pointer) asociat secvenței.
 - Acest indicator avansează secvențial, practic după execuția oricărei operații asupra secvenței.

- În plus oricărei secvențe i se asociază un operator boolean standard **sfârșit de fișier** notat cu $Eof(f : tip_secventa)$ care permite sesizarea sfârșitului secvenței.
- Modul concret de acces la componente, precum și posibilitățile de prelucrare efectivă, rezultă din semantica operatorilor definiți pentru tipul de date abstract secvență [1.4.5.1.a].
- Implementările recente ale limbajelor Pascal și C introduc în setul de instrucțiuni accesibile programatorului și operatori pentru prelucrarea secvențelor implementate ca și fișiere secvențiale.
- În secvența [1.4.5.1.b] apare o astfel de implementare definită în limbajul Pascal. Variante asemănătoare stau la dispoziție și în limbajul C.

TDA Secvență

Modelul matematic: secvență de elemente de același tip. Un indicator la secvență indică elementul următor la care se poate realiza accesul. Accesul la elemente este strict secvențial.

Notatii:

tip_de_baza - tipul unui element al secvenței. Nu poate fi de tip secvență;
f - variabilă secvență;
e - variabilă de *tip_de_baza*; [1.4.5.1.a]
b - valoare booleană;
numeFisierDisc - șir de caractere.

Operatori:

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

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

ResetSecvență(*f*) - procedură care mută indicatorul la începutul secvenței *f* și 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 sfârșit de fișier al lui *f*;

FurnizeazăSecvență(*f*, *e*) - procedură care acționează în regim de consultare. Atâta vreme cât $Eof(f)$ este fals, furnizează în *e* elementul curent al secvenței *f* și avansează indicatorul acesteia;

DepuneSecvență(*f*,*e*) - pentru secvența *f* deschisă în regim de scriere, procedura copiază valoarea lui *e* în elementul curent al secvenței *f* și avansează indicatorul acesteia;

Adaugă(*f*) - procedură care deschide secvența *f* în regim de scriere, poziționând indicatorul la sfârșitul acesteia, cu posibilitatea de a adăuga elemente noi numai la sfârșitul secvenței.

InchideSecvență(*f*) - închide secvența.

Implementarea tipului secvență - Varianta Pascal}

```
TYPE tip_de_baza = ... ;
     tip_secvență = TEXT;

VAR   f: tip_secvență;                                [1.4.5.1.b]
       e: tip_de_baza;
       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)
```

1.4.5.2. Tipul de date abstract fișier cu acces direct

- Principalul dezavantaj al structurii secvență constă în faptul că accesul la componentele sale se realizează în manieră **strict secvențială** prin intermediul unui pointer a cărui poziționare este inaccesibilă programatorului în mod direct.
- Din acest motiv, pornind de la tipul de date abstract secvență, a fost dezvoltată o structură mai flexibilă numită **fișier cu acces direct**.
- Astfel, unui fișier cu acces direct i se asociază un **indicator** prin intermediul căruia pot fi accesate **oricare** din componentele fișierului.
- În acest scop fiecare **componentă** a fișierului este identificată printr-un **indice** care îi precizează **poziția** în fișier exact ca și la structura tablou.
 - Numerotarea componentelor începe cu 0,1,2,... și continuă până la DimensiuneFișier-1.
- Crearea și modificarea structurii **fișier cu acces direct** se realizează însă tot în manieră **secvențială**.

- 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 secvența [1.4.5.2.a] iar în secvența [1.4.5.2.b] un exemplu de implementare Pascal.

TDA Fișier cu acces direct

Modelul matematic: secvență de elemente de același tip. Un indicator asociat fișierului 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,... și continuă până la *DimensiuneFișier-1*.

Notatii:

tip_de_baza - tipul unui element al fișierului. Nu poate fi de tip fișier;
f - fișier cu acces direct;
e - variabilă de *tip_de_baza*; [1.4.5.2.a]
i - indice în fișier.

Operatori:

Atribuie(*f*,*numeFișierDisc*) - atribuie variabilei fișier *f* numele unui fișier disc precizat;

Rescrie(*f*) - procedură care mută indicatorul fișierului la începutul acestuia. Dacă fișierul *f* nu există el este creat. Dacă *f* există, vechea sa variantă se pierde și se creează un nou fișier *f* vid. Indicatorul asociat fișierului ia valoarea 0;

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

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

i= **DimensiuneFișier**(*f*) - funcție care returnează numărul curent de componente al fișierului *f* (numărul de componente memorate în *f*);

i= **PozițieFișier**(*f*) - funcție care returnează numărul înregistrării curente (cea precizată de indicatorul fișierului). La deschiderea unui fișier *PozițieFișier* ia valoarea 0;

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

FurnizeazăFișier(*f*,*e*) - procedură care furnizează în *e* elementul curent al fișierului *f* și avansează indicatorul acestuia;

Depune(*f*,*e*) - procedură care depune valoarea lui *e* în fișierul *f* în poziția precizată de indicatorul fișierului și avansează indicatorul la elementul următor;

ÎnchideFișier(*f*) - închide fișierul *f*.

{Implementarea tipului secvență cu acces direct- Varianta Pascal}

```
TYPE tip_de_baza = ... ;
     tip_fisier_acces_direct = FILE OF tip_de_baza;
     tip_indice = integer;

VAR  f: tip_fisier_acces_direct;
     e: tip_de_baza;
     i: tip_indice;
```

{ Rescrie (<i>f</i>)}	rewrite(<i>f</i>)
{ Reset (<i>f</i>)}	reset(<i>f</i>)
{ <i>i</i> := DimensiuneFișier (<i>f</i>)}	<i>i</i> := filesize(<i>f</i>)
{ <i>i</i> := PozițieFișier (<i>f</i>)}	<i>i</i> := filepos(<i>f</i>)
{ Caută (<i>f</i> , <i>i</i>)}	seek(<i>f</i> , <i>i</i>)
{ DepuneFișier (<i>f</i> , <i>e</i>)}	write(<i>f</i> , <i>e</i>); writeln(...);
{ FurnizeazăFișier (<i>f</i> , <i>e</i>)}	read(<i>f</i> , <i>e</i>); readln(...);
{ ÎnchideFișier (<i>f</i>)}	close(<i>f</i>)

-
- Variante similare de implementare ale **TDA secvență** există și pentru limbajele C respectiv C++.
 - Dacă la nivel principial conceptele prezentate rămân valabile, la nivel de implementare pot să apară diferențieri importante.