

Structuri de date și algoritmi

upT Universitatea
Politehnica
Timișoara

P-ța Victoriei nr. 2
RO 300006 - Timișoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 2

Ș.I. dr.ing. Adriana ALBU

adriana.albu@upt.ro

<http://www.aut.upt.ro/~adrianaa>

1. Structuri de date fundamentale (partea întâia)

1.1 Preliminarii

- Caracteristici ale sistemelor de calcul:
 - viteza de lucru
 - capacitatea de memorare
 - posibilitățile de acces la informațiile memorate
- Sistemele de calcul prelucrează informații
 - reprezintă o abstractizare a lumii reale
 - se concretizează într-o mulțime de date
- Constituirea informațiilor furnizate unui program comportă două etape (adeseori se întrepătrund):
 - stabilirea abstractizării valabile pentru rezolvarea problemei, în urma căreia rezultă un set de date inițial
 - stabilirea modului de reprezentare în sistem a acestor date

1.2 Tipuri de date / 1.2.1 Conceptul de tip de dată

- E necesar ca fiecare constantă, variabilă, expresie sau funcție să se încadreze unui anumit tip de dată
- Un tip de data se caracterizează prin:
 - mulțimea valorilor
 - grad (nivel) de structurare
 - set de operatori specifici
- Precizarea tipurilor de date se realizează prin **declarații**
 - explicite – preced textual utilizarea obiectelor încadrate în acele tipuri
 - implicite – unele tipuri sunt recunoscute implicit prin reprezentare

1.2.1 Conceptul de tip de dată

➤ Caracteristicile conceptului de tip de dată

- determină în mod univoc **mulțimea valorilor** pe care le poate asuma un element încadrat în tipul respectiv (constante, variabile sau valori generate de un operator sau o funcție)
- tipul unui element sintactic poate fi dedus din forma sa de prezentare sau din declarația sa explicită, fără a fi necesară execuția unor procese de calcul suplimentare
- fiecare operator sau funcție acceptă argumente de un tip precizat și furnizează rezultate de asemenea de un tip precizat
- presupune un anumit **grad de structurare** a informației, grad care e evidențiat de nivelul de organizare asociat tipului de dată

➤ Datorită acestor caracteristici

- compilatoarele verifică legalitatea și compatibilitatea unor construcții de limbaj, încă în faza de compilare (fără a fi necesară execuția)

1.2.1 Conceptul de tip de dată

➤ Metode de structurare

- prin agregare:
 - definirea unor tipuri de date noi prin agregare (conglomerare), pornind de la tipuri existente (anterior definite)
 - valorile tipurilor rezultate sunt conglomerate de valori ale tipurilor constitutive
 - dacă există un singur tip constitutiv, acesta se numește tip de bază
- prin încuibare:
 - definirea unor tipuri de date în interiorul altor tipuri

➤ Cele două metode pot fi combinate

- => un anumit nivel de structurare, bazat pe o ierarhie de structuri
- tipurile de la baza structurării trebuie să fie tipuri primitive (atomi)

1.2.1 Conceptul de tip de dată

- Tipuri primitive nestructurate (fundamentale):
 - tipuri standard, numite și **predefinite**, care au de regulă corespondență cu reprezentări asociate arhitecturii hardware a sistemului de calcul
 - **definite de utilizator** prin enumerarea valorilor constitutive ale tipului
- Dacă între valorile individuale ale tipului există o relație de ordonare, atunci tipul de dată este ordonat sau **scalar**
 - majoritatea tipurilor primitive sunt scalare
- Metodele de structurare de bază generează tipuri de date structurate:
 - statice
 - dinamice
 - (sunt definite de utilizator)

1.2.2 Conceptul de tip de data abstract (TDA)

➤ Un TDA poate fi definit drept o asociere între:

- un model matematic (MM)
- un set de operatori definit pentru valorile manipulate de modelul matematic

➤ Un TDA

- generalizează noțiunea de tip de dată primitiv
- încapsulează elementele specifice definiției sale
 - încapsulează alături de structura de date corespunzătoare modelului matematic și procedurile sau funcțiile care materializează operatorii

➤ La rândul său, o procedură sau o funcție

- este un element fundamental de programare care:
 - generalizează noțiunea de operator
 - încapsulează elementele specifice operatorului definit prin procedură sau funcție
 - încapsularea are ca rezultat definirea procedurii sau funcției într-un singur loc în program (încapsulare cod)

1.2.2 Conceptul de tip de data abstract (TDA)

- MM <-> structura de date
- Set de operatori <-> proceduri sau funcții

- În urma definirii, un TDA poate fi tratat în continuare ca un tip primitiv de dată
 - utilizatorul nemaifiind preocupat de maniera de proiectare a tipului de data, ci doar de maniera de acces (interfață)
- TDA permit de asemenea o abordare ierarhica în construirea tipurilor structurate abstracte (ex. TDA lista)

1.2.2 Conceptul de tip de data abstract (TDA)

➤ Avantajele utilizării TDA

- În urma definirii unui TDA, toate prelucrările se vor realiza în termenii operatorilor definiți asupra acestui tip
- Indiferent de maniera concretă de implementare a TDA, forma operatorilor, respectiv **interfața** acestora cu utilizatorul **rămâne nemodificată**
 - permite modificarea cu ușurință a implementării cu condiția păstrării prototipului funcției asociată operatorului
- Nu există nicio limită asupra numărului de operații care pot fi aplicate instanțelor unui model matematic precizat
 - cu specificația că fiecare set de operații definește și se refera la un TDA distinct

1.2.2 Conceptul de tip de data abstract (TDA)

➤ Implementarea unui TDA

- Prin implementarea unui TDA se înțelege translatarea (exprimarea) sa în termenii unui limbaj de programare:
 - definirea structurii de date care materializează modelul matematic al TDA
 - scrierea procedurilor (funcțiilor) care implementează operatorii precizați în definirea TDA (utilizând noțiunile limbajului de programare sau operatorii definiți deja)
- Un TDA se implementează pornind de la tipurile de bază specifice limbajului de programare și utilizând facilitățile de structurare puse la dispoziție de limbaj

➤ Prin implementare un TDA devine tip de data (TD)

1.2.2 Conceptul de tip de data abstract (TDA)

- TD sunt utilizate pentru declararea unor variabile de tipul respectiv
 - aceste variabile se numesc **instanțe** ale tipului de dată
 - procesul de declarare a variabilelor se numește **instanțiere**
- Variabila – un element sintactic precizat printr-un nume simbolic
 - i se asociază o **zonă de memorie** (de o anumită dimensiune, specifică tipului de dată), al cărei conținut îl reprezintă valoarea curentă a variabilei
- Structura de date – o colecție specifică de variabile, aparținând unuia sau mai multor tipuri de date, asociate în diverse moduri, cu scopul de a facilita accesul, prelucrarea și relațiile dintre entitățile constitutive

Rezumat pentru 1.2

Tip de date abstract (TDA)
asociere între un model matematic (MM) și un set de operatori specifici

Tip de date (TD)
implementare a unui TDA într-un limbaj de programare

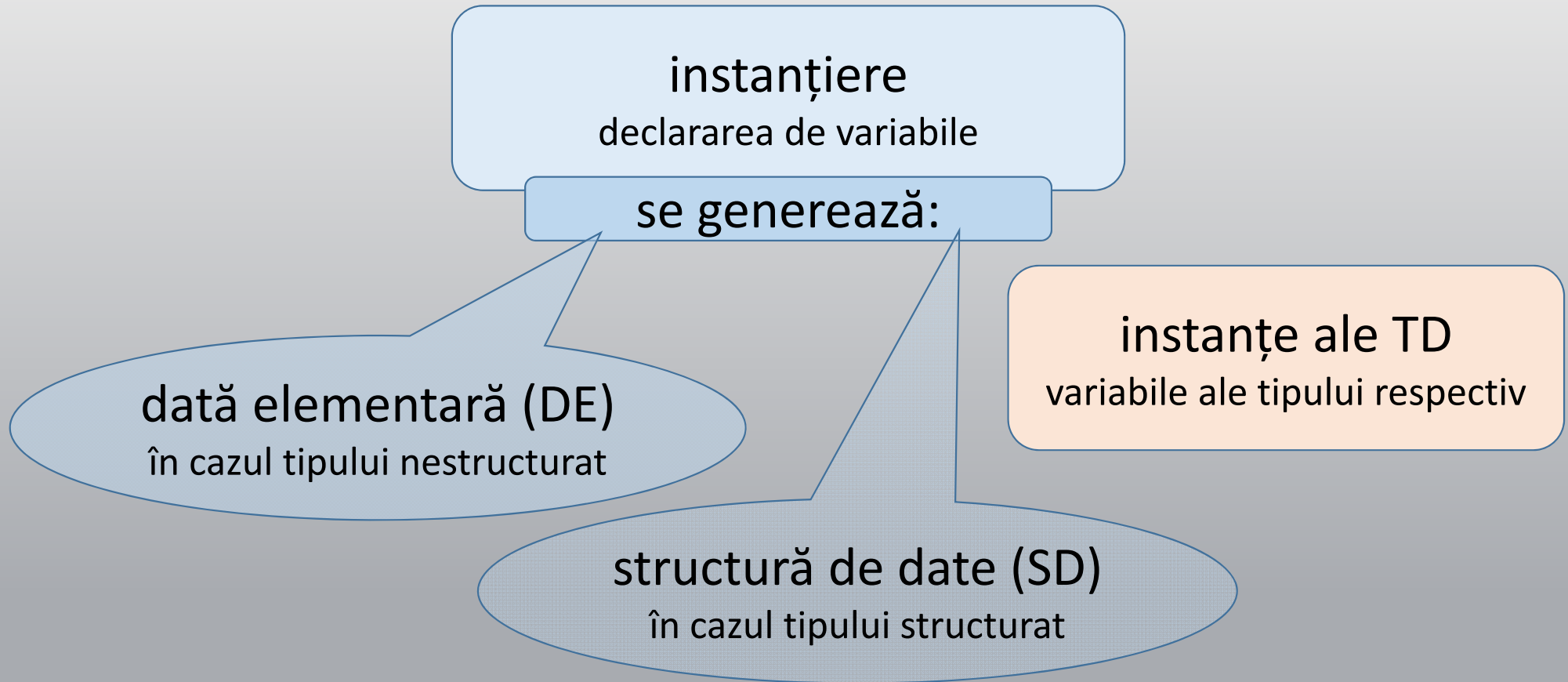
se caracterizează prin:

- mulțimea valorilor (pe care le pot lua elementele tipului respectiv)
- un anumit grad (nivel) de structurare (organizare) a informației
- set de operatori specifici

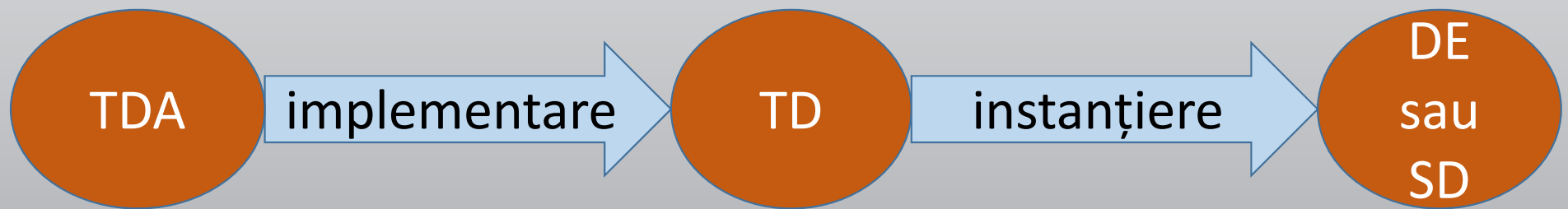
pot fi:

- nestructurate
- structurate

Rezumat pentru 1.2



Rezumat pentru 1.2



1.3 Tipuri primitive nestructurate /

1.3.1 Tipuri primitive standard (predefinite)

- Fac parte din categoria tipurilor nestructurate
- Sunt numite standard deoarece sunt implementate direct prin caracteristicile hardware ale sistemului
 - întreg
 - real
 - boolean
 - caracter
- TDA întreg, real, boolean, caracter sunt definite prin limbaj, pentru utilizare fiind necesară doar instanțierea (se încadrează în categoria tipurilor de date)

1.3.1 Tipuri primitive standard (predefinite)

➤ Tipul întreg

➤ este format din submulțimea reprezentabilă a numerelor întregi

➤ operatori:

- atribuire
- comparare
- aritmetici (+, -, x, /, %)
- operatorul DIV – împărțire întreagă, cu ignorarea restului
 - $m - n < (m \text{ DIV } n) \times n \leq m$
- operatorul MOD – modulo, furnizează restul împărțirii întregi
 - $(m \text{ DIV } n) \times n + (m \text{ MOD } n) = m$
- Uzual sunt mai multe categorii de tipuri întregi (signed, unsigned, short, long)

1.3.1 Tipuri primitive standard (predefinite)

➤ Tipul întreg

- MM – o mulțime de elemente scalare cu valori în mulțimea numerelor întregi ex.: $\{-20, \dots, -2, -1, 0, 1, 2, \dots, 20\}$

- Notatii

- i, j, k – variabile întregi
- inz – întreg diferit de zero
- inn – întreg non-negativ
- e – valoare întreagă
- b – valoare booleană

Operatori:

AtribuireIntregi(i,e)
AdunareIntregi (i,j) -> k
ScadereIntregi(i,j) -> k
InmultireIntregi(i,j) -> k
ImpartireIntregi(i,inz) -> k
Modulo(i,inz) -> inn
EgalZero(i) -> b
MaiMareCaZero(i) -> b

- Într-un limbaj de programare implementarea acestor operatori se realizează în mod direct

1.3.1 Tipuri primitive standard (predefinite)

➤ Tipul real

- Implementează o submulțime reprezentabilă a numerelor reale
- Operatori: +, -, x, /
- Calculele se realizează cu o anumită **aproximație**, funcție de lungimea zonei de memorie utilizată în reprezentarea valorilor reale (rotunjiri cauzate de efectuarea calculelor cu un număr finit de zecimale)
- Implementarea operatorilor se realizează direct prin construcții sintactice ale limbajelor de programare
- Uzual sunt mai multe categorii de tipuri reale (float, double)

1.3.1 Tipuri primitive standard (predefinite)

➤ Tipul boolean

- Implementează valorile logice standard Adevărat și Fals
- În limbajul C acest tip lipsește fiind substituit de valorile întregi:
 - 1 sau diferit de zero – adevărat
 - 0 – fals
- Operatori specifici: operatorii logici
 - conjuncție – AND, &&
 - reuniune – OR, ||
 - negație – NOT, !
- Operatorul **comparație** aplicat asupra valorilor oricărui tip de date conduce la rezultate de tip boolean
 - poate fi atribuit unei variabile de tip logic
 - poate fi utilizat ca operand al unui operator logic într-o expresie booleană

1.3.1 Tipuri primitive standard (predefinite)

➤ Tipul caracter

- Mulțimea de caractere reprezentabile de regulă la un dispozitiv de afișare (mulțimea valorilor depinde de dispozitivul de ieșire)
- Au fost stabilite diverse convenții de codificare a caracterelor:
 - ISO (International Standard Organisation)
 - ASCII (American Standard Code for Information Interchange)
- Uzual se utilizează valori întregi fără semn în domeniul $[0, 255]$, reprezentabile pe un octet
- Codurile sunt grupate (submulțimi ordonate și coerente) pe litere mari, mici, cifre, semne de punctuație, alte caractere reprezentabile etc.
 - principiul de codificare permite determinarea apartenenței unui caracter la una dintre submulțimi
- Unele limbaje de programare pun la dispoziție funcții de transfer între tipul caracter și alte tipuri (în C: `atoi()`, `atof()`, `itoa()`)

1.3 Tipuri primitive nestructurate / 1.3.2 Tipul enumerare

- Oferă o manieră de a utiliza în cadrul programelor numere întregi în locul noțiunilor abstracte
- Este un tip definit de utilizator prin enumerarea tuturor componentelor sale
- $\text{card}(T)$
 - cardinalitatea tipului T – numărul valorilor distincte aparținând unui tip de dată T
 - uzual cardinalitatea este utilizată drept o măsură a cantității de memorie necesară reprezentării tipului T
- Definirea unui tip enumerare, introduce nu numai un nou identificador de tip, ci în același timp introduce identificadorii (simbolurile) care precizează valorile noului tip

1.3 Tipuri primitive nestructurate / 1.3.2 Tipul enumerare

➤ enum

```
enum day {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
```

- tip scalar
- constante sunt asimilate cu numerele întregi atribuite succesiv în timpul procesului de declarare a tipului
 - excepție: situația în care utilizatorul face atribuire explicit

```
enum day today, tomorrow;
```

```
enum month {Jan=31, Feb=28, Mar=31, Apr=30, May, Jun=30, Jul, Aug=31, Sep=30, Oct, Nov=30, Dec};
```

➤ Operatori:

- atribuire
- comparație
- succesor, predecesor
- transformare (forțare) de tip – conversia uzuală este cea între tipul enumerare și întreg

```
today=Sunday;
```

```
if (today==Sunday) tomorrow=Monday;  
else tomorrow=today+1;
```

1.3 Tipuri primitive nestructurate / 1.3.3 Tipul subdomeniu

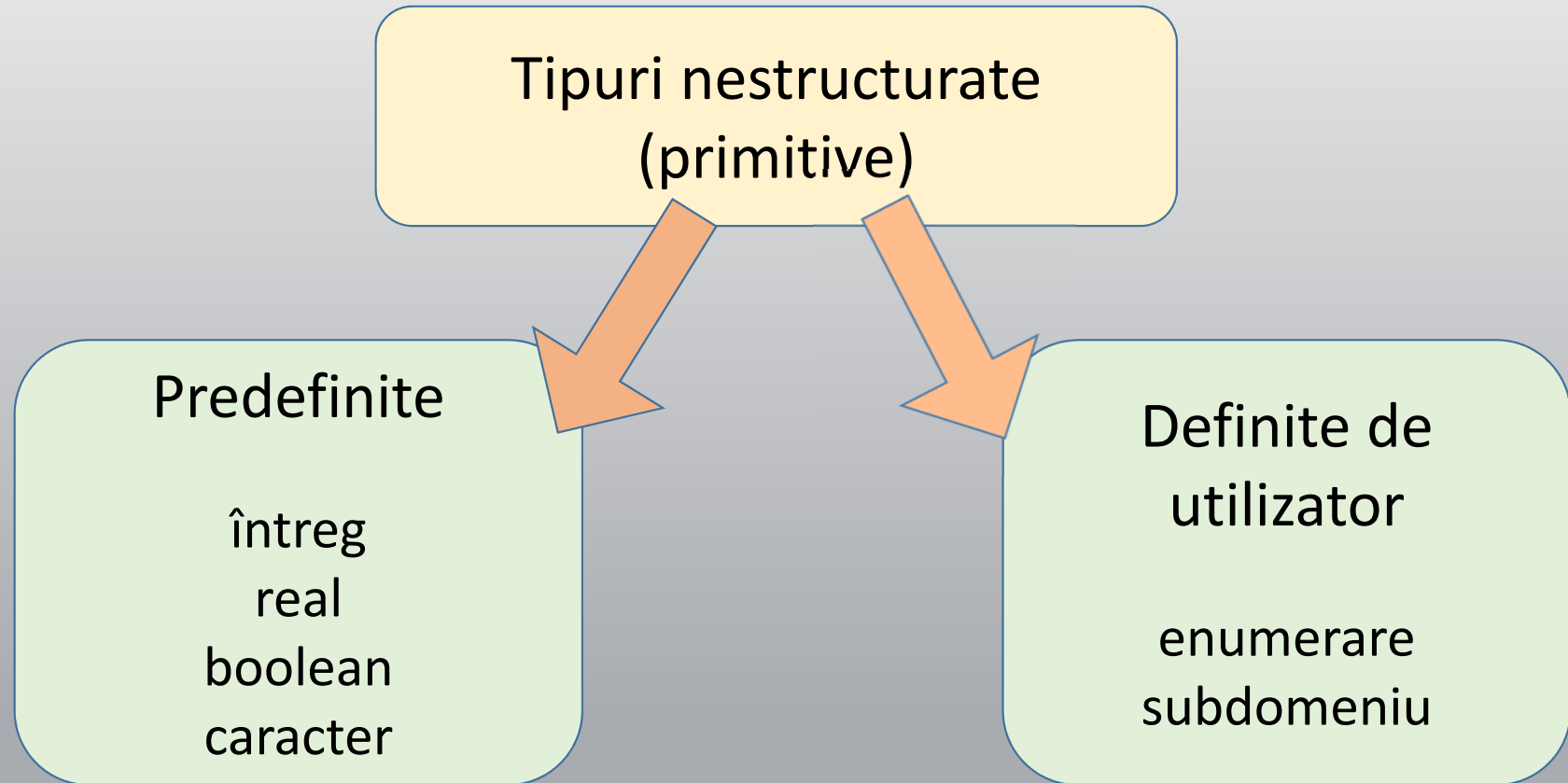
- Precizează un domeniu restrâns în care poate lua valori un anumit tip
- Se definește în raport cu un tip deja existent, numit tip scalar asociat

TYPE TipSubdomeniu = min..max;

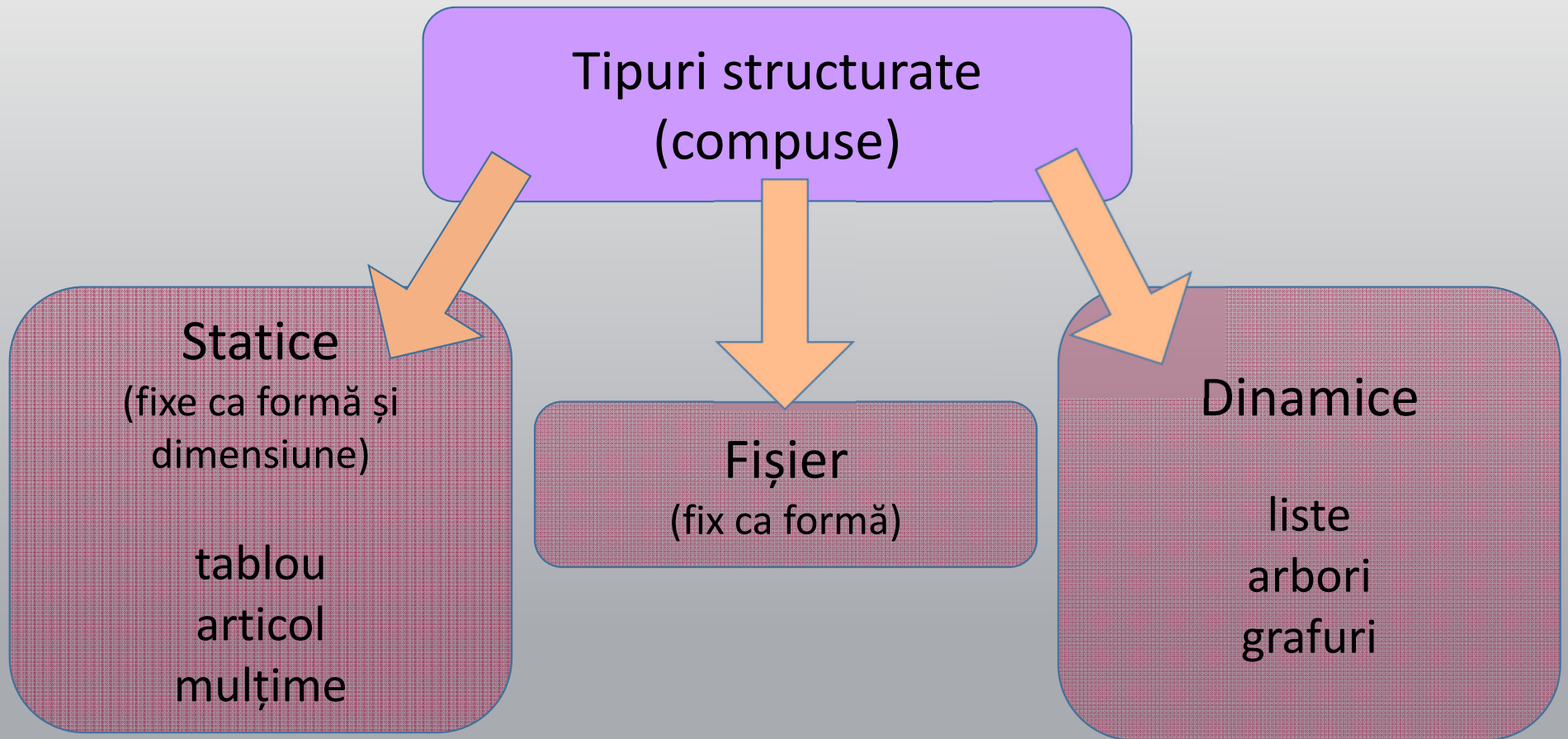
TYPE AN=2020..2023;

- Implementarea tipului subdomeniu impune verificarea de către compilator a legalității atribuirilor; în cazul atribuirii unor variabile, verificările se produc doar în timpul execuției programului. Din acest motiv scade eficiența codului generat, dar crește siguranța execuției.
- (Nu e specific limbajului C)

Rezumat pentru 1.3



Intro pentru 1.4



1.4 Tipuri structurate / 1.4.1 Structura tablou. TDA tablou

- Construcția se bazează pe facilitățile hard ale sistemului de calcul, respectiv pe mecanismul de adresare indexată
- Structură de date **omogenă** ale cărei componente aparțin aceluiași tip constitutiv, numit **tip de bază**
- Structură cu **acces direct** (random acces)
 - toate elementele sale sunt direct și în mod egal accesibile
- Structură **statică**, cu dimensiune și formă fixă
- **Indicii de acces** la elementele tabloului aparțin unui tip scalar
 - indicii selectează pozițional componenta dorită
- Operatori: constructor, selector, atribuire

TDA tablou

➤ MM

- secvență de elemente de același tip
- indicele asociat fiecărei componente aparține unui tip ordinal finit
- există o corespondență biunivocă între valoarea indicelui și componentele tabloului

➤ Notatii:

- TipElement a – tablou unidimensional;
- i – TipIndice;
- e – obiect de TipElement

➤ Operatori:

- DepuneTablou (a, i, e); // a[i]=e;
- TipElement FurnizeazaDinTablou (a, i); // e=a[i];

1.4.2 Tehnici de căutare în tablouri

➤ Căutarea (regăsirea informației)

- operație frecventă
- => una dintre tehnicile cele mai abordate și studiate în programare
- viteza cu care se efectuează această operație într-o bază de date de dimensiuni mari **influențează în mare măsură eficiența** unei aplicații
- presupune efectuarea unor operații de **comparație**
- => uneori elementele tabloului se identifică printr-o **cheie**, ce aparține unui tip scalar (poate fi constituită dintr-un singur câmp sau o combinație de câmpuri)
- uzual, returnează indicele elementului din tablou pentru care cheia coincide cu cheia de căutare dată, respectiv indicele elementului a cărui valoare coincide cu valoarea de căutat dată, în cazul în care nu se utilizează o cheie pentru identificare

1.4.2 a) Căutarea liniară

- Parcurgerea secvențială (traversarea) a tabloului
 - unica metodă de căutare dacă nu avem nicio informație apriorică asupra tabloului
 - realizată într-o manieră ordonată prin incrementarea (decrementarea) indicelui
- Căutarea elementului x într-un tablou unidimensional a
- Rezultatul:
 - găsirea elementului de valoare x și eventuala returnare a indicelui ($a[i]=x$)
 - negăsirea elementului x și depășirea tabloului ($i > N-1$ dacă indicii aparțin $0 \dots N-1$)

Căutarea liniară

```
int i=0;
while ( (i<N) && (a[i] !=x) ) i++;
if (a[i] !=x)
    printf("Nu exista elementul cautat");
else
    printf("Elem. cautat este pe poz. %d", i);
```

➤ **invariant:** condiția care dacă este îndeplinită, permite rămânerea în buclă (respectiv reluarea buclei)

- $(0 \leq i < N) \ \&\& \ (a[i] \neq x)$

➤ **condiția de terminare:**

- $((i == N) \ || \ (a[i] == x))$

Aprecierea performanțelor căutării liniare

- Criteriul: numărul de comparații efectuate până la găsirea elementului căutat
- Caz particular:
 - x pe poziția i \rightarrow se efectuează $2 * (i + 1)$ comparații
 - timpul de execuție proporțional cu i
- Cel mai defavorabil caz:
 - x pe poziția $N - 1$ $\rightarrow 2 * N$ comparații
- În cazul în care elementul x se găsește cu siguranță în tablou, numărul de comparații cel mai probabil este $2 * N / 2$ (apreciere timp mediu)
- **Performanța căutării liniare:** în medie un element este găsit după parcurgerea a jumătate dintre elementele tabloului
- Reducerea timpului de execuție se poate obține prin reducerea numărului de operații efectuate în fiecare ciclu

1.4.2 b) Căutarea liniară. Metoda fanionului

- Simplificarea invariantului se poate realiza garantând cel puțin o „potrivire” în procesul de căutare (prezența elementului x în tablou)
- Se completează tabloul cu o locație suplimentară, în care se plasează elementul căutat x

...

```
a[N]=x;
```

```
i=0;
```

```
while(a[i] != x) i++;
```

```
if(i==N)
```

```
    printf("nu exista elementul cautat");
```

```
else
```

```
    printf("Elem. cautat este pe poz. %d", i);
```

...

Căutarea liniară. Metoda fanionului

➤ Elimină

- n teste în caz de căutare fără succes
- i teste în caz de căutare cu succes (reducere a timpului cu 20-50%)

➤ Alte variante de reducere a timpului în căutările liniare:

- păstrarea datelor ordonate după cheie
 - permite abandonarea căutării la un moment dat
- păstrarea datelor ordonate după frecvența cu care sunt căutate
 - timpul mediu se reduce simțitor

1.4.2 c) Căutarea binară

- Informații suplimentare referitoare la organizarea datelor prezente în tablou => accelerarea căutării
- Elementele tabloului sunt ordonate conform unui anumit criteriu:
 - $A_k : 0 < k < N : a_{k-1} \leq a_k$
- Principiul căutării binare:
 - se compară elementul căutat x cu elementul aleatoriu a_m
 - dacă $a_m < x$, căutarea continuă în intervalul care conține elemente cu indici mai mari decât m
 - dacă $a_m > x$, căutarea continuă în intervalul care conține elemente cu indici mai mici decât m
 - dacă $a_m == x$, elementul a fost găsit și căutarea se oprește
 - procesul continuă până când se găsește elementul x sau intervalul în care se execută căutarea devine vid

Căutarea binară

➤ indicii s și d precizează limita stângă și dreaptă a intervalului

```
int s=0, d=N-1, gasit=0, m;  
while ((s<=d) && (!gasit)) {  
    m=(s+d)/2; //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;  
}
```

➤ Invariant: $(s \leq d) \ \&\& \ (A_k : 0 \leq k < s : a_k < x) \ \&\& \ (A_k : d < k < N : a_k > x)$

➤ Eficiența algoritmului este influențată de alegerea lui m :

- scopul urmărit este de a reduce în fiecare pas intervalul în care se face căutarea

Căutarea binară

- Dacă fiecare intrare în buclă asigură înjumătățirea intervalului:
 - după prima trecere vor rămâne $N/2$ elemente de procesat
 - după k treceri vor rămâne $N/2^k$ elemente de procesat
- Procesul continuă până la un interval care conține 1 element

$$N/2^k < 1 \quad \rightarrow \quad N < 2^k \quad \rightarrow \quad \log_2 N < k \quad (\text{notația } \lg N)$$

- Numărul maxim de treceri: $\log_2 N$ (rotunjit superior)
- Căutarea binară, numită și bisecție, are la bază procesul eliminării:
 - ori de câte ori se face o comparație în procesul de căutare, sunt posibile două soluții pentru alegerea noului interval de căutare

1.4.2 d) Căutarea binară performată

➤Simplificarea invariantului prin renuntarea la conditia „!gasit”

```
int s=0, d=N, m;  
while (s<d) {  
    m= (s+d) /2;  
    if (a[m]<x) s=m+1;  
    else d=m;  
}  
if(d==N) printf ("elementul nu exista")  
if(d<N)  
    if(a[d]==x) printf("elem exista");  
    else printf("elem nu exista");
```

➤Găsește elementul x cu cel mai mic indice

1.4.2 e) Căutarea prin interpolare

➤ Creșterea performanței:

- determinarea cu mai multă precizie a intervalului în care se face căutarea
- căutare binară: $m = (s+d)/2 = s+(d-s)/2$
- căutare prin interpolare: $m = s + (d-s) (x-a[s]) / (a[d]-a[s])$

➤ În cazul unei distribuții uniforme a elementelor, $(x-a[s]) / (a[d]-a[s])$ va accelera procesul de căutare;

- numărul de treceri estimat: $\lg(\lg(N))$

➤ Dezavantaje:

- depinde de distribuția elementelor; orice abatere afectează performanța
- este performantă pentru valori foarte mari ale lui N

➤ Avantaje:

- se utilizează când elementele tabloului sunt structuri complexe
- pentru căutare externă, care necesită costuri suplimentare de acces

Vă mulțumesc!

Structuri de date și algoritmi

upT Universitatea
Politehnica
Timișoara

P-ța Victoriei nr. 2
RO 300006 - Timișoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 3

Ș.I. dr.ing. Adriana ALBU

adriana.albu@upt.ro

<http://www.aut.upt.ro/~adrianaa>

1. Structuri de date fundamentale (partea a doua)

Rezumat pentru cursul precedent

Tip de date abstract (TDA)
asociere între un model matematic (MM) și un set de operatori specifici

Tip de date (TD)
implementare a unui TDA într-un limbaj de programare

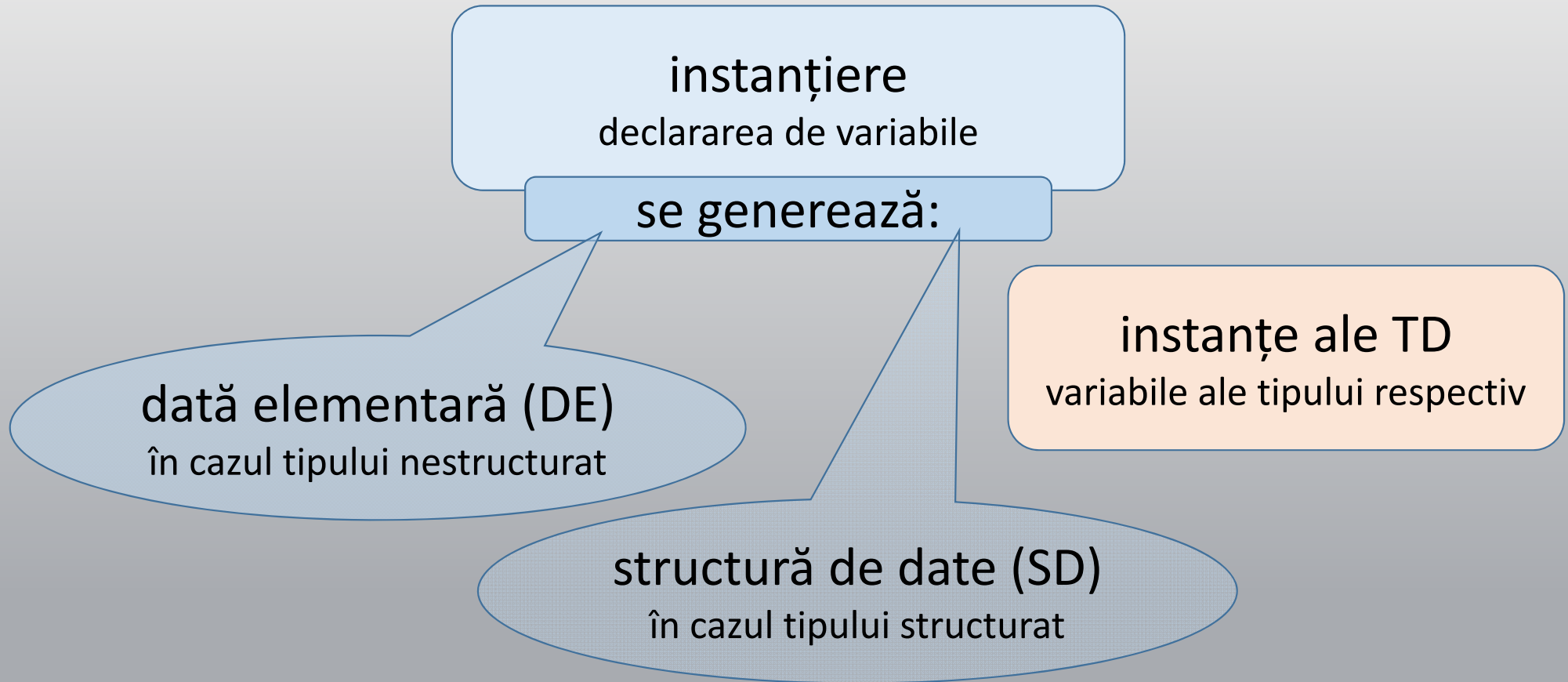
se caracterizează prin:

- mulțimea valorilor (pe care le pot lua elementele tipului respectiv)
- un anumit grad (nivel) de structurare (organizare) a informației
- set de operatori specifici

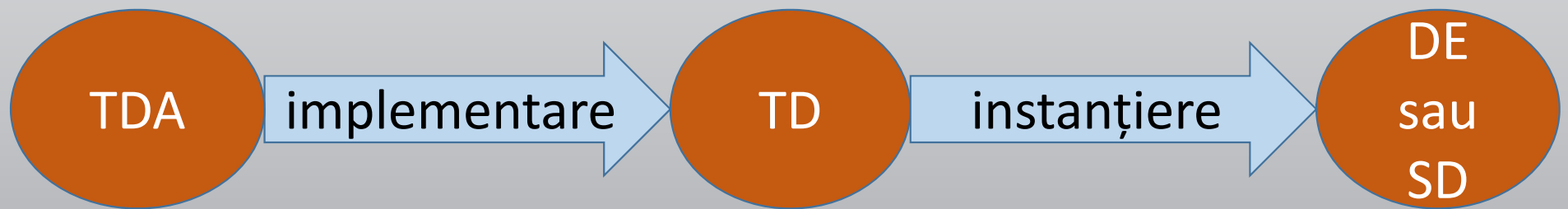
pot fi:

- nestructurate
- structurate

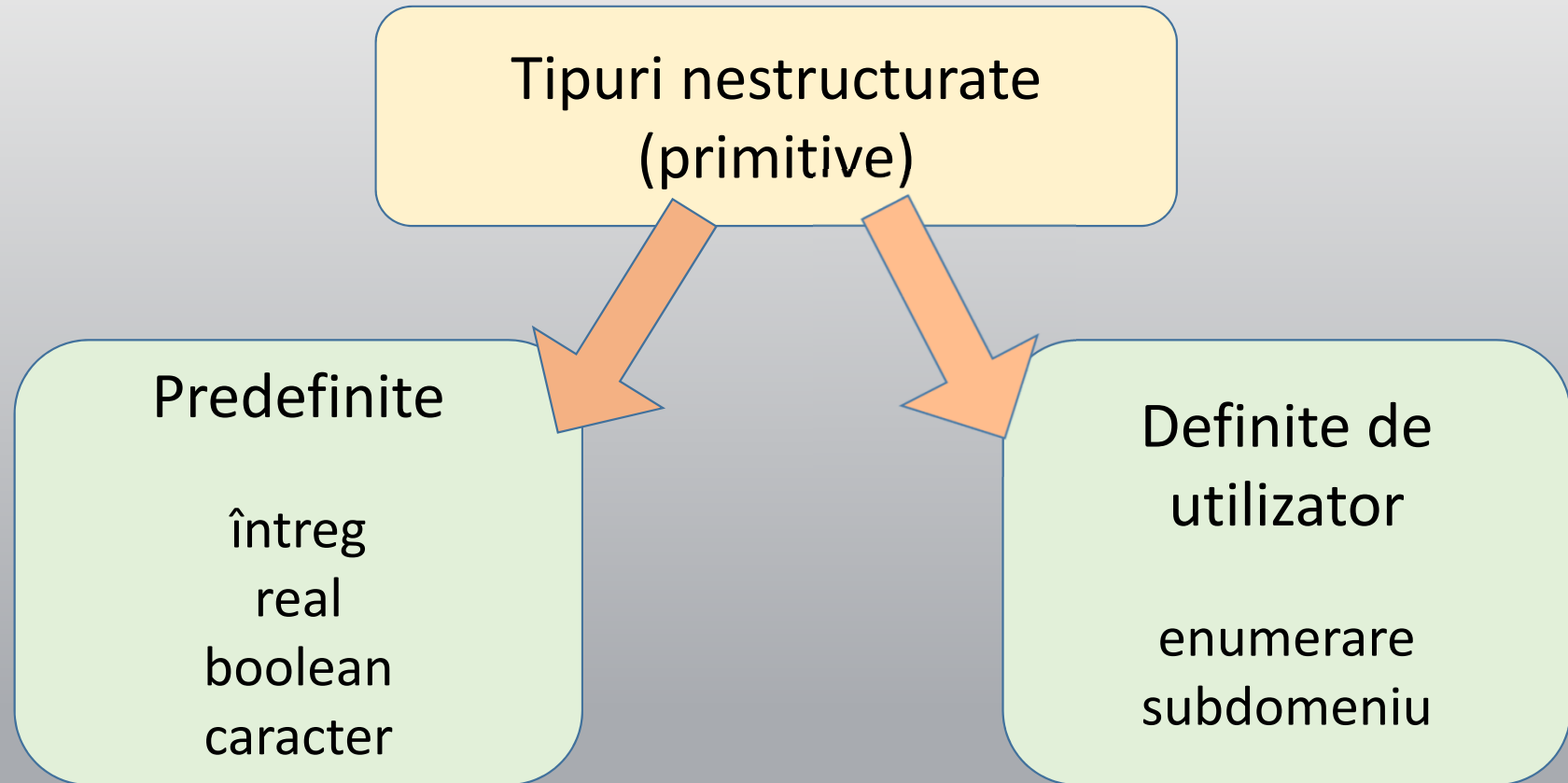
Rezumat pentru cursul precedent



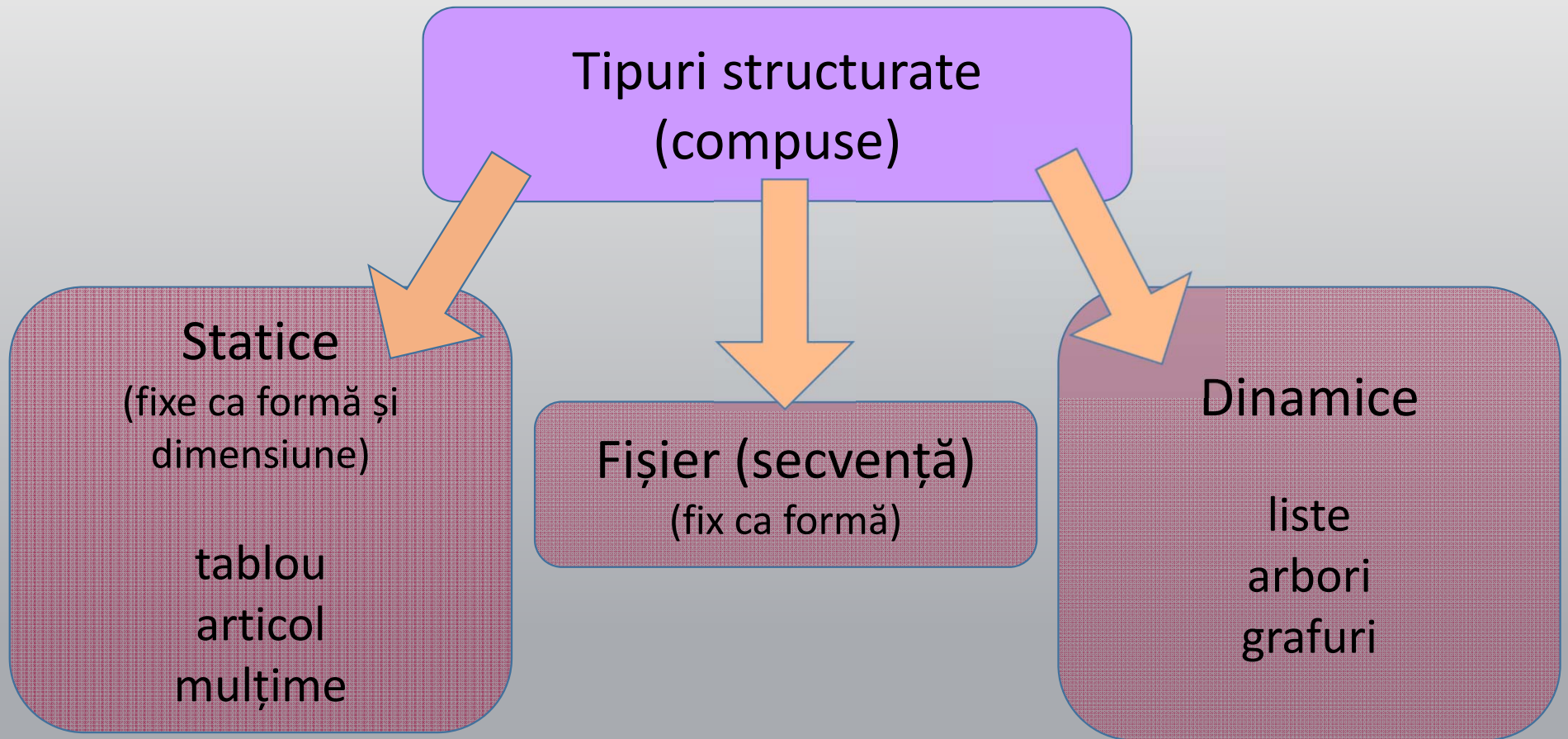
Rezumat pentru cursul precedent



Rezumat pentru cursul precedent



Rezumat pentru cursul precedent



1.4.3 Structura de date articol. TDA articol

- Se obține prin agregare, respectiv prin **reuniunea unor elemente aparținând mai multor tipuri constitutive** (structurate la rândul lor), într-un tip complex, structurat, numit **articol** (înregistrare)
- Mulțimea de valori asociată unui tip articol: totalitatea combinațiilor posibile ale valorilor tipurilor constitutive, selectând câte o singură valoare din fiecare tip

```
typedef struct{  
    tip1 nume1;  
    tip2 nume2;  
    ...  
    tipn numen;  
}articol;  
articol a;
```


1.4.3 Structura de date articol. TDA articol

- Articolul – structură de date **neomogenă**
- Selecția componentelor se realizează prin **identificatori de câmp** precizați la definirea tipului și **operatori de selecție**

```
articol a, *pa;  
articol tab[N];  
a.numel=...;  
pa=tab; //pa=&tab[0];  
pa->numel=...;  
tab[0].numel=...;
```

- Selecția prin identificatori de câmp (nume de câmp) este o **selecție fixă**, realizată în urma unor elemente cunoscute încă în faza de compilare

TDA articol

➤ MM:

- colecție finită de elemente numite câmpuri, care pot să aparțină unor TD diferite
- există o corespondență biunivocă între lista identificatorilor de câmpuri și colecția de elemente

➤ Notatii:

- a – instanță a tipului articol
- id – identificator de câmp
- e – valoare a tipului asociat lui id

➤ Operatori:

- $DepuneArticol(a, id, e)$
- $FurnizeazaArticol(a, id) \rightarrow e$

Articol cu variante

- Se utilizează când este necesar ca două sau mai multe tipuri de date să fie definite ca variante ale aceluiași tip

```
typedef enum{NOTA,CALIFICATIV}tip_e;
typedef struct{
    char materie[20];
    tip_e evaluare; //camp selector
    union{
        float n;
        char c;
    }rezultat;
}examen;
examen e;
e.evaluare=NOTA;
e.rezultat.n=9.5;
```

- Pentru identificarea variantei curente s-a introdus **discriminatorul de tip** sau **câmpul selector**

- Mulțimea valorilor tipului articol cu variante
- rezultă din reuniunea tipurilor constitutive
 - cardinalitatea tipului articol este suma cardinalităților tipurilor constitutive

1.4.4 Structura de date mulțime. TDA mulțime

➤ Este un tip structurat fundamental, definit sau nu prin construcții sintactice:

`TYPE TipMultime = SET OF T0;`

```
TYPE indice = 1..10;
```

```
TYPE multimeIndice = SET OF indice;
```

- $2^{10} = 1024$ valori
- `[], [1], [2], ..., [10], [1,2], [1,3], ..., [1,2,3,4,5,6,7,8,9,10]`

- T_0 – tip de bază
- mulțimea valorilor lui T_0 – mulțime de bază
- mulțimea de valori pentru `TipMultime` = puterea mulțimii de bază

`card(TipMultime) = $2^{\text{card}(T_0)}$`

➤ Fiecare valoare din mulțimea de baza este „prezentă” sau „absentă” în fiecare valoare asociată tipului mulțime

➤ O valoare a unei variabile de tip mulțime poate fi construită:

- Static – prin asignarea variabilei cu o constantă a tipului
- Dinamic – prin asignarea variabilei cu o expresie de calcul având drept operanzi mulțimi încadrate în același tip de bază

TDA mulțime

➤ MM: elementele tipului mulțime

- aparțin unui tip ordonat finit
- sunt membre ale unei mulțimi matematice

➤ Notatii:

- TipElement – tip de bază
- S, T, V – mulțimi ale căror elemente aparțin lui TipElement
- e – valoare a lui TipElement
- b – valoare booleană

➤ Operatori:

- specifici (legi de compoziție): atribuire, reuniune, scădere, intersecție;
- relaționali: egalitate, inegalitate, incluziune etc.
- DepuneMultime(S,T); EgalitateMultime(S,T)->b; ApartineMultime(S,e)->b;
Submultime(S,T)->b; Reuniune(S,T)->V; Intersectie(S,T)->V

1.4.5 Reprezentarea structurilor de date abstracte

- Un program poate fi conceput, realizat și verificat pornind de la principiile care stau la baza nivelului de abstractizare utilizat
 - Utilizatorul este astfel eliberat de detaliile legate de implementarea conceptelor abstracte la nivel fizic
- **Reprezentare**
 - exprimarea structurilor abstracte în termenii tipurilor de date standard, implementate pe sistemul de calcul
 - la nivelul reprezentării se realizează corespondența dintre structura de date abstractă și memoria fizică
- **Memoria**
 - tablou format din celule individuale numite locații
 - indicii acestor locații se numesc adrese
 - dimensiunea memoriei – dată de cardinalitatea tipului
 - locațiile de memorie sau multiplii acestora formează unități de informație (de memorie)

Reprezentarea tablourilor

- Reprezentarea în memorie a unui tablou cu elemente de tip T
 - se realizează prin transformarea într-un tablou având drept componente unități de informație, situate într-o zonă contiguă de memorie
- Adresa i asociată componentei j a tabloului se poate afla dacă se cunosc:
 - adresa de început i_0 (adresa de bază)
 - dimensiunea s în unități de memorie (biți, octeți) a componentelor
$$i = i_0 + (j-1) * s$$
- În multe situații, dimensiunea unei componente nu este identică cu dimensiunea unității de memorie, astfel încât anumite părți ale unității de informație pot rămâne **neutilizate** în cadrul alocării de memorie
- Rotunjirea numărului de unități de informație la următorul întreg se numește aliniere (umplere)

Reprezentarea tablourilor

➤ Factor de umplere:

- u = cantitatea de memorie necesară / cantitate de memorie utilizată

➤ În alocarea memoriei se realizează un compromis:

- aliniere – reduce gradul de utilizare a memoriei
- renunțarea la aliniere – conduce la acces ineficient la nivelul componentelor

➤ În situația în care factorul de umplere este 0.5

- există posibilitatea ca în cadrul aceleiași unități de informație să se păstreze mai multe componente, în urma unui proces de **împachetare**
- accesul la o componenta a unei structuri împachetate presupune atât determinarea adresei j a unității de informație, cât și adresa relativă k în cadrul unității

➤ Soluție:

- păstrarea structurilor împachetate
- despachetarea lor înainte de utilizare

Reprezentarea articolelor

- Metoda de structurare fiind prin agregarea unor componente de tipuri diferite conduce la un calcul al adresei fiecărei componente funcție de următoarele elemente:
 - adresa de început a articolului
 - distanța relativă (deplasament, offset, adresă relativă) k_i a componentei i față de începutul articolului, măsurată în unități de informație
 - s_j = dimensiunea în unități de informație a unei componente j
 - $\Rightarrow k_i = s_1 + s_2 + s_3 + \dots + s_{i-1}$
- Identificatorii componentelor, cunoscuți încă în faza de compilare
 - reprezintă în fapt deplasamentele (adresele relative) asociate fiecărei componente în cadrul articolului
 - acest mod de acces rezolvă și problema legată de împachetare

Reprezentarea mulțimilor

- Mulțimea se reprezintă în memorie printr-o construcție numită **funcție caracteristică** C_m reprezentând un tablou unidimensional de valori logice în care componenta i va specifica prezența sau absența valorii i în mulțime
- $C_m = \text{card}(T_0)$
- Reprezentarea valorilor logice ca bit
 - avantajează implementarea operatorilor pentru tipul mulțime prin suportul hardware (funcții logice SI, SAU, deplasări)
 - conduce la restricții în raport cu cardinalitatea tipului mulțime, funcție de lungimea vectorului binar utilizat în reprezentare

1.4.6 Structura de date secvență. TDA secvență

- Structurile de date prezentate (tablou, articol, mulțime)
 - sunt structuri de date statice (dimensiune de memorie fixă, cunoscută încă în faza de compilare) și au cardinalitate finită
- Structura de date secvență și structurile dinamice (liste, arbori, grafuri)
 - nu se pot încadra într-o dimensiune finită a cardinalității și necesită o abordare specifică
- Definiția recursivă: O structura secvență, având tipul de baza T_0 , se definește
 - fie drept o secvență vidă
 - fie drept o concatenare a unei secvențe având tipul de bază T_0 cu o valoare a tipului T_0
- Notatii:
 - $S_0 = \langle \rangle$
 - $S_i = \langle S_{i-1}, s_i \rangle, s_i \in T_0$
- Fiecare valoare a tipului secvență conține un număr finit de componente
 - dar acest număr este nemărginit, putându-se construi o secvență mai lungă

1.4.6 Structura de date secvență. TDA secvență

➤ Datorită cardinalității infinite

- volumul de memorie necesar reprezentării nu poate fi cunoscut în timpul compilării
- e necesar un mecanism de **alocare dinamică** a memoriei în timpul execuției

➤ Dacă un limbaj de programare dispune de funcții sistem care permit:

- **alocarea** și eliberarea memoriei
- legarea (înlănțuirea) și referirea dinamică a componentelor

atunci cu ajutorul instrucțiunilor limbajului pot fi gestionate structuri dinamice (structuri avansate de date)

➤ Maniera de definire a structurii secvență o încadrează în categoria structurilor de date avansate

➤ Alegerea unei reprezentări potrivite, cât și a unui set specific de operatori, permite includerea acestei structuri de date în categoria structurilor fundamentale, având operatorii direct implementați în limbaj

Structura fișier secvențial (fișier)

- Este un caz particular de secvență rezultat în urma restrângerii setului de operatori astfel încât este posibil doar **accesul secvențial** la componentele structurii (secvenței)
- Caracteristici:
 - structură omogenă
 - numărul componentelor, numit și lungime a secvenței, nu este cunoscut
 - cardinalitate infinită (o maniera de implementare constă în înregistrarea secvenței pe suporturi de memorie externă, reutilizabilă)
 - structura ordonată; ordonarea elementelor este stabilită de ordinea în timp a adăugării componentelor
 - în fiecare moment este accesibilă o singură componentă, numită componentă curentă, precizată printr-un indicator (pointer) asociat secvenței
 - indicatorul (pointerul) avansează secvențial, după fiecare operație executată asupra secvenței (fișierului)
 - uzual i se asociază un operator care semnalează sfârșitul fișierului (EOF)

TDA secvență (fișier secvențial)

➤ MM

- secvență omogenă de elemente (unic tip constitutiv, numit și tip de bază)
- un indicator asociat secvenței, indică spre următorul element la care se poate realiza accesul

➤ Notatii

- TipElement – tip de bază (nu poate fi la rândul său secvență)
- f – secvență; e – valoare de TipElement; b – valoare booleană

➤ Operatori

- Rescrie(f) – poziționează indicatorul la începutul secvenței și pregătește secvența pentru scriere
- DepuneSecventa (f, e) – pentru o secvență deschisă în regim de scriere, operatorul copiază valoarea e în elementul următor al secvenței și avansează indicatorul
- EOF(f) -> b
- FurnizeazaSecventa(f, e) – returnează în e valoarea următorului element al secvenței și avansează indicatorul; secvența trebuie deschisă în regim de consultare
- Adauga(f) – deschide secvența în regim de scriere, poziționând indicatorul la sfârșit

Vă mulțumesc!

Structuri de date și algoritmi

upT Universitatea
Politehnica
Timișoara

P-ța Victoriei nr. 2
RO 300006 - Timișoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 4

Ș.I. dr.ing. Adriana ALBU

adriana.albu@upt.ro

<http://www.aut.upt.ro/~adrianaa>

2. Noțiuni despre algoritmi

3. Tehnici de sortare (partea întâia)

2 Noțiuni despre algoritmi / 2.1 Definiții, caracteristici

- **Algorithm** = metoda de rezolvare a unei probleme (clase de probleme)
- Elemente constitutive: Orice algoritm ia în considerare un
 - set de **date inițiale**, prelucrate pe baza unui
 - set de **reguli de transformare**, cu scopul obținerii unui
 - set de **date finale**
- Algoritmii diferă în mod esențial prin natura și ordinea în care se executa relațiile de transformare
- Caracteristici:
 - generalitatea
 - finitudinea
 - unicitatea – determinism

Algoritmii sunt **abstractizări** (elemente abstracte) care prin implementare devin program (proceduri, funcții)

2 Noțiuni despre algoritmi / 2.2 Analiza algoritmilor

➤ Analiza algoritmilor urmărește două obiective:

- **precizarea predictivă a comportamentului** unui algoritm în timpul execuției sale
- **compararea** unor algoritmi și **ierarhizarea** acestora în raport cu **performanțele** lor

➤ Se bazează pe ipoteza că toate sistemele de calcul pe care se execută algoritmii sunt convenționale

- execută la un moment dat o singură instrucțiune, care durează un timp finit
- timpul total al execuției = suma timpilor necesari execuției tuturor instrucțiunilor

➤ Obiectivele analizei se realizează prin:

- **determinarea operațiilor** realizate în cadrul algoritmului și a **costurilor** lor relative, exprimate în **timp de execuție** (costul unei instrucțiuni poate fi cunoscut aprioric, cu excepția operațiilor cu șiruri de caractere și a anumitor bucle dinamice)
- **aprecierea performanței** algoritmului prin execuția sa efectivă, utilizând seturi speciale de date de intrare, astfel încât să fie evidențiat comportamentul algoritmului, dar și extremele acestui comportament (cel mai defavorabil, cel mai favorabil)

2 Noțiuni despre algoritmi / 2.3 Notatii asimptotice

Analiza algoritmilor se desfășoară în două etape

Analiza apriorică

- determinarea unei **funcții care mărginește asimptotic timpul de execuție** al algoritmului și care depinde de anumiți parametri relevanți (dimensiune date de intrare, dimensiune tablou etc).
- utilizarea notațiilor asimptotice permite **determinarea ordinului de mărime al timpului de execuție**

Analiza efectuată posterior implementării

- realizarea profilului algoritmului, respectiv **execuția algoritmului** pentru diferite seturi de date de intrare și **măsurarea timpului** de execuție
- această analiză va confirma sau va infirma rezultatele analizei apriorice

2 Noțiuni despre algoritmi / 2.3 Notatii asimptotice

➤ **Ordinul de mărime** al timpului de execuție

- caracteristică a eficienței algoritmului
- permite compararea relativă a algoritmilor alternativi

➤ Studiul eficienței asimptotice a unui algoritm

- se realizează utilizând mărimi de intrare suficient de mari pentru a face relevant
 - ordinul de mărime al timpului de execuție
 - limita la care tinde timpul de execuție odată cu creșterea nelimitată a mărimilor de intrare
 - (ex. $1000n$ vs. n^2)

2.3 Notății asimptotice / Notăția Θ (theta)

- Prin definiție, fiind dată o funcție $g(n)$ prin notația $\Theta(g(n))$ se desemnează o mulțime de funcții, definită prin relația:
 - $\Theta(g(n)) = \{f(n): \text{există constante pozitive } c_1 > 0, c_2 > 0, n_0 > 0$
astfel încât $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
pentru orice $n \geq n_0\}$
- Reformulare: O funcție $f(n)$ aparține mulțimii $\Theta(g(n))$ dacă există constantele pozitive c_1 și c_2 astfel încât funcția $f(n)$ să poată fi cuprinsă între $c_1 g(n)$ și $c_2 g(n)$ pentru un n suficient de mare
- Deși $\Theta(g(n))$ reprezintă o mulțime de funcții se utilizează frecvent notația:
 - $f(n) = \Theta(g(n))$
- $g(n)$ reprezintă o **margină asimptotică strânsă**
- Definiția notației (noțiunii) Θ este validă doar dacă $f(n)$ este o funcție asimptotic crescătoare în raport cu n și dacă n ia valori suficient de mari

2.3 Notății asimptotice / Notăția Θ (theta)

- $f(n)$ – funcție polinomială (asimptotic crescătoare pentru n suficient de mare)
- $g(n)$ poate fi estimată din $f(n)$ dacă se neglijează termenii de ordin inferior și se alege pentru c_1 o valoare $<$ coeficientul termenului de ordinul cel mai mare, iar pentru c_2 o valoare mai mare

➤ Ex:

- $f(n) = n^2 + 100n + \log_{10}n + 1000$
- rata de creștere a termenilor lui $f(n)$ în raport cu n

n	$f(n)$	n^2	$100n$	$\log_{10}n$	1000
1	1101	1	100	0	1000
10	2101	100	1000	1	1000
100	21002	10000	10000	2	1000
1000	1101003	1000000	100000	3	1000
10000	101001004	100000000	1000000	4	1000

- $f(n) = an^3 + bn^2 + cn + d \Rightarrow f(n) = \Theta(n^3)$
- funcție polinomială de grad 0 $\Rightarrow \Theta(1)$

2.3 Notății asimptotice / Notăția O (O mare)

- Precizează **marginea asimptotic superioară**, într-o manieră similară notației Θ :
 - $O(g(n)) = \{f(n) : \text{există constante pozitive } c > 0 \text{ și } n_0 > 0$
astfel încât $0 \leq f(n) \leq cg(n)$
pentru orice $n \geq n_0\}$
- Notăția O are drept scop stabilirea unei ordonări relative a funcțiilor
 - ceea ce se compară este rata relativă de creștere a funcțiilor și nu valorile lor în anumite puncte
- $f(n) = O(g(n))$; rata de creștere a lui $f(n)$ este \leq cu cea a lui $g(n)$; uzual se alege marginea cea mai apropiată
- Notăția Θ este mai restrictivă decât notația O; $\Theta(g(n))$ este inclusă în $O(g(n))$; Θ nu se referă la orice intrare
- Notăția **O descrie cazul de execuție cel mai defavorabil** și, prin implicație, mărginește comportamentul algoritmului pentru orice intrare

2.3 Notății asimptotice / Notăția O (O mare)

➤ Ex.

- $1000n > n^2$ pentru valori mici ale lui n , dar n^2 crește cu o rată mult mai mare decât $1000n$; n^2 are un ordin de mărime mai mare decât $1000n$

➤ Ex.

- $f(n)=2n^2+3n+1=O(n^2)$; $2n^2+3n+1 \leq cn^2$; \Rightarrow o mulțime de perechi c și n_0 ; $f(n)$ și $g(n)$ cresc cu aceeași rată

➤ Pentru un n dat, timpul de execuție depinde de configurația particulară a intrării de dimensiune n

➤ Timpul de execuție (TE) nu este o funcție de n ; exprimarea „TE este $O(g(n))$ ” referă cazul cel mai defavorabil al TE

➤ Uzual definiția lui O nu se aplică în mod formal, ci se utilizează rezultate cunoscute care permit ordonarea funcțiilor după rata de creștere:

- $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n) < O(\log^2 n) < O(10^n) < O(n^m)$

2.3 Notății asimptotice / Notăția O (O mare)

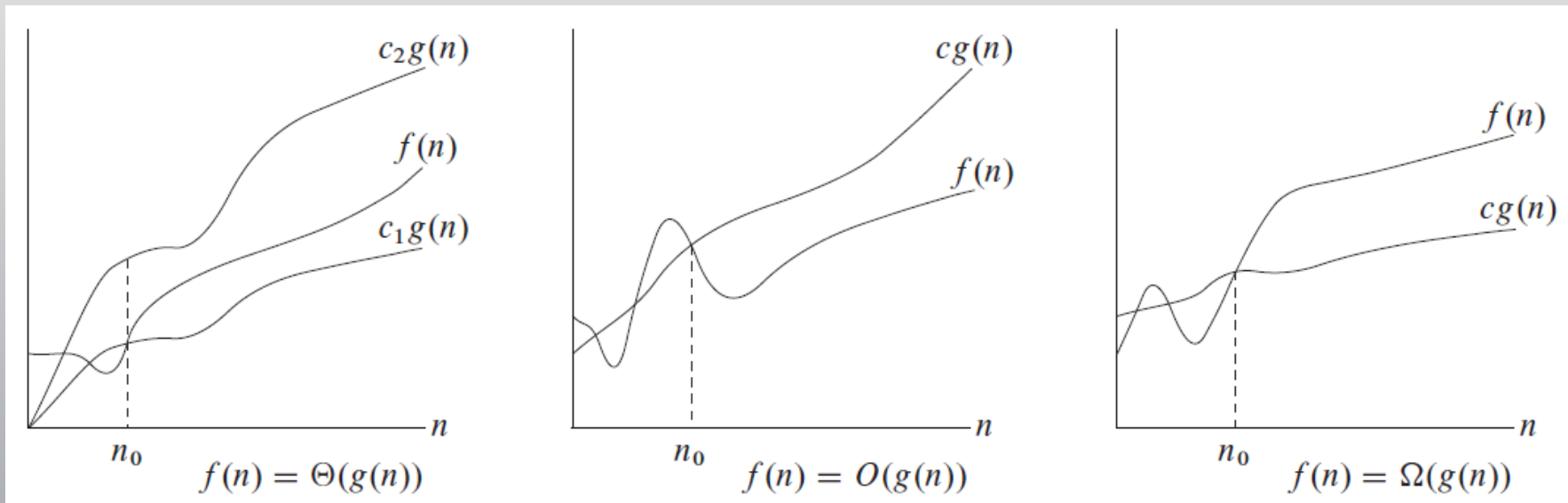
➤ Proprietăți ale notației O:

- R1. $O(k) < O(n)$ pentru orice constantă k
- R2. ignorarea constantelor: $kf(n) = O(f(n))$
- R3. tranzitivitate: $f(n) = O(g(n))$ și $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
- R4. suma
 - $f_1(n) = O(f(n))$
 - $f_2(n) = O(g(n))$
 - $f_1(n) + f_2(n) = \max(O(f(n)), O(g(n)))$
- R5. produsul
 - $f_1(n) = O(g_1(n))$
 - $f_2(n) = O(g_2(n))$
 - $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$
- R6. Dacă $f(n)$ este un polinom de grad k , atunci $f(n) = O(n^k)$
- R7. $\log^k n = O(n)$ pentru orice constantă k
- R8. $\log_a n = O(\log_b n)$ pentru orice $a > 1$ și $b > 1$
- R9. $\log_a n = O(\lg n)$ unde $\lg n = \log_2 n$

2.3 Notății asimptotice / Notăția Ω (omega)

- Precizează marginea asimptotică inferioară
 - $\Omega(g(n)) = \{f(n) : \text{există constante pozitive } c > 0 \text{ și } n_0 > 0$
astfel încât $0 \leq cg(n) \leq f(n)$
pentru orice $n \geq n_0\}$
- Utilizare: $f(n) = \Omega(g(n))$, pentru a preciza **cazul cel mai favorabil** al execuției unui algoritm
- Prin implicație, $\Omega(g(n))$ va mărgini inferior TE pentru orice intrare arbitrară a algoritmului indiferent de dimensiunea lui n și structura intrării
- $f(n) = \Theta(g(n))$ doar dacă $f(n) = O(g(n))$ și $f(n) = \Omega(g(n))$

2.3 Notății asimptotice



2.3 Notății asimptotice / Notăția o (o mic)

➤ Precizează marginea asimptotică superioară **lejeră**

- $o(g(n)) = \{f(n) : \text{pentru orice constantă } c > 0 \text{ există o constantă } n_0 > 0$
astfel încât $0 \leq f(n) < cg(n)$
pentru orice $n \geq n_0\}$

➤ $f(n) = o(g(n))$ dacă $f(n) = O(g(n))$ și $f(n) \neq \Theta(g(n))$

➤ În cazul notației o (o mic) funcția f devine nesemnificativă în raport cu g când $n \rightarrow \text{infinit}$

➤ $\lim_{n \rightarrow \text{infinit}} f(n)/g(n) = 0$ pentru $n \rightarrow \text{infinit}$

2.3 Notății asimptotice / Notăția ω (omega mic)

➤ Precizează marginea asimptotică inferioară **lejeră**

- $\omega(g(n)) = \{f(n) : \text{pentru orice constantă } c > 0 \text{ există o constantă } n_0 > 0$
astfel încât $0 \leq cg(n) < f(n)$
pentru orice $n \geq n_0\}$

➤ $f(n)$ devine din ce în ce mai semnificativă față de $g(n)$ pe măsura ce n crește

➤ $\lim f(n)/g(n) = \text{infinit}$ pentru $n \rightarrow \text{infinit}$

Proprietati generale ale notațiilor asimptotice

➤ Tranzitivitate

- $f(n)=\Theta(g(n))$ și $g(n)=\Theta(h(n)) \rightarrow f(n)=\Theta(h(n))$
- $f(n)=O(g(n))$ și $g(n)=O(h(n)) \rightarrow f(n)=O(h(n))$
- ...

➤ Reflexivitate

- $f(n)=O(f(n))$
- $f(n)=\Theta(f(n))$

➤ Simetria

- $f(n)=\Theta(g(n)) \rightarrow g(n)=\Theta(f(n))$

➤ Simetria transpusa

- $f(n)=O(g(n)) \quad g(n)=\Omega(f(n))$
- $f(n)=o(g(n)) \quad g(n)=\omega(f(n))$

Aprecierea timpului de execuție al algoritmilor

- **Notăția asimptotică O este utilizată pentru aprecierea timpului de execuție al unui algoritm** (timpul de execuție absolut, complexitate temporală)
- Un algoritm a cărui complexitate temporală este spre exemplu $O(n^2)$ va rula ca program întotdeauna în $O(n^2)$ unități de timp, indiferent de natura implementării sale
- **Timpul de execuție al unui program depinde de:**
 - dimensiunea și structura datelor de intrare
 - caracteristicile sistemului de calcul
 - eficiența codului generat

Aprecierea timpului de execuție al algoritmilor

➤ Considerații:

- fiecare instrucțiune se execută în același interval de timp (unit. de timp)
 - numărul de ciclări pentru instrucțiunile de ciclare este maxim
 - instrucțiunile condiționale se execută pe ramura cea mai lungă
 - instrucțiunile consecutive produc adunarea timpilor de execuție
 - instrucțiunile de ciclare imbricate → produs
- Cu regulile precizate anterior, timpul de execuție este **supraestimat** în unele situații, dar nu este niciodată **subestimat**
- O strategie generală pentru calculul timpului de execuție al algoritmilor impune analiza de la „interior spre exterior”
- de exemplu dacă algoritmul conține apeluri de funcții, timpul de execuție al acelor funcții trebuie analizat mai întâi
- În cazul apelurilor recursive se încearcă fie transformarea recursivității într-o iterație (structură de buclă), fie găsirea unei relații de recurență

Profilul performanței algoritmului

- Profilarea este o activitate prin care se determină **exact** timpii de execuție a algoritmului pentru o anumită implementare și pentru seturi diferite de date de intrare
- Profilul performanței **confirmă** aprecierea timpului de execuție realizată în **analiza apriorică**
 - pentru aceasta se vor furniza seturi de date de intrare de **dimensiuni** din ce în ce mai mari, respectiv seturi de date de intrare corespunzătoare cazurilor „**cel mai favorabil**” și „**cel mai defavorabil**”
- Prin compararea rulării pe același sistem de calcul a mai multor algoritmi care realizează aceeași funcție se face o analiză a **performanței algoritmilor**
- Prin compararea rulării aceluiași algoritm pe sisteme de calcul diferite se poate face o analiză a **performanței sistemului**

2. Noțiuni despre algoritmi

3. Tehnici de sortare (partea întâia)

3.1 Conceptul de sortare / Sortarea tablourilor

- Sortarea este o activitate fundamentală cu caracter universal
- Prin sortare se înțelege **ordonarea după un criteriu precizat** a unei mulțimi de elemente, cu scopul facilitării operației de căutare a unui element dat
- Algoritmii de sortare
 - subliniază **interdependența** între structura de date și modul de proiectare a algoritmului
 - permit **analiza comparativă** a algoritmilor, cu evidențierea avantajelor și dezavantajelor asociate diferitelor implementări
- Structura de date supusă sortării:
 - tablou de elemente (structurate sau nu, cu sau fără câmp cheie)
- Operația de sortare constă în permutarea elementelor tabloului astfel încât să fie satisfăcută relația de ordonare (ordine crescătoare, descrescătoare, cu sau fără egalitate)
 - ex: $a_1 \leq a_2 \leq a_3 \leq \dots \leq a_n$

3.1 Conceptul de sortare / Sortarea tablourilor

- O metodă de sortare este **stabilă** dacă în urma procesului de sortare elementele identice (cheile identice, dacă se utilizează un câmp cheie) nu își schimbă ordinea relativă
- Funcție de locul în care se afla elementele de sortat rezultă următoarele categorii:
 - sortare **internă** – elementele de sortat sunt stocate în memoria internă
 - sortare **externă** – elementele de sortat sunt stocate pe suport extern
- O **cerință impusă** algoritmilor de sortare legată de eficiență este utilizarea în procesul de sortare a unei zone de memorie cât mai redusă
- Algoritmii care nu utilizează zone suplimentare de memorie realizează o sortare „in situ” (pe loc)

3.1 Conceptul de sortare / Sortarea tablourilor

➤ Elementele care influențează **performanța** algoritmilor de sortare, respectiv **timpul de execuție** a acestora sunt:

- numărul de comparații C
- numărul de mișcări M

dependente la rândul lor de **dimensiunea n** a tabloului (nr. elementelor de sortat)

➤ Funcție de complexitatea temporală, metodele de sortare sunt:

- **directe**
 - simple, potrivite pentru a explicita mecanismele de sortare $\rightarrow O(n^2)$
- **avansate**
 - conduc prin implementare la mai puține operații
 - sunt mai complexe în detalii
 - își dovedesc utilitatea pentru valori mari ale lui $n \rightarrow O(n \lg n)$

3.1 Conceptul de sortare / Sortarea tablourilor

➤ **Metodele de sortare directe** au la bază **principiile** (pt. ordonare crescătoare):

- **insertie** – se ia un element de sortat la întâmplare (până la epuizarea elementelor supuse sortării) și se inserează în structura sortată la locul potrivit
- **selectie** – se parcurg toate elementele, se alege cel mai mic și se plasează pe prima poziție; se reia algoritmul pentru cele rămase, până la epuizarea elementelor supuse sortării
- **schimburi** (interschimbare) – în toate perechile de elemente care se pot imagina, se plasează elementul cel mai mic pe primul loc

3.2 Tehnica sortării prin inserție

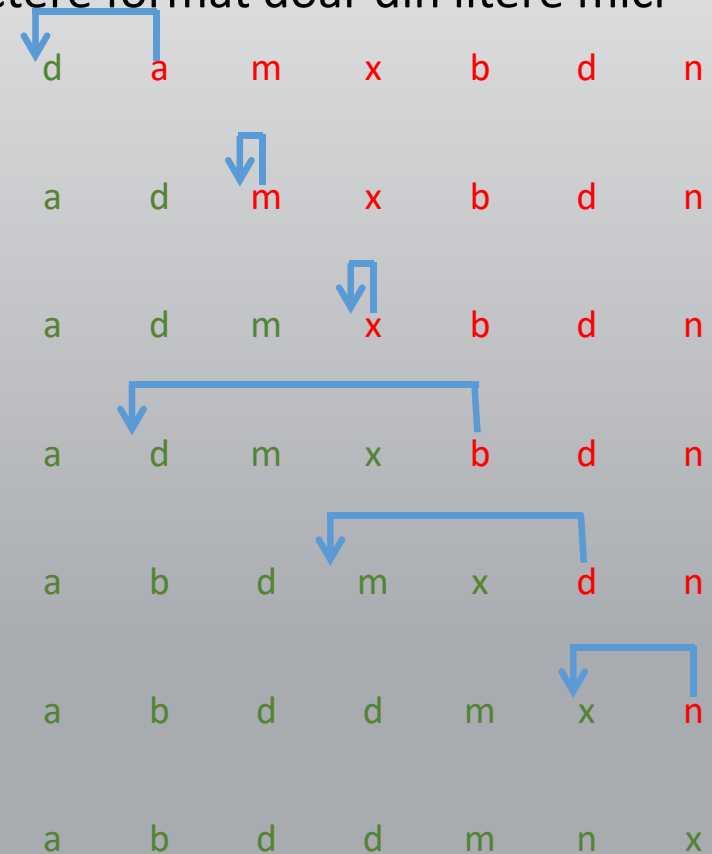
- Elementele de sortat $a_1, a_2, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n$
 - sunt în mod conceptual divizate în 2 secvențe:
 - secvența destinație a_1, a_2, \dots, a_{i-1}
 - secvența sursă a_i, a_{i+1}, \dots, a_n
- Selecția locului de inserție pentru elementul a_i se va face parcurgând secvența destinație de la dreapta la stânga, oprirea realizându-se pe primul element $a_j \leq a_i$ sau pe prima poziție, dacă un astfel de element a_j nu este găsit
- Simultan cu parcurgerea secvenței destinație pentru căutarea locului de inserție se face și deplasarea la dreapta a fiecărui element testat, până la îndeplinirea condiției

3.2 Tehnica sortării prin inserție

➤ Exemplu:

- sortarea în ordine crescătoare a unui șir de caractere format doar din litere mici

```
void insertSort(char *s, int n){  
    int i, j;  
    char t;  
    for (i=1; i<n; ++i){  
        t=s[i];  
        j=i-1;  
        while(j>=0 && t<s[j]){  
            s[j+1]=s[j];  
            j--;  
        }  
        s[j+1]=t;  
    }  
}
```



3.2 Tehnica sortării prin inserție – analiză

- Pentru pasul i al ciclului for numărul de comparații asociat C_i depinde de ordinea inițială a elementelor, fiind:
 - cel puțin 1
 - cel mult $i-1$
 - presupunând că toate permutările sunt în mod egal posibile, C_i poate fi considerat pentru cazul mediu ca fiind $i/2$
 - Ciclul for se va repeta de $n-1$ ori pentru $i=1, \dots, n-1$, rezultând astfel C_{\min} , C_{\max} și C_{med}
- Numărul de mișcări M_i realizat în pasul i al ciclului for are două componente:
 - C_i componenta determinată de bucla while
 - 2 sau 3 atribuiri – cele exterioare ciclului while
- M_{\min} , M_{\max} și M_{med} se calculează considerând cele $n-1$ repetări ale ciclului for
- Sortarea prin inserție este stabilă
- **Valorile maxime** pt. C și M se obțin când șirul inițial este ordonat invers
- **Performantele sunt scăzute** deoarece deplasarea elementelor se realizează de fiecare dată cu o singură poziție

3.2 Tehnica sortării prin inserție

Algoritm de inserție binară

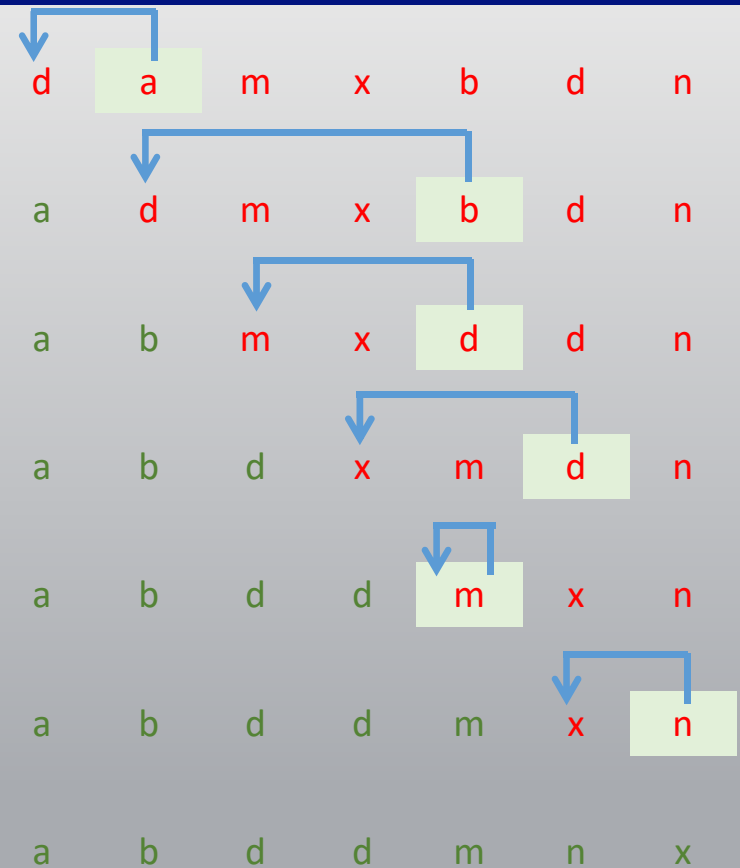
- Secvența destinație este deja o structură ordonată
 - => căutarea locului inserției se poate face aplicând tehnica de **căutare binară**
- Căutarea binară **reduce numărul de comparații**:
 - într-un interval de i chei, locul pentru inserarea elementului va fi găsit după $\log_2 i$ pași (rotunjit superior)
 - pentru cele $n-1$ repetări ale ciclului for: $C = O(n \log_2 n)$
- Numărul mișcărilor însă rămâne nemodificat: $O(n^2)$
- Performanța algoritmului de inserție binară **rămâne** în domeniul $O(n^2)$
 - costul operației de interschimbare este mai mare decât cel necesar comparației a două elemente
 - **performantă** atunci când șirul inițial este ordonat invers
 - **performanțe scăzute** față de inserția prin căutare liniară dacă șirul inițial este deja sortat

3.3 Tehnica sortării prin selecție

- Principiul constă în căutarea într-o secvență a elementului cu cheie minimă (a_k) și plasarea lui pe prima poziție (a_{i+1}) prin interschimbare
- Procedurul se reia pentru cele $n-1, n-2, \dots, 1$ elemente rămase
- $a_1, a_2, \dots, a_i, a_{i+1}, \dots, a_k, \dots, a_n$
- Este o metodă opusă ca principiu sortării prin inserție, căutarea executându-se în secvența sursă

3.3 Tehnica sortării prin selecție

```
void selectSort(char *s, int n){
    char t;
    int i, j, k;
    for (i=0; i<n-1; ++i){
        k=i;
        t=s[i];
        for(j=i+1; j<n; ++j){
            if(s[j]<t){
                k=j;
                t=s[j];
            }
        }
        s[k]=s[i];
        s[i]=t;
    }
}
```



3.3 Tehnica sortării prin selecție – analiză

- În pasul i al ciclului for se vor executa $i - 1$ comparații
 - numărul comparațiilor C este **independent de ordinea inițială**, metoda comportându-se mai puțin natural decât sortarea prin inserție
- Ciclul for se va repeta pentru $i = 1, 2, \dots, n-1$, rezultând astfel:
 - $C_{\min} = C_{\max} = C = (n-1)(n-2)/2 \Rightarrow O(n^2)$
- Numărul mișcărilor M este de cel puțin 3 pentru fiecare valoare a lui i
 - acest minim poate să apară dacă elementele sunt deja sortate
 - $M_{\min} = 3(n-1)$
 - dacă elementele sunt inițial în ordine inversă, numărul de mișcări M este maxim:
 - $M_{\max} = n^2/4$ (rotunjit inferior) + $3(n-1)$
 - M_{med} se obține printr-un raționament probabilistic care ia în considerare toate permutările posibile asociate unui număr de elemente
 - pentru fiecare permutare numărul mișcărilor este determinat de numărul elementelor având proprietatea de a fi mai mici decât termenii precedenți

3.3 Tehnica sortării prin selecție – analiză

- În procesul de sortare se parcurg secvențe de lungimi $n, n-1, n-2, \dots, 1$
 - în urma însumării mișcărilor de pe fiecare secvență va rezulta:
 - $M_{\text{med}} = n \ln n \Rightarrow O(n \ln n)$
- Performanța în raport cu numărul de mișcări situează sortarea prin selecție pe **primul loc** în ierarhia algoritmilor de sortare directă
- O varianta îmbunătățită a sortării prin selecție se obține dacă se determină doar poziția minimului, eliminându-se astfel atribuirea aferentă instrucțiunii condiționale \Rightarrow sortarea prin selecție performantă:
 - $M_{\text{min}} = 3(n-1)$

3.4 Tehnica sortării prin interschimbare

➤ Principiul metodei:

- se compară pe rând și se interschimbă între ele toate elementele alăturate, până când tabloul este în întregime sortat

➤ Sortarea se va realiza într-o manieră ordonată, executând **treceți repetate** prin tablou

- în cazul sortării crescătoare, la fiecare trecere, cel mai mic element al secvenței procesate se va deplasa spre capătul din stânga

➤ Pentru o mulțime de n elemente supuse sortării sunt necesare $n-1$ treceți prin tablou, asigurate de un for exterior

➤ Bucula interioară va executa comparațiile și interschimbările, dacă acestea sunt necesare

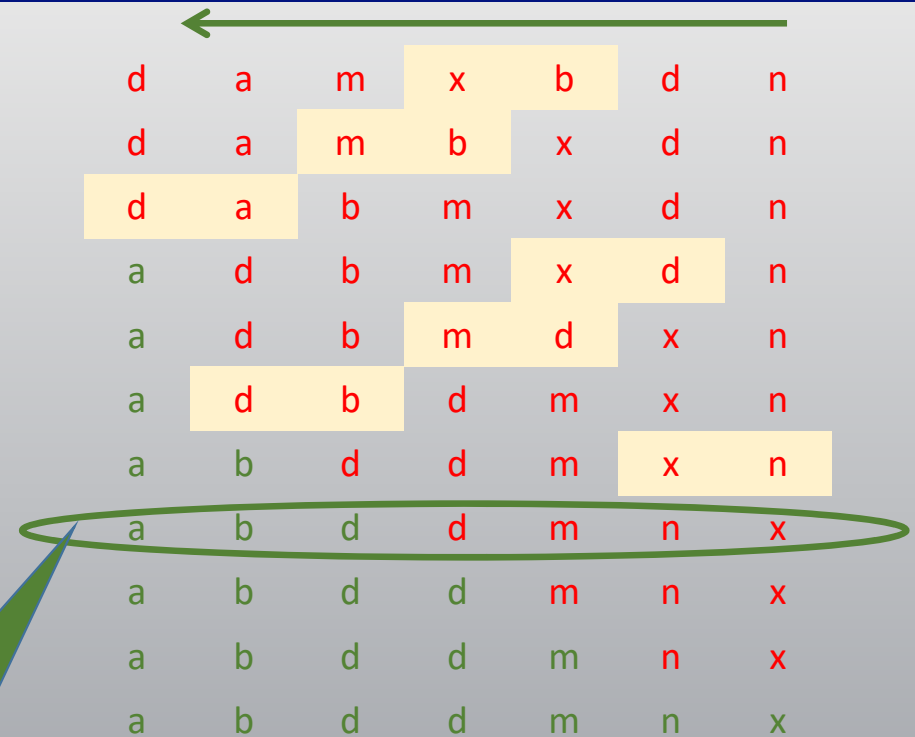
➤ Se pot evidenția și aici cele două secvențe conceptuale, sursă și destinație

➤ În pasul k se vor compara de fiecare dată $n-k$ elemente

- după k pași, k elemente vor fi deja ordonate
- tabloul va fi în întregime ordonat după $n-1$ pași

3.4 Tehnica sortării prin interschimbare

```
void bubbleSort(char *s, int n){  
    int i, j;  
    char t;  
    for(i=1; i<n; ++i)  
        for(j=n-1; j>=i; --j){  
            if(s[j-1]>s[j]){  
                t=s[j-1];  
                s[j-1]=s[j];  
                s[j]=t;  
            }  
        }  
}
```



Deși în acest punct
tabloul este sortat,
algoritmul continuă
verificările

3.4 Tehnica sortării prin interschimbare

- În multe cazuri sortarea se termină înainte de a epuiza toate reluările precizate prin bucla for exterioară
 - o îmbunătățire se obține dacă reluarea ciclului este condiționată de efectuarea unei schimbări în pasul precedent
- Altă îmbunătățire se obține dacă se memorează indicele k al ultimei schimbări
 - toate elementele aflate sub acest indice fiind deja sortate
- Algoritmul prezintă o asimetrie în comportament:
 - un element „ușor” (valoare mică în cazul sortării crescătoare), aflat pe poziția inferioară, ajunge la locul lui într-o singură trecere
 - un element „greu” (valoare mare în cazul sortării crescătoare), aflat pe prima poziție a tabloului are nevoie de $n-1$ treceri pentru a ajunge la locul potrivit
- Ex: 83 12 18 22 24 04

3.4 Tehnica sortării prin interschimbare

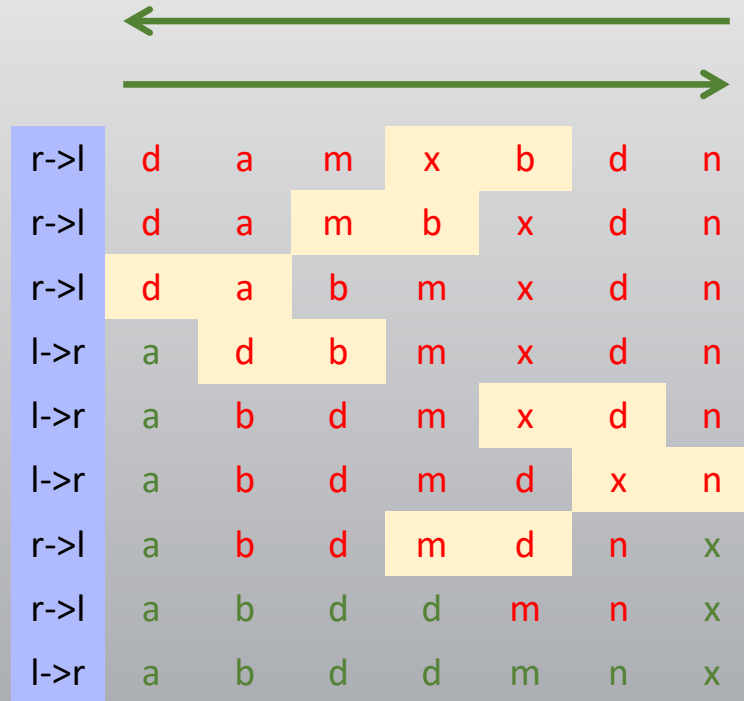
➤ Sortare amestecată

```
void shakerSort(char *s, int n){
    int j, k, l, r;
    char t;
    l=1; r=n-1; k=n-1;
    do{
        for(j=r; j>=l; --j){
            if(s[j-1]>s[j]){
                t=s[j-1]; s[j-1]=s[j]; s[j]=t;
                k=j;
                /*memoreaza ultima inversare*/
            }
        }
    }
```

```
        l=k+1;
        for(j=l; j<=r; ++j){
            if(s[j-1]>s[j]){
                t=s[j-1];
                s[j-1]=s[j];
                s[j]=t;
                k=j;
            }
        }
        r=k-1;
    }while(l<=r);
}
```

3.4 Tehnica sortării prin interschimbare

➤ Sortare amestecată



3.4 Tehnica sortării prin interschimbare – analiză

➤ Bubblesort:

- Numărul de comparații C_i efectuate în pasul i este $i-1$:
 - $C = 1+2+3+\dots+(n-2) = (n^2-3n+2)/2 \Rightarrow O(n^2)$
- Numărul de mișcări M depinde de starea de ordonare inițială:
 - $M_{\min} = 0$
 - $M_{\max} = 3C = 3(n^2-3n+2)/2$
 - $M_{\text{med}} = 3(n^2-3n+2)/4$

➤ Shakersort:

- Se reduce numărul de comparații:
 - $C_{\min} = n-1$; $C_{\text{med}} = 1/2(n^2 - n(k + \ln n)) \Rightarrow O(n^2)$
- Numărul de interschimbări rămâne același $\Rightarrow O(n^2)$
- Distanța parcursă de fiecare element este în medie $n/3$ locuri

Vă mulțumesc!

Structuri de date și algoritmi

upT Universitatea
Politehnica
Timișoara

P-ța Victoriei nr. 2
RO 300006 - Timișoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 5

Ș.I. dr.ing. Adriana ALBU

adriana.albu@upt.ro

<http://www.aut.upt.ro/~adrianaa>

3. Tehnici de sortare (partea a doua)

3.5 Tehnica sortării prin diminuarea incrementului – Shell

➤ Generalizează tehnica sortării prin inserție

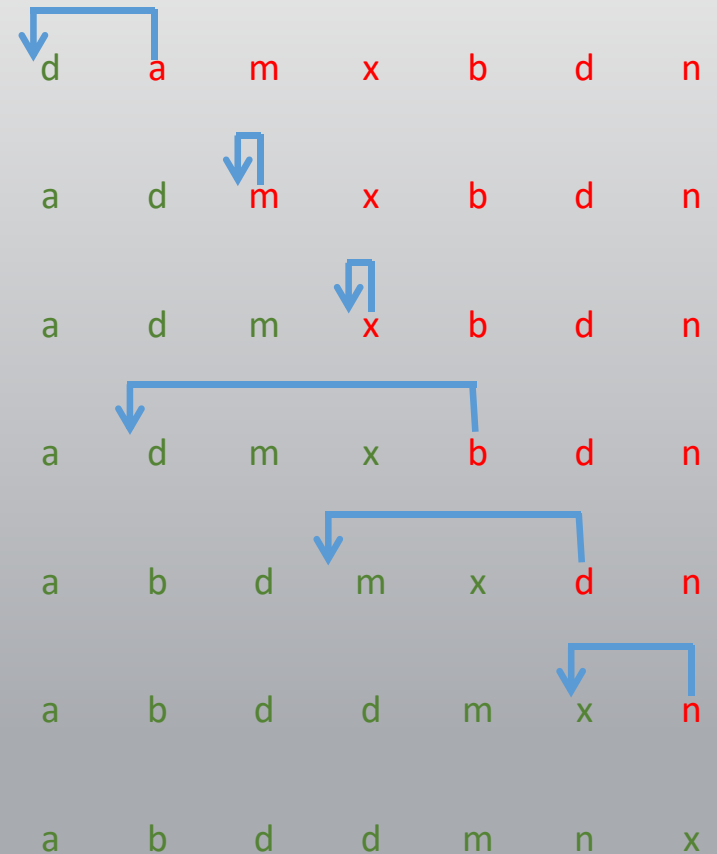
➤ Recapitulare – tehnica sortării prin inserție:

➤ Sortarea prin inserție

- nu e performantă când elemente apropiate ca valoare sunt depărtate ca poziționare în tablou

➤ Shell Sort

- reduce distanța între elemente de valori apropiate
- execută un număr mai mic de mutări



3.5 Tehnica sortării prin diminuarea incrementului – Shell

➤ **Principiul** metodei de sortare prin inserție cu diminuarea incrementului:

- se realizează **sortări prin inserție repetate** asupra elementelor tabloului
- se folosește un „**pas de sortare**” – **incrementul** secvenței supusă sortării
 - un **sir descrescător** de valori

➤ „Pasul de sortare”

- precizează **distanța** între elementele care alcătuiesc secvența supusă sortării
- este posibilă orice secvență descrescătoare de incrementi
 - cu condiția ca **ultimul increment să fie 1**
- notații: h_1, h_2, \dots, h_t ; $h_t=1$; $h_k > h_{k+1}$; (exemplu secvență incrementi: 3, 2, 1)
- Nu este indicat ca incrementii să fie puteri ale aceleiași baze, pentru a nu se prelucra aceleași elemente de mai multe ori (4, 2, 1)

➤ Fiecare sortare succesivă profită de sortările anterioare

3.5 Tehnica sortării prin diminuarea incrementului – Shell

➤ Fiecare *sortare-h* implică

- relativ puține elemente
- elemente care sunt deja sortate

➤ Nu este o sortare stabilă – poate modifica ordinea elementelor cu valori egale

➤ Garantează că o secvență *sortată-k*, rămâne *sortată-k* și după o *sortare-l*

➤ O eficiență sporită pentru sortarea shell se obține dacă între diferitele secvențe de parcurgere au loc cât mai puține interacțiuni

➤ Sunt recomandate diferite secvențe de incrementi:

- Knuth

- $h_{k-1} = 3h_k + 1; h_t = 1; t = \log_3 n - 1$
- $h_t=1, h_{t-1}=4, h_{t-2}=13, \dots$

- Hibbard

- $h_{k-1} = 2h_k + 1; h_t = 1; t = \log_2 n - 1$
- $1, 3, 7, 15, \dots, 2^k - 1$

$O(n^{5/4})$

3.5 Tehnica sortării prin diminuarea incrementului – Shell

```
void shellSort(char *s, int n){
    int i, j, k, w;
    char x;
    int h[3]={3,2,1};
    for(w=0;w<3;w++){
        k=h[w];
        for(i=k;i<n;++i){
            x=s[i];
            j=i-k;
            while(x<s[j]&& j>=0){
                s[j+k]=s[j];
                j=j-k;
            }
            s[j+k]=x;
        }
    }
}
```

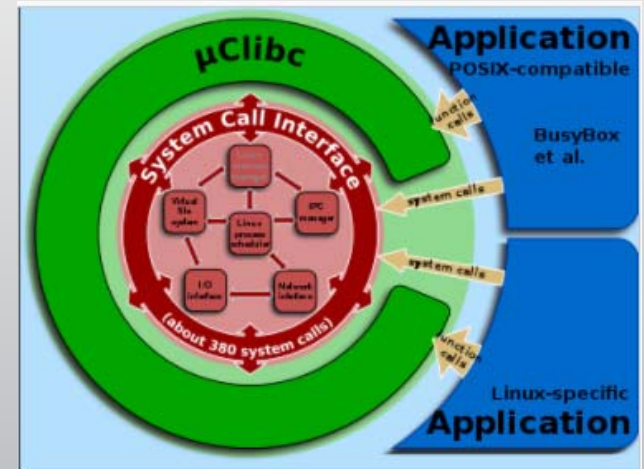
Exemplu: tablou: d, a, m, y, b, d, n, p, c, t
incrementi: 3, 2, 1

Trecere 1, Pas 3	3 sub-tablouri	d	a	m	y	b	d	n	p	c	t
		d	a	d	y	b	m	n	p	c	t
		d	a	d	n	b	m	y	p	c	t
		d	a	c	n	b	d	y	p	m	t
		d	a	c	n	b	d	t	p	m	y
Trecere 2, Pas 2	2 sub-tablouri	d	a	c	n	b	d	t	p	m	y
		c	a	d	n	b	d	t	p	m	y
		b	a	c	n	d	d	t	p	m	y
		b	a	c	d	d	n	t	p	m	y
		b	a	c	d	d	n	m	p	t	y
Tr. 3, Pas 1	1 sub-tablouri	b	a	c	d	d	n	m	p	t	y
		a	b	c	d	d	n	m	p	t	y
		a	b	c	d	d	m	n	p	t	y

3.5 Tehnica sortării prin diminuarea incrementului – Shell

➤ Aplicații care utilizează Shell Sort:

- **μClibc** – mică bibliotecă C standard
 - pentru sisteme de operare bazate pe Linux dedicate sistemelor embedded și dispozitivelor mobile
 - potrivit pentru microcontrolere
 - free, open-source
- **bzip2** – program de compresie a fișierelor
 - free, open-source

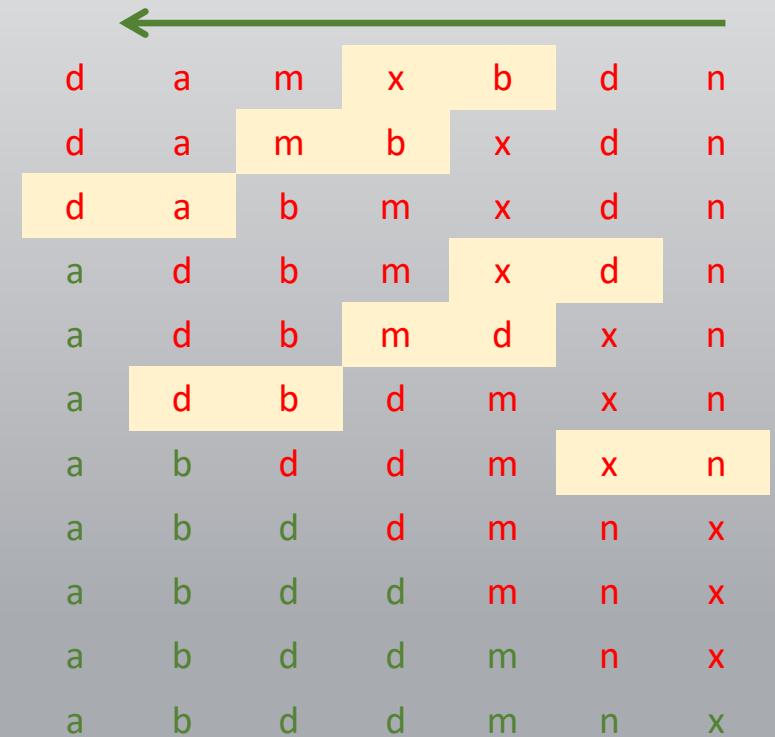


bzip2

3.6 Tehnica sortării prin partiționare – quickSort

➤ Creșterea eficienței metodei de sortare prin interschimbare se poate obține mărinđ distanța pe care se face deplasarea elementelor

➤ Recapitulare – tehnica sortării prin interschimbare:



3.6 Tehnica sortării prin partiționare – quickSort

➤ Principiul metodei (1):

- se consideră tabloul de sortat a_1, a_2, \dots, a_n
- se selectează un element oarecare x (pivot)
- se parcurge tabloul de la stânga la dreapta până la întâlnirea primului element $a_i > x$
- se parcurge tabloul de la dreapta la stânga până la întâlnirea primului element $a_j < x$
- se interschimbă a_i cu a_j și procesul continuă în aceeași manieră din punctul în care a rămas, până când parcurgerile se întâlnesc

repetă

- caută primul element $a[i] \geq x$ prin parcurgere stanga \rightarrow dreapta
- caută primul element $a[j] \leq x$ prin parcurgere dreapta \rightarrow stanga
- dacă $i \leq j$ atunci
 - interschimbă $a[i]$ cu $a[j]$

pană când parcurgerile se întâlnesc ($i > j$)

3.6 Tehnica sortării prin partiționare – quickSort

➤ Principiul metodei (2):

- în urma acestui proces tabloul este împărțit în două partiții:
 - partiția stângă cu elemente $< x$
 - partiția dreaptă cu elemente $> x$
- se aplică aceeași procedură celor două partiții astfel rezultate
 - până la partiții banale de câte 1 element

3.6 quickSort – alegerea pivotului

➤ primul element

- este o alegere acceptabilă dacă elementele se află inițial într-o ordine aleatorie
- dacă tabloul este deja sortat sau elementele sunt în ordine inversă, va rezulta practic o slabă partiționare

➤ un element la întâmplare

➤ un element rezultat ca mediană unui grup de trei elemente

- mediana unui grup de N numere – este cel mai mare în raport cu $N/2$ numere
- alegerea celor trei elemente la extreme și mijloc – elimina cazul defavorabil al unei intrări gata sortate

➤ uzual, elementul situat ca poziție la mijlocul intervalului

3.6 quickSort – implementare recursivă

```
void qsR(char *s, int st, int dr){
    int i=st, j=dr; char x, y;
    x = s[(st+dr)/2];
    do{
        while(s[i]<x && i<dr) i++;
        while(x<s[j] && j>st) j--;
        if(i<j){
            y=s[i];s[i]=s[j];s[j]=y;
            i++;j--;
        }
    }while (i<=j);
    if(st<j) qsR(s, st, j);
    if(i<dr) qsR(s, i, dr);
}
```

Apel:

```
void quickSort(char *s, int n){
    qsR(s, 0, n-1);
}
```

3.6 quickSort – implementare iterativă

- Recursivitatea este substituită printr-o iterație: toate partițiile amânate ca procesare sunt menținute într-o stivă
 - se introduce intervalul inițial de sortat în stivă (limitele acestuia - push)
 - repetă
 - se extrag din stivă limitele intervalului => interval curent (pop)
 - repetă
 - se partiționează intervalul curent până când terminare partiționare
 - dacă există interval stâng atunci
 - se introduc limitele sale în stivă (push)
 - dacă există interval drept atunci
 - se introduc limitele sale în stivă (push)
- până când stiva se golește

3.6 quickSort – implementare iterativă

```
void qsI(char *s, int st, int dr){
    int i, j; char x, y;
    int stiva[dr-st+1]; //creare stiva
    int top=-1; //initializare varf
    stiva[++top]=st; //depune stanga
    stiva[++top]=dr; //depune dreapta
    while(top >= 0){ //cat timp stiva nu e goala
        dr=stiva[top--]; //extrage dreapta
        st=stiva[top--]; //extrage stanga
        x=s[(st+dr)/2]; i=st; j=dr;
        do{
            while(s[i]<x && i<dr) i++;
            while(x<s[j] && j>st) j--;
            if(i<=j){
                y=s[i];s[i]=s[j];s[j]=y;
                i++;j--;
            }
        }while (i<=j);
    }
```

```
        if(st<j){
            /*salvare limite
            partitie stanga*/
            stiva[++top]=st;
            stiva[++top]=j;
        }
        if(i<dr){
            /*salvare limite
            partitie dreapta*/
            stiva[++top]=i;
            stiva[++top]=dr;
        }
    } //de la while
} //de la functie
```

Apel:

```
void quickSort(char *s, int n){
    qsI(s, 0, n-1);
}
```

3.6 quickSort – analiză

➤ Comparații pentru o partiționare C_0

- $C_0 = n$ – după alegerea unui pivot x sunt necesare n comparații

➤ Mișcări pentru o partiționare M_0

- se determină prin metode probabilistice
- presupunând că elementele partiției curente au valorile $1, 2, \dots, n$ și pivotul ales are valoarea x , probabilitatea ca un element să fie $\geq x$ este $(n-x+1)/n$
- numărul de schimbări necesare: $(n-x)(n-x+1)/n$
- deoarece x poate fi ales ca orice valoare $1, 2, \dots, n$, va rezulta un număr mediu de mișcări realizate pentru o partiționare, aproximat la: **$M_0 = n/6$**

➤ Presupunând că de fiecare dată în procesul de partiționare va fi selectat drept pivot mediana (elementul situat ca valoare în mijlocul partiției, când aceasta este ordonată), atunci numărul **maxim de treceri** va fi **$\log_2 n$**

- $C = n \log_2 n$
- $M = (n/6) \log_2 n$ (**cazul optim**)

3.6 quickSort – analiză

- În **situații reale**, performanța medie este cu aproximativ 40% inferioară performanței optime
 - performanțe moderate pentru valori mici ale lui n
- Algoritmul se comportă straniu:
 - situația cea mai favorabilă o reprezintă tabloul inițial ordonat invers
 - situația cea mai defavorabilă, tabloul inițial ordonat
- În situația în care la fiecare partiționare se alege ca pivot elementul cu valoare **extremă** (cel mai mare sau cel mai mic)
 - performanța degenerază la $O(n^2)$
 - n partiționări în loc de $\log_2 n$

3.6 quickSort – unde e folosit?

- E potrivit atunci când poate fi folosită o metodă recursivă
- E recomandat când performanțele contează
- Fiind o tehnică divide-et-impera (divide-and-conquer) permite paralelizarea execuției

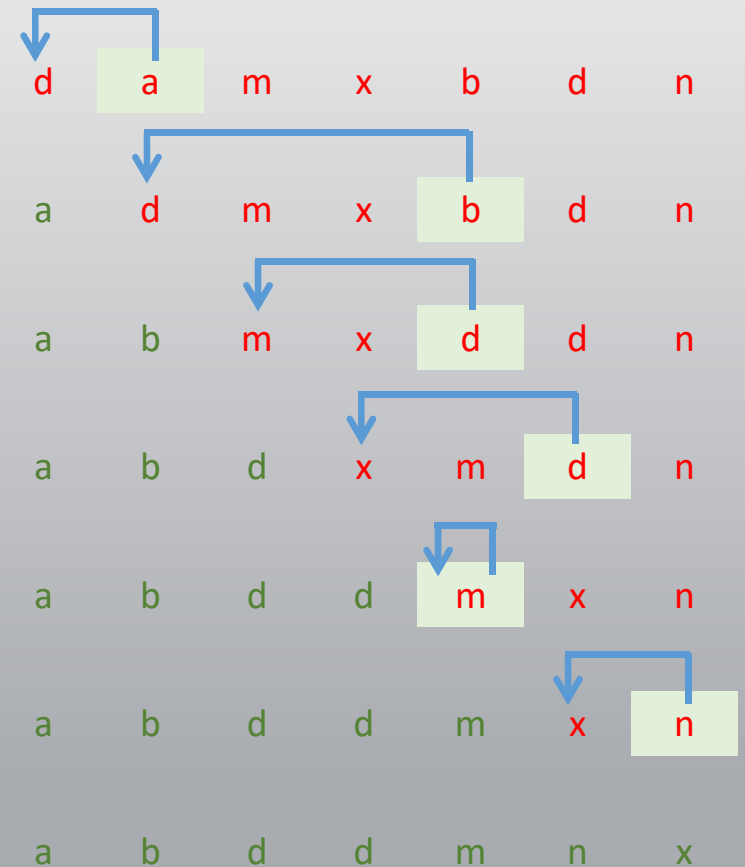
3.7 Tehnica sortării prin metoda ansamblelor – heapSort

➤ Generalizează metoda sortării prin selecție

➤ Recapitulare – metoda sortării prin selecție:

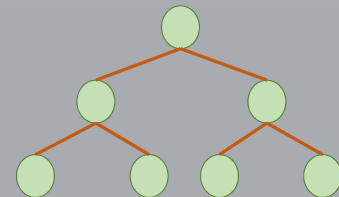
➤ Demonstrație heapSort

- https://www.youtube.com/watch?v=MtQL_I15KhQ



3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- La o primă trecere prin cele n elemente de sortat se va determina prin $n/2$ comparații **cel mai mic element al fiecărei perechi**
- În următoarea trecere, prin $n/4$ comparații se determina cel mai mic element din perechile formate din elementele selectate în pasul precedent
- Cele $n/2 + n/4 + \dots + 1 = n-1$ comparații vor conduce la un **arbore de selecție** având ca rădăcină elementul minim
- Sortarea constă în extragerea minimului din rădăcina arborelui de selecție și refacerea acestuia, ceea ce va conduce la un nou minim în rădăcină, care la rândul său poate fi eliminat
- După n astfel de pași, arborele de selecție devine vid și procesul de sortare este încheiat
- $O((n-1) + n \log_2 n) \rightarrow O(n \log_2 n)$



3.7 Tehnica sortării prin metoda ansamblelor – heapSort

➤ Dezavantaje ale arborelui de selecție:

- Arborele utilizat pentru sortarea a n elemente necesită $2n-1$ locații de memorie
- În procesul de sortare, rămân locații lipsite de informații care sunt sursa unor comparații inutile
- Nu se poate realiza o sortare „in situ”, necesitând o structură de date suplimentară pentru păstrarea elementelor sortate

➤ Aceste dezavantaje sunt eliminate în structura de date numită ansamblu sau heap:

➤ Un ansamblu (heap) este definit drept o secvență de elemente

- $h_s, h_{s+1}, h_{s+2}, \dots, h_d$
- astfel încât $h_i \leq h_{2i}$ și $h_i \leq h_{2i+1}$ pentru orice $i=s\dots d/2$

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- Reprezentarea ansamblului cu ajutorul structurii de date tablou presupune etapele:
 - se numerotează elementele ansamblului, nivel cu nivel, de sus în jos și de la stânga la dreapta
 - se asociază elementelor ansamblului locațiile unui tablou $h[n]$, astfel încât elementului h_i al ansamblului îi va corespunde locația $h[i]$ a tabloului
 - în cazul sortării în ordine crescătoare, în locația $h[1]$ se va afla elementul de valoare minimă
- Construcția ansamblului pornește de la observația:
 - dacă se dispune de ansamblul $h_{s+1}, h_{s+2}, \dots, h_d$ se poate obține ansamblul extins h_s, h_{s+1}, \dots, h_d , dacă elementul adăugat se plasează în rădăcina arborelui, fiind apoi deplasat „în jos” spre locul lui, de-a lungul drumului indicat de componentele cele mai mici
 - această manieră de deplasare „în jos” conservă caracteristicile care definesc un ansamblu

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- Se consideră tabloul h în care se introduc elementele din care se dorește a se construi ansamblul
- Elementele plasate începând cu indicele $n/2$ formează deja un ansamblu deoarece nu există nicio pereche de indici i și j care să satisfacă relația $j=2i$ sau $j=2i+1$
 - aceste elemente formează structura de bază a ansamblului („frunzele” arborelui binar asociat)
- Ansamblul este extins spre stânga cu câte un element la fiecare pas, până când se ajunge la prima poziție a tabloului
- În fiecare pas, elementul nou introdus migrează la locul potrivit conform unei proceduri de deplasare

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

```
deplasare(s, d):  
i=s; /*i precizeaza nivelul curent in ansamblu*/  
j=2*i; /*j precizeaza nivelul urmator*/  
x=h[i];  
cat timp exista niveluri in ansamblu (j<=d) si locul nu a fost  
gasit executa  
    - selecteaza drept elem. curent cel mai mic elem. de pe  
    nivelul urmator j (h[j] sau h[j+1])  
    - daca x > elem. curent atunci  
        - se deplaseaza elem curent de pe nivelul urmator pe  
        cel curent (h[i] ← h[j])  
        - avanseaza parcurgerea la nivelul urmator (i=j;  
j=2*i;)  
    altfel  
        return (locul a fost gasit)  
se plaseaza x la locul lui (h[i] ← x)
```

3.7 Tehnica sortării prin metoda ansamblelor – heapSort

- **Construcția** ansamblului „in situ” necesită repetarea procedurii deplasare pentru toate cele $n/2$ elemente în afara șirului de bază:

```
s = (n/2) + 1;
while (s > 1) {
    s = s - 1; deplasare(s, n);
}
```

- **Sortarea** heapsort „in situ” presupune execuția a n pași de deplasare, după fiecare pas selectându-se elementul de pe poziția 1 (vârful ansamblului) care se interschimbă cu elementul x de pe poziția n
- Se reduce dimensiunea ansamblului cu 1 la dreapta și se lasă componenta x să „migreze” la locul potrivit aplicând procedura de deplasare:

```
d = n;
while (d > 1) {
    h[1] <-> h[d]
    d = d - 1;
    deplasare(1, d);
}
```

3.7 heapSort – analiză

- **Construcția** ansamblului se realizează în $n/2$ pași, fiecare pas deplasând elemente de-a lungul a $\log_2(n/2)$ poziții, respectiv $\log_2((n/2) + 1) \dots \log_2(n-1) \rightarrow O(n \log_2 n)$
- **Sortarea** necesită $n-1$ deplasări cu cel mult $\log_2(n-1), \log_2(n-2), \dots, 1$ mișcări
- Mai sunt necesare $n-1$ mișcări pentru a plasa elementele sortate începând cu indicele 1, elementul minim
- $O((n/2)\log_2(n-1) + (n-1)\log_2(n-1) + (n-1)) \quad C \rightarrow O(n \log_2 n)$
- Numărul mediu de mișcări: $M = (n \log_2 n) / 2$ (determinat statistic)
- Este dificil de stabilit cazul cel mai defavorabil
- Metoda heapsort **este eficientă dacă elementele de sortat sunt inițial apropiate de ordonarea inversă**

3.8 Tehnica sortării binSort

- Procesul de sortare poate fi mult accelerat dacă sunt **informații apriorice** asupra elementelor de sortat
- Se presupune că se lucrează cu un set de chei de tip întreg aparținând intervalului $[0, n-1]$, fără duplicate (sunt distincte)
 - elementele tabloului sunt de fapt permutări ale indicilor
- Dacă se renunță la restricția „in situ” se vor utiliza **două tablouri** de dimensiunea n , a și b , sortarea elementelor din a în b realizându-se într-o singură trecere prin plasarea fiecărui element la locul potrivit în b :
 - $b[i]$ va fi locul cheii având valoarea i
- Metoda poate fi realizată și „in situ” prin utilizarea unor **variabile** pentru păstrarea valorilor până la găsirea locului
- Procedura de sortare constă deci în verificarea fiecărei chei și introducerea ei în locul (*bin-ul*) corespunzător

3.8 Tehnica sortării binsort

- O altă metodă (tot „in situ”) urmărește ca într-o anumită etapă i
 - cât timp pe poziția i nu se găsește valoarea i , ci o valoare $j \neq i$, se interschimbă valorile de pe pozițiile i și j
 - $v[i] \leftrightarrow v[v[i]]$
- Performanța: $O(n)$
- Constrângeri:
 - domeniu limitat
 - chei unice
- Dacă se acceptă și chei identice ($m = nr$ de chei), performanța scade: $O(n+m)$

tablou: 5 3 0 4 6 2 1

Etapă

0	ind	0	1	2	3	4	<u>5</u>	6
	val	<u>5</u>	3	0	4	6	2	1
	ind	0	1	<u>2</u>	3	4	5	6
	val	<u>2</u>	3	0	4	6	5	1
1	ind	0	1	2	<u>3</u>	4	5	6
	val	0	<u>3</u>	2	4	6	5	1
	ind	0	1	2	3	<u>4</u>	5	6
	val	0	<u>4</u>	2	3	6	5	1
	ind	0	1	2	3	4	5	<u>6</u>
	val	0	<u>6</u>	2	3	4	5	1
2, 3, ...	ind	0	1	2	3	4	5	6
	val	0	1	2	3	4	5	6

3.9 Tehnica sortării prin determinarea distribuțiilor

- Se consideră un tablou de n elemente, cu cheile cuprinse în intervalul $[0, m-1]$
- Dacă $m < n$ rezultă că tabloul va conține și elemente identice
- Ideea algoritmului de sortare este:
 - de a contoriza într-o primă trecere numărul de elemente pentru fiecare valoare posibilă de cheie (distribuțiile)
 - iar apoi într-o a doua trecere de a muta elementele direct în poziția lor ordonată
- Sunt necesare **două structuri de date auxiliare**: un tablou pentru contorizarea distribuțiilor și un tablou în care se va păstra structura sortată
- Performanța operației de sortare: $O(n)$

Vă mulțumesc!

Structuri de date și algoritmi

upT Universitatea
Politehnica
Timișoara

P-ța Victoriei nr. 2
RO 300006 - Timișoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 6

Ș.I. dr.ing. Adriana ALBU

adriana.albu@upt.ro

<http://www.aut.upt.ro/~adrianaa>

3. Tehnici de sortare (partea a treia)

3.10 Sortări utilizând baze de numerație (radix-sort)

- Metodele de sortare prezentate consideră elementele de sortat (cheile) drept entități, utilizate ca atare în operațiile de comparație și interschimbare asociate procesului de sortare
- Metodele de sortare de tip „radix” iau în considerare **proprietățile digitale** ale numerelor, respectiv posibilitatea de a reprezenta numerele în diferite baze de numerație și de a utiliza în procesul de sortare componentele asociate reprezentării (biți – baza 2, cifre zecimale – baza 10, cifre hexazecimale etc.)
- Reprezentarea în baza m conduce la metode radix- m
- Procesul de sortare **impune operații de acces la componentele reprezentării**, individuale sau grupate funcție de metoda de sortare
 - pentru reprezentarea în baza 2 sunt necesare funcții de acces la un grup contiguu de biți
- Considerând componenta **cea mai semnificativă la stânga** și cea mai puțin semnificativă la dreapta, examinarea componentelor asociate reprezentării se poate face fie de la stânga la dreapta, fie de la dreapta la stânga

3.10.1 Sortare radix prin interschimbare (sortare după ranguri)

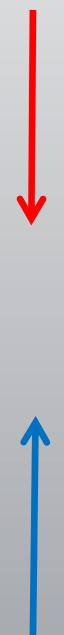
- Se bazează pe observația că **rezultatul comparației** a două elemente este dat în primă fază de valoarea componentei celei mai semnificative, respectiv de valoarea primei componente care diferă
- Elementele sunt procesate de la stânga la dreapta, începând cu componenta cea mai semnificativă
- Principiul metodei radix-2:
 - printr-un procedeu de partiționare asemănător lui quickSort (deci prin interschimbare) se obțin succesiv câte două partiții
 - prima conținând elemente care încep cu 0
 - cea de-a doua elemente care încep cu 1
 - se continuă procesul pentru următorul bit, până la epuizarea tuturor biților asociați reprezentării

3.10.1 Sortare radix prin interschimbare (sortare după ranguri)

```
radix_schimb(int s, int d, int b){ /*b - nr. de biti*/
    int t, i, j;
    if((d>s) && b>=0){
        i=s;
        j=d;
        b=b-1;
        do{
            while((biti(a[i].cheie,b,1)==0) && (i<j)) i=i+1;
            while((biti(a[j].cheie,b,1)==1) && (i<j)) j=j-1;
            t=a[i]; a[i]=a[j];a[j]=t;
        }while(i != j);
        if(biti(a[d].cheie,b,1)==0)
            j=j+1; /*refacere lungime partitie */
        radix_schimb(s, j-1, b-1);
        radix_schimb(j, d, b-1);
    }
}
```


3.10.1 Sortare radix prin interschimbare (sortare după ranguri)

➤ Exemplu: 7, 3, 15, 7, 10, 4, 1



7	0111	7	0111	7	0 1 11	1	0001	1	0001
3	0011	3	0011	3	0011	3	0011	3	0011
15	1 111	1	0001	1	0 0 01	7	01 1 1	4	0100
7	0111	7	0111	7	0111	7	0111	7	0111
10	1010	10	1 010	4	0100	4	01 0 0	7	0111
4	0100	4	0 100	10	1010	10	1010	10	1010
1	0 001	15	1111	15	1111	15	1111	15	1111

3.10.2 Sortarea radix directă (sortare prin distribuție)

- Se realizează sortarea după un bit, procesând biții de la dreapta la stânga, începând cu cel mai puțin semnificativ
- Sortarea după un bit trebuie să fie stabilă, astfel încât la sortarea bitului de indice i , toți biții cuprinși între i și lungimea elementului să fie deja sortați
- Se utilizează funcția `biti(a[i].cheie, k, 1)`; $k=0, 1, 2, \dots, b-1$
- Sunt necesare b treceri; b = numărul de biți asociați elementului
- Pentru sortarea după un bit se aplică **metoda sortării cu determinarea distribuțiilor**, fiind necesar un **tablou suplimentar** de dimensiune 2^1
- „sortare digitală” sau „sortare prin metoda buzunarelor”
- Numărul de treceri corespunde numărului de biți (ranguri), cu avantajul că sortarea în cadrul unei treceri se face independent de trecerea anterioară, ceea ce simplifică algoritmul

3.10.2 Sortarea radix directă (sortare prin distribuție)

➤ Exemplu: 7, 3, 15, 7, 10, 4, 1 m=1

7	0111	10	1010	4	0100	1	0001	1	0001
3	0011	4	0100	1	0001	10	1010	3	0011
15	1111	7	0111	10	1010	3	0011	4	0100
7	0111	3	0011	7	0111	4	0100	7	0111
10	1010	15	1111	3	0011	7	0111	7	0111
4	0100	7	0111	15	1111	15	1111	10	1010
1	0001	1	0001	7	0111	7	0111	15	1111
Distribuții									
0	1	0	1	0	1	0	1		
2	5	2	5	3	4	5	2		

3.10.2 Sortarea radix directă (sortare prin distribuție)

- **Reducerea numărului de treceri** se obține prelucrând un grup de m biți în locul unuia singur $\Rightarrow b/m$ treceri și un tablou pentru contorizarea distribuțiilor de dimensiune 2^m ; b – multiplu de m
 - Ex. $m = 2$
- Sortarea radix directă depinde de evaluarea lui m
 - pentru $m=b$ sortarea degenerază în sortare cu determinarea distribuțiilor
- Dezavantaj: pentru sortarea tabloului a este necesar tabloul destinație b , fiind necesară readucerea lui în a înaintea fiecărei treceri
- Variantă: în fiecare trecere să se realizeze 2 pași de sortare:
 - 1) $a \rightarrow b$
 - 2) $b \rightarrow a$

3.10.2 Sortarea radix directă (sortare prin distribuție)

➤ Exemplu: 7, 3, 15, 7, 10, 4, 1 $m=2$

7	0111
3	0011
15	1111
7	0111
10	1010
4	0100
1	0001

Distribuții				
i	0	1	2	3
D[i]	1	1	1	4
D[i]	1	2	3	7
Indici pentru mutare				
	0	1	2	6
				5
				4
				3

4	0100
1	0001
10	1010
7	0111
3	0011
15	1111
7	0111

Distribuții				
i	0	1	2	3
D[i]	2	3	1	1
D[i]	2	5	6	7
Indici pentru mutare				
	1	4	5	6
	0	3		
		2		

1	0001
3	0011
4	0100
7	0111
7	0111
10	1010
15	1111

Analiza sortării radix

- Sortarea radix prin interschimbare are **aceleași performante că și quicksort**
 - $C = n \log_2 n$
- Ambele sortări radix (prin schimburi și directă) utilizează $n * b$ comparații de biți pentru a sorta n elemente după b biți
- Sortarea radix directă poate sorta n elemente de lungime b biți în b/m treceri, utilizând un **spațiu suplimentar** de memorie
 - necesar tabloului de distribuții, cu dimensiunea 2^m
 - un tablou suplimentar, de dimensiunea celui de sortat
- Pornind de la observația „creșterea lui m reduce numărul de treceri”, sortarea radix directă poate conduce la performanțe apropiate de cele oferite de o sortare liniară, pentru un m suficient de mare

3.11 Sortarea indirectă (Sortarea tablourilor cu elemente de mari dimensiuni)

- În cazul în care dimensiunea elementelor de sortat este mare, „**costul**” **mișcării** acestora în cadrul tabloului în cazul metodelor de sortare prezentate poate deveni semnificativ, conducând astfel la diminuarea performanțelor
- Sortarea indirectă presupune accesarea elementelor tabloului de sortat prin intermediul unui **tablou suplimentar** numit tablou **de indici**, care va conține indicatori (pointeri) spre elementele tabloului
- Sortarea se va realiza în tabloul de indici, urmând ca tabloul original să fie sortat ulterior într-o singură trecere
- Considerând tabloul de sortat a și tabloul de indici p , în procesul de sortare se vor compara elementele din a (accesul în a realizându-se prin intermediul indicilor din p) și se vor mișca (muta) elementele din p
- În multe situații este suficientă doar sortarea tabloului de indici, fără a trebui reordonate elementele tabloului

3.12 Sortare externă. Sortarea fișierelor secvențiale

3.12.1 Tehnica sortării prin interclasare (mergeSort)

- **Accesul strict secvențial** la componentele structurii secvență conduce la modificări ale tehnicilor de sortare
 - restricție severă comparativ cu accesul direct specific structurii de tablou
- Principiul de bază al acestor metode de sortare este **interclasarea**
 - presupune combinarea a două sau mai multe secvențe ordonate într-o singură secvență ordonată, prin selecții repetate ale componentelor curent accesibile, în manieră secvențială
 - Ex. : interclasarea secvenței 1,3,4,5,8,9 cu secvența 2,6,7,10,14,17
- Aplicarea metodei de sortare prin interclasare constă în construirea unor secvențe sortate, începând cu secvențe de lungime 1, și interclasarea lor
 - 1,2,3,4,5,6,7,8,9,10,14,17
- Fișierul secvențial supus sortării este simulat printr-un tablou

3.12.1 Tehnica sortării prin interclasare (mergeSort)

➤ E o tehnică divide-et-impera (divide-and-conquer)

➤ Rezultă astfel etapele:

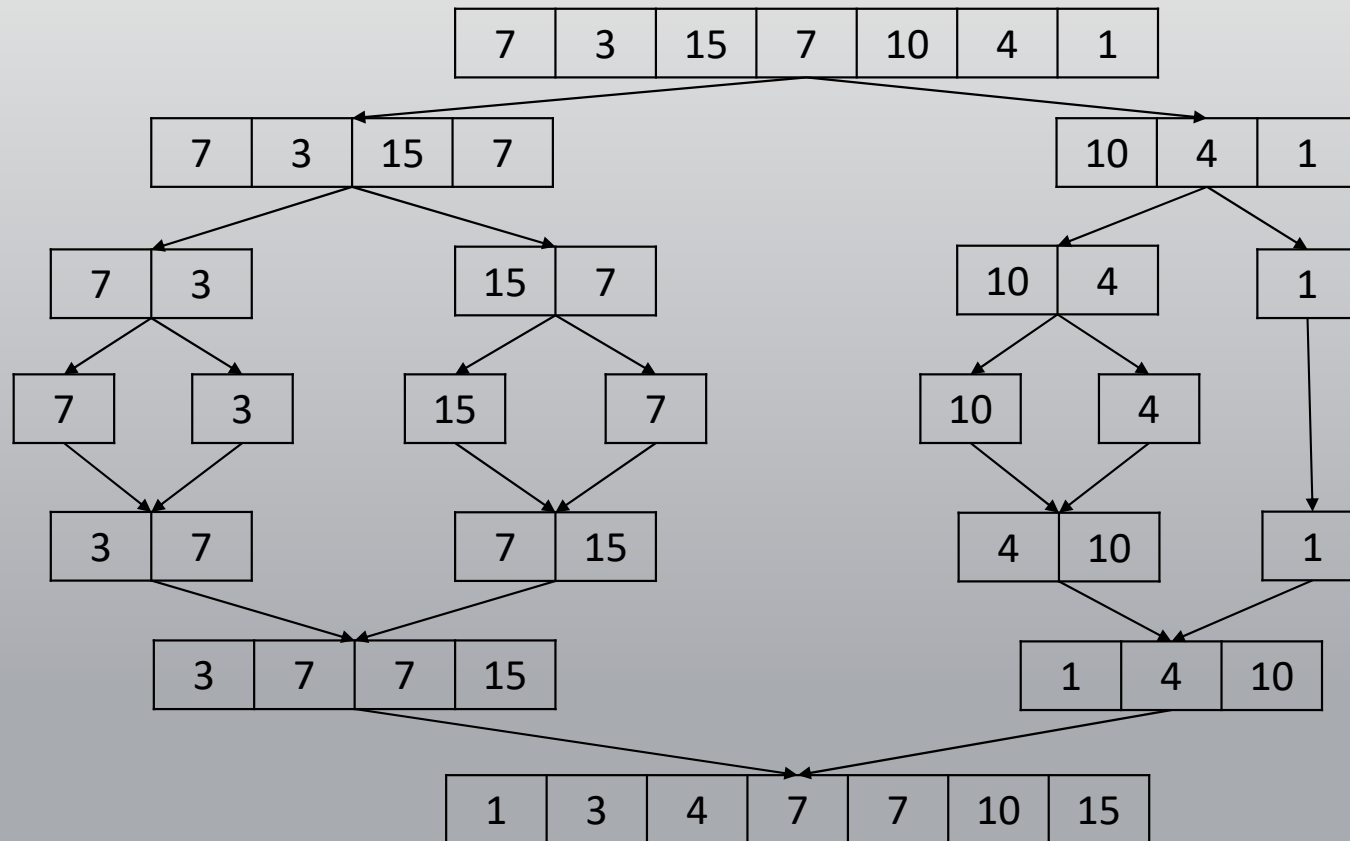
1. *Divide*: Se împarte secvența de interclasat a în două sub-secvențe b și c , egale sau diferind cu 1 ca lungime (defalcare, înjumătățire) \Rightarrow două subprobleme, instanțe ale aceleiași probleme
2. *Conquer*: Se rezolvă recursiv subproblemele
3. *Combine*: Se interclasează soluțiile b și c ale subproblemelor în a , combinând câte un element din fiecare în perechi ordonate

```
mergeSort(A, 0, length(A)-1)
```

```
mergeSort(A, p, r) :  
    if (p > r)  
        return  
    q = (p+r)/2  
    mergeSort(A, p, q)  
    mergeSort(A, q+1, r)  
    merge(A, p, q, r)
```

3.12.1 mergeSort

➤ Exemplan: 7, 3, 15, 7, 10, 4, 1



3.12.1 Tehnica sortării prin interclasare (mergeSort)

- Defalcarea se oprește când sub-secvențele de sortat au dimensiune 1
- Apoi se realizează interclasarea, prin următorii pași:
 - S-a ajuns la sfârșitul uneia dintre sub-secvențe?
 - Nu:
 - Se compară elementele curente din cele două sub-secvențe
 - Se copiază elementul mai mic în secvența de sortat
 - Se mută cursorul după elementul copiat
 - Da:
 - Se copiază restul sub-secvenței nevide

3.12.1 Tehnica sortării prin interclasare (mergeSort)

```
/*Interclasare două sub-secvențe L și M în secvența arr*/  
void merge(int arr[], int p, int q, int r) {  
    /*Creare L ← A[p..q] și M ← A[q+1..r]*/  
    int n1 = q - p + 1;  
    int n2 = r - q;  
    int L[n1], M[n2];  
  
    for (int i = 0; i < n1; i++)  
        L[i] = arr[p + i];  
    for (int j = 0; j < n2; j++)  
        M[j] = arr[q + 1 + j];  
  
    /*Refacere indecsi*/  
    int i, j, k;  
    i = 0;  
    j = 0;  
    k = p;
```

3.12.1 Tehnica sortării prin interclasare (mergeSort)

```
/*Atata timp cat exista elemente  
in ambele sub-secvnte, elementul mai  
mic se plaseaza in arr*/
```

```
while (i < n1 && j < n2) {  
    if (L[i] <= M[j]) {  
        arr[k] = L[i];  
        i++;  
    } else {  
        arr[k] = M[j];  
        j++;  
    }  
    k++;  
}
```

```
/*La epuizarea uneia dintre  
sub-secvnte, se copiaza in arr  
elementele ramase in cealalta sub-  
secvnta*/
```

```
while (i < n1) {  
    arr[k] = L[i];  
    i++;  
    k++;  
}  
  
while (j < n2) {  
    arr[k] = M[j];  
    j++;  
    k++;  
}  
}
```

3.12.1 mergeSort – analiză

- Trecheri: $\log_2 n$
- În fiecare trecere sunt copiate toate cele n elemente, rezultând astfel numărul total de **mișcări**
 - $M = n \log_2 n$
- Numărul total de **comparații** este mai mic decât numărul de mișcări, deoarece operația de copiere a resturilor nu presupune comparații
 - $C \leq M$

3.13 Concluzii privind sortarea tablourilor

- Metodele de sortare prezentate conduc la următoarele clase de performanțe:
 - $O(n^2)$ – metodele directe, simple
 - $O(n \log_2 n)$ – metodele avansate, complexe
 - $O(n)$ – dacă se dispune de informații suplimentare, zone suplimentare de memorie
- Beneficiile inserției binare față de inserția simplă sunt nesemnificative, chiar negative în cazul tablourilor gata sortate
- Inserția prin schimburi (*bubbleSort*)
 - cea mai puțin eficientă în raport cu selecția și inserția, chiar și în varianta sa îmbunătățită ca sortare amestecată
 - performantă în cazul tablourilor gata sortate
- Dacă se ia în considerare dimensiunea elementelor, selecția directă se situează pe primul loc în cadrul metodelor directe, *bubbleSort* pierde din performanță, iar *quickSort* este considerată cea mai rapidă

3.13 Concluzii privind sortarea tablourilor

➤ Tehnica *quickSort*

- superioară tehnicii *heapSort* (factor 2 la 3).
- sortează un tablou sortat invers practic cu aceeași viteză cu care sortează un tablou gata ordonat

➤ În situațiile în care dispunem de informații suplimentare asupra cheilor de sortat, respectiv dacă utilizăm tablouri suplimentare în sortare, performanța crește semnificativ, ajungându-se chiar la $O(n)$

- *binSort* (limitează domeniul cheilor)
- determinarea distribuțiilor se apropie de performanța $O(n)$

➤ *Radix* (interschimburi și directă) concurează *quickSort*

➤ *Radix* cu tablouri suplimentare pentru distribuții poate ajunge la $O(n)$

➤ Sortarea indirectă câștigă performanță prin renunțarea la mișcarea elementelor

Vă mulțumesc!

Structuri de date și algoritmi

upT Universitatea
Politehnica
Timișoara

P-ța Victoriei nr. 2
RO 300006 - Timișoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 7

Ș.I. dr.ing. Adriana ALBU

adriana.albu@upt.ro

<http://www.aut.upt.ro/~adrianaa>

5. Structura de date listă (partea întâia)

5.1 TDA listă

➤ Structură de date dinamică, flexibilă

➤ MM

- o secvență de 0 sau mai multe elemente aparținând unui tip numit tip de bază
- a_1, a_2, \dots, a_n ; a_i nodurile listei
- n – lungimea listei; $n \geq 0$
- dacă $n \geq 1$
 - a_1 – primul nod
 - a_n – ultimul nod
- ordonată liniar funcție de poziția nodurilor
 - a_i precede pe a_{i+1}
 - a_i succede pe a_{i-1}

➤ Notatii:

- TipLista l ;
- TipPozitie p ;
- TipNod x ;

➤ Operatori:

- inserează nod în listă
- șterge nod din listă
- caută nod în listă
- următorul nod în listă
- primul nod în listă
- ListaVida(l);
- InsertieInceput(x, l);
- InsertieDupa(x, l, p);

5.2 Tehnici de implementare / a. Tablouri

```
#define LungMax ...  
typedef ... TipNod; //în funcție de ce se stochează  
typedef int TipIndice;  
typedef struct{  
    TipNod noduri[Lungmax];  
    TipIndice ultim;  
} TipLista;  
TipLista Lista;
```

5.2 Tehnici de implementare / b. Cursori

```
#define LungMax ...
typedef ... TipNod;
typedef int TipCursor;
typedef TipCursor TipLista;
typedef struct{
    TipNod nod_lista;
    TipCursor urm;
}TipCelula;
TipCelula zona[Lungmax];
TipLista L, M, Disponibil;
/*L, M, Disponibil sunt
intrarile in diferite liste*/
```

- E specifică limbajelor care nu dispun de pointeri
- Într-un tablou se pot grupa mai multe liste care conțin același tip de elemente
- Operații :
 - Inserare:
 - se suprimă prima locație din Disponibil
 - se înlanțuie în listă pe poziția dorită
 - Ștergere:
 - se suprimă din listă și se inserează în Disponibil

5.2 Tehnici de implementare / c. Pointeri

```
typedef struct nod{  
    int cheie;  
    struct nod *urm;  
    ... info;  
}TipNod;
```

```
typedef struct nod *TipPointerNod;  
typedef TipPointerNod TipLista;  
TipLista prim;
```

➤ Operații cu liste înlănțuite:

- Inserare
 - la început, la sfârșit, după nodul curent, la poziția nodului curent etc.
- Ștergere
 - nodul următor nodului curent, nodul curent etc.
- Traversare liste

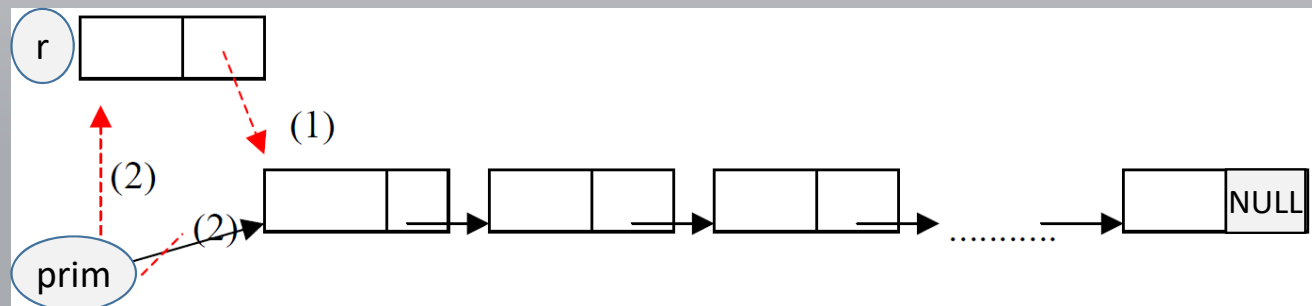
5.3 Tehnici utilizând structura de date listă simplu înlănțuită

- a) Inserarea unui nod la început (capul listei)
- b) Inserarea unui nod la sfârșit (coada listei)
- c) Inserarea unui nod în interiorul listei:
 - după un anumit nod (se cunoaște predecesorul)
 - înaintea unui anumit nod (se cunoaște succesorul)
- d) Crearea unei liste:
 - prin inserări repetate la început
 - prin inserări repetate la sfârșit
 - prin inserări într-un anumit punct, astfel încât să se păstreze lista ordonată
- e) Ștergerea unui nod:
 - când se cunoaște predecesorul
 - ștergerea nodului curent
- f) Traversarea listei
- g) Tehnica celor doi pointeri

5.3 a) Inserarea unui nod la început (capul listei)

```
typedef ... t_date;  
struct nod{  
    t_date date;  
    struct nod *urm;  
}*prim, *q, *r, *ultim;
```

```
r=(struct nod *)malloc(sizeof(struct nod));  
r->urm=prim; // (1)  
prim=r; // (2)
```



5.3 b) Inserarea unui nod la sfârșit (coada listei)

➤ Varianta listei accesată prin `prim`

- necesită parcurgerea listei până la sfârșit

```
for (q=prim; q->urm!=NULL; q=q->urm) ;  
r=(struct nod *)malloc(sizeof(struct nod)) ;  
r->urm=NULL ;  
q->urm=r; /*nu functioneaza in cazul listei vide*/
```

➤ Varianta listei accesată prin `prim` și `ultim`:

```
r=(struct nod *)malloc(sizeof(struct nod)) ;  
r->urm=NULL ;  
if (ultim==NULL)  
    prim=ultim=r ;  
else {  
    ultim->urm=r ;  
    ultim=r ;  
}
```

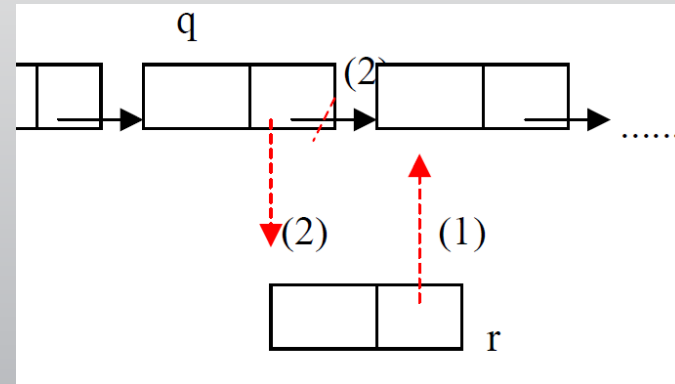
5.3 c) Inserarea unui nod în interiorul listei

➤ după un anumit nod (*q) (se cunoaște predecesorul)

```
r=(struct nod *)malloc(sizeof(struct nod));
```

```
r->urm=q->urm; // (1)
```

```
q->urm=r; // (2)
```



➤ înaintea unui anumit nod (*q) (se cunoaște succesul)

```
r=(struct nod *)malloc(sizeof(struct nod));
```

```
*r=*q; /*inclusiv campul de inlantuire*/
```

```
q->urm=r; /*se va completa nodul q cu informatia dorita*/
```

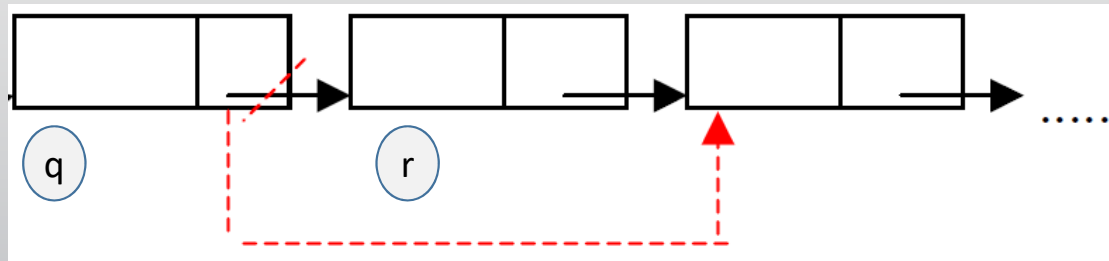
5.3 d) Crearea unei liste simplu înlănțuite

- prin inserări repetate la sfârșit
 - rezultă o listă în ordina inserării nodurilor
- prin inserări repetate la început
 - rezultă o listă în ordine inversă inserării nodurilor
- prin inserări într-un anumit punct
 - se păstrează lista ordonată

5.3 e) Ștergerea unui nod

➤ când se cunoaște predecesorul nodului (*q)

```
r=q->urm;  
q->urm=r->urm;  
free(r);
```



➤ ștergerea nodului curent (*q)

```
r=q->urm;  
*q=*r; /*inclusiv campul de inlantuire*/  
free(r);
```

➤ Obs. q->urm !=NULL

5.3 f) Traversarea listei

```
void procesare(struct nod *...)  
{  
...  
}  
...  
for (q=prim; q!=NULL; q=q->urm)  
    procesare(q) ;
```

5.3 g) Tehnica celor doi pointeri

- Crearea unei liste ordonate prin tehnica celor doi pointeri
 - q_2 îl precede pe q_1
- Cei doi pointeri avansează simultan până când cheia lui q_1 devine mai mare sau egală cu x – nodul care se inserează
 - nodul x se inserează după q_2

Vă mulțumesc!

Structuri de date și algoritmi

upT Universitatea
Politehnica
Timișoara

P-ța Victoriei nr. 2
RO 300006 - Timișoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 8

Ș.I. dr.ing. Adriana ALBU

adriana.albu@upt.ro

<http://www.aut.upt.ro/~adrianaa>

5. Structura de date listă (partea a doua)

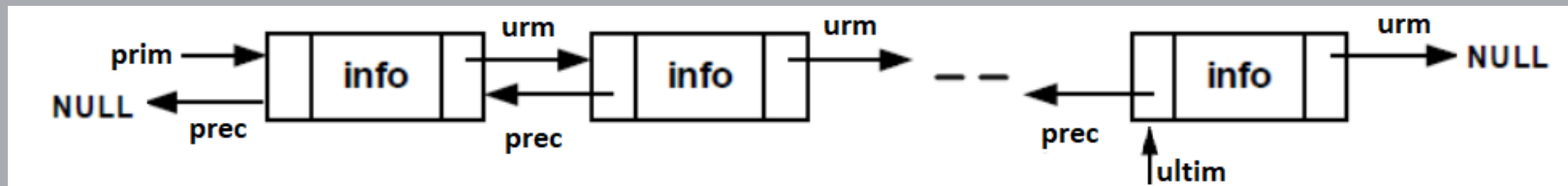
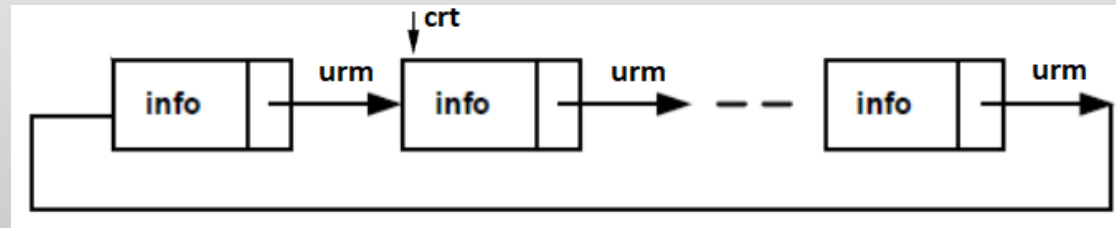
5.4 Alte tipuri de liste

1. Liste simplu înlănțuite ordonate

2. Liste circulare

3. Liste dublu înlănțuite

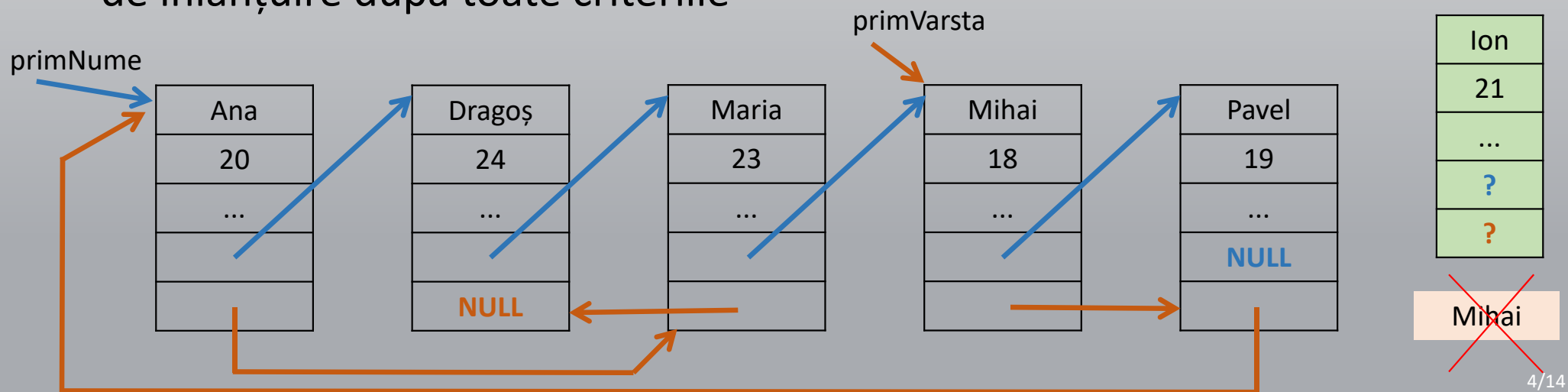
```
typedef ... TipElement;  
typedef struct nod{  
    TipElement elem;  
    struct nod *prec;  
    struct nod *urm;  
}TipNod;
```



5.4 Alte tipuri de liste

4. Liste multiplu înlănțuite

- structura nodului conține **mai multe câmpuri de înlănțuire**, asociate unor criterii diferite de ordonare
- acestui tip de lista i se asociază un număr de **pointeri de început** egal cu numărul câmpurilor de înlănțuire
- toate operațiunile de **inserare și ștergere** trebuie să refacă valoarea câmpurilor de înlănțuire după toate criteriile

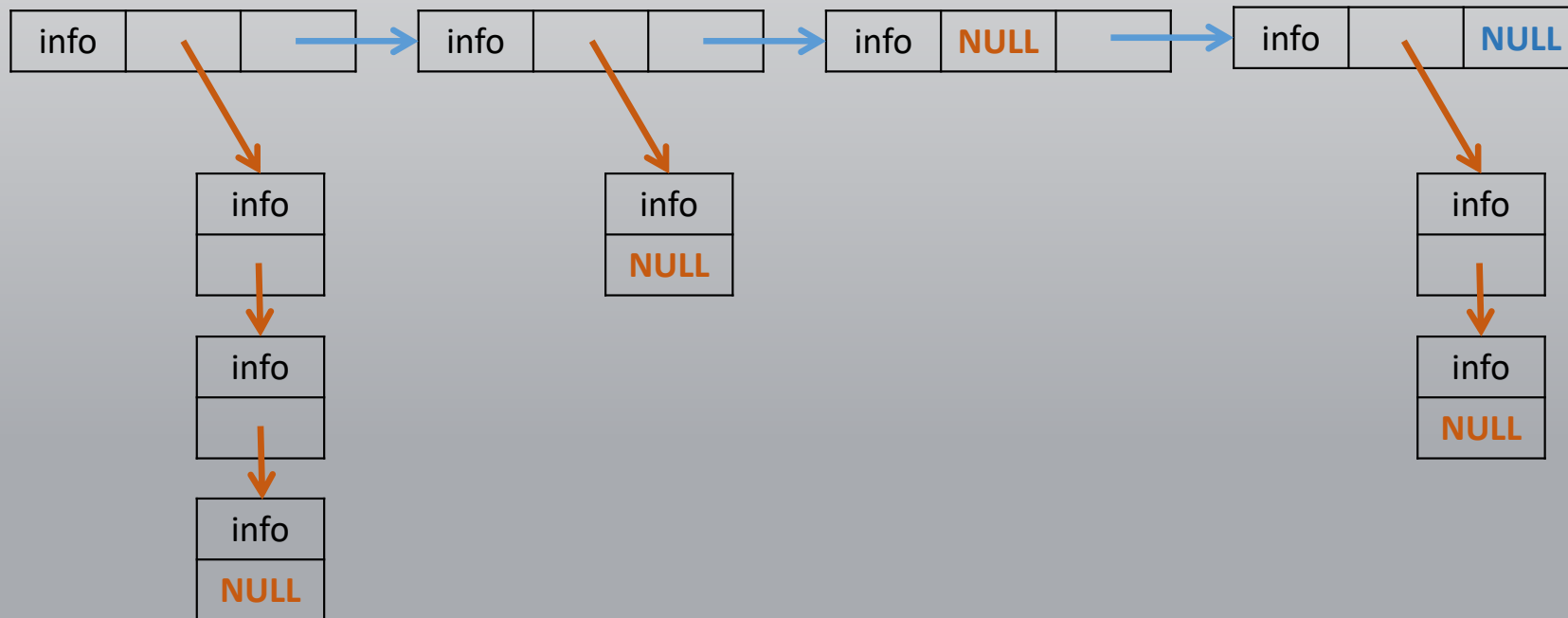


5.4 Alte tipuri de liste

5. Liste generalizate. Liste cu sub-liste

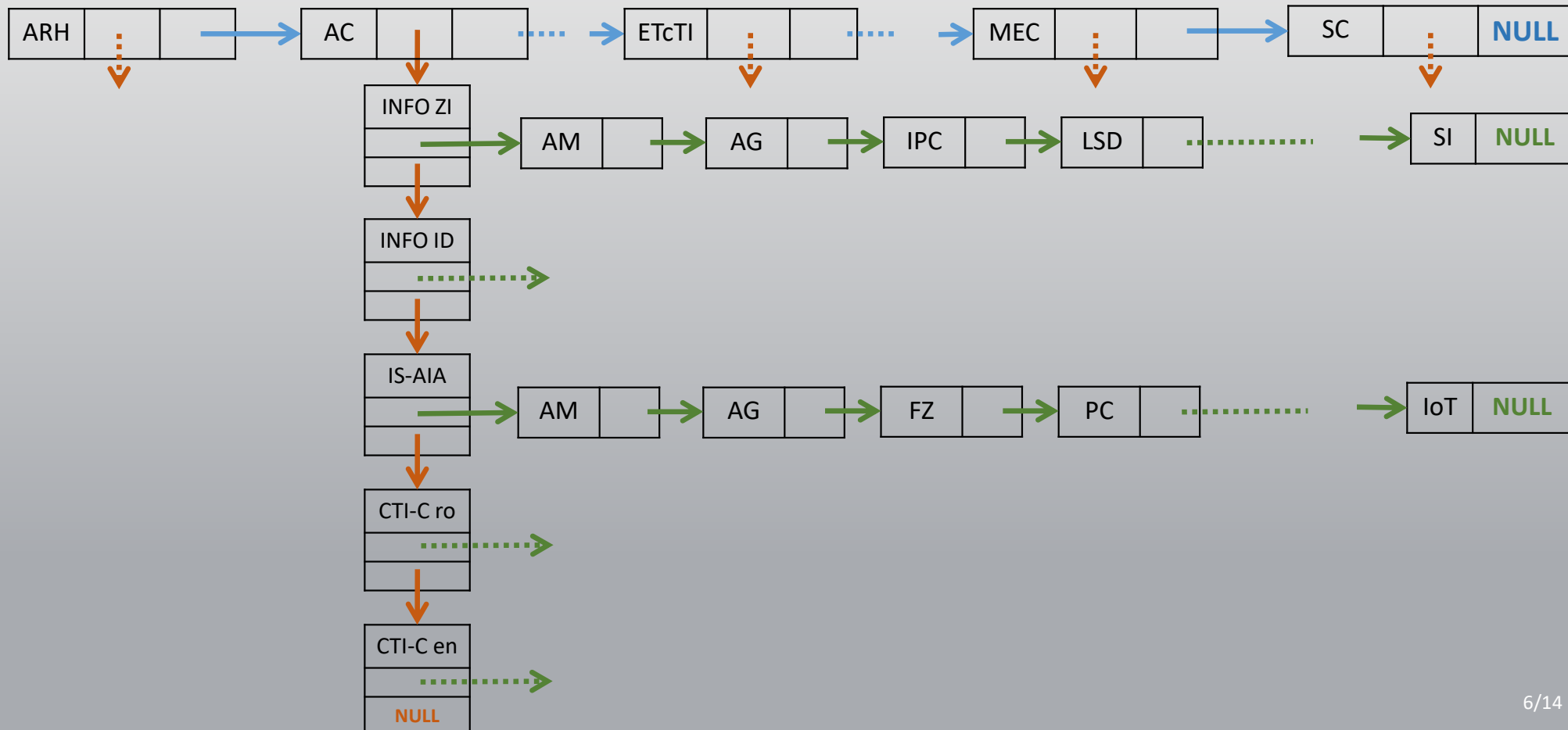
- uzual se identifică o listă principală și sub-liste asociate nodurilor acesteia
- uzual tipul nodurilor din lista principală diferă de tipul nodurilor din sub-liste

prim



Liste cu sub-liste – Exemplu

prim



5.5 Structuri derivate din tipul listă / 5.5.1 Structura de date stivă

➤ TDA stivă

➤ MM

- tip special de listă
- inserările și ștergerile se fac într-un singur capăt, numit uzual vârful stivei
- se aplica principiul „ultimul intrat – primul ieșit” (LIFO – *last in first out*)

➤ Notății

- TipStiva s;
- TipNod x;
- boolean b;

➤ Operatori

- Initializare(s);
- TipNod VarfStiva(s);
- TipNod pop(s);
- push(s,x);
- boolean StivaVida(s);

➤ Implementare: tablouri; pointeri



5.5.2 Structura de date coadă

➤ TDA coadă

➤ MM

- tip special de listă
- elementele sunt inserate la un capăt al cozii (după nodul „ultim”) și sunt șterse la celălalt capăt al cozii (nodul „prim”) – FIFO – *first in first out*

➤ Notății

- TipCoada c;
- TipElement x;
- boolean b;

➤ Operatori

- Initializare (TipCoada c);
- TipElement Prim(TipCoada c);
- adauga(TipCoada c, TipElement x);
- extrage(TipCoada c);
- boolean CoadaVida(TipCoada c);
- boolean CoadaPlina(TipCoada c);
 - pentru implementarea cu tablouri



Implementare coada

a) Pointeri – doi pointeri (prim, ultim)

```
typedef ... TipElement;  
typedef struct nod{  
    TipElement element;  
    struct nod *urm;  
}TipNod;  
typedef tipNod* TipPointerNod;  
typedef TipPointerNod{  
    TipPointerNod prim, ultim;  
}TipCoadă;
```

Implementare coada

b) Tablouri

➤ Liniare

- adăugarea se face în $O(1)$, iar ștergerea în $O(n)$

➤ Circulare

- adăugarea și ștergerea se realizează în $O(1)$
- inserarea presupune avansul indicatorului „ultim”, iar ștergerea avansul indicatorului „prim”
- coada se rotește în sensul acelor de ceasornic după cum se adaugă sau se scot elemente din ea

Cozi bazate pe prioritate

➤ Tip special de coadă:

- elementelor din structură li se asociază o prioritate, iar la extragere va fi extras (șters) de fiecare dată elementul cu prioritatea cea mai mare

➤ Operatori:

- inserare;
- extragere (elementul cu prioritatea cea mai mare);
- schimba_prioritate;
- suprima (șterge) un element oarecare;
- reuneste doua cozi;
- inlocuire
 - se înlocuiește „cel mai mare” element cu un nou element
 - constă de fapt dintr-o inserare urmată de extragerea „celui mai mare”
- schimba
 - o suprimare urmată de inserare

Implementare cozi cu prioritate

a) Liste neordonate (cu pointeri, cu tablouri)

```
void insereaza(Tip Elem x){
    N++; a[N-1]=x;          // O(1)
}

TipElem extrage(){
    TipElem x;
    int j, max=0;
    for(j=1; j<N; j++)
        if(a[j].cheie>a[max].cheie) max=j; //O(N)
    x=a[max];
    a[max]=a[N-1]; N--;
    return x;
}
```

Implementare cozi cu prioritate

b) Liste ordonate (cu pointeri, cu tablouri)

- inserare $\rightarrow O(N)$
- extragere $\rightarrow O(1)$

c) Ansamblu – arbore binar, parțial ordonat, implementat cu tablou

- $a_i \geq a_{2i}$
- $a_i \geq a_{2i+1}$
- $i = 1 \dots N/2$

➤ Obs. Operațiile implementate pe structura de tip coadă cu prioritate pot fi utilizate pentru implementarea unor algoritmi de sortare

- spre exemplu utilizarea repetată a operației de inserare urmată de extragere conduce la obținerea unei secvențe sortate

Vă mulțumesc!

Structuri de date și algoritmi

upT Universitatea
Politehnica
Timișoara

P-ța Victoriei nr. 2
RO 300006 - Timișoara
Tel: +4 0256 403000
Fax: +4 0256 403021
rector@rectorat.upt.ro
www.upt.ro

Domeniul de studii: Informatică/ Specializarea: Informatică

SDA – Cursul 9

Ș.I. dr.ing. Adriana ALBU

adriana.albu@upt.ro

<http://www.aut.upt.ro/~adrianaa>

4. Structura de date șir

4.1 TDA șir

➤ MM: secvență finită de caractere c_1, c_2, \dots, c_n

- $n \rightarrow$ lungimea secvenței; $n=0 \Rightarrow$ șir vid

➤ Notatii:

- s, sub, u – șiruri
- c – valoare de tip caracter
- b – boolean
- $poz, lung$ – întregi pozitivi

➤ Operatorii: implementați în termenii limbajului de programare (primitivi) sau dezvoltați de programator

➤ Operatori:

- CreeazaSirVid(s)
- InsereazaSir(sub, s, poz)
- SirVid(s)-> b
- FurnizeazaCar(s, poz)-> c
- AdaugaCar(s, c)
- LungSir (s)-> $lung$
- Pozitie(sub, s)-> poz
- StergeToateSubsir(sub, s)
- Concat(u, s)
- StergeSir($s, poz, lung$)
- CopiazaSubsir($u, s, poz, lung$)

4.2 Implementarea TDA sir

4.2.1 Implementarea cu tablouri

➤ articol format dintr-un întreg (lungimea șirului) și un tablou de caractere

```
#define LungMax ...
typedef int TipLungime; /*0... LungMax*/
typedef int TipIndice; /*0... LungMax-1*/
typedef char tab[LungMax];
typedef struct{
    TipLungime lungime; /*pentru eficiență*/
    tab sir;
}TipSir;
TipSir s;
```

➤ Ex.:

- Creeaza_Sir_Vid(TipSir s); Lung_Sir (TipSir s); Adauga_Car(TipSir s, char c); → $O(1)$
- Copiaza_Subsir(TipSir u, TipSir s, TipIndice poz, TipLungime lung); → $O(n)$

➤ Implementare simplă, operatori performanți, dar cu o utilizare ineficientă a spațiului de memorie

4.2.1 Implementarea cu tablouri

➤ Exemple din C, biblioteca string.h

```
char str[50];
```

```
strcpy(str, "Ana are mere");  
printf("%s", str); // Ana are mere
```

```
printf("%c", str[4]); // a  
str[8]=toupper(str[8]); // => Ana are Mere
```

```
strcat(str, "!"); // => Ana are Mere!
```

```
printf("%s", strchr(str, 'n')); // na are Mere!  
printf("%s", strstr(str, "are")); // are Mere!
```

➤ ineficient

```
for(i=0; i<strlen(str); i++)  
    //...
```

➤ mai bine

```
int n=strlen(str);  
for(i=0; i<n; i++)  
    //...
```

4.2.2 Implementarea tipului șir prin tabele

- Toate șirurile utilizate la un moment dat se păstrează într-un singur tablou numit *Heap*
- Se asociază o tabelă auxiliară (tablou auxiliar) care păstrează evidența șirurilor, numita tabelă de șiruri
 - o locație în tabela de șiruri referă un șir printr-un articol compus din 2 elemente:
 - lungimea șirului curent
 - un indicator în *Heap* care precizează poziția (începutul) șirului

4.2.2 Implementarea tipului șir prin tabele

```
#define LungimeHeap ...
#define LungimeTabela ...
typedef struct{
    int lungime; /*0...LungimeHeap*/
    int indicator; /*0...LungimeHeap-1*/
}ElemTablou;
typedef int TipSir;
typedef ElemTablou tab1[LungimeTabela];
typedef char tab2[LungimeHeap];
tab1 TabelaDeSiruri;
tab2 Heap;
int Disponibil;
int NumarDeSiruri;
```

➤ Lungimea limitată a șirurilor complică operatorii de tip adăugare caracter, ștergere subșir etc.

- → noi instanțe ale șirurilor implicate, la sfârșitul tabloului *Heap*

➤ Copierea unui șir sursă într-un șir destinație se obține poziționând câmpul indicator al destinației spre indicatorul șirului sursa în *Heap*

- → economie de memorie

4.2.3 Implementarea șirurilor cu pointeri

- Se bazează pe reprezentarea șirurilor drept o colecție de celule înlănțuite, fiecare celulă conținând unul sau mai multe caractere
- Accesul la caracterele șirului se face doar secvențial

```
typedef struct cel{  
    char ch;  
    struct cel *urm;  
}Celula;  
typedef Celula *PointerCelula;  
typedef struct{  
    int lungime;  
    PointerCelula cap, coada;  
}TipSir;  
TipSir s;
```

- Se utilizează în special când lungimea șirurilor este imprevizibilă

4.3 Tehnici de căutare / 4.3.1 Căutare tabelară

- Pentru a stabili coincidența a două șiruri e necesar să se stabilească coincidența tuturor caracterelor corespunzătoare din șirurile comparate

```
#define M ...  
typedef char sir[M];  
sir x,y;  
 $x==y \longleftrightarrow A_j: 0 \leq j < M : x_j == y_j$ 
```

- Lungimea unui șir poate fi precizată:

- explicit (implementările anterioare)
- implicit, prin precizarea unui caracter fanion care să marcheze terminarea șirului

- **Căutarea tabelară** parcurge un tablou de șiruri, pentru fiecare intrare în tablou verificând prezența șirului căutat x prin aplicarea tehnicii de căutare liniară

- în cazul unui terminator de șir coincidența apare dacă ultimul caracter identic este terminatorul de șir

- Structuri imbricate

4.3 Tehnici de căutare în șiruri / 4.3.2 Căutarea directă

- Are drept scop stabilirea poziției primei apariții a șirului p în șirul s
 - rezultatul căutării este indicele i al primei apariții

```
char s[N];
```

```
char p[M]; /* M << N */
```

- Precizarea coincidenței se face cu $P(i,j)$:

$$P(i,j) \quad A_k: 0 \leq k < j : s_{i+k} == p_k$$

- În cazul în care $P(i,j)$ este adevărat, avem o coincidență de j caractere începând cu poziția i în șirul s

- pentru a avea o coincidență pe lungimea M este necesar să fie adevărat $P(i,M)$

- Șirul p este deplasat paralel cu șirul s până la găsirea lui p sau până când numărul caracterelor netestate din s este mai mic decât dimensiunea lui p

4.3 Tehnici de căutare în șiruri / 4.3.2 Căutarea directă

- Căutarea directă lucrează destul de repede dacă se presupune că nepotrivirea dintre s și p apare după cel mult câteva comparații

- $\rightarrow O(N)$

s:	ABABABCDE	ABABABCDE	ABABABCDE
p:	ABCD	ABCD	ABCD

- Cel mai defavorabil caz este când nepotrivirea dintre s și p apare totdeauna pe ultimul caracter din $p \rightarrow O(N*M)$

- s: AAAAAAAAAAABAAAAA.....AAAAA

- p: AAAAAAAAAAB

- Eficiență scăzută: modelul p avansează cu o singură poziție după o nepotrivire

4.3 Tehnici de căutare în șiruri / 4.3.3 Optimizări

➤ Optimizările presupun preprocesare

- Căutarea Boyer-Moore

- se bazează pe ideea de a realiza comparația dintre modelul p și șirul s , începând cu ultimul caracter al modelului

- Căutarea Knuth-Morris-Pratt

- ia în considerare sub-modele ale modelului căutat
- preprocesarea – identificarea celor mai lungi secvențe de caractere din model care sunt prefix al modelului
- la medie, performanțe $O(N+M)$
- <https://www.youtube.com/watch?v=cH-5KcgUcOE>

Vă mulțumesc!