



STRUCTURI DE DATE ȘI TIPURI DE DATE ABSTRACTE

Șl. Dr. Ing. Șerban Radu

Departamentul de Calculatoare

Facultatea de Automatică și Calculatoare



Structuri de date fundamentale

- Prima structură de date folosită a fost structura de **vector**, utilizată în operațiile de sortare a colecțiilor și a fost prezentă în primele limbaje de programare pentru aplicații numerice
- Un **vector** este o **colecție de date de același tip**, în care elementele colecției sunt identificate prin indici ce reprezintă poziția relativă a fiecărui element în vector



Structuri de date fundamentale

- La început se puteau declara și utiliza numai vectori cu dimensiuni fixe, stabilite la scrierea programului și care nu mai puteau fi modificate la execuție
- Introducerea tipurilor pointer și a alocării dinamice de memorie a permis utilizarea de vectori cu dimensiuni stabilite și/sau modificate în cursul execuției programelor

Structuri de date fundamentale

- Gruparea mai multor date, de tipuri diferite, într-o singură entitate, numită **structură**, a permis definirea unor noi tipuri de date de către programatori și utilizarea unor date dispersate în memorie, dar legate prin pointeri: liste înlănțuite, arbori și altele
- Astfel de colecții se pot extinde dinamic pe măsura necesităților și permit un timp mai scurt pentru anumite operații, cum ar fi operația de eliminare a unei valori dintr-o colecție



Clasificări ale structurilor de date

- O **structură de date** este caracterizată prin relațiile dintre elementele colecției și prin operațiile posibile cu această colecție
- Există mai multe feluri de colecții (structuri de date), care pot fi clasificate după câteva criterii



Clasificări ale structurilor de date

- În funcție de relațiile dintre elementele colecției
 - **Colecții liniare** (secvențe, liste), în care fiecare element are un singur succesor și un singur predecesor
 - **Colecții arborescente** (ierarhice), în care un element poate avea mai mulți succesori (fii), dar un singur predecesor (părinte)



Clasificări ale structurilor de date

- În funcție de rolul pe care îl au în aplicații și după operațiile asociate colecției
 - Structuri de căutare (mulțimi și tabele asociative)
 - Structuri de păstrare temporară a datelor (liste, stive, cozi)



Tipuri abstracte de date

- Un tip abstract de date este definit numai prin operațiile asociate (prin modul de utilizare), fără referire la modul concret de implementare (cu elemente consecutive sau cu pointeri sau alte detalii de memorare)



Tipuri abstracte de date

- În programare este utilă o abordare în (cel puțin) două etape:
 - o etapă de concepție (de proiectare), care include alegerea tipurilor abstracte de date și a algoritmilor necesari
 - o etapă de implementare (de codificare), care include alegerea structurilor concrete de date, scrierea de cod și folosirea unor funcții de bibliotecă

Tipuri abstracte de date

- În faza de proiectare nu trebuie stabilite structuri fizice de date, iar aplicația trebuie gândită în tipuri abstracte de date
- În faza de implementare putem decide ce implementări alegem pentru tipurile abstracte decise în faza de proiectare
- Ideea este de a separa interfața (modul de utilizare) de implementarea unui anumit tip de colecție



Eficiența structurilor de date

- Unul din argumentele pentru studiul structurilor de date este acela că alegerea unei structuri nepotrivite de date poate influența negativ eficiența unor algoritmi, sau că alegerea unei structuri adecvate poate reduce memoria ocupată și timpul de execuție a unor aplicații care folosesc intens colecții de date



Eficiența structurilor de date

- Este important ce structuri de date sunt folosite când sunt necesare căutări frecvente într-o colecție de date după conținut
- Căutarea într-un vector sau într-o listă înlănțuită este ineficientă pentru un volum mare de date și astfel au apărut tabele de dispersie (“hash tables”), arbori de căutare echilibrați și alte structuri optimizate pentru operații de căutare



Eficiența structurilor de date

- Influența alegerii structurii de date asupra timpului de execuție al unui program stă și la baza introducerii tipurilor abstracte de date: un program care folosește tipuri abstracte poate fi mai ușor modificat prin alegerea unei alte implementări a tipului abstract folosit, pentru îmbunătățirea performanțelor



Eficiența structurilor de date

- Problema alegerii unei structuri de date eficiente pentru un tip abstract nu are o soluție unică, deși există anumite recomandări generale în acest sens
- Sunt mai mulți factori care pot influența această alegere și care depind de aplicația concretă

Tipuri referință

- Caracterul **&** folosit după tipul și înaintea numelui unui parametru formal (sau al unei funcții) arată compilatorului că pentru acel parametru se primește **adresa**, și nu valoarea argumentului efectiv

Tipuri referință

- Spre deosebire de un parametru pointer, un **parametru referință** este folosit de utilizator în interiorul funcției, la fel ca un parametru transmis prin valoare, dar compilatorul va genera automat indirectarea prin pointerul transmis (în programul sursă nu se folosește explicit operatorul de indirectare *)



//schimbă între ele două valori

```
void schimb ( int & x, int & y ) {
```

```
    int temp;
```

```
    temp = x;
```

```
    x = y;
```

```
    y = temp;
```

```
}
```

//ordonare vector

```
void sort ( int a[ ], int n) {
```

```
.....
```

```
    if (a[ i ] > a[ i + 1 ])
```

```
        schimb ( a[ i ], a [ i + 1 ] );
```

```
.....
```

```
}
```



Tipuri referință

- Referințele **simplifică** utilizarea unor parametri modificabili de tip pointer, eliminând necesitatea unui **pointer la pointer**



Sintaxa declarării unui tip referință

- **tip & nume**

- nume poate fi:

- ☐ numele unui parametru formal
- ☐ numele unei funcții (urmat de lista argumentelor formale)
- ☐ numele unei variabile (mai rar)

Efectul caracterului &

- Compilatorul creează o variabilă *nume* și o variabilă pointer la variabila *nume*, inițializează variabila pointer cu adresa asociată lui *nume* și reține că orice referire ulterioară la *nume* va fi tradusă printr-o indirectare prin variabila pointer omonimă creată

Utilizarea de tipuri generice

- O colecție poate conține valori numerice de diferite tipuri și lungimi sau șiruri de caractere sau alte tipuri agregat (structuri), sau pointeri (adrese)
- Se dorește ca operațiile cu un anumit tip de colecție să poată fi scrise ca funcții generale, adaptabile pentru fiecare tip de date ce poate face parte din colecție

Realizarea unei colecții generice


- 1) Prin utilizarea de tipuri generice pentru elementele colecției în subprogramele ce realizează operații cu colecția
- Pentru a folosi astfel de funcții, ele trebuie adaptate la tipul de date cerut de o aplicație
- Adaptarea se face parțial de către compilator (prin macro-substituție) și parțial de către programator (care trebuie să dispună de codul sursă pentru aceste subprograme)

Realizarea unei colecții generice

- 2) Prin utilizarea unor colecții de pointeri la un tip neprecizat (*void **) și a unor argumente de acest tip în subprograme, urmând ca înlocuirea cu un alt tip de pointer (la date specifice aplicației) să se facă la execuție
- Utilizarea unor astfel de funcții este mai dificilă, dar utilizatorul nu trebuie să intervină în codul sursă al subprogramelor și are mai multă flexibilitate în adaptarea colecțiilor la diverse tipuri de date

Exemplu

- Definirea unui vector cu componente de un tip T neprecizat în funcții, dar precizat în programul care folosește mulțimea
- Funcțiile sunt corecte dacă tipul T este un tip numeric, pentru că operațiile de comparație și de atribuire depind în general de tipul datelor
- Operațiile de citire-scriere a datelor depind de asemenea de tipul T , dar ele fac parte în general din programul de aplicație



```
// mulțimi de elemente de tipul T
#define M 1000           // dimensiune maximă vector
typedef int T ;          // tip componente mulțime
typedef struct {
    T v[M];              // vector cu date de tipul T
    int dim;              // dimensiune vector
} Vector;

    // operații cu un vector de obiecte
void initV (Vector & a ) {    // inițializare vector
    a.dim = 0;
}
```

```
void addV ( Vector & a, T x) {  
    // adaugă pe x la vectorul a  
    assert (a.n < M);  
    // verifică dacă mai este loc în vector  
    a.v [a.n++] = x;  
}  
  
int findV ( Vector a, T x) {    // caută pe x în vectorul a  
    int j;  
    for ( j = 0; j < a.dim; j++)  
        if( x == a.v[j] )  
            return j;           // găsit în poziția j  
    return -1;                  // negăsit  
}
```




Pentru operațiile de atribuire și comparare, avem două posibilități:


a) Definirea unor operatori generalizați, modificați prin macro-substituție:

```
#define EQ(a,b) ( a==b)           // egalitate  
#define LT(a,b) (a < b)          // mai mic decât
```

Exemplu de funcție care folosește acești operatori:



```
int findV ( Vector a, T x) {    // caută pe x în vectorul a
    int j;
    for ( j = 0; j < a.dim; j++)
        if( EQ (x, a.v[j]) )    // comparație la egalitate
            return j;           // găsit în poziția j
    return -1;                  // negăsit
}
```




Pentru o mulțime de șiruri de caractere, trebuie operate următoarele modificări în secvențele anterioare:

```
#define EQ(a,b) ( strcmp(a,b)==0)           // egalitate
#define LT(a,b) ( strcmp(a,b) < 0)          // mai mic
typedef char * T ;
```

b) Transmiterea funcțiilor de comparare, atribuire, ca argumente la funcțiile care le folosesc (fără a impune anumite nume acestor funcții)

Exemplu:

```
typedef char * T;
typedef int (*Fcmp) ( T a, T b);
int findV ( Vector a, T x, Fcmp cmp) {
    // caută pe x în vectorul a
    int j;
    for ( j = 0; j < a.dim; j++)
        if ( cmp (x, a.v[j] ) == 0 )    // comparație la egalitate
            return j;                  // găsit în poziția j
    return -1;                          // negăsit
}
```




Tipul neprecizat T al elementelor unei colecții este de obicei fie un tip numeric, fie un tip pointer (inclusiv de tip *void **)

Avantajul principal al acestei soluții este simplitatea programelor, dar ea nu se poate aplica pentru colecții de colecții (un vector de liste, de exemplu) și nici pentru colecții neomogene

Utilizarea de pointeri generici

- A doua soluție pentru o colecție generică este o colecție de pointeri la orice tip (*void **), care vor fi înlocuiți cu pointeri la datele folosite în fiecare aplicație
- În acest caz, funcția de comparare trebuie transmisă ca argument funcțiilor de inserție sau de căutare în colecție
- Exemplu de vector generic cu pointeri



```
#define M 100           // dimensiune maximă vector
typedef void * Ptr;      // pointer la un tip neprecizat
typedef int (* fcmp) (Ptr, Ptr); // tip funcție de comparare
typedef void (* fprnt) (Ptr); // tip funcție de afișare

typedef struct {         // tipul vector
    Ptr v[M];            // un vector de pointeri
    int dim;             // număr elemente în vector
} Vector;

void initV (Vector & a) { // inițializare vector
    a.dim = 0;
}
```



//afișare date de la adresele conținute în vector

```
void printV ( Vector a, fprintf print ) {
```

```
    int i;
```

```
    for(i = 0; i < a.dim; i++)
```

```
        printf (a.v[i]);
```

```
    printf ("\n");
```

```
}
```

// adăugare la sfârșitul unui vector de pointeri

```
void addV ( Vector & a, Ptr p) {
```

```
    assert (a.dim < M);
```

```
    a.v [a.dim ++] = p;
```

```
}
```



// căutare în vector de pointeri

```
int findV ( Vector v, Ptr p, fcmp cmp) {
```

```
    int i;
```

```
    for (i = 0; i < v.dim; i++)
```

```
        if ( cmp (p, v.v[i]) == 0)
```

```
            return i;
```

```
    return -1;
```

```
}
```

// găsit în poziția i

// negăsit

Avantaje față de colecția cu date de un tip neprecizat

- Funcțiile pentru operații cu colecții pot fi compilate și puse într-o bibliotecă și nu este necesar codul sursă
- Se pot crea colecții cu elemente de tipuri diferite, pentru că în colecție se memorează adresele elementelor, iar adresele pot fi reduse la tipul comun *void**
- Se pot crea colecții de colecții: vector de vectori, listă de liste, vector de liste



Dezavantaj al colecțiilor de pointeri

- Complexitatea unor aplicații, cu erorile asociate unor operații cu pointeri
- Pentru o colecție de numere trebuie alocată memorie dinamic pentru fiecare număr, ca să obținem câte o adresă distinctă, pentru a fi memorată în colecție