Tehnici de programare

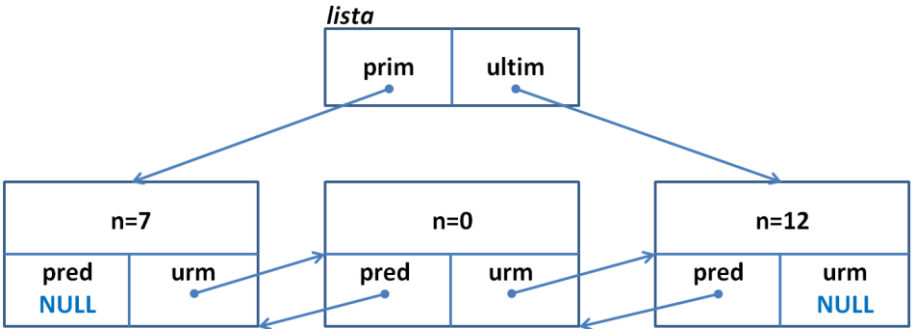
liste dublu înlănțuite; tipuri de date abstracte

În general, pe măsură ce complexitatea unui program crește, structurarea tipurilor de date devine cel puțin la fel de importantă precum algoritmul în sine. O mare parte a succesului de care se bucură limbajele de programare orientate pe obiecte provine din faptul că ele facilitează abstractizarea, încapsularea și refolosirea tipurilor de date. Programele mai mari folosesc diverse tipuri de date și algoritmii vor fi asociați acestora. De exemplu, o bază de date poate să conțină o ierarhie de categorii, iar în fiecare dintre aceste categorii se află produse (ex: produsul *"Stick USB 64GB"* se poate afla în subcategoria *"PC-uri și periferice -> Periferice -> Memorii USB"*. În acest caz, multe din funcțiile folosite în program (adăugare, ștergere, căutare, sortare, etc) vor fi implementate atât pentru produsele în sine cât și pentru ierarhia de categorii. De exemplu, nu vom mai avea funcția *"adaugare"*, fiindcă această funcție nu ne spune nimic despre ce anume se adaugă (produs sau categorie), ci toate aceste nume de funcții vor fi concepute în așa fel încât să fie asociate cu structura de date asupra căreia acționează (ex: *adaugaProdus* și *adaugaCategorie*).

Un **tip de date** (TD) este o asociere între datele în sine (o structură de date, uniune, etc) și toate operațiile definite asupra lor (asociate cu ele). De exemplu, tipul de date *Produs* poate fi asociat cu operațiile adăugare, afișare, etc ce acționează asupra lui. În general funcțiile care definesc aceste operații vor avea pe prima poziție obiectul asupra căruia acționează (ex: *afisareProdus(Produs *p)*). Sunt mai multe convenții de denumire a acestor operații, printre care:

- afisareProdus(Produs *p) - prima oară se pune operația (*afișare*), iar apoi numele structurii de date asupra căreia acționează (*Produs*). Este o convenție care pentru unii utilizatori este mai comodă, deoarece respectă ordinea gramaticală a frazei (predicat complement).
- Produs_afisare(Produs *p) - prima oară se pune numele structurii, iar apoi, după _ (underscore), operația executată. Un avantaj al acestei convenții este faptul că IDE-urile moderne au facilitatea de autocompletare și, atunci când începem să scriem numele unei structuri, vor apărea automat toate operațiile definite pe acea structură. Această convenție reprezintă și translatarea în C a operațiilor din limbajele orientate pe obiecte (ex: *Produs::afisare* în C++).

În continuare vom discuta unele modalități de implementare ale tipurilor de date în C și le vom exemplifica folosind liste dublu înlănțuite. O listă dublu înlănțuită păstrează pentru fiecare element atât un pointer către următorul element din listă (*urm* - următor), cât și un pointer către elementul anterior (*pred* - predecesor). Vom folosi o implementare care păstrează și un pointer către ultimul element din listă:



Avantajul major al listelor dublu înlănțuite față de cele simplu înlănțuite este faptul că, dacă avem un pointer la un element din listă, toate operațiile relative la acel element (ștergere, inserare înainte de el, inserare după el) au complexitatea O(ct). La o listă simplu înlănțuită, ștergerea unui element către care avem un pointer sau inserarea înainte de el au complexitate O(n), deoarece prima oară trebuie parcursă lista pentru a afla predecesorul acelui

element. În situații uzuale, de multe ori se lucrează cu un pointer către un element al listei (de exemplu rezultat dintr-o căutare sau din iterarea listei), astfel încât acest avantaj este semnificativ.

Dezavantajele listelor dublu înlănțuite sunt consumul mai mare de memorie, deoarece fiecare element stochează doi pointeri și complexitatea mai mare a operațiilor, deoarece trebuie menținută sincronizarea mai multor pointeri. Din aceste motive, de obicei listele dublu înlănțuite se folosesc în cazul aplicațiilor care au colecții cu un număr mediu sau mare de elemente (unde complexitatea O(ct) este un avantaj semnificativ față de O(n)) și în care sunt necesare multe operații de inserare sau ștergere.

Putem lua în considerare cazul unui editor de text în care este deschis un document mare. Dacă documentul este implementat cu vectori, pentru fiecare ștergere sau inserare trebuie să mutăm zone de memorie de ordinul MB, ceea ce este foarte lent. Dacă documentul este implementat cu liste simplu înlănțuite și ștergerea sau inserarea au loc înainte de elementul curent, trebuie prima oară să parcurgem lista pana la elementul predecesor celui în care se face modificarea, operație de asemenea lentă. La implementarea cu liste dublu înlănțuite, ștergerea sau inserarea se poate face direct în orice punct dorim, fără a avea nevoie de niciun fel de operații suplimentare și fără a fi necesară reorganizarea elementelor deja existente.

Exemplul 1: Să se implementeze o propoziție folosind o listă dublu înlănțuită. Fiecare element din listă este un cuvânt de maxim 15 caractere sau un semn de punctuație. Programul va avea un meniu cu următoarele opțiuni: 1-propoziție nouă (introducere cuvinte până la punct, exclusiv); 2-afișare; 3-șterge cuvânt; 4-ieșire:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Cuvant{
   char text[16];              // max 15 litere+terminator
   struct Cuvant *pred;        // inlantuire la predecesor
   struct Cuvant *urm;         // inlantuire la urmator
   }Cuvant;

// aloca un nou cuvant si ii seteaza campul text
// campurile pred si urm raman neinitializate
Cuvant *Cuvant_nou(const char *text)
{
   Cuvant *c=(Cuvant*)malloc(sizeof(Cuvant));
   if(!c){
      printf("memorie insuficienta");
      exit(EXIT_FAILURE);
      }
   strcpy(c->text,text);
   return c;
}

typedef struct{
   Cuvant *prim;       // primul cuvant din lista
   Cuvant *ultim;      // ultimul cuvant din lista
   }Propozitie;

// initializare propozitie noua
void Propozitie_init(Propozitie *p)
{
   p->prim=p->ultim=NULL;
}

// adauga un cuvant la sfarsitul propozitiei
```

```
void Propozitie_adauga(Propozitie *p,Cuvant *c)
{
    c->pred=p->ultim;           // predecesorul cuvantului este ultimul cuvant din lista
    if(p->ultim){               // daca mai sunt si alte cuvinte in lista
        p->ultim->urm=c;                // ultimul cuvant din lista va pointa catre noul cuvant
        }else{                  // altfel, daca c este primul cuvant din lista
        p->prim=c;                      // seteaza si inceputul listei la el
        }
    p->ultim=c;                 // seteaza sfarsitul listei pe noul cuvant
    c->urm=NULL;                // dupa cuvantul introdus nu mai urmeaza niciun cuvant
}

// cauta un text in propozitie si daca il gaseste returneaza un pointer la cuvantul respectiv
// daca nu-l gaseste, returneaza NULL
Cuvant *Propozitie_cauta(Propozitie *p,const char *text)
{
    Cuvant *c;
    for(c=p->prim;c;c=c->urm){
        if(!strcmp(c->text,text))return c;
        }
    return NULL;
}

// sterge un cuvant din propozitie
void Propozitie_sterge(Propozitie *p,Cuvant *c)
{
    if(c->pred){                // cuvantul nu este primul in propozitie
        c->pred->urm=c->urm;            // campul urm al predecesorului lui c va pointa la cuvantul de dupa c
        }else{                  // cuvantul este primul in propozitie
        p->prim=c->urm;                 // seteaza inceputul listei pe urmatorul cuvant de dupa c
        }
    if(c->urm){                 // cuvantul nu este ultimul din propozitie
        c->urm->pred=c->pred;           // campul pred al cuvantului de dupa c va pointa la cuvantul de dinainte de c
        }else{                  // cuvantul este ultimul din propozitie
        p->ultim=c->pred;               // seteaza sfarsitul listei pe predecesorul lui c
        }
    free(c);
}

// elibereaza cuvintele din memorie si reinitializeaza propozitia ca fiind vida
void Propozitie_elibereaza(Propozitie *p)
{
    Cuvant *c,*urm;
    for(c=p->prim;c;c=urm){
        urm=c->urm;
        free(c);
        }
    Propozitie_init(p);
}

int main()
{
    Propozitie p;
    int op;     // optiune
    char text[16];
```

```
    Cuvant *c;

    Propozitie_init(&p);                    // initializare propozitie vida
    do{
        printf("1 - propozitie noua\n");
        printf("2 - afisare\n");
        printf("3 - stergere cuvant\n");
        printf("4 - iesire\n");
        printf("optiune: ");scanf("%d",&op);
        switch(op){
            case 1:
                Propozitie_elibereaza(&p);      // elibereaza posibila propozitie anterioara
                for(;;){
                    scanf("%s",text);
                    // intre ultimul cuvant si punct trebuie sa existe un spatiu, pentru ca punctul sa fie considerat separat
                    if(!strcmp(text,"."))break;     // atentie: "." este un sir de caractere, nu o litera (char)
                    Cuvant *c=Cuvant_nou(text);
                    Propozitie_adauga(&p,c);
                    }
                break;
            case 2:
                for(c=p.prim;c;c=c->urm)printf("%s ",c->text);
                printf(".\n");
                break;
            case 3:
                printf("cuvant de sters:");scanf("%s",text);
                c=Propozitie_cauta(&p,text);
                if(c){
                    Propozitie_sterge(&p,c);
                    }else{
                    printf("cuvantul \"%s\" nu se gaseste in propozitie\n",text);
                    }
                break;
            case 4:break;
            default:printf("optiune invalida");
            }
        }while(op!=4);
    return 0;
}
```

La introducerea unui cuvânt trebuie lăsat un spațiu înainte de punct, pentru ca acesta să se considere separat. Fiecare propoziție trebuie inițializată înainte de a fi folosită, folosind *Propozitie_init*. Unele operații, gen eliberarea listei din memorie sau căutarea unui element sunt practic la fel ca la listele simplu înlănțuite. Alte operații, precum adăugarea la sfârșit, sunt ceva mai complexe fiindcă trebuie setați mai mulți pointeri. Și aceste operații sunt similare operațiilor cu liste simplu înlănțuite (implementarea folosind un pointer la ultimul element), deoarece în ambele cazuri complexitatea este O(ct).

Operația care diferențiază în acest exemplu listele dublu înlănțuite de cele simplu înlănțuite este cea de ștergere. Funcția *Propoziție_sterge* primește ca parametri o listă și un pointer la cuvântul de șters, cuvânt care știm sigur că se află în listă. În cazul nostru, acest pointer a fost obținut ca rezultat al operației de căutare în listă. Se poate constata că *Propoziție_sterge* are complexitate O(ct), deoarece numărul de operații executate este constant, nedepinzând de lungimea listei. Dacă am fi folosit liste simplu înlănțuite, pentru a șterge elementul pointat ar fi trebuit prima oară să iterăm toate elementele până la predecesorul elementului de șters, iar apoi să modificăm câmpul *urm* al predecesorului. Astfel, complexitatea operației ar fi fost O(n), deoarece în cazul cel mai defavorabil (elementul de șters se află la sfârșitul listei), ar fi trebuit iterate toate elementele din listă.

Pentru liste scurte, de doar câteva elemente, timpul necesar pentru iterarea unei liste simplu înlănțuite este compensat de timpul mai mic necesar pentru alte operații, deoarece acestea sunt mai simple decât la listele dublu înlănțuite. Dacă în schimb lista trece de un anumit număr de elemente și operațiile de ștergere sau inserare înainte de elementul curent sunt frecvente, atunci deja diferența între cele două tipuri de date devine semnificativă.

Aplicații propuse

Note:
- toate listele sunt dublu înlănțuite
- nu se vor folosi vectori, decât pentru șiruri de caractere

Aplicația 8.1: Să se modifice exemplul 1 astfel încât el să numere de câte ori apare fiecare cuvânt în propoziție. Pentru aceasta, cuvintele vor fi adăugate doar cu litere mici și fiecare cuvânt va avea asociat un contor. Dacă un cuvânt nou nu există în propoziție, el va fi adăugat. Altfel, dacă el există deja, doar se va incrementa contorul cuvântului existent. La afișare, pentru fiecare cuvânt se va afișa și contorul său.

Aplicația 8.2: La exemplul 1 să se adauge operația de inserare a unui cuvânt. Pentru aceasta se cere un cuvânt de inserat și un cuvânt succesor. Dacă succesorul există în propoziție, cuvântul de inserat va fi inserat înaintea sa. Dacă succesorul nu există în lista, cuvântul de inserat va fi adăugat la sfârșitul listei.

Aplicația 8.3: Să se scrie un program care primește un nume de fișier în linia de comandă. Programul va citi toate liniile din fișier într-o listă care este mereu sortată în ordine alfabetică. O linie poate avea maxim 1000 de caractere. Pentru ca lista să fie mereu sortată alfabetic, adăugarea unei linii noi se face prin inserarea ei la poziția corectă din listă, astfel încât să se mențină proprietatea de sortare. În final se va afișa lista.

Aplicația 8.4: Să se scrie un program care implementează o listă de categorii, fiecare categorie având asociată o listă de produse. O categorie se definește prin numele său. Un produs se definește prin nume și preț. Programul va prezenta utilizatorului un meniu cu următoarele opțiuni: 1-adaugă categorie; 2-adaugă produs (prima oară cere o categorie și apoi un produs pe care îl adaugă la acea categorie); 3-afișare categorii (afișează doar numele tuturor categoriilor); 4-afișare produse (cere o categorie și afișează toate produsele din ea); 5-ieșire

Aplicația 8.5: Să se imlementeze o listă dublu înlănțuită care gestionează un parc de automobile. Informaţiile relative la un automobil sunt: codul mașinii (număr între 1 si 9000), numărul de locuri (între 1 și 9), puterea (în cai putere între 1 și 500), marca, culoarea, anul fabricaţiei maşinii (între 1800 și 2017). Parcul conține n automobile, datele sunt citite de la tastatură. Să se scrie următoarele funcții de gestiune a mașinilor:

a) Introducerea unui automobil nou în listă la începutul listei;
b) Ștergerea unui anumit automobil din listă, al cărui cod e citit de la tastatură;
c) Afișarea întregii liste pe ecran;
d) Afișarea pe ecran doar a automobilelor cu un anumit număr de locuri, citit de la tastatură;
e) Ordonarea listei în funcție de anul fabricației.

Să se definească structura pentru o mașină cu ajutorul structurilor cu câmpuri pe biți astfel încât spațiul ocupat să fie minim.