Tehnici de programare

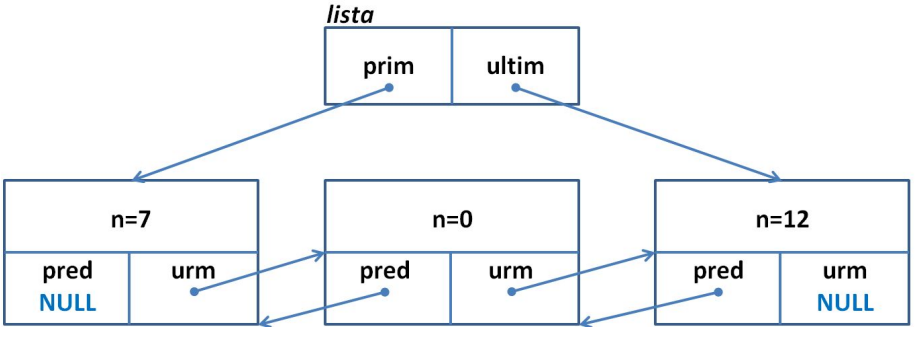
liste dublu înlănţuite; tipuri de date abstracte

În general, pe măsură ce complexitatea unui program creşte, structurarea tipurilor de date devine cel puţin 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 şi refolosirea tipurilor de date. Programele mai mari folosesc diverse tipuri de date şi algoritmii vor fi asociaţi acestora. De exemplu, o bază de date poate să conţină 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 şi periferice -> Periferice -> Memorii USB"*. În acest caz, multe din funcţiile folosite în program (adăugare, ştergere, căutare, sortare, etc) vor fi implementate atât pentru produsele în sine cât şi pentru ierarhia de categorii. De exemplu, nu vom mai avea funcţia *"adaugare"*, fiindcă această funcţie nu ne spune nimic despre ce anume se adaugă (produs sau categorie), ci toate aceste nume de funcţii vor fi concepute în aşa fel încât să fie asociate cu structura de date asupra căreia acţionează (ex: *adaugaProdus* şi *adaugaCategorie*).

Un **tip de date** (TD) este o asociere între datele în sine (o structură de date, uniune, etc) şi toate operaţiile definite asupra lor (asociate cu ele). De exemplu, tipul de date *Produs* poate fi asociat cu operaţiile adăugare, afişare, etc ce acţionează asupra lui. În general funcţiile care definesc aceste operaţii vor avea pe prima poziţie obiectul asupra căruia acţionează (ex: *afisareProdus(Produs *p)*). Sunt mai multe convenţii de denumire a acestor operaţii, printre care:

- afisareProdus(Produs *p) - prima oară se pune operaţia (*afişare*), iar apoi numele structurii de date asupra căreia acţionează (*Produs*). Este o convenţie 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), operaţia executată. Un avantaj al acestei convenţii este faptul că IDE-urile moderne au facilitatea de autocompletare şi, atunci când începem să scriem numele unei structuri, vor apărea automat toate operaţiile definite pe acea structură. Această convenţie reprezintă şi translatarea în C a operaţiilor din limbajele orientate pe obiecte (ex: *Produs::afisare* în C++).

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



Avantajul major al listelor dublu înlănţuite faţă de cele simplu înlănţuite este faptul că, dacă avem un pointer la un element din listă, toate operaţiile relative la acel element (ştergere, inserare înainte de el, inserare după el) au complexitatea O(ct). La o listă simplu înlănţuită, ştergerea 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 situaţii 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 înlănţuite sunt consumul mai mare de memorie, deoarece fiecare element stochează doi pointeri şi complexitatea mai mare a operaţiilor, deoarece trebuie menţinută sincronizarea mai multor pointeri. Din aceste motive, de obicei listele dublu înlănţuite se folosesc în cazul aplicaţiilor care au colecţii cu un număr mediu sau mare de elemente (unde complexitatea O(ct) este un avantaj semnificativ faţă de O(n)) şi în care sunt necesare multe operaţii de inserare sau ştergere.

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

Exemplul 1: Să se implementeze o propoziţie folosind o listă dublu înlănţuită. Fiecare element din listă este un cuvânt de maxim 15 caractere sau un semn de punctuaţie. Programul va avea un meniu cu următoarele opţiuni: 1-propoziţie nouă (introducere cuvinte până la punct, exclusiv); 2-afişare; 3-şterge cuvânt; 4-ieşire:

```
#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 spaţiu înainte de punct, pentru ca acesta să se considere separat. Fiecare propoziţie trebuie iniţializată înainte de a fi folosită, folosind *Propozitie_init*. Unele operaţii, gen eliberarea listei din memorie sau căutarea unui element sunt practic la fel ca la listele simplu înlănţuite. Alte operaţii, precum adăugarea la sfârşit, sunt ceva mai complexe fiindcă trebuie setaţi mai mulţi pointeri. Şi aceste operaţii sunt similare operaţiilor cu liste simplu înlănţuite (implementarea folosind un pointer la ultimul element), deoarece în ambele cazuri complexitatea este O(ct).

Operaţia care diferenţiază în acest exemplu listele dublu înlănţuite de cele simplu înlănţuite este cea de ştergere. Funcţia *Propoziţie_sterge* primeşte ca parametri o listă şi un pointer la cuvântul de şters, cuvânt care ştim sigur că se află în listă. În cazul nostru, acest pointer a fost obţinut ca rezultat al operaţiei de căutare în listă. Se poate constata că *Propoziţie_sterge* are complexitate O(ct), deoarece numărul de operaţii executate este constant,

nedepinzând de lungimea listei. Dacă am fi folosit liste simplu înlănţuite, pentru a şterge elementul pointat ar fi trebuit prima oară să iterăm toate elementele până la predecesorul elementului de şters, iar apoi să modificăm câmpul *urm* al predecesorului. Astfel, complexitatea operaţiei ar fi fost O(n), deoarece în cazul cel mai defavorabil (elementul de şters se află la sfârşitul listei), ar fi trebuit iterate toate elementele din listă.

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

Tipuri de date abstracte (TDA)

După cum se poate constata, cu cât creşte complexitatea unui tip de date, cu atât devine mai complex şi codul (funcţiile) care operează asupra sa. În exemplul de mai sus, chiar dacă s-au implementat doar câteva operaţii, totuşi codul este destul de mare. Ar fi util să putem refolosi cât mai mult din acest cod, de exemplu dacă în loc de o listă de cuvinte avem nevoie de o listă de persoane. Pentru aceasta putem crea un *tip de date abstract* (TDA), care poate opera asupra unei liste indiferent ce tip au elementele sale. În acest sens, tipul elementului este abstractizat, dar tipul colecţiei (listă dublu înlănţuită) rămâne acelaşi. Pentru implementarea TDA în C++ se foloseşte mecanismul *template*, iar în C ne putem folosi de posibilităţile preprocesorului de a genera cod.

Până acum *Propozitie* a fost un *tip de date* (TD) concret, în sensul că implementat specific pentru elemente de tip *Cuvant*. Dacă luăm în considerare implementarea TD *Propozitie*, vom constata următoarele:

- Structura *Propozitie* nu depinde în niciun fel de structura internă a elementului listei. În *Propoziţie* avem doar doi pointeri către o structură care poate conţine orice.
- Funcţiile *Propozitie_init*, *Propozitie_adauga*, *Propozitie_sterge* şi *Propozitie_elibereaza* au nevoie doar ca elementul listei să aibă două câmpuri specifice: *pred* şi *urm*. În afară de aceste câmpuri, funcţiile respective nu mai depind de nimic din structura elementului.
- Funcţia *Propozitie_cauta* depinde de structura elementului listei prin faptul ca foloseşte pentru comparare un câmp numit *text* al elementului, iar acest câmp trebuie să fie de tipul *şir de caractere*. Deci această funcţie depinde de elementele concrete ale listei.

Putem astfel constata că, cu excepţia funcţiei *Propozitie_cauta*, nimic din celelalte funcţii sau structuri de date nu depinde de structura unui element. Singura cerinţă este ca un element să aibă câmpurile *pred* şi *urm*. Cu această constatare, am putea implementa un TDA numit LISTAD (lista dublu înlănţuită), folosind posibilităţile preprocesorului C, astfel încât acest TDA să accepte orice fel de element pentru listă. Pentru acest TDA avem nevoie doar de două elemente: numele unui element şi numele TDA-ului în sine.

În principiu putem implementa tot TDA-ului sub forma unui macro (ex: LISTAD(ELEMENT,NUME)), aşa cum am discutat la laboratorul despre preprocesorul C, dar asta ar însemna să scriem un singur macro foarte mare, în care trebuie să fim atenţi ca fiecare linie să se termine cu \ (backslash). Vom prezenta în continuare o altă metodă pentru a face aceasta:

- tot TDA-ul îl vom implementa într-un fişier antet separat, pe care îl vom denumi ”listad.h”
- înainte de a include ”listad.h” în programul nostru, va trebui să definim în program două definiţii: LISTAD_NUME şi LISTAD_ELEMENT. Prima definiţie este numele TDA pe care vrem să-l generăm (*Propozitie*), iar a doua definiţie este numele elementului listei (*Cuvant*).
- în interiorul ”listad.h” vom folosi peste tot definiţiile de mai sus pentru a genera numele structurii de date şi a funcţiilor asociate. Deoarece pentru funcţii trebuie să concatenăm numele structurii de numele operaţiei, iar operatorul de concatenare ## se poate folosi doar în interiorul unui macro, vom avea un macro LISTAD_FN(LISTAD_NUME,OPERATIE) care va concatena LISTAD_NUME de OPERATIE.
- la sfârşitul fişierului ”listad.h” vom şterge definiţiile LISTAD_NUME şi LISTAD_ELEMENT pentru a le putea refolosi ulterior cu alte valori (redefinindu-le încă o dată şi incluzând încă o dată fişierul ”listad.h”). Ştergerea unei definiţii se poate face cu directiva **#undef** *nume*

Exemplul 2: Programul anterior, folosind TDA-ul LISTAD, va fi compus din două fişiere: ”listad.h”, care implementează TDA-ul şi ”ex2.c”, care foloseşte acest TDA:

```
// listad.h
#if !defined(LISTAD_NUME) || !defined(LISTAD_ELEMENT)
   #error "LISTAD_NUME si LISTAD_ELEMENT trebuie definite inainte de a include listad.h"
#endif

typedef struct{
   LISTAD_ELEMENT *prim;
   LISTAD_ELEMENT *ultim;
   }LISTAD_NUME;

#ifndef LISTAD_FN
   // aceste macrouri sunt definite doar daca listad.h nu a mai fost inclus anterior
   #define LISTAD_FNAUX(NUME,OPERATIE)     NUME##_##OPERATIE
   #define LISTAD_FN(NUME,OPERATIE)     LISTAD_FNAUX(NUME,OPERATIE)
#endif

void LISTAD_FN(LISTAD_NUME,init)(LISTAD_NUME *lista)
{
   lista->prim=lista->ultim=NULL;
}

void LISTAD_FN(LISTAD_NUME,adauga)(LISTAD_NUME *lista,LISTAD_ELEMENT *e)
{
   e->pred=lista->ultim;
   if(lista->ultim){
      lista->ultim->urm=e;
      }else{
      lista->prim=e;
      }
   lista->ultim=e;
   e->urm=NULL;
}

void LISTAD_FN(LISTAD_NUME,sterge)(LISTAD_NUME *lista,LISTAD_ELEMENT *e)
{
   if(e->pred){
      e->pred->urm=e->urm;
      }else{
      lista->prim=e->urm;
      }
   if(e->urm){
      e->urm->pred=e->pred;
      }else{
      lista->ultim=e->pred;
      }
   free(e);
}

void LISTAD_FN(LISTAD_NUME,elibereaza)(LISTAD_NUME *lista)
{
   LISTAD_ELEMENT *e,*urm;
```

```
    for(e=lista->prim;e;e=urm){
        urm=e->urm;
        free(e);
        }
    LISTAD_FN(LISTAD_NUME,init)(lista);
}

// definitiile trebuie sterse pentru a putea refolosi listad.h de mai multe ori in cadrul aceluiasi fisier
#undef LISTAD_NUME
#undef LISTAD_ELEMENT
```

Programul principal va deveni:

```
// ex2.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct Cuvant{
    char text[16];
    struct Cuvant *pred;
    struct Cuvant *urm;
    }Cuvant;

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;
}

#define LISTAD_NUME         Propozitie      // numele TDA
#define LISTAD_ELEMENT      Cuvant          // tipului unui element din lista
#include "listad.h"

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;
}

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

```
  Propozitie_init(&p);
  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);
           for(;;){
              scanf("%s",text);
              if(!strcmp(text,"."))break;
              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 începutul fişierului ”listad.h” s-a testat dacă au fost definite *LISTAD_NUME* şi *LISTAD_ELEMENT*, iar dacă nu au fost definite se emite eroare. Macroul *LISTAD_FN(NUME,OPERATIE)* a avut nevoie de un macrou auxiliar *LISTAD_FNAUX(NUME,OPERATIE)* pentru a se transforma argumentul *NUME* în valoarea sa efectivă (conţinutul definiţiei *LISTAD_NUME*). Dacă s-ar fi folosit doar *”#define LISTAD_FN(NUME,OPERATIE) NUME##_##OPERATIE”*, atunci ”*LISTAD_FN(LISTAD_NUME,init)*” s-ar fi înlocuit prin ”*LISTAD_NUME_init*”.

Atât *LISTAD_FNAUX* cât şi LISTAD_FN se definesc doar dacă este prima includere a fişierului ”listad.h”. Dacă acest fişier a mai fost inclus anterior, macrourile respective sunt deja definite, astfel încât nu mai trebuie să fie definite încă o dată. Spre deosebire de ele, definiţiile *LISTAD_NUME* şi *LISTAD_ELEMENT* se şterg la sfârşitul fişierul *”listad.h”*, astfel încât ele să poată fi redefinite ulterior cu alte valori.

În programul principal (*ex2.c*) a fost nevoie doar de trei linii pentru a folosi TDA-ul *LISTAD*: definirea celor două definiţii *LISTAD_NUME* şi *LISTAD_ELEMENT* şi includerea fişierului ”listad.h”. Dacă ar fi fost nevoie de mai multe structuri de date care să folosească acest TDA, el ar fi putut fi inclus de mai multe ori, cu *LISTAD_NUME* şi

LISTAD_ELEMENT având valori diferite. Se poate constata că funcţia *Propozitie_cauta* nu a putut fi inclusă în TDA, deoarece ea presupune că elementul listei trebuie să aibă un câmp numit *“text”*, de tipul *şir de caractere*.

Aplicaţii propuse

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

Aplicaţia 10.1: Să se modifice exemplul 1 astfel încât el să numere de câte ori apare fiecare cuvânt în propoziţie. Pentru aceasta, cuvintele vor fi adăugate doar cu litere mici şi fiecare cuvânt va avea asociat un contor. Dacă un cuvânt nou nu există în propoziţie, el va fi adăugat. Altfel, dacă el există deja, doar se va incrementa contorul cuvântului existent. La afişare, pentru fiecare cuvânt se va afişa şi contorul său.

Aplicaţia 10.2: La exemplul 1 să se adauge operaţia de inserare a unui cuvânt. Pentru aceasta se cere un cuvânt de inserat şi un cuvânt succesor. Dacă succesorul există în propoziţie, cuvântul de inserat va fi inserat înaintea sa. Dacă succesorul nu există în lista, cuvântul de inserat va fi adăugat la sfârşitul listei.

Aplicaţia 10.3: Să se adauge la TDA-ul LISTAD operaţia de inserare cu 3 parametri: lista de inserat (*lista*), elementul înainte de care se face inserarea (*pos*) şi elementul de inserat (*e*). Dacă *pos!=NULL*, atunci *e* se va insera înainte de *pos*. Dacă *pos==NULL*, atunci *e* se va insera la sfârşitului listei.

Aplicaţia 10.4: Folosind TDA-ul LISTAD să se declare două tipuri de liste: una care conţine elemente de tip *int* şi alta care conţine elemente de tip *double*. Se citesc de la tastatură numere reale până când se tastează 0. Dacă numerele nu au parte zecimală, se vor depune în prima listă, altfel în cea de a doua. În final se vor afişa ambele liste.

Aplicaţia 10.5: Să se scrie un program care primeşte un nume de fişier în linia de comandă. Programul va citi toate liniile din fişier 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 poziţia corectă din listă, astfel încât să se menţină proprietatea de sortare. În final se va afişa lista.

Aplicaţia 10.6: Să se scrie un program care implementează o listă de categorii, fiecare categorie având asociată o listă de produse. O categorie se defineşte prin numele său. Un produs se defineşte prin nume şi preţ. Programul va prezenta utilizatorului un meniu cu următoarele opţiuni: 1-adaugă categorie; 2-adaugă produs (prima oară cere o categorie şi apoi un produs pe care îl adaugă la acea categorie); 3-afişare categorii (afişează doar numele tuturor categoriilor); 4-afişare produse (cere o categorie şi afişează toate produsele din ea); 5-ieşire