4. Liste.

Generalităţi.

O listă liniară este o secvenţă finită de elemente: $a_1, a_2, \ldots, a_n$ ($n \geq 0$), având toate acelaşi tip (lista este o *colecţie omogenă* de elemente). Se definesc:
- **n** lungimea listei
- a_1 primul element din listă
- a_n ultimul element din listă
- **p** poziţia elementului a_p

Elementele listei sunt aranjate liniar, în sensul că fiecare element a_p al listei (exceptând ultimul, $p<n$) are un *succesor* a_{p+1} şi un *predecesor* a_{p-1}(exceptând primul, $p>1$).

Ordinea elementelor este semnificativă în listă şi un element poate apare de mai multe ori (poate avea duplicate).

Numărul elementelor din listă **n** defineşte *lungimea listei*. O listă cu lungimea zero (fără elemente) este o *listă vidă*.

Într-o listă **L**, de lungime **n**, fiecărui element $x \in L$ i se asociază în mod unic o *poziţie* **p** sau *iterator*. Aceasta poate fi un întreg numit şi *index* caz în care $1 \leq p \leq n$ sau poate fi o adresă (pointer). Accesul la elementele listei se poate face pe baza poziţiei.

În cazul implementării listelor prin tablouri, poziţia unui element corespunde *indicelui* acestuia în tablou. Dacă lista se implementează printr-o listă înlănţuită, atunci se admite înlocuirea poziţiei întregi printr-un *pointer la element* (*adresă de nod*)

Operaţii specifice listelor.

Asupra unei liste se pot efectua următoarele ***operaţii fundamentale***:

- *inserarea unui element*, adică adăugarea unui element **x** listei **L** în orice poziţie (inclusiv înaintea primului element sau după ultimul element). Distingem o inserare înaintea unei poziţii şi o inserare după o poziţie, prima situaţie fiind considerată mai generală.
- *ştergerea unui element*, se poate face indicând o poziţie în listă a elementului ce urmează a fi şters sau o valoare de element, caz în care se şterge prima apariţie a valorii în listă.
- *căutarea unui element,* având ca rezultat *poziţia din listă* sau *poziţia de după ultimul element* (sau **NULL**), după cum elementul **x** există sau nu în listă.

Pentru o listă nevidă a fost introdusă noţiunea de *capul listei* (*head*) pentru a desemna primul element din listă şi *restul* sau *coada listei* (*tail*) – pentru ansamblul celorlalte elemente.

Lista poate fi *ordonată* sau *neordonată*. În cazul unei liste ordonate, elementele listei aparţin unei mulţimi ordonabile liniar şi fiecare element respectă o *regulă de precedenţă* faţă de elementul următor. Pentru o listă ordonată crescător regula de precedenţă este ≤..

Sunt posibile două modalităţi de parcurgere a unei liste*: înainte* – începând cu primul element şi avansând spre ultimul şi *înapoi* – se începe cu ultimul şi se avansează până la primul.

Pentru traversare se utilizează un *iterator*, caracterizat prin:

- poziţia de start (poziţia primului element la traversarea înainte, sau a ultimului element la traversarea înapoi)

- poziţia de oprire (după ultimul element la traversarea înainte, sau înaintea primului element la traversarea înapoi)

- direcția de avans (înainte – către elementul următor sau înapoi – către elementul precedent).

Elementele listei pot fi de orice tip predefinit: **`int, double, char`**, şir de caractere sau mai general structuri definite de utilizator. În specificarea TAD se foloseşte **`TipElem`**.

Pentru a folosi liste cu elemente de orice tip (liste generice) sunt posibile două soluții:

1. definirea în fişierul de interfață a tipului elementelor listei. Exemplu:

```
typedef double TipElem;
```

Soluția nu este flexibilă, deoarece utilizarea a două liste cu elemente de tip diferit presupune duplicarea codului.

2. un element al listei conține nu o valoare ci o referință (un pointer) la o dată, care poate fi de orice tip (pointer generic **`void*`**).

Alocarea de memorie este lăsată pe seama utilizatorului (clientului), ceea ce presupune următoarele operații suplimentare:

- la inserarea unui element în listă (cu **`L_Insert()`**), acesta trebuie să fie alocat şi referit printr-un pointer

- la inspectarea unui element din listă, referit de un iterator (cu **`L_Get()`**), întrucât acesta rămâne în listă, se alocă, printr-un pointer memorie, în care se copiază elementul extras

- la preluarea unui element care se şterge din listă cu **`L_Remove()`**, este necesar numai un pointer, care să se lege la informația extrasă.

Specificarea TDA listă.

Domeniile folosite de TDA sunt: **`TipElem, Pozitie, Lista, int.`**
Semnăturile funcțiilor specifice TAD Listă sunt:

- Inițializează o listă vidă **`new : → Lista`**
- Şterge o listă: **`delete: Lista →`**
- Determină lungimea unei liste: **`size : Lista → int`**
- Test listă vidă: **`empty: Lista → int`**
- Afişarea elementelor unei liste: **`print : Lista →`**
- Şterge elementul dintr-o poziție dată**.** Răspuns valoarea elementului şters.
  `remove : Lista × Pozitie → TipElem`
- Caută asociativ o valoare în listă (răspuns poziția primei apariții)
  `find : Lista × TipElem → Pozitie`
- Extrage elementul dintr-o poziție dată
  `get : Lista × Pozitie → TipElem`
- Modifică valoarea elementului dintr-o poziție dată.
  `modif : Lista × Pozitie × TipElem → Lista`
- Inserează o valoare *înaintea* unei poziții date
  `insert : Lista × Pozitie × TipElem → Lista`
- Poziționare pe primul element din listă **`begin : Lista → Pozitie`**
- Poziționare pe ultimul element din listă **`rbegin : Lista → Pozitie`**
- Test sfârşit de listă (poziție după ultimul element) **`end : Lista → Pozitie`**
- Test poziție dinaintea primului element) **`rend : Lista → Pozitie`**
- Poziționare pe elementul următor din listă **`next : Lista × Pozitie → Pozitie`**
- Poziționare pe elementul precedent din listă **`prev : Lista × Pozitie → Pozitie`**

Traversarea înainte (respectiv înapoi) a listei poate fi reprezentată prin:

```
for(p=begin(L); p!=end(L); p=next(L,p)
      vizitare element din poziția p;
```

respectiv

```
for(p=rbegin(L); p!=rend(L); p=prev(L,p)
      vizitare element din poziția p;
```

Poziția de după ultimul elementare sens şi pentru o listă vidă. Poziția primului element din listă nu are sens pentru o listă vidă; o vom defini totuşi pentru o listă vidă, ca poziția după ultimul element. În mod asemănător, la o traversare înapoi a listei, poziția de start (poziția ultimului element), pentru o listă vidă se consideră poziția după ultimul element traversat (poziția dinaintea primului element). Cu aceste convenții operațiile rămân consistente pentru o listă vidă (ciclurile se repetă de zero ori).

În cazul inserării înaintea unei poziții realizată cu **`insert(L, p, x)`**, adăugarea de elemente la sfârşitul listei se face cu: **`insert(L, end(L), x)`**.

În caz că lista este vidă, parametrul **`p`** nu este semnificativ, inserarea trebuind să asigure creerea unei liste cu un element.

Remarcăm că inserarea înaintea unei poziții **`p`**, în caz că lista este implementată printr-un tablou presupune modificarea lui **`p`**, deoarece toate elementele, începând cu cel din poziția **`p`** se deplasează cu o poziție la dreapta. Elementul inserat va ocupa vechea poziție a lui **`p`**. Elementul înaintea căruia s-a făcut inserarea se va afla în poziția **`p+1`**, deci poziția lui este modificată.

În cazul ştergerii unui element indicat de iteratorul **`p`**, după ştergere, iteratorul ar fi invalidat, întrucât elementul pe care l-a indicat nu mai există. Poziția **`p`**, după ştergere va referi elementul situat după elementul şters.

Aceste considerații ne determină, ca la implementarea funcțiilor de inserare şi de ştergere, iteratorul **`p`** să fie declarat ca un parametru variabil.

Interfața TDA Lista.

Un program care utilizează un TDA este un program multifişier. El cuprinde în mod minimal următoarele fişiere: aplicația, interfața şi implementarea.
Interfața cuprinde:

- definirea tipurilor şi structurilor de date folosite de TDA
- declararea operațiilor TDA, care este independentă de implementare

Definirea listei, care este specifică implementării, este ascunsă în fişierul de implementare

Atât aplicația cât şi implementarea includ interfața. Pentru a se evita includerea multiplă, în interfață se utilizează o directivă condițională de forma:

```
#ifndef _SIMBOL
#define _SIMBOL
  declarații operații TAD
#endif
```

Ca exemplificare considerăm TDA Listă, implementată cu tablouri şi fişierele: **`test.c`** – aplicația cu liste, **`Alist.h`** – interfața şi **`Alist.c`** – implementarea.

Sunt necesare următoarele operații:
- compilarea implementării: **`gcc –c –o Alist.o Alist.c`**
- compilarea aplicației: **`gcc –c –o test.o test.c`**
- creerea executabilului: **`gcc –o test test.o Alist.o`**

Cele trei operații se realizează economic cu utilitarul **`make`**.

```
//fisierul de interfata List.h
#ifndef _LIST_H
#define _LIST_H
// reprezentare specifica implementarii
struct nod; //definirea este ascunsa in implementare
struct lista;
typedef struct lista *Lista;          //pointer opac
typedef struct nod   *List_Iter;  //implementarea cu liste inlantuite
typedef int List_Iter;            //implementarea cu tablouri

typedef int (*PFC)(void*, void*);     //pointer la functie

// declarare operatii independente de implementare
// initializare lista vida (constructor)
Lista L_New();        // implementarea cu liste inlantuite
Lista L_New(int cap); //implementarea cu tablouri

void L_Delete(Lista *L); // sterge lista (destructor)

int L_Empty(Lista L); // test lista vida
int L_Size(Lista L);  // lungime lista

// intoarce adresa elementului din pozitia p
void *L_Get(Lista L, List_Iter p);

// modifica elementul din pozitia p
void L_Modif(Lista L, List_Iter p, void *x);

// intoarce pozitia elementului in lista L
//(NULL daca nu este)
List_Iter L_Find(Lista L, void *x, PFC egal);

// sterge elementul din pozitia p
void *L_Remove(Lista L, List_Iter p);

// insereaza un element inainte de pozitia p
void L_Insert(Lista L, List_Iter p, void *x);

// intoarce pozitia primului element din lista
List_Iter L_Begin(Lista L);

// intoarce pozitia ultimului element din lista
List_Iter L_RBegin(Lista L);

// test pozitie dupa ultimul nod
List_Iter L_End(Lista L);

// test pozitie dinaintea primului nod
List_Iter L_REnd(Lista L);

// intoarce pozitia urmatoare pozitiei p
List_Iter L_Next(Lista L, List_Iter p);

// intoarce pozitia dinaintea pozitiei p
List_Iter L_Prev(Lista L, List_Iter p);
#endif
```

Aplicații cu liste..

1. Prezentăm mai întâi complet o aplicație simplă:

```
#include <stdio.h>
#include <stdlib.h>
#include "List.h"

//afisarea elementelor din lista de întregi
void L_Print (Lista L){
  List_Iter p;
  for(p=L_Begin(L); p!=L_End(L); p=L_Next(L,p))
    printf("%2d ", *(int*)L_Get(L));
  printf("\n");
}

//compararea a 2 elemente referite prin pointeri generici
int fcint(void *x, void *y){
   return *(int*)x - *(int*)y;
}

//copiază x în memorie alocata dnamic; intoarce pointer la memorie
int *aloca(int x){
   int *a;
   a = (int*)malloc(sizeof(int));
   *a = x;
   return a;
}

int main(){
  Lista L;
  //creaza o lista vida ce poate avea cel mult 10 elemente
  L = L_New(10);
  printf("Lungime initiala:%2d\n", L_Size(L));
  List_Iter p;
  p = L_Begin(L);
  L_Insert(L, p, aloca(3));
  L_Insert(L, p, aloca(5));
  p = L_Begin(L);
  L_Insert(L, p, aloca(2));
  p = L_End(L);
  L_Insert(L, p, aloca(10));
  L_Print (L);
  int *x5 = (int*)malloc(sizeof(int));
  *x5 = 3;
  //cauta elementul referit de x5 in lista {3,5,2,10}
  p = L_Find(L, x5, fcint);
  if(p==0)
    printf("Elementul nu a fost gasit\n");
  else
    printf("Element gasit in pozitia %2d\n",p);
  L_Delete(L);
}
```

2. Sa se creeze o listă **L** având elemente valori întregi, introduse de la tastatură şi terminate prin 0, care nu aparţine listei şi apoi să se elimine duplicatele elementelor.

```
/* crearea unei noi liste */
void creare (Lista *L){
  int x, *px;
  List_Iter p;
  L = L_New();
  p = L_Begin(L);
  do{
     scanf("%d", &x);
     if (x != 0) {
        px = (int*)malloc(sizeof(int));
        *px = x;
       L_Insert (L, p, px);
       p = L_Next(L, p);
    }
  } while(x);
}

void duplicate (Lista L){
  List_Iter p,  q;
  int x1, x2;
  p = L_Begin(L);
  while (p!=L_End(L)){
     x1 = *(int*)L_Get(L, p);
     q = L_Next(L, p);
     while (q!=L_End(L)){
        x2 = *(int*)L_Get(L, q);
        if (x1==x2)
           L_Remove(L, &q);
        else
           q = L_Next(L, q);
     }
     p = L_Next(L, p);
  }
}
```

3. Două numere lungi, care pot avea lungimea cel mult 100 de caractere sunt citite de la tastatură ca două şiruri de caractere. Să se calculeze şi să se afişeze suma acestora. Un număr lung va fi păstrat ca o listă de caractere, cu primul element – cifra cea mai semnificativă.

Pentru adunarea cifrelor începând cu cele mai puţin semnificative se face o traversare înapoi a listelor. După epuizarea numărului cu lungime mai mică se continuă cu celălalt număr, cu eventuala propagare a transportului. Inserarea cifrelor în numărul lung sumă se face la începutul listei.

```
Lista suma(Lista L1, Lista L2){
  Lista L=L_New();
  Lista L3;
  int l1, l2, lm, lM;           /*lungime liste */
  int i;
  l1=L_Size(L1);
  l2=L_Size(L2);
```

```
  lm=(l1>l2)? l2 : l1;
  lM=(l1>l2)? l1 : l2;
  L_Iter p1, p2;
  void *x;
  int s, t=0;                    /*suma si transport */
  p1=L_RBegin(L1);
  p2=L_RBegin(L2);
  for(i=0; i<lm; i++){           /*aduna pe lungime comuna*/
     s = t + *(char*)L_Get(L1,p1) + *(char*)L_Get(L2,p2)-2*'0';
     t = s/10;
     s = s%10;
     x = (char*)malloc(1);
     *x = s;
     L_Insert(L, L_Begin(L), x);
  }
  if(lm==l1)
     L3 = L2;
  else
     L3 = L1;
  /*adauga cifrele celui mai lung si propaga transport*/
  for( ; i<lM; i++){
     s = t +  *(char*)L_Get(L3,p2) - '0';
     t = s/10;
     s = s%10;
     x = (char*)malloc(1);
     *x = s;
     L_Insert(L, L_Begin(L), x);
  }
  return L;
}
```

În funcția **main()** se citesc cele două şiruri de caractere reprezentând cele două numere lungi, se crează listele prin inserarea cifrelor numerelor, se adună listele şi se face o traversare a listei sumă, afişând cifrele extrase.

```
int main(){
  char s1[100], s2[100];         /*numerele lungi*/
  char *x;                       /*pointer la element*/
  L_Iter p;
  //citire numere
  gets(s1);
  gets(s2);
  //initializare liste
  Lista L1=L_New();
  Lista L2=L_New();
  Lista L =L_New();  /*lista suma*/
  //creere liste prin inserare elemente din s1 si s2
  for(x=s1; x; x++)
    L_Insert(L1, L_End(L1), x);
  for(x=s2; x; x++)
    L_Insert(L2, L_End(L2), x);
  L = suma(L1, L2);
```

```
  for(p=L_Begin(L); p!=L_End(L); p=L_Next(p,L))
     printf("%c", *(char*)L_Get(L, p)+"0');
  printf("\n");
  return 0;
}
```

4. **n** cursuri sunt specificate prin numele cursului şi numărul maxim de studenți care-l pot urmări. Opțiunile studenților pentru aceste cursuri sunt precizate prin numele, prenumele şi numele cursului. Aceste elemente sunt trecute toate pe o linie, fiind separate prin spații libere.

Un student poate avea mai multe opțiuni pentru cursuri diferite, numele lui apărând în fiecare opțiune.

Opțiunile studenților sunt considerate în ordinea aparției lor. După înscrierea numărului maxim admis de studenți la un curs, opțiunile ulterioare pentru acest curs sunt ignorate.

Stabiliți pentru fiecare curs, câți studenți s-au înscris, şi care sunt aceştia.

Pentru fiecare student, care a avut cel puțin o opțiune luată în considerare, stabiliți care sunt cursurile la care a fost înscris.

Rezolvare: Fiecare curs va fi specificat prin: nume curs, număr maxim studenți admişi şi lista studenților înscrişi. Numărul de locuri ocupate va fi lungimea acestei liste.

```
typedef  struct{
   int   max;
   char  *numecrs;
   Lista studenti;
} curs;
```

Un student va fi precizat, în final prin nume şi o listă de cursuri la care a fost înscris:

```
typedef{
   char  *numestd;
   Lista cursuri;
} stud;
```

Vom crea o listă de cursuri **Lista c** şi o listă de studenți **Lista s**. O prezentare a algoritmului în pseudocod este:

```
inițializare listă cursuri c
for fiecare curs
   citeşte nume curs şi număr maxim studenți
   inițializare listă studenți înscrişi la curs
inițializare listă studenți s
for fiecare opțiune
   citeşte şi separă nume student şi nume curs
   caută nume curs în lista de cursuri c
   dacă nu s-a găsit ignoră opțiunea
   altfel
      dacă nu s-a depăşit capacitatea
         inserează nume în lista studenților înscrişi la curs
      altfel
         ignoră opțiunea
   caută nume student în lista studenților s
   dacă este găsit
      adaugă curs la lista cursurilor studentului
   altfel
      adaugă student la lista studenților
```

```
      adaugă curs în lista cursurilor studentului
for fiecare curs din lista c
   afisează nume curs şi număr studenţi înscrişi
   afişează lista studenţilor înscrişi la acest curs
for fiecare student din lista s
   afişează nume student şi număr cursuri urmărite
   afişează listă cursuri urmate


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Lista.h"

typedef  struct{
   int   max;
   char  *numecrs;
   Lista studenti;
} curs;

typedef struct{
   char *numestd;
   Lista cursuri;
} stud;

char* strdup(char *s){
   char *p = (char*)malloc(strlen(s)+1);
   strcpy(p, s);
   return p;
}

int cpsir(void *p, void *q){
    return strcmp((char*)p, (char*)q);
}
int main(){
   Lista c;
   Lista s;
   Lista t;
   int n, i;
   List_Iter p, q;
   char linie[80], *pl, *pch;
   curs *pc;
   stud *ps;
   c = L_New();
   scanf("%d", &n);
   for(i=0; i<n; i++){
      gets(linie);              /*citeste nume curs si nr.max stud*/
      pl = strrchr(linie, ' ');  /*separa nr.max stud la curs*/
      *pl = 0;
      pc = (curs*)malloc(sizeof(curs));
      pc->max = atoi(pl+1);
      pc->numecrs = strdup(linie);
```

```
      pc->studenti = L_New();
      L_Insert(c, L_End(c), pc);
   }
   s = L_New();
   while(gets(linie)){
      pl = strrchr(linie, ' ');
      *pl = 0;
      pch = strdup(pl+1);
      p = L_Find(s, pl, cpsir);
      if(p==L_End(s)){
         ps = (stud*)malloc(sizeof(stud));
         ps->numestd = strdup(linie);
         ps->cursuri = L_New();
         L_Insert(ps->cursuri, L_End(ps->cursuri), pch);
         L_Insert(s, L_End(s), ps);
      }
      else{
         t = ((stud*)L_Get(s, p))->cursuri;
         L_Insert(t, L_End(t), pch);
      }
   }
   for(p=L_Begin(c); p!=L_End(c); p=L_Next(c,p)){
      printf("%s\n", ((curs*)L_Get(c, p))->numecrs);
      t = ((curs*)L_Get(c, p))->studenti;
      printf("%3d inscrisi\n", L_Size(t));
      for(q=L_Begin(t); q!=L_End(t); q=L_Next(t,q))
         printf("%s\n", (char*)L_Get(t, q));
   }
   for(p=L_Begin(s); q!=L_End(s); p=L_Next(s,p))
      printf("%s\n", ((stud*)L_Get(s, p))->numestd);
      t = ((stud*)L_Get(s,p))->cursuri;
      printf("urmeaza %2d cursuri\n", L_Size(t));
      for(q=L_Begin(t); q!=L_End(t); q=L_Next(t,q))
         printf("%s\n", (char*)L_Get(t, q));
   return 0;
}
```

Implementarea listelor.

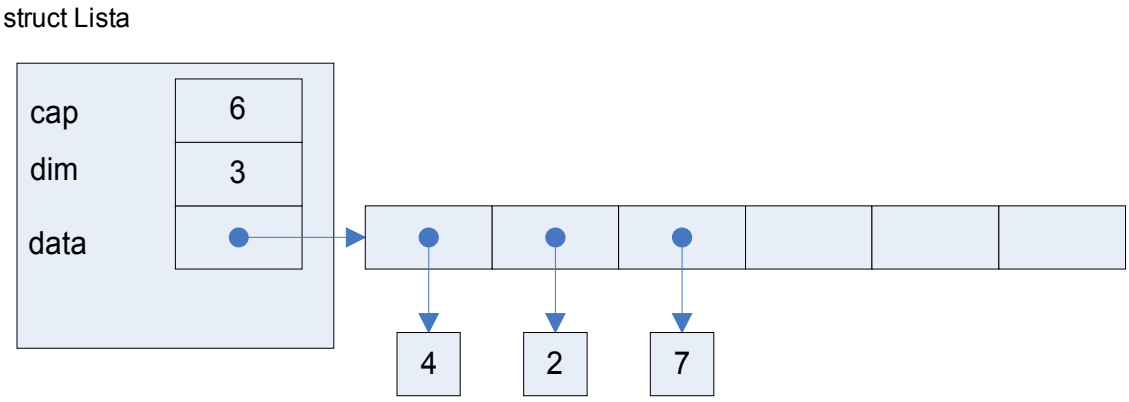
1. Implementare cu tablouri.

Elementele listei sunt memorate într-un tablou, alocat dinamic prin funcția de inițializare a listei.
Vom distinge între numărul elementelor alocate pentru tablou **cap** şi numărul de elemente curent folosite **dim**; în general **dim ≤ cap**.

```
// aceasta definitie apare in fisierul de implementare
struct lista {
           int  dim;    //dimensiune efectiva
           int  cap;    //capacitate (dimensiune alocata)
           void **data; //tablou pointeri la date
};
```

Pentru liste reprezentate prin tablouri poziția este un întreg (index).

```
typedef int List_Iter;
```



Prin inițializarea listei se face o alocare estimativă **cap** şi lungimea efectivă este **dim=0**.

```
Lista L_new(int cap){
  Lista L = (Lista)malloc(sizeof(struct lista));
  L->dim = 0;
  L->cap  = cap;
  L->data = (void**)malloc(L->cap * sizeof(void*));
  return L;
}
```

Lungimea unei liste şi testul de listă vidă inspectează pur şi simplu câmpul **dim** al structurii **Lista**.în caz că lista există.

```
int L_Size(Lista L) {
   assert(L);
   return L->dim;
}

int L_Empty(List L) {
   assert(L);
   return L->dim==0;
}
```

Complexitatea operațiilor **L_Size()** şi **L_Empty()** este $\Theta(1)$

Validarea unui indice (poziție) se face verificând încadrarea sa între limite:

```
int VerifPoz(Lista L, List_Iter p){
   assert(L);
   if(L_Empty(L))
      return 1;
   if(p < 1 || p > L->dim)
      return 0;
   return 1;
}
```

Complexitate **VerifPoz()** este $\Theta(1)$.

Traversarea înainte presupune: poziționarea pe primul element din listă, furnizarea poziției de după ultimul element şi avansul la următorul element.

```
List_Iter L_Begin(Lista L){
```

```
   assert(L );
   return 1;
}

List_Iter L_End(Lista L){
   assert(L);
   return L_dim+1;
}

List_Iter L_Next(Lista L, List_Iter p){
   assert(L && !L_Empty(L));
   return p+1;
}
```

Traversarea înapoi presupune: poziționarea pe ultimul element din listă, furnizarea poziției de dinainte de primul element şi trecerea la elementul precedent.

```
List_Iter L_RBegin(Lista L){
   assert(L);
   return L_dim;
}

List_Iter L_REnd(Lista L){
   assert(L);
   return 0;
}

List_Iter L_Prev(Lista L, List_Iter p){
   assert(L && !L_Empty(L));
   return p-1;
}
```

Extragerea unui element dintr-o poziție dată, verifică în prealabil că poziția există în listă:

```
void *L_Get(Lista L, List_Iter p){
   assert(L && !L_Empty(L) && VerifPoz(L,p));
   return L->data[p-1];
}
```

Funcția de modificare a unui element dintr-o poziție dată, verifică de asemeni validitatea poziției:

```
void L_Modif(Lista L, List_Iter p, void *x){
   assert(L && !L_Empty(L) && VerifPoz(L,p));
   L->data[p-1] = x;
}
```

Complexitatea funcțiilor `L_Get()` şi `L_Modif()` este $\Theta(1)$

Căutarea unui element întoarce poziția primei apariții a acestuia sau poziția după ultimul element dacă nu este găsit (pozițiile în listă încep cu 1).

```
List_Iter L_Find(Lista L, void *x, PFC Egal){
  List_Iter p;
  assert(L);
  if(L_Empty(L))
    return L_End(L);
  for(p=L_Begin(L); p!=L_End(L,p); p=L_Next(L,p))
    if((*Egal)(L_Get(L, p), x)) return p;
  return L_End(L);
```

```
}
```

Complexitatea funcției **L_Find()** este **Θ(size)**

Inserarea, respectiv ştergerea unui element creşte, repectiv scade **dim**.
Ştergerea elementului din pozitia **p** deplasează la stânga elementele situate după această poziție.

```
void *L_Remove(Lista L, List_Iter *p){
   void *rez=NULL;
   int i;
   assert(L && !L_Empty(L) && VerifPoz(L,*p));
   rez = L->data[*p-1];
   for(i=*p-1; i<L->dim-1; i++)
      L->data[i] =L->data[i+1];
   L->dim--;
   return rez;
}
```

Complexitatea funcției **L_Remove()** este **Θ(size).**

Modificarea dimensiunii alocate **cap** se va face dacă **dim=cap** şi se face o inserare.
Pentru a evita o realocare frecventă, vom folosi ca strategie dublarea dimensiunii.

```
void ModifDim(Lista *L){
  Lista L1;
  L1 = L_New(2*(*L)->cap);
  L1->dim = (*L)->dim;
  memcpy(L1->data, (*L)->data, (*L)->cap);
  L_Delete(L);
  *L = L1;
}
void L_Insert(Lista L, List_Iter p, void *x){
   assert(L && VerifPoz(L, p));
   int i;
   if(L_Empty(L)){
     l->data[0] = x;
   }
   else
   { if(L->dim == L->cap)
        ModifDim(&L);
     for(i=L->dim; i>=p; i--)
        L->data[i]=L->data[i-1];
     L->data[p-1] = x;
   }
   L->dim++;
}
```

Complexitatea funcțiilor **L_Insert()** şi **ModifDim()** este **Θ(size).**

4.4.2. Implementare cu listă dublu înlănțuită.

Vom considera lista ca o structură dinamică, alcătuită din noduri, dispersate în memorie. Un nod va conține în afara unei referințe la valoarea elementului adresele de legătură la nodul următor şi la nodul precedent.

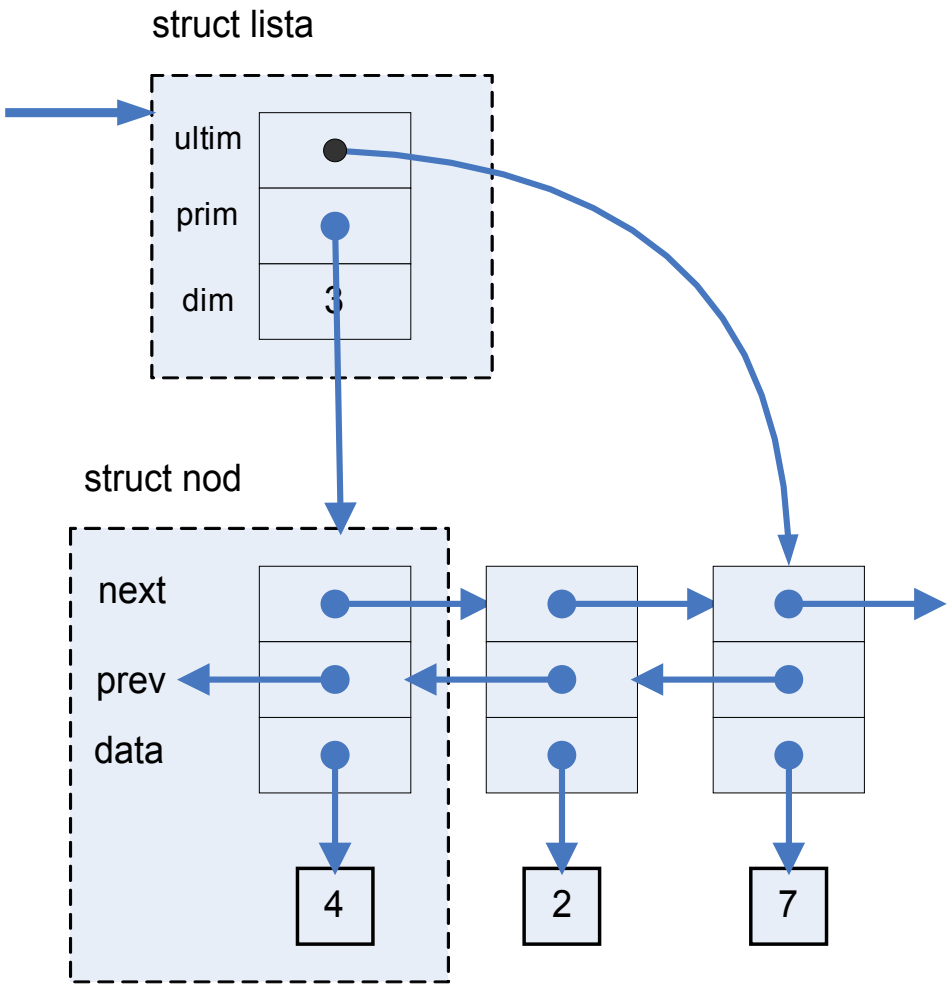
```
struct nod {
      void       *data;
      struct nod *next;
      struct nod *prev;
};
```

O poziție în listă va fi o adresă de nod, tip declarat prin:

```
typedef struct nod *List_Iter;
```

Tipul **Lista** va fi un pointer la o structură formată din adresele primului nod, a ultimului nod şi din lungimea listei:

```
struct lista{
      struct     nod *prim;
      struct     nod *ultim;
      int        dim;
};
```



Funcția de inițializare a unei liste alocă memorie structura listă şi îi inițializează câmpurile la 0.

```
Lista L_New(){
  Lista L = (Lista)malloc(struct lista);
  L->dim = 0;
  L->prim = NULL;
  L->ultim = NULL;
  return L;
}
```

Traversarea înainte a listei presupune: poziționarea pe primul element, testul că s-a ajuns la ultimul element din listă şi avansul la următorul element.

Poziția primului element din listă este indicată de câmpul **prim**:

```
List_Iter L_Begin(Lista L){
   assert(L);
   return L->prim;
}
```

Ultimul nod din listă are legătura la elementul următor nulă

```
List_Iter L_End(Lista L){
   assert(L);
   return NULL;
}
```

In cazul avansării la următorul element, se verifică dacă nu s-a ajuns în afara listei.

```
List_Iter L_Next(Lista L, List_Iter p){
   assert(L && !L_Empty(L) && VerifPoz(L,p));
   return p->next;
}
```

Funcțiile **L_New()**, **L_Begin()**, **L_End()** şi **L_Next()** au complexitate $\Theta(1)$.

Pentru traversarea înapoi a listei se definesc funcțiile:

```
List_Iter L_RBegin(Lista L){
   assert(L);
   return L->ultim;
}

List_Iter L_REnd(Lista L){
   assert(L);
   return NULL;
}

List_Iter L_Prev(Lista L, List_Iter p){
   assert(L && !L_Empty(L) && VerifPoz(L,p));
   return p->prev;
}
```

Funcțiile care utilizează o poziție în listă trebuie să se asigure că acea poziție există într-adevăr în listă.

In acest scop vom defini o funcție de verificare a poziției, nedeclarată în interfață, deci ascunsă utilizatorului - **VerifPoz(Lista L,List_Iter p)**, care caută poziția **p** în câmpul de legătură al fiecărui element din listă..

```
int VerifPoz(Lista L, List_Iter p){
  List_Iter poz = L->prim;
  if(p==NULL) return 1;
  while(poz!=NULL){
    if(poz==p) return 1;
    poz = poz->next;
  }
  return 0;
}
```

Complexitate **VerifPoz()** este **Θ(size).**

Extragerea unui element dintr-o poziție dată, verifică în prealabil că poziția există în listă:

```
void *L_Get(Lista L, List_Iter p){
   assert(L && !L_Empty(L) && VerifPoz(L,p));
   return p->data;
}
```

Funcția de modificare a unui element dintr-o poziție dată, verifică de asemeni validitatea poziției:

```
void L_Modif(Lista L, List_Iter p, void *x){
   assert(L && !L_Empty(L) && VerifPoz(L,p));
   p->data = x;
}
```

Căutarea asociativă a unei valori presupune traversarea listei, oprită la găsirea valorii sau la epuizarea listei, situație în care rezultatul căutării este NULL.

```
List_Iter L_Find(Lista L, void *x, PFC egal){
  List_Iter p;
  for(p=L_Begin(L); p!=L_End(L); p=L_next(L,p))
    if((*egal)(L_Get(L, p), x)) return p;
  return NULL;
}
```

Funcția **L_Find()** are complexitate **Θ(size).**

Ştergerea nodului dintr-o poziție dată **p** din listă ar invalida iteratorul. Pentru a evita aceasta vom utiliza un alt iterator pentru ştergerea nodului şi vom actualiza poziția iteratorului pe următorul element. Se reface legătura acesteia, izolând nodul de şters şi se eliberează memoria ocupată de nod.

```
void *L_Remove(Lista L, List_Iter *p){
  assert(L && !L_Empty(L) && VerifPoz(L,*p));
  L_Iter q=*p;        //alt iterator pentru a sterge nodul
  *p = L_Next(L,*p);  //actualizare iterator
  x = q->data;
  q->data = NULL;
  if(q->prev)         //izolarea nodului de sters
     q->prev->next = q->next;
  else
```

```
      L->prim = q->next;
  if(q->next)
      q->next->prev = q->prev;
  else
      L->ultim = q->prev;
  free(q);
  L->dim--;
  return x;
}
```

Funcția **L_Remove()** are complexitate **Θ(size)**.

```
void L_insert(Lista L, List_Iter p, void *x) {
  List_Iter nou;
  if(!VerifPoz(L, p))
      return;    /*pozitia p nu exista*/
  nou=(List_Iter)malloc(sizeof(struct nod));
  nou->data = x;
  if(p){
      nou->next = p;
      if(p->prev){
        nou->prev = p->prev;
        p->prev->next = nou;
        p->prev = nou;
      }
      else
      {
        nou->prev = NULL;
        L->prim = nou;
      }
      p->prev = nou;
  }
  else
  {
      nou->next = NULL;
      nou->prev = L->ultim;
      if(L->ultim){
         L->ultim->next = nou;
         L->ultim = nou;
      }
      else
         L->ultim = L->prim = nou;
  }
  L->dim++;
}
```

Ştergerea unei liste o transformă într-o listă vidă (având numai antet).

```
void L_Delete(Lista *L) {
  List_Iter crt, prc;
  crt = (*L)->prim;
  while(crt != NULL){
    prc = crt;
```

```
    crt = crt->next;
    free(prc);
  }
  free(*L);
}
```

Complexitate **L_Insert()** şi **L_Delete()** este **Θ(size).**

Probleme propuse.

1. Un colier este reprezentat prin pietrele care intră în alcatuirea lui, fiecare tip de piatră fiind specificată printr-o literă. În acest fel fiecare colier poate fi reprezentat printr-un şir de caractere. Se citesc mai multe şiruri de caractere reprezentând coliere. Să se stabilească
   1) colierele contin cel mai mare numar de pietre diferite.
   2) colierele identice (egale, permutate circular egale sau permutate circular şi reflectate egale

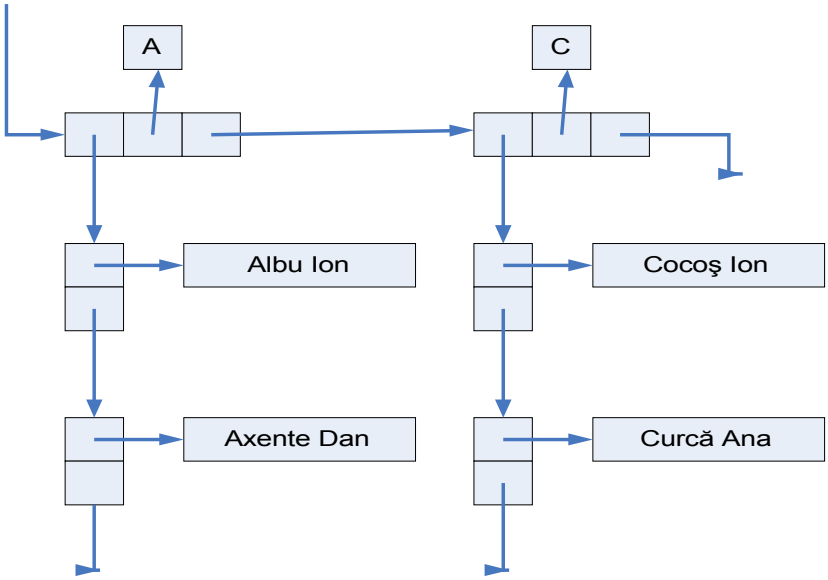
2. Un traseu de metrou din Bucuresti este de tipul dus-întors şi constă dintr-o listă de staţii specificate prin şiruri de caractere (de exemplu o listă începe cu Republica şi se termină cu Preciziei). Cele `n` trasee ale metroului bucurestean sunt citite din fluxul de intrare, staţiile fiind separate între ele prin spatii libere. Fiecare traseu este terminat printr-un sfarşit de linie. Pentru fiecare traseu se va crea o listă, având ca elemente numele staţiilor de pe traseu. În final toate traseele vor fi reprezentate printr-un vector de liste cu elemente şiruri. Se citesc de asemeni două staţii: `start` şi `stop`.
Se cere să se stabilească:
   1) dacă există trasee cere să conţină cele două staţii, şi în caz afirmativ se vor afişa:
      -traseul prin cele două staţii terminale,
      -numarul minim de staţii intermediare între `start` şi `stop` şi -numele acestor staţii.
   2) dacă putem ajunge de la staţia start la staţia stop, schimbând metroul într-o staţie intermediara, şi în caz afirmativ se vor afişa staţiile de pe cele două trasee astfel:
      -o primă linie începe cu staţia start şi se continuă cu staţiile de pe primul traseu, pană la staţia de schimb, inclusiv aceasta -a doua linie va cuprinde staţiile de pe cel de-al doilea traseu, începând cu staţia de schimb şi termînând cu staţia stop.

Indicaţie: Se crează două mulţimi de trasee care trec prin `start`, respectiv `stop`. Intersectia acestora determină staţia de tranzit.

3. Un polinom **p** este reprezentat printr-o listă de termeni sortaţi crescător după exponent. Scrieţi:
a) o funcţie care adună două polinoame, funcţie având semnătura:
```
Lista AdPol(Lista p1,Lista p2);
```
b) o funcţie care înmulţeşte două polinoame, funcţie având semnătura:
```
Lista MulPol(Lista p1, Lista p2);
```

4. Un număr lung este reprezentat ca o listă de caractere. Scrieţi:
a) o funcţie care adună două numere lungi, funcţie având semnătura:
```
Lista aduna(Lista n1, Lista n2);
```
b) o funcţie calculează derivata de ordinul **k** a unui polinom, funcţie având semnătura:
```
Lista DerPol(Lista p, int k);
```

5. O agendă are următoarea structură:



Lista de litere şi listele de nume sunt sortate.Definiți funcțiile:
`void Insert(Lista L, char *nume);` care inserează un nume în l
`void Remove(Lista *L, char *nume);` care şterge un nume din listă.