1 Laborator 2: Liste Simplu Inlantuite

1.1 Obiective

Scopul acestui laborator este de a familiariza studenţii cu operaţii cu structuri de date de tip listă. Listele sunt reprezentate prin structuri dinamice de date, adică structuri al căror număr de componente se modifică în timpul executării programului. Aspectul dinamic al structurii determină apariţia în timp a unor componente noi care trebuie legate în structura dinamică de celelalte componente. Apare astfel nevoia ca elementele structurii să fie înlănţuite, fiecare componentă cuprinzând pe lângă informaţia propriu-zisă şi o informaţie suplimentară, o informaţie de legătură prin care se realizează înlănâuirea.

În lucrare sunt prezentate operaţiile importante asupra listelor simplu înlănţuite şi particularităţile structurilor de tip stivă şi coadă.

1.2 Noţiuni teoretice

1.2.1 Definiţie listă

Lista este o mulţime finită şi ordonată de elemente de acelaşi tip. Elementele listei se numesc noduri. Listele pot fi organizate în două moduri:

1. Sub formă statică, de tablou, caz în care ordinea elementelor din listă este dată de tipul tablou unidimensional.
2. Sub formă de liste dinamice, în care ordinea nodurilor este stabilită prin pointeri. Nodurile listelor dinamice sunt alocate în memoria heap. Listele dinamice se numesc liste înlănţuite, putând fi simplu sau dublu înlănţuite.

Structura unui nod poate fi următoarea:

```
typedef struct nodetype
{
  int key; /* an optional field */
  ... /* other useful data fields */
  struct nodetype *next; /* link to next node */
} NodeT;
```

Modelul unei liste simplu înlănţuite

Modelul unei liste simplu înlănţuite este dat în Figura 1.1.

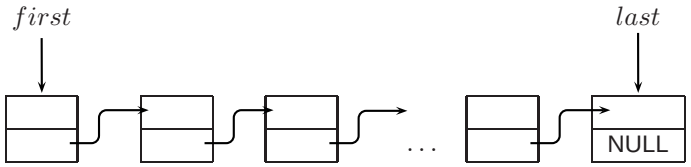

Figure 1.1: Listă simplu înlănţuită

Pointerii *first* şi *last* vor fi declaraţi astfel:

```
NodeT *fist, *last;
```

1.2.2 Operaţii cu liste simplu înlănţuite

Crearea unei liste simplu înlănţuite

Pentru crearea unei liste simplu înlănţuite se pot considera următorii paşi:

1. Iniţial lista este vidă. În cod pointerii first şi last primesc valoarea $NULL$, adică $first = NULL, last = NULL$.
2. Se alocă spaţiu pentru nodul p care va fi introdus în listă:

```
/* aloca spatiu pentru un nod nou */
p = ( NodeT * )malloc( sizeof( NodeT ));
```

3. Se plasează datele în nodul adresat de p. Implementarea depinde de tipul de dată stocat de nod. Pentru tipuri de date primitive se poate folosi o asignare de tipul: $*p = data$.
4. Se realizează legăturile necesare pentru adăugarea nodului în listă:

```
p->next = NULL;  /* nodul este adaugat in lista */
if ( last != NULL ) /* daca lista nu este vida */
  last->next = p;
else
  first = p; /* daca lista este vida p va fi primul nod */
last = p;
```

Accesul la un nod al unei liste simplu înlănţuite

Nodurile unei liste pot fi accesate *secvenţial*, extrăgând informaţia utilă. De obicei o parte din informaţia de la nivelul nodului este folosită ca o *cheie* care ajută să identificăm un nod sau să gasim o anumită informaţie. Căutarea nodului după cheie se face liniar, el putând fi prezent sau nu în listă.

O funcţie de căutare a unui nod care are cheia 'givenKey' va conţine secvenţa de program de mai jos; ea returnează adresa nodului respectiv în caz de găsire sau pointerul NULL în caz contrar:

```
NodeT *p;
p = first;
while ( p != NULL )
  if ( p->key = givenKey )
  {
    return p; /* key found at address p */
  }
  else
    p = p->next;
  return NULL; /* not found */
```

Inserarea unui nod într-o listă simplu înlănţuită

Nodul de inserat va fi creat asa cum este prezentat în paragraful §1.2.2. Presupunem că acest nod este referit de pointerul p.

- Dacă lista este vidă, acest nod va fi singur în listă:

```
if ( first == NULL )
{
  first = p;
  last = p;
  p->next = NULL;
}
```

- Dacă lista nu este vidă, modul în care se face inserarea depinde de poziţia unde va fi inserat nodul. Astfel se pot identifica următoarele cazuri:
  1. Inserare înaintea primului nod:

```
if ( first != NULL )
{
  p->next = first;
  first = p;
}
```

  2. Inserare după ultimul nod:

```
if ( last != NULL )
{
  p->next = NULL;
  last->next = p;
  last = p;
}
```

  3. Inserare înaintea unui nod dat de o cheie 'key'. În acest caz se execută doi paşi:
     a) Se caută nodul care are cheia *givenKey*:

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
while ( q != NULL )
{
```

```
  if ( q->key == givenKey ) break;
  q1 = q;
  q = q->next;
}
```

b) Se inserează nodul de pointer *p*, făcând legăturile corespunzătoare:

```
if ( q != NULL )
{
 /* node with key givenKey has address q */
  if ( q == first )
  { /* insert after first node */
    p->next = first;
    first = p;
  }
  else
  {
    q1->next = p;
    p->next = q;
  }
}
```

4. Inserarea după un nod care are cheia *key*. Şi în acest caz se vor executa doi paşi:
   a) Se caută nodul care conţine *givenKey*:

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
while ( q != NULL )
{
  if ( q->key == givenKey ) break;
  q1 = q;
  q = q->next;
}
```

   b) Se inserează nodul de pointer *p*, şi se ajustează legăturile:

```
if ( q != NULL )
{
  p->next = q->next; /* node with key givenKey has address q */
  q->next = p;
  if ( q == last ) last = p;
}
```

Ştergerea unui nod dintr-o listă simplu înlănţuită

La ştergerea unui nod se vor avea în vedere următoarele probleme: lista poate fi vidă, lista poate conţine un singur nod sau lista poate conţine mai multe noduri. De asemenea se poate cere ştergerea primului nod, a ultimului nod sau a unui nod dat printr-o anumită cheie.

Astfel pot exista următoarele cazuri:

1. Stergerea primului nod dintr-o listă:

```
NodeT *p;
if ( first != NULL )
{ /* non-empty list */
  p = first;
  first = first->next;
  free( p ); /* free up memory */
  if ( first == NULL ) /* list is now empty */
    last = NULL;
}
```

2. Ştergerea ultimului nod dintr-o listă:

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
if ( q != NULL )
{  /* non-empty list */
  while ( q != last )
  {  /* advance towards end */
    q1 = q;
    q = q->next;
```

```
    }
    if ( q == first )
    { /* only one node */
      first = last = NULL;
    }
    else
    { /* more than one node */
      q1->next = NULL;
      last = q1;
    }
    free( q );
  }
```

3. Ştergerea unui nod care are cheia *givenKey*

```
NodeT *q, *q1;
q1 = NULL; /* initialize */
q = first;
/* search node */
while ( q != NULL )
{
  if ( q->key == givenKey ) break;
  q1 = q;
  q = q->next;
}
if ( q != NULL )
{   /* found a node with supplied key */
  if ( q == first )
  { /* is the first node */
    first = first->next;
    free( q ); /* release memory */
    if ( first == NULL ) last = NULL;
  }
  else
  { /* other than first node */
    q1->next = q->next;
    if ( q == last ) last = q1;
    free( q ); /* release memory */
  }
}
```

Ştergerea unei liste simplu înlănţuite

Pentru o ştergere completă a unei liste, va fi şters fiecare nod ca în exemplul de mai jos:

```
NodeT *p;
while ( first != NULL )
{
  p = first;
  first = first->next;
  free( p );
}
last = NULL;
```

1.2.3 Stive

Stiva este o listă simplu inlantuita specială în care adăugarea sau scoaterea unui element se face la un singur capăt al listei, numit **vârful stivei**. Elementul introdus primul în stivă poartă numele de **baza stivei**.

Stiva se poate asemăna unui vraf de farfurii aşezat pe o masă, modalitatea cea mai comodă de a pune o farfurie fiind in vârful stivei, şi tot de aici e cel mai simplu să se ia o farfurie.

Datorită locului unde se acţionează asupra stivei, aceste structuri se mai numesc structuri de tip **LIFO (Last In First Out)**, adică 'ultimul sosit - primul ieşit'. Modelul unei **stive** este dat de Figura 1.2.

Principalele operaţii pe o stivă sunt următoarele:

push – adaugarea unui element in varful stivei;
pop – stergerea unui element din varful stivei;
delete – stergerea intregii stive.

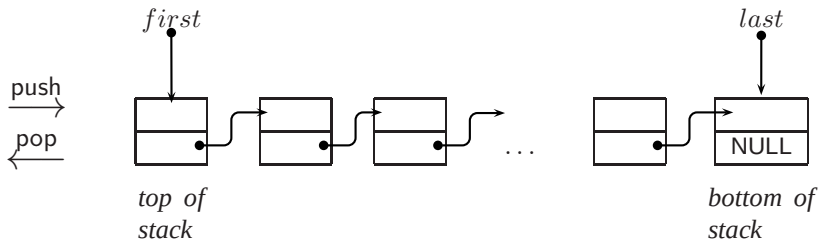

Figure 1.2: Modelul unei stive.

1.2.4 Cozi

Coada reprezintă o altă categorie specială de listă simplu înlănţuită, în care elementele se adaugă la un capăt (sfărşit) şi se suprimă la celălalt capăt (început).

Coada este o listă simplu înlănţuită care funcţionează conform algoritmului FIFO (First In First Out), adică 'primul venit - primul servit'.

Modelul intuitiv al acestei structuri este coada care se formează la un magazin: lumea se aşează la coadă la sfârşitul ei, cei care se găsesc la începutul cozii sunt serviţi şi părăsesc apoi coada.

Modelul cozii care va fi avut în vedere în consideraţiile următoare este dat în Figura 1.3. Operaţiile principale sunt:

enqueue – introducerea unui element în coadă – funcţia se realizează prin inserarea după ultimul nod;
dequeue – scoaterea unui element din coadă – funcţia se realizează prin ştergerea primului nod;
delete – ştergerea cozii – funcţia se realizează conform paragrafului §1.2.2.

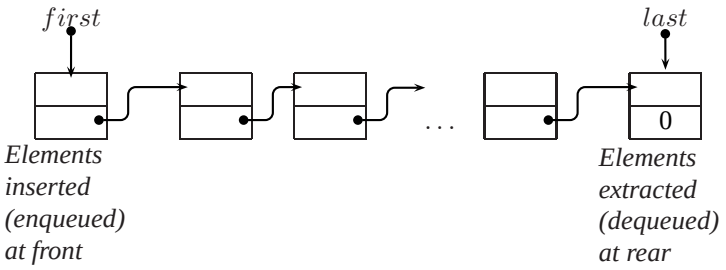

Figure 1.3: A queue model.

1.3 Mersul lucrarii

Studiaţi codul prezentat în laborator şi utilizaţi acest cod pentru rezolvarea problemelor obligatorii.

Pentru problemele propuse datele de intrare şi datele de ieşire sunt în fişiere care sunt date ca argumente de la linia de comandă.

1.3.1 Probleme Obligatorii

1.1. Se dă un garaj pentru camioane. Drumul de access al garajului poate să conţină oricâte camioane. Garajul are o singură uşă astfel încât doar ultimul camion care a intrat pote să iasă primul (conform modelului stivă). Fiecare camion este identificat de un număr întreg pozitiv (truck_id). Scrieţi un program care să trateze diferite tipuri de mutări ale camioanelor, şi să permită următoarele comenzi:

a) `Pe_drum` (truck_id);
b) `Intră_in_garaj` (truck_ id);
c) `Iese_din_garaj` (truck_id);
d) `Afişează_camioane` (`garaj` sau `drum`);

Dacă se doreşte scoaterea unui camion care nu este cel mai aproape de intrarea garajului se va afişa un mesaj de eroare `Camionul` x `nu este la uşa garajului`.

1.2. Se consideră problema anterioară a camioanelor dintr-un garaj, dar de această dată garajul are două uşi legate printr-un drum circular. O uşă este folosită doar pentru intrarea camioanelor în garaj, iar altă uşă este folosită pentru ieşirea din garaj. Camioanele pot ieşi din garaj doar în ordinea în care au intrat (conform modelului coadă).

Date de intrare:

```
Pe_drum(2)
Pe_drum(5)
Pe_drum(10)
Pe_drum(9)
Pe_drum(22)
Afişează_camioane(drum)
Afişează_camioane(garaj)
Intra_in_garaj(2)
Afişează_camioane(drum)
Afişează_camioane(garaj)
Intra_in_garaj(10)
Intra_in_garaj(25)
Iese_din_garaj(10)
Iese_din_garaj(2)
Afişează_camioane(drum)
Afişează_camioane(garaj)
```

Date de iesire:

```
drum:␣2␣5␣10␣9␣22
garaj:␣nimic
drum:␣5␣10␣9␣22
garaj:␣2
error:␣25␣nu␣este␣pe␣drum!
error:␣10␣nu␣este␣la␣ieşire!
drum:␣2␣5␣9␣22
garaj:␣10
```

1.3.2 Probleme Opţionale

1.3. Să se definească şi să se implementeze funcţiile pentru structura de date definită după cum urmează:

```
typedef struct node
{
  struct node *next;
  void *data;
} NodeT;
```

folosind modelul dat în Figura 1.4. Celulele de date conţin o cheie numerică şi un cuvânt – string, de exemplu numele unui student şi numărul carnetului de student.

.

I/O description. Operaţiile se vor codifica în modul următor: `cre`= crează listă vidă, `del` *key*= şterge nod care are cheia *key* din listă, `dst`=şterge primul nod (nu santinela), `dla`=şterge ultimul nod (nu santinela), `ins` *data*= inserează un element în ordinea crescătoare a cheilor, `ist` *data*= inserează un nod având datele *data* ca prim nod în listă, `ila` *data*= inserează un nod având datele *data*ca ultim nod în listă, `prt`= afişează lista.

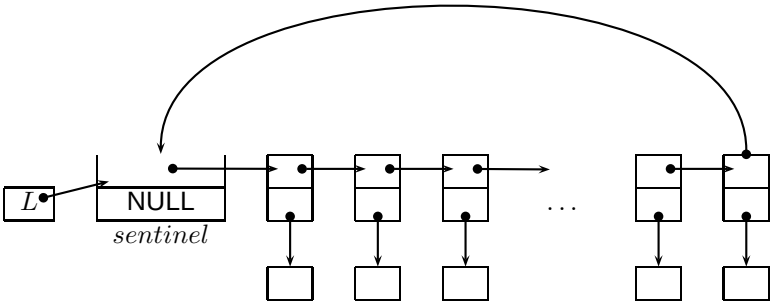

Figure 1.4: Model de listă pentru problema 1.3.

1.4. Sortarea topologică. Elementele unei mulţimi M sunt notate cu litere mici din alfabet. Se citesc perechi de elemente x, y (x, y aparţin mulţimii M) cu semnificaţia că elementul x precede elementul y. Să se afişeze elementele mulţimii M într-o anumită ordine, încât pentru orice elemente x, y cu proprietatea că x precede pe y, elementul x să fie afişat înaintea lui y.

1.5. Să se scrie programul care creează două liste ordonate crescător după o cheie numerica şi apoi le interclasează. .
I/O description. Intrare:

```
i1␣23␣47␣52␣30␣2␣5␣-2
i2␣-5␣-11␣33␣7␣90
p1
p2
m
p1
p2
```

Ieşire:

```
1:␣-2␣2␣5␣23␣30␣47␣52
2:␣-11␣-5␣7␣33␣90
1:␣-11␣5␣2␣2␣5␣7␣23␣...
2:␣vidă
```

Astfel, "comenzile" acceptate sunt: `i`n=inserează în lista $n \in \{1, 2\}$, `p`n=afişează lista n, `m`=interclasează listele.

1.6. Operaţii cu matrici rare: să se conceapă o stuctură dinamică eficientă pentru reprezentarea matricelor rare. Să se scrie operaţii de calcul a sumei şi produsului a două matrice rare. Afisarea se va face in forma naturală. .
I/O description. Intrare:

```
m1␣40␣40
(3,␣3,␣30)
(25,␣15,␣2)
m2␣40␣20
(5,␣12␣1)
(7␣14␣22)
m1+m2
m1*m2
```

, unde `m1`=citeşte elementele matricii `m1`, şi tripletele următoare sunt $(row, col, value)$ pentru matrice. Citirea se termină atunci când e dată o altă comandă sau se întâlneşte sfârşitul de fişier. **E.g.** `m1+m2`=adună matricea 1 la matricea 2, and `m1*m2`= înmulţeşte matricea `m1` cu matricea `m2`. Afişarea rezultatelor se va face tot sub forma de triplete.

1.7. Operaţii cu polinoame: să se conceapă o structură dinamică eficientă pentru reprezentarea în memorie a polinoamelor. Se vor scrie funcţii de calcul a sumei, diferenţei şi produsului a două polinoame. .
I/O description. Intrare:

```
p1=3x^7+5x^6+22.5x^5+0.35x-2
p2=0.25x^3+.33x^2-.01
p1+p2
p1-p2
p1*p2
```

, Ieşire:

```
<Afişează␣suma␣polinoamelor>
<Afişează␣diferenţa␣polinoamelor>
<Afişează␣produsul␣polinoamelor>
```