2 Liste Simplu Inlantuite. Vectori

2.1 Obiective

Scopul acestei sesiuni de laborator este de a ne familiariza cu implementarea operatiilor fundamentale pe tipurile de data abstracta lista. Se vor prezenta si discuta implementari ale operatiilor pe liste simplu inlantuite si pe vectori.

2.2 Notiuni teoretice

*Intrebare: Care este diferenta intre un **tip de data abstracta** si o **structura de date**?*

2.2.1 Definitie lista

Tipul de data abstracta **lista** este o multime finita si ordonata (nu neaparat sortata, dar elementele apar unul dupa celalalt, intr-o anumita ordine) de elemente de acelasi tip; poate contine duplicate.

Intrebare: Care sunt operatiile pe care le definim de regula pe tipul de data abstracta lista? (pentru verificarea raspunsului, vezi notitele de curs)

Colectiile de obiecte, prin urmare si listele, pot fi implementate in doua moduri diferite:

1. Utilizand *alocare secventiala*, lista fiind de fapt un tablou unidimensional (*en. array*).
2. Utilizand *alocare inlantuita* sau *dinamica*, in care ordinea nodurilor este stabilita prin referinte, stocate la nivelul fiecarui element (*nod*). Nodurile listelor dinamice sunt alocate in memoria heap. Listele dinamice se numesc liste inlantuite (*en. linked lists*), putand fi simplu sau dublu inlantuite.

Ambele reprezentari prezinta avantaje si dezavantaje: in cazul *alocarii secventiale* avem acces imediat la orice element al colectiei (accesul pe baza de index), dar dimensiunea alocata (capacitatea) este fixa; prin urmare, fie nu utilizam spatiul eficient, fie vom avea nevoie sa alocam un tablou de dimensiune mai mare si sa copiem continutul colectiei, element cu element (pentru cazul in care dimensiunea colectiei ajunge sa depaseasca capacitatea initiala). *Alocarea inlantuita* are avantajul de a utiliza spatiu proportional cu dimensiunea colectiei, insa accesul la un element se face pe baza unei referinte (i.e. nu e direct).

2.2.2 Implementarea unei liste utilizand alocarea înlantuita

Modelul unei liste simplu inlantuite este data in Figura 2.1.

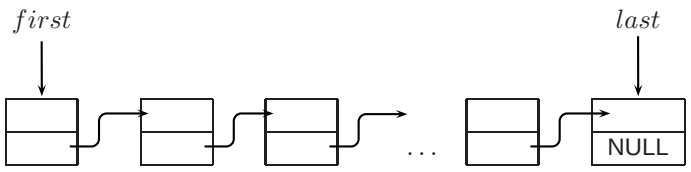

Figure 2.1: Lista simplu înlantuita

Structura de baza a unui nod este urmatoarea:

```
typedef struct nodetype
{
  int key; /* key field */
  ... /* other data fields */
  struct nodetype *next; /* reference/pointer to next node */
} NodeT;
```

Lista se identifica cel putin prin referinta spre primul element ($first$). Pentru eficienta, de regula listele simplu inlantuite ofera si referinta catre ultimul element ($last$, in cazul operatiilor de adaugare/stergere la finalul listei). Pointerii *first* si *last* vor fi declarati astfel:

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

Ideal ar fi ca acestia sa fie declarati local, in functia *main* a aplicatiei, si transmisi ca parametru diferitelor functii (prin referinta, pentru cazul in care trebuie modificati). O versiune mai simplu de implementat, dar mai putin flexibila, este declararea globala a acestora. Totodata, odata cu declararea lor acestia ar trebui initializati la *NULL* (daca sunt declarati global sunt initializati implicit la declarare):

```
first=NULL; last=NULL;
```

Exercitiu: Creati - in mediul de lucru ales - un proiect nou de tip aplicatie consola, si definiti o structura de nod de lista inlantuita care sa contina urmatoarele campuri: key - de tip intreg si referinta catre urmatorul nod din structura. Definiti pointerii catre inceputul si sfarsitul unei liste ce contine astfel de noduri.

Cautarea intr-o lista simplu înlantuita

Nodurile unei liste simplu inlantuite pot fi accesate *secvential*, extragând informatia utila. De obicei o parte din informatia de la nivelul nodului este folosita ca si *cheie* care ajuta sa identificam un nod sau sa gasim o anumita informatie. Cautarea dupa cheie se face liniar, parcurgand lista nod cu nod. Mai jos prezentam secventa de parcurgere a unei liste, in cautarea nodului din lista care are cheia $givenKey$:

```
/* p - current node, used to traverse the list */
NodeT* p = first;
while ( p != NULL ) /* while not reached the end of the list */
  if ( p->key == givenKey ) /* found node having givenKey*/
  {
      ...  /* key found at address p */
  }
  else
     p = p->next;
```

Ex. 1 — Implementati functia **NodeT* search(NodeT* first, int givenKey)** care cauta in lista $first$ nodul care are cheia $givenKey$. Functia returneaza adresa nodului, respectiv NULL daca acesta nu exista.

Inserarea unui nod intr-o lista simplu inlantuita

Orice operatie de inserare a unui element intr-o lista simplu inlantuita primeste de regula informatia de inserat, creeaza un nod nou cu aceasta informatie, si il introduce in lista la pozitia corespunzatoare, in functie de strategia de inserare (la inceput, la final, inainte/dupa o anumita cheie, la o anumita pozitie, in ordine, etc).

Codul corespunzator partii de creare a nodului este:

```
NodeT *p = ( NodeT * )malloc( sizeof( NodeT )); /* allocate memory */
p->key   = givenKey; /* copy key (assume type integer) in node pointed to by p*/
p->next  = NULL; /* initialize pointer to next element to NULL*/
... /* copy the rest of the data in the node pointed to by p*/
```

Asadar, nodul nou creat este referit de pointerul p. In continuare vom prezenta diferite fragmente de cod, in functie de strategiile diferite de introducere a acestuia in lista, respectiv tratarea cazului in care lista este vida (primul punct din paragrafele urmatoare).

- Daca lista este vida, acest nod va fi singur în lista:

```
if ( *first == NULL )
{
  *first        = p;
  *last         = p;
}
```

- Daca lista nu este vida, modul în care se face inserarea depinde de pozitia unde va fi inserat nodul (data de strategia specifica de inserare). Astfel se pot identifica urmatoarele cazuri:

  1. Inserare înaintea primului nod:

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

  2. Inserare dupa ultimul nod:

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

3. Inserare înaintea unui nod dat de o cheie $beforeKey$. In acest caz se executa doi pasi:
   a) Se cauta nodul care are cheia *beforeKey*:

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

   b) Se introduce nodul a carui adresa o avem in p in lista, realizand legaturile corespunzatoare:

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

4. Inserarea dupa un nod care are cheia *afterKey*. Si în acest caz se vor executa doi pasi:
   a) Se cauta nodul care contine *afterKey*:

```
NodeT *q = *first;
while ( q != NULL )
{
  if ( q->key == afterKey )
        break;
  q = q->next;
}
```

   b) Se introduce nodul p in lista, ajustand legaturile:

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

Ex. 2 — Implementati functiile **void insert_first(NodeT** first, NodeT** last, int givenKey)** ,**void insert_last(NodeT** first, NodeT** last, int givenKey)**, respectiv **void insert_after_key(NodeT** first, NodeT** last, int afterKey, int givenKey)** care insereaza un nod nou intr-o lista simplu inlantuita, utilizand strategiile sugerate de numele functiilor.

Ex. 3 — Completati programul scris pana acum astfel: inserati ca prim element, pe rand, cheile 4 si 1; inserati cheia 3 ca ultim element; cautati cheia 2; cautati cheia 3; inserati cheia 22 dupa cheia 4; inserati cheia 25 dupa cheia 3; afisati pe ecran continutul listei - cheile din lista;

Stergerea unui nod dintr-o lista simplu inlantuita

Operatia de stergere poate prezenta si ea mai multe strategii de stergere: stergerea primului nod, a ultimului nod sau a unui nod dat printr-o anumita cheie. Se vor avea în vedere urmatoarele probleme: lista poate fi vida, lista poate contine un singur nod sau lista poate contine mai multe noduri.

1. Stergerea primului nod dintr-o lista:

```
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. Stergerea ultimului nod dintr-o lista:

```
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. Stergerea 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 */
  }
}
```

Pentru o stergere complet a a unei liste, se va sterge primul element in mod repetat, pana cand lista ramane goala.

Ex. 4 — Implementati functiile **void delete_first(NodeT** first, NodeT** last)**, **void delete_last(NodeT** first, NodeT** last)**, **void delete_key(NodeT** first, NodeT** last, int givenKey)** care sterg nodul corespunzator din lista simplu inlantuita data de $first$ si $last$ (primul, ultimul, respectiv nodul avand cheia $givenKey$).

Ex. 5 — Completati programul scris pana acum astfel: stergeti primul nod, stergeti ultimul nod, stergeti nodul avand

cheia 22, stergeti nodul avand cheia 8; afisati pe ecran continutul listei - cheile din lista; stergeti apoi toata lista; afisati continutul listei.

2.2.3 Implementarea unei liste utilizand alocarea secventiala

Cealalta alternativa pentru a implementa operatiile tipului de data abstracta lista este utilizarea alocarii secventiale - tablou unidimensional ($vector$). Vom avea nevoie, evident, sa alocam un tablou de o anumita dimensiune maxima ($capacity$), dar si sa cunoastem numarul de elemente care se gasesc la fiecare moment in structura noastra ($size$).

Modelul unei liste implementate ca si vector este dat in Figura 2.2.

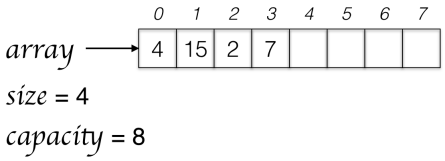

Figure 2.2: Lista implementata ca si vector

Prin urmare, pentru a implementa lista si operatiile dorite, vom avea nevoie de un vector de o anumita dimensiune (capacitate), si de doi intregi: $capacity$ si $size$.

Cautarea unei anumite chei intr-un vector se face liniar, parcurgand vectorul pana la $size - 1$ si verificand, element cu element.

Pentru a insera elementul $elem$ la inceputul vectorului (i.e. operatia $insert\_first$), toate elementele vectorului se muta cu o pozitie spre dreapta, dimensiunea ($size$) va creste cu 1, si noul element va fi plasat pe pozitia 0 a vectorului:

```
int i;
for(i=size-1; i>=0; i--)
        arr[i+1]=arr[i]; //shift all elements one position to the right, to make room
size++; // increase size
arr[0]=elem; // place element in the first array cell
```

Evident, trebuie verificat in prealabil ca nu s-a atins capacitatea vectorului (i.e. $size < capacity$). In caz contrar, va trebui intai sa re-alocam un vector de capacitate mai mare (dublam capacitatea de fiecare data cand vectorul este plin si avem nevoie sa inseram un element nou), si sa copiem in el toate elementele existente in cel curent. Celelalte operatii de inserare ($insert\_last$, $insert\_before\_key$, $insert\_after\_key$) se trateaza similar.

Pentru operatiile de stergere, tratarea celulelor care raman libere se poate face in doua moduri: fie sa "compactam" vectorul dupa fiecare stergere (prin mutarea tuturor elementelor de dupa cel sters inspre stanga cu o pozitie - similar cu inserarea), fie sa marcam celula stearsa cu o valoare speciala, si compactarea sa se realizeze la anumite momente, sincron.

Ex. 6 — Implementati urmatoarele operatii pe un vector de intregi: **int search(int* arr, int size, int key)**, **void insert_first(int* arr, int* size, int key)**, **void insert_last(int* arr, int* size, int key)**, **void delete_first(int* arr, int* size)**, **void delete_last(int* arr, int* size)**, **void delete_key(int* arr, int* size, int key)**, **void print(int* arr, int size)**.
Testati functiile scrise, dupa cum urmeaza: alocati un vector de capacitate 10; inserati, pe rand, pe prima pozitie, cheile 5, 2 si 7. Inserati, pe ultima pozitie, cheile 12 si 13; afisati continutul vectorului; afisati adresa(indexul) cheii 2; afisati adresa cheii 20; stergeti elementul de pe prima pozitie; stergeti elementul cu cheia 12; afisati continutul vectorului (cheile).

2.3 Mersul lucrarii

Studiati codul prezentat în laborator si utilizati acest cod pentru rezolvarea exercitiilor obligatorii, prezentate pe parcursul lucrarii. La finalul sesiunii de laborator, este obligatoriu ca fiecare student sa prezinte codul (compilabil, executabil) cerut in exercitiile de pe parcursul lucrarii de laborator.

2.3.1 Probleme Optionale

1. Sa se implementeze o functie care inverseaza o lista simplu inlantuita.

2. Sa se implementeze o functie care insereaza un element intr-o lista simplu inlantuita ordonata.

3. Sa se implementeze o functie care gaseste elementul de la pozitia $length - k$ dintr-o lista simplu inlantuita (k dat), parcurgand lista o singura data (si utilizand o cantitate constanta de memorie aditionala).

4. O *listă simplu înlănţuită circulară* este lista simplu înlănţuită al cărei ultim element este legat de primul element, (ultimul element are ca urmator element primul element). Prin urmare, vom folosi un singur pointer $pNode$ pentru a indica un element din listă – "primul" element. Figura 2.3 arată modelul unei astfel de liste. Se cere sa implementati o listă simplu înlănţuită circulară, avand urmatoarele operatii: **NodeTC* search(NodeTC* pNode, int key)**, **void insert_first(NodeTC** pNode, int key)**, **void insert_after_key(NodeTC** pNode, int afterKey, int key)**, **void delete_key(NodeTC** pNode, int key)**, **void print(NodeTC* pNode)**.

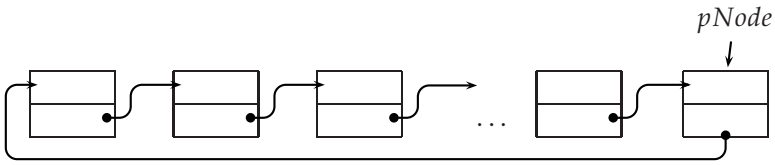

Figure 2.3: Modelul unei liste circulare simplu înlănţuite

2.3.2 Probleme Aplicative

1. Sa se scrie programul care creeaza doua liste ordonate crescator dupa o cheie numerica si apoi le interclaseaza. .
I/O description. Intrare:

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

Iesire:

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

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

2. Se dau doua fisiere text F1.txt si F2.txt care contin elemente numere intregi ordonate crescator. Fiecare numar apare o singura data in fisier. Cititi datele din cele doua fisiere si stocati informatia in cate o lista simplu inlantuita. Scrieti o functie care returneaza o lista ce reprezinta intersectia celor doua liste si afisati lista returnata. Scrieti o functie care returnreaza o lista ce reprezinta reuniunea celor doua liste si afisati-o.

3. Se da o lista liniara simplu inlantuita care stocheaza informatii despre studentii unei grupe (numele si o nota) vezi Figura 2.4. Se mai da un pointer la unul din studentii grupei a carui nota este sub 5. Se cere eliminarea studentului cu nota mai mica de 5 din grupa de studenti (folosind doar pointerul catre acel student – nu se va traversa intreaga lista).

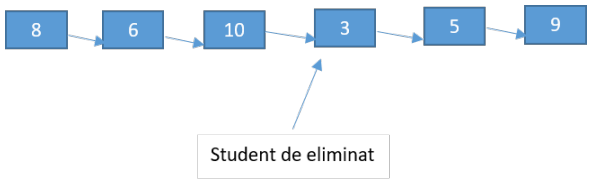

Figure 2.4: Date problema

4. Operatii cu matrici rare: sa se conceapa o stuctura dinamica eficienta pentru reprezentarea matricelor rare. Sa se scrie operatii de calcul a sumei si produsului a doua matrice rare. Afisarea se va face in forma naturala. .
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`=citeste elementele matricii `m1`, si tripletele urmatoare sunt $(row, col, value)$ pentru matrice. Citirea se termina atunci când e data o alta comanda sau se întâlneste sfârsitul de fisier. **E.g.** `m1+m2`=aduna matricea 1 la matricea 2, and `m1*m2`= înmulteste matricea `m1` cu matricea `m2`. Afisarea rezultatelor se va face tot sub forma de triplete.

5. Operatii cu polinoame: sa se conceapa o structura dinamica eficienta pentru reprezentarea în memorie a polinoamelor. Se vor scrie functii de calcul a sumei, diferentei si produsului a doua 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
```

Iesire:

```
<Afiseaza␣suma␣polinoamelor>
<Afiseaza␣diferenta␣polinoamelor>
<Afiseaza␣produsul␣polinoamelor>
```

6. Sa se defineasca si sa se implementeze functiile pentru structura de date definita dupa cum urmeaza:

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

folosind modelul dat în Figura 2.5. Celulele de date contin o cheie numerica si un cuvânt – string, de exemplu numele unui student si numarul carnetului de student.

.
I/O description. Operatiile se vor codifica în modul urmator: `cre`= creaza lista vida, `del` key= sterge nod care are cheia key din lista, `dst`=sterge primul nod (nu santinela), `dla`=sterge ultimul nod (nu santinela), `ins` $data$= insereaza un element în ordinea crescatoare a cheilor, `ist` $data$= insereaza un nod având datele $data$ ca prim nod în lista, `ila` $data$= insereaza un nod având datele $data$ca ultim nod în lista, `prt`= afiseaza lista.

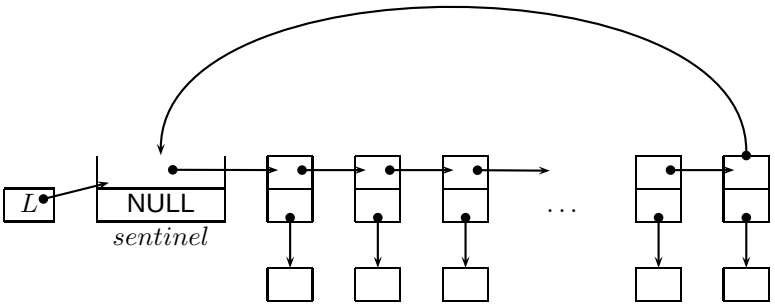

Figure 2.5: Model de lista pentru problema 2.6.

7. *Scrieti un program pentru a inversa o lista simplu inlantuita in mod eficient (timp 0(n) si spatiu aditional O(1)). Lista este implementata folosind pointer la first.
Pentru lista:
1->2->3->4->NULL

Se obtine lista inversa:
4->3->2->1->NULL

Pentru lista:
7->2->9->4->5->NULL

Se obtine lista inversa:
5->4->9->2->7->NULL