Laborator 2: Liste Simplu Înlănțuite, Vectori

1 Obiective

Scopul acestei sesiuni de laborator este de a ne familiariza cu implementarea operațiilor fundamentale pe tipurile de data abstracta listă. Se vor prezenta și discuta implementări ale operațiilor pe liste simplu înlănțuite și pe vectori.

2 Noțiuni teoretice

*Întrebare: Care este diferența între un **tip de dată abstractă** și o **structură de date**?*

2.1 Definiție listă

Tipul de dată abstractă **lista** este o mulțime finită și ordonata (nu neapărat sortată, dar elementele apar unul după celalalt, într-o anumită ordine) de elemente de același tip; poate conține duplicate.

Întrebare: Care sunt operațiile pe care le definim de regulă pe tipul de dată abstractă listă? (pentru verificarea răspunsului, vezi notițele de curs)

Colecțiile de obiecte, prin urmare și listele, pot fi implementate în două moduri diferite:

1. Utilizând *alocare secvențială*, lista fiind de fapt un tablou unidimensional (*en. array*).
2. Utilizând *alocare înlănțuită* sau *dinamică*, în care ordinea nodurilor este stabilită prin referințe, stocate la nivelul fiecărui element (*nod*). Nodurile listelor dinamice sunt alocate în memoria heap. Listele dinamice se numesc liste înlănțuite (*en. linked lists*), putând fi simplu sau dublu înlănțuite.

Ambele reprezentări prezintă avantaje și dezavantaje: în cazul *alocării secvențiale* avem acces imediat la orice element al colecției (accesul pe baza de index), dar dimensiunea alocata (capacitatea) este fixă; prin urmare, fie nu utilizăm spațiul eficient, fie vom avea nevoie să alocăm un tablou de dimensiune mai mare și să copiem conținutul colecției, element cu element (pentru cazul în care dimensiunea colecției ajunge să depășească capacitatea inițială). *Alocarea înlănțuită* are avantajul de a utiliza spațiu proporțional cu dimensiunea colecției, însă accesul la un element se face pe baza unei referințe (i.e. nu e direct).

2.2 Reguli generale

Înainte de a discuta cum creăm liste alocate dinamic, poate ar fi bine să discutăm reguli generale de lucru cu memoria dinamică.

1. Fiecare variabilă se declară pe un rând nou și se inițializează. Când citești mult cod ajută mult să poți identifica foarte rapid tipul. **Pointerii se inițializează cu NULL**.
2. Pointerii se verifică dacă sunt sau nu NULL. Dacă nu sunt NULL putem să accesăm conținutul.
3. După ce eliberăm memoria (free) pointer-ul se pune pe NULL.

Dacă urmăm regulile de mai sus, stim că dacă un pointer nu este NULL acesta are o valoare validă. Astfel evităm să:

- Accesăm un pointer neinițializat (valoare random ≠ NULL).
- Erori de tipul "use after free".
- Erori de tipul "double free".

Discuție

Cum satisfacem regula item 1 dacă alocăm dinamic memoria. Noua memorie obținută ar trebui și ea inițializată. De obicei se inițializează memoria cu 0. Valoarea 0 are ca efect:

- Pointerii or să fie inițializați pe NULL ✓
- String-urile or să fie inițializate la string vid ✓
- 0 este de cele mai multe ori neutru și pentru câmpuri numerice ✓

Pentru a obține acest efect putem folosi doua abordări:

- Folosim funcția **calloc**. Alocă memoria dorită și o inițializează la 0.
- Folosim funcțiile **malloc** și **memset**. Memoria o să fie alocată de malloc și inițializarea se face de memset.

Ce variantă ar trebui să alegem?
Răspuns scurt: Se recomandă **malloc** și **memset**.
Pentru a înțelege de ce avem acea recomandare putem să comparăm două exemple:

1.
```c
cal_ptr = calloc(10000000000000, sizeof(char));  /* Aprox 9 TB de RAM*/
```

2.
```c
mal_ptr = calloc(10000000000000 * sizeof(char));   /* Aprox 9 TB de RAM*/
if (mal_ptr != NULL) { /* Facem verificarea ca sa respectam regula 2. Nu accesam
    pana nu stim ca nu ii NULL*/
    memset(mal_ptr, 0, 10000000000000 * sizeof(char));
}
```

La prima impresie am putea zice că primul exemplu funcționează corect. Primim un pointer valid și totul merge bine. Problema apare dacă chiar încercăm să accesam fiecare poziție din memoria primită. Inevitabil o să crape programul deoarece sistemul nu are 9 TB RAM. Pe când apare problema putem fi la mii de linii de cod față de sursa problemei.

Soluția a doua o să crape cel mai probabil la **memset**. Realizăm rapid că am pus prea multe 0 la alocare și rezolvam rapid problema, fără ore întregi de debug.

2.3 Implementarea unei liste utilizând alocarea înlănțuită

Modelul unei liste simplu înlănțuite este dată în Figura 1.

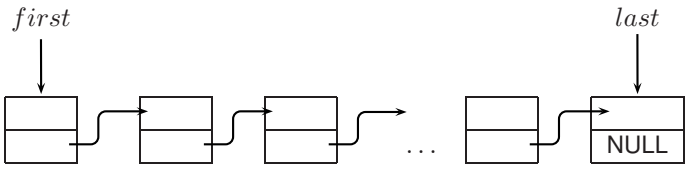

Figura 1: Lista simplu înlănțuită

Structura de bază a unui nod este următoarea:

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

Lista se identifica cel puțin prin referință spre primul element (*first*). Pentru eficiență, de regulă listele simplu înlănțuite oferă și referință către ultimul element (*last*, în cazul operațiilor de adăugare/ștergere la finalul listei). Astfel structura de bază a unei liste poate să fie următoarea:

```
typedef struct
{
  SLL_NODE *first; /* reference/pointer to the first node */
  SLL_NODE *last;  /* reference/pointer to the last node */
  ... /* other data fields */
} SL_LIST;
```

Exercițiu: Creați - în mediul de lucru ales - un proiect nou de tip aplicație consola, și definiți o structură de nod de listă înlănțuită care să conțină următoarele câmpuri: key - de tip întreg și referința către următorul nod din structură. Definiți structura listă care să conțină pointerii către începutul și sfârșitul unei liste ce conține astfel de noduri.

2.3.1 Căutarea într-o listă simplu înlănțuită

Nodurile unei liste simplu înlănțuite pot fi accesate *secvențial*, extrăgând informația utilă. De obicei o parte din informația de la nivelul nodului este folosita ca și *cheie* care ajută să identificăm un nod sau să găsim o anumită informație. Căutarea după cheie se face liniar, parcurgând lista nod cu nod. Mai jos prezentam secventa de parcurgere a unei liste, în căutarea nodului din lista care are cheia *givenKey*:

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

Ex. 1 — Implementați funcția:

```
SLL_NODE *sll_search(const SL_LIST *list, int givenKey)
```

care caută în listă nodul care are cheia *givenKey*. Funcția returnează adresa nodului, respectiv NULL dacă acesta nu există.

2.3.2 Inserarea unui nod într-o listă simplu înlănțuită

Orice operație de inserare a unui element într-o listă simplu înlănțuită primește de regula informația de inserat, creează un nod nou cu această informație, și îl introduce în listă la poziția corespunzătoare, în funcție de strategia de inserare (la început, la final, înainte/după o anumita cheie, la o anumită poziție, în ordine, etc).

Codul corespunzător parții de creare a listei este:

```
SL_LIST *list = NULL;

list = (SL_LIST *)malloc(sizeof(SL_LIST));  /* allocate memory */
if (list != NULL) {
    memset(list, 0, sizeof(SL_LIST));  /* initialize all the fields by 0 */
    ... /* copy the rest of the data in the list pointed to by list */
}
```

Codul corespunzător parții de creare a nodului este:

```
SLL_NODE *node = NULL;

node = (SLL_NODE *)malloc(sizeof(SLL_NODE)); /* allocate memory */
if (node != NULL) {
    memset(node, 0, sizeof(SLL_NODE));  /* initialize all the fields by 0 */
    node->key = value;  /* copy key (assume type integer) in node pointed to by node */
    /* next pointer is initialized by memset */
    ... /* copy the rest of the data in the node pointed to by node */
}
```

Așadar, nodul nou creat este referit de pointerul *node*. În continuare vom prezenta diferite fragmente de cod, în funcție de strategiile diferite de introducere a acestuia în listă, respectiv tratarea cazului în care lista este vidă (primul punct din paragrafele următoare).

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

```
if ( list->first == NULL )
{
  list->first    = node;
  list->last     = node;
}
```

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

  1. Inserare înaintea primului nod:

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

  2. Inserare după ultimul nod:

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

  3. Inserare înaintea unui nod dat de o cheie $beforeKey$. În acest caz se executa doi pași:

     (a) Se cauta nodul care are cheia *beforeKey*:

```
SLL_NODE *crt_node = NULL; /* initialize */
SLL_NODE *trail = NULL;

crt_node = list->first;
while ( crt_node != NULL )
{
  if ( crt_node->key == beforeKey ) break;
  trail    = crt_node;
  crt_node = crt_node->next;
}
```

     (b) Se introduce nodul a cărui adresă o avem în variabila *node* în listă, realizând legăturile corespunzătoare:

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

4. Inserarea după un nod care are cheia *afterKey*. Și în acest caz se vor executa doi pași:
   (a) Se cauta nodul care conține *afterKey*:

```
SLL_NODE *crt_node = NULL; /* initialize */

crt_node = list->first;
while ( crt_node != NULL )
{
  if ( crt_node->key == afterKey ) break;
  crt_node = crt_node->next;
}
```

   (b) Se introduce nodul *node* în lista, ajustând legăturile:

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

Ex. 2 — Implementați funcțiile:

```
void sll_insert_first(SL_LIST *list, int given_key);
void sll_insert_last(SL_LIST *list, int given_key);
void sll_insert_after_key(SL_LIST *list, int given_key, int value);
```

care inserează un nod nou într-o lista simplu înlănțuită, utilizând strategiile sugerate de numele funcțiilor.

Ex. 3 — Completați programul scris pana acum astfel:

- •inserați ca prim element, pe rand, cheile 4 și 1
- •inserați cheia 3 ca ultim element
- •căutați cheia 2
- •căutați cheia 3
- •inserați cheia 22 după cheia 4
- •inserați cheia 25 după cheia 3
- •afișați pe ecran conținutul listei - cheile din listă

Ștergerea unui nod dintr-o lista simplu înlănțuită

Operația de ștergere poate prezenta și ea mai multe strategii de ștergere: ștergerea primului nod, a ultimului nod sau a unui nod dat printr-o anumita cheie. Se vor avea în vedere următoarele probleme: lista poate fi vidă, lista poate conține un singur nod sau lista poate conține mai multe noduri.

1. Ștergerea primului nod dintr-o listă:

```
SLL_NODE *node = list->first;  /* initialize */

if ( node != NULL )
{ /* non-empty list */
  list->first = list->first->next;

  free( node ); /* free up memory */
  node = NULL;

  if ( list->first == NULL ) /* list is now empty */
    list->last = NULL;
}
```

2. Ștergerea ultimului nod dintr-o lista:

```
SLL_NODE *crt_node = NULL; /* initialize */
SLL_NODE *trail = NULL;

crt_node = list->first;
if ( crt_node != NULL )
{  /* non-empty list */
  while ( crt_node != list->last )
  {  /* advance towards end */
    trail = crt_node;
    crt_node = crt_node->next;
  }
  if ( crt_node == list->first )
  { /* only one node */
    list->first = list->last = NULL;
  }
  else
  { /* more than one node */
    trail->next = NULL;
    list->last = trail;
  }
  free( crt_node );
  crt_node = NULL;
}
```

3. Ștergerea unui nod care are cheia *givenKey*

```
SLL_NODE *crt_node = NULL; /* initialize */
SLL_NODE *trail = NULL;

/* search node */
crt_node = list->first;
while (crt_node != NULL) {
    if (crt_node->key == givenKey) {
        break;
    }
    trail = crt_node;
    crt_node = crt_node->next;
}
if ( crt_node != NULL ) {   /* found a node with supplied key */
  if ( crt_node == list->first ) { /* is the first node */
    list->first = list->first->next;
    if ( list->first == NULL ) list->last = NULL;
  } else { /* other than first node */
    trail->next = crt_node->next;
    if ( crt_node == list->last ) list->last = trail;
  }

  free( crt_node ); /* release memory */
  crt_node = NULL;
}
```

Pentru o ștergere completă a unei liste, se va șterge primul element în mod repetat, până când lista rămâne goală.

Ex. 4 — Implementați funcțiile:

```
void sll_delete_first(SL_LIST *list);
void sll_delete_last(SL_LIST *list);
void sll_delete_key(SL_LIST *list, int givenKey);
```

care șterg nodul corespunzător din lista simplu înlănțuită dată.

Ex. 5 — Completati programul scris pana acum astfel:

- •ștergeți primul nod
- •ștergeți ultimul nod
- •ștergeți nodul având cheia 22
- •ștergeți nodul având cheia 8
- •afișați pe ecran conținutul listei - cheile din listă
- •ștergeți apoi toată lista
- •afișați conținutul listei

2.4 Implementarea unei liste utilizând alocarea secvențială

Cealaltă alternativă pentru a implementa operațiile tipului de data abstracta lista este utilizarea alocării secvențiale - tablou unidimensional (*vector*). Vom avea nevoie, evident, să alocam un tablou de o anumită dimensiune maximă (*capacity*), dar și să cunoaștem numărul de elemente care se găsesc la fiecare moment în structura noastră (*size*).

Modelul unei liste implementate folosind un vector este dat în Figura 2.

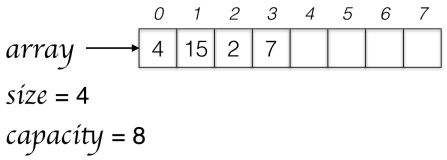

Figura 2: Lista implementata ca si vector

Prin urmare, pentru a implementa lista și operațiile dorite, vom avea nevoie de un vector de o anumită dimensiune (capacitate), și de doi întregi: *capacity* și *size*.

```
typedef struct {
    int *array; /* change this type to store other types of values */
    int size;
    int capacity;
} SEQ_LIST; /* SEQUENTIAL_LIST */
```

Căutarea unei anumite chei într-un vector se face liniar, parcurgând vectorul până la $size - 1$ și verificând, element cu element.

Pentru a insera elementul *elem* la începutul vectorului (i.e. operația $insert\_first$), toate elementele vectorului se muta cu o poziție spre dreapta, dimensiunea (*size*) va crește cu 1, și noul element va fi plasat pe poziția 0 a vectorului:

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

Evident, trebuie verificat în prealabil ca nu s-a atins capacitatea vectorului (i.e. $size < capacity$). În caz contrar, va trebui întâi să re-alocăm un vector de capacitate mai mare (dublam capacitatea de fiecare data când vectorul este plin și avem nevoie să inserăm un element nou), și să copiem în el toate elementele existente în cel curent. Celelalte operații de inserare ($insert\_last$, $insert\_before\_key$, $insert\_after\_key$) se tratează similar.

Pentru operațiile de ștergere, tratarea celulelor care rămân libere se poate face în două moduri: fie să "compactăm" vectorul după fiecare ștergere (prin mutarea tuturor elementelor de după cel șters înspre stânga cu o poziție - similar cu inserarea), fie sa marcam celula ștearsă cu o valoare specială, și compactarea să se realizeze la anumite momente, sincron.

Ex. 6 — Implementati urmatoarele operatii pe un vector de intregi:

```
SEQ_LIST *create_seq_list(int init_capacity);
void destroy_seq_list(SEQ_LIST *list);
void print(const SEQ_LIST *list);
void seql_insert_first(SEQ_LIST *list, int key);
void seql_insert_last(SEQ_LIST *list, int key);
int  seql_search(const SEQ_LIST *list, int key); /* return value is the position*/
void seql_delete_first(SEQ_LIST *list);
void seql_delete_last(SEQ_LIST *list);
void seql_delete_key(SEQ_LIST *list, int key);
```

Testați funcțiile scrise, după cum urmează:

- Alocați un vector de capacitate 8
- Inserați, pe rând, pe prima poziție, cheile 5, 2 și 7
- Afișați conținutul vectorului (cheile)
- Afișați adresa (indexul) cheii 2
- Afișați adresa cheii 20
- Ștergeți elementul de pe prima poziție
- Ștergeți elementul cu cheia 12
- Afișați conținutul vectorului (cheile)

3 Mersul lucrării

Studiați codul prezentat în laborator și utilizați acest cod pentru rezolvarea exercițiilor obligatorii, prezentate pe parcursul lucrării. La finalul sesiunii de laborator, este obligatoriu ca fiecare student sa prezinte codul (compilabil, executabil) cerut în exercițiile de pe parcursul lucrării de laborator.

3.1 Probleme Opționale

1. Să se implementeze o funcție care inversează o lista simplu înlănțuită.
2. Să se implementeze o funcție care inserează un element într-o lista simplu înlănțuită ordonată.
3. Să se implementeze o funcție care găsește elementul de la poziția $length - k$ dintr-o lista simplu înlănțuită (k dat), parcurgând lista o singura dată (și utilizând o cantitate constantă de memorie adițională).
4. O *listă simplu înlănțuită circulară* este lista simplu înlănțuită al cărei ultim element este legat de primul element, (ultimul element are ca următor element primul element). Prin urmare, vom folosi un singur pointer $first$ pentru a indica un element din listă – "primul" element. Figura 3 arată modelul unei astfel de liste.

```
typedef struct {
    SLL_NODE *first;
} CIRC_SL_LIST;
```

Se cere sa implementați o listă simplu înlănțuită circulară, având următoarele operații:

```
void csll_print(const CIRC_SL_LIST *list);
void csll_insert_first(CIRC_SL_LIST *list, int key);
SLL_NODE* csll_search(const CIRC_SL_LIST *list, int key);
void csll_insert_after_key(CIRC_SL_LIST *list, int afterKey, int key);
void csll_delete_key(CIRC_SL_LIST *list, int key);
```

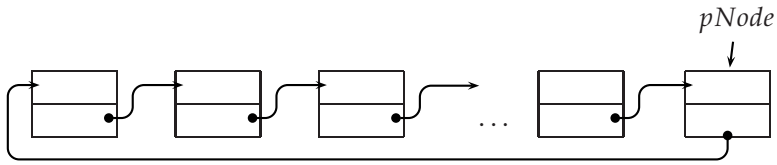

Figura 3: Modelul unei liste circulare simplu înlănţuite

3.2 Probleme Aplicative

1. Să se scrie programul care creează doua liste ordonate crescător după o cheie numerica și 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
```

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: in=inserează în lista $n \in \{1, 2\}$, pn=afișează lista n, m=interclasează listele.

2. Operații cu matrici rare: să se conceapă o structură dinamică eficientă pentru reprezentarea matricelor rare. Să se scrie operații de calcul a sumei și produsului a două matrice rare. Afișarea se va face în 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* = citește elementele matricii *m1*, și tripletele următoare sunt $(row, col, value)$ pentru matrice. Citirea se termină atunci când e dată o altă comandă sau se întâlnește sfârșitul de fișier.

m1+m2 = adună matricea 1 la matricea 2, și *m1*m2* = înmulțește matricea *m1* cu matricea *m2*. Afișarea rezultatelor se va face tot sub formă de triplete.

3. Operații cu polinoame: să se conceapă o structura dinamica eficientă pentru reprezentarea în memorie a polinoamelor. Se vor scrie funcții de calcul a sumei, diferenței și 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
```

Ieșire:

```
<Afișează_suma_polinoamelor>
<Afișează_diferența_polinoamelor>
<Afișează_produsul_polinoamelor>
```

4. Să se definească și să se implementeze funcțiile pentru structura de date definită după cum urmează:

```
typedef struct node
{
  struct node *next;
  void *data;
} GEN_SLL_NODE;  /* GENERIC_SINGLE_LINKED_LIST_NODE */
```

folosind modelul dat în Figura 4. Celulele de date conțin o cheie numerică și un cuvânt – string, de exemplu numele unui student și numărul carnetului de student.

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

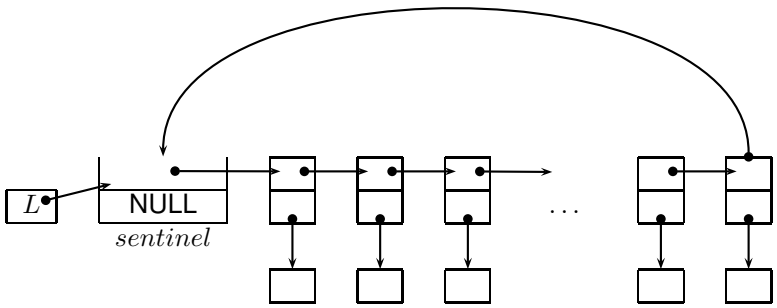

Figura 4: Model de lista pentru problema