# Liste liniare alocate dinamic

Mirel Coşulschi
mirelc@central.ucv.ro

Mihai Gabroveanu
mihaiug@central.ucv.ro

May, 2023

## 1 Liste

**Definiţia 1** *Lista liniară reprezintă o mulţime de elemente omogene (de acelaşi tip) cu următoarele caracteristici:*

- *elementele verifică o relaţie liniară de succesiune (fiecare element are un singur succesor şi un singur predecesor, cu excepţia primului şi ultimului element);*
- *fiecare element are o anumită poziţie în listă.*

*Un tip special de listă este lista vidă (fără niciun element).*

Conform definiţiei 1, o listă este o colecţie de elemente între care este stabilită o anumită ordine. Avem mai multe forme sub care poate fi întâlnită o listă:

- *listă liniară simplu înlănţuită* – fiecare nod conţine o referinţă către elementul următor (`next`). Ultimul element nu are element succesor aşa cum se poate observa în figura 1.


Fig. 1: Structura generală a unei liste liniare simplu înlănţuită

- *listă circulară simplu înlănţuită* – elementul succesor al ultimului element al listei este primul element (a se vedea figura 2).

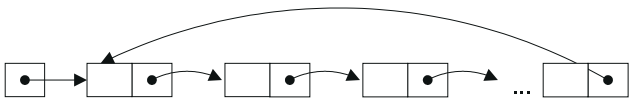


Fig. 2: Structura generală a unei liste circulare simplu înlănţuită

- *listă liniară dublu înlănţuită* – fiecare nod conţine o referinţă către elementul precedent (`prec`) şi către elementul următor (`next`). Primul element nu are element precedent, iar ultimul element nu are element succesor (a se vedea figura 3).

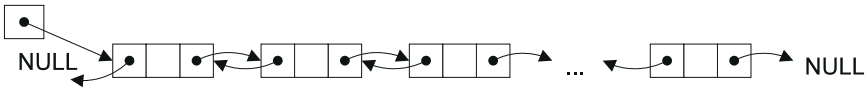


Fig. 3: Structura generală a unei liste liniare dublu înlănţuită

- *listă circulară dublu înlănţuită* – elementul succesor al ultimului element al listei este primul element, iar elementul predecesor al primului element al listei devine ultimul element al listei (a se vedea figura 4).

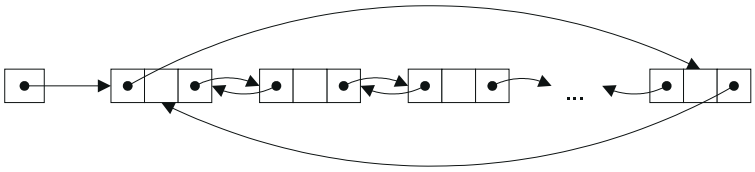


Fig. 4: Structura generală a unei liste circulare dublu înlănţuită

## 1.1 Listă liniară simplu înlănţuită

Pentru o listă simplu înlănţuită ordinea elementelor este indicată explicit printr-un câmp de informaţie ce este întâlnit în cadrul fiecărui element (`data`), şi un câmp de legătură ce indică elementul următor (`next`).

În figura 5 se poate observa structura generală a unui element.

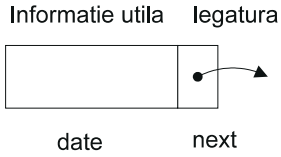


Fig. 5: Structura generală a unui nod într–o listă simplu înlănţuită

Tipul element al unei liste simplu înlănţuite se poate declara în limbajul *C* astfel:

```
typedef struct nod {
    TipOarecare data;
    struct nod* next;
} Nod;
```

Declaraţia unei variabile de tip *Nod*:

```
Nod element;
```

Câmpurile variabilei de tip `Nod` vor fi accesate prin `element.data` şi `element.next`, iar toată înregistrarea prin `element`.

Declaraţia unei variabile de tip pointer la *Nod*:

```
Nod* p;
```

Variabila `p` este de tip *pointer la Nod* (variabila păstrează adresa unei zone de memorie ce conţine date numerice de tip `Nod`). Câmpurile variabilei de tip `Nod` vor fi accesate prin `p->data` şi `p->next`, iar toată înregistrarea prin `*p`.

Constanta `NULL` are o semnificaţie specială[1]: `NULL` este o constantă simbolică definită în fişierul header `<stdio.h>`.

Un pointer ce are valoarea `NULL` semnifică un pointer ce are valoarea 0 (prin convenţie, indică faptul că face referire către o zonă de memorie nevalidă / nealocată). Valoarea `NULL` pentru câmpul `next` semnifică faptul că nu mai avem un element următor pentru înregistrarea curentă.

Pentru a prelucra o listă avem nevoie de un pointer către primul element al listei (cunoscut şi sub denumirea de *capul listei*) (`cap`, `head`, `first`). Dacă acesta are valoarea `NULL` se consideră că lista este vidă (nu conţine niciun element). Deseori, pentru a evita unele operaţii în plus, este indicat să păstrăm şi un pointer către ultimul element al listei (`ultim`, `tail`, `last`).

### 1.1.1 Inserarea unui element într-o listă

Pentru inserarea unui element într–o listă avem nevoie de doi pointeri:
*cap* – adresa primului element al listei
*p* – adresa unui element ce urmează să fie inserat în listă

1. *inserarea unui element la începutul listei* - se poate realiza prin următoarea secvenţă de instrucţiuni (a se vedea figura 6):

```
Nod *head, *p;
...
p->next = head;
head = p;
...
```

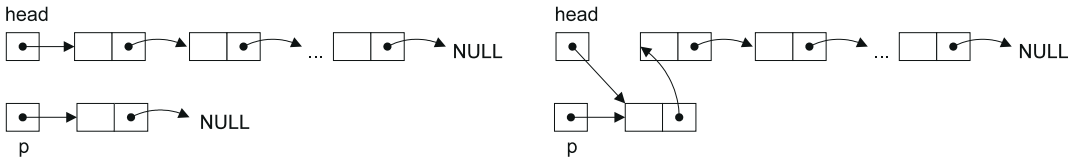


Fig. 6: Inserare în listă la început

2. *inserarea unui element in interiorul listei* - în vederea efectuării acestei operaţii avem nevoie de o referinţă către elementul după care se realizează inserarea. Să considerăm că numele acestui pointer este `q` (a se vedea figura 7).

```
Nod *q, *p;
...
p->next = q->next;
q->next = p;
...
```

[1]It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. `https://www.unf.edu/~wkloster/2220/ppts/cprogramming_tutorial.pdf`

Dacă se specifică un element `q` **înaintea căruia** trebuie efectuată operaţia de inserare, atunci lista trebuie parcursă de la început pentru localizarea elementului anterior lui `q`.

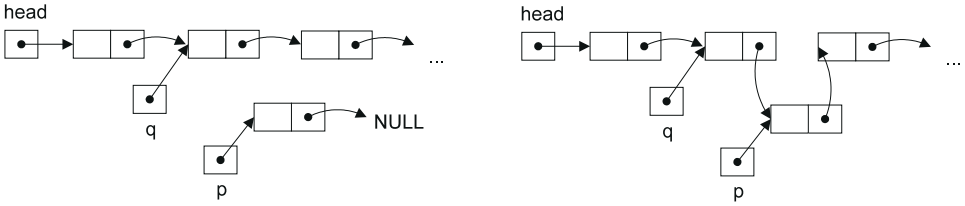


Fig. 7: Inserarea unui element în interiorul listei

3. *inserarea unui element la sfârşitul listei* - se poate realiza prin următoarea secvenţă de instrucţiuni (a se vedea figura 8):

```
Nod *head, *p, *q;
...
if (head == NULL) {
  head = p;
} else {
    q = head;
    while (q->next != NULL) {
      q = q->next;
    }

    q->next = p;
}
...
```

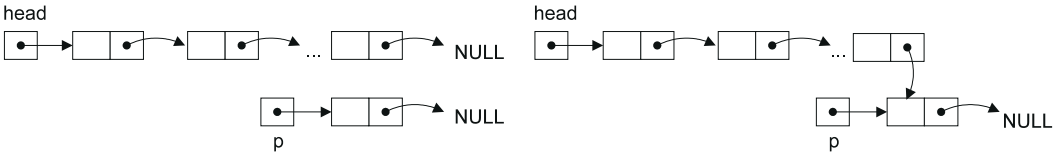


Fig. 8: Inserare în listă la sfârşit

### 1.1.2 Ştergerea unui element într-o listă

Prin operaţia de ştergere înţelegem eliminarea unui element din cadrul înlănţuirii. Acel element poate fi utilizat în continuare sau memoria ocupată de el poate fi eliberată.

Pentru ştergerea unui element dintr-o listă vom folosi aceiaşi doi pointeri ca şi în secţiunea anterioară:

*cap* – adresa primului element al listei
*p* – adresa unui element ce se doreşte să fie şters din listă

1. *ştergerea primului element dintr-o listă* - se poate realiza prin următoarea secvenţă de instrucţiuni (a se vedea figura 9):

```
Nod *head, *p;
...
```

```
p = head;
head = head->next;
// se poate elibera zona de memorie alocata elementului eliminat din lista
// prin intermediul functiei free().
...
```

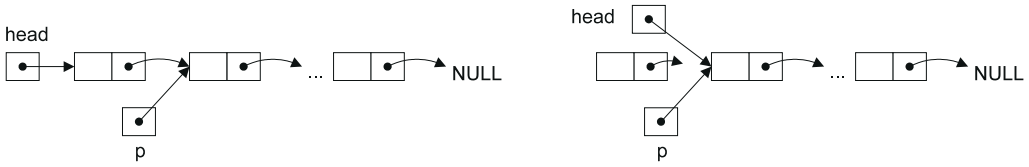


Fig. 9: Ştergerea primului element dintr-o listă

2. *ştergerea unui element din interiorul listei* - se poate realiza prin următoarea secvenţă de instrucţiuni (a se vedea figura 10).

   În vederea efectuării acestei operaţii este necesară o referinţă către elementul precedent celui care se elimină din listă. Fie `q` acest pointer (`q->next == p`).

```
Nod *head, *p;
...
p = q->next;
q->next = p->next;
// se poate elibera zona de memorie alocata elementului eliminat din lista
// prin intermediul functiei free().
...
```

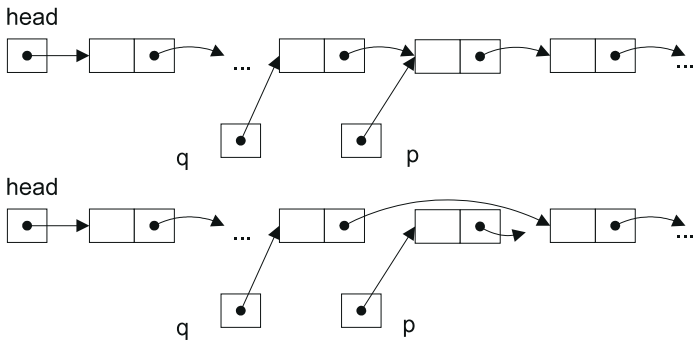


Fig. 10: Ştergerea unui element din interiorul listei

## 1.2 Aplicaţie - Listă liniară simplu înlănţuită

Să considerăm o listă liniară simplu înlănţuită unde informaţia păstrată în elementele listei este de tip întreg:

```
typedef struct node {
    int inf;           // valoarea elementului curent
    struct node* next; // adresa urmatorului element din lista
} Node;
```

Se defineşte tipul lista `List` astfel:

```
typedef struct list {
    Node* head;      // adresa primului element din lista
} List;
```

Prezentăm în continuare codul sursă al unui program scris în limbajul *C* corespunzător implementării principalelor operaţii asupra elementelor unei liste liniare simplu înlănţuită (varianta I).

Listing 1: `sll-intregi-v1.c`

```
// Implementarea pricipalelor operatii cu o lista liniara simplu inlantuita (LLSI)
// Pentru o lista vom pastra doar adresa primului element (head).

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#define LMAX 35

typedef struct node {
    int inf;           // valoarea elementului curent
    struct node* next; // adresa urmatorului element din lista
} Node;

typedef struct list {
    Node* head;      // adresa primului element din lista
} List;

char menu[][LMAX] = {"1. Inserare", "2. Stergere", "3. Cautare", "4. Afisare",
                     "5. Ordonare elemente lista", "0. Iesire"};
char submenu1[][LMAX] = {"1. Inserare la inceput", "2. Inserare la sfarsit",
                         "3. Inserare dupa un element", "0. Return"};
char submenu2[][LMAX] = {"1. Stergerea primului element",
                         "2. Stergerea ultimului element",
                         "3. Stergerea unui element", "0. Return"};

/*
*  Functia aloca memorie pentru un element al listei si initializeaza campurile
* inf si next.
* @param value - valoarea intreaga cu care se va initializa campul inf
* @return      - adresa zonei de memorie unde a fost alocat spatiu pentru
*                un element
*/
Node* createNode(int value) {
   Node* p;

   p = (Node*)malloc(sizeof(Node));
   p->inf = value;
   p->next = NULL;

   return p;
}

/*
```

```
*  Functia initializeaza o lista cu lista vida (campurile head si tail vor avea
* valoarea NULL).
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
*/
void initList(List* list) {
  list->head = NULL;
}

/*
*  Functia cauta un element cu o valoare specificata intre elementele unei liste
* liniare simplu inlantuita.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
* @param value - o valoare intreaga de cautat in lista
* @return  - adresa primului element ce are informatia egala cu valoarea cautata
*            sau NULL daca nu exista un astfel de element
*/
Node* searchElement(List* list, int value) {
  Node *current;

  current = list->head;
  while ((current != NULL) && (current->inf != value)) {
    current = current->next;
  }

  return current;
}

/*
*  Functia intoarce adresa ultimului element al unei liste.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
* @return  - adresa ultimului element al unei liste
*            sau NULL daca lista este vida
*/
Node* getLastElement(List* list) {
  Node *current;

  current = list->head;
  if (current != NULL) {
    while (current->next != NULL) {
      current = current->next;
    }
  }

  return current;
}

/*
*   Adauga un element cu o valoare specificata la inceputul unei liste
* liniare simplu inlantuita.
```

```
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
* @param value - o valoare intreaga de adaugat la lista
*/
void insertBeforeHead(List* list, int value) {
  Node* p;

  p = createNode(value);      // se creaza un element nou

  p->next = list->head;       // se adauga elementul curent la inceputul listei
  list->head = p;             // primul element devine elementul curent
}

/*
*   Adauga un element cu o valoare specificata la sfarsitul unei liste
* liniare simplu inlantuita.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
* @param value - o valoare intreaga de adaugat la lista
*/
void insertAfterEnd(List* list, int value) {
  Node* p;
  Node* current;

  p = createNode(value);      // se creaza un element nou

  if (list->head == NULL) {
    list->head = p;
  } else {
      current = getLastElement(list);
      current->next = p;
  }
}

/*
*   Insereaza un element cu o valoare specificata dupa un element a carui
* informatie este egala cu o alta valoare specificata value2. Daca nu
* exista in cadrul listei liniare un element cu valoarea value2, atunci
* inserarea nu se realizeaza.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
* @param value1 - valoare intreaga de adaugat la lista
* @param value2 - valoarea intreaga a elementului dupa care se va adauga noul
*                 element la lista
* @return  - 1 daca inserarea se efectueaza cu succes
*            0 daca inserarea nu se efectueaza
*/
int insertAfterElement(List* list, int value1, int value2) {
  Node *p, *current;

  // cauta elementul cu informatia egala cu value2
  current = searchElement(list, value2);
```

```
  if (current != NULL) {      // daca elementul exista
    p = createNode(value1);   // se creaza un element nou

    // se adauga elementul nou creat dupa elementul curent
    p->next = current->next;
    current->next = p;
  }

  return (current != NULL);
}

/*
*  Sterge elementul aflat la inceputul unei liste liniare simplu inlantuita.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
*/
void deleteHead(List* list) {
  Node* current;

  if (list->head != NULL) {
    current = list->head;
    list->head = current->next;

    free(current);
  }
}

/*
*  Sterge elementul aflat la sfarsitul unei liste liniare simplu inlantuita.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
*/
void deleteTail(List* list) {
  Node* current;
  Node* previous;

  if (list->head != NULL) {      // daca lista nu este vida
    current = list->head;
    if (current->next == NULL) { // daca lista are un singur element
      list->head = NULL;
    } else {
        previous = current;
        current = current->next;
        while (current->next != NULL) {
          previous = current;
          current = current->next;
        }

        previous->next = NULL;
    }
```

```
    free(current);
  }
}

/*
*   Sterge dintr-o lista liniara simplu inlantuita primul element a carui
* informatie este egala cu o valoare specificata.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
* @param value - valoarea elementului ce urmeaza sa fie sters din lista
* @return      - 1 daca stergerea a fost efectuata cu succes
*                0 daca stergerea nu a putut fi efectuata
*/
int deleteElementByValue(List* list, int value) {
  Node *current, *previous;
  int found = 0;

  if (list->head != NULL) {         // daca lista nu este vida
    current = list->head;
    if (current->inf == value) {    // daca elementul de sters este primul
      deleteHead(list);             // sterge primul element din lista
      found = 1;                    // stergerea a fost efectuata
    } else {
        previous = current;         // elementul anterior celui curent
        current = current->next;    // trecem la al doilea element
        // cautam in lista elementul ce trebuie sters
        while ((current != NULL) && (current->inf != value)) {
          previous = current;
          current = current->next;
        }

        if (current != NULL) {      // daca elementul a fost gasit
          found = 1;                // stergerea va fi realizata
          previous->next = current->next; // sarim peste elementul curent

          free(current);
        }
    }
  }

  return found;
}

/*
*   Ordoneaza crescator dupa valoarea campului de informatie elementele unei
* liste liniare simplu inlantuita.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
*/
void sortList(List* list){
  Node *p, *q;
  int tmp;
```

```
  p = list->head;
  // cat timp nu am ajuns la sfarsitul listei
  while (p) {      // echivalent cu: while (p != NULL)
    q = p->next;   // elementul urmator celui curent
    while (q) {    // cat timp nu am ajuns la sfarsitul listei
      if (p->inf > q->inf) {
        tmp = p->inf; p->inf = q->inf; q->inf = tmp;
      }

      q = q->next;          // trecem la elementul urmator
    }

    p = p->next;            // trecem la elementul urmator
  }
}

/*
*  Afiseaza valorile elementelor unei liste liniare simplu inlantuita.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
*/
void printAll(List* list) {
  Node *current = list->head;

  printf("[");
  while (current) {
    printf("%d", current->inf);
    current = current->next;

    if (current != NULL) {
      printf(", ");
    }
  }
  printf("]\n");
}

/*
*  Cisteste un caracter de la dispozitivul standard de intrare
* @return  - codul ASCII al unui caracter citit de la tastatura
*/
int input() {
  int ch = 0;

  ch = _getch();

  return ch;
}

/*
*  Afiseaza sirurile de caractere trimise ca argument al functiei.
* @param menuItems - un tablou de siruri de caractere
```

```
* @param n          - numarul de siruri de caractere din tablou
*/
void printMenu(char menuItems[][LMAX], int n) {
  int i;

  for (i = 0; i < n; i++) {
    printf("%s\n", menuItems[i]);
  }
  printf("\n");
}

/*
*  Afiseaza elementele submeniului 1, citeste optiunea utilizatorului si, in
* functie de aceasta, realizeaza inserarea unui element intr-o lista.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
*               ce pastreaza adresa primului element al unei liste liniare.
*/
void dosubmenu1(List* list) {
  int ch;
  int value, value2;

  printMenu(submenu1, sizeof(submenu1) / sizeof(submenu1[0]));
  while ((ch = input()) != '0') {
    switch (ch) {
      case '1': printf("Dati valoarea elementului de inserat:");
                scanf("%d", &value);
                insertBeforeHead(list, value);
                break;
      case '2': printf("Dati valoarea elementului de inserat:");
                scanf("%d", &value);
                insertAfterEnd(list, value);
                break;
      case '3': printf("Dati valoarea elementului de inserat:");
                scanf("%d", &value);
                printf("Dati valoarea elementului dupa care se insereaza:");
                scanf("%d", &value2);

                if (insertAfterElement(list, value, value2) == 0) {
                  printf("Elementul cu valoarea %d nu a fost gasit!\n", value2);
                }
                break;
      default: printf("Comanda necunoscuta!\n");
    }

    printMenu(submenu1, sizeof(submenu1) / sizeof(submenu1[0]));
  }
}

/*
*  Afiseaza elementele submeniului 2, citeste optiunea utilizatorului si, in
* functie de aceasta, realizeaza stergerea unui element dintr-o lista.
* @param list - pointer catre un element de tip List: acesta are un camp, head,
```

```
*               ce pastreaza adresa primului element al unei liste liniare.
*/
void dosubmenu2(List* list) {
  int ch;
  int value;

  printMenu(submenu2, sizeof(submenu2) / sizeof(submenu2[0]));
  while ((ch = input()) != '0') {
    switch (ch) {
      case '1': printf("Se incearca stergerea primului element din lista!\n");
                deleteHead(list);
                break;
      case '2': printf("Se incearca stergerea ultimului element din lista!\n");
                deleteTail(list);
                break;
      case '3': printf("Dati valoarea elementului de sters:");
                scanf("%d", &value);

                if (deleteElementByValue(list, value) == 0) {
                  printf("Elementul cu valoarea %d nu a fost gasit!\n", value);
                }
                break;
      default: printf("Comanda necunoscuta!\n");
    }

    printMenu(submenu2, sizeof(submenu2) / sizeof(submenu2[0]));
  }
}

int main() {
  List list;
  Node* p;
  int ch;
  int value;

  initList(&list);

  printMenu(menu, sizeof(menu) / sizeof(menu[0]));
  while ((ch = input()) != '0') {
    switch (ch) {
      case '1': dosubmenu1(&list);
                break;
      case '2': dosubmenu2(&list);
                break;
      case '3': printf("Dati valoarea elementului de cautat: ");
                scanf("%d", &value);

                p = searchElement(&list, value);
                if (p == NULL) {
                  printf("Elementul cu valoarea %d nu a fost gasit!\n", value);
                } else {
                    printf("A fost gasit elementul cu valoarea %d!\n", value);
```

```
                }
                break;
      case '4': printf("Afisarea elementelor listei:\n");
                printAll(&list);
                break;
      case '5': printf("Ordonarea elementelor listei dupa valoare.\n");
                sortList(&list);
                break;
      default:  printf("Comanda necunoscuta!\n");
    }

    printMenu(menu, sizeof(menu) / sizeof(menu[0]));
  }

  return 0;
}
```

Dacă se doreşte ca adăugarea unui element la sfârşitul listei (după ultimul elemente existent) să se efectueze într-un timp constant, indiferent de dimensiunea listei, trebuie să păstăm şi adresa ultimului element:

```
typedef struct list {
    Node* head;      // adresa primului element din lista
    Node* tail;      // adresa ultimului element din lista
} List;
```

Prezentăm în continuare codul sursă al variantei a II-a ce include principalele operaţii asupra elementelor unei liste liniare simplu înlănţuită.

Listing 2: `sll-intregi-v2.c`

```
// Implementarea pricipalelor operatii cu o lista liniara simplu inlantuita (LLSI)
// Pentru o lista vom pastra atat adresa primului element (head) cat si adresa
// ultimului element (tail).

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

#define LMAX 35

typedef struct node {
    int inf;           // valoarea elementului curent
    struct node* next; // adresa urmatorului element din lista
} Node;

typedef struct list {
    Node* head;      // adresa primului element din lista
    Node* tail;      // adresa ultimului element din lista
} List;

char menu[][LMAX] = {"1. Inserare", "2. Stergere", "3. Cautare", "4. Afisare",
                     "5. Ordonare elemente lista", "0. Iesire"};
char submenu1[][LMAX] = {"1. Inserare la inceput", "2. Inserare la sfarsit",
                         "3. Inserare dupa un element", "0. Return"};
```

```
char submenu2[][LMAX] = {"1. Stergerea primului element",
                         "2. Stergerea ultimului element",
                         "3. Stergerea unui element", "0. Return"};

/*
*  Functia aloca memorie pentru un element al listei si initializeaza campurile
* inf si next.
* @param value - valoarea intreaga cu care se va initializa campul inf
* @return      - adresa zonei de memorie unde a fost alocat spatiu pentru
*                un element
*/
Node* createNode(int value) {
   Node* p;

   p = (Node*)malloc(sizeof(Node));
   p->inf = value;
   p->next = NULL;

   return p;
}

/*
*  Functia initializeaza o lista cu lista vida (campurile head si tail vor avea
* valoarea NULL).
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void initList(List* list) {
  list->head = list->tail = NULL;
}

/*
*  Functia cauta un element cu o valoare specificata intre elementele unei liste
* liniare simplu inlantuita.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param value - o valoare intreaga de cautat in lista
* @return  - adresa primului element ce are informatia egala cu valoarea cautata
*            sau NULL daca nu exista un astfel de element
*/
Node* searchElement(List* list, int value) {
  Node *current;

  current = list->head;
  while ((current != NULL) && (current->inf != value)) {
    current = current->next;
  }

  return current;
}

/*
```

```
*   Adauga un element cu o valoare specificata la inceputul unei liste
* liniare simplu inlantuita.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param value - o valoare intreaga de adaugat la lista
*/
void insertBeforeHead(List* list, int value) {
  Node* p;

  p = createNode(value);      // se creaza un element nou

  p->next = list->head;       // se adauga elementul curent la inceputul listei
  list->head = p;             // primul element devine elementul curent
  if (list->tail == NULL) {   // daca lista nu avea niciun element
    list->tail = p;           // atunci head = tail = elementul curent
  }
}

/*
*   Adauga un element cu o valoare specificata la sfarsitul unei liste
* liniare simplu inlantuita.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param value - o valoare intreaga de adaugat la lista
*/
void insertAfterEnd(List* list, int value) {
  Node* p;

  p = createNode(value);      // se creaza un element nou

  if (list->tail == NULL) {   // daca lista nu avea niciun element
    list->head = p;           // atunci head = tail = elementul curent
  } else {
      list->tail->next = p; // se adauga elementul curent la sfarsitul listei
  }

  list->tail = p;             // ultimul element devine elementul curent
}

/*
*   Insereaza un element cu o valoare specificata dupa un element a carui
* informatie este egala cu o alta valoare specificata value2. Daca nu
* exista in cadrul listei liniare un element cu valoarea value2, atunci
* inserarea nu se realizeaza.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param value  - valoare intreaga de adaugat la lista
* @param value2 - valoarea intreaga a elementului dupa care se va adauga noul
*                 element la lista
* @return  - 1 daca inserarea se efectueaza cu succes
*            0 daca inserarea nu se efectueaza
*/
```

```
int insertAfterElement(List* list, int value1, int value2) {
  Node *p, *current;

  // cauta elementul cu informatia egala cu value2
  current = searchElement(list, value2);

  if (current != NULL) {      // daca elementul exista
    p = createNode(value1);   // se creaza un element nou

    // se adauga elementul nou creat dupa elementul curent
    p->next = current->next;
    current->next = p;
    if (list->tail == current) {
      list->tail = p;
    }
  }

  return (current != NULL);
}

/*
*  Sterge elementul aflat la inceputul unei liste liniare simplu inlantuita.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void deleteHead(List* list) {
  Node* current;

  if (list->head != NULL) {
    current = list->head;
    list->head = list->head->next;

    if (list->head == NULL) {
      list->tail = NULL;
    }

    free(current);
  }
}

/*
*  Sterge elementul aflat la sfarsitul unei liste liniare simplu inlantuita.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void deleteTail(List* list) {
  Node* current;

  if (list->tail != NULL) {
    current = list->head;
    if (list->head == list->tail) {
      list->head = list->tail = NULL;
```

```
    } else {
        while (current->next != list->tail) {
          current = current->next;
        }

        list->tail = current;
        current = current->next;
        list->tail->next = NULL;
    }

    free(current);
  }
}

/*
*   Sterge dintr-o lista liniara simplu inlantuita primul element a carui
* informatie este egala cu o valoare specificata.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param value - valoarea elementului ce urmeaza sa fie sters din lista
* @return      - 1 daca stergerea a fost efectuata cu succes
*                0 daca stergerea nu a putut fi efectuata
*/
int deleteElementByValue(List* list, int value) {
  Node *current, *previous;
  int found = 0;

  if (list->head != NULL) {          // daca lista nu este vida
    if (list->head->inf == value) {  // daca elementul de sters este primul
      deleteHead(list);              // sterge primul element din lista
      found = 1;                     // stergerea a fost efectuata
    } else {
        previous = NULL;             // elementul anterior celui curent
        current = list->head;        // elementul curent
        // cautam in lista elementul ce trebuie sters
        while ((current != NULL) && (current->inf != value)) {
          previous = current;
          current = current->next;
        }

        if (current != NULL) {       // daca elementul a fost gasit
          found = 1;                 // stergerea va fi realizata
          previous->next = current->next; // sarim peste elementul curent

          if (current == list->tail) { // daca este ultimul
            list->tail = previous;   // actualizam ultimul element
          }

          free(current);
        }
    }
  }
```

```
  return found;
}

/*
*   Ordoneaza crescator dupa valoarea campului de informatie elementele unei
* liste liniare simplu inlantuita.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void sortList(List* list){
  Node *p, *q;
  int tmp;

  p = list->head;
  // cat timp nu am ajuns la sfarsitul listei
  while (p) {      // echivalent cu: while (p != NULL)
    q = p->next;   // elementul urmator celui curent
    while (q) {    // cat timp nu am ajuns la sfarsitul listei
      if (p->inf > q->inf) {
        tmp = p->inf; p->inf = q->inf; q->inf = tmp;
      }

      q = q->next;          // trecem la elementul urmator
    }

    p = p->next;            // trecem la elementul urmator
  }
}

/*
*  Afiseaza valorile elementelor unei liste liniare simplu inlantuita.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void printAll(List* list) {
  Node *current = list->head;

  printf("[");
  while (current) {
    printf("%d", current->inf);
    current = current->next;

    if (current != NULL) {
      printf(", ");
    }
  }
  printf("]\n");
}

/*
*  Cisteste un caracter de la dispozitivul standard de intrare
```

```
* @return  - codul ASCII al unui caracter citit de la tastatura
*/
int input() {
  int ch = 0;

  ch = _getch();

  return ch;
}

/*
*  Afiseaza sirurile de caractere trimise ca argument al functiei.
* @param menuItems - un tablou de siruri de caractere
* @param n         - numarul de siruri de caractere din tablou
*/
void printMenu(char menuItems[][LMAX], int n) {
  int i;

  for (i = 0; i < n; i++) {
    printf("%s\n", menuItems[i]);
  }
  printf("\n");
}

/*
*  Afiseaza elementele submeniului 1, citeste optiunea utilizatorului si, in
* functie de aceasta, realizeaza inserarea unui element intr-o lista.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void dosubmenu1(List* list) {
  int ch;
  int value, value2;

  printMenu(submenu1, sizeof(submenu1) / sizeof(submenu1[0]));
  while ((ch = input()) != '0') {
    switch (ch) {
      case '1': printf("Dati valoarea elementului de inserat:");
                scanf("%d", &value);
                insertBeforeHead(list, value);
                break;
      case '2': printf("Dati valoarea elementului de inserat:");
                scanf("%d", &value);
                insertAfterEnd(list, value);
                break;
      case '3': printf("Dati valoarea elementului de inserat:");
                scanf("%d", &value);
                printf("Dati valoarea elementului dupa care se insereaza:");
                scanf("%d", &value2);

                if (insertAfterElement(list, value, value2) == 0) {
                  printf("Elementul cu valoarea %d nu a fost gasit!\n", value2);
```

```
                }
                break;
      default: printf("Comanda necunoscuta!\n");
    }

    printMenu(submenu1, sizeof(submenu1) / sizeof(submenu1[0]));
  }
}

/*
*  Afiseaza elementele submeniului 2, citeste optiunea utilizatorului si, in
* functie de aceasta, realizeaza stergerea unui element dintr-o lista.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void dosubmenu2(List* list) {
  int ch;
  int value;

  printMenu(submenu2, sizeof(submenu2) / sizeof(submenu2[0]));
  while ((ch = input()) != '0') {
    switch (ch) {
      case '1': printf("Se incearca stergerea primului element din lista!\n");
                deleteHead(list);
                break;
      case '2': printf("Se incearca stergerea ultimului element din lista!\n");
                deleteTail(list);
                break;
      case '3': printf("Dati valoarea elementului de sters:");
                scanf("%d", &value);

                if (deleteElementByValue(list, value) == 0) {
                  printf("Elementul cu valoarea %d nu a fost gasit!\n", value);
                }
                break;
      default: printf("Comanda necunoscuta!\n");
    }

    printMenu(submenu2, sizeof(submenu2) / sizeof(submenu2[0]));
  }
}

int main() {
  List list;
  Node* p;
  int ch;
  int value;

  initList(&list);

  printMenu(menu, sizeof(menu) / sizeof(menu[0]));
  while ((ch = input()) != '0') {
```

```
    switch (ch) {
      case '1': dosubmenu1(&list);
                break;
      case '2': dosubmenu2(&list);
                break;
      case '3': printf("Dati valoarea elementului de cautat: ");
                scanf("%d", &value);

                p = searchElement(&list, value);
                if (p == NULL) {
                  printf("Elementul cu valoarea %d nu a fost gasit!\n", value);
                } else {
                    printf("A fost gasit elementul cu valoarea %d!\n", value);
                }
                break;
      case '4': printf("Afisarea elementelor listei:\n");
                printAll(&list);
                break;
      case '5': printf("Ordonarea elementelor listei dupa valoare.\n");
                sortList(&list);
                break;
      default:  printf("Comanda necunoscuta!\n");
    }

    printMenu(menu, sizeof(menu) / sizeof(menu[0]));
  }

  return 0;
}
```

## 1.3 Aplicaţie - Listă dublu înlănţuită

*Să se implementeze o listă dublu înlănţuită ale cărei elemente să fie studenţii unei facultăţi. Programul va conţine funcţii pentru:*

- *crearea listei vide.*
- *afişarea elementelor listei.*
- *căutarea unui student în listă.*
- *adăugarea unui student în listă (la început, la sfârşit, după un anumit nod specificat).*
- *ştergerea unui anumit student din listă.*

Să considerăm o listă liniară dublu înlănţuită unde informaţia păstrată în elementele listei este de tip `Student`:

```
typedef struct nod {
    Student info;
    struct nod* prev;   // pointer catre nodul precedent
    struct nod* next;   // pointer catre nodul succesor
} Nod;
```

Se defineşte tipul lista `DLList` (*double linked list*) astfel:

```
typedef struct dllist {
    Nod* head;      // adresa primului element al listei
    Nod* tail;      // adresa ultimului element al listei
} DLList;
```

Prezentăm în continuare codul sursă al unui program scris în limbajul *C* ce implementează operaţiile asupra elementelor unei liste liniare dublu înlănţuită, solicitate în enunţul problemei.

Listing 3: `dll-studenti.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>

/*
*  Definirea tipului Student.
*/
typedef struct student {
    char nume[20];      // numele studentului
    int grupa;          // grupa
    int anul;           // anul de studiu
} Student;

/*
*  Definirea tipului Nod, ce descrie un element al unei liste liniare dublu
* inlantuita.
*/
typedef struct nod {
    Student info;
    struct nod* prev;   // pointer catre nodul precedent
    struct nod* next;   // pointer catre nodul succesor
} Nod;

typedef struct dllist {
    Nod* head;      // adresa primului element al listei
    Nod* tail;      // adresa ultimului element al listei
} DLList;

/*
*  Functia citeste informatii despre un student.
* @param p - adresa unei zone de memorie ce pastreaza informatii de tip Student
*/
void citesteInformatiiStudent(Student* p) {
  _flushall();
  printf("Nume:");  gets(p->nume);
  printf("Grupa:"); scanf("%d", &p->grupa);
  printf("Anul:");  scanf("%d", &p->anul);
}

/*
```

```
*  Functia afiseaza informatiile aferente unui student.
*/
void afiseazaInformatiiStudent(Student s) {
  printf("Nume: %-20s Grupa: %4d  Anul: %d\n", s.nume, s.grupa, s.anul);
}

/*
*  Functia initializeaza o lista liniara dublu inlantuita cu lista vida
* (campurile head si tail vor avea valoarea NULL).
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void initList(DLList* l) {
  l->head = l->tail = NULL;
}

/*
*  Functia aloca memorie pentru un element al listei si initializeaza campurile
* info, next si prev.
* @param s - informatii despre un student
* @return  - adresa zonei de memorie unde a fost alocat spatiu pentru
*            un element
*/
Nod* creazaNod(Student s) {
   Nod* p;

   p = (Nod*)malloc(sizeof(Nod));
   p->info = s;
   p->next = NULL;
   p->prev = NULL;

   return p;
}

/*
*  Functie pentru adaugarea unui element in capul unei liste.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param s - informatii despre un student
*/
void adaugaCap(DLList* list, Student s) {
  Nod* prim = list->head;
  Nod* ultim = list->tail;
  Nod *q;

  // Alocam spatiu pentru a retine nodul care se adauga
  q = creazaNod(s);

  q->next = prim;
  if (prim == NULL) { // daca lista este vida
    ultim = q;        // se adauga primul element
  } else {
```

```
      prim->prev = q;
  }

  prim = q;

  list->head = prim;
  list->tail = ultim;
}

/*
*  Functie pentru adaugarea unui element la sfarsitul unei liste.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param s    - informatii despre un student
*/
void adaugaCoada(DLList* list, Student s) {
  Nod* prim = list->head;
  Nod* ultim = list->tail;
  Nod* q;

  // Alocam spatiu pentru a retine nodul care se adauga
  q = creazaNod(s);

  q->prev = ultim;
  if (prim == NULL) { // daca lista este vida
    prim = q;         // se adauga primul elemnt
  } else {
      ultim->next = q;
  }

  ultim = q;

  list->head = prim;
  list->tail = ultim;
}

/*
*  Functie pentru adaugarea unui student dupa un nod specificat 'pNod'.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param pNod - adresa unui element al listei dupa care se efectueaza inserarea
* @param s    - informatii despre un student
*/
void adaugaStudent(DLList* list, Nod* pNod, Student s) {
  Nod* ultim = list->tail;
  Nod* q;

  // Alocam spatiu pentru a retine nodul care se adauga
  q = creazaNod(s);

  q->prev = pNod;
  q->next = pNod->next;
```

```
  if (pNod == ultim) {
    ultim = q;        // se adauga dupa ultimul nod din lista
  } else {
      pNod->next->prev = q;
  }

  pNod->next = q;

  list->tail = ultim;
}

/*
*   Functia creaza o lista liniara dublu inlantuita.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void creareLista(DLList* list) {
  Student s;
  char c;

  do {
    citesteInformatiiStudent(&s);
    adaugaCoada(list, s);

    printf("Continuati (d/n):");
    _flushall();
    c = getchar();
  } while ((c == 'd') || (c == 'D'));
}

/*
*   Functia afiseaza elementele unei liste.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void afiseazaLista(DLList* list) {
  Nod *p;

  p = list->head;
  while ( p ) {
    afiseazaInformatiiStudent(p->info);
    p = p->next;
  }
}

/*
*   Functia afiseaza elementele unei liste de la sfarsit spre inceput.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
*/
void afiseazaListaInvers(DLList* list) {
  Nod *p;
```

```
  p = list->tail;
  while ( p ) {
    afiseazaInformatiiStudent(p->info);
    p = p->prev;
  }
}

/*
*  Functia cauta un student dupa nume in lista. Functia returneaza adresa nodului
* unde se afla studentul daca acesta a fost gasit, si NULL daca nu a fost gasit.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param s    - structura ce contine informatiile de identificare ale
*               studentului cautat
*/
Nod* cautaStudent(DLList* list, Student s) {
  Nod *p;

  p = list->head;
  while ( p && (strcmp(p->info.nume, s.nume) != 0) ) {
    p = p->next;
  }

  return p;
}

/*
*  Functie utilizata pentru stergerea unei inregistrari de tip Student din lista.
* @param list - pointer catre o zona de memorie unde se afla adresele capetelor
*               unei liste (adresa primului element si adresa ultimului element)
* @param s - structura ce contine informatiile de identificare ale
*            studentului ce urmeaza sa fie sters din lista.
*/
void stergeStudent(DLList* list, Student s) {
  Nod* p = list->head;
  Nod* u = list->tail;
  Nod* q;

  q = cautaStudent(list, s);
  if ( q ) {
    if ((p == u) && (p == q)) {
      // Avem un singur element in lista
      p = u = NULL;
    } else {
        if (p == q) {
          // Se sterge primul element din lista
          p->next->prev = NULL;
          p = p->next;
        } else {
            if (u == q) {
              // Se sterge ultimul element din lista
```

```
              u->prev->next = NULL;
              u = u->prev;
            } else {
                // Se sterge un nod interior
                q->prev->next = q->next;
                q->next->prev = q->prev;
            }
        }
    }

    free(q);

    list->head = p;
    list->tail = u;
    printf("Studentul a fost sters.\n");
  } else {
      printf("Studentul nu a fost gasit.\n");
  }
}

int main() {
  Nod *p;
  Student s;
  char c;
  DLList list;

  initList(&list);

  do {
    printf("1. Creare lista studenti.\n");
    printf("2. Afisare lista studenti.\n");
    printf("3. Afisare lista studenti in ordine inversa.\n");
    printf("4. Cautare student.\n");
    printf("5. Adaugare student la inceput.\n");
    printf("6. Adaugare student la sfarsit.\n");
    printf("7. Adaugare student dupa un alt student.\n");
    printf("8. Stergere student.\n");
    printf("9. Iesire.\n");

    printf("Selectati o comanda [1-9].\n");
    _flushall();
    c = getchar();

    switch ( c ) {
      case '1': creareLista(&list);
                break;
      case '2': afiseazaLista(&list);
                break;
      case '3': afiseazaListaInvers(&list);
                break;
      case '4': printf("Dati numele studentului cautat:");
                _flushall();
```

```
                gets(s.nume);

                p = cautaStudent(&list, s);
                if ( p ) {
                  printf("Studentul a fost gasit:\n");
                  afiseazaInformatiiStudent(p->info);
                } else {
                    printf("Studentul nu a fost gasit.\n");
                }
                break ;
     case '5': printf("Dati datele pentru studentul care se adauga:\n");
                citesteInformatiiStudent(&s);
                adaugaCap(&list, s);
                break;
     case '6': printf("Dati datele pentru studentul care se adauga:\n");
                citesteInformatiiStudent(&s);
                adaugaCoada(&list, s);
                break;
     case '7': printf("Dati numele studentului dupa care se adauga:\n");
                _flushall();
                gets(s.nume);

                p = cautaStudent(&list, s);
                if ( p ) {
                  printf("Studentul a fost gasit:\n");
                  afiseazaInformatiiStudent(p->info);

                  printf("Dati datele pentru studentul care se adauga:\n");
                  citesteInformatiiStudent(&s);
                  adaugaStudent(&list, p, s);
                } else {
                    printf("Studentul nu a fost gasit.\n");
                }
                break;
     case '8': printf("Dati numele studentului care se sterge:");
                _flushall();
                gets(s.nume);

                stergeStudent(&list, s);
                break;
    }

    printf("Apasati o tasta!\n");
    _getch();
  } while (c != '9');

  return 0;
}
```

# References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introducere în Algoritmi*, Computer Libris Agora, Cluj-Napoca, 1999.

[2] M. Coşulschi, M. Gabroveanu, *Practica programării în C*, Editura Universitaria, Craiova, 2014.