Liste liniare alocate dinamic. Probleme rezolvate

Mirel Coşulschi
mirelc@central.ucv.ro

May, 2023

1 Probleme cu liste

1. Se dă o listă liniară simplu înlănţuită ce conţine n numere naturale (fiecare valoare este stocată în câte un nod al listei).

   Să se implementeze:

   - o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
   - o funcţie ce primeşte ca parametru un pointer la începutul unei liste liniare simplu înlănţuită şi inversează ordinea nodurilor din listă (fără alocare de memorie auxiliară pentru o nouă listă).

     Exemplu: pentru lista $1 \to 2 \to 3$ rezultă lista $3 \to 2 \to 1$.

   Date de intrare

   Fişierul de intrare `list.in` conţine pe prima linie numărul n, iar pe linia următoare n numere naturale separate prin spaţii.

   Date de ieşire

   Fişierul de ieşire `list.out` va conţine pe o linie numerele fiecărui nod al listei separate prin spaţii, după inversarea listei.

   $1 \leq n \leq 100.000$.

   Rezolvare:

   Variant a I-a:

   Listing 1: `llsi-invers-v1.c`

```c
#include <stdio.h>
#include <stdlib.h>

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;
```

```
/*
*  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 (campul head va 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 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 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;
  }
}

/*
*  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");
}

/*
*  Functia primeste ca parametru un pointer la inceputul unei liste liniare
* simplu inlantuita si inverseaza ordinea nodurilor din lista (fara alocare
* de memorie auxiliara pentru o noua 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 reverse(List* list) {
  Node *previous, *current, *follower;
```

```
  previous = NULL;
  current = list->head;
  while (current != NULL) {
    follower = current->next;

    current->next = previous;

    previous = current;
    current = follower;
  }

  list->head = previous;
}

int main() {
  FILE *fin, *fout;
  List list;
  int n, i, value;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  initList(&list);

  fscanf(fin, "%d", &n);
  for (i = 0; i < n; i++) {
    fscanf(fin, "%d", &value);

    insertAfterEnd(&list, value);
  }

  printAll(&list);

  reverse(&list);

  printAll(&list);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

Variant a II-a:

Listing 2: `llsi-invers-v2.c`

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

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;

/*
*  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 (campul head va 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;
}

/*
*   Creaza o lista liniara simplu inlantuita prin adaugari succesive a cate
* unui element cu o valoare specificata la sfarsitul unei liste.
* @param fin  - pointer catre o structura de tip FILE; reprezinta fisierul
*               text cu datele de intrare
* @param list - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al unei liste
*               liniare.
*/
void createList(FILE* fin, List* list) {
  Node* p;
  Node* current;
  int i, n, value;

  initList(list);

  fscanf(fin, "%d", &n);
  for (i = 0; i < n; i++) {
    fscanf(fin, "%d", &value);
```

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

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

    current = p;
  }
}

/*
*  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");
}

/*
*  Functia primeste ca parametru un pointer la inceputul unei liste liniare
* simplu inlantuita si inverseaza ordinea nodurilor din lista (fara alocare
* de memorie auxiliara pentru o noua 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 reverse(List* list) {
  Node *previous, *current, *follower;

  previous = NULL;
  current = list->head;
  while (current != NULL) {
    follower = current->next;

    current->next = previous;
```

```
    previous = current;
    current = follower;
  }

  list->head = previous;
}

int main() {
  FILE *fin, *fout;
  List list;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  createList(fin, &list);

  printAll(&list);

  reverse(&list);

  printAll(&list);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

2. Se dau două liste liniare simplu înlănţuite ce conţin n respectiv m numere naturale (fiecare valoare este stocată în câte un nod al listei).

   Să se implementeze:

   - o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
   - o funcţie ce primeşte ca parametri doi pointeri la începuturile a două liste simplu înlănţuite sortate şi întoarce o listă simplu înlănţuită sortată ce conţine toate elementele din cele două liste.

     Exemplu: pentru listele $1 \to 2 \to 5 \to 9$ şi $2 \to 3 \to 7 \to 8 \to 10$ rezultă lista $1 \to 2 \to 2 \to 3 \to 5 \to 7 \to 8 \to 9 \to 10$.

   Date de intrare

   Fişierul de intrare `list.in` conţine pe prima linie numerele n şi m, pe linia a doua n numere naturale separate prin spaţii iar pe linia a treia m numere naturale separate prin spaţii.

   Date de ieşire

   Fişierul de ieşire `list.out` va conţine pe o linie numerele fiecărui nod al listei obţinute în urma interclasării celor două liste, separate prin spaţii.

   $1 \leq n, m \leq 100.000$.

   Rezolvare:

Listing 3: `llsi-interclasare.c`

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

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;

/*
*  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 (campul head va 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;
}

/*
*   Creaza o lista liniara simplu inlantuita prin adaugari succesive a cate
* unui element cu o valoare specificata la sfarsitul unei liste.
* @param fin  - pointer catre o structura de tip FILE; reprezinta fisierul
*               text cu datele de intrare
* @param n    - numarul de elemente al listei rezultate
* @param list - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al unei liste
*               liniare.
*/
void createList(FILE* fin, int n, List* list) {
```

```
  Node* p;
  Node* current;
  int i, value;

  initList(list);

  for (i = 0; i < n; i++) {
    fscanf(fin, "%d", &value);

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

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

    current = p;
  }
}

/*
*  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");
}

/*
*  Functia primeste ca parametri doi pointeri la inceputurile a doua liste
* simplu inlantuite sortate si intoarce o lista simplu inlantuita sortata
* ce contine toate elementele din cele doua liste.
* @param list1 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al primei
*                liste liniare.
* @param list2 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al celei de-a
*                doua liste liniare.
```

```
* @param list3 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al listei
*                rezultata in urma interclasarii elementelor celor doua
*                liste.
*/
void merge(List* list1, List* list2, List* list3) {
  Node *current1, *current2, *current3;

  initList(list3);

  current1 = list1->head;
  current2 = list2->head;
  current3 = list3->head;
  while ((current1 != NULL) || (current2 != NULL)) {
    if ((current2 == NULL)
        || ((current2 != NULL) && (current1 != NULL)
             && (current1->inf <= current2->inf))) {
      if (current3 == NULL) {
        list3->head = list1->head;
      } else {
          current3->next = current1;
      }

      current3 = current1;
      current1 = current1->next;
    } else {
        if (current2 != NULL) {
          if (current3 == NULL) {
            list3->head = list2->head;
          } else {
              current3->next = current2;
          }

          current3 = current2;
          current2 = current2->next;
        }
    }
  }
}

int main() {
  FILE *fin, *fout;
  List list1, list2, list3;
  int n, m;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  fscanf(fin, "%d %d", &n, &m);
  createList(fin, n, &list1);

  createList(fin, m, &list2);
```

```
  printAll(&list1);
  printAll(&list2);

  merge(&list1, &list2, &list3);

  printAll(&list3);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

3. Se dă o listă liniară simplu înlănţuită ce conţine n numere naturale (fiecare valoare este stocată în câte un nod al listei).

   Să se implementeze:

   - o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
   - o funcţie ce primeşte ca parametru un pointer la începutul unei liste liniare simplu înlănţuită şi şterge elementul aflat în mijlocul listei. Ştergerea trebuie să se facă printr-o singură parcurgere a listei.

     Exemple:

     – pentru lista vidă, rezultă lista vidă;
     – pentru lista 1, rezultă lista vidă;
     – pentru lista $1 \to 2$, rezultă lista 2;
     – pentru lista $1 \to 2 \to 3$, rezultă lista $1 \to 3$;
     – pentru lista $1 \to 2 \to 3 \to 4$, rezultă lista $1 \to 3 \to 4$;
     – pentru lista $1 \to 2 \to 3 \to 4 \to 5$, rezultă lista $1 \to 2 \to 4 \to 5$.

   Date de intrare

   Fişierul de intrare `list.in` conţine pe prima linie numărul n, iar pe linia următoare n numere naturale separate prin spaţii.

   Date de ieşire

   Fişierul de ieşire `list.out` va conţine pe o linie numerele fiecărui nod al listei separate prin spaţii, după ştergerea elementului aflat în mijlocul listei.

   $1 \leq n \leq 100.000$.

   Rezolvare:

Listing 4: `llsi-delete-middle.c`

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

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;

/*
*  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) {
  // ...
}

/*
*  Functia initializeaza o lista cu lista vida (campul head va 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) {
  // ...
}

/*
*  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) {
  // ...
}

/*
*   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) {
  // ...
}
```

```
/*
*  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) {
  // ...
}

/*
*  Functia primeste ca parametru un pointer la inceputul unei liste liniare
* simplu inlantuita si inverseaza ordinea nodurilor din lista (fara alocare
* de memorie auxiliara pentru o noua 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 deleteMiddle(List* list) {
  Node *previous, *current, *middle;

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

    if (previous) {
      previous->next = middle->next;
    } else {
        list->head = middle->next;
    }

    free(middle);
  }
}

int main() {
  FILE *fin, *fout;
  List list;
  int n, i, value;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  initList(&list);
```

```
  fscanf(fin, "%d", &n);
  for (i = 0; i < n; i++) {
    fscanf(fin, "%d", &value);

    insertAfterEnd(&list, value);
  }

  printAll(&list);

  deleteMiddle(&list);

  printAll(&list);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

4. Se dă o listă liniară simplu înlănţuită ce conţine n numere naturale (fiecare valoare este stocată în câte un nod al listei).

   Să se implementeze:

   - o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
   - o funcţie ce primeşte ca parametri un pointer la începutul unei liste liniare simplu înlănţuită şi un element şi adaugă elementul în mijlocul listei. Adăugarea trebuie să se facă printr-o singură parcurgere a listei.

     Exemple (elementul ce trebuie adăugat este X):

     - pentru lista vidă, rezultă lista X;
     - pentru lista 1, rezultă lista $X \to 1$;
     - pentru lista $1 \to 2$, rezultă lista $1 \to X \to 2$;
     - pentru lista $1 \to 2 \to 3$, rezultă lista $1 \to X \to 2 \to 3$;
     - pentru lista $1 \to 2 \to 3 \to 4$, rezultă lista $1 \to 2 \to X \to 3 \to 4$;
     - pentru lista $1 \to 2 \to 3 \to 4 \to 5$, rezultă lista $1 \to 2 \to X \to 3 \to 4 \to 5$.

   Date de intrare

   Fişierul de intrare `list.in` conţine pe prima linie numerele nX, iar pe linia următoare n numere naturale separate prin spaţii.

   Date de ieşire

   Fişierul de ieşire `list.out` va conţine pe o linie numerele fiecărui nod al listei separate prin spaţii, după adaăugarea elementului X în mijlocul listei.

   $1 \leq n \leq 100.000$.

   Rezolvare:

Listing 5: `llsi-insert-middle.c`

```c
#include <stdio.h>
#include <stdlib.h>

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;

/*
*  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) {
  // ...
}

/*
*  Functia initializeaza o lista cu lista vida (campul head va 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) {
  // ...
}

/*
*  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) {
  // ...
}

/*
*   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) {
  // ...
}

/*
*  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) {
  // ...
}

/*
*  Functia primeste ca parametri un pointer la inceputul unei liste liniare
* simplu inlantuita si valoarea unui element. Se adauga elementul in
* mijlocul listei. Adaugarea se face printr-o singura parcurgere a listei.
* @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 naturala de adaugat la lista
*/
void insertMiddle(List* list, int value) {
  Node *previous, *current, *middle;
  Node *p;

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

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

    p->next = middle;
    if (previous) {
      previous->next = p;
    } else {
        list->head = p;
    }
  } else {
      list->head = p;
```

```
  }
}

int main() {
  FILE *fin, *fout;
  List list;
  int n, i, value, x;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  initList(&list);

  fscanf(fin, "%d %d", &n, &x);
  for (i = 0; i < n; i++) {
    fscanf(fin, "%d", &value);

    insertAfterEnd(&list, value);
  }

  printAll(&list);

  insertMiddle(&list, x);

  printAll(&list);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

5. Se dă o listă liniară simplu înlănţuită ce conţine n numere naturale (fiecare valoare este stocată în câte un nod al listei).

   Să se implementeze:

   - o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
   - o funcţie ce primeşte ca parametru un pointer la începutul unei liste liniare simplu înlănţuită şi construieşte două liste în care se vor afla toate elementele de pe poziţii pare, respectiv impare, în aceeaşi ordine.

     Exemplu: pentru lista $1 \to 2 \to 2 \to 3 \to 5 \to 7 \to 8 \to 9$, rezultă listele $1 \to 2 \to 5 \to 8$ şi $2 \to 3 \to 7 \to 9$.

   Date de intrare

   Fişierul de intrare `list.in` conţine pe prima linie numărul n, iar pe linia următoare n numere naturale separate prin spaţii.

   Date de ieşire

   Fişierul de ieşire `list.out` va conţine pe prima linie numerele fiecărui nod al primei liste, şi pe a doua linie valorile fiecărui nod din cea de-a doua listă, separate prin spaţii.

$1 \leq n \leq 100.000$.

Rezolvare:

Listing 6: `llsi-split.c`

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

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;

/*
*  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) {
  // ...
}

/*
*  Functia initializeaza o lista cu lista vida (campul head va 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) {
  // ...
}

/*
*  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) {
  // ...
}

/*
*   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) {
  // ...
}

/*
*  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) {
  // ...
}

/*
*  Functia primeste ca parametri un pointer la inceputul unei liste liniare
* simplu inlantuita si construieste doua liste in care se vor afla toate
* elementele de pe pozitii pare, respectiv impare, in aceeasi ordine.
* @param list - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al unei liste
*               liniare.
* @param listA - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al listei cu
*               elementele de pe pozitiile impare din prima lista.
* @param listB - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al listei cu
*               elementele de pe pozitiile pare din prima lista.
*/
void split(List* list, List* listA, List* listB) {
  Node *current1, *current2, *current3;
  int odd;

  initList(listA);
  initList(listB);

  current1 = list->head;
  odd = 1;
  while (current1) {
    if (odd) {
      if (listA->head) {
        current2->next = current1;
      } else {
          listA->head = current1;
      }
      current2 = current1;
    } else {
```

```
        if (listB->head) {
          current3->next = current1;
        } else {
            listB->head = current1;
        }
        current3 = current1;
    }

    odd = 1 - odd;
    current1 = current1->next;
  }

  if (listA->head) {
    current2->next = NULL;
  }

  if (listB->head) {
    current3->next = NULL;
  }
}

int main() {
  FILE *fin, *fout;
  List list, listOdd, listEven;
  int n, i, value;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  initList(&list);

  fscanf(fin, "%d", &n);
  for (i = 0; i < n; i++) {
    fscanf(fin, "%d", &value);

    insertAfterEnd(&list, value);
  }

  printAll(&list);

  split(&list, &listOdd, &listEven);

  printAll(&listOdd);
  printAll(&listEven);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

6. Se dă o listă liniară simplu înlănţuită ce conţine n numere naturale (fiecare valoare este

stocată în câte un nod al listei).

Să se implementeze:

- o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
- o funcţie ce primeşte ca parametri un pointer la începutul unei liste liniare simplu înlănţuită şi un element X, şi reordonează nodurile din listă astfel încât toate nodurile cu valori mai mici sau egale decât X să apară înaintea nodurilor cu valori mai mari decât X. Nu se vor aloca noduri noi!

  Exemplu: pentru lista $3 \to 6 \to 5 \to 4 \to 2$ şi valoarea 3, o posibilă listă rezultat este $2 \to 3 \to 4 \to 6 \to 5$.

Date de intrare

Fişierul de intrare `list.in` conţine pe prima linie numerele `n X`, iar pe linia următoare n numere naturale separate prin spaţii.

Date de ieşire

Fişierul de ieşire `list.out` va conţine pe o linie numerele fiecărui nod al listei separate prin spaţii, după reordonarea listei.

$1 \leq n \leq 100.000$.

Rezolvare:

Listing 7: `llsi-reorder-v1.c`

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

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;

/*
*  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) {
  // ...
}

/*
*  Functia initializeaza o lista cu lista vida (campul head va 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) {
  // ...
}

/*
*  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) {
  // ...
}

/*
*   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) {
  // ...
}

/*
*  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) {
  // ...
}

/*
*  Functia primeste ca parametri un pointer la inceputul unei liste liniare
* simplu inlantuita si valoarea unui element. Se reordoneaza nodurile din
* lista astfel incat toate nodurile cu valori mai mici sau egale decat X
* sa apara inaintea nodurilor cu valori mai mari decat X. Nu se vor aloca
* noduri noi.
* @param list - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al unei liste
*               liniare.
* @param x    - o valoare naturala
*/
```

```
void reorder(List* list, int x) {
  Node *current1, *current2, *current3;
  List listA, listB;

  initList(&listA);
  initList(&listB);

  current1 = list->head;
  while (current1) {
    if (current1->inf <= x) {
      if (listA.head) {
        current2->next = current1;
      } else {
          listA.head = current1;
      }
      current2 = current1;
    } else {
        if (listB.head) {
          current3->next = current1;
        } else {
            listB.head = current1;
        }
        current3 = current1;
    }

    current1 = current1->next;
  }

  if (listA.head != NULL) {
    list->head = listA.head;
    current2->next = listB.head;
  } else {
      list->head = listB.head;
  }

  if (listB.head != NULL) {
    current3->next = NULL;
  }
}

int main() {
  FILE *fin, *fout;
  List list;
  int n, i, value, x;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  initList(&list);

  fscanf(fin, "%d %d", &n, &x);
  for (i = 0; i < n; i++) {
```

```
    fscanf(fin, "%d", &value);

    insertAfterEnd(&list, value);
  }

  printAll(&list);

  reorder(&list, x);

  printAll(&list);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

7. Se dau două liste liniare simplu înlănţuite ce conţin n respectiv m numere întregi (fiecare valoare este stocată în câte un nod al listei).

   Să se implementeze:

   - o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
   - o funcţie ce primeşte ca parametri doi pointeri la începuturile a două liste simplu înlănţuite, A şi B, şi întoarce o listă simplu înlănţuită C, pentru care fiecare nod de pe poziţia i este suma valorilor nodurilor asociate din A şi B. Mai exact, nodul i din C reţine suma dintre valoarea nodului i din A şi valoarea nodului i din B. Dacă una dintre listele primite este mai lungă decât cealaltă, se consideră că nodurile asociate lipsă din cealaltă listă conţin valoarea 0, adică se păstrează valorile din lista mai lungă.

     Exemplu: pentru listele A: $3 \to 7 \to 29 \to 4$ şi B: $2 \to 4 \to 3$, va rezulta lista C: $5 \to 11 \to 32 \to 4$.

   Date de intrare

   Fişierul de intrare `list.in` conţine pe prima linie numerele `n m`, pe linia a doua n numere întregi separate prin spaţii (elementele primei liste) iar pe linia a treia m numere întregi separate prin spaţii (elementele celei de-a doua liste).

   Date de ieşire

   Fişierul de ieşire `list.out` va conţine pe o linie numerele fiecărui nod al listei obţinute în urma "adunării" valorilor nodurilor asociate celor două liste, separate prin spaţii.

   $1 \leq n, m \leq 100.000$.

   Rezolvare:

Listing 8: `llsi-suma-v1.c`

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

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;

/*
*  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) {
  // ...
}

/*
*  Functia initializeaza o lista cu lista vida (campul head va 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) {
  // ...
}

/*
*   Creaza o lista liniara simplu inlantuita prin adaugari succesive a cate
* unui element cu o valoare specificata la sfarsitul unei liste.
* @param fin  - pointer catre o structura de tip FILE; reprezinta fisierul
*               text cu datele de intrare
* @param n    - numarul de elemente al listei rezultate
* @param list - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al unei liste
*               liniare.
*/
void createList(FILE* fin, int n, List* list) {
  // ...
}

/*
*  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) {
  // ...
```

```
}

/*
*  Functia primeste ca parametri doi pointeri la inceputurile a doua liste
* simplu inlantuite si intoarce o lista simplu inlantuita pentru care
* fiecare nod de pe pozitia i este suma valorilor nodurilor asociate din
* list1 si list2. Mai exact, nodul i din list3 retine suma dintre valoarea
* nodului i din list1 si valoarea nodului i din list2.
* @param list1 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al primei
*                liste liniare.
* @param list2 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al celei de-a
*                doua liste liniare.
* @param list3 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al listei
*                rezultata in urma adunarii valorilor nodurilor celor
*                doua liste.
*/
void addlist(List* list1, List* list2, List* list3) {
  Node *current1, *current2, *current3;
  Node *p;

  initList(list3);

  current1 = list1->head;
  current2 = list2->head;
  current3 = list3->head;
  while ((current1 != NULL) || (current2 != NULL)) {
    p = createNode(0);      // se creaza un element nou
    if (current1) {
      p->inf += current1->inf;
      current1 = current1->next;
    }

    if (current2) {
      p->inf += current2->inf;
      current2 = current2->next;
    }

    if (list3->head == NULL) {
      list3->head = p;
    } else {
        current3->next = p;
    }

    current3 = p;
  }
}

int main() {
  FILE *fin, *fout;
```

```
  List list1, list2, list3;
  int n, m;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  fscanf(fin, "%d %d", &n, &m);
  createList(fin, n, &list1);

  createList(fin, m, &list2);

  printAll(&list1);
  printAll(&list2);

  addlist(&list1, &list2, &list3);

  printAll(&list3);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

8. Se dau două liste liniare simplu înlănţuite cu n respectiv m elemente, ale căror noduri stochează în ordine inversă cifrele câte unui număr natural reprezentat în baza 10 (primul nod al unei liste stochează cea mai puţin semnificativă cifră).

   Să se implementeze:

   - o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
   - o funcţie ce primeşte ca parametri doi pointeri la începuturile a două liste simplu înlănţuite, A şi B (ce reprezintă cifrele a două numere naturale), şi întoarce o listă simplu înlănţuită C, ce stochează suma celor două numere. Lista C se va construi în timp ce se parcurg listele A şi B.

     Exemplu: pentru listele A: $4 \to 3 \to 2 \to 9$ şi B: $6 \to 6 \to 7$, rezultă lista C: $0 \to 0 \to 0 \to 0 \to 1$.

   Date de intrare

   Fişierul de intrare `list.in` conţine pe prima linie numerele `n m`, pe linia a doua n numere naturale separate prin spaţii (elementele primei liste) iar pe linia a treia m numere naturale separate prin spaţii (elementele celei de-a doua liste).

   Date de ieşire

   Fişierul de ieşire `list.out` va conţine pe o linie numerele fiecărui nod al listei obţinute în urma "adunării" valorilor nodurilor asociate celor două liste, separate prin spaţii.

   $1 \leq n, m \leq 100.000$.

   Rezolvare:

Listing 9: `llsi-bignumber-add-v1.c`

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

#define BASE 10

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;

/*
*  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) {
  // ...
}

/*
*  Functia initializeaza o lista cu lista vida (campul head va 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) {
  // ...
}

/*
*   Creaza o lista liniara simplu inlantuita prin adaugari succesive a cate
* unui element cu o valoare specificata la sfarsitul unei liste.
* @param fin  - pointer catre o structura de tip FILE; reprezinta fisierul
*               text cu datele de intrare
* @param n    - numarul de elemente al listei rezultate
* @param list - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al unei liste
*               liniare.
*/
void createList(FILE* fin, int n, List* list) {
  // ...
}

/*
```

```
*  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) {
  // ...
}

/*
*  Functia primeste ca parametri doi pointeri la inceputurile a doua liste
* simplu inlantuite ce contin cifrele a doua numere naturale, si intoarce
* o lista simplu inlantuita, ce stocheaza suma celor doua numere.
* @param list1 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al primei
*                liste liniare.
* @param list2 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al celei de-a
*                doua liste liniare.
* @param list3 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al listei
*                rezultata in urma adunarii valorilor nodurilor celor
*                doua liste.
*/
void bignumberAdd(List* list1, List* list2, List* list3) {
  Node *current1, *current2, *current3;
  Node *p;
  int carry = 0;

  initList(list3);

  current1 = list1->head;
  current2 = list2->head;
  current3 = list3->head;
  while ((current1 != NULL) || (current2 != NULL)) {
    p = createNode(carry);

    if (current1) {
      p->inf += current1->inf;
      current1 = current1->next;
    }

    if (current2) {
      p->inf += current2->inf;
      current2 = current2->next;
    }

    carry = p->inf / BASE;
    p->inf = p->inf % BASE;

    if (list3->head == NULL) {
      list3->head = p;
```

```
    } else {
        current3->next = p;
    }

    current3 = p;
  }

  if (carry > 0) {
    p = createNode(carry);
    current3->next = p;
  }
}

int main() {
  FILE *fin, *fout;
  List list1, list2, list3;
  int n, m;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  fscanf(fin, "%d %d", &n, &m);
  createList(fin, n, &list1);

  createList(fin, m, &list2);

  printAll(&list1);
  printAll(&list2);

  bignumberAdd(&list1, &list2, &list3);

  printAll(&list3);

  fclose(fin);
  fclose(fout);

  return 0;
}
```

9. Se dă o listă liniară simplu înlănţuită ce conţine n numere naturale (fiecare valoare este stocată în câte un nod al listei).

   Să se implementeze:

   - o funcţie ce crează o listă liniară simplu înlănţuită, prin inserări succesive la sfârşitul listei;
   - o funcţie ce primeşte ca parametru un pointer la începutul unei liste liniare simplu înlănţuită sortată crescător şi elimină nodurile cu valori duplicate (păstrând un singur nod din fiecare grup de duplicate).

     Exemplu: pentru lista $1 \to 2 \to 2 \to 3 \to 5 \to 5 \to 5 \to 9$ rezultă lista $1 \to 2 \to 3 \to 5 \to 9$.

Date de intrare

Fişierul de intrare `list.in` conţine pe prima linie numărul n, iar pe linia următoare n numere naturale separate prin spaţii.

Date de ieşire

Fişierul de ieşire `list.out` va conţine pe o linie numerele fiecărui nod al listei separate prin spaţii.

$1 \leq n \leq 100.000$.

Rezolvare:

Listing 10: `llsi-remove-duplicate-v1.c`

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

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;

/*
*  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) {
  // ...
}

/*
*  Functia initializeaza o lista cu lista vida (campul head va 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) {
  // ...;
}

/*
*   Creaza o lista liniara simplu inlantuita prin adaugari succesive a cate
* unui element cu o valoare specificata la sfarsitul unei liste.
* @param fin  - pointer catre o structura de tip FILE; reprezinta fisierul
*               text cu datele de intrare
* @param n    - numarul de elemente al listei rezultate
```

```
* @param list - pointer catre un element de tip List: acesta are un camp,
*               head, ce pastreaza adresa primului element al unei liste
*               liniare.
*/
void createList(FILE* fin, int n, List* list) {
  // ...
}

/*
*  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) {
  // ...
}

/*
*  Functia primeste ca parametru un pointer la inceputul unei liste liniare
* simplu inlantuita sortata crescator si elimina nodurile cu valori
* duplicate (pastrand un singur nod din fiecare grup de duplicate).
* @param list1 - pointer catre un element de tip List: acesta are un camp,
*                head, ce pastreaza adresa primului element al unei
*                liste liniare.
*/
void removeduplicates(List* list) {
  Node *current1, *current2;
  Node *p = NULL;

  current2 = list->head;
  current1 = current2->next;
  while (current1 != NULL) {
    if (current2->inf != current1->inf) {
      current2->next = current1;
      current2 = current1;
    } else {
        p = current1;
    }

    current1 = current1->next;
    if (p) {
      free(p);
      p = NULL;
    }
  }

  current2->next = NULL;
}

int main() {
  FILE *fin, *fout;
```

```
  List list1;
  int n;

  fin = fopen("list.in", "r");
  fout = fopen("list.out", "w");

  fscanf(fin, "%d", &n);
  createList(fin, n, &list1);

  printAll(&list1);

  removeduplicates(&list1);

  printAll(&list1);

  fclose(fin);
  fclose(fout);

  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.