**Ministerul Educației și Cercetării**

|  |
| --- |
| **Universitatea Tehnica a Moldovei** |

**Facultatea Calculatoare, Informatica si Microelectronica**

**Structuri de date și Algoritmi**

**Raport**

|  |
| --- |
| Lucrarea de Laborator nr. 5  Tema: Algoritmi de prelucrare a listelor liniare simplu înlănțuite, listelor dublu înlănțuite, listelor circulare, cozilor și stivelor |

|  |  |
| --- | --- |
| A efectuat stundent(ul/a) grupei: | SI-212 |
| Numele si prenumele elev(ului/ei): | Vozian Vladimir |
|  |  |
| Profesorul: | Mititelu Vitalie |

|  |
| --- |
| Chișinău 2022 |

**Scopul lucrării:**

Obținerea deprinderilor practice de implementare și de utilizare a tipurilor abstracte de date „Listă liniară simplu înlănțuită”, „Listă liniară dublu înlănțuită”, „Listă circulară”, „Stivă”, „Coadă” în limbajul C cu asigurarea operațiilor de prelucrare de bază ale listei.

**Sarcina pe variante:**

16. Scrieți un program, care din trei liste simplu înlănțuite să selecteze într-o listă nouă mai întâi numerele divizibile la 8, 4 și 2, apoi numerele negative impare. Se cere de a-l diviza în 3 subliste.

**Rezumat**

O listă liniară simplu înlănțuită conține elemente (noduri) a căror valori constau din două părți: informația utilă și informația de legătură. Informația utilă reprezintă informația propriu-zisă memorată în elementul liste (numere, șiruri de caractere, etc.), iar informația de legătură precizează adresa următorului element al listei. În C/C++ putem folosi următorul tip de date pentru a memora elementele unei liste liniare simplu înlănțuite alocate dinamic:

struct nod{

int info;

nod \* urm;

};

Câmpul info al tipului nod reprezintă informația utilă – în acest caz un număr întreg, iar câmpul urm este de tip pointer la nod și reprezintă informația de legătură.

În program vom folosi o variabilă de tip pointer (de exemplu prim) pentru a memora adresa primului element al listei și fiecare element al listei, începând cu primul, va memora în câmpul urm adresa elementului următor. Excepție face ultimul element al listei care va memora în câmpul urm valoarea NULL.

Observații:

* La început prim va avea valoarea NULL, cu semnificația că lista este vidă. Dacă la un moment dat lista redevine vidă (de exemplu se șterg toate elementele ei) variabila prim va avea valoarea NULL.
* Elementele listei sunt variabile dinamice, create cu ajutorul operatorului C++ new și gestionate prin intermediul pointerilor. Variabila prim este de tip pointer, dar este (în cele ce urmează) statică.
* Fiind variabile dinamice, pentru elementele listei se alocă memorie în HEAP.
* Informațiile de legătură ocupă memorie. Spațiul de memorie ocupat de un pointer depinde de versiunea compilatorului folosit; în general este de 4 octeți. Astfel, fiecare element al unei liste de tipul de mai sus va ocupa în memorie 4+4=8 octeți.
* Accesul la un nod al listei se face prin parcurgerea nodurilor care îl preced.

O secvență C++ care conține declarațiile corespunzătoare poate fi:

struct nod{

int info;

nod \* urm;

};

nod \* prim = NULL;

În continuare vom prezenta secvențe/funcții C++ pentru principalele operații:

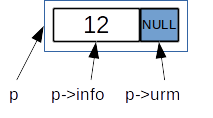
* crearea unui element nou,
* adăugarea unui element la sfârșitul listei,
* adăugarea unui element la începutul listei,
* parcurgerea listei,
* ștergere unui element din listă,
* inserarea unui element în listă.

Funcțiile care urmează vor avea ca parametru adresa primului element al listei și eventual alți parametri. În funcție de situație, parametrul care reprezintă adresa primului element ale listei va fi transmis prin valoare sau prin referință.

Crearea unui element nou

Numeroase operații cu liste solicită crearea unui nou element/nod. Pentru aceasta trebuie să ținem cont de următoarele:

* Nodurile sunt variabile dinamice. Crearea unui nou nod înseamnă crearea unei variabile dinamice. Acest lucru se face cu ajutorul operatorului C++ new, care are ca rezultat adresa variabilei nou create. Aceasta va fi memorată într-un pointer de tip nod \*. Să-l numim p: nod \* p = new nod;
* Nodurile sunt variabile de tip structură, cu câmpurile info și urm. Accesul la câmpuri se va face prin intermediul pointerilor, cu ajutorul operatorului ->, astfel: p->info și p->urm. Accesul la câmpuri se poate face și după dereferențierea pointer-ului: (\* p).info și (\* p).urm.
* Nodul nou creat va fi inclus într-o listă. p->urm va memora adresa următorului element, sau NULL dacă nu există următorul element!
* Rezumat:
  + p este pointer la nod; este de tip nod \*;
  + \*p este variabila de tip nod – este nod din listă
  + p->info este informația utilă din nodul listei, de tip int
  + p->urm este pointer. Memorează adresa elementului următor!



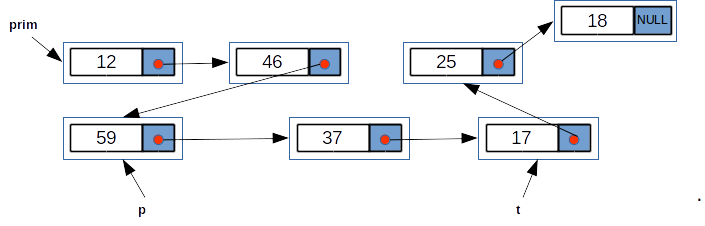
Secvența C++:

nod \* p = new nod;

p->info = ..... ; // cin >> p->info;

p->urm = NULL;

Ne imaginăm lista în felul următor; săgețile simbolizează legăturile dintre nodurile listei. Vârful săgeții reprezintă elementul următor. Ultimul element nu are săgeată. Valoarea corespunzătoare din câmpul urm este NULL.



În exemplul de mai sus au loc următoarele relații:

* valoarea pointerului prim este adresa elementului cu valoarea 12;
* prim->info==12
* prim->urm->info==46
* prim->urm este adresa elemenului cu valoarea 46
* prim->urm->urm==p
* p->info==59
* p->urm->urm==t
* t->info==17
* t->urm->info==25
* t->urm->urm->info==18
* t->urm->urm->urm==NULL
* t->urm->urm->urm->info nu există. Rezultatul acestei expresii este impredictibil!

Adăugarea unui element la finalul listei

Un antet posibil pentru funcția care adaugă un element la finalul liste ar putea fi:

void AdaugaFinal(nod \* & prim , int val);

Parametru prim este transmis prin referință pentru a trata corespunzător situația când lista este vidă. În acesta caz, valoare de intrare a lui prim este NULL, iar valoarea de ieșire este adresa primului element al listei – element nou creat.

Practic, vom trata două situații:

* dacă prim este NULL, creăm un nod nou, care va fi primul și totodată ultimul element al listei, memorăm în el valoarea dorită și prim devine adresa acestui nod;
* în caz contrar, identificăm ultimul nod al listei și nodul nou creat devine succesor al ultimului element și totodată ultimul element al listei.

void AdaugaFinal(nod \* & prim , int x)

{

// creăm nod nou

nod \* q = new nod;

q -> info = x;

q -> urm = NULL;

// adăugă noul nod la listă

if(prim == NULL)

{ // lista este vidă

prim = q;

}

else

{ // lista nu este vidă

nod \* t = prim;

while(t -> urm != NULL)

t = t -> urm;

t -> urm = q;

}

}

Adăugarea unui element la începutul listei

Un antet posibil pentru funcția care adaugă element la începutul liste ar putea fi:

void AdaugaInceput(nod \* & prim , int val);

Parametru prim este transmis prin referință deoarece la fiecare apel al funcției primul element se modifică; se creează un element nou care devine prim element al listei. Astfel, adresa primului element se modifică.

Procedăm astfel:

* prim memorează adresa primului element
* creăm un element nou: nod \* t = new nod;
* memorăm în el informația utilă: t->info = ....
* îl plasăm în listă înaintea primului element: t->urm = prim;
* elementul nou creat este reținut ca prim element al listei: prim = t;

Obs: Nu este necesară tratarea diferențiată a situațiilor când lista este vidă (prim==NULL), respectiv când lista conține elemente (prim memorează adresa primului element). În ambele situații atribuirea prim = t; are efectul dorit!

void AdaugaFinal(nod \* & prim , int x)

{

// creăm nod nou

nod \* t = new nod;

t -> info = x;

// legam nodul de lista

t -> urm = prim;

// valoarea lui prim se modifică, pentru a ieși din funcție cu valoarea corectă

prim = t;

}

Parcurgerea listei

Parcurgerea listei reprezintă vizitarea succesivă a elementelor pentru a realiza diverse operații cu valorile lor. Un antet posibil pentru o funcție care parcurge lista poate fi:

void Parcurgere(nod \* prim);

Parcurgerea se realizează secvențial, element cu element:

* folosim un pointer nod \* p în care vom memora, pe rând, adresele elementelor din listă;
* începem de la primul element al listei: p = prim;
* cât timp nu am trecut de ultimul element:
  + prelucrăm elementul curent (p->info)
  + trecem la următorul element: p = p->urm;

void Parcurgere(nod \* prim)

{

nod \* p = prim;

while(p != NULL)

{

//prelucrăm nodul curent

// trecem la următorul nod

p = p->urm;

}

}

Ștergerea unui element

Ștergerea unui element al listei constă în două etape: ștergerea propriu-zisă a variabilei dinamice în care este stoca nodul de șters și refacerea legăturilor, astfel încât lista să fie consistentă. Tehnic, modul de ștergere diferă după cum nodul de șters este primul din listă sau nu.

Dacă ștergem primul element al listei vom proceda astfel:

* memorăm adresa primului nod într-un pointer auxiliar: nod \* t = prim;
* nodul de după prim devine primul nod al listei: prim = prim->urm;
* ștergem variabila adresată de t: delete t;

Dac ștergem un element oarecare al listei, trebuie să cunoaștem într-un pointer oarecare, să spunem p, adresa elementului din fața nodului de șters. Acest lucru este necesar pentru refacerea corectă a legăturilor dintre elementele listei:

* vom șterge elementul situat în listă după cel cu adresa memorată în p, adică vom șterge p->urm;
* memorăm adresa nodului de șters înt-un pointer auxiliar: nod \* t = p->urm;
* corectăm adresa elementului de după p: p->urm = t->urm;
* ștergem variabila adresată de t: delete t;

Inserarea unui nou element

Și inserarea se face diferit, în funcție de poziția noului nod în listă; inserarea unui nod nou înaintea primului nod al listei (adresa sa este memorată în pointer-ul prim) se face astfel:

* creăm un nod nou: nod \* t = new nod;
* memorăm valoarea dorită în acest nod: t->info = ...;
* îl legăm de primul nod al listei: t->urm = prim;
* nodul nou creat devine primul din listă: prim = t;

Dacă nodul nou creat nu va fi primul din listă, îl vom insera după un nod cu adresa cunoscută, memorată în pointer-ul p:

* creăm un nod nou: nod \* t = new nod;
* memorăm valoarea dorită în acest nod: t->info = ...;
* în inserăm în listă:
  + nodul nou creat va fi înaintea nodului de după p: t->urm = p->urm;
  + nodul nou creat va fi plasat după nodul p: p-urm = t;

**Codul deplin al programului:**

#include <stdlib.h>

#include <stdio.h>

struct list{

    int data;

    struct list \*next;

};

void append(struct list \*\*head, int data){

    struct list \*temp = (struct list\*)malloc(sizeof(struct list));

    temp -> data = data;

    temp -> next = NULL;

    if(!(\*head)){

        \*head = temp;

        return;

    }

    struct list \*last = \*head;

    temp -> data = data;

    temp -> next = NULL;

    while(last -> next)

        last = last -> next;

    last -> next = temp;

}

void afisare(struct list \*head){

    while(head){

        printf("%d ", head -> data);

        head = head -> next;

    }

    printf("\n");

}

struct list\* numDiv(struct list \*head){

    struct list \*newList = NULL;

    struct list \*temp = head;

    while(temp){

        if(temp -> data % 8 == 0)

            append(&newList, temp -> data);

        temp = temp -> next;

    }

    temp = head;

    while(temp){

        if(temp -> data % 4 == 0)

            append(&newList, temp -> data);

        temp = temp -> next;

    }

    temp = head;

    while(temp){

        if(temp -> data % 2 == 0)

            append(&newList, temp -> data);

        temp = temp -> next;

    }

    temp = head;

    while(temp){

        if(temp -> data % 2 != 0 && temp -> data < 0)

            append(&newList, temp -> data);

        temp = temp -> next;

    }

    return newList;

}

int numNode(struct list \*head){

    int n = 0;

    if(!head)

        return 0;

    while(head){

        n++;

        head = head -> next;

    }

    return n;

}

int main(){

    struct list \*head = NULL;

    for(int i = -20; i < 20; i++)

        append(&head,i);

    printf("Lista initiala:\n");

    afisare(head);

    struct list \*newList = numDiv(head);

    printf("\nLista noua:\n");

    afisare(newList);

    struct list \*temp = newList, \*l1 = NULL, \*l2 = NULL, \*l3 = NULL;

    int node = numNode(head) / 3, i = 1;

    while(temp){

        if(i <= node)

            append(&l1, temp -> data);

        if(node < i && i <= 2 \* node)

            append(&l2, temp -> data);

        if(2 \* node < i && i <= numNode(head))

            append(&l3, temp -> data);

        i++;

        temp = temp -> next;

    }

    printf("\nl1) ");

    afisare(l1);

    printf("l2) ");

    afisare(l2);

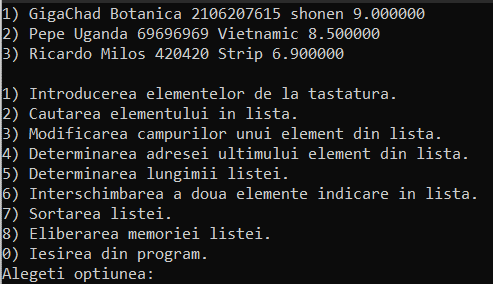
    printf("l3) ");

    afisare(l3);

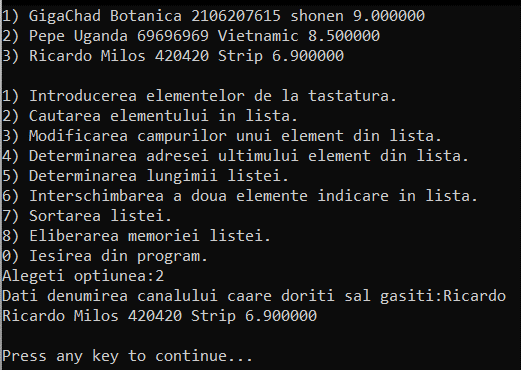
    return 0;

}

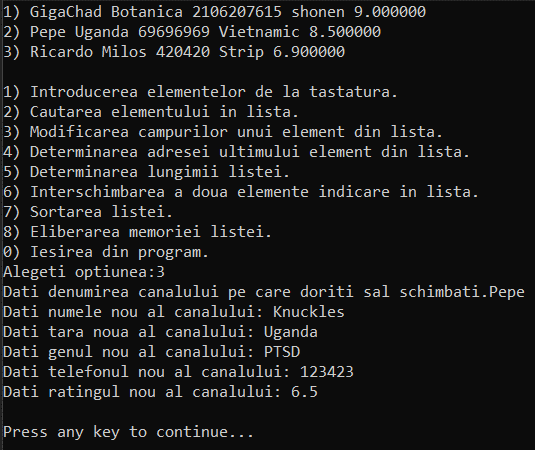
**Exemple de execuție a programului:**



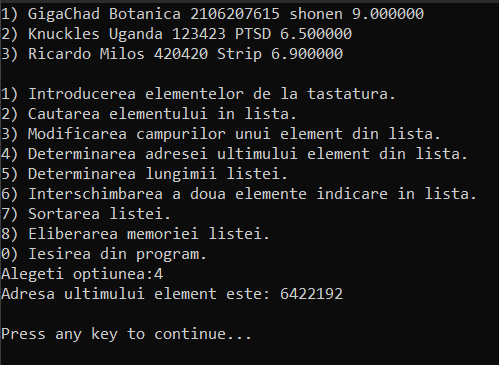
Cautarea elementului din listă:



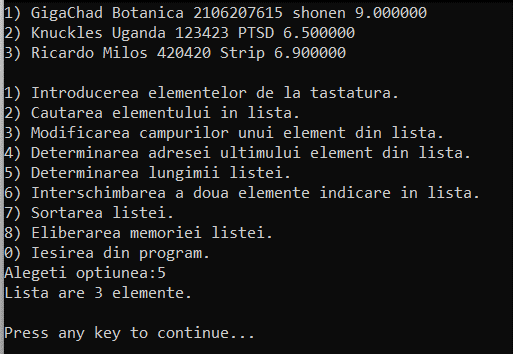
Modificarea câmpului unui element:



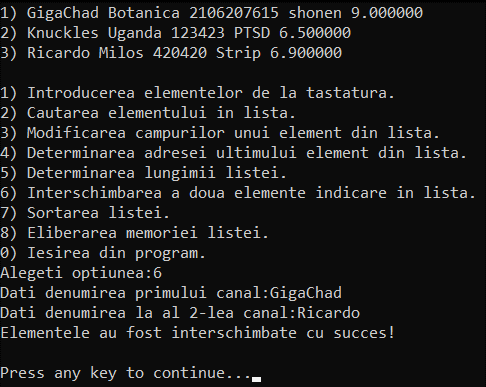
Adresa ultimului element:



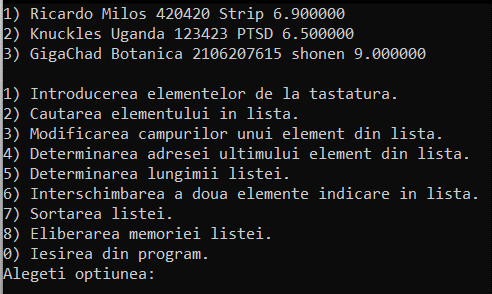
Determinarea lungimii listei:



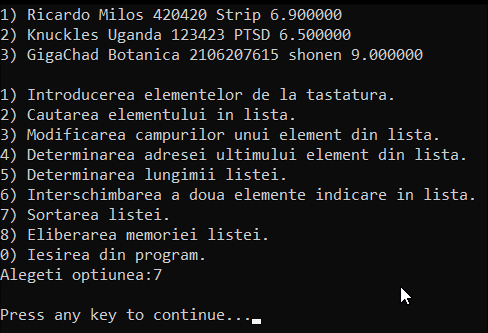
Interschimbarea a doua elemente:

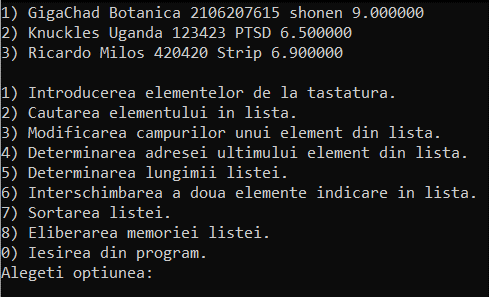


Aici vedem ca sau schimbat cu succes:

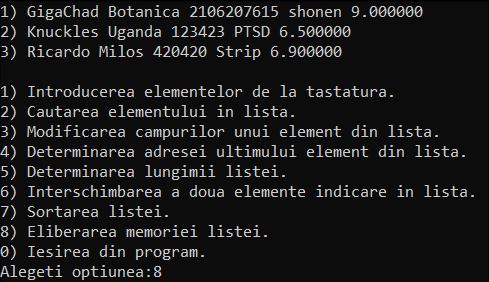


Sortarea listei:





Eliberarea memoriei:



**Concluzii pe baza efectuării lucrării de laborator:**

* Deci am lucrat cu lista simplu lănțuită și am îndeplinit sarcinele similare cu laboratoarele trecute când am lucrat cu structurile.
* Lucrul asupra lucrării este util în întelegerea mai profundă a lucrului cu listele simplu lănțuite și a operării mai efectivă cu informația.

**Surse**

<https://www.pbinfo.ro/articole/19576/liste-liniare-simplu-inlantuite-alocate-dinamic>