INFORMATICĂ · CLASA A XI-A · STRUCTURI DE DATE (INTENSIV) · LECȚIA 18

Pointeri, alocare dinamică și liste simplu înlănțuite

Durată: 60 min Nivel: intensiv informatică Înainte: lecțiile x-12, xi-02

CE VEI ȘTI DUPĂ LECȚIE

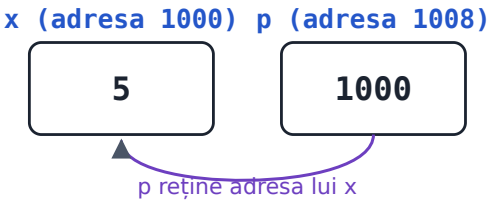
- Înțelegi ce este un pointer: o variabilă care memorează o adresă.
- Aloci și eliberezi memorie dinamic cu new și delete.
- Construiești o listă simplu înlănțuită: adăugare la început / la sfârșit, parcurgere, ștergere, inserare.

1. De ce avem nevoie de pointeri

Până acum, memoria programului a fost alcătuită din variabile cu nume (`x` , `v[100]`). Uneori nu știi dinainte **câte** date vor veni: un vector `v[100]` e prea mic sau prea mare. Soluția: ceri memorie **în timpul rulării**, exact cât ai nevoie. Pentru a ajunge la această memorie, folosești **adrese**, iar o variabilă care păstrează o adresă se numește **pointer**.

2. Pointeri: adresă și valoare

Fiecare variabilă stă la o **adresă** în memorie, ca o casă la o stradă. Un pointer este un „bilet cu adresa casei".



p nu conține 5, ci adresa unde se află 5 · *p este valoarea de la adresa din p

Pointer

Scriere	Se citește	Exemplu
`int *p;`	p este pointer spre int	
`&x`	adresa lui x	`p = &x;`
`*p`	valoarea de la adresa din p	`cout << *p;` → 5
`*p = 10;`	schimbă valoarea de la adresă	acum `x == 10`
`NULL` (sau `nullptr`)	„nu arată nicăieri"	`p = NULL;`

Prima utilizare practică: o funcție care modifică argumentele (cum făcea `&` în lecția XI.2).

```cpp
#include <iostream>
using namespace std;

void schimba(int *a, int *b)
{
    int t = *a;       // valoarea de la adresa a
    *a = *b;
    *b = t;
}

int main()
{
    int x, y;
    cin >> x >> y;
    schimba(&x, &y);  // dăm adresele lui x și y
    cout << x << ' ' << y;
    return 0;
}
```

DE CE AȘA?

De ce `&x` la apel? Funcția are nevoie de **adresa** variabilelor, ca să poată modifica originalele. Dacă am transmite `x` prin valoare, `schimba` ar lucra pe o copie și `x`, `y` ar rămâne neschimbate (exact ca la parametri prin valoare, lecția XI.2).

3. Alocare dinamică: `new` și `delete`

```cpp
int *q = new int;       // o variabilă nouă (fără nume), accesibilă doar prin q
*q = 7;
delete q;               // o eliberăm când nu mai e necesară

int *v = new int[n];    // un vector de n elemente, n aflat la rulare
delete[] v;             // eliberare pentru vectori
```

```cpp
#include <iostream>
using namespace std;

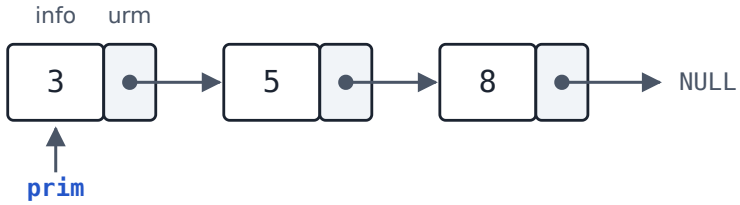
int main()
{
    int n;
    cin >> n;
    int *v = new int[n];               // exact n elemente
    int suma = 0;
    for (int i = 0; i < n; i++)
    {
        cin >> v[i];
        suma += v[i];
    }
    cout << suma;
    delete[] v;                        // dăm memoria înapoi
    return 0;
}
```

DE CE AȘA?

De ce `delete[]` ? Memoria cerută cu `new` nu se eliberează singură cât timp programul rulează; dacă uiți `delete`, memoria „se scurge” (memory leak). La programele mici nu se vede, la servere care rulează luni de zile devine problemă gravă. **De ce `v[i]` merge pe un pointer?** `v[i]` înseamnă „elementul de la adresa v + i”; vectorii și pointerii sunt strâns legați.

4. Lista simplu înlănțuită

O **listă** este o înșiruire de **noduri**. Fiecare nod conține **informația** și **adresa nodului următor**. Ultimul nod are `urm = NULL`. Ținem minte doar adresa primului nod (`prim`); de acolo ajungem la toate celelalte.



fiecare nod: info (valoarea) + urm (adresa nodului următor); ultimul: urm = NULL

Lista

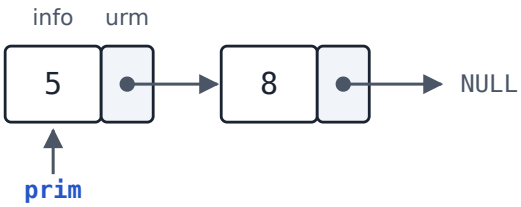
```
struct nod
{
    int info;       // valoarea
    nod *urm;       // adresa nodului următor
};
nod *prim = NULL;   // lista goală
```

Pe un exemplu cu adrese inventate:

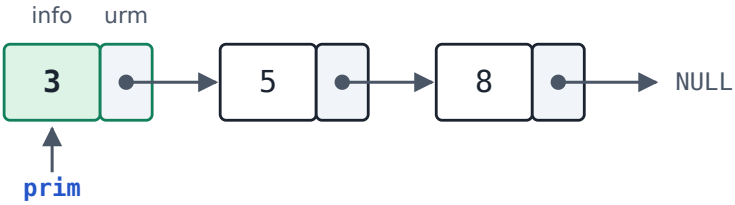
Nod	Adresa	info	urm
#1	100	3	120
#2	120	5	140
#3	140	8	NULL

`prim` conține 100. Dacă `p = prim`, atunci `p->info` este 3 (prescurtare pentru `(*p).info`), iar `p = p->urm` mută p pe nodul 2.

Adăugare la început



înainte



după adaugaInceput(3): nodul nou devine primul

Adăugare la început

```
void adaugaInceput(nod *&prim, int x)
{
    nod *p = new nod;
    p->info = x;
    p->urm = prim;      // nodul nou arată spre vechiul prim
    prim = p;           // nodul nou devine primul
}
```

Adăugare la sfârșit și parcurgere

Pentru adăugarea la sfârșit ținem și un pointer `ultim`.

```cpp
#include <iostream>
using namespace std;

struct nod
{
    int info;
    nod *urm;
};

void adaugaInceput(nod *&prim, int x)
{
    nod *p = new nod;
    p->info = x;
    p->urm = prim;
    prim = p;
}

void adaugaSfarsit(nod *&prim, nod *&ultim, int x)
{
    nod *p = new nod;
    p->info = x;
    p->urm = NULL;
    if (prim == NULL) prim = ultim = p;      // prima valoare
    else { ultim->urm = p; ultim = p; }
}

void afisare(nod *prim)
{
    for (nod *p = prim; p != NULL; p = p->urm)
        cout << p->info << ' ';
    cout << '\n';
}

void elibereaza(nod *&prim)
{
    while (prim != NULL)
    {
        nod *q = prim;
        prim = prim->urm;
        delete q;
    }
}

int main()
{
    int n, x;
    cin >> n;
    nod *L1 = NULL, *ultim = NULL, *L2 = NULL;
    for (int i = 1; i <= n; i++)
    {
        cin >> x;
        adaugaSfarsit(L1, ultim, x);
        adaugaInceput(L2, x);
    }
    afisare(L1);
    afisare(L2);
    elibereaza(L1);
```

```
    elibereaza(L2);
    return 0;
}
```

DE CE AȘA?

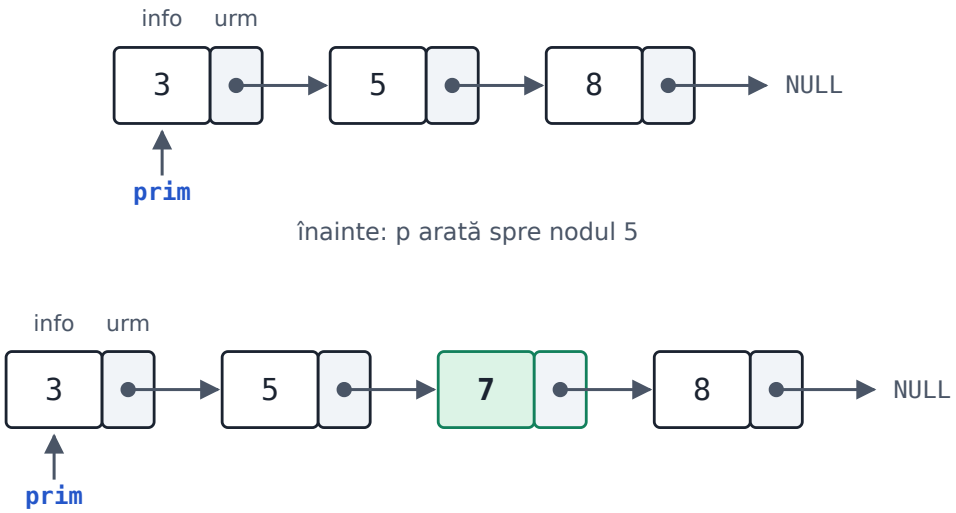
De ce `nod *&prim` (pointer prin referință)? Funcția trebuie să poată schimba pointerul `prim` din `main` (când lista era goală, `prim` devine nodul nou). Fără `&`, funcția ar modifica o copie a pointerului și lista din `main` ar rămâne goală. **De ce `p = p->urm` în `for`?** Este „pasul" pe listă: p trece pe nodul următor; când ajunge la `NULL`, am terminat. **De ce `elibereaza` salvează `q` înainte de `prim = prim->urm`?** După `delete q` nodul dispare; trebuie să știm unde era următorul.

DE ȘTIUT

Prima listă (`L1`) păstrează ordinea citirii (adăugare la sfârșit), a doua (`L2`) o inversează (adăugare la început): același cod pentru stivă, deoarece o stivă implementată cu listă adaugă și scoate la început.

5. Inserare și ștergere: ordinea contează

Inserarea unui nod după p



Inserare

```
nod *q = new nod;
q->info = x;
q->urm = p->urm;    // 1) nodul nou arată spre restul listei
p->urm = q;         // 2) p arată spre nodul nou
```

Pe adresele din tabel, inserăm 7 după nodul 5 (p = 120, q alocat la 160):

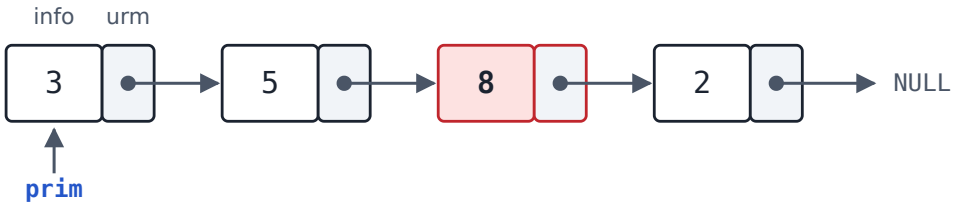
Pas	Instrucțiune	Efect
1	`q->info = 7`	nodul 160: info = 7

Pas	Instrucțiune	Efect
2	`q->urm = p->urm`	nodul 160: urm = 140 (continuă cu 8)
3	`p->urm = q`	nodul 120: urm = 160

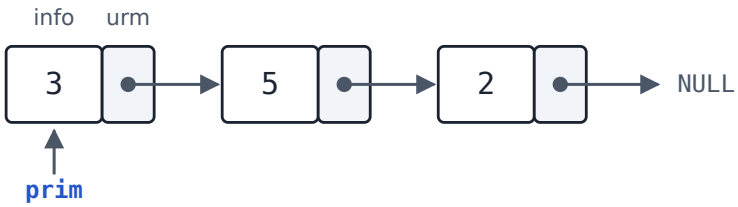
ATENȚIE — GREȘELI FRECVENTE

Dacă faci întâi `p->urm = q`, pierzi adresa 140 (nodul 8): nu mai există nicio variabilă care să o rețină, iar restul listei devine inaccesibil.

Ștergerea nodului de după p



înainte: se șterge nodul 8 (cel de după p, adică de după 5)



după: p->urm = q->urm; delete q;

Ștergere

```
nod *q = p->urm;    // nodul de șters
p->urm = q->urm;    // îl ocolim
delete q;           // eliberăm memoria
```

Ștergerea primei apariții a valorii x

Trei cazuri: lista goală, ștergi chiar primul nod, ștergi din interior.

```cpp
#include <iostream>
using namespace std;

struct nod { int info; nod *urm; };

void adaugaSfarsit(nod *&prim, nod *&ultim, int x)
{
    nod *p = new nod;
    p->info = x; p->urm = NULL;
    if (prim == NULL) prim = ultim = p;
    else { ultim->urm = p; ultim = p; }
}

void stergeValoare(nod *&prim, int x)
{
    if (prim == NULL) return;
    if (prim->info == x)                    // cazul 1: primul nod
    {
        nod *q = prim;
        prim = prim->urm;
        delete q;
        return;
    }
    for (nod *p = prim; p->urm != NULL; p = p->urm)
        if (p->urm->info == x)              // cazul 2: nodul de după p
        {
            nod *q = p->urm;
            p->urm = q->urm;
            delete q;
            return;
        }
}

int main()
{
    int n, x, v;
    cin >> n;
    nod *prim = NULL, *ultim = NULL;
    for (int i = 1; i <= n; i++) { cin >> x; adaugaSfarsit(prim, ultim, x); }
    cin >> v;
    stergeValoare(prim, v);
    for (nod *p = prim; p != NULL; p = p->urm) cout << p->info << ' ';
    return 0;
}
```

DE CE AȘA?

De ce primul nod e caz aparte? Pentru a șterge un nod trebuie să modifici `urm` al **predecesorului**; primul nod nu are predecesor, deci modifici `prim`. **De ce `p->urm != NULL` în buclă?** Ne oprim la penultimul nod ca să putem privi `p->urm->info` fără să accesăm `NULL`.

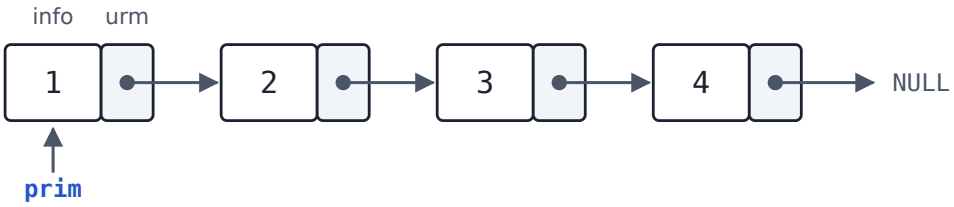
Inserare după valoarea y

```cpp
#include <iostream>
using namespace std;

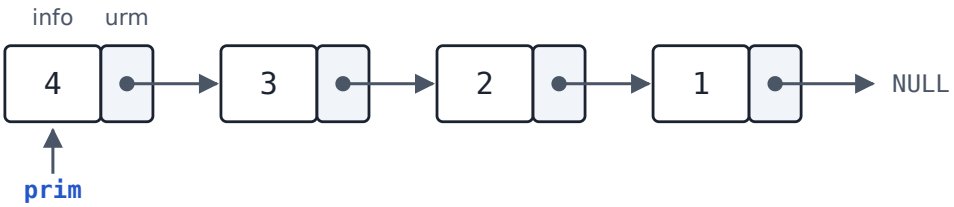
struct nod { int info; nod *urm; };

int main()
{
    int n, x, y, v;
    cin >> n;
    nod *prim = NULL, *ultim = NULL;
    for (int i = 1; i <= n; i++)
    {
        cin >> x;
        nod *p = new nod;
        p->info = x; p->urm = NULL;
        if (prim == NULL) prim = ultim = p;
        else { ultim->urm = p; ultim = p; }
    }
    cin >> y >> v;                          // inserăm v după prima apariție a lui y
    for (nod *p = prim; p != NULL; p = p->urm)
        if (p->info == y)
        {
            nod *q = new nod;
            q->info = v;
            q->urm = p->urm;
            p->urm = q;
            break;
        }
    for (nod *p = prim; p != NULL; p = p->urm) cout << p->info << ' ';
    return 0;
}
```

6. Inversarea listei



înainte



după inversare: fiecare legătură este întoarsă, prim arată spre fostul ultim

Inversare

Trei pointeri: `anterior` (inițial `NULL`), `curent` (pornește de la `prim`), `urmator`. La fiecare pas: reținem următorul, întoarcem legătura curentă, avansăm.

Iterația	curent	urmator	`curent->urm` devine	anterior după pas
1	1	2	NULL	1
2	2	3	1	2
3	3	4	2	3
4	4	NULL	3	4

La final `prim = anterior` (nodul 4).

```
#include <iostream>
using namespace std;

struct nod { int info; nod *urm; };

int main()
{
    int n, x;
    cin >> n;
    nod *prim = NULL, *ultim = NULL;
    for (int i = 1; i <= n; i++)
    {
        cin >> x;
        nod *p = new nod;
        p->info = x; p->urm = NULL;
        if (prim == NULL) prim = ultim = p;
        else { ultim->urm = p; ultim = p; }
    }
    nod *anterior = NULL, *curent = prim;
    while (curent != NULL)
    {
        nod *urmator = curent->urm;     // 1) reținem restul
        curent->urm = anterior;         // 2) întoarcem legătura
        anterior = curent;              // 3) avansăm
        curent = urmator;
    }
    prim = anterior;
    for (nod *p = prim; p != NULL; p = p->urm) cout << p->info << ' ';
    return 0;
}
```

DE CE AȘA?

De ce reținem `urmator` înainte să schimbăm `curent->urm`? După schimbare, legătura spre restul listei dispare; fără `urmator` am pierde restul (aceeași idee ca la inserare).

7. Vector sau listă?

Operație	Vector	Listă
acces la elementul k	O(1)	O(k) (parcurgi de la început)
inserare / ștergere în mijloc, având pointerul	O(n) (deplasări)	O(1)
adăugare la sfârșit	O(1)	O(1) cu pointer `ultim`
memorie	fixă, declarată dinainte	exact cât e nevoie, plus câte un pointer pe nod
căutare	O(n)	O(n)

8. Greșeli frecvente

Greșeala	Efect	Corect
uiți `NULL` la ultimul nod	parcurgerea nu se oprește	`p->urm = NULL` la creare
`p->urm = q` înainte de `q->urm = p->urm`	pierzi restul listei	întâi q, apoi p
`delete q` înainte să salvezi `q->urm`	folosești memorie eliberată	salvezi întâi
funcție care schimbă `prim` fără `*&`	lista din main nu se schimbă	`nod *&prim`
acces `p->info` cu `p == NULL`	blocare (segmentation fault)	verifici `p != NULL`
uiți `delete`	scurgere de memorie	eliberezi ce ai alocat

9. La BAC

LA BAC

Grilele cu liste (clasa a X-a / a XI-a) dau o listă cu valori și o secvență de instrucțiuni cu pointeri („`p = prim->urm; p->urm = p->urm->urm;`") și întreabă ce se afișează sau ce valoare are un nod. Metoda sigură: desenează lista, notează numele pointerilor sub nodurile pe care arată și execută instrucțiunile una câte una, mutând săgețile.

10. Exersează pe hârtie

1. Desenează lista după operațiile: adaugaInceput 3, adaugaInceput 5, adaugaSfarsit 8, ștergerea primului nod. Ce valori conține?

2. Lista este 2 → 4 → 6 → 8. Ce se afișează după `p = prim; p = p->urm; p->urm = p->urm->urm;` urmată de parcurgere?

3. Care este valoarea lui `p->info` după `p = prim->urm->urm;` pe lista 7 → 1 → 9 → 4?

4. Scrie o funcție care numără nodurile cu informație pară.

5. Scrie o funcție care returnează adresa ultimului nod (sau `NULL` pentru lista goală).

6. Cum ștergi ultimul nod al unei liste? (Atenție la lista cu un singur nod.)

7. Scrie o funcție care inserează x într-o listă sortată crescător, păstrând ordinea.

8. (BAC) Pe lista 1 → 2 → 3 → 4 → 5 se execută `q = prim; while (q->urm->urm != NULL) q = q->urm;`. Pe ce nod arată q la final?

9. De ce nu poți accesa direct elementul k într-o listă ca într-un vector? Câți pași faci pentru elementul 100?

10. Dă un exemplu de problemă în care lista este mai bună decât vectorul și unul în care vectorul este mai bun.

11. Mai departe (pentru cine vrea mai mult)

Lista dublu înlănțuită. Fiecare nod are și `pred` (nodul anterior): poți merge în ambele sensuri și ștergi un nod fără să cauți predecesorul. STL are `std::list` gata făcută.

Lista circulară. Ultimul nod arată spre primul; folosită la „elevii în cerc" (Josephus) fără copiere.

Detectarea ciclului (Floyd, „broasca și iepurele"). Doi pointeri, unul avansează 1 pas, altul 2; dacă se întâlnesc, lista are ciclu.

Stiva și coada cu liste. Stiva: adaugi și scoți la început; coada: adaugi la sfârșit (`ultim`) și scoți de la început. Merge mai bine decât vectorul când nu știi câte elemente vor fi.

Pointeri inteligenți (C++ modern). `unique_ptr` și `shared_ptr` eliberează memoria automat; în programele reale se evită `new` / `delete` direct.

Rezumat

- `&x` = adresa, `*p` = valoarea de la adresă, `p->camp` = câmpul nodului, `NULL` = fără adresă.
- `new` aloci, `delete` eliberezi; `nod *&prim` când funcția trebuie să schimbe pointerul.
- Lista: noduri `{info, urm}`, `prim`, uneori `ultim`; parcurgere: `for (p = prim; p != NULL; p = p->urm)`.
- La inserare: întâi `q->urm = p->urm`, apoi `p->urm = q`. La ștergere: salvezi, ocolești, `delete`.

Probleme de lucru (pbinfo)

Începe cu acestea, în ordine: sunt alese să urmeze pașii din lecție.

- [ ] #1168 **fadaugarelalista** — ușoară
  https://www.pbinfo.ro/probleme/1168/fadaugarelalista
- [ ] #1169 **fafisarelista** — ușoară
  https://www.pbinfo.ro/probleme/1169/fafisarelista

- [ ] #1170 **fadaugareinaintelista** — ușoară
  https://www.pbinfo.ro/probleme/1170/fadaugareinaintelista

Temă (pbinfo)

Trimite soluțiile pe pbinfo și notează punctajul obținut. Bifează ce ai terminat.

- [ ] #1171 **fstergeprimuldinlista** — ușoară
  https://www.pbinfo.ro/probleme/1171/fstergeprimuldinlista
- [ ] #1172 **flistanumarare** — ușoară
  https://www.pbinfo.ro/probleme/1172/flistanumarare
- [ ] #1176 **flistastergere** — ușoară
  https://www.pbinfo.ro/probleme/1176/flistastergere
- [ ] #1179 **flistainserare** — ușoară
  https://www.pbinfo.ro/probleme/1179/flistainserare
- [ ] #1476 **fsortare** — ușoară
  https://www.pbinfo.ro/probleme/1476/fsortare
- [ ] #3789 **flistaoglindire** — ușoară
  https://www.pbinfo.ro/probleme/3789/flistaoglindire

Pentru cei care vor mai mult

Probleme mai grele sau de concurs, pentru cine vrea să meargă dincolo de BAC.

- [ ] #3743 **fadaugareinceputlldi** — ușoară
  Listă dublu înlănțuită.
  https://www.pbinfo.ro/probleme/3743/fadaugareinceputlldi

Grile din subiectele de BAC (pbinfo)

Exerciții de tip grilă din variantele de Bacalaureat, rezolvate online pe pbinfo, cu verificare automată a răspunsului.

- [ ] #71 **Exercițiul #71**
  listă simplu înlănțuită: ce rămâne după o secvență de instrucțiuni cu pointeri
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=71
- [ ] #75 **Exercițiul #75**
  câmpul info al nodurilor: efectul unei secvențe pe listă
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=75
- [ ] #92 **Exercițiul #92**
  listă cu numere întregi: ce afișează o parcurgere
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=92

- [ ] #114 **Exercițiul #114**
  listă circulară: ce înseamnă parcurgerea
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=114

- [ ] #166 **Exercițiul #166**
  valori date în ordine: ce se afișează după ștergere / inserare
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=166

- [ ] #187 **Exercițiul #187**
  listă cu cel puțin 4 noduri: efectul operațiilor asupra legăturilor
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=187

- [ ] #196 **Exercițiul #196**
  câmpul urm: ce se afișează la parcurgere
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=196

- [ ] #287 **Exercițiul #287**
  listă dublu înlănțuită: câmpurile st și dr
  https://www.pbinfo.ro/?pagina=itemi-evaluare&id=287

Citește și exersează în plus

- Pointeri. Operații cu pointeri https://www.pbinfo.ro/articole/7659/pointeri-operatii-cu-pointeri
- Mecanismul alocării dinamice https://www.pbinfo.ro/articole/7662/mecanismul-alocarii-dinamice
- Liste liniare simplu înlănțuite alocate dinamic https://www.pbinfo.ro/articole/19576/liste-liniare-simplu-inlantuite-alocate-dinamic
- VisuAlgo — Linked List — inserări și ștergeri pe liste, animate pas cu pas (engleză)
  https://visualgo.net/en/list
- learncpp.com — Pointers — pointeri explicați de la zero, cu imagini (engleză)
  https://www.learncpp.com/cpp-tutorial/introduction-to-pointers/
- cppreference — std::list — lista dublu înlănțuită din STL, gata făcută (engleză)
  https://en.cppreference.com/w/cpp/container/list

Răspunsuri

Exersează pe hârtie

1. adaugaInceput 3 → 3; adaugaInceput 5 → 5, 3; adaugaSfarsit 8 → 5, 3, 8; ștergerea primului → **3, 8**.

2. `p = prim` (nodul 2); `p = p->urm` (nodul 4); `p->urm = p->urm->urm` (4 arată acum spre 8, nodul 6 este ocolit). Se afișează **2 4 8**.

3. `prim->urm` este nodul 1, `prim->urm->urm` este nodul 9: `p->info = 9`.

4. `int nrPare(nod *prim) { int c = 0; for (nod *p = prim; p != NULL; p = p->urm) if (p->info % 2 == 0) c++; return c; }`

5. `nod *ultimNod(nod *prim) { if (prim == NULL) return NULL; nod *p = prim; while (p->urm != NULL) p = p->urm; return p; }`

6. Dacă lista are un singur nod: `delete prim; prim = NULL;`. Altfel, cauți penultimul nod (`p->urm->urm == NULL`), faci `delete p->urm; p->urm = NULL;`.

7. Dacă lista e goală sau `x <= prim->info`, adaugi la început. Altfel avansezi p cât timp `p->urm != NULL && p->urm->info < x`, apoi inserezi după p.

8. Condiția `q->urm->urm != NULL` oprește bucla când după q mai este un singur nod. Pe lista cu 5 noduri, q ajunge pe nodul **4** (penultimul).

9. Nodurile nu sunt consecutive în memorie; ajungi la elementul k doar urmând legăturile, deci pentru al 100-lea faci 99 de pași (`p = p->urm` de 99 ori).

10. Lista e mai bună: o coadă de așteptare la care se adaugă și se elimină mereu la capete, număr necunoscut de elemente. Vectorul e mai bun: o listă de note la care accesezi des elementul k (acces direct, O(1)).