Tehnici de programare

analiza algoritmilor; liste simplu înlănțuite

Algoritmii pot fi analizați din multe puncte de vedere:

- după criterii de performanță, cum sunt timpul de execuție sau memoria necesară, incluzând cazuri particulare, de exemplu cazul cel mai defavorabil, cazul cel mai des întâlnit în practică sau cazul cel mai favorabil
- după facilitățile suplimentare pe care algoritmul le furnizează. Exemple: indiferent de operațiile asupra ei, o colecție rămâne tot timpul sortată; pointerii la elementele unei colecții se pot folosi și după operații gen adăugare sau ștergere.
- ținând cont de portabilitatea unei implementări: se utilizează doar facilități standard ale limbajului de programare sau sunt necesare instrucțiuni sau funcții care sunt disponibile doar pe un anumit compilator, platformă hardware sau sistem de operare
- după necesități administrative: unii algoritmi sunt mai simplu de implementat și testat, aspect important când există termene limită strânse care trebuie respectate

O notație des întâlnită pentru performanța unui algoritm este **O(expr)**, unde *expr* este o funcție, în general exprimată în funcție de variabila **n**. Notația *O(...)* (en: big O notation) reprezintă numărul de operații efectuate în cazul cel mai defavorabil, în funcție de numărul *n* de elemente de intrare.

Fie un vector cu dimensiune fixă (fără redimensionare dinamică), și *n* numărul curent de elemente din el. Câteva exemple despre cum se calculează *O(...)* pentru diverse operații cu acest vector:

- *adăugarea unui element* - **O(ct)**, deoarece indiferent de dimensiunea vectorului, este nevoie de un număr constant de operații (ex: *v[n++]=e;*)
- *ștergerea unui element* - **O(n)**, deoarece în cazul cel mai defavorabil (ștergerea primului element din vector), trebuie să mutăm toate celelalte *n* elemente la stânga cu o poziție. Strict vorbind, este nevoie de *n-1* operații, dar la limită, când *n* tinde la infinit, -1 devine nesemnificativ.
- *inserarea unul element* - **O(n)**, deoarece în cazul cel mai defavorabil (inserarea pe prima poziție din vector), trebuie să mutăm toate celelalte *n* elemente la dreapta cu o poziție.
- *căutarea unui element* - **O(n)**, deoarece în cazul cel mai defavorabil (elementul căutat nu este în vector), trebuie parcurse toate elementele vectorului

În caz că vectorul se redimensionează în funcție de numărul elementelor, putem avea mai multe implementări. În cea mai simplă implementare, vectorul crește cu câte un element la fiecare adăugare. În această implementare, operația de adăugare are complexitatea *O(n)*, deoarece în cazul cel mai defavorabil, *realloc* trebuie să copieze toată zona anterioară de memorie într-o nouă zonă mai mare, deci avem nevoie de *n* operații.

Pentru a se elimina necesitatea unei realocări de memorie la fiecare adăugare de element în vector, în general se optează pentru următoarea soluție: de fiecare dată când nu mai este loc în vector, dimensiunea acestuia crește la dublul dimensiunii anterioare. Astfel dimensiunea vectorului va fi 0, 1, 2, 4, 8, 16, ….

În această situație, operația de adăugare în cazul cel mai defavorabil (vector plin) va avea complexitatea *O(n)* (deoarece este nevoie de realocare), dar, cu cât *n* crește, cu atât cazul cel mai defavorabil devine din ce în ce mai rar. De exemplu, când *n* crește de la 1024 la 2048, deci avem 1023 poziții libere, la toate aceste poziții libere adăugarea va fi de complexitate *O(ct)*, deoarece nu este nevoie de realocare. Abia la a 1024-a adăugare avem complexitate *O(n)*, deoarece atunci vectorul va fi din nou realocat.

Pentru acest gen de comportament, în care cazul cel mai defavorabil este garantat că devine din ce în ce mai rar, tinzând la 0, se utilizează notația **O(...)+** (**+** se citește **amortizat**) și pentru complexitate se consideră cazul a cărui probabilitate tinde la infinit.
Pentru exemplul cu vectorul, complexitatea adăugării va fi **O(ct)+** (*complexitate constantă amortizată*), deoarece, lăsând la o parte operația de realocare, adăugarea unui element în vector va avea complexitatea *O(ct)*.

În tabelul următor sunt date diverse complexități, cu exemple de algoritmi și cum crește numărul de operații necesare dacă n crește de la 10 la 100:

Tip algoritm	**Exemple de algoritmi**	**n=10**	**n=100**
Constant O(ct)	- adăugare de elemente în vector fix - calcul valoare absolută	1	1
Logaritmic O($\log_2(n)$)	- căutare binară într-un vector	3.3	6.6
Linear O(n)	- căutare sau ștergere într-un vector	10	100
Linearitmic O($n*\log_2(n)$)	- quicksort	33.2	664
Pătratic O(n^2)	- bubblesort	100	10000
Exponențial O(2^n)	- problema comis-voiajorului	1024	10^{30}

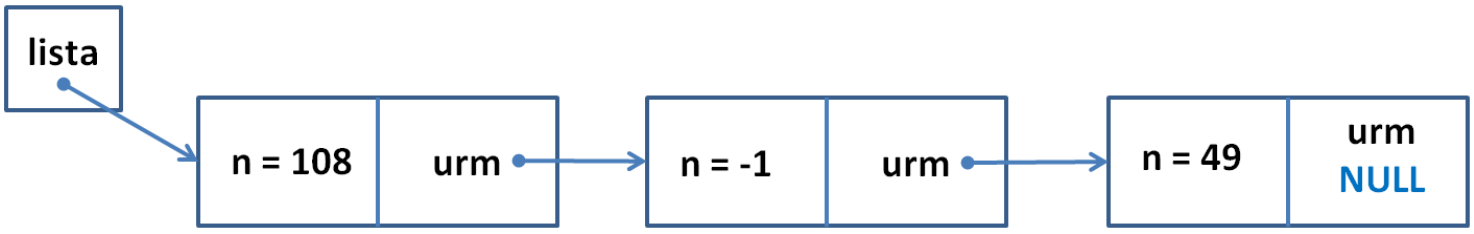
Pentru a ne da seama de diferențele dintre aceste tipuri de complexități, să presupunem că avem de sortat un vector de 1,000,000 elemente. Folosind *quicksort*, am avea de efectuat ~20,000,000 de operații. Dacă un calculator execută 1,000,000 de asemenea operații/secundă, înseamnă că vom avea nevoie de 20 de secunde pentru sortare. Dacă am fi folosit *bubblesort*, am avea de efectuat 1,000,000,000,000 operații, deci va fi nevoie de aproape 278 ore.

Liste simplu înlănțuite

O listă este o structură simplă de date și există multe posibilități de implementare a ei. În continuare vom discuta unele dintre aceste implementări și le vom compara atât între ele, cât și cu vectorii, pentru a evidenția avantajele și dezavantajele lor principale.

O listă este formată dintr-o succesiune de elemente alocate distinct, fiecare element având pe lângă informația utilă și un pointer la următorul element din listă. Acest pointer la următorul element se numește **urm**(ător, en: next). Pointerul *urm* al ultimului element din listă va fi *NULL*, pentru a marca faptul că nu pointează la nimic. Toată lista va fi accesibilă prin intermediul unui pointer la primul element din listă.

De exemplu, vom considera că un element din listă are ca formație utilă un număr întreg *n*. Lista are 3 elemente {108, -1, 49} și este pointată de un pointer numit *lista*. Structura memoriei arată astfel:



Exemplul 1: Să se scrie un program care creează lista din figura de mai sus și îi afișează conținutul:

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

```c
// un element al listei
typedef struct elem{
    int n;                      // informatia utila
    struct elem *urm;       // camp de inlantuire catre urmatorul element
    }elem;

// aloca un nou element si ii seteaza campurile corespunzatoare
elem *nou(int n,elem *urm)
{
    elem *e=(elem*)malloc(sizeof(elem));
    if(!e){
        printf("memorie insuficienta");
        exit(EXIT_FAILURE);
        }
    e->n=n;
    e->urm=urm;
    return e;
}

// adauga un element la inceputul listei
// returneaza noua lista
elem *adaugaInceput(elem *lista,int n)
{
    return nou(n,lista);
}

// adauga un element la sfarsitul listei
// returneaza noua lista
elem *adaugaSfarsit(elem *lista,int n)
{
    if(!lista)return nou(n,NULL);
    elem *p=lista,*urm;
    for(urm=p->urm;urm;p=urm,urm=p->urm){}
    p->urm=nou(n,NULL);
    return lista;
}

void afisare(elem *lista)
{
    for(;lista;lista=lista->urm){
        printf("%d ",lista->n);
        }
    putchar('\n');
}

// elibereaza memoria ocupata de o lista
void eliberare(elem *lista)
{
    elem *p;
    while(lista){
        p=lista->urm;
        free(lista);
        lista=p;
        }
}
```

```
int main()
{
   // varianta 1: construieste direct lista dorita
   elem *lista1=nou(108,nou(-1,nou(49,NULL)));
   afisare(lista1);
   eliberare(lista1);
   // varianta 2: adauga la inceputul listei elementele in ordinea inversa
   elem *lista2=NULL;
   lista2=adaugaInceput(lista2,49);
   lista2=adaugaInceput(lista2,-1);
   lista2=adaugaInceput(lista2,108);
   afisare(lista2);
   eliberare(lista2);
   // varianta 3: adauga la sfarsitul listei
   elem *lista3=NULL;
   lista3=adaugaSfarsit(lista3,108);
   lista3=adaugaSfarsit(lista3,-1);
   lista3=adaugaSfarsit(lista3,49);
   afisare(lista3);
   eliberare(lista3);
   return 0;
}
```

Un element al listei a fost implementat prin structura *elem*. Un aspect interesant în definirea acestei structuri este următorul: numele *"elem"* este simultan atât numele unei structuri (*struct elem*) cât și un nume introdus de *typedef*.

Limbajul C consideră că numele structurilor fac parte din alt spațiu de nume decât cel pentru variabile, funcții sau *typedef*. Din acest motiv, putem să avem de exemplu definiții de forma *"struct persoana persoana;"*. Deși *persoana* identifică atât numele unei structuri cât și al unei variabile, din cauză ele fac parte din spații diferite de nume, ele nu se consideră ca fiind definire multiplă. Din acest punct de vedere, o construcție de forma *"typedef struct nume{...}nume;"* este echivalentă cu *"struct nume{...}; typedef struct nume nume;"*.

În exemplu a fost nevoie de acest gen de declarare din cauză că în interiorul structurii (deci înainte de numele definit de *typedef*) s-a folosit *"struct elem *urm;"*. Astfel, a trebuit să dăm un nume structurii, pentru a defini câmpul *urm* și, pentru a nu mai introduce alt nume, am folosit același nume ca și cel pentru *typedef*.

Funcția *nou* creează un nou element. Complexitatea acestei funcții este O(ct), deoarece nu depinde de numărul de elemente din listă.

Funcția *eliberare* eliberează memoria ocupată de o listă. Se constată că înainte de a se elibera un element, se memorează câmpul *urm* al elementului respectiv. Altfel, dacă prima oară s-ar fi eliberat memoria, câmpul *urm* ar fi devenit indisponibil și nu ar mai fi putut fi preluat. Complexitatea acestei funcții este O(n), deoarece trebuie eliberat fiecare element din listă. Dacă se dorește refolosirea pointerului care pointa la începutul listei, după ce aceasta a fost eliberată, pointerul va trebui reinițializat cu NULL, pentru a pointa astfel din nou la o listă vidă.

Funcția *adaugaInceput* adaugă un element la începutul listei și returnează noua listă. Adăugarea la început este simplă: câmpul *urm* al noului element va pointa la vechea listă, iar lista rezultată după adăugare va începe cu elementul nou introdus. Se constată că complexitatea lui *adaugaInceput* este O(ct).

Funcția *adaugaSfarsit* adaugă un element la sfârșitul listei și returnează noua listă. Pentru aceasta, dacă lista nu este vidă, trebuie parcurse toate elementele, până când se ajunge la ultimul element (cel al cărui câmp *urm* este *NULL*). Acestui ultim element i se setează câmpul *urm* cu adresa noului element și se returnează lista originală. Din cauză că la adăugarea un nou element trebuie iterată toată lista, complexitatea lui *adaugaSfarsit* este O(n).

Să considerăm acum cazul în care dorim să citim dintr-un fișier *n* elemente într-o listă, astfel încât ele să apară în listă în ordinea citirii. Vom folosi două metode și vom evalua complexitatea fiecărei metode:

1. Adăugăm fiecare element la începutul listei, folosind *adaugaInceput*, iar apoi inversăm ordinea elementelor din listă. Operația de inversare se poate implementa cu complexitate de O(n), astfel încât avem O(ct)*n+O(n) => O((ct+1)*n). Constanta cu care se înmulțește *n* nu modifică tipul complexității, ci doar mărește uniform numărul de operații, deci complexitatea acestui algoritm va fi O(n).
2. Adăugăm fiecare element la sfârșitul listei, folosind *adaugaSfarsit*. Ținând cont că la fiecare inserare trebuie parcursă toată lista de până atunci, pentru n elemente vom avea în total: 0+1+...+n-1 => n*(n-1)/2 operații. Păstrând membrul cel mai semnificativ, obținem pentru acest algoritm o complexitate $O(n^2)$, ceea ce este foarte mult în comparație cu prima metodă.

Comparând cele două metode, să considerăm cazul în care se pot face 1,000,000 de operații/secundă. Primul algoritm va termina adăugarea a 1,000,000 de elemente într-o secundă, iar al doilea în ~278 ore. Din acest motiv, în limbajele de programare care folosesc acest tip de implementare a listelor (OCaml, Lisp, Scheme, …) se evită adăugarea la sfârșit de listă, folosindu-se în schimb mai des prima metodă (adăugări la început și în final inversare).

Pentru a se urmări vizual acest aspect, se poate înlocui funcția *main* din programul de mai sus cu următoarea funcție, care adaugă 1,000,000 de elemente în listă și afișează din 1000 în 1000 câte elemente au fost deja adăugate:

```
int main()
{
   elem *lista1=NULL;
   int i;
   for(i=0;i<1000000;i++){
      lista1=adaugaSfarsit(lista1,i);
      if(i%1000==0)printf("%d\n",i);
      }
   return 0;
}
```

Se remarcă din această analiză importanța covârșitoare pe care o are folosirea unui algoritm cât mai adaptat problemei de rezolvat. Față de performanța pe care o poate aduce un algoritm cât mai bun pentru o anumită problemă, orice alte optimizări, incluzând folosirea unor calculatoare mai puternice, au doar o importanță secundară.

Conform modelului din exemplul 1, în care orice operație care modifică lista va returna noua listă, vom mai implementa și operația de ștergere a unui element din listă:

```
// sterge un element din lista
// returneaza noua lista
// folosire:  lista=sterge(lista,numar);
elem *sterge(elem *lista,int n)
{
   elem *pred;          // predecesor
   elem *crt;           // element curent
   for(pred=NULL,crt=lista;crt;pred=crt,crt=crt->urm){
      if(crt->n==n){
         if(pred==NULL){      // sterge primul element din lista
            lista=lista->urm;
            }else{            // sterge un element din interiorul listei
            pred->urm=crt->urm;
            }
         free(crt);
         return lista;
```

```
        }
      }
   return lista;                    // nu s-a gasit elementul in lista
}
```

Pentru ștergerea unui element din listă, a fost nevoie să se memoreze predecesorul elementului curent (*crt*). Dacă s-a găsit elementul de șters, câmpul *urm* al predecesorului se va seta la elementul de după elementul care se șterge. Complexitatea acestui algoritm este O(n), deoarece în cazul cel mai defavorabil trebuie parcursă toată lista.

Dacă dorim să facem o comparație între liste simplu înlănțuite și vectori cu redimensionare dinamică (folosind dublarea capacității la fiecare realocare), putem consulta tabelul următor:

Operație	**Lista**	**Vector**
adăugare la început	O(ct)	O(n) - trebuie deplasate toate elementele la dreapta
adăugare la sfârșit	O(n) - trebuie iterată toată lista	O(ct)+ - complexitate constant amortizată
căutare în mulțime neordonată	O(n)	O(n)
căutare în mulțime ordonată	O(n) - chiar dacă lista e ordonată, căutarea are loc tot liniar	$O(\log_2 n)$ - căutare binară
ștergere	O(n) - trebuie parcurse elementele până la cel vizat	O(n) - trebuie deplasate elemente la stânga
adresare indexată v[i]	O(i) - trebuie iterate *i* elemente	O(ct)
adresa unui element se menține după operații de adăugare sau ștergere - proprietate utilă atunci când adresele elementelor sunt memorate și în alte locuri	da	nu - dacă se dorește aceasta, se pot folosi vectori de pointeri la obiecte alocate dinamic; în acest caz, adresa obiectului rămâne fixă
reprezentare compactă în memorie - proprietate utilă pentru optimizarea folosirii *memoriei cache* a CPU	nu - fiecare element este alocat dinamic separat și are nevoie de un câmp suplimentar (*urm*)	da - elementele sunt alocate într-o zonă continuă de memorie și un element ocupă doar atâta memorie câtă îi este necesară

Din tabelul de mai sus, reiese faptul că vectorii au mai multe avantaje, printre care: inserarea la sfârșit este simplă, indexarea este și ea simplă (ceea ce este util în multe formule matematice) și reprezentarea în memorie este compactă.

Listele simplu înlănțuite în schimb permit o adăugare simplă la început și totodată mențin constantă adresa unui element (acest aspect se poate implementa și cu vectori).

Implementarea unei liste simplu înlănțuite cu memorarea ultimului element

După cum s-a spus la începutul laboratorului, există multe metode de implementare a unei liste, fiecare cu avantaje și dezavantaje specifice. De exemplu, la implementarea anterioară, operația de adăugare a unui element la sfârșitul listei are complexitatea O(n), deoarece trebuie iterată toată lista. Pentru simplificarea acestei operații, putem să memorăm adresa ultimului element.

Exemplul 2: Să se implementeze o listă folosind o structură de date care memorează atât adresa primului, cât și a ultimului element din listă:

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

// un element al listei
typedef struct elem{
    int n;                      // informatia utila
    struct elem *urm;       // camp de inlantuire catre urmatorul element
    }elem;

// aloca un nou element si ii seteaza campurile corespunzatoare
elem *nou(int n,elem *urm)
{
    elem *e=(elem*)malloc(sizeof(elem));
    if(!e){
        printf("memorie insuficienta");
        exit(EXIT_FAILURE);
        }
    e->n=n;
    e->urm=urm;
    return e;
}

typedef struct{
    elem *prim;     // primul element din lista
    elem *ultim;    // ultimul element din lista
    }Lista;

void init(Lista *lista)
{
    lista->prim=NULL;
    lista->ultim=NULL;
}

// adauga un element la inceputul listei
void adaugaInceput(Lista *lista,int n)
{
    elem *prim=lista->prim;
    lista->prim=nou(n,prim);
    if(!prim){      // lista initiala vida - trebuie setat si ultim pe elementul adaugat
        lista->ultim=lista->prim;
        }
}

// adauga un element la sfarsitul listei
void adaugaSfarsit(Lista *lista,int n)
{
    elem *e=nou(n,NULL);
    if(lista->ultim){                   // adaugare in lista nevida
        lista->ultim->urm=e;
        }else{                          // adaugare in lista vida
        lista->prim=e;
        }
    lista->ultim=e;
}
```

```
void afisare(Lista *lista)
{
    elem *crt;
    for(crt=lista->prim;crt;crt=crt->urm){
        printf("%d ",crt->n);
        }
    putchar('\n');
}

// elibereaza memoria ocupata de o lista
void eliberare(Lista *lista)
{
    elem *p,*crt=lista->prim;
    while(crt){
        p=crt->urm;
        free(crt);
        crt=p;
        }
}

int main()
{
    Lista lista;
    init(&lista);
    adaugaSfarsit(&lista,108);
    adaugaSfarsit(&lista,-1);
    adaugaSfarsit(&lista,49);
    afisare(&lista);
    eliberare(&lista);
    return 0;
}
```

În această implementare, *elem* a rămas neschimbat, dar lista în sine a fost implementată printr-o structură *Lista*, care conține câmpurile *prim* (primul element din listă) și *ultim* (ultimul element din listă). Inițial, această structură trebuie inițializată, folosind funcția *init*. Fiecare operație cu o listă va primi un pointer către această structură (transfer prin adresă, astfel încât să se poată opera asupra structurii).

Se constată că adăugarea la începutul listei a devenit puțin mai complicată, deoarece trebuie să mențină sincronizat și câmpul *ultim*, dar complexitatea acestei operații rămâne O(ct).

Marele avantaj al folosirii câmpului *ultim* este faptul că adăugarea la sfârșit de listă are acum complexitatea O(ct). Aceasta se petrece datorită faptului că nu mai trebuie iterată lista până se ajunge la ultimul element, ci sunt suficiente doar câteva operații de actualizare a pointerilor.

Aplicații propuse

Note:
- dacă nu se specifică altfel, aplicațiile se referă la liste care nu memorează adresa ultimului element
- pentru fiecare algoritm implementat, determinați care este complexitatea O(...) a lui

Aplicația 7.1: Să se scrie o funcție care primește două liste și returnează 1 dacă ele sunt identice, altfel 0.

Aplicația 7.2: Să se scrie o funcție care primește o listă și returnează lista respectivă cu elementele inversate. Funcția va acționa doar asupra listei originare, fără a folosi vectori sau alocare de noi elemente.

Aplicația 7.3: Să se scrie o funcție care primește ca parametri două liste și returnează o listă care reprezintă reuniunea elementelor lor, fiecare element apărând o singură dată, chiar dacă în listele originare el este duplicat.

Aplicația 7.4: Pentru implementarea listei care memorează adresa ultimului element, să se scrie o funcție *șterge*, care șterge din listă un element dat.

Aplicația 7.5: Să se scrie o funcție care primește două liste și returnează 1 dacă ele sunt egale, indiferent de ordinea și numărul elementelor, altfel returnează 0.
Exemple: listele {1, 7, 3, 1, 3} și {7, 1, 3, 7} sunt egale. Listele {1,2} și {2} nu sunt egale.

Aplicația 7.6: Să se scrie o funcție care primește ca parametri o listă (posibil vidă) de elemente sortate și un element. Funcția va insera în listă noul element, astfel încât lista să rămână sortată. Folosind aceasta funcție, să se scrie o funcție de sortare a unei liste, care primește ca parametru o listă nesortată și returnează una sortată. Programul nu va folosi niciun vector.

Aplicația 7.7: Pentru implementarea listei care memorează ultimul element, să se scrie o funcție care primește două liste sortate și returnează lista sortată care conține toate elementele lor. Pentru lista rezultată se va folosi doar operația de adăugare la sfârșit de listă.
Exemplu: {1, 2, 7, 8} și {2, 9} -> {1, 2, 2, 7, 8, 9}