Tabele de dispersie (hash tables)

Ne propunem să creăm o structură de date eficientă care să poată face următoarele operații cât mai repede posibil: *Inserează*, *Caută* și *Sterge*. Ideea din spatele hashing-ului este memorarea unui element într-un tablou sau listă, în funcție de cheia sa. Pe cazul mediu toate aceste operații să necesite *O(1)* timp.
Să vedem cum:
Elementele sunt puse într-un tablou alocat static pe pozițiile cheilor lor. Prin adresare directă, un element cu cheia *k* va fi memorat în locatia *k*. Toate cele 3 operații sunt extrem de simple (necesită doar o accesare de memorie), dar dezavantajul este că această tehnică "mănâncă" foarte multă memorie: *O(|U|)*, unde *U* este universul de chei.

hash (engl. *hash* = a toca, tocatură).

Algoritmii de căutare care folosesc hash constau din două părți distincte.

1. Prima parte este de a calcula o funcție hash care transformă cheia de căutare într-o adresă de tabel. In mod ideal, chei diferite s-ar mapa la adrese diferite, dar de multe ori două sau mai multe chei diferite pot fi distribuite la aceeași adresă de tabel.
2. A doua parte a unui algoritm de căutare care foloseşte hashing este un proces de rezolvare a coliziunilor. Una din metodele de de rezolvare a coliziunilor pe care o vom studia utilizează liste înlănţuite, și este, prin urmare, imediat utilizabilă în situații dinamice în care numărul de chei de căutare este dificil de prezis în avans.

Hashing este un bun exemplu de un compromis spațiu-timp. În cazul în care nu este *nici o limitare de memorie*, atunci am putea face orice căutare cu doar un acces la memorie pur și simplu folosind cheia ca o adresă de memorie, la fel ca în căutarea indexată. Acest ideal de multe ori nu poate fi atins deoarece cantitatea de memorie necesară este prohibitivă atunci când cheile sunt lungi. Pe de altă parte, în cazul în care nu ar exista *nici o limitare de timp*, atunci am putea obține răspunsul cu doar o cantitate minimă de memorie cu ajutorul unei metode de căutare secvențiale. Hashing oferă o modalitate de a folosi atât o cantitate rezonabilă de memorie și cât şi de timp care să ţină un echilibru între aceste două extreme. În particular, putem atinge orice echilibru alegem, pur și simplu prin ajustarea dimensiunii tabelei de hash, nu prin rescrierea codului sau alegerea de algoritmi diferiţi.

Hashing este o problemă clasică din informatică: diverşii algoritmi au fost temeinic studiați și sunt utilizaţi pe scară largă. Vom vedea că, pe baza unor ipoteze generoase, nu este nerezonabil să se aștepte ca operațiile de căutare și inserare în *tabela de simboluri* să aibă loc în timp constant, independent de mărimea tabelei.

Această așteptare este performanța teoretic optimă pentru orice implementare a tabelei de simboluri, dar hashing-ul nu este un panaceu universal, din două motive principale:

- timpul de rulare depinde de lungimea cheii, care poate fi o responsabilitate în aplicații practice, cu chei lungi.
- hash nu oferă implementări eficiente pentru alte operații ale tabelei de simboluri, cum ar fi *select* sau *sort*.

Primul pas în a rezolva problema memoriei este de a folosi *O(N)* memorie în loc de *O(|U|)*, unde *N* este numărul de elemente adăugate în hash. Ceea ce trebuie să abordăm este calculul *funcței de hashing*, care transformă *cheile* în *adrese ale tabelului.*

Astfel, un element cu cheia *k* nu va fi memorat în locatia *k*, ci în *h(k)*, unde *h: U -> {0, 1, ..., N-1}* - o funcție aleasă aleator, dar deterministă (*h(x)* va returna mereu aceeași valoare pentru un anumit *x* în cursul rulării unui program). Acest calcul aritmetic este în mod normal, simplu de implementat, dar trebuie să se procedeze cu prudență, pentru a evita diversele capcane subtile.

Funcția de hashing depinde de tipul cheie. Strict vorbind , avem nevoie de o funcție de hashing diferită pentru fiecare tip de cheie , care ar putea fi folosită . Pentru eficiență, în general se evită conversia de tip explicită , străduindu-se în schimb pentru o intoarcere la ideea de a considera *reprezentarea binară* a cheii într-un cuvânt mașină ca un întreg pe care îl putem folosi pentru calcule aritmetice . A fost o practică comună pe calculatoarele timpurii de a vedea o valoare de cheie la un moment dat ca *șir* și la altul ca *un întreg*. În unele limbaje de nivel înalt este dificil să se scrie programe care depind de modul în care cheile sunt reprezentate pe un anumit calculator, pentru că astfel de programe , prin natura lor , sunt dependente de mașină și, prin urmare, nu sunt portabile. Funcțiile hash , în general, sunt dependente de procesul de transformare a cheii în numere întregi , astfel încât independența și eficiența mașinii sunt uneori dificil de realizat simultan în implementări de hashing . Putem transforma de obicei chei întregi simple sau în virgulă flotantă, cu doar o singură operație mașină , dar cheile de string-uri și alte tipuri de chei compuse necesită mai multă atenție și mai multă atenție la eficiență.

Metode de transformare a cheilor:

- Variabilele de tip string pot fi transformate în numere în baza 256 prin înlocuirea fiecărui caracter cu codul său *ASCII*.
- Variabilele de tip dată se pot converti la întreg prin formula:
  X = A * 366 + L * 31 + Z
  unde A, L şi Z sunt respectiv anul, luna si ziua datei considerate. De fapt, aceasta funcție aproximează numărul de zile scurse de la inceputul secolului I.
- Analog, variabilele de tip oră se pot converti la întreg cu formula:
  X = (H * 60 + M) * 60 + S
  unde H, M si S sunt respectiv ora, minutul şi secunda considerate, sau cu formula
  X = ((H * 60 + M) * 60 + S) * 100
  dacă se ţine cont şi de sutimile de secundă. De data aceasta, funcția este surjectivă (oricărui număr întreg din intervalul 0 - 8.639.999 îi corespunde în mod unic o oră).

- In majoritatea cazurilor, datele sunt structuri care conţin numere şi stringuri. O bună metodă de conversie constă în alipirea tuturor acestor date şi în convertirea la baza 256. Caracterele se convertesc prin simpla înlocuire cu codul *ASCII* corespunzător, iar numerele prin convertirea în baza 2 şi taierea în "bucăţi" de câte opt biti. Rezultă numere cu multe cifre (prea multe chiar şi pentru tipul long long), care sunt supuse unei operații de împărțire cu rest. Cum? De ce?

Exemplu simplificat:

Presupunem că avem o tabelă cu 101 poziții și cheia
A K E Y = 00001 01011 00101 11001 (Cod pe 5 biți)=44217_{10} ≡80 (mod 101)
Baza 32 (semne) =>
A K E Y = $1 * 32^3 + 11 * 32^2 + 5 * 32^1 + 25 * 32^0$
Dar dacă
V E R Y L O N G K E Y =
101100010110010110010110001111011100011101011001011100 1=
$22*32^{10}+5*32^9+18*32^8+25*32^7+*32^6+15*32^5+14*32^4+7*32^3+11*32^2+5*32^1+25$
$=_{\text{Horner}}$
(((((((((22*32+5)32+18)32+25)32+12)32+15)32+14)32+7)32+11)32+5)32+25

Funcţii de dispersie foarte des folosite

1. Metoda împărţirii cu rest

Funcţia hash este: h(x) = x mod M unde M este numărul de intrări în tabelă. Problema care se pune este să-l alegem pe M cât mai bine, astfel încât numărul de coliziuni pentru oricare din intrări să fie cât mai mic. De asemenea, trebuie ca M să fie cât mai mare, pentru ca media numărului de chei repartizate la aceeaşi intrare să fie cât mai mică. Totuşi, experienţa arată ca nu orice valoare a lui M este bună.

Funcţiile de hash întorc un număr între *0* şi *M-1*, unde *M* este dimensiunea maximă a tabelei de hash. Este recomandat ca *M* să fie ales un număr prim şi să se evite alegerea lui $M=2^k$.

Motivul?

Exemplul 1:

Presupunem că avem o tabelă cu 32 poziții și cheia

A K E Y = 00001 01011 00101 11001 (Cod pe 5 bits)=44217_{10} ≡80 (mod 101)

Baza 32 (semne) =>

A K E Y = $1 * 32^3 + 11 * 32^2 + 5 * 32^1 + 25 * 32^0$

Dar dacă

V E R Y L O N G K E Y =

10110001011001011001011000111101110001110101100101110 01=

$22*32^{10}+5*32^9+18*32^8+25*32^7+*32^6+15*32^5+14*32^4+7*32^3+11*32^2+5*32^1+25$

$=_{Horner}$

(((((((((22*32+5)32+18)32+25)32+12)32+15)32+14)32+7)32+11)32+5)32+25

valoarea mod32 ar fi intotdeauna valoarea ultimei litere din cheie

Exemplul 2:

Din aceleaşi motive, alegerea unei valori ca 1000 sau 2000 nu este prea inspirată, deoarece ţine cont numai de ultimele 3-4 cifre ale reprezentării zecimale.

2. Metoda înmulţirii

Funcţia hash este h(x) = [M * {x*A}] 0 < A < 1, iar prin {x*A} se înţelege partea fracţionară a lui x*A, adică x*A - [x*A].

Exemplu:
dacă alegem M = 1234 şi A = 0.3, iar x = 1997, atunci avem h(x) = [1234 * {599.1}] = [1234 * 0.1] = 123. Se observă că funcţia h produce numere între 0 şi M-1. Într-adevăr $0 \leq \{x*A\} < 1$ $0 \leq M * \{x*A\} < M$. *(Catalin Francu)*

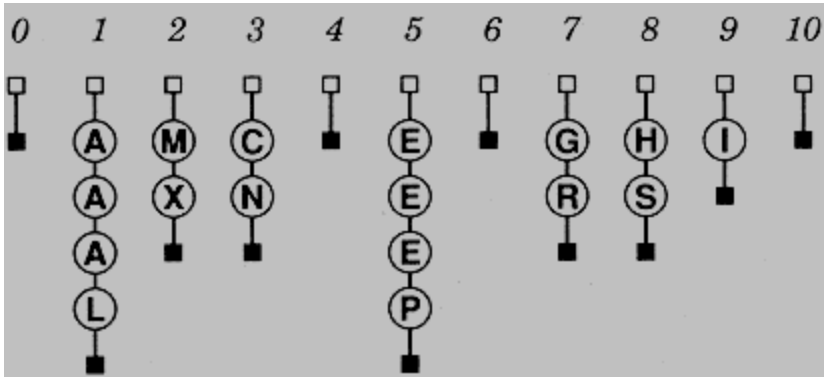
Observatie: valoarea lui M nu mai are o mare importanţă. M poate fi cât de mare ne convine, eventual o putere a lui 2. În practică, s-a observat că dispersia este mai bună pentru unele valori ale lui A şi mai proastă pentru altele;
Donald Knuth propune valoarea $A = \frac{\sqrt{5}-1}{2} \approx 0.618034$

Dacă funcția aleasă are comportament cât mai apropiat de o generare de numere aleatoare, elementele vor fi "împrăștiate" în tabel în mod uniform. Pentru fiecare input, fiecare ieșire ar trebui să fie într-un anumit sens, la fel de probabilă. Ideal ar fi ca fiecare element să fie stocat singur în locația lui. Acest lucru însă nu este posibil, pentru că $N < |U|$ și, deci, de multe ori mai multe elemente vor fi repartizate în aceeasi locație.
Acest fenomen se numeste coliziune.

Metode de rezolvare a coliziunilor:

- Înlănțuire
- Liste statice
- Adresare deschisă
- Double hashing-ul lui Mihai Pătrașcu

Înlănțuire



În fiecare poziție din tabel ținem o listă înlănțuită; *insert*, *delete* și *search* parcurg toată lista.

Pe un caz pur teoretic, toate cele *N* elemente ar putea fi repartizate în aceeași locație, însă pe cazuri practice lungimea medie a celui mai lung lanț este de *lg(N)*.

Varianta: în loc de listă, arbori

Liste statice

Varianta îmbunătățită a metodei anterioare:
pentru că lungimea unui lanț este cel mult *lg(N)*, putem să folosim, în loc de liste înlănțuite, vectori alocați dinamic de lungime *lg(N)* - sau *lg(N) + 3*

se elimină pointerii.

```
//implementare adaptand procedurile de la tabele de
// simboluri pentru M liste statice
#include <stdlib.h>
#include "Item.h"
typedef struct STnode* link;
struct STnode { Item item; link next; };
static link *heads, z;
static int N, M;

void STinit(int max)
{ int i;
N = 0; M = max/5;
heads = malloc(M*sizeof(link));
z = NEW(NULLitem, NULL);
for (i = 0; i < M; i++) heads[i] = z;
}
Item searchR(link t, Key v)
{
     if (t== z) return NULLitem;
     if (eq(key(t->item), v)) return t->item;
     return searchR(t->next, v);
}
Item STsearch(Key v)
{ return searchR(heads[hash(v, M)], v); }

void STinsert(Item item)
{ int I = hash(key(item), M);
heads[i] = NEW(item, heads[i]); N++; }

void STdelete(Item item)
int i = hash(key(item), M);
heads[i]= deleteR(heads[i], item); }
```

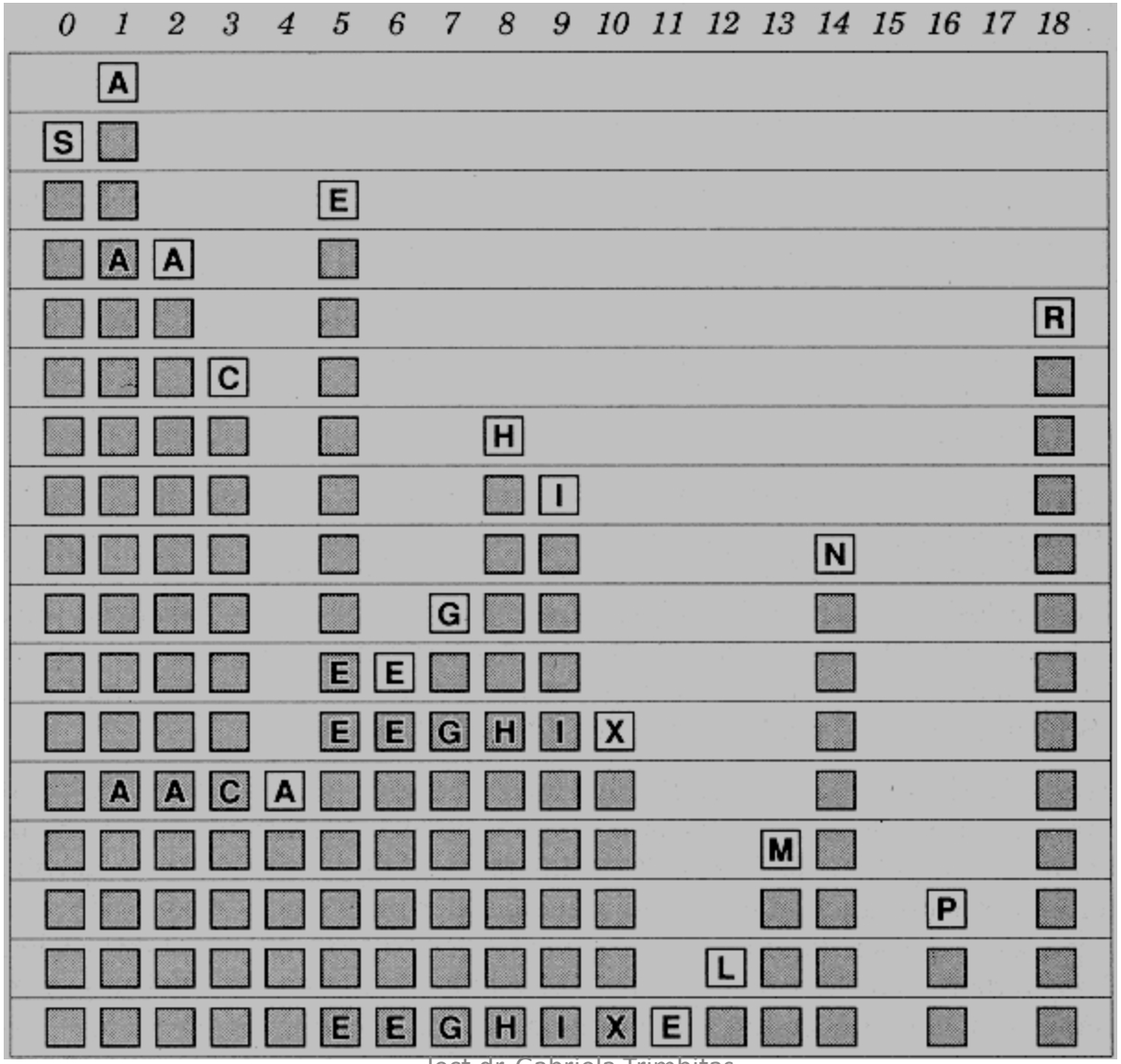
Adresare deschisă

Prin adresare deschisă, toate elementele sunt memorate în tabela de dispersie. Pentru a realiza operațiile cerute, verificăm succesiv tabela de dispersie până când fie găsim o locație liberă (in cazul *Insert*), fie găsim elementul căutat (pentru *Cauta*, *Sterge*). Insă, în loc să căutăm tabelă de dispersie în ordinea *0, 1, ..., N-1*, șirul de poziții examinate depinde de cheia ce se inserează. Pentru a determina locațiile corespunzătoare, extindem funcția de hashing astfel încât să conțină și numărul de verificare ca un al doilea parametru *h: U * {0, 1, ..., N-1} -> {0, 1, ..., N-1}*. Astfel, când vom insera un element, verificăm mai întâi locația *h(k, 0)*, apoi *h(k, 1)* etc. Când ajungem să verificăm *h(k, N)* putem să ne oprim pentru că tabelă de dispersie este plină. Pentru căutare aplicăm aceeași metodă; dacă ajungem la *h(k, N)* sau la o poziție goală, înseamnă că elementul nu există. Stergerile se fac însă mai greu, pentru că nu se poate "șterge" pur și simplu un element deoarece ar strica toată tabela de dispersie . In schimb, se marchează locația ce trebuie ștearsă cu o valoare *STERS* și se modifică funcția *Insert* astfel încât să vadă locațiile cu valoarea *STERS* ca poziții goale.

Exemplu:

Schlüssel:	A	S	E	A	R	C	H	I	N	G	E	X	A	M	P	L	E
Hash:	1	0	5	1	18	3	8	9	14	7	5	5	1	13	16	12	5

M=19



Această implementare a unei tabele de dispersie păstrează articolele într-o tabelă cu lungime dublă faţă de numărul maxim de articole care se aşteaptă a fi introduse, articole iniţializate cu valori nule (NULL values).
Pentru a insera un nou articol, îi calculăm poziţia în tabelă, iar dacă este deja ocupată se caută spre dreapta folosind *macroul null* pentru a testa dacă o poziţie este ocupată. Pentru *a căuta* un articol cu o cheie dată îi calculăm poziţia şi apoi scanăm pentru a găsi concordanţa sau ne oprim dacă am ajuns la o poziţie neocupată.
Funcţia STinit setează M astfel încât ne aşteptăm ca tabela să fie jumătate plină, pentru ca celelalte operaţii să aibă nevoie de puţine încercări dacă funcţia de hashing produce valori apropiate de valori aleatoare.

```
#include <stdlib.h>
#include "Item.h"
#define null(A) (key(st[A]) == key(NULLitem))
static int N, M;
static Item *st;
void STinit(int max)
  { int i;
    N = 0; M = 2*max;
    st = malloc(M*sizeof(Item));
    for (i = 0; i < M; i++) st[i] = NULLitem;
  }
int STcount() { return N; }
void STinsert(Item item)
  { Key v = key(item);
    int i = hash(v, M);
    while (!null(i)) i = (i+1) % M;
    st[i] = item; N++;
  }
Item STsearch(Key v)
  { int i = hash(v, M);
    while (!null(i))
      if eq(v, key(st[i])) return st[i];
      else i = (i+1) % M;
    return NULLitem;
  }
```

Program Hashing cu adresare deschisa

```
/************************************************************
* You can use all the programs on  www.c-program-*example.com
*for personal and learning purposes. For *permissions to use
*the programs for commercial purposes,
* contact info@c-program-example.com
* To find more C programs, do visit www.c-program-example.com
and browse!
*                       Happy Coding
************************************************************/
#include<stdio.h>
#include<conio.h>
void main() {
 int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 int n, value;
 int temp, hash;
 clrscr();
 printf("\nEnter the value of n(table size):");
 scanf("%d", &n);
 do {
  printf("\nEnter the hash value");
  scanf("%d", &value);
  hash = value % n;
  if (a[hash] == 0) {
   a[hash] = value;
   printf("\na[%d]the value %d is stored", hash, value);
  } else {
   for (hash++; hash < n; hash++) {
    if (a[hash] == 0) {
     printf("Space is allocated give other value");
     a[hash] = value;
     printf("\n a[%d]the value %d is stored", hash, value);
     goto menu;
    }
   }
  for (hash = 0; hash < n; hash++) {
     if (a[hash] == 0) {
      printf("Space is allocated give other value");
      a[hash] = value;
      printf("\n a[%d]the value %d is stored", hash, value);
      goto menu;
     }
    }
   printf("\n\nERROR\n");
   printf("\nEnter '0' and press 'Enter key' twice to exit");

  }

  menu:

  printf("\n Do u want enter more");

  scanf("%d", &temp);

 }

 while (temp == 1);

 getch();

}
```

Unde sunt greselile???

Program Hashing cu adresare deschisa corectat

```
#include<stdio.h>
#include<conio.h>
void main() {
 int a[1000]/* = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };*/
 int n,i,value;
 int temp, hash;
 clrscr();
 printf("\nEnter the value of n (modulus):");
 scanf("%d", &n);
 for (i = 0; i < 1000, i++) a[i] = 0;
 do {
  printf("\nEnter the key value");
  scanf("%d", &value);
  hash = value % n;
  if (a[hash] == 0) {
   a[hash] = value;
   printf("\na[%d]the value %d is stored", hash, value);
  } else {
   for (hash++; hash < n; hash++) {
    if (a[hash] == 0) {
     printf("Space is allocated to another key");
     a[hash] = value;
     printf("\n a[%d]the value %d is stored", hash, value);
     goto menu;
    }
   }
```

```
for (hash = 0; hash < n; hash++) {
  if (a[hash] == 0) {
    printf("Space is allocated to another key");
    a[hash] = value;
    printf("\n a[%d]the value %d is stored", hash, value);
     goto menu;
    }
   }
   printf("\n\nERROR\n");
   printf("\nEnter '0' and press 'Enter key' twice to exit");

  }

  menu:

  printf("\n Press 1 to enter another key ");

  scanf("%d", &temp);

 }

 while (temp == 1);

 getch();

}
```

Proprietatea 1: Înlănțuirea separată reduce numărul de comparații pentru căutare secvențială cu un factor M (în medie), folosind spațiu suplimentar pentru M înlănțuiri.

Proprietatea 2: Într-o tabelă de dispersie care este mai puțin de 2/3 plină adresarea deschisă necesită mai puțin de 5 teste.

Double hashing-ul lui Mihai Pătrășcu

O îmbunătățire foarte mare la tabelă de dispersie este... încă o tabelă de dispersie . Vom avea 2 tabele, fiecare cu propria ei funcție de hashing, iar coliziunile le rezolvăm prin înlănțuire; cand inserăm un element, il vom adăuga în tabela în care intră într-un lanț mai scurt.

Căutarea se face în ambele tabele în locațiile returnate de cele 2 funcții de hashing; ștergerea la fel.

Lungimea celui mai lung lanț va fi, în medie, $lg(lg(N))$.

În practică, lungimea unui astfel de lanț nu va depăși *4* elemente, pentru că cel mai mic N pentru care $lg(lg(N)) = 5$ este $2^{32} \sim 4.000.000.000$!!!

⇨ În loc de liste folosim vectori statici de dimensiune *4*.


Dorinţa oamenilor cu adevărat mari este să-i facă pe alţi oameni fericiţi.
Blaise Pascal
cuvinte celebre.ro