Tabele de Dispersie.

Definiții şi terminologie.

Tabelele de dispersie reprezintă o modalitate foarte eficientă de implementare a dicționarelor.

Acestea asigură complexitate constantă **O(1)** în medie, pentru operațiile de inserare, ştergere şi căutare.

Dispersia nu presupune ordonarea informației păstrate. Operațiile cu arbori, care presupun ordonarea informației între elemente sunt mai puțin eficiente (**O(log n)**).

Dacă:

- structura de date (tabela de dispersie) e accesată prin valoarea unei chei în **O(1)**
- funcția care transformă cheia într-o poziție într-un tabel are complexitate **O(1)**

atunci inserarea / ştergerea / căutarea se vor face cu complexitate **O(1)**

Dispersia implementează această idee folosind:

- o tabelă de dispersie
- o funcție de dispersie

Dacă funcția de dispersie calculează pentru două chei diferite aceeaşi valoare de dispersie, se produce o ***coliziune***. Aceesul la cheile aflate în coliziune nu se mai face cu complexitate constantă.

Pentru a reduce coliziunile se folosesc tabele de dimensiuni foarte mari. (eficiența temporală mai bună este anihilată de eficiența spațială mai proastă).

Tabelele de dispersie pot fi *deschise* (sau *externe*) şi *închise* (sau *interne*)**.**

Funcții de dispersie.

Fie **K** – o *mulțime de chei* (întregi, şiruri de caractere, forme complexe de biți) şi **B** – o *mulțime de valori de dispersie* (bini). Vom considera **B={0,1,...,MAX-1}**.

Aplicația **hash : K → B** poartă numele de ***funcție de dispersie***, şi asociază mulțimii cheilor o mulțime de valori de dispersie.

Este de dorit ca această funcție să fie ***injectivă***, adică la două chei diferite $k_1 \neq k_2$ să corespundă valori de dispersie diferite $hash(k_1) \neq hash(k_2)$, în caz contrar apare o ***coliziune***.

Funcția de dispersie trebuie să fie aleasă astfel încât să asigure o dispersie cât mai uniformă, care să minimizeze numărul de coliziuni.

In general numărul cheilor efectiv memorate este mult mai mic decât numărul total de chei posibile**.**

Utilizarea adresării directe, având complexitate **O(1)** ar conduce la folosirea neeficientă a unor tablouri foarte mari care ar conține relativ puține elemente.

Căutarea în dicționar se face cu complexitate **O(1)** dacă:

- se reprezintă datele printr-un tabel de dispersie **H**
- se găseşte o funcție de dispersie care mapează cheile în indici din tabelul de dispersie în mod unic: $k_1 \neq k_2 \Rightarrow hash(k_1) \neq hash(k_2)$
- memorează elementul (**k,i**) în **H[hash(k)]**

Alegerea funcției de dispersie

Fie **MAX** dimensiunea tabelului de dispersie. Pentru valoarea de dispersie **0:MAX-1** se introduce tipul **Index**:

```
typedef unsigned int Index;
```

Pentru chei întregi se folosesc:

1. ***Metoda împărțirii*** : **hash(k) = k % MAX** în care **MAX** trebuie să fie un ***număr prim*** care ***să nu fie apropiat de o putere a lui 2***.

```
Index Hash(void *k, Index MAX){
   return *(int*)k % nextprim(MAX);
}
```

2. ***Metoda înmulțirii***: reține o parte din biții părții fracționare a produsului **k*A**, în care **A** este o constantă (Knuth recomandă pentru **A** raportul de aur **(√5-1)/2**) . De exemplu dacă dimensiunea tabelei de dispersie este de 1024, este suficient un index de 16 biți, deci se calculează partea fracționară a lui **A**: 2^{16}***A = 40503** și se selectează cei mai semnificativi 10 biți, prin deplasare dreapta cu 6 poziții.

```
Index Hash(void *k, Index MAX){
  Index c=40503;
  return (c * *(int*)k) >> 6;

}
```

Dacă cheia este un şir de caractere,

```
typedef char *Index;
```

se foloseşte una din funcțiile:

3. ***Adunarea codurilor caracterelor*****.**

```
Index Hash(void *k, Index MAX){
  Index vd=0;
  while(*(char*)k != 0)
    vd += *(char*)k++;
  return ( vd % nextprim(MAX) );
}
```

4. ***Metoda sau-exclusiv*****.**

Dintr-un tablou de numere generate aleator se selectează acele numere care au ca indici caracterele cheii şi face **sau-exclusiv** între ele:

```
unsigned char Rand8[256];
Index Hash(void *k, Index MAX){
  Index vd = 0;
  while(*(char*)k)
    vd = Rand8[vd ^ *(char*)k++];
  return (vd % nextprim(MAX));
}
```

5. ***Metoda acumulării polinomiale*****.**

Se partiționează biții cheii în componente de lungime fixă (8, 16 sau 32 de biți), fie acestea $a_0, a_1, ..., a_{n-1}$.

Se evaluează polinomul $p(z)=a_0+a_1z+...+a_{n-1}z^{n-1}$, într-un punct dat z (cu rezultate bune se ia z=**32**), ignorând depăşirile.

Se evaluează polinomul cu schema lui Horner, în $O(n)$.

$p_i(z)=a_{n-i-1}+zp_{i-1}(z)$, $i=1:n-1$

```
Index Hash(void *k, Index MAX){
  Index vd=0;
  while(*(char*)k != 0)
    vd = (vd<<5)+*(char*)k++;
  return ( vd % nextprim(MAX) );
}
```

6. ***Dispersie universală***

O familie de funcții de dispersie este universală dacă pentru orice $0 \le j, k \le M-1$

$Prob(h(j)=h(k)) \le 1/M$

- Se alege p prim între M şi $2M$
- Se alege aleatoriu $0<a<p$ şi $0\le b<p$ şi se defineşte:

  $hash(k)=(ak+b)\%p\%N$

Mulțimea tuturor funcțiilor h astfel definite este universală.

Demonstrație:

$f(k)=(ak+b)\%p$ şi $g(k)=k \% M \Rightarrow h(k)=g(f(k))$

Presupunem că am avea coliziuni: $f(k)=f(j)$

$$aj + b - \left\lfloor \frac{aj + b}{p} \right\rfloor \cdot p = ak + b - \left\lfloor \frac{ak + b}{p} \right\rfloor \cdot p$$

$$a(j - k) = \left(\left\lfloor \frac{aj + b}{p} \right\rfloor - \left\lfloor \frac{ak + b}{p} \right\rfloor \right) \cdot p$$

$a(j-k)$ este deci multiplu de p, dar ambii factori sunt mai mici ca p, rezultă că $a(j-k) = 0$, adică $j=k$ contradicție ! de unde rezultă că f nu produce coliziuni.

Dacă f nu produce coliziuni, atunci numai g ar putea produce coliziuni.

Fixăm un număr x. Dintre cei p întregi $y=f(k)$, diferiți de x, numărul pentru care $g(y)=g(x)$ este cel mult $\lceil p/N \rceil - 1$

Întrucât există p selecții posibile pentru x, numărul de funcții care ar putea produce o coliziune între j şi k este cel mult:

$p(\lceil p/N \rceil - 1) \le p(p-1)/N$

Există $p(p-1)$ funcții h, astfel că probabilitatea unei coliziuni este cel mult:

$$\frac{p(p-1)/N}{p(p-1)} = \frac{1}{N}$$

adică mulțimea tuturor funcțiilor posibile este universală.

Putem da pentru orice funcție de dispersie semnătura comună:

```
Index hash(void *cheie, Index ind);
```

Utilizatorul îşi poate defini propria funcție de dispersie, transmisă ca pointer la funcție în lista de parametri a operațiilor primitive. În acest scop vom defini tipul pointer la funcția de dispersie:

```
typedef  Index (*PFD)(void *, Index);
```

Strategii de rezolvare a coliziunilor.

In cazul *dispersiei deschise* (cu *înlănțuire*), coliziunile se rezolvă prin punerea lor într-o listă înlănțuită asociată valorii de dispersie comune.Se crează astfel un tablou de liste de coliziune.

În cazul *dispersiei închise*:

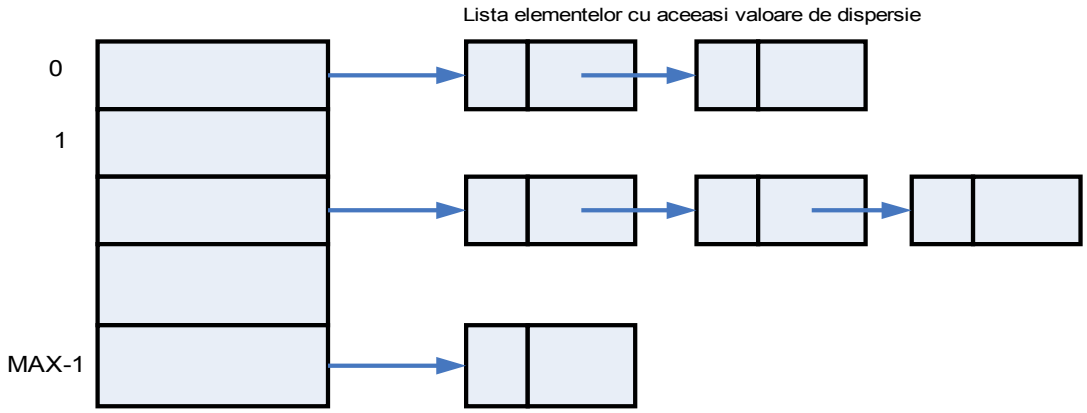
- toate elementele se memorează în tabelul de dispersie
- nu se mai folosesc pointeri
- la producerea unei coliziuni se verifică alte celule, până când se găseşte una liberă.Celulele verificate formează un *lanț de coliziune* $h_0(x), h_1(x), \ldots$ unde

$h_i(x)=(hash(x)+F(i))\%M$ cu $F(0)=0$

Funcția **F** dă strategia de rezolvare a coliziunii.
Întrucât toate datele se introduc în tabela de dispersie este necesar un tabel cu dimensiune mai mare.
În general factorul de încărcare ar trebui să fie sub 0.5 pentru dispersia închisă.

Dispersie deschisă (sau **înlănțuită)**

Dimensiunea tabloului listelor de coliziuni este numărul total de valori de dispersie **MAX.**



Inlantuirea coliziunilor in cazul dispersiei deschise

Operațiile specifice sunt:

TD TD_New (int m) -crează un tabel de dispersie cu **m** liste de coliziune vide,

`void TD_Insert (TD h, void *k, void *info, PFC comp, PFD hash)` - insereaza perechea **`(k, info)`** în lista cu numărul **`hash(k).`** Compararea cheilor se face cu funcția **`comp`**.

`List_Iter TD_Search (TD h, void *k, PFC comp, PFD hash)` - caută cheia **`k`** în lista de coliziune **`hash(k)`** şi întoarce adresa nodului din lista de coliziune care conține cheia

`void *TD_Remove (TD h, void *k, PFC comp, PFD hash)` – şterge din lista de coliziune cu numărul **`hash(k)`** elementul cu cheia **`k.`** Funcția întoarce adresa informației asociate cheii.

`void *TD_Get(Lista L,List_Iter p)` – obține informația asociată cheii din lista de coliziune, din nodul dat ca parametru.

`int TD_Empty(TD h)` – test tabel de dispersie vid

`int TD_Size(TD h)` – numărul de elemente memorat în tabelul de dispersie.

In cazul cel mai nefavorabil, complexitatea acestor operații este **`O(n)`**.
In medie, complexitatea acestor operații este **`O(1+λ)`**, în care factorul de încărcare
`λ = n/MAX`, unde

- **`n`** este numărul elementelor memorate
- **`MAX`** este dimensiunea tabelei de dispersie.

(în cazul dispersiei deschise **`λ`** poate fi supraunitar).
Dispersia cu înlănțuire separată are dezavantajul că necesită pointeri, alocare de memorie, deci este lentă.

Interfață dispersie deschisă.

```
// Dispersie deschisa
#ifndef TD_H
#define TD_H
#include "list.h"

typedef int (*PFC)(void*, void*);
typedef int (*PFD)(void*, int);

struct TabDisp;
typedef struct TabDisp *TD;

TD    TD_New (int m);
List_Iter TD_Search (TD h, void *k, PFC comp, PFD hash);
void  TD_Insert (TD h, void *k, void *info, PFC comp, PFD hash);
void  *TD_Remove(TD h, void *k, PFC comp, PFD hash);
void  TD_Delete (TD *h);
int   TD_Empty (TD h);
int   TD_Size (TD h);
#endif
```

Implementare dispersie deschisă.

Pentru a păstra structura nodurilor **`TAD Listă`**, vom grupa asocierea (cheie, informație) într-o structură şi vom defini funcții pentru creerea acestei asocieri şi pentru selectarea componentelor. Aceste funcții vor fi interne secțiunii de implementare, deci inaccesibile utilizatorului.

```
typedef struct per{
```

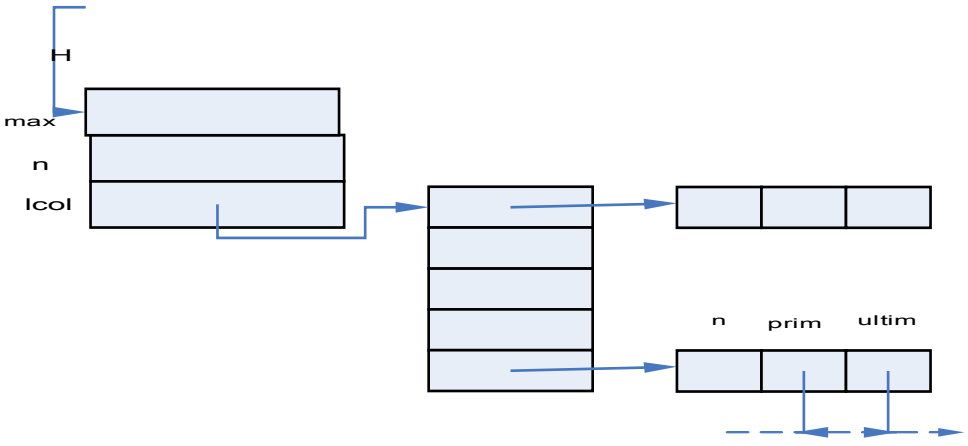
```
  void       *ch;     //cheia elementului
  void       *info;   //informația elementului
} *pereche;

//creaza perechea (cheie,informatie) intorcand adresa asocierii
void *dublet(void *k, void *i){
  pereche p =(pereche)malloc(sizeof(struct per));
  p->ch = k;
  p->info = i;
  return p;
}

//selecteaza cheia din asocierea (cheie, informatie)
void *sel_cheie(pereche p){ return p->ch; }

//selecteaza informatia din asocierea (cheie, informatie)
void *sel_info(pereche p){ return p->info; }
```

Tabela de dispersie va conține dimensiunea max, numărul de elemente şi tabloul listelor de coliziune.

```
struct TabDisp{
  int max;            //dimensiune TD
  int n;              //numar efectiv elemente din TD
  List *lcol;         //tabloul de liste de coliziune
};
```

Inițializarea tabelei de dispersie presupune:

- alocarea de memorie pentru structura tabelă de dispersie şi inițializarea ei
- alocarea de memorie pentru tabloul de liste de coliziune şi inițializarea fiecărei liste ca listă vidă




```
TD TD_New (int m){
   TD h;
   h=(TD)malloc(sizeof(struct TabDisp));
   h->max=m;
   h->n=0;
   h->lcol=(Lista*)malloc(m*sizeof(Lista));
   int i;
```

```
   for(i=0; i<m; i++)
      h->lcol[i]=L_New();
   return h;
}
```

Funcția **TD_Search()** întoarce poziția nodului cu cheia **k** în lista de coliziune a elementelor cu aceeaşi valoare de dispersie sau **NULL**, dacă elementul nu există în tabela de dispersie.

```
List_Iter TD_Search (TD h, void *k, PFC comp, PFD hash){
   List_Iter p;
   p=L_Find(h->col[hash(k, h->max)], k, comp);
   return p;
}
```

Funcția **TD_Insert()** caută mai întâi elementul cu cheia **k** în tabela de dispersie. Dacă îl găseşte, el nu mai trebuie inserat, altfel se crează un nod în care se pune cheia **k** şi informația asociată **info**, şi se inserează în lista selectată de funcția de dispersie.

```
void TD_Insert (TD h, void *k, void *info, PFC comp, PFD hash){
  List_Iter p;
  p=TD_Search (H, k, comp, hash);
  if(p) return;       //exista, nu se mai pune
  void *d = dublet(k, info);
  L_Insert(h->lcol[hash(k,h->max)], p, d);
  h->n++;
}
```

Penru funcția **TD_Remove()** se caută mai întâi elementul în tabela de dispersie, în lista corespunzăoare cheii. Dacă nu se găseşte, nu avem ce şterge, altfel se şterge elementul din listă şi se scade numărul elementelor din listă.

```
void *TD_Remove (TD h, void *k, PFC comp, PFD hash){
  List_Iter p;
  p=TD_Search (h, k, comp, hash);
  if(!p) return NULL;
  void *d=L_Remove(h->lcol[hash(k,h->max)],p);
  h->n--;
  return sel_info((struct per *)d);
}

void TD_Delete(TD *h){
  int i;
  for(i=0; i<(*h)->max; i++)
     L_Delete(&(*h)->lcol[i]);
  free(*h);
}
```

Dispersia închisă.

Dacă funcția inițială de dispersie este **h(x)**, notată **h(x, 0)**, atunci celelalte funcții de dispersie se obțin astfel:

1. Verificarea liniară – adoptă o funcție liniară de **i**, de obicei **F(i)=i**. Celulele alăturate sunt verificate ciclic în căutarea unei celule libere. Dacă tabelul este suficient de mare se va găsi o celulă liberă, dar timpul de căutare poate deveni foarte mare.

h(x,i)=(h(x)+i) % MAX

Lanțul de coliziune poate avea dimensiunea maximă M; el se termină mai devreme, dacă se întâlnește o celulă marcată **LIBER**.

Verificarea liniară favorizează creerea de blocuri vecine ocupate – ***aglomerări (ciorchini) primare***, care pot apare și pentru tablouri puțin ocupate și degrada performanțele.

2. Verificarea pătratică – folosește **F(i)=i²** și elimină aglomerările primare. Dacă dimensiunea tabloului este număr prim, un nou element poate fi întotdeauna inserat dacă tabelul este cel puțin pe jumătate gol.

h(x,i)=(h(x)+i²) % MAX

Metoda verificării pătratice generează cel mult o secvență de **MAX** funcții de dispersie, dar produce *aglomerări secundare*

3. Dispersia dublă – folosește o a doua funcție de dispersie **h_2(k)** și tratează coliziunile punând elementul în prima celulă liberă dintre

h(x,i)=(h(x)+ih_2(x)) % MAX,

- a doua funcție de dispersie **h_2(k)** nu poate lua valoarea 0
- dimensiunea tabelei trebuie să fie un număr prim, pentru a permite verificarea tuturor celulelor
- a doua funcție de dispersie **h_2(k)** este de forma:

h_2(k)=q-k%q în care **q** este prim și **q < M**

Metoda dispersiei duble generează cel mult o secvență de **MAX²** funcții de dispersie, dar nu produce nici *aglomerări primare* nici *secundare.* Factorul de încărcare **λ=n/MAX** afectează performanțele tabelei de dispersie.

Pentru verificarea liniară, funcțiile primitive sunt:

TD_Search (h, k, comp, hash)

- caută cheia **k** mai întâi în **h[hash(k)]**
- dacă nu o găsește, continuă căutarea în **h[hash(k)+i]** până când:
  - o găsește pe **k** - succes
  - dă peste o poziție liberă în tabelul de dispersie - eșec
  - a efectuat **MAX** tentative - eșec

TD_Remove (h, k, comp, hash)

- caută cheia **k** mai întâi în **H[hash(k)]**
- dacă găsește pe **k**, îi pune marcajul **STERS**, fără a-l șterge efectiv -succes.
- Dacă nu-l găsește, continuă căutare în **h[hash(k)+i]** până când:
  - îl găsește în proba **i** și îi pune marcajul **STERS** - succes
  - găsește o celulă liberă sau face **MAX** încercări - eșec

TD_Insert (h, k, i, comp, hash)

- caută cheia **k** mai întâi în **h[hash(k)]**
- dacă **k** este găsit, el nu va mai fi inserat
- dacă poziția poziția cercetată are marcajul **LIBER** sau **STERS** este pusă cheia în poziția respectivă și marcajul pe **OCUPAT**
- dacă poziția este **OCUPAT** se încearcă punerea elementului în poziția **h[hash(k)+i]**. Se fac cel mult **MAX** tentative.

Interfață dispersie închisă.

```
#ifndef _HT_probQ
#define _HT_probQ
struct  TabDisp;
typedef struct TabDisp *TD;
typedef int (*PFC)(void*, void*);
typedef int (*PFD)(void*, int);

TD    TD_New (int m);
int   TD_Search (TD h, void *k, PFC comp, PFD hash);
void  TD_Insert (TD h, void *k, void *inf, PFC comp, PFD hash);
void  TD_Remove (TD h, void *k, PFC comp, PFD hash);
void  TD_Delete (TD *h);
int   TD_Empty (TD h);
int   TD_Size (TD h);
TD    TD_ReDisp(TD h, PFC comp, PFD hash);
#endif
```

Implementare dispersie închisă.

În cazul dispersiei închise nu se face o ştergere efectivă a elementelor din tabela de dispersie, deoarece aceasta este costisitoare, necesitând deplasări de elemente, cu complexitate **O(MAX)**. Poziţiile din tabela de dispersie vor fi marcate cu valorile **OCUPAT** şi **LIBER**. La ştergerea unui element în tabela de dispersie, acesta nu va primi marcajul **LIBER**, deoarece s-ar întrerupe lanţul de valori aflate în coliziune, marcând în mod fals sfârşitul acestora. Astfel dacă am avea două elemente în coliziune, ele ar fi memorate unul după celălalt în tabela de dispersie. Dacă se şterge primul, marcându-l ca **LIBER**, o căutare a celui de-al doilea element ar testa mai întâi pe primul, ar găsi poziţia **LIBER** şi ar ajunge la concluzia greşită că elementul căutat nu există. Pentru ştergere se va folosi o altă valoare a marcajului: **STERS** care permite continuarea căutării.
Un element din tabela de dispersie va fi descris aşadar prin: cheie, valoare asociată şi marcaj de ocupare.

Un grup de elemente aflate în coliziune poate avea lungimea cel mult **MAX** sau se poate termina mai repede printr-o celulă marcată **LIBER**. Trecerea la următorul element din grup, în cazul verificării pătratice se face adunând la poziţia iniţială **p** pe $\mathbf{i}^2$ (sau adăugând **2*i-1** la poziţia curentă)

```
enum Marcaj{OCUPAT, LIBER, STERS};
struct Elem{
  void  *ch;
  void  *info;
  enum Marcaj mark;
};
struct TabDisp{
  int max;
  int n;
  struct Elem *TabCel;
};

TD TD_New(int m){
  TD h;
  int i;
  h=(TD)malloc(sizeof(struct TabDisp));
  h->max=nextPrim(m);
```

```
  h->n=0;
  h->TabCel=(struct Elem*)malloc(sizeof(struct Elem)*h->max);
  for(i=0; i<h->max; i++)
    h->TabCel[i].mark=LIBER;
  return h;
}
int TD_Search (TD h, void *k, PFC comp, PFD hash){
  int p;
  int i=0;
  p=hash(k, h->max);
  while(i < h->max &&
        h->TabCel[p].mark != LIBER &&
        comp(h->TabCel[p].ch, k)!=0)
  {
    p += 2* ++i - 1;
    if(p >= h->max)
      p -= h->max;
  }
  if(i > h->max || h->TabCel[p].mark==LIBER)
    return h->max; // esec - nu a gasit
  return p;        // succes
}
void TD_Insert(TD h, void *k, void *inf, PFC comp, PFD hash){
  int p = TD_Search (h, k, comp, hash);
  if(p < h->max) return; //cheia exista deja
  int i=0;
  while(i < h->max && h->TabCel[p].mark==OCUPAT)
  { p += 2* ++i - 1;
    if(p >= h->max)
      p -= h->max;
  }
  assert(i < h->max);
  h->TabCel[p]->ch = k;
  h->TabCel[p]->info = inf;
  h->TabCel[p].mark = OCUPAT;
  h->n++;
}
void TD_Remove (TD h, void *k, PFC comp, PFD hash){
  int p = TD_Search (H, k, comp, hash);
  if(p >= h->max) return; //nu exista cheia in TD
  int i=0;
  while(i < h->max &&
        h->TabCel[p].mark!=LIBER &&
        comp(h->TabCel[p].ch, k)!=0)
  {
    p += 2* ++i - 1;
    if(p >= h->max)
      p -= h->max;
  }
  if(h->TabCel[p].mark==OCUPAT &&
```

```
      comp(h->TabCel[p].ch, k)==0)   {
         h->TabCel[p].mark = STERS;
         h->n--;
   }
}
```

Redispersare.

Dacă factorul de încărcare al tabelei de dispersie depăşeşte 0.5, operațiile de inserare pot eşua. În această situație se preferă reconstruirea tabelei de dispersie cu dimensiune dublată. Schimbarea dimensiunii presupune recalcularea poziției ocupate de elemente în noua tabelă de dispersie. Operația este scumpă, cu complexitate **O(n)**, dar se face rar.

```
TD TD_ReDisp(TD h, PFC comp, PFD hash){
  struct Elem *TCv = h->TabCel;
  int n = h->n;
  int mv = h->max;
  h = TD_New (2*mv);
  h->n = n;
  int i;
  for(i=0; i<mv; i++)
    if(TCv[i].mark==OCUPAT)
      TD_Insert(h, TCv[i].ch, TCv[i].info, comp, hash);
  free(TCv);
  return h;
}
```

Eficiența operațiilor în Tabele de dispersie.

Este strict legată de numărul de coliziuni. Dacă numărul de elemente memorate se apropie de dimensiunea tabelei de dispersie (factorul de încărcare se apropie de 1) numărul de coliziuni creşte rapid.

Alegerea unei funcții de dispersie eficiente reduce numărul de coliziuni.
Dimensiunea tabelei de dispersie trebuie să fie număr prim.

Strategia de rezolvare a coliziunilor influiențează numărul acestora.

Distribuția cheilor influiențează de asemenea numărul de coliziuni. Este de dorit ca aceste chei să fie distribuite aleator.

Numărul mediu de comparații pentru o căutare, aplicând diferite strategii de rezolvare a coliziunilor:

Număr de comparații	Căutare cu succes	Căutare fără succes
Încercări liniare	$\frac{1}{2}\left(1 + \frac{1}{1 - \lambda}\right)$	$\frac{1}{2}\left(1 + \frac{1}{(1 - \lambda)^2}\right)$
Încercări pătratice	$-\frac{\ln(1 - \lambda)}{\lambda}$	$\frac{1}{1 - \lambda}$
Înlănțuire	$1 + \frac{\lambda}{2}$	λ

Probleme propuse.

Dați conținutul tabelei de dispersie obținută prin inserarea caracterelor **E X A M E N P A R T I A L**, în această ordine într-un tabel, inițial vid cu **M=5** liste, folosind înlănțuire separată cu liste neordonate.

Se foloseşte funcția de dispersie **11 k mod M** pentru a transforma litera cu numărul de ordine **k** din alfabet într-un indice în tabel , De exemplu, hash(I) = hash(9) = 99 % 5 = 4.

Dați conținutul tabelei de dispersie obținută prin inserarea caracterelor **E X A M E N P A R T I A L**, în această ordine într-un tabel, inițial vid de dimensiune **M=16** , folosind verificarea lineară.

Se foloseşte funcția de dispersie **11 k mod M** pentru a transforma litera cu numărul de ordine **k** din alfabet într-un indice în tabel , De exemplu, hash(I) = hash(9) = 99 % 5 = 4.

Dați conținutul tabelei de dispersie obținută prin inserarea caracterelor **E X A M E N P A R T I A L**, în această ordine într-un tabel, inițial vid de dimensiune **M=16** , folosind dispersia dublă. Use the hash function **11k mod M** pentru proba inițială şi funcția de dispersie secundară **(k mod 3) + 1**.