# Hashing. Aplicații

# *Cuprins*


- ***Introducere*..........................................................................3**
- ***Funcții hash*........................................................................4**
  - ***Metoda Diviziunii (restul împărțirii la m)*..................................4**
  - ***Metoda înmulțirii*...........................................................................4**
  - ***Metoda probabilistică (Hashing-ul universal)*..........................4**
- ***Interpretarea cheilor ca numere naturale*....................................5**
- ***Coliziunile*..........................................................................5**
  - ***Chaining (înlănţuire)*......................................................................5**
  - ***Open-addressing (adresare deschisă)*.........................................6**
- ***Distribuția cheilor*..........................................................................6**
  - ***Examinarea liniară*...........................................................................7**
  - ***Examinarea pătratică*.........................................................................7**
  - ***Hashing dublu*..........................................................................7**
- ***Hashing-ul perfect*..........................................................................7**
- ***Hash-uri în STL*..........................................................................8**
- ***Aplicații*..........................................................................8**
  - ***Algoritmul Rabin - Karp*..........................................................................8**
  - ***Alte probleme*..........................................................................9**
- ***Legături*..........................................................................9**
- ***Bibliografie*..........................................................................9**

## Introducere

Multe aplicații necesită o mulțime dinamică pentru care să se aplice numai operațiile specifice pentru dicționare: *inserează*, *caută*, *șterge*. O tabelă de dispersie (*tabelă hash*) este o structură eficientă de date pentru implementarea dicționarelor.

O tabelă de dispersie este o generalizare a noţiunii mai simple de tablou.

***Adresarea directă*** într-un tablou foloseşte abilitatea noastră de a examina o poziţie arbitrară în timp $O(1)$. Adresarea directă este aplicabilă numai în cazul în care ne permitem sa alocăm un tablou care are câte o poziţie pentru fiecare cheie posibilă.

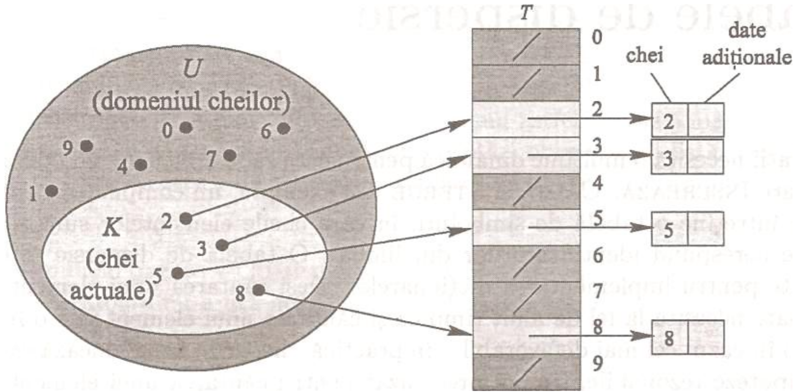


*Figura 1.*
*Implementarea unei mulţimi dinamice printr-un tablou cu **adresare directă** $T$.*
*Fiecare cheie din universul $U = \{0,1,2, \dots ,9\}$ corespunde unui indice în tablou.*
*Mulţimea $K = \{2,3,5,8\}$ a cheilor efective determină locaţiile din tablou care conţin pointeri către elemente.*
*Celelalte locaţii, haşurate mai întunecat, conţin $NIL$.*

Când numărul cheilor memorate efectiv este relativ mic faţă de numărul total de chei posibile**,** ***tabelele de dispersie*** devin o alternativă eficientă la adresarea directă într-un tablou, deoarece se foloseşte un tablou de mărime proporţională cu numărul de chei memorate efectiv.

În loc să se folosească direct cheia ca indice în tablou, indicele este calculat pe baza cheii, folosind o funcţie de dispersie (***hash***), care face maparea spaţiului de chei într-un spaţiu de adrese.

Funcţia generează o adresă printr-un calcul simplu, aritmetic sau logic asupra cheii sau asupra unei părţi a cheii.

Să considerăm $A$ o mulțime cu $n$ valori care trebuie stocate în structura de date (aceste valori sunt denumite chei) și $T$ un vector cu $m$ componente (denumit *tabel hash*).

Vom considera o ***funcție hash*** prin care se asociază fiecărei chei un număr întreg din intervalul $[0, m - 1]$:

$$h: A \to [0, m - 1]$$

Funcția hash are proprietatea că valorile ei sunt uniform distribuite în intervalul specificat. Aceste valori vor fi utilizate ca indici în vectorul $T$. Mai exact, cheia $x$ va fi plasată în tabela hash pe poziția $h(x)$.

Datorită faptului că spaţiul de chei este de obicei mult mai mare decât spaţiul de adrese, se poate întâmpla ca mai multe chei să aibă aceeaşi adresă. Aceasta se numeşte ***coliziune*** între înregistrări.

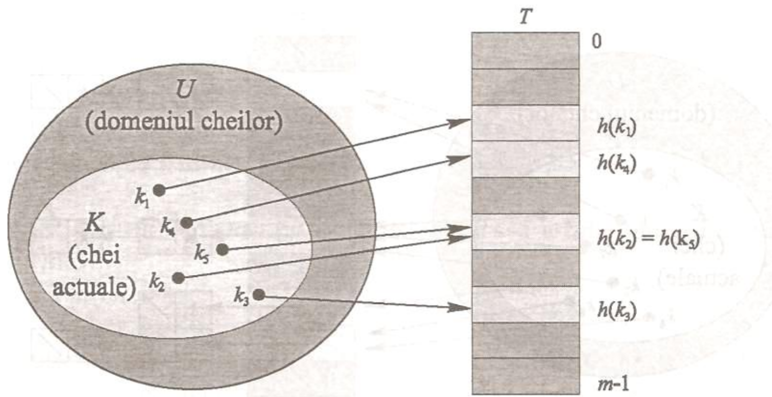


*Figura 2.*
*Folosirea unei **funcţii de dispersie h** pentru a transforma chei în poziţii din tabela de dispersie. Cheile $k_2$ şi $k_5$ se transformă în aceeaşi poziţie, deci sunt în coliziune.*

## Funcții hash

### Metoda Diviziunii (restul împărțirii la m)

$$h(x) = x \,\%\, m$$

În acest caz este indicat să evităm ca m să fie de forma $2^p$ (pentru că atunci $h(x)$ ar avea ca valoare ultimii $p$ biți ai lui $x$). Exceptând cazul în care știm că toate secvențele binare de lungime $p$ apar cu acceași probabilitate ca ultimi $p$ biți ai cheii, este de preferat să utilizăm o funcție în care se utilizează toți biții cheii.

Un număr prim apropiat de o putere a lui 2 este adesea o bună alegere pentru $m$. Se alege un număr prim pentru ca înmulțirea a doi termeni care au suferit operația modulo să dea rezultat nul numai daca unul dintre aceștia este nul.

### Metoda înmulțirii

$$h(x) \;=\; [m * (frac(x * y)]$$

Se înmulțește cheia $x$ cu un număr subunitar $0 < y < 1$, considerăm partea fracționară a lui $x * y$, o înmulțim cu $m$ și reținem doar partea întreagă a rezultatului.

Un avantaj al acestei metode este că nu există valori particulare pentru $m$ care să trebuiască evitate ca la *Metoda Diviziunii*, prin urmare de obicei $m$ poate fi o putere a lui 2, pentru a implementa eficient funcția hash.

### Metoda probabilistică (Hashing-ul universal)

Dacă un adversar "răutăcios" va alege cheile astfel încât toate să aibă aceeași valoare pentru o anumită funcție hash, atunci obținem un timp de execuție liniar. Soluția ar fi să alegem o funcție hash independentă de cheile ce urmează a fi stocate.

Ideea de bază este să selectăm funcția hash la întâmplare dintr-o listă predefinită de funcții hash la începutul execuției programului. Se numește *familie universală de funcții hash* o mulțime $H$ de funcții definite pe $A$ cu valori în $\{0, 1, \dots, m-1\}$ astfel încât pentru orice pereche de chei $x, y$ ($x \neq y$) numărul funcțiilor hash din familie pentru care $h(x) = h(y)$ este cel mult $\frac{|H|}{m}$. Cu alte cuvinte, dacă alegem aleator o funcție din această familie, probabilitatea de obține o coliziune pentru cheile $x$ și $y$ este $< \frac{1}{m}$.

Un exemplu care se comportă foarte bine în practică este următorul: se generează aleator un număr natural impar $r$. Valoarea funcției pentru un întreg $k$ se află înmulțind $k$ cu $r$ și păstrând cei mai semnificativi $p$ biți ai rezultatului (înmulțirea se face pe *32 de biți*, și ignorăm *overflow*).

## ***Interpretarea cheilor ca numere naturale***

Majoritatea funcţiilor de dispersie presupun universul cheilor din mulţimea $\mathbb{N} = \{0,1,2,\dots\}$ a numerelor naturale. Astfel, dacă cheile nu sunt numere naturale, trebuie găsită o modalitate pentru a le interpreta ca numere naturale.

De exemplu, o cheie care este un ***şir de caractere*** poate fi interpretată ca un întreg într-o bază de numeraţie aleasă convenabil. Prin urmare, identificatorul $pt$ poate fi interpretat ca o pereche de numere zecimale $(112, 116)$, pentru că $p = 112$ şi $t = 116$ în mulţimea codurilor ASCII; atunci $pt$ exprimat ca un întreg în baza 128 devine $(112 \cdot 128) + 116 = 14452$.

În mod obişnuit, în orice aplicaţie se poate crea direct o asemenea metodă simplă de a interpreta fiecare cheie ca un număr natural (posibil mare).

## ***Coliziunile***

Așa cum spuneam și mai devreme, datorită faptului că spaţiul de chei este de obicei mult mai mare decât spaţiul de adrese, se poate întâmpla ca mai multe chei să aibă aceeaşi adresă. Aceasta se numeşte *coliziune* între înregistrări.

Prin urmare este necesar şi un mecanism de rezolvare a coliziunilor. Există două metode de rezolvare a coliziunilor.

### ***Chaining (înlănţuire)***

Toate valorile sinonime (care au aceeaşi valoare *hash)* vor fi plasate într-o listă înlănţuită. Astfel în vectorul $T$ pe poziţia $i$ vom reţine un pointer către începutul listei înlănţuite formate din toate valorile $x$ pentru care $h(x) = i$.

*Operaţia de inserare* se execută în $O(1)$ (se inserează valoarea $x$ la începutul listei $T[h(x)]$ ).

*Operaţia de căutare* presupune căutarea elementului cu cheia $x$ în lista înlănţuită $T[h(x)]$.

*Operaţia de stergere* presupune căutarea elementului cu cheia $x$ în lista înlănţuită $T[h(x)]$, apoi ştergerea din listă a acestuia.

Ultimele două operaţii au timpul de execuţie proporţional cu lungimea listei $T[h(x)]$.

Analizând complexitatea în cazul cel mai defavorabil, lungimea listei poate fi $O(n)$ (atunci când toate elementele sunt sinonime).

Analizând complexitatea în medie, observăm că lungimea listei depinde de cât de uniform distribuie funcţia *hash* cheile.

Dacă presupunem că funcţia *hash* va distribui uniform valorile, atunci lungimea medie a listei este $\frac{n}{m}$ (această valoare este denumită factor de încărcare a tabelei), deci timpul de execuţie va fi $O(1 + n/m)$.

### *Open-addressing (adresare deschisă)*

În tabela *hash* nu sunt memoraţi pointeri, ci sunt memorate elementele mulţimii $A$. Astfel în tabelă se vor afla poziţii ocupate cu elemente din $A$ şi poziţii libere (acestea vor fi marcate cu o valoare specială pe care o vom denumi $NIL$).

În acest caz, pentru a insera un element $x$ în tabela *hash* se vor executa mai multe încercări, până când determinăm o poziţie liberă, în care putem plasa elementul $x$. Poziţiile care se examinează depind de cheia $x$.

În acest caz, funcţia *hash* va avea doi parametri: cheia $x$ şi numărul încercării curente (numerotarea încercărilor va începe de la 0 ):

$$h: A \times \{0, 1, \dots, m-1\} \to \{0, 1, \dots, m-1\}$$

*Operaţia de inserare:* Pentru a insera elementul $x$ în tabela $T$ se examinează în ordine poziţiile $h(x, 0)$, $h(x, 1)$, ..., $h(x, m-1)$. Această secvenţă de poziţii trebuie să fie o permutare a mulţimii $\{0, 1, \dots, m-1\}$.

*Operaţia de căutare:* Căutarea unui element în tabelă presupune examinarea aceleiaşi secvenţe de poziţii ca şi inserarea. Dacă în tabel nu se efectuează ştergeri, căutarea se termină fără succes la întâlnirea primei poziţii libere.

*Operaţia de ştergere:* Ştergerea unui element dintr-o tabelă hash este dificilă. Nu putem doar marca poziţia $i$ pe care este plasată cheia $x$ ca fiind liberă. În acest mod, căutare oricărei chei pentru care pe parcursul inserării am examinat poziţia $i$ şi am găsit-o ocupată se va termina fără succes.

O soluţie ar fi ca la ştergerea unui element pe poziţia respectivă să nu plasăm valoarea $NIL$, ci o altă valoare specială (pe care o vom denumi $DELETED$). Funcţia de inserare poate fi modificată astfel încât să insereze elementul $x$ pe prima poziţie pe care se află $NIL$ sau $DELETED$.

Dacă alegem această soluţie, timpul de execuţie al căutării nu mai este proporţional cu factorul de încărcare a tabelei.

Din acest motiv, tehnica de rezolvare a coliziunilor prin înlănţuire este mai utilizată pentru cazul în care structura de date suportă şi operaţii de ştergere.

## ***Distribuţia cheilor***

Pentru ca distribuţia cheilor să fie uniformă, funcţia *hash* trebuie să genereze cu egală probailitate oricare dintre cele $m!$ permutări ale mulţimii $\{1, 2, \dots, m-1\}$.

Pentru a determina secvenţa poziţiilor de examinat se utilizează 3 tehnici. Fiecare dintre aceste 3 tehnici garantează că pentru orice cheie secvenţa poziţiilor de examinat este o

permutare a mulţimii $\{0,1,\ldots,m-1\}$. Nici una dintre acestea nu asigură o distribuţie perfect uniformă.

### Examinarea liniară

Considerând o funcţie *hash* "obişnuită" $h': U \rightarrow \{0,1,\ldots,m-1\}$ (denumită funcţie *hash* auxiliară), tehnica examinării liniare utilizează funcţia:

$$h(x,i) = (h'(x) + i) \% m$$

*Observaţii*

- Deoarece prima poziţie din secvenţă determină întreaga secvenţă de poziţii de examinat, deducem că există doar m secvenţe de examinare distincte.
- Examinarea liniară este uşor de implementat, dar generează un fenomen denumit *primary clustering* (secvenţa de poziţii ocupate devine mare, şi astfel timpul mediu de execuţie al căutării creşte).

### Examinarea pătratică

Examinarea pătratică utilizează o funcţie *hash* de forma:

$$h(x,i) = (h'(x) + c_1 i + c_2 i^2) \% m$$

unde $h'$ este funcţia *hash* auxiliară, iar $c_1$ şi $c_2 \neq 0$ sunt constante auxiliare.

*Observaţie*

- Această metodă funcţionează mai bine decât examinarea liniară, dar pentru a garanta faptul că întreaga tabelă este utilizată, $m$, $c_1$ şi $c_2$ trebuie să îndeplinească anumite condiţii.

### Hashing dublu

Hashing-ul dublu utilizează o funcţie de forma:

$$h(x,i) = \big(h_1(x) + i h_2(x)\big) \% m$$

unde $h_1$ şi $h_2$ sunt funcţii *hash* auxiliare.

Pentru ca întreaga tabelă să fie examinată valoarea $h_2(x)$ trebuie să fie realtiv primă cu $m$. O metodă sigură este de a alege pe $m$ o putere a lui 2, iar $h_2$ să producă numai valori impare. Sau $m$ să fie un număr prim, iar $h_2$ să producă numere $< m$.

*Observaţie*

- Hashingul dublu este mai eficient decât cel liniar sau pătratic.

## Hashing-ul perfect

Există aplicaţii în care mulţimea cheilor nu se schimbă (de exemplu, cuvintele rezervate ale unui limbaj de programare, numele fişierelor scrise pe un CD, etc). Hashing-ul poate fi utilizat şi pentru probleme în care mulţimea cheilor este statică (odată ce au fost inserate cheile în tabela *hash,* aceasta nu se mai schimbă). În acest caz, timpul de execuţie în cazul cel mai defavorabil este $O(1)$.

Hashing-ul se numeşte *hashing perfect* dacă nu există coliziuni.

*Hashing-ul perfect* se realizează pe două niveluri.

**La primul nivel**, este similar cu hashing-ul cu înlănţuire: cele $n$ chei vor fi distribuite cu ajutorul unei funcţii *hash* $h$ în $m$ locaţii.

Dar în loc de a crea o listă înlănţuită cu toate cheile distribuite în locaţia $i$, la cel de **al doilea nivel** vom utiliza câte o tabelă *hash* secundară $S_i$ care are asociată o funcţie *hash* $h_i$ pentru fiecare locaţie în care există coliziuni.

Pentru a garanta faptul că nu vor exista coliziuni în tabela *hash* secundară, dimensiunea $m_i$ a tabelei *hash* secundare $S_i$ trebuie să fie cel puţin egală cu ${n_i}^2$ (unde $n_i$ este numărul de chei care sunt distribuite în locaţia $i$) .

Evident, descrierea grupurilor de la primul nivel trebuie să conţină şi funcţia *hash* aleasă pentru grupul respectiv, precum şi dimensiunea spaţiului de memorie alocat tabelei hash secundare.

O alegere a funcţiei hash $h$ dintr-o familie universală de funcţii hash asigură că dimensiunea totală a spaţiului de memorie utilizat este $O(n)$.

## *Hash-uri în STL*

- *hash_map*
- *hash_multimap*

(Pentru mai multe detalii consultaţi documentaţia STL sau cplusplus)

## *Aplicaţii*

### *Algoritmul Rabin - Karp*

Fie $T$ un şir de $n$ caractere şi $P$ un pattern de $m$ caractere.
Să se verifice dacă $P$ apare sau nu ca subsecvenţă în şirul $T$.

***Soluţie:***

Fie $b$ numărul de caractere distincte care apar în $T$. Un şir de lungime $m$ poate fi considerat un număr în baza $b+1$ având $m$ cifre (se folosesc cifre de la 1 la $b$).

Se asociază pattern-ului $P$ o valoare $p$ (şirul de caractere este considerat ca un număr întreg în baza $b+1$*)*.

Pentru fiecare secvenţă de $m$ caractere din şirul $T$ se calculează $t_h$, valoarea în baza $b+1$ a subsecvenţei din $T$ care începe la poziţia $i$ ($T[i \dots i+m-1]$). Evident, $p = t_i$ dacă şi numai dacă $T[i \dots i+m-1]$.

Valorile $t_i$ pentru i= $1, \dots\ i+m-1$ se pot calcula astfel:

$$t_{i+1} = (b+1) * (t_i - (b+1)^{m-1} * T[i]) + T[i+m] \ .$$

Singura dificultate constă în faptul că $p$ *şi* $t_i$ pot fi numere foarte mari. Prin urmare, operaţiile cu ele nu se execută în timp constant.

Se va lucra cu $p$ şi $t_k$ mod $Q$, unde $Q$ este un număr prim convenabil ales, aşadar se foloseşte o funcţie de dispersie. Pot apărea probleme deoarece $p \bmod Q = t_i \bmod Q$ nu implică faptul că $p = t_i$ (apar coliziuni). Dar dacă $p \bmod Q \neq t_i \bmod Q$, atunci sigur $P \neq T[i \dots i+m-1]$.

Aşadar, algoritmul *Rabin - Karp* este unul probabilistic, eficienţa în detectarea potrivirii pattern-ului depinzând de funcţia de dispersie. Pentru a elimina potrivirile *"false"*, se poate aplica o verificare normală pentru fiecare potrivire detectată sau se pot folosi mai multe funcţii de dispersie.

### *Alte probleme*

- Hashuri *(Infoarena)*
- Loto *(Infoarena)*
- Eqs *(Infoarena)*
- Oite *(Infoarena)*
- Strmatch *(Infoarena)*
- Abc2 *(Infoarena)*
- String *(Infoarena)*
- Circular *(Campion)*
- Repeat *(Campion)*
- Registration system *(Codeforces)*
- Ograzi *(Infoarena)*
- Rk *(Infoarena)*
- Patrate3 *(Infoarena)*
- Flori2 *(Infoarena)*
- Banana *(Infoarena)*
- Take5 *(Infoarena)*
- Radio2 *(Infoarena)*
- Petr# *(Codeforces)*

## ***Legături***

- **MIT** - Lecture 7: Hashing, Hash Functions
- **MIT** - Lecture 8: Universal Hashing, Perfect Hashing

## ***Bibliografie***


- Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest, *"Introducere în algoritmi"*, Editura Computer Libris Agora, Cluj Napoca, 2000
- Emanuela Cerchez, Marinel Şerban, *"Programarea în limbajul C/C++ pentru liceu"* volumul III, Editura Polirom, Iaşi, 2006
- Dana Lica, Mircea Paşoi, *"Fundamentele programării"* volumul III, Editura L&S SOFT, Bucureşti, 2006.



*Cohal Alexandru*
*alexandru.c04@gmail.com*
*alexandru.cohal@yahoo.com*


*Ianuarie 2012*