

# TABELA DE DISPERSIE

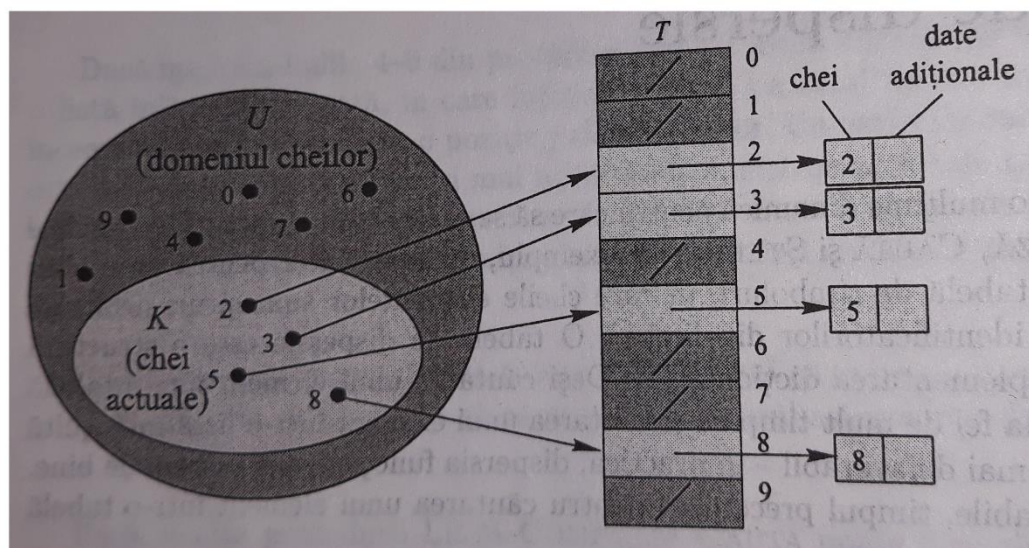
## Hash Table

- Este o structură de date **eficientă** pentru implementarea dicționarilor (și nu numai).
- Exemplu: un compilator păstrează o **tabelă de simboluri**, în care cheia este șirul de caractere corespunzător unui identificator
- TD poate fi folosită pentru implementarea containerelor pe care operațiile specifice sunt: **adăugare** element, **căutare** element, **ștergere** element. Ex: dicționare, colecții, mulțimi
  - JAVA
    - HashMap (dicționar reprezentat folosind o tabelă de dispersie)
    - HashSet (mulțime reprezentată folosind o tabelă de dispersie)
  - STL
    - unordered\_set (mulțime reprezentată folosind o tabelă de dispersie)
    - unordered\_map (dicționar reprezentată folosind o tabelă de dispersie).
- TD este o generalizare a noțiunii mai simple de **tabelă cu adresare directă**
- **Notății**
  - $n$  – numărul de elemente din container
  - un element  $e$  din container este o pereche cheie ( $c$ ) – valoare ( $v$ ) (***TElement*** = ***TCheie***  $\times$  ***TValoare***)
  - $U$  – **domeniul** (universul) cheilor
  - $K$  – domeniul actual al cheilor (mulțimea cheilor efectiv memorate în container)

### Tabelă cu adresare directă

- Notății și presupuneri
  - Presupunem chei numere naturale, chei distincte
  - Domeniul cheilor  $U = \{0, 1, 2, \dots, m-1\}$  -  $m$  relativ mic
  - $K$  – domeniul actual al cheilor (mulțimea cheilor efectiv memorate în container)
- Tabela cu adresare directă este memorată sub forma unui vector  $T[0..m-1]$ 
  - Locația  $T[c]$  va corespunde cheii  $c$  (la acea locație se memorează cheia și datele adiționale asociate acesteia)
  - Dacă o cheie  $c \notin K$ , atunci  $T[c]$  va conține NIL (sau o valoare specială care marchează locație goală)
  - $T[c]$  poate memora un pointer spre elementul având cheia  $c$  sau chiar elementul (cheia și valoarea asociată)

## Exemplu



Considerăm următoarea reprezentare:

### **TElement**

$c$ : TCheie

$v$ : TValoare

### **TabelaAdresareDirecta**

$m$ : Întreg

$e$ : TElement[0.. $m-1$ ]

Cele trei operații (**căutare**, **adăugare**, **ștergere**) pe o tabelă cu adresare directă sunt sumarizate mai jos:

### **CAUTĂ** ( $T, c$ )

//pre:  $T$  este o tabelă cu adresare directă,  $c$  este o cheie, de tip TCheie

@ returnează  $T.e[c]$

### **ADAUGĂ** ( $T, e$ )

//pre:  $T$  este o tabelă cu adresare directă,  $e$  este de tip TElement

@  $T.e[e.c] \leftarrow e$

### **ȘTERGE** ( $T, c$ )

//pre:  $T$  este o tabelă cu adresare directă,  $c$  este de tip TCheie

@  $T.e[c] \leftarrow \text{NIL}$

### ▪ **Observații**

- o tabelă cu adresare directă funcționează bine dacă universul cheilor este mic
- complexitatea timp a operațiilor este  $\theta(1)$
- spațiul de memorare este  $\theta(|U|)$

### ▪ **Dezavantaje**

- dacă universul  $U$  este mare, memorarea tabelului  $T$  poate fi nepractică, sau chiar imposibilă, dată fiind memoria disponibilă.
- dacă mulțimea  $K$  este mică relativ la  $U$ , rămâne mult spațiu nefolosit  $\Rightarrow$  gestionare inefficientă a spațiului de memorare.

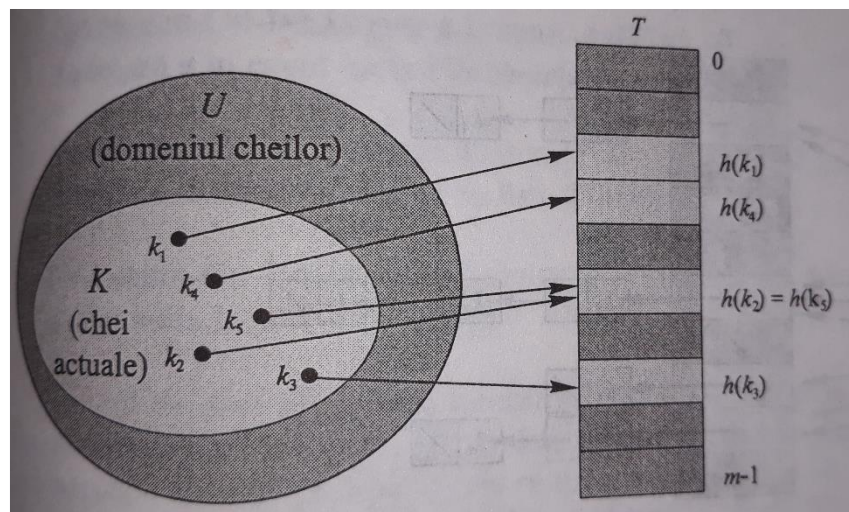
## PROBLEMĂ

Sugerați cum se poate implementa o tabelă cu adresare directă în care cheile elementelor memorate nu sunt neapărat distincte și elementele pot avea date adiționale.

## Tabela de dispersie

- $T[0..m-1]$ 
  - $m$  – număr locații din tabelă
- reduce spațiul de memorare la  $\theta(|K|)$  - eficientizare a spațiului de memorare (mai ales când  $K$  este mult mai mică decât )
- complexitatea timp *medie* pentru toate operațiile pe TD (adăugare, căutare, ștergere) este  $\theta(1)$ .
  - **căutarea** unui element într-o TD poate necesita  $\theta(n)$  în caz *defavorabil* (ca și căutarea în liste)
    - în practică, dispersia funcționează foarte bine
    - timpul *mediu* preconizat pentru căutarea este  $\theta(1)$
- se definește o **funcție de dispersie** (*hash function*)  $d: U \rightarrow \{0, 1, \dots, m-1\}$ 
  - $d(c)$  este **valoarea de dispersie** a cheii  $c$
  - vom spune  **$C$  se dispersează** în locația  $d(c)$
- dacă două chei  $c_1$  și  $c_2$  se dispersează în aceeași locație, adică  $d(c_1) = d(c_2)$ , spunem că avem o **coliziune**
  - evitarea totală a coliziunilor este imposibilă
    - deoarece  $|U| > m$ , sigur există două chei care să fie în coliziune
  - minimizare numărului de coliziuni
    - printr-o alegere potrivită a funcției de dispersie

**Exemplu** În figura de mai jos, cheile sunt notate cu  $k$  (*keys*), iar funcția de dispersie prin  $h$  (*hashing function*).



- **dispersia perfectă** (*perfect hashing, perfect hash function*)

- fără coliziuni
  - când se cunosc cheile (mulțimea de chei este statică – ex. compilatoare)
- vom discuta în cursul 10
- cum se face **adăugarea unui nou element**  $e=(c, v)$ ?
  - se calculează locația de dispersie a cheii  $c$ ,  $i = d(c)$
  - dacă locația  $i$  este liberă, atunci se adaugă elementul la locația  $i$
  - dacă la locația  $i$  mai e memorat un alt element  $\Rightarrow$  **rezolvare coliziune**
    - 2 tipuri de metode de dispersie
      - **dispersia deschisă** (*open hashing*)
        - cheile sunt stocate în liste înlanțuite atașate celulelor unei TD.
      - **dispersia închisă** (*closed hashing*)
        - cheile sunt stocate în interiorul TD fără a utiliza liste înlanțuite.
    - 3 metode de rezolvare a coliziunilor
      - **prin liste independente (înlanțuire)**
        - dispersie deschisă
      - **prin liste întrepătrunse**
        - dispersie deschisă
      - **prin adresare deschisă**
        - dispersie închisă
- **funcție de dispersie bună**
  - este ușor de calculat (folosește operații aritmetice simple) -  $\theta(1)$
  - produce cât mai puține coliziuni.

### Interpretarea cheilor ca numere naturale

- Majoritatea funcțiilor de dispersie presupun universul cheilor din mulțimea numerelor naturale
- În cazul în care cheile nu sunt numere naturale, trebuie găsită o modalitate de a le interpreta ca numere naturale – o funcție care asociază fiecărei chei un număr natural (implementare ***hashCode:TCheie***  $\rightarrow \{0, 1, 2, \dots\}$ )
  - identificatorul **pt** poate fi interpretat ca un număr în baza **128** –  $(pt)_{128} = 112 \cdot 128 + 116 = 14452$ .
  - pentru un șir de caractere putem considera suma codurilor ASCII ale caracterelor.
  - ...
- În cazul în care în container elementele sunt de tip **TElement** (nu au asociată o cheie - ex. mulțime, colecție), ***hashCode:TElement***  $\rightarrow \{0, 1, 2, \dots\}$
- Pp. în cele ce urmează că avem chei naturale.

### Funcții de dispersie

- O funcție de dispersie bună satisface (aproximativ) *ipoteza dispersiei uniforme simple* (**Simple Uniform Hashing - SUH**): fiecare cheie se poate dispersa cu aceeași probabilitate în oricare din cele  $m$  locații.
  - $P(d(c) = j) = \frac{1}{m}, \forall j = 0, \dots, m - 1 \quad \forall c \in U$
  - $P(d(c_1) = d(c_2)) = \frac{1}{m}, \quad \forall c_1, c_2 \in U$

- dacă  $P(c)$  este probabilitatea de a alege cheia  $c$ , atunci  $\sum_{c:d(c)=j} P(c) = \frac{1}{m} \forall j = 0, 1, \dots, m-1$ 
  - în general, nu se poate verifica această condiție, deoarece nu se cunoaște distribuția de probabilitate  $P$
- dacă această ipoteză ar fi verificată, atunci se minimizează numărul de coliziuni
- în practică se pot folosi tehnici euristice pentru a crea funcții de dispersie care să se comporte bine.

## I. Metoda diviziunii

- Dispersia prin diviziune
- $d(c) = c \bmod m$
- Experimental: valori bune pentru  $m$  sunt numerele prime nu prea apropiate de puteri exacte ale lui 2 (ex: 13,...)
- $m=13$ 
  - $c=63 \Rightarrow d(c)=11$
  - $c=26 \Rightarrow d(c)=0$
- ex: pentru a reține  $n=2000$  șiruri de caractere (1 caracter = 8 biți)
  - 3 elemente, în medie, într-o coliziune
  - $\Rightarrow m=701$  (apropiat de  $2000/3$ , nu e apropiat de o putere a lui 2)
  - $\Rightarrow d(c) = c \bmod 701$

## II. Metoda înmulțirii

- $d(c) = [m \cdot (c \cdot A \bmod 1)]$  unde " $c \cdot A \bmod 1$ " reprezintă partea fracționară a lui  $c \cdot A$  ( $c \cdot A - [c \cdot A]$ )
- Valoarea lui  $m$  nu e critică (în general este o putere a lui 2)
- Knuth: valoarea optimă pentru  $A$  este  $\frac{\sqrt{5}-1}{2} \approx 0.6180339887$  (golden ratio-1)
  - $m = 13, A = 0.6180339887$  (Knuth)
  - $c=63 \Rightarrow d(c) = [13 * \text{frac}(63 * A)] = 12$
  - $c=52 \Rightarrow d(c) = [13 * \text{frac}(52 * A)] = 1$
  - $c=129 \Rightarrow d(c) = [13 * \text{frac}(129 * A)] = 9$

## III. Dispersia universală

- $c = \langle c_1, c_2, \dots, c_k \rangle$
- $d(c) = (\sum_{i=1}^k c_i \cdot x_i) \bmod m$  unde  $\langle x_1, x_2, \dots, x_k \rangle$  este o secvență de numere aleatoare fixate (selectate de-a lungul inițializării funcției de dispersie)
- apropiată de ipoteza SUH

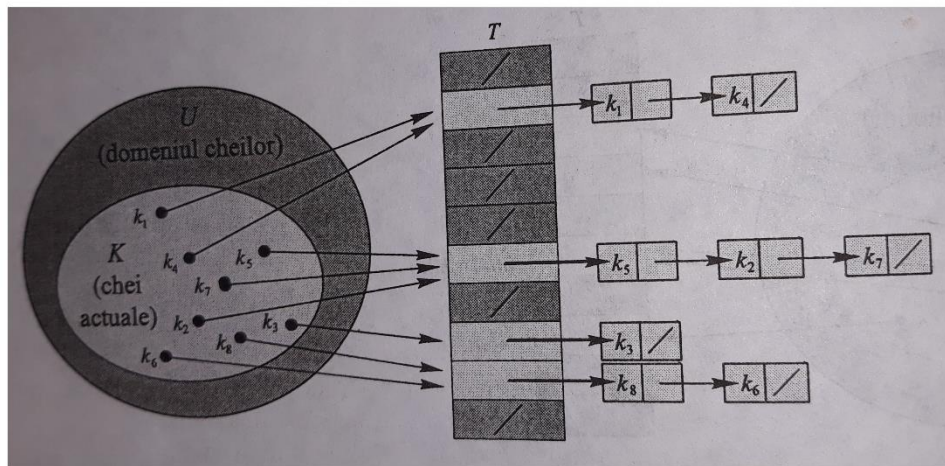
## Observație

- în cazul în care cheile nu sunt numere naturale, funcția de dispersie  $d$  (una din cele definite anterior) se definește nu pe cheia  $c$ , ci pe **hashCode**-ul acesteia
  - ex:  $d(c) = \text{hashCode}(c) \bmod m$

## A. Rezolvare coliziuni prin *liste independente (înlănțuire)* – SEPARATE CHAINING

- Elementele care se dispersează în aceeași locație (sunt într-o coliziune), vor fi puse într-o listă înlănțuită.
  - în general, alocare dinamică pentru memorarea înlănțuirilor
  - listele pot fi simplu sau dublu înlănțuite
- Locația  $j$  conține un pointer către capul listei înlănțuite a elementelor care se dispersează în locația  $j$  (dacă această listă e vidă, se memorează NIL).
- Operațiile sunt ușor de implementat.

**Exemplu** În figura de mai jos, cheile sunt notate cu  $k$  (*keys*), iar funcția de dispersie prin  $h$  (*hash function*).



Dacă  $m=10$ ,  $K=\{11, 21, 31, 5, 15, 7, 17, 27\}$ ,  $d(c) = c \bmod m$ , atunci

- $d(11)=d(21)=d(31)=1$
- $d(5)=d(15)=5$
- $d(7)=d(17)=d(27)=7$

și tabela va fi

0	
1	→ 11 → 21 → 31
2	
3	
4	
5	→ 5 → 15
6	
7	→ 7 → 17 → 27
8	
9	



## Reprezentare și operații

### **TElement**

$c$ :TCheie  
 $v$ :TValoare

### **Container**

$m$ :Întreg  
 $l$ :TListă[0.. $m$ -1]

- $d$  este funcția de dispersie,  $d:TCheie \rightarrow \{0, 1 \dots m-1\}$
- pp. cheia are o singură valoare asociată
- Container poate fi, de ex., dicționar, mulțime, colecție.
  - în cazul mulțimii/colecției, **TCheie=TElement** și nu există valoare asociată cheii.

### **CAUTĂ** ( $C, ch$ )

// pre:  $C$  este un container reprezentat sub forma unei TD (coliziuni prin înlanțuire),  $ch$  este de tip  
// TCheie

@ caută elementul cu cheia  $ch$  în lista  $C.l[d(ch)]$

### **ADAUGĂ** ( $C, e$ )

// pre:  $C$  este un container reprezentat sub forma unei TD (coliziuni prin înlanțuire),  $e$  este de tip  
// TElement

@ se adaugă elementul  $e$  în capul listei înlanțuite  $C.l[d(e.c)]$

### **ȘTERGE** ( $T, ch$ )

// pre:  $C$  este un container reprezentat sub forma unei TD (coliziuni prin înlanțuire),  $ch$  este de tip  
// TCheie

@ se șterge elementul cu cheia  $ch$  din lista înlanțuită  $C.l[d(ch)]$

## Observații

- Este posibil ca listele independente să fie memorate ordonat după cheie sau valoare
- Funcția de dispersie este considerată *bună* dacă listele au aproximativ aceeași lungime
- Dacă apar multe liste de vide sau liste prea lungi, se modifică  $m \Rightarrow$  redispersare (**rehashing**)

## Timp defavorabil pentru operații

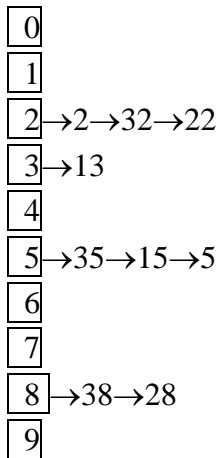
Pp  $n$  este numărul elementelor din container.

- **CAUTĂ** –  $O(n)$ 
  - toate elementele se dispersează în aceeași locație ( $\theta(n)$ ) – dacă elementul nu e găsit)
- **ADAUGĂ** –  $\theta(1)$ 
  - se poate adăuga la începutul listei înlanțuite
- **ȘTERGE** – presupune
  - (1) căutare nod în lista înlanțuită + (2) ștergere nod  $\Rightarrow O(n)$

### Exemplu

$m=10, d(c)=c \bmod m$

c	5	15	13	22	28	35	38	32	2
d(c)	5	5	3	2	8	5	8	2	2



### Iterator

- dacă listele sunt simplu înlănțuite cu alocare dinamică
- iteratorul va memora
  - o referință  $c$  către containerul reprezentat folosind o TD cu coliziuni prin liste independente
  - poziția curentă  $pozCurent$  din tabelă (indică lista înlănțuită iterată)
  - adresa unui nod (pointer)  $curent$  din lista înlănțuită de la poziția  $pozCurent$

#### Telement

$c$ :TChieie

$v$ :TValoare

#### Nod

$e$ :TElement

$urm$ :↑Nod

#### Container

$m$ :Întreg

$l$ :↑Nod [0..m-1]

#### IteratorContainer

$c$ :Container //referință (în implementare)

$pozCurent$ :Întreg

$curent$ :↑Nod

Operațiile pe iterator sunt descrise în Pseudocod, în continuare.

Pe lângă operațiile uzuale ale iteratorului (*creează*, *prim*, *valid*, *element*, *următor*), avem nevoie de o operație auxiliară **deplasare** care, dacă lista de la locația curentă  $pozCurent$  a fost iterată până la final ( $curent$  devine invalid), deplasează  $pozCurent$  pe următoarea locație din tabelă care conține o listă nevidă și poziționează  $curent$  pe primul nod din această listă.

- în exemplul anterior, dacă  $pozCurent=3$  și s-a terminat de iterat lista de la poziția 3, mută  $pozCurent$  pe 5, iar  $curent$  va indica 35.
- această operația NU va fi în interfața iteratorului (secțiunea publică), ci în implementare (secțiunea privată)

#### Subalgoritm deplasare ( $i$ ) este

{pre:  $i$ : IteratorContainer}

{post: deplasează iteratorul pe prima listă nevidă care urmează după locația  $pozCurent$ }

{incrementăm  $pozCurent$  cât timp nu s-a epuizat tabela și lista de la poziția  $pozCurent$  e vidă}



**CâtTimp** ( $i.pozCurent < i.c.m$ )  $\wedge$  ( $i.c.l[i.pozCurent] = \text{NIL}$ ) **execută**

$i.pozCurent = i.pozCurent + 1$

**SfCâtTimp**

{dacă nu s-a epuizat tabela}

**Dacă**  $i.pozCurent < i.c.m$  **atunci**

$i.curent \leftarrow i.c.l[i.pozCurent]$

**SfDacă**

**SfSubalgoritm**

**Subalgoritm creează** ( $i, c$ ) este

$i.c \leftarrow c$

$i.pozCurent \leftarrow 0$

{căutăm prima listă nevidă, pentru a poziționa iteratorul}

deplasare( $i$ )

**SfSubalgoritm**

**Subalgoritm prim** ( $i$ ) este

$i.pozCurent \leftarrow 0$

deplasare( $i$ )

**SfSubalgoritm**

**Funcția valid**( $i$ ) este

{locația curent iterată nu depășește numărul de locații din tabelă și nodul curent este valid }

**valid**  $\leftarrow (i.pozCurent < i.c.m) \wedge (i.curent \neq \text{NIL})$

**SfFuncție**

**Subalgoritm element** ( $i, e$ ) este

{pre:  $i$  este valid}

$e \leftarrow [i.curent].e$

**SfSubalgoritm**

**Subalgoritm urmator** ( $i$ ) este

{pre:  $i$  este valid}

$i.curent \leftarrow [i.curent].urm$

{dacă s-a terminat de iterat lista curentă, căutăm prima listă nevidă, pentru a repositiona iteratorul}

**Dacă**  $i.curent = \text{NIL}$  **atunci**

$i.pozCurent = i.pozCurent + 1$

deplasare( $i$ )

**SfDacă**

**SfSubalgoritm**

### Observație

- complexitatea iterării unui container cu  $n$  elemente, reprezentat folosind o TD cu  $m$  locații și liste independente este  $\theta(n + m)$

În directorul asociat cursului 8, găsiți implementarea parțială, în limbajul C++, a containerului **Colecție** (reprezentarea este sub forma unei TD în care coliziunile sunt reprezentate prin înlănțuire).

## Analiza dispersiei cu înlănțuire

### Notatii și presupuneri

- $\alpha = \frac{n}{m}$  *factorul de încărcare* al tabeli (numărul mediu de elemente memorate într-o înlănțuire)
- Pp. că timpul de calcul al funcției de dispersie este  $\theta(1)$  (!! la timpul de căutare se adaugă și timpul de calcul al funcției de dispersie)

- La **căutare** apar 2 cazuri
  - Căutare **cu succes** (găsim elementul)
  - Căutare **fără succes** (nu găsim elementul)

**Teorema 1.** Într-o TD în care coliziunile sunt rezolvate prin înlănțuire, în *ipoteza dispersiei uniforme simple* (SUH), o căutare **fără succes**, necesită, în *medie*, un timp  $\theta(1 + \alpha)$ .

În ipoteza SUH, fiecare listă are aceeași lungime,  $\alpha$ , iar o cheie se poate dispersa, cu aceeași probabilitate, în orice locație (poate fi în oricare dintre liste)

- (1) căutarea fără succes necesită iterarea unei liste  $\Rightarrow \alpha$
- (2) calcul funcției de dispersie  $\Rightarrow 1$
- Din (1) și (2)  $\Rightarrow \theta(1 + \alpha)$ .

**Teorema 2.** Într-o TD în care coliziunile sunt rezolvate prin înlănțuire, în *ipoteza dispersiei uniforme simple* (SUH), o căutare **cu succes**, necesită, în *medie*, un timp  $\theta(1 + \alpha)$ .

### Intuiție

- probabilitatea ca o cheie să se disperseze într-una din liste este  $\frac{1}{m}$
- în lista de pe poziția  $j$ , elementul poate fi găsit după 1, 2, ....  $\alpha$  pași  $\Rightarrow$  timpul mediu este aproximativ

$$\frac{1}{m} \sum_{j=1}^m \sum_{i=1}^{\alpha} \frac{i}{\alpha} \in \theta(1 + \alpha)$$

## CONCLUZII

- Dacă  $n = O(m) \Rightarrow \alpha = \frac{O(m)}{m} = O(1) \Rightarrow$  **căutarea** necesită, în *medie*, timp constant  $\theta(1)$
- Adăugarea necesită  $\theta(1)$
- Dacă listele sunt dublu înlănțuite atunci ștergerea unui nod se poate face în  $\theta(1)$

$\Rightarrow$  **TOATE OPERAȚIILE (adăugare, căutare, ștergere) POT FI EXECUTATE ÎN MEDIE ÎN  $\theta(1)$**

### Observații

- Pentru memorarea listele independente se pot folosi și arbori echilibrați, ceea ce va reduce complexitatea timp în caz defavorabil la căutare de la  $\theta(n)$  la  $\theta(\log_2 n)$ .

- Rezolvarea coliziunilor prin liste independente se mai numește și **dispersie deschisă** (*open hashing*) sau **adresare închisă** (*closed addressing*)
  - elemente sunt memorate în afara tabelui.

## **PROBLEME**

1. Presupunem că folosim o funcție de dispersie aleatoare  **$d$**  pentru a dispersa  $n$  chei distincte într-o tabelă  $T$  de dimensiune  $m$ . Care este numărul mediu de coliziuni? (cardinalul probabil al mulțimii  $\{(x, y) \in T \times T : d(x) = d(y)\}$ )
2. Presupunem că folosim o TD în care coliziunile sunt rezolvate prin înlănțuire (liste independente), dar fiecare listă este ordonată după cheie. Care va fi timpul de execuție pentru **căutare** (cu succes, fără succes), **adăugare** și **ștergere**?
3. Arătați că dacă  $|U| > n \cdot m$ , atunci există o submulțime a lui  $U$  de mărime  $n$  ce conține chei care se dispersează toate în aceeași locație, astfel încât timpul de căutare pentru dispersia cu înlănțuire, în cazul cel mai defavorabil, este  $\theta(n)$ .