TABELA DE DISPERSIE
Hash Table

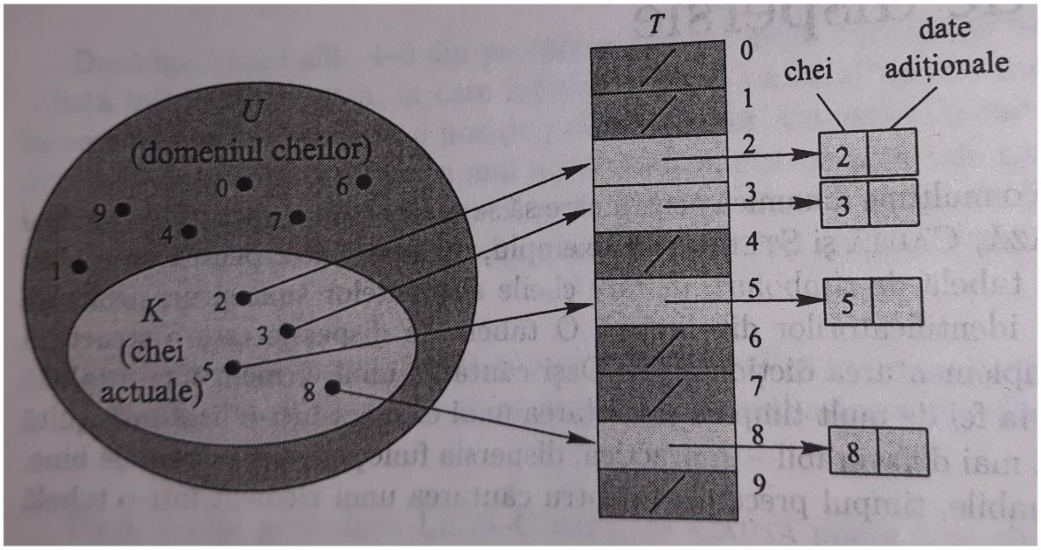
- Este o structură de date **eficientă** pentru implementarea dicționarelor (și nu numai).
- Exemplu: un compilator păstrează o **tabelă de simboluri**, în care cheia este șirul de caractere corespunzător unui identificator
- TD poate fi folosită pentru implementarea containerelor pe care operațiile 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**
- **Notații**
  - *n* – numărul de elemente din container
  - un element *e* din container este o pereche cheie (*c*) – valoare (*v*) ($\boldsymbol{TElement} = \boldsymbol{TCheie} \times \boldsymbol{TValoare}$)
  - *U* – **domeniul** (universul) cheilor
  - *K* – domeniul actual al cheilor (mulțimea cheilor efectiv memorate în container)

Tabelă cu adresare directă

- Notații și presupuneri
  - Presupunem chei numere naturale, chei distincte
  - Domeniul cheilor $U = \{0,1,2...,m-1\}$ - *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	**TabelaAdresareDirecta**
c:TCheie	*m*:Întreg
v:TValoare	*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*]←*e*

ȘTERGE (*T*, *c*)
//pre: *T* este o tabelă cu adresare directă, *c* este de tip TCheie
@ *T*.*e*[*c*]←NIL

- **Observații**
  - o tablelă cu adresare directă funcționează bine dacă universul cheilor este mic
  - complexitatea timp a operațiilor este $\theta(\mathbf{1})$
  - spațiul de memorare este $\theta(|\boldsymbol{U}|)$
- **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 ⇒ gestionare ineficientă a spațiului de memorare.

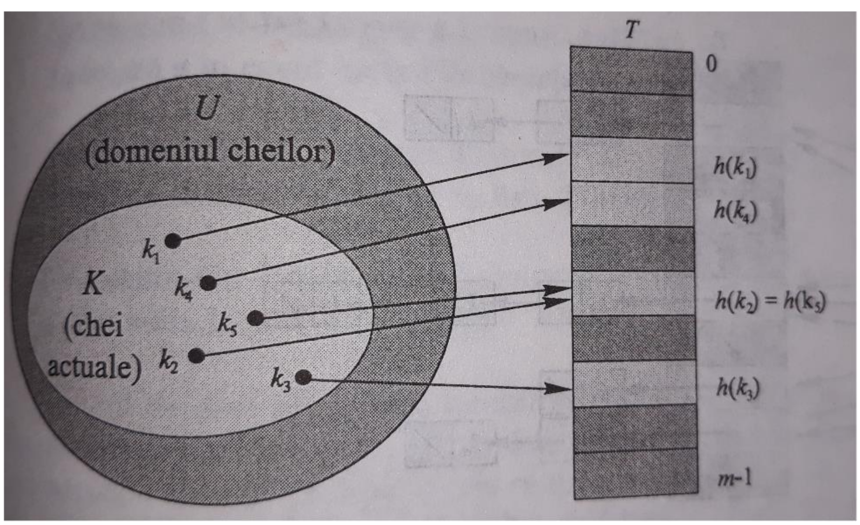
PROBLEMĂ

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

- $T[0..m\text{-}1]$
  - m – număr locații din tabelă
- reduce spațiul de memorare la $\theta(|\boldsymbol{K}|)$ - 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 $\theta(\mathbf{1})$.
  - **căutarea** unui element într-o TD poate necesita $\theta(\boldsymbol{n})$ în caz ***defavorabil*** (ca și căutarea în liste)
    - în practică, dispersia funcționează foarte bine
    - timpul ***mediu*** preconizat pentru căutare este $\theta(\mathbf{1})$
- se definește o **funcție de dispersie** (*hash function*) $\boldsymbol{d}: \boldsymbol{U} \rightarrow \{\mathbf{0}, \mathbf{1}, \ldots \boldsymbol{m} - \mathbf{1}\}$
  - $\boldsymbol{d}(\boldsymbol{c})$ este **valoarea de dispersie** a cheii c
  - vom spune că o cheie $\boldsymbol{c}$ se **dispersează** în locația $\boldsymbol{d}(\boldsymbol{c})$
- dacă două chei c_1 și c_2 se dispersează în aceeași locație, adică $\boldsymbol{d}(\boldsymbol{c_1}) = \boldsymbol{d}(\boldsymbol{c_2})$, spunem că avem o **coliziune**
  - evitatarea 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*, $\boldsymbol{i = d(c)}$
  - 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 înlănțuite atașate celulelor unei TD.
        - se mai numește **adresare închisă** (*close adressing*)
      - **dispersia închisă** (*closed hashing*)
        - cheile sunt stocate în interiorul TD fără a utiliza liste înlănțuite.
    - 3 metode de rezolvare a coliziunilor
      - **prin liste independente (înlănțuire)**
        - dispersie deschisă
      - **prin liste întrepătrunse**
        - combinație între *liste independente* și *adresare deschisă*
      - **prin adresare deschisă**
        - dispersie închisă
- **funcție de dispersie bună**
  - este ușor de calculat (folosește operații aritmetice simple) - $\theta(\mathbf{1})$
  - 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 -$\boldsymbol{hashCode}: \boldsymbol{TCheie} \rightarrow \{\mathbf{0}, \mathbf{1}, \mathbf{2} \ldots\}$)
  - identificatorul **pt** poate fi interpretat ca un număr în baza **128** – $(\mathbf{pt})_{128}$=112·128+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), $\boldsymbol{hashCode}: \boldsymbol{TElement} \rightarrow \{\mathbf{0}, \mathbf{1}, \mathbf{2} \ldots\}$
- 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.
  - $\boldsymbol{P(d(c) = j) = \frac{1}{m}, \forall j = 0, \ldots, m-1 \quad \forall c \in U}$
    - $\boldsymbol{P\left(d(c_1) = d(c_2)\right) = \frac{1}{m}, \quad \forall\, c_1, c_2 \in U}$

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

I. Metoda diviziunii

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

II. Metoda înmulțirii

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

  m = 13, A = 0.6180339887 (Knuth)
    - o $c=63 \Rightarrow d(c) = [13 * \text{frac}(63 * A)] = 12$
    - o $c = 52 \Rightarrow d(c) = [13 * \text{frac}(52 * A)] = 1$
    - o $c=129 \Rightarrow d(c) = [13 * \text{frac}(129 * A)] = 9$

III. Dispersia universală

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

Observație

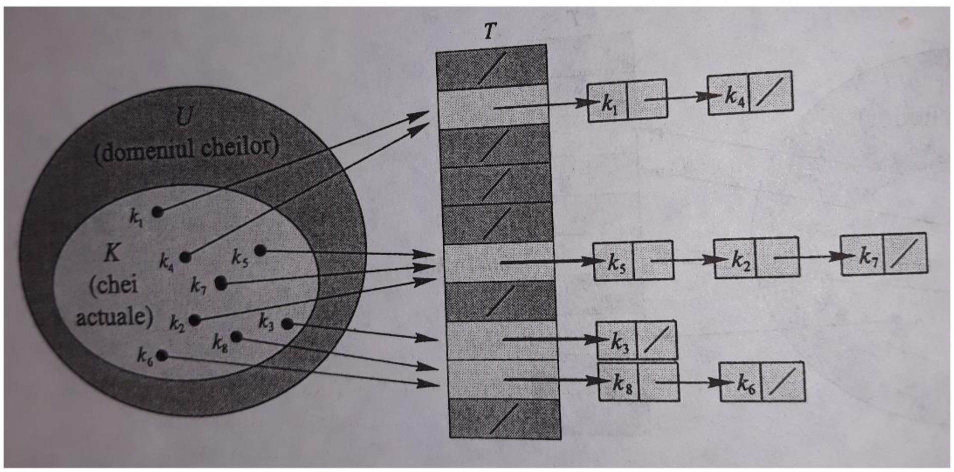
- o în cazul în care cheile nu sunt numere naturale, funcția de dispersie d (una din cele definite anterior) se definește nu pe cheia c, ci pe ***hashCode***-ul acesteia

➢ ex: $\boldsymbol{d(c) = hashCode(c) \, mod \, m}$

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 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	**Container**
c:TCheie	*m*:Întreg
v:TValoare	*l*:TListă[0..*m*-1]

- *d* este funcția de dispersie, $\boldsymbol{d}: \boldsymbol{TCheie} \rightarrow \{\mathbf{0}, \mathbf{1} \ldots \boldsymbol{m}-\mathbf{1}\}$
- 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 înlănțuire), *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 înlănțuire), *e* este de tip
// TElement
@ se adaugă elementul *e* în capul listei înlănțuite *C*.*l*[*d*(*e*.*c*)]

ȘTERGE (*T*, *ch*)
// pre: *C* este un container reprezentat sub forma unei TD (coliziuni prin înlănțuire), *ch* este de tip
// TCheie
@ se șterge elementul cu cheia *ch* din lista înlănțuită *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* ⇒ redispersare (**rehashing**)
    - dublare *m* (*m* « 1)

Timp defavorabil pentru operații

Pp *n* este numărul elementelor din container.

- **CAUTĂ** – $O(n)$
    - toate elementele se dispersează în aceeași locație ($\boldsymbol{\theta}(\boldsymbol{n})$ – dacă elementul nu e găsit)
- **ADAUGĂ** – $\boldsymbol{\theta}(\mathbf{1})$
    - se poate adăuga la începutul listei înlănțuite
- **ȘTERGE** – presupune
    - (1) căutare nod în lista înlănțuită + (2) ștergere nod ⇒ $O(n)$

<u>Exemplu</u>

m=10, *d*(*c*)=*c* mod *m*

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

0
1
2→2→32→22
3→13
4
5→35→15→5
6
7
8→38→28
9

<u>Iterator</u>

- 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:TCheie
v:TValoare

Nod
e:TElement
urm:↑Nod

Container
m:Întreg
l:↑Nod [0..m-1]

IteratorContainer
c:Container //**referință (în implementare)**
pozCurent:Intreg
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*) ∧ (*i.c.l* [*i.pozCurent*] = NIL) **execută**
i.pozCurent = *i.pozCurent* + 1
SfCâtTimp
{dacă nu s-a epuizat tabela}
Dacă *i.pozCurent* < *i.c.m* **atunci**
i.curent ← *i.c.l* [*i.pozCurent*]
SfDacă
SfSubalgoritm

Subalgoritm creează (*i*, *c*) este
i.c ← *c*
i.pozCurent ← 0
{căutăm prima listă nevidă, pentru a poziționa iteratorul}
deplasare(*i*)
SfSubalgoritm

Subalgoritm prim (*i*) este
i.pozCurent ← 0
deplasare(*i*)
SfSubalgoritm

Funcția valid(*i*) este
{locația curent iterată nu depășește numărul de locații din tabelă și nodul curent este valid }
valid ← (*i.pozCurent* < *i.c.m*) ∧ (*i.curent* ≠ NIL)
SfFuncție

Subalgoritm element (*i*, *e*) este
{pre: *i* este valid}
e ← [*i.curent*].*e*
SfSubalgoritm

Subalgoritm urmator (*i*) este
{pre: *i* este valid}
i.curent ← [*i.curent*].*urm*
{dacă s-a terminat de iterat lista curentă, căutăm prima listă nevidă, pentru a repoziționa iteratorul}
Dacă *i.curent* = 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 $\boldsymbol{\theta}(\boldsymbol{n}+\boldsymbol{m})$

În directorul asociat cursului 8, găsiti 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

Notații și presupuneri

- $\boldsymbol{\alpha} = \frac{\boldsymbol{n}}{\boldsymbol{m}}$ *factorul de încărcare* al tabelei (numărul mediu de elemente memorate într-o înlănțuire)
- $\boldsymbol{\alpha} > \mathbf{1}$
- Pp. că timpul de calcul al funcției de dispersie este $\boldsymbol{\theta}(\mathbf{1})$ (!! 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 $\boldsymbol{\theta}(\mathbf{1}+\boldsymbol{\alpha})$.

În ipoteza SUH, fiecare listă are aceeași lungime, $\boldsymbol{\alpha}$**,** 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 $\Rightarrow \boldsymbol{\alpha}$
- (2) calcul funcției de dispersie $\Rightarrow \mathbf{1}$
- Din (1) și (2) $\Rightarrow \boldsymbol{\theta}(\mathbf{1}+\boldsymbol{\alpha})$.

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 $\boldsymbol{\theta}(\mathbf{1}+\boldsymbol{\alpha})$.

Intuiție

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

$$\frac{\mathbf{1}}{\boldsymbol{m}}\sum_{\boldsymbol{j}=\mathbf{0}}^{\boldsymbol{m}-\mathbf{1}}\sum_{\boldsymbol{i}=\mathbf{1}}^{\propto}\frac{\boldsymbol{i}}{\propto} \in \boldsymbol{\theta}(\mathbf{1}+\boldsymbol{\alpha})$$

CONCLUZII

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

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

Observații

- Pentru memorarea listele independente se pot folosi și arbori echilibrați, ceea ce va reduce complexitatea timp în caz defavorabil la căutare de la $\theta(\boldsymbol{n})$ la $\boldsymbol{\theta}(\boldsymbol{log_2 n})$.
- Rezolvarea colizunilor prin liste independente se mai numește și **dispersie deschisă** (*open hashing*) sau **adresare închisă** (*closed adressing*)
  - elemente sunt memorate în afara tabelei.

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 $\{(x, y) \in TCheie \times TCheie: d(x) = d(y)\}$)
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 $|\boldsymbol{U}| > \boldsymbol{n} \cdot \boldsymbol{m}$**,** atunci există o submulțime a lui U de mărime *n* ce conține chei care se dispersează toate în aceeași locatie, astfel încât timpul de căutare pentru dispersia cu înlănțuire, în cazul cel mai defavorabil, este $\boldsymbol{\theta}(\boldsymbol{n})$**.**