

TABELA DE DISPERSIE

- continuare -

C. Rezolvare coliziuni prin adresare deschisă – OPEN ADDRESSING

- Toate elementele sunt memorate în interiorul tabelii, nu există liste memorate în afara tabelii.
 - Se mai numește și **dispersie închisă** (*closed hashing*).
 - **rezolvarea coliziunilor prin liste întrepătrunse** (variante B de rezolvare a coliziunilor) este o combinație între *open addressing* și *separate chaining*
 - Fiecare intrare în tabelă conține fie un element al containerului, fie un marcaj pentru locație liberă (ex. NIL).
 - Nu se folosesc pointeri pentru înlănțuiri.
 - Factorul de încărcare este subunitar $\alpha < 1$, altfel tabela este plină
 - Dezavantaj: tabela se poate umple ($\alpha = 1$). Soluție: se crește m , ceea ce presupune redispersarea elementelor.
 - Avantaj: spațiul de memorie suplimentar (nu se memorează pointeri) oferă tabelii un număr mai mare de locații pentru același spațiu de memorie, putând rezulta coliziuni mai puține și acces rapid.
 - Secvența de locații care se examinează nu se determină folosind *pointeri*, ci se *calculează*
 - La **adăugare** în tabelă, se examinează succesiv locațiile, până se găsește o locație liberă în care să se adauge cheia (elementul). În loc să fie fixată ordinea de verificare a tabelii (ex: 0,1,2,...,m-1, ca la vectori) care ar necesita timp $\theta(m)$, secvența de poziții examinate depinde de cheia (elementul) care se inserează.
- Se extinde funcția de dispersie $d: U \times \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$
 - al doilea argument al funcției se numește *număr de verificare*
 - Pentru o cheie $c \in U$ secvența $\langle d(c, 0), d(c, 1), \dots, d(c, m-1) \rangle$ se numește *secvența de verificare* a cheii c

Cerință

- secvența de verificare a oricărei chei $c \in U$ $\langle d(c, 0), d(c, 1), \dots, d(c, m-1) \rangle$ trebuie să fie o permutare a mulțimii $\langle 0, 1, \dots, m-1 \rangle$
 - pentru ca la adăugarea oricărei chei să fie verificate toate locațiile din tabelă

Ipoteza dispersiei uniforme simple (SUH)

- Pentru orice cheie $c \in U$, permutarea $\langle d(c, 0), d(c, 1), \dots, d(c, m-1) \rangle$ poate să apară sub forma oricărei permutări a $\langle 0, 1, \dots, m-1 \rangle$

Sunt 3 metode pentru a stabili secvența de verificare a unei chei c , a.î. să se asigure faptul că $c \in U$ $\langle d(c, 0), d(c, 1), \dots, d(c, m-1) \rangle$ este o permutare a mulțimii $\langle 0, 1, \dots, m-1 \rangle$. Acestea vor fi descrise, în continuare.

C.1. Verificare liniară – LINEAR PROBING

$$d(c, i) = (d'(c) + i) \bmod m \quad \forall i = 0, 1, \dots, m - 1$$

- $d': U \rightarrow \{0, \dots, m - 1\}$ este o funcție de dispersie uzuală (ex: $d'(c) = c \bmod m$)

- Fiind dată o cheie c , secvența ei de verificare este $\langle d'(c), d'(c) + 1, d'(c) + 2, \dots, m - 1, 0, 1, \dots, d'(c) - 1 \rangle$
- Problema: **grupare primară** – se formează șiruri lungi de locații ocupate, crescând timpul mediu de căutare

EXEMPLU 1. Adresare deschisă cu verificare liniară

- $m=10$
- $d(c, i) = (d'(c) + i) \bmod m \quad \forall i = 0, 1, \dots, m - 1$
- $d'(c) = c \bmod m$

c	5	15	13	22	20	35	30	32	2
d'(c)	5	5	3	2	0	5	0	2	2

Pas 1. Inițializare

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Pas 2. Adăugăm cheia 5

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	NIL	NIL	NIL	NIL	NIL	5	NIL	NIL	NIL	NIL

Pas 3. Adăugăm cheia 15

- Secvența de verificare a cheii este $\langle 5, 6, 7, 8, 9, 0, 1, 2, 3, 4 \rangle$
- Prima locație liberă din secvență este locația 6, acolo se va adăuga

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	NIL	NIL	NIL	NIL	NIL	5	15	NIL	NIL	NIL

....

La final, tabela va fi

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	20	30	22	13	32	5	15	35	2	NIL

C.2. Verificare pătratică – QUADRATIC PROBING

$$d(c, i) = (d'(c) + c_1 \cdot i + c_2 \cdot i^2) \bmod m \quad \forall i = 0, 1, \dots, m-1$$

$d': U \rightarrow \{0, \dots, m-1\}$ este o funcție de dispersie uzuală (ex: $d'(c) = c \bmod m$), $c_1 \neq 0$ și $c_2 \neq 0$ sunt constante auxiliare fixate la inițializarea funcției de dispersie.

- Constantele $c_1 \neq 0$ și $c_2 \neq 0$ se pot determina euristic
 - Pentru a asigura faptul că secvența de verificare este o permutare $\{0, \dots, m-1\}$
 - m se alege putere a lui 2 și $c_1 = c_2 = 0.5$
 - m se alege număr prim de forma $4 \cdot k + 3$, $d'(c) = c \bmod m$ și $c_1 = 0$, $c_2 = (-1)^i$
- Fiind dată o cheie c , prima poziție examinată este $d'(c)$, după care următoarele poziții examinate sunt decalate cu cantități ce depind într-o manieră pătratică de locația anterior examinată.
- Problema: **grupare secundară** – dacă 2 chei au aceeași poziție de start a verificării, atunci secvența lor de verificare coincide (dacă $d(c', 0) = d(c'', 0) \Rightarrow d(c', i) = d(c'', i) \quad \forall i = 0, 1, \dots, m-1$)
- Experimental: funcționează **mai bine** decât **verificarea liniară**

EXEMPLU 2. Adresare deschisă cu verificare pătratică

- $m=8$
- $d(c, i) = (d'(c) + c_1 \cdot i + c_2 \cdot i^2) \bmod m \quad \forall i = 0, 1, \dots, m-1$
- $d'(c) = c \bmod m$
- $c_1 = c_2 = 0.5$

c	5	15	13	2	0	32
d'(c)	5	7	5	2	0	0

Pas 1. Inițializare

Indice	0	1	2	3	4	5	6	7
Cheie	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Pas 2, 3. Adăugăm cheile 5, 15

Indice	0	1	2	3	4	5	6	7
Cheie	NIL	NIL	NIL	NIL	NIL	5	13	15

Pas 4 Adăugăm cheia 13

- Secvența de verificare a cheii este <5, 6, ...>
- Prima locație liberă din secvență este locația 6, acolo se va adăuga

.....

La final, tabela va fi

Indice	0	1	2	3	4	5	6	7
Cheie	0	32	2	NIL	NIL	5	13	15

C.3. Dispersia dublă – DOUBLE HASHING

$$d(c, i) = (d_1(c) + i \cdot d_2(c)) \bmod m \quad \forall i = 0, 1, \dots, m-1$$

d_1 și d_2 sunt funcții de dispersie aleatoare.

- Este considerată una dintre cele mai bune metode disponibile pentru adresarea deschisă
- Fiind dată o cheie c , prima poziție examinată este $d_1(c)$, după care următoarele poziții examinate sunt decalate față de poziția anterioară cu $d_2(c) \bmod m$.
- $d_2(c)$ și m să fie prime între ele pentru a fi parcursă întreaga tabelă
- **Exemplu**
 - m prim
 - $d_1(c) = c \bmod m$ $d_2(c) = (1 + c \bmod m')$
 - m' se alege de obicei $m-1$ sau $m-2$
- Performanța dispersiei duble apare ca fiind foarte apropiată de performanța schemei ideale a dispersiei uniforme ($\theta(m^2)$ secvențe de verificare posibile pentru o cheie)
 - în cazurile C1 și C2, numărul secvențelor de verificare posibile pentru o cheie e doar $\theta(m)$

EXEMPLU 3. Adresare deschisă cu dispersie dublă

- $m=13$
- $d(c, i) = (d_1(c) + i \cdot d_2(c)) \bmod m \quad \forall i = 0, 1, \dots, m-1$
- $d_1(c)=c \bmod m, d_2(c)=1 + c \bmod (m-2)$

c	79	69	96	14
d₁(c)	1	4	5	1
d₂(c)	3	4	9	4

Adăugăm cheile 79, 69, 96 și tabela devine

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12
Cheie	NIL	79	NIL	NIL	69	96	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Pentru cheia 14, care e în coliziune, secvența de verificare e 1, 5, 9,

- prima poziție liberă e 9, adăugăm

Tabela finală e

Indice	0	1	2	3	4	5	6	7	8	9	10	11	12
Cheie	NIL	79	NIL	NIL	69	96	NIL	NIL	NIL	14	NIL	NIL	NIL

Presupuneri și notații:

- Pp. că în container memorăm doar chei
- O locație din tabelă va conține:
 - NIL (constantă simbolică) – dacă locația e liberă (nu conține o cheie)
 - O cheie din container

Reprezentarea containerului folosind o TD cu adresare deschisă

Container

m: Intreg {nr.de locații din tabelă}

ch: TCheie[0..m-1] {cheile din container}

d: TFuncție {funcția de dispersie asociată}

ADĂUGARE

Dacă adăugăm o cheie c , determinăm locația la care ar trebui memorată în tabelă ($i=d(c)$), după care vom avea două situații

- Locația i este liberă (NIL, în convenția noastră) \Rightarrow caz favorabil, memorăm cheia
- Locația i nu este liberă \Rightarrow avem coliziune
 - verificăm, pe rând, locațiile din tabelă, în ordinea dată de secvența de verificare $< d(c, 0), d(c, 1), \dots, d(c, m-1) >$
 - dacă găsim o locație liberă, adăugăm
 - dacă toate locațiile din secvența de verificare sunt ocupate \Rightarrow tabela este plină

Subalgoritmul ADAUGĂ (c, ch) este

{ c :Container, ch :TCheie}

$i \leftarrow 0$ {numărul de verificare}

$gasit \leftarrow \text{fals}$ {nu am găsit poziția de adăugare}

repetă

$j \leftarrow c.d(ch, i)$ {locația de verificat}

(*) dacă $c.ch[j] = \text{NIL}$ atunci

$c.ch[j] \leftarrow ch$ {memorez cheia}

$gasit \leftarrow \text{adev}$ {am găsit poziția unde putem adăuga}

altfel

$i \leftarrow i+1$ {căutăm mai departe pe secvența de verificare}

sfdacă

până_când ($i=c.m$) sau ($gasit$)

dacă $i=c.m$ atunci {tabela e plină}

@ depășire tabelă

Sfdacă

sfADAUGĂ

CĂUTARE

- pp. că vrem să căutăm cheia c
- o căutăm în ordinea dată de secvența de verificare a cheii $\langle d(c, 0), d(c, 1), \dots, d(c, m - 1) \rangle$ până dăm de o locație liberă (marcată cu NIL)
- dacă găsim cheia undeva pe secvența de verificare \Rightarrow **căutare cu succes**, altfel **căutare fără succes**
- în **Exemplu 1**
 - o căutăm **35 (cu succes)**: căutăm în ordinea 5, 6, 7
 - o găsim pe poziția 7
 - o căutăm **45 (fără succes)**: căutăm în ordinea 5, 6, 7, 8, NIL
 - nu găsim cheia pe secvența de verificare

Funcția CAUTĂ (c, ch) este

$\{c: \text{Container}, ch: \text{TCheie}\}$

$i \leftarrow 0$ {numărul de verificare}

$gasit \leftarrow \text{fals}$ {nu am găsit cheia}

repetă

$j \leftarrow c.d(ch, i)$ {locația de verificat}

dacă $c.ch[j] = ch$ atunci {am găsit cheia}

$gasit \leftarrow \text{adev}$

altfel

$i \leftarrow i + 1$ {căutăm mai departe pe secvența de verificare}

sfdacă

până_când ($c.ch[j] = \text{NIL}$) sau ($i = c.m$) sau ($gasit$)

CAUTĂ $\leftarrow gasit$

sfCAUTĂ

STERGERE

Ștergerea unei chei c

- identificăm locația i la care este memorată cheia (ca și la căutare)
- nu putem marca locația i cu NIL (ca și cum ar fi liberă), deoarece ar fi imposibil să mai accesăm orice cheie a cărei inserare a verificat locația i și a găsit-o liberă

Sunt două soluții la ștergere

1. O soluție este de a marca la locația i o valoare specială, ȘTERS (în loc de NIL)

- vom modifica ADAUGĂ astfel încât să trateze locațiile marcate cu ȘTERS ca și cum ar fi libere
 - o linia marcată cu (*) în ADAUGĂ o vom înlocui cu
 - dacă ($c.ch[j] = \text{NIL}$) sau ($c.ch[j] = \text{ȘTERS}$) atunci
- nu este necesar să modificăm CAUTĂ

2. Prin deplasări ale datelor, astfel încât să nu mai existe riscul de a nu regăsi o cheie

Ștergere prin deplasări de date

Ilustrăm ideea ștergerii prin deplasări de date, presupunând verificarea liniară a tabeli.

- vrem să ștergem cheia c
- o localizăm (căutând pe secvența de verificare)
- fie i locația pe care se află cheia c și care trebuie ștearsă

Va trebui să ne asigurăm că ștergerea locației i (marcarea acesteia cu NIL) nu afectează regăsirea cheilor care au fost memorate pornind de la locația i (la adăugare, au găsit locația ocupată).

Pas 1. Luăm, pe rând, locațiile în ordinea secvenței de verificare $j = i+1, i+2, \dots, m-1, 0, \dots$ până la o zonă liberă

- Fie p locația unde trebuia memorată data (cheia, în cazul nostru), de la locația j
 - $p = d'(ch[j])$
 - Întrebare

Dacă am șterge cheia de la locația i (am marca locația de la i cu NIL), am putea regăsi cheia de la locația j , pornind de la p (iterând pe secvența de verificare, fără a da de o locație liberă)?

- Avem de tratat două cazuri, fiecare caz cu 3 subcazuri

- $i < j$

1a) $0 \leq p \leq i$

0 p i j $m-1$

➤ (*) se mută data (cheia, în cazul nostru) de la locația j la locația i și se continuă cu ștergerea locației j

- $ch[i] \leftarrow ch[j]$

- $i \leftarrow j$

➤ Salt la **Pas 1**.

1b) $i < p \leq j$

0 i p j $m-1$

➤ nu e necesară mutare de date (ștergerea lui i nu afectează)

➤ Salt la **Pas 1**.

1c) $j < p \leq m-1$

0 i j p $m-1$

➤ se efectuează mutarea de date (*) de la 1a)

➤ Salt la **Pas 1**.

- $j < i$

2a) $0 \leq p \leq j$

0 p j i $m-1$

- nu e necesară mutare de date (ștergerea lui i nu afectează)
- Salt la **Pas 1**.

2b) $j < p \leq i$

0 j p i $m-1$

- se efectuează mutarea de date (*) de la 1a)
- Salt la **Pas 1**.

2c) $i < p \leq m-1$

0 j i p $m-1$

- nu e necesară mutare de date (ștergerea lui i nu afectează)
- Salt la **Pas 1**.

Pas 2. La încheierea structurii repetitive de la **Pas 1**, se poate șterge cheia de locația de la i .

Exemplu Considerăm exemplul 1 - adresare deschisă cu verificare liniară

- $m=10$
- $d(c, i) = (d'(c) + i) \bmod m \quad \forall i = 0, 1, \dots, m-1$
- $d'(c) = c \bmod m$

c	5	15	13	22	20	35	30	32	2
d'(c)	5	5	3	2	0	5	0	2	2

Vrem să ștergem cheia 5.

Tabela inițială este

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	20	30	22	13	32	5	15	35	2	NIL

i	j	Locația j liberă?	p	Caz
5	6	NU	5	1a) \Rightarrow mutare date
6	7	NU	5	1a) \Rightarrow mutare date
7	8	NU	2	1a) \Rightarrow mutare date
8	9	DA \Rightarrow STOP	-	Pas 2 , se șterge cheia de la i

Tabela finală va fi

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	20	30	22	13	32	15	35	2	NIL	NIL

Iteratorul pe un container reprezentat folosind o TD cu adresare deschisă este simplu de implementat

- se iterează vectorul asociat, iterând doar pe pozițiile pe care e memorată o cheie diferită de NIL (în convenția noastră).
- $\theta(m)$

În directorul asociat cursului, găsiți implementarea parțială a containerului Colecție reprezentat folosind o TD cu adresare deschisă și verificare liniară.

Analiza dispersiei cu adresare deschisă

1. Teorema. Într-o TD cu adresare deschisă, în ipoteza *dispersiei uniforme simple* (SUH), cu factor de încărcare $\alpha = \frac{n}{m} < 1$ numărul mediu de verificări este CEL MULT

CORMEN, THOMAS H. - LEISERSON, CHARLES - RIVEST, RONALD R.: Introducere în algoritmi. Cluj-Napoca: Editura Computer Libris Agora, 2000.

- $\frac{1}{1-\alpha}$ pentru adăugare și căutare fără succes
- $\frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha}$ pentru căutare cu succes

Demonstrațiile sunt schițate și în directorul asociat cursului, subdirectorul **Demonstratii Complexitati - TD Adresare deschisă.**

CONCLUZII

- Dacă α e constant $\Rightarrow \theta(1)$ în medie pentru operații
- Caz defavorabil $O(n)$

Observație

- în cazul în care se folosește vector dinamic pentru implementarea tabeli, analiza anterioară a complexităților este valabilă pentru cazul **amortizat**
- deși, din perspectivă teoretică, adresarea deschisă e mai performantă decât cea închisă, în bibliotecile existente (Java, STL), TD sunt implementate prin liste independente
 - Java – *HashTable*
 - se poate preciza, prin constructor, capacitatea inițială a tabeli și factorul de încărcare
 - implicit $m=11$, factorul de încărcare maxim e $\alpha=0.75$
 - redispersare dacă se depășește factorul de încărcare implicit
 - STL - *unordered_map/set*
 - implicit $m=8$, factorul de încărcare maxim e $\alpha=1$
 - redispersare dacă se depășește factorul de încărcare implicit

PROBLEME

1. Considerând o tabelă de dispersie cu adresare deschisă, scrieți un algoritm pentru operația de **ștergere** și modificați operația **adaugă** și **caută** pentru a încorpora valoarea specială **ȘTERS**.
2. Se consideră o tabelă de dispersie cu adresare deschisă, cu dispersie uniformă și factor de încărcare 0.5. Dați margini superioare pentru numărul mediu de verificări într-o căutare cu succes și o căutare fără succes.

2. DISPERSIA PERFECTĂ (PERFECT HASHING)

- Scop - să nu existe coliziuni.
 - Cât de mare să fie tabela încât să fim siguri că nu sunt coliziuni?
 - Dacă $m=N^2$, atunci tabela este fără coliziuni cu probabilitatea cel puțin 0.5.
 - N – numărul de chei din container
 - Impractic.
- Soluție - *Dispersia perfectă (perfect hashing)*
 - Doar dacă avem o colecție **STATICĂ** de chei (nu se adaugă chei)
 - Se folosește o TD de dimensiune N (tabela **primară**)
 - În locul listelor independente se folosește o altă TD (tabela **secundară**)
 - Tabela secundară de la o locație i se va construi cu dimensiunea n_i^2 unde n_i este numărul de elemente din acea tabelă (numărul de coliziuni de la locația i).
 - Tabela secundară se va construi cu o altă funcție de dispersie și va fi reconstruită până nu va avea coliziuni.
 - Se poate demonstra că spațiul total de memorare pentru tabelele secundare este cel mult $2*N \Rightarrow O(N)$.
- Fie p numărul prim mai mare decât cea mai mare cheie.
- Funcțiile de dispersie se aleg dintr-o familie de *funcții de dispersie universală*.
 - $d_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m$
 - $1 \leq a \leq p - 1, 0 \leq b \leq p - 1$ (a și b selectate aleator la inițializarea funcției de dispersie)
 - $m = N$
- Performanța în caz **defavorabil** este $\theta(1)$ (se caută cel mult 2 poziții – cea din TD principală și secundară)

EXEMPLU

- 15 litere: I, N, S, X, E,....
- $N=m=15$
- Fiecărei litere îi asociem ca *hashCode* numărul de ordine al literei în alfabet
- Dacă $a=3$ și $b=2$ (alese aleator)
- $p=29$

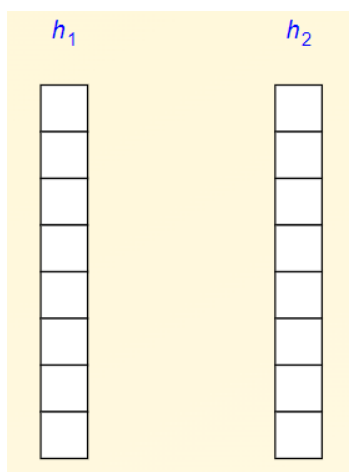
Litera	I	N	S	X	E
<i>hashCode</i>	9	14	19	24	5
<i>d(hashCode)</i>	0	0	1	1	2

- **Coliziuni**
 - poziția 0 – I, N
 - poziția 1 – S, X
 - poziția 2 – E
 - ...
- Pentru pozițiile unde nu avem coliziuni (ex. poziția 2) avem o TD secundară cu un singur element și $d(x)=0$
- Pentru pozițiile cu 2 elemente, vom avea o TD secundară cu 4 elemente și diferite funcții de dispersie, alese din același *univers*, cu diferite valori aleatoare pentru a și b .
- De ex., pentru poziția 0, putem defini $a=4$ și $b=11$ și vom avea
 - $d(I)=d(9)=2$, $d(N)=d(14)=1$
- Pentru poziția 1, să pp. că avem $a=5$ și $b=2$.
 - $d(S)=d(19)=2$, $d(X)=d(24)=2 \Rightarrow$ coliziune
 - Alegem alte valori pentru a și b – de ex. $a=2$ și $b=13$. Vom avea
 - $d(S)=d(9)=2$, $d(X)=d(14)=3$

3. ALTE VARIANTE DE DISPERSIE

1. Dispersia Cuckoo (*Cuckoo hashing*)

- Se folosesc 2 TD cu două funcții de dispersie diferite
 - Fiecare tabelă e mai mult de jumătate *goală*
 - [articol](#) 2004
 - performanța scade semnificativ dacă factorul de încărcare α depășește 0.5.
 - Ex. TD Cuckoo conținând spațiu de memorare pentru 16 chei



- Se poate garanta că un element va fi fie în prima, fie în a doua tabelă.
- **Căutarea și ștergerea** sunt simple (elementul se va localiza în una din cele 2 TD)
- **Inserarea unei chei c**
 - Se încearcă adăugarea în prima tabelă. Dacă e liber, se adaugă.
 - Dacă poziția în prima tabelă e ocupată de cheia c' , se scoate c' din prima tabelă, se adaugă noul element c . Elementul scos din prima tabelă c' se va adăuga în a doua. Dacă poziția în a doua tabelă e ocupată de c'' , se va scoate acel element (în locul său se va adăuga elementul c' din prima tabelă) și c'' se va adăuga în prima tabelă. Se va repeta procesul până se va obține o poziție liberă. Dacă se revine în aceeași poziție de start (există un ciclu) se face re-dispersare (**rehashing**)
 - **rehashing**
 - complexitate timp caz defavorabil $O(n)$
 - complexitate timp amotizată $O(1)$
- Generalizare - **D-Cuckoo Hashing**
 - utilizarea unui număr fixat (ex. 4) de TD interne

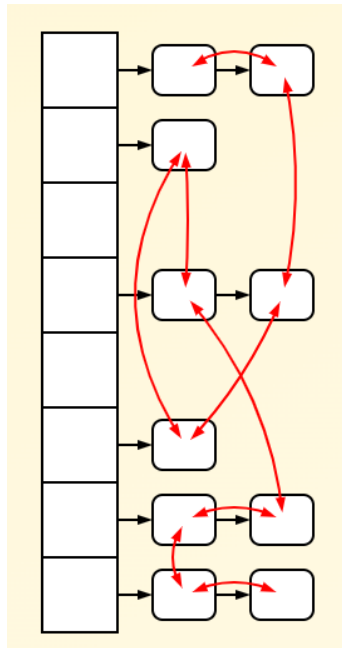
2. Dispersia Robin Hood (*Robin Hood hashing*)

- 1986 – [teză doctorat](#)
 - 2018 - [Concurrent RH hashing](#)
- Bazată pe *adresare deschisă*
- La inserarea unor noi chei, vechile chei sunt deplasate a.i. toate cheile sunt păstrate „suficient” de aproape de locația la care se dispersează.
 - deviația standard a distanțelor cheilor față de locația lor de dispersie este minimizată
- folosește noțiunea de **lungime a secvenței de verificare** a cheilor (*probe sequence length* - PSL)
 - numărul de locații necesar pentru a găsi cheia pe secvența de verificare
- **proprietăți**
 - operația de **căutare**
 - complexitate timp $O(1)$ amortizat, $O(\log_2 n)$ defavorabil
 - căutare fără succes rapidă
 - eficiență dpdv spațiului de memorare
 - nu se memorează liste înălțuite sau legături adiționale
 - funcționează bine pentru α mai c decât ≈ 0.9 .

3. Liste independente interconectate (*Linked hashing*)

- JAVA – *LinkedHashMap*
 - *HashMap* din Java folosește rezolvare coliziuni prin liste independente (inițial $m=16$)
 - Dacă $\alpha > 0.75$, se face redispersare (*rehashing*) – m se dublează
 - Java 8 – în locul listelor înălțuite se folosesc arbori binari de căutare echilibrați $\Rightarrow \theta(\log_2 n)$ caz defavorabil pentru căutare
- Combină ideea de TD și listă înălțuită.

- Păstrează o listă dublu înlănțuită cu toate elementele din TD într-o anumită ordine (implicit -în ordinea în care au fost adăugate în dicționar)
- Fiecare intrare în tabelă (*Entry*) – nod care memorează perechea $\langle c, v \rangle$ - are 2 pointeri adiționali spre perechea *anterioară* și cea *următoare* (adresele sunt ale nodurilor din lista dublu înlănțuită)
- Reprezentare dublu înlănțuită => *adăugare* și *ștergere* în $O(1)$



- Datorită mecanismului de memorare, cheile vor fi returnate (la iterare) în ordinea în care au fost adăugate (varianta implicită – *insertion order* – se poate modifica la *access order*: de la cea mai recent accesată la cea mai devreme accesată).
- Aceeași performanță ca și *HashMap*
- Ca și implementarea *HashMap*, *LinkedHashMap* nu e sincronizată (nu funcționează cu acces concurent)
- **Dezavantaj** – spațiu de memorare suplimentar pentru lista înlănțuită
- **Avantaj** - iterare în $\theta(n)$ (față de $\theta(n + m)$)

Alte variante

- **D-left Hashing**
 - 2006, [articol](#)
- **Hopscotch Hashing**
 - 2008, [articol](#)
-