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.
  - 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 $2 \cdot N \Rightarrow O(N)$.
- Fie *p* numărul prim mai mare decât cea mai mare cheie.
- Funcțiile de dispersie se aleg dintr-o familie de *funcții de dispersie universală*.
  - $d_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m$
  - $1 \le a \le p-1, 0 \le b \le p-1$ (*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
hashCode	9	14	19	24	5
d(*hashCode*)	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
  - $d(\text{I})=d(9)=2$, $d(\text{N})=d(14)=1$
- Pentru poziția 1, să pp. că avem $a=5$ și $b=2$.
  - $d(\text{S})=d(19)=2$, $d(\text{X})=d(24)=2$ => coliziune
  - Alegem alte valori pentru *a* și *b* – de ex. $a=2$ și $b=13$. Vom avea
    - $d(\text{S})=d(9)=2$, $d(\text{X})=d(14)=3$

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*
- 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**)

2. Liste independente interconectate (*Linked hashing*)

- JAVA – *LinkedHashMap*
  - *HashMap* din Java folosește rezolvare coliziuni prin liste independente (inițial m=16)
  - Dacă α>0.75, se face redispersare (*rehashing*) – m se dublează
  - Java 8 – în locul listelor înlănțuite se folosesc arbori binari de căutare echilibrați $\Rightarrow \theta(\log_2 n)$ caz defavorabil pentru căutare
- Combină ideea de TD și listă înlănțuită.
  - 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 <c,v> - are 2 pointeri adiționali spre perechea *anterioară* și cea *următoare* (adresele sunt ale nodurilor din lista dublu înlănțuită)
- 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 perfomanță 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 $\theta(n+m)$)