TABELA DE DISPERSIE
- continuare –

B. Rezolvare coliziuni prin *liste întrepătrunse (întrepătrunderea listelor)* – COALESCED CHAINING

- Toate listele (care memorează coliziuni) se memorează în tabelă, nu sunt liste în afara tabelei (vezi lista înlănțuită cu înlănțuiri reprezentate pe tablou)
- Este o combinație între **separate chaining** și **open adressing** (*closed hashing*)
    - considerată o tehnică de **adresare deschisă** (*open adressing*)
    - nu sunt elemente memorate în afara tabelei.
    - se consideră mai bună decât rezolvarea coliziunilor prin liste independente (elementele se memorează în tabelă, nu se memorează liste separate)
- Nu se folosesc pointeri pentru memorarea înlănțuirilor
- Factorul de încărcare este subunitar $\alpha < 1$, altfel tabela este plină
- Gestiunea spațiului liber – secvențial (stânga-dreapta, dreapta-stânga)
    - listă înlănțuită a spațiului liber nu aduce un avantaj dpdv al complexității timp
- Dezavantaj: tabela se poate umple ($\alpha = 1$). Soluție: se crește *m*, ceea ce presupune redispersarea elementelor.
- Experimental: funcția de dispersie se consideră bună dacă spațiul de memorie e ocupat mai puțin de 75% ($\alpha < 0.75$)
-

Teoremă. Într-o TD în care coliziunile sunt rezolvate prin liste întrepătrunse, în *ipoteza dispersiei uniforme simple* (SUH), o **TOATE** operațiile (**adăugare, căutare, ștergere**), necesită, în ***medie***, un timp $\theta(1)$.

Donald E. Knuth, The Art of Computer Programming, Second edition, University of Stanford, 1998

- Timpul mediu pentru **căutare fără succes** $T(\alpha) \approx 1 + \frac{1}{4}(e^{2\alpha} - 1 - 2\alpha)$
- Timpul mediu pentru **căutare cu succes** $T(\alpha) \approx 1 + \frac{1}{8\alpha}(e^{2\alpha} - 1 - 2\alpha) + \frac{\alpha}{4}$

Demonstrațiile sunt schițate în directorul asociat cursului, subdirectorul **Demonstratii Complexitati - TD Liste Intrepatrunse.**

Observație

- în cazul în care se folosește vector dinamic pentru implementarea tabelei, analiza anterioară a complexităților este valabilă pentru cazul **amortizat.**

Reprezentare

Ca și la lista simplu înlănțuită în care înlănțuirile sunt memorate pe tablou, table va memora doi vectori

- un vector care memorează elementele
- un vector care memorează legăturile între elemente (sub forma unor indici).

Presupuneri:

- Se memorează doar cheile.

- Chei distincte.
- Dacă o locație nu are legătură spre o altă locație din tabelă, se memorează **-1** în câmpul de legătură (*urm*).
- Chei naturale (se memorează **-1** dacă locația e liberă)
- Spațiul liber e gestionat secvențial (de la stânga la dreapta) – sau de la dreapta la stânga: *primLiber* indică prima poziție liberă din tabelă

  Container

  m: Intreg //capacitatea tabelei
  ch: TCheie[] //cheile
  urm: Intreg[] //legaturile
  primLiber:Intreg //prima locatie libera

  Funcția de dispersie este d:TCheie$\rightarrow\{0,1\ldots,m\text{-}1\}$

EXEMPLU

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

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

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 (-1, în convenția noastră) ⇒ caz favorabil, memorăm cheia
  - dacă *i* e chiar *primLiber*, atunci se actualizează primul liber
- Locația *i* nu este liberă ⇒ avem coliziune
  - dacă *primLiber* = -1 (tabela este plină) ⇒ redimensionare: mărim *m*, ceea ce presupune redispersarea elementelor (*rehashing*)
  - memorăm cheia *c* la *primLiber*
  - ultimul nod din lista înlănțuită care începe de la locația *i* este legat de *primLiber*
  - se actualizează primul liber

Pas 1. Inițializare

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

- *primLiber* = 0

Pas 2. Adăugăm cheia 5

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

- *primLiber* = 0

Pas 3. Adăugăm cheia 15

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

- *primLiber* = 1

Pas 3. Adăugăm cheia 13

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	15	-1	-1	13	-1	5	-1	-1	-1	-1
Următor	-1	-1	-1	-1	-1	0	-1	-1	-1	-1

- *primLiber* = 1

Pas 4. Adăugăm cheia 22

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	15	-1	22	13	-1	5	-1	-1	-1	-1
Următor	-1	-1	-1	-1	-1	0	-1	-1	-1	-1

- *primLiber* = 1

Pas 5. Adăugăm cheia 20

Indice	0	1	2	3	4	5	6	7	8	9
Cheie	15	20	22	13	-1	5	-1	-1	-1	-1
Următor	1	-1	-1	-1	-1	0	-1	-1	-1	-1

- *primLiber* = 4

.....

La final, tabela va fi

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

subalgoritm ***actPrimLiber***(*c*) **este**
//se actualizează *primLiber* după ce locația a fost ocupată
//operația nu este în interfața containerului
    c.primLiber←*c.primLiber*+1
    câttimp (*c.primLiber*≤ *c.m*-1) și (*c.ch*[*c.primLiber*] ≠-1) **execută**
        c.primLiber←-*c.primLiber*+1
    sfcâttimp
sf*actPrimLiber*

- algoritmul de adăugare la sfârșitul listei înlănțuite (în caz de coliziune) – **LISCH** (***Late Insertion Standard Coalesced Hashing***)

subalgoritm ***adaugă***(*c*, *ch*) **este**
//pre: *c* e containerul, *ch* cheia care se adaugă
    i←*c.d*(*ch*)
    dacă *c.ch*[i]=-1 **atunci** //locația e liberă, memorăm

```
            c.ch[i] ←ch
            dacă i=c.primLiber atunci
                  actPrimLiber(c)
            sfdacă
      altfel
            //adăugăm la finalul listei înlănțuite care este memorată de la locația i
            // dacă mai găsim cheia, ne oprim
            câttimp (i≠-1) și (c.ch[i] ≠ch) execută
                  j←i
                  i←c.urm[i]
            sfcâttimp
            dacă i≠-1 atunci //am mai găsit cheia
                  @ cheie existentă
             altfel
                  dacă c.primLiber ≤ c.m-1 atunci  //tabela nu este plină
                        c.ch[c.primLiber] ←ch
                        c.urm[j] ←c.primLiber
                        actPrimLiber(c)
                   altfel
                        @ depășire tabelă
                  sfdacă
            sfdacă
     sfdacă
sfadaugă
```

CĂUTARE

- pp. că vrem să căutăm cheia *ch*
- o căutăm în lista înlănțuită care pornește de la locația de dispersie a cheii *ch*, adică *d*(*ch*)
- dacă găsim cheia în lista înlănțuită ⇒ **căutare cu succes**, altfel **căutare fără succes**
- exemplu
    - căutăm **35 (cu succes)**: 5→15→20→**35**
    - căutăm **45 (fără succes)**: 5→15→20→35→ 30→ **-1**

ȘTERGERE

- pp. că vrem să ștergem cheia *ch*
- localizăm cheia
- exemplu
    - *ch*=5, identificăm locația unde află cheia, *i*=5

      (5,**5**) →(0,**15**) →(1,**20**) →(4,**35**) →(6,**30**)

Observație. Într-o înlănțuire (de ex,. înlănțuirea care pornește de la locația 5, nu avem doar chei care se află în aceeași coliziune). De ex. cheile 5, 15, 35 și 20, 30 se află în două coliziuni diferite.

Dacă am memora -1 pe poziția *i* (la care se află cheia *ch*), nu s-ar mai regăsi cheile care au fost memorate pornind de la acea locație.

Cum vom face ștergerea? Prin deplasări ale cheilor, în lista înlănțuită care pornește de la poziția *i*.

- **Pas 1** pe înlănțuirea care pornește de la *i*, caut prima locație *j* care memorează o cheie *c* care ar fi trebuit memorată la *i* ($d(c)=i$)
- dacă găsesc locația *j*, atunci
  - mut cheia *c* la locația *i*
  - continui cu ștergerea locației *j* ($i \leftarrow j$, salt la Pas 1)
- dacă nu găsesc locația *j*, înseamnă că pot șterge nodul de la locația *i*, deoarece ștergerea acestuia nu mai afectează regăsirea altor elemente (ștergerea acestui nod se reduce la ștergerea unui nod din lista simplu înlănțuită)
  - memorez -1 pe poziția cheii și legăturii de la *i*
  - legătura precedentului lui *i* din înlănțuire o setez pe legătura lui *i*

$\Rightarrow$ (5,**15**) →(0,**20**) →~~(1,20)~~→(4,**35**) →(6,**30**)

- tabela rezultată în urma ștergerii

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

Iteratorul pe un container reprezentat folosind o TD cu liste întrepătrunse este simplu de implementat

- se iterează vectorul asociat, iterând doar pe pozițiile pe care e memorată o cheie diferită de -1 (în convenția noastră).
- $\theta(m)$

În directorul asociat cursului, găsiți implementarea parțială a containerului Colecție reprezentat folosind o TD cu liste întrepătrunse.

Variante pentru îmbunătățirea performanței (reducerea numărului de locații verificate în caz de coliziune)

- organizarea tabelei - zonă separată (***primary area***) pentru elementele care nu sunt în coliziune și zonă separată pentru elementele care sunt în coliziuni (***overflow area***)
  - $Address\ Factor = |primary\ area| / dimensiunea\ tabelei$
  - o valoare de aprox. 0.86 pentru *Address Factor* conduce la o performanță aproape optimală pentru ***căutare*** pentru majoritatea valorilor lui α.
- modalitatea de înlănțuire a elementelor dintr-o coliziune
  - dublu înlănțuit
- modalitatea de selectare a spațiului liber (*primLiber*)
  - de la dreapta spre stânga (de la finalul tabelei)

- alegerea aleatoare a spațiului liber (doar 1% îmbunătățire) – alg. de adăugare **REISCH** (R - *random*)
- alternarea selecției spațiului liber între începutul și finalul tabelei - **BLISCH** (B – *bidirectional*)
- experimental: pentru $\alpha > 0.2$ **LISCH** e mai performant

IMPLEMENTAREA OPERAȚIILOR DE <u>CĂUTARE</u> ŞI <u>ŞTERGERE</u> LA SEMINARUL 6