SDA - Seminar 6

- Termen pentru “Stadiul proiectului” – azi până la ora 12 noapte

1. **Iterator pe Dicționar Ordonat reprezentat sub formă de tabelă de dispersie (rezolvare coliziuni prin liste independen, te).**

- Presupunem:
  - Memorăm doar cheile
  - Avem chei întregi

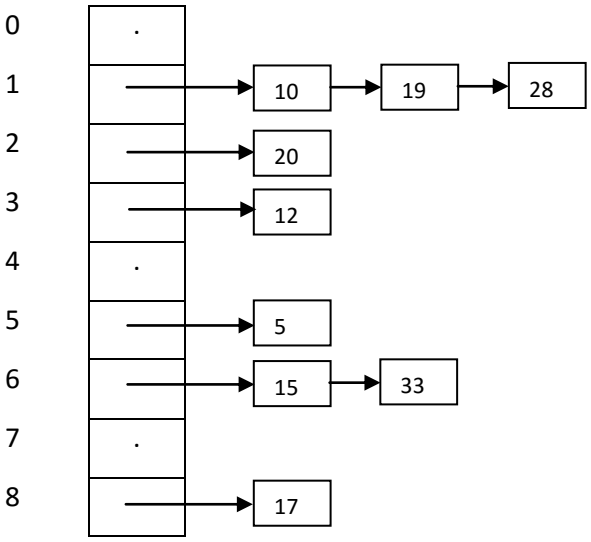
De ex:

- cheile din dicționar: 5, 28, 19, 15, 20, 33, 12, 17, 10 – cheile sunt unice!
- TD
  - m = 9
  - Dispersie prin diviziune:
    - hashCode(c) = hc
    - d(c) = hc mod m

c	5	28	19	15	20	33	12	17	10
d(c)	5	1	1	6	2	6	3	8	1

- d(c) poate să aibă duplicate – se numesc coliziuni

TD:



Iterator:

- Dacă parcurg cu iterator, elementele vor fi afișate: 5, 10, 12, 15, 17, 19, 20, 28, 33
- Dacă iterez folosind iteratorul -> complexitatea să rămână Θ(n)

Reprezentare:

NodT:	DicționarOrdonat:	IteratorDicționar:
e: TElement	m: Intreg	d: DicționarOrdonat
urm: ↑NodT	l : (↑NodT)[]	l: TListă
	d: TFuncție	curentNod: ↑NodT
	R: relație	

```
subalgoritm creeaza(it, d):
      it.d ← d
      interclaseazaListe (d, it.l)
      it.curentNod ← it.l.prim
sf_subalgoritm
```

- interclaseazaListe interclasează listele:
  - prima listă cu a 2-a, după care rezultatul cu a 3-a, etc.
  - toate listele deodată folosind un ansamblu
- Operațiile valid, următor, element au complexitate Θ(1)

Complexitatea interclasării:

$$\left.\begin{matrix}\text{TD cu m poziții} \\ Dicționar\ cu\ n\ elemente\end{matrix}\right\} \Rightarrow nr\ mediu\ de\ elemente\ într-o\ listă: \frac{n}{m} = \alpha\ (factor\ de\ încărcare)$$

Interclasare prima lista cu a 2-a, etc.:

- lista1 + lista2 => lista12 => α + α = 2α
- lista12 + lista3 => lista123 => 2α + α = 3α
- lista123 + lista4 => lista1234 => 3 α + α = 4 α
- …

Total interclasare: $2\alpha + 3\alpha + \cdots + m\alpha \approx \left.\begin{matrix}\frac{m*(m+1)}{2}\alpha \\ \alpha = \frac{n}{m}\end{matrix}\right\} \rightarrow \frac{m\,(m+1)}{2}\ \frac{n}{m} \Rightarrow \in\ \theta\ (n*m) \approx\ \theta(n)$

(m – constant)

Toate listele deodată, folosind un ansamblu:

- Punem primul nod din fiecare listă într-un ansamblu
- Scoatem nodul minim și adăugăm în ansamblu următorul lui nod (dacă există)
- Ansamblul va conține maxim *k* elemente în orice moment (*k* este numărul de liste, 1 ≤ *k* ≤ m) => înălțimea ansamblului e O($\log_2$ k)

- Complexitatea interclasării:
  - O(n $\log_2$k), dacă k > 1
  - Θ(n), dacă k = 1
- k este mai mic sau egal cu m => $\log_2$ k e constant

2. Dicționar – reprezentare cu o tabelă de dispersie – rezolvare coliziuni prin liste întrepătrunse

- Presupunem:
  - Memorăm doar chei
  - Avem chei întregi

De ex:

- 5, 18, 16, 15, 13, 31, 26
- TD:
  - m = 13
  - dispersie prin diviziune

c	5	18	16	15	13	31	26
d(c)	5	5	3	2	0	5	0

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	18	13	15	16	31	5	26						
urm	~~-1~~ 1	~~-1~~ 4	-1	-1	~~-1~~ 6	~~-1~~ 0	-1	-1	-1	-1	-1	-1	-1

primLiber = ~~0~~ ~~1~~ ~~4~~ ~~6~~ 7

- PrimLiber se ia de la stânga la dreapta, nu mai este înlănțuit
- Într-o înlănțuire pot avea elemente care aparțin unor coliziuni diferite. De ex. coliziunea care începe pe poz 5: 5 (5) – 18 (5) – 13 (0) – 31 (5) – 26 (0)

Reprezentare:

TElement:
c: TCheie
v: TValoare

Dicționar:
m: Întreg
e: (TElement)[]
urm:(0,…, m-1)[]
primLiber: Întreg (0,...,m-1)
d: TFuncție

```
subalgoritm creează (d):
      @ se inițiază funcție de dispersie d
      @ se inițiază m
      pentru i ← 0, m-1 execută
            d.e[i] ← -1
            d.urm[i] ← -1
      sf_pentru
      d.primLiber ← 0
sf_subalgoritm
Complexitate: Θ(m)
```

```
Funcția caută(d, c):
      i ← d.d(c)
      câttimp (i ≠ -1 și d.e[i] ≠ c) execută
            i ← d.urm[i]
      sf_câttimp
      dacă i = -1 atunci
            caută ← -1
      altfel
            caută ← i
sf_funcție
Complexitate: O(m) dar în medie Θ(1)

Subalgoritm adăugare – s-a făcut la curs!!!
```

Ștergerea: șterg cheia 5

- **Problema:** risc să pierd legătura spre o anumită cheie
- Nu pot trata ca ștergere dintr-o listă înlănțuită pentru că anumite elemente nu pot ajunge înaintea poziției unde s-ar dispersa. De exemplu, nu pot muta 26 în locul lui 5 (pentru că 26 se dispersează pe poziția 0, iar înlănțuirea care pornește de la poziția 0 nu trece prin poziția 5).

	0	1	2	3	4	5	6	7	8	9	10	11	12
c	~~18~~ 13	13	15	16	31	~~5~~ 18	26						
urm	~~1~~ 4	~~4~~ -1	-1	-1	6	0	-1	-1	-1	-1	-1	-1	-1

primLiber: 7

Pași:

1. Nu pot pune e[5] = -1 și urm[5] = -1 pentru că pierd legătura spre 18 (și la o căutare nu voi mai găsi elementele 18 și 31).
2. Caut elemente (pe înlănțuire) care se dispersează în poziția de unde șterg (poziția 5).
    a. Dacă nu există astfel de elemente, șterg elementul ca și cum aș șterge un nod dintr-o listă simplu înlănțuită
    b. Dacă există, atunci mut elementul pe poziția de unde șterg, și repet procesul de ștergere pentru poziția de unde am mutat.

- șterg cheia 5, care e pe poziția 5
- caut primul element care se dispersează pe poziția 5 => 18
- mut 18 pe poziția 5
- acum vreau să șterg cheia 18, care e pe poziția 0
- caut primul element care se dispersează pe poziția 0 => 13
- mut 13 pe poziția 0
- acum vreau să șterg cheia 13, care e pe poziția 1
- caut primul element care se dispersează pe poziția 1 => nu este
- șterg cheia 13, modificând legăturile

```
subalgoritm sterge(d, c) este
    i ← d.d(c)
    j ← -1 {precedentul lui i, când șterg îmi trebuie nodul de dinainte}
    {parcurgem tabela să vedem dacă poz i are vre-un anterior}
    k ← 0
    câttimp (k < d.m și j = -1) execută
        dacă d.urm[k] = i atunci
            j ← k
        sf_dacă
    sf_câttimp
    {localizez cheia care trebuie stearsa. Setez si precedentul}
    câttimp i ≠ -1 și d.c[i] ≠ c execută
        j ← i
        i ← d.urm[i]
    sf_câttimp
    dacă i= -1 atunci
        @cheia nu există
    altfel
        {caut altă cheie care se dispersează în i}
        gata ← fals {devine adevărat când nu se dispersează nimic în i}
        repetă
            p ← d.urm[i] {prima pozitie verificata}
            pp ← i {anteriorul pozitiei p}
            câttimp p ≠ -1 și d.d(d.e[p]) ≠ i execută
                pp ← p
                p ← d.urm[p]
            sf_câttimp
            dacă p = -1 atunci
                gata ← adevarat
            altfel
                d.e[i] ← d.e[p]
                j ← pp
                I ← p
            sf_dacă
        până_când gata
        {șterg cheia de pe poziția i}
        dacă j ≠ -1 atunci
            d.urm[j] ← d.urm[i]
        sf_dacă
        d.e[i] ← -1
        d.urm[i] ← -1
        dacă d.primLiber > i atunci
            d.primLiber ← i
        sf_dacă
    sf_dacă
sf_subalgoritm
```