SDA – Seminar 4

- **Cuprins**:
  - Algoritmi de sortare
    - BucketSort
    - Sortarea Lexicografică
    - Radix Sort
  - Interclasarea a două liste simplu înlănțuite, alocate dinamic

Algoritmi de sortare

Să ne amintim, din seminarul 2, complexitățile alogoritmilor de sortare cunoscuți.

Algoritm	Complexitate timp				Complexitate spațiu
	CF	CD	CM	Total	
Căutare secvențială	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$O(n)$	$\Theta(1)$
Căutare binară	$\Theta(1)$	$\Theta(\log_2 n)$	$\Theta(\log_2 n)$	$O(\log_2 n)$	$\Theta(1)$
Sortare prin selecție	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(1)$ – *in place**
Sortare prin inserție	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$ – *in place*
Sortare prin metoda bulelor	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	$O(n^2)$	$\Theta(1)$ – *in place*
Sortare rapidă (*QuickSort*)	$\Theta(n \log_2 n)$	$\Theta(n^2)$	$\Theta(n \log_2 n)$	$O(n^2)$	$\Theta(1)$ – *in place*
Sortare prin interclasare (*Merge Sort*)	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n \log_2 n)$	$\Theta(n)$ - *out of place*

Toți algoritmii de sortare din tabelul anterior sunt **algoritmi de sortare prin comparare**. De ce? Deoarece sortarea unui șir se efectuează pe bază de comparații între elementele șirului. Se poate arăta că orice algoritm de sortare prin comparare necesită Ω(n log2n) comparații, în cel mai defavorabil caz.

În seminarul curent, se vor prezenta algoritmi de sortare având **complexitate liniară**, dar care sunt aplicabili doar dacă datele de intrare verifică anumite constrângeri. Așadar, aceștia **nu** vor fi algoritmi de sortare prin comparare.

A. BucketSort

Enunțul problemei

- Se dă un șir S de *n* perechi (cheie, valoare), $cheie_i \in \{0, 1, \ldots, N-1\}, i=1,n$
- Se cere să se sorteze S după chei

Exemplu

Se dă un șir S: (7, d) (1, c) (3, b) (7, g) (3, a) (7, e)

n = 6

N = 9 (sau 8, sau 10 etc.)

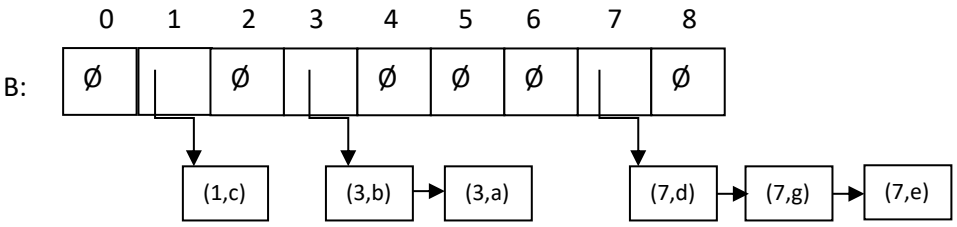
=> (1, c) (3, b) (3, a) (7, d) (7, g) (7, e)

Observăm că se cere ca sortarea să se efectueze strict după chei. Prin urmare, perechile având aceeași cheie fi consecutive în șirul sortat, însă ordinea lor este irelevantă. Dacă există mai multe perechi cu o aceeași cheie, vor exista mai multe soluții. Alegerea uneia dintre soluții se poate face după criteriul păstrării ordinii inițiale a perechilor având aceeași cheie.

Ideea algoritmului:

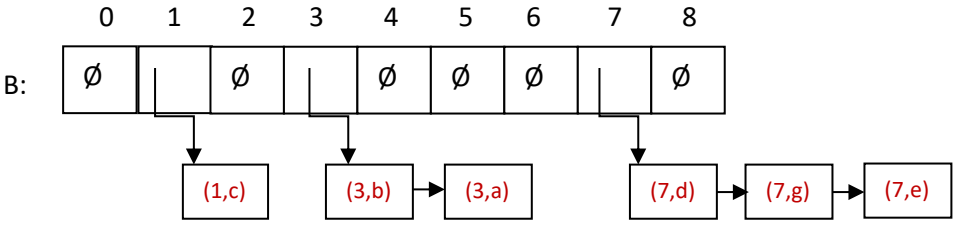
- Se folosește un șir (tablou unidimensional = vector) de șiruri (sau liste, numite în contextul algoritmului *bucket*-uri) auxiliar, *B*, de dimensiune *N => B[0…N-1]*
- Într-un prim pas, fiecare pereche din S se mută în B în lista de pe indexul corespunzător (egal cu cheia), adică în lista (B[c]), adăugându-se la finalul acesteia.

B la finalul primului pas, pentru exemplul dat:



- În cel de-al doilea pas, se parcurge B (de la 0 la N-1) și se mută, în ordine, perechile din fiecare subșir (adică element șir) al lui B la finalul secvenței S (care a fost golită la pasul anterior).

Observăm că de-al doilea pas conduce la obținerea șirului ordonat: (1, c) (3, b) (3, a) (7, d) (7, g) (7, e) .



Pentru a descrie în Pseudocod algoritmul, presupunem următoarele operații pe elementele de tip listă ale tabloului B și pe secvența S.

- vidă (Listă)

- prim (Listă): TPoziție
- șterge (Listă, TPoziție)
- adaugăSfârșit(Listă, (c,v))

Cu alte cuvinte, considerăm că elementele tabloului B și secvența S sunt de tip generic, de tipul (TAD) Listă.

Descrierea Pseudocod a algoritmului BucketSort, conform ideii descrise mai sus:

```
Algoritm BucketSort(S, N) este:
//presupunem că șirul B există
    //Primul pas al algoritmului (conform ideii algoritmului):
    Câttimp ¬ vida (S) execută: //cât timp mai există perechi în secvența / lista S
         p ← prim (S)            //obținem poziția primeia dintre ele
        (c, v) ← șterge(S, p)   //o ștergem din secvență
         adaugaSfarsit(B[c], (c,v)) //și o adăugăm la sfârșitul listei B[c], adică a
listei de pe indexul egal cu cheia
    sf_câttimp
   //Al doilea pas al algoritmului (conform ideii algoritmului):
    Pentru i ← 0, N-1, execută: //pentru fiecare listă din B, pe rând,
            Câttimp ¬ vida (B[i]) execută: //cât timp mai există perechi în lista
curentă
                  p ← prim (B[i])   //obținem poziția primeia dintre ele
                  (c, v) ← șterge (B[i], p) //o ștergem din listă
                  adaugaSfarsit (S, (c,v))   /și o adăugăm la sfârșitul secvenței /
listei S
            sf_câttimp
    sf_pentru
sf_algoritm
```

Complexitate: Θ(N + n)

Justificare:

- Prima structură repetitivă, *Câttimp,* efectuează *n* pași, *n* fiind lungimea / numărul de elemente ale listei *S*

- Operațiile *vidă, prim, sterge și adaugă* se pot efectua în timp constant. De exemplu, dacă S și listele din B sunt reprezentate ca LDI alocate dinamic (sau ca LSI alocate dinamic, dar reținându-se și pointerul la ultimul nod), atunci atât ștergerea de la început, cât și adăugarea la sfârșit se vor efectua în timp constant.

=> Complexitatea primului pas la algoritmului este Θ(n)

- A doua structură repetitivă, *Pentru,* efectuează N pași, N-1 fiind limita superioară pentru valorile întregi ale cheilor, iar N, lungimea tabloului B

- Structura repetitivă interioară , *Câttimp,* efectuează la fiecare pas, pentru fiecare *i,* un număr de pași egal cu lungimea listei *B[i] =>* În total , cele 3 operații din *Câttimp* se efectuează de un număr de ori egal cu suma lungimilor listelor din B, adică B[0], B[1], …, B[N-1]. Dar, având în vedere faptul că în aceste liste, inițial vide, au fost distribuite cele *n* perechi din S, în primul pas al algoritmului, suma lungimilor lor este *n.* Revenind, cele 3 operații din *Câttimp,* care se pot efectua în timp constant, se afectuează, în total de *n* ori

=> Complexitatea celui de-al doilea pas al algoritmului este $\Theta(N+n) <=> \Theta(max\{N,n\})$ *(a se observa similaritatea cu problema 4 din seminarul 2)*

=> Complexitatea algoritmului BucketSort este $\Theta(N+n) <=> \Theta(max\{N,n\})$

Dacă $N \in O(n)$ => Complexitatea algoritmului BucketSort este $\Theta(n)$, deci **liniară**.

Observații:

- Pentru a aplica întocmai algoritmul prezentat, este necesar ca cheile să fie numere naturale cel mult egale cu N-1, acesta pentru a avea o corespondență directă între chei și indecși în șirul B (adică a adăuga, în primul pas al algoritmului, o pereche (c,v) în lista B[c])

Dar dacă am avea chei din mulțimea {a, a+1, …, b}, a și b fiind numere naturale? Cum am putea adapta algoritmul (astfel încât să gestionăm eficient spațiul de memorare)? Care este numărul minim de indecși de care avem nevoie în B? În lista de pe care dintre indecșii din B am adăuga o pereche (c,v) ?

- ✓ Numărul minim de indecși de care avem nevoie în B este egal cu numărul de valori posibile pentru c, adică b-a+1. Astfel, vom avea în B listele B[0], B[1], …, B[b-a].
- ✓ În primul pas al algoritmului, o pereche (c,v) se va adăuga în lista B[**c-a**]. Observăm că o pereche având cheia a va fi adăugată în lista B[0], iar o pereche având cheia b va fi adăugată în lista B[b-a].

Dar dacă am avea chei din mulțimea {-a, -a+1, …, a-1, a}, a fiind un numer natural? Care este numărul minim de indecși de care avem nevoie în B? În lista de pe care dintre indecșii din B am adăuga o pereche (c,v) ?

- ✓ Numărul minim de indecși de care avem nevoie în B este egal cu numărul de valori posibile pentru c, adică 2*a+1. Astfel, vom avea în B listele B[0], B[1], …, B[2*a].
- ✓ În primul pas al algoritmului, o pereche (c,v) se va adăuga în lista B[**c+a**]. Observăm că o pereche având cheia -a va fi adăugată în lista B[0], iar o pereche având cheia a va fi adăugată în lista B[2*a].

Dar dacă am avea chei din mulțimea {‘A’, ‘B’, … , ‘Z’}? Care este numărul minim de indecși de care avem nevoie în B? În lista de pe care dintre indecșii din B am adăuga o pereche (c,v) ?

- ✓ Numărul minim de indecși de care avem nevoie în B este egal cu numărul de valori posibile pentru c, adică cu numărul de litere din alfabel. Considerăm alfabetul latin modern, care conține 26 de litere. Astfel, vom avea în B listele B[0], B[1], …, B[25].
- ✓ În primul pas al algoritmului, o pereche (c,v) se va adăuga în lista B[**ASCII(c)-ASCII(‘A’)**]. Observăm că o pereche având cheia ‘A’ va fi adăugată în lista B[0], iar o pereche având cheia ‘Z’ va fi adăugată în lista B[25].

- O a doua observație este faptul că algoritmul păstrează ordinea inițială a perechilor având aceeași cheie, ceea ce înseamnă că, în versiunea prezentată, BucketSort este un algoritm de sortare **stabil**.

În general, un algoritm de sortare se numește **stabil** dacă păstrează ordinea inițială a elementelor egale, în raport cu relația de ordine după care se efectuează sortarea.

Cum am putea adapta algoritumul prezentat pentru a-l putea aplica pentru a ordona un șir de numere reale din intervalul [0, 1)? Indicație: se alege prima cifră de după virgulă pe post de cheie (fictivă). Care este numărul de indecși de care avem nevoie în B? În lista de pe care dintre indecșii din B am adăuga o pereche (c,v)?

✓ Vom avea 10 liste (sau *bucket*-uri) în B: B[0], B[1], …, B[9], una pentru fiecare valoare posibilă pentru prima cifră de după virgulă.

✓ O valoare v se va ***insera,*** în primul pas al algoritmului, în lista ***ordonată*** B[[v*10]] (indexul este egal cu partea întreagă a produsului v*10). Adăugarea la final poate fi substituită de inserare într-o listă ordonată, astfel încât să obținem liste (*bucket*-uri) ordonate. O alternativă ar fi să adăugam în continuare la sfârșit, dar să sortăm *bucket*-urile ulterior primului pas al algoritmului.

În această variantă, algoritmul BucketSort **nu** este unul stabil (conform definiției de mai sus).

Observații:

- Dacă datele de intrare urmează o distribuție uniformă în intervalul [0, 1), se obține un timp de execuție liniar.

- Se poate generaliza, alegând ca B să aibă dimensiune N (care poate fi chiar n, adică numărul de elemente din lista de ordonat). În acest caz, BucketSort divizează intervalul [0,1) în N (eventual n) subintervale egale și apoi distribuie cele n elemente în aceste subintervale.

În acest caz, dacă B are dimensiune N, deci indici între 0 și N-1, în lista de pe care dintre indici va fi distribuită o valoare v din [0,1)?

✓ v -> B[[v*N]] (indexul este egal cu partea întreagă a produsului v*N)

Puteți vizualiza aplicarea algoritmului pe un exemplu aici:
https://www.youtube.com/watch?v=VuXbEb5ywrU
În acest exemplu, elementele se adaugă în bucket-uri la sfârșit, urmând ca la finalul primului pas al algoritmului fiecare *bucket* să fie ordonat individual., folosind sortarea prin inserție. După cum am mai precizat, o alternativă ar fi fost să se insereze elementele în bucket-uri astfel încât acestea să rămână ordonate.

B. Sortare Lexicografică

Enunțul problemei

Se dă o secvență S de *n d*-tupluri, adică tupluri cu *d* componente / de dimensiune *d*, de forma (x_1, x_2, ..., x_d).
Se cere să se sorteze S în ordine lexicografică.

Observație: (x_1, x_2, ... ,x_d) < (lexicografic) (y_1, y_2, ..., y_d) $\Leftrightarrow$ $x_1 < y_1$ $\vee$ ($x_1=y_1 \wedge ((x_2,..., x_d) < (y_2, ..., y_d))$)
- Se face compararea după prima dimensiune, după aceea după a 2-a, și așa mai departe.

Exemplu

Se dă secvența S: (7, 4, 6), (5, 1, 5), (2, 4, 6), (2, 1, 4), (3, 2, 4)

n = 5

d = 3 (componentele secvenței S sunt triplete)

=> (2, 1, 4), (2, 4, 6), (3, 2, 4), (5, 1, 5), (7, 4, 6)

Ideea algoritmului

Vom folosi:
- C_i – obiect comparator (relație de ordine) care compară 2 tupluri după dimensiunea i; avem câte un astfel de obiect comparator pentru fiecare dintre cele *d* dimensiuni
- *stableSort(S, c)* – algoritm de sortare **stabilă** care folosește un comparator c

✓ Sortarea Lexicografică apelează algoritmul *stableSort* de *d* ori, o dată pentru fiecare dimensiune

Cu care dintre dimensiuni începem sortarea? În ce ordine considerăm cele d dimensiuni în apelurile algoritmului *stableSort*?

✓ În ordine **inversă**, adică de la cea mai puțin semnificativă (dimensiunea / componenta *d*) la cea mai semnificativă (prima dimensiune / componentă).

Observație: Dacă am începe ordonarea cu prima dimensiune și finaliza-o cu ultima dimensiune, pentru exemplul considerat, s-ar obține:

- Inițial: S: (**7**, 4, 6), (**5**, 1, 5), (**2**, 4, 6), (**2**, 1, 4), (**3**, 2, 4)
- Ulterior ordonării după **prima** componentă: S: (**2**, **4**, 6), (**2**, **1**, 4), (**3**, **2**, 4), (**5**, **1**, 5), (**7**, **4**, 6)
- Ulterior ordonării după **a doua** componentă: S: (2, **1**, **4**), (5, **1**, **5**), (3, **2**, **4**), (2, **4**, **6**), (7, **4**, **6**)
- Ulterior ordonării după **a treia** componentă: S**:** (2, 1, **4**), (3, 2, **4**), (5, 1, **5**), (2, 4, **6**), (7, 4, **6**)

Observăm că ordinea obținută **nu** este cea lexicografică.

Dacă, însă, începem cu ultima dimensiune și finalizăm cu prima, obținem:

- Inițial: S: (**7**, 4, 6), (**5**, 1, 5), (**2**, 4, 6), (**2**, 1, 4), (**3**, 2, 4)
- Ulterior ordonării după **a treia** componentă: S: (**2**, **1**, 4), (**3**, **2**, 4), (**5**, **1**, 5), (**7**, **4**, 6), (**2**, **4**, 6)
- Ulterior ordonării după **a doua** componentă: S: (2, **1**, **4**), (5, **1**, **5**), (3, **2**, **4**), (7, **4**, **6**), (2, **4**, **6**)
- Ulterior ordonării după **prima** componentă: S**:** (2, 1, **4**), (2, 4, **6**), (3, 2, **4**), (5, 1, **5**), (7, 4, **6**)

Observăm că ordinea obținută este cea lexicografică.

 Este necesar ca algoritmul de sortare care se aplică (pentru a sorta după o anumită dimensiune) să fie un algoritm de sortare stabil?

✓ Da.

Altfel, în cazul exemplului considerat, ulterior sortării după prima componentă s-ar putea obține:

- Ulterior ordonării după **a doua** componentă: S: (2, **1**, **4**), (5, **1**, **5**), (3, **2**, **4**), (7, **4**, **6**), (2, **4**, **6**)
- Ulterior ordonării după **prima** componentă: S**:** (2, 4, **6**), (2, 1, **4**), (3, 2, **4**), (5, 1, **5**), (7, 4, **6**)

Observăm că ordinea obținută **nu** este cea lexicografică.
Intuiție: Dacă algoritmul de sortare folosit nu este stabil, atunci la sortarea după dimensiunea *i* nu asigură păstrarea ordinii obținute prin sortarea după dimensiunea *i+1*.

Descrierea Pseudocod a algoritmului se Sortare Lexicografică:

```
Algoritm LexicographicSort(S):
    Pentru i ← d, 1 exec: //pentru fiecare dimensiune i a tuplurilor din S, începând cu
ultima și finalizând cu prima
        stableSort(S, cᵢ) //sortăm tuplurile din S după dimensiunea i
    sf_pentru
sf_subalgoritm
```

Exemplu

- Inițial: S: (**7**, 4, 6), (**5**, 1, 5), (**2**, 4, 6), (**2**, 1, 4), (**3**, 2, 4) => d = 3
- Ulterior ordonării după d=3 componentă: S: (**2**, **1**, 4), (**3**, **2**, 4), (**5**, **1**, 5), (**7**, **4**, 6), (**2**, **4**, 6)
- Ulterior ordonării după d=2 componentă: S: (2, **1**, **4**), (5, **1**, **5**), (3, **2**, **4**), (7, **4**, **6**), (2, **4**, **6**)
- Ulterior ordonării după d=1 componentă: S**:** (2, 1, **4**), (2, 4, **6**), (3, 2, **4**), (5, 1, **5**), (7, 4, **6**)

Complexitatea timp a algoritmului de Sortare Lexivografică (în funcție de complexitatea timp a algoritmului de sortare stabilă folosit):

✓ Θ (d * T(n)) , unde T(n) – complexitatea algoritmului *stableSort*

C. Radix Sort

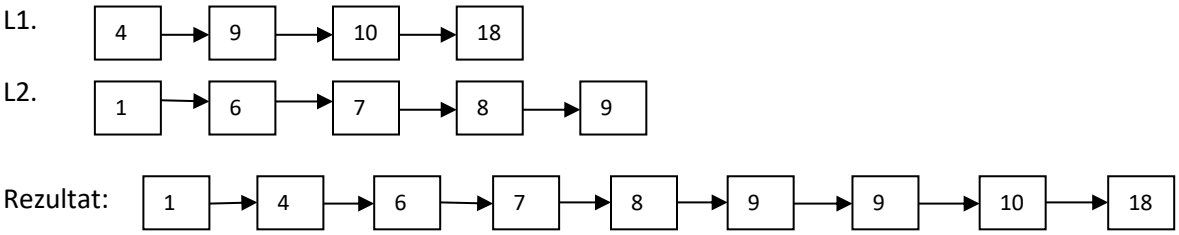
- *RadixSort* este o specializare a sortării lexicografice, în care algoritmul de sortare stabilă folosit este BucketSort => toate elementele din tupluri trebuie să fie numere naturale din mulțimea {0, 1,…, N-1} (altfel, este necesară adaptarea asocierilor din chei și indecși).

- Complexitatea *RadixSort* va fi, prin urmare Θ (d * (n + N))

Interclasarea a două liste simplu înlănțuite, alocate dinamic

Enunț Scrieți o procedură care interclasează două liste ordonate (LO) reprezentate pe liste simplu înlănțuite (LSI) alocate dinamic. Analizați complexitatea operației.

Exemplu



Reprezentarea TAD LO (Listă Ordonată) folosind LSI alocată dinamic:

NodT:
e: TComparabil
urm: ↑NodT

LO:
prim: ↑NodT
R: Relație: TComparabil x TComparabil -> {A, F}

Descrierea în Pseudocod a algoritmului de interclasare a două liste ordonate L1 și L2, reprezentate la LSI alocate dinamic, astfel încât interclasarea să **nu** afecteze L1 și L2, ci să construiască o nouă listă rezultat, LR:

```
Subalgoritm interclasare (L1, L2, LR):
    curentL1 ← L1.prim    //începem parcurgerea listei L1 cu primul ei nod
    curentL2 ← L2.prim    //începem parcurgerea listei L2 cu primul ei nod
    primLR ← NIL          //inițial, lista rezultat este vidă
    ultimLR ← NIL         //reținem și pointerul la ultimul nod al listei rezultat, pentru a
eficientiza adăugarea la finalul ei
    câttimp curentL1 ≠ NIL și curentL2 ≠ NIL execută //până când mai avem noduri de
parcurs în ambele liste
```

```
        aloca(nou) //alocăm un nou nod
        [nou].urm ← NIL //pe care îl vom adăuga la finalul listei rezultat, după ce îl completăm cu informaţia utilă
        dacă L1.R([curentL1].e,[curentL2].e) atunci //verificăm care dintre cele 2 elemente curente din L1 şi L2 este mai mic în raport cu relaţia de ordine R
            [nou].e ← [curentL1].e  //dacă acesta este elementul curent din L1, atunci el va fi completat ca informaţie utilă în noul nod
            curentL1 ← [curentL1].urm //şi avansăm în L1
        altfel
            [nou].e ← [currentL2].e //elementul curent din L2 va fi completat ca informaţie utilă în noul nod
            curentL2 ← [curentL2].urm //şi avansăm în L2
        sf_dacă
        dacă primLR = NIL atunci //verificăm dacă noul nod pe care îl adăugăm îl LR este şi primul
            primLR ← nou //în caz afirmativ, iniţializăm primLR (cu pointerul la primul nod al listei rezultat)
        altfel
            [ultimLR].urm ← nou //altfel, facem legătura de la ultimul nod actual din LR la acest nod
        sf_dacă
        ultimLR ← nou //noul nod devine ultimul nod din lista rezultat
    sf_câttimp
   //După ce am epuizat cel puţin una dintre liste, ne rămâne să parcurgem lista în continuare lista rămasă (dacă au rămas noduri neparcuse într-una dintre liste) şi să preluăm elementele din nodurile rămase în noi noduri pe care să le adăugăm la sfârşitul listei rezultat
    curent ← curentL1 // presupunem că mai avem noduri de parcurs în L1
    dacă curentL1 = NIL atunci //în caz contrar
        curent ← curentL2 //presupunem că mai avem noduri de parcurs în L2
    sf_dacă
    câttimp curent ≠ NIL execută //cât timp mai avem noduri de parcurs în lista ne-epuizată (OBS: dacă am parcurs integral ambele LSI, acest ciclu nu va efectua niciun pas)
        alocă (nou) // alocăm un nou nod
        [nou].urm ← NIL //pe care îl vom adăuga la finalul listei rezultat, după ce îl completăm cu informaţia utilă
        [nou].e ← [curent].e //elementul din nodul curent din LSI încă neparcursă integral va fi completat ca informaţie utilă în noul nod
        curent ← [curent].urm //şi avansăm în LSI de intrare
        dacă primLR = NIL atunci //verificăm dacă noul nod pe care îl adăugăm îl LR este şi primul (<= dacă una dintre L1 şi L2 a fost vidă)
            primLR ← nou //în caz afirmativ, iniţializăm primLR (cu pointerul la primul nod al listei rezultat)
        altfel
            [ultimLR].urm ← nou //altfel, facem legătura de la ultimul nod actual din LR la acest nod
        sf_dacă
    ultimLR ← nou //noul nod devine ultimul nod din lista rezultat
    sf_câttimp
LR.prim ← primLR //iniţializăm câmpul prim al listei rezultat cu pointerul la primul ei nod
//OBS (considerăm că s-a iniţializat câmpul R în prealabil, în constructor)
Sf_subalgoritm
```

Complexitatea algoritmului de interclasare:

- ✓ Θ(n + m), unde n este lungimea listei L1, iar m este lungimea listei L2.