Bârdiță Mihai-Andrei, Gr. 1, An 1, 23.01.2022

Examen PA

1. [7, 10, 3, 15, 8, 1, 12, 18, 6, 9]

a) Sortare prin enumerare

Fiecare cheie este comparată cu toate celelalte numărându-se câte sunt mai mici decât cheia curentă. În felul acesta se găsește poziția pe care trebuie trecută fiecare cheie.

```
function Enumerare (arr)
    for i=0 to k do          //k = cheia
        c[i] ← 0
    end_for
    for j=0 to n do
        c[arr[j]] ← c[arr[j]] + 1
    end_for
    for i=1 to k do
        c[i] ← c[i] + c[i-1]
    end_for
    for j = n-1 to 0 do
        B[c[arr[j]]-1] ← arr[j]   //B = arrayul sortat
        c[arr[j]] ← c[arr[j]] - 1
    end_for
end_function
```

6) Metoda Bulelor

1. Parcurgem vectorul și pt. oricare 2 elemente vecine care sunt mai mari, le interschimbăm valorile.
2. După o singură parcurgere, ultimul element este cel mai mare, deci repetăm parcurgerea
3. dacă o parcurgere nu conține interschimbări atunci vectorul este sortat.

```
function Bule(arr, n)
    repeat
        sortat <- true
        // parcurg lista până când sortat este fals.
        for i=0 to n-1 do
            // interschimbă valorile
            if arr[i] > arr[i+1] do
                aux <- arr[i]
                arr[i] <- arr[i+1]
                arr[i+1] <- aux
                sortat <- false
            end-if
        end-for
    until sortat = false
end-function
```

c) Quick sort

Alege un element capivot și partiază matricea dată în jurul pivotului ales. Ținta este de a pune toate elementele mai mici decât pivotul la stânga și toate elementele mai mari în dreapta.

start_ndx = indexul de start
end_ndx = indexul final

```
function Quicksort(arr, start_ndx, end_ndx)
    if start_ndx < end_ndx do
        partitie_index ← partition(arr, start_ndx, end_ndx)

        //recursivitate
        Quicksort(arr, start_ndx, partitie_index - 1)
        Quicksort(arr, partitie_index + 1, end_ndx)
    end_if
end_function
```

2. Operații în listă și stivă (static și dinamic)

O listă este o colecție de $n \geq 0$ elemente $x[1], x[2], \ldots, x[n]$ toate de același tip. Operații:
- accesul la un element
- ștergere element
- inserare element

Într-o stivă elementul șters este cel mai recent adăugat (LIFO - Last In, First Out).

```
function Pune_in_Stiva(S, x)        // S=stiva, adăugare, push
    if varf(S) < n do
        varf(S) ← varf(S) + 1
        S[varf(S)] ← x
    end_if
end_function

function Scoate_din_Stiva(S)        // S=stiva, ștergere, POP
    if varf(S) ≠ 0 do
        varf(S) ← varf(S) - 1
        return S[varf(S) + 1]       // întoarce rezultatul
    end_if
    return NULL
end_function
```

```
function acces_element (arr, x)   // verifică dacă există x
    for i=0 to n do
        if arr[i] = x do
            return true
        end_if
    end_for
    return false
end_function
```

3. Coada circulară (inserare, ștergere)

Într-o coadă elementul șters este cel care a stat cel mai mult în cadrul structurii (adică primul).
FIFO - First In, First Out.

```
function adaugă_la_coadă (val)   // inserare, push în coadă
    if Rear ≠ Max do   // ca să nu depășească
        if Rear = 0 do
            Front ← Front + 1
        end_if
        Rear ← Rear + 1
        Coadă[Rear] ← val   // se adaugă valoarea în coadă
    end_if   return NULL
end_function
```

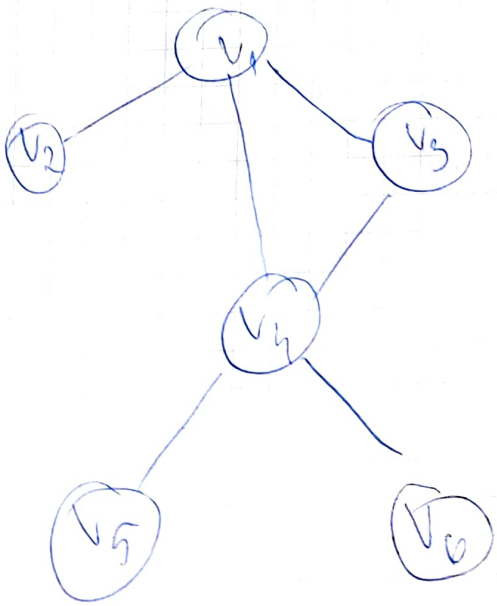
```
function stergе_din_coada ( &x ) // & pt. a modifica lista
    if Front ≠ Max do // să nu depășească
        if Front = Rear do
            Rear ← Rear + 1
        end_if
        x ← Coadă[Front] // scoate valoarea x
        Front ← Front + 1
    end_if
    return NULL
end_function
```

4. Grafuri (reprezentare și parcurgere)

Grafurile sunt de două tipuri: orientate și neorientate.
Un graf G este o pereche ordonată (X, T), unde
X = mulțimea vârfurilor
T = — // — muchiilor

Luăm graful:

Reprezentarea unui graf



Matricea de adiacență este:

$A =$

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	0	0	0	0
v_2	1	0	1	1	0	0
v_3	0	1	0	1	0	0
v_4	0	1	1	0	1	1
v_5	0	0	0	1	0	0
v_6	0	0	0	1	0	0

Parcurgerea grafului poate fi de tip DFS (în lungime) sau BFS (în adâncime)

iterativ:

DFS

fie s o stivă, adaugă (operație de adăugare), v - nod

```
s.add(v)
if v ≠ descoperit do
    v ← descoperit
    foreach muchie adiacentă lui v do
        s.add(vecin)
    end_foreach
end_if
```

BFS

```
function BFS(v)
    foreach vârf k adiacent lui v do
        if vf. k ≠ vizitat do
            BFS(k) // recursivitate
        end-if
    end-foreach
end-function
```