Proiectarea algoritmilor

Problema sortării

Mitică Craus

Cuprins

Sortare - Considerații generale

- Sortarea este una dintre problemele cele mai importante atât din punct de vedere practic, cât și teoretic.
- Cea mai generală formulare și mai des utilizată:

  Fie dată o secvență $(v_0, \ldots, v_{n-1})$ *cu componentele* v_i *dintr-o mulțime total ordonată. Problema sortării constă în determinarea unei permutări* π *astfel încât* $v_{\pi(0)} \leq v_{\pi(1)} \leq \cdots \leq v_{\pi(n-1)}$ *și în rearanjarea elementelor din secvență în ordinea dată de permutare.*

- O altă formulare:

  Fie dată o secvență de structuri statice $(R_0, \ldots, R_{n-1})$*, unde fiecare structură* R_i *are o valoare cheie* K_i*. Peste mulțimea cheilor* K_i *este definită o relație de ordine totală. Problema sortării constă în determinarea unei permutări* π *cu proprietatea* $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$ *și în rearanjarea structurilor în ordinea* $(R_{\pi(0)}, \ldots, R_{\pi(n-1)})$.

- Dacă secvența este memorată în memoria internă a calculatorului, atunci avem *sortare internă*.
- Dacă secvența este înregistrată într-un fișier memorat pe un suport extern, atunci avem *sortare externă*.
- În această lecție ne ocupăm numai de sortarea internă.

Considerații generale - contiunuare

- Vom simplifica formularea problemei presupunând că secvența dată este memorată într-un tablou unidimensional.
- Acum problema sortării poate fi formulată astfel:

  Intrare: *n și tabloul* $(\mathtt{a[i]} \mid i = 0, \ldots, n-1)$ *cu* $a[i] = v_i$, $i = 0, \ldots, n-1$.
  Ieșire: *tabloul* a *cu proprietățile:* $a[i] = w_i$ *pentru* $i = 0, \ldots, n-1$, $w_0 \leq \cdots \leq w_{n-1}$ *și* $(w_0, \ldots, w_{n-1})$ *este o permutare a secvenței* $(v_0, \ldots, v_{n-1})$*; convenim să notăm această proprietate prin* $(w_0, \ldots, w_{n-1}) = \mathrm{Perm}(v_0, \ldots, v_{n-1})$.

- Un algoritm de sortare este stabil dacă păstrează ordinea relativă inițială a elementelor egale.
- Există foarte mulți algoritmi care rezolvă problema sortării interne.
- Vom prezenta numai câteva metode pe care le considerăm cele mai semnificative.

Sortare bazată pe comparații

- Determinarea permutării se face comparând la fiecare moment două elemente $a[i]$ și $a[j]$ ale tabloului supus sortării.
- Scopul comparării poate fi diferit:
  - pentru a rearanja valorile celor două componente în ordinea firească (sortare prin interschimbare), sau
  - pentru a insera una dintre cele două valori într-o subsecvență ordonată deja (sortare prin inserție), sau
  - pentru a selecta o valoare ce va fi pusă pe poziția sa finală (sortare prin selecție).
- Decizia apartenenței unei metode la una dintre subclasele de mai sus are un anumit grad de subiectivitate.
- Exemplu: selecția naivă ar putea fi foarte bine considerată o metodă bazată pe interschimbare.

Sortarea prin interschimbarea elementelor vecine (*bubble-sort*)

- Notăm cu $SORT(\mathtt{a})$ predicatul care ia valoarea *true* dacă și numai dacă tabloul a este sortat.
- Metoda *bubble-sort* se bazează pe următoarea definiție a predicatului $SORT(\mathtt{a})$:

$$SORT(\mathtt{a}) \iff (\forall i)(0 \leq i < n-1) \Rightarrow a[i] \leq a[i+1]$$

- O pereche (i,j), cu $i < j$, formează o *inversiune (inversare)*, dacă $a[i] > a[j]$.
- Pe baza definiției de mai sus vom spune că tabloul a este sortat dacă și numai dacă nu există nici o inversiune $(i, i+1)$.
- Metoda *bubble-sort* propune parcurgerea iterativă a tabloului a și, la fiecare parcurgere, ori de câte ori se întâlnește o inversiune $(i, i+1)$ se procedează la interschimbarea $\mathtt{a[i]} \leftrightarrow \mathtt{a[i+1]}$.

Sortarea prin interschimbarea elementelor vecine (*bubble-sort*) - continuare

- La prima parcurgere, elementul cel mai mare din secvență formează inversiuni cu toate elementele aflate după el și, în urma interschimbărilor realizate, acesta va fi deplasat pe ultimul loc care este și locul său final.
- În iterația următoare, se va întâmpla la fel cu cel de-al doilea element cel mai mare.
- În general, dacă subsecvența $\mathtt{a}[\mathtt{r}+1..\mathtt{n}-1]$ nu are nici o inversiune la iterația curentă, atunci ea nu va avea inversiuni la nici una din iterațiile următoare.
- Aceasta permite ca la iterația următoare să fie verificată numai subsecvența $\mathtt{a}[0..\mathtt{r}]$.
- Terminarea algoritmului este dată de faptul că la fiecare iterație numărul de interschimbari este micșorat cu cel puțin 1.

Algoritmul *buble-sort* - pseudocod

```
procedure Sort(a, n)
    ultim ← n-1
    while (ultim > 0) do
        n1  ← ultim - 1
        ultim  ← 0
        for i ← 0 to n1 do
            if (a[i] > a[i+1])
                then interschimba(a[i], a[i+1])
                     ultim  ← i
end
```

Evaluarea algoritmului

- Cazul cel mai favorabil este întâlnit atunci când secvența de intrare este deja sortată, caz în care algoritmul `bubbleSort` execută $O(n)$ operații.
- Cazul cel mai nefavorabil este obținut când secvența de intrare este ordonată descrescător și, în această situație, procedura execută $O(n^2)$ operații.

Sortare *impar-par*

- Esența metodei *impar-par* constă în alternarea secvenței de operații `swap(a[i],a[i+1]), i=0,2,4,...` cu secvența `swap(a[i],a[i+1]), i=1,3,5,...`
- După cel mult $n/2$ execuții a fiecăreia din cele două secvențe de operații, elementele tabloului a vor fi sortate.
- Metoda nu conduce la un algoritm superior algoritmului `bubbleSort`.
- Algoritmul de sortare prin metoda *impar-par* are calitatea de a fi paralelizabil, datorită faptului că perechile care interacționează sunt disjuncte.

Algoritmul de sortare *impar-par* - pseudocod

```
procedure imparparSort(a, n)
   for faza ← 1 to n do
       if faza este impara
          then for i ← 0 to n-2 step 2 do
                   swap(a[i], a[i+1])
       if faza este para
          then for i ← 1 to n-2 step 2 do
                   swap(a[i], a[i+1])
end
```

Exemplu de execuție a algoritmului de sortare *impar-par*

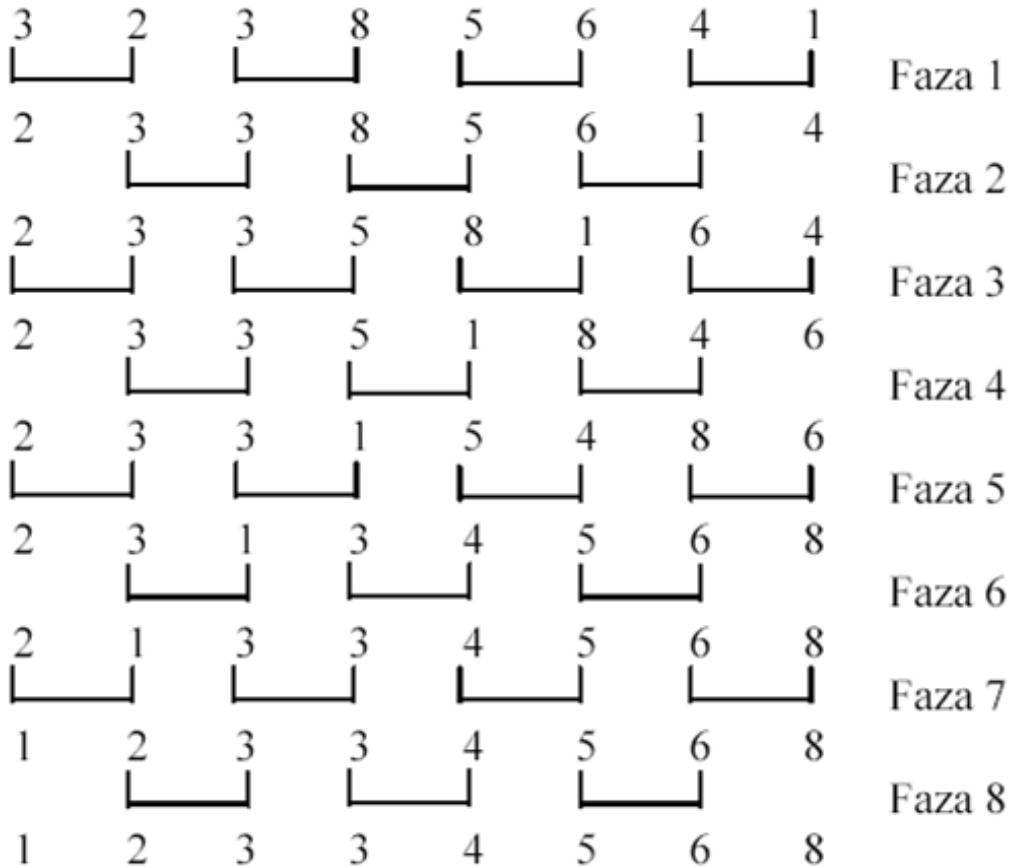


Figura 1 : Exemplu de execuție a algoritmului de sortare *impar-par* pentru $n = 8$

Sortare prin inserție directă

- Algoritmul sortării prin inserție directă consideră că în pasul k, elementele `a[0..k-1]` sunt sortate crescător, iar elementul `a[k]` va fi inserat, astfel încât, după această inserare, primele elemente `a[0..k]` să fie sortate crescător.
- Inserarea elementului `a[k]` în secvența `a[0..k-1]` presupune:
  1. memorarea elementului într-o variabilă temporară;
  2. deplasarea tuturor elementelor din vectorul `a[0..k-1]` care sunt mai mari decât `a[k]`, cu o poziție la dreapta (aceasta presupune o parcurgere de la dreapta la stânga);
  3. plasarea lui `a[k]` în locul ultimului element deplasat.

Algoritmul de sortare prin inserție directă - pseudocod

```
procedure Sort(a, n)
    for k ← 1 to n-1 do
        i ← k-1
        temp ← a[k]
        while ((i ≥ 0) and (temp < a[i])) do
            a[i+1] ← a[i]
            i ← i-1
        if (i ≠ k-1) then a[i+1] ← temp
end
```

Evaluarea algoritmului

- Căutarea poziției i în subsecvența `a[0..k-1]` necesită $O(k-1)$ timp.
- Timpul total în cazul cel mai nefavorabil este $O(1+\cdots+n-1)=O(n^2)$.
- Pentru cazul cel mai favorabil, când valoarea tabloului la intrare este deja în ordine crescătoare, timpul de execuție este $O(n)$.

Sortare prin metoda lui Shell

- În algoritmul *insertionSort* elementele se deplasează numai cu câte o poziție o dată și prin urmare timpul mediu va fi proporțional cu n^2, deoarece fiecare element se deplasează în medie cu $n/3$ poziții în timpul procesului de sortare.
- Din acest motiv s-au căutat metode care să îmbunătățească inserția directă, prin mecanisme cu ajutorul cărora elementele fac salturi mai lungi, în loc de pași mici.
- O asemenea metodă a fost propusă în anul 1959 de Donald L. Shell, metodă pe care o vom mai numi *sortare cu micșorarea incrementului*.
- Exemplul următor ilustrează ideea generală care stă la baza metodei.

Sortare prin metoda lui Shell - exemplu

- Presupunem $n = 9$. Sunt executați următorii pași (figura 2):
  1. *Prima parcurgere a secvenței*. Se împart elementele în grupe de câte trei sau două elemente (valoarea incrementului $h_1 = 4$) care se sortează separat:
  $$(a[0], a[4], a[8]), \ldots, (a[3], a[7])$$
  2. *A doua parcurgere a secvenței*. Se grupează elementele în două grupe de câte trei elemente (valoarea incrementului $h_2 = 3$): $(a[0], a[3], a[6]) \ldots (a[2], a[5], a[8])$ și se sortează separat.
  3. *A treia trecere*. Acest pas termină sortarea prin considerarea unei singure grupe care conține toate elementele. În final, cele nouă elemente sunt sortate.

Sortare prin metoda lui Shell - exemplu (continuare)

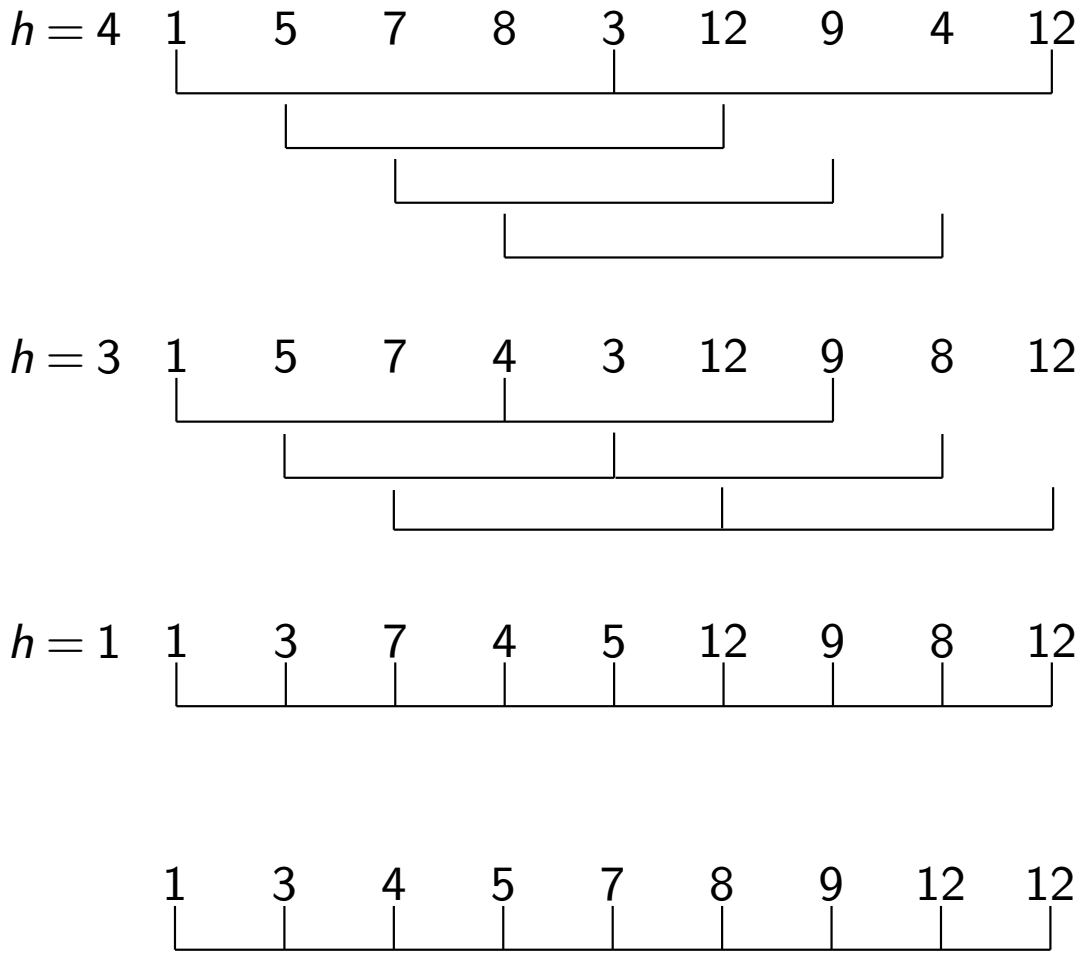


Figura 2 : Sortarea prin metoda lui Shell

Sortare prin metoda lui Shell - exemplu (continuare)

- Fiecare dintre procesele intermediare de sortare implică fie o sublistă nesortată de dimensiune relativ scurtă, fie una aproape sortată, astfel că inserția directă poate fi utilizată cu succes pentru fiecare operație de sortare.
- Prin inserțiie intermediare, elementele tind să conveargă rapid spre destinația lor finală.
- Secvența de incremente 4, 3, 1 nu este obligatorie; poate fi utilizată orice secvență $h_i > h_{i-1} > \cdots > h_0$, cu condiția ca ultimul increment h_0 să fie 1.

Algoritmul de sortare prin metoda lui Shell - pseudocod

- Presupunem că numărul de valori de incrementare este memorat de variabila `nIncr` și că acestea sunt memorate în tabloul $(\texttt{valIncr[h]} \mid 0 \leq k \leq nIncr - 1)$.

```
procedure ShellSort(a, n)
   for k ← nIncr-1 downto 0 do
       h ← valIncr[h]
       for i ← h to n-1 do
           temp ← a[i]
           j ← i-h
           while ((j ≥ 0) and (temp < a[j])) do
              a[j + h] ← a[j]
              j ← j-h
           if (j+h ≠ i) then a[j+h] ← temp
end
```

Evaluarea algoritmului - cazul cel mai favorabil și cazul cel mai nefavorabil

- Presupunem că elementele din secvență sunt diferite și dispuse aleator.
- Denumim operația de sortare corespunzătoare primei faze h_t-*sortare*, apoi h_{t-1}-*sortare* etc.
- O subsecvență pentru care $a[i] \leq a[i+h]$, pentru $0 \leq i \leq n-1-h$, este denumită *h-ordonată*.
- Considerăm pentru început cea mai simplă generalizare a inserției directe, și anume cazul când avem numai două valori de incrementare: $h_1 = 2$ și $h_0 = 1$.
- Cazul cel mai favorabil este obținut când secvența de intrare este ordonată crescător și sunt executate $2O(n) = O(n)$ comparații și nici o deplasare.
- Cazul cel mai nefavorabil, când secvența de intrare este ordonată descrescător, necesită $2O(1+2+3+(\frac{n}{2}-1)) + O(1+2+3+(n-1)) = O(n^2)$ comparații și $2O(1+2+3+(\frac{n}{2}-1)) + O(1+2+3+(n-1)) = O(n^2)$ deplasări.

Evaluarea algoritmului - cazul mediu

- În cea de-a doua parcurgere a tabloului a, avem o secvență 2-ordonată de elemente $a[0], a[1], \ldots, a[n-1]$.
- Este ușor de văzut că numărul de permutări $(i_0, i_1, \ldots i_{n-1})$ ale mulțimii $\{0, 1, \ldots, n-1\}$ cu proprietatea $i_k \leq i_{k+2}$, pentru $0 \leq k \leq n-3$, este $C_n^{[\frac{n}{2}]}$, deoarece obținem exact o permutare 2-ordonată pentru fiecare alegere de $[\frac{n}{2}]$ elemente care să fie puse pe poziții impare.

Evaluarea algoritmului - cazul mediu (continuare)

- Fiecare permutare 2-ordonată este egal posibilă după ce o subsecvență aleatoare a fost 2-ordonată.
- Determinăm numărul mediu de inversări între astfel de permutări.
- Fie A_n numărul total de inversări peste toate permutările 2-ordonate de $\{0, 1, \ldots, n-1\}$.
- Relațiile $A_1 = 0$, $A_2 = 1$, $A_3 = 2$ sunt evidente. Considerând cele șase cazuri 2-ordonate, pentru $n = 4$,

$$0\,1\,2\,3 \quad 0\,2\,1\,3 \quad 0\,1\,3\,2 \quad 1\,0\,2\,3 \quad 1\,0\,3\,2 \quad 2\,0\,3\,1$$

  vom găsi $A_4 = 0+1+1+2+3 = 8$.
- În urma calculelor, care sunt un pic dificile, se obține pentru A_n o formă destul de simplă:

$$A_n = \left[\frac{n}{2}\right] 2^{n-2}$$

- Numărul mediu de inversări într-o permutare aleatoare 2-ordonată este:

$$\frac{\left[\frac{n}{2}\right] 2^{n-2}}{C_n^{[\frac{n}{2}]}}$$

- După aproximarea lui Stirling, aceasta converge asimptotic către $\frac{\sqrt{\pi}}{128 n^{\frac{3}{2}}} \approx 0.15 n^{\frac{3}{2}}$.

Evaluarea algoritmului - cazul mediu (continuare)

Teorema (1)

Numărul mediu de inversări executate de algoritmul lui Shell pentru secvența de incremente $(2,1)$ *este* $O(n^{\frac{3}{2}})$.

Evaluarea algoritmului - continuare

Teorema (2)

Dacă $h \approx \left(\frac{16n}{\pi}\right)^{\frac{1}{3}}$, *atunci algoritmul lui Shell necesită timpul* $O(n^{\frac{5}{3}})$ *pentru cazul cel mai nefavorabil. (Knuth, 1976, pag. 89).*

Evaluarea algoritmului - continuare

Teorema (3)

Dacă secvența de incremente $h_{t-1}, \ldots, h_0$ *satisface condiția*

$$h_{s+1} \bmod h_s = 0 \text{ pentru } 0 \leq s < t-1,$$

atunci timpul de execuție pentru cazul cel mai nefavorabil este $O(n^2)$.

- O justificare intuitivă a teoremei de mai sus este următoarea:
  - Dacă $h_s = 2^s$, $0 \leq s \leq 3$, atunci o 8-sortare urmată de o 4-sortare, urmată de o 2-sortare nu permite nici o interacțiune între elementele de pe pozițiile pare și impare.
  - Fazei finale (1-sortaree) îi vor reveni $O(n^{\frac{3}{2}})$ inversări.
  - O 7-sortare urmată de o 5-sortare, urmată de o 3-sortare amestecă astfel lucrurile încât în faza finală (1-sortarea) nu vor fi mai mult de $2n$ inversări.

Teorema (4)

Timpul de execuție în cazul cel mai nefavorabil a algoritmului `ShellSort` *este* $O(n^{\frac{3}{2}})$, *atunci când* $h_s = 2^{s+1} - 1,\ 0 \leq s \leq t-1, t = [\log_2 n] - 1$.

Sortare prin selecție

- Strategiile de sortare incluse în această clasă se bazează pe următoarea schemă :
  - La pasul curent se selectează un element din secvență și se plasează pe locul său final.
  - Procedeul continuă până când toate elementele sunt plasate pe locurile lor finale.
- După modul în care se face selectarea elementului curent, metoda poate fi mai mult sau mai puțin eficientă.

Sortare prin selecție naivă

- Este o metodă mai puțin eficientă, dar foarte simplă în prezentare.
- Se bazează pe următoarea caracterizare a predicatului $SORT(\mathrm{a})$:

$$SORT(\mathrm{a}) \iff (\forall i)(0 \leq i < n) \Rightarrow a[i] = \max\{a[0], \ldots, a[i]\}$$

- Ordinea în care sunt așezate elementele pe pozițiile lor finale este $n-1, n-2, \ldots, 0$.
- O formulare echivalentă este:

$$SORT(\mathrm{a}) \iff (\forall i)(0 \leq i < n) : a[i] = \min\{a[i], \ldots, a[n]\},$$

caz în care ordinea de așezare este $0, 1, \ldots, n-1$.

Algoritmul de sortare prin selecție naivă - pseudocod

```
procedure naivSort(a, n)
for i ← n-1 downto 1 do
    locmax ← 0
    maxtemp ← a[0]
    for j ← 1 to i do
        if (a[j] > maxtemp)
           then locmax ← j
                maxtemp ← a[j]
    a[locmax] ← a[i]
    a[i] ← maxtemp
end
```

Evaluarea algoritmului

- Timpul de execuție este $O(n^2)$ pentru toate cazurile, adică algoritmul `NaivSort` are timpul de execuție $\Theta(n^2)$.

Comparație între BubbleSort și NaivSort

.

- În sortarea prin metoda bulelor sunt efectuate mai puține comparații decât în cazul selecției naive,.
- Paradoxal, sortarea prin metoda bulelor este aproape de două ori mai lentă decât selecția naivă.
  - Explicație: În sortarea prin metoda bulelor sunt efectuate multe schimbări în timp ce selecția naivă implică o mișcare redusă a datelor.
- În tabelul din figura 3 sunt redați timpii de execuție (în sutimi de secunde) pentru cele două metode obținuți în urma a zece teste pentru $n = 1000$.

Nr. test	BubbleSort	NaivSort
1	71	33
2	77	27
3	77	28
4	94	38
5	82	27
6	77	28
7	83	32
8	71	33
9	71	39
10	72	33

Figura 3 : Compararea algoritmilor BubbleSort și NaivSort

Sortare prin selecție sistematică

- Se bazează pe structura de date de tip max-*heap*.
- Metoda de sortare prin selecție sistematică constă în parcurgerea a două etape:
  1. construirea pentru secvența curentă a proprietății MAX-HEAP(a);
  2. selectarea în mod repetat a elementului maximal din secvența curentă și refacerea proprietății MAX-HEAP pentru secvența rămasă.
- Definiție:

  a) Tabloul $(\mathtt{a}[\mathtt{i}] \mid i = 0, \ldots, n-1)$ are proprietatea MAX-HEAP, dacă $(\forall k)(1 \leq k < n \Rightarrow a\left[\frac{k-1}{2}\right] \geq a[k])$.
  - Notăm această proprietate prin MAX-HEAP(a).

  b) Tabloul a are proprietatea MAX-HEAP începând cu poziția ℓ dacă $(\forall k)(\ell \leq \frac{k-1}{2} < k < n \Rightarrow a\left[\frac{k-1}{2}\right] \geq a[k])$.
  - Notăm această proprietate cu MAX-HEAP$(\mathtt{a}, \ell)$.

Etapa 1. Construirea pentru secvența curentă a proprietății MAX-HEAP(a)

- Tabloul a are lungimea n. Inițial, are loc MAX-HEAP(a, $\frac{n}{2}$).
- Dacă are loc MAX-HEAP(a, $\ell+1$) atunci se procedează la introducerea lui $a[\ell]$ în grămada deja construită a[$\ell+1$..n-1], astfel încât să obținem MAX-HEAP(a, ℓ).
- Procesul se repetă până când ℓ devine 0.

```
procedure intrInGr(a, n, ℓ)
    j ← ℓ
    esteHeap ← false
    while ((2*j+1 ≤ n-1) and not esteHeap)
        k ← j*2+1
        if ((k < n-1) and (a[k] < a[k+1])
            then k ← k+1
        if (a[j] < a[k])
            then swap(a[j], a[k])
            else esteHeap ← true
        j ← k
end
```

Etapa 2: Selectarea în mod repetat a elementului maximal din secvența curentă și refacerea proprietății MAX-HEAP pentru secvența rămasă

- Deoarece inițial avem MAX-HEAP(a), rezultă că pe primul loc se găsește elementul maximal din $a[0..n-1]$.
- Punem acest element la locul său final prin interschimbarea $a[0] \leftrightarrow a[n-1]$. Acum $a[0..n-2]$ are proprietatea MAX-HEAP începând cu 1.
- Refacem MAX-HEAP pentru această secventă prin introducerea lui $a[0]$ în grămada $a[0..n-2]$, după care punem pe locul său final cel de-al doilea element cel mai mare din secvența de sortat.
- Procedeul continuă până când toate elementele ajung pe locurile lor finale.

Algoritmul de sortare prin selecție sistematică - pseudocod

```
procedure heapSort(a,n)
    n1 ← ⌈(n−1)/2⌉
    for ℓ ← n1 downto 0 do
        intrInGr(a, n, ℓ)
    r ← n-1
    while (r ≥ 1) do
        swap(a[0], a[r])
        intrInGr(a, r, 0)
        r ← r-1
end
```

Evaluarea algoritmului

- Considerăm $n = 2^k - 1$.
- În faza de construire a proprietății MAX-HEAP pentru toată secvența de intrare sunt efectuate următoarele operații:
  - se construiesc vârfurile de pe nivelurile $k-2, k-3, \ldots$;
  - pentru construirea unui vârf de pe nivelul i se vizitează cel mult câte un vârf de pe nivelurile $i+1, \ldots, k-1$;
  - la vizitarea unui vârf sunt executate două comparații.

  Rezultă că numărul de comparații executate în prima etapă este cel mult:

$$\sum_{i=0}^{k-2} 2(k-i-1)2^i = (k-1)2 + (k-2)2^2 + \cdots + 1 \cdot 2^{k-1} = 2^{k+1} - 2(k+1)$$

- În etapa a II-a, dacă presupunem că $a[r]$ se găsește pe nivelul i, introducerea lui $a[0]$ în grămada $\mathtt{a[1..r]}$ necesită cel mult $2i$ comparații. Deoarece i ia succesiv valorile $k-1, k-2, \ldots, 1$, rezultă că în această etapă numărul total de comparatii este cel mult:

$$\sum_{i=1}^{k-1} 2i2^i = (k-2)2^{k+1} + 4$$

Evaluarea algoritmului - continuare

- Numărul total de comparații este cel mult

$$\begin{aligned} C(n) &= 2^{k+1} - 2(k+1) + (k-2)2^{k+1} + 4 \\ &= 2^{k+1}(k-1) - 2(k-1) \\ &= 2k(2^k - 1) - 2(2^k - 1) \\ &= 2n\log_2 n - 2n \end{aligned}$$

- Complexitatea algoritmului `heapSort` este $O(n \log n)$.

Sortare prin numărare clasică

- Presupunem că tipul `TElement`, peste care sunt definite secvențele supuse sortării, conține numai două elemente: 0 și 1.
- Sortarea unui tablou `a : array[1..n] of TElement` se face astfel:
  - Se determină numărul n_1 de elemente din a egale cu 0 și numărul n_2 de elemente egale cu 1. Aceasta se poate face în timpul $\Theta(n)$.
  - Apoi se pun în tablou n_1 elemente 0 urmate de n_2 elemente 1. Și cea de-a doua etapă necesită $\Theta(n)$ timp.
- Rezultă un algoritm de sortare cu timpul de execuție $\Theta(n)$.
- Obținerea unui algoritm de sortare cu timp de execuție liniar a fost posibilă datorită faptului că s-au cunoscut informații suplimentare despre elementele supuse sortării, anume că mulțimea univers conține numai două elemente.

Sortare prin numărare clasică - continuare

- Algoritmul de sortare prin numărare se utilizează atunci când mulțimea univers conține puține elemente.
- Presupunem că $\mathbb{U} = \{0, 1, \ldots, k-1\}$ cu relația de ordine naturală.
- În prima etapă a algoritmului va fi calculat vectorul p: $p[i]$ = numărul de elemente din a mai mici decât sau egale cu i, $i = 0, 1, \ldots k-1$.
- În etapa a doua a algoritmului tabloul a va fi parcurs de la dreapta la stânga și va fi păstrată invariantă următoarea proprietate:
  $\mathscr{P}$: $p[i]$ va fi poziția ultimului element i transferat din a în tabloul final b,
  $i = k-1, k-2, \ldots 0$.
- Propritetatea $\mathscr{P}$ poate fi păstrată prin decrementarea componentei din p imediat după ce elementul corespunzător din a a fost transferat.

Algoritmul de sortare prin numărare - pseudocod

```
procedure sortNumarare(a, b, n)
    for i ← 0 to k-1 do
        p[i] ← 0
    for j ← 0 to n-1 do
        p[a[j]] ← p[a[j]]+1
    for i ← 1 to k-1 do
        p[i] ← p[i-1] + p[i]
    for j ← n-1 downto 0 do
        i ← a[j]
        b[p[i]-1] ← i
        p[i] ← p[i]-1
end
```

Evaluarea algoritmului

- Determinarea tabloului `p` necesită $O(k+n)$ timp, iar transferul $O(n)$ timp.
- Algoritmul `sortNumarare` are complexitatea timp $O(n+k)$.
- În practică, algoritmul este aplicat pentru $k = O(n)$, caz în care rezultă un timp de execuție liniar a pentru `sortNumarare`.

Sortare prin metoda "bingo" (K. Berman, J. Paul, "Algorithms: Sequential, Parallel and Distributed", Thomson Learning, 2005)

- Numărul elementelor secvenței de sortat este n.
- Numărul elementelor distincte este m, $1 \leq m \leq n$.
- Exemplu: sortarea unei liste de adrese poștale după codul poștal.
- Complexitatea: $O(nm)$.
- Daca $m \leq \log n$, algoritmul `bingoSort` este la fel de performant ca și `heapSort` și `mergeSort`.
- Ideea este de plasa, în fiecare iterație, una din valorile distincte pe pozițiile finale.
- Valoarea care este obiectul procesării într-o iterație se numeste "bingo".

Algoritmul de sortare prin metoda ”bingo” - pseudocod

```
procedure bingoSort(a, n)
    MaxMin(a,n,valMax,valMin)
    bingo ← valMin
    urm ← 0
    urmBingo ← valMax
    while (bingo<valMax) do
        posStart ← urm
        for i ← posStart to n-1 do
            if (A[i]=bingo)
                then interschimba(A[i],A[urm])
                     urm ← urm+1
                else if A[i]<urmBingo then urmBingo ← A[i]
        bingo ← urmBingo
        urmBingo ← valMax
end
```

Sortare prin distribuire

- Algoritmii de sortare prin distribuire presupun cunoașterea de informații privind distribuția acestor elemente.
- Aceste informații sunt utilizate pentru a distribui elementele secvenței de sortat în „pachete" care vor fi sortate în același mod sau prin altă metodă, după care pachetele se combină pentru a obține lista finală sortată.

Sortarea cuvintelor

- Presupunem că avem n fișe, iar fiecare fișă conține un nume ce identifică în mod unic fișa (cheia).
- Se pune problema sortării manuale a fișelor. Pe baza experienței câștigate se procedează astfel:
  - Se împart fișele în pachete, fiecare pachet conținând fișele ale căror cheie începe cu aceeași literă.
  - Apoi se sortează fiecare pachet în aceeași manieră după a doua literă, apoi etc.
  - După sortarea tuturor pachetelor, acestea se concatenează rezultând o listă liniară sortată.
- Vom încerca să formalizăm metoda de mai sus într-un algoritm de sortare a șirurilor de caractere (cuvinte).
- Presupunem că elementele secvenței de sortat sunt șiruri de lungime fixată m definite peste un alfabet cu k litere.
- Echivalent, se poate presupune că elementele de sortat sunt numere reprezentate în baza k. Din acest motiv, sortarea cuvintelor este denumită în engleză *radix-sort* (cuvântul *radix* traducându-se prin *bază*).

Sortarea cuvintelor - continuare

- Dacă urmăm ideea din exemplul cu fișele, atunci algoritmul ar putea fi descris recursiv astfel:
  1. Se împart cele n cuvinte în k pachete, cuvintele din același pachet având aceeași literă pe poziția i (numărând de la stânga la dreapta).
  2. Apoi, fiecare pachet este sortat în aceeași manieră după literele de pe pozițiile $i+1, \ldots, m-1$.
  3. Se concatenează cele k pachete în ordinea dată de literele de pe poziția i. Lista obținută este sortată după subcuvintele formate din literele de pe pozițiile $i, i+1, \ldots, m-1$.
- Inițial se consideră $i = 0$. Apare următoarea problemă:
  - Un grup de k pachete nu va putea fi combinat într-o listă sortată decât dacă cele k pachete au fost sortate complet pentru subcuvintele corespunzătoare.
  - Este deci necesară ținerea unei evidențe a pachetelor, fapt care conduce la utilizarea de memorie suplimentară și creșterea gradului de complexitate a metodei.

Sortarea cuvintelor - continuare

- O simplificare majoră apare dacă împărțirea cuvintelor în pachete se face parcurgând literele acestora de la dreapta la stânga.
- Procedând așa, observăm următorul fapt surprinzător:
  - după ce cuvintele au fost distribuite în k pachete după litera de pe poziția i, cele k pachete pot fi combinate înainte de a le distribui după litera de pe poziția $i-1$.
- Exemplu: Presupunem că alfabetul este $\{0 < 1 < 2\}$ și $m = 3$. Cele trei faze care cuprind distribuirea elementelor listei în pachete și apoi concatenarea acestora într-o singură listă sunt sugerate grafic în figura 4.

Sortarea cuvintelor - continuare

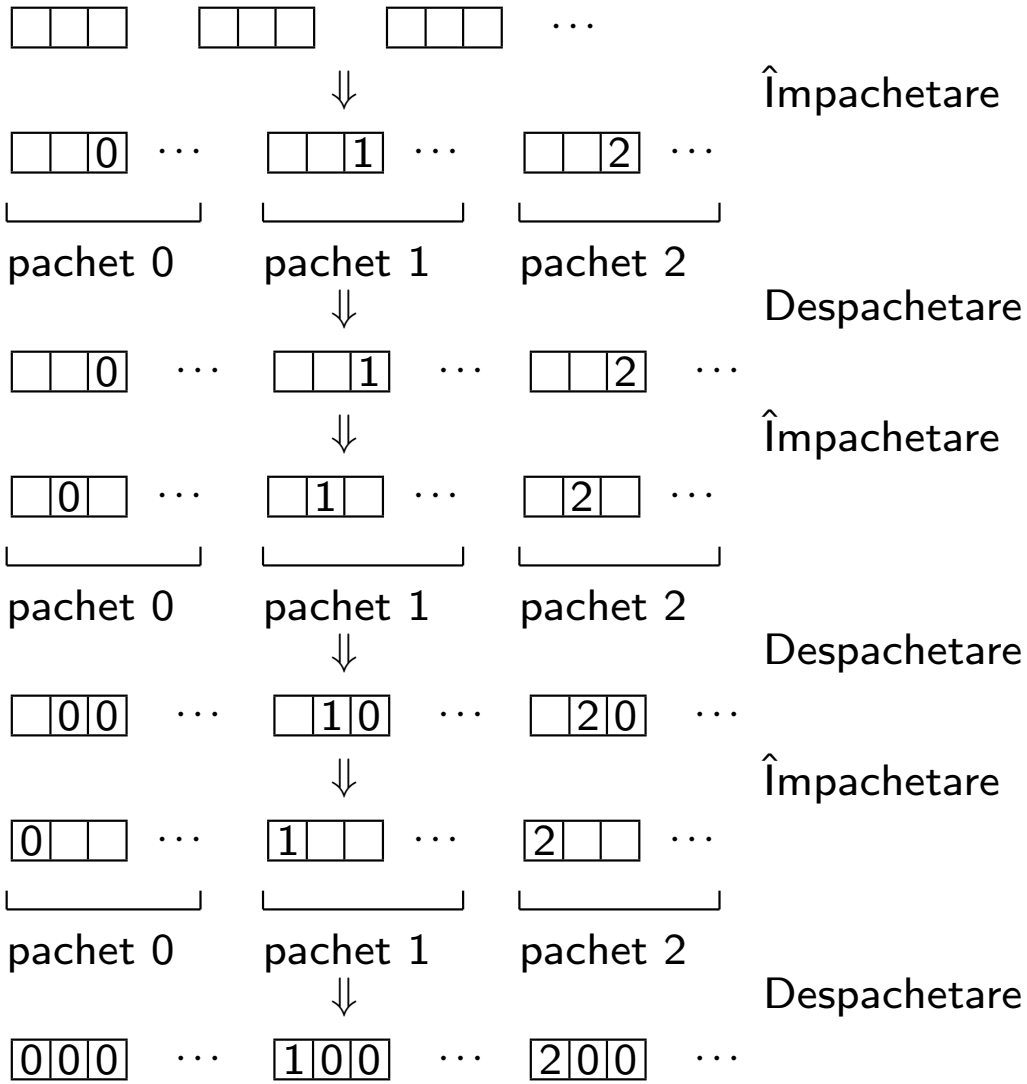


Figura 4 : Sortare prin distribuire

Algoritmul de sortare distribuire - descriere

- Pentru gestionarea pachetelor vom utiliza un tablou de structuri de pointeri numit `pachet`, cu semnificația următoare:
  - $pachet[i]$ este structura de pointeri $pachet[i].prim$ și $pachet[i].ultim$,
  - $pachet[i].prim$ face referire la primul element din lista ce reprezintă pachetul i și
  - $pachet[i].ultim$ face referire la ultimul element din lista corespunzătoare pachetului i.
- Etapa de distribuire este realizată în modul următor:
  1. Inițial, se consideră listele $pachet[i]$ vide.
  2. Apoi se parcurge secvențial lista supusă distribuirii și fiecare element al acesteia este distribuit în pachetul corespunzător.
- Etapa de combinare a pachetelor constă în concatenarea celor k liste $pachet[i],\ i = 0, \ldots, k-1$.

Algoritmul de sortare distribuire - pseudocod

```
procedure radixSort(L , m)
for i ← m-1 downto 0 do
    for j ← 0 to k-1 do
        pachet[j] ← listaVida()
    while (not esteVida(L)) do /* împachetare */
       w ← citeste(L, 0)
       elimina(L, 0)
       insereaza(pachet[w[i]], w)
    for j ← 0 to k-1 do /* despachetare */
        concateneaza(L, pachet[j])
end
```

Evaluarea algoritmului

- Distribuirea în pachete presupune parcurgerea completă a listei de intrare, iar procesarea fiecărui element al listei necesită $O(1)$ operații.
  - Faza de distribuire se face în timpul $O(n)$, unde n este numărul de elemente din listă.
- Combinarea pachetelor presupune o parcurgere a tabloului `pachet`, iar adăugarea unui pachet se face cu $O(1)$ operații, cu ajutorul tabloului `ultim`.
  - Faza de combinare a pachetelor necesită $O(k)$ timp.
- Algoritmul `radixSort` are un timp de execuție de $O(m \cdot n)$.

Sortare topologică - considerații generale

- Se aplică la secvențe cu elemente din mulțimi parțial ordonale.
- Exemplu de relație de ordine este parțială: $a_1 < a_0, a_1 < a_2 < a_3$.
- Problema constă în a crea o listă liniară care să fie compatibilă cu relația de ordine, adică, dacă $a_i < a_j$, atunci a_i va precede pe a_j în lista finală.
- Pentru exemplul nostru, lista liniară finală va putea fi (a_1, a_0, a_2, a_3), sau (a_1, a_2, a_0, a_3), sau (a_1, a_2, a_3, a_0).

Definitia

Fie $(S, \leq)$ o mulțime parțial ordonată finită și $a = (a_0, a_1, \ldots, a_{n-1})$ o liniarizare a sa. Spunem că secvența a este *sortată topologic*, dacă $\forall i, j : a_i < a_j \Rightarrow i < j$.

Sortare topologică - considerații generale (continuare)

Teorema (1)

Orice mulțime parțial ordonată finită $(S, \leq)$ poate fi sortată topologic.

Demonstrație.

Vom extinde relația $\leq$ la o relație de ordine totală.
Fie elementele distincte $a, b \in S$ ce nu pot fi comparate, *i.e.*, nu are loc nici $a < b$ și nici $b < a$.
Extindem relația $<$, considerând $x < y$ pentru orice x cu $x \leq a$ și orice y cu $b \leq y$.
Procedeul continuă până când sunt eliminate toate perechile incomparabile.
O mulțime total ordonată poate fi sortată, iar secvența sortată respectă ordinea parțială. □

Grafuri și digrafuri

- Un *graf* este o pereche $G = (V, E)$, unde V este o mulțime ale cărei elemente sunt numite *vârfuri*, iar E este o mulțime de perechi neordonate $\{u, v\}$ de vârfuri, numite *muchii*.
- Considerăm numai cazul când V și E sunt finite.
- Dacă $e = \{u, v\}$ este o muchie, atunci u și v se numesc extremitățile muchiei e; mai spunem că e este *incidentă* în u și v sau că vârfurile u și v sunt *adiacente (vecine)*.
- Dacă G conține muchii de forma $\{u, u\}$, atunci o asemenea muchie se numește *buclă*, iar graful se numește *graf general* sau *pseudograf*.
- Un *digraf* este o pereche $D = (V, A)$, unde V este o mulțime de vârfuri, iar A este o mulțime de perechi ordonate (u, v) de vârfuri, numite arce.
- Considerăm numai cazul când V și A sunt finite.
- Dacă $a = (u, v)$ este un arc, atunci spunem că u este *extremitatea inițială* (*sursa*) a lui a și că v este *extremitatea finală* (*destinația*) lui a; mai spunem că u este un *predecesor imediat* al lui v și că v este un *succesor imediat* al lui u.
- Formulări echivalente: a este *incident din* u și *incident în (spre)* v, sau a este *incident cu* v *spre interior* și a este *incident cu* u *spre exterior*, sau a *pleacă din* u *și sosește în* v.

Grafuri și digrafuri

- Orice graf $G = (V, E)$ poate fi reprezentat ca un digraf $D = (V, A)$ considerând pentru fiecare muchie $\{i, j\} \in E$ două arce $(i, j), (j, i) \in A$.
- Cu aceste reprezentări, operațiile tipului Graf pot fi exprimate cu ajutorul celor ale tipului Digraf. Astfel, inserarea/ștergerea unei muchii este echivalentă cu inserarea/ștergerea a două arce.

Reprezentarea digrafurilor prin matricea de adiacență

- Digraful D este reprezentat printr-o structură cu trei câmpuri:
  - `D.n` = numărul de vârfuri,
  - `D.m` = numărul de arce și
  - un tablou bidimensional `D.a` de dimensiune $n \times n$ astfel încât:

$$D.a[i,j] = \begin{cases} 0 & \text{dacă } (i,j) \notin A, \\ 1 & \text{dacă } (i,j) \in A. \end{cases}$$

  pentru $i,j = 0,\ldots,n-1$.

- Dacă `D` este reprezentarea unui graf, atunci matricea de adiacență este simetrică:

$$D.a[i,j] = D.a[j,i], \text{ pentru orice } i,j.$$

- În loc de valorile 0 și 1 se pot considera valorile booleene *false* și respectiv *true*.

Reprezentarea digrafurilor prin liste de adiacență

- Reprezentarea prin liste de adiacență exterioară:
  - Digraful D este reprezentat printr-o structură asemănătoare cu cea de la matricele de adiacență.
  - Matricea de adiacență este înlocuită cu un tablou unidimensional de n liste liniare, implementate prin liste simplu înlănțuite și notate cu `D.a[i]` pentru $i = 0, \ldots, n-1$.
  - Lista `D.a[i]` conține vârfurile destinatare ale arcelor care pleacă din i (= lista de adiacență exterioară).
- Reprezentarea prin liste de adiacență interioară:
  - Lista `D.a[i]` conține vârfurile surse ale arcelor care sosesc în i (= lista de adiacență interioară).

Exemplu de reprezentare a grafurilor prin liste de adiacență

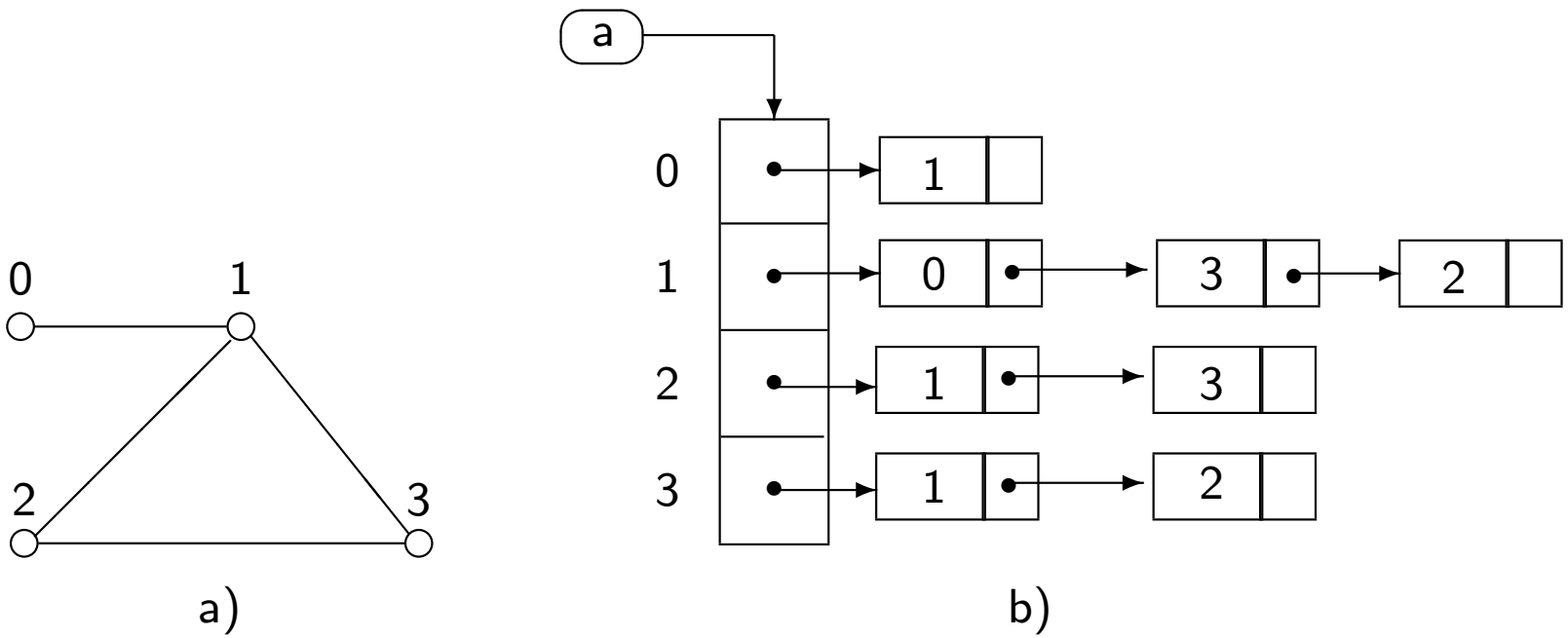


Figura 5 : Graf reprezentat prin liste de adiacență înlănțuite

Exemplu de reprezentare a digrafurilor prin liste de adiacență exterioară

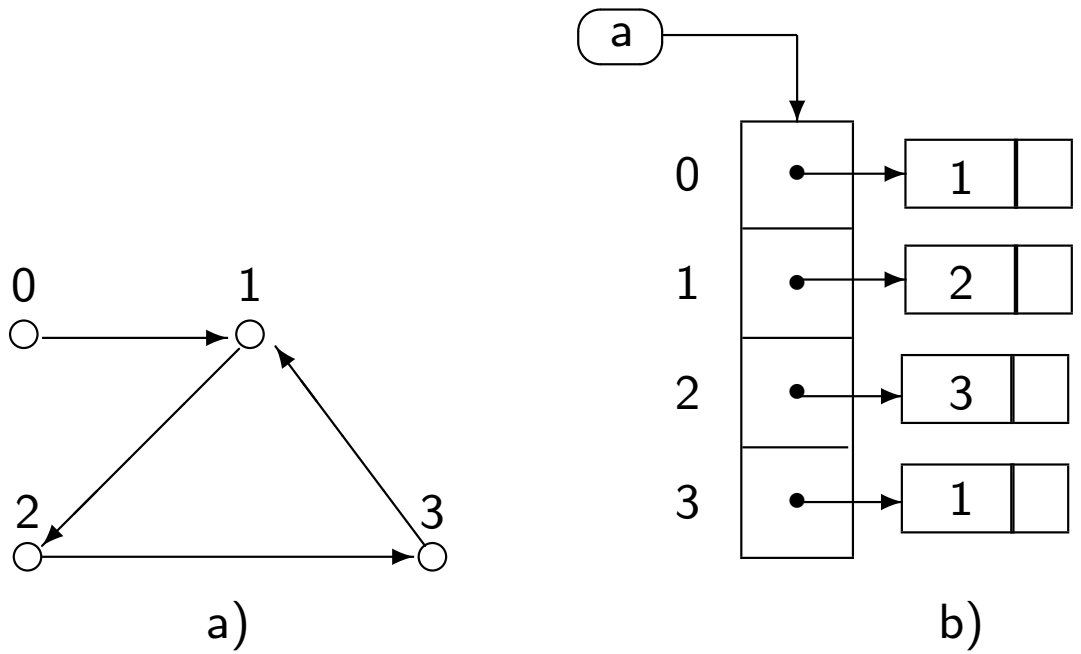


Figura 6 : Digraf reprezentat prin liste de adiacență exterioară înlănțuite

Explorarea DFS (Depth First Search) și BFS (Breadth First Search

- Explorarea DFS:

```
procedure DFS(D, i0, viziteaza(), S)
   for i ← 0 to D.n-1 do
       p[i]  ← D.a[i].prim
       S[i]  ← 0
   Sa ← stivaVida() /* Sa = mulţimea vârfurilor active */
   viziteaza(i0)
   S[i0] ← 1
   push(Sa, i0)
   while (not esteStivaVida(Sa)) do
      i ← top(Sa)
      if (p[i] = NULL)
         then pop(Sa)
         else j ← p[i]->elt
              p[i] ← p[i]->succ
              if S[j] = 0
                 then viziteaza(j)
                      S[j] ← 1
                      push(Sa, j)
end
```

- Explorarea BFS se obține prin reprezentarea mulțimii *Sa* printr-o coadă.

Exemplu de explorare

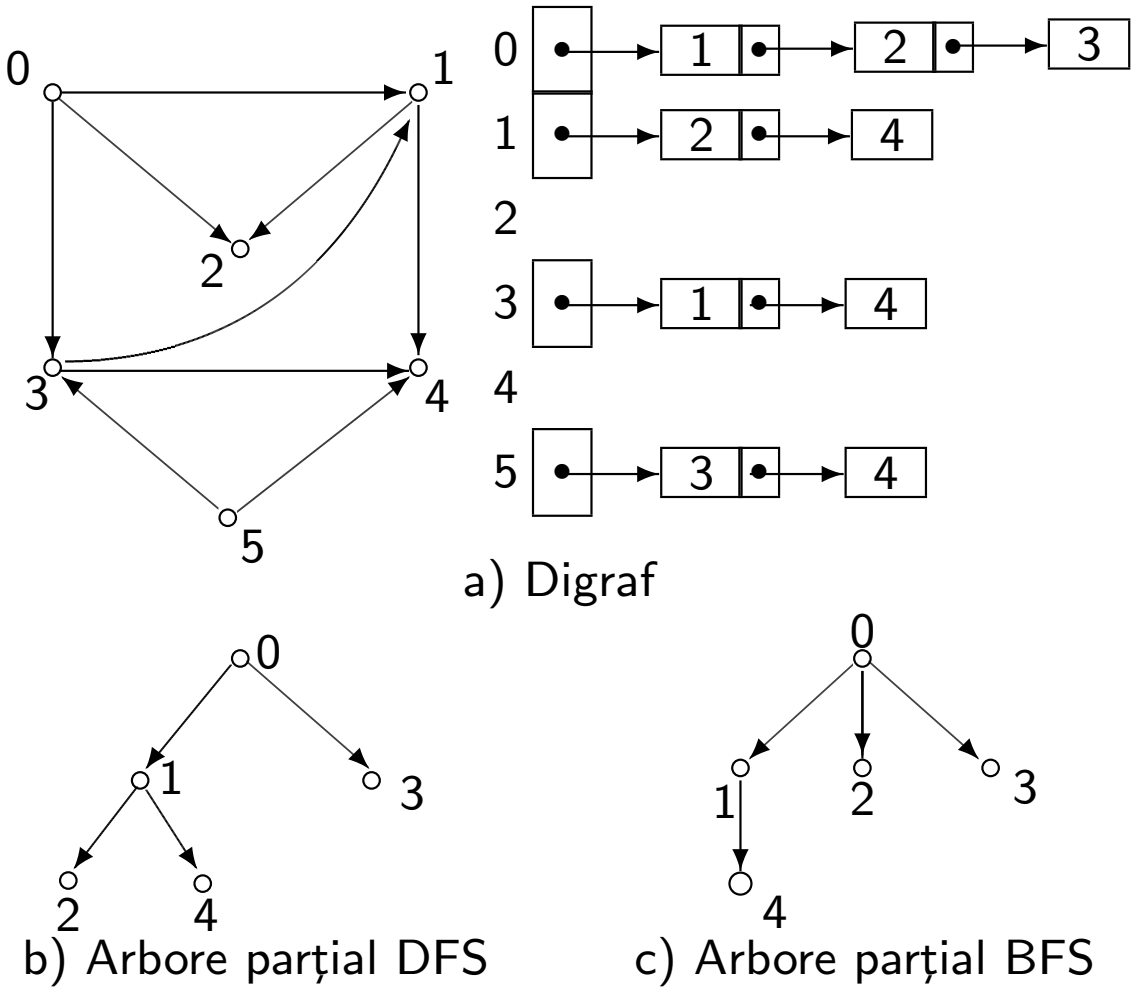


Figura 7 : Explorarea unui digraf

Mulțimi parțial ordonate finite și digrafuri aciclice

- Există o legătură strânsă între mulțimile parțial ordonate finite și digrafurile aciclice (digrafuri fără circuite numite pe scurt dag-uri).
- Orice mulțime parțial ordonată $(S, \leq)$ definește un dag $D = (S, A)$, unde există arc de la a la b, dacă $a < b$ și nu există $c \in S$ cu proprietatea $a < c < b$.
- Reciproc, orice dag $D = (V, A)$ definește o relație de ordine parțială $\leq$ peste V, dată prin: $u \leq v$, dacă există un drum de lungime ≥ 0 de la u la v.
- De fapt, $\leq$ este închiderea reflexivă și tranzitivă a lui A (se mai notează $\leq = A^*$).
- Sortarea topologică a unui dag constă într-o listă liniară a vârfurilor astfel încât dacă există arc de la u la v, atunci u precede pe v în listă, pentru oricare două vârfuri u și v.
- Vârfurile care candidează pentru primul loc în lista sortată topologic au proprietatea că nu există arce incidente spre interior (care sosesc în acel vârf) și se numesc *surse*.

Sortare topologică - metoda DFS

- Reamintim că în timpul explorării DFS, un vârf poate fi întâlnit de mai multe ori.
- Notăm cu $f[v]$ momentul când vârful v este întâlnit ultima dată (când lista de adiacență este epuizată).
- Descrierea algoritmului:
  1. Apelează algoritmul DFS pentru a determina momentele de terminare $f[v]$ pentru orice vârf v.
  2. De fiecare dată când un vârf este terminat este adăugat la începutul unei listei înlănțuite.
  3. Lista înlănțuită finală va conține o sortare topologică a vârfurilor.
- Corectitudinea algoritmului se bazează pe următorul rezultat.

Lema (1) (T. Cormen, C. Leiserson, R. Rivest, (2000), Introducere în algoritmi, Computer Libris Agora, Cluj)

Un digraf D este aciclic dacă și numai dacă explorarea DFS nu produce arce înapoi.

Sortare topologică - metoda BFS

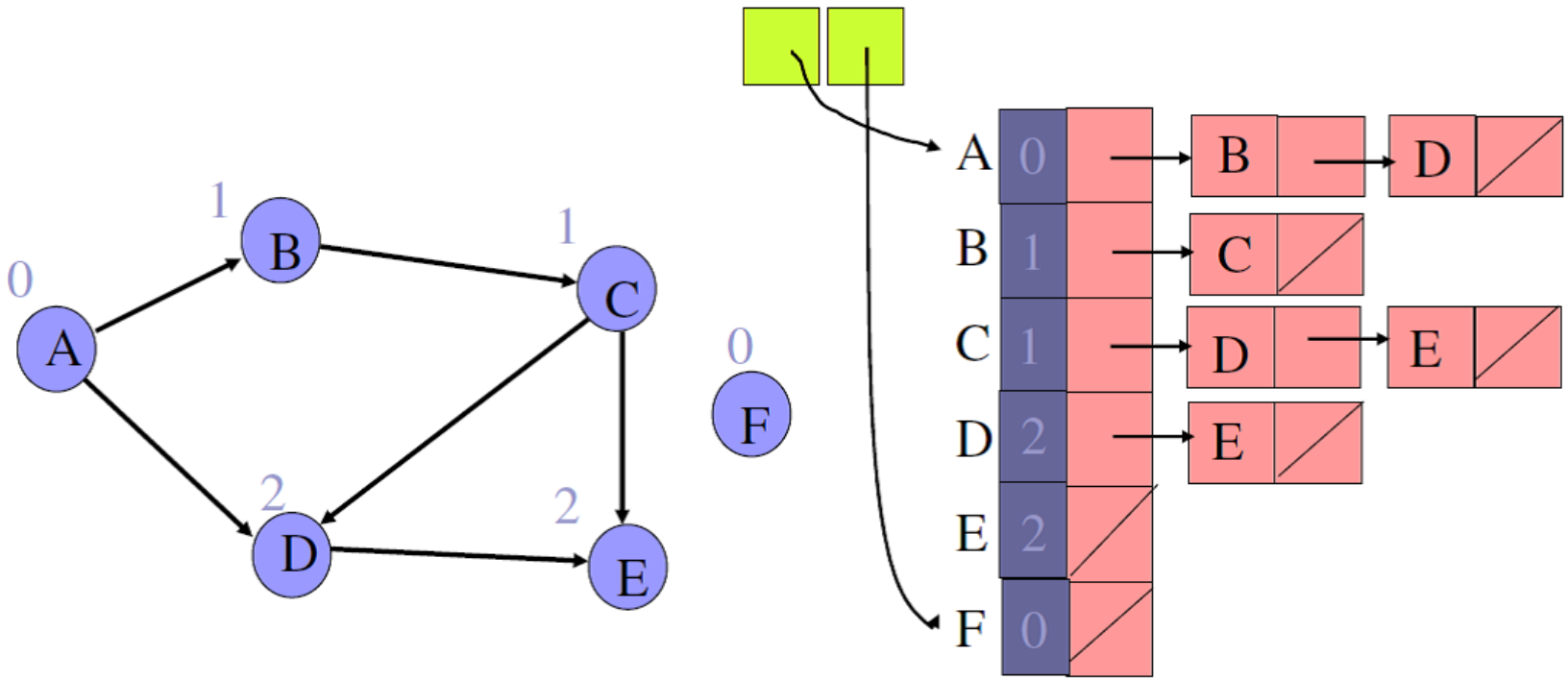
- Presupunem că pentru dag-ul D sunt create atât listele de adiacență interioară, cât și cele de adiacență exterioară.
- Listele de adiacență interioară vor fi utilizate la determinarea vârfurilor sursă (vârfuri fără predecesori); acestea au listele de adiacență interioară vide.
- Descrierea algoritmului:
  1. Inițializează coada cu vârfurile sursă.
  2. Extrage un vârf u din coadă pe care-l adaugă la lista sortată parțial.
  3. Elimină din reprezentarea (acum parțială) a lui D vârful u și toate arcele (u, v).
  4. Dacă pentru un astfel de arc lista de adiacență interioară a vârfului v devine vidă, atunci v va fi adăugat la coadă.
  5. Repetă pașii 2-4 până când coada devine vidă.

Sortare topologică - algoritm BFS

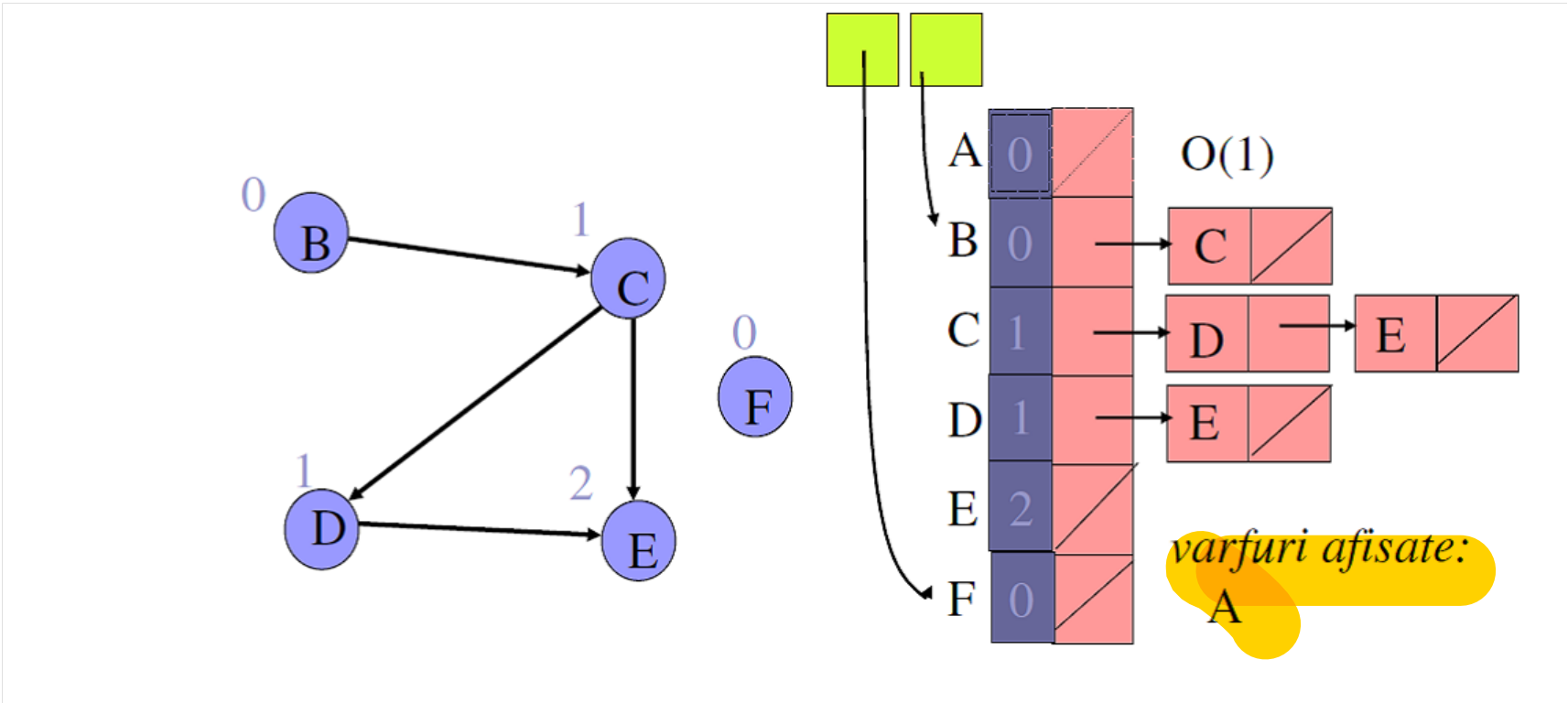
- Extindem structura `D`, care reprezintă digraful D, cu tabloul `np[1..n]`.
- `D.np[u]` conține numărul predecesorilor vârfului u.
- `L` este lista care conține varfurile digrafului D în ordine topologică.

```
procedure sortareTopologicaBFS(D,np)
   coadaVida(C) //initializeaza coada C
   // insereaza in C varfurile fara predecesori
   for u ← 0 to D.n-1 do
       if D.np[u]=0 then insereaza(C,u)
   // construieste lista varfurilor (afiseaza) in ordine topologica
   for k ← 0 to D.n-1 do
       if esteVida(C)
          then return ("Graful contine cicluri")
       u ← elimina(C)
       inserează(L, u) // inserarea se face la sfarsitul listei
       p ← D.a[u]
       while p≠NULL do
          v ← p->elt //v este un succesor imedial al lui u
          D.np[v] ← D.np[v]-1
          if D.np[v]=0
             then insereaza(C,v)
          p ← p->succ
end
```

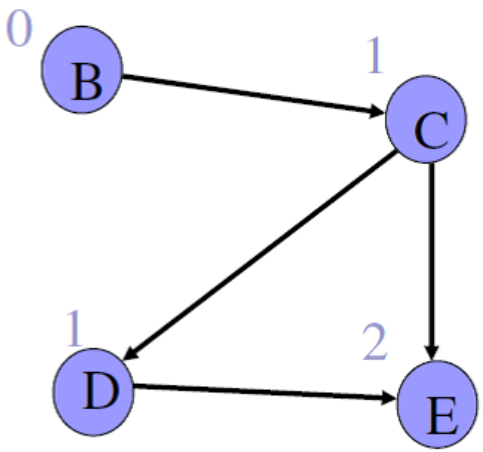
Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS

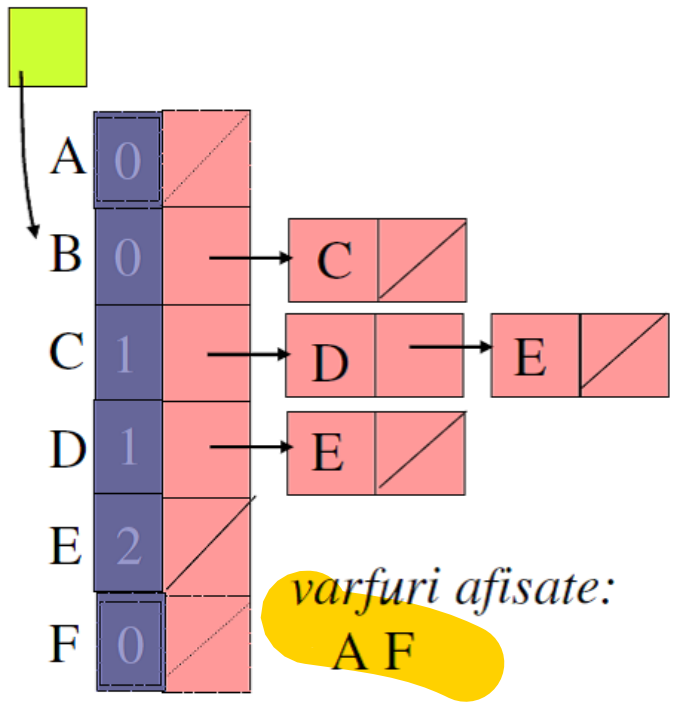


Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS

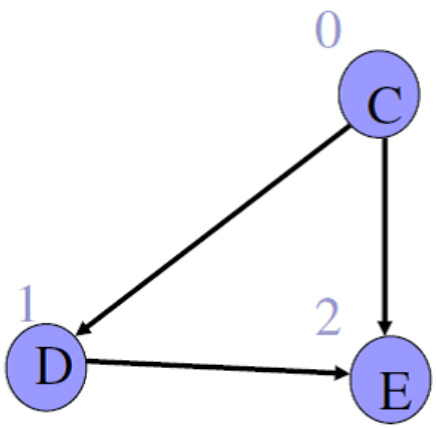


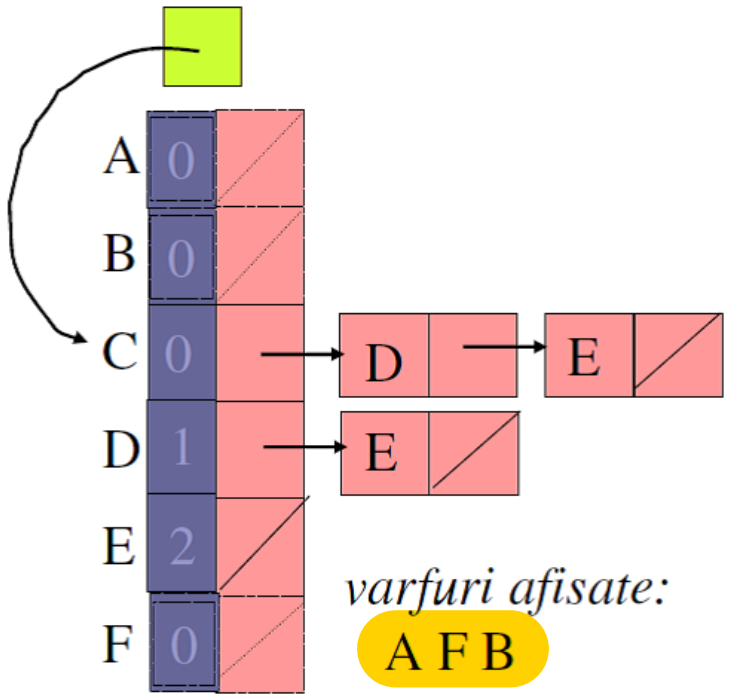
Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS



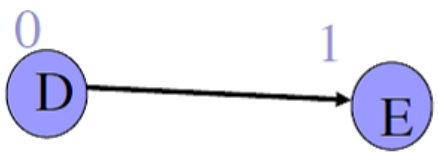


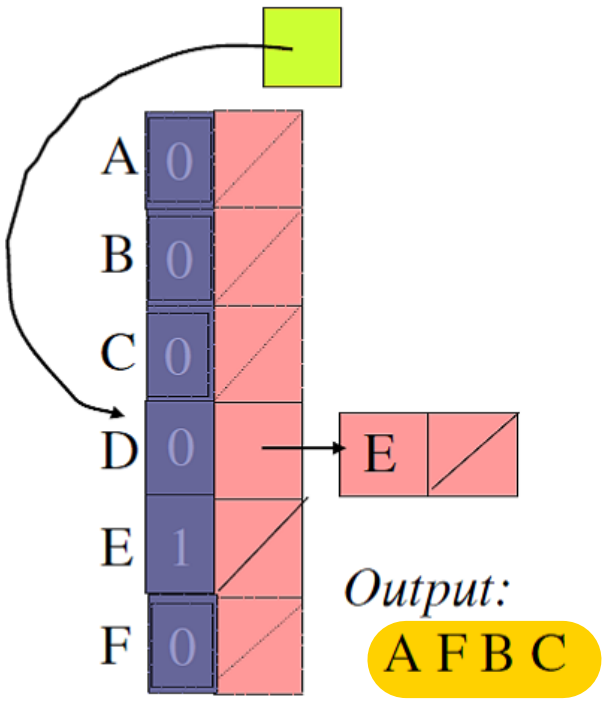
Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS



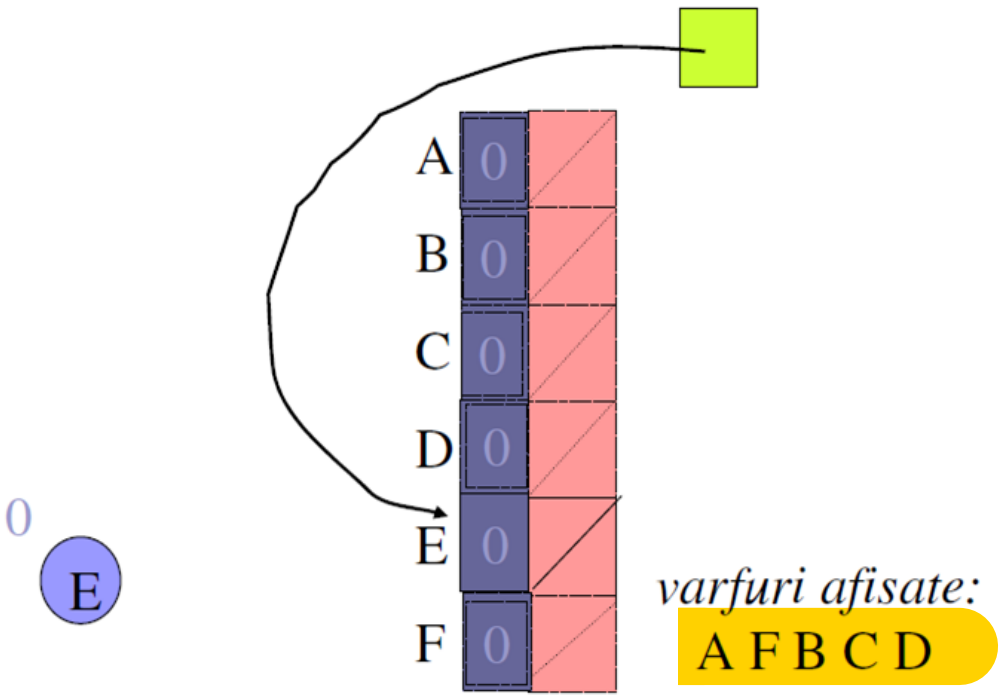


Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS



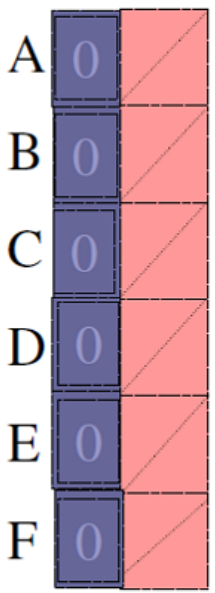


Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS



Exemplu de execuție a algoritmului de sortare toplogică prin metoda BFS

FINAL!



varfuri afisate:
A F B C D E

Complexitatea timp

- Compusă din timpul pentru:
  - identificarea vârfurilor fără predecesori: $O(n)$
  - ștergerea muchiilor: $O(m)$
  - afișarea vârfurilor: $O(n)$
- Timp total : $O(n+m)$