

Notatia asimptotica

- comportarea lui $T(n)$ cind $n \rightarrow \infty$

Formal

- $f : \mathcal{N} \rightarrow \mathcal{R}_+$ (f asimptotic pozitiva)

$$\mathcal{O}(g) := \{f \mid \exists c > 0, \exists n_0 \text{ a.i. } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$$\Omega(g) := \{f \mid \exists c > 0, \exists n_0 \text{ a.i. } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$$\Theta(g) :=$$

$$\{f \mid \exists c_1, c_2 > 0, \exists n_0 \text{ a.i. } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0\}$$

Sortarea prin insertie

Sortarea prin insertie (directa)

```
void InsDir (int A[], int n)
{ //sortare prin insertie directa a vectorului A[1..n]
for (i = 2; i <= n; i++)
{
    x = A[i];
    //se cauta locul valorii x in destinație
    j = i - 1;
    while ( (j > 0) && (x < A[j]) )
    {
        A[j + 1] = A[j];
        j--;
    }
    //inserarea lui x la locul lui
    A[j + 1] = x;
}
```

MergeSort

MergeSort

MergeSort(A, 1, n)

 if $n = 1$ sortat

 MergeSort(A, 1, n/2)

 MergeSort(A, n/2, n)

 Merge cei 2 subvectori ordonati

Merge = interclasare

- **input:** 2 vectori ordonati crescator $a[1..n]$ si $b[1..n]$
- **output:** vector $c[1..2n] = a \cup b$ ordonat crescator
- $\Theta(n)$

Structuri de Date

Elementare

- avem de reprezentat **multimi** (finite, de date omogene)
 - statice - componenta nu se schimba in timp
 - dinamice - componenta se schimba in timp
- multimi ... pe care facem diverse **operatii**
- ... in scopul **rezolvării unor probleme**

Operatii de baza

- Traversarea
 - operatia care acceseaza fiecare element al structurii, o singura data, in vederea procesarii (*vizitarea* elementului)
- Cautarea
 - se cauta un element cu cheie data in structura
 - *cu sau fara* succes
 - consta dintr-o traversare - eventual incompleta a structurii, in care vizitarea revine la comparatia cu elementul cautat
 - problema cheilor multiple - gasirea primei aparitii, a tuturor aparitiilor

Operatii de baza (cont.)

- Inserarea
 - adaugarea unui nou element structurii, cu pastrarea tipului structurii
 - Stergerea
 - extragerea unui element al structurii (eventual in vederea unei procesari), cu pastrarea tipului structurii pe elementele ramase
-
- Inserarea si Stergerea - reprezentarea multimilor cu caracter ***dinamic***
 - ***costuri mici***

Clase principale de structuri

- Lineare
- Nelineare
 - arborescente
 - grafuri

Structuri Lineare

- În alocare
 - statica - vectori
 - dinamica - liste înlăntuite
- Operatii de i/o (inserari/stergeri)
 - fara restrictii i/o
 - cu restrictii la i/o (stive si cozi)

Cautarea (unei valori date intr-o str. lineară în alocare statică)

```
void SearchLin (int A[], int n, int Val, int Loc)
{      /* caută liniar valoarea Val în A[1..n] și returnează
Loc = 0 dacă nu o găsește, și o valoare Loc ∈ [1..n] dacă
o găsește pe componenta A[Loc] */
    int i;
    Loc = 0;
    i = 1;
    while ((i <= n) && (A[i] != Val))
    {
        i++;
    }
    if (i <= n)
        Loc = i;
}
```

Complexitate (costuri) - cautare lineară

- În funcție de **componente accesate (comparații)**

p_i = probabilitatea evenimentului $Val=A[i]$ (gasim valoarea căutată pe componenta i), $i \in [1..n]$.

q = probabilitatea ca Val să nu se găsească în $A[1..n]$.

$$\text{Avem } \sum p_i + q = 1.$$

Pentru fiecare $i \in [1..n+1]$, pentru a decide că prima apariție a lui Val este pe componenta $A[i]$, facem i comparații.

Numărul mediu de comparații va fi:

$$C = \sum p_i i + q(n+1).$$

Complexitate (costuri) - căutare lineară (cont.)

Cazul căutării cu succes:

- Val se găsește precis în vector, i.e. $q=0$
- se găsește cu probabilitate egală pe oricare din componente, i.e. $p_1 = \dots = p_n = 1/n$

$$C = (1/n) (1 + 2 + \dots + n) = (n+1)/2$$

numărul mediu de comparații în cazul **căutării cu succes.**

Cazul căutării fără succes:

- se traversează toata structura, se accesează $n+1$ comp.

$$C = n+1$$

Caz particular - vector ordonat crescător

- Structură lineară în alocare statică (sequentiala)
- organizare suplimentara

$$A[1] \leq A[2] \leq \dots \leq A[n]$$

- Informatie in plus
 - permite imbunatatirea cautarii lineare

- Alta cautare - cautarea binara
 - necesita modificarea algoritmilor de inserare

Cautarea lineară într-un vector sortat

```
void SearchLinOrd (int A[], int n, int Val, int Loc)
{
    int i;
    Loc = 0;
    i = 1;
    while ((i <= n) && (A[i] < Val))
        i++;
    if (i <= n)
        if (A[i] == Val) // căutare cu succes
        {
            Loc = i;
            printf("element gasit pe pozitia %d ", Loc);
        }
        else // A[i] > Val
            printf("căutare fără succes");
    else
        printf("căutare fără succes");
}
```

Căutarea binara (intr-un vector sortat)

$A[1..n]$ un vector cu $A[1] \leq A[2] \leq \dots \leq A[n]$

Algoritmul de căutare binară:

- (1) Se începe cu segmentul definit de indicii $Left = 1$ și $Right = n$
- (2) Pentru fiecare subvector $A[Left..Right]$ se repetă:
 - (a) Se calculează mijlocul segmentului
$$Mid := (Left + Right) / 2$$
 - (b) Se compară Val cu $A[Mid]$:
 - dacă $Val = A[Mid]$ căutarea se termină cu succes;
 - dacă $Val < A[Mid]$ se reia pasul (2) pe $[Left..Mid-1]$;
 - dacă $Val > A[Mid]$ se reia pasul (2) pe $[Mid+1..Right]$.

```
void SearchBin(int A[], int n, int Val, int Loc)
{ int Left, Right, Mid;
    Left = 1; Right = n;
    Mid = (Left + Right)/2;
    Loc = 0;
    while ((Left <= Right) && (Val != A[Mid] ))
    {
        if (Val < A[Mid]) // se continuă pe subintervalul din
stânga
            Right = Mid-1;
        else // Val > A[Mid] - se continuă pe subintervalul din
dreapta
            Left = Mid+1;
        Mid = (Left + Right)/2;
    }
    if (A[Mid] == Val) //căutare cu succes
    { Loc = Mid;
        Printf("element gasit pe pozitia %d ", Loc);}
    else
        Loc = 0; //căutare fără succes
}
```

Căutarea binara - complexitate

$C(n)$ = numărul de comparații pe care îl necesită căutarea binară pe un vector cu n componente.

După fiecare comparație dimensiunea segmentului pe care căutăm se reduce la jumătate.

Dacă după $C(n)$ comparații am încheiat căutarea, atunci

$$2^{C(n)} > n > 2^{C(n)-1}$$

de unde

$$C(n) = \lfloor \log_2 n \rfloor + 1$$

complexitatea căutării binare - $O(\log_2 n)$

complexitatea căutării lineare (secventiale) - $O(n)$

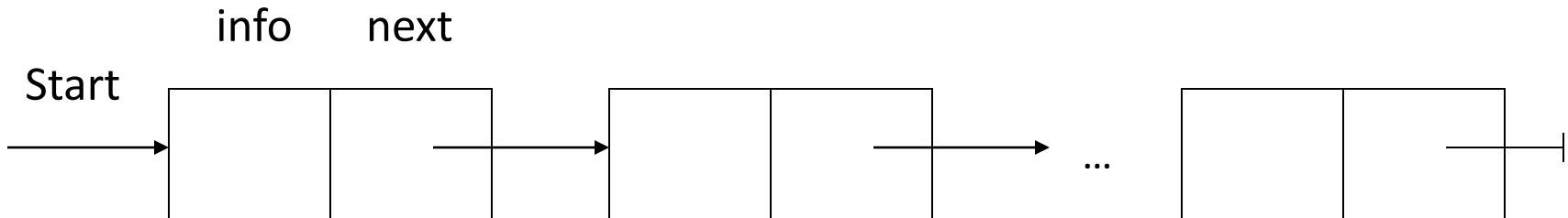
Structuri lineare in alocare dinamica: liste (liste simplu înlanțuite)

- elementele listei s.n. noduri
- fiecare nod conține:
 - (1) un câmp, pe care se reprezintă un element al mulțimii; (de obicei vom identifica elementul cu valoarea de pe un singur câmp, numit câmp cheie;) în algoritmii care urmează putem presupune că elementul ocupă un singur câmp, *info*;
 - (2) un pointer către nodul următor, *next*.

Liste simplu inlantuite

```
type  pnod = ^nod;
      nod = record
          info: integer;
          next: pnod
      end
```

```
typedef struct nlsi{
    int info;
    struct nlsi *next;
} Inod;
```



Alte tipuri de liste. Aplicatii.

- cu nod marcaj
- circulare
- dublu inlantuite
- alte inlantuiri
 - liste de liste
 - masive

Liste cu nod marcaj

O listă, *Start*, cu nod marcaj, va conține în variabila *Start* adresa acestui nod, iar lista efectivă, în care în fiecare nod avem reprezentat un element al mulțimii de date, va fi $\text{Start} \uparrow .next$.

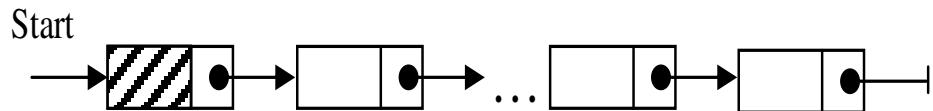


Fig. 2.1.1. Listă cu nod marcaj.

O listă cu nod marcaj vidă va conține doar nodul marcaj.



Fig.2.1.2. Listă vidă cu nod marcaj.

-- Se modifica (simplifica) inserarile/stergerile

Liste circulare

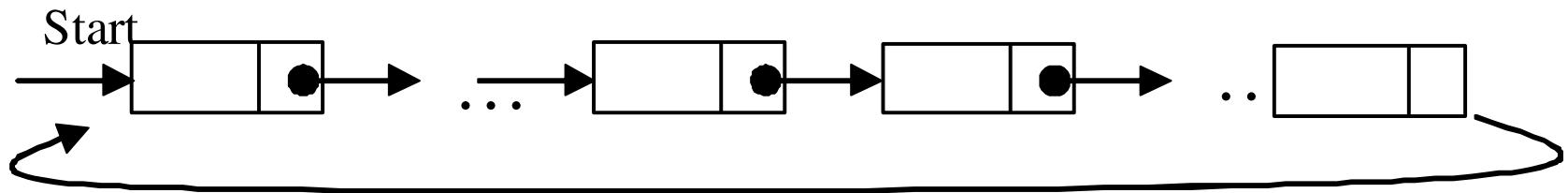


Fig.2.1.3. Listă circulară.

-- utilă pentru aplicațiile în care este nevoie să facem parcurgeri repetitive ale listei

-- testul de nedepășire al structurii nu va mai fi de tipul $p \neq nil$

Liste circulare cu nod marcaj

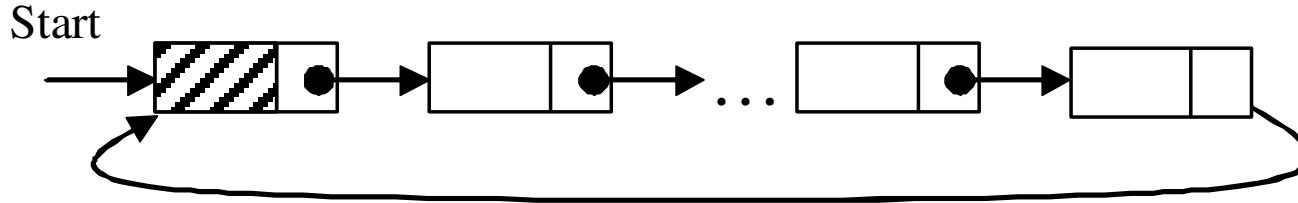


Fig.2.1.4. Listă circulară cu nod marcaj.

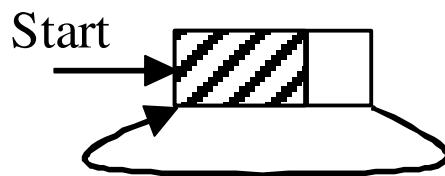


Fig.2.1.5. Listă circulară vidă cu nod marcaj.

-- cautare: Se introduce valoarea căutată Val pe câmpul $info$ al nodului marcaj cu $Start \uparrow.info := Val$. Se începe căutarea în lista $Start \uparrow.next$.

Loc = pointerul returnat de operația de căutare. Dacă $Loc \neq Start$ căutarea este cu succes, iar dacă $Loc = Start$ căutarea este fără succes.

Liste dublu înlănțuite.



Fig.2.1.6. Nod într-o listă dublu înlănțuită.

- inserari/stergeri: parcurgerea cu cautarea locului se poate face cu un singur pointer
- parcurgeri în ambele sensuri
- **cost**: locatii in plus !

Structuri lineare cu restrictii la i/o: Stive si Cozi

Stiva -- Utilizator

- LIFO (Last In First Out): ultimul introdus este primul extras
- $\text{Push}(\text{Stack}, \text{Val})$ - inserarea in timp cst.
- $\text{Pop}(\text{Stack}, X)$ - stergerea/extragerea in timp cst.

Stiva -- Implementare

- LIFO (Last In First Out): ultimul introdus este primul extras
- locul unic pt. ins./stergeri: virf, baza... (*Top*)
- *Push(Stack, Val)* - inserarea valorii *Val* in stiva *Stack*
 - **Overflow (supradepasire)** - inserare in stiva plina
- *Pop(Stack, X)* - stergerea/extragerea din stiva *Stack* a unei valori care se depune in *X*
 - **Underflow (subdepasire)** - extragere din stiva goala

Coada -- Utilizator

- FIFO (First In First Out): primul introdus este primul extras
- $\text{Insert}(\text{Queue}, \text{Val})$ - inserarea in timp cst.
- $\text{Delete}(\text{Queue}, X)$ - stergerea/extragerea in timp cst.

Coada -- Implementare

- FIFO (First In First Out): primul introdus este primul extras
- capat pt. inserari: sfirsit, spate ... (*Rear*)
- capat pt. stergeri: inceput, fata ... (*Front*)
- *Insert(Queue, Front, Rear, Val)* - inserarea
 - **Overflow (supradepasire)** - inserare in coada plina
- *Delete(Queue, Front, Rear, X)* - stergerea/extragerea
 - **Underflow (subdepasire)** - extragere din coada goala

Arbore - def. recursiva

O structură de arbore (**k -arbore**), T , de un anume tip de bază este

- (a) fie o structură vidă (adică $T = \emptyset$);
- (b) fie este nevidă, deci conține un nod de tipul de bază, pe care-l vom numi rădăcină și îl vom nota $\text{root}(T)$, plus un număr finit de structuri disjuncte de arbori de același tip, T_1, T_2, \dots, T_k , numiți subarborii lui T (sau fiii lui $\text{root}(T)$).

Reprezentarea ca graf a unui **k -arbore**:

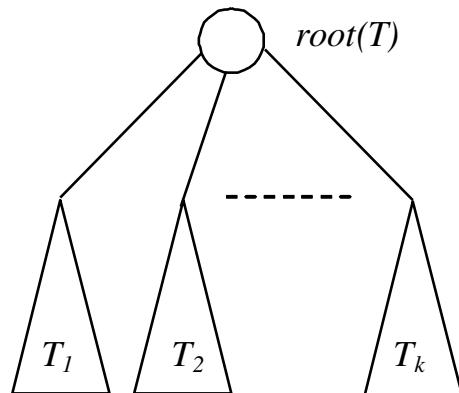


Fig.3.1.1. Un k -arbore T , cu nodul rădăcină $\text{root}(T)$ și fii săi, T_1, T_2, \dots, T_k .

Arbore - liberi

Definiție. Se numește **graf** $G = (X, V)$ o pereche formată din două mulțimi, mulțimea X a nodurilor sau vîrfurilor grafului, și mulțimea V a muchiilor grafului, unde o muchie $v \in V$ este o pereche ordonată de noduri $v=(x,y)$, $x,y \in X$.

Un graf neorientat este un graf în care perechea (x,y) se identifică cu perechea (y,x) .

Un graf fără cicluri este un graf în care, pornind de la un vîrf dat nu putem ajunge din nou la el folosind muchii.

Definiție. Se numește **arbore** un graf $H = (X, V)$ care este neorientat, conex, fără cicluri, cu un nod precizat numit rădăcină. Pentru orice vârf $x \in X$, există un număr finit de vârfuri $x_1, \dots, x_n \in X$ asociate lui x , numite descendenți direcți (sau filii) lui x .

Arbore liberi - mai multe caracterizari

Arbore - terminologie

arborii ordonați, arbori în care există o relație de ordine între descendenții unui nod

arborii neordonați, arbori în care nu există o relație de ordine între descendenții unui nod

Arbore - terminologie

gradul arborelui = întregul k care reprezintă numărul maxim de fii ai unui nod.

Fiecare nod al arborelui îi vom asocia un *nivel* în felul următor:

- (a) rădăcina se află la nivelul 0,
- (b) dacă un nod se află la nivelul i atunci fiii săi sunt la nivelul $i+1$.

Numim *înălțime* (sau *adâncime*) a unui arbore nivelul maxim al nodurilor sale.

Se numește *terminal sau frunză* un nod fără descendenți.

Se numește *nod interior* orice nod care nu e terminal.

Un 2-arbore ordonat se numește *arbore binar*. (fiu stâng, respectiv fiu drept.)

Arbore - aplicatii

- Cautare – cautare, inserare, stergere
- Implementari de cozi cu prioritati – ins si del
- Sortare
- *Reprezentari*
 - Proceduri de decizie – de estimat costuri
 - ‘Expresii’ – de ‘evaluat’ -parcurgeri
 - Codificare – (de construit arb cu propr ‘bune’)
- *Reprezentari de multimi*
 - Union-Find problem (pb. reuniunii si apartenentei) – ex de combinare (merge)

Arbore (oarecari) - traversari

Se presupune data o reprezentare în care fiecare nod are trei câmpuri:

info: pentru informația din noduri

nrfii: valoarea NF , întreagă, reprezintă numărul de fii ai nodului

fii: vector de dimensiune NF , cu componente pointeri, astfel încât, $fii[J]$ este pointer către fiul J al nodului, iar $J = 1, 2, \dots, NF$.

Algoritmul de **traversare** este următorul:

1. Se pornește de la rădăcină.
2. La fiecare nod curent:
 - (a) se procesează *info*
 - (b) se introduc într-o structură ajutătoare fiii nodului curent în vederea procesării ulterioare.
3. Se extrage din structura ajutătoare un alt nod și se reia de la punctul 2.

Ca **structură ajutătoare** putem folosi una dintre structurile liniare pe care le cunoaștem, **stiva** sau **coada**.

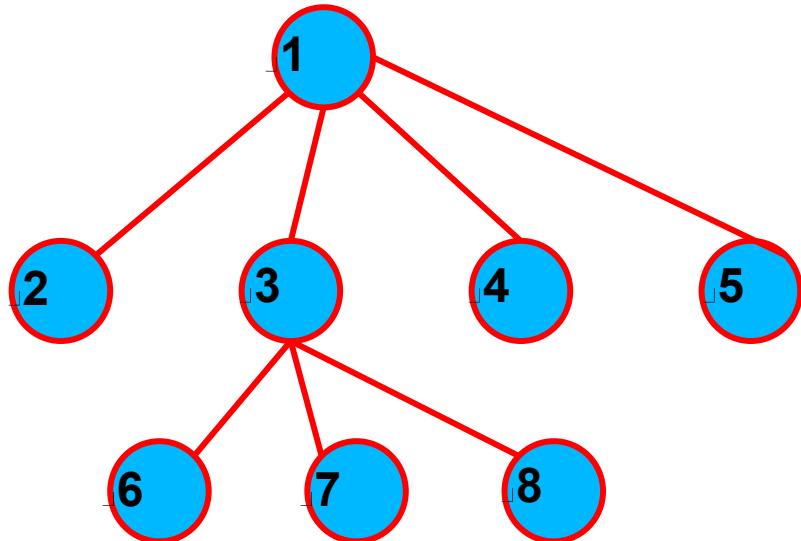
Dacă folosim o coadă obținem *traversarea în lățime* (*breadth first*) a arborelui.

Dacă folosim o stivă obținem *traversarea în adâncime* (*depth first*) a arborelui.

Arbore (oarecari) - traversare

In adancime

```
void DF (nod *p)
{
    int i;
    if (p != NULL)
    {
        printf("%d ", p → info);
        for (i=0; i<p → NF; i++)
            DF (p → fii[i]);
    }
}
```



DF (Adancime):
1, 2, 3, 6, 7, 8, 4, 5

Arbore (oarecari) - traversare

In latime (pe nivele)

BF (Latime):
1, 2, 3, 4, 5, 6, 7, 8

```
void BF (nod *rad)
{
    nod *p; int i, q = -1;
    prim = ultim = 0;
    Adauga(rad); // in coada se adauga radacina
    do
    {
        p = Extrage_nod(); // extrag un nod din coada
        if (p!=NULL)
        {
            printf("%d ", p → info);
            for (i=0; i<p → NF; i++)
                Adauga(p → fii[i]); // adaug
        }
        in coada fii nodului
    }
}while(p);
}
```

Arbore binari

Un arbore binar (2-arbore ordonat) T este:

- (1) fie un arbore vid ($T = \emptyset$).
- (2) fie e nevid, și atunci conține un nod numit rădăcină, împreună cu doi subarbore binari disjuncți numiți subarborele stâng, respectiv subarborele drept.

Arborei binari

Exemplu - reprezentari expresii aritmetice

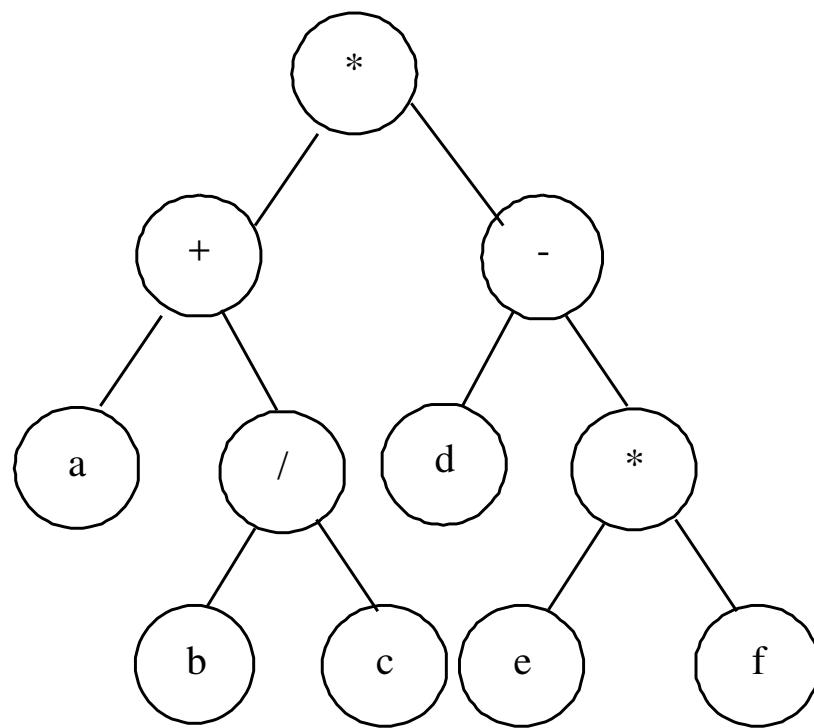


Fig.3.2.1. Arborele binar ce reprezintă expresia aritmetică $(a + (b / c)) * (d - (e * f))$.

Arbore binari - traversari in adancime

- În Preordine (RSD) (*Rădăcină Stânga Dreapta*)
- În Inordine (SRD) (*Stânga Rădăcină Dreapta*)
- În Postordine (SDR)(*Stânga Dreapta Rădăcină*)

Arbore de expresie -- preordine -- notatia prefix

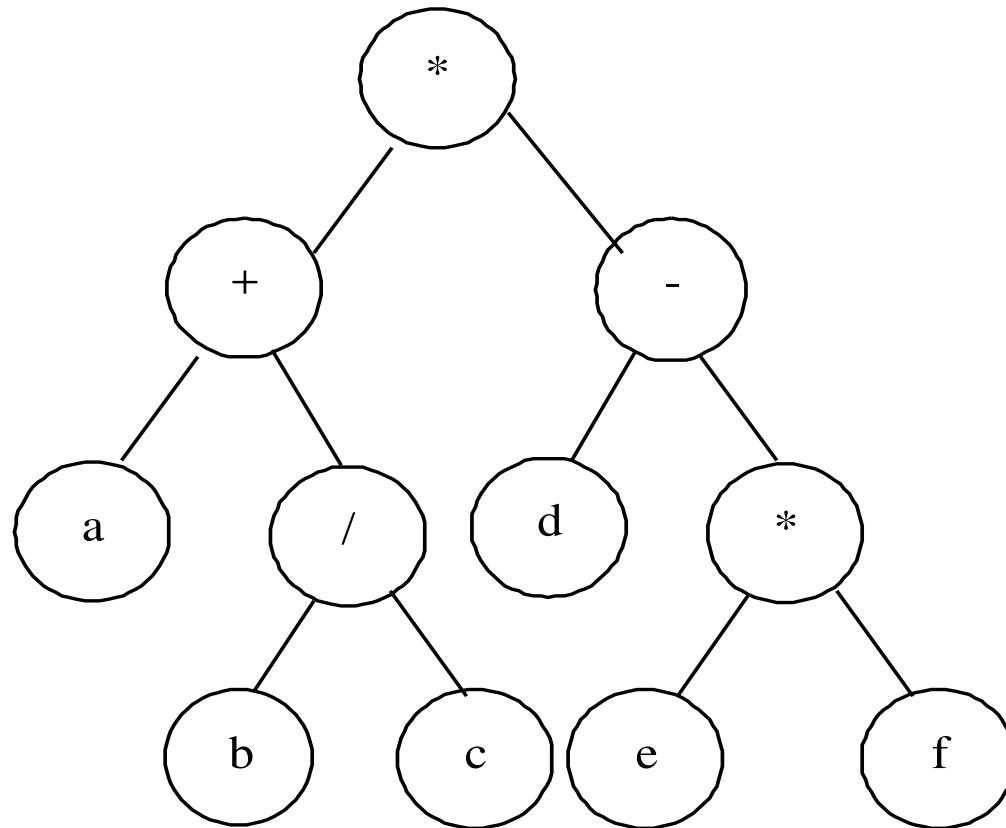


Fig.3.2.1. Arborele binar ce reprezintă expresia aritmetică $(a + (b / c)) * (d - (e * f))$.

* + a / b c - d * e f

Arbore de expresie -- inordine -- notatia infix

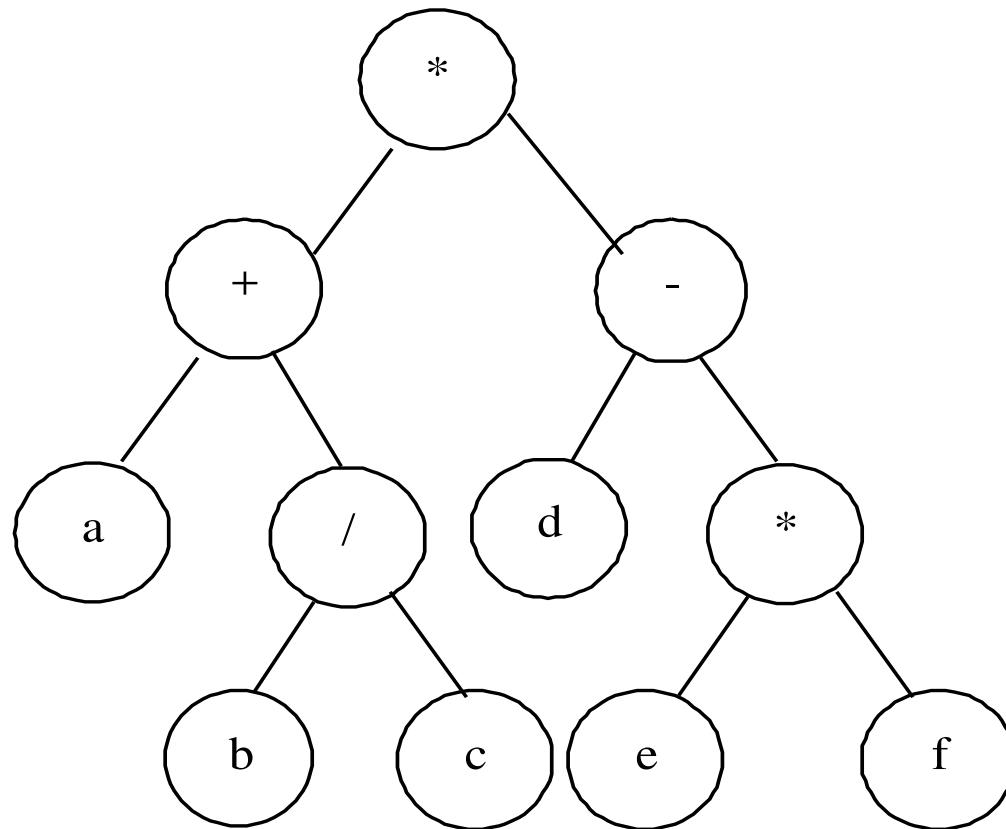


Fig.3.2.1. Arborele binar ce reprezintă expresia aritmetică $(a + (b / c)) * (d - (e * f))$.

a + b / c * d - e * f ... (a + (b / c)) * (d - (e * f))

Arbore de expresie -- postordine -- notatia postfix

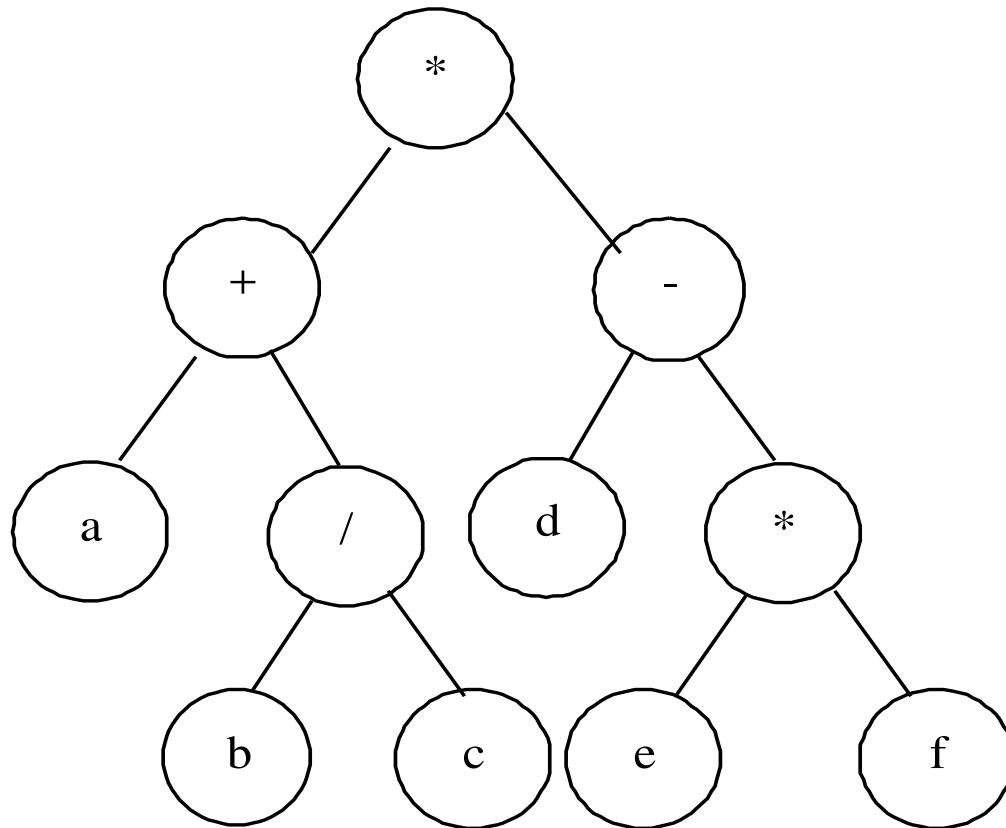


Fig.3.2.1. Arborele binar ce reprezintă expresia aritmetică $(a + (b / c)) * (d - (e * f))$.

a b c / + d e f * - *

Arbore binare de căutare

Arbore binari de cautare

- **cautare**
- **inserare**
- **stergere**
- analog al listei ordonate (performanta operatiei de cautare)

Arbore binari de cautare - definitii

Structura

- Arbore binar

- cu chei de un tip total ordonat

- pentru orice nod u al său avem relațiile:

```
struct arbore
    {int info;
     arbore *left;
     arbore *right;
    };
```

arbore *Root;

(1) $info[u] > info[v]$, pentru orice v in $left[u]$

(2) $info[u] < info[w]$, pentru orice w in $right[u]$.

Arbore binari de cautare – def recursiva

Un arbore binar de cautare T este:

(1) fie un arbore vid ($T = \emptyset$).

(2) fie e nevid,
și atunci conține un nod numit rădăcină, cu *info* de un tip totul ordonat

împreună cu doi subarbore binari de cautare disjuncti (numiți subarborele stâng, *left*, respectiv subarborele drept, *right*)
si astfel incit

(1') $\text{info}[\text{root}(T)] > \text{info}[\text{root}(\text{left}[T])]$

(2') $\text{info}[\text{root}(T)] < \text{info}[\text{root}(\text{right}[T])]$

Arbore binari de căutare – Chei multiple

- 1) **Contorizare aparitii**
- 2) **Reprezentare efectiva** a cheilor multiple:

Numim arbore binar de căutare ***nestrict la stânga*** un arbore binar T cu proprietatea că în fiecare nod u al său avem relațiile:

- (3) $\text{info}[u] \geq \text{info}[v]$, pentru orice v in $\text{left}[u]$
- (4) $\text{info}[u] < \text{info}[w]$, pentru orice w in $\text{right}[u]$.

Analog, arbore binar de căutare ***nestrict la dreapta***, cu relațiile:

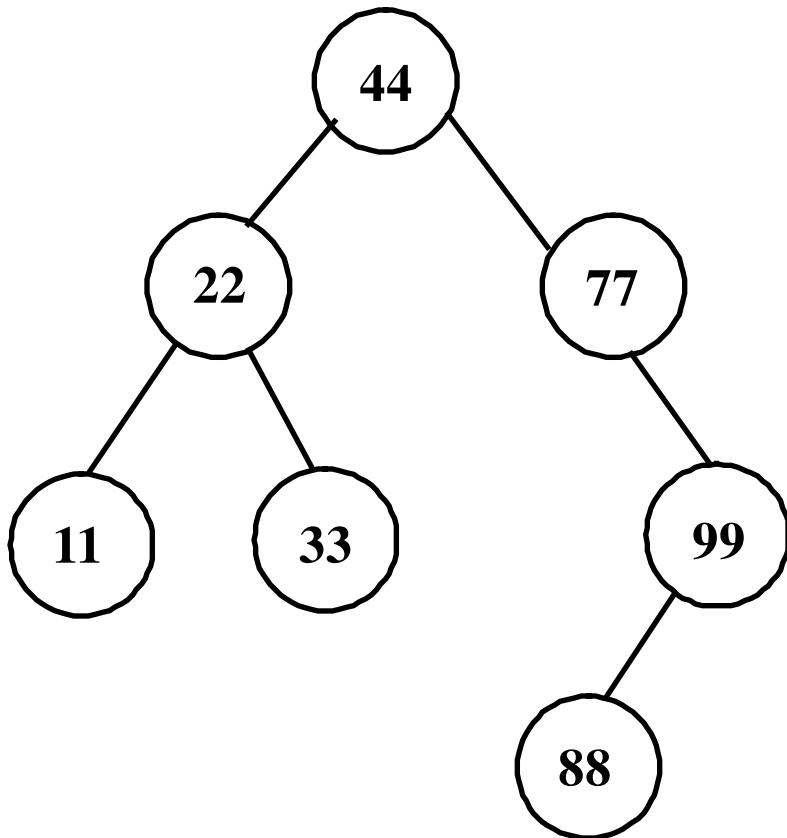
- (5) $\text{info}[u] > \text{info}[v]$, pentru orice v in $\text{left}[u]$
- (6) $\text{info}[u] \leq \text{info}[w]$, pentru orice w in $\text{right}[u]$.

Legatura cu sortarea

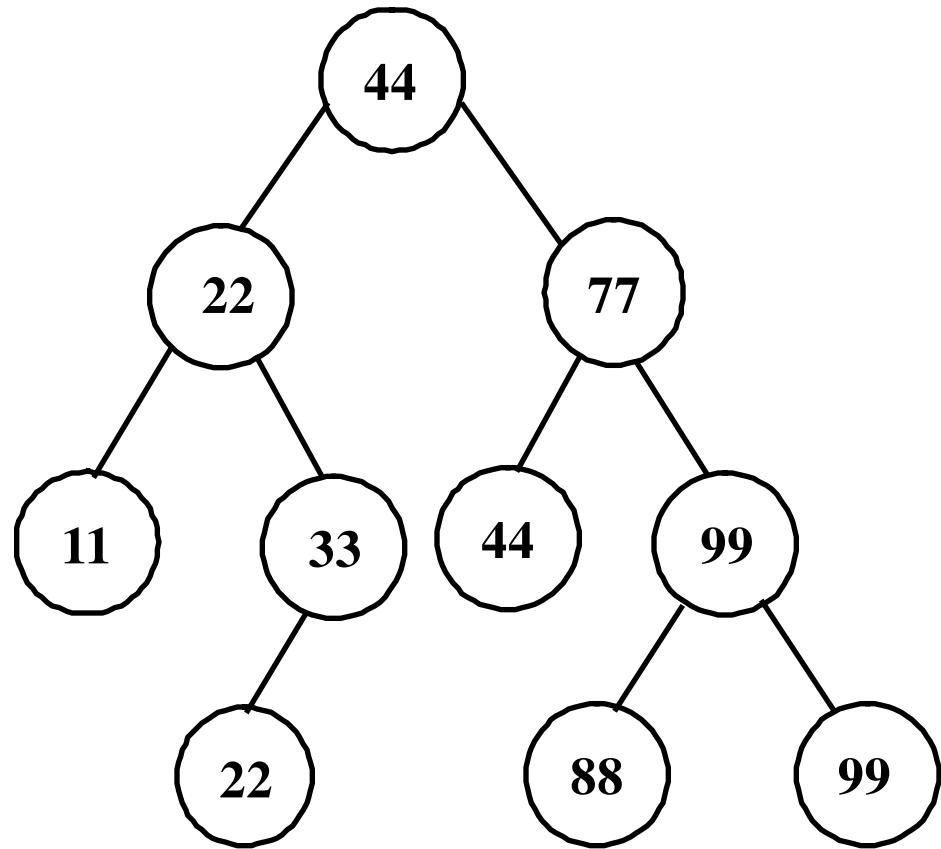
parcurserea în inordine (SRD) a unui abc produce o listă ordonată crescător a cheilor.

De aceea structura de arbore binar de căutare este potrivită pentru seturi de date pe care, pe lângă inserări și ștergeri, se cere destul de frecvent ordonarea totală a cheilor.

Arbore binar de căutare -exemple



(a) Arbore binar de căutare strict.



(b) Arbore binar de căutare nestrict la dreapta. Cheile 22, 44 și 99 sunt chei multiple.

Stergere nod din a.b.c.

(Avem ptr p pe nodul de sters, si tp pe tatal sau)

Caz 1) Nodul de sters are cel mult 1 fiu nevid

tp = unicul fiu nevid

Caz 2) Nodul de sters are ambii fii nevizi

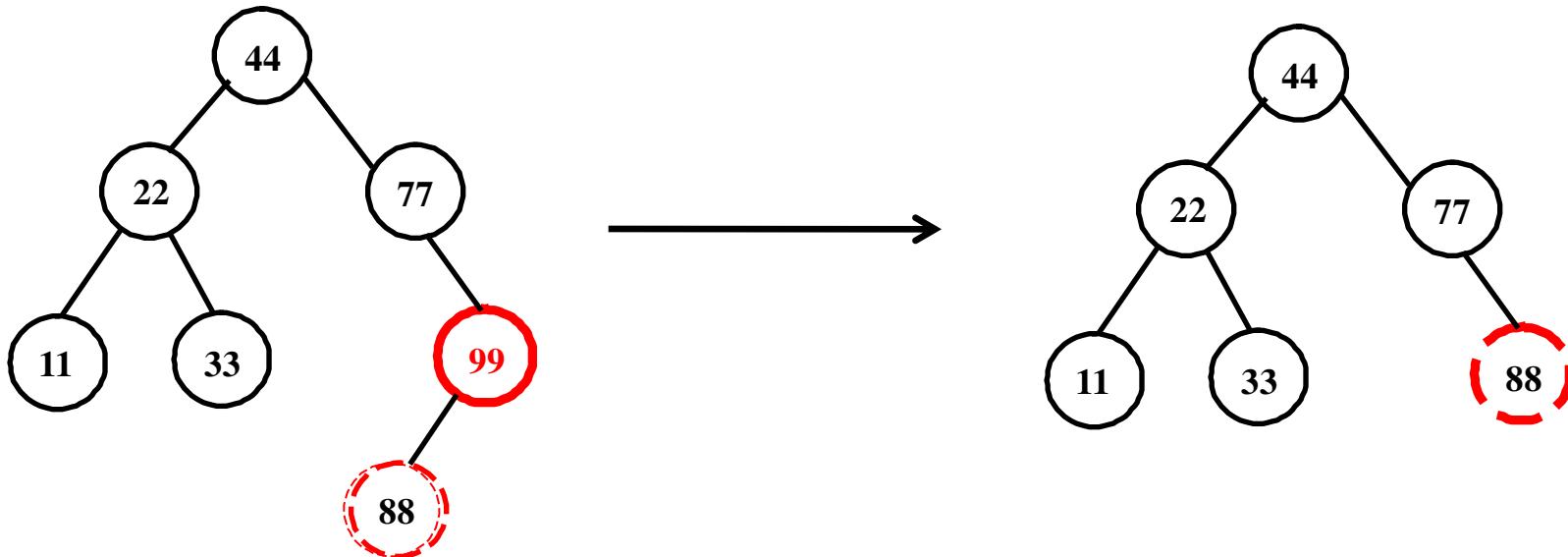
eliminam doar *valoarea info(p)*

cautam o alta valoare in arbore, care
sa poata *urca* in acest nod (*separator*)
sa fie intr-un nod usor de sters (caz 1)

Stergere nod din a.b.c.

Caz 1) Nodul de sters are cel mult 1 fiu nevid

Exemplu: - nodul de sters este 99

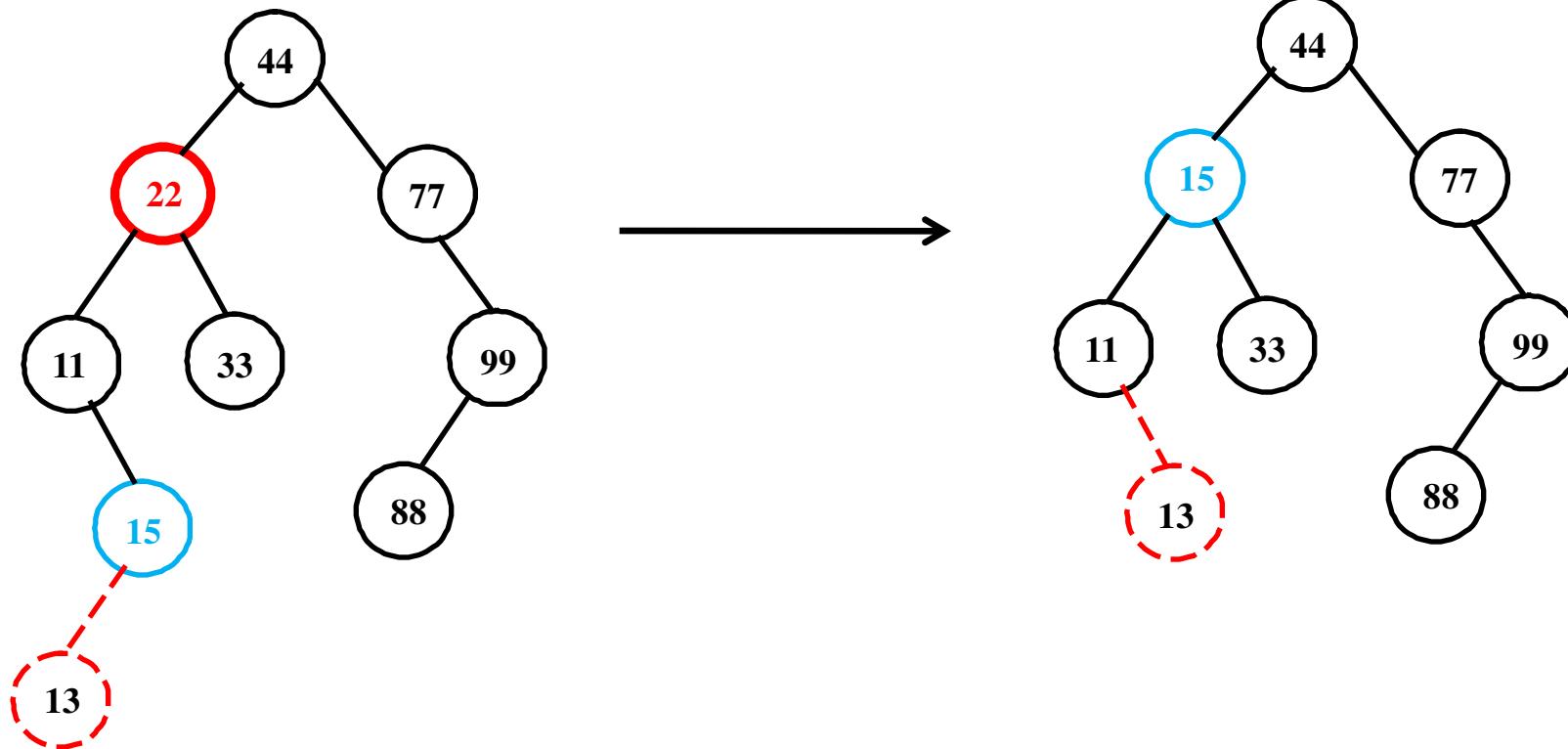


Subarborele stang (nevid) al nodului cu cheia 99 devine subarbore drept pentru nodul cu cheia 77.

Stergere nod din a.b.c.

Caz 2) Nodul de sters are ambii fii nevizi

Exemplu: - nodul de sters este 22



- Se determina predecesorul in inordine (15)
- Valoarea cheii se copiaza in locatia nodului de sters
- Se sterge nodul cu cheia 15

Complexitatea operațiilor la arborele binar de căutare.

Operațiile de inserare și ștergere de noduri într-un arbore binar de căutare depind în mod esențial de operația de căutare.

Căutarea revine la **parcurgerea, eventual incompletă, a unei ramuri**, de la rădăcină până la un nod interior în cazul căutării cu succes, sau până la primul fiu vid întâlnit în cazul căutării fără succes (și al inserării).

Performanța căutării depinde de **lungimea ramurilor pe care se caută**;

- lungimea medie a ramurilor,

- lungimea maximă = adâncimea arborelui.

Forma arborelui, deci și **adâncimea** depind, cu algoritmii dați, **de ordinea introducerii cheilor**

- **cazul cel mai nefavorabil**, în care adâncimea arborelui este n , numărul nodurilor din arbore, adică performanța căutării rezultă $O(n)$.

- Avem (vom demonstra) **o limită inferioară pentru adâncime** de ordinul lui $\log_2 n$, ceea ce însemenă că performanța operației de căutare nu poate coborâ sub ea.

Ne punem problema dacă putem atinge această valoare optimă.

Arbore binari de cautare echilibrati AVL

Procedeul clasic de construire a arborelui binar de căutare ne dă un arbore a cărui formă depinde foarte mult de **ordinea în care sunt furnizate valorile nodurilor**. În cazul cel mai general nu obținem un arbore de înălțime minimă.

Cazul **cel mai favorabil**, în care obținem înălțime minimă, este cel în care ni se furnizează pe rând mijloacele intervalelor (subintervalelor) vectorului sortat.

Cazul cel mai nefavorabil

-valorile vin în ordine crescătoare (sau descrescătoare),
- orice ‘secventa de cautare’
caz în care arborele binar de căutare obținut este degenerat (**listă înlănțuită**)

Problemă: cum **modificăm algoritmul de construcție astfel încât să obținem înălțime minimă** pentru arbore, pentru a îmbunătăți timpul de căutare?

Să observăm că la inserarea unui nou element crește cu 1 înălțimea subarborelui în care s-a făcut inserția. Ne propunem, pentru noua metodă de construcție, următorul criteriu: diferența dintre înălțimile fiului stâng și cel drept să nu depășească pe 1.

Arbore binari de cautare echilibrati AVL definitie

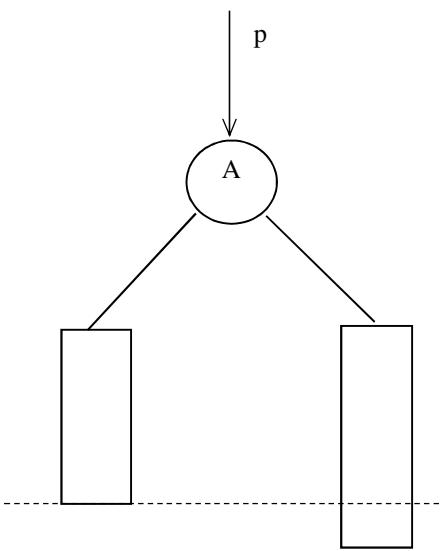
Definiție Se numește *arbore binar de căutare echilibrat AVL* (Adelson-Velskii-Landis) un arbore care în fiecare nod are proprietatea că înălțimile subarborilor stâng și drept diferă cu cel mult 1.

Pentru un nod dat, fie h_l și h_r înălțimile subarborelui stâng, respectiv drept. Avem trei situații posibile în acest nod, codificate cu valorile variabilei $bal = h_r - h_l$, pe care o numim *factor de echilibru*, în felul următor:

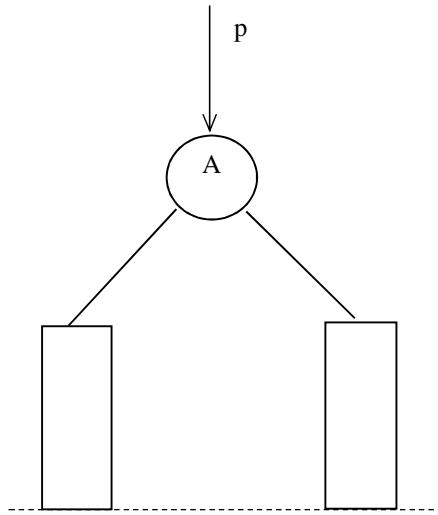
$$bal = \begin{cases} 1, & h_l = h_r - 1 \\ 0, & h_l = h_r \\ -1, & h_l = h_r + 1 \end{cases}$$

Informația despre valoarea factorului de echilibru în fiecare nod p^\uparrow al unui arbore o vom scrie într-un nou câmp al lui p^\uparrow , câmpul bal : -1..1.

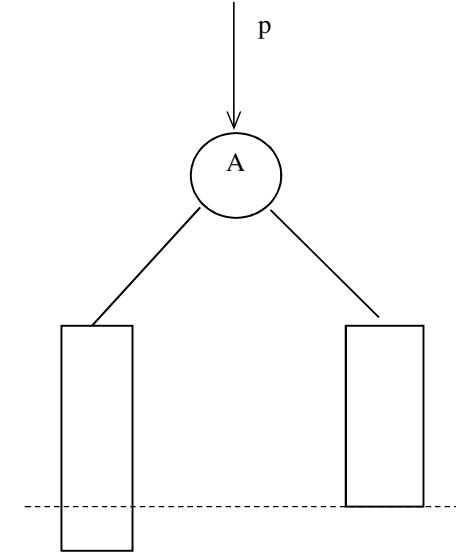
Algoritm de inserare - cu operatii suplimentare, *re-echilibrari*



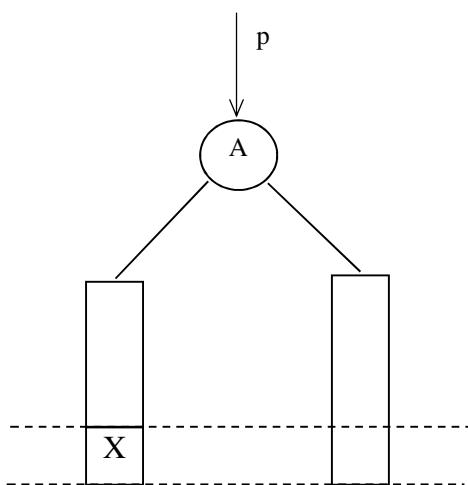
(a) $p \rightarrow \text{bal} = 1$



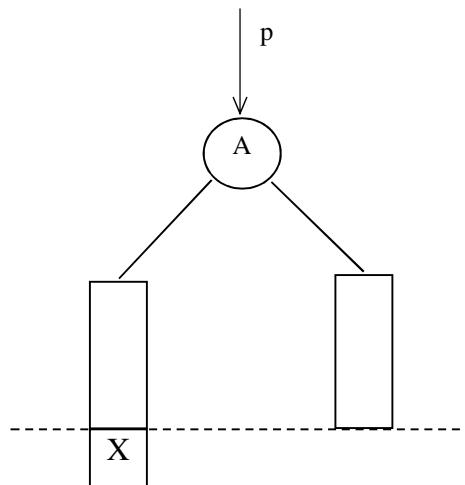
(b) $p \rightarrow \text{bal} = 0$



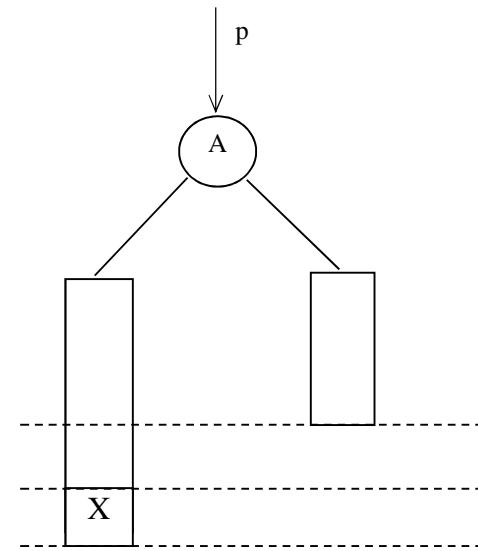
(c) $p \rightarrow \text{bal} = -1$



noul $p \rightarrow \text{bal} = 0$
(a')



noul $p \rightarrow \text{bal} = -1$
(b')



(c')

Arbore binari stricti

Proprietati si Aplicatii

Un *arbore binar strict* este un arbore binar în care fiecare nod are fie nici un fiu, fie exact doi fii.

Nodurile cu doi copii se vor numi *noduri interne*, iar cele fără copii se vor numi *noduri externe* sau *frunze*. Ele pot fi de alt tip decât nodurile interne.

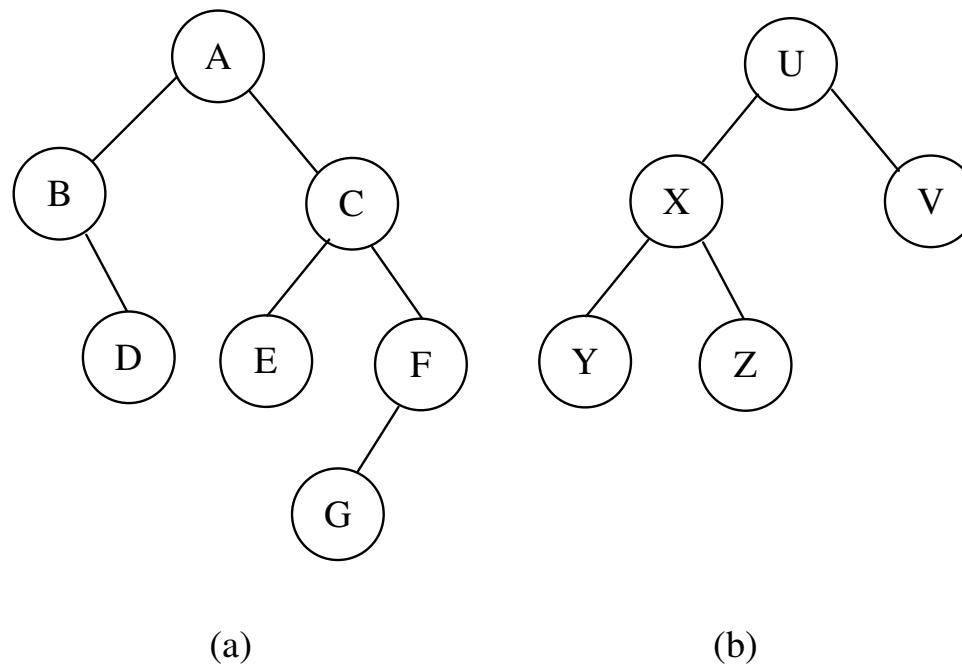


Fig.4.1.1. (a) Un arbore binar nestrikt. (b) Arbore binar strict.

Exemple de aplicații ale structurii de arbore binar strict

- reprezentari de expresii aritmetice cu operatori binari
- algoritmi
- proceduri de decizie
- codificare binara

Proprietăți ale arborilor binari stricti

N_E = numărul nodurilor externe și

N_I = numărul nodurilor interne ale unui arbore binar strict

Propoziția 1. Într-un arbore binar strict, numărul nodurilor externe și al celor interne sunt legate prin relația:

$$N_E = N_I + 1.$$

Demonstrație.

$$2N_I = N_I + N_E - 1,$$

$$N_E = 2N_I - N_I + 1 = N_I + 1,$$

lungime externă a unui arbore binar strict = suma lungimilor drumurilor de la rădăcină până la fiecare nod extern.

E = multimea frunzelor

lungime internă a unui arbore binar strict = suma lungimilor drumurilor de la rădăcină la toate nodurile interioare.

I = multimea nodurilor interioare

$$L_E = \sum_{x \in E} l(r, x) \quad L_I = \sum_{y \in I} l(r, y).$$

r este rădăcina, $l(r, x)$ lungimea drumului de la r la nodul x .

(Drumul de la rădăcină la un nod se măsoară în număr de arce.)

Arbore binari stricti

Aplicatii (continuare):

- la **codificare**
- la **interclasarea optimala** a mai multor siruri

Aplicatii ale arborilor binari la codificare

Fie V un alfabet finit. S.n. *cod binar* pe V o functie $c:V\rightarrow\{0,1\}^*$, *injectiva*. Se extinde in mod canonic la o functie $c:V^*\rightarrow\{0,1\}^*$.

1. Coduri, de lungime: *fixa*, sau *variabila*.
2. Coduri cu *proprietatea prefix*: pt. oricare x,y din V , $c(x)$ nu este prefix al lui $c(y)$.

Probleme: ***codificarea si decodificarea***

Unui *cod binar* pe V , *cu proprietatea prefix*, i se poate asocia un arbore binar, cu arce etichetate cu 0 (fiu stg.) si 1 (fiu dr.). Codul literei x , $c(x) =$ sirul de etichete al drumului unic de la radacina pina la un nod ce contine x .

Exemplu:

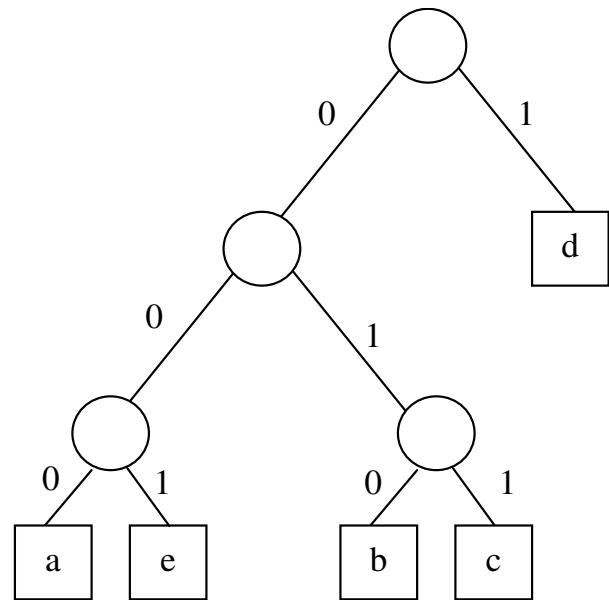


Fig. 4.1.7

- a 000
- b 010
- c 011
- d 1
- e 001

Deci

- constructia unui cod binar cu proprietatea prefix = constructia unui a.b.s. cu literele din V in frunze
- codificare: pt. fiecare frunza, drumul unic de la radacina la ea
- decodificare: drumuri de la rad. la frunze

Problema: daca am avea **informatie suplimentara** despre fiecare litera a lui V , de exemplu informatie despre **frecventa ei de aparitie**, am putea construi un cod ***optim*** (relativ la informatie)/ si care ar fi definitia lui ***optim*** ?

Lungimea codului unei litere sa fie invers proportionala cu frecventa ei de aparitie ... minimizarea lungimii mesajelor

Arborei binari stricți, cu ponderi

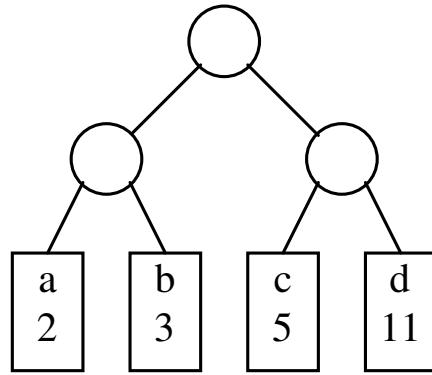
Fie T un arbore binar strict, mulțimea frunzelor $E = \{a_1, a_2, \dots, a_n\}$. T se va numi *cu ponderi* dacă există o funcție cu valori reale $w: E \rightarrow R$, cu alte cuvinte, dacă fiecare frunză a_i are asociat un număr real w_i numit *ponderea* ei.

T a.b.s. cu ponderi, putem considera *lungimea externă ponderată* a lui T , un număr real dat de

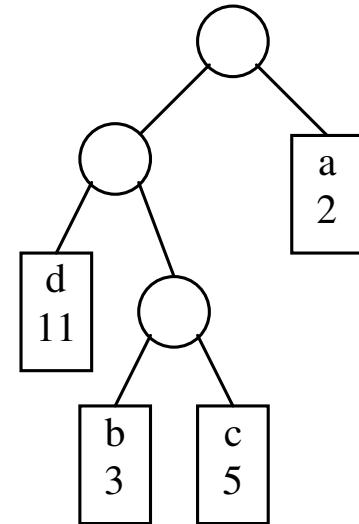
$$L_E^w = \sum_{i=1}^n w_i l_i = \sum_{i=1}^n w_i l(r, a_i).$$

- a. b. s.: la un număr fixat de frunze, pentru care din ei se minimizează lungimea externă? (Propoziția 4): lungimea externă se minimizează pentru aceia care au frunzele repartizate pe cel mult două niveluri adiacente.
- a. b. s. cu ponderi: pentru un număr fixat de frunze, cu ponderi date, fixate și ele $\{(a_1, w_1), \dots, (a_n, w_n)\}$, pentru care arbori se atinge *lungimea externă ponderată minimă*?

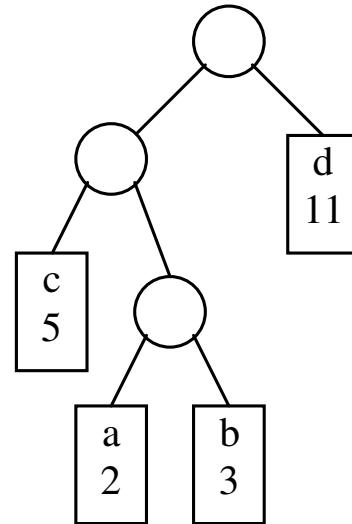
Exemplu: variatia lungimii externe ponderate



(A) $L=42$



(B) $L=48$



(C) $L=36$

Trei a.b.s. pt. frunzele cu ponderi: $(a, 2)$, $(b, 3)$, $(c, 5)$, $(d, 11)$.

Problema: date fiind n ponderi, notate w_1, w_2, \dots, w_n , să se găsească printre toți arborii binari stricți cu n frunze, unul, nu neapărat unic, care să aibă cea mai mică lungime externă ponderată.

Algoritmul lui Huffman

Algoritmul lui Huffman

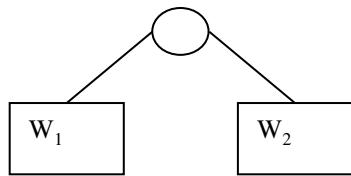
Demonstratie: existenta solutiei

Pentru $n = 1$ soluția va fi arborele binar strict cu un singur nod de tip frunză, cu unica pondere w_1 . Lungimea lui externă ponderată va fi 0.

(ipot. ind.) Presupunem că din $n - 1$ ponderi date, putem construi un arbore binar strict cu lungime externă ponderată minimă.

Fie date n ponderi, w_1, w_2, \dots, w_n și să presupunem că w_1 și w_2 sunt cele mai mici dintre ele. Să considerăm acum sirul de $n - 1$ ponderi, $w_1 + w_2, w_3, \dots, w_n$.

Conform ipotezei de inducție, putem construi un a. b. s., T' , asociat acestui sir de ponderi, cu lungime externă ponderată minimă, L' . Să înlocuim în arborele T' frunza cu ponderea $w_1 + w_2$ cu un subarbore de forma



Arborele obtinut astfel, T , va fi soluție pentru ponderile $w_1, w_2, w_3, \dots, w_n$ căci lungimea lui externă ponderată va fi L , cu $L = L' + w_1 + w_2$, iar L' este minimă și w_1 și w_2 sunt și ele cele mai mici ponderi din sir.

Algoritmul lui Huffman

Presupunem date n ponderi, în ordine descrescătoare

$$w_1 \geq w_2 \geq \dots \geq w_{n-1} \geq w_n.$$

Pasul 1. Algoritmul formează n arbori binari stricți, fiecare de tip frunză, cu câte o pondere w_i asociată ei. Avem o *pădure* cu n arbori.

Pasul 2. Se „leagă“ doi subarbori cu ponderi minime din pădure, cu ajutorul unui nod interior pentru care ei devin cei doi fii, iar acest arbore va fi arbore binar strict cu ponderea asociată egală cu suma ponderilor arborilor pe care i-am legat, pondere pe care o vom asocia nodului intern.

Am redus în felul acesta cu 1 numărul de subarbori din pădure. Se reia pasul iterativ (2) pînă când obținem un singur arbore.

În general, la iteratia k , algoritmul are $n - k - 1$ arbori binari stricți cu ponderi. „Leagă“ doi arbori cu ponderile cele mai mici conform procedeului descris anterior, producând $n - k$ arbori binari stricți cu ponderi. La al $(n - 1)$ -lea pas iterativ, vom avea un singur arbore. S.n. **arbore Huffman** asociat ponderilor date.

Algoritmul lui Huffman (cont.)

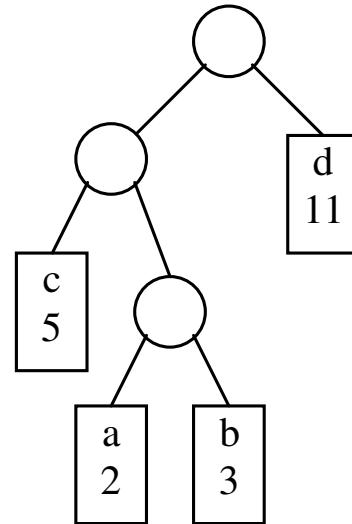
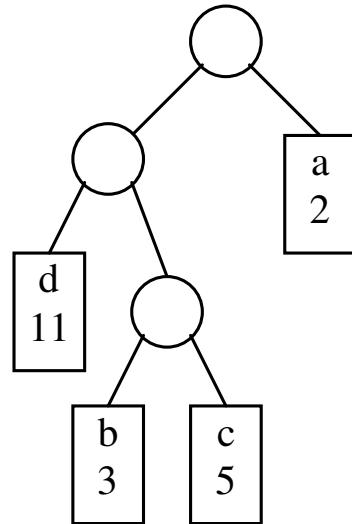
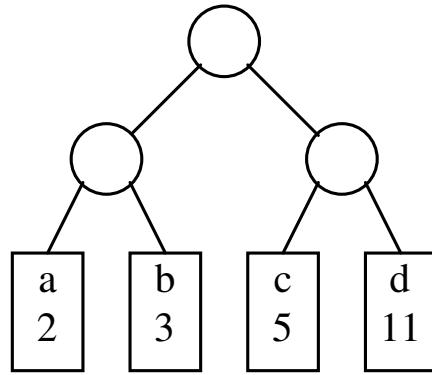
Un arbore binar strict obtinut prin aplicarea algoritmului lui Huffman se va numi *arbore Huffman* asociat ponderilor $\{w_1, w_2, \dots, w_n\}$.

Obs. 1: *algoritmul* ia in considerare doar *ponderile*.

Informatica continuta in frunze este *importanta* pentru aplicatii, natura ei *difera* de la o aplicatie la alta.

Obs. 2: arborele Huffman **nu** este in general unic.

Exemplu: variatia lungimii externe ponderate

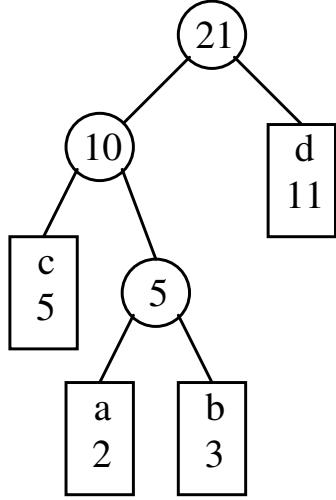


(A) $L=42$

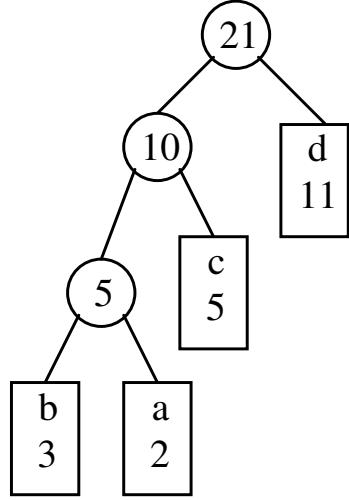
(B) $L=48$

(C) $L=36$

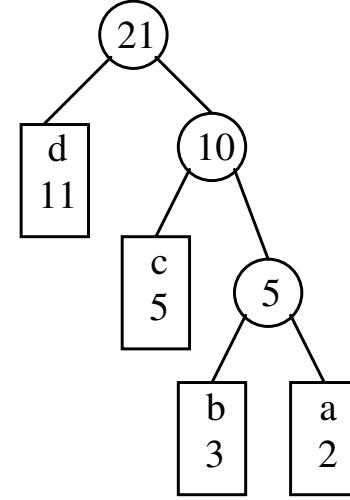
Trei a.b.s. pt. frunzele cu ponderi: $(a, 2)$, $(b, 3)$, $(c, 5)$, $(d, 11)$.
Arborele din fig. (C) este Huffman, $L = 36$ minima.



(C) L=36



(D) L=36



(E) L=36

Arbore Huffman pt. frunzele cu ponderi $\{(a, 2), (b, 3), (c, 5), (d, 11)\}$.

$L = 36$ minima.

Aplicatii ale arborilor binari (stricti) la codificare

Problemele pe care ni le punem sunt:

- (a) De *construit un cod* binar cu proprietatea prefix peste un alfabet dat V .
- (b) Să *codificăm* cuvinte peste V adică: dat fiind orice cuvânt peste V , $a_1a_2 \dots a_n \in V^*$, să producem codul asociat lui, din codurile literelor. Prin formula

$$c(a_1a_2 \dots a_n) = c(a_1)c(a_2) \dots c(a_n).$$

- (c) Să *decodificăm* şiruri de biţi, adică: dat fiind orice cuvânt $u \in \{0, 1\}^*$, să decidem dacă există sau nu un cuvânt $x \in V^*$ al cărui cod este u și, în cazul afirmativ, să spunem care este acest x (unic) cu proprietatea $c(x) = u$.
- (a) echivalenta cu problema constructiei unui a.b.s. pentru multime de frunze V .

Aplicatii ale arborilor Huffman la codificare

Problema: daca am avea **informatie suplimentara** despre fiecare litera a lui V , de exemplu informatie despre **frecventa ei de aparitie**, am putea construi un cod *optim* (relativ la informatie)/ si care ar fi definitia lui *optim* ?

Lungimea codului unei litere sa fie invers proportionala cu frecventa ei de aparitie ... minimizarea lungimii mesajelor

Aplicatii ale arborilor Huffman la codificare

alfabetul de codificat, V , se dă împreună cu o repartiție de probabilitate pe el; cu alte cuvinte avem o funcție

$$w: V \rightarrow [0, 1]$$

cu proprietatea

$$\sum_{a \in V} w(a) = 1, a \in V.$$

Problema *constructiei unui cod* cu proprietatea prefix, devine o problemă de construcție a unui arbore binar strict *cu ponderi*, având ca frunze elementele lui V cu ponderile (probabilitățile) asociate, adică, mulțimea frunzelor va fi

$$E = \{(a, w(a)) \mid a \in V\}.$$

In plus, *codul sa minimizeze lungimea medie a mesajelor*, devine ... a.b.s. cu *lungime externa ponderata minima*.

Aplicatii ale arborilor Huffman la codificare

Un cod binar peste un alfabet V dat, împreună cu o probabilitate $w : V \rightarrow [0, 1]$, cod asociat unui arbore binar strict cu ponderi construit cu algoritmul lui Huffman, se numește *cod Huffman*.

(a) *Construcția codului*: revine la aplicarea algoritmului lui Huffman pentru construirea unui arbore binar strict cu lungime externă ponderată minimă pentru mulțimea de frunze ponderate

$$E = \{(a, w(a)) \mid a \in V\}.$$

(b) *Codificarea*. Fie cărui caracter $a \in V$ i se asociază codul constând din sirul de etichete al arcelor ce compun drumul de la rădăcină la frunza asociată caracterului a .

(c) *Decodificarea* unui sir de biți revine la parcurgeri repetitive ale către unui drum în arborele lui Huffman, începând de la rădăcină, conform convenției: în cazul în care caracterul (bitul) curent este 0, parcurgerea continuă pe fiul stâng, dacă este 1, parcurgerea continuă pe fiul drept. De fiecare dată când ajungem într-o frunză, am terminat de decodificat un caracter și reluăm parcurgerea de la rădăcină pentru restul sirului de biți. Dacă o asemenea parcurgere se termină într-un nod interior, atunci sirul $u \in \{0, 1\}^*$ de decodificat nu este valid, adică nu există nici un cuvânt $x \in V^*$, astfel încât $c(x) = u$.

ASD - Heapsort (II)

1 Sortarea prin selectie folosind structuri arborescente (Sortarea cu ansamble/HeapSort)

1.1 Arbori partial ordonati si ansamblu

Def Se numeste **arbore partial max-ordonat** un arbore binar cu chei de un tip total ordonat si cu proprietatea ca in orice nod **u** al sau avem relatiile:

$$\begin{cases} \text{info}[u] > \text{info}[\text{root}(\text{left}[u])], \text{ daca } \text{left}[u] \text{ este nevid} \\ \text{info}[u] > \text{info}[\text{root}(\text{right}[u])], \text{ daca } \text{right}[u] \text{ este nevid} \end{cases}$$

Pentru \forall nod u : $\begin{cases} \text{info}[u] > \text{info}[v], \forall v \in \text{left}[u] \\ \text{info}[u] > \text{info}[w], \forall w \in \text{right}[u] \end{cases}$
 \Rightarrow **cheia maxima se va afla in radacina**

Conceptul de arbore partial min-ordonat se defineste analog.

Def: Arbore binar **complet pe niveluri** este un a.b. cu toate nivelurile pline, eventual cu exceptia ultimului nivel, unde toate nodurile vor fi aliniate cel mai la stanga.

Acest tip de arbore binar se poate reprezenta ca vector (deci alocare statica si acces in timp 1 la fi si la tata).

Def: Se numeste ansamblu (max-ansamblu) un arbore binar max-ordonat si complet pe niveluri, reprezentat ca vector.

Conceptul de min-ansamblu se defineste analog.

1.2 Inserarea intr-un ansamblu

Se urmeaza pasii:

1. Se pune nodul de inserat (**nod**) pe ultimul nivel al arborelui, aliniat cel mai la stanga. (*arborele ramane complet*).
2. Se repeta (eventual pana la radacina) comparatia intre **info[nod]** si **info[tata[nod]]**
 - (a) Daca $info[nod] < info[tata[nod]]$ atunci am gasit locul lui nod in ansamblu (noua cheie nu violeaza conditia de arbore max-ordonat)
 - (b) Daca nu, interschimb nod cu tata[nod] si reluam de la (a).

```
procedure InsHeap(Ans, n, Val) in ansamblul Ans[1...n] se insereaza Val
    n ← n + 1      creste dimensiunea ansamblului cu 1
    p ← n          p = indice pentru nodul curent
    While p > 1 do      cat timp nu am ajuns la radacina
        Tata ← p div 2
        If Val ≤ Ans[Tata] then
            Ans[p] ← Val      se insereaza - cazul (a)
            exit                am terminat
        Else
            Ans[p] ← Ans[Tata]      se coboara tatal in locul fiului
            p ← Tata                nodul curent se reactualizeaza
        endIf
    endWhile
    Ans[1] ← Val      inserarea in radacina - cazul (b)
endproc
```

1.3 Construirea unui ansamblu. Asamblarea

Se realizeaza prin inserari repetitive: daca avem cel putin o cheie, atunci ea se va insera in radacina (un arbore cu un nod este ansamblu).

- pt fiecare valoare noua se foloseste algoritmul de inserare **InsHeap**.

Asamblarea, operatie specifica, este construirea unui ansamblu din inserari repetitive de chei care se afla deja in locatiile unui vector:

- A[1] este ansamblu
- la fiecare pas iterativ j se apeleaza procedura de inserare in ansamblul A[1..j] a valorii A[j+1], pentru $j=1, \dots, n-1$.

```
procedure Asamblare (A,n)
    For  $j \leftarrow 1, n-1$  do
        InsHeap(A,j,A[j+1])
    endFor
endproc
```

1.4 Extragerea maximului sau decapitarea unui ansamblu

1. Se extrage valoarea din radacina in vederea procesarii;
2. Se inlocuieste radacina cu ultimul nod (arborele binar ramane complet, dar eventual nu mai e max-ordonat)
3. Coboram noua radacina la locul ei prin comparatii cu cel mai mare dintre fii.

procedure DelHeap (Ans, n, Val)

$Val \leftarrow Ans[1]$	<i>extragerea radacinii</i>
$Last \leftarrow Ans[n]$	<i>Last e ultimul elem din ans ce tb inserat in Radacina, apoi tb sa cautam locul lui</i>
$n \leftarrow n - 1$	<i>scade dimensiunea ansamblului</i>

$p \leftarrow 1; l \leftarrow 2; r \leftarrow 3$ *p indice nod curent, l - fiu stang,r - fiu drept*

While ($r \leq n$) **do** *test de nedepasire a structurii*

If ($Last \geq Ans[l]$) **AND** ($Last \geq Ans[r]$) **then**

$Ans[p] \leftarrow Last$ *inserarea*

 exit *am terminat*

endIf

If $Ans[l] \geq Ans[r]$ **then** *continuam pe ramura stanga*

$Ans[p] \leftarrow Ans[l]$

$p \leftarrow l$ *reactualizarea lui p*

Else *continuam pe ramura dreapta*

$Ans[p] \leftarrow Ans[r]$

$p \leftarrow r$ *reactualizarea lui p*

endIf

$l \leftarrow 2 * p; r \leftarrow l + 1;$ *actualizarea fililor lui p*

endWhile

If ($l=n$) **AND** ($Last < Ans[l]$) **then**

$Ans[p] \leftarrow Ans[l]$

$p \leftarrow l$

endIf

$Ans[p] \leftarrow Last$ *inserarea propriu-zisa a lui Last la locul lui*

endproc

1.5 Complexitatea operatiilor intr-un ansamblu

Sortarea cu ansamble

1. (Pas 0) Se asambleaza vectorul $A[1..n]$. Maximul va fi pe $A[1]$;
2. (Pas 1) Se decapiteaza ansamblul $A[1..n]$, cu reasamblarea lui $A[1..n-1]$ si se pune maximul pe $A[n]$;
3. (Pas j) Se decapiteaza ansamblul $A[1..n-j+1]$, cu reasamblarea lui $A[1..n-j]$ si se pune maximul pe $A[n-j+1]$.

Dupa $n-1$ pasi iterativi vectorul A este sortat.

```
procedure HeapSort (A,n) sortam vectorul A[1..n]
    Asamblare (A,n)
    While  $n > 1$  do
        DelHeap (A,n,Val)
         $A[n + 1] \leftarrow Val$ 
    endWhile
endproc
```

Complexitate: $O(n \lg n)$.

Quick Sort

2012

Algoritmul

QuickSort ca Divide and Conquer

```
procedure QS(Left, Right)
    if (Right - Left + 1) = 1 then
        // (Right - Left + 1) reprezintă dimensiunea vectorului
        return;                                // nu facem nimic
    else
        // Următorul apel corespunde pasului Divide.
        Partition(Left,Right,q);                // Returnează pivotul în q.
        // Pivotul este pus la locul său final.
        QS(Left, q-1);                        // 1 apel recursiv
        QS(q+1, Right);                      // 2 apel recursiv
        // Pentru pasul Combine nu trebuie să facem nimic.
    endif
endproc
```

Procedura Partition

Pentru a sorta vectorul $A[1..n]$ apelul principal va fi $QS(1,n)$;

Scurtă animație a algoritmului

Despre partii la Quick Sort

Partiția Hoare

Procedura Partition

Procedura de partiționare Hoare pe subvectorul $A[Left..Right]$ este următoarea:

Partiția Hoare

Procedura Partition

Procedura de partiționare Hoare pe subvectorul $A[Left..Right]$ este următoarea:

```
procedure Partition(Left, Right)
    // iLeft=indicele curent pentru parcurgerea (2a)
    // iRight=indicele curent pentru parcurgerea (2b)
    iLeft:=Left; iRight=Right;           // inițializarea indicilor curenți pentru parcurgeri
    x:=A[(Left+Right)div 2];           // algearea pivotului în poziție mediană
    repeat                                // partitura
        while A[iLeft] < x do          // (2a)
            iLeft:= iLeft+1;
        endwhile
        while A[iRight] > x do          // (2b)
            iRight:= iRight - 1;
        endwhile
        if iLeft ≤ iRight then          // (3)
            Interschimbă(A[iLeft], A[iRight]);
            iLeft:= iLeft+1;
            iRight:= iRight-1;
        endif
        until (iLeft > iRight)
endproc
```

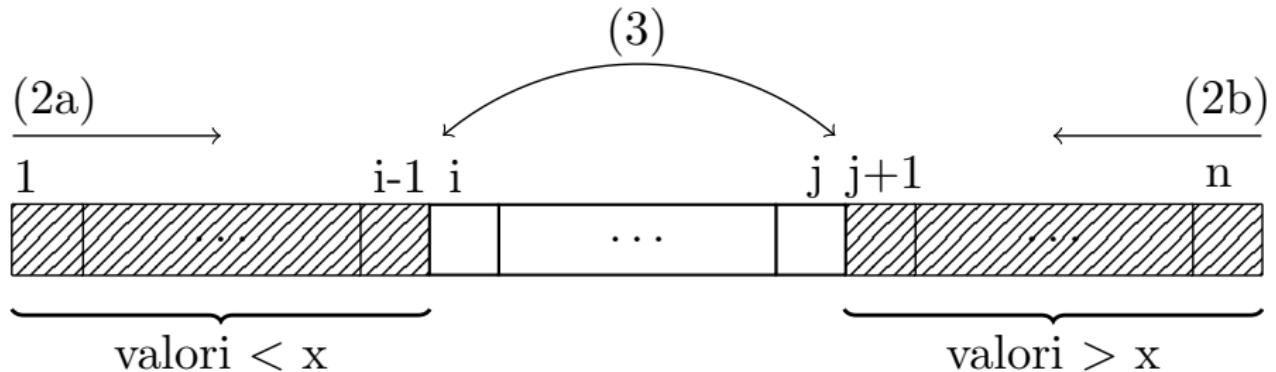


Figure : Procedura de partitōnare.

Exemplu:

x:=A[6]

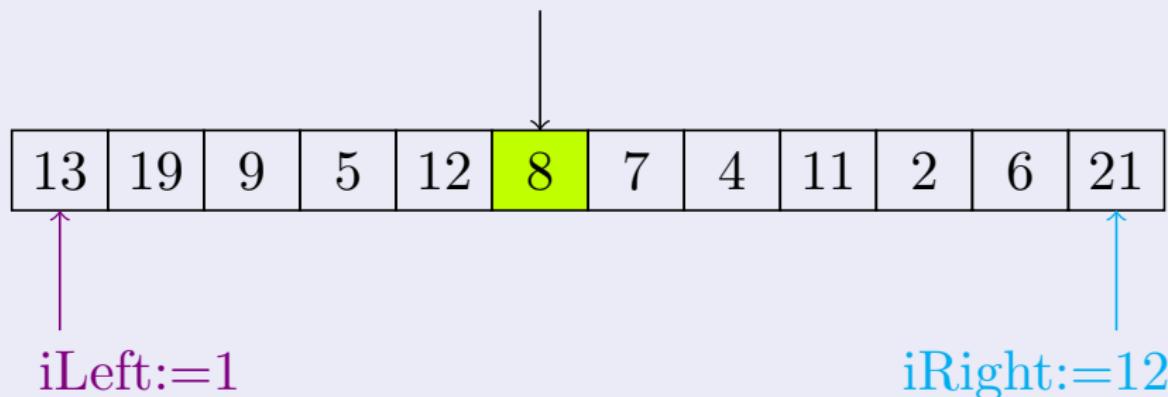


Figure : vectorul A inițial

```
x:=A[(Left+Right)div 2];  
    iLeft:=1; iRight=12;
```

Exemplu:

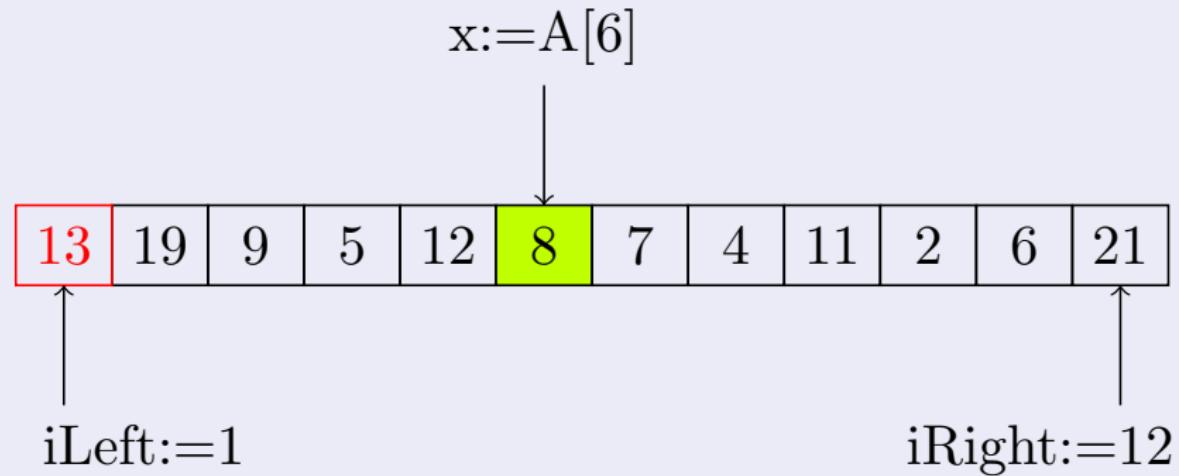


Figure : (2a)

$A[1] \geq x$, deci nu putem avansa cu $iLeft$.

Exemplu:

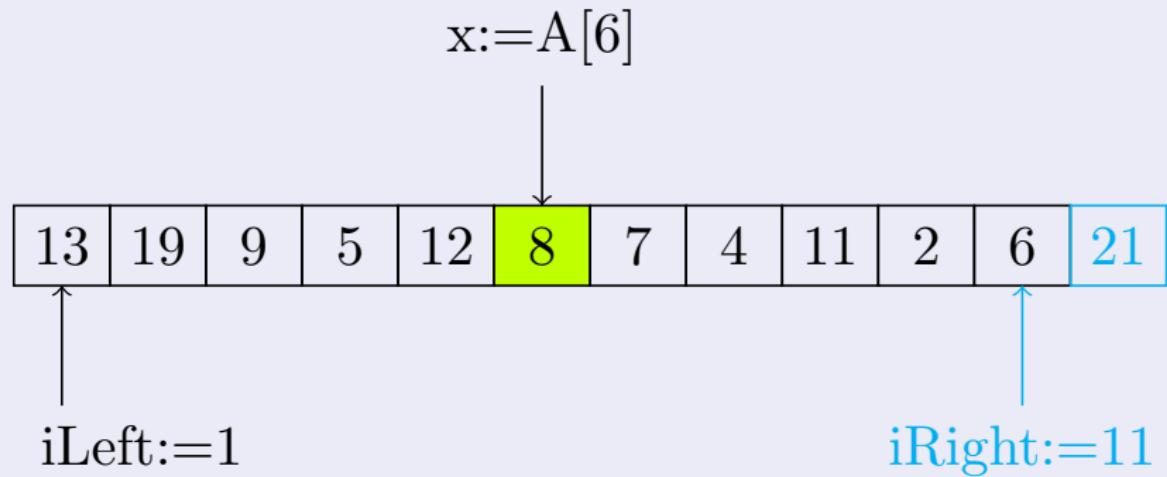


Figure : (2b)

$A[12] > x$, deci $iRight := iRight - 1$.

Exemplu:

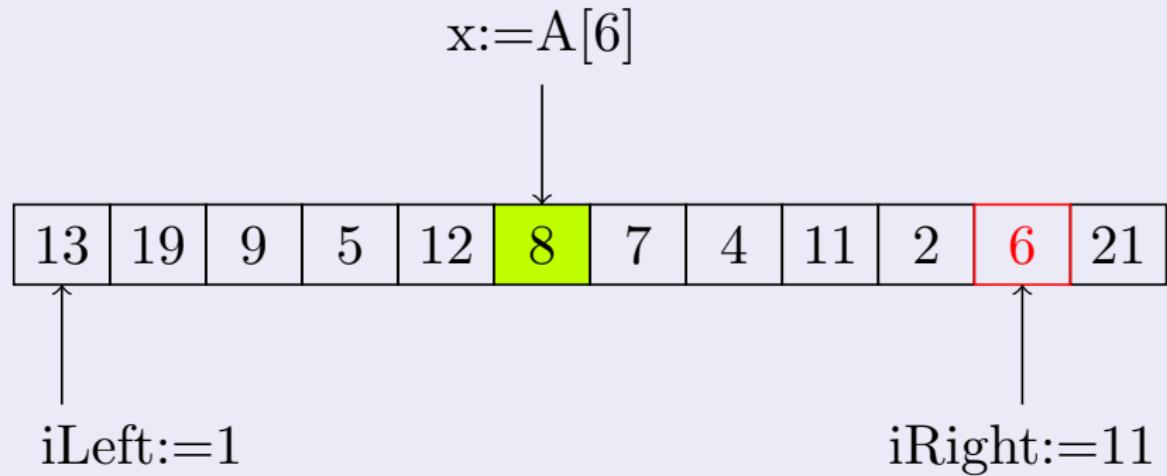


Figure : (2b)

$A[11] \leq x$, deci $iRight$ nu mai avansează.

Exemplu:

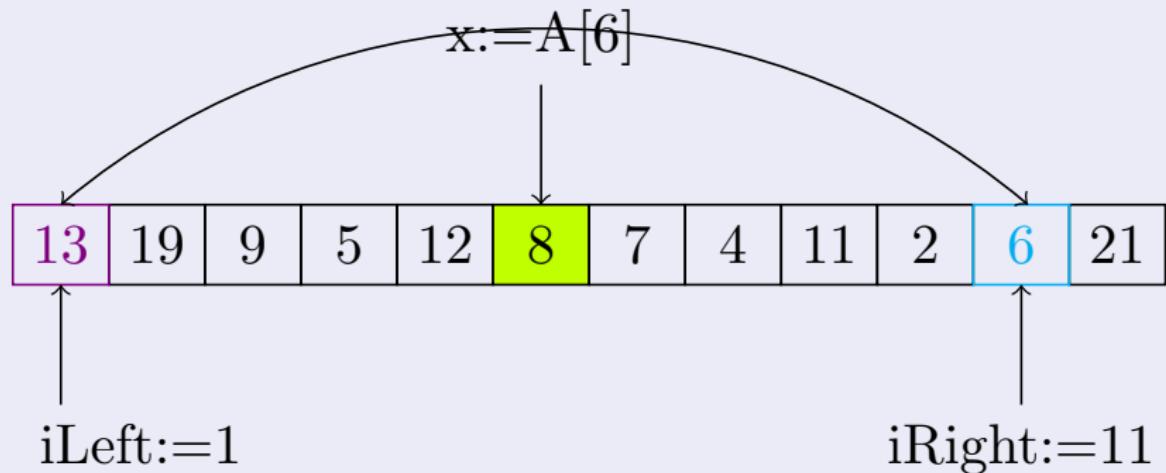


Figure : (3)

Trebuie să interschimbăm $A[1]$ cu $A[11]$.

Exemplu:

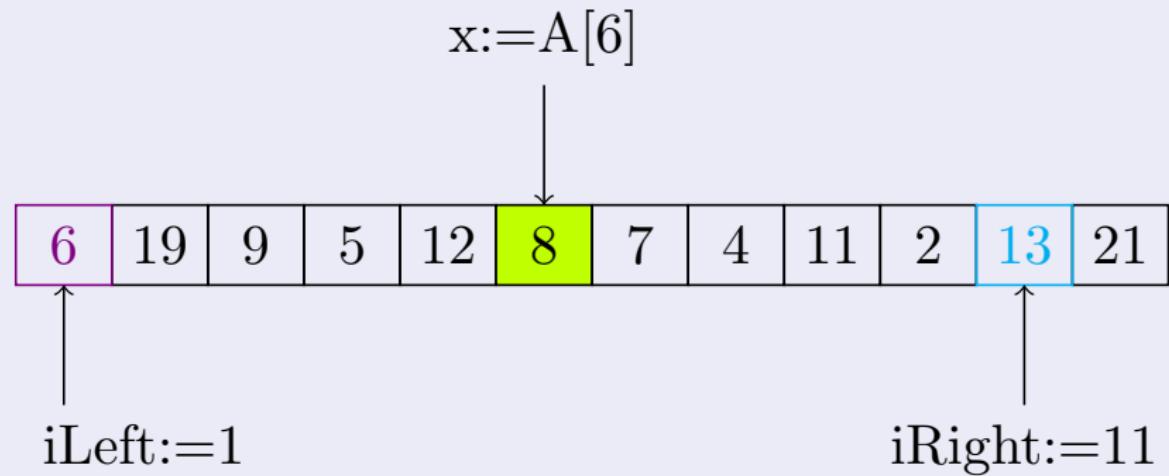


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

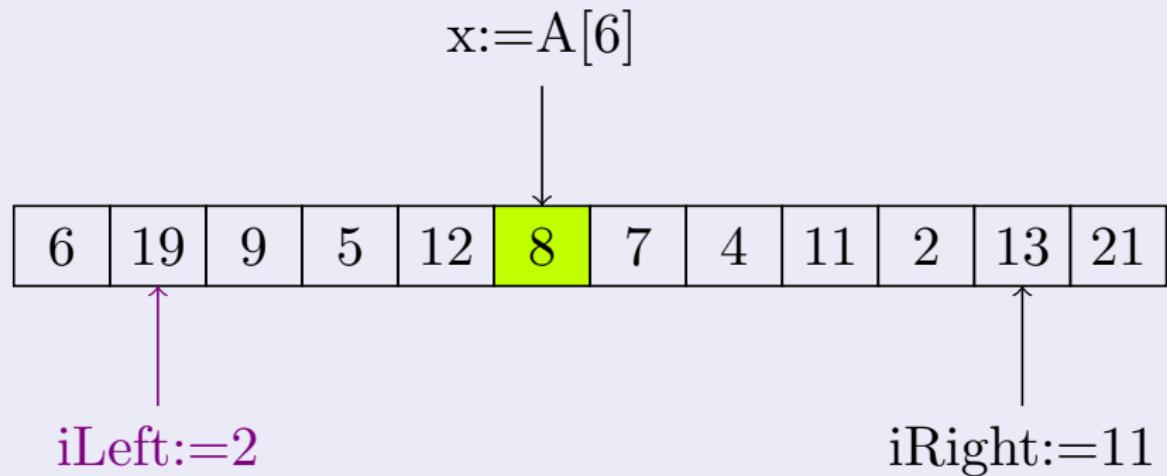


Figure : (3)

$iLeft := iLeft + 1;$

Exemplu:

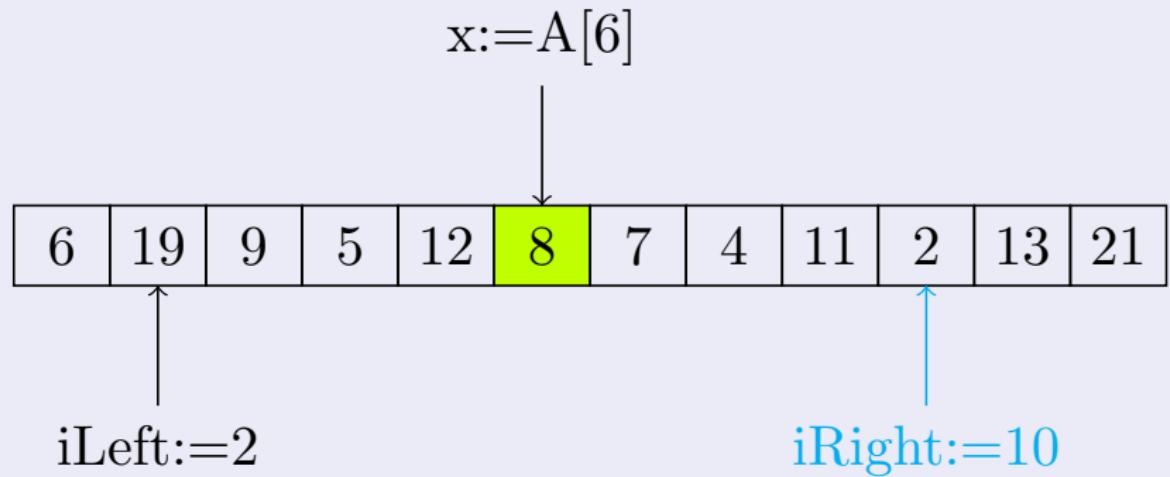


Figure : (3)

$iRight := iRight - 1;$

Exemplu:

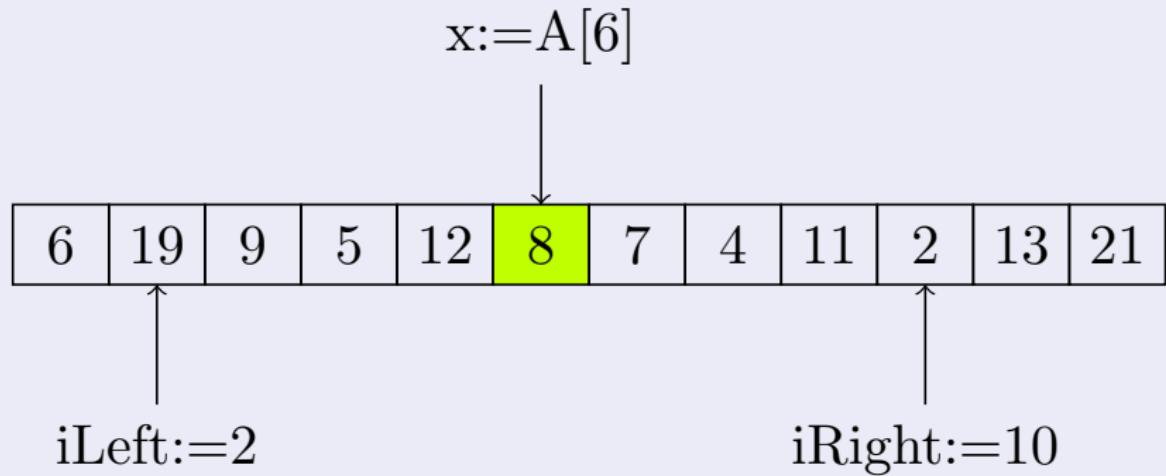


Figure : După prima iterație a ciclului repeat

Avem $2 = iLeft \leq iRight = 10$, deci continuăm procedura de partitie.

Exemplu:

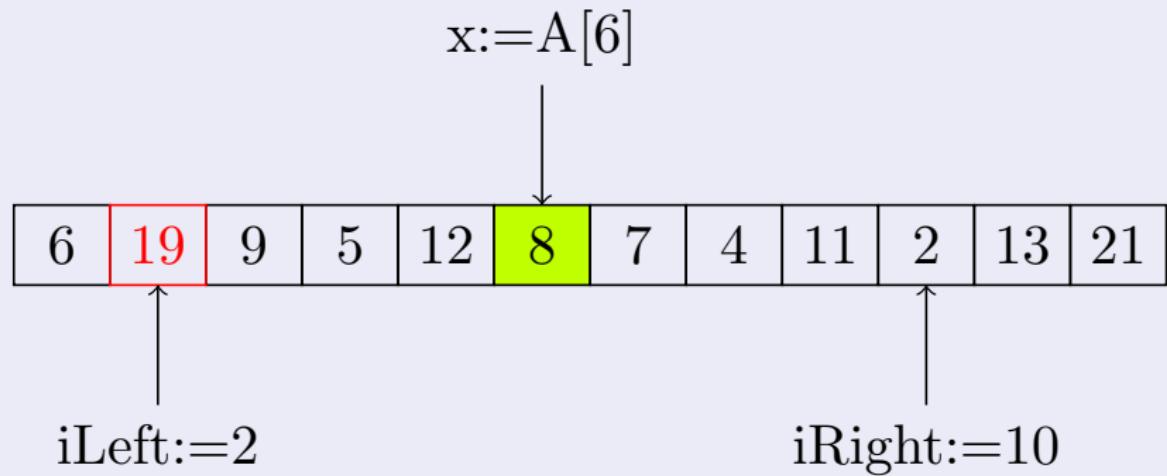


Figure : (2a)

$A[2] \geq x$, deci nu putem avansa cu $iLeft$.

Exemplu:

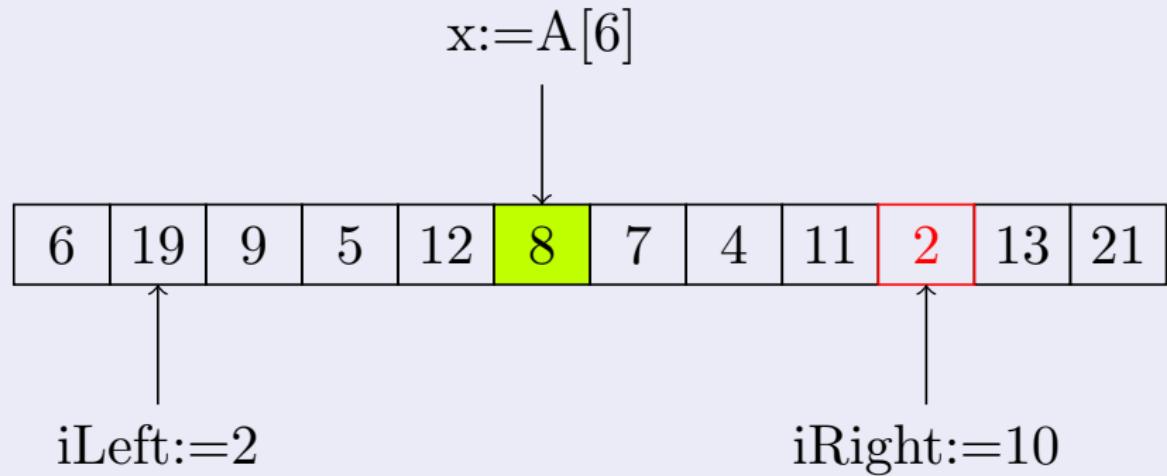


Figure : (2b)

$A[10] \leq x$, deci nu putem avansa cu $iRight$.

Exemplu:

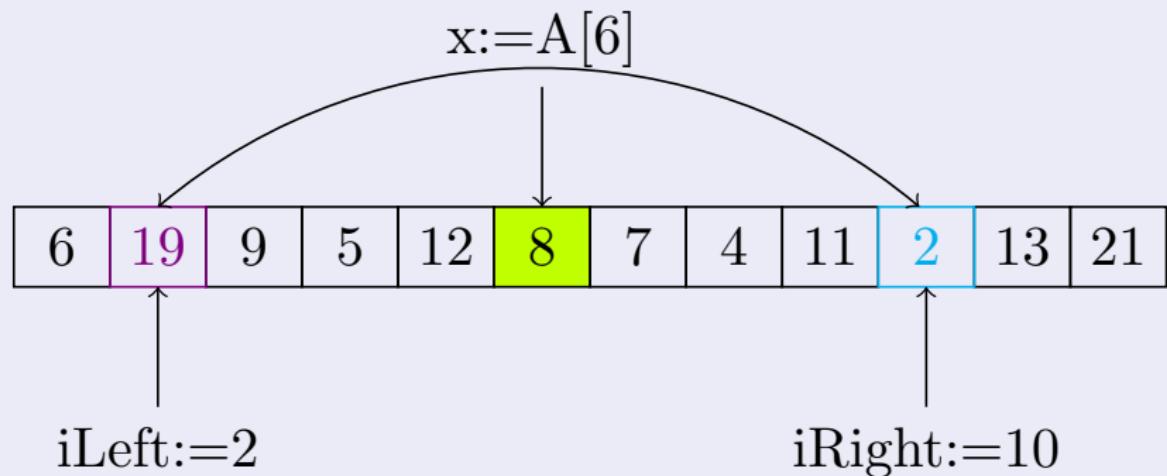


Figure : (3)

Trebuie să interschimbăm $A[2]$ cu $A[10]$.

Exemplu:

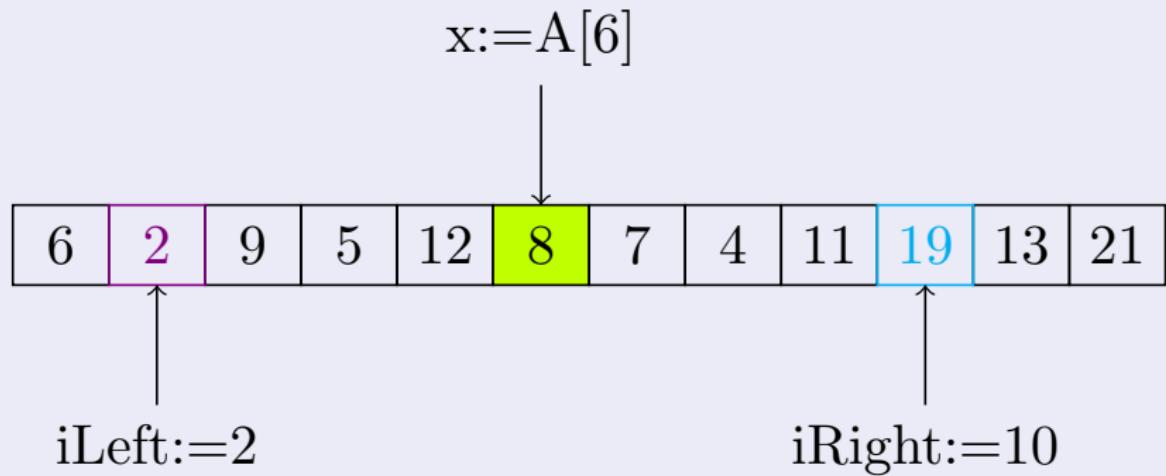


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

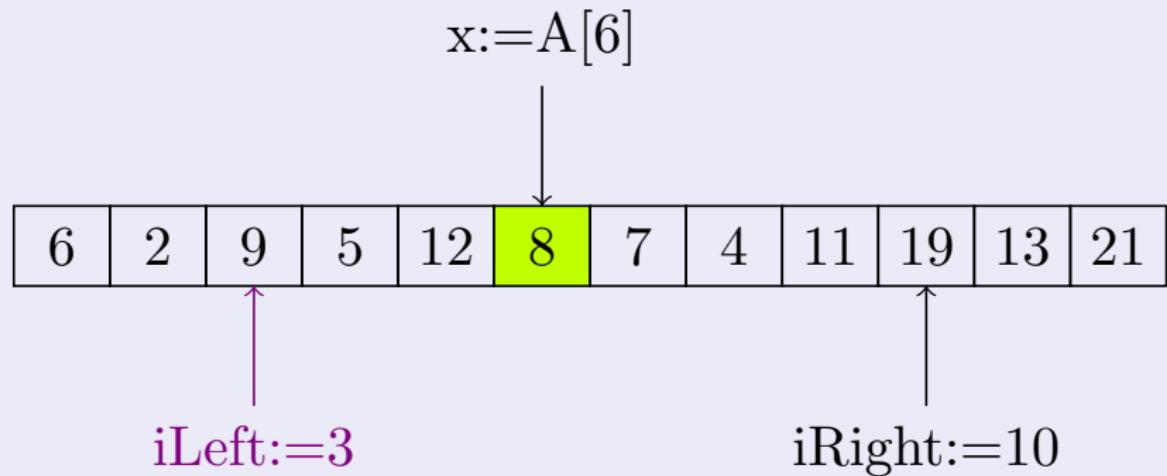


Figure : (3)

$iLeft := iLeft + 1;$

Exemplu:

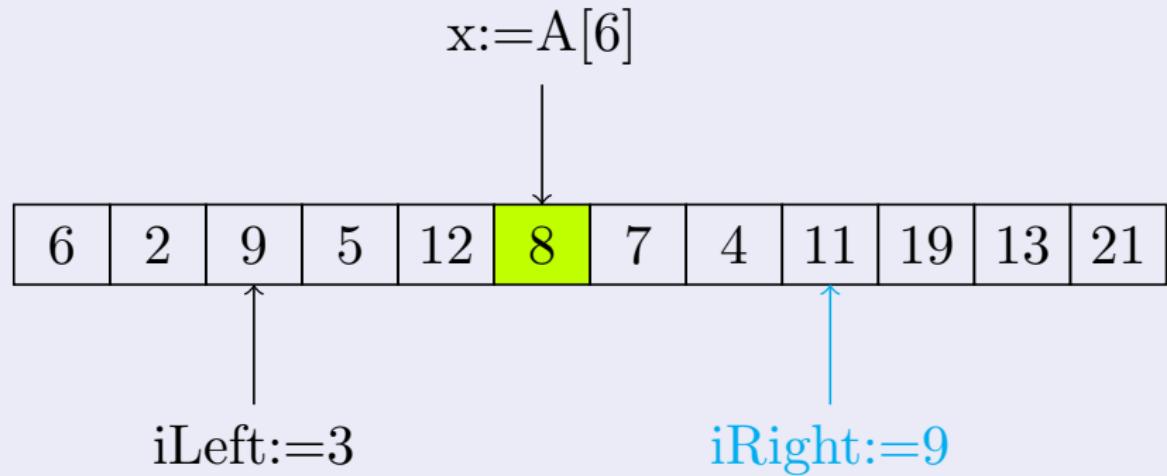


Figure : (3)

$iRight := iRight - 1;$

Exemplu:

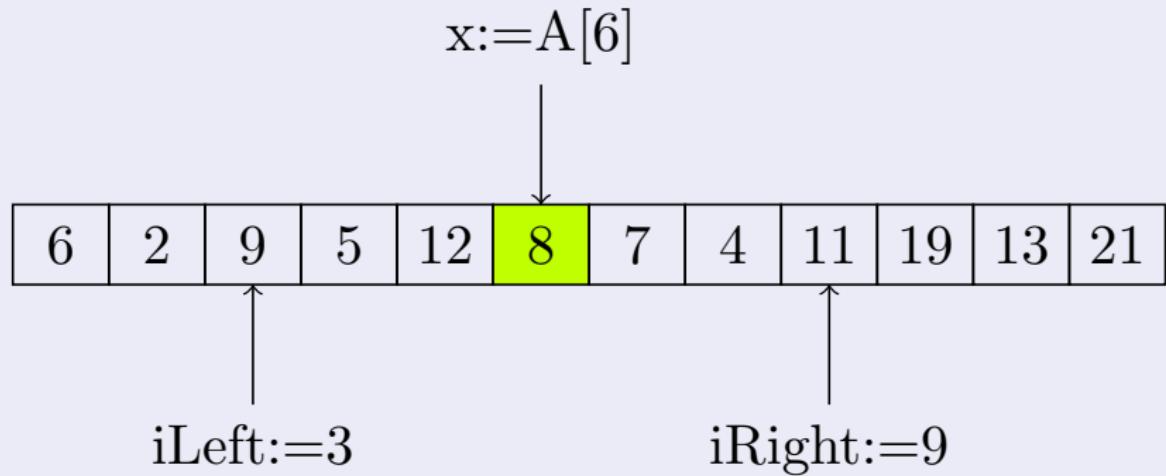


Figure : (Vectorul după a două iterăție a ciclului repeat)

Avem $3 = \underline{iLeft} \leq iRight = 9$, deci continuăm procedura de partitie.

Exemplu:

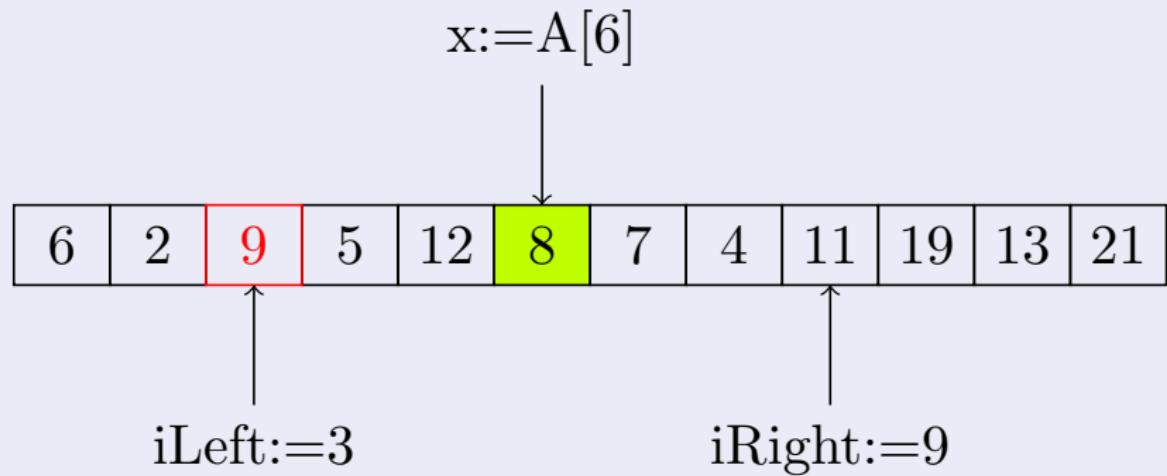


Figure : (2a)

$A[3] \geq x$, deci nu putem avansa cu $iLeft$.

Exemplu:

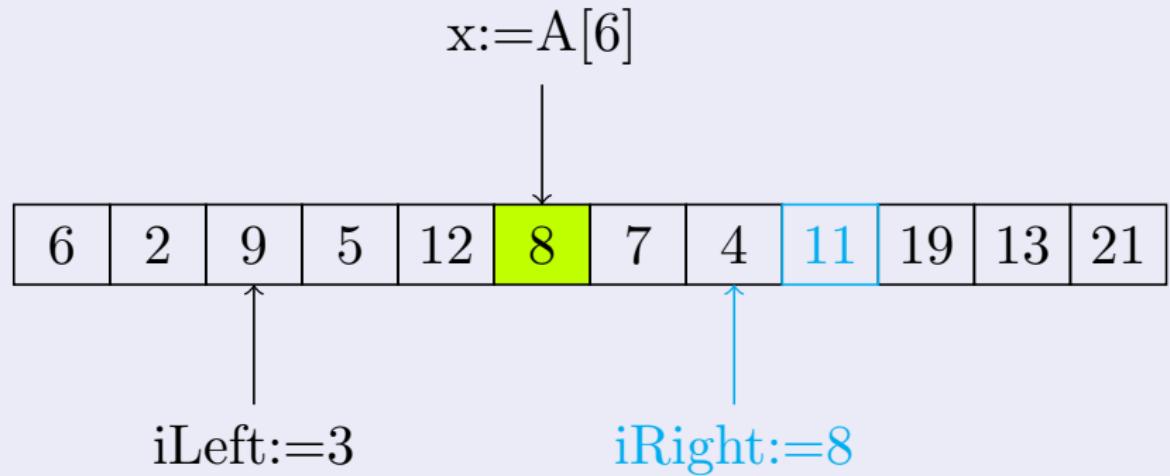


Figure : (2b)

$A[9] > x$, deci $iRight := iRight - 1$.

Exemplu:

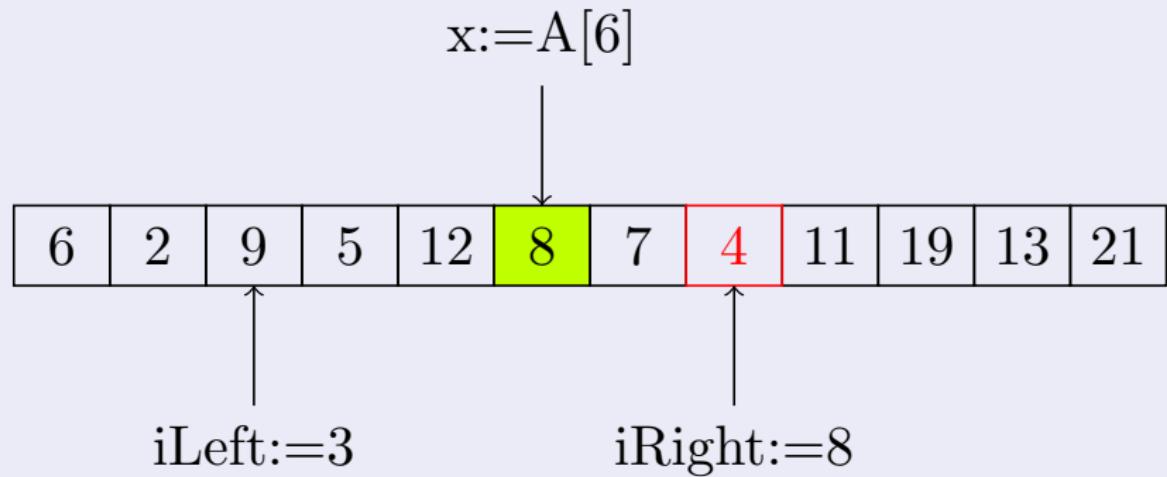


Figure : (2b)

$A[8] \leq x$, deci nu putem avansa cu $iRight$.

Exemplu:

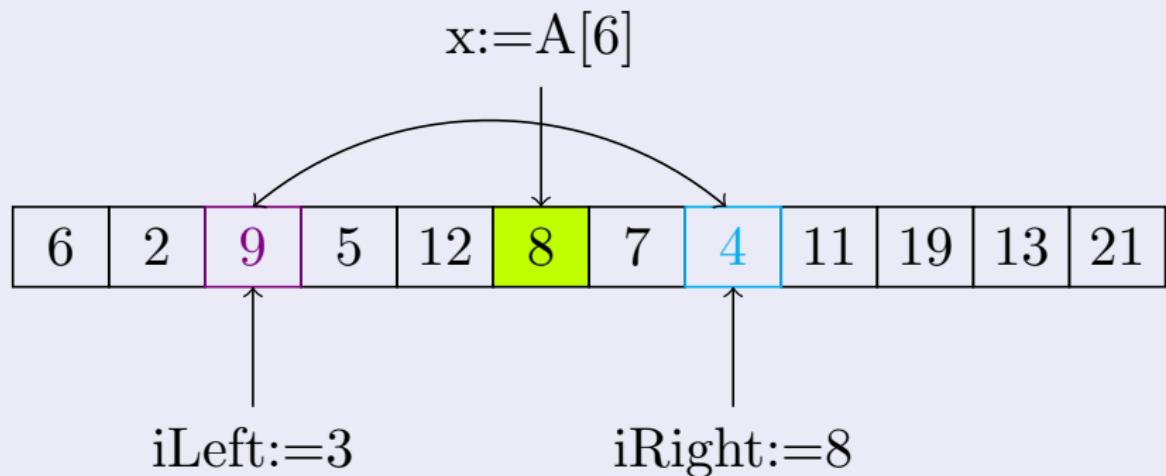


Figure : (2b)

Trebuie să interschimbăm $A[3]$ cu $A[8]$.

Exemplu:

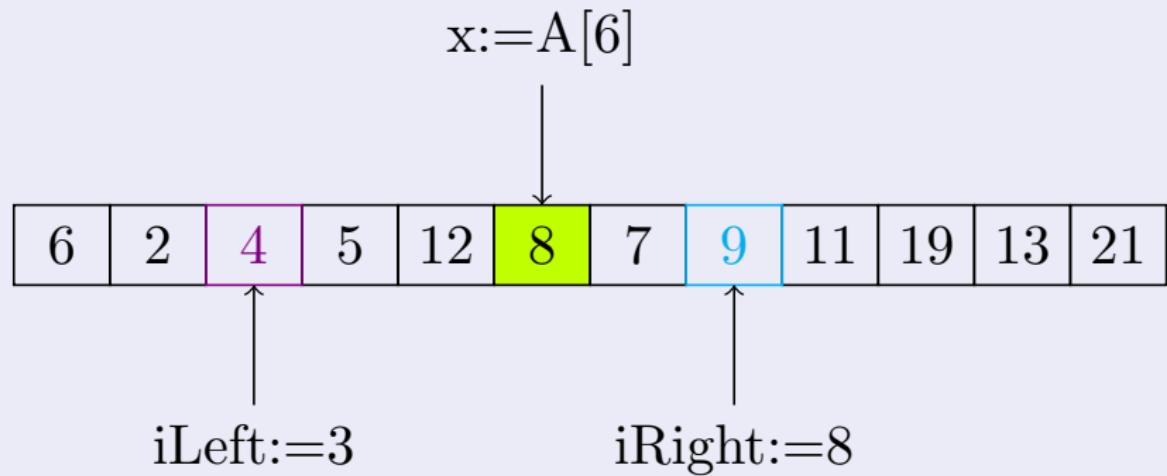


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

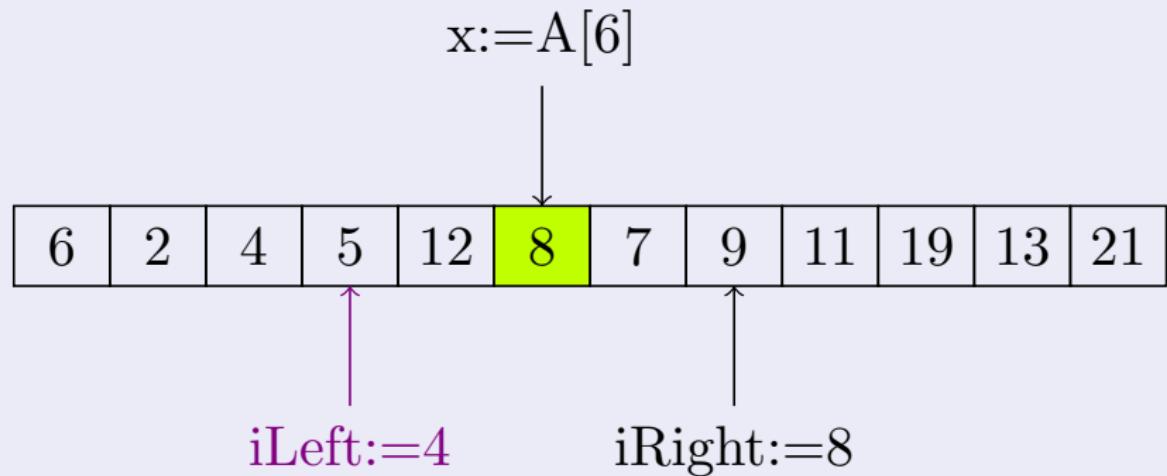


Figure : (3)

$iLeft := iLeft + 1;$

Exemplu:

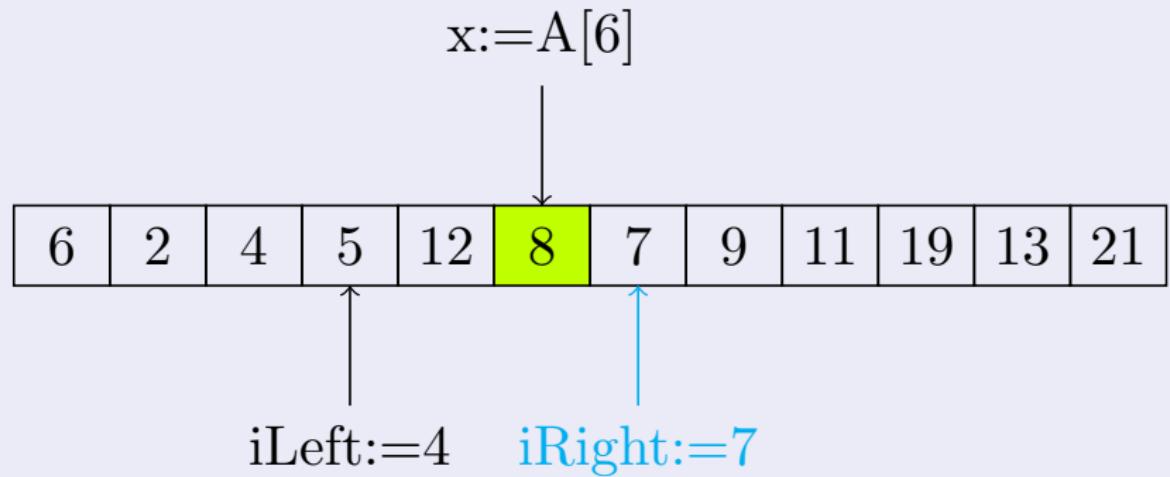


Figure : (3)

$iRight := iRight - 1;$

Exemplu:

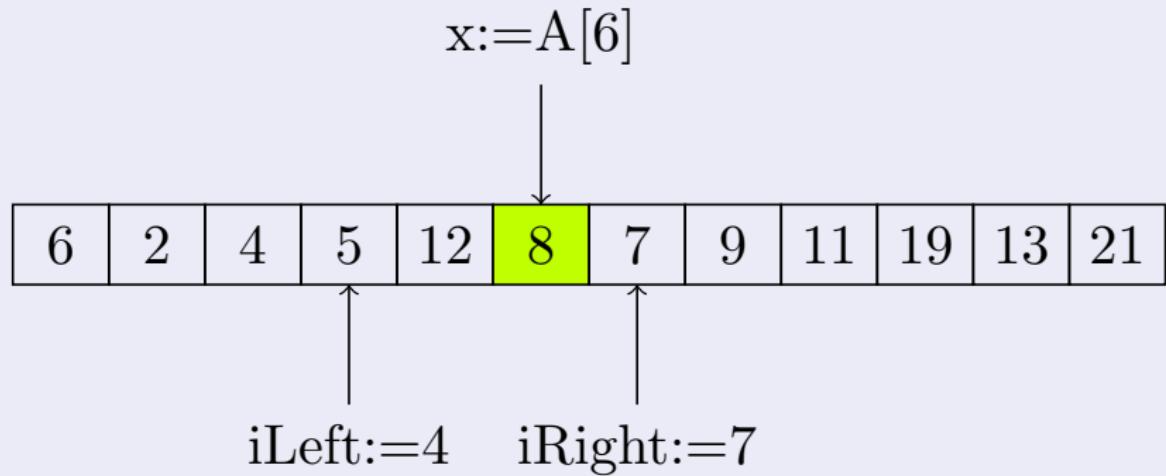


Figure : (Vectorul după a treia iterare a ciclului repeat)

Avem $4 = iLeft \leq iRight = 7$, deci continuăm procedura de partitie.

Exemplu:

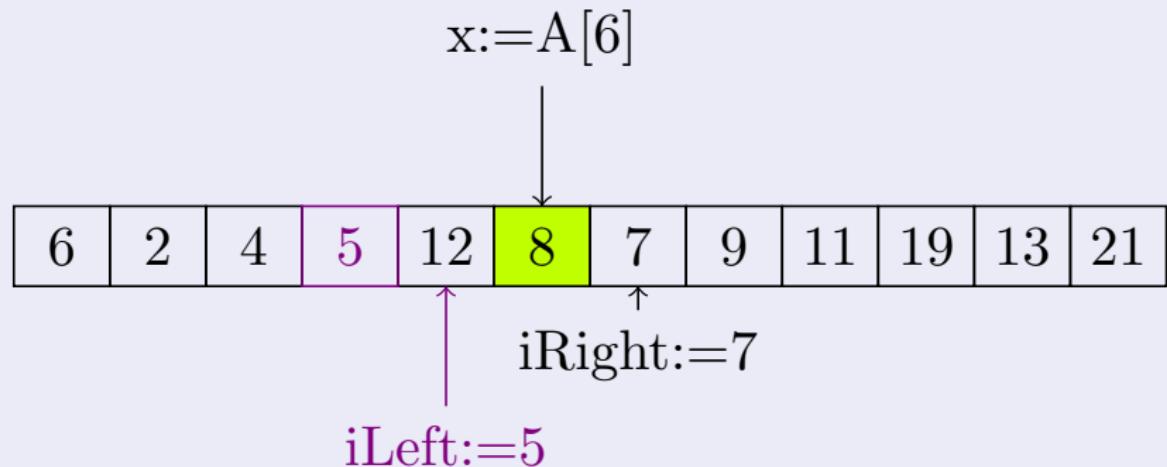


Figure : (2a)

$A[4] < x$, deci $iLeft := iLeft + 1$.

Exemplu:

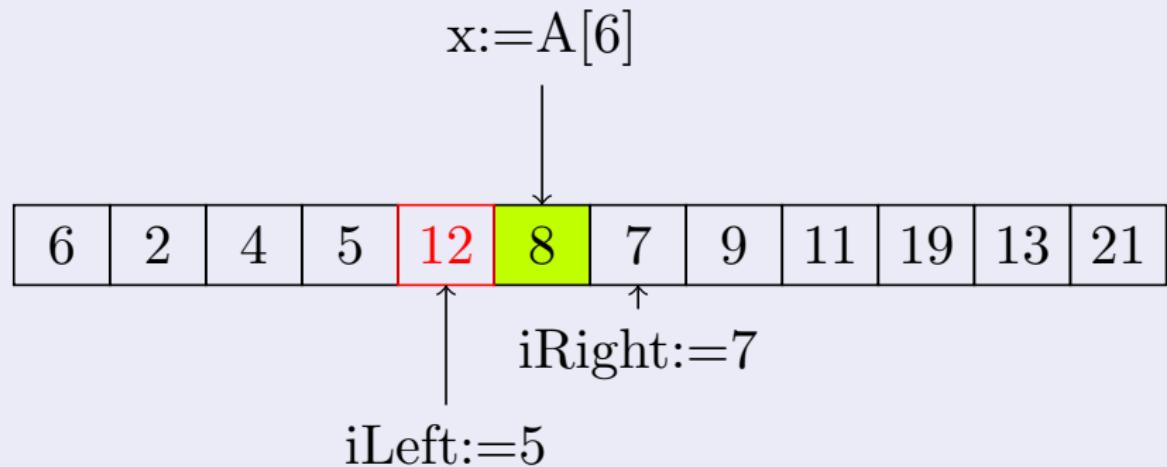


Figure : (2a)

$A[5] \geq x$, deci nu putem avansa cu $iLeft$.

Exemplu:

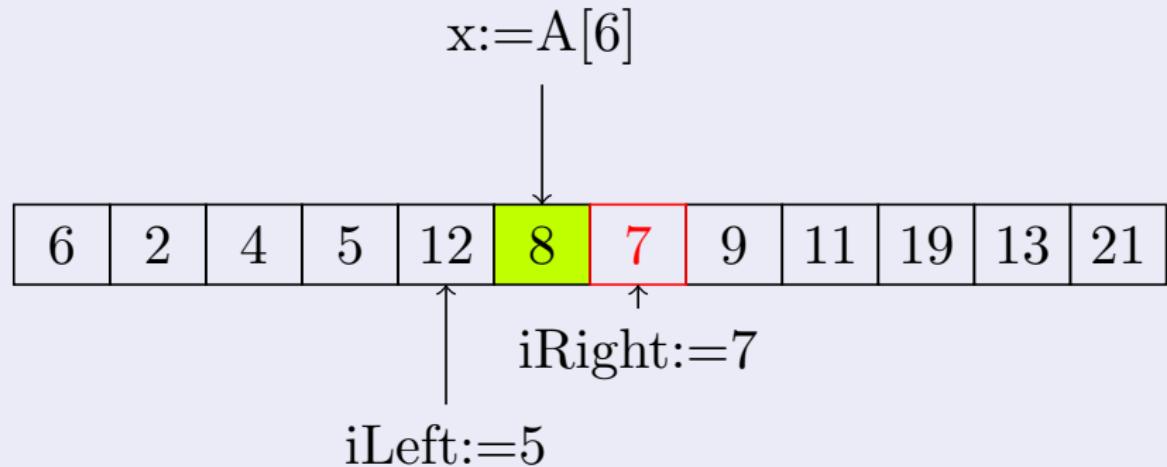


Figure : (2b)

$A[7] \leq x$, deci nu putem avansa cu $iRight$.

Exemplu:

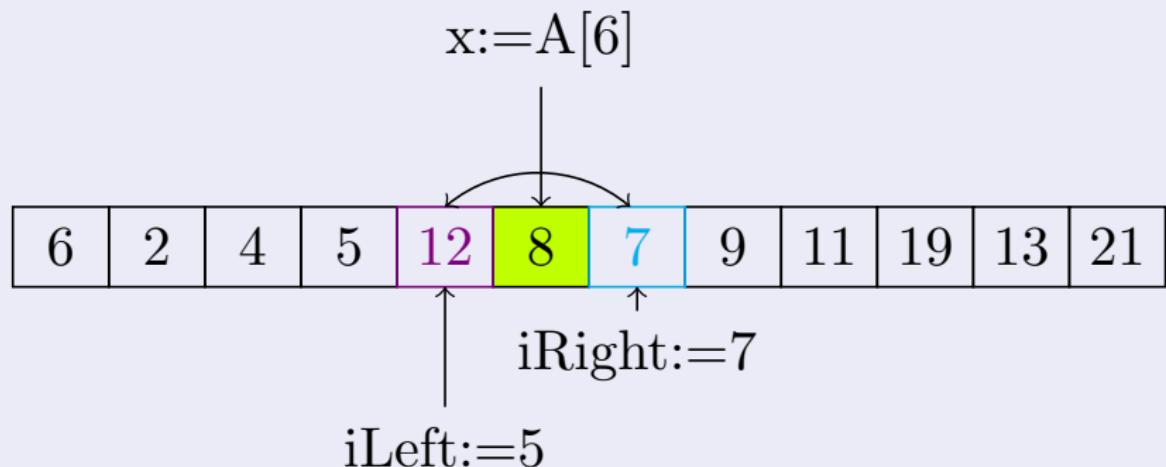


Figure : (3)

Trebuie să interschimbăm $A[5]$ cu $A[7]$.

Exemplu:

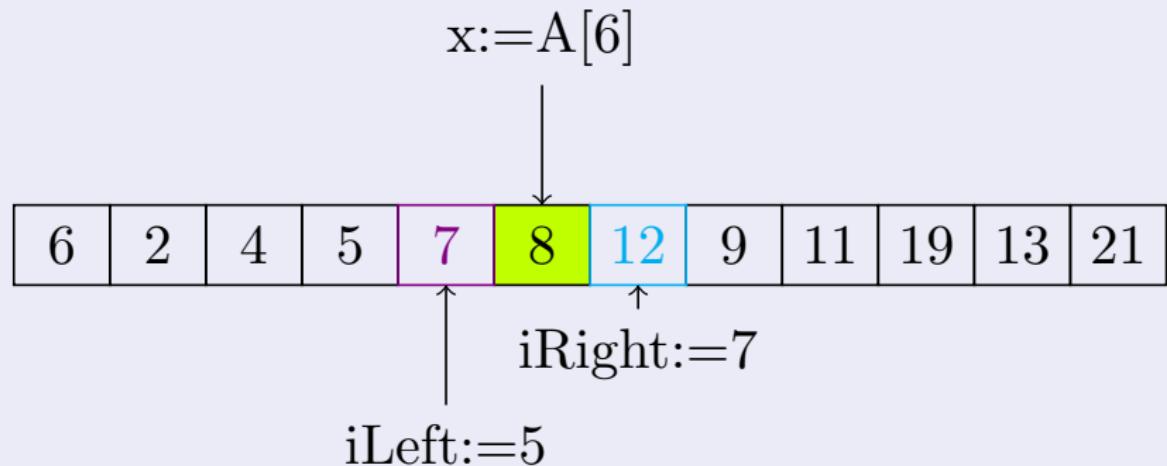


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

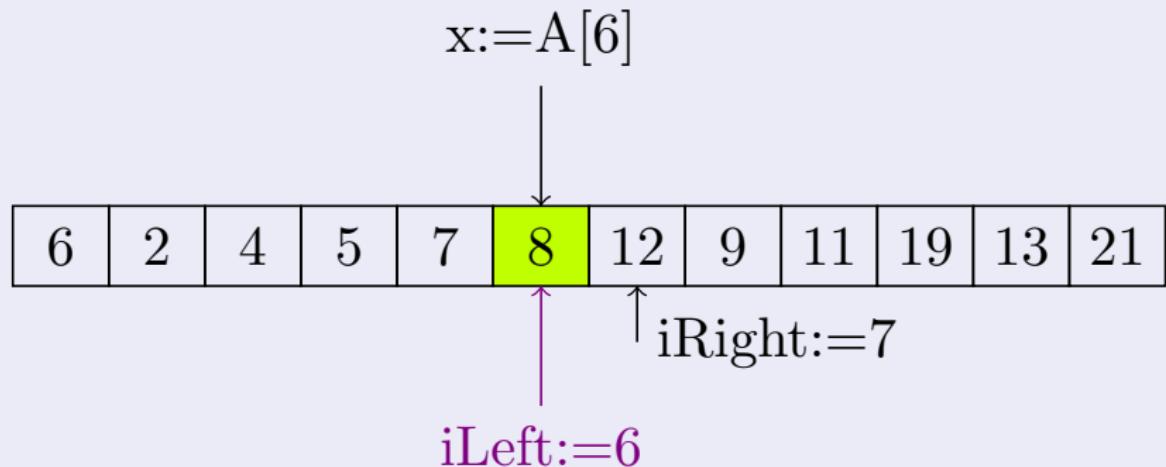


Figure : (3)

$iLeft := iLeft + 1;$

Exemplu:

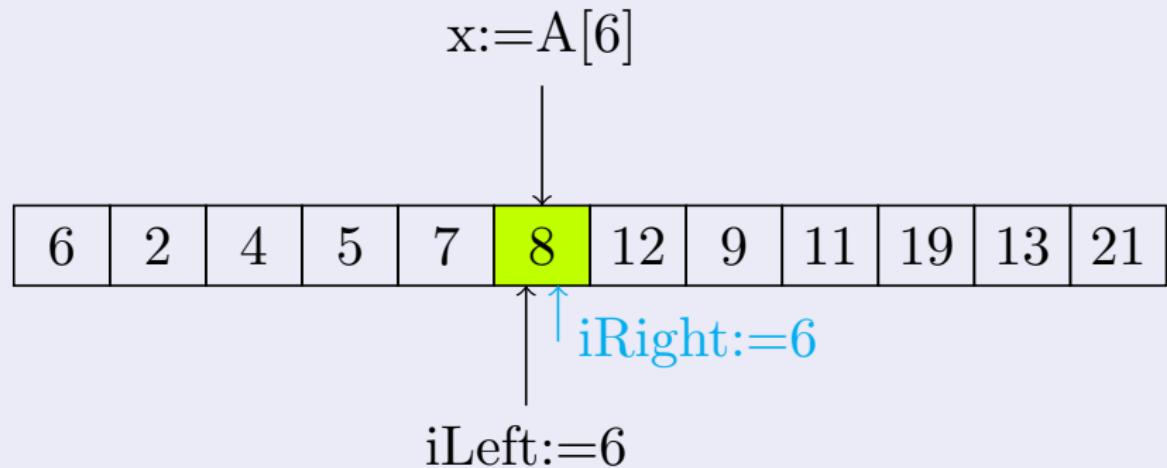


Figure : (3)

$iRight := iRight - 1;$

Exemplu:

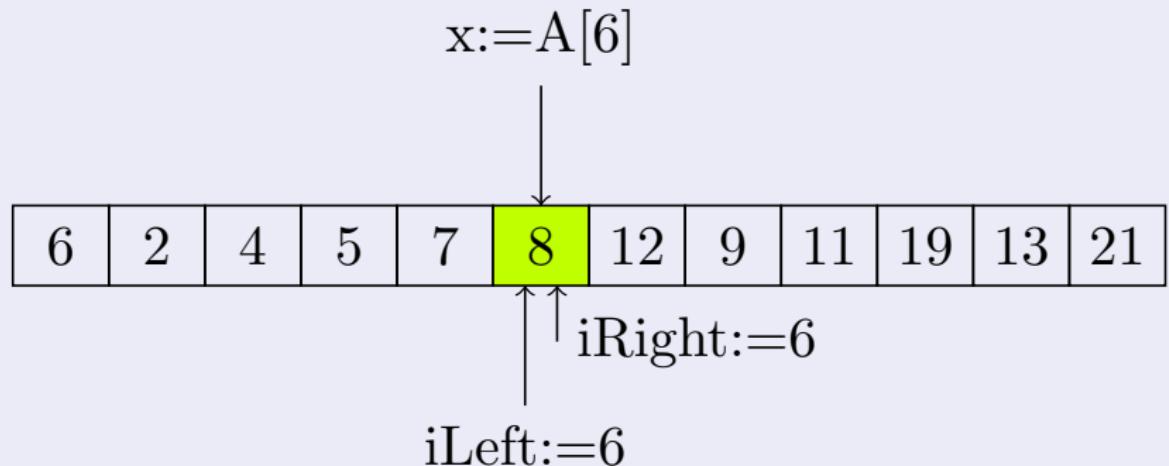


Figure : Vectorul după a patra iteratie a ciclului repeat

Avem $6 = iLeft \leq iRight = 6$, deci continuăm procedura de partitie.

Exemplu:

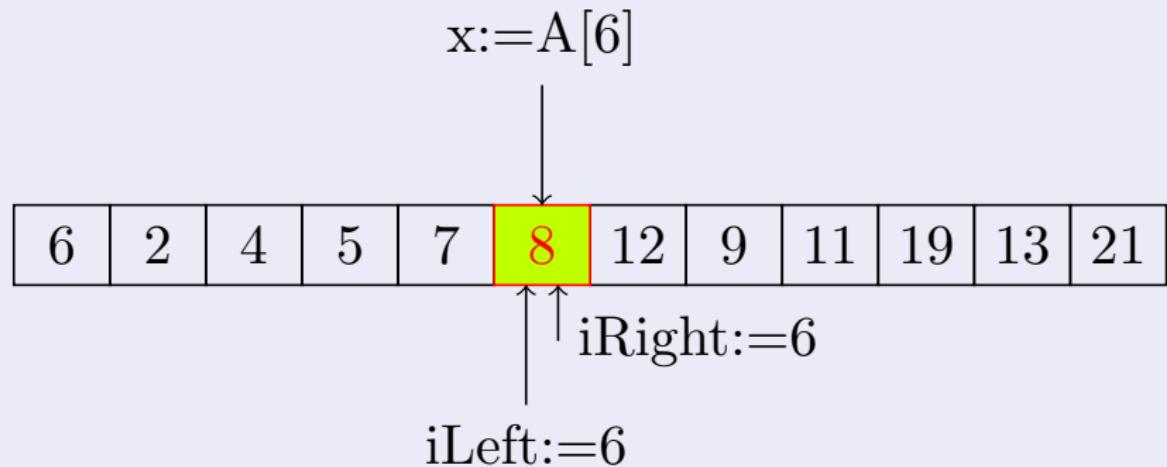


Figure : Vectorul după a patra iterație a ciclului repeat

$A[6] \geq x$, deci nu putem avansa cu $iLeft$.

Exemplu:

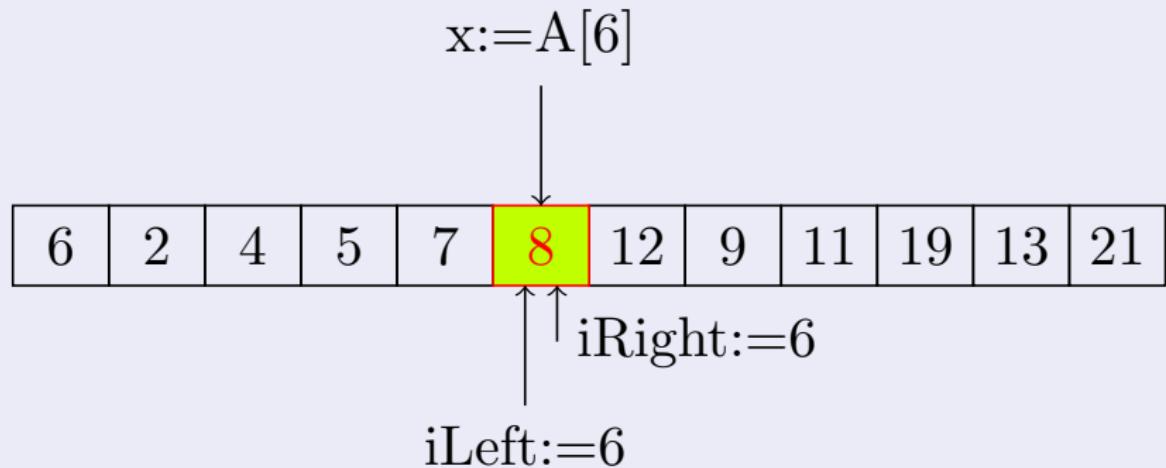


Figure : Vectorul după a patra iteratie a ciclului repeat

$A[6] \leq x$, deci nu putem avansa cu $iRight$.

Exemplu:

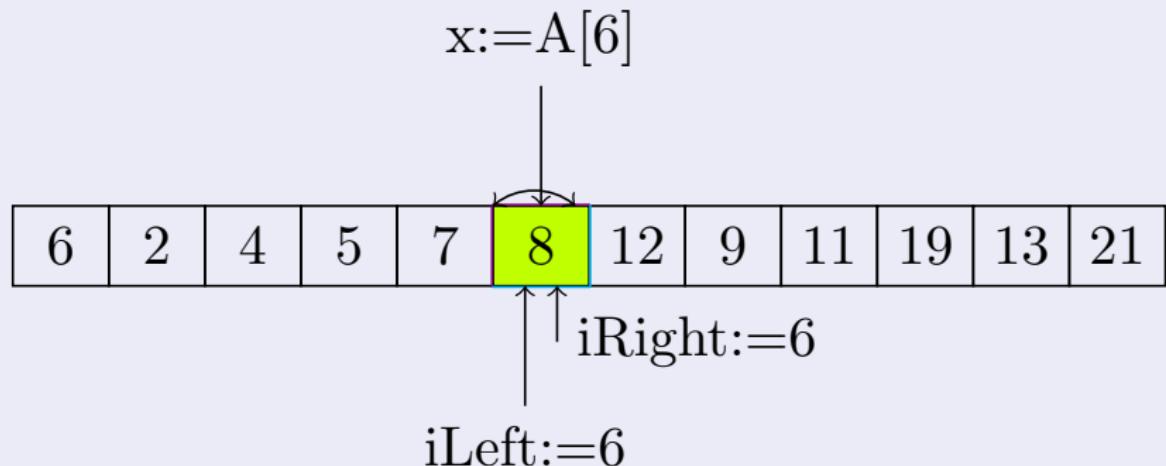


Figure : Vectorul după a patra iteratie a ciclului repeat

Trebuie să interschimbăm $A[6]$ cu $A[6]$.

Exemplu:

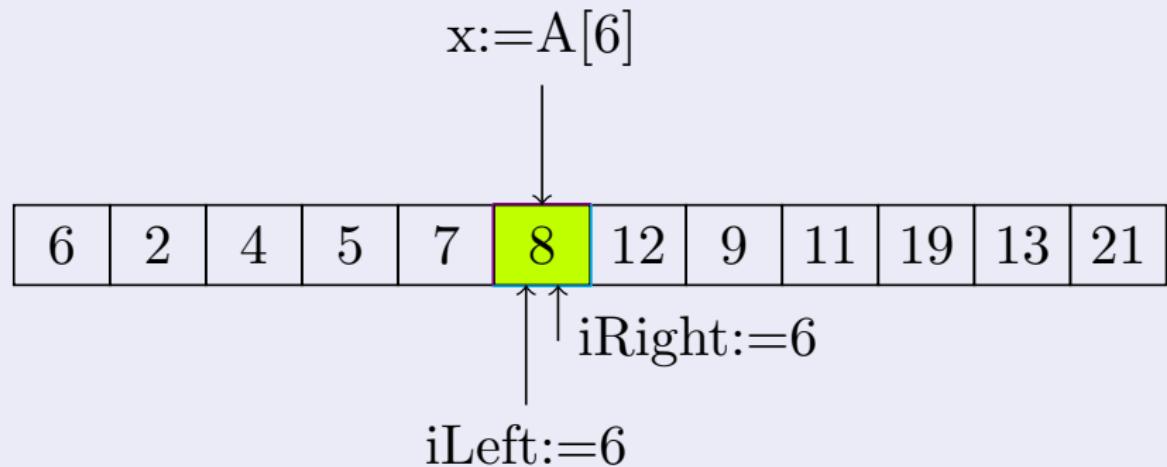


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu:

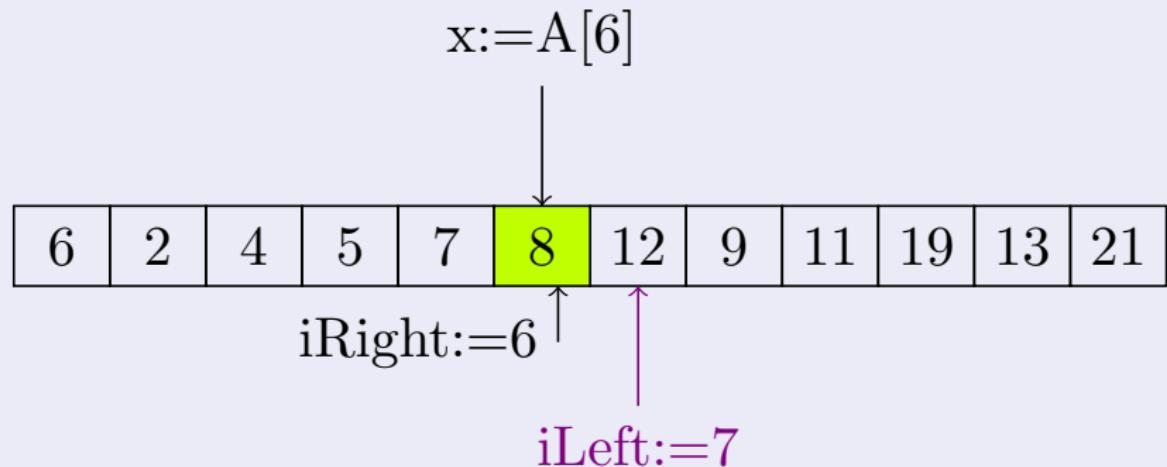


Figure : (3)

$iLeft := iLeft + 1;$

Exemplu:

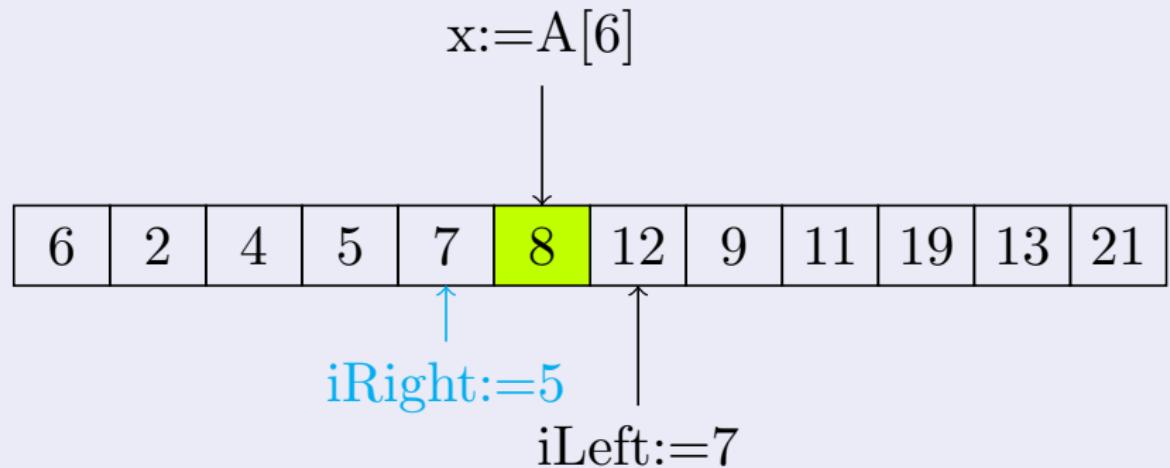


Figure : (3)

$iRight := iRight - 1;$

Exemplu:

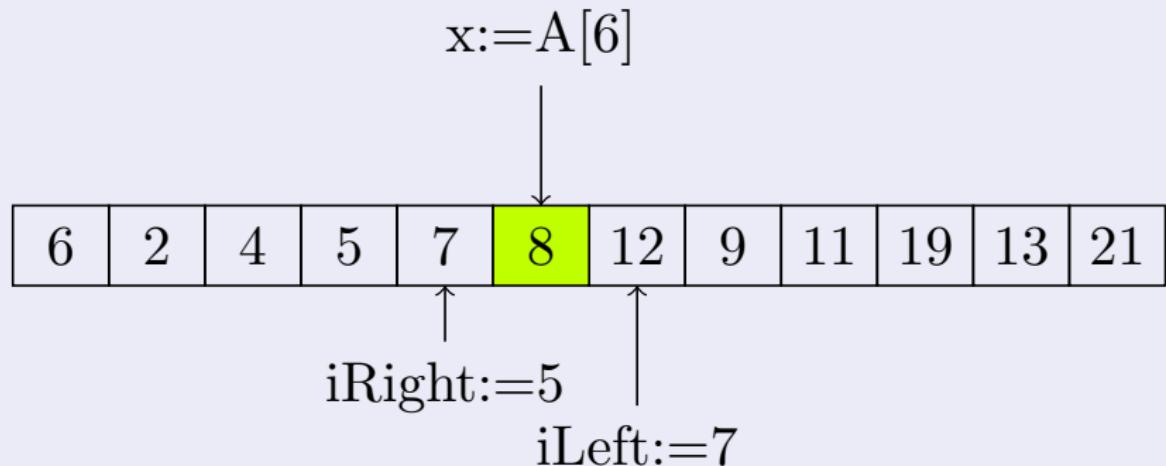


Figure : Vectorul după a patra iteratie a ciclului repeat

Avem $7 = iLeft > iRight = 5$, deci încheiem procedura de partitie.

Exemplu:



Figure : Final

Am obținut partitia: $A[1..5]$, $A[7..12]$.

Ce se întâmplă dacă vectorul conține chei multiple?

Un alt exemplu..

Exemplu pentru chei multiple:

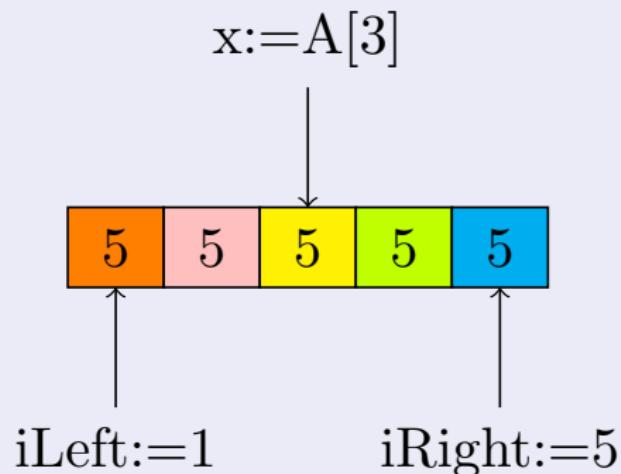


Figure : vectorul A inițial

```
x := A[(Left+Right)div 2];  
iLeft := 1; iRight = 5;
```

Exemplu pentru chei multiple:

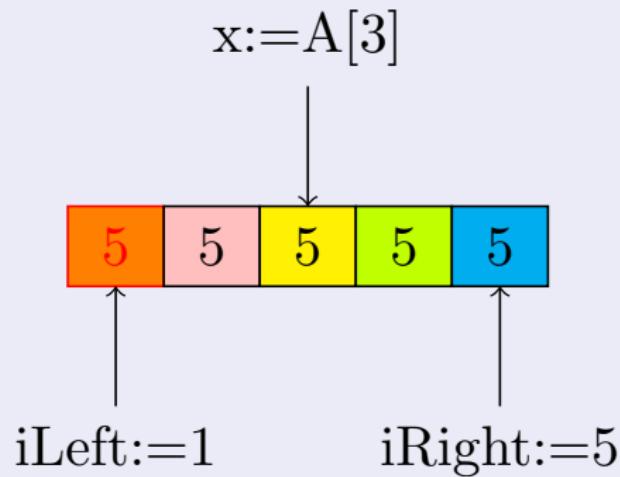


Figure : (2a)

$A[1] \geq x$, deci nu putem avansa cu $iLeft$.

Exemplu pentru chei multiple:

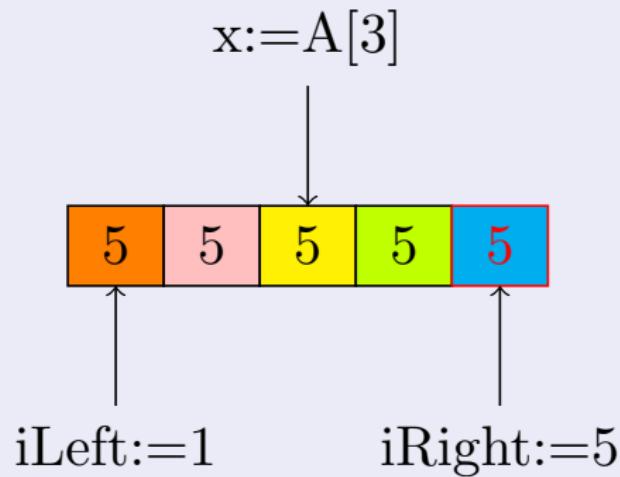


Figure : (2b)

$A[5] \leq x$, deci nu putem avansa cu $iRight$.

Exemplu pentru chei multiple:

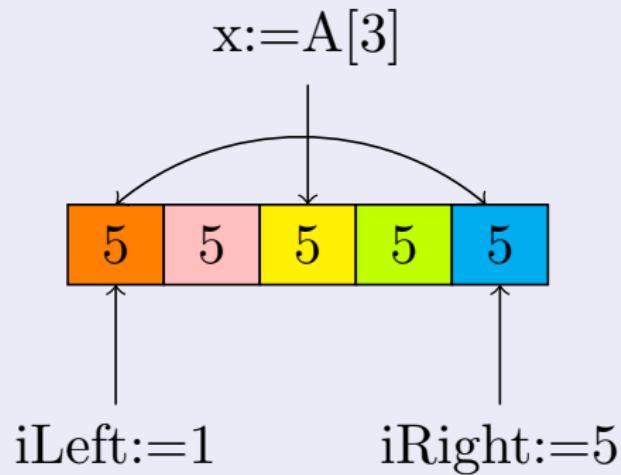


Figure : (2b)

Trebuie să interschimbăm $A[1]$ și $A[5]$.

Exemplu pentru chei multiple:

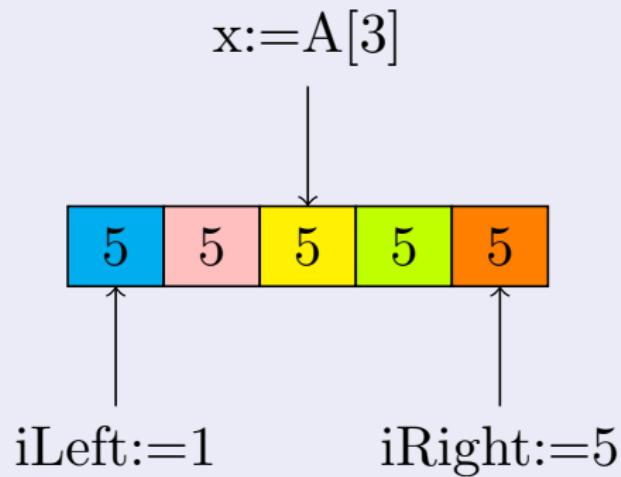


Figure : (3)

Vectorul obținut după interschimbare.

Exemplu pentru chei multiple:

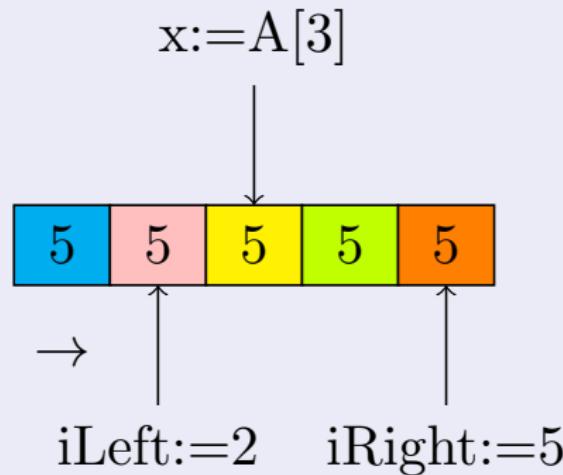


Figure : (3)

$iLeft := iLeft + 1;$

Exemplu pentru chei multiple:

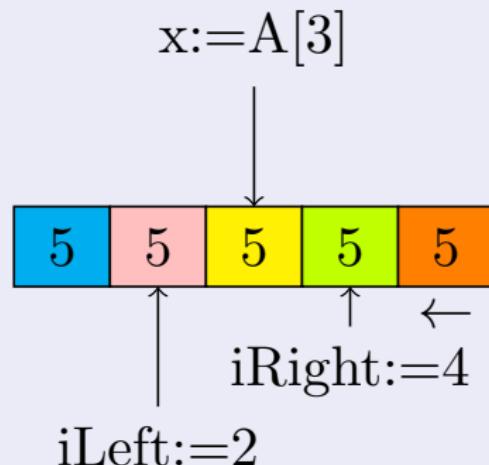


Figure : (3)

$iRight := iRight - 1;$

Exemplu pentru chei multiple:

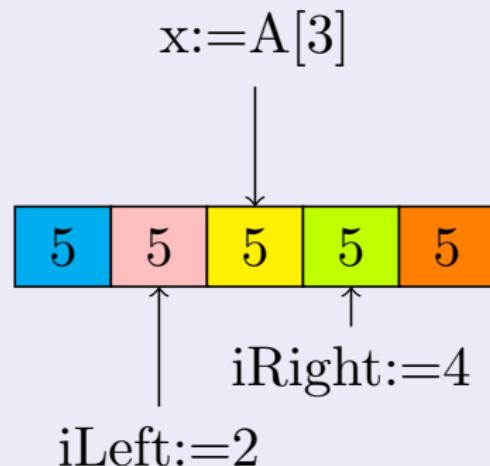


Figure : (3)

Avem $2 = iLeft \leq iRight = 4$, deci continuăm procedura de partiție.

Exemplu pentru chei multiple:

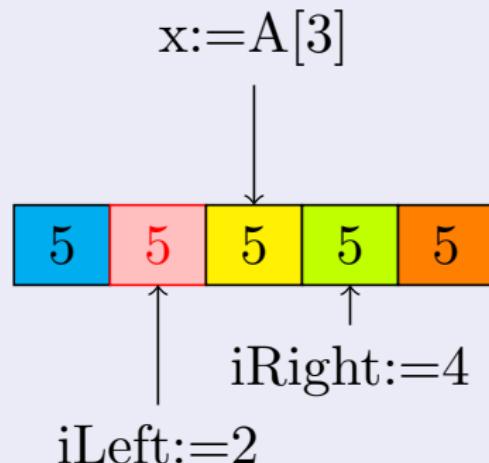


Figure : (3)

$A[2] \geq x$, deci nu putem avansa cu $iLeft$.

Exemplu pentru chei multiple:

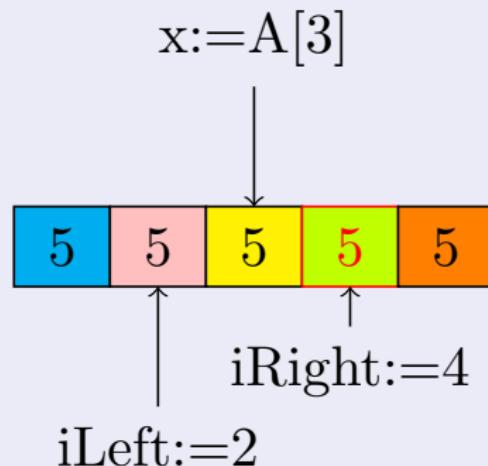


Figure : După prima iterare a ciclului repeat

$A[4] \leq x$, deci nu putem avansa cu $iRight$.

Exemplu pentru chei multiple:

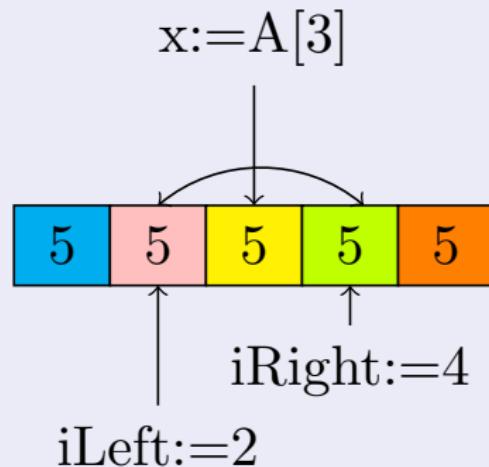


Figure : (2a)

Trebuie să interschimbăm $A[2]$ și $A[4]$.

Exemplu pentru chei multiple:

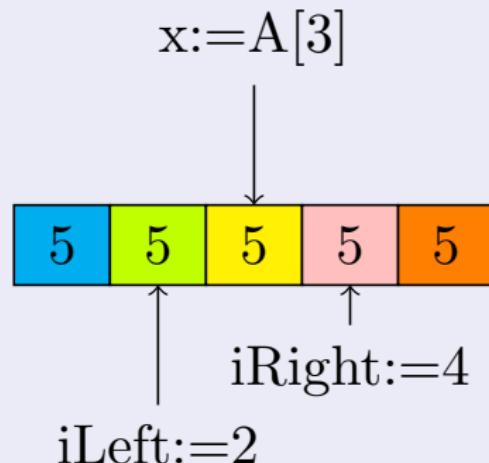


Figure : (2b)

Vectorul obținut după interschimbare.

Exemplu pentru chei multiple:

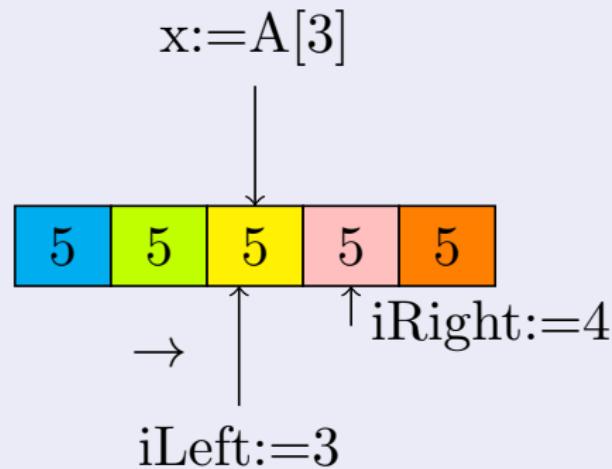


Figure : (3)

$iLeft := iLeft + 1;$

Exemplu pentru chei multiple:

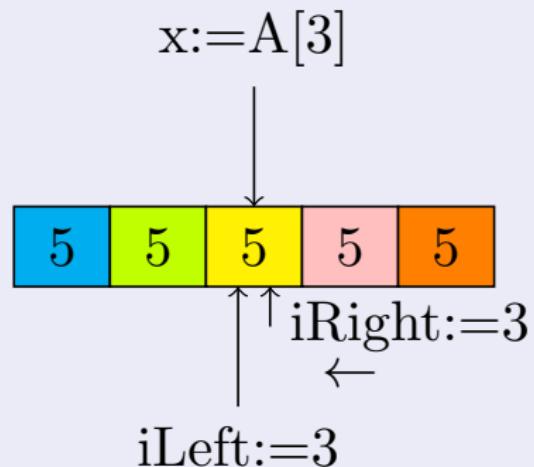


Figure : (3)

$iRight := iRight - 1;$

Exemplu pentru chei multiple:

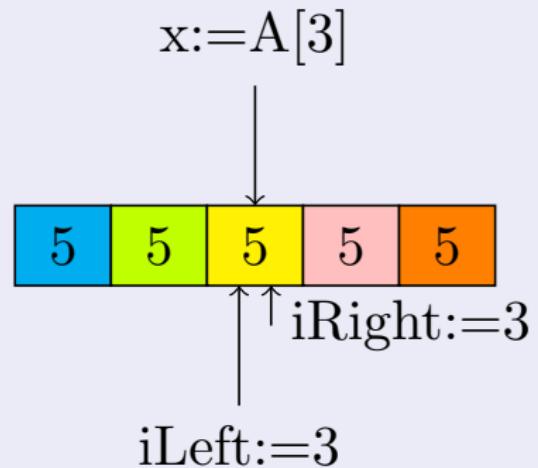


Figure : (3)

Avem $3 = iLeft \leq iRight = 3$, deci continuăm procedura de partiție.

Exemplu pentru chei multiple:

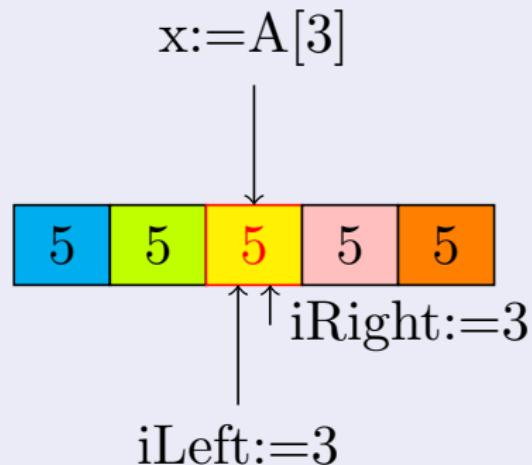


Figure : (3)

$A[3] \geq x$, deci nu putem avansa cu $iLeft$.

Exemplu pentru chei multiple:

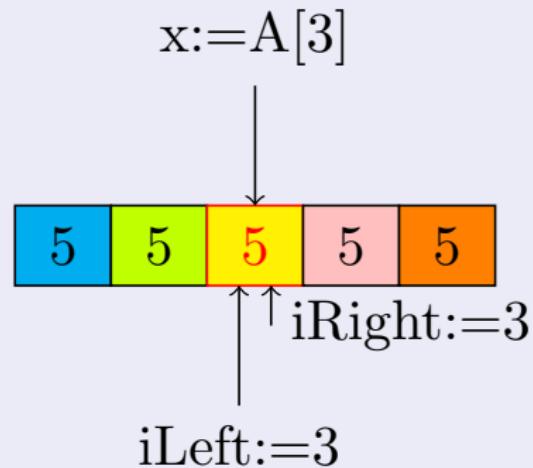


Figure : (Vectorul după a doua iteratie a ciclului repeat)

$A[3] \leq x$, deci nu putem avansa cu $iRight$.

Exemplu pentru chei multiple:

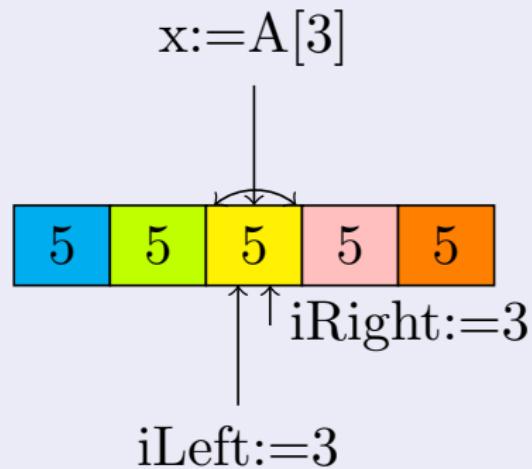


Figure : (2a)

Trebuie să interschimbăm $A[3]$ și $A[3]$.

Exemplu pentru chei multiple:

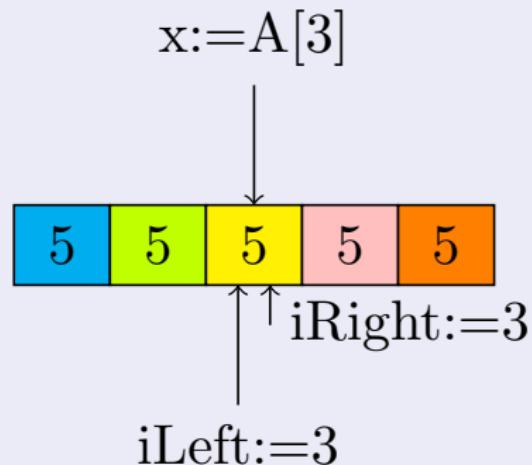


Figure : (2b)

Vectorul obținut după interschimbare.

Exemplu pentru chei multiple:

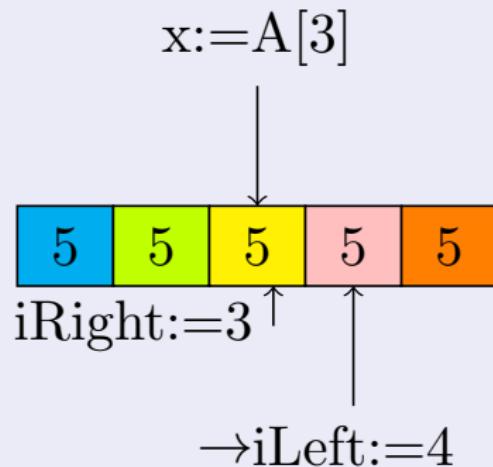


Figure : (2b)

iLeft := iLeft+1;

Exemplu pentru chei multiple:

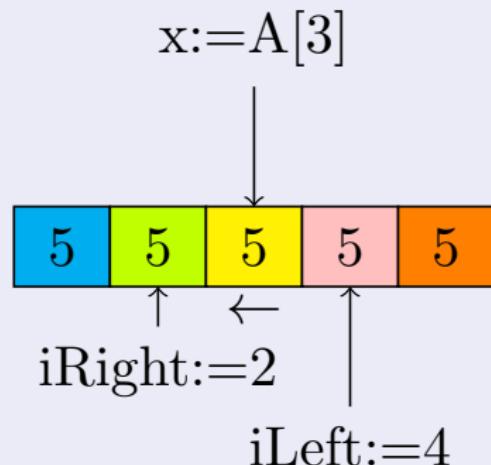


Figure : (2b)

$iRight := iRight - 1;$

Exemplu pentru chei multiple:

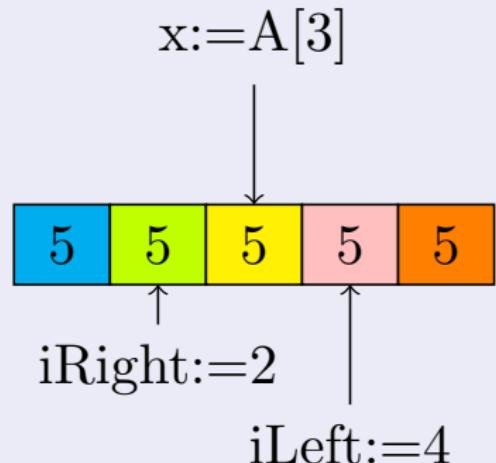


Figure : (3)

Avem $4 = iLeft > iRight = 2$, deci încheiem procedura de partitie.

Exemplu pentru chei multiple:

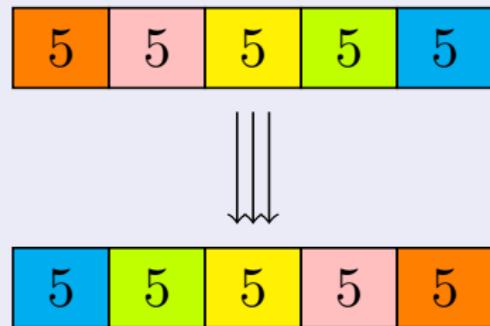


Figure : Final

Observăm că deși valorile erau egale, acestea și-au schimbat poziția.

Exemplu pentru chei multiple:



Figure : Final

Am obținut partitura: $A[1..2]$, $A[4..5]$.
Observăm că avem $iLeft < iRight$ și partitura dată:
 $A[Left..iRight]$ și $A[iLeft..Right]$.

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

- Să presupunem că, pentru partaționarea vectorului $A[Left..Right]$ alegem $x = A[Left]$.

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

- Să presupunem că, pentru partaționarea vectorului $A[Left..Right]$ alegem $x = A[Left]$.
- Atunci, pasul (2a) din algoritmul de partaționare (parcurserea de la stânga la dreapta a vectorului până la primul indice i pentru care $A[i] \geq x$) nu mai are sens, căci acest indice este chiar $Left$.

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

- Să presupunem că, pentru partaționarea vectorului $A[Left..Right]$ alegem $x = A[Left]$.
- Atunci, pasul (2a) din algoritmul de partaționare (parcurserea de la stânga la dreapta a vectorului până la primul indice i pentru care $A[i] \geq x$) nu mai are sens, căci acest indice este chiar $Left$.
- Executăm (2b), adică parcurgem de la dreapta la stânga vectorul până la primul indice j pentru care $A[j] \leq x$.

Partiția cu pivot într-o extremitate

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partiția. Ce se întâmplă dacă alegem o **valoare situată la una dintre extremități**?

- Să presupunem că, pentru partaționarea vectorului $A[Left..Right]$ alegem $x = A[Left]$.
- Atunci, pasul (2a) din algoritmul de partaționare (parcurserea de la stânga la dreapta a vectorului până la primul indice i pentru care $A[i] \geq x$) nu mai are sens, căci acest indice este chiar $Left$.
- Executăm (2b), adică parcurgem de la dreapta la stânga vectorul până la primul indice j pentru care $A[j] \leq x$.
- Pasul (3) devine:

```
if Left < j then  
    interschimbă (A[Left], A[j])
```

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.
- Procedeul continuă până când cele două parcurgeri se întâlnesc, adică atât timp cât mai există componente în vector care nu au fost comparate cu pivotul.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.
- Procedeul continuă până când cele două parcurgeri se întâlnesc, adică atât timp cât mai există componente în vector care nu au fost comparate cu pivotul.
- La sfârșit obținem valoarea pivot x plasată la locul ei final în vector.

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.
- Procedeul continuă până când cele două parcurgeri se întâlnesc, adică atât timp cât mai există componente în vector care nu au fost comparate cu pivotul.
- La sfârșit obținem valoarea pivot x plasată la locul ei final în vector.
- Fie loc indicele lui A ce va conține pe x . Avem atunci:
$$A[k] \leq x \quad \forall k \in [1..loc - 1]$$
$$A[k] \geq x \quad \forall k \in [loc + 1..n]$$

- Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$.
- Reluăm acum parcurgerea de tip (2a) de la stânga la dreapta. Parcurgerea de tip (2b) nu mai are sens.
- La sfârșit interschimbăm. Observăm că valoarea pivot participă din nou la interschimbare.
- Procedeul continuă până când cele două parcurgeri se întâlnesc, adică atât timp cât mai există componente în vector care nu au fost comparate cu pivotul.
- La sfârșit obținem valoarea pivot x plasată la locul ei final în vector.
- Fie loc indicele lui A ce va conține pe x . Avem atunci:
$$A[k] \leq x \quad \forall k \in [1..loc - 1]$$
$$A[k] \geq x \quad \forall k \in [loc + 1..n]$$
- Deci subintervalele ce trebuie procesate în continuare sunt $A[1..loc - 1]$ și $A[loc + 1..n]$.

Procedura Partition2

Dăm mai jos noua procedură de partiționare.

- La parcgeri ea va lăsa pe loc valorile egale cu pivotul.
- Indicele *loc* ține minte tot timpul componenta pe care se află valoarea pivot.
- La sfârșit o transmite procedurii apelante pentru a putea fi calculate noile capete ale subvectorilor rezultați.

```

procedure Partition2 (Left, Right, loc)
    // loc este indicele pe care se va plasa în final valoarea  $x = A[Left]$ 
    // inițializarea indicilor  $i$  și  $j$  pentru parcurgerile de la stânga la dreapta, respectiv de la
    dreapta la stânga
    i:= Left; j:= Right;
    loc:= Left                                         // inițializarea indicelui loc
    while  $i < j$  do
        // parcurgerea de la dreapta la stânga, urmată de interschimbare
        while ( $A[loc] \leq A[j]$ ) and ( $j \neq loc$ ) do
            j:= j-1
        endwhile
        if  $A[loc] > A[j]$  then
            Interschimbă( $A[loc]$ ,  $A[j]$ )
            loc:= j
        endif
        // parcurgerea de la stânga la dreapta, urmată de interschimbare
        while ( $A[i] \leq A[loc]$ ) and ( $i \neq loc$ ) do
            i:= i +1
        endwhile
        if  $A[i] > A[loc]$  then
            Interschimbă ( $A[loc]$ ,  $A[i]$ )
            loc:= i
        endif
    endwhile
endproc

```

Avantaje față de Partition2

- Pivot într-o extremitate, dar parurge restul, făcând **separarea valorilor, într-un singur sens.**

Partiția Lomuto

Avantaje față de Partition2

- Pivot într-o extremitate, dar parurge restul, făcând **separarea valorilor, într-un singur sens.**
- Reducerea numărului de interschimbări: Lomuto face $m + 1$, față de $2m$ făcute de Partition2.

Partiția Lomuto

```
procedure PartitionLomuto(Left, Right, loc)
    x:=A[Right];                                // algearea pivotului în intr-o extremitate
    // loc este indicele pe care se va plasa în final valoarea x = A[Right]
    loc:=Left-1;
    // inițializarea indicelui j pentru parcurgerea de la stânga la dreapta (un
    singur sens)
    j:= Left;
    while j ≤ Right do
        if A[j] ≤ x then
            loc:=loc+1;
            Interschimbă(A[loc],A[j]);
        endif
        j:=j+1;
    endwhile
    if loc ≥ Right then
        loc:=loc-1;
    endif
endproc
```

Exemplu:

Left:=1

Right:=12

x:=A[12]

loc:=0

j:=1

13	19	9	5	12	21	7	4	11	2	6	8
----	----	---	---	----	----	---	---	----	---	---	---

```
x:=A[Right];  
loc:=Left-1; j:= Left
```

Exemplu:

Left:=1

13

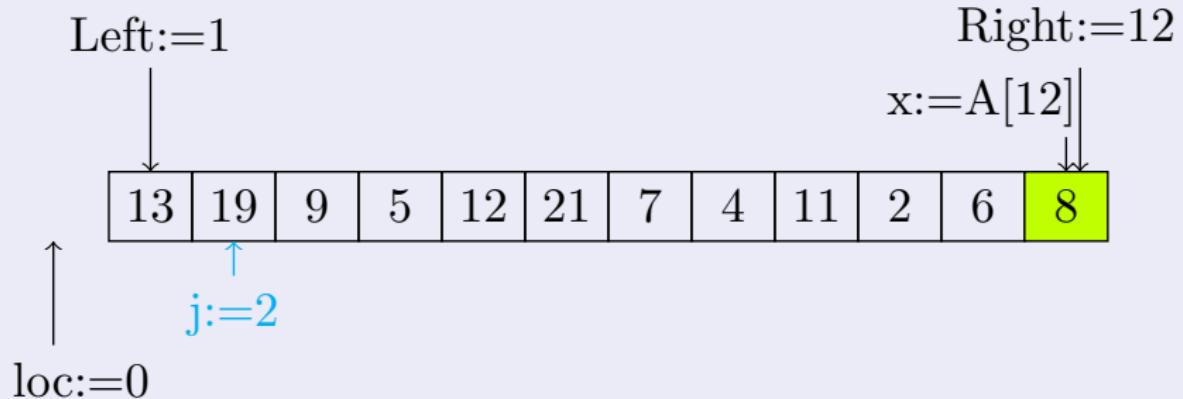
Right:=12

x:=A

↑
j:=
loc:=0

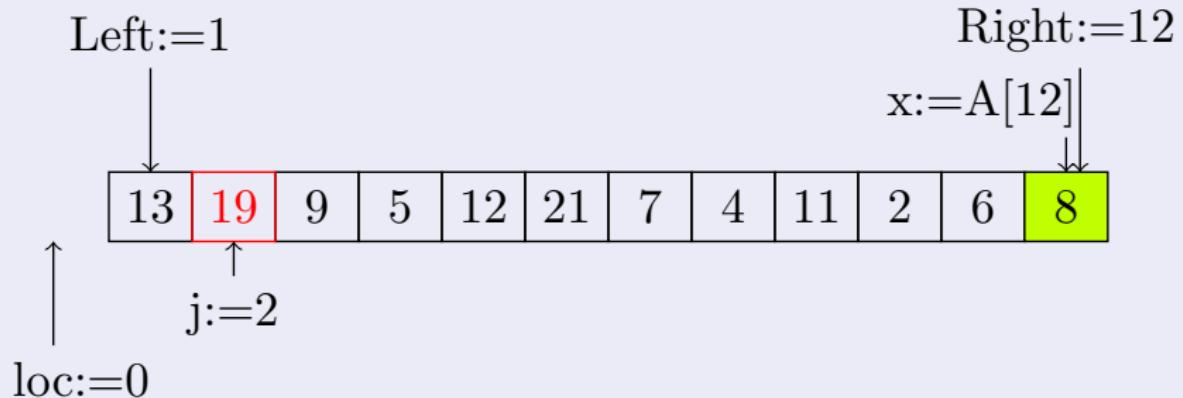
$A[1] > x$.

Exemplu:



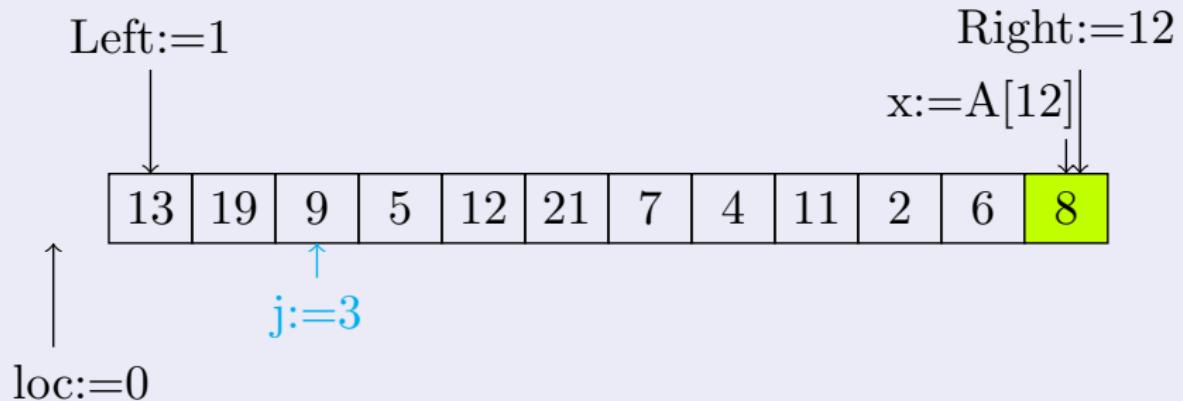
Avansam cu j .

Exemplu:



$A[2] > x.$

Exemplu:



Avansam cu j.

Exemplu:

Left:=1

Right:=12

x:=A[12]

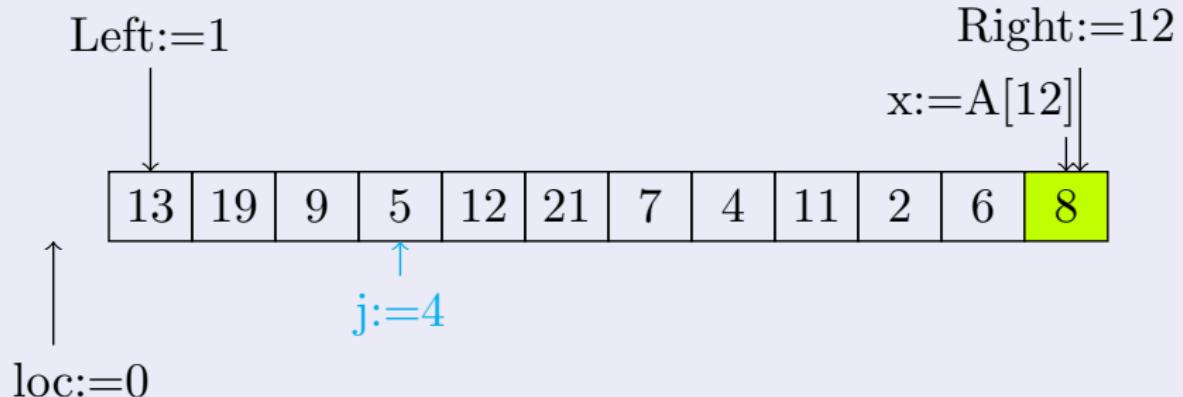


j:=3

loc:=0

$A[3] > x$.

Exemplu:



Avansam cu j .

Exemplu:

Left:=1

1

↑
loc:=0

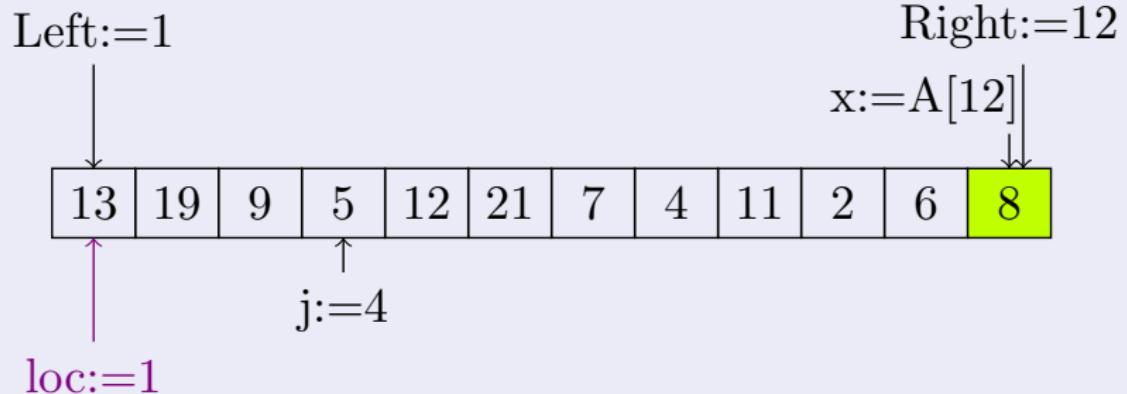
j:=4

Right:=12

x:=A[12]

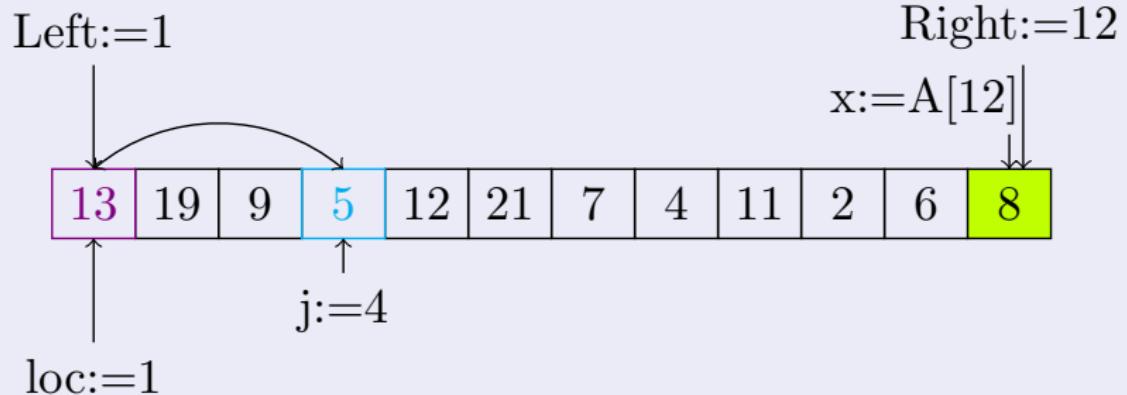
A[4] ≤ x.

Exemplu:



Avansăm cu loc.

Exemplu:



Trebuie să interschimbăm $A[1]$ cu $A[4]$.

Exemplu:

Left:=1



Right:=12

x:=

A[12]



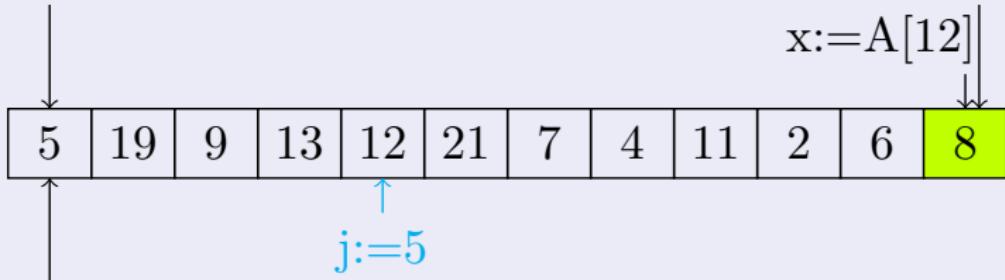
loc:=1

j:=4

Vectorul după interschimbare.

Exemplu:

Left:=1



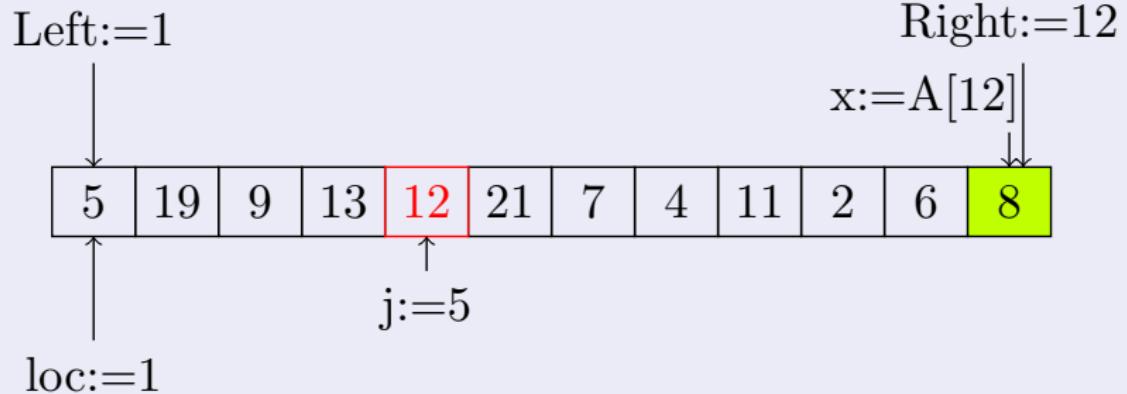
Right:=12

$x := A[12]$

loc:=1

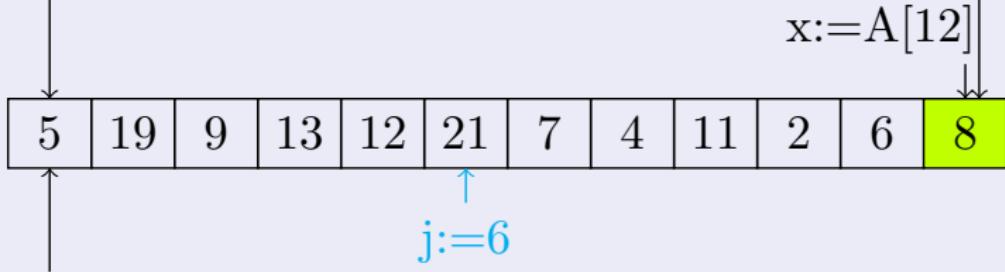
Avansăm cu j .

Exemplu:



Exemplu:

Left:=1



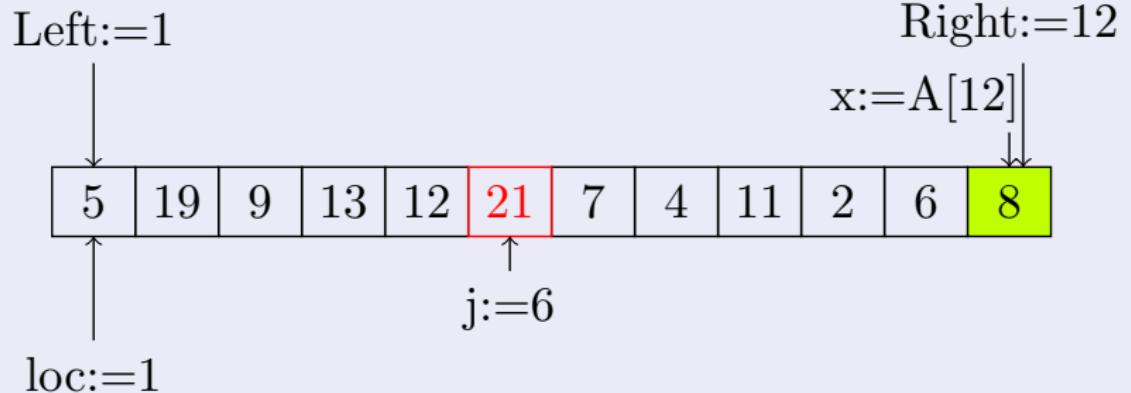
Right:=12

$x := A[12]$

loc:=1

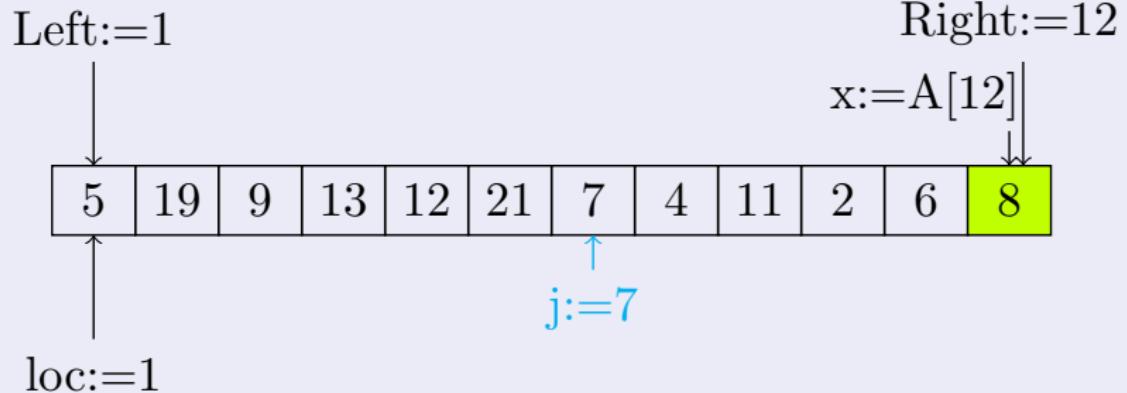
Avansăm cu j .

Exemplu:



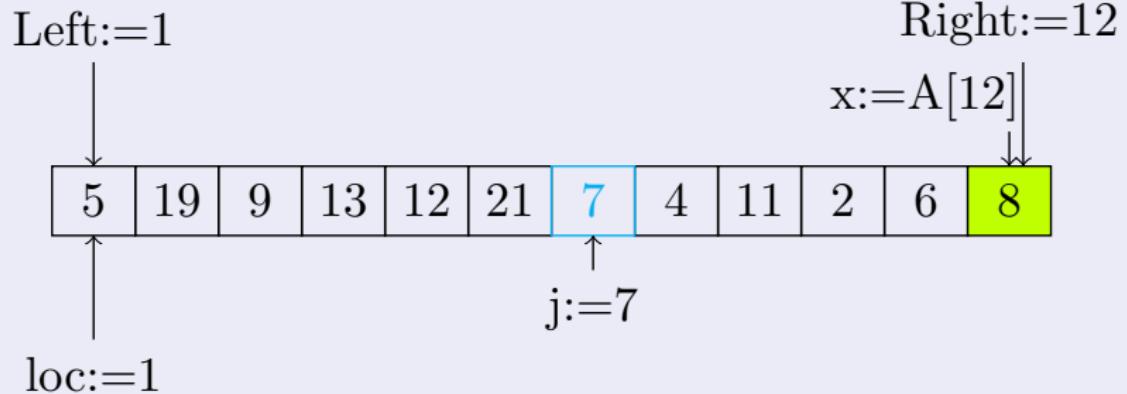
A[6] > x.

Exemplu:



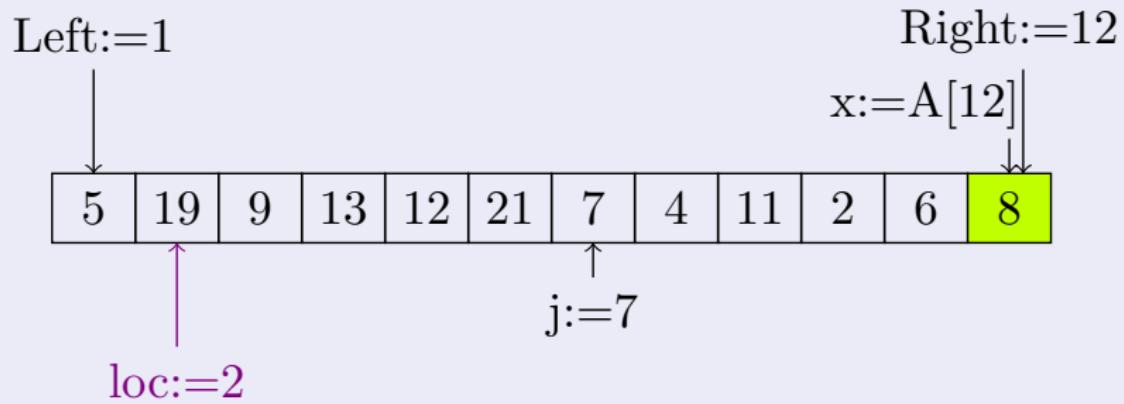
Avansăm cu j .

Exemplu:



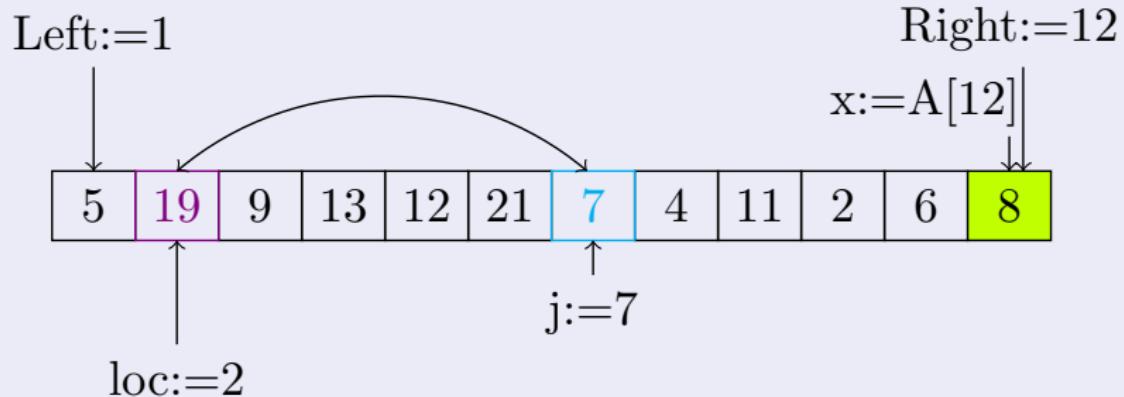
$$A[7] \leq x.$$

Exemplu:



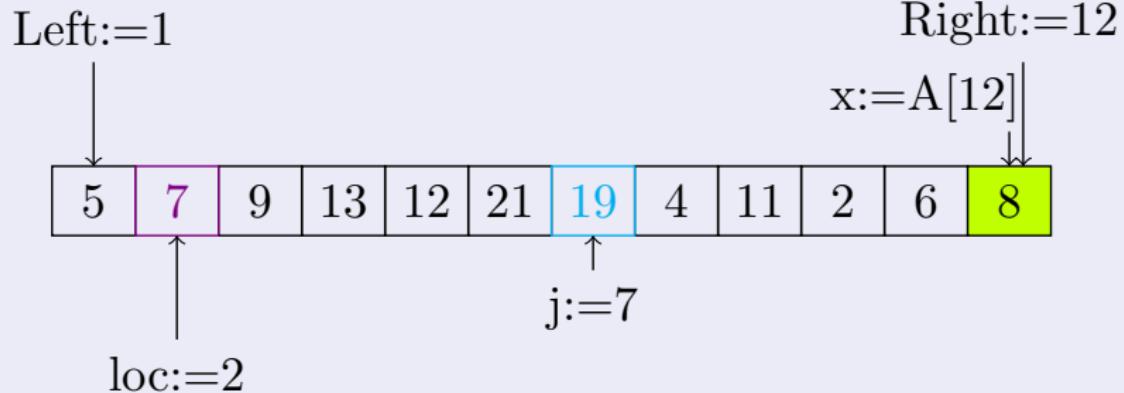
Avansăm cu loc.

Exemplu:



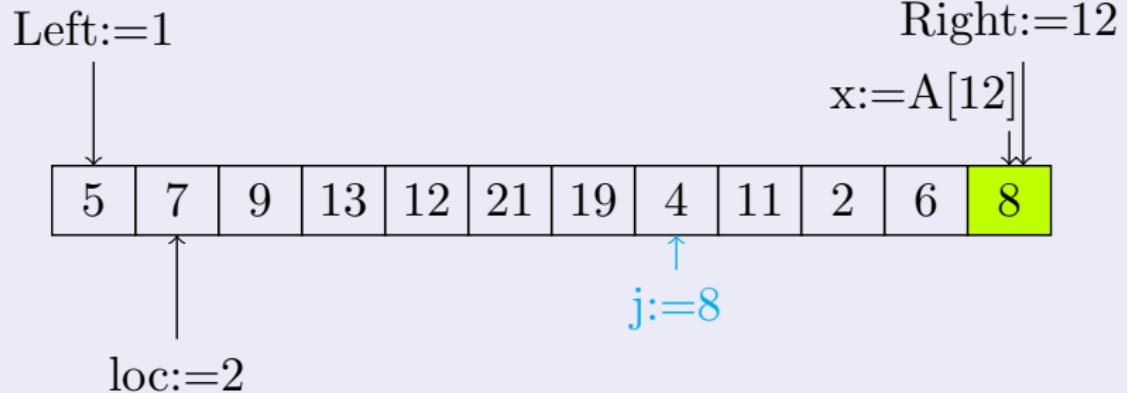
Trebuie să interschimbăm $A[2]$ cu $A[7]$.

Exemplu:



Vectorul după interschimbare.

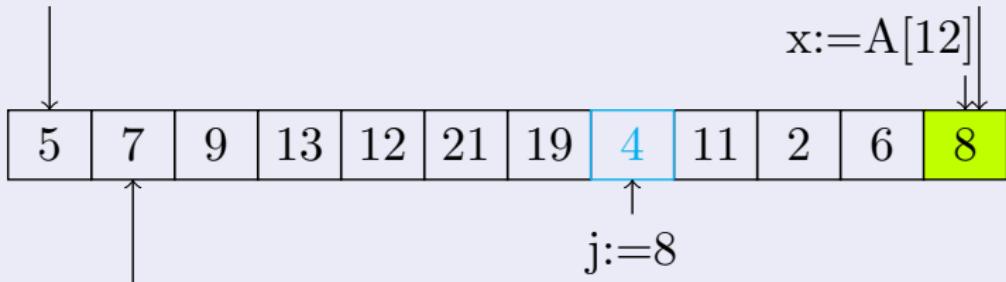
Exemplu:



Avansăm cu j.

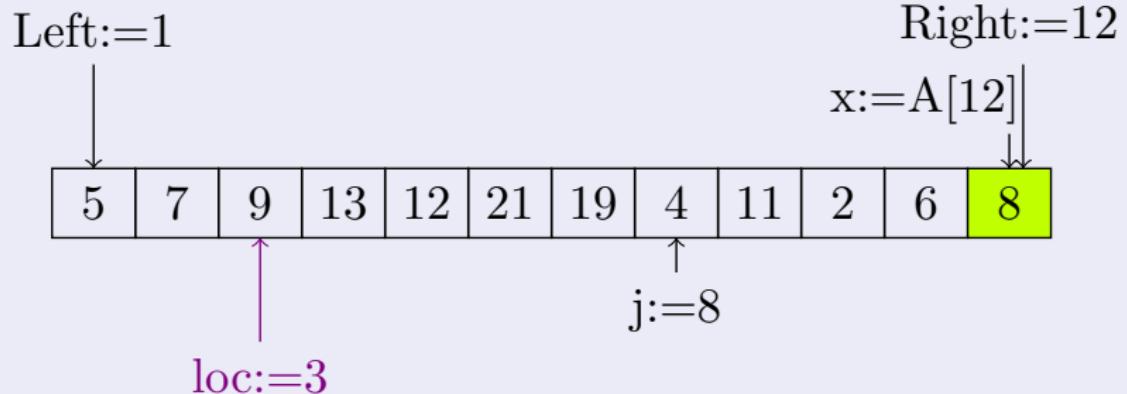
Exemplu:

Left:=1



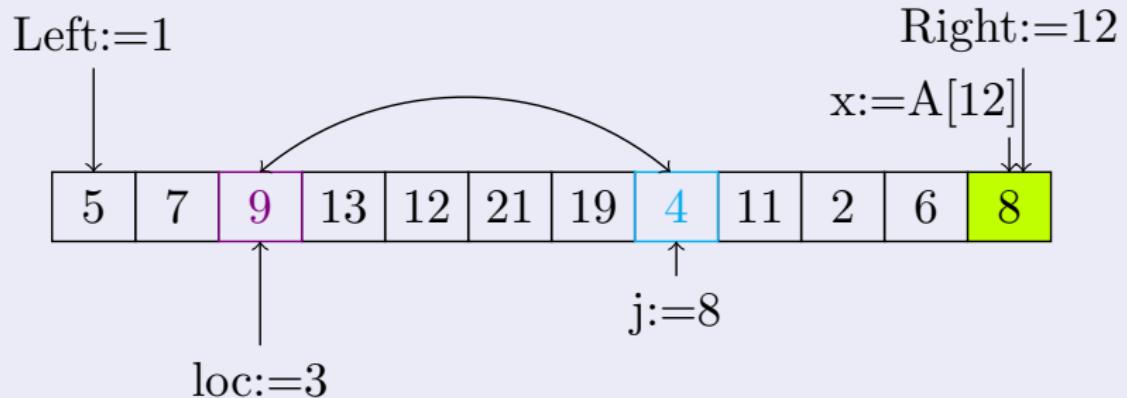
$$A[8] \leq x.$$

Exemplu:



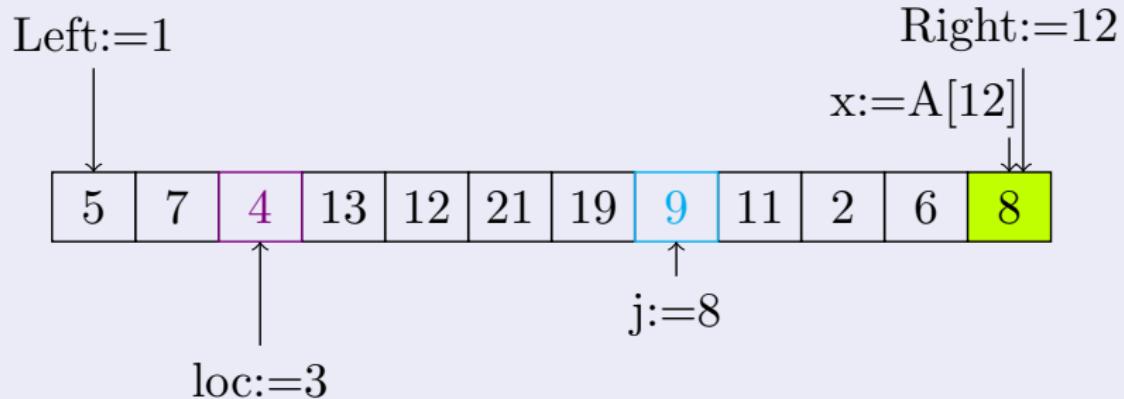
Avansăm cu loc.

Exemplu:



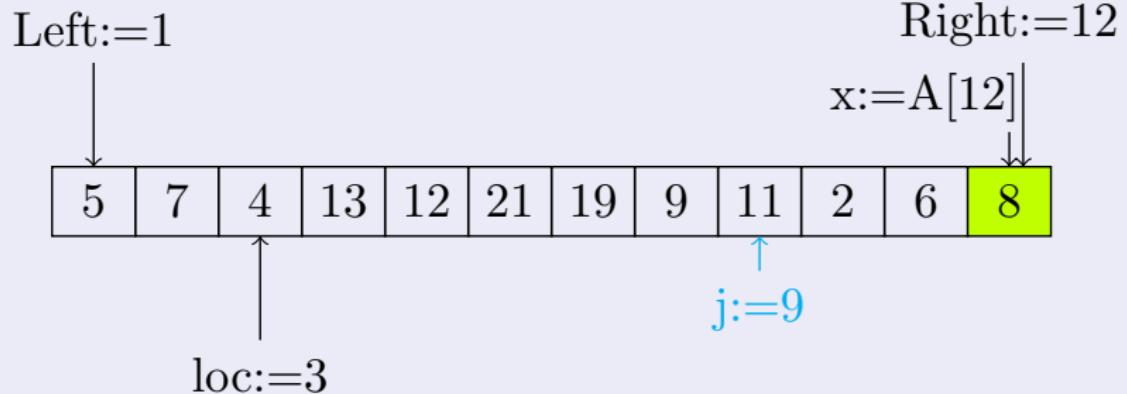
Trebuie să interschimbăm $A[3]$ cu $A[8]$.

Exemplu:



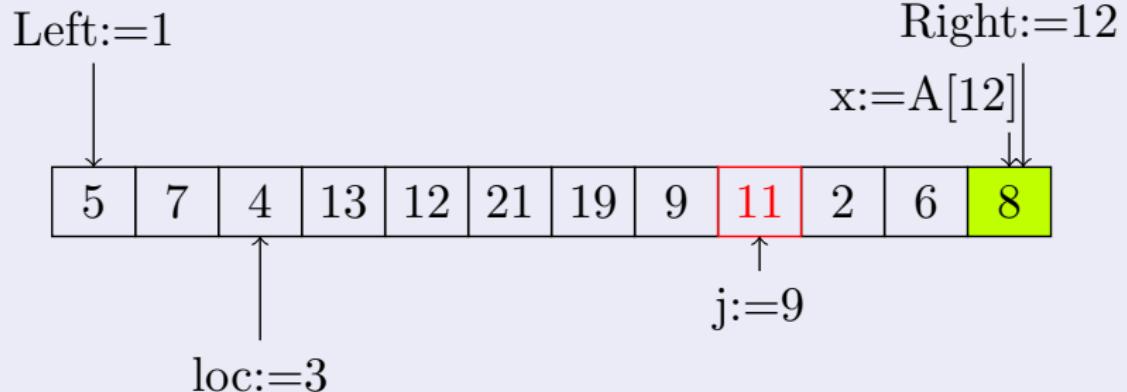
Vectorul după interschimbare.

Exemplu:

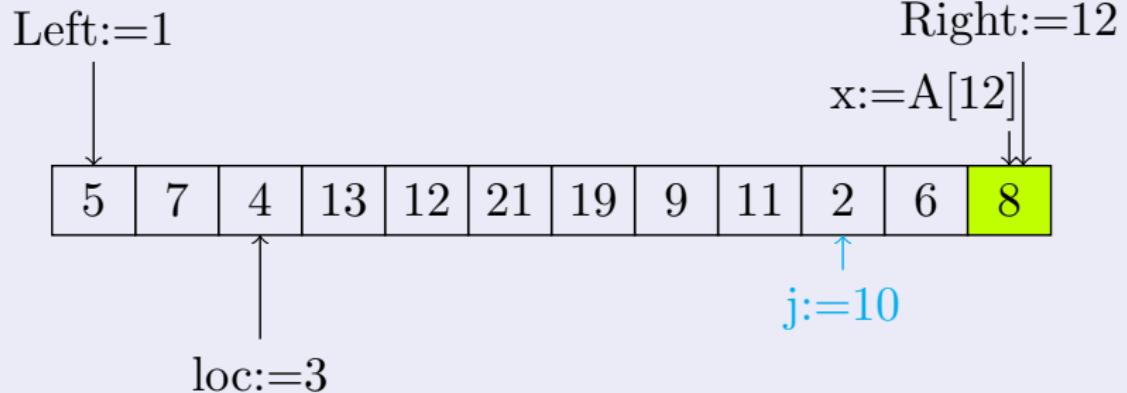


Avansăm cu j .

Exemplu:

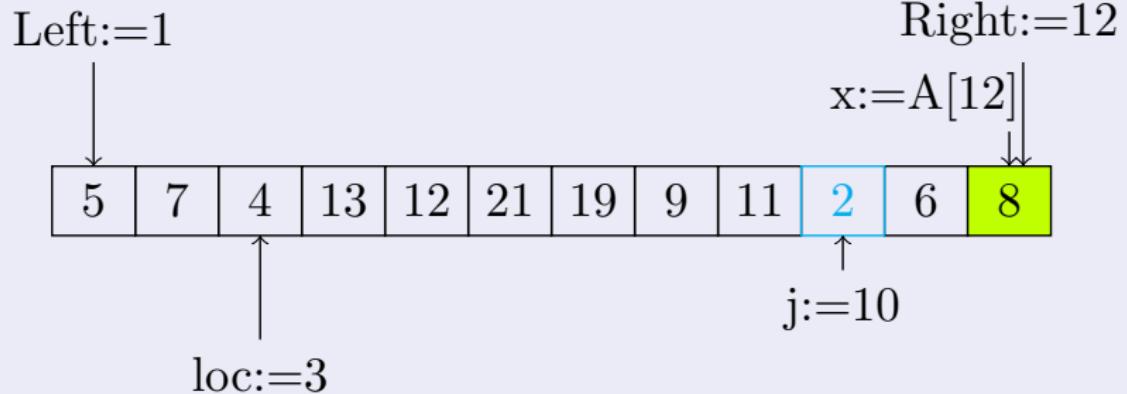


Exemplu:



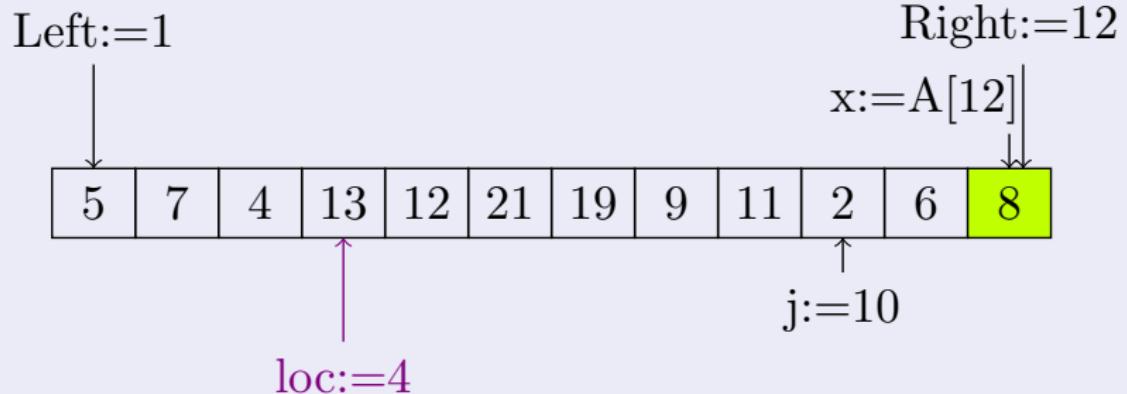
Avansăm cu j .

Exemplu:



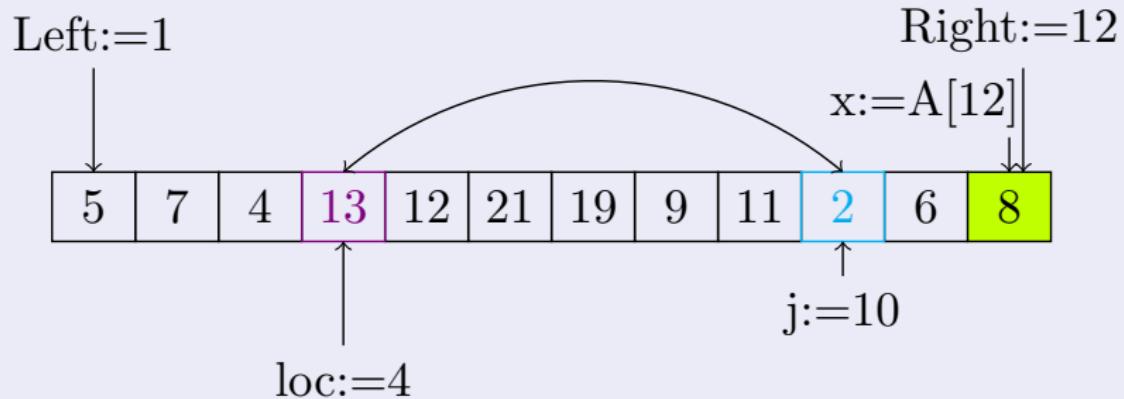
$A[10] \leq x.$

Exemplu:



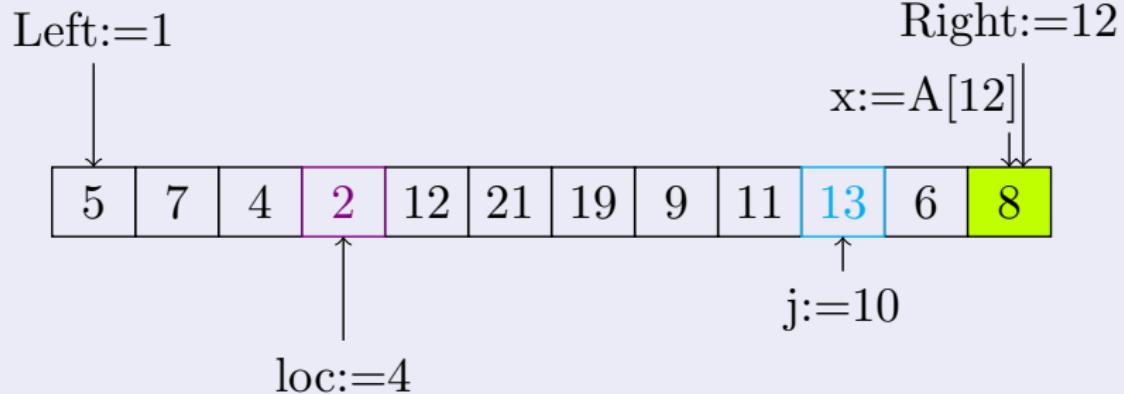
Avansăm cu loc.

Exemplu:



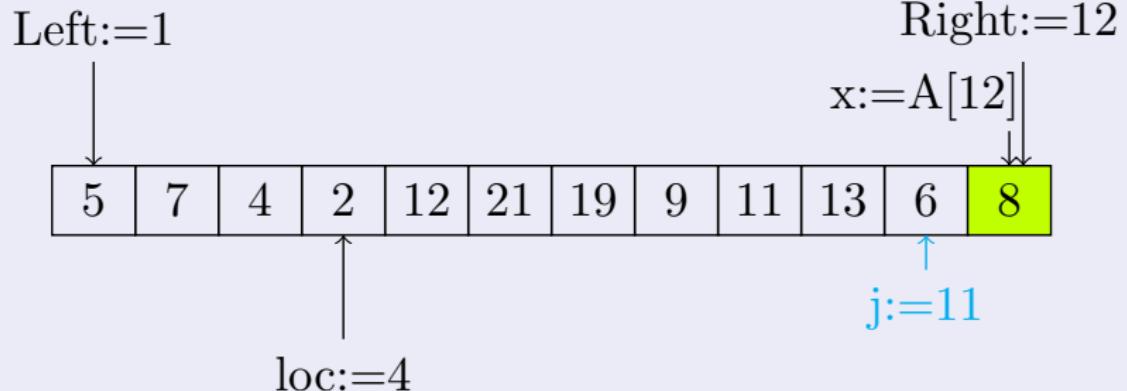
Trebuie să interschimbăm $A[4]$ cu $A[10]$.

Exemplu:



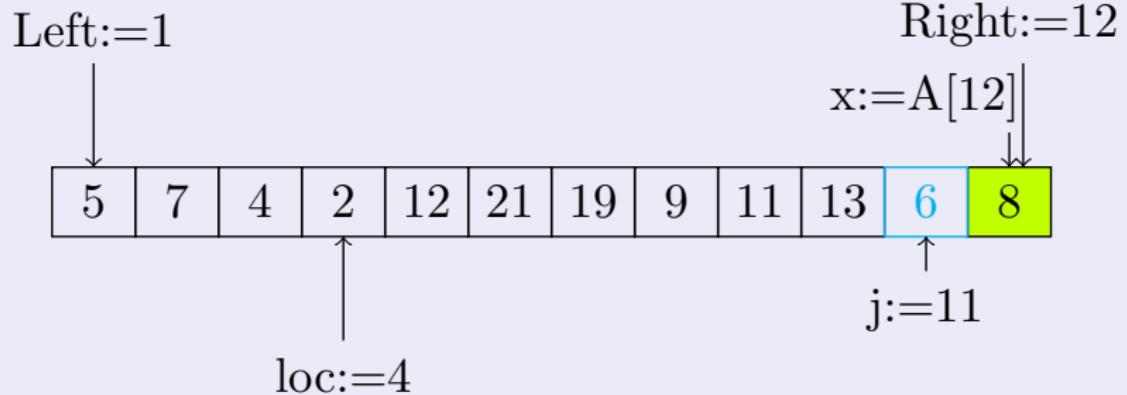
Vectorul după interschimbare.

Exemplu:



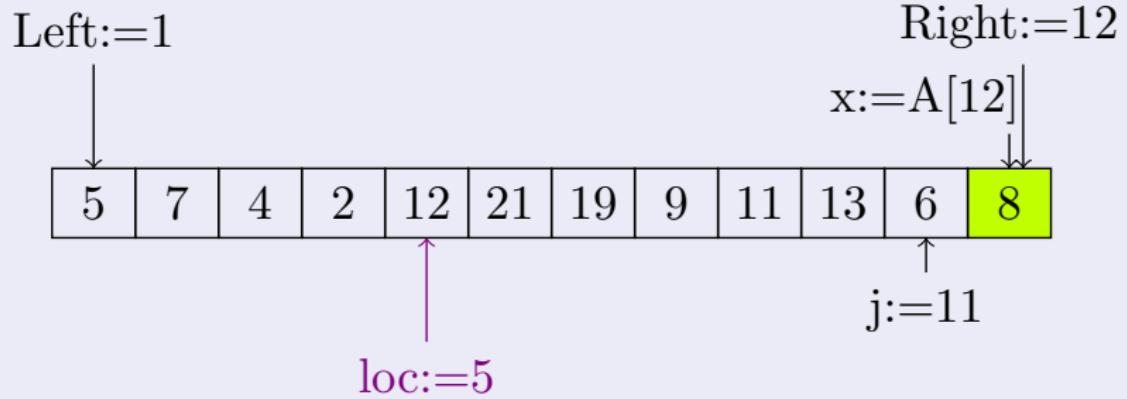
Avansăm cu j.

Exemplu:



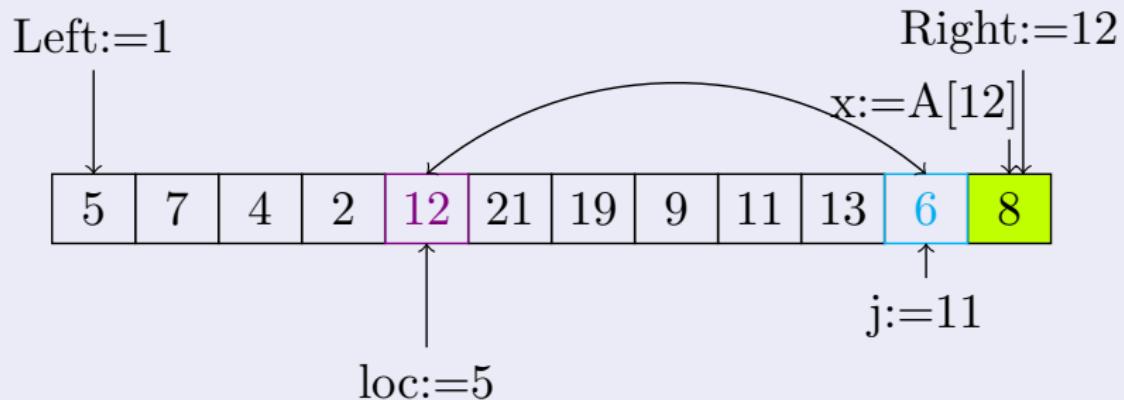
$$A[11] \leq x.$$

Exemplu:



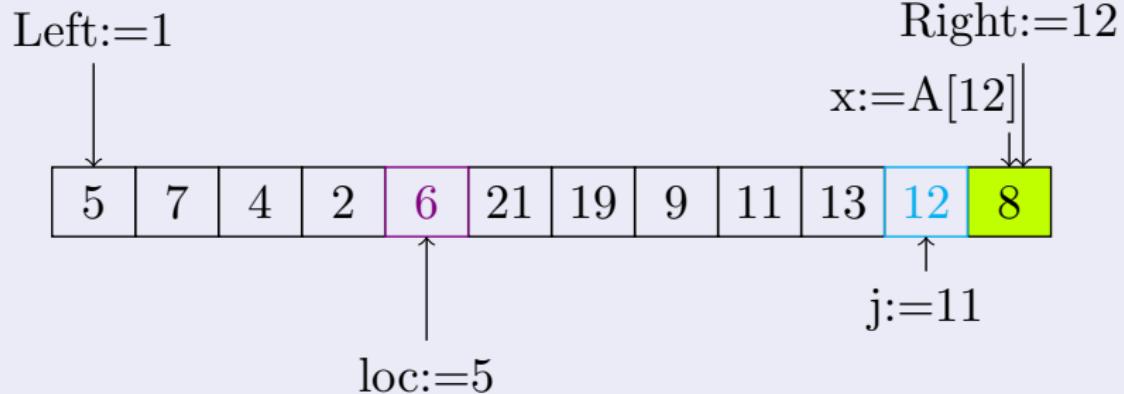
Avansăm cu loc.

Exemplu:



Trebuie să interschimbăm $A[5]$ cu $A[11]$.

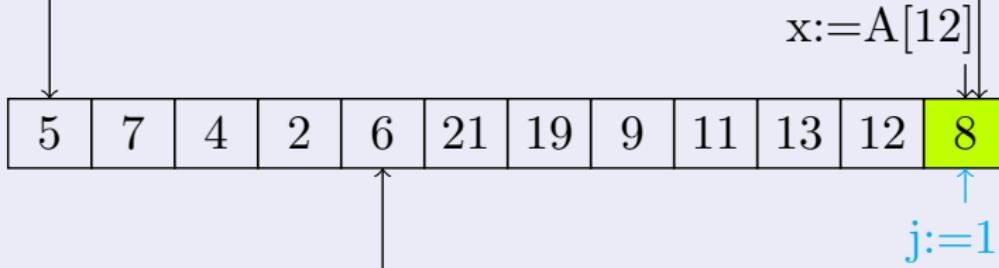
Exemplu:



Vectorul după interschimbare.

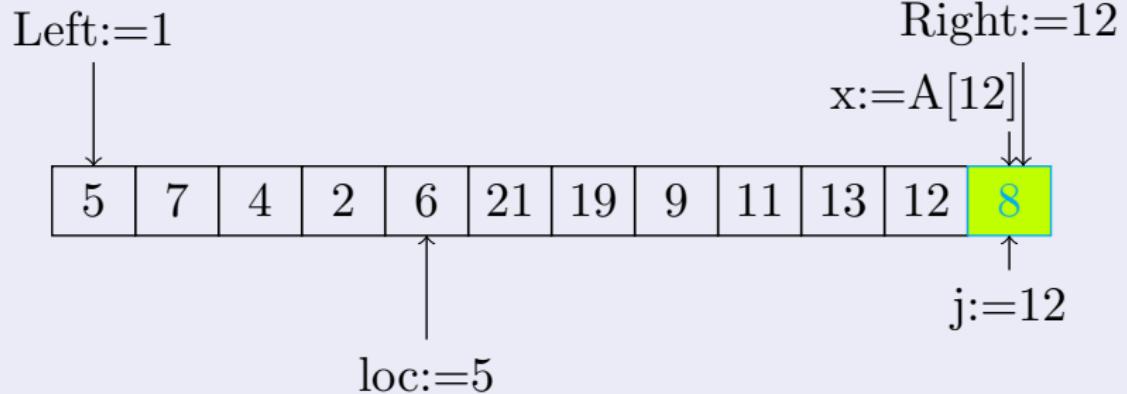
Exemplu:

Left:=1



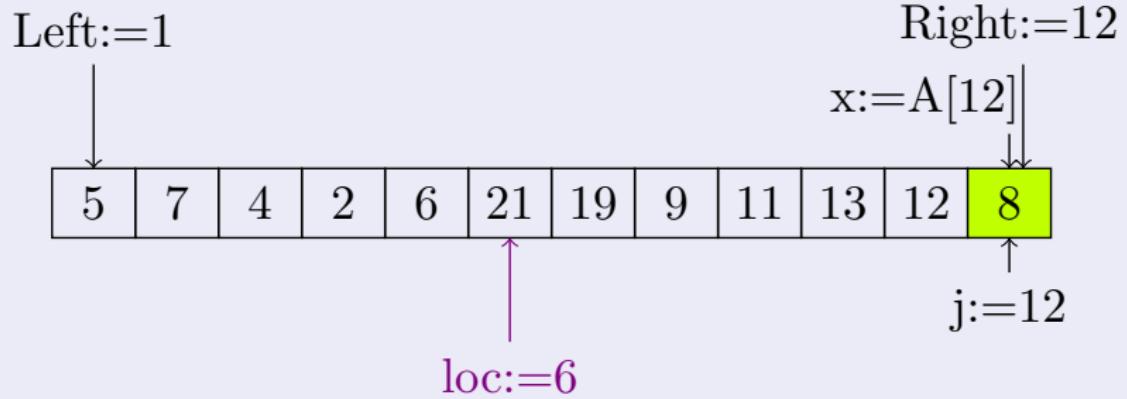
Avansăm cu j .

Exemplu:



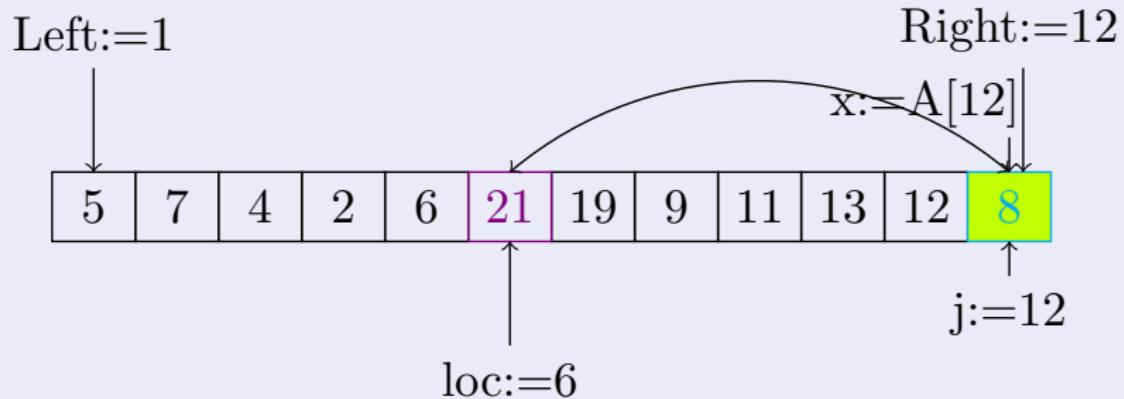
$$A[8] \leq x.$$

Exemplu:



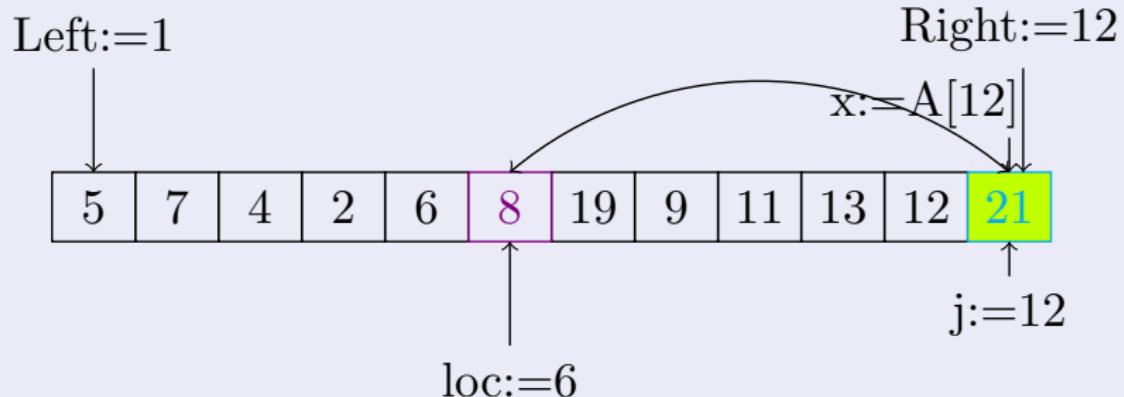
Avansăm cu loc.

Exemplu:



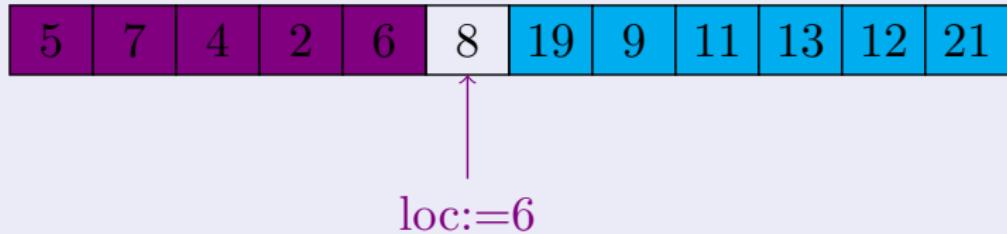
Trebuie să interschimbăm $A[6]$ cu $A[12]$.

Exemplu:



Vectorul după interschimbare. Algoritmul se termină.

Exemplu:



A 6-a valoare din A este pe poziția loc = 6. Am obținut partiția:
A[1..5], A[7..12].

Tabele de dispersie

Ne punem problema menținerii unui set finit de date, fiecare dintre ele identificată printr-o cheie unică, și notăm cu K mulțimea cheilor din acest set, numită *mulțimea cheilor actuale*. Prin natura lor, fie că sunt întregi de lungime fixă, dar mare, fie că sunt stringuri de caractere, cheile actuale K fac parte dintr-o mulțime U , de cardinalitate mult mai mare decât K , numită *universul cheilor*, lucru pe care îl notăm $|K| \ll |U|$. Pe această mulțime dorim să facem operații de inserare și ștergere, și, de asemenea, foarte frecvent, operații de căutare. Ideal ar fi ca operația de căutare să aibă timp $O(1)$ sau apropiat de el, în orice caz timp constant (și mic) în raport cu cheile din U . Structura de date care ne permite accesul în timp $O(1)$ la orice componentă este vectorul. Dacă ne-am putea permite menținerea în memorie a unui vector T a cărui mulțime de indici să fie chiar U , atunci pentru orice cheie actuală $k \in K$, în locația $T[k]$ am putea menține data identificată de cheia k . Un asemenea T se numește **tabel cu adresare directă**. Din păcate, pentru multe probleme concrete nu ne putem permite această soluție deoarece ar conduce la o risipă de spațiu. Dăm mai jos câteva exemple.

1. Vrem să menținem înregistrări cu date despre cei 68 de angajați ai unei firme. Fiecare angajat este identificat cu ajutorul unui număr de cod compus din 4 cifre. Cardinalul mulțimii cheilor actuale este $|K| = 68$, dar universul cheilor U este format din mulțimea tuturor întregilor cu 4 cifre în scriere zecimală, deci $|U| = 10^4 = 10000$. Un tabel cu adresare directă ar însemna să menținem în memorie un vector cu 10000 de componente, dintre care doar 68 ar fi efectiv folosite.
2. Vrem să menținem înregistrări cu date despre populația României. Fiecare locuitor este identificat cu ajutorul unor chei unice, codul numeric personal, un întreg compus din 12 cifre. Cardinalul mulțimii cheilor actuale este aproximativ 22 de milioane, $|K| = 22 \cdot 10^6$. Universul cheilor va fi mulțimea tuturor întregilor de 12 cifre, deci va avea dimensiunea $|U| = 10^{12}$.
3. Anuarul telefonic al unei localități menține informații despre abonații identificați cu ajutorul numelui și prenumelui, deci un sir de maximum 20 de caractere deci va avea dimensiunea $|U| = 26^{20}$.

În toate situațiile descrise mai sus este nerealistă folosirea unui tabel cu adresare directă, adică a unui tabel T indexat chiar după U . Trebuie să găsim o mulțime de indici pentru T , de cardinal m , mult mai mic decât $|U|$, eventual apropiat de cardinalul lui K , și o metodă de a dispersa cheile din K pe mulțimea de indici $\{0, 1, \dots, m - 1\}$. Cu alte cuvinte, avem nevoie de o funcție, definită pe universul cheilor și cu valori în mulțimea indicilor T :

$$h : U \rightarrow \{0, 1, \dots, m - 1\}$$

cu ajutorul căreia să găsim locul fiecărei date din mulțimea noastră: data identificată printr-o cheie k , $k \in K$, să se afle în locația $T[h(k)]$ a tabelului T .

O asemenea funcție se numește **funcție de dispersie**. Pentru a fi o funcție bună de dispersie, funcția h trebuie să îndeplinească anumite cerințe, și anume:

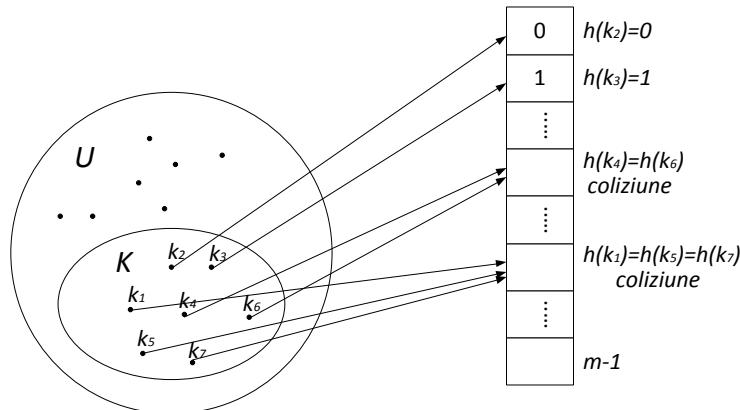
1. Să se poată calcula rapid. Timpul de calcul al lui $h(k)$ ne va da timpul de acces la componenta $T[h(k)]$.
2. Codomeniul ei să fie cât mai mic. Este dezirabil ca m să fie cât mai apropiat de $|K|$, în orice caz diferența dintre ele să fie acceptabilă ca dimensiune.
3. h să fie cât mai aproape de o funcție surjectivă, adică să avem cât mai puține locații goale. Această cerință se leagă de precedenta.
4. h să fie cât mai aproape de o funcție injectivă pe K . Cu alte cuvinte, pentru chei $k_1, k_2 \in K$, $k_1 \neq k_2$ să avem $h(k_1) \neq h(k_2)$.

Numim *coliziune* a cheilor k_1 și k_2 situația în care $h(k_1) = h(k_2)$. Cerința 4 se exprimă și sub forma „ne dorim să avem cât mai puține coliziuni”. Din păcate coliziunile nu pot fi evitate complet și trebuie găsite metode de rezolvare a lor.

Pentru a folosi un tabel de dispersie programatorul trebuie să rezolve două probleme: să aleagă o funcție de dispersie și să opteze pentru o metodă de rezolvare a coliziunilor. Aceasta din urmă se împarte în două mari clase :

- rezolvarea coliziunilor prin înlățuire(când locația $T[h(k)]$ conține o listă înlățuită a tuturor înregistrărilor cu chei ce au colizionat cu k) și
- rezolvarea coliziunilor prin adresare directă, când se folosesc metode mai sofisticate de căutare a unei locații libere în T pentru o cheie care colizionează.

Ne vom ocupa pe rând de cele 2 tipuri de probleme.



Functii de dispersie

Ne ocupam în acest paragraf de prezentarea câtorva tehnici euristice de creare a unor funcții de dispersie bune. Am exprimat informal în paragraful precedent câteva cerințe pentru o asemenea funcție

$$h : U \rightarrow \{0, 1, \dots, m - 1\}.$$

Vom lucra la **ipoteza hashingului simplu uniform (HSU)**, care, informal, spune că orice cheie este distribuită prin funcția h cu probabilitate egală în oricare din locațiile $\{0, \dots, m - 1\}$. Mai precis, pentru oricare $h \in K$ probabilitatea ca $h(k) = i$ este $\frac{1}{m}$ pentru orice $i \in [0 \dots m - 1]$, și este independentă de restul cheilor. Dacă P este probabilitatea de distribuție a cheilor din K în U , adică dacă $P(k)$ =probabilitatea de a extrage cheia k din U , atunci ipoteza HSU se formulează:

$$\sum_{\{k|h(k)=i\}} P(k) = \frac{1}{m}, \forall i = 0, 1, \dots, m - 1$$

Deoarece P este de obicei necunoscută, nu este în general posibil să verificăm că ipoteza HSU este satisfăcută .

În cele ce urmează vom interpreta cheile și numerele naturale, adică vom considera că $U < N$. Atunci când cheile sunt stringuri, metoda folosită pentru a le converti la numere naturale este urmatoarea: se înlocuiește fiecare caracter cu codul său ASCII și se calculează valoarea șirului rezultat ca întreg în baza 128.

De exemplu: AB se scrie $6566_{128} = 65 \cdot 128 + 66 = 8386$ ABC se scrie $656667_{128} = 65 \cdot 128^2 + 66 \cdot 128 + 67 = 1073605$

Metoda diviziuni

Se numesc funcții de dispersie obținute prin metoda diviziunii funcțiile de forma

$$h : U \rightarrow \{0, \dots, m - 1\}, h(k) = k \bmod m.$$

Ele sunt printre cele mai comune pentru că sunt ușor de calculat.

Alegerea unei astfel de funcții revine practic la alegerea lui m , dimensiunea tabelului de dispersie. Deci, ea va fi guvernată, pe de o parte, de dimensiunile lui K și de modul în care rezolvăm coliziunile. De exemplu, dacă ne propunem să le rezolvăm prin înlățuire și să menținem în medie 3 date într-o locație, m ar putea fi apropiat de $\frac{|K|}{3}$. Dacă însă vrem ca toate elementele din K să-și găsească locul în tabel, atunci am putea alege un m apropiat de $\frac{3|K|}{2}$, acceptând deci o risipă de spațiu egală cu $\frac{|K|}{2}$.

Dar acestea nu sunt singurele considerente care guvernează alegerea lui m . Important este ca cheile să fie bine dispersate pe multimea $\{0, 1, \dots, m - 1\}$ și să avem cât mai puține coliziuni. Vom evita valori pentru m de forma lui 2, deoarece, dacă $m = 2$, atunci $h(k) = k \bmod 2^p$ reprezintă doar ultimii p biți

din scrierea binară a lui k . Am pierdut în felul acesta informația conținută în ceilalți biți, lucru nerecomandabil: dacă nu avem indicații contrare, trebuie să facem ca $h(k)$ să depindă de toți biții lui k . Numerele apropiate de puteri ale lui 2 sunt și ele de evitat. Un exemplu de rezultat care ne conduce la această decizie este următorul: dacă $m = 2^p - 1$ și cheile k sunt siruri de caractere în baza 2^p , atunci 2 siruri care diferă doar printr-o transpoziție a două caractere adiacente vor avea aceeași valoare prin h .

Vom evita și valori pentru m de forma puterilor lui 10, căci dacă am alege un asemenea m , atunci $h(k)$ nu ar fi decât de o parte a caracterelor ce apar în scrierea lui k în baza 10. Valori bune pentru această metodă sunt m , un număr prim și cât mai departe de puteri ale lui 2. Evident, ne putem propune să indexăm tabelul după multimea de indici $\{1, 2, \dots, m\}$, în caz în care funcțiile din această clasă vor fi de forma $h(k) = k \bmod m + 1$.

Să considerăm exemplul firmei cu 68 de angajați, identificați cu chei întregi cu 4 cifre în scrierea în baza 10. Să presupunem că alegem pentru tabel o dimensiune apropiată de $\frac{3|K|}{2} = \frac{3}{2} \cdot 68$. O bună alegere pentru m ar fi $m = 97$, deoarece este prim, apropiat de $\frac{3}{2} \cdot 68$ și destul de departe de puteri ale lui 2.

Să considerăm cheile $k_1 = 3205$, $k_2 = 7148$ și $k_3 = 2345$.

Cu funcția de dispersie dată de metoda diviziunii

$$h(k) = k \bmod 97$$

Obținem pentru aceste chei următoarele adrese din tabel: $h(k_1) = 3205 \bmod 97 = 4$ $h(k_2) = 7148 \bmod 97 = 67$ $h(k_3) = 2345 \bmod 97 = 17$

Metoda multiplicării

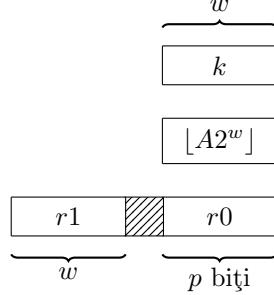
Fie A o constantă pozitivă subunitară, $0 < A < 1$. Să considerăm funcția dată de

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

$kA \bmod 1$ reprezintă partea fracționară a lui kA , $kA - [kA]$, care se inmulțește cu m , iar rezultatul se trunchiază la cel mai apropiat întreg. Observăm că valoarea lui m nu e critică pentru această metodă, iar puterile lui 2 sunt alegeri bune pentru m , pentru că funcția se va calcula ușor. Fie $m = 2^p$. Atunci

$$h(k) = \lfloor 2^p(kA - [kA]) \rfloor = \lfloor kA2^p - 2^p[kA] \rfloor$$

Fie w lungimea unui cuvânt mașina în biți și presupunem că k se poate reprezenta pe w biți de forma



$$K[A2^w] = r_1 2^w + r_0$$

Deoarece $kA2^p = (kA2^w)2^{p-w}$ înseamnă că $h(k) = (r_1 2^w + r_0)2^{p-w}$.

Knuth recomandă pentru A numărul de aur, $\frac{\sqrt{5}+1}{2}$, a căruia valoare aproximată cu 7 zecimale este 1,6180339. Revenind la exemplul nostru, să alegem pentru această metodă $A = 0,6180339^1$ și $m = 2^6 = 64$. Atunci, pentru cele 3 chei din exemplu și funcția $h(k) = \lfloor m(kA \text{mod} 1) \rfloor$, avem: $h(k_1) = h(3205) = 51$ deoarece partea fracțională a lui $3205 \cdot A$ este 0,7986, care, înmulțit cu 64 ne dă 51,1107, a cărui parte întreagă este 51. Analog, obținem $h(k_2) = h(7148) = 45$ și $h(k_3) = h(2345) = 18$.

Metoda împachetării

Metoda împachetării sau folding generează funcții de dispersie în felul următor. Fiecare cheie k , despre care am făcut presupunerea că este întreg, este partionată în bucăți de lungimi egale (eventual cu excepția ultimei), k_1, k_2, \dots, k_r . Dacă k este un întreg scris în baza 10 bucațile vor fi siruri de caractere în baza 10, de aceeași lungime fixă, l , pe care din nou le putem considera întregi. O funcție de dispersie

$$h : U \rightarrow [0 \dots 10^2 - 1]$$

este dată de formula:

$$h(k) = (k_1 + k_2 + \dots + k_r) \text{mod} 10^l$$

ceea ce înseamnă că se adună cele r bucăți, ca întregi, și din sumă se păstrează doar l cifre, cele mai puțin semnificative. Alte funcții de dispersie se obțin inversând ordinea cifrelor în toate bucațile pare sau în toate bucațile impare. De exemplu:

$$h(k) = (\overline{k_1} + k_2 + \overline{k_3} + \dots) \text{mod} 10^l$$

unde k este întregul obținut prin inversarea ordinii caracterelor. Pe exemplul nostru metoda împachetării se aplică în felul următor: spargem fiecare cheie de

¹Pentru calcularea valorii $h(k)$ nu păstrăm decât partea fracțională a produsului kA . Prin urmare, este suficient să alegem A ca fiind partea fracțională a numărului $\frac{\sqrt{5}+1}{2}$, adică 0,6180339, care este chiar $\frac{\sqrt{5}+1}{2}$.

4 caractere în 2 bucăți de câte 2 caractere, pe care le adunăm și omitem, dacă e cazul, cifra cea mai nesemnificativă pentru ca rezultatul să aibă lungimea tot de 2 caractere. Avem deci $h(k_1) = h(3205) = 32 + 5 = 37$ $h(k_2) = h(7148) = (71 + 48)\text{mod}100 = 19$ $h(k_3) = h(2345) = 23 + 45 = 68$ Pentru funcția $h'(k) = k_1 + \overline{k_2}$, în care a doua funcție o inversăm în oglindă înainte de adunare, obținem $h'(3205) = 32 + \overline{05} = 32 + 50 = 82$ $h'(7148) = 71 + \overline{48} = (71 + 84)\text{mod}100 = 55$ $h'(2345) = 23 + \overline{45} = 23 + 54 = 77$

Metoda pătratului

Presupunem că l este lungimea fixată a lui $h(k)$ ca întreg în baza 10. Fie

$$h : U \rightarrow [0 \dots 10^l], \quad h(k) = (k^2 \text{div} 10^{c_1})\text{mod}10^{c_2}.$$

Se ridică la pătrat, iar din întregul lung astfel obținut se elimină c_1 cifre dintre cele mai puțin semnificative și c_2 cifre dintre cele mai semnificative, păstrându-se exact l cifre din „centrul” lui k^2 . Cu alte cuvinte c_1 și l sunt fixate, iar c_2 se obține din formula:

$$l = \text{lung}(k^2) - c_1 - c_2.$$

Să aplicăm această metodă exemplului nostru. O cheie de 4 cifre va avea pătratul de 7 sau 8 cifre, dintre care să eliminăm $c_1 = 3$ dintre cele din dreapta, să păstrăm $l = 2$ cifre, și să eliminăm și restul de cifre „centrale”, adică a 4-a și a 5-a din stânga. $k_1 = 3205 : k_1^2 = 3205^2 = 10272025$ și $h(k_1) = 72$ $k_2 = 7148 : k_2^2 = 7148^2 = 51093904$ și $h(k_2) = 93$ $k_3 = 2345 : k_3^2 = 2345^2 = 5499025$ și $h(k_3) = 99$