1. STRUCTURI DE DATE

În scopul utilizării eficiente a unui calculator este important să se definească relațiile structurale existente în cadrul mulțimii datelor de prelucrat precum şi metodele de reprezentare şi manipulare a unor asemenea structuri.

Un prim exemplu de structură de date este tabloul. El a fost utilizat până acum fără a preciza că reprezintă o metodă de structurare a datelor şi fără a-i preciza în mod explicit proprietățile.

Astfel, fiind date K mulțimi de numere naturale, $A_{n_i} = \{1, 2, ..., A_{n_k}\}$, $i = \overline{1,k}$, unde A_{n_i} conține primele n_i numere naturale, atunci tabloul este o funcție $f : A_{n_1} \times .... \times A_{n_k} \to T$, în care T este o mulțime oarecare.

Dacă $k = 1$, tabloul este unidimensional, numit vector. Dacă $k = 2$, tabloul este o matrice. Structurarea elementelor din mulțimea T într-un tablou ne va permite să ştim care este primul element sau ultimul element din tablou. Se va putea accesa elementul al i-ulea dintr-un vector sau elementul de pe linia i şi coloana j a unei matrici.

În continuare vom depăşi noțiunea de tip de dată care este strâns legată de un anumit limbaj. Vom încerca să structurăm datele de un tip oarecare în aşa fel încât prelucrarea lor să fie eficientă. Structurile pe care le vom studia vor fi independente de limbajul de programare utilizat. Proprietățile lor şi tehnicile de operare cu aceste structuri de date vor fi general valabile. Vom lua în considerație deci, structura datelor dar şi clasa de operații ce se vor executa asupra datelor.

1.1.STRUCTURI DE DATE ELEMENTARE

1.1.1. Liste liniare

Lista liniară este un prim exemplu şi poate unul din cele mai simple, de structuri de date.

***Definiție*:** *O listă liniară este o colecție de $n \geq 0$ de elemente $x[1], ..., x[n]$ toate de acelaşi tip. Proprietățile structurale ale listei se reduc la pozițiile relative liniare ale elementelor. Astfel dacă $n > 0$, atunci $x[1]$ este primul element, iar $x[n]$ ultimul element. Pentru $1 < k < n$, elementul al k-lea, $x[k]$, este precedat de elementul $x[k-1]$ şi urmat de elementul $x[k+1]$.*

Principalele operații care se efectuează asupra listelor sunt :

1. Accesul la un element al listei (pentru examinare sau modificare).
2. Ştergerea unui element.
3. Inserarea unui element în listă.

În această secțiune vom folosi tablourile pentru a implementa listele liniare.

Primele liste liniare pe care le vom prezenta sunt *stivele* şi *cozile*. Acestea sunt caracterizate de faptul că operațiile de ştergere şi inserare se fac pe pziții prestabilite.

1.1.1.1. Stive

Într-o stivă, elementul şters este elementul cel mai recent inserat. Stiva implementează principiul *ultimul sosit, primul servit* (*last in, first out* - **LIFO**).

Fie $S[1...n]$ tabloul care implementează stiva. Atributul $\text{varf}(S)$ indică elementul cel mai recent introdus. Acesta este şi primul element care va parasi stiva.

Practic, la un moment dat stiva conține elementele $S[1]...,S[\text{var}\,f\,(s)]$. Dacă $\text{varf}(S)=0$ atunci stiva este goală iar dacă $\text{varf}(S)=n$ atunci stiva este plină.

Operația de inserare o vom numi *Pune_in_stiva* iar cea de ştergere o vom numi *Scoate_din_stiva*.

procedure *Pune_in_Stiva (S, x)*
  if $\text{varf}(S)<n$ **then**
    $\text{varf}(S)\leftarrow \text{var}\,f\,(S)+1$
    $S[\text{varf}(S)]\leftarrow x$
  else write(„Depăşire, exces de elemente")
  end if
end_proc

function *Scoate_din_Stiva* (*S*) : Întoarce un element de acelaşi tip cu elementele din stivă
  if $\text{varf}(S)=0$ **then**
    write((„Subdepăşire, lipsă elemente")
      return *Null*
    else
      $\text{varf}(S)\leftarrow \text{var}\,f\,(S)-1$
      return $S[\text{varf}(S)+1]$
    end if
end_function

1.1.1.2. Cozi

Într-o coadă elementul şters este cel care a stat cel mai mul în cadrul structurii. Coada implemnetează principiul *primul sosit, primul servit* (*first in, first out* - **FIFO**).

Fie $Q[1...n]$ tabloul care implementează coada. Într-o coadă toate ieşirile se fac de pe poziția *fața cozii* al cărui indice este dat de atributul $\text{prim}(Q)$ iar toate intrările se fac pe poziția indicată de atributul $\text{ultim}(Q)$.

Pentru implementarea structurii de coadă vom privi tabloul *Q* ca pe o structură circulară. Mai precis, ultimul element $Q[n]$ va avea un următor pe $Q[1]$. Incrementarea indicilor din tablou se va face cu funcția

function *Increment* (*Q,i*) : Întoarce o valoare întreagă între *1* şi *n*
  if $i<n$ **then return** *i+1*

```
    else return 1
    end if
  end_function
```

Inițial $\text{prim(Q)} = \text{ultim(Q)} = 0$ desemnând coadă vidă. De fiecare dată când se va produce acest eveniment, vom seta atributele cu aceste valor.

Pentru coadă vom folsi tot două operații *Pune_in_Coada* şi *Scoate_din_Coada*.

```
procedure Pune_in_Coada (Q, x)
  if ultim(Q) = 0 and prim(Q) = 0 then
      prim(Q) ← ultim(Q) ← 1
      Q[ultim(Q)] ← x
  else
      if ultim(Q) = Increment(Q, ultim(Q)) then
            write(„Depăşire, exces de elemente”)
      else
            ultim(Q) ← Increment(Q, ultim(Q))
            Q[ultim(Q)] ← x
      end if
  end if
end_proc

function Scoate_din_Coada (Q) : Întoarce un element de acelaşi tip cu elementele din coadă
  if prim(Q) = 0 then
      write((„Subdepăşire, lipsă elemente”)
         return Null
      else
          x ← Q[prim(Q)]
          if ultim(Q) = prim(Q) then
              prim(Q) ← ultim(Q) ← 0
          else
              prim(Q) ← Increment(Q, prim(Q))
          end if
          return x
      end if
  end_function
```

1.1.2. Liste înlănțuite

Un alt tip de alocare a listelor este *”alocarea înlănțuită”*. Pentru a realiza o astfel de alocare, nodurile listei vor conține două tipuri de informație:

– câmpurile ce caracterizează informația structurală a nodului. Le vom numi într-un cuvânt *INFO*;

– câmpuri de legătură reprezentând informaţia de secvenţă, legătura cu elementele adiacente în listă. Le vom numi câmpuri $LINK$. Aceste câmpuri vor fi de tip referinţă la nodurile listei (conţinutul lor va fi adresa de memorie a nodurilor adiacente).

Listele înlănţuite cu un singur câmp de legătură se numesc *"liste simplu înlănţuite"*. În cazul lor legătura se face, spre exemplu, la următorul element din listă. În cazul în care există în plus legătură spre nodul precedent, lista se numeşte *"dublu înlănţuită"*.

Vom prfezenta operaţiile care se pot efectua asupra unei liste înlănţuite considerând cazul listelor dublu înlanţuite.

Fie lista dublu înlănţuită *L* şi fie *x* un elemnt al acesteia. Notăm cu $\mathrm{Info}(x)$ informaţia asociată, cu $\mathrm{Prec}(x)$ câmpul pointer spre elementul precedent în listă iar cu $\mathrm{Urm}(x)$ câmpul pointer spre următorul element în listă. Dacă $\mathrm{Prec}(x) = \mathrm{Null}$ atunci elementul *x* este primul din listă (nu are element predecesor) iar dacă $\mathrm{Urm}(x) = \mathrm{Null}$ elemntul *x* este ultimul din listă (nu are element următor).

Atributul $\mathrm{Prim}(L)$ desemnează adresa primului element din listă. Dacă lista este vidă atunci $\mathrm{Prim}(L) = \mathrm{Null}$.

1.1.2.1. Parcurgere şi căutare

Parcurgerea listei se face cu algoritmul

```
procedure Parcurge_Lista (L)
    x ← Prim(L)
    while x ≠ Null do
        x ← Urm(x)
    end while
end_procedure
```

iar căutarea elemntului cu o anumită informaţie se face cu algoritmul

```
function Caută_in_Lista (L, Y) : Întoarce o referinţă la elemetele listei
    x ← Prim(L) ;  t ← Null
    while x ≠ Null do
        if Info(x) = Y then
            t ← x
            break
        else
            x ← Urm(x)
        end if
    end while
end_function
```

1.1.2.2. Adăugarea unui element în listă

Prin această operație înțelegem aşezarea unui element după ultimul element din listă. Pentru aceasta ar trebui să parcurgem lista de la primul element până la ultimul şi după aceea să adăugăm noul element. Ar însemna un timp de execuție $\Theta(n)$ dacă lista ar avea *n* elemente. Este mult mai simplu să folosim un nou argument al listei, $\mathrm{Ultim(L)}$, o referință către ultimul element al listei. Când lista este vidă vom avea $\mathrm{Prim(L) = Null},\ \mathrm{Ultim(L) = Null}$.

procedure *Adaug_in_Lista* (*L, Y*)
  Alocă spațiu pentru elementul *x*
  $\mathrm{Info(x)} \leftarrow \mathrm{Y}$
  $\mathrm{Prec(x)} \leftarrow \mathrm{Null};\ \mathrm{Urm(x)} \leftarrow \mathrm{Null}$
  if $\mathrm{Prim(L) = Null}$ **then**
    $\mathrm{Prim(L)} \leftarrow \mathrm{x};\ \mathrm{Ultim(L)} \leftarrow \mathrm{x}$
  else
    $\mathrm{Urm(Ultim(L))} \leftarrow \mathrm{x}$
    $\mathrm{Prec(x)} \leftarrow \mathrm{Ultim(L)}$
    $\mathrm{Ultim(L)} \leftarrow \mathrm{x}$
  end if
end_procedure

1.1.2.3. Inserarea unui element în listă

Prin această operație înțelegem aşezarea unui elemnt în listş înaintea unui element deja existent. Fie acesta $\mathrm{x_1}$.

procedure *Insereaza_in_Lista* (*L,* $\mathrm{x_1}$ *, Y*)
  Alocă spațiu pentru elementul *x*
  $\mathrm{Info(x)} \leftarrow \mathrm{Y}$
  $\mathrm{Prec(x)} \leftarrow \mathrm{Prec(x_1)};\ \mathrm{Urm(x)} \leftarrow \mathrm{x_1}$
  if $\mathrm{Prim(L) = x_1}$ **then**
    $\mathrm{Prim(L)} \leftarrow \mathrm{x}$
  else
    $\mathrm{Urm(Prec(x_1))} \leftarrow \mathrm{x}$
  end if
  $\mathrm{Prec(x_1)} \leftarrow \mathrm{x}$
end_procedure

1.1.2.4. Ştergerea unui element dintr-o listă

Fie $\mathrm{x_1}$ elemntul care trebuie şters.

procedure *Sterge_din_Lista* (*L,* $\mathrm{x_1}$)
  if $\mathrm{Prim(L) = x_1}$ **then**

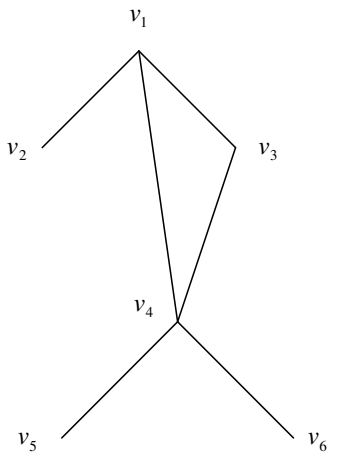
```
        Prec(Urm(x1)) ← Null
        Prim(L) ← Urm(x1)
    else
        Urm(Prec(x1)) ← Urm(x1)
    end if
    if Urm(x1) ≠ Null then
        Prec(Urm(x1)) ← Prec(x1)
    else
        Ultim(L) ← Prec(x1)
    end if
      Eliberează spațiul ocupat de x1
end_procedure
```

1.2.GRAFURI

1.2.1. Grafuri neorientate

***Definiția* 1:** *Un graf* G *este pereche ordonată* (X, Γ) *(vom nota* $G=(X, \Gamma)$*), unde* X *este o mulțime finită şi nevidă de elemente numite vârfuri, iar* Γ *este o mulțime de perechi de elemente ale lui* X *numite muchii.*

De obicei reprezentăm graful în plan ca o figură formată din puncte (vârfurile) şi segmente de dreaptă sau curbă (muchiile).



O *muchie* este deci o submulțime $\{u,v\}\subset X$ ale cărei elemente se numesc *extremitățile muchiei*. Pentru o astfel de muchie vom folosi notațiile (u,v) sau (v,u) având aceeaşi semnificație şi deci nereprezentând muchii diferite. În conformitate cu această notație vom numi graful definit mai sus *graf neorientat*. În continuare prin graf vom înțelege un graf neorientat.

Dacă un vârf $v\in X$ aparține unei muchii $e\in\Gamma$ $(v\in e)$ spunem că v *este incident cu* e. Dacă $u,v\in e$ (extremitățile muchiei) spunem că u şi v *sunt adiacente*. Dacă

$e_1, e_2 \in \Gamma$ sunt muchii distincte şi au un vârf comun atunci spunem că e_1 şi e_2 *sunt adiacente*.

Un graf este complet dacă oricare două vârfuri ale sale sunt adiacente.

Gradul unui vârf $v \in X$, notat $\deg(v)$ este numărul de muchii incidente lui v. Dacă $\deg(v) = 0$ atunci vârful v *este izolat*, iar dacă $\deg(v) = 1$, atunci vârful v *este terminal*.

***Definiția* 2:** *Fie $G = (X, \Gamma)$ un graf şi $u, v \in X$ (u şi v nu sunt neapărat distincte). Se numeşte lanț în graful G succesiunea de muchii:*

$$(u, u_1), (u_1, u_2), ..., (u_n, v), \quad \text{unde} \quad u_1, u_2, ..., u_n \in X$$

Mai spunem că $(u, u_1, ..., u_n, v)$ este un lanț cu extremitățile u şi v.

Un lanț este elementar dacă, cu excepția eventuală a extremităților, celelalte vârfuri diferă.

***Exemplu*:** (v_1, v_3, v_4, v_5)

Un lanț elementar pentru care extremitățile u şi v sunt egale $(u = v)$ se numeşte *ciclu*.

***Exemplu*:** (v_1, v_3, v_4, v_1)

***Definiția* 3:** *Fie $G = (X, \Gamma)$ un graf neorientat. Un subgraf al lui G este definit ca fiind graful $H = (Y, \Delta)$, unde $Y \subset X$, iar Δ este formată din toate muchiile din Γ care unesc vârfuri din Y.*

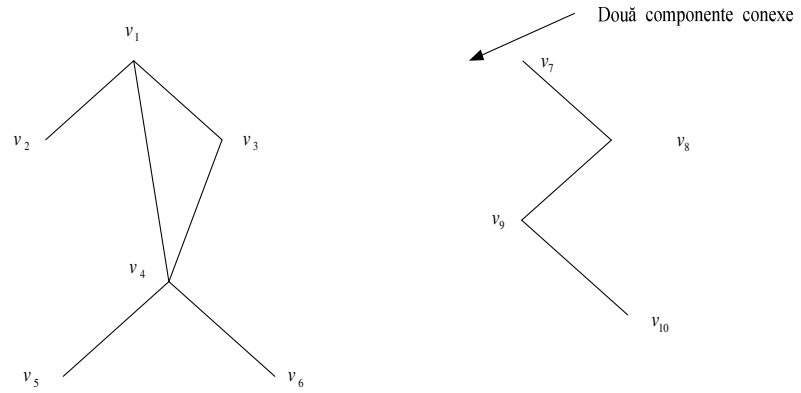
Un graf parțial al lui G este graful (X, Δ) în care $\Delta \subset \Gamma$. Graful parțial se mai numeşte şi *subgraf de acoperire al lui_G*.

Fie $G = (X, \Gamma)$ un graf şi $Y \subset X$ o submulțime nevidă a lui X. Numim subgraf al lui G *indus* de Y şi îl notăm $<Y>$ acel subgraf care are ca noduri mulțimea Y iar orice două vârfuri din Y sunt adiacente în $<Y>$ dacă şi numai dacă sunt adiacente în G.

Un graf $G = (X, \Gamma)$ este *conex* dacă oricare două vârfuri v_1 şi v_2 sunt unite printr-un lanț (mai spunem că v_1 şi v_2 *sunt conectate*).

Dacă un graf nu este conex se pune problema determinării *componentelor sale conexe*, o componentă conexă fiind un subgraf conex maximal, adică un subgraf conex în care nici un vârf din subgraf nu este unit cu unul din afară printr-o muchie din graful inițial.

Determinarea componentelor conexe se poate realiza pentru că relația *"u este conectat cu v"* unde $u, v \in X$, este o relație de echivalență. Ea va determina partiționarea lui X în clase de echivalență $X_1, ..., X_k$ iar grafurile induse $<X_i>$ $i = \overline{1, n}$ sunt conexe. Ele vor fi componentele conexe ale lui G.



Un ciclu care conține toate vârfurile grafului se numeşte *ciclu hamiltonian*. Într-un astfel de caz putem considera că vârfurile şi muchiile din ciclul hamiltonian formează un subgraf de acoperire al grafului inițial. Graful care conține un ciclu hamiltonian poartă numele de *graf hamiltonian*.

1.2.1.1. Reprezentarea unui graf în calculator

Există trei metode de reprezentare a unui graf în scopul prelucrării lui pe calculator. Două dintre ele sunt mai eficiente şi pe acestea le vom prezenta.

***Metoda* 1**

Definiție: *Fie* $G=(X,\ \Gamma)$ *un graf şi* $n=|X|,\ \ p=|\Gamma|$. *Se numeşte matrice de adiacență a grafului* G, *matricea* $A_{nxn}=(a_{ij})$ *matricea ale cărei elemente satisfac relația* $a_{ij}=\begin{cases}1, & dacă\ (x_i x_j)\in\ \Gamma\\ 0, & dacă\ (x_i x_j)\in\ \Gamma\end{cases}$, *unde* $X=\{x_1,x_2,...,x_n\}$ *şi unde s-a ales o ordine pe mulțimea vârfurilor.*

Observație:

1. În funcție de ordinea aleasă pe mulțimea vârfurilor matricea de adiacență este unic determinată.

2. În cazul grafului neorientat matricea de adiacență este simetrică (în raport cu diagonala principală).

Exemplu: pentru graful de mai sus (pag.1) matricea de adiacență este:

$$A=\begin{array}{c}\\ x_1\\ x_2\\ x_3\\ x_4\\ x_5\\ x_6\end{array}\begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6\\ 0 & 1 & 0 & 0 & 0 & 0\\ 1 & 0 & 1 & 1 & 0 & 0\\ 0 & 1 & 0 & 1 & 0 & 0\\ 0 & 1 & 1 & 0 & 1 & 1\\ 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0\end{pmatrix}$$

***Metoda* 2**

Dacă pe mulțimea X a vârfurilor grafului $G=(X,\ \Gamma)$ s-a ales o ordine atunci, pentru fiecare vârf, putem stabili lista vârfurilor adiacente cu acesta.

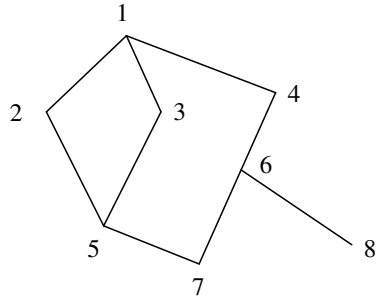
vârf	lista vârfurilor adiacente
x_1	$x_2,\ x_3,\ x_4$
x_2	x_1
x_3	$x_1,\ x_4$
x_5	x_4
x_6	x_4

1.2.1.2. Parcurgerea unui graf

În scopul utilizării informației din vârfurile unui graf este necesară parcurgerea grafului (trecerea prin fiecare vârf). Pe lângă vizitarea efectivă a unui vârf sunt necesare acțiuni care să permită parcurgerea în continuare a celorlalte vârfuri nevizitate.

1.2.1.2.1. Algoritmul ″BF″ (breadth first)

Acest algoritm realizează o parcurgere *″în lățime″* a grafului în sensul următor: se vizitează un vârf inițial, etichetat spre exemplu cu i, apoi vecinii acestuia (vârfurile adiacente lui i), apoi descendenții încă nevizitați ai acestora, etc. De exemplu, pentru graful din figură, metoda va furniza următoarea ordine de vizitare: 1, 2, 3, 4, 5, 6, 7, 8.



Vom folosi un vector $viz\ [n],\ n = |x|$ în care:

$$viz[i]=\begin{cases}0 & vârful\ \ i\ \ nu\ \ a\ \ fost\ \ vizitat\\ 1 & vârful\ \ i\ \ a\ \ fost\ \ vizitat\end{cases}$$

Vom mai folosi o coadă ζ în care vom introduce vârfurile vizitate. Prelucrarea nodului (vârfului) din fața cozii presupune introducerea în coadă a tuturor vecinilor acestuia, încă nevizitați, urmată de vizitarea lor.

procedure $BF\ (i,\ n)$

i: vârful de la care se pleacă

n: numărul de vârfuri; $n = |x|$

ζ: coada cu indicii F (fața) şi R (spatele)

VIZITARE: funcție care prelucrează informația dintr-un vârf

for $k = 1,\ n$ **do**

$viz[k] \leftarrow 0$

next k

VIZITARE (i); $viz[i] \leftarrow 1$; $F \leftarrow R \leftarrow 1$; $\zeta(R) \leftarrow 1$;

while $F \le R$ **do**

for $k = 1,\ n$ **do**

if $\exists\ (x[\zeta[F]],\ x[k] \in \Gamma)$ and $(viz[k] = 0)$ **then**

$R \leftarrow R+1$; ζ ® $\leftarrow k$;

VIZITARE (k); $viz[k] \leftarrow 1$

end if

next k

$F \leftarrow F+1$

end while

end_proc.

Să demonstrăm că algoritmul lucrează corect, adică sunt vizitate toate vârfurile j care sunt legate printr-un lanț de vârful i.

Fie $d\ (i, j)$ lungimea minimă (măsurată în muchii) a unui lanț care uneşte pe i cu j. Dacă nu există un astfel de lanț atunci $d\ (i, j) = +\infty$. Arătăm prin inducție că $\forall\ m \in N$, algoritmul vizitează toate vârfurile j cu $d\ (i, j) = m$.

Pentru $m = 0$, este evident pentru că vârful i este vizitat la început.

Presupunem că sunt vizitate toate vârfurile j cu $d\ (i, j) \le m$.

Fie j un vârf cu $d\ (i, j) = m+1$ şi fie j predecesorul lui j. Atunci $d\ (i, k) = m$ şi dacă k este vizitat, conform ipotezei de inducție. Cum vizitarea lui k este însoțită de introducerea lui k în coadă, iar aceasta generează prelucrarea lui k rezultă că vârful j va fi vizitat.

***Observație*:** Algoritmul BF parcurge vârfurile legate de i prin lanțuri în ordinea crescătoare a distanței față de i.

Algoritmul BF vizitează, după un vârf k, pe primul dintre vecinii acestuia încă nevizitați.

1.2.1.2.2. Algoritmul ″D″ (depth)

Se deosebeşte de algoritmul BF prin faptul că, după prelucrarea vârfului k, se trece la prelucrarea ultimului vecin al lui k dintre cei încă nevizitați. Aceasta se va concretiza prin utilizarea unei stive în care sunt introduse vârfurile vizitate.

procedure $D\ (i,\ n)$

i: vârful de la care se pleacă

$n:\ n = |x|$

ζ : stiva pentru vârfuri vizitate cu indicatorul T pentru vârful stivei

for $k = 1,\ n$ **do**

$viz\ [k] \leftarrow 0$

next *k*

VIZITARE $(i);\ viz\ [i] \leftarrow 1;\ T \leftarrow 1;\ \zeta\ [T] \leftarrow i;\ 1 \leftarrow T;$

while $T \neq 0$ **do**

$l_1 \leftarrow 1;$

for $k = 1,\ n$ **do**

if $\exists\ (x[\zeta[T]],\ x[k] \in \Gamma)$ and $(viz\ [k] \leftarrow 0)$ **then**

$l \leftarrow l+1;\ \zeta\ [l] \leftarrow k;$

VIZITARE $(k);\ viz\ [k] \leftarrow 1$

end_if

next k

if $l \neq l_1$ **then** $T \leftarrow 1$

else $T \leftarrow T-1;$

$l \leftarrow T;$

end_if

end_while

end_proc.

1.2.1.2.3. Algoritmul ″DF ″ (depth first)

Algoritmul încearcă să meargă în *"adâncimea grafului"* ori de câte ori este posibil. Astfel, prelucrarea vârfului v_k, care are vecinii $v_k^1, \ldots, v_k^p$, înseamnă prelucrarea primului dintre aceşti vecini care este nevizitat, să presupunem v_k^j, $1 \le j \le p$. Deci se va trece la prelucrarea vârfului v_k^j şi a vecinilor nevizitați ai acestuia. Abia după ce acest lucru se încheie se revine la prelucrare în continuare a vecinilor încă nevizitați ai lui v_k.

Va trebui să utilizăm o stivă care, la pasul curent, să ne dea posibilitatea de a memora vârful v_k pentru a reveni la el şi a prelucra vecinii lui rămaşi nevizitați. Vom mai folosi un vector ultim $[n]$, $n = |x|$ cu elemente având semnificația: $ultim\,[k] = ultimul$ vizitat dintre vecinii lui k.

procedure $DF\ (i,\ n)$

i: vârful de la care se pleacă

$n:\ n = |x|$

S: stiva şi T indicând vârful stivei

for $k = 1,\ n$ **do**

$viz\ [k] \leftarrow 0;\ ultim\ [k] \leftarrow 0;$

next k

VIZITARE $(i);\ viz\ [i] \leftarrow 1;\ j \leftarrow i;\ T \leftarrow 0$

repeat

$k \leftarrow 0;\ l \leftarrow ultim\ [j] + 1;$

while $(k = 0)$ and $(1 \le n)$ **do**

if $\exists\ (x[j],\ x[l],\ x[l] \in \Gamma)$ **then** $k \leftarrow l$

else $l \leftarrow l + 1;$

end_if

end_while

if $k = 0$ **then**

if $T \neq 0$ **then**

$j \leftarrow S\ [T];\ T \leftarrow T - 1;$

end if

else

if $viz\ [k] = 0$ **then**

VIZITARE $(k);\ viz\ [k] \leftarrow 1;$

$T \leftarrow T + 1;\ S\ [T] \leftarrow j;$

$ultim\ [\ j\ ] \leftarrow\ k\ ;\ \ j \leftarrow k\ ;$

else $ultim\ [\ j\ ] \leftarrow\ k\ ;$

end_if

end_if

until $T = 0\ ;$

end_proc

***Observație*:** Algoritmul *"DF"* poate fi utilizat pentru problemele care cer găsirea ieşirii dintr-un labirint.

1.2.1.2.4. Determinarea componentelor conexe ale unui graf

Utilizăm una din metodele de parcurgere a unui graf: în general *"BF"* sau *"D"*. Vom putea da un răspuns la întrebarea *"dacă un graf este conex sau nu"*. În caz negativ vom putea determina componentele conexe ale grafului.

procedure conex (n)

$n\colon\ n = |\ x\ |$

ζ : coada; vom utiliza parcurgerea BF

$viz\ [\ n\ ]$: tablou care arată dacă un vârf a fost vizitat sau nu. El nu va mai fi inițializat cu procedura BF .

for $k = 1,\ n$ **do**

$viz\ [\ k\ ] \leftarrow 0\ ;$

next *k*

repeat

$k \leftarrow 0\ ;\ i \leftarrow 1\ ;$

while $(k = 0)$ and $(i \le n)$ **do**

if $viz\ [\ i\ ] = 0$ **then** $k \leftarrow i\ ;$

else $i \leftarrow i + 1\ ;$

end_if

end_while

if $k \neq 0$ **then**

$BF\ \ (k,\ n,\ \mathrm{var}\,\zeta,\ viz)$

if $|\ \zeta\ | = n$ **then**

write („*Graful este conex*");

end_if

for $i=1,\ |\zeta|$ **do** write $(\zeta\ [i])$ **next** *i*

end_if

until $k=0;$

end_proc

Pe lângă răspunsul la întrebarea de mai sus, problema are şi o utilitate practică. Spre exemplu, dacă se dă o rețea de oraşe şi drumurile dintre ele se poate pune întrebarea *"dacă se poate ajunge din oraşul A în oraşul B"*. Rezolvarea se face astfel: parcurgând graful corespunzător rețelei cu plecarea din vârful A se determină componenta conexă care îl conține pe A. Dacă această componentă îl conține şi pe B, răspunsul este afirmativ.

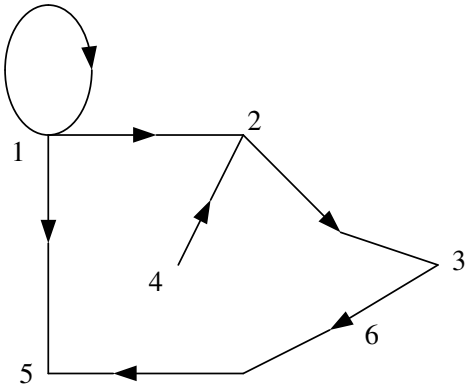
1.2.1.2.5. Determinarea ciclurilor hamiltoniene dintr-un graf neorientat

Această problemă nu are o soluție optimă. Pentru ea se foloseşte metoda backtracking şi rezolvarea este aceeaşi cu cea de la problema comis-voiajorului.

1.2.1.3. Grafuri orientate

***Definiție*:** Un graf orientat G este o pereche ordonată $(X,\ \Gamma)$, $G=(X,\ \Gamma)$, unde X este o mulțime finită şi nevidă de *vârfuri*, iar Γ este o mulțime de perechi ordonate de vârfuri numite *arce*. Orice $arc\ (i,\ j)\in\Gamma$ are un sens de parcurgere, de la extremitatea sa inițială i la extremitatea finală j. Un arc de forma $(i,\ i)$ se numeşte *buclă*.

Noțiunile de *adiacență şi incidență* sunt aceleaşi ca la grafurile neorientate. În cazul grafurilor orientate noțiunea de lanț îşi are corespondentul în noțiunea *drum* iar noțiunea de ciclu îşi are corespundentul în noțiunea de *circuit*.



$(4, 2, 3, 6)$ *este drum*

$(6, 5, 1, 2)$ *nu este drum*

Succesiunea $(u_1, u_2), (u_2, u_3), \ldots, (u_{n-1}, u_n)$ este drum dacă $\forall\, i = \overline{2, n-1}$, u_i este extremitatea finală pentru (u_{i-1}, u_i) şi extremitatea inițială pentru (u_i, u_{i+1}).

Metodele de reprezentare ale grafurilor neorientate se mențin şi la grafurile orientate (nu ştiu dacă a doua ?).

În cazul grafurilor orientate observăm că matricea de adiacență nu mai este simetrică.

Fiind dat un graf orientat $G = (X, \Gamma)$, vom ataşa fiecărui arc $(i, j) \in \Gamma$ o valoare mai mare sau egală cu zero şi pe care o vom nota $\cos(i, j)$, în felul următor:

$$\cos(i, j) = \begin{cases} c, & \text{dacă } (i, j) \in \Gamma \\ 0, & \text{dacă } i = j \\ +\infty, & \text{dacă } (i, j) \notin \Gamma \end{cases}$$

Observăm că un graf orientat poate fi dat şi prin matricea costurilor sale.

Dintre problemele care se pun în legătură cu grafurile orientate o vom studia pe cea a determinării *circuitului hamiltonian*.

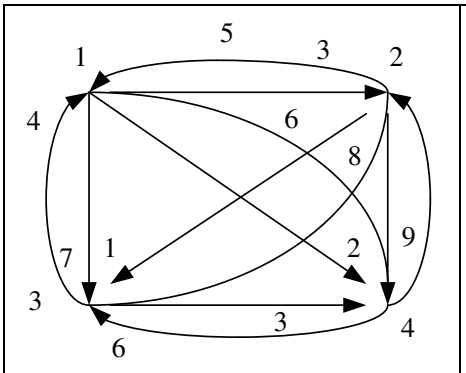
Circuitul hamiltonian al unui graf orientat este circuitul care trece o singură dată prin fiecare vârf şi care are un cost total minim. În general, prin costul unui drum înțelegem suma costurilor arcelor sale.

1.2.1.3.1. Problema circuitului hamiltonian într-un graf orientat

Se pot întâlni diverse formulări ale acestei probleme, în funcție de interpretarea care se dă valorii $\cos(i, j)$ ataşată arcului (i, j). Această valoare poate reprezenta, spre exemplu: lungimea arcului, timpul necesar parcurgerii arcului, costul parcurgerii arcului.

***Rezolvare*:** Fără a restrânge generalitatea, presupunem că circuitul va pleca din vârful etichetat cu 1 întorcându-se tot la el. Notăm matricea de cost $C_{i,j}$, $i, j = \overline{1, n}$.

Fie graful



$$\begin{Bmatrix} 0 & 3 & 7 & 2 \\ 5 & 0 & 1 & 9 \\ 4 & 8 & 0 & 3 \\ 6 & 2 & 6 & 0 \end{Bmatrix}$$

Să presupunem că am determinat circuitul hamiltonian.

Fie $i = \{1, 2, \ldots, n\}$ unul din vârfurile circuitului. Atunci drumul de la i la 1 este de cost minim. Dacă j este vârful care urmează în drum după i, atunci şi drumul de la j la 1 este de cost minim. Această observație ne determină să dorim metoda programării dinamice.

Relațiile de recurență

Fie $S \subseteq X \setminus \{1\}$ o mulțime de vârfuri din x. Dacă $n = |x|$, atunci $d = |S| \leq n - 1$. Fie $i \in S$ un vârf din circuit. Notăm cu $l(i, k)$, costul minim al drumului de la i la 1 care trece prin k vârfuri din S diferite între ele şi diferite de i. Dacă j este următorul vârf pe acest drum atunci:

$$l(i,k) = \min\{c_{ij} + l(j, k-1)\}, \quad \text{unde } i, j \in S \text{ şi } k = |S|, \quad j \neq i$$

Când $k = 0$, drumul de la i la 1 nu trece prin nici un vârf intermediar şi deci $l(i, k) = c_{i1}$.

Costul minim al circuitului va fi $l(1, n-1)$.

In cadrul algoritmului vom folosi:

$c[1....n, 1....n]$: matricea costurilor

$l[1....n, 1....n]$: matricea costurilor drumurilor minime

$v[0....n]$: vectorul vârfurilor în ordinea inversă a pozițiilor în circuit.

procedure hamilton;

$\min k \leftarrow +\infty$;

for *i=2* **to** *n* **do** {pasul de inițializare. Se determină drumul de cost minim pentru $k = 0$}

$l[i, 0] \leftarrow c[i, l]$

```
      if min k > l [i, 0] then
        min k ← l [i, 0];
        v [0] ← 1;
      end if
next i
l [1, 0] ← + ∞;
for k=1 to n-1 do {Se determină circuitul minim care trece prin k
    vârfuri intermediare, 1 ≤ k ≤ n−1}
      min k ← + ∞;
      for i=1 to n do {Se determină drumul de cost minim care pleacă
              din i, ajunge în 1 şi trece prin alte k vârfuri
              diferite între ele şi diferite de i}
        min i ← + ∞
        for j=1 to n do
              if j ≠ i then
                    if min i > c [i, j] + l [j, k−1] then
                          min i ← c [i, j] + l [j, k−1];
                    end if
              end if
        next j
        l [i, k] ← min i;
            if min k > l [i, k] then
              min k ← l [i, k];
              v [k] ← i;
            end if
      next i
    next k
    write ('circuitul minim are costul ', l [1, n−1])
    write ('ordinea vârfurilor din circuit este: ');
    for i=n-1 to 0 do
      write (v [ i ])
    next i
    write (1);
  end_proc
```

1.2.1.3.2. Determinarea drumurilor de cost minim care pleacă din acelaşi vârf

Fie un graf orientat $G=(X,\ \Gamma)$ şi $i_0 \in X$ un vârf al său. Ne interesează să determinăm pentru toate vârfurile $j \in X$, $j \neq i_0$ dacă există drum de la i_0 la j. Mai mult, pentru fiecare j ne interesează drumul de cost minim dintre cele care unesc i_0 cu j. Vom folosi algoritmul Dijkstra care implementează o metodă de tip Greedy.

Vom genera drumurile minime în ordinea crescătoare a lungimilor lor. La fiecare pas. Mulțimea S a vârfurilor $j \in X$ cu proprietatea că există cel puțin un drum de la i_0 la j, se va îmbogăți. Inițial luăm $S = \{ i_0 \}$.

Presupunem că am determinat m drumuri minime de la i_0 la mulțimea de vârfuri $S = \{ i_1, ..., i_m \} \subset X$. Dorim să mai adăugăm un vârf la mulțimea $S \Leftrightarrow$ să găsim un vârf $i_{m+1} \in X \setminus S$, astfel încât drumul de la i_0 la i_{m+1} este cel mai scurt dintre cele care unesc pe i_0 de vârfurile din $X \setminus S$.

***Observație*:** Exceptând pe i_{m+1} acest drum va trece doar prin vârfuri din S. Întradevăr, dacă $\exists\ j \in X \setminus S$ un vârf care se află pe acest drum atunci drumul de la i_0 la j ar fi mai scurt decât cel de la i_0 la i_{m+1} şi am contrazice alegerea lui i_{m+1}.

Alegerea lui i_{m+1}

Pentru această alegere vom folosi un vector $d\ [1, ..., n]$ unde:

$$d_i = \begin{cases} distanța\ de\ cost\ minim\ de\ la\ i_0\ la\ i\ dacă\ i \in S \\ lungimea\ drumului\ de\ cost\ minim\ \ dacă\ i \notin S \\ de\ la\ i_0\ la\ i\ trecând\ numai\ prin\ vârfuri\ din\ S \end{cases}$$

Inițial vom lua $d_i = cost\ (i_0,\ i),\ i = \overline{1,\ n}$.

Mulțimea S o vom reprezenta printr-un vector caracteristic $s\ [1, .., n]$ cu $s_i = \begin{cases} 1,\ dacă\ i \in S \\ 0,\ dacă\ i \notin S \end{cases}$. În aceste condiții i_{m+1} va fi acel vârf din $X \setminus S$ pentru care $d_{i^{m+1}} = \min d_j\ ,\ X \setminus S$ (*). Vom simboliza $S \leftarrow S \cup \{ i_{m+1} \}$ prin $S\, i_{m+1} = 1$.

Adăugarea lui i_{m+1} la S va genera şi modificări asupra valorilor d_j cu $j \in X \setminus (S \cup \{ i_{m+1} \})$. Aceasta pentru că la vârfurile $j \in X \setminus (S \cup \{ i_{m+1} \})$ este posibil să se ajungă şi prin drumuri ce trec prin i_{m+1}.

Deci, pentru $\forall\ j \in X \setminus (S \cup \{ i_{m+1} \})$, dacă $d_{im+1} + cost\ (i_{m+1},\ j) < d_j$

atunci avem $d_j \leftarrow d_{im+1} + cost\ (i_{m+1},\ j)$.

Este evident că, pentru un astfel de vârf j, va trebui să memorăm predecesorul său, i_{m+1}, în cazul în care inegalitatea de mai sus este adevărată. Vom folosi un vector $TATA[1,\dots,n]$, în care vom face atribuirea $TATA[j] \leftarrow i_{m+1}$, atunci când realizăm $d_j \leftarrow d_{im+1} + \text{cost}(i_{m+1}, j)$.

Inițial $TATA[i_0] = 0$ şi $TATA[i] = 0$, $\forall\, i \in X$ pentru care $\text{cost}(i_0, i) = +\infty$.

În rest vom inițializa $TATA[i] = i_0$ pentru $\forall\, i \in X$ cu $\text{cost}(i_0, i) < +\infty$.

Algoritmul se va opri când nu vom mai putea găsi cu formula $(*)$ un $k \in X \setminus S$ astfel încât $d_k = \min d_j,\ j \notin S,\ d_k < +\infty$.

$\forall\, i \in S \Leftrightarrow (\forall\, i = \overline{1, n}\ cu\ s_i = 1)$ va fi vârful final al unui drum ce pleacă din i_0 şi va avea lungimea minimă d_i. Drumul va fi format din vârfurile:

i, $TATA[i]$, $TATA[TATA[i]]$, ... , i_0 luate în ordine inversă.

procedure *DIJKSTRA* (i_0)

$s[1,\dots,n]$, $d[1,\dots,n]$, $TATA[1,\dots,n]$, unde $n = |S|$

for *i=1* **to** *n* **do**

$s[i] \leftarrow 0$; $d[i] \leftarrow \text{cost}[i, j]$ {Pasul de inițializare}

if $d[i] < +\infty$ **then**

$TATA[i] \leftarrow i_0$

else $TATA[i] \leftarrow 0$;

end if

next i

$s[i_0] \leftarrow 1$; $TATA[i_0] \leftarrow 0$;

repeat

$d[k] \leftarrow \text{min\_}X \setminus S\,(k)$;

if $d[k] < +\infty$ **then**

$s[k] \leftarrow 1$;

for *j=1* **to** *n* **do**

if $(s[j] = 0)$ and $(d[k] + \text{cost}(k, j) < d(j))$ then

$(d[j] \leftarrow d[k] + \text{cost}(k, j))$;

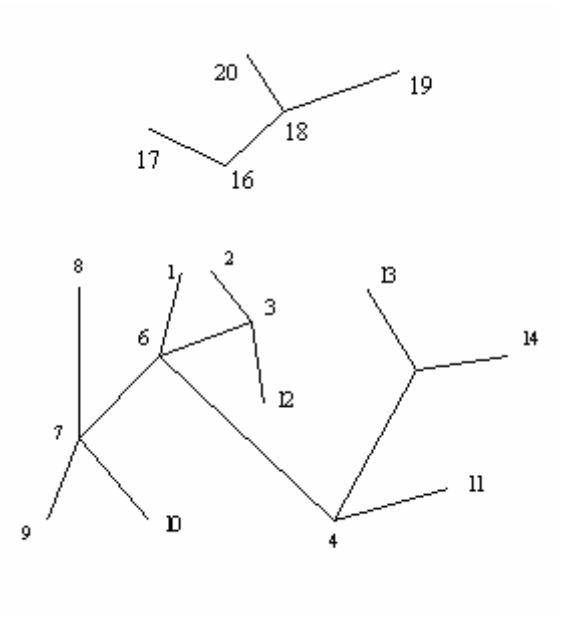
$TATA[j] \leftarrow k$;

```
            end if
          next j
        end if
      until d [k] = ∞
      for i=1 to n do
        if (i ≠ i0) and (d [i] < + ∞) then
              write (d [i]);
              List_drum (i);
        end if
      next i
    end_proc

    procedure List_drum (i);
      if i ≠ 0 then
        List drum (TATA [i]);
        write (i);
      end if
    end_proc
```

1.3. ARBORI

***Definiție*:** *Un arbore este un graf conex şi fără cicluri.*



***Observaţie*:** În definiţie, prin graf, înţelegem un graf neorientat.

O reprezentare a unui arbore, care să-i justifice denumirea, se obţine dacă aşezăm *pe niveluri* vărfurile sale. Acest lucru se realizează astfel:

1. Se alege un vărf al arborelui, pe care îl vom numi *rădăcina arborelui* şi se aşează pe nivelul 1.

2. Pe fiecare nivel $i>1$ se aşează vărfurile care sunt legate de rădăcină printr-un lanţ de lungime $i-1$.

3. Se trasează muchiile.

***Observaţii*:** Plasarea vărfurilor unui arbore pe diverse niveluri implică existenţa unei ordini a acestora generată de chiar aşezarea lor. În contextul aşezării pe niveluri mai observăm:

1. Numim *descendenţi* ai unui vărf toate vărfurile plasate pe niveluri inferioare celui al vărfului dat şi legate printr-un lanţ de acesta.

2. Se numeşte *descendent direct* al unui vărf acel descendent al vărfului dat, legat de acesta printr-o muchie şi plasat pe nivelul imediat inferior.

3. Un vărf de pe nivelul $i>1$ este legat printr-o muchie de un singur vărf de pe nivelul $i-1$. În plus, vărfurile de pe nivelul i nu sunt legate prin muchii (altfel ar exista cicluri în arbore).

4. Aşezarea pe niveluri a vărfurilor unui arbore ne dă posibilitatea să considerăm că arborele este un graf orientat.

***Definiţie*:** Se numeşte pădure un graf neorientat şi fără cicluri.

Evident, componentele conexe ale unei păduri sunt arbori. De aici rezultă că şi pădurea poate fi considerată graf orientat.

1.3.1. Reprezentarea şi parcurgerea arborilor şi pădurilor

Există mai multe metode de reprezentare a arborilor oarecare.

Reprezentarea arborelui determină şi posibilitatea traversării lui. În consecinţă metodele de reprezentare sunt strâns legate de metodele de parcurgere a arborelui.

Există trei metode de parcurgere a unui arbore oarecare: A – postordine, A preordine şi parcurgere pe niveluri.

Studiul nostru nu se axează în mare măsură pe studiul arborilor oarecare. De aceea vom da doar metoda de parcurgere pe niveluri şi o metodă de reprezentare adecvată ei.

Fie un arbore cu vărfurile $\{1, 2, ..., n\}$. Fie $i_0 = \overline{1, n}$ rădăcina lui. Ataşăm acestui arbore tabloul $TATA\ [1\ ...\ n\ ]$ cu valorile:

$$TATA\ [i\ ] = \begin{cases} 0, & dacă\ i = i_0 \\ k \in \{1, ..., n\} & dacă\ i \neq i_0 \end{cases}$$

Pentru a parcurge pe niveluri acest arbore vom considera încă un tablou $viz\,[1\ldots n]$ care va avea valorile $viz\,[i] = nivel_i \;\; i = \overline{1,\, n}$, unde $nivel_i$ este nivelul pe care este aşezat vârful i, dacă i a fost vizitat şi $viz\,[i] = 0$, dacă i nu a fost vizitat.

procedure Traversare_pe_niveluri (i_0)

i_0: rădăcina arborelui

$VIZ\,[1.n]$

for *i=1* **to** *n* **do**

$VIZ\,[i] \leftarrow 0;$

next i

VIZITARE (i_0); $VIZ\,[i_0] \leftarrow 1$; $niv \leftarrow 2$;

repeat

continuă ← false;

for *i=1* **to** *n* **do**

if $(VIZ\,[i] \leftarrow 0)$ and $(VIZ\,[TATA[i]] = niv-1)$ **then**

VIZITARE (i); $VIZ\,[i] \leftarrow niv$;

if (not continuă) **then** continuă ← true;

end_if

end_if

next i

$niv \leftarrow niv+1$;

until (not continuă);

end_proc

În cazul unei păduri cu $n>0$ vârfuri fie $i_0^1, \ldots, i_0^n \in \{1, \ldots, n\}$ rădăcinile celor k arbori ce o compun. Procedura de mai sus poate fi modificată astfel încât să selecteze toate rădăcinile (vârfurile care au componenta corespunzătoare din vectorul $TATA$ egală cu 0). Pot fi vizitați arborii în parte sau se pot vizita toate vârfurile pădurii aflate pe nivelul curent (se vizitează $i_0^1,\ i = 1,\, k$ apoi descendenții direcți ai acestora, apoi descendenții descendenților.....).

1.3.2. Arborele parțial de cost minim al unui graf neorientat

Fie $G = (X,\, \Gamma)$ un graf neorientat. Ataşăm fiecărei muchii a grafului o valoare pozitivă astfel:

$$COST\ (i,\ j) = COST\ (j,\ i) = \begin{cases} \infty, & dacă\ \ i = j \\ c > 0, & dacă\ \exists\ (i,\ j) \in \Gamma \end{cases}$$

Se pune problema să determinăm un graf parțial şi conex $H = (X,\ \Delta)$ al lui G astfel încât suma costurilor muchiilor din Δ să fie minimă. Observăm că, în acest caz, graful căutat este un arbore.

Problema enunțată este cunoscută ca *problema conectării oraşelor cu cost minim.* Arborele care va fi determinat va purta numele de *arbore parțial de cost minim* corespunzător grafului G.

Există doi algoritmi pentru această problemă, amândoi bazați pe metoda Greedy. În cazul algoritmului lui Kruskal, arborele este cunoscut prin muchiile sale. Inițial arborele este vid şi, la pasul curent, se adaugă muchia cu costul cel mai mic dintre cele neincluse încă în arbore. Algoritmul lui Prim ia în considerație şi vârfurile arborelui la fiecare pas.

1.3.2.1. Algoritmul Kruskal

Strategia algoritmului ne impune să ordonăm de la început muchiile grafului inițial în ordinea crescătoare a costurilor lor. Fie $m = |\Gamma|$ numărul de muchii din graf.

Considerăm matricea $MAT\ [1...m,\ 1...3\ ]$ în care fiecare linie corespunde unei muchii din graf; linia 1 corespunde muchiei cu costul cel mai mic, linia nu corespunde muchiei cu costul cel mai mare.

Pentru $k = \overline{1,\ n}$, $MAT\ [k,\ 1\ ]$ şi $MAT\ [k,\ 2\ ]$ reprezintă vârfurile muchiei k, iar $MAT\ [k,\ 3\ ]$ reprezintă costul ataşat muchiei.

Inițial, cele $n = |x|$ vârfuri din graf se consideră că formează o pădure cu n arbori. Fiecare vârf este un arbore cu rădăcina vârful respectiv. În final aceşti arbori sunt reuniți într-unul singur.

La fiecare pas doi arbori ai pădurii se reunesc într-un singur arbore. Această reuniune se face conform următorului principiu: dacă muchia de la pasul curent are vârfurile situate în arbori diferiți, atunci aceşti arbori se vor reuni.

Vom considera un vector $RAD\ [1...n\ ]$ (RADăcină). Fie $i_0^1, ..., i_0^n$ rădăcinile celor k arbori de la pasul curent. Fie $n_1, ..., n_k$ numărul de vârfuri din fiecare arbore dintre cei k arbori de la pasul curent. Semnificația componentelor vectorului RAD este următoarea:

$$RAD\ [i\ ] = \begin{cases} -n_j, & dacă \quad i = i_0^j \qquad unde\ \ j \in \{1, ..., k\} \\ i_0^j, & dacă \quad i \notin \{i_0^1, ..., i_0^k\}\ şi \\ & \qquad i\ este\ vârf\ în\ arborele\ i_0^j \end{cases}$$

$i = \overline{1,\ n}$

Inițial, $RAD\,[\,i\,] = -1,\ i = \overline{1,\ n} \Leftrightarrow$ fiecare vârf constituie un arbore.

Aflarea arborelui din care face parte un vârf

Pentru un vârf $i \in \{1,\ldots,n\}$ va trebui să aflăm acel vârf j care este rădăcină a arborelui din care face parte vârful i.

function $APART\,(i)$
  i: vârful căutat
  $j \leftarrow i$;
  while $RAD\,[\,j\,] \geq 0$ **do**
    $j \leftarrow RAD\,[\,j\,]$;
  end while
  $APART \leftarrow j$;
end_func

Reuniunea a doi arbori

Se realizează în momentul când găsim o muchie cu vârfurile situate în doi arbori de rădăcini diferite. Pentru ca muchia să fie luată în considerare se realizează reuniunea celor doi arbori. Aceasta va urmări principiul: arborele cu număr de vârfuri mai mic este adăugat la cel cu număr de vârfuri mai mare. Fie $(i,\ j)$ o astfel de muchie şi $k1 = RAD\,[\,i\,],\ k2 = RAD\,[\,j\,]$, iar $k1 \neq k2$.

procedure $REUN\,(k1,\ k2)$
  $nr \leftarrow RAD\,[\,k1\,] + RAD\,[\,k2\,]$;
  if $abs\,(\,RAD\,[\,k1\,]) < abs\,(\,RAD\,[\,k2\,])$ **then**
    $RAD\,[\,k1\,] \leftarrow k2;\ RAD\,[\,k2\,] \leftarrow nr$;
  else
    $RAD\,[\,k2\,] \leftarrow k1;\ RAD\,[\,k1\,] \leftarrow nr$;
  end_if
end_proc

procedure $KRUSKAL\,(n,\ m)$
  n: numărul de vârfuri din graf
  m: numărul de muchii din graf

$COST$: costul total al arborelui parțial

$RAD\ [1...n\ ]$: vectorul cu semnificația amintită

$MAT\ [1...m,\ 1...3\ ]$: matricea muchiilor şi a costurilor

$COST \leftarrow 0;\ \ j \leftarrow 1$;continuă ← true;

while $(j \leq m)$ and continuă **do**

$K1 \leftarrow APART\ (MAT\ [\ j,\ 1\ ])$;

$K2 \leftarrow APART\ (MAT\ [\ j,\ 2\ ])$;

if $k1 \neq k2$ **then**

$REUN\ (k1,\ k2)$;

write $(\ MAT\ [\ j,\ 1\ ],\ MAT\ [\ j,\ 2\ ],\ MAT\ [\ j,\ 3\ ])$;

$COST \leftarrow COST + MAT\ [\ j,\ 3\ ]$;

end_if

if $abs\ (\ RAD\ [\ k1\ ] = -\ n)$ or $abs\ (\ RAD\ [\ k2\ ] = -\ n)$

then continuă ← false

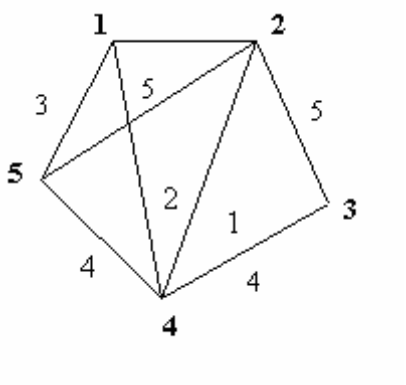
else $j \leftarrow j + 1$;

end_if

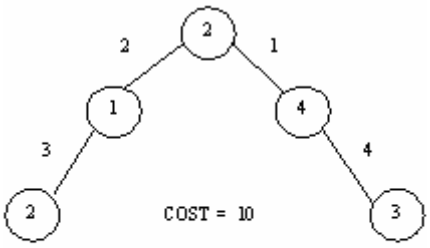
end_while

end_proc.

***Exemplu*:**

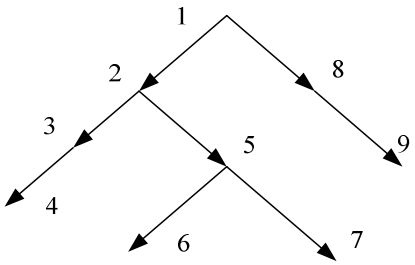


$$\Rightarrow MAT = \begin{pmatrix} 2 & 4 & 1 \\ 1 & 2 & 2 \\ 1 & 4 & 2 \\ 1 & 5 & 3 \\ 4 & 5 & 4 \\ 3 & 4 & 4 \\ 2 & 3 & 5 \\ 2 & 5 & 5 \end{pmatrix}$$



1.3.3. Arbori binari

***Definiție*:** Un arbore binar este un arbore în care fiecare vârf are cel mult doi descendenți, făcându-se distincția clară între descendentul stâng şi cel drept al fiecărui vârf.



***Observații*:**

1. Orientarea de la arborii oarecare (generată de plasarea vârfurilor pe niveluri) se menține şi în cazul arborilor binari.

2. Reprezentarea pe niveluri determină să nu mai fie necesar sensul arcelor.

3. Fiecare vârf poate fi considerat rădăcină pentru un subarbore. Evident, pentru fiecare vârf vom avea subarborele stâng şi subarborele drept, eventual unul din ei sau amândoi putând fi arbori vizi (fără nici un vârf).

1.3.3.1. Reprezentarea arborilor binari in calculator

a. Reprezentarea secvențială (statică)

Forma de reprezentare secvențială *"standard"* este cea în care pentru fiecare vârf i se precizează descendenții săi direcți, stâng $ST[i]$ şi drept $DR[i]$, eventual informația $INFO[i]$ ataşată vârfului. Cele trei tablouri vor avea dimensiunea n (numărul de vârfuri din arbore). În acest context este necesar a se preciza indicele vârfului rădăcină. Lipsa unui descendent se simbolizează prin $ST[i]=0$ sau $DR[i]=0$.

***Exemplu*:** Pentru arborele binar de mai sus avem:

$$\begin{cases} ST = (2,3,4,0,6,0,0,0,0) \\ DR = (8,5,0,0,7,0,0,9,0) \\ RAD = 1 \end{cases}$$

b. Reprezentarea înlănțuită (dinamică)

Fiecărui vârf al arborelui binar i se ataşează următorul tip de înregistrare:

$x = \{ LLINK,\ INFO,\ RLINK \}$

unde:

$LLINK$: referința spre descendentul stâng;

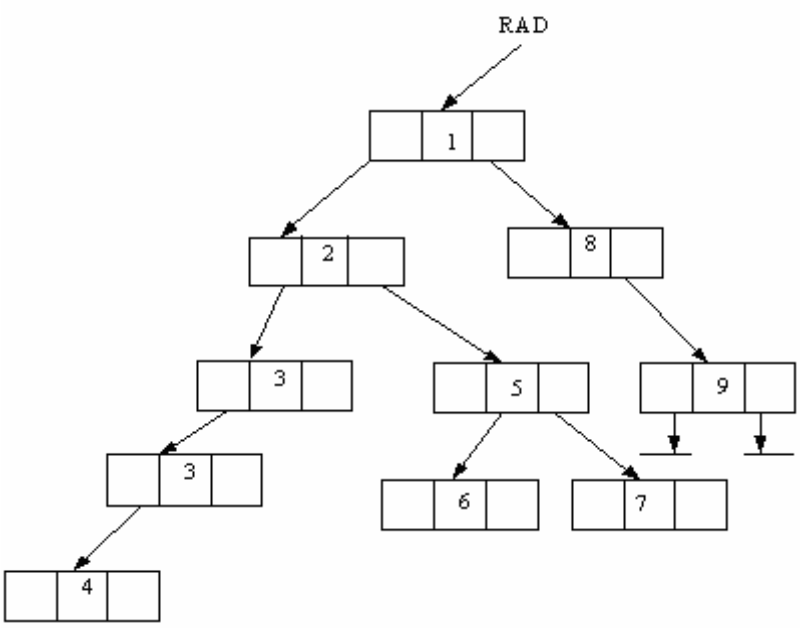
$RLINK$: referința spre descendentul drept;

$INFO$: informația ataşată vârfului.

Absența unui descendent se simbolizează prin referința nulă, Φ.

În acest context trebuie precizată o variabilă referință RAD, la vârful rădăcină al arborelui.

Pentru arborele binar din exemplu, reprezentarea înlănțuită este următoarea:



1.3.3.2. Parcurgerea (traversarea) arborilor binari

Cele două tipuri de reprezentare sunt adecvate pentru cele trei metode de parcurgere a arborilor binari.

1. Parcurgerea în *preordine*: se vizitează rădăcina, apoi se parcurge în preordine subarborele stâng şi apoi tot în preordine subarborele drept.

1. Parcurgerea în *inordine*: se parcurge în inordine subarborele stâng, apoi se vizitează rădăcina şi în final se parcurge tot în inordine subarborele drept.

2. Parcurgerea în *postordine*: se parcurge în postordine subarborele stâng, apoi în postordine subarborele drept şi în final se vizitează rădăcina.

Pentru fiecare din cele trei metode de parcurgere vom prezenta două versiuni. Prima versiune utilizează o stivă, iar cea de-a doua versiune este recursivă. Vom utiliza reprezentarea înlănțuită a arborelui binar. Procedurile scrise pentru reprezentarea secvențială rămân ca temă.

Stiva (utilizată în procedurile din prima versiune) va conține noduri cu următoarea structură $Y = \{R, LINK\}$, unde R este referința spre un vârf al arborelui binar şi $LINK$ este referința la următorul nod din stivă.

***Observație*:** Referința R este adresa primului vârf din arbore care urmează să fie vizitat. La un moment dat, elementele din stivă conțin referințe la toate vârfurile arborelui care urmează să fie vizitate.

1.3.3.2.1. Parcurgerea în preordine

***Cu stivă*:**

procedure PREORD(RADarb)

$RAD\ arb$: referință la rădăcina arborelui (de tip X)

$VÂRF$ stivă: referință la vârful stivei (de tip Y)

P: referință la vârfurile arborelui

$INSEREAZĂ$, $EXTRAGE$: proceduri de lucru cu stiva

VÂRFstivă $\leftarrow \Phi$; $P \leftarrow RAD\,arb$;

continuă $\leftarrow$ true;

while continuă **do**

while $P \neq \Phi$ **do**

PRELUCREAZĂ $(INFO(P))$;

INSEREAZĂ (P) ;

$P \leftarrow RLINK\ (P)$;

end while

```
    if VÂRFstivă ≠ Φ then
      EXTRAGE (P);
      P ← RLINK (P);
    else continuă ← false;
    end if
  end while
end proc
```

***Recursiv*:**

```
procedure PREORD (P)
  P: referință la vârfurile arborelui (de tip X)
  if P ≠ Φ then
    PRELUCREAZĂ (INFO(P));
    PREORD (LLINK(P));
    PREORD (RLINK(P));
  end if
end proc
```

Apelul celor două proceduri:

(1) ⇒ PREORD (RADarb)
(2) ⇒ PREORD (RADarb)

⇒ 3 2 1 6 4 5 8 7 9

1.3.3.2.2. Parcurgerea în inordine

***Cu stivă*:**

```
procedure INORD(RADarb)
  VÂRFstiva ← Φ  P ← RAD arb;
  continuă ← true;
  while continuă do
    while P ≠ Φ do
      INSEREAZĂ (P);
      P ← LLINK (P);
    end while
    if VÂRFstiva ≠ Φ then
      EXTRAGE (P) ;
```

PRELUCREAZĂ $(INFO(P))$;

$P \leftarrow RLINK(P)$;

else continuă ← false;

end if

end while

end proc

***Recursiv*:**

procedure INORD(P)

if $P \neq \Phi$ **then**

INORD $(LLINK(P))$;

PRELUCREAZĂ $(INFO(P))$;

INORD $(RLINK(P))$;

end if

end proc

Apel: (1) ⇒ INORD (RADarb)

(2) ⇒ INORD (RADarb)

⇒ 1 2 3 4 5 6 7 8 9

1.3.3.2.3. Parcurgerea în postordine

***Cu stivă*:**

procedure POSTORD (RADarb)

P, P_1: referință la vârful arborelui

VÂRFstivă: referință la vârful stivei

cont1, cont2: var. booleene

P ← RAD arb;

cont1 ← cont2 ← true;

while cont1 **do**

while cont2 **do**

if $(LLINK(P) \rightarrow \Phi)$ or $(RLINK(P) \rightarrow \Phi)$ **then** INSEREAZĂ (P);

if LLINK(P)≠Φ **then** $P \leftarrow LLINK(P)$ **else** $P \leftarrow RLIND(P)$;

end if

else cont2 ← false;

end if

end while {cont2}

```
    PRELUCREAZĂ (INFO(P));
    if VÂRF stiva = Φ then cont1 ← false
    else
        EXTRAGE (P1);
        if (P = LLINK (P1)) and (RLINK (P1) ≠ Φ) then
                INSEREAZĂ (P1); P ← RLINK (P1);
                cont2←true;
        else
                P ← P1; cont2 ← false;
        end if
      end if
    end while {cont1}
  end proc

  Recursiv:
  procedure POSTORD(P)
   if P ≠ Φ then
     POSTORD (LLINK(P));
     POSTORD (RLINK(P));
     PRELUCREAZĂ (INFO(P));
   end if
  end proc

  Apel: (1) ⇒ POSTORD (RAD arb);
        (2) ⇒ POSTORD (RAD arb);
⇒ 1 2 5 4 7 9 8 6 3
```

1.3.3.3. Arbori binari de sortare

Fie M o mulțime de elemente. Considerăm că pe această mulțime s-a instituit o relație de ordine ″< ″. Dacă tipul $x = \{LLINK, INFO, RLINK\}$ caracterizează nodurile unui arbore binar, să considerăm că $INFO(x) \in M$.

Dacă P şi Q sunt două referințe la nodurile arborelui binar, şi dacă:

1. $INFO(Q) < INFO(P)$, pentru $\forall Q$ care referă un nod din subarborele stâng al lui P;

2. $INFO\ (P) < INFO\ (Q)$, pentru $\forall\ Q$ care referă un nod din subarborele drept al lui P.

Atunci arborele binar este *arbore binar de sortare*.

Aceeaşi definiție pentru metoda de reprezentare secvențială.

Fie $\{1,...,n\}$ mulțimea de etichete ataşată nodurilor unui arbore binar şi $ST\ [1,...,n]$, $DR\ [1,...,n]$ şi $INFO\ [1,...,n]$ vectorii care definesc arborele. Fie $INFO\ (i) \in M$ pentru $\forall\ i = \overline{1,\ n}$. Dacă considerăm doi indici $i, j = \{1, 2, ..., n\}$ şi dacă:

1. $INFO\ [\ j\ ] < INFO\ [\ i\ ]$ pentru $\forall\ j$ din subarborele stâng al lui i;

2. $INFO\ [\ i\ ] < INFO\ [\ j\ ]$ pentru $\forall\ j$ din subarborele drept al lui i.

atunci arborele este un *arbore binar de sortare*.

1.3.3.3.1. Căutarea şi inserarea într-un arbore binar de sortare

Fie M o mulțime ordonată şi un arbore binar de sortare astfel încât $\forall\ P$ referință la nodurile arborelui avem $INFO\ (P) \in M$. Dacă $Y \in M$ se pune problema să determinăm dacă:

- $\exists\ P$ referință la nodurile arborelui astfel încât $Y = INFO\ (P)$ (versiunea înlănțuită);

sau

- $\exists\ i \in \{1,...,n\}$ astfel încât $Y = INFO\ [\ i\ ]$ (versiunea secvențială).

În caz negativ se va insera în arbore un nod nou în locul corespunzător.

procedure Caut_insert (var RefRadArb ; Y p, găsit)

RefRadArb, P, P_1: referințe la nodurile arborelui

$Y \in M$: informația care trebuie găsită

găsit: var booleană

$P_1 \leftarrow \Phi$; P←RefRadArb; găsit ← false;

while (not găsit) and $(P \neq \Phi)$ **do**

if $Y = INFO\ (P)$ **then** găsit ← true

else

$P_1 \leftarrow P$;

if $Y < INFO\ (P)$ **then** $P \leftarrow LLINK\ (P)$ **else** $P \leftarrow RLINK\ (P)$

end if

end if

```
    end while
    if (not găsit) then
      Alocă spațiu pentru P; INFO (P) ← Y;
      LLINK (P) ← RLINK (P) ← Φ
      if P1 = Φ then RefRadArb ← P
      else
          if Y < INFO (P1) then LLINK (P1) ← P else RLINK (P1) ← P;
          end if
      end if
    end if
  end_proc
```

***Apel*:** Fie variabilele globale : RAD referință la rădăcina arborelui, Y o variabilă cu proprietatea $Y \in M, P$ referință la nodurile arborelui, găsit variabila booleană, atunci apelul este *Cant_insert (RAD, Y, P, găsit)*. Dacă găsit = true, atunci în P vom găsi referință la nodul căutat. Altfel P va referi nodul care s-a inserat.

***Observație*:** Procedura poate fi apelată şi când arborele este vid $\Leftrightarrow$ $RAD = \Phi$. Repetând apelul se vor insera nodurile arborelui. Poziția nodurilor va depinde de ordinea în care este furnizată informația Y.

Iată o procedură recursivă care va realiza doar inserarea într-un arbore. Poate fi apelată şi când arborele este vid, $RAD = \Phi$.

```
  procedure INSERT (Y ; P)
    Y ∈ M : informația din nodul care se inserează
    P : referință la nodurile arborelui
    if (P = Φ) then
      Alocă spațiul pentru P;
      INFO(P)←Y;
      LLINK(P) ← RLINK(P)←Φ;
    else
      if Y < INFO (P) then INSERT (Y, LLINK (P))
                      else INSERT (Y, RLINK (P));
      end if
    end if
  end_proc
```

Dacă RAD este referința globală la rădăcina arborelui, atunci apelul va fi $INSERT\ (Y,\ RAD)$.

1.4.STRUCTURI HEAP ŞI APLICAȚII

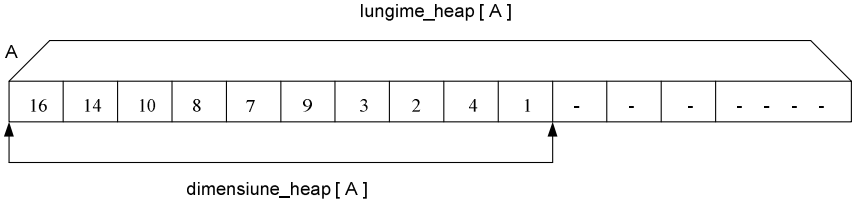
Structura de date *heap* este un vector care poate fi vizualizat ca un arbore binar aproape complet (dacă arborele are n niveluri, atunci vârfurile de pe nivelurile $1 - n - 1$ au câte doi descendenți, nivelul n, cel al frunzelor, este parțial plin de la stânga la dreapta).

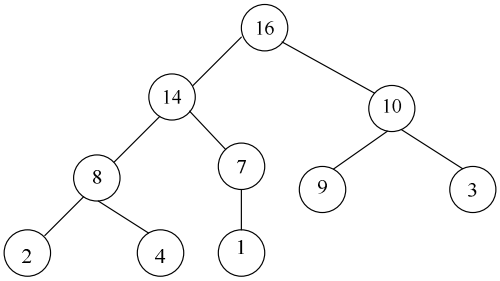
Un heap A are două atribute:

– lungime_heap $[A]$ ≡ numărul elementelor din vector;

– dimensiune_heap $[A]$ ≡ numărul de elemente ale heap-ului care sunt memorate în vector.

Evident dim_heap $[A]$ ≤ lungime_heap $[A]$. Rădăcina arborelui este $A\ [1]$.





Fie un indice $1 \le i \le \dim\_heap\ [A]$. Corespunzător unui nod al arborelui, definim:

$$\begin{cases} Parinte\ (i) \overset{def}{\equiv} \left\lfloor \frac{i}{2} \right\rfloor \\ Stânga\ (i) \overset{def}{\equiv} 2i \\ Dreapta\ (i) \overset{def}{\equiv} 2i + 1 \end{cases}$$

Proprietatea structurii heap: pentru orice nod diferit de rădăcină, valoarea ataşată acestuia este mai mică sau cel mult egală cu valoarea asociată părintelui său.

Altfel spus:

$\forall\ i \neq 1$ avem $A\ [Parinte\ (i)] \le A\ [i]$

$1 < i \le \dim\_heap\ [A]$

***Definiție*:** *Înălțimea unui nod al arborelui este numărul muchiilor aparținând celui mai lung drum de la nodul respectiv la o frunză.*

1.4.1. Reconstituirea proprietății de heap

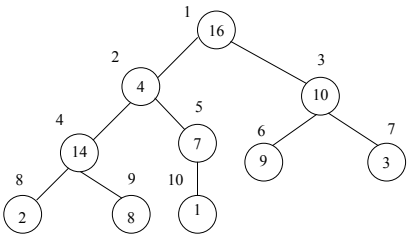
Fie A un vector şi i un indice din vector. Asociem vectorului A un arbore binar şi presupunem că subarborii care au ca rădăcini pe $S\tan ga\,(i)$ şi $D\text{reapta}\,(i)$ sunt heap-uri. Este posibil ca $A[i]$ să fie mai mic decât descendenții săi, şi deci, pentru i nu este îndeplinită proprietatea de heap.

Algoritmul următor reconstruieste proprietatea de heap.

Se determină cel mai mare element dintre $A[i]$, $A[Stânga(i)]$ şi $A[Dreapta\,(i)]$ şi fie $\max$ indicele acestuia. Dacă $i=\max$, procedura se termină pentru că A este un heap. Dacă $i\neq\max$, atunci cel mai mare element este unul dintre descendenți. Rezultă că se interschimbă $A[i]\leftrightarrow A[\max]$.

Acum nodul i şi descendenții săi au proprietatea de heap, dar $A[\max]$ are valoarea inițială a lui $A[i]$ şi este posibil ca el să nu mai îndeplinească proprietatea de heap (adică subarborele de rădăcină $A[\max]$ să nu mai fie heap). Atunci se reia procedeul de mai sus pentru indicele $\max$.

```
procedure Reconstitue_heap (A, i)
  l ← Stănga (i)
  r ← Dreapta (i)
  if  (l ≤ dim_heap [A] and A[l] > A[i]) then
          max ← l
      else
     end_if       max ← i
     if (r ≤ dim_heap [A] and A [r] > A [max]) then
           max ← r
     end_if
      if (max ≠ i) then
           A [i] ↔ A [max]
           Reconstitue_heap (A, max)
      end_if
end_proc
```

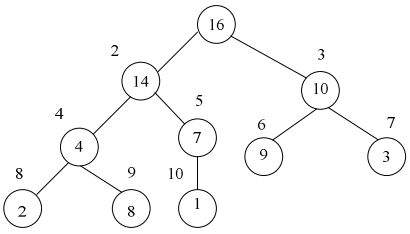


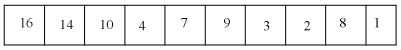
Reconstituire_heap (A, 2)

l = 4, r = 5 ⇒ max = 4

şi

A [2] ↔ A [4] ⇔ 4 ↔ 14



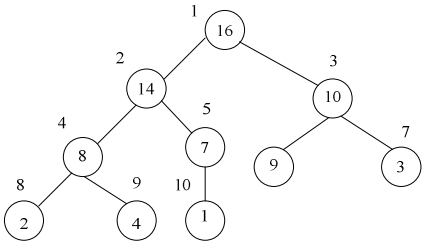


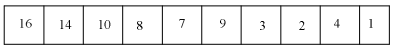
Reconstituire_heap (A, 4)

l = 8, r = 9 ⇒ max = 9

şi

A [9] ↔ A [4] ⇔ 4 ↔ 8





1.4.2. Construirea unui heap

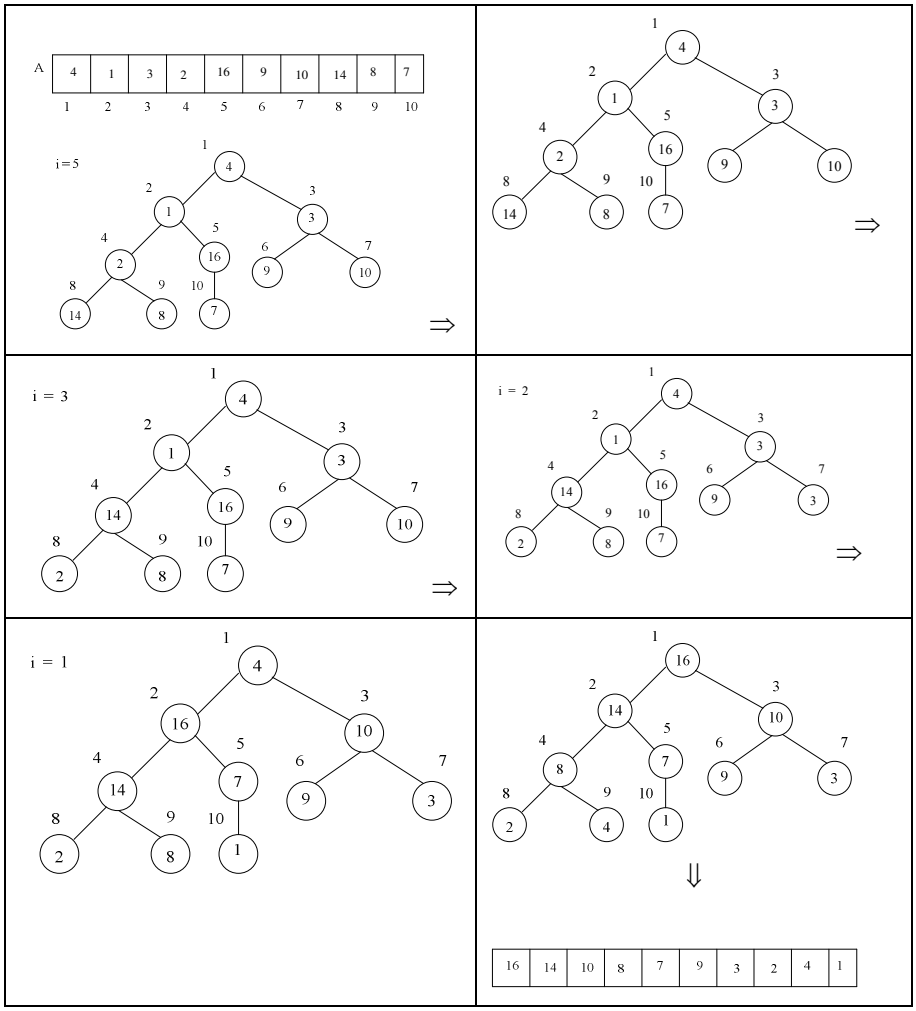
Fiind dat vectorul $A[1,...n]$ se pune problema transformării lui în heap, astfel încât $\text{lungime}[A] = n$.

Din relațiile $(*)$ deducem că elementele $A\left[\frac{n}{2}+1\right],...,A[n]$ sunt frunze în arborele asociat şi pot fi considerate ca heap-uri formate din câte un element.

Algoritmul construieşte_heap va lua în considerație doar elementele $A[1],...,A\left[\frac{n}{2}\right]$ şi va apela algoritmul Reconstituire_heap pentru fiecare din ele.

Se impune ca subarborii având ca rădăcină descendenți ai nodului i, unde $1 \le i \le \left[\frac{n}{2}\right]$, să fie deja heap-uri atunci când se trece la prelucrarea lui i.

```
procedure Construieşte_heap (A)
    dim_heap [A] ← lungime [A]
    for i = [dim_heap [A] / 2] to 1 do
            Reconstitue_heap (A, i)
    next i
end_proc
```



1.4.3. Aplicatii ale structurii heap

1.4.3.1. Algoritmul HeapSort

Algoritmul HeapSort este un exemplu de utilizare a structurii de heap.

Fie vectorul $A[1,...,n]$ ale cărui elemente se vor ordona crescător folosind proprietatea de heap a lui A.

Inițial vom transforma vectorul A în heap. Apelul subprogramului *Construieşte_heap* (A) va aşeza pe poziția $A[1]$ cel mai mare element din vector.

Vom interschimba: $A[1] \leftrightarrow A[n]$.

În continuare vom lua în considerație doar elementele $A[1,...(n-1)]$, iar dim_heap$[A] \leftarrow$ dim_heap$[A]$-1. Subarborii nodului rădăcină $A[1]$ au proprietatea de heap. Eventual, doar nodul $A[1]$ nu îndeplineşte aceasta proprietate. Pentru aceasta se apelează Reconstituire_heap $(A,1)$.

procedure HeapSort (A)

Construieşte_heap (A);

```
    for i = lungime [A] to 2 do
            A[i] ↔ A[1]
            dim_heap[A] ← dim_heap[A] – 1;
            Reconstituire_heap (A,1);
    next i
  end_proc
```

1.4.3.2. Cozi de priorități

O aplicație frecventă a structurii heap o reprezintă cozile de priorități.

Coada de priorități este o structură de date care conține o mulțime S de elemente, fiecare având asociată o cheie.

Operațiile dintr-o coadă de priorități sunt:

- Inserează $(s,x) \Leftrightarrow S \leftarrow S \cup \{x\}$;
- Maxim $(S) \Leftrightarrow$ returnează elementele din S cu cea mai mare cheie;
- Extrage_Max $(S) \Leftrightarrow$ returnează şi elimină elementele din S cu cea mai mare cheie.

O aplicație a cozilor de priorități este planificarea lucrărilor pe calculatoare partajate. Fiecare lucrare are o prioritate. În coada de priorități se memorează lucrările şi prioritățile lor. Când o lucrare se termină sau este suspendată, se extrage din coadă lucrarea cu cea mai mare prioritate şi se execută (subprogramul Extrage_Max). Oricând se poate insera o nouă lucrare în coada de priorități (subprogramul Inserează).

```
  procedure Extrage_Max (A)
    if dim_heap[A] < 1 then
          write "Eroare depăşire inf. heap"
    else
       maxim ← A[1];
       A[1]  ← A[dim_heap[A]]
       dim_heap[A] ← dim_heap[A] – 1;
       Reconstituire_heap (A,1);
       return maxim;
    end if
  end_proc

  procedure Inserează_în_heap (A, cheie)
       dim_heap[A] ← dim_heap[A] + 1
```

$i \leftarrow$ dim_heap [A]

while (i > 1 and A [Părinte (i)] < cheie) **do**

A [i] ← A [Părinte (i)]

i ← Părinte (i)

end_while

A [i] ← cheie

end_proc.

1.5.TABELE DE DISPERSIE

Multe aplicații necesită utilizarea unei mulțimi de elemente *M* asupra căreia se aplică doar operațiile specifice dicționarelor, *Insereaază, caută, Şterge*.

O *tabelă* (*un tablou unidimensional*) este o structură de date eficientă pentru implementarea unui dicționar.

1.5.1. Tabele cu adresare directă

Presupunem că fiecărui element din mulțimea *M* i se asociază o cheie aleasă dintr-un univers $U = \{0,1,...,m-1\}$, unde *m* nu este foarte mare. Vom presupune că nu există două elemente care să aibă aceeaşi cheie.

Pentru a reprezenta mulțimea *M* folosim o *tabelă cu adresare directă*, practic, un tablou $T[0..m-1]$ în care fiecare locație (element de tablou) corespunde unei chei din *U*. Un element din *M* având cheia $k \in U$ este referit de locația *k* din tabelă, adică $T[k]$. Dacă mulțimea *M* nu conține un element cu cheia *k*, atunci $T[k] = Null$.

Operațiile dintr-o tabelă cu adresare deschisă sunt :

function Cauta(*T, k*) : întoarce o valoare din mulțimea *M*
k : cheie
return $T[k]$;
end_func

procedure Insert(*T,x*)
x : element din mulțimea *M*
$T[\text{cheie}(x)] \leftarrow x$;
end_proc

procedure Şterg(*T,x*)
x : element din mulțimea *M*
$T[\text{cheie}(x)] \leftarrow Null$;
end_proc

Observație : Operațiile de mai sus au compexitatea *O(1)*.

1.5.2. Tabele de Dispersie

Utilizarea tabelelor cu adresare directă este uneori nepractică sau chiar imposibilă din punct de vedere al memoriei disponibile. Acesta se întâmplă când

universul *U* este mare (*m* foarte mare). Mai mult, mulțimea *K* a cheilor efectiv memorate poate fi mică relativ la *U* iar majoritatea spațiului alocat pentru tabla *T* să fie irosit.

În cazul $\mathrm{card(K)} << \mathrm{card(U)}$ se folosesc *tabelele de dispersie* care necesită un spațiu de memorie mult mai mic. Prin dispersie, une element $\mathrm{x} \in \mathrm{M}$ având cheia $\mathrm{k} \in \mathrm{U}$ ($\mathrm{cheie(x)} = \mathrm{k}$) este memorat în locația $\mathrm{h(k)}$, unde *h* este *funcția de dispersie*

$$\mathrm{h} : \mathrm{U} \to \{0,1,...\mathrm{m}-1\}.$$

Funcția de dispersie este folosită pentru a calcula locația din tabela *T* pe baza cheii *k*. Mulțimea de indici din tabelă, $\{0,1,...\mathrm{m}-1\}$, este, în general, diferită de universul cheilor.

Spunem că un elemnet cu cheia *k se dispersează* în locația $\mathrm{h}(k)$ sau mai spunem că $\mathrm{h}(k)$ este *valoarea de dispersie* a cheii *k*.

Funcțiile de dispersie reduc simțitor domeniul indicilor tabloului *T* şi deci, dimensiunea acestuia. Există totuşi o problemă. Când $\mathrm{card}(U) > m$ (cazul cel mai frecvent) este clar că două chei diferite se pot dispersa în aceeaşi locație,

$\mathrm{h}(k_1) = h(k_2)$ pentru $k_1 \neq k_2$,

adică are loc o *coliziune*.

1.5.3. Rezolvarea coliziunilor prin înlănțuire

Presupunem că mai multe chei se dispersează în aceeaşi locație de indice *j*, $0 \le j \le m-1$. Putem rezolva această problemă folosind *tabele de dispersie cu înlănțuire*. Pentru astfel de tabele locația $\mathrm{T}[j]$ va conține un indicator spre capul unei liste înlănțuite care conține toate elementele ce se dispersează în locația *j*. Dacă nu există astfel de elemente atunci $\mathrm{T}[j] = Null$. Operațiile de cautare, inserare şi ştergere vor fi următaorele.

function Cauta(*T, x*) : întoarce *true* sau *false*
Caută elementul x în lista $\mathrm{T}[h(cheie(x))]$ *şi întoarce true sau false*
end_func

procedure Insert(*T,x*)
Inserează elementul x în lista $\mathrm{T}[h(cheie(x))]$
end_proc

procedure Şterg(*T,x*)
Şterege elementul x din lista $\mathrm{T}[h(cheie(x))]$
end_proc

1.5.4. Rezolvarea coliziunilor prin adresare deschisă

Prin adresare *deschisă* toate elementele sunt memorate în interiorul tabelei de dispersie. Prin urmare, fiecare locație din tabela de dispersie conține fie un element $\mathrm{x} \in \mathrm{M}$ fie o valoare *Null*.

Pentru a realiza inserarea folosind adresarea deschisă se examinează succesiv locațiile tabelei până se găseşte o locație liberă. Şirul pozițiilor verificate nu este în ordinea *0,1,...m-1*. Acest şir depinde de cheia asociată elementului ce se inserează. Funcția de dispersie va depinde de această cheie şi de poziția curentă din tabelă,

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

În cadrul adresării deschise, pentru o cheie $k \in U$ secvența de verificare este $h(k,0), h(k,1),...,h(k,m-1)$. Operațiile *caută* şi *inserează* sunt următoarele.

function Cauta(*T,x*) : *întoarce o valoare între 0 şi m-1 dacă elementul x a fost găsit şi -1 în caz contrar*

```
k ← cheie(x);
i ← 0;
repeat
   j ← h(k,i);
  if ( T[j] = x ) then
         return j;
  else i ← i+1;
  end if
until (i = m or T[j] = Null);
return -1;
end func
```

function Insereaza(*T,x*) : *întoarce true sau false după cum operația a reuşit sau nu*

```
k ← cheie(x);
i ← 0;
while (I < m)
   j ← h(k,i);
  if ( T[j] = S or T[j] = Null) then
         T[j] = x
         return true
  else i ← i+1;
  end if
end while;
return false;
end func
```

Într-o tabelă de dispersie cu adresare deschisă operația de ştergere este mai deosebită. Când ştergem un element din locația *i* nu se poate marca această locație ca fiind liberă folosind valoarea *Null*. Aceasta ar da peste cap căutarea unui element *x* a cărui inserare anterioară a găsit locația *i* ocupată. De aceea vom marca locația care se şterge printr-o valoare *S*.

function Sterge(*T,x*) : *întoarce o valoare între 0 şi m-1 dacă elementul a fost găsit şi şters şi -1 în caz contrar*

```
k ← cheie(x);
i ← 0;
repeat
```

```
        j ← h(k,i);
        if ( T[j] = x ) then
                T[j] = S;
                return j;
        else i ← i+1;
        end if
    until (i = m or T[j] = Null);
    return -1;
end func
```

1.5.4.1. *Exemple de funcții de dispersie*

Majoritatea funcțiilor de dispersie presupun universul cheilor din mulțimea $U \subset N$.

Metoda divizării

Fie tabela $T[0..m-1]$ şi $k \in U$ o cheie. Atunci,

$h(k) = k \bmod m$

În acest caz se indică $m \neq 2^p$, $p \in N$. În general se alege *m* număr prim care este apropiat de o putere a lui *2*.

Metoda înmulțirii

Fie $0 < A < 1$. Pentru $k \in U$ definim

$h(k) = \lfloor m \cdot (kA - \lfloor kA \rfloor) \rfloor$.

Un exemplu de alegere a valorii A este $A = (\sqrt{5} - 1)/2$.

Dispersia dublă

$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$,

unde h_1 şi h_2 sunt două funcții de dispersie auxiliare

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$ iar $m' < m$.

Poziția de plecare este $T[h_1(k)]$.

Dacă h' este o funcție de dispersie auxiliară se pot utiliza

Verificarea liniară

$h(k,i) = (h'(k) + i \cdot) \bmod m$

sau

Verificarea pătratică

$h(k,i) = (h'(k) + c_1 i \cdot + c_2 i^2) \bmod m$, unde $c_1, c_2 \neq 0$ sunt două constante.

1.6. STRUCTURI DE DATE PE MULȚIMI DISJUNCTE

O structură de date pentru mulțimi disjuncte memorează o colecție $S = \{S_1, S_2, ..., S_n\}$ de mulțimi disjuncte dinamice.

Fiecare mulțime este identificată printr-un reprezentant care este unul dintre elementele mulțimii. În anumite aplicații, nu are importanță care membru este folosit ca reprezentant; important este ca atunci când cerem reprezentantul unei mulțimi de mai multe ori, să primim acelaşi rezultat. În alte aplicații, reprezentantul este ales cel mai mic sau cel mai mare element din mulțime (dacă elementele pot fi ordonate).

Fie x un obiect care poate fi elementul unei mulțimi.

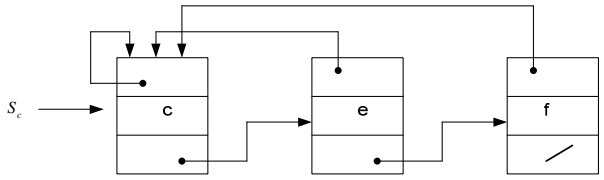
Definim operațiile:

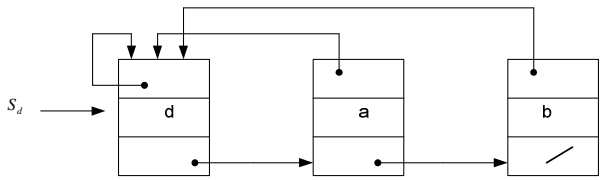
• Formează_Mulțime(x):	creează o nouă mulțime cu un singur element, x. Aceasta va fi şi reprezentantul mulțimii. Este necesar ca x să nu fie deja într-o altă mulțime (mulțimile sunt disjuncte).
• Reuneşte(x, y):	reuneşte mulțimile dinamice S_x şi S_y care conțin pe x şi respectiv pe y. Evident, S_x şi S_y sunt disjuncte. Reprezentantul noii mulțimi $S_x \cup S_y$ poate fi oricare element al acesteia, dar, în practică, se alege sau reprezentantul lui S_x sau cel al lui S_y. După formarea lui $S_x \cup S_y$ se şterg mulțimile S_x şi S_y din colecție.
• Găseşte_Mulțime(x):	returnează reprezentantul unic al mulțimii care îl conține pe x.

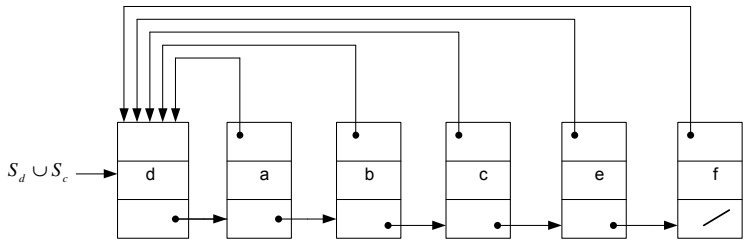
1.6.1. Reprezentarea mulțimilor disjuncte prin liste înlănțuite

Listele înlănțuite reprezintă o modalitate simplă de a reprezenta structura mulțimilor disjuncte. Reprezentantul unei mulțimi se consideră a fi primul obiect din lista corespunzătoare mulțimii.

Un obiect din listă conține un element al mulțimii, o referință (pointer) către următorul element din listă şi o referință către reprezentantul mulțimii.



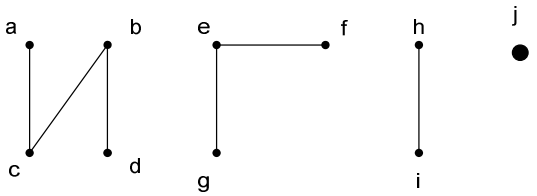




Operațiile Formează_Mulțime şi Găseşte_Mulțime necesită timpul $O(1)$.

O secvență de m operații Formează_Mulțime, Găseşte_Mulțime şi Reuneşte, dintre care $n\,(n < m)$ sunt operații Formează_Mulțime (sau Găseşte_Mulțime), se execută în timpul $\theta\,(m + n \lg n)$.

Aplicație a mulțimilor disjuncte este determinarea componentelor conexe ale unui graf neorientat.



Pentru graful neorientat $G = (V[G], E[G])$, fie $V[G]$ mulțimea vârfurilor, iar $E[G]$ mulțimea muchiilor.

procedure *Componente_Conexe* (G)

for $v \in V[G]$ **do**

Formează_Mulțime (v)

next v

for $(u, v) \in E(G)$ **do**

if Găseşte_Mulțime(u) ≠ Găseşte_Mulțime(v) **then**

Reuneşte (u, v)

end if

next (u, v)

end_procedure

După ce subprogramul Componente_Conexe a fost executat ca un pas de preprocesare, procedura Aceeaşi_Componentă răspunde la întrebarea dacă două vârfuri sunt în aceeaşi componentă conexă.

```
function Aceeaşi_Componentă (u, v)
  if Găseşte_Mulțime (u) = Găseşte_Mulțime (v)
     then return true
     else return false
  end if
end_function
```

Atunci când vârfurile grafului sunt statice – nu se schimbă în timp – componentele conexe pot fi determinate într-un timp mai mic prin algoritmul de căutare rapidă.

Există o situație când muchiile sunt adăugate dinamic şi este necesar să menținem componentele conexe pe măsură ce se adaugă o nouă muchie. Implementarea cu structuri mulțimi disjuncte reprezentate cu liste dinamice este mult mai eficientă în acest caz.