10. Structura de date graf

- În problemele care apar în programare, matematică, inginerie în general şi în multe alte domenii, apare adeseori necesitatea reprezentării unor **relaţii arbitrare** între diferite obiecte, respectiv a **interconexiunilor** dintre acestea.
- Spre exemplu, dându-se **traseele aeriene ale unui stat** se cere să se precizeze drumul optim dintre două oraşe.
    - Criteriul de optimalitate poate fi spre exemplu timpul sau preţul, drumul optim putând să difere pentru cele două situaţii.
- **Circuitele electrice** sunt alte exemple evidente în care interconexiunile dintre obiecte joacă un rol central.
    - Piesele (tranzistoare, rezistenţe, condensatoare) sunt interconectate prin fire electrice.
    - Astfel de circuite pot fi reprezentate şi prelucrate de către un sistem de calcul în scopul rezolvării unor probleme simple cum ar fi: "*Sunt toate piesele date conectate în acelaşi circuit*?" sau a unor probleme mai complicate cum ar fi: "*Este funcţional un anumit circuit electric*?".
- Un al treilea exemplu îl reprezintă **planificarea activităţilor**, în care obiectele sunt task-uri (activităţi, procese) iar interconexiunile precizează care dintre activităţi trebuiesc finalizate înaintea altora.
    - Întrebarea la care trebuie să ofere un răspuns este: "*Când trebuie planificată fiecare activitate?*".
- **Structurile de date** care pot modela în mod natural situaţii de natura celor mai sus prezentate sunt cele derivate din **conceptul matematic** cunoscut sub denumirea de **graf**.
- **Teoria grafurilor** este o ramură majoră a matematicii combinatorii care în timp a fost şi este încă intens studiată.
    - Multe din proprietăţile importante şi utile ale grafurilor au fost demonstrate, altele cu un grad sporit de dificultate îşi aşteaptă încă rezolvarea.
- În cadrul capitolului de faţă vor fi prezentate doar câteva din proprietăţile fundamentale ale grafurilor în scopul înţelegerii algoritmilor fundamentali de prelucrare a structurilor de date graf.
    - Ca şi în multe alte domenii, studiul algoritmic al grafurilor respectiv al structurilor de date graf, este de dată relativ recentă astfel încât alături de algoritmii fundamentali cunoscuţi de mai multă vreme, mulţi dintre algoritmii de mare interes au fost descoperiţi în ultimii ani [Se88].

10.1. Definiţii

- Un graf, în cea mai largă accepţiune a termenului, poate fi definit ca fiind o colecţie de **noduri** şi **arce**.

- Un **nod** este un **obiect** care poate avea un nume şi eventual alte proprietăţi asociate.
- Un **arc** este o **conexiune** neorientată între două noduri.

- Notând cu `N` mulţimea nodurilor şi cu `A` mulţimea arcelor, un graf `G` poate fi precizat formal prin enunţul `G=(N,A)`.
  - În figura 10.1.a. (a),(b) apar două exemple de grafuri.

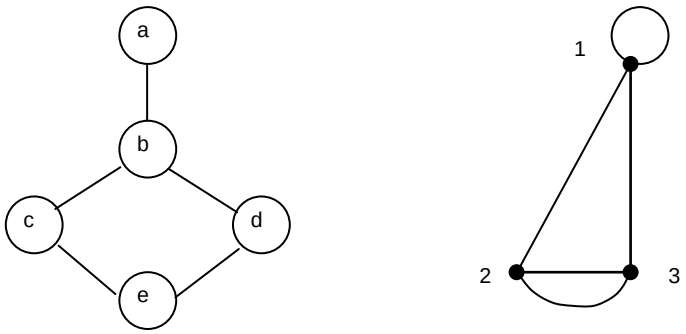


Fig.10.1.a. Exemple de grafuri

- **Ordinul** unui graf este numărul de noduri pe care acesta le conţine şi se notează cu `|G|`.
- **Arcele** definesc o **relaţie de incidenţă** între perechile de noduri.
- Două noduri conectate printr-un arc se numesc **adiacente**, altfel ele sunt **independente**.
- Dacă `a` este un **arc** care leagă nodurile `x` şi `y (a~(x,y))`, atunci `a` este **incident** cu x,y.
  - Singura proprietate presupusă pentru această relaţie este **simetria** [10.1.a].

`(x,y)∈A =>(y,x)∈A` unde `A` este mulţimea arcelor. `[10.1.a]`

- Un graf poate fi definit astfel încât să aibă un arc `a~(x,x)`.
  - Un astfel de arc se numeşte **buclă** ("**loop**").
  - Dacă relaţia de incidenţă este **reflexivă**, atunci fiecare nod conţine o astfel de buclă.
- Există şi posibilitatea ca să existe mai multe arce care conectează aceeaşi pereche de noduri. Într-un astfel de caz se spune că cele două noduri sunt conectate printr-un **arc multiplu**.
  - În figura 10.1.a (b) este reprezentat un graf cu buclă şi arc multiplu.
- Grafurile în care nu sunt acceptate arce multiple se numesc **grafuri simple**.
- Numărul de arce incidente unui nod reprezintă **gradul** nodului respectiv.
- Se numeşte **graf regulat** acel graf în care toate nodurile sunt de acelaşi grad.
- Se numeşte **graf complet de ordinul** `n` şi se notează cu K_n, acel graf în care fiecare pereche de noduri este adiacentă.
  - În figura 10.1.b. apar reprezentate grafurile complete până la ordinul 6.

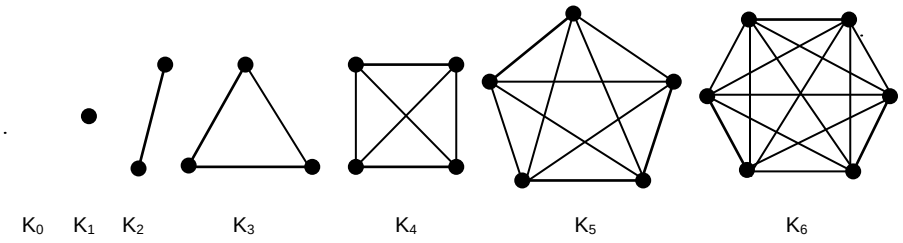


Fig.10.1.b. Exemple de grafuri complete

- Un **graf** se numeşte **planar** dacă el poate fi astfel reprezentat într-un plan, încât oricare două arce ale sale se intersectează numai în noduri [GG78].
- O teoremă demonstrată de Kuratowski precizează că orice **graf neplanar** conţine cel puţin unul din următoarele grafuri de bază:
  - (1) **5-graful complet** (K_5), sau
  - (2) **Graful utilitar** în care există două mulţimi de câte trei noduri, fiecare nod fiind conectat cu toate nodurile din cealaltă mulţime.

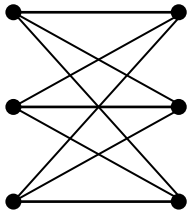


Fig.10.1.c. Graf utilitar

- Un **graf** se numeşte **bipartit** dacă nodurile sale pot fi partiţionate în două mulţimi distincte N_1 şi N_2 astfel încât orice arc al său conectează un nod din N_1 cu un nod din N_2.
  - Spre exemplu **graful utilitar** este în acelaşi timp un **graf bipartit.**
- Fie `G=(N,A)` un graf cu mulţimea nodurilor `N` şi cu mulţimea arcelor `A`. Un **subgraf** al lui `G` este graful `G'=(N',A')` unde:
  - (1) `N'` este o submulţime a lui `N`.
  - (2) `A'` constă din arce `(x,y)` ale lui `A`, astfel încât `x` şi `y` aparţin lui `N'`.
- În figura 10.1.d apare un exemplu de **graf** (a) şi un **subgraf** al său (b).

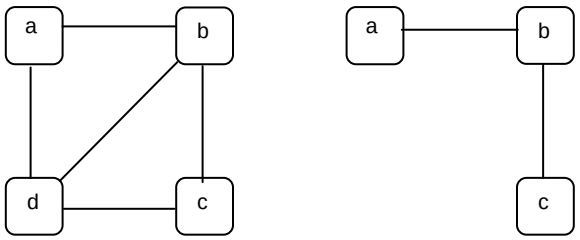


Fig.10.1 d. Graf şi un subgraf al său

- Dacă mulţimea de arce `A'` conţine **toate arcele** `(x,y)` ale lui `A` pentru care atât `x` cât şi `y` sunt în `N'`, atunci `G'` se numeşte **subgraf indus** al lui `G` [AH85].

- Un **graf** poate fi **reprezentat** în **manieră grafică** marcând nodurile sale şi trasând linii care materializează arcele.
    - Reprezentarea unui graf nu este unică. Spre exemplu fig.10.1.e (a) respectiv (b) reprezintă unul şi acelaşi graf.

- În acelaşi timp însă, un **graf** poate fi conceput ca şi un **tip de date abstract**, independent de o anumită reprezentare.
    - Un graf, poate fi definit spre exemplu precizând doar **mulţimea nodurilor** şi **mulţimea arcelor** sale.

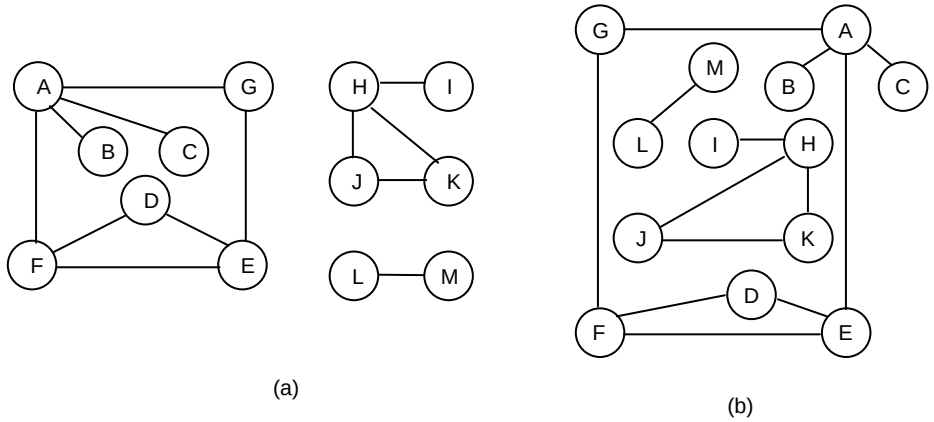


Fig.10.1.e. Reprezentări echivalente ale unui grat

- În anumite aplicaţii, cum ar fi exemplul cu traseele aeriene, poziţia nodurilor (oraşelor) este precizată fizic prin amplasarea lor pe harta reală a statului, rearanjarea structurii fiind lipsită de sens.

- În alte aplicaţii însă, cum ar fi planificarea activităţilor, sunt importante nodurile şi arcele ca atare independent de dispunerea lor geometrică.
    - În cadrul capitolului de faţă vor fi abordaţi algoritmi generali, care prelucrează colecţii de noduri şi arce, făcând abstracţie de dispunerea lor geometrică, cu alte cuvinte făcând abstracţie de **topologia** grafurilor.

- Se numeşte **drum** (“**path**”) de la nodul `x` la nodul `y`, aparţinând unui graf, o **secvenţă de noduri** $n_1, n_2, \ldots, n_j$ în care nodurile succesive sunt conectate prin arce aparţinând grafului.
    - **Lungimea** unui drum este egală cu numărul de arce care compun drumul.
    - La limită, un singur nod precizează un drum la el însuşi de lungime zero.

- Un **drum** se numeşte **simplu** dacă toate nodurile sale, exceptând eventual primul şi ultimul sunt distincte.

- Un **ciclu** (**buclă**) este un drum simplu de lungime cel puţin 1, care începe şi se sfârşeşte în acelaşi nod.

- Dacă există un drum de la nodul `x` la nodul `y` se spune că acel drum **conectează** cele două noduri, respectiv nodurile `x` şi `y` sunt **conectate**.

- Un **graf** se numeşte **conex**, dacă de la fiecare nod al său există un drum spre oricare alt nod al grafului, respectiv dacă oricare pereche de noduri aparţinând grafului este conectată.

- Intuitiv, dacă nodurile se consideră obiecte fizice, iar conexiunile fire care le leagă, atunci un **graf conex** rămâne unitar, indiferent de care nod ar fi "suspendat în aer".

- Un graf care nu este conex este format din **componente conexe.**
  - Spre exemplu**,** graful din fig.10.1.a este format din trei componente conexe.
- O **componentă conexă** a unui graf G este de fapt un **subgraf indus maximal conectat** al său [AH 85].
- Un **graf** se numeşte **ciclic** dacă conţine cel puţin un ciclu.
  - Un **ciclu** care include **toate arcele grafului** o singură dată se numeşte **ciclu eulerian** (**hamiltonian**).
  - Este uşor de observat că un asemenea ciclu există numai dacă graful este conex şi gradul fiecărui nod este par.
- Un **graf conex aciclic** se mai numeşte şi **arbore liber**.
  - În fig.10.1.f apare un graf constând din două componente conexe în care fiecare componentă conexă este un arbore liber.
  - Se remarcă în acest sens, observaţia ca **arborii** sunt de fapt cazuri particulare ale **grafurilor**.
  - Un grup de arbori neconectaţi formează o **pădure** ("**forest**").

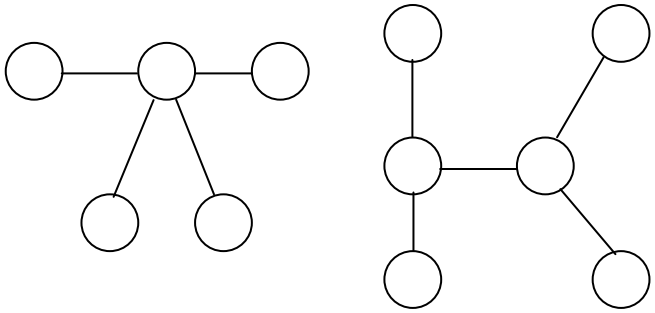


Fig.10.1.f. Graf aciclic format din două componente conexe

- Un **arbore de acoperire** ("**spanning tree**") al unui graf, este un **subgraf** care conţine toate nodurile grafului iniţial, dar dintre conexiuni numai atâtea câte sunt necesare formării unui arbore.
  - Se face precizarea că termenul de "**acoperire**" în acest context are sensul termenului "**cuprindere**".
- În figura 10.1.g este prezentat un graf (a) şi un arbore de acoperire al grafului (b).

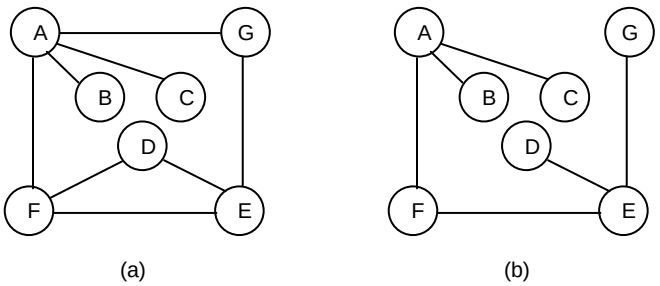


Fig.10.1.g. Graf şi un arbore de acoperire al grafului

- Un **arbore liber** poate fi transformat într-un **arbore ordinar** dacă se "suspendă" arborele de un nod considerat drept **rădăcină** şi se orientează arcele spre rădăcină.
- **Arborii liberi** au două proprietăţi importante:
  - (1) Orice **arbore liber** cu `n` noduri conţine exact `n-1` arce (câte un arc la fiecare nod, mai puţin rădăcina).
  - (2) Dacă unui **arbore liber** i se adaugă un arc el devine obligatoriu un **graf ciclic**.
- De aici rezultă două **consecinţe** importante şi anume:
  - (1) Un graf cu `n` noduri şi mai puţin de `n-1` arce nu poate fi conex.
  - (2) Pot exista grafuri cu `n` noduri şi `n-1` arce care nu sunt arbori liberi. (Spre exemplu dacă au mai multe componente conexe).
- Notând cu `n` numărul de **noduri** ale unui graf şi cu `a` numărul de **arce**, atunci `a` poate lua orice valoare între `0` şi `(1/2)n(n-1)`.
  - Graful care conţine toate arcele posibile este **graful complet** de ordinul `n` (K_n).
  - Graful care are relativ puţine arce (spre exemplu $a<n\log_2 n$) se numeşte **graf rar** ("**sparse**").
  - Graful cu un număr de arce apropiat de graful complet se numeşte **graf dens**.
- Dependenţa fundamentală a **topologiei unui graf** de doi parametri (`n` şi `a`), face ca studiul comparativ al algoritmilor utilizaţi în prelucrarea grafurilor să devină mai complicat din cauza posibilităţilor multiple care pot să apară.
  - Astfel, presupunând că un algoritm de prelucrare a unui graf necesită un efort de calcul de ordinul $O(n^2)$ în timp ce un alt algoritm care rezolvă aceeaşi problemă necesită un efort de ordinul $O((n+a)\log_2 n)$ paşi, atunci, în cazul unui graf cu `n` noduri şi `a` arce, este de preferat primul algoritm dacă graful este dens, respectiv al doilea dacă graful este rar.
- Grafurile prezentate până în prezent se numesc şi **grafuri neorientate** şi ele reprezintă cea mai simplă categorie de grafuri.
- Prin asocierea de informaţii suplimentare nodurilor şi arcelor, se pot obţine categorii de grafuri mai complicate.
- Astfel, într-un **graf ponderat** ("**weighted graph**"), fiecărui arc i se asociază o valoare (de regulă pozitivă) numită **pondere** care poate reprezenta spre exemplu o distanţă sau un cost.
- În cadrul **grafurilor orientate** ("**directed graphs**"), arcele sunt orientate, având un sens precizat, de la x la y spre exemplu.
  - În acest caz `x` se numeşte coada sau sursa arcului iar `y` vârful sau destinaţia sa.
  - Pentru reprezentarea arcelor orientate se utilizează săgeţi sau segmente direcţionate (fig.10.1.h. (a), (b),(c)).
- **Grafurile orientate ponderate** se mai numesc şi **reţele** ("**networks**").
- Informaţiile suplimentare referitoare la noduri şi arce nuanţează şi în acelaşi timp complică manipularea grafurilor care le conţin.

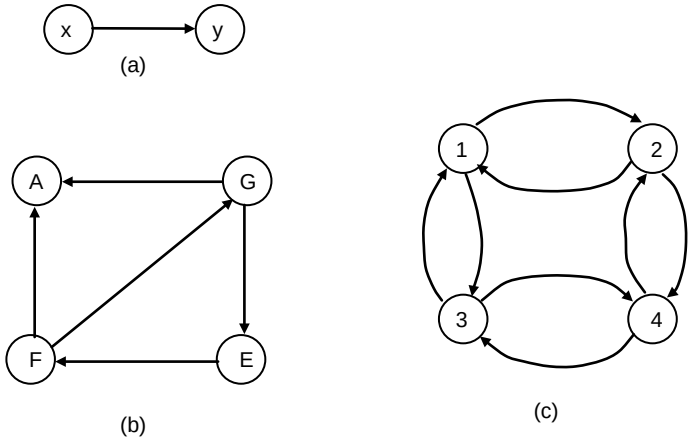


Fig.10.1.h. Grafuri orientate

10.2. Tipul de date abstract graf

- Pentru a defini **tipul de date abstract** (TDA) **graf** este necesară:
  - (1) Precizarea **modelului matematic** care conturează conceptul de graf prezentat în paragraful anterior.
  - (2) Precizarea **setului de operatori** definiţi pe acest model.
- În cele ce urmează se prezintă două variante de definire a unui **TDA graf**, una extinsă şi alta mai restrânsă**.**

10.2.1. TDA graf. Varianta 1 (Shiflet)

- În cadrul variantei Shiflet [Sh90] un **graf** este considerat ca şi o structură de noduri şi arce.
  - Fiecare nod are o cheie care identifică în mod univoc nodul.
  - Modelul matematic, notaţiile utilizate şi setul de operatori preconizat pentru această variantă apar în [10.2.1.a]:

```
------------------------------------------------------------
TDA Graf    (Varianta 1 - Shiflet)              [10.2.1.a]

Modelul matematic: graful definit în sens matematic.

Notaţii:
     TipGraf      - tipul de date abstract graf;
     TipElement   - tipul asociat porţiunii element a nodului
     TipCheie     - tipul asociat porţiunii cheie a unui
                    element
     TipInfo      - tipul corespunzator porţiunii de
                    informaţie a unui element
     TipIndicNod  - tip referinţă la structura unui nod
     TipIndicArc  - tip referinţă la structura unui arc

Operatori:
```

1. ***InitGraf****(TipGraf g)* - procedură care creează graful vid *g;*

2. *boolean* ***GrafVid****(TipGraf g)* - operator care returnează **true** dacă graful este vid respectiv **false** în caz contrar;

3. *boolean* ***GrafPlin****(TipGraf g)* - operator boolean care returnează **true** dacă graful este plin. Se precizează faptul că această funcţie este legată direct de maniera de implementare a grafului. Valoarea ei adevărată presupune faptul că zona de memorie alocată structurii a fost epuizată şi în consecinţă nu se mai pot adăuga noi noduri.

4. *TipCheie* ***CheieElemGraf****(TipGraf g, TipElement e)* - operator care returnează cheia elementului *e* aparţinând grafului *g*.

5. *boolean* ***CautaCheieGraf****(TipGraf g, TipCheie k)* - operator boolean care returnează valoarea adevărat dacă cheia *k* este găsită în graful *g*.

6. ***IndicaNod****(TipGraf g, TipCheie k, TipIndicNod* indicNod)* - operator care face ca *IndicNod* să indice acel nod din *g* care are cheia *k*, presupunând că un astfel de nod există.

7. ***IndicaArc****(TipGraf g, TipCheie k1, TipCheie k2, TipIndicArc* indicArc)* - operator care face ca *indicArc* să indice arcul care conectează nodurile cu cheile k_1 şi k_2 din graful *g*, presupunând că arcul există. În caz contrar *indicArc* ia valoarea indicatorului vid.

8. *boolean* ***ArcVid****(TipGraf g, TipIndicArc indicArc)* - operator boolean care returnează valoarea adevărat dacă arcul indicat de *indicArc* este vid.

9. ***InserNod****(TipGraf* g, TipElement e)* - operator care inserează un nod *e* în graful *g* ca un nod izolat (fără conexiuni). Se presupune că înaintea inserţiei în *g* nu există nici un nod care are cheia identică cu cheia lui *e*.

10. ***InserArc****(TipGraf* g, TipCheie k1, TipCheie k2)*- operator care inserează în *g* un arc incident nodurilor având cheile k_1 şi k_2. Se presupune că cele două noduri există şi că arcul respectiv nu există înaintea inserţiei.

11. ***SuprimNod****(TipGraf* g, TipIndicNod indicNod)* - operator care suprimă din *g* nodul precizat de *indicNod*, împreună cu toate arcele incidente. Se presupune că înaintea suprimării, un astfel de nod există.

12. ***SuprimArc****(TipGraf* g, TipIndicArc indicArc)* - operator care suprimă din *g*, arcul precizat de *indicArc*. Se

presupune că înaintea suprimării un astfel de arc există.

13. ***ActualizNod****(TipGraf* g, TipIndicNod indicNod, TipInfo x)* - operator care plasează valoarea lui x în porţiunea "informaţie" a nodului indicat de *indicNod* din graful *g*. Se presupune că *indicNod* precizează un nod al grafului.

14. *TipElement* ***FurnizeazaNod****(TipGraf g, TipIndicNod indicNod)* - operator care returnează valoarea elementului memorat în nodul indicat de *indicNod* în graful *g*.

15. ***TraversGraf****(TipGraf g, Vizită (ListăArgumente))* - operator care realizează traversarea grafului *g*, executând pentru fiecare element al acestuia procedura *Vizită(ListaArgumente)*, unde *Vizită* este o procedură specificată de utilizator iar *ListăArgumente* este lista de parametri a acesteia.

10.2.2. TDA graf. Varianta 2 (Decker)

- În viziunea lui Rick Decker [De89], **tipul de date abstract graf** constă:
  - (1) Dintr-o mulţime Poziţii care materializează mulţimea nodurilor grafului.
  - (2) O mulţime Atomi, care materializează conţinuturile nodurilor grafului.
  - (3) O relaţie **R** astfel încât p**R**q (unde p,q aparţin mulţimii Poziţii) precizează existenţa unui arc care conectează nodurile p şi q.
- Drept urmare un **graf** poate fi definit drept o **tripletă** (P,R,f) unde:
  - P este o **submulţime** finită a mulţimii Poziţie.
  - f este o **funcţie** definită pe P cu valori în mulţimea Atomi.
  - R este **relaţia simetrică** în P (şi ireflexivă dacă nu se permit bucle), definită anterior.
- De regulă mulţimea poziţiilor (nodurilor) unui graf g se precizează prin notaţia N(g).
- Deşi nu este absolut necesar, se poate defini mulţimea Arce sau A(g) ca fiind mulţimea tuturor mulţimilor formate din două elemente {p,q} pentru care p**R**q.
- În acest context, **TDA graf** este definit după cum urmează [10.2.2.a.].

TDA Graf (Varianta 2 - Decker) [10.2.2.a]

Modelul matematic: graful definit în sensul precizat în paragraful &10.2.2.

Notatii:

TipGraf - tipul de date abstract graf
TipPozitie - tipul asociat nodurilor grafului
TipAtom - tipul asociat porţiunii de informaţii a

unui nod
TipArc - tipul asociat arcelor grafului

Operatori:

1. ***Crează****(TipGraf g)* – operator care crează graful vid *g*.
2. *boolean* ***Adiacent****(TipPozitie p, TipPozitie q, TipGraf g)* – operator care returnează valoarea **true** dacă şi numai dacă *p***R***q*, adică dacă în graf există un arc de la nodul *p* la nodul *q*.
3. ***Modifică****(TipAtom a, TipPozitie p, TipGraf g)* – modifică atomul asociat poziţiei *p* făcând *f(p)=a*. Cu alte cuvinte, zonei de date a nodului indicat de $p \in N(g)$ i se conferă valoarea *a*.
4. *TipAtom* ***Furnizează****(TipPozitie p, TipGraf g)* – operator care returnează valoarea *f(p)*, cu alte cuvinte zona de date a nodului indicat de $p \in N(g)$.
5. ***SuprimNod****(TipPozitie p, TipGraf g)* - operator care suprimă nodul indicat de *p* din *g* împreună cu toate arcele incidente. Se consideră valabilă precondiţia $p \in N(g)$.
6. ***SuprimArc****(TipArc e, TipGraf g)* - operator care extrage pe *e* din mulţimea arcelor lui *g*. Se consideră valabilă precondiţia $e \in A(g)$.
7. ***InserNod****(TipPozitie p, TipGraf g)* – operator care adaugă pe *p* mulţimii poziţiilor lui *g* fără a-l modifica pe R. Cu alte cuvinte, nodul indicat de *p* se adaugă la *g*, fără nici un arc de conexiune cu un alt nod al grafului. Se presupune valabilă precondiţia $p \notin N(g)$.
8. ***InserArc****(TipArc e, TipGraf g)* – operator care include pe *e* în mulţimea arcelor lui *g* modificând relaţia structurală R. Această operaţie presupune iniţial valabile următoarele afirmaţii:
   a) dacă $e=\{p,q\}$ atunci $p,q \in N(g)$ şi
   b) $e \notin A(g)$.

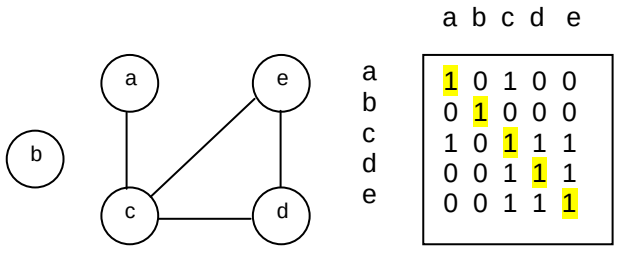
10.3. Tehnici de implementare a tipului de date abstract graf

- În vederea prelucrării grafurilor concepute ca şi tipuri de date abstracte (TDA) cu ajutorul sistemelor de calcul, este necesară la primul rând stabilirea **modului lor de reprezentare.**
- Această activitate constă de fapt din desemnarea unei structuri de date concrete care să materializeze în situaţia respectivă tipul de date abstract graf.
- În cadrul paragrafului de faţă se prezintă mai multe posibilităţi, alegerea depinzând, ca şi pentru marea majoritate a tipurilor de date deja studiate:
  - (1) De natura grafurilor de implementat.
  - (2) De natura şi frecvenţa operaţiilor care se execută asupra lor.

- În esenţă se cunosc **două modalităţi majore** de implementare a grafurilor: una bazată pe **matrici de adiacenţe** iar celalată bazată pe **structuri de adiacenţe**.

10.3.1. Implementarea grafurilor cu ajutorul matricilor de adiacenţe

- Cel mai direct mod de reprezentare al unui tip de date abstract graf îl constituie **matricea de adiacenţe** (“**adjacency matrix**”).
- Dacă se consideră graful `G=(N,A)` cu mulţimea nodurilor `N={1,2,...,n}`, atunci **matricea de adiacenţe** asociată grafului `G`, este o matrice `A[n,n]` de elemente booleene, unde `A[x,y]` este adevărat dacă şi numai dacă în graf există un arc de la nodul `x` la nodul `y`.
    - Adesea elementelor booleene ale matricii sunt înlocuite cu întregii 1 (adevărat), respectiv 0 (fals).
- Primul pas în reprezentarea unui graf printr-o matrice de adiacenţe constă în stabilirea unei **corespondenţe** între **numele nodurilor** şi **mulţimea indicilor matricei**.
    - Această corespondenţă poate fi realizată:
        - (1) În **mod implicit** prin alegerea corespunzătoare a tipului de bază al mulţimii N.
        - (2) În **mod explicit** prin precizarea unei **asocieri** definite pe **mulţimea nodurilor** cu valori în **mulţimea indicilor** matricei.
- În cazul **corespondenţei implicite** cel mai simplu mod de implementare constă în “a denumi” nodurile cu **numere întregi** care coincid cu indicii de acces în **matricea de adiacenţe**.
    - Numele nodurilor pot fi de asemenea litere consecutive ale alfabetului sau în cazul limbajului Pascal, constante ale unui tip enumerare definit de utilizator, în ambele situaţii existând posibilitatea conversiei directe a tipurilor respective în tipul întreg prin funcţii specifice de limbaj.
- În cazul **corespondenţei explicite**, pentru implementarea asocierii pot fi utilizate:
    - Tehnici specifice simple cum ar fi cele bazate pe tablouri sau liste.
    - Tehnici mai avansate bazate spre exemplu pe arbori binari sau pe metoda dispersiei.
- Pentru o urmărire facilă a algoritmilor, în cadrul capitolului de faţă, se va utiliza o **metodă implicită** conform căreia nodurile vor avea numele format dintr-o singură literă.
- În figura 10.3.1.a. apare reprezentarea bazată pe matrice de adiacenţe (b) a grafului (a).



(a) (b)

Fig.10.3.1.a. Graf (a) reprezentat prin matrice de adiacenţe (b).

- Se observă faptul că reprezentarea are un **caracter simetric** întrucât fiind vorba despre un **graf neorientat**, arcul care conectează nodul `x` cu nodul `y` este reprezentat prin **două valori** în matrice: `A[x,y]` respectiv `A[y,x]`.
    - În astfel de situaţii, deoarece matricea de adiacenţe este simetrică, ea poate fi memorată pe jumătate, element care pe lângă avantaje evidente are şi dezavantaje.
        - Pe de-o parte **nu** toate limbajele de programare sunt propice unei astfel de implementări.
        - Pe de altă parte **algoritmii** care prelucrează astfel de matrici sunt mai complicaţi decât cei care prelucrează matrici integrale.
- În prelucrarea grafurilor se poate face presupunerea că un nod este conectat cu el însuşi, element care se reflectă în valoarea "adevărat" memorată în toate elementele situate pe diagonala principală a matricei de adiacenţe.
    - Acest lucru nu este însă obligatoriu şi poate fi reconsiderat de la caz la caz.
- În continuare se prezintă două studii de caz pentru implementarea TDA graf cu ajutorul matricilor de adiacenţe.

10.3.1.1. Studiu de caz 1.

- După cum s-a precizat, un graf este definit prin **două mulţimi**: **mulţimea nodurilor** şi **mulţimea arcelor** sale.
- În vederea prelucrării, un astfel de graf trebuie furnizat drept dată de intrare algoritmului care realizează această activitate.
- În acest scop, este necesar a se preciza modul în care se vor introduce în memoria sistemului de calcul elementele celor două mulţimi.
- (1) O posibilitate în acest sens o reprezintă **citirea directă**, ca dată de intrare a **matricii de adiacenţe**, metodă care nu convine în cazul matricilor rare.
- (2) O altă posibilitate o reprezintă următoarea:
    - În prima etapă se citesc **numele nodurilor** în vederea asocierii acestora cu indicii matricei de adiacenţe.
    - În etapa următoare, se citesc **perechile de nume de noduri** care definesc **arce** în cadrul grafului.
        - Pornind de la aceste perechi se generează **matricea de adiacenţe.**
    - Se face precizarea că prima etapă poate să lipsească dacă în implementarea asocierii se utilizează o **metodă implicită**.
- În secvenţa [10.3.1.1.a] apare un exemplu de program pentru **crearea** unei matrice de adiacenţe.
    - Corespondenţa nume nod-indice este realizată implicit prin funcţia ***`index(n)`***, care are drept parametru numele nodului şi returnează indicele acestuia.
    - Din acest motiv, prima etapă se reduce la citirea valorilor `N` şi `A` care reprezintă numărul de noduri, respectiv numărul de arce ale grafului.

- Ordinea în care se furnizează perechile de noduri în etapa a doua nu este relevantă, întrucât matricea de adiacenţe nu este în nici un mod influenţată de această ordine.

```
-------------------------------------------------------------
/*Cazul 1. Implementarea TDA Graf utilizând matrici de
adiacenţe - varianta pseudocod*/

int maxN = 50; /* N = numărul maxim de noduri*/
int j,x,y;
int N; /* N = numărul curent de noduri*/
int A; /* A = numărul curent de arce*/

Tip_Nod n1,n2;
boolean graf[maxN,maxN]; /* matricea de adiacenţe */


  *citeste(N,A);                              [10.3.1.1.a]
  /*iniţializare matrice de adiacenţe*/
  pentru (x=1 la N)
    pentru (y=1 la N)
      graf[x,y]= false;
  /*iniţializare diagonală principală a MA*/
  pentru (x=1 la N)
    graf[x,x]= true;
  /*construcţie matrice de adiacenţe pentru graf*/
  pentru (j=1 la A)
    |*citeste(n1,n2);
    |x=index(n1); y=index(n2);
    |graf[x,y]=true;
    |graf[y,x]=true
    |□ /*pentru*/
/*Creare matrice de adiacenţe*/
-------------------------------------------------------------
```

- După cum se observă în cadrul secvenţei, tipul variabilelor `n1` şi `n2` este `TipNod` care nu este precizat.
- De asemenea nu este precizat nici codul aferent funcţiei ***index***, acestea depinzând direct de maniera de reprezentare a nodurilor.
  - Spre exemplu `n1` şi `n2` pot fi de tip caracter (int) iar funcţia ***index*** o expresie de forma `n1-`a´`.

10.3.1.2. Studiu de caz 2

- Studiul de caz 2 prezintă o metodă mai elaborată de implementare a unui TDA graf.
- Reprezentarea presupune definirea tipurilor şi structurilor de date în conformitate cu secvenţa [10.3.1.2.a].

```
-------------------------------------------------------------
/*Cazul 2. Implementarea TDA Graf utilizând matrici de
adiacenţe - varianta C*/

const Numar_Noduri = .......;

typedef Tip_Cheie .......;
```

```
typedef Tip_Info  .......;

/*definire tip structură element (nod)*/
typedef struct Element
     {
        Tip_Cheie Cheie;
        Tip_Info Info;
     } Tip_Element;

/*definire tip tablou elemente*/
typedef Tip_Element Tip_Tablou_Elemente[Numar_Noduri];

/*definire tip matrice de adiacenţe*/
typedef boolean Tip_Matr_Adj[Numar_Noduri,Numar_Noduri];

/*definire tip structură graf*/
typedef struct Graf
     {
        int contor;                          /*[10.3.1.2.a]*/
        Tip_Tablou_Elemente noduri;
        Tip_Matr_Adj arce;
     } Tip_Graf

/*definire structură arc*/
 typedef struct Arc
     {
        int linie;
        int coloana;
     } Tip_Arc

Tip_Graf  g;
Tip_Cheie  k,k1,k2;
Tip_Element  e;
int indicNod;
Tip_Arc indicArc;
-----------------------------------------------------------------
```

- În accepţiunea acestei reprezentări, **graful** din fig. 10.3.1.2.a.(a) va fi implementat prin următoarele elemente:
  - (1) `contor` – care precizează numărul de noduri.
  - (2) `noduri` – tabloul care păstrează nodurile propriu-zise.
  - (3) `arce` – matricea de adiacenţe a grafului (fig.10.3.1.2.a.(b)).

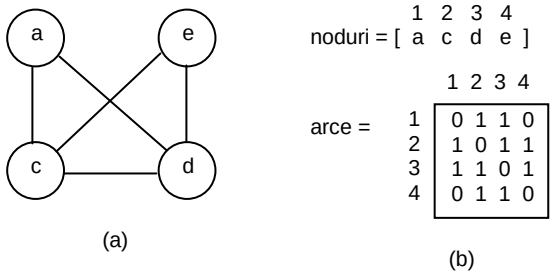
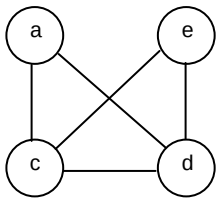


Fig.10.3.1.2.a. Reprezentarea elaborată a unui graf utilizând matrice de adiacenţe

- În anumite situaţii, pentru simplificare, nodurile nu conţin alte informaţii în afara cheii, caz în care `Tip_Element` = `Tip_Cheie`.
    - Alteori nodurile nu conţin nici un fel de informaţii (nici măcar cheia) situaţie în care interesează numai **numele nodurilor** în vederea identificării lor în cadrul reprezentării.
- În continuare se fac unele **consideraţii** referitoare la implementarea în acest context a setului de operatori extins (Varianta 1, (Shiflet)).
- (1) Operatorii ***InitGraf***, ***GrafVid***, ***GrafPlin***, împreună cu ***InserNod*** şi ***SuprimNod*** se referă în regim de consultare sau modificare la contorul care păstrează **numărul nodurilor grafului:** `g.contor`.
- (2) Informaţia conţinută de tabloul `noduri` poate fi **ordonată** sau **neordonată**.
    - Dacă tabloul `noduri` este **ordonat**, localizarea unei chei în cadrul operatorilor ***CautăCheieGraf*** sau ***IndicaNod*** se poate realiza prin tehnica **căutării binare.**
    - Dacă tabloul `noduri` este **neordonat** localizarea unei chei se poate realiza prin tehnica **căutării liniare**.
    - Operatorul ***CautăCheieGraf*** indică numai dacă cheia este prezentă sau nu în graf.
    - Operatorul ***IndicaNod****(TipGraf g, TipCheie k, TipIndicNod * indicNod)* asignează lui ***IndicNod*** indexul nodului din graful `g`, care are cheia egală cu `k`.
    - Operatorul ***IndicaArc****(TipGraf g, TipCheie k1, TipCheie k2, TipArc * indicArc)* se comportă într-o manieră similară, returnând indexul nodului `k1` în variabila de ieşire `indicArc.linie` şi indexul nodului `k2` în `indicArc.coloană`.
- (3) **Inserţia** unui nod depinde de asemenea de maniera de organizare a datelor în cadrul tabloului `noduri`.
    - (a) Dacă `noduri` este un **tablou neordonat**, se incrementează `g.contor` şi se memorează nodul de inserat în `g.noduri[g.contor]`.
        - După cum rezultă din fig.10.3.1.2.b, care reprezintă inserţia nodului `b` în graful (a), nodul nou introdus este izolat, adică în matricea de adiacenţe se introduce valoarea fals pe linia `g.contor` şi pe coloana `g.contor` (d).



(a)

contor = 4

	1	2	3	4
noduri = [	a	c	d	e]

arce =

	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	1
4	0	1	1	0

(b)

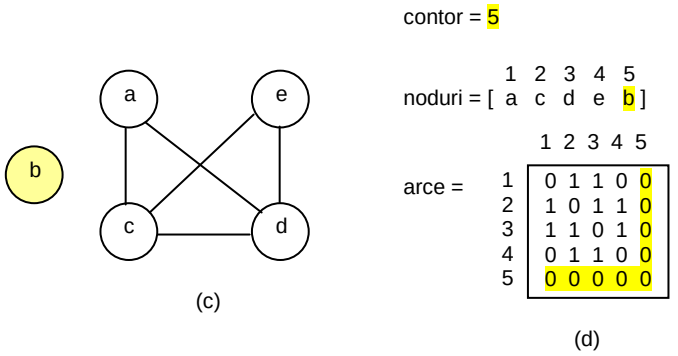


Fig.10.3.1.2.b. Inserţia unui nod într-un graf (varianta tablou `noduri` neordonat)

- Procedura efectivă de inserţie a unui nod nou în acest context apare în secvenţa [10.3.1.2.b].

```
-------------------------------------------------------------
/*Inserţia unui nod. (Tabloul noduri neordonat) - varianta
pseudocod*/

procedure InserNod(Tip_Graf* g, Tip_Element e)
  int i,j;
                                          /*[10.3.1.2.b]*/
  /*incrementare contor noduri*/
  (*g).contor= (*g).contor+1;
  /*se plasează nodul nou*/
  (*g).noduri[(*g).contor]= e;
  /*se iniţializează matricea de adiacenţe pentru nodul
    nou*/
  pentru (i=1 la (*g).contor) /*iniţializare coloană nod*/
    (*g).arce[i,(*g).contor]= false;
  pentru (j=1 la (*g).contor) /*iniţializare linie nod*/
    (*g).arce[(*g).contor,j]= false;
/*InserNod*/
-------------------------------------------------------------
```

  - (b) Dacă în tabloul `noduri` informaţiile sunt **ordonate**, atunci:
    - În prealabil trebuie determinat locul în care se va realiza inserţia.
    - Se mută elementele tabloului `g.noduri` pentru a crea loc noului nod.
    - Se mută liniile şi coloanele matricei de adiacenţe `g.arce` pentru a crea loc liniei şi coloanei corespunzătoare noului nod.
    - În final se realizează inserţia propriu-zisă prin completarea tabloului noduri şi a matricei de adiacenţe.

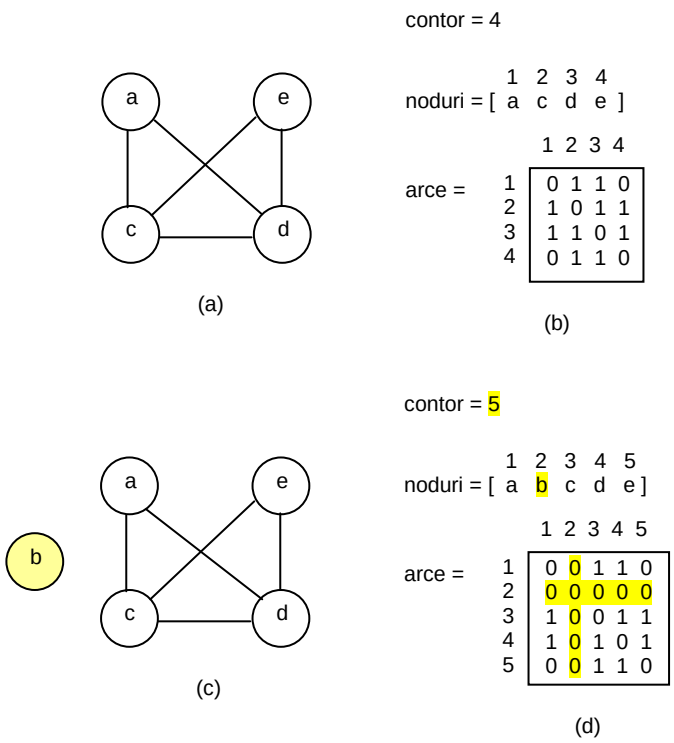


Fig.10.3.1.2.c. Inserţia unui nod într-un graf (varianta tablou `noduri` ordonat)

- (4) Într-o manieră similară, la **suprimarea** nodului cu indexul `indicNod`, trebuiesc efectuate mişcări în tablourile `g.noduri` şi `g.arce`.
  - Se presupune că se doreşte suprimarea nodului `c` din structura graf din figura 10.3.1.2.b.
  - (a) Dacă tabloul `noduri` este **neordonat**.
    - În vederea suprimării, `indicNod` are valoarea `2` precizând nodul `c`, iar suprimarea propriu-zisă presupune ştergerea nodului din `g.Noduri` şi modificarea matricei de adiacenţe prin excluderea arcelor conexe nodului `c`.
    - Suprimarea nodului `c` se poate realiza prin mutarea ultimului element al tabloului `noduri` în locul său.
    - Pentru păstrarea corectitudinii reprezentării, este necesară ştergerea arcelor conexe lui `c` din matricea de adiacenţe.
      - Pentru aceasta se copiază **linia** şi **coloana** corespunzătoare ultimului nod din matricea `g.arce`, adică nodul `b`, peste linia şi coloana nodului care a fost suprimat.
    - În final se decrementează variabila `g.contor`.
    - Rezultatul suprimării apare în figura 10.3.1.2.d.

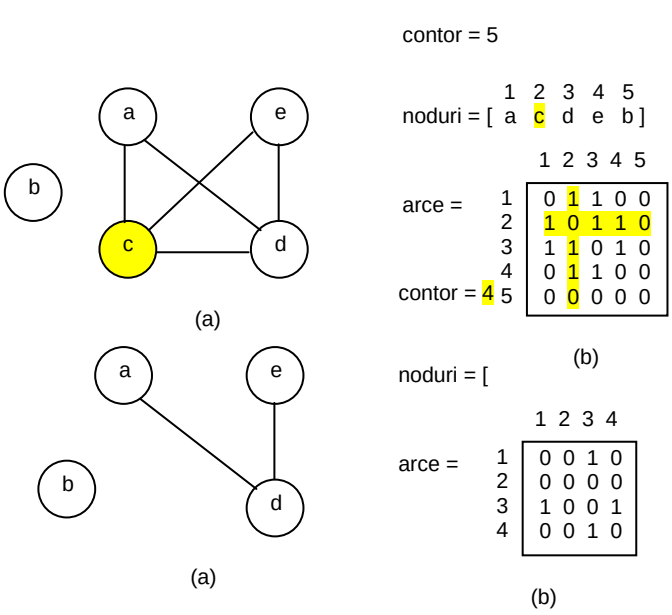


Fig.10.3.1.2.d. Suprimarea unui nod dintr-o structură graf (varianta tablou `noduri` neordonat)

- Procedura care implementează aceste activităţi se numeşte ***SuprimNod*** şi apare în secvenţa [10.3.1.2.c].

```
-------------------------------------------------------------
/*Suprimarea unui nod. (Tabloul noduri neordonat) – varianta
pseudocod*/

procedure SuprimNod(TipGraf * g, int indicNod)

  int i,j;

  /*actualizare tablou de noduri*/       /*[10.3.1.2.c]*/
  /*mutarea ultimului nod în tabloul de noduri*/
  (*g).noduri[indicNod]= (*g).noduri[(*g).contor];
  /*actualizare matrice de adiacenţe*/
  pentru (j=1 la (*g).contor) /*mutare linie în matricea
                                arce*/
    (*g).arce[indicNod,j]= (*g).arce[(*g).contor,j];
  pentru (i=1 la (*g).contor) /*mutare coloană în matricea
                               arce*/
    (*g).arce[i,indicNod]= (*g).arce[i,(*g).contor];
  /*decrementare contor noduri*/
  (*g).contor= (*g).contor-1;
/*SuprimNod*/
-------------------------------------------------------------
```

- (b) Dacă tabloul `noduri` este **ordonatat**, atunci **suprimarea** presupune :
  - Mutarea cu o poziţie a tuturor nodurilor care au indexul mai mare ca `indicNod` din tabloul `noduri`.
  - Mutarea liniilor aflate sub linia lui `c` cu o poziţie în sus.
  - Mutarea coloanelor situate la dreapta coloanei lui `c` cu o poziţie spre stânga (fig. 10.3.1.2.e).

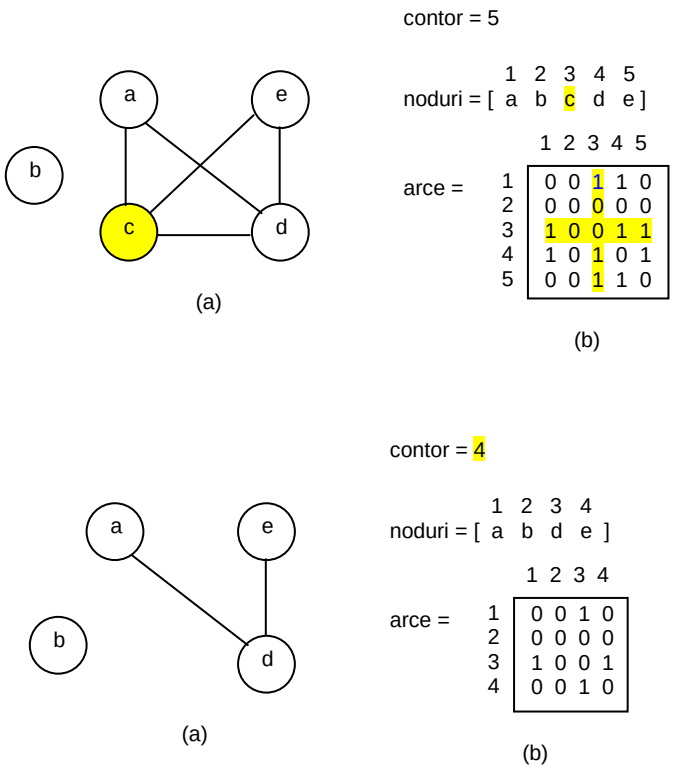


Fig.10.3.1.2.e. Suprimarea unui nod dintr-o structură graf (varianta tablou `noduri` ordonat)

- După cum se observă **suprimarea unui nod** presupune implicit şi **suprimarea arcelor** conexe lui.

- (5) Există însă posibilitatea de a **şterge arce** fără a modifica mulţimea nodurilor.
  - În acest scop se utilizează procedura ***SuprimArc***`(TipGraf* g, TipArc indicArc)` secvenţa [10.3.1.2.d].
    - Datorită simetriei reprezentării stergerea unui arc presupune două modificări în matricea de adiacenţe.

```
-------------------------------------------------------------
{Suprimarea unui arc – varianta pseudocod}

procedure SuprimArc(Tip_Graf * g, Tip_Arc indicArc)
                                              /*[10.3.1.2.d]*/
/*suprimă arcul (linie, coloană) din matricea de adiacenţe*/
  (*g).arce[indicArc.linie, indicArc.coloana]= false;
  (*g).arce[indicArc.coloana, indicArc.linie]= false;
/*SuprimArc*/

-------------------------------------------------------------
```

- În concluzie în studiul de caz 2, **crearea unei structuri de date pentru un TDA graf** presupune două etape:
  - (1) Precizarea nodurilor grafului, implementată printr-o suită de apeluri ale procedurii ***InserNod*** (câte un apel pentru fiecare nod al grafului).
  - (2) Conectarea nodurilor grafului, implementată printr-o suită de apeluri ale procedurii ***InserArc*** (câte un apel pentru fiecare arc al grafului).

- În general reprezentarea bazată pe **matrice de adiacenţe** este eficientă în cazul **grafurilor dense**.
    - Din punctul de vedere al spaţiului de memorie necesar reprezentării, matricea de adiacenţe necesită n^2 locaţii de memorie pentru un graf cu n noduri.
    - În plus mai sunt necesare locaţii de memorie pentru memorarea informaţiilor aferente celor n noduri.
    - Crearea grafului necesită un efort proporţional cu $O(n)$ pentru noduri şi aproximativ $O(n^2)$ paşi pentru arce, mai precis $O(a)$.
- În consecinţă, de regulă, utilizarea reprezentării bazate pe **matrice de adiacenţe** conduce la algoritmi care necesită un efort de calcul de ordinul $O(n^2)$.

10.3.2. Implementarea grafurilor cu ajutorul structurilor de adiacenţe

- O altă manieră de reprezentare a TDA graf o **constituie structurile de adiacenţe** ("**adjacency-structures**").
    - În cadrul acestei reprezentări, fiecărui nod al grafului i se asociază **o listă de adiacenţe** în care sunt înlănţuite toate nodurile cu care acesta este conectat.
- În continuare se prezintă două studii de caz pentru implementarea grafurilor cu ajutorul structurilor de adiacenţă.

10.3.2.1. Studiu de caz 1

- Implementarea structurii de adiacenţe se bazează în cazul 1 de studiu pe **liste înlănţuite simple**.
    - Începuturile listelor de adiacenţe sunt păstrate într-un tablou `Stradj` **indexat** prin intermediul nodurilor.
    - Iniţial în acest tablou se introduc înlănţuiri vide, urmând ca inserţiile în liste să fie de tipul "la începutul listei".
    - Adăugarea unui arc care conectează nodul x cu nodul y în cadrul acestui mod de reprezentare presupune în cazul grafurilor neorientate:
        - (1) Inserţia nodului x în lista de adiacenţe a lui y.
        - (2) Inserţia nodului y în lista de adiacenţe a nodului x.
- Un exemplu de program care construieşte o astfel de structură apare în secvenţa [10.3.2.1.a].

```
-----------------------------------------------------------------
/*Cazul 1. Construcţia unui graf utilizând structuri de
adiacenţe implementate cu ajutorul listelor înlănţuite simple
- varianta pseudocod C-like*/

const maxN = 100;

typedef struct Tip_Nod* Ref_Tip_Nod;

/*structura unui nod*/
typedef struct Tip_Nod
 {
    int nume;
```

```
   Ref_Tip_Nod urm;
} Tip_Nod;

/*structura de adiacenţe*/
Ref_Tip_Nod  StrAdj[maxN];

int j,x,y;
int N; /*N= numărul de noduri ale grafului*/
int A; /*A= numărul de arce ale grafului*/
Ref_Tip_Nod  u,v;

*citeste(N,A); /*N= număr noduri; A= număr arce*/
/*iniţializare structură de adiacenţe*/
pentru (j=0 la N-1)
  StrAdj[j]= null;
/*construcţie graf*/
pentru (j=1 la A)  /*adăugare arce*/
   |*citeste(n1,n2);                          /*[10.3.2.1.a]*/
   |x= index(n1); y= index(n2);
   |*aloca(u); /*aloca memorie pentru nodul u*/
   |u->nume= x; u->urm= StrAdj[y];
   |StrAdj[y]= u; /*inserţie în faţă a lui x în lista lui y*/
   |*aloca(v); /*aloca memorie pentru nodul v*/
   |v->nume= y; v->urm= StrAdj[x];
   |StrAdj[x]= v /*inserţie în faţă a lui y în lista lui x*/
   |□ /*pentru*/
/*construcţie graf utilizând structuri de adiacenţe*/
```

- În figura 10.3.2.1.a se prezintă reprezentarea grafică a structurii construite pornind de la graful (a) din aceeaşi figură.
- Se face precizarea că datele de intrare (arcele) au fost furnizate în următoarea ordine: `(a,c), (a,d), (c,e), (c,d)` şi `(d,e)`.

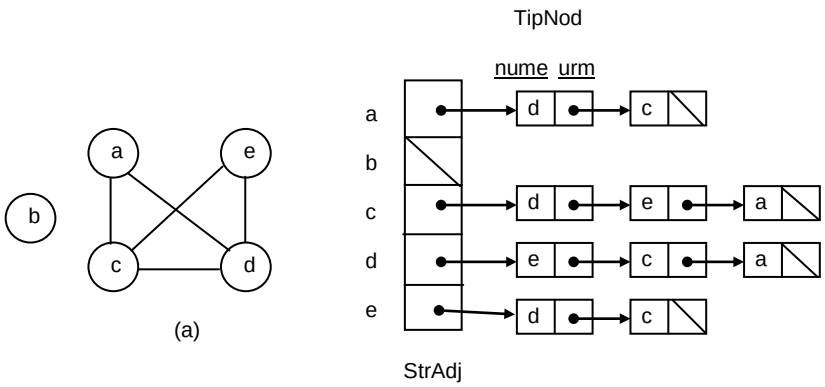


Fig.10.3.2.1.a. Graf şi structura sa de adiacenţe

- Se observă că un arc oarecare `(x,y)` este evidenţiat în două locuri în cadrul structurii, atât în lista de adiacenţe a lui `x`, cât şi în cea a lui `y`.
    - Acest mod redundant de evidenţiere îşi dovedeşte utilitatea în situaţia în care se cere să se determine într-o manieră eficientă care sunt nodurile conectate la un anumit nod x.
- Pentru acest mod de reprezentare, contează ordinea în care sunt prezentate arcele respectiv perechile de noduri, la intrare.

- Astfel, un acelaşi graf poate fi reprezentat ca structură de adiacenţe în moduri diferite.

- Ordinea în care apar arcele în lista de adiacenţe, afectează la rândul ei ordinea în care sunt prelucrate arcele de către algoritm.
  - Funcţie de natura algoritmilor utilizaţi în prelucrare această ordine poate să influenţeze sau nu rezultatul prelucrării.

10.3.2.2. Studiu de caz 2

- În acest cazul 2 de studiu, implementarea structurilor de adiacenţe se bazează pe **structuri multilistă**.
  - Astfel, **o structură de adiacenţe** este de fapt o **listă înlănţuită** a nodurilor grafului.
  - Pentru fiecare nod al acestei liste se păstrează o **listă a arcelor**, adică o listă înlănţuită a cheilor nodurilor adiacente.
  - Cu alte cuvinte, o **structură de adiacenţe** în acest context este o **listă de liste**.
  - În consecinţă, fiecare nod al listei nodurilor va conţine două înlănţuiri, una indicând nodul următor, cealaltă, lista nodurilor adiacente.

- În figura 10.3.2.2.a apare structura de adiacenţe (b) a grafului (a).

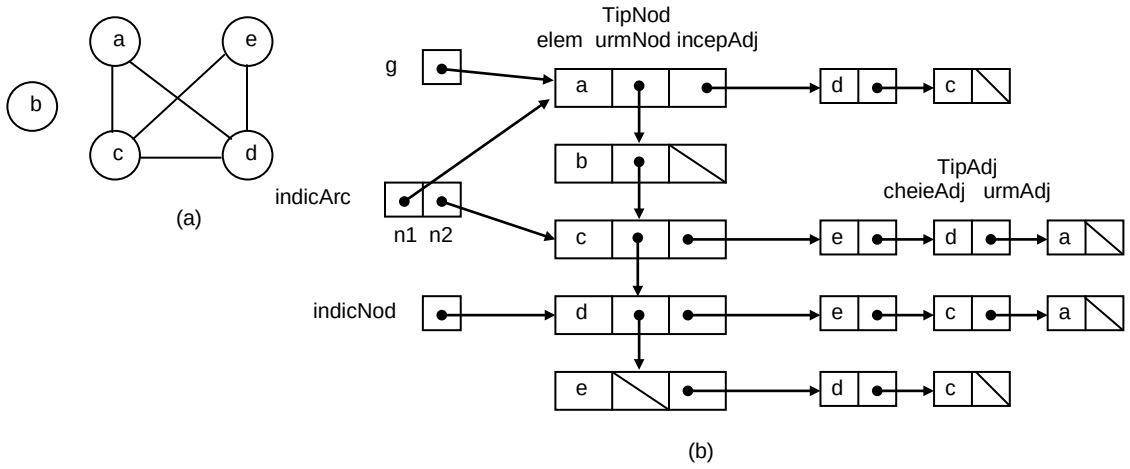


Fig.10.3.2.2.a. Reprezentarea unui graf ca şi o structură de adiacenţe utilizînd structuri multilistă

- Implementarea C a structurii multilistă apare în secvenţa [10.3.2.2.a].

```
-------------------------------------------------------------
/*Cazul 2. Implementarea grafurilor utilizând structuri de
adiacenţe implementate cu ajutorul structurilor multilistă -
varianta C*/

typedef ...  Tip_Cheie;
typedef ...  Tip_Info;

/*definire tip structura element*/
typedef struct Element
   {
```

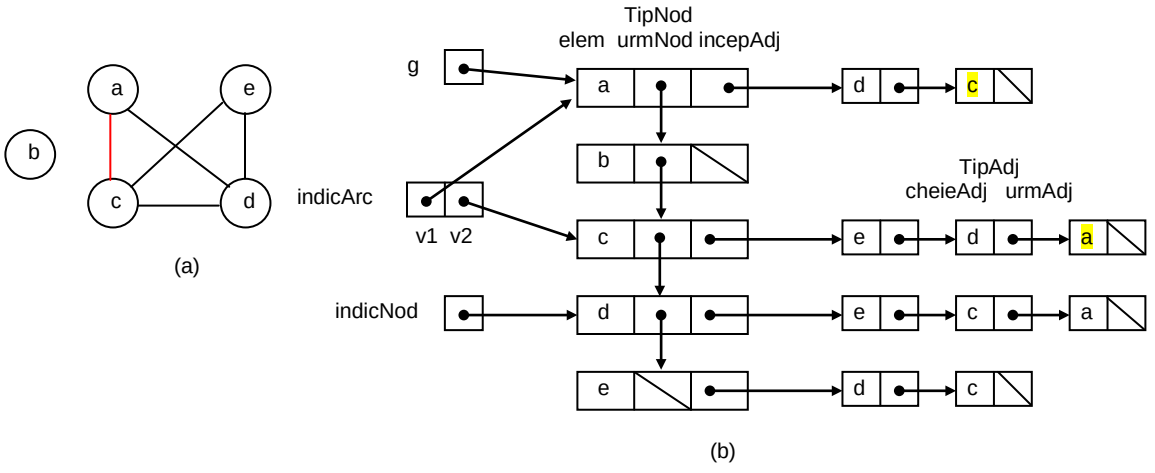
```
      Tip_Cheie cheie;
      Tip_Info  info;
   } Tip_Element

/*definire referinţă la structura nod al listei de
adiacenţe*/
typedef struct* Adj Ref_Tip_Adj;

/*definire tip structura nod al listei de adiacenţe*/
typedef struct Adj
   {
      Tip_Cheie cheie_Adj;
      Ref_Tip_Adj urm_Adj;
   } Tip_Adj;

/*definire referinţă la structura nod al listei nodurilor*/
typedef struct * Nod Ref_Tip_Nod;

/*definire tip structura graf*/
typedef Ref_Tip_Nod   Tip_Graf;

/*definire tip structura nod al listei nodurilor*/
typedef struct Nod
   {
      Tip_Element elem;
      Ref_Tip_Nod urm_Nod;
      Ref_Tip_Adj incep_Adj;       /*[10.3.2.2.a]*/
   } Tip_Nod

/*definire tip structura arc*/
typedef struct Arc
   {
      Ref_Tip_Nod n1,n2;
   } Tip_Arc

TipGraf  g;
Ref_Tip_Nod indic_Nod;
Tip_Arc     indic_Arc;
Tip_Cheie   k,k1,k2;
Tip_Element e;
-------------------------------------------------------------
```

- Se face precizarea că valorile aferente nodurilor sunt păstrate integral în lista de noduri, în lista de arce apărând numai cheile.
    - Este posibil ca câmpul `info` să lipsească şi deci `Tip_Element= Tip_Cheie`.
- În figura 10.3.2.2.a apare reprezentarea unei structuri de adiacenţe implementate cu multiliste cu precizarea câmpurilor aferente.
    - De asemenea sunt prezentate ca exemplu variabilele `g`, `indicNod` şi `indicArc`, evidenţiindu-se structura fiecăreia.
- În cadrul acestei structuri de date:
    - Operatorii ***`CautăCheieGraf`***, ***`IndicăNod`*** şi ***`IndicăArc`*** aparţinând setului de operatori extins (varianta 1) utilizează tehnica **căutării liniare** în liste înlănţuite pentru determinarea unui nod a cărui cheie este cunoscută.

- **Inserţia unui nod** nou se realizează simplu la începutul listei nodurilor.
- Operatorul ***InserArc***`(Tip_Graf * g, Tip_Cheie` k_1`, Tip_Cheie` k_2`)` presupune inserţia lui k_1 în lista de adiacenţe a lui k_2 şi reciproc.
  - Şi în acest caz inserţia se realizează cel mai simplu la începutul listei.
- **Suprimarea unui arc** precizat spre exemplu de indicatorul `indicArc` presupune extragerea a două noduri din două liste de adiacenţe diferite.
  - Astfel în figura 10.3.2.2.a, variabila `indicArc` conţine doi pointeri n_1 şi n_2, care indică cele două noduri conectate din lista de noduri.
  - În vederea suprimării arcului care le conectează este necesar ca fiecare nod în parte să fie suprimat din lista de adiacenţe a celuilalt.
  - În cazul ilustrat, pentru a suprima arcul `(a,c)=(c,a)` se scoate `a` din lista lui `c`, respectiv `c` din lista lui `a`.

- Procedura care realizează **suprimarea unui arc** în această manieră a apare în secvenţa [10.3.2.2.b].
  - Se face precizarea că procedura ***SuprimArc*** este redactată în termenii setului de operatori aplicabili obiectelor aparţinând lui **TDA Listă** [Vol 1. &6.2.1].

```
-------------------------------------------------------------
/*Suprimarea unui arc. În implementare se utilizează
operatorii definiţi pentru TDA Lista înlănţuită simplă -
varianta C-like*/

procedure SuprimArc(Tip_Graf g, Tip_Arc indic_Arc)

  Ref_Tip_Adj ik1,ik2;                    /*10.3.2.2.b*/

  /*caută cheia nodului n1 în lista de adiacenţe a lui n2*/
  ik1= Cauta(indic_Arc.n1->elem.cheie,
                  indic_Arc.n2->incep_Adj);
  /*caută cheia nodului n2 în lista de adiacenţe a lui n1*/
  ik2= Cauta(indic_Arc.n2->elem.cheie,
                  indic_Arc.n1->incep_Adj);
  /*suprimă nodul n1 din lista de adiacenţe a lui n2*/
  Suprima(ik1,indic_Arc.n2->incep_Adj);
  /*suprimă nodul n2 din lista de adiacenţe a lui n1*/
  Suprima(ik2,indic_Arc.n1->incep_Adj);
/*SuprimArc*/
-------------------------------------------------------------
```



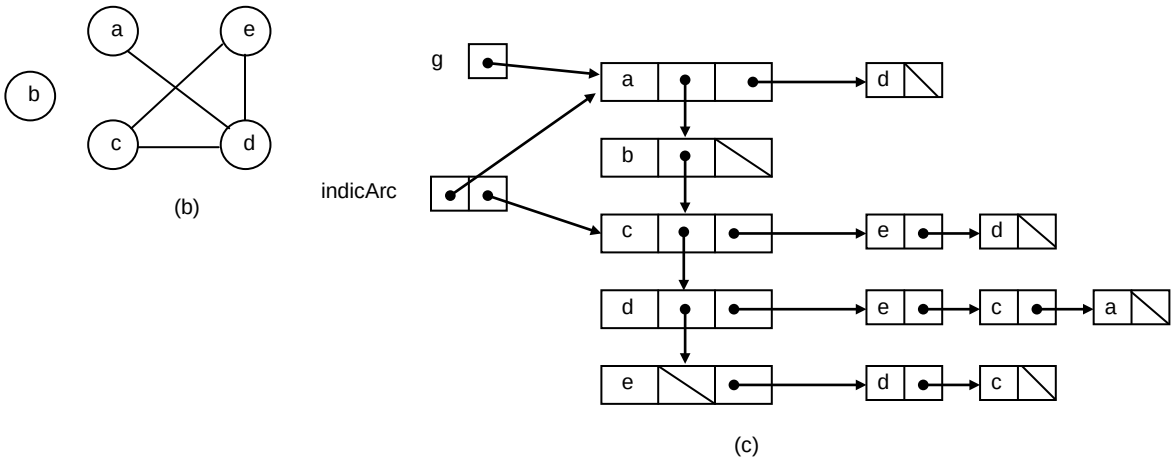


Fig.10.3.2.2.b. Structură de adiacenţe (c) după suprimarea arcului `(a,c)`. Graful original (a), graful după suprimare (b).

- În figura 10.3.2.2.b apare structura de adiacenţe (c) aferentă grafului (b) după suprimarea arcului `(a,c)`din graful original (a)`.`
- **Suprimarea unui nod** dintr-o structură graf, presupune nu numai suprimarea propriu-zisă a nodului respectiv ci în plus suprimarea tuturor arcelor incidente acestui nod.
  - În acest scop, se determină cheia k1 a nodului de suprimat.
  - În continuare, atât timp cât mai există elemente în lista de adiacente a lui `k1` se realizează următoarea secvenţă de operaţii:
    - (1) Se determină cheia `k2` a primului element din lista de adiacenţe a lui `k1`.
    - (2) Se suprimă arcul `(k1,k2)` suprimând pe `k2` din lista lui `k1` şi pe `k1` din lista lui `k2`.
    - (3) Se reia procesul de la (1).
  - În final se suprimă nodul `k1` din lista nodurilor grafului.
- Procedura care realizează suprimarea unui nod apare în secvenţa [10.3.2.2.c]
  - Se face de asemenea precizarea că procedura ***SuprimNod*** este redactată în termenii setului de operatori aplicabili obiectelor aparţinând lui **TDA Listă** [Vol 1. &6.2.1].

```
-------------------------------------------------------------
/*Suprimarea unui nod. În implementare se utilizează
operatorii definiţi pentru TDA Listă înlanţuită simplă -
varianta pseudocod*/

procedure SuprimNod(Tip_Graf g, Ref_Tip_Nod indic_Nod)

  Tip_Cheie k1,k2;
  Ref_Tip_Nod  indic_Nod2;
  Ref_Tip_Adj  ik1,curent;                /*[10.3.2.2.c]*/

  k1= indic_Nod->elem.cheie; /*k1 este cheia nodului de
                                          suprimat*/
  curent= indic_Nod->incep_Adj; /*curent indică lista de
```

```
                                        adiacenţe a lui k1*/
  cat timp (not Fin(curent))
     k2= (Primul(curent))->cheieAdj; /*k2 indică primul
                                        element al listei k1*/
     /*suprimă pe k2 din lista lui k1*/
     Suprima(Primul(curent),curent);
     /*caută nodul k2 în graf*/
     indicNod2= Cauta(k2,g);
     /*caută pe k1 în lista lui k2*/
     ik1= Cauta(k1,indicNod2->incepAdj);
     /*suprimă pe k1 din lista lui k2*/
     Suprima(ik1,indicNod2->incepAdj);
     □ /*cat timp*/
  Suprima(indicNod,g) /*suprimă nodul k1 din graf*/
/*SuprimNod*/
-----------------------------------------------------------------
```

- În fig.10.3.2.2.c apare reprezentată structura de adiacenţe (c) a grafului (b) rezultată în urma suprimării nodului d din graful original (a).

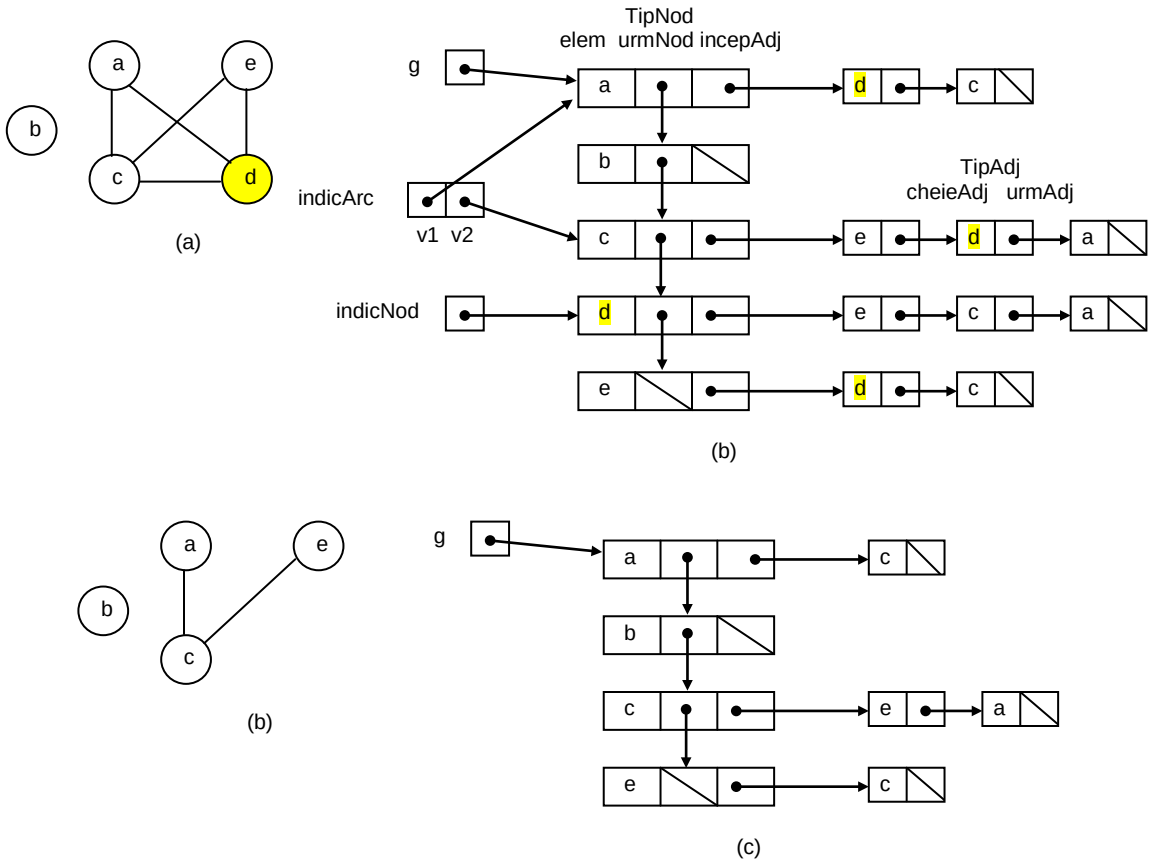


Fig.10.3.2.2.c. Structura de adiacenţe (c) a grafului (b) rezultată în urma suprimării nodului d din graful original (a).

10.3.2.3. Studiu de caz 3

- Cel de al treilea studiu de caz prezintă o altă variantă de implementare a structurilor de adiacenţă bazată pe **structuri multilistă.**

- Exemplul se referă la varianta Deker de reprezentare a **TDA Graf** [De89].

- Modificarea faţă de varianta anterioară constă în faptul că în **listele de adiacenţe** ale nodurilor, cheile nodurilor adiacente sunt înlocuite cu **pointeri** la nodurile corespunzătoare din **lista nodurilor**.

- Structura de adiacenţe, conformă acestei reprezentări a grafului (a) din fig.10.3.2.3.a apare în aceeaşi figura (b).

- Deşi modificarea efectuată conduce la o reprezentare grafică mai complicată, implementarea operatorilor definiţi asupra unui **TDA graf** devine mai simplă.

  - Programatorul **nu** trebuie să fie timorat de mulţimea înlănţuirilor prezente în structura multilistă, deoarece menţinerea evidenţei acestora revine programului.

  - De fapt, la nivel de implementare lucrurile se simplifică.

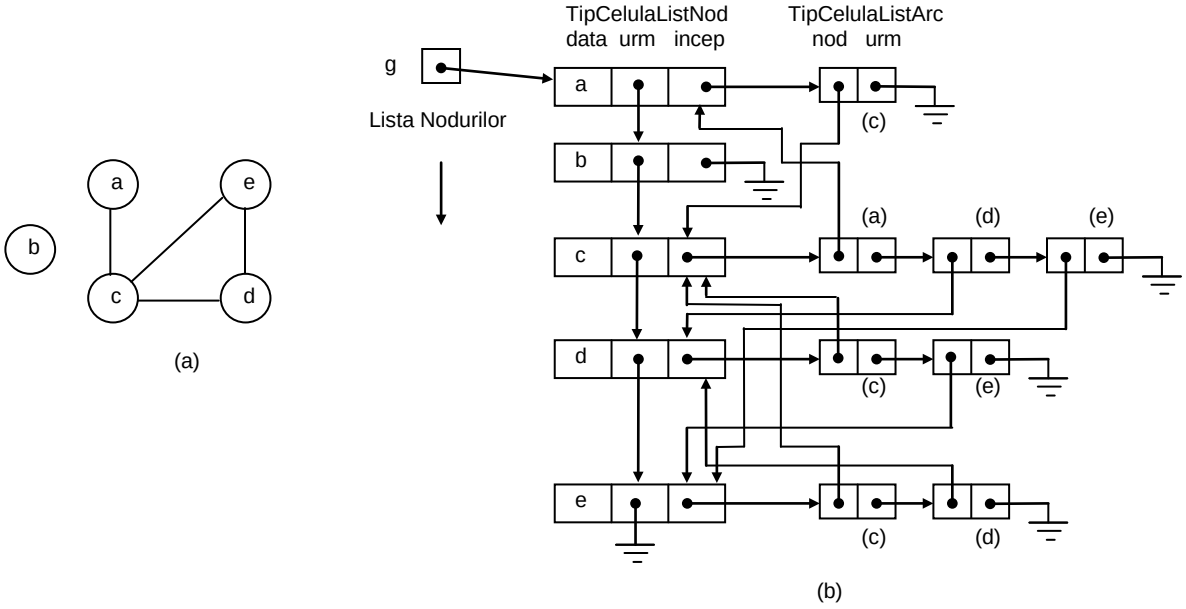


Fig.10.3.2.3.a. Structură de adiacenţe pentru un graf

- În continuare se prezintă în accepţiunea acestei reprezentări, implementarea setului de operatori restrâns definit asupra **TDA graf** (Varianta 2 (Decker)).

  - **Structurile de date** utilizate în reprezentare apar în secvenţa [10.3.2.3.a].

  - **Codul** aferent operatorilor apare în secvenţa [10.3.2.3.b].

  - Se face precizarea că în concordanţă cu maniera de definire a tipurilor de date, din lista de parametri a fost omis parametrul `g:TipGraf`, prezenţa sa nefiind considerată necesară.

```
-------------------------------------------------------------
{Cazul 3. Implementarea grafurilor utilizând structuri de
adiacenţe implementate cu ajutorul structurilor multilistă.
Se utilizează TDA Graf (Varianta 2 (Decker)) - implementare
PASCAL}

type
  RefTipPozitie = ^TipCelulaListNod;
  RefListArc = ^TipCelulaListArc;
                                                  [10.3.2.3.a]
```

```
  TipCelulaListNod = record  {celulă în lista nodurilor}
                       data : TipAtom; {informaţia
                                        aparţinând nodului}
                       urm  : RefTipPozitie; {următoarea
                                   celulă în lista nodurilor}
                       incep: RefListArc {referinţă la
                                   lista de noduri adiacente}
                     end;

  TipCelulaListArc = record {celulă în lista de adiacenţe}
                       data: ...; {informaţia ataşată
                                                arcului}
                       nod : RefTipPozitie; {nodul
                                             destinaţie}
                       urm : RefListArc {înlănţuirea în
                                        lista de adiacenţe}
                     end;

  TipArc = record  {structură arc}
             nod1,nod2: RefTipPozitie
           end;

  TipGraf = RefTipPozitie; {structură graf}
-------------------------------------------------------------

procedure Creaza(var g: TipGraf);
  begin
    g:= nil
  end; {Creaza}

function Adiacent(p,q: RefTipPozitie): boolean;
{Primeşte pointerii p şi q la lista de noduri a lui g. Caută
  în lista de arce a lui p, celula care indica nodul q}

 var gata: boolean; {specifică terminarea traversării listei
                     de adiacenţe}
    curent: RefListArc; {pointer la celula curentă a listei
                         de adiacenţe }
 begin {Adiacent}
    Adiacent:= false;
    curent:= p^.incep;{începutul listei de adiacenţe a lui
                       p}
    gata:= false;
    while not gata do
      if curent=nil then
        gata:= true
      else
        if curent^.nod = q then                [10.3.2.3.b]
          begin
            gata:= true;
            Adiacent:= true
          end
        else
          curent:= curent^.urm
  end; {Adiacent}

procedure Modifica(a: TipAtom; var p: RefTipPozitie);
  begin
```

```
    p^.data := a
  end; {Modifica}

function Furnizeaza(p: RefTipPozitie ): TipAtom;
  begin
    Furnizeaza := p^.data
  end; {Furnizeaza}

procedure SuprimNod(p: RefTipPozitie; var g: TipGraf);
  var NodCurent: RefTipPozitie;

  procedure SuprimaCel(n: RefTipPozitie; var început:
                       RefListArc);
     {procedură recursivă care suprimă din lista indicată de
      "început", celula având câmpul nod egal cu n}
    var temp: RefListArc;

    begin
      if inceput<>nil then
        if inceput^.nod = n then
          begin
            temp:= inceput;
            inceput:= inceput^.urm;
            DISPOSE(temp)
          end
        else
          SuprimaCel(n,inceput^.urm)
    end; {SuprimaCel}

  procedure StergeList(var inceput: RefListArc);
    {suprimă recursiv toate elementele listei indicate de
     început}
    begin
      if inceput<>nil then
        begin
          StergeList(inceput^.urm);
          DISPOSE(inceput)
        end
    end; {StergeList}

  begin {SuprimNod}
    NodCurent:= g;
    WHILE NodCurent<>nil do {parcurge lista de noduri a
                             grafului}
      begin
        if p<>NodCurent then
          SuprimaCel(p,NodCurent^.incep); {pentru fiecare
            nod al grafului, parcurge lista sa de adiacenţe
            şi suprimă nodul p}
        NodCurent:= NodCurent^.urm              [10.3.2.3.b]
      end
    StergeList(p^.incep); {şterge lista de adiacenţe a
                  nodului indicat de p}
    SuprimaCel(g,p) {suprimă nodul p}
end; {SuprimNod}

procedure InserNod(var p: RefTipPozitie; var g: TipGraf);
  begin
```

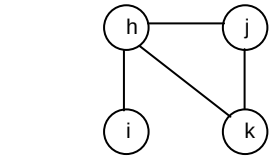
```
    New(p);
    p^.urm:= g;    {inserţie în faţă}
    p^.incep:= nil;
    g:= p
  end; {InserNod}

procedure InserArc(e: TipArc; var g: TipGraf);
  var temp: RefListArc;
  begin
    if not Adiacent(e.nod1,e.nod2) then
      begin
        temp:= e.nod1^.incep; {inserţie nod2 în lista nod1}
        New(e.nod1^.incep);
        e.nod1^.incep^.nod:= e.nod2;
        e.nod1^.incep^.urm:= temp;
        temp:= e.nod2^.incep; {inserţie nod1 în lista nod2}
        New(e.nod2^.incep);
        e.nod2^.incep^.nod:= e.nod1;
        e.nod2^.incep^.urm:= temp
      end
  end; {InserArc}

procedure SuprimArc(e: TipArc; var g: TipGraf)

  procedure SuprimaCel(n: RefTipPozitie; var inceput:
                       RefListArc);
   {procedură recursivă care suprimă din lista indicată de
    "început", celula având câmpul nod egal cu n}
            .
            . {ca mai sus}
            .
    end; {SuprimaCel}                          {[10.3.2.3.b]}

  begin {SuprimArc}
    if Adiacent(e.nod1,e.nod2) then
      begin
        SuprimaCel(e.nod2,e.nod1^.incep); {suprimare nod2
                                           din lista nod1}
        SuprimaCel(e.nod1,e.nod2^.incep)  {suprimare nod1
                                           din lista nod2}
      end
  end; {SuprimArc}
-------------------------------------------------------------
```

10.3.3. Considerente referitoare la stabilirea modului de reprezentare a unui TDA graf.

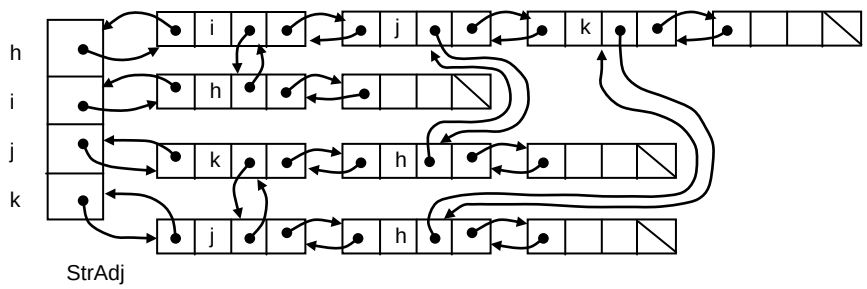
- Din punct de vedere al eficienţei, alegerea unei anumite reprezentări pentru o structură de date depinde esenţial de **natura operaţiilor** care se execută asupra ei.
    - Acest lucru este valabil şi în cazul grafurilor.
- (1) Un prim aspect care poate fi luat în considerare în **alegerea reprezentării**, este cel legat de **dinamica grafului** avut în vedere.

- Dacă graful este puternic şi imprevizibil **variabil în timp**, la baza reprezentării trebuie să stea o **structură dinamică**, altfel poate fi utilizată o **structură statică.**

- (2) Un al doilea aspect se referă la **frecvenţa** cu care se execută diferitele **operaţii** asupra grafului de reprezentat.
  - De regulă, astfel de operaţii includ:
    - (a) **Identificarea** (căutarea) unui nod sau unui arc.
    - (b) **Inserţia** unui nod sau unui arc.
    - (c) **Suprimarea** unui nod sau arc.
    - (d) **Traversarea** unui graf.
  - (a) Referitor la **identificarea** sau **căutarea** unui nod sau arc al unui graf:
    - În cazul reprezentării bazate pe **matrice de adiacenţe timpul de acces** la un element al matricei (grafului) **nu** depinde de dimensiunea acesteia.
    - În cazul reprezentării prin **structuri de adiacenţe**, **timpul de acces** depinde de poziţia în listă a elementului căutat, respectiv de ordinea de creare a structurii.
    - În consecinţă, reprezentarea bazată pe **matrici de adiacenţe** este mai utilă în cazul algoritmilor în care se verifică frecvent prezenţa arcelor sau nodurilor într-un graf.
  - (b) **Inserţia** nodurilor se realizează simplu în ambele reprezentări.
  - (c) În ceea ce priveşte **suprimarea** nodurilor (arcelor) ea se realizează simplu în cazul reprezentării bazate pe **matrice de adiacenţe** şi mai complicat în cazul **structurii de adiacenţe**.
    - În ultimul caz, pentru a şterge un nod x şi toate arcele conexe, nu este suficient a se anula lista sa de adiacenţe, ci în plus, nodul x trebuie căutat şi suprimat din listele de adiacenţe ale tuturor nodurilor care apar în propria sa listă.
    - Această situaţie poate fi rezolvată într-o manieră eficientă, **înlănţuind între ele nodurile corespunzătoare unui anumit arc** şi reprezentând structura de adiacenţe prin **liste dublu înlănţuite** (fig.10.3.3).



(a)



(b)

Fig. 10.3.3. Exemplu de structură de adiacenţe complexă

- În aceste condiţii la suprimarea unui arc, nodurile corespunzătoare pot fi uşor înlăturate din listele de adiacenţe parcurgând înlănţuirile.
- Este însă evident faptul că înlănţuirile suplimentar introduse, deşi simplifică suprimarea, complică atât structura în ansamblul ei cât şi modul ei de prelucrare (spre exemplu inserţia).
- Din acest motiv, ele nu trebuiesc introduse decât în situaţia în care sunt absolut necesare.

- (d) Traversarea grafurilor va fi abordată într-un paragraf ulterior. Ea depinde în manieră esenţială de modul de reprezentare al structurii.

- (3) **Grafurile orientate** şi cele **ponderate** pot fi reprezentate prin structuri similare.
  - (a) Spre exemplu, pentru **grafurile orientate** se pot utiliza aceleaşi reprezentări pentru structura graf, exceptând faptul că un **arc** este reprezentat doar **o singură dată** în structură.
    - Astfel arcul de la nodul `x` la nodul `y` este reprezentat prin valoarea adevărat în câmpul `A[x,y]` al **matricei de adiacenţe**, sau prin apariţia lui `y` în lista de adiacenţe a lui `x` în reprezentarea bazată pe **structuri de adiacenţe**.
    - De fapt un **graf neorientat** poate fi conceput ca şi un **graf orientat** cu arce direcţionate în ambele sensuri, între toate perechile de noduri conectate.
  - (b) În cazul **grafurilor ponderate**, se pot de asemenea utiliza aceleaşi reprezentări cu diferenţa că:
    - În **matricea de adiacenţe** se memorează **ponderile arcelor** în locul valorilor booleene.
      - Se utilizează o **pondere inexistentă convenită** pentru a reprezenta valoarea (“false”) adica absenţa conexiunii.
    - În nodurile **listelor de adiacenţe** se includ **câmpuri suplimentare** pentru memorarea ponderii arcelor.
- (4) În unele cazuri, în vederea modelării unor situaţii mai complicate, sau pentru a reduce regia unor algoritmi complecşi, nodurilor şi arcelor unui graf li se pot asocia şi alte **informaţii**.
  - Aceste informaţii pot fi păstrate în cadrul structurii propriu-zise care materializează graful sau de la caz la caz în **structuri conexe**.
- (5) În ceea ce priveşte **spaţiul de memorie** necesar se pot face următoarele consideraţiuni.
  - Presupunând că numărul nodurilor grafului de reprezentat este `n` şi numărul arcelor `a`.
    - În implementarea bazată pe tablouri, **matricea de adiacenţe** împreună cu **tabloul nodurilor** necesită un spaţiu de memorie proporţional cu $n+n^2$.
      - Pentru un graf cu multe noduri şi arce relativ puţine, risipa de spaţiu este mare.

- În reprezentarea bazată pe **structuri înlănţuite** fiecare nod apare odată în lista nodurilor şi fiecare arc `(x,y)` consumă două noduri din listele de adiacenţe, unul pentru `x` şi altul pentru `y`.
    - Astfel în reprezentarea bazată pe **structuri de adiacenţe**, spaţiul de memorie necesar este proporţional cu `n+2a`.
    - Mai trebuie luat în considerare şi faptul că înlănţuirile ca atare consumă ele însele spaţiu de memorie.
- Elementul hotărâtor în departajarea performanţelor celor două metode de reprezentare, din punctul de vedere al **spaţiului de memorie** necesar îl reprezintă **numărul de arce** `a`.
    - După cum s-a precizat în §10.1, **numărul de arce** `a` poate lua valoari în domeniul `[0,(1/2)n(n-1)]`.
    - În consecinţă din valoarea maximă a numărului de arce `a=(1/2)n(n-1)` se deduce relaţia $2a=n^2-n<n^2$.
    - Rezultă că $2a<n^2$, deci reprezentarea înlănţuită în principiu este mai eficientă ca cea matricială, din punctul de vedere considerat.
- După cum s-a mai precizat însă la o analiză mai aprofundată trebuie luat în considerare şi **spaţiul de memorie auxiliar** necesar depozitării informaţiilor aferente nodurilor şi mai ales înlănţuirilor propriu-zise.

10.4. Tehnici fundamentale de traversare a grafurilor

- Rezolvarea eficientă a problemelor curente referitoare la grafuri, presupune de regulă, **traversarea** (vizitarea sau parcurgerea) nodurilor şi arcelor acestora într-o manieră sistematică.
- În acest scop s-au dezvoltat două **tehnici fundamentale**, una bazată pe **căutarea în adâncime**, cealaltă bazată pe **căutarea prin cuprindere**.

10.4.1. Traversarea grafurilor prin tehnica căutării ”în adâncime” (”Depth-First Search”)

- **Căutarea în adâncime** este una dintre tehnicile fundamentale de traversare a grafurilor.
    - **Căutarea în adâncime** constituie nucleul în jurul căruia pot fi dezvoltaţi numeroşi algoritmi eficienţi de prelucrare a grafurilor.
    - Spre exemplu, **traversarea în preordine a arborilor** îşi are originea în această tehnică.
- **Principiul căutării** “**în adâncime**” într-un graf `G` este următorul:
    - (1) Se marchează iniţial toate nodurile grafului `G` cu marca “`nevizitat`”.
    - (2) Căutarea debutează cu selecţia unui nod `n` a lui `G` pe post de nod de pornire şi cu marcarea acestuia cu “`vizitat`”.
    - (3) În continuare, fiecare nod nevizitat adiacent lui `n` este “căutat” la rândul său, aplicând în mod recursiv acelaşi algoritm de “căutare în adâncime”.
    - (4) Odată ce toate nodurile la care se poate ajunge pornind de la `n` au fost vizitate în maniera mai sus precizată, cercetarea lui `n` este terminată.

- (5) Dacă în graf au rămas noduri nevizitate, se selectează unul dintre ele drept nod nou de pornire şi procesul se repetă până când toate nodurile grafului au fost vizitate.

- Această tehnică se numeşte căutare "**în adâncime**" ("**depth-first**") deoarece parcurgerea grafului se realizează înaintând "în adâncime" pe o direcţie aleasă atâta timp cât acest lucru este posibil.
  - Spre **exemplu**, presupunând că `x` este **ultimul nod vizitat**, căutarea în adâncime selectează un arc neexplorat conectat la nodul x.
  - Fie `y` nodul corespunzător acestui arc.
    - Dacă nodul `y` a fost deja vizitat, se caută un alt arc neexplorat conectat la `x`.
    - Dacă `y` nu a fost vizitat anterior, el este marcat "`vizitat`" şi se iniţiază o nouă căutare începând cu nodul `y`.
  - În momentul în care se epuizează căutarea pe toate drumurile posibile pornind de la `y`, se revine la nodul `x`, (principiul recursivităţii) şi se continuă în aceeaşi manieră selecţia arcelor neexplorate ale acestui nod, până când sunt epuizate toate posibilităţile care derivă din `x`.
  - Se observă clar **tendinţa iniţială de adâncire**, de îndepărtare faţă de sursă, urmată de o revenire pe măsura epuizării tuturor posibilităţilor de traversare.

- Se consideră:
  - O **structură graf** într-o reprezentare oarecare.
  - Un tablou `marc` ale cărui elemente corespund nodurilor grafului.
    - În tabloul `marc` se memorează faptul că un nod al grafului a fost sau nu vizitat.

- Schiţa de principiu a **algoritmului de căutare în adâncime** apare în secvenţa [10.4.1.a].

```
-------------------------------------------------------------
/*Cautarea "în adancime". Schiţa de principiu a
 algoritmului. Varianta pseudocod*/

subprogram CautaInAdincime(Tip_Nod x)
  Tip_Nod y;

  marc[x]= vizitat;                          /*[10.4.1.a]*/
  pentru (fiecare nod y adiacent lui x)
    daca (marc[y] este nevizitat)
      CautaInAdincime(y);
/*CautaInAdincime*/
-------------------------------------------------------------
```

10.4.1.1. Căutarea "în adâncime", varianta CLR

- O variantă mai elaborată a **traversării în adâncime** este cea propusă de Cormen, Leiserson şi Rivest [CLR92].
  - Conform acesteia, pe parcursul traversării, nodurile sunt **colorate** pentru a marca stările prin care trec.

- Din aceste motive **traversarea grafurilor** este cunoscută şi sub denumirea de "**colorare a grafurilor**".
    - Culorile utilizate sunt alb, gri şi negru.
- Toate nodurile sunt iniţial colorate în **alb**, în timpul traversării sunt colorate în **gri** iar la terminarea traversării sunt colorate în **negru**.
    - Un nod **alb** care este descoperit prima dată în traversare este colorat în **gri**.
    - Un nod este colorat în **negru** când toate nodurile adiacente lui au fost descoperite.
        - În consecinţă, nodurile gri şi negre au fost deja întâlnite în procesul de traversare, ele marcând modul în care avansează traversarea.
    - Un nod colorat în **gri** poate avea şi noduri adiacente albe.
        - Nodurile **gri** marchează frontiera între nodurile descoperite şi cele nedescoperite şi ele se păstrează de regulă în structura de date (stiva) asociată procesului de traversare.
- În procesul de traversare se poate construi un **subgraf asociat traversării**, subgraf care include arcele parcurse în traversare şi care este de fapt un **graf de precedenţe**.
    - Acest subgraf este un **graf conex aciclic**, adică un **arbore**, care poate fi simplu reprezentat prin tehnica **"indicator spre părinte".**
- **Tehnica de construcţie a subgrafului** este următoarea:
    - Atunci când în procesul de căutare se ajunge de la nodul `u` la nodul `v`, nodul `v` nefiind vizitat încă, acest lucru se marchează în subgraful asociat `s` prin `s[v]=u`, adică `u` este părintele lui `v`.
    - **Graful de precedenţe** este descris formal conform relaţiilor [10.4.1.1.a].

$$G_{pred} = (N, A_{pred})$$
$$A_{pred} = \{(s[v], v): v \in N \ \&\ s[v] \neq nil\} \qquad [10.4.1.1.a]$$

        - **Subgrafurile predecesorilor** asociate **căutării în adâncime** într-un **graf** precizat, formează o **pădure** de **arbori de căutare în adâncime**.
    - Pe lângă crearea propriu-zisă a subgrafului predecesorilor fiecărui nod i se pot asocia două **mărci de timp** ("**timestamps**"):
        - (1) `i[v]` memorează **momentul descoperirii** nodului `v` (colorarea sa în **gri**).
        - (2) `f[v]` memorează **momentul terminării** explorării nodurilor adiacente lui `v` (colorarea sa în **negru**).
    - **Mărcile de timp** sunt utilizate în mulţi algoritmi referitori la grafuri şi ele precizează în general comportamentul lor în timp.
        - În cazul de faţă, pentru simplitate, **timpul** este conceput ca un întreg care ia valori între `1` şi `2|N|`, întrucât este incrementat cu `1` la fiecare **descoperire** respectiv **terminare** de examinare a fiecăruia din cele `|N|` noduri ale grafului.
- Varianta de **căutare în adâncime** propusă de Cormen, Lesiserson şi Rivest apare în secvenţa [10.4.1.1.b].

```
/*Cautarea "în adancime". Schiţa de principiu a
algoritmului. Varianta 2 (Cormen, Leiserson, Rivest} –
format pseudocod*/

procedure TraversareInAdincime(Tip_Graf G)
[1]  pentru (fiecare nod u∈N(G))
[2]      culoare[u]=alb;
[3]      sp[u]=null;
         □ /*pentru*/
[4]  timp=0;
[5]  pentru (fiecare nod u∈N(G))
[6]    daca (culoare[u] este alb)
[7]        CautareInAdincime(u);
/*TraversareInAdancime*/                    /*[10.4.1.1.b]*/

procedure CautareInAdancime(Tip_Nod u)
[1]  culoare[u]=gri;
[2]  timp=timp+1; i[u]=timp;
[3]  pentru (fiecare nod v adiacent lui u)
[4]    daca (culoare[v] este alb)
[5]      sp[v]=u;
[6]      CautareInAdancime(v)
         □ /*daca*/
[7]  culoare[u]=negru;
[8]  timp=timp+1; f[u]=timp;
/*CautareInAdancime*/
```

- **Analiza algoritmului.**
  - Liniile 1-3 şi 5-7 ale lui ***TraversareInAdâncime*** se execută pentru fiecare nod, deci necesită un timp proporţional cu O*(N)*, excluzând timpul necesar execuţiei apelurilor procedurii de căutare propriu-zise.
  - Procedura ***CăutareInAdâncime*** este apelată exact odată pentru fiecare nod v∈N, deoarece ea este invocată doar pentru noduri albe şi primul lucru pe care îl face este să coloreze respectivul nod în gri.
  - Liniile 3-6 ale procedurii ***CăutareInAdâncime*** se execută pentru fiecare arc, deci într-un interval de timp proporţional cu `|Adj[v]|` unde este valabilă formula [10.4.1.1.c].

$$\sum_{v \in N} |Adj[v]| = O(a) \qquad [10.4.1.1.c]$$

  - În consecinţă rezultă că costul total al execuţiei liniilor 3-6 ale procedurii ***CautareInAdâncime*** este O*(A)*.
  - **Timpul total de execuţie** al **traversării prin căutare în adâncime** este deci O*(N+A)*.
- În continuare se detaliază această tehnică de căutare pentru diferite modalităţi de implementare a grafurilor.

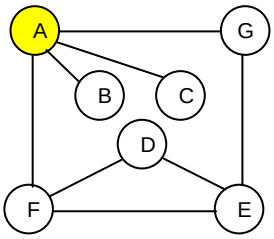
10.4.1.2. Căutare ”în adâncime” în grafuri reprezentate prin structuri de adiacenţe

- Procedura care implementează **căutarea în adâncime** în grafuri reprezentate prin **structuri de adiacenţe** apare în secvenţa [10.4.1.2.a].
  - Procedura **`Traversare1`** completează tabloul `marc[1..maxN]` pe măsură ce sunt traversate (vizitate) nodurile grafului.
  - Tabloul `marc` este poziţionat iniţial pe zero, astfel încât `marc[x]=0` indică faptul că nodul `x` nu a fost încă vizitat.
  - Pe parcursul traversării câmpul `marc` corespunzător unui nod `x` se completează în momentul începerii vizitării cu valoarea “`id`”, valoare care se incrementează la fiecare nod vizitat şi care indică faptul că nodul `x` este cel de-al `id`-lea vizitat.
  - Procedura de traversare utilizează procedura recursivă **`CautaInAdâncime`** care realizează vizitarea tuturor nodurilor aparţinătoare acelei componente conexe a grafului, căreia îi aparţine nodul furnizat ca parametru.
  - Se face precizarea că procedura **`Traversare1`** din secvenţa [10.4.1.2.a] se referă la reprezentarea **TDA graf** bazată pe **structuri de adiacenţă implementate cu ajutorul listelor înlănţuite simple**.

```
-------------------------------------------------------------
/*Traversarea "în adancime" a grafurilor reprezentate prin
structuri de adiacenţe implementate cu ajutorul listelor
înlănţuite simple - varianta pseudocod*/

int marc[maxN]; /*tabloul marc*/
int id;

subprogram CautaInAdincime(int x);
  Ref_Tip_Nod t;

  id= id+1; marc[x]= id;
  *scrie(t->nume);
  t= Stradj[x];
  cat timp (t<>null)                          /*10.4.1.2.a*/
    |daca (marc[t->nume]==0)
    |    CautaInAdincime(t->nume);
    |t= t->urm
    |□ /*cat timp*/
/*CautaInAdincime*/

subprogram Traversare1;
  int x;

  id= 0;
  pentru (x=1 la N)
    marc[x]= 0;
  pentru (x=1 la N)
    daca (marc[x]==0)
     |CautaInAdincime(x);
     |*scrie_rand_ecran (writeln);
     |□ /*daca*/
 /*Travesare1*/
```

- **Vizitarea unui nod** presupune parcurgerea tuturor arcelor conexe lui, adică parcurgerea listei sale de adiacenţe şi verificarea pentru fiecare arc în parte, dacă el conduce la un nod care a fost sau nu vizitat.
    - În caz că nodul este nevizitat procedura se apelează recursiv pentru acel nod.
- Procedura **`Traversare1`**, parcurge tabloul `marc` şi apelează procedura **`CautaInAdâncime`** pentru nodurile nevizitate, până la traversarea integrală a grafului.
    - Trebuie observat faptul că fiecare apel al procedurii **`CautaInAdâncime`** realizat din cadrul procedurii **`Traversare1`** asigură parcurgerea unei **componente conexe a grafului** şi anume a componentei conexe care conţine nodul selectat.
- În figura 10.4.1.2.a apare **urma execuţiei** algoritmului de căutare în adâncime pentru graful (a) din figură.
    - **Structura de adiacenţe** aferentă grafului apare în aceeaşi figură (b).
    - **Evoluţia conţinutului stivei** apare în în aceeaşi figură (c).
    - Se menţionează faptul că nodurile structurii de adiacenţe au ataşat un **exponent fracţionar** care precizează:
        - La **numărător** momentul la care **este descoperit nodul**, adică momentul la care este introdus în stivă (colorarea în gri).
        - La **numitor**, momentul **terminării explorării nodului**, adică momentul la care este scos din stivă (colorarea în negru).
- Graful este **traversat** drept consecinţă a apelului **`CautăInAdâncime`**`(A)`, efectuat în ciclul **`for`** al procedurii **`Traversare1`**.
- În continuare se prezintă la nivel de detaliu parcurgerea prin **căutare în adîncime** a grafului (a) din figura 10.4.1.2.a.
    - Nodul de pornire A este introdus în stivă la momentul 1.
    - Pentru nodul A se parcurge lista sa de adiacenţe, primul arc traversat fiind AF, deoarece F este primul nod din lista de adicenţe a lui A.
    - În continuare se apelează procedura **`CautăInAdâncime`** pentru nodul F, în consecinţă nodul F este introdus în stivă la momentul 2 şi se traversează arcul FA, A fiind primul nod din lista de adiacenţe a lui F.
    - Întrucât nodul A a fost deja descoperit (intrarea sa în tabloul `marc` conţine o valoare nenulă), se alege în continuare arcul FE, E fiind nodul următor în lista de adiacenţe a lui F.
    - Nodul E se introduce în stivă la momentul 3.
    - Se traversează în continuare arcul EG (G se introduce în stivă la momentul 4) G fiind primul nod din lista de adiacenţe a lui E.
    - În continuare se traversează arcul GE respectiv GA (nodurile E respectiv A fiind deja descoperite), moment în care se termină parcurgerea listei de adiacenţe a lui G, fapt realizat la timpul 5.

- Se elimină G din stivă, se revine în lista nodului E şi se continuă parcurgerea listei sale de adiacenţe traversând arcele EF (nodul F a fost deja vizitat) şi ED.
- Ca atare nodul D este descoperit la momentul 6 şi în continuare vizita lui D presupune traversarea arcelor DE şi DF care niciunul nu conduce la un nod nou.
- Terminarea parcurgerii listei de adiacenţe a lui D are drept consecuinţă finalizarea vizitării lui şi scoaterea din stivă la momentul 7.
- Se revine în lista de adiacenţe a lui E. Deoarece D a fost ultimul nod din lista de adiacenţe a lui E, vizita lui E se încheie la momentul 8 şi revine în nodul F a cărui vizitare se încheie prin parcurgerea arcului FD (D deja vizitat).
- Se elimină F din stivă la momentul 9, se revine în lista lui A şi se găseşte nodul C nevizitat încă.
- C se introduce în stivă la momentul 10, i se parcurge lista care îl conţine doar pe A deja vizitat şi este extras din stivă la momentul 11.
- Se continuă parcurgerea listei de adiacenţe a nodului A şi se găseşte nodul B care suferă un tratament similar lui C, la momentele 12 respectiv 13.
- În final se ajunge la sfârşitul listei lui A, A se scoate din stivă la momentul 14 şi procesul de traversare se încheie.



(a)

$^{1/14}A \rightarrow {}^{2/}F \rightarrow {}^{10/}C \rightarrow {}^{12/}B \rightarrow G$

$^{12/13}B \rightarrow A$

$^{10/11}C \rightarrow A$

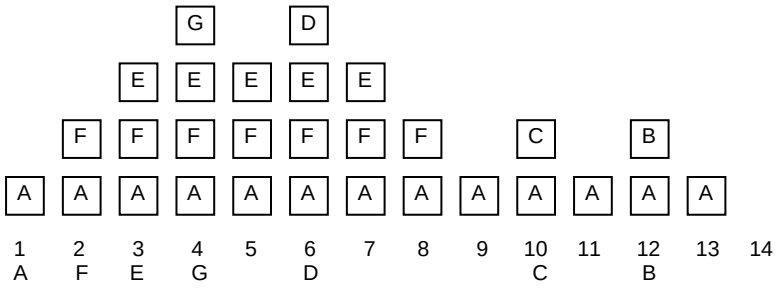
$^{6/7}D \rightarrow F \rightarrow E$

$^{3/8}E \rightarrow {}^{4/}G \rightarrow F \rightarrow {}^{6/}D$

$^{2/9}F \rightarrow A \rightarrow {}^{3/}E \rightarrow D$

$^{4/5}G \rightarrow E \rightarrow A$

(b)



(c)

Fig.10.4.1.2.a. Urma execuţiei algoritmului recursiv de căutare în adâncime

- Un alt mod de a urmări desfăşurarea operaţiei de căutare în adâncime este acela de a redesena graful traversat pornind de la **apelurile recursive** ale procedurii **`CautaInAdâncime`**, ca în figura 10.4.2.1.b.

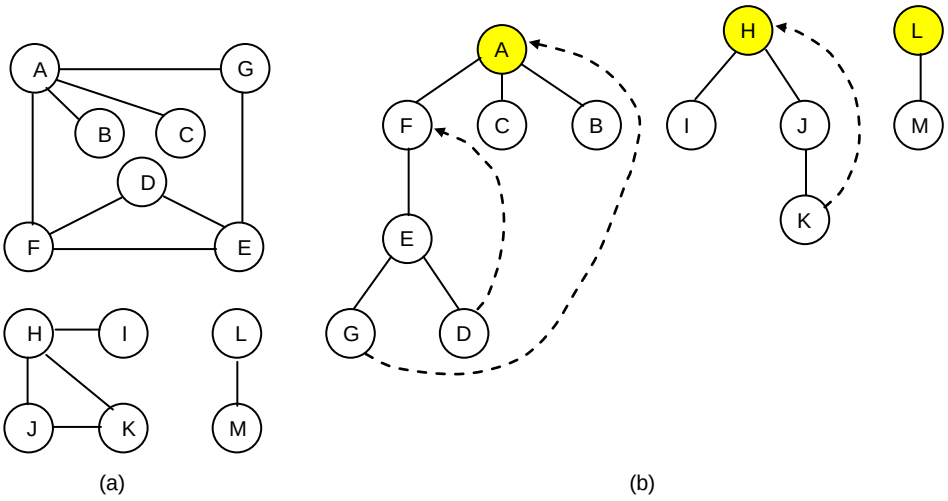


Fig.10.4.2.1.b. Arbori de căutare în adâncime

- În figura 10.4.2.1.b:
  - O **linie continuă** indică faptul că nodul aflat la extremitatea sa inferioară, a fost găsit în procesul de căutare în lista de adiacenţe a nodului aflat la extremitatea sa superioară şi nefiind vizitat la momentul considerat, s-a realizat pentru el un apel recursiv al procedurii de căutare.
  - O **linie punctată** indică un nod descoperit în lista de adiacenţe a nodului sursă pentru care apelul recursiv nu se realizează, deoarece nodul a fost deja vizitat sau este în curs de vizitare, adică este memorat în stiva asociată prelucrării.
    - Ca atare condiţia din instrucţia **`if`** a procedurii **`CautăInAdâncime`** nu este îndeplinită nodul fiind deja marcat cu `vizitat` în tabloul `marc` şi în consecinţă pentru acest nod **nu** se realizează un apel recursiv al procedurii.
- Aplicând această metodă, pentru fiecare componentă conexă a unui graf se obţine un **arbore de acoperire** ("**spanning tree**") numit şi **arbore de căutare în adâncime** al componentei.
  - Traversarea în **preordine** a arborelui de căutare în adâncime**,** furnizează nodurile în ordinea în care sunt **prima dată** întâlnite în procesul de căutare.
  - Traversarea în **postordine** a arborelui de căutare în adâncime, furnizează nodurile în ordinea în care **cercetarea lor se încheie**.
- Este important de subliniat faptul că întrucât ramurile arborilor de acoperire, materializează arcele grafului, **mulţimea (pădurea) arborilor de căutare în adâncime** asociaţi unui graf reprezintă **o altă metodă de reprezentare grafică** a **grafului**.
- O proprietate esenţială a **arborilor de căutare în adâncime** pentru grafuri neorientate este aceea că **liniile punctate** indică întotdeauna un **strămoş** al nodului în cauză.
- În orice moment al execuţiei algoritmului, nodurile grafului se împart în trei **clase**:

- (1) **Clasa I-a** conţine nodurile pentru care procesul de vizitare s-a terminat (colorate în **negru**).
- (2) **Clasa II-a** conţine nodurile care sunt în curs de vizitare (colorate în **gri**).
- (3) **Clasa III-a** conţine nodurile la care nu s-a ajuns încă (colorate în **alb**).

- (1) În ceea ce priveşte **clasa I-a** de noduri, datorită modului de implementare a procedurii de căutare, nu va mai fi selectat nici un arc care indică vreun astfel de nod, motiv pentru care aceste arce **nu** se reprezintă în structura arbore.
- (2) În ceea ce priveşte **clasa a III-a** de noduri, aceasta cuprinde nodurile pentru care se vor realiza apeluri recursive şi arcurile care conduc la ele vor fi marcate cu **linie continuă** în arbore.
- (3) Mai rămân nodurile din **clasa II-a**: acestea sunt nodurile care au apărut cu siguranţă în drumul de la nodul curent la rădăcina arborelui. Ele sunt colorate în gri şi sunt memorate în stiva asociată căutarii.
  - Ca atare, orice arc procesat care indică vreunul din aceste noduri apare reprezentat cu **linie punctată** în **arborele de căutare în adâncime**.
- În concluzie:
  - Arcele marcate **continuu** în figura 10.4.1.2.b se numesc **arce de arbore.**
  - Arcele marcate **punctat** se numesc **arce de retur.**
- Din punct de vedere **formal**, dacă `x` şi `y` sunt noduri ale grafului ce urmează a fi traversat atunci:
  - (1) **Arcul de arbore** este acel arc `(x,y)` al grafului pentru care instanţa de apel a procedurii recursive **`CautăInAdâncime`**`(x)` apelează instanţa **`CautăInAdâncime`**`(y)`.
    - La momentul apelului, nodul `y` aparţine clasei a III-a el fiind colorat în **alb**, adică nevizitat**.**
  - (2) **Arcul de retur** `(x,y)` este un arc al grafului întâlnit în procesul de vizitare al nodului `x`, adică întâlnit în lista lui de adiacenţe şi care conduce la un nod `y` aparţinând clasei I-a sau a II-a.
    - Cu alte cuvinte `y` este un nod pentru care procesul de vizitare s-a terminat (colorat în **negru**) sau care se află în stiva asociată traversării (colorat în **gri**).
    - Arcul de retur`(x,y)` indică de fapt un **nod strămoş** al nodului x.

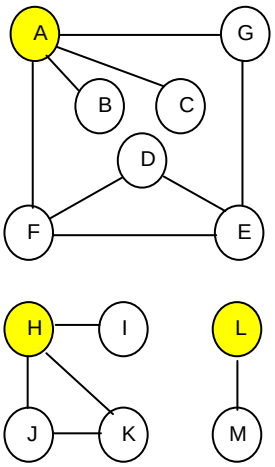
10.4.1.3. Căutare ”în adâncime” în grafuri reprezentate prin matrici de adiacenţe

- Procedura care implementează **căutarea în adâncime** în grafuri reprezentate prin **matrici de adiacenţe** apare în secvenţa [10.4.1.3.a].
  - Traversarea listei de adiacenţe a unui nod din **structura de adiacenţe**, se transformă în parcurgerea **liniei** corespunzătoare nodului din **matricea de adiacenţe**, căutând valori adevărate (care marchează arce).

- Şi în acest caz, selecţia unui arc care conduce la un nod **nevizitat** este urmată de un **apel recursiv** al procedurii de căutare pentru nodul respectiv.

- Datorită modului diferit de reprezentare a grafului, arcele conectate la noduri sunt examinate într-o altă ordine, motiv pentru care **arborii de căutare în adâncime** care alcătuiesc pădurea corespunzătoare grafului diferă ca formă.

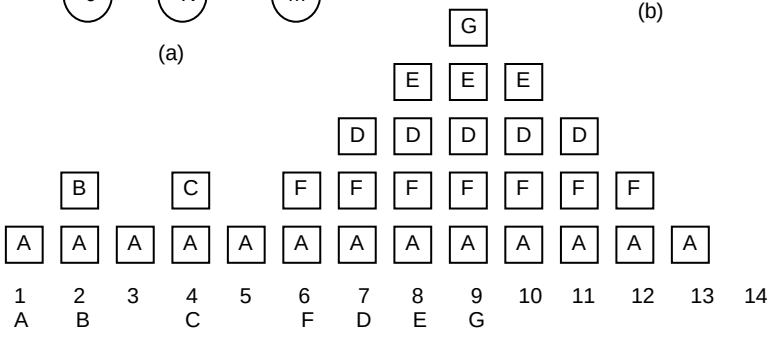
```
-------------------------------------------------------------
/*Căutare "în adâncime" în grafuri reprezentate prin matrici
de adiacenţe - varianta pseudocod*/

boolean A[numarNoduri]; /*matricea de adiacenţe*/
int marc[numarNoduri];  /*tabloul marc pentru evidenţa
                          nodurilor*/
int id; /*contor de noduri*/

procedure CautaInAdincime1(int x);
  int t;

  id= id+1; marc[x]= id;
  *scrie(x);                                  /*10.4.1.3.a*/
  pentru (t=1 la N)
    daca (A[x,t])
      daca (marc[t]==0)
        CautaInAdincime1(t);
/*CautaInAdincime1*/
-------------------------------------------------------------
```



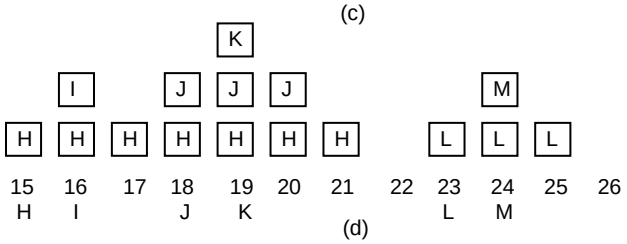
(a)

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	1	0	0	1	1						
B	1	0	0	0	0	0	0						
C	1	0	0	0	0	0	0						
D	0	0	0	0	1	1	0						
E	0	0	0	1	0	1	1						
F	1	0	0	1	1	0	0						
G	1	0	0	0	1	0	0						
H								0	1	1	1		
I								1	0	0	0		
J								1	0	0	1		
K								1	0	0	1		
L												0	1
M												1	0

(b)



(c)



(d)

Fig.10.4.1.3.a.Căutare în adâncime în grafuri reprezentate prin matrice de adiacenţe

- Pentru graful din figura 10.4.1.3.a (a), a cărui matrice de adiacenţe apare în aceeaşi figură (b), evoluţia stivei asociate traversării apare în (c) şi (d) iar arborii de căutare în adâncime corespunzători apar în fig. 10.4.1.3.b.

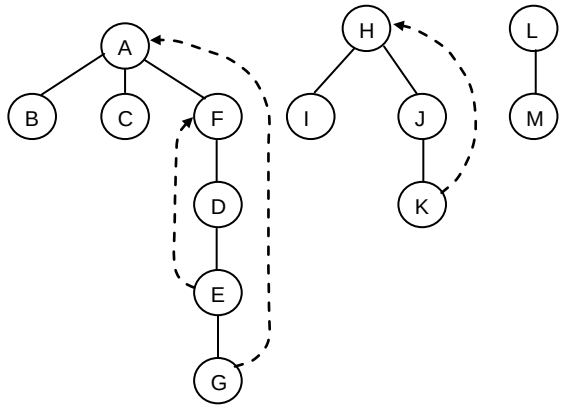


Fig.10.4.1.3.b. Pădure de arbori de căutare în adâncime în grafuri reprezentate prin matrice de adiacenţe

- Se observă unele diferenţe faţă de pădurea de arbori reprezentată în fig. 10.4.2.1.b. corespunzătoare aceluiaşi graf.
    - Prin aceasta se subliniază faptul că **o pădure de arbori de căutare în adâncime** nu este altceva decât o altă manieră de reprezentare a unui graf, a cărei alcătuire particulară depinde de:
        - (1) Metoda de traversare a grafului.
        - (2) Reprezentarea internă utilizată pentru graf.
- Din punct de vedere al **eficienţei**, căutarea în adâncime în grafuri reprezentate prin matrici de adiacenţe necesită un timp proporţional cu $O(n^2)$.
    - Acest lucru este evident întrucât în procesul de traversare este verificat fiecare element al matricei de adiacenţe.
- Căutarea în adâncime rezolvă unele **probleme fundamentale** ale prelucrării grafurilor.
- (1) Deoarece procedura de parcurgere a unui graf se bazează pe traversarea pe rând a componentelor sale conexe, **numărul componentelor conexe** ale unui graf poate fi determinat simplu contorizând numărul de apeluri ale procedurii **`CautăInAdâncime`** efectuat din ultima linie a procedurii **`Traversare1`**.
- (2) Căutarea în adâncime permite verificarea simplă a **existenţei ciclurilor** într-un graf.
    - Astfel, un graf conţine un **ciclu**, dacă şi numai dacă procedura **`CautăInAdâncime`** descoperă o valoare diferită de zero în tabloul `marc`.
    - Această înseamnă că se parcurge un arc care conduce la un nod care a mai fost vizitat, deci graful conţine un **ciclu**.
    - În cazul grafurilor **neorientate** trebuie însă să se ţină cont de reprezentarea dublă a fiecărui arc, care poate produce confuzii.

- La reprezentarea grafurilor prin păduri de arbori de căutare, **liniile punctate** sunt acelea care închid **ciclurile**.

10.4.1.4. Căutare ”în adâncime” nerecursivă

- Este cunoscut faptul că orice algoritm **recursiv** poate fi transformat într-un algoritm echivalent **iterativ**.
  - Tehnica realizării acestei transformări se bazează pe definirea şi utilizarea de către programator a unei structuri de date **stivă** [Cr00].
- În secvenţa [10.4.1.4.a] apare echivalentul iterativ al algoritmului de căutare în adâncime bazat pe această tehnică.
- Graful se consideră reprezentat prin **structuri de adiacenţe** implementate cu ajutorul **listelor înlănţuite simple**.

```
-----------------------------------------------------------------
/*Traversarea prin CA a grafurilor reprezentate prin SA
implementate cu ajutorul listelor înlănţuite simple - se
utilizează TDA Stiva - Varianta iterativă*/

const maxN = 100;

typedef struct Nod * Ref_Tip_Nod;

/*structura unui nod*/
typedef struct Nod
 {
    int nume;
    Ref_Tip_Nod urm;
 } Tip_Nod;

 /*structura de adiacenţe*/
 Ref_Tip_Nod  StrAdj[maxN];

 int id; /*contor noduri*/
 int x;  /*nod curent*/
 int marc[maxN]; /*tablou evidenţă noduri*/

 Tip_Stiva s; /*stiva*/

 void CautaInAdincimeNerecursiv(int x)
   {
     Ref_Tip_Nod t;

     push(x,s); /*amorsare proces*/
     do
       {
         x= varfSt(s); pop(s);
         id= id + 1; marc[x]= id; /*colorare în negru*/
         *scrie(x);
         t= Stradj[x];
         while (t<>null)
           {
             if (marc[t^.nume]==0)       /*10.4.1.4.a*/
               {
```

```
                push(t^.nume);  /*nodul curent în stivă*/
                marc[t^.nume]=-1; /*colorare în gri*/
              }
            t= t^.urm;
          }
       }
    while (stivid(s));
  } /*CautaInAdincimeNerecursiv*/

void Traversare2
{
   id= 0; /*iniţializare contor noduri*/
   initializeaza(s); /*iniţializare stivă*/
   /*initializare - colorare în alb a tuturor nodurilor*/
   for (x=1;x<=N;x++)
      marc[x]= 0;
   /*traversare nerecursivă graf*/
   for (x=1;x<=N;x++)
      if (marc[x]==0)
        {
          CautaInAdincimeNerecursiv(x);
          *scrie_rand;
        };
 } /*Traversare2*/
-------------------------------------------------------------
```

- Nodurile care au fost atinse dar nu au fost încă cercetate sunt păstrate într-o structură de date stivă `s`.
  - Pentru gestionarea **stivei** sunt utilizaţi operatorii specifici definiţi pe **TDA Stivă**: ***initializează(s)***, ***push(x,s)***, ***pop(s)***, ***stivid(s)*** şi ***varfSt(s)*** (Vol.1 &6.5.3.1).
- În timpul vizitării unui nod, pe măsura parcurgerii nodurilor sale adiacente, se introduc în stivă acele noduri care nu au fost încă vizitate şi care nu sunt deja în stivă.
  - Introducerea în stivă a unui nod se marchează cu valoarea `-1` în tabloul `marc`, ceea ce echivalează cu colorarea lui în **gri**.
- În **varianta recursivă**, păstrarea evidenţei nodurilor parţial vizitate este realizată în mod transparent pentru utilizator de către variabila locală `t` a procedurii recursive **`CautăInAdâncime`**.
- În **varianta iterativă** păstrarea evidenţei nodurilor parţial vizitate se poate realiza memorând în stivă numele nodurilor parcurse de variabila `t`, respectiv valorile `t^.nume`.
- În plus, pentru a cunoaşte cu precizie starea nodurilor, se utilizează tehnica de a marca în tabloul `marc` diferitele categorii de noduri cu **valori distincte**. Asfel:
  - (1) Se marchează cu valoarea 0 nodurile care **nu** au fost încă atinse (nodurile **albe**).
  - (2) Se marchează cu valoarea –1 nodurile care au fost descoperite în procesul de vizitare şi care au fost introduse în stivă (nodurile **gri**).
  - (3) Se marchează cu o valoare pozitivă corespunzătoare ordinii de parcurgere, nodurile care au fost extrase din stivă şi pentru care vizitarea este în curs de desfaşurare (nodurile **negre**).

- Se observă faptul că în acest caz, nodurile în curs de vizitare sunt marcate în negru, în momentul în care sunt extrase din stivă.

- Această metodă a fost aplicată în procedura **`Traversare2`** din secvenţa [10.4.1.4.a].

- În figura 10.4.1.4.a se prezintă modul în care se desfăşoară **căutarea în adâncime** în **variantă nerecursivă** pentru graful din figură (a).

    - Forma simplificată a reprezentării bazate pe structuri de adiacenţe a grafului apare în (b), iar evoluţia stivei asociate în (c).

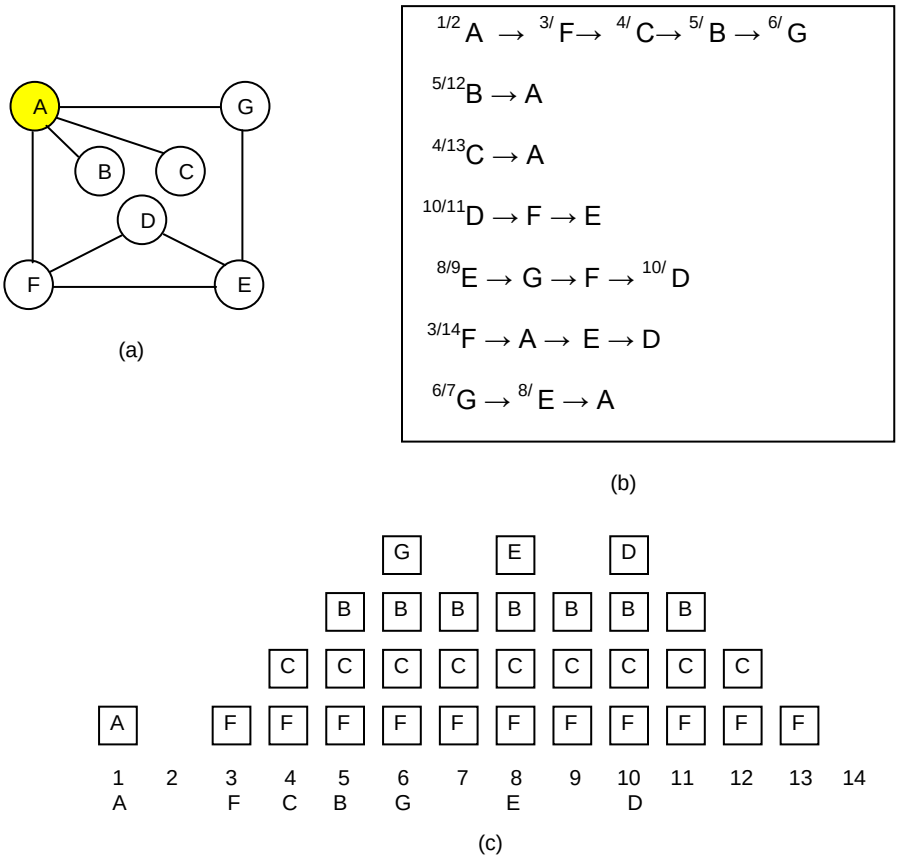


Fig.10.4.1.4.a. Căutare în adâncime iterativă. Evoluţia conţinutului stivei.

- Presupunând că primul nod pentru care se apelează procedura **`CautăÎnAdâncimeNerecursiv`** este nodul `A`, acesta este introdus în stivă în cadrul procesului de amorsare a traversării, după care este imediat scos din stivă procedându-se la explorarea listei lui de adiacenţe.

- În lista sa de adiacenţe se descoperă nodurile `F`, `C`, `B`, şi `G` care se introduc în stivă în această ordine.

- Terminându-se vizitarea listei nodului `A`, se revine şi se extrage din stivă în vederea vizitării, ultimul nod introdus adică nodul `G` procedându-se la parcurgerea listei sale de adiacenţe.

- Se ajunge astfel la nodul `E` nevizitat încă, care este şi el introdus în stivă.

- Se parcurge în continuare nodul `A`, deja vizitat, prezent în lista de adiacenţe a nodului `G`, moment la care vizitarea listei lui `G` se încheie si se revine în bucla principală.

- Se trece la vizitarea nodului `E` care este extras din stivă.
- Procesul continuă în aceeaşi manieră, adică în lista de adicenţe a lui `E` este descoperit nodul `D` ş.a.m.d. până la golirea integrală a stivei de noduri.
- **Procesul integral** poate fi urmărit în figura 10.4.1.4.a (c), în care sunt marcate şi momentele la care este introdus în stivă respectiv este extras din stivă fiecare nod.
    - Acestă informaţie este de altfel precizată şi în reprezentarea schematică a structurii de adiacenţe a grafului, unde spre exemplu notaţia $^{3/14}$`F` precizează faptul că nodului `F` este introdus în stivă la momentul `3` şi este extras la momentul `14`.
- Din implementare se observă că în cazul **variantei iterative**, ordinea de parcurgere a nodurilor **nu este identică** cu cea rezultată din **varianta recursivă**.
    - Acest lucru poate fi însă evitat simplu, dacă în varianta iterativă, nodurile se introduc în stivă în ordinea inversă apariţiei lor în lista de adiacenţe.
    - În mod inerent, vor rezulta şi **arbori de căutare în adâncime** cu o structură specifică.
- De regulă însă, în marea majoritate a situaţiilor reale, nu este necesar ca parcurgerea să se raporteze strict la ordinea rezultată din modelul recursiv.
- De obicei, din punctul de vedere al scopului urmărit toate variantele de parcurgere în adâncime sunt echivalente.

10.4.1.5. Analiza căutării ”în adâncime”

- Se consideră un graf `G` cu `a` arce şi `n` noduri implementat prin **structuri de adiacenţe**.
- **Traversarea** sa în baza **tehnicii de căutare în adâncime** necesită un timp de calcul de ordinul O*(n+a)*.
    - Termenul `n` provine din parcurgerea nodurilor şi marcarea acestui fapt în tabloul `marc.`
    - Termenul `a` provine din parcurgerea tuturor arcelor.
- Presupunând că `n<<a`, timpul necesar căutării este O*(a)*.
    - Acest lucru rezultă din faptul că pentru nici un nod procedura de căutare **nu** este apelată mai mult decât odată, deoarece după primul apel **`CautăInAdâncime`**`(x)`, `marc[x]` este asignat cu `id`-ul curent şi în continuare nu se mai realizează nici un apel pentru nodul `x`.
    - În procesul de traversare sunt însă verificate **toate arcele grafului**.
    - În consecinţă timpul total petrecut cu parcurgerea listelor de adiacenţe este proporţional cu suma acestor liste deci este O*(a)*.

10.4.2. Traversarea grafurilor prin tehnica căutării ”prin cuprindere” (”Breadth-First Search”)

- O altă manieră sistematică de traversare a nodurilor unui graf o reprezintă **căutarea prin cuprindere** (“**breadth first search**”).
- **Căutarea prin cuprindere** se bazează pe următoarea tehnică:
  - Pentru fiecare nod vizitat `x`, se caută în imediata sa vecinătate “**cuprinzând**” în vederea vizitării toate nodurile adiacente lui.
- Pentru implementarea acestei tehnici de parcurgere, în locul stivei din metoda de căutare anterioară, pentru reţinerea nodurilor vizate se poate utiliza o **structură de date coadă**.
- Schiţa de principiu a algoritmului de parcurgere apare în secvenţa [10.4.2.a].
  - Se precizează faptul că s-a utilizat o structură de date **coadă** `(Q)` asupra căreia acţionează operatori specifici [Vol.1,&6.5.4.1, Cr00].

```
-------------------------------------------------------------
/*Căutare prin cuprindere. Schiţa de principiu. Varianta 1
pseudocod - se utilizeaza TDA Coada*/

subprogram CautaPrinCuprindere(Tip_Nod x, Tip_Multime T)
/*se parcurg toate nodurile adiacente lui x prin căutare
prin cuprindere. Se construieşte mulţimea T care include
arborele de parcugere prin cuprindere*/

  Coada_De_Noduri Q;
  Tip_Nod  x,y;
  int marc[maxN]; /*tabloul marc*/        /*[10.4.2.a]*/

  marc[x]= vizitat; /*marcheaza nodul x vizitat*/
  Adauga(x,Q);   /*se introduce nodul de pornire în coadă*/
  cat timp (not Vid(Q)) executa
   |x= Cap(Q);  /*citeste în x nodul din capul cozii*/
   |Scoate(Q);  /*scoate nodul din coadă*/
   |pentru (fiecare nod y adiacent lui x) executa
   |  daca (marc[y] este nevizitat)
   |   |marc[y]= vizitat;
   |   |Adauga(y,Q); /*adaugă-l pe y în coadă*/
   |   |INSERTIE((x,y),T) /*inserează arcul (x,y) în
   |   |                    arborele de parcurgere asociat*/
   |   |□ /*daca*/
   |□ /*cat timp*/
 /*CautaPrinCuprindere*/
-------------------------------------------------------------
```

- Algoritmul din secvenţa [10.4.2.a] inserează cu ajutorul operatorului ***INSERTIE*** arcele parcurse ale grafului într-o mulţime `T` care materializează traversarea şi despre care se presupune că este iniţial vidă.
- Se presupune de asemenea că tabloul `marc` este iniţializat integral cu marca “`nevizitat`”.
- Procedura lucrează pentru o singură componentă conexă.
  - Dacă graful **nu** este conex, procedura **`CautăPrinCuprindere`** trebuie apelată pentru fiecare componentă conexă în parte.

- Se atrage atenţia asupra faptului că în cazul căutării prin cuprindere, un nod trebuie marcat cu `vizitat` **înaintea** introducerii sale în coadă pentru a se evita plasarea sa de mai multe ori în această structură.

10.4.2.1. Căutarea "prin cuprindere", varianta CLR

- **Căutarea prin cuprindere** este una dintre cele mai cunoscute metode de căutare, utilizate printre alţii de către **Djikstra** şi **Prim** în celebrii lor algoritmi [CLR92].
- Ca şi în cazul căutării în adâncime, pentru a ţine evidenţa procesului de căutare nodurile sunt colorate în alb, gri şi negru.
- Toate nodurile sunt colorate iniţial alb şi ele devin mai târziu gri, apoi negre.
    - La prima descoperire a unui nod, acesta este colorat în gri.
    - La terminarea vizitării unui nod, acesta este colorat în negru.
    - Nodurile gri şi negre sunt noduri deja descoperite în procesul de căutare, dar ele sunt diferenţiate pentru a se asigura funcţionarea corectă a căutării.
    - Nodurile gri care pot avea ca adiacenţi şi noduri albe, marchează frontiera dintre nodurile vizitate şi cele nevizitate.
    - De fapt, nodul curent `n` se colorează în negru când toate nodurile adiacente lui au fost deja vizitate.
- Şi în cazul căutării prin cuprindere se poate construi un **subgraf** `sp` al predecesorilor nodurilor vizitate.
    - Ori de câte ori un nod `v` este descoperit pentru prima oară în procesul de căutare la parcurgerea listei de adiacenţe a nodului `u`, acest lucru se marchează prin `sp[v]=u`.
- După cum s-a mai precizat **subgraful predecesorilor** este de fapt un **arbore liber**, reprezentat printr-un tablou liniar în baza relaţiei "indicator spre părinte".
- Procedura **`TraversarePrinCurpindere`** din secvenţa [10.4.2.1.a] presupune graful `G=(N,A)` reprezentat prin **structuri de adiacenţe**.
    - Culoarea curentă a fiecărui nod `u∈N` este memorată în tabloul `culoare[u]`.
    - Predecesorul nodului `u`, adică nodul în lista căruia a fost descoperit, este înregistrat în tabloul `sp[u]`.
    - Dacă `u` **nu** are predecesor (adică este nodul de pornire) se marchează acest lucru prin `sp[u]=`**`null`**.
- În cadrul procesului de **traversare prin cuprindere a grafului**, se poate calcula şi **distanţa** de la nodul de start la fiecare din nodurile grafului.
    - Distanţa de la sursă la nodul curent `u` calculată de către algoritm este memorată în `d[u]`.
    - Unitatea de măsură a distanţei este **numărul de arce traversate.**
- Algoritmul de traversare utilizează o **coadă `FIFO`** notată cu `Q` pentru a gestiona nodurile implicate în procesul de căutare, scop în care face uz de operatorii consacraţi pentru această structură de date.

```
/*Căutarea prin cuprindere. Schiţa de principiu. Varianta 2
pseudocod (Cormen, Leiserson, Rivest) - se utilizeaza TDA
Coada*/

  Tip_Coada Q; /*structura COADA*/

  CautaPrin Cuprindere(Tip_Graf G, Tip_Nod s)
[1]  pentru (fiecare nod u ∈ N(G)-{s}) /*s este nodul de
                                        start*/
[2]   |culoare[u]=alb;
[3]   |d[u]=∞;
[4]   |sp[u]=null;
      □ /*pentru*/
[5]  culoare[s]=gri;  /*s este nodul de start*/
[6]  d[s]=0;
[7]  sp[s]=null;
[8]  Initializeaza(Q); Adauga(s,Q);      /*10.4.2.1.a*/
[9]  cat timp not Vid(Q)do
[10]  |u=Cap(Q);
[11]  |pentru (fiecare v ∈ Adj[u])
[12]  |  daca (culoare[v]=alb)
[13]  |    |culoare[v]=gri;
[14]  |    |d[v]=d[u]+1;
[15]  |    |sp[v]=u;
[16]  |    |Adauga(v,Q);
      |    □ /*daca*/
[17]  |  Scoate(Q);
[18]  |  culoare[u]=negru;
      □ /*cat timp*/
```

- **Funcţionarea algoritmului.**
- Liniile 1- 4 iniţializează structurile de date, adică:
  - Toate nodurile `u` cu excepţia nodului de start sunt marcate cu `alb` în tabloul `culoare`.
  - Distanţele corespunzătoare tuturor nodurilor sunt setate pe ∞ (`d[u]=∞`).
  - Părintele fiecărui nod este iniţializat cu `nil` (`sp[u]=null`).
- Linia 5 marchează nodul `s` furnizat ca parametru al procedurii de căutare cu gri, el fiind considerat **sursa** (originea) procesului de căutare.
- Liniile 6 şi 7 iniţializează `d[s]` pe zero şi `sp[s]` cu **`null`**.
- Linia 8 iniţializează coada `Q` şi îl adaugă pe `s` în coadă.
  - De altfel coada `Q` va conţine numai noduri colorate în **gri**.
- Bucla principală a programului apare între liniile 9-18 şi ea iterează atâta vreme cât există noduri (gri) în coadă.
  - Nodurile gri din coadă sunt noduri descoperite în procesul de căutare în liste de adiacenţe care nu au fost încă epuizate.
- Linia 10 furnizează nodul gri `u` aflat în capul cozii `Q`.
- Bucla **`for`** (liniile 11-16) parcurge fiecare nod `v` al listei de adiacenţe a lui `u`.

- Dacă v este alb, el nu a fost încă descoperit, ca atare algoritmul îl descoperă executând liniile 13-16 adică:
    - Nodul v este colorat în gri.
    - Distanţa d[v] este setată pe d[u]+1 indicînd creşterea acesteia cu o unitate în raport cu părintele nodului.
    - u este memorat ca şi părinte al lui v.
    - În final v este adăugat în coada Q la sfârşitul acesteia.
- După ce toate nodurile listei de adiacenţe a lui u au fost examinate, u este extras din coada Q şi colorat în negru (liniile 17-18).
- **Analiza performanţei**.
- Se analizează **timpul de execuţie** al algoritmului pentru un graf G=(N,A).
- După iniţializare **nici un nod** nu mai este ulterior colorat în alb, ca atare testul din linia 12 asigură faptul că fiecare nod este adăugat cozii cel mult odată şi este scos din coadă **cel mult odată**.
- Operaţiile ***Adauga*** şi ***Scoate*** din coadă consumă un timp O*(1)*, ca atare timpul total dedicat operării cozii Q este O*(N)*.
- Deoarece lista fiecărui nod este scanată integral înaintea scoaterii nodului din coadă, această operaţie se realizează cel mult odată pentru fiecare nod.
    - Întrucât suma lungimilor tuturor listelor de adiacenţă este O*(A)*, timpul total necesar pentru scanarea listelor de adiacenţe este O*(A)*.
- Regia iniţializării este O*(N)* deci timpul total de execuţie al procedurii **CautăPrinCurpindere** este O*(N+A)*.
- În concluzie, căutarea prin cuprindere necesită un timp de execuţie liniar cu numărul de noduri şi cu mărimea listelor de adiacenţe ale reprezentării grafului G.

10.4.2.2. Căutare ”prin cuprindere” în grafuri reprezentate prin structuri de adiacenţe

- În secvenţa [10.4.2.2.a] apare un exemplu de procedură care parcurge un graf în baza tehnicii de **parcurgere prin cuprindere**, utilizând o **structură de date coadă**.
- Graful se consideră reprezentat cu ajutorul **structurilor de adiacenţe** implementate cu **liste înlănţuite simple**.

```
-------------------------------------------------------------
{Traversarea prin cuprindere a grafurilor reprezentate prin
SA implementate cu ajutorul listelor înlănţuite simple - se
utilizeaza TDA Coada - varianta pseudocod}

int x;
int id; /*contor noduri*/
int mark[maxN]; /*tablou evidenţă noduri*/
Tip_Coada Q;

subprogram CautaPrinCuprindere(int x)
  Ref_Tip_Nod t;
```

```
  Adauga(x,Q); /*amorsare proces de căutare*/
  repeta
    x= Cap(Q); Scoate(Q);
    id= id+1; marc[x]= id; /*colorare în negru*/
    *scrie(x);
    t= Stradj[x];
    cat timp (t<>null)
      daca (marc[t->nume]=0)              /*[10.4.2.2.a]*/
        Adauga(t->nume,Q);
        marc[t->nume]=-1 /*colorare în gri*/
        □ /*dacă*/
      t= t->urm;
      □ /*cât timp*/
  pana cand Vid(Q)
  □ /*repetă*/
/*CautaPrinCuprindere*/

program ParcurgerePrinCuprindere;

  id=0;
  Initializeaza(Q);
  /*iniţializare tablou de noduri*/
  pentru (x= 1 la N)
    marc[x]= 0;
  /*căutare prin cuprindere în graf*/
  pentru (x= 1 la N)
    daca (marc[x] este 0)
      CautaPrinCuprindere(x);
      *scrie_rând (writeln);
      □ /*daca*/
/*ParcurgerePrinCuprindere*/
-------------------------------------------------------------
```

- În figura 10.4.4.2.a se prezintă un exemplu de traversare prin cuprindere a unui graf reprezentat prin structuri de adiacenţe.

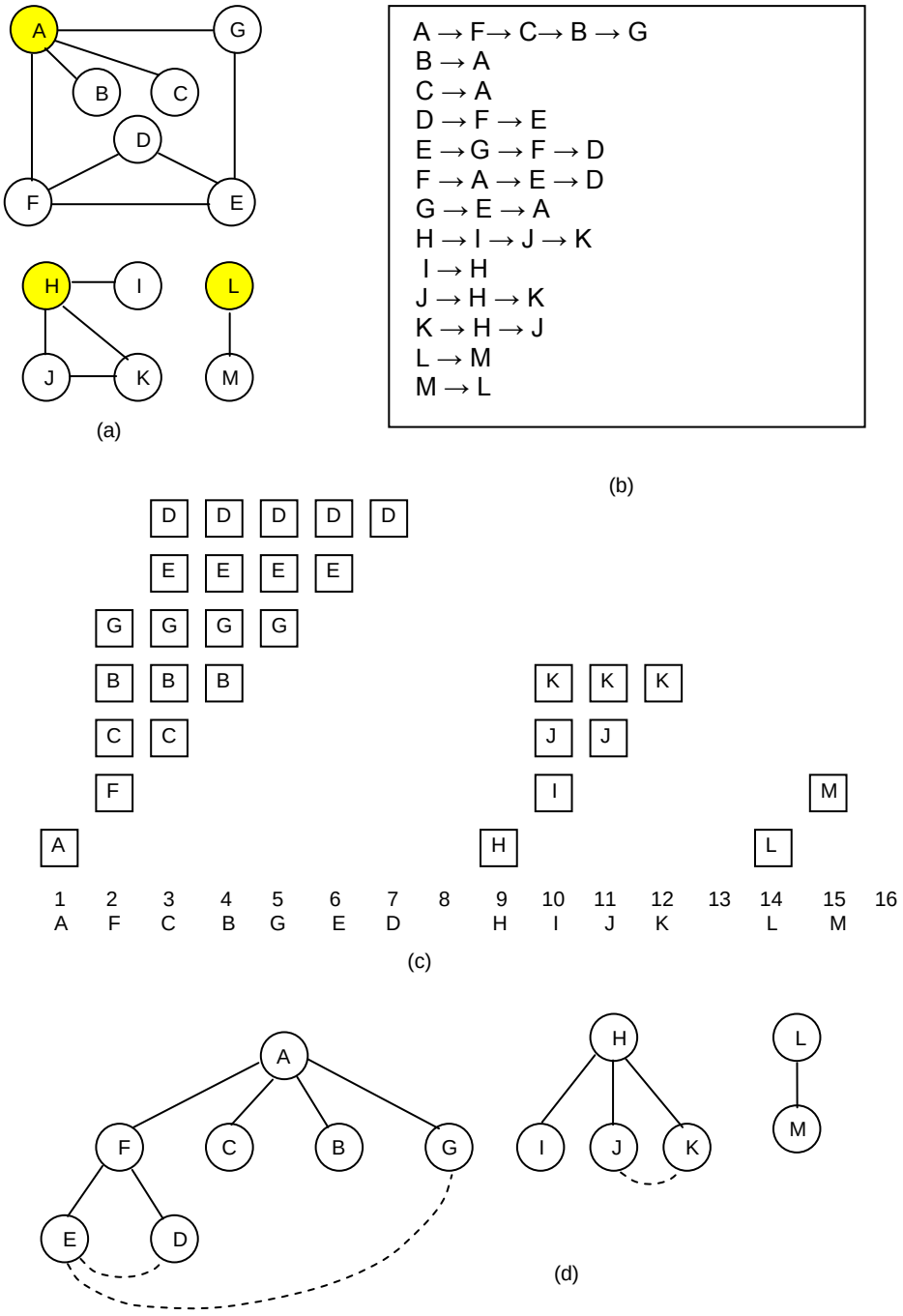


Fig.10.4.2.2.a. Traversarea prin cuprindere a unui graf

- Se consideră graful din figura 10.4.2.2.a (a) reprezentat prin **structura de adiacenţe** din aceeaşi figură (b).
  - Ordinea de parcurgere a arcelor va fi următoarea: `AF,AC,AB,AG,FA,FE,FD,CA,BA,GE,GA,DF,DE,EG,EF,ED,HI,HJ,HK,IH,JH,JK,KH,KJ,IM,MI`
  - Evoluţia conţinutului cozii pe parcursul traversării şi ordinea în care sunt traversate nodurile apar în fig. 10.4.2.2.a (c).
- În mod similar cu traversarea bazată pe căutarea în adâncime, se poate construi o **pădure de arbori de acoperire** ("**spanning trees**") specifică **căutării prin cuprindere**.
  - În acest caz, un arc `(x,y)` se consideră **ramură** a **arborelui de căutare prin cuprindere**, dacă în bucla **`pentru`** a secvenţei [10.4.2.a], respectiv **`cat timp`** a secvenţei [10.4.2.2.a] nodul `y`, respectiv `t->urm,` este vizitat întâia dată venind dinspre nodul `x`.

- În cazul **căutării prin cuprindere** în grafuri neorientate, fiecare arc al grafului care **nu** este o ramură a **arborelui de căutare prin cuprindere**, este un **arc de trecere** care conectează două noduri dintre care niciunul **nu** este strămoşul celuilalt.
- Arborii de căutare aferenţi parcurgerii grafului (a) din figura fig. 10.4.2.2.a apar în aceeaşi figură (d).
    - După cum s-a precizat, aceşti arbori cuprind acele arce care conduc întâia dată la un anumit nod.
- Utilizând arbori de căutare prin cuprindere, verificarea existenţei **ciclurilor** în cadrul unui graf, poate fi realizată în O*(n)* unităţi de timp, indiferent de numărul de arce.
    - După cum s-a precizat în §10.1, un graf cu `n` noduri şi `n` sau mai multe arce, trebuie să aibă cel puţin un **ciclu**.
    - Cu toate acestea şi un graf cu `n` noduri şi `n-1` sau mai puţine arce poate avea cicluri, dacă conţine două sau mai multe componente conexe.
- O modalitate sigură de a determina **ciclurile** unui graf este aceea de a-i construi **pădurea arborilor de căutare prin cuprindere.**
    - Fiecare **arc de trecere**, reprezentat punctat în figură, închide un **ciclu simplu** care cuprinde arcele arborelui care conectează cele două noduri, prin cel mai apropiat strămoş comun al lor.

10.4.2.3. Analiza căutării ”prin cuprindere”

- Din punctul de vedere al **timpului de execuţie**, complexitatea algoritmului de **traversare (căutare) prin cuprindere** este aceeaşi ca şi la căutarea în adâncime.
- Fiecare nod al grafului este plasat în coadă o singură dată, astfel corpul buclei **`cat timp`** (secvenţa [10.4.2.2.a]) se execută o singură dată pentru fiecare nod.
- Fiecare arc `(x,y)` este examinat de două ori, odată pentru `x` şi odată pentru `y`.
- Astfel, dacă graful are `n` noduri şi `a` arce, timpul de execuţie al algoritmului de căutare prin cuprindere este O*(max (n,a))* dacă utilizăm reprezentarea prin structuri de adiacenţă.
    - Deoarece în general $a \geq n$, de regulă se va considera timpul de execuţie al căutării prin cuprindere O*(a)*, ca şi în cazul căutării în adâncime.

10.4.3. Comparaţie între tehnicile fundamentale de traversare a grafurilor

- **Traversarea** unui graf, indiferent de metoda utilizată, are principial un **caracter unitar,** particularizarea rezultând din **structura de date** utilizată drept suport în implementare.
- În ambele tehnici de traversare, nodurile pot fi divizate în trei **clase**:
    - (1) Clasa "**arbore**"– care cuprinde nodurile care au fost extrase din structura de date utilizată în traversare.
        - Sunt nodurile deja vizitate, adică cele colorate în **negru**;
    - (2) Clasa "**vecinătate**" – care cuprinde nodurile adiacente nodurilor traversate sau în curs de traversare.

        - Aceste noduri au fost luate în considerare dar nu au fost încă vizitate şi se găsesc introduse în **structura de date** utilizată în traversare.
        - Sunt nodurile colorate în **gri**;
    - (3) Clasa "**neîntâlnite**" - care cuprinde nodurile la care nu s-a ajuns până la momentul considerat.
        - Sunt nodurile colorate în **alb**.
- Un **arbore de căutare** ia naştere conectând fiecare nod al grafului, cu nodul care a cauzat introducerea sa în **structura de date** utilizată în parcurgere.
- Pentru a parcurge în mod sistematic o **componentă conexă** a unui graf, deci pentru a implementa o procedură "parcurge":
    - (1) Se introduce un nod oarecare (nodul de pornire) al componentei, în clasa "**vecinătate**" şi toate celelalte noduri în clasa "**neîntâlnite**".
    - (2) În continuare, până la vizitarea **tuturor nodurilor grafului** se aplică următorul procedeu:
        - Se selectează un nod – fie acesta x – din clasa "**vecinătate**" şi se mută în clasa "**arbore**".
        - Se trec în clasa "**vecinătate**" toate nodurile din clasa "**neîntâlnite**" care sunt adiacente lui x.
        - Se reia procedeul de la (2).
- **Metodele de parcurgere a grafurilor** se diferenţiază după maniera în care **sunt alese nodurile** care se trec din clasa "**vecinătate**" în clasa "**arbore**".
    - (1) La parcurgerea "**în adâncime**" se alege din vecinătate nodul **cel mai recent întâlnit** (ultimul întâlnit) ceea ce corespunde cu utilizarea unei **stive** pentru păstrarea nodurilor din clasa "**vecinătate**".
    - (2) La parcurgerea "**prin cuprindere**" se alege nodul **cel mai devreme întâlnit** (primul întâlnit) ceea ce presupune păstrarea într-o **coadă** a nodurilor din clasa "**vecinătate**".
    - (3) Se pot utiliza în acest scop şi alte structuri de date, spre exemplu **cozi bazate pe prioritate** în cazul grafurilor ponderate.
- Contrastul dintre primele două metode de parcurgere este şi mai evident în cazul **grafurilor de mari dimensiuni**.
    - Căutarea "**în adâncime**" se avântă în profunzime de-a lungul arcelor grafului memorând într-o stivă punctele de ramificaţie.
    - Căutarea "**prin cuprindere**" mătură prin extindere radială graful, memorând într-o coadă frontiera locurilor deja vizitate.
    - La căutarea "**în adâncime**" graful este explorat căutând noi noduri, cât mai departe de punctul de plecare şi luând în considerare noduri mai apropiate numai în situaţia în care nu se poate înainta mai departe.
    - Căutarea "**prin cuprindere**" acoperă complet zona din jurul punctului de plecare mergând mai departe numai când tot ceea ce este în imediata sa apropiere a fost parcurs.
    - Este însă evident faptul că în ambele situaţii, ordinea efectivă de parcurgere a nodurilor depinde:

- (1) Pe de o parte de **structura de date** utilizată pentru implementarea grafului.
- (2) Pe altă parte de **ordinea** în care sunt introduse iniţial nodurile în această structură.

- În afara diferenţelor rezultate din manierele de operare ale celor două metode, se remarcă diferenţe fundamentale în ceea ce priveşte **implementarea**.
  - Căutarea "**în adâncime**" poate fi simplu exprimată în manieră **recursivă** ea bazându-se pe o structură de date **stivă.**
  - Căutarea "**prin cuprindere**" admite o implementare simplă **iterativă** fiind bazată pe o structură de date **coadă**.
- Acesta este încă un exemplu care evidenţiază cu pregnanţă legătura strânsă care există între o anume **structură de date** şi **algoritmul** care o prelucrează.

10.5. Aplicaţii ale traversării grafurilor

- În cadrul paragrafului de faţă se prezintă câteva dintre aplicaţiile tehnicilor de traversare a grafurilor.
- Se au în vedere în acest context:
  - Arborii de acoperire.
  - Tehnici de determinare a arborilor de acoperire.
  - Unele aspecte legate de conexitate.
  - Punctele de articulaţie ale unui graf.
  - Componentele biconexe ale unui graf.

10.5.1. Arbori de acoperire ("Spanning Trees"). Determinarea arborilor de acoperire

- Una din aplicaţiile traversării grafurilor o reprezintă determinarea unui **arbore de acoperire** ("**spanning tree**") pentru un graf.
  - După cum s-a mai precizat, orice **graf conex aciclic** este un **arbore liber**.
- Un **arbore de acoperire** al unui graf `G`, este un **arbore liber** construit pornind de la anumite arce ale lui `G`, într-o astfel de manieră încât el să conţină toate nodurile lui `G`.
  - Un alt mod de a preciza **arborii de acoperire** este acela de a-i considera drept cea mai mică colecţie de arce aparţinînd unui graf, care permite comunicaţia între ori care două noduri ale grafului (&10.1.).
- În prezentarea tehnicilor fundamentale de traversare a grafurilor s-a făcut precizarea că fiecărei traversări a unui graf i se poate asocia un **arbore de acoperire**, respectiv o **pădure** ("**forest**") de astfel de arbori dacă graful conţine mai multe componente conexe. Există de fapt, câte un **arbore** pentru fiecare **componentă**.
  - Arborii de acoperire sunt denumiţi şi **arbori de căutare în adâncime** respectiv **arbori de căutare prin cuprindere** în dependenţă de metoda de traversare utilizată în determinarea lor.

- În continuare se vor prezenta unele **tehnici de determinare** a arborilor de acoperire pentru ambele tipuri de traversări.
- Înainte de prezentarea efectivă a acestor tehnici, se subliniază faptul că o altă posibilitate de evidenţiere a arborilor de acoperire o reprezintă construcţia **subgrafului de precedenţă** `sp` prezentat în variantele Cormen, Leiserson şi Rivest de traversare a grafurilor atât pentru **căutarea în adâncime** cât şi pentru **căutarea prin cuprindere**.
    - După cum s-a precizat subgraful `sp` este un **arbore liber** reprezentat prin tehnica "indicator spre părinte" care materializează de fapt **arborele de acoperire** specific.

10.5.1.1. Determinarea unui arbore de căutare "în adâncime"

- Aspectele teoretice ale acestei tehnici au fost precizate în &10.4.1.
- În continuare se prezintă **algoritmul** care determină un **arbore de căutare în adâncime** pornind de la traversarea grafurilor prin tehnica **căutării în adâncime**.
- **Tehnica** este simplă:
    - Ori de câte ori se vizitează un nod nevizitat încă, se marchează arcul care leagă ultimul nod curent de noul nod.
    - Toate arcele astfel marcate sunt **arce de arbore** ale arborelui de căutare în adâncime.
    - Restul arcelor, adică cele nemarcate, sunt **arce de retur** care se trasează punctat în reprezentarea arborelui.
- Procedura care implementează această tehnică pentru căutarea în adâncime, se numeşte **`ArboreDeAcoperireCA`** şi apare în secvenţa [10.5.1.1.a].
- Se fac următoarele precizări:
    1) Graful se consideră reprezentat prin **structuri de adiacenţe** implementate cu ajutorul structurii de date multilistă **varianta Decker**, prezentată în studiul de caz 3 din paragraful &10.3.2.3.
    2) Pentru marcarea nodurilor vizitate, structura unui nod implementat de articolul `TipCelulăListNod` din secvenţa [10.3.2.3.a], se suplimentează cu câmpul `marc` de tip `boolean`. La iniţializarea procedurii câmpul se pune pe "`false`" pentru toate nodurile grafului, urmând a deveni "`true`" în momentul vizitării.
    3) În vederea marcării arcelor **arborelui de căutare în adâncime**, structura unei celule aparţinând unei liste de adiacenţe, respectiv articolul `TipCelulăListArc` din aceeaşi secvenţă [10.3.2.3.a], se completează cu câmpul `marcArb` de tip `boolean`, a cărui valoare iniţial falsă devine adevărată dacă **arcul** este **de arbore**, respectiv rămâne falsă dacă **arcul** este **de retur**.
- Se face de asemenea precizarea că **arborele de căutare în adâncime** corespunzător unui graf **nu** este **unic**.
    - Astfel pentru un acelaşi graf pot fi obţinuţi diferiţi arbori de căutare în adâncime funcţie de:
        - (1) Modul şi ordinea în care se crează structura de adiacenţe aferentă.

- (2) Nodul cu care se începe parcurgerea.
- (3) Ordinea în care sunt parcurse nodurile adiacente.

```
-------------------------------------------------------------
{Determinarea unui arbore de acoperire minim pentru CA în
grafuri reprezentate prin SA implementate cu multiliste
(varianta Decker) – implementare PASCAL}

procedure ArboreDeAcoperireCA(g: TipGraf);

  var p: RefTipPozitie;
      e: RefListArc;

  procedure ConstructieACA(p: TipPozitie);

    var arcCurent: RefListArc;
        q: TipPozitie;
    begin
      p^.marc:= true; {marcarea cu vizitat a nodului curent}
      arcCurent:= p^.incep; {lista de adiacenţe}
      while arcCurent <> nil do
        begin
          q:= arcCurent^.nod;
          if not q^.marc then              [10.5.1.1.a]
            begin
              arcCurent^.marcArb:= true;
              {arcul arborelui CA poate fi afişat; de
               asemenea poate fi marcată şi cealaltă copie a
               lui arcCurent, deoarece în această
               implementare fiecare arc este reprezentat de
               două ori}
              ConstructieACA(q)
            end;
          arcCurent:= arcCurent^.urm
        end
    end; {ConstructieCA}

  begin {ArboreDeAcoperireCA}
    p:= g; {p – pointer pentru lista nodurilor}
    while p <> nil do {marcarea cu nevizitat a nodurilor}
      begin
        p^.marc:= false;
        e:= p^.incep; {e–pointer pentru lista de adiacente}
        while e <> nil do {marcarea cu nevizitat a arcelor}
          begin
            e^.marcArb:= false;
            e:= e^.urm
          end;
        p:= p^.urm
      end;
    p:= g;
    ConstructieACA(p)
  end; {ArboreDeAcoperireCA}
-------------------------------------------------------------
```

10.5.1.2. Determinarea unui arbore de căutare „prin cuprindere”

- Într-o manieră asemănătoare, pornind de la algoritmul traversării prin cuprindere a unui graf se poate concepe procedura **ArboreDeAcoperireCC** care determină un **arbore de căutare prin cuprindere** (secvenţa [10.5.1.2.a]).
- Toate precizările făcute anterior rămânând valabile şi pentru această procedură.
- În plus se face precizarea că pentru gestionarea **structurii de date coadă** au fost utilizaţi **operatori specifici** [Cr00].

```
{Determinarea unui arbore de acoperire minim pentru CC în
grafuri reprezentate prin SA implementate cu multiliste
(varianta Decker) - se utilizează TDA Coadă - implementare
PASCAL}

procedure ArboreDeAcoperireCC(g: TipGraf);

  var C: TipCoada; {de poziţii}
      n,m,e: RefTipPozitie;
      arcCurent: RefListArc;
  begin
    n:= g;  {n - pointer pentru lista nodurilor}
    while n <> nil do {marcarea cu „nevizitat” a nodurilor}
      begin
        n^.marc:= false;
        e:= n^.incep; {e-pointer pentru lista de adiacente}
        while e <> nil do {marcarea cu nevizitat a arcelor}
          begin
            e^.marcArb:= false;
            e:= e^.urm
        end;
        n:= n^.urm
      end;
    Initializeaza(C); {iniţializare coadă}
    n:= g;            {n este nodul de pornire}
    Adauga(n,C);      {amorsare parcurgere}
    n^.marc:= true;
    while not Vid(C) do
      begin
        n:= Cap(C);
        Scoate(C);
        arcCurent:= n^.incep;                {[10.5.1.2.a]}
        while arcCurent <> nil do
          begin
            m:= arcCurent^.nod;
            if not m^.marc then
              begin
                Adauga(m,C);
                arcCurent^.marcArc:= true
                {se poate afişa arcul de arbore;
                 se poate marca şi cealaltă copie a lui
                 arcCurent}
              end;
            arcCurent:= arcCurent^.urm
          end
      end
```

```
  end; {ArboreDeAcoperireCC}
-------------------------------------------------------
```

10.5.2. Grafuri şi conexiuni

- În cadrul grafurilor, noţiunea de **conexiune** joacă un rol central şi ea este strâns legată de noţiunea de **arc**, respectiv de noţiunea de **drum.**
- În paragraful §10.1. se definesc pornind de la aceste elemente noţiunile de **graf conex**, respectiv de **componentă conexă** a unui graf.
    - (1) Se reaminteşte că un **graf conex** este acela în care pentru fiecare nod al său există un drum spre oricare alt nod al grafului.
    - (2) Un graf care **nu** este **conex** este format din **componente conexe**.
- Deoarece în anumite situaţii prelucrarea grafurilor este simplificată dacă grafurile sunt partajate în componentele lor conexe, în continuare se vor prezenta unele **tehnici de determinare a componentelor conexe** ale unui graf.
- De asemenea sunt prezentate noţiunile de **graf biconex** şi **punct de articulaţie** precum şi tehnicile determinării acestora.

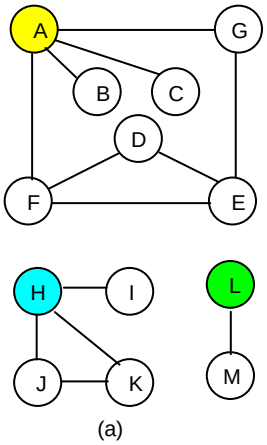
10.5.2.1. Determinarea componentelor conexe ale unui graf

- Oricare dintre **metodele de traversare** a grafurilor prezentate în paragraful anterior poate fi utilizată pentru determinarea componentelor conexe ale unui graf.
    - Acest lucru este posibil deoarece toate metodele de traversare se bazează pe aceeaşi **strategie generală** a vizitării tuturor nodurilor dintr-o componentă conexă înainte de a trece la o altă componentă.
- O manieră simplă de a vizualiza componentele conexe este aceea de a modifica **procedura recursivă de traversare prin căutare în adâncime** spre exemplu aşa cum se sugerează în procedura **ComponenteConexe** secvenţa [10.5.2.1.a].

```
-------------------------------------------------------------
/*Determinarea componentelor conexe ale unui graf
reprezentat prin SA implementate cu ajutorul listelor
înlănţuite simple - varianta pseudocod*/

int id;  /*contor noduri*/
int marc[maxN];  /*tablou evidenţă noduri*/

subprogram Componenta(int x);
/*parcurge componenta conexa căreia îi aparţine nodul x*/
  ref_tip_nod t;

  id= id+1; marc[x]= id; /*marchează nodul vizitat*/
  *scrie(t->nume);
  t= StrAdj[x]; /*t - pointer in lista de adiacenţe */
  cât timp (t<>null)/*parcurgere lista de adiacenţe alui x*/
   | daca (marc[t->nume] este 0)
   |   Componenta(t->nume);
   | t= t->urm;
   | □ /*cât timp*/
```

```
/*Componenta*/                                    /*10.5.2.1.a*/

subprogram ComponenteConexe;
/*apelează componentele conexe ale grafului*/
  int x;

  id= 0;
  pentru (x= 1 la N)
    marc[x]= 0;
  pentru (x= 1 la N)
    dacă marc[x]=0
      Componenta(x);
      *scrie_rand (writeln); /*afişează componenta curentă*/
      □ /*dacă*/
/*ComponenteConexe*/
```

- Alte variante ale procedurii de căutare în adâncime cum ar fi cea aplicată grafurilor reprezentate prin matrice de adiacenţe sau cea nerecursivă, precum şi căutarea prin cuprindere, modificate în aceeaşi manieră, vor evidenţia aceleaşi componente conexe, dar nodurile vor fi vizualizate în altă ordine.
- Funcţie de natura prelucrărilor ulterioare ale grafului pot fi utilizate şi alte metode.
- Astfel spre **exemplu**, se poate introduce tabloul `invmarc` ("inversul" tabloului `marc`) care se completează ori de câte ori se completează tabloul `marc`, respectiv când `marc[x]= id` se asignează şi `invmarc[id]= x.`
  - În tabloul `invmarc` intrarea `id` conţine indexul celui de-al `id`-lea nod vizitat. În consecinţă, nodurile aparţinând aceleeaşi componente conexe sunt **contigue** adică ocupă poziţii alăturate.
  - În acest tablou, indexul care precizează **o nouă componentă conexă**, este dat de valoarea lui `id` din momentul în care procedura **`Componenta`** este apelată din procedura **`ComponenteConexe`**.
  - Aceste valori pot fi memorate separat sau pot fi marcate în tabloul `invmarc`, spre exemplu primind valori **negative** (opuse).
- În fig.10.5.2.1.a (c) se prezintă valorile pe care le conţin aceste tablouri, în urma execuţiei procedurii **`ComponenteConexe`** asupra grafului (a), reprezentat prin structura de adiacenţe din aceeaşi figură (b).



(a)

A → F→ C→ B → G
B → A
C → A
D → F → E
E → G → F → D
F → A → E → D
G → E → A
H → I → J → K
I → H
J → H → K
K → H → J
L → M
M → L

(b)

x	1	2	3	4	5	6	7	8	9	10	11	12	13
nume[x]	A	B	C	D	E	F	G	H	I	J	K	L	M
marc[x]	1	7	6	5	3	2	4	8	9	10	11	12	13
invmarc[x]	-1	6	5	7	4	3	2	-8	9	10	11	-12	13
ordine	A	F	E	G	D	C	B	H	I	J	K	L	M

(c)

Fig. 10.5.2.1.a. Evidenţierea componentelor conexe ale unui graf

- În activitatea practică, este deosebit de avantajoasă utilizarea unor astfel de tehnici de partajare a grafurilor în componente conexe în vederea prelucrării ulterioare a acestor componente în cadrul unor algoritmi complecşi.
    - Astfel, algoritmii de complexitate mai ridicată sunt eliberaţi de detaliile prelucrării unor grafuri care nu sunt conexe şi în consecinţă devin mai simpli.

10.5.2.2. Puncte de articulaţie şi componente biconexe

- În anumite situaţii este util a se prevedea **mai mult decât un singur drum** între nodurile unui graf cu scopul de a rezolva posibile căderi ale unor puncte de contact (noduri).
    - Astfel, spre exemplu în reţeaua feroviară există mai multe posibilităţi de a ajunge într-un anumit loc.
    - De asemenea într-un circuit integrat liniile principale de comunicaţie sunt adesea dublate astfel încât circuitul rămâne încă în funcţiune dacă vreo componentă cade.
- Un **punct de articulaţie** al unui **graf conex** este un **nod**, care dacă este suprimat, **graful se rupe** în două sau mai multe bucăţi.
- Un graf care **nu** conţine **puncte de articulaţie** se numeşte **graf biconex**.
    - Într-un **graf biconex**, fiecare pereche de noduri este conectată prin cel puţin **două** drumuri distincte.
    - Un graf care **nu** este **biconex** se divide în **componente biconexe**, acestea fiind mulţimi de noduri mutual accesibile via două drumuri distincte.

- În fig. 10.5.2.2.a apare un graf conex care însă nu este biconex.
- **Punctele de articulaţie** ale acestui graf sunt:
    - A care leagă pe B de restul grafului.
    - H care leagă pe I de restul grafului.
    - L care leagă pe M de restul grafului.
    - G prin a cărui suprimare graful se divide în trei părţi.
- În concluzie în cadrul grafului din figură există şase **componente biconexe**:
    - (1) Grupul de noduri{A,C,G,D,E,F}
    - (2) Grupul de noduri {G,J,H,K}
    - (3) Nodul individual B
    - (4) Nodul individual I
    - (5) Nodul individual L
    - (6) Nodul individual M.

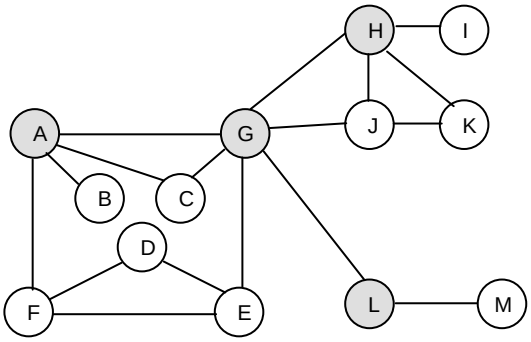


Fig.10.5.2.2.a. Graf care nu este biconex

- În acest context una din problemele care se ridică este aceea a determinării **punctelor de articulaţie** ale unui graf.
    - Se face precizarea că aceasta este una din mulţimea de probleme deosebit de importante referitoare la conexitatea grafurilor.
- Astfel, ca şi un **exemplu de aplicaţie al conexităţii grafurilor** poate fi prezentată o **reţea** care este de fapt un graf în care nodurile comunică unele cu altele.
    - În legătură cu această reţea se ridică următoarea întrebare fundamentală: "Care este **capacitatea de supravieţuire** a reţelei atunci când unele dintre nodurile sale cad?"
    - Cu alte cuvinte în ce condiţii, în urma căderii unor noduri reţeaua rămâne încă funcţională?
- Un **graf** are **conexitatea k** sau altfel spus are **numărul de conexitate egal cu k** dacă prin suprimarea a oricare **k-1** noduri ale sale graful rămâne conex [FK69].

- Spre exemplu, **un graf are conexitatea doi**, dacă şi numai dacă **nu are puncte de articulaţie**, cu alte cuvinte, dacă şi numai dacă este **biconex**.

- Cu cât numărul de conexitate al unui graf este mai mare, cu atât capacitatea de supravieţuire a grafului la căderea unora din nodurile sale este mai mare.

10.5.2.3. Determinarea punctelor de articulaţie ale unui graf

- În vederea determinării **punctelor de articulaţie** ale unui **graf conex**, poate fi utilizată, printr-o extensie simplă, **traversarea grafurilor prin tehnica căutării în adâncime**.

  - **Absenţa punctelor de articulaţie** precizează un **graf biconex**.

- Se consideră spre **exemplu**, graful din figura 10.5.2.2.a şi un arbore de căutare în adâncime asociat, ca şi cel din fig.10.5.2.3.a.

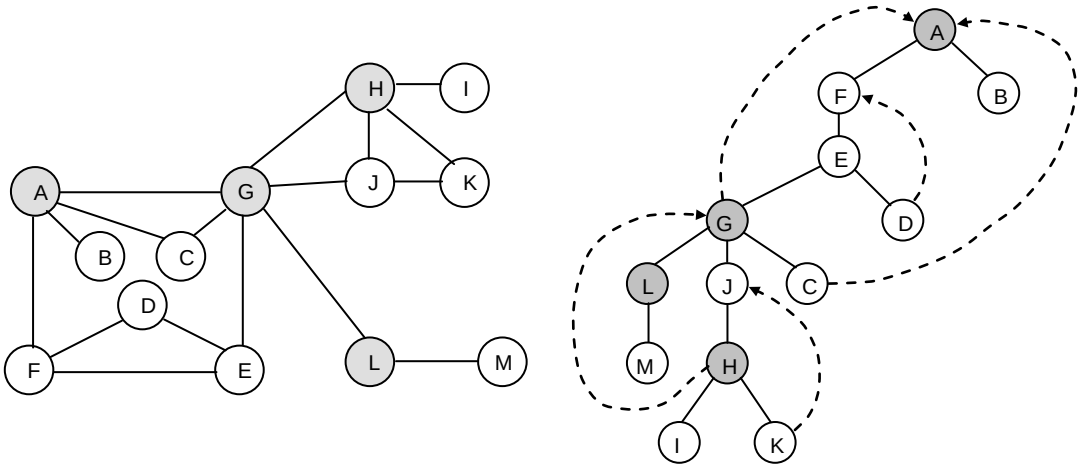


Fig.10.5.2.3.a. Arbore de căutare în adâncime pentru determinarea punctelor de articulaţie ale unui graf conex

  - Se observă că suprimarea nodului `E` **nu** conduce la dezmembrarea grafului deoarece ambii fii ai acestuia, `G` respectiv `D` sunt conectaţi prin **arce de retur** (linii punctate) cu noduri situate deasupra în arbore.

  - Pe de altă parte suprimarea lui `G` conduce la scindarea grafului deoarece nu există astfel de alternative pentru nodurile `L` sau `J`.

- Un **nod** oarecare `x` al unui graf **nu este un punct de articulaţie** dacă **fiecare fiu** `y` al său are printre descendenţi vreun nod conectat (printr-o linie punctată) cu un nod situat în arbore deasupra lui `x`, cu alte cuvinte, dacă există o conexiune alternativă de la `x` la `y`.

  - Această verificare **nu** este valabilă pentru **rădăcina** arborelui de căutare în adâncime deoarece nu există noduri situate "deasupra" acesteia.

  - În consecinţă**, rădăcina** este un **punct de articulaţie** dacă are doi sau mai mulţi fii deoarece singurul drum care conectează fiii rădăcinii trece prin rădăcina însăşi.

- Determinarea **punctelor de articulaţie** poate fi implementată pornind de la **căutarea în adâncime**, prin transformarea procedurii de căutare într-o **funcţie** care returnează pentru nodul furnizat ca parametru, **cel mai înalt punct** din cadrul arborelui de

căutare întâlnit în timpul căutării, adică nodul cu cea mai mică valoare memorată în tabloul `marc`.

- Algoritmul implementat în forma funcţiei **`Articulaţie`** apare în secvenţa [10.5.2.3.a].

```
/*Determinarea punctelor de articulaţie ale unui graf
reprezentat prin SA - varianta pseudocod*/

int function articulatie(int x)
/*returnează cel mai înalt nod din arborele de căutare care
poate fi întâlnit pornind de la x*/
  Ref_Tip_Nod t;
  int m,min;

  id= id+1; marc[x]= id; min= id; /*marcare nod curent*/
  t= StrAdj[x]; /*t - pointer in lista de adiacenţe*/
  cât timp (t<>null) /*prelucrare structură de adiacenţe*/
   |dacă (marc[t->nume]==0)                  /*10.5.2.3.a*/
   |   |m= articulatie(t->nume);
   |   |dacă (m<min)
   |   |  min= m;
   |   |dacă (m>=marc[x])
   |   |  *scrie(x); /*x este punct de articulaţie*/
   |   |□ /*dacă*/
   |  altfel
   |     dacă (marc[t->nume]<min)
   |       min= marc[t->nume];
   |t= t->urm;
   |□ /*cât timp*/
  returneaza min;
/*Articulatie*/
```

- Referitor la această funcţie se fac următoarele precizări:
  - (1) La căutarea în adâncime într-un graf, valoarea lui `marc[x]` pentru orice nod `x` al grafului precizează **numărul de ordine** al nodului `x` în cadrul traversării.
    - Aceeaşi ordine rezultă şi din traversarea în preordine a nodurilor arborelui de căutare în adâncime.
    - Cu cât nodul este traversat mai devreme, cu atât numărul său de ordine în `marc` este mai mic şi cu atât poziţia sa în cadrul arborelui de căutare este mai înaltă.
    - **Concluzia**: descendenţii unui nod vor avea în tabloul `marc` numere mai mari decât nodul părinte şi vor fi situaţi sub acesta în arborele de căutare.
  - (2) Pentru fiecare nod `x`, valoarea `min` returnată de funcţia **`Articulaţie`** este **cel mai mic număr de ordine** întâlnit în cadrul traversării.
    - Acest număr poate corespunde chiar lui `x` sau oricărui nod `z` la care se poate ajunge pornind de la `x`, coborând zero sau mai multe arce ale

arborelui până la un descendent `w` (`w` poate fi chiar `x`), iar apoi urmând un arc de retur `(w,z)`.

- Valoarea `min` se calculează pentru toate nodurile `x`, traversând arborele în **postordine**.
  - Astfel, când se procesează nodul `x` valoarea `min` a fost deja calculată pentru toţi fiii `y` ai lui `x`.

- (3) Pentru un nod oarecare `x`, valoarea `min` asociată este **cea mai mică valoare** dintre:
  - Numărul de ordine al lui `x`.
  - Numărul de ordine al oricărui nod `z` pentru care există în arborele de căutare în adâncime un arc de retur `(x,z)`.
  - Valoarea lui `min` pentru toţi fiii `y` ai lui `x`.

- (4) **Punctele de articulaţie** se găsesc astfel:
  - **Rădăcina este punct de articulaţie** dacă şi numai dacă are doi sau mai mulţi fii.
  - **Un nod** `x`, altul decât rădăcina este un **punct de articulaţie** dacă şi numai dacă există vreun fiu `y` al lui `x` astfel încât valoarea `min` pentru `y` este **mai mare sau egală** cu numărul de ordine al lui `x.`
    - În acest caz, `x` deconectează pe `y` şi descendenţii acestuia de restul grafului.

- (5) Dacă valoarea `min` pentru **toţi fiii** `y` ai lui `x` este **mai mică** decât numărul de ordine al lui `x`, atunci există cu siguranţă un drum pe care se poate ajunge de la oricare din fiii `y` ai lui `x` înapoi la un strămoş propriu `z` a lui `x` (nodul care are numărul de ordine egal cu `min` pentru `y`).
  - În consecinţă suprimarea nodului `x` **nu** conduce la deconectarea nici unui fiu `y` al lui `x` sau a descendenţilor săi de restul grafului [AH85].

- Funcţia **`Articulaţie(x)`** determină de fapt în manieră recursivă **cel mai înalt punct al arborelui** care poate fi atins via un arc de retur pentru **orice descendent** al nodului `x` furnizat ca parametru şi utilizează această informaţie pentru a stabili dacă `x` este un punct de articulaţie.
  - După cum s-a precizat, aceasta presupune o simplă comparaţie între valoarea `m,` adică minimul determinat pentru nodul descendent curent şi `marc[x]` adică numărul de ordine al lui `x`.
  - În plus, mai este necesar a se verifica dacă `x` nu este cumva **rădăcina** arborelui de căutare în adâncime, cu alte cuvinte, dacă `x` nu este cumva primul nod parcurs în apelul funcţiei **`Articulaţie`** pentru componenta conexă a grafului care conţine nodul `x`.
    - Această verificare se face în afara funcţiei recursive, motiv pentru care ea nu este prezentă în secvenţa [10.5.2.3.a].

- Această secvenţă care de fapt afişează punctele de articulaţie, poate fi uşor extinsă pentru a realiza şi alte prelucrări asupra **punctelor de articulaţie** sau a **componentelor biconexe**.

- Deoarece procedeul derivă din **traversarea grafurilor prin tehnica căutării în**

adâncime, efortul său de execuţie este proporţional cu $O(n+a)$ în reprezentarea grafurilor prin structuri de adiacenţe, respectiv cu $O(n^2)$ în reprezentarea bazată pe matrice de adiacenţe, n reprezentând numărul de noduri, iar a numărul de arce al grafului.