PROGRAMARE ȘI STRUCTURI DE DATE

CURS 13

Lect. dr. Oneț-Marian Zsuzsanna

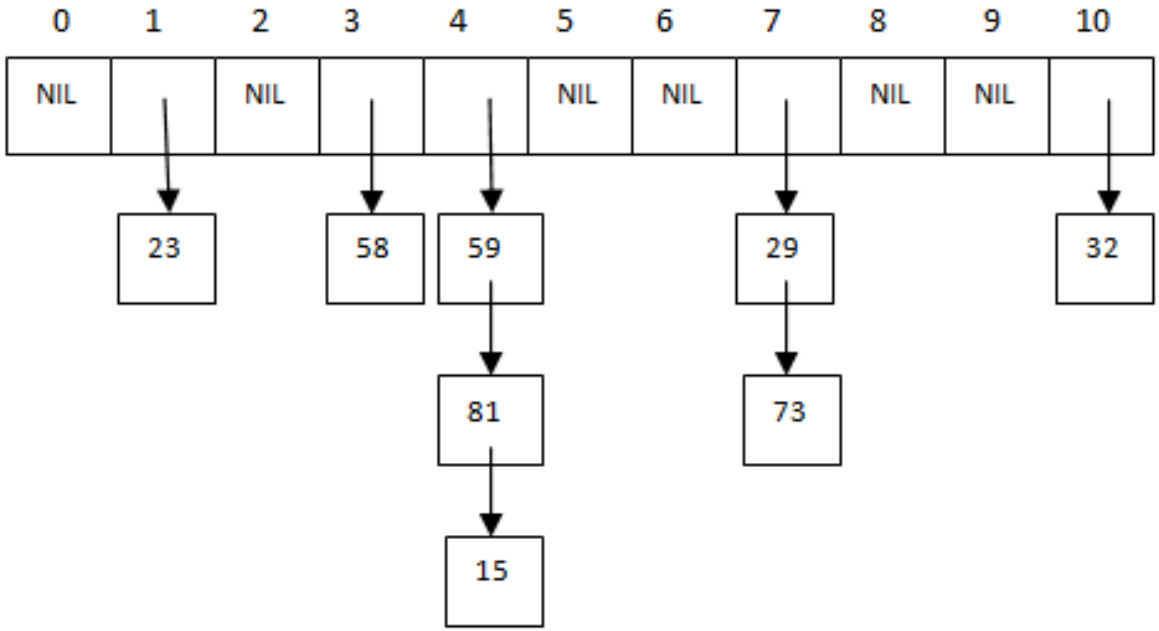
Facultatea de Matematică și Informatică UBB
în colaborare cu NTT Data

În cursul 11 și 12

- Lista simplu înlănțuită
- Lista dublu înlănțuită
- Stiva și Coada
- Tabela de dispersie
  - Rezolvare coliziuni cu liste independente
  - Rezolvare coliziuni cu adresare deschisă

Cuprins

TD - Rezolvare coliziuni cu Liste Independente



TD - Adresare deschisă

- O altă metodă de a rezolva coliziuni este prin *adresare deschisă*.
- La adresare deschisă fiecare element se pune pe o poziție în tabela de dispersie (nu în noduri care sunt în afara tabelei).
- Ideea de bază la adresare deschisă este următoarea:
  - Calculăm poziția elementului.
  - Dacă poziția este ocupată, calculăm o altă poziție, și așa mai departe până găsim o poziție liberă.
  - Dacă am verificat toate pozițiile și nu am găsit poziție liberă, tabela este plină și trebuie să o mărim.

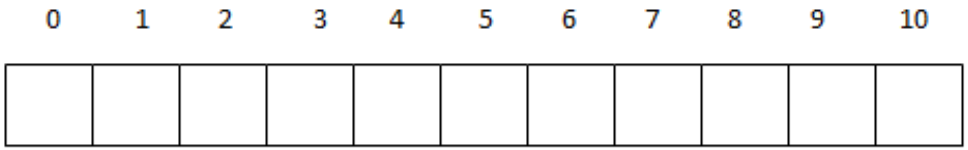
TD - adresare deschisă

- Pentru a putea face verificări consecutive, trebuie să modificăm un pic funcția de dispersie.
  - Până acum am folosit funcția de dispersie: $d(c) = (c \ mod \ m)$, dar această funcție ne dă aceeași valoare, indiferent de câte ori o apelăm
- Introducem în funcția de dispersie, ca parametru, numărul de încercare, notat cu i
  - Vom avea funcția de dispersie: $d(c, i)$.
  - Prima dată apelăm $d(c, 0)$. Dacă poziția este ocupată, calculăm $d(c, 1)$. Dacă poziția este ocupată apelăm $d(c, 2)$
  - În momentul în care găsim o poziție liberă, punem elementul acolo. Dacă i ajunge la m am încercat toate pozițiile, și nu este poziție liberă.
  - Indiferent unde am găsit loc liber (la ce valoare a lui i), la adăugarea/ștergerea/căutarea următoare începem cu $i = 0$ din nou.

TD - adresare deschisă

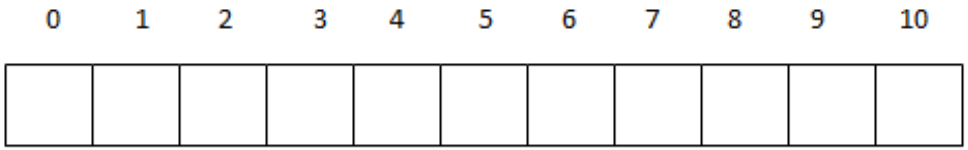
- Există mai multe variante de a defini funcția de dispersie $d(c, i)$. Noi vom folosi o variantă numită *verificare pătratică*: $d(c, i) = ((d'(c) + c_1 * i + c_2 * i^2) \ mod \ m)$
  - $d'(c)$ este o funcție de dispersie uzuală, de ex: $d'(c) = (c \ mod \ m)$
  - c_1 și c_2 sunt 2 constante
  - i este numărul de încercare.

TD - AD - Exemplu



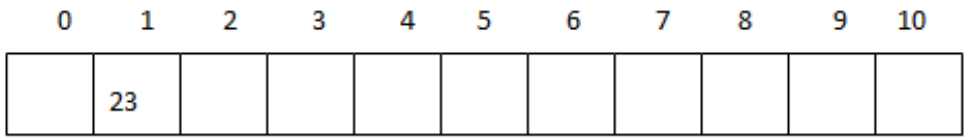
- Vom folosi funcția de dispersie:
  $d(c, i) = ((c \ mod \ m) + i + 2 * i^2) \ mod \ m)$
- Să adăugăm numărul 23.

TD - AD - Exemplu



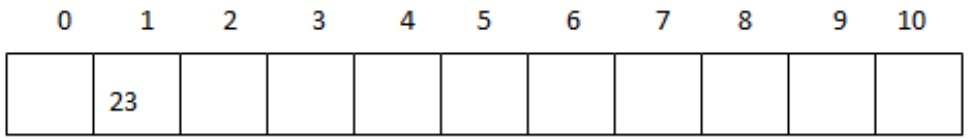
- Vom folosi funcția de dispersie:
  $d(c, i) = ((c \ mod \ m) + i + 2 * i^2) \ mod \ m)$
- Să adăugăm numărul 23.
- d(23, 0) = ((23 mod 11) + 0 + 0) mod m = 1
- Poziția 1 este liberă, punem elementul acolo.

TD - AD - Exemplu



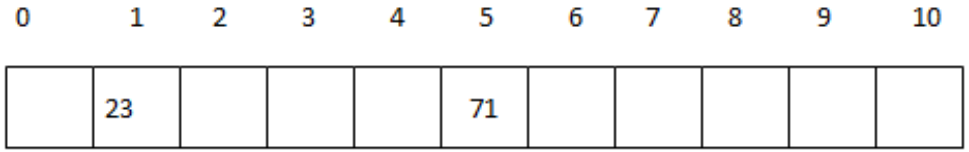
- Să adăugăm numărul 71.

TD - AD - Exemplu



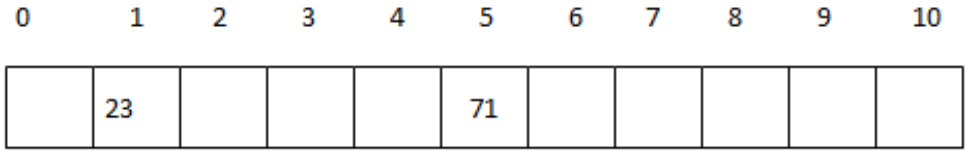
- Să adăugăm numărul 71.
- $d(71, 0) = ((71 \ mod \ 11) + 0 + 0) \ mod \ m = 5$

TD - AD - Exemplu



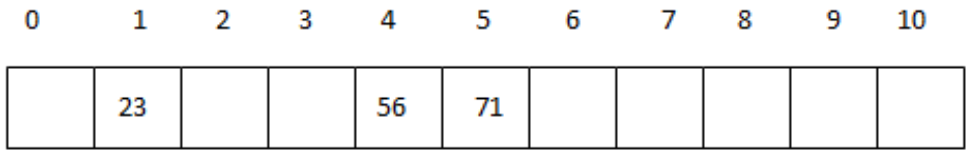
- Să adăugăm numărul 56.

TD - AD - Exemplu



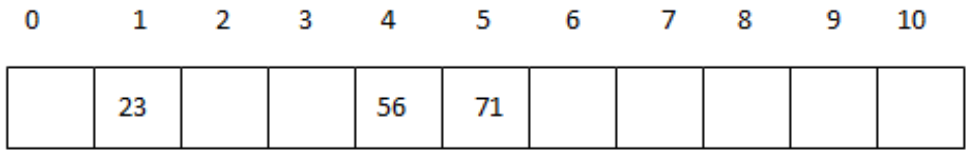
- Să adăugăm numărul 56.
- $d(56, 0) = ((56 \ mod \ 11) + 0 + 0) \ mod \ m = 1$
- Poziția 1 este ocupată, calculăm poziția următoare:
- $d(56, 1) = ((56 \ mod \ 11) + 1 + 2 * 1) \ mod \ m = 4$
- Poziția 4 este liberă.

TD - AD - Exemplu



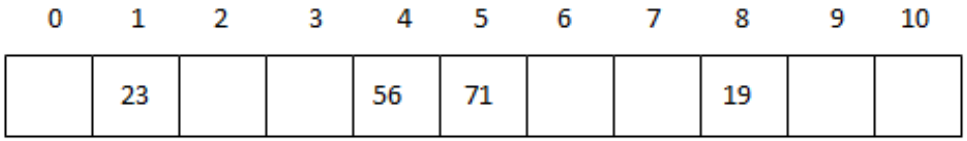
- Să adăugăm numărul 19.

TD - AD - Exemplu



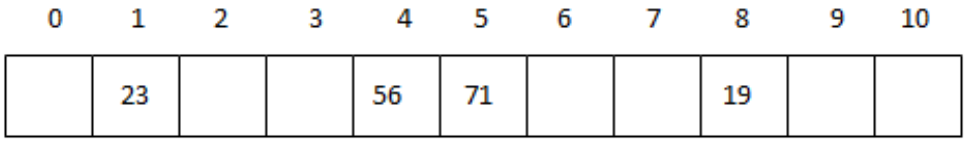
- Să adăugăm numărul 19.
- $d(19, 0) = ((19 \ mod \ 11) + 0 + 0) \ mod \ m = 8$

TD - AD - Exemplu



- Să adăugăm numărul 7.

TD - AD - Exemplu



- Să adăugăm numărul 7.
- $d(7, 0) = ((7 \ mod \ 11) + 0 + 0) \ mod \ m = 7$

TD - AD - Exemplu

0	1	2	3	4	5	6	7	8	9	10
	23			56	71		7	19		

- Să adăugăm numărul 48.

TD - AD - Exemplu

0	1	2	3	4	5	6	7	8	9	10
	23			56	71		7	19		

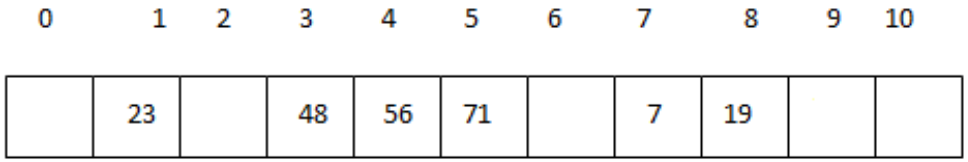
- Să adăugăm numărul 48.
- $d(48, 0) = ((48 \ mod \ 11) + 0 + 0) \ mod \ m = 4$
- Poziția 4 este ocupată, calculăm poziția următoare:
- $d(48, 1) = ((48 \ mod \ 11) + 1 + 2 * 1) \ mod \ m = 7$
- Poziția 7 este ocupată, calculăm poziția următoare:
- $d(48, 2) = ((48 \ mod \ 11) + 2 + 2 * 4) \ mod \ m = 3$
- Poziția 3 este liberă.

TD - AD - Exemplu

0	1	2	3	4	5	6	7	8	9	10
	23		48	56	71		7	19		

- Să căutăm elementul 45.

TD - AD - Exemplu

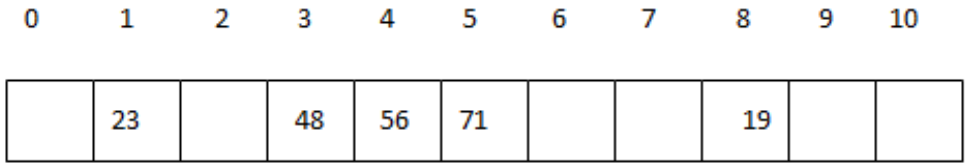


- Să căutăm elementul 45.
- Calculăm pe rând pozițiile unde 45 ar trebui să fie până găsim numărul 45 sau găsim o poziție liberă.
- $d(45, 0) = ((45 \ mod \ 11) + 0 + 0) \ mod \ m = 1$ - nu e acolo
- $d(45, 1) = ((45 \ mod \ 11) + 1 + 2 * 1) \ mod \ m = 4$ - nu e acolo
- $d(45, 2) = ((45 \ mod \ 11) + 2 + 2 * 4) \ mod \ m = 0$ - e poziție liberă
- 45 nu se găsește în tabela de dispersie.

TD - AD - Ștergere

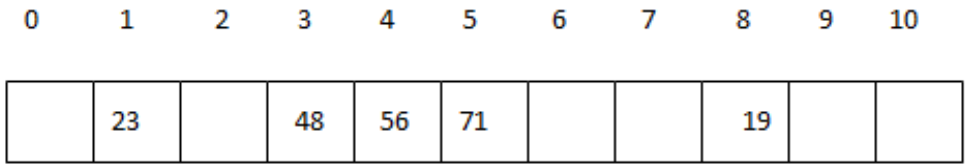
- A șterge un element din TD cu adresare deschisă este o operație delicată, pentru că nu putem doar să ștergem elementul.
- De ce?
- Să presupunem că vrem să ștergem numărul 7.
- Calculăm poziția pentru numărul 7:
- $d(7, 0) = ((7 \ mod \ 11) + 0 + 0) \ mod \ m = 7$
- Pe poziția 7 am găsit elementul. Să presupunem că doar îl ștergem din tabelă.

TD - AD- Exemplu



- Să căutăm elementul 48.

TD - AD- Exemplu



- Să căutăm elementul 48.
- Calculăm pe rând pozițiile unde 48 ar trebui să fie până găsim numărul 48 sau găsim o poziție liberă.
- $d(48, 0) = ((48 \ mod \ 11) + 0 + 0) \ mod \ m = 4$ - nu e acolo
- $d(48, 1) = ((48 \ mod \ 11) + 1 + 2 * 1) \ mod \ m = 7$ - e poziție liberă
- 48 nu se găsește în tabela de dispersie.

TD - AD - Exemplu

- Deși acum pare a fi un pas logic să mutăm 48 pe locul 7 când ștergem elementul 7, adevărul este că în momentul în care noi ștergem elementul 7 nu știm ce elemente vin *după el*. Poate poziția 7 a participat în mai multe secvențe de verificare.
- De aceea la ștergere în general pur și simplu se înlocuiește elementul șters cu o valoara specială, care să arate că elementul de pe locul respectiv a fost șters.

TD - AD - Exemplu

- Deși acum pare a fi un pas logic să mutăm 48 pe locul 7 când ștergem elementul 7, adevărul este că în momentul în care noi ștergem elementul 7 nu știm ce elemente vin *după el*. Poate poziția 7 a participat în mai multe secvențe de verificare.
- De aceea la ștergere în general pur și simplu se înlocuiește elementul șters cu o valoara specială, care să arate că elementul de pe locul respectiv a fost șters.

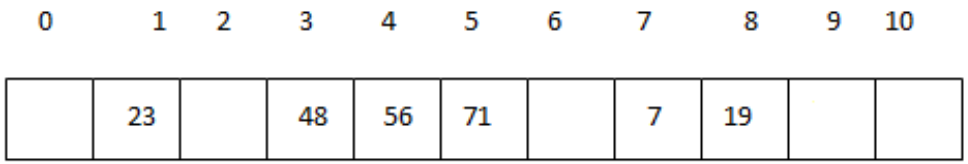
0	1	2	3	4	5	6	7	8	9	10
	23		48	56	71		Sters	19		

TD - AD - Valoarea *STERS*

- Cum lucrăm cu această valoare *STERS*?
  - La adăugare, pozițiile cu valoarea *STERS* sunt tratate ca și cum ar fi libere, putem pune element nou pe poziția respectivă.
  - La căutare, pozițiile cu valoarea *STERS* sunt tratate ca și cum ar fi ocupate, nu oprim căutarea dacă găsim valoare *STERS*.
  - La ștergere, la partea la care căutăm elementul de șters, pozițiile cu valoarea *STERS* sunt tratate ca și cum ar fi ocupate (continuăm căutarea).

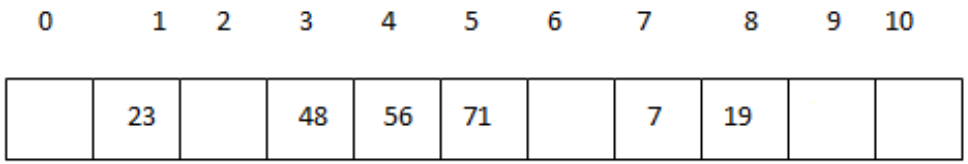
TD - AD - Iterator

- Având TD din figură, în ce ordine credeți că un iterator ar trebui să afișeze elementele?



TD - AD - Iterator

- Având TD din figură, în ce ordine credeți că un iterator ar trebui să afișeze elementele?



- O ordine posibilă este să afișăm elementele în ordinea: 23, 48, 56, 71, 7,19

TD - AD - Iterator

- Iteratorul pentru o TD cu adresare deschisă seamănă foarte mult cu iterator pentru un vector dinamic, cu excepția că trebuie să ne asigurăm că tot timpul suntem pe o poziție unde este un element.

TD - AD - Reprezentare

- Cum reprezentăm o tableă de dispersie cu adresare deschisă?

TD - AD - Reprezentare

- Cum reprezentăm o tableă de dispersie cu adresare deschisă?

TD:
elem: Întreg[]
m: Întreg *//capacitatea vectorului*
d: TFunctie *//funcția de dispersie*

TD - AD - Reprezentare

- Cum reprezentăm o tableă de dispersie cu adresare deschisă?

TD:
elem: Întreg[]
m: Întreg //*capacitatea vectorului*
d: TFunctie //*funcția de dispersie*

- Reprezentarea seamană mult cu reprezentarea de la Vector Dinamic, dar nu avem nevoie de câmpul *len*. Totuși, dacă vrem să știm câte elemente sunt în tabla de dispersie, putem adăuga acest câmp.

TD - AD - adaugare

- Ce ar trebui să facă operația de adăugare?

TD - AD - adaugare

- Ce ar trebui să facă operația de adăugare?

subalgoritm adauga(e: Întreg) **este:**
    i, poz: Întreg
    i = 0
    poz = this.d(e, i)
    câttimp $i < this.m - 1$ **ȘI** @this.elem[poz] nu e liber sau STERS
execută
        i = i + 1
        poz = this.d(e, i)
    sf_câttimp
    dacă @this.elem[poz] este ocupat **atunci**
        @trebuie alocat o tabelă mai mare și adăgat acolo
    altfel
        this.elem[poz] = e
    sf_dacă
sf_subalgoritm

TD - AD - adaugare

- Cât este complexitatea pentru *adaugare*?

TD - AD - adaugare

- Cât este complexitatea pentru *adaugare*?
  - Caz defavorabil: $O(n)$
  - Caz mediu: $\Theta(1)$

TD - Elemente nenumerice

- Ce se întâmplă dacă vrem să punem într-o TD elemente care nu sunt numere?
- Dacă vrem să stocăm stringuri, sau elemente de tip Film, Carte, etc.
  - Dacă elementele nu sunt numere, trebuie cumva să le transformăm într-un număr.
  - În general funcția care face această transformare, se numește *hashCode*.
  - Funcția hashCode primește un element și returnează un număr pentru acel element.
  - Dacă lucrăm cu funcția hashCode, funcția de dispersie devine:
    - $d(c) = hashCode(c)\ mod\ m$
    - $d(c, i) = ((hashCode(c)\ mod\ m) + c_1 * i + c_2 * i^2)\ mod\ m$

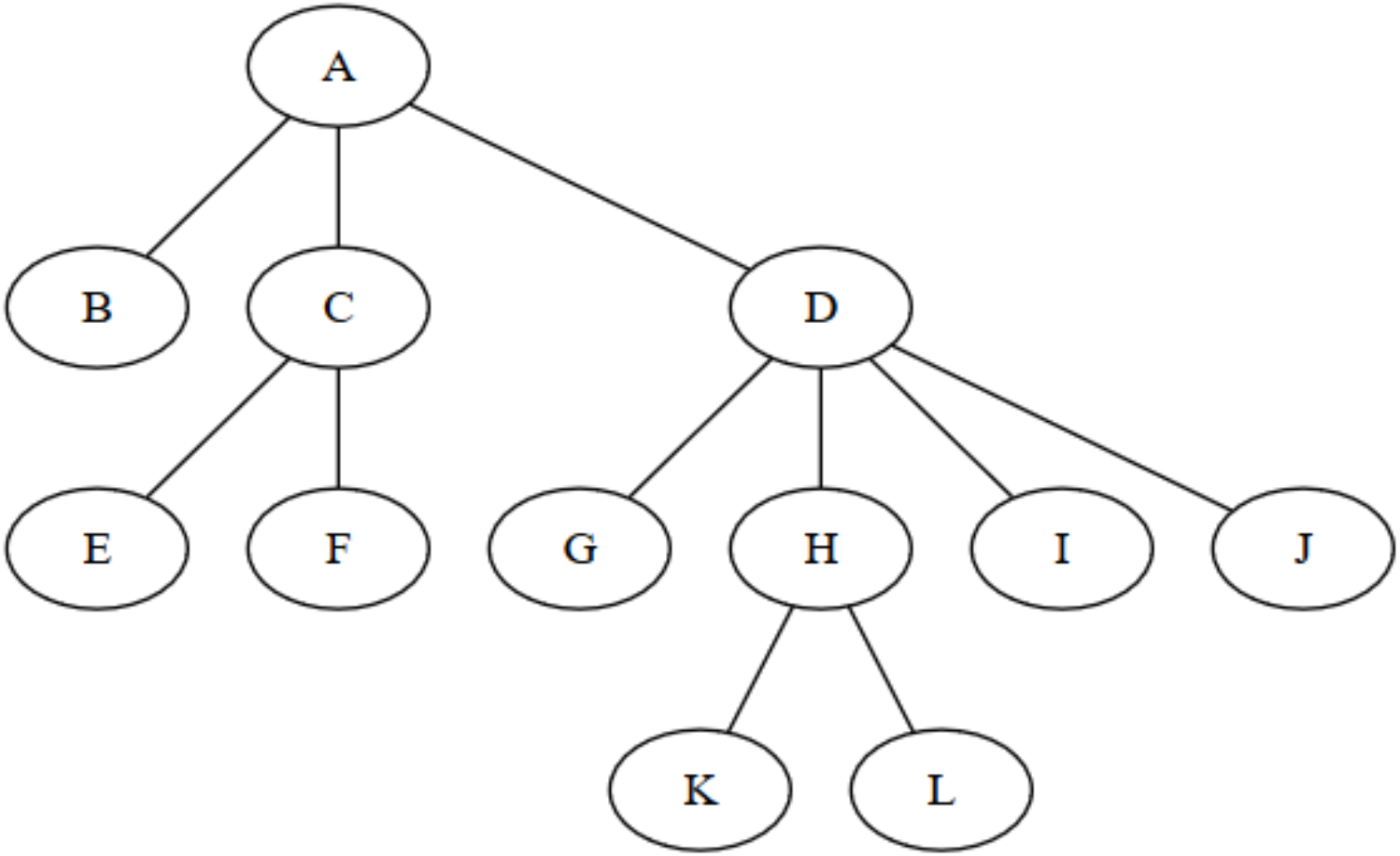
TD - hashCode

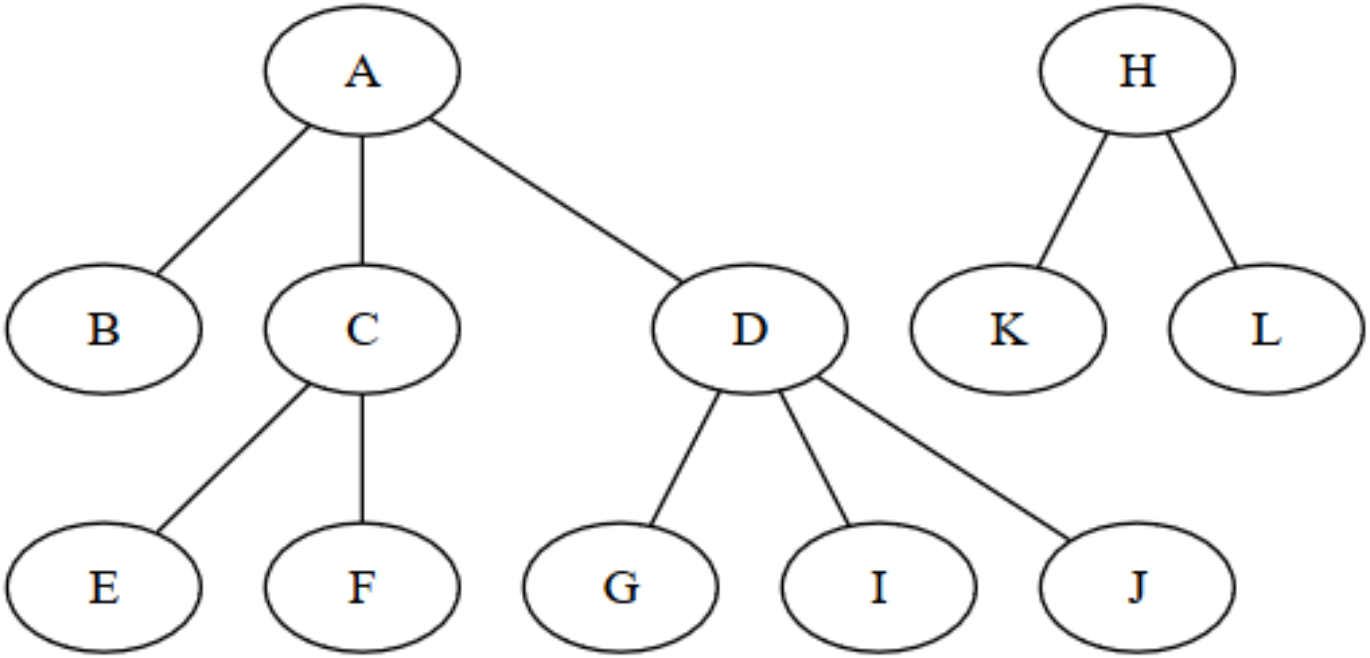
- Cum se poate defini funcția hashCode?
  - Dacă elementele sunt Stringuri, putem considera:
    - Lungimea stringului
    - Suma codurilor ASCII pentru caracterele
    - Suma codurilor ASCII pentru caracterele înmulțite cu poziția pe care este caracterul
  - Dacă elementele sunt obiecte de tip Film, Carte, Medicament, etc.
    - Dacă obiectele au ceva identificator unic, putem considera acel identificator.
    - Calculăm hashCode pe baza câmpurilor din obiecte
- O funcție hashCode este bună dacă rareori returnează aceeași valoare pentru 2 obiecte diferite.

Arbori

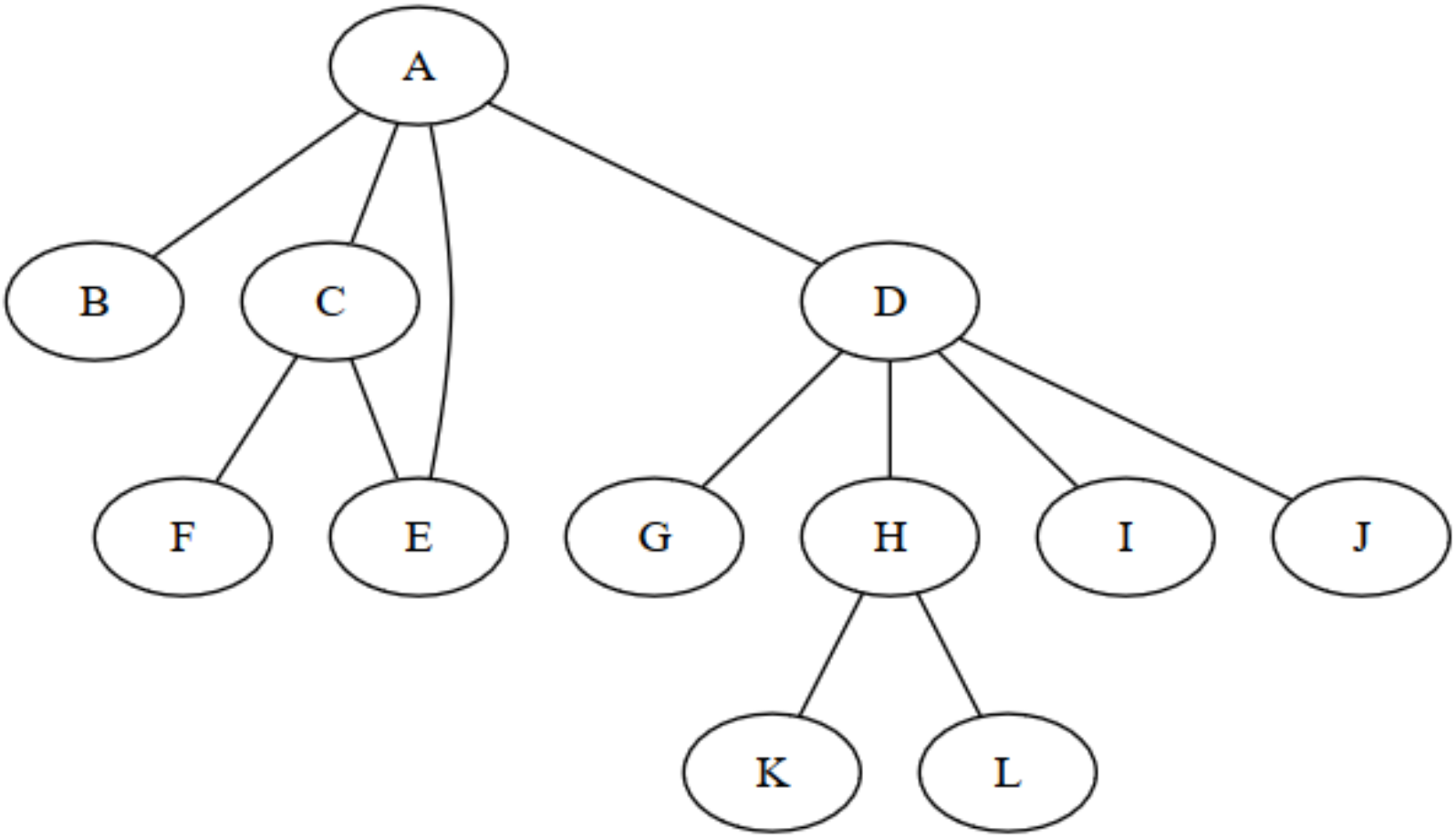
- Un arbore este o structura de date alcătuită din noduri, fiecare nod poate să aibă 0 sau mai mulți descendenți (copii) și fiecare nod are un singur părinte.
- Într-un arbore, toate nodurile trebuie să fie conectate (trebuie să existe un drum de a ajunge de la fiecare nod, la fiecare alt nod).
- Nu pot exista cicluri într-un arbore: pornind de la un nod și parcurgând alte noduri, nu putem ajunge înapoi la nodul de la care am plecat.
- Într-un arbore cu *n* noduri există *n-1* legături.

- Exemplu de arbore:





- Nu este arbore (nu este conectată), nu pot ajunge de la nodul A la L (de exemplu).

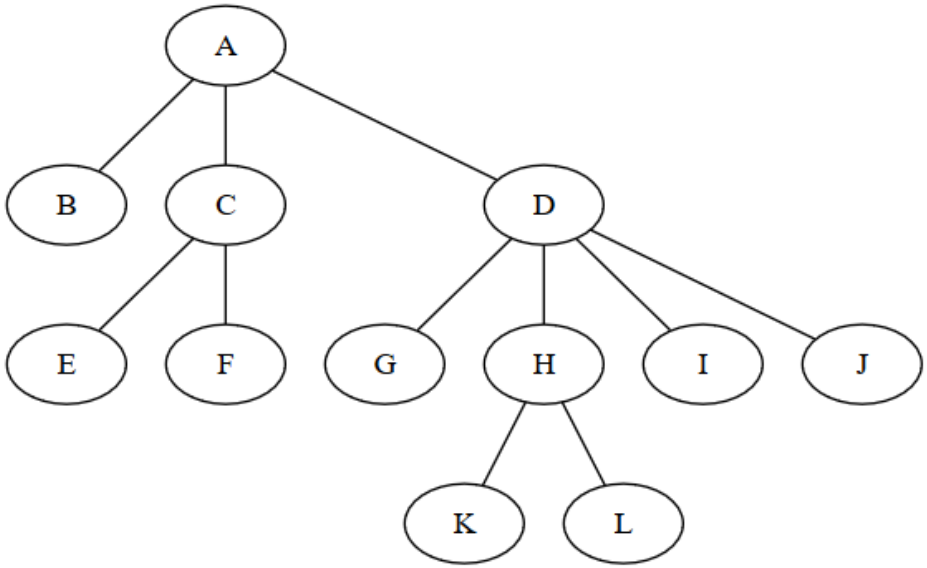


- Nu este arbore (există un ciclu A - C - E - A)

Terminologie

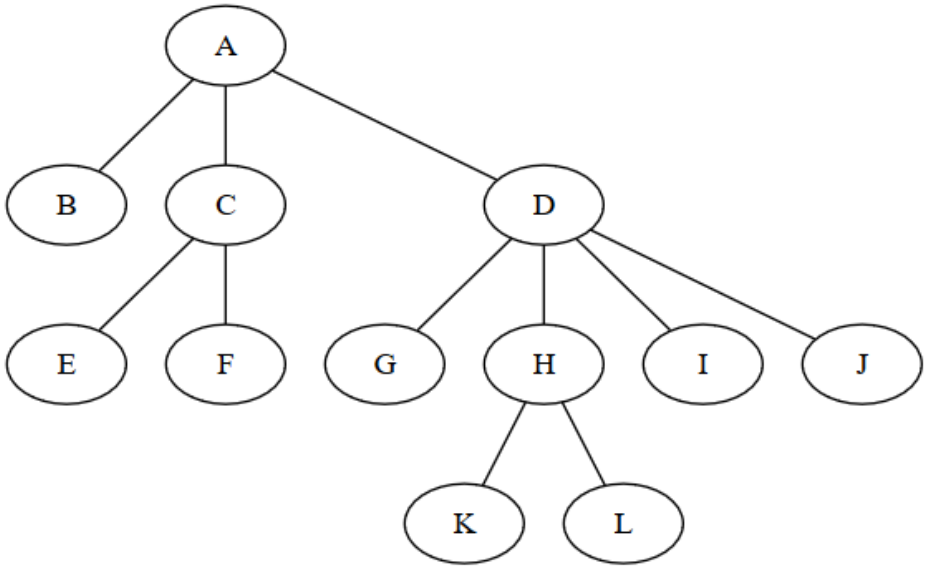
- Nodul care nu are părinte se numește *rădăcină*. Într-un arbore există o singură rădăcină.
- Nodurile care nu au descendenți se numesc *frunze*.
- *Înălțimea* arborelui este cel mai lung drum (ca număr de legături) de la rădăcina la o frunză.

Terminologie



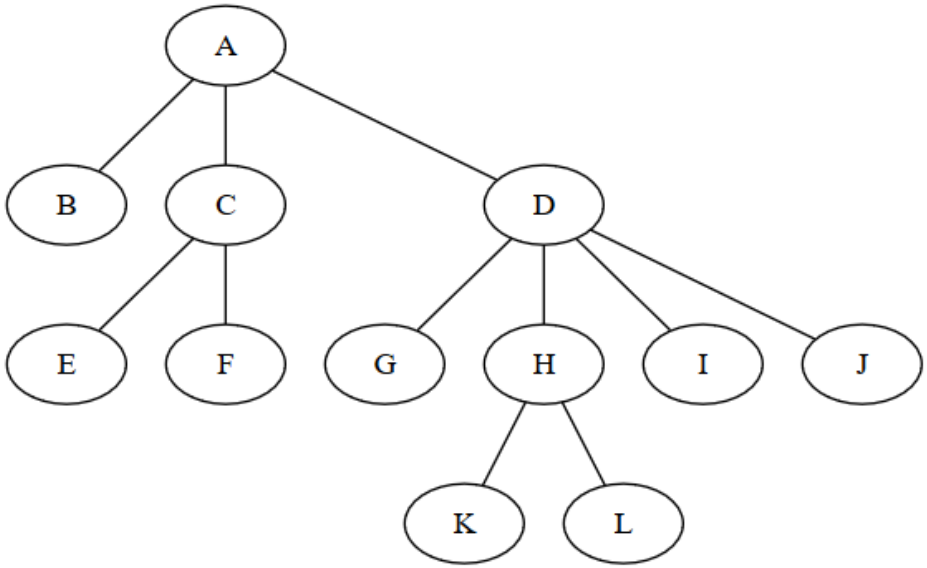
- Rădăcina

Terminologie



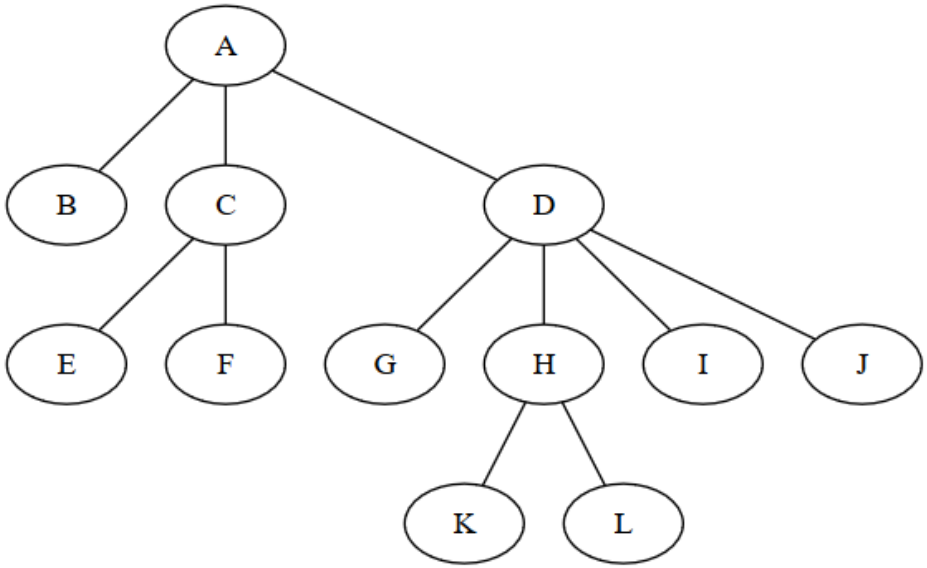
- Rădăcina - nodul cu A
- Frunze -

Terminologie



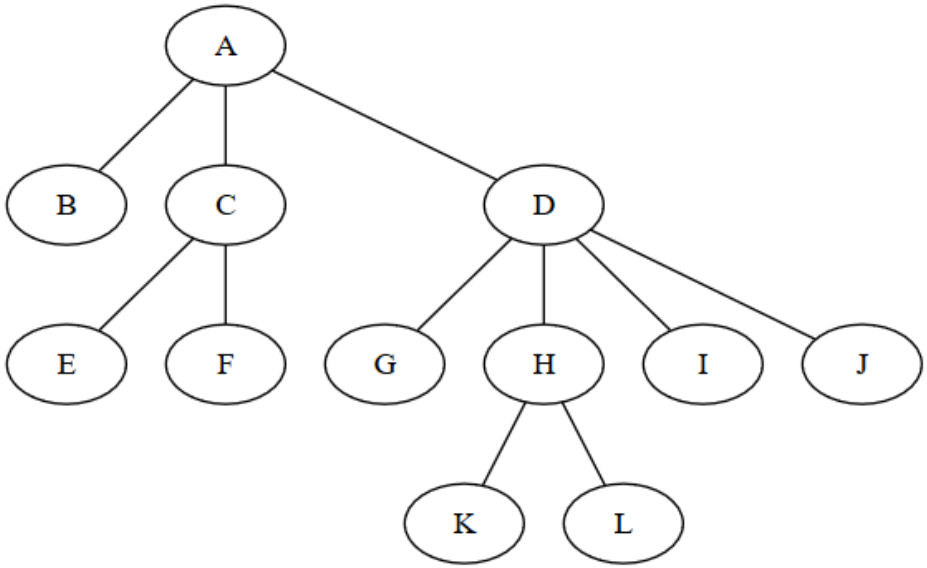
- Rădăcina - nodul cu A
- Frunze - nodurile cu B, E, F, G, K, L, I, J
- Descendenții lui D:

Terminologie



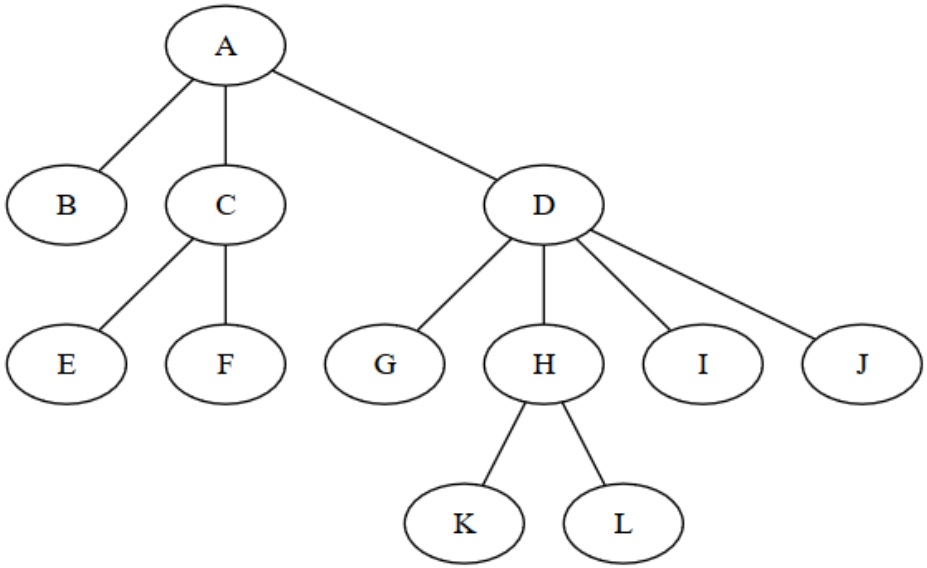
- Rădăcina - nodul cu A
- Frunze - nodurile cu B, E, F, G, K, L, I, J
- Descendenții lui D:G, H, I , J
- Părintele lui D:

Terminologie



- Rădăcina - nodul cu A
- Frunze - nodurile cu B, E, F, G, K, L, I, J
- Descendenții lui D:G, H, I , J
- Părintele lui D: A
- Înălțimea arborelui:

Terminologie

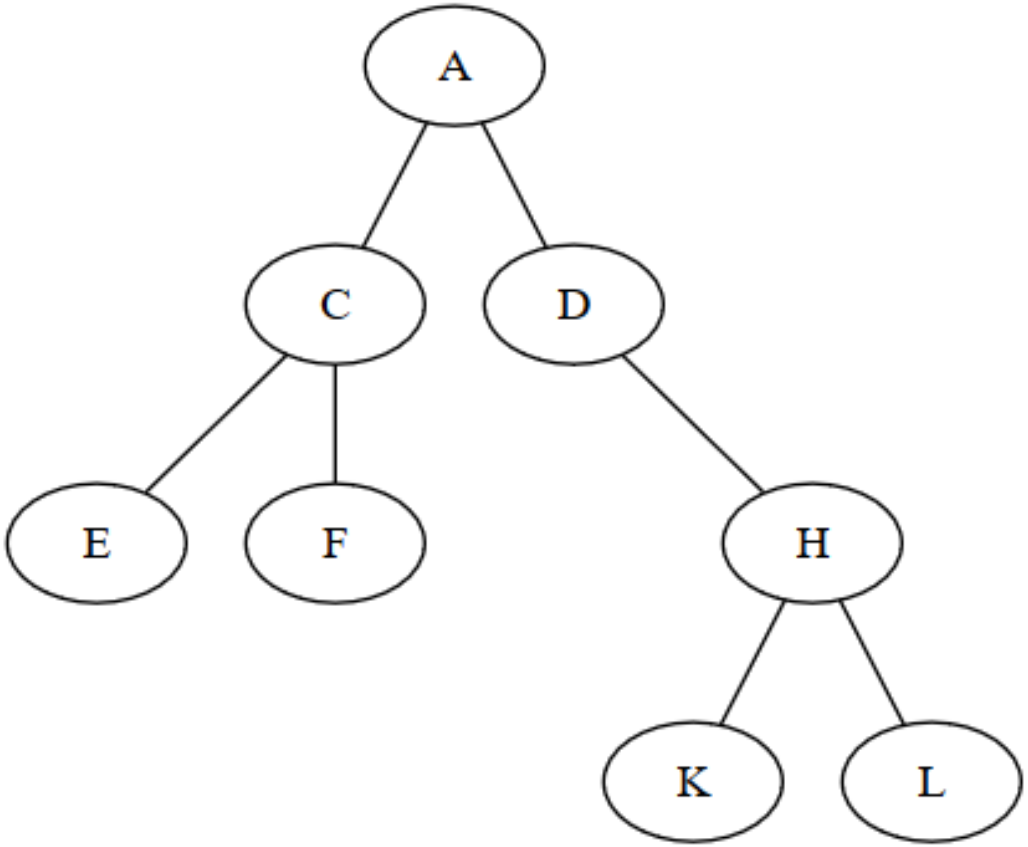


- Rădăcina - nodul cu A
- Frunze - nodurile cu B, E, F, G, K, L, I, J
- Descendenții lui D:G, H, I , J
- Părintele lui D: A
- Înălțimea arborelui: 3
- Sunt 12 noduri și 11 legături

- În funcție de numărul maxim de descendenți care pot exista pentru un nod, putem vorbi de:
  - Arbore Binar (maxim 2 descendenți - care se numesc descendent stâng și descendent drept)
  - Arbore N-ar (mai mult de 2 descendenți)

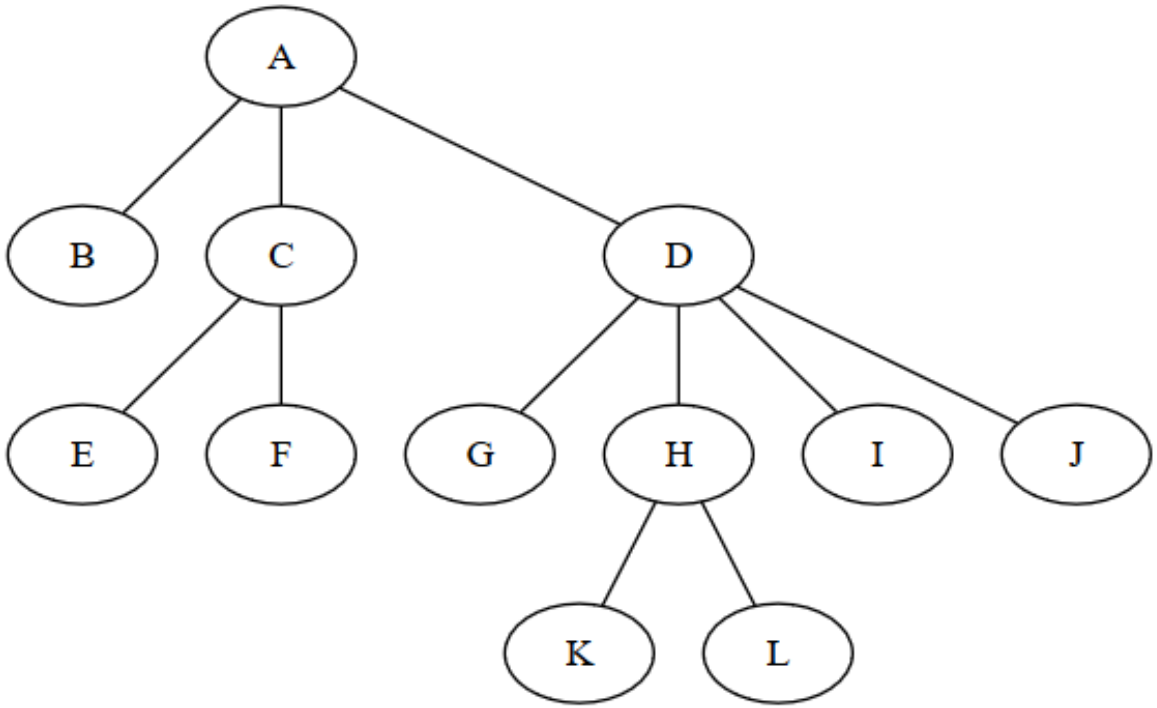
Arbore Binar

- Exemplu de Arbore Binar



Arbore N-ar

- Exemplu de Arbore N-ar



Operații pentru arbori

- Într-un arbore (binar sau n-ar) nu aveam acele operații cu care suntem obișnuiți de la celelalte structuri de date:
  - Nu aveam adăugare
  - Nu avem ștergere
  - Nu avem căutare
  - etc.
- Motivul este că arbori (arbore n-ar și arbore binar simplu) nu prea sunt folosiți ca reprezentare pentru containere.

- Atunci la ce putem folosi arbori, dacă nu la reprezentarea unor containere?
  - Arbori n-ari pot fi folosiți în probleme de căutare
  - Arbori binari pot fi folosiți pentru reprezentare de arbore genealogic, de expresii aritmetice, etc.

Probleme de căutare

- La problemele de căutare/optimizare căutăm o soluție pentru diferite probleme, pentru care nu putem găsi în mod "simplu" soluția
- De exemplu: a găsi o strategie pentru a câștiga jocul X-O (sau șah, sau alt joc), a completa un Sudoku, problema comis-voiajor, etc.
- În continuare ne concentrăm asupra jocului X-O

Jocul X-O

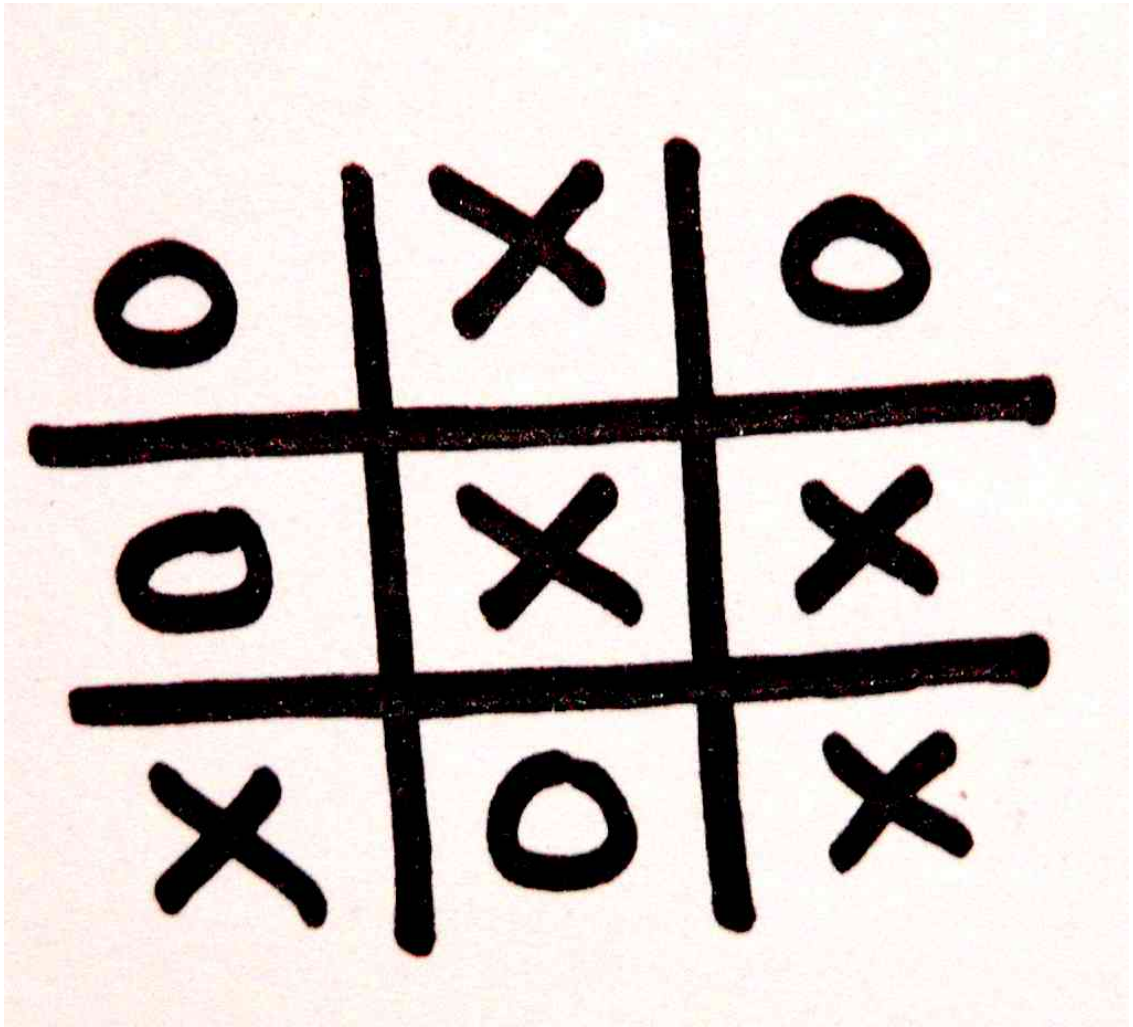


Figure: Sursă: https://kidavalanche.wordpress.com/2010/02/03/tic-tac-toe-magic-trick-prediction/

Probleme de căutare

- Pentru a rezolva o problemă cu metode de căutare, este necesară modelarea problemei:
  - Trebuie să existe un spațiu al stărilor posibile (pentru X-O este mulțimea tuturor posibilităților de a avea o parte din tablă completată).
  - Trebuie să avem o stare inițială (la X-O este tabla vidă).
  - Trebuie să avem una sau mai multe stări finale (la X-O este o tablă în care cineva a câștigat sau nu a câștigat nimeni, dar tabla este plină).
  - Trebuie să putem ajunge dintr-o stare într-o altă stare (la X-O punând un X sau un O pe tablă trecem dintr-o stare în alta).

Probleme de căutare

- Având toate discutate mai înainte, putem imagina un arbore al stărilor, care are ca rădăcină starea inițială, are ca frunze stările finale, și pentru fiecare stare, descendenții sunt stările în care putem ajunge din starea respectivă.

- Aceste arbori în general sunt mari, cu foarte multe noduri (pentru că sunt foarte multe stări)

- Pentru X-O avem:
  - O stare inițială
  - Primul jucător poate pune X în 9 locuri
  - Jucătorul 2 poate pune O în 8 locuri
  - Primul jucător poate pune X în 7 locuri
  - etc.
  - În total sunt 9*8*7*6*5*4*3*2*1 = 362,880 stări (de fapt sunt mai puține, pentru ca jocul se poate termina după 5 ture, și atunci nu mai continuăm)

Arbore cu stări pentru XO

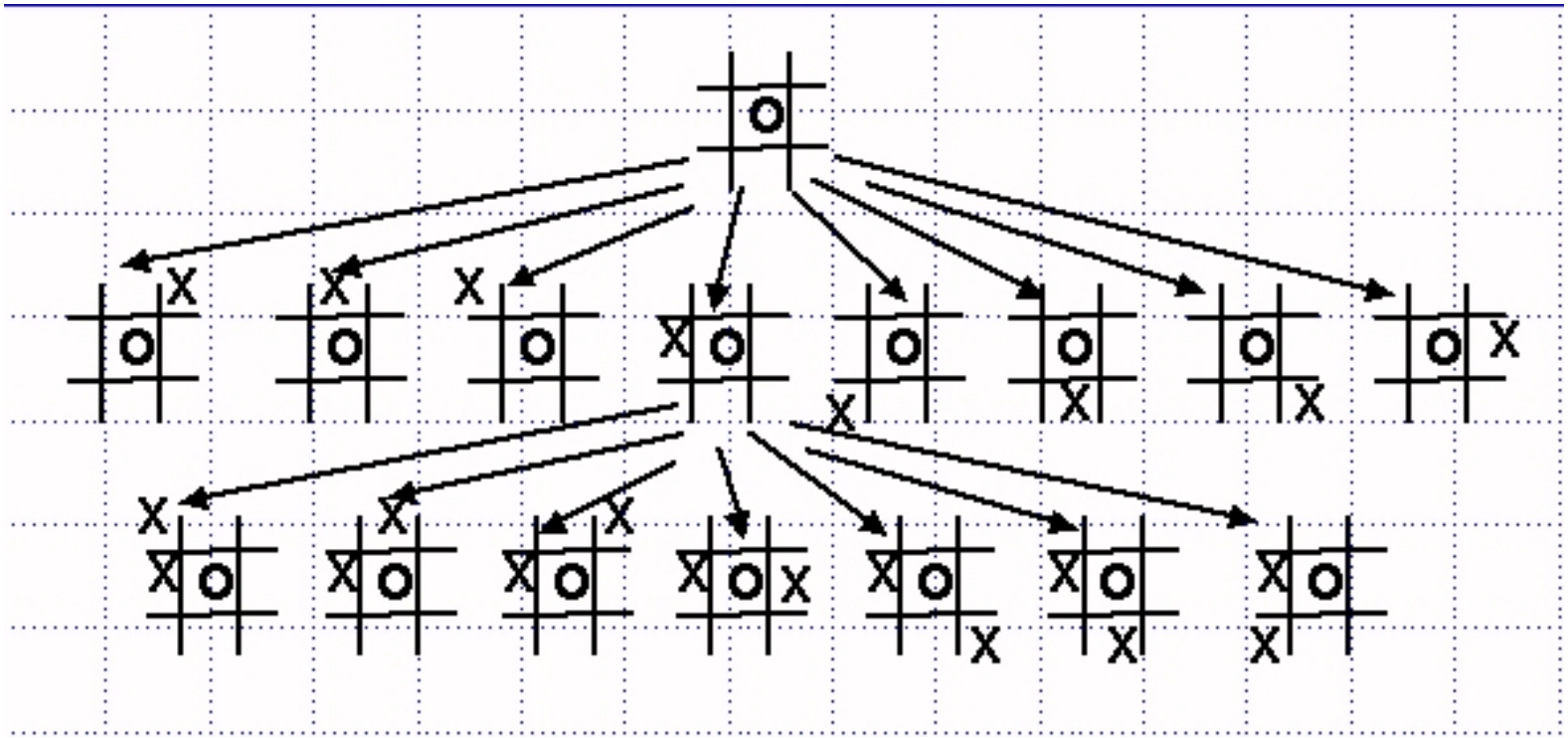


Figure: Sursa: http://www.doc.mmu.ac.uk/STAFF/D.McLean/kbsnotes/searchin/statesp.htm

Arbore cu stări pentru XO

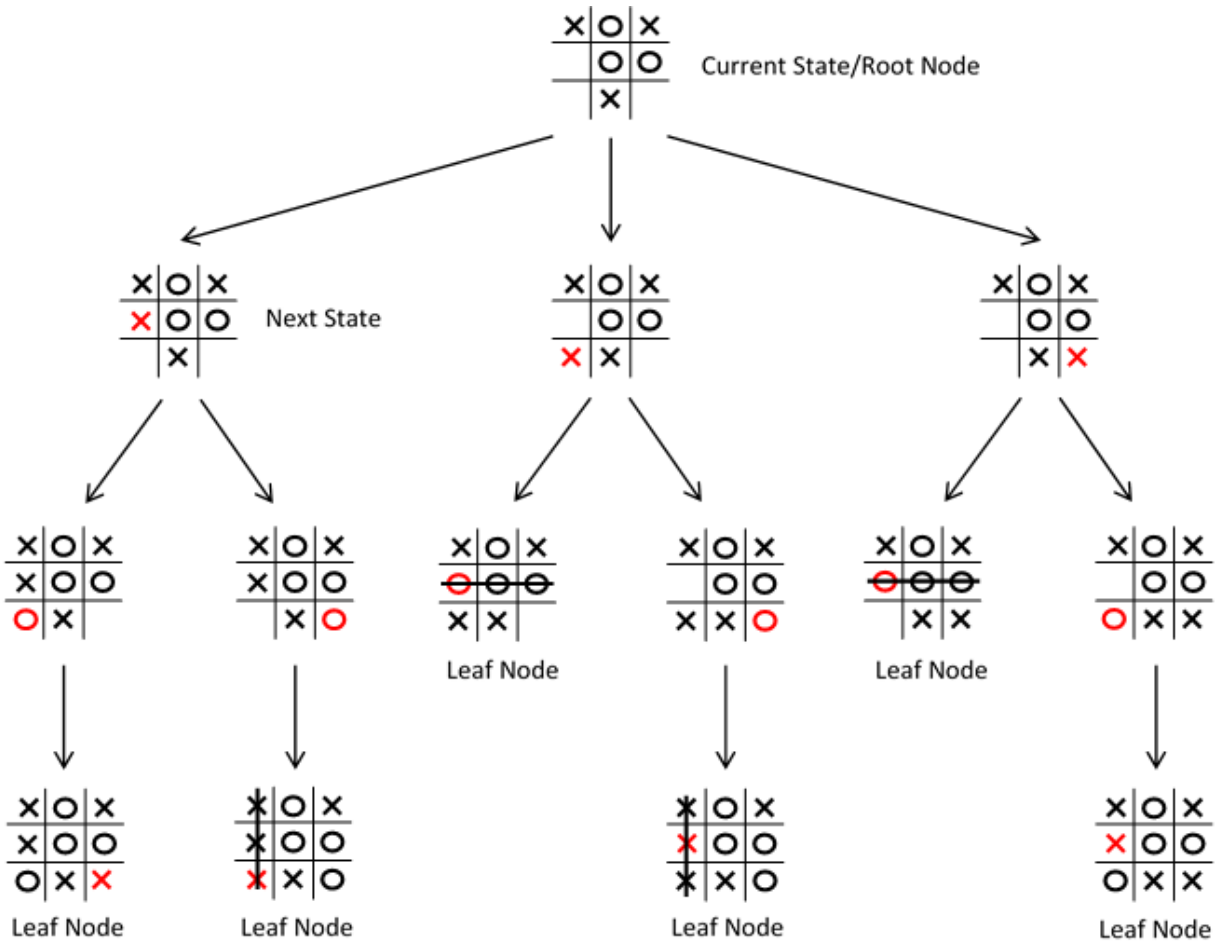


Figure: Sursa: https://www.eleceng.adelaide.edu.au/students/wiki/projects/index.php/Projects:42_Rule-based_AI_Agent_Development:_Tic_Tac_Toe

Probleme de căutare

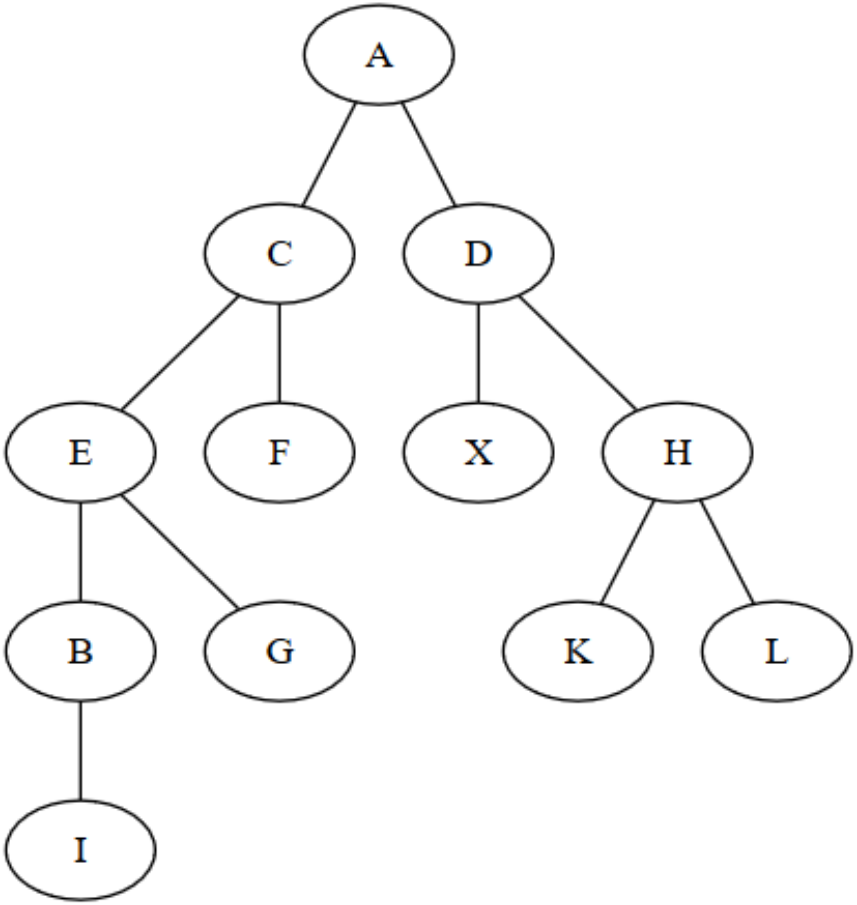
- Scopul nostru este să găsim un drum, de la starea inițială la o stare finală
- Pentru a găsi acest drum, trebuie să parcurgem arborele (adică să vizităm, toate nodurile - până găsim cel căutat).
- Există 2 variante simple de parcurgere:
  - Parcurgere în adâncime (DFS - Depth First Search)
  - Parcurgere în lățime (BFS - Breadth First Search)

Parcurgere în adâncime

- Parcurgerea în adâncime este o parcurgere în care luăm primul descendent al unui nod, după care luăm primul descendent al acestui nod, și așa mai departe până ajungem la o frunză.

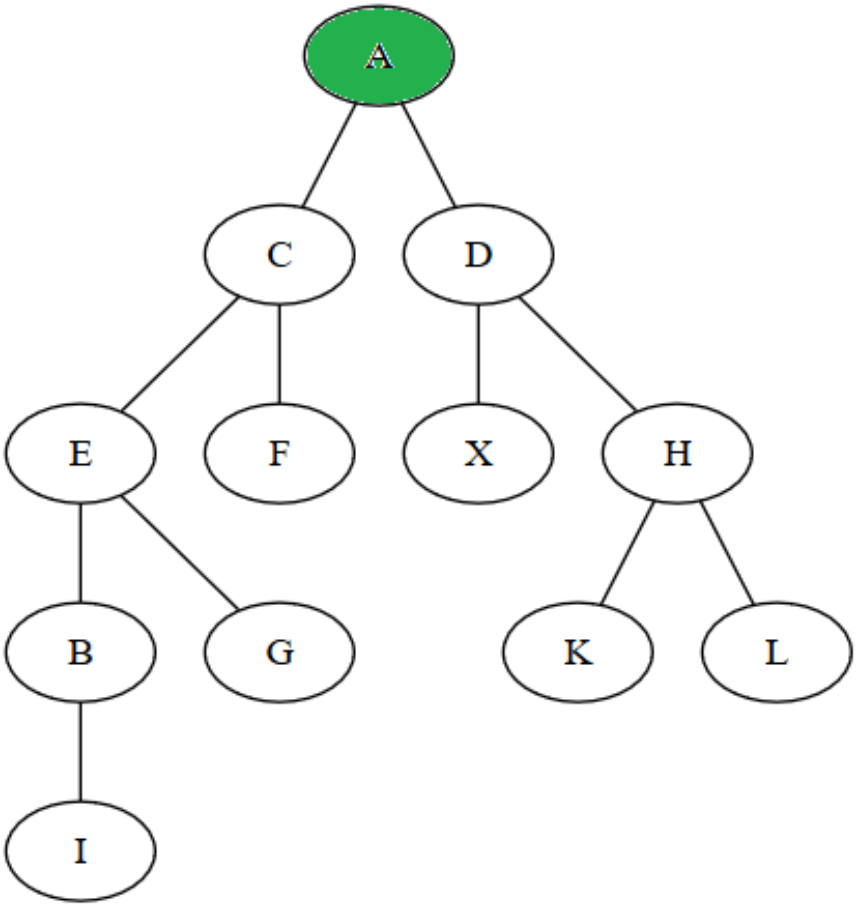
- După ce ajungem la o frunză, urcăm înapoi la părinte și dacă mai are noduri nevizitate, luăm pe primul și iar coborăm până la o frunză.

Parcurgere în adâncime



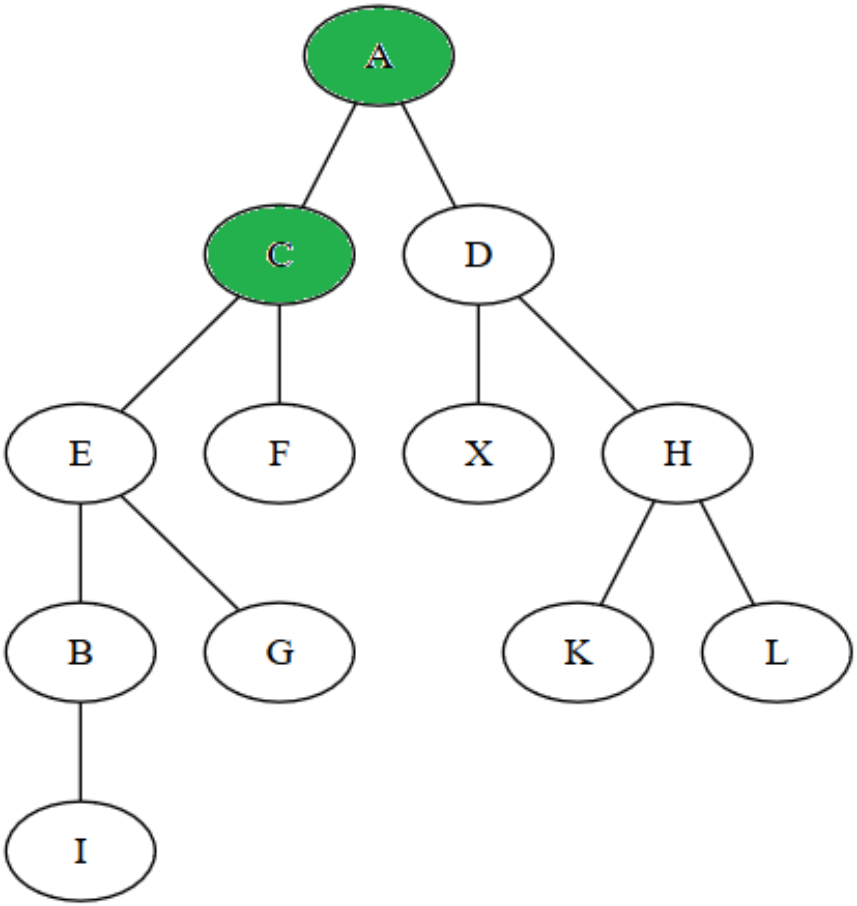
- Să vedem pas cu pas parcurgerea în adâncime pentru arborele de mai sus.

Parcurgere în adâncime



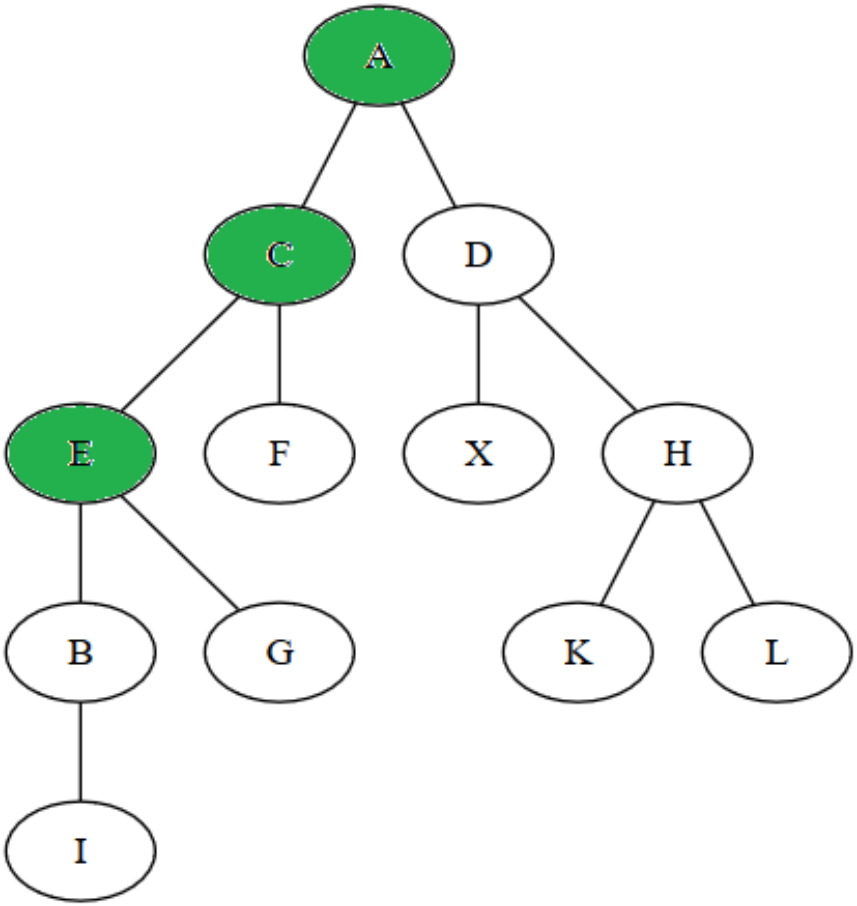
- Începem cu rădăcina.

Parcurgere în adâncime



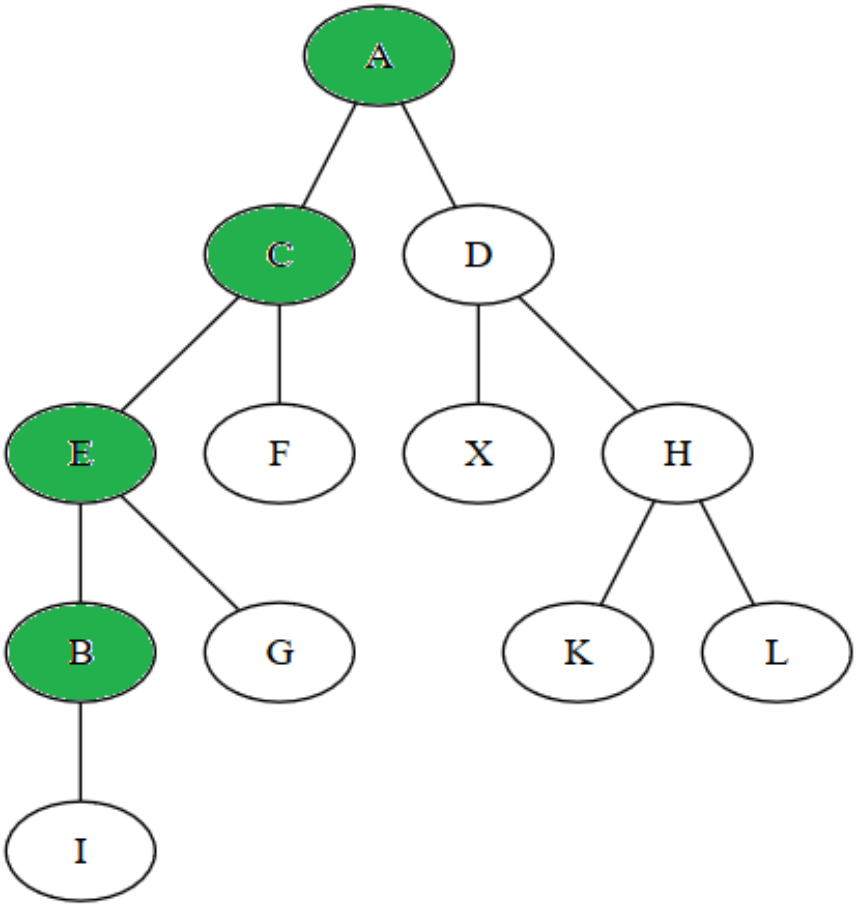
- Continuăm cu primul descendent al lui A, C.

Parcurgere în adâncime



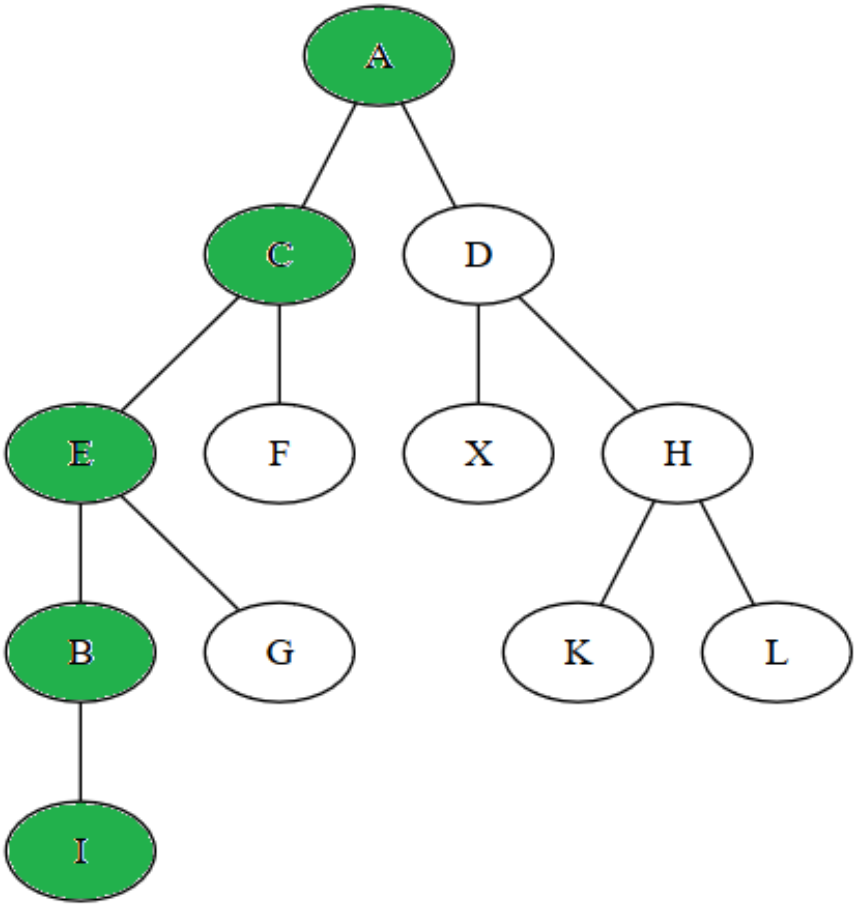
- Continuăm cu primul descendent al lui C, E.

Parcurgere în adâncime



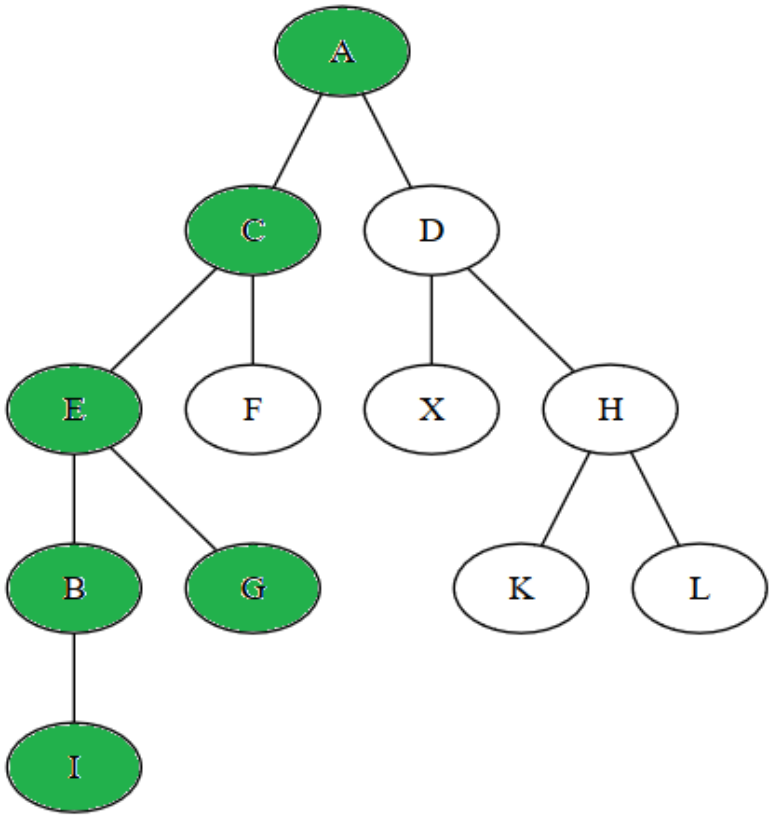
- Continuăm cu primul descendent al lui E, B.

Parcurgere în adâncime



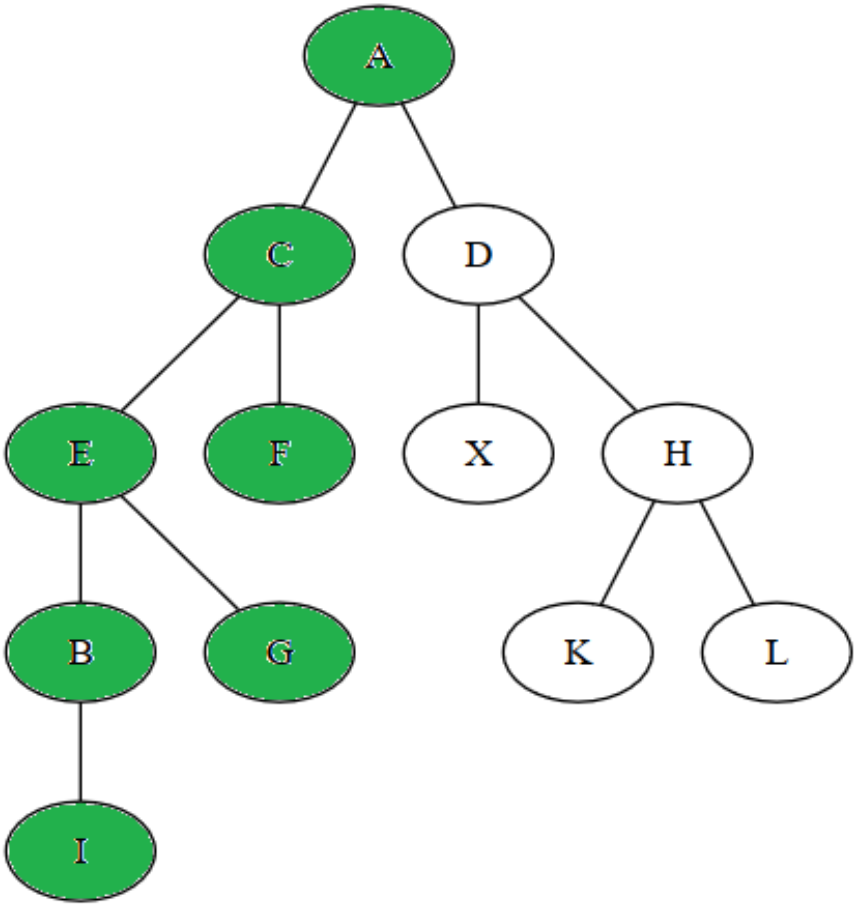
- Continuăm cu primul descendent al lui B, I.

Parcurgere în adâncime



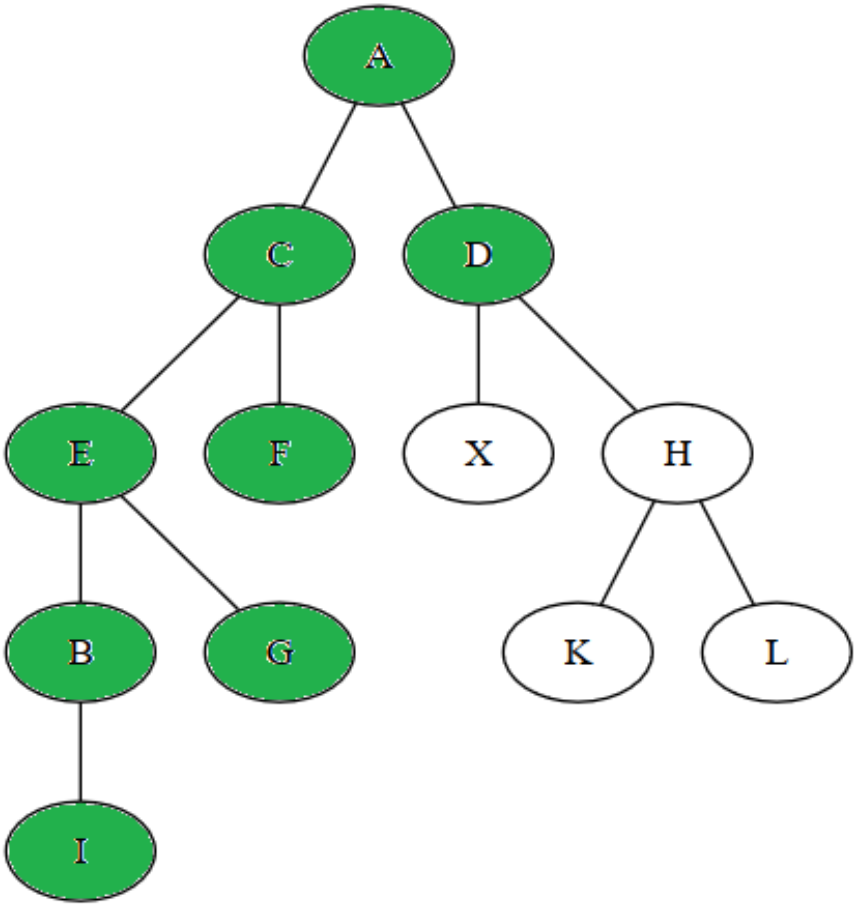
- I nu are descendent (deci nu putem merge *în jos*), urcăm la B. B nu are alt descendent, urcăm la E. E are alt descendent, deci mergem pe G.

Parcurgere în adâncime



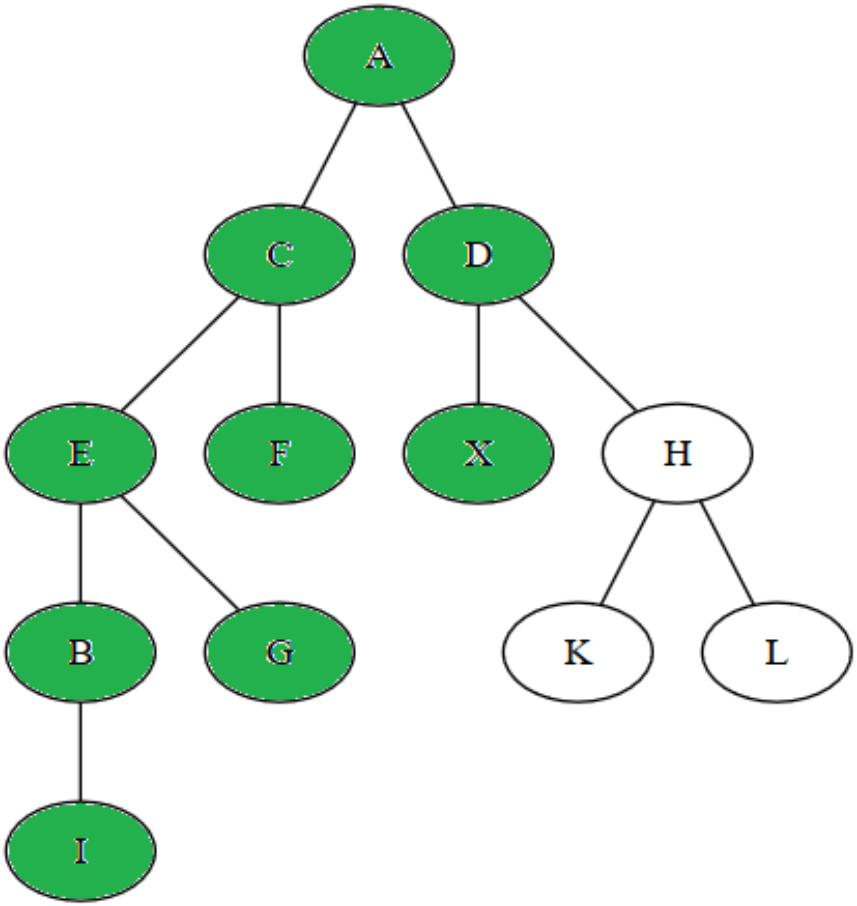
- G nu are descendent, urcăm la E. E nu mai are alt descendent, urcăm la C. C are alt descendent, deci mergem pe F.

Parcurgere în adâncime



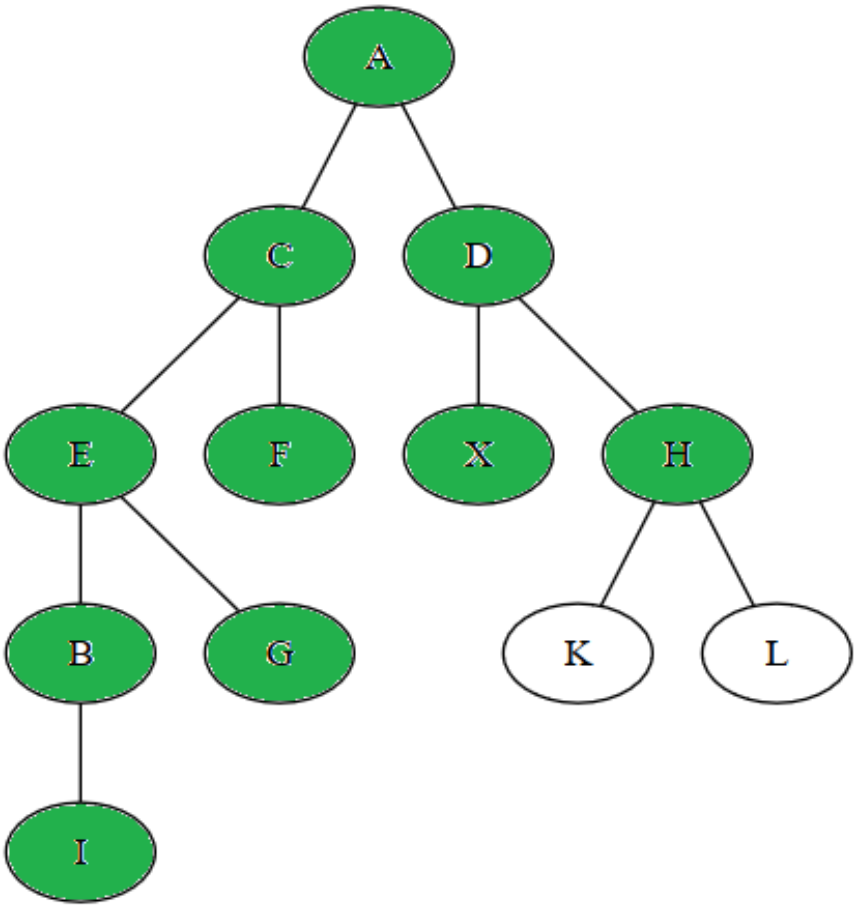
- F nu are descendent, urcăm la C. C nu are alt descendent, urcăm la A. A are alt descendent, mergem pe D.

Parcurgere în adâncime



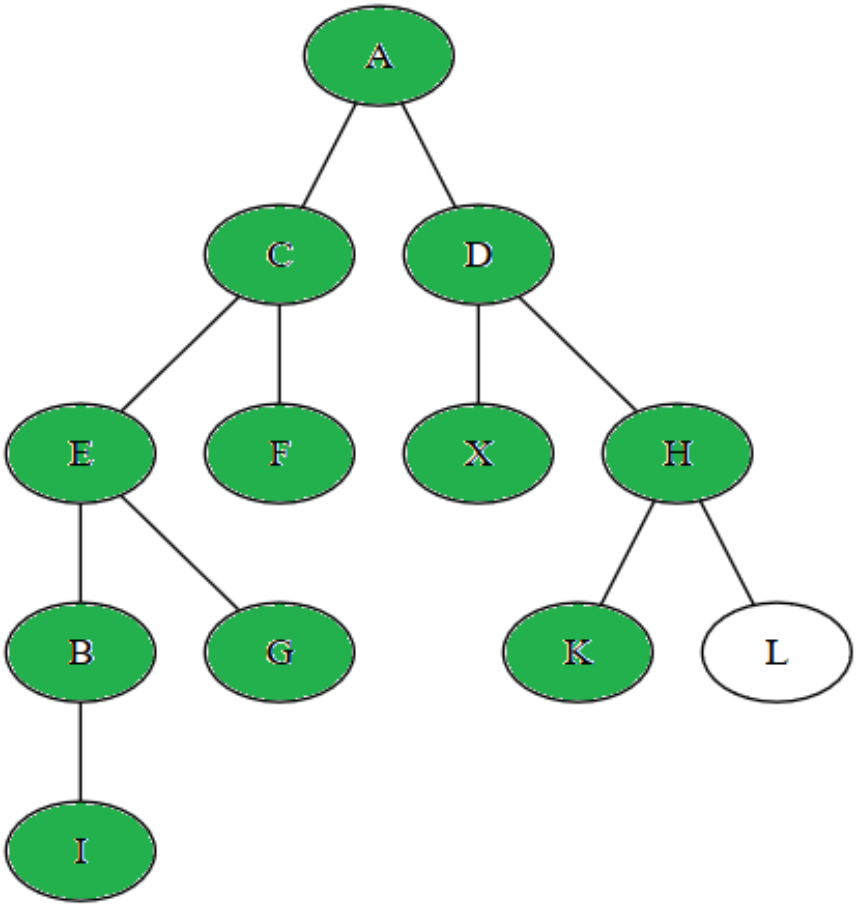
- Mergem pe primul descendent al lui D, X.

Parcurgere în adâncime



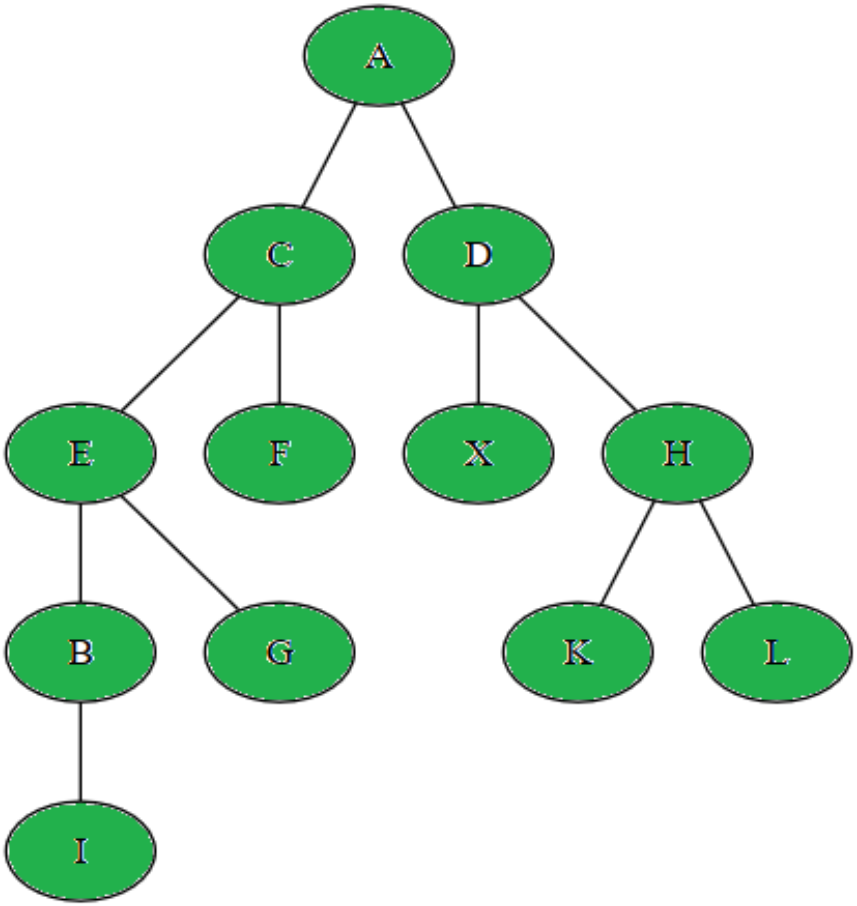
- X nu are descendent, urcamăm la D. Mergem pe al 2-lea descendent al lui D, H.

Parcurgere în adâncime



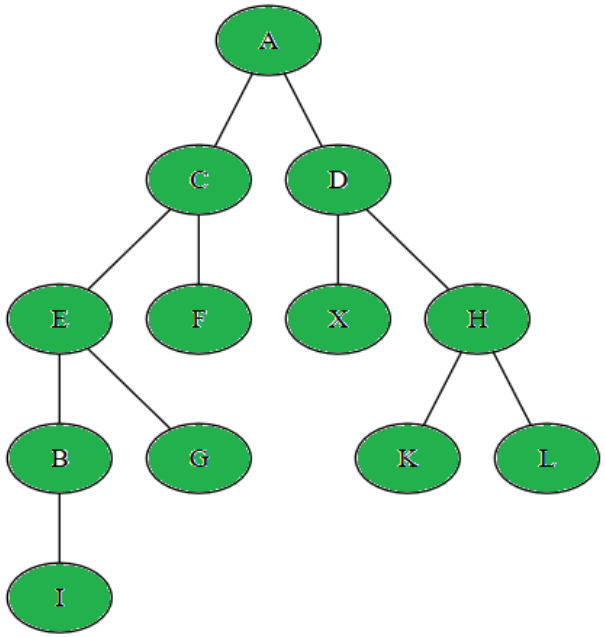
- Mergem pe primul descendent al lui H, K.

Parcurgere în adâncime



- K nu are descendent, urcăm la H. H are alt descendent, coborăm pe L.

Parcurgere în adâncime

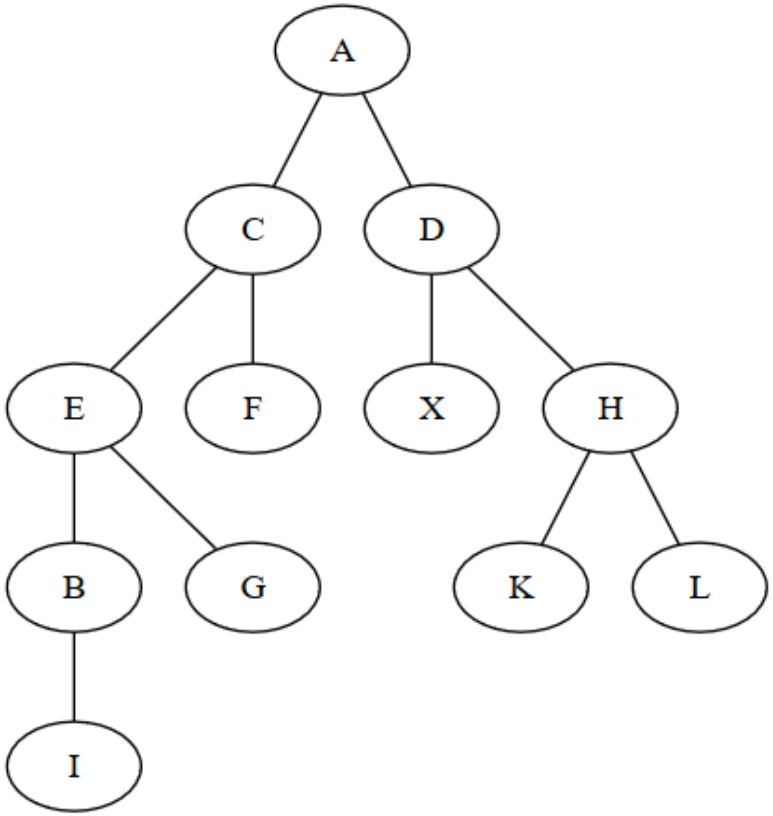


- L nu are descendent, urcăm la H. H nu are alt descendent, urcăm la D. D nu are alt descendent, urcăm la A. A nu are alt descendent, dar nici părinte, deci am terminat de parcurs nodurile.
- Ordinea în care am parcurs nodurile este: A, C, E, B, I, G, F, D, X, H, K, L

Parcurgerea în adâncime

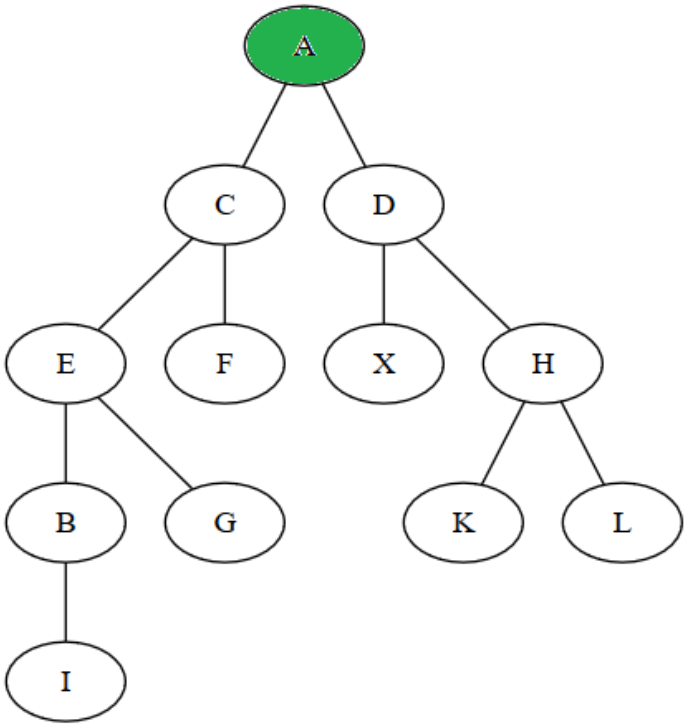
- Pe desen pare simplu. Nici implementarea nu este mai complicată. Pur și simplu avem nevoide de o *stivă* în care punem nodurile.
  - Pornim cu o stivă în care punem rădăcina arborelui.
  - Cât timp stiva nu este vidă scoatem un element din stivă (vizităm nodul corespunzător)
  - Dacă nodul scos din stivă are descendenți, adăugăm descendenții lui în stivă.

Parcurgere în adâncime



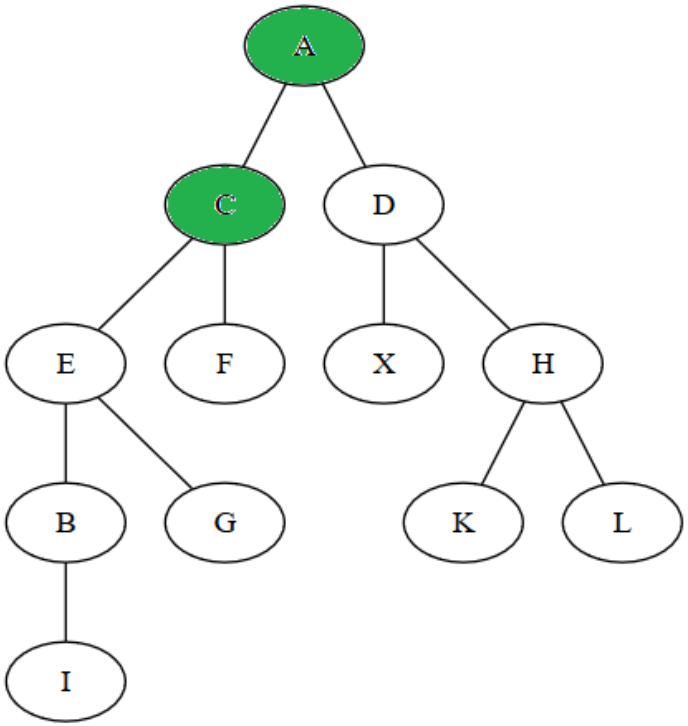
- Adăugăm în Stiva rădăcina. Presupunem că un nod este *vizitat*, când este șters din stivă.
- Stiva: A

Parcurgere în adâncime



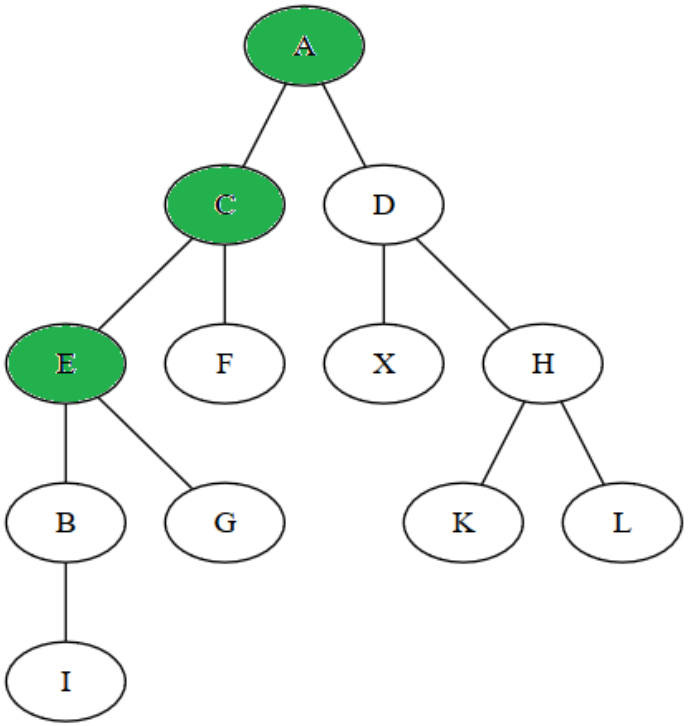
- Ștergem din stivă un element (A), îl vizităm, și adăugăm descendenții lui A în stivă (de la dreapta la stânga).
- Stiva: D, C

Parcurgere în adâncime



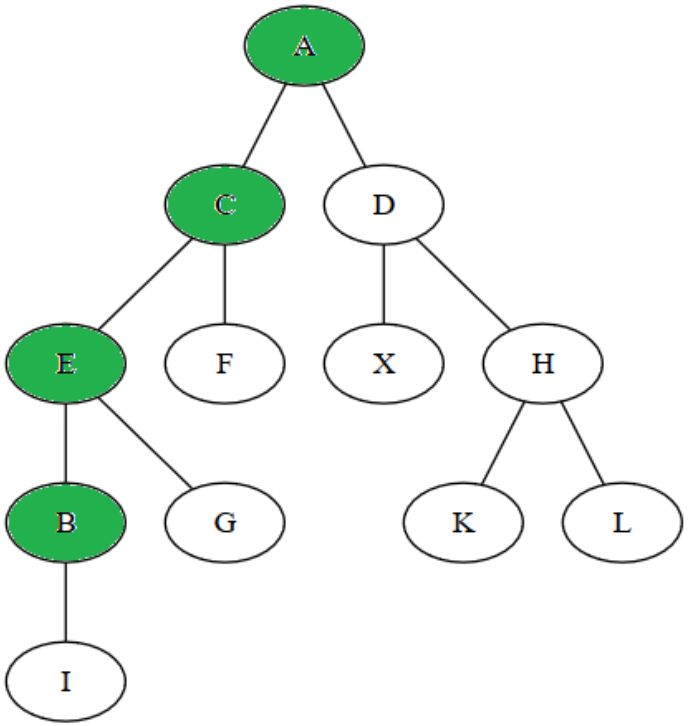
- Ștergem din stivă un element (C), îl vizită, și adăugăm descendenții lui C în stivă (de la dreapta la stânga).
- Stiva: D, F, E

Parcurgere în adâncime



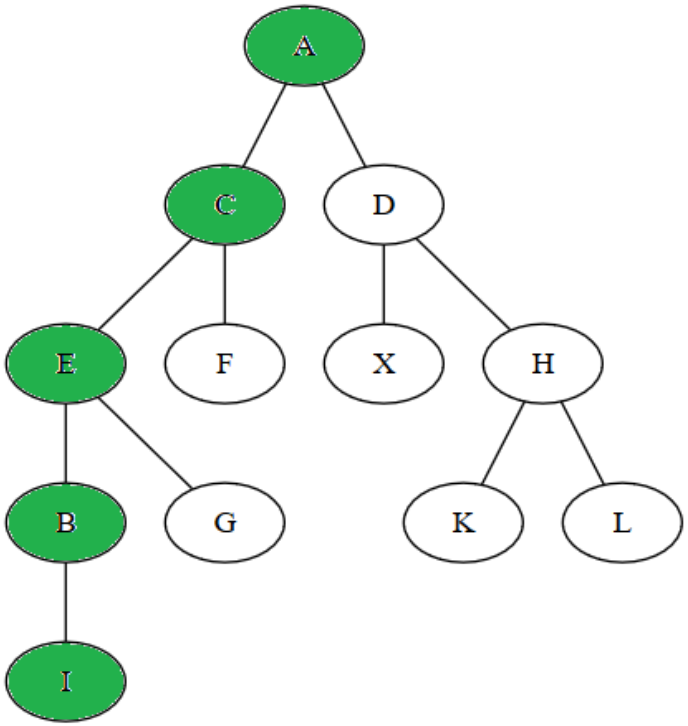
- Ștergem din stivă un element (E), îl vizită, și adăugăm descendenții lui E în stivă (de la dreapta la stânga).
- Stiva: D, F, G, B

Parcurgere în adâncime



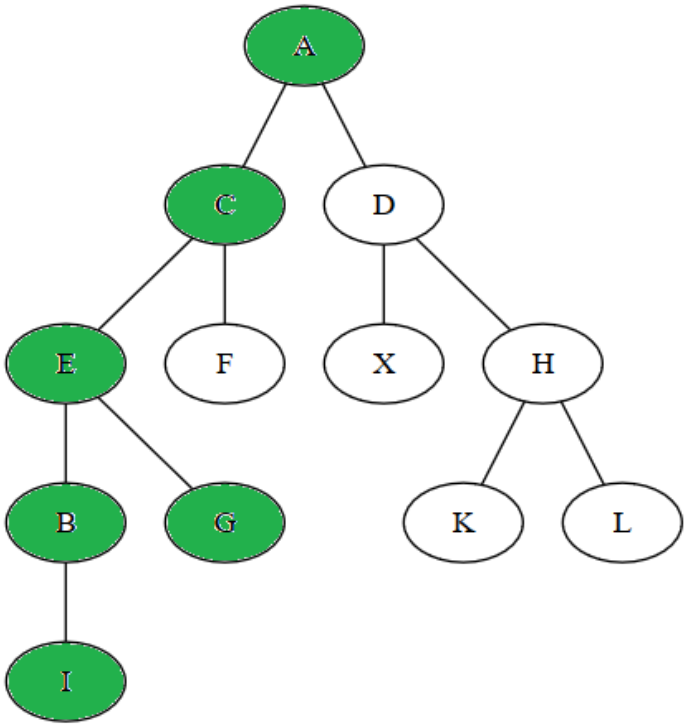
- Ștergem din stivă un element (B), îl vizită, și adăugăm descendentul lui B în stivă
- Stiva: D, F, G, I

Parcurgere în adâncime



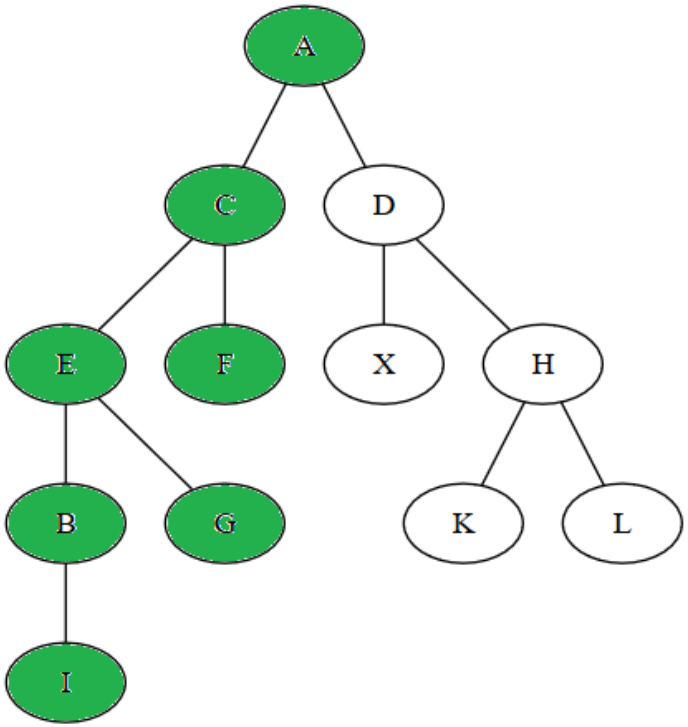
- Ștergem din stivă un element (I), îl vizităm, și nu adăugăm nimic în stivă, pentru că I nu are descendenți.
- Stiva: D, F, G

Parcurgere în adâncime



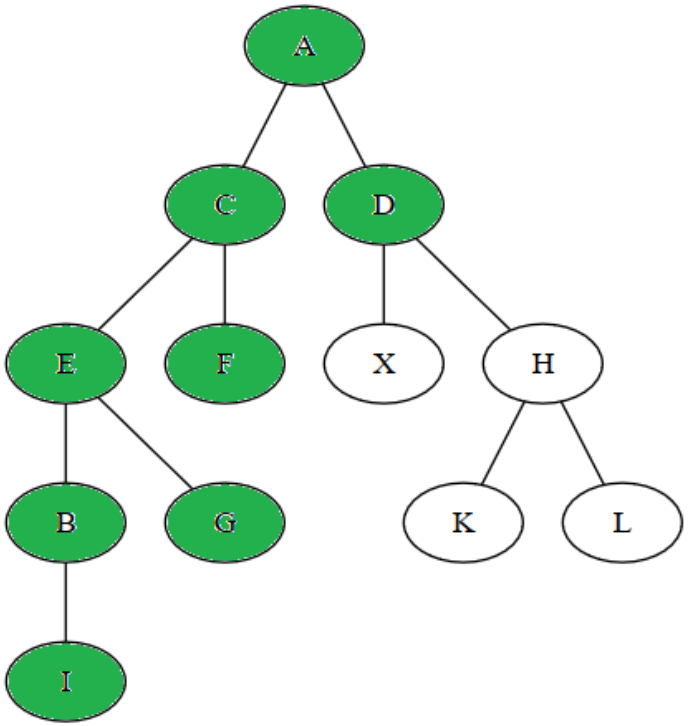
- Ștergem din stivă un element (G), îl vizităm, și nu adăugăm nimic în stivă, pentru că G nu are descendenți.
- Stiva: D, F

Parcurgere în adâncime



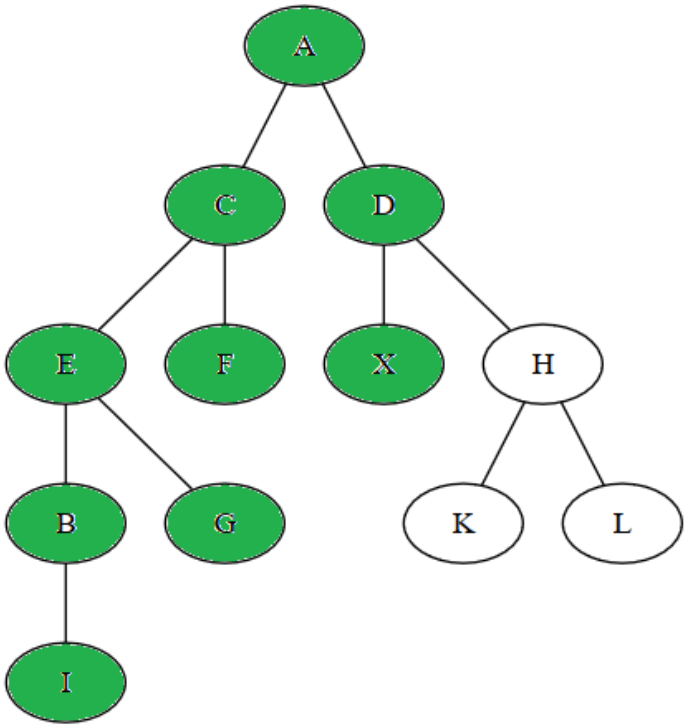
- Ștergem din stivă un element (F), îl vizităm, și nu adăugăm nimic în stivă, pentru că F nu are descendenți.
- Stiva: D

Parcurgere în adâncime



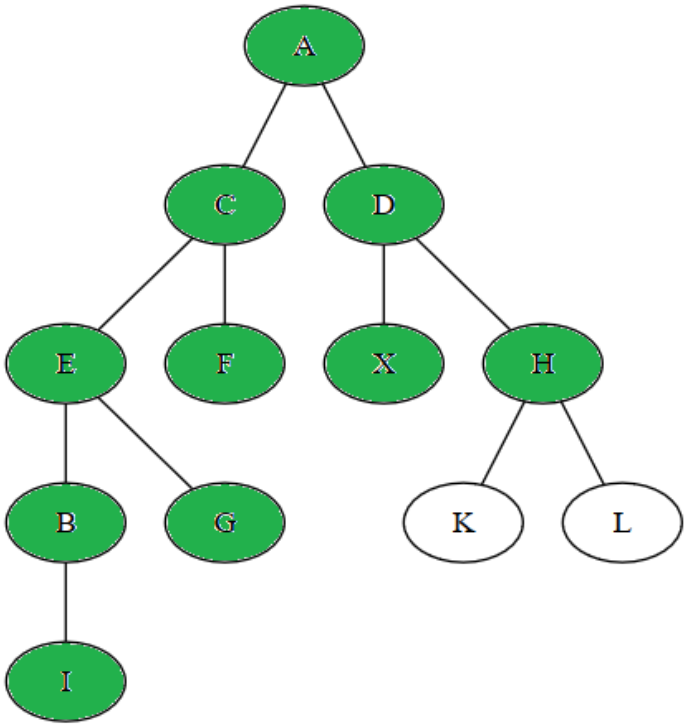
- Ștergem din stivă un element (D), îl vizităm, și adăugăm în stivă descendenții lui D (de la dreapta la stânga)
- Stiva: H, X

Parcurgere în adâncime



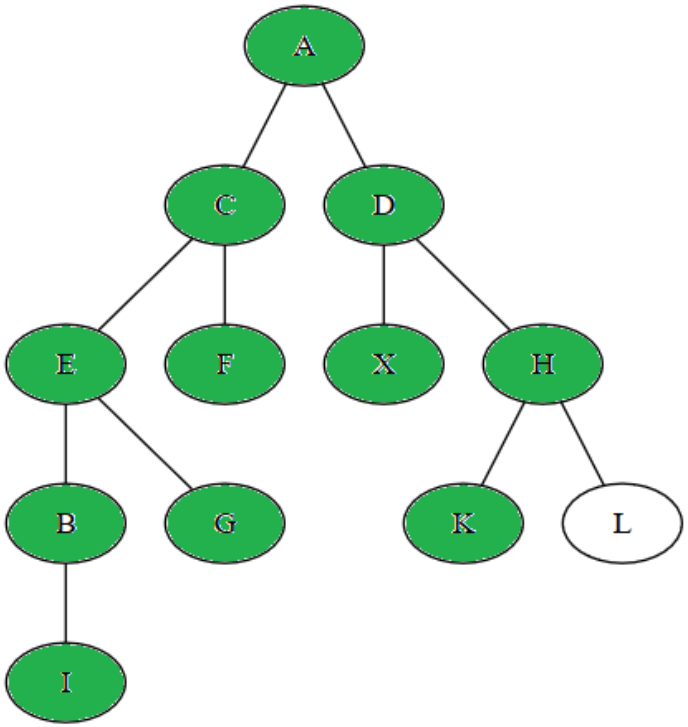
- Ștergem din stivă un element (X), îl vizităm, și nu adăugăm nimic în stivă, pentru că X nu are descendenți.
- Stiva: H

Parcurgere în adâncime



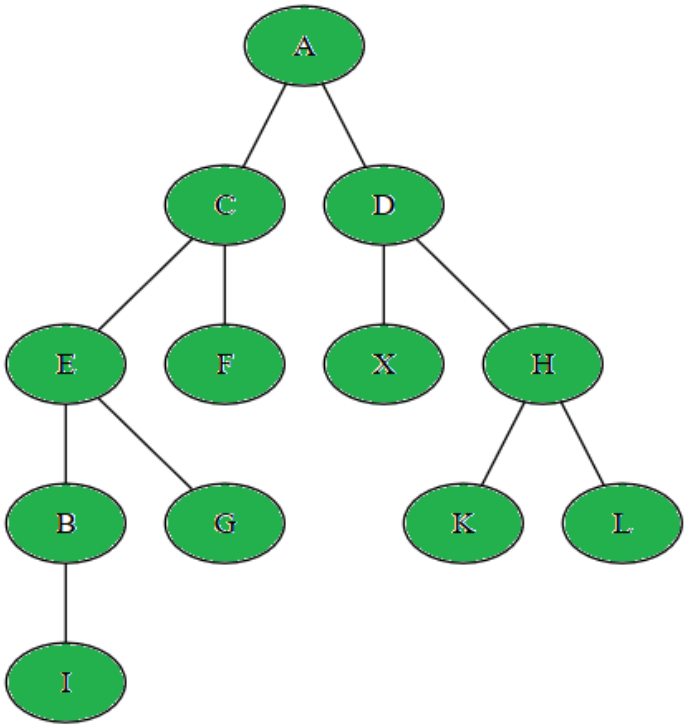
- Ștergem din stivă un element (H), îl vizităm, și adăugăm descendenții lui H în stivă (de la dreapta la stânga)
- Stiva: L, K

Parcurgere în adâncime



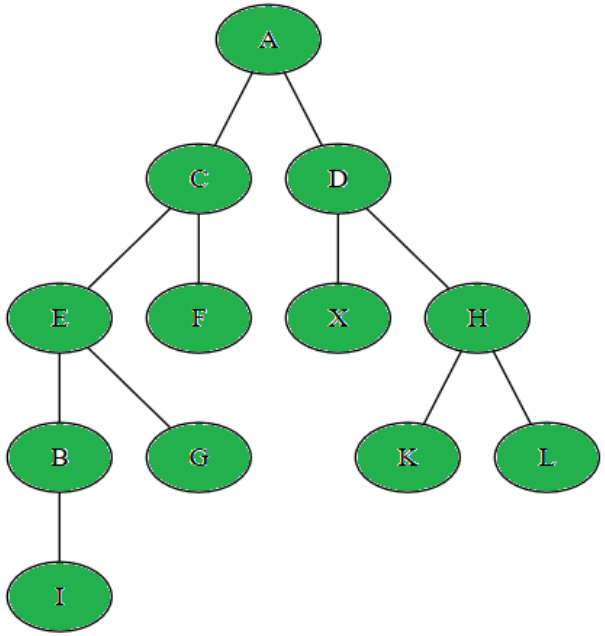
- Ștergem din stivă un element (K), îl vizităm, și nu adăugăm nimic în stivă, pentru că K nu are descendenți.
- Stiva: L

Parcurgere în adâncime



- Ștergem din stivă un element (L), îl vizităm, și nu adăugăm nimic în stivă, pentru că L nu are descendenți.
- Stiva:

Parcurgere în adâncime

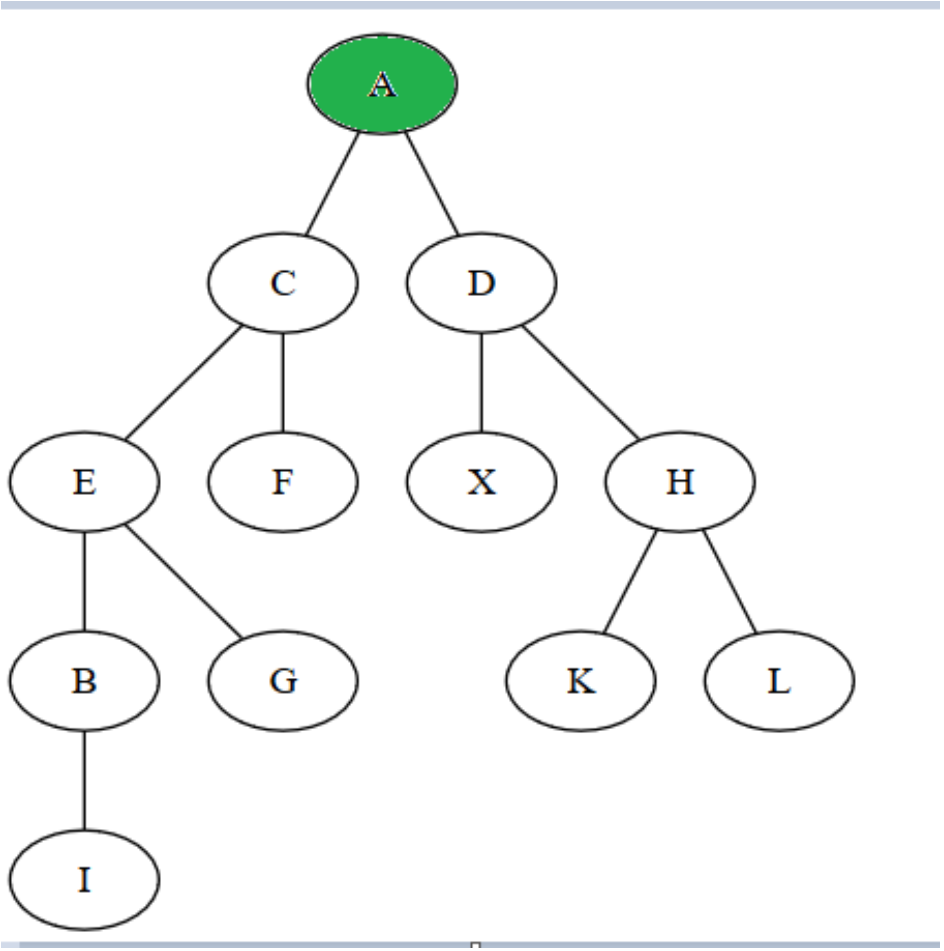


- Stiva este vidă, deci am terminat de parcurs toate nodurile.
- Ordinea în care am parcurs nodurile este: A, C, E, B, I, G, F, D, X, H, K, L

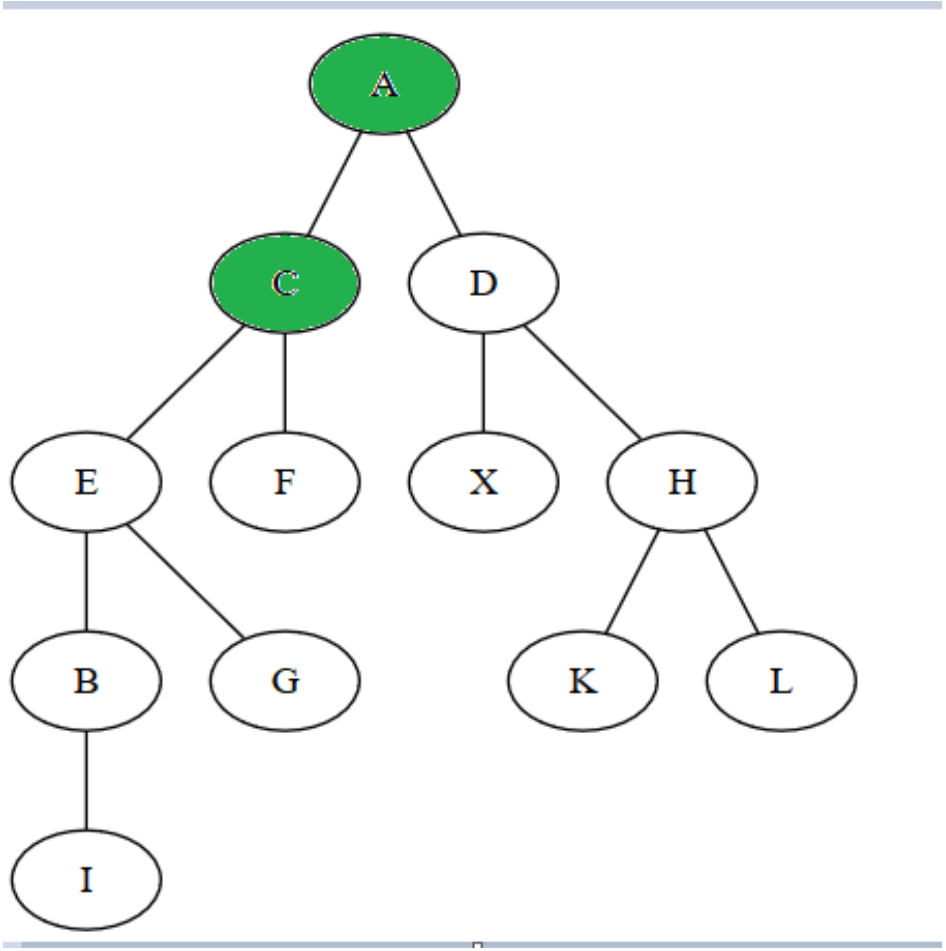
Parcurgere în lățime

- Cealaltă parcurgere des întâlnită la arbori n-ari este parcurgerea în lățime.
- Pornim de la rădăcină.
- Vizităm pe rând toți descendenții rădăcinii, după aceea descendenții descendenților și așa mai departe.
- Practic, doar când terminăm cu toate nodurile de pe un nivel, trecem la nodurile de pe nivelul următor.

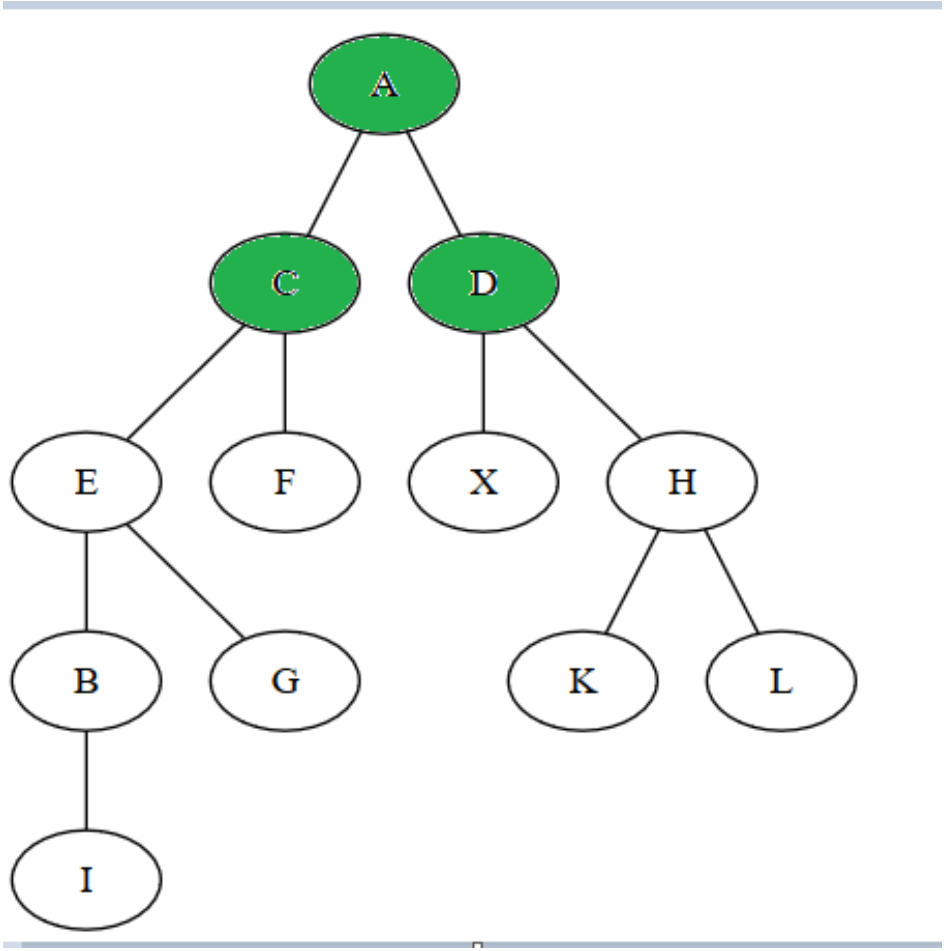
Parcurgere în lățime



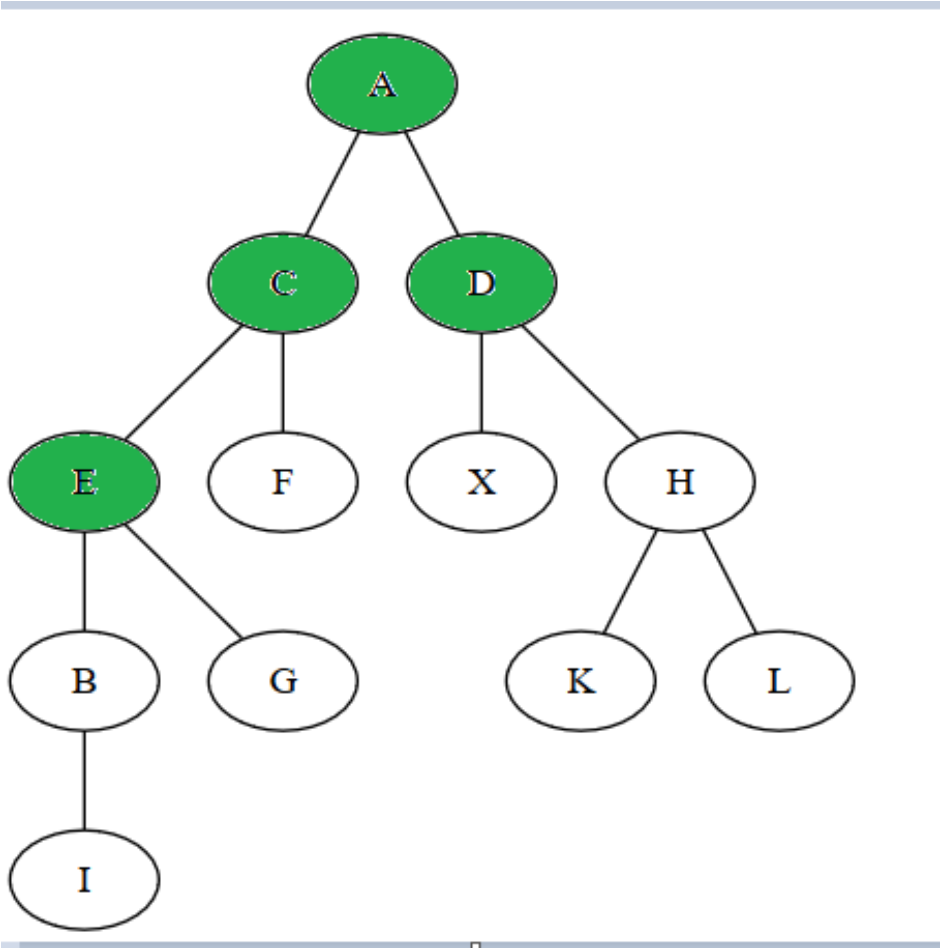
Parcurgere în lățime



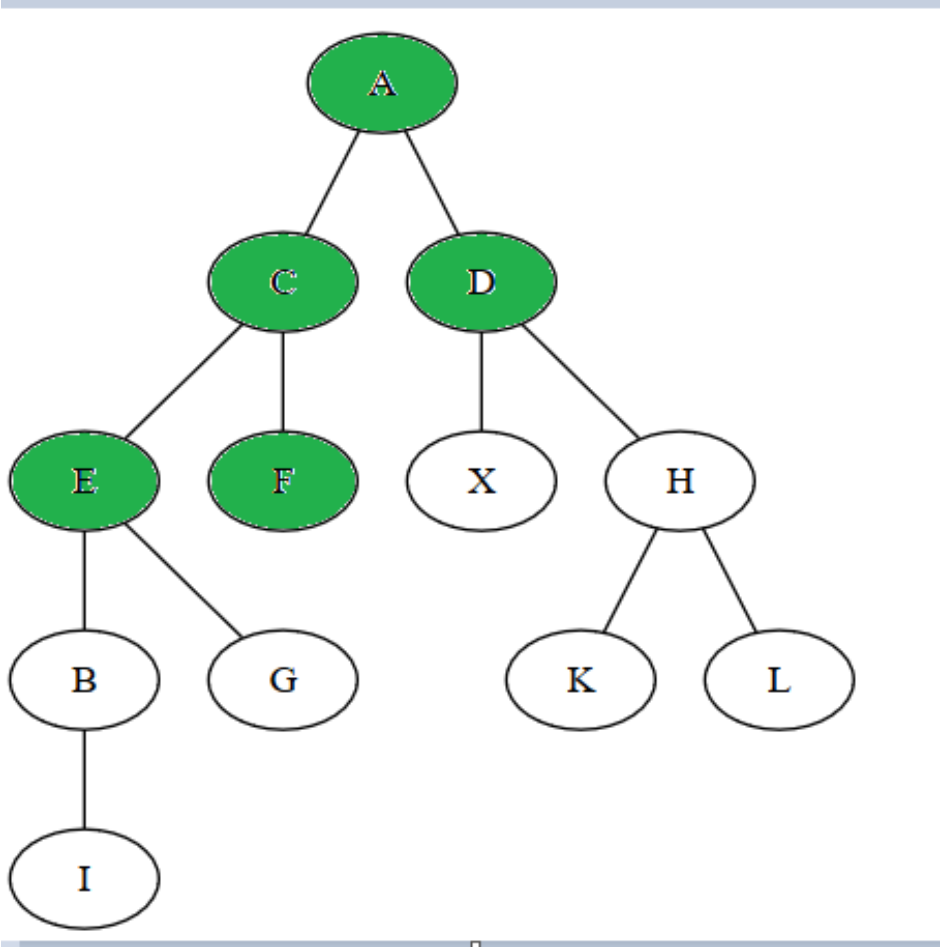
Parcurgere în lățime



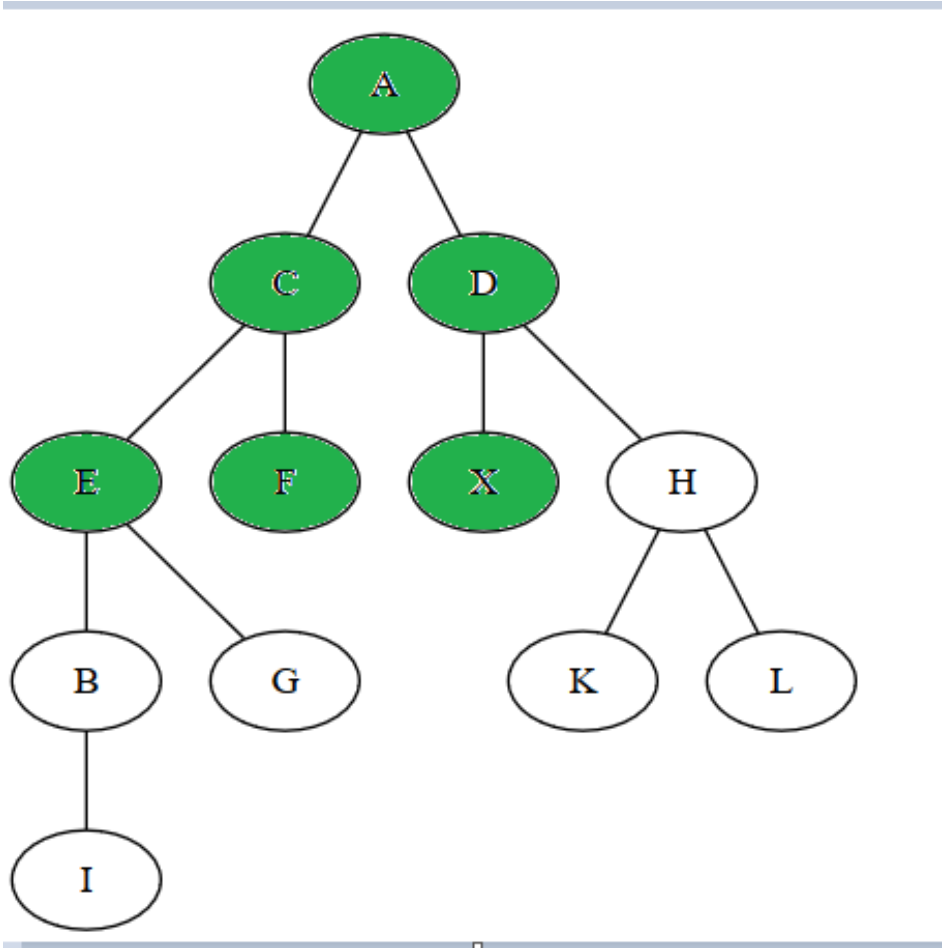
Parcurgere în lățime



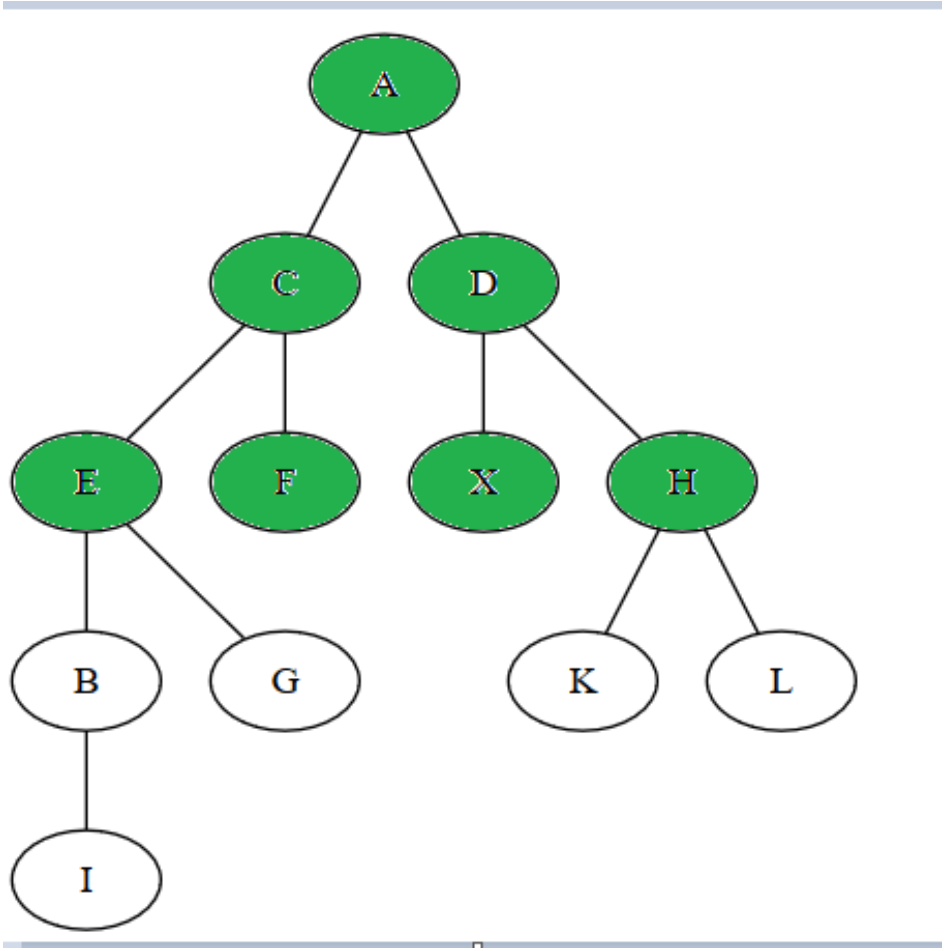
Parcurgere în lățime



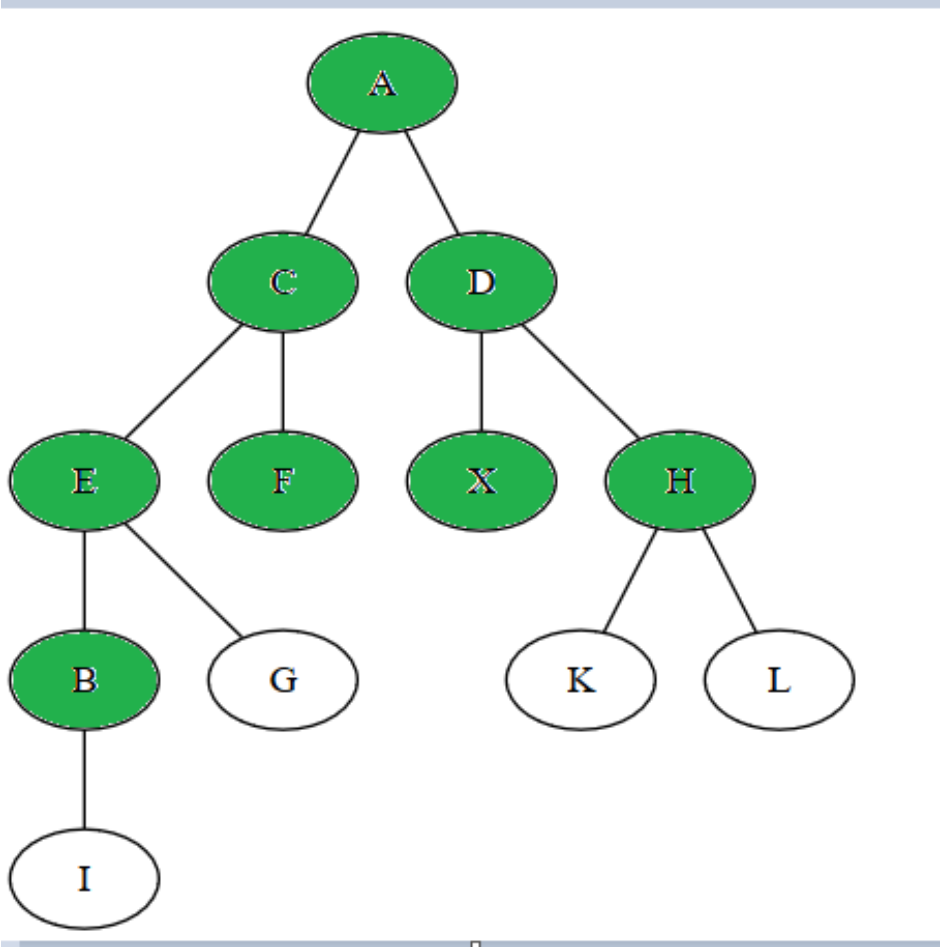
Parcurgere în lățime



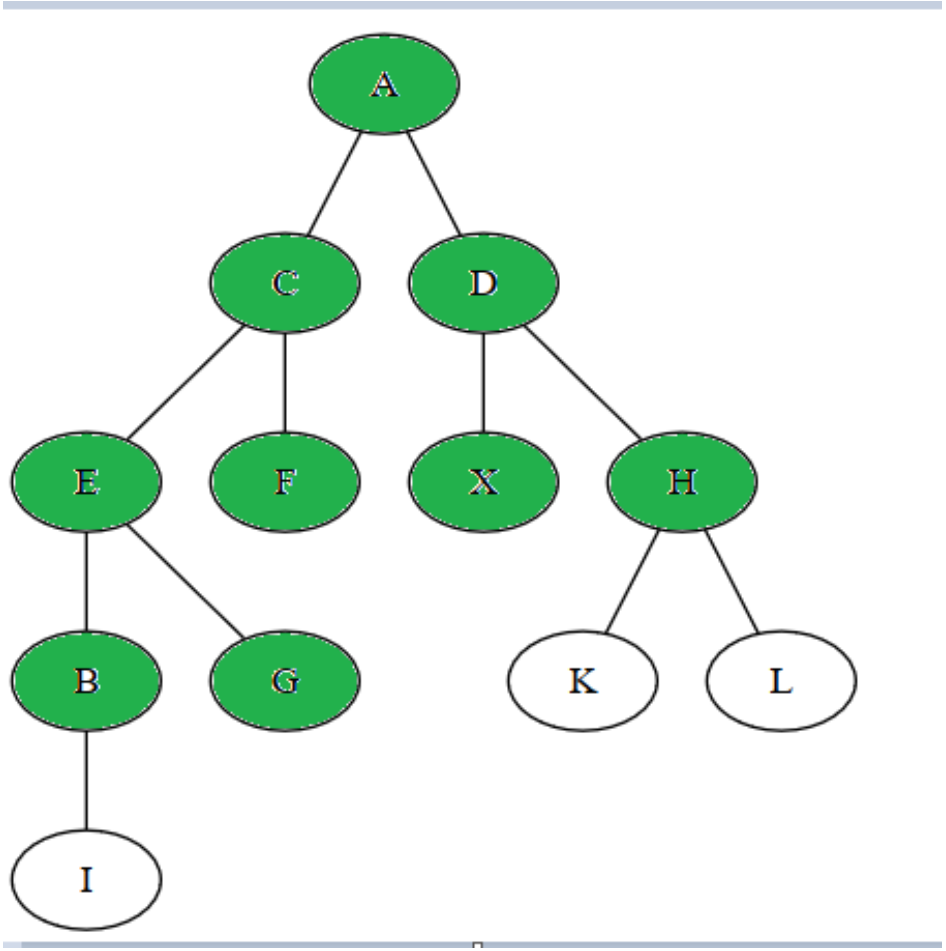
Parcurgere în lățime



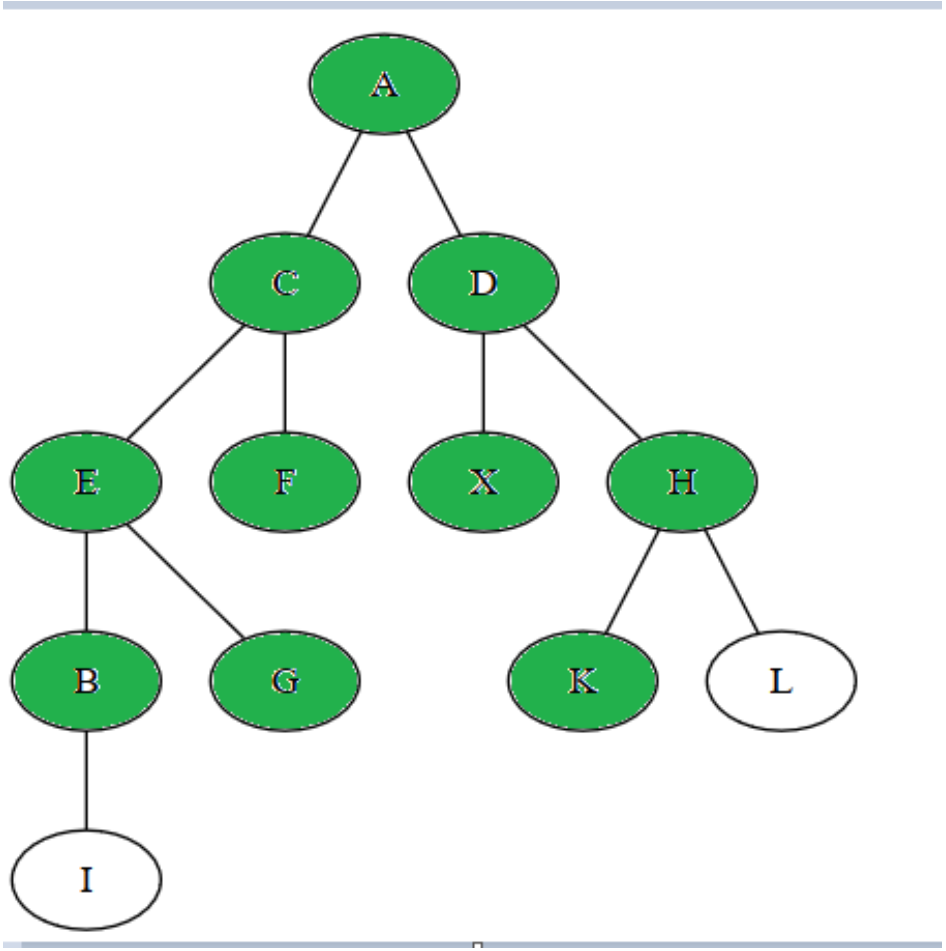
Parcurgere în lățime



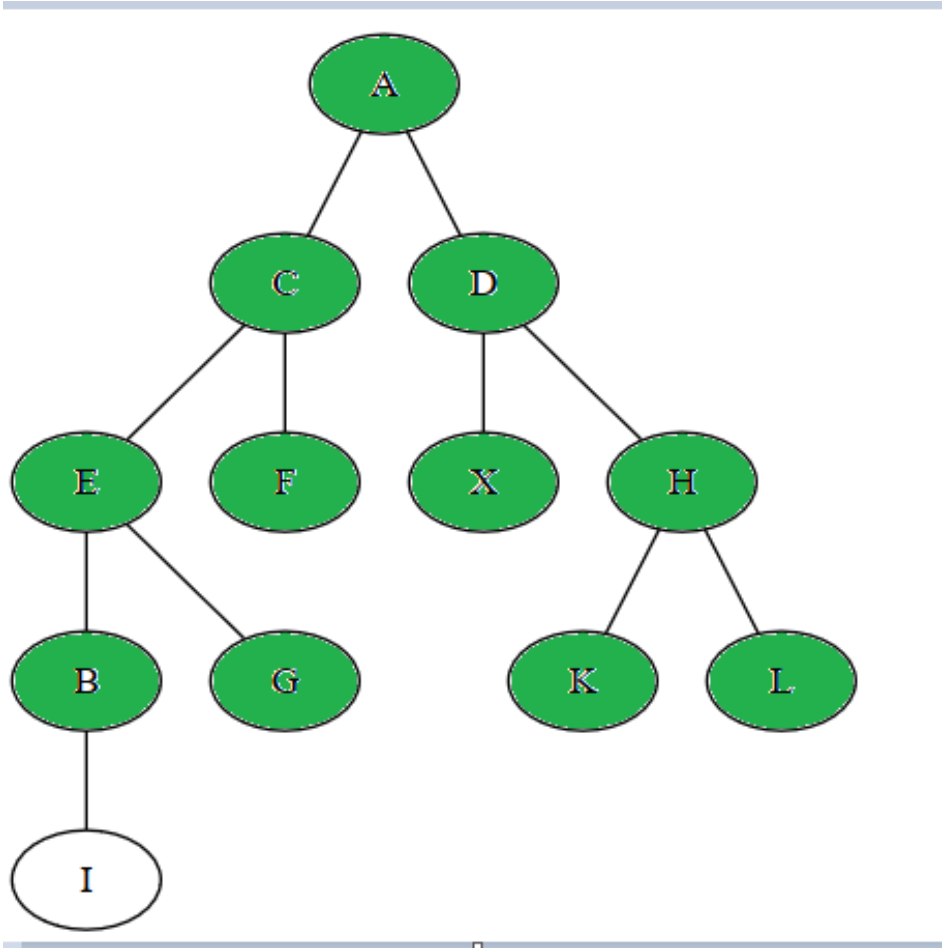
Parcurgere în lățime



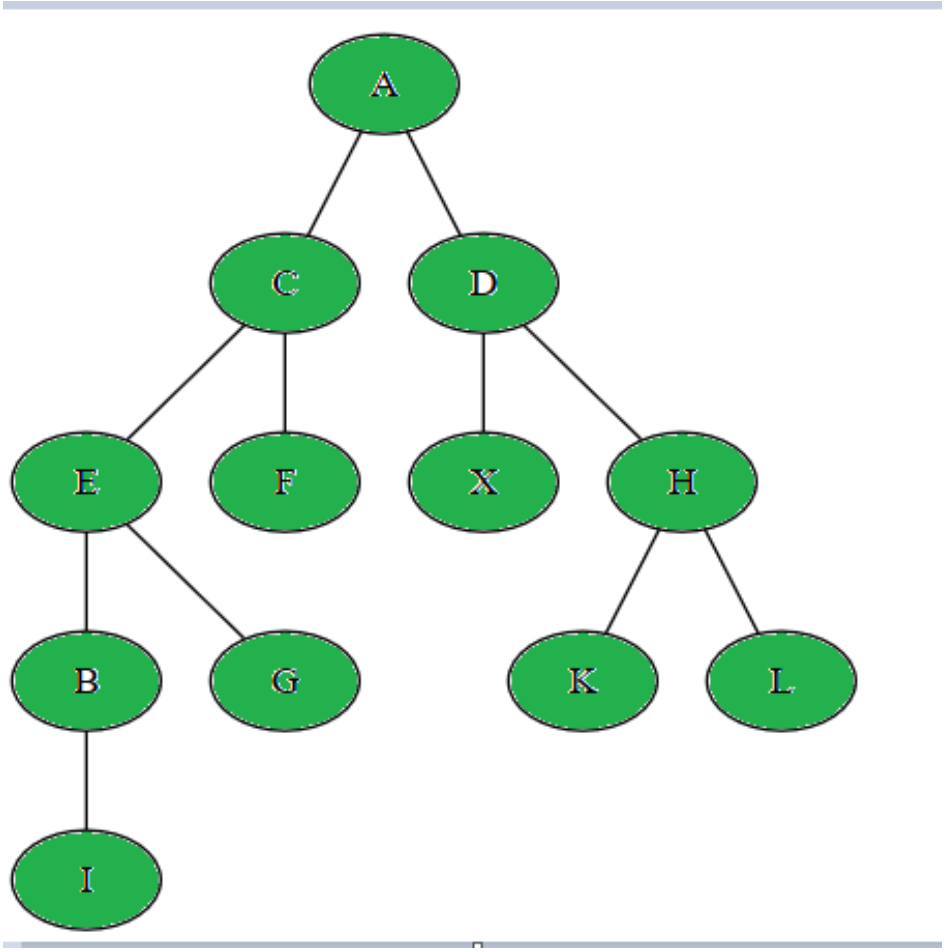
Parcurgere în lățime



Parcurgere în lățime



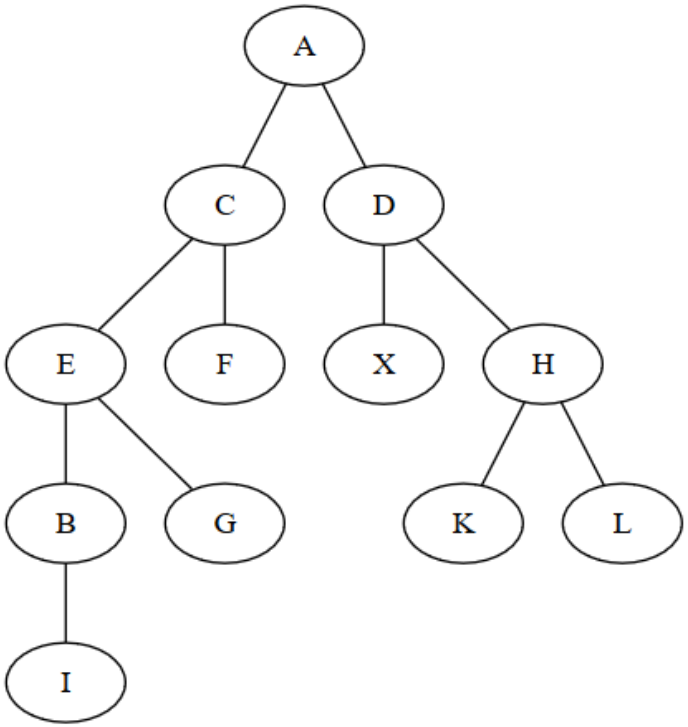
Parcurgere în lățime



Parcurgere în lățime

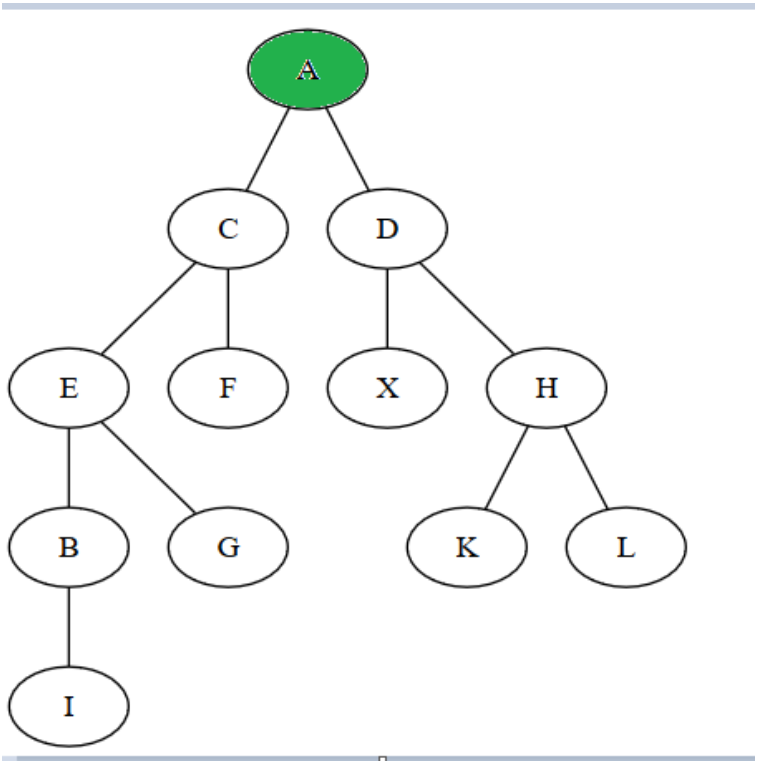
- Implementarea parcurgerii în lățime este identică cu implementarea parcurgerii în adâncime, doar se folosește o *coadă* în loc de stivă:
  - Pornim cu o coadă în care punem rădăcina arborelui.
  - Cât timp coada nu este vidă scoatem un element din coadă (vizităm nodul corespunzător)
  - Dacă nodul scos din coadă are descendenți, adăugăm descendenții lui în coadă.

Parcurgere în lățime



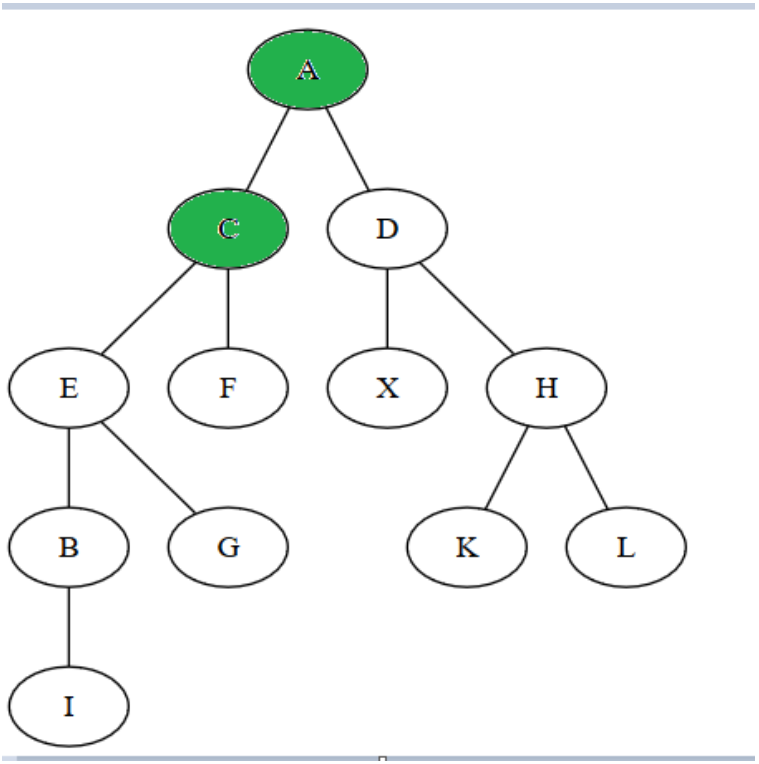
- Adăugăm în coadă rădăcina. Presupunem că un nod este *vizitat*, când este șters din coadă.
- Coada: A

Parcurgere în lățime



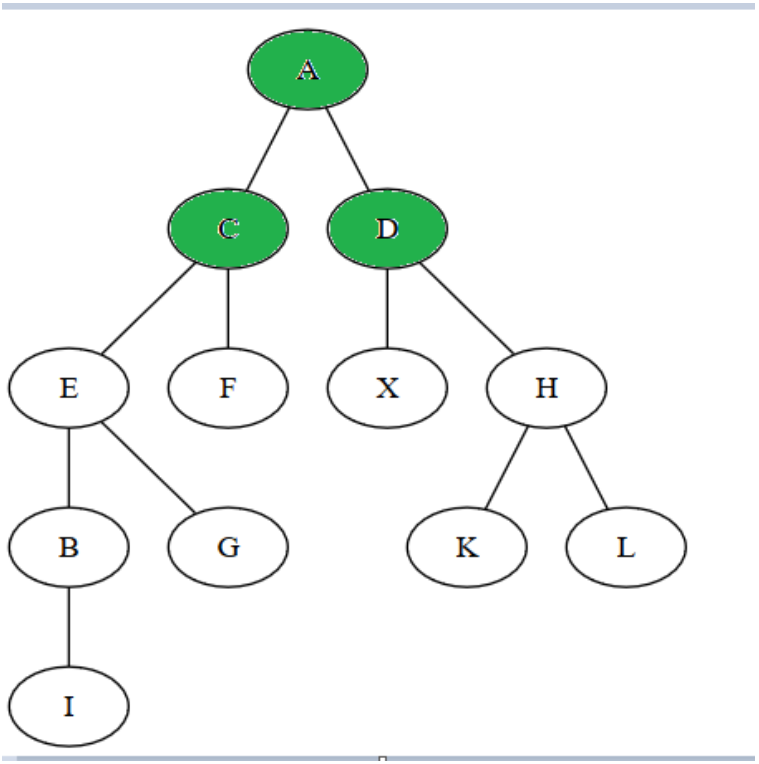
- Ștergem din coadă un element (A), îl vizităm, și adăugăm descendenții lui A în coadă.
- Coada: C, D

Parcurgere în lățime



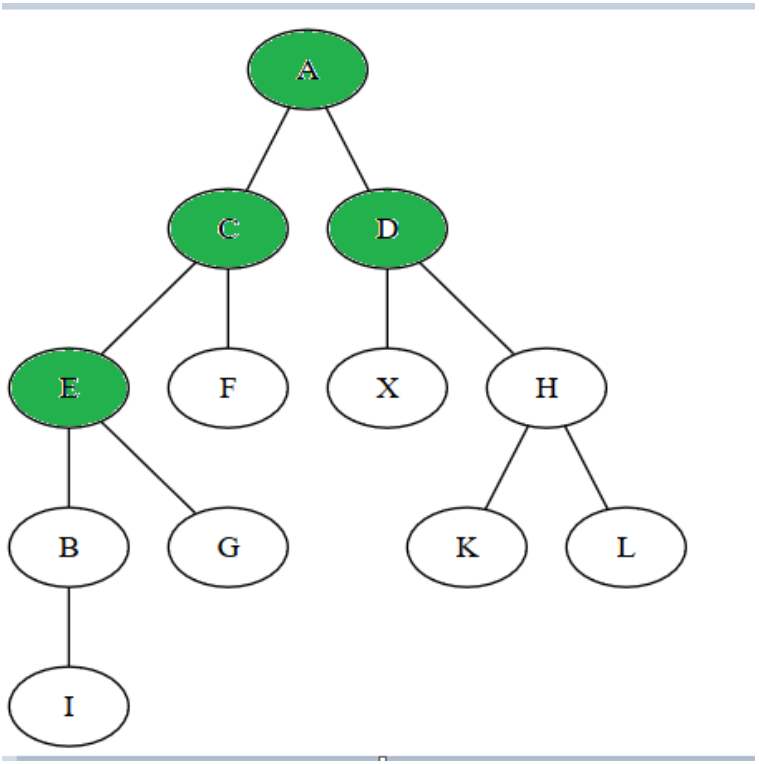
- Ștergem din coadă un element (C), îl vizităm, și adăugăm descendenții lui C în coadă
- Coada: D, E, F

Parcurgere în lățime



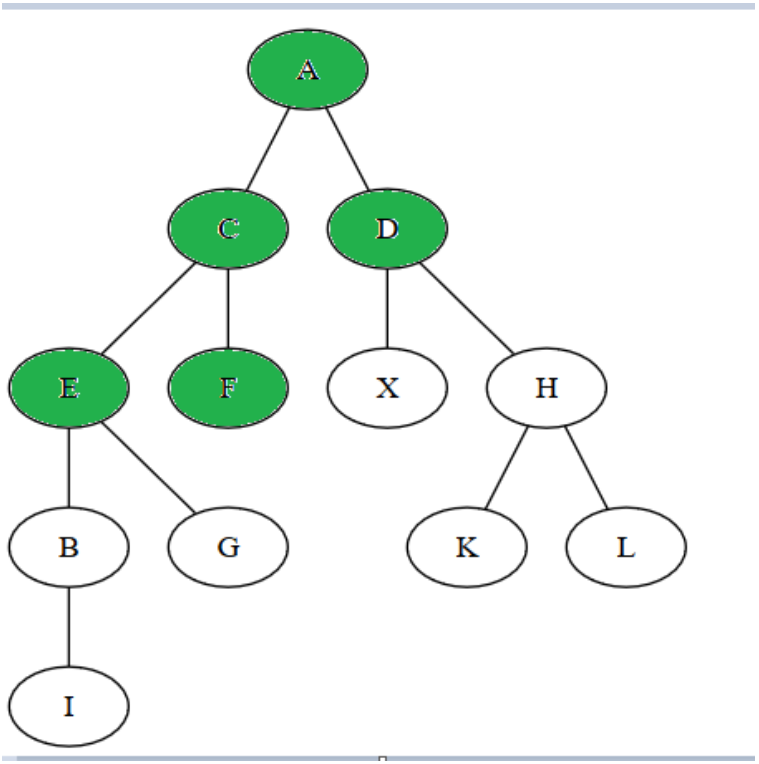
- Ștergem din coadă un element (D), îl vizităm, și adăugăm descendenții lui D în coadă
- Coada: E, F, X, H

Parcurgere în lățime



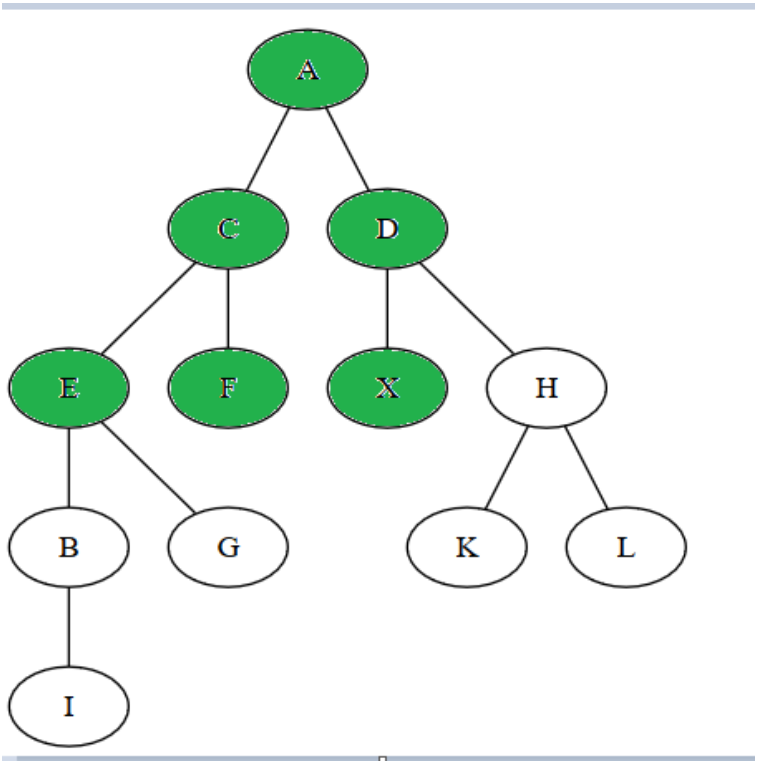
- Ștergem din coadă un element (E), îl vizităm, și adăugăm descendenții lui E în coadă
- Coada: F, X, H, B, G

Parcurgere în lățime



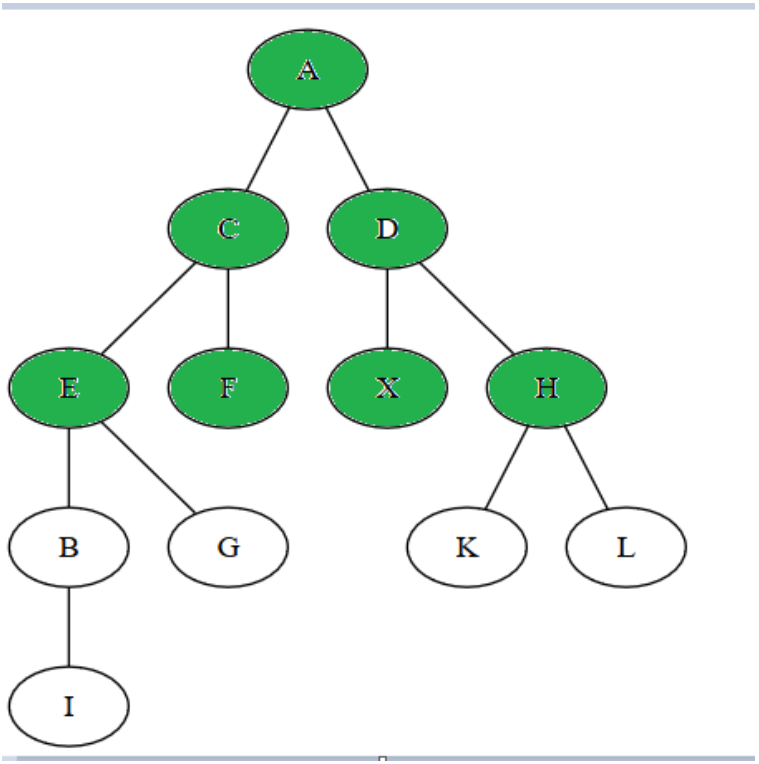
- Ștergem din coadă un element (F), îl vizităm, și nu adăugăm nimic în stivă, pentru că F nu are descendenți.
- Coada: X, H, B, G

Parcurgere în lățime



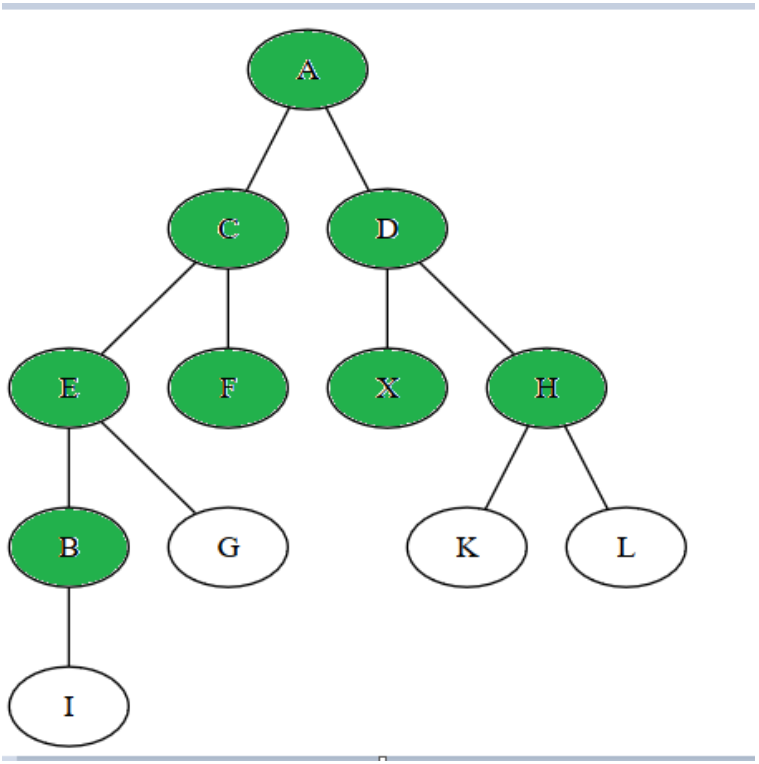
- Ștergem din coadă un element (X), îl vizităm, și nu adăugăm nimic în stivă, pentru că X nu are descendenți.
- Coada: H, B, G

Parcurgere în lățime



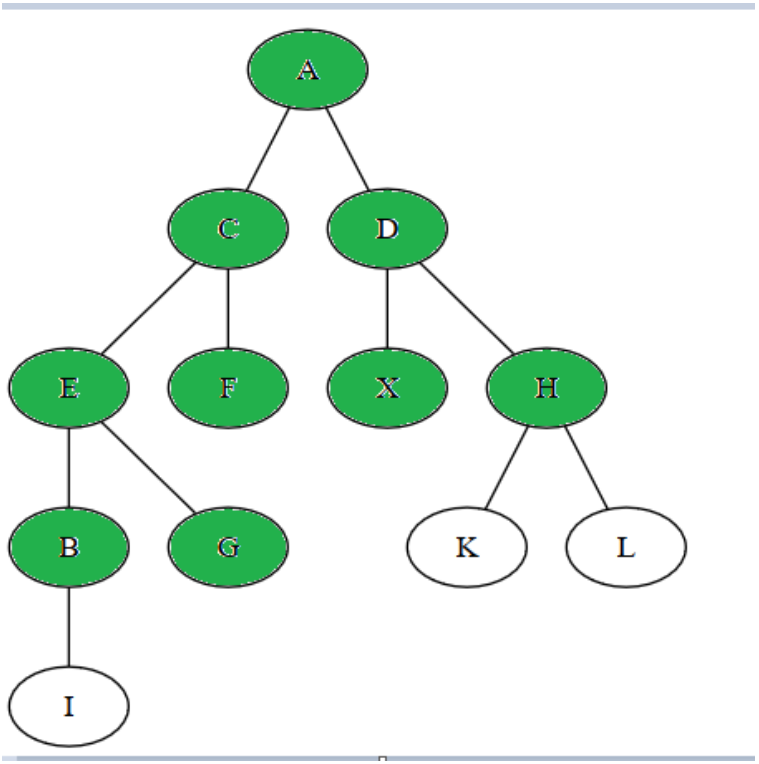
- Ștergem din coadă un element (H), îl vizităm, și adăugăm în coadă descendenții lui H.
- Coada: B, G, K, L

Parcurgere în lățime



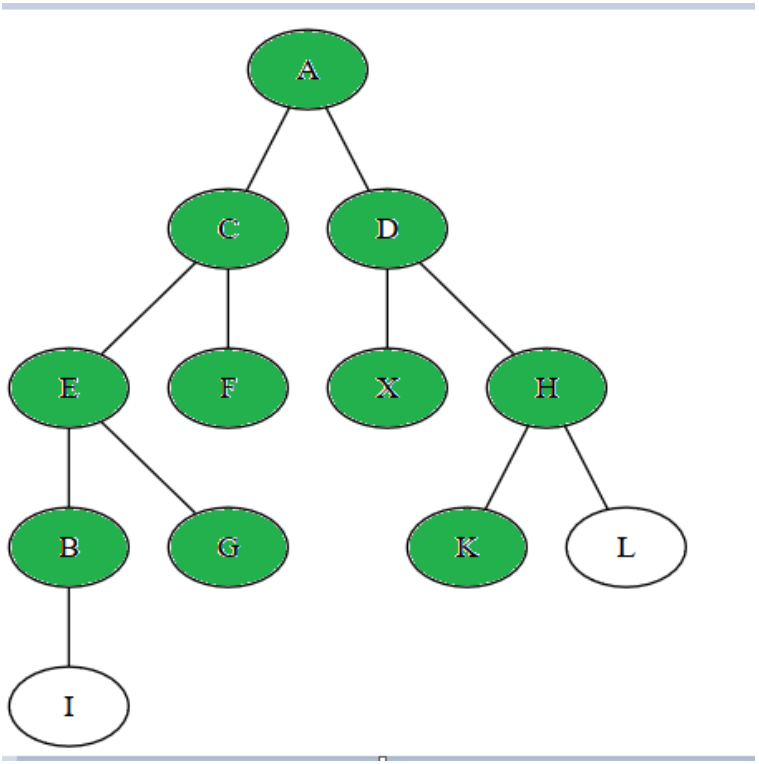
- Ștergem din coadă un element (B), îl vizităm, și adăugăm în coadă descendentul lui B
- Coada: G, K, L, I

Parcurgere în lățime



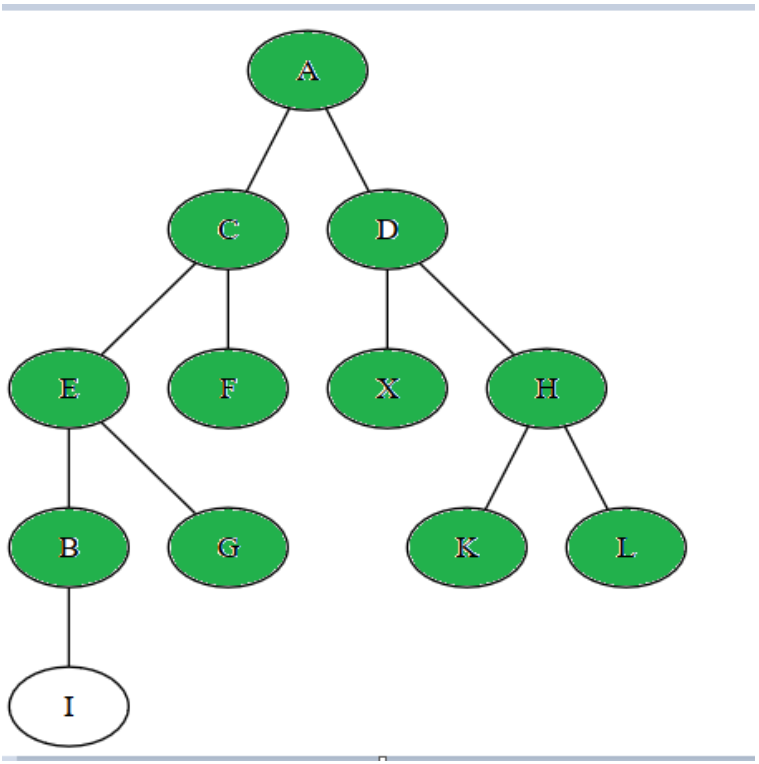
- Ștergem din coadă un element (G), îl vizităm, și nu adăugăm nimic în coadă, pentru că G nu are descendenți.
- Coada: K, L, I

Parcurgere în lățime



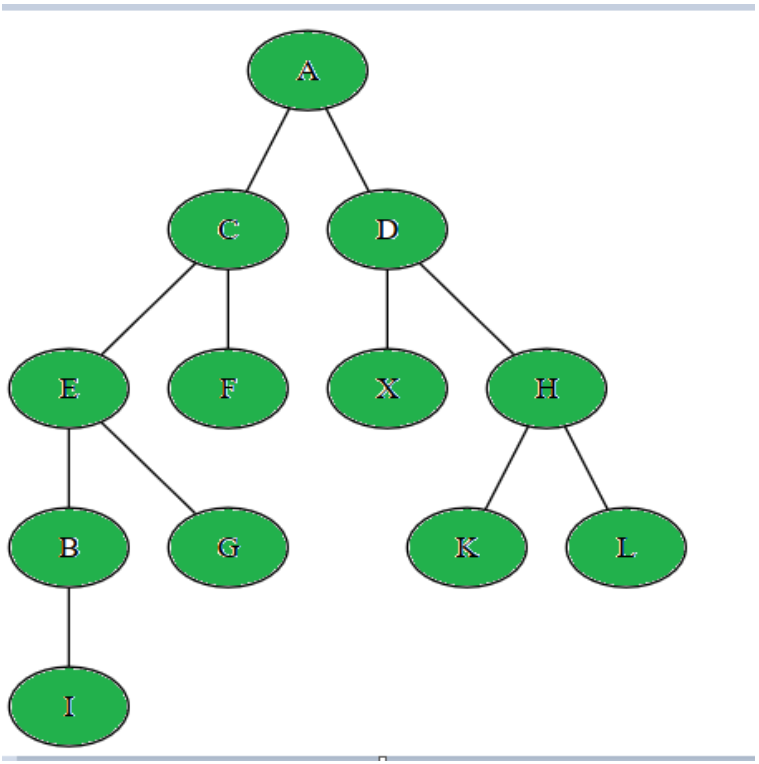
- Ștergem din coadă un element (K), îl vizităm, și nu adăugăm nimic în coadă
- Coada: L, I

Parcurgere în lățime



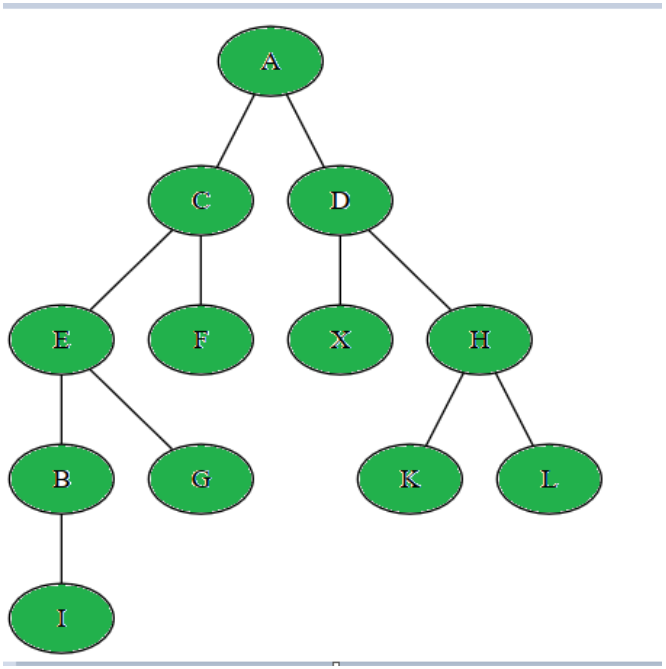
- Ștergem din coadă un element (L), îl vizităm, și nu adăugăm nimic în coadă, pentru că L nu are descendenți.
- Coada: I

Parcurgere în lățime



- Ștergem din coadă un element (I), îl vizităm, și nu adăugăm nimic în coadă, pentru că I nu are descendenți.
- Coada:

Parcurgere în lățime



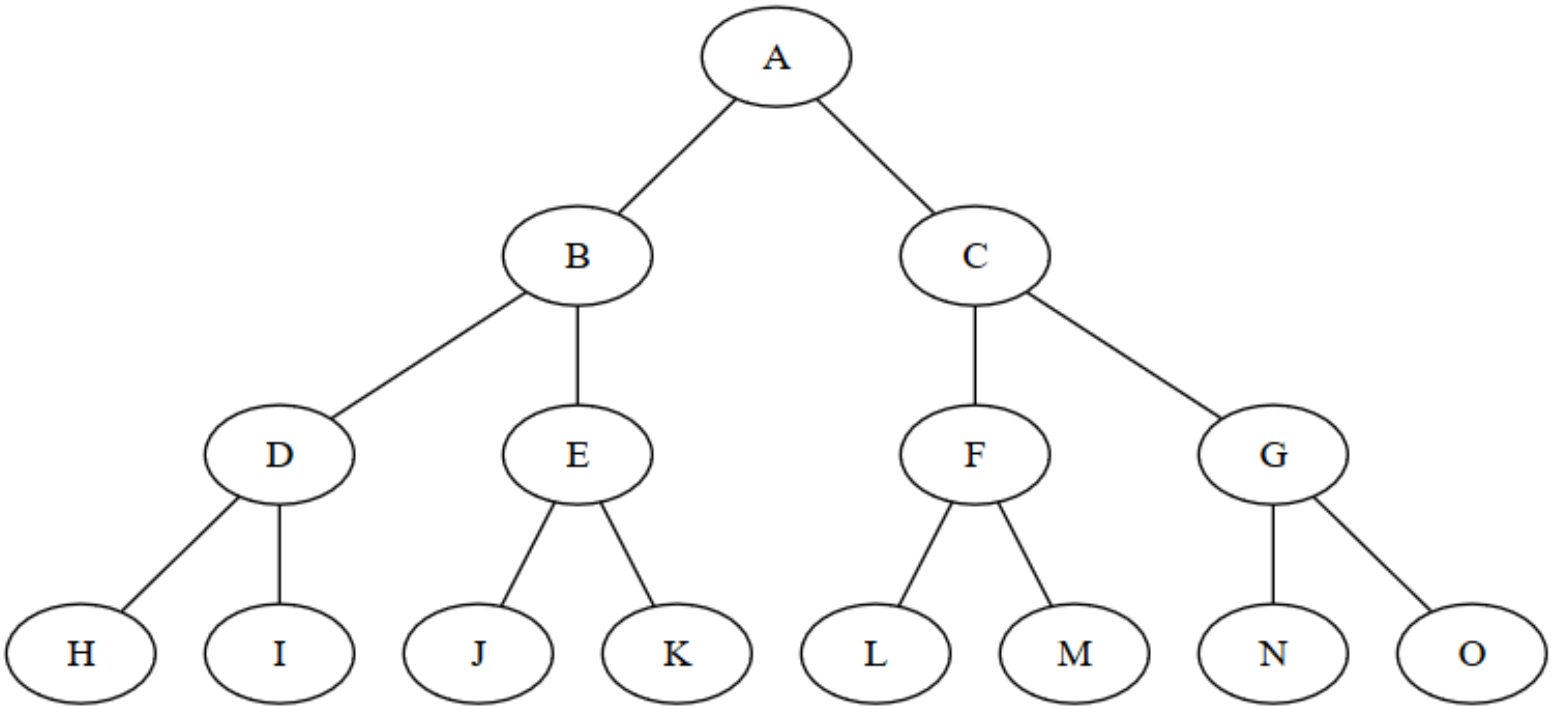
- Coada este vidă, deci am terminat de parcurs toate nodurile.
- Ordinea în care am parcurs nodurile este: A, C, D, E, F, X, H, B, G, K, L, I

Arbori binari

- Un arbore în care fiecare nod are maxim 2 descendenți, se numește **arbore binar**.
- Pentru un arbore binar avem nume specială pentru descendenți, există descendent/fiu *stâng* și descendent/fiu *drept*.
- Dacă un nod are un singur descendent, trebuie specificat dacă e fiul stâng sau cel drept.

Arbori binari - Terminologie

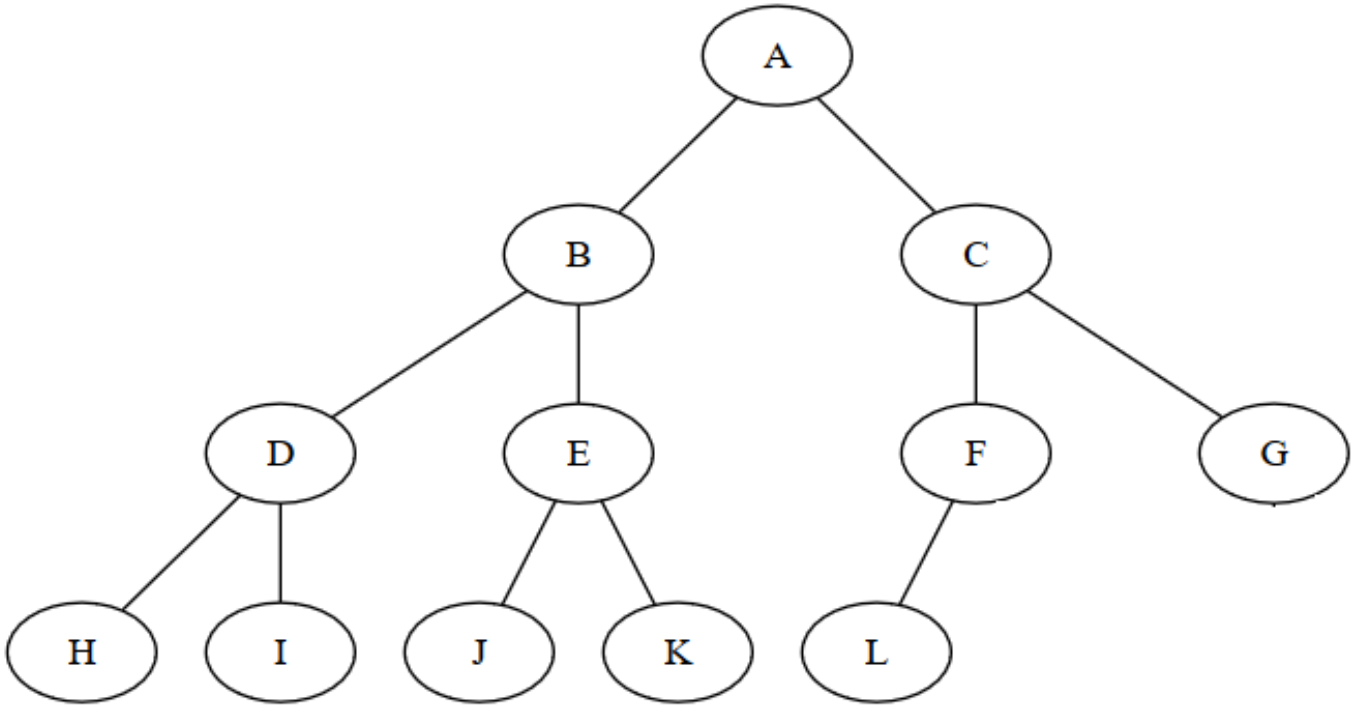
- Un arbore binar se numește *plin* dacă pe fiecare nivel are numărul maxim de noduri posibile.



- Un arbore binar plin cu k niveluri, are $2^k - 1$ noduri.

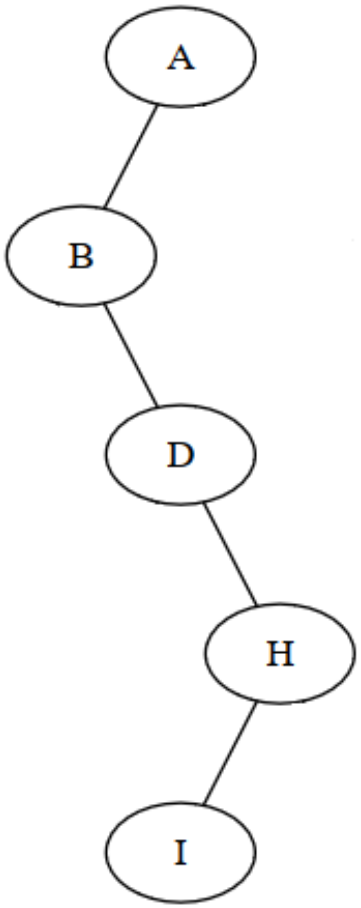
Arbori binari - Terminologie

- Un arbore binar se numește *complet* dacă poate fi obținut dintr-un arbore binar plin, eliminând niște noduri de la ultimul nivel, de la dreapta spre stânga (să avem structură de ansamblu).



Arbori binari - Terminologie

- Un arbore binar se numește *degenerat* dacă are n noduri pe n niveluri (și înălțime $n - 1$).



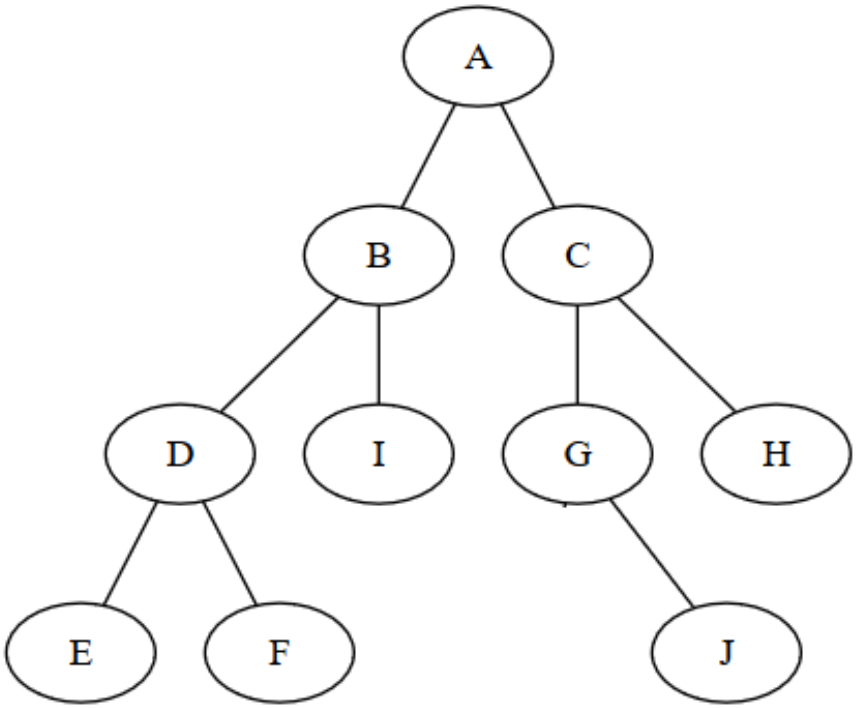
Arbori binari - Parcurgere

- Pe lângă cele 2 parcurgeri discutate pentru un arbore n-ar (DFS și BFS), la un arbore binar mai există 3 parcurgeri:
  - Parcurgere în *preordine*
  - Parcurgere în *inordine*
  - Parcurgere în *postordine*

Parcurgere în preordine I

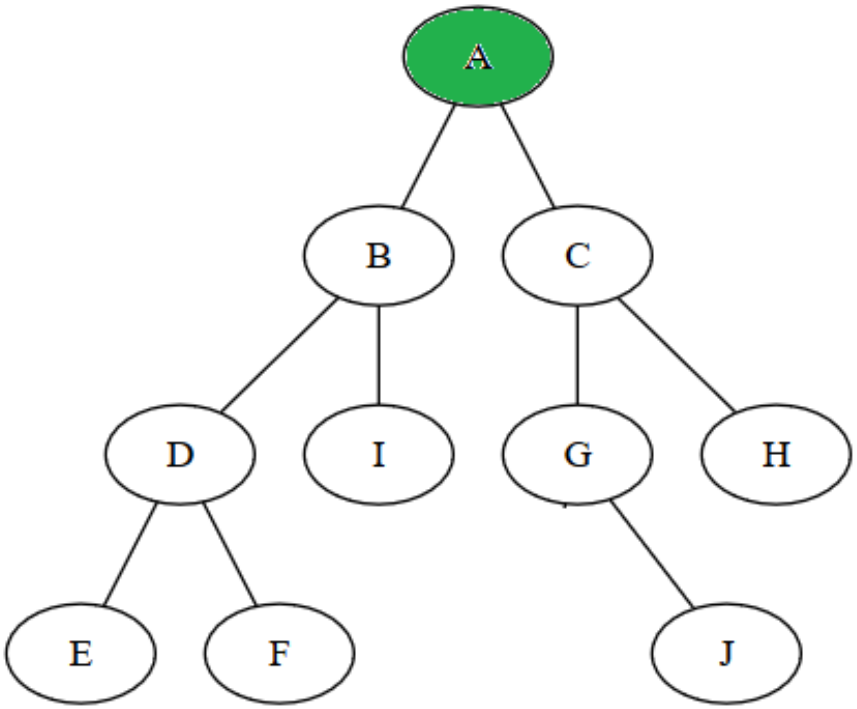
- La parcurgere în *preordine* tot timpul se vizitează prima dată părintele, după care subarborele stâng, după care subarborele drept. Când vizităm subarbori, tot în preordine le vizităm.
- Să vedem un exemplu concret

Parcurgere în preordine II



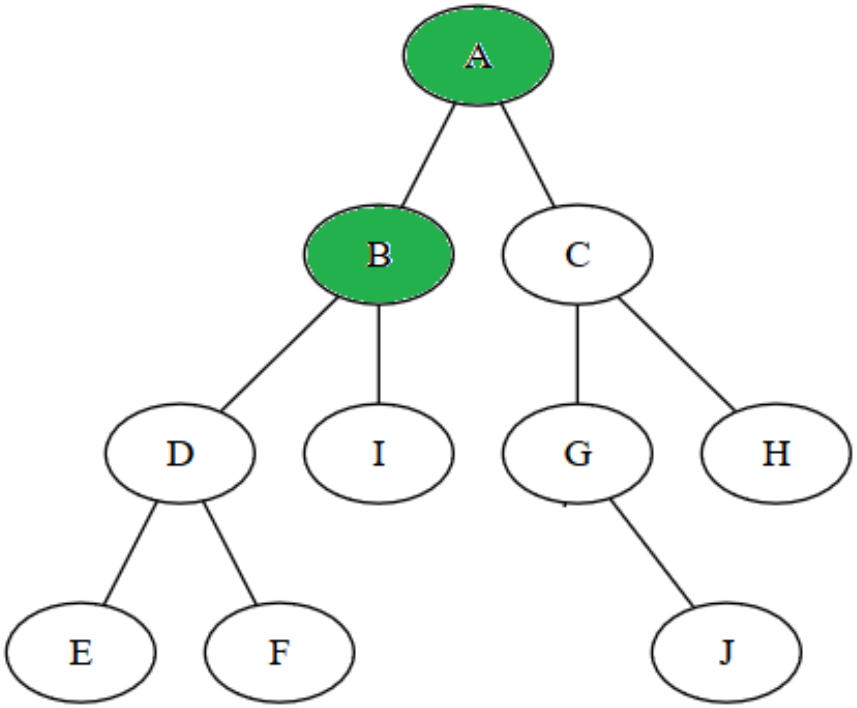
Parcurgere în preordine III

- Începem cu rădăcina (după care vizităm subarborele stâng - tot în preordine - și subarborele drept - tot în preordine).



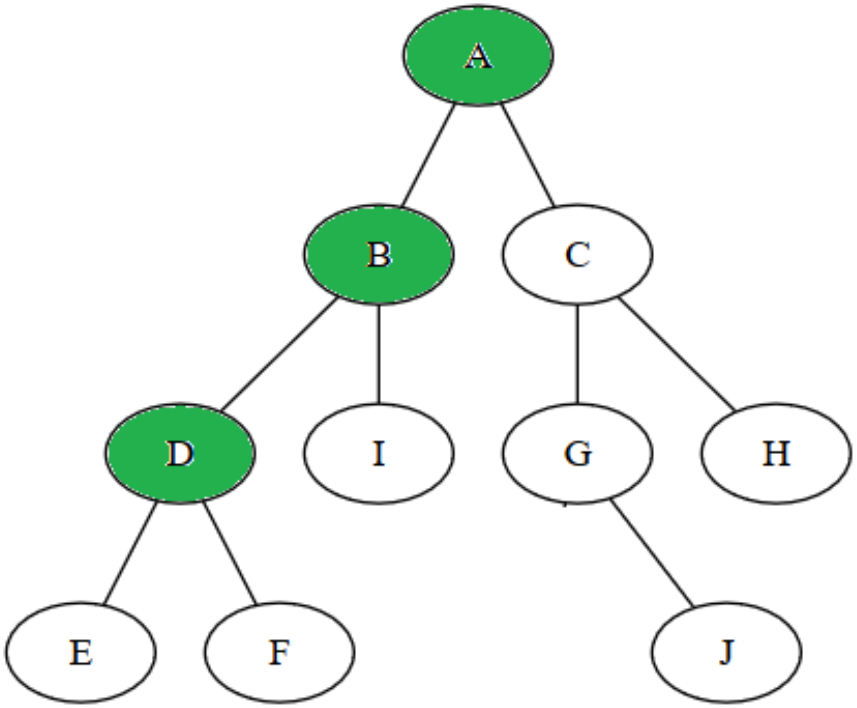
Parcurgere în preordine IV

- Continuăm cu subarborele stâng (aici iar vizităm rădăcina, subarbore stâng și subarbore drept).



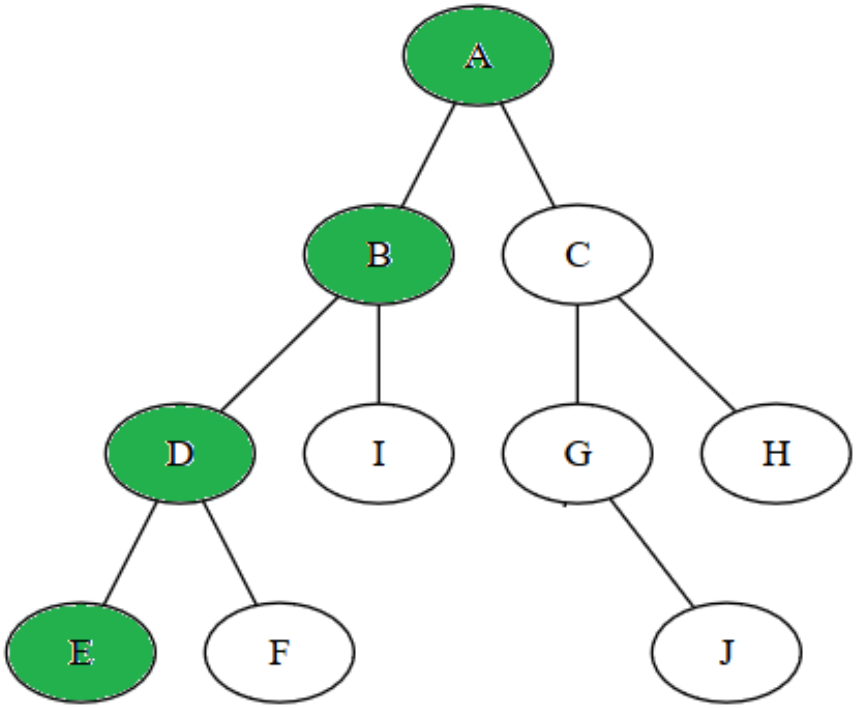
Parcurgere în preordine V

- Continuăm cu subarbore stâng al lui B (rădăcina, subarbore stâng, subarbore drept)



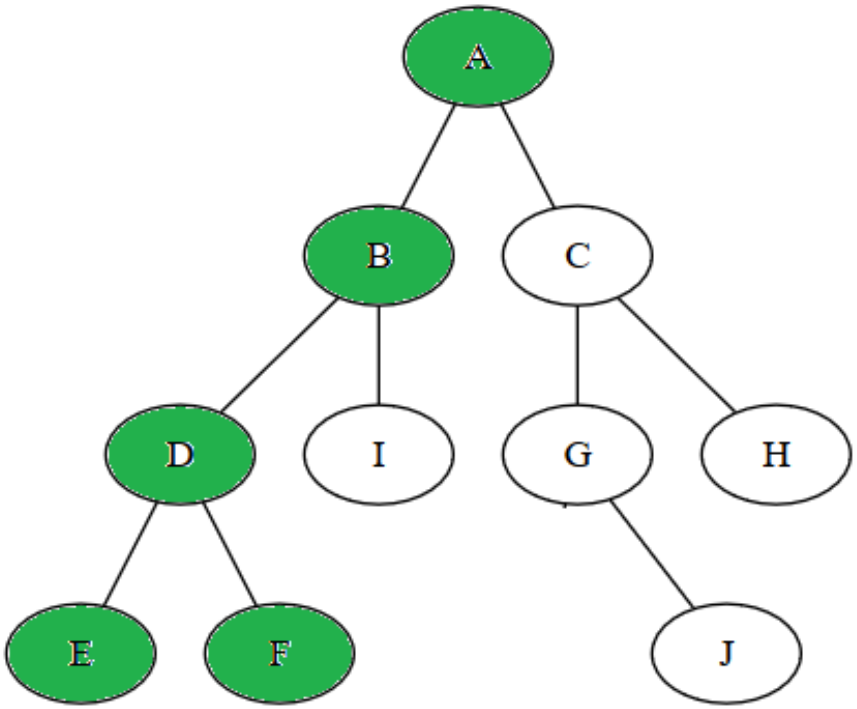
Parcurgere în preordine VI

- Continuăm cu subarborele stâng al lui D (rădăcina, subarbore stâng, subarbore drept)



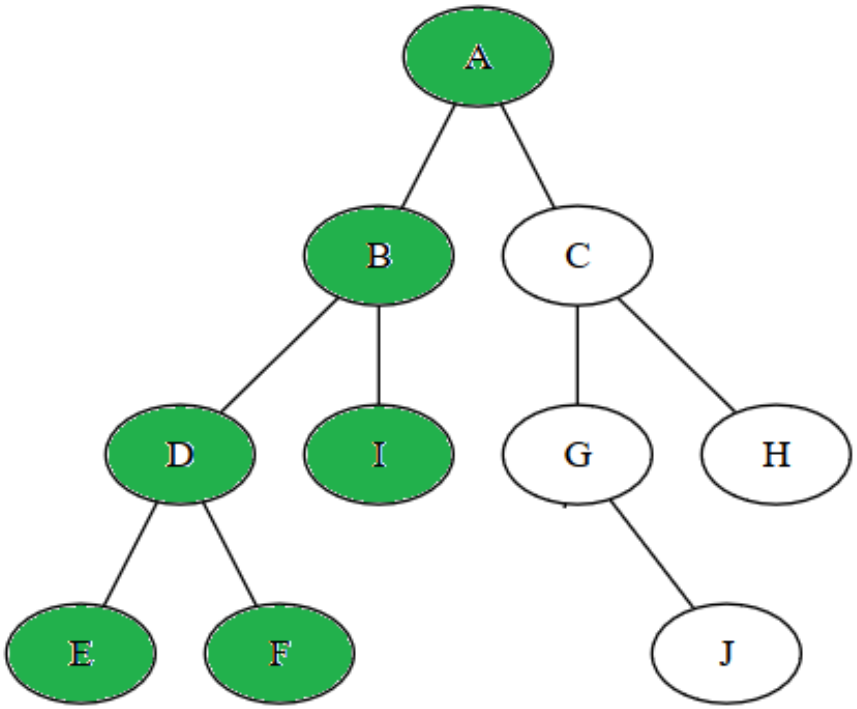
Parcurgere în preordine VII

- E nu are descendenți, mergem înapoi la D și vizităm subarborele drept.



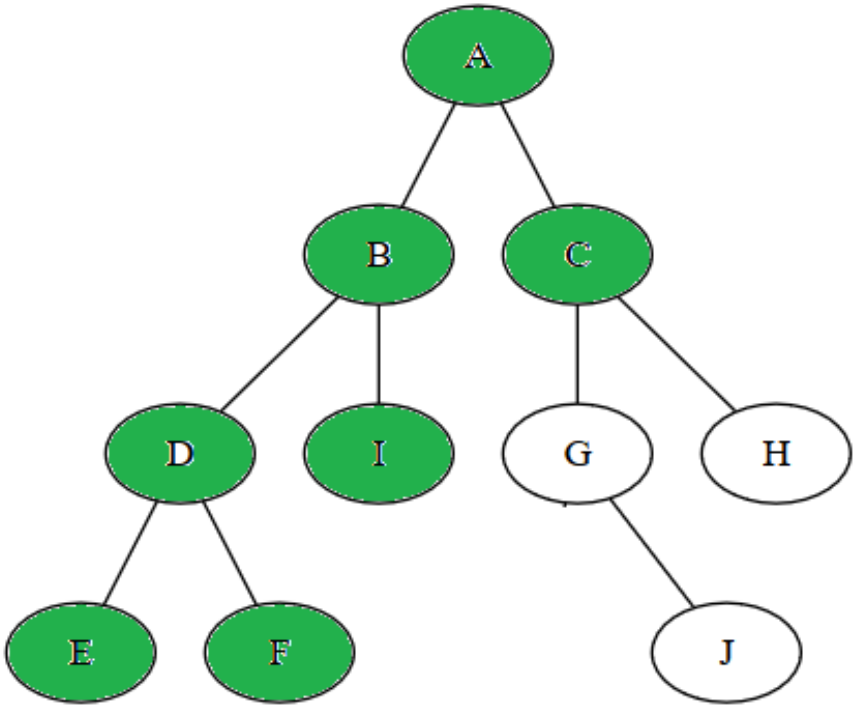
Parcurgere în preordine VIII

- F nu are descendenți, mergem înapoi la B, și vizităm subarborele drept.



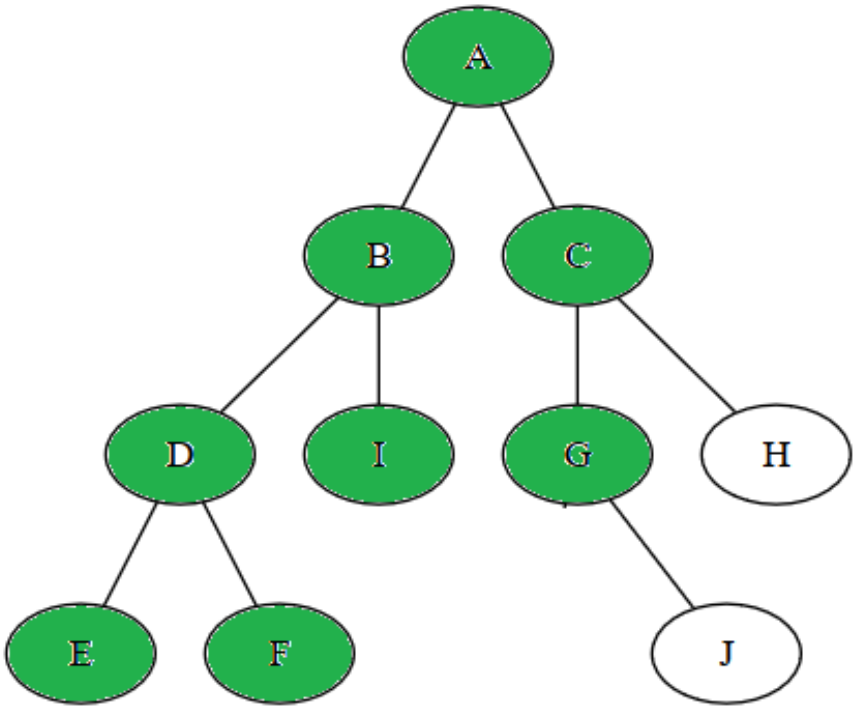
Parcurgere în preordine IX

- I nu are descendenți, urcăm la A și vizităm subarborele drept (rădăcina, subarbore stâng, subarbore drept).



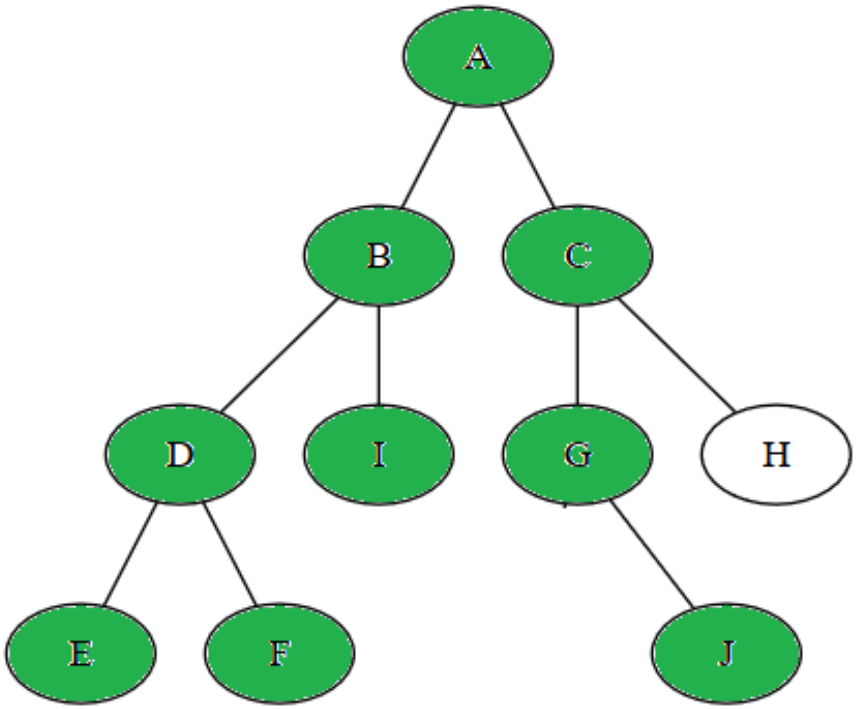
Parcurgere în preordine X

- Mergem pe subarbore stâng al lui C.



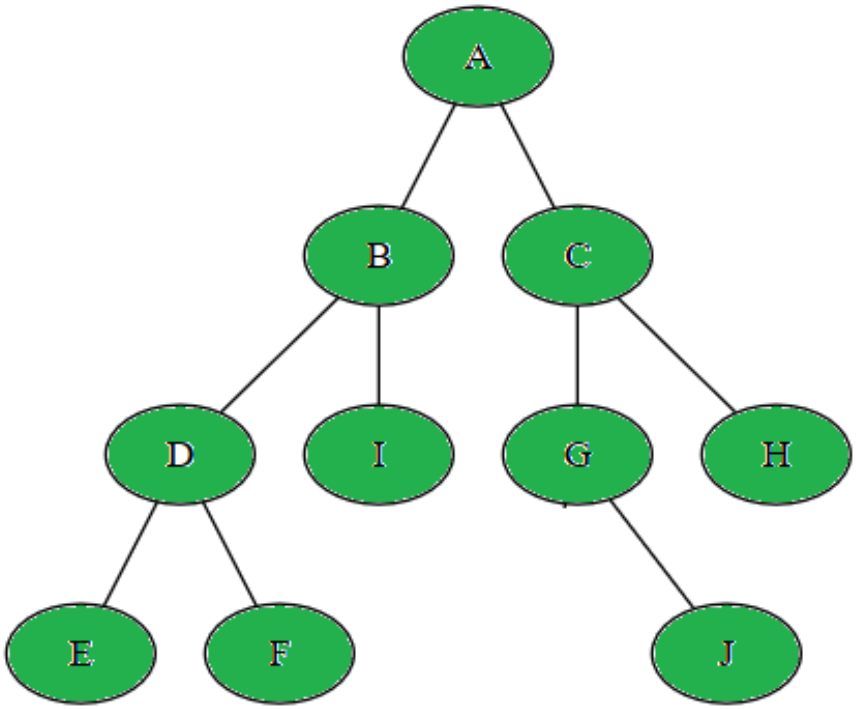
Parcurgere în preordine XI

- G nu are subarbore stâng, mergem pe cel drept.



Parcurgere în preordine XII

- J nu are descendenți, urcăm la C și vizităm subarborele drept.



Parcurgere în preordine XIII

- Acum am vizitat toate nodurile. Ordinea în care ele au fost vizitate: A, B, D, E, F, I, C, G, J, H (aceeași ordine ca la DFS, pentru că DFS e un fel de parcurgere preordine).

Parcurgere în preordine XIV

- Subalgoritmul de parcurgere poate fi implementat cel mai ușor în mod recursiv. Pentru implementare ne trebuie o reprezentare pentru un arbore. Arborele fiind alcătuit din noduri, ne trebuie structura nod:

Nod:
info: TElem
stang: ↑ Nod
drept: ↑ Nod

- Iar arborele binar nu are altceva decât o rădăcină

ArboreBinar:
rad: ↑ Nod

Parcurgere în preordine XV

```
subalgoritm parcurgePreordine() este
    preordineRecursiv(this.rad)
sf_subalgoritm
```

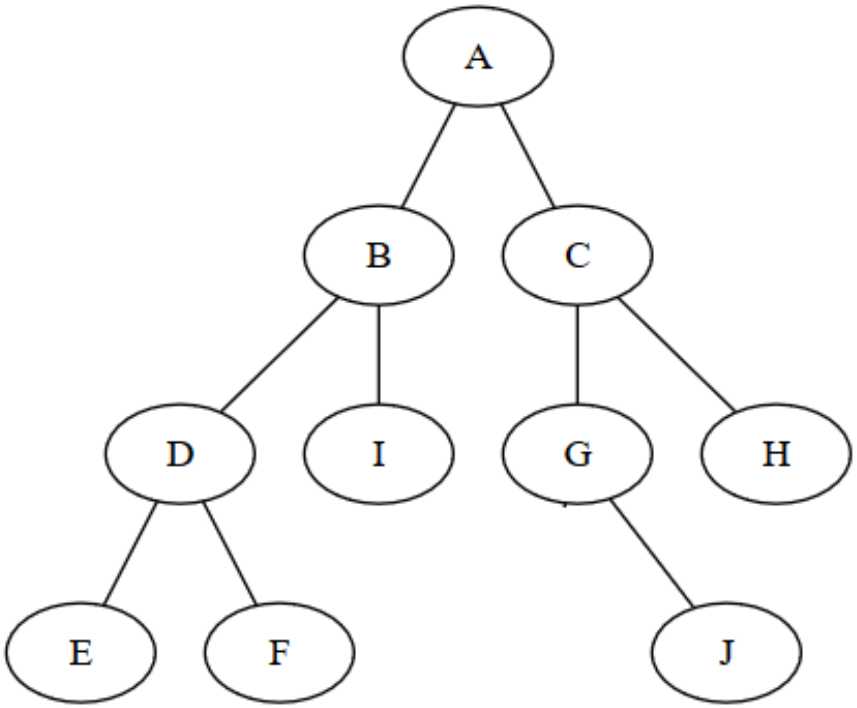
Parcurgere în preordine XVI

```
subalgoritm preordineRecursiv (nod: ↑ Nod) este
//nod este un nod. Poate fi NIL
   dacă nod ≠ NIL atunci
      scrie [nod].info
      preordineRecursiv([nod].stang)
      preordineRecursiv([nod].drept)
   sf_dacă
sf_subalgoritm
```

Parcurgere în inordine I

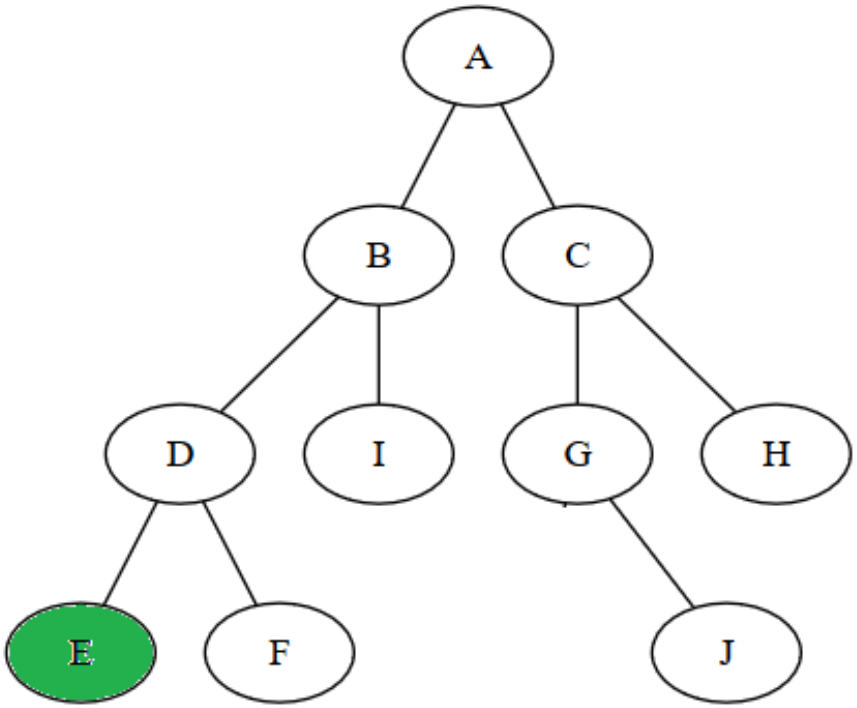
- La parcurgere în *inordine* tot timpul se vizitează prima dată subarborele stâng, urmat de părintele, urmat de subarborele drept. Când vizităm subarbori, tot în inordine le vizităm.
- Să vedem un exemplu concret:

Parcurgere în inordine II



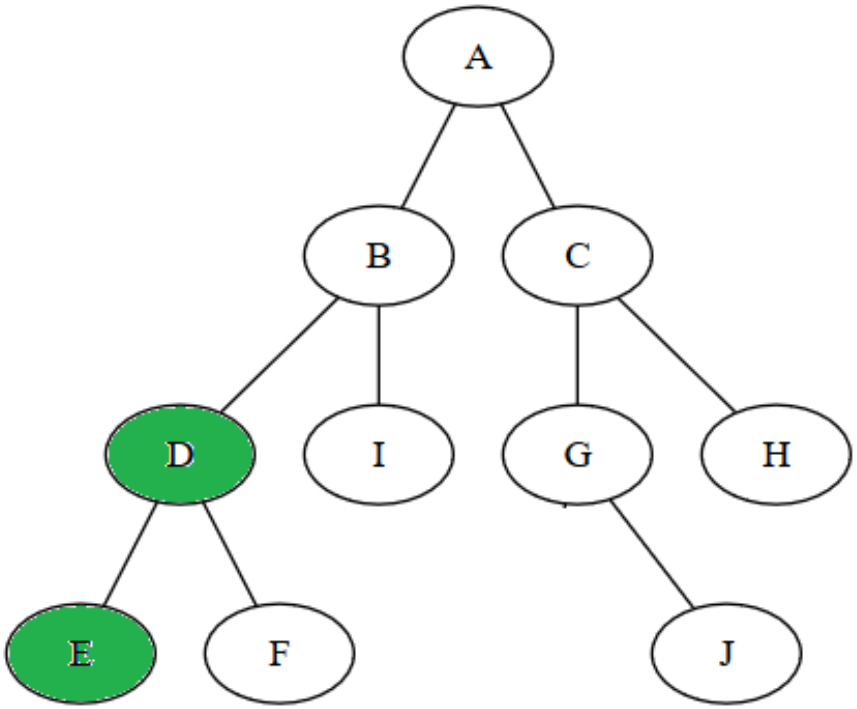
Parcurgere în inordine III

- Începem cu subarbore stâng de la rădăcina. Subarborele tot în inordine se vizitează, deci încep cu subarborele stâng și așa mai departe, până găsesc un nod care nu are subarbore stâng.



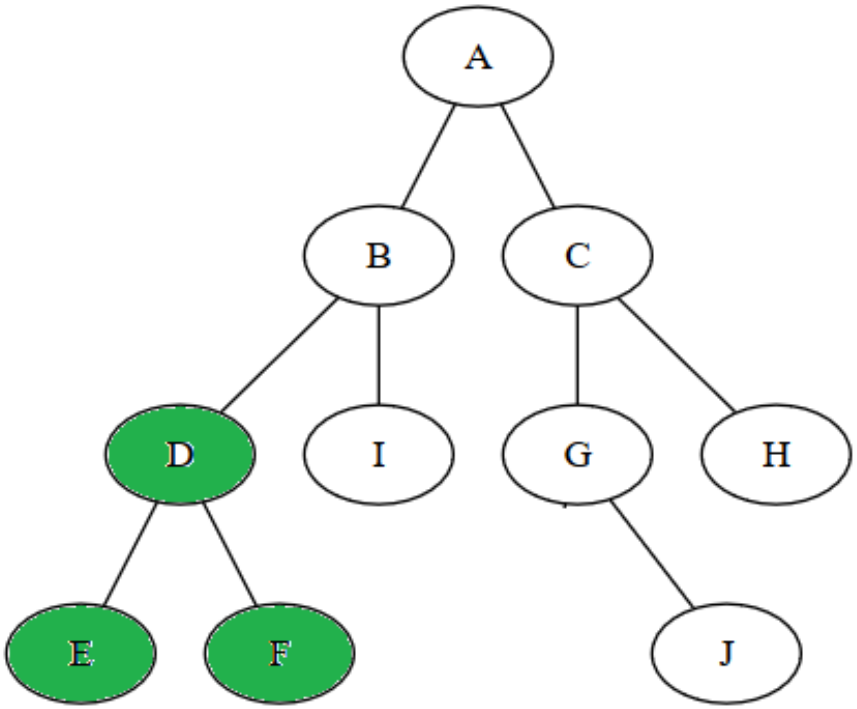
Parcurgere în inordine IV

- După subarbore stâng vizitat, vizităm rădăcina.



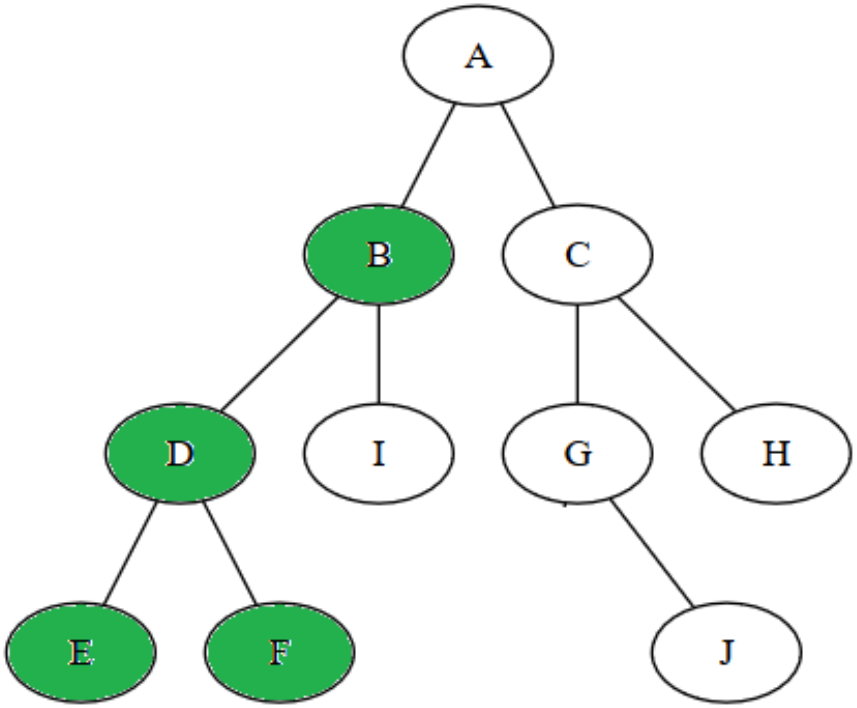
Parcurgere în inordine V

- Continuăm cu subarbore drept al lui D.



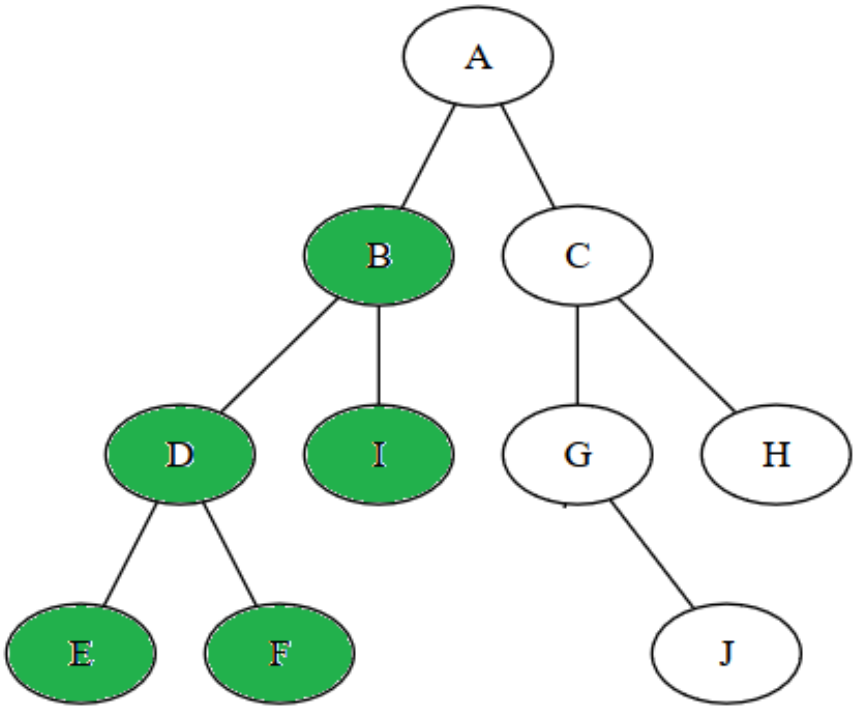
Parcurgere în inordine VI

- Continuăm cu B (care are subarborele stâng vizitat deja).



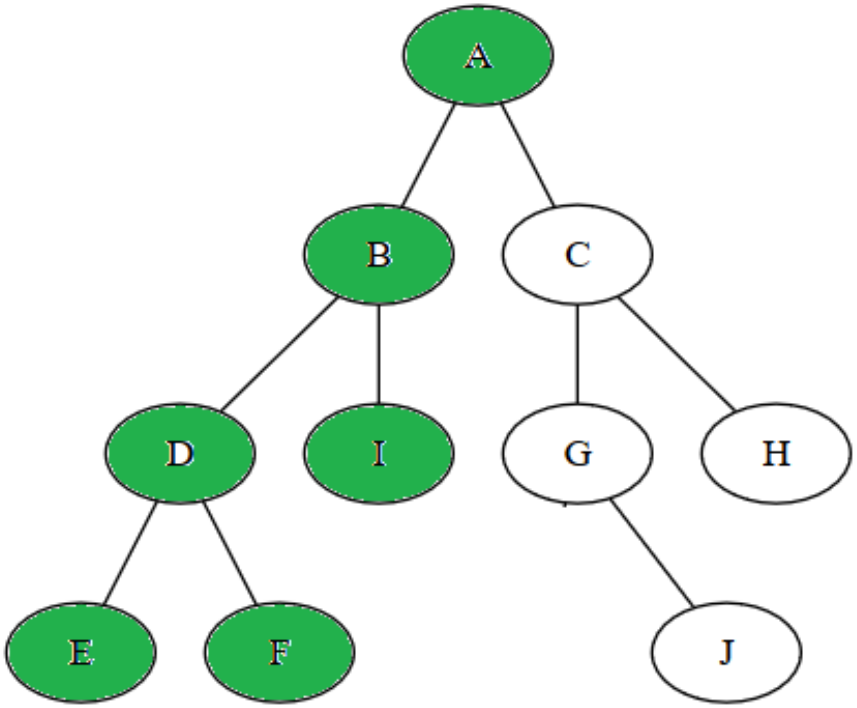
Parcurgere în inordine VII

- Mergem pe subarborele drept de la B.



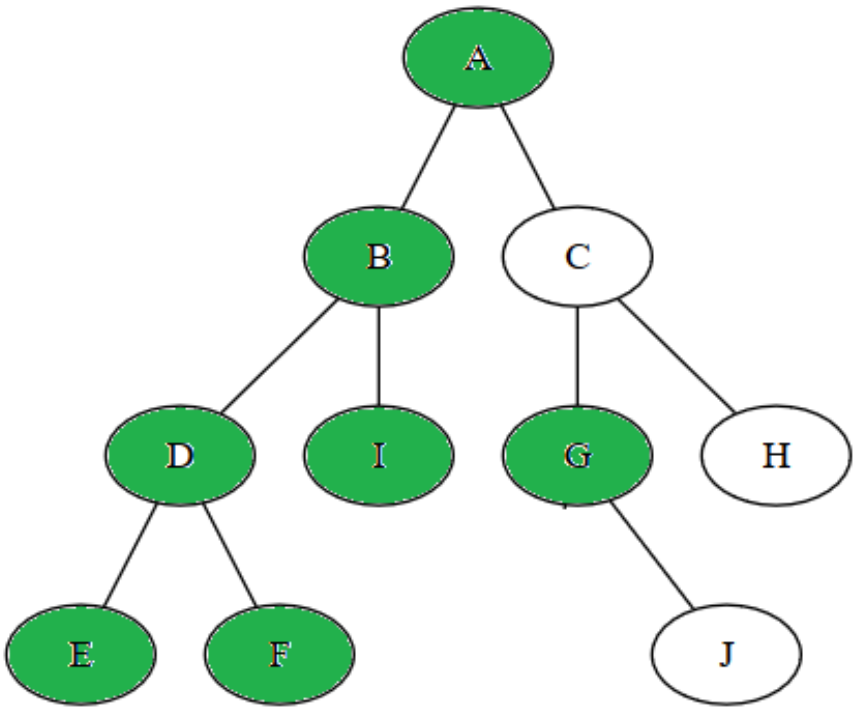
Parcurgere în inordine VIII

- Rădăcina A are deja subarborele stang vizitat, acum vizităm A-ul.



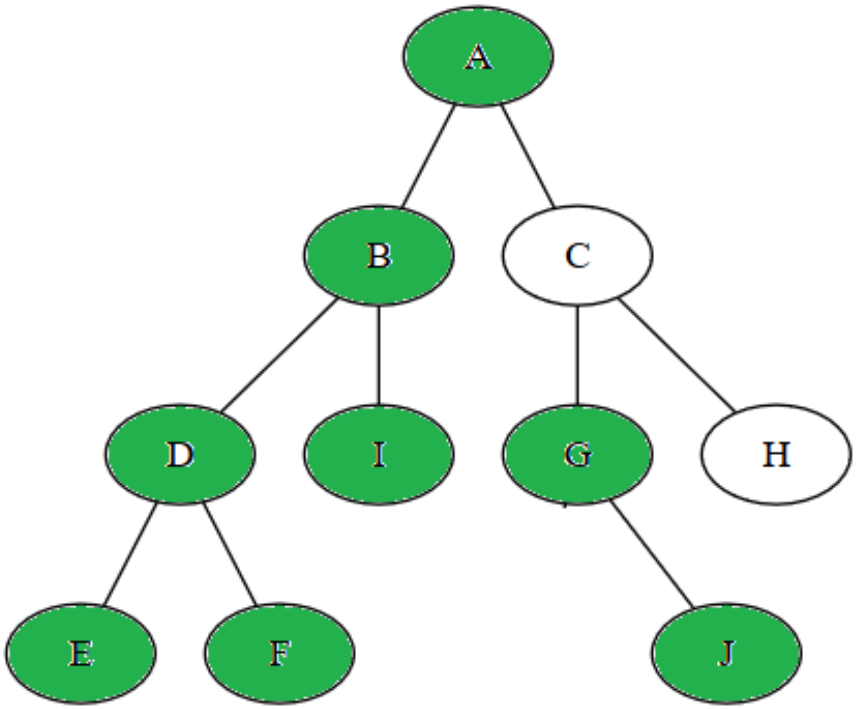
Parcurgere în inordine IX

- Mergem pe subarbore drept de la A. Aici iar ne trebuie subarbore stâng prima dată, și cum G nu are subarbore stâng, G va fi primul nod vizitat.



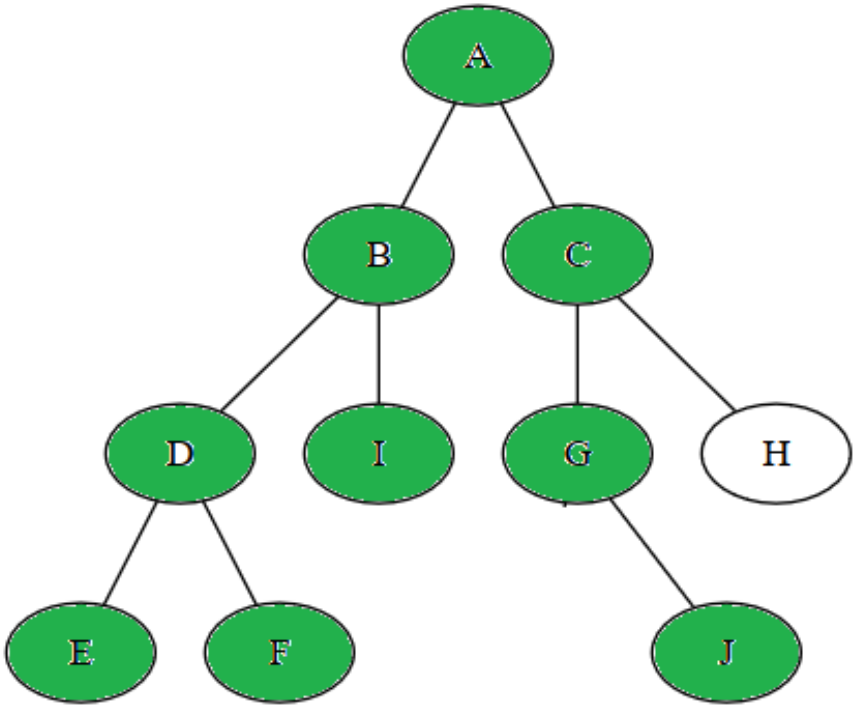
Parcurgere în inordine X

- Mergem pe subarbore drept al lui G.



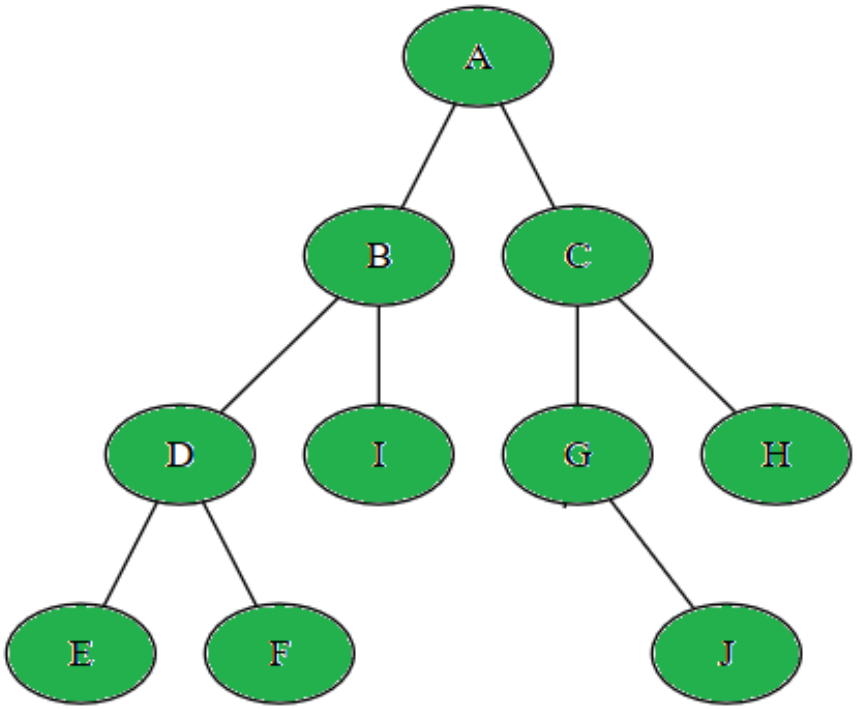
Parcurgere în inordine XI

- Acum putem vizita C (avem subarborele stâng vizitat deja)



Parcurgere în inordine XII

- La final mai vizităm subarbore drept de la C, H.



Parcurgere în inordine XIII

- Acum am vizitat toate nodurile. Ordinea în care ele au fost vizitate: E, D, F, B, I, A, G, J, C, H
- Implementarea parcurgerii se face similar cu preordine, și folosim aceeași reprezentare.

Parcurgere în inordine XIV

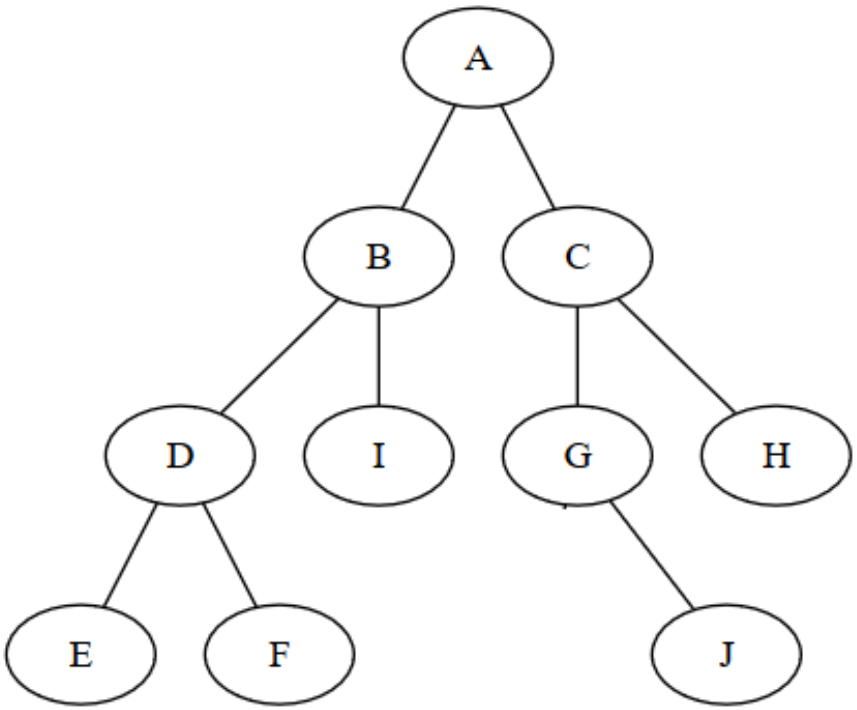
subalgoritm parcurgeInordine() **este**
    inordineRecursiv(this.rad)
sf_subalgoritm

Parcurgere în inordine XV

```
subalgoritm inordineRecursiv (nod: ↑ Nod) este
//nod este un nod. Poate fi NIL
   dacă nod ≠ NIL atunci
      inordineRecursiv([nod].stang)
      scrie [nod].info
      inordineRecursiv([nod].drept)
   sf_dacă
sf_subalgoritm
```

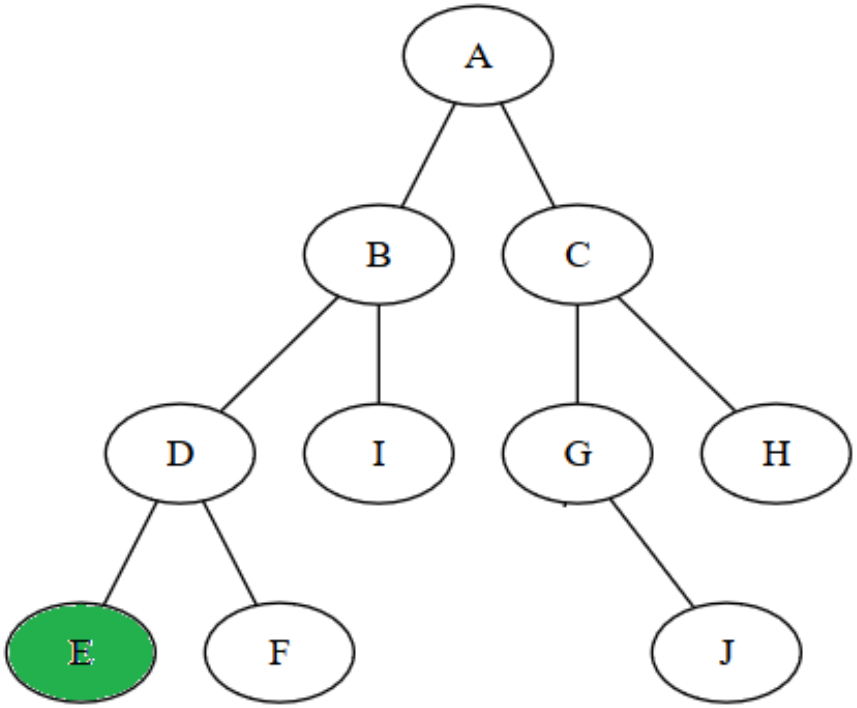
Parcurgere în postordine I

- La parcurgere în *postordine* tot timpul se vizitează prima dată subarbore stâng, urmat de subarbore drept, și la final rădăcina.
- Să vedem un exemplu concret



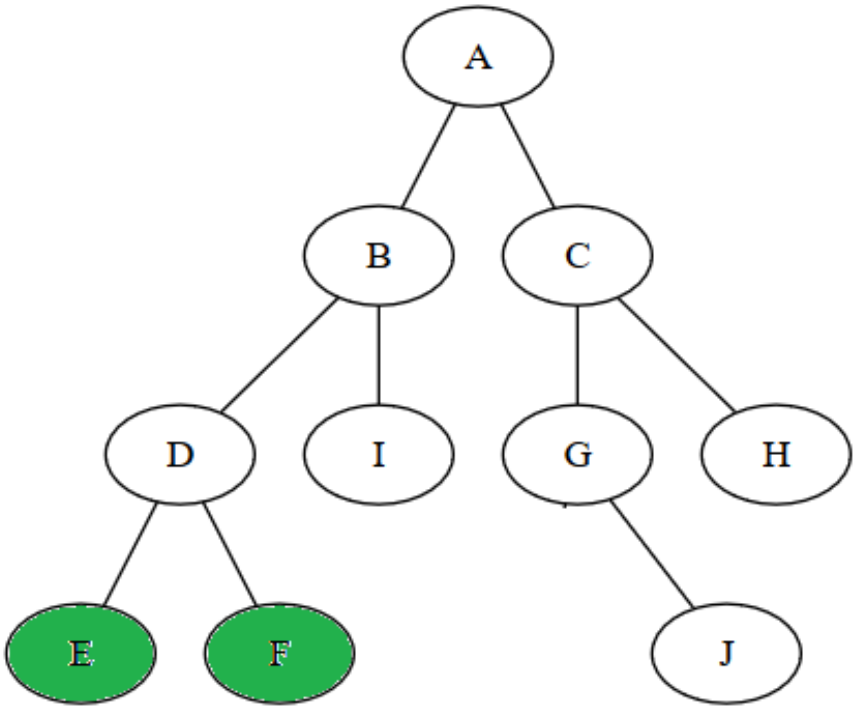
Parcurgere în postordine II

- Începem cu subarbore stâng și prima dată vizităm nodul E.



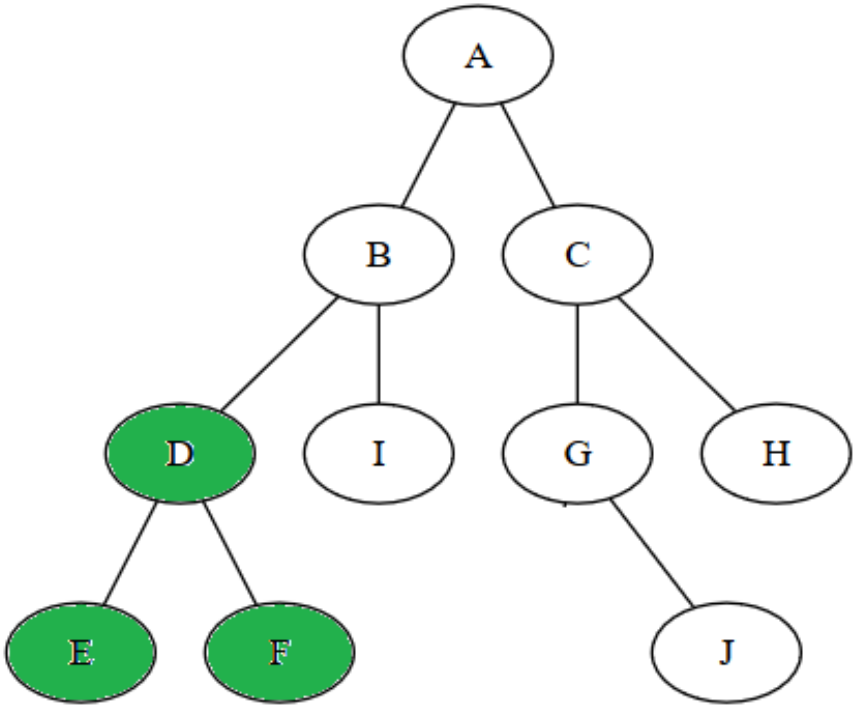
Parcurgere în postordine III

- Continuăm cu subarborele drept al rădăcinii lui E, adică D.



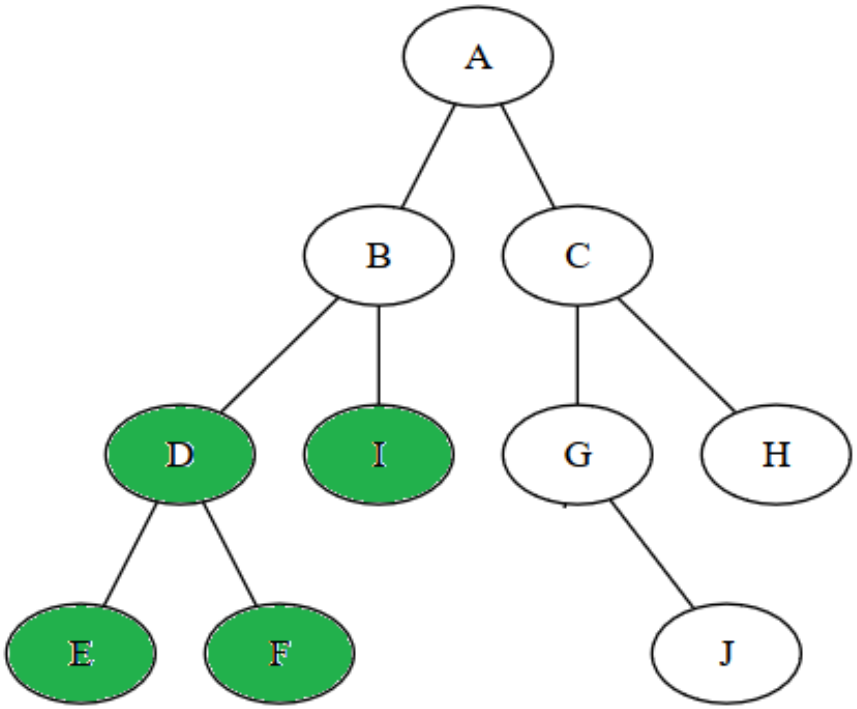
Parcurgere în postordine IV

- După ce am vizitat subarborele drept și stâng vizităm rădăcina.



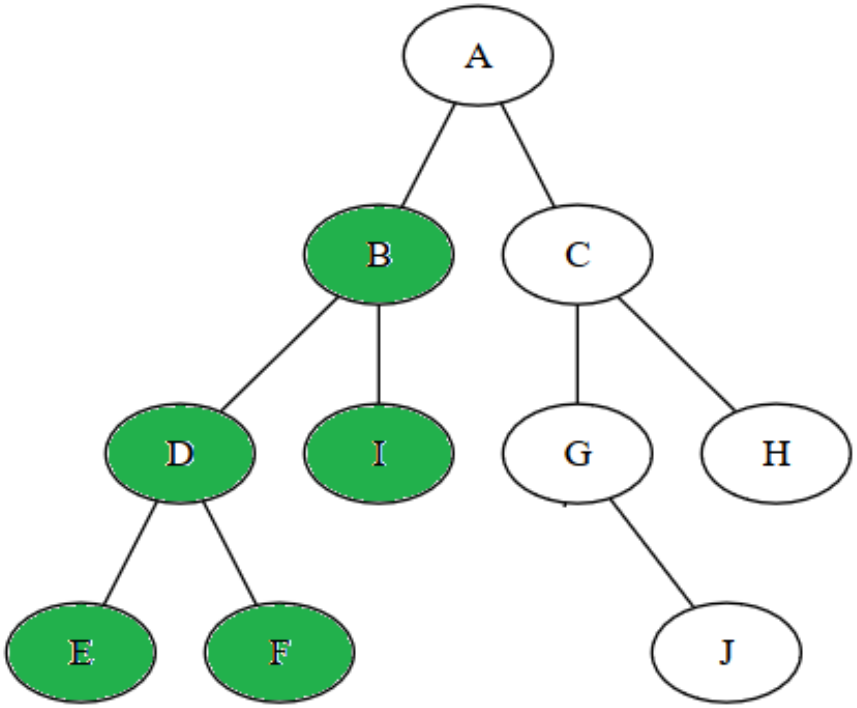
Parcurgere în postordine V

- Continuăm cu subarborele stâng al lui B.



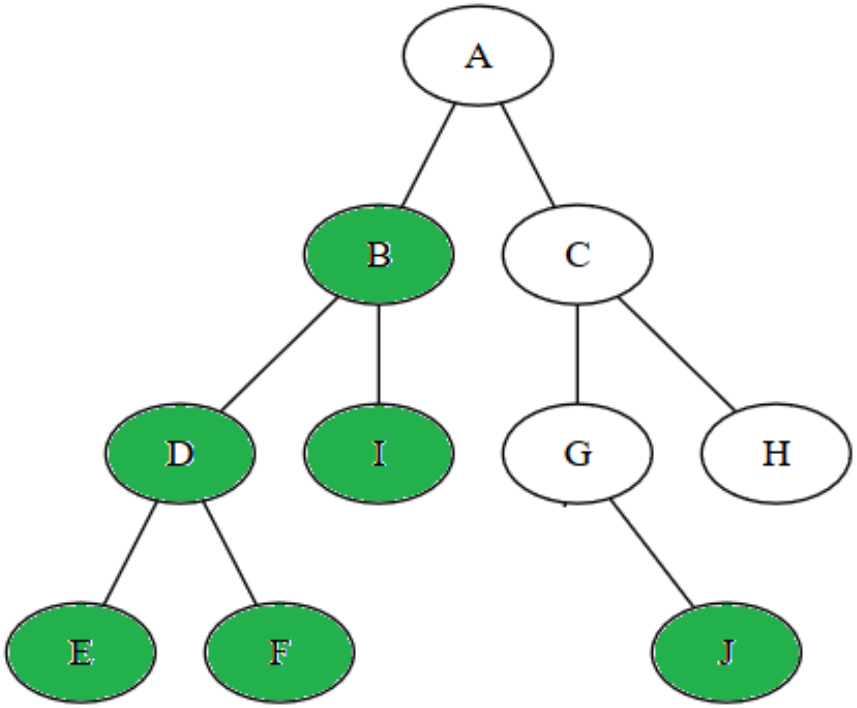
Parcurgere în postordine VI

- Acum vizităm B (subarborele stâng și drept e vizitat deja).



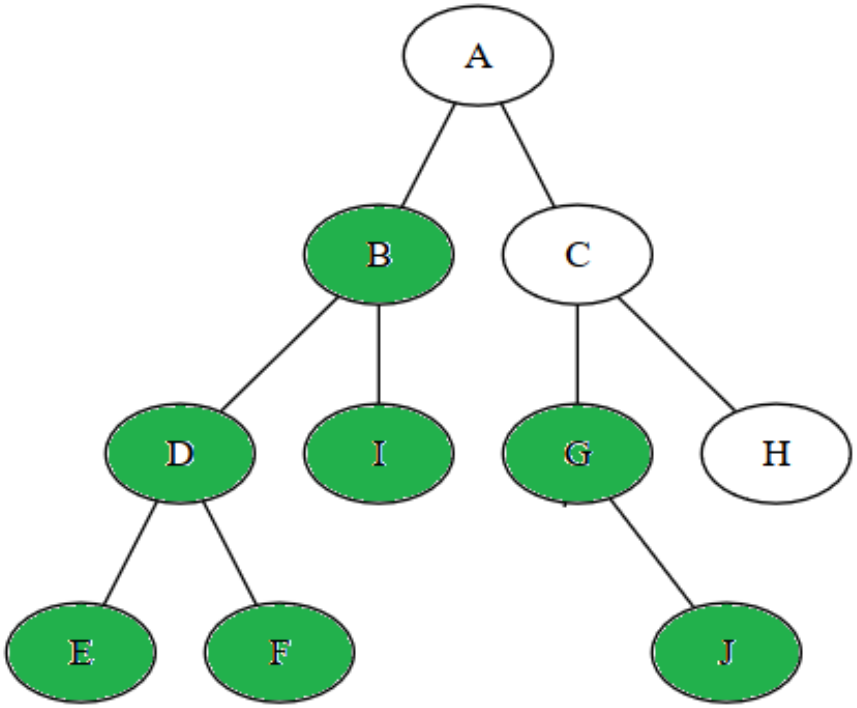
Parcurgere în postordine VII

- Am vizitat subarborele stâng pentru rădăcina A, acum vizităm subarborele drept. Începem cu J.



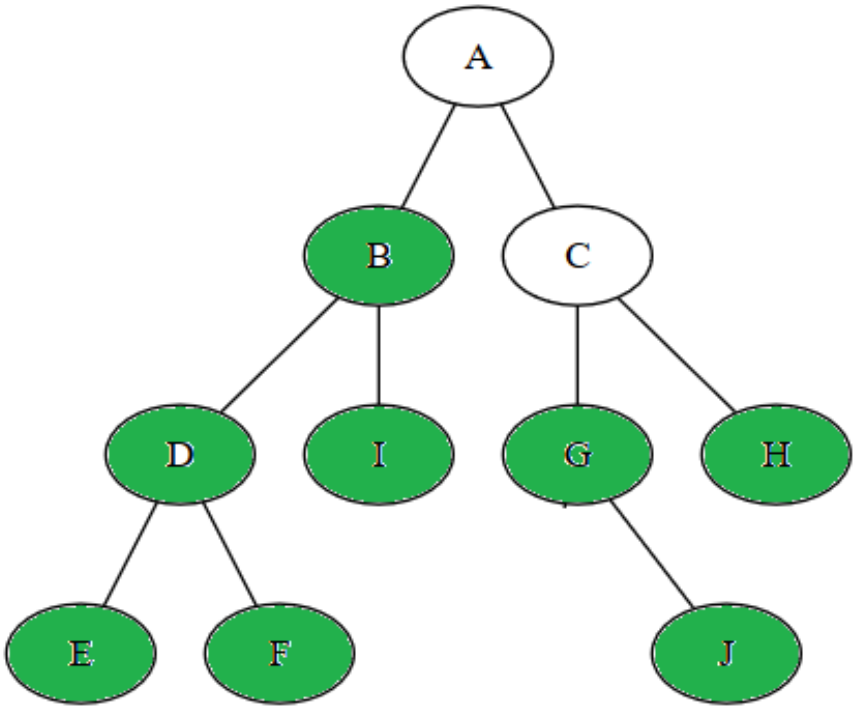
Parcurgere în postordine VIII

- Putem vizita G (subarbore stâng nu are, cel drept este vizitat).



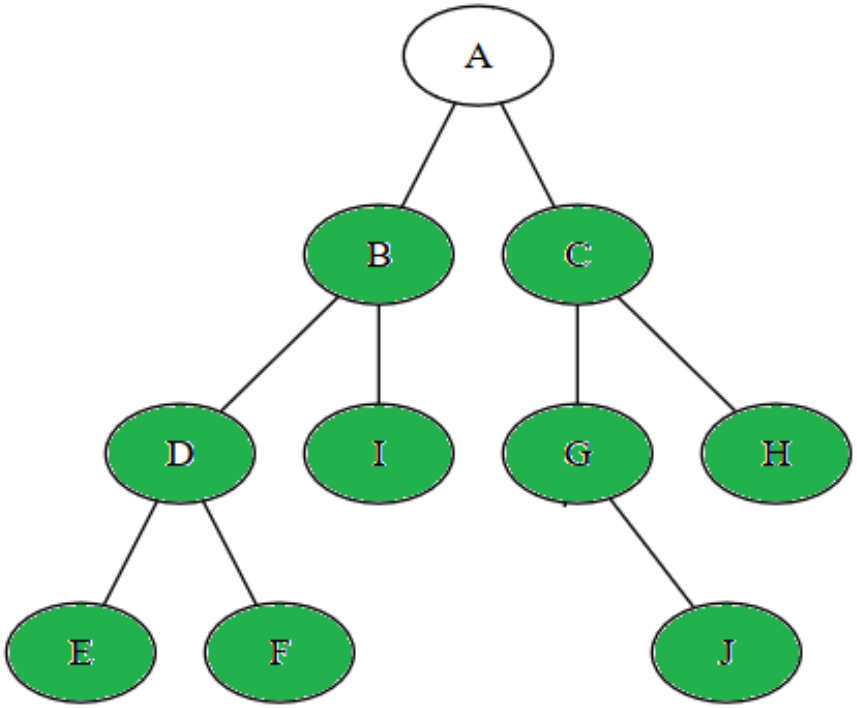
Parcurgere în postordine IX

- Mergem pe subarbore drept al lui C.



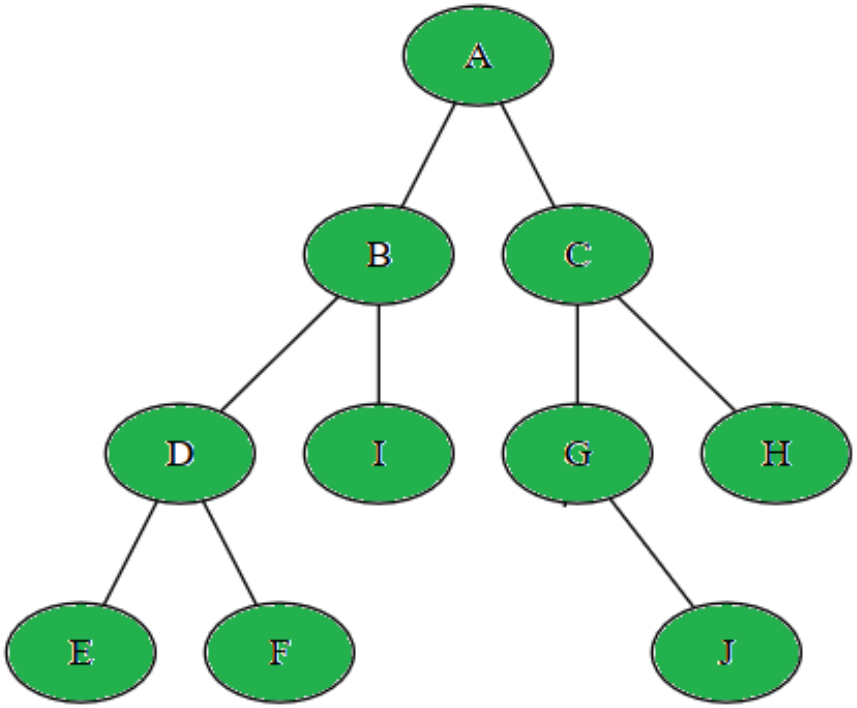
Parcurgere în postordine X

- Vizităm C.



Parcurgere în postordine XI

- La final vizităm rădăcina A.



Parcurgere în postordine XII

- Acum am vizitat toate nodurile. Ordinea în care ele au fost vizitate: E, F, D, I, B, J, G, H, C, A.
- Implementarea parcurgerii se face similar cu preordine și inordine, și folosim aceeași reprezentare.

Parcurgere în postordine XIII

```
subalgoritm parcurgePostordine() este
    postordineRecursiv(this.rad)
sf_subalgoritm
```

Parcurgere în postordine XIV

```
subalgoritm postordoneRecursiv (nod: ↑ Nod) este
//nod este un nod. Poate fi NIL
   dacă nod ≠ NIL atunci
      postordineRecursiv([nod].stang)
      postordineRecursiv([nod].drept)
      scrie [nod].info
   sf_dacă
sf_subalgoritm
```

Preordine, inordine, postordine

- Cele 3 parcurgeri pot fi reținute cel mai ușor așa:
  - Subarborele stâng este vizitat tot timpul înainte de subarborele drept.
  - Poziția rădăcinii se modifica:
    - PREordine: R, S, D
    - INordine: S, R, D
    - POSTordine: S, D, R

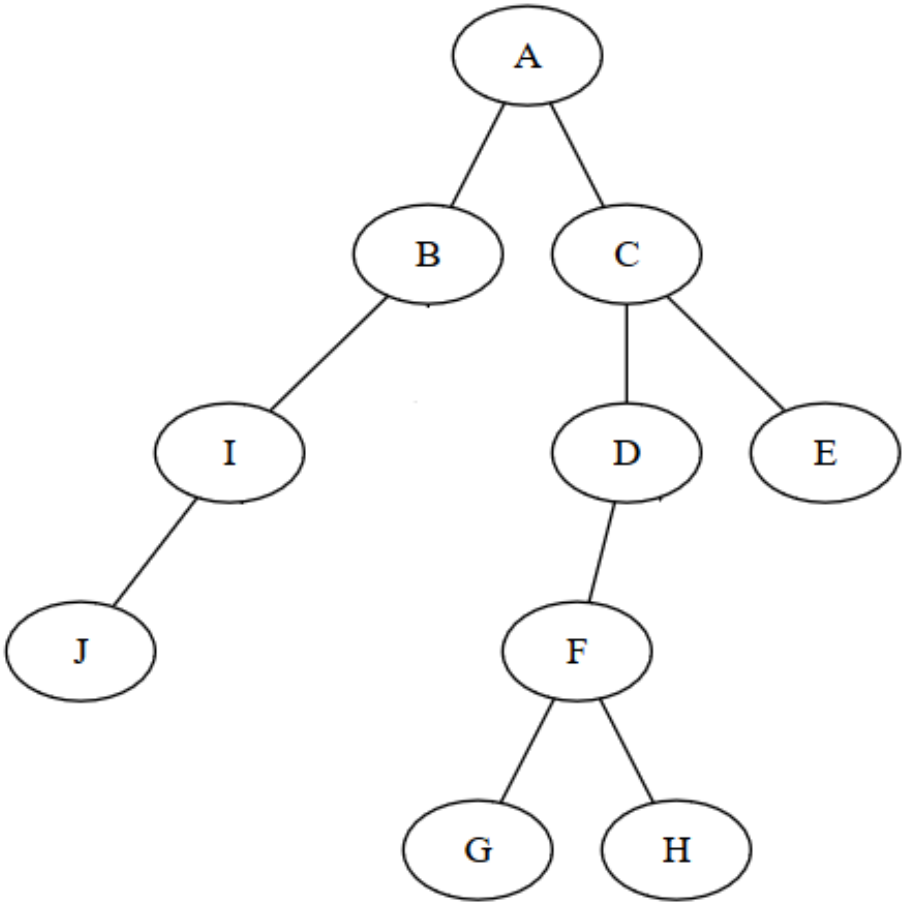
Întrebare I

- Dacă am pentru un arbore parcurgerea în inordine și în preordine, cum pot reconstrui arborele?
- Cum arată arborele care are următoarele noduri:
  - J I B A G F H D C E (inordine)
  - A B I J C D F G H E (preordine)

Întrebare II

- Idee:
  - Din preordine știu care e rădăcina (A).
  - Inordine este S, R, D, dacă știu rădăcina, pot găsi sânga (J I B) și dreapta (G F H D C E)
  - Deci subarborele stâng conține J I B (inordine) B I J (preordine)
  - Subarborele drept conține G F H D C E (inordine) C D F G H E (preordine)
  - Continuăm recursiv

Întrebare III



Codificare Huffman

- O problemă în care se folosesc arbori binari este codificarea Huffman: având un text, vrem să-i atribuim un cod binar (adică o secvență de 0 și 1) fiecărui simbol (literă) din text.
- În codificare Huffman, codurile au lungimi diferite, în principiu vrem ca acele simboluri care apar mai des, să primească un cod mai scurt.
- Dacă codurile au lungimi diferite, nu orice secvență de 0 și 1 reprezintă un cod valid: nici-un cod nu poate să fie prefixul unui alt cod.
- De exemplu - dacă avem următoarele coduri:
  - a - 01
  - b - 11
  - c - 1101
  - d - 1
- dacă vrem să decodăm textul 110101 nu știm dacă este: *ddaa*, sau *ca*, sau *baa*

Codificare Huffman

- Pentru codificare Huffman aveem nevoie de un text în care să numărăm frecvențele simbolurilor (sau ne trebuie direct frecvențele).
- Pentru fiecare simbol creăm un arbore binar format doar dintr-un nod (care conține simbolul respectiv).
- Adăugăm aceste arbori într-o coadă cu priorități, iar prioritatea pentru fiecare arbore este frecvența simbolului.

Codificare Huffman

- Cât timp în coada cu priorități sunt minim 2 elemente, ștergem din coada arborii cu prioritatea minimă. Construim din acești doi arbori un arbore nou, care are acești 2 arbori ca descendent stâng și drept. Punem noul arbore înapoi în Coada cu priorități și prioritatea lui este suma priorităților pentru cei 2 arbori uniți.

- La final, când coada cu priorități conține un singur element, acel arbore ne dă codurile. Fiecare frunză din arbore conține un simbol, iar codul simbolului este construit traversând de la rădăcină spre nodul frunză, adăugând în cod 0 când mergem pe stânga și 1 când mergem pe dreapta.

Codificare Huffman - exemplu

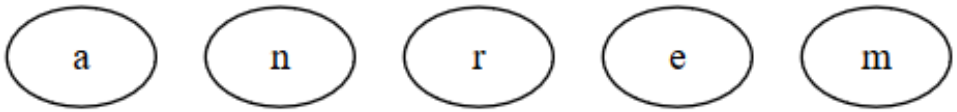
- Să codificăm textul: *ana are mere* (fără spații)

Codificare Huffman - exemplu

- Să codificăm textul: *ana are mere* (fără spații)
- Prima dată avem nevoie de literele și frecvența lor:
  - a - 3
  - n - 1
  - r - 2
  - e - 3
  - m - 1

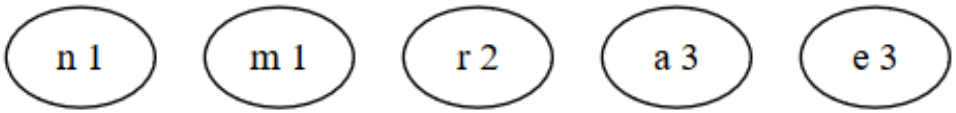
Codificare Huffman - exemplu

- Să codificăm textul: *ana are mere* (fără spații)
- Prima dată avem nevoie de literele și frecvența lor:
  - a - 3
  - n - 1
  - r - 2
  - e - 3
  - m - 1
- Vom consturi câte un arbore binar pentru fiecare simbol, alcătuit doar dintr-un singur nod:



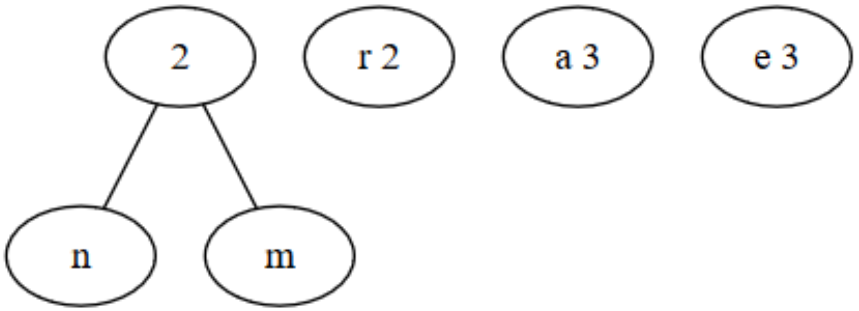
Codificare Huffman - exemplu

- Adăugă aceste noduri într-o Coadă cu Priorități în care prioritatea va fi frecvența simbolului (cifra de după literă)



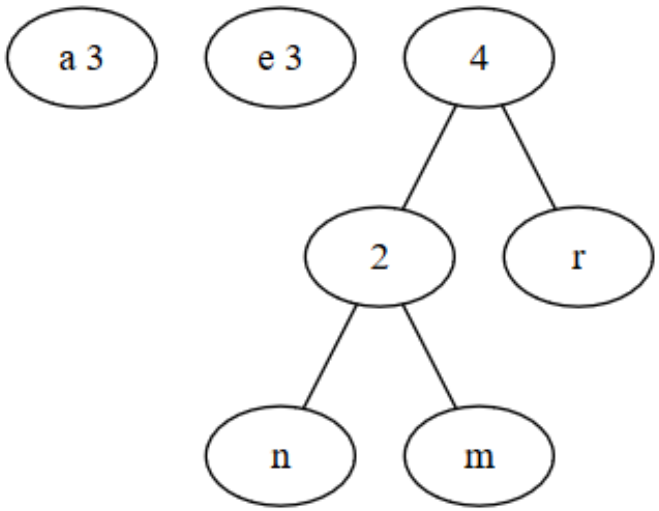
Codificare Huffman - exemplu

- Ștergem 2 elemente din coada cu priorități (arborele cu n și arborele cu m) le unim și punem arborele rezultat înapoi în coadă cu prioritatea 2



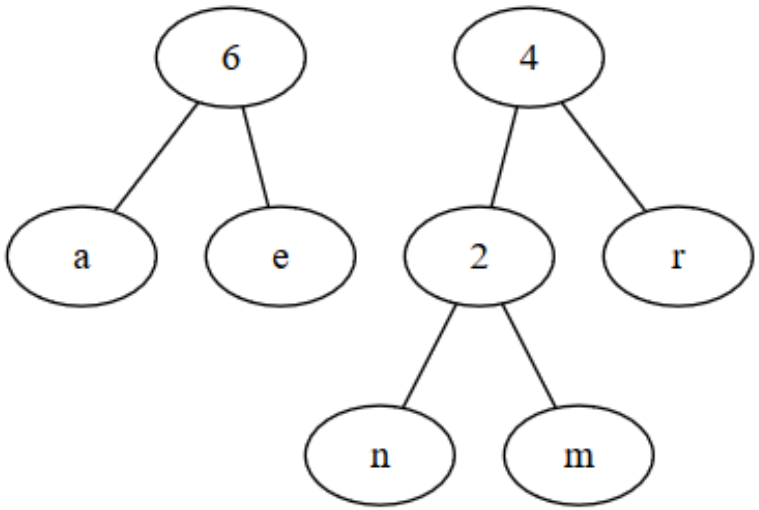
Codificare Huffman - exemplu

- Ștergem 2 elemente din coada cu priorități (cei doi arbori cu prioritatea 2) le unim și punem arborele rezultat înapoi în coadă cu prioritatea 4



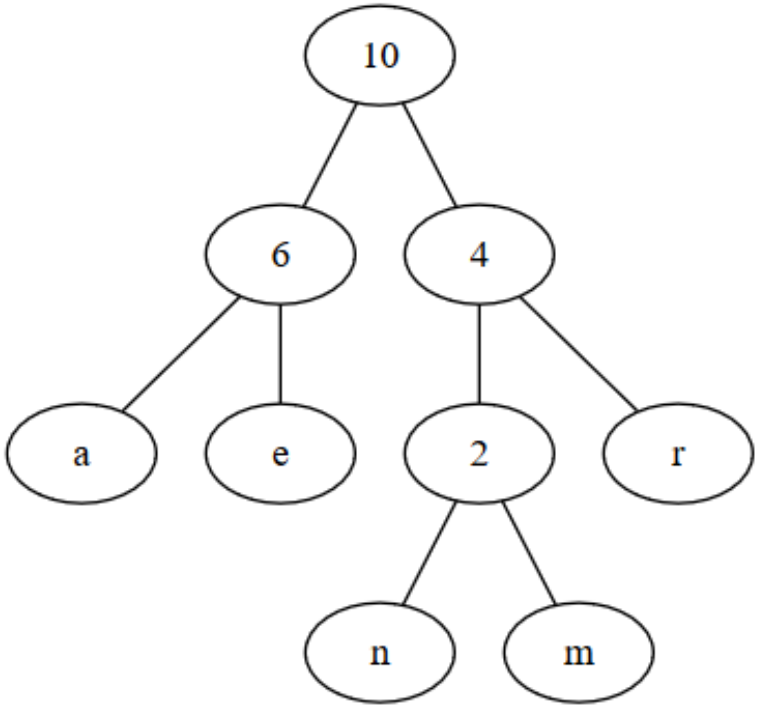
Codificare Huffman - exemplu

- Ștergem 2 elemente din coada cu priorități (cei doi arbori cu prioritatea 3) le unim și punem arborele rezultat înapoi în coadă cu prioritatea 6



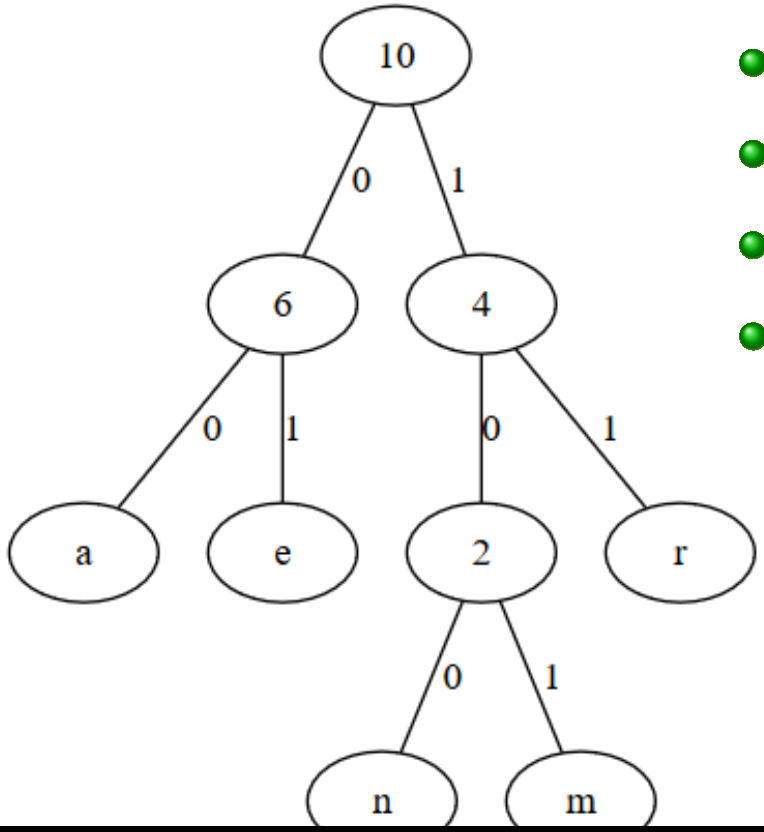
Codificare Huffman - exemplu

- Ștergem 2 elemente din coada cu priorități le unim și avem un singur arbore



Codificare Huffman - exemplu

- Pentru a determina codul pentru fiecare simbol, traversăm de la rădăcina la o frunză, adunăm 0 la cod când mergem pe subarborele stâng și 1 când mergem pe cel drept.



- a - 00
- e - 01
- n - 100
- m - 101
- r - 11

Codificare Huffman - exemplu

- Pentru a decoda un mesaj, pornim de la rădăcină și mergem în stânga dacă codul curent este 0 și în dreapta dacă codul e 1. Când ajungem la o frunză, am decodat o literă, reîncepem de la rădăcină.

- Să decodăm codul: 10100110101

Codificare Huffman - exemplu

- Pentru a decoda un mesaj, pornim de la rădăcină și mergem în stânga dacă codul curent este 0 și în dreapta dacă codul e 1. Când ajungem la o frunză, am decodat o literă, reîncepem de la rădăcină.

- Să decodăm codul: 10100110101
- Pornim de la rădăcină. Codul curent este 1, mergem la dreapta, de acolo stânga și iar dreapta. Am ajuns la litera *m*.
- Continuăm de la rădăcină cu restul codului: 00110101
- etc.