Logică și structuri discrete

Arbori

Marius Minea
marius@cs.upt.ro

http://cs.upt.ro/~marius/curs/lsd/

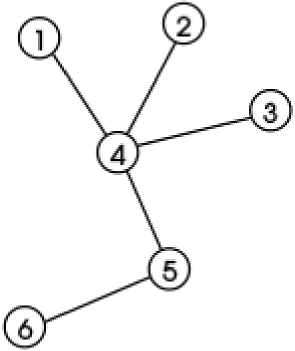
11 decembrie 2017

Arbori

Un arbore e un *graf conex fără cicluri*.
conex = drum între orice 2 noduri (din 1 sau mai mulți pași)

E compus din *noduri* și *ramuri* (muchii).

$\Rightarrow$ un arbore cu n noduri are $n - 1$ ramuri
(demonstrăm prin *inducție*)



Imagine: `http://en.wikipedia.org/wiki/File:Tree_graph.svg`

Un arbore cu n noduri are n-1 muchii

Demonstrăm prin inducție: $n = 1$ e trivial (un nod fără muchii).

Fie mulțimea *tuturor drumurilor* într-un arbore cu $n > 1$ noduri.

Cum arborele e aciclic, un drum are doar noduri distincte
(altfel, un drum $v_i ... v_j$ cu $v_i = v_j$ e un ciclu).

$\Rightarrow$ un drum are cel mult n noduri $\Rightarrow$ numărul de drumuri e *finit*
$\Rightarrow$ există un drum de *lungime maximă* $v_{i_1} v_{i_2} \ldots v_{i_k}$
Atunci, v_{i_1} e legat doar de v_{i_2}, altfel, $v_{alt} v_{i_1} v_{i_2} \ldots v_{i_k}$ ar fi mai lung!

Ștergem nodul v_{i_1} și muchia (v_{i_1}, v_{i_2}). Graful obținut rămâne conex
(niciun drum în graful inițial nu are v_{i_1} ca nod interior).
E evident și aciclic (nu am adăugat muchii noi), deci e un arbore.

Din ipoteza de inducție, având $n - 1$ noduri, are $n - 2$ muchii,
deci graful inițial avea cu un nod și o muchie în plus, q.e.d.

Încercați să demonstrați separat și:
Un graf *conex* cu n noduri are *cel puțin* $n - 1$ muchii
Un graf *aciclic* cu n noduri are *cel mult* $n - 1$ muchii

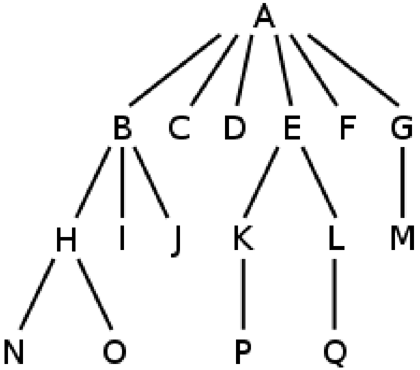
Arbore cu rădăcină

Deobicei identificăm un nod anume numit *rădăcina*, și *orientăm* muchiile în același sens față de rădăcină

Orice nod în afară de rădăcină are un unic *părinte*

Un nod poate avea mai mulți *copii* (*fii*)

Nodurile fără copii se numesc noduri *frunză*

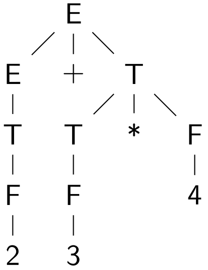


Imagine: `http://en.wikipedia.org/wiki/File:N-ary_to_binary.svg`

Arbori în informatică

Arborii sunt un mod natural de a reprezenta structuri *ierarhice*

- sistemul de fișiere (subarborii sunt cataloagele)
- arborele sintactic într-o gramatică (ex. expresie)
- ierarhia de clase în programarea orientată pe obiecte
- fișierele XML (elementele conțin alte elemente)

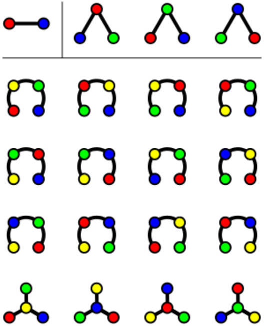



```
<order>
  <item>
    <title="Data Structures"/>
    <price="24.99"/>
  </item>
  <item>
    <title="Mathematical Logic"/>
    <price="39.99"/>
  </item>
<order>
```

Arbori ordonați și neordonați

Ordinea dintre copii poate conta (ex. arbore sintactic) sau nu

Arborii neordonați
cu 2 – 4 noduri:



(argumentați: de ce sunt 12 variante liniare cu 4 noduri?)

Există n^{n-2} arbori neordonați cu n noduri (*formula lui Cayley*)
Demonstrație frumoasă: reprezentând un arbore ca șir de $n-2$ numere de la 1 la n (citiți opțional despre *codul Prüfer*):
șterge numărul minim, scrie nr. la care era legat, până rămân 2 noduri

Imagine: `http://en.wikipedia.org/wiki/File:Cayley's_formula_2-4.svg`

Arborele definit recursiv

Un arbore e un *nod* cu 0 sau mai mulți *subarbori*

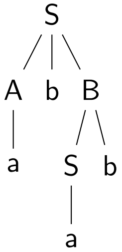
⇒ o *listă* de subarbori (frunzele au lista vidă)

În funcție de problemă, nodurile conțin *informație*

În exemplu: toate nodurile au informație de același *tip*

```
type ’a tree = T of ’a * ’a tree list

let t = T(’S’,[T(’A’,[T(’a’,[])]); T(’b’,[]);
  T(’B’,[T(’S’, [T(’a’,[])]);T(’b’,[])])])
```



Definiție recursivă: cu arbore vid

Definiția dată: bună când arborele totdeauna există (ex.: expresie)

Uneori, arborele poate fi vid (ex.: pentru reprezentarea de mulțimi).
Putem defini atunci:

Un arbore e fie arborele *vid* sau un *nod* cu mai mulți *subarbori*

⇒ extindem tipul anterior cu o valoare pentru arborele vid

$$\textit{tip\_nou} = \textit{tip\_vechi} \cup \{\textit{valoare-specială}\}$$

```
type 'a option = None | Some of 'a
```

indică în ML o valoare de tipul `'a` care poate eventual lipsi

⇒ lucrăm cu valori de tipul `'a tree option`

`None` sau `Some t`, cu `t` de tip `'a tree` definit anterior:

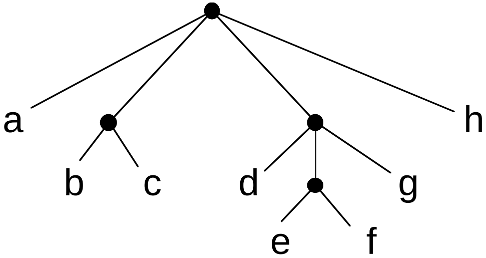
```
type 'a tree = T of 'a * 'a tree list
```

```
let f = function (* parametru arbore, vid sau nu *)
  | None -> (* cazul de prelucrare pentru arborele vid *)
  | Some T(r, tl) -> (* radacina r, lista de copii tl *)
```

Arbori neetichetați

Uneori, nu avem informație utilă decât în nodurile frunză:
⇒ reprezentăm explicit varianta de nod frunză

```
type 'a tree = L of 'a | T of 'a tree list
```



⇒ arborele e echivalent cu o *listă ierarhică* (listă de liste)
[a, [b, c], [d, [e, f], g], h]
dar o listă de liste trebuie să fie uniformă pe nivele (același tip)

```
T [L 'a'; T [L 'b'; L 'c'];
   T [L 'd'; T [L 'e'; L 'f']; L 'g']; L 'h']
```

Imagine: R. M. Keller: Computer Science: Abstraction to Implementation

Arbori binari

Într-un arbore binar, fiecare nod are cel mult doi copii, identificați ca fiul stâng și fiul drept (oricare/ambii pot lipsi)

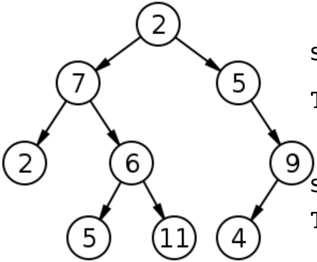
⇒ un arbore binar e fie: arborele vid
un nod cu cel mult doi subarbori

```
type 'a bintree = Nil | T of 'a bintree * 'a * 'a bintree
```

Instanțiind pentru noduri întregi:

```
type inttree = Nil | T of inttree * int * inttree
```

Un arbore binar de înălțime n are cel mult $2^{n+1} - 1$ noduri



subarborele stâng:

```
T (T(Nil, 2, Nil), 7,
   T(T(Nil, 5, Nil), 6, T(Nil, 11, Nil)))
```

subarborele drept:

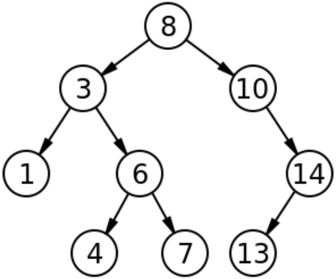
```
T (Nil, 5, T(T(Nil, 4, Nil), 9, Nil))
```

Imagine: `http://en.wikipedia.org/wiki/File:Binary_tree.svg`

Arbori binari de căutare

Memorează valori sortate în ordine:
pentru fiecare nod,
subarborele stâng are valori mai mici,
subarborele drept are valori mai mari

Pot fi folosiți pentru a reprezenta *mulțimi*



Căutarea: recursiv în subarborele potrivit

```
let bsearch x = (* cauta x in arbore *)
  let rec srchx = function (* x fixat mai sus *)
    | Nil -> false
    | T (left, v, right) ->
       v = x || srchx (if x < v then left else right)
  in srchx
```

https://en.wikipedia.org/wiki/Binary_search_tree

Arbori strict binari

engl. strictly binary tree, proper binary tree (binar propriu-zis)

Fiecare nod care nu e frunză are *exact* doi copii

de exemplu, un arbore pentru expresii cu operanzi binari

```
type 'a bintree = L of 'a | T of 'a bintree * 'a * 'a bintree
```

dacă avem același tip în frunze și celelalte noduri

Arbore strict binar cu n frunze $\Rightarrow$ $n-1$ noduri ce nu sunt frunze

Un arbore strict binar de înălțime n are cel mult 2^n frunze

Parcurgerea arborilor

în *preordine*: întâi rădăcina, apoi subarborii

în *inordine*: arborele stâng, apoi rădăcina, apoi arborele drept

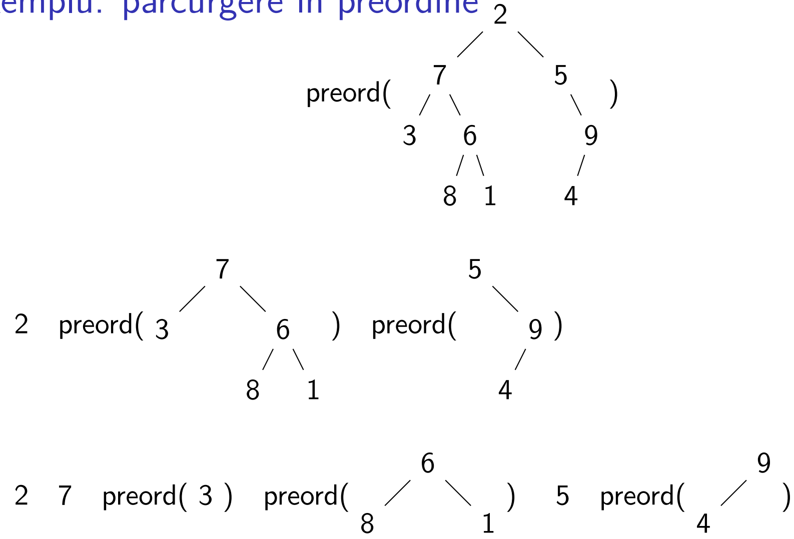
în *postordine*: întâi subarborii, apoi rădăcina

subarborii se parcurg și ei în ordinea indicată

Pentru expresii, obținem astfel formele prefix, infix și postfix

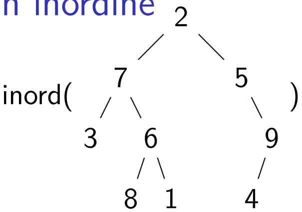
Parcurgerea în pre-/ post-ordine e definită la fel pentru orice arbori (nu doar binari).

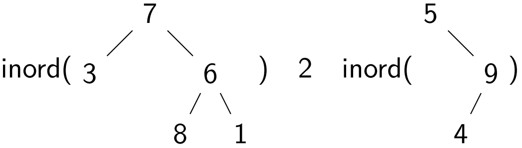
Exemplu: parcurgere în preordine

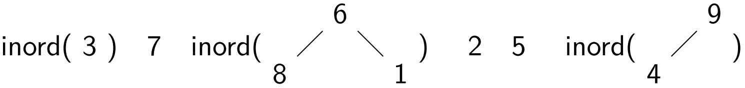


2 7 3 6 8 1 5 9 4

Exemplu: parcurgere în inordine

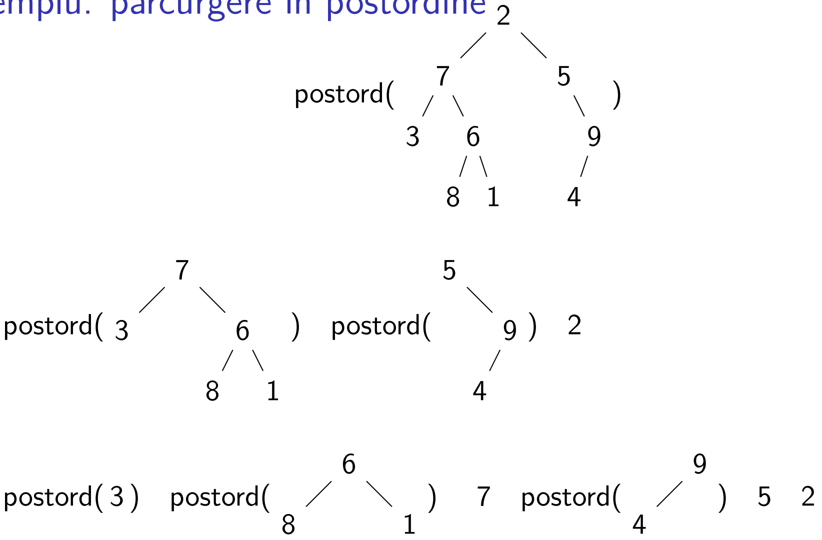






3 7 8 6 1 2 5 4 9

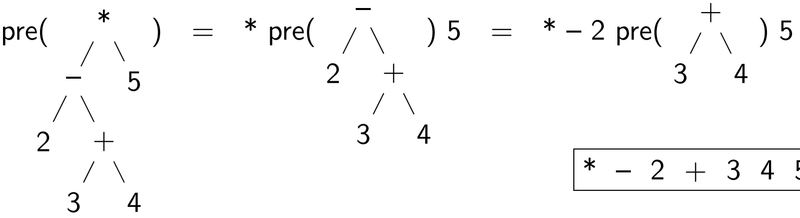
Exemplu: parcurgere în postordine



3 8 1 6 7 4 9 5 2

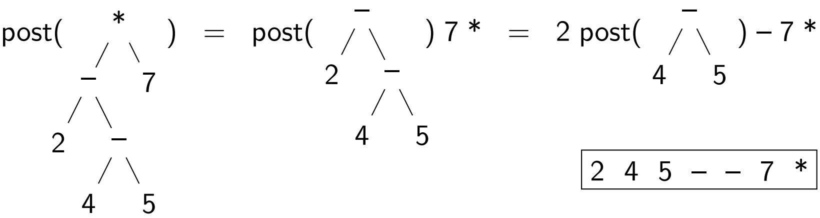
Parcurgeri pentru arbori de expresii

Parcurgere în *preordine* ⇒ expresii *prefix*



$\boxed{*\ -\ 2\ +\ 3\ 4\ 5}$

Parcurgere în *postordine* ⇒ expresii *postfix*

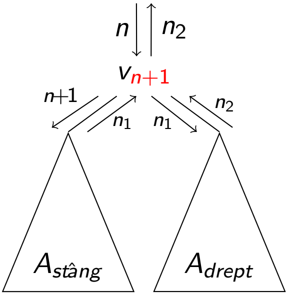


$\boxed{2\ 4\ 5\ -\ -\ 7\ *}$

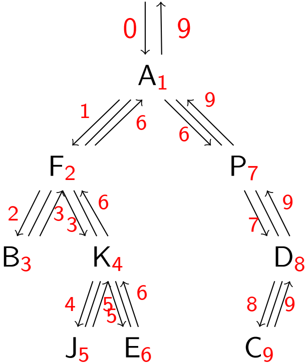
Traversări cu numărarea nodurilor (1)

Funcția primește și returnează *ultimul* număr folosit la etichetare
(dacă un arbore primește n, primul nod va fi etichetat cu $n+1$)
Diferența între numărul returnat și primit e nr. de noduri din arbore.

Preordine

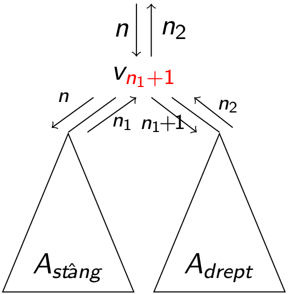


$n_1 - (n+1) = nr\_noduri(A_{stang})$
$n_2 - n_1 = nr\_noduri(A_{drept})$

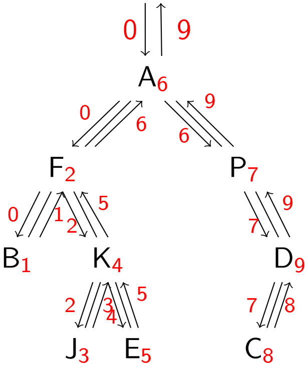


Traversări cu numărarea nodurilor (2)

Inordine

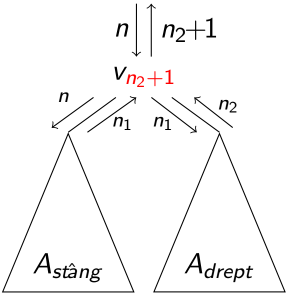


$n_1 - n = nr\_noduri(A_{stang})$
$n_2 - (n_1+1) = nr\_noduri(A_{drept})$



Traversări cu numărarea nodurilor (3)

Postordine



$n_1 - n = nr\_noduri(A_{stang})$
$n_2 - n_1 = nr\_noduri(A_{drept})$