TREES

by

David Toman

University of Waterloo

List of Slides

1
2 Tree Data Structure
3 Example
4 On Different Trees
5 Practical Considerations
6 Applications
7 Syntactic (expression) Trees
8 Abstract Syntax Tree
9 PASCAL Variant Record
10 Tree Traversal
11 DFS Example: Pretty-Print
12 DFS without Recursion?
13 DFS no Recursion, no Stack?
14 How to Implement BFS?
15 Trees with unbounded fan-out
16 Example: Unix **`find(1)`** Command
17 Arbitrary Trees as Binary Trees
18 Example
19 Summary

Tree Data Structure

- an *inductive* type of the form

$$\texttt{tree}(\tau_1, \tau_2) = \texttt{leaf}(\tau_1) + \texttt{node}(\tau_2 \times \texttt{tree}(\tau_1, \tau_2) \times \texttt{tree}(\tau_1, \tau_2))$$

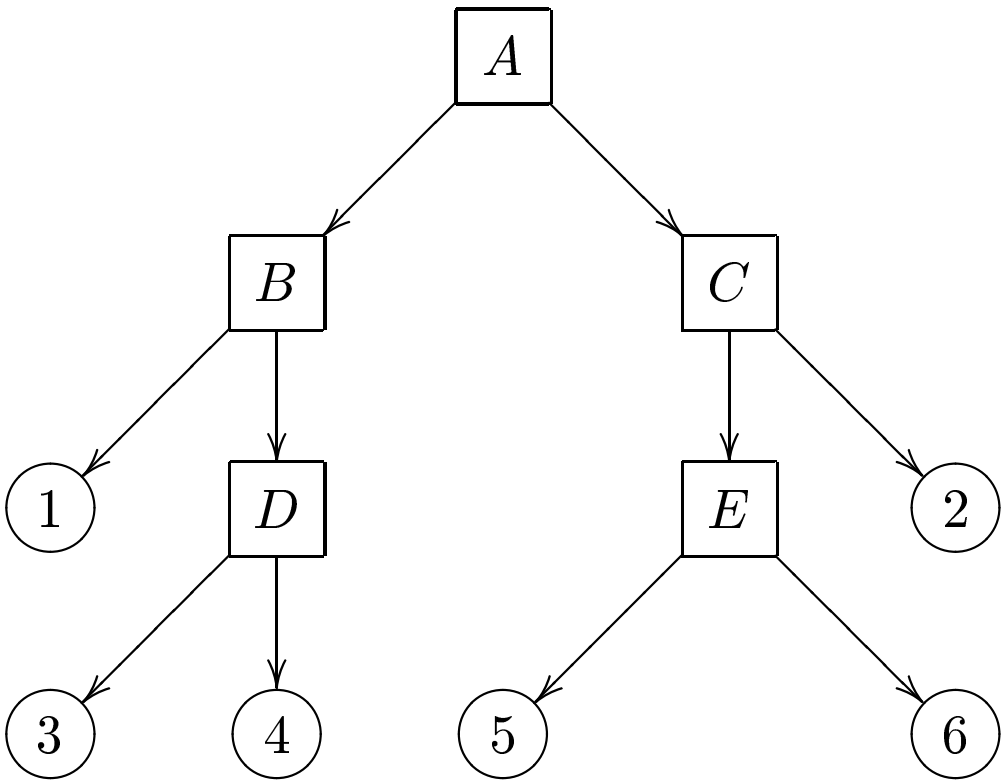
  τ_1: type of values stored in *leaves*
  τ_2: type of values stored in *internal nodes*

- above: a *binary* tree
  $\Rightarrow$ all internal nodes have exactly two children

- ADT operations:

  (1) **`is-leaf`**?
  (2) **`is-node`**? followed by
  `get-value`, **`get-tree1`**, and **`get-tree2`**.

Example



τ_1 : **integer** τ_2 : **character**

On Different Trees

- most “common” binary tree

$$\mathtt{tree}(\tau) = \mathtt{leaf} + \tau \times \mathtt{node}(\mathtt{tree}(\tau) \times \mathtt{tree}(\tau))$$

- binary tree, all τ-values in leaves

$$\mathtt{tree}(\tau) = \mathtt{leaf}(\tau) + \mathtt{node}(\mathtt{tree}(\tau) \times \mathtt{tree}(\tau))$$

- unary-binary tree all values in internal nodes

$$\begin{aligned} \mathtt{tree}(\tau) &= \mathtt{leaf} \\ &+ \mathtt{n1}(\tau \times \mathtt{tree}(\tau)) \\ &+ \mathtt{n2}(\tau \times \mathtt{tree}(\tau) \times \mathtt{tree}(\tau)) \end{aligned}$$

- expression trees

  $\Rightarrow$ many different kinds of nodes

- trees with varying fan-out

- . . .

Practical Considerations

- Binary trees:

  $\Rightarrow$ common practice—all nodes are the same:

```
node = record
         value: tau;
         tree1, tree2: ^node;
       end;
```

  a **`leaf`** commonly represented by a **`nil`**.

Applications

Two main application areas for trees:

1. we care about the tree structure:

   $\Rightarrow$ expression trees

   the structure of the tree captures structure of a syntactic object (e.g., a *program*)

   $\Rightarrow$ the particular shape of the tree is important

2. we don't:

   $\Rightarrow$ search trees

   a tree data structure is used to implement *another* data structure/ADT where its particular shape does not matter as long as it obeys some basic invariant(s).

   $\Rightarrow$ tree properties (in particular tree *depth*) matter.

Syntactic (expression) Trees

- abstract syntax:

$$
\begin{array}{rcl}
cmd & ::= & \textbf{skip} \\
 & | & loc(\tau) := exp(\tau) \\
 & | & cmd \ ; \ cmd \\
 & | & \textbf{if } exp(bool) \textbf{ then } cmd \textbf{ else } cmd \textbf{ fi} \\
 & | & \textbf{while } exp(bool) \textbf{ do } cmd \textbf{ od}
\end{array}
$$

- tree representation:

$$
\begin{array}{rcl}
\texttt{c_tree} & = & \texttt{skip} \\
 & + & \texttt{asgn}(\texttt{loc}, \texttt{e_tree}) \\
 & + & \texttt{seq}(\texttt{c_tree}, \texttt{c_tree}) \\
 & + & \texttt{test}(\texttt{e_tree}, \texttt{c_tree}, \texttt{c_tree}) \\
 & + & \texttt{loop}(\texttt{e_tree}, \texttt{c_tree})
\end{array}
$$

Abstract Syntax Tree

$loc_1 := 0;$ **while** $loc_1 = 0$ **do** $loc_1 := loc_1 + 1$ **od**

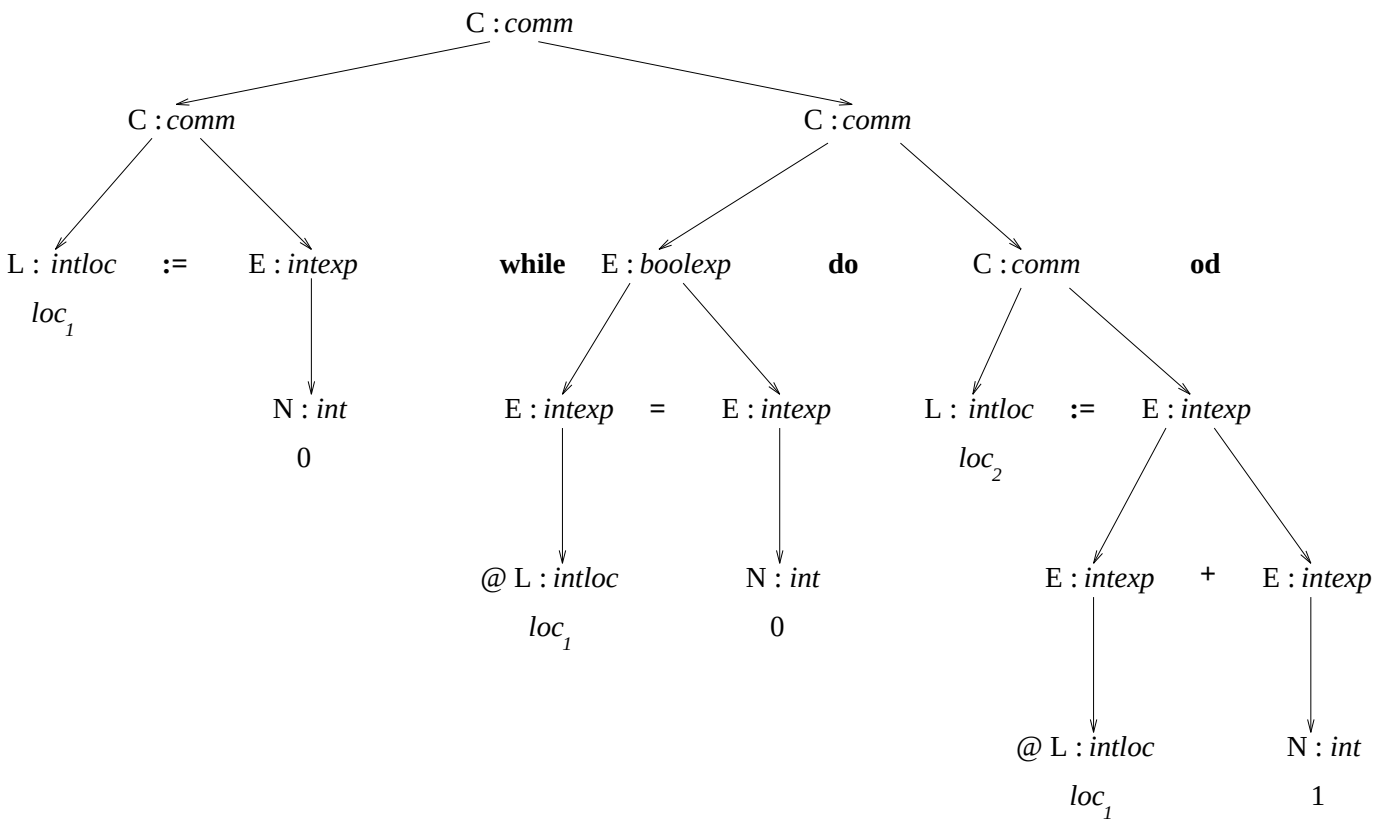


Figure 1.3 (corrected)

PASCAL Variant Record

```
type
  nodes = (skip, asgn, seq, test, loop);

  c_treePtr = ^c_tree;

  c_tree = record
      case node: nodes of
         skip: ( );
         asgn: ( aloc: locptr;
                 aexp: e_treePtr );
         seq:  ( scmd1, scmd2: c_treePtr );
         test: ( texp: e_treePtr;
                 tcmd1, tcmd2: c_treePtr );
         loop: ( lexp: e_treePtr;
                 lcmd: c_treePtr );
      end;
```

$\Rightarrow$ in JAVA: the individual cases turn into
specializations of an *abstract class*

Tree Traversal

A **canonical** algorithm for manipulating trees:

recursion on the structure of the tree

- depth-first (DFS):

  $\Rightarrow$ the ordering of recursive calls determines

  the order in which subtrees are "visited"‘

  $\Rightarrow$ for binary trees this yields:

  the pre-, in-, and post-order traversals

- breadth-first (BFS):

  $\Rightarrow$ level-by-level traversal

DFS Example: Pretty-Print

```
procedure printcmd(t:treePtr);
begin
  case t^.node of
    skip: write('SKIP');
    asgn: begin
            printloc(t^.aloc);
            write(':='); printexp(t^.aexp);
    seq:  begin
            printcmd(t^.scmd1);
            write('; '); printcmd(t^.scmd2);
          end;
    test: begin
            write('if '); printexp(t^.texp);
            write(' then '); printcmd(t^.tcmd1);
            write(' else '); printcmd(t^.tcmd2);
            write(' fi');
          end;
    loop: begin
            write('while '); printexp(t^.lexp);
            write(' do '); printcmd(t^.lcmd);
            write(' od');
end; end;
```

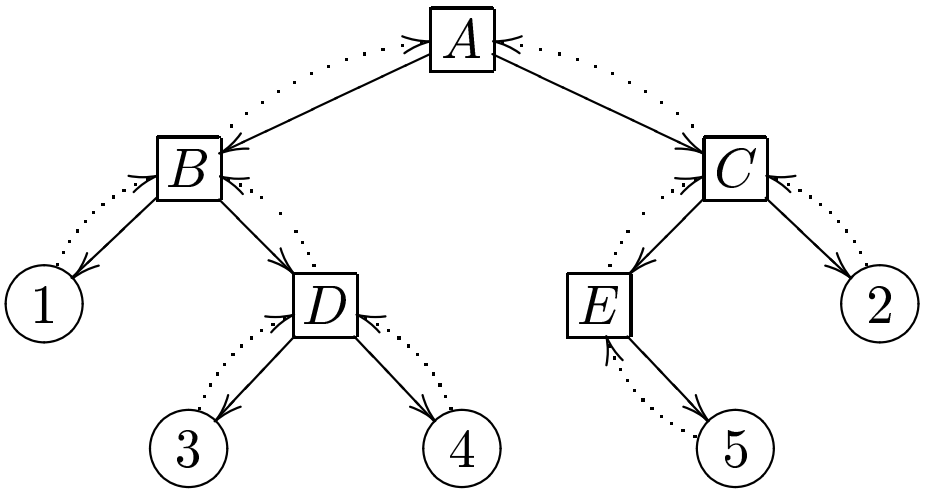
DFS without Recursion?

- DFS can use a *stack* ADT to simulate the recursion

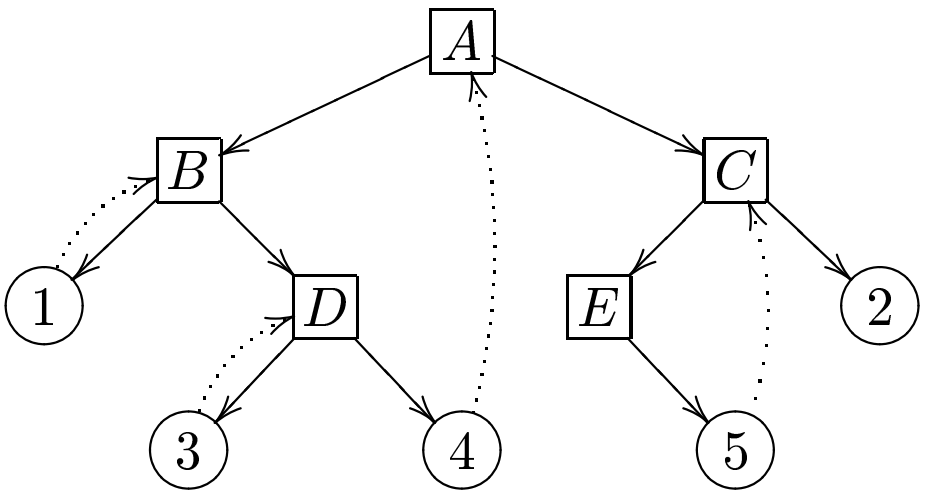
```
procedure DFS-tree(t:tree)
  var s: stack;
      n,n': node;

  push(s,root(t));
  while ( s is not empty ) do
    n := pop(s);

    { process n here }

    for all n' child of n do
                    push(s,n');
  od;
```

DFS no Recursion, no Stack?

IDEA: additional space for pointers back in the tree
$\Rightarrow$ every node has a "parent" reference:



$\Rightarrow$ threaded tree fixes traversal to be "in-order":



[for details on threaded trees see Knuth: TACP I]

How to Implement BFS?

- use a *queue* ADT instead of stack:

```
procedure BFS-tree(t:tree)
  var q: queue;
      n,n': node;

  enqueue(q,root(t));
  while ( q is not empty ) do
    n := dequeue(q);

    {process 'n' here}

    for all n' child of n do
                      enqueue(q,n');
  od;
```

Trees with unbounded fan-out

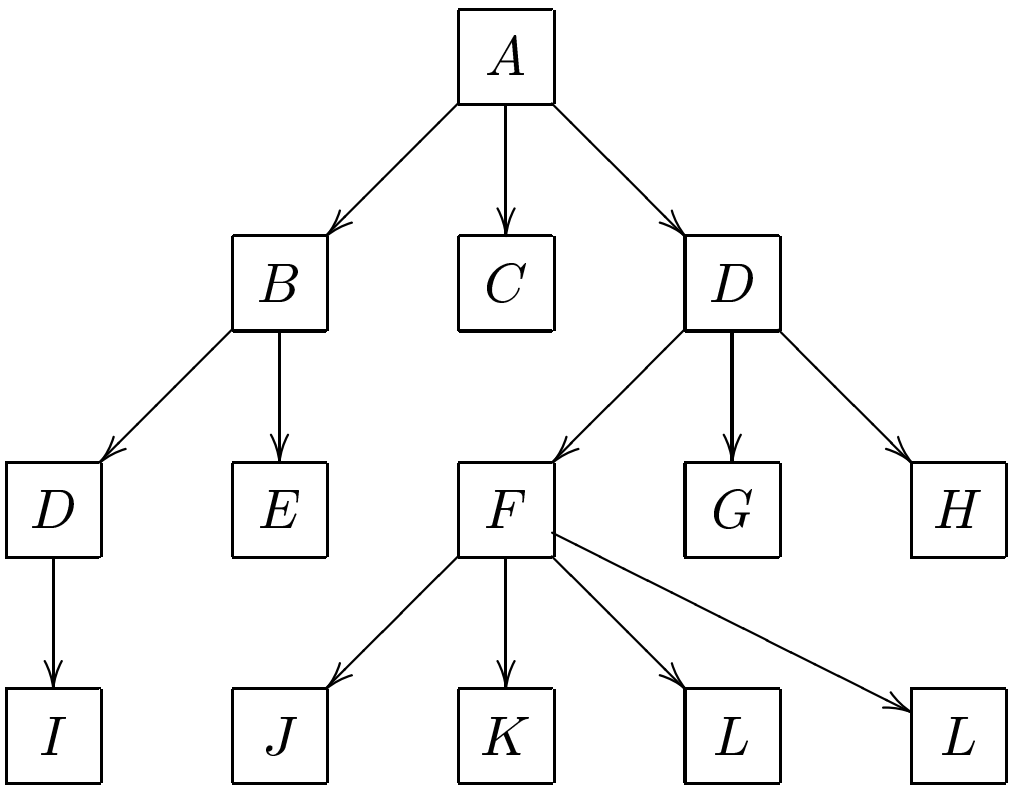
- in every node store a sequence of trees

  $\Rightarrow$ we know how to store sequences: in **lists**

$$\texttt{tree}(\tau) = \tau \times \texttt{list}(\texttt{tree}(\tau))$$

  $\Rightarrow$ leaves: nodes with "empty list" of children

- pictorially:



Example: Unix `find(1)` Command

- data structure:

```
dir_tree = record
             name: str;
             chidren: list(dir_tree);
           end;
```

- traversal—mutual recursion:

```
procedure file-list(t:dir_tree,p:str)
  if (name matches pattern)
        write(p); writeln(t^.name);
  dir-list(t^.children,p+name+'/');

procedure dir-list(t:list(dir_tree),p:str)
  if (t<>nil) then
      file-list(t^.value,p);
      dir-list(t^next,p);
```

- can be simplified!

Arbitrary Trees as Binary Trees

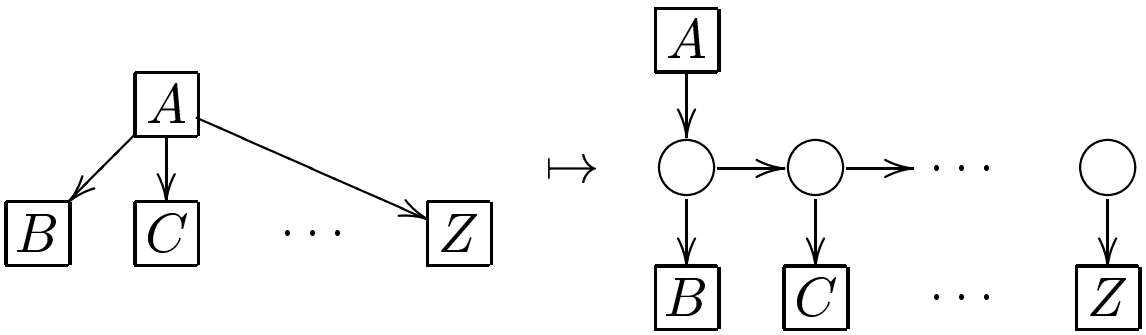
- observation: list is a simple tree

  $\Rightarrow$ all nodes contain

  (1) a "value"—a tree

  (1) a "next"—a list of trees or nil

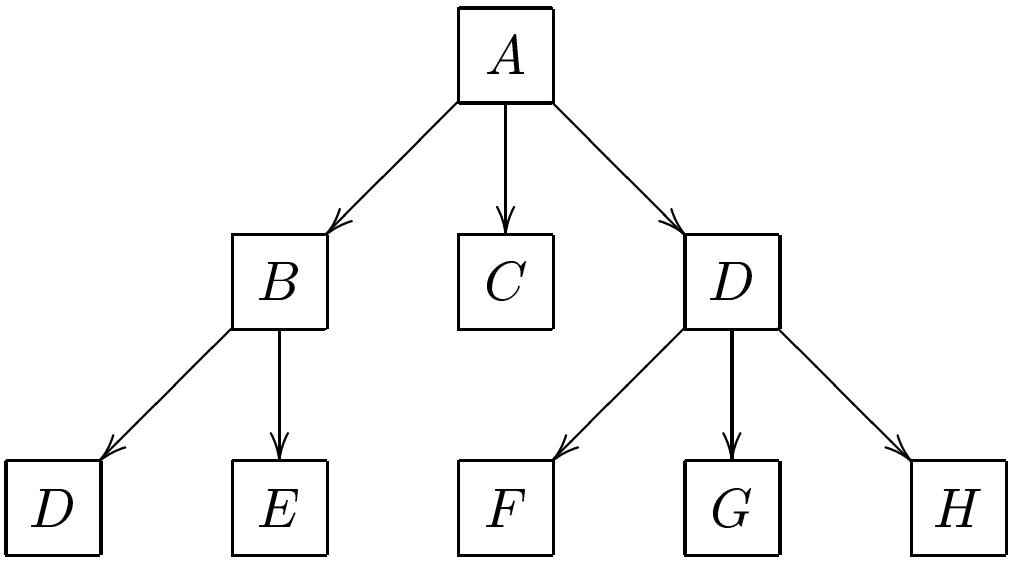
- a tree of trees:



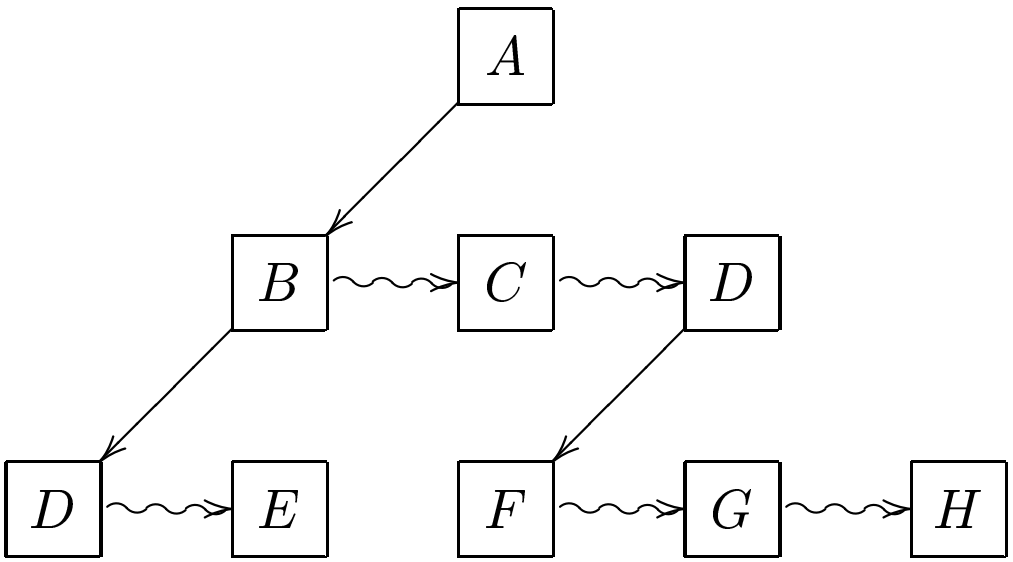
- the "list" nodes are not necessary . . .

Example

- original tree:



- binary-encoded version



Summary

- tree-structured objects are very common

  $\Rightarrow$ in particular for describing syntax

- trees come in various shapes

  $\Rightarrow$ binary trees

  $\Rightarrow$ trees with bounded fan-out (e.g., unary-binary)

  $\Rightarrow$ expression trees fall in this category

  $\Rightarrow$ unbounded fanout trees

- tree algorithms:

  $\Rightarrow$ canonical recursion based on tree structure

- all trees can be simulated by *binary* trees