Trees

FPLI 2023, week 2

Morten Rhiger
Roskilde University

Version 2023-03-14 08:09:22

March 14, 2023

Trees are recursive datatypes, used for

- hierarchies,
- directory structure,
- document object model (DOM) in browsers,
- syntax of programs,
- efficient datatypes,
- and more.

Binary trees in F#

A tree (of αs) is either

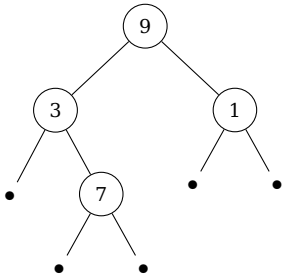
- a *leaf* (an empty tree), or
- a *node* that contains an α and a left and a right branch, both trees (of αs).

```
type 'a tree =
  | LEAF
  | NODE of 'a * 'a tree * 'a tree
```

Binary trees in F#

A tree (of αs) is either

- a *leaf* (an empty tree), or
- a *node* that contains an α and a left and a right branch, both trees (of αs).




```
type 'a tree =
  | LEAF
  | NODE of 'a * 'a tree * 'a tree
```

```
let t = NODE (9, NODE (3, LEAF,
                          NODE (7, LEAF, LEAF)),
                 NODE (1, LEAF, LEAF))
```

Functions on trees

We (almost) always define functions on trees by pattern matching against

- `LEAF`, and
- `NODE (`x`, `l`, `r`)`

Such functions call themselves recursively on l and r.

Functions on trees

We (almost) always define functions on trees by pattern matching against

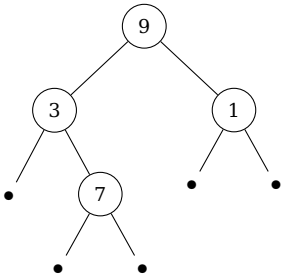
- `LEAF`, and
- `NODE (`x`, `l`, `r`)`

Such functions call themselves recursively on l and r.

```
let rec f = function
  | LEAF            -> ...
  | NODE (x, l, r) ->
    ... f l ... f r ...
```

Computing the “depth” of trees

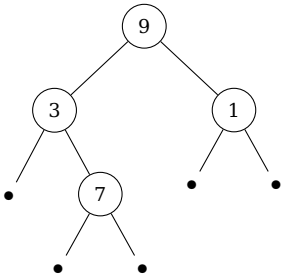
```
let rec depth = function
  | LEAF           -> 0
  | NODE (x, l, r) ->
    1 + max (depth l) (depth r)
```



Computing the “depth” of trees

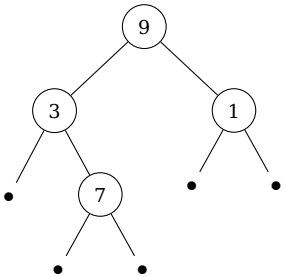
```
let rec depth = function
  | LEAF           -> 0
  | NODE (x, l, r) ->
    1 + max (depth l) (depth r)
```

depth t $\longmapsto$ 3



The sum of the elements of a tree

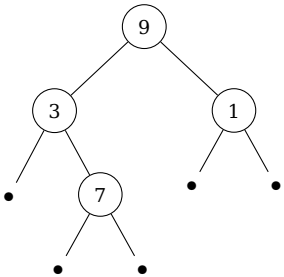
```
let rec sum = function
  | LEAF ->           // ?
  | NODE (x, l, r) -> // ?
```



The sum of the elements of a tree

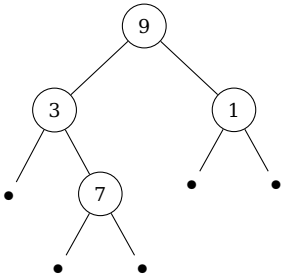
```
let rec sum = function
  | LEAF ->           // ?
  | NODE (x, l, r) -> // ?

sum t ⟼ 20
```



The elements of a tree, “in-order”

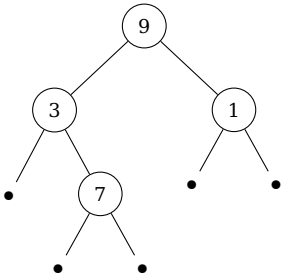
```
let rec inorder = function
  | LEAF           -> []
  | NODE (x, l, r) ->
    inorder l @ [x] @ inorder r
```



The elements of a tree, “in-order”

```
let rec inorder = function
  | LEAF           -> []
  | NODE (x, l, r) ->
    inorder l @ [x] @ inorder r
```

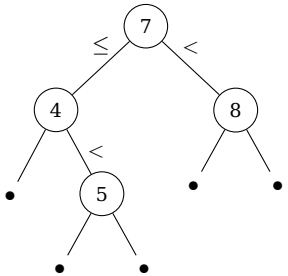
inorder t $\longmapsto$ [3; 7; 9; 1]



Tree sort

In an *ordered tree,* all nodes `NODE (`v`, `L`, `R`)` have the property that

elements in $L \leq v <$ elements in R.



Tree sort

Add one value to an ordered tree

```
let rec add v = function
  | LEAF           ->
```

Tree sort

Add one value to an ordered tree

```
let rec add v = function
  | LEAF           -> NODE (v, LEAF, LEAF)
  | NODE (w, l, r) ->
```

Tree sort

Add one value to an ordered tree

```
let rec add v = function
  | LEAF           -> NODE (v, LEAF, LEAF)
  | NODE (w, l, r) -> if v <= w then
                        NODE (w, add v l, r)
                      else
                        NODE (w, l, add v r)
```

Tree sort

Add many values to an ordered tree

```
let rec makeTree = function
  | []      -> LEAF
  | x :: xs -> add x (makeTree xs)
```

Tree sort

Add many values to an ordered tree

```
let rec makeTree = function
  | []      -> LEAF
  | x :: xs -> add x (makeTree xs)
```

```
let treesort xs = inorder (makeTree xs)
```

```
treesort [5; 2; 3; 1; 4] ⟼ [1; 2; 3; 4; 5]
```