

INF231:
Functional Algorithmic and Programming
Lecture 7: Tree-based structures

Academic Year 2020 - 2021

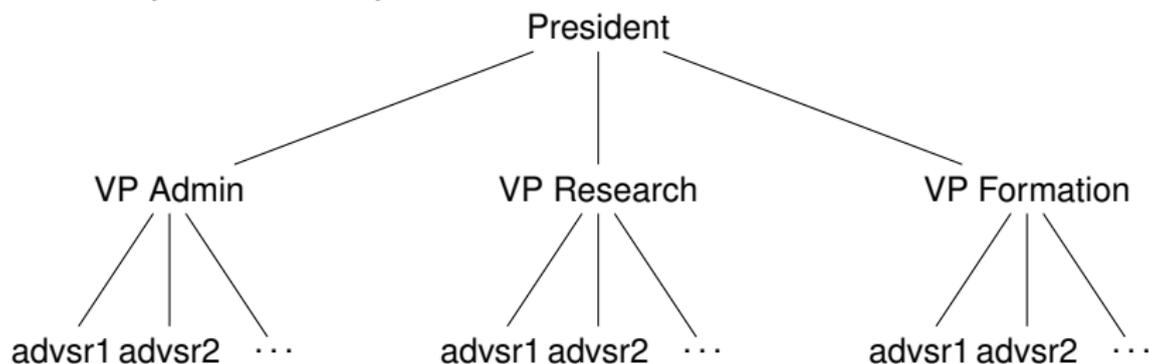
$f(x)$



About Trees

Some motivation and intuition

Presidency of a University:



Remark

- ▶ "root" at the topmost level
- ▶ nodes with/without "subtrees"
- ▶ **Hierarchical** structure
- ▶ Implicit order or **hierarchy**... or not
- ▶ possible repetition

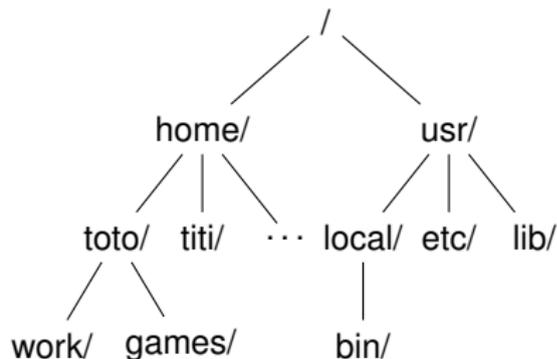
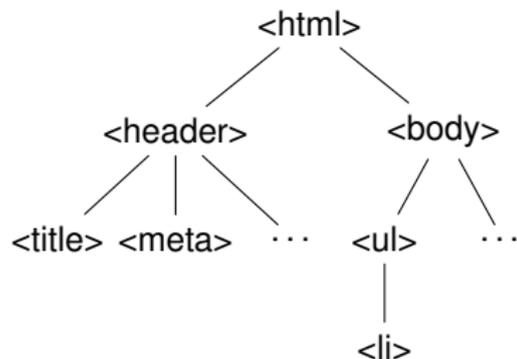


About Trees

Some motivation and intuition

Widely used in computer science mostly because of its notion of hierarchy:
(contrarily to lists)

- ▶ sorting
- ▶ storing "efficiently"
(e.g., a file system where files are organized in directories)
- ▶ compiling: programs are represented with trees
- ▶ structured documents, e.g., a web page
- ▶ modelling



Defining trees

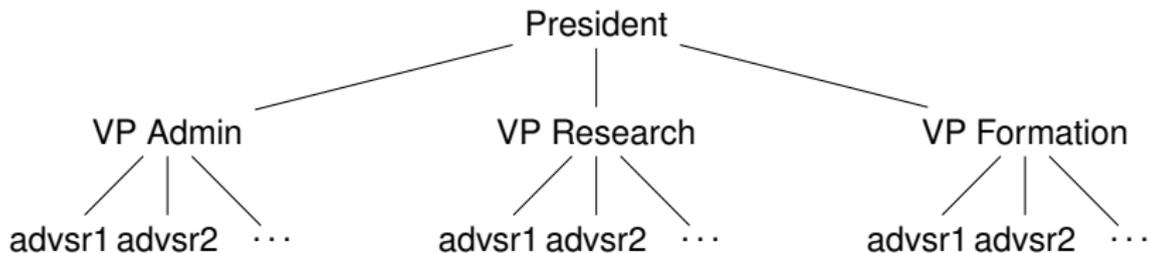
The definition

Definition (Tree)

A tree is a hierarchical recursive data structure which is either:

- ▶ empty
- ▶ a **node** containing a data and (sub) trees

Stores together data *of the same type* (similarly to list)



Defining trees

Some vocabulary

Vocabulary

- ▶ The topmost node is called the **root**
- ▶ The data associated to a node is called its **label** / **content**
- ▶ The sub trees of a node are called the **children**
- ▶ The node directly containing subtrees is called the **father** of the subtrees
- ▶ The node containing subtrees is called an **ancestor** of the subtrees
- ▶ A node with an empty tree is called a **leaf** or a **terminal node**
- ▶ A **branch** of a tree is the list of nodes corresponding to a path from the root to a leaf
- ▶ **level** of a node: length of the branch to this node
- ▶ **depth** of a tree: the maximal level of the nodes in the tree
- ▶ **size** of a tree: the number of nodes in the tree

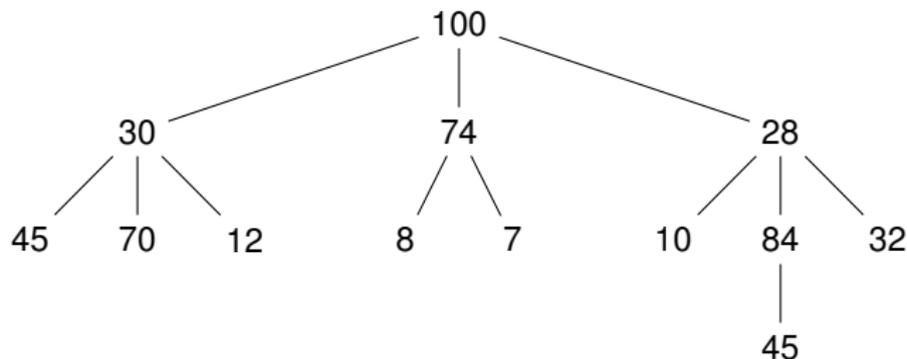
Remark Constraints can be put on, e.g.,

- ▶ the (maximal) number of children a node can have (e.g., binary trees: 2 children per node)
- ▶ how labels are ordered in the tree



An example

Example (A tree)



- ▶ root: 100
- ▶ labels: 100, 30, 64, 28, 45, 70, 12, 8, 7, 10, 84, 32
- ▶ leaves: 45, 70, 12, 8, 7, 10, 84, 32
- ▶ children of 30 are 45, 70, 12
- ▶ 100 is the direct father of 30
- ▶ 100 is at level 0, 7 is at level 2
- ▶ the depth of the tree is 3
- ▶ [100;30;12] is a path

Outline

Binary trees

Binary Search Trees

Binary trees

Definition and example

Definition (Binary Tree)

A tree is a **binary tree** if each node has *at most two children* (possibly empty)

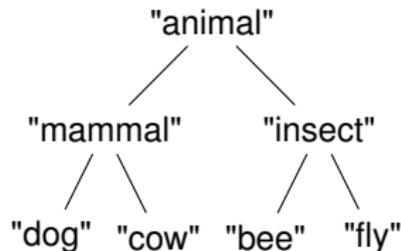
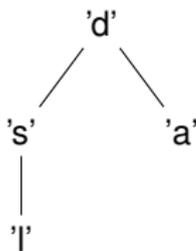
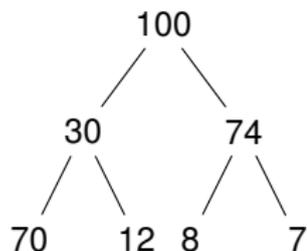
Mathematically:

$$Bt(Elt) = \{EmptyT\} \cup \{Node(tL, e, tR) \mid e \in Elt \wedge tL, tR \in Bt(Elt)\}$$

Example (Binary trees of integers)

$$Bt(\mathbb{N}) = \{EmptyT\} \cup \{Node(tL, e, tR) \mid e \in \mathbb{N} \wedge tL, tR \in Bt(\mathbb{N})\}$$

Example (Binary tree)



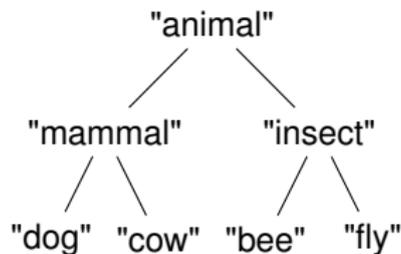
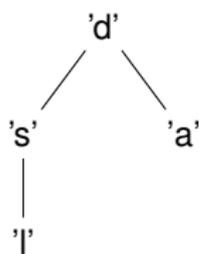
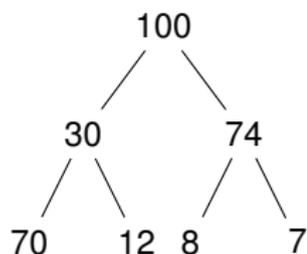
Binary trees

Vocabulary

Vocabulary

- ▶ The first (resp. second) child is also called the left-hand (resp. right-hand) subtree
- ▶ A binary tree t is **complete** if $\text{size}(t) = 2^{\text{depth}(t)} - 1$

Example (Binary tree)



Binary trees of integers

In OCaml

```
type binary_tree =  
  | Empty  
  | Node of int * binary_tree * binary_tree
```

```
type binary_tree =  
  | Empty  
  | Node of binary_tree * int * binary_tree
```

```
type binary_tree =  
  | Empty  
  | Node of binary_tree * binary_tree * int
```

Remark

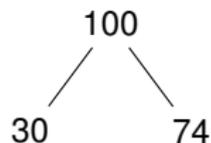
- ▶ Three equivalent definitions
- ▶ The type `binary_tree` has two constructors
- ▶ The constructor `EmptyT` (empty tree) is a constant
- ▶ The constructor `Node` is doubly recursive



Binary trees

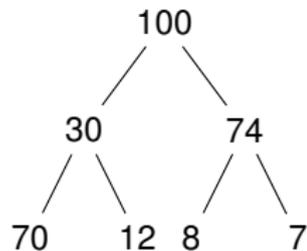
Examples

Example (Defining a tree in OCaml)



```
let bt1 =  
  Node (100,  
        Node (30,EmptyT,EmptyT),  
        Node (74,EmptyT,EmptyT)  
  )
```

Example (Another tree)



```
let bt2 =  
  Node(  
    100,  
    Node(30,  
      Node(70,EmptyT,EmptyT),  
      Node(12,EmptyT,EmptyT)  
    ),  
    Node(74,  
      Node(8,EmptyT,EmptyT),  
      Node(7,EmptyT,EmptyT)  
    )  
  )
```

Some (classical) functions on trees

Example (Depth)

The maximal level of the nodes

```
let rec depth (t:binary_tree):int=  
  match t with  
  | EmptyT → 0  
  | Node (_, t1, t2) → 1+ max (depth t1) (depth t2)
```

Exercise

Define the two following functions

- ▶ `sum`: returns the sum of the elements of a tree
- ▶ `maximum` returns the maximal integer in the tree. Warning this function should not be called on an empty tree

Binary trees

Let's parameterise binary trees

We can parameterise binary trees by a type (polymorphism)

```
type  $\alpha$  binary_tree =  
  | EmptyT  
  | Node of  $\alpha$  *  $\alpha$  binary_tree *  $\alpha$  binary_tree
```

Many possible sorts of binary trees: `int binary_tree`,
`char binary_tree`, `string binary_tree`,...

Remark The element of type α can be placed equivalently in the middle or on the right □

DEMO: Defining some binary trees

Polymorphic Binary trees

Some functions

Example (Belongs to)

Is an element of type α in an α `binary_tree`?

```
let rec belongsto (elt: $\alpha$ ) (t: $\alpha$  bintree):bool =  
  match t with  
  | Empty  $\rightarrow$  false  
  | Node (e,tl,tr)  $\rightarrow$  (e=elt) || belongsto elt tl || belongsto elt tr
```

Example (The list of labels of a tree)

Given an α `binary_tree`, returns the α list of labels

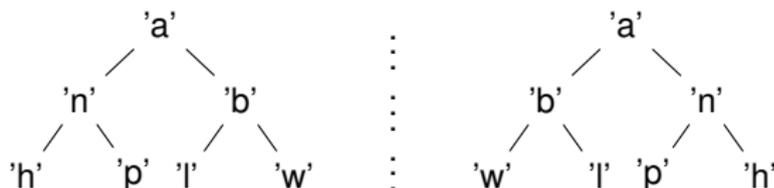
```
let rec labels (t: $\alpha$  bintree): $\alpha$  list=  
  match t with  
  | Empty  $\rightarrow$  []  
  | Node (elt,tl,tr)  $\rightarrow$  (labels tl)@elt::(labels tr)
```

Polymorphic Binary trees

Some functions - let's practice

Exercise: Define the following functions

- ▶ `size`: returns the size of the tree (the number of nodes)
- ▶ `leaves`: given an α bintree returns the α list of leaves of this tree
- ▶ `maptree`: applies a given function to all elements of an α bintree
- ▶ `mirror`: returns the mirror image of an α bintree



Browsing a binary tree

Given a binary search tree, several functions are defined by "browsing the tree"

When encountering a Node (`elt`, `lst`, `rst`), there are several possibilities according to the "moment" when `elt` is treated:

- ▶ treat `elt`, then browse `lst`, then browse `rst`: **prefix** browsing
- ▶ treat `lst`, then browse `elt`, then browse `rst`: **infix** browsing
- ▶ browse `lst`, then browse `rst`, then treat `elt`: **suffix** browsing

Iterators on binary tree

Iterator on a binary tree: `fold_left_right_root`: applies a function `f`

- ▶ to the root, and
- ▶ the results of left subtrees and right subtrees

```
let rec fold_lrr (f:α → β → β → β) (acc:β) (t:α bintree):β =  
  match t with  
  | Empty → acc  
  | Node (elt, l, r) →  
    let rl = fold_lrr f acc l  
      and rr = fold_lrr f acc r  
    in f elt rl rr
```

Defining functions using iterators

Using the function `fold_lrr`, redefine the following functions:

- ▶ `size`
- ▶ `depth`
- ▶ `mirror`

Pathes in a tree

Exercise: Pathes in a binary tree: function `paths`

The purpose is to define a function that computes maximal pathes in a tree:

- ▶ How can we represent a path and a set of pathes?
- ▶ Define a function `add_to_each` that adds an element as the head to each path in a set of pathes
- ▶ Using the previously defined function define the function `paths`

Binary trees

Some properties and how to prove them

Properties of size and depth

- ▶ $depth(t) \leq size(t)$
- ▶ $size(t) \leq 2^{depth(t)-1}$

How to prove them?

Structural induction to prove some property P

Consider $Bt(ElT) = \{EmptyT\} \cup \{Node(tL, e, tR) \mid e \in ElT \wedge tL, tR \in Bt(ElT)\}$

To show that $\forall t \in Bt(ElT) : P(t)$

- ▶ prove $P(ElT)$
- ▶ prove

$$\forall tL, tR \in Bt(ElT) : P(tL) \wedge P(tR) \Rightarrow (\forall e \in ElT : P(Node(tL, e, tR)))$$

Exercise: some proofs

Prove the above properties using structural induction

Outline

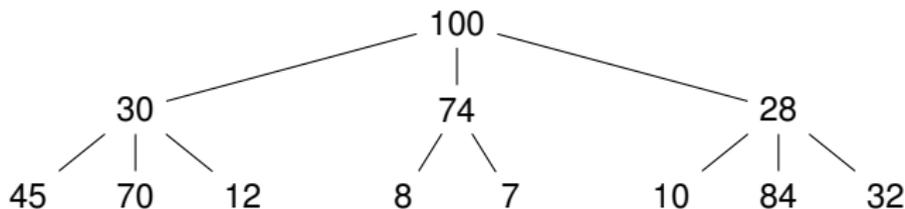
Binary trees

Binary Search Trees

Motivation

Let us come back on the `belongsto` function:

```
let rec belongsto (elt:α) (t:α bintree):bool =  
  match t with  
  | Empty → false  
  | Node (e,tl,tr) → (e=elt) || belongsto elt tl || belongsto elt tr
```



How can we be sure that an element does not belong to the tree?

↔ one has to browse the whole tree

(similarly to what would happen with a list)

Search time depends on the size of the tree

→ solution consists in sorting the elements of the tree

Binary Search Tree: definition

Definition: Binary Search Tree (BST)

A binary search tree is a binary tree s.t. for every node of the tree of the form $\text{Node}(e, l_{st}, r_{st})$, where e is the data carried out by the node, and l_{sb} (resp. r_{sb}) is the left (resp. right) sub tree of the node, we have:

- ▶ l_{st}, r_{st} are binary search trees
- ▶ elements of l_{st} are all lesser than or equal to e
- ▶ e is (strictly) lesser than all elements in r_{st}

Remark

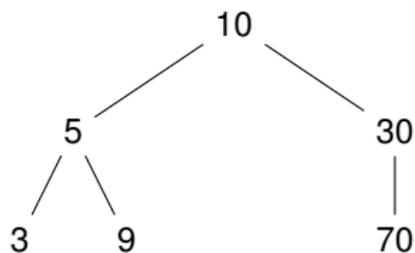
- ▶ Binary search trees suppose that the set of the elements of the tree has a total ordering relation
- ▶ "lesser than" in the definition is understood w.r.t. this ordering relation
- ▶ Elements can be of any type: `int`, `string`, `students...` as long as there is an ordering relation



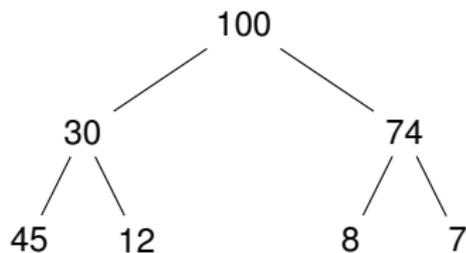
Binary Search Tree

(counter) Example

Example (A binary search tree)



Example (NOT a binary search tree)



Revisiting the `belongsto` function

We can exploit the property of binary search trees

Example (Does an element belong to a binary search tree?)

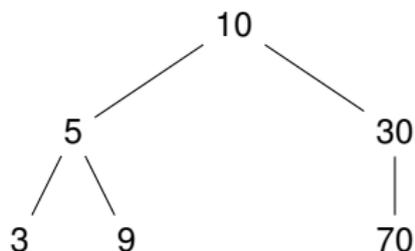
```
let rec belongsto (elt:α) (t:α bst):bool=
  match t with
  | Empty → false
  | Node (e, lbst, rbst) →
    (e=elt)
    || (e>elt) && belongsto elt lbst
    || (e<elt) && belongsto elt rbst
```

One subtree examined at each recursive call

Some sort of "dichotomic" search

An execution of `belongsto`

Let's search 9 in the following tree:



$9 < 10 \Rightarrow$ searching 9 in Node(10,...,...)
 $9 > 5 \Rightarrow$ searching 9 in Node(5,...,...)
 $9 = 9 \Rightarrow$ true

DEMO: Tracing `belongsto` 9 ...

Browsing a tree

Given a binary search tree, how to put the elements in order in a list?

↔ browsing the tree

When encountering a `Node (elt, lst, rst)`, there are several possibilities according to the "moment" when `elt` is treated:

- ▶ place `elt`, then `lst`, then `rst`: **prefix** browsing
- ▶ place `lst`, then `elt`, then `rst`: **infix** browsing
- ▶ place `lst`, then `rst`, then `elt`: **suffix** browsing

Following the property of binary search trees, the infix browsing gives us the solution:

```
let rec tolistinorder (t:  $\alpha$  bst):  $\alpha$  list =  
  match t with  
  | Empty → []  
  | Node (elt, lbst, rbst) →  
    (tolistinorder lbst)@elt::tolistinorder rbst
```

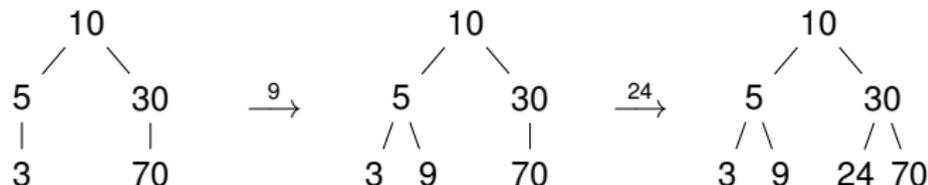
Insertion in a binary search tree

Insert the element as a leaf (simplest method)

Objective: insert an element elt in a binary search tree t

- ▶ preserve the binary search tree property
- ▶ insert the element as a leaf of the tree

Example (Inserting two elements)



Idea: recursively distinguish two cases

- ▶ t is empty, then by inserting elt we obtain $\text{Node}(elt, \text{Empty}, \text{Empty})$
- ▶ t is not empty, then it is of the form $\text{Node}(e, lbst, rbst)$, then
 - ▶ if $elt \leq e$, then elt should be placed in $lbst$
 - ▶ if $elt > e$, then elt should be placed in $rbst$

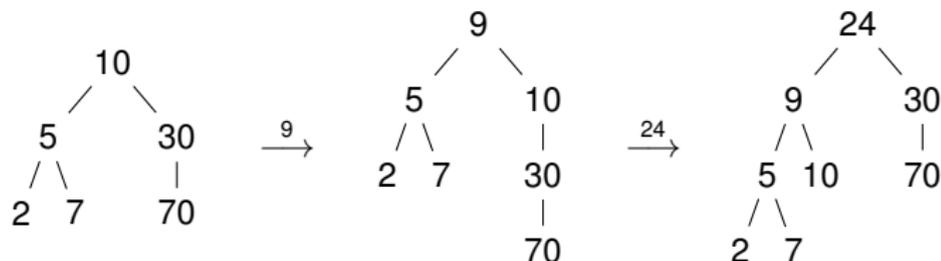
Insertion in a binary search tree

Insert the element as a root

Objective: insert an element elt in a binary search tree t

- ▶ preserve the binary search tree property
- ▶ insert the element as the *root* of the tree

Example (Inserting two elements)



Idea: proceed in two steps:

- ▶ "cut" the tree into two binary search subtrees l and r s.t.
 - ▶ l contains all elements smaller than elt
 - ▶ r contains all elements greater than elt
- ▶ Build the tree $Node(elt, l, r)$

Binary Search Tree

Let's practice insertion

Exercise: insertion as a leaf

Define the function `insert` that inserts an element in a BST, as a leaf

Exercise: insertion as the root

Define the functions:

- ▶ `cut` that cuts a binary search tree as described before
- ▶ `insert` that inserts an element in a binary tree as the root, using `cut`

Exercise: Binary Search Tree creation

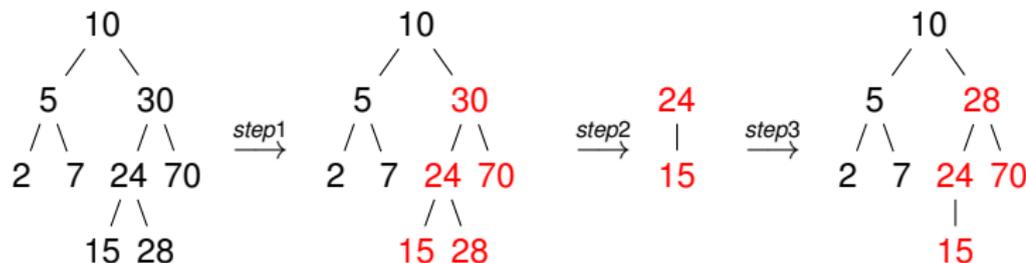
Define two functions `create_bst` that, given a list of elements create a binary search tree of the elements in the list, using the two insertion methods

Suppressing an element in a BST

Suppressing an element in `elt` a BST consists in:

1. Identify the subtree `Node(elt, lst, rst)`
(where suppression should occur)
2. Suppress the greatest element `max` of `lst`
→ we obtain a BST `lstprime`
3. Build the tree `Node(max, lstprime, rst)`

Example (Suppressing 30)



Binary Search Tree

Let's practice suppression

Exercise: suppression in a tree

Define the functions:

- ▶ `remove_max` that remove the greatest element in a tree
To ease the definition of the subsequent function, it is better if this function returns both the maximal element and the new tree
- ▶ `suppression` that suppresses an element in a BST

Exercise: Is a Binary Tree a Binary Search Tree?

Define the function `is_bst` that checks whether a binary tree is a BST

Conclusion

Summary:

About trees:

- ▶ Hierarchical "objects"
- ▶ Doubly recursive data type
- ▶ Two variants (binary trees and binary search trees) (there exist many others)
- ▶ Several functions to manipulate them