Definizione di nuovi tipi in CAML

type giorno = Lun | Mar | Mer | ... | Dom ;;
type defined

enumerazione dei valori (che sono nomi)

l'enumerazione dei valori dà il loro ordine: Lun < Mar < Mer ... < Dom

valori costanti

```
# Lun ;;
- : giorno = Lun

# Lun < Ven ;;
- : bool = true

# Ven < Lun ;;
- : bool = false
```

Enumerare i valori è il modo più semplice per definire un nuovo tipo.

C'è la possibilità di definire nuovi tipi basandosi su tipi già definiti!

Vogliamo definire un nuovo tipo che raccolga valori bool e int

NO!

```
type union = bool | int ;;
```

non mi può fare perché non saprebbe più che tipo ha 5

→ int oppure union !!

```
type union = B of bool | N of int ;;
```

costruttori di valori

```
# B ;;
- : bool -> union = <fun>
# B true ;;
- : union = B true
```

B che prende un valore di tipo bool lo fa diventare di tipo union

B true : union

true : bool

B true
the union

true
the type bool

N 5
the type union

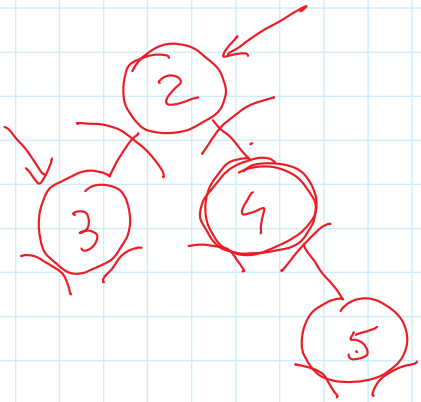
5
int

type union = B of bool | N of int ;;

Alberi binari

Alberi con queste caratteristiche:

- un albero vuoto è un albero binario
- un valore della radice, più un sottoalbero binario sinistro più un sottoalbero " destro è un albero binario



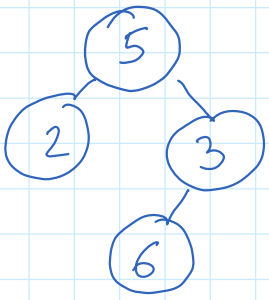
Gli elementi di un albero binario, comunemente, si chiamano nodi.
Le foglie sono i nodi con sottoalberi sinistro e destro vuoti.
Il tipo dei valori contenuti nei nodi può essere qualsiasi (tipo polimorfo)

```
type 'a btree = Void |
     Node of 'a *('a btree) *('a btree);;
```

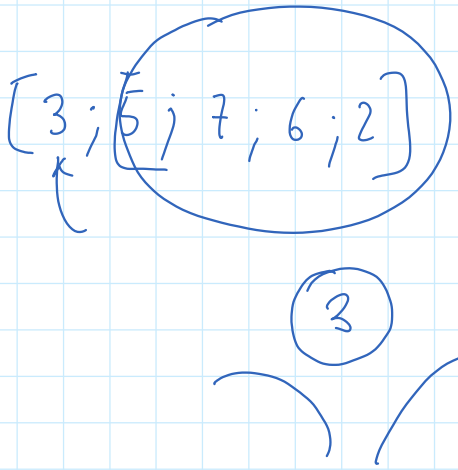
```
# Void;;
- : 'a btree = Void
```

```
# Node;;
- : 'a * 'a btree * 'a btree -> 'a btree = <fun>
```

```
# Node (5, Void, Void);;
- : int btree = Node (5, Void, Void)
```




```
Node (5, Node (2, Void, Void),
         Node (3, Node (6, Void, Void), Void));;
- : int btree = ------
```



[] ⟶ Void

```
let rec split l = match l with
    [] -> ([],[])
  | [x] -> ([x],[])
  | x::y::ys -> let (l1,l2) = split ys
                in (x::l1, y::l2);;
```

```
let ft (x,y) = x ;;
ft : 'a * 'b -> 'a = <fun>
let sd (x,y) = y ;;
sd : 'a * 'b -> 'b = <fun>
let rec split l = match l with
```

```
let rec split l = match l with
    [] -> ([],[])
  | [x] -> ([x],[])
  | x::y::ys -> let r = split ys
                in (x::fst r, y::snd r);;
split : 'a list -> 'a list * 'a list = <fun>
```

```
let rec buildt l = match l with
    [] -> Void
  | x::xs -> let (l1, l2) = split xs
             in Node(x, buildt l1,
                        buildt l2);;
```

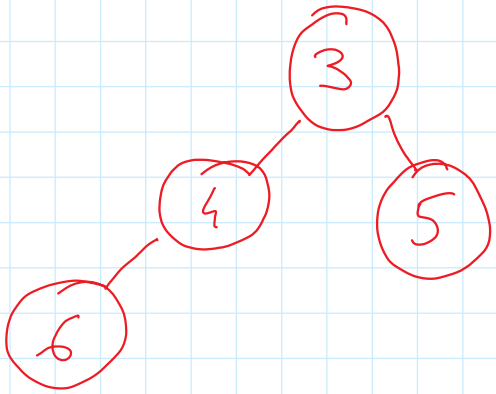
[3;4;5;6]

= Node(3, buildt [4;6], buildt [5])

= Node(3, Node(4, buildt [6], buildt []),
          Node(5, buildt [], buildt []))

= Node(3, Node(4, Node(6, buildt [], buildt []),
                  Void),
          Node(5, Void, Void))

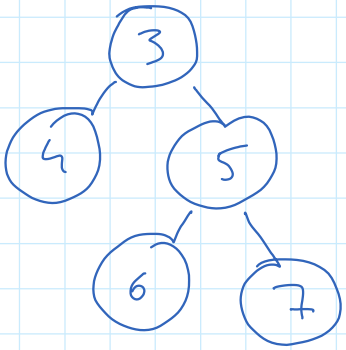
= Node(3, Node(4, Node(6, Void, Void), Void),
          Node(5, Void, Void))


3
4
5
6

Linearizzazione di un albero binario

linearizzazione anticipata

radice ; linearizzazione anticipata del sott. sinistro ;
" " " " destro

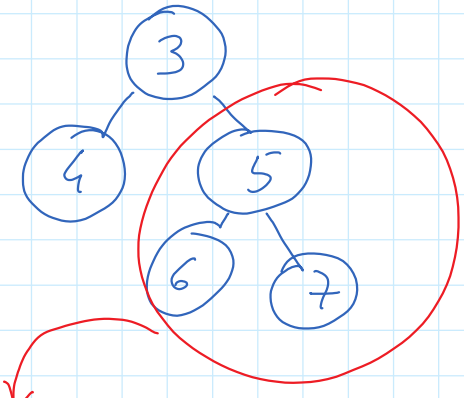


[3; 4; 5; 6; 7]

---

lin. differita

lin. differita del sott. sinistro ;
" " " " destro ;
radice

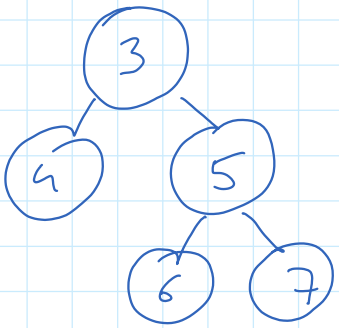


[4; 6; 7; 5; 3]

[4; 6; 7; 5; 3]

- lin simmetrica

lin. simm. sott sinistro; radice;
" " " destro



[4; 3; 6; 5; 7]

```
let rec lins bt = match bt with
    Void -> []
  | Node(x, lbt, rbt) ->
        lins lbt @ [x] @ lins rbt ;;
```

$=$

```
        lins lbt @ (x :: lins rbt) ;;
```

```
lins : 'a btree -> 'a list = <fun>
```

```
let rec lina bt = match bt with
    Void -> []
  | Node (x, lbt, rbt) ->
        x :: (lina lbt @ lina rbt);;
```

lina : 'a btree -> 'a list

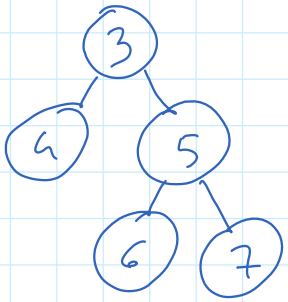
```
let rec lind bt = match bt with
    Void -> []
  | Node (x, lbt, rbt) ->
      (lind lbt) @ (lind rbt) @ [x] ;;
```

lind : 'a btree -> 'a list

Profondità: la lunghezza del massimo cammino dalla radice a una foglia

Albero vuoto ha profondità ∅

" " 1



" " 3

```
let max n m = if n > m then n
                       else m ;;
max : 'a -> 'a -> 'a = <fun>

let rec prof bt = match bt with
    Void -> ∅
  | Node (x, lbt, rbt) ->
        1 + max (prof lbt) (prof rbt) ;;

prof : 'a btree -> int
```

prof : 'a btree → int

```
let rec member el bt = match bt with
    Void -> false
  | Node (x, lbt, rbt) when el = x -> true
  | Node (x, lbt, rbt) when el <> x ->
      (member el lbt) or (member el rbt);;
```

```
if member el lbt then true
   else member el rbt ;;
```

Cercare il valore massimo di un albero non vuoto.

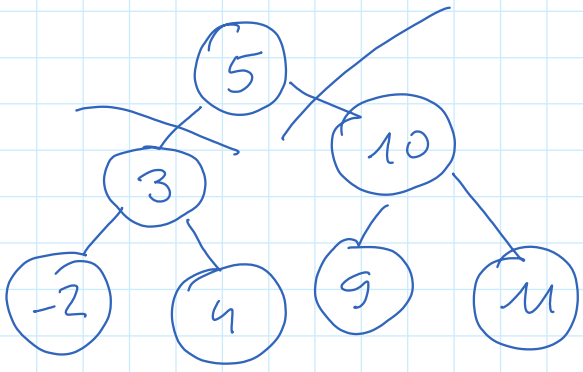
```
let rec maxv bt = match bt with
    Node(x, Void, Void) -> x
  | Node(x, Void, rbt) when rbt <> Void
        -> let m = maxv rbt
           in max x m
  | Node(x, lbt, Void) when lbt <> Void
        -> let m = maxv lbt
           in max x m
  | Node(x, lbt, rbt) when lbt <> Void &
                           rbt <> Void
        -> let m1 = maxv lbt
           in let m2 = maxv rbt
              in
                 max (max x m1) m2 ;;
```

max (max × (maxv lbt)) (maxv rbt)

Albero binario di ricerca:

Un albero binario in cui

- il valore della radice è >= a tutti i valori del sott. sinistro e < di " " " " " destro,
- i sott. sinistro e destro sono alberi binari di ricerca



[-2; 3; 4; 5; 9; 10; 11]

```
let rec member el bt = match bt with
    Void -> false
  | Node (x, lbt, rbt) when el = x -> true
  | Node (x, lbt, rbt) when el < x ->
                    member el lbt
```

| Node (x, lbt, rbt) when el > x →
member el rbt ;;

Un albero binario di ricerca
bilanciato in modo simmetrico può dà
una lista ordinata in modo
non decrescente (ci possono essere
elementi uguali)