Definizione di nuovi tipi CAML

Mediante enumerazione di nuovi valori (nomi)

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

Nuovo tipo, che si chiama giorno, i cui valori sono gli identificatori Lun, Mar, ..., Dom

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

```
# Lun < Mer ;;
- : bool = true
```

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

```
# let feriale x = x < Sab ;;
feriale : giorno -> bool = <fun>
```

```
# feriale Lun ;;
- : bool = true
```

```
# feriale Dom ;;
- : bool = false
```

Tipo unione

type unione = bool | int ;;

non è possibile!

perché, per fare inferenza di tipi, CAML deve associare a ogni valore un unico tipo

5 potrebbe avere tipo int ma anche tipo unione

type unione = N of int | B of bool;

sono costruttori di valori del tipo unione.

N è un operatore (che va lanciato / indicato) che prende un valore int e lo fa diventare un valore unione.

B è un operatore (" " ") che prende un valore bool

11) che prende un valore bool e lo fa diventare un valore di tipo unione

```
# type unione = N of int | B of bool ;;

# N ;;
- : int -> unione = <fun>

# B ;;
- : bool -> unione = <fun>
```

costruttori di valori

```
# 5 ;;
- : int = 5

# N 5 ;;
- : unione = N 5
```

```
# let f x =
    match x with
    (N n) -> n ;;
```

costruttori di valori si possono usare nei pattern

```
f : unione -> int = <fun>
```

```
let f x = match x with
    N n -> n
  | B b -> b ;;
```

errore di tipo

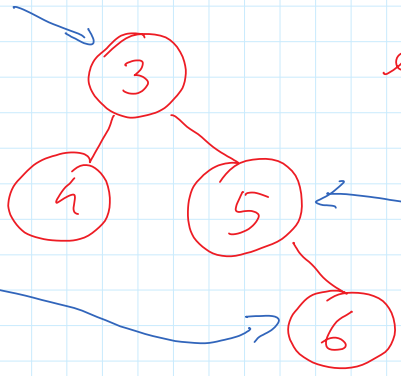
f : unione -> ?

Albero binario

1) Un albero vuoto è un albero binario

2) Un nodo (contenente informazione) con due sottalberi (sinistro e destro) che sono alberi binari, è un albero binario.

Es:



è un albero binario

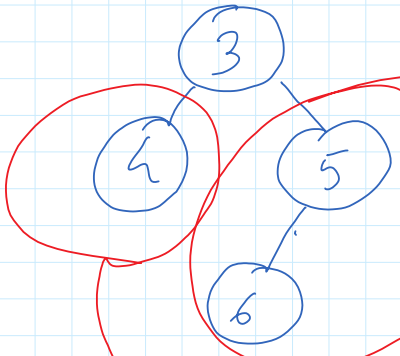
Un albero binario in cui il sott. sinistro e il sott. destro sono vuoti

Un albero binario in cui il sott. sinistro è vuoto e quello destro no!

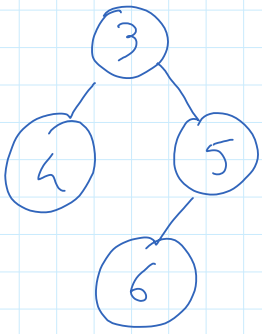
```
type 'a btree = Void | Node of
                'a * 'a btree * 'a btree ;;
```

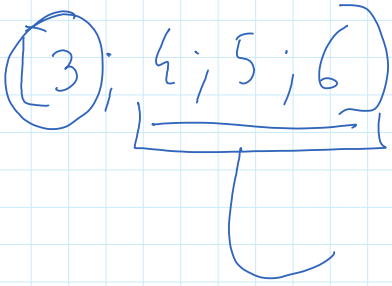
tipo che mi chiama btree

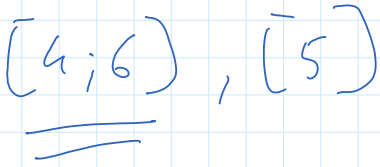
parametro (l'informazione dei nodi è di tipo generico 'a)

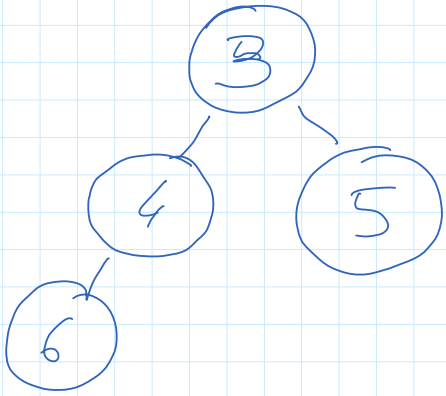



```
# Node (3, Node (4, Void, Void),
          Node (5, Node (6, Void, Void), Void)
        ) ;;
- : int btree = Node( ..... )
```










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

```
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
  [] -> ([],[])
| [x] -> ([x],[])
| x::y::ys -> let c = split ys
              in (x::ft c, y::sd c);;

split : 'a list -> 'a list * 'a list
```

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

buildt : 'a list → 'a btree =< fun >

tipo l — tipo ris

buildt [3;4;5;6];;

= { def buildt }

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

split [4;5;6] = ([4;6],[5])

split [6] = ([6],[])

= { " }

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

Node(5, buildt [], buildt []))

= { def buildt }

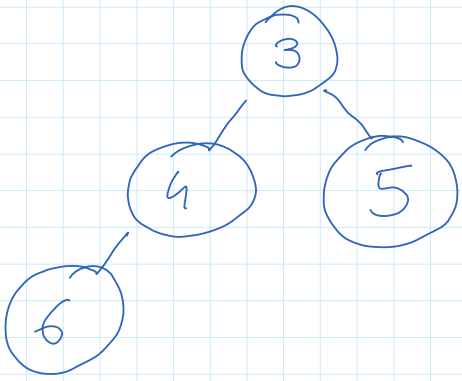
Node(3, Node(4, Node(6, buildt [], buildt []), Void), Void)

, Void),

Node (5, Void, Void)

=

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



← [3; 4; 5; 6]

Dato un albero binario costruire una lista.

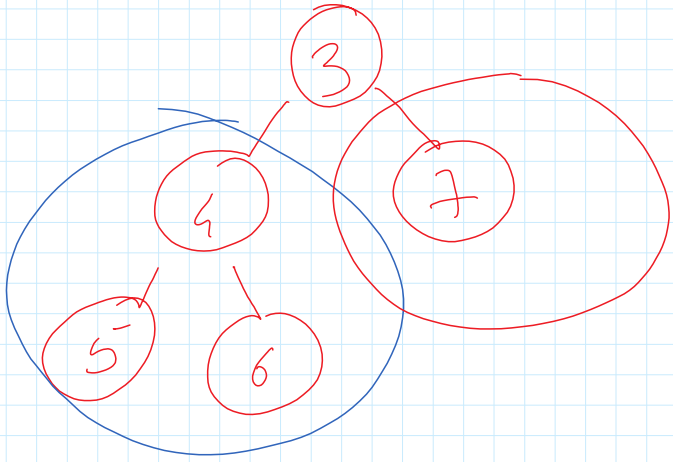
LINEARIZZAZIONE di un albero binario.

3 modi classici

linearizzazione anticipata

Costruire una lista inserendo gli elementi dell'albero nel modo seguente:

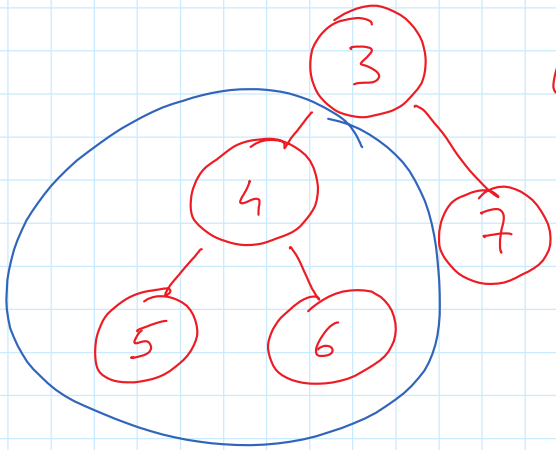
- prima la radice
- la linearizzazione anticipata del sottoalbero sinistro.
- la " " del sottoalbero destro.



[3; 4; 5; 6; 7] linearizzazione anticipata

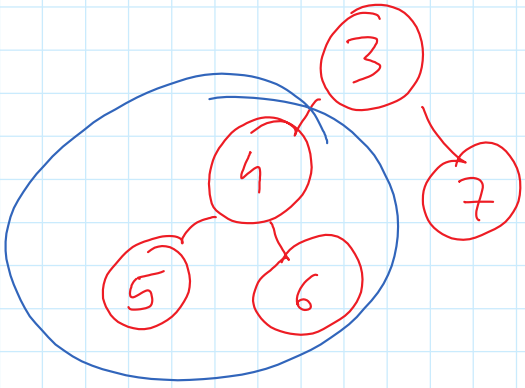
linearizzazione differita

- prima le l.d. del sott. sinistro
- poi le l.d. " " destro
- infine la radice



[5; 6; 4; 7; 3]

- linearizzazione simmetrica
  - prima le l.s. del sott. sinistro
  - poi la radice
  - infine le l.s. del sott. destro



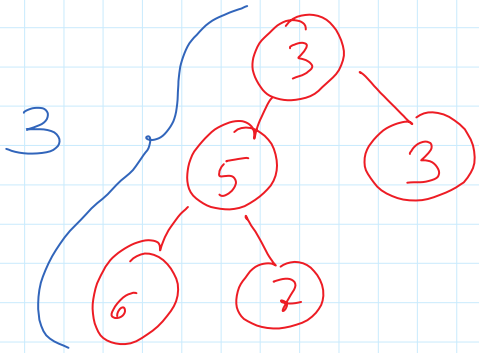
[5; 4; 6; 3; 7]

```
let rec lima bt =
        match bt with
    Void -> []
  | Node(x, lbt, rbt) ->
        x::(lima lbt @ lima rbt);;
lima : 'a btree -> 'a list = <fun>

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

lind : 'a btree -> 'a list = <fun>

let rec lins bt =
        match bt with
    Void -> []
```

```
| Node(x, lbt, rbt) ->
    (lins lbt) @ (x :: lins rbt);;
lins : 'a btree -> 'a list = <fun>
```

Profondità.

Numero minimo di nodi dalla radice a una foglia.



profondità: 3

l'albero vuoto ha profondità ∅

```
let max x y =
    if x < y then y
    else x ;;
max: 'a -> 'a -> 'a = <fun>

let rec profondita bt =
    match bt with
    Void -> ∅
  | Node (x, lbt, rbt) ->
      let p1 = profondita lbt
```

```
let p1 = profondita lst
and   p2 = profondita rst
in
(max p1 p2) + 1 ;;
profondita : 'a btree -> int = <fun>
```

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

```
memberbt : 'a -> 'a btree -> bool = <fun>
```

```
-> if memberbt el lbt then true
   else memberbt el rbt ;;
```

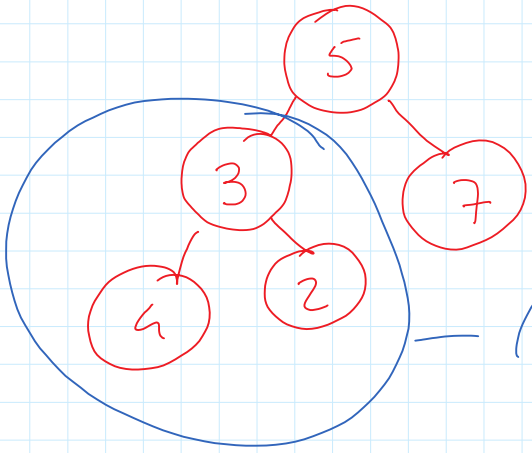
Cercare il valore massimo in un albero binario NON VUOTO

```
let rec maxbt bt =
        match bt with
    Node(x, Void, Void) -> x
  | Node(x, Void, rbt) when rbt <> Void
        -> max x (maxbt rbt)
  | Node(x, lbt, Void) when lbt <> Void
        -> max x (maxbt lbt)
  | Node(x, lbt, rbt) when lbt <> Void
                       & rbt <> Void
        -> max x (max (maxbt lbt)
                      (maxbt rbt));;

maxbt : 'a btree -> 'a
```

lunedì 9 dicembre 2019 10:52

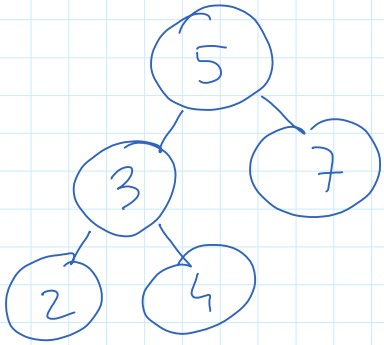
Albero binario di ricerca

- il valore della radice è maggiore o uguale a tutti i valori del sottoalbero sinistro ed è minore di tutti i valori del sottoalbero destro.
- il sottoalbero sinistro e il sottoalbero destro sono alberi binari di ricerca.



Non è un albero bin di ricerca

— non è un a.b.r.



E' un albero binario di ricerca

```
let rec memberbt el bt =
    match bt with
      Void -> false
    | Node(x, lbt, rbt) when el = x -> true
    | Node(x, lbt, rbt) when el < x
                -> memberbt el lbt
    | Node(x, lbt, rbt) when el > x
                -> memberbt el rbt ;;
memberbt : 'a -> 'a btree -> bool = <fun>
```