DEFINIZIONE DI NUOVI TIPI

Tipo enumerazione

```
# type giorno = Lun | Mar | Mer | Gio | Ven | Sab | Dom ;;
type giorno defined
```

nome del tipo → giorno

valori del tipo → Lun | Mar | Mer | Gio | Ven | Sab | Dom

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

```
# 10 ;;
- : int = 10
```

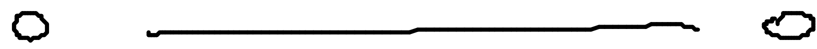
L'ordine in cui elenchiamo i valori induce <, =, <>, >, <=, >=

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

```
# let feriale x = (x >= Lun) & (x <= Ven);;
feriale : giorno -> bool = <fun>
```

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

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



Tipi definiti sulla base di altri tipi

```
# type tag = Tagn of int | Tagb of bool;;
type tag defined
```

costruttori di valori di tipo tag

```
# Togn 3;;
- : tog = Togn 3

# Togb true;;
- : tog = Togb true

# Togn;;
Togn : int -> tog = <fun>

# Togb;;
Togb : bool -> tog = <fun>
```

TIPI POLIMORFI

Il nome del tipo è preceduto da una ennupla di variabili di tipo

```
# type ('a, 'b) pair = Pair of 'a * 'b
  type pair defined

# Pair ;;
  Pair : 'a * 'b -> ('a, 'b) pair = <fun>
# Pair (5, "pippo") ;;
  - : (int, string) pair = (5, "pippo")
```

TIPI RICORSIVI

```
# type nat = zero | Succ of nat ;;
```

$$N = \{\emptyset\} \cup \{s(x) \mid x \in N\}$$

$$N^1 = \{\emptyset\} \qquad N^2 = \{\emptyset, s(\emptyset)\} \qquad N^3 = \{\emptyset, s(\emptyset), s(s(\emptyset))\}$$

```
# let foo x = match x with
        zero -> false |
    succ (y) -> true ;;

  foo : nat -> bool = <fun>
```

Succ : nat → nat

ESPRESSIONI

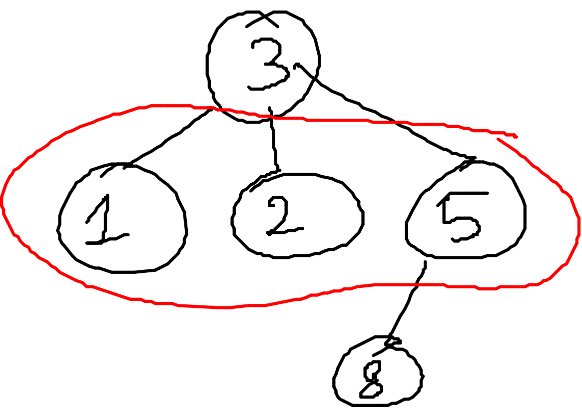
```
# type aop = plus | minus | times | div

# type aexp = Num of int |
              Exp of aexp * aop * aexp

Exp ::= Num | Exp op Exp
op ::= * | + | - | /
```

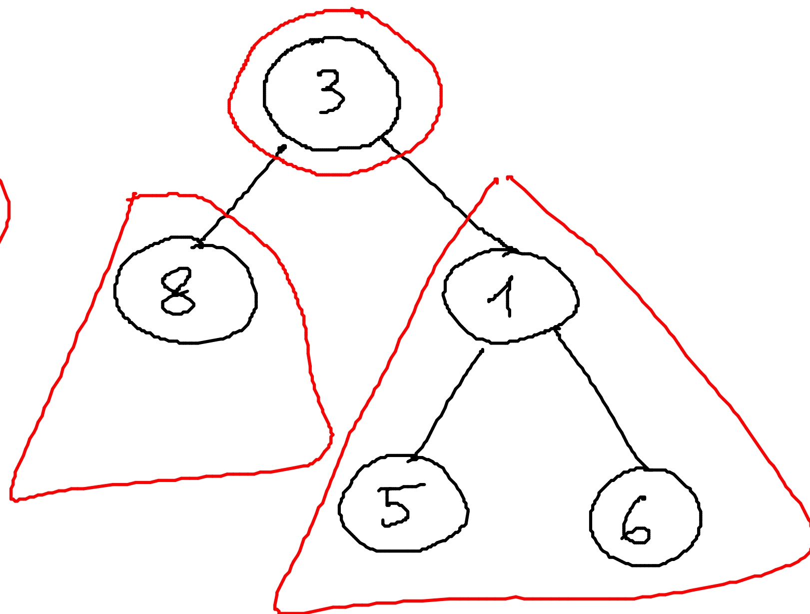
Rappresentazione di 3 + 4

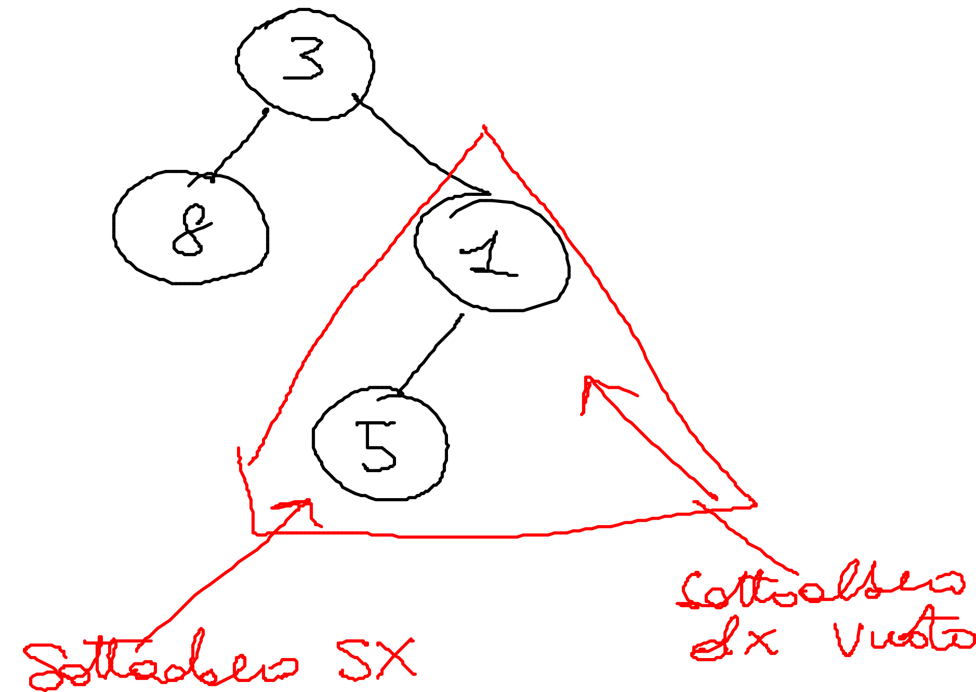
```
# Exp (Num (3), plus, Num (4));;
- : aexp = " "

# let rec eval e = match e with
      Num (x) -> x
  | Exp (e1, plus, e2) -> (eval e1) + (eval e2)
  | Exp (e1, minus, e2) -> (eval e1) - (eval e2)
        :
                                      ;;
eval : aexp -> int = <fun>
```

ALBERI BINARI

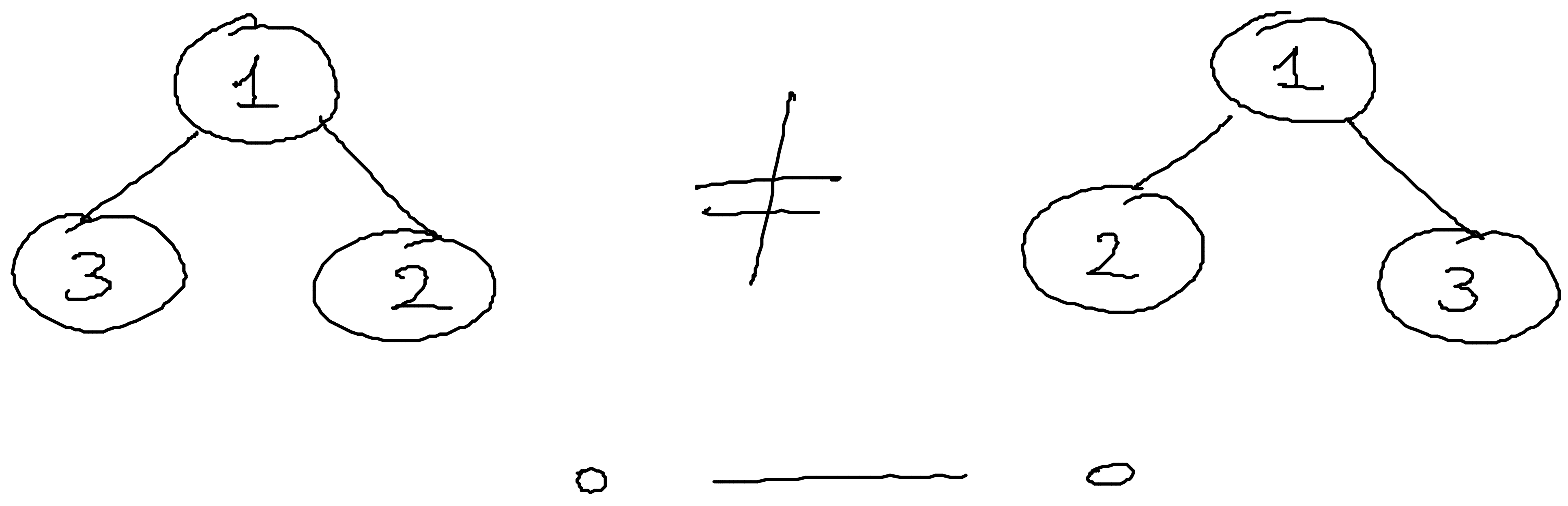
- albero vuoto
- nodo (radice) che contiene un'informazione e DUE (sotto)alberi binari (sottoalbero sinistro e sottoalbero destro)



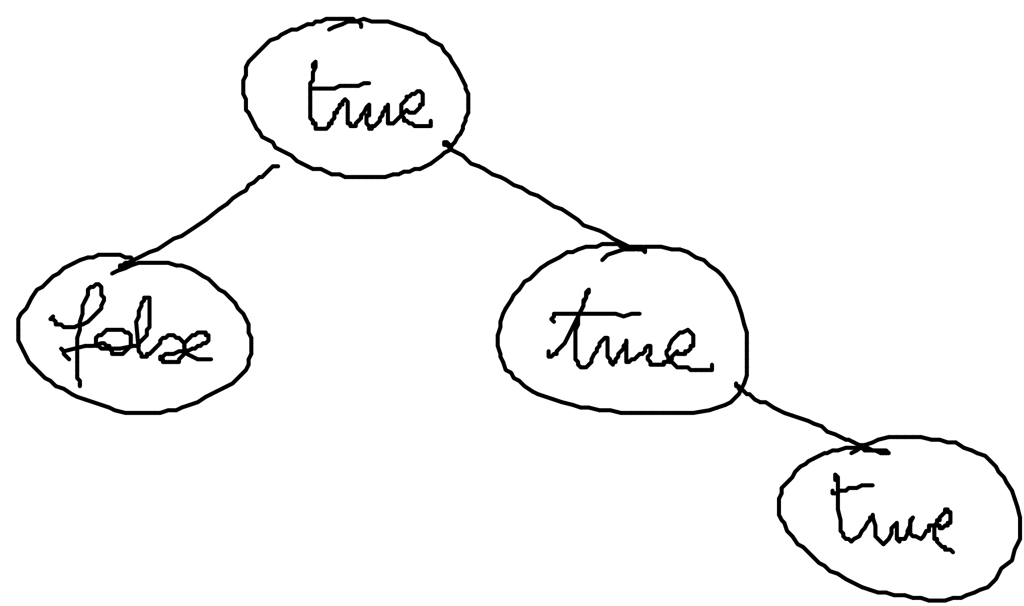


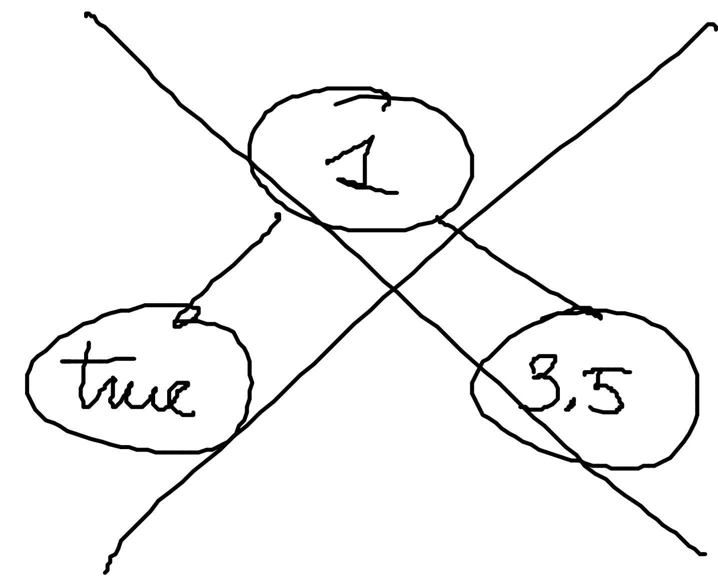


I sottoalberi sono ORDINATI

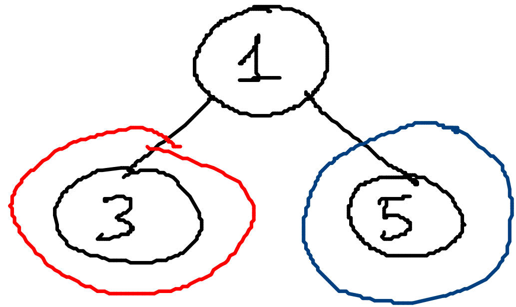


L' informazione contenuta nei nodi può essere di qualunque tipo





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




```
Node (1, Node (3, Void, Void), Node (5, Void, Void))
```

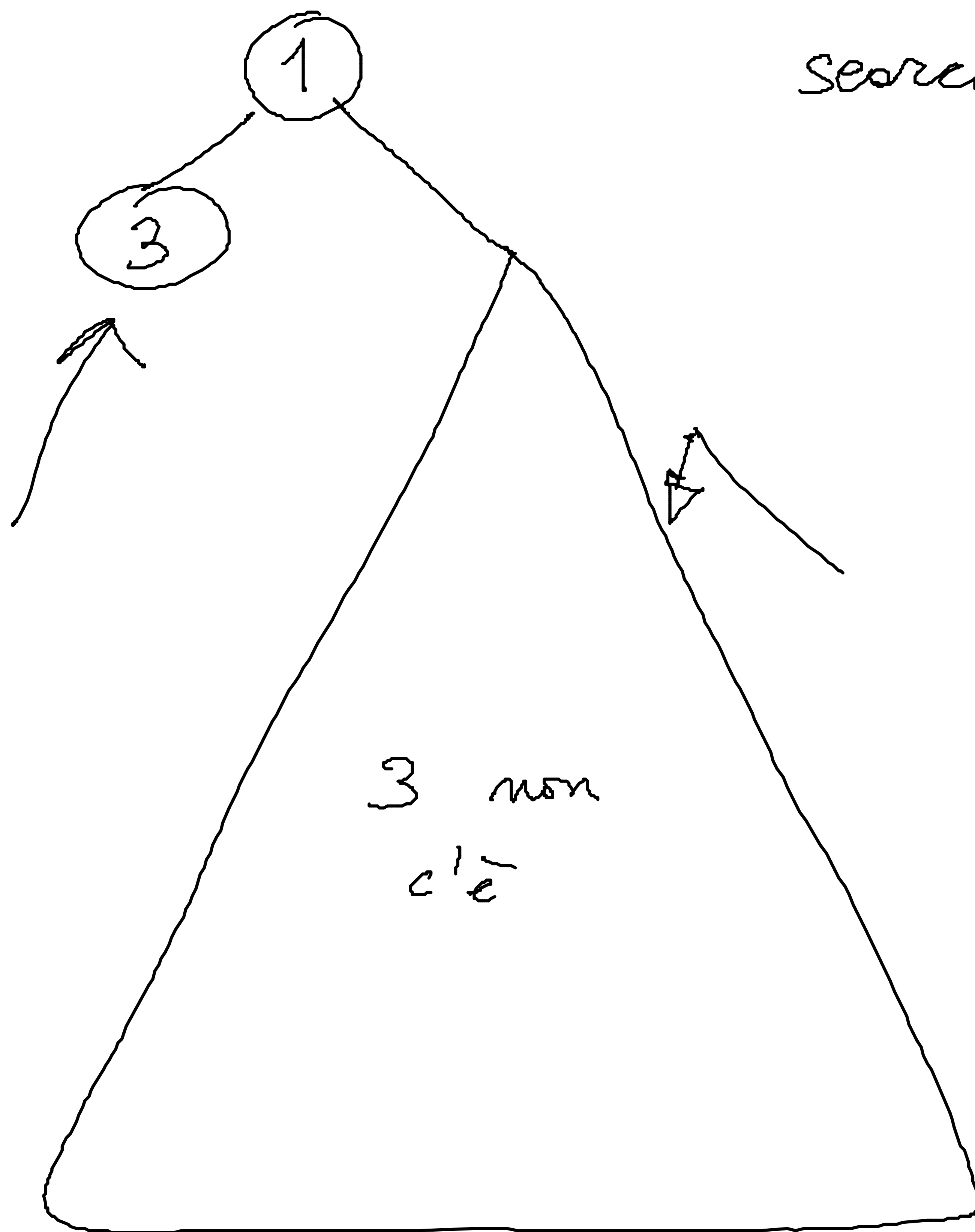
Ricerca di un valore nell'albero

```
# let rec search el bt = match bt with
        Void -> false |
    Node (x, lt, rt) when x = el -> true |
    Node (x, lt, rt) when x <> el ->
            (search el lt) or (search el rt);;
  search : 'a -> 'a btree -> bool = <fun>
```

```
# let rec member el l = match l with
        [] -> false |
      y::ys -> if y = el then true else member el ys;;
```

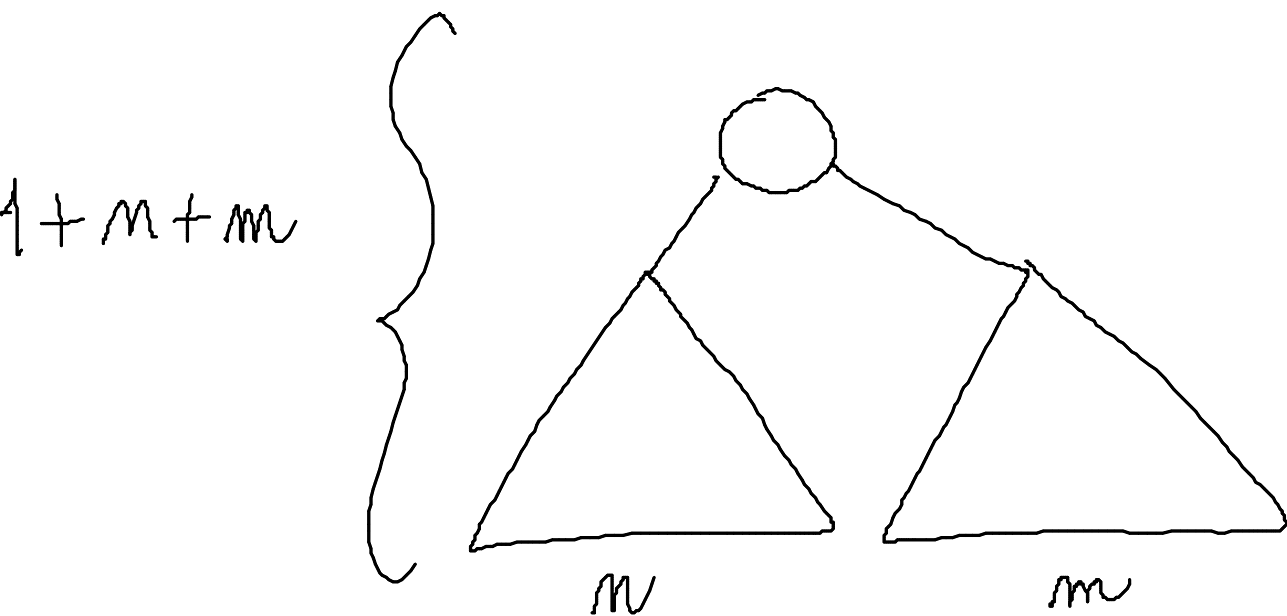
bt

search 3 bt



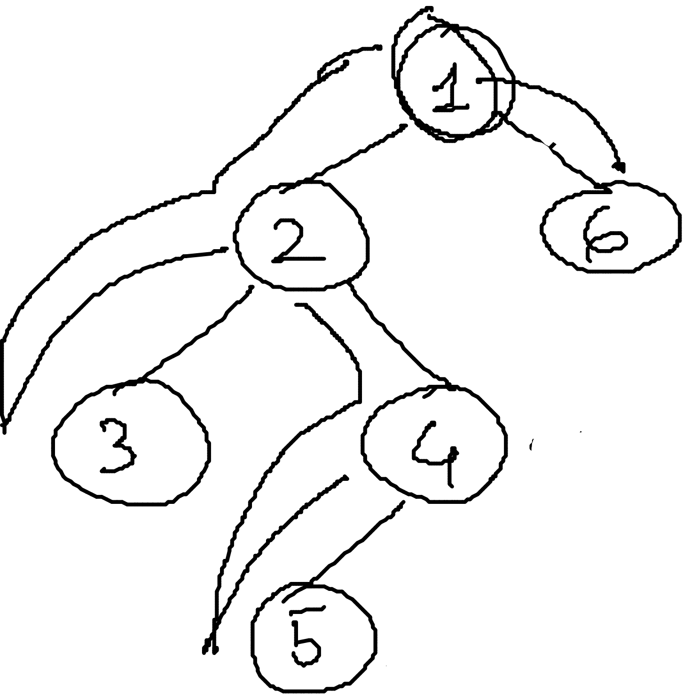
Numero di nodi di un albero (cfr. length su liste)

```
# let rec count bt = match bt with
      Void -> 0 |
    Node (_, lt, rt) -> 1 + (count lt) + (count rt);;

                  count : 'a btree -> int = <fun>
```



LINEARIZZAZIONE DI UN ALBERO

Dato un albero binario costruire una lista che contiene tutti i valori nei nodi dell'albero

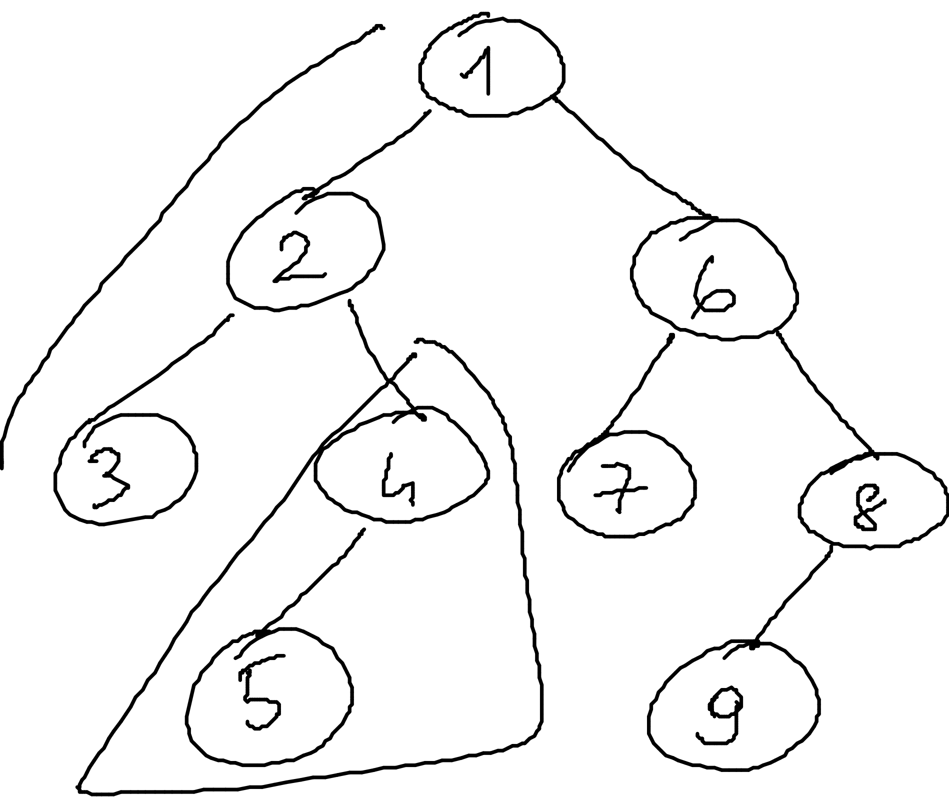


[1; 2; 3; 4; 5; 6] ANTICIPATA

[1; 2; 3; 4; 5; 6]

[3; 2; 5; 4; 1; 6] SIMMETRICA

→ sottoalbero sinistro +
radice +
sottoalbero destro



ANTICIPATA

radice + sinistro + destro

[1; 2; 3; 4; 5; 6; 7; 8; 9]

SIMMETRICA

sin + radice + destro

[3; 2; 5; 4; 1; 7; 6; 9; 8]

POSTICIPATA

sin + des + radice

[3; 5; 4; 2; 7; 9; 8; 6; 1]

ANTICIPATA

```
let rec prelin bt = match bt with
    Void -> []
  | Node (x, lt, rt) -> x::((prelin lt)@(prelin rt))

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

radice — sottoalbero sin — sottoalbero destro

```
# let rec postlin bt = match bt with
        Void -> [ ] |
 Node (x, lt, rt) -> (postlin lt) @ (postlin rt) @ [x];;

# let rec simlin bt = match bt with
        Void -> [] |
 Node (x, lt, rt) -> (simlin lt) @ [x] @ (simlin rt);;
```

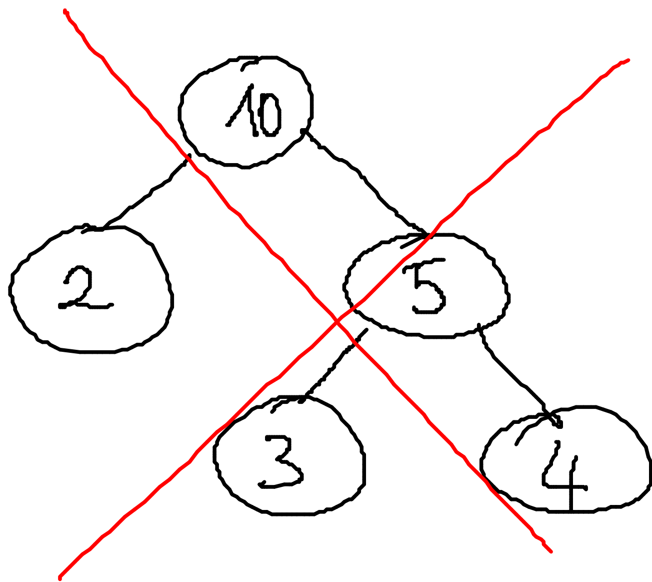
ALBERI BINARI DI RICERCA

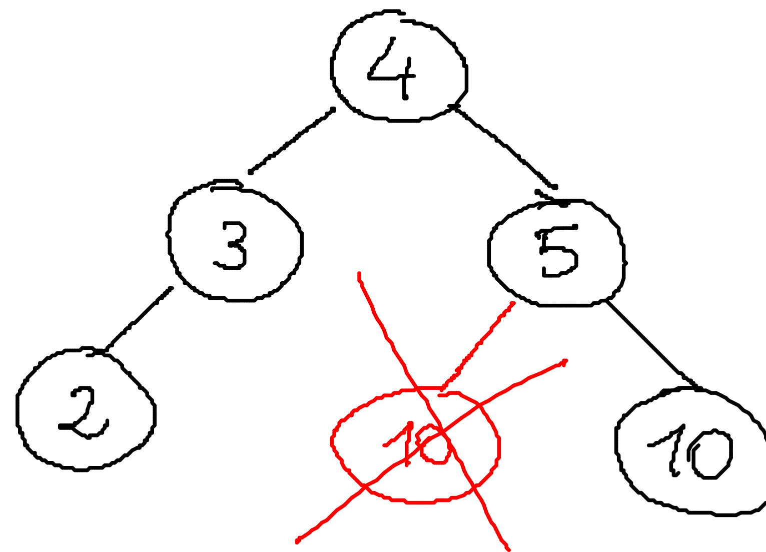
1. la radice è $>$ di tutti i nodi del sottoalbero sinistro

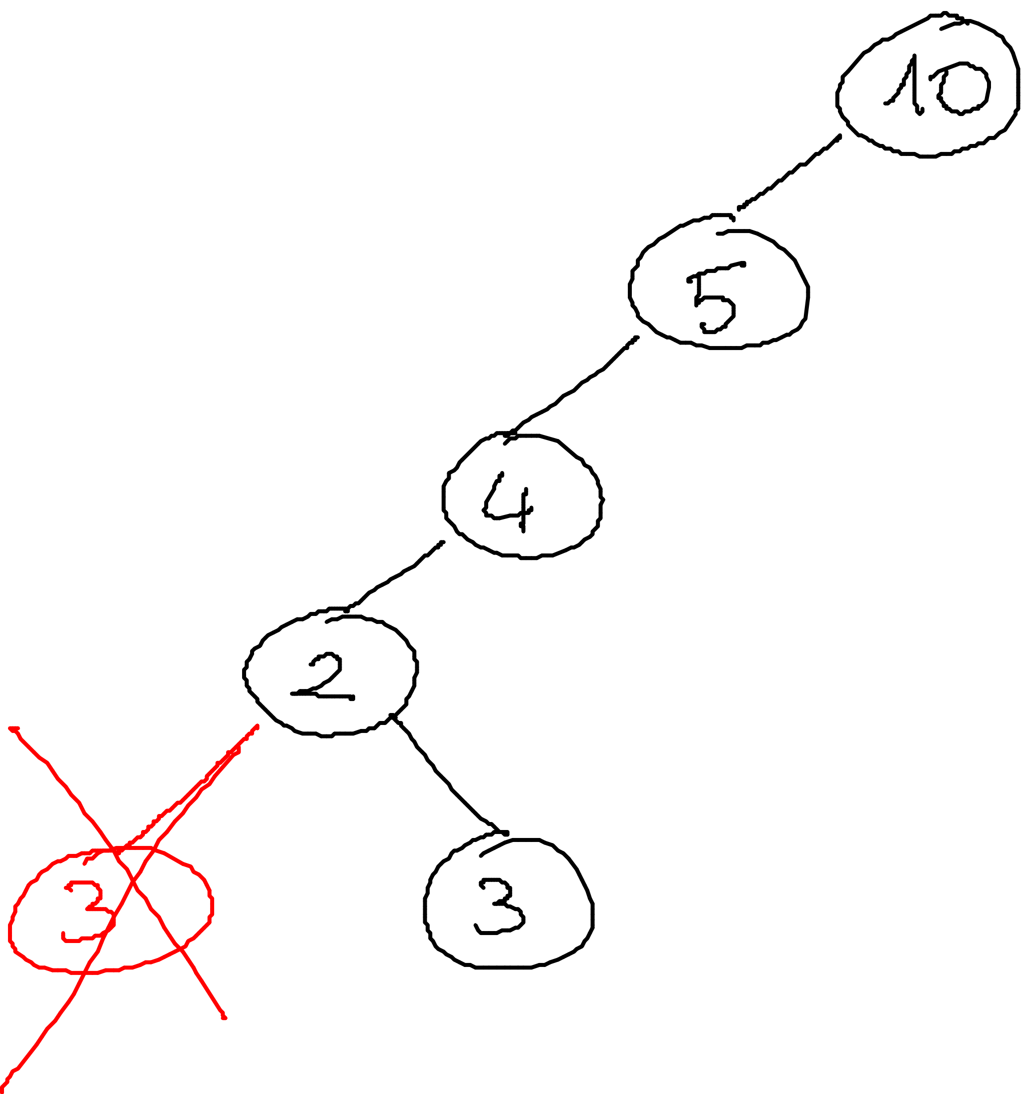
2. la radice è $<$ di tutti i nodi del sottoalbero destro

- tutti i sottoalberi soddisfano 1 e 2

No!






10
5
4
2
3
3

Ricerca di un elemento in un albero binario di ricerca

```
# let rec binsearch el sbt = match sbt with
        Void -> false |
  Node (x, _, _) when x = el -> true |
  Node (x, lt, _) when el < x -> binsearch el lt
  Node (x, _, rt) when el > x -> binsearch el rt
```