Albero binario di ricerca

- la radice ha un valore >= di tutti i valori nel sotto albero sinistro e < di tutti quelli nel sott. destro
- i sottoalberi sinistro e destro sono alb. bin. di ricerca

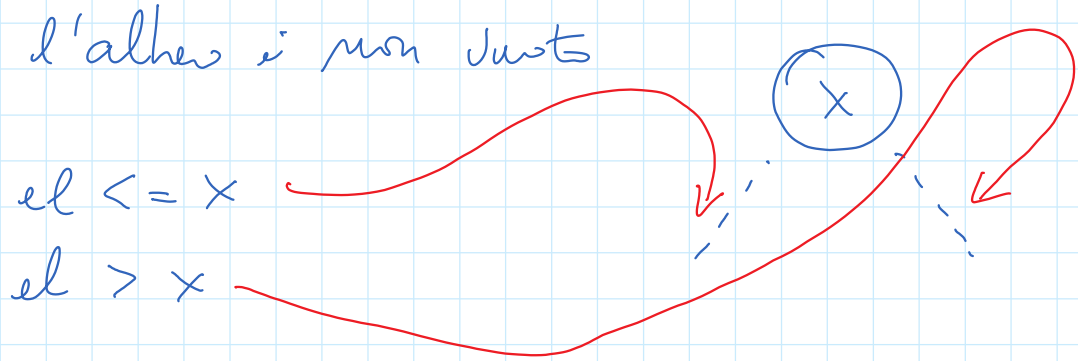
```
let rec member el bt = match bt with
    Void -> false
  | Node (x, lbt, rbt) when x = el -> true
  | Node (x, lbt, rbt) when x <> el ->
        if el < x then member el lbt
                  else member el rbt ;;
member : 'a -> 'a btree -> bool
```

Inserimento di un valore in un albero binario di ricerca.

Intuitivamente (dobbiamo inserire el)

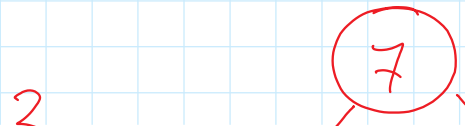
Se l'albero è vuoto → Node(el, Void, Void)

Se l'albero è non vuoto

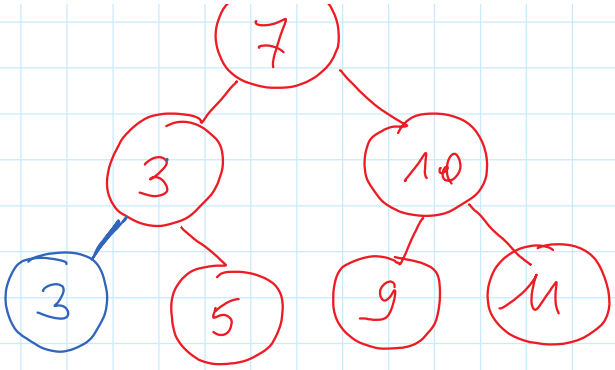



```
let rec ins el bt = match bt with
    Void -> Node(el, Void, Void)
  | Node(x, lbt, rbt) ->
      if el <= x
      then Node(x, ins el lbt, rbt)
      else Node(x, lbt, ins el rbt);;
```

ins : 'a -> 'a btree -> 'a btree



3



Data una lista costruire un alb. bin di ricerca con i valori della lista.

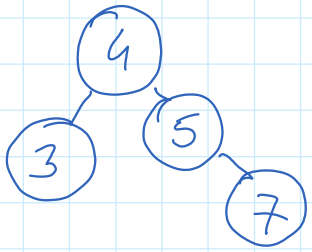
[3;7;5;4]

[7;5;4)

[5;4]

[4]

[]

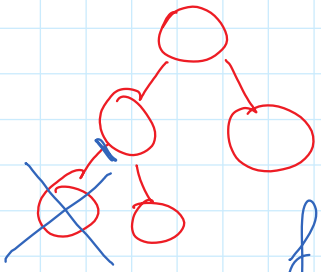


```
let rec buildt l = match l with
    [] -> Void
  | x::xs -> ins x (buildt xs) ;;

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

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

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

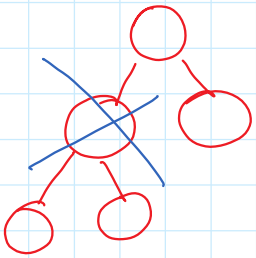
```
let sort l = fins (midt l);;
sort : 'a list -> 'a list =<fun>
```

sort l ordina la lista l in modo non decrescente

canc el bt
cancella tutte le foglie di bt che contengono el



facile!



Complicato

```
let rec canc el bt = match bt with
        Void -> Void
      | Node(x, Void, Void) ->
          if x = el then Void
          else Node(x, Void, Void)
      | Node(x, lbt, rbt) when lbt <> Void or
                               rbt <> Void ->
          Node(x, canc el lbt, canc el rbt);;
```

bt ok!

canc : 'a -> 'a btree -> 'a btree = <fun>

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

avere già definito (non occorre ridefinirle)
map, filter, foldr, forall, exists

Usando foldr

prec : int list → int → int → bool

che data una lista di interi, l, e due interi n e m, restituisce true se nella lista tutte le occorrenze di n precedono tutte le occorrenze di m; false altrimenti.

$[x_1, x_2, \ldots, x_n]$ ← foldr

$f\ x_1\ (f\ x_2\ \ldots\ (f\ x_n\ a)\ \ldots)$

(true, false)
q
true

```
let prec l n m =
    let f x (r, b) =
        if not r then (r, b)
        else if x = n then (r, true)
        else if x = m then
```

```
else if x = m then
    if b then (false, b)
         else (n, b)
    else (n, b)
in let (a, b) = fold f (true, false) e
in a ;;
```

Usando foldr

listesomme : int list → int list

che, data una lista di interi, restituisce la

lista delle somme degli elementi che si trovano tra due occorrenze del valore ∅.

Ad esempio

listesomme [1; 2; ∅; 3; 4; ∅; 5; ∅; 8] =

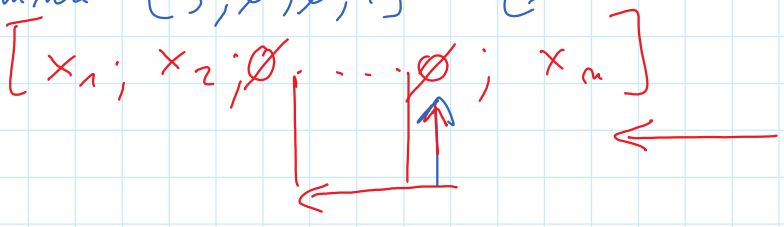
[7; 5]

listesomme [1; 2; 3; ∅] = []

listesomme [∅; 2; 3; ∅; 4] = [5]

listesomme [3; 4] = []

listesomme [3; ∅; ∅; 4] = [∅]



(l, s, b)

l: lista risultato

s: somma degli elementi che incontriamo

b: abbiamo incontrato uno ∅?

```
let listesomme l =
    let f x (l1, s, b) =
        if x = 0 then
            if not b then (l1, s, true)
                     else (s::l1; 0, b)
        else if b then (l1, s+x, b)
                  else (l1, s, b)
    in let (a,b,c) = foldr f ([],0,false) l
    in a ;;
```

Usando foldr

Remuno : int list → bool
che data una lista di interi restituisca true se la lista contiene solo occorrenze di 0 e 1 ed è ordinata in modo non decrescente (prima tutti gli 0 e poi tutti gli 1), false altrimenti.

(b1 , b2)
va tutto bene
ho incontrato uno 0

```
let remuno l =
    let f x (b1, b2) =
        if not b1 then (b1, b2)
        else if x <> 0 & x <> 1 then (false, b2)
             else if x = 0 then (b1, true)
                  else if b2 then (false, b2)
                       else (b1, b2)
    in let (b1, b2) = foldr f (true, false) l
    in b1 ;;
```

Con ricorsione esplicita

zerouno : int list → bool

che data una lista di interi, restituisca true se la lista contiene solo occorrenze di 0 e 1 e nella lista ci sono tanti 0 quanti 1, false altrimenti.

```
let rec check l = match l with
        [] -> true
      | x::xs when x<>0 & x<>1 -> false
      | x::xs when x=0 or x=1 ->
                    check xs ;;
```

check : int list → bool

```
let rec conta el l = match l with
        [] -> 0
      | x::xs when x=el -> 1 + conta el xs
      | x::xs when x<>el -> conta el xs ;;
```

conta : 'a → 'a list → int

```
let zerouno l =
```

```
let rec uno l =
  check l && (cnte 0 l) = (cnte 1 l) ;;
```

```
let numeri l =
    let rec num l a0 a1 = match l with
          [] -> a0 = a1
        | x :: xs when x <> 0 & x <> 1 -> false
        | 0 :: xs -> num xs (a0+1) a1
        | 1 :: xs -> num xs a0 (a1+1)
    in num l 0 0 ;;
```

```
let zeroumo l =
  let rec zu l = match l with
      [] -> (0, 0, true)
    | x::xs when x<>0 & x<>1 -> (0, 0, false)
    | 0::xs -> let (a,b,c) = zu xs
               in (1+a, b, c)
    | 1::xs -> let (a,b,c) = zu xs
               in (a, b+1, c)
  in let (a,b,c) = zu l
     in c & a=b ;;
```