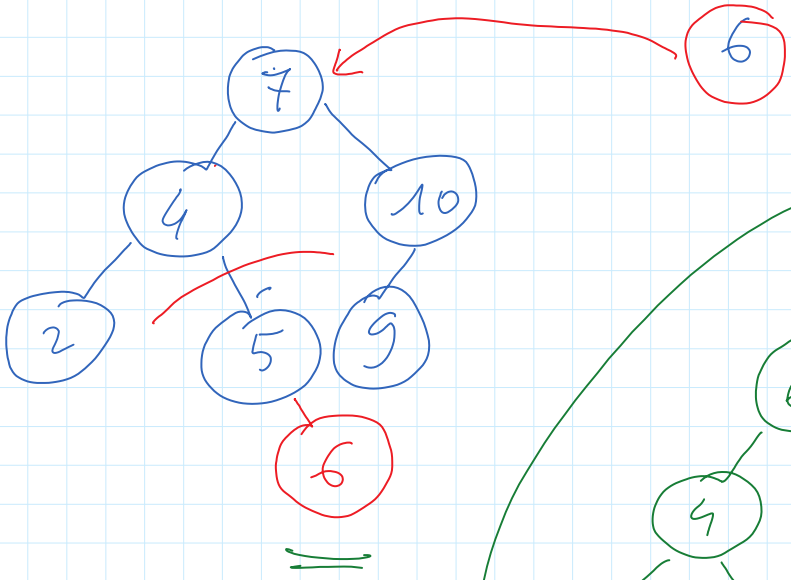
Alberi binari di ricerca

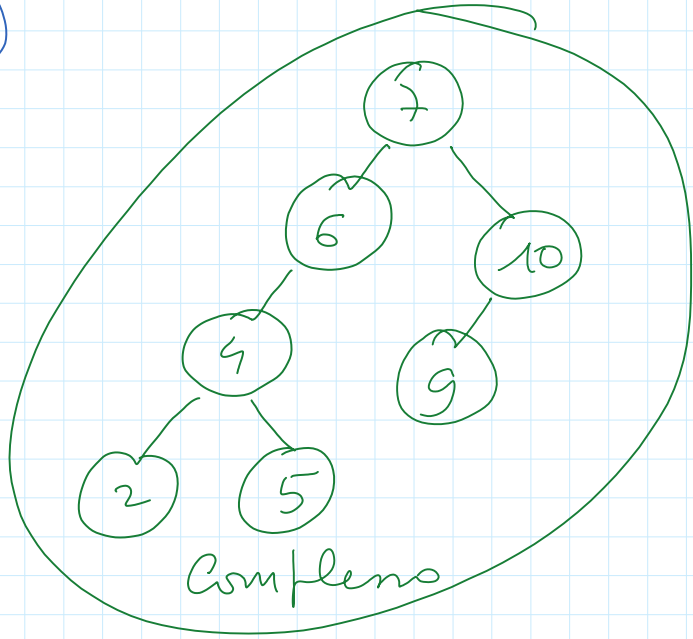
- la radice ha un valore maggiore o uguale a tutti i valori del sott. sinistro e minori di tutti i valori del sott. destro.
- I sott. sinistro e destro sono alberi binari di ricerca

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

mercoledì 11 dicembre 2019 11:22

Inserimento di un valore in un albero binario di ricerca

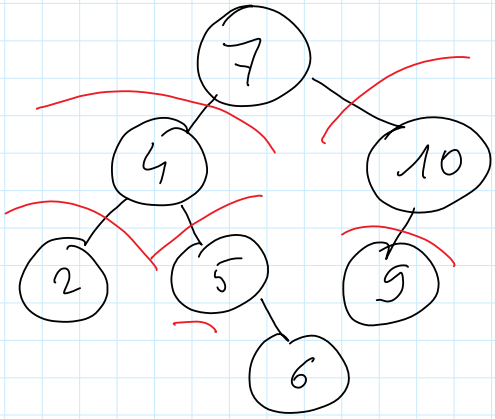




```
let rec insbt el bt =
        match bt with
    Void -> Node(el, Void, Void)
  | Node(x, lbt, rbt) when el <= x ->
        Node(x, insbt el lbt, rbt)
  | Node(x, lbt, rbt) when el > x ->
        Node(x, lbt, insbt el rbt);;
insbt : 'a -> 'a btree -> 'a btree = <fun>
```

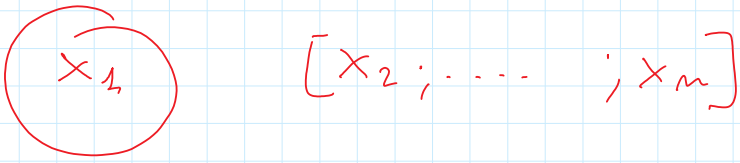
Possiamo utilizzare un a.b.r. per ordinare una lista?

Se faccio la linearizzazione simmetrica di un a.b.r. ottengo una lista ordinata (in modo non decrescente)



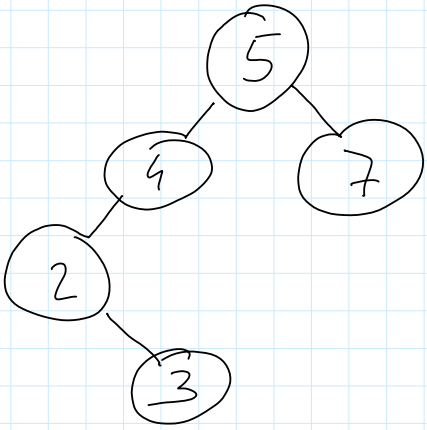
[2; 4; 5; 6; 7; 9; 10]

$[x_1; x_2; \ldots; x_n]$




```
let rec buildsbt l =
        match l with
          [] -> Void
        | x::xs -> insbt x (buildsbt xs);;
buildsbt : 'a list -> 'a btree = <fun>
```

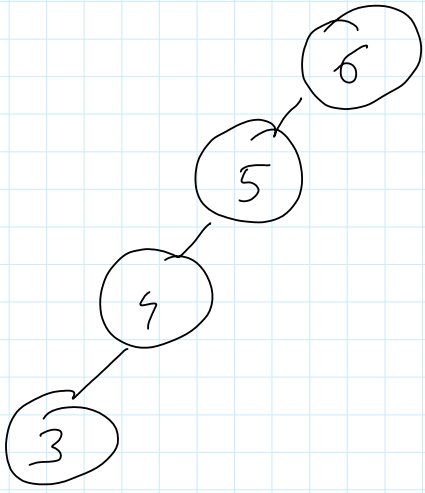
```
buildsbt [3;7;2;4;5]
= {def ....}
insbt 3 (buildsbt [7;2;4;5])
        :
insbt 3 (insbt 7 (insbt 2 (insbt 4
        (insbt 5 Void))))
```



buildsbt [3;4;5;6]

⋮

insbt 3 (insbt 4 (insbt 5 (insbt 6 Void)))



```
let sort l =
    lins (buildsbt l) ;;
```

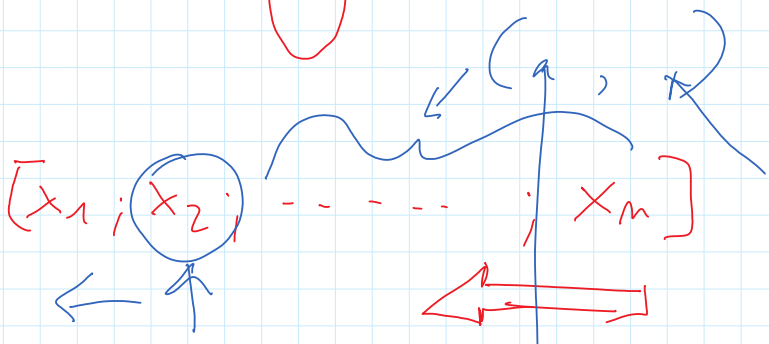
Usare le folds per definire una funzione

prec : 'a list → 'a → 'a → bool

che data una lista, l, e due valori, n e m, restituisce true sse nella lista, tutte le occorrenze di n precedono tutte le occorrenze di m.

prec [3; 4; 3; 5; 2; 10; 2] 3 2 = true

prec [3; 4; 2; 3; 5; 2; 10; 2] 3 2 = false



n precede m ho incontrato un valore n?

(b1, b2)
↑ ==
ho incontrato x.

fin qui tutto bene.

```
let fsc l x m =
  let f x (b1, b2) =
    if not b2 then (b1, b2)
    else if x = x then (true, b2)
    else if x = m then
      if b1 then (b1, false)
      else (b1, b2)
    else (b1, b2)
  in
  let (b1, b2) = foldr f (false, true) l
  in b2 ;;
```

Usando foldr

listesomme : int list → int list

che, data una lista di interi, l, restituisce le liste delle somme dei valori compresi tra due occorrenze di ∅

listesomme [1; 2; ∅; 3; 4; ∅; 5; ∅; 8] = [7; 5]

7 5

listesomme [∅; 2; 3; ∅; 4] = [5]

listesomme [3; 4] = []

listesomme [3; ∅; ∅; 4] = [∅]

[x1; x2; . . . . xn]

somma parziale, abbiamo incontrato ∅?

(, , liste)

(somme degli elementi che precedono lo $\emptyset$ incontrato,
abbiamo incontrato $\emptyset$?,
liste rimaste)

(s, b, l1)

```
let listesomme l =
    let f x (s, b, l1) =
        if x = ∅ then
            if b then (∅, b, s::l1)
                 else (∅, true, l1)
        else
            if b then (s+x, b, l1)
                 else (s, b, l1)
    in
    let (s, b, l1) = foldbr f (∅, false, []) l
    in l1 ;;
```

Usando folds definire

zerouno : int list → bool

che data una lista di interi restituisca true sse la lista contiene solamente valori 0 e 1 ed è ordinata in modo non decrescente.

[0;0;0;1;1;] ——
[0;0;0] —— true
[1;1] ——

[0;1;0] ——
[0,3,0] —— false

(b1 , b2)

b1 ↑ ; b2 ↑ fin qui tutto bene

abbiamo incontrato 0 ?

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

Controllare che la lista contenga solo 0 e 1 e che il numero di occorrenze di 0 è uguale a quello di 1

check [0; 1; 0; 0; 1;1] = true

check [0; 2; 1] = false

check [0; 1; 0] = false

```
let rec ru l =
    match l with
    [] -> true
  | x :: xs when x <> 0 & x <> 1 -> false
  | x :: xs when x = 0 or x = 1 ->
                ru xs ;;
zu : int list -> bool = <fun>

let rec nocc el l =
    match l with
    [] -> 0
```

```
| x::xs when x=el -> 1 + nocc el xs
| x::xs when x<>el -> nocc el xs ;;
```

```
nocc : 'a -> 'a list -> int = <fun>
```

```
let check l =
  (rev l) = (nocc 1 l) = (nocc 0 l) ;;
```

```
let check l =
  let rec checka l n0 n1 =
    match l with
    [] -> n0 = n1
  | x::xs -> if x<>0 & x<>1 then false
             else if x=0
               then checka xs (n0+1) n1
               else checka xs n0 (n1+1)
in checka l 0 0 ;;
```

```
let check l =
    let rec check_aux l =
        match l with
        [] -> (0, 0, true)
      | x::xs -> if x<>0 && x<>1 then (0, 0, false)
                 else let (n0, n1, b) = check_aux xs
                      in if not b then (n0, n1, b)
                         else if x=0 then (n0+1, n1, b)
                              else (n0, n1+1, b)
    in
        let (n0, n1, b) = check_aux l
        in b && n0 = n1 ;;
```