
PROGRAMMAZIONE 2

16. OCaml: una veloce introduzione

Ocaml

OCAML è un avanzato **linguaggio di programmazione** appartenente alla famiglia di linguaggi di programmazione **ML**. È sviluppato e distribuito come *software open source* dalla **INRIA**. O'Caml è stato creato nel **1996** come successore del **CAML**.

La sigla *CAML* originariamente derivava da *Categorical Abstract Machine Language*, mentre la *O* iniziale è stata aggiunta per sottolineare l'estensione ad oggetti del nuovo linguaggio. O'Caml condivide le caratteri

OCAML

condivide le caratteristiche dei linguaggi **funzionali** e di quelli **imperativi**, ma contiene elementi di **programmazione orientata agli oggetti** e alcune differenze sintattiche.

caratteristiche molto potenti del linguaggio sono un **sistema di tipi statico**, la **deduzione del tipo delle variabili**, il **polimorfismo** parametrico, il **pattern matching**, le **first class lexical closures**, l'uso dei **functors (parametric modules)**, la **gestione delle eccezioni**, e un **garbage collection** incrementale.

Lo stile funzionale

- In Java (ma anche in C) l'effetto osservabile di un programma è la **modifica** dello stato

```
temp = pair.x;  
pair.x = pair.y;  
pair.y = temp;
```

- In **OCaML** il risultato della computazione è la **produzione** di un nuovo valore

```
let x = 5 in (x,x)
```

Value-oriented programming

- Programmazione funzionale: “value-oriented programming”
 - un programma **OCaML** è una espressione
 - una espressione **OCaML** denota un valore
- L’esecuzione di un programma **OCaML** può essere vista come una sequenza di passi di calcolo (semplificazioni di espressioni) che alla fine produce un valore

Espressioni

- **Sintassi: regole di buona formazione**
- **Semantica**
 - regole di **type checking** (tipo o errore)
 - **regole di esecuzione** che garantiscono che espressioni tipate producono un valore

Valori

- Un **valore** è una espressione che non deve essere valutata ulteriormente
 - **34** è un **valore** di tipo **int**
 - **34+17** è un'espressione di tipo **int** ma non è un **valore!!!**

Valore di una espressione

- La notazione $\langle \text{exp} \rangle \Rightarrow \langle \text{val} \rangle$ indica che l'espressione $\langle \text{exp} \rangle$ quando valutata calcola il valore $\langle \text{val} \rangle$

$3 \Rightarrow 3$ (valori di base)

$3+4 \Rightarrow 7$

$2*(4+5) \Rightarrow 18$

$\text{eval}(e) = v$ meta-notazione per $e \Rightarrow v$

Dichiarazioni: **let**

- Sintassi: **let** $x = e1$ **in** $e2$
 - x *identifier*
 - $e1$, $e2$ *espressioni*
 - **let** $x = e1$ **in** *espressione*
 - $x = e1$ *binding*
- **let** $x = 2$ **in** $x + x$
let **inc** $x = x + 1$ **in** **inc** 10
let $y = \text{"ciao"}$ **in** (**let** $z = \text{" 2"}$ **in** y^z)

Espressioni: **let**

- Sia **e** l'espressione **let x = e1 in e2**
- **Regola di valutazione**
 - **eval (e1) = v1**
 - sostituire il valore **v1** per tutte le occorrenze di **x** in **e2** ottenendo l'espressione **e2'**
 - **subst (e2 , x, v1) = e2'**
 - **eval (e2') = v**
 - **eval (e) = v**

Regola di valutazione: **let**



$$\frac{\text{eval}(e_1) = v_1 \quad \text{subst}(e_2, x, v_1) = e_2' \quad \text{eval}(e_2') = v}{\text{eval}(\text{let } x = e_1 \text{ in } e_2) = v}$$

Esempio

- `eval(let x = 1 + 4 in x * 3)`
 - `eval(1 + 4) = 5`
- `eval(let x = 5 in x*3)`
 - `subst(x * 3, x, 5) = 5 * 3`
 - `eval(5 * 3) = 15`
- `eval(let x = 1 + 4 in x * 3) = 15`

let binding = scope

```
let x = 42 in
  (* y non è visibile *)
  x + (let y = "3110" in
        (* y è visibile *)
        int_of_string y)
```

scope: overlapping

```
let x = 5 in ((let x = 6 in x) + x)
```

due casi

```
((let x = 6 in x) + 5)
```

```
((let x = 6 in 5) + 5)
```

scope: overlapping

```
let x = 5 in ((let x = 6 in x) + x)
```

due casi

```
((let x = 6 in x) + 5)
```

```
((let x = 6 in 5) + 5)
```

scope: overlapping

```
let x = 5 in ((let x = 6 in x) + x)
```

due casi

```
((let x = 6 in x) + 5)
```

```
((let x = 6 in 5) + 5)
```

Alpha conversione

- L'identità puntuale delle variabili legate non ha alcun senso!!
- In matematica
 - $f(x) = x * x$
 - $f(y) = y * y$sono la medesima funzione!!
- In informatica
 - `let x = 5 in ((let x = 6 in x) + x)`
 - `let x = 5 in ((let y = 6 in y) + x)`

Dichiarazione di funzioni

```
let f (x : int) : int =  
    let y = x * 10 in  
        y * y;;
```

Funzioni Ricorsive

```
let rec pow x y =  
  if y = 0 then 1  
  else x * pow x (y - 1);;
```

Qual'è il tipo???

Applicazione di funzioni

```
let f (x : int) : int =  
    let y = x * 10 in  
        y * y;;
```

```
f 5;;
```

```
- : int = 2500
```

Applicazione di funzioni

```
let rec pow x y =  
    if y = 0 then 1  
    else x * pow x (y - 1);;
```

```
pow 2 3;;  
-: int 8
```

Dichiarazione

- La valutazione di una dichiarazione di una funzione è la funzione stessa
 - le funzioni sono valori
 - quindi possiamo passarle come parametro ad altre funzioni
 - possono essere restituite come valore da una funzione

Applicazione di funzione

- $\text{eval}(e_0 \ e_1 \ \dots \ e_n) = v'$ se
 - $\text{eval}(e_0) = \text{let } f \ x_1 \ \dots \ x_n = e$
 - $\text{eval}(e_1) = v_1 \ \dots \ \text{eval}(e_n) = v_n$
 - $\text{subst}(e, x_1, \dots, x_n, v_1, \dots, v_n) = e'$
 - $\text{eval}(e') = v'$

Esempi

- `let double(x : int) : int = 2 * x;;`
- `let square(x : int) : int = x * x;;`
- `let quad(x : int) : int =
 double (double x);;`
- `let fourth (x : int) : int =
 square (square x);;`

Esempi

- `let twice ((f : int -> int), (x : int)) :
int = f (f x)`
- `let quad (x : int) : int =
twice (double, x)`
- `let fourth (x : int) : int =
twice (square, x)`
- *twice: higher-order function: una funzione da funzioni ad altri valori*

OCaML Liste

- `let lst = [1; 2; 3];;`
- `let empty = [];;`
- `let longer = 5::lst;;`
- `let another = 5::1::2::3::[]`

Liste: sintassi

- **[]** la **lista vuota**
 - **nil** derivato dal LISP
- **e1 :: e2** inserisce l'elemento **e1** in testa alla **lista e2**
 - **::** = LISP **cons**
- **[e1; ...; en]** notazione sintattica per la lista
e1 :: ... :: en :: []

Accedere a una lista

- Strutturalmente una lista può essere
 - `[]` la lista vuota
 - la lista ottenuta mediante una operazione di `cons` di un elemento a una lista
- *Usare il pattern matching* per accedere agli elementi della lista
- `let empty lst = match lst with`
 - `[] -> true`
 - `|h::t -> false`

Ricorsione sulle liste

- `let rec sum xs = match xs with`
 - `[] -> 0`
 - `| h::t -> h + sum t`
- `let rec concat ss = match ss with`
 - `[] -> ""`
 - `| s::ss' -> s ^ (concat ss')`
- `let rec append lst1 lst2 = match lst1 with`
 - `[] -> lst2`
 - `| h::t -> h::(append t lst2)`

Pattern Matching

- **match** e **with**

```
    p1 -> e1
  |
  | p2 -> e2
  |
  | ...
  | pn -> en
```

- Le espressioni **pi** si chiamano *pattern*

Pattern Matching

- Il pattern `[]` “match” solamente il valore `[]`

- **match** `[]` **with**

```
    [] -> 0  
    | h::t -> 1
```

(* restituisce il valore 0*)

- **match** `[]` **with**

```
    [] -> 1  
    | h::t -> 0
```

(* restituisce il valore 1*)

Pattern Matching

- Il pattern `h::t` “match” una qualsiasi lista con almeno un elemento, e inoltre ha l’effetto di legare quell’ elemento alla variabile `h` e la lista rimanente alla variabile `t`
- ```
match [1; 2; 3] with
 [] -> 0
 | h::t -> h
 (* restituisce il valore 1 *)
```
- ```
match [1; 2; 3] with  
  [] -> 0  
  | h::t -> t (* restituisce il valore [2; 3] *)
```

Altri esempi: liste

- Il pattern **a :: []** “match” tutte le liste con esattamente un elemento
- Il pattern **a :: b** “match” tutte le liste con almeno un elemento
- Il pattern **a :: b :: []** “match” tutte le liste con esattamente due elementi
- Il pattern **a :: b :: c :: d** “match” tutte le liste con almeno tre elementi

Un esempio più complicato

- ```
let rec drop_val v lst = match lst with
 [] -> []
| h::t ->
 let t' = drop_val v t in
 if h = v
 then t' else h::t'
```

# Un altro esempio

---

- `let rec max_list = function`
  - `[] -> ???`
  - `| h::t -> max h (max_list t)`
- Cosa mettiamo al posto di ??? ?
  - `min_int` è una scelta possibile ...
  - o sollevare una exception ...
- In Java, avremmo potuto restituire **null**...
  - ...ma siamo in OCaml, che ci fornisce una altra soluzione

# Option type

---

- ```
let rec max_list = function
    [] -> None
  | h::t -> match max_list t with
    | None -> Some h
    | Some x -> Some (max h x)

(* max_list : 'a list -> 'a option *)
```

Iteratori

- **let rec map f = function**
 [] -> []
 | x :: xs -> (f x) :: (map f xs)

(map : ('a -> 'b) -> 'a list -> 'b list)

Parametro implicito di tipo lista

Definire tipi di dato in OCaml

OCaml permette al programmatore di definire nuovi tipi di dato

```
type giorno =  
  Lunedì  
  | Martedì  
  | Mercoledì  
  | Giovedì  
  | Venerdì  
  | Sabato  
  | Domenica
```

Dichiarazione di tipo

Nome del tipo

Costruttori

I costruttori definiscono i valori del tipo di dato
Quindi **Sabato** ha tipo **giorno**
[Venerdì, Sabato, Domenica]
ha tipo **giorno list**

Pattern Matching

Il pattern matching fornisce un modo efficiente per accedere ai valori di un tipo di dato

```
let string_of_g (g : giorno) : string =  
  match g with  
  | Lunedì -> "Lunedì"  
  | Martedì -> "Martedì"  
  |  
  | Domenica -> "Domenica"
```

Il pattern matching **segue** la struttura sintattica dei valori del tipo di dato: i costruttori

Astrazioni sui dati

- Avremmo potuto rappresentare il tipo di dato *giorno* tramite dei semplici valori interi
 - Lunedì = 1, Martedì = 2, ..., Domenica = 7
- Ma...
 - il tipo di dato primitivo *int* fornisce un insieme di operazioni differenti da quelle significative sul tipo di dato *giorno*, Mercoledì
 - Domenica *non* avrebbe alcun senso
 - esistono un numero maggiore di valori interi che di valori del tipo *giorno*
- Morale: i linguaggi di programmazione moderni (Java, OCaml, C#, C++, ...) forniscono strumenti per definire tipi di dato

OCaML Type

- I costruttori possono anche trasportare “valori”

Dichiarazione da valutare

```
# type foo =  
  Nothing  
  | Int of int  
  | Pair of int * int  
  | String of string;;
```

Risultato

```
type foo = Nothing | Int of int | Pair of int * int |  
  String of string
```

Valori del tipo `foo`

```
Nothing  
Int 3  
Pair (4, 5)  
String "hello"...
```

Pattern matching

```
let get_count (f : foo) : int =  
  match f with  
    | Nothing -> 0  
    | Int(n) -> n  
    | Pair(n,m) -> n + m  
    | String(s) -> 0
```

Cosa calcola???

```
Nothing  
Int 3  
Pair (4, 5)  
String "hello"...
```

Tipi di dato ricorsivi

```
# type tree =  
    Leaf of int  
  | Node of tree * int * tree;;  
  
type tree = Leaf of int | Node of tree * int * tree
```

```
let t1 = Leaf 3  
let t2 = Node(Leaf 3, 2, Leaf 4)  
let t3 = Node(Leaf 3, 2, Node(Leaf 5, 4, Leaf 6))
```

Quali alberi rappresentano???

Tipi di dato ricorsivi

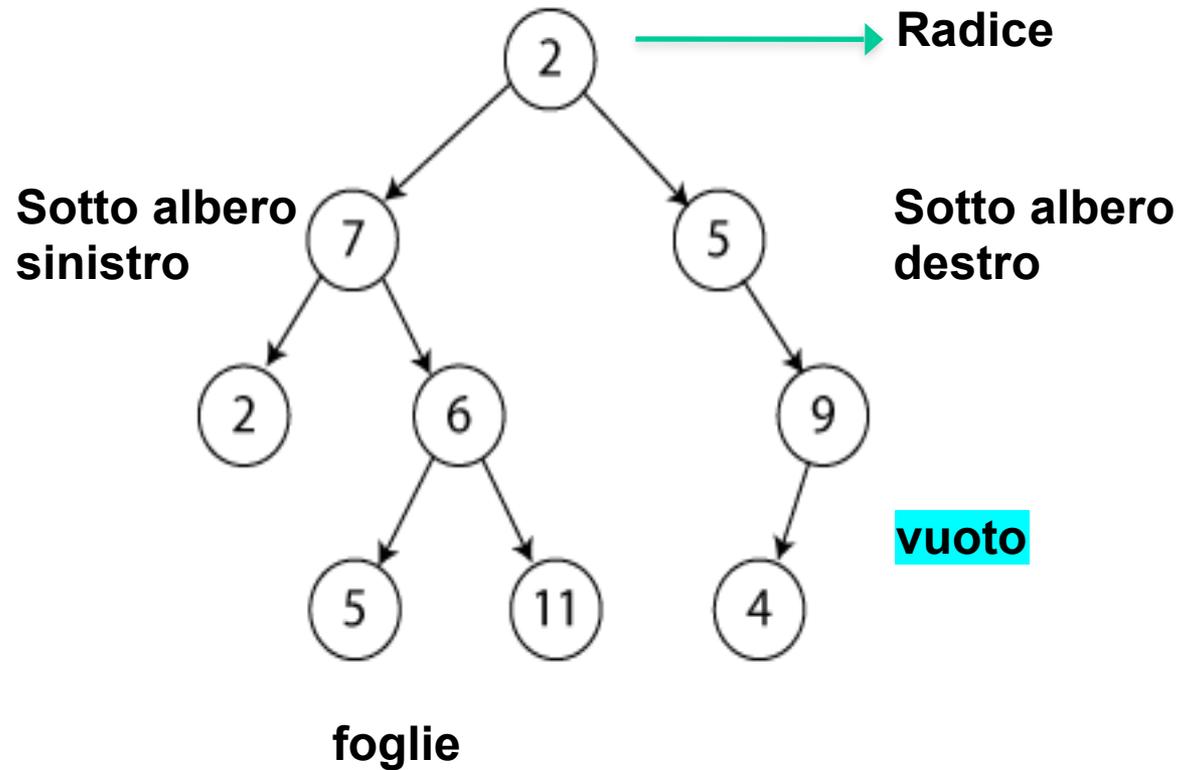
```
# type tree =  
  Leaf of int  
  | Node of tree * int * tree;;
```

```
type tree = Leaf of int | Node of tree * int * tree
```

Quanti di voi hanno programmato con strutture dati del tipo tree?

Alberi binari

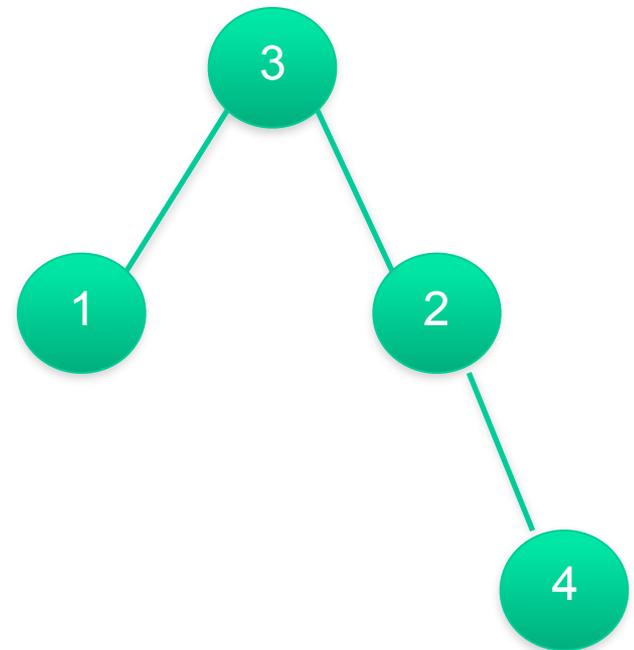
Li avete visti a algoritmica!!!!



Alberi binari in OCaml

```
# type tree =  
    Empty  
  | Node of tree * int * tree;;
```

```
let t : tree =  
Node(Node(Empty,1, Empty), 3,  
Node(Empty,2, Node(Empty, 4,Empty)))
```



Ricerca in un albero

```
let rec contains (t : tree) (n : int): bool =  
  match t with  
  | Empty -> false  
  | Node (lt, x, rt) ->  
  
    x = n || (contains lt n) || (contains rt n)
```

La funzione *contains* effettua una ricerca del valore *n* sull'albero *t*
Caso peggiore: deve visitare tutto l'albero

Alberi binari di ricerca

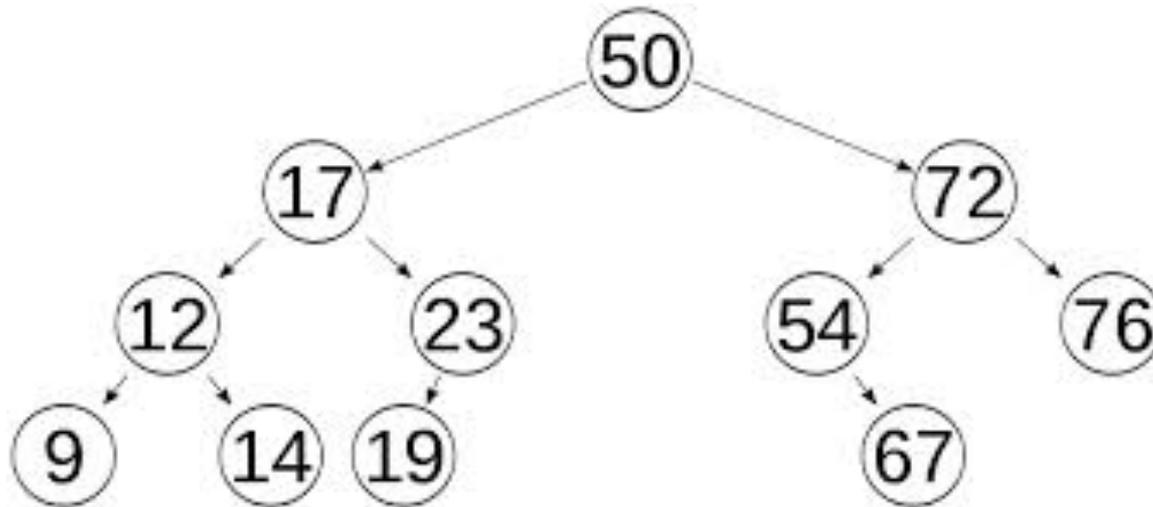
Idea: ordinare i dati sui quali viene fatta la ricerca

Un albero binario di ricerca (BST) è un albero binario che deve soddisfare alcune proprietà invarianti aggiuntive

INVARIANTE DI RAPPRESENTAZIONE (ricorsivo)

- **Node** (lt, x, rt) è un **BST** se
 - lt e rt sono **BST**
 - tutti i nodi di lt contengono valori $< x$
 - tutti i nodi di rt contengono valori $> x$
- **Empty** (l'albero vuoto) è un **BST**

Esempio



**L'invariante di rappresentazione dei BST è soddisfatto
Come si dimostra?**

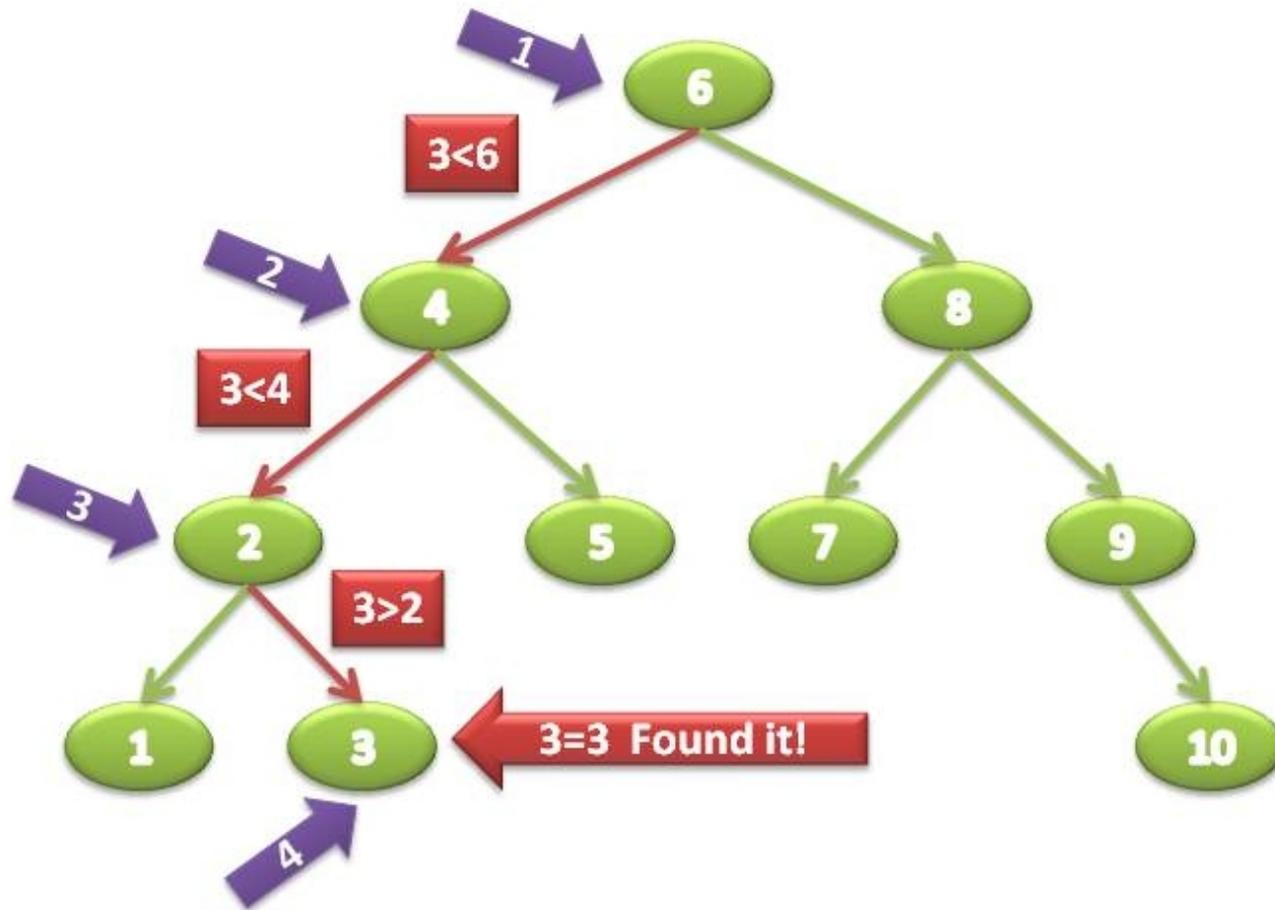
Ricerca su un BST

```
(* Ipotesi: t è un BST *)
let rec lookup (t : tree) (n : int) : bool =
  match t with
  | Empty -> false
  | Node (lt, x, rt) ->
    if x = n then true
    else if n < x then (lookup lt n)
    else (lookup rt n)
```

Osservazione 1: L'invariante di rappresentazione guida la ricerca

Osservazione 2: La ricerca può restituire valori non corretti se applicata a un albero che non soddisfa l'invariante di rappresentazione

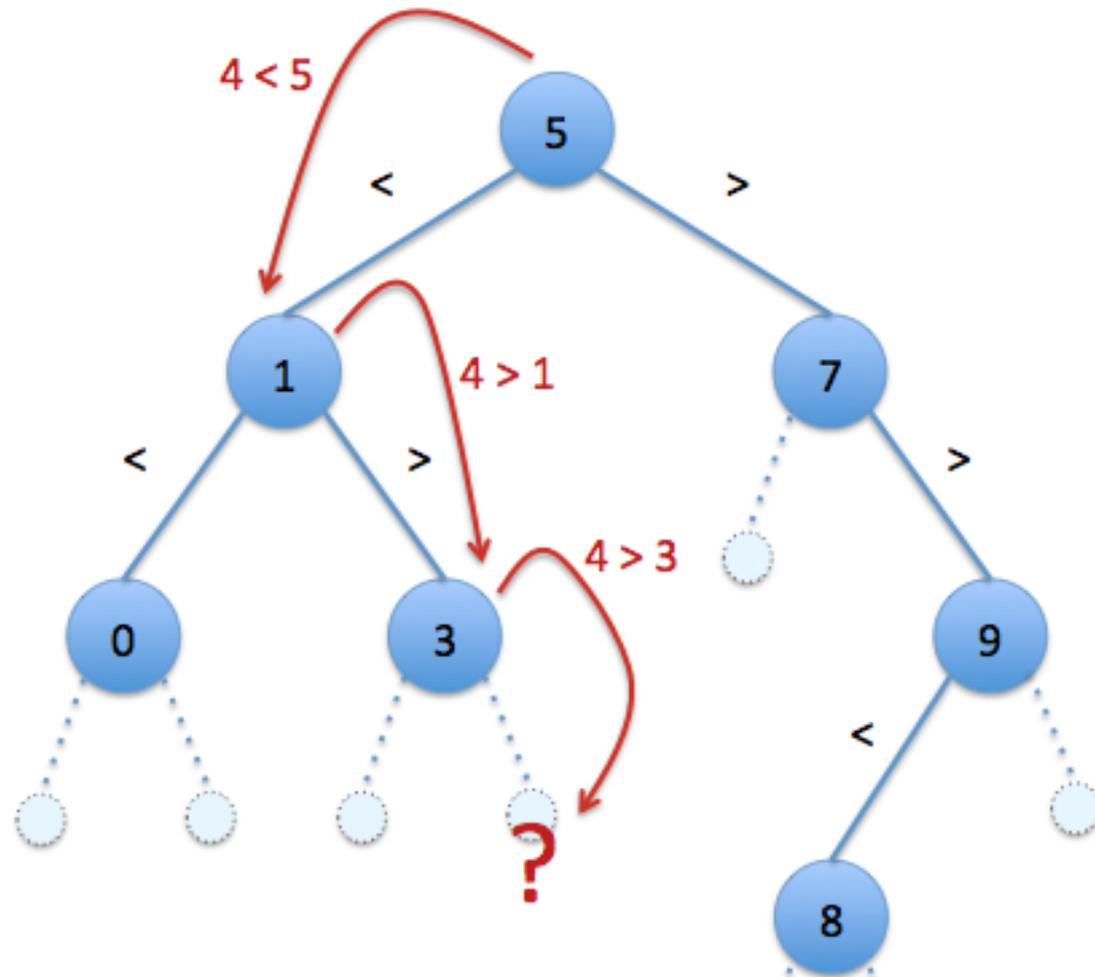
lookup(t, 3)



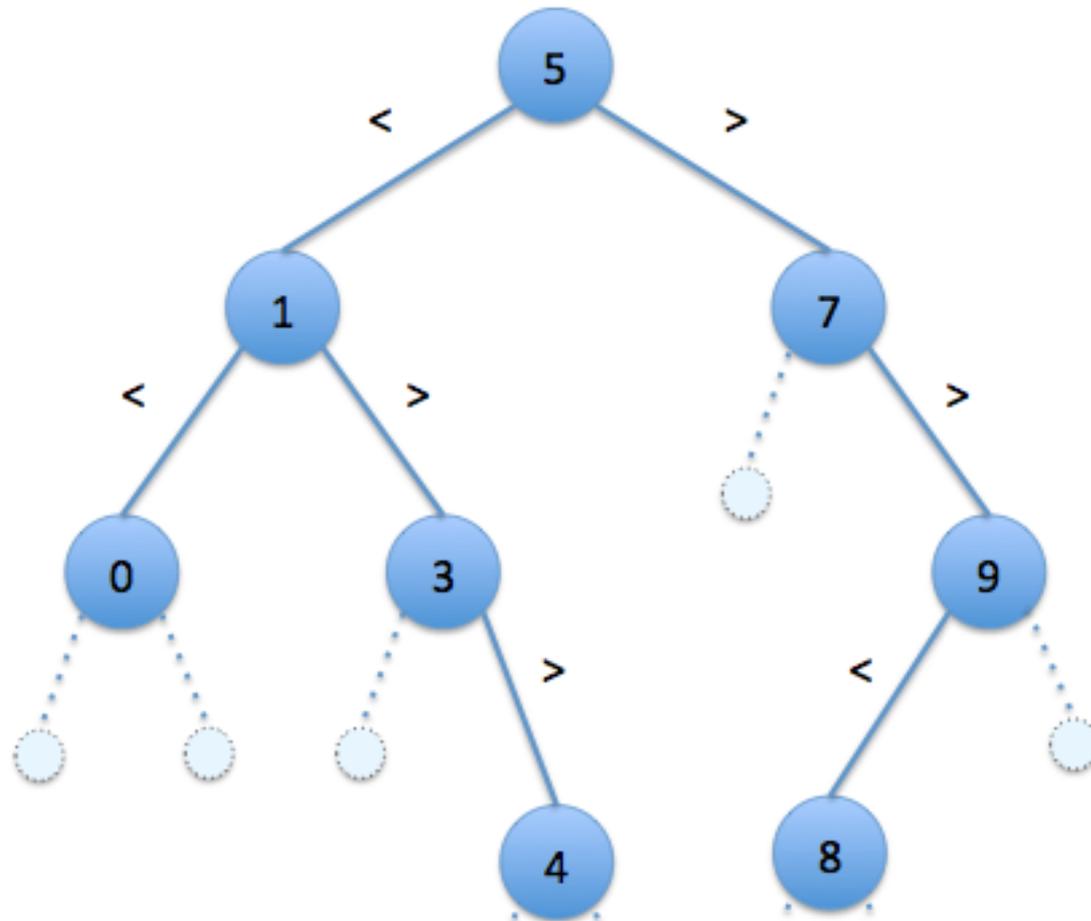
Come costruiamo un BST???

- **Opzione 1**
 - costruiamo un albero e poi controlliamo (check) se vale l’invariante di rappresentazione
- **Opzione 2**
 - definire le funzioni che costruiscono BST a partire da BST (ad esempio, la funzione che inserisce un elemento in un BST e restituisce un BST)
 - definire una funzione che costruisce il BST vuoto
 - tutte queste funzioni soddisfano l’invariante di rappresentazione, pertanto “per costruzione” otteniamo un BST
 - non si deve effettuare nessun controllo a posteriori!!

insert(t, 4)



insert(t, 4)

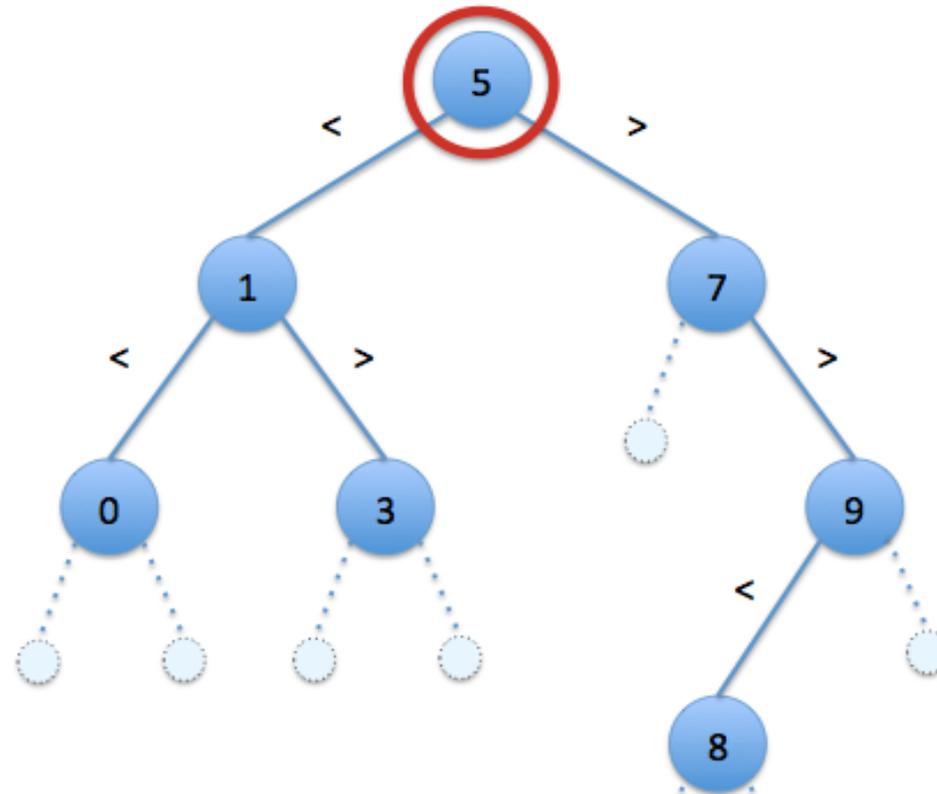


Inserimento

```
let rec insert (t : tree) (n : int):tree =
  match t with
  | Empty -> Node (Empty , n, Empty )
  | Node (lt, x, rt) ->
    if x = n then t else
      if n < x then Node(insert lt n, x, rt)
      else Node (lt, x, insert rt n)
```

Per quale motivo l'albero costruito dalla funzione insert è un BST???

delete(t,5)



L'operazione di rimozione è più complicata: si deve promuovere la foglia 3 a radice dell'albero!!!

delete

```
let rec delete (t : tree) (n : int):tree =
  match t with
  | Empty -> Empty
  | Node (lt, x, rt) ->
    if x = n then
      begin match (lt,rt) with
        (Empty, Empty) -> Empty
        | (Node_, Empty) -> lt
        | (Empty, Node_) -> rt
        | _ ->
          let m = tree_max lt in
            Node(delete lt m, m, rt)
          end
      end
    else if n < x then Node(delete lt n, x, rt)
      else Node(lt, x, delete rt n)
```

Funzione ausiliaria

```
let rec tree_max (t : tree) : int =  
  match t with  
  | Node(_, x, Empty) -> x  
  | Node(_, _, rt) -> tree_max rt  
  | _ -> failwith "tree_max called on Empty"
```

L'invariante di rappresentazione garantisce che il valore max si trova nella parte più a destra dell'albero