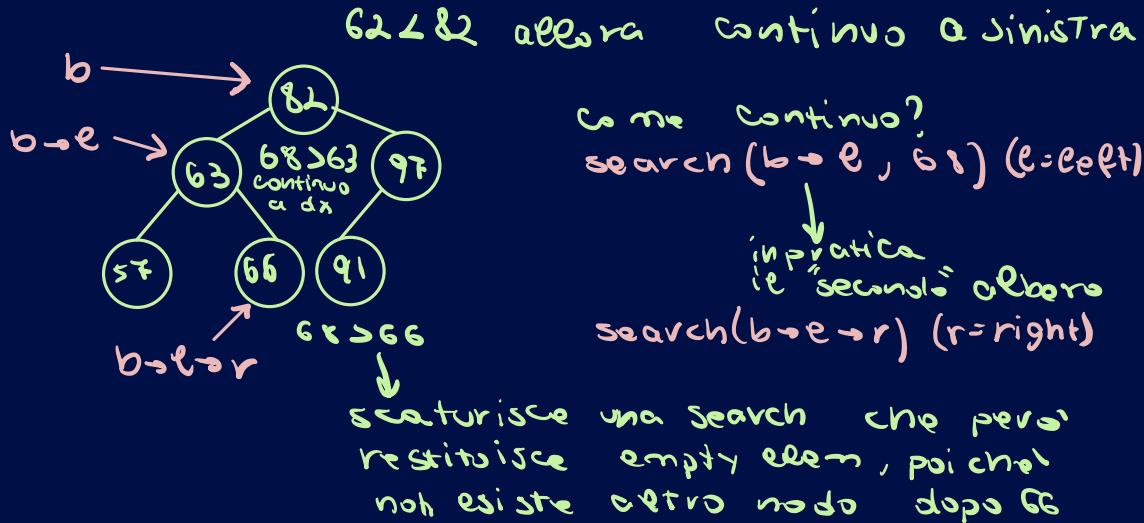


## ★ SEARCH

search(b, k)

search(b, 68)

Devo partire per forza dalla radice



## search

$\Theta(h)$  nel caso peggiore, con h altezza dell'albero

Nel testo AHU

```

Search
function MEMBER ( x: elementype; A : SET ) : boolean;
{ returns true if x is in A , false otherwise }
begin
  if A = nil then
    return (false) { x is never in Ø }
  else if x = A.t.element then
    return (true)
  else if x < A.t.element then
    return (MEMBER(x, A.t.leftchild))
  else { x > A.t.element }
    return (MEMBER(x, A.t.rightchild))
end; { MEMBER }
```

su set NON dictionary

chiave

quindi cerco solo  
una chiave NON  
coppia chiave  
valore

no! no. Poco, deve  
restituire valore o  
empty elem

{ chiamate  
ricorsive

Fig. 5.2. Testing membership in a binary search tree.

## COMPLESSITÀ

★ O<sub>1</sub> se cerco elem che e' in prima posiz (caso migliore)  
(Sia nel caso mai distribuito, sia quello completo)

★ O<sub>n</sub> se cerco elem che non e'  
(mi fermo solo a voeta su ogni piano) (caso peggiore)  
(Sia nel caso mai distribuito, sia quello completo)

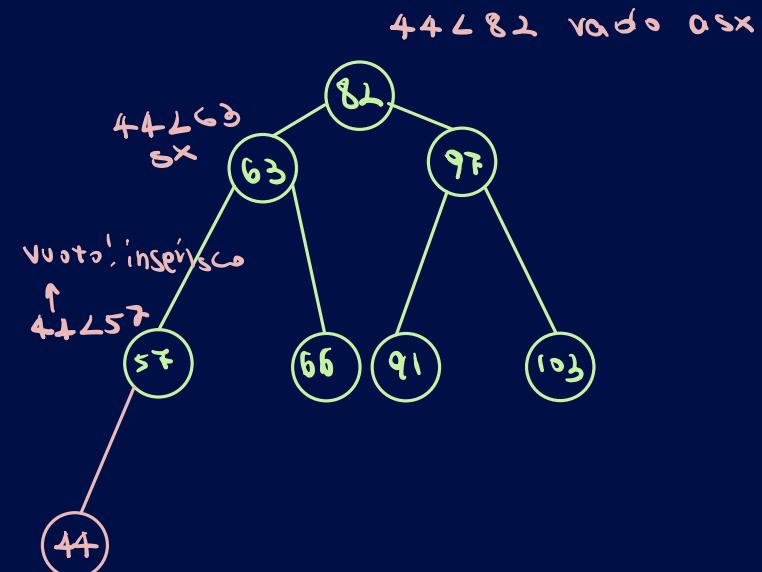
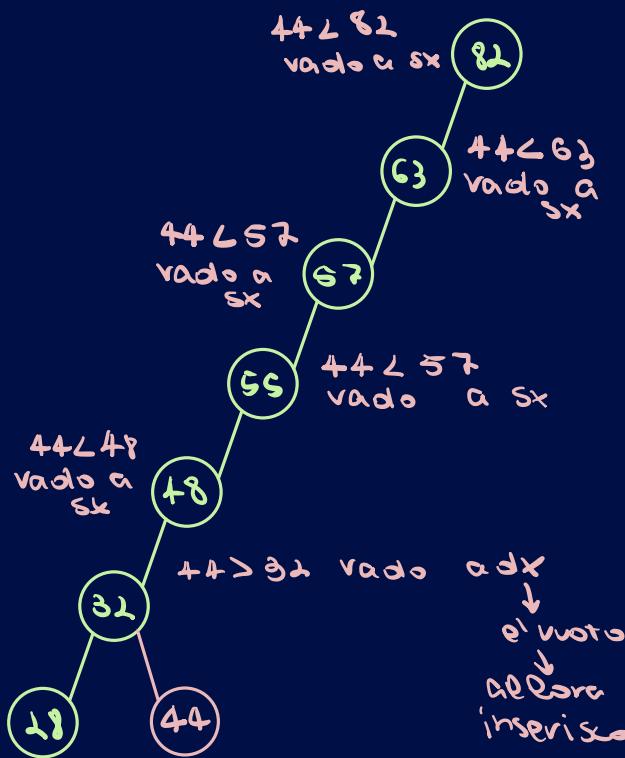
N.B. Nel caso completo n puo' essere piu' vicino a n o log n per le sue regole

Nel caso sbilanciato posso dire anche O<sub>n</sub> perch' per come e' fatto guardo tutti i nodi

Oh sarebbe  $0h+1$   
 $0$        $h=1$   
 $1$       ma i livelli  
 sono 2  
 pero' come sappiamo le  
 costanti non sono rilevanti  
 quindi  
 $0h+1 \Rightarrow 0h$

## ★ INSETTATORE

insert (root, 44, value)



In pratica e' uguale alla **search** ma invece che restituire emptyelem, dove c'e' il "vuoto" inserisco l'elemento

**insertElem**  
 $\Theta(h)$  nel caso peggiore, con  $h$  altezza dell'albero

Nel testo AHU

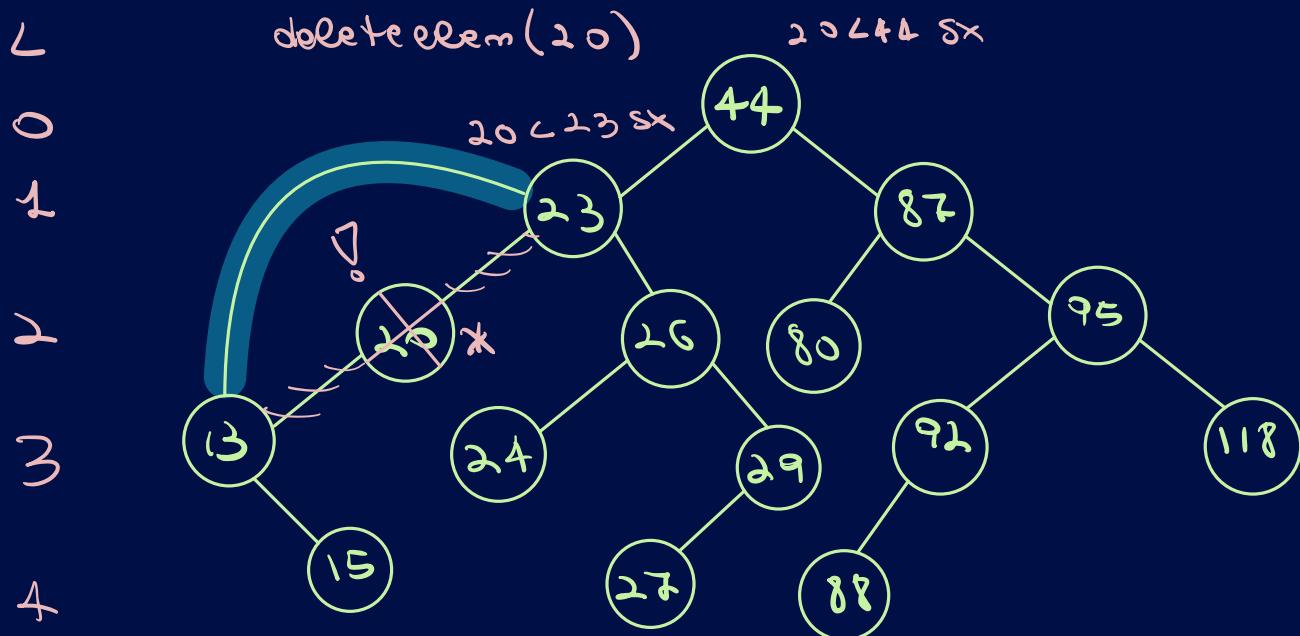
```

procedure INSERT ( x: elementtype; var A: SET );
{ add x to set A }
begin
  if A = nil then begin
    New(A);
    A^.element := x;
    A^.leftchild := nil;
    A^.rightchild := nil
  end
  else if x < A^.element then
    INSERT(x, A^.leftchild)
  else if x > A^.element then
    INSERT(x, A^.rightchild)
  { if x = A^.element, we do nothing; x is already in the set }
end; { INSERT }
  
```

→ Se sono arrivati in un punto vuoto

Fig. 5.3. Inserting an element into a binary search tree.

## ★ DELETE ELEMENT



\* 20 ha un solo figlio: quindi il nodo che puntava a 20 deve ora puntare all'unico figlio di 20 (13)

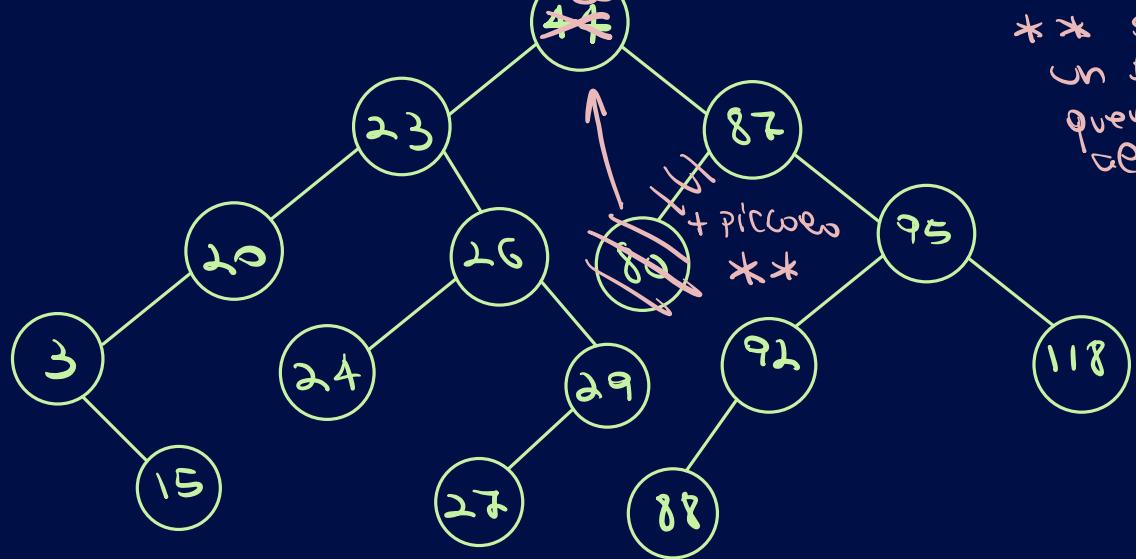
se il nodo ha 0 figli, lo elimino e basta

se il nodo ha + figli

ex voglio eliminare 44 → devo mantenere le condizioni per cui  $\langle \text{rad} \rangle > \text{rad}$

Quindi o si sceglie il + grande a sx o il + piccolo a dx da sostituire al posto di 44

sceglio il + piccolo di dx → facile da trovare:  
 sarà il + a sinistra  
 nel "RAMO" di dx  
 e che non abbia + figli  
 ↓  
 in questo caso ho



**deleteMin**  
è ausiliaria di deleteElem,  $\Theta(h)$  nel caso peggiore)

o AHU

```

function DELETEMIN ( var A: SET ) : elementtype;
  { returns and removes the smallest element from set A }
begin
  if A.leftchild = nil then begin    se il nodo non ha figli
    { A points to the smallest element } lo elimina
    DELETEMIN := A.element;
    A := A.rightchild;
    { replace the node pointed to by A by its right child }
  end
  else { the node pointed to by A has a left child }
    DELETEMIN := DELETEMIN(A.leftchild) chiamata ricorsiva
end; { DELETEMIN } return

```

Fig. 5.4. Deleting the smallest element.

**FUNZIONE AUSILIARIA**  
che elimina il nodo  
+ piccole