

Algoritmi e Strutture Dati

Alberi Binari di Ricerca (BST)

Maria Rita Di Berardini, Emanuela Merelli¹

¹Dipartimento di Matematica e Informatica
Università di Camerino

Alberi Binari di Ricerca (Binary Search Trees – BST)

Un albero binario di ricerca è un particolare tipo di albero binario

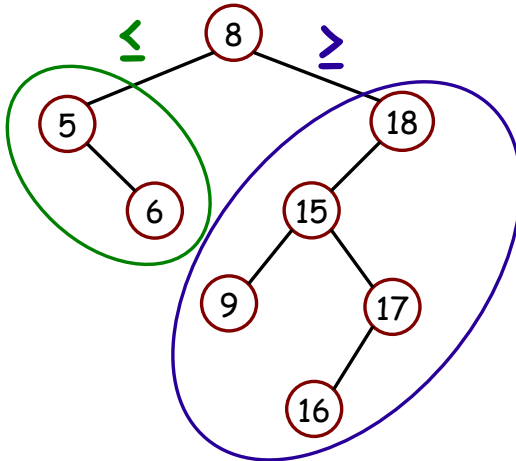
Ogni nodo u è un oggetto costituito da diversi campi: *key* (più eventuali dati satellite) un campo *left*, *right* e *parent* che puntano rispettivamente al figlio sinistro, al figlio destro e al padre u

Le chiavi sono sempre memorizzate in modo che sia verificata la **proprietà dell'albero binario di ricerca**:

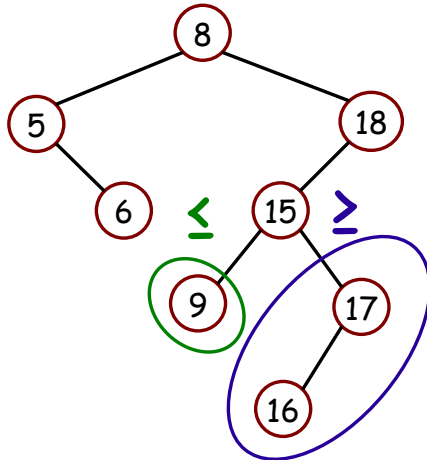
Sia x un nodo di un albero binario di ricerca; allora:

- *se y è un nodo del **sottoalbero sinistro** di x allora $key[y] \leq key[x]$*
- *se y è un nodo del **sottoalbero destro** di x allora $key[y] \geq key[x]$*

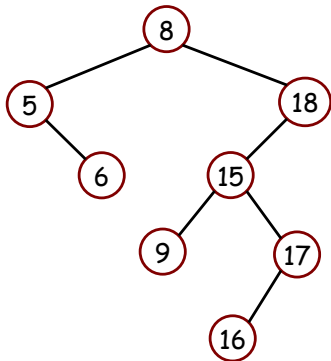
Alberi Binari di Ricerca



Alberi Binari di Ricerca



Alberi Binari di Ricerca e visita in ordine simmetrico



Inorder-Tree-Walk(x)

if $x \neq \text{NIL}$

then

Inorder-Tree-Walk($\text{left}[x]$)

stampa $\text{key}[x]$

Inorder-Tree-Walk($\text{right}[x]$)

Inorder-Tree-Walk(root)

5, 6, 8, 9, 15, 16, 17, 18

elena tutte le chiavi di un BST
in modo ordinato

Esercizio

Dimostrare che, se x è la radice di un sottoalbero di n nodi la chiamata **Inorder-Tree-Walk**(x) richiede un tempo $\Theta(n)$

Sia $T(n)$ il tempo richiesto dalla chiamata **Inorder-Tree-Walk**(x) quando x è la radice di un (sotto)albero di n nodi

Se $n = 0$ (cioè se $x = \text{NIL}$) $T(n) = c$ per qualche costante $c > 0$

Se $n > 0$ e k è il numero di nodi del sottoalbero sinistro, allora $T(n) = T(k) + T(n - k - 1) + d$ per qualche costante $d > 0$

Applichiamo il metodo della sostituzione per dimostrare che $T(n) = (c + d)n + c$ per ogni $n \geq 0$ e quindi che $T(n) = \Theta(n)$

Esercizio

Applichiamo il metodo della sostituzione per dimostrare che $T(n) = (c + d)n + c$ per ogni $n \geq 0$ e quindi che $T(n) = \Theta(n)$

Procediamo per induzione su n

Caso Base ($n = 0$): $T(n) = c = (c + d)0 + c$

Per $n > 0$ abbiamo che

$$\begin{aligned}
 T(n) &= T(k) + T(n - k - 1) + d \\
 &= ((c + d)k + c) + ((c + d)(n - k - 1) + c) + d \\
 &= (c + d)k + c + (c + d)(n - k) - (c + d) + c + d \\
 &= (c + d)n + c
 \end{aligned}$$

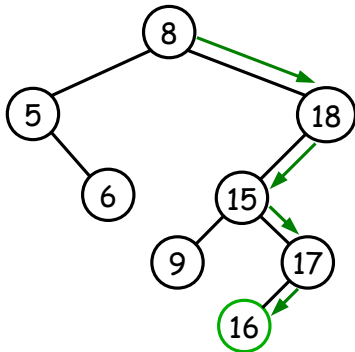
Operazioni su Alberi Binari di Ricerca

I BST sono strutture dati sulle quali vengono realizzate molte delle operazioni definite su insiemi dinamici, tra le quali:

**Search, Insert, Delete, Minumun, Maximum,
Predecessor e Successor**

Ricerca di un elemento

Su ogni nodo, usiamo la proprietà dell'albero binario di ricerca per decidere se proseguire a destra o a sinistra



Cerchiamo 16

Il numero di confronti necessari per individuare un elemento in un albero binario di ricerca è pari, al più, all'altezza dell'albero

Ricerca di un elemento

Tree-Search(x, k)

if $x = \text{NIL}$ or $k = \text{key}[x]$
then return x

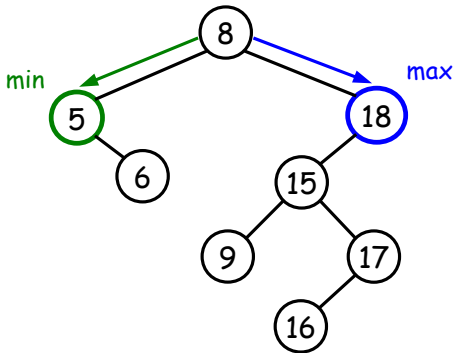
if $k < \text{key}[x]$
then return **Tree-Search**($\text{left}[x], k$)
else return **Tree-Search**($\text{right}[x], k$)

Iterative-Tree-Search(x, k)

while $x \neq \text{NIL}$ or $k \neq \text{key}[x]$
do if $k < \text{key}[x]$
then $x \leftarrow \text{left}[x]$
else $x \leftarrow \text{right}[x]$
return x

Minimo e massimo

Seguiamo i puntatori *left* (per **Tree-Minimum**) e *right* (per **Tree-Maximum**) dalla radice fin quando non si incontra NIL



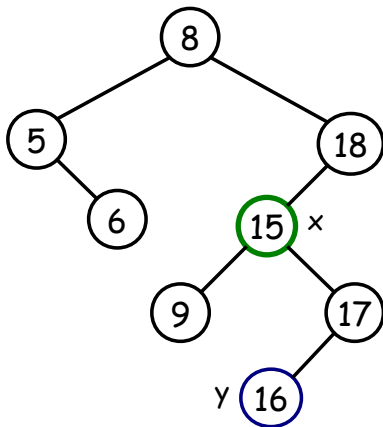
Minimo e massimo

```
Tree-Minimum( $x$ )  
  while  $left[x] \neq \text{NIL}$   
    do  $x \leftarrow left[x]$   
return  $x$ 
```

```
Tree-Maximum( $x$ )  
  while  $right[x] \neq \text{NIL}$   
    do  $x \leftarrow right[x]$   
return  $x$ 
```

Successore e Predecessore

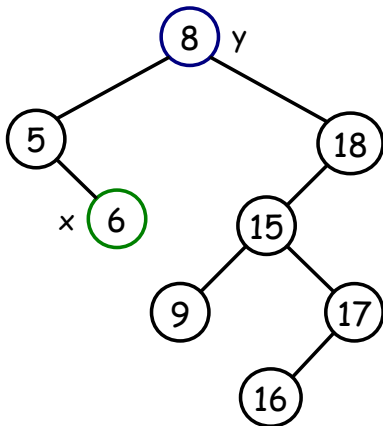
Successore - caso 1: x ha un sottoalbero destro



$y = \text{Successor}(x) =$
 $\text{Tree-Minimum}(\text{right}[x]) = y$

Successore e Predecessore

Successore - caso 2: x non ha un sottoalbero destro, ma ha un successore y



$y = \text{Successor}(x)$ è l'antenato più prossimo di x il cui figlio sinistro è anch'esso un antenato di x

Successore e Predecessore

- Quale è il successore di x se questo non ha un figlio destro?
- Non possiamo scendere a sinistra perchè le chiavi contenute nel sottoalbero sinistro sono tutte più piccole di $key[x]$
- Possiamo solo provare a risalire l'albero di padre in padre
- Sia $y = p[x]$ (il padre di x) e assumiamo che x sia il figlio destro di y (cioè che $x = right[y]$). In questo caso $key[y]$ è minore o uguale a $key[x]$ e quindi y non è il successore che stiamo cercando
- Dobbiamo risalire ancora di un livello esaminando il padre di y
- Continuiamo a salire di livello finchè non individuiamo un nodo il cui figlio sinistro è un antenato di x , questo nodo contiene la prima chiave più grande di $key[x]$

Successore e Predecessore

```
Tree-Successor( $x$ )  
  if  $right[x] \neq \text{NIL}$   
    then return Tree-Minimum( $right[x]$ )  
  
   $y \leftarrow p[x]$   
  while  $x \neq \text{NIL}$  and  $x = right[y]$   
    do  
       $x \leftarrow y$   
       $y \leftarrow p[y]$   
  
  return  $y$ 
```


Successore e Predecessore

L'algoritmo per il calcolo del predecessore di un dato nodo è del tutto simmetrico

Seguiamo il puntatore p dal nodo x fino, al più, alla radice

Di nuovo, il costo di entrambe le operazioni è proporzionale all'altezza dell'albero

Ricapitolando

Abbiamo dimostrato il seguente teorema

Teorema: le operazioni su insiemi dinamici **Search**, **Minumun**, **Maximum**, **Predecessor** e **Successor** possono essere eseguite su un albero binario di ricerca di altezza h in in tempo $O(h)$

Inserimento

L'algoritmo **Tree-Insert**(T, z) lavora in maniera molto simile alla ricerca

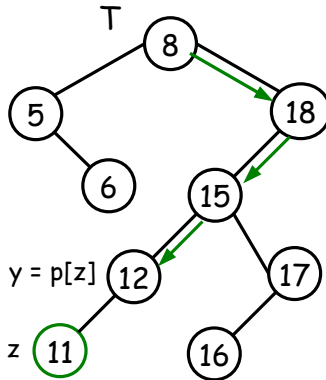
Cerca la corretta posizione di z nell'albero identificando così il nodo y che diventerà padre di z

Infine, appende z come figlio sinistro/destro di y in modo che sia mantenuta la proprietà dell'albero binario di ricerca

Come la altre primitive su alberi binari di ricerca, la procedura **Tree-Insert** richiede un tempo $O(h)$

Inserimento

Tree-Insert(T, z) con $\text{key}[z] = 11$



Tree-Insert(T, z)

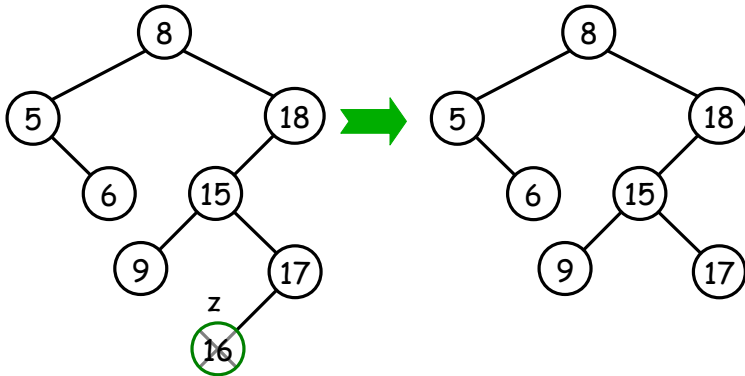
```

 $y \leftarrow \text{NIL}$            ▶ cerca un cammino discendente dalla radice fino ad una
 $x \leftarrow \text{root}[T]$     ▶ foglia;  $x$  segue il cammino,  $y$  punta al padre di  $x$ 
while  $x \neq \text{NIL}$ 
    do  $y \leftarrow x$ 
        if  $\text{key}[z] < \text{key}[x]$ 
            then  $x \leftarrow \text{left}[x]$ 
            else  $x \leftarrow \text{right}[x]$ 
▶ usciti da questo ciclo  $y$  è il puntatore al padre del nuovo nodo
 $p[z] \leftarrow y$ 
if  $y = \text{NIL}$ 
    then  $\text{root}[T] \leftarrow z$ 
    else if  $\text{key}[z] < \text{key}[y]$ 
        then  $\text{left}[y] \leftarrow z$ 
        else  $\text{right}[y] \leftarrow z$ 

```

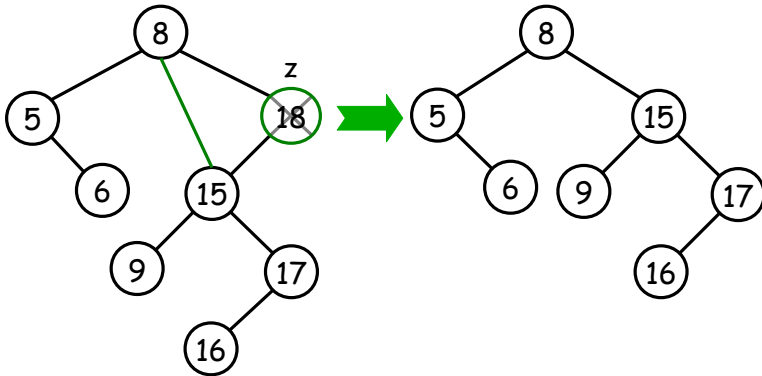
Cancellazione

Tree-Delete(T, z) quando il nodo z è una foglia



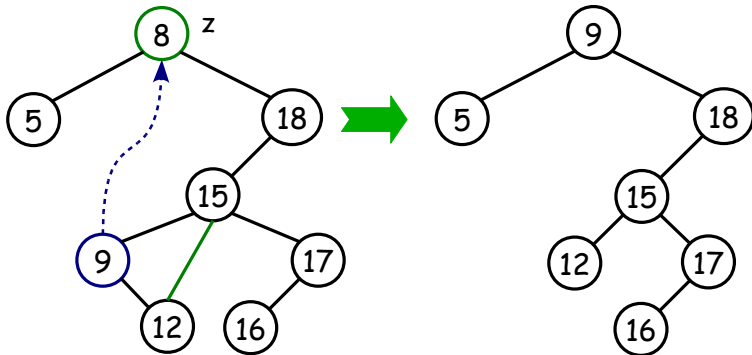
Cancellazione

Tree-Delete(T, z) quando z ha esattamente un figlio



Cancellazione

Tree-Delete(T, z) quando z ha due figli



La chiave di z viene sostituita con quella di $y = \text{Tree-Successor}(T, z)$ ed eliminiamo il nodo y (è il nodo più a sinistra del sottoalbero destro di z ed ha al più un figlio, quello destro)

Tree-Delete(T, z)

```

if  $left[z] = \text{NIL}$  or  $right[z] = \text{NIL}$     ▷  $z$  ha al più un figlio
    then  $y \leftarrow z$ 
    else  $y \leftarrow \text{Tree-Successor}(z)$     ▷  $y$  è il nodo da eliminare
if  $left[y] \neq \text{NIL}$ 
    then  $x \leftarrow left[y]$ 
    else  $x \leftarrow right[y]$ 
    ▷  $x$  è il figlio non NIL di  $y$  (se esiste) altrimenti è NIL
if  $x \neq \text{NIL}$     then  $p[x] \leftarrow p[y]$ 
if  $p[y] = \text{NIL}$ 
    then  $root[T] \leftarrow x$ 
    else if  $left[p[y]] = y$ 
        then  $left[p[y]] \leftarrow x$ 
        else  $right[p[y]] \leftarrow x$ 
if  $y \neq z$     ▷ se il nodo eliminato è il successore di  $z$ 
    then  $key[z] \leftarrow key[y]$     ▷ più altri dati satellite
return  $y$ 
  
```

Ricapitolando

Abbiamo dimostrato il seguente teorema

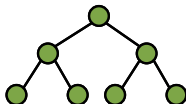
Teorema: le operazioni su insiemi dinamici **Insert** e **Delete** possono essere eseguite su un albero binario di ricerca di altezza h in in tempo $O(h)$

Costo delle Operazioni

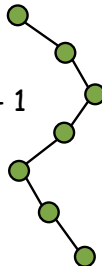
Tutte le operazioni su BST hanno un costo $O(h)$ dove h è l'altezza dell'albero binario di ricerca

È fondamentale mantenere l'albero bilanciato, in questo caso infatti l'altezza è logaritmica nel numero di nodi

$$h = \log_2 (n + 1) - 1$$



$$h = n - 1$$



Costo delle Operazioni

Quando l'albero è sbilanciato $h = n - 1$ e le operazioni hanno un costo lineare invece che logaritmico

Soluzione:

- introduciamo alcune proprietà aggiuntive sui BST per mantenerli bilanciati (Alberi AVL, Alberi Rosso-Neri)
- le operazioni dinamiche sull'albero devono preservare le proprietà introdotte per mantenere il bilanciamento
- paghiamo in termini di una maggiore complessità di tali operazioni