Algoritmi e Strutture Dati

Alberi Binari di Ricerca (BST)

Maria Rita Di Berardini, Emanuela Merelli[1]

[1]Dipartimento di Matematica e Informatica
Università di Camerino

A.A. 2006/07

Alberi Binari di Ricerca (Binary Search Trees – BST)

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

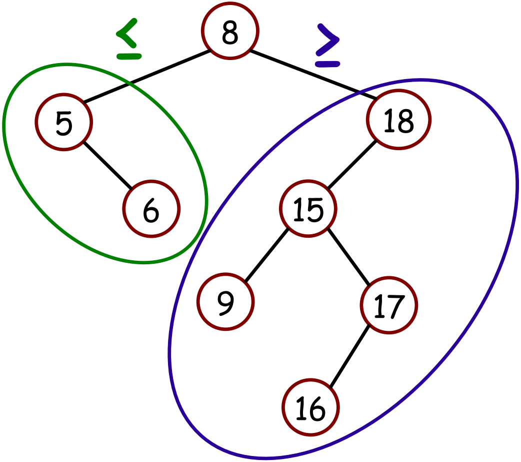
Ogni nodo u è un oggetto costituito da diversi campi: *key* (più eventuali dati satellite) un campo *left*, *right* e *parent* che puntano rispettivamente al figlio sinistro, al figlio destro e al padre u

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

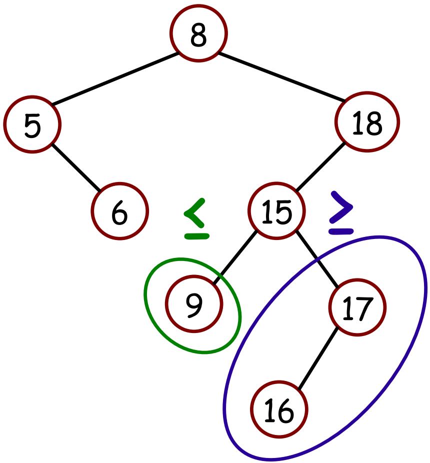
Sia x un nodo di un albero binario di ricerca; allora:

- *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 $key[y] \geq key[x]$*

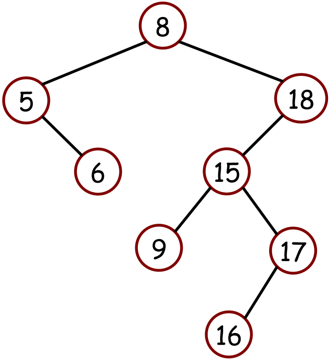
Alberi Binari di Ricerca



Alberi Binari di Ricerca



Alberi Binari di Ricerca e visita in ordine simmetrico



Inorder-Tree-Walk(x)
  if $x \neq \mathrm{NIL}$
   then
      Inorder-Tree-Walk($left[x]$)
      stampa $key[x]$
      Inorder-Tree-Walk($right[x]$)

Inorder-Tree-Walk($root$)

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

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

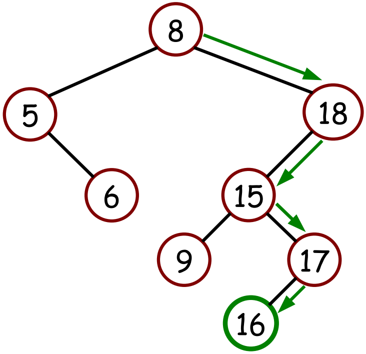
Operazioni su Alberi Binari di Ricerca

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 usa la proprietà dell'albero binario di ricerca per decidere se proseguire a destra o a sinistra



Ricerchiamo 16

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

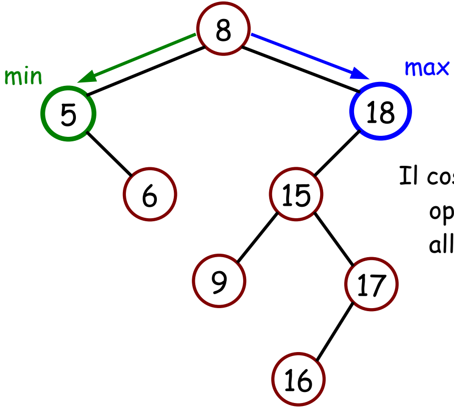
Ricerca di un elemento

Tree-Search(x, k)
  if $x = \mathrm{NIL}$ or $k = key[x]$
    then return x
  if $k < key[x]$
    then return Tree-Search$(left[x], k)$
    else return Tree-Search$(right[x], k)$

Iterative-Tree-Search(x, k)
  while $x \neq \mathrm{NIL}$ or $k \neq key[x]$
    do if $k < key[x]$
      then $x \leftarrow left[x]$
      else $x \leftarrow right[x]$
  return x

Minimo e massimo

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



Il costo di entrambe le operazioni è proporzionale all'altezza dell'albero

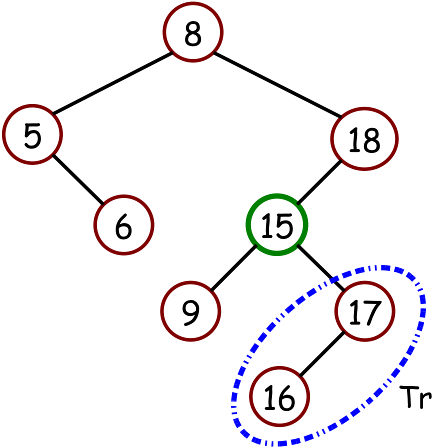
Minimo e massimo

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

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

Successore e Predecessore

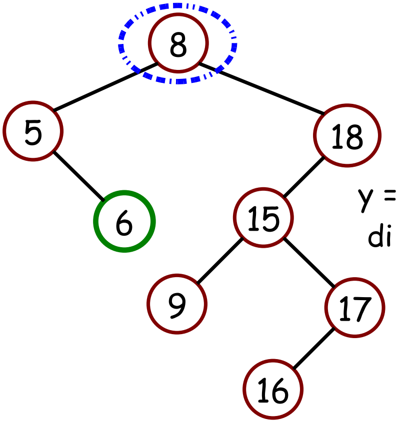
Successore (caso 1): x ha un sottoalbero destro



Successor(x) =
Tree-Minimum(right[x]) = 16

Successore e Predecessore

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



y = l'antenato più prossimo di x il cui figlio sinistro è anche un antenato si x

Successore e Predecessore

Tree-Successor(x)

if $right[x] \neq \mathrm{NIL}$

then return **Tree-Minumum**$(right[x])$

$y \leftarrow p[x]$

while $x \neq \mathrm{NIL}$ 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

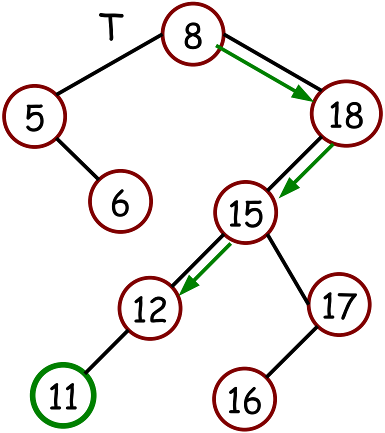
Cerca la corretta posizione di z nell'albero identificando così il nodo y che diventerà padre di z

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 key[z] = 11

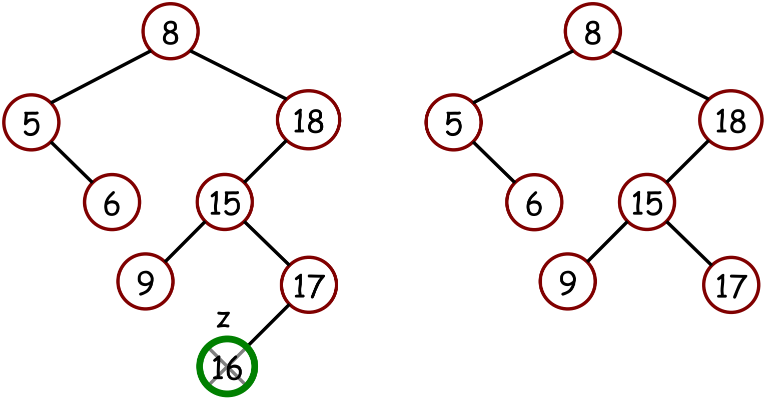


Tree-Insert(T, z)

```
y ← NIL              ▷ L'algoritmo cerca un cammino discendente dalla radice
x ← root[T]          ▷ fino ad una foglia; x segue il cammino, y punta al padre di x
while x ≠ NIL
      do  y ← x
          if  key[z] < key[x]
              then  x ← left[x]
              else  x ← right[x]
▷ usciti da questo ciclo y è il puntatore al padre del nuovo nodo
p[z] ← y
if y = NIL
   then  root[T] ← z
   else if  key[z] < key[y]
            then  left[y] ← z
            else  righ[y] ← z
```

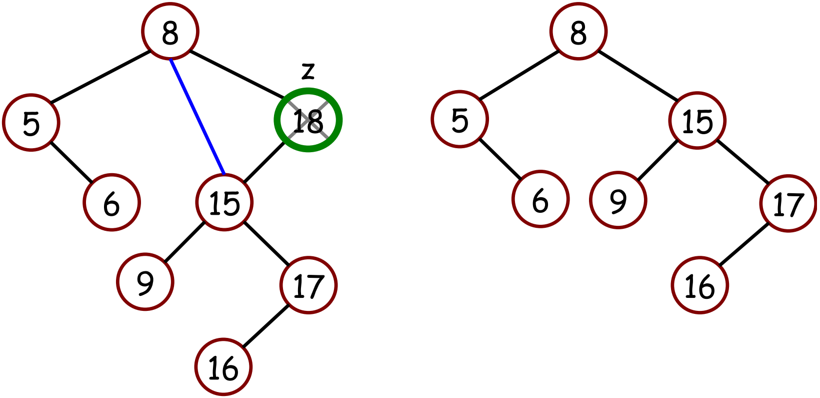
Cancellazione

Tree-Delete(T,z): z è una foglia



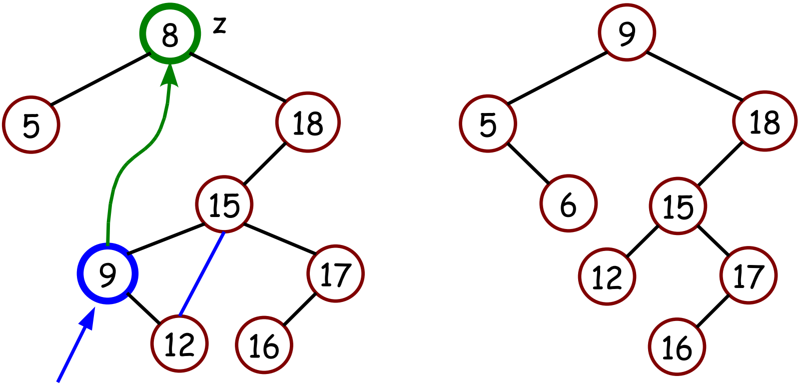
Cancellazione

Tree-Delete(T,z): z ha solo un figlio



Cancellazione

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



Tree-Successor(T, z) è il nodo più a sinistra del sottoalbero destro di z, ha al più un figlio (destro)

Tree-Delete(T, z)

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

Ricapitolando

Abbiamo dimostrato il seguente teorema

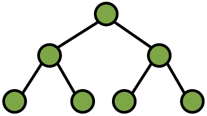
Teorema: le operazioni su insiemi dinamici Insert e Delete possono essere eseguite su un albero binario di ricerca di altezza h in in tempo $O(h)$

Costo delle Operazioni

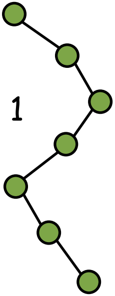
Tutte le operazioni su BST hanno un costo $O(h)$ dove h è l'altezza dell'albero binario di ricerca

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