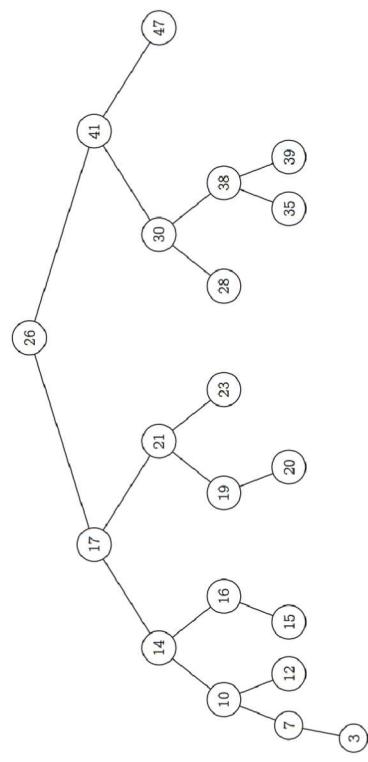


- **p**: puntatore al padre di x (`NIL` se è la radice)

Binary Search Tree

Operazioni e Scopo



Definizione

Un BST è un albero binario non necessariamente completo dove

- I nodi contengono **chiavi numeriche**, tutti i nodi contenuti nel **sottoalbero sinistro** **contengono una chiave minore** alla chiave di x
- Dato un nodo x qualsiasi, tutti i nodi contenuti nel **sottoalbero destro** **contengono una chiave maggiore** alla chiave di x

Definizione: Nodo

Ogni nodo x ha quindi la seguente struttura

- **key**: chiave di x
- **data**: dati satellite legati alla chiave di x
- **left**: puntatore al figlio sinistro di x (`NIL` se inesistente)
- **right**: puntatore al figlio destro di x (`NIL` se inesistente)

Operazioni

- Visita ordinata
- Massimo e minimo
- Ricerca
- Inserimento
- Successore e predecessore di un nodo
- Cancellazione di un nodo
- Costruzione da un vettore

Il BST è una struttura molto in contesti simili a quelli di una hash table, ovvero di frequente ricerca, inserimento e eliminazione di dati.

Nonostante abbiano un costo delle operazioni maggiore rispetto alle hash table nel caso medio (che lavorano in tempo costante), i BST permettono di operare in tempo $\Theta(h)$ (dove h è l'altezza dell'albero) **anche nel caso pessimo**.

Mantenendo quindi un'altezza bilanciata da entrambi i lati, per esempio utilizzando dei red-black tree, si può mantenere un caso pessimo di $\Theta(\log(n))$

Visita Ordinata

Idea
 Stampa (o esegui qualsiasi altra operazione) le chiavi dei nodi del BST ordinate

Massimo e Minimo

```
1 if x != NIL  
2   INORDER-TREE-WALK(x.left) / / valori < x.key  
3   print(x.key)  
4   INORDER-TREE-WALK(x.right) // valori > x.key
```

Parametri: $x = \text{nodo radice}$

Considerando che l'albero è sbilanciato, possiamo scrivere l'equazione ricorsiva come

$$T(n) = T(n - k_i - 1) + T(k_i) + 1$$

\$k_i\$ rappresenta lo sbilanciamento al piano i , con un valore $k_i \in [0, n-1]$

Possiamo procedere per induzione ponendo che il costo sia n

- $T(1) = 1 \implies$ caso base verificato
- $T(n) = n - \cancel{k_i - 1} + \cancel{k_i + 1} = n$

① Costo Computazionale

$$\Theta(n)$$

TREE-MAXIMUM(x)

```
1 while x.right != NIL  
2   x = x.right  
3 return x
```

Parametri: $x = \text{nodo radice}$

Return: nodo con chiave maggiore

TREE-MINIMUM(x)

```
1 while x.left != NIL  
2   x = x.left  
3 return x
```

④ Osservazione: Rispetto del Lower Bound

Essendo il costo per ordinare un BST $\Theta(n)$, per il lower bound dei comparison sort, la costruzione del BST dovrà avere un costo di almeno $\Theta(n \log(n))$.

Similmente allo heap, il BST è quindi una via di mezzo tra un vettore completamente disordinato ed uno ordinato

- Nello heap abbiamo una costruzione in $\Theta(\log(n))$ e un ordinamento in $\Theta(n \log(n))$
- Nel BST abbiamo una costruzione in $\Theta(n \log(n))$ e un ordinamento in $\Theta(n)$

Ricerca

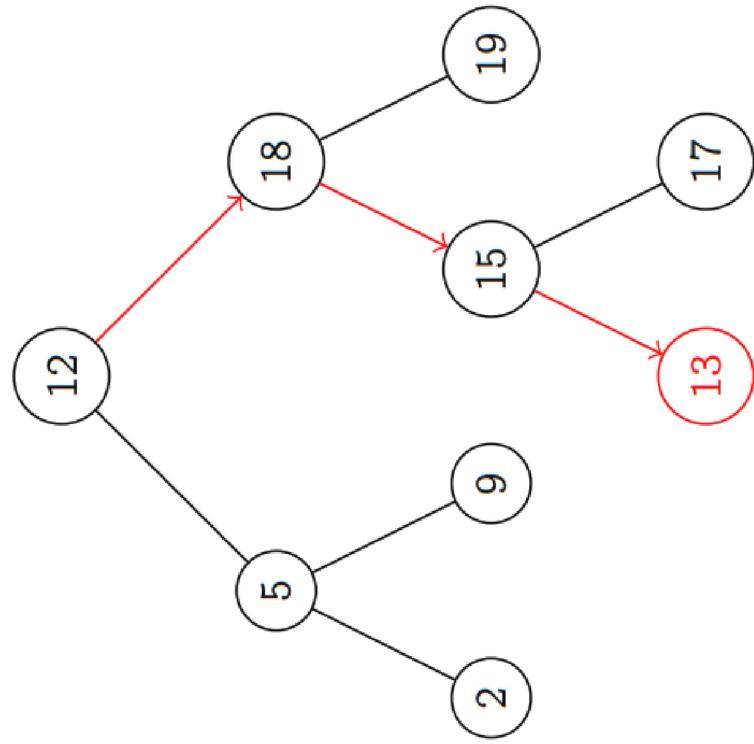
② Idea

- **Massimo:** se un nodo ha un figlio destro, questo sarà più grande di lui
- **Minimo:** se un nodo ha un figlio sinistro, questo sarà più piccolo di lui

③ Idea

Scorri l'albero scegliendo se cercare a sinistra o destra a seconda se il valore è minore o maggiore della chiave corrente

$\Theta(h)$



① Ricerca Iterativa

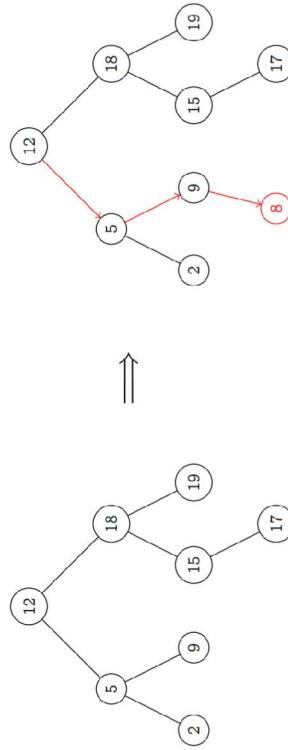
ITERATIVE-TREE-SEARCH(x, k)

```

1 while x != NIL and k != x.key
2   if k < x.key
3     x = x.left
4   else
5     x = x.right
6 return x
  
```

Inserimento

Idea
Operiamo come per la [ricerca](#) fino ad arrivare alla posizione vuota in cui inseriremo il nostro valore



TREE-INSERT(T, z)

```

1 x = T.root
2 y = NIL // padre della root
  
```

TREE-SEARCH(x, k)

```

1 if x == NIL or x.key == k
2   return x
3 if k < x.key
4   return TREE-SEARCH(x.left, k)
5 else
6   return TREE-SEARCH(x.right, k)
  
```

Parametri: x =nodo radice, k =chiave da cercare

Return: nodo con chiave k , NIL se inesistente

① Costo Computazionale

Return: nuova radice dell'albero

```

3 while x != NIL // scorrimento analogo alla ricerca
4     y = x // salvataggio del nodo padre ad ogni ciclo
5     if z.key < x.key
6         x = x.left
7     else
8         x = x.right
9     z.p = y
10    if y == NIL // albero vuoto
11        T.root = z
12    else if z.key < y.key
13        y.left = z
14    else
15        y.right = z

```

Parametri: $T=BT$, $z=nodo da inserire$

④ **Costo Computazionale**
 $\Theta(h)$

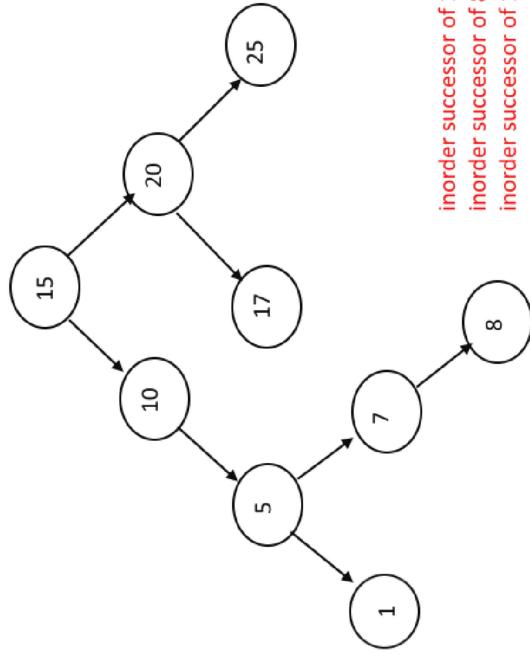
Successore e Predecessore

Il successore di un nodo x è il nodo più piccolo tra i maggiori di x

- Se ha **figli a destra**, il successore sarà il **minimo** tra loro
- In caso contrario, essendo i figli sinistri tutti minori, devo **cercare tra i padri**: risalgo l'albero finché x è figlio destro, per poi prendere il padre

Il predecessore, analogamente, sarà

- Se ha figli a sinistra, il predecessore sarà il **massimo** tra loro
- In caso contrario, essendo i figli destri tutti maggiori, devo cercare tra i padri: risalgo l'albero finché x è figlio sinistro, per poi prendere il padre



inorder successor of 1 is 5
inorder successor of 8 is 10
inorder successor of 15 is 17

④ **Inserimento Ricorsivo**

RECURSIVE-TREE-INSERT(x, p, z)

```

1   if x == NIL
2       x = z
3       x.p = p
4   else if z.key < x.key
5       x.left = RECURSIVE-TREE-INSERT(x.left, x, z)
6   else
7       x.right = RECURSIVE-TREE-INSERT(x.right, x, z)
8   return x

```

Parametri: $x=nodo radice$, $p=nodo padre$ (NIL se x è la radice), $z=nodo da inserire$

TREE-SUCCESSOR(x)

- Se ha **un solo figlio**, andrà inserito al posto di z
 - Se z ha **due figli**, andrà scambiato di posto con il suo successore ed eliminato di nuovo z ricorsivamente
- ^idea-cancellazione
Una rappresentazione di ogni caso è

```

1 if x.right != NIL
2   return TREE-MINIMUM(x.right)
3 y = x.p
4 while y != NIL and x == y.right
5   x = y
6   y = y.p
7 return y

```

Parametri: x =nodo di cui trovare il successore
Return: nodo successore di x , NIL se inesistente

TREE-PREDECESSOR(x)

```

1 if x.left != NIL
2   return TREE-MAXIMUM(x.left)
3 y = x.p
4 while y != NIL and x == y.left
5   x = y
6   y = y.p
7 return y

```

Parametri: x =nodo di cui trovare il predecessore
Return: nodo predecessore di x , NIL se inesistente

④ Costo Computazionale

$$\Theta(h)$$

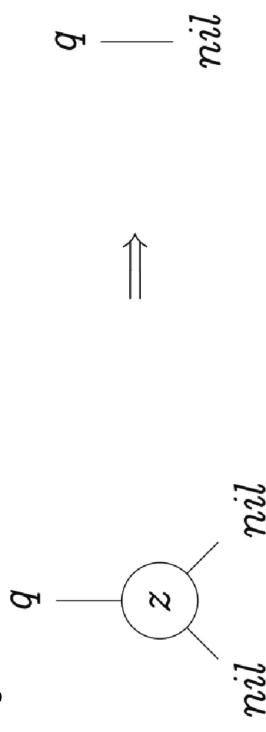
Cancellazione

💡 Idea

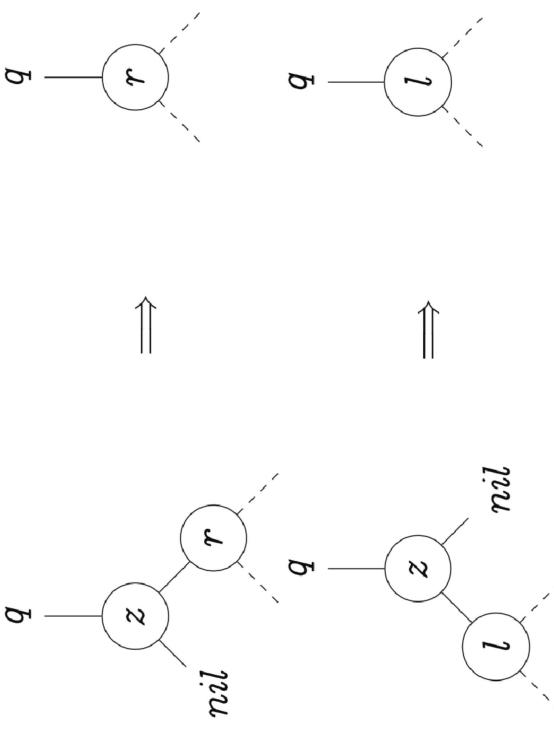
Per mantenere la struttura del BST, la cancellazione di un nodo si comporterà in modo diverso a seconda del numero di figli del nodo z da eliminare

- Se **non ha figli**, il nodo va semplicemente rimosso

- Nessun figlio:

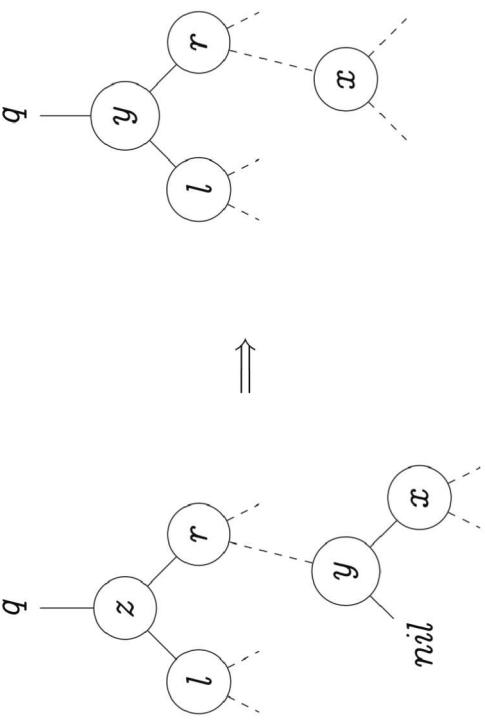


- Un solo figlio:



Per mantenere la struttura del BST, la cancellazione di un nodo si comporterà in modo diverso a seconda del numero di figli del nodo z da eliminare

- Due figli:



```

22     else // z è il figlio destro
23     z.p.right = x
24
25     if x != NIL // punta il nuovo padre
26     x.p = z.p
  
```

Parametri: $T = \text{BST}$, $z = \text{nodo da eliminare}$

④ Costo Computazionale

 $\Theta(h)$

TREE-DELETE(T, z)

```

1 let x be a pointer to a node
2
3 if z.left != NIL and z.right != NIL // entrambi i figli
4   x = TREE-SUCCESSOR(z)
5   z.data = x.data
6   z.key = x.key
7   TREE-DELETE(T, x)
8
9   if z.left == NIL and z.right == NIL // nessun figlio
10   x = NIL
11   else if z.left == NIL and z.right != NIL // solo
12   destro
13   x = z.right
14   sinistro
15   x = z.left
16
17   // imposta x al posto di z
18   if z.p == NIL // z è la radice
19   T.root = x
20   else
21   if z.p.left == z // z è il figlio sinistro
22   z.p.left = x
23
24   if x != NIL // punta il nuovo padre
25   x.p = z.p
  
```

Costruzione (da sistemare)

④ Problema

Dato un vettore V di n elementi, costruire un BST

Si come sappiamo che la costruzione del BST deve richiedere un costo $\Omega(n \log(n))$, ordiniamo il vettore e costruiamo il BST come una **lista partendo dal fondo**

💡 Idea

Verrà costruito un BST completamente sbilanciato a destra

TREE-BUILD(V, n)

```

1 SORT(V)
2
3 let T be an empty BST
4 prev = NIL
5 for i=N downto 1
6   let n be a new empty BST node
  
```

```

7   n.left = RECURSION-TREE-BUILD-EVEN(V, p, m - 1)
8   n.right = RECURSION-TREE-BUILD-EVEN(V, m + 1, r)
9
10  if n.left != NIL
11    n.left.p = n
12    if n.right != NIL
13      n.right.p = n
14
15  return n

```

Parametri: V=vettore, n=numero di elementi
Return: BST

④ Costo Computazionale

$\Theta(n \log(n))$

Costruzione Bilanciata

④ Problema

Dato un vettore V di n elementi, costruire un BST perfettamente bilanciato

💡 Idea

Prendendo il mediano di un vettore, possiamo costruire un BST perfettamente bilanciato inserendolo e chiamando ricorsivamente sulle due metà

Parametri: V=vettore, p=indice di partenza di V, r=indice di fine di V
Return: nodo radice del BST bilanciato

TREE-BUILD-EVEN(V, n)

```

1  SORT(V)
2
3  let T be an empty BST
4  T.root = RECURSION-TREE-BUILD-EVEN(V, 0, n)
5  return T

```

Parametri: V=vettore, n=numero di elementi
Return: BST bilanciato

Sapendo che l'ordinamento richiede $\Theta(n \log(n))$, troviamo l'equazione ricorsiva della parte restante.

$$T(n) = 2T\left(\frac{n}{2}\right) + 1$$

RECURSION-TREE-BUILD-EVEN(V, p, r)

```

1  if p > r
2    return NIL
3
4  let n be an empty BST node
5  m = floor((r - p) / 2)
6  n.key = V[m]

```

④ Costo Computazionale: Bilanciato

$\Theta(n \log(n))$

Red-Black Tree

Proprietà: Relazione Altezza Nera-Altezza dell'altezza

Siccome le operazioni sui BST sono sempre dipendenti dall'altezza dell'albero h , vogliamo imporre delle regole per essere sicuri di mantenere bilanciato l'albero e quindi avere altezza logaritmica

💡 Idea

In particolare, dati due nodi padre e figlio

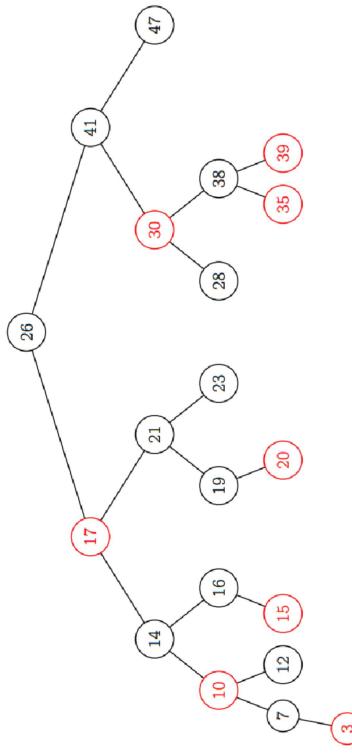
- Se il **padre è rosso** allora $bh(\text{padre}) = bh(\text{figlio})$
- Se il **padre è nero** allora $bh(\text{padre}) = bh(\text{figlio}) - 1$

⚠ Definizione

Un Red-Black Tree è un BST che soddisfa le seguenti proprietà

- Ogni nodo è rosso o nero
- La radice è nera
- Se un nodo è rosso, entrambi i figli sono neri
- Per ogni nodo x , tutti i cammini da x verso le foglie dell'albero radicato in esso hanno lo stesso numero di nodi neri

I nodi avranno quindi una proprietà in più $x.color$



Abbiamo quindi due diverse altezze per ogni nodo x

- **Altezza $h(x)$:** il **numero di nodi** incontrati partendo da x incluso e scendendo verso la foglia più lontana da x
- **Altezza nera $bh(x)$:** il **numero di nodi neri** incontrati partendo da x incluso e scendendo verso una qualsiasi foglia

Per definizione, da un nodo v l'altezza nera è sempre **minore o uguale** dell'altezza

$$bh(v) \leq h(v)$$

$$bh(\text{padre}) - 1 \leq bh(\text{figlio}) \leq bh(\text{padre})$$

Di conseguenza

ⓘ Costo delle Operazioni

Le possibili operazioni su BST svolte su un Red-Black Tree avranno un costo computazionale sempre legato all'altezza dell'albero, che sarà però limitata in quanto bilanciato

$$\Theta(h) = \Theta(\log(n))$$

Upper Bound dell'Altezza

💡 Idea

Per dimostrare che un Red-Black Tree è sempre ben bilanciato, vogliamo trovare un upper-bound per l'altezza

⌚ Teorema

Sia T un Red-Black Tree con n nodi

$$h(T) \leq 2 \log_2(n + 1) = \Theta(\log(n))$$

Dimostrazione

Sia x un nodo dell'albero T , definiamo

- $t(x)$: sottoalbero radicato in x
- $|t(x)|$: numero di nodi di $t(x)$

Dimostriamo che

$$|t(x)| \geq 2^{bh(x)-1}$$

Tenendo presente la relazione tra altezza e altezza nera, procediamo per induzione

Siano L, R rispettivamente il figlio sinistro e destro di x

$$|t(x)| = |t(L)| + |t(R)| + 1$$

Verifichiamo il caso base

$$h(x) = 1 \implies |t(x)| = 1$$

$$\begin{aligned} h(x) = 1 &\implies bh(x) \in \{0, 1\} \\ &\implies 2^{bh(x)} - 1 \in \{0, 1\} \\ &\implies |t(x)| \geq 2^{bh(x)} - 1 \end{aligned}$$

Sapendo che i figli L, R hanno un'altezza minore di x , possiamo procedere con il passo induttivo

$$\begin{aligned} |t(x)| &= |t(L)| + |t(R)| + 1 \\ &\geq 2^{bh(L)} - 1 + 2^{bh(R)} - 1 + 1 \\ &= 2^{bh(L)} + 2^{bh(R)} - 1 \\ &\geq 2^{bh(x)-1} + 2^{bh(x)-1} - 1 \\ &= 2^{bh(x)} - 1 \end{aligned}$$

$$|t(x)| \geq 2^{bh(x)} - 1$$

Abbiamo quindi verificato l'ipotesi sul numero di nodi.

Nel caso peggiore di altezza, avremo i nodi neri e rossi alternati, di conseguenza possiamo dire che

$$bh(T.root) \geq \frac{h(T.root)}{2}$$

Quindi, utilizzando l'ipotesi verificata e questa proprietà

$$\begin{aligned} n &\geq 2^{bh(T.root)} - 1 \\ n &\geq 2^{h(T.root)/2} - 1 \\ n+1 &\geq 2^{h(T.root)/2} \\ \log_2(n+1) &\geq \frac{h(T.root)}{2} \end{aligned}$$

$$h(T.root) \leq 2 \log_2(n+1)$$

$$|t(T.root)| = n$$