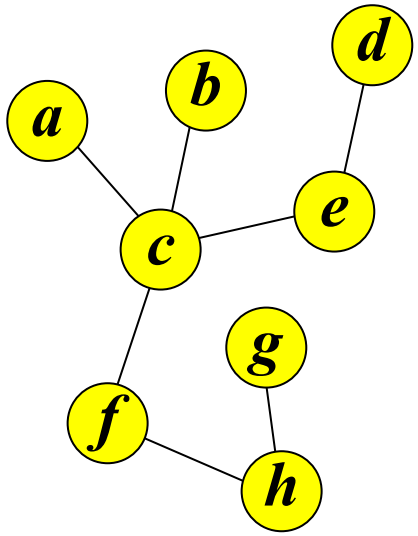
Strutture Dati, Algoritmi e Complessità

ALBERI BINARI DI RICERCA

AA 2023-2024

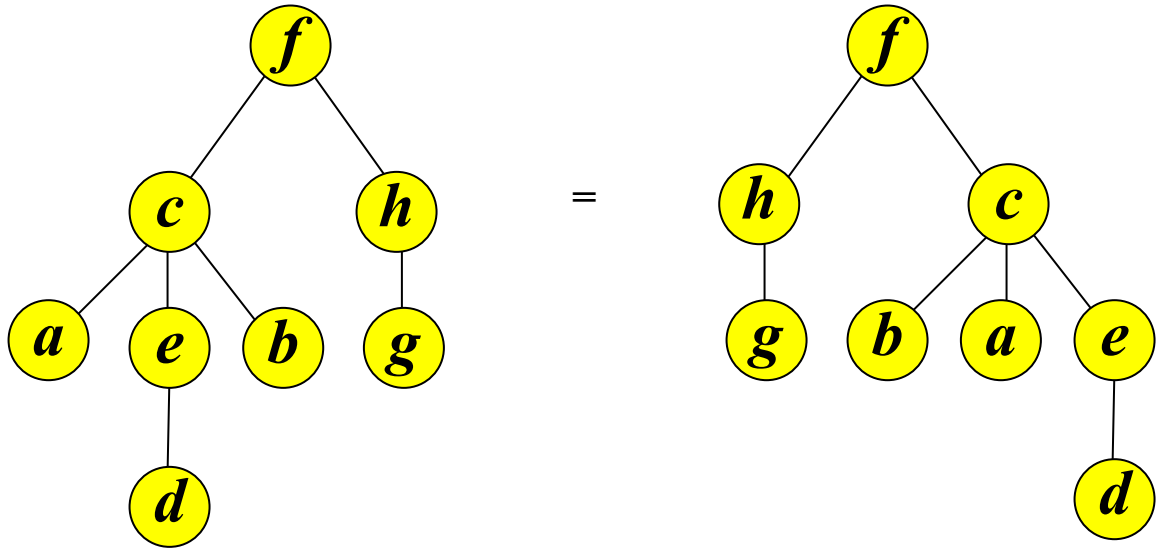
Alberi

Alberi liberi : grafi non orientati connessi e senza cicli (DAG connessi)



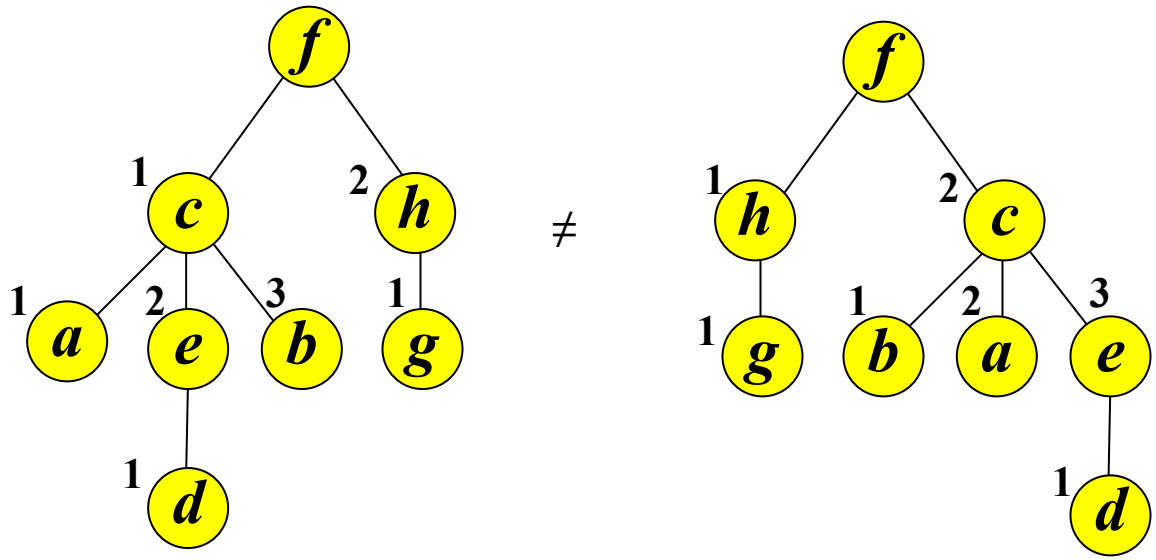
Alberi

Alberi radicati: alberi liberi in cui un vertice è stato scelto come radice.



Alberi

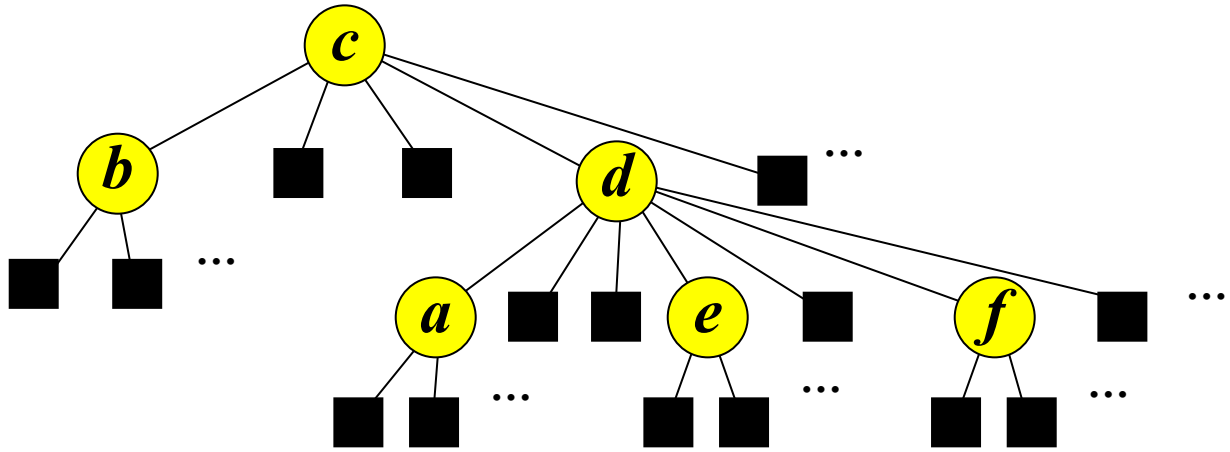
Alberi ordinati : alberi radicati con un ordine tra i figli di un nodo.



Alberi

Alberi posizionali : alberi radicati in cui ad ogni figlio di un nodo è associata una posizione.

- Le posizioni che non sono occupate da un nodo sono posizioni vuote (nil o ■).

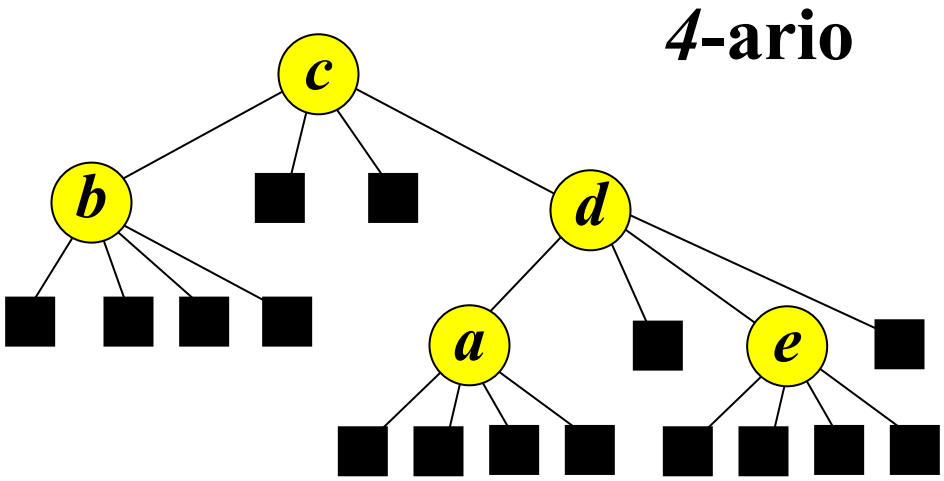


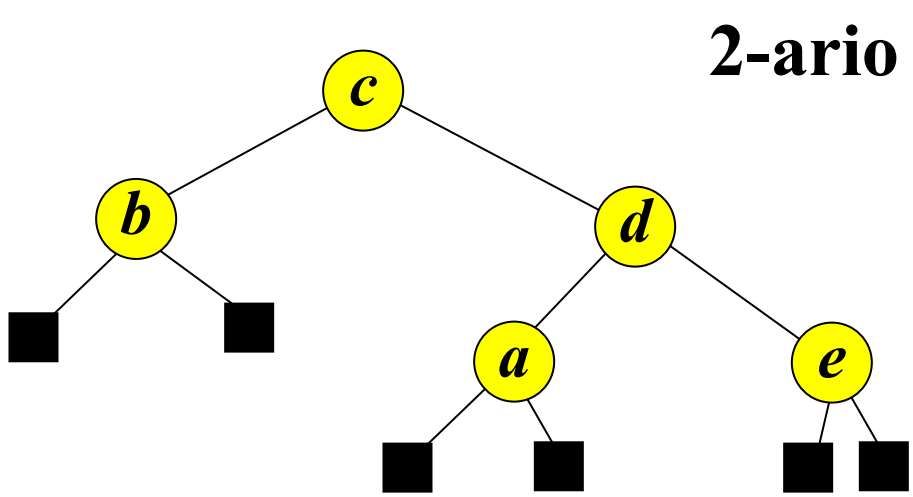
Alberi

Alberi k-ari : alberi posizionali in cui ogni posizione maggiore di ***k*** è vuota.

Alberi binari : alberi ***k***-ari con ***k* = 2.**

- Il figlio in posizione **1** si dice ***figlio sinistro*** e quello in posizione **2** si dice ***figlio destro.***

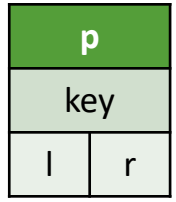


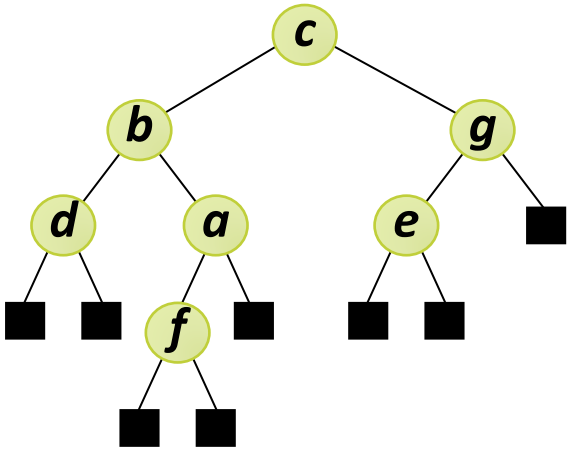


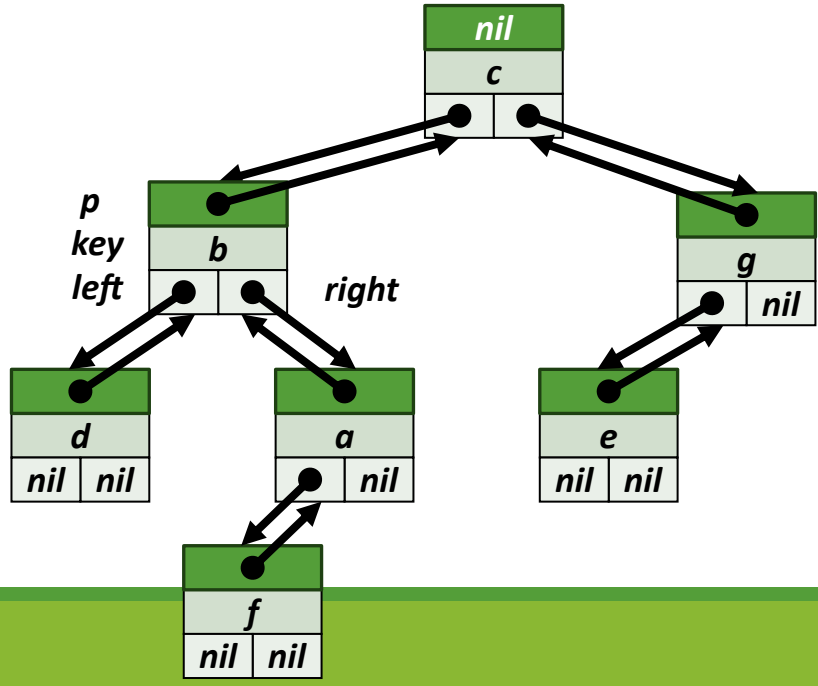
Rappresentazione di alberi binari

Ogni nodo dell'albero è composto dai seguenti attributi

- p: puntatore al nodo padre
- key: chiave del nodo (memorizza la sua etichetta)
- l: puntatore al figlio sinistro
- r: puntatore al figlio destro





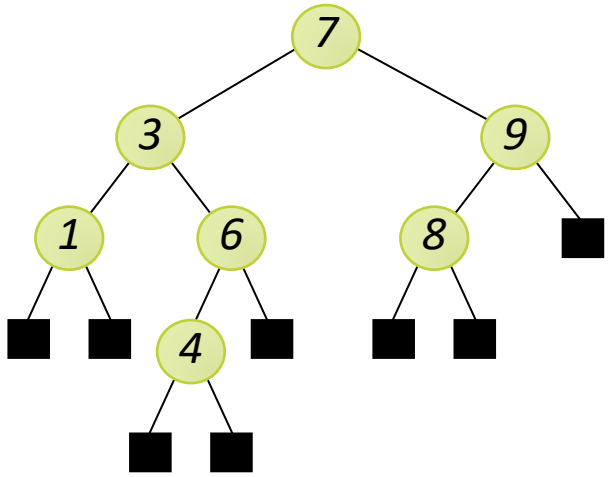


Alberi Binari di Ricerca (BST)

Un albero binario di ricerca (BST) è un albero binario che soddisfa la seguente proprietà

Sia x un nodo di un albero binario di ricerca.

- Se y è un nodo del sottoalbero sinistro di x allora y.key ≤ x.key.
- Se y è un nodo del sottoalbero destro di x allora y.key ≥ x.key.



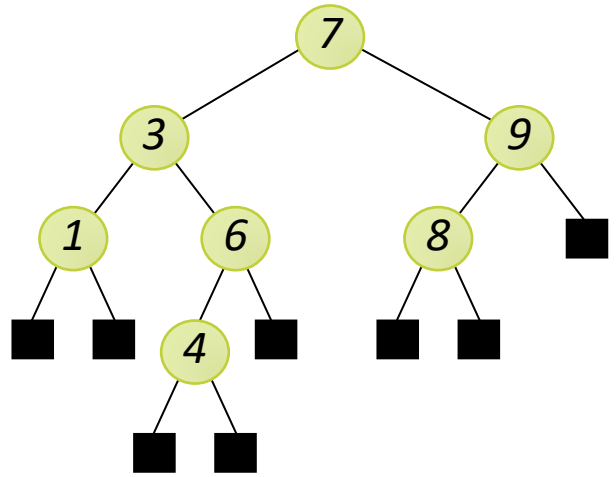
Visita di un albero binario

Esistono 3 principali metodi per visitare i nodi dell'albero

- attraversamento simmetrico o visita in ordine (*inorder tree walk*)
  - la chiave della radice di ogni sottoalbero viene stampata nel mezzo dei valori del suo sottoalbero sinistro e destro (e quindi le chiavi sono stampante in ordine crescente)
- attraversamento anticipato (*preorder tree walk*)
  - stampa la radice prima dei valori dei suoi sottoalberi
- attraversamento posticipato (*postorder tree walk*)
  - stampa la radice dopo i valori dei suoi sottoalberi

Attraversamento Simmetrico

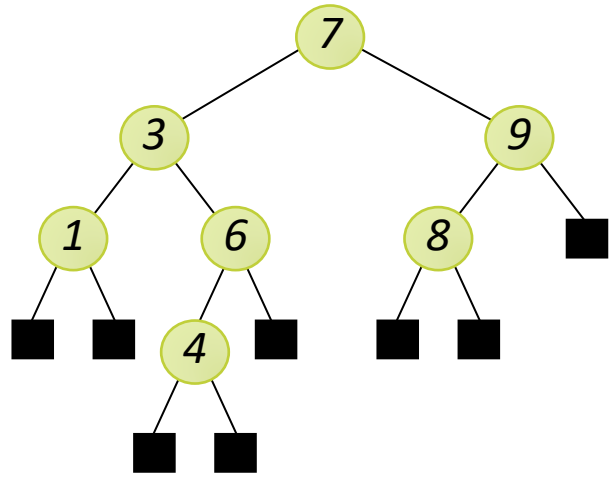
```
Inorder(x)
    if x ≠ nil then
        Inorder(x.left)
        print x.key
        Inorder(x.right)
```



1 3 4 6 7 8 9

Attraversamento Anticipato

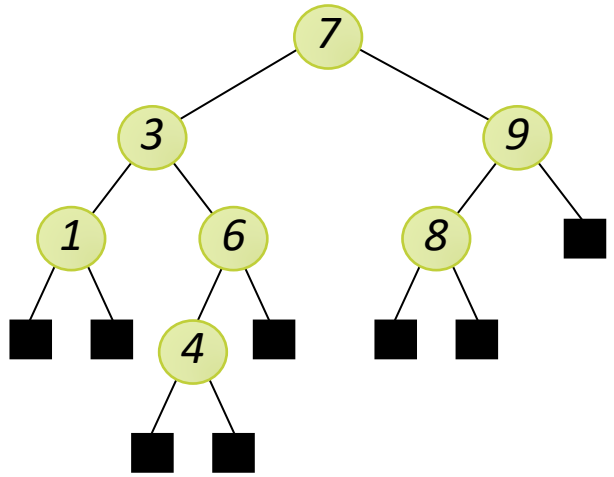
```
Pre-order(x)
    if x ≠ nil then
        print x.key
        Pre-order(x.left)
        Pre-order(x.right)
```



7 3 1 6 4 9 8

Attraversamento Posticipato

```
Post-order(x)
    if x ≠ nil then
        Post-order(x.left)
         Post-order(x.right)
        print x.key
```



1 4 6 3 8 9 7

Analisi della complessità

TEOREMA
Se *x* è la radice di un sottoalbero con *n* nodi allora Inorder(*x*) richiede un tempo Θ (n)

DIMOSTRAZIONE

Sia T(n) il tempo richiesto da Inorder(x) per visitare un sottoalbero di n nodi

- T(n) è banalmente Ω(n) (devo visitare tutti gli n nodi)
- Mi resta quindi da dimostrare che T(n) è O(n)

Analisi della complessità

DIMOSTRAZIONE

n=0

- Per visitare un nodo mi serve una quantità piccola di tempo (test if x ≠ *nil*)
  - -> T(0) = c

n>0

- Consideriamo i due sottoalberi e assumiamo che
  - il sottoalbero sinistro abbia k nodi
  - il sottoalbero destro ne ha n-k-1
  - T(n) ≤ T(k) + T(n-k-1) + d
- Applicando il metodo della sostituzione ottengo
  - $T(0) = c = (c + d)0 + c$
  - T(n) ≤ T(k) + d + T(n-k-1) =
  - = (c + d)k +c+d+(c +d)(n-k-1)+c
  - = (c +d)n +c

Interrogazioni di un albero binario di ricerca

Un albero binario di ricerca supporta le seguenti interrogazioni

- SEARCH(x, key)
  - data la radice x e una chiave key restituisce il puntatore a un nodo con chiave key (se esiste) o Nil
- MINIMUM(x)/MAXIMUM(x)
  - dato un albero con radice x, restituisce l'elemento la cui chiave è minima/massima
- SUCCESSOR(x)/PREDECESSOR(x)
  - dato un albero con radice x, restituisce il nodo successore/precedente a x rispetto all'attraversamento simmetrico

Ricerca

METODO RICORSIVO

```
Search(x,k)
   if x == nil or k == x.key
      return x
   if k < x.key
      return Search(x.left,k)
   else return Search(x.right,k)
```

METODO ITERATIVO

```
Search(x,k)
   while x ≠ nil and k ≠ x.key
      if k < x.key
         x = x.left
      else x=x.right
   return x
```

Costo della ricerca O(h)

dove h è l'altezza dell'albero

Minimo e Massimo

MINIMUM(X)

```
Minimum(x)
  while x.left ≠ nil
    x = x.left
  return x
```

MAXIMUM(X)

```
Maximum(x)
  while x.right ≠ nil
    x = x.right
  return x
```

Costo O(h)

dove h è l'altezza dell'albero

Successore e Predecessore

SUCCESSOR(X)

```
Successor(x)
   if x.right ≠ nil
      return Minimum(x.right)
   else
      y = x.p
   while y ≠ nil and x = y.right
      x = y, y = x.p
   return y
```

PREDECESSOR(X)

Il predecessore si ottiene mettendo *left* al posto di *right* e Maximum al posto di Minimum.

Costo O(h)

dove h è l'altezza dell'albero

Inserimento e Cancellazione

Le operazioni di inserimento e cancellazione agiscono modificando il contenuto della struttura dati

Nel caso degli alberi binari di ricerca queste operazioni devono essere realizzare preservando la proprietà dell'ordinamento delle chiavi

E' necessario quindi verificare se un inserimento/cancellazione impattano tale proprietà ed in caso aggiornare l'albero per ripristinarla

Inserimento

IDEA

- attacco il nuovo nodo come una voglia
  - mi basta quindi trovare una foglia che possa fargli «da padre» senza compromettere le proprietà dell'albero binario

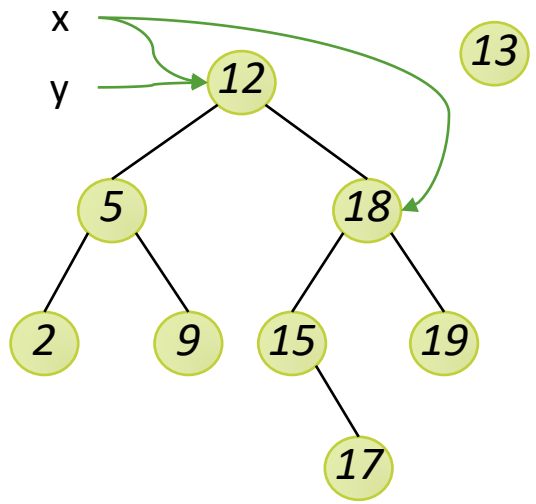
```
Insert(T, z)              // z.left = z.right = nil
  x = T.root, y = nil     // y padre di x
  while x ≠ nil           // cerco dove mettere z
    y = x
    if z.key < y.key
       x = y.left
    else x = y.right
  z.p = y      // metto z al posto della foglia x
  if y == nil
     T.root = z
  elseif z.key < y.key
     y.left = z
  else y.right = z
```

Costo O(h)

dove h è l'altezza dell'albero

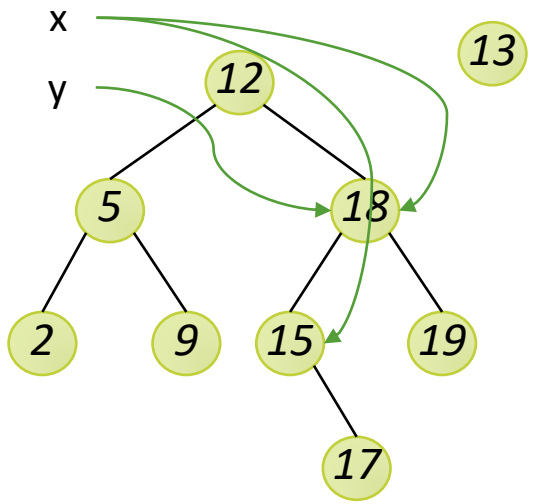
Esempio

```
Insert(T, z)              // z.left = z.right = nil
  x = T.root, y = nil     // y padre di x
  while x ≠ nil           // cerco dove mettere z
    y = x
    if z.key < y.key
       x = y.left
    else x = y.right
  z.p = y      // metto z al posto della foglia x
  if y == nil
     T.root = z
  elseif z.key < y.key
     y.left = z
  else y.right = z
```



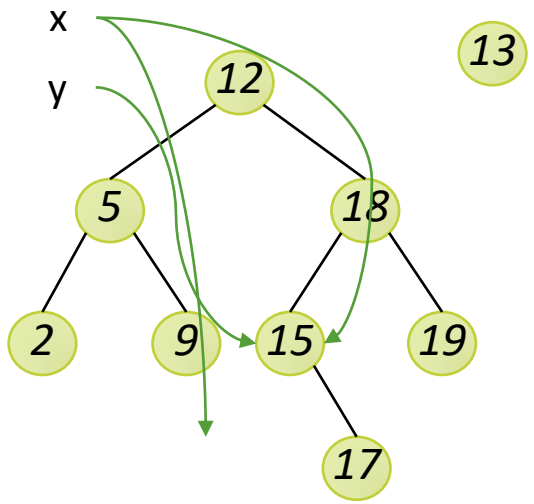
Esempio

```
Insert(T, z)              // z.left = z.right = nil
  x = T.root, y = nil     // y padre di x
  while x ≠ nil           // cerco dove mettere z
    y = x
    if z.key < y.key
       x = y.left
    else x = y.right
  z.p = y      // metto z al posto della foglia x
  if y == nil
     T.root = z
  elseif z.key < y.key
     y.left = z
  else y.right = z
```



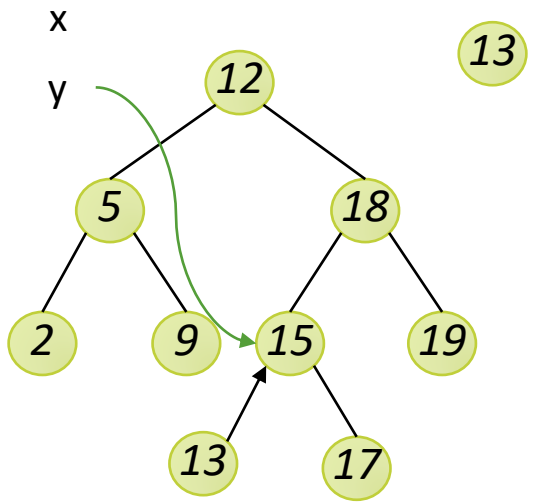
Esempio

```
Insert(T, z)              // z.left = z.right = nil
  x = T.root, y = nil     // y padre di x
  while x ≠ nil           // cerco dove mettere z
    y = x
    if z.key < y.key
       x = y.left
    else x = y.right
  z.p = y      // metto z al posto della foglia x
  if y == nil
     T.root = z
  elseif z.key < y.key
     y.left = z
  else y.right = z
```



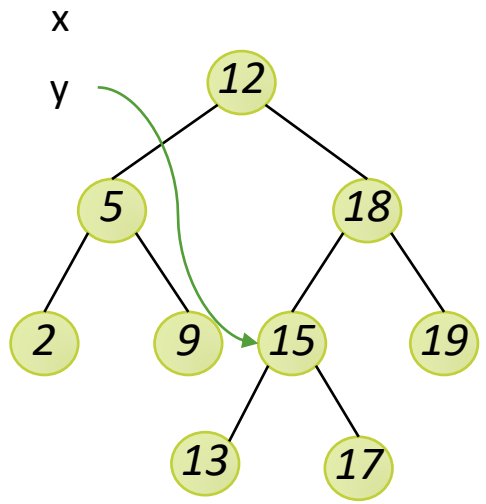
Esempio

```
Insert(T, z)              // z.left = z.right = nil
  x = T.root, y = nil     // y padre di x
  while x ≠ nil           // cerco dove mettere z
    y = x
    if z.key < y.key
       x = y.left
    else x = y.right
  z.p = y      // metto z al posto della foglia x
  if y == nil
     T.root = z
  elseif z.key < y.key
     y.left = z
  else y.right = z
```



Esempio

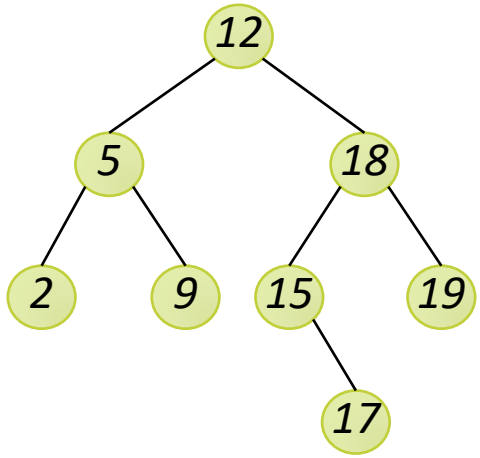
```
Insert(T, z)                // z.left = z.right = nil
  x = T.root, y = nil       // y padre di x
  while x ≠ nil             // cerco dove mettere z
    y = x
    if z.key < y.key
       x = y.left
    else x = y.right
  z.p = y       // metto z al posto della foglia x
  if y == nil
     T.root = z
  elseif z.key < y.key
     y.left = z
  else y.right = z
```



Cancellazione

A seconda del nodo *z* che vogliamo cancellare ci si può trovare in 3 diverse situazioni

1. *z* è una foglia (ha 0 figli)
   - caso facile: lo eliminiamo mettendo a nil il puntatore del padre

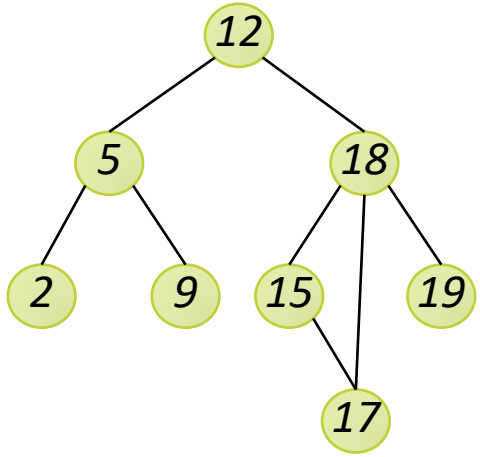


Cancellazione

A seconda del nodo *z* che vogliamo cancellare ci si può trovare in 3 diverse situazioni

1. *z* è una foglia (ha 0 figli)
   - caso facile: lo eliminiamo mettendo a nil il puntatore del padre

2. *z* ha un solo figlio *w*
   - caso facile: *w* viene «promosso» e prende il posto di *z* nell'albero (devo semplicemente aggiornare il puntatore del padre di *z* per farlo puntare a *w*)



Cancellazione

A seconda del nodo *z* che vogliamo cancellare ci si può trovare in 3 diverse situazioni

1. *z* è una foglia (ha 0 figli)
   - caso facile: lo eliminiamo mettendo a nil il puntatore del padre

2. *z* ha un solo figlio *w*
   - caso facile: *w* viene «promosso» e prende il posto di *z* nell'albero (devo semplicemente aggiornare il puntatore del padre di *z* per farlo puntare a *w*)

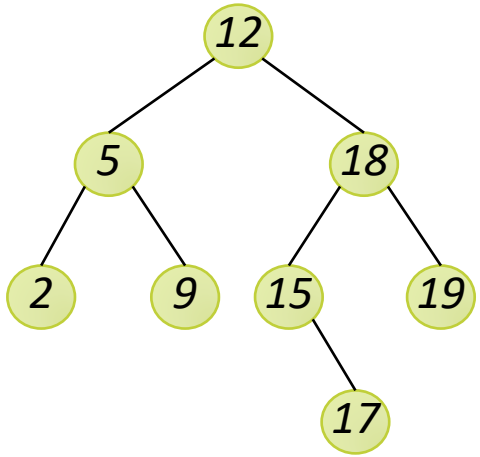
3. *z* ha 2 figli
   - caso più complicato
     1. troviamo il successore y di z (nel sottoalbero destro)
     2. spostiamo y nella posizione di z
     3. ricolleghiamo i sottoalberi

Come ricollegare i sottoalberi dipende da come erano collegati y e z

Cancellazione

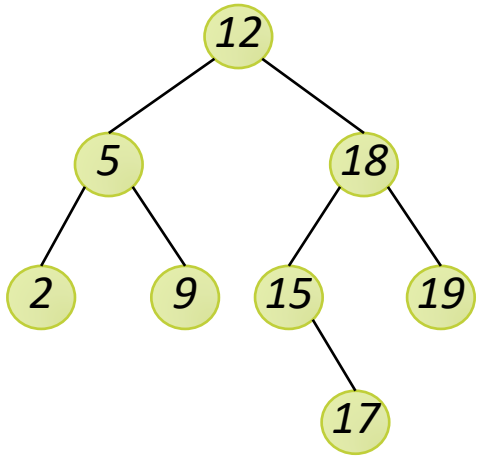
z ha 2 figli

- trovo il successore y di z (che sarà nel sottoalbero destro di z e non ha figlio sinistro)
- stacco y dalla sua posizione corrente
- se y è figlio destro di z semplicemente lo faccio risalire e gli attacco il vecchio sottoalbero sinistro di z
- altrimenti scambio y con il suo figlio destro e poi faccio salire y al posto di z



Cancellazione

3. *z* ha 2 figli
   - trovo il successore y di z (che sarà nel sottoalbero destro di z e non ha figlio sinistro)
   - stacco y dalla sua posizione corrente
     1. se y è figlio destro di z semplicemente lo faccio risalire e gli attacco il vecchio sottoalbero sinistro di z
     2. altrimenti scambio y con il suo figlio destro e poi faccio salire y al posto di z



Cancellazione

Durante la cancellazione di un nodo, i sottoalberi devono spostarsi all'interno dell'albero di ricerca binario

- Transplant sostituisce un sottoalbero, come figlio di suo padre, con un altro sottoalbero (di fatto fa lo swap)

Quando Transplant sostituisce il sottoalbero radicato nel nodo u con il sottoalbero radicato nel nodo v, il padre del nodo u diventa il padre del nodo v, e il padre di u alla fine ha v come figlio (destro o sinistro).

Cancellazione

Gestisce i casi 1 e 2

Gestisce caso 3

```
Delete(T, z)
  if z.left == nil
     Transplant(T, z, z.right)
  elseif z.right == nil
     Transplant(T, z, z.left)
  else y = Minimum(z.right)
     if y ≠ z.right
        Transplant(T, y, y.right)
        y.right = z.right
        y.right.p = y
     Transplant(T, z, y)
     y.left = z.left
     y.left.p = y
```

Costo O(h)

dove h è l'altezza dell'albero

```
Transplant(T, u, v)
  if u.p == nil
     T.root = v
  elseif u == u.p.left
     u.p.left = v
  else u.p.right = v
if v ≠ nil
   v.p = u.p
```

Alberi Bilanciati

OSSERVAZIONE

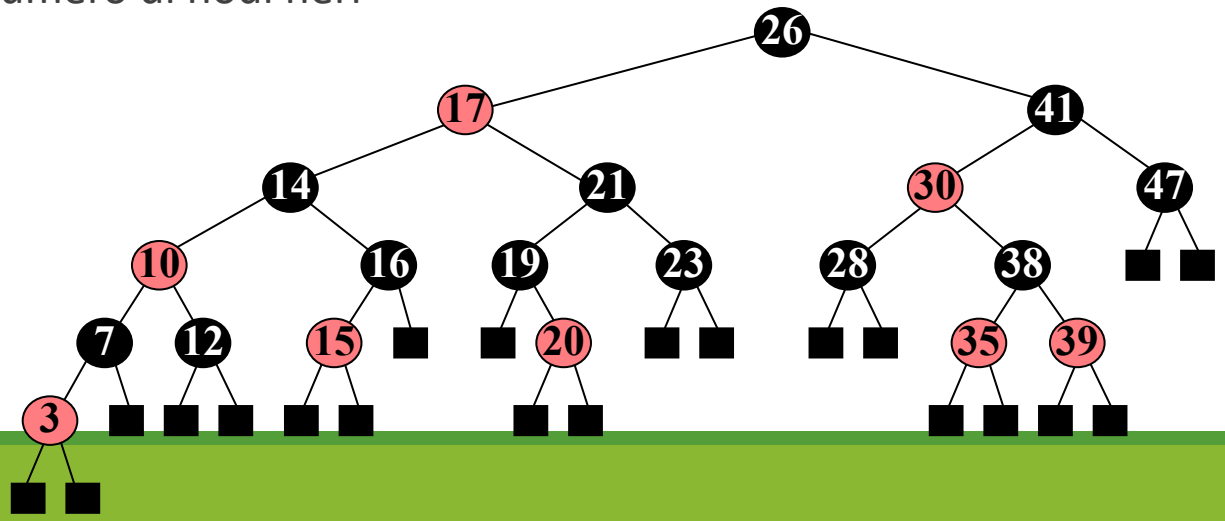
Tutte le le operazioni sugli alberi binari di ricerca hanno complessità proporzionale all'altezza *h* dell'albero.

- Per rendere efficiente l'utilizzo di un albero binario di ricerca è fondamentale mantenere l'albero bilanciato
- Un albero binario di ricerca con *n* nodi è bilanciato se ha altezza O(log n)

Alberi Rosso-Neri

Un albero rosso-nero è un albero binario di ricerca che soddisfa le seguenti condizioni, note come *proprietà degli alberi rosso-neri*:

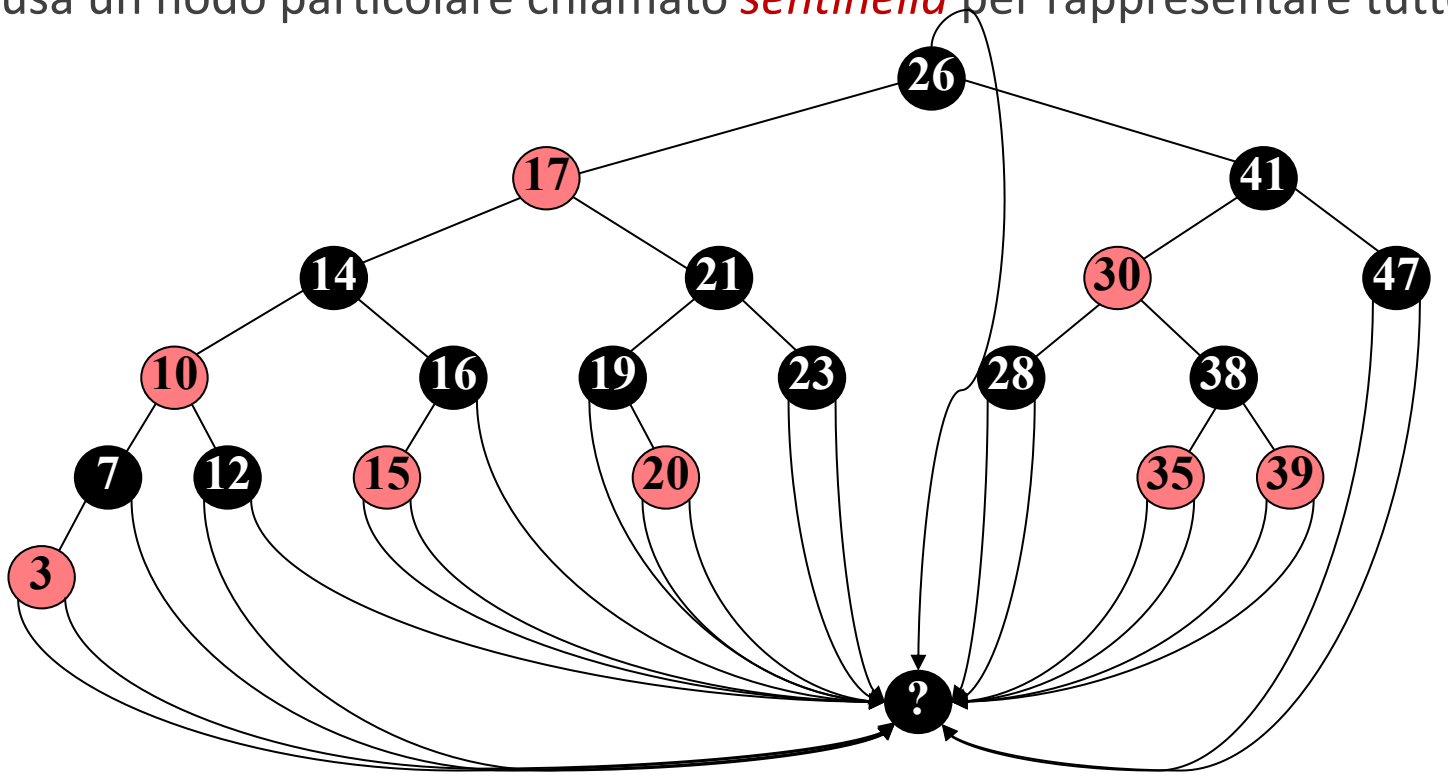
1. ogni nodo è o rosso o nero;
2. la radice è nera;
3. ogni foglia (***nil***) è nera;
4. Se un nodo è rosso allora entrambi i figli sono neri;
5. per ogni nodo ***x***, tutti i cammini semplici che vanno da x alle foglie sue discendenti contengono lo stesso numero di nodi neri



Rappresentazione di un albero rosso-nero

Si aggiunge un attributo per rappresentare il colore del nodo

Su usa un nodo particolare chiamato *sentinella* per rappresentare tutte i nodi nil



Alberi Rosso-Neri

Un albero rosso-nero è un albero binario di ricerca che soddisfa le seguenti condizioni, note come *proprietà degli alberi rosso-neri*:

1. ogni nodo è o rosso o nero;
2. la radice è nera;
3. ogni foglia (***nil***) è nera;
4. Se un nodo è rosso allora entrambi i figli sono neri;
5. per ogni nodo ***x***, tutti i cammini semplici che vanno da x alle foglie sue discendenti contengono lo stesso numero di nodi neri

DEFINIZIONE

L'altezza nera di un nodo *x*, denotata con *bh(x)* è data dal numero di nodi neri lungo un cammino semplice che inizia da *x* (ma non lo include) e termina in una foglia

Alberi rosso-neri

Proprietà: Un albero rosso-nero con ***n*** nodi interni ha altezza

$$h \leq 2 \log_2(n+1)$$

DEFINIZIONE

L'altezza nera di un nodo *x*, denotata con *bh(x)* è data dal numero di nodi neri lungo un cammino semplice che inizia da *x* (ma non lo include) e termina in una foglia

Alberi rosso-neri

DIMOSTRAZIONE

- Dimostriamo prima (per induzione) la seguente proprietà:
  il sottoalbero radicato in un nodo x qualunque contiene almeno $2^{bh(x)}-1$ nodi interni

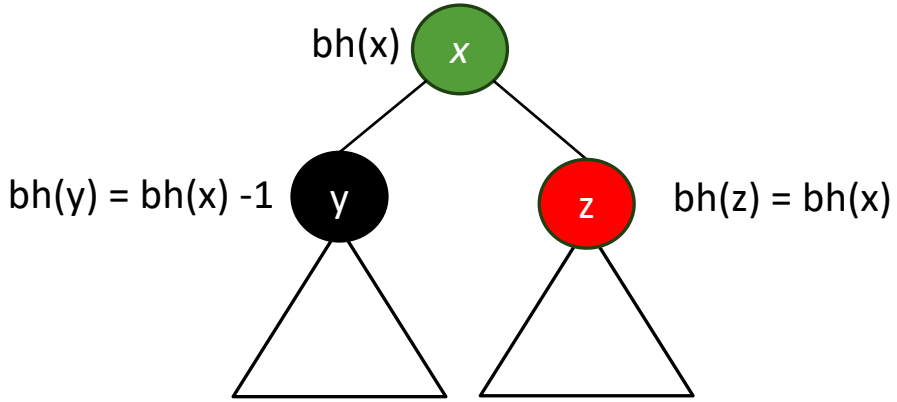
- PASSO BASE
  - h=0 (x è una foglia)
    - Se x è una foglia bh(x)=0
    - $2^0-1 = 1-1 = 0$

Alberi rosso-neri

DIMOSTRAZIONE

- Dimostriamo prima (per induzione) la seguente proprietà:
  il sottoalbero radicato in un nodo x qualunque contiene almeno $2^{bh(x)}-1$ nodi interni

- PASSO INDUTTIVO
  - h>0 (x è un nodo interno)
  - x ha due figli, uno dei quali (o entrambi) potrebbe essere una foglia

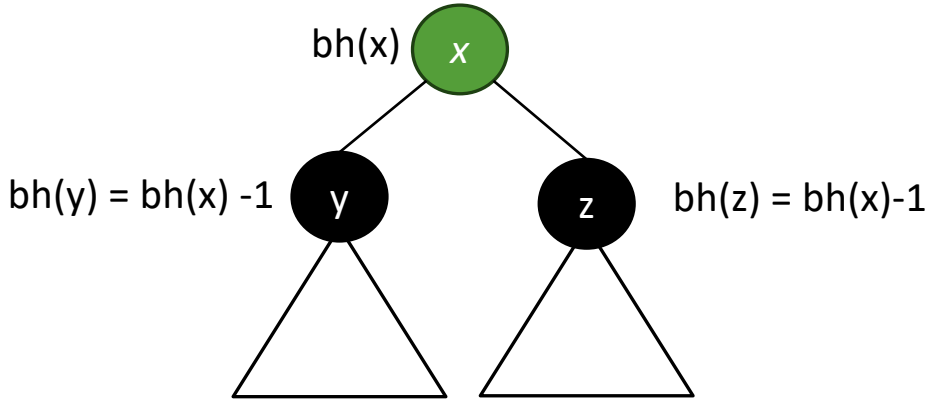


Alberi rosso-neri

DIMOSTRAZIONE

- Dimostriamo prima (per induzione) la seguente proprietà:
  il sottoalbero radicato in un nodo x qualunque contiene almeno $2^{bh(x)}-1$ nodi interni

- PASSO INDUTTIVO
  - h>0 (x è un nodo interno)
  - x ha due figli, uno dei quali (o entrambi) potrebbe essere una foglia



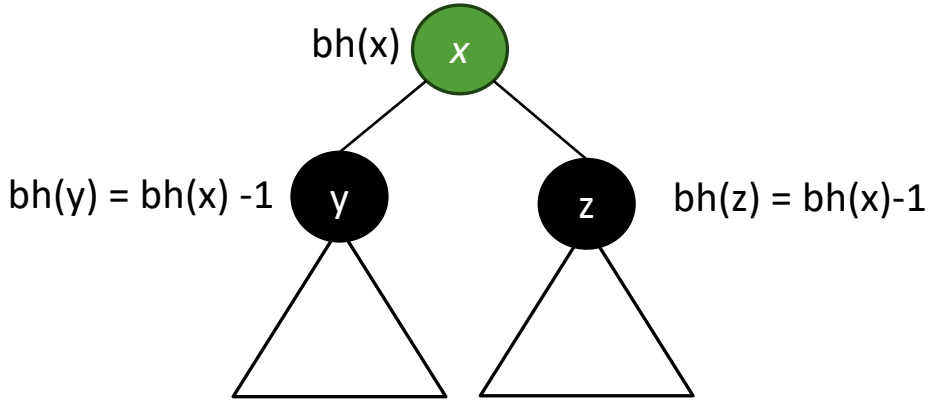
Per l'ipotesi induttiva, poiché l'altezza dei figli di x è inferiore all'altezza h di x, possiamo affermare che ciascun figlio ha almeno $2^{bh(x)-1}-1$ nodi interni

Alberi rosso-neri

DIMOSTRAZIONE

- Dimostriamo prima (per induzione) la seguente proprietà:
  il sottoalbero radicato in un nodo x qualunque contiene almeno $2^{bh(x)}-1$ nodi interni

- PASSO INDUTTIVO
  - Quindi, il sottoalbero radicato in x ha almeno $(2^{bh(x)-1}-1) + (2^{bh(x)-1}-1) + 1$ nodi interni da cui segue la proprietà



Alberi rosso-neri

DIMOSTRAZIONE

- Per concludere la dimostrazione, ragioniamo ora sull'altezza dell'albero *h*
  - Per la proprietà 4 degli alberi rosso-neri, almeno la metà dei nodi su un cammino verso le foglie sono neri
  - ne segue che bh(root) ≥ h/2 ->
  - $n \geq 2^{h/2} - 1$
  - $(n+1) \geq 2^{h/2}$
  - $\log(n+1) \geq h/2$
  - $h \leq 2\log(n+1)$

Alberi rosso-neri

Conseguenza:
Su di un albero rosso-nero con n nodi interni le operazioni *Search*, *Minimum*, *Maximum*, *Successor* e *Predecessor* richiedono tutte un tempo

$$O(\log n)$$

Alberi Rosso-Neri

Negli alberi *rosso-neri* le operazioni ***Insert*** e ***Delete*** sono opportunamente modificate inserendo una fare di ribilanciamento per garantire un'altezza dell'albero

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