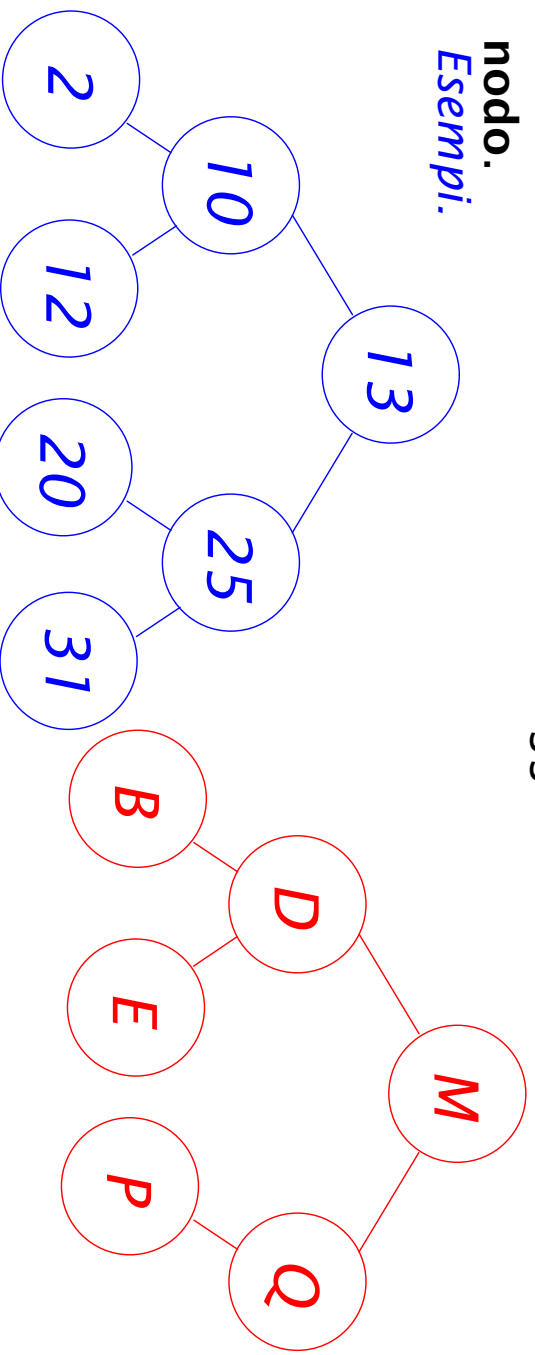


Albero binario di ricerca (BST)

Un albero binario si dice **di ricerca** se, per ogni **nodo**, tutte le etichette del sottoalbero sinistro sono minori dell'etichetta del **nodo**, e tutte le etichette del sottoalbero destro sono maggiori dell'etichetta del **nodo**.

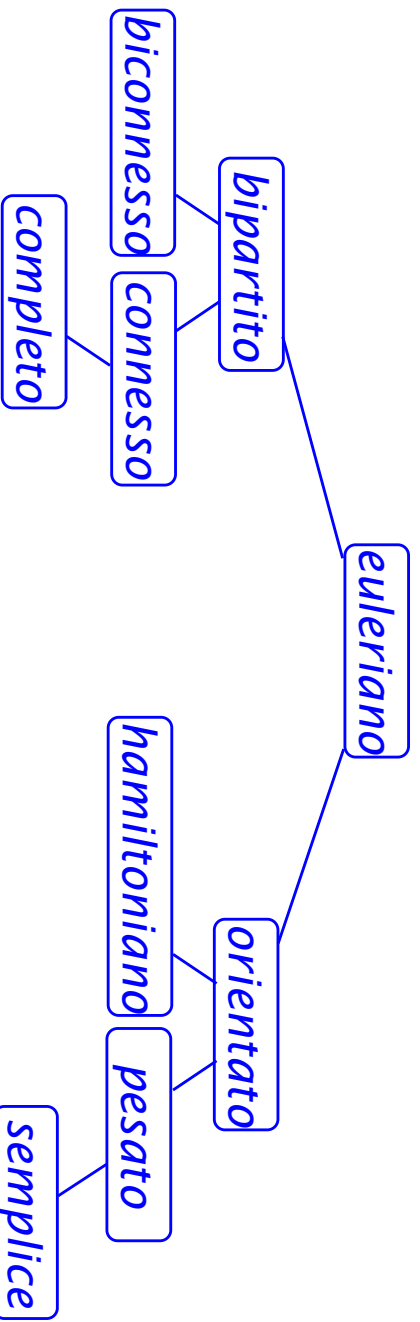
Esempi.



Minore e maggiore hanno senso per qualunque relazione di ordinamento totale.

Esempio di albero binario di ricerca etichettato

con parole dell'alfabeto



Verificare che è binario di ricerca.

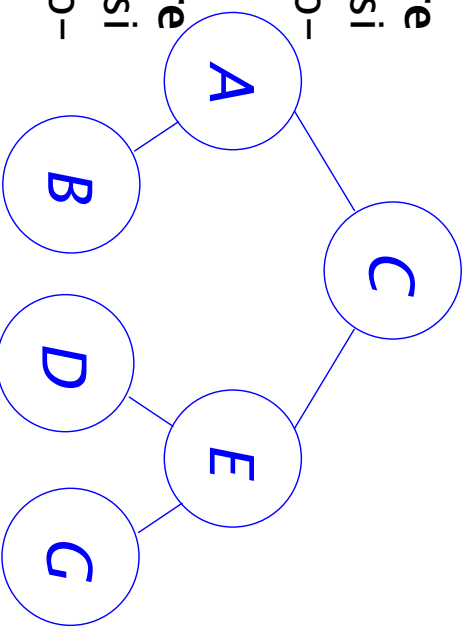
Come attraversare l'albero per ottenere una sequenza di parole ordinata alfabeticamente??

Inorder!

Ricerca su un BST

Decidere se una chiave si trova in un BST. Algoritmo:

- Se il BST è vuoto si restituisca null
- Se la chiave coincide con l'etichetta della radice, si restituisca la radice
- Se la chiave è **minore** dell'etichetta della radice, si esegua l'algoritmo sul sotto-albero **sinistro** della radice
- Se la chiave è **maggiore** dell'etichetta della radice, si esegua l'algoritmo sul sotto-albero **destro** della radice



Implementazione della ricerca su un BST

```
public IntBSTNodo ricerca (IntBSTNodo p, int val)
{
    while (p != null)
    {
        if (val == p.key) return p;
        else if (val < p.key) p = p.left;
        else p = p.right;
    }
    return null;
}
```

Questi sono gli assegnamenti che influenzano la complessità asintotica

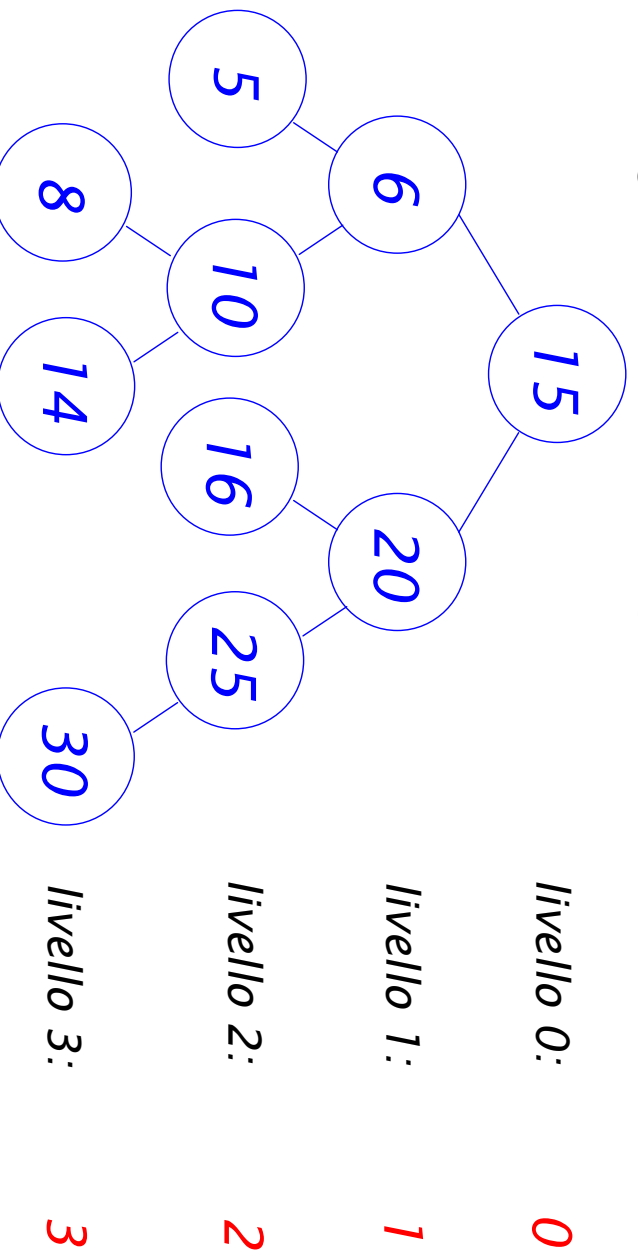
Esercizio: scrivere una versione ricorsiva del metodo

Ricerca su un BST Ricorsiva

```
public BTreeNode search( Comparable x )      {  
    return  search( x, root );              }  
  
private BTreeNode search( Comparable x, BTreeNode t )  
{  
    if( t == null )  
        return null;  
    if( x.compareTo( t.info ) < 0 )  
        return search( x, t.left );  
    else if(x.compareTo( t.info ) > 0 )  
        return search( x, t.right );  
    else  
        return t;      // match      }  
}
```

Complessità della ricerca su un BST

assegnamenti necessari per la ricerca di una chiave al



Quindi il numero di assegnamenti dipende dalla posizione in cui si trova la chiave cercata.

Complessità della ricerca su un BST

- Il numero **massimo** di assegnamenti è **uguale** all'altezza del BST (ovvio)

Quindi $T_p(n) = O(h)$

- Il numero **medio** di assegnamenti è **proporzionale** all'altezza del BST (si può dimostrare)

Quindi $T_{me}(n) = O(h)$

Per esprimere la complessità in funzione del numero di nodi n , ci serve una relazione che legghi l'altezza h ad n .

Complessità della ricerca su un BST

- Se il BST è **completo**, sappiamo che

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

quindi $T_p(n) = T_{me}(n) = O(h) = O(\lg n)$

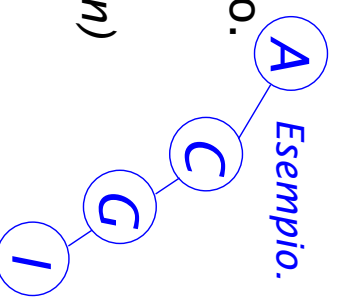
- Se il BST è **bilanciato**, si dimostra che

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

quindi $T_p(n) = T_{me}(n) = O(h) = O(\lg n)$

- Se il BST è **sbalanciato**, studiamo solo la configurazione di massimo sbilanciamento. Ovviamente $h = n - 1$

quindi $T_p(n) = T_{me}(n) = O(h) = O(n)$



Discussione

Pertanto:

- Più l'albero è sbilanciato, più la complessità della ricerca si avvicina a quella della ricerca su una lista, ossia lineare
- Più l'albero è bilanciato, più la complessità della ricerca si avvicina a quella logaritmica

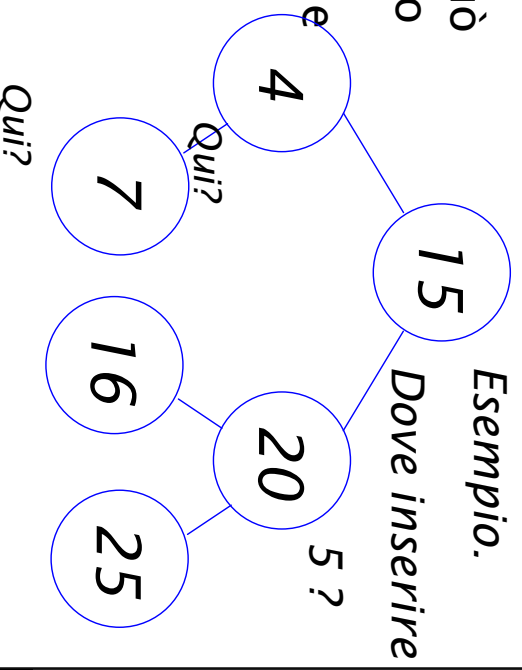
Vorremmo dover lavorare con alberi bilanciati!

Inserimento in un BST

Estendere un BST con (un nuovo nodo contenente) una nuova chiave, mantenendo le proprietà di BST. Algoritmo:

- Se il BST è **vuoto**, si inserisca la chiave in un nuovo nodo che sarà radice
- Si **ricerchi** la chiave. Se si trova, si restituisca false
- Si determini la **foglia** che può essere **padre** del nuovo nodo
- Si innesti il nuovo nodo come **figlio** della foglia trovata

*È più semplice inserire nuove **foglie**.*



Implem. dell'inserimento in un BST

```
public boolean inserisci (int val)
{ if (root == null)
    root = new IntBSTNode(val);          //BST vuoto
  else
  { IntBSTNode p = root, prev = null;
    while (p != null)
    { if (val == p.key)
        return false;    //val già presente
      prev = p;
      if (val < p.key)
        p = p.left;
      else
        p = p.right;
    }
    if (val < prev.key)
      prev.left = new IntBSTNode(val);
    else
      prev.right = new IntBSTNode(val);
    }
  return true;
}
```

Inserimento Ricorsivo

```
public void insert(Comparable x)
{
    root = insert(x, root);
}

private BTNode insert(Comparable x, BTNode t)
{
    if (t == null)
        t = new BTNode( x );
    else if (x.compareTo(t.info) < 0)
        t.left=insert(x, t.left);
    else if (x.compareTo(t.info) > 0)
        t.right = insert(x, t.right);
    else ;    // E' un duplicato;
    return t;
}
```

Discussione

- La **complessità asintotica** dell'inserimento è analoga a quella della ricerca
- L'algoritmo di inserimento visto **non mantiene il bilanciamento** del BST
(ad esempio, se le chiavi vengono inserite in ordine crescente, si ottiene un BST sbilanciato a destra)
- Esercizio: simulare il funzionamento del metodo su vari esempi

Predecessore e Successore

Dato un insieme totalmente ordinato, ha senso parlare di **predecessore** e di **successore nell'insieme** di un elemento dell'insieme.

Esempio.

Dato l'insieme $A = \{b, e, f, i, m, n, r, t, v, w, y\}$
e il suo elemento t

r è il predecessore in A di t

v è il successore in A di t

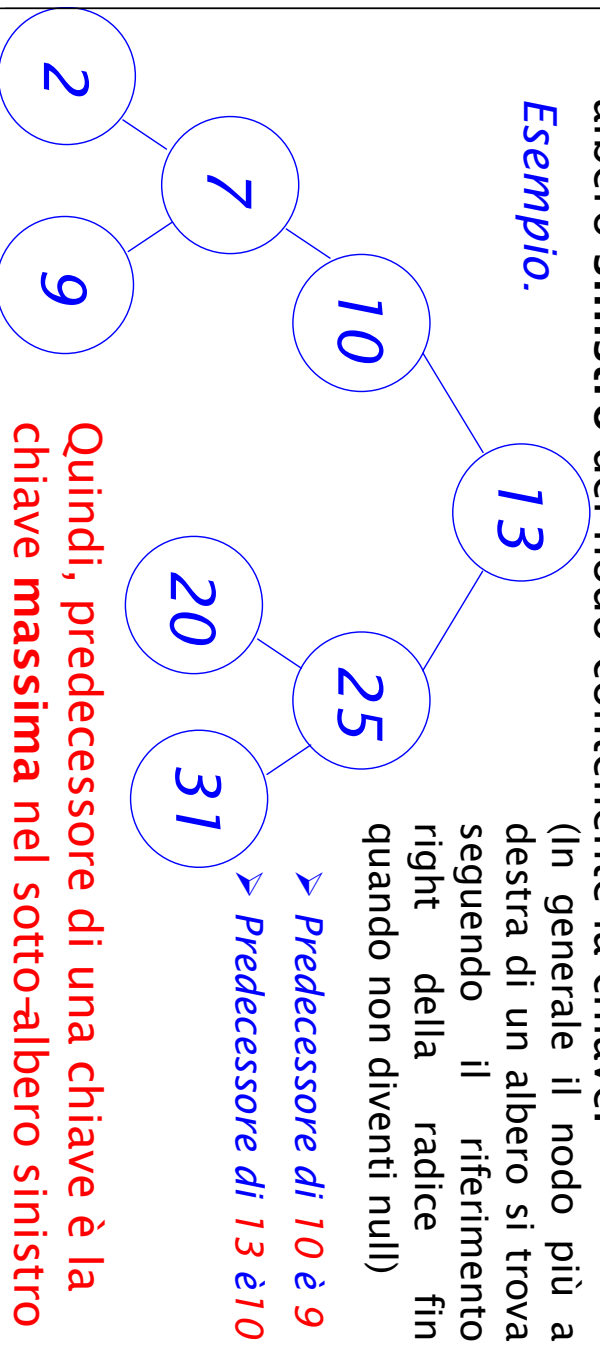
Il predecessore di b e il successore di y non sono definiti.

Si ricordi che un BST è etichettato con elementi di un insieme totalmente ordinato (altrimenti non avrebbero senso le relazioni di minore e maggiore).

Predecessore di una chiave in un BST

trascuriamo gli altri casi
Dato un BST e una sua chiave che si trova in un nodo avente sotto-albero sinistro non vuoto, il **predecessore** della chiave (nell'insieme di etichette del BST) si trova nel nodo più a **destra** del sotto-albero **sinistro** del nodo contenente la chiave.

Esempio.

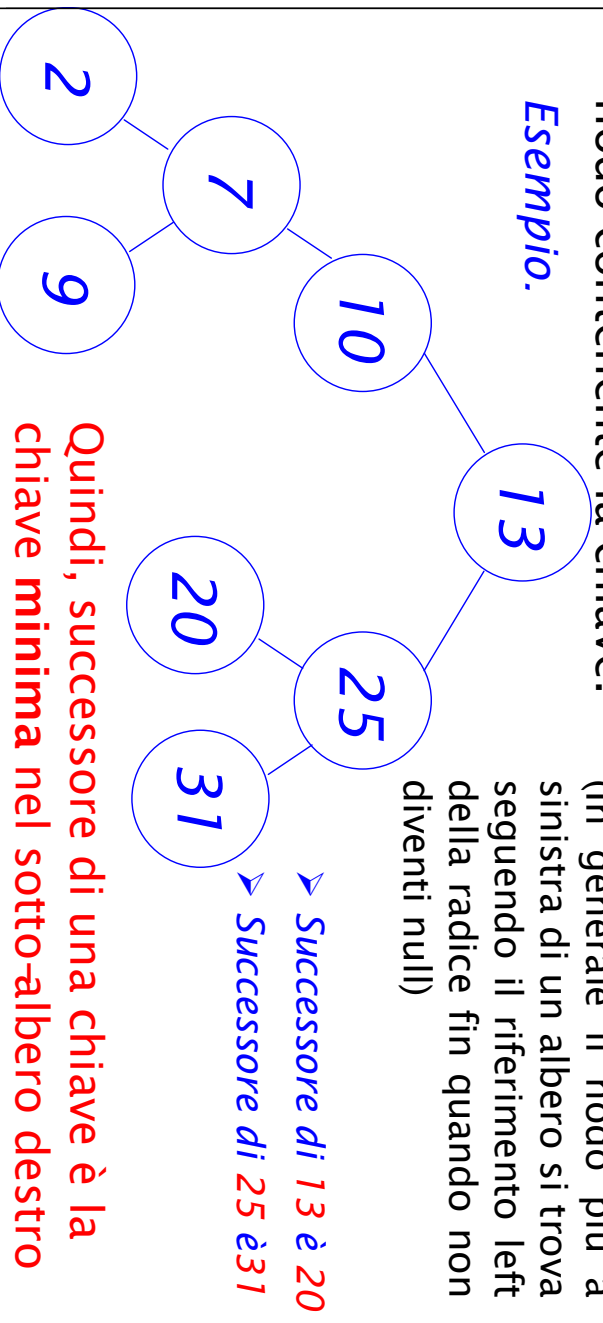


Quindi, predecessore di una chiave è la **chiave massima** nel sotto-albero sinistro

Successore di una chiave in un BST

trascuriamo gli altri casi
Dato un BST e una sua chiave che si trova in un nodo avente sotto-albero destro non vuoto, il **successore** della chiave (nell'insieme di etichette del BST) si trova nel nodo più a **sinistra** del sotto-albero **destro** del nodo contenente la chiave.

Esempio.



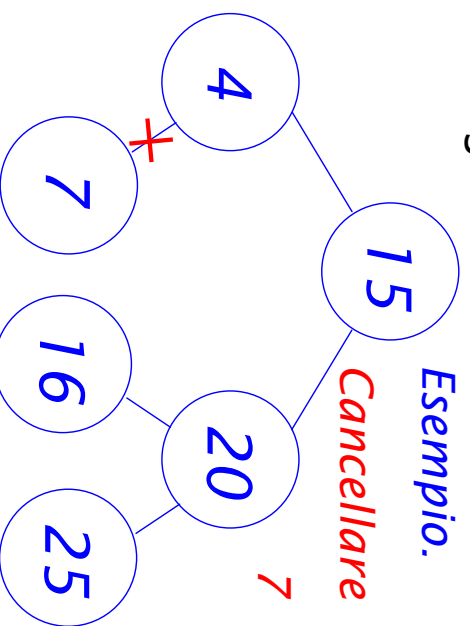
Quindi, successore di una chiave è la **chiave minima** nel sotto-albero destro

Cancellazione da un BST

Cancellare da un BST (un nuovo nodo contenente) una certa chiave, mantenendo le proprietà di BST. Algoritmo:

0. Si **ricerchi** la chiave. Se **non** si trova, si restituisca false

1. Se la chiave si trova in una **foglia**, si metta a **null** il riferimento del nodo padre alla foglia



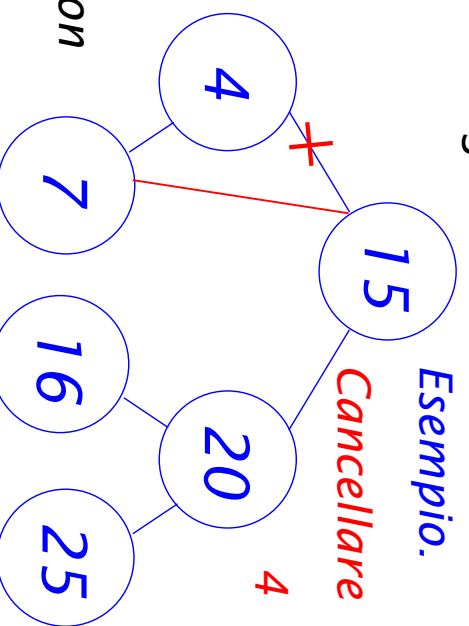
Cancellazione da un BST

Cancellare da un BST (un nuovo nodo contenente) una certa chiave, mantenendo le proprietà di BST. Algoritmo:

0. Si **ricerchi** la chiave. Se **non** si trova, si restituisca false

1. Se la chiave si trova in una **foglia**, si metta a **null** il riferimento del nodo padre alla foglia

2. Se la chiave si trova in un nodo avente **un solo sotto-albero**, si faccia puntare al figlio il riferimento del nodo padre al nodo



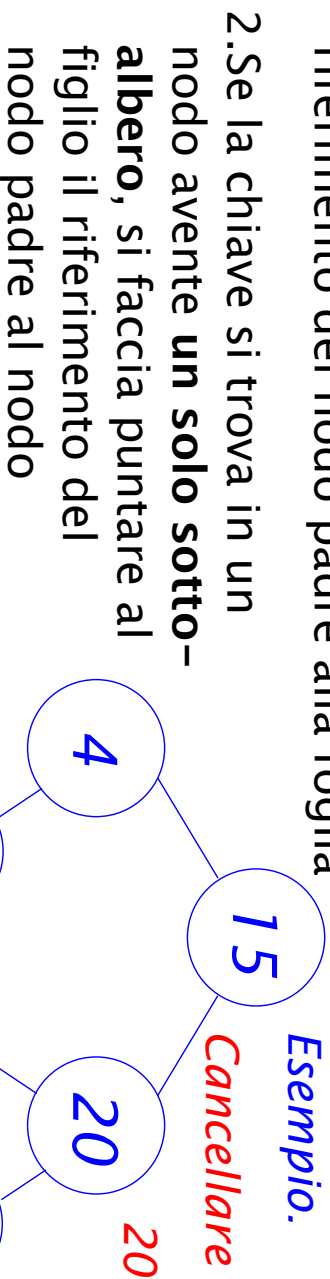
In entrambi i casi, l'altezza non aumenta.

Cancellazione da un BST

Cancellare da un BST (un nuovo nodo contenente) una certa chiave, mantenendo le proprietà di BST. Algoritmo:

0. Si **ricerchi** la chiave. Se **non** si trova, si restituisca false

1. Se la chiave si trova in una **foglia**, si metta a **null** il riferimento del nodo padre alla foglia



3. Se la chiave si trova in un nodo con **due sotto-alberi**, si esegua **fusione** dei due sotto-alberi o **sostituzione** della chiave

Implem. della cancellazione da un BST

```
public int cancel1a (int val)
{
    IntBSTNodo nodo, p = root, prev = null;
    while (p != null && p.key != val)
    {
        prev = p;
        if (val < p.key) p = p.left;
        else p = p.right;
    }
    nodo = p;
    if (p != null && p.key == val)
    {
        if (nodo.right == null) nodo = nodo.left; //passi 1 e 2
        else if (nodo.left == null) nodo = nodo.right; //del1'alg
        else fondisottoalberi //passo 3
            sostituisciChiave //in alternativa al prec1
    }
    if (p == root) root = nodo; // continua
    else if (prev.left == p) prev.left = nodo; //passi 1 e 2
    else prev.right = nodo; //del1'alg
    return 0; // cancellazione effettuata
}
...
```

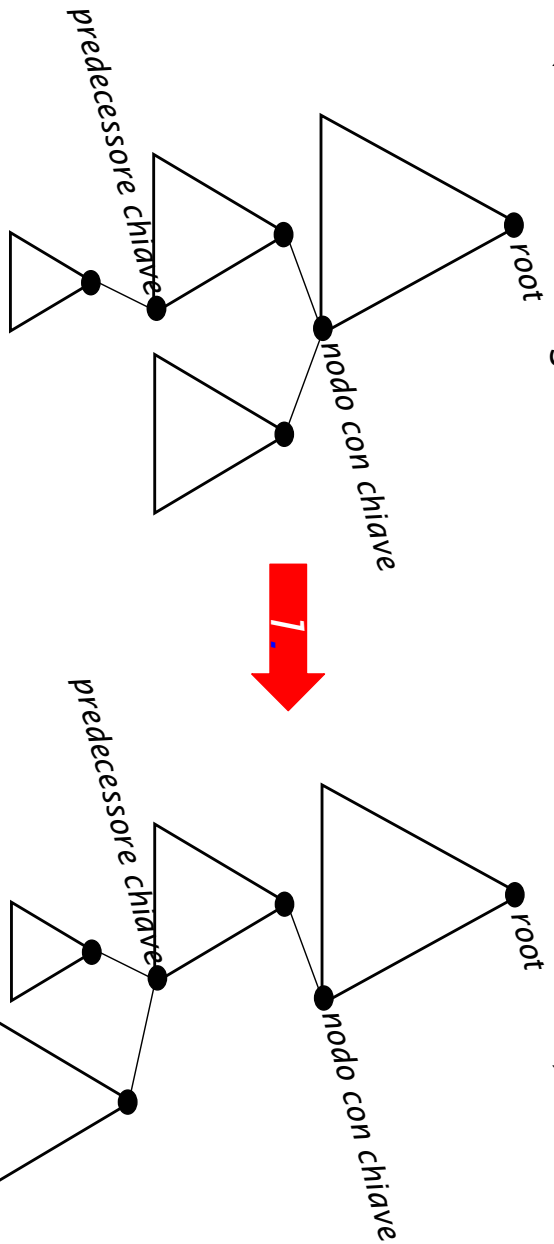
Implem. della cancellazione da un BST

```
...  
} // casi limite  
else if (root != null)  
    return -1;           // chiave non presente nel BST  
else  
    return -2;           //BST vuoto  
} // fine metodo
```

Fusione di due sotto-alberi

Serve a cancellare una chiave che si trova in un nodo avente due sotto-alberi.

1. Si innesti il sotto-albero destro del nodo contenente il **predecessore** della chiave da cancellare (il riferimento right del nodo da cancellare diventa null)

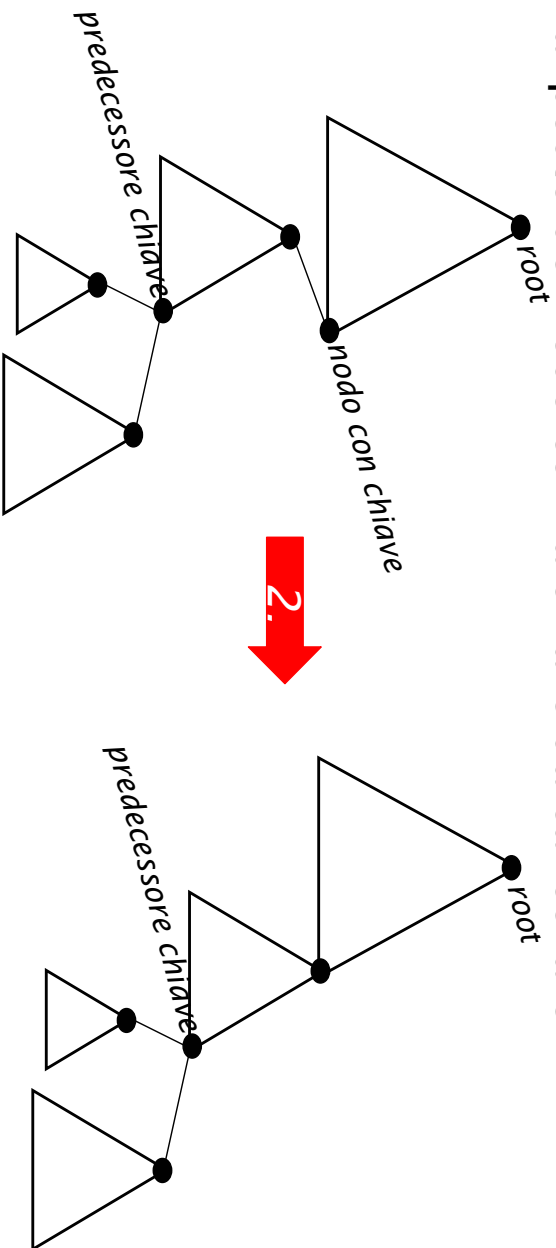


Fusione di due sotto-alberi

Serve a cancellare una chiave che si trova in un nodo avente due sotto-alberi.

1.

2. Si innesti il sotto-albero sinistro modificato come da (1) al posto del nodo con la chiave da cancellare



Implem. della fusione di due sotto-alberi

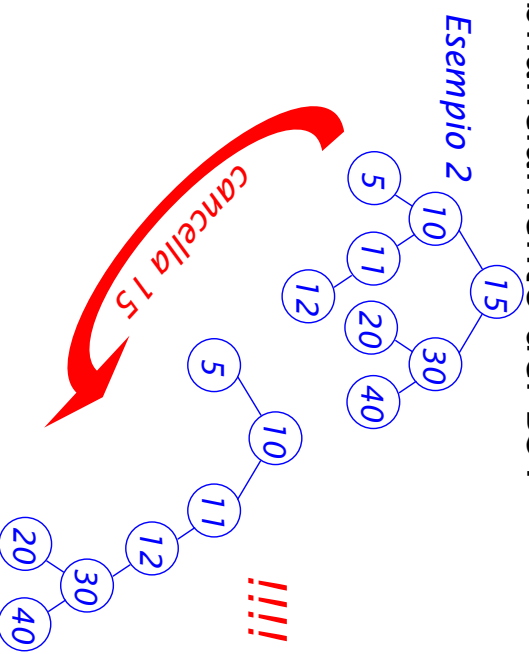
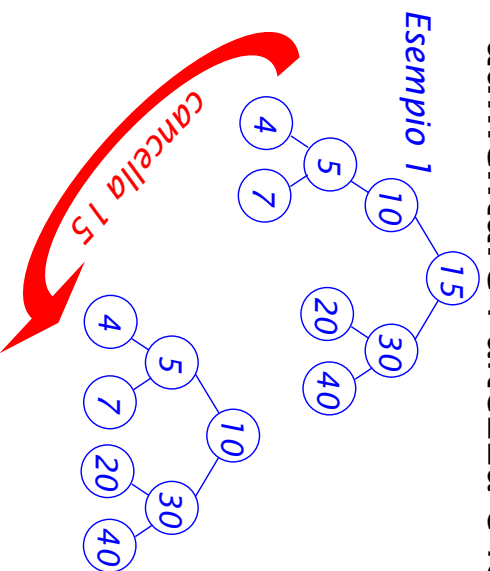
```
// fondiSottoAlberi
// da inserire nel codice della cancellazione

{ IntBSTNodo tmp = nodo.left;
  while (tmp.right != null)
    tmp = tmp.right;
  tmp.right = nodo.right;
  nodo = nodo.left;
}
```

La fusione cancella il nodo di cui fonde i sotto-alberi.

Discussione

- La cancellazione per fusione può **diminuire** o **aumentare** l'altezza e lo sbilanciamento del BST

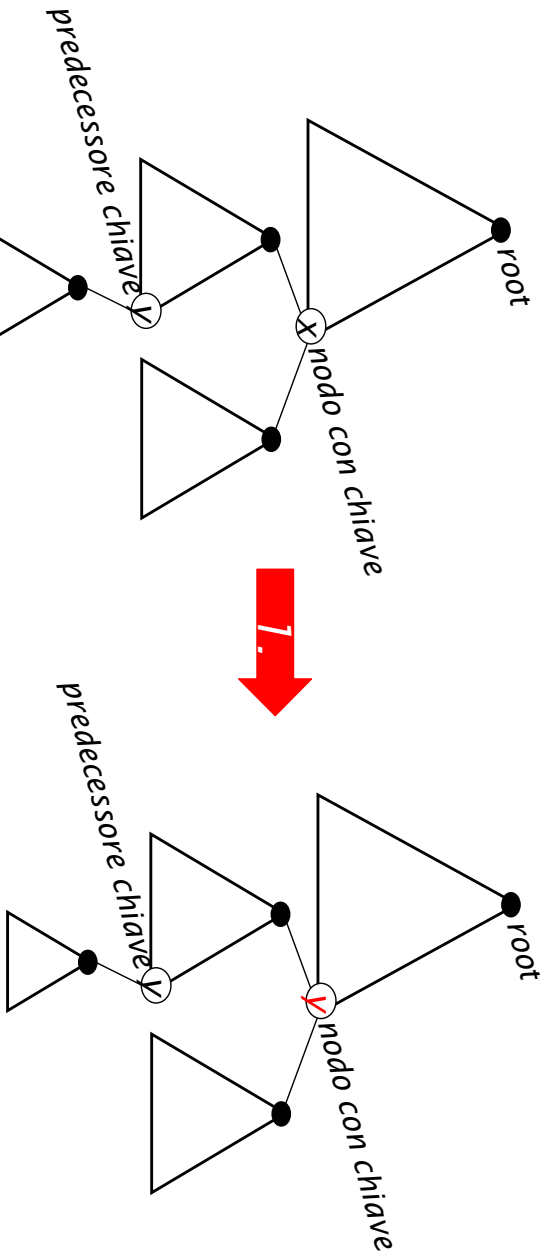


- E' possibile fare la fusione **speculare** rispetto a quella vista: 1'. Si innesti il sotto albero sinistro alla sinistra del nodo contenente il **successore** della chiave; 2'. Si innesti il sotto albero destro modificato come da (1') al posto del nodo con la chiave

Sostituzione di una chiave

Serve a cancellare una chiave che si trova in un nodo avente due sotto-alberi.

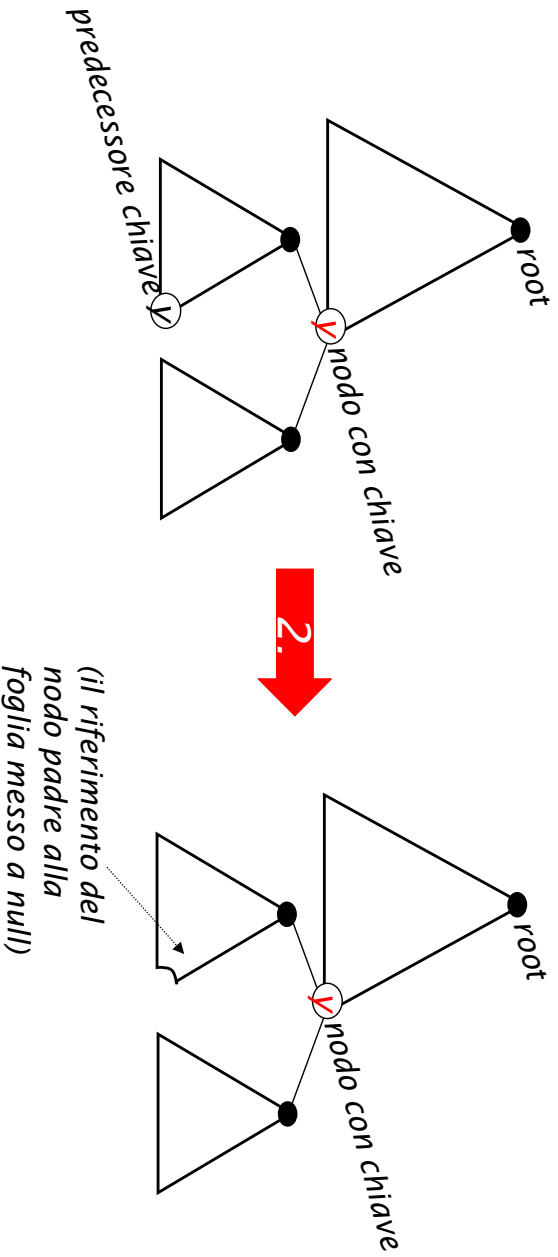
1. Si sostituisca la chiave col suo predecessore



Sostituzione di una chiave

Serve a cancellare una chiave che si trova in un nodo avente due sotto-alberi.

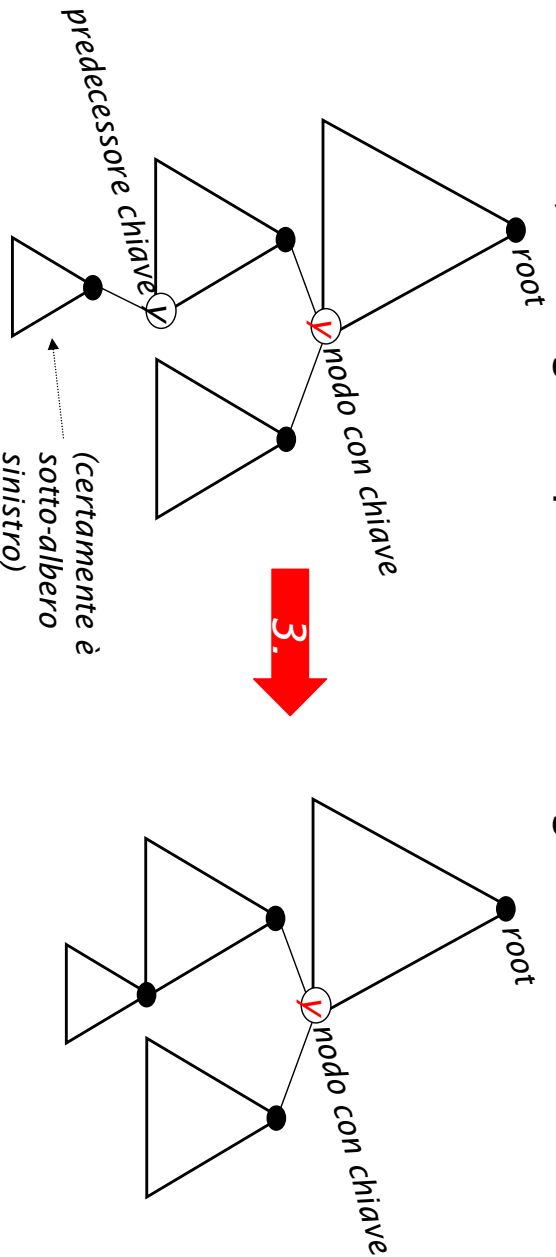
1. ...
2. Se il nodo che conteneva il predecessore è una foglia, si esegua il passo 1 dell'algoritmo di cancellazione



Sostituzione di una chiave

Serve a cancellare una chiave che si trova in un nodo avente due sotto-alberi.

1. ...
2. ...
3. Se il nodo che conteneva il predecessore ha un sotto-albero, si esegua il passo 2 dell'algoritmo di cancellaz.



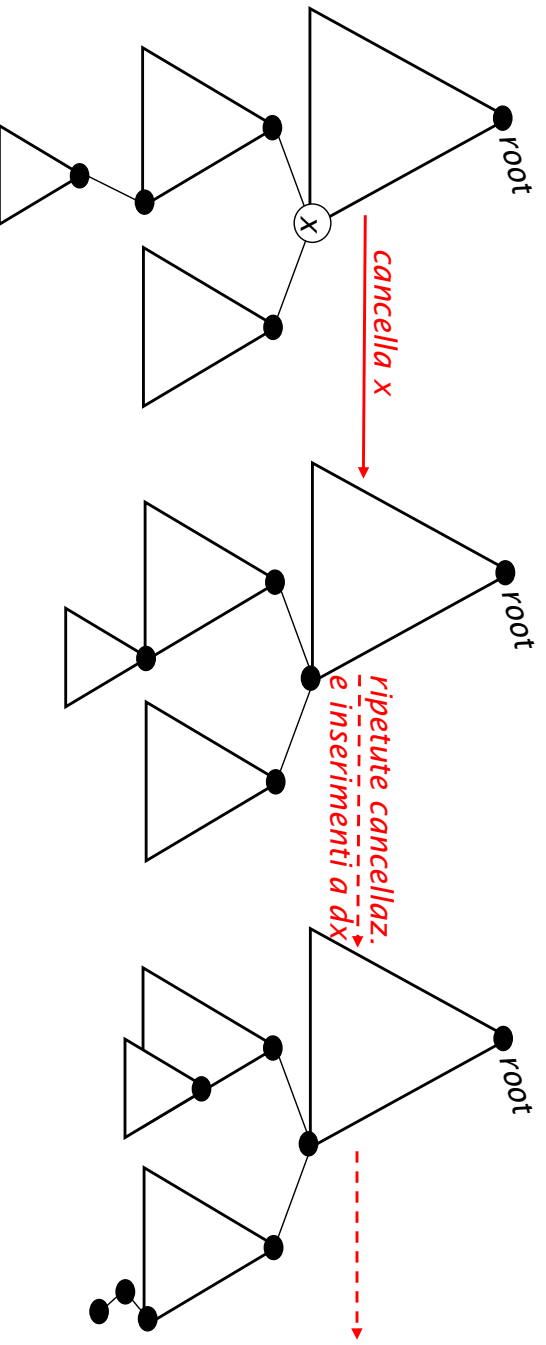
Implem. della sostituzione di una chiave

```
// sostituisciChiave
// da inserire nel codice della cancellazione
{ IntBSTNode tmp = nodo.left;
  IntBSTNode previous = nodo;
  while (tmp.right != null)
  { previous = tmp;
    tmp = tmp.right;
  }
  nodo.key = tmp.key;
  if (previous == nodo)
    previous.left = tmp.left;
  else
    previous.right = tmp.left;
}
```

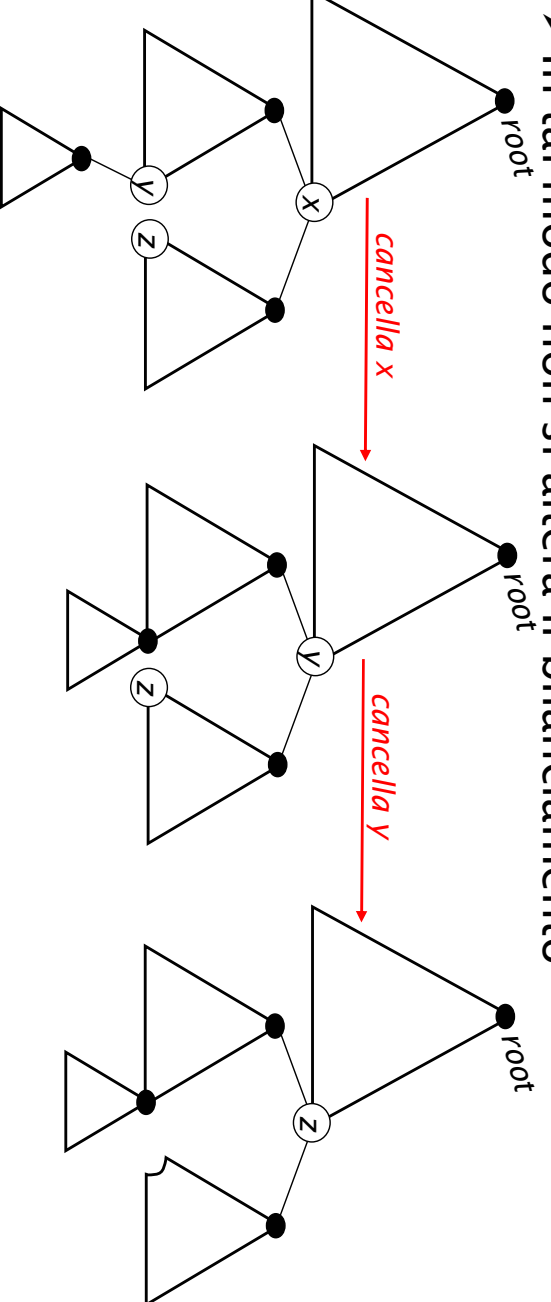
Esercizio: scrivere una versione ricorsiva del metodo

Discussione

- La cancellazione per sostituzione diminuisce l'altezza del BST
- Ma ripetute cancellazioni per sostituzione (dal sotto-albero **sinistro**) inframmezzate da inserimenti nel sotto-albero **destro** aumentano lo sbilanciamento



Discussione

- Si può ovviare al problema rendendo l'algoritmo simmetrico, ossia alternando una sostituzione col predecessore ad una sostituzione col successore
 - In tal modo non si altera il bilanciamento
- 
- In tutti i casi, la complessità asintotica della cancellazione è analoga a quella della ricerca

Ottimizzare le operazioni su BST

- Abbiamo visto che la complessità asintotica di ricerca, inserimento e cancellazione è **lineare nell'altezza del BST**
- Ne segue che $h = \Omega(\lg n)$ sempre, mentre $h = O(\lg n)$ quando il BST è **completo**. Si dimostra che vale $h = O(\lg n)$ anche quando il BST semplicemente è bilanciato
- In generale, vogliamo minimizzare l'altezza del BST. Fissato il numero di nodi, possiamo aumentare il bilanciamento saturando tutte le posizioni disponibili per i nodi
- L'algoritmo di bilanciamento – che non trattiamo – ovviamente deve mantenere le proprietà del BST!

Alberi rosso-neri

- Esistono algoritmi - che qui non trattiamo - di inserimento o cancellazione che garantiscono il bilanciamento di alberi chiamati rosso-neri
- Pertanto, ricerca, inserimento e cancellazione su alberi rosso-neri sono $O(\lg n)$