Algoritmi e Strutture Dati

Alberi di ricerca

Domenico Fabio Savo

Dizionari

Gli **alberi di ricerca** sono usati per realizzare in modo efficiente il **tipo di dato *Dizionario***

tipo `Dizionario`:
dati: un insieme S di coppie $(elem, chiave)$
operazioni:

`insert`$(elem\ e, chiave\ k)$
aggiunge a S una nuova coppia (e, k)

`delete`$(elem\ e)$
cancella da S l'elemento e

`search`$(chiave\ k) \rightarrow elem$
se la chiave k è presente in S restituisce un elemento e ad essa associato, e `null` altrimenti

Alberi binari di ricerca
(BST = binary search tree)

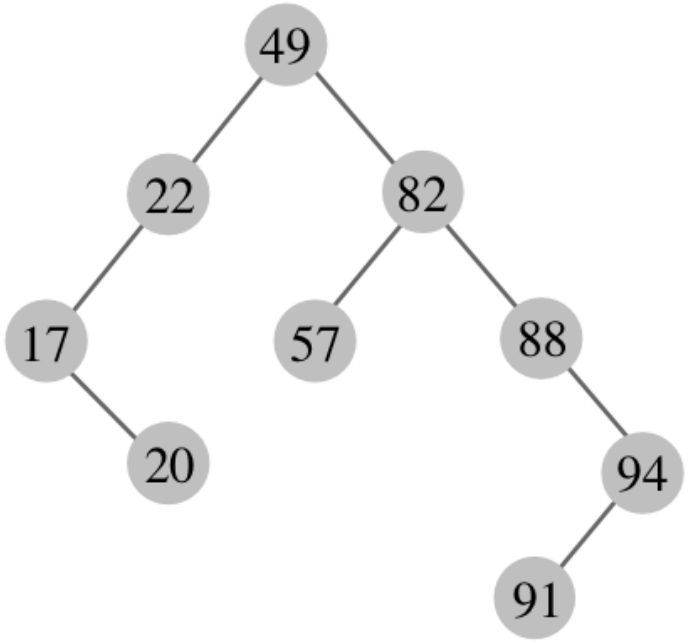
Definizione

Un BST è albero binario che soddisfa le seguenti proprietà:

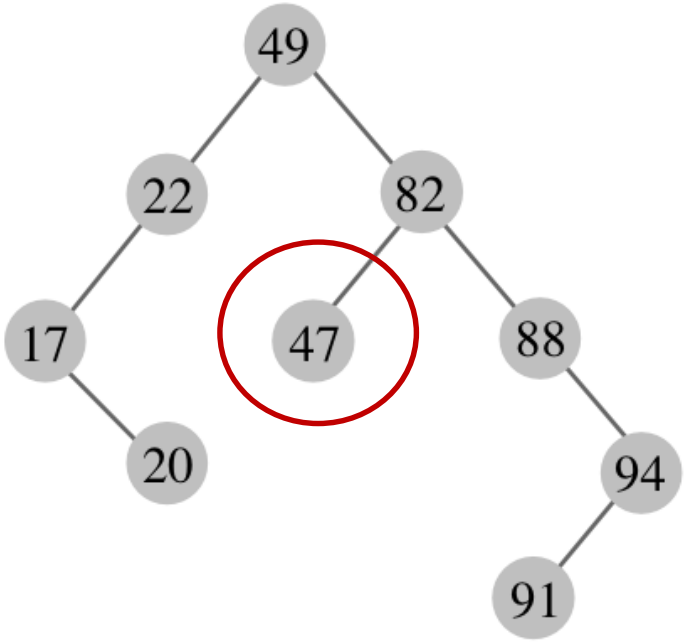
1. ogni nodo ***v*** contiene un elemento **elem(v)** cui è associata una chiave **chiave(v)** presa da un dominio totalmente ordinato
2. le chiavi nel sottoalbero sinistro di v sono $\leq$ chiave(*v*)
3. le chiavi nel sottoalbero destro di v sono $\geq$ chiave(*v*)

Le proprietà 2 e 3 sono dette "proprietà di ricerca".

Esempi



Albero binario
di ricerca



Albero binario
non di ricerca

Search

Idea: Traccia un cammino nell'albero partendo dalla radice: su ogni nodo sfruttiamo le proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro

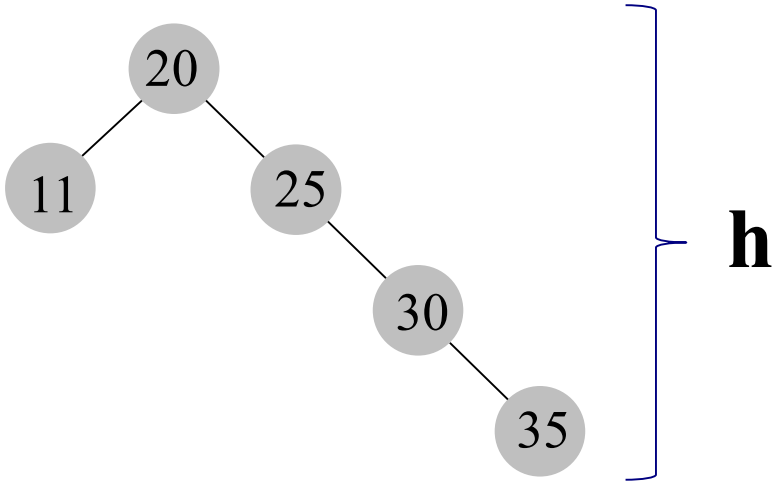
algoritmo `search`$(chiave\ k) \rightarrow elem$

1. $v \leftarrow$ radice di T
2. **while** $(\ v \neq$ `null` $)$ **do**
3. **if** $(\ k = chiave(v)\ )$ **then return** $elem(v)$
4. **else if** $(\ k < chiave(v)\ )$ **then** $v \leftarrow$ figlio sinistro di v
5. **else** $v \leftarrow$ figlio destro di v
6. **return** `null`

Search: analisi

L'approccio adottato nella funzione *search* ricorda molto da vicino quello della ricerca binaria. Il suo costo però è **O(h)**, dove **h** è l'altezza dell'albero, e non **O(log n)**.

Questa differenza è dovuta al fatto che un BST potrebbe non essere bilanciato.



Insert

Un nuovo nodo viene sempre aggiunto come foglia. L'inserimento può quindi essere implementato come segue:

1. crea un nuovo nodo **u** con **elem(u)=e** e **chiave(u)=k**
2. cerca la chiave **k** nell'albero, identificando così il nodo **v** che diventerà padre di **u**
3. appendi **u** come figlio sinistro/destro di **v** in modo che siano mantenute le proprietà di ricerca

Insert: analisi

- I passi 1. e 3. richiedono un numero di passi costante e posso quindi essere eseguiti in tempo **O(1)**.
- Il passo 2. richiede di effettuare una ricerca nell'albero che, come abbiamo visto, richiede tempo **O(h)**.

- In totale, anche il costo per l'inserimento è quindi **O(h)**

Delete

- Come avviene in molte strutture dati, cancellare è più difficile che inserire.

- Per implementare l'operazione di *delete* introducimo due nuove operazioni:

  – *max(nodo u)* → *nodo*
  – *pred(nodo u)* → *nodo*

Max

- Questa operazione permette di individuare il valore massimo in un albero radicato in **u**.
- Grazie alle proprietà di ricerca, per trovare il massimo è sufficiente, partendo da **u**, scendere verso destra nell'albero finché è possibile.
- Il tempo di esecuzione è chiaramente **O(h)**.

algoritmo `max`$(nodo\ u) \rightarrow nodo$

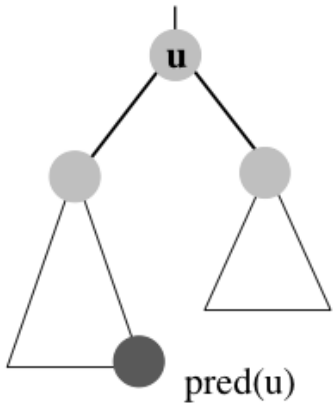
1. $v \leftarrow u$
2. **while** (figlio destro di $v \neq$ `null`) **do**
3. $v \leftarrow$ figlio destro di v
4. **return** v

Ricerca del predecessore (1/3)

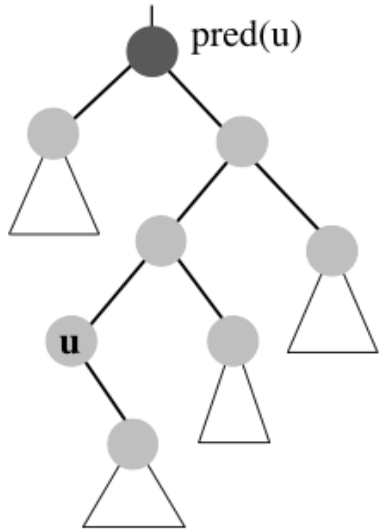
- Il predecessore di un nodo **u** è il nodo **v** dell'albero avente **massima** chiave minore o uguale alla chiave di **u**.

- Per trovare il predecessore di **u** possiamo distinguere due casi:
  1. **u** ha un figlio sinistro: in tal caso **pred(u)** è il massimo del sottoalbero sinistro di **u**.
  2. **u** non ha un figlio sinistro: **pred(u)**, se esiste, è il più basso antenato di **u** (ovvero, l'antenato di **u** con massima profondità nell'albero) il cui figlio destro è anch'esso antenato di **u**. Per trovarlo, risaliamo da **u** verso la radice fino ad incontrare la prima "svolta a sinistra".

Ricerca del predecessore (2/3)



Caso 1: u ha un figlio sinistro



Caso 2: u non ha un figlio sinistro

Ricerca del predecessore (3/3)

```
algoritmo pred(nodo u) → nodo
1.    if ( u ha figlio sinistro sin(u) ) then
2.        return max(sin(u))
3.    while ( parent(u) ≠ null e u è figlio sinistro di suo padre ) do
4.        u ← parent(u)
5.    return parent(u)
```

Anche la ricerca del predecessore di un nodo **u** ha costo **O(h)**, dove **h** è l'altezza dell'albero

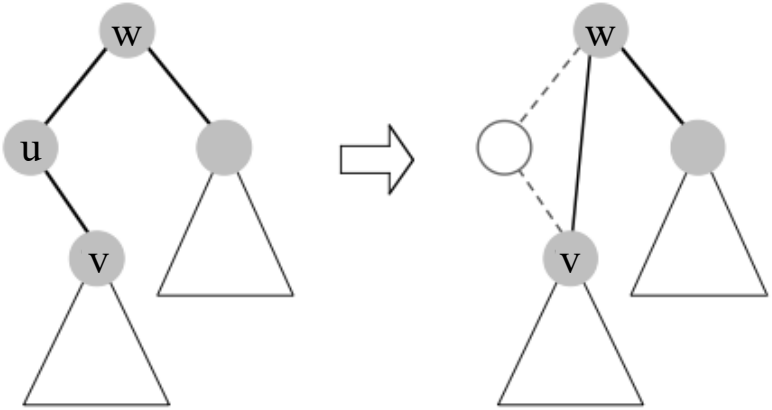
Delete – caso 1

Sia **u** il nodo contenente l'elemento **e** da cancellare. Distinguiamo tre casi:

Caso 1: **u** è una foglia: in tal caso è sufficiente eliminarla

Delete – caso 2

Caso 2: **u** ha un unico figlio: sia **v** l'unico figlio di **u**
Se **u** è la radice dell'albero, **v** diviene la nuova radice. Altrimenti, dopo aver individuato il genitore **w** di **u,** l'arco **(w, u)**, viene sostituito dall'arco **(w, v)**

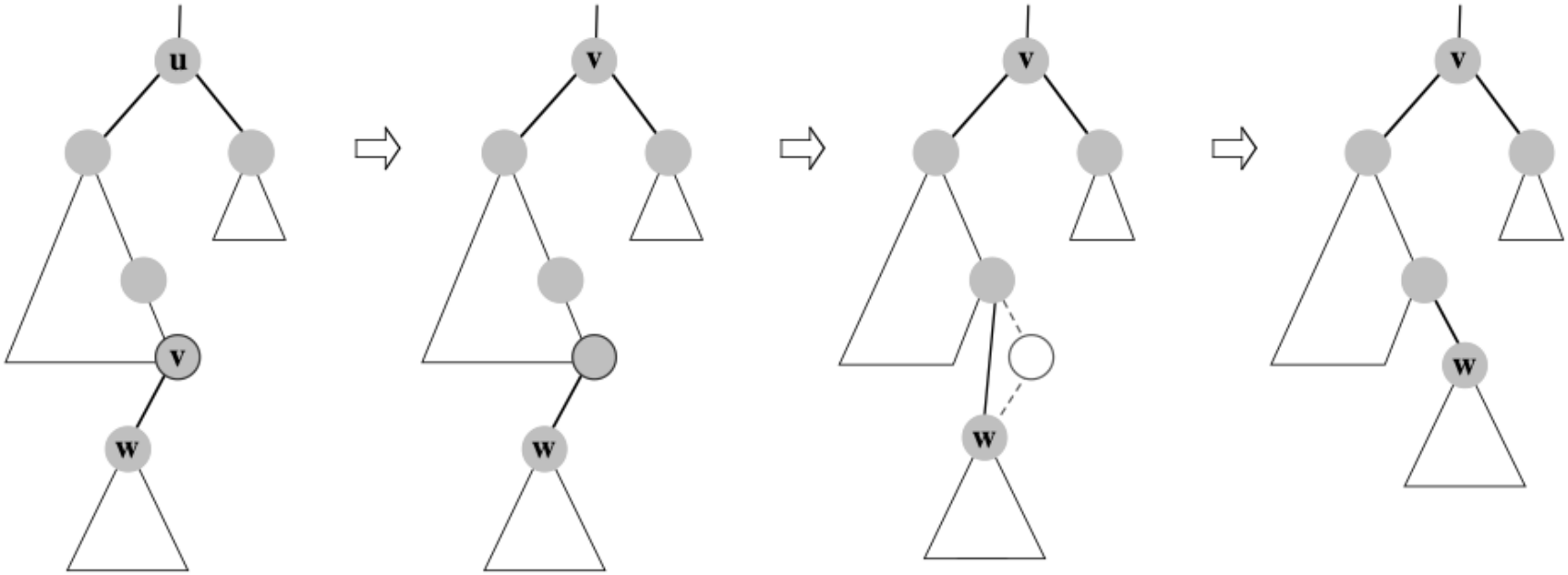


Delete – caso 3

Caso 3: **u** ha due figli: ci riconduciamo ad uno dei casi precedenti nel seguente modo. Individuiamo il *predecessore* **v** di **u**. Tale predecessore è sicuramente il massimo del sottoalbero sinistro di **u** poiché **u** ha due figli. Copiamo **v** in **u** ed eliminiamo il vecchio nodo **v** applicando uno dei casi precedenti.

Delete – caso 3

Caso 3: **u** ha due figli: lo sostituisco con il predecessore (v) e rimuovo fisicamente il predecessore (che ha un solo figlio)



Delete: analisi

- Si può verificare che le proprietà di ricerca sono mantenute.

- La cancellazione di un nodo interno richiede l'individuazione del nodo da cancellare nonché del suo predecessore (caso 3). Quindi il costo della cancellazione è **O(h)**

BST – riassumendo

- Tutte le operazioni hanno costo **O(h)** dove **h** è l'altezza dell'albero
- **O(n)** nel caso peggiore (alberi molto sbilanciati e profondi)

Alberi AVL
(Adel'son-Vel'skii e Landis)

Definizioni

Fattore di bilanciamento di un nodo v =

| altezza del sottoalbero sinistro di v –
altezza del sottoalbero destro di v |

Un albero si dice **bilanciato in altezza** se ogni nodo **v** ha fattore di bilanciamento **≤ 1**

Alberi AVL = alberi binari di ricerca bilanciati in altezza

Altezza di alberi AVL

Si può dimostrare che un albero AVL con **n** nodi ha altezza **O(log n)**

Segue che, utilizzando l'algoritmo *search* introdotto per i BST, possiamo implementare la ricerca in un AVL con costo **O(log n)**

Implementazione delle operazioni

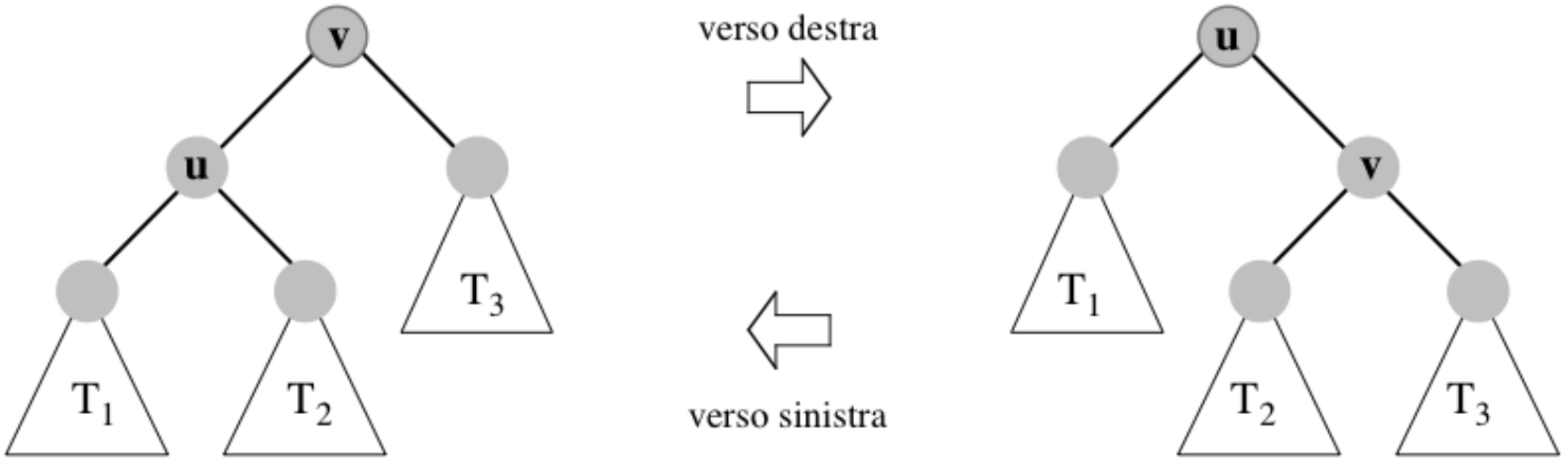
- L'operazione *search* procede come in un BST

- Ma inserimenti e cancellazioni potrebbero sbilanciare l'albero

➡ Dobbiamo mantenere il bilanciamento (utilizziamo le **rotazioni…**)

La rotazione di base

- Nella *rotazione di base* (anche detta *rotazione semplice*), un nodo **perno** viene fatto ruotare verso destra o verso sinistra (i due casi sono simmetrici).
- La rotazione mantiene la proprietà di ricerca
- Richiede tempo **O(1)**



Ribilanciamento tramite rotazioni

- Le rotazioni sono effettuate su nodi sbilanciati
- Sia **v** un nodo con fattore di bilanciamento ≥ 2
- Esiste almeno un sottoalbero **T** di **v** che lo sbilancia
- A seconda della posizione di **T** si hanno 4 casi:

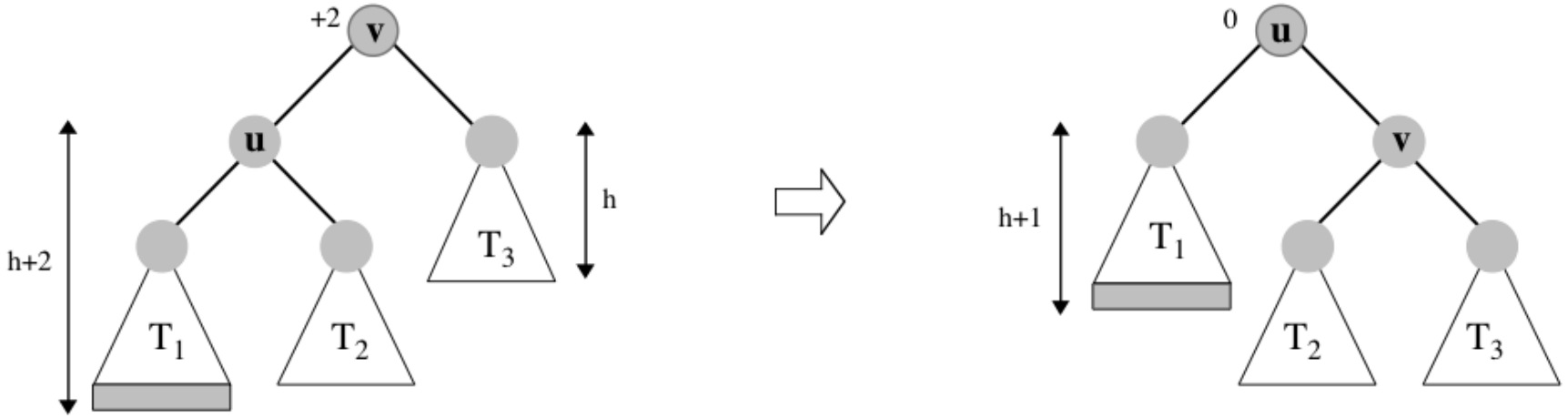
Se esiste un solo sottoalbero **T** che causa lo sbilanciamento:

Sinistra - sinistra	(SS)	T è il sottoalbero sinistro del figlio sinistro di v
Destra - destra	(DD)	T è il sottoalbero destro del figlio destro di v
Sinistra - destra	(SD)	T è il sottoalbero destro del figlio sinistro di v
Destra - sinistra	(DS)	T è il sottoalbero sinistro del figlio destro di v

- I quattro casi sono simmetrici a coppie (SS con DD e SD con DS).

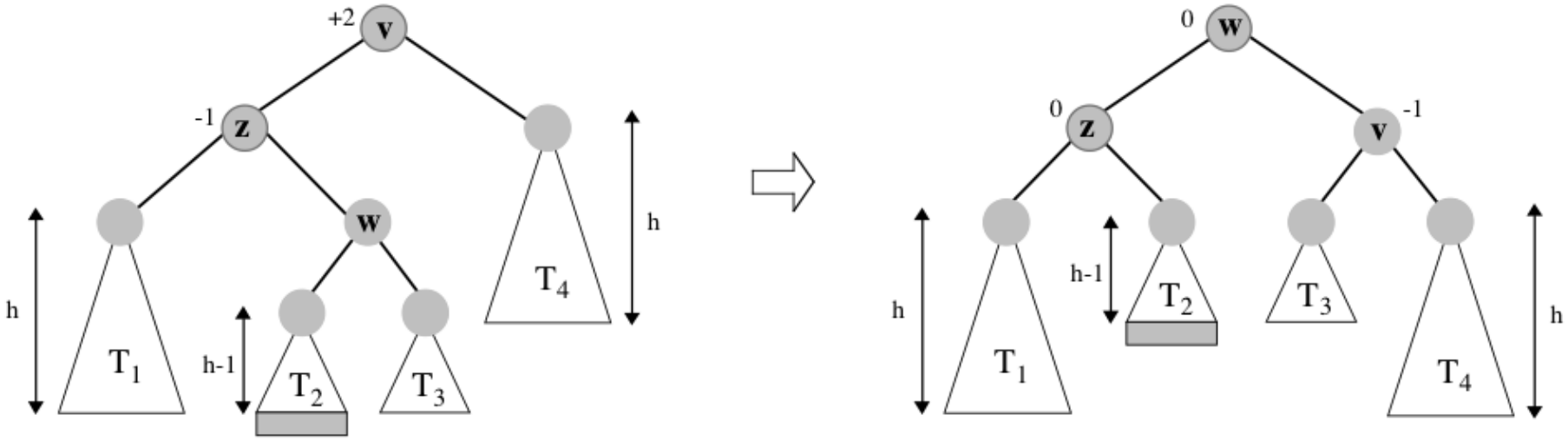
Rotazione SS

- Applicare una *rotazione semplice* verso **destra** su **v**
- L'altezza dell'albero coinvolto nella rotazione passa da **h+3** a **h+2**



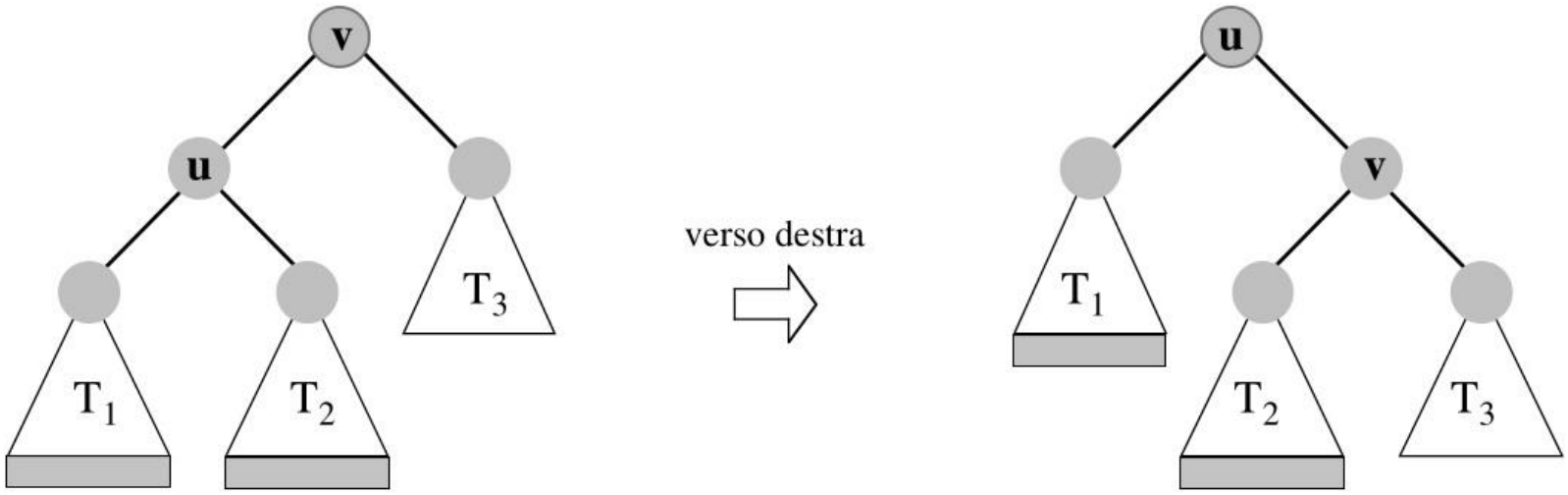
Rotazione SD

- Richiede di applicare due rotazioni. Sia **z** il figlio sinistro di **v**: l'albero che sbilancia **v** è il sottoalbero destro di **z**. Effettuiamo una prima rotazione semplice di **z** verso sinistra, seguita da una rotazione semplice di **v** verso destra.



Più sottoalberi che sbilanciano

- Nel caso in cui il sottoalbero che crea lo sbilanciamento non è unico (T1 e T2 nella figura) è sufficiente effettuare una rotazione nel verso opportuno.



insert(elem e, chiave k)

1. Crea un nuovo nodo **u** con:

   elem(u) = e

   chiave(u)= k
2. Inserisci **u** come in un BST (quindi come foglia)
3. Ricalcola i fattori di bilanciamento dei nodi nel cammino dalla radice a **u**: sia **v** il più profondo nodo con fattore di bilanciamento pari a ±2 (nodo critico)
4. Esegui una rotazione opportuna su **v**

Osservazione: una sola rotazione è sufficiente, poiché l'altezza dell'albero coinvolto diminuisce di 1

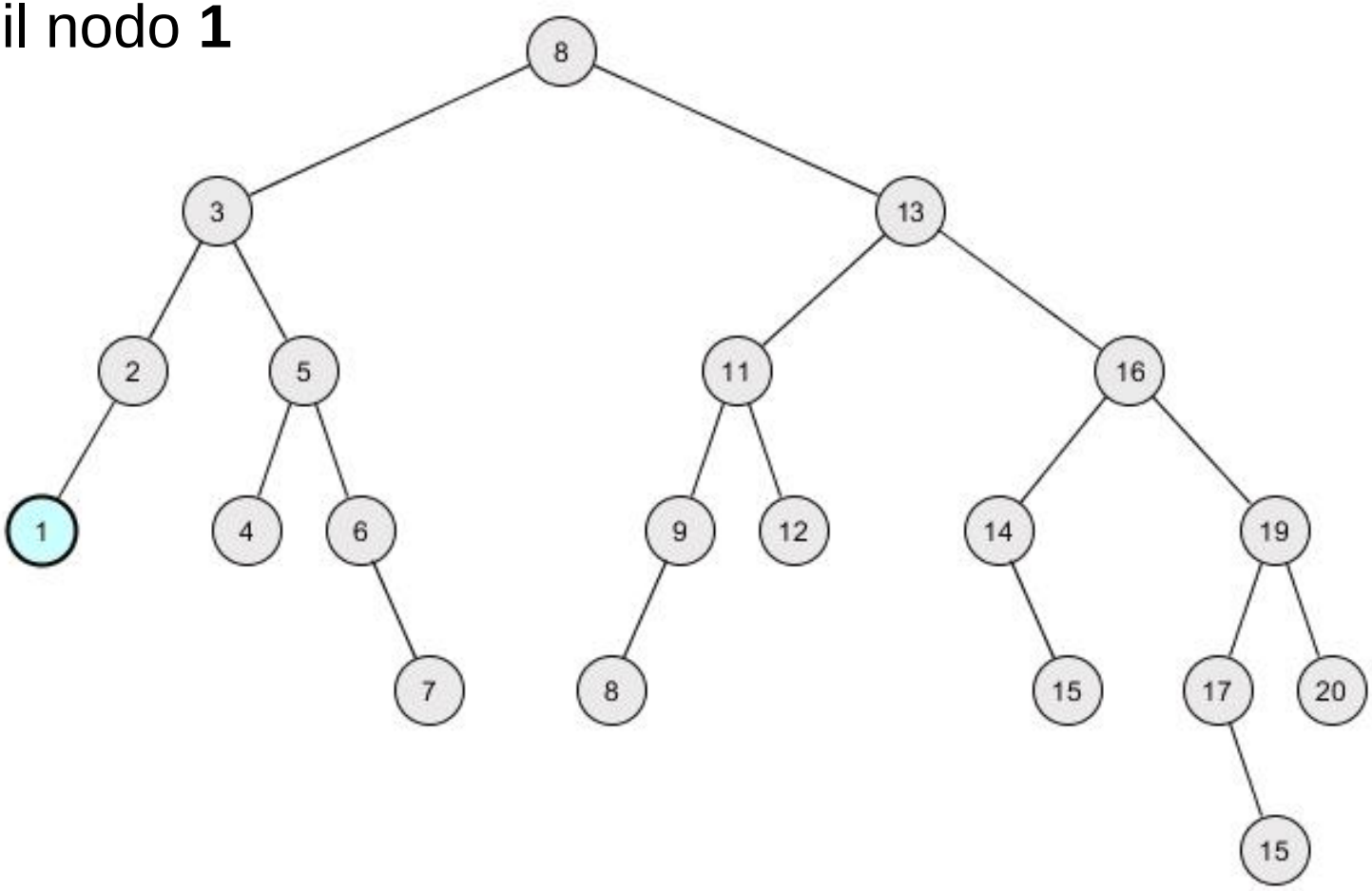
delete(elem e)

1. Cancella il nodo come in un BST
2. Ricalcola i fattori di bilanciamento dei nodi nel cammino dalla radice al padre del nodo eliminato fisicamente
3. Ripercorrendo il cammino dal basso verso l'alto, esegui l'opportuna rotazione semplice o doppia sui nodi sbilanciati

Osservazione: potrebbero essere necessarie **O(log n)** rotazioni

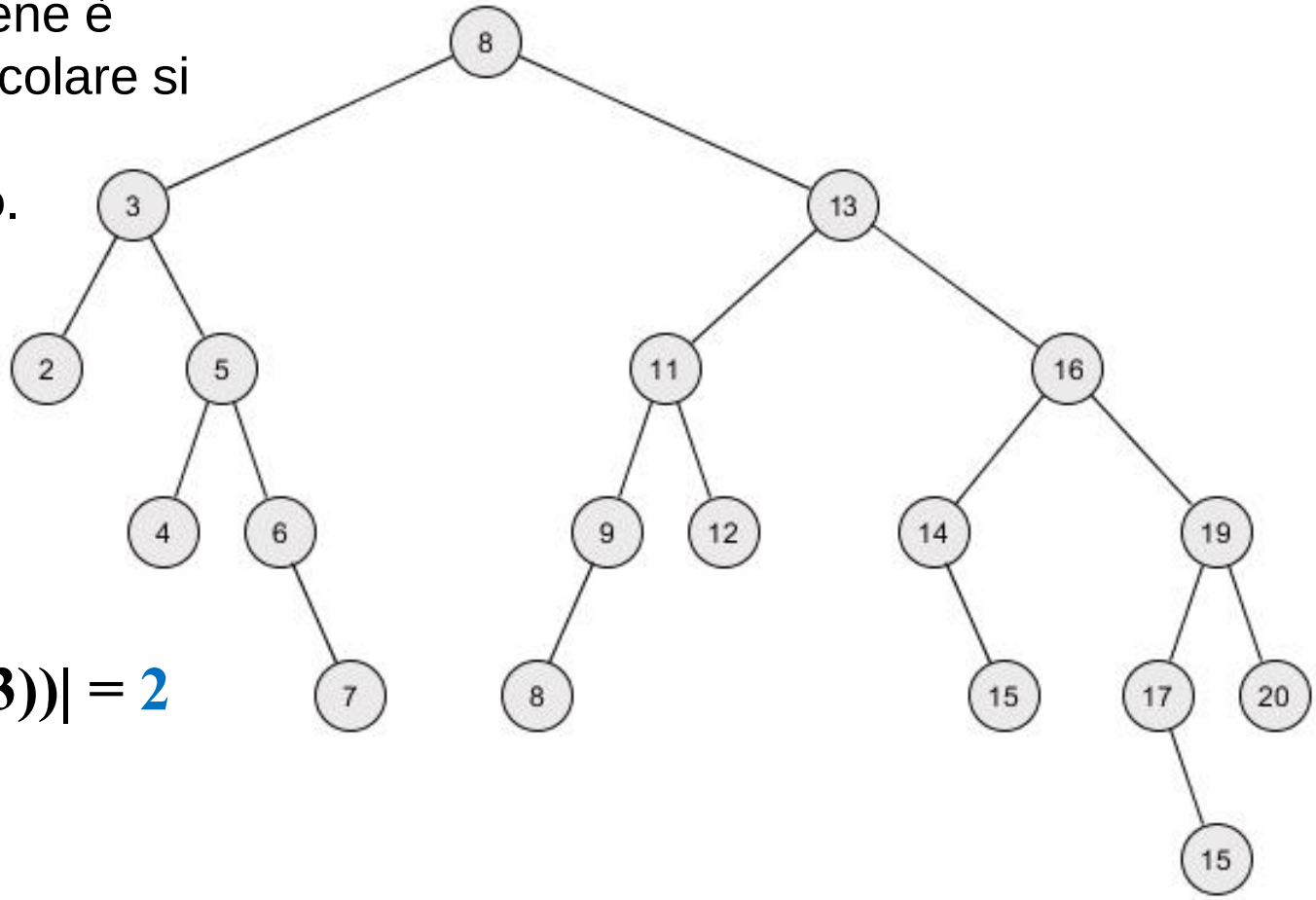
Cancellazione – esempio (1/7)

Supponiamo di voler cancellare il nodo **1**



Cancellazione – esempio (2/7)

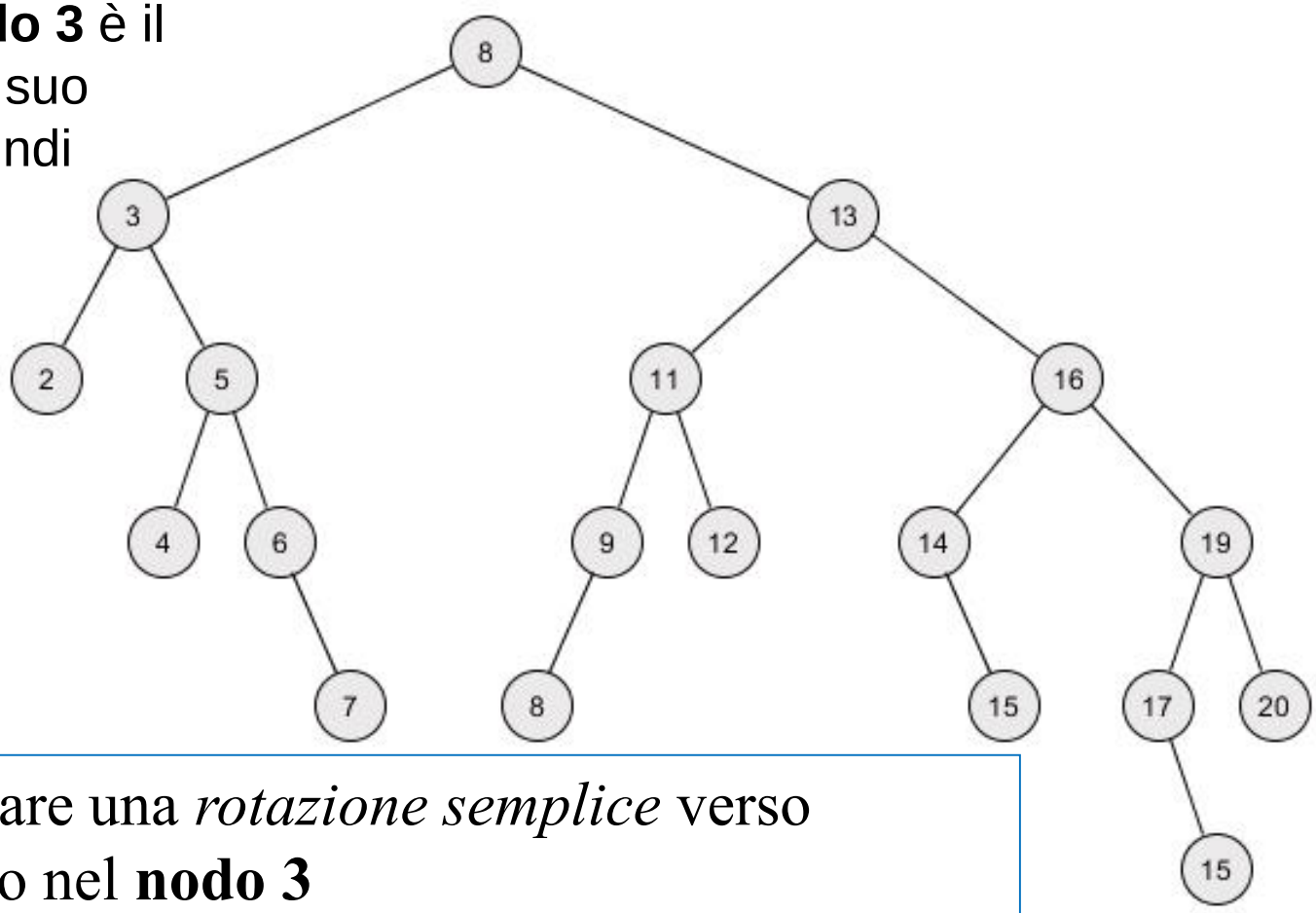
L'albero che si ottiene è sbilanciato, in particolare si individua il **nodo 3** come **nodo critico**.



$$|h(sx(3)) - h(dx(3))| = 2$$

Cancellazione – esempio (3/7)

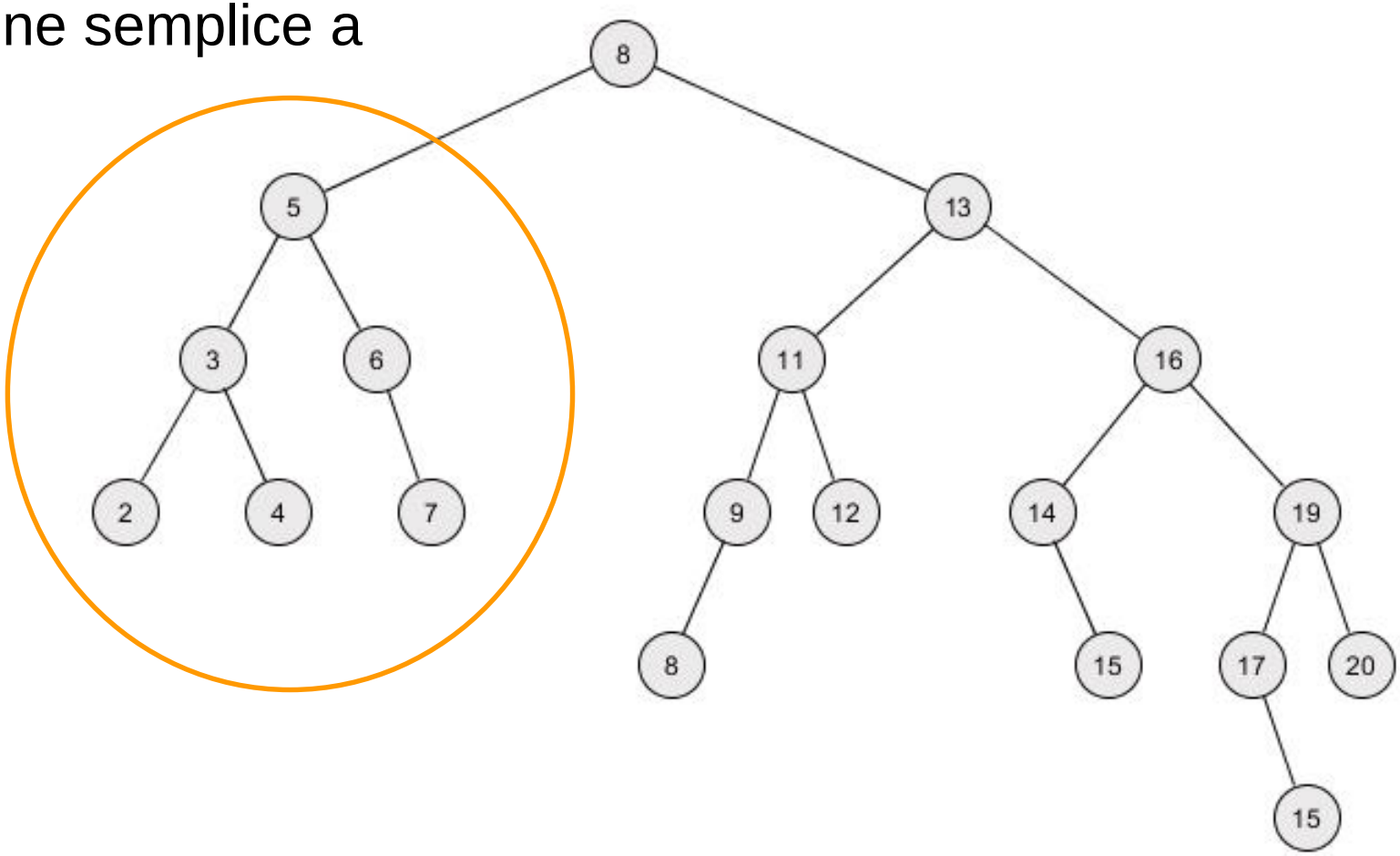
A sbilanciare il **nodo 3** è il sotto albero dx del suo figlio dx. Siamo quindi nel caso **DD**



Dobbiamo applicare una *rotazione semplice* verso **sinistra** con perno nel **nodo 3**

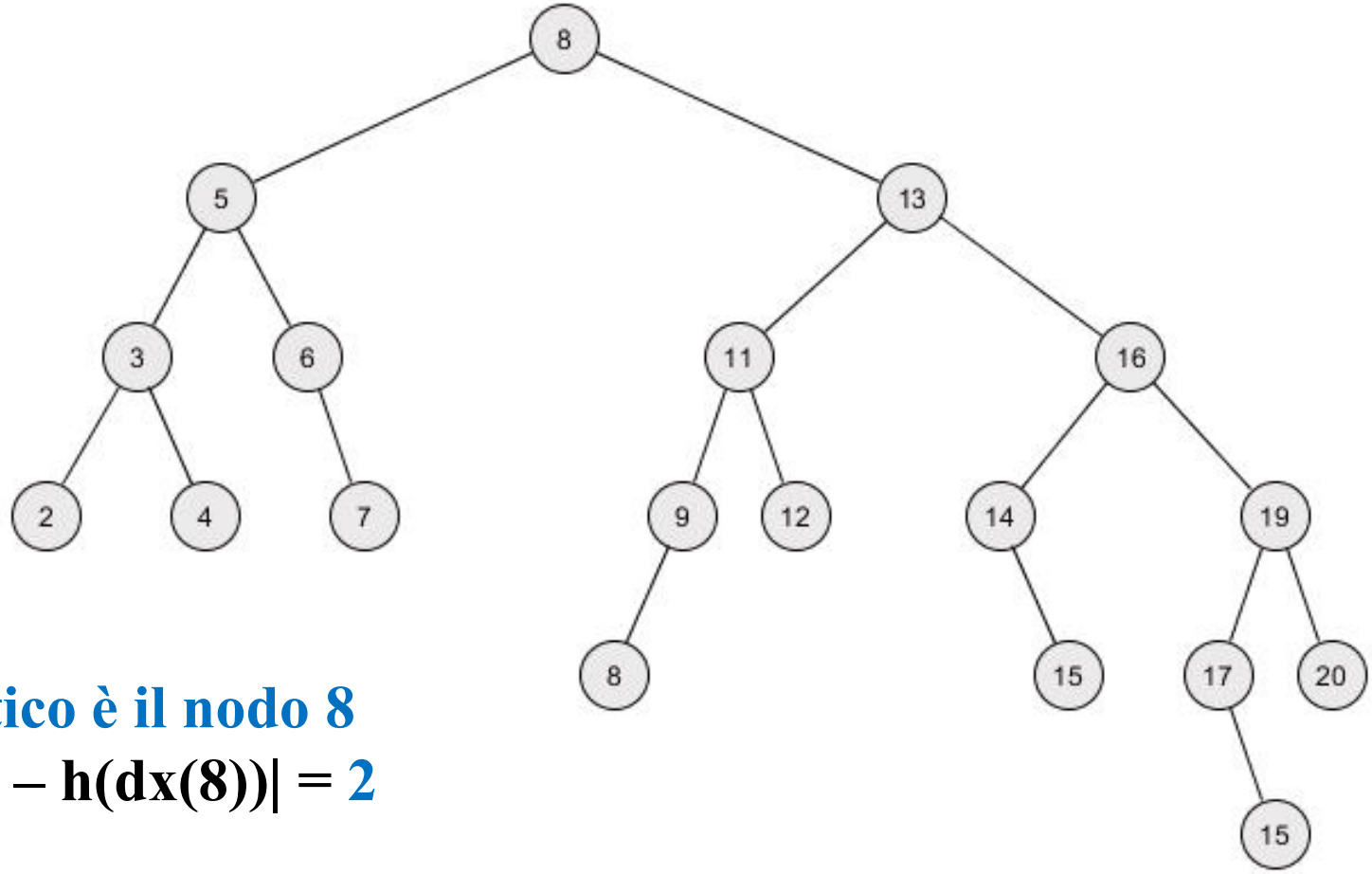
Cancellazione – esempio (4/7)

Rotazione semplice a sinistra



Cancellazione – esempio (5/7)

L'albero è ancora sbilanciato!

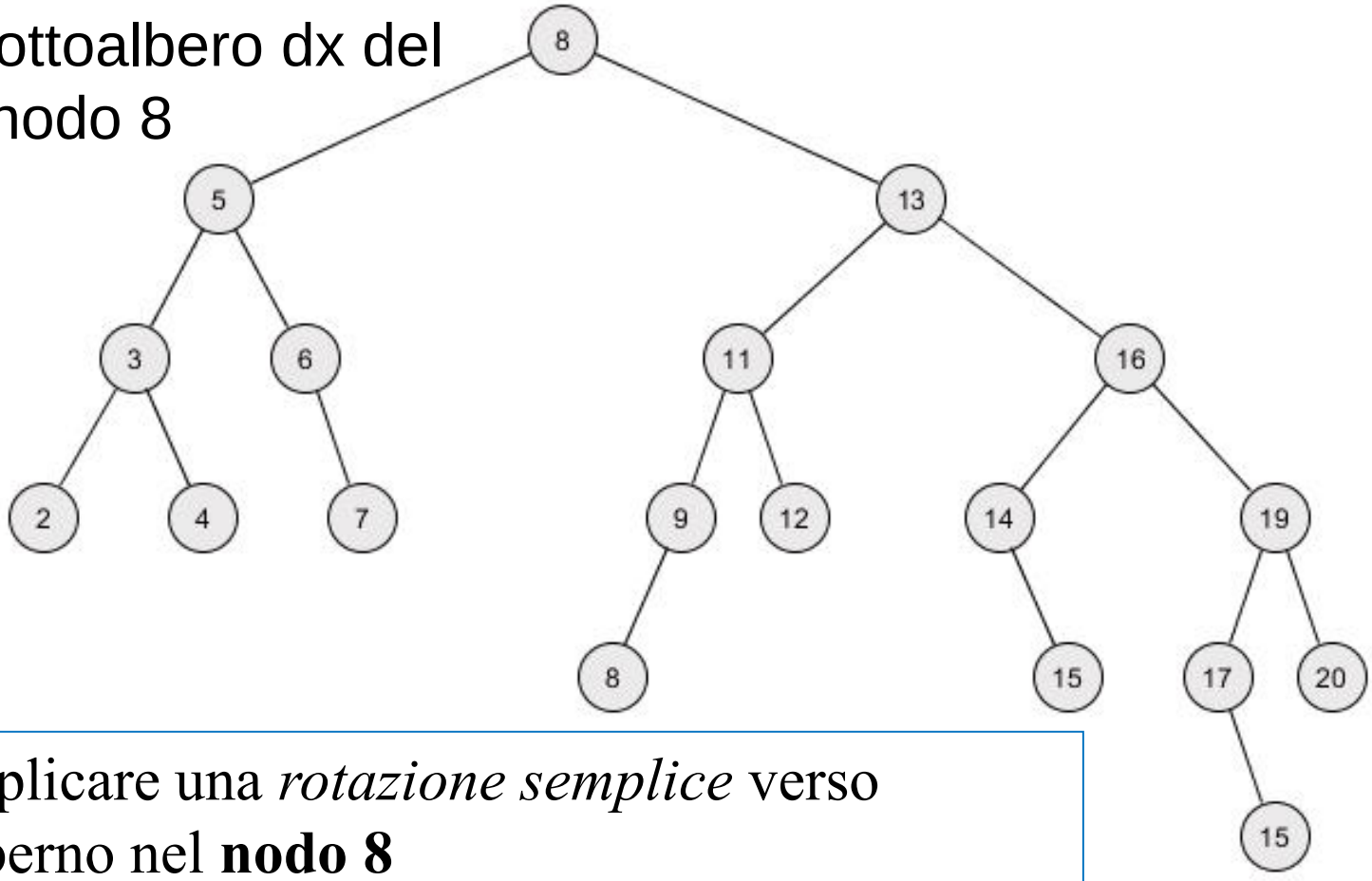


Il nodo critico è il nodo 8

→ |h(sx(8)) – h(dx(8))| = 2

Cancellazione – esempio (6/7)

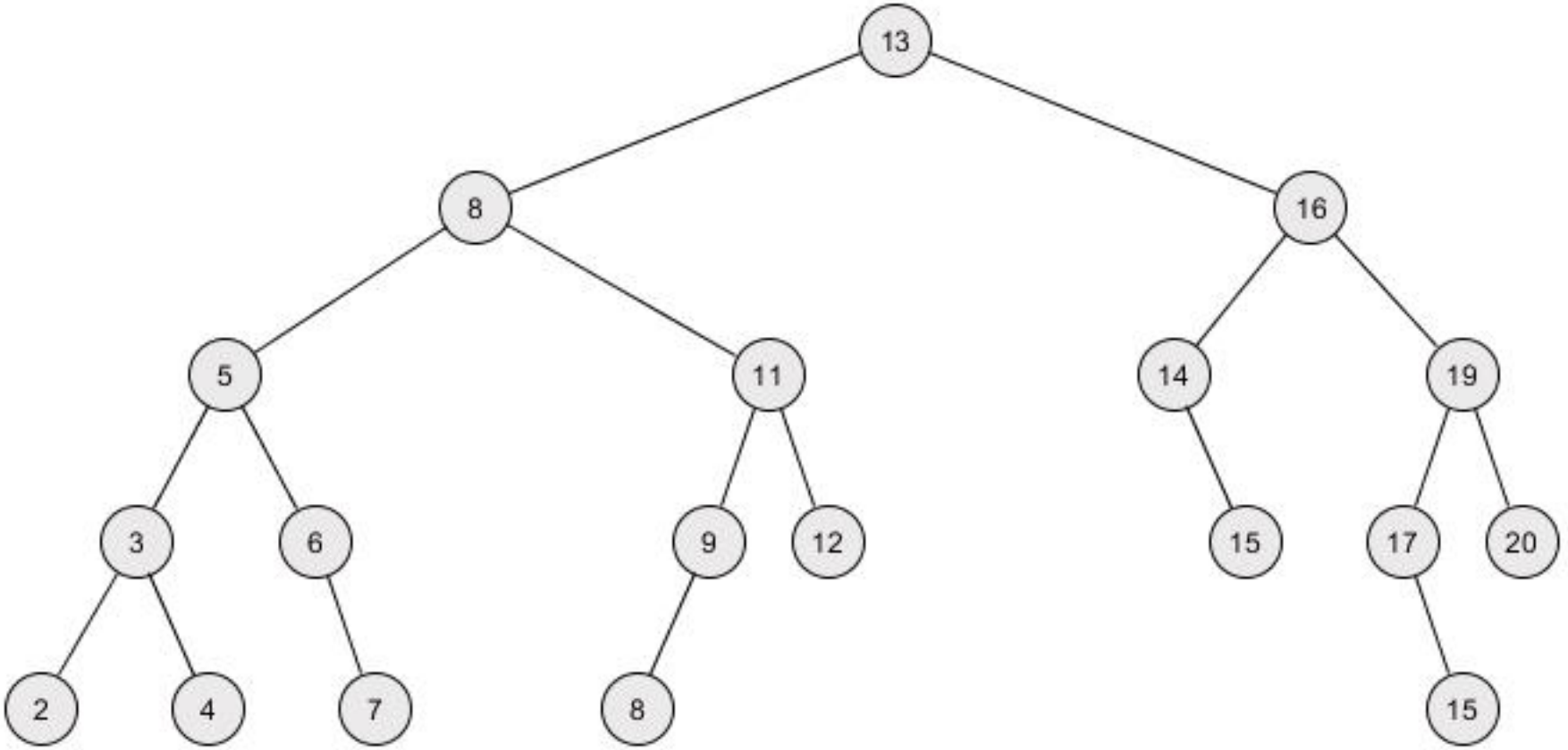
Siamo ancora nel caso **DD:** a sbilanciare l'albero radicato nel nodo 8 è il sottoalbero dx del figlio dx del nodo 8



Dobbiamo applicare una *rotazione semplice* verso **sinistra** con perno nel **nodo 8**

Cancellazione – esempio (8/8)

Dopo la *rotazione semplice* verso **sinistra** con perno nel **nodo 8** otteniamo finalmente un albero bilanciato



Classe AlberoAVL

classe `AlberoAVL` **estende** `AlberoBinarioDiRicerca`:

dati: $S(n) = O(n)$

albero binario di ricerca T ereditato, più il fattore di bilanciamento di ogni nodo.

operazioni:

`search` $(chiave\ k) \rightarrow elem$ $T(n) = O(\log n)$

ereditata.

`insert`$(elem\ e, chiave\ k)$ $T(n) = O(\log n)$

chiama `insert`() ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(1)$ rotazioni.

`delete`$(elem\ e)$ $T(n) = O(\log n)$

chiama `delete`() ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(\log n)$ rotazioni.

Costo delle operazioni

- Tutte le operazioni hanno costo **O(log n)** poiché l'altezza dell'albero è **O(log n)** e ciascuna rotazione richiede solo tempo costante