Algoritmi e Strutture Dati

Alberi di ricerca

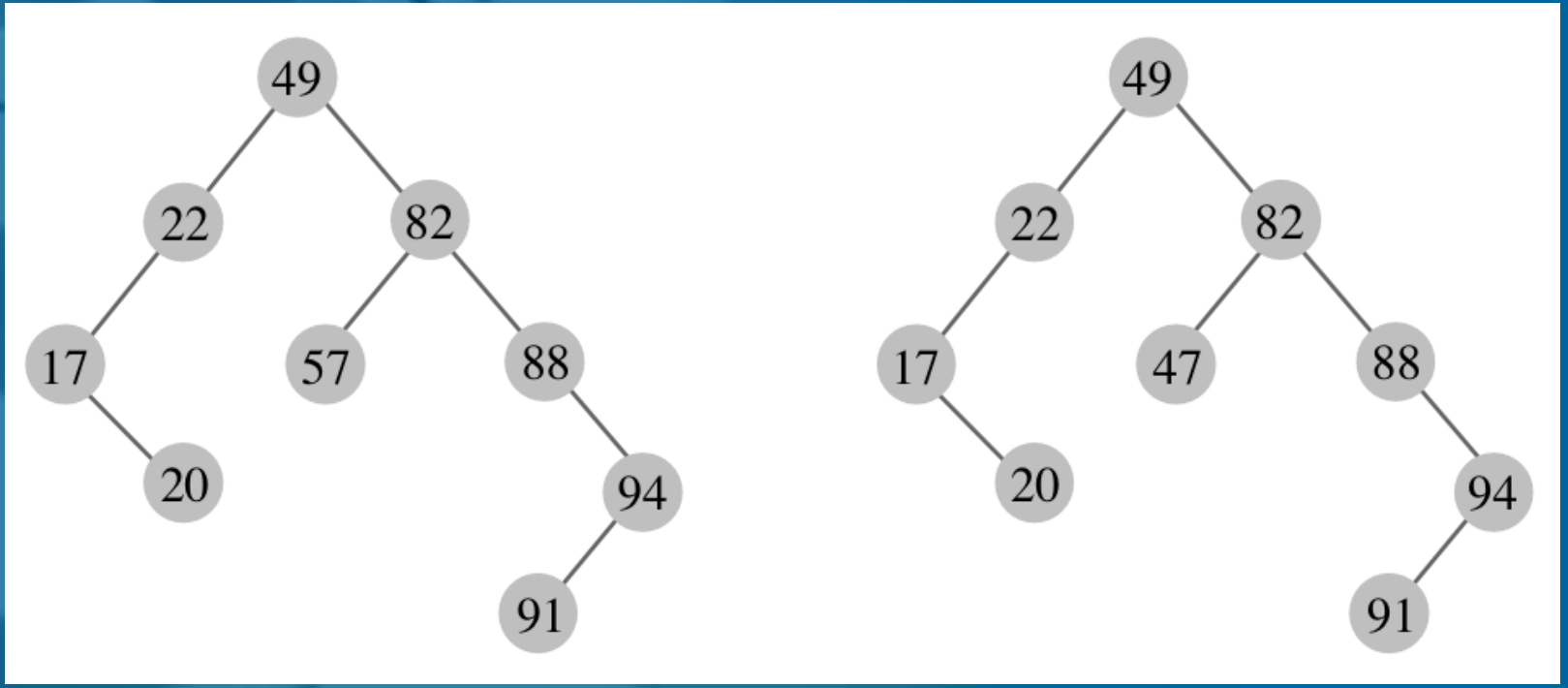
Alberi binari di ricerca
(BST = binary search tree)

Definizione

Albero binario che soddisfa le seguenti proprietà

- ogni nodo *v* contiene un elemento elem(v) cui è associata una chiave chiave(v) presa da un dominio totalmente ordinato
- le chiavi nel sottoalbero sinistro di v sono ≤ chiave(*v*)
- le chiavi nel sottoalbero destro di v sono ≥ chiave(*v*)

Esempi



Albero binario di ricerca

Albero binario non di ricerca

search(chiave k) -> elem

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

```
algoritmo search(chiave k) → elem
1.    v ← radice di T
2.    while ( v ≠ null ) do
3.        if ( k = chiave(v) ) then return elem(v)
4.        else if ( k < chiave(v) ) then v ← figlio sinistro di v
5.        else v ← figlio destro di v
6.    return null
```

insert(elem e, chiave k)

1. Crea un nuovo nodo u con elem=e e chiave=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 sia mantenuta la proprietà di ricerca

Ricerca del massimo

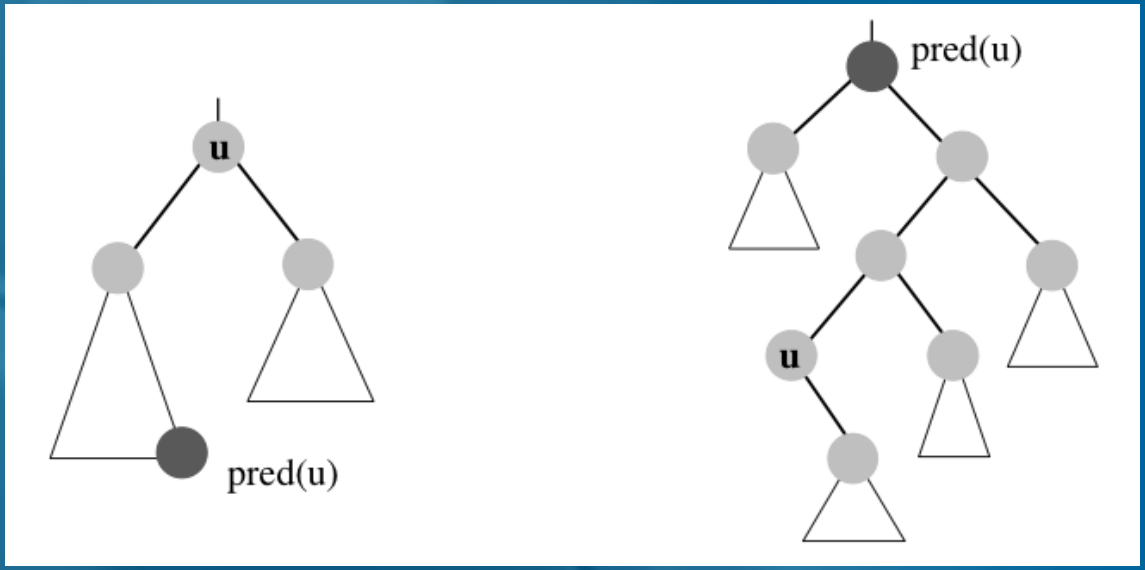
algoritmo `max`$(nodo\ u) \rightarrow nodo$

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

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

1. **if** (u ha figlio sinistro $sin(u)$) **then**
2. **return** `max`$(sin(u))$
3. **while** ($parent(u) \neq null$ e u è figlio sinistro di suo padre) **do**
4. $u \leftarrow parent(u)$
5. **return** $parent(u)$

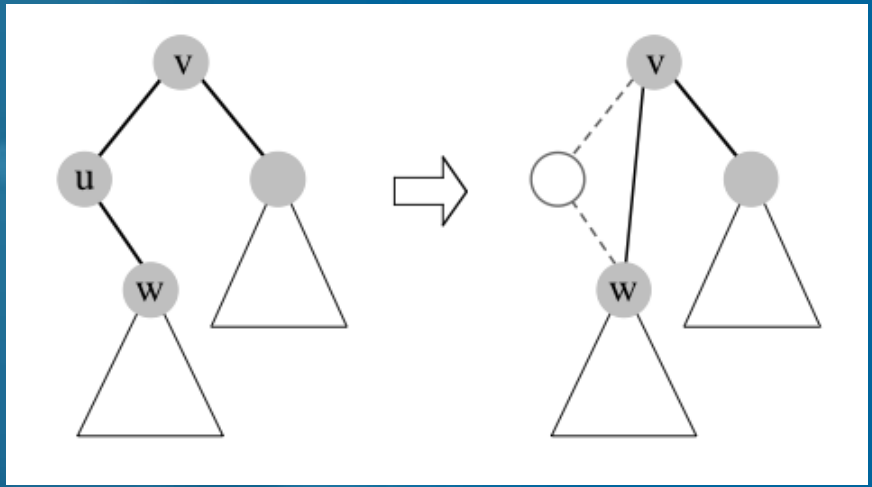


delete(elem e)

Sia u il nodo contenente l'elemento e da cancellare:

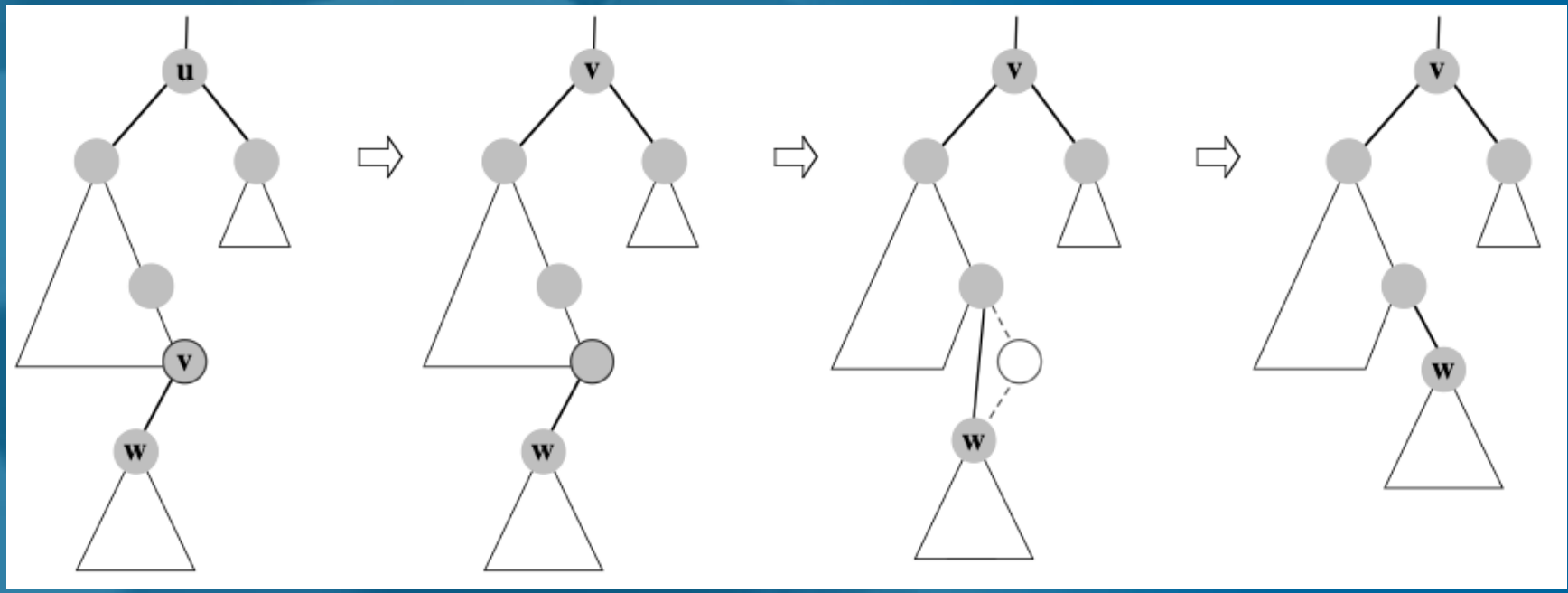
1) u è una foglia: rimuovila

2) u ha un solo figlio:



delete(elem e)

3) u ha due figli: sostituiscilo con il predecessore (v) e rimuovi fisicamente il predecessore (che ha un solo figlio)



Costo delle operazioni

- 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

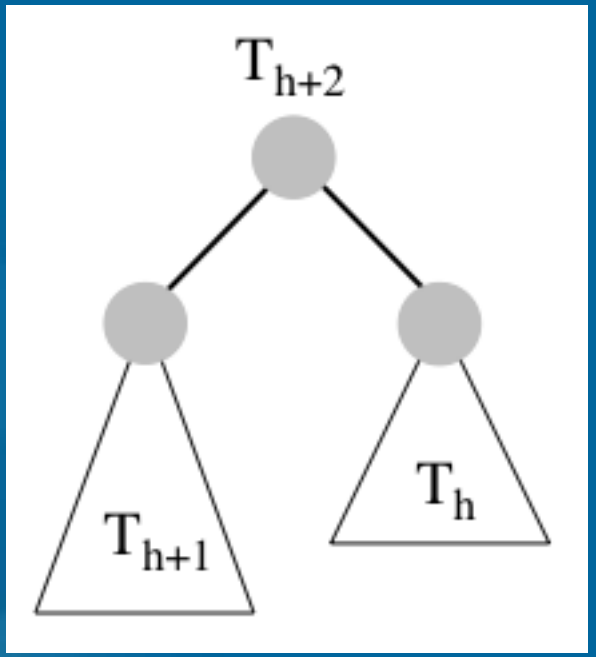
Alberi AVL = alberi binari di ricerca bilanciati in altezza

Altezza di alberi AVL

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

Idea della dimostrazione: considerare, tra tutti gli AVL di altezza h, quelli con il minimo numero di nodi n_h (alberi di Fibonacci)

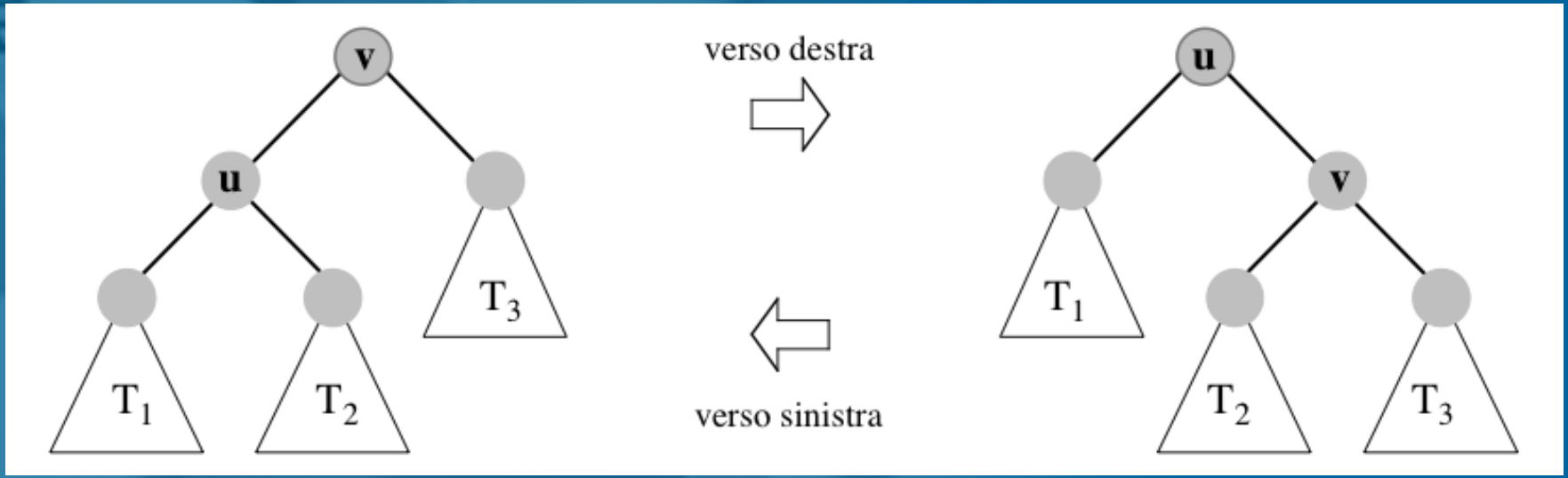
Si ha: $n_h = 1 + n_{h-1} + n_{h-2} = F_{h+3} - 1$



Implementazione delle operazioni

- L'operazione search procede come in un BST
- Ma inserimenti e cancellazioni potrebbero sbilanciare l'albero
- Manteniamo il bilanciamento tramite opportune rotazioni

Rotazione di base



- Mantiene la proprietà di ricerca
- Richiede tempo O(1)

Ribilanciamento tramite rotazioni

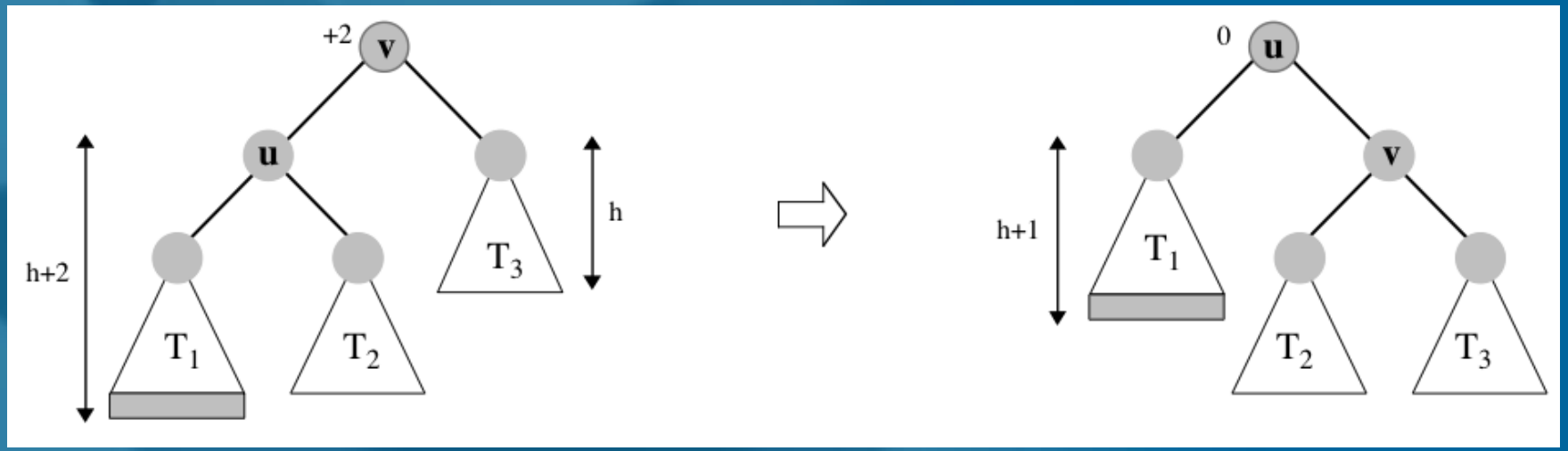
- Le rotazioni sono effettuate su nodi sbilanciati
- Sia v un nodo con fattore di bilanciamento ≥ 2
- Esiste un sottoalbero T di v che lo sbilancia
- A seconda della posizione di T si hanno 4 casi:

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

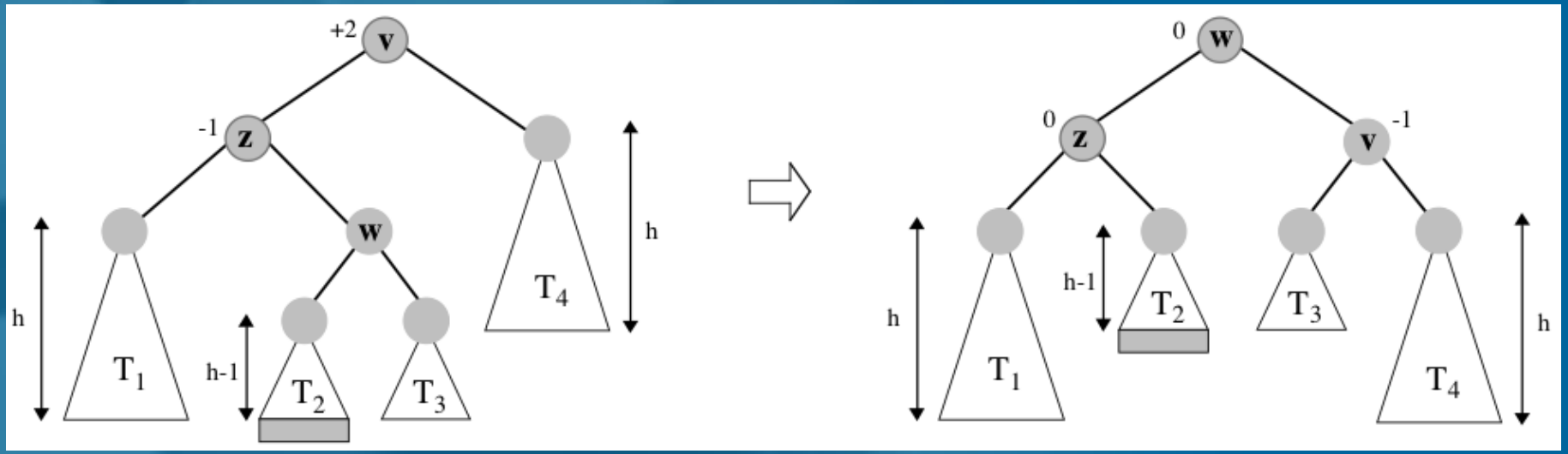
Rotazione SS

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



Rotazione SD

- Applicare due rotazioni semplici: una sul figlio del nodo critico, l'altra sul nodo critico
- L'altezza dell'albero coinvolto nella rotazione passa da h+3 a h+2



insert(elem e, chiave k)

1. Crea un nuovo nodo u con elem=e e chiave=k
2. Inserisci u come in un BST
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

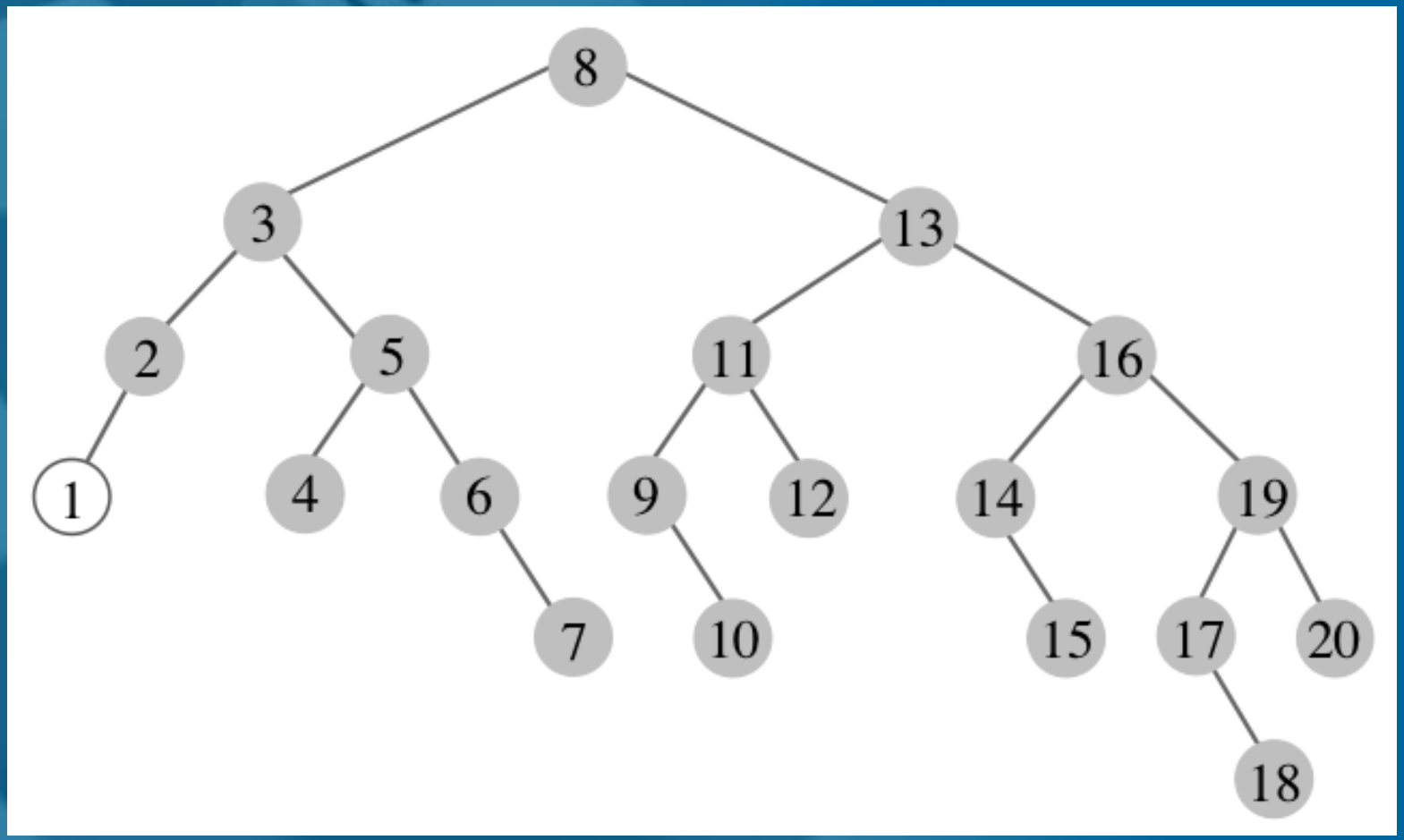
- Oss.: una sola rotazione è sufficiente, poiché l'altezza dell'albero coinvolto diminuisce di 1

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 (che potrebbe essere il predecessore del nodo contenente e)
3. Ripercorrendo il cammino dal basso verso l'alto, esegui l'opportuna rotazione semplice o doppia sui nodi sbilanciati

- Oss.: potrebbero essere necessarie O(log n) rotazioni

Cancellazione con rotazioni a cascata



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) poché l'altezza dell'albero è O(log n) e ciascuna rotazione richiede solo tempo costante

Riepilogo

- Mantenere il bilancimento sembra cruciale per ottenere buone prestazioni
- Esistono vari approcci per mantenere il bilanciamento:
  - Tramite rotazioni
  - Tramite fusioni o separazioni di nodi (alberi 2-3, B-alberi)
- In tutti questi casi si ottengono tempi di esecuzione logaritmici nel caso peggiore
- E' anche possibile non mantenere in modo esplicito alcuna condizione di bilancimento, ed ottenere tempi logaritmici ammortizzati su una intera sequenza di operazioni (alberi auto-aggiustanti)