Alberi Bilanciati di Ricerca

Moreno Marzolla
Dip. di Scienze dell'Informazione
Università di Bologna

marzolla@cs.unibo.it
http://www.moreno.marzolla.name/

Copyright © 2009, Moreno Marzolla, Università di Bologna
(http://www.moreno.marzolla.name/teaching/ASD2009/)

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ *or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.*

Introduzione

- Abbiamo visto che in un ABR è possibile inserire, rimuovere e individuare nodi data la corrispondente chiave in tempo O(h) con h=altezza dell'albero
  - Un albero binario completo con n nodi ha altezza $h=\Theta(\log n)$
- Tuttavia, inserimenti e rimozioni di nodi possono "sbilanciare" l'albero
  - Domanda: individuare una sequenza di *n* inserimenti in un ABR inizialmente vuoto tali che al termine, l'albero risultante abbia altezza $\Theta(n)$
- Il nostro obbiettivo: mantenere bilanciato un ABR, anche a seguito di inserimenti/rimozioni di nodi

Alberi AVL

- Un albero AVL è un albero di ricerca (quasi) bilanciato
- Un albero AVL con n nodi supporta le operazioni insert(), delete(), lookup() con costo O(log n) nel caso pessimo
- Adelson-Velskii, G.; E. M. Landis (1962). *"An algorithm for the organization of information"*. Proceedings of the USSR Academy of Sciences 146: 263–266

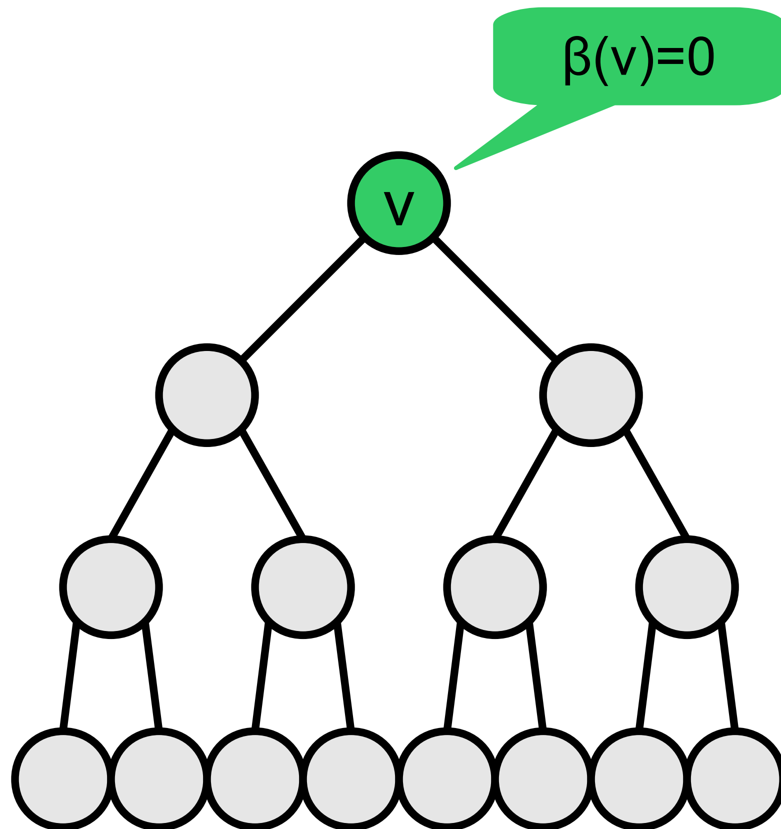
Alcune definizioni

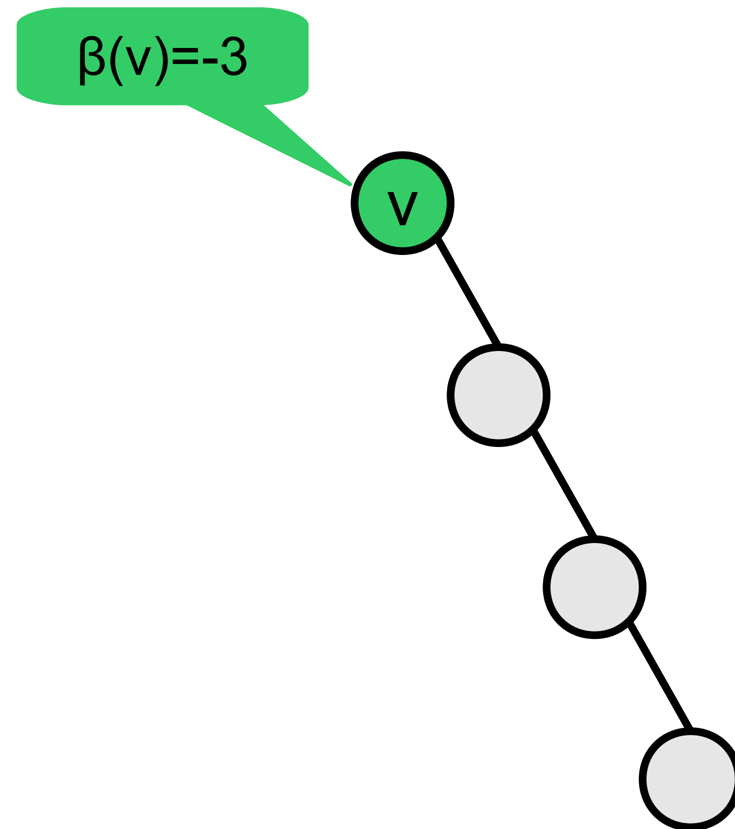
- Fattore di bilanciamento
  - Il *fattore di bilanciamento* $\beta(v)$ di un nodo v è dato dalla differenza tra l'altezza del sottoalbero sinistro e del sottoalbero destro di v:

$$\beta(v) = \text{altezza}(\text{sin}(v)) - \text{altezza}(\text{des}(v))$$

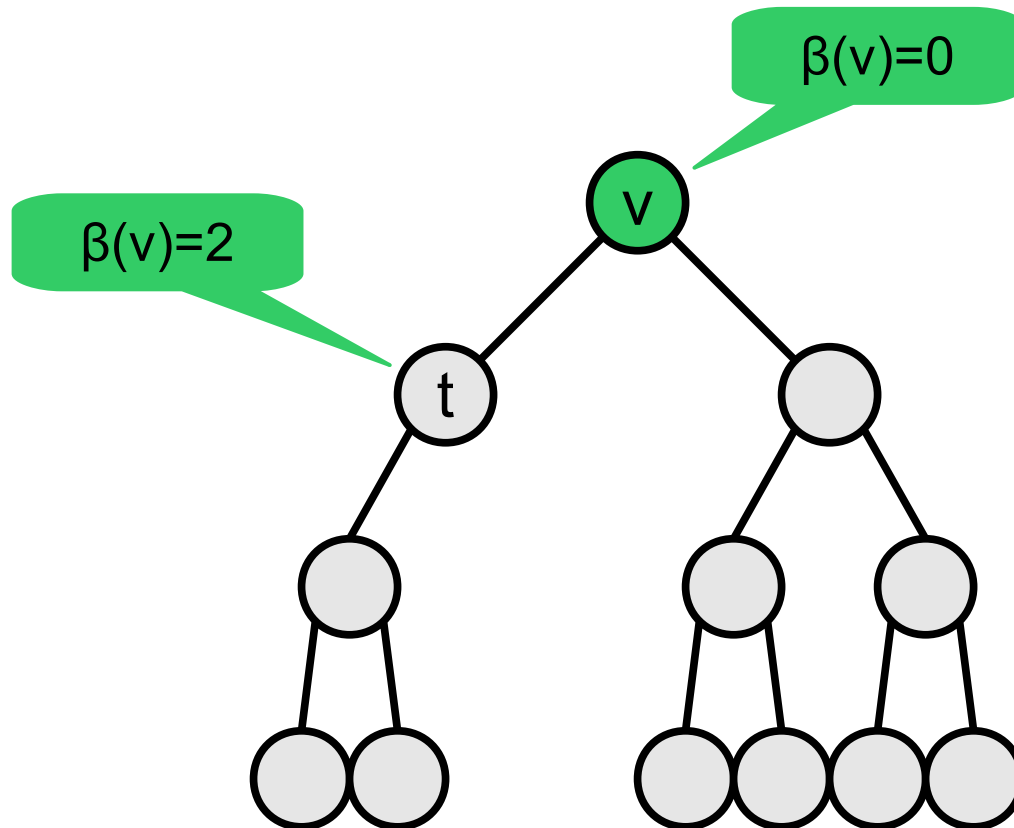
- Bilanciamento in altezza
  - Un albero si dice bilanciato in altezza se le altezze dei sottoalberi sinistro e destro di ogni nodo differiscono al più di uno
  - In altre parole, un albero è bilanciato in altezza se per ogni suo nodo v, si ha $|\beta(v)| \leq 1$
- Definizione: un albero AVL è un ABR bilanciato in altezza

Esempio



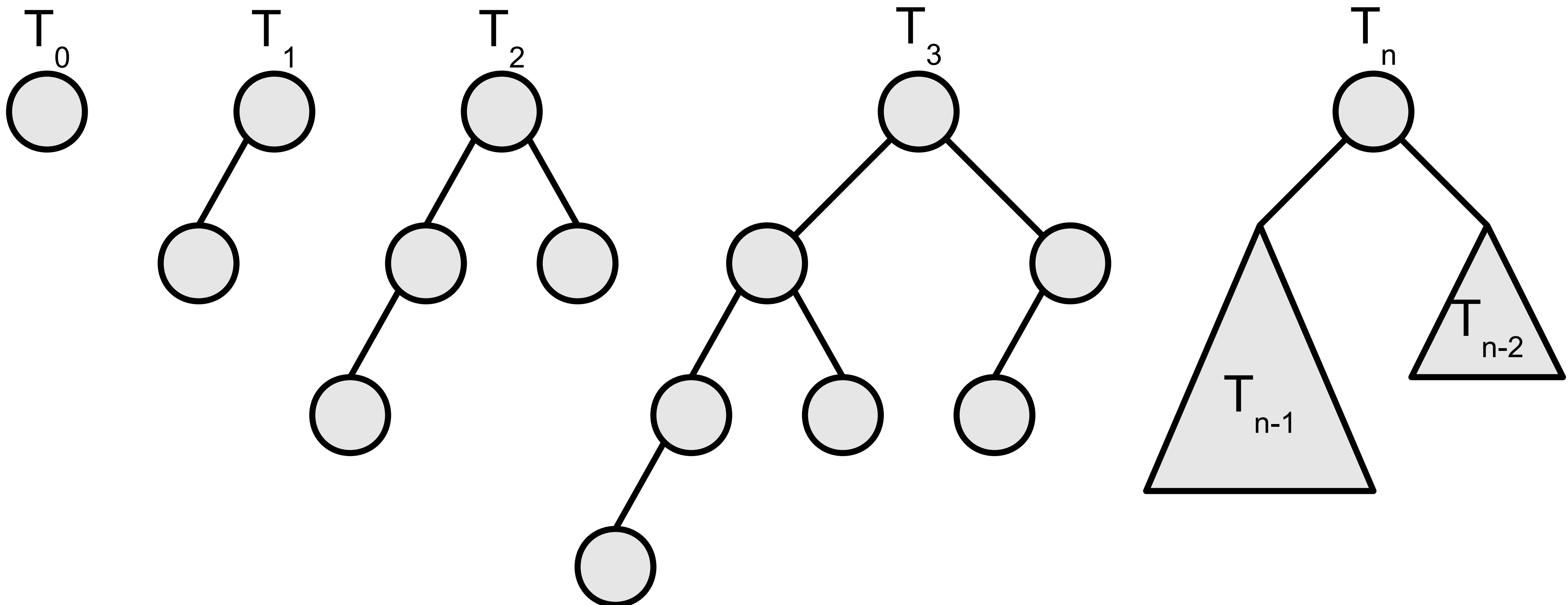


Esempio



Altezza di un albero AVL

- Per valutare l'altezza di un albero AVL, consideriamo gli alberi "più sbilanciati" che si possano costruire
- Alberi di Fibonacci



Altezza di un albero Fibonacci

- Consideriamo un albero di Fibonacci di altezza h. Sia n_h il numero dei nodi
- Per costruzione si ha

$$n_h = n_{h-1} + n_{h-2} + 1$$

- Dimostriamo che

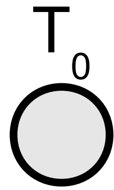
$$n_h = F_{h+3} - 1$$

ove F_n è l'n-esimo numero di Fibonacci

Altezza di un albero Fibonacci

$$n_h = F_{h+3} - 1$$

- Base: h=0
  - $n_0 = 1$
  - $F_3 = 3$



- Passo induttivo

$$\begin{aligned} n_h &= n_{h-1} + n_{h-2} + 1 \\ &= (F_{h+2} - 1) + (F_{h+1} - 1) + 1 \\ &= F_{h+2} + F_{h+1} - 1 \\ &= F_{h+3} - 1 \end{aligned}$$

Altezza di un albero di Fibonacci

- Quindi ricapitolando: un albero di Fibonacci di altezza h ha $F_{h+3} - 1$ nodi
- Ricordiamo che

$$F_h = \Theta(\phi^h), \phi \approx 1.618$$

da cui otteniamo

$$n_h = F_{h+3} - 1 = \Theta(\phi^h)$$

e possiamo quindi concludere che

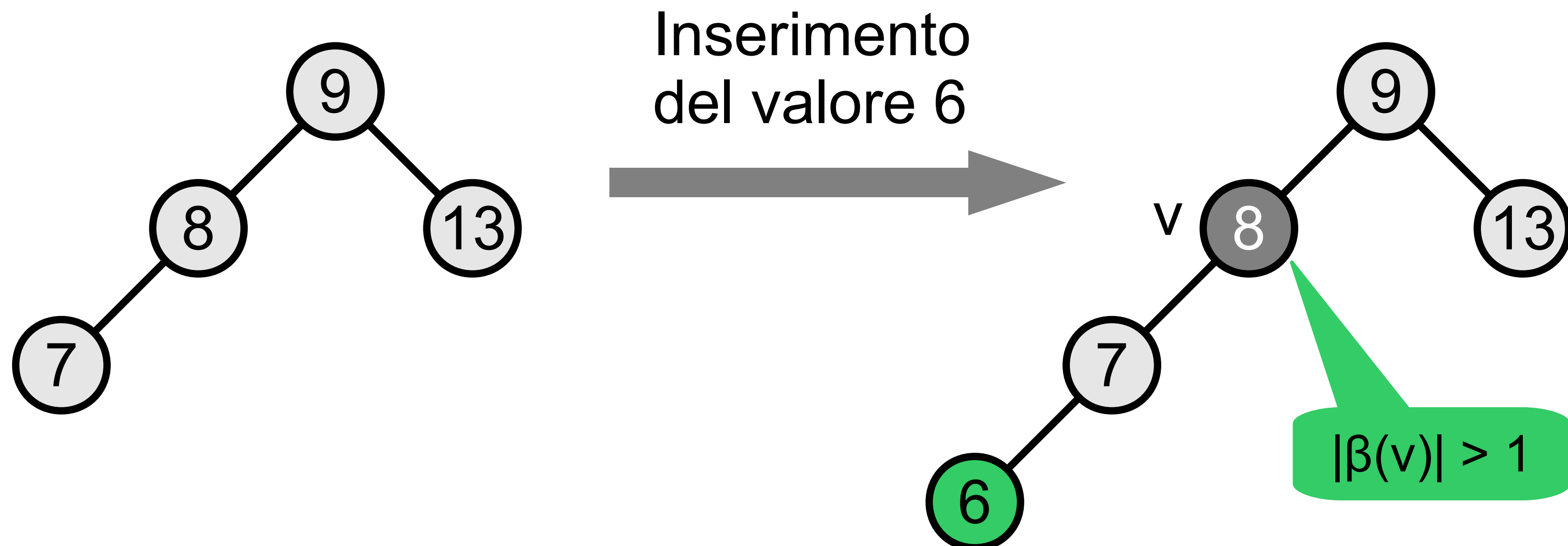
$$h = \Theta(\log n_h)$$

Conclusione

- Poiché...
  - l'albero di Fibonacci con n nodi è quello che tra tutti gli alberi AVL con n nodi ha altezza massima;
  - l'altezza di un albero di Fibonacci con n nodi è proporzionale a (log n)
- ...si conclude che:
  - l'altezza di un albero AVL con n nodi è O(log n)

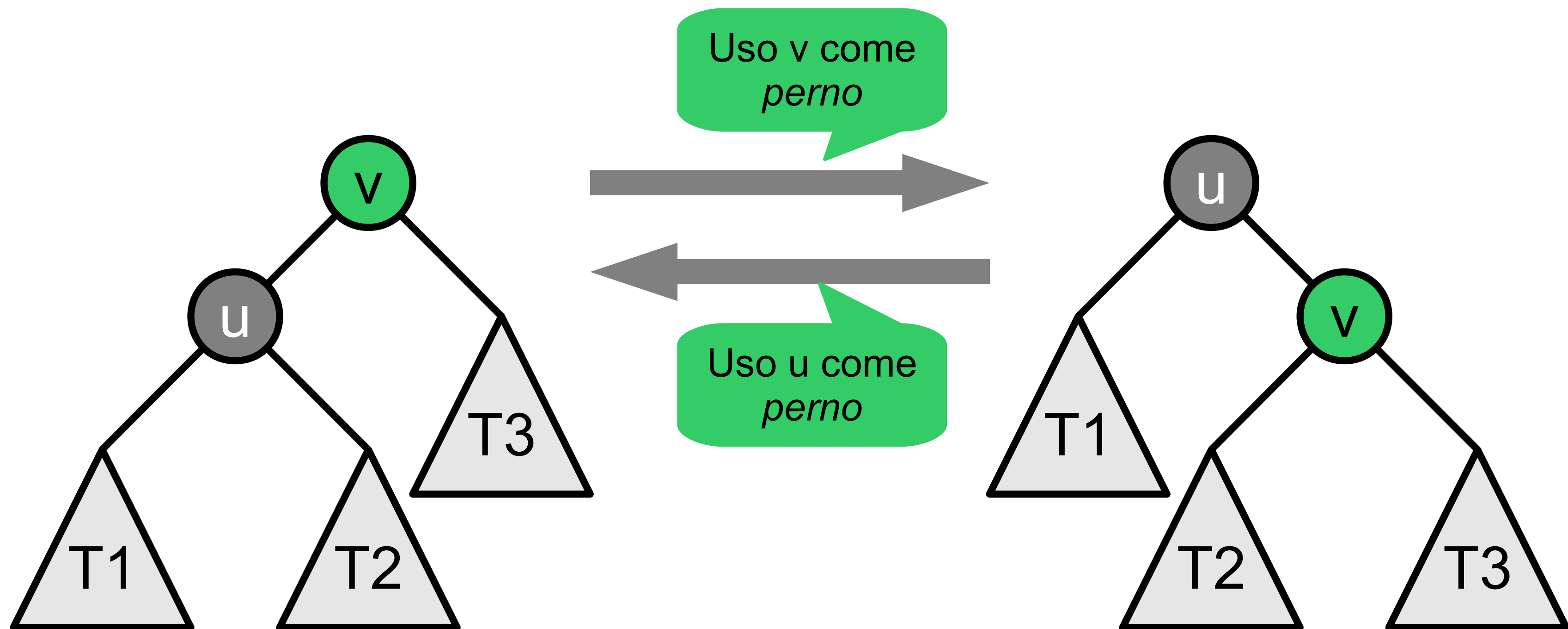
Mantenere il bilanciamento

- La ricerca in un albero AVL viene effettuata come in un generico ABR
- Inserimenti e rimozioni invece richiedono di essere modificati per mantenere il bilanciamento dell'albero
- Esempio



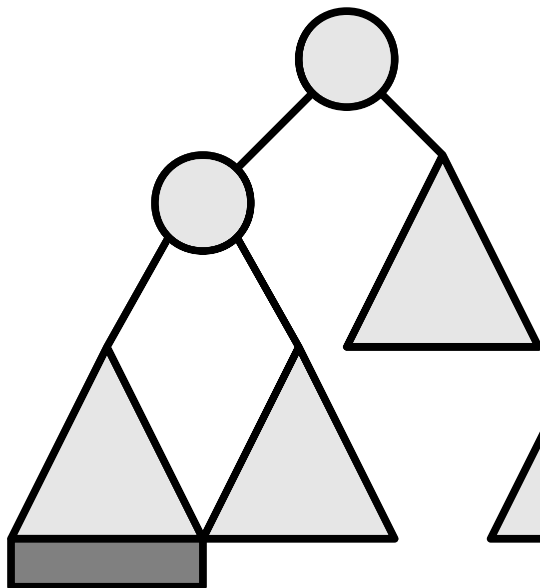
Rotazioni

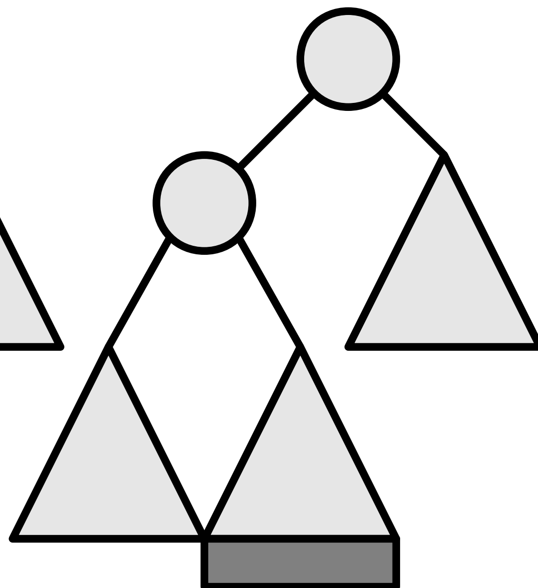
- L'operazione fondamentale per ribilanciare l'albero è la rotazione semplice
  - Domanda: dimostrare che la rotazione semplice preserva la proprietà d'ordine degli ABR

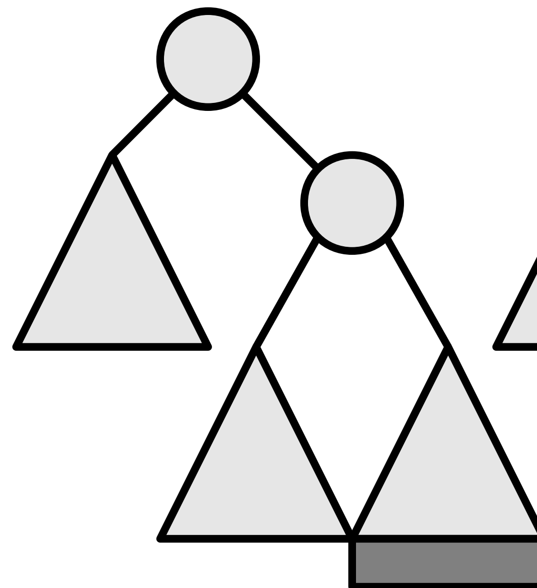


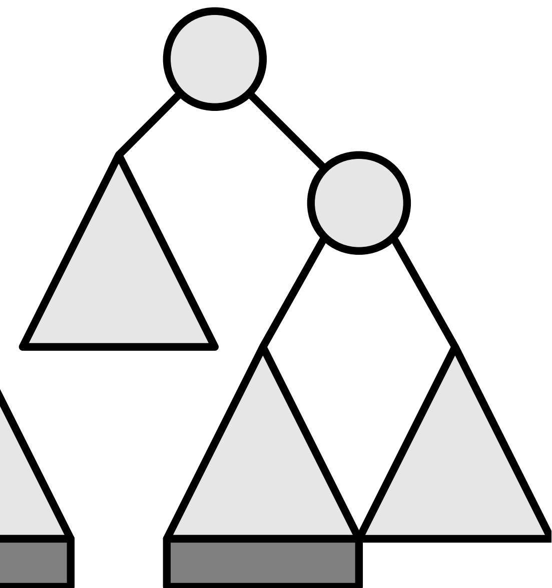
Rotazioni

- Supponiamo che a seguito di un inserimento o cancellazione, una parte dell'albero sia sbilanciata
- Abbiamo quattro casi (simmetrici due a due)



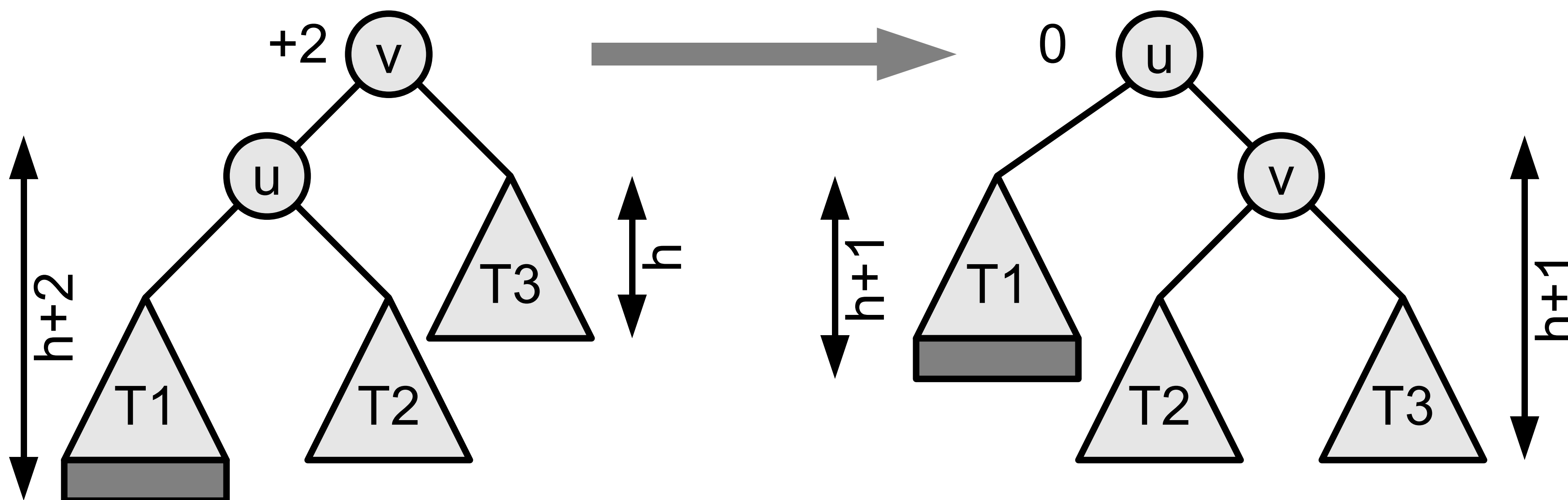




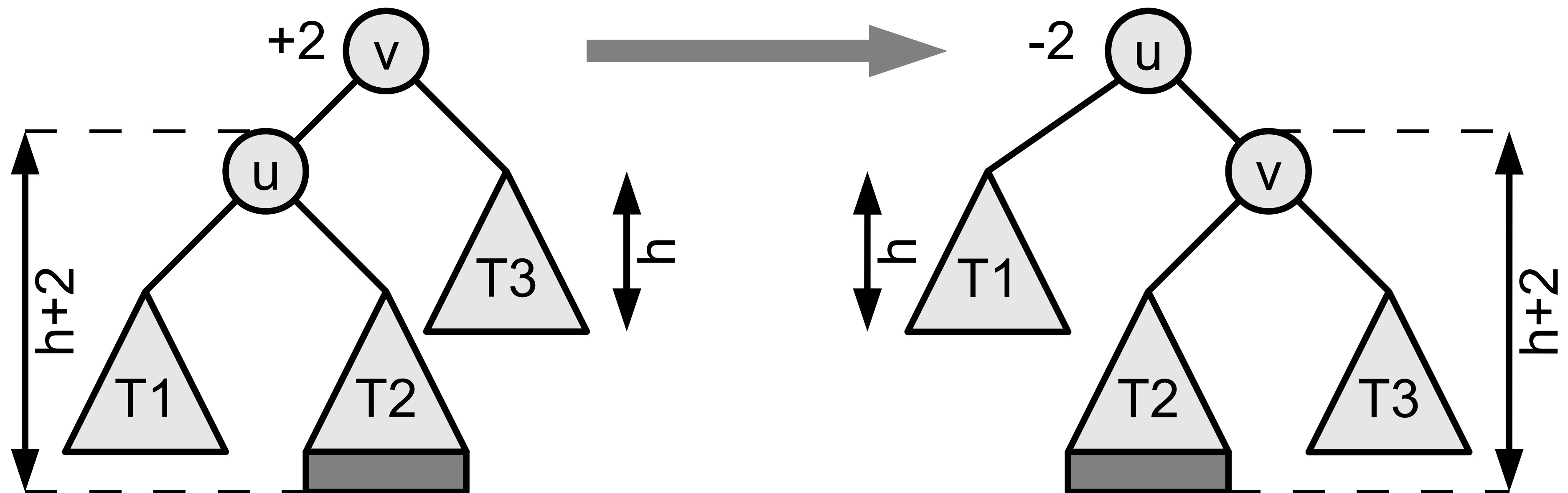


Ribilanciamento: rotazione SS

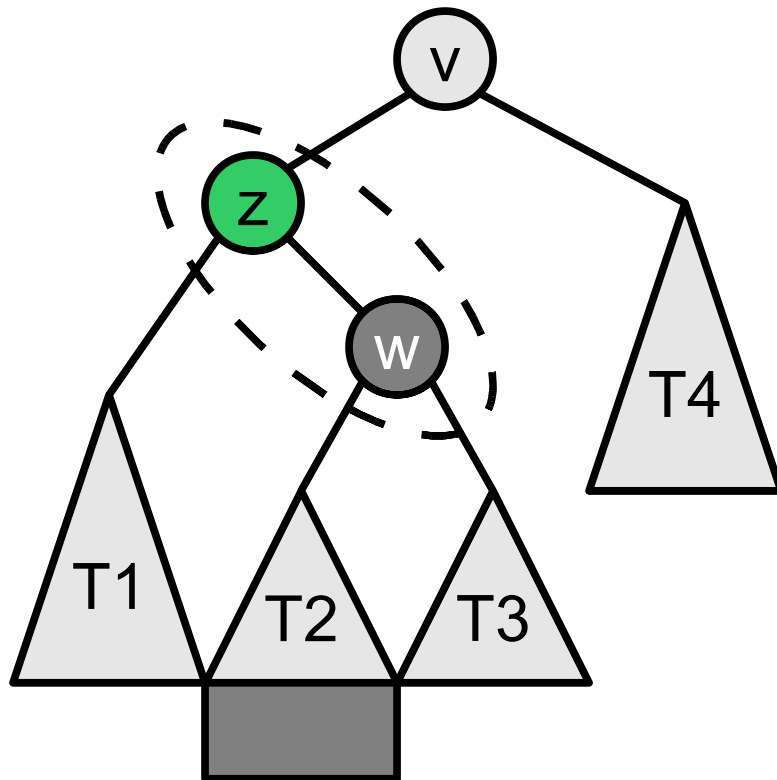
- Si applica una rotazione semplice verso destra su v
- Ha costo O(1)

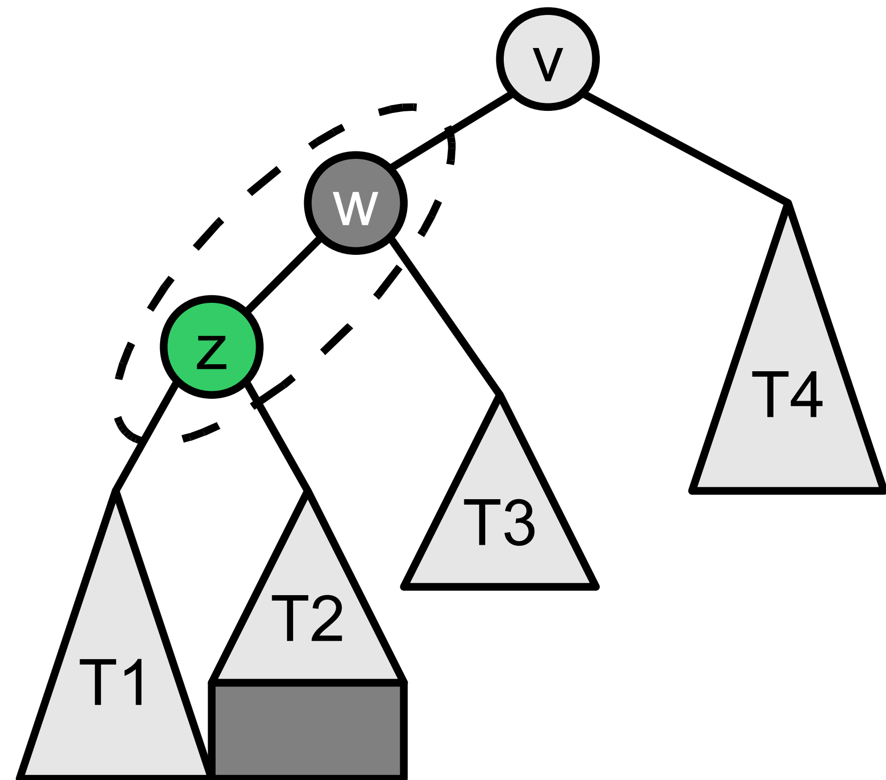


Ribilanciamento: rotazione SD (non funziona!)

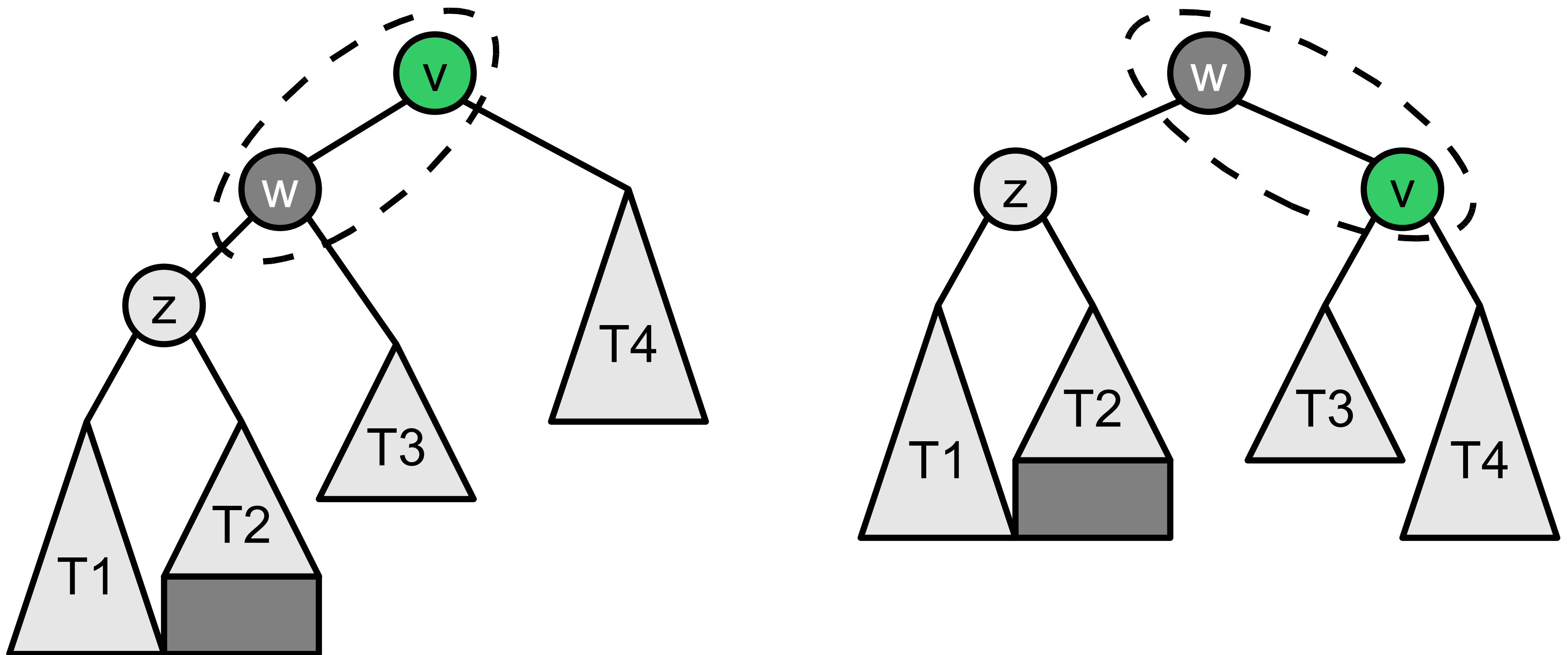


Ribilanciamento: rotazione SD primo passo

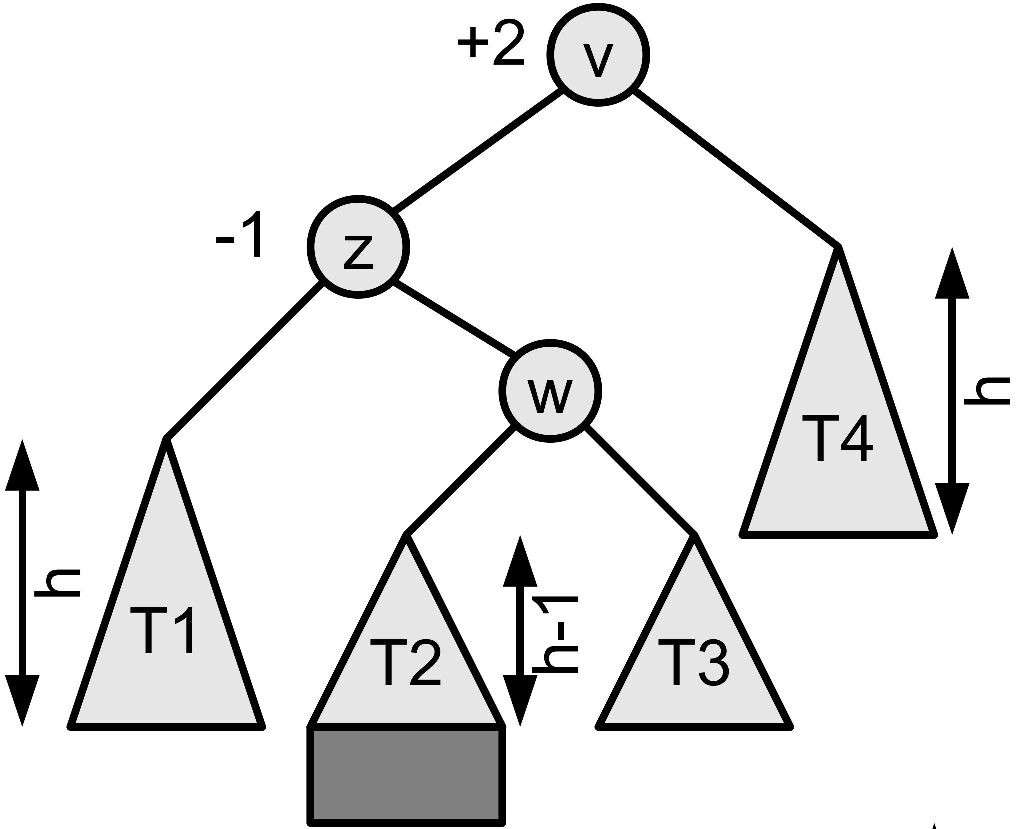




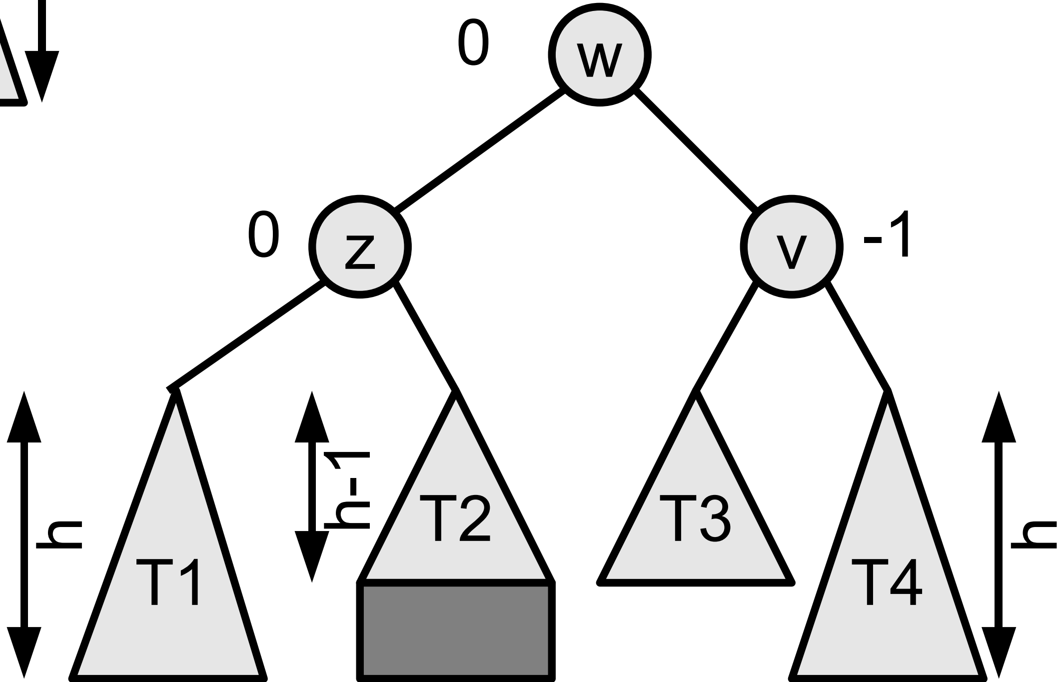
Ribilanciamento: rotazione SD secondo passo



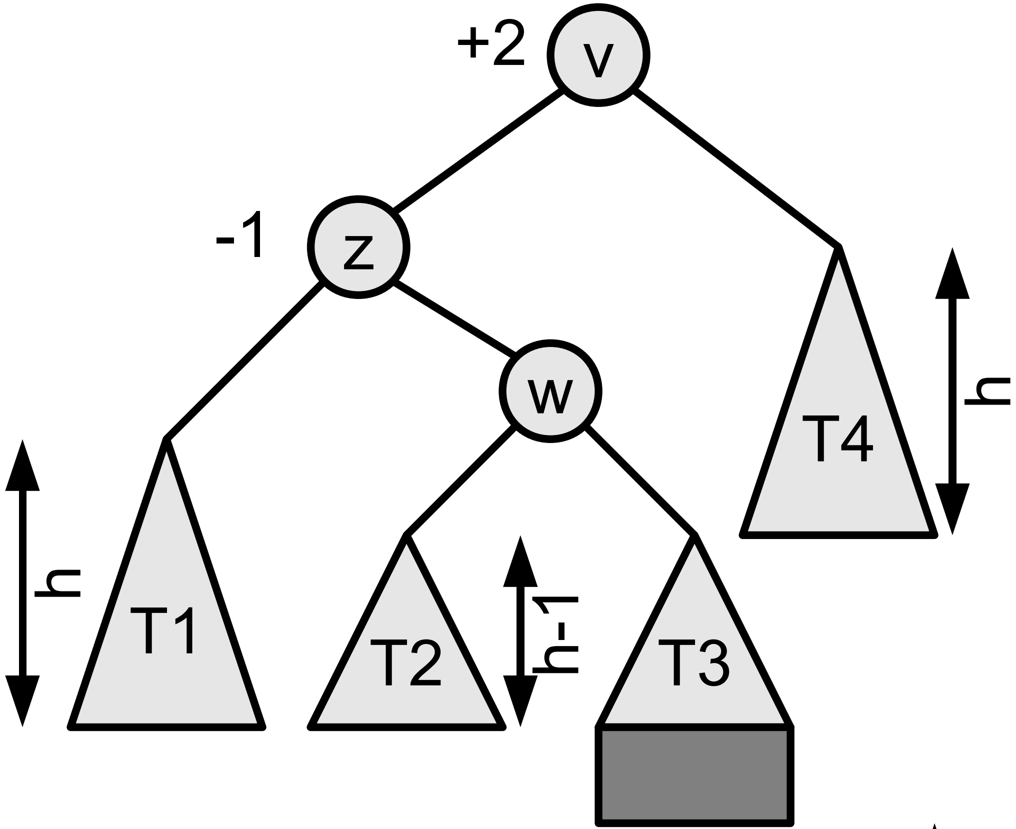
Ribilanciamento: rotazione SD caso 1



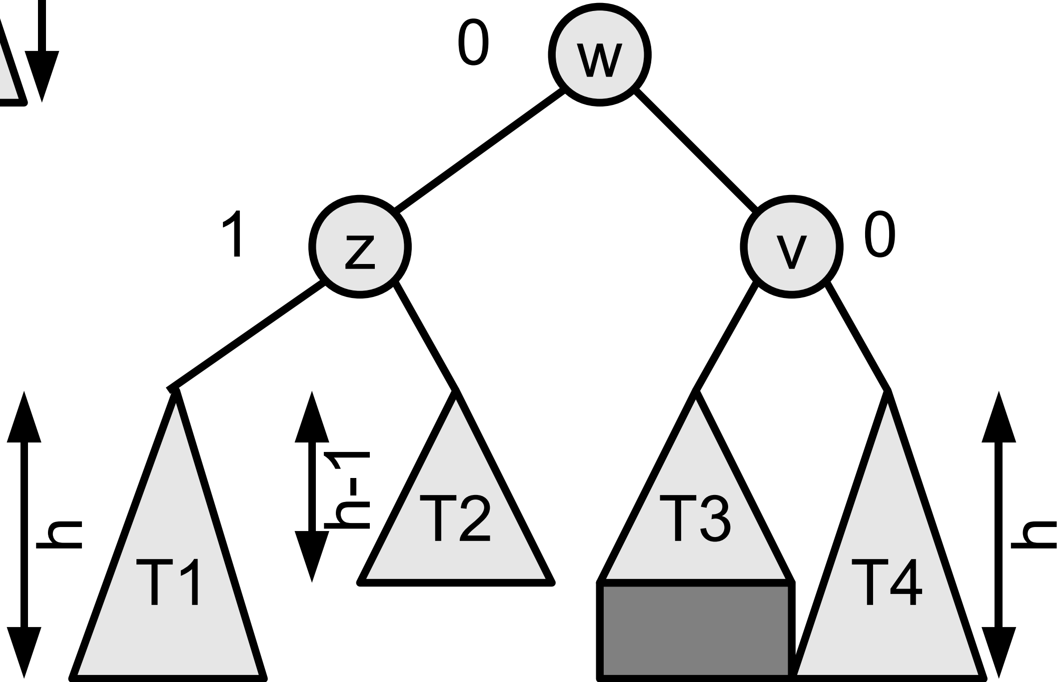
Rotazione doppia: la prima a sinistra con perno z, la seconda a destra con perno v



Ribilanciamento: rotazione SD caso 2



Rotazione doppia: la prima a sinistra con perno z, la seconda a destra con perno v



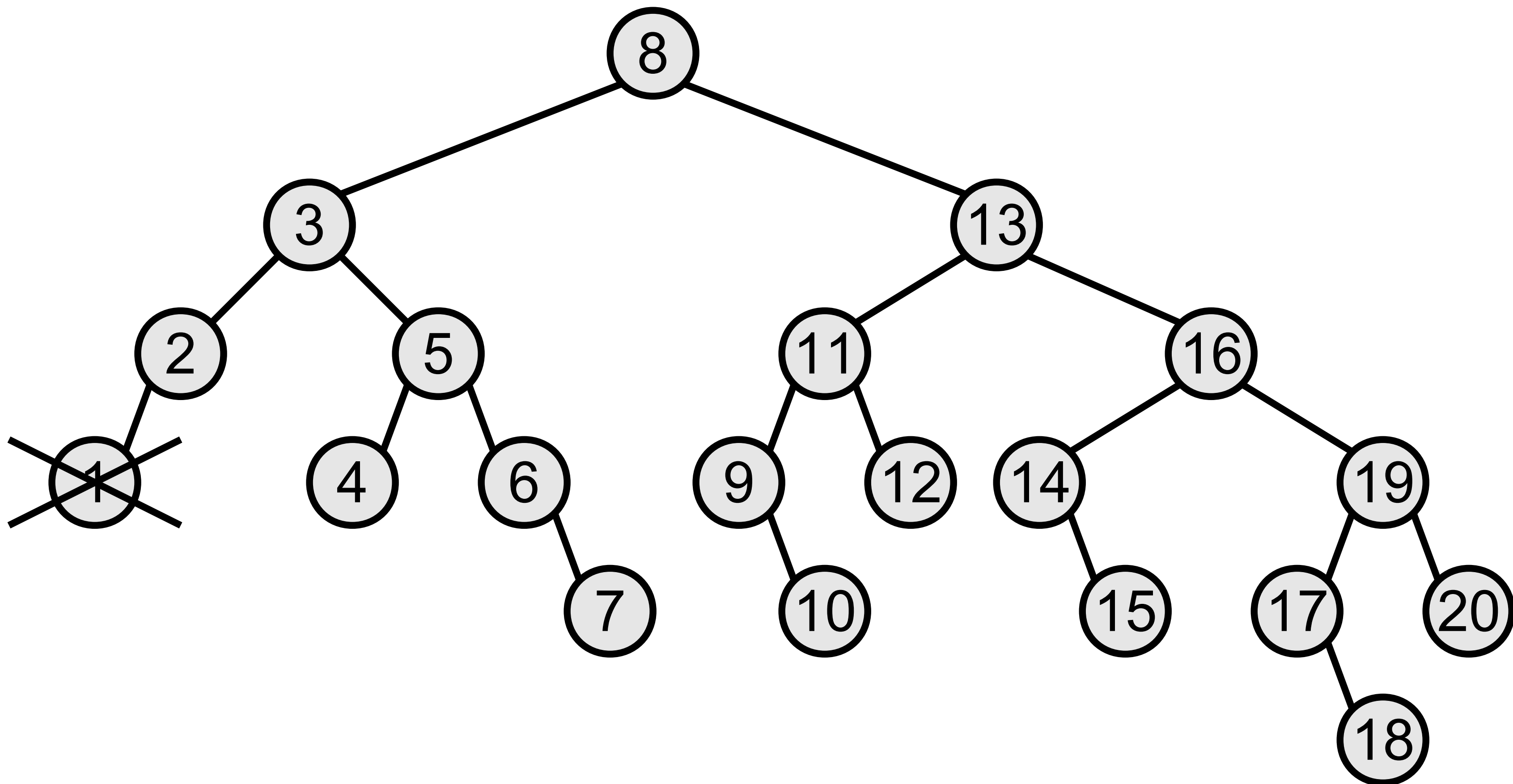
Alberi AVL: Inserimento

- Si inserisce il nuovo valore come per gli ABR
- Si ricalcolano tutti i fattori di bilanciamento mutati
  - Al più il ricalcolo riguarderà un cammino dalla foglia appena inserita fino alla radice, quindi ha costo O(log n)
- Se un nodo presenta fattore di bilanciamento ±2 (nodo critico), occorre ribilanciare l'albero mediante una delle rotazioni viste
  - Nota: in caso di inserimento, il nodo critico è unico
- Costo complessivo: O(log n)

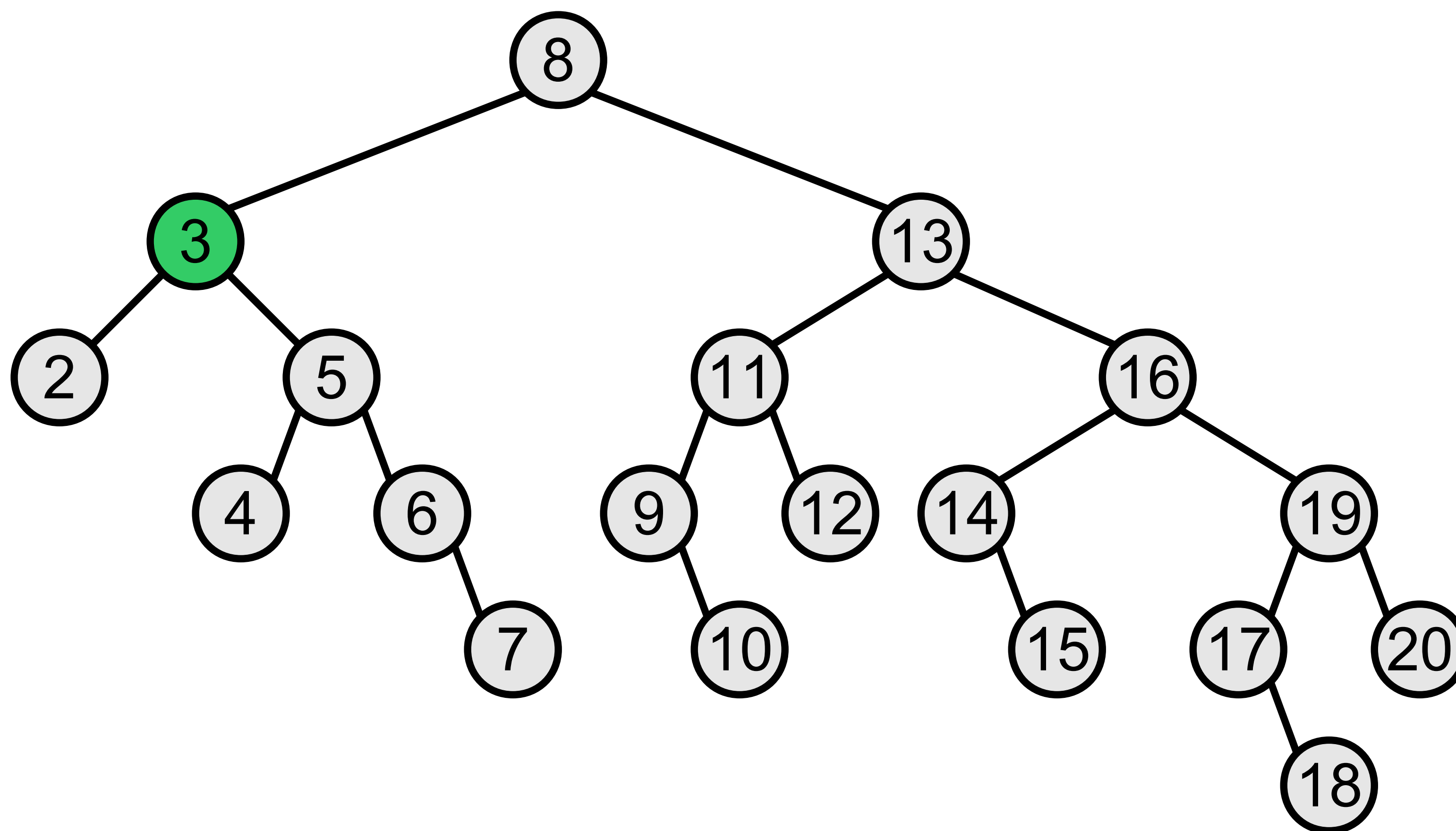
Alberi AVL: Rimozione

- Si rimuove il nodo come per gli ABR
- Si ricalcolano tutti i fattori di bilanciamento mutati
  - Al più il ricalcolo riguarderà un cammino dal padre del nodo eliminato fino alla radice, quindi ha costo O(log n)
- Per ogni nodo con fattore di bilanciamento ±2, occorre ribilanciare l'albero mediante una delle rotazioni viste
  - Nota: nel caso della rimozione, possono comparire più nodi con fattori di bilanciamento ±2
- Costo complessivo: O(log n)

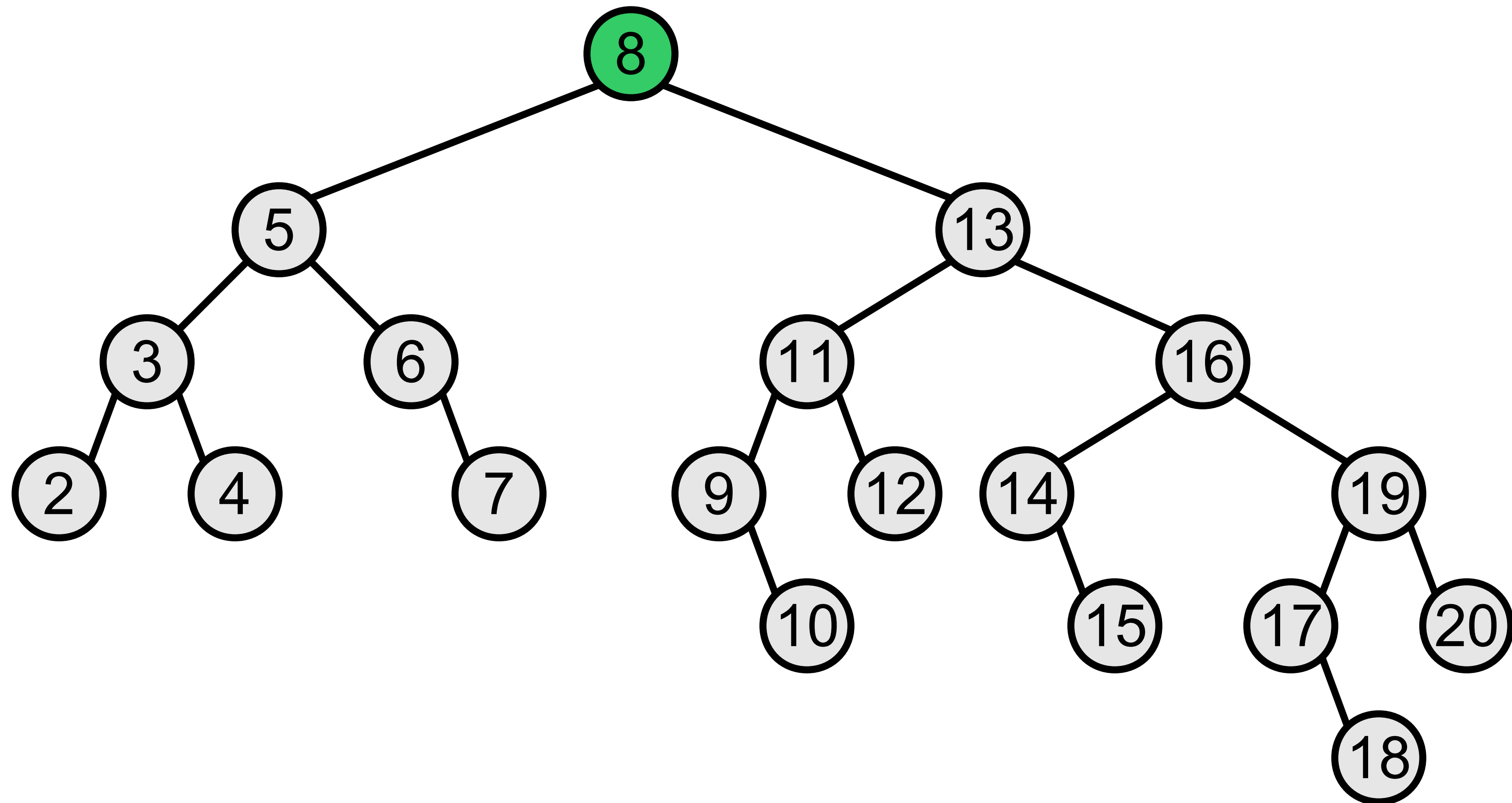
Esempio: cancellazione con rotazioni a cascata



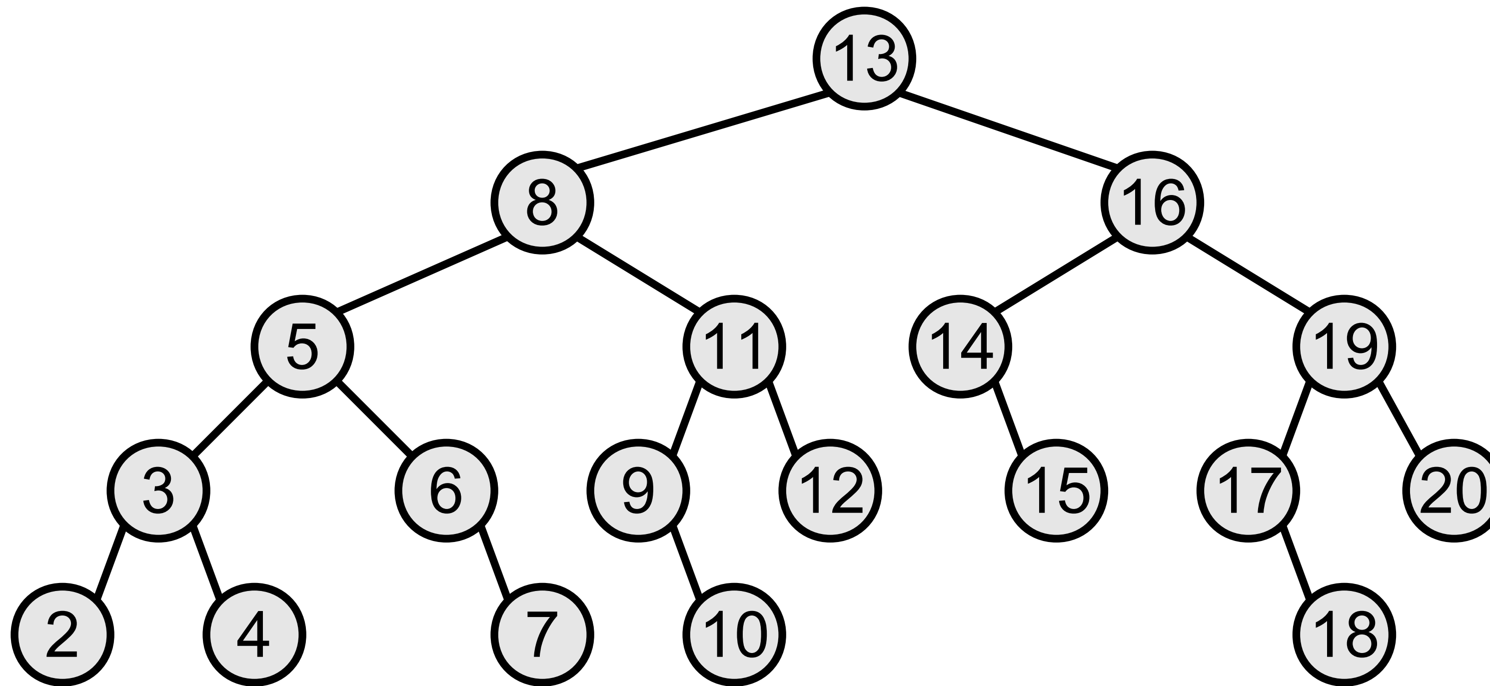
Applicare rotazione a sinistra su 3



Applicare rotazione a sinistra su 8



Albero ribilanciato



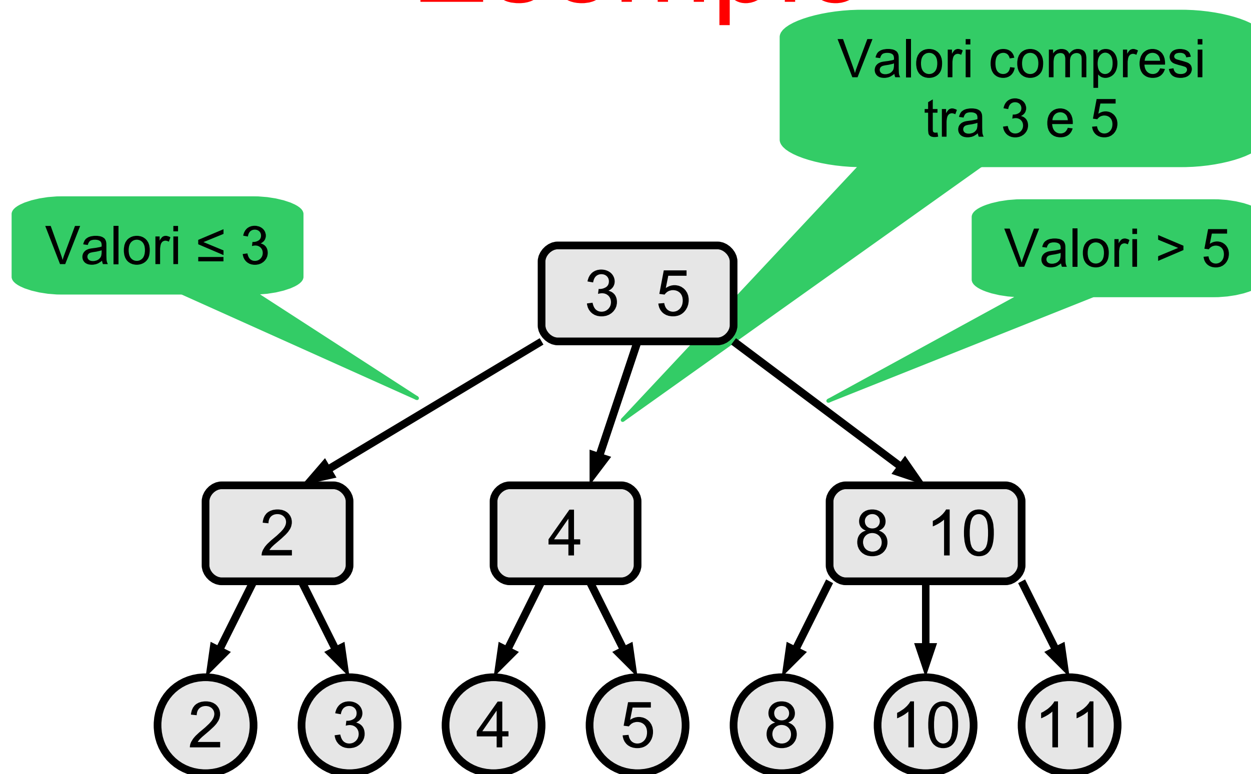
Alberi AVL: Riassunto

- search(Key k)
  - O(log n) nel caso peggiore
- insert(Key k, Item t)
  - O(log n) nel caso peggiore
- delete(Key k)
  - O(log n) nel caso peggiore

Alberi 2-3

- Definizione: un albero 2-3 è un albero in cui:
  - ogni nodo interno ha 2 o 3 figli e tutti i cammini radice-foglia hanno la stessa lunghezza
  - le foglie contengono le chiavi con i valori associati, e sono ordinate da sinistra verso destra in ordine di chiave crescente
  - Ogni nodo interno v mantiene due informazioni
    - S[v] è la massima chiave nel sottoalbero radicato nel figlio sinistro
    - M[v] è la massima chiave nel sottoalbero radicato nel figlio centrale (se v ha solo due figli, conterrà solo S[v])

Esempio



Altezza degli alberi 2-3

- Sia T un albero 2-3 con *n* nodi, *f* foglie ed altezza *h*. Allora valgono le seguenti disuguaglianze

$$2^{h+1}-1 \leq n \leq (3^{h+1}-1)/2$$
$$2^{h} \leq f \leq 3^{h}$$

- In particolare, possiamo concludere che l'altezza di un albero 2-3 è Θ(log n)

Altezza degli alberi 2-3 Dimostrazione

- Induzione su h. se h=0, l'albero consiste di un solo nodo foglia e le relazioni sono verificate
- Se h>0, consideriamo l'albero 2-3 T' privo dell'ultimo livello (le foglie). Sia n' e f' il numero di nodi e foglie di T'
  - Per ipotesi induttiva $2^{h-1} \leq f' \leq 3^{h-1}$
  - Poiché ogni foglia di T' puó avere 2 o 3 figli, si ottiene

$$2 \times 2^{h-1} \leq f \leq 3 \times 3^{h-1}$$
$$2^h \leq f \leq 3^h$$

Altezza degli alberi 2-3 Dimostrazione

- Per il numero di nodi, l'ipotesi induttiva è

$$2^h - 1 \leq n' \leq (3^h - 1)/2$$

- Osserviamo che n = n' + f, da cui combinando

$$2^h - 1 \leq n' \leq (3^h - 1)/2$$

$$2^h \leq f \leq 3^h$$

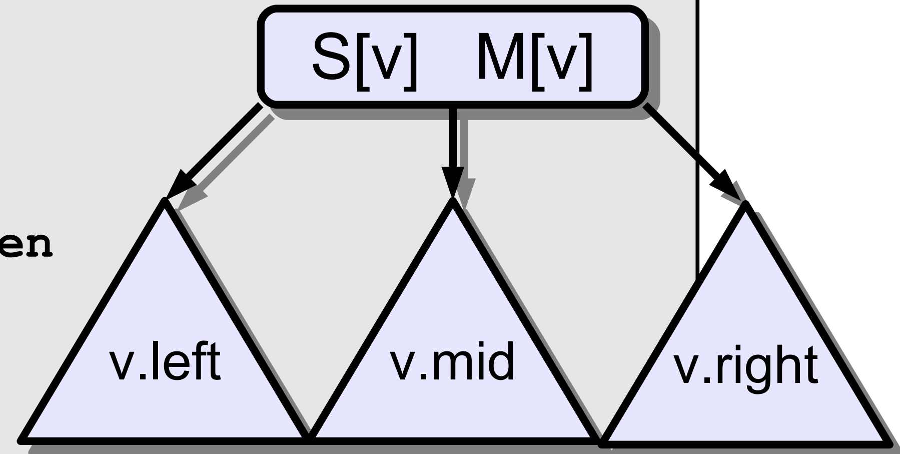
si ottiene

$$2^h + 2^h - 1 \leq n \leq (3^h - 1)/2 + 3^h$$

$$2^{h+1} - 1 \leq n \leq (3^{h+1} - 1)/2$$

Ricerca

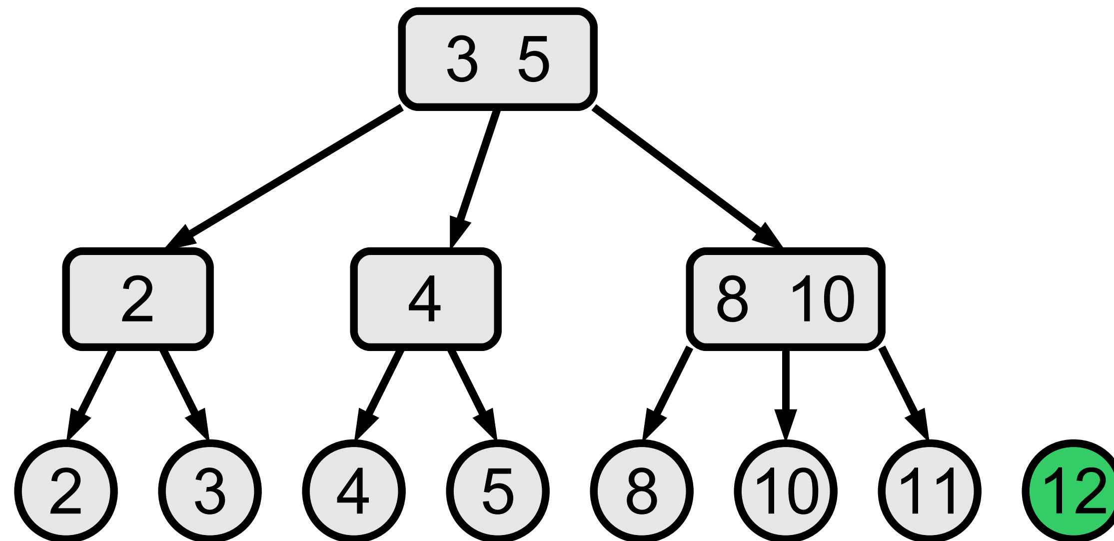
```
Algorithm 23search( T, k )
    if ( T == null ) then
        return null;
    endif
    node v = T.root;
    if ( v è una foglia ) then
        if ( chiave di v == k ) then
            return v;
        else
            return null;
        endif
    else // v non è una foglia
        if ( k ≤ S[v] ) then
            return 23search( v.left, k );
        elseif ( v.right != null && k > M[v] ) then
            return 23search( v.right, k );
        else
            return 23search( v.mid, k );
        endif
    endif
```



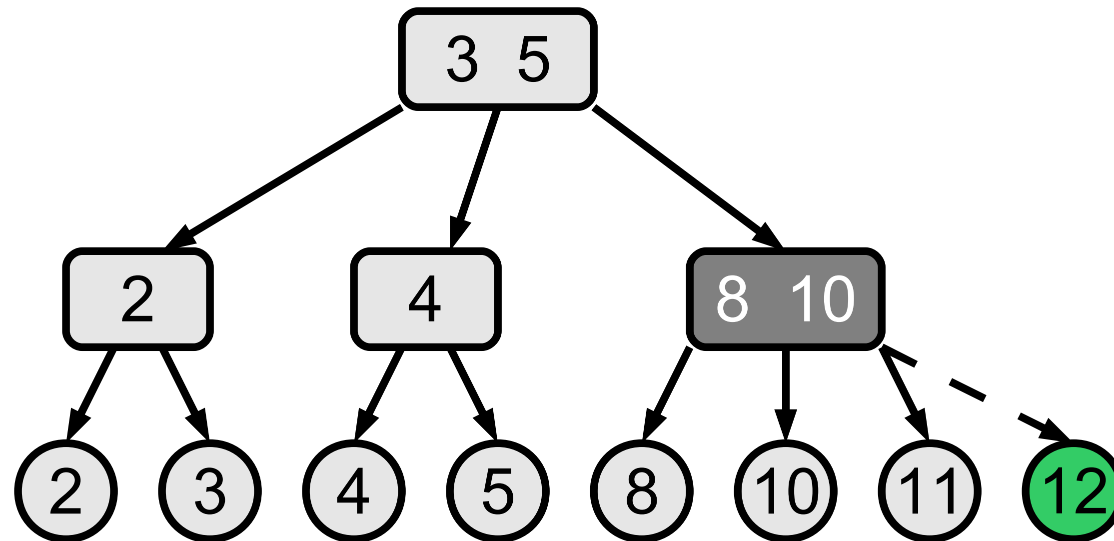
Inserimento

- Si crea una foglia v con chiave k
- Si individua (effettuando l'operazione di ricerca) un nodo u nel penultimo livello che diventerà padre di v
- Si aggiunge v come figlio di u, se possibile
  - Se u ha già 3 figli, occorre effettuare una operazione di *separazione* (split) che potrebbe propagarsi verso la radice

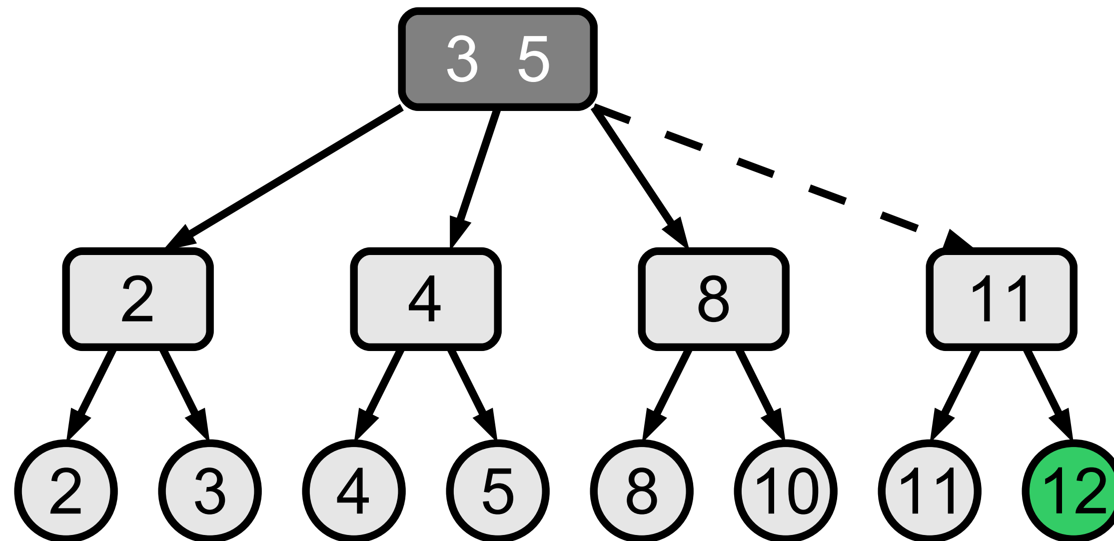
Esempio



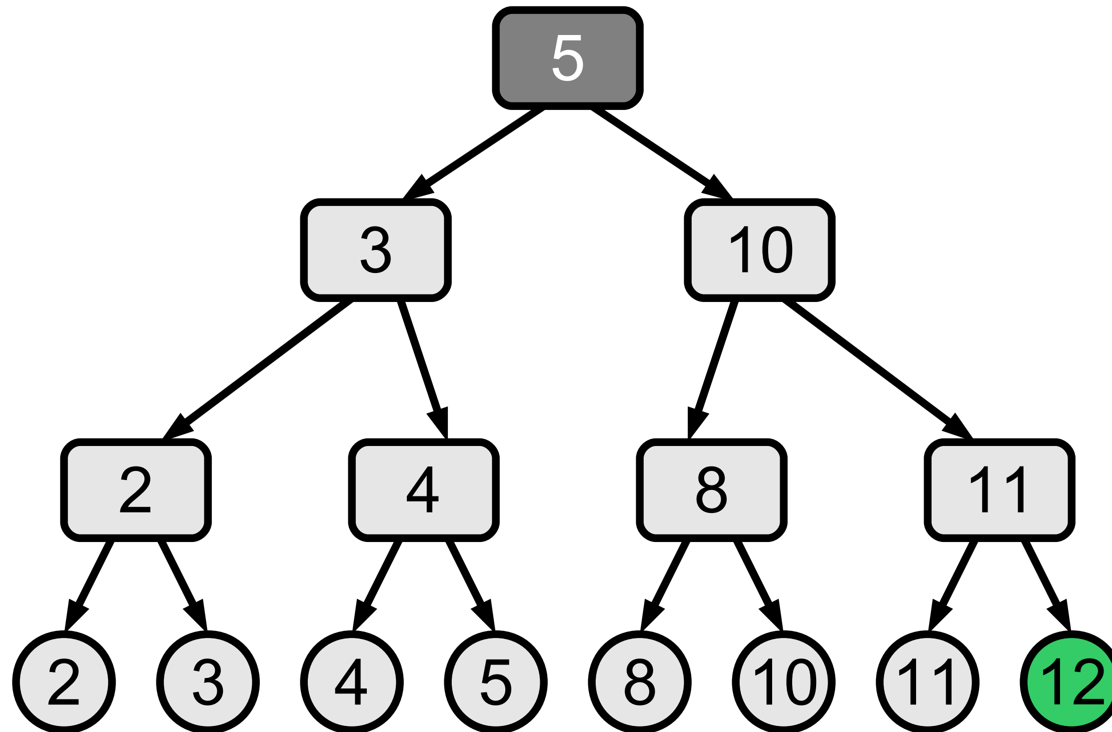
Esempio



Esempio



Esempio

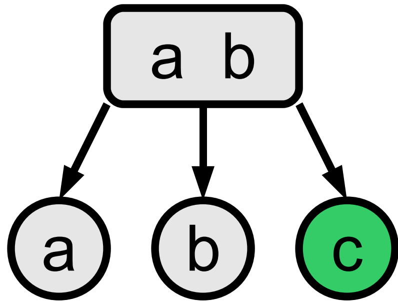
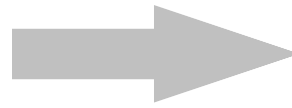
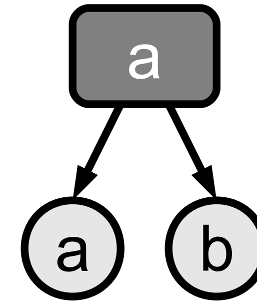
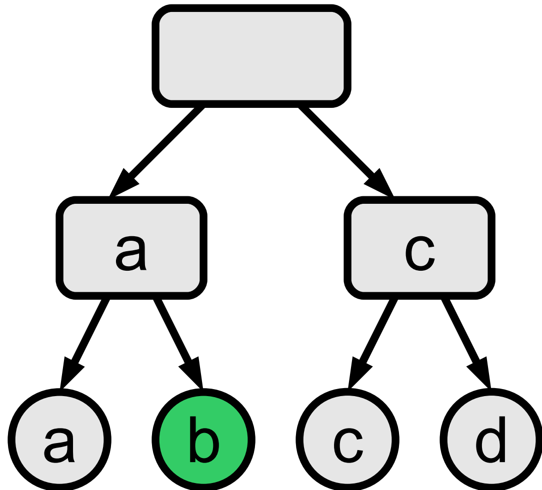
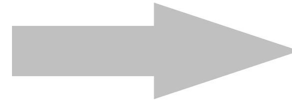
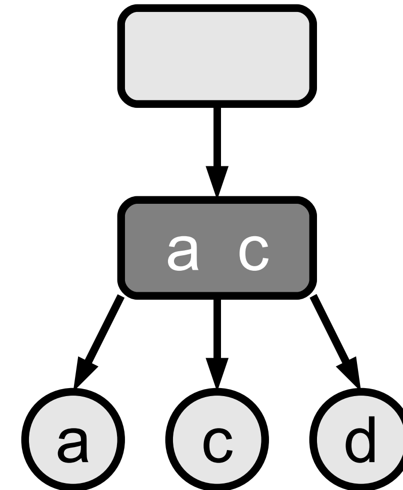
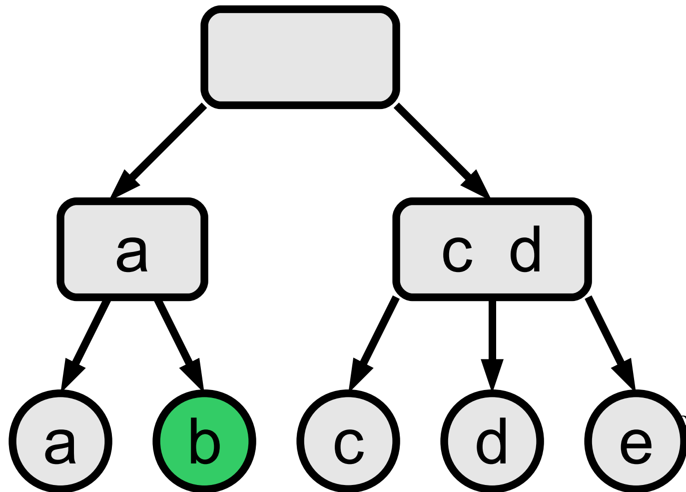
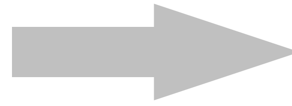
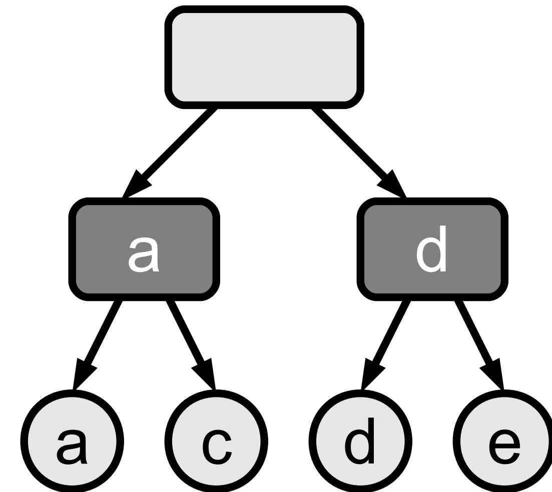


Inserimento: costo

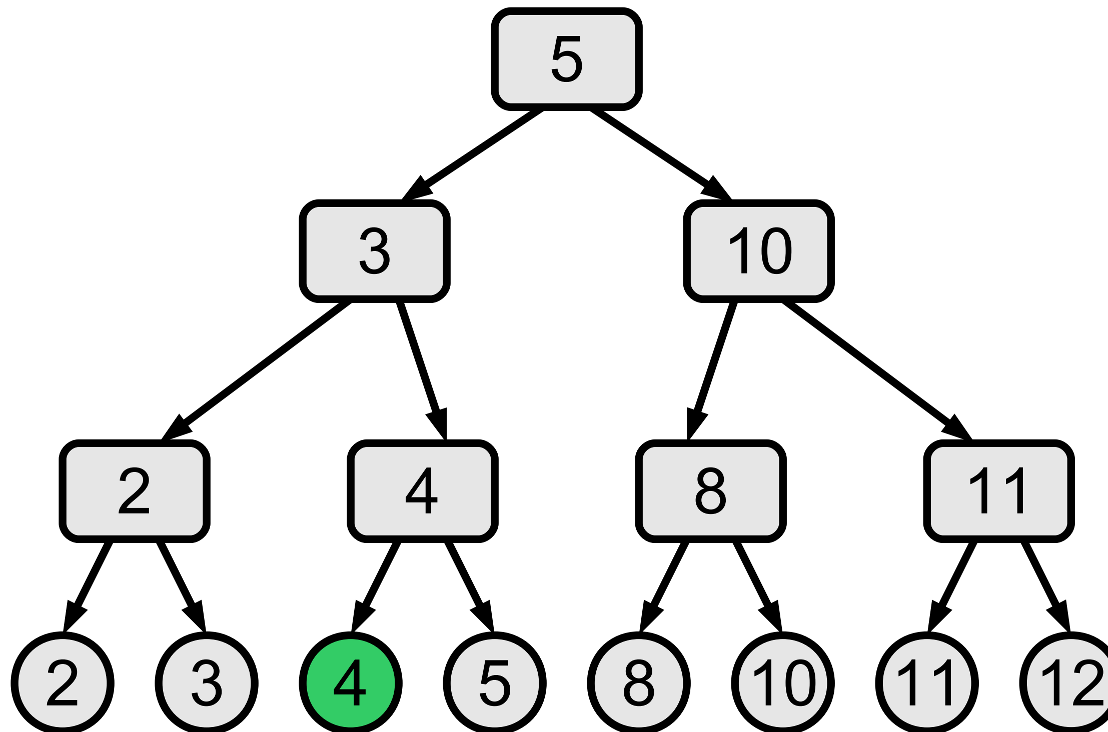
- O(log n) per individuare il padre del nuovo nodo
- O(log n) split nel caso peggiore, ciascuno avente costo O(1)
- Complessivamente il costo di una operazione di inserimento è O(log n)

Cancellazione

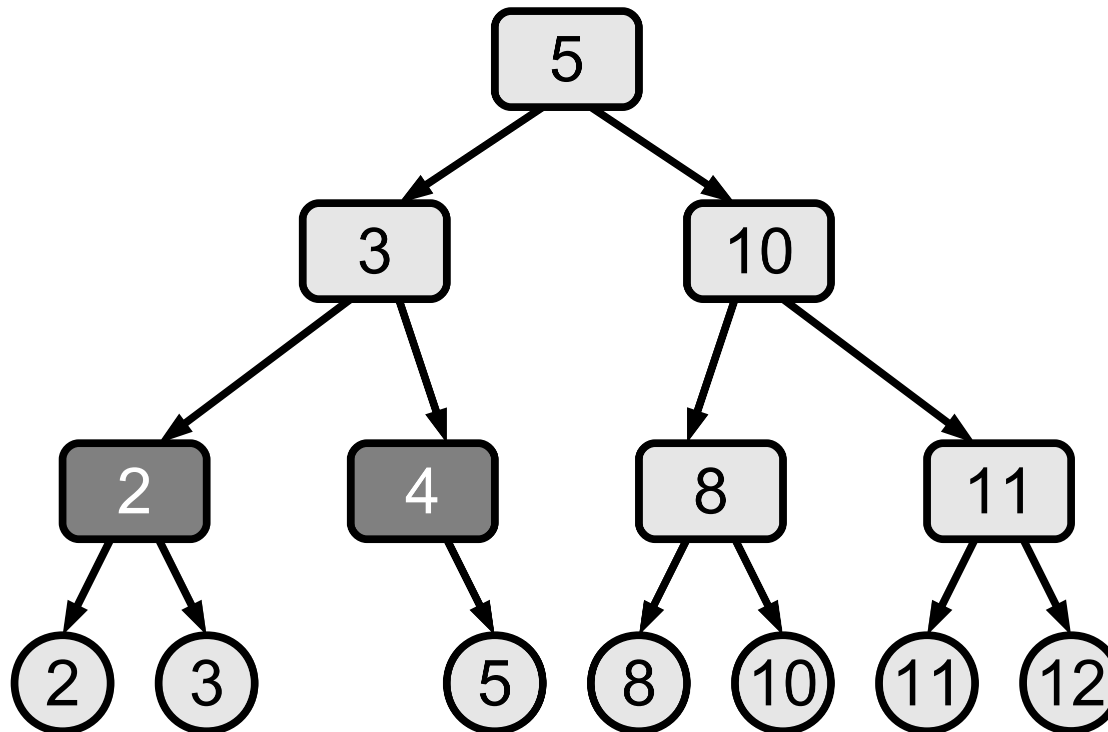
- Si individua la foglia v con la chiave da cancellare
- Si rimuove v, staccandola dal proprio padre u
  - Se u aveva 2 figli, rimane con un unico figlio e pertanto viola la proprietà degli alberi 2-3. Si rende quindi necessario fondere u con uno dei vicini.
  - L'operazione di fusione potrebbe propagarsi fino alla radice


a b
a
b
c


a
a
b

a
c
a
b
c
d


a c
a
c
d

a
c d
a
b
c
d
e


a
d
a
c
d
e

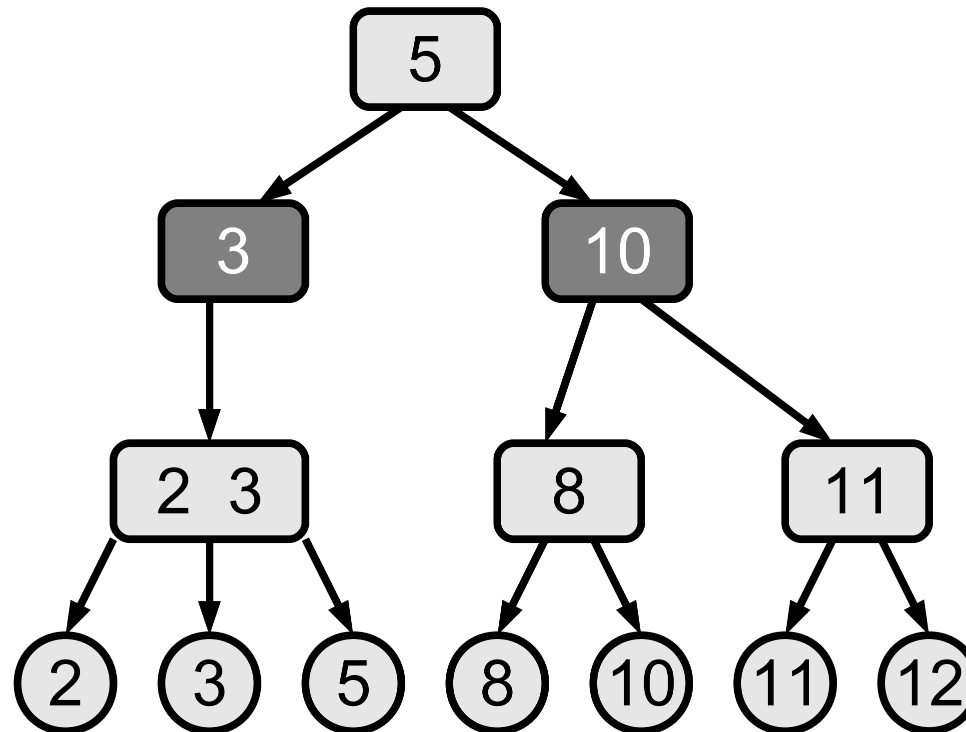
Esempio



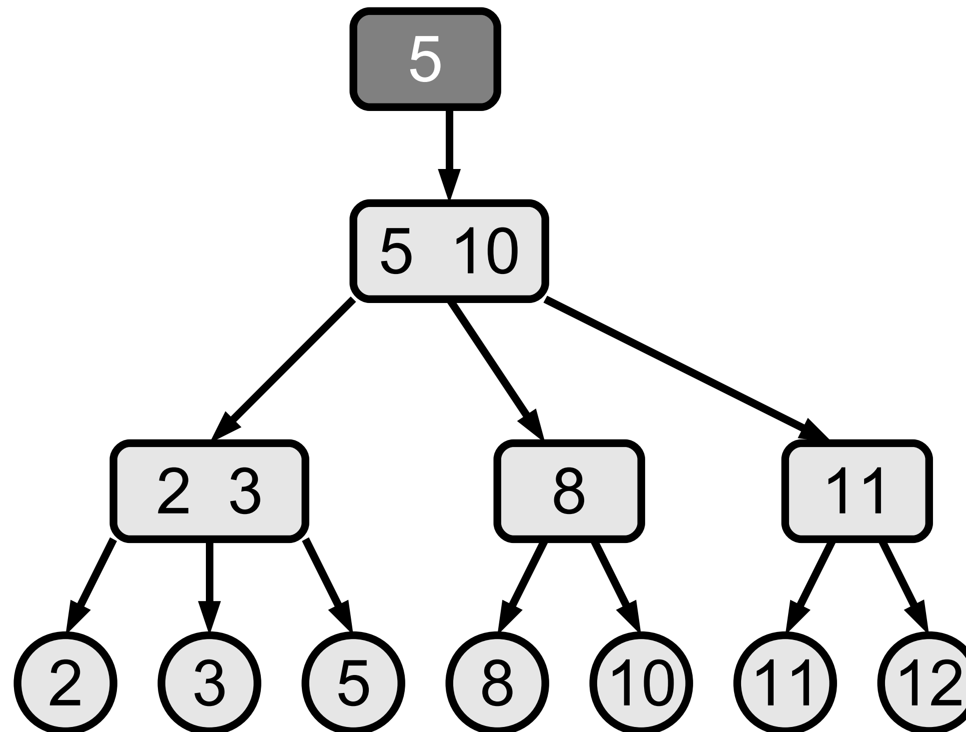
Esempio



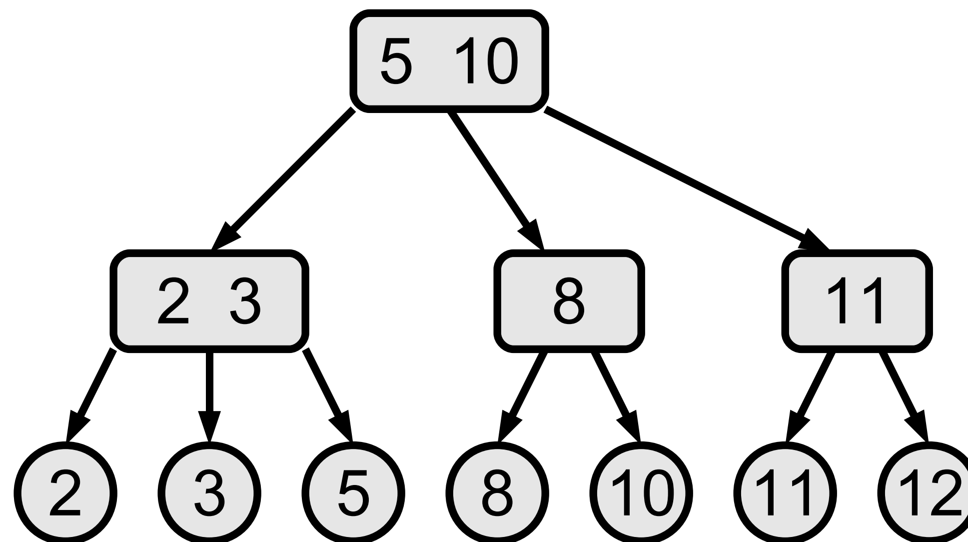
Esempio



Esempio



Esempio

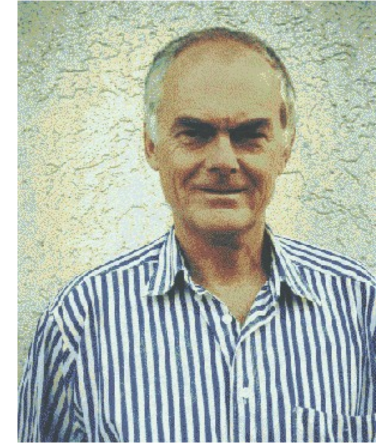


Alberi 2-3: Riassunto

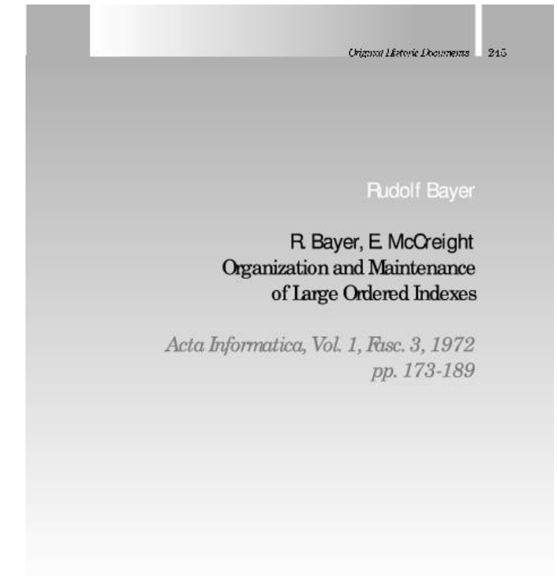
- search(Key k)
  - O(log n) nel caso peggiore
- insert(Key k, Item t)
  - O(log n) nel caso peggiore
- delete(Key k)
  - O(log n) nel caso peggiore

B-Tree

Prof. Rudolf Bayer



- Struttura dati usata in applicazioni che necessitano di gestire insiemi di chiavi ordinate
- Una variante (B+-Tree) è diffusa in:
  - **Filesystem**: btrfs, NTFS, ReiserFS, NSS, XFS, JFS per indicizzare i metadati
  - **Database relazionali**: IBM DB2, Informix, Microsoft SQL Server, Oracle 8, Sybase ASI, PostgreSQL, Firebird, MySQL per indicizzare le tabelle



Original Historic Documents 215

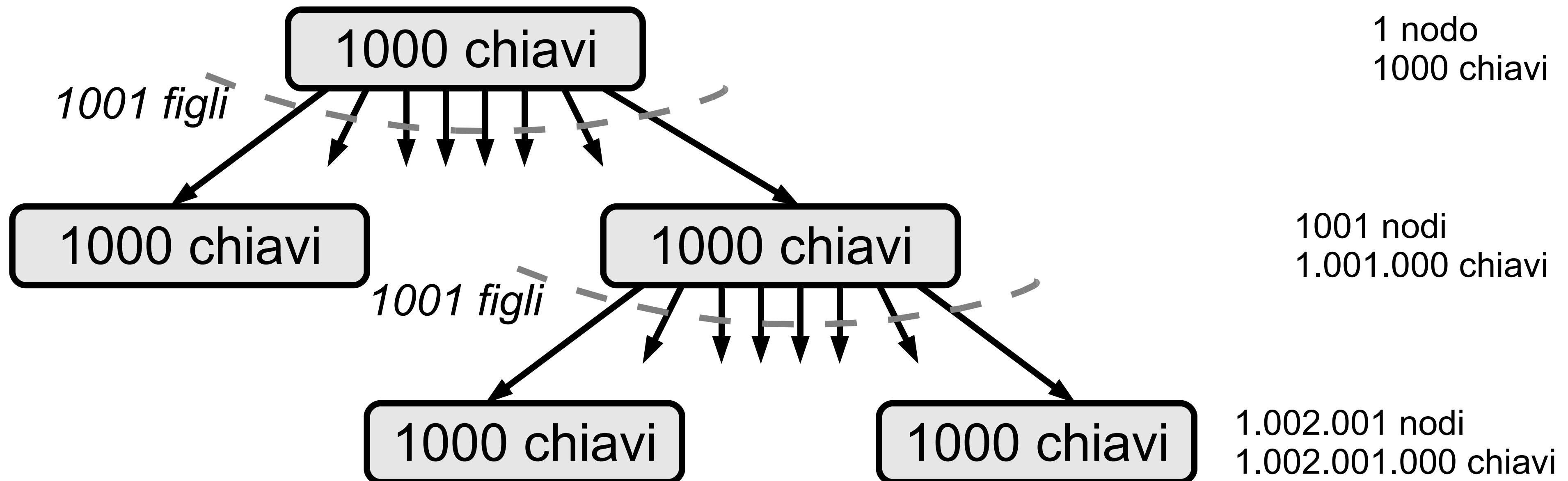
Rudolf Bayer

R. Bayer, E. McCreight
Organization and Maintenance
of Large Ordered Indexes

Acta Informatica, Vol. 1, Fasc. 3, 1972
pp. 173-189

B-Tree

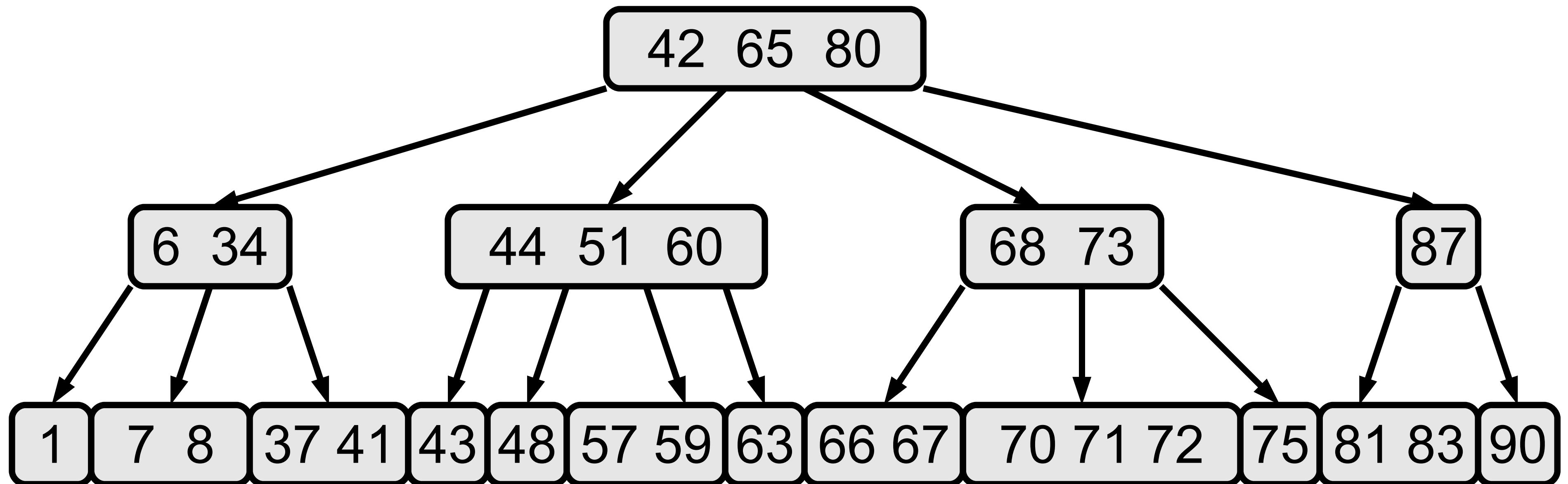
- Poiché ogni nodo può avere un numero elevato di figli, i B-Tree possono efficientemente indicizzare grandi quantità di dati su memoria esterna (disco magnetico), riducendo il numero di operazioni di I/O

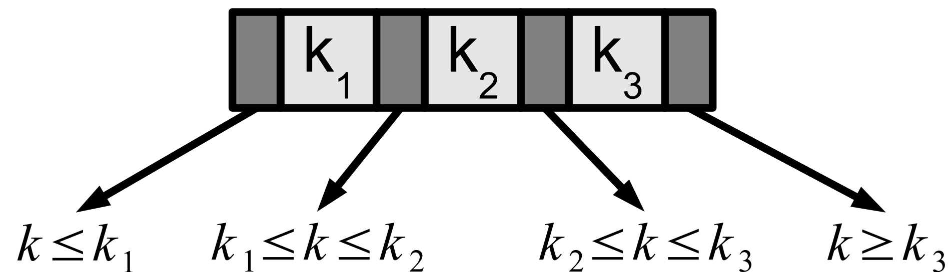


B-Tree

- Un B-Tree di grado t (≥2) ha le seguenti proprietà
  - tutte le foglie hanno la stessa profondità
  - ogni nodo v diverso dalla radice mantiene k(v) chiavi ordinate:
    $$chiave_1(v) \leq chiave_2(v) \leq \ldots \leq chiave_{k(v)}(v)$$
    tali che $t-1 \leq k(v) \leq 2t-1$
  - la radice mantiene almeno 1 ed al più 2t-1 chiavi ordinate
  - ogni nodo interno v ha k(v)+1 figli
  - le chiavi $chiave_i(v)$ separano gli intervalli di chiavi memorizzati in ciascun sottoalbero. Se c_i è una qualunque chiave nell'i-esimo sottoalbero di un nodo v, allora $c_1 \leq chiave_1(v) \leq c_2 \leq chiave_2(v) \leq \ldots \leq c_{k(v)} \leq chiave_{k(v)}(v) \leq c_{k(v)+1}$

Esempio di B-Tree con t=2





Altezza di un B-Tree

- Un B-Tree contenente n chiavi ha altezza

$$h \leq \log_t \frac{n+1}{2}$$

- Dimostrazione
  - Tra tutti i B-Tree di grado t, quello più alto è quello col minor numero di figli per nodo (cioè con t figli)
  - 1 nodo ha profondità zero (la radice)
  - 2 nodi hanno profondità 1
  - 2t nodi hanno profondità 2
  - $2t^2$ nodi hanno profondità 3
  - …
  - $2t^{i-1}$ nodi hanno profondità i

Altezza di un B-Tree

- Numero complessivo di nodi di un B-Tree di altezza h

$$1+\sum_{i=1}^{h} 2t^{i-1}$$

- Poiché ogni nodo (tranne la radice) contiene esattamente t-1 chiavi, possiamo scrivere che il numero n di chiavi soddisfa

$$n \geq 1+(t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$\sum_{i=1}^{h} t^{i-1} = \sum_{i=0}^{h-1} t^{i} = \frac{t^h-1}{t-1}$$

$$= 1+2(t-1)\frac{t^h-1}{t-1} = 2t^h-1$$

Altezza di un B-Tree

- Da $n \geq 2t^h - 1$

  si ricava $t^h \leq \frac{n+1}{2}$

  e passando al logaritmo in base t si conclude

$$h \leq \log_t \frac{n+1}{2}$$

Operazioni sui B-Tree
Ricerca

- È una generalizzazione della ricerca binaria degli ABR
  - Ad ogni passo, si cerca la chiave nel nodo corrente
  - Se la chiave è presente, ci si ferma
  - Se non è presente, si prosegue la ricerca in un sottoalbero che la può contenere

```
algorithm search(radice v di un B-Tree, chiave x) → elem
    i ← 1
    while (i≤k(v) && x>chiave_i(v)) do
        i ← i+1;
    endwhile
    if (i≤k(v) && x==chiave_i(v)) then
        return elem_i(v);
    else
        if (v è una foglia) then
            return null
        else
            return search(i-esimo figlio di v, x);
        endif
    endif
```

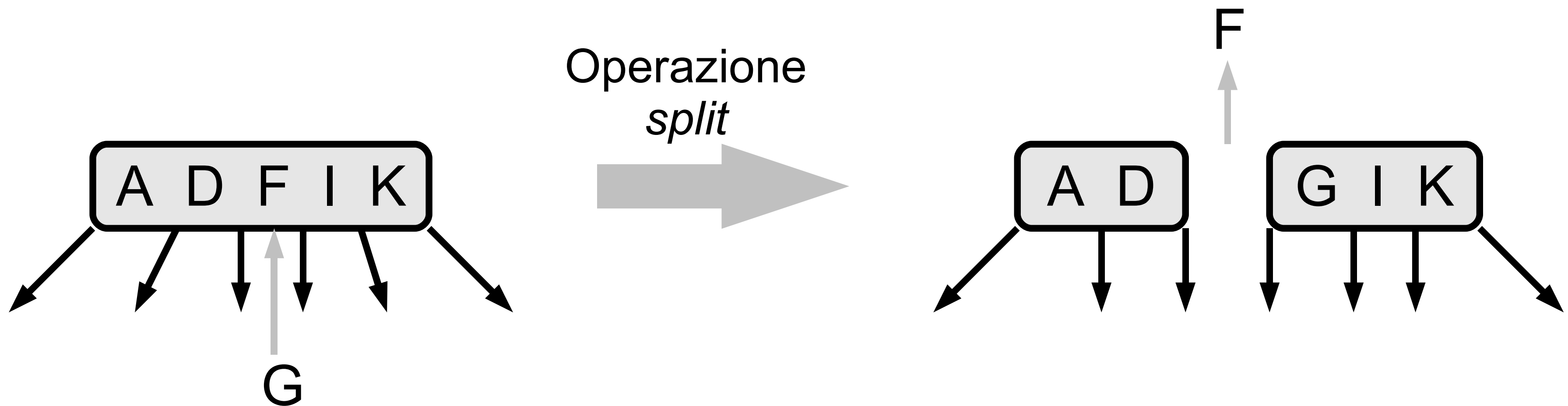
Operazioni sui B-Tree
Ricerca

- Costo computazionale
  - I nodi visitati sono $O(\log_t n)$
  - Ciascuna visita costa $O(t)$ se facciamo una scansione lineare delle chiavi;
  - Totale $O(t \log_t n)$
    - Poiché le chiavi sono ordinate in ogni nodo, possiamo effettuare una ricerca binaria in tempo $O(\log t)$ anziché $O(t)$. Quindi il costo totale scende a $O(\log t \log_t n) = O(\log n)$ (usando il cambiamento di base dei logaritmi)

Operazioni sui B-Tree Inserimento

- Si individua (mediante una ricerca) una foglia f in cui inserire la chiave k
- Se la foglia non è piena (ha meno di 2t-1 chiavi) si inserisce k nella posizione appropriata e l'inserimento termina
- Se la foglia è piena (ha 2t-1 chiavi) allora
  - Il nodo f viene diviso in due (operazione *split*), e la sua t-esima chiave viene spostata nel padre di f
  - Se il padre di f aveva già 2t-1 chiavi, occorre dividerlo allo stesso modo, e proseguire verso la radice
  - Nel caso peggiore (quando tutto il cammino da f alla radice è composto da nodi pieni) gli split successivi producono una nuova radice

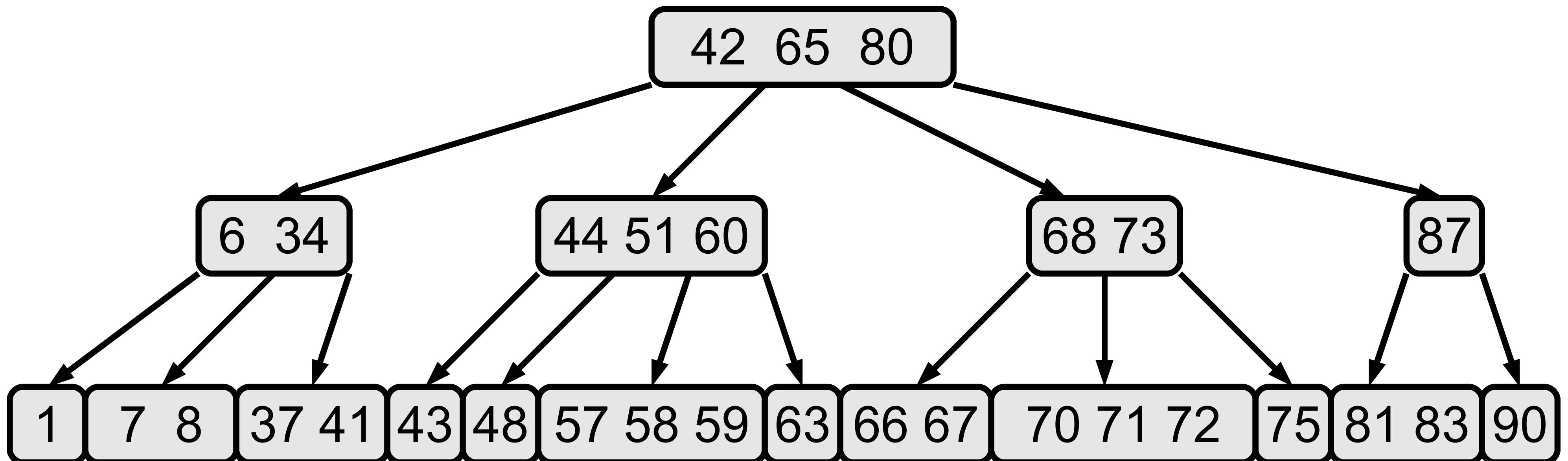
Operazioni sui B-Tree
Inserimento



- Costo computazionale
  - I nodi visitati sono $O(\log_t n)$
  - Ciascuna visita costa $O(t)$ nel caso peggiore, a causa delle operazioni di split
  - Totale $O(t \log_t n)$

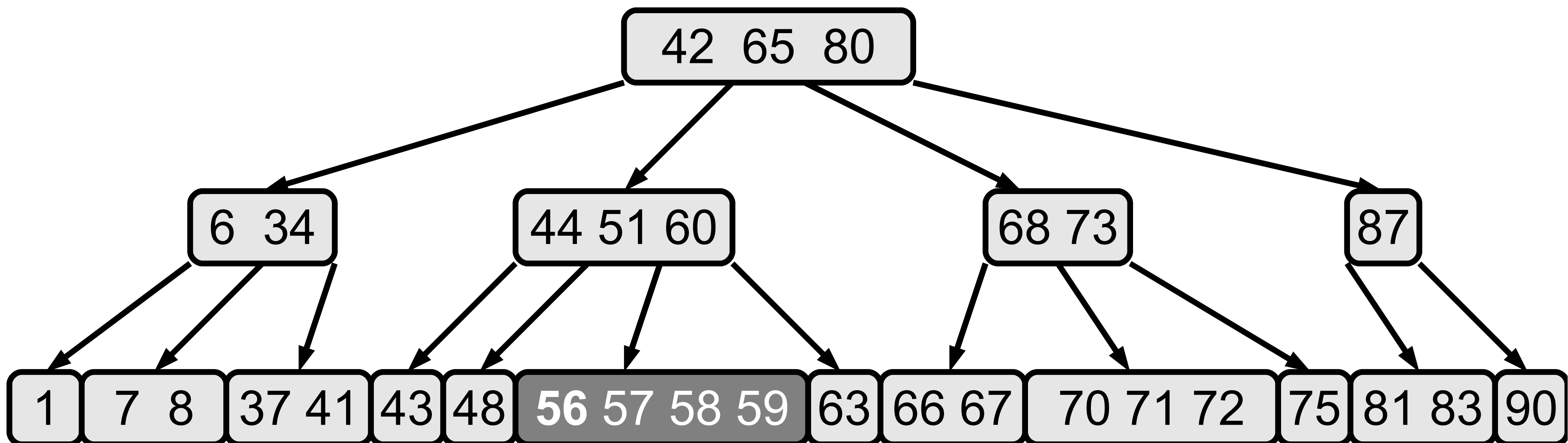
Operazioni sui B-Tree
Inserimento

- Esempio (t=2)

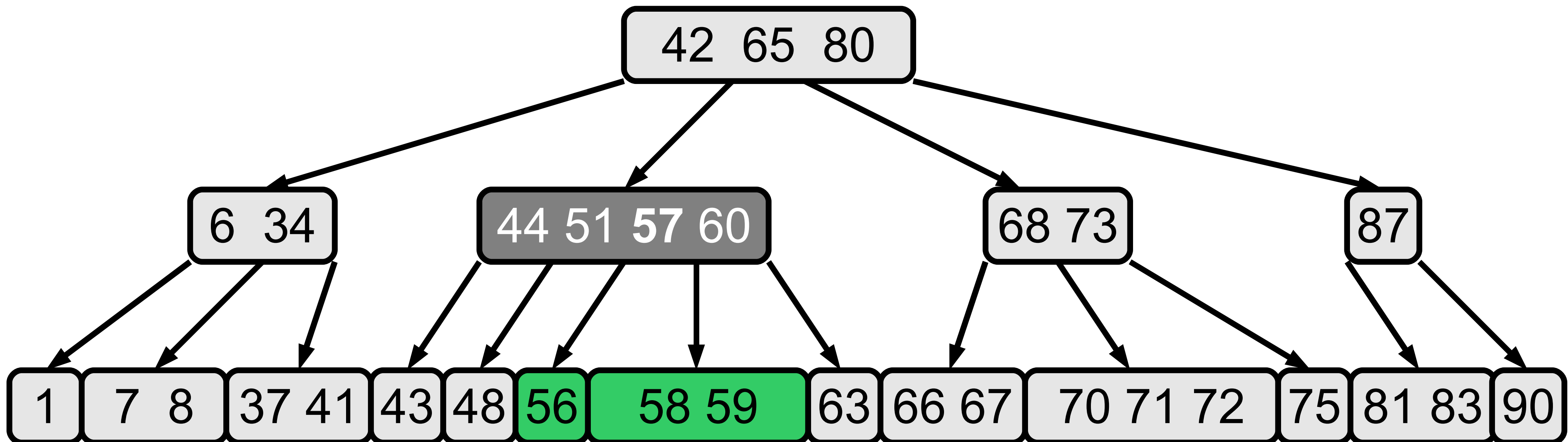


Operazioni sui B-Tree
Inserimento

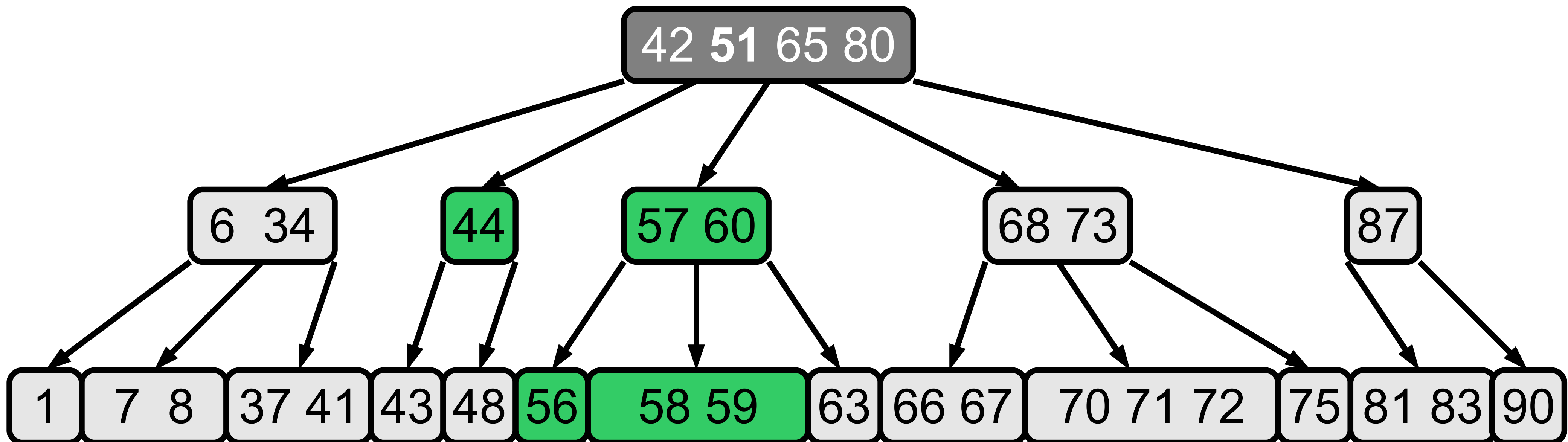
- Inserisco 56



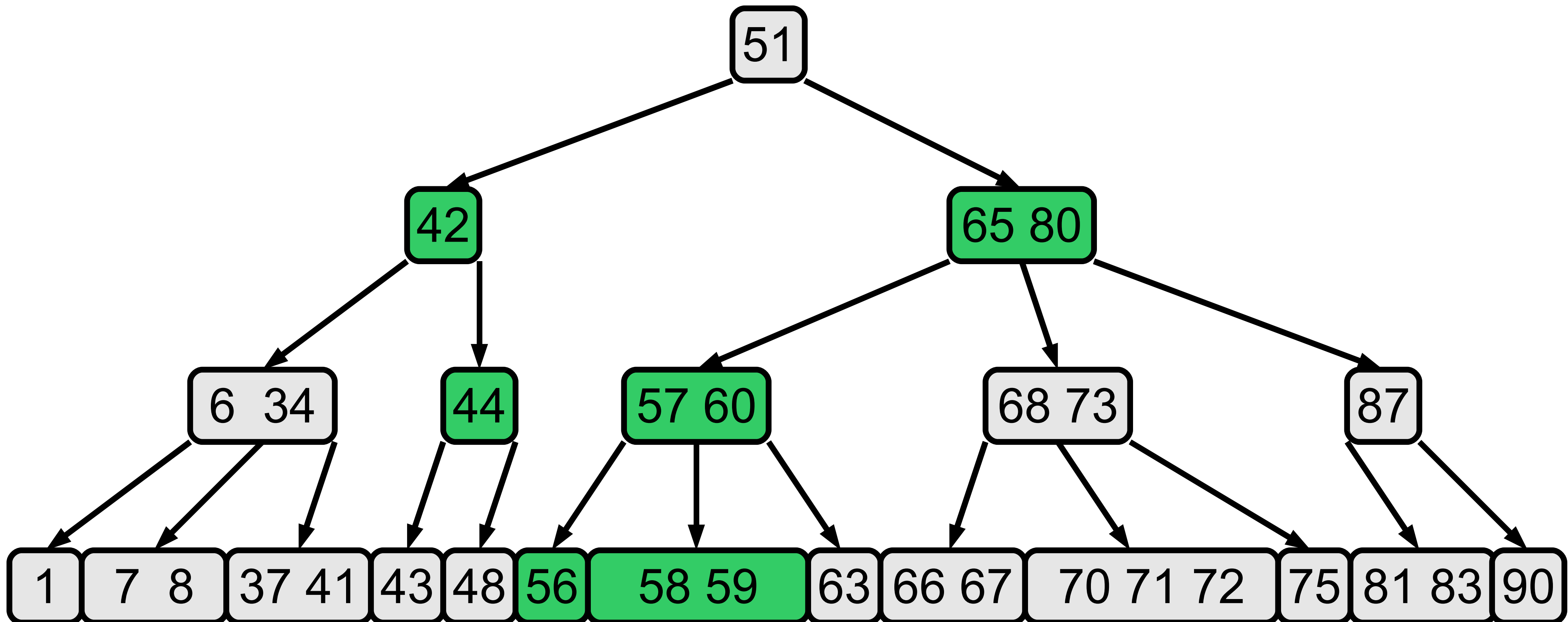
Operazioni sui B-Tree
Inserimento



Operazioni sui B-Tree
Inserimento



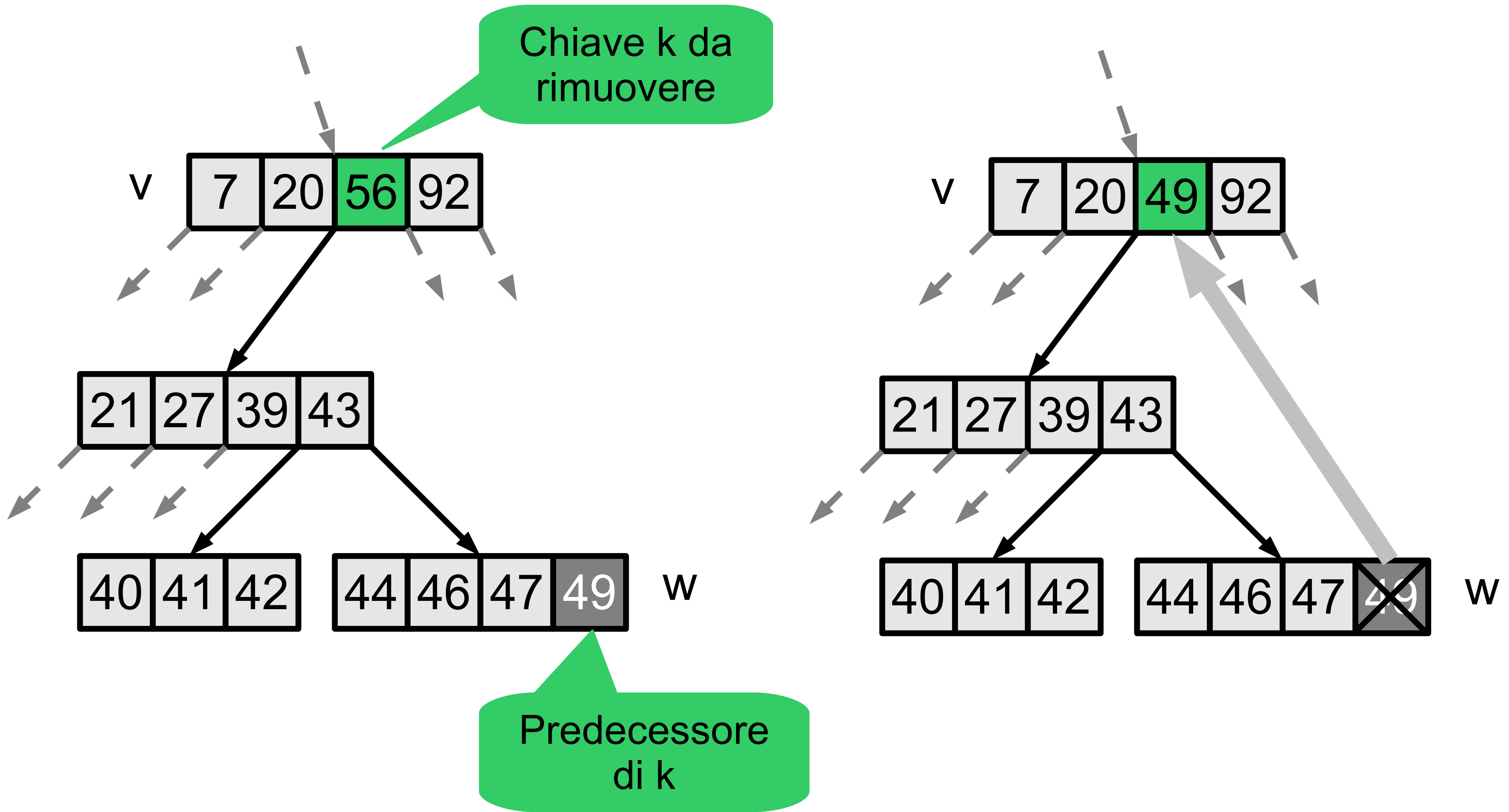
Operazioni sui B-Tree
Inserimento



Operazioni sui B-Tree
Cancellazione

- Se la chiave k da rimuovere si trova in un nodo v che **non** è una foglia
    - Individua il nodo w contenente il valore predecessore di k
    - Sposta la chiave massima in w al posto della chiave k da rimuovere
    - Ci si riconduce al caso successivo rimuovendo la max chiave da w
- Se la chiave k da rimuovere si trova in una foglia v
    - Se la foglia contiene più di t-1 chiavi, si rimuove k e si termina
    - Se la foglia contiene t-1 chiavi, rimuovendo k si scende al di sotto del limite minimo. Esaminiamo la situazione dei fratelli adiacenti
        - Se almeno uno dei fratelli adiacenti ha >t-1 chiavi, redistribuiamo le chiavi
        - Se nessuno dei fratelli adiacenti ha >t-1 chiavi, effettuiamo una operazione di *fusione*

Operazioni sui B-Tree Cancellazione da nodo interno

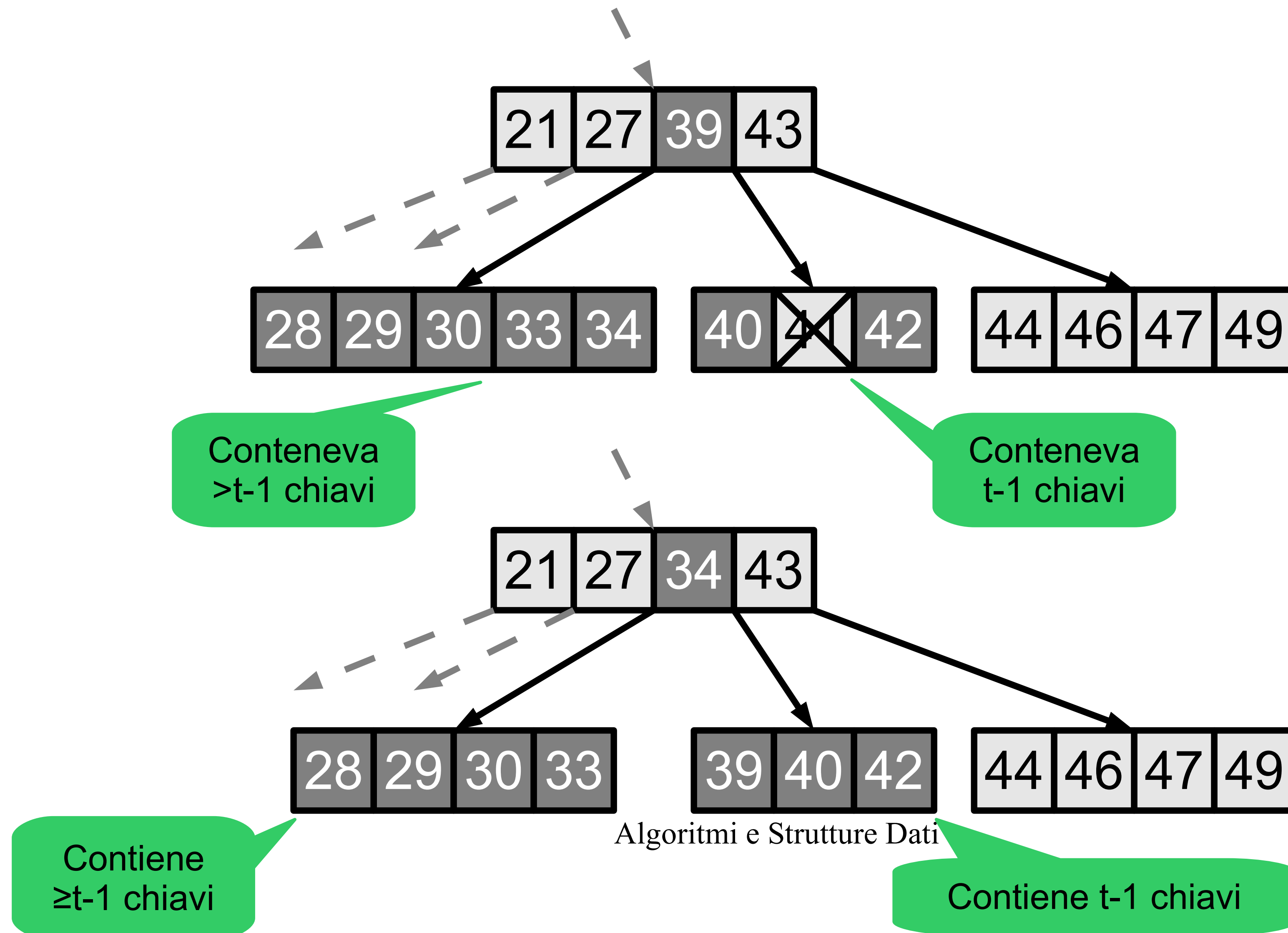


Operazioni sui B-Tree Cancellazione da una foglia

- Primo caso: la foglia contiene > t-1 chiavi
  - In tal caso basta rimuovere la chiave dalla foglia, e la procedura di cancellazione termina (la foglia ora contiene ≥t-1 chiavi)
- Secondo caso: la foglia contiene <u>esattamente</u> t-1 chiavi. Ci sono due possibilità
  - Redistribuire le chiavi con un fratello adiacente
  - Fondere la foglia con un fratello adiacente

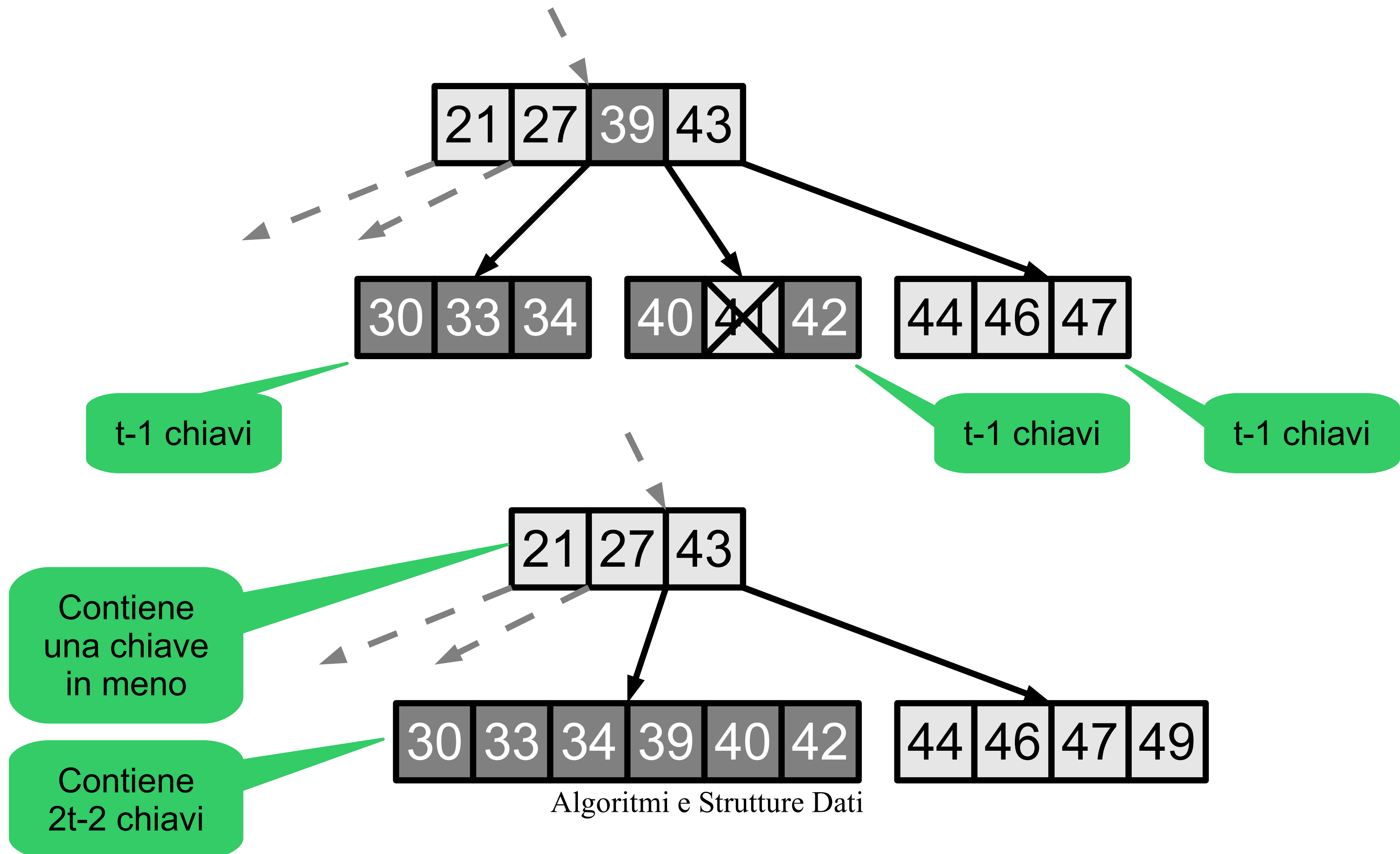
Operazioni sui B-Tree
Rimozione da foglia quasi vuota—caso 1

- Consideriamo un frammento di B-Tree con t=4



Operazioni sui B-Tree
Rimozione da foglia quasi vuota—caso 2 (fusione)

- Consideriamo un frammento di B-Tree con t=4



Riepilogo

	search	insert	delete
Array ordinato	O(log n)	O(n)	O(n)
Lista non ordinata	O(n)	O(1)	O(n)
ABR	O(h)	O(h)	O(h)
Albero AVL	O(log n)	O(log n)	O(log n)
Albero 2-3	O(log n)	O(log n)	O(log n)
B-Tree	$O(\log t \log_t n) = O(\log n)$	$O(t \log_t n)$	$O(t \log_t n)$

NB: tutti i costi si riferiscono al caso pessimo