Alberi binari di ricerca

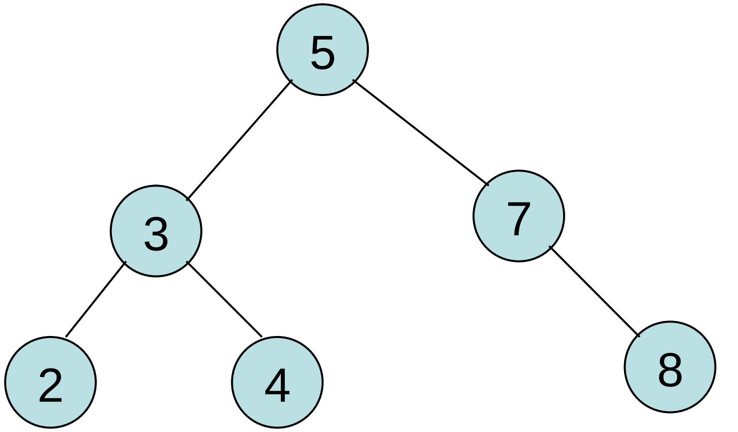
- ✓Definizione
- ✓Visita dell’albero inorder
- ✓Ricerca
- ✓Ricerca minimo, massimo e successore.
- ✓Inserimento ed eliminazione di un nodo
- ✓Problema del bilanciamento dell’albero

Albero binario

- Un albero binario è un albero dove ogni nodo ha al **massimo due figli**.
- Tutti i nodi tranne la radice ha un nodo padre.
- Le foglie dell'albero non hanno figli.
- In aggiunta, ogni nodo ha una chiave.

Per rappresentare un albero binario si possono usare dei puntatori. Ogni nodo ha un puntatore al padre, al figlio sinistro e a quello destro. Inoltre, ad ogni nodo ha associato una chiave.

Se un figlio o il padre è mancante, il relativo campo è uguale a NIL.

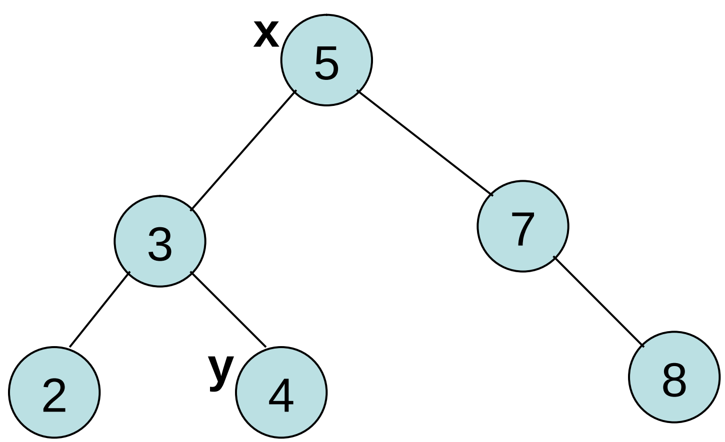


Albero binario di ricerca

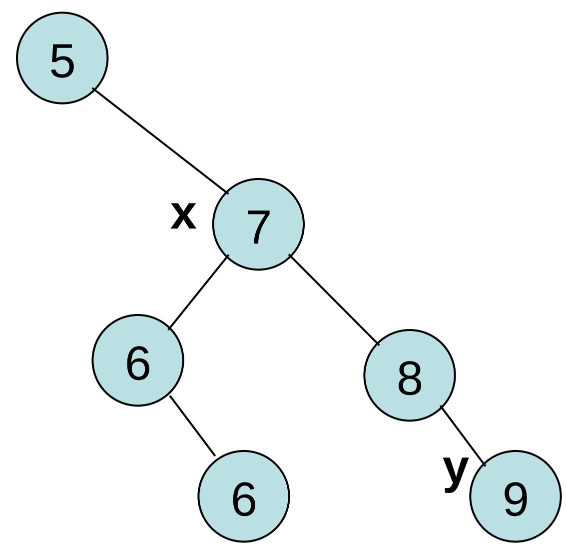
Definizione:

Sia x un nodo di un albero binario di ricerca.

1. Se **y** è un nodo appartenente al **sottoalbero sinistro di x** allora si ha **key[y] ≤ key[x]**.
2. Se **y** è un nodo appartenente al **sottoalbero destro di x** allora si ha **key[y] ≥ key[x]**.



key[y] ≤ key[x]



key[x] ≤ key[y]

Visita Inorder

INORDER-TREE-WALK(x)
1. ***if*** *x ≠ NIL*
2. ***then***
3. INORDER-TREE-WALK(left[x])
4. ***print*** *key[x]*
5. INORDER-TREE-WALK(right[x])

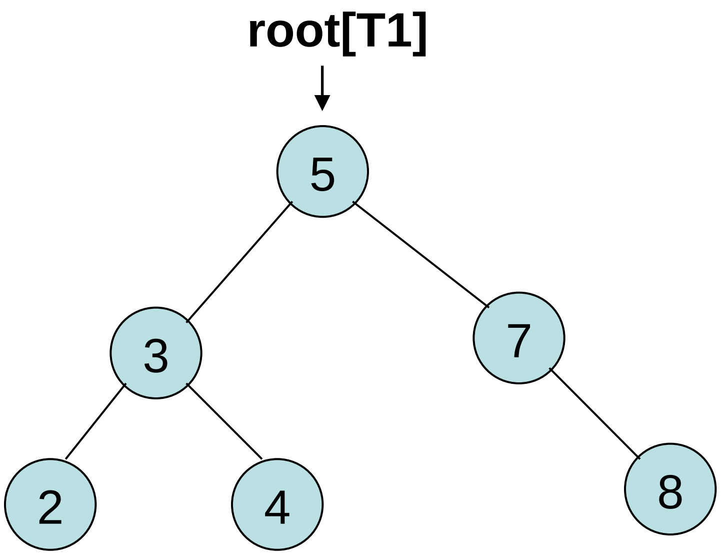
E’ un **algoritmo ricorsivo**!

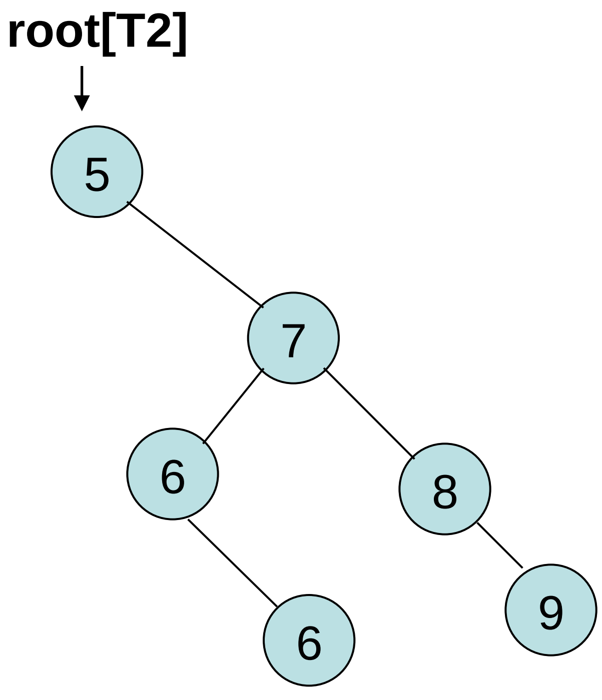
Richiede tempo **Θ(n)** con un albero di n nodi.

Nel caso sia dato in ingresso un nodo x di un albero binario di ricerca vengono **stampati in ordine crescente le chiavi del sottoalbero** che ha come radice x stesso.

Questo è dovuto al fatto che la chiave del nodo x viene stampato dopo le chiavi del suo sottoalbero sinistro e prima di quelle del suo sottoalbero destro.

Visita Inorder





INORDER-TREE-WALK(root[T1]):

2, 3, 4, 5, 7, 8

INORDER-TREE-WALK(root[T2]):

5, 6, 6, 7, 8, 9

L’operazione di ricerca

TREE-SEARCH(x,k)

1. ***if*** *x = NIL or k = key[x]*
2. ***then return*** *x*
3. ***if*** *k < key[x]*
4. ***then return*** TREE-SEARCH(left[x],k)
5. ***else return*** TREE-SEARCH(right[x],k)

Dato in ingresso il puntatore alla radice dell’albero e una chiave, l’algoritmo restituisce il nodo con chiave uguale a k oppure NIL se non esiste.

L’algoritmo discende l’albero con una chiamata ricorsiva **sfruttando le proprietà dell’albero** binario di ricerca.

Quindi non è necessario vedere tutti i nodi ma solo **O(h)**, pari all’altezza h dell’albero.

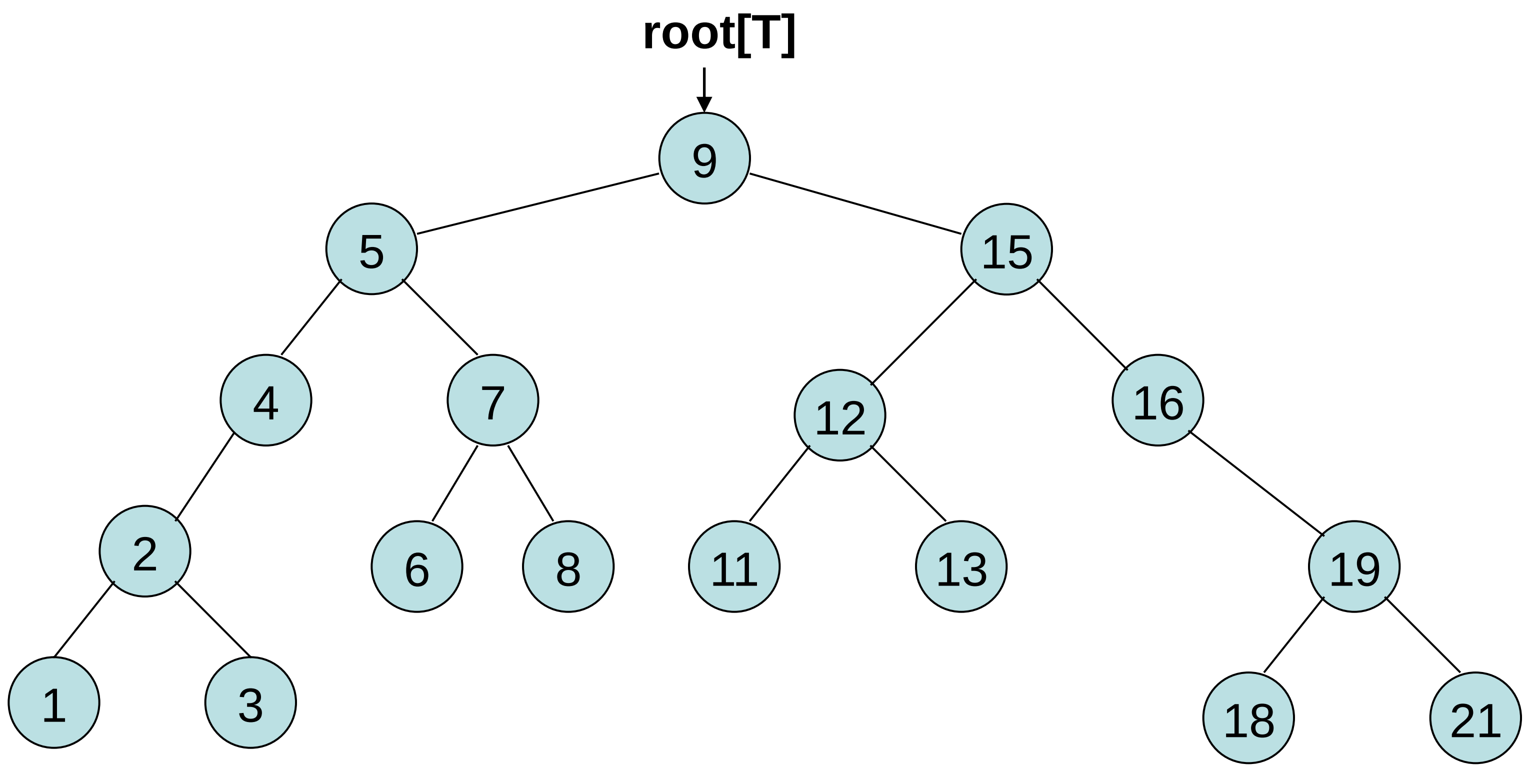
L’operazione di ricerca

```
INTERACTIVE-TREE-SEARCH(x,k)
    while x ≠ NIL or k ≠ key[x]
        do  if k < key[x]
                then x ← left[x]
                else x ← right[x]
    return x
```

Lo stesso algoritmo può essere implementato **usando un ciclo while**. In genere, è più efficiente.

Dunque, **il tempo per la ricerca** di un nodo è pari a **O(h)**, con all’altezza h dell’albero.

L’operazione di ricerca



TREE-SEARCH(root[T],18):

9 → 15 → 16 → 19 → **18** Trovato!

TREE-SEARCH(root[T],10):

9 → 15 → 12 → 11 → **NIL** NON Trovato!

Minimo

TREE-MINIMUM(x)

1. **while** *left[x] ≠ NIL*

2. **do** x ← *left[x]*

3. **return** x

Si utilizza le proprietà dell'albero binario di ricerca.
Partendo dal nodo x si ha che:

a. Se **x ha un figlio sinistro** allora il minimo è nel sottoalbero sinistro, poiché ogni nodo y_s di questo è tale per cui $key[y_s] \leq key[x]$.

b. Se **x non ha un figlio sinistro** allora ogni nodo y_d nel sottoalbero destro è tale per cui $key[x] \leq key[y_d]$. Quindi, x è il minimo del sottoalbero con radice x.

Massimo

TREE-MAXIMUM(x)

1. ***while*** *right[x]* ≠ *NIL*

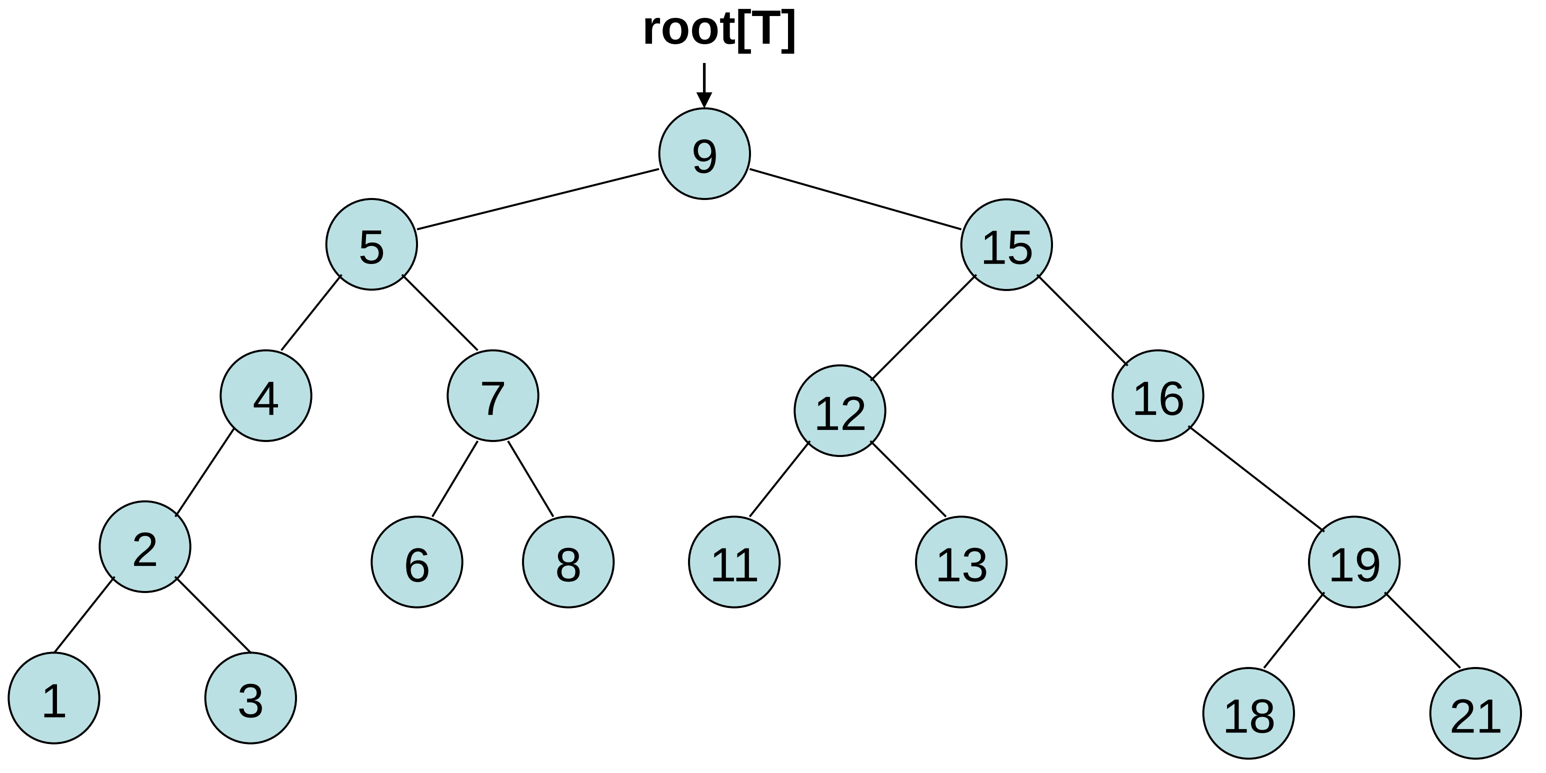
2. ***do*** x ← *right[x]*

3. **return** x

L'algoritmo **risulta simmetrico** rispetto a TREE-MINIMUM(x).

Il tempo di esecuzione per trovare il massimo o il minimo in un albero binario di ricerca è al massimo pari all'altezza dell'albero, ossia **O(h)**.

Minimo e Massimo



TREE-MINIMUM(root[T]):

9 → 5 → 4 → 2 → **1** <u>Trovato!</u>

TREE-MAXIMUM(root[T]):

9 → 15 → 16 → 19 → **21** <u>Trovato!</u>

Successore

Supponiamo che nel nostro albero ci siano solo <u>chiavi distinte</u>. Il **successore di un nodo x** è il nodo y con la chiave più piccola maggiore di key[x]:

succ(key[x]) = min {y in T: key[x] < key[y]}.

Sfruttando la struttura dell'albero binario di ricerca è possibile trovare il successore di un nodo senza dover confrontare le chiavi nei nodi.

Nel seguente algoritmo restituisce il successore di un nodo x oppure NIL se x è il nodo con la chiave più grande.

Successore

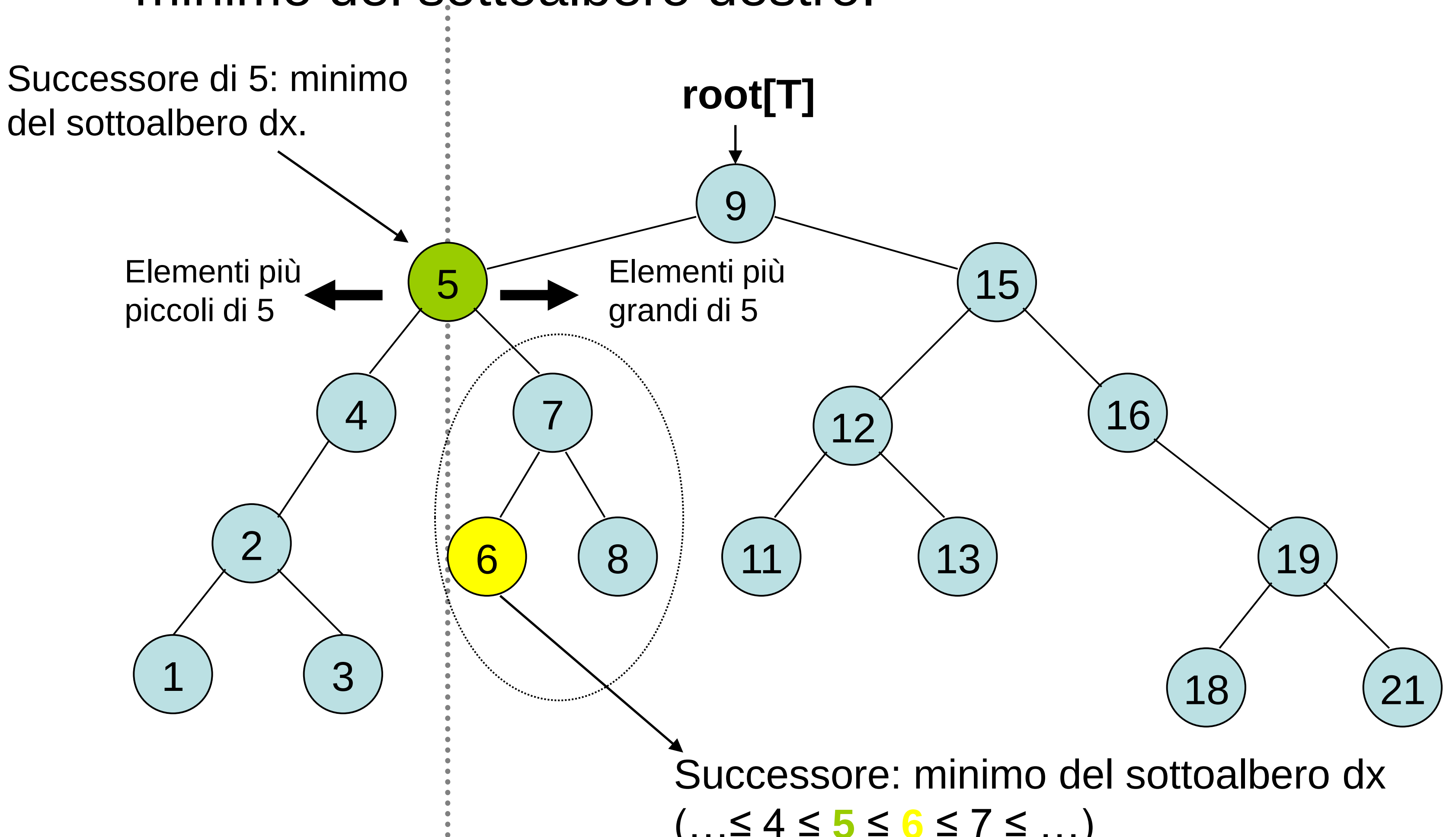
```
TREE-SUCCESSOR(x)
    if right[x] ≠ NIL                    // esiste il sottoalbero dx?
       then return TREE-MINIMUM(right[x])
    y ← p[x]                             // il sottoalbero dx non esiste
    while y ≠ NIL and x = right[y]
          do x ← y
             y ← p[y]
    return y                             // se y = NIL, x non ha un successore
```

a. Se **il nodo ha un figlio destro**, il successore di x è il minimo del sottoalbero destro.

b. Se **il nodo non ha un figlio destro**, si risale l’albero finché il nodo di provenienza sta a sinistra. In questo caso il nodo di partenza risulta essere il massimo del sottoalbero sinistro di y. Quindi, y è il suo successore.

Successore

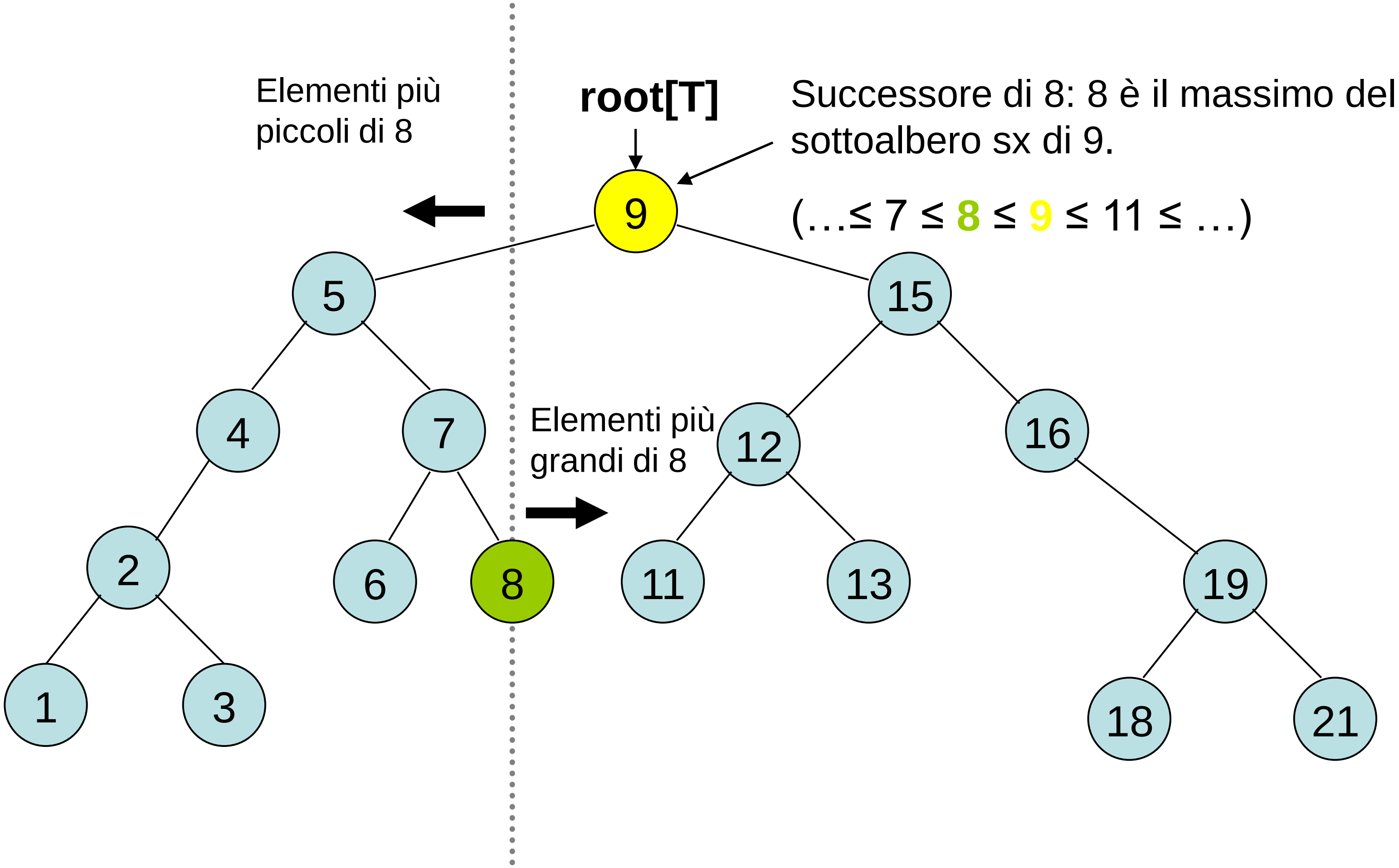
a. Se **il nodo x ha un figlio destro**, il successore di x è il minimo del sottoalbero destro.



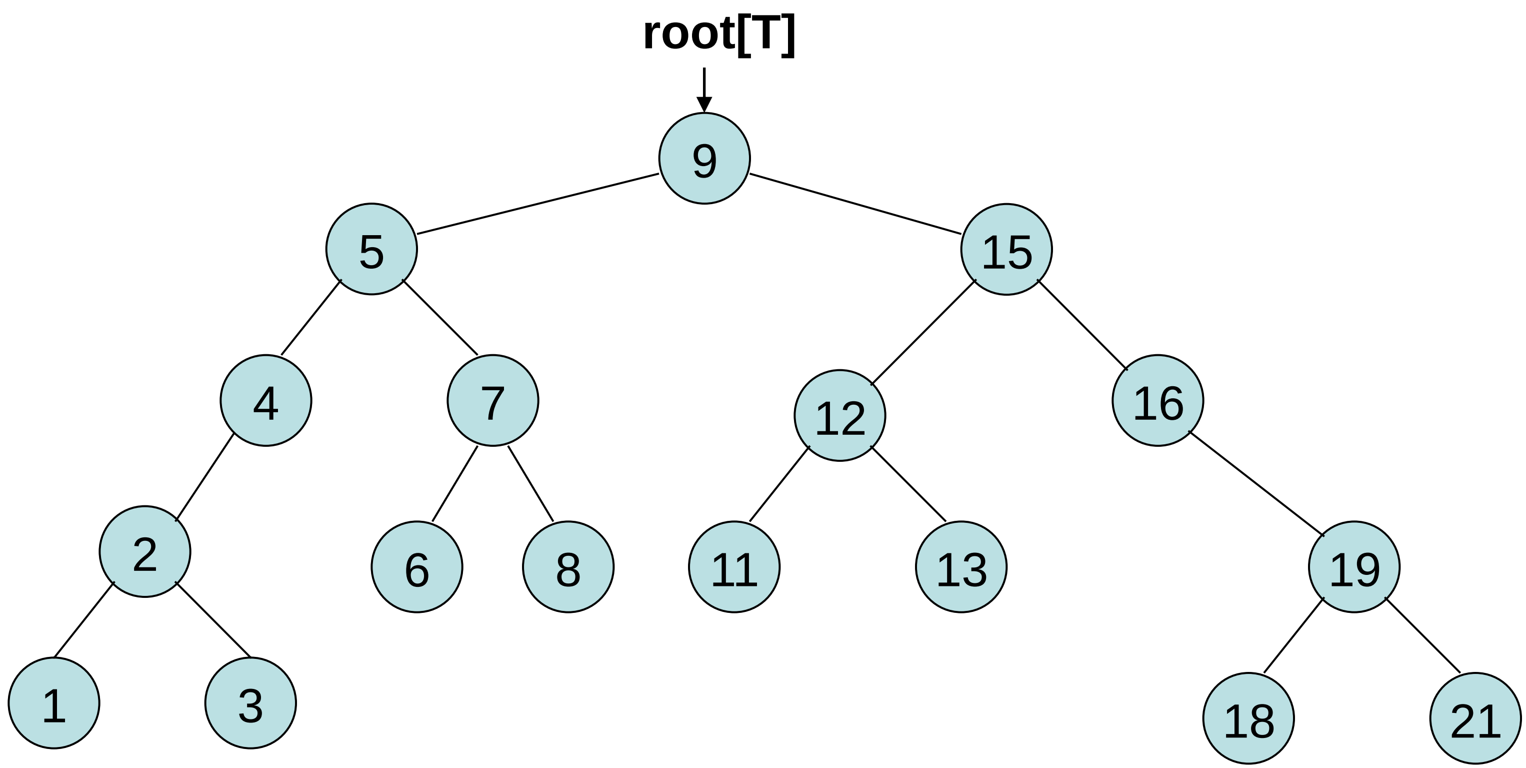
Successore: minimo del sottoalbero dx
(…≤ 4 ≤ 5 ≤ 6 ≤ 7 ≤ …)

Successore

a. Se **il nodo x non ha un figlio destro**: se y è il suo successore, x è il massimo del sottoalbero sx di y.



Successore



TREE-SUCCESSOR(x con key[x] = 5):

5 → 7 → **6**

Trovato nel sottoalbero destro.

TREE-SUCCESSOR(x con key[x] = 8):

8 → 7 → 5 → **9**

Trovato risalendo l’albero.

Successore

Il tempo necessario per **trovare il successore** è pari a **O(h)**, dove h è l'altezza dell'albero.

Si effettua un cammino non più lungo della distanza massima tra la radice e una foglia.

Il nodo y che precede x nella visita inorder è il nodo y con la chiave più grande minore di key[x] (key[y] ≤ key[x]).

Per trovare il nodo che precede nella visita inorder, si utilizza un algoritmo simmetrico **TREE-PREDECESSOR(x)**.

TREE-PREDECESSOR(x) richiede tempo **O(h)** per motivi analoghi.

Inserimento e rimozione

- Quando si inserisce o si rimuove un elemento **la struttura dell’albero cambia**.
- L’albero modificato deve **mantenere le proprietà** di un albero binario di ricerca.
- La struttura dell’albero varia a seconda della sequenza di dati da inserire o rimuovere.
- L’**inserimento** risulta essere un’operazione **immediata**.
- La **rimozione** di un elemento è **più complicata**, proprio perché bisogna essere certi che l’albero rimanga un albero binario di ricerca.

Inserimento

```
TREE-INSERT(T,z)
y ← NIL                                   // padre
x ← root[T]                               // figlio
while x ≠ NIL                             // while finché si raggiunge la
                                       posizione dove inserire z (x = NIL)
    do y ← x                              // memorizza il padre
        if key[z] < key [x]               // scendi nel figlio giusto
          then x ← left[x]
          else x ← right[x]
p[z] ← y                                  // inserisci z come figlio di y
if y = NIL                                // y = NIL albero vuoto
   then root[T] ← z
   else if key[z] < key [y]               // y punta a z
      then left[y] ← z                    // z figlio sinistro key[z] < key [y]
      else  right[y] ← z                   // z figlio destro key[x] ≤ key [y]
```

Inserimento

Per inserire z si usano **due puntatori y e x**.

Il puntatore a x scende l'albero , mentre y punta al padre di x.

Nel **ciclo while** i due puntatori (x e y) scendono l'albero. x scende al figlio sinistro o destro a seconda dell'esito del confronto di key[z] con key[x]. Ci si ferma quando x = NIL e x occupa la posizione in cui z verrà inserito.

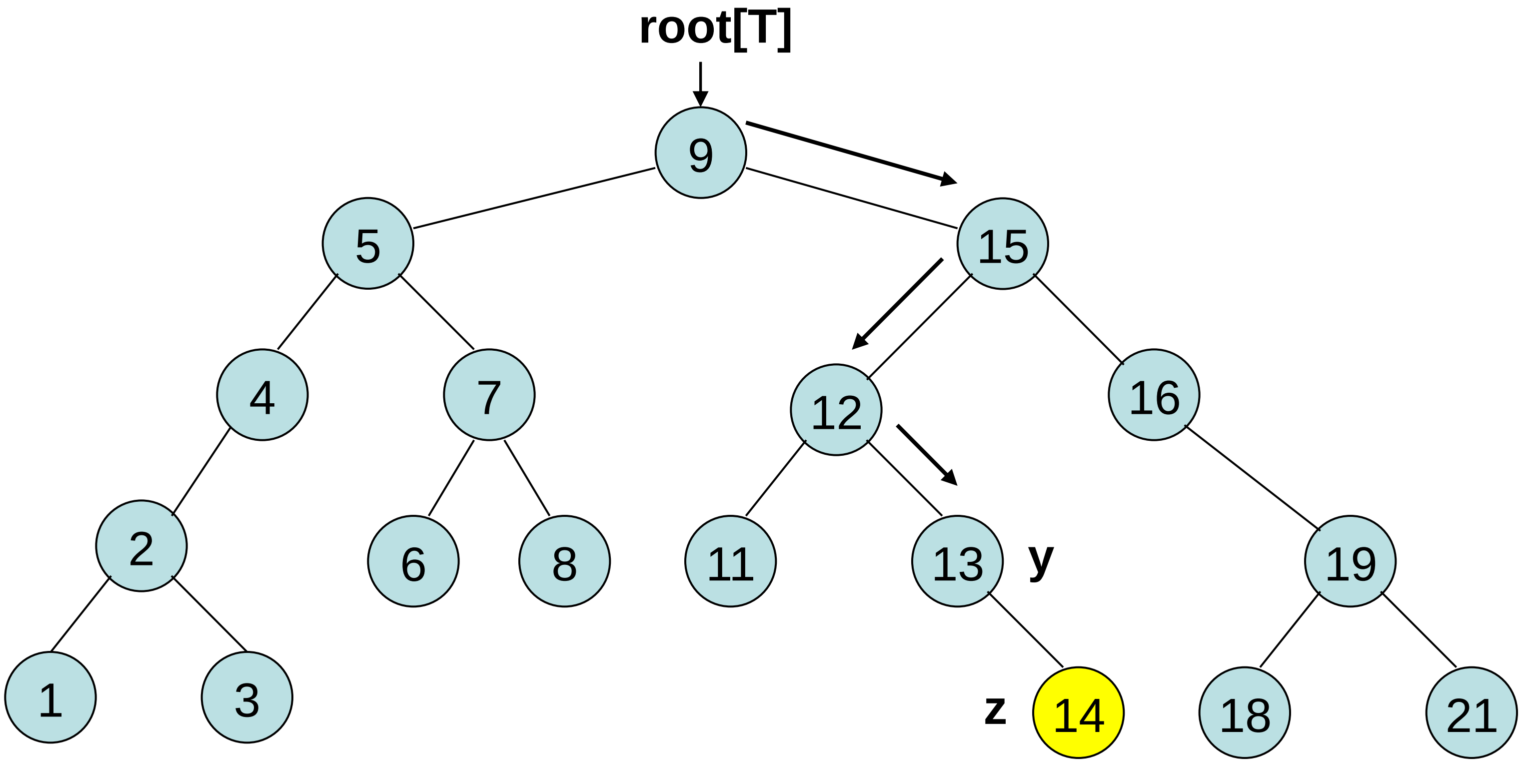
Nelle **linee 8-13** viene effettuato l'effettivo **inserimento**.

Il tempo necessario per l'inserimento è **O(h)**, ossia non più del cammino massimo tra la radice e una foglia (cioè h l'altezza).

Inserimento

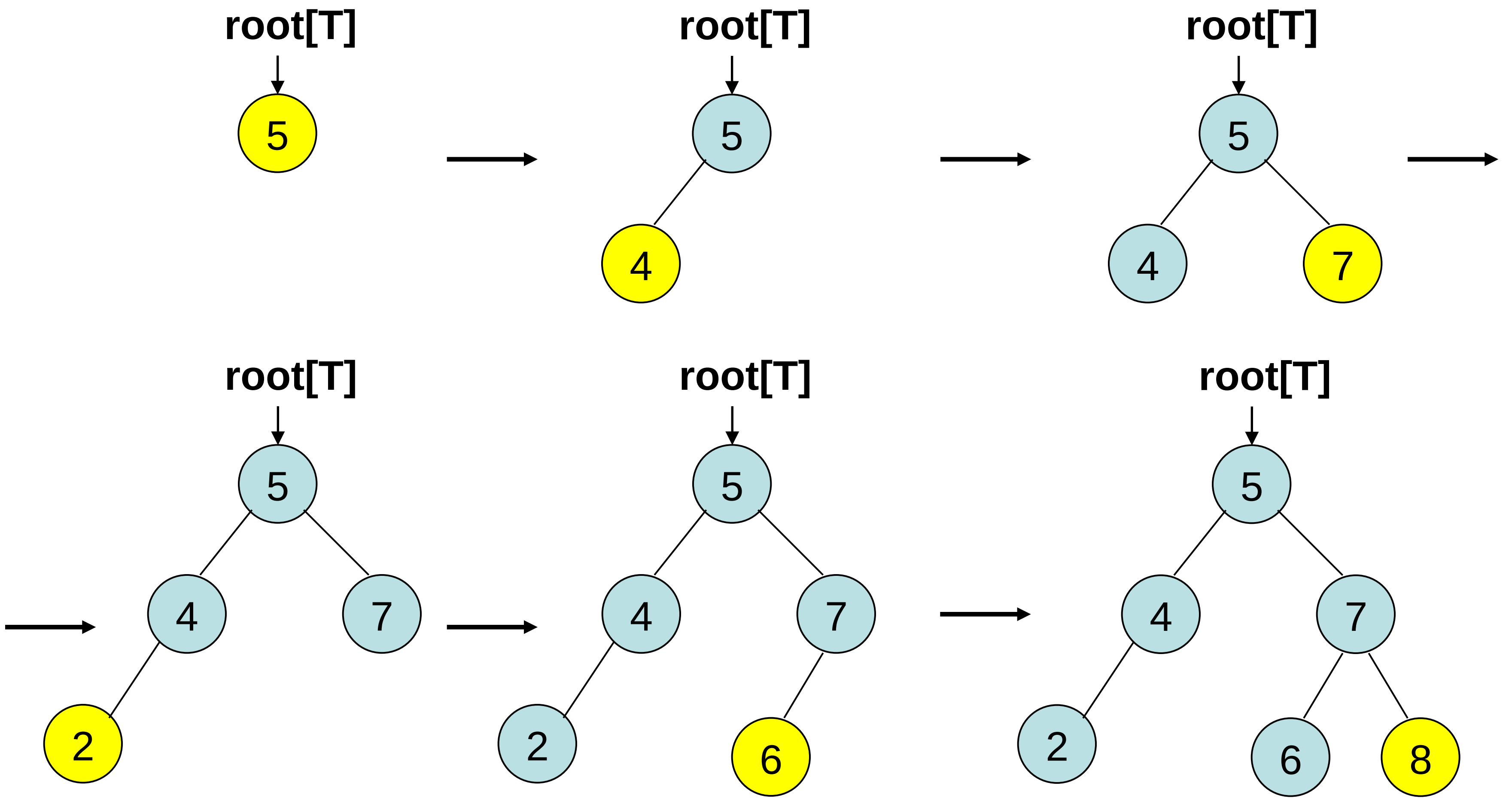
TREE-INSERT(T, z):

z con key[z] = 14



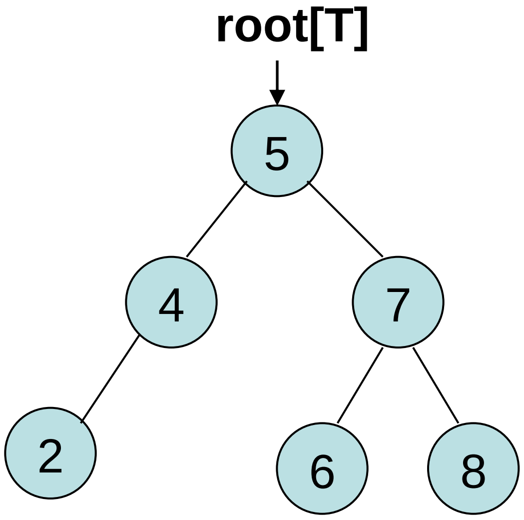
Inserimento

TREE-INSERT(T, z)
sequenza <5, 4, 7, 2, 6, 8>



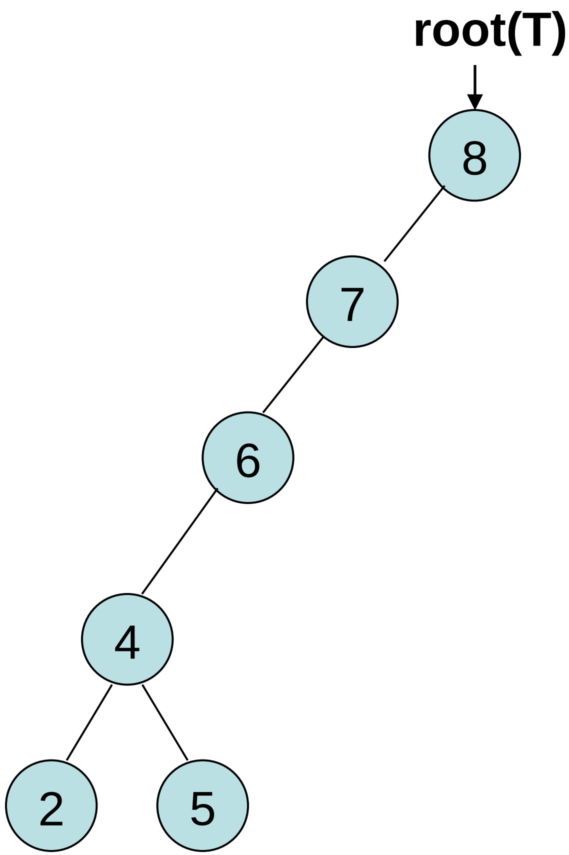
Inserimento

TREE-INSERT(T, z)
sequenza <5, 4, 7, 2, 6, 8>



Inserimento

TREE-INSERT(T, z)
sequenza <8, 7, 6, 4, 5, 2>



La struttura dell'albero risulta diversa a seconda della sequenza di inserimento!

Rimozione

```
TREE-DELETE(T,z)
if left[z] = NIL or right[z] = NIL
    then y ← z                          // z ha 0 o 1 figlio
    else y ← TREE-SUCCESSOR(z)          // z ha due figli, trova succ(z)
if left[y] ≠ NIL                        // x punta ad eventuale
    then x ← left[y]                    // unico figlio di y, altrimenti NIL
    else x ← right[y]
if x ≠ NIL                              // se y ha il figlio
    then p[x] ← p[y]                    // taglia fuori y
if p[y] = NIL
    then root[T] ← x                    // se y è la radice
    else if y = left[p[y]]              // altrimenti
            then left[p[y]] ← x         // completa eliminazione di y
            else right[p[y]] ← x
if  y ≠ z                               // se y è il successore
     then key[z] ← key[y]               // copia y in z
          copia anche altri attributi di y in z
return y
```

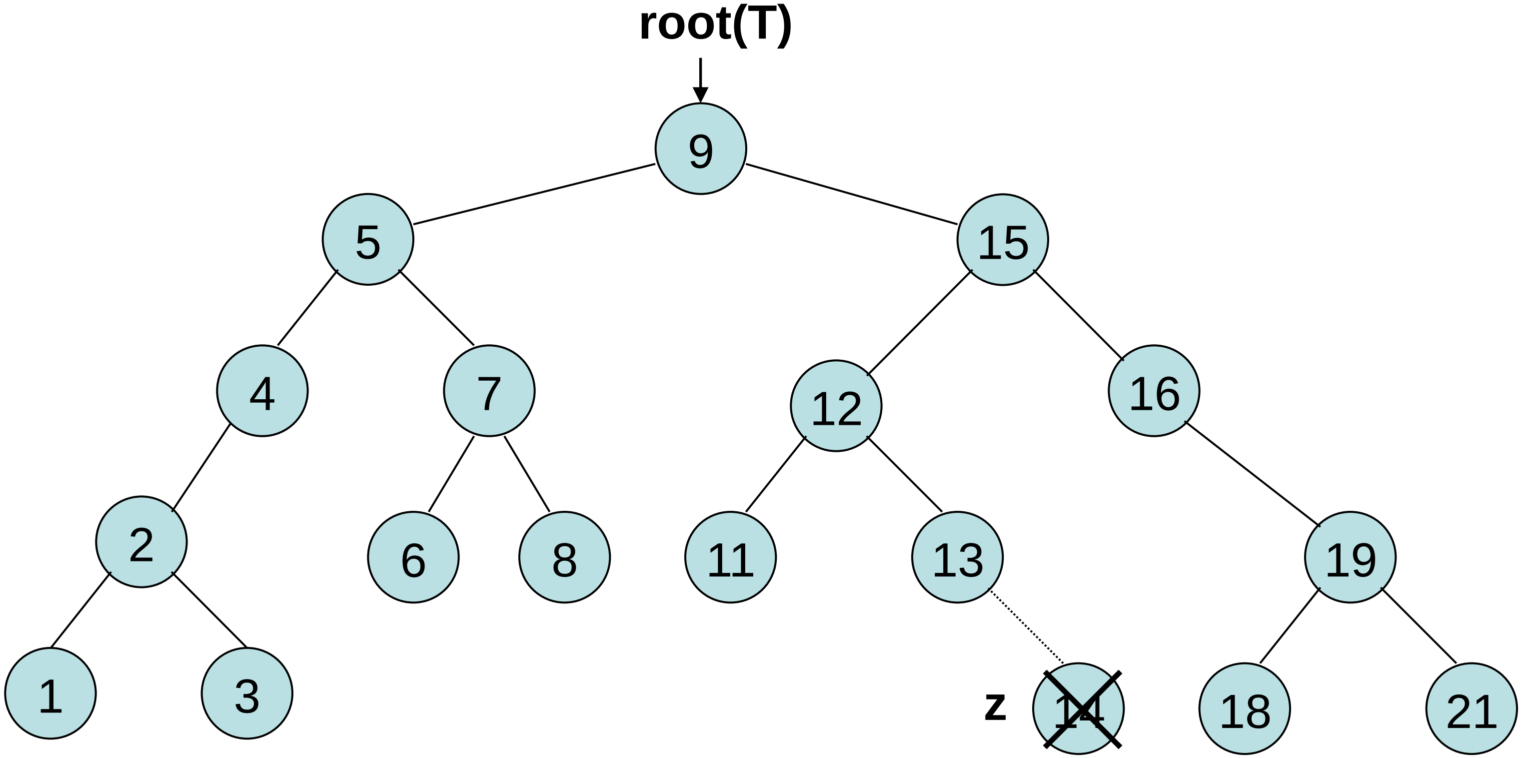
Rimozione

Ci sono tre casi:

1. Se **z non ha figli**, allora si modifica p[z] che punta non più a z, ma a NIL.
2. Se **z ha un unico figlio**, allora si taglia fuori z dall'albero, facendo puntare p[z] all'unico figlio di z.
3. Se **z ha due figli**, allora si individua il successore, ossia il minimo del suo sottoalbero destro. Il **successore y** ha nessun figlio o 1 figlio. Quindi **y prende il posto di z**, riconducendosi al caso 1 e 2. Alla fine i dati in y vengono copiati in z.

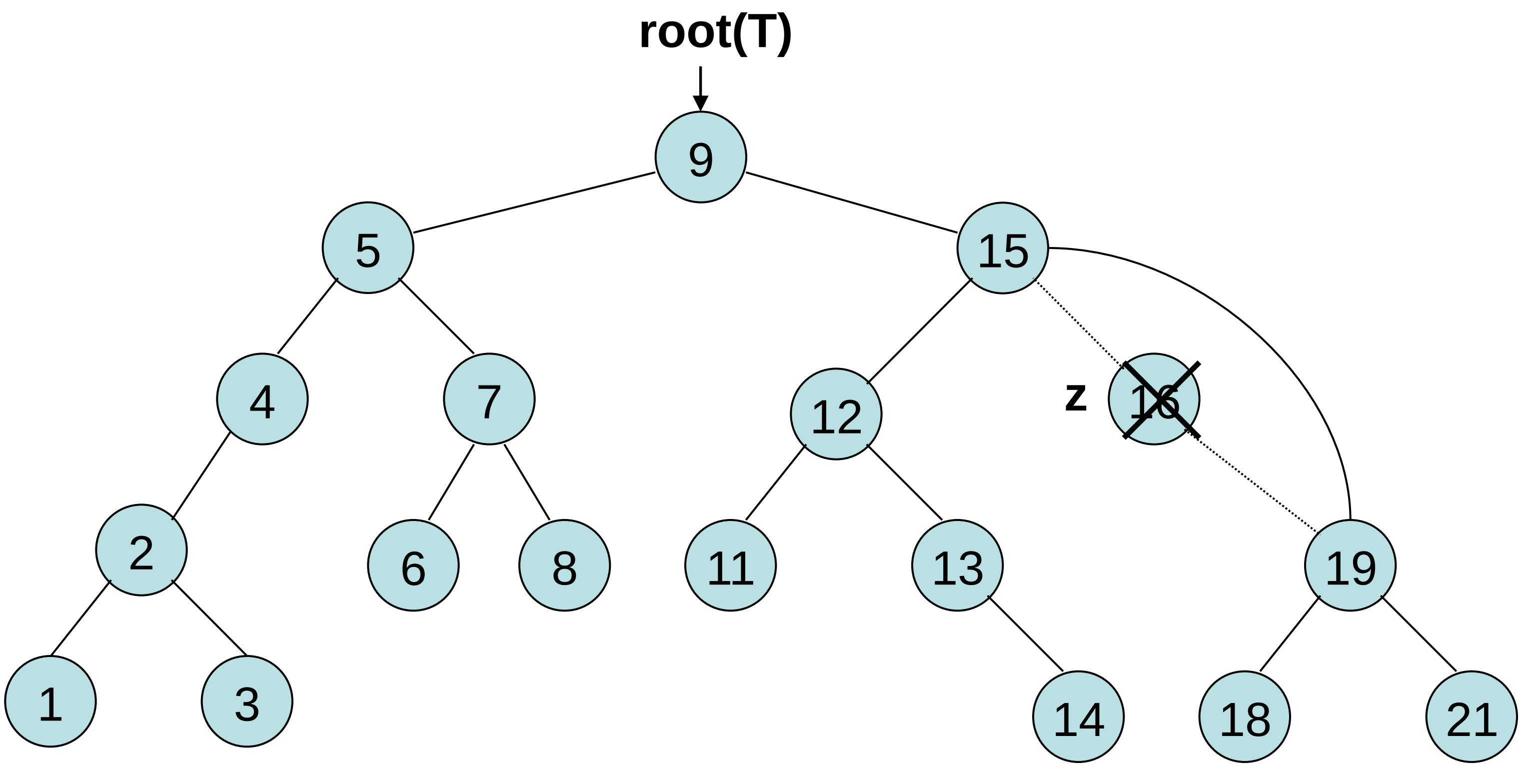
Rimozione

TREE-DELETE(T, z)
Caso 1: z senza figli.



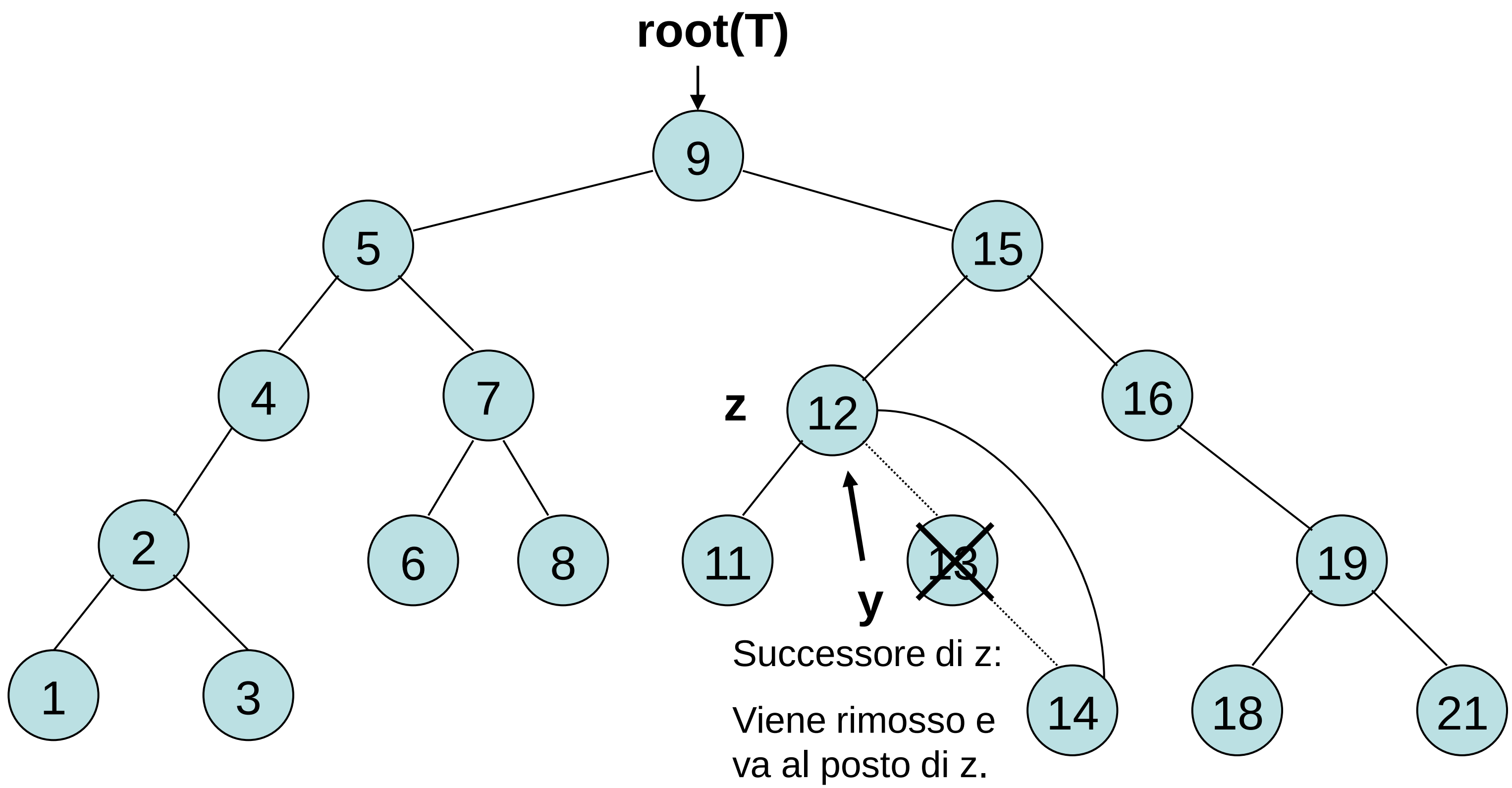
Rimozione

TREE-DELETE(T, z)
Caso 2: z con 1 figlio.



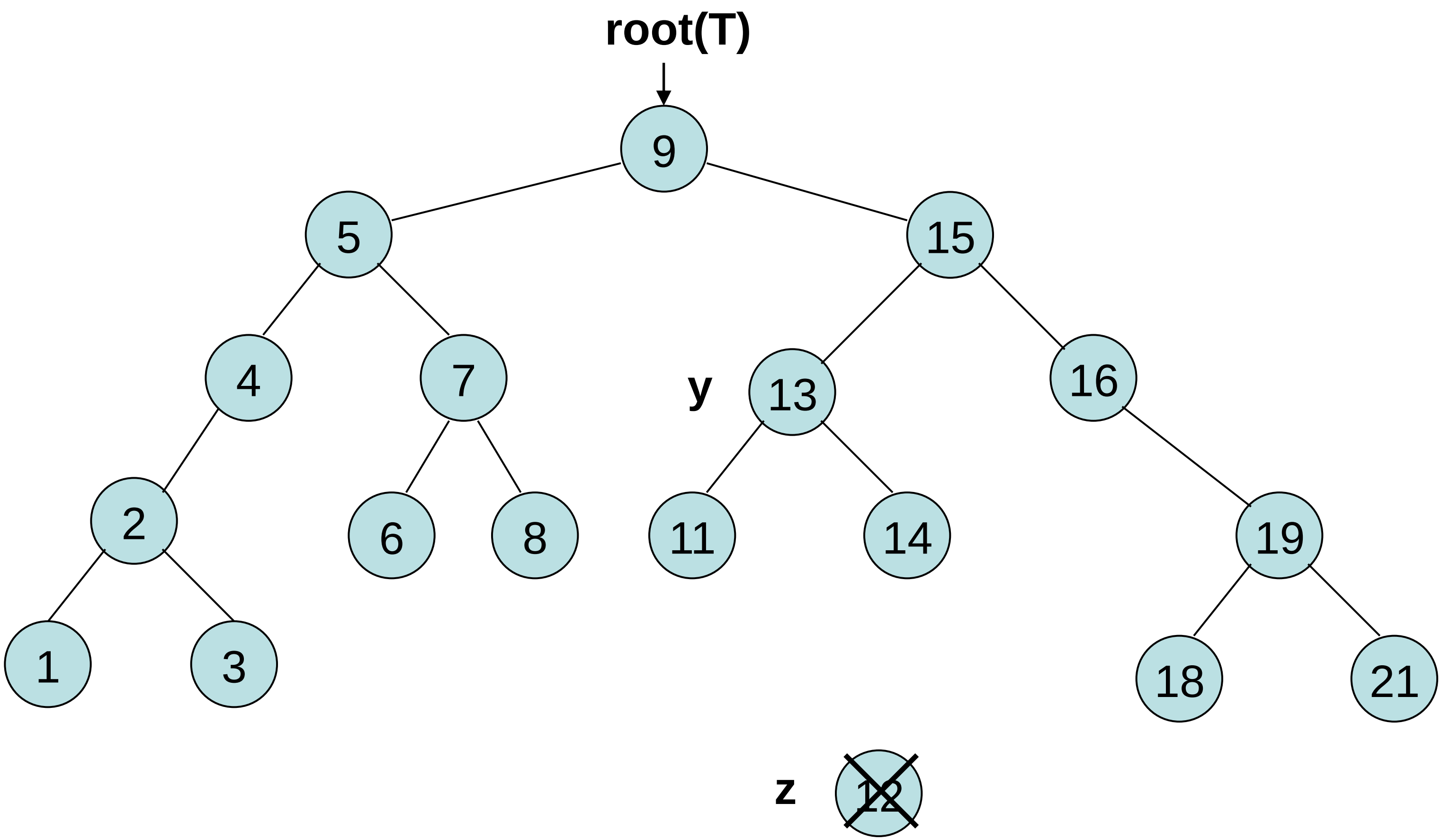
Rimozione

TREE-DELETE(T, z)
Caso 3: z con 2 figli.



Rimozione

TREE-DELETE(T, z)
Caso 3: z con 2 figli.



Rimozione

- L’operazione di rimozione può richiede pochi passi, ma può succedere che TREE-SUCCESSOR(z) venga eseguito.
- **TREE-SUCCESSOR(z) è O(h)**, dove h è l’altezza dell’albero. Quindi, la **rimozione richiede tempo O(h)**.
- Riassumendo:

  TREE-INSERT() e **TREE-DELETE()** richiedono tempo **O(h)**, dove h è l’altezza dell’albero, ossia il cammino massimo tra la radice e una foglia.

Alberi binari di ricerca

- Gli alberi di ricerca binari di ricerca sono **strutture di dati** sulle quali vengono realizzate molte delle operazioni definite **sugli insiemi dinamici**.
- **Alcune operazioni** sono: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT e DELETE.
- Queste operazioni richiedono un **tempo proporzionale all'altezza h dell'albero**.
- **E' importante che l'albero sia bilanciato**, in modo da non ricondurci ad una catena lineare. In questo caso, se si sono stati inseriti n nodi, si ha un tempo medio pari a $\Theta(n)$.

Alberi binari di ricerca

- Se l'albero è **bilanciato**, **l'altezza** dell'albero è pari a **O(log(n))**. Dunque, le operazioni sono eseguite nel caso peggiore con un tempo $\Theta(\log(n))$.
- Si può dimostrare che l'altezza di un albero binario di ricerca **costruito in modo casuale è O(log(n))**.
- Nella pratica, non si può garantire che gli alberi binari di ricerca siano sempre bilanciati!
- Ci sono varianti che danno questa garanzia. In questi casi le prestazioni nel caso peggiore per le operazioni di base sono O(log(n)) (vedi gli RB-alberi).