Alberi binari di ricerca

Algoritmi e Strutture Dati + Lab

Informatica
Università degli Studi di Bari “Aldo Moro”

Nicola Di Mauro

Introduzione

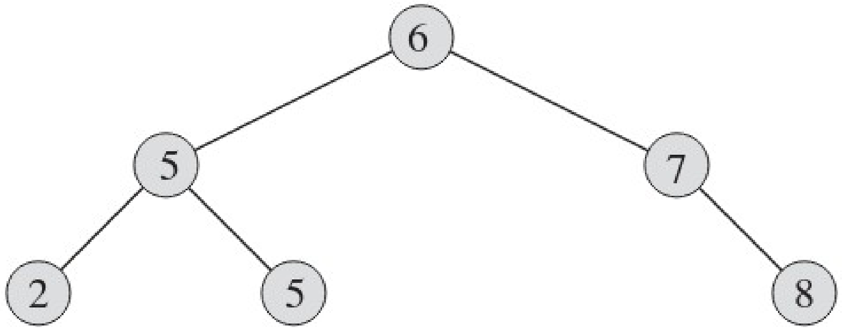
- Gli alberi di ricerca sono strutture dati che supportano molte operazioni sugli insiemi dinamici
  - SEARCH(S, k)
    - Query che dato un insieme S e un valore chiave k restituisce un elemento x di S
  - MINIMUM(S) e MAXIMUM(S)
    - Query su un insieme S totalmente ordinato che restituiscono un elemento di S con la chiave più piccola, risp. con la chiave più grande
  - SUCCESSOR(S,x) e PREDECESSOR(S, x)
    - Query che dato un elemento x, la cui chiave appartiene ad S, restituisce l'elemento restituisce il prossimo elemento più grande, risp. più piccolo
  - INSERT(S, x) e DELETE(S,x)
    - Operazioni di modifica che inseriscono, risp. rimuovono, elementi da S
- Gli alberi di ricerca possono essere utilizzati come dizionari o come code di priorità

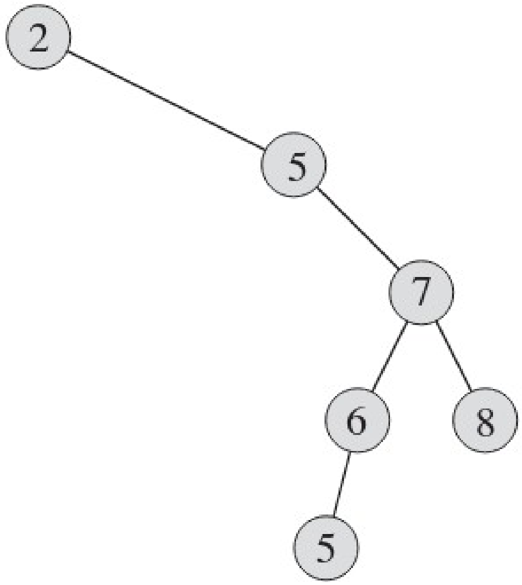
Definizione

- Un albero binario di ricerca è organizzato in un albero binario
  - Dato un nodo x le chiavi nel sotto-albero sinistro sono al più *x.key* e le chiavi nel sotto-albero destro sono almeno *x.key*
  - Lo stesso insieme di valori può essere rappresentato con alberi diversi
    - La complessità delle operazioni di ricerca dipende dalla profondità dell'albero

Definizione /2

- Le chiavi vengono memorizzate in modo tale da soddisfare la proprietà di un albero binario di ricerca (BST)
  - Sia x un nodo in un BST. Se y è un nodo nel sotto-albero sinistro (risp. destro) di x, allora $y.key \leqslant x.key$ (risp. $x.key \leqslant y.key$)





Stampa di un BST

- Sfruttando la proprietà di un BST possiamo stampare in ordine gli elementi di un BST con una funzione ricorsiva di visita inorder

```
INORDER-TREE-WALK(BST T)
   INORDER-TREE-WALK(T, T.root())

INORDER-TREE-WALK(BST T, Node n)
   if (!T.sx_empty(n))
      INORDER-TREE-WALK(T, T.sx(n))
   print n.key
   if (!T.dx_empty(n))
      INORDER-TREE-WALK(T, T.dx(n))
```

- Complessità lineare rispetto al numero dei nodi del BST

Operazioni in un BST

- Spesso si ha la necessità di cercare una chiave in un BST
- Oltre alla operazione di SEARCH, un BST mette a disposizione anche operazioni di MINIMUN, MAXIMUM, SUCCESSOR e PREDECESSOR
  - Complessità lineare rispetto all'altezza dell'abero

Specifica

- Operatori
  - create() → BST
  - empty(BST) → boolean
  - insert(BST, KEY) → BST
  - erase(BST, KEY) → BST
  - search(BST, KEY) → NODE
  - minimum(BST) → NODE
  - maximum(BST) → NODE
  - predecessor(BST, NODE) → NODE
  - successor(BST, NODE) → NODE

Ricerca

- La procedura ha una complessità lineare rispetto alla profondità del BST

```
TREE-SEARCH(BST T, KEY k)
   TREE-SEARCH(T, T.root())

TREE-SEARCH(BST T, Node n, KEY k)
   if (n.key == k)
      return n
   if (k < n.key && !T.sx_empty(n))
      return TREE-SEARCH(T, T.sx(n), k)
   if (k > n.key && !T.dx_empty(n))
      return TREE-SEARCH(T, T.dx(n), k)
   return NULL
```

Minimo e Massimo

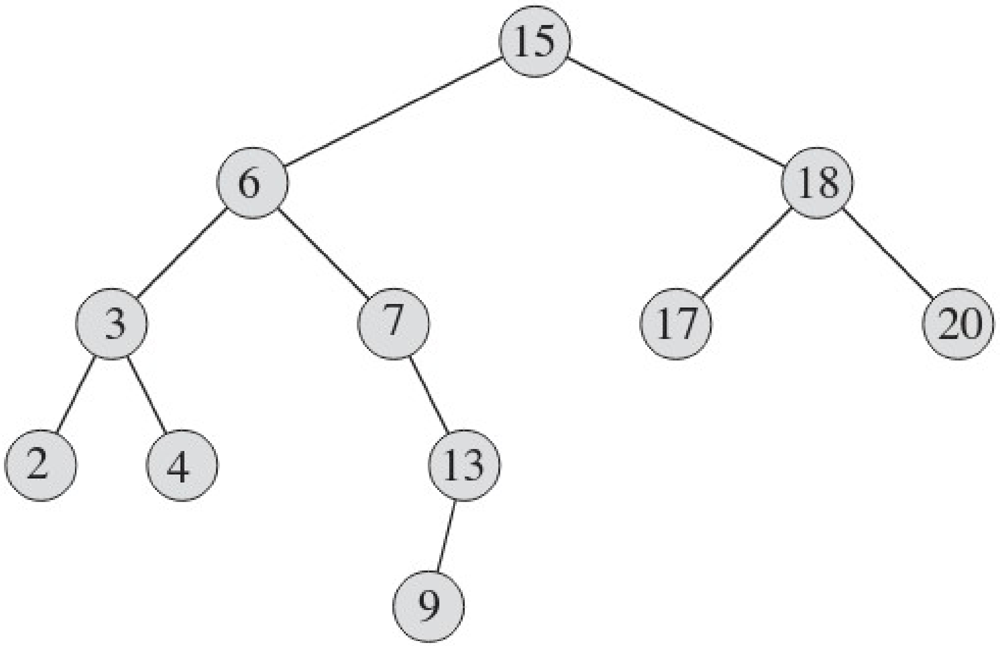
- Complessità lineare rispetto alla profondità del BST

```
TREE-MINIMUM(BST T)
   TREE-MINIMUM(T, T.root())

TREE-MINIMUM(BST T, Node n)
   if (T.sx_empty(n))
      return n
   return TREE-MINIMUM(T, T.sx(n))

TREE-MAXIMUM(BST T)
   TREE-MAXIMUM(T, T.root())

TREE-MAXIMUM(BST T, Node n)
   if (T.dx_empty(n))
      return n
   return TREE-MAXIMUM(T, T.sx(n))
```

Successore e predecessore

- Dato una chiave, a volte è necessario individuare il suo successore/predecessore nell'ordinamento

```
TREE-SUCCESSOR(BST T, Node n)
   if (!T.dx_empty(n))
   return TREE-MINIMUM(T, T.dx(n))
   if (T.root() != n)
   p = T.parent(n)
   while (T.dx(p) == n)
   n = p
   p = T.parent(n)
   return p
   return NULL
```



- Se il nodo n ha un figlio destro il successore è il minimo del sotto-albero destro
  - Il successore di 15 è 17
- Altrimenti il successore è il più piccolo antenato di n il cui figlio sinistro è anche antenato di n
  - Il successore di 13 è 15

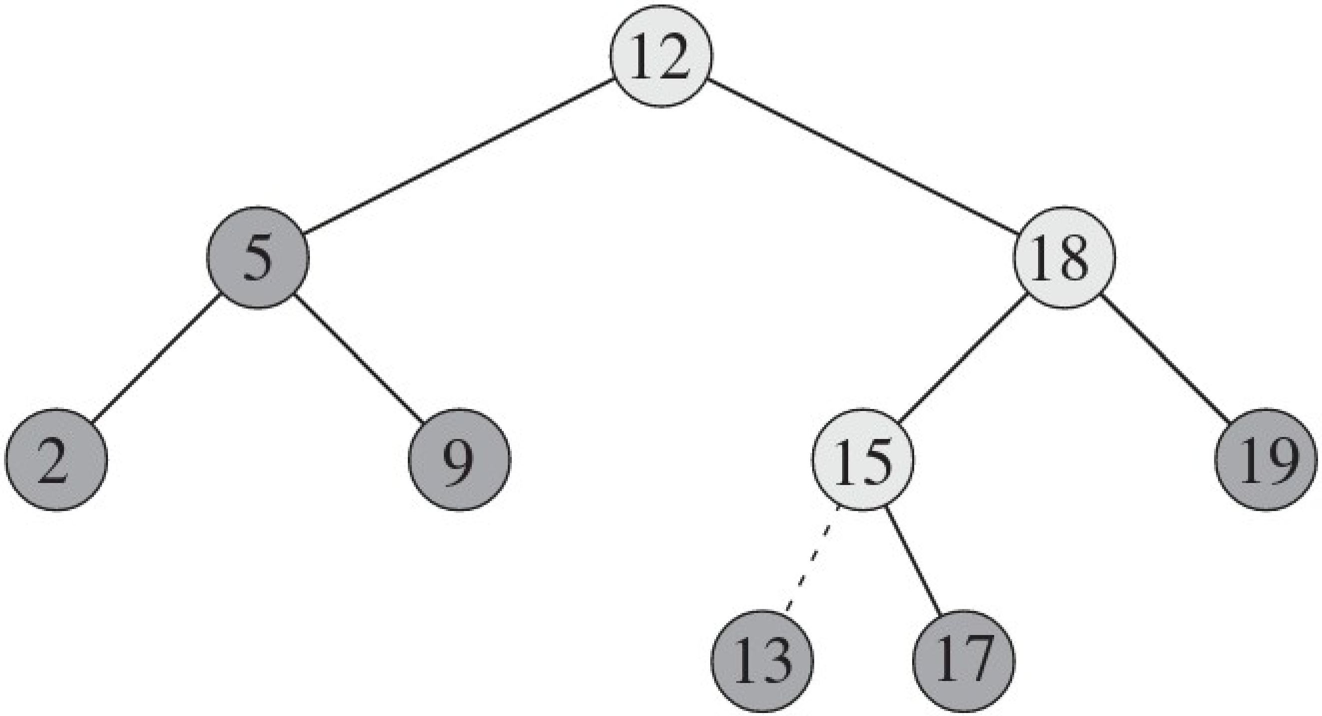
Inserimento

- Inserire una chiave mantenendo la proprietà di BST

```
TREE-INSERT(BST T, KEY k)
    if (T.empty())
            T.ins_root()
            n.key = k
            T.write(T.root(), n)
    else
        x = T.root()
        found = true
        while (found)
            if (k < x.key && !T.sx_empty()) x = T.sx(x)
            elif (k > x.key && !T.dx_empty()) x = T.sx(x)
            else found = false
        if (k < x.key)
            T.ins_sx(x)
            n.key = k
            T.write(T.sx(x), n)
        else
            T.ins_dx(x)
            n.key = k
            T.write(T.dx(x), n)
```

Inserimento /2

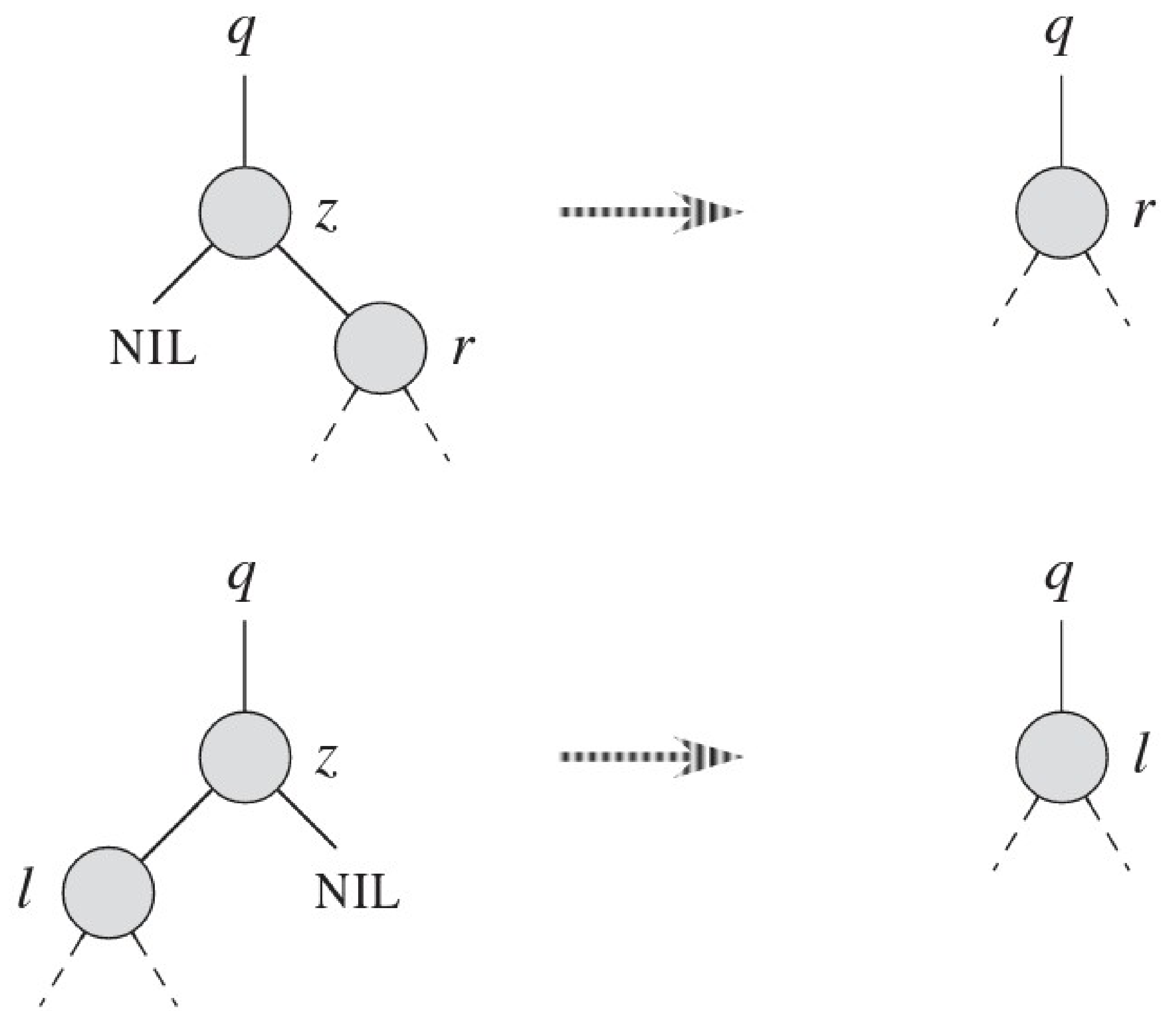
- Inserimento della chiave 13



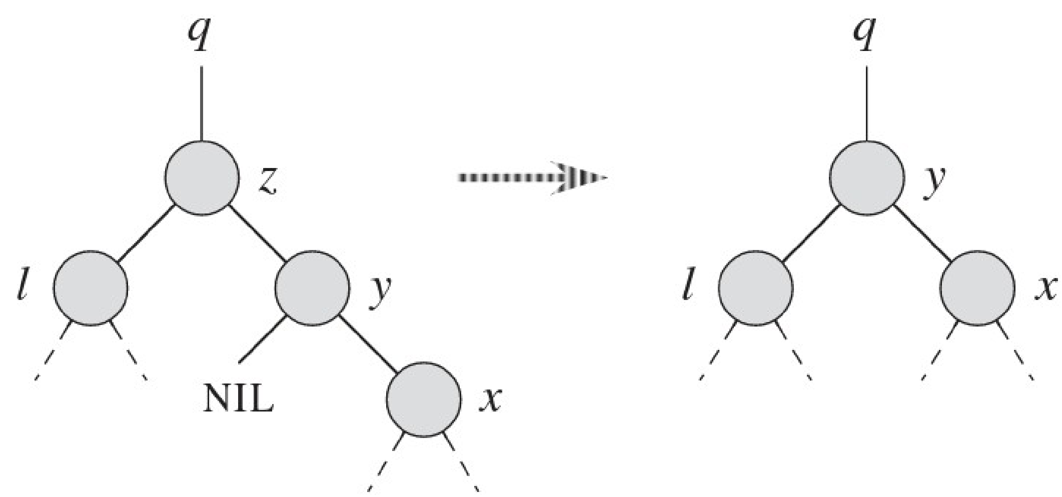
Cancellazione

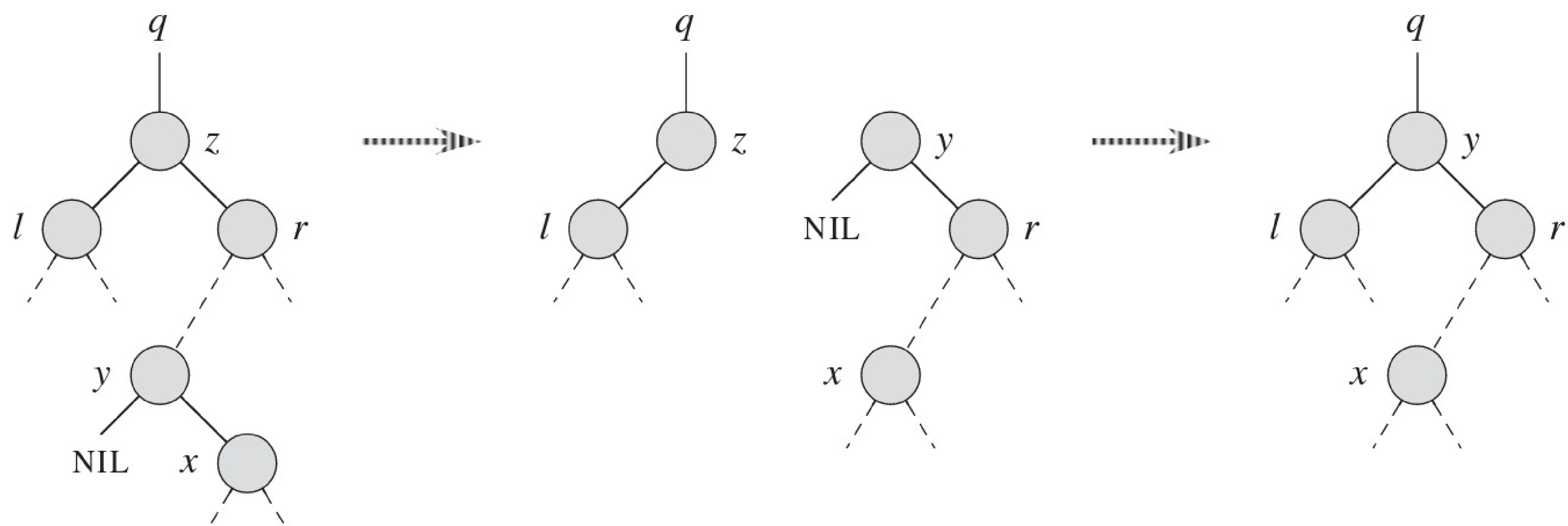
- Rimozione di una chiave *k* memorizzata in un nodo *z* del BST
  - Se *z* è foglia rimuoviamo il nodo *z* dal BST
  - Se *z* ha un solo figlio *y* allora facciamo occupare ad *y* la posizione di *z* nel BST
  - Se *z* ha due figli, allora cerchiamo il successore *y* di *z*, e facciamo occupare ad *y* la posizione di *z* nel BST
    - y si troverà nel sotto-albero destro di z
    - Si dovrà distinguere fra
      - a) y figlio destro di z, e
      - b) y appartenente al sotto-albero destro di z ma non figlio destro di z
        - Si noti che in questo caso y non ha figlio sinistro, è il minimo di un sotto-albero (si veda la definizione di successore)

Cancellazione /2



Cancellazione /3





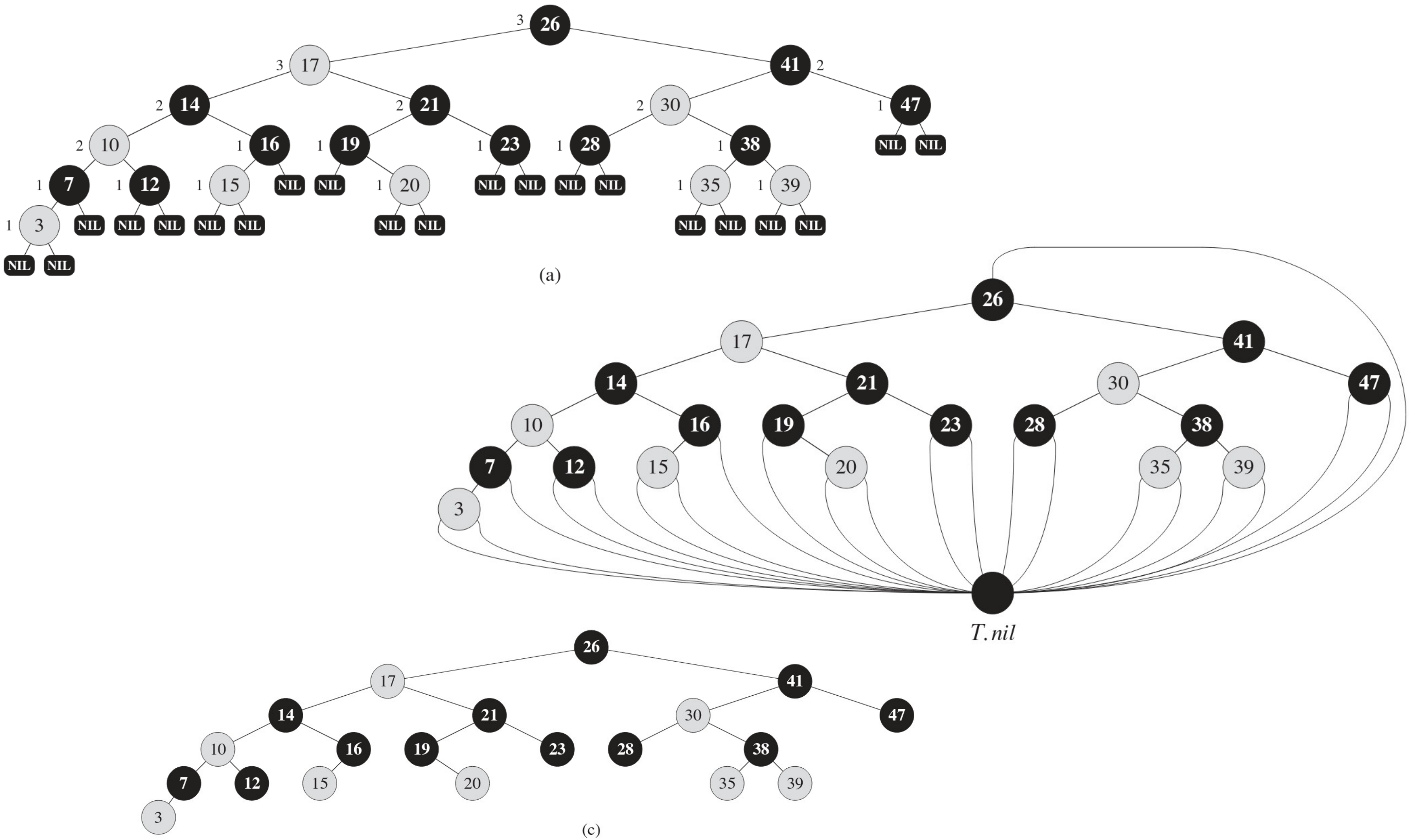
Alberi rosso-neri

- Abbiamo visto che in un BST le operazioni hanno complessità lineare rispetto all'altezza dell'albero
  - Se l'altezza dell'albero aumenta le prestazioni del BST degradano
- Gli alberi rosso-neri (RBT) sono uno dei tanti schemi di bilanciamento di BST che garantiscono complessità log-lineare nel caso pessimo
- Un albero rosso-nero è un BST in cui ogni nodo memorizza anche il proprio colore, ROSSO o NERO
  - Imponendo un vincolo sul colore dei nodi su un path dalla radice ad una foglia, i RBT assicurano che nessun path ha lunghezza doppia di un altro (l'albero è approssimativamente bilanciato)

Proprietà

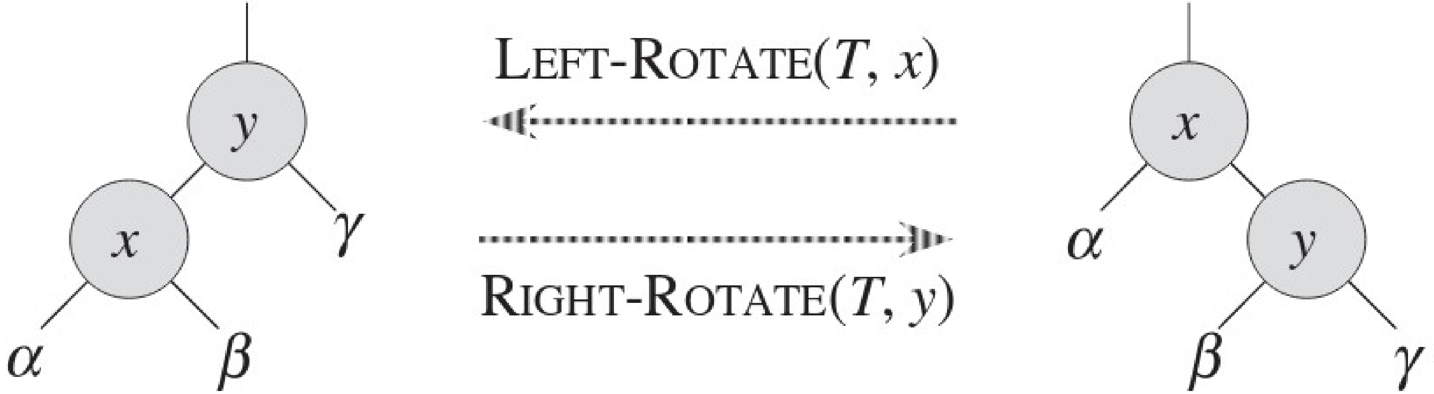
- UN RBT soddisfa le seguenti proprietà
  - Ogni nodo è rosso o nero
  - La radice è nera
  - Ogni foglia è nera
  - Se un nodo è rosso, entrambi i figli sono neri
  - Per ogni nodo, tutti i path dal nodo ad una foglia contengono lo stesso numero di nodi neri
- Si dimostra che un RBT con n nodi interni ha altezza al più 2lg(n+1)
  - Ne consegue che la ricerca in un RBT ha complessità log-lineare

Rappresentazioni equivalenti



Rotazioni

- Le operazioni di inserimento e cancellazione per BST potrebbero dare un BST che viola le proprietà di RBT
- Per ripristinare le proprietà si dovrà cambiare il colore di alcuni nodi nell'albero
- Questa si ottiene con delle operazioni di rotazione che preservano le proprietà di un BST



AVL Tree

- Un AVL tree è un BST bilanciato in altezza
  - Per ogni nodo x, le altezze del sotto-albero sinistro e destro di x differiscono di al più 1
- Per implementare un AVL si mantiene un attributo extra per ogni nodo, la sua altezza
- Per l'inserimento usiamo l'inserimento classico per i BST
  - L'albero potrebbe non essere bilanciato in altezza
    - L'altezza dei sotto-alberi sinistro e destro di qualche nodo potrebbero differire di 2
    - Usando delle rotazioni si rendono tali alberi bilanciati in altezza
- Alternativamente si può usare una procedura ricorsiva che inserisce un nodo in un AVL tree conservando la proprietà di un AVL in tempo log-lineare

Treaps

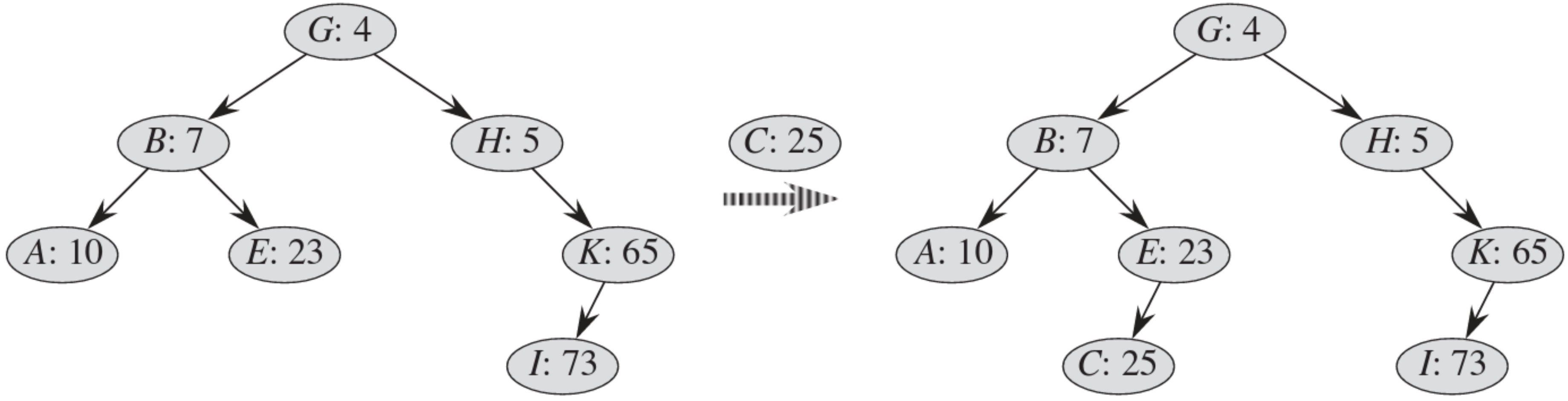
- L'inserimento di n chiavi in un BST potrebbe dar luogo ad un albero molto sbilanciato, degradando le prestazioni della ricerca di chiavi
- Si può dimostrare che BST costruiti a random tendono ad essere bilanciati
  - Dato un insieme di elementi si inseriscono gli elementi nel BST seguendo l'ordine di una loro permutazione random
- **Problema**: se gli elementi vengono forniti uno alla volta come possiamo costruire in modo random il corrispondente BST?
  - La soluzione è quella di usare i treaps

Treaps /2

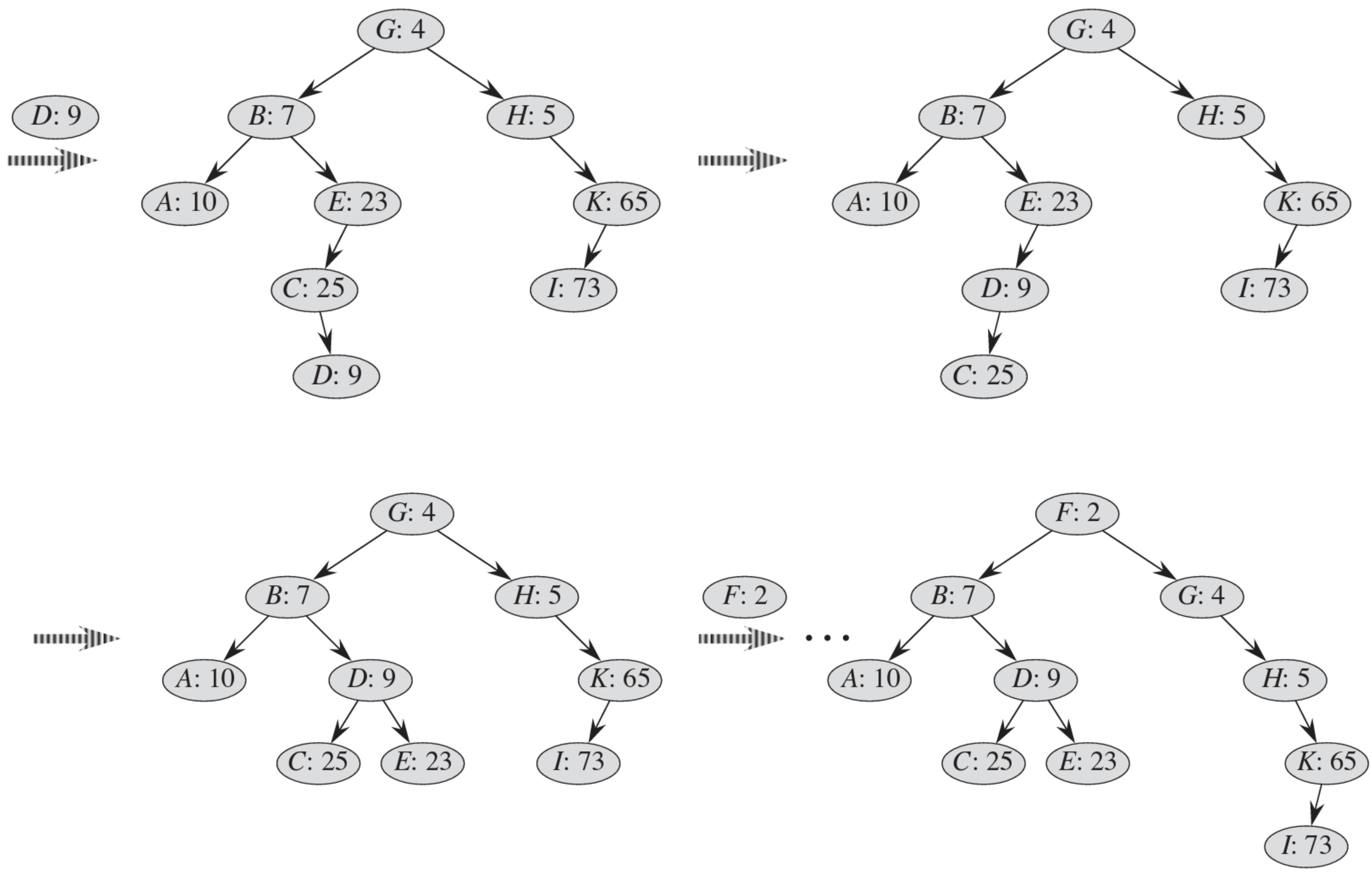
- Un treap è un BST che memorizza i nodi in modo diverso dal tradizionale
  - Ad ogni nodo, oltre alla chiave è associata una priorità, un numero random scelto indipendentemente per ogni nodo
    - **Tutte le chiavi e le priorità sono distinte**
  - I nodi in un treap sono ordinati in modo tale che le chiavi obbediscono alle proprietà di un BST e le priorità alla proprietà di un min-heap
    - Se v è figlio sinistro di u, allora v.key < u.key
    - Se v è figlio destro di u, allora v.key > u.key
    - Se v è figlio di u, allora v.priority > u.priority
- Il nome treap
  - Ha le caratteristiche di un BST e di un heap

Treaps / 3

- Supponiamo di inserire i nodi x1, x2, …, xn in un treap
  - Il treap risultante è un albero che si sarebbe ottenuto se i nodi fossero stati inseriti in un normale BST nell'ordine dato dalle loro priorità (scelte random)



Treaps /4


D: 9
G: 4
B: 7
H: 5
A: 10
E: 23
K: 65
C: 25
I: 73
D: 9
G: 4
B: 7
H: 5
A: 10
E: 23
K: 65
D: 9
I: 73
C: 25
G: 4
B: 7
H: 5
A: 10
D: 9
K: 65
C: 25
E: 23
I: 73
F: 2
...
F: 2
B: 7
G: 4
A: 10
D: 9
H: 5
C: 25
E: 23
K: 65
I: 73