

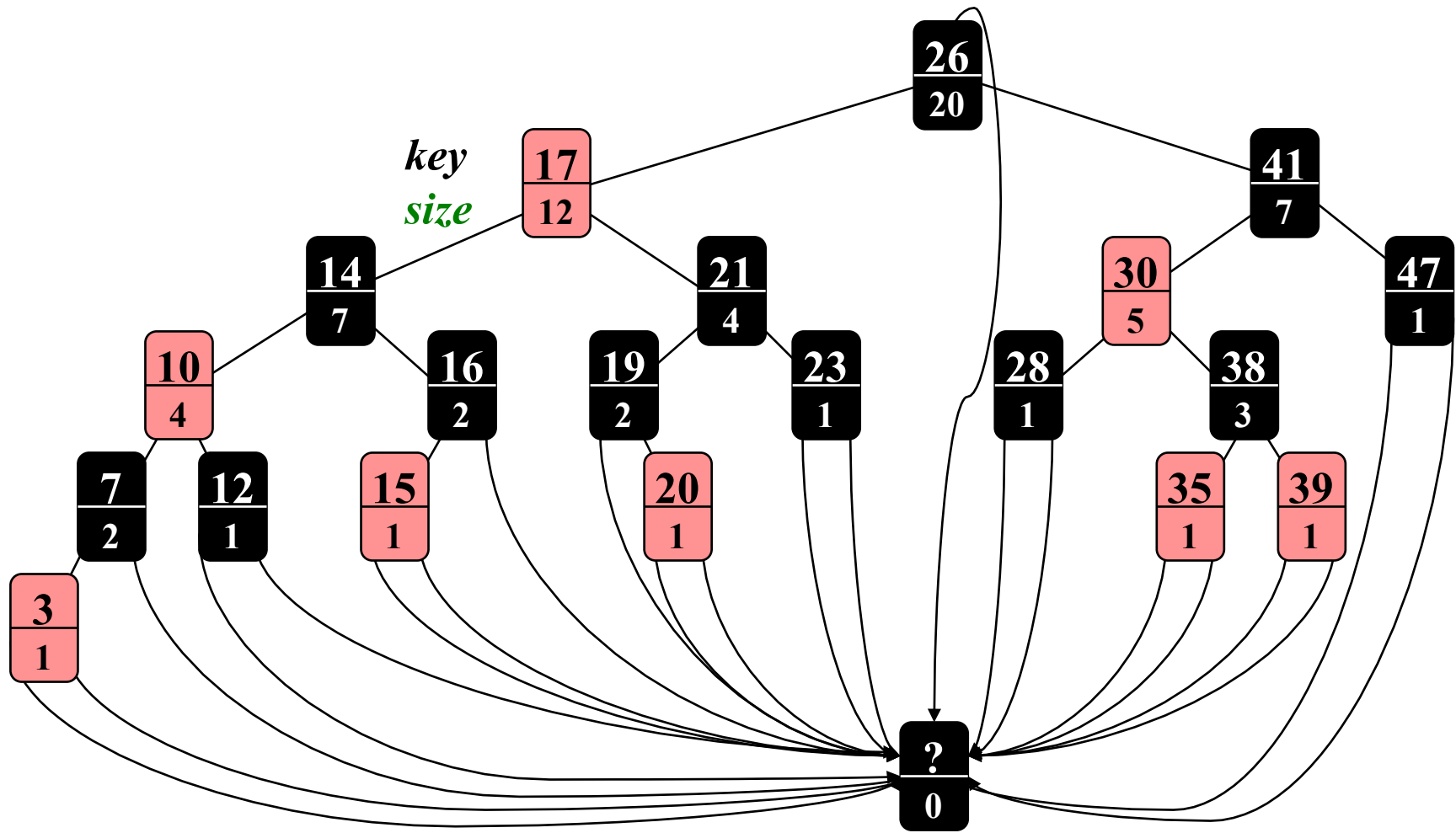
## Come aumentare gli alberi

La soluzione di alcuni problemi algoritmici richiede la progettazione di una struttura dati appropriata.

Spesso una tale struttura si può ottenere aumentando strutture dati note.

Supponiamo ci serva una struttura dati su cui poter eseguire, oltre alle operazioni previste per gli alberi di ricerca, anche l'operazione di statistica ordinale *Select*( $k$ ) che ritorna il nodo con la chiave  $k$ -esima.

Un modo per farlo è aumentare gli alberi rosso-neri aggiungendo a ciascun nodo  $x$  un ulteriore campo intero  $x.size$  in cui memorizzare il numero di nodi interni del sottoalbero di radice  $x$ .



## Osservazione.

Se usiamo la sentinella *T.nil* e poniamo *T.nil.size* = 0 allora per ogni nodo interno *x* vale l'equazione

$$x.size = 1 + x.left.size + x.right.size$$

Possiamo realizzare facilmente *Select*:

*Select*( $x$ ,  $k$ ) //  $1 \leq k \leq x.size$

// Trova il nodo con chiave  $k$ -esima

// nel sottoalbero di radice  $x$

$i = 1 + x.left.size$

if  $i == k$

return  $x$

elseif  $i > k$

return *Select*( $x.left$ ,  $k$ )

else

return *Select*( $x.right$ ,  $k-i$ )

Possiamo realizzare facilmente anche l'operazione inversa ***Rank***( $x$ ) che trova la posizione  $k$  della chiave di  $x$  nella sequenza ordinata delle chiavi dell'albero

***Rank***( $T, x$ )

*// Trova la posizione  $k$  della chiave di  $x$*

$i = 1 + x.left.size$

$y = x$

**while**  $y.p \neq T.nil$

**if**  $y == y.p.right$

$i = i + 1 + y.p.left.size$

$y = y.p$

**return**  $i$

Naturalmente dobbiamo modificare *RB-Insert* e *RB-Delete* per mantenere aggiornato il campo *size* dei nodi

```
RB-Insert (T, z)    // z.left = z.right = T.nil  
    Insert (T, z)  
    z.color = RED  
    RB-Insert-Fixup (T, z)
```

*RB-Insert* ha due fasi.

Nella prima si scende dalla radice fino ad una foglia che viene quindi sostituita con il nuovo nodo.

Nella seconda si ripristinano le proprietà violate.

Per la prima fase basta mettere **1** nel campo *size* del nuovo nodo  $z$  e aumentare di **1** il campo *size* di tutti i nodi incontrati scendendo verso la foglia che verrà sostituita con  $z$ .



***Insert***( $T, z$ )

$z.size = 1$  // istruzione aggiunta

$x = T.root, y = T.nil$

**while**  $x \neq T.nil$

$x.size = x.size + 1$  // istruzione aggiunta

$y = x$

**if**  $z.key < y.key$

$x = y.left$

**else**  $x = y.right$

$z.p = y$

**if**  $y == T.nil$

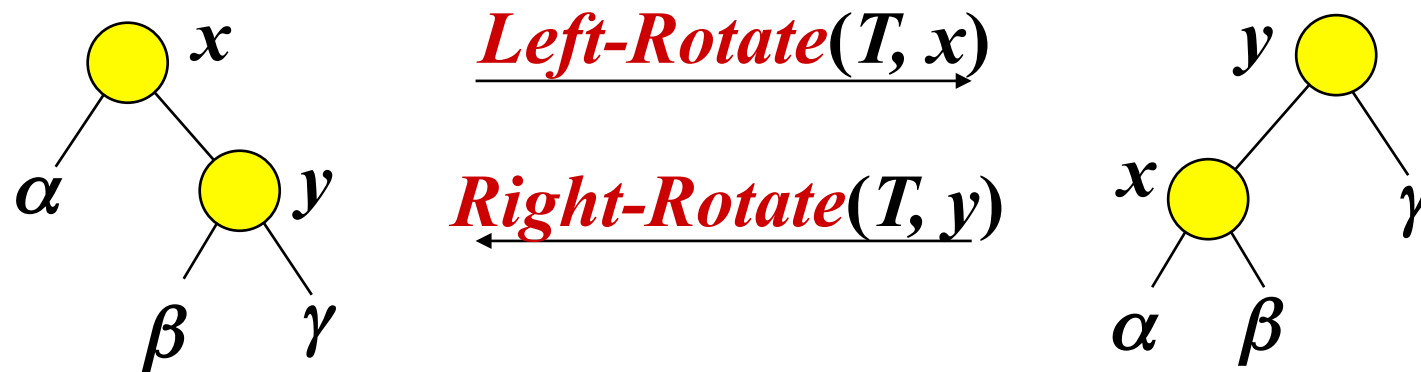
$T.root = z$

**elseif**  $key[z] < key[y]$

$x.left = z$

**else**  $x.right = z$

Nella seconda fase le modifiche alla struttura sono dovute alle rotazioni.



I campi *size* dei nodi diversi da  $x$  e  $y$  rimangono invariati.

Basta quindi ricalcolare i campi *size* dei due nodi  $x$  e  $y$  usando la relazione:

$$x.size = 1 + x.left.size + x.right.size$$

***Left-Rotate***( $T, x$ )

$y = x.right$

$x.right = y.left, y.left.p = x$

$y.p = x.p$

**if**  $x.p == T.nil$

$T.root = y$

**elseif**  $x == x.p.left$

$x.p.left = y$

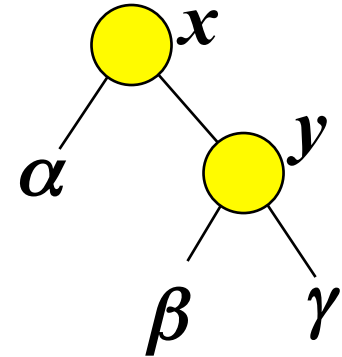
**else**

$x.p.right = y$

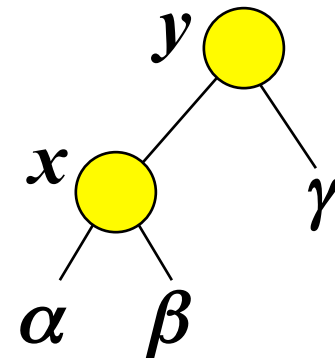
$x.p = y, y.left = x$

$y.size = x.size$  // istruzioni aggiunte

$x.size = 1 + x.left.size + x.right.size$



***LeftRot***( $T, x$ )



Anche la ***RB-Delete*** ha due fasi:

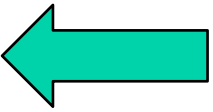
Nella prima viene tolto un nodo  $y$  avente uno dei sottoalberi vuoto sostituendolo con la radice dell'altro sottoalbero.

Per questa fase basta diminuire di 1 il campo ***size*** di tutti i nodi nel cammino dalla radice a tale nodo.

```

RB-Delete(T, z)           // z ≠ T.nil
  if z.left == T.nil or z.right == T.nil
    y = z
  else y = Successor(z), z.key = y.key
  if y.left == T.nil
    x = y.right
  else x = y.left
  x.p = y.p // mette x al posto di y
  if y.p == T.nil
    T.root = x
  elseif y == y.p.left
    y.p.left = x
  else y.p.right = x
  if y.color == BLACK
    RB-Delete-Fixup(T, x)

```


*w* = *x.p* // istruzioni aggiunte  
 while *w* ≠ *T.nil*  
   *w.size* = *w.size* - 1, *w* = *w.p*

Nella seconda fase di ***RB-Delete*** le modifiche alla struttura dell'albero sono dovute alle rotazioni e quindi è sufficiente la modifica delle rotazioni già vista per ***RB-Insert***.

Le istruzioni aggiunte (quelle in verde) non aumentano la complessità delle operazioni ***RB-Insert*** e ***RB-Delete***.

## Teorema generale dell'aumento

L'aumento di una struttura dati richiede quattro passi:

1. scelta della struttura dati di base;
2. scelta delle ulteriori informazioni da memorizzare nella struttura;
3. verifica che esse si possano mantenere durante le operazioni della struttura di base senza aumentarne la complessità asintotica;
4. sviluppo delle nuove operazioni.

Per gli alberi rosso-neri c'è un teorema che ci facilita il passo 3.

### *Teorema dell'aumento*

Sia  $x.\textit{field}$  un nuovo campo che aumenta un albero rosso-nero  $T$ .

Se il valore di  $x.\textit{field}$  si può calcolare in tempo  $O(1)$  usando soltanto le altre informazioni presenti in  $x$  e quelle presenti nei figli  $x.\textit{left}$  e  $x.\textit{right}$  comprese  $x.\textit{left}.\textit{field}$  e  $x.\textit{right}.\textit{field}$  allora il campo  $x.\textit{field}$  si può mantenere aggiornato eseguendo *RB-Insert* e *RB-Delete* senza aumentare la complessità  $O(\log n)$  di tali operazioni.



## Osservazione

Il campo  $x.size$  soddisfa tale proprietà

$$x.size = 1 + x.left.size + x.right.size$$

Se usiamo la sentinella  $T.nil$  e poniamo  $T.nil.size = 0$  questa formula vale per ogni nodo interno, compresi quelli senza figli.

## Dimostrazione

L'idea è che una modifica di  $x$ .*field* implica la modifica del campo *field* degli antenati di  $x$  ma non degli altri nodi.

*RB-Insert*( $T, z$ )    *//*  $z.left = z.right = T.nil$

*Insert*( $T, z$ )

$z.color = RED$

*RB-Insert-Fixup*( $T, z$ )

Nella prima fase *Insert* il nodo  $z$  aggiunto non ha figli e quindi  $z$ .*field* si può calcolare direttamente in tempo costante.

Basta quindi ricalcolare il campo *field* di  $z$  e di tutti i suoi antenati (tempo  $O(\log n)$ ).

***Insert***( $T, z$ )

$x = T.root, y = T.nil$

**while**  $x \neq T.nil$

$y = x$

**if**  $z.key < y.key$

$x = y.left$

**else**  $x = y.right$

$z.p = y$

**if**  $y == T.nil$

$T.root = z$

**elseif**  $z.key < y.key$

$x.left = z$

**else**  $x.right = z$

$w = z$

**while**  $w \neq T.nil$

*Ricalcola-Field*( $w$ ),  $w = w.p$

tempo  $O(\log n)$

Nella seconda fase ***RB-Insert-Fixup*** l'unico caso che può essere ripetuto è il caso 1 che non richiede rotazioni.

Negli altri casi vengono eseguite al più 2 rotazioni e ciascuna di esse richiede il ricalcolo del campo ***field*** dei due nodi ruotati e dei loro antenati.

Tempo  $O(\log n)$ .

### ***Osservazione***

Nel caso del campo ***size*** non occorre ricalcolarlo negli antenati ma questo non è sempre vero.

***Left-Rotate***( $T, x$ )

$y = x.right$

$x.right = y.left, y.left.p = x$

$y.p = x.p$

**if**  $x.p == T.nil$

$T.root = y$

**elseif**  $x == x.p.left$

$x.p.left = y$

**else**

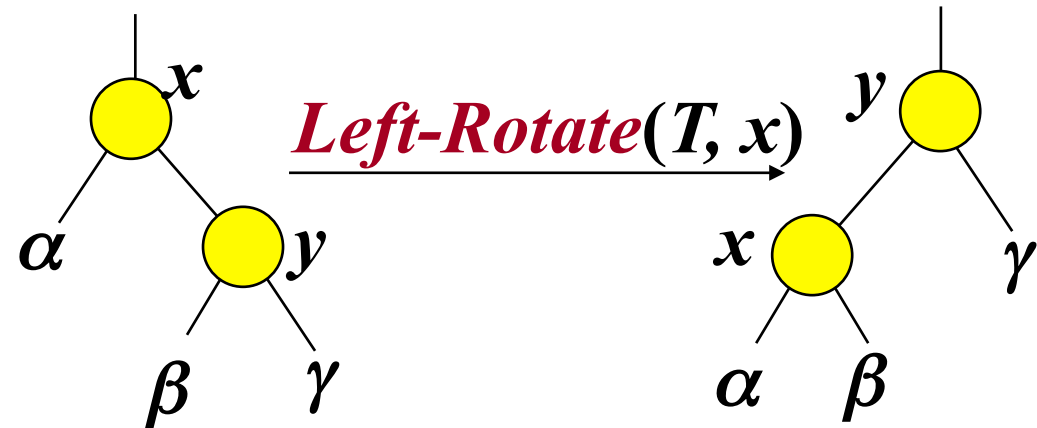
$x.p.right = y$

$x.p = y, y.left = x$

$w = x$

**while**  $w \neq T.nil$

*Ricalcola-Field*( $w$ ),  $w = w.p$



Tempo  $O(\log n)$

Nella prima fase di *RB-Delete* viene sostituito un nodo  $y$  con un suo figlio  $x$ .

Basta quindi ricalcolare il campo *field* di tutti gli antenati di  $x$ .

Tempo  $O(\log n)$ .

Nella seconda fase l'unico caso che può essere ripetuto è il caso 2 che non effettua rotazioni.

Negli altri casi vengono eseguite al più 3 rotazioni e ciascuna di esse richiede il ricalcolo del campo *field* dei due nodi ruotati e dei loro antenati.

Tempo  $O(\log n)$ .

## Alberi di intervalli

Vogliamo aumentare gli alberi rosso-neri per ottenere una struttura dati che supporta operazioni su un insieme dinamico di intervalli  $[a,b]$  con  $a$  e  $b$  numeri reali.

Oltre a *Insert* e *Delete* vogliamo una operazione *Search*( $a,b$ ) che ritorna un nodo dell'albero il cui intervallo ha intersezione non vuota con l'intervallo  $[a,b]$ .

Un intervallo  $[a,b]$  si rappresenta con i due numeri reali  $a$  e  $b$ .

Dunque ogni nodo  $x$  di un albero di intervalli ha due campi

$$x.\textit{low} = a \quad \text{e} \quad x.\textit{high} = b$$

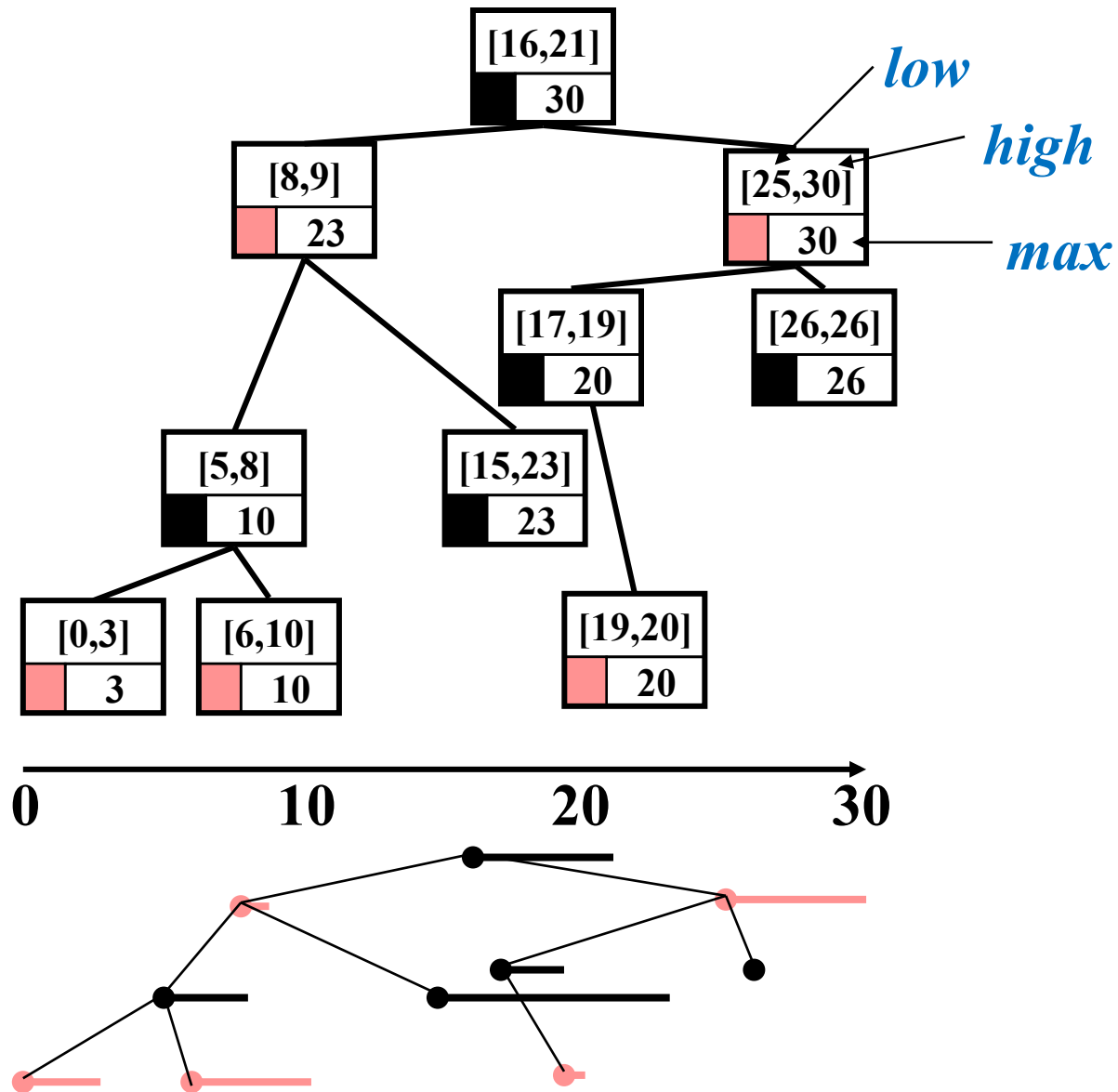
in cui sono memorizzati gli estremi di  $[a,b]$ .

Il campo  $x.\textit{low}$  è usato come chiave mentre  $x.\textit{high}$  viene trattato come informazione associata.

Aggiungiamo inoltre un campo  $x.\textit{max}$  che mantiene il valore massimo tra gli estremi degli intervalli contenuti nel sottoalbero di radice  $x$ .

Per la sentinella poniamo  $T.\textit{nil}.\textit{max} = -\infty$





$x.max$  si può calcolare in tempo  $O(1)$  come massimo tra  $x.high$ ,  $x.left.max$  e  $x.right.max$

$$x.max = \text{MAX}(x.high, x.left.max, x.right.max)$$

Se poniamo  $T.nil.max = -\infty$  questa vale anche quando  $x.left$  e/o  $x.right$  sono  $T.nil$ .

Dunque, per il teorema dell'aumento, il campo  $max$  si può mantenere eseguendo ***RB-Insert*** e ***RB-Delete*** senza aumentare la complessità asintotica di tali operazioni.

Vediamo ora come realizzare

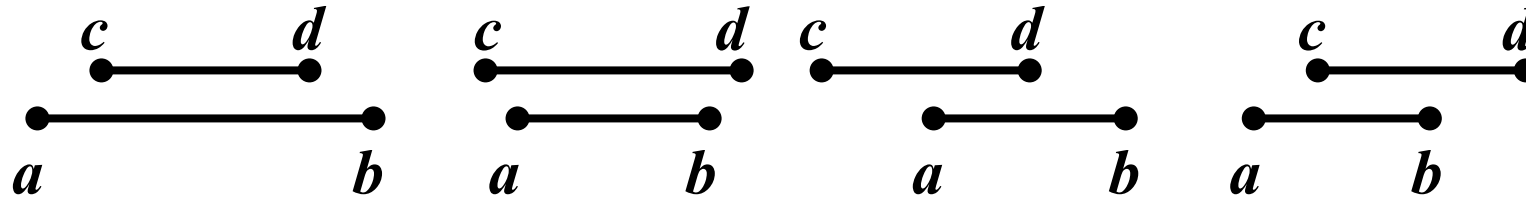
***Interval-Search***( $a, b$ )

che ritorna un nodo dell'albero il cui intervallo ha intersezione non vuota con l'intervallo  $[a, b]$ .

Dobbiamo però prima decidere come si controlla se due intervalli si intersecano.

Per gli intervalli vale la tricotomia.

1.  $[a,b]$  e  $[c,d]$  si intersecano



2.  $[a,b]$  è alla sinistra di  $[c,d]$  ( $b < c$ )



3.  $[a,b]$  è alla destra di  $[c,d]$  ( $d < a$ )



Usando questa proprietà due intervalli  $[a,b]$  e  $[c,d]$  si intersecano se e solo se i casi 2 e 3 non sono veri

Quindi il test di intersezione è semplicemente:

$$(b \geq c) \text{ and } (d \geq a)$$

e si esegue in tempo costante  $O(1)$

***Interval-Search***( $x$ ,  $a$ ,  $b$ )

// cerca un nodo nel sottoalbero di radice  $x$

// il cui intervallo interseca  $[a,b]$

**if**  $x == T.nil$  **or** “[ $a,b$ ] interseca  $x$ ”

**return**  $x$

// altrimenti o [ $a,b$ ] è alla sinistra di  $x$  o

// [ $a,b$ ] è alla destra di  $x$

**if**  $x.left.max \geq a$

// se [ $a,b$ ] non interseca nessun intervallo nel

// sottoalbero sinistro allora certamente non

// interseca nessun intervallo nel sottoalbero destro

// Posso limitarmi a cercare nel sottoalbero sinistro

**return** *Interval-Search*( $x.left$ ,  $a$ ,  $b$ )

**else**

*//  $x.left.max < a$*

*//  $[a,b]$  non interseca gli intervalli nel sottoalbero*

*// sinistro*

*// Posso limitarmi a cercare nel sottoalbero destro*

**return** *Interval-Search*(*x.right*, *a*, *b*)

Complessità limitata dall'altezza dell'albero.

Quindi  $O(\log n)$ .