Gli alberi binari di ricerca (BST: Binary Search Trees)

Gianpiero Cabodi e Paolo Camurati

Dip. Automatica e Informatica
Politecnico di Torino

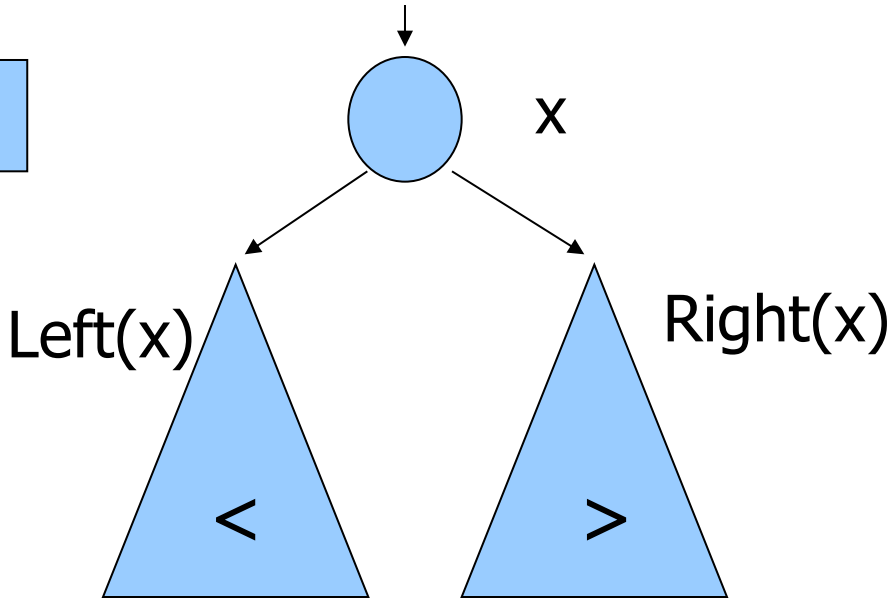
Alberi binari di ricerca (BST)

ADT albero binario con proprietà:

∀nodo x vale che:

- ∀ nodo y ∈Left(x), key[y] < key[x]
- ∀ nodo y ∈Right(x), key[y] > key[x]

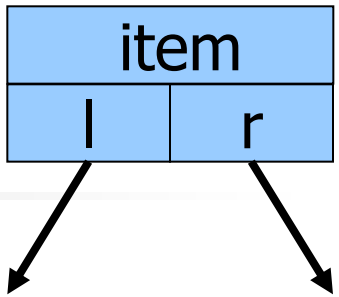
chiavi distinte!



ADT di I categoria BST

BST.h

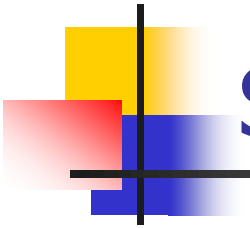
```
typedef struct  binarysearchtree *BST;

BST    BSTinit() ;
Item    BSTmin(BST) ;
Item    BSTmax(BST) ;
void   BSTinsert_leafI(BST,Item) ;
void   BSTinsert_leafR(BST,Item) ;
void   BSTinsert_root(BST,Item) ;
Item   BSTsearch(BST, Key) ;
void   BSTsortinorder(BST) ;
void   BSTsortpreorder(BST) ;
void   BSTsortpostorder(BST) ;
```



BST.c

```
#include <stdlib.h>
#include "Item.h"
#include "BST.h"
typedef struct BSTnode* link;
struct BSTnode { Item item; link l; link  r;} ;
struct binarysearchtree { link head; link z; };

link NEW(Item item, link l, link r) {
  link x = malloc(sizeof *x);
  x->item = item; x->l = l; x->r = r;
  return x;
}
BST BSTinit( ) {
  BST bst = malloc(sizeof *bst) ;
  bst->head = ( bst->z = NEW(IT , , NULL));
  return bst;
}
```

nodo sentinella



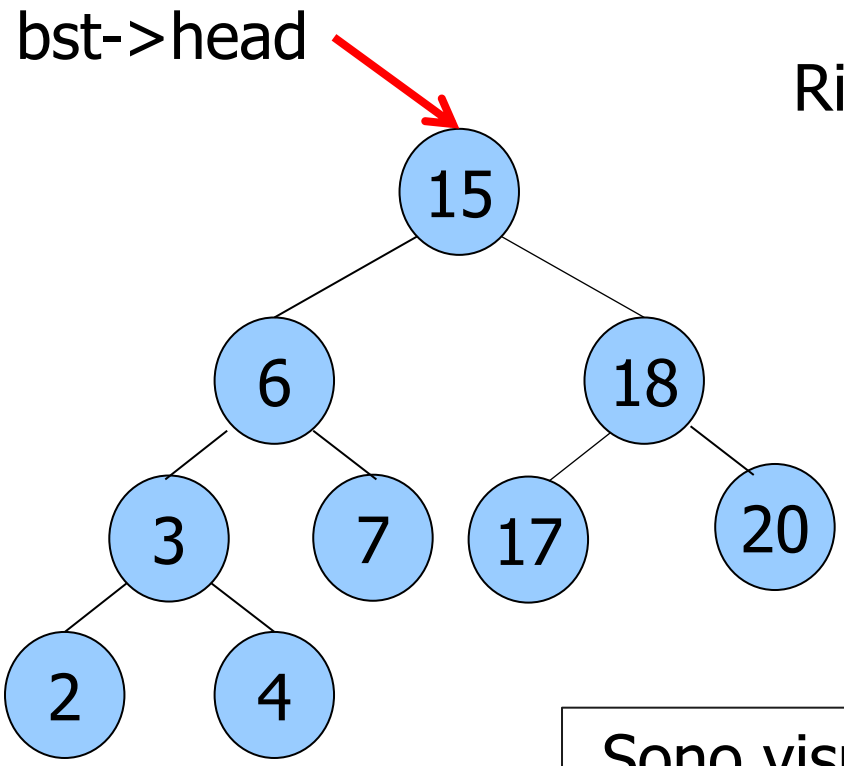
Search

Ricerca ricorsiva di un nodo che contiene un item con una chiave data:

- percorrimento dell'albero dalla radice
- terminazione: la chiave dell'item cercato è uguale alla chiave del nodo corrente (search hit) oppure si è giunti ad un albero vuoto (search miss)
- ricorsione: dal nodo corrente
  - su sottoalbero sinistro se la chiave dell'item cercato < della chiave del nodo corrente
  - su sottoalbero destro altrimenti

```
Item searchR(link h, Key k, link z) {
  if (h == z)
    return ITEMsetvoid();
  if (KEYcompare(k, KEYget(h->item))==0)
    return h->item;
  if (KEYcompare(k, KEYget(h->item))==-1)
    return searchR(h->l, k, z);
  else
    return searchR(h->r, k, z);
  }

Item BSTsearch(BST bst, Key k) {
  return searchR(bst->head, k, bst->z);
}
```

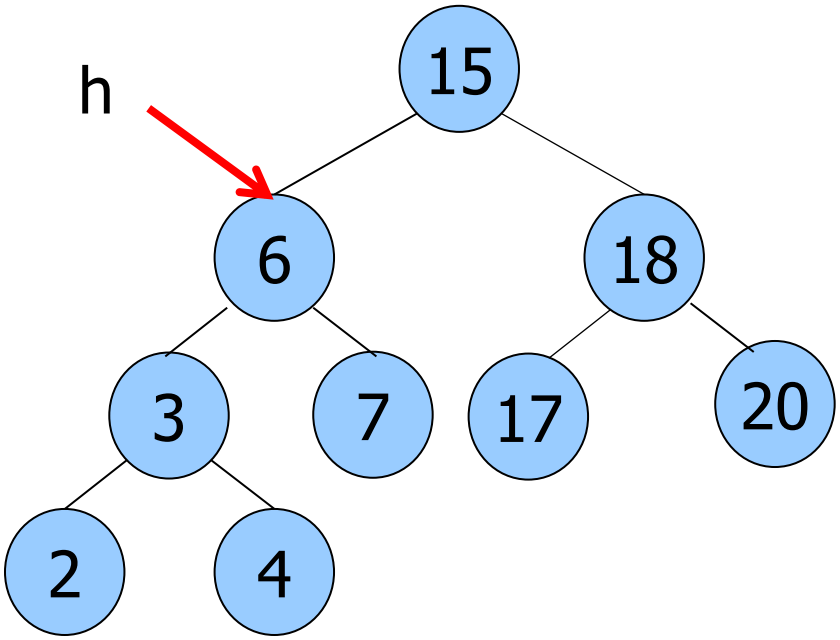
Esempio



Ricerca dell'item con chiave 7

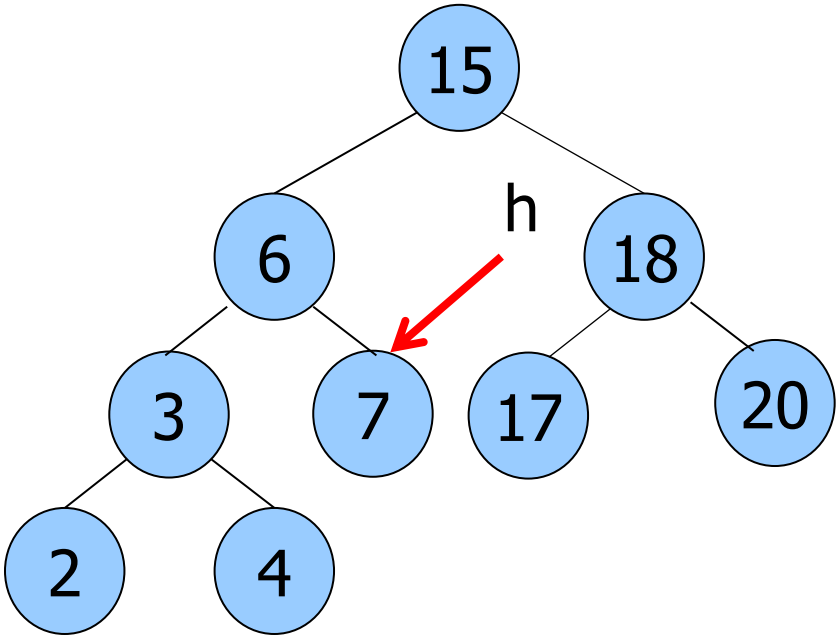
Sono visualizzate solo le chiavi intere, non gli item

Ricerca dell'item con chiave 7



Ricerca dell'item con chiave 7

Hit!



Min

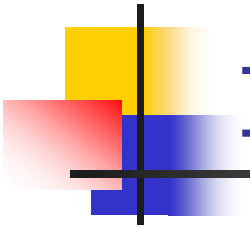
- seguire il puntatore al sottoalbero sinistro finchè esiste

```
Item minR(link h, link z) {
  if (h == z) return ITEMsetvoid();
  if (h->l == z) return (h->item);
  return minR(h->l, z);
}

Item BSTmin(BST bst) {
  return minR(bst->head, bst->z);
}
```

Max

- seguire il puntatore al sottoalbero destro finchè esiste

```
Item maxR(link h, link z) {
  if (h == z) return ITEMsetvoid();
  if (h->r == z) return (h->item);
  return maxR(h->r, z);
}

Item BSTmax(BST bst) {
  return maxR(bst->head, bst->z);
}
```



Insert (in foglia)

Inserire in un albero binario di ricerca un nodo che contiene un item $\Rightarrow$ mantenimento della proprietà:

- se il BST è vuoto, creazione del nuovo albero
- inserimento ricorsivo nel sottoalbero sinistro o destro a seconda del confronto tra la chiave dell'item e quella del nodo corrente
- inserimento iterativo: prima si ricerca la posizione, poi si appende il nuovo nodo.

RICORSIVO

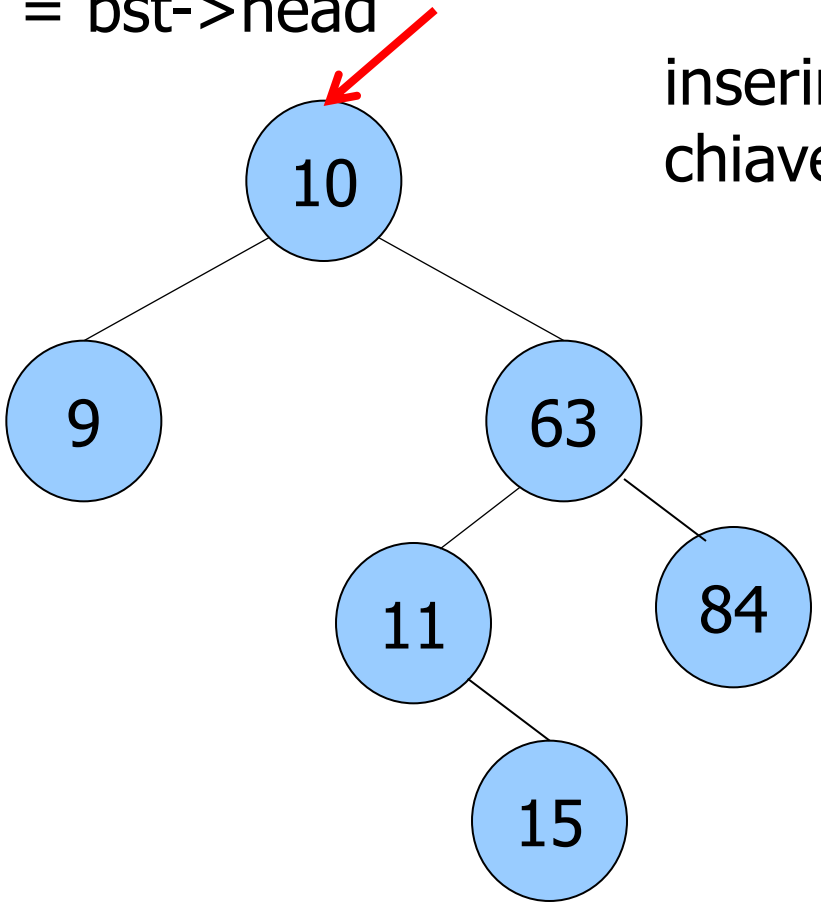
```
link insertR(link h, Item x, link z) {
  if (h == z)
    return NEW(x, z, z);
  if (ITEMless(x, h->item))
    h->l = insertR(h->l, x, z);
  else
    h->r = insertR(h->r, x, z);
  return h;
}

void BSTinsert_leafR(BST bst, Item x) {
  bst->head = insertR(bst->head, x, bst->z);
}
```

ITERATIVO

```
void BSTinsert_leafI(BST bst, Item x) {
  link p = bst->head, h = p;
  if (bst->head == bst->z) {
    bst->head = NEW(x, bst->z, bst->z);
    return;
  }
  while (h != bst->z) {
    p = h;
    h = (ITEMless(x, h->item)) ? h->l : h->r;
  }
  h = NEW(x, bst->z, bst->z);
  if (ITEMless(x, p->item))
    p->l = h;
  else
    p->r = h;
}
```

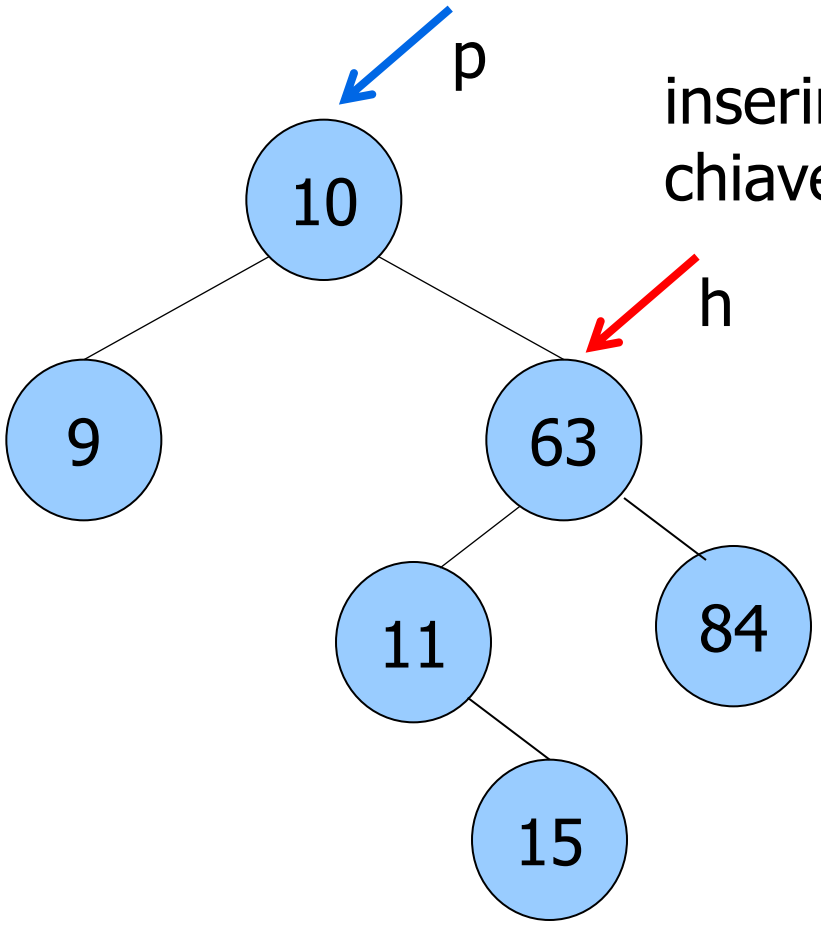
Esempio

h =p = bst->head



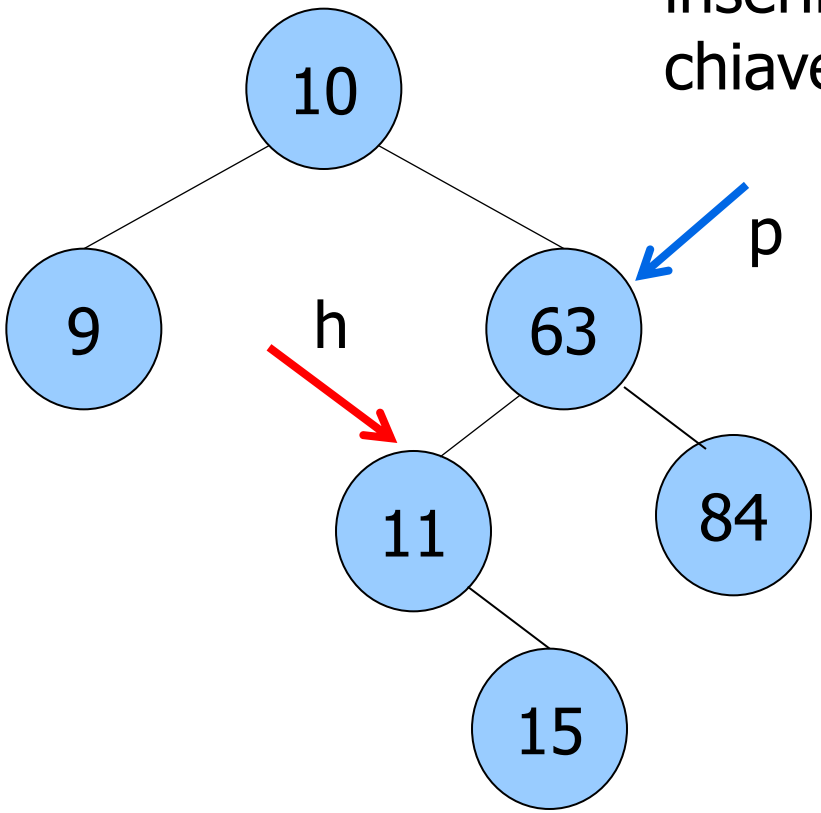
inserimento dell'item con chiave 62

NB: per semplicità si riporta solo il campo chiave dell'item

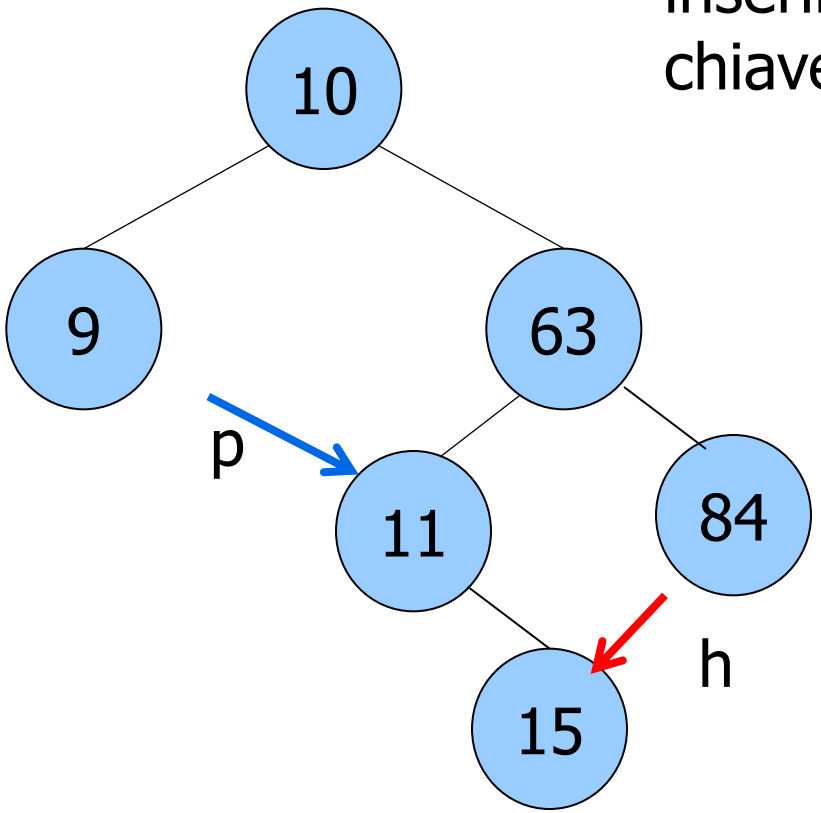


inserimento dell'item con chiave 62

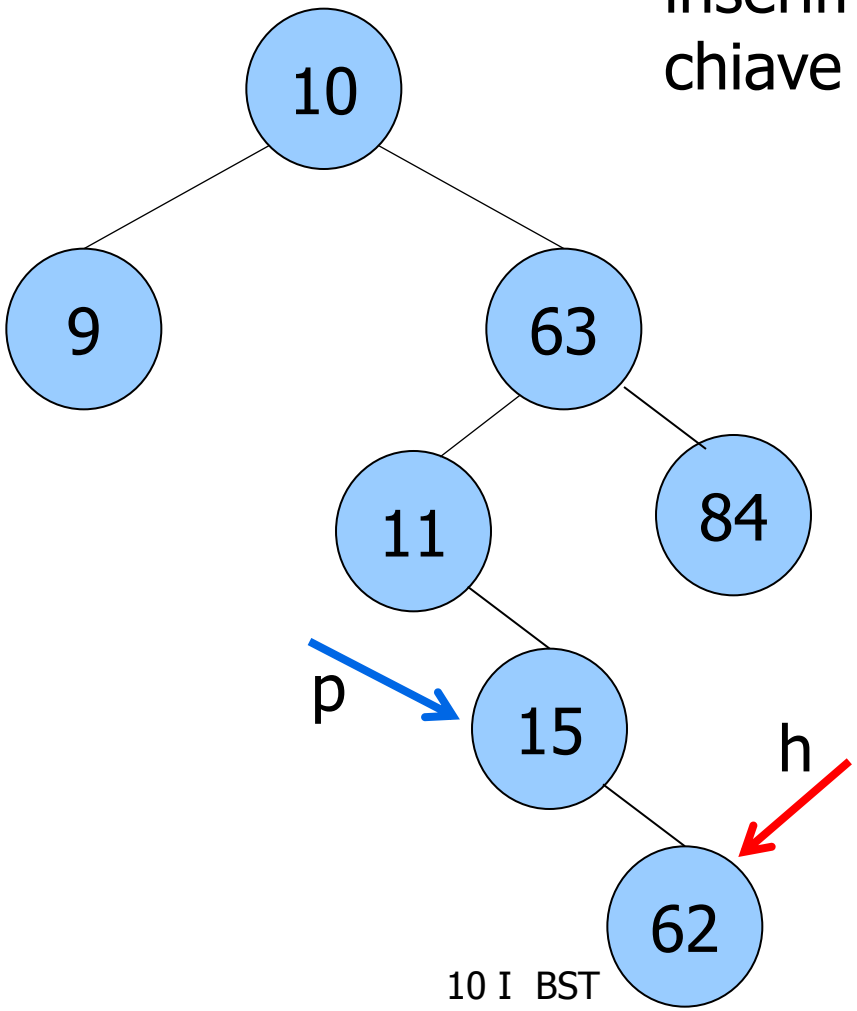
inserimento dell'item con chiave 62



inserimento dell'item con chiave 62

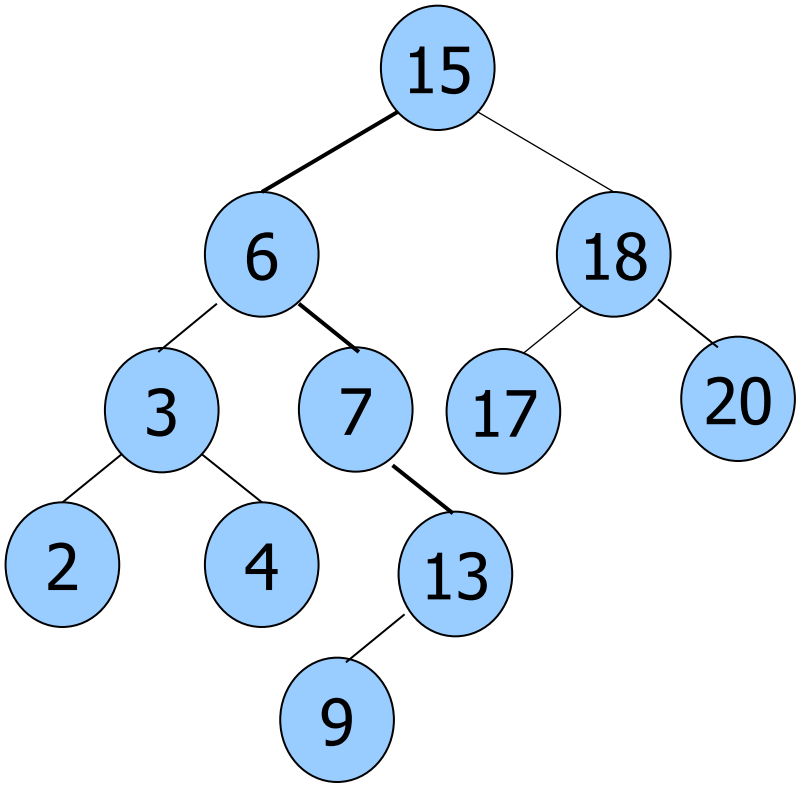


inserimento dell'item con chiave 62



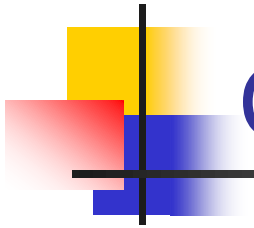
Sort

Attraversamento in-ordine: ordinamento crescente delle chiavi.



2 3 4 6 7 9 13 15 17 18 20

La chiave mediana (inferiore) di un insieme di n elementi è l'elemento che si trova in posizione $\lfloor (n + 1)/2 \rfloor$ nella sequenza ordinata degli elementi dell'insieme

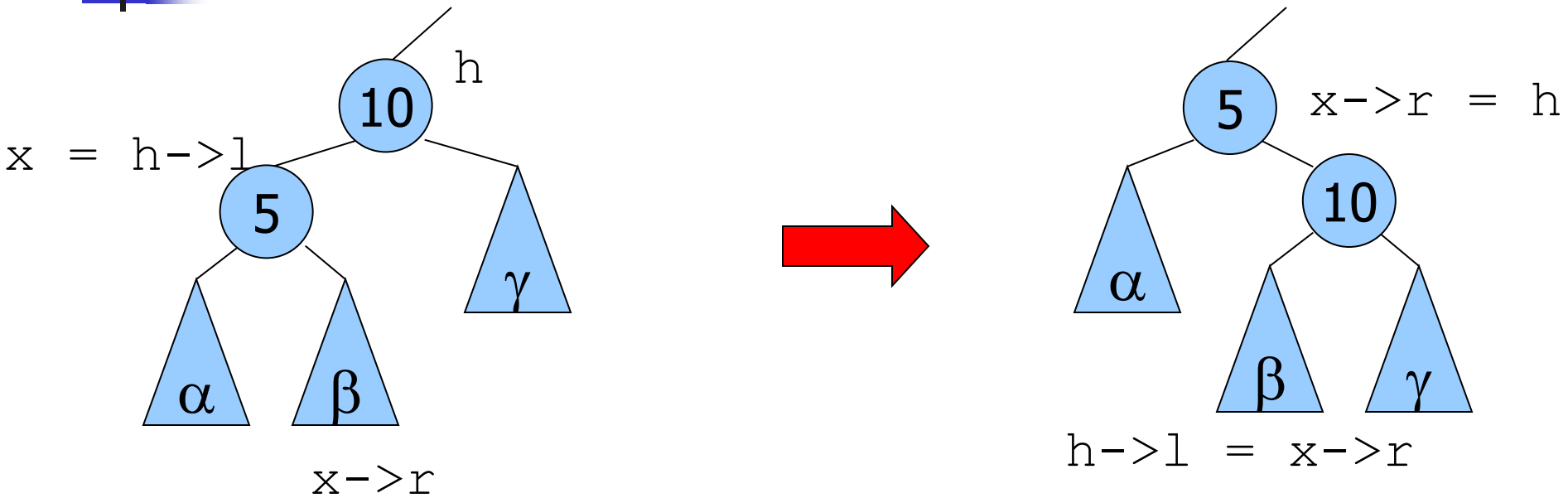


Complessità

Le operazioni hanno complessità $T(n) = O(h)$:

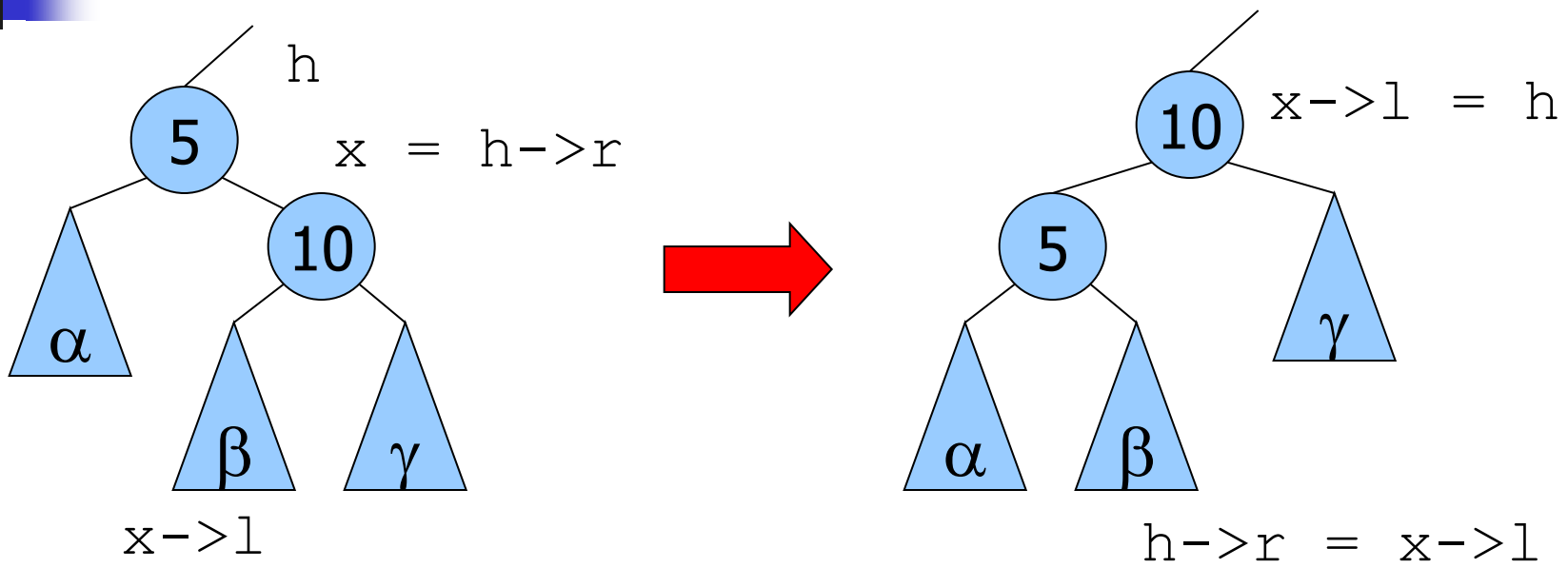
- albero con n nodi completamente bilanciato
  - altezza $h = \alpha(\log_2 n)$
- albero con n nodi completamente sbilanciato ha
  - altezza $h = \alpha(n)$
- $O(\log n) \leq T(n) \leq O(n)$

Rotazione a destra di BST



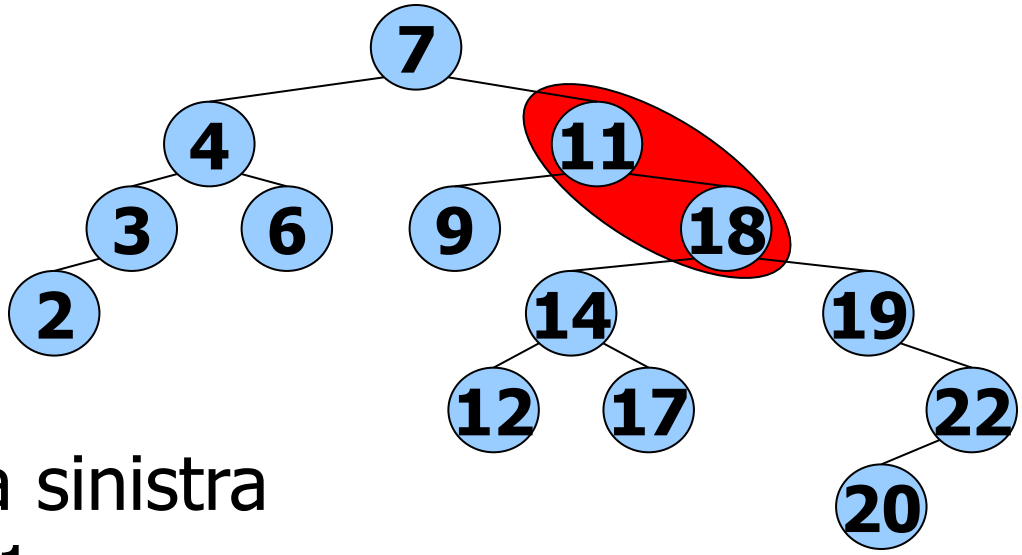

```
link rotR(link h) {
  link x = h->l;
  h->l = x->r;
  x->r = h;
  return x;
}
```

Rotazione a sinistra di BST

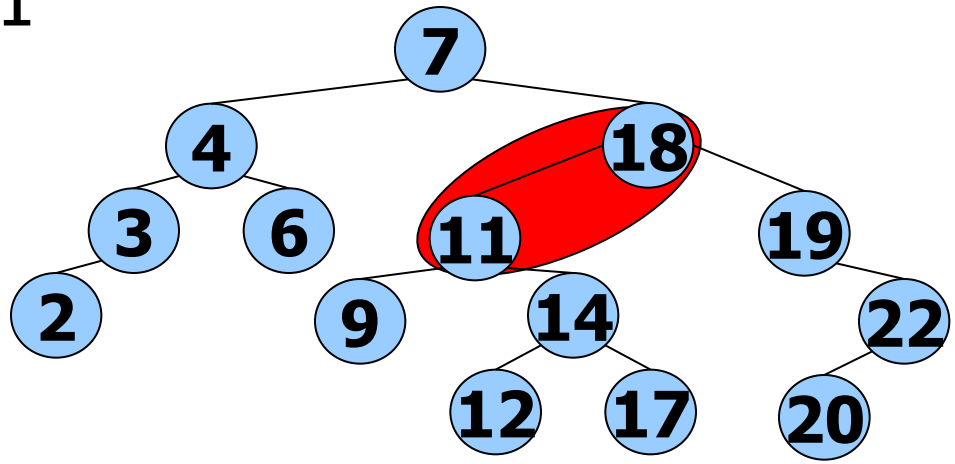


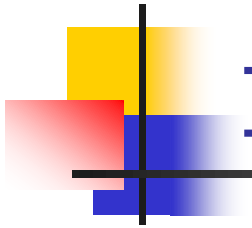

```
link rotL(link h) {
  link x = h->r;
  h->r = x->l;
  x->l = h;
  return x;
}
```

Esempio



Rotazione a sinistra attorno a 11



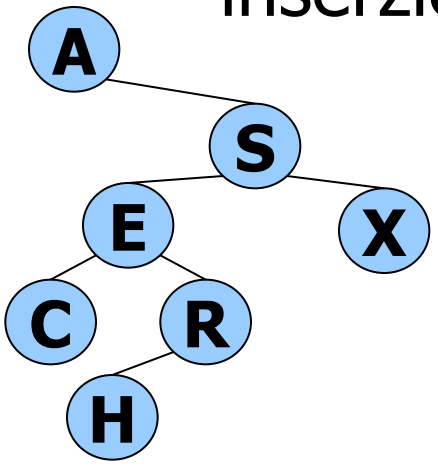


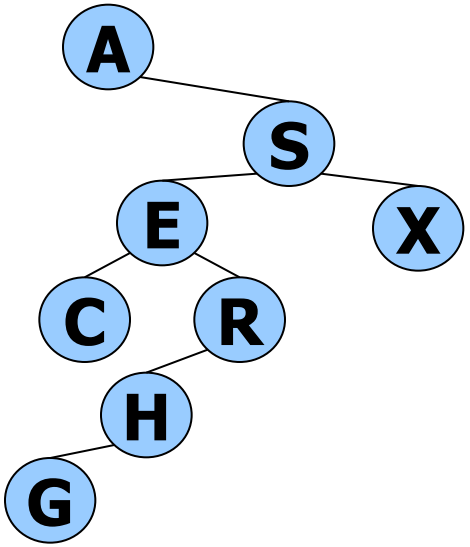
Inserimento alla radice

- Inserimento dalle foglie a scelta e non obbligatorio
- Nodi più recenti nella parte alta del BST
- Inserimento ricorsivo alla radice:
  - inserimento nel sottoalbero appropriato
  - rotazione per farlo diventare radice dell'albero principale.

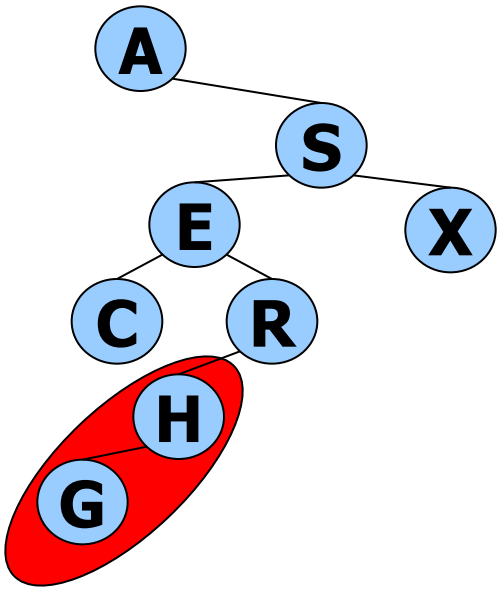
Esempio

inserzione di G

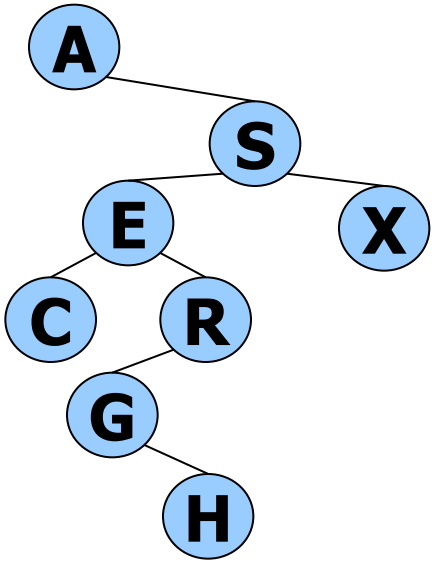


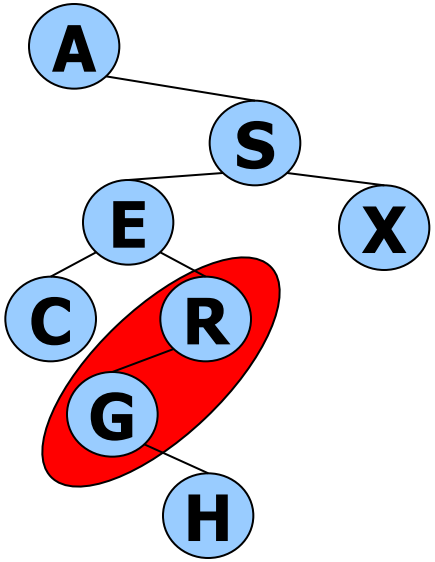


inserzione di G alla radice del sottoalbero opportuno

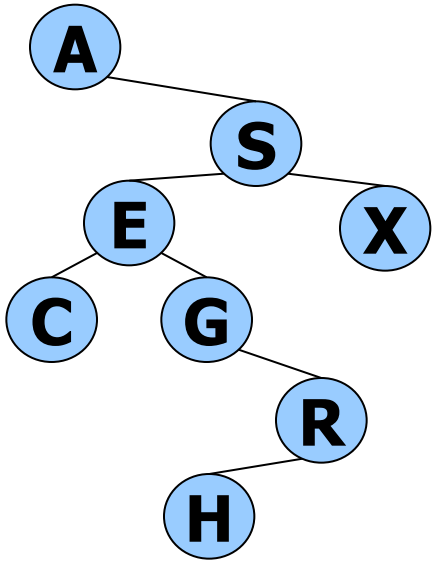


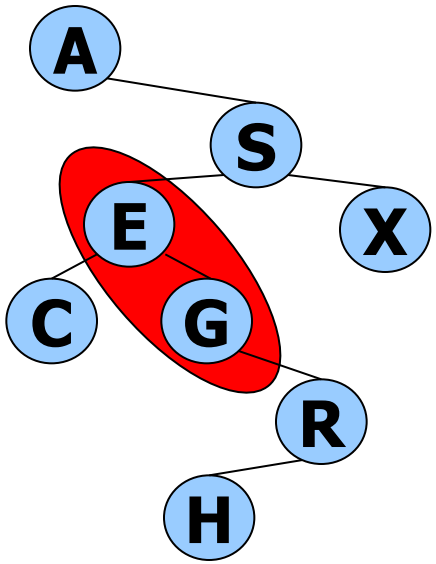
rotazione a DX



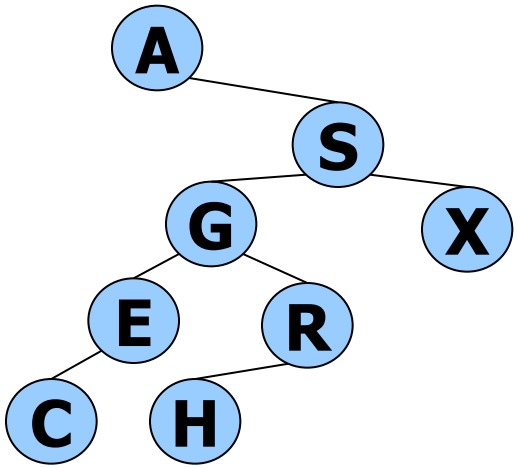


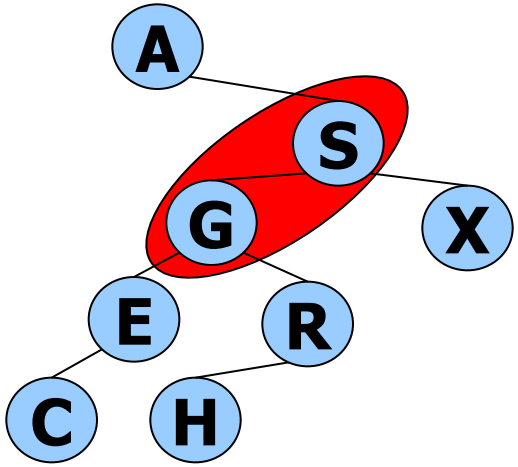
rotazione a DX



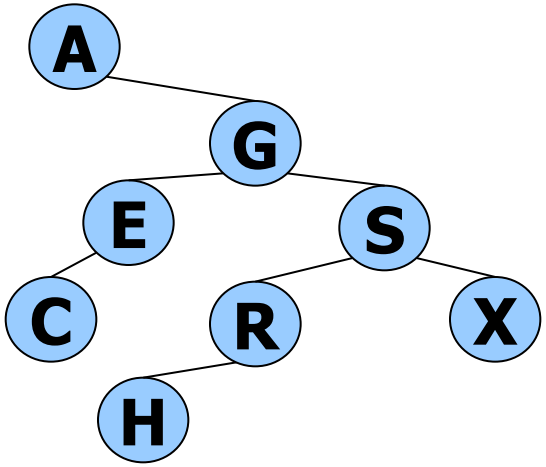


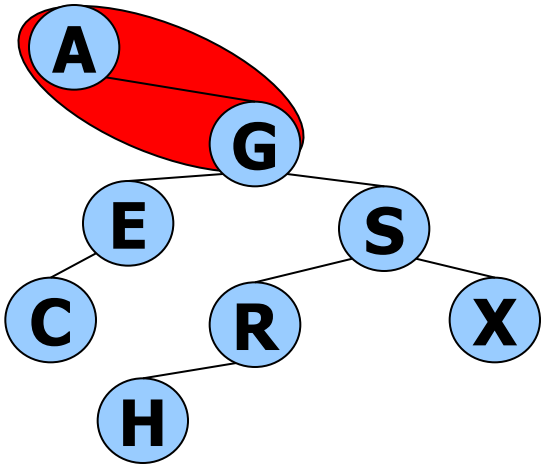
rotazione a SX



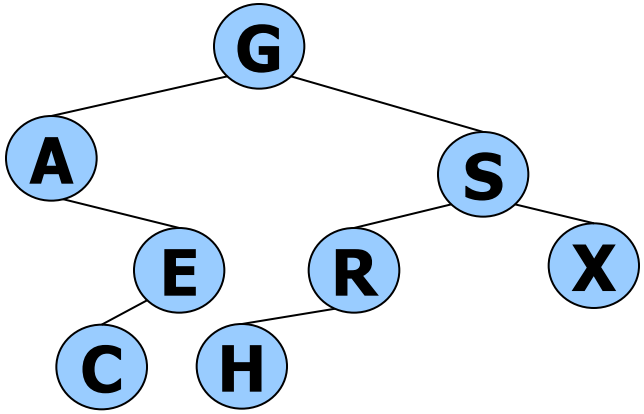


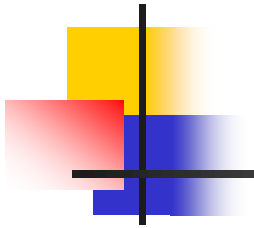
rotazione a DX





rotazione a SX





```
link insertT(link h, Item x, link z) {
  if ( h == z)
    return NEW(x, z, z );
  if (ITEMless(x, h->item)) {
    h->l = insertT(h->l, x, z);
    h = rotR(h);
  }
  else  {
    h->r = insertT(h->r, x, z);
    h = rotL(h);
  }
  return h;
}

void BSTinsert_root(BST bst, Item x) {
    bst->head = insertT(bst->head, x, bst->z);
}
```

Estensioni dei BST elementari

Al nodo elementare si possono aggiungere informazioni che permettono lo sviluppo semplice di nuove funzioni:

- puntatore al padre
- numero di nodi dell'albero radicato nel nodo corrente.

Queste informazioni devono ovviamente essere gestite (quando necessario) da tutte le funzioni già viste.

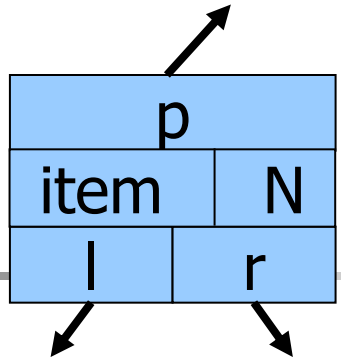
ADT di I categoria BST

nuove funzioni
funzioni modificate

BST.h

```
typedef struct  binarysearchtree *BST;
BST    BSTinit() ;
int    BSTcount(BST) ;
int    BSTempty(BST) ;
Item   BSTmin(BST) ;
Item    BSTmax(BST) ;
void   BSTinsert_leafI(BST,Item) ;
void   BSTinsert_leafR(BST,Item) ;
void   BSTinsert_root(BST,Item) ;
Item   BSTsearch(BST,Key) ;
void   BSTdelete(BST,Key) ;
Item   BSTselect(BST,int) ;
void   BSTsortinorder(BST) ;
void   BSTsortpreorder(BST ;
void   BSTsortpostorder(BST) ;
Item   BSTsucc(BST,Key) ;
Item   BSTpred(BST,Key) ;
```

Order-Statistic BST



BSTnode

BST.c

puntatore al padre

dimensione sottoalbero

```
#include <stdlib.h>
#include "Item.h"
#include "BST.h"
typedef struct BSTnode* link;
struct BSTnode {Item item;link p;link l;link  r;int N;};
struct binarysearchtree { link head; int N; link z; };
link NEW(Item item, link p, link l, link r, int N){
  link x = malloc(sizeof *x);
  x->item = item;
  x->p = p; x->l = l; x->r = r; x->N = N;
  return x;
}
BST BSTinit( ) {
  BST bst = malloc(sizeof *bst) ;
  bst->N = 0;
  bst->head =(bst->z=NEW(ITEMsetvoid(), NULL, NULL, NULL, 0));
  return bst;
}
```

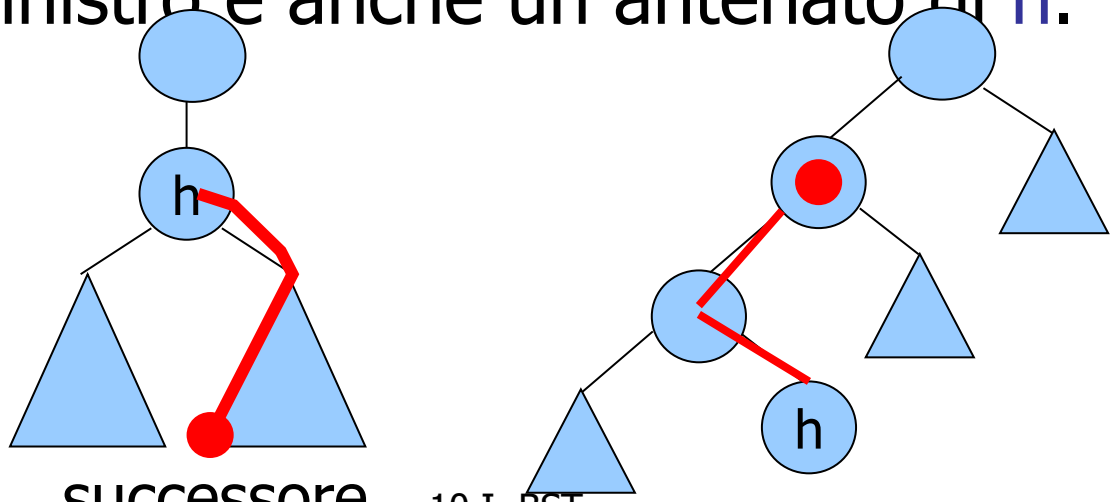
```
int BSTcount(BST bst) {
  return bst->N;
}

int BSTempty(BST bst) {
  if ( BSTcount(bst) == 0)
    return 1;
  else
    return 0;
}
```

Successore

Successore di un item: se esiste, nodo h con un item con la più piccola chiave > della chiave di item, altrimenti item vuoto

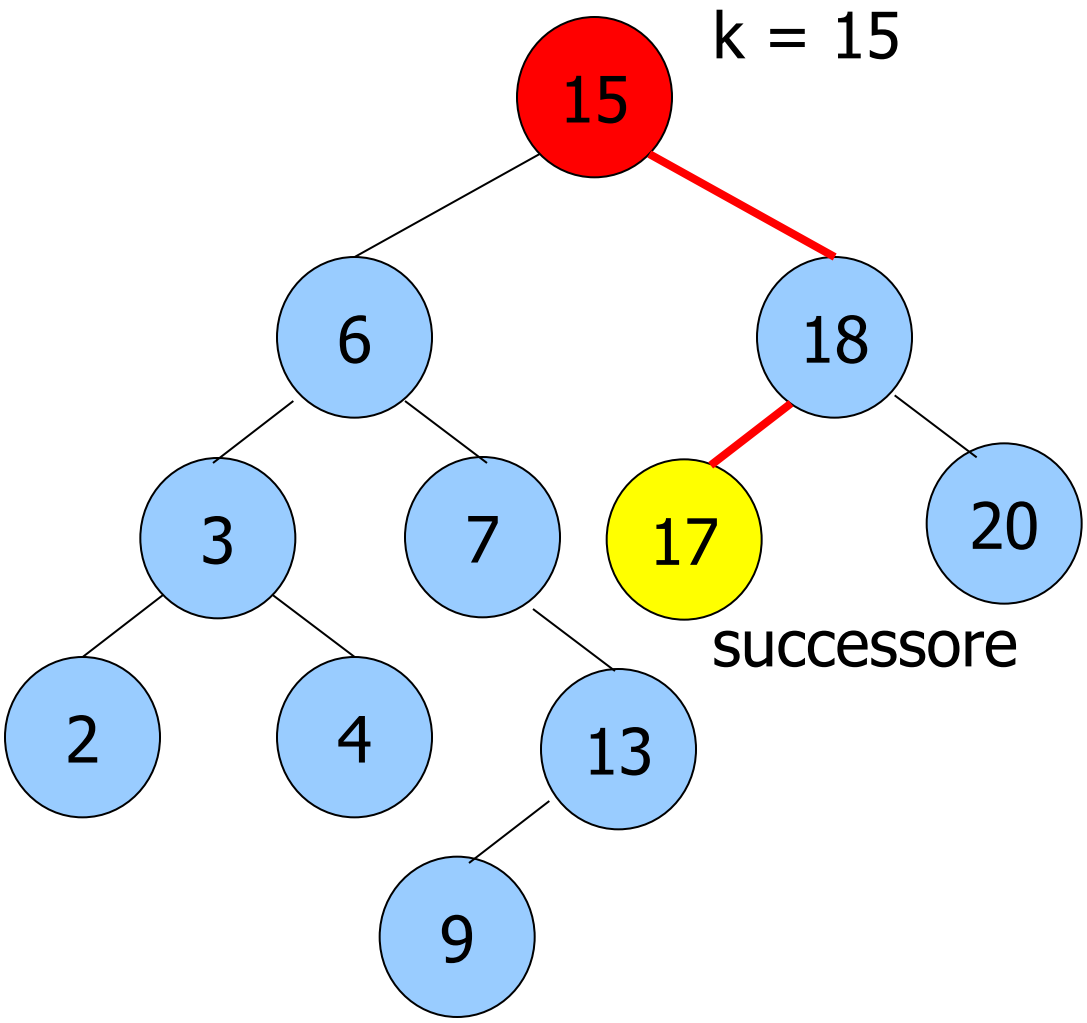
Due casi:

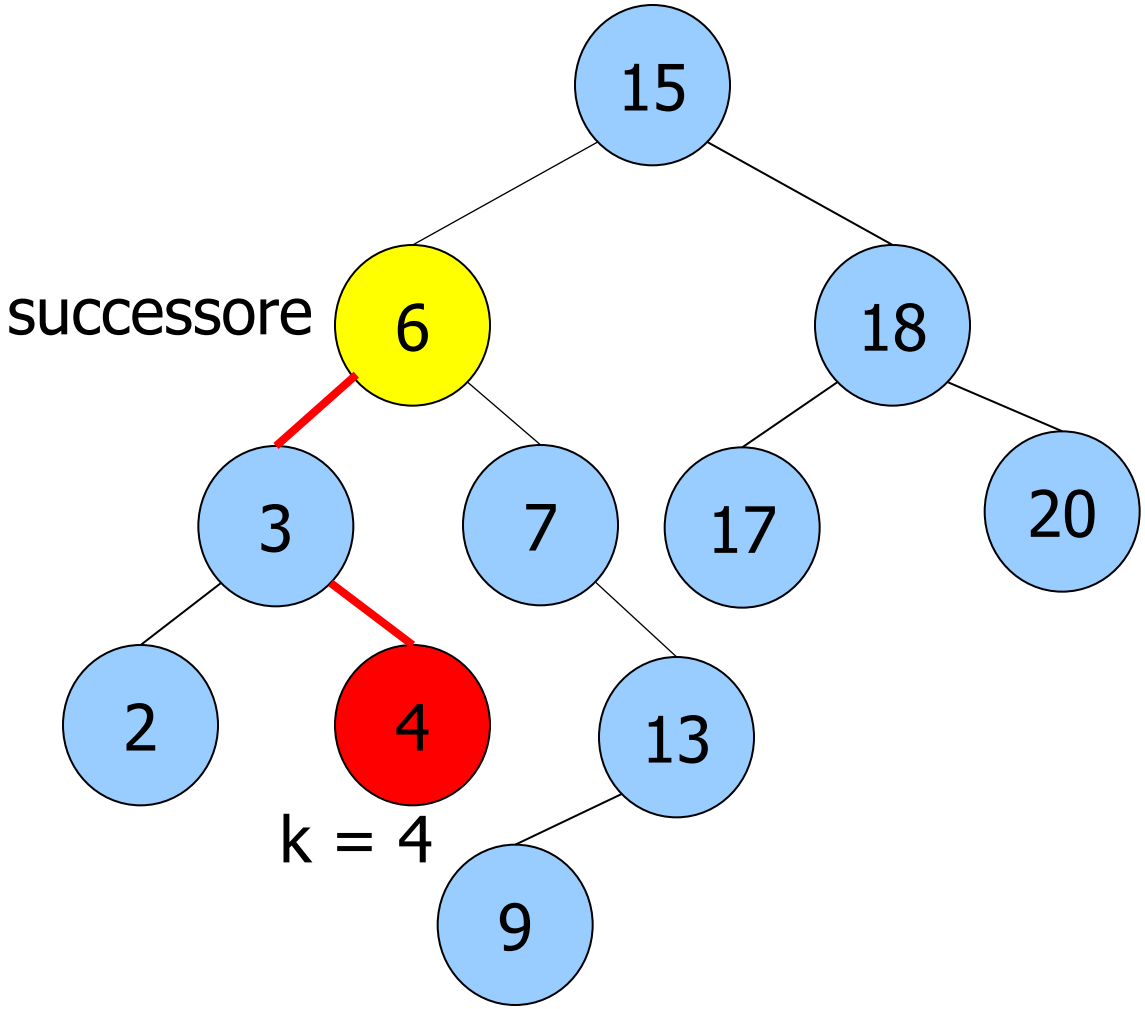
- ∃ Right(h): succ(key(h)) = min(Right(h))
- ∃ Right(h): succ(key(h)) = primo antenato di h il cui figlio sinistro è anche un antenato di h.



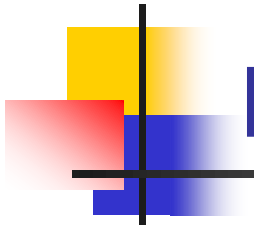
successore

Esempio




15
successore
6
18
3
7
17
20
2
4
13
k = 4
9

```
Item searchSucc(link h, Key k, link z) {
  link p;
  if (h == z) return ITEMsetvoid();
  if (KEYcompare(k, KEYget(h->item))==0) {
    if (h->r != z) return minR(h->r, z);
    else {
      p = h->p;
      while (p != z && h == p->r) {
        h = p;
        p = p->p;
      }
      return p->item;
    }
  }
  if (KEYcompare(k, KEYget(h->item))==-1)
    return searchSucc(h->l, k, z);
  else return searchSucc(h->r, k, z);
}
Item BSTsucc(BST bst, Key k) {
  return searchSucc(bst->head, k, bst->z);
}
```



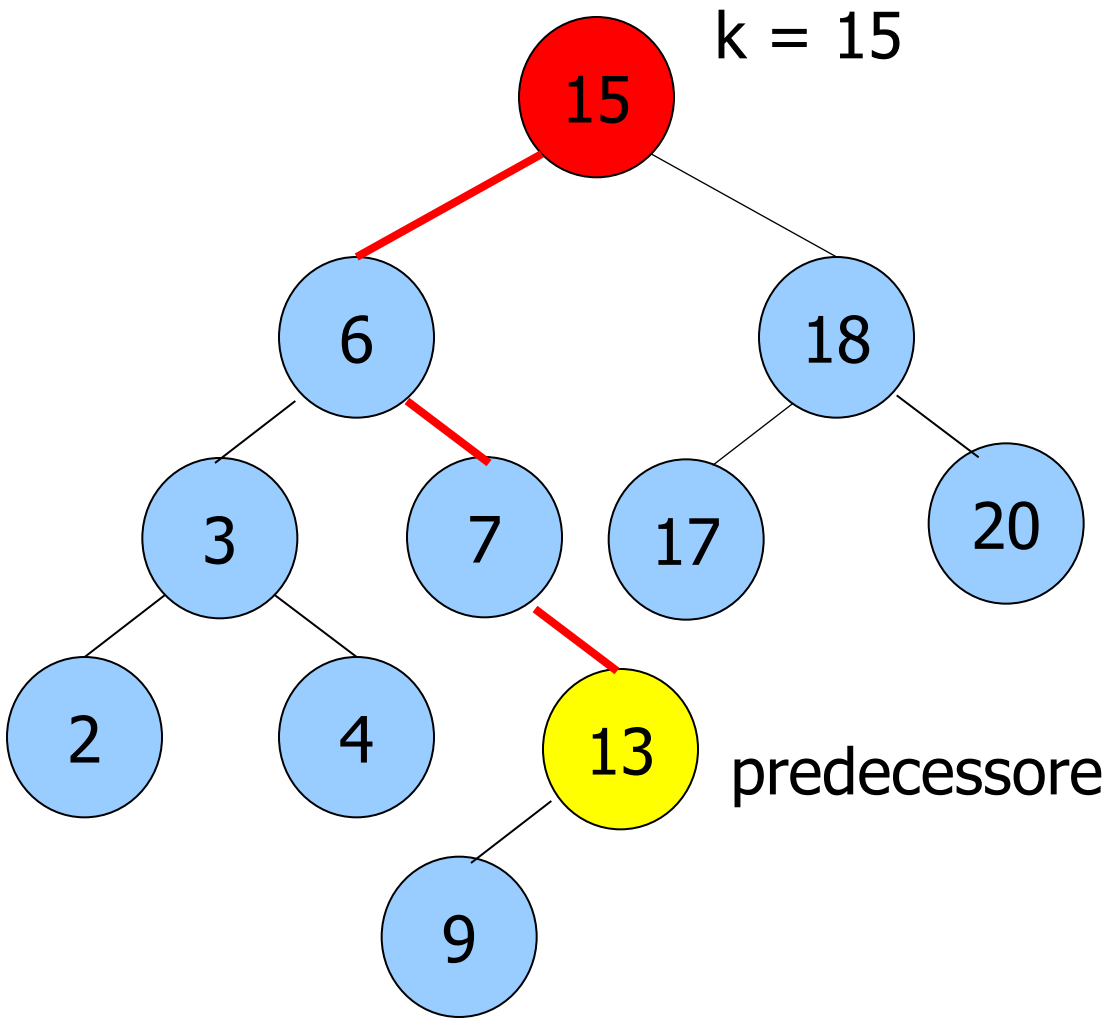
Predecessore

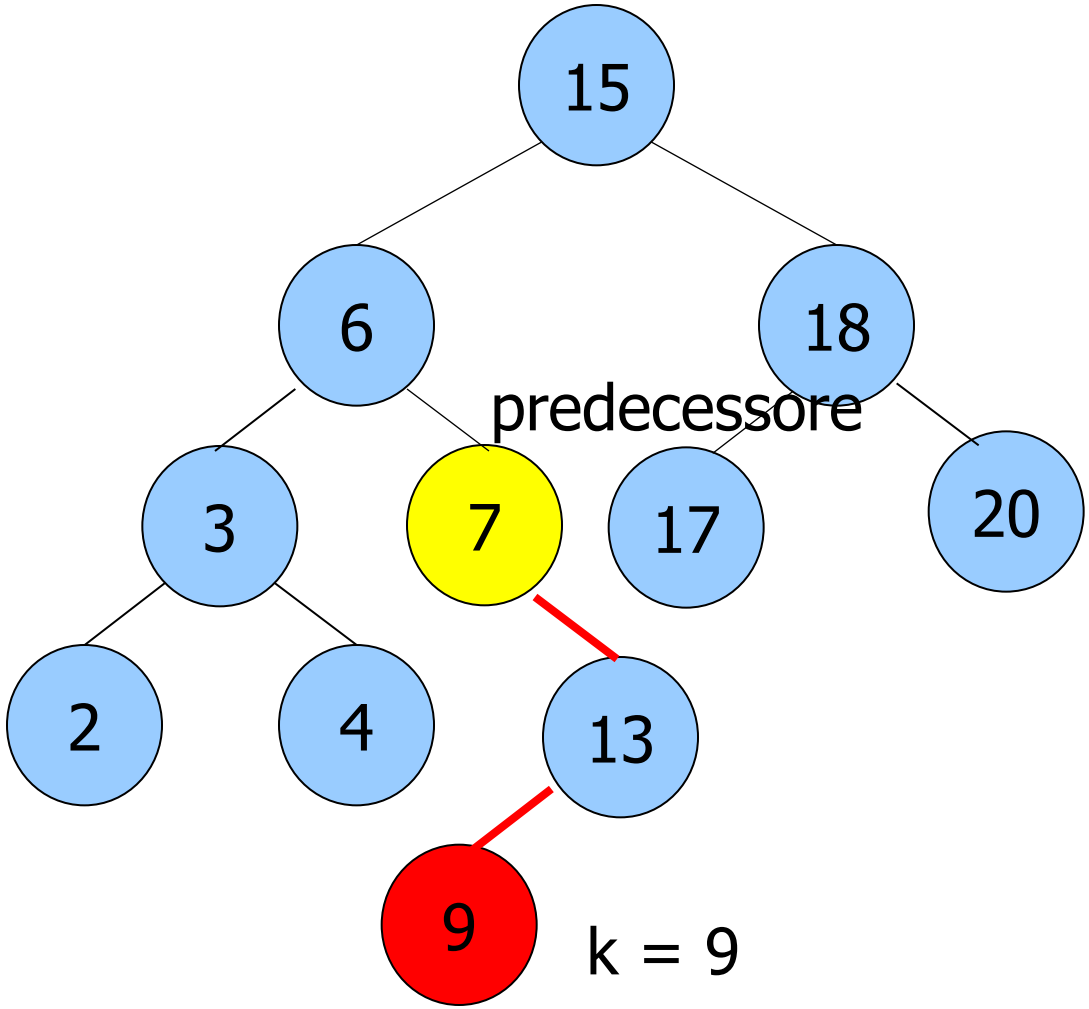
Predecessore di un item: nodo h con item con la più grande chiave < della chiave di item.

Due casi:

- $\exists$ Left(h): pred(key(h)) = max(Left(h))
- $\nexists$ Left(h): pred(key(h)) = primo antenato di h il cui figlio destro è anche un antenato di h.

Esempio




15
6
18
predecessore
3
7
17
20
2
4
13
9
k = 9

```
Item searchPred(link h, Key k, link z) {
  link p;
  if (h == z) return ITEMsetvoid();
  if (KEYcompare(k, KEYget(h->item))==0) {
    if (h->l != z) return maxR(h->l, z);
    else {
      p = h->p;
      while (p != z && h == p->l) {
        h = p;
        p = p->p;
      }
      return p->item;
    }
  }
  if (KEYcompare(k, KEYget(h->item))==-1)
    return searchPred(h->l, k, z);
  else return searchPred(h->r, k, z);
}
Item BSTpred(BST bst, Key k) {
  return searchPred(bst->head, k, bst->z);
}
```

Insert (in foglia)

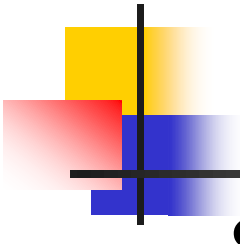
RICORSIVO

```
link insertR(link h, Item x, link z) {
  if (h == z)
    return NEW(x, z, z, z, 1);
  if (ITEMless(x, h->item)) {
    h->l = insertR(h->l, x, z);
    h->l->p = h;
  }
  else {
    h->r = insertR(h->r, x, z);
    h->r->p = h;
  }
  (h->N)++;
  return h;
}

void BSTinsert_leafR(BST bst, Item x) {
  bst->head = insertR(bst->head, x, bst->z);
  bst->N++;
}
```

Insert (in foglia)

ITERATIVO

```
void BSTinsert_leafI(BST bst, Item x) {
  link p = bst->head, h = p;
  if (bst->head == bst->z) {
    bst->head = NEW(x, bst->z, bst->z, bst->z, 1);
    bst->N++;
    return;
  }
  while (h != bst->z) {
    p = h;
    h->N++;
    h = (ITEMless(x, h->item)) ? h->l : h->r;
  }
  h = NEW(x, p, bst->z, bst->z, 1);
  bst->N++;
  if (ITEMless(x, p->item))
    p->l = h;
  else
    p->r = h;
}
```

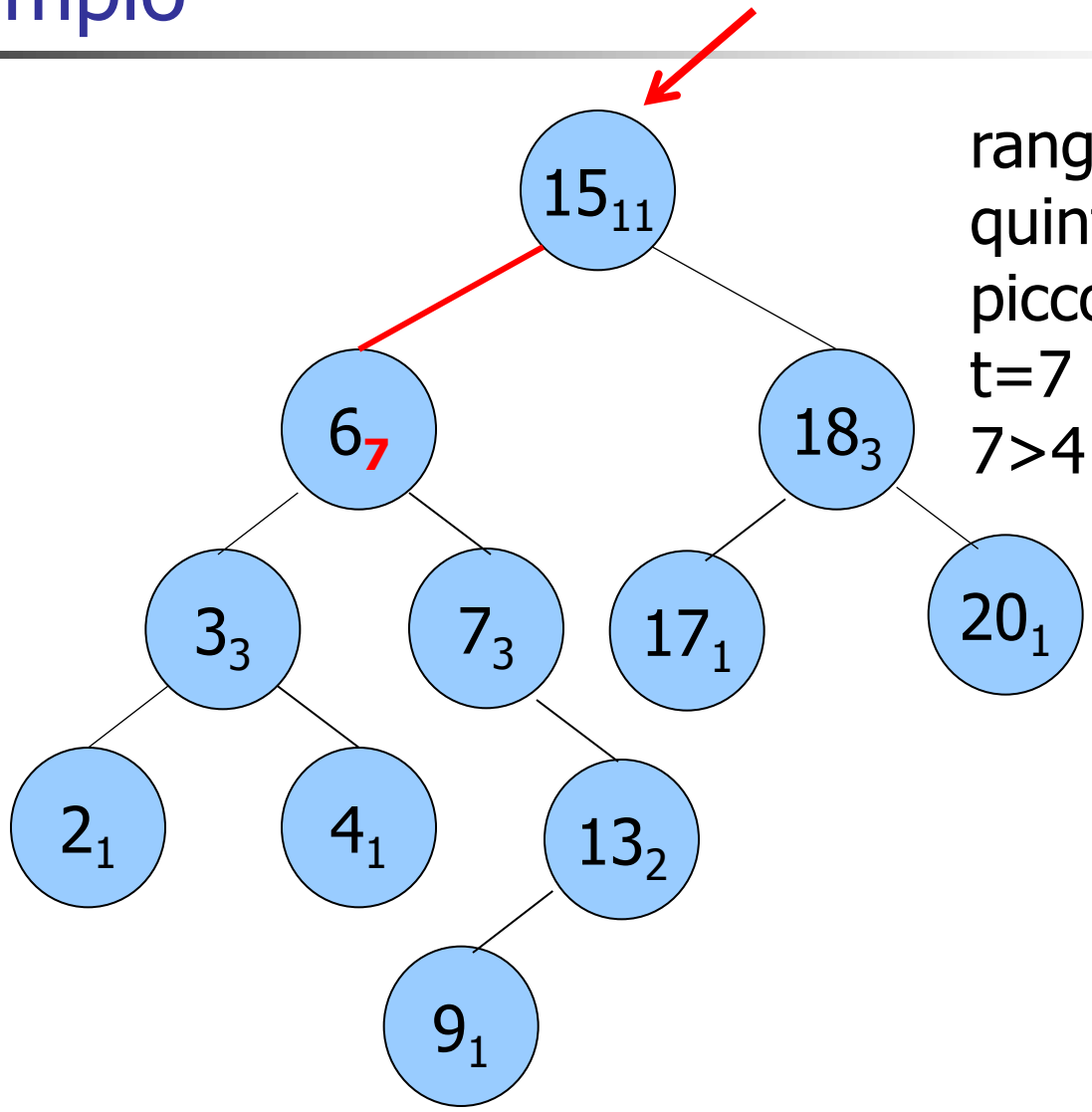


Select

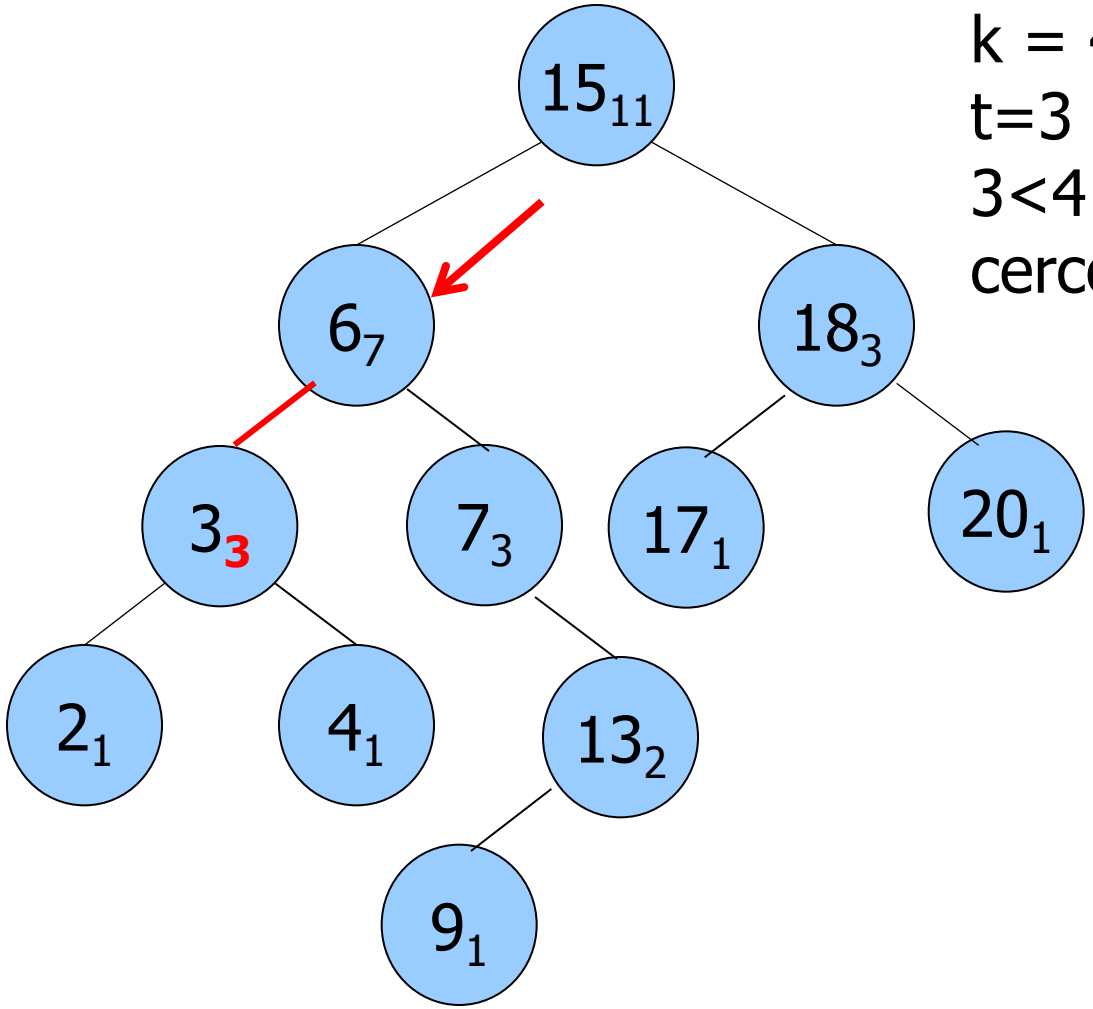
Selezione dell'item con la k-esima chiave più piccola (rango k = chiave in posizione k nell'ordinamento, ad esempio se k=0 item con chiave minima): t è il numero di nodi del sottoalbero sinistro:

- t = k: ritorno la radice del sottoalbero
- t > k: ricorsione nel sottoalbero sinistro alla ricerca della k-esima chiave più piccola
- t < k: ricorsione nel sottoalbero destro alla ricerca della (k-t-1)-esima chiave più piccola

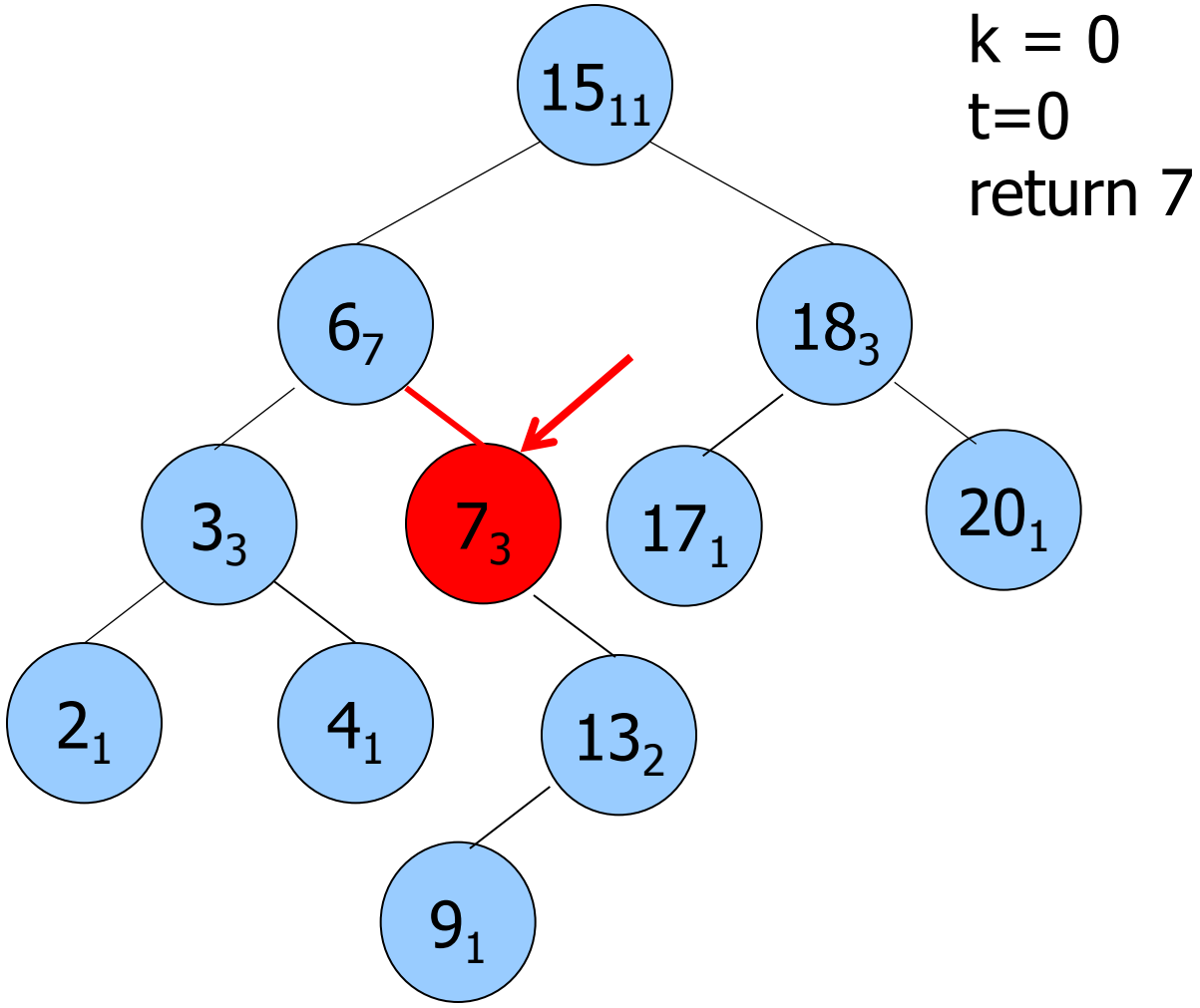
Esempio



rango = k = 4
quinta più
piccola chiave
t=7
7>4 scendo a SX

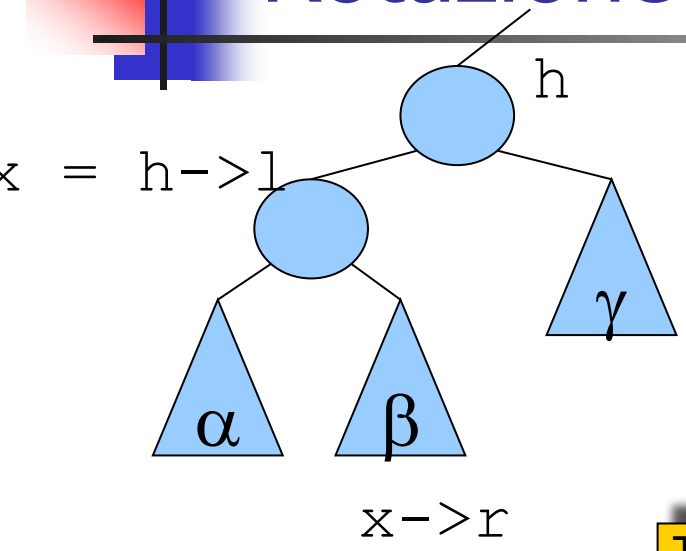


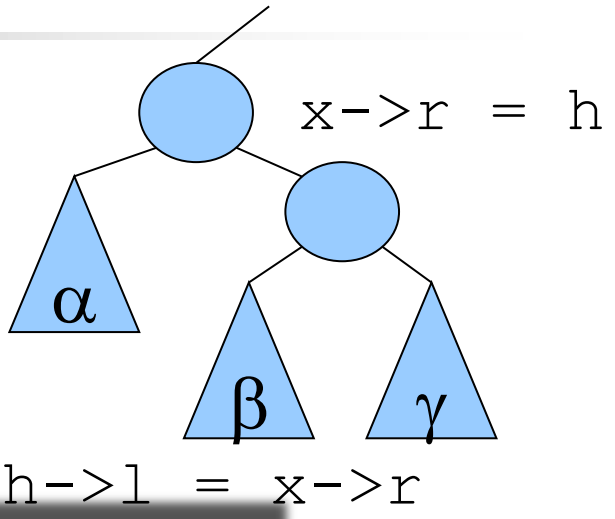
k = 4
t=3
3<4 scendo a DX
cerco k=4-3-1=0


15_{11}
6_7
18_3
3_3
7_3
17_1
20_1
2_1
4_1
13_2
9_1
k = 0
t=0
return 7

```
Item selectR(link h, int k, link z) {
  int t;
  if (h == z)
    return ITEMsetvoid();
  t = (h->l == z) ? 0 : h->l->N;
  if (t > k)
    return selectR(h->l, k, z);
  if (t < k)
    return selectR(h->r, k-t-1, z);
  return h->item;
}

Item BSTselect(BST bst, int k) {
  return selectR(bst->head, k, bst->z);
}
```

Rotazione a destra di BST



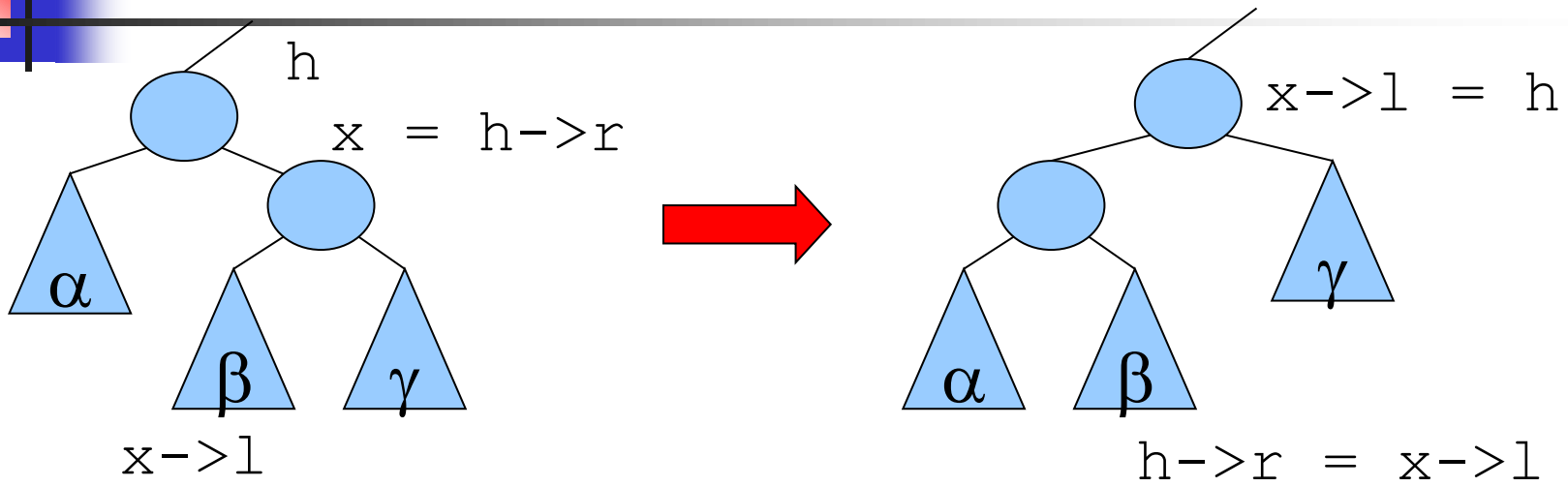



```
link rotR(link h) {
  link x = h->l;
  h->l = x->r;
  x->r->p = h;
  x->r = h;
  x->p = h->p;
  h->p = x;
  x->N = h->N;
  h->N = h->r->N + h->l->N +1;
  return x;
}
```

aggiornamento puntatore al padre

aggiornamento dimensione sottoalberi

Rotazione a sinistra di BST




```
link rotL(link h) {
  link x = h->r;
  h->r = x->l;
  x->l->p = h;
  x->l = h;
  x->p = h->p;
  h->p = x;
  x->N = h->N;
  h->N = h->l->N + h->r->N +1;
  return x;
}
```

aggiornamento puntatore al padre

aggiornamento dimensione sottoalberi

Inserimento alla radice

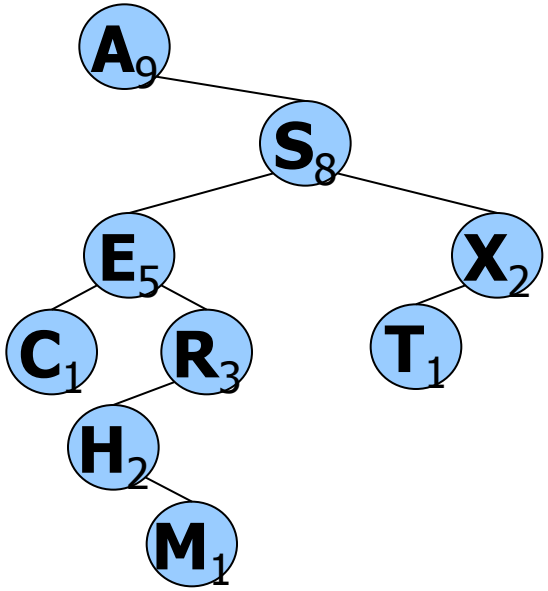
```
link insertT(link h, Item x, link z) {
  if ( h == z)
    return NEW(x, z, z, z, 1);
  if (ITEMless(x, h->item)) {
    h->l = insertT(h->l, x, z);
    h = rotR(h);
    h->N++;
  }
  else {
    h->r = insertT(h->r, x, z);
    h = rotL(h);
    h->N++;
  }
  return h;
}

void BSTinsert_root(BST bst, Item x) {
  bst->head = insertT(bst->head, x, bst->z);
  bst->N++;
}
```

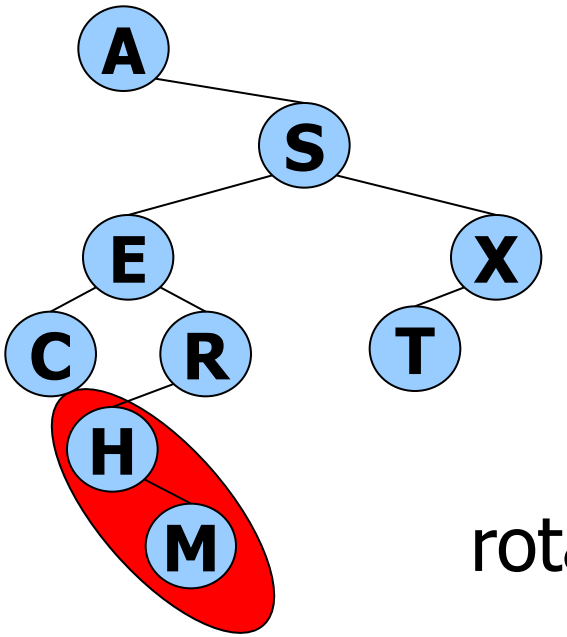
Partition

- Riorganizza l'albero avendo l'item con la k-esima chiave più piccola nella radice:
  - poni il nodo come radice di un sottoalbero:
    - t > k: ricorsione nel sottoalbero sinistro, partizionamento rispetto alla k-esima chiave più piccola, al termine rotazione a destra
    - t < k: ricorsione nel sottoalbero destro, partizionamento rispetto alla (k-t-1)-esima chiave più piccola , al termine rotazione a sinistra
  - Sovente il partizionamento si fa attorno alla chiave mediana

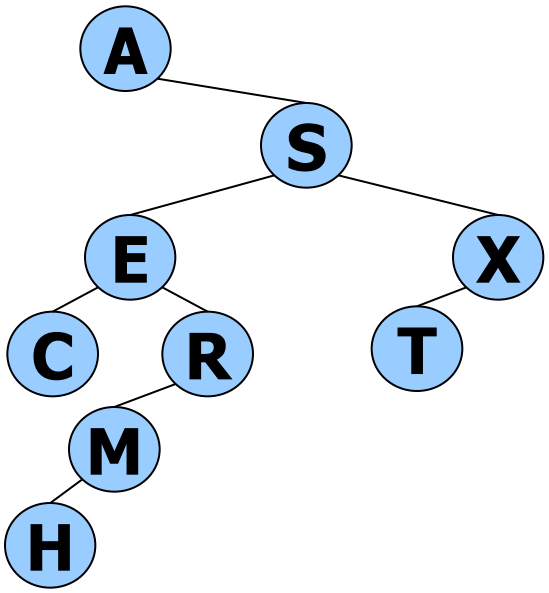
Esempio

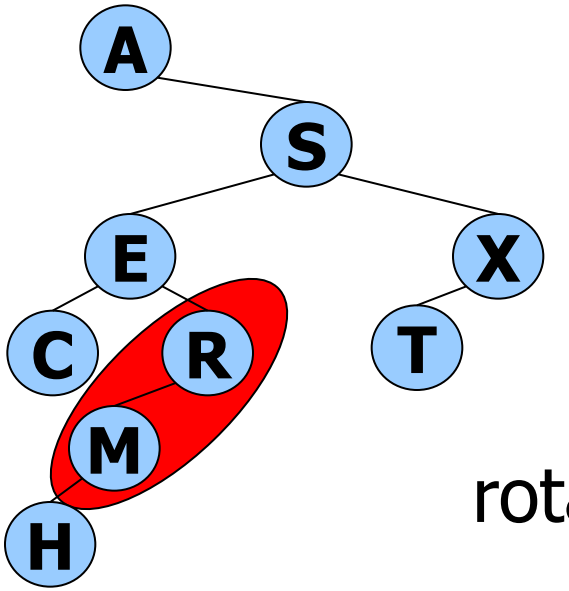
Partizionamento rispetto alla
5a chiave più piccola (M, k=4)



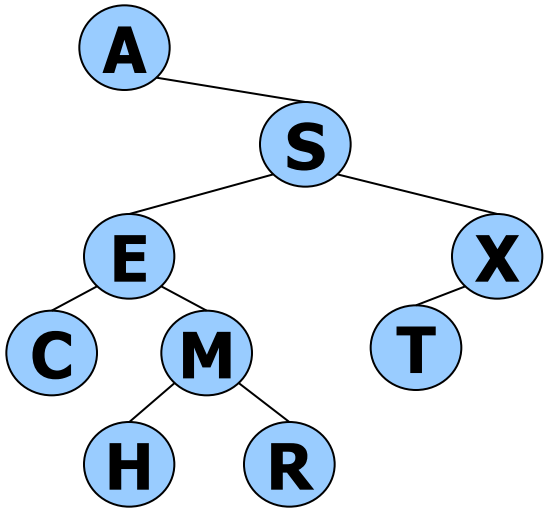


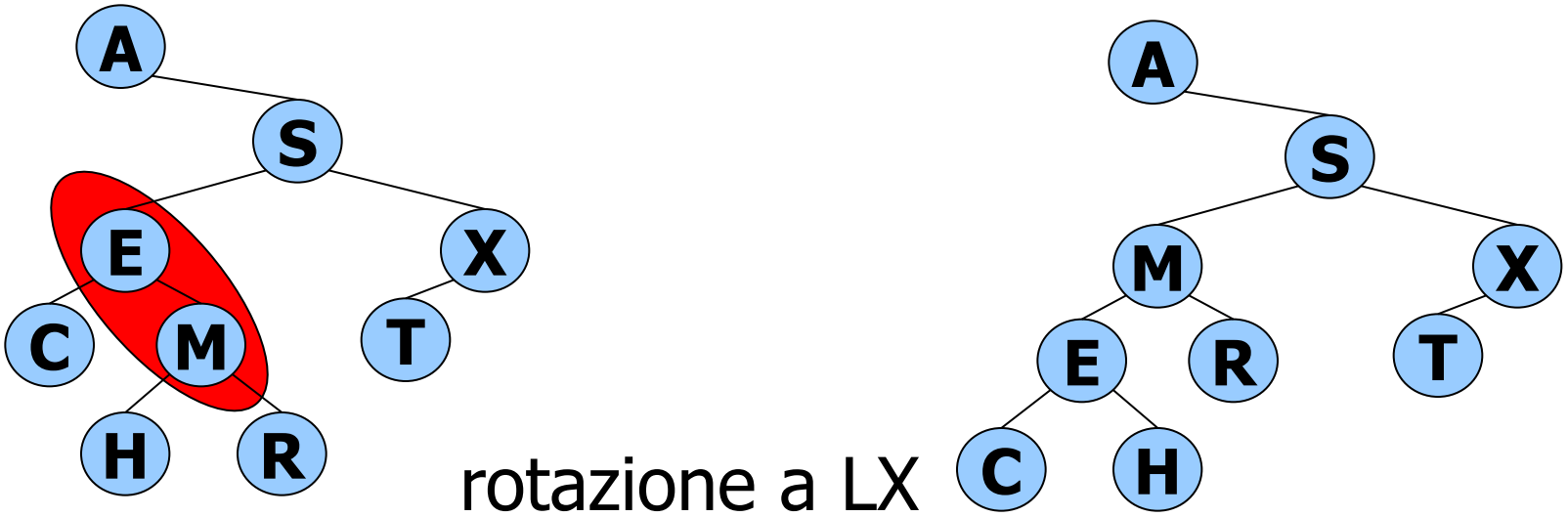
rotazione a SX

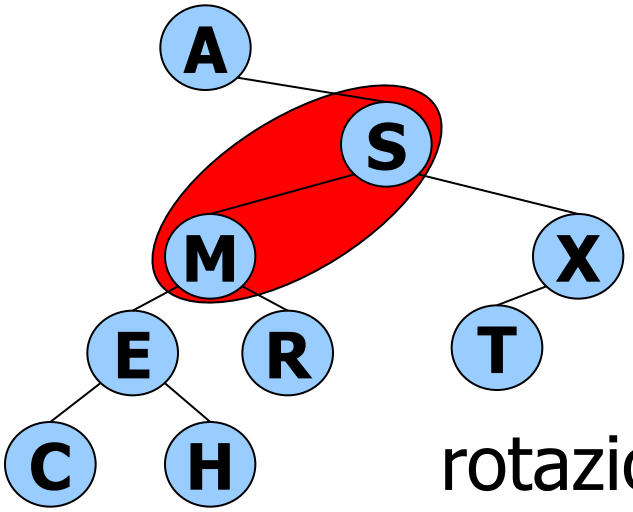




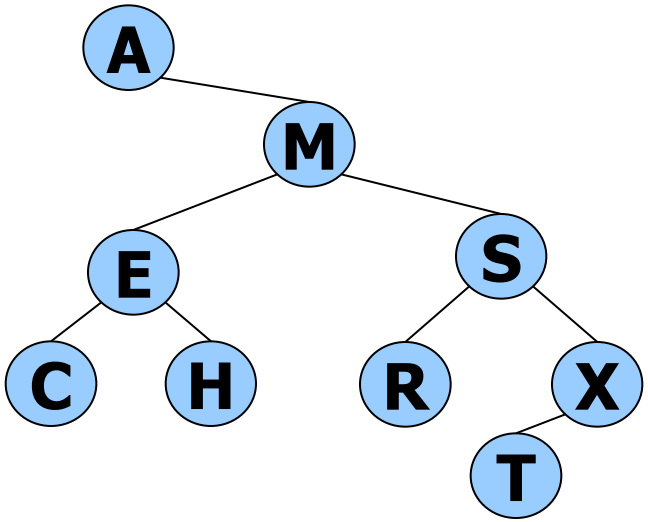
rotazione a DX

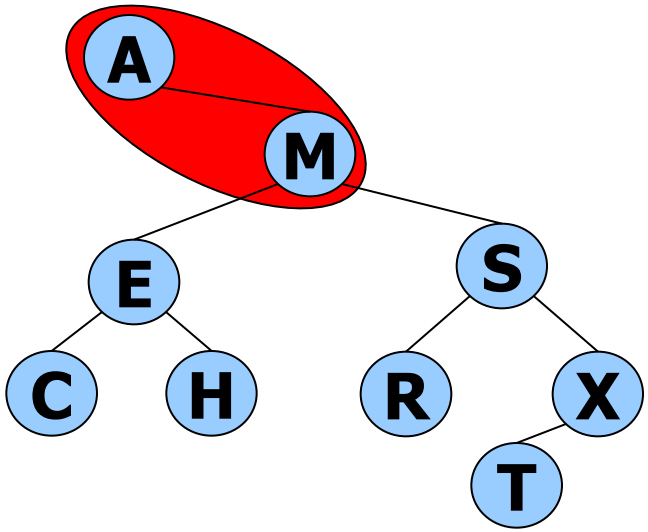


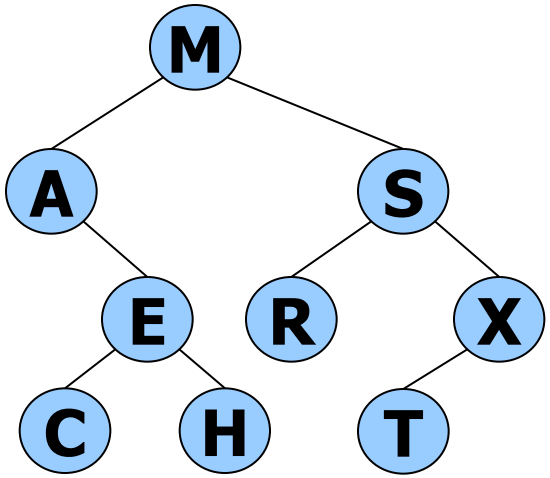

A
S
E
X
C
M
T
H
R
rotazione a LX
A
S
M
X
E
R
T
C
H



rotazione a DX

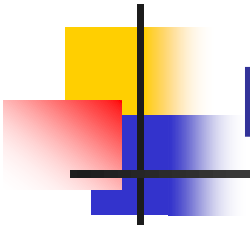






rotazione a SX

```
link partR(link h, int k) {
  int t  = h->l->N;
  if ( t > k) {
    h->l = partR(h->l, k);
    h = rotR(h);
  }
  if ( t < k) {
    h->r = partR(h->r, k-t-1);
    h = rotL(h);
  }
  return h;
}
```



Delete

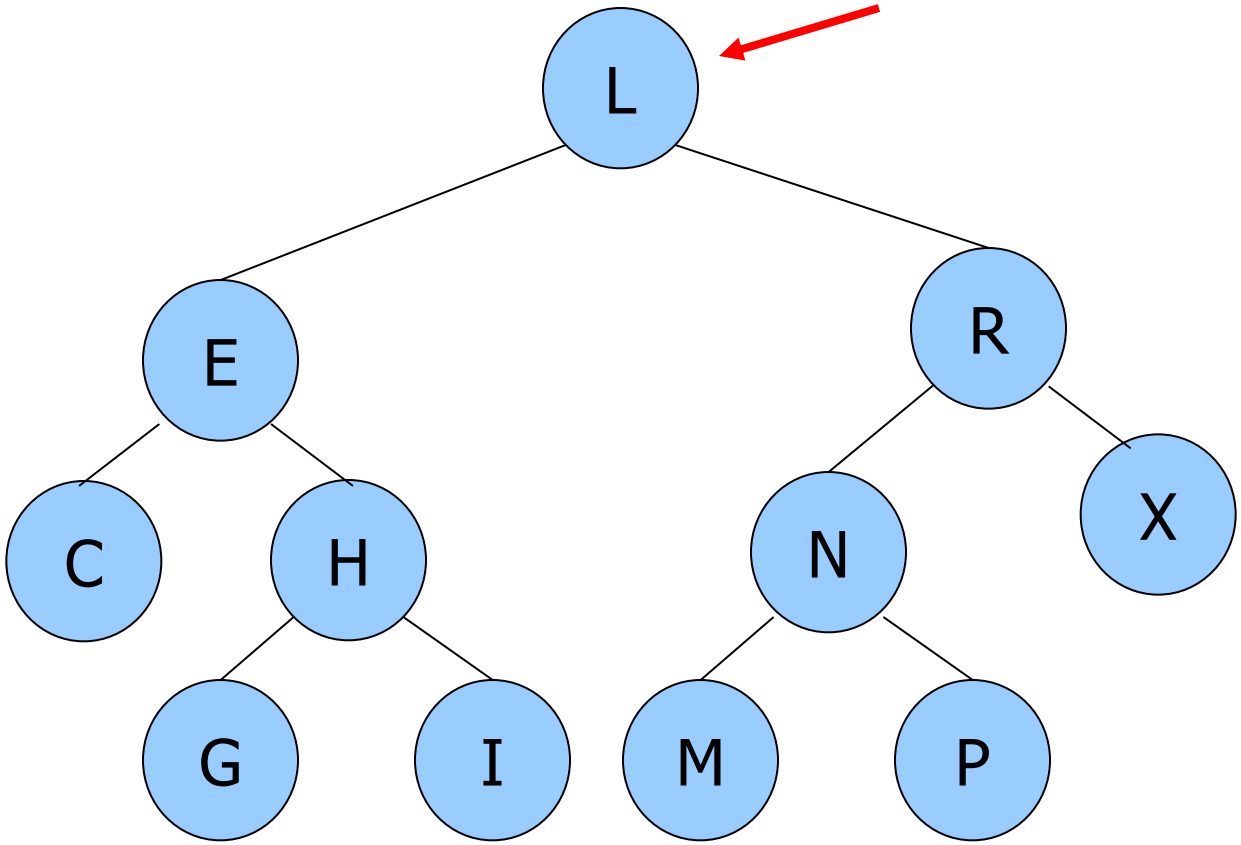
Per cancellare da un albero binario di ricerca un nodo con item con chiave k bisogna mantenere:

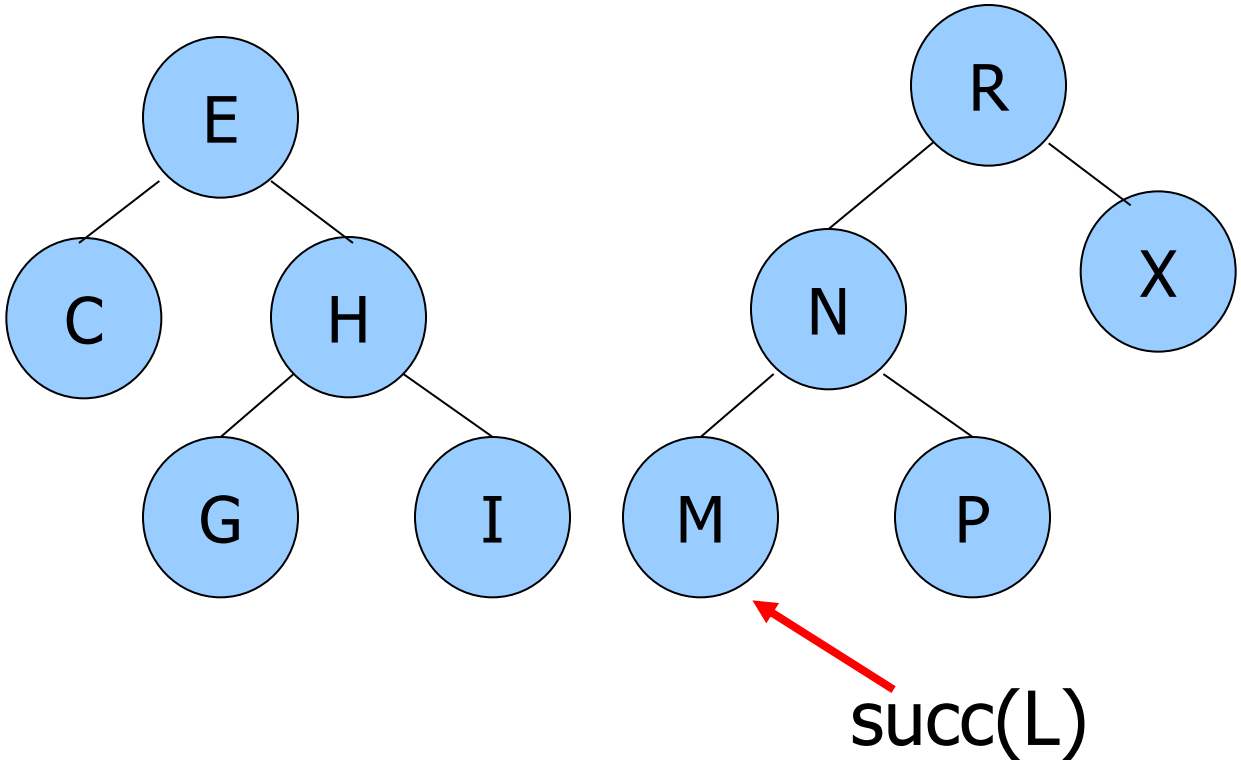
- la proprietà dei BST
- la struttura ad albero binario

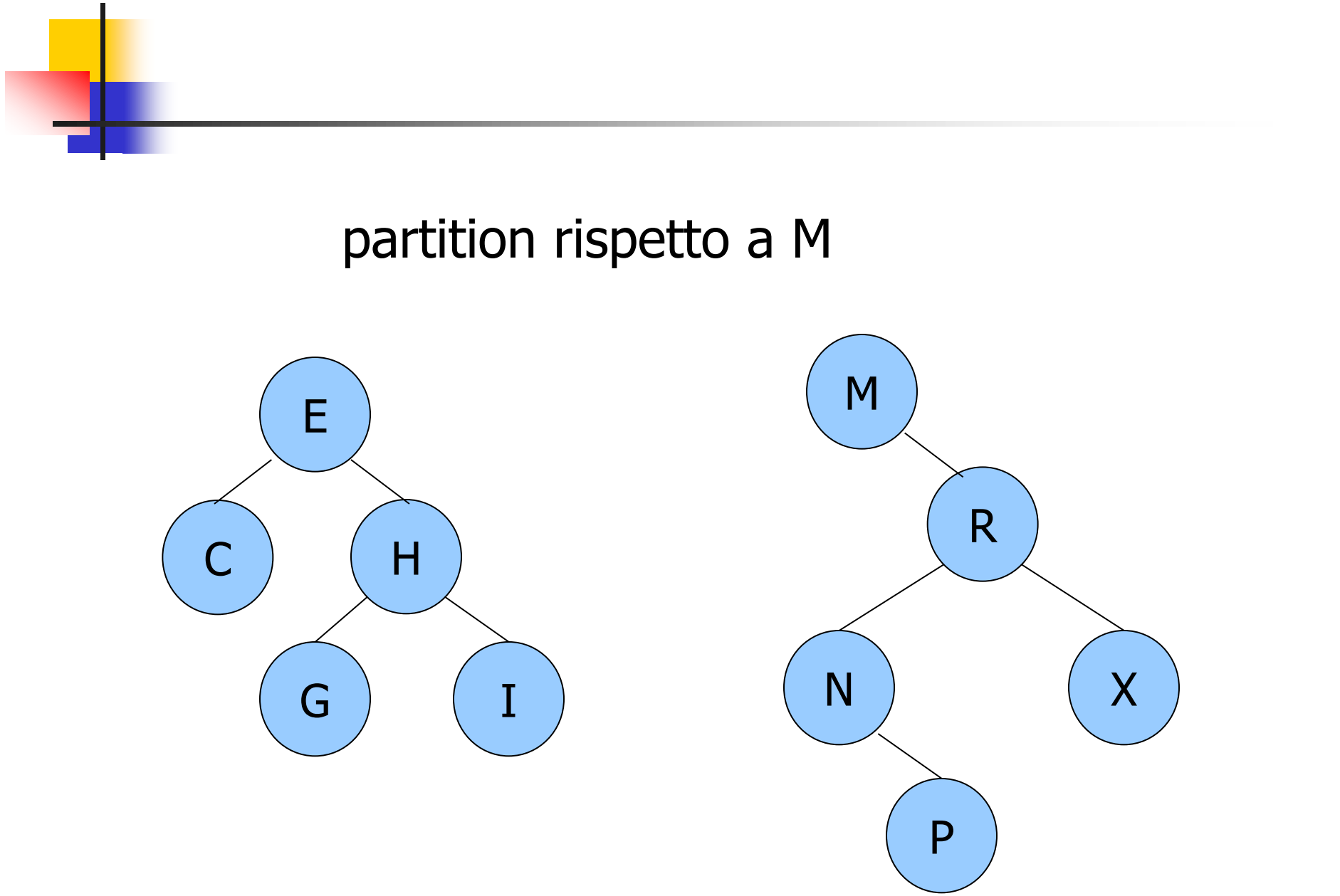
Passi:

- controllare se il nodo con l'item da cancellare è in uno dei sottoalberi. Se sì, cancellazione ricorsiva nel sottoalbero
- se è la radice, eliminarlo e ricombinare i 2 sottoalberi. La nuova radice è il succ o il pred dell'item cancellato.

Cancellazione di una radice

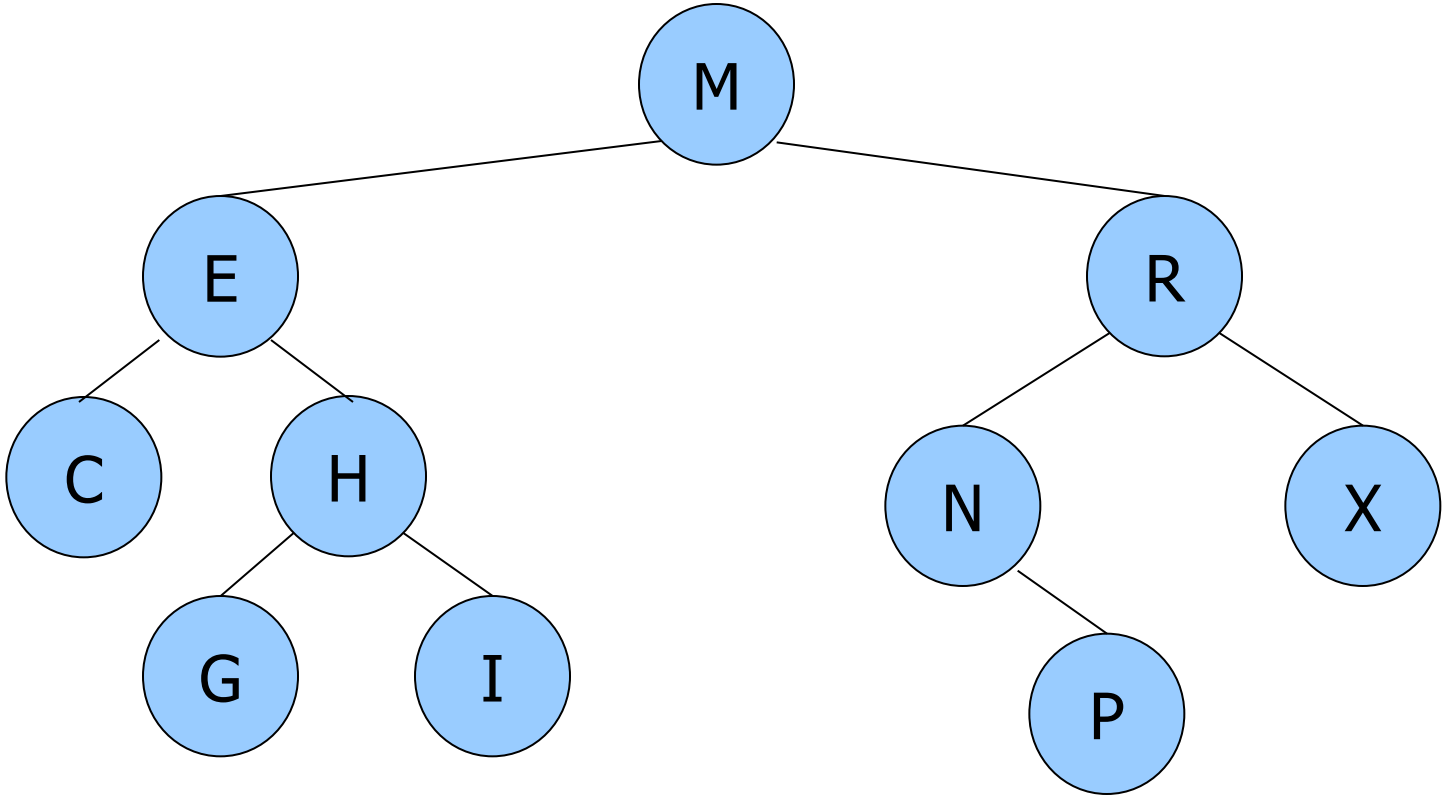



E
C
H
G
I
R
N
X
M
P
succ(L)



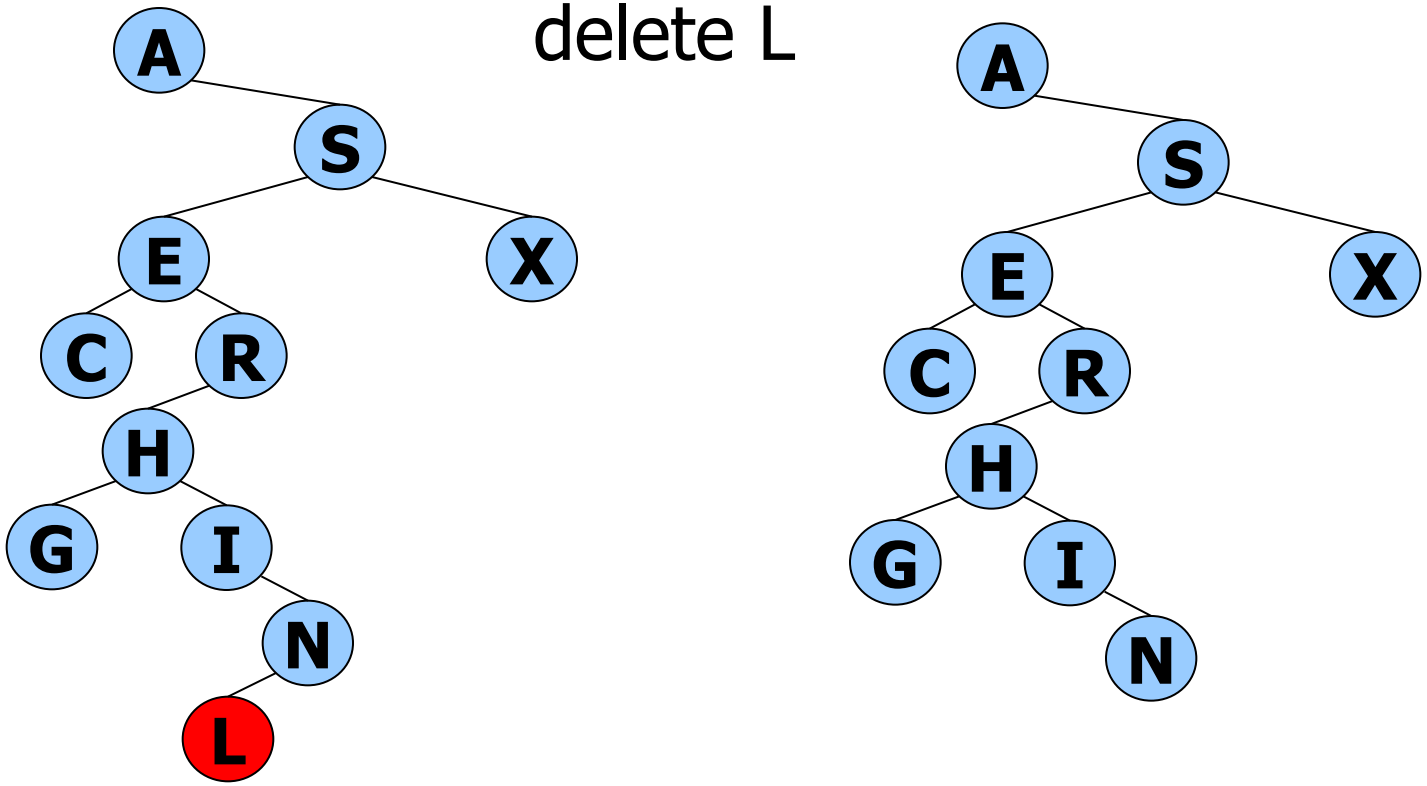
partition rispetto a M

ricostruzione dell'albero con radice M

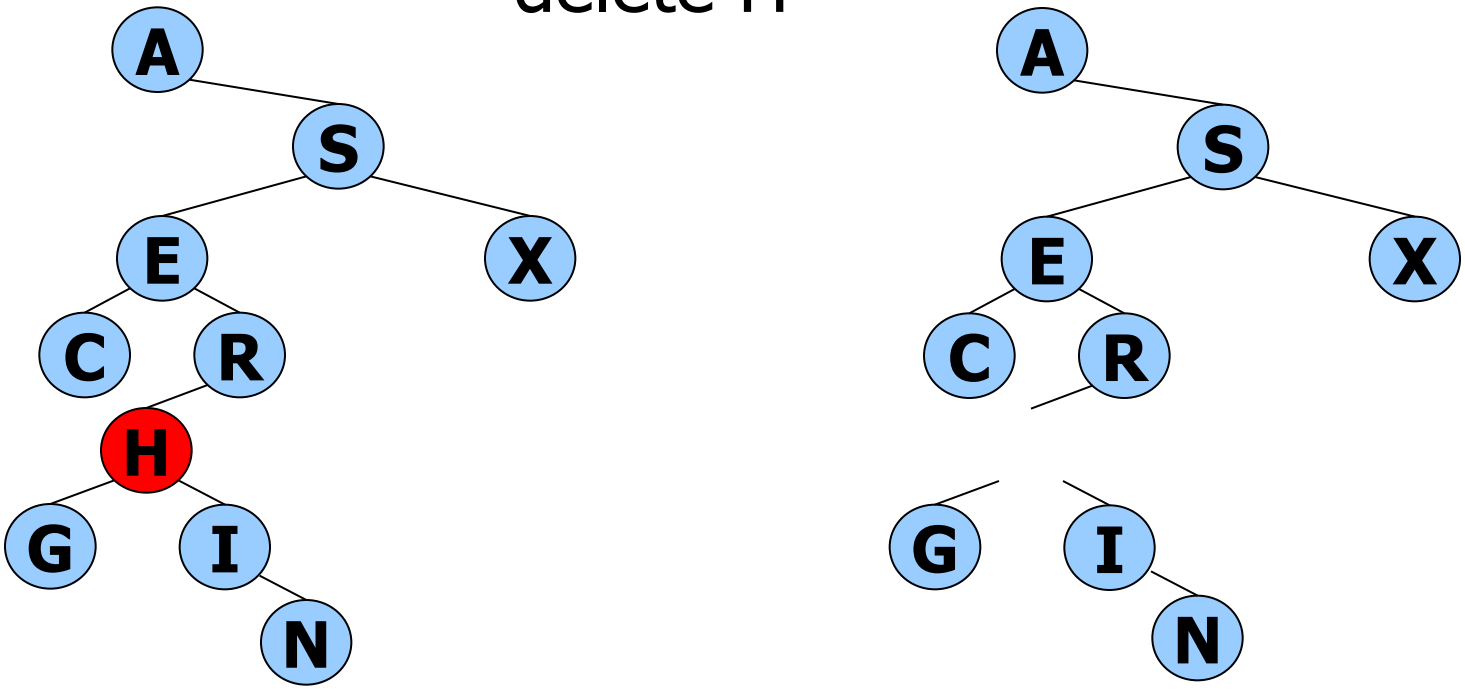


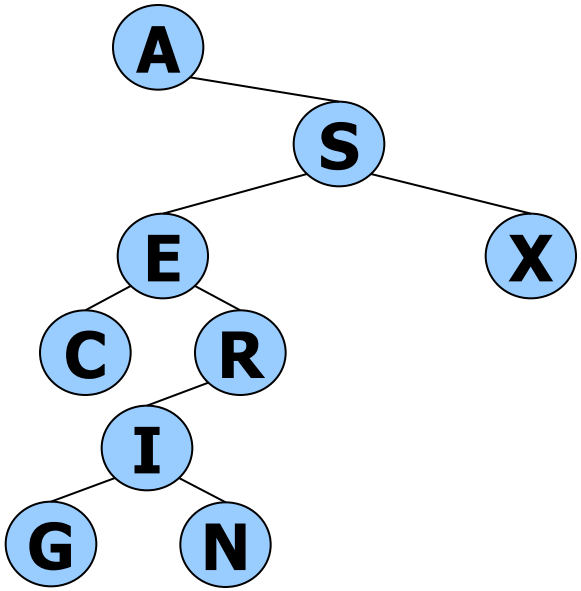
Esempio

cancellazione in sequenza di L, H, E

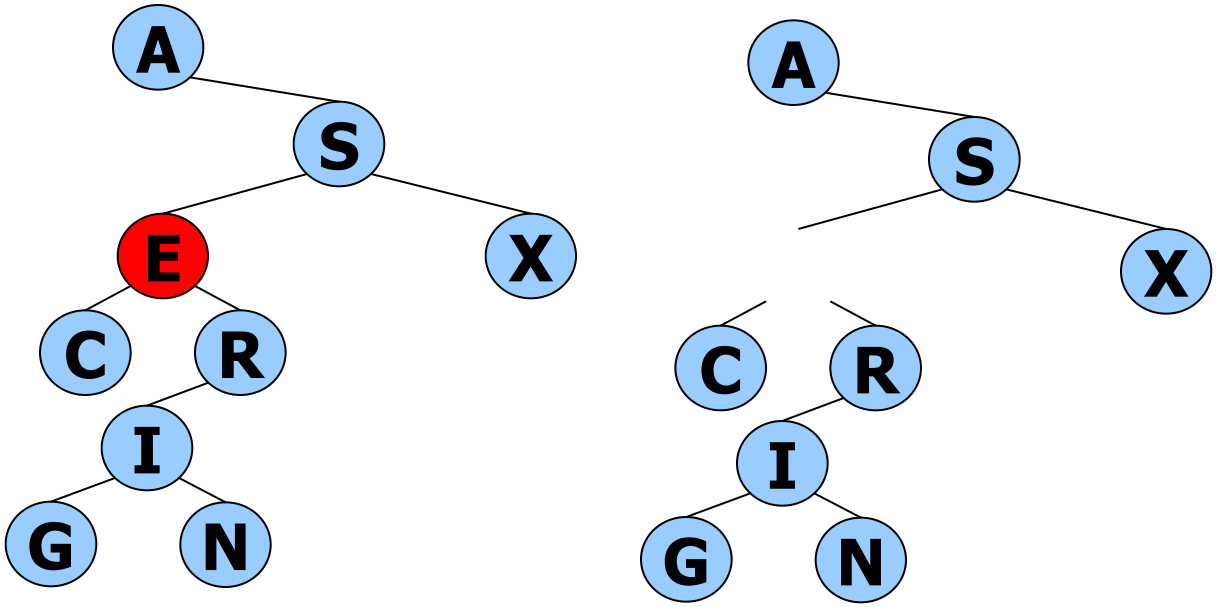


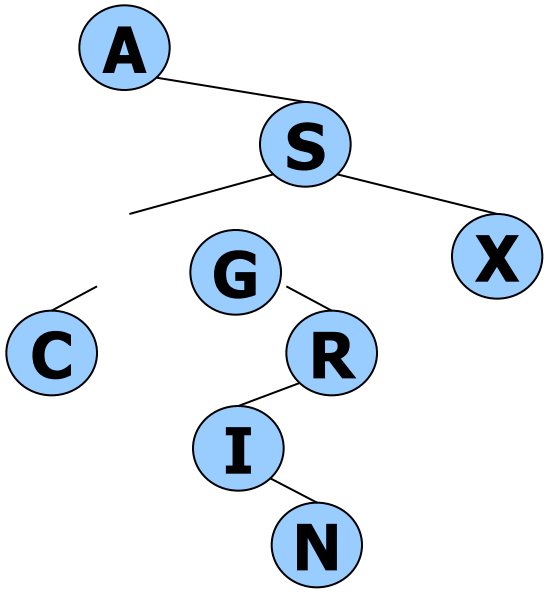
delete H

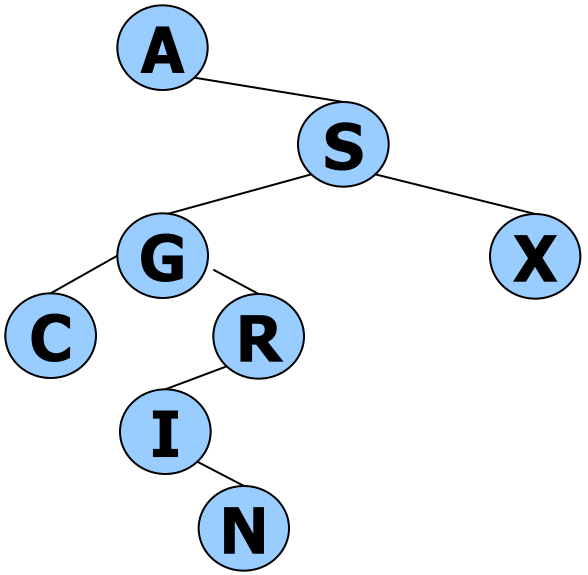


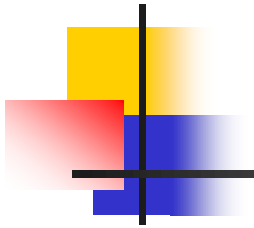

A
S
E
X
C
R
I
G
N

delete E




A
S
G
X
C
R
I
N


A
S
G
X
C
R
I
N



aggiornamento puntatore al padre

```
link joinLR(link a, link b, link z) {
  if (b == z)
    return a;
  b = partR(b, 0);
  b->l = a;
  a->p = b;
  b->N = a->N + b->r->N +1;
  return b;
}
```

aggiornamento dimensione sottoalberi

```
link deleteR(link h, Key k, link z) {
  link y, p;
  if (h == z) return z;
  if (KEYcompare(k, KEYget(h->item))==-1)
    h->l = deleteR(h->l, k, z);
  if (KEYcompare(k, KEYget(h->item))==1)
    h->r = deleteR(h->r, k, z);
  (h->N)--;
  if (KEYcompare(k, KEYget(h->item))==0) {
    y = h;   p = h->p;
    h = joinLR(h->l, h->r, z);  h->p = p;
    free(y);
  }
  return h;
}
void BSTdelete(BST bst, Key k) {
  bst->head = deleteR(bst->head, k, bst->z);
  bst->N--;
}
```

aggiornamento del numero di nodi

aggiornamento puntatore al padre

Estensioni delle strutture dati

Prima di sviluppare una nuova struttura dati è bene valutare se si possano «estendere» strutture esistenti con informazioni opportune.

Procedura:

1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

Esempio: Order-Statistic BST

Procedura:

1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

dimensione del sottoalbero

BST

O(1)

```
Item BSTselect(BST, int);
```

Interval BST

Intervallo chiuso: coppia ordinata di reali $[t_1, t_2]$, dove $t_1 \leq t_2$ e $[t_1, t_2] = \{t \in \Re: t_1 \leq t \leq t_2\}$.

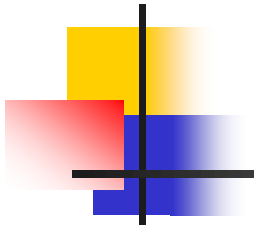
L'item intervallo $[t_1, t_2]$ può essere realizzato da una `struct` con campi `low` $= t_1$ e `high` $= t_2$.

Gli intervalli i e i' hanno intersezione se e solo se:

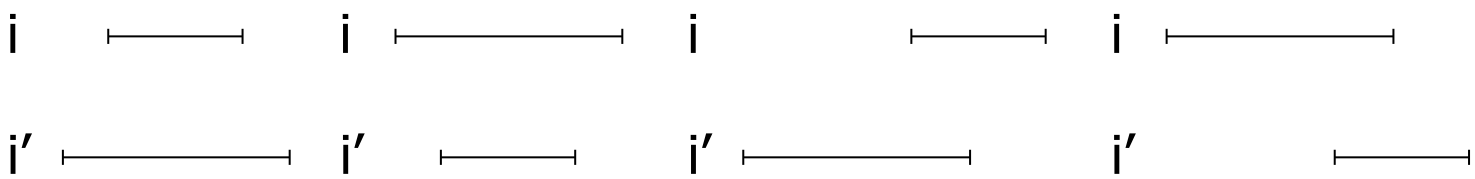
`low[i]` $\leq$ `high[i']` && `low[i']` $\leq$ `high[i]`.

$\forall$ i, i' vale la seguente tricotomia:

a. i e i' hanno intersezione
b. `high[i]` $\leq$ `low[i']`
c. `high[i']` $\leq$ `low[i]`

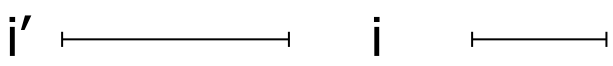


caso a

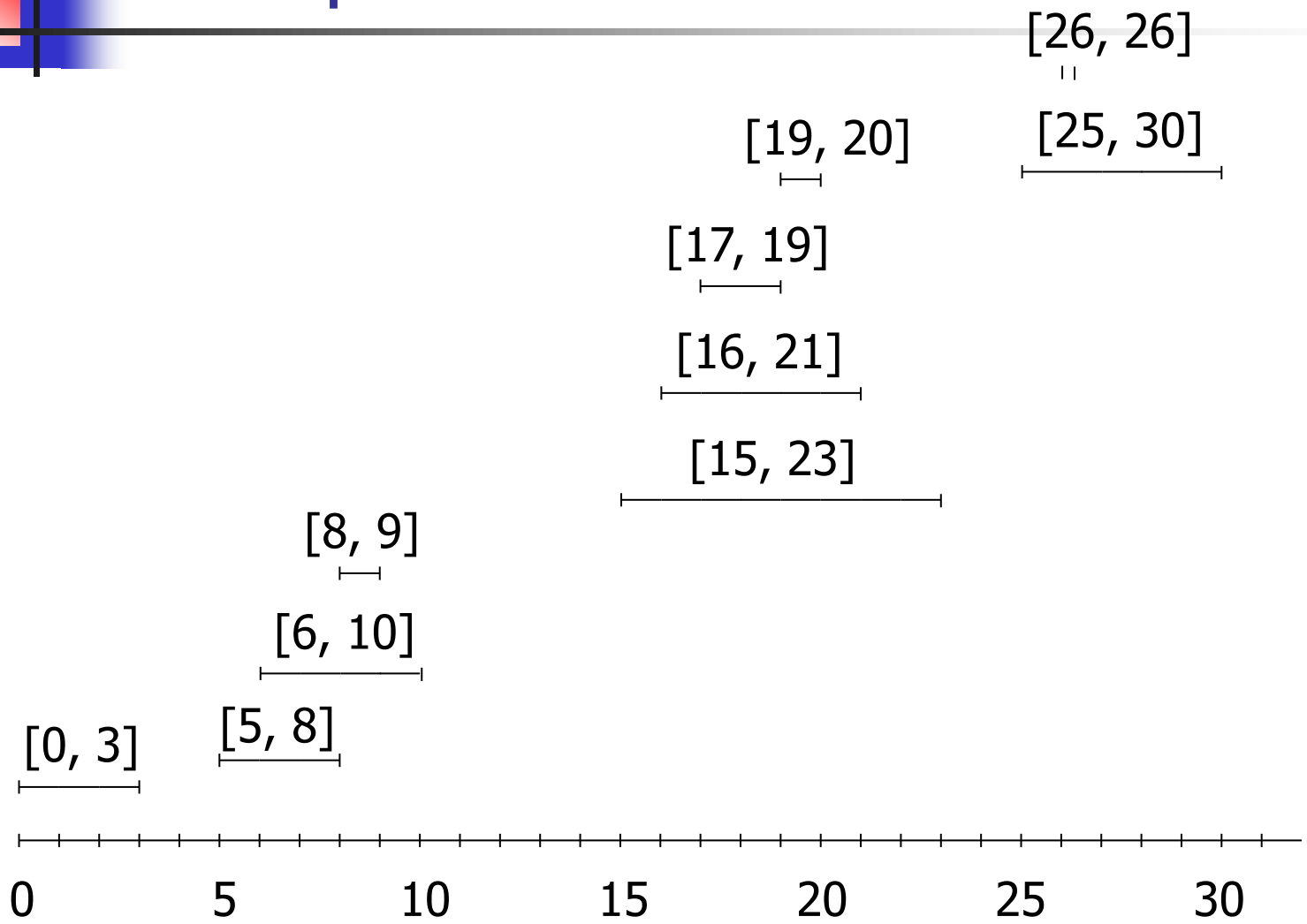


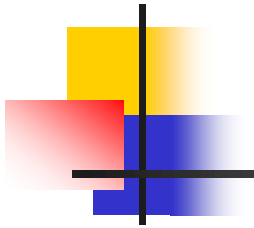
caso b

caso c



Esempio





BST con inserzione secondo l'estremo inferiore

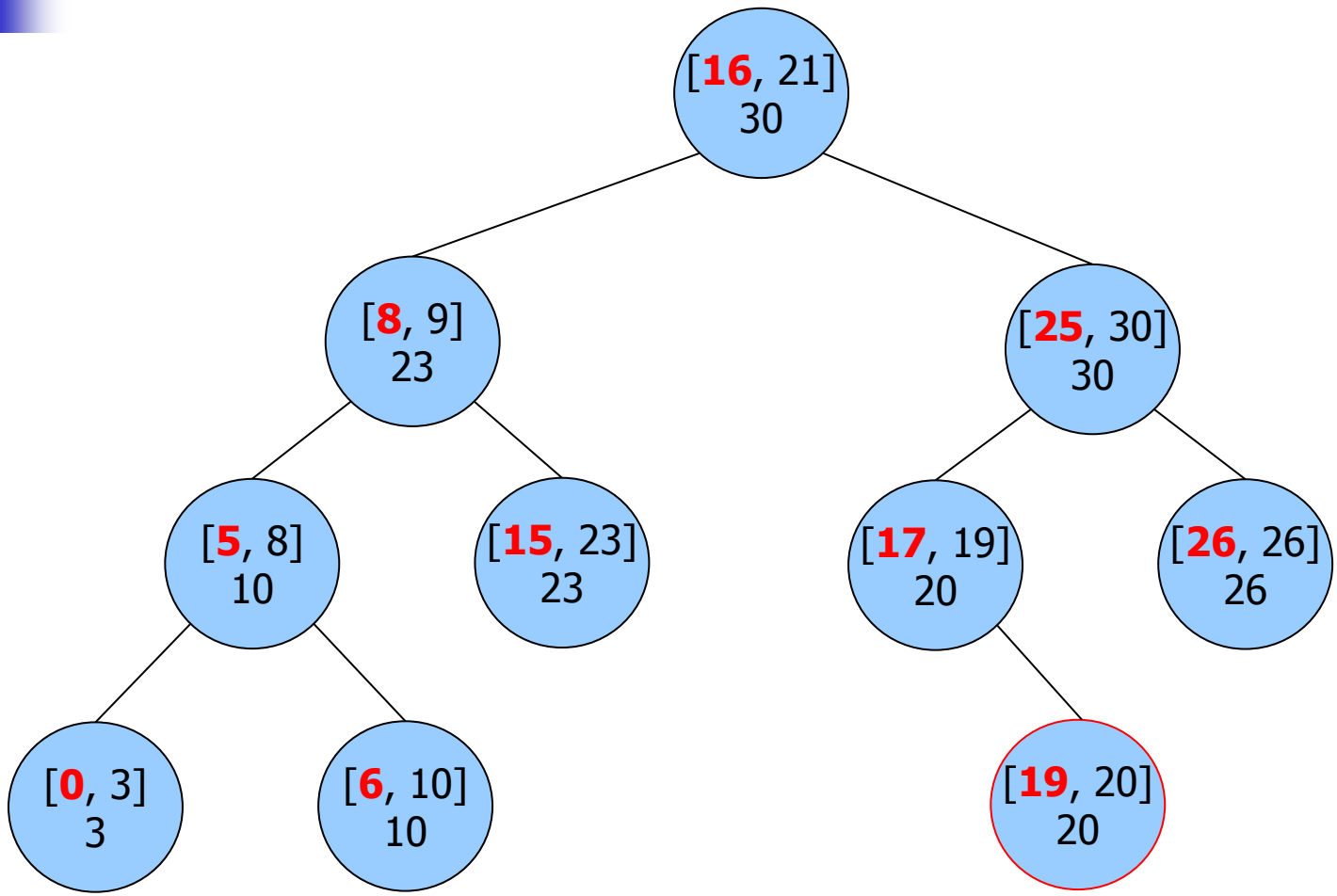
`max`: massimo `high` del sottoalbero

Proce

1. identificare la struttura dati candidata
2. identificare le informazioni supplementari
3. verificare di poter mantenere le informazioni supplementari senza alterare la complessità delle operazioni esistenti
4. sviluppare nuove operazioni.

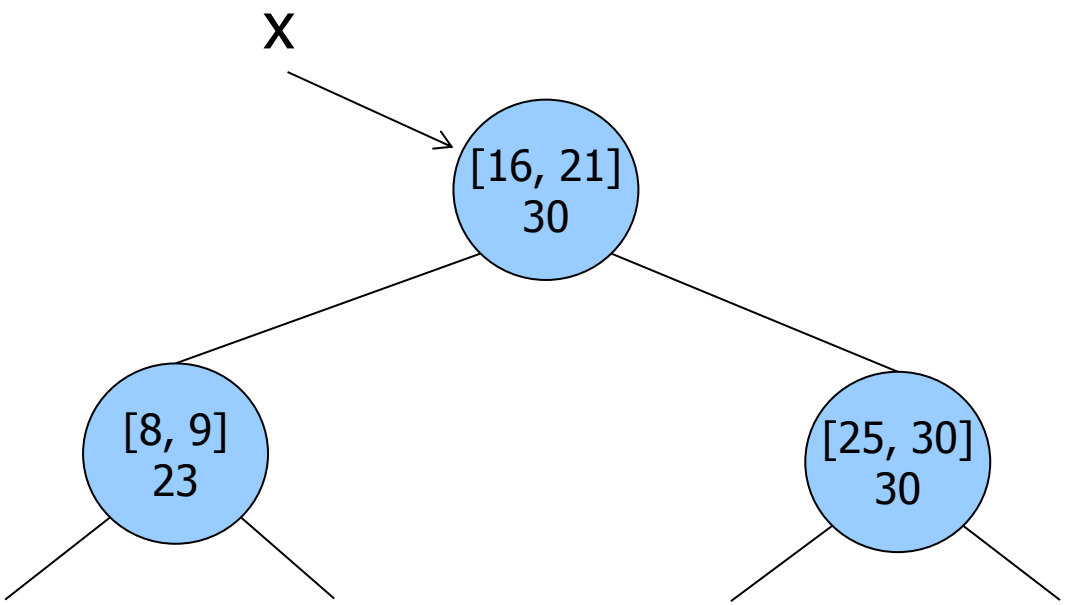
O(1)

```
Item IBSTsearch(IBST, Item);
```


[16, 21]
30
[8, 9]
23
[25, 30]
30
[5, 8]
10
[15, 23]
23
[17, 19]
20
[26, 26]
26
[0, 3]
3
[6, 10]
10
[19, 20]
20

Il calcolo di max è di complessità $\Theta(1)$:

x->max = max (high(x), x->left->max, x->right->max)



Operazioni su d (intervalli) di tipo `Item`:

```
Item ITEMscan();
void ITEMshow(Item data);
Item ITEMsetvoid();
int  ITEMhigh();
int  ITEMlow();
```

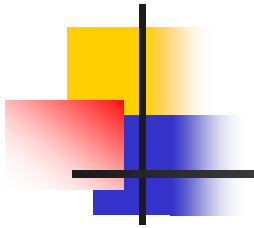
leggi intervallo da tastiera

visualizza intervallo

crea intervallo vuoto

estremo superiore

estremo inferiore



intersezione tra intervalli

controllo relazione =

```
int ITEMoverlap(Item data1,Item data2);
int ITEMeq(Item data1, Item data2);
int ITEMless(Item data1, Item data2);
int ITEMless_int(Item data1, int data2);
int ITEMcheckvoid(Item data);
```

controllo intervallo vuoto

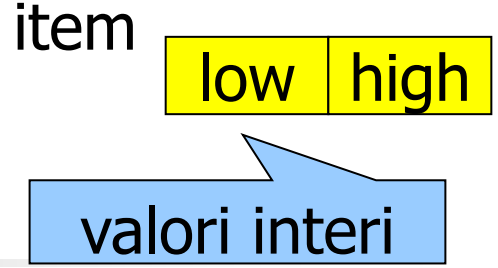
controllo relazione <

controllo in fase di ricerca

ADT di I categoria Item

Item.h

```
typedef struct item* Item;

Item ITEMscan();
void ITEMshow(Item data);
Item ITEMsetvoid();
int ITEMhigh();
int ITEMlow();
int ITEMoverlap(Item data1, Item data2);
int ITEMeq(Item data1, Item data2);
int ITEMless(Item data1, Item data2);
int ITEMless_int(Item data1, intm data2);
int ITEMcheckvoid(Item data);
```

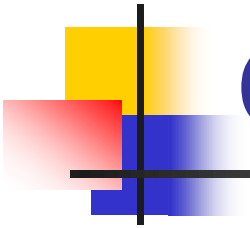


Item.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "Item.h“

struct item { int low; int high; };

Item ITEMscan() {
  int value;
  Item data = (Item) malloc(sizeof(struct item));
  if (data == NULL) return ITEMsetvoid();
  else {
    scanf("%d", &value); data->low = value;
    scanf("%d", &value); data->high = value;
  }
  return data;
}
```

```
void ITEMshow(Item data) {
  printf("[%d, %d] ", data->low, data->high);
}

Item ITEMsetvoid() {
  Item tmp = (Item) malloc(sizeof(struct item));
  if (tmp != NULL) { tmp->low = -1; tmp->high = -1; }
  return tmp;
}

int ITEMhigh(Item data) { return data->high; }

int ITEMlow(Item data) { return data->low; }

int ITEMoverlap(Item data1, Item data2) {
  if ((data1->low<=data2->high)&&(data2->low<=data1->high))
    return 1;
  else return 0;
}
```

```
int ITEMeq(Item data1, Item data2) {
  if ((data1->low==data2->low)&&(data1->high==data2->high))
    return 1;
  else return 0;
}

int ITEMless(Item data1, Item data2) {
  if ((data1->low < data2->low))
    return 1;
  else
    return 0;
}

int ITEMcheckvoid(Item data) {
  if ((data->low == -1) && (data->high == -1))
    return 1;
  else
    return 0;
}
```



Operazioni

```
Item IBSTsearch(IBST, Item) ;
```

cerca un item (intervallo) nell'Interval BST e ritorna il primo intervallo che lo interseca

```
void IBSTinsert(IBST, Item) ;
```

inserisci un item (intervallo) nell'Interval BST

```
void   IBSTdelete(IBST, Item) ;
```

cancella un item (intervallo) dall'Interval BST

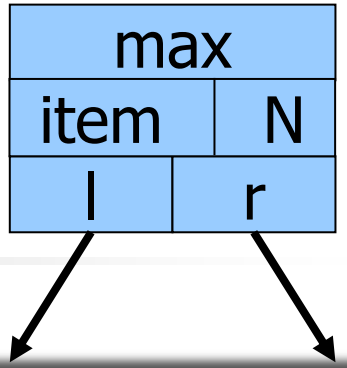
ADT di I categoria Interval BST

IBST.h

```
typedef struct  intervalbinarysearchtree *IBST;

void   IBSTinit(IBST) ;
void   IBSTinsert(IBST, Item) ;
void   IBSTdelete(IBST, Item) ;
Item   IBSTsearch(IBST, Item) ;
int    IBSTcount(IBST);
int    IBSTempty(IBST);
void   IBSTsortinorder(IBST) ;
void   IBSTsortpreorder(IBST) ;
void   IBSTsortpostorder(IBST) ;
```

nuove funzioni

funzioni modificate



IBST.c

```
#include <stdlib.h>
#include <stdio.h>
#include "Item.h"
#include "IBST.h“

typedef struct IBSTnode *link;

struct IBSTnode {Item item; link l, r; int N; int max;} ;

struct intervalbinarysearchtree {link head;int N;link z;};

link NEW(Item item, link l, link r, int N, int max) {
  link x = malloc(sizeof *x);
  x->item = item;
  x->l = l; x->r = r;  x->N = N;
  x->max = max;
  return x;
}
```

massimo high del sottoalbero

```
IBST IBSTinit( ) {
  IBST ibst = malloc(sizeof *ibst) ;
  ibst->N = 0;
  ibst->head=(ibst->z=NEW(NULLitem,NULL,NULL,0,-1));
  return ibst;
}

int max (int a, int b, int c) {
  int m = a;
  if (b > m) m = b;
  if (c > m) m = c;
  return m;
}

int IBSTcount(IBST ibst) { return ibst->N; }

int IBSTempty(IBST ibst) {
  if (IBSTcount(ibst) == 0) return 1;
  else return 0;
}
```

Insert

```
link insertR(link h, Item item, link z) {
  if (h == z)
    return NEW(item, z, z, 1, ITEMhigh(item));
  if (ITEMless(item, h->item)) {
    h->l = insertR(h->l, item, z);
    h->max = max(h->max, h->l->max, h->r->max);
  }
  else {
    h->r = insertR(h->r, item, z);
    h->max = max(h->max, h->l->max, h->r->max);
  }
  (h->N)++;
  return h;
}

void IBSTinsert(IBST ibst, Item item) {
  ibst->head = insertR(ibst->head, item, ibst->z);
  ibst->N++;
}
```

rotL/rotR

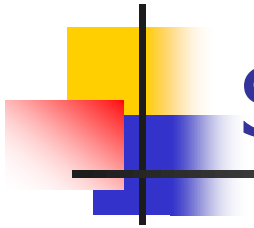
```
link rotL(link h) {
  link x = h->r;
  h->r = x->l;
  x->l = h;
  x->N = h->N;
  h->N = h->l->N + h->r->N +1;
  h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
  x->max = max(ITEMhigh(x->item), x->l->max, x->r->max);
  return x;
}
link rotR(link h) {
  link x = h->l;
  h->l = x->r;
  x->r = h;
  x->N = h->N;
  h->N = h->r->N + h->l->N +1;
  h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
  x->max = max(ITEMhigh(x->item), x->l->max, x->r->max);
  return x;
```

partR/joinLR

```
link partR(link h, int k) {
  int t  = h->l->N;
  if (t > k) {
    h->l = partR(h->l, k);
    h = rotR(h);
  }
  if (t < k) {
    h->r = partR(h->r, k-t-1);
    h = rotL(h);
  }
  return h;
}
link joinLR(link a, link b, link z) {
  if (b == z)  return a;
  b = partR(b, 0);
  b->l = a;
  b->N = a->N + b->r->N +1;
  b->max = max(ITEMhigh(b->item), a->max, b->r->max);
  return b;
}
```

Delete

```
link deleteR(link h, Item item, link z) {
  link x;
  if (h == z) return z;
  if (ITEMless(item, h->item)) {
    h->l = deleteR(h->l, item, z);
    h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
  }
  if (ITEMless(h->item, item)) {
    h->r = deleteR(h->r, item, z);
    h->max = max(ITEMhigh(h->item), h->l->max, h->r->max);
  }
  (h->N)--;
  if (ITEMeq(item, h->item)) {
    x = h; h = joinLR(h->l, h->r, z); free(x);
  }
  return h;
}
void IBSTdelete(IBST ibst, Item item) {
  ibst->head=deleteR(ibst->head,item,ibst->z); ibst->N--;
}
```



Search

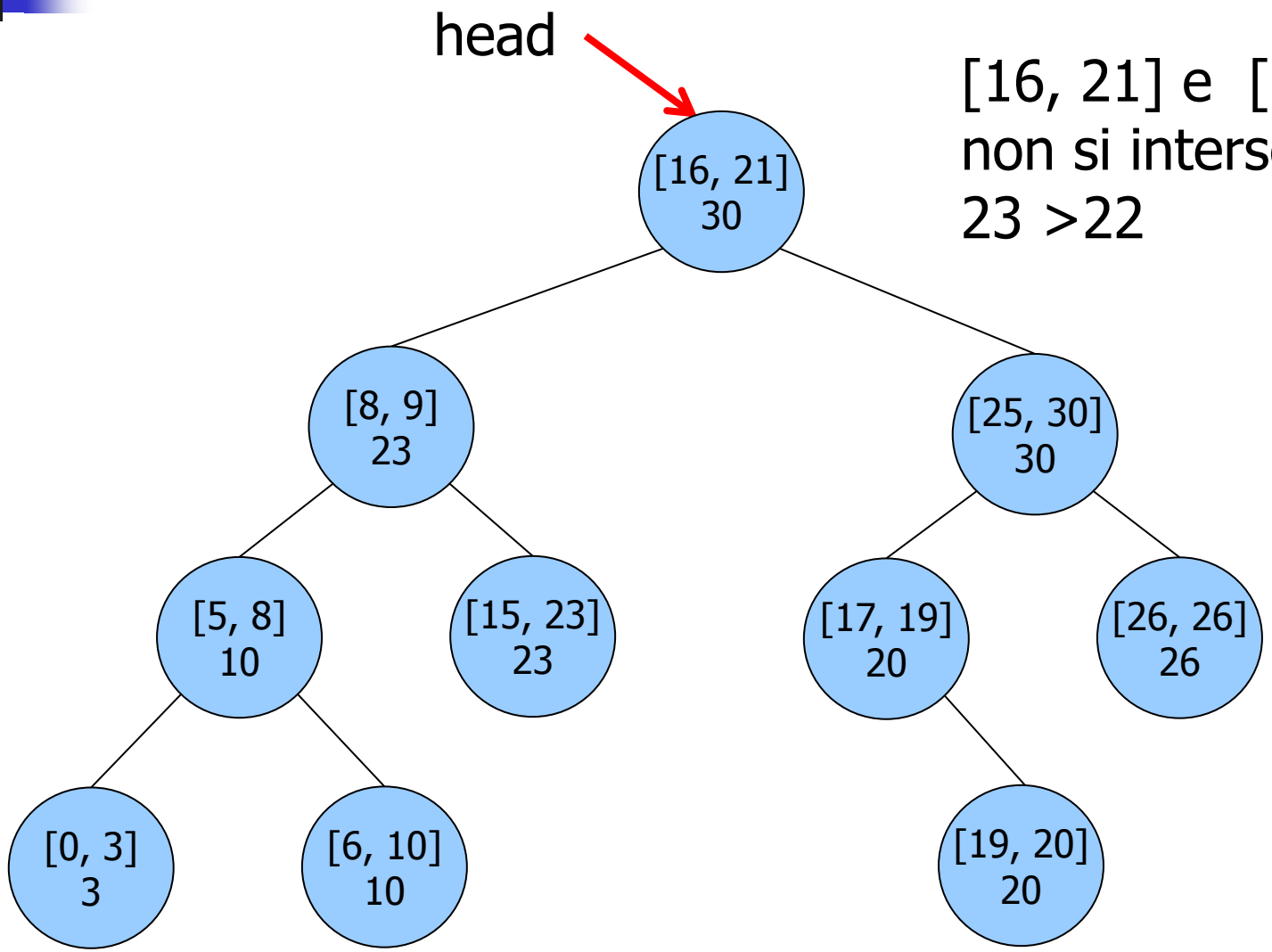
Ricerca di un nodo h con intervallo che interseca l'intervallo i:

- percorrimento dell'albero dalla radice
- terminazione: trovato intervallo che interseca i oppure si è giunti ad un albero vuoto
- ricorsione: dal nodo h
  - su sottoalbero sinistro se h->l->max ≥ low[i]
  - su sottoalbero destro se h->l->max < low[i]

```
Item searchR(link h, Item item, link z) {
  if (h == z)
    return ITEMsetvoid();
  if (ITEMoverlap(item, h->item))
    return h->item;
  if (ITEMless_int(item, h->l->max))
    return searchR(h->l, item, z);
  else
    return searchR(h->r, item, z);
  }

Item IBSTsearch(IBST ibst, Item item) {
  return searchR(ibst->head, item, ibst->z);
```

Esempio

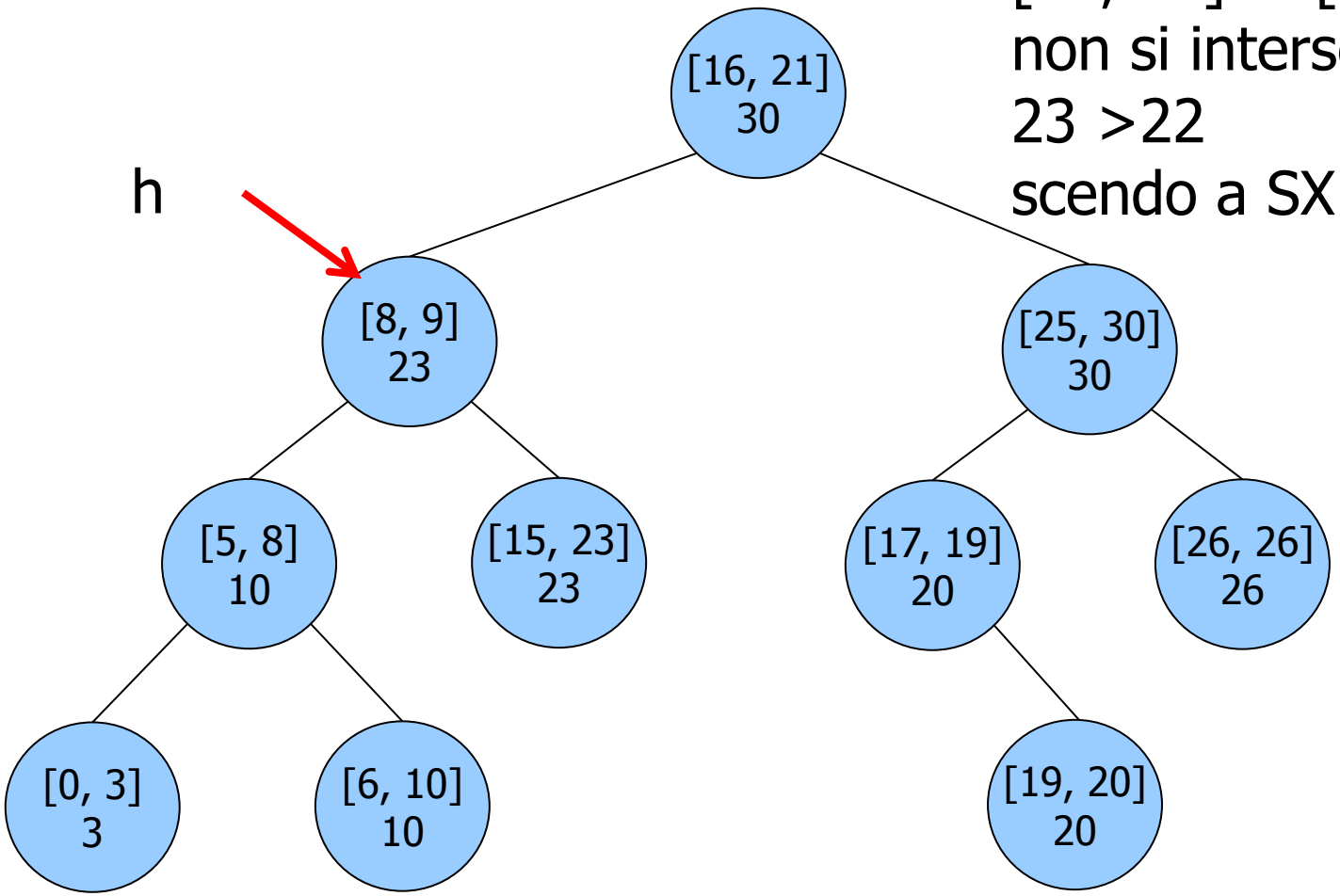
Ricerca di un intervallo che interseca [22, 25]



[16, 21] e [22, 25] non si intersecano
23 >22

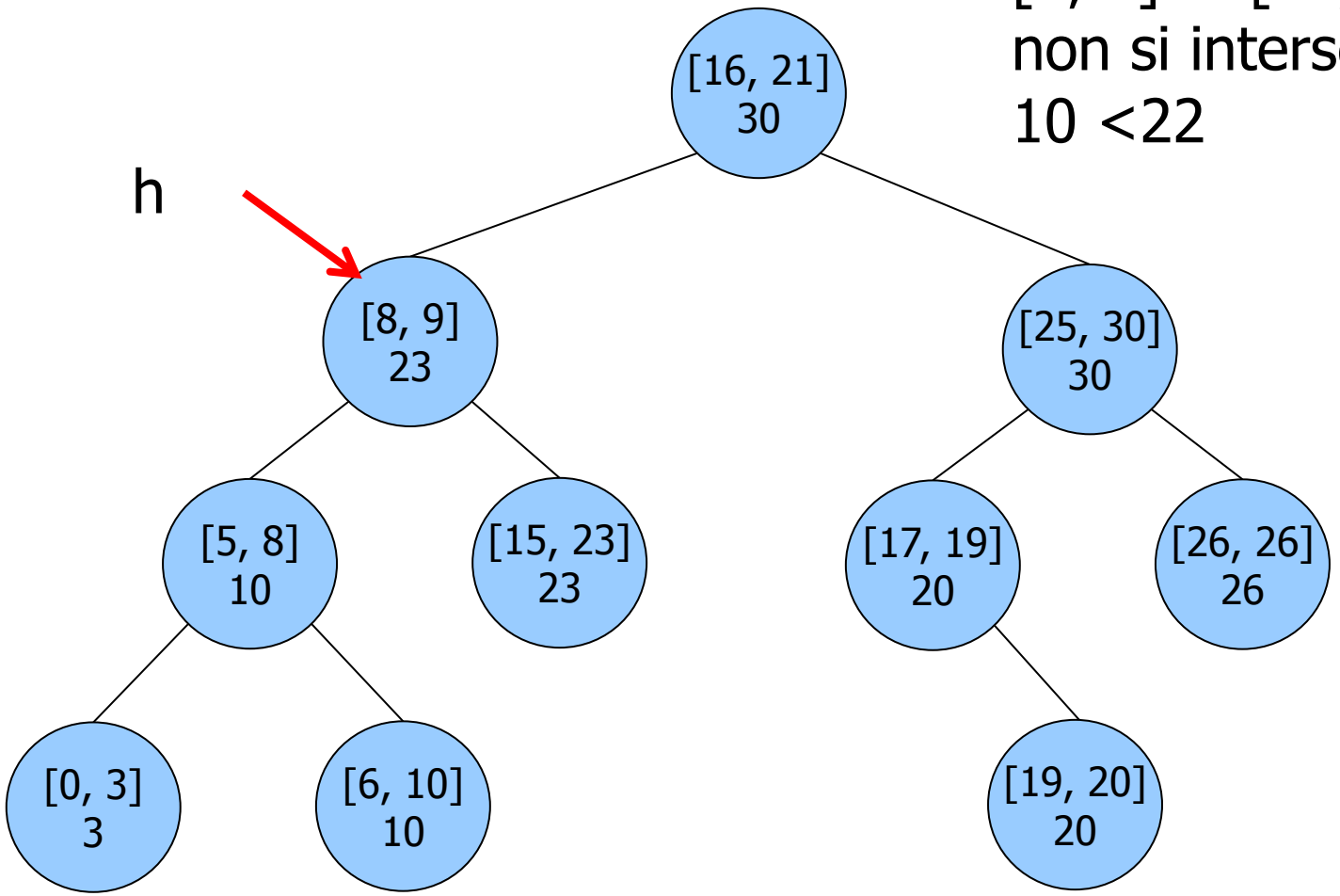
Ricerca di un intervallo che interseca [22, 25]

[16, 21] e [22, 25]
non si intersecano
23 >22
scendo a SX



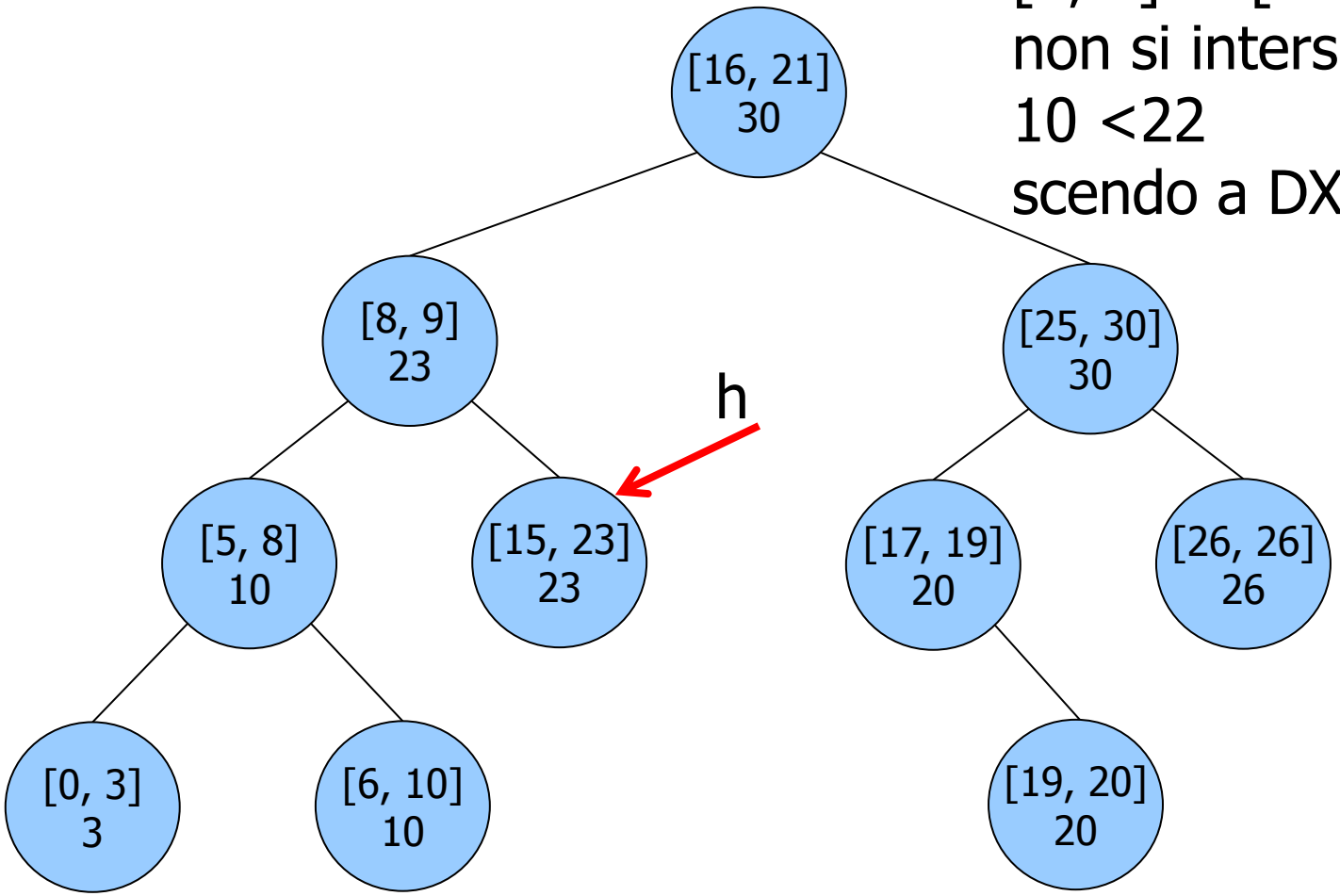
Ricerca di un intervallo che interseca [22, 25]

[8, 9] e [22, 25] non si intersecano
10 <22



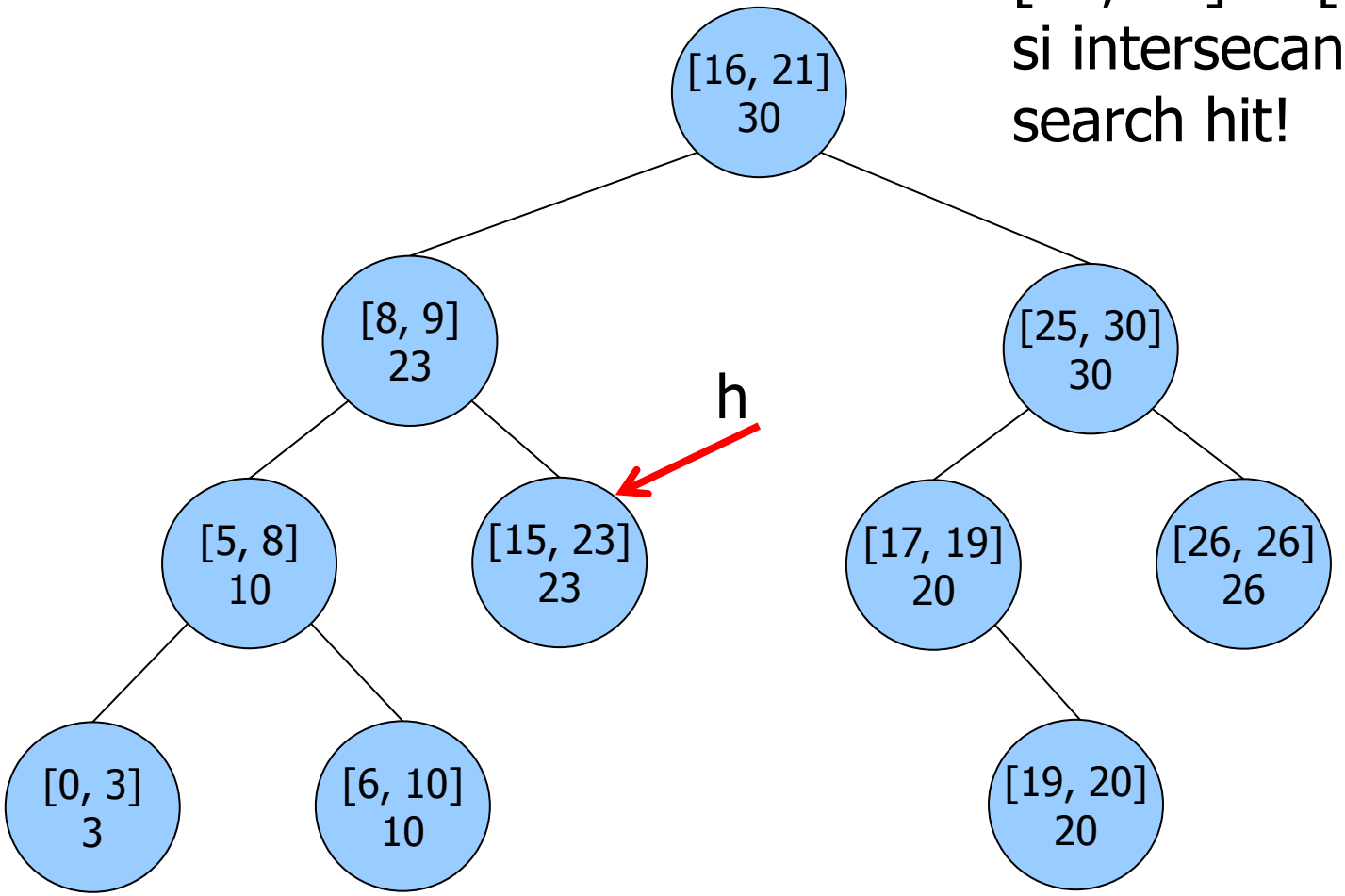
Ricerca di un intervallo che interseca [22, 25]

[8, 9] e [22, 25]
non si intersecano
10 <22
scendo a DX



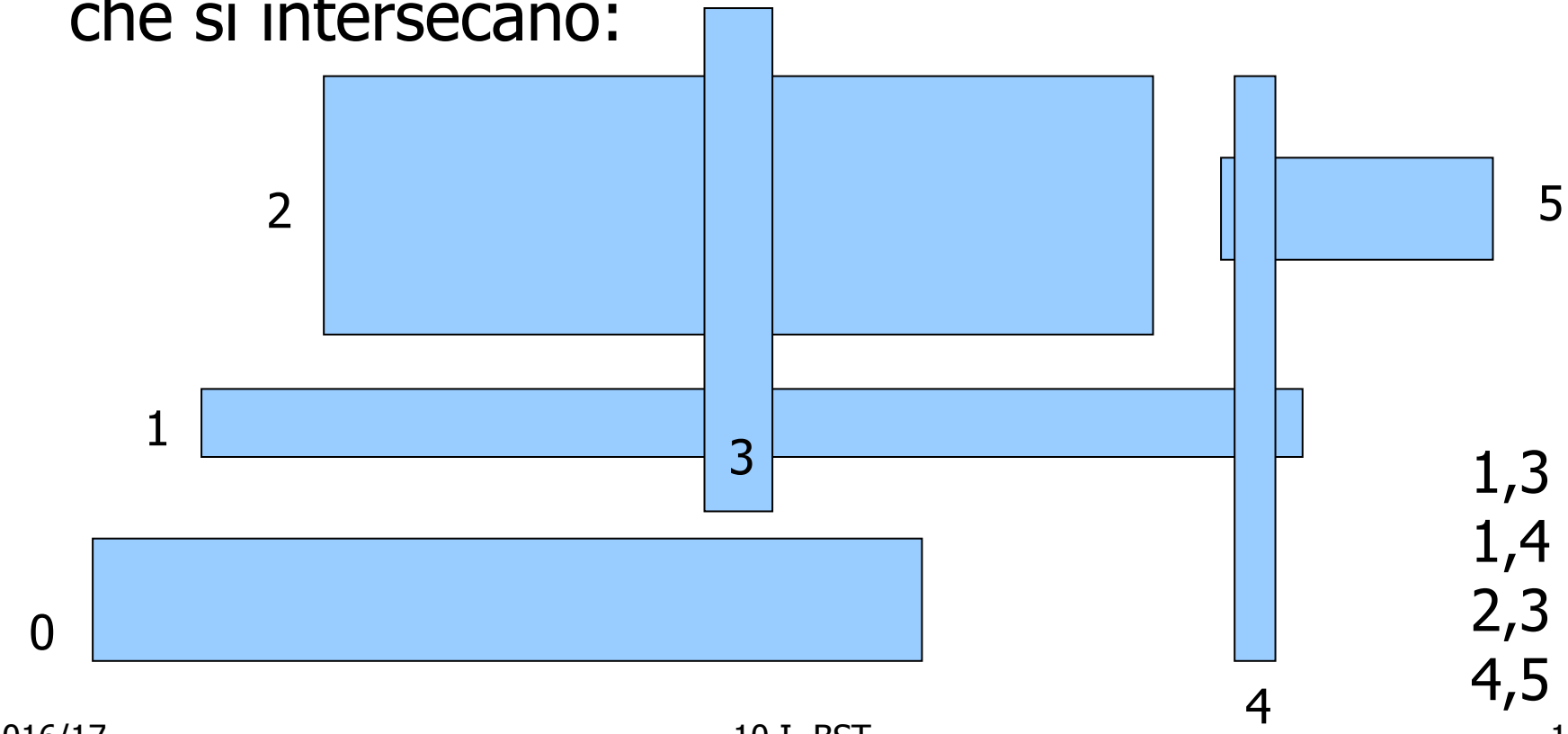
Ricerca di un intervallo che interseca [22, 25]

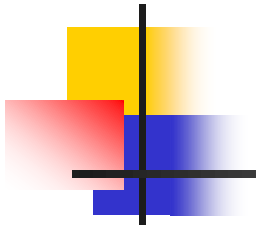
[15, 23] e [22, 25] si intersecano search hit!



Applicazioni degli I-BST

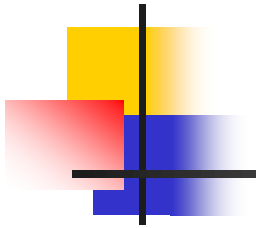
Dati N rettangoli disposti parallelamente agli assi ortogonali, determinare tutte le coppie che si intersecano:





Applicazione al CAD elettronico: verificare se le piste si intersecano in un circuito elettronico.

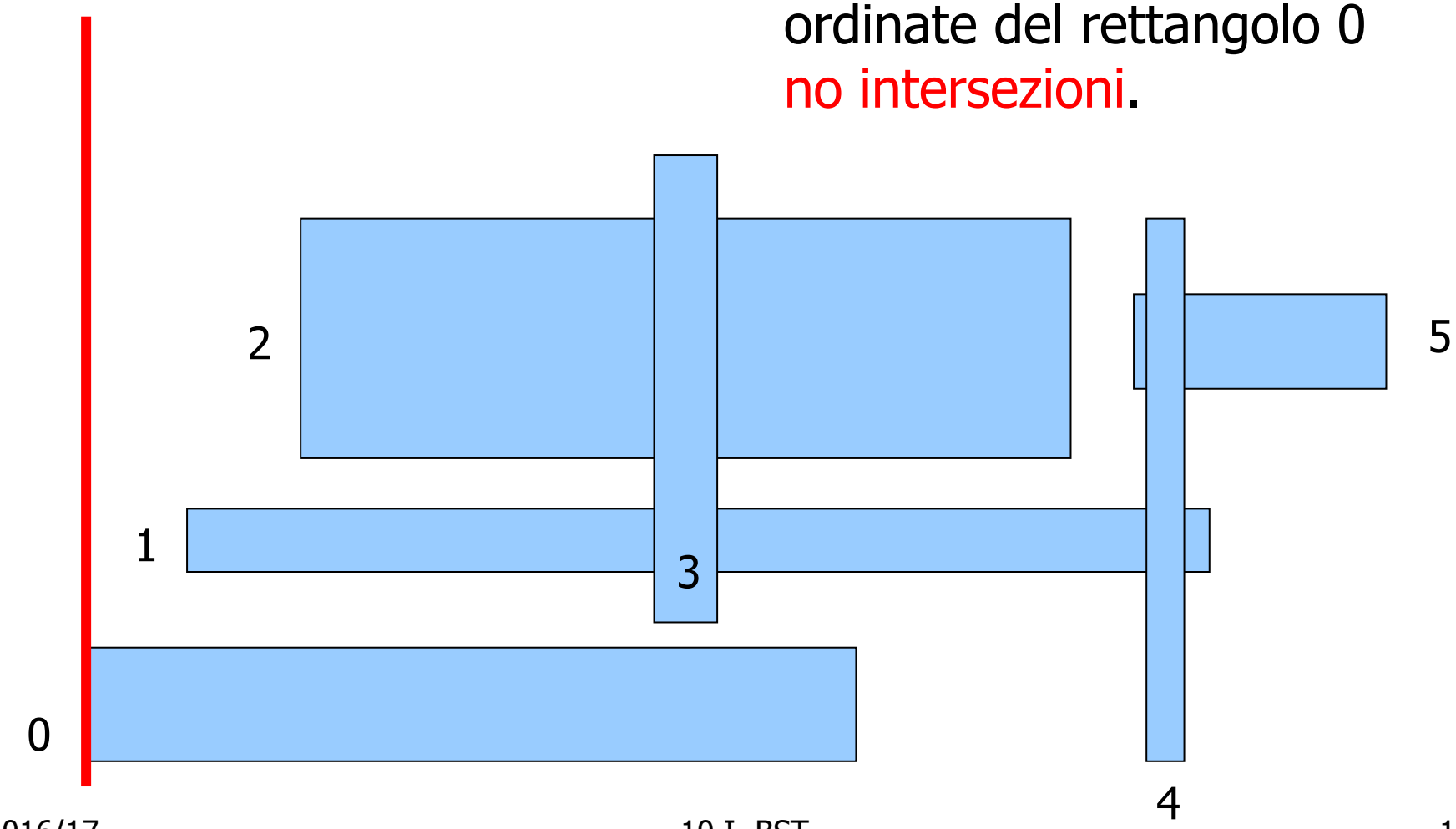
Algoritmo banale: controllare l'intersezione tra tutte le coppie di rettangoli, complessità $O(N^2)$.



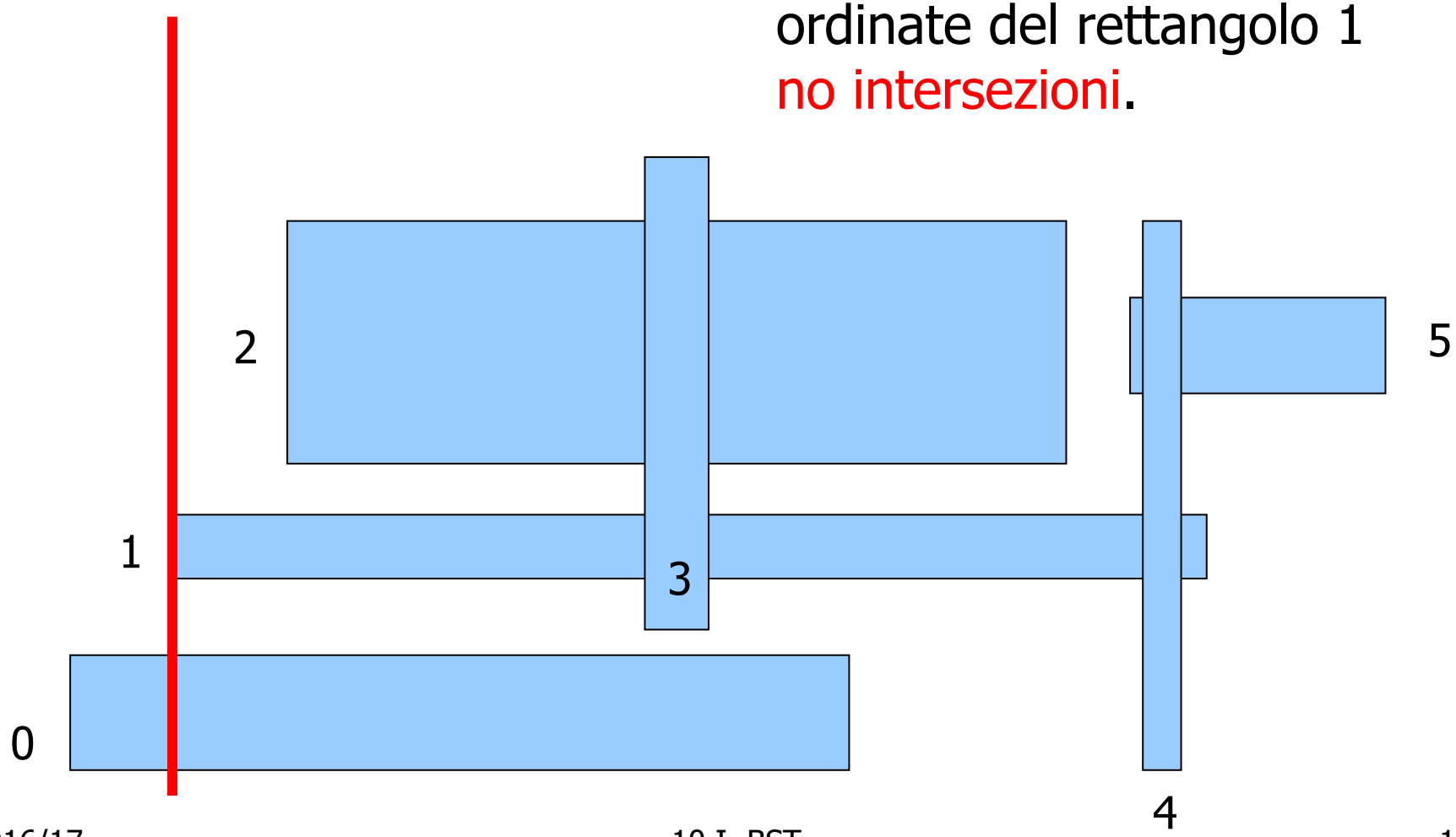
Algoritmo efficiente: complessità O(NlogN), applicabilità a VLSI e oltre:

- ordina i rettangoli per ascisse dell'estremo sinistro crescenti
- itera sui rettangoli per ascisse crescenti:
  - quando incontri l'estremo sinistro, inserisci in un I-BST l'intervallo delle ordinate e controlla l'intersezione
  - quando incontri l'estremo destro, rimuovi l'intervallo delle ordinate dall'I-BST.

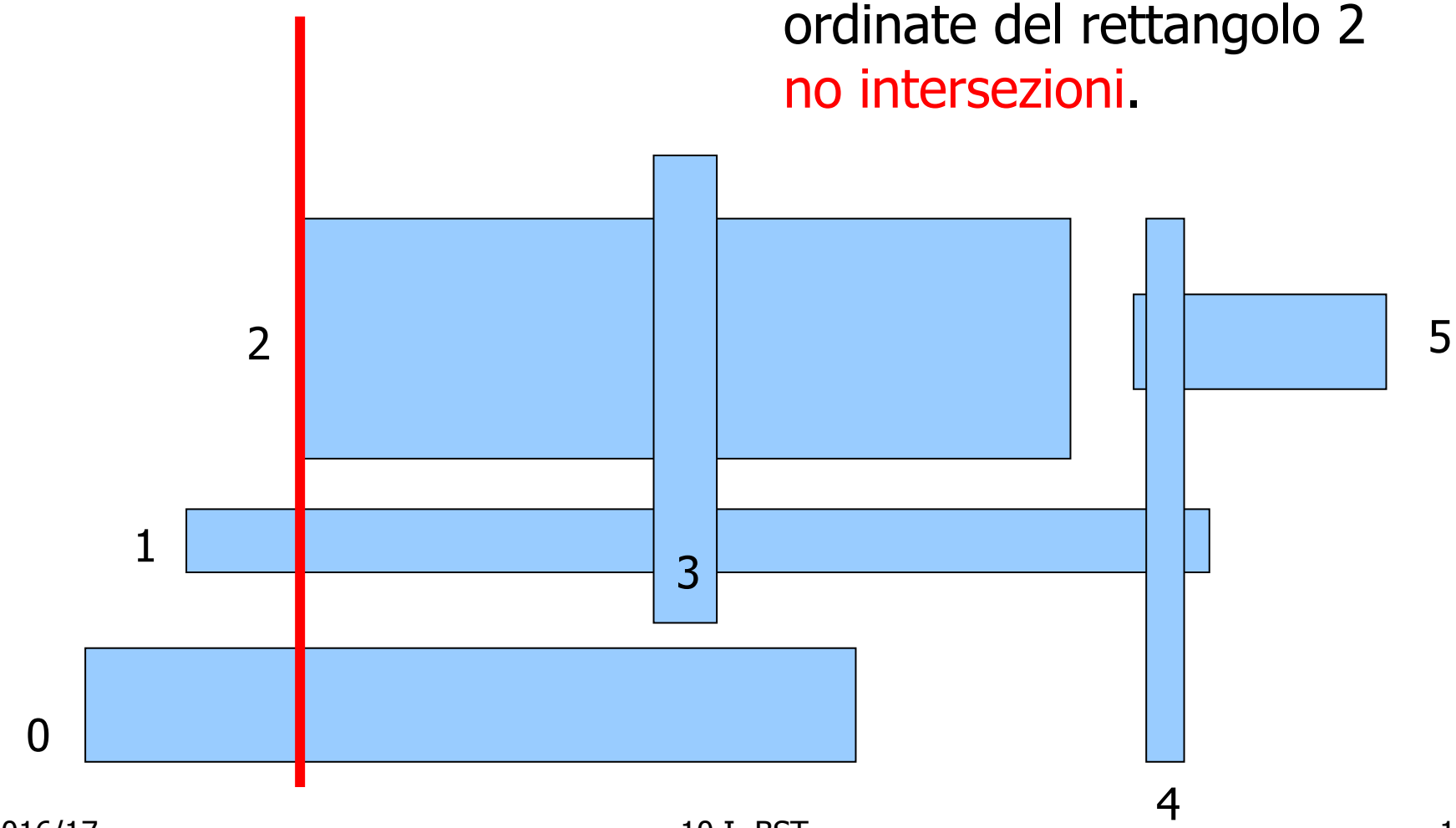
inserisci intervallo ordinate del rettangolo 0
no intersezioni.



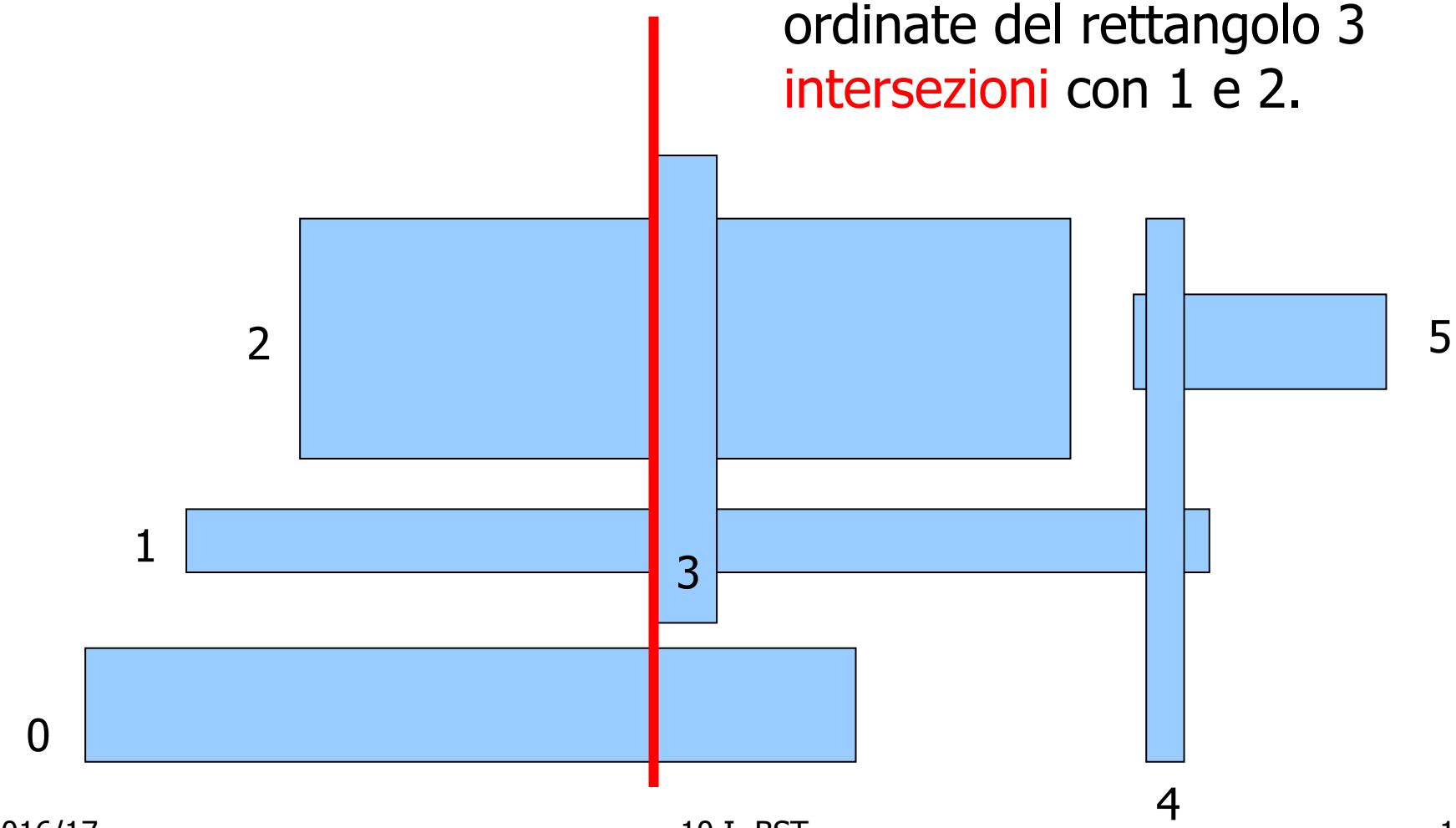
inserisci intervallo ordinate del rettangolo 1
no intersezioni.



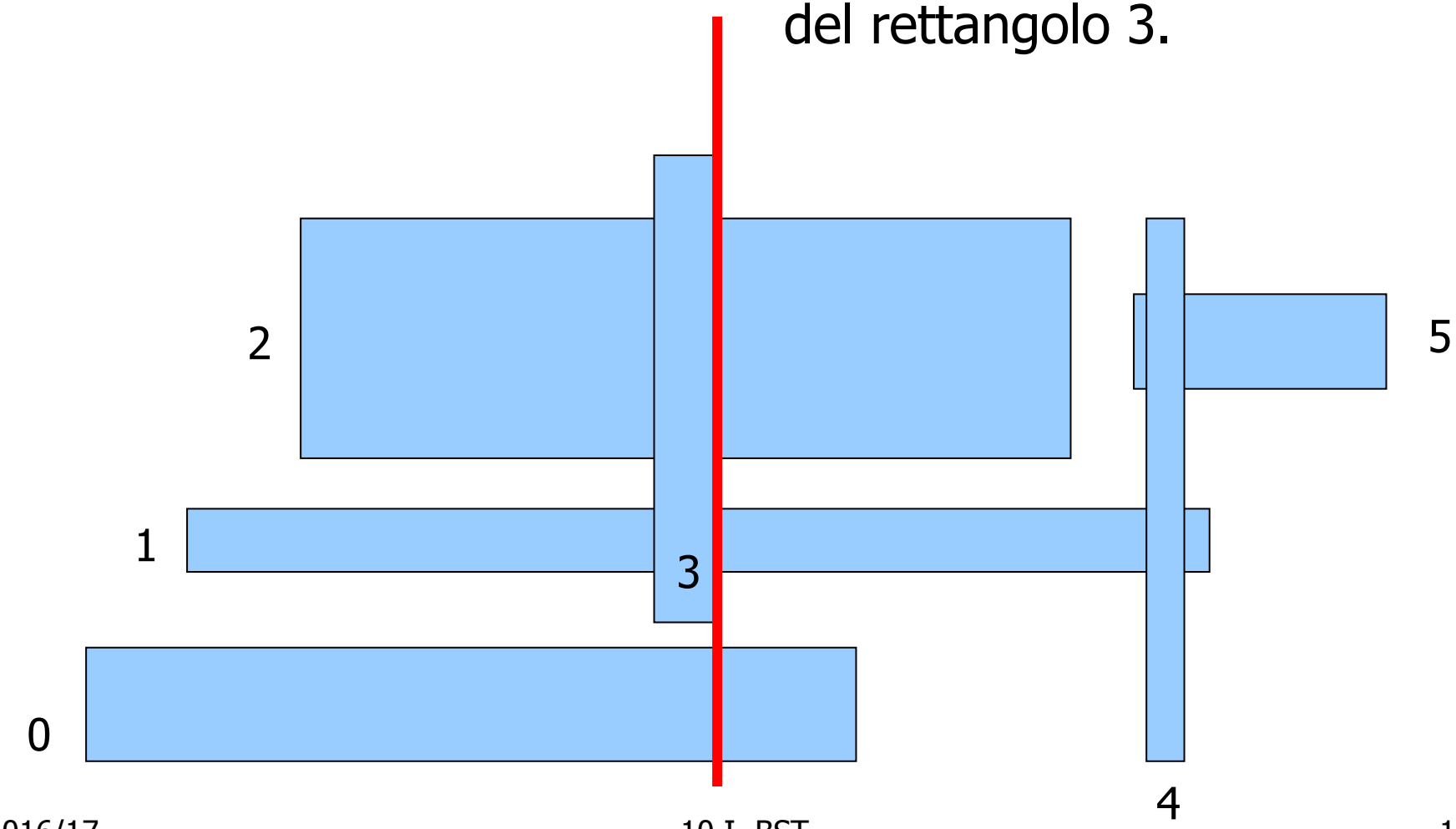
inserisci intervallo ordinate del rettangolo 2
no intersezioni.



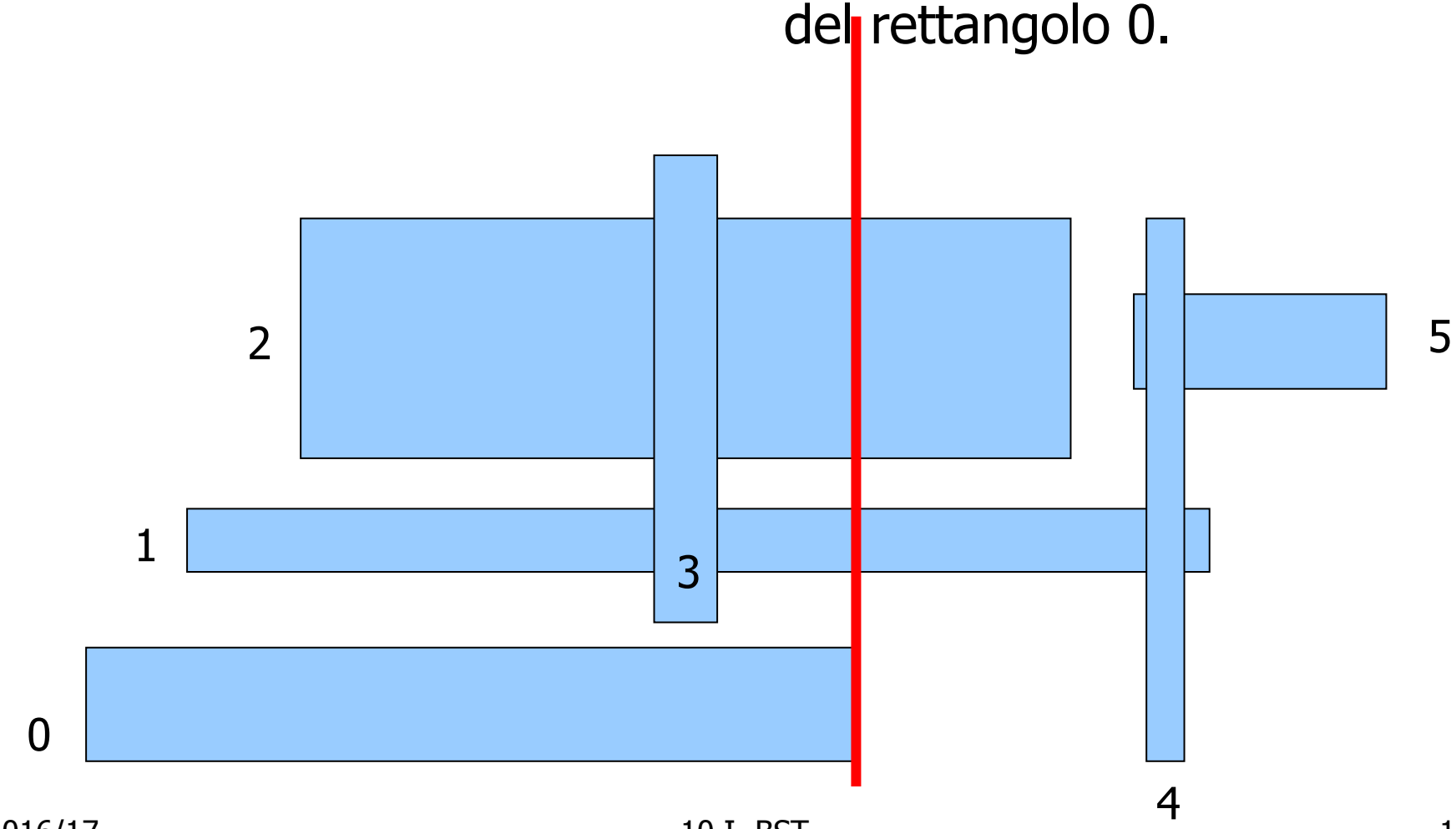
inserisci intervallo
ordinate del rettangolo 3
intersezioni con 1 e 2.



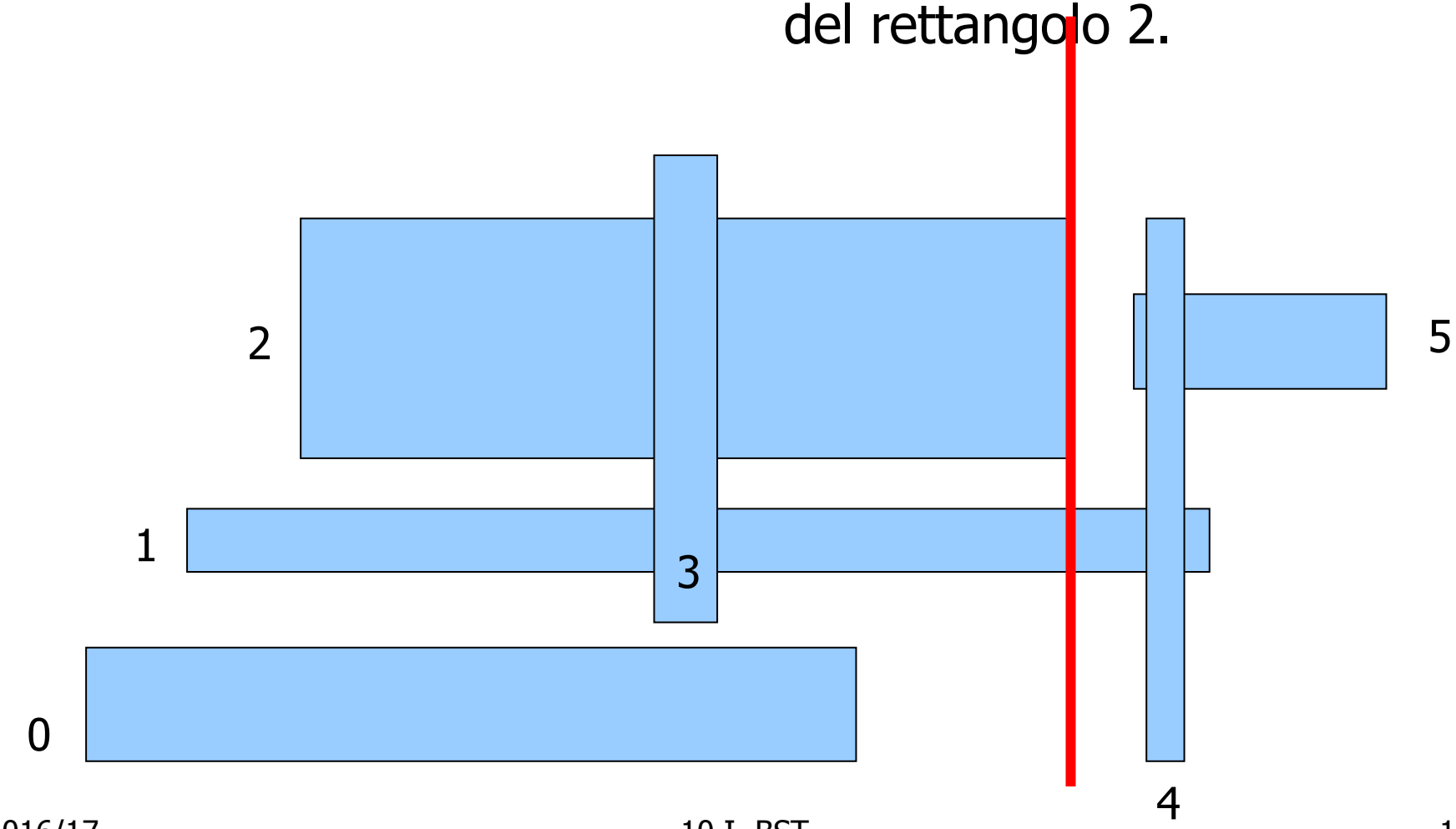
rimuovi intervallo ordinate del rettangolo 3.



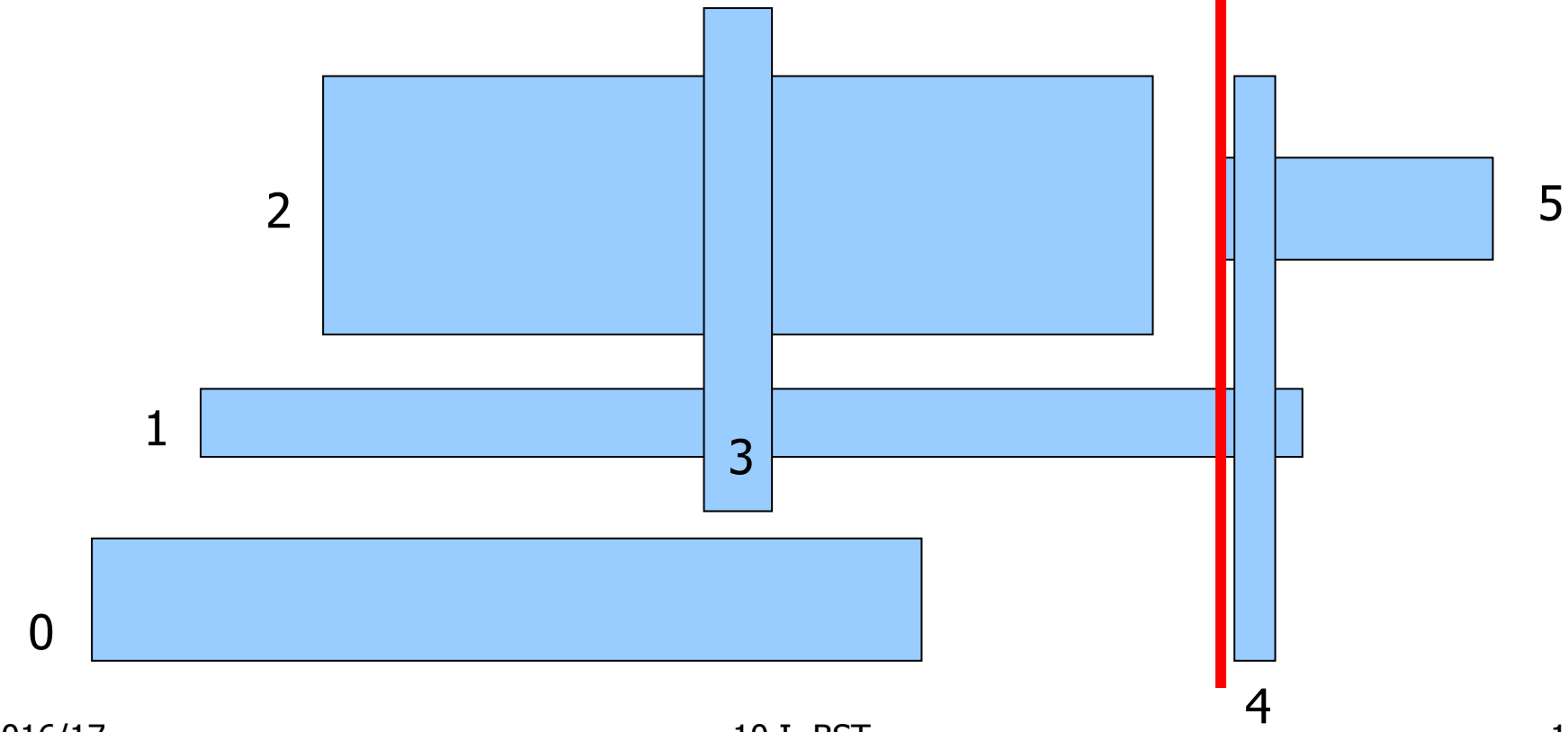
rimuovi intervallo ordinate del rettangolo 0.



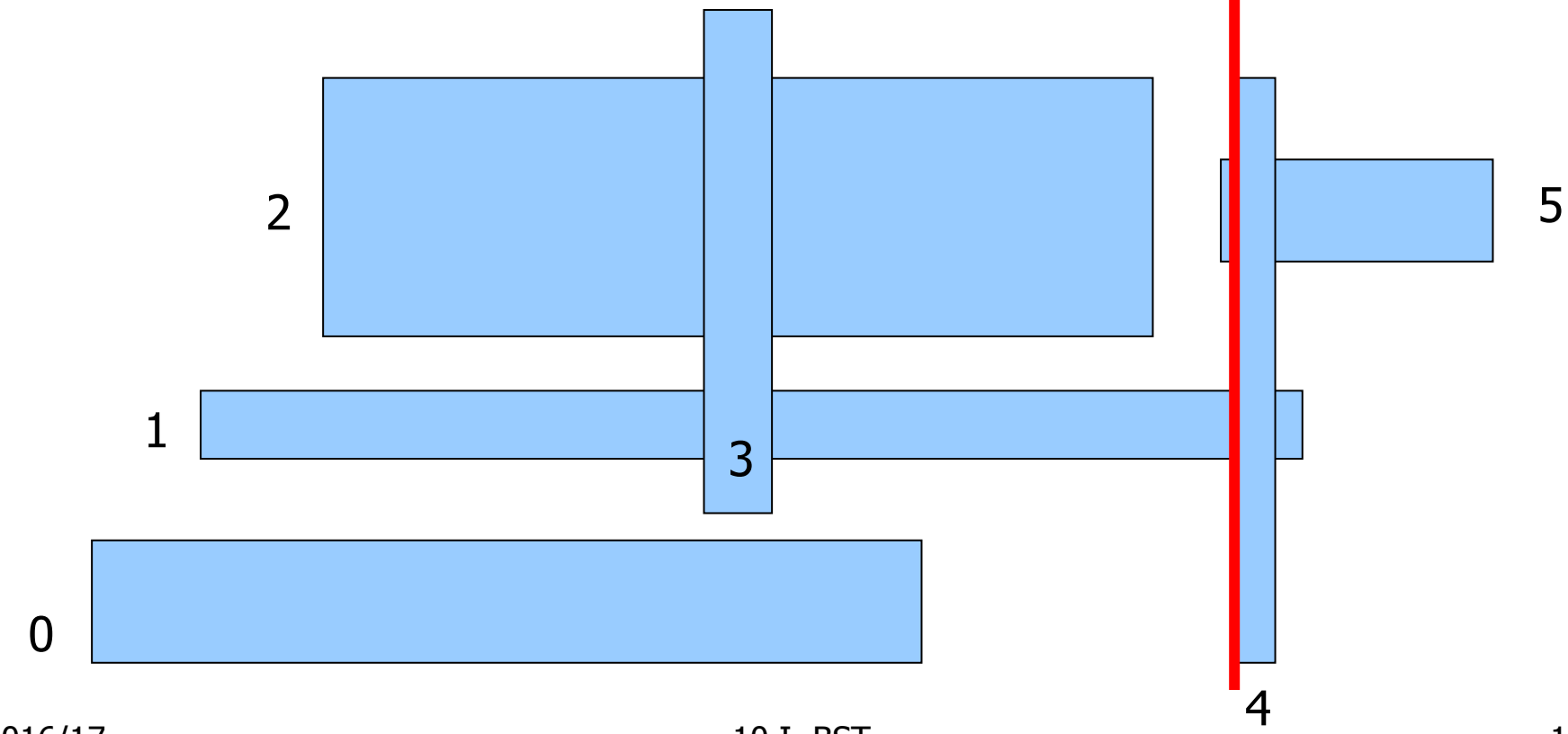
rimuovi intervallo ordinate del rettangolo 2.



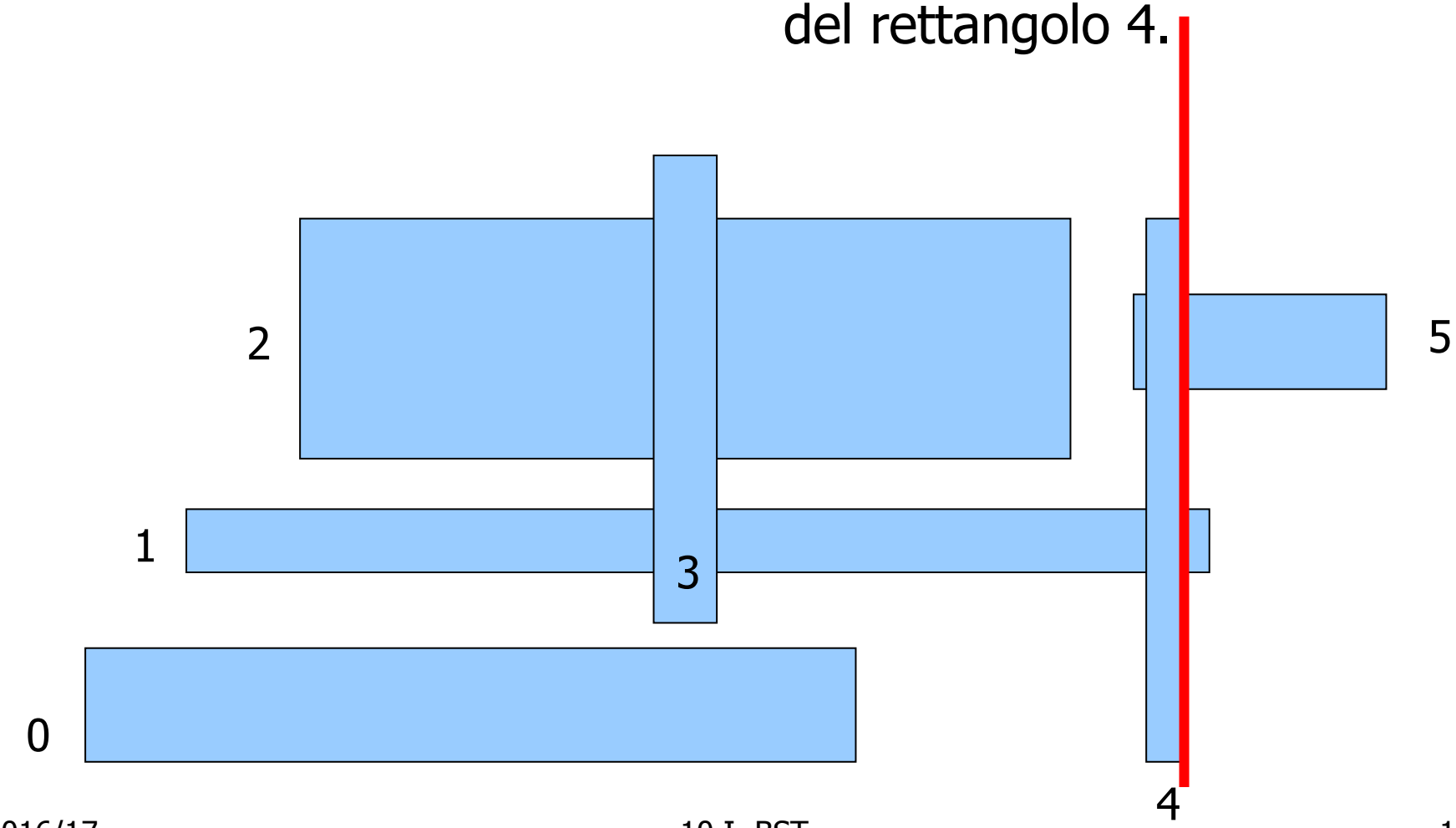
inserisci intervallo
ordinate del rettangolo 5
no intersezioni.



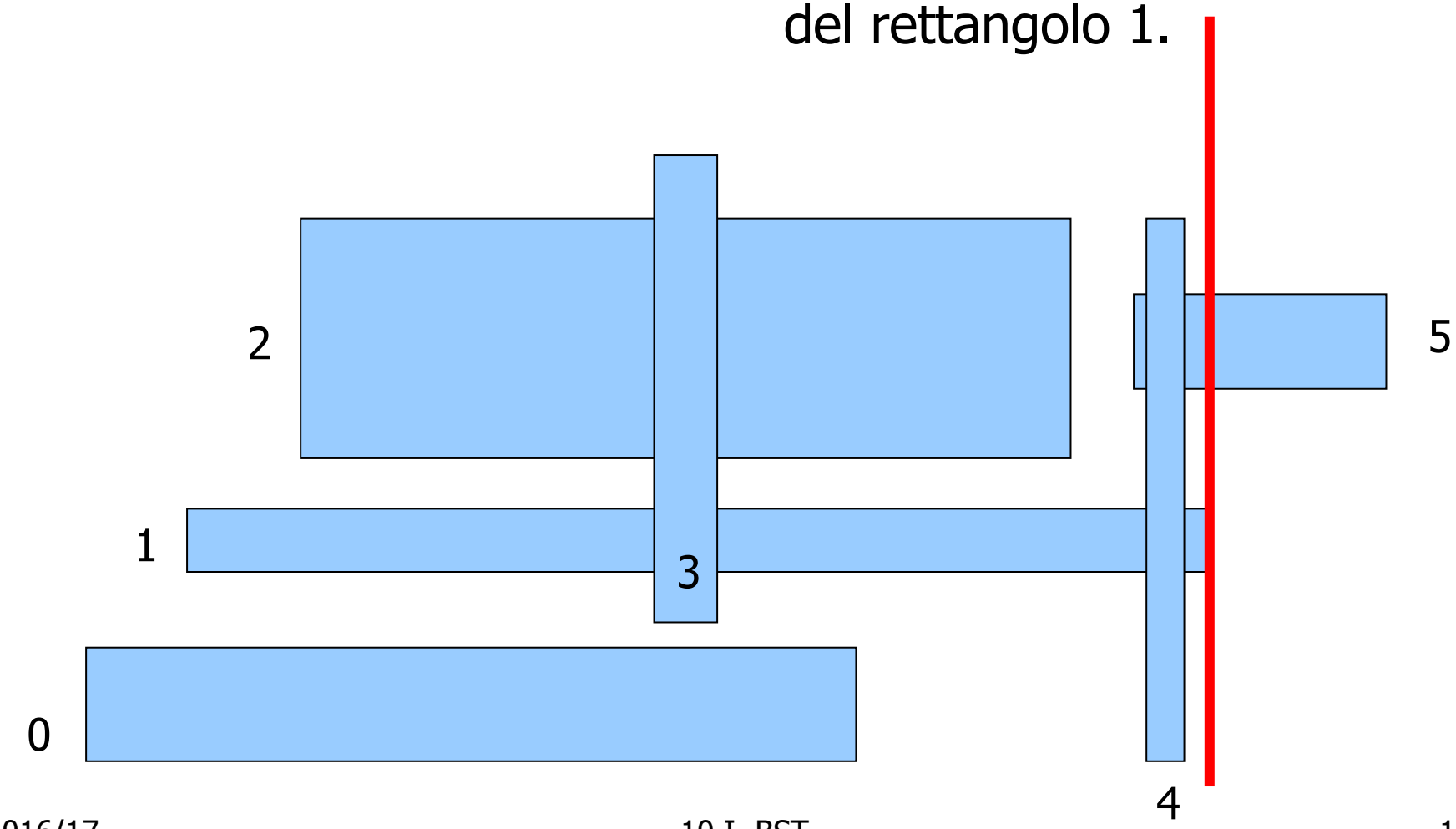
inserisci intervallo
ordinate del rettangolo 4
intersezioni con 1 e 5.



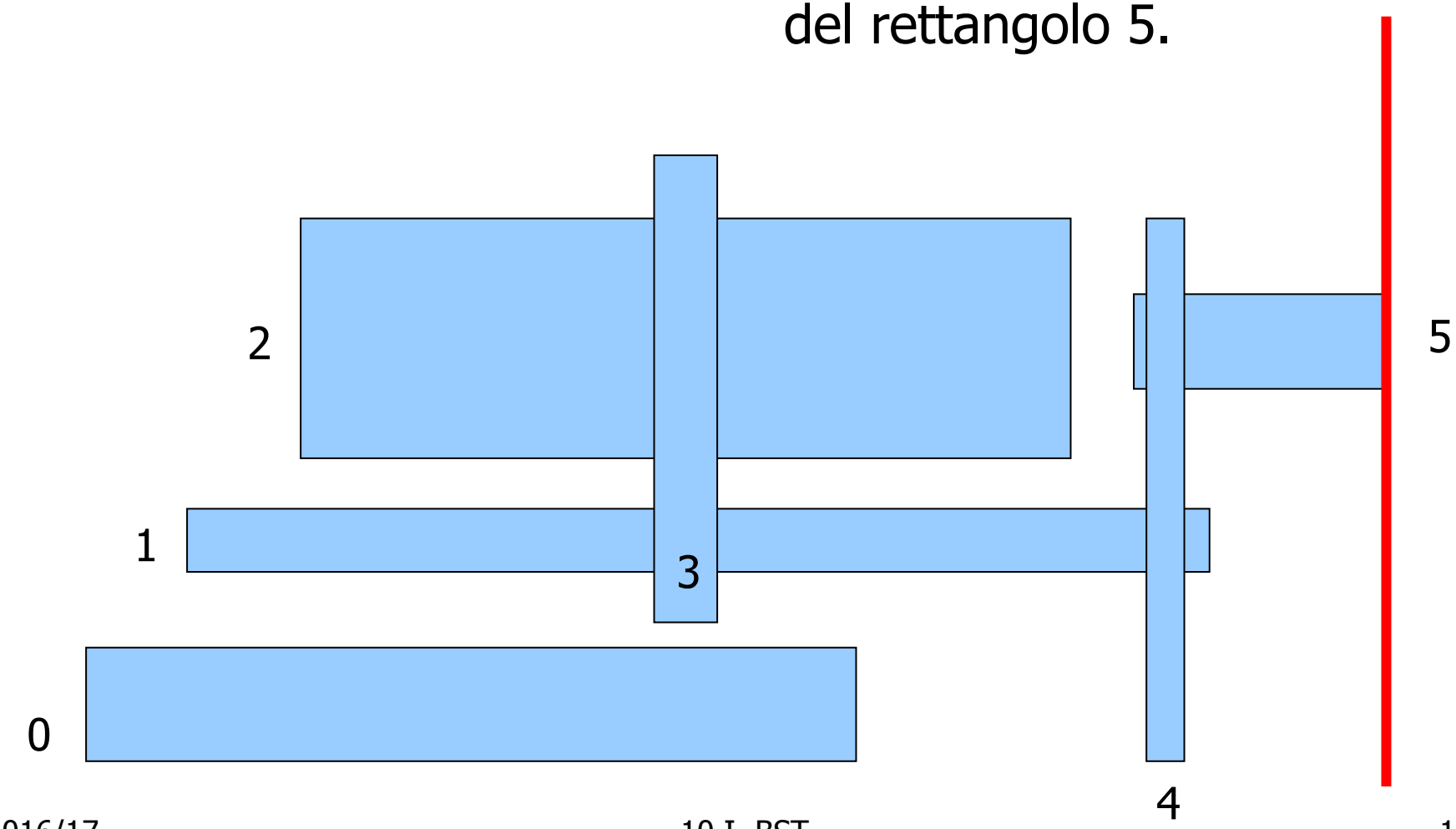
rimuovi intervallo ordinate del rettangolo 4.

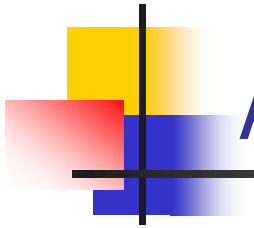


rimuovi intervallo ordinate del rettangolo 1.



rimuovi intervallo ordinate del rettangolo 5.



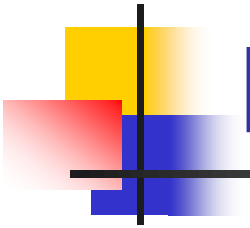


Analisi

Ordinamento: O(NlogN)

Se l'IBST è bilanciato:

- ogni inserzione/cancellazione costa O(logN), quindi per N rettangoli O(NlogN)
- N ricerche con R intersezioni costano O(NlogN + RlogN).



Riferimenti

- Alberi binari
  - Sedgewick 5.6, 5.7
- Binary Search Tree
  - Cormen 13.1, 13.2, 13.3
  - Sedgewick 12.5, 12.8, 12.9
- Order-statistic BST
  - Cormen 15.1
- Interval BST
  - Cormen 15.3