Esercizio 1

Si introduca una nuova operazione sugli alberi binari di ricerca, incrChiave(T,k,d).
Questa operazione, applicata ad un AVL T contenente un nodo con chiave k, restituisce l'AVL T in cui la chiave k è incrementata di d.
Il tempo di esecuzione dell'operazione deve essere O(log n), dove n è il numero dei nodi nell'albero.

Esercizio 1: soluzione

Per realizzare l'operazione, incrChiave(T,k,d), nei tempi prescritti, basta cancellare il nodo di chiave k e inserire il nodo di chiave k+d, usando inserimenti e cancellazione su AVL.

Esercizio code di priorità

Si costruisca un algoritmo per fondere in un unico array ordinato k arrays ordinati.
Il numero totale di elementi è n. L'algoritmo deve girare in O(n lg k).

Suggerimento: si usi un min-heap di k elementi per la fusione.

Esempio

B1	4	9	10	20	**B2**	3	5	**B3**	1	8	21	40	**B4**	15	50	80

All'inizio si inseriscono le terne (x,0,j) in un array MIN, che con la build-min-heap diventa un min-heap. La terna è necessaria perché si deve poter risalire da ogni elemento, x, all'array di appartenenza, j, e alla sua posizione in quell'array, 0. Il costo qui è O(k).

MIN	(1,0,3)	(3,0,2)	(4,0,1)	(15,0,4)

Poi si entra in un ciclo che consente di riempire tutto l'array A. Si estrae il minimo, (x,i,j) e così si ottiene l'elemento, x , da inserire nell'array finale A, poi si inserisce l'elemento (Bj[i+1],i+1,j) nel min-heap.
Naturalmente si deve ogni volta controllare che il successivo elemento da inserire nel min-heap ci sia in Bj. Se Bj è "esaurito" si passa direttamente all'estrazione del minimo nel min-heap e si prosegue come descritto sopra.

Esercizio code di priorità: sol

In input abbiamo le k sequenze ordinate B1,…,Bk.
Si definisce un min heap di k elementi, Min, e un array, A, di n elementi che conterrà tutti gli elementi ordinati.
Si copiano i primi k elementi di ogni sequenza ordinata in Min, ma ricordando per ogni elemento la posizione e l'array di provenienza, per esempio in una terna (x,i,j) dove x è l' elemento di indice i di Bj, per 0≤ j ≤ k-1 e 0 ≤ i ≤ Bj.length. Con Buil-Min-heap si trasforma Min in un min-heap di k elementi, dove il confronto è solo sulla prima componente delle terne.
Si entra in un ciclo for su un indice h di scorrimento di A, da 0 a n-1, nel quale si estrae il minimo dal min-heap, (x,i,j), lo si mette in A[h] e, se l'array Bj contiene altri elementi si inserisce nel min-heap (y,i+1,j), dove y = Bj[i+1], l'elemento successivo a quello inserito.
La fase di inizializzazione del min heap costa O(k). Poiché estrazione del minimo e inserimento in un min-heap di k elementi costano lg k, tutto l'algoritmo ha un costo O(n lg k).

Esercizio ABR

Scrivete un algoritmo che, preso in input un ABR T, restituisce il puntatore all'elemento mediano nell'albero.
Se T ha n nodi e se n è dispari ci sono esattamente n/2 più grandi e n/2 più piccoli, se n è pari ci sono n/2-1 più piccoli e n/2 più grandi, dove n/2 indica la parte intera della divisione.
Il mediano può essere individuato visitando ogni nodo una sola volta, tranne al più 2h nodi.

Soluzione. L'idea è di fare due visite dell'albero: una da sinistra partendo dal minimo e proseguendo calcolando il successivo di ogni nodo e l'altra da destra partendo dal massimo e proseguendo calcolando il precedente.

Le due visite devono terminare, se n è pari quando il nodo visitato da "sinistra" è uguale al nodo visitato da "destra", se n è dispari quando il successivo del nodo corrente nella prima visita è uguale al nodo corrente nella visita da "destra".

MedianoABR(T)
input : T è un albero binario
prec: T è un ABR
postc: restituisce il puntatore al nodo che contiene il valore mediano.

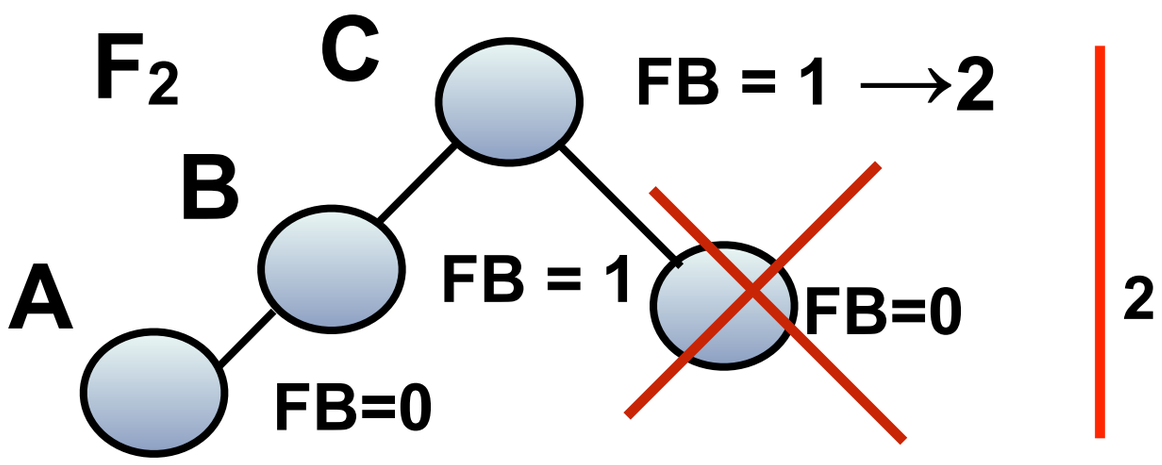
```
if T == NULL  or T è una foglia return T
x = MINIMUM(T)
y = MAXIMUM(T)
z = SUCCESSOR(x)
if z==y then return x   // il caso di due nodi,
w = PREDECESSOR(y)
while (z ≠ NIL and w≠ NIL and (SUCCESSOR(z) ≠ w  or z ≠ w)) do
z è il successivo di x in T e w il precedente di y in T
    x = z
    z = SUCCESSOR(x)
    y = w
    w = PREDECESSOR(y)
return z
```

Esercizio cancellazione

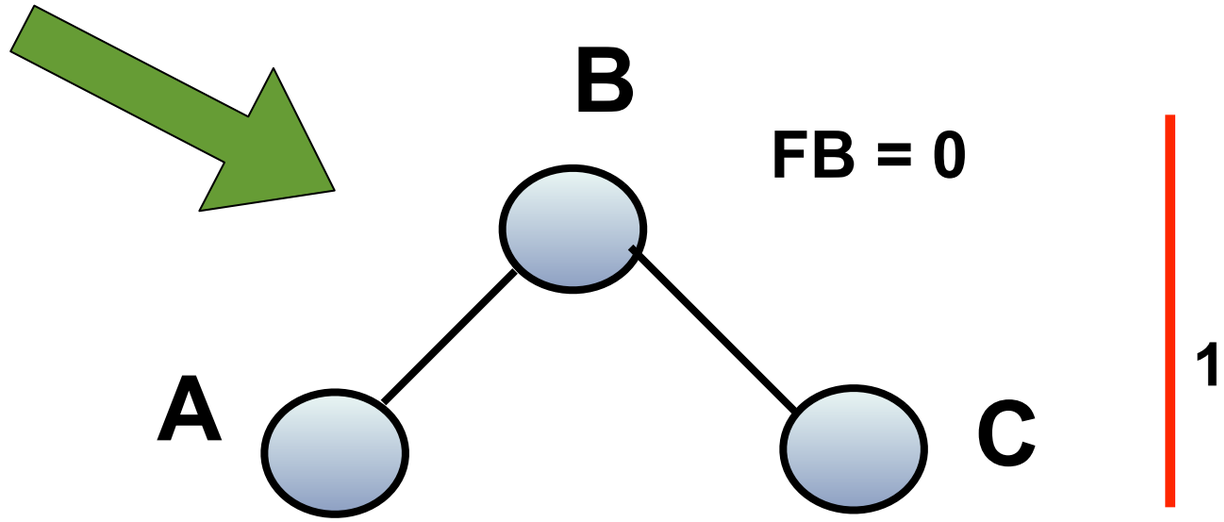
In quale caso la cancellazione di un nodo in un AVL comporta la necessità di fare un numero di rotazioni ribilancianti che dipende dall'altezza dell'albero?

Quante rotazioni esattamente sono necessarie in funzione dell'altezza?

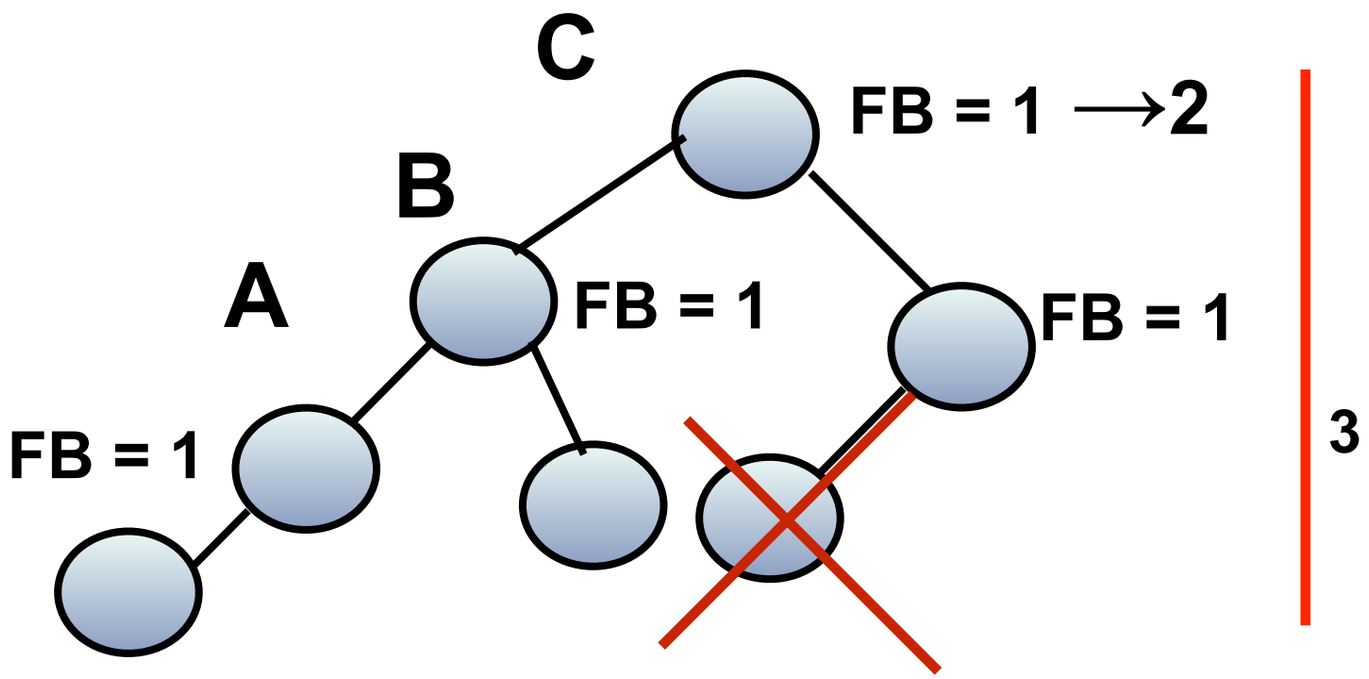
cancellazione sbilanciante in F_2



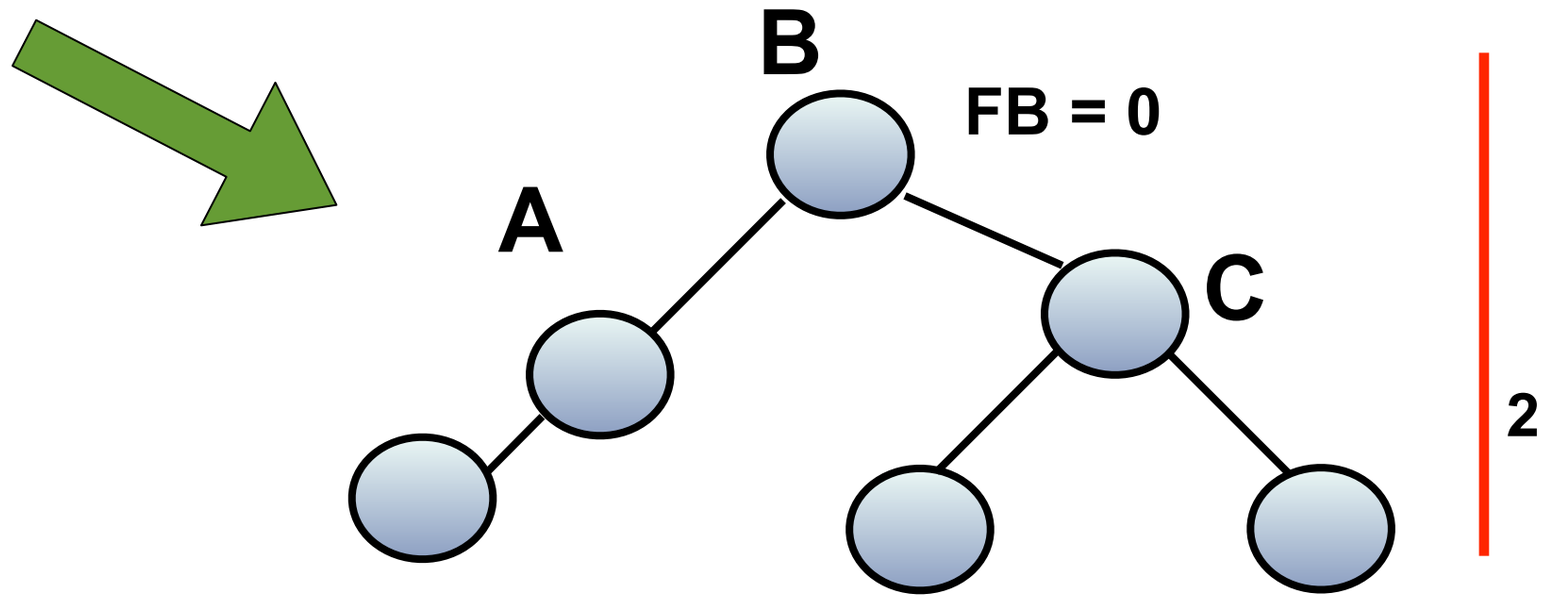
cancellazione e rotazione in F_2



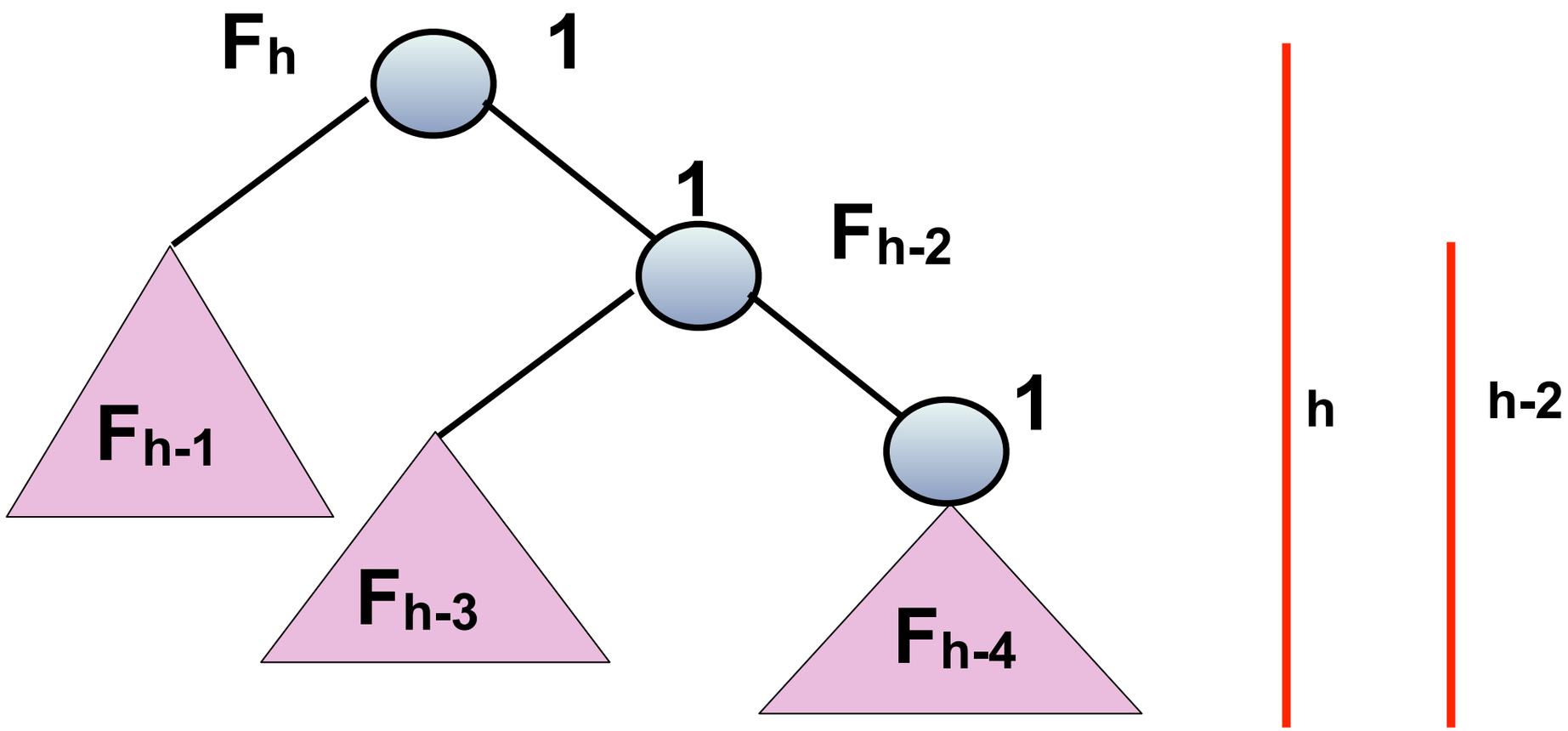
La cancellazione in F_3



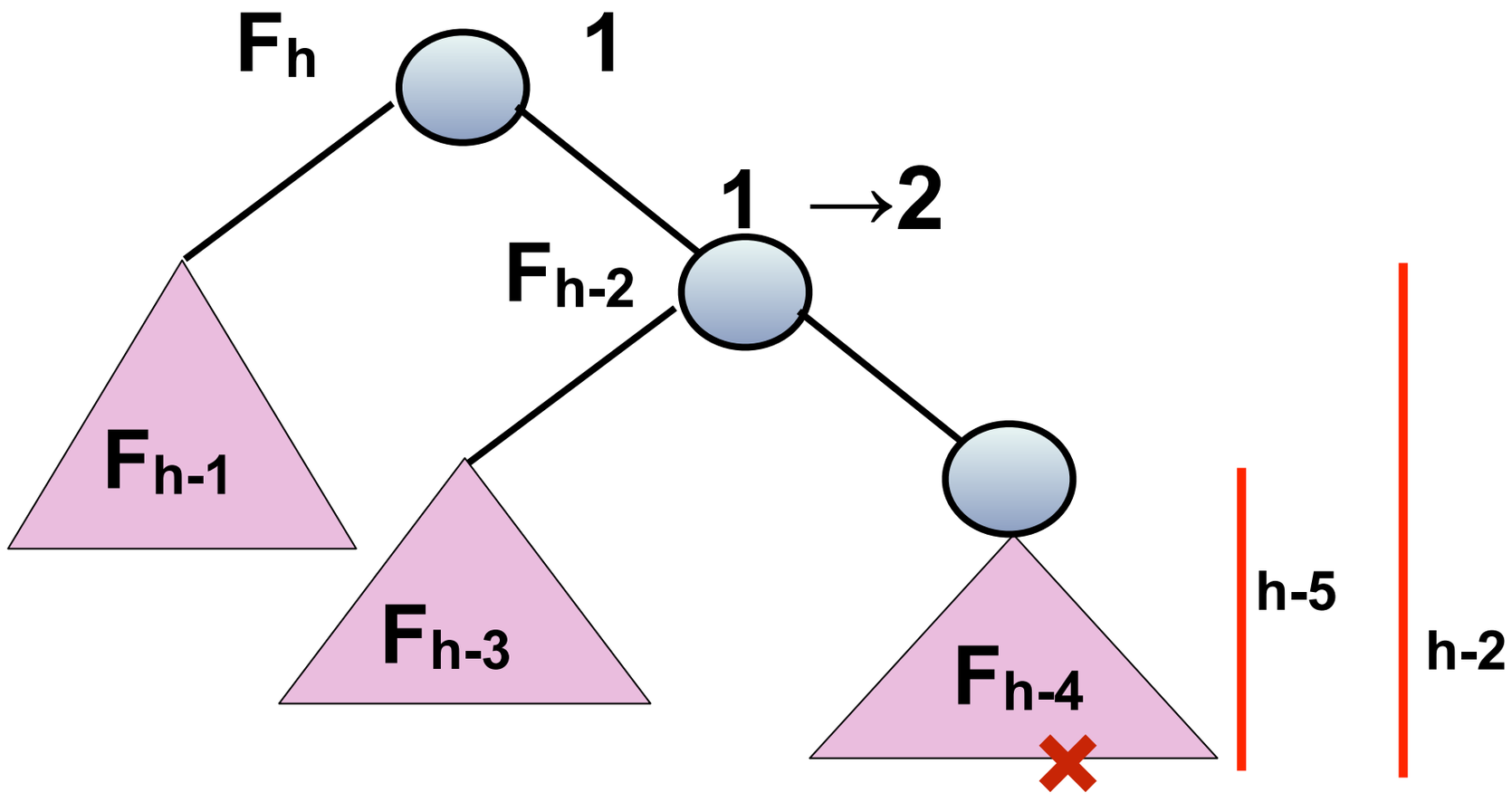
cancellazione e rotazione in F_3



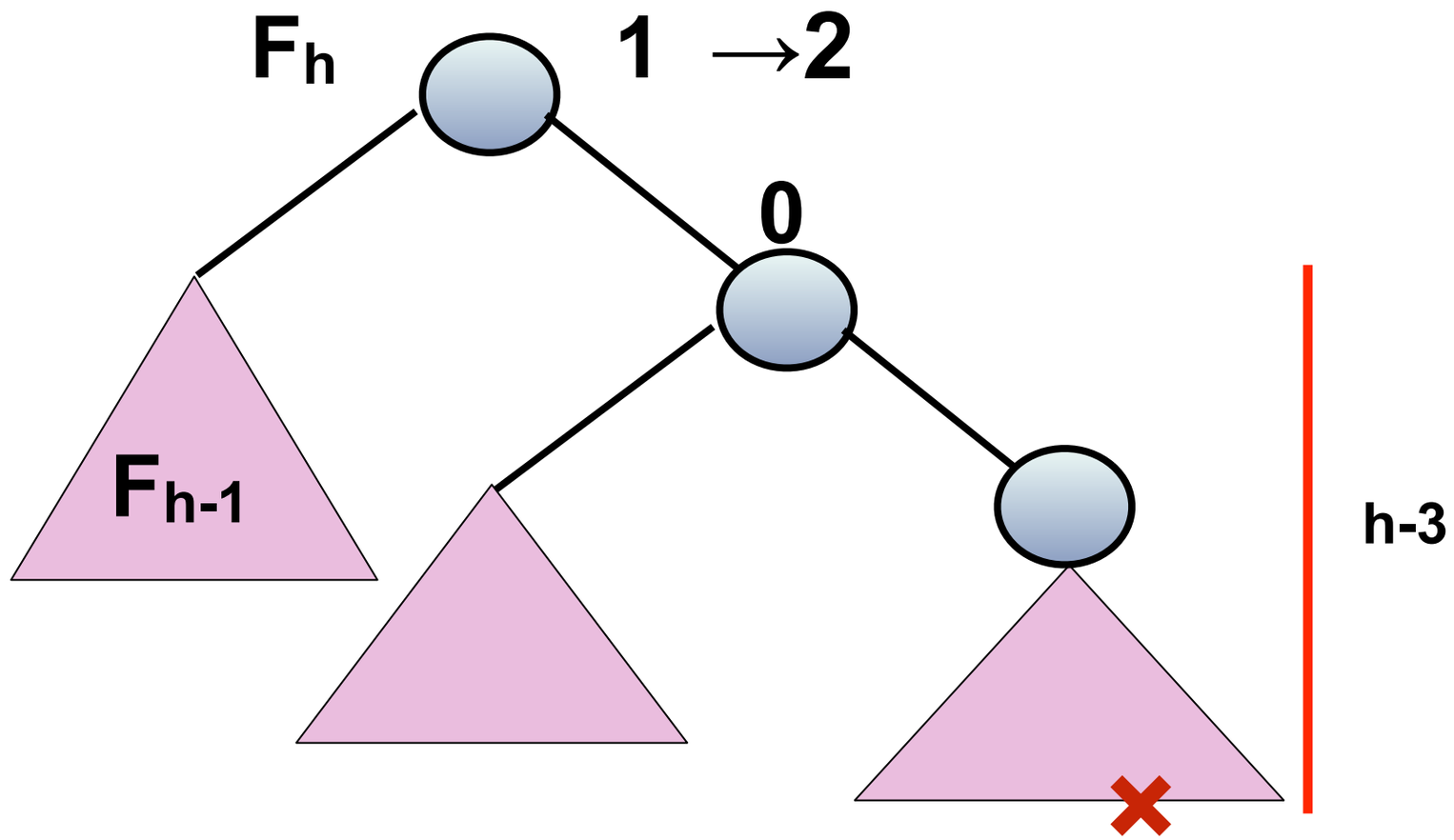
Cancellando un nodo nel sotto albero destro F_{h-2} di F_h ed effettuando le opportune rotazioni, si ottiene un sotto albero destro ribilanciato ma di altezza di uno inferiore.



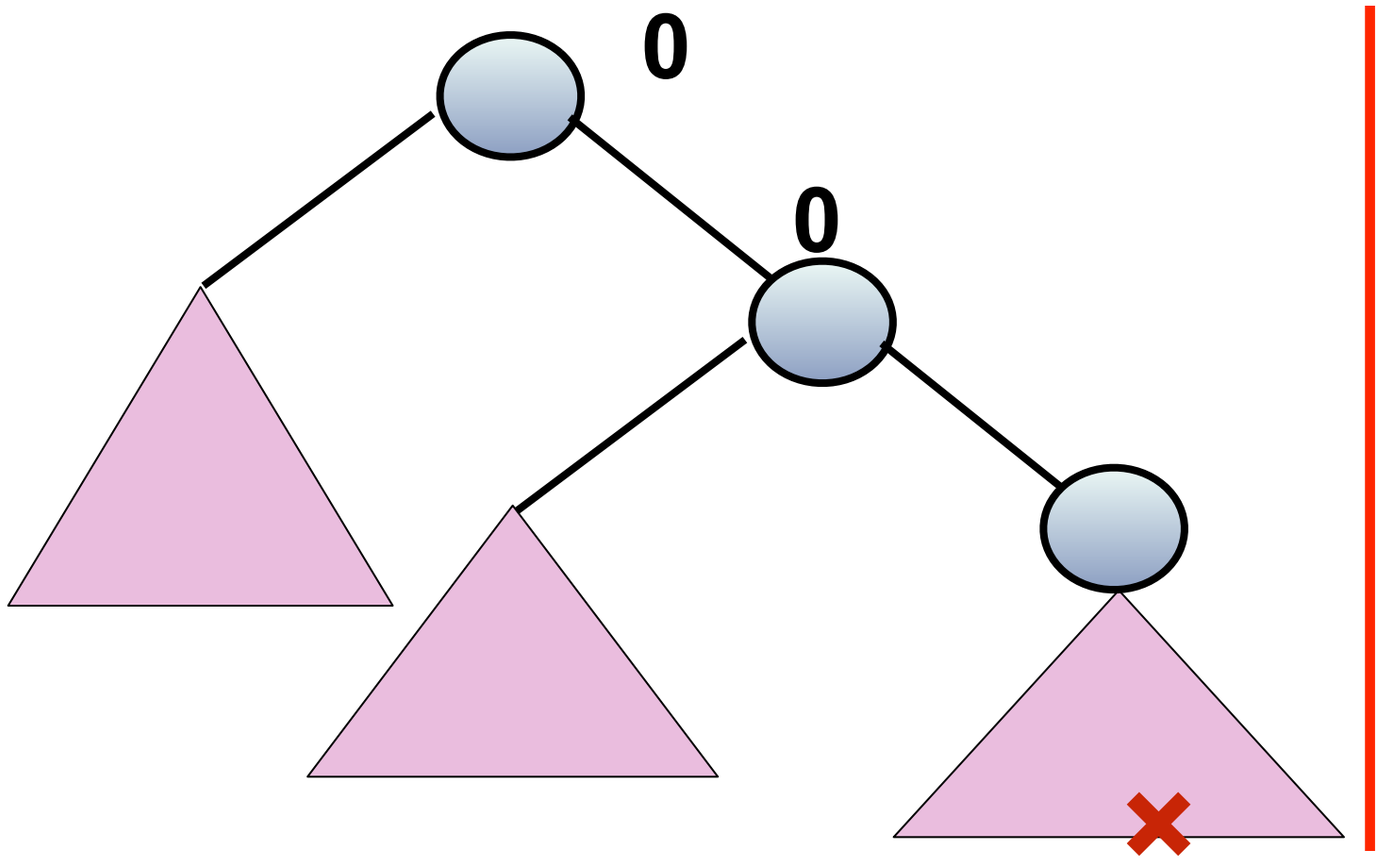
Alberi di Fibonacci: cancellazione nodo sotto albero destro che riduce l'altezza di 1



Dopo la rotazione il sotto albero destro è ribilanciato ma di altezza h-3, quindi anche la radice si sbilancia e serve una nuova rotazione



Dopo la rotazione l'albero di Fibonacci di altezza h è ribilanciato ma di altezza h-1.



F_2 e F_3 si ribilanciano con 1 rotazione, in generale F_h si ribilancia con h/2 rotazioni, pr ogni $h \geq 2$

Per induzione.

Passo induttivo:

F_{h-2} si ribilancia con (h-2)/2 rotazioni, per ipotesi induttiva, ma poi basta una rotazione per ribilanciare tutto l'albero, quindi il numero totale di rotazioni necessarie è h/2 - 1 + 1 = h/2