Alberi Binari di Ricerca

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **17(a)** [**21/11/23**]

SAPIENZA
UNIVERSITÀ DI ROMA

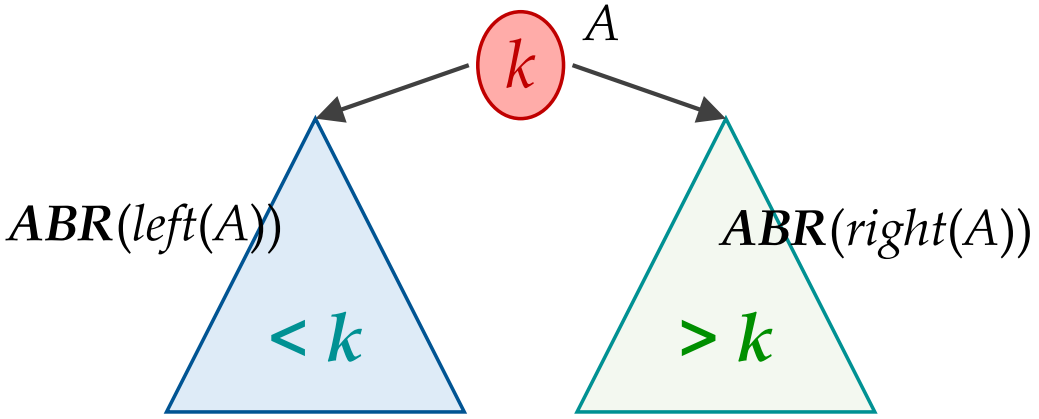
Alberi binari di ricerca

Un **albero binario di ricerca [ABR]** è un albero binario in cui:

- la **chiave nella radice** è **maggiore** di **tutte** le **chiavi** contenute nel **sottoalbero sinistro** e **minore** di **tutte** le **chiavi** contenute nel **sottoalbero destro**.
- i **due sottoalberi** sono **ABR**.

Gli alberi binari di ricerca si prestano a rappresentare bene **insiemi dinamici** e **code con priorità**. Come tale, **assumeremo sempre** le **chiavi** di un ABR siano tutte **distinte**.

Vedremo le implementazioni di inserimenti, rimozioni, e di ricerche, *min*, *max*, *pred* e *succ* sono $\Theta(h)$, cioè **proporzionali all'altezza dell'albero**.



ABR: qualche def. e proprietà

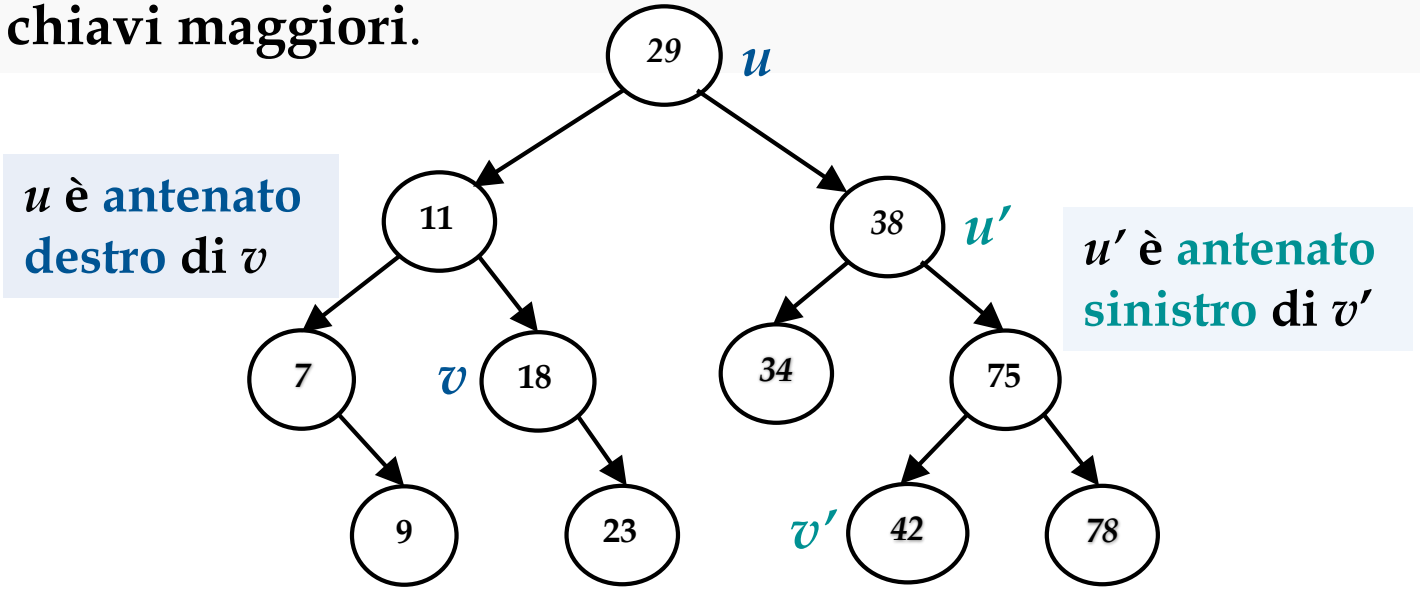
Diciamo che un nodo u è un **antenato destro** [**sinistro**] di un nodo v se $v \in left(u)$ [$v \in right(u)$]. Simmetricamente diremo che v è un discendente sinistro [destro] di u.

In un ABR, ho sempre che $key(v) < key(u)$ se u è un antenato destro di v, e $key(v) > key(u)$ se u è un antenato sinistro di v.

Osserviamo che la **visita in-order** di un **albero binario di ricerca produce** una **sequenza di chiavi** ordinata in modo **crescente**.

Infatti, in una visita in-order ogni nodo **appare dopo i suoi antenati sinistri** e **dopo i suoi discendenti sinistri**, che hanno **chiavi minori**.

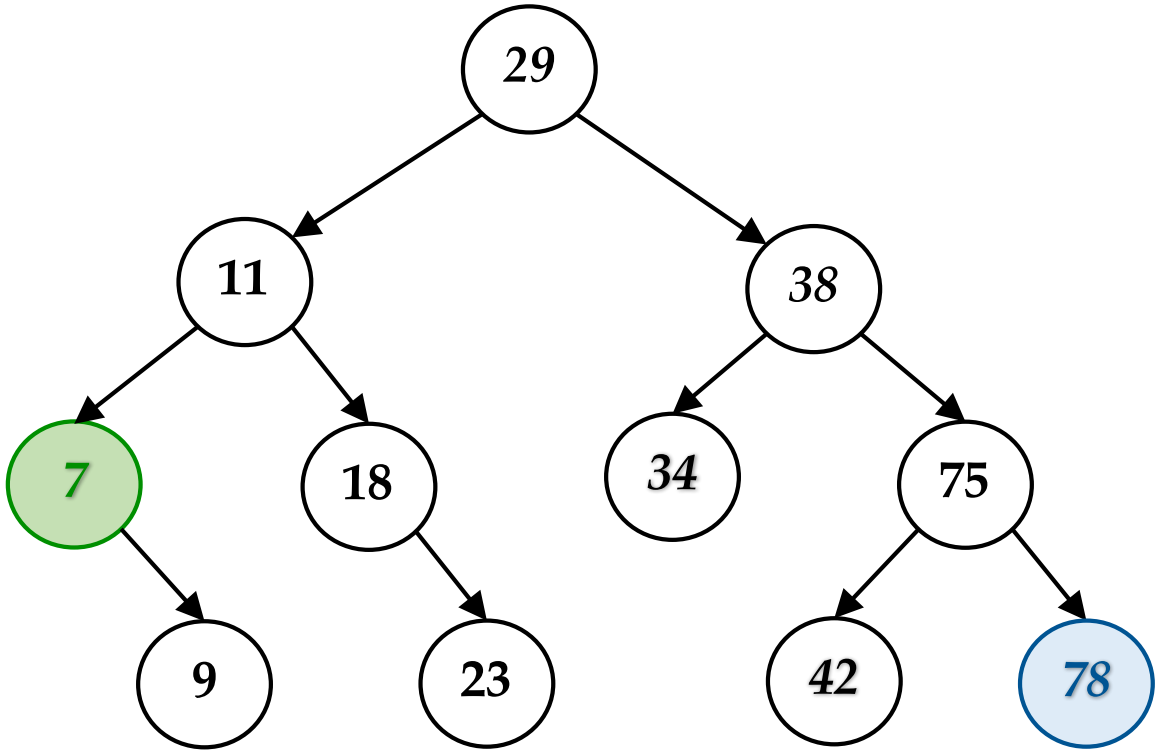
Appare **prima degli antenati destri** e dei **discendenti destri** che hanno **chiavi maggiori**.



minimo/massimo di un ABR

In un albero binario di ricerca, l'elemento **minimo** [risp. **massimo**] è necessariamente un nodo **raggiungibile** dalla radice andando **solo a sinistra** [risp. **solo a destra**].

Il **minimo** non **ha mai discendenti sinistri**, **né antenati sinistri** ed è il primo nodo in una visita inorder. Analogamente, il **massimo** **non ha antenati/discendenti destri**, ed è l'ultimo in una visita inorder.



min/max ABR: pseudocodice

Lo pseudocodice è molto semplice, sia in versione **ricorsiva** (di cui mostriamo il **minimo a sinistra**) che **iterativa** (mostriamo il **massimo a destra**).

Di fatto si percorre **linearmente cammino** tutto a **sinistra per il minimo** e del cammino tutto a **destra per il massimo**.

Siccome percorre un cammino, ha **complessità $\Theta(h)$**

$\Theta(h)$

```
def minABR(B):
# REQ: B è ABR
    if B == EMPTY:
        return +∞
    if B->lft == EMPTY:
        return B->key
    return minABR(B->lft)
```

$\Theta(h)$

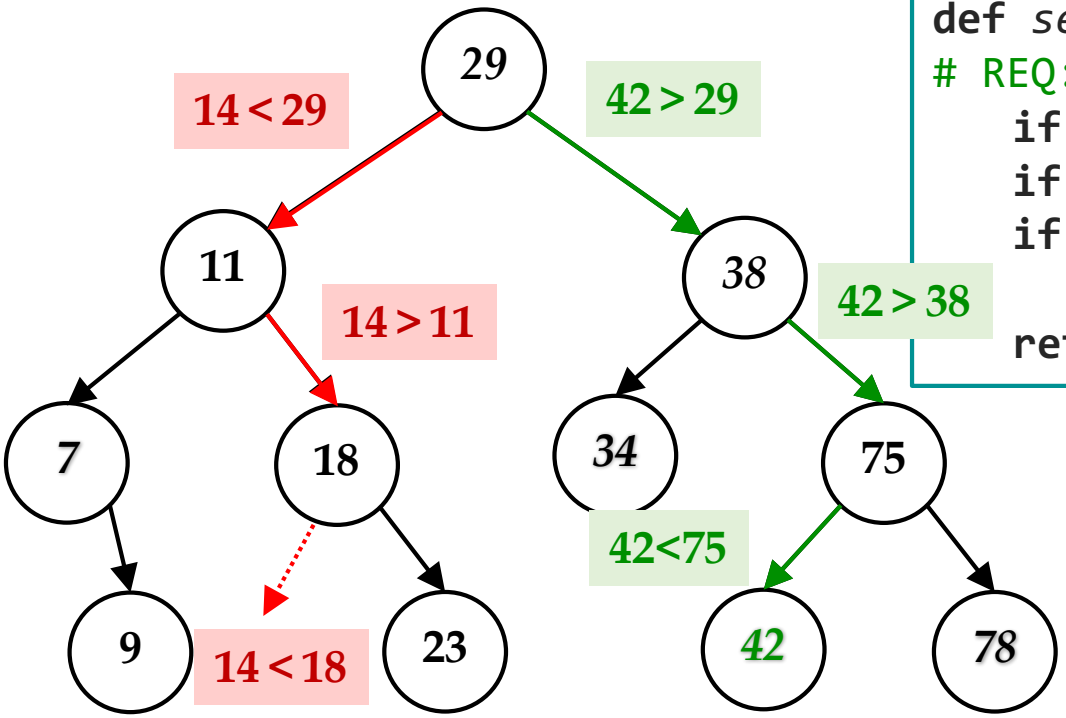
```
def maxABR(B):
# REQ: B è ABR
    if B == EMPTY:
        return -∞
    while B->rgt ≠ EMPTY:
        B = B->rgt
    return B->key
```

Ricerca in un ABR

Un albero binario di ricerca è naturalmente pensato per una **sorta di ricerca binaria:** in ogni nodo B, cercando una chiave x ho 3 casi:

- $key(B) = x$: ho finito con **successo** e **torno B**;
- $key(B) < x$: x, **se c'è**, si **trova** necessariamente **a destra**;
- $key(B) > x$: x, **se c'è**, si **trova** necessariamente **a sinistra**.

Se B è EMPTY **ho terminato** la ricerca **con insuccesso**




```
def searchABR(B, x):
# REQ: B è ABR
    if B == EMPTY: return NULL
    if B->key == x: return B
    if B->key > x:
        return searchABR(B->lft, x)
    return searchABR(B->rgt, x)
```

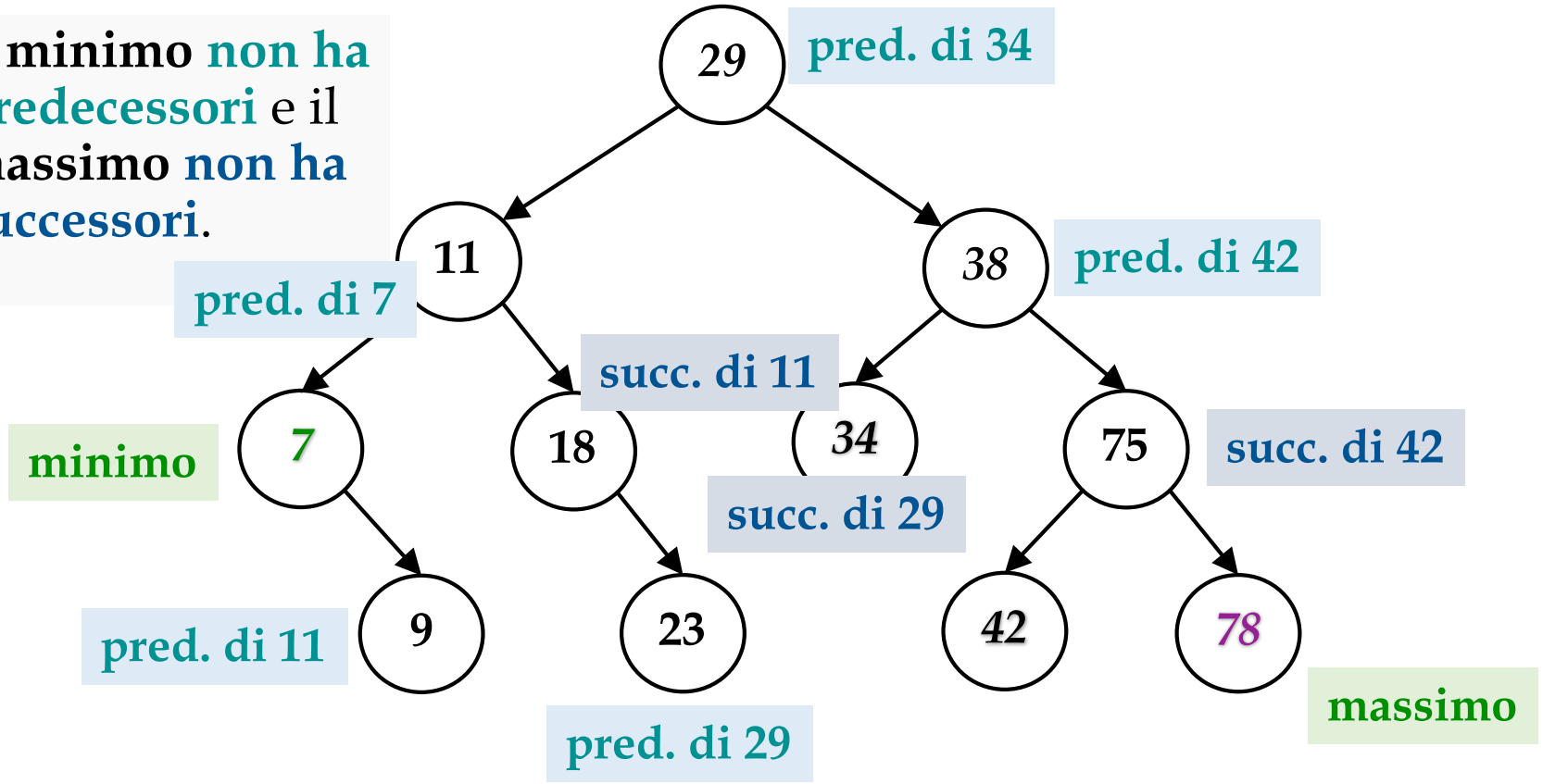
$\Theta(h)$

pred/succ in un ABR

Il **predecessore** [**successore**] di una chiave *k* è:

- il **massimo** [**minimo**] del sottoalbero **sinistro** [**destro**] se *k* è in un **nodo con sottoalbero sinistro** [**destro**],
- il **più vicino antenato sinistro** [**destro**], se *k* è in un nodo senza sottoalbero **destro** [**sinistro**].

Il **minimo non ha predecessori** e il **massimo non ha successori**.



pred/succ in ABR: pseudocodice

Lo pseudocodice di **succ** sfrutta le **proprietà viste**. Si cerca l'elemento x: se il nodo ha il **sottoalbero destro**, si calcola $min(right(x))$. Altrimenti si restituisce x (questo **fa capire che il predecessore non è stato trovato**) e risalendo si setta il predecessore quando **si torna dall'ultima mossa a destra**.

Il calcolo del **pred** è del tutto **simmetrico** [⏵ **Esercizio**].

$\Theta(h)$

```
def succ(B, x):
# REQ: B è ABR, x ∈ B
    p = succAux(B, x)
    if p == x: return -∞ # x è min(B)
    return p
```

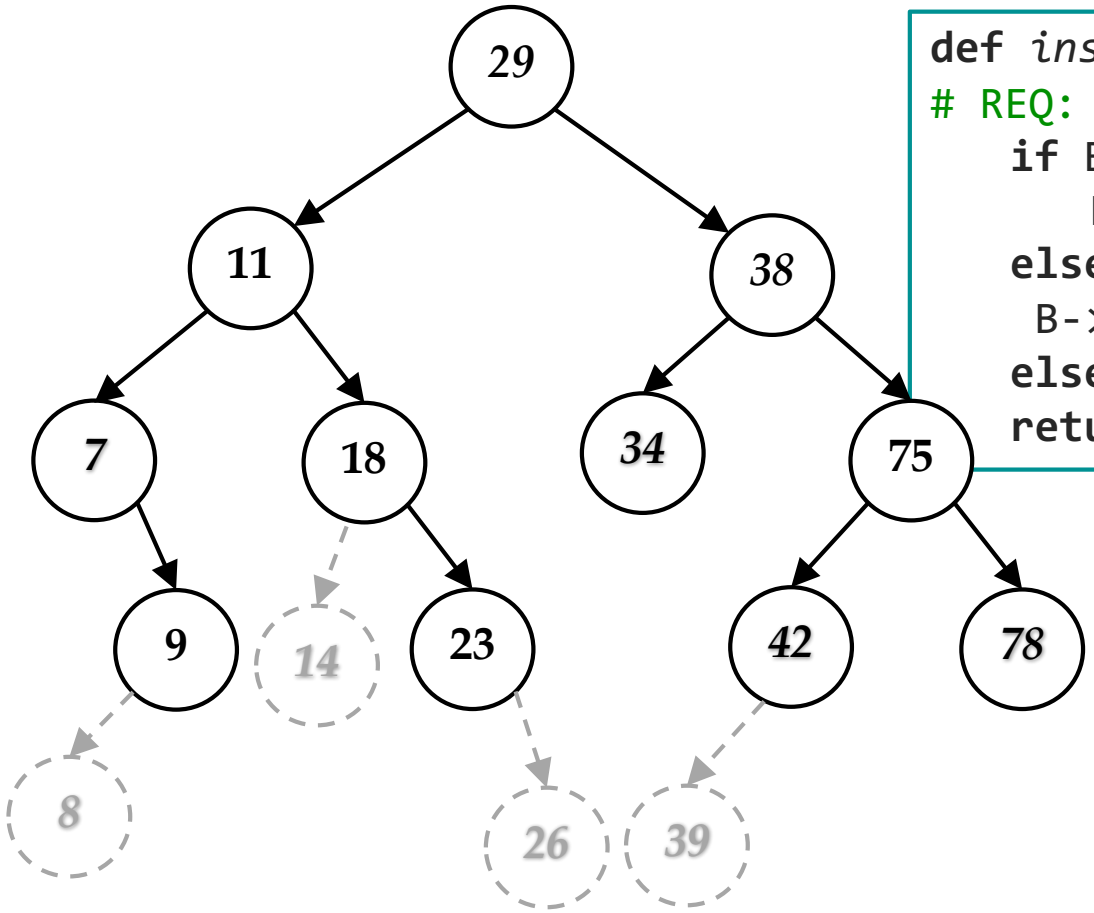
```
def succAux(B, x):
# REQ: B è ABR, x ∈ B
    if B->key == x:
        if B->rgt ≠ EMPTY:
            return min(B->rgt))
        else: return x
    if B->key > x:
        return succAux(B->lft, x)
    p = succAux(B->rgt, x)
    if p == x: return B->key
    return p
```

Iterativamente è **più complicato** [⏵ **Esercizio**]: o si **mantiene la sequenza delle mosse** o si sfrutta il **pointer up** (se presente) per sapere se una mossa è a destra (se $x = up(left(x))$) o a sinistra (se $x = up(right(x))$).

Inserimento in un ABR

Per inserire una chiave x in un ABR, è sufficiente trovare **l'unico punto** in cui l'aggiunta di x **mantiene la proprietà di ABR**.

Quindi, è **come una ricerca**: la differenza è che quando arrivo a NULL è il momento di fare l'inserimento, altrimenti significa che il nodo è già presente e **non faccio nulla**.



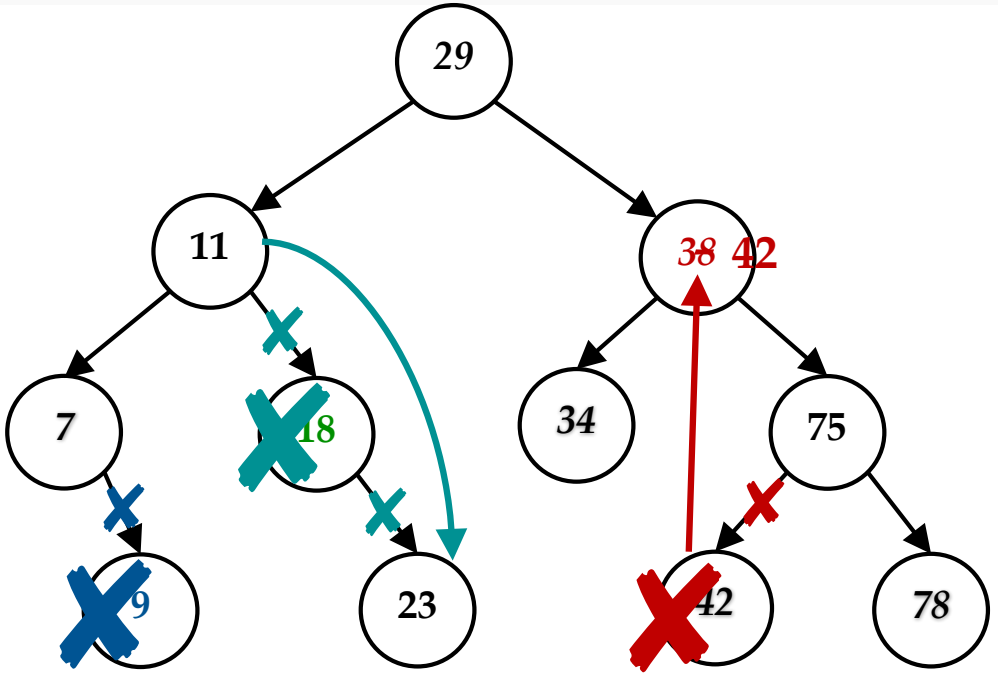

```
def insABR(B, x):
# REQ: B è ABR, x ∉ B
    if B == EMPTY:
        B = mkLf(x)
    else if B->key > x:
     B->lft = insABR(B->lft, x)
    else B->rgt = insABR(B->rgt, x)
    return B
```

$\Theta(h)$

Rimozione in un ABR

Questa è l'operazione **più difficile**, perché occorre **garantire** di **mantenere** la proprietà di essere un **ABR**. Distinguiamo 3 casi:

- **foglia**: è sufficiente rimuovere il nodo (e il pointer al nodo)
- **1 figlio**: è sufficiente sostituire il nodo con il figlio (**cortocircuito**)
- **2 figli**: la chiave del nodo si può **sostituire col predecessore** o il **successore** (e tale nodo va rimosso dal suo posto).
  Osservare che in **questo caso** successore e predecessore **esistono necessariamente** e sono in **uno dei due casi precedenti**!



rimozione da ABR: pseudocodice

$\Theta(h)$

```
def remove(B, x):
# REQ: B è ABR, x ∈ B
    if B->key == x: # nodo da rimuovere
        if isLeaf(B):
            delete(B)
            return NULL
        if B->rgt == EMTPY:
            tmp = B->lft
            delete(B)
            return tmp
        if B->lft == EMTPY:
            tmp = B->rgt
            delete(B)
            return tmp
        m = min(B->rgt)
        B->key = m
        B->rgt = remove(B->rgt, m)
        return B
    if B->key > x:
        B->lft = remove(B->lft, x)
    else: B->rgt = remove(B->rgt, x)
    return B
```

caso *foglia*

caso *un solo figlio* [*sinistro*]

caso *un solo figlio* [*destro*]

caso *due figli*

è *equivalente lavorare sul max del sinistro*

un minimo non ha mai figlio sinistro

caso *ricorsivo:* notare che al ritorno *riattacco i pointer*

Osservazioni e Variazioni sugli ABR

Operazioni come **`remove`** e **`insert`** (per assicurarsi di rispettare le precondizioni) andrebbero chiamate seguendo il protocollo:

```
if searchABR(B, x) ≠ NULL: remove(B, x)
if searchABR(B, x) == NULL: insABR(B, x)
```

[possono esserci altri motivi per cui sapete che $x \in B$ o $x \notin B$]

Soprattutto nel caso in cui **sia presente il pointer up**, qualche operazione si potrebbe **semplificare lavorando con i nodi** (in particolare **`remove`** che non necessiterebbe di una ricerca).

In tutte le funzioni (anche min/max, pred/succ) è in tal caso conveniente tornare un **pointer al nodo** invece del **valore**.

◗ **Esercizio**: modificare **`min`**/**`max`**/**`pred`**/**`succ`** in modo da restituire il nodo.

◗ **Esercizio**: modificare **`remove`**/**`insert`** avendo come parametro il nodo invece del valore da rimuovere e avendo il pointer up. Ricordatevi di risistemare quest'ultimo!