Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Dizionari: Alberi binari di ricerca

Angelo Monti

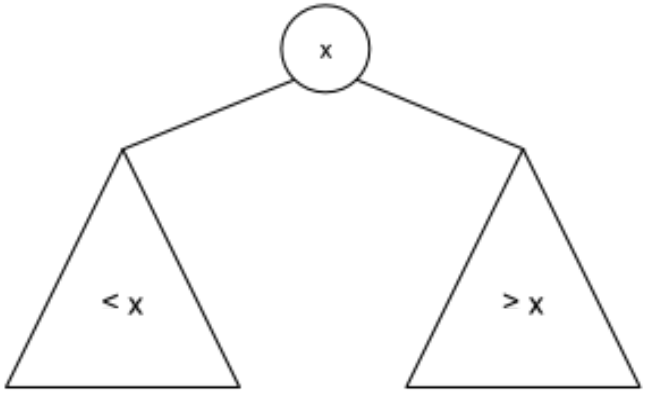
SAPIENZA
UNIVERSITÀ DI ROMA

Sulla base delle slides a cura di T. Calamoneri e G. Bongiovanni per il corso di informatica generale AA 2019/2020

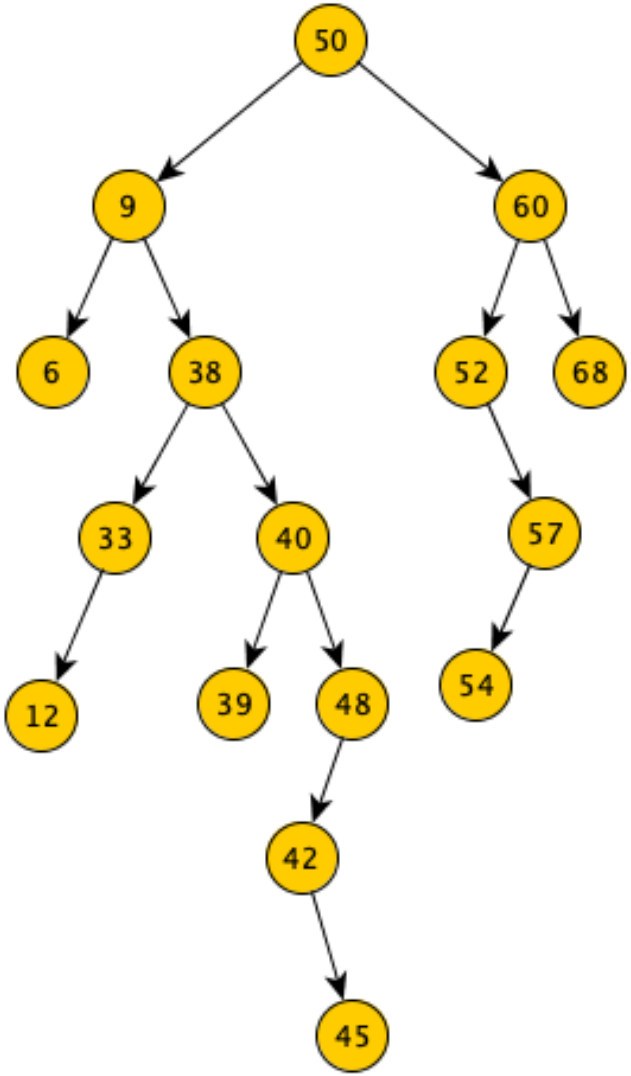
ABR

Un albero binario di ricerca (ABR) è un albero nel quale vengono mantenute le seguenti proprietà:

- ogni nodo contiene una chiave
- il valore della chiave contenuta in ogni nodo è maggiore della chiave contenuta in ciascun nodo del suo sottoalbero sinistro (se esiste)
- il valore della chiave contenuta in ogni nodo è minore della chiave contenuta in ciascun nodo del suo sottoalbero destro (se esiste)



ABR



Gli ABR sono strutture dati che supportano tutte le operazioni già definite in relazione agli insiemi dinamici più alcune altre:

- Operazioni di interrogazione che non alterano l'albero
- Operazioni di manipolazione che ne modificano la struttura

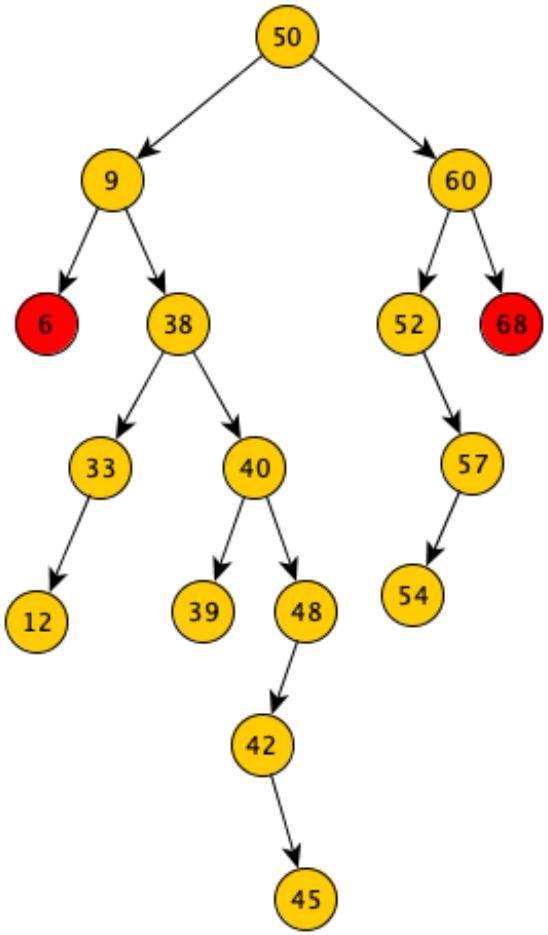
Operazioni di interrogazione:

- $Search(T, k)$: restituisce un puntatore all'elemento con chiave di valore k in T se questo è presente, $None$ altrimenti;
- $Minimum(T)$ / $Maximum(T)$: restituisce un puntatore all'elemento di minimo/massimo valore presente in T;
- $Predecessor(T, p)$ / $Successor(T, p)$: restituisce un puntatore all'elemento presente in T con il valore che precederebbe/seguirebbe il valore contenuto nel nodo puntato da p in una sequenza ordinata.

Operazioni di manipolazione:

- $Insert(T, k)$: inserisce un elemento di valore k in T;
- $Delete(T, p)$: elimina da T l'elemento puntato da p.

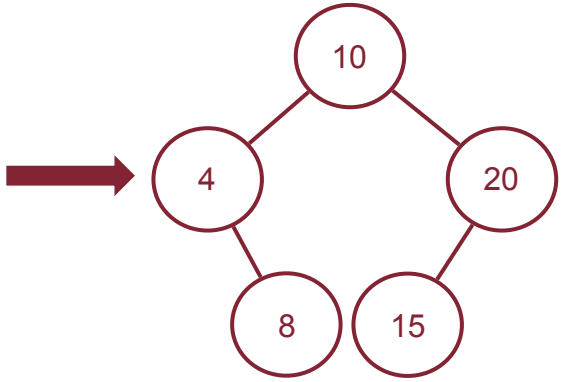
Un ABR può essere usato sia come dizionario che come coda di priorità: il minimo è sempre nel nodo più a sinistra, il massimo in quello più a destra



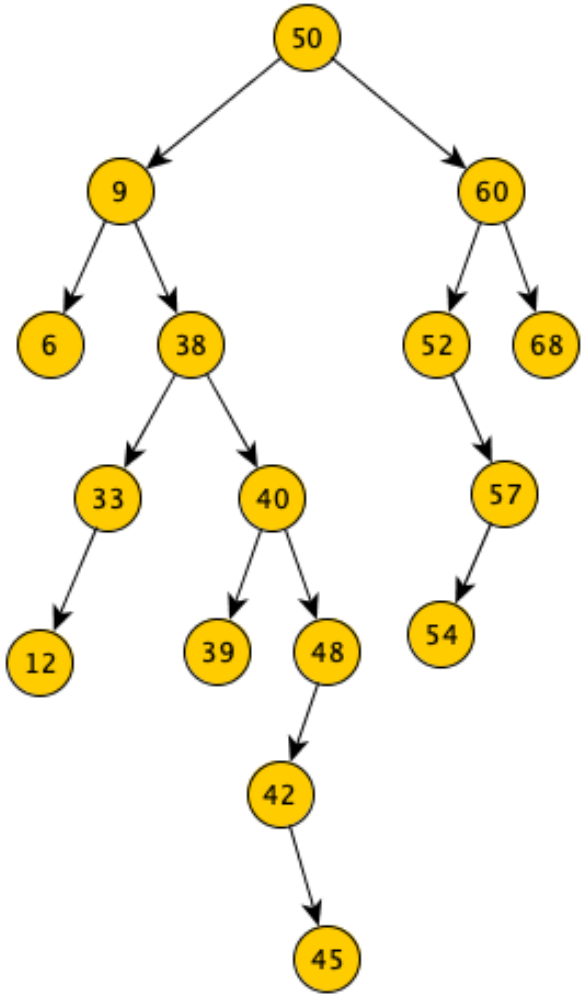
minimo:6
massimo: 68

Un ABR può essere usato sia come dizionario che come coda di priorità: il minimo è sempre nel nodo più a sinistra, il massimo in quello più a destra.

N.B. il nodo più a sinistra non necessariamente è una foglia: può anche essere il nodo più a sinistra che non ha un figlio sinistro; analoga considerazione per il nodo più a destra.



Per elencare tutte le chiavi in ordine crescente basta compiere una visita in-ordine.



stampa chiavi in inorder:

6 9 12 33 38 39 40 42 45 48 50 52 54 57 60 68

Per elencare tutte le chiavi in ordine crescente basta compiere una visita in-ordine.

Dunque un ABR può anche essere visto come una struttura dati su cui eseguire un algoritmo di ordinamento, costituito di due fasi:

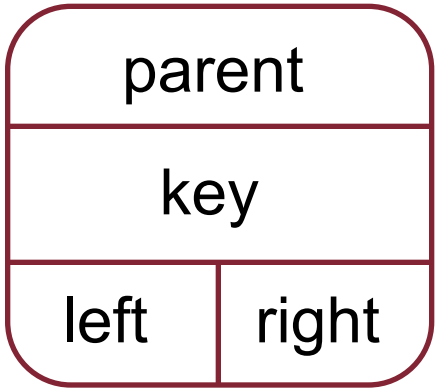
- inserimento di tutte le *n* chiavi da ordinare in un ABR, inizialmente vuoto;
- visita in-ordine dell'ABR appena costruito.

Il costo computazionale di tale algoritmo è:

$$T(costruzione\ ABR) + T(visita) = T(costruzione\ ABR) + \Theta(n)$$

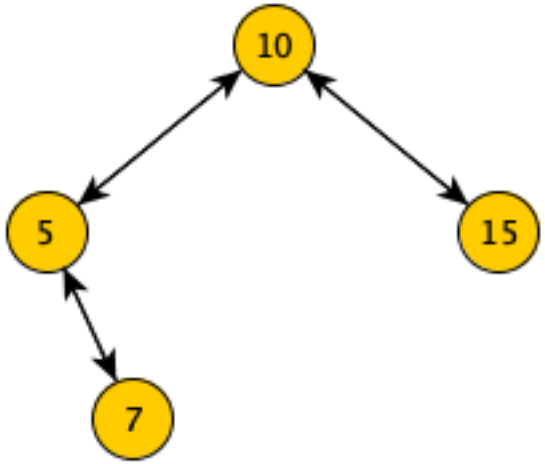
Più avanti determineremo il costo della costruzione di un ABR.

Nel seguito assumiamo che l'ABR sia memorizzato tramite record a puntatori e che ciascun nodo abbia oltre ai soliti puntatori ai figli destro e sinistro anche un puntatore al padre (questo risulterà utile in casi in cui da un nodo bisogna risalire ai suoi antenati).



```
class NodoABR:
    def __init__(self, key=None, left=None, right=None, parent=None):
        self.key = key
        self.left = left
        self.right=right
        self.parent=parent
```

```
>>> p=NodoABR(10)
>>> p.left=NodoABR(5)
>>> p.left.parent=p
>>> p.right=NodoABR(15)
>>> p.right.parent=p
>>> p.left.right=NodoABR(7)
>>> p.left.right.parent=p.left
```



Di seguito viene data la funzione $generaABR(n)$ che genera un albero di ricerca di n nodi con chiavi nell'intervallo $[1 \ldots 5 \cdot n]$.

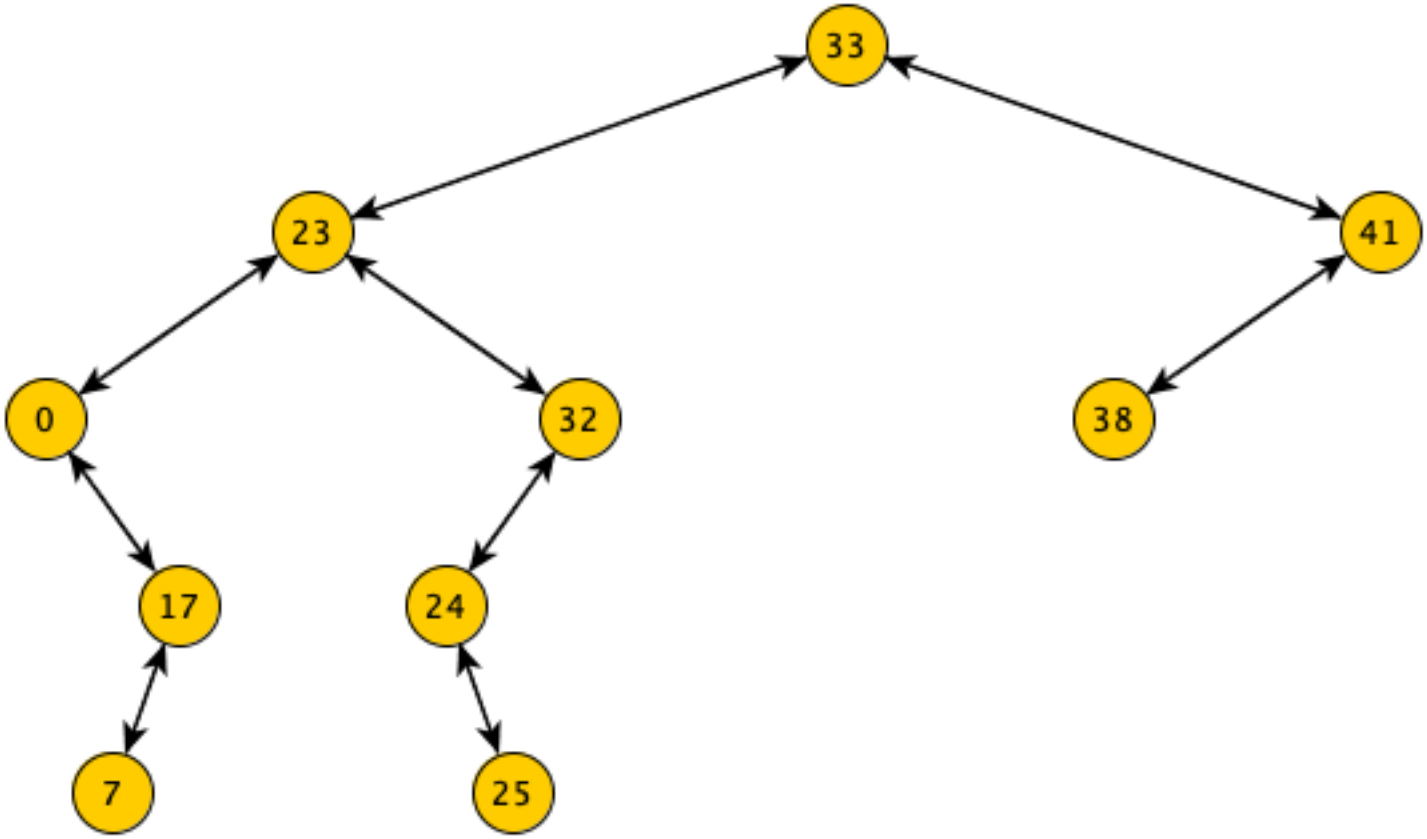
```
import random

def generaABR(n):
    if n==0: return None
    #genera la struttura di un albero binario  con n nodi
    p= generaABR1(n)
    #genera la lista delle chiavi per gli n nodi dell'albero
    lista=random.sample([x for x in range(5*n)],n)
    lista.sort(reverse=True)
    #inserisce le chiavi n chiavi negli n nodi dell'albero
    inserisciChiavi(p,lista)
    return p
```

```
def generaABR1(n):
    if n==0: return None
    p=NodoABR()
    n-=1
    if n>0:
        s=random.randint(0,n)
        p.left=generaABR1(s)
        if p.left:
            p.left.parent=p
        p.right=generaABR1(n-s)
        if p.right:
            p.right.parent=p
    return p
```

```
def inserisciChiavi( p,lista):
    if p:
        inserisciChiavi(p.left,lista)
        a=lista.pop()
        p.key=a
        inserisciChiavi(p.right,lista)
```

```
>>> r=generaABR(10)
>>> stampa(r)
33
| 23
| | 0
| | | -
| | | 17
| | | | 7
| | | | | -
| | | | | -
| | | | -
| | 32
| | | 24
| | | | -
| | | | 25
| | | | | -
| | | | | -
| | | -
| 41
| | 38
| | | -
| | | -
| | -
```



ABR – Ricerca

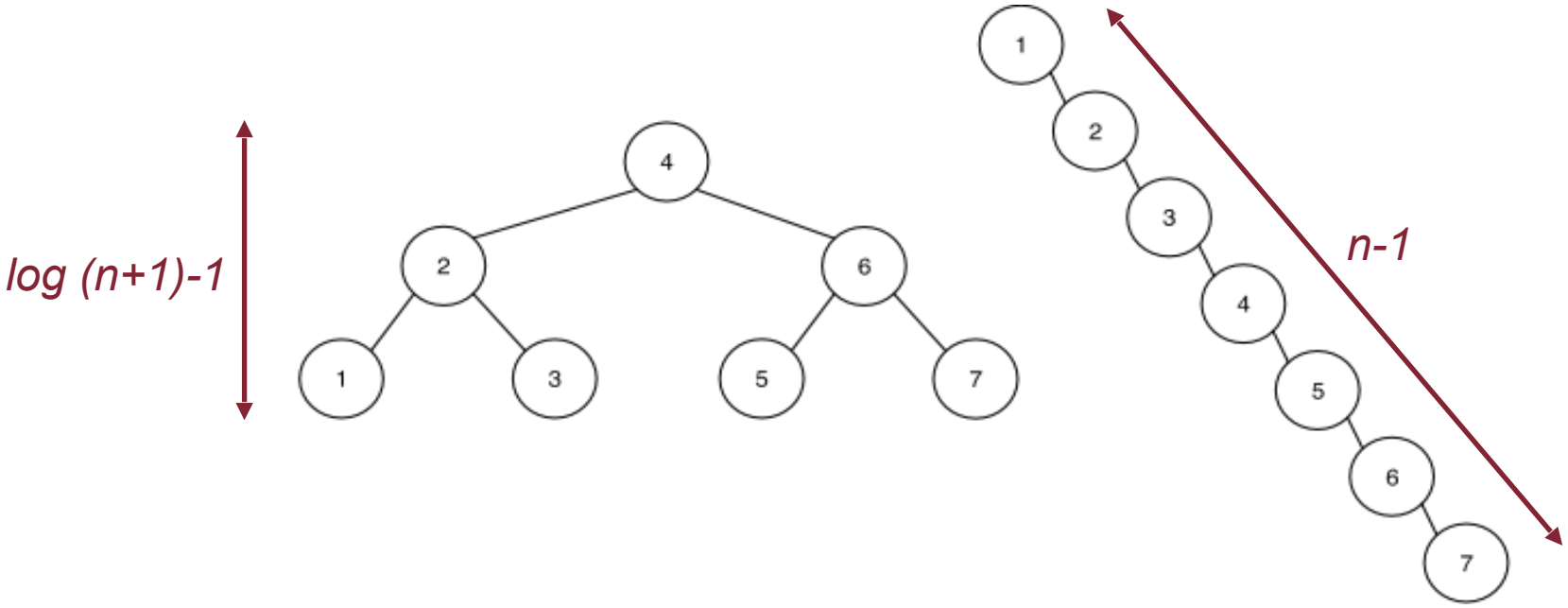
Concettualmente simile alla ricerca binaria: la funzione riceve il puntatore alla radice dell'albero e la chiave da ricercare ed esegue una discesa dalla radice che viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino.

```
def ABR_ricerca(p,k):
    if p == None or p.key == k:
        return p
    if k < p.key:
        return ABR_ricerca(p.left, k)
    else:
        return ABR_ricerca(p.right,k)
```

ABR – ricerca

Considerando che la ricerca su un ABR ricorda molto da vicino la ricerca binaria che ha costo computazionale $O(\log n)$, come mai non riusciamo a garantire un costo logaritmico anche per la ricerca su un ABR?

Problema: l'ABR non include fra le sue proprietà alcunché in relazione alla sua altezza.



ABR – ricerca

Quindi: se vogliamo garantire che il costo computazionale della ricerca su ABR sia limitata superiormente da un logaritmo, dovremo ricorrere a qualche tecnica che ci permetta di tenere sotto controllo la crescita dell'altezza: ***bilanciamento in altezza***.

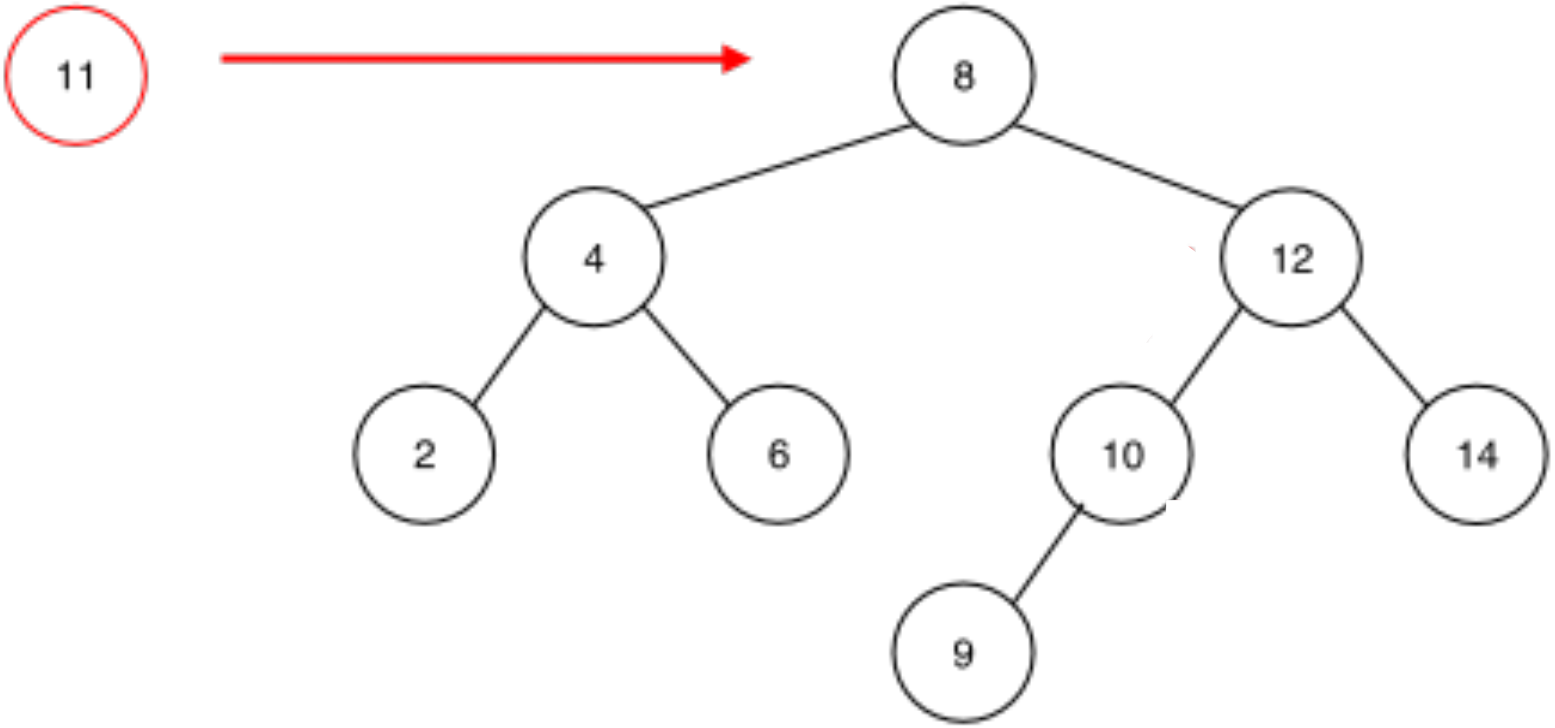
ABR – inserimento

- Si esegue una discesa che, come nel caso della ricerca, viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino.

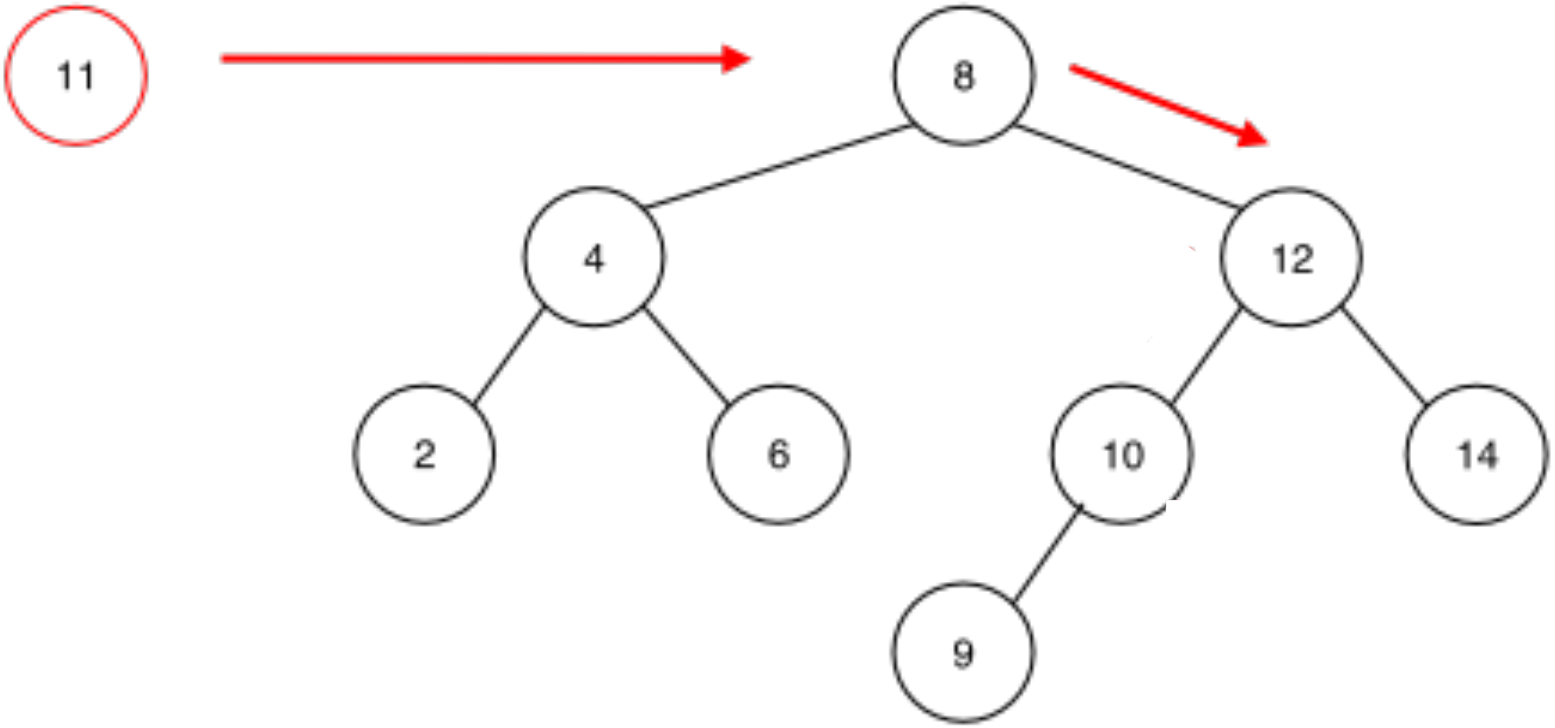
- Quando si arriva al punto di voler proseguire la discesa verso un puntatore vuoto (*None*) allora, in quella posizione, si aggiunge un nuovo nodo contenente il valore da inserire.

- Nota che: Il padre di tale nuovo nodo potrebbe essere una foglia (entrambi i suoi figli sono *None*), ma, più in generale, è un nodo a cui manca il figlio corrispondente alla direzione che si dovrebbe prendere per proseguire.

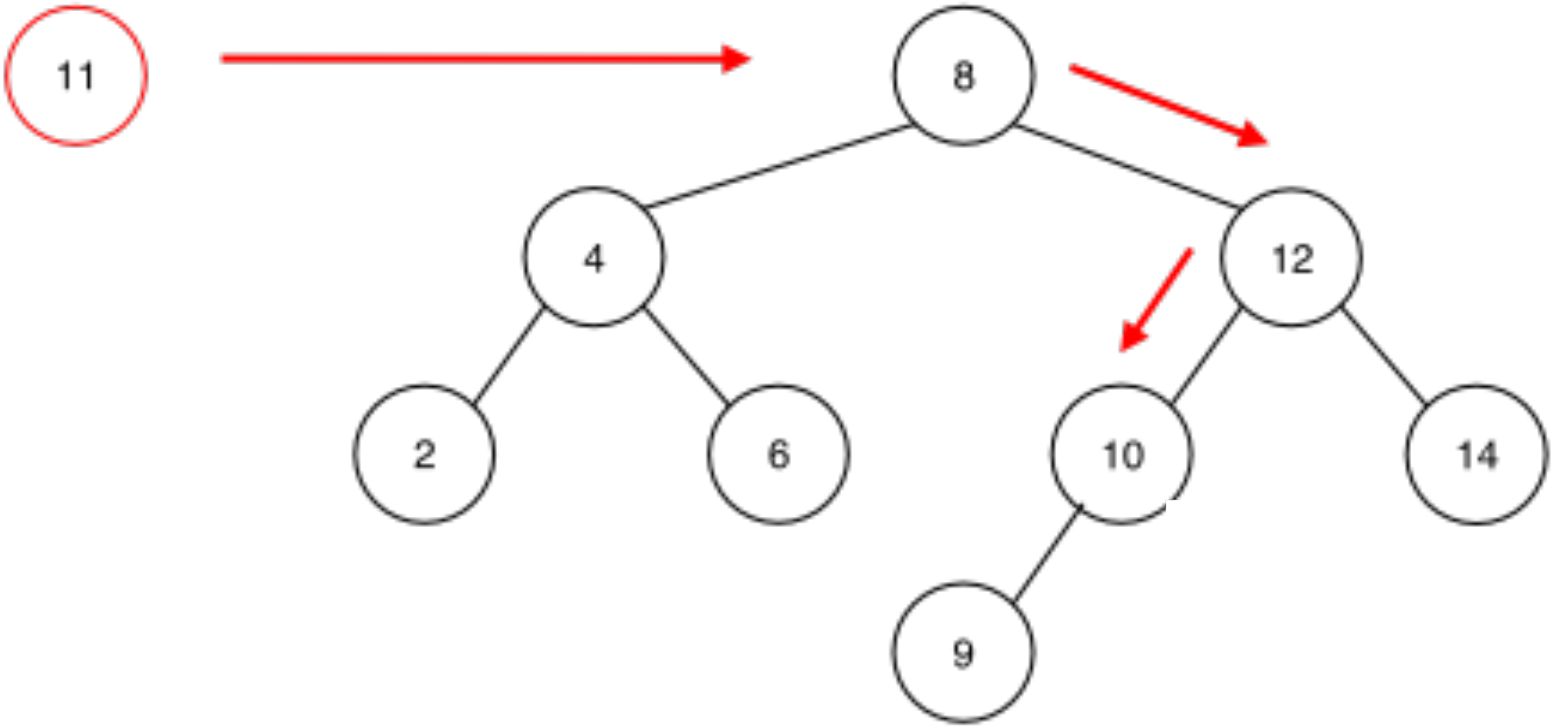
ABR – inserimento



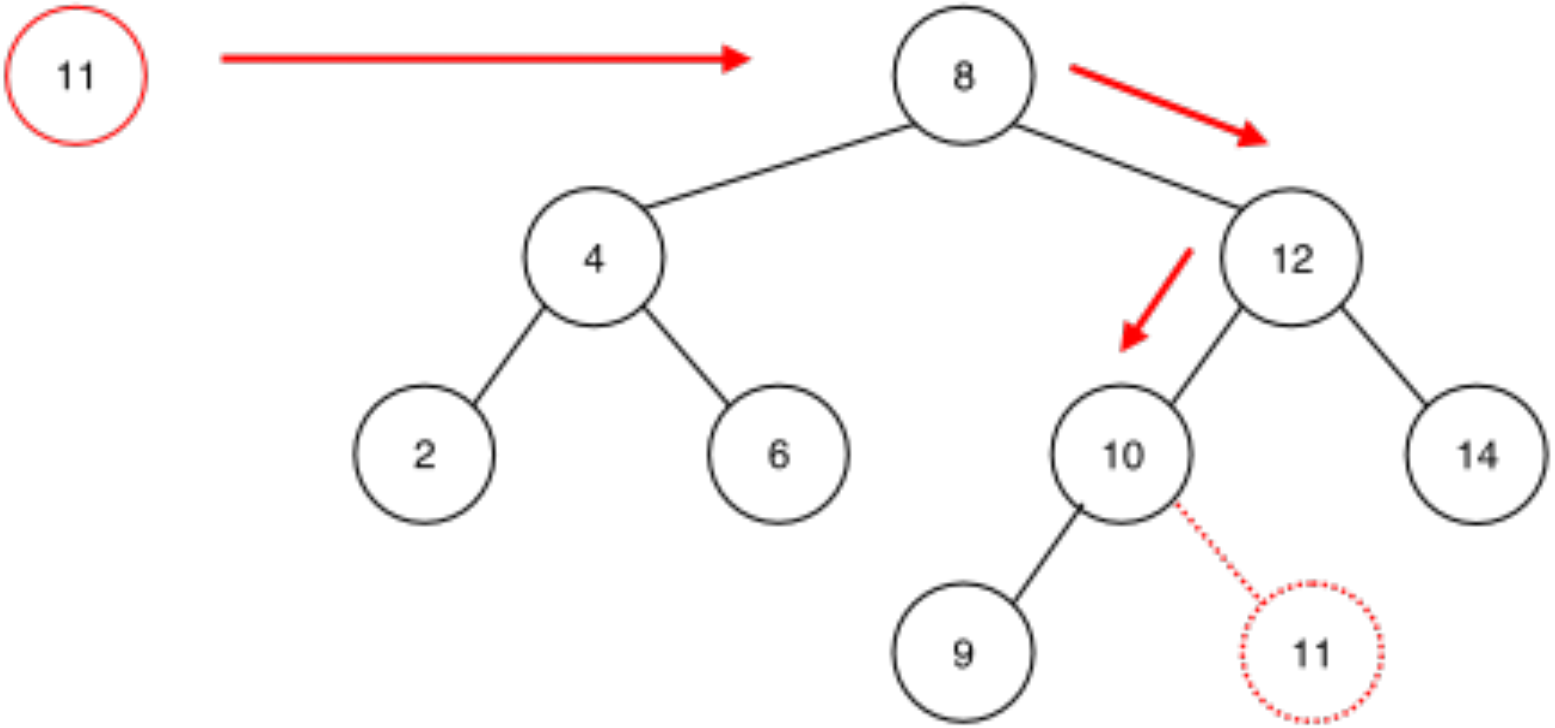
ABR – inserimento



ABR – inserimento



ABR – inserimento



```
def ABR_insert(p, k):
    y, x = None, p  # y punta sempre al padre di x
    z=NodoABR(k)
    while x != None:  # discesa alla prima posizione disponibile
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = y.right
    if y == None:  # se albero inizialmente vuoto
        p = z
    else:
        if z.key < y.key:
            y.left = z
        else:
            y.right = z
        z.parent=y
    return p    #p potrebbe essere cambiato
```

```
Def ABR_insert(p, z):
    y, x = None, p  # y punta sempre al padre di x
    z=NodoABR(k)
    while x != None: # discesa alla prima posizione disponibile
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = y.right
    if y == None: # se albero inizialmente vuoto
        p = z
    else:
        if z.key < y.key:
            y.left = z
        else
            y.right = z
         z.parent=y
    return p      #p potrebbe essere cambiato
```

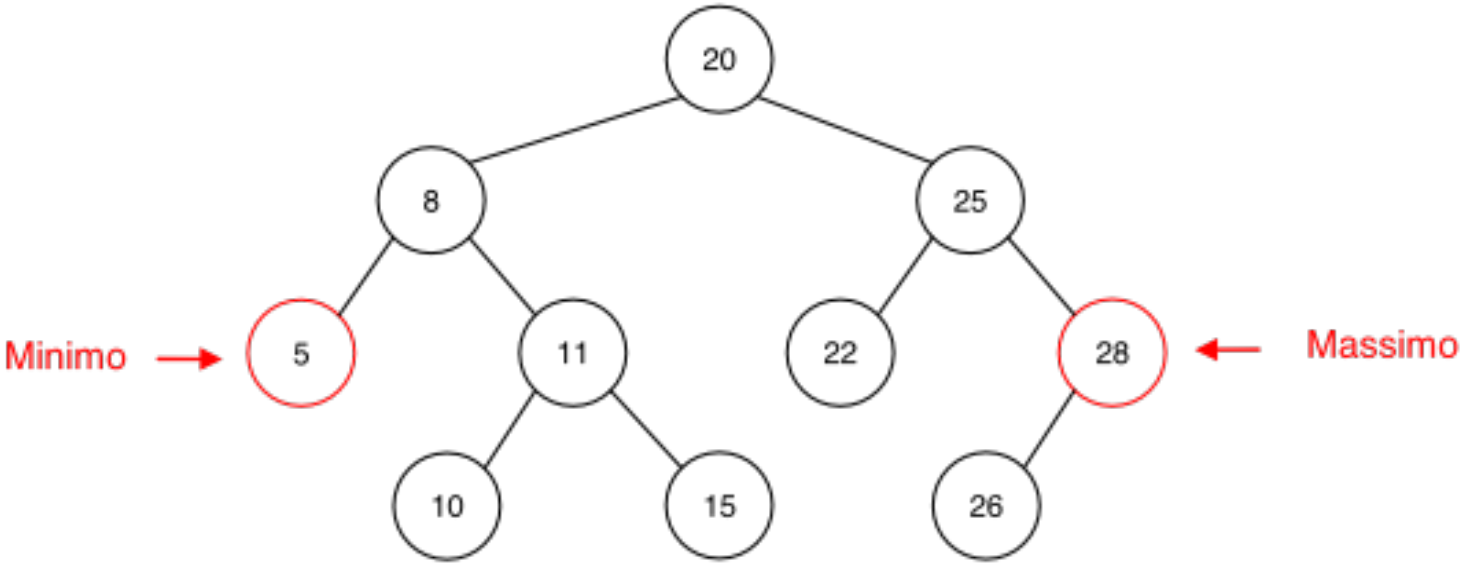
Il ciclo viene eseguito al massimo h volte quindi $T(n) = O(h)$

ABR – minimo e massimo

Minimo e massimo

Il minimo (massimo) si trova nel nodo più a sinistra (destra), quindi per trovarlo si scende sempre a sinistra (destra) a partire dalla radice. Ci si ferma quando si arriva a un nodo che non ha figlio sinistro (destro): quel nodo contiene il minimo (massimo).

$$T(n) = O(h)$$



ABR – predecessore

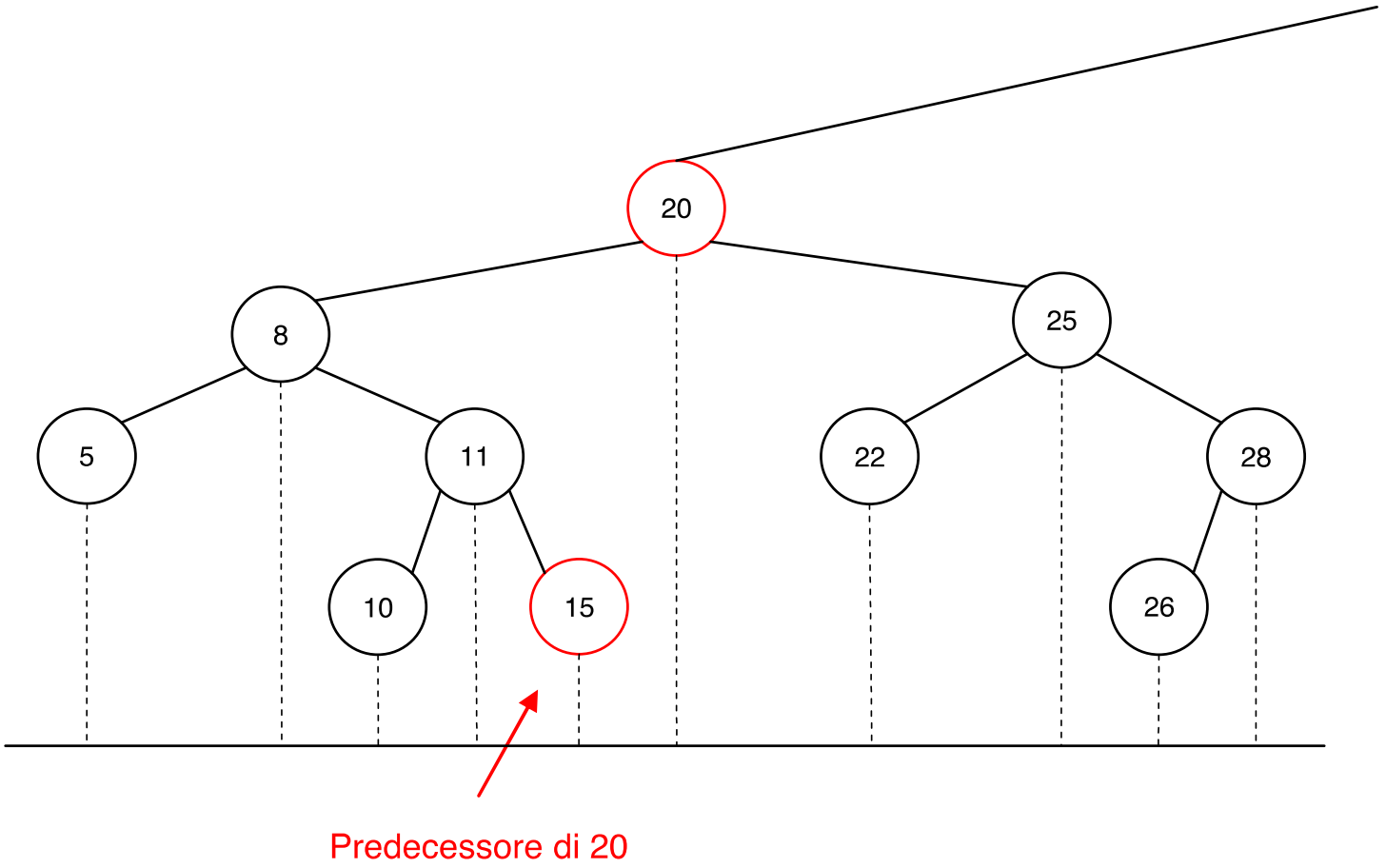
Ricordiamo che:

- per predecessore si intende il nodo dell'albero contenente la chiave che *precederebbe immediatamente k se le chiavi fossero ordinate*

- *Vedremo ora che il costo di questa funzione è limitato dall'altezza h dell'albero.*

ABR – predecessore

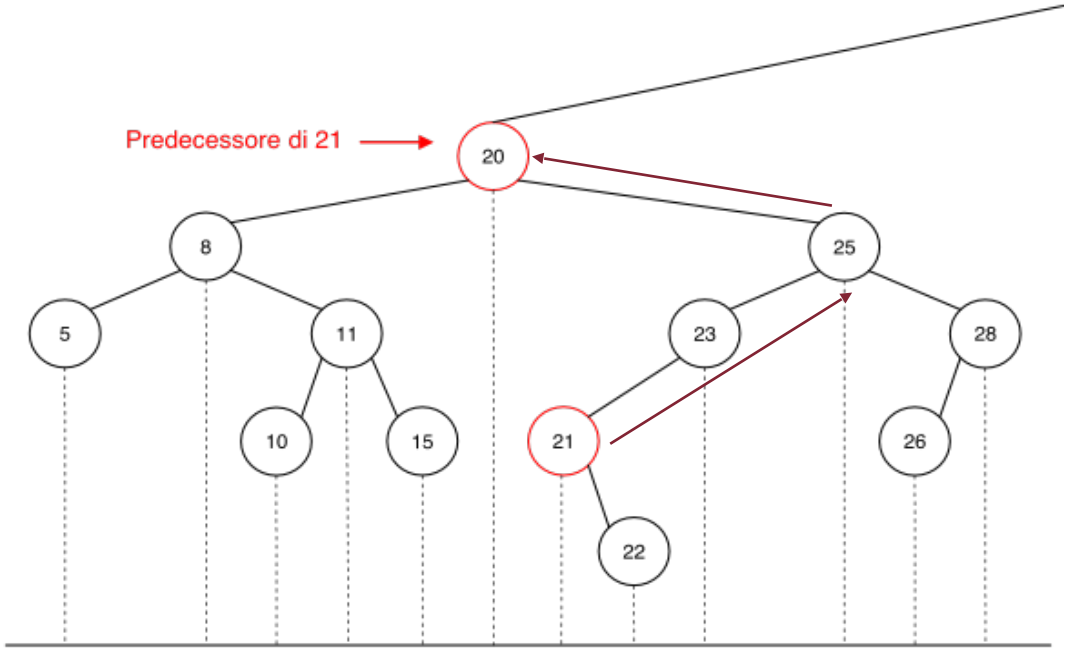
Caso 1: Se il nodo che contiene k **ha** il sottoalbero sinistro, il suo predecessore è il massimo di tale sottoalbero:



ABR – predecessore

Caso 2: se il nodo che contiene *k* **non** ha sottoalbero sinistro vuol dire che esso è il nodo più a sinistra di un certo sottoalbero, e quindi il minimo di tale sottoalbero. Per trovare il predecessore di *k* bisogna quindi:

- risalire alla radice di quel sottoalbero, il che significa ***salire a destra*** finché è possibile;
- una volta giunti nella radice del sottoalbero, si risale (con un singolo passo di ***salita a sinistra***) a suo padre che è il predecessore di *k*.



ABR – risalita verso destra o verso sinistra?

Come si fa a sapere se il passo di risalita dal nodo x al padre di x avviene salendo verso destra o verso sinistra?

Bisogna controllarlo esplicitamente con un test, ad ogni passo, ***prima*** di salire:

- `if x == x.parent.left` allora la risalita sarà verso destra
- `if x == x.parent.right` allora la risalita sarà verso sinistra

ABR – successore

Una situazione perfettamente simmetrica esiste per il problema di trovare il successore di un nodo.

- il successore è il nodo che contiene la chiave che *seguirebbe immediatamente k se le chiavi fossero ordinate*.

Anche questa operazione richiede una discesa lungo un singolo cammino a partire dalla radice oppure una singola risalita verso la radice.

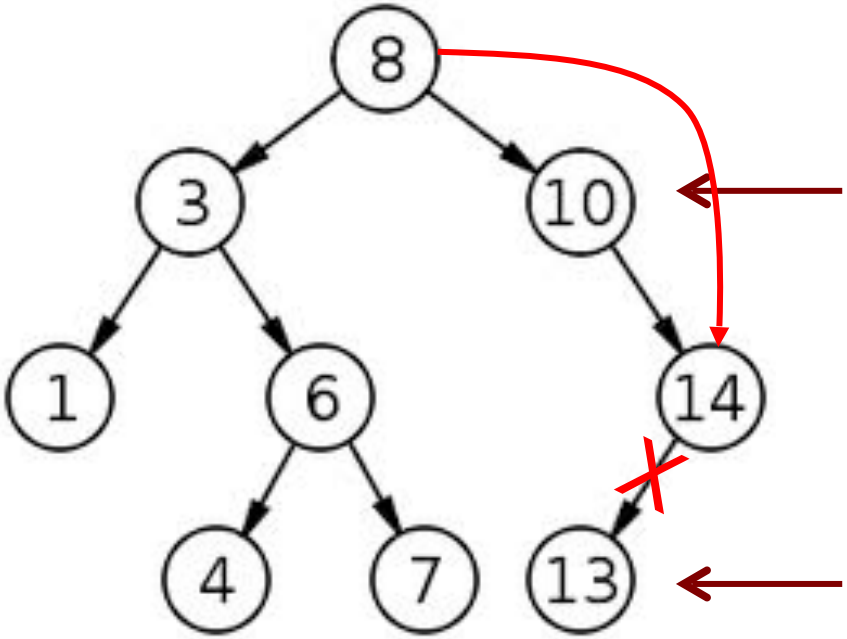
Il costo della funzione è dunque limitato superiormente dall'altezza dell'albero: $O(h)$.

ABR – cancellazione

Per eliminare un nodo in un ABR distinguiamo 3 casi in funzione del numero di figli che ha il nodo da cancellare:

- **Caso 1**: se il nodo è una foglia lo si elimina, ponendo a *None* l'opportuno campo nel nodo padre; Nell'esempio eliminazione di 13.

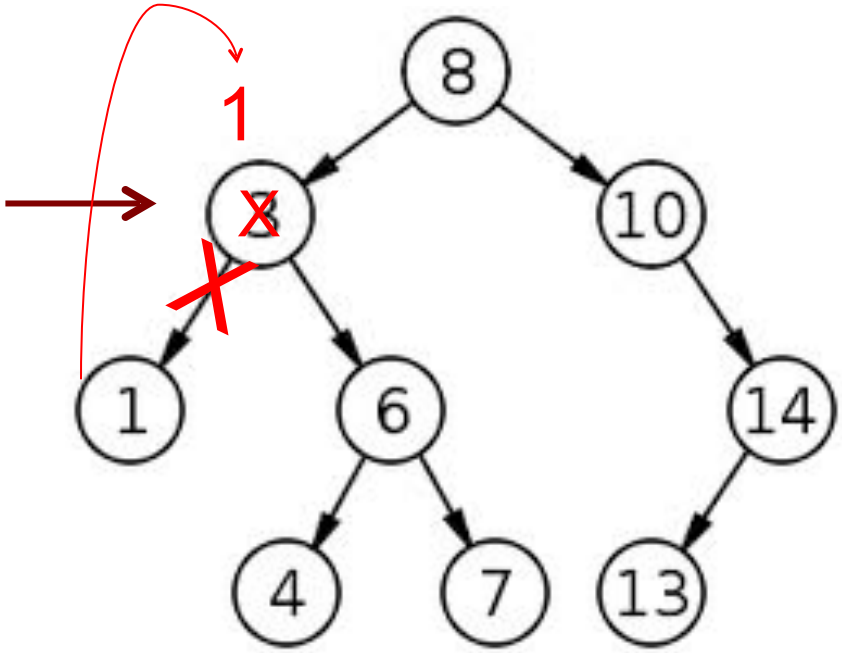
- **Caso 2**: se il nodo ha un solo figlio lo si "cortocircuita", cioè si collegano direttamente fra loro suo padre e il suo unico figlio, indipendentemente che sia destro o sinistro; Nell'esempio eliminazione di 10.



ABR – cancellazione

Caso 3: se il nodo ha entrambi i figli lo si sostituisce col predecessore (o col successore), che va quindi tolto (ossia eliminato) dalla sua posizione originale (operazione che ricade in uno dei due casi precedenti).

N.B. Per trovare il predecessore del nodo da cancellare siamo sicuramente nel *caso 1* dell'algoritmo per la ricerca del predecessore perché il nodo da cancellare ha due figli.



Esercizio svolto 1

Esercizio. Progettare un algoritmo che, dati due ABR T_1 e T_2, rispettivamente con n_1 ed n_2 nodi, ed altezza h_1 ed h_2, dia in output un ABR che contiene tutti gli $n_1 + n_2$ nodi. Fare le opportune osservazioni sul costo computazionale e sull'altezza dell'ABR risultante, come funzione di h_1 e h_2.

Soluzione. Inseriamo nell'albero con il maggior numero di nodi (senza perdere di generalità sia esso T_1) i nodi dell'altro albero uno ad uno…

Esercizio svolto 1

segue Esercizio.

… Sapendo che l'operazione di inserimento in un ABR ha un costo dell'ordine dell'altezza dell'albero si ha, nel caso peggiore, che il costo computazionale è:

$$
\begin{aligned}
T &= O(h_1) + O(h_1 + 1) + O(h_1 + 2) + \ldots + O(h_1 + n_2 - 1) \\
&= n_2 \cdot O(h_1) + O(1 + 2 + \ldots + n_2 - 1) \\
&= n_2 \cdot O(h_1) + O(n_2^2) \\
&= n_2 \cdot O(n_1) + O(n_2^2) \\
&= O(n_2 \cdot n_1) \quad \textit{essendo per ipotesi } n_1 > n_2
\end{aligned}
$$

Oss. questo procedimento è corretto perché un albero binario di ricerca ***non*** è necessariamente bilanciato, e quindi poco importa che l'altezza dell'albero risultante possa diventare $O(h_1 + n_2)$.

Esercizio svolto 2

Esercizio. Progettare un algoritmo che, dati due ABR T_1 e T_2, rispettivamente con n_1 ed n_2 nodi, ed altezza h_1 ed h_2, dia in output un ABR che contiene tutti gli $n_1 + n_2$ nodi, assumendo che tutti gli elementi in T_1 siano minori di quelli in T_2.

Soluzione. Oltre alla soluzione dell'esercizio precedente, proponiamo un altro approccio per tentare di sfruttare l'ipotesi…

Esercizio svolto 2

segue Esercizio.

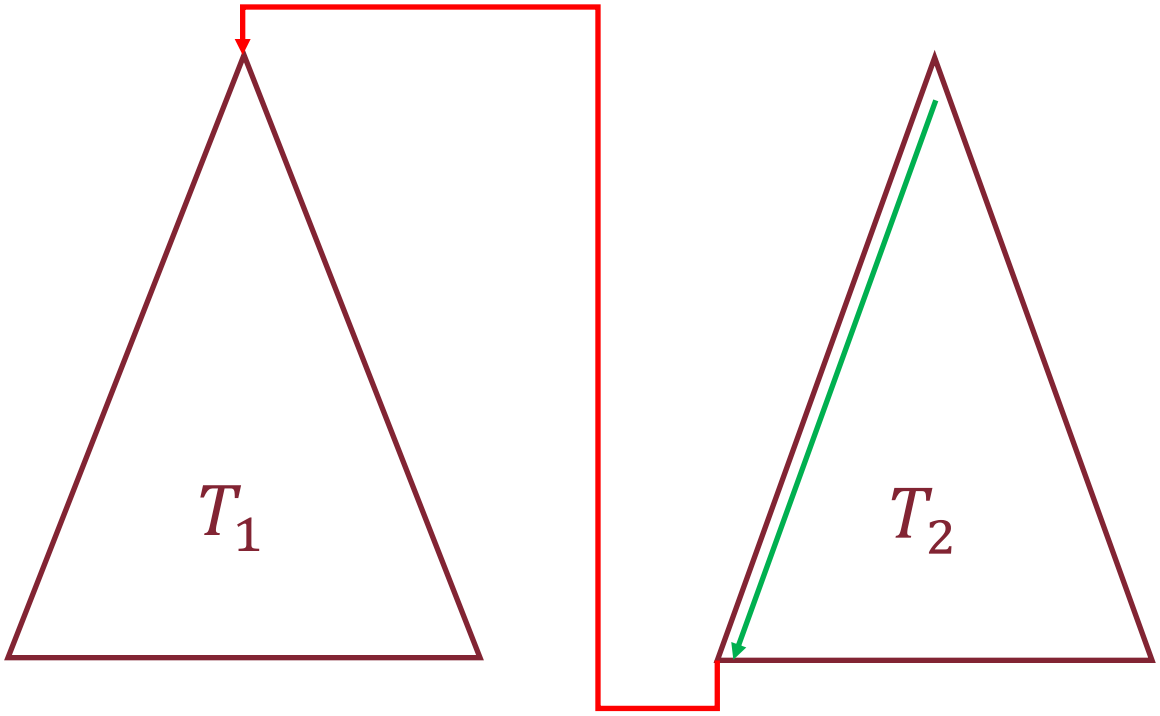
… “appendiamo” l’albero T_1 come figlio sinistro del minimo di T_2. Questo si può sempre fare facilmente perché il minimo di un ABR è il nodo più a sx che non possiede figlio sx, pertanto è sufficiente:

- settare un puntatore sulla radice di T_2,
- scendere verso il figlio sx finché esso esista
- giunti al nodo che non ha figlio sx (che è il minimo dell’albero), agganciargli a sx la radice di T_1.

Il costo è dominato dal costo della ricerca del minimo, cioè da $O(h_2)$.

Esercizio svolto 2

segue Esercizio.



Esercizio svolto 3

Esercizio. Scrivere lo pseudocodice dell'inserimento di una nuova chiave *z* in un ABR memorizzato mediante la notazione posizionale.

Assunzioni:

- Il vettore contiene *n* posizioni (indici da 0 a n-1);
- nel vettore si usa il simbolo '-' per denotare un nodo dell'albero mancante.

Esercizio svolto 3

segue Esercizio.

```
def ABR_insert(A, z):
    n=len(A)
    x = 0
    while x < n AND A[x] != '-': #discesa
        if z < A[x]:
            x = 2*x +1
        else:
            x = 2*x+2
    if x < n
        A[x] = z
```

Costo computazionale: $O(h)$.

Nota: $h = O(\log n)$

Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi

Per gli esercizi seguenti assumete che l'ABR sia memorizzato tramite puntatori e poi date anche la soluzione nel caso che sia memorizzato tramite notazione posizionale.

- Scrivere lo pseudocodice (sia iterativo che ricorsivo) della funzione che calcola il minimo in un ABR.
- Scrivere lo pseudocodice (sia iterativo che ricorsivo) della funzione che calcola il massimo in un ABR.
- Scrivere lo pseudocodice (sia iterativo che ricorsivo) della funzione che calcola il predecessore di un valore dato in un ABR.
- Scrivere lo pseudocodice (sia iterativo che ricorsivo) della funzione che calcola il successore di un valore dato in un ABR.
- Scrivere lo pseudocodice(sia iterativo che ricordavo) della funzione che cancella un nodo con un valore dato in un ABR