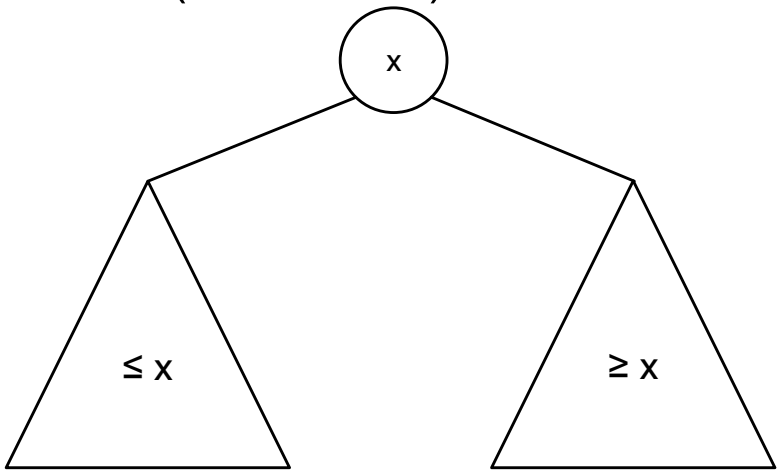
Corso di Introduzione agli algoritmi
Prof.ssa Tiziana Calamoneri

Dizionari: Alberi Binari di Ricerca

ABR (1)

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



ABR (2)

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

Operazioni di interrogazione:

- Search(T, k): restituisce un puntatore all'elemento con chiave di valore *k* in *T* se questo è presente, NULL 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*.

ABR (3)

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.

ABR (4)

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:

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

Più avanti determineremo il costo della costruzione di un ABR.

ABR - ricerca (1)

Ricerca

Concettualmente simile alla ricerca binaria: si esegue una discesa dalla radice che viene guidata dai valori memorizzati nei nodi che si incontrano lungo il cammino.

```
Funzione ABR_search_ricors(p: puntatore all'albero; k: chiave)
if ((p = NULL) or (key[p] = k))
    return p
if (k < key[p])
    return ABR_search (left[p], k)
else
    return ABR_search (right[p], k)
```

$T(h)=\Theta(1)$ (caso base)+

$\Theta(1)$+

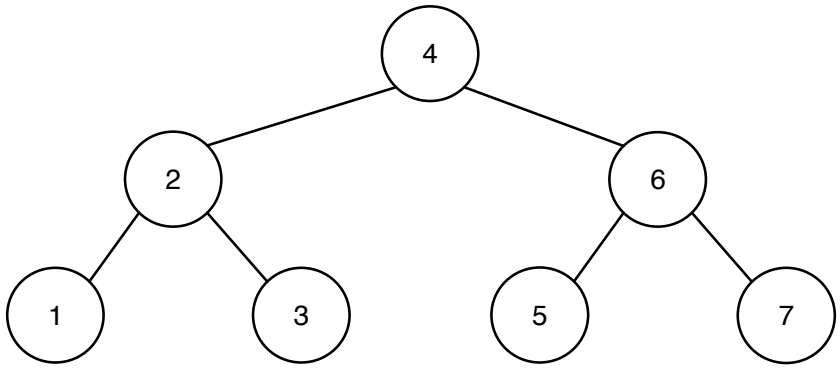
$T(h-1)=$

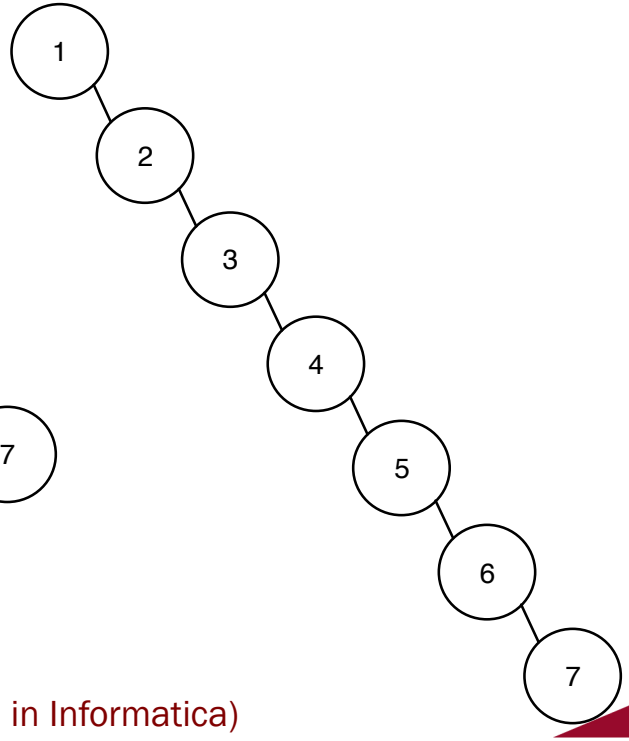
$\Theta(h)$ nel caso peggiore (ric. senza successo)

ABR - ricerca (2)

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

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





ABR - ricerca (3)

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

ABR – inserimento (1)

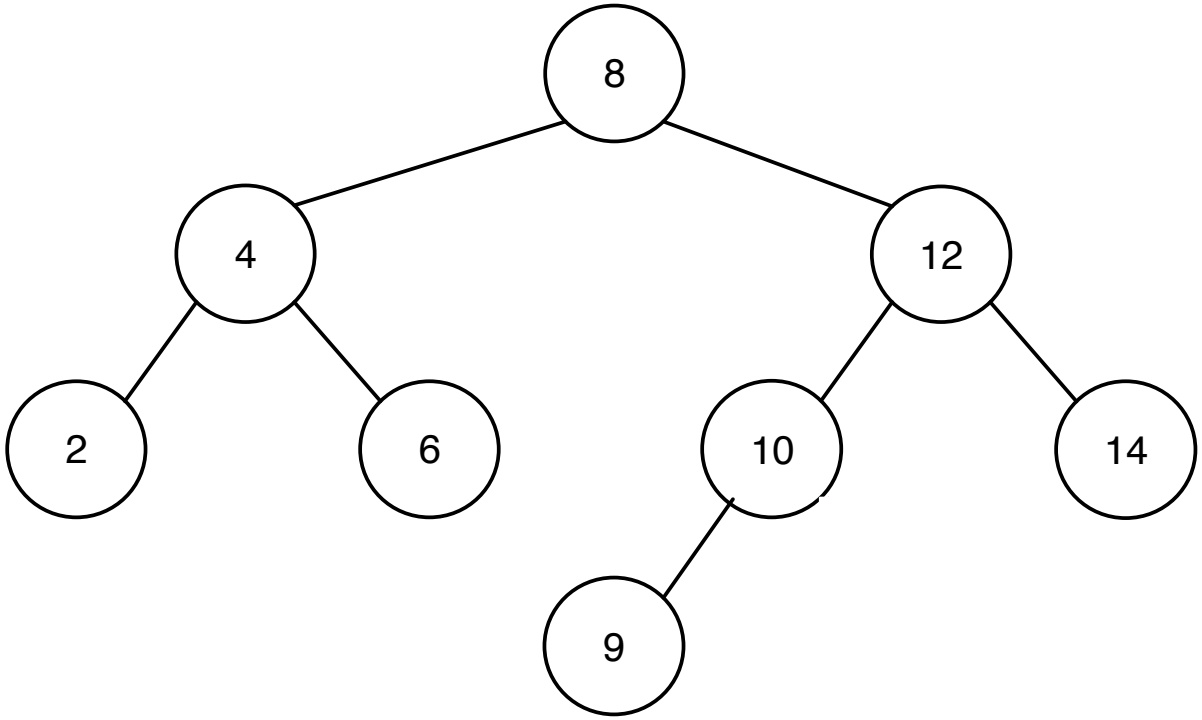
Inserimento

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 (NULL) allora, in quella posizione, si aggiunge un nuovo nodo contenente il valore da inserire.

Il padre di tale nuovo nodo potrebbe essere una foglia (entrambi i suoi figli sono NULL), ma, più in generale, è un nodo a cui manca il figlio corrispondente alla direzione presa.

ABR – inserimento (2)



ABR – inserimento (3)

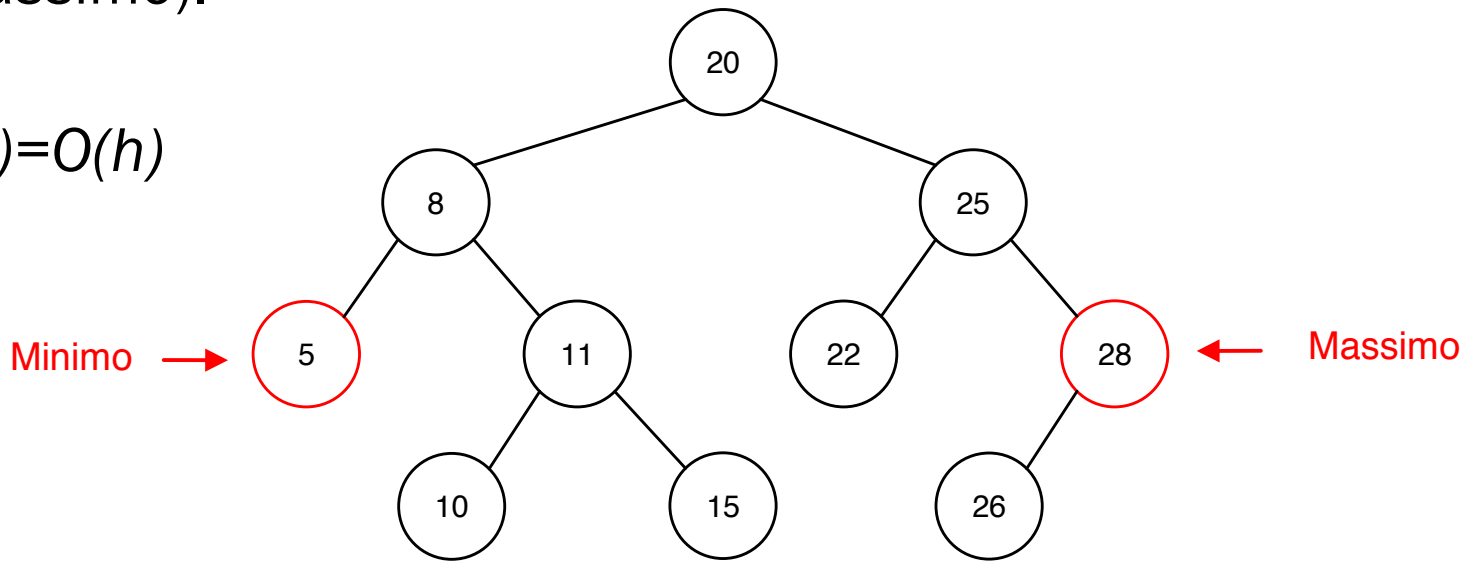
```
Funzione ABR_insert (p: punt. alla radice; z: punt al nodo da inserire)
    y ← NULL; x ←  p                    // y punta sempre al padre di x
    while (x ≠ NULL)                    // discesa alla prima pos. disponibile
          y ←  x
          if key[z] < key[x]
                    x ←  left[x]
          else
                    x ← right[x]
    parent[z] ←  y                      // collegam. padre - nodo da inserire
    if (y = NULL)                       // se albero inizialmente vuoto
          p ←  z
    else
          if (key[z] < key[y])
                    left[y] ←  z
          else
                    right[y] ←  z
    return p                            //p potrebbe essere cambiato
```

ciclo eseguito al max *h* volte quindi *T(h)=O(h)*

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(h)=O(h)

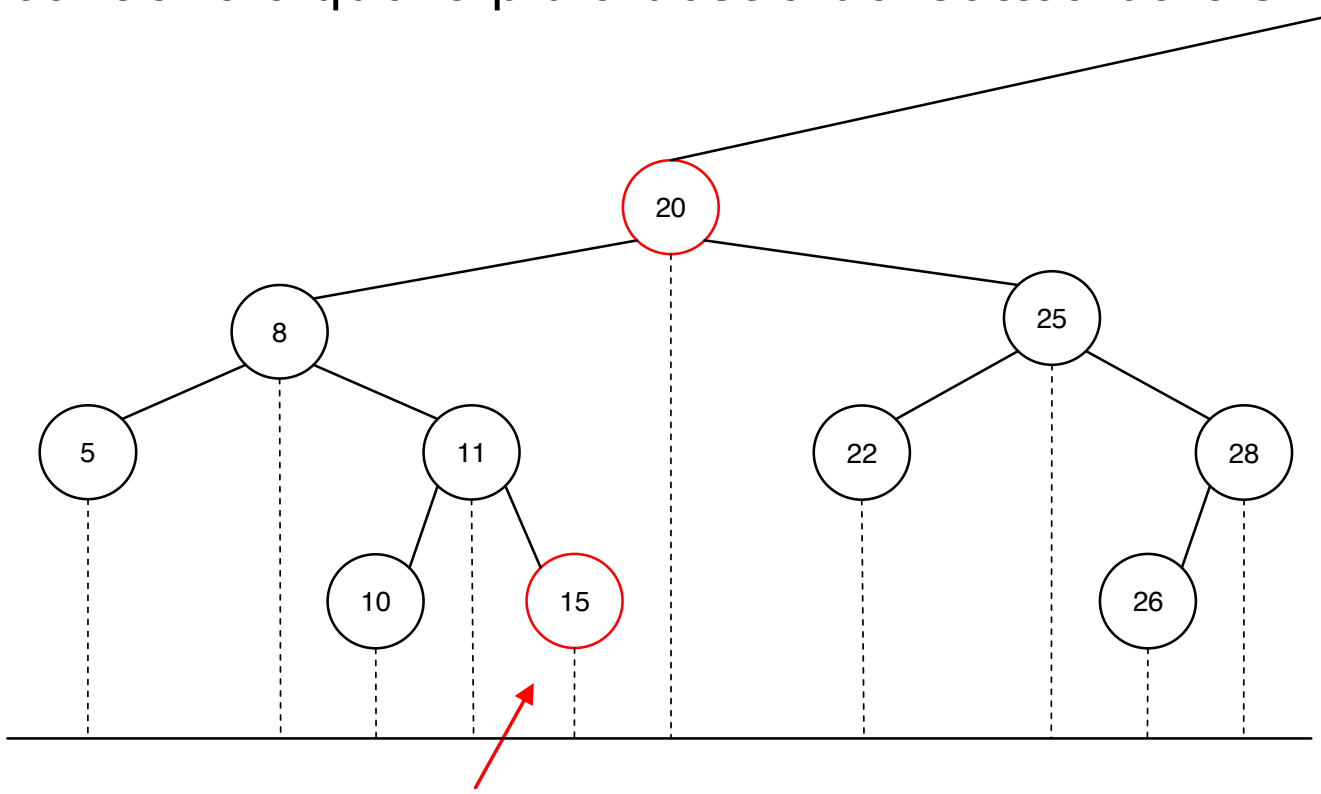


predecessore e successore:

Ricordiamo che per predecessore si intende il nodo dell'albero contenente la chiave che precederebbe immediatamente *k* se le chiavi fossero ordinate; il successore è il nodo che contiene la chiave che seguirebbe immediatamente *k* se le chiavi fossero ordinate.

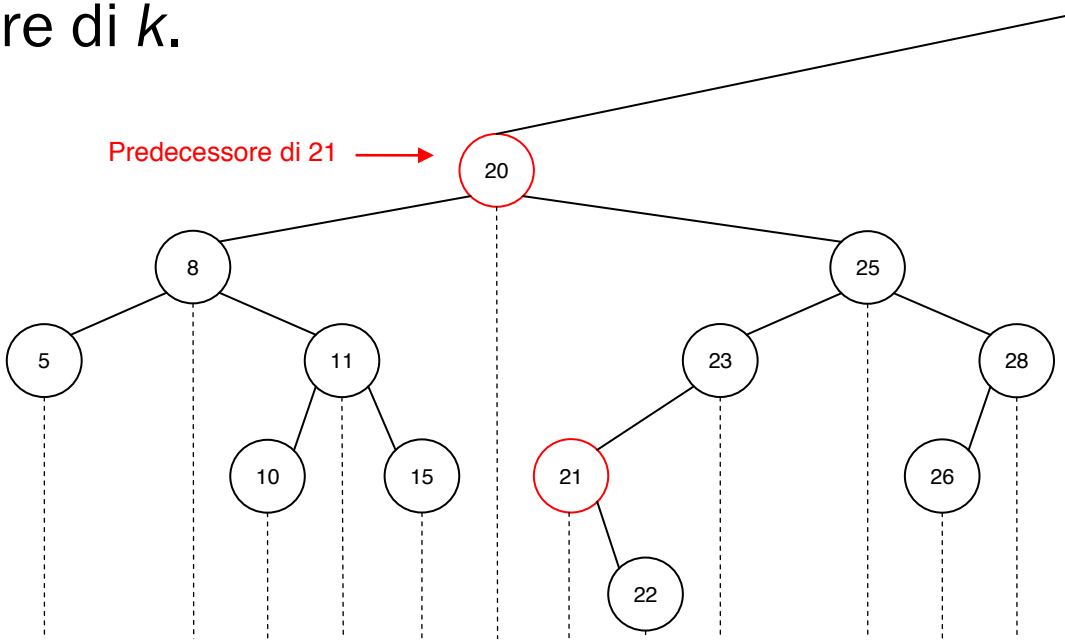
Concentriamoci sulla ricerca del predecessore.

Caso 1: se la chiave *k* è contenuta in un nodo che ha un sottoalbero sinistro, allora il predecessore di *k* è il massimo delle chiavi contenute nel sottoalbero sinistro, e quindi il nodo che la contiene è quello più a destra del sottoalbero sinistro.



ABR – predecessore e successore (3)

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 finchè 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*.



Una situazione perfettamente simmetrica esiste per il problema di trovare il successore di un nodo. Entrambe tali operazioni richiedono una discesa lungo un singolo cammino a partire dalla radice oppure una singola risalita verso la radice.

Per entrambe le funzioni il costo è limitato superiormente dall'altezza dell'albero: $O(h)$.

ABR – cancellazione (1)

cancellazione:

Problema: Se la chiave da eliminare è contenuta in un nodo che ha entrambi i figli è necessario riaggiustare l'albero dopo l'eliminazione per evitare che si disconnetta, il che produrrebbe di conseguenza due ABR separati.

Riaggiustare l'albero significa trovare un nodo da collocare al posto del nodo che va eliminato, così da mantenere l'albero connesso e da garantire il mantenimento della proprietà fondamentale degli ABR.

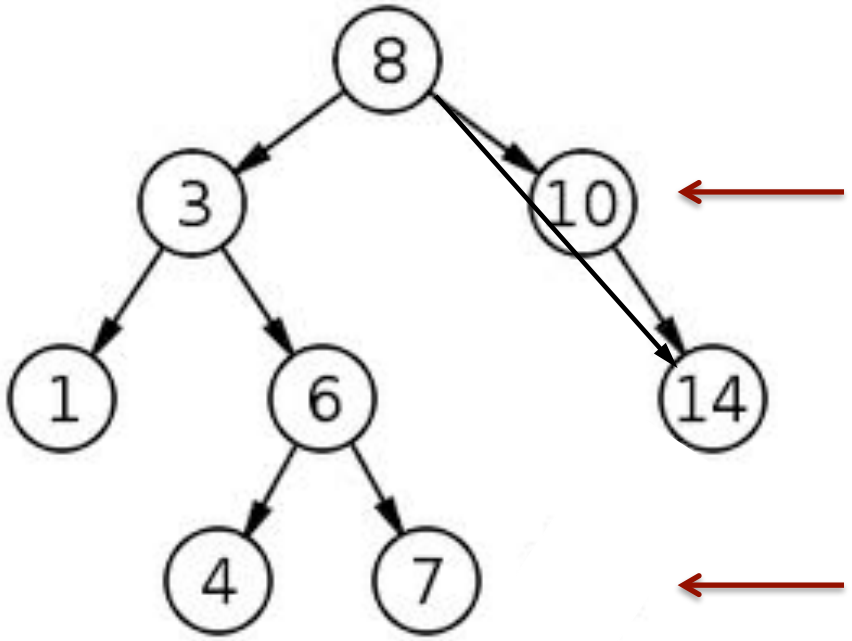
Tale nodo può quindi essere solamente il predecessore o il successore del nodo da eliminare.

ABR – cancellazione (2)

Per eliminare un nodo in un ABR:

se il nodo è una foglia lo si elimina, ponendo a *NULL* l'opportuno campo nel nodo padre;

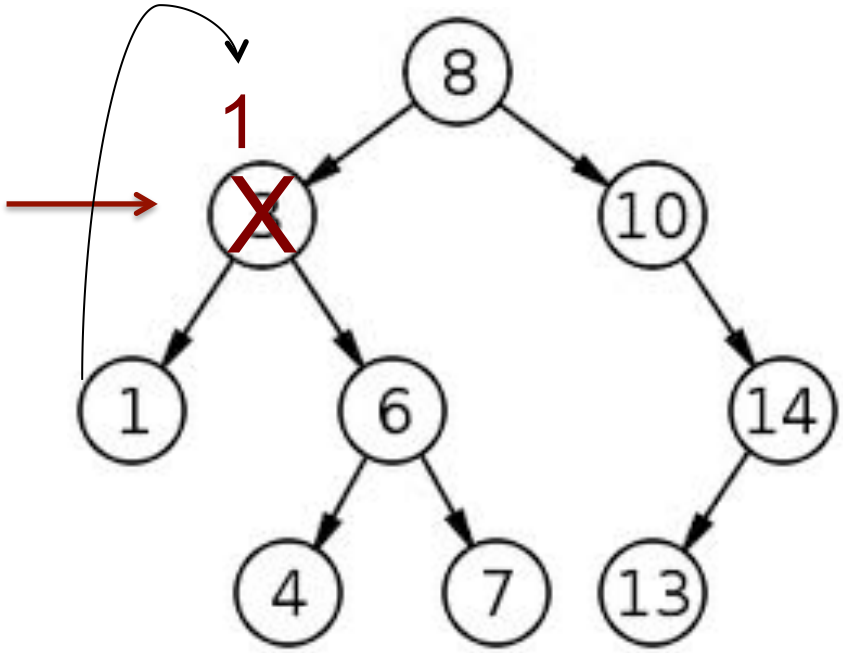
se il nodo ha un solo figlio lo si "cortocircuita", cioè si collegano direttamente fra loro suo padre e il suo unico figlio, indipendentemente che sia destro o sinistro;



ABR – cancellazione (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. Ci si riduce ad un caso precedente.

N.B. Per trovare il predecessore (o il successore) del nodo da cancellare: sicuramente caso 1 dell'algoritmo perché il nodo da cancellare ha due figli.



Esercizi

- 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.
- Progettare un algoritmo che, dati due ABR *T1* e *T2*, rispettivamente con *n1* ed *n2* nodi, ed altezza *h1* ed *h2*, dia in output un ABR che contiene tutti gli *n1*+*n2* nodi. Fare le opportune osservazioni sul costo computazionale e sull'altezza dell'ABR risultante, come funzione di *h1* e *h2*.