

Algoritmi e Strutture Dati

Capitolo 6

Il problema del dizionario

Il tipo dato Dizionario

tipo Dizionario:

dati: un insieme S di n elementi di tipo $elem$ a cui sono associate chiavi **distinte** di tipo $chiave$ prese da un universo totalmente ordinato.

operazioni:

$insert(elem\ e, chiave\ k)$
aggiunge a S una nuova coppia (e, k)

$delete(elem\ e)$ ←
cancella da S l'elemento e

Suppongo sempre che mi venga dato un riferimento diretto all'elemento da cancellare

[] → $elem$
se la chiave k è presente in S restituisce l'elemento e ad essa associato, e null altrimenti

Applicazioni: gestione archivi di dati

Obiettivo

Fornire un'implementazione di un dizionario di n elementi che consenta di fare **tutte** le operazioni descritte in tempo $O(\log n)$.

Quattro implementazioni elementari

1. Array non ordinato
2. Array ordinato
3. Lista non ordinata
4. Lista ordinata

NOTA BENE: Come per le **code di priorità**, si noti che il **dizionario** è un tipo di dati **dinamico** (cioè di dimensione variabile), in quanto soggetto ad **inserimenti** e **cancellazioni**. L'uso degli array va quindi inteso pensando alla loro versione **dinamica**, che implica **riallocazioni/deallocazioni** di memoria che **raddoppiano/dimezzano** lo spazio utilizzato. Con tale accorgimento, i costi di riallocazione/deallocazione sono assorbiti (asintoticamente) dai costi per le **insert** e le **delete**

Array non ordinato

Tengo traccia in una variabile di appoggio del numero n di elementi effettivamente presenti nel dizionario (**dimensione logica** dell'array), e gestisco la **dimensione fisica** dell'array mediante allocazione dinamica

- **Insert**: $O(1)$ (inserisco in fondo all'array)
- **Delete**: $O(1)$ (poiché mi viene fornito il riferimento diretto all'elemento da cancellare, lo posso cancellare in $O(1)$ sovracopiando l'ultimo elemento)
- **Search**: $O(n)$ (devo scorrere l'array); nel caso migliore costa $O(1)$, ovviamente

Array ordinato

Gestione dinamica come sopra; l'array viene inoltre tenuto **ordinato** in ordine **crescente** o **decrescente** indifferentemente

- **Insert**: $O(n)$ (trovo in $O(n)$ mediante scorrimento **da destra verso sinistra** la giusta posizione, e poi faccio $O(n)$ spostamenti verso destra); nel caso migliore costa $O(1)$, grazie all'accorgimento della scansione da destra verso sinistra, come facevamo in **InsertionSort2**;
- **Delete**: $O(n)$ (devo fare $O(n)$ spostamenti verso sinistra); nel caso migliore costa $O(1)$, ovviamente
- **Search**: $O(\log n)$ (**ricerca binaria**); nel caso migliore costa $O(1)$, ovviamente

Lista non ordinata

La considero **bidirezionale**, e mantengo un puntatore alla **testa** ed uno alla **coda**



- **Insert**: $O(1)$ (inserisco in coda o in testa)
- **Delete**: $O(1)$ (poiché mi viene fornito il riferimento diretto all'elemento da cancellare, lo posso cancellare in $O(1)$ agendo sui puntatori)
- **Search**: $O(n)$ (devo scorrere la lista); nel caso migliore costa $O(1)$, ovviamente

Lista ordinata

La considero **bidirezionale** e ordinata in ordine **crescente** o **decrescente** indifferentemente



- **Insert**: $O(n)$ (devo prima scorrere la lista in $O(n)$ per trovare la giusta posizione, poi inserisco in $O(1)$ agendo sui puntatori); nel caso migliore costa $O(1)$, ovviamente
- **Delete**: $O(1)$ (poiché mi viene fornito il riferimento diretto all'elemento da cancellare, lo posso cancellare in $O(1)$ agendo sui puntatori)
- **Search**: $O(n)$ (devo scorrere la lista); nel caso migliore costa $O(1)$, ovviamente

Implementazioni elementari

	Insert	Delete	Search
Array non ord.	$O(1)$	$O(1)$	$O(n)$
Array ordinato	$O(n)$	$O(n)$	$O(\log n)$
Lista non ordinata	$O(1)$	$O(1)$	$O(n)$
Lista ordinata	$O(n)$	$O(1)$	$O(n)$

☹️ Ognuno dei 4 metodi elementari ha almeno un'operazione che costa $O(n)$. Voglio fare meglio...

Lower bound $\Omega(\log n)$ per la ricerca

- Ricordiamo che per il problema della ricerca di un elemento in un insieme non ordinato si applica il lower bound banale di $\Omega(n)$. Tuttavia, ad esempio, nel caso di insiemi ordinati la ricerca binaria costa $O(\log n)$. Possiamo migliorarla?
- Consideriamo l'albero di decisione di un qualsiasi algoritmo che risolve il problema della ricerca in un insieme di n elementi tramite confronti
- L'albero deve contenere almeno $n+1$ foglie (ogni foglia specifica una tra le n posizioni dove si può trovare l'elemento, più la foglia "non trovato")
- Un albero binario con k foglie in cui ogni nodo interno ha al più due figli, ha altezza

$$h(k) \geq \log k \text{ (vedi lower bound sull'ordinamento)}$$

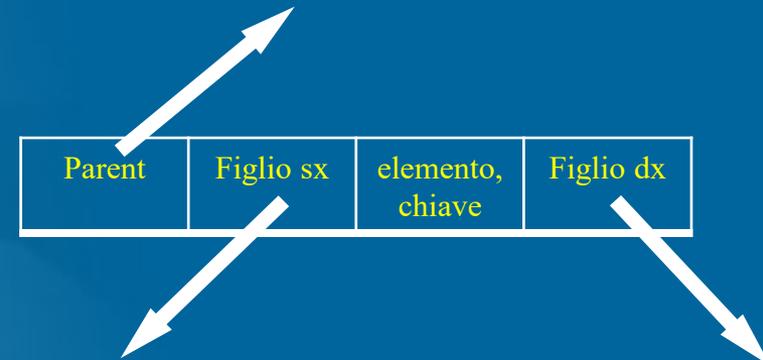
⇒ L'altezza $h(n+1)$ dell'albero di decisione è $\Omega(\log n)$

☺ La ricerca binaria quindi è ottimale e non può essere ulteriormente migliorata

⇒ Proviamo a definire una struttura a puntatori su cui applicare una qualche forma di ricerca binaria!

Alberi Binari di Ricerca (ABR, o BST = Binary Search Tree)

Definizione di ABR

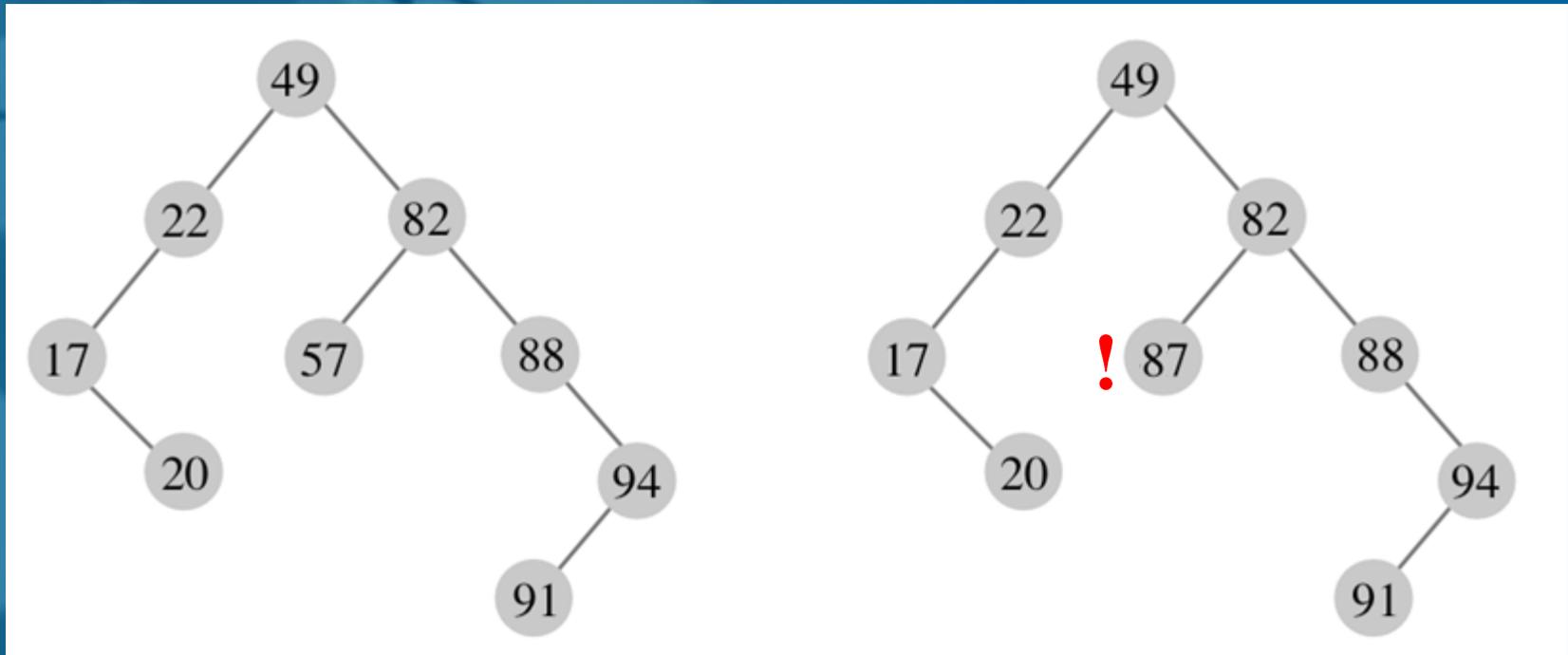


Implementazione del dizionario mediante un **albero binario** in cui ogni nodo v contiene una coppia **(elem(v), chiave(v))** del dizionario, nonché un **puntatore al padre parent(v)**, un **puntatore al figlio sinistro sin(v)** e un **puntatore al figlio destro des(v)**, e soddisfa le seguenti proprietà:

- le chiavi nel **sottoalbero sinistro** di v sono $<$ **chiave(v)**
- le chiavi nel **sottoalbero destro** di v sono $>$ **chiave(v)**

⇒ Vedremo che tali proprietà inducono un **ordinamento totale** sulle chiavi del dizionario!

Esempi



Albero binario
di ricerca

Albero binario
non di ricerca

Visita simmetrica di un ABR

- **Visita in ordine simmetrico** – dato un nodo v , elenco prima il sotto-albero sinistro di v (ricorsivamente in ordine simmetrico), poi il nodo v , poi il sotto-albero destro di v (ricorsivamente in ordine simmetrico)

```

algoritmo Inorder-tree-walk(node  $v$ )
if ( $v \neq \text{null}$ )
    then Inorder-tree-walk( $\text{sin}(v)$ )
        stampa chiave( $v$ )
        Inorder-tree-walk( $\text{des}(v)$ )
  
```

- **Inorder-tree-walk(radice dell'ABR)** visita tutti i nodi dell'ABR
- **Analisi complessità:** la complessità della procedura considerata è $T(n) = \Theta(n)$. Infatti:

$$T(n) = T(n') + T(n'') + O(1) \quad \text{con } n' + n'' = n - 1$$

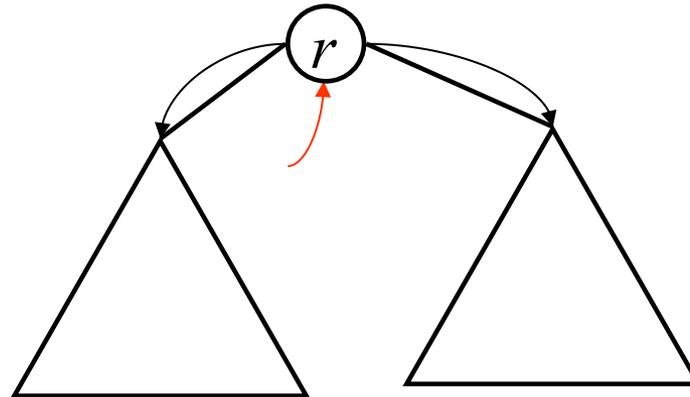
Proprietà della visita simmetrica di un ABR

Inorder-tree-walk(radice dell'ABR) visita i nodi dell'ABR in ordine crescente rispetto alla chiave!

Verifica: Indichiamo con h l'altezza dell'albero. Per induzione sull'altezza dell'ABR:

Base ($h=0$): banale (l'ABR consiste di un unico nodo);

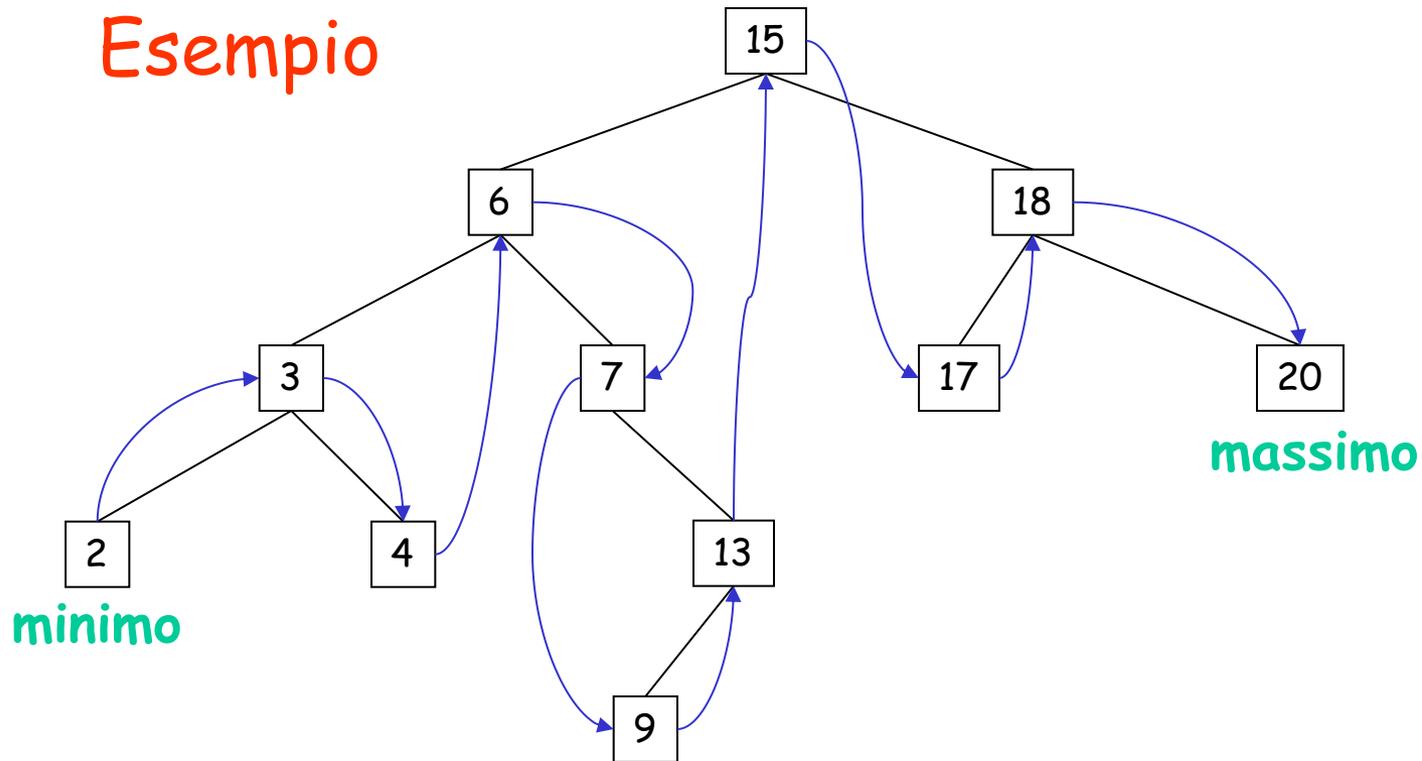
Passo induttivo (h generico): ipotizzo che la procedura sia corretta per $h-1$



Albero di altezza $\leq h-1$.
Tutti i suoi elementi sono
minori della radice

Albero di altezza $\leq h-1$.
Tutti i suoi elementi sono
maggiore della radice

Esempio

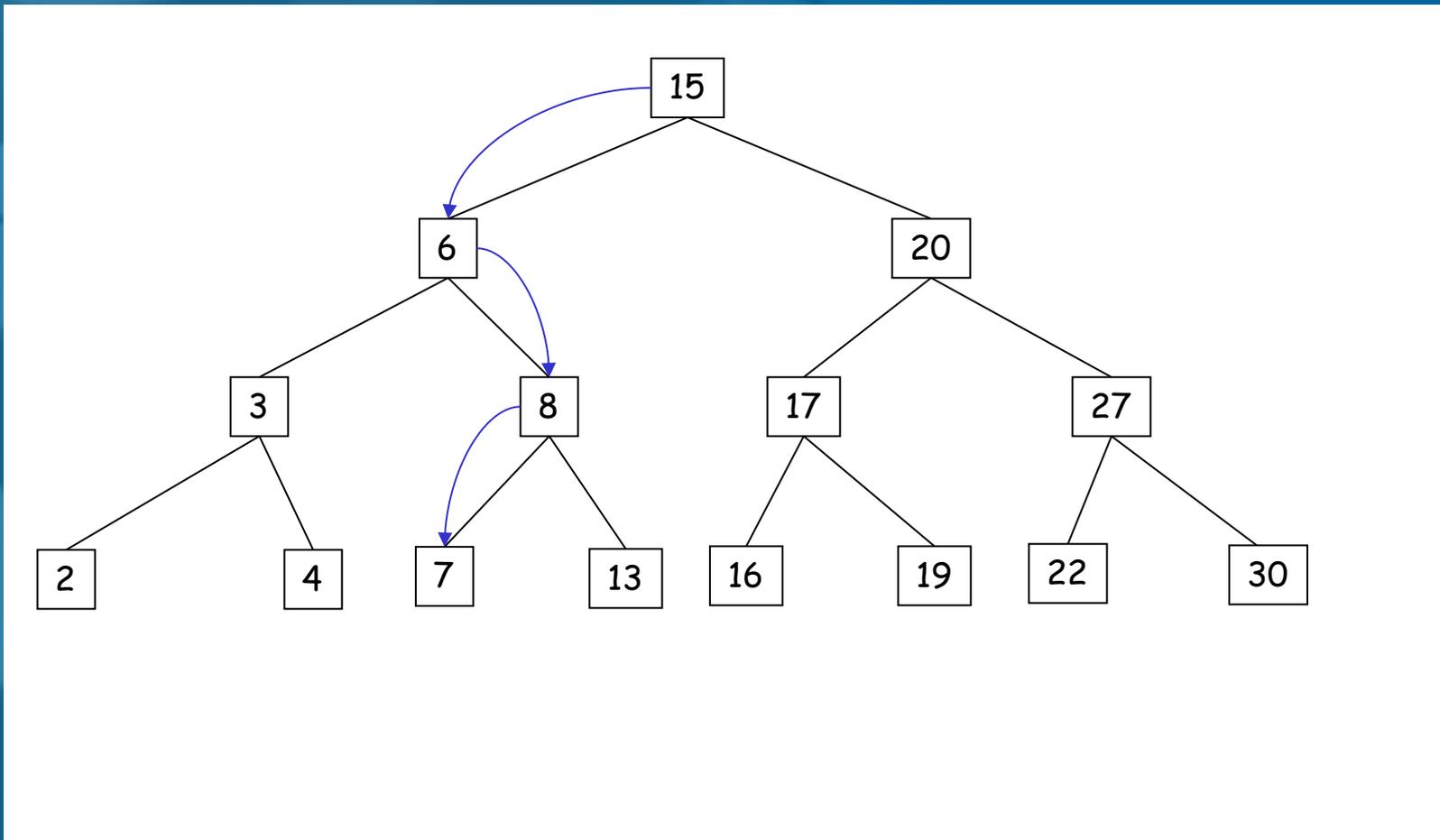


search(chiave k) \rightarrow elem

Analizziamo ora i costi di **search**, **insert** e **delete** su un ABR. Iniziamo dall'operazione di **search**. Per la proprietà di ordinamento totale appena mostrata, similmente alla ricerca binaria si traccia un cammino nell'albero partendo dalla radice: su ogni nodo, se la chiave cercata è diversa da quella del nodo, si decide se proseguire la ricerca nel sottoalbero **sinistro** o **destro**, a seconda se la chiave cercata è **minore** o **maggiore** della chiave del nodo corrente

```
algoritmo search(chiave k)  $\rightarrow$  elem
1.    $v \leftarrow$  radice di  $T$ 
2.   while ( $v \neq \text{null}$ ) do
3.     if ( $k = \text{chiave}(v)$ ) then return elem( $v$ )
4.     else if ( $k < \text{chiave}(v)$ ) then  $v \leftarrow \text{sin}(v)$ 
5.     else  $v \leftarrow \text{des}(v)$ 
6.   return null
```

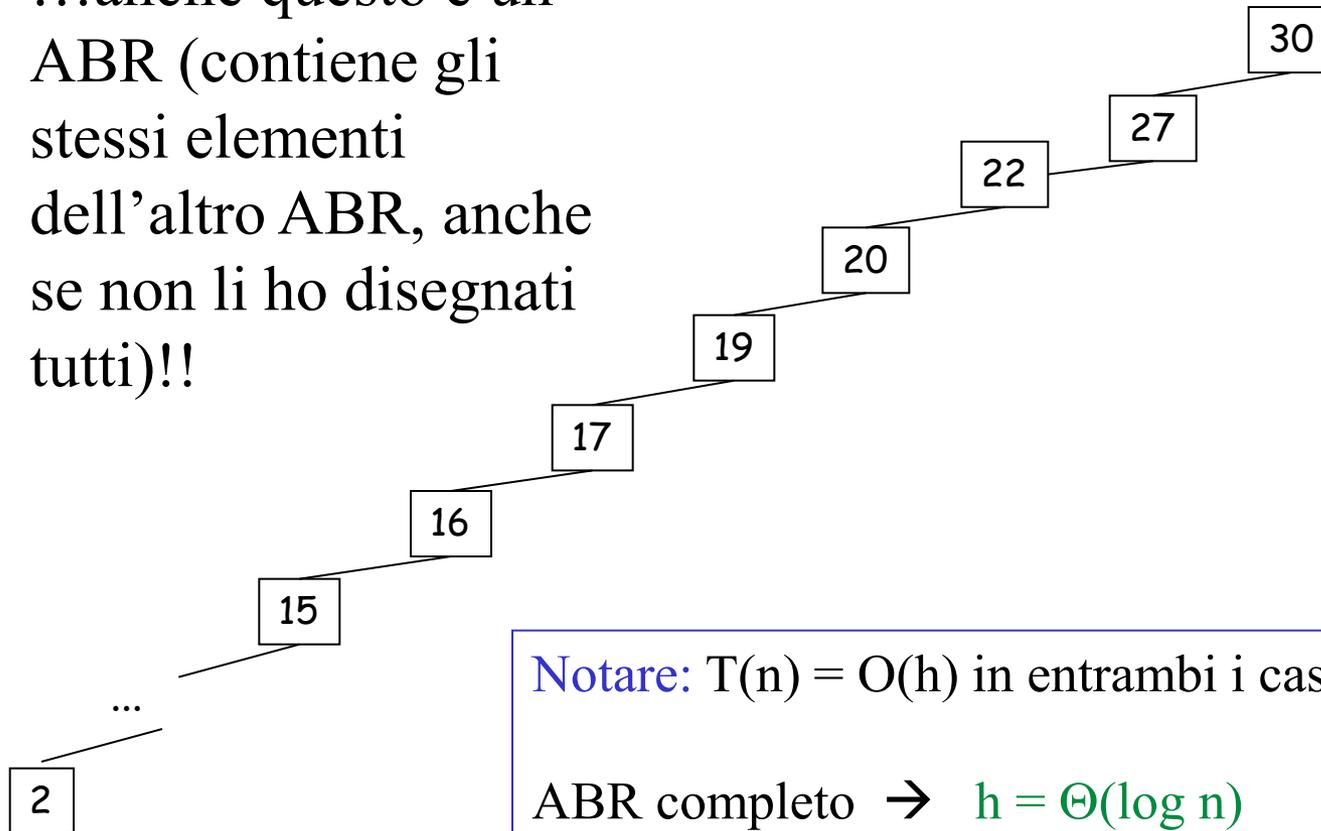
search(7)



Confronto con la ricerca binaria

- La complessità della procedura di ricerca considerata è $T(n) = O(h)$, ove h è l'altezza dell'ABR. Si noti che nel caso migliore, potrebbe costare $O(1)$
 - Nell'esempio precedente, l'ABR era **completo**, e quindi $h = \Theta(\log n)$
 - Per le proprietà dell'ABR, quando esso è completo, per ogni nodo v la chiave associata è l'elemento **mediano** nell'insieme ordinato delle chiavi associate ai nodi costituiti dal sottoalbero sinistro di v , da v , e dal sottoalbero destro di v
- ⇒ Ad ogni discesa di livello, dimezzo lo spazio di ricerca, in modo analogo a quanto avveniva per l'array ordinato!!
- ... ma un ABR non sempre è completo...

...anche questo è un ABR (contiene gli stessi elementi dell'altro ABR, anche se non li ho disegnati tutti)!!



Notare: $T(n) = O(h)$ in entrambi i casi, però:

ABR completo $\rightarrow h = \Theta(\log n)$

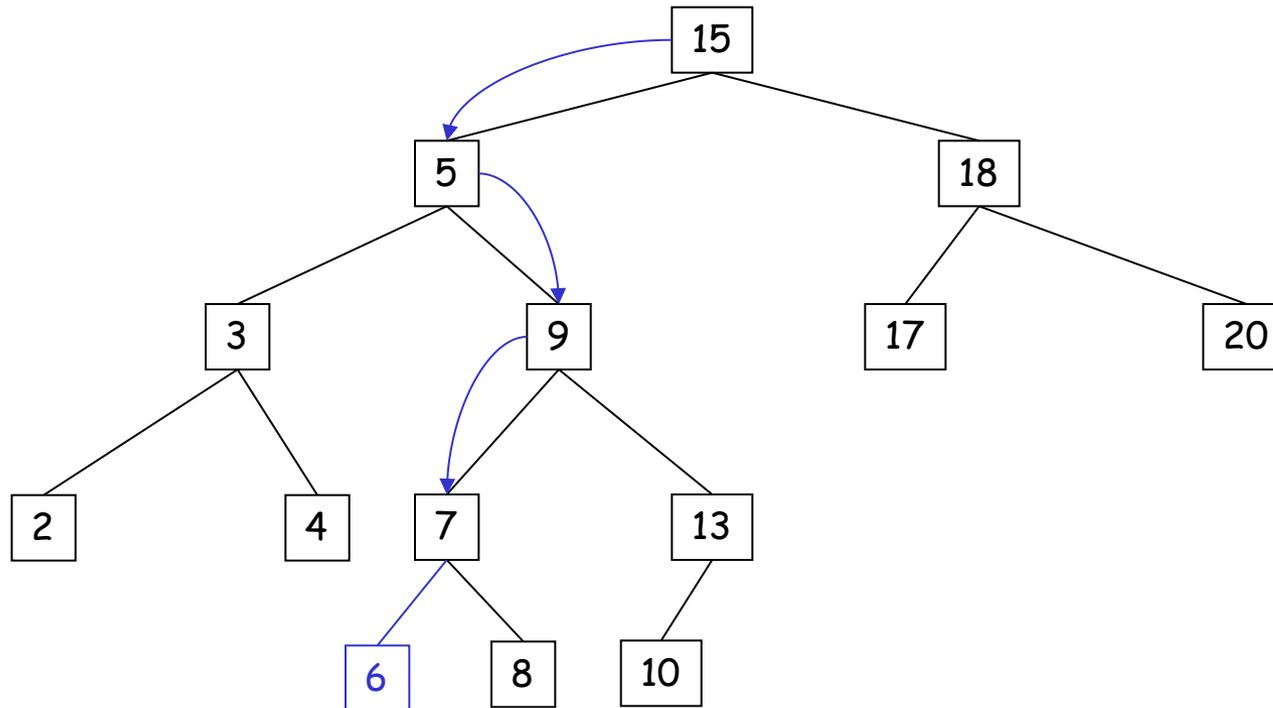
ABR "linearizzato" $\rightarrow h = \Theta(n)$

insert(elem e, chiave k)

La **insert** viene eseguita simulando una **search**, in modo da inserire il nuovo elemento proprio nella giusta posizione dell'ABR:

1. Cerca la chiave **k** nell'albero (che per l'ipotesi di univocità sulle chiavi non comparirà nell'ABR), identificando così il nodo **v** che diventerà padre del nodo che conterrà il nuovo elemento; tale nodo **v** deve essere un nodo dal quale la ricerca di **k** non può proseguire, e quindi **v** non ha sottoalbero sinistro e/o destro
 2. Crea un nuovo nodo **u** con **elem=e** e **chiave=k**
 3. Appendi **u** come figlio sinistro/destro di **v** in modo che sia mantenuta la proprietà di ordinamento totale
- ⇒ La complessità della procedura considerata è $T(n) = O(h)$, ove **h** è l'altezza dell'ABR

insert(e,6)



Se seguo questo schema l'elemento e viene posizionato nella posizione giusta. Infatti, per costruzione, ogni antenato di e si ritrova e nel giusto sottoalbero.

Interrogazioni ausiliarie su un ABR

Per implementare la **delete**, faremo invece uso delle seguenti operazioni ausiliare sull'ABR:

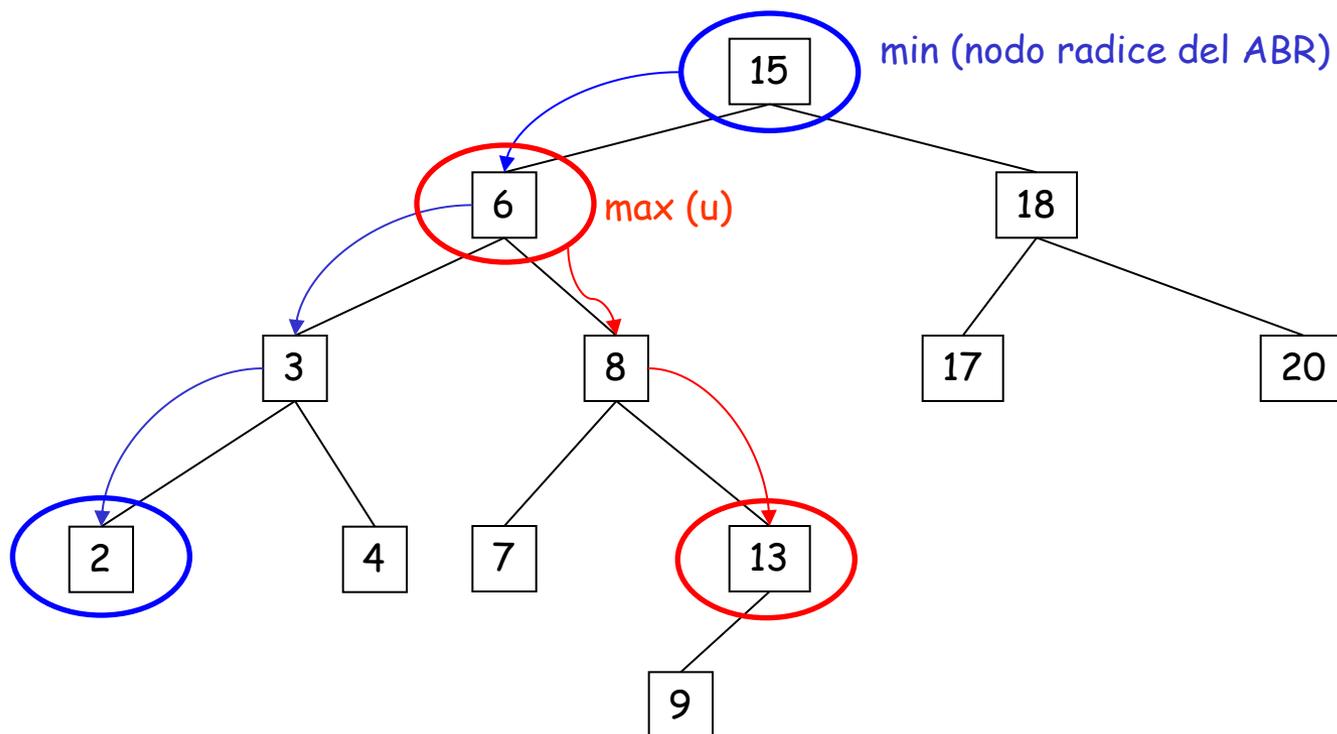
- **max(nodo u)** – dato un nodo **u** di un ABR, restituisce il nodo del sottoalbero dell'ABR radicato in **u** avente chiave **più grande** (si noti che potrebbe essere **u** stesso, se **u** non ha un sottoalbero destro)
- **min(nodo u)** – dato un nodo **u** di un ABR, restituisce il nodo del sottoalbero dell'ABR radicato in **u** avente chiave **più piccola** (si noti che potrebbe essere **u** stesso, se **u** non ha un sottoalbero sinistro)
- **predecessor(nodo u)** – dato un nodo **u** di un ABR, restituisce il nodo dell'ABR con chiave **immediatamente più piccola** di quella associata ad **u** (o NULL se **u** contiene l'elemento minimo dell'ABR).
- **successor(nodo u)** – dato un nodo **u** di un ABR, restituisce il nodo dell'ABR con chiave **immediatamente più grande** di quella associata ad **u** (o NULL se **u** contiene l'elemento massimo dell'ABR).

Ricerca del massimo/minimo

```
algoritmo max(nodo  $u$ )  $\rightarrow$  nodo  
1.    $v \leftarrow u$   
2.   while (  $des(v) \neq \text{null}$  ) do  
3.      $v \leftarrow des(v)$   
4.   return  $v$ 
```

- La procedura $\text{min}(\text{nodo } u)$ si definisce in maniera del tutto analoga cambiando “destro” con “sinistro”
 \Rightarrow La complessità della procedura considerata è $T(n) = O(h(u))$, ove n è il numero di nodi dell'ABR e $h(u)$ è l'altezza del sottoalbero radicato in u , e quindi $h(u) = O(h)$ (h è l'altezza dell'ABR)
- Se l'argomento $\text{nodo } u$ coincide con la radice dell'albero, allora min e max restituiscono il nodo con chiave **minima** e **massima** dell'intero ABR, rispettivamente

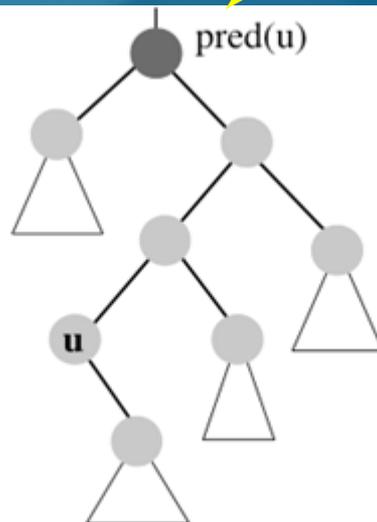
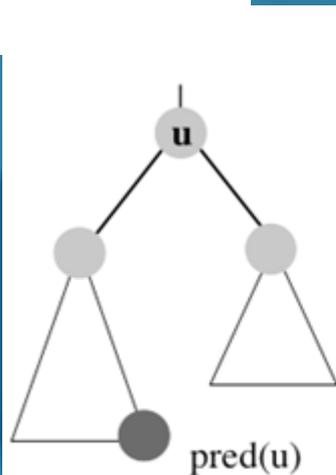
Esempio di esecuzione



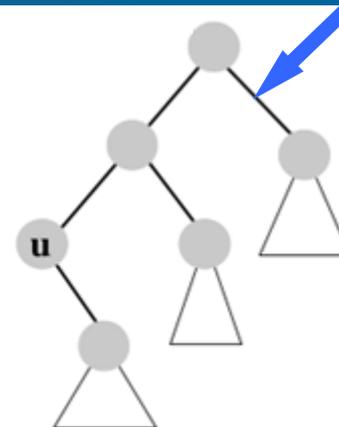
Ricerca del predecessore

Complessità: $O(h)$

Caso 1: u ha un sottoalbero sinistro: **max** del sottoalbero sinistro



Caso 2: u non ha un sottoalbero sinistro: allora $pred(u)$ è l'antenato più prossimo di u il cui sottoalbero **destro** contiene u (si noti infatti che u è il **minimo** tra gli elementi più grandi di $pred(u)$, e quindi u è il **successore** di $pred(u)$, ovvero $pred(u)$ è il predecessore di u); se tale antenato non esiste, allora u è il minimo dell'albero, e quindi non ha predecessore



Ricerca del predecessore

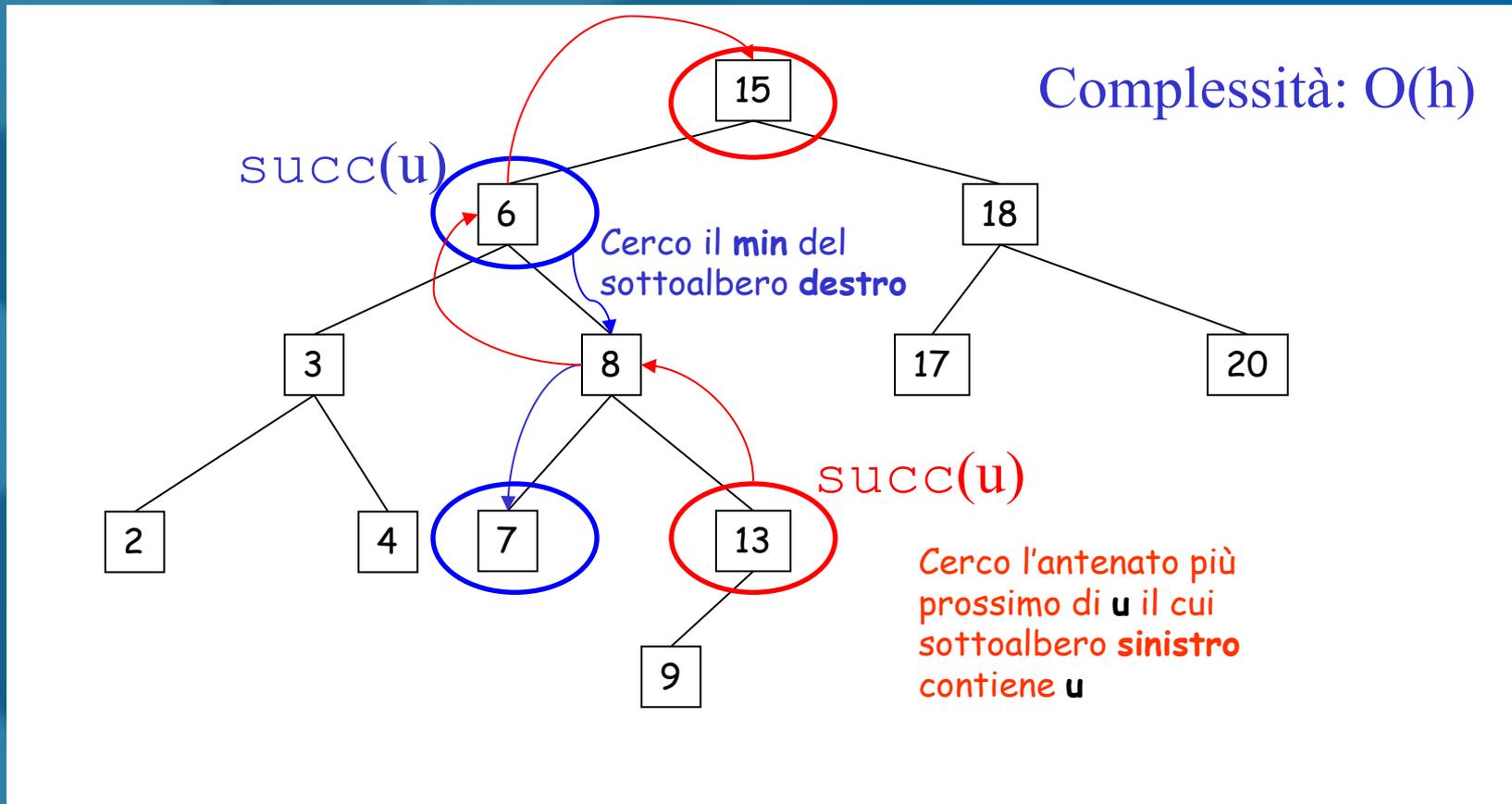
algoritmo $\text{pred}(\text{nodo } u) \rightarrow \text{nodo}$

Complessità: $O(h)$

1. **if** (u ha figlio sinistro $\text{sin}(u)$) **then**
2. **return** $\text{max}(\text{sin}(u))$
3. **while** ($\text{parent}(u) \neq \text{null}$ e u è figlio sinistro di suo padre) **do**
4. $u \leftarrow \text{parent}(u)$
5. **return** $\text{parent}(u)$

Ricerca del successore

La ricerca del **successore** di un nodo è simmetrica: cambio “sinistro” con “destro” e “max” con “min”

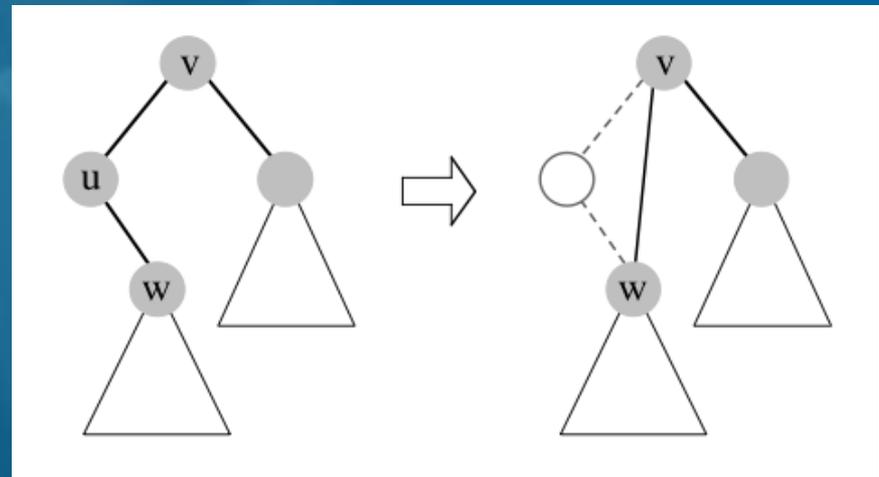


delete(elem e)

Siamo pronti per trattare l'operazione di cancellazione. Sia **u** il nodo contenente l'elemento **e** da cancellare, e si ricordi che viene fornito un puntatore diretto al nodo **u**; ci sono 3 possibilità:

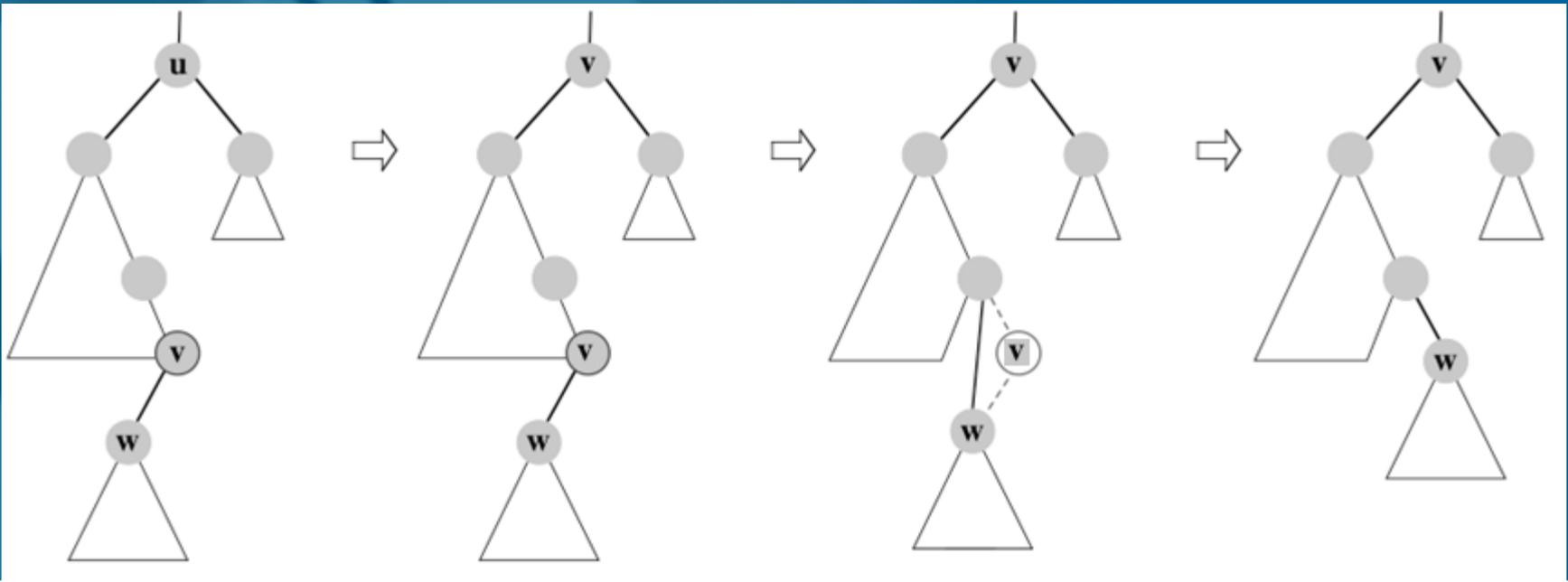
1) **u** è una foglia: rimuovila direttamente

2) **u** ha un solo figlio:
aggiralo



delete(elem e)

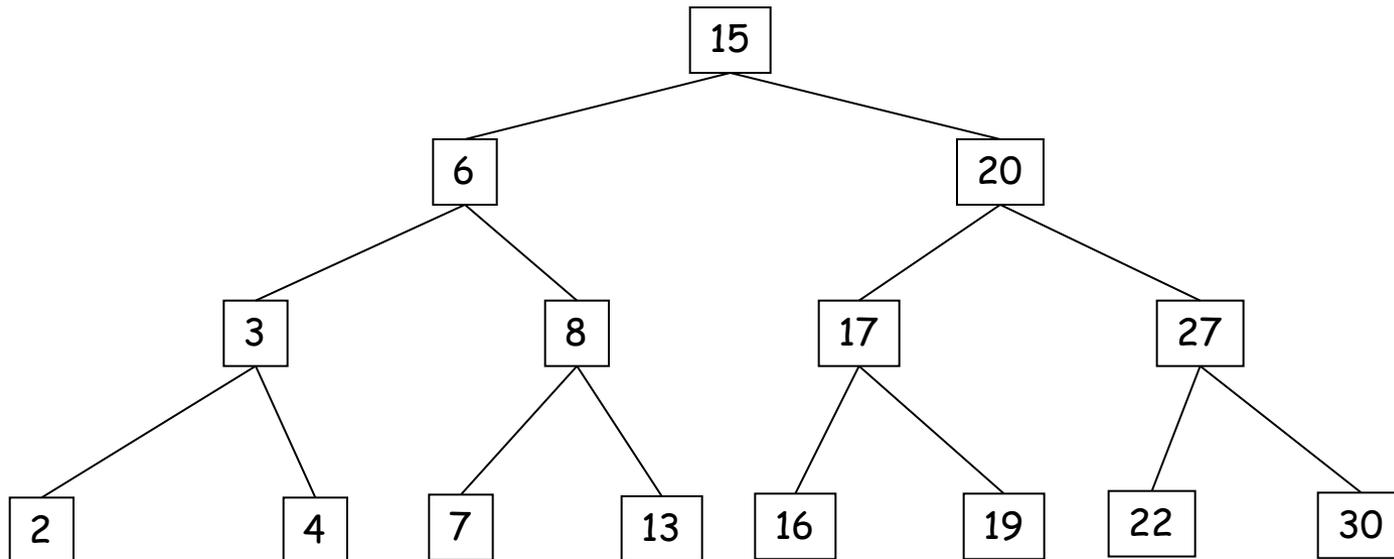
3) **u** ha due figli: sostituisilo con il **predecessore**, e rimuovi fisicamente il predecessore; tale predecessore sarà il massimo del sottoalbero sinistro (caso 1 dell'algoritmo **pred**), in quanto **u** ha un sottoalbero sinistro; quindi, tale predecessore deve avere al più **un solo** figlio (non può avere il **figlio destro!**), e ricadremo quindi in uno dei 2 casi precedenti. Si noti che si può analogamente usare allo stesso scopo il **successore** di **u**. Vediamo un esempio in cui si ricade nel caso 2):



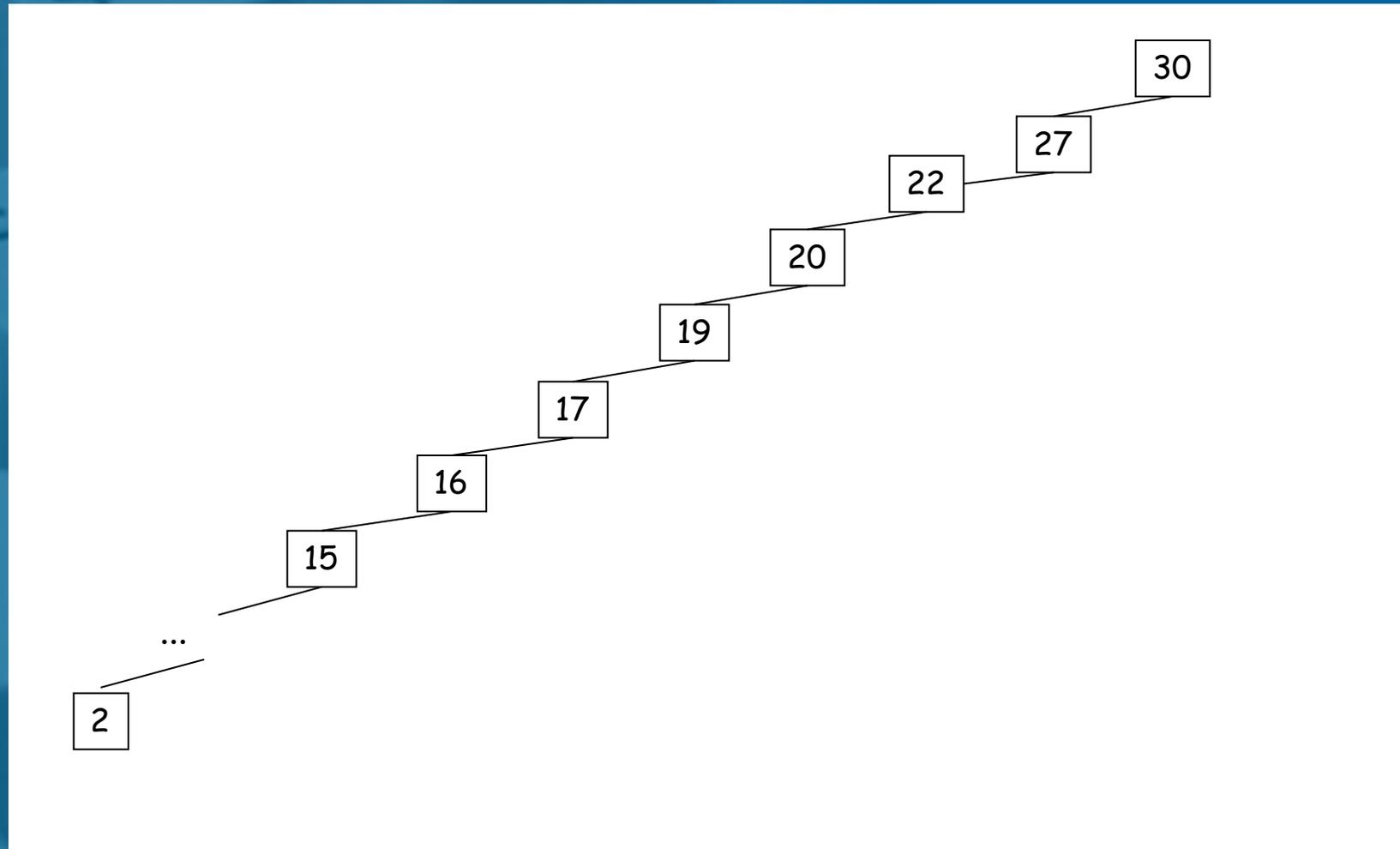
Costo dell'operazione di cancellazione

- Nei casi 1) e 2) $T(n)=O(1)$, mentre nel caso 3) $T(n)=O(h(u))=O(h)$
- Ricapitolando, le operazioni di **ricerca**, **inserimento** e **cancellazione** hanno costo $O(h)$ dove h è l'altezza dell'albero
- 😊 Per alberi molto “bilanciati”, $h = \Theta(\log n)$
- 😞 ...ma per alberi molto “sbilanciati”, $h = \Theta(n)$

Un albero binario di ricerca molto “bilanciato”...



Un albero binario di ricerca molto “sbilanciato”...



Esercizi di approfondimento

1. La **visita in ordine anticipato** di un ABR funziona come segue: dato un nodo v , elenca prima il nodo v , poi il sotto-albero sinistro di v (in ordine anticipato), e infine il sotto-albero destro di v (in ordine anticipato). Fornire lo pseudocodice di tale visita, nonché una sua esecuzione su un ABR **quasi completo** di altezza 4 (con chiavi a piacere).
2. La **visita in ordine posticipato** di un ABR funziona come segue: dato un nodo v , elenca prima il sotto-albero sinistro di v (in ordine posticipato), poi il sotto-albero destro di v (in ordine posticipato), e infine il nodo v . Fornire lo pseudocodice di tale visita, nonché una sua esecuzione su un ABR con 20 nodi di **altezza 5** (con chiavi a piacere).