

# 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  $e$

$search(chiave\ k) \rightarrow elem$   
se la chiave  $k$  è presente in  $S$  restituisce un elemento  $e$  ad essa associato, e null altrimenti

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

**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 soggetta 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 **decrescente**

- **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**; (notate una qualche differenza con la **Insert** su array ordinati che abbiamo definito per le **code di priorità**?)
- **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)

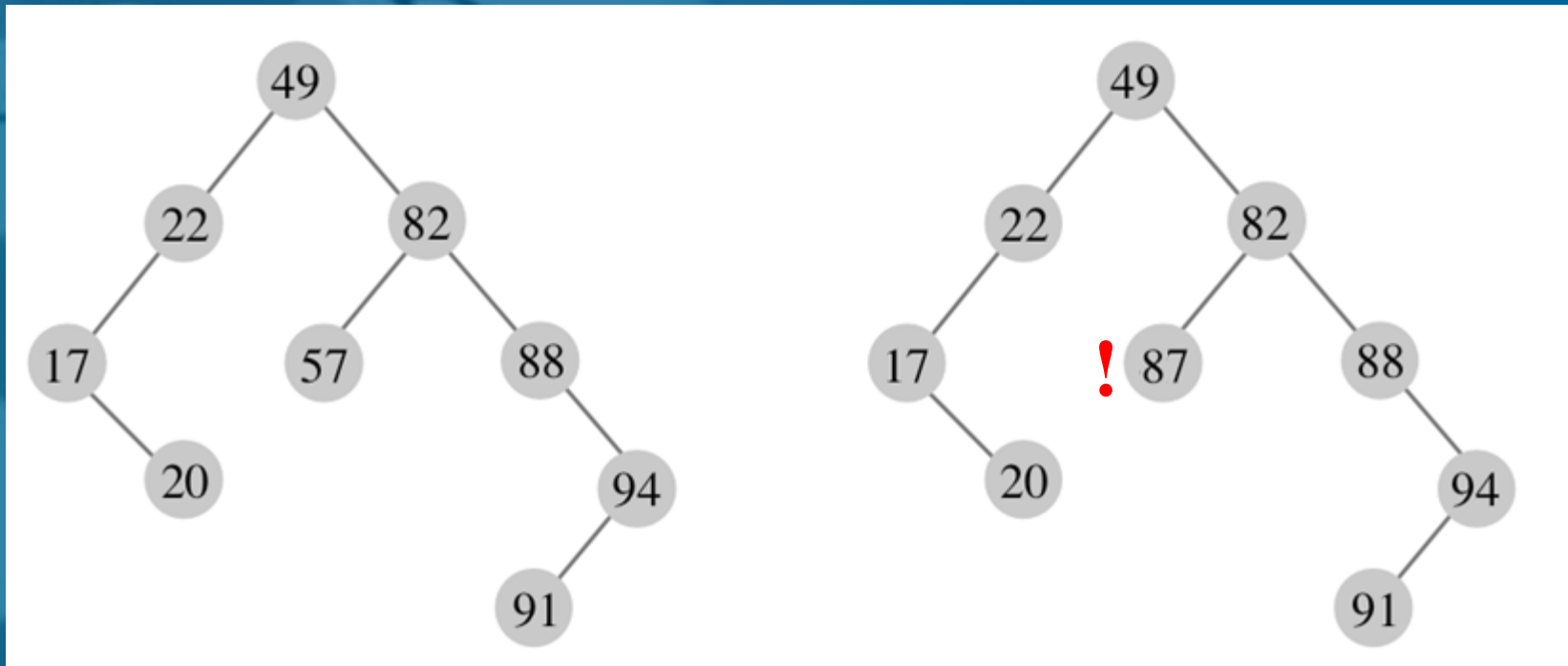
# Definizione di ABR

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

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

⇒ 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  $v$  (in ordine simmetrico), poi il nodo  $v$ , poi il sotto-albero destro di  $v$  (in ordine simmetrico)

```

algoritmo Inorder-tree-walk(node v)
if (v ≠ null)
    then Inorder-tree-walk(sin(v))
        stampa chiave(v)
        Inorder-tree-walk(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:

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

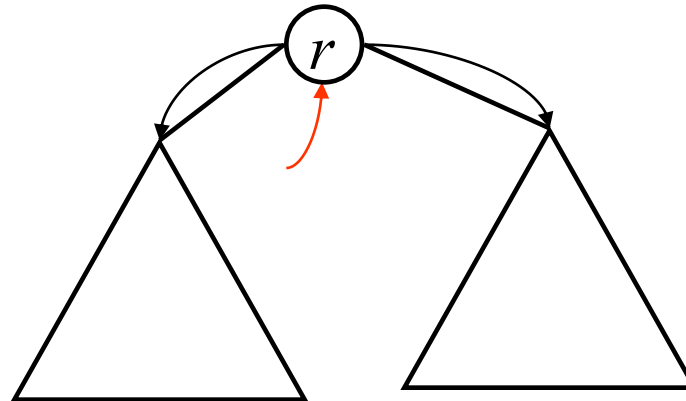
# Proprietà della visita simmetrica di un ABR

**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);

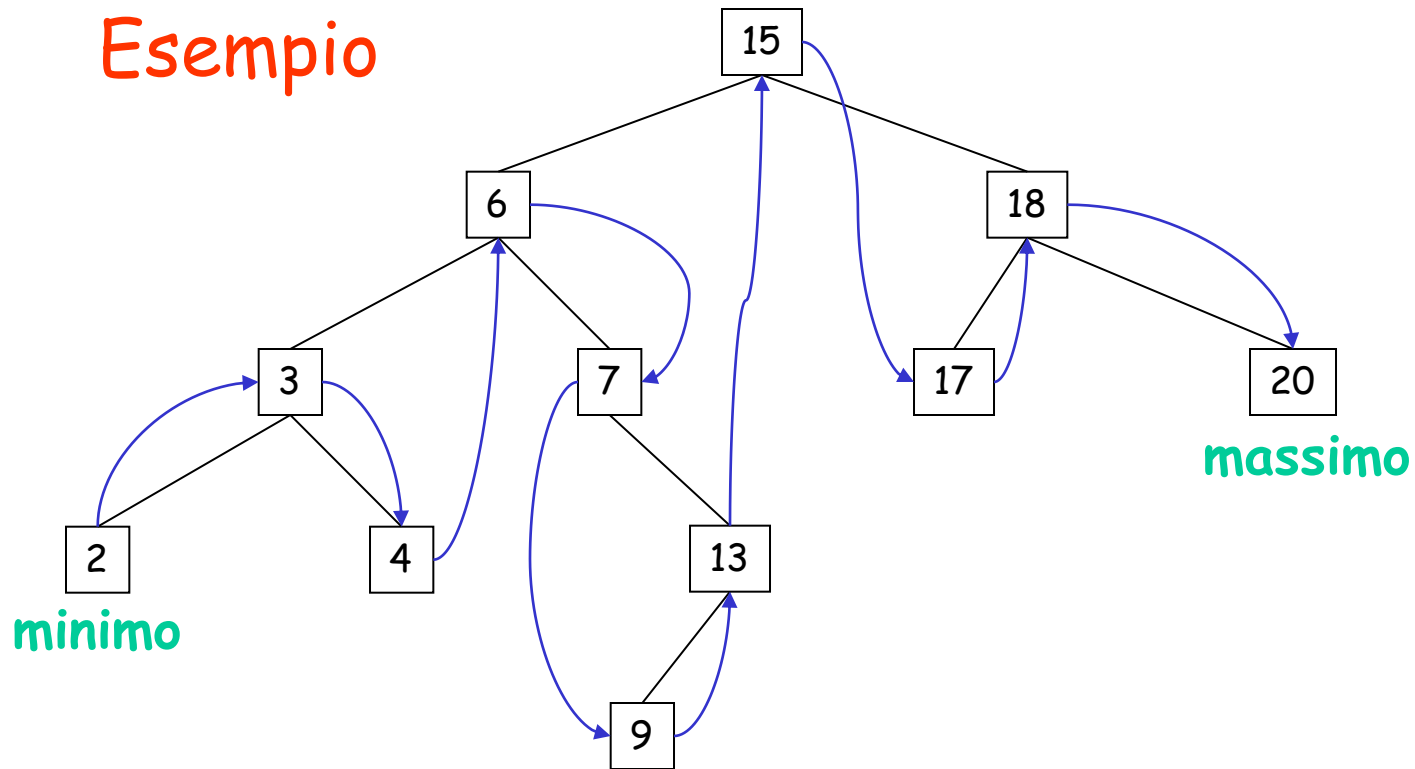
**Passo induttivo ( $h$  generico):** ipotizzo che la procedura sia corretta per  $h-1$



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  
maggiori della radice

## Esempio



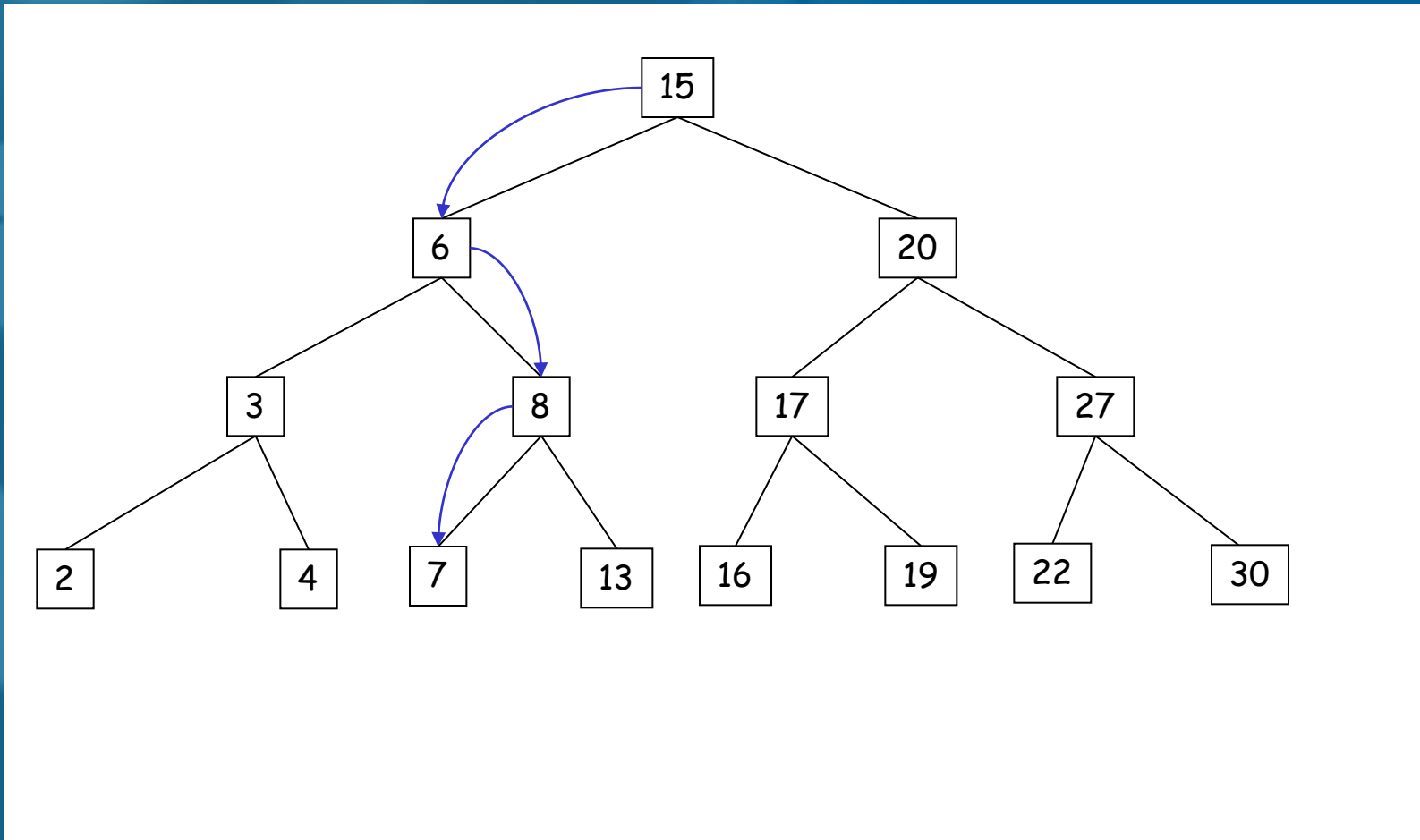


## search(chiave $k$ ) $\rightarrow$ elem

Traccia un cammino nell'albero partendo dalla radice: su ogni nodo, usa la proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro

```
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
```

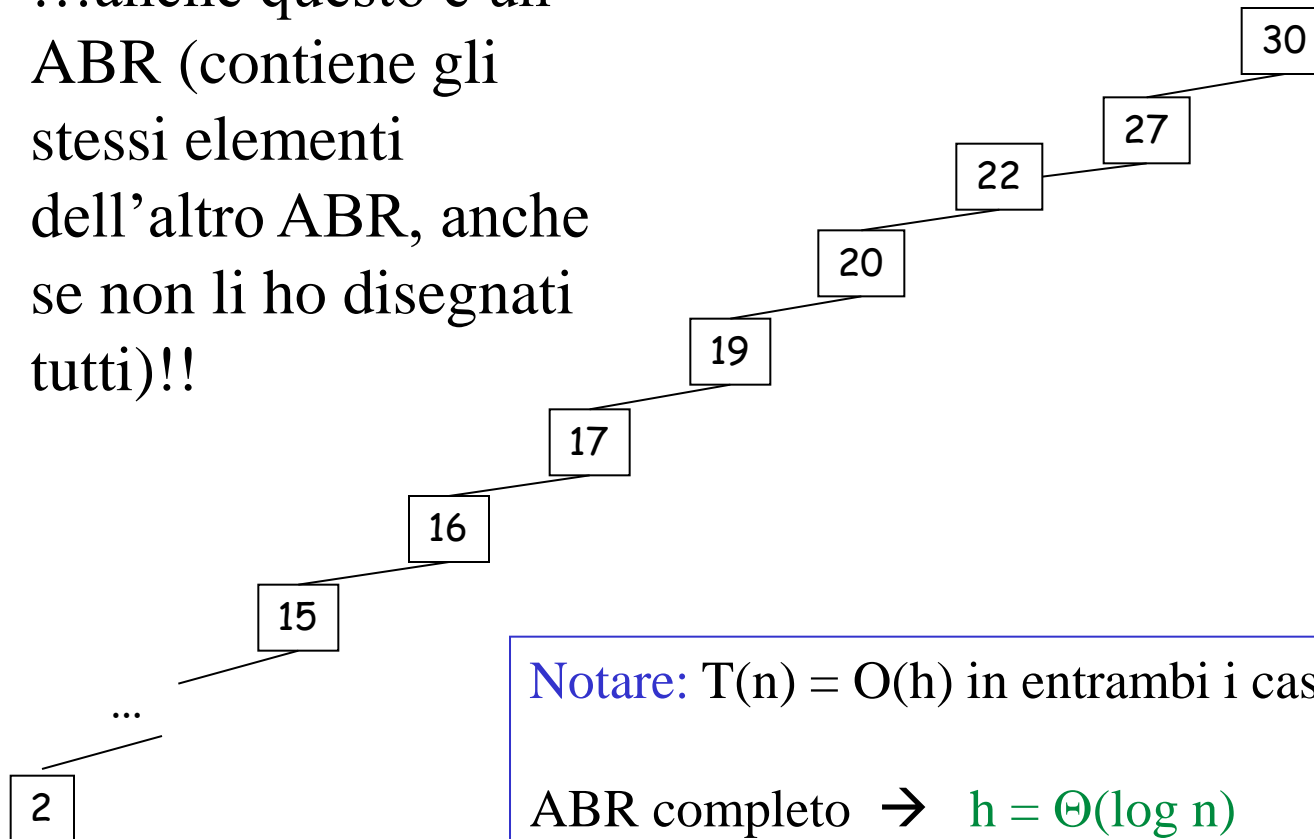
search(7)



# 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:

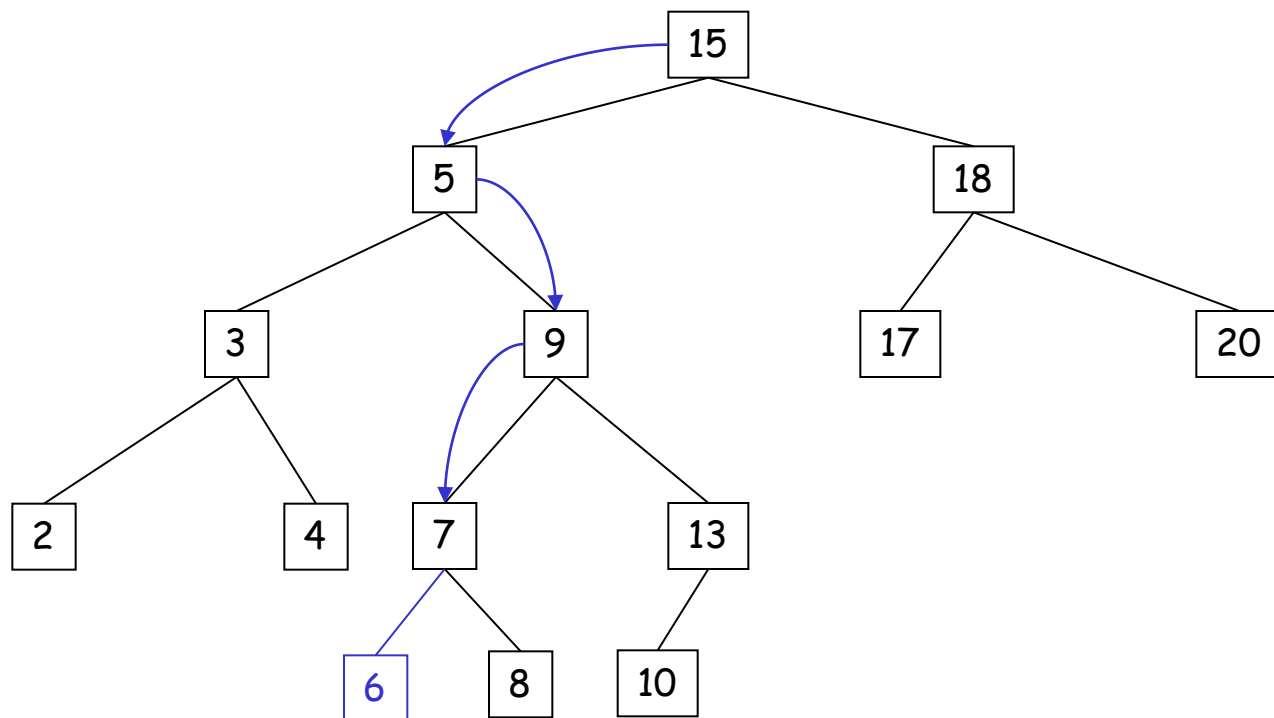
ABR completo  $\rightarrow h = \Theta(\log n)$

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

## insert(elem e, chiave k)

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.

# 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).