UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

MODULO 2: Alberi

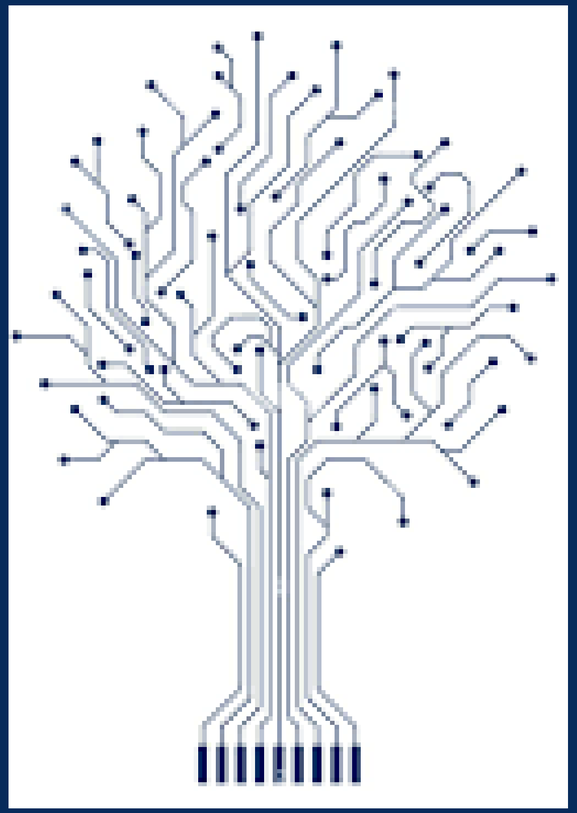
Prof.ssa Giulia Cisotto

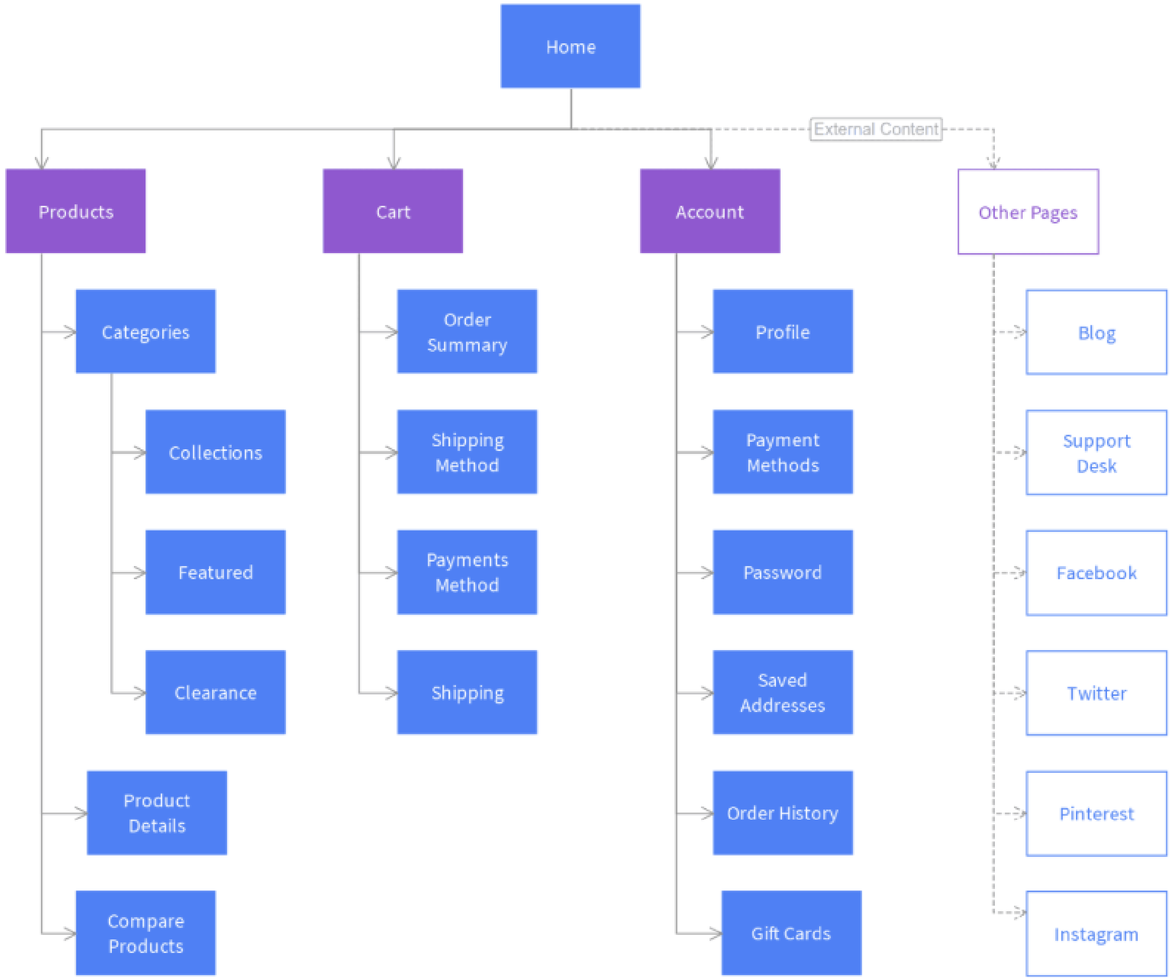
giulia.cisotto@units.it

Trieste, 30 aprile 2025

AGENDA DI OGGI

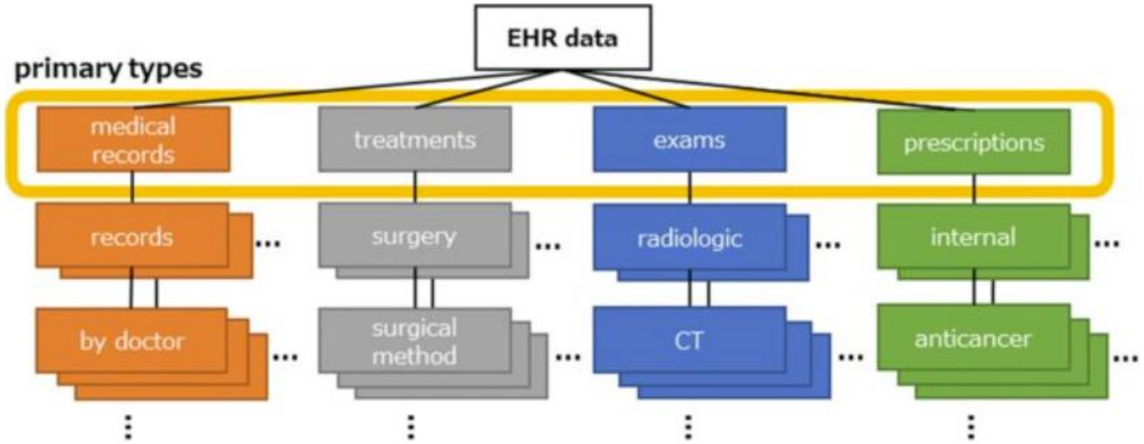
- Alberi binari
- Grafi



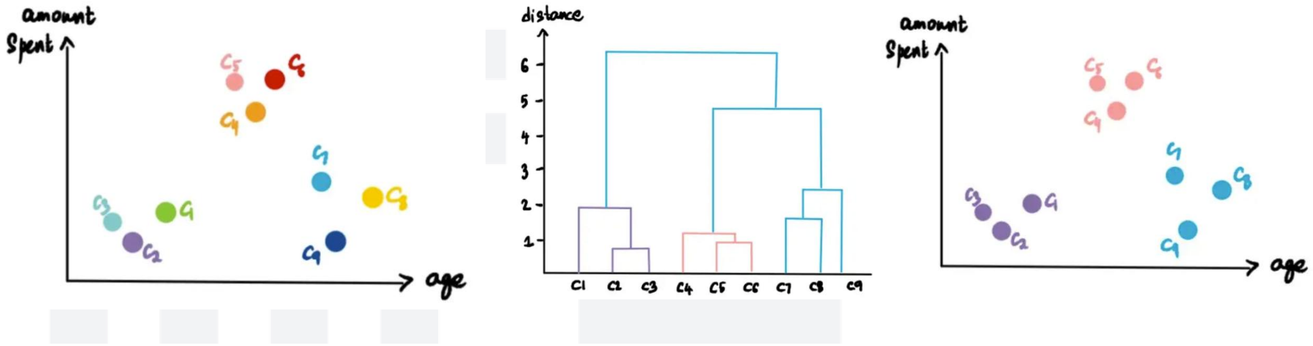


Sito web di e-commerce (link)

The example tree structure of EHR data types and primary types



Dendrogramma in clustering gerarchico (unsupervised **machine learning**)



UNIVERSITÀ DEGLI STUDI DI TRIESTE

ALBERI

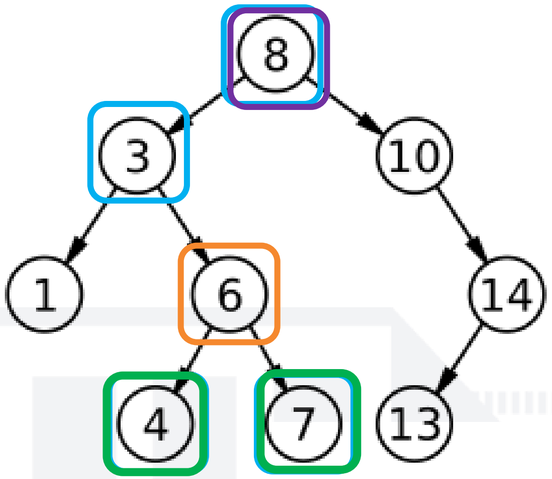
Sono strutture dati ASTRATTE e ORDINATE dove ogni oggetto della struttura è definito dai seguenti elementi:

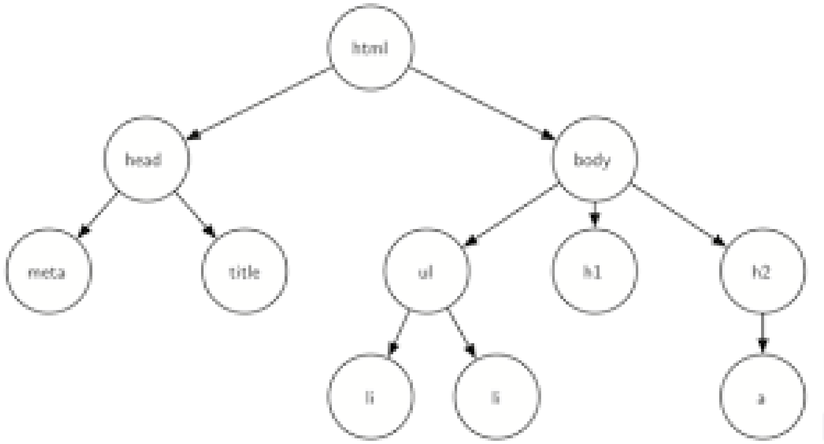
- **Chiave** (key) è un identificativo dell'oggetto
- **Contenuto** (dato che può essere un numero, una stringa, un'ulteriore struttura dati..)
- **Riferimenti** (o puntatori) all'oggetto precedente e agli oggetti successivi

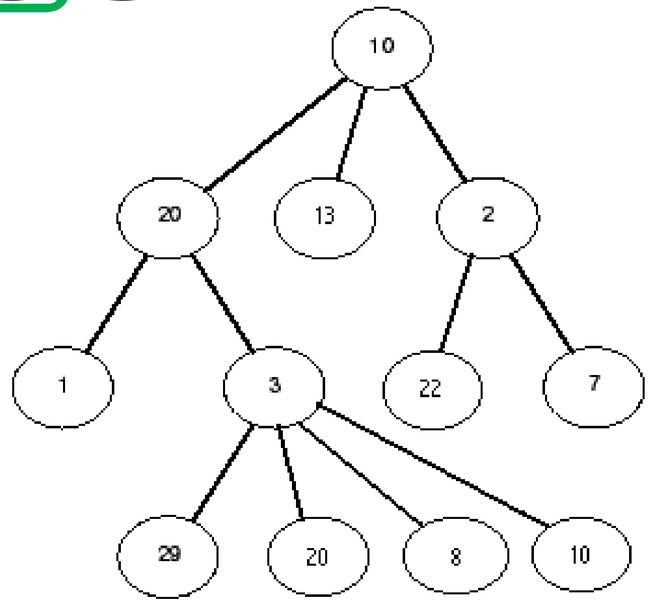
Vincoli (proprietà per poter definire la struttura proprio «un albero»):

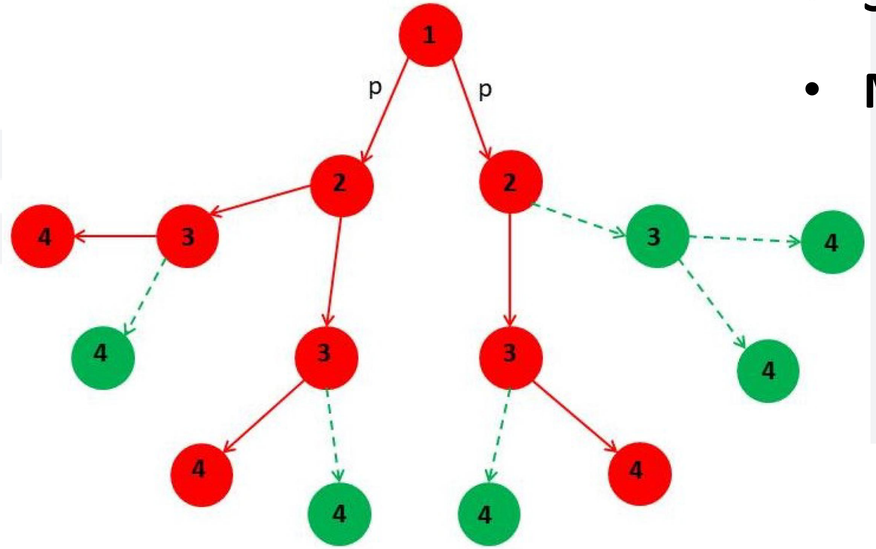
1) Per ogni oggetto c'è un solo oggetto precedente
2) Non ci sono cicli, ovvero non posso ritornare ciclicamente su un oggetto già visitato soltanto andando avanti

ALBERI: ESEMPI E DEFINIZIONI AGGIUNTIVE









- **Genitore (parent)**
- **Figlio (children)**
- **Radice (root)**
- **Nodo foglia (leaf node)**
- **Predecessore**
- **Successore**
- **Minimo e massimo**

VINCOLI E VANTAGGI

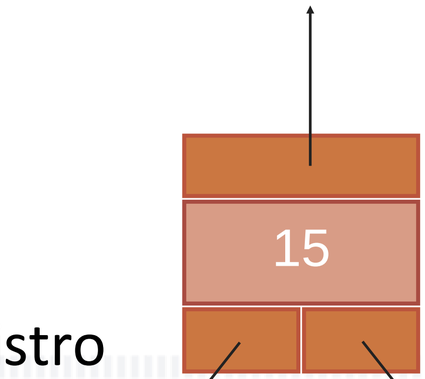
A seconda dell'applicazione posso imporre nuovi vincoli e «costringere» l'inserimento di nuovi elementi in determinate posizioni e «vietarne» l'inserimento in altre.

Intuitivamente: inserimento richiederà più «attenzione» (e tempo), ma la ricerca sarà avvantaggiata, sapendo già dove sarà – con maggior probabilità – l'elemento cercato.

Una delle tipologie di alberi maggiormente utilizzata sono gli:

ALBERI BINARI DI RICERCA O BINARY SEARCH TREE (BST)

BST: NODO DI UN ALBERO BINARIO



Reference al nodo genitore

Contenuto del nodo (in particolare la *chiave*)

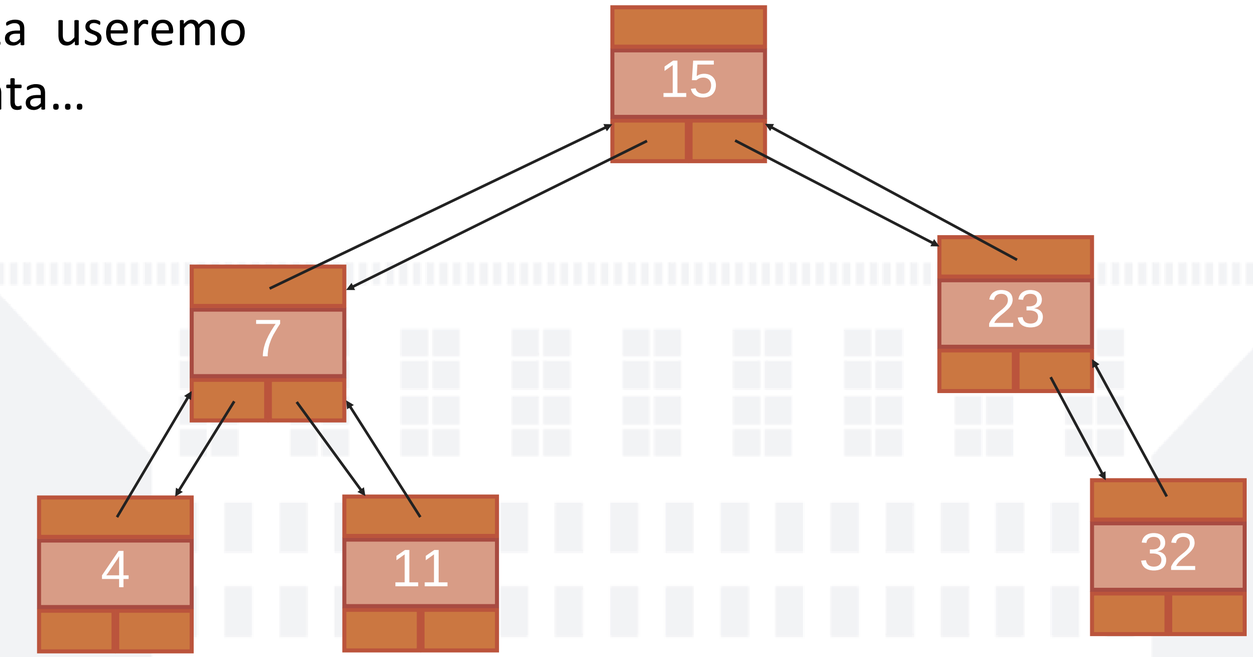
Reference al figlio sinistro

Reference al figlio destro

A seconda dell'implementazione e delle operazioni da svolgere potremmo non avere il riferimento al nodo genitore, avere un riferimento al nodo "fratello", etc.

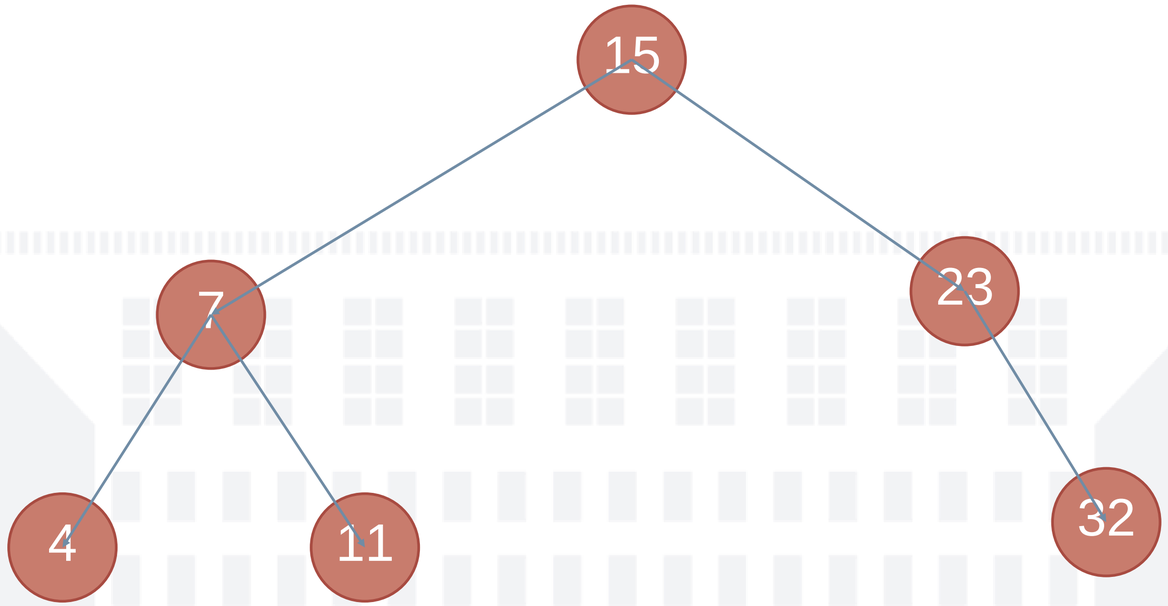
BST: RAPPRESENTAZIONE GRAFICA

Qui visualizziamo tutti i riferimenti, ma spesso per chiarezza useremo una notazione più stilizzata…



… ricordando che in ogni caso tutti questi reference continuano ad esistere

BST: RAPPRESENTAZIONE GRAFICA

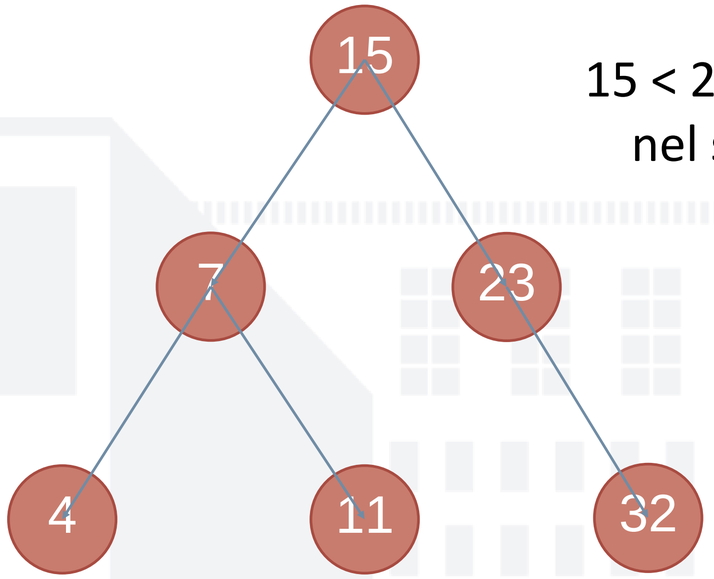


BST: VINCOLI

- Richiede **chiavi totalmente ordinate** (es. interi)
- Ogni nodo dell'albero contiene una chiave
- Ogni nodo dell'albero ha la seguente **proprietà**:
  - Tutti i nodi nel sottoalbero sinistro hanno chiave *minore* della chiave nel nodo
  - Tutti i nodi del sottoalbero destro hanno chiave *maggiore* della chiave nel nodo

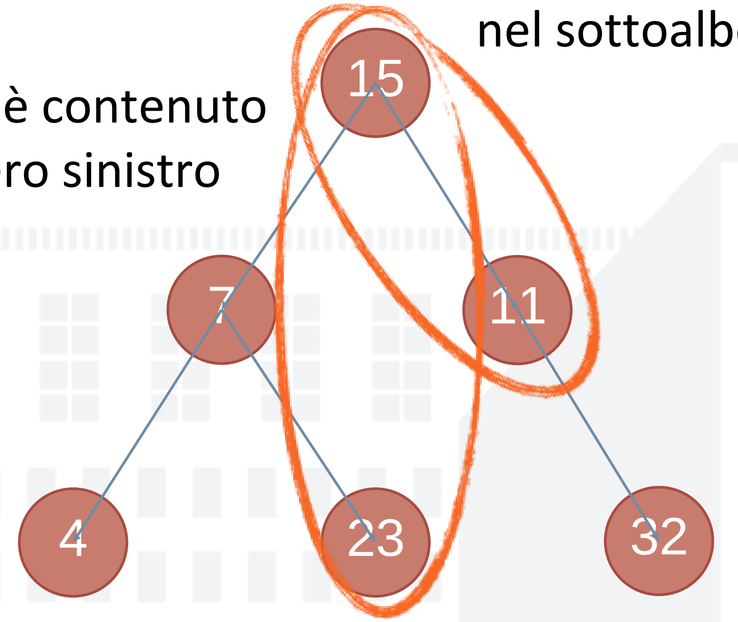
- **Ogni sottoinsieme dell'albero è, a sua volta, un BST** che gode delle stesse proprietà.

ALBERO BINARIO DI RICERCA



15 < 23, ma 11 è contenuto nel sottoalbero sinistro

15 > 11, ma 11 è contenuto nel sottoalbero destro



E’ un albero binario di ricerca

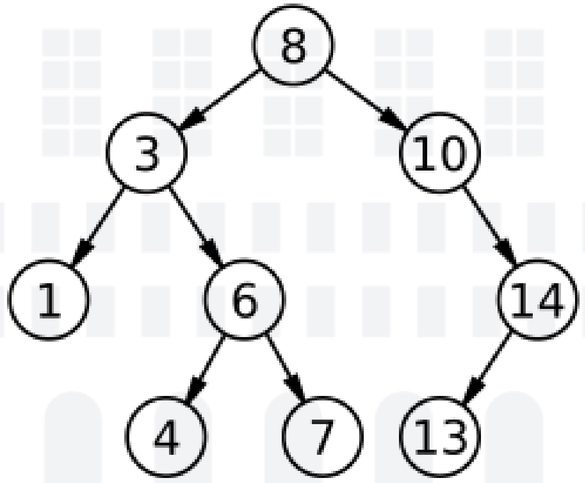
NON è un albero binario di ricerca

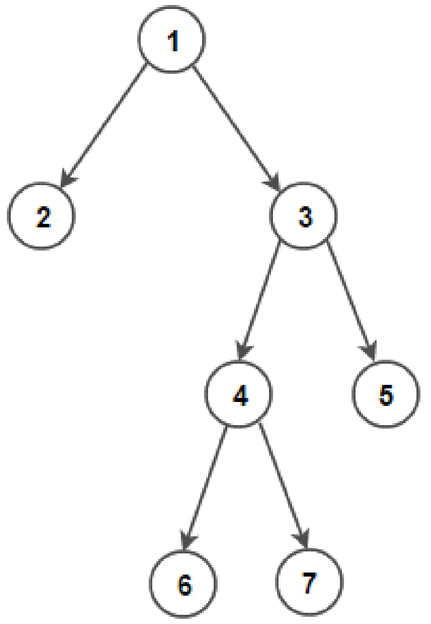
ALBERI BINARI: ULTERIORI DEFINIZIONI UTILI

ALTEZZA DI UN ALBERO (h)**:** distanza (in termini di step tra un nodo e l'altro) che porta dalla *root* al nodo foglia più basso possibile

PROFONDITA' DI UN NODO: distanza dalla *root* ad un nodo specifico.

ALBERO BILANCIATO: è un albero in cui la differenza assoluta tra l'altezza del sottoalbero sinistro e destro *per ogni nodo* è 0 o 1.





NOTA: Per un albero binario e bilanciato, $h \approx log_2 n$.

BST: RICERCA

- Dato un albero binario e una chiave, la ricerca può essere suddivisa in ***quattro casi***:
- **Albero vuoto**: l'elemento non è contenuto
  - La chiave della radice è quella che stiamo cercando: abbiamo trovato l'elemento
- La chiave che stiamo cercando è **minore** della chiave nella radice: cerchiamo ricorsivamente nel sottoalbero di **sinistra**
- La chiave che stiamo cercando è **maggiore** della chiave nella radice: cerchiamo ricorsivamente nel sottoalbero di **destra**

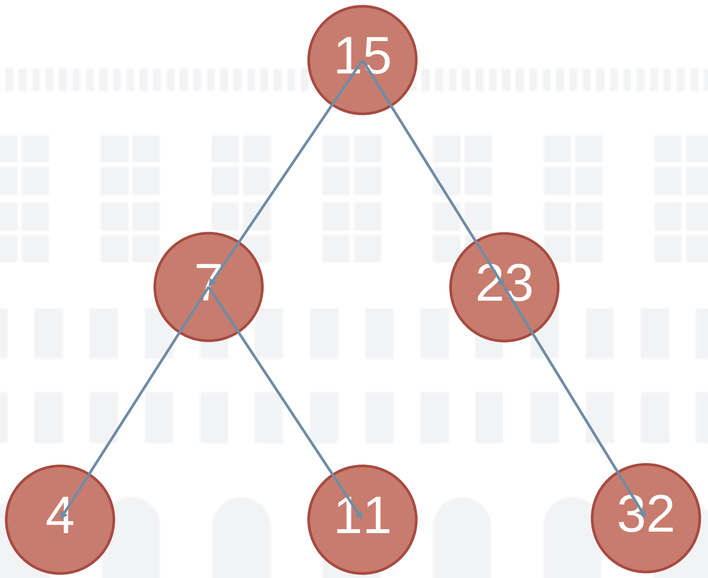
BST: PSEUDOCODICE DELLA RICERCA

```
Parametri: Nodo radice, key
if radice is None
   return None # l'albero è vuoto, quindi sicuramente la chiave non esiste
if radice.key == key
   return radice # abbiamo trovato un nodo con la chiave che cercavamo
if key < radice.key
   return ricerca(radice.left, key) # ricerca nel sottoalbero sinistro
else # ovvero key > radice.key
   return ricerca(radice.right, key) # ricerca nel sottoalbero destro
```

Ricorsività!!

ALBERO BINARIO: RICERCA

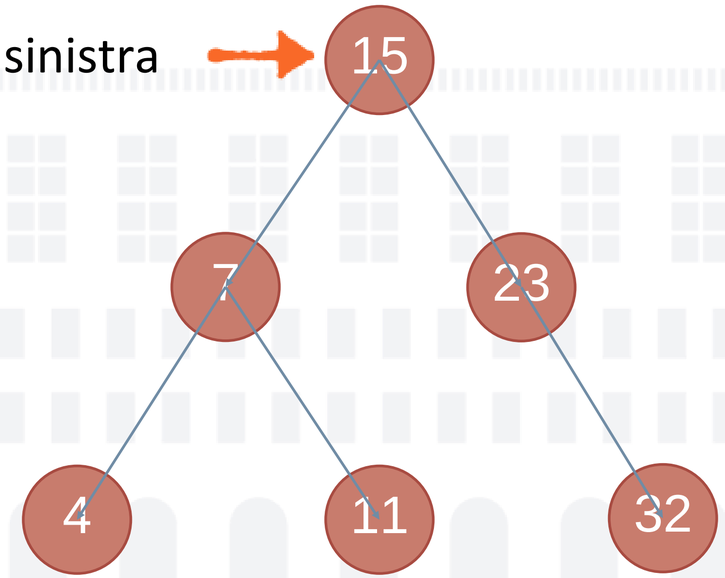
Supponiamo di voler trovare l’elemento di chiave 11



ALBERO BINARIO: RICERCA

Supponiamo di voler trovare l’elemento di chiave 11

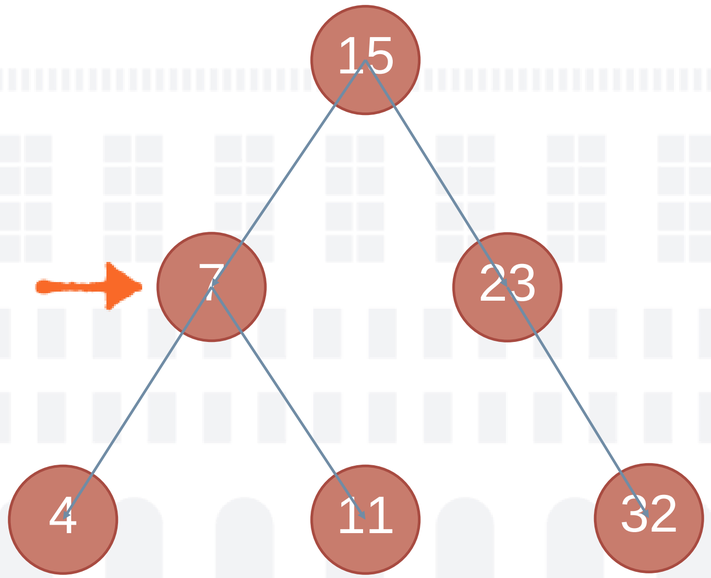
11 < 15, quindi ci muoviamo a sinistra



ALBERO BINARIO: RICERCA

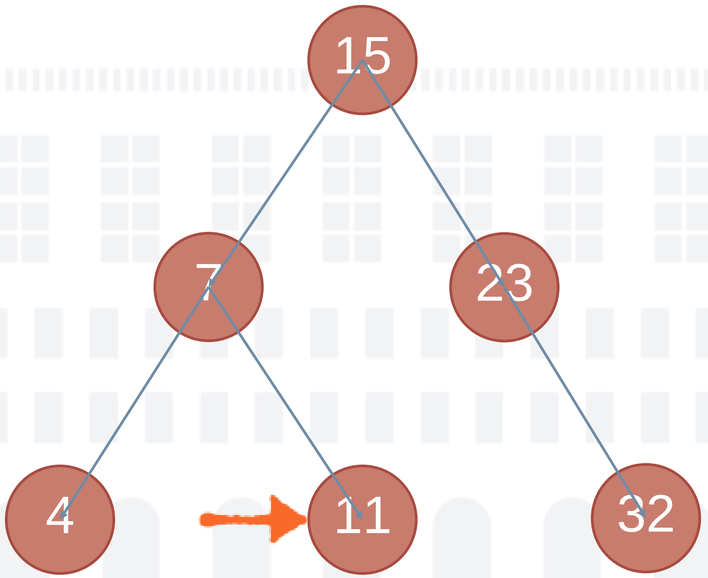
Supponiamo di voler trovare l'elemento di chiave 11

11 > 7, quindi ci muoviamo a destra



ALBERO BINARIO: RICERCA

Supponiamo di voler trovare l’elemento di chiave 11

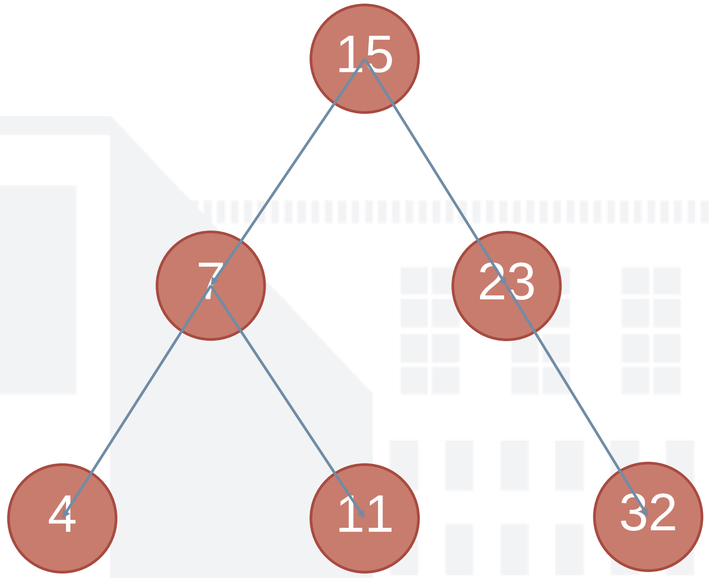


Trovato!

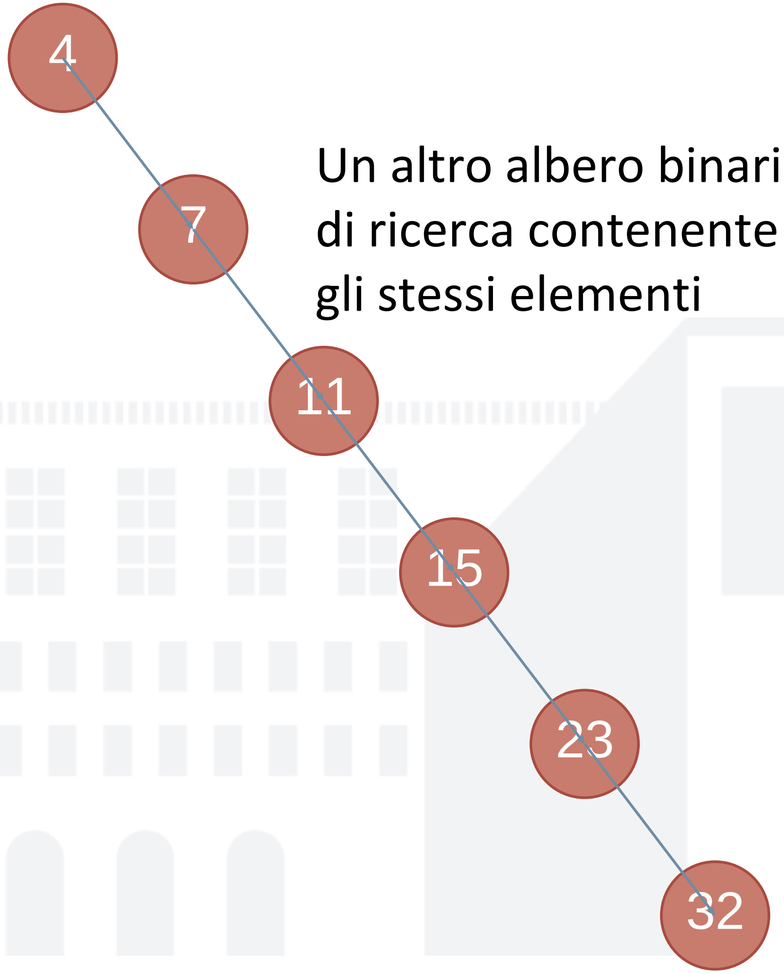
BST: COMPLESSITA' DELLA RICERCA

- Facciamo un ***numero costante di passi prima*** di ogni chiamata ricorsiva…
- … E facciamo un ***numero di chiamate ricorsive*** che è limitato dall'**altezza dell'albero** (ad ogni chiamata ricorsiva scendiamo di un livello)
- Quindi $O(h)$ dove h è l'altezza dell'albero.

BST BILANCIATI E SBILANCIATI:



Albero binario di ricerca



Un altro albero binario
di ricerca contenente
gli stessi elementi

CONSEGUENZE

- **Nel caso migliore** abbiamo un albero con la profondità minima necessaria a contenere tutti gli elementi, un BST bilanciato, quindi la ricerca è particolarmente efficiente: $\boldsymbol{O(logn)}$
- **Nel caso peggiore** abbiamo qualcosa di *simile ad una lista concatenata*, la profondità dell'albero è lineare rispetto al numero di elementi e **la ricerca richiede tempo** $\boldsymbol{O(n)}$

VISITA IN PRE-ORDINE, IN-ORDINE E POST-ORDINE

- Gli alberi binari generalmente hanno diversi modi in cui possono enumerare gli elementi (visitandoli tutti)
- I tre modi principali differiscono solo per il momento in cui viene visitato il nodo corrente:
  - **Pre-ordine**: nodo corrente, sottoalbero *sinistro*, sottoalbero destro
  - **In-ordine**: sottoalbero *sinistro*, nodo corrente, sottoalbero destro
  - **Post-ordine**: sottoalbero *sinistro*, sottoalbero destro, nodo corrente

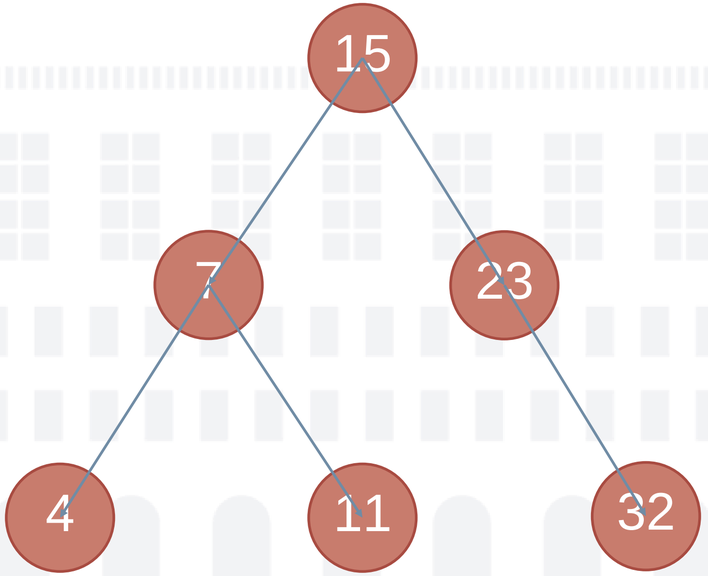
VISITA DI UN BST: ESEMPIO

Visita in pre-ordine: 15 7 4 11 23 32

Visita in ordine: 4 7 11 15 23 32

Visita in post-ordine: 4 11 7 32 23 15

La visita in ordine visita sempre gli elementi **in ordine del valore della chiave**



Visita in pre-ordine: parto dalla radice, entro nel sottoalbero sinistro. Il «parent» è il nodo corrente, quindi visito il suo sottoalbero… Una volta esaurito il sottoalbero sinistro, allora riparto dalla radice e ripeto per il sottoalbero di destra.

Visita in post-ordine: parto dal basso a sinistra, visito la prima foglia a sinistra, poi l'eventuale «fratello», poi il parent, così esaurisco il sottoalbero sinistro. Poi riparto dalla foglia più a sinistra del sottoalbero di destra e ripeto…

ALBERI BINARI: INSERIMENTO (1/2)

L’inserimento avviene in modo simile alla ricerca

Data una chiave k dobbiamo trovare un posto libero per inserire quella chiave **rispettando la proprietà dell’albero binario di ricerca**

Confrontiamo k con la chiave nella radice:

- Se k è maggiore, andrà inserita a destra della radice
- Se k è minore, andrà inserita a sinistra della radice

ALBERI BINARI: INSERIMENTO (2/2)

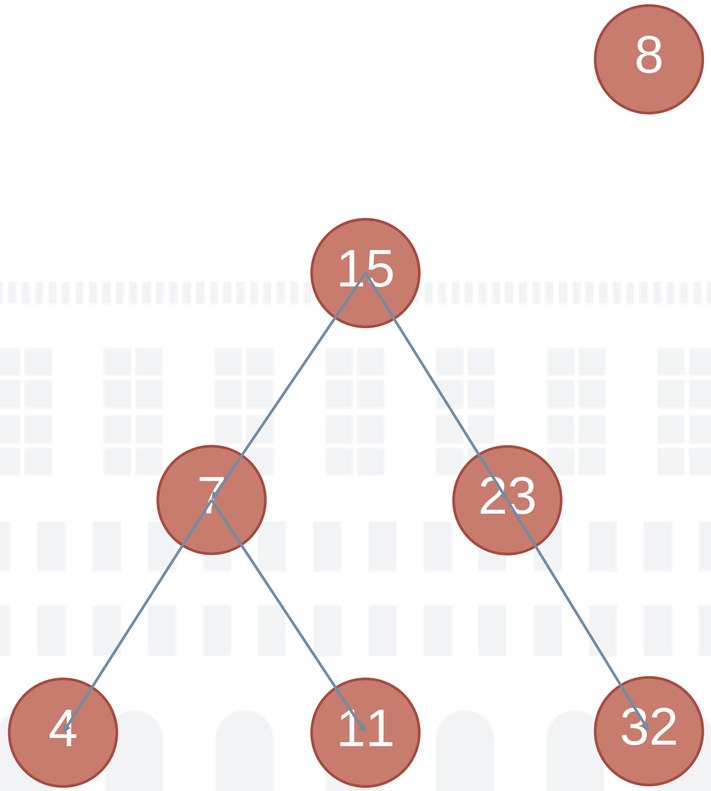
Supponiamo di dover proseguire a sinistra (k minore del valore nella radice). Abbiamo *due casi*:

- **La radice non ha un figlio sinistro**: possiamo direttamente inserire k come figlio sinistro della radice
- **La radice ha un figlio sinistro**: chiamiamo ricorsivamente l'inserimento nell'albero di radice il figlio sinistro

Simmetricamente se si prosegue a destra.

ALBERO BINARIO: INSERIMENTO

Supponiamo di voler inserire un elemento di chiave 8

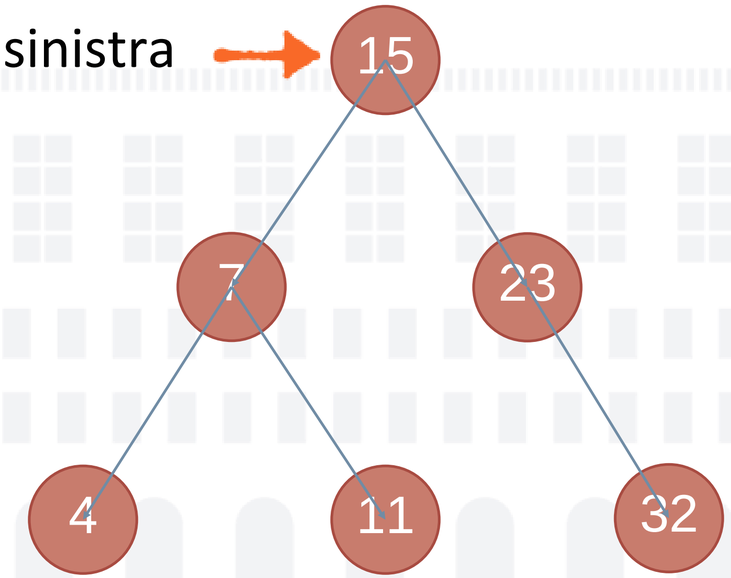


ALBERO BINARIO: INSERIMENTO

Supponiamo di voler inserire un elemento di chiave 8

8

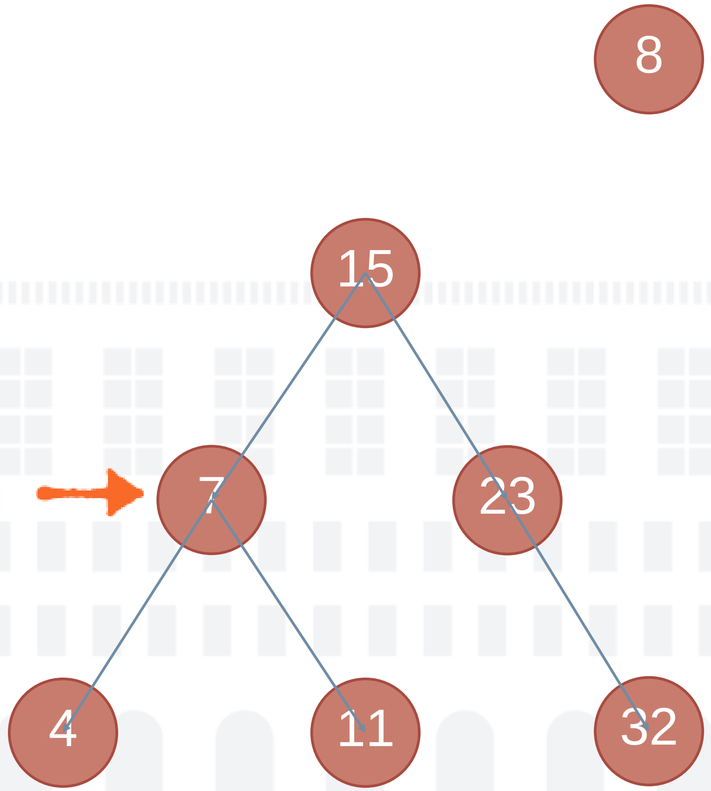
8 < 15, quindi ci muoviamo a sinistra



ALBERO BINARIO: INSERIMENTO

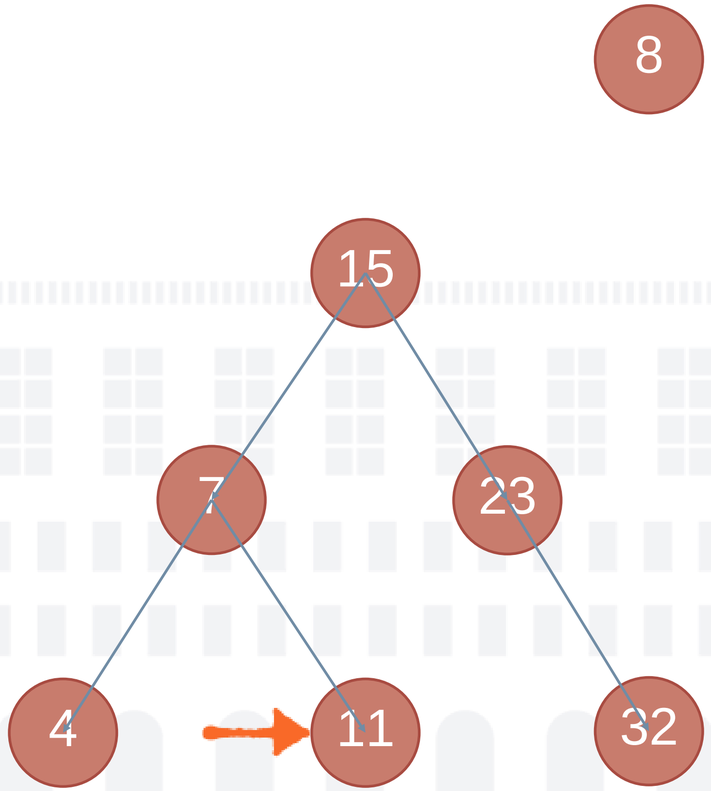
Supponiamo di voler inserire un elemento di chiave 8

8 > 7, quindi ci muoviamo a destra



ALBERO BINARIO: INSERIMENTO

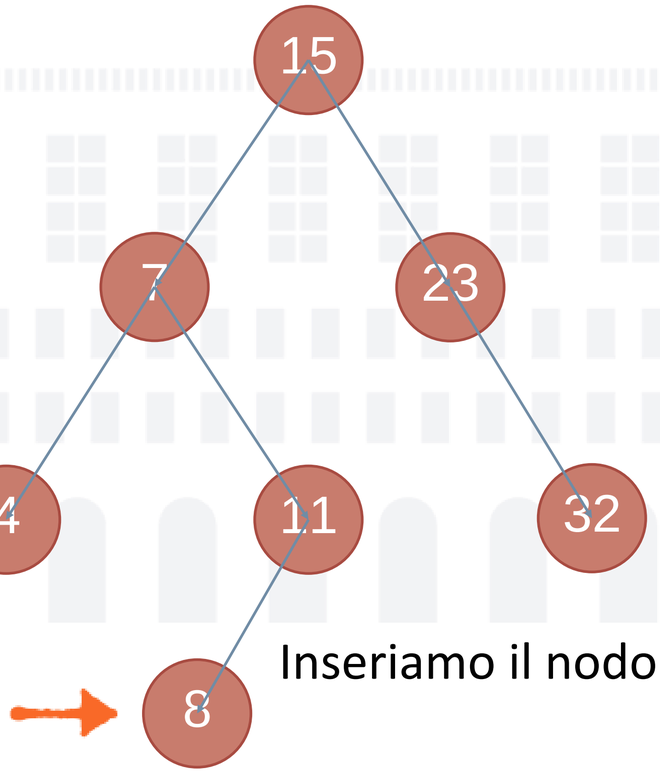
Supponiamo di voler inserire un elemento di chiave 8



8 < 11, ci dovremmo muovere a sinistra,
ma il nodo non ha un figlio sinistro

ALBERO BINARIO: INSERIMENTO

Supponiamo di voler inserire un elemento di chiave 8



Inseriamo il nodo “8” come figlio sinistro di “11”

ALBERI BINARI: PSEUDOCODICE DELL'INSERIMENTO

```
Parametri: Nodo radice, key
if radice is None
   return Nodo(key) # l'albero è vuoto, ritorniamo un nuovo albero
if key < radice.key
   if radice.left is None # figlio sinistro libero per l'inserimento
      radice.left = nuovo_nodo(key)
   else # chiamata ricorsiva sul sottoalbero sinistro
      inserisci(radice.left, key)
else # ovvero key ≥ radice.key
   if radice.right is None # figlio destro libero per l'inserimento
      radice.right = nuovo_nodo(key)
   else # chiamata ricorsiva sul sottoalbero sinistro
      inserisci(radice.right, key)
```

ALBERI BINARI: COMPLESSITA' DELL'INSERIMENTO

Prima di effettuare una chiamata ricorsiva effettuiamo **un numero costante di passi**.

Ogni chiamata ricorsiva ci fa scendere di un livello nell'albero.

Quindi la complessità dell'inserimento dipende, *come per la ricerca*, dalla profondità dell'albero: $\boldsymbol{O(h)}$

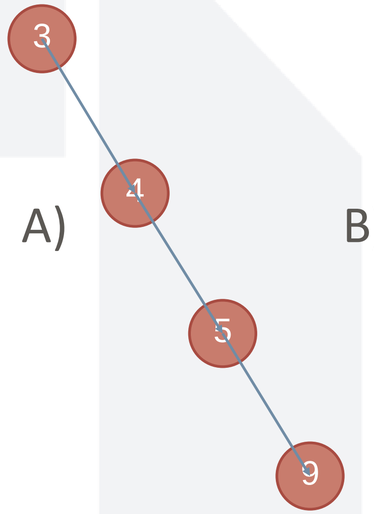
QUIZ: INSERIMENTO

Supponiamo di inserire in un albero inizialmente vuoto i seguenti valori nell'ordine in cui appaiono:

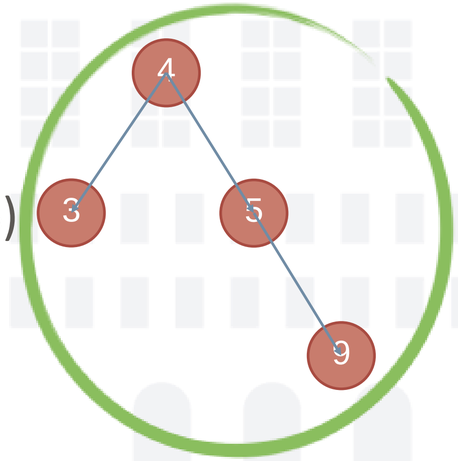
4 5 9 3

Quale di questi è l'albero risultante?

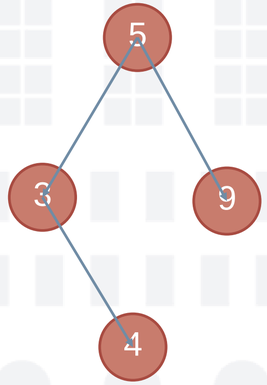
A)



B)



C)



D)

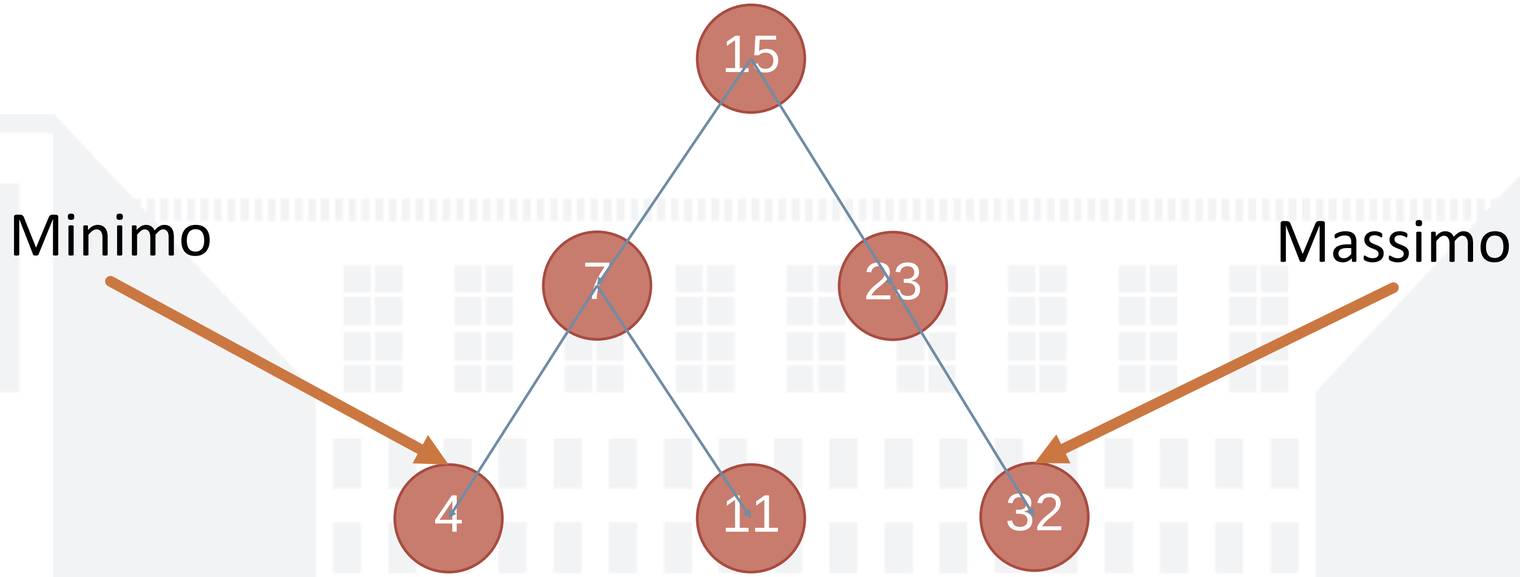
ALBERI BINARI: MASSIMO E MINIMO

Per definizione il **massimo** sarà il nodo “più a destra” nell’albero.

È sufficiente muoversi dalla radice verso destra fino a quando si incontra un nodo senza figlio destro: il valore che contiene è il massimo.

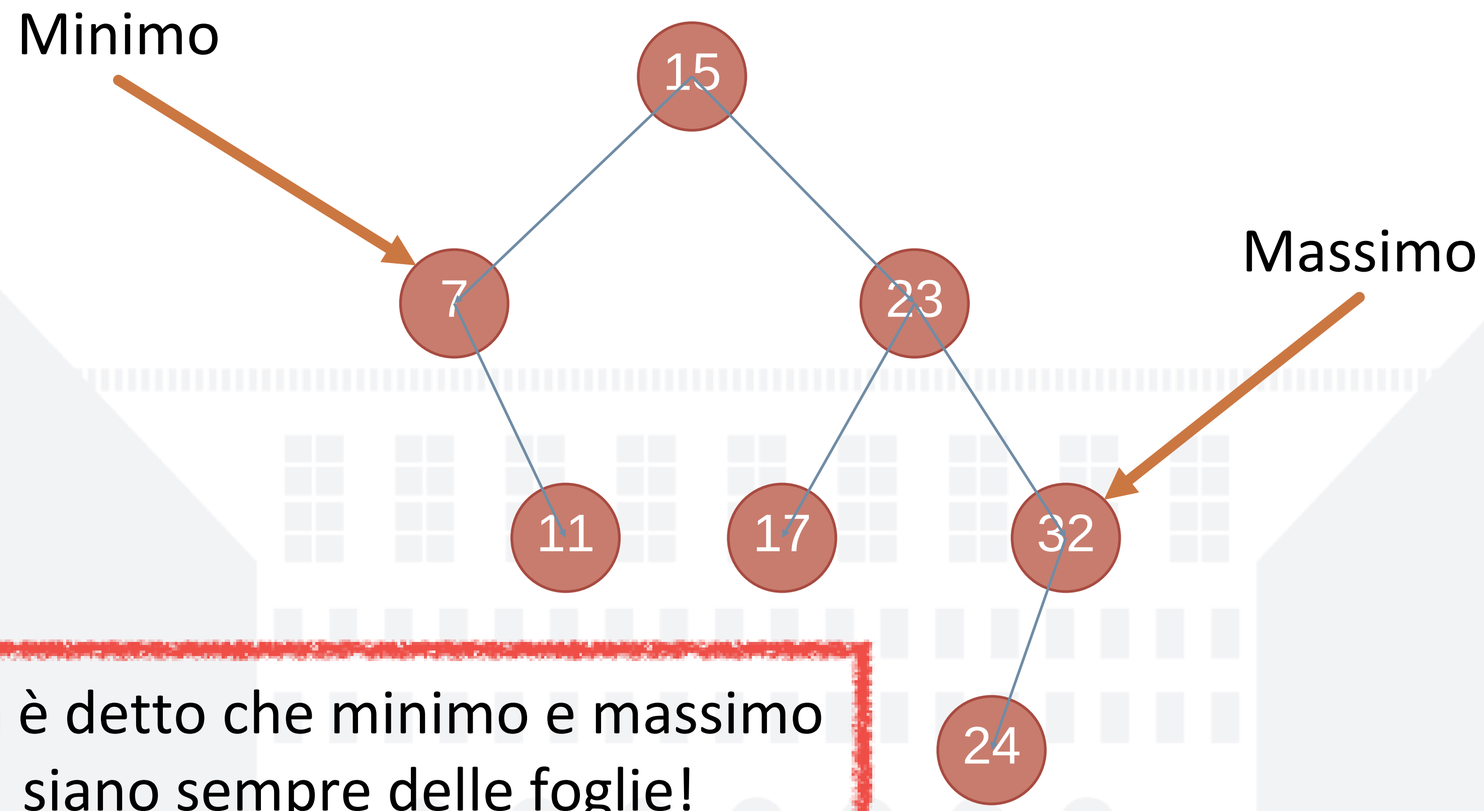
Simmetricamente per il **minimo**: è sufficiente continuare a spostarsi verso sinistra.

ALBERO BINARIO DI RICERCA



Dato che dobbiamo “scendere” fino alle foglie, nel caso peggiore siamo comunque limitati dall’altezza dell’albero per trovare il minimo o il massimo

ALBERO BINARIO DI RICERCA



Non è detto che minimo e massimo siano sempre delle foglie!

ALBERI BINARI: **IDENTIFICAZIONE DEL SUCCESSORE DI UN NODO**

Ricerca del successore di un nodo, identificato con la sua chiave. La ricerca del predecessore è equivalente e simmetrica.

Caso semplice: il nodo (di chiave k) ha un figlio destro

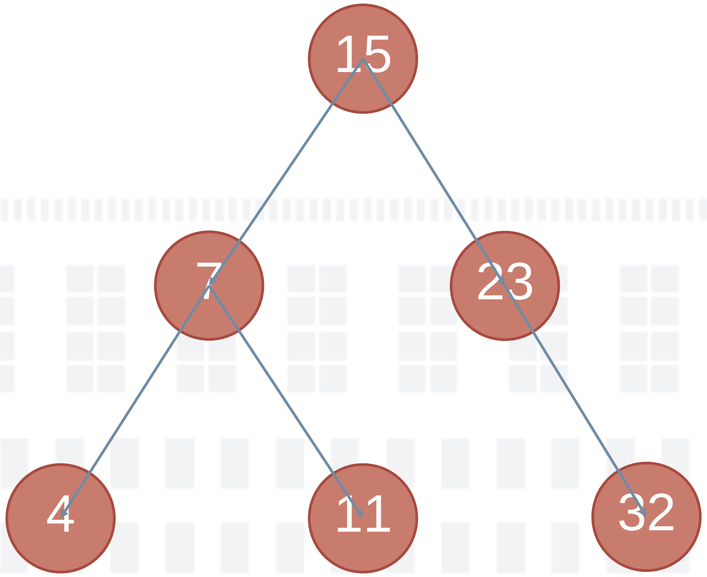
- Allora la chiave più piccola strettamente più grande di k è il minimo del sottoalbero avente radice il figlio destro e quindi sarà il successore di k.

Caso difficile: il nodo non ha figlio destro

- Allora dobbiamo risalire nella catena di genitori

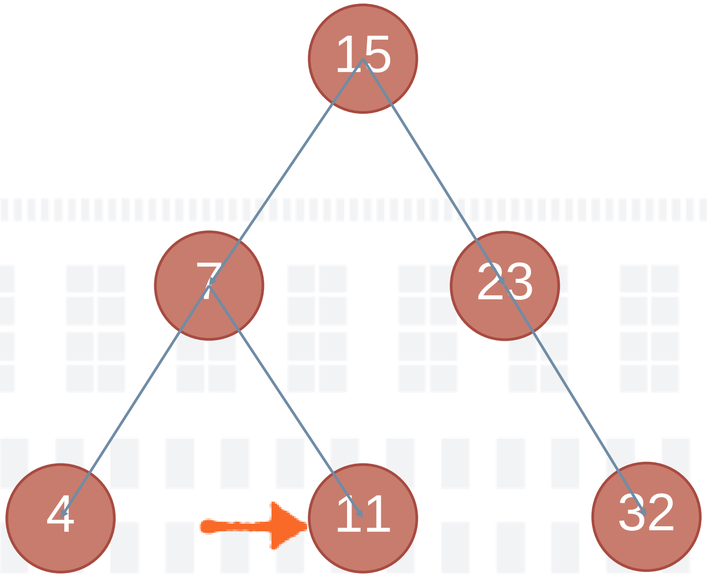
ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di “11”



ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di “11”

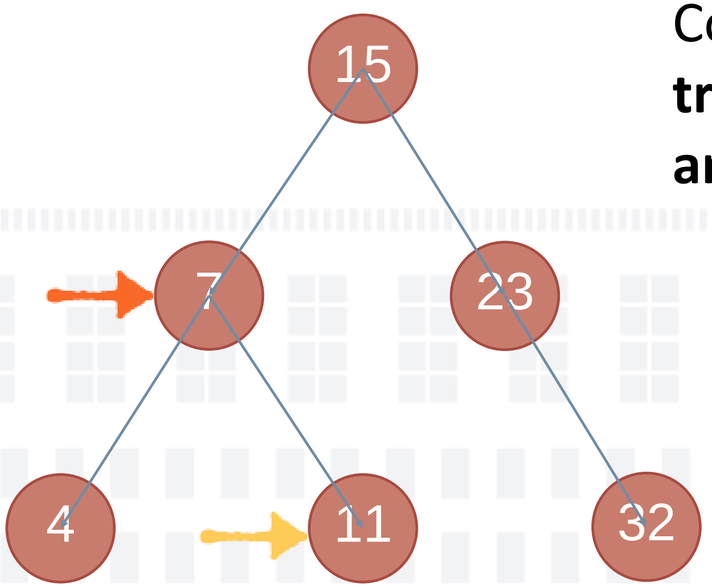


“11” non ha un figlio destro, quindi cerchiamo tra i genitori

ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di “11”

“7” è il genitore di “11”, ma è minore, dato che proveniamo dal figlio destro

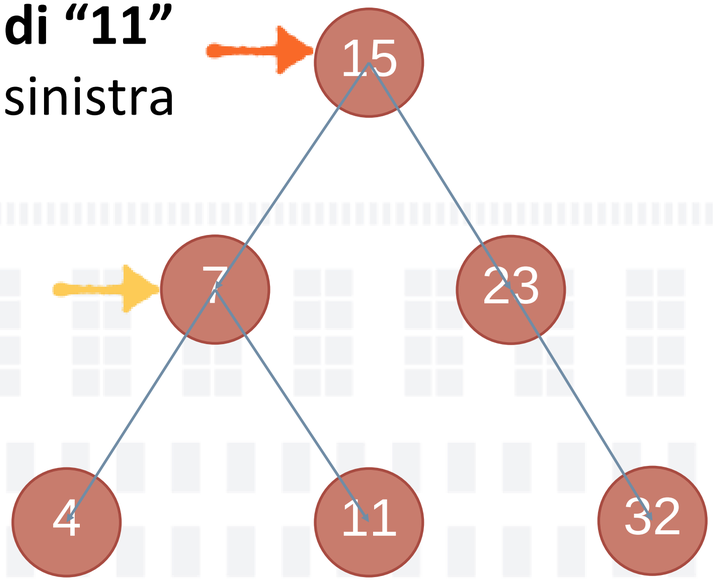


Continuiamo a risalire **finché non troviamo un genitore al quale arriviamo dal figlio di sinistra**

ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di “11”

Abbiamo trovato **il successore di “11”**
dato che arriviamo dal figlio di sinistra



ALBERI BINARI: PSEUDOCODICE DEL SUCCESSORE

```
Parametri: Nodo
if nodo.right is not None:
   return minimo(nodo.right)
x = nodo
y = nodo.parent
while y is not None and y.right is x  # finché esiste un genitore
                                      # e proveniamo dal figlio di destra
   x = x.parent                       # saliamo di un livello
   y = y.parent
return y
```

ALBERI BINARI: SUCCESSORE

La **complessità** di trovare il successore può essere divisa in *due casi*:

- Uguale alla complessità di trovare il minimo, se esiste un figlio destro
- Se il figlio destro non esiste, dobbiamo risalire l'albero… ma il numero di volte che risaliamo è comunque limitato dall'altezza dell'albero.

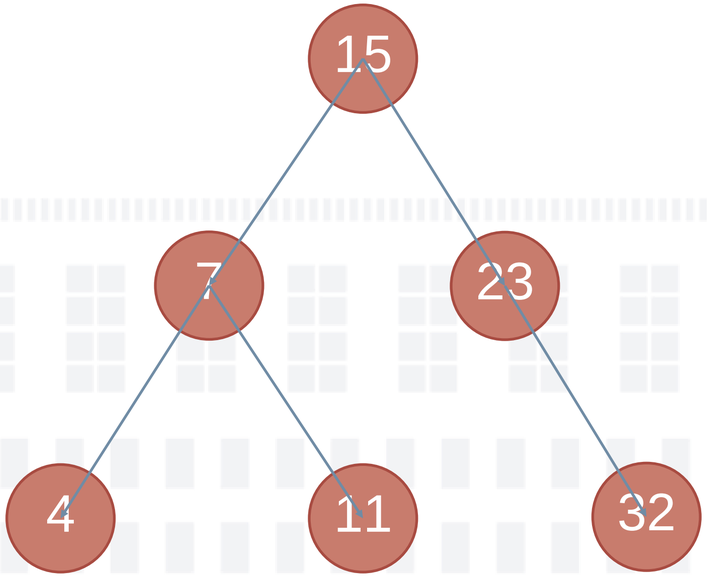
Quindi trovare il successore richiede al più un tempo $\boldsymbol{O(h)}$.

ALBERI BINARI: **CANCELLAZIONE**

- La rimozione di un elemento è l'operazione più complessa
- Abbiamo tre casi a seconda del numero di figli che ha un nodo:
  - Il nodo non ha figli: la rimozione è facile
  - Il nodo ha un solo figlio: facciamo prendere al figlio il posto del genitore
  - Il nodo ha due figli… questo è più complesso

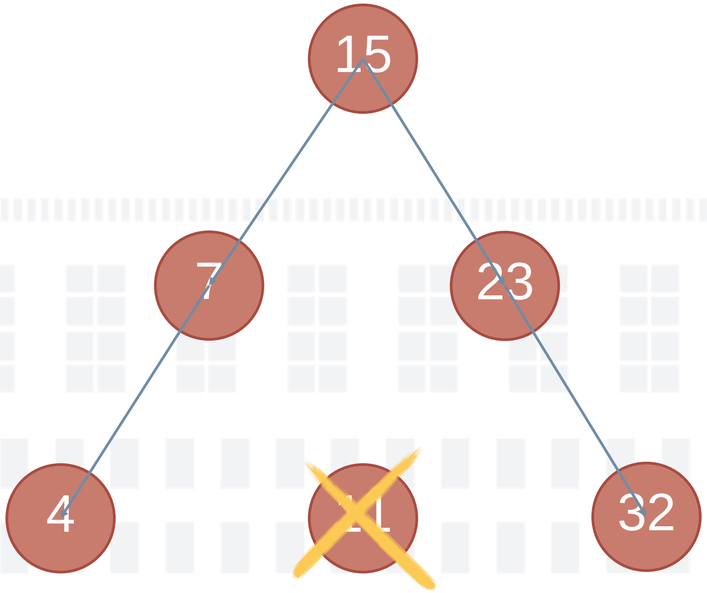
ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare “11”



ALBERO BINARIO: CANCELLAZIONE

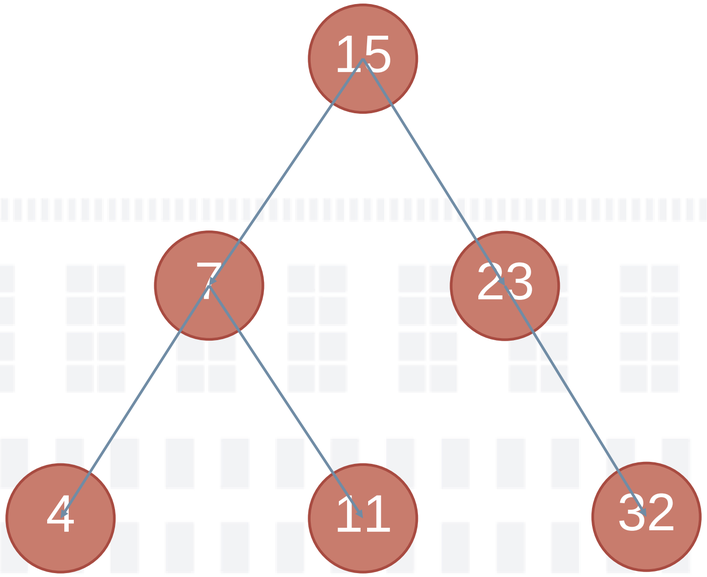
Supponiamo di voler cancellare “11”



“11” Non ha figli, quindi possiamo eliminarlo senza problemi

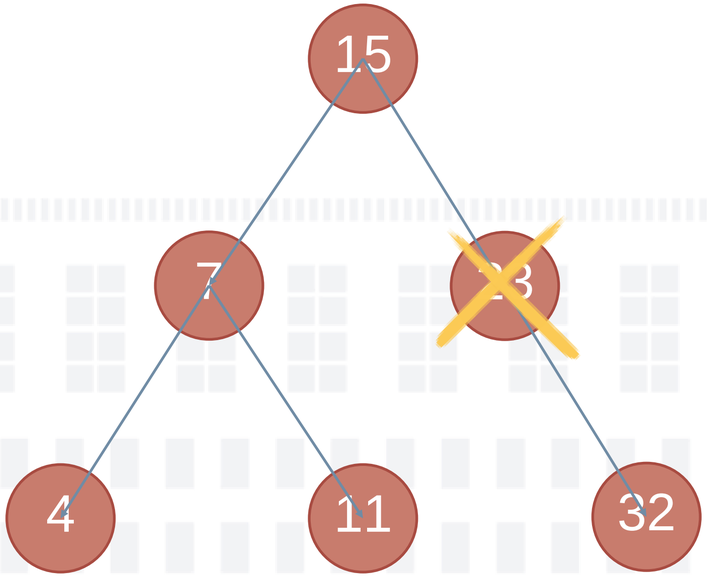
ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare “23”



ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare “23”



Avendo un solo figlio, possiamo rimpiazzare “23”
col suo sottoalbero destro

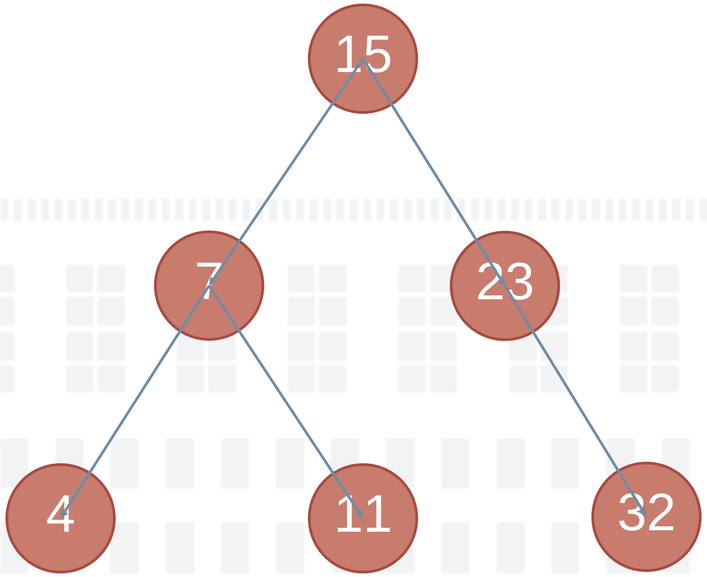
ALBERI BINARI: CANCELLAZIONE

Per cancellare **nel caso di due figli** possiamo dividere la procedura in *due passi*:

1. Mettiamo il successore del nodo da cancellare nel posto del nodo cancellato (*il successore è necessariamente nel sottoalbero destro*)
2. Dato che *il successore* era in una posizione con solamente un figlio (è il minimo del sottoalbero, *non può avere figlio sinistro*), possiamo eliminarlo senza problemi (equivalentemente possiamo usare il predecessore)

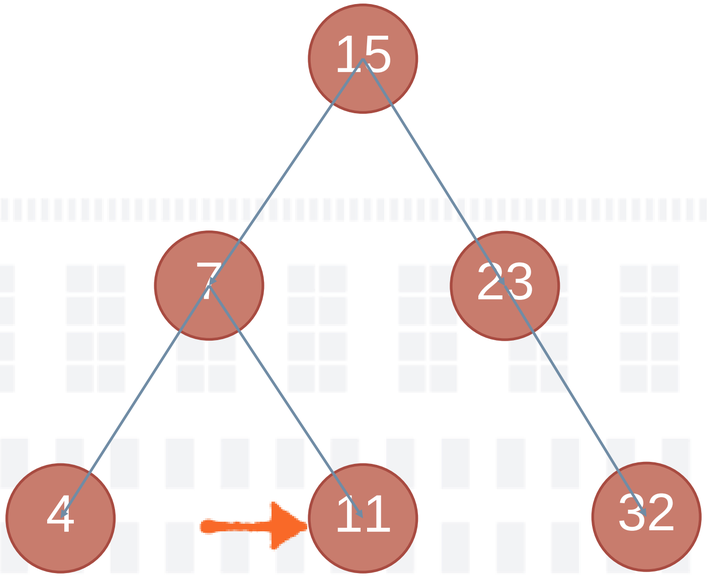
ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare “15”



ALBERO BINARIO: CANCELLAZIONE

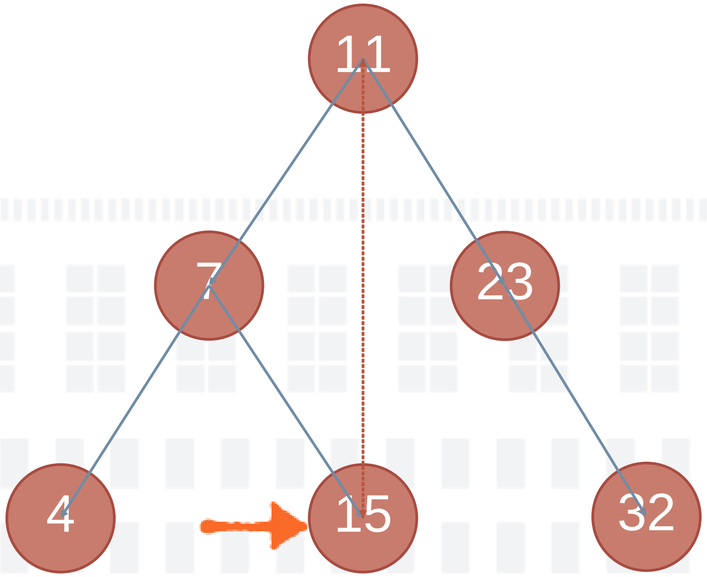
Supponiamo di voler cancellare “15”



Individuiamo **il predecessore di “15”**
(sarebbe identico usando il successore)

ALBERO BINARIO: CANCELLAZIONE

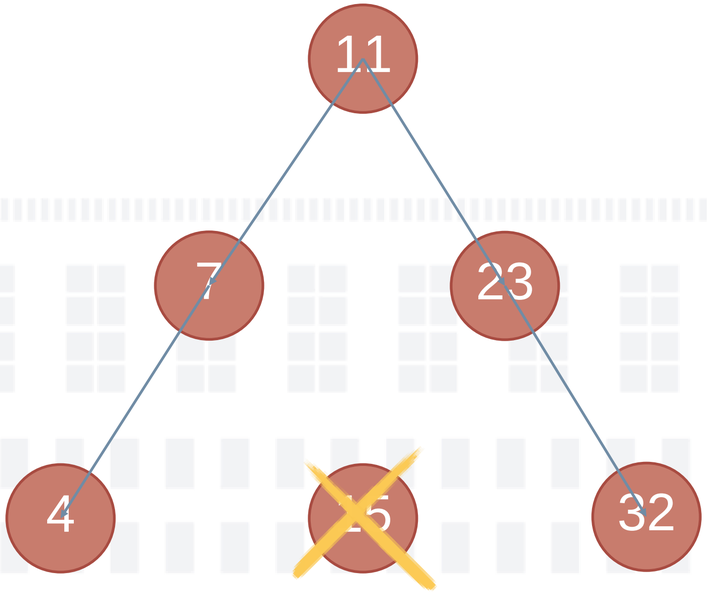
Supponiamo di voler cancellare “15”



Scambiamo di posto il nodo da cancellare ed il predecessore

ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare “15”



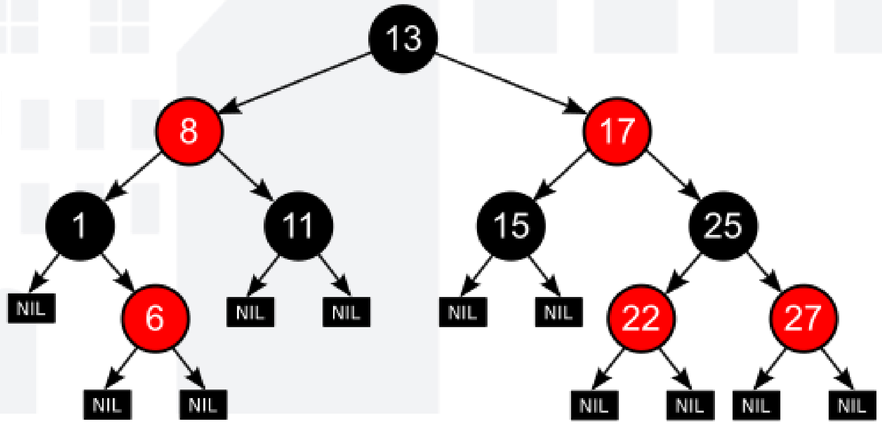
Eliminiamo il nodo, cosa possibile dato che ha zero o un figlio

ALBERI BINARI: CANCELLAZIONE

- La **complessità** della cancellazione dipende dal tipo di rimozione che dobbiamo fare:
  - Rimuovere un nodo con zero o un figlio richiede tempo costante
  - Rimuovere un nodo con due figli richiede di trovare il nodo successore, che richiede tempo $O(h)$
- Anche in questo caso la complessità dipende dall'altezza

ALBERI BINARI: DISCUSSIONE

- Tutte le operazioni viste dipendono dall'altezza dell'albero
- A seconda dell'ordine in cui vengono inseriti i dati è possibile ottenere grandi differenze in termini di altezza:
  - Alberi bilanciati a profondità $\boldsymbol{O(logn)}$
  - Ma nei casi più sbilanciati la profondità è $\boldsymbol{O(n)}$
- **Esistono varianti di alberi in grado di mantenere il bilanciamento** (es., alberi rosso-neri)



ALBERI BINARI: IMPLEMENTAZIONE (IN MEMORIA)

Lista concatenata doppia

Ogni nodo ha **tre campi**:

- Dato
- Puntatore al figlio sinistro
- Puntatore al figlio destro

- I nodi sono collegati tramite puntatori, **non contigui** come un array.

Alternative (meno usate):

- **Array**: memorizzando i nodi in posizioni calcolate (es. heap-style: sinistro = 2i, destro = 2i+1).
- **Hash table**: per rappresentazioni non strutturate.

IN SINTESI: COMPLESSITA' PER BST

OPERAZIONE	CASO PEGGIORE	CASO MEDIO	*Caso medio con albero bilanciato*
ACCEDERE AD UN ELEMENTO	$O(n)$	$O(h)$	$O(logn)$
RICERCA	$O(n)$	$O(h)$	$O(logn)$
INSERIMENTO	$O(n)$	$O(h)$	$O(logn)$
RIMOZIONE	$O(n)$	$O(h)$	$O(logn)$
SPAZIO OCCUPATO	$O(n)$	$O(n)$	$O(n)$

Nota. Lo spazio occupato in realtà sarebbe 3 × n, dove n = spazio occupato dal dato da memorizzare nel nodo + 2 puntatori ai due figli + riferimento al parent (opzionale). Tuttavia, rimane comunque O(n). *Come mai?*

Materiale per la lezione

- Cormen et al. **CAP. 3.12**