DIZIONARI
ALBERI BINARI

INFORMATICA

DUE PROBLEMI DI DATA SCIENCE

Collezione di dati totalmente ordinabili

1) NEAREST NEIGHBOUR

2) PERCENTILI

DYNAMIC SET / DIZIONARI

- A volte le operazioni che vogliamo fare sono più complesse di quelle consentite da stack e code
- Supponiamo di avere un insieme di valori, o di coppie chiave e valore e di volerli organizzare in un insieme che però possiamo modificare dinamicamente.
- Esempi di coppie chiave valore (k, v):
  studenti (chiave: matricola),
  automobili (chiave: targa),
  persone (chiave: codice fiscale)

DYNAMIC SET / DIZIONARI

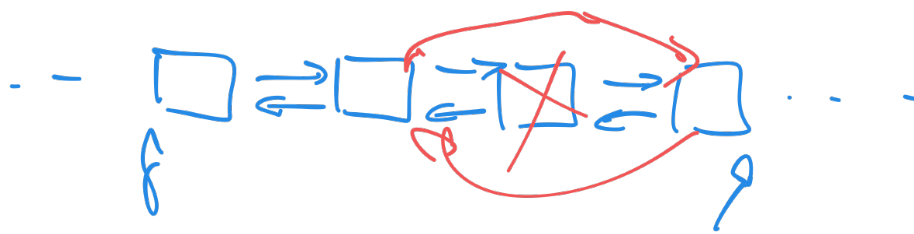
- Le operazioni di base dei dizionari / insiemi dinamici sono:
  - **Inserimento** (*insert*). Inserisce un elemento nel dizionario
  - **Rimozione** (*delete*). Rimuove un elemento dal dizionario
  - **Ricerca** (*search*). Ritorna l'elemento nel dizionario con la chiave fornita come argomento (se esiste)

DYNAMIC SET / DIZIONARI

- Se l'insieme delle chiavi è totalmente ordinato possiamo definire le seguenti operazioni:
  - **Minimo**. Ritorna l'oggetto con la chiave di valore minimo
  - **Massimo**. Ritorna l'oggetto con la chiave di valore massimo
  - **Successore.** Dato un oggetto ritorna il successivo (rispetto al valore della chiave)
  - **Predecessore**. Dato un oggetto ritorna il successivo (rispetto al valore della chiave)

DYNAMIC SET / DIZIONARI

- Vedremo diversi modi di implementare in modo efficiente un dizionario che sono parte di due famiglie:
  - Alberi binari di ricerca
  - Tabelle hash
- Prima però vediamo perché le liste concatenante non sono la scelta migliore



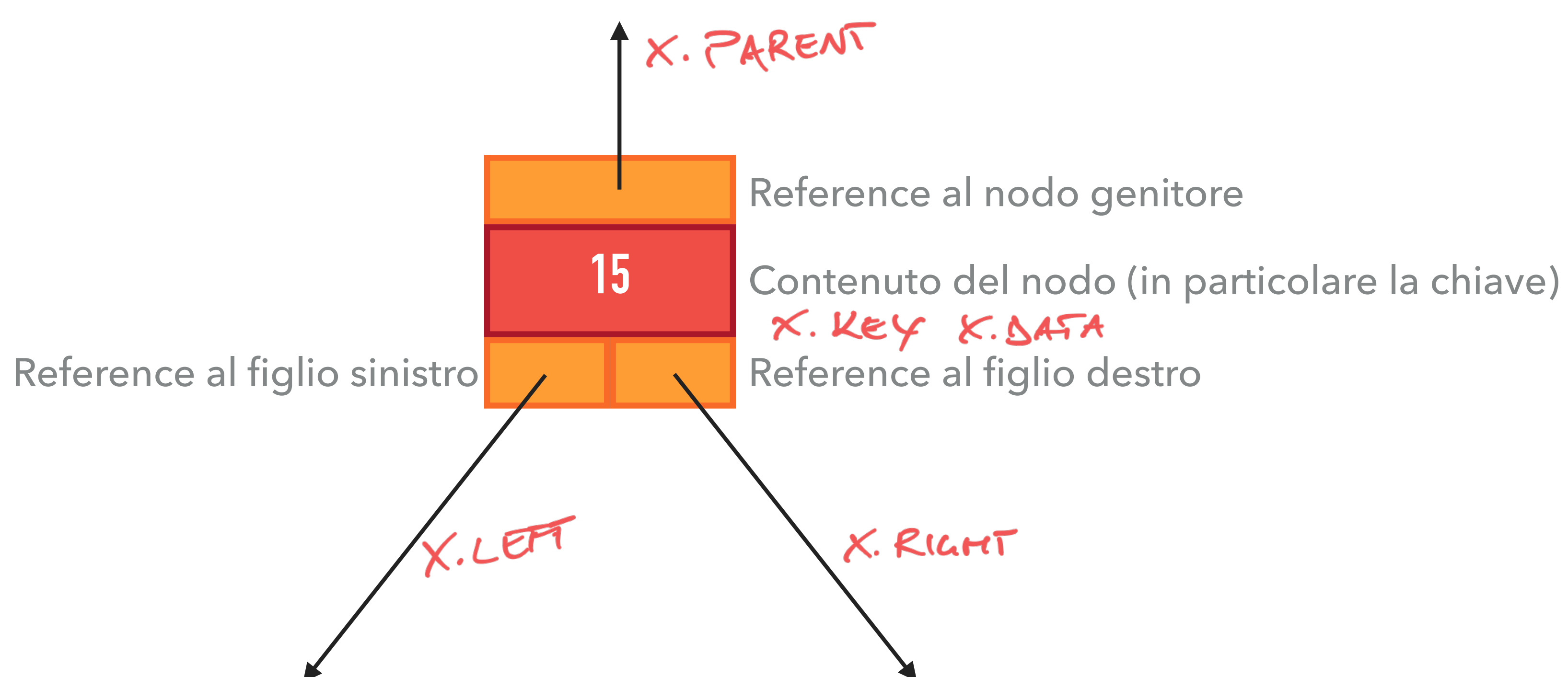
DYNAMIC SET / DIZIONARI CON LISTE CONCATENATE

- L’inserimento può essere effettuato rapidamente in testa o in coda (tempo costante)…
- …ma in quel caso la ricerca richiede di scorrere tutta la lista (tempo lineare)
- Non possiamo migliorare la situazione tenendo la lista ordinata, perché non possiamo effettuare efficientemente la ricerca binaria (l’accesso a posizioni arbitrarie richiede tempo lineare)

ALBERI BINARI: RAPPRESENTAZIONE

- Abbiamo già visto gli alberi binari (heapsort)
- Però per utilizzi "generici" potrebbe essere utile avere una rappresentazione che non si basa su tenere tutto all'interno di un array
- Rappresentiamo quindi ogni nodo come l'oggetto che deve essere contenuto (dato che noi interessa solo la chiave rappresenteremo solo quella) e un insieme di reference ai nodi figli e al nodo genitore

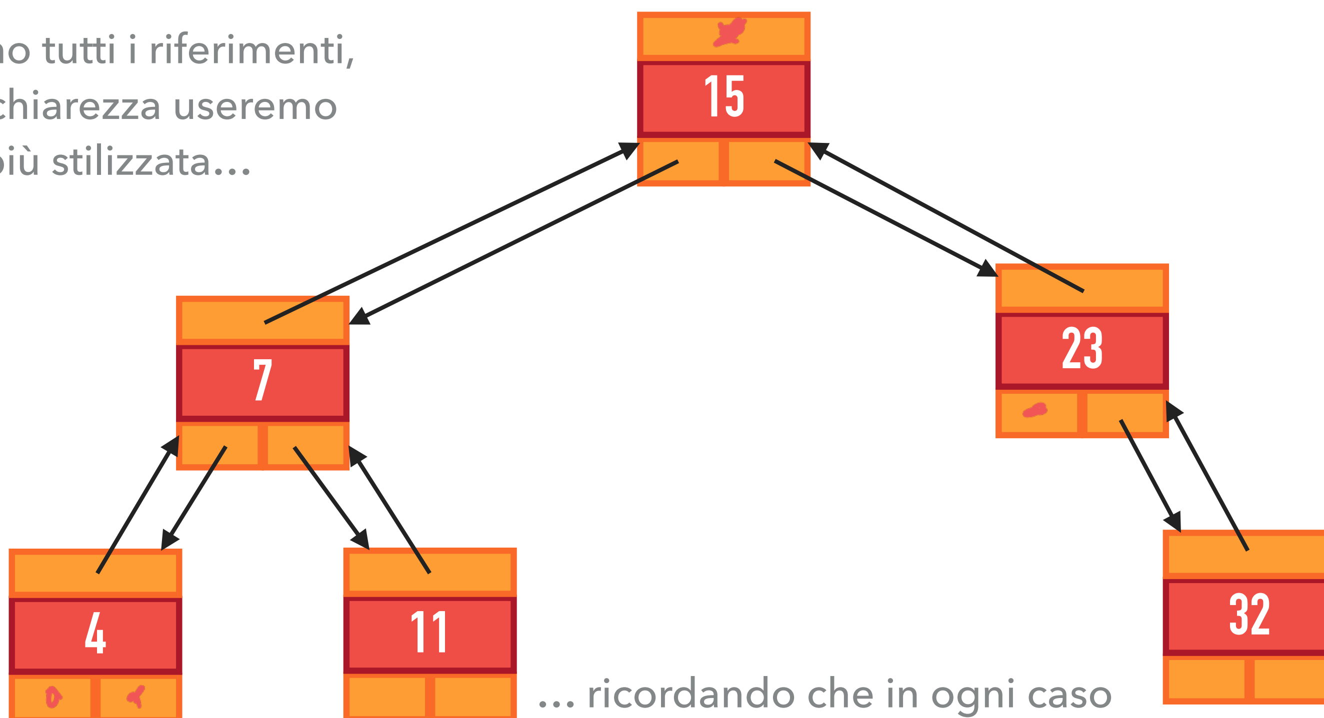
NODO DI UN ALBERO BINARIO



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

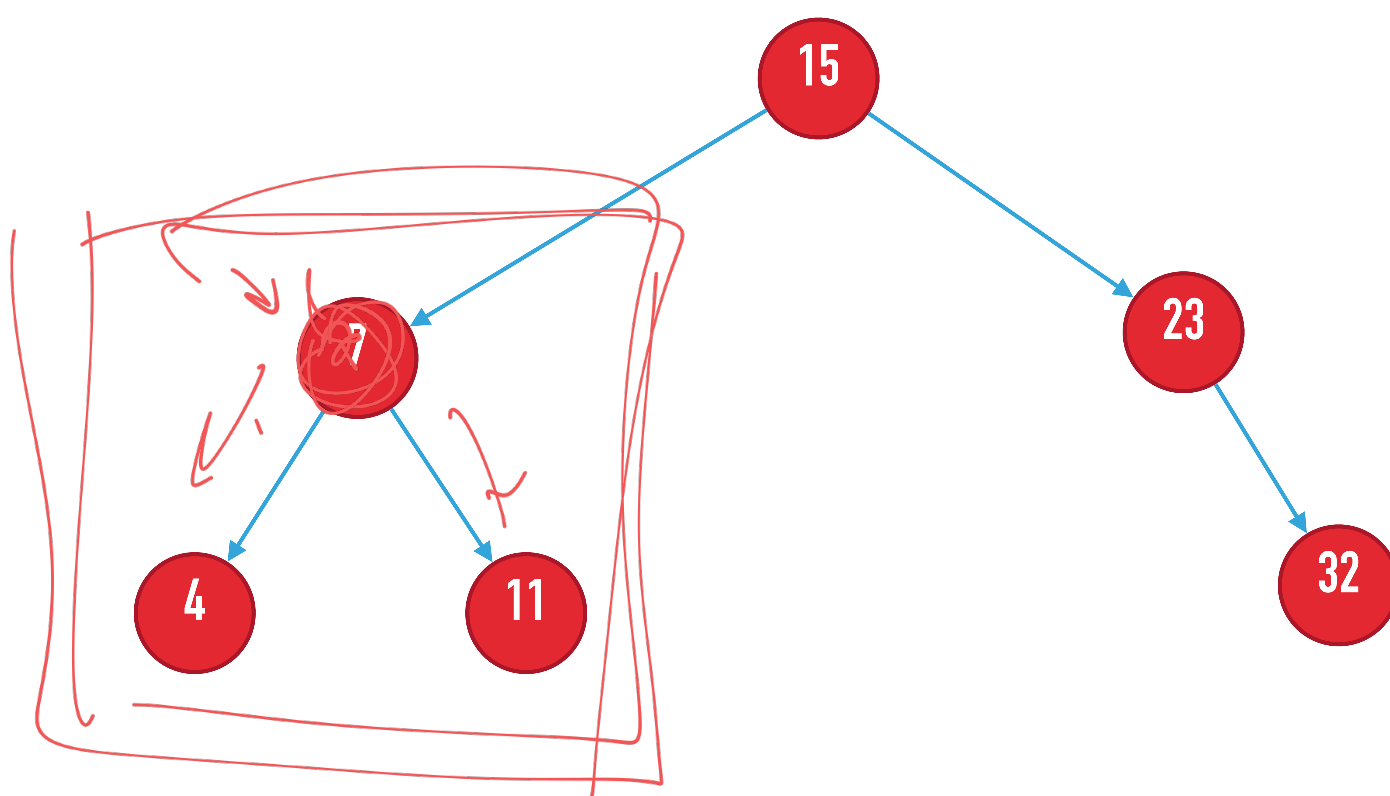
ALBERO BINARIO RAPPRESENTAZIONE GRAFICA

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

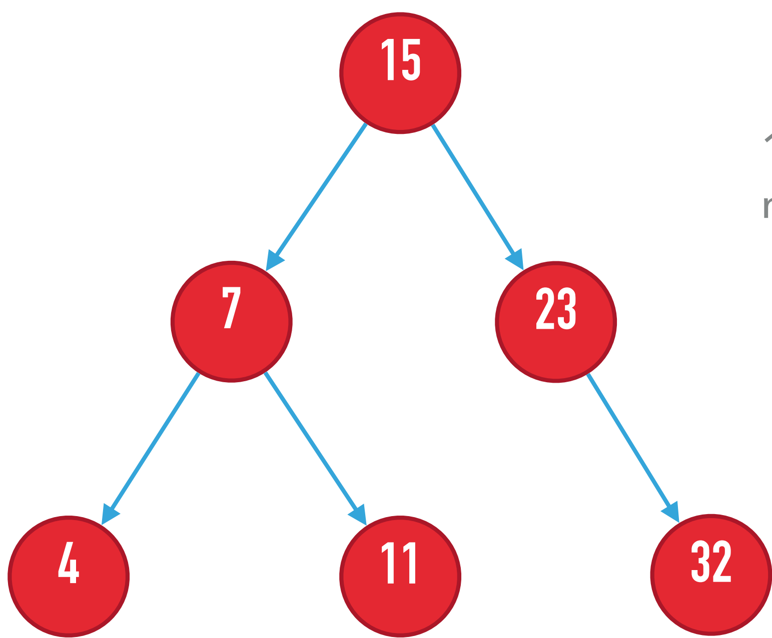
ALBERO BINARIO RAPPRESENTAZIONE GRAFICA



ALBERI BINARI DI RICERCA

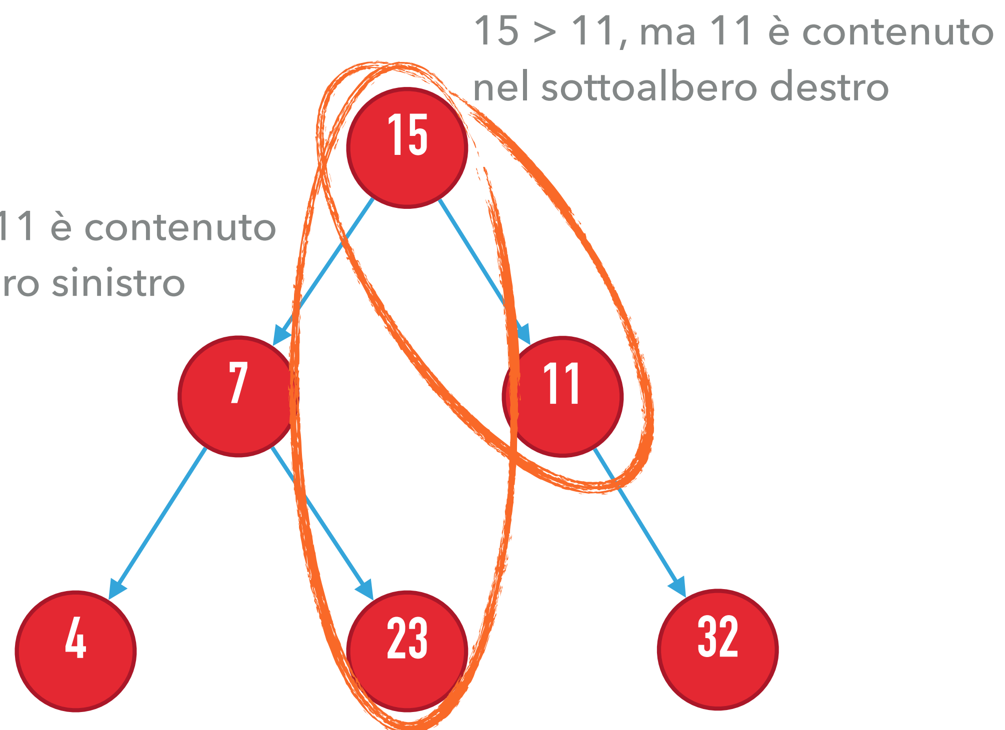
- Un albero binario di ricerca (o binary search tree – BST) richiede chiavi che siano totalmente ordinate (e.g., interi)
- Ogni nodo dell'albero contiene una chiave . NODO n, n.KEY
- Ogni nodo dell'albero ha la seguente proprietà:
  - Tutti i nodi nel sottoalbero sinistro (T(x.left)) hanno chiave *minore* della chiave nel nodo : $\forall x \in T(n.left),\ x.KEY \leq n.KEY$
  - Tutti i nodi del sottoalbero destro hanno chiave *maggiore* della chiave nel nodo : $\forall x \in T(n.right),\ x.KEY \geq n.KEY$

ALBERO BINARIO DI RICERCA



É un albero binario di ricerca

15 < 23, ma 11 è contenuto nel sottoalbero sinistro



NON è un albero binario di ricerca

ALBERI BINARI: RICERCA

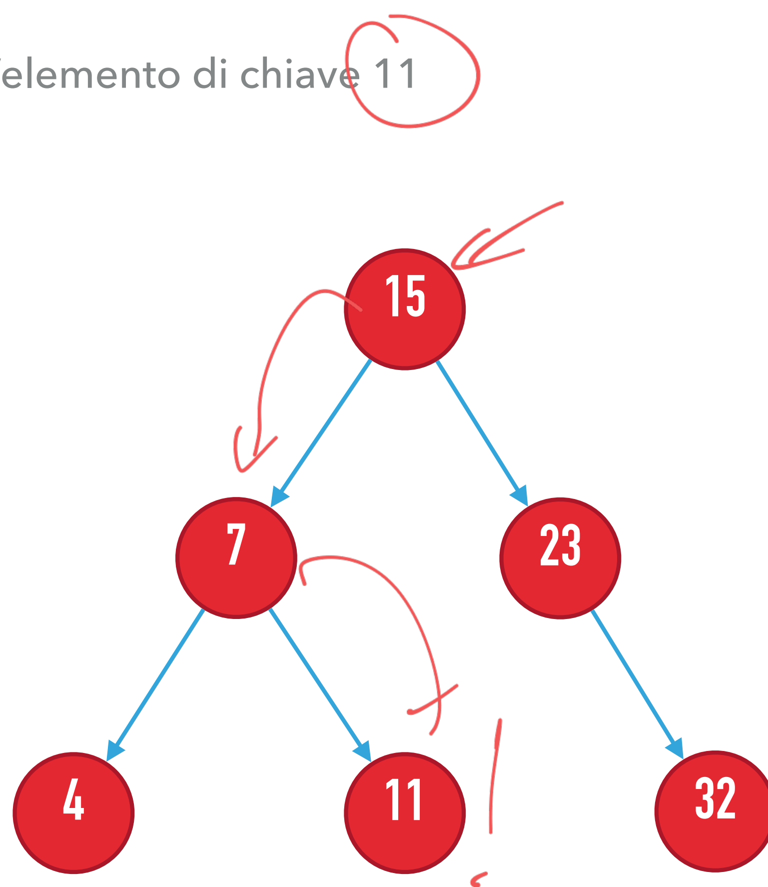
- Dato un albero binario la ricerca può essere suddivisa in quattro casi:
- Albero vuoto: l'elemento non è contenuto
- La chiave della radice è quella che stiamo cercando: abbiamo trovato l'elemento
- La chiave che stiamo cercando è **minore** della chiave nella radice: cerchiamo ricorsivamente nel sottoalbero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

ALBERI BINARI: 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

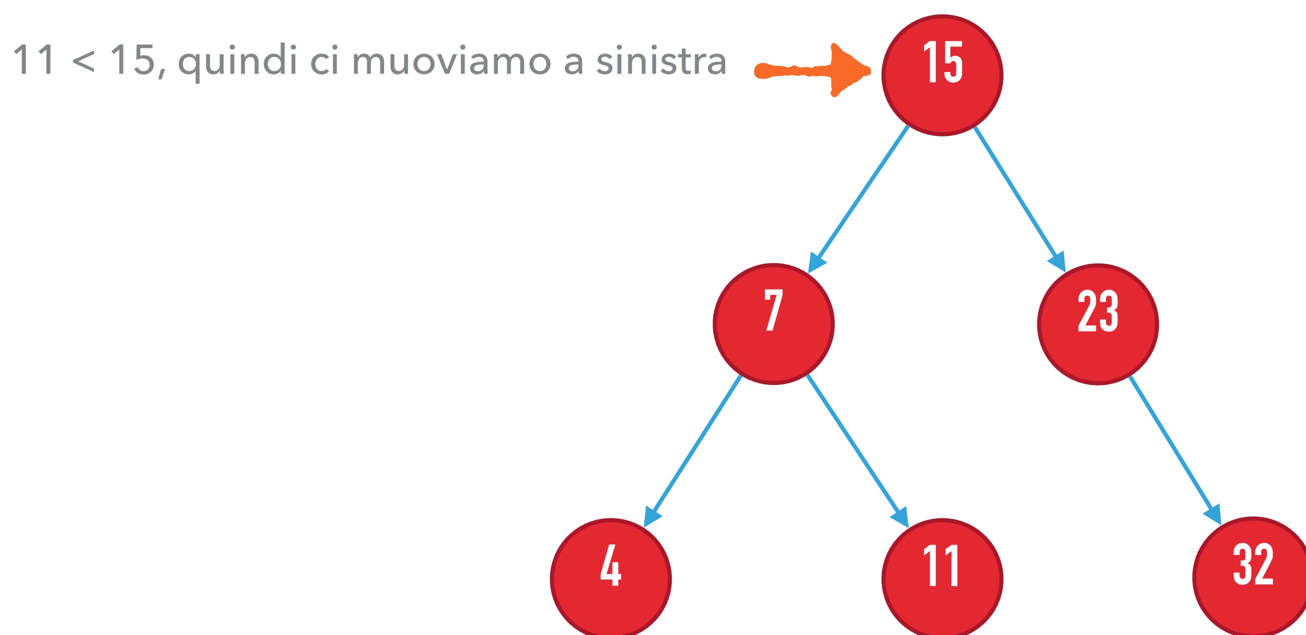
ALBERO BINARIO: RICERCA

Supponiamo di voler trovare l'elemento di chiave 11



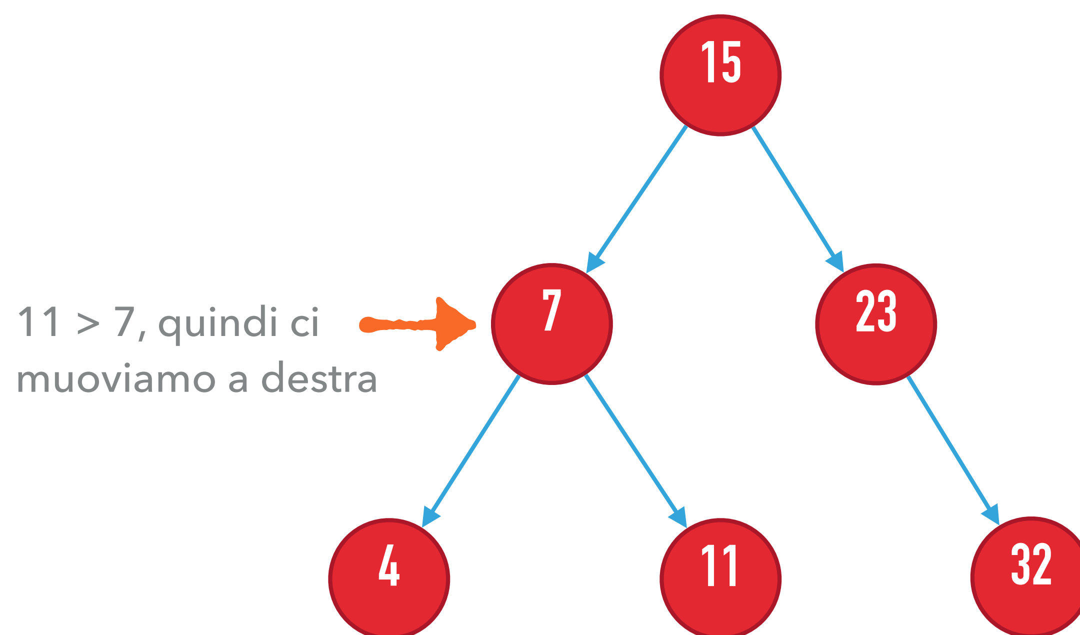
ALBERO BINARIO: RICERCA

Supponiamo di voler trovare l'elemento di chiave 11



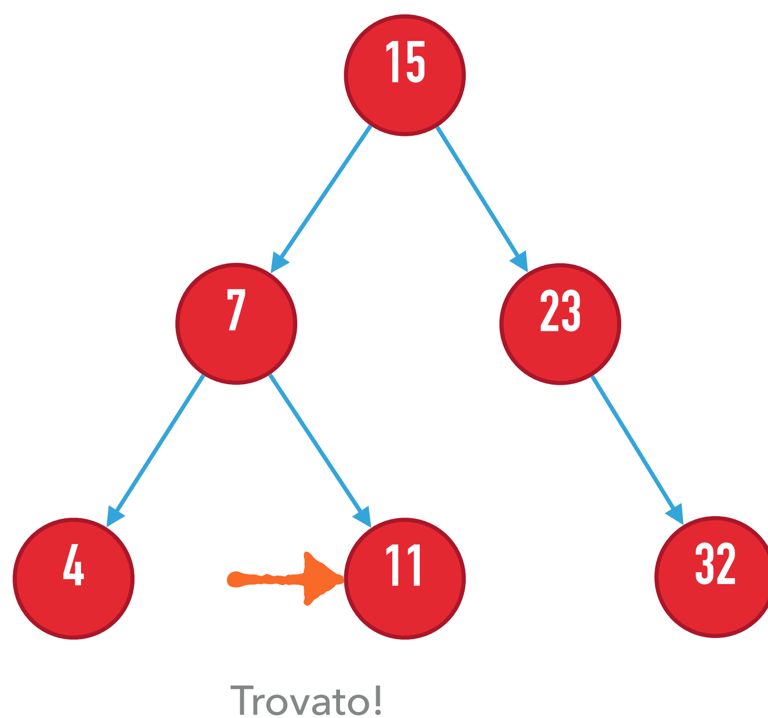
ALBERO BINARIO: RICERCA

Supponiamo di voler trovare l'elemento di chiave 11



ALBERO BINARIO: RICERCA

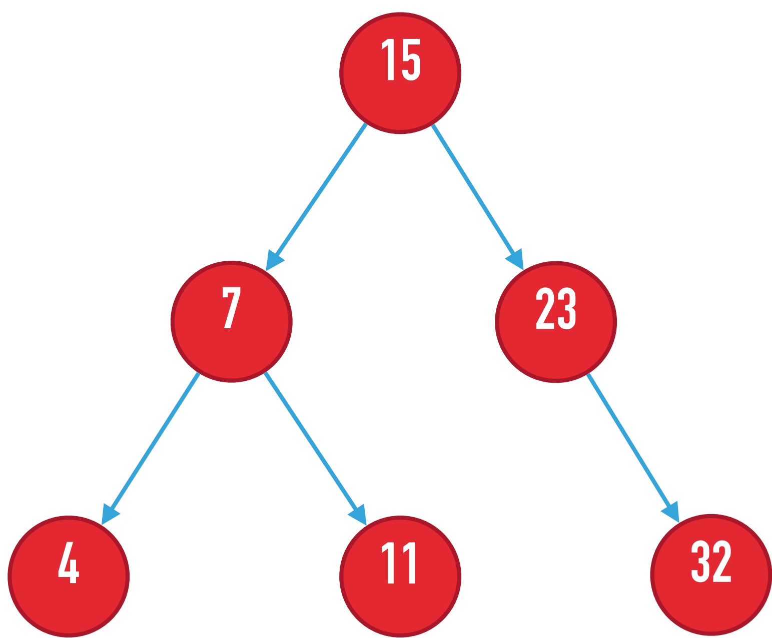
Supponiamo di voler trovare l'elemento di chiave 11



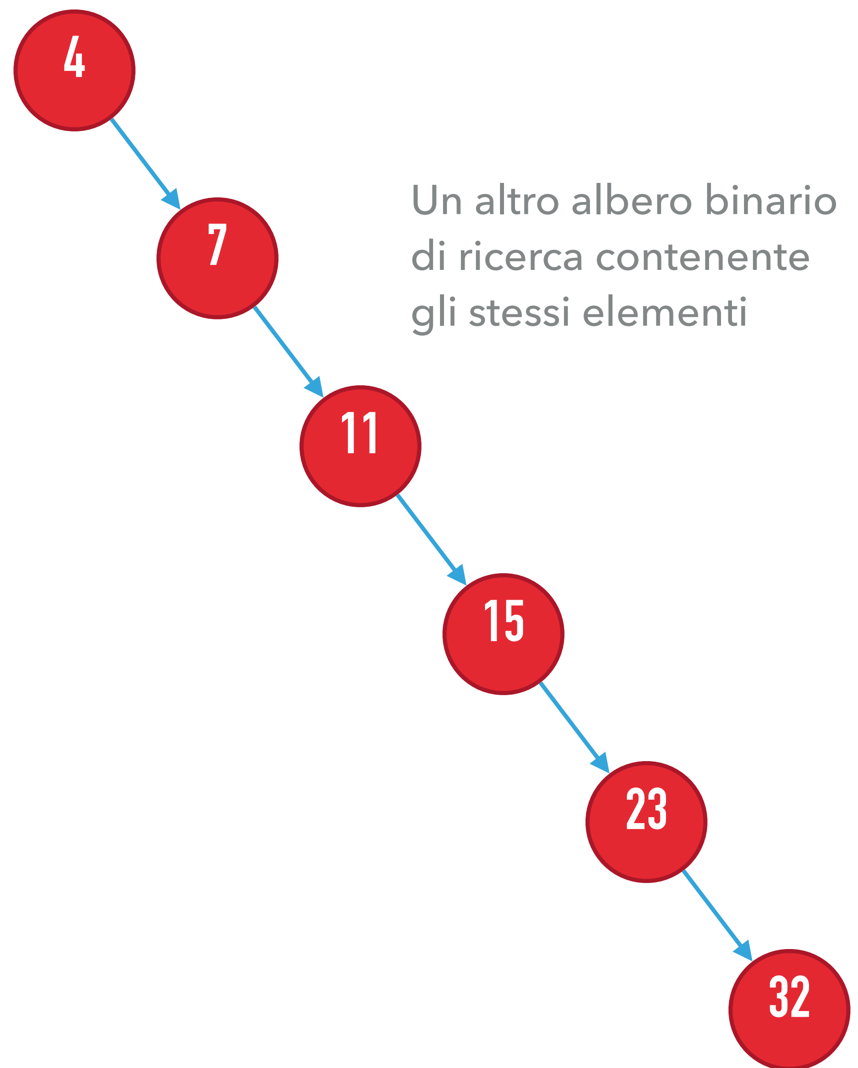
ALBERI BINARI: RICERCA

- Quale è la complessità 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.
- Quindi $O(\log n)$?

ALBERO BINARIO DI RICERCA



Albero binario di ricerca



Un altro albero binario di ricerca contenente gli stessi elementi

ALBERI BILANCIATI E SBILANCIATI

- Un albero binario più essere più o meno sbilanciato
- Nel caso migliore abbiamo un albero con la profondità minima necessaria a contenere tutti gli elementi, quindi la ricerca avviene in tempo $O(\log n)$
- 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 $O(n)$
- Vedremo più avanti metodi per mantenere gli alberi bilanciati

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 del 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

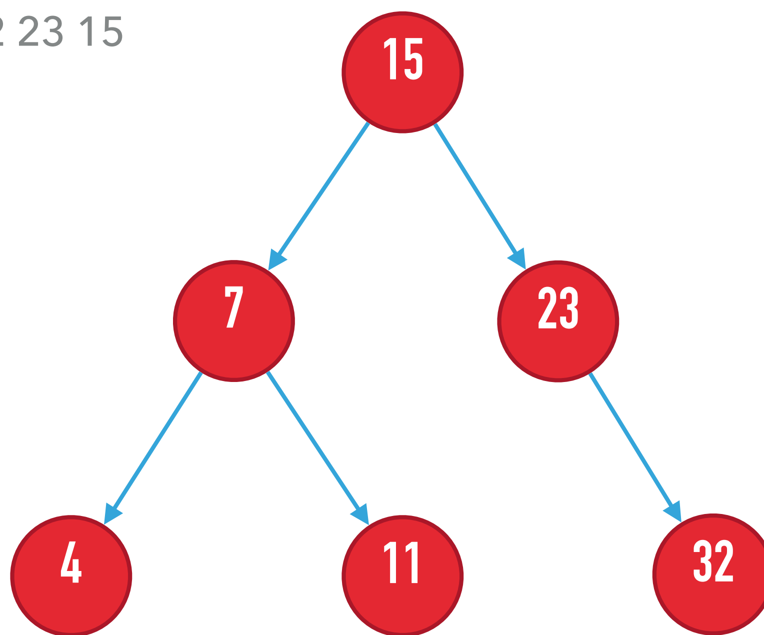
ALBERO BINARIO: VISITE

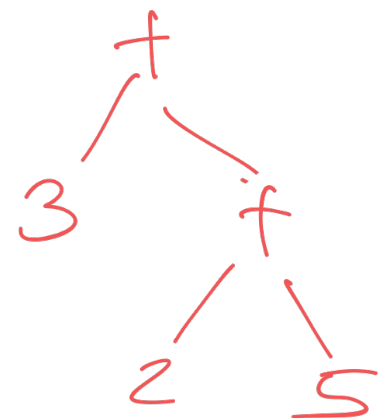
Visita in pre-ordine: 15 7 4 11 23 32

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

Visita in ordine: 4 7 11 15 23 32

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





ALBERI BINARI: INSERIMENTO

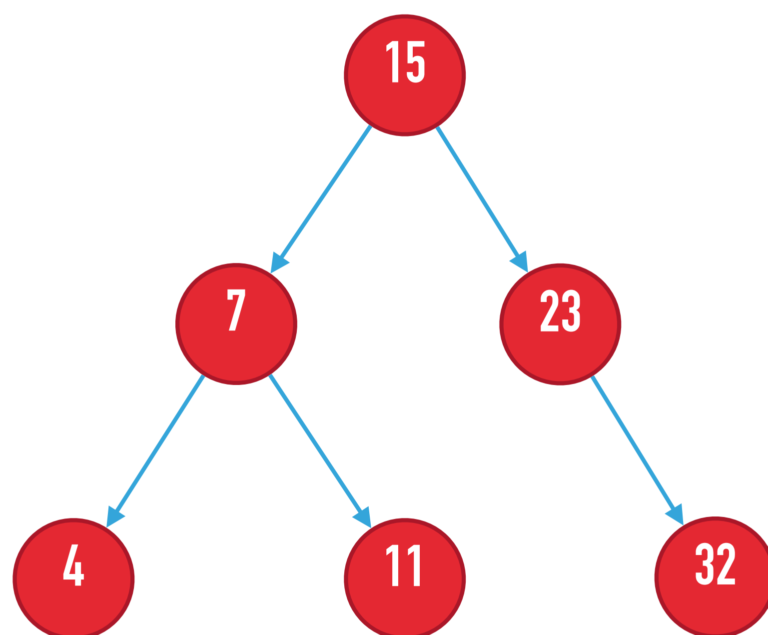
- L’inserimento avviene in modo simile alla ricerca
- Data una chiave k dobbiamo trovare un posto libero per inserire quella chiave **rispettando la proprietà dell’albero binario di ricerca**
- Confrontiamo k con la chiave nella radice:
  - Se k è maggiore, andrà inserita a destra della radice
  - Se k è minore, andrà inserita a sinistra della radice

ALBERI BINARI: INSERIMENTO

- Supponiamo di dover proseguire a sinistra (i.e., 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

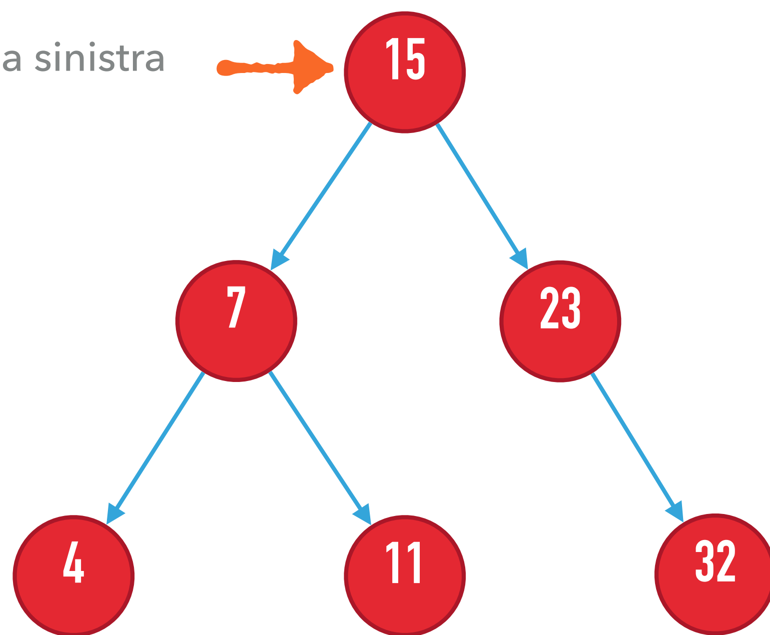
Supponiamo di voler inserire un elemento di chiave 8



ALBERO BINARIO: INSERIMENTO

Supponiamo di voler inserire un elemento di chiave 8

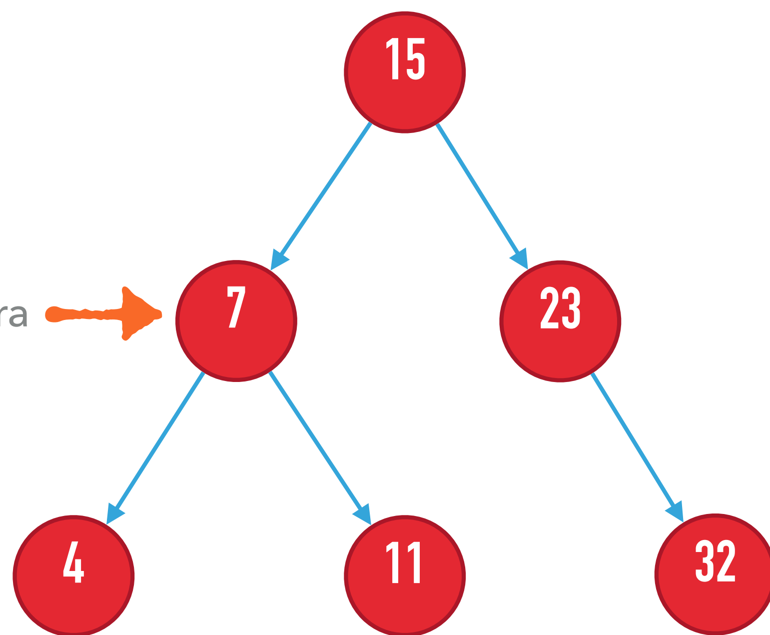
8 < 15, quindi ci muoviamo a sinistra



ALBERO BINARIO: INSERIMENTO

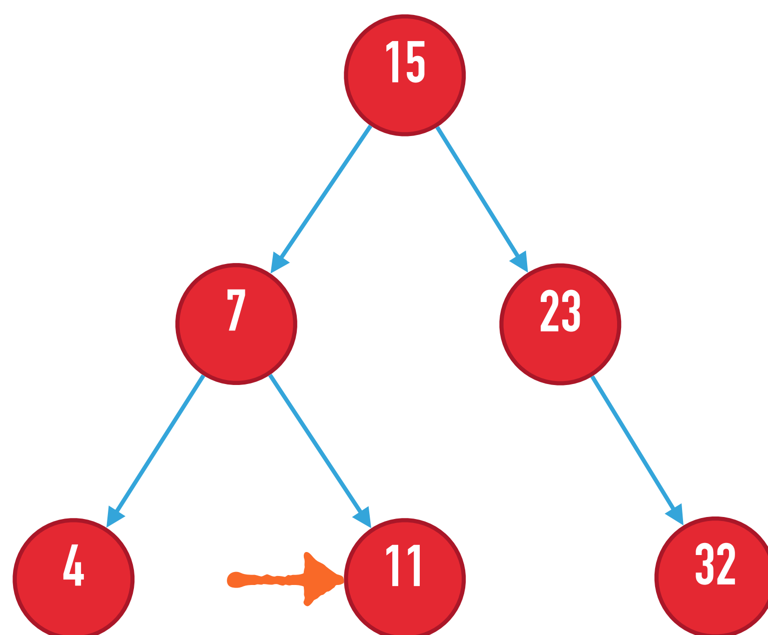
Supponiamo di voler inserire un elemento di chiave 8

8 > 7, quindi ci muoviamo a destra



ALBERO BINARIO: INSERIMENTO

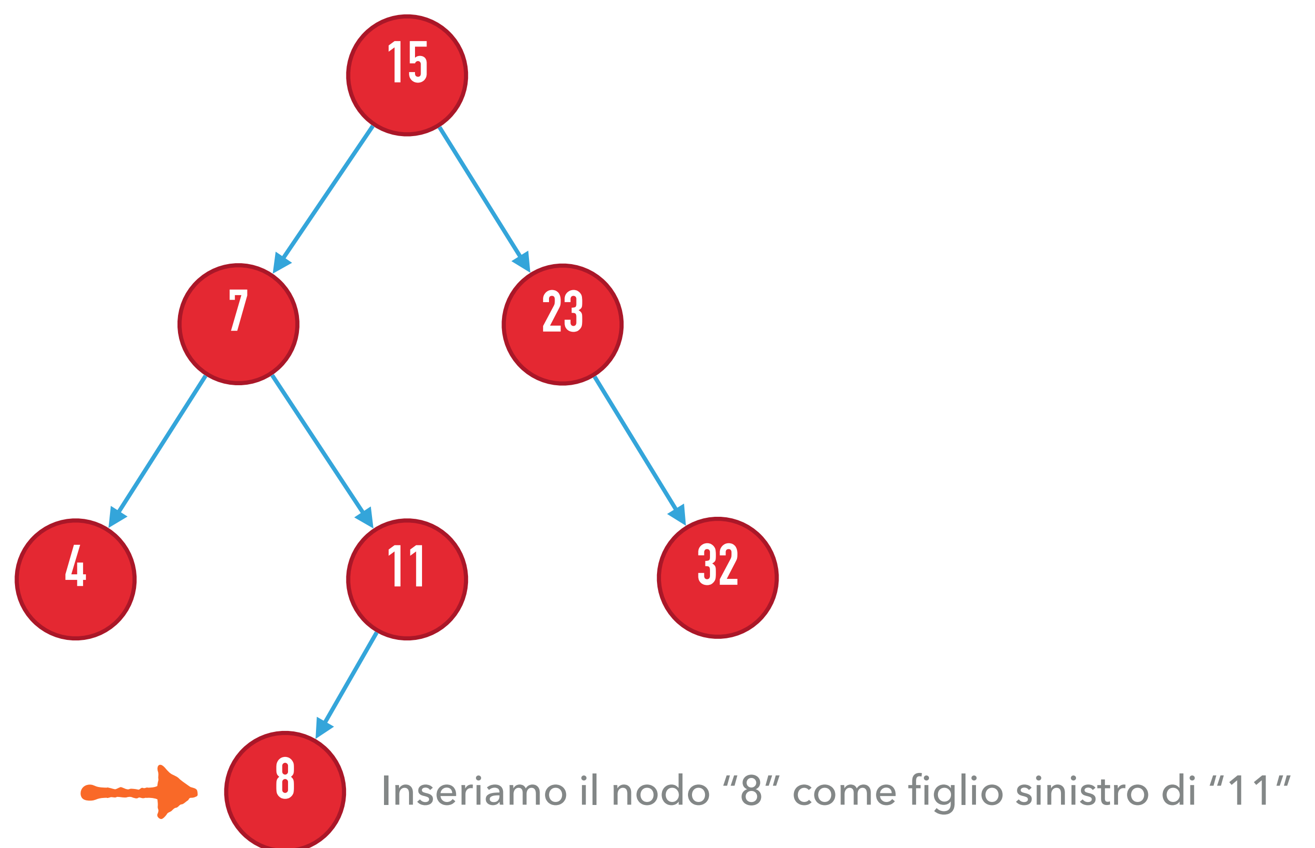
Supponiamo di voler inserire un elemento di chiave 8



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



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: 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: $O(h)$

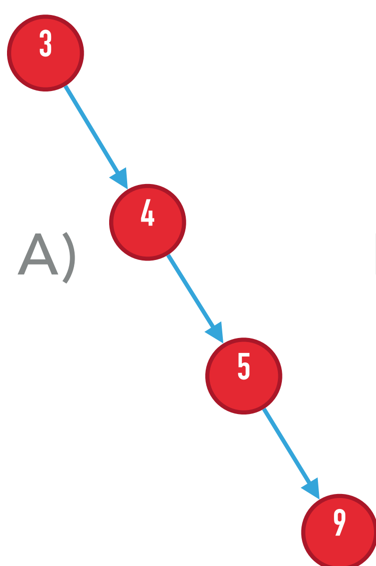
QUIZ: INSERIMENTO

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

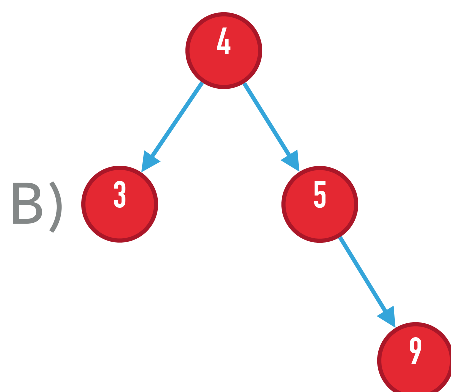
4 5 9 3

Quale di questi è l'albero risultante?

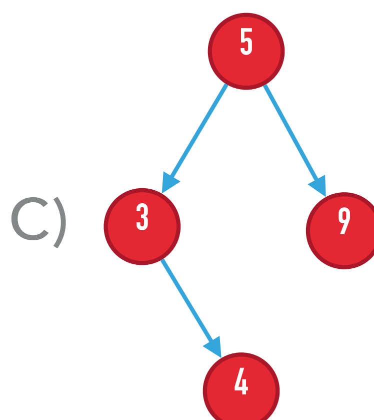
A)



B)



C)

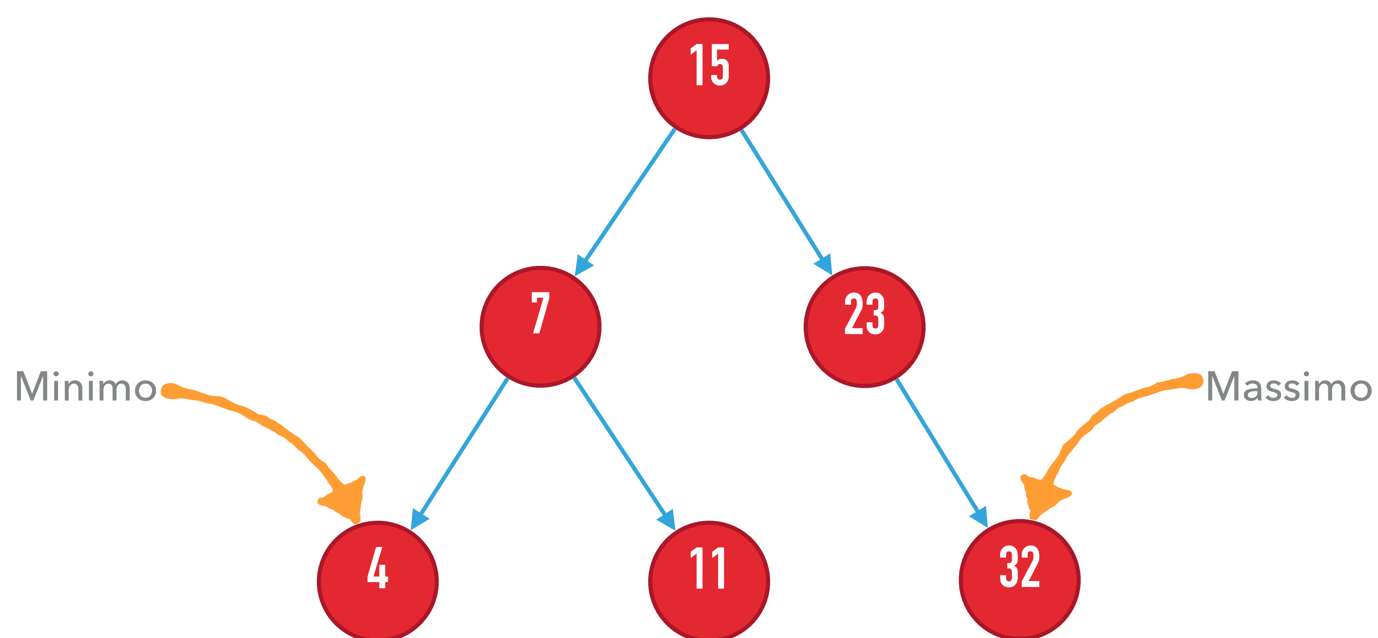


D)

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

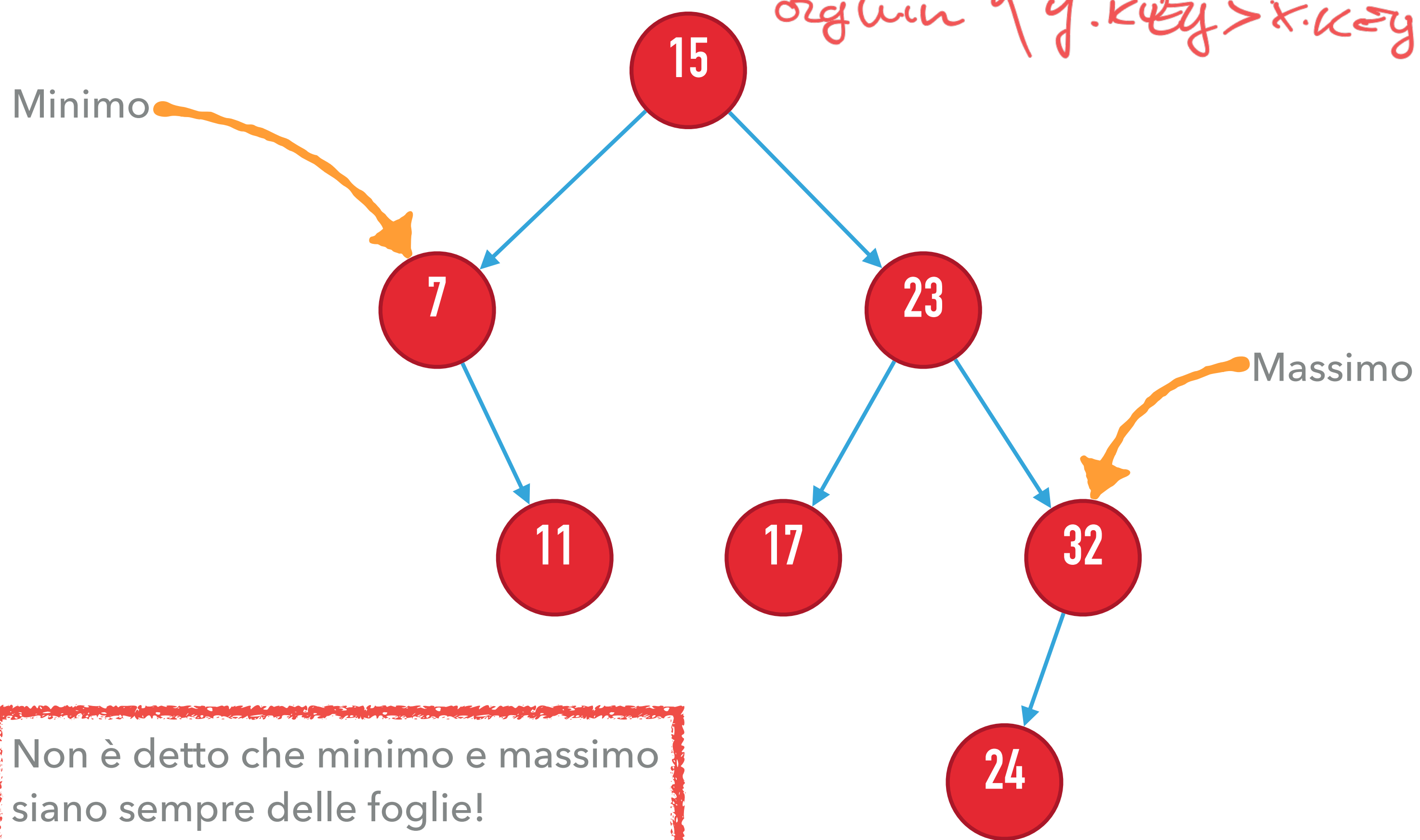


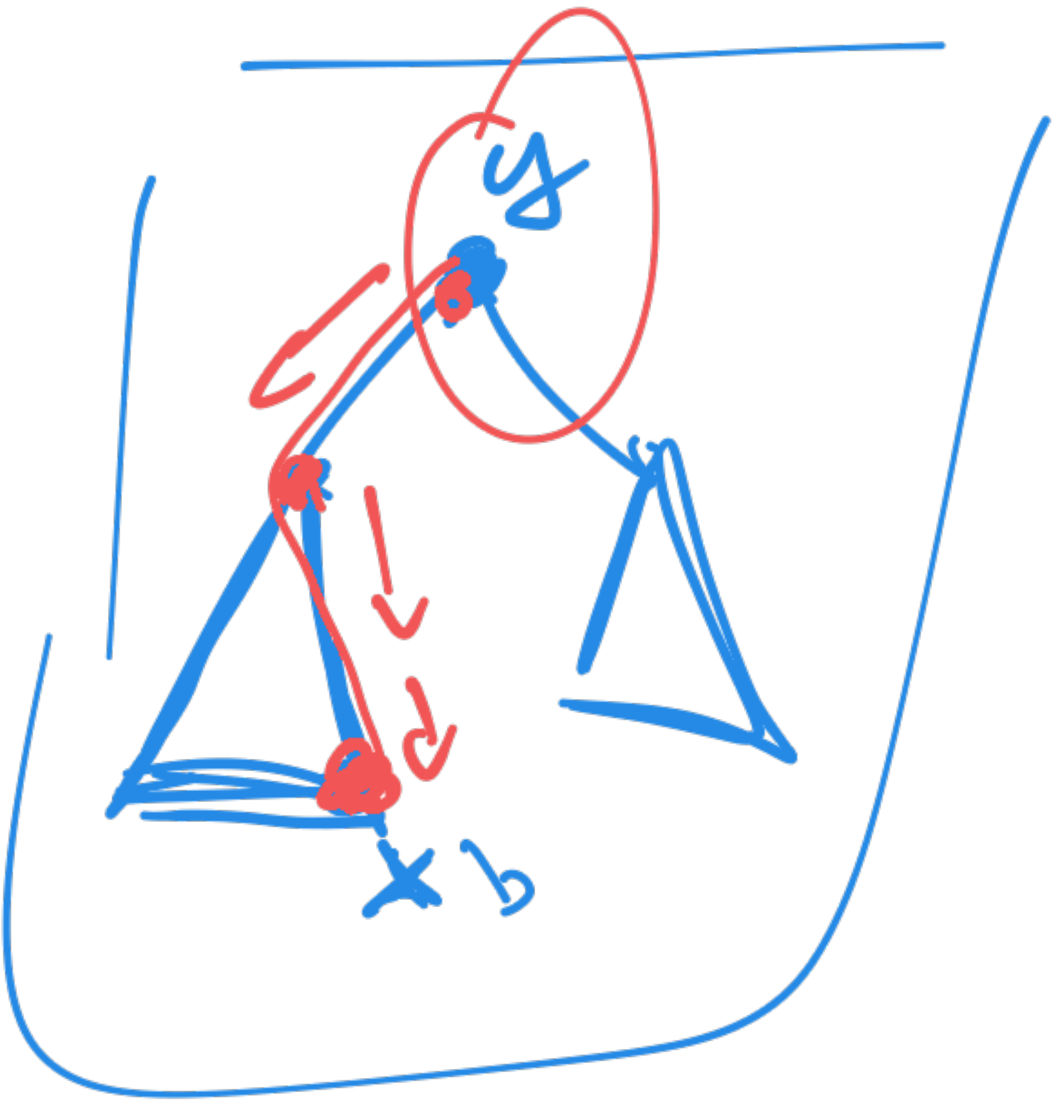
Dato che dobbiamo "scendere" fino alle foglie, nel caso peggiore siamo comunque limitati dall'altezza dell'albero per trovare il minimo o il massimo

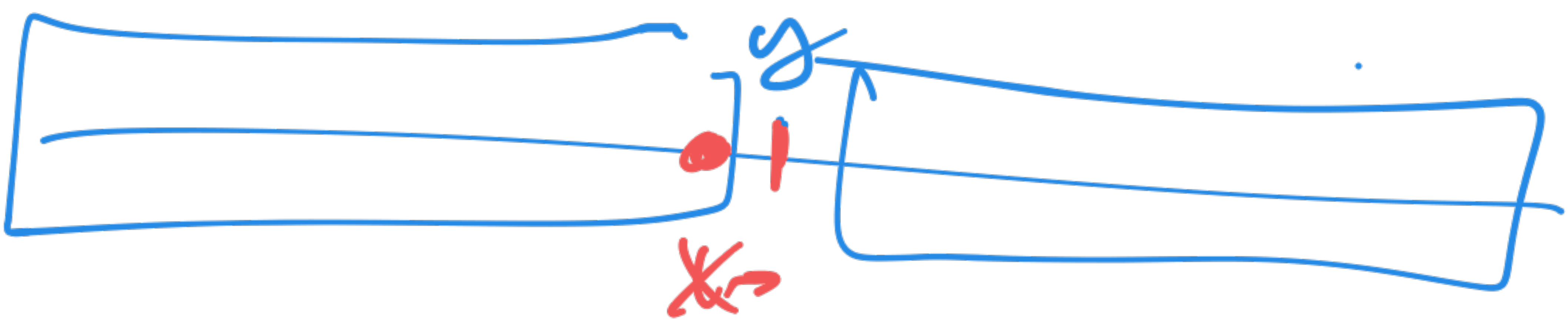
ALBERO BINARIO DI RICERCA

Successore di x, $X = \{x_1 \dots x_n\}$

$\text{argmin} \{ y.key > x.key \mid y \in X \}$




y
b

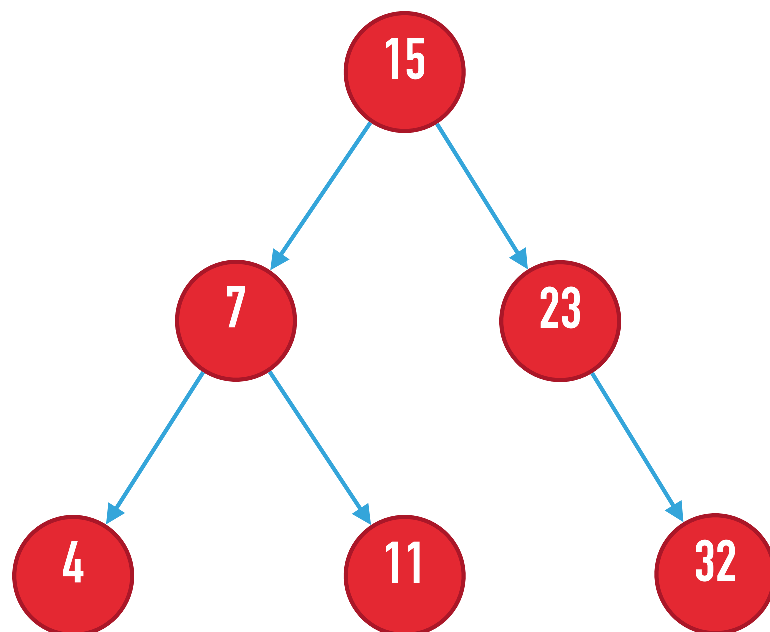

y

ALBERI BINARI: SUCCESSORE

- Vediamo come implementare il successore, il predecessore è simmetrico
- 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
- Caso difficile: il nodo non ha figlio destro
  - Allora dobbiamo risalire nella catena di genitori

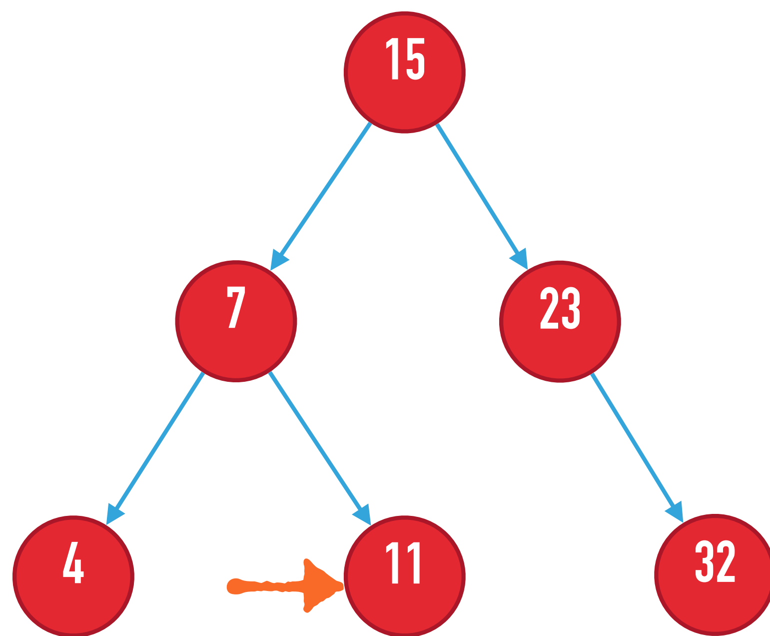
ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di “11”



ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di “11”



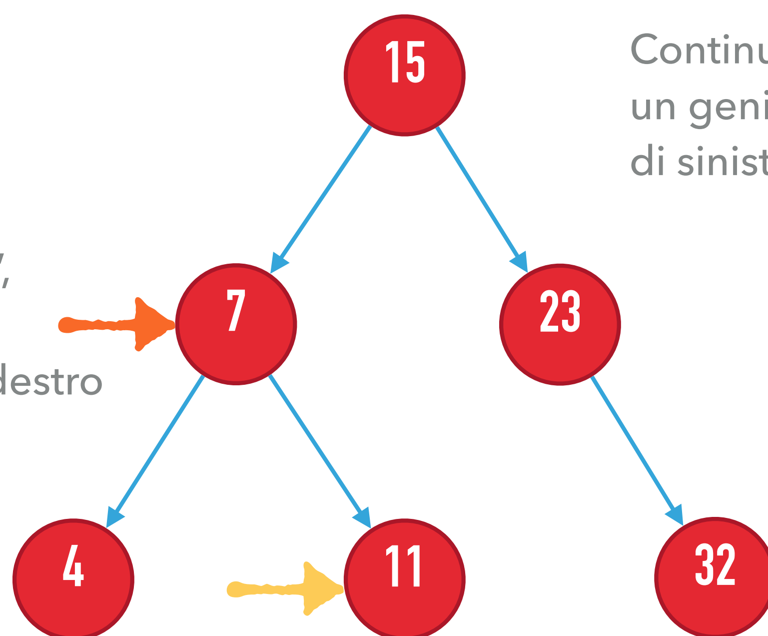
“11” non ha un figlio destro, quindi cerchiamo tra i genitori

ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di "11"

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

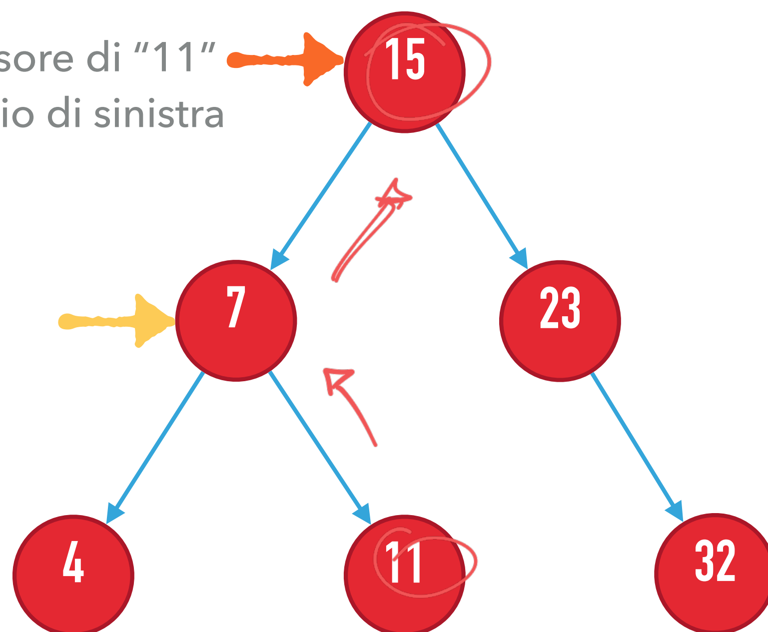
"7" è il genitore di "11", ma è minore, dato che proveniamo dal figlio destro



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

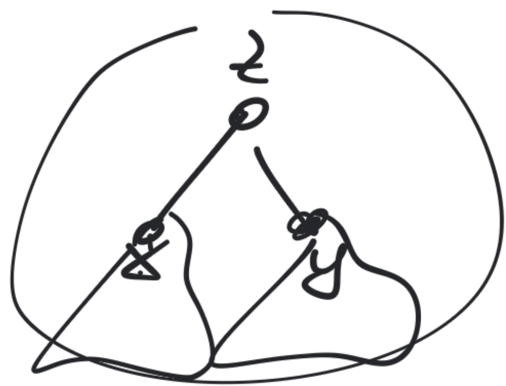
- Parametri: Nodo
- if nodo.right is not None:
  - return minimo(nodo.right)
- x = nodo
- y = nodo.parent
- While y is not None and y.right is x
  - # finché esiste un genitore e proveniamo dal figlio di destra
  - x = x.parent # saliamo di un livello
  - y = y.parent
- return y

WHY IT WORKS?

SIA x UN NODO E y IL SUO SUCCESSORE. ($y \neq$ NONE)

CHIAMO $T(x)$ L'ALBERO RADICATO IN x.

ALLORA $x \in T(y)$ o $y \in T(x)$.



Per assurdo se $\exists z$ antenato di x e y

$(z = LCA(x, y))$

se $z \neq x$ e $z \neq y$

$\Rightarrow x.KEY < z.KEY < y.KEY$ ⚡

ALBERI BINARI: SUCCESSORE

- La complessità di trovare il successore può essere divisa in due casi:
  - Uguale alla complessità di trovare il minimo se esiste u 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 tempo $O(h)$