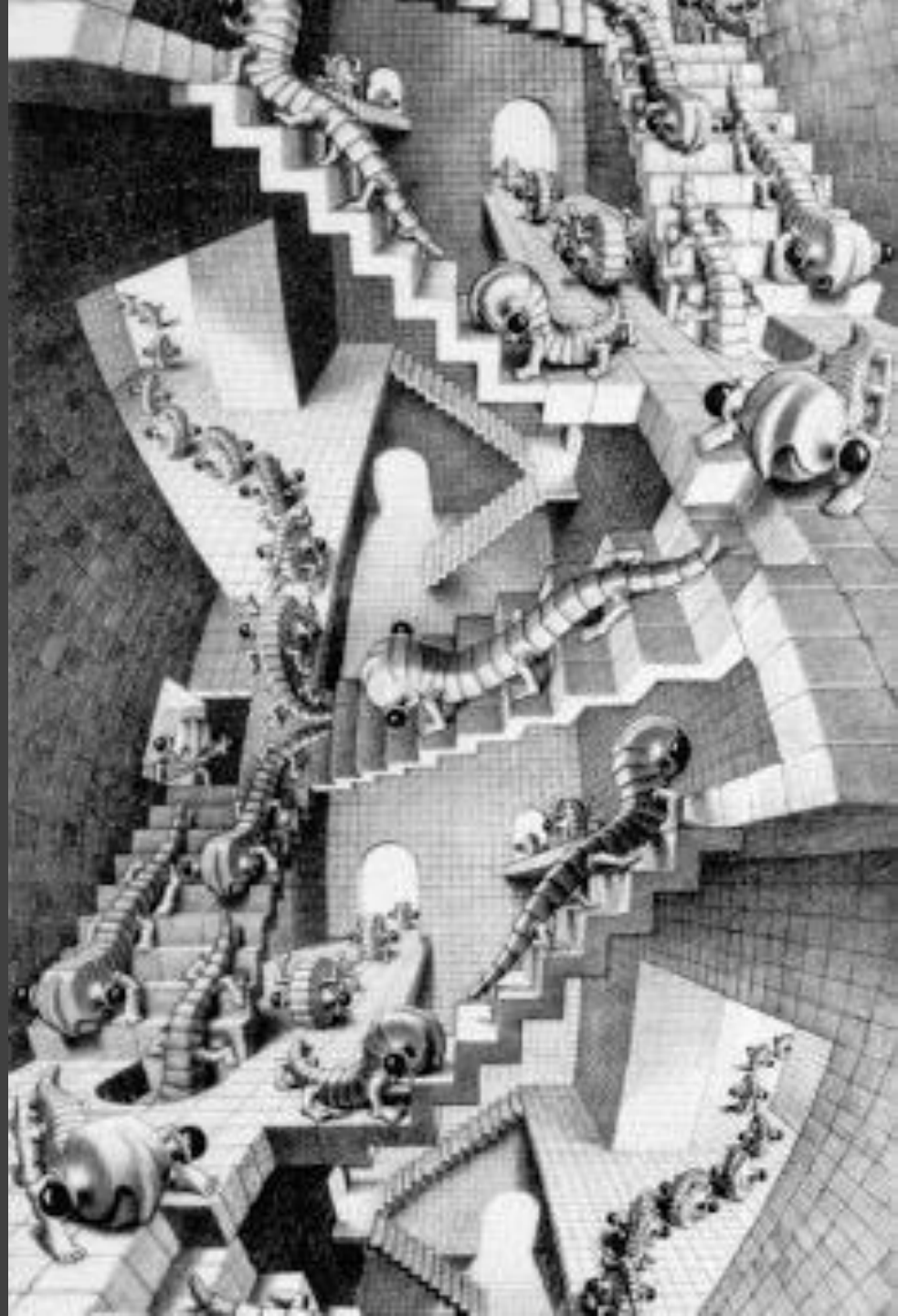
Università Roma Tre – Dipartimento di Matematica e Fisica
IN440 – Ottimizzazione Combinatoria
Prof. Marco Liverani

Algoritmi elementari su grafi



Visita di un grafo

- Dato un grafo $G = (V, E)$ l'operazione di visita consiste nell'esplorazione del grafo, a partire da un vertice scelto arbitrariamente, percorrendo gli spigoli che da un vertice portano in un altro, fino ad aver raggiunto tutti i vertici del grafo
- La strategia di visita deve permettere di arrivare in ciascun vertice (se il grafo è connesso e se l'orientazione degli spigoli lo permette, nel caso di grafi orientati) in modo efficiente, nel minor numero di passi possibile
- Soprattutto la strategia di visita deve evitare di entrare in un *loop* infinito, percorrendo un eventuale ciclo presente nel grafo senza mai uscirne
- La visita del grafo è assimilabile all'esplorazione di un labirinto: ad ogni passo ci troviamo su un vertice del grafo da cui si dipartono una o più strade; l'esecutore dell'algoritmo di visita non ha una vista d'insieme, al massimo può lasciare traccia dei vertici in cui è già passato in precedenza
- Due sono le strategie risolutive principali:
  - **Visita in ampiezza**: si visita un vertice di distanza k dal vertice di partenza, solo dopo aver visitato tutti i vertici di distanza $k-1$
  - **Visita in profondità**: si cerca di allontanarsi più velocemente possibile dal vertice di partenza, spingendosi «in profondità» verso zone inesplorate del grafo

Visita di un grafo

- Le due strategie di visita di un grafo, in ampiezza e in profondità, differiscono per l'ordine con cui vengono esaminati i vertici incontrati nel corso della visita:
  - l'algoritmo di **visita in ampiezza** visita i vertici nell'ordine con cui questi vengono incontrati: per far questo, memorizza i vertici ancora da visitare in una **coda** (una struttura dati di tipo FIFO – *first in, first out*) man mano che questi vengono incontrati nel corso della visita
  - l'algoritmo di **visita in profondità**, invece, visita ciascun vertice non appena questo viene incontrato nel corso del procedimento, rimandando la visita di altri vertici adiacenti: per far questo, memorizza i vertici, man mano che li incontra, in una **pila** (una struttura dati di tipo LIFO – *last in, first out*)
- Per il resto i due algoritmi sono talmente simili da poter essere rappresentati dalla stessa pseudo-codifica, in cui si ottiene l'uno o l'altro algoritmo, a seconda che la struttura dati Q, sia trattata come una coda o come una pila

Algoritmo 10 Visita(G, s)

Input: Il grafo G e una sorgente $s \in V(G)$
Output: L'albero di visita $T \subseteq G$ con radice in s

```
1:  Q = {s}, T = ({s}, ∅)
2:  marca s
3:  fintanto che Q ≠ ∅ ripeti
4:      sia u il primo elemento di Q
5:      per ogni v ∈ N(u) ripeti
6:          se v non è marcato allora
7:              aggiungi v a Q
8:              marca v
9:              π(v) = u nell'albero di visita T
10:         fine-condizione
11:         sia u il primo elemento di Q
12:     fine-ciclo
13:     estrai da Q il primo elemento
14: fine-ciclo
```

Visita in ampiezza

- L'algoritmo di visita in ampiezza (**BFS – *Breadth First Search***) riceve in input il grafo G e un vertice s come punto di partenza della visita
- Per tenere traccia dell'ordine con cui dovranno essere visitati i vertici, utilizza una ***coda*** Q in cui inserisce i vertici man mano che questi vengono incontrati (riga 9) e da cui li estrae all'inizio della visita del vertice stesso (riga 5)
- I vertici vengono anche colorati durante il processo di visita, per caratterizzarli con uno stato:
  - *bianco*: vertice mai visitato (assumiamo che inizialmente i vertici siano tutti bianchi)
  - *grigio*: il vertice è stato incontrato durante la visita, era bianco (mai incontrato prima) e lo si accoda in Q per visitarlo quando sarà il momento (righe 8-9)
  - *nero*: il vertice è stato estratto dalla coda e sono stati esplorati tutti gli spigoli uscenti dal vertice stesso; la visita del vertice è conclusa (riga 14)

Algoritmo 11 BFS(G, s)

Input: Il grafo G ed una sorgente $s \in V(G)$
Output: La sequenza di vertici visitati su G a partire da s

```
1:  Q = {s}, T = ({s}, ∅)
2:  D(v) = ∞ per ogni v ∈ V(G), D(s) = 0
3:  marca s di grigio e tutti gli altri di bianco
4:  fintanto che Q ≠ ∅ ripeti
5:      estrai un elemento v da Q
6:      per ogni u ∈ N(v) ripeti
7:          se u non è marcato allora
8:              marca u di grigio
9:              aggiungi u a Q
10:             D(u) = D(v) + 1
11:             aggiungi il vertice u e lo spigolo (v, u) all'albero T
12:         fine-condizione
13:     fine-ciclo
14:     marca v di nero
15: fine-ciclo
```

Visita in ampiezza

- Gli elementi vengono inseriti nella coda Q (e successivamente estratti per essere elaborati) nell'ordine in cui sono stati incontrati
- Visitando un vertice v vengono inseriti nella coda tutti i vertici adiacenti a v che non siano già stati visitati (righe 6-13)
- Quindi l'algoritmo visita prima tutti i vertici a distanza 1 da s (tutti i vertici adiacenti ad s), poi i vertici a distanza 2 da s (quelli adiacenti agli adiacenti ad s) e così via, allontanandosi sempre solo di un passo da s
- In questo modo, durante la visita, è possibile calcolare la ***distanza*** di ciascun vertice v dalla sorgente s: tale distanza è pari alla distanza del vertice «padre» di v da s incrementata di 1
- Durante la visita gli spigoli percorsi vengono aggiunti a T, un ***albero di cammini*** da s a tutti gli altri vertici (riga 11)

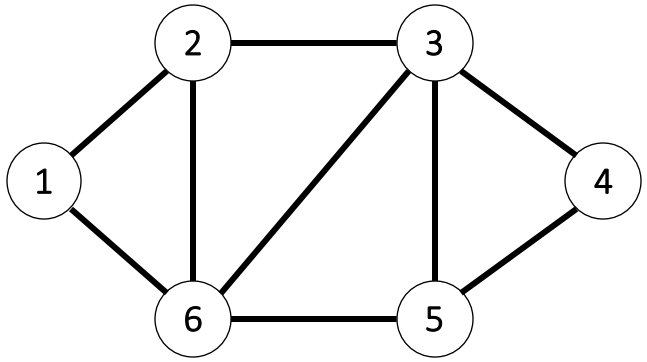
Algoritmo 11 BFS(G, s)

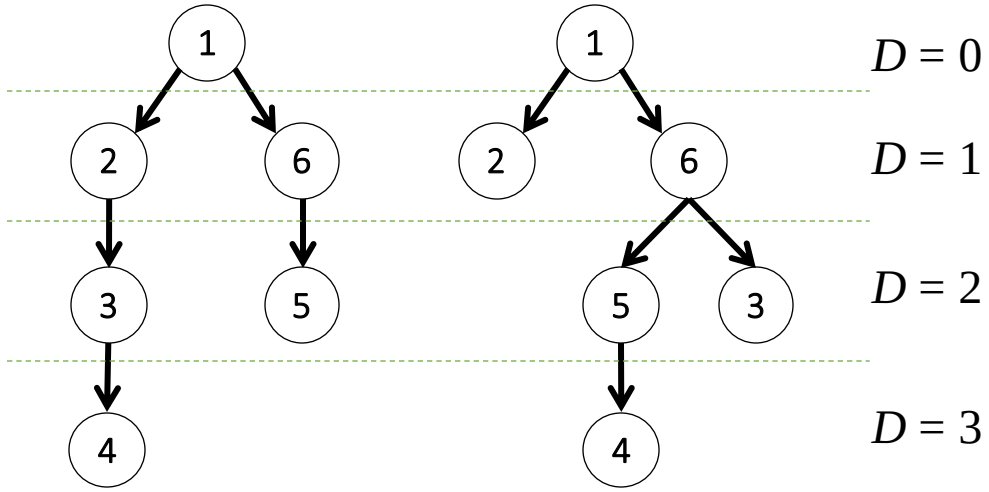
Input: Il grafo G ed una sorgente $s \in V(G)$
Output: La sequenza di vertici visitati su G a partire da s

1: $Q = \{s\}, T = (\{s\}, \emptyset)$
2: $D(v) = \infty$ per ogni $v \in V(G), D(s) = 0$
3: marca s di grigio e tutti gli altri di bianco
4: **fintanto che** $Q \neq \emptyset$ **ripeti**
5: estrai un elemento v da Q
6: **per ogni** $u \in N(v)$ **ripeti**
7: **se** u non è marcato **allora**
8: marca u di grigio
9: aggiungi u a Q
10: $D(u) = D(v) + 1$
11: aggiungi il vertice u e lo spigolo (v, u) all'albero T
12: **fine-condizione**
13: **fine-ciclo**
14: marca v di nero
15: **fine-ciclo**

Visita in ampiezza

- Le distanze $D(i)$ calcolate dall'algoritmo sono invarianti rispetto all'ordine con cui vengono presi in esame i vertici adiacenti a v





Algoritmo 11 BFS(G, s)

Input: Il grafo G ed una sorgente $s \in V(G)$
Output: La sequenza di vertici visitati su G a partire da s

1: $Q = \{s\}, T = (\{s\}, \emptyset)$
2: $D(v) = \infty$ per ogni $v \in V(G)$, $D(s) = 0$
3: marca s di grigio e tutti gli altri di bianco
4: **fintanto che** $Q \neq \emptyset$ **ripeti**
5: estrai un elemento v da Q
6: **per ogni** $u \in N(v)$ **ripeti**
7: **se** u non è marcato **allora**
8: marca u di grigio
9: aggiungi u a Q
10: $D(u) = D(v) + 1$
11: aggiungi il vertice u e lo spigolo (v, u) all'albero T
12: **fine-condizione**
13: **fine-ciclo**
14: marca v di nero
15: **fine-ciclo**

Visita in profondità

- L'algoritmo di visita in profondità (**DFS – *Depth First Search***) adotta un approccio opposto, rispetto al BFS: ad ogni iterazione identifica un vertice u adiacente a v non ancora visitato e si sposta su quel vertice, ripetendo la medesima operazione, senza prima completare la visita degli altri vertici adiacenti a v (su cui tornerà successivamente)
- L'algoritmo si presta bene ad essere implementato in forma ricorsiva
- Lo strumento con cui viene tenuta traccia dell'ordine di visita dei vertici, per riprendere a ritroso la visita di vertici precedentemente visitati per esplorare altri vertici adiacenti, è una struttura dati di tipo *stack* (pila), definita dalla struttura di chiamate ricorsive dell'algoritmo

Algoritmo 12 DFS(G, s)

Input: Il grafo G ed una sorgente $s \in V(G)$
Output: La sequenza di vertici visitati su G a partire da s

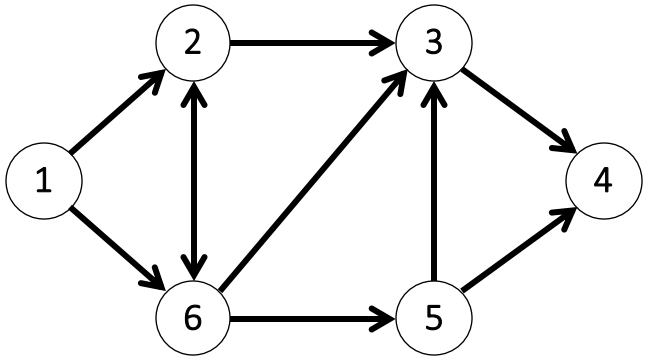
```
1: per ogni u ∈ V(G) ripeti
2:     colora u di bianco: c(u) = 0
3: fine-ciclo
4: per ogni u ∈ V(G) ripeti
5:     se u non è marcato allora
6:         VISITA(G, u)
7:     fine-condizione
8: fine-ciclo
```

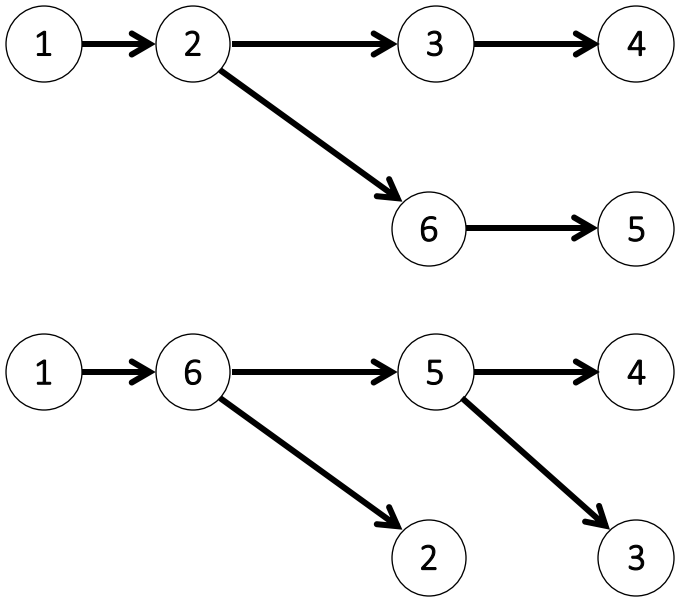
oppure:
VISITA(G, s)

VISITA(G, u)

```
1: colora u di grigio: c(u) = 1
2: per ogni v ∈ N(u) ripeti
3:     se v non è marcato allora
4:         VISITA(G, v)
5:     fine-condizione
6: fine-ciclo
7: colora u di nero: c(u) = 2
```

Visita in profondità





Non vengono calcolate le distanze, gli alberi di visita possono essere completamente diversi l'uno dall'altro

Algoritmo 12 DFS(G, s)

Input: Il grafo G ed una sorgente $s \in V(G)$
Output: La sequenza di vertici visitati su G a partire da s

1: **per ogni** $u \in V(G)$ **ripeti**
2: colora u di bianco: $c(u) = 0$
3: **fine-ciclo**
4: **per ogni** $u \in V(G)$ **ripeti**
5: **se** u non è marcato **allora**
6: VISITA(G, u)
7: **fine-condizione**
8: **fine-ciclo**

VISITA(G, u)

1: colora u di grigio: $c(u) = 1$
2: **per ogni** $v \in N(u)$ **ripeti**
3: **se** v non è marcato **allora**
4: VISITA(G, v)
5: **fine-condizione**
6: **fine-ciclo**
7: colora u di nero: $c(u) = 2$

Confronto fra i due algoritmi di visita di un grafo

- I due algoritmi sono analoghi, anche se producono alberi di visita con caratteristiche differenti
- La differenza di comportamento è dettata dall'uso di due ***strutture dati diverse*** per gestire l'elenco dei vertici individuati, ma ancora da visitare, come si osserva confrontando la formulazione iterativa dei due algoritmi:
  - una ***coda*** Q per l'algoritmo BFS (struttura dati di tipo FIFO: *First In First Out*)
  - una ***pila*** S (stack) per l'algoritmo DFS (struttura dati di tipo LIFO: *Last In First Out*)

Algoritmo 11 BFS(G, s)

Input: Il grafo G ed una sorgente $s \in V(G)$
Output: La sequenza di vertici visitati su G a partire da s

```
sia Q = ⟨s⟩ una coda, sia T = ({s}, ∅) un albero
per ogni v ∈ V(G) ripeti
    D(v) = ∞, colore(v) = bianco
fine-ciclo
D(s) = 0, colore(s) = grigio
fintanto che Q ≠ ∅ ripeti
    sia u il primo elemento della coda Q
    per ogni v ∈ N(u) ripeti
        se colore(v) = bianco allora
            colore(v) = grigio
            accoda v in fondo alla coda Q
            D(v) = D(u) + 1
            aggiungi il vertice v e lo spigolo (u, v) all'albero T
        fine-condizione
        sia u il primo elemento della coda Q
    fine-ciclo
    estrai dalla coda Q il primo elemento u
    colore(u) = nero
fine-ciclo
```

Algoritmo 12 DFS(G, s)

Input: Il grafo G ed una sorgente $s \in V(G)$
Output: La sequenza di vertici visitati su G a partire da s

```
sia S = ⟨s⟩ una pila, sia T = ({s}, ∅) un albero
per ogni v ∈ V(G) ripeti
    colore(v) = bianco
fine-ciclo
colore(s) = grigio
fintanto che S ≠ ∅ ripeti
    sia u l'elemento in cima alla pila S
    per ogni v ∈ N(u) ripeti
        se colore(v) = bianco allora
            colore(v) = grigio
            impila v in cima alla pila S
            aggiungi il vertice v e lo spigolo (u, v) all'albero T
        fine-condizione
        sia u l'elemento in cima alla pila S
    fine-ciclo
    estrai l'elemento u in cima alla pila S
    colore(u) = nero
fine-ciclo
```

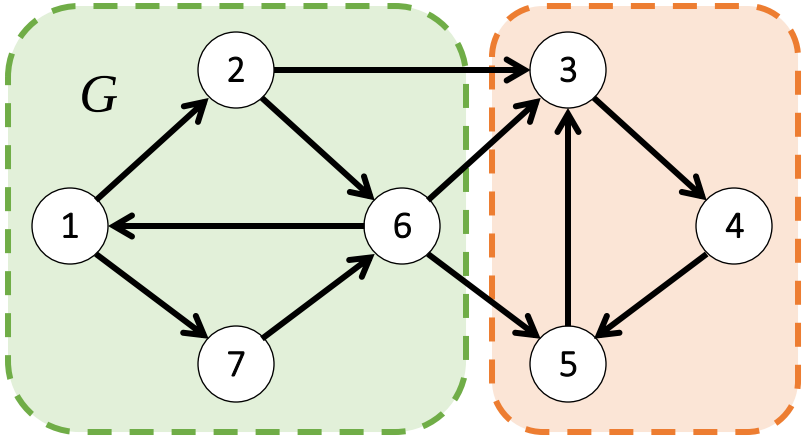
Complessità degli algoritmi di visita

- Dal confronto dei due algoritmi in forma iterativa risulta evidente che entrambi debbano avere la medesima complessità computazionale
- In entrambi i casi la ***complessità computazionale*** è $O(n + m)$, infatti:
  - nelle righe 1–5 viene eseguita una fase di inizializzazione lineare nel numero di vertici, $O(n)$
  - con i due cicli nidificati di BFS e DFS in forma iterativa (e con il singolo ciclo della funzione Visita di DFS in forma ricorsiva), vengono presi in esame tutti gli spigoli del grafo per cercare un vertice ancora da visitare, eseguendo così $O(m)$ operazioni

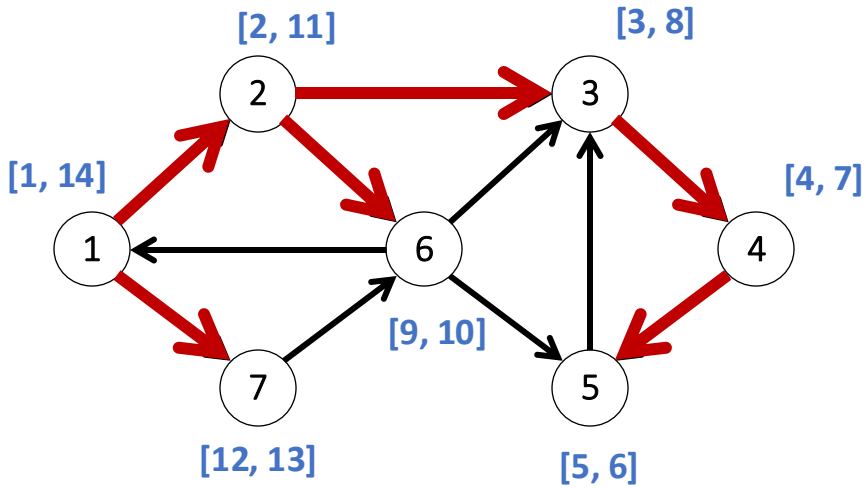
Applicazioni degli algoritmi di visita

- È possibile modificare gli algoritmi di visita per ricavare informazioni aggiuntive su un grafo G*; ad esempio:*
  - **verifica della connessione** del grafo (se al termine dell'esecuzione dell'algoritmo BFS o DFS qualche vertice è ancora bianco, allora il grafo non è connesso)
  - **componenti fortemente connesse** del grafo G orientato come sottoalberi dell'albero di visita DFS
  - verifica di **aciclicità** del grafo (se durante la visita incontro un vertice di colore grigio (che non sia il padre del vertice che stiamo visitando) allora nel grafo è presente almeno un ciclo
  - **ordinamento topologico** dei vertici di un grafo G *orientato* e *aciclico*: se $(u, v) \in E(G)$ allora $u \leq v$ nell'ordinamento topologico; si ricava con la visita in profondità DFS, sulla base dell'ordine inverso dei tempi di fine visita
  - **distanza tra vertici** mediante BFS
  - **cammino di lunghezza minima** tra due vertici mediante BFS

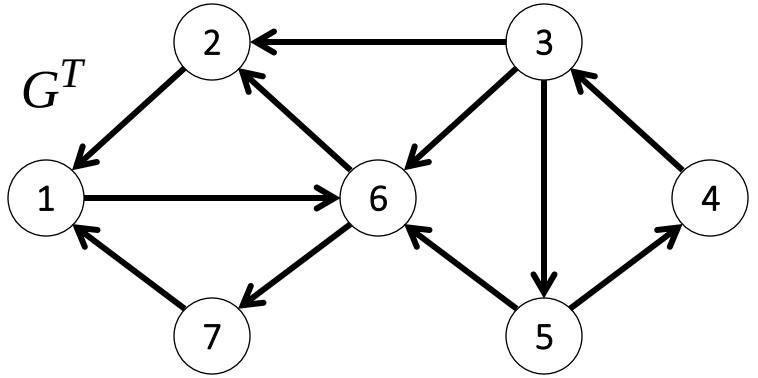
Componenti Fortemente Connesse



- In un grafo orientato le ***componenti fortemente connesse*** sono i sottografi in cui ogni coppia di vertici è mutuamente raggiungibile: u e v appartengono alla stessa componente fortemente connessa se e solo se esiste una cammino p: $v \rightsquigarrow u$ e un cammino p': $u \rightsquigarrow v$

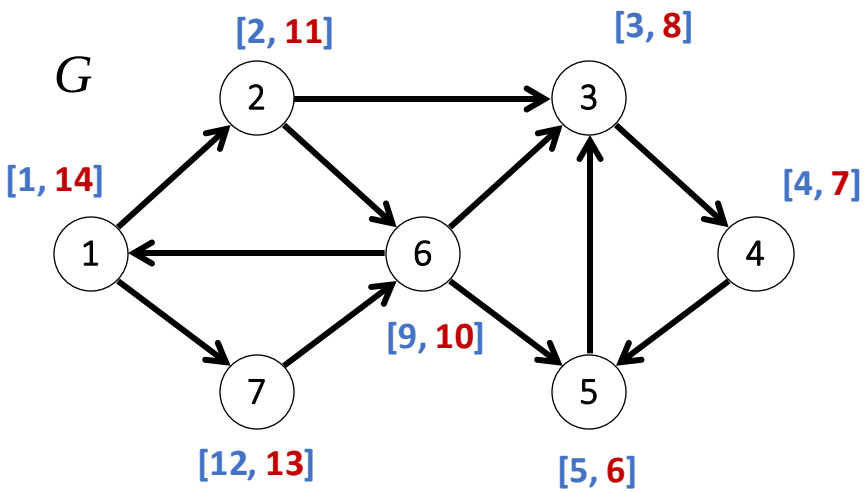


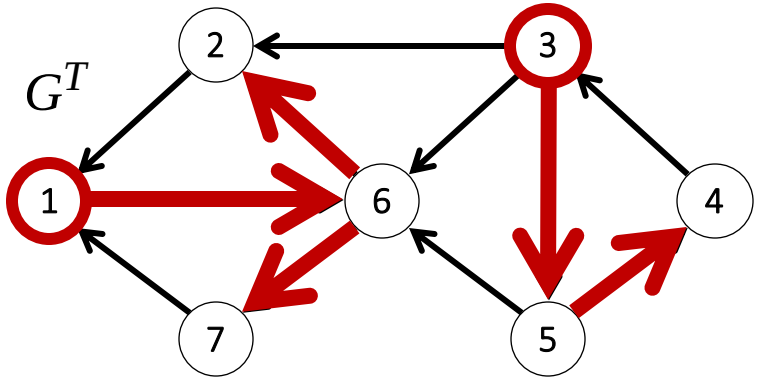
- Nell'algoritmo DFS si può tenere una variabile «contatore» che indichi il «tempo» di inizio e fine visita di ogni vertice, corrispondente all'inizio della chiamata alla funzione VISITA(G, v) e alla fine della stessa funzione



- Il ***grafo trasposto*** di un grafo orientato G si indica con G^T ed è il grafo con gli stessi vertici di G e gli spigoli con l'orientamento opposto: E(GT) = $\{(u, v)$ se e solo se $(v, u) \in E(G)\}$

Componenti Fortemente Connesse





$C_1 = \{1, 2, 6, 7\}$ $C_2 = \{3, 4, 5\}$

Algoritmo 13 COMPONENTIFORTEMENTECONNESSE(G)

Input: Il grafo G orientato

Output: Le componenti fortemente connesse di G

1: DFS(G) e calcola i tempi finali f_v per ogni $v \in V(G)$
2: calcola G^T
3: esegui DFS(G^T), ma nel ciclo principale considera i vertici in ordine decrescente rispetto a $\{f_v\}$
4: ciascun albero della foresta prodotta al passo precedente è una componente fortemente connessa di G

Algoritmo 12 DFS(G, s)

Input: Il grafo G ed una sorgente $s \in V(G)$

Output: La sequenza di vertici visitati su G a partire da s

1: **per ogni** $u \in V(G)$ **ripeti**
2: colora u di bianco: $c(u) = 0$
3: **fine-ciclo**
4: **per ogni** $u \in V(G)$ **ripeti**
5: **se** u non è marcato **allora**
6: VISITA(G, u)
7: **fine-condizione**
8: **fine-ciclo**

Componenti Fortemente Connesse

- La complessità dell'algoritmo è $O(n \log_2 n + m)$, infatti:
  - visita in profondità di G: $O(n + m)$
  - calcolo del grafo trasposto G^T: $O(m)$
  - ordinamento dei vertici in base al tempo di fine visita: $O(n \log_2 n)$
  - visita in profondità di G^T : $O(n + m)$

Algoritmo 13 COMPONENTIFORTEMENTECONNESSE(G)

Input: Il grafo G orientato

Output: Le componenti fortemente connesse di G

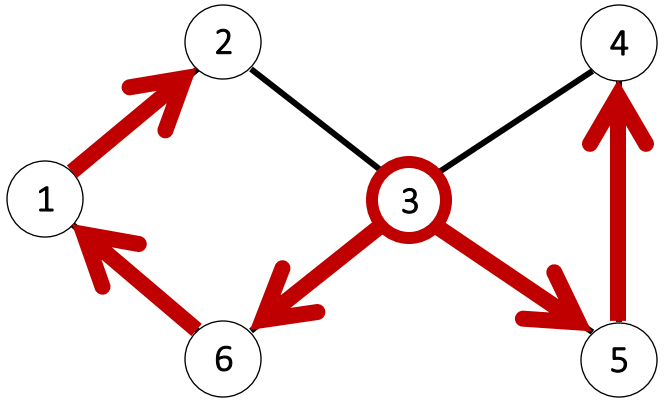
1: DFS(G) e calcola i tempi finali f_v per ogni $v \in V(G)$
2: calcola G^T
3: esegui DFS(G^T), ma nel ciclo principale considera i vertici in ordine decrescente rispetto a $\{f_v\}$
4: ciascun albero della foresta prodotta al passo precedente è una componente fortemente connessa di G

- Correttezza dell'algoritmo:
  - le componenti fortemente connesse di G sono le stesse di G^T
  - se esiste uno spigolo da C_1 a C_2 in G non esiste lo spigolo da C_1 a C_2 in G^T
  - i vertici con tempo di fine visita maggiore sono quelli che in una certa componente connessa hanno il maggior numero di discendenti, quindi nel grafo trasposto avranno il maggior numero di antenati e dunque visitandoli per primi circoscrivo la componente fortemente connessa a cui appartengono

Punti di articolazione e ponti

- In G *non orientato connesso* un ***punto di articolazione*** è un vertice che, se rimosso, sconnette il grafo
- Algoritmo per la verifica dei punti di articolazione:
  - eseguo DFS(G, v)
  - se nell'albero ottenuto v ha almeno due figli allora è un punto di articolazione

- Infatti questo significa che per passare dalla componente a cui appartiene il primo figlio alla componente a cui appartiene il secondo è indispensabile passare nuovamente per v; quindi eliminando v le due componenti si sconnetterebbero



- In G non orientato e connesso un ***ponte*** è uno spigolo che, se rimosso, sconnette il grafo
- Verifica di un ponte: condizione necessaria e sufficiente è che (u,v) è un ponte se e solo se non giace su un ciclo semplice di G

Alberi binari di ricerca

- Supponiamo di dover gestire un archivio A di n elementi, su cui è possibile definire una relazione d'ordine:
  - Eseguire una ricerca per verificare se l'elemento x è presente in archivio, richiede una complessità di $O(n)$
  - l'inserimento di un elemento richiede un tempo di $O(1)$ perché è sufficiente aggiungere l'elemento in coda all'archivio
  - l'ordinamento dell'intero archivio richiede un tempo di $O(n \log_2 n)$

- Se l'archivio è ordinato in ordine crescente, allora:
  - possiamo procedere alla ricerca con un criterio di ricerca «dicotomica» riducendo il tempo di ricerca a $O(\log_2 n)$
  - tuttavia per tenere ordinato l'archivio dobbiamo eseguire almeno $O(n)$ operazioni ad ogni inserimento
  - l'ordinamento non richiede tempi aggiuntivi, perché l'archivio è sempre ordinato

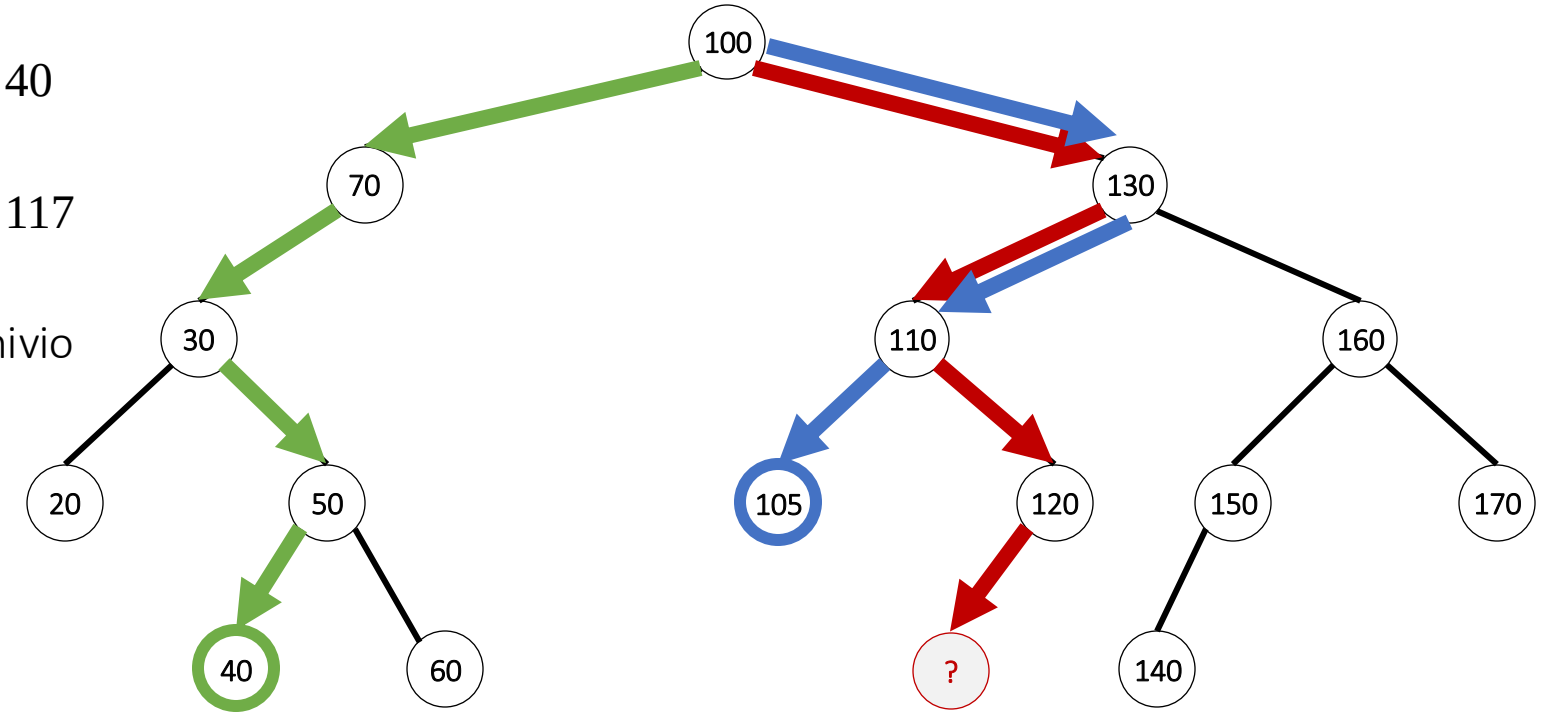
- Vogliamo costruire una struttura dati per ridurre al tempo stesso il numero di operazioni compiute per cercare un elemento, per inserirlo e per ordinare l'intero archivio (o una parte di esso)

Alberi binari di ricerca

- Un albero binario di ricerca $A = (V, E)$ è un albero radicato con le seguenti caratteristiche:
  - ogni vertice v ha al massimo due figli: left(v) e right(v)
  - gli elementi dell'archivio assegnati a v e ai suoi figli sono tali che left(v) $\leq v <$ right(v)

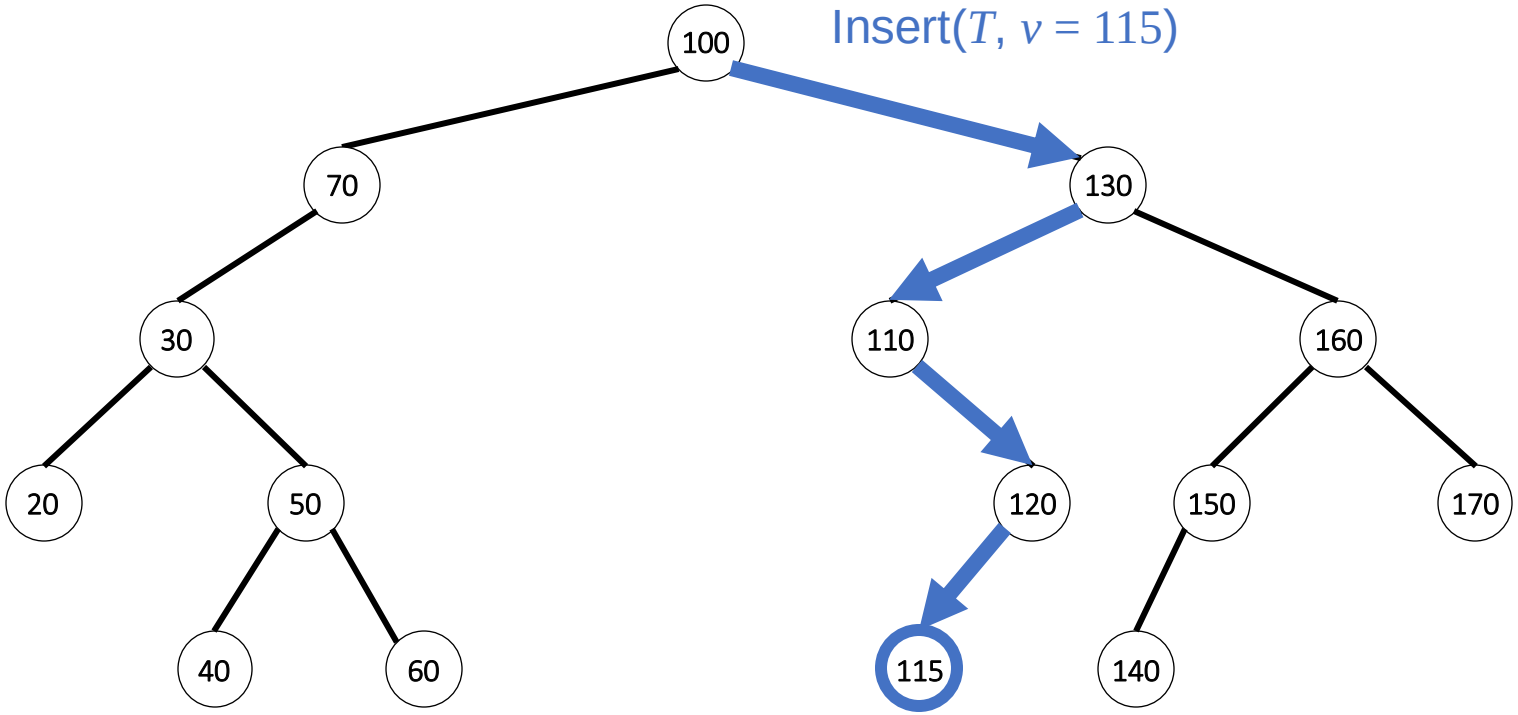
- Esempio: sia $A = \{20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160, 170\}$, $|A| = n = 14$

  - Verificare se esiste l'elemento $x = 40$
  - Verificare se esiste l'elemento $x = 117$
  - Inserire l'elemento $x = 105$ in archivio



Alberi binari di ricerca: inserimento

- Inserimento di un elemento v nell'albero T

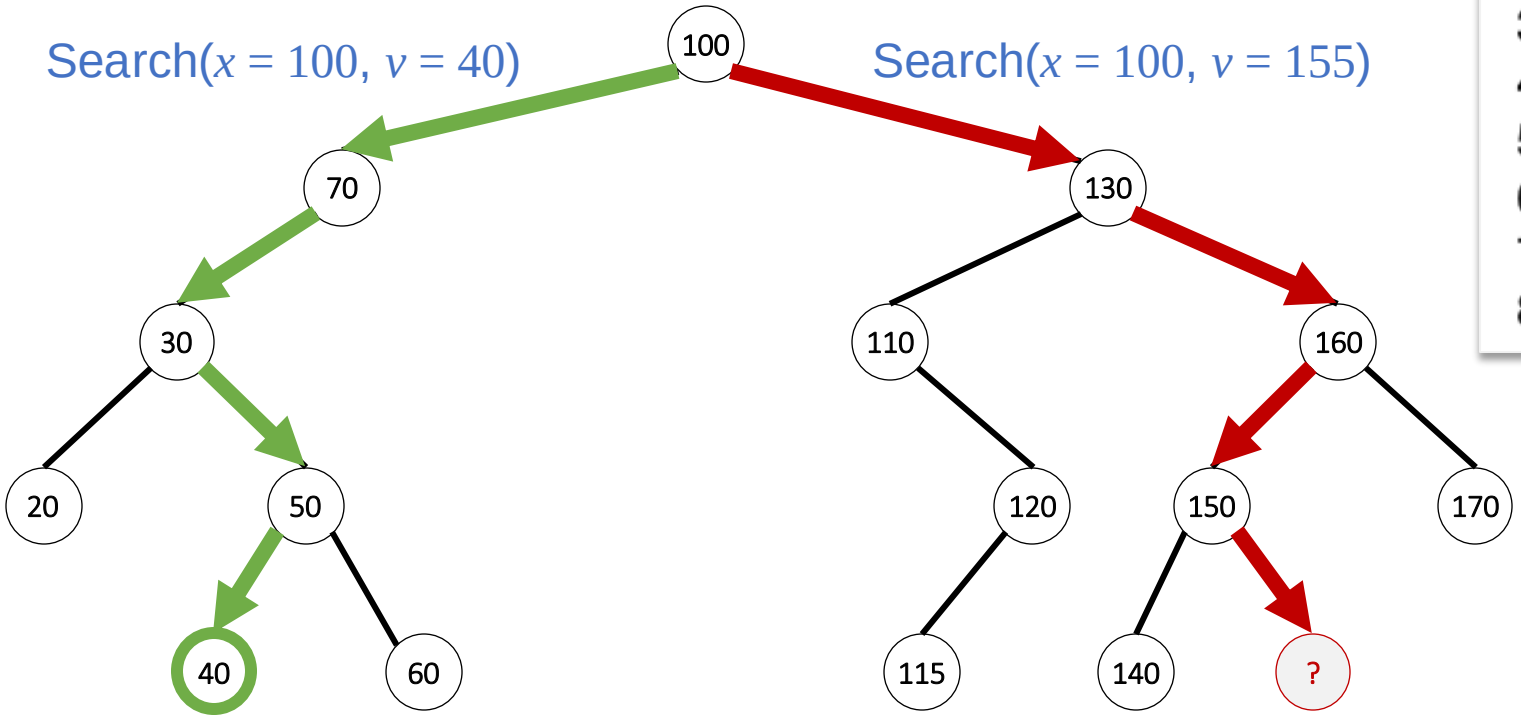



```
Insert(T, v)
 1: x = root(T), y = null
 2: fintanto che x ≠ null ripeti
 3:     y = x
 4:     se info(v) < info(x) allora
 5:         x = left(x)
 6:     altrimenti
 7:         x = right(x)
 8:     fine-condizione
 9: fine-ciclo
10: p(v) = y
11: se y = null allora
12:     root(T) = v
13: altrimenti
14:     se info(v) < info(p(v)) allora
15:         left(p(v)) = v
16:     altrimenti
17:         right(p(v)) = v
18:     fine-condizione
19: fine-condizione
```

- Complessità dell'algoritmo: $O(h(T))$

Alberi binari di ricerca: ricerca

- Ricerca l'elemento v nel sottoalbero con radice in x



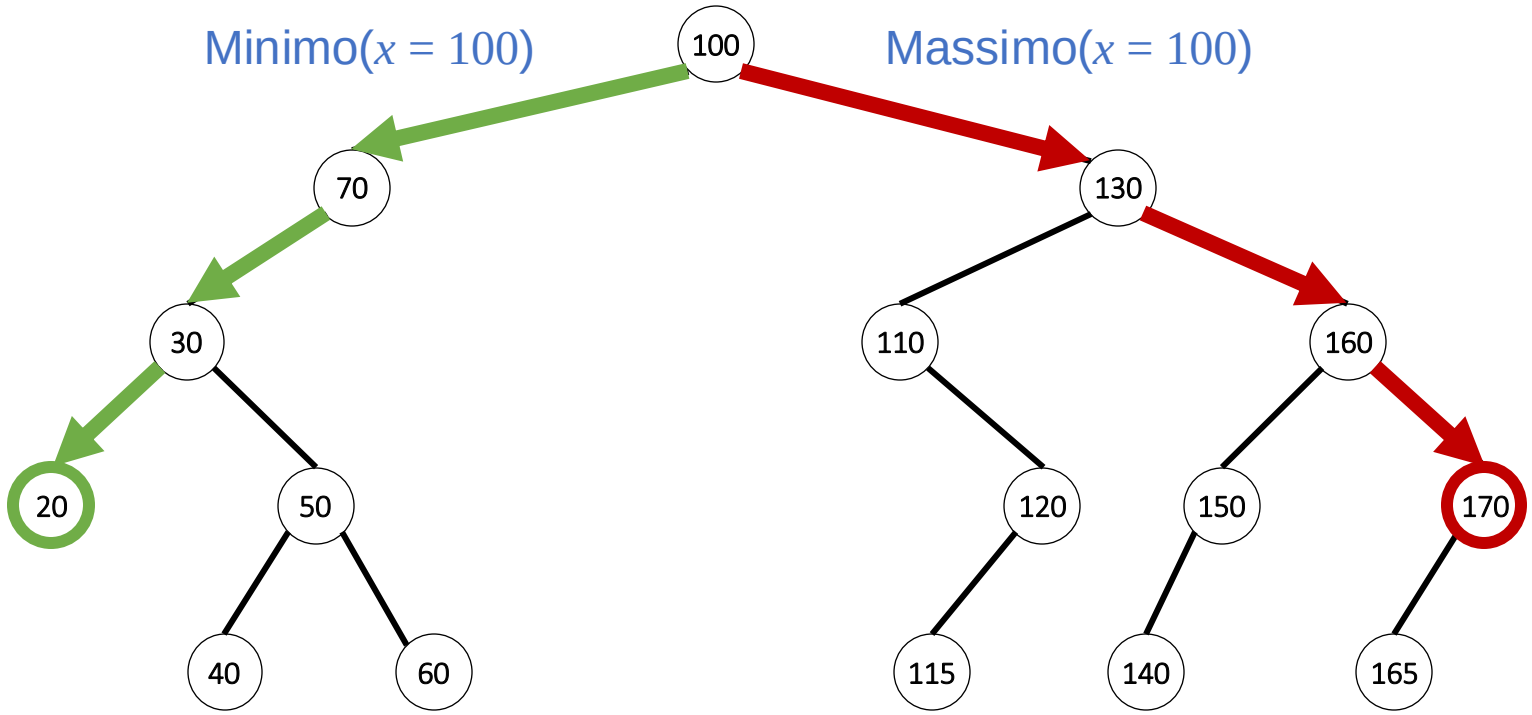
Search(x, v)

1: **fintanto che** $x \neq$ null e $v \neq$ info(x) **ripeti**
2: **se** $v <$ info(x) **allora**
3: $x =$ left(x)
4: **altrimenti**
5: $x =$ right(x)
6: **fine-condizione**
7: **fine-ciclo**
8: restituisci x

- Complessità dell'algoritmo: $O(h(T))$

Alberi binari di ricerca: minimo e massimo

- Ricerca dell'elemento minimo o dell'elemento massimo nel sottoalbero di T con radice in x



Minimo(x)

1: **fintanto che** left(x) $\neq$ null **ripeti**
2: x = left(x)
3: **fine-ciclo**
4: restituisci x

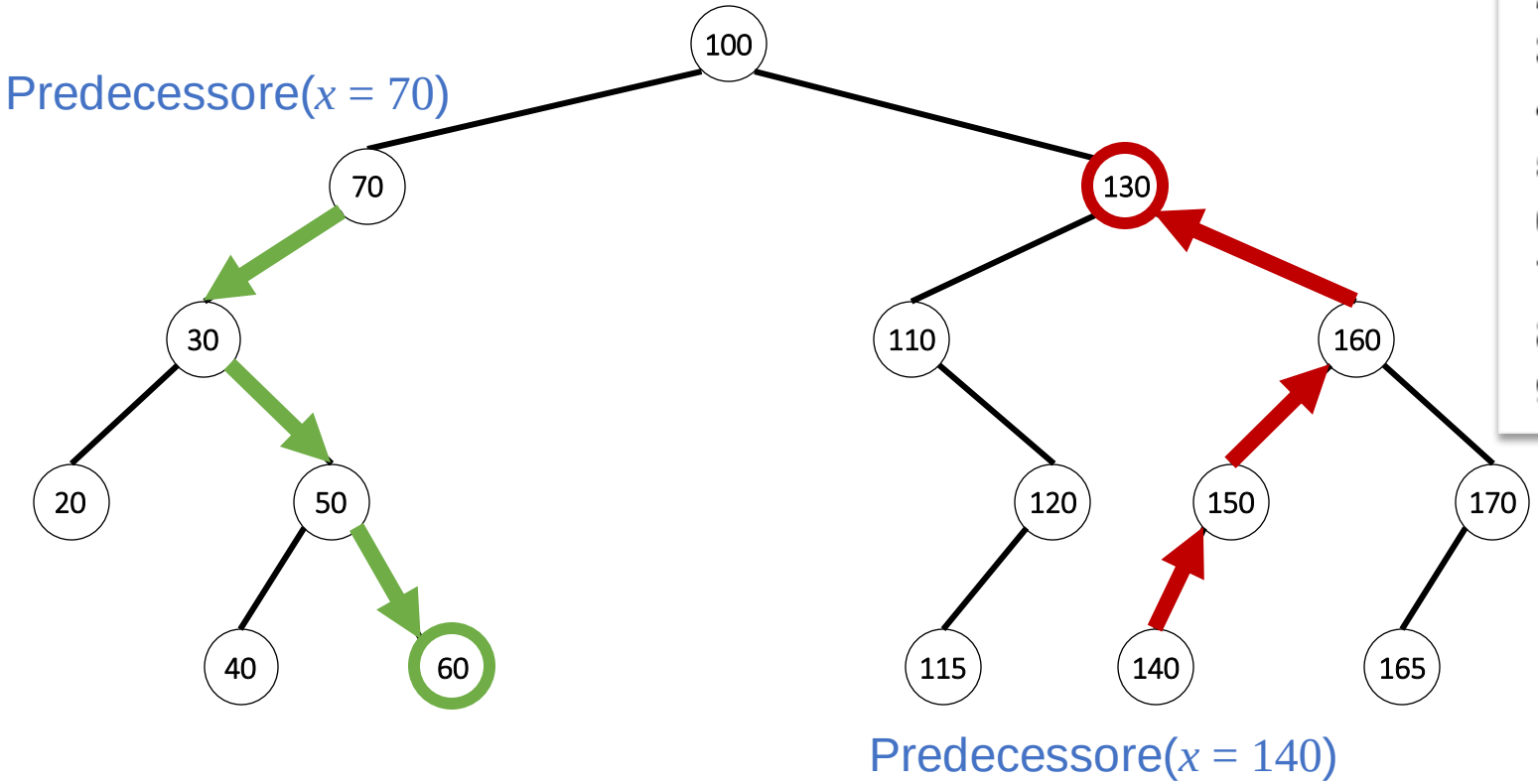
Massimo(x)

1: **fintanto che** right(x) $\neq$ null **ripeti**
2: x = right(x)
3: **fine-ciclo**
4: restituisci x

- Complessità dell'algoritmo: $O(h(T))$

Alberi binari di ricerca: predecessore

- Ricerca del predecessore di un elemento x sull'albero T (predecessore: il più grande elemento minore di x)



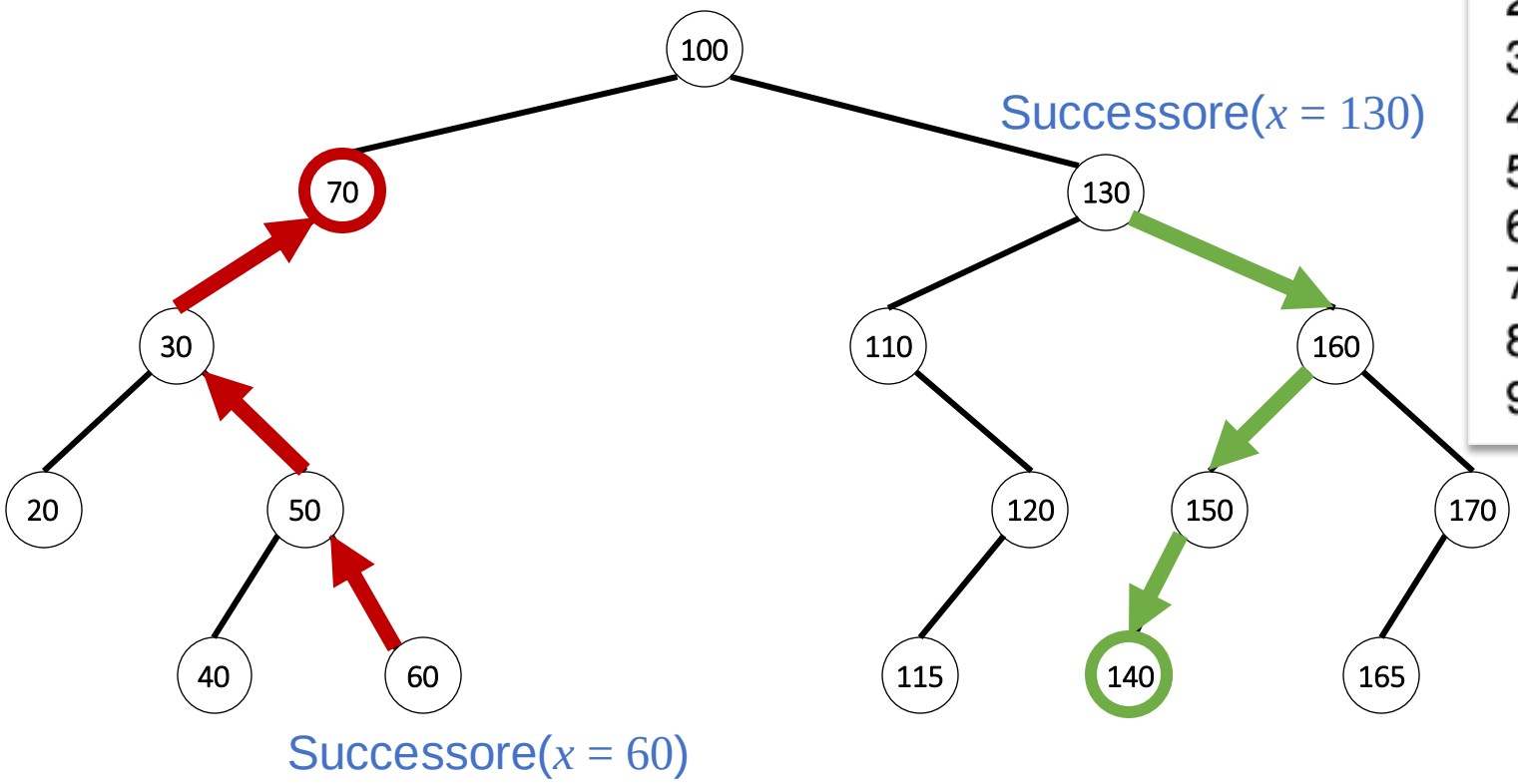
Predecessore(x)

1: **se** left(x) $\neq$ null **allora**
2: y = Massimo(left(x))
3: **altrimenti**
4: y = p(x)
5: **fintanto che** $y \neq$ null e x = left(y) **ripeti**
6: $x = y$, y = p(x)
7: **fine-ciclo**
8: **fine-condizione**
9: restituisci y

- Complessità dell'algoritmo: $O(h(T))$

Alberi binari di ricerca: successore

- Ricerca del successore di un elemento x sull'albero T (successore: il più piccolo elemento maggiore di x)



Successore(x)

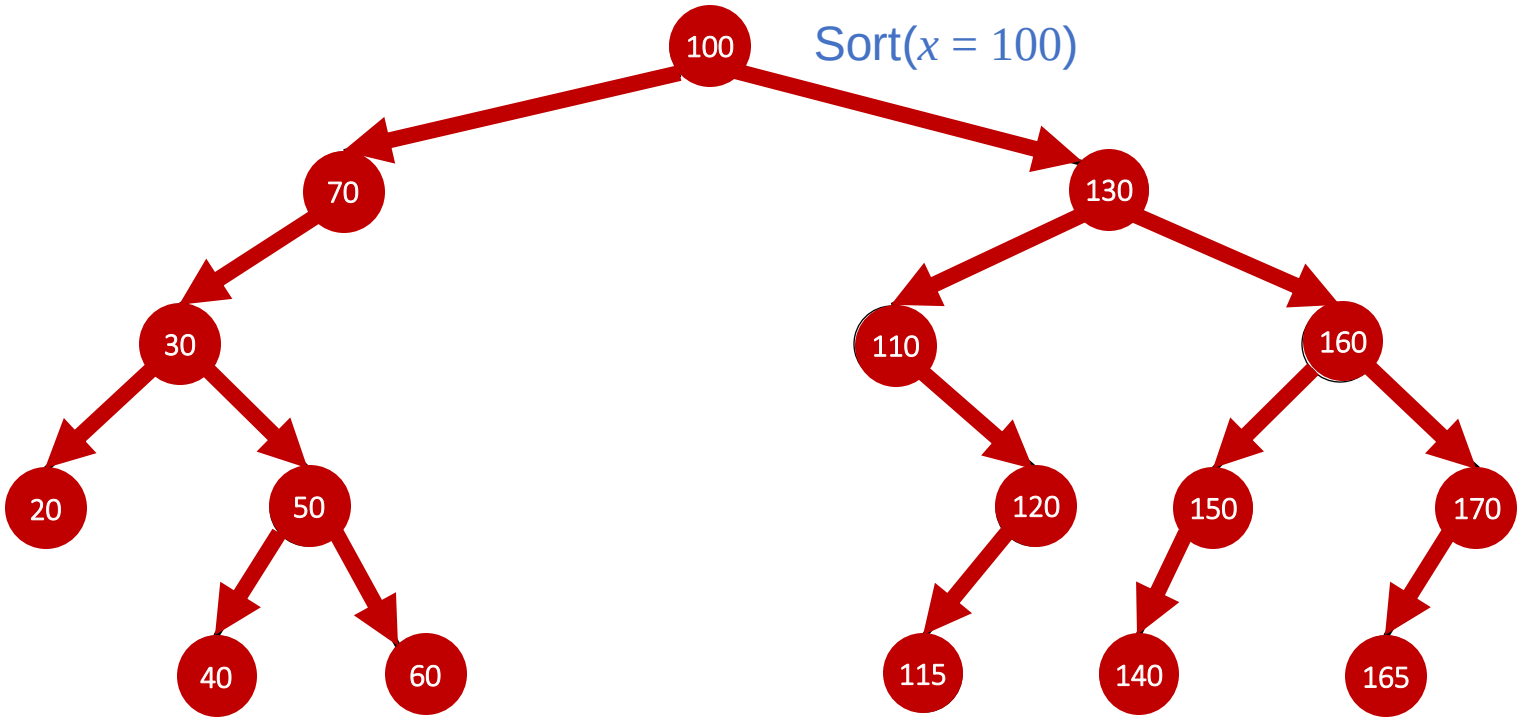
1: **se** right(x) $\neq$ null **allora**
2: y = Minimo(right(x))
3: **altrimenti**
4: y = p(x)
5: **fintanto che** $y \neq$ null e x = right(y) **ripeti**
6: $x = y$, y = p(x)
7: **fine-ciclo**
8: **fine-condizione**
9: restituisci y

- Complessità dell'algoritmo: $O(h(T))$

Alberi binari di ricerca: ordinamento

- Ordina in ordine crescente i vertici del sottoalbero di T con radice in x:
  - prima ordina gli elementi nel sottoalbero sinistro di x (elementi minori di x)
  - poi visualizza x
  - infine ordina gli elementi nel sottoalbero destro di x (elementi maggiori di x)

```
Sort(x)
1: se x ≠ null allora
2:     Sort(left(x))
3:     scrivi x
4:     Sort(right(x))
5: fine-condizione
```



20
30
40
50
60
70
100
110
115
120
130
140
150
160
165
170

- Complessità dell'algoritmo: $O(n)$

Riferimenti bibliografici

- Cormen, Leiserson, Rivest, Stein, «*Introduzione agli algoritmi e strutture dati*», terza edizione, McGraw-Hill (Cap. 12, Cap. 22)
- Alfred Aho, John Hopcroft, Jeffrey Ullman, «Data Structures and Algorithms», Addison-Wesley, 1987