# Algoritmi e Strutture Dati

## Esercitazione 3

Domenico Fabio Savo

# Esercitazione: visita di alberi

In molti casi, è necessario attraversare un albero visitandone tutti i nodi. Diversamente da collezioni di oggetti rappresentati in modo lineare (es: liste), dove è possibile attraversare la struttura senza incontrare ramificazioni, nel caso degli alberi è necessario invece seguire tutti i rami possibili a partire dalla radice.

Un algoritmo di "visita_generica" di un albero è il seguente:

**algoritmo** `visitaGenerica`$(nodo\ r)$

1. $S \leftarrow \{r\}$
2. **while** $(S \neq \emptyset)$ **do**
3. estrai un nodo $u$ da $S$
4. *visita il nodo* $u$
5. $S \leftarrow S \cup \{$ figli di $u\ \}$

# Esercitazione: visita di alberi

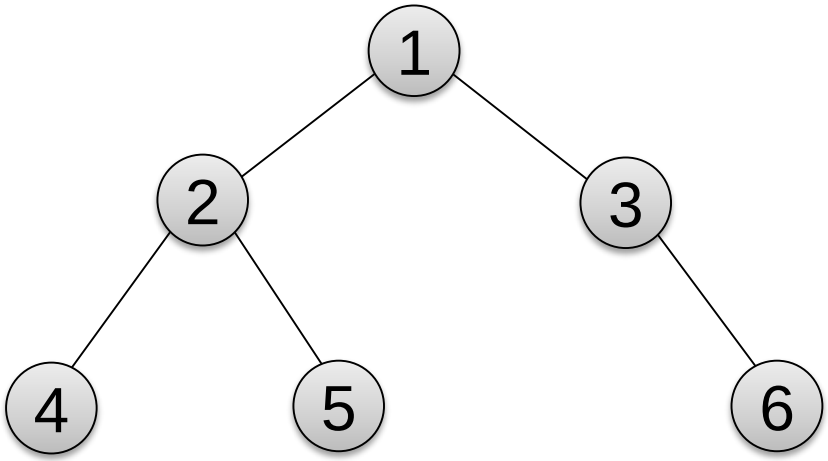
La disciplina di accesso definita sulla struttura **S** influisce sull'ordine in cui i nodi dell'albero saranno visitati.

Se implementiamo **S** con una **coda** (disciplina di accesso di tipo FIFO) allora otterremo una visita dell'albero in **ampiezza**.

```
algoritmo visitaBFS(nodo r)
    Coda C
    C.enqueue(r)
    while (not C.isEmpty()) do
        u ←C.dequeue()
        if (u ≠ null) then
            visita il nodo u
            C.enqueue(figlio sinistro di u)
            C.enqueue(figlio destro di u)
```

# Esercitazione: visita di alberi

**Esercizio:** Si consideri il seguente albero.

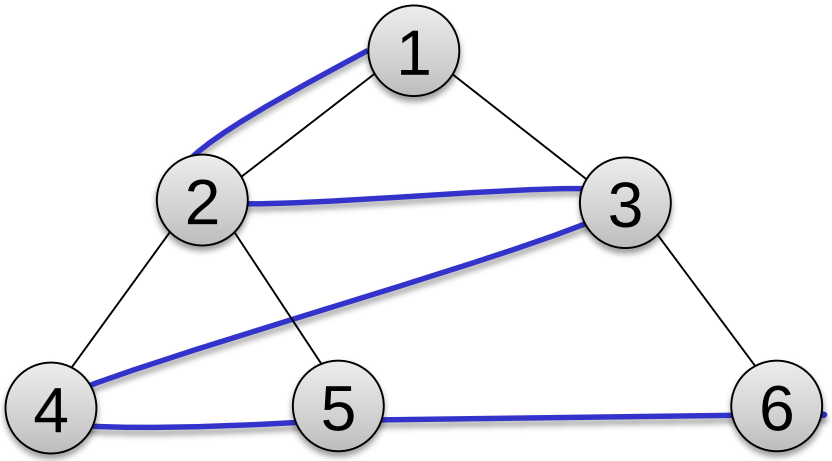



Indicare in quale ordine i nodi dell'albero verranno visitati dall'algoritmo **`visitaBFS`** partendo dal nodo **`1`**

# Esercitazione: visita di alberi

## Soluzione:

| Coda C | |
|---|---|
| 1 | |
| 3 2 | 1 |
| 5 4 3 | 2 |
| 6 5 4 | 3 |
| 6 5 | 4 |
| 6 | 5 |
| | 6 |



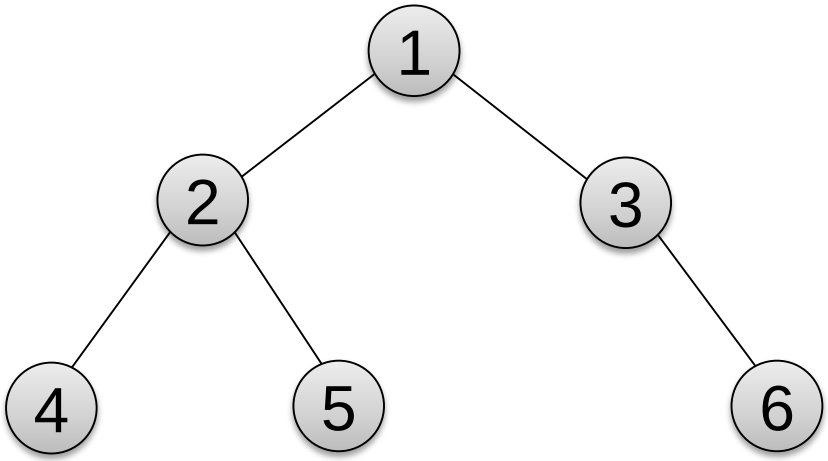
# Esercitazione: visita di alberi

Se implementiamo **S** con una **pila** (disciplina di accesso di tipo LIFO) allora otterremo una visita dell'albero in **profondità**.

```
algoritmo visitaDFS(nodo r)
    Pila S
    S.push(r)
    while (not S.isEmpty()) do
        u ← S.pop()
        if (u ≠ null) then
            visita il nodo u
            S.push(figlio destro di u)
            S.push(figlio sinistro di u)
```

# Esercitazione: visita di alberi

**Esercizio:** Si consideri il seguente albero.

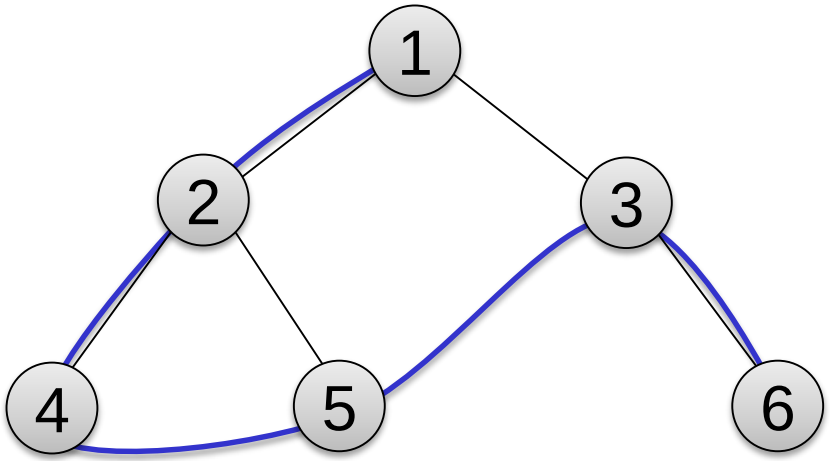



Indicare in quale ordine i nodi dell'albero verranno visitati dall'algoritmo **`visitaDFS`** partendo dal nodo **`1`**

# Esercitazione: visita di alberi

## Soluzione:

| Pila S | |
|---|---|
| 1 | |
| 2 3 | 1 |
| 4 5 3 | 2 |
| 5 3 | 4 |
| 3 | 5 |
| 6 | 3 |
| | 6 |



# Esercitazione: visita di alberi

Osserviamo che esiste una realizzazione ricorsiva dell'algoritmo di visita in profondità. Si noti che la pila S non appare esplicitamente nell'algoritmo: l'uso della ricorsione permette di usare la *pila dei record di attivazione* delle chiamate ricorsive.

**algoritmo** `visitaDFSRicorsiva`(*nodo r*)

1. **if** (*r* = null) **then return**
2. *visita il nodo r*
3. `visitaDFSRicorsiva`(*figlio sinistro di r*)
4. `visitaDFSRicorsiva`(*figlio destro di r*)

# Esercitazione: visita di alberi

Posizionando diversamente l'operazione di visita del nodo r rispetto alle chiamate ricorsive possiamo ottenere varianti dell'algoritmo in cui l'ordine di visita dei nodi dell'albero cambia.

Tre varianti classiche della visita in profondità sono:

- **Visita in preordine**: si visita prima la radice, poi si effettuano le chiamate ricorsive sul figlio sinistro e destro.
- **Visita simmetrica:** si effettua prima la chiamata ricorsiva sul figlio sinistro, poi si visita la radice, e infine si effettua la chiamata ricorsiva sul figlio destro.
- **Visita in postordine:** si effettuano prima le chiamate ricorsive sul figlio sinistro e destro, e infine si visita la radice.

# Esercitazione: visita di alberi

**algoritmo** `visitaDFS-Pre`(*nodo r*)

1. **if** (*r* = null) **then return**
2. *visita il nodo r*
3. `visitaDFSRicorsiva`(*figlio sinistro di r*)
4. `visitaDFSRicorsiva`(*figlio destro di r*)

Visita in preordine

**algoritmo** `visitaDFS-Sim`(*nodo r*)

1. **if** (*r* = null) **then return**
2. `visitaDFSRicorsiva`(*figlio sinistro di r*)
3. *visita il nodo r*
4. `visitaDFSRicorsiva`(*figlio destro di r*)

Visita simmetrica

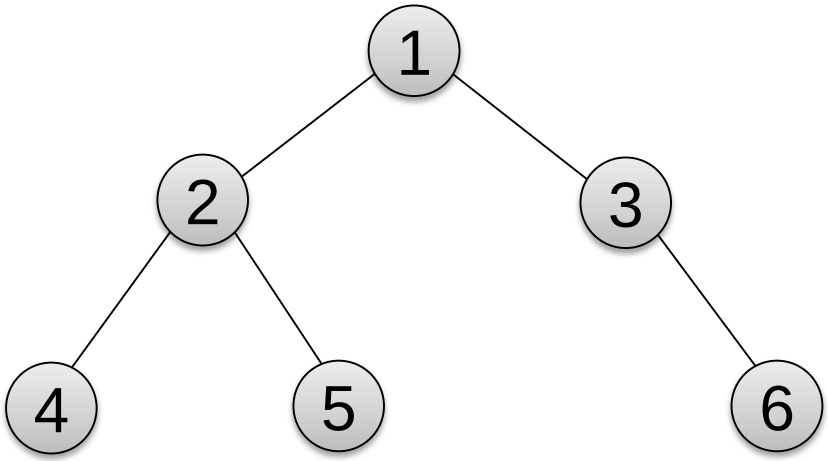
**algoritmo** `visitaDFS-Post`(*nodo r*)

1. **if** (*r* = null) **then return**
2. `visitaDFSRicorsiva`(*figlio sinistro di r*)
3. `visitaDFSRicorsiva`(*figlio destro di r*)
4. *visita il nodo r*

Visita postordine

# Esercitazione: visita di alberi

**Esercizio:** Si consideri il seguente albero.

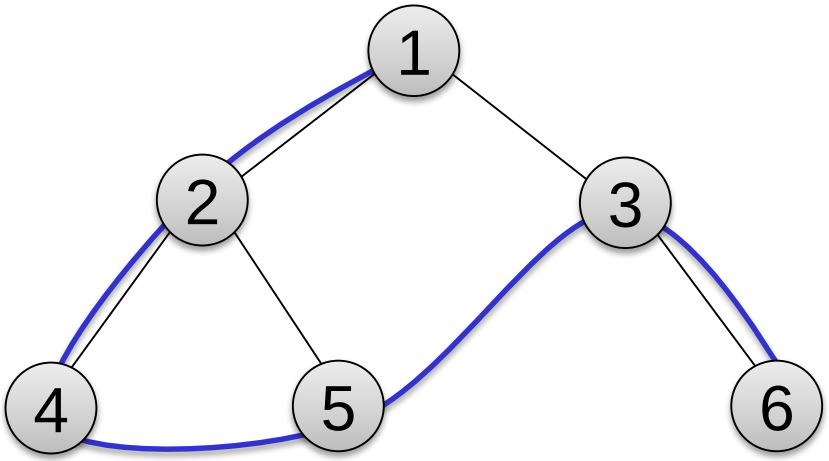



Indicare in quale ordine i nodi dell'albero verranno visitati dagli algoritmi **`visitaDFS-Pre`**, **`visitaDFS-Sim`**, e **`visitaDFS-Post`** partendo dal nodo **`1`**

# Esercitazione: visita di alberi

## Soluzione:

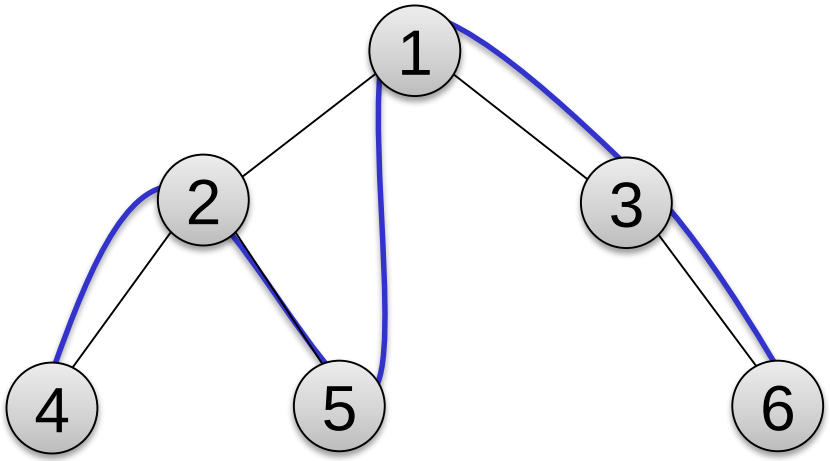
Visita in preordine:
1 2 4 5 3 6



# Esercitazione: visita di alberi

## Soluzione:

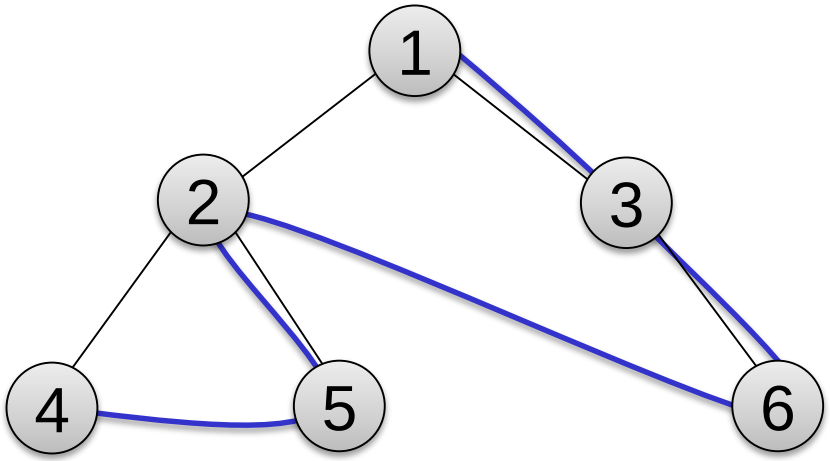
Visita simmetrica:
4 2 5 1 3 6



# Esercitazione: visita di alberi

## Soluzione:

Visita in postordine:
4 5 2 6 3 1



# Heap: esercizio

Considerate la seguente sequenza di numeri:

80, 15, 45, 13, 14, 31, 40, 9, 11, 13, 5

Assumendo che siano memorizzati in un array secondo la rappresentazione posizionale, discutere, motivando la risposta, se la sequenza rappresenta un heap.

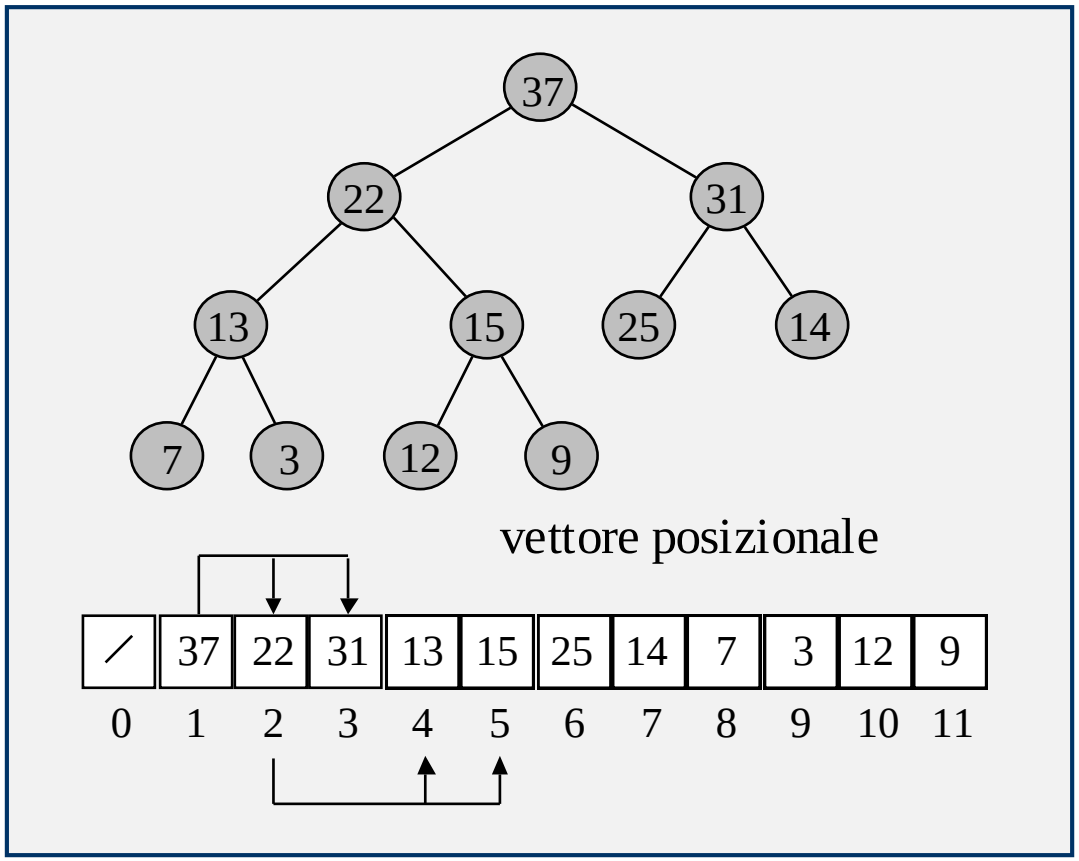
# Heap: caratteristiche

Un albero **H** è un **heap** se:

1. **H** è un albero binario;
2. **H** è completo fino al penultimo livello
3. Il valore dell'elemento in un nodo di **H** è sempre maggiore o uguale al valore degli elementi dei suoi figli, ovvero:

$$\textbf{chiave}(v) \geq \textbf{chiave}(\textbf{sin}(v))$$
$$\textbf{chiave}(v) \geq \textbf{chiave}(\textbf{des}(v))$$

# Heap: rappresentazione con vettore posizionale

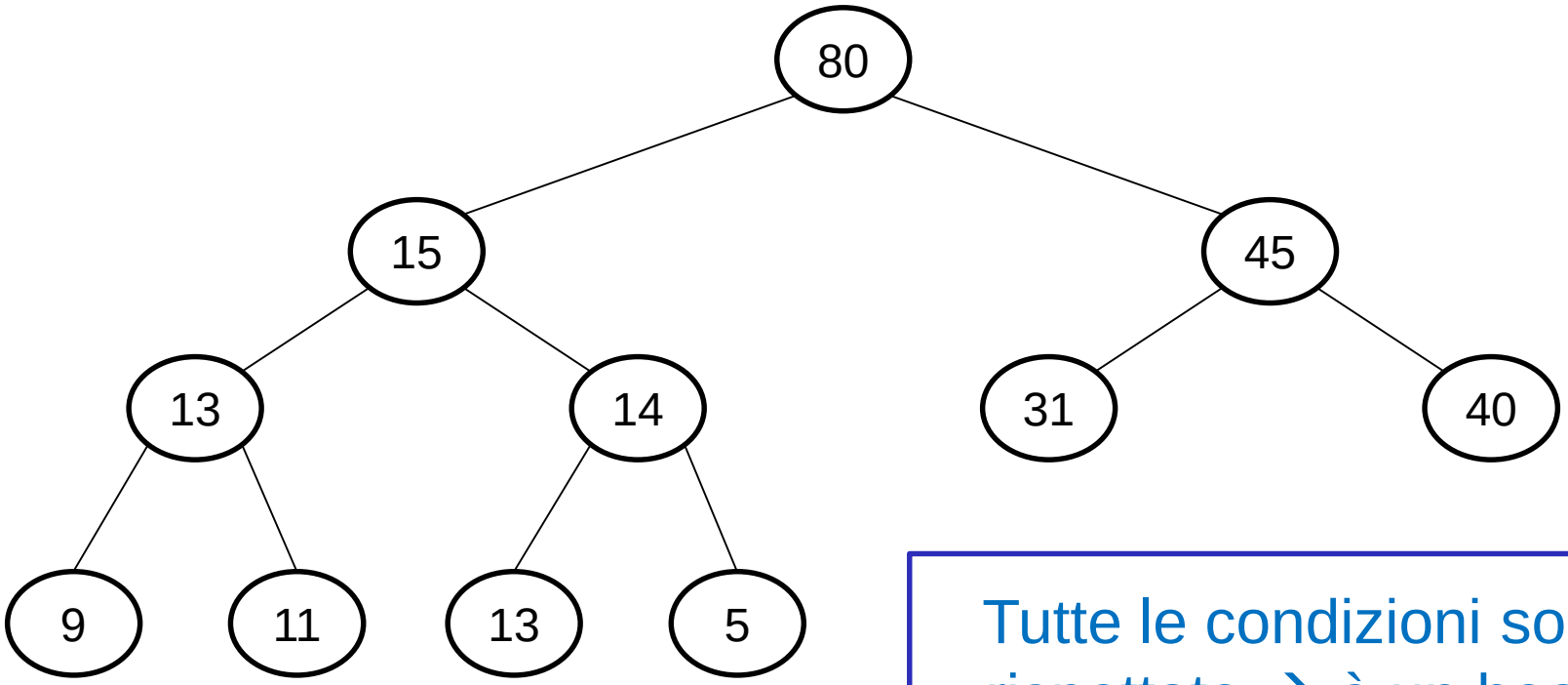



**sin(i)** = 2i

**des(i)** = 2i+1

# Heap: esercizio

80, 15, 45, 13, 14, 31, 40, 9, 11, 13, 5




Tutte le condizioni sono rispettate → è un heap!

# La moneta più leggera

Siano date **N** monete d'oro tutte dello stesso peso, tranne una che è **più leggera** delle altre, ed una bilancia con due piatti, su ciascuno dei quali è possibile mettere un numero qualunque di monete. È inoltre possibile sapere se le monete sui piatti hanno lo stesso peso, o quale dei due è più leggero.

Progettare un algoritmo per trovare la moneta "**leggera**" che richieda un numero di pesate nel caso peggiore pari a **O(log N)**.

# La moneta più leggera: soluzione (1/2)

Utilizziamo il seguente algoritmo ricorsivo.
Assumiamo che |M| sia una potenza di 2

```
Pesa(insieme di monete M)
   if(|M| = 1) then return "trovata!"
   else metti M/2 monete sul piatto A ed M/2 monete
   sul piatto B;
   if(A è più leggero) then Pesa(A)
   else Pesa(B)
```

# La moneta più leggera: soluzione (2/2)

Analizziamo l'algoritmo *Pesa(M)*

Ad ogni iterazione l'algoritmo effettua un confronto con un costo **O(1)** e poi effettua effettuata la chiamata ricorsiva **Pesa(M/2)**. Se l'insieme di monete dato in input ha cardinalità pari ad uno l'algoritmo termina.

La sua relazione di ricorrenza è:

$$T(n) = \begin{cases} O(1) + T(n/2) & \text{se } n>1 \\ 1 & \text{se } n=1 \end{cases}$$

→

Uguale a quella della ricerca binaria pertanto **T(n) = O(log n).**