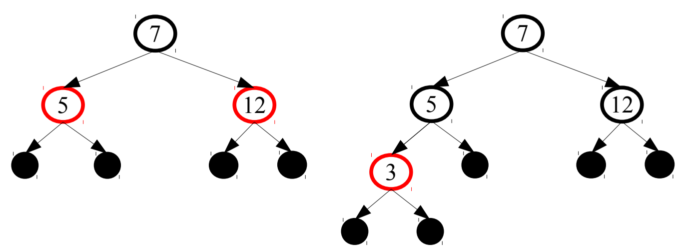
# *Algoritmi e Strutture Dati*
# *07/01/2014*

## Esercizio 1

La complessità dell'algoritmo è $O(n \log^2 n)$, in quanto si tratta di tre cicli annidati, dove il primo è eseguito $n$ volte, mentre i due cicli interni sono eseguiti ognuno $\log n$ volte.

## Esercizio 2

Si consideri l'albero della figura a sinistra qui sotto e si consideri l'inserimento della chiave 3. La chiave viene inserita a sinistra del 5, colorata di rosso. Ma poiché 5 è colorato di rosso, ci ritroviamo nel caso (3) della procedura di inserimento. Il padre e zio vengono colorati di nero, e il problema si sposta alla radice, che viene colorata di rosso. Poichè la radice ora è rossa, ma non ha padri, ci ritroviamo nel caso 1 e la radice viene colorata di rosso. L'altezza nera è ora pari a 2.




## Esercizio 3

**Parte 1:** La prima parte si risolve modificando una visita di Erdos, evitando di visitare stanze in cui ci siano mostri.

```
erdos(GRAPH G, NODE s, integer[] erdős, NODE[] p, boolean[] M)
    QUEUE S ← Queue()
    S.enqueue(s)
    foreach u ∈ G.V() − {s} do erdős[u] ← ∞

    erdős[s] ← 0
    while not S.isEmpty() do
        NODE u ← S.dequeue()
        foreach v ∈ G.adj(u) do                          % Esamina l'arco (u, v)
            if erdős[v] = ∞ and not M[u] then            % Il nodo u non è già stato scoperto e non c'è mostro
                erdős[v] ← erdős[u] + 1
                S.enqueue(v)
```

Il valore cercato si trova in erdos$[d]$. Sarebbe possibile migliorare ulteriormente l'algoritmo bloccando la ricerca quando si raggiunge $d$; in ogni caso, la ricerca nel caso pessimo richiede $O(m + n)$.

**Parte 2:** In questo caso, invece, è possibile utilizzare l'algoritmo di Dijsktra definendo i pesi in modo adeguato. Il peso degli archi è pari a 1 se il nodo di destinazione contiene un mostro, è pari a 0 altrimenti. La complessità risultante è pari a $O(n^2)$ o $O(m \log n)$, a seconda della struttura di dati utilizzata.

## Esercizio 4

E' sufficiente utilizzare una soluzione greedy che ordina i libri per altezza (costo $O(n \log n)$) e poi piazza i libri in ordine decrescente, utilizzando uno scaffale fino a quando questo contiene ancora libri. Il costo totale è dominato dall'ordinamento.
Dimostriamo informalmente la correttezza. Si consideri una soluzione ottima e si consideri il libro $l_1$ di altezza massima, e sia $s_1$ lo scaffale in cui si trova. Si consideri ora il secondo libro $l_2$ in ordine di altezza, e sia $s_2$ lo scaffale in cui si trova. Se $s_1 \neq s_2$, ci sono due possibilità:

- Se lo scaffale $s_1$ non è pieno (contiene meno di $L$ libri), si sposti il libro $l_2$ nello scaffale $s_1$; l'altezza dello scaffale $s_1$ non cambia, in quanto $y[l_2] \leq y[l_1]$ e $l_1$ resta nello scaffale $s_1$; può diminuire l'altezza dello scaffale $s_2$, in quanto abbiamo rimosso un libro.
- Altrimenti, si effettui uno scambio fra il libro $l_2$ e un libro $l_x$ contenuto in $s_1$, tale che $l_x \neq l_1$. L'altezza dello scaffale $s_1$ non cambia, in quanto $y[l_2] \leq y[l_1]$ e $l_1$ resta nello scaffale $s_1$; può diminuire l'altezza dello scaffale $s_2$, in quanto $y[l_x] \leq y[l_2]$.

Tuttavia, essendo la soluzione iniziale ottima, la soluzione ottenuta in questo modo può avere costo al più uguale a quella ottima, e quindi essere ottima anch'essa.
Abbiamo così ottenuto una soluzione ottima in cui due libri più alti sono nello stesso scaffale; ripetendo il procedimento di scambio, si può ottenere una soluzione ottima in cui uno scaffale contiene i primi $L$ libri più alti .
Si consideri ora il sottoproblema ottenuto considerando i restanti $n - L$ libri, e si riapplichi il procedimento; questo mostra che una qualunque soluzione ottima può essere trasformata nella soluzione proposta nel codice seguente.

```
integer altezza(integer[] y, integer n, integer L)
  sort(y, n)
  integer altezza ← 0
  integer left ← 0
  for i ← n downto 1 do
      if left = 0 then
          { Aggiungi libro a nuovo scaffale }
          altezza ← altezza + y[i]
          left ← L
      left ← left − 1
  return altezza
```