

## Complessità del calcolo

### Caso pessimo

$$T_M(n) = \max \{T_M(x), |x| = n\}$$

$$S_M(n) = \max \{T_M(x), |x| = n\}$$

### Notazioni

- O-grande: limite asintotico superiore.  
Data  $g(n)$ ,  $O(g(n)) = \{f(n) \mid \exists c, n_0 (c, n_0 > 0 : \forall n \geq n_0 0 \leq f(n) \leq cg(n))\}$
- $\Omega$ -grande: limite asintotico inferiore.  
Data  $g(n)$ ,  $\Omega(g(n)) = \{f(n) \mid \exists c, n_0 (c, n_0 > 0 : \forall n \geq n_0 0 \leq cg(n) \leq f(n))\}$
- $\Theta$ -grande: limite asintotico sia superiore sia inferiore.  
Data  $g(n)$ ,  $\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 (c_1, c_2, n_0 > 0 : \forall n \geq n_0 0 \leq c_1g(n) \leq f(n) \leq c_2g(n))\}$

## Teoremi di accelerazione lineare

- Se  $L$  è accettato da una MT  $M$  a  $k$  nastri con complessità  $S_M(n)$ , per ogni  $c > 0 (c \in R)$  si può costruire una MT  $M'$  a  $k$  nastri con complessità  $S_{M'}(n) < cS_M(n)$
- Se  $L$  è accettato da una MT  $M$  a  $k$  nastri con complessità  $S_M(n)$ , si può costruire una MT  $M'$  a 1 nastro (*non* a nastro singolo) con complessità  $S_{M'}(n) = S_M(n)$
- Se  $L$  è accettato da una MT  $M$  a  $k$  nastri con complessità  $S_M(n)$ , per ogni  $c > 0 (c \in R)$  si può costruire una MT  $M'$  a 1 nastro con complessità  $S_{M'}(n) < cS_M(n)$
- Se  $L$  è accettato da una MT  $M$  a  $k$  nastri con complessità  $T_M(n)$ , per ogni  $c > 0 (c \in R)$  si può costruire una MT  $M'$  (a  $k + 1$  nastri) con complessità  $T_{M'}(n) = \max \{n + 1, cT_M(n)\}$

### Conseguenze pratiche

- Lo schema di dimostrazione è valido per qualsiasi tipo di modello di calcolo, quindi anche per calcolatori reali (es.: aumentare il parallelismo fisico (16bit  $\rightarrow$  32bit  $\rightarrow$  ...)).
- Aumentando la potenza di calcolo in termini di risorse disponibili si può aumentare la velocità di esecuzione, ma il miglioramento è al più lineare.
- Miglioramenti di grandezza superiore possono essere ottenuti solo cambiando algoritmo e non in modo automatico.

## Macchina RAM

### Glossario

- *Accumulatore*: è la prima cella del modello della memoria, indicata con  $M[0]$ .
- *Immediato*: è un numero intero.
- *ADD*: Sono le operazioni elementari di somma (ADD), sottrazione (SUB), moltiplicazione (MULT), divisione (DIV).

### Costi Logaritmici

Il costo della copia di un numero  $n$  da una cella all'altra è tante micro-operazioni elementari quanti sono i bit necessari a codificare  $n$ , cioè  $\log(n)$ .  
Il costo dell'accesso ad una cella di posizione  $n$ -esima è

l'apertura di  $\log(n)$  gate logici ad altrettanti banchi di memoria.

In forma sintetica:

$$l(i) = \text{if } i = 0 \text{ then } 1 \text{ else } \lfloor \log_2 |i| \rfloor + 1$$

### Teorema di correlazione polinomiale

Sotto "ragionevoli" ipotesi di criteri di costo (il criterio di costo costante per la RAM non è ragionevole) se un problema è risolvibile mediante un modello di calcolo  $M_1$  con complessità  $C_1(n)$ , allora è risolvibile da qualsiasi altro modello di calcolo  $M_2$  con complessità  $C_2(n) \leq P_2(C_1(n))$ , essendo  $P_2$  un opportuno polinomio.

Comando		Operazione	Complessità	Descrizione
LOAD	X	$M[0] = M[X]$	$l(x)$	Carica in $M[0]$ il contenuto della cella X
LOAD=	X	$M[0] = X$	$l(x) + l(M[x])$	Carica in $M[0]$ l'immediato X
LOAD*	X	$M[0] = M[M[X]]$	$l(x) + l(M[x]) + l(M[M[x]])$	Carica in $M[0]$ dall'indirizzo $M[X]$
STORE	X	$M[X] = M[0]$	$l(x) + l(M[0])$	Carica in $M[X]$ il contenuto di $M[0]$
STORE*	X	$M[X] = M[M[0]]$	$l(x) + l(M[x]) + l(M[0])$	Carica in $M[X]$ dall'indirizzo $M[0]$
ADD	X	$M[0] = M[0] + M[X]$	$l(M[0]) + l(x) + l(M[x])$	Carica in $M[0]$ il risultato dell'operazione
ADD=	X	$M[0] = M[0] + X$	$l(M[0]) + l(x)$	
ADD*	X	$M[0] = M[0] + M[M[X]]$	$l(M[0]) + l(x) + l(M[x]) + l(M[M[x]])$	
READ	X	$M[X] = \text{read}()$	$l(\text{input value}) + l(x)$	Salva in X il valore letto in input
READ*	X		$l(\text{input value}) + l(x) + l(M[x])$	
WRITE	X	$\text{write}(M[X])$	$l(x) + l(M[x])$	Scrive in output il valore di X
WRITE=	X	$\text{write}(X)$	$l(x)$	Scrive in output l'immediato X
WRITE*	X	$\text{write}(M[M[X]])$	$l(x) + l(M[x]) + l(M[M[x]])$	Scrive in output l'indirizzo di X
JUMP	label	$PC = b(\text{label})$	1	Salta alla label indicata
JZ	label	$\text{if } M[0] == 0$	$l(M[0])$	Salta alla label indicata se l'accumulatore è 0.
JGZ	label	$\text{if } M[0] > 0$	$l(M[0])$	Salta alla label indicata se l'accumulatore è maggiore di 0.
HALT			1	Interrompe l'esecuzione del programma.

## Algoritmi

Si adotta il criterio di **costo costante** (manipoliamo numeri che non richiedono quantità di memoria molto più grandi della dimensione dell'input).

Ogni istruzione viene eseguita in un tempo costante  $c_i$ .

### Complessità di un algoritmo *divide et impera*

- Si divide il problem in  $b$  sottoproblemi, ciascuno con dimensione  $\frac{1}{b}$ .
- Se il problema ha dimensione  $n$  piccola a sufficienza ( $n < c$ ,  $c$  costante caratteristica del problema), esso può essere risolto in tempo costante ( $\Theta(1)$ ).
- $D(n)$  è il costo di dividere il problema, e  $C(n)$  è il costo di ricombinare i sottoproblemi.  $T(n)$  è il costo per risolvere il problema totale.

$$\text{Equazione di Ricorrenza} \quad T(n) = \begin{cases} \Theta(1) & \text{se } n < c \\ D(n) + aT(\frac{n}{b}) + C(n) & \text{altrimenti} \end{cases}$$

## Risoluzione di ricorrenze

### Metodo della sostituzione

- formulare un'ipotesi di soluzione
- sostituire la soluzione nella ricorrenza, e dimostrazione (per induzione) che è in effetti una soluzione
- *Esempio:*
  1.  $T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + n$
  2. Supponiamo  $T(n) = O(n \log_2(n))$
  3. Mostro che  $T(n) \leq cn \log_2(n)$  per una opportuna costante  $c$
  4. Supponiamo che valga per  $T(\lfloor \frac{n}{2} \rfloor)$ , cioè  $T(\lfloor \frac{n}{2} \rfloor) \leq c \lfloor \frac{n}{2} \rfloor \log_2(\lfloor \frac{n}{2} \rfloor)$
  5. Sostituendo in  $T(n)$  ottengo  $T(n) \leq 2c \lfloor \frac{n}{2} \rfloor \log_2(\lfloor \frac{n}{2} \rfloor) + n \leq cn \log_2(\frac{n}{2}) + n = cn \log_2(n) - cn \log_2(2) + n = cn \log_2(n) + n \leq cn \log_2(n)$ , basta che  $c > 1$
  6. Mostriamo che la disuguaglianza vale nel caso limite  $n = 1$ :  $T(1) = 1 \rightarrow T(1) = 1 \leq c \log_2(1) = 0$  non funziona.
  7. Siccome  $T(n) \leq cn \log_2(n)$  deve valere solo da un certo  $n_0$  in poi, ci basta prendere una costante  $c$  tale che  $T(2) = 4 \leq 2c \log_2(2)$  e  $T(3) = 5 \leq 3c \log_2(3)$ , cioè  $c \geq 2$

### Teorema dell'esperto (Master Theorem)

- Data la ricorrenza  $T(n) = aT(\frac{n}{b}) + f(n)$ , in cui  $a \geq 1$ ,  $b > 1$ ,  $\lfloor \frac{n}{b} \rfloor$  o  $\lceil \frac{n}{b} \rceil$ .
  1. se  $f(n) = O(n^{\log_b a - \varepsilon})$  per qualche  $\varepsilon > 0$ , allora  $T(n) = \Theta(n^{\log_b a})$
  2. se  $f(n) = \Theta(n^{\log_b a})$ , allora  $T(n) = \Theta(n^{\log_b a} \log(n))$
  3. se  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  per qualche  $\varepsilon > 0$ , e  $af(\frac{n}{b}) \leq cf(n)$  per qualche  $c < 1$  e per tutti gli  $n$  grandi a sufficienza, allora  $T(n) = \Theta(f(n))$
- Se  $f(n) = \Theta(n^k)$ , con  $k$  una qualche costante:
  1. se  $k < \log_b a \rightarrow T(n) = \Theta(n^{\log_b a})$
  2. se  $k = \log_b a \rightarrow T(n) = \Theta(n^k \log(n))$
  3. se  $k > \log_b a \rightarrow T(n) = \Theta(n^k)$
- Se la ricorrenza è del tipo:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq m \leq h \\ \sum_{1 \leq i \leq h} a_i T(n - i) + cn^k & \text{se } n > m \end{cases}$$

- Posto  $a = \sum_{1 \leq i \leq h} a_i$
- Allora segue che:
  - se  $a = 1$  allora  $T(n) = O(n^{k+1})$
  - se  $a \geq 2$  allora  $T(n) = O(a^n n^k)$
- *Esempio:*  $T(n) = T(n-1) + \Theta(n) \rightarrow T(n) = O(n^2)$

## Complessità degli algoritmi di ordinamento

Nome	Caso Pessimo	Caso Ottimo
Insertion Sort	$O(n^2)$	$O(n)$
Merge Sort	$\Theta(n \log(n))$	$\Theta(n \log(n))$
Heapsort	$O(n \log(n))$	$O(n \log(n))$
Quicksort	$\Theta(n^2)$	$\Theta(n \log(n))$
Counting Sort	$O(n + k)$	$O(n + k)$

## Insertion Sort

```

INSERTION-SORT(A)
  for j = 2 to A.length
    key = A[j]
    i = j - 1
    while i > 0 and A[i] > key
      A[i+1] = A[i]
      i = i - 1
    A[i+1] = key

```

## Merge Sort

```

MERGE-SORT(A, p, r)
  if p < r
    q = ⌊(p+r)/2⌋
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)

MERGE (A, p, q, r)
  n1 = q-p+1
  n2 = r-q
  CreaArray(L[1...n1+1] e R[1...n2+1])
  for i = 1 to n1
    L[i] = A[p+i-1]
  for j = 1 to n2
    R[j] = A[q+j]
  L[n1+1] = ∞
  R[n2+1] = ∞
  i = 1
  j = 2
  for k = p to r
    if L[i] <= R[j]
      A[k] = L[i]
      i = i+1
    else
      A[k] = R[j]
      j = j+1

```

## Heapsort

```

PARENT(i)
  return ⌊i/2⌋
LEFT(i)
  return 2*i
RIGHT(i)
  return 2*i+1

MAX-HEAPIFY(A, i)
  l = LEFT(i)
  r = RIGHT(i)
  if l <= A.heapsize and A[l] > A[i]
    max = l
  else
    max = i
  if r <= A.heapsize
    and A[r] > A[max]
    max = r
  if max != i
    swap A[i] ↔ A[max]
    MAX-HEAPIFY(A, max)

```

```

BUILD-MAX-HEAP(A)
  A.heapsize = A.length
  for i = A.length/2 to 1
    MAX-HEAPIFY(A, i)

HEAPSORT(A)
  BUILD-MAX-HEAP(A)
  for i = A.length to 2
    swap A[1] ↔ A[i]
    A.heapsize = A.heapsize - 1
    MAX-HEAPIFY(A, 1)

```

## Quicksort

```

QUICKSORT(A, p, r)
  if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q-1)
    QUICKSORT(A, q+1, r)

PARTITION(A, p, r)
  x = A[r]
  i = p-1
  for j = p to r-1
    if A[j] <= x
      i = i+1
      swap A[i] ↔ A[j]
  swap A[i+1] ↔ A[r]
  return i+1

```

## Counting Sort

```

COUNTING-SORT(A, B, k)
  for i = 0 to k
    C[i] = 0
  for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
  for i = 1 to k
    C[i] = C[i] + C[i-1]
  for j = A.length to 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1

```

## Strutture dati

$S$  = collezione,  $k$  = chiave

### Operazioni comuni

- SEARCH( $S$ ,  $k$ )
- INSERT( $S$ ,  $k$ )
- DELETE( $S$ ,  $k$ )
- MINIMUM( $S$ )
- MAXIMUM( $S$ )
- SUCCESSOR( $S$ ,  $k$ )
- PREDECESSOR( $S$ ,  $k$ )

### Pila/Stack

- Politica **LIFO** (Last In First Out)
- POP( $S$ ) cancella l'elemento in cima alla pila e lo restituisce
- PUSH( $S$ ) aggiunge un elemento in cima alla pila
- $S.top$  è l'elemento in cima alla pila
- Se può contenere al massimo  $n$  elementi, si implementa come un array di dimensione  $n$

### Code/Queue

- Politica **FIFO** (First In First Out)
- ENQUEUE( $S$ ) inserisce un elemento in fondo alla coda
- DEQUEUE( $S$ ) cancella il primo elemento dalla coda
- $S.head$  è l'elemento nella coda da più tempo
- $S.tail$  è la posizione dove verrà inserito il nuovo elemento

### Lista doppiamente concatenata

- Ogni oggetto  $x$  della lista  $L$  è costituito da 3 attributi:
  - **key** è il contenuto
  - **next** è il puntatore all'oggetto seguente
  - **prev** è il puntatore all'oggetto precedente
- Se  $x.next == nil$ ,  $x$  non ha successore
- Se  $x.prev == nil$ ,  $x$  non ha predecessore
- $L.head$  è il puntatore al primo elemento della lista

### Costi di una lista

Search  $T(n) = O(n)$   
 Insert  $T(n) = O(1)$   
 Delete  $T(n) = O(1)$

### Dizionario/Dictionary

- Supporta solo le operazioni di INSERT, DELETE, SEARCH
- Agli oggetti di un dizionario si accede tramite le chiavi, che sono numeri interi
- Se la cardinalità  $m$  dell'insieme delle possibili chiavi è piccola, conviene l'indirizzamento diretto, cioè un array di dimensione  $m$  dove ogni chiave  $k$  è mappata alla cella corrispondente

### Costi di un dizionario

Search  $T(n) = O(1)$   
 Insert  $T(n) = O(1)$   
 Delete  $T(n) = O(1)$

### Tabella Hash/Hash Table

- Ho una funzione hash  $h(k) \in N$  che converte una chiave di qualsiasi tipo in un intero tra 0 e  $m$
- Se la dimensione  $m$  della mia tabella è tale che  $m \ll |U|$ , ci sono sicuramente chiavi tali che  $h(k_1) = h(k_2)$ : in questo caso ho delle collisioni.

### Concatenamento

- *Idea*: gli oggetti che vengono mappati sullo stesso slot vengono messi in una lista  $L$  concatenata.

Search  $T(n) = |M[h(k)]|$   $M$  è la hash table  
 Insert  $T(n) = O(1)$   $x \notin M$   
 Delete (1)  $T(n) = O(1)$  double-linked  $L$   
 Delete (2)  $T(n) = |M[h(k)]|$  single-linked  $L$

- Nel caso pessimo si ha la complessità di una ricerca in una lista di  $n$  elementi, cioè  $T(n) = O(n)$ .
- $\alpha = \frac{n}{m}$  è il *fattore di carico*
- $0 \leq n \leq |U| \rightarrow 0 \leq \alpha \leq \frac{|U|}{m}$

### Ipotesi di hashing uniforme semplice

- Ogni chiave ha la stessa probabilità  $\frac{1}{m}$  di finire in una qualsiasi delle  $m$  celle di  $T$ , indipendentemente dalle chiavi inserite. La lunghezza media di una lista è quindi:

$$E(n_j) = \frac{1}{m} \sum_{i=1}^m n_i = \frac{n}{m} = \alpha$$

- Il tempo medio per cercare una chiave  $k$ , sia che sia presente o meno nella lista, è

$$T(n) = \Theta(1 + \alpha) \rightarrow T(n) = O(1) \quad (\text{in media})$$

### Indirizzamento aperto

- La tabella contiene tutte le chiavi, senza memoria aggiuntiva  $\rightarrow \alpha \leq 1$
- *Idea*: si calcola l'indice dello slot in cui memorizzare l'oggetto. Se è già occupato, se ne cerca un altro libero.
- Quando si cancella un oggetto, si inserisce nello slot un valore convenzionale come **DELETED**. La complessità dipende però dalla sequenza di ispezione della funzione di hash anziché dal fattore di carico

### Tecniche di ispezione

Introduciamo una seconda funzione di hash  $h'(k)$ .

- Lineare
  - $h(k, i) = (h'(k) + i) \bmod m$

- Soffre del fenomeno dell'addensamento (clustering) primario, cioè lunghe sequenze di celle consecutive, che aumentano il tempo di ricerca
- Quadratica
  - $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$
  - $c_1$  e  $c_2 (\neq 0)$  sono costanti ausiliarie, scelte opportunamente
  - Soffre del fenomeno dell'addensamento secondario: chiavi con la stessa posizione iniziale danno luogo alla stessa sequenza d'ispezione
- Doppio hashing
  - $h(k, i) = (h_1(k) + i h_2(k)) \bmod m$
  - $h_1$  e  $h_2$  sono funzioni di hash ausiliarie
  - Il numero di sequenze d'ispezione è ora  $\Theta(m^2)$ , perchè ogni coppia  $(h_1(k), h_2(k))$  produce una sequenza di ispezione distinta

## Albero Binario/Binary Tree

- È composto da 3 elementi:
  - un nodo detto *radice*
  - un albero binario detto *sottoalbero sinistro*
  - un albero binario detto *sottoalbero destro*
- A ogni nodo è associata una chiave

## Binary Search Tree

- Per tutti i nodi  $x$  del BST, se  $l$  è un nodo nel sottoalbero sinistro, allora  $l.\text{key} \leq x.\text{key}$ ; se  $r$  è un nodo del sottoalbero destro, allora  $r.\text{key} \geq x.\text{key}$

## Attraversamento simmetrico/(in order)

Restituisce i nodi ordinati se l'albero è un BST:

- Prima si visita il sottoalbero sinistro e si restituiscono i suoi nodi
- Si restituisce la radice
- Si visita il sottoalbero destro e si restituiscono i suoi nodi

## Successore

Il *successore* di un oggetto  $x$  in un BST è l'elemento  $y$  tale che  $y.\text{key}$  è la più piccola tra le chiavi che sono più grandi di  $x.\text{key}$ , cioè è il minimo del sottoalbero destro di  $x$ . Se il sottoalbero di  $x$  è vuoto, il successore di  $x$  è il primo elemento  $y$  che si incontra risalendo nell'albero da  $x$  tale che  $x$  è nel sottoalbero sinistro di  $y$  (salgo dai "right" finchè non risalgo da un "left")

## Predecessore

Il *predecessore* di un oggetto  $x$  in un BST è l'elemento  $y$  tale che  $y.\text{key}$  è la più grande tra le chiavi che sono più piccole di  $x.\text{key}$ , cioè è il massimo del sottoalbero sinistro di  $x$ . Se il sottoalbero di  $x$  è vuoto, il predecessore di  $x$  è il primo elemento  $y$  che si incontra risalendo nell'albero da  $x$  tale che  $x$  è nel sottoalbero destro di  $y$  (salgo dai "left" finchè non risalgo da un "right")

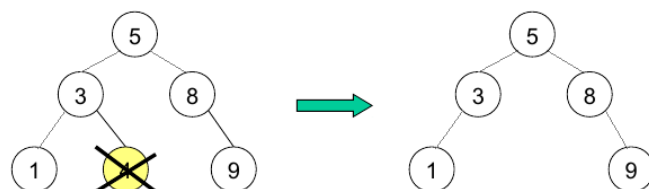
## Inserimento

Scendo nell'albero finchè non si raggiunge il posto in cui il nuovo elemento deve essere inserito, e lo si aggiunge come foglia.

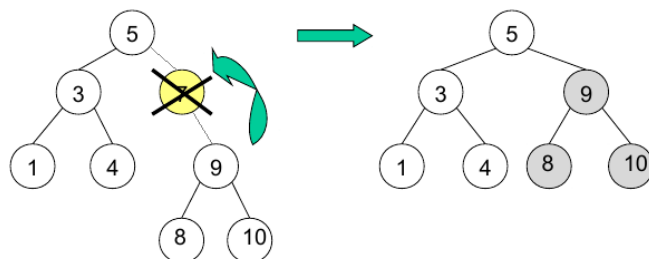
## Cancellazione

Ci sono 3 possibili casi per cancellare un nodo  $z$ :

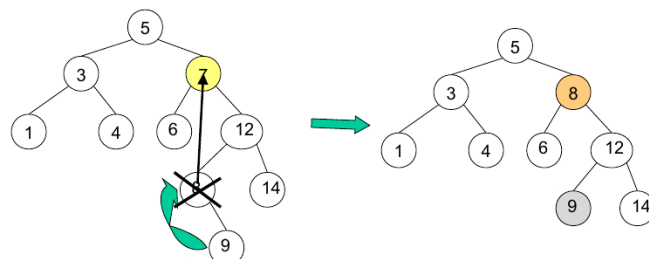
- Il nodo  $z$  non ha sottoalberi: si mette a **nil** il puntatore del padre di  $z$
- Il nodo  $z$  ha 1 sottoalbero: bisogna spostare il sottoalbero di  $z$  in su di un livello.
- Il nodo  $z$  ha 2 sottoalberi: bisogna trovare il successore di  $z$ , copiare la chiave del successore in  $z$  e cancellare il successore.



Il nodo  $z$  non ha sottoalberi



Il nodo  $z$  ha un sottoalbero



Il nodo  $z$  ha due sottoalberi

## Complessità degli alberi binari

Un albero si dice *bilanciato* se per ogni nodo  $x$  le altezze dei due sottoalberi di  $x$  differiscono al massimo di 1.  $h$  è l'altezza dell'albero.

$$h = \begin{cases} \Theta(\log(n)) & \text{albero bilanciato} \\ \Theta(n) & \text{caso pessimo} \end{cases}$$

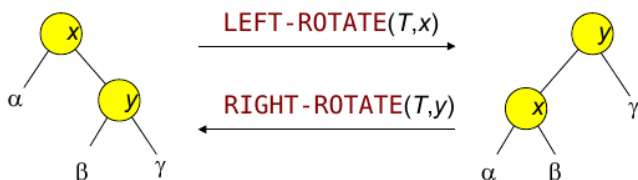
Operazione	Complessità
In-order walk	$T(n) = \Theta(n)$
Search	$T(n) = O(h)$
Max/Min	$T(n) = O(h)$
Successor	$T(n) = O(h)$
Insert	$T(n) = O(h)$
Delete	$T(n) = O(h)$

## Alberi rosso-neri

- Sono BST abbastanza bilanciati, tali che  $h = O(\log(n))$  ed è possibile realizzare tutte le operazioni più importanti in  $T(n) = O(\log(n))$
- Un ramo non è mai più lungo del doppio di un altro
- Un BST è un albero RB se soddisfa le seguenti proprietà:

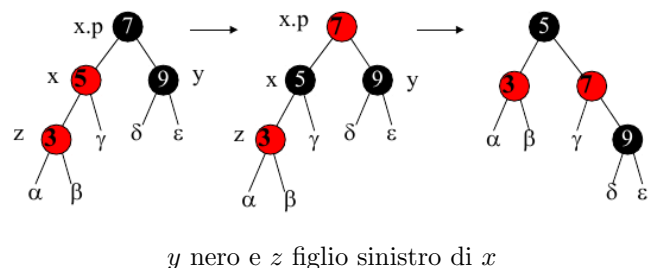
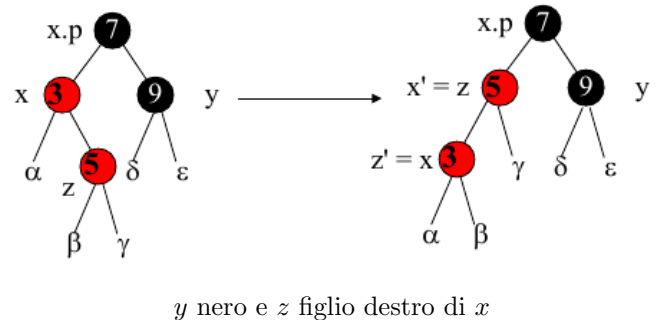
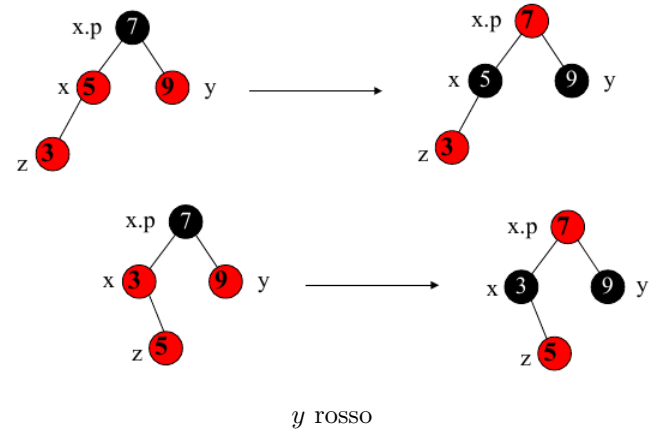
1. Ogni nodo è *rosso* oppure *nero*
2. La radice è *necessari*
3. Le foglie (**nil**) sono tutte nere
4. I figli di un nodo rosso sono entrambi neri
5. Per ogni nodo  $x$  tutti i cammini da  $x$  alle foglie sue discendenti contengono lo stesso numero  $bh(x)$  di nodi neri.  $bh(x)$  è l'*altezza nera* del nodo  $x$ , che non viene conteggiato nella sua altezza

- Per comodità le foglie **nil** si usa un nodo sentinella  $T.nil$
- Un albero RB con  $n$  nodi interni (cioè con chiavi), ha altezza  $h \leq 2\log_2(n+1)$ . Pertanto, **SEARCH**, **MINIMUM**, **MAXIMUM**, **SUCCESSOR**, **PREDECESSOR** richiedono tempo  $T(n) = O(\log(n))$
- **INSERT** e **DELETE** sono modificate per poter rispettare le proprietà degli alberi RB, ma richiedono sempre  $T(n) = O(\log(n))$
- Per realizzare inserimenti e cancellazioni si usa il meccanismo delle rotazioni, con **ROTATE-LEFT** e **ROTATE-RIGHT**



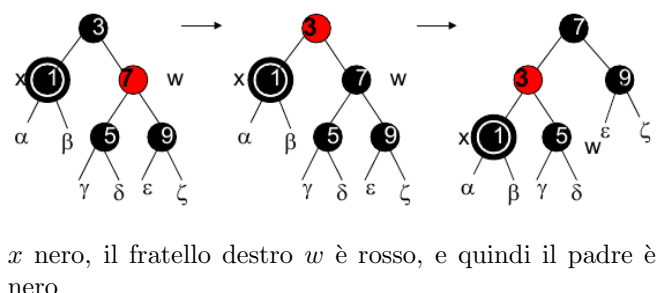
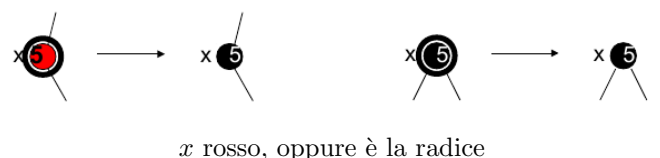
## Inserimento

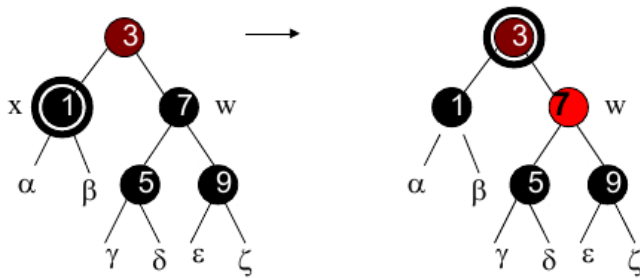
Analogo a quello dei BST, ma per ristabilire le proprietà degli alberi RB si usa **RB-INSERT-FIXUP**, che viene invocato sempre su un nodo  $z$  tale che  $z.color = RED$ , e al massimo  $O(\log(n))$  volte.



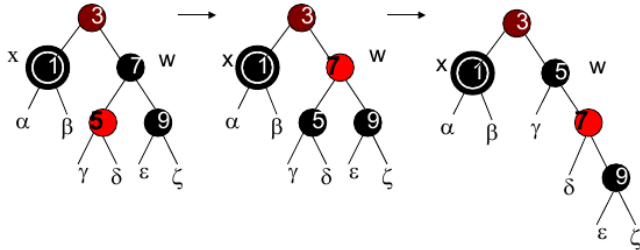
## Cancellazione

Analogo a quello dei BST. Se viene cancellato un nodo rosso non c'è bisogno di modificare i colori dei nodi, e quello che prende il suo posto è per forza nero.

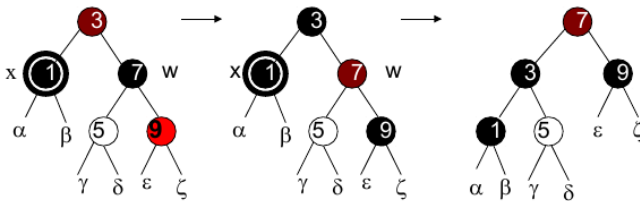




$x$  nero, il fratello destro  $w$  è nero con figli neri



$x$  nero, il fratello destro  $w$  è nero con figlio sinistro rosso e figlio destro nero



$x$  nero, il fratello destro  $w$  è nero con figlio destro rosso

## Grafi

Un grafo è una coppia  $G = (V, E)$  in cui:

- $V$  è un insieme di nodi detti vertici
- $E$  è un insieme di archi detti lati/edges. Un arco è una connessione fra due vertici, detti quindi adiacenti. Un arco è una coppia  $(u, v) \in E \subseteq V^2$
- $|V|$  è il numero di vertici nel grafo, mentre  $|E|$  è il numero di archi. Inoltre  $0 \leq |E| \leq |V|^2$

I grafi esistono orientanti e *non orientati*.

## Rappresentazione in memoria

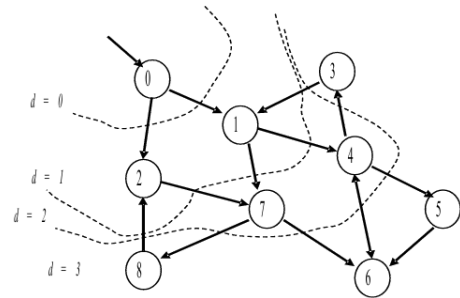
- *Liste di adiacenza*: array di liste, una per nodo. Per ogni nodo, la lista contiene i nodi adiacenti ad esso. Ha complessità spaziale  $S_L = \Theta(|V| + |E|)$
- *Matrice di adiacenza*:  $m_{ij} = 1$  se c'è un arco dal nodo  $i$  al nodo  $j$ , altrimenti è 0. Ha complessità spaziale  $S_M = \Theta(|V|^2)$

## Visita in ampiezza/Breadth-First Search

Ha complessità  $T_{BFS} = O(|V| + |E|)$ .

- Usa una politica FIFO (coda)
- Visito tutti i nodi a distanza 1 da  $s$  (sorgente)
- Quando visito un nodo  $u$ , salvo la sua distanza da  $s$  in un attributo  $u.dist$
- Coloro i nodi che visito (*bianco* se è da visitare, *grigio* se è già stato visitato, ma bisogna completare la visita dei nodi adiacenti, *nero* dopo che abbiamo visitato tutti i suoi nodi adiacenti)

- All'inizio tutti i nodi sono bianchi tranne  $s$  che è grigio.
- I nodi da visitare sono in una coda (inizialmente solo  $s$ )
- A ogni iterazione, cancello dalla coda un elemento  $u$  e ne visitiamo i nodi adiacenti che sono ancora bianchi (la cui distanza da  $s$  sarà  $u.dist+1$ )



BFS

## Visita in profondità/Depth-First Search

Ha complessità  $T_{DFS} = O(|V| + |E|)$ .

- Usa una politica LIFO (stack)
- Tiene traccia di quando un nodo è aggiunto alla pila ( $u.d$ ) e di quando viene tolto ( $u.f$ )

## Ordinamento Topologico

Dato un grafo orientato aciclico (DAG), un ordinamento topologico è un ordinamento lineare dei nodi del grafo tale che, se nel DAG c'è un arco  $(u, v)$ , allora il nodo  $u$  precede  $v$  nell'ordinamento (cioè che le frecce "vanno solo in una direzione"). L'ordinamento ottenuto rispetta la precedenza tra nodi/eventi.

- Visito il DAG con un algoritmo DFS
- quando coloro un nodo  $u$  di nero, inserisco  $u$  in testa alla lista
- Una volta visitati tutti i nodi, ottengo un ordinamento topologico

Ha la stessa complessità di DFS, cioè  $T_{TS} = O(|V| + |E|)$ .

## Cammini Minimi

$$\delta(u, v) = \begin{cases} \min \{w(p) : u \xrightarrow{p} v\} & \text{se } \exists \text{ path } p \text{ da } u \text{ a } v \\ \infty & \text{altrimenti} \end{cases}$$

- $w : E \rightarrow R$  è la *funzione di peso*
- $d[v] = \delta(s, v)$  è detto *stima di cammino minimo* dalla sorgente  $s$  a  $v$
- All'inizio vale  $d[v] = \infty$
- viene ridotto col procedere dell'algoritmo, ma  $d[v] \leq \delta(s, v)$
- $P_i[v]$  è il *predecessore* di  $v$  nel cammino minimo da  $s$ , se non esiste è **nil**

## Rilassamento di un lato

- se  $d[v] > d[u] + w(u, v)$  allora  $d[v] = d[u] + w(u, v)$  e  $P_i[v] = u$



## Bellman-Ford

- Rilasso un passo alla volta partendo da  $s$
- Ad ogni passo avanzo nei cammini
- Al  $|V|$ -esimo passo sicuramente avrò toccato tutti i nodi raggiungibili
- *non* converge se ci sono cicli negativi

```

BELLMAN-FORD(adj, s)
    V = vector of nodes of adj
    allocate vectors d, Pi of size adj

    for i = 0 to |adj| - 1
        d[i] =  $\infty$ 
    d[s] = 0
    repeat for |V| - 1 times
        for u in V
            for v in adj[u]
                RELAX(u, v, adj, d, Pi)
    return [d, Pi]

```

## Pseudocodice

### Ricerca Binaria Ricorsiva

```

BIN-SEARCH-REC(array, p, r, v)
    if p > r
        return nil
    if v < A[p] or v > A[r]
        return nil
    q = (p+r)/2
    if A[q] == v
        return q
    if A[q] > v
        return BIN-SEARCH-REC(A, p, q-1, v)
    else
        return BIN-SEARCH-REC(A, q+1, r, v)

```

### In Order Tree Walk

```

INORDER-TREE-WALK(x)
    if x != nil
        INORDER-TREE-WALK(x.left)
        print x.key
        INORDER-TREE-WALK(x.right)

```

### Tree Search

```

TREE-SEARCH(x, k)
    if x == nil or k == x.key
        return x
    if k < x.key
        return TREE-SEARCH(x.left, k)
    else
        return TREE-SEARCH(x.right, k)

```

### Tree Minimum

```

TREE-MINIMUM(x)
    while x.left != nil
        x = x.left
    return x

```

## Tree Maximum

```

TREE-MAXIMUM(x)
    while x.right != nil
        x = x.right
    return x

```

## Tree Successor

```

TREE-SUCCESSOR(x)
    if x.right != nil
        return TREE-MINIMUM(x.right)
    y = x.p
    while y != nil and x = y.right
        x = y
        y = y.p
    return y

```

## Tree Insert

```

TREE-INSERT(T, z)
    y = nil
    x = T.root
    while x != nil
        y = x
        if z.key < x.key
            x = x.left
        else x = x.right
    z.p = y
    if y == nil
        T.root = z
    else if z.key < y.key
        y.left = z
    else
        y.right = z

```

## Tree Delete

$y$  e' il nodo da eliminare,  $x$  e' quello con cui lo sostituiamo

```

TREE-DELETE(T, z)
    if z.left == nil or z.right == nil
        y = z
    else
        y = TREE-SUCCESSOR(z)
    if y.left != nil
        x = y.left
    else
        x = y.right
    if x != nil
        T.root = x
    else if y == y.p.left
        y.p.left = x
    else
        y.p.right = x
    if y != z
        z.key = y.key
    return y

```

## Rotazione a sinistra

```

ROTATE-LEFT(T, z)
  y = x.right
  x.right = y.left

  if y.left != T.nil
    y.left.p = x
  y.p = x.p
  if x.p == T.nil
    T.root = y
  else if x == x.p.left
    x.p.left = y
  else
    x.p.right = y
  y.left = x
  x.p = y

```

## RB Tree Insert

```

RB-TREE-INSERT(T, z)
  y = T.nil
  x = T.root
  while x != T.nil
    y = x
    if z.key < x.key
      x = x.left
    else x = x.right
  z.p = y
  if y == T.nil
    T.root = z
  else if z.key < y.key
    y.left = z
  else
    y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED
  RB-INSERT-FIXUP(T, z)

```

## RB Insert Fixup

```

RB-INSERT-FIXUP(T, z)
  if z == T.root
    T.root.color = BLACK
  else
    x = z.p
    if x.color == RED
      if x == x.p.left
        y = x.p.right
        if y.color == RED
          x.color = BLACK
          y.color = BLACK
          x.p.color = RED
          RB-INSERT-FIXUP(T, x.p)
        else if z == x.right
          z = x
          LEFT-ROTATE(T, z)
          x = z.p
          x.color = BLACK
          x.p.color = RED
          RIGHT-ROTATE(T, x.p)
        else //...
          (come il contenuto dell'if
          associato, scabio left e right)

```

## RB Delete

```

RB-DELETE(T, z)
  if z.left == T.nil
    or z.right == T.nil
    y = z
  else
    y = TREE-SUCCESSOR(z)
  if y.left != T.nil
    x = y.left
  else
    x = y.right
  x.p = y.p
  if y.p == T.nil
    T.root = x
  else if y == y.p.left
    y.p.left = x
  else
    y.p.right = x
  if y != z
    z.key = y.key
  if y.color == BLACK
    RB-DELETE-FIXUP(T, x)
  return y

```

## RB Delete Fixup

```

RB-DELETE-FIXUP(T, x)
  if x.color == RED
    or x.p == T.nil
    x.color = BLACK
  else if x == x.p.right
    w = x.p.right
    if w.color == RED
      w.color = BLACK
      x.p.color = RED
      LEFT-ROTATE(T, x.p)
      w = x.p.right
    if w.left.color == BLACK
      and w.right.color == BLACK
      w.color = RED
      RB-DELETE-FIXUP(T, x.p)
    else if w.right.color == BLACK
      w.left.color = BLACK
      w.color = RED
      ROTATE-RIGHT(T, w)
      w = x.p.right
    w.color = x.p.color
    x.p.color = BLACK
    ROTATE-LEFT(T, x.p)
  else
    // come le righe 4-21, scambio
    // "left" ↔ "right"

```

## Breadth First Search

```

BFS(G, s)
  for each  $u \in G.V - \{s\}$ 
    u.color = WHITE
    u.dist =  $\infty$ 
  s.color = GREY
  s.dist = 0
  Q = nil
  ENQUEUE(Q, s)
  while Q != nil
    u = DEQUEUE(Q)
    for each  $v \in u.Adj$ 
      if v.color == WHITE
        v.color = GREY
        v.dist = u.dist + 1
        ENQUEUE(Q, v)
  u.color = BLACK

```

## Depth First Search

```

DFS(G)
  for each  $u \in G.V$ 
    u.color = WHITE
  time = 0
  for each  $u \in G.V$ 
    if u.color = WHITE
      DFS-VISIT(u)

```

```

DFS-VISIT(u)
  u.color = GREY
  time = time + 1
  u.d = time
  for each  $v \in u.Adj$ 
    if v.color = WHITE
      DFS-VISIT(v)
  u.color = BLACK
  u.f = time = time + 1

```

## Ordinamento Topologico

```

TOPOLOGICAL-SORT(G)
  L = nil
  for each  $u \in G.V$ 
    u.color = WHITE
  for each  $u \in G.V$ 
    if u.color = WHITE
      TOPSORT-VISIT(L, u)
  return L

```

```

TOPSORT-VISIT(L, u)
  u.color = GREY
  for each  $v \in u.Adj$ 
    if v.color = WHITE
      TOPSORT-VISIT(L, v)
  ... // crea l'elemento di lista x
  x.key = u
  LIST-INSERT(L, x)
  u.color = BLACK

```