Complessità

è una funzione che descrive la dimensione di un certo problema contando il numero di istruzioni da eseguire per risolverlo e quindi estraendo il tempo di esecuzione; è una funzione generica, in quanto necessita di conoscere l'algoritmo risolutivo del problema, che ovviamente non è sempre lo stesso.

Contando tutte le istruzioni, moltiplicandole per il numero di cicli e considerando sempre il branch più lungo delle condizioni, si nota che man mano che i dati crescono di numero, tutte le costanti prima descritte vengono a perdere di valore rispetto alle grandezze moltiplicative: questo ci porta a non considerare mai le costanti e a usare invece gli **ordini di grandezza**.

Ordini di grandezza

Da un certo punto $\bar{n}$ in poi

- l'ordine di f è minore o uguale dell'ordine di g

  $f \in O(g) \Leftrightarrow \exists c > 0 \exists \bar{n} \forall n > \bar{n}, f(n) \leq c \cdot g(n)$

- l'ordine di f è maggiore o uguale dell'ordine di g

  $f \in \Omega(g) \Leftrightarrow \exists c > 0 \exists \bar{n} \forall n > \bar{n}, f(n) \geq c \cdot g(n)$

- l'ordine di f è uguale all'ordine di g

  $f \in \Theta(g) \Leftrightarrow \exists c > 0 \exists \bar{n} \forall n > \bar{n}, f(n) = c \cdot g(n) \Leftrightarrow f \in O(g) \cap f \in \Omega(g)$

<u>Proprietà</u>

- $f \in O(g) \Leftrightarrow g \in \Omega(f)$
- Se $f_1 \in O(g)$, $f_2 \in O(g)$ allora $(f_1 + f_2) \in O(g)$
- Se $f \in O(g_1)$ allora $f \in O(g_1 + g_2)$

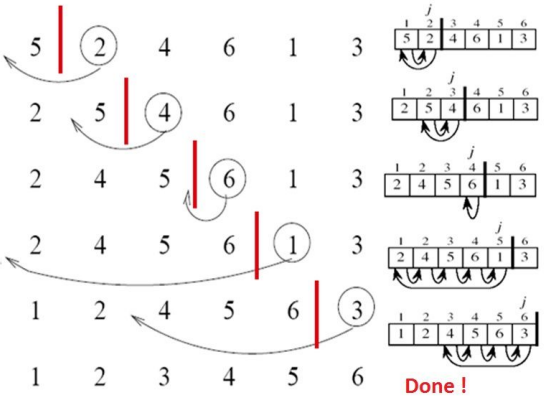
<u>Master's theorem</u>

Se $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ con $a \geq 1, b \geq 1$ si ha che

- se $f(n) \in O(n^{\log_b a - \varepsilon})$ con $\varepsilon > 0$ allora $T(n) \in \Theta(n^{\log_b a})$
- se $f(n) \in \Theta(n^{\log_b a})$ allora $T(n) \in \Theta(f(n) \cdot ln(n))$
- se $a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n)$ con $c < 1, \forall n > \bar{n}$ e se $f(n) \in \Omega(n^{\log_b a + \varepsilon})$ con $\varepsilon > 0$ allora $T(n) \in \Theta(f(n))$

Algoritmi di Ordinamento

Insertion Sort



L'algoritmo agisce su due sottosequenze, una ordinata, prima dell'invariante *i,* e una non ordinata, dopo l'invariante *i*. L'algoritmo prende un elemento dalla sottosequenza non ordinata, e lo va a posizionare nella posizione corretta nella sotto sequenza ordinata.
Termina velocemente in caso di sequenze già parzialmente ordinate.

- Stabile.
- In loco.
- Caso pessimo: $\Theta(n^2)$

```
INSERTION_SORT(A):
   for j = 2 to length[A]
       key = A[i]
       i = j-1
       while i > 0 and A[i] > key
           A[i + 1] = A[i]
           i = i-1
       A[i+1] = key;
```

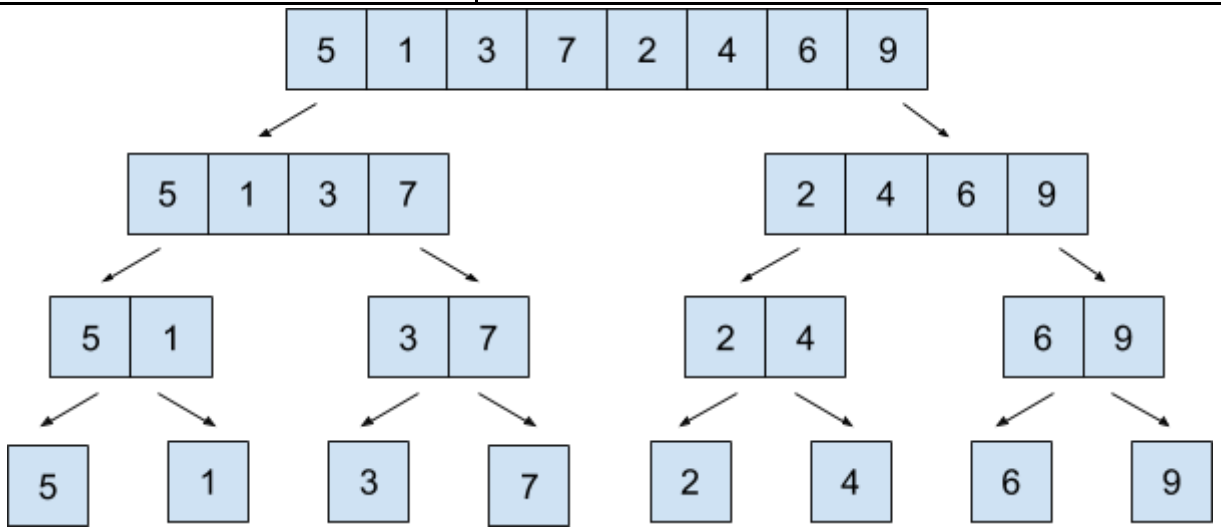
Merge Sort

Divide l'array in due sotto-array e riapplica l'algoritmo sulle due metà; il caso base ordina gli elementi.

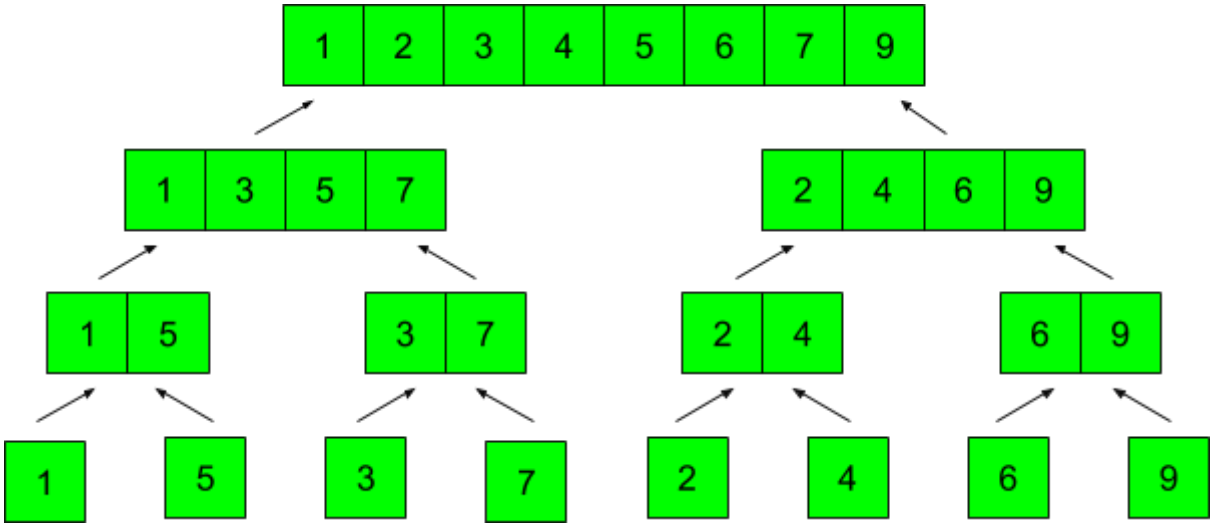
- Stabile.
- NON in loco.
- Caso pessimo: $\Theta(n \cdot logn)$

```
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):
    i = 0, j = p, k = q + 1
    while (j <= q or k <= r)
        if j <= q and (k > j or A[j] <= A[i])
            B[i] = A[j], j++
        else
            B[i] = A[k], k++
        i++
    for i = 0 to (r - p + 1)
        A[p + i] = B[i]
```



ARRAY PRIMA DI ESSERE ORDINATO



ARRAY ORDINATO

Heap Sort

Definizioni:

- **Livello Completo:** un livello si dice completo quando contiene il massimo numero di nodi.

- **Albero Completo:** Un albero si dice completo quando ogni livello contiene il numero massimo di nodi.
- **Albero Semicompleto:** Quando l'albero non è completo, ma ogni livello è stato riempito dalla radice verso le foglie e da sinistra verso destra.
    - Può essere rappresentato da un array riempito prendendo gli elementi in ordine partendo dalla radice verso le foglie e da sinistra verso destra.

Proprietà di uno heap:

- ❏ È un **albero binario semicompleto**(=riempito da sinistra verso destra) con chiave su cui è definita una **relazione di ordinamento**.
- ❏ Per ogni **nodo**, la sua **chiave è maggiore o uguale** rispetto alle chiavi dei **figli**.
- ❏ Il numero di livelli di un albero è **esponenziale in profondità**.

- **heapify()**: procedura che mantiene o corregge le proprietà di un' heap.
    - Complessità: $\Theta(n \log n)$.
- **buildHeap()**: costruisce un' heap a partire da un array non ordinato.
    - Complessità: $\Theta(n)$.
- **heapsort()**: ordina un array
    - non è stabile
    - In loco.
    - Complessità: $\Theta(n \log n)$.

```
HEAPIFY(A, i):
    //trovo il nodo più grande
    l = LEFT[i], r = RIGHT[i]
    if l <= HEAP_SIZE[i] and A[l] > A[i]:
        LARGEST = l
    else:
        LARGEST = i
    if r <= HEAP_SIZE[i] and A[i] > A[LARGEST]:
        LARGEST = r
    if LARGEST != i:
        //scambio con root e porto quindi
        //il max in ultima posizione
        SWAP(A[i], A[LARGEST])
        HEAPIFY(A, LARGEST)
```

```
BUILD_HEAP(A):
    HEAP_SIZE[A] = LENGTH[A]
    for i = LENGTH[A]/2 to 1:
        //ordino
        HEAPIFY(A, i)
```

```
HEAP_SORT(A):
    //costruisci lo heap
    BUILD_HEAP(A)
    for i = LENGTH[A] to 2:
        //scambia e elimina
        SWAP(A[i], A[1])
        HEAP_SIZE[A]--
        //porta in giù
        HEAPIFY(A, 1)
```

Quick Sort

È un algoritmo del tipo divide-et-impera, che dato un array, prende il primo elemento e mette tutti gli elementi minori a sinistra e tutti i maggiori a destra, poi spezza l'array in due sotto-array in base alla posizione in cui è finito il primo elemento e ripete l'algoritmo sui due sotto-array. In una versione più evoluta, la partition può anche prendere un elemento casuale (non per forza il primo o l'ultimo) per cercare di ridurre al minimo i casi peggiori. Inoltre è possibile rimuovere la ricorsione di coda di Quicksort.

```
PARTITION(A, p, r):
    x = A[p]
    i = p-1
    j = r+1
    while true:
        j--
       repeat:
           j--
       until A[j] <= x
       repeat:
           i++
       until A[i] >= x
        if i < j:
             SWAP(A[i], A[j])
        else:
           return j
```

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

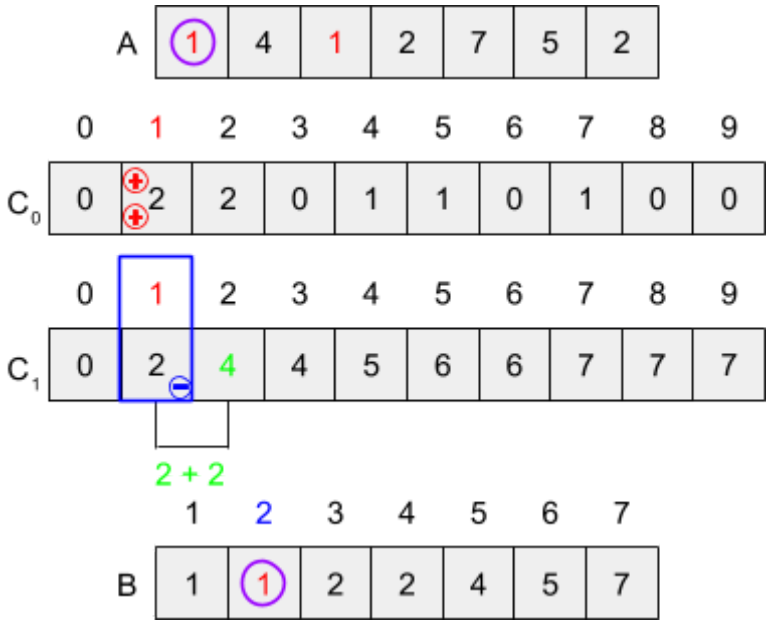
- NON stabile.
- In loco.
- Caso pessimo: $O(n^2)$
- Caso migliore: $\Theta(n \log n)$
- Caso medio con RandomizePartition: $O(n \log n)$

Name	Best	Average	Worst	Stable	Method
Quicksort	$n \log n$ variation is n	$n \log n$	n^2	Typical in-place sort is not stable; stable versions exist.	Partitioning
Merge sort	$n \log n$	$n \log n$	$n \log n$	Yes	Merging
Heapsort	$n \log n$	$n \log n$	$n \log n$	No	Selection
Insertion sort	n	n^2	n^2	Yes	Insertion

Ordinare un array in tempo inferiore di nlog(n) è possibile: basta infatti effettuare confronti limitati o addirittura nessun confronto tra coppie.

Counting Sort

Ordina un array di numeri interi [1, …, k], contando prima la cardinalità di ogni elemento e poi posizionando ogni elemento in ordine tante volte quanto vale la sua cardinalità, sperando di ottenere un algoritmo con complessità lineare. Consideriamo ad esempio un range [0..9]

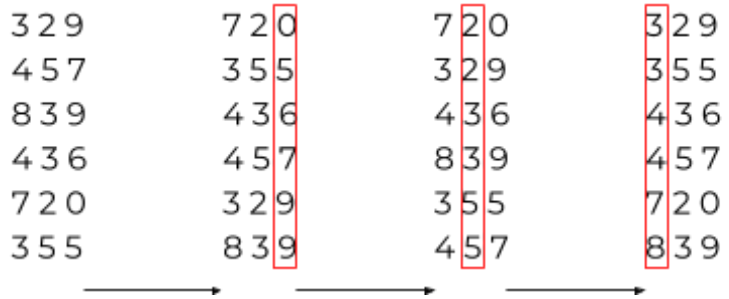



```
COUNTING_SORT(A, k):
    for i = 1 to k:
        count[i] = 0
    for j = 1 to LENGTH(A):
        count[A[j]]++
    for i = 2 to k:
        count[i] = count[i] + count[i - 1]
    for j = LENTH(A) down to 1:
         B[count[A[j]]] = A[j]
         count[A[j]]--
    return B
```

- Stabile.
- NON in loco.
- $\Theta(n+k)$

Radix Sort

Dato un array di interi di w bit, partendo a guardare dal bit meno significativo, smista in due bucket in dividendo gli 0 dagli 1, poi impila i bucket in ordine e reitera passando al prossimo bit verso sinistra. L'algoritmo si basa totalmente sulla sua stabilità, infatti in questo modo, mantenendo l'ordine precedente e impilando i bucket appena ordinati, si ottiene, arrivati al bit più significativo, un array ordinato. L'ordinamento di ogni array viene fatto col counting sort [0..9]



- Stabile
- NON in loco
- Caso pessimo: $O(wn)$

Bucket Sort

Dato un array di numeri razionali appartenenti al dominio $[0,1)$, essi vengono smistati in 10 bucket che dividono il dominio uniformemente (bucket da 0.00 a 0.09, da 0.10 a 0.19, …, da 0.90 a 0.99) e poi ogni bucket viene ordinato con un altro algoritmo; alla fine, concatenando i bucket dal primo all'ultimo si ottiene l'array ordinato. Utile se conosco .a densità di un problema

Name	Best	Average	Worst	Stable
Counting sort	—	$n+r$	$n+r$	Yes
LSD Radix Sort	—	$n \cdot \frac{k}{d}$	$n \cdot \frac{k}{d}$	Yes
Bucket sort (uniform keys)	—	$n+k$	$n^2 \cdot k$	Yes

- Stabile
- NON in loco
- Caso pessimo: $O(n^2)$
- Caso migliore: $\Theta(n)$

Algoritmi di Selezione

PROBLEMA DELLA SELEZIONE

Permettono di trovare l'elemento all'indice indicato di un array non per forza ordinato.
Se prendo in input un array con un numero n di elementi, su cui è definita una relazione di ordinamento totale, posso accedere in tempo costante all'elemento in posizione i.
Problema:

- Ordinando un array con un algoritmo di ordinamento, quindi almeno in nlogn, il problema allora appartiene in $O(nlogn)$, in quanto l'accesso all'elemento in posizione i-esima è costante.
- Se prendo un array non ordinato, il problema rientra in $\Omega(n)$, perchè devo confrontare tutti gli elementi dell'array.

```
RAND_SELECT(A, p, r, i):
    if p == r:
        return A[p]
    else:
        q = RANDOMIZE_PARTITION(A, p, r)
        if s <= k:
            return RAND_SELECT(A, p, q, r)
        else:
            return RAND_SELECT(A, q+1, r, i-k)
```

ALGORITMO PER LA RICERCA IN TEMPO LINEARE (“SELECT” select)

1) Dividi l’array in $\lfloor n/5 \rfloor$ gruppi di 5 elementi più un gruppo di al più 5 elementi.
2) Calcola il mediano di ogni gruppo.
3) Con `select` ricorsiva calcola il mediano dei mediani(x).
4) Partiziona su x e su k la cardinalità di sx
5) `return (i <= k) ? select(sx, i) : select(dx, i)`

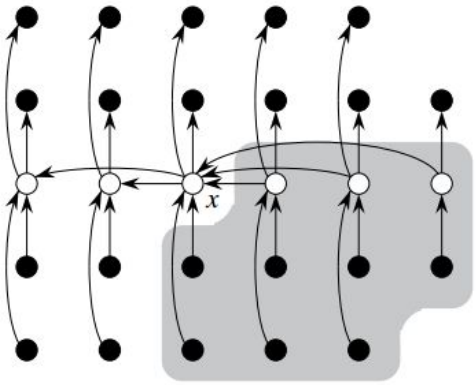


Figure 9.1 Analysis of the algorithm SELECT. The n elements are represented by small circles, and each group of 5 elements occupies a column. The medians of the groups are whitened, and the median-of-medians x is labeled. (When finding the median of an even number of elements, we use the lower median.) Arrows go from larger elements to smaller, from which we can see that 3 out of every full group of 5 elements to the right of x are greater than x, and 3 out of every group of 5 elements to the left of x are less than x. The elements known to be greater than x appear on a shaded background.

Strutture dati

Stack (LIFO)

Rappresenta una pila, dalla quale gli elementi vengono estratti in ordine inverso a quello di arrivo, a partire dall'ultimo inserito.
Viene solitamente implementato usando un array S con un attributo supplementare S.TOP che contiene l'indice dell'ultimo elemento inserito.

Interfaccia

- **`EMPTY(S)`** : svuota lo stack
- **`IS_EMPTY(S)`** : controlla se lo stack è vuoto
- **`PUSH(S,x)`** : inserisce in testa nello stack l'elemento x
- **`POP(S)`** **`x`** : estrae dallo stack l'elemento in testa x
- **`TOP(S)`** **`x`** : legge l'elemento in testa x SENZA ESTRARLO

Queue (FIFO)

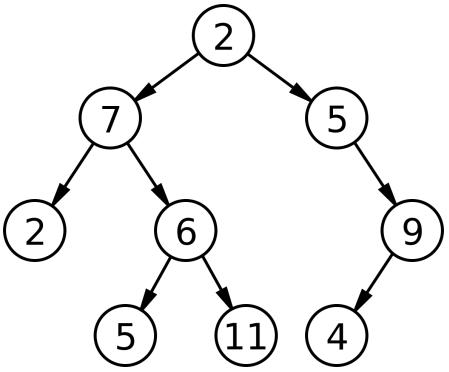
Rappresenta una coda, dalla quale gli elementi vengono estratti in ordine di arrivo, a partire dal primo arrivato.
Viene implementato usando un array circolare Q, che permette di inserire elementi ricominciando dall'inizio dell'array nel caso in cui si sia raggiunta la fine.

Interfaccia

- **`EMPTY(Q)`** : svuota la queue
- **`IS_EMPTY(Q)`** : controlla se la queue è vuota
- **`ENQUEUE(Q,x)`** : inserisce in coda nella queue l'elemento x
- **`HEAD(Q)`** **`x`** : estrae dalla queue l'elemento x, il primo arrivato rispetto agli altri
- **`DEQUEUE(Q)`** : toglie il primo elemento arrivato SENZA RITORNARLO

Albero binario di ricerca

È un albero binario con n elementi (ogni nodo ha al massimo due figli, uno destro e uno sinistro) dove il figlio destro è maggiore o uguale al padre e il figlio sinistro è minore o uguale al padre.



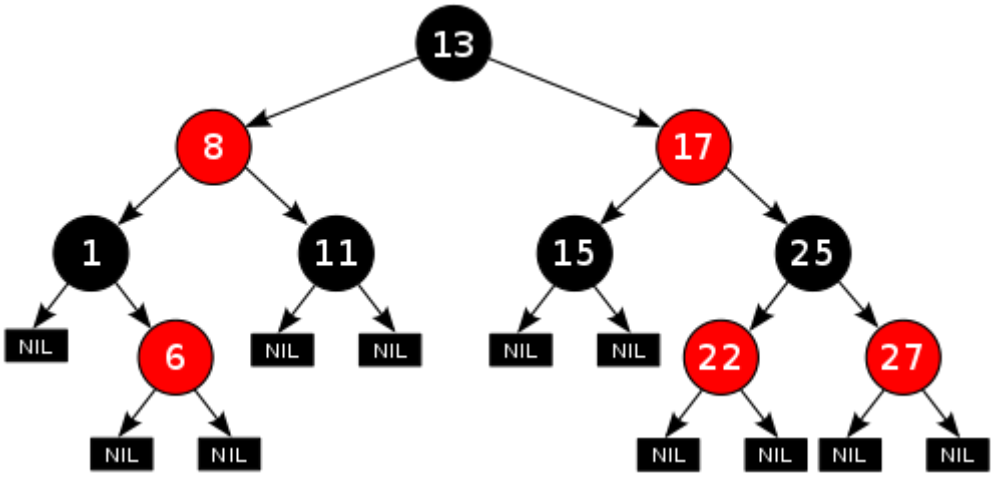

```
// x : puntatore alla radice
// k : valore chiave da cercare

SEARCH(x,k):
    if x == null:
        return null
    if x.key == k:
        return x
    if x.key > k:
        return SEARCH(x.left, k)
    else:
        return SEARCH(x.right, k)
```

Operazioni base

- Inserimento: $\Theta(logn)$
- Bilanciamento: $\Theta(n)$
- Ricerca: $\Theta(logn)$

RB-alberi di ricerca



Un RB-albero è un albero di ricerca con le seguenti proprietà aggiuntive

Proprietà:

1) Ogni nodo può essere o **rosso** o **nero.**
2) Ogni foglia è nera.
3) I figli di un rosso sono neri.
4) Ogni cammino radice-foglia contiene lo stesso numero di nodi neri.
5) Ogni foglia è rappresentata puntando ad un solo nodo sentinella nero, passato per indirizzo, che contiene la chiave null.

Altezza NERA

> **bh(x)**: è il numero di nodi neri di un cammino qualsiasi (grazie alla proprietà 4) da x a una foglia escludendo x.

Dato un nodo x, nel sottoalbero radicato in x ci sono almeno $\mathbf{2^{bh(x)}-1}$ **nodi interni**

Altezza

Dato un RB-albero con n nodi e di altezza incognita h, considerando $n \geq 2^{bh(x)} - 1, h \geq 2bh(x)$ si ottiene che $n \geq 2^{\frac{h}{2}} - 1$ e quindi che $h \leq 2log_2(n+1)$

Aggiunta di un nodo

Dopo l'inserimento di un nodo x nel RB-albero (usando l'algoritmo di inserimento in un albero binario di ricerca) non è detto che esso sia ancora un RB-albero, infatti bisogna correggere l'anomalia.

```
INSERT(T, x):
    TREE_INSERT(T, x)
    x.COLOR = RED
    while x != root and x.P.COLOR == RED:
        if x.P == x.P.P.LEFT:
            y = x.P.P.RIGHT
            if y.COLOR == RED:
                // caso 1
                x.P.P.COLOR = RED
                y.P.COLOR = BLACK
                y.COLOR = BLACK
            else:
                if x == x.P.RIGHT:
                    // caso 2
                    x = x.P
                    ROTATE_LEFT(T, x)
                // caso 3
                x.P.COLOR = BLACK
                y.P.P.COLOR = RED
                ROTATE_RIGHT(T, x.P.P)
                x = root
```

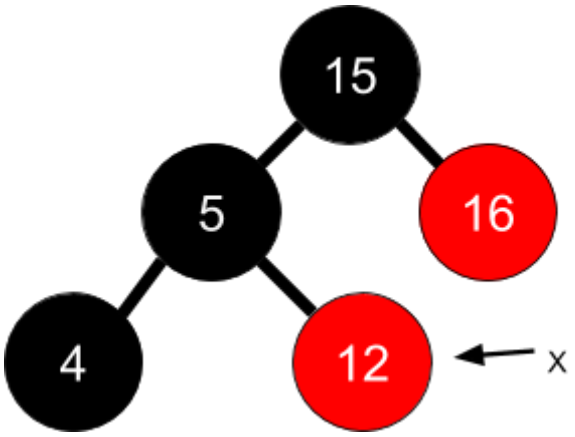
Il tempo di correzione dell'anomalia x è logaritmico $O(logn)$ in quanto il numero di iterazioni è dato dalla profondità dell'albero; il numero di rotazioni massimo è 2 in quanto i casi 2 e 3 sono terminali.

ELIMINAZIONE DI UN NODO DA UN RB ALBERO

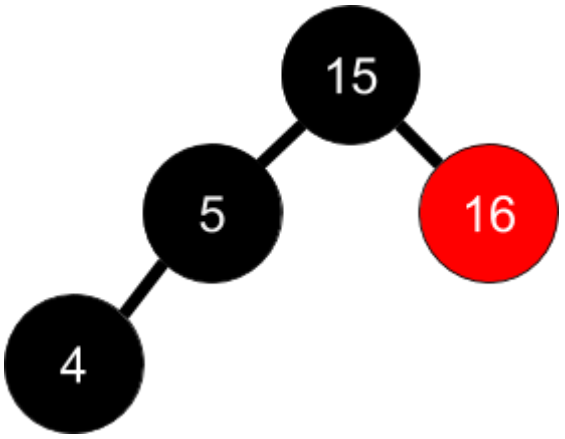
Eliminazione di un nodo rosso (non radice)

CASO 1 - NODO SENZA FIGLI

PRIMA



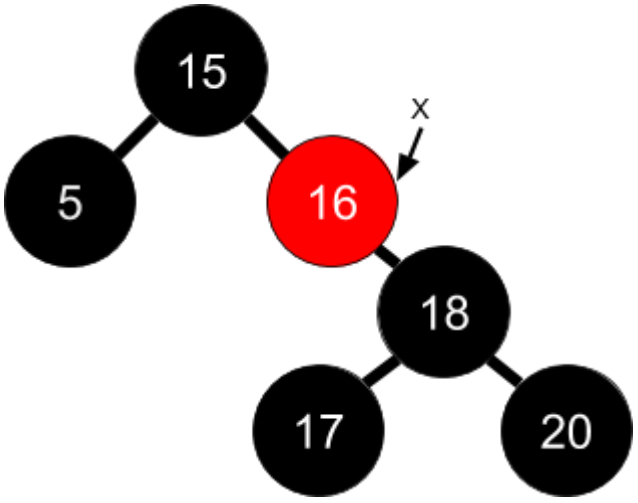
DOPO



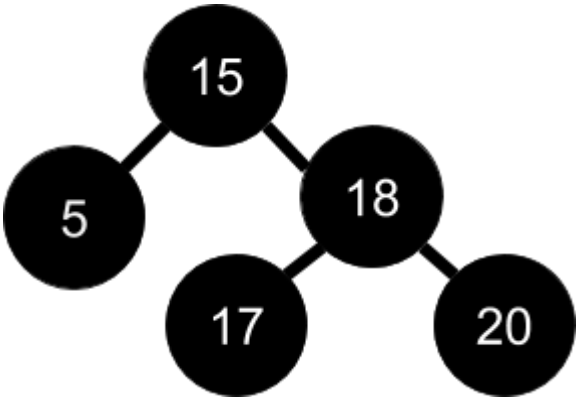
1. Elimino il nodo

CASO 2 - NODO CON UN FIGLIO

PRIMA



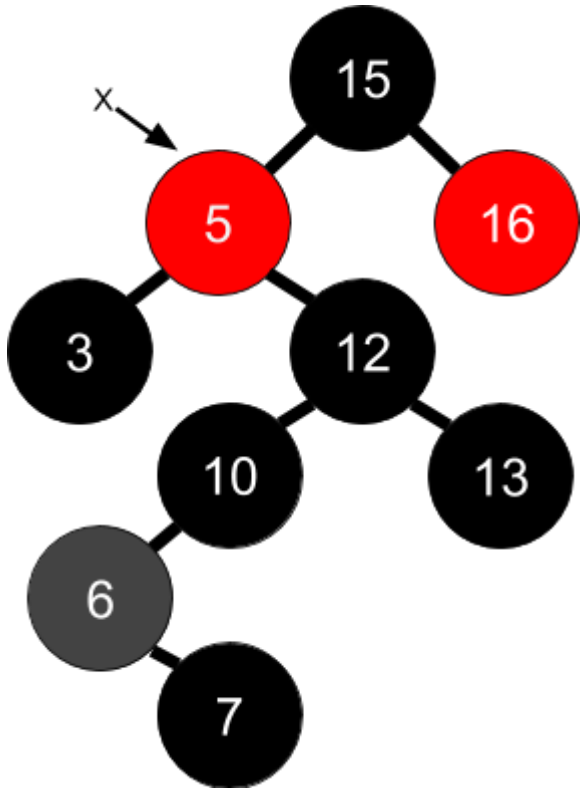
DOPO



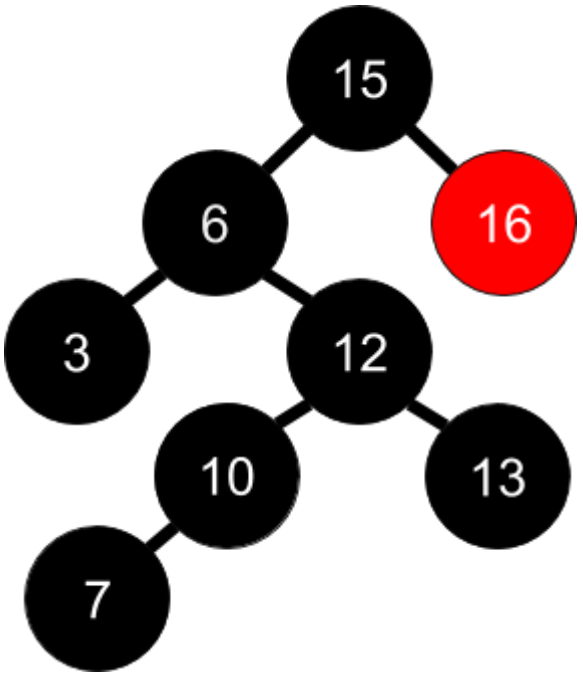
1. Rimuovo il nodo
2. Cambio il parent del figlio con x.P

CASO 3 - NODO CON DUE FIGLI

PRIMA



DOPO



1. Prendo il nodo più piccolo del sottoalbero dx che non abbia un ramo sx
2. Sposto il nodo più piccolo al posto del nodo da cancellare
3. Nel caso il nodo spostato abbia un sottoalbero dx, lo attacco come figlio sx

NOTA: il colore del nodo che sposti viene mantenuto

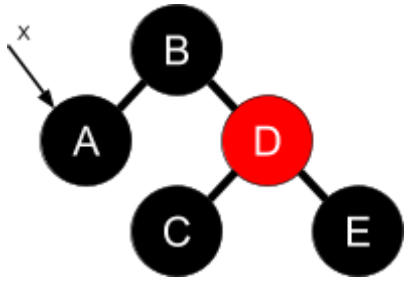
COMPLESSITÁ:

- h = log(n)

Eliminazione di un nodo nero (non radice)

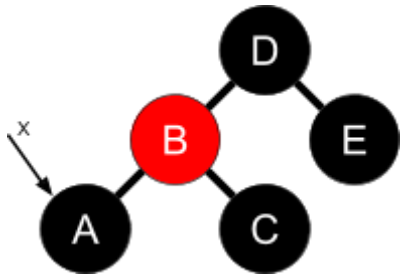
CASO 1 - ABCDE

PRIMA



1. Scambia i colori di D e B
2. Ruota a sx su B

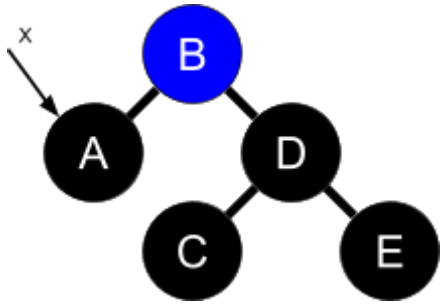
DOPO



NOTA: l'anomalia si sposta verso il BASSO

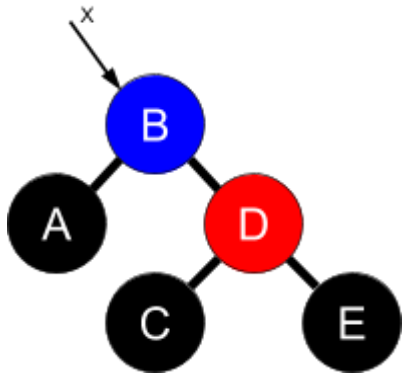
CASO 2 - ABCDE

PRIMA



1. Togli un nero da A e D
2. Aggiungi un nero a B

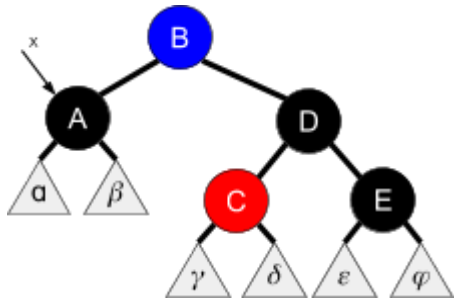
DOPO



NOTA: l'anomalia si sposta verso l'ALTO

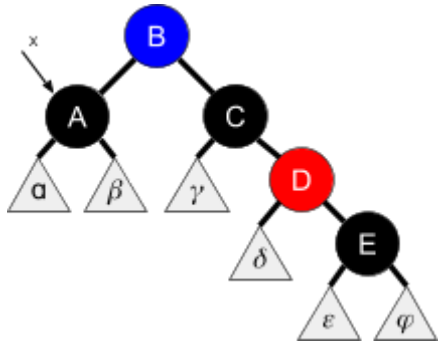
CASO 3 - ABCDE

PRIMA



1. Scambia i colori di C e D
2. Ruota a dx su D

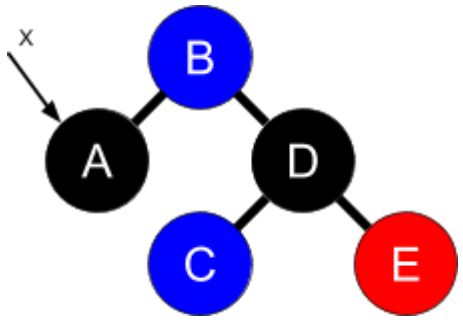
DOPO



NOTA: l'anomalia non si sposta

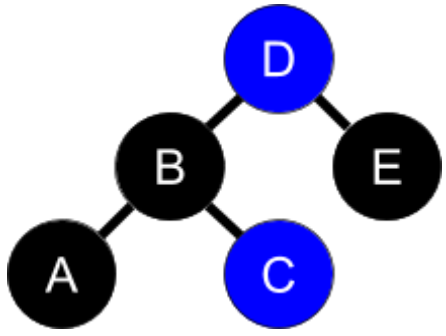
CASO 4 - ABCDE

PRIMA



1. Scambia i colori di D e B
2. Aggiungi un nero a B
3. Aggiungi un nero a E
4. Ruota a sx su B

DOPO



NOTA: ho eliminato X

caso terminale

CONSIDERAZIONI:

- Se comincio col caso 1, nella iterazione successiva si può verificare il caso 2, 3 o 4:
    - Se si verifica il caso 2 propago l'anomalia al padre (che è rosso) e quindi mi Sposto in un caso terminale.
    - Se si verifica il caso 3, poi avrò il caso 4 e terminerò.
    - Se si verifica il caso 4, terminerò.
- Se comincio col caso 2:
    - Posso ripeterlo al massimo log(n) volte.
    - Mi sposto (dopo una ripetizione) in un caso terminale.
- Subito dopo il caso 3 passo al caso 4.
- Il caso 4 è terminale.

COMPLESSITÁ:

- al massimo 3 rotazioni
- $\Theta(nlogn)$

IDEA DELL'ALGORITMO

```
REMOVE(T, x):
    while x != null and x.COLOR != RED:
        // da considerare anche per i casi simmetrici
        if caso1:
            ...
        else if caso2:
            ...
        else if caso3:
            ...
        else:  // caso4
            ...
```

Arricchire un RB-albero

TEOREMA
Se un campo di un elemento di un albero dipende solo dai nodi sottostanti, il campo è mantenibile in tempo logaritmico

SIZE
È un campo che indica per ogni nodo il numero di nodi che stanno nel suo sotto-albero.
L'aggiornamento di questo campo è mantenibile in tempo logaritmico $\Theta(logn)$.

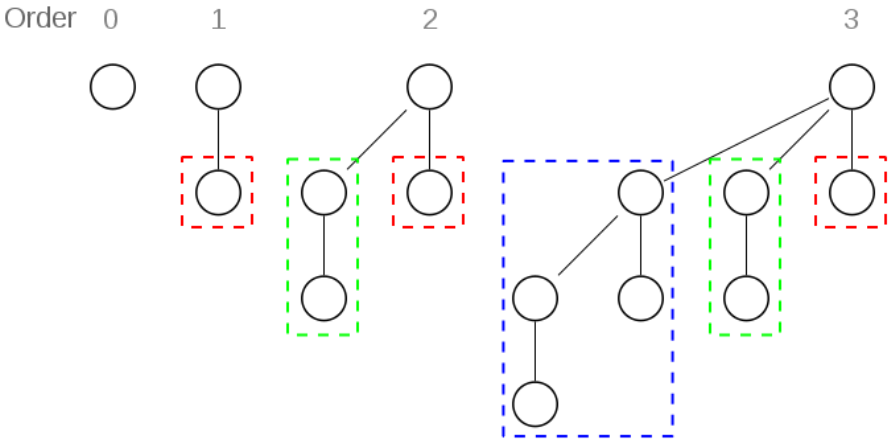
RANK(X)
È una funzione che, usando i campi size dei nodi, ottiene la posizione del nodo X tra tutti gli elementi dell'albero ordinati.

Questa funzione, dato un nodo, scala l'albero fino alla radice e fa calcoli solo sul campo size, che si mantiene in tempo logaritmico; quindi questa funzione impiega a sua volta un tempo logaritmico $\Theta(logn)$.

Unione di RB-alberi

La soluzione più semplice è trasformare il RB-albero più piccolo in un array e poi aggiungere ogni elemento uno per uno all'altro RB-albero. Questo algoritmo impiega un tempo $\Theta(nlogn)$

Alberi binomiali



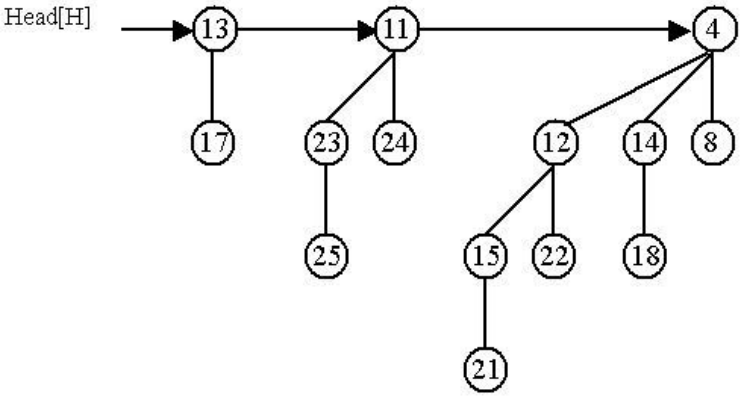
Gli alberi binomiali sono definiti ricorsivamente, in quanto ogni albero di ordine k contiene tutti gli altri alberi di ordine inferiore a k fino a 0. Sono detti binomiali perché ad ogni livello il numero di nodi equivale al valore corrispondente nel triangolo di Tartaglia, che si ottiene con la formula del binomio di Newton.

Proprietà di un albero B_k:

1) Ha 2^k nodi
2) L'altezza è k
3) A profondità *i* ci sono $\binom{k}{i}$
4) I figli della radice sono radici di $B_{k-1}, \ldots, B_0$

Heap binomiale

È un heap che si basa su un albero binomiale e quindi ha le seguenti proprietà.



Proprietà di uno heap HB_k:

- è una lista di alberi binomiali
- ogni albero è ordinato secondo uno heap

- per ogni dimensione c'è al più un albero (es $HB_7 = B_0 + B_1 + B_2$)
- un heap binomiale di n nodi contiene $\lfloor \log_2 n \rfloor$ alberi

Unione di due heap binomiali

Questa operazione è riconducibile alla somma binaria: infatti, se consideriamo le radici degli alberi binomiali nell'heap come bit che sono settati a 1 o 0 in base alla presenza o meno di un albero, allora il risultato della somma è la rappresentazione binaria della presenza o meno degli alberi binomiali nell'heap risultante dall'unione.

es.

$HB_7 = B_0 + B_1 + B_2 = 000111$

$HB_{51} = B_0 + B_1 + B_4 + B_5 = 110011$

$000111 + 110011 \ = \ 111010 \ = B_1 + B_3 + B_4 + B_5 = HB_{58}$

I due heap vengono uniti in un unico array, mantenendo l'ordine di grandezza delle radici. Poi a partire dalla prima radice, si collassano le altre al suo interno; quando ci sono 3 radici che hanno un albero di dimensione uguale, il puntatore si sposta a destra di uno e l'algoritmo continua.

Inserimento ed estrazione di un nodo

- L'inserimento viene ricondotto alla unione di un albero binomiale HB_1 con l'heap.
- La rimozione di un nodo consiste nel prendere tutti i suoi figli, metterli in una lista così da creare un nuovo HB e poi unire i due alberi dopo aver rimosso il nodo.

Stili implementativi

Programmazione Dinamica

Dato un problema di grande complessità, trovo una **soluzione ricorsiva su sottoproblemi** eliminando le iterazioni ripetute inutilmente (usando la **MEMOIZZAZIONE**) ed ottenendo un algoritmo di complessità nettamente inferiore, ma comunque non lineare/logaritmica.

Programmazione Greedy

Dato un problema di grande complessità, lo si risolve subito **senza snocciolarlo in sottoproblemi**, implementando quindi una soluzione senza pensare in modo dinamico, che molte volte ha complessità molto più alta rispetto a lineare o logaritmica.

Insiemi

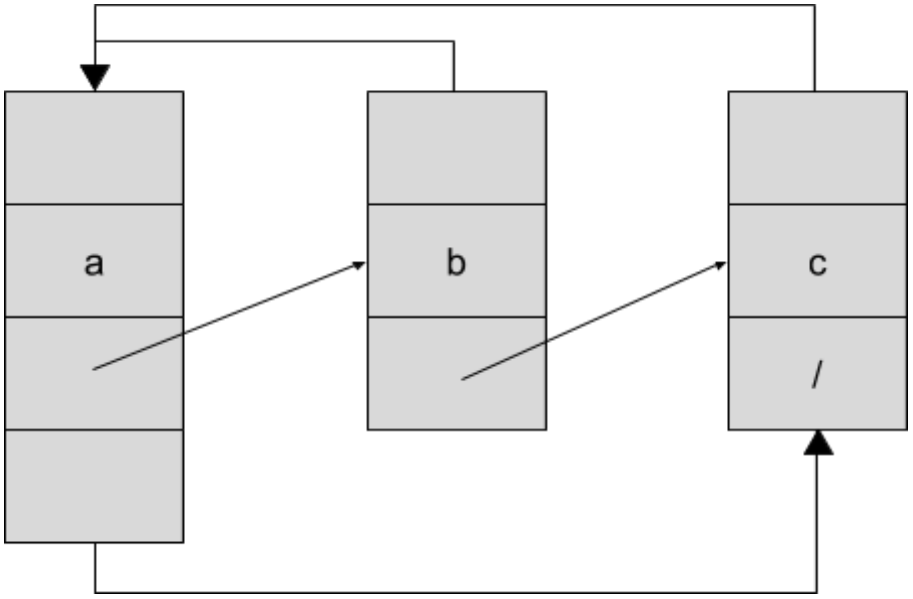
Insiemi disgiunti

Gli insiemi disgiunti possono essere rappresentati sfruttando le seguenti ipotesi:

- l'intersezione tra due insiemi è uguale a ∅
- $\{s_1, s_2 \dots s_3\}$ è una collezione di insiemi disgiunti definiti su un insieme di oggetti x
- ogni insieme ha un rappresentante, usato come comparatore per gli insiemi: se due insiemi hanno lo stesso rappresentante, allora fanno parte dello stesso insieme. Questa operazione prende il nome di **`IS_SAME_SET(x, y)`**

Interfaccia

- **`MAKE_SET(x)`** : crea un insieme con solo *x*
- **`UNION(x, y)`** : unisce gli insiemi di cui fanno parte *x* e *y*
- **`FIND_SET(x)`** : restituisce il rappresentante relativo all'insieme a cui appartiene *x*



Gli insiemi disgiunti vengono rappresentati con delle liste concatenate. Ogni membro della lista mantiene una referenza al rappresentante. In questa maniera le operazioni **`MAKE_SET(S)`** e **`FIND_SET(S,x)`** sono risolvibili in O(c) ma **`UNION(S)`** no.

Per effettuare la union, infatti, devo cambiare tutti i puntatori al rappresentante dei vari oggetti presenti in uno dei due insiemi (ovviamente mi converrà tenere il rappresentante dell'insieme col numero maggiore di elementi e cambiare gli elementi dell'insieme minore). L'alternativa (cambiare solo il primo puntatore al rappresentante) mi farebbe perdere molto tempo con la **`MAKE_SET`**, alzando di molto la complessità media.

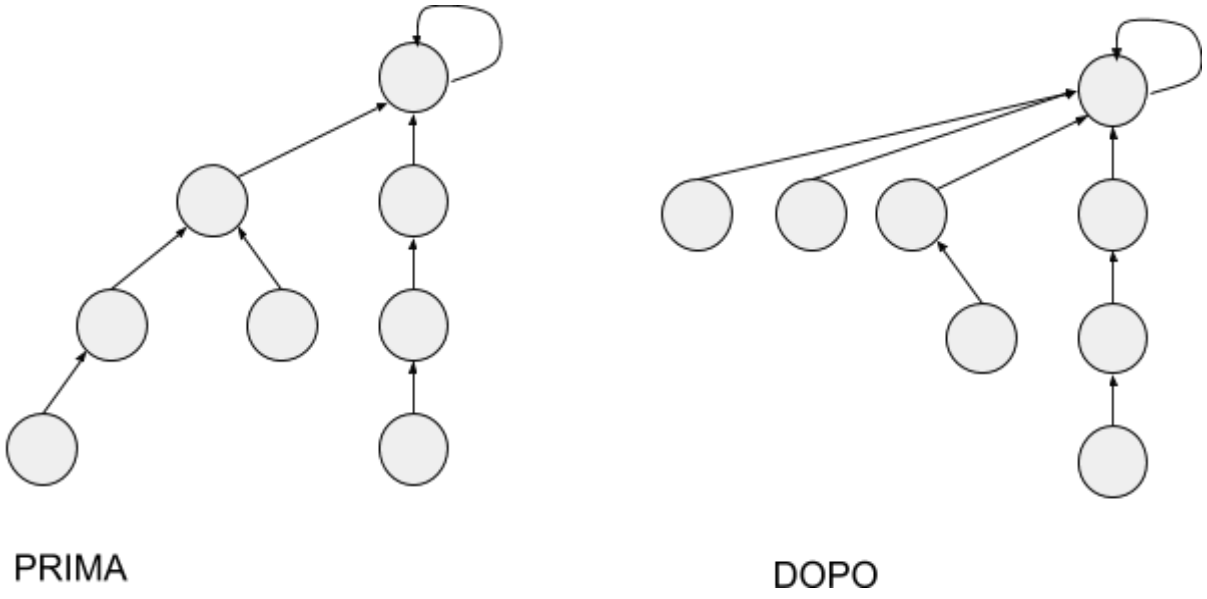
La tecnica che ci permette di velocizzare la **`UNION`** è l'**unione per rango**, mettendo un campo size nel rappresentante e cambiando il suo puntatore per farlo puntare al rappresentante dell'insieme con cui è stato appena unito. In questo modo cambio puntatore al massimo $nlogn$ volte, garantendo quindi una complessità di O($m + n \log n$).

Rappresento quindi gli insiemi come alberi dove la radice è il rappresentante. Questa nuova configurazione mi permette di lasciare solo la complessità della **`MAKE_SET`** inalterata mentre la **`FIND_SET`** e la **`UNION`** avranno complessità O(n). Inserisco anche una limitazione

al numero di rango massimo (alla profondità massima), ottenendo quindi rango 1. 2 elementi minimo, rango 2. 4 elementi minimo, rango 3 8 elementi minimo e via di seguito. Questo mi porta ad avere una profondità massima di *log n*.

La tecnica di compressione dei cammini

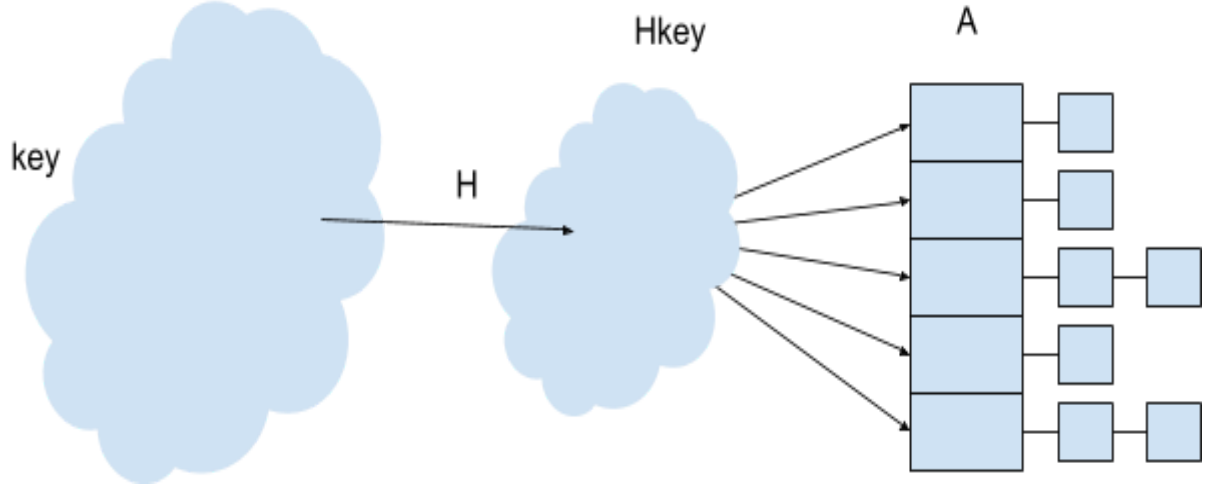
Quando viene fatta la **`FIND_SET`** di un elemento *x* per trovare il suo parent (rappresentante), quindi scorrere tutto il cammino, vengono anche trovato il parent di tutti i nodi intermedi, appiattendo la struttura.



Funzioni Hash

Tabelle hash

A ogni variabile in memoria è associata la cella corrispondente: devo quindi comprimere gli indici.



In qualsiasi modo io faccia la mia funzione hash, posso avere delle collisioni (passo da un dominio più grande ad uno più piccolo) con probabilità 1/*m*. Questa probabilità è dettata dal fatto che vengano mappati due elementi nella stessa cella; per questo si usa una lista concatenata di elementi. Nella probabilità 1/*m*, *m* è il numero di elementi dell'array associato

alla cella su cui si mappa. Cerco di ridurre il problema creando una lista concatenata associata agli elementi di A. Il fattore di carico è $\alpha = n/m$, dove α corrisponde alla lunghezza media della lista associata ad A, *n* il numero di oggetti ed *m* quello di celle (con $\alpha \leq 1$ e $n < ½m$).

Funzione hash

La funzione hash migliore è H(k) = $m\ (KA\ \%\ 1)$ con 0 < *A* < 1 per non avere una funzione critica nei confronti di *m* (dovrei avere *m* numero primo e potenza di 2 per andare bene). Una volta calcolato l'hash, lo approssimo per difetto. Il valore ideale per *A* è 0.618 (sezione aurea).

Indirizzamento aperto

Al posto di associare una lista ad ogni cella, se la cella è occupata inserisco il valore nella cella libera immediatamente sotto. Quando leggo, se la cella che ottengo della tabella non è quella che mi serviva, vado a leggere la cella dopo. Questo metodo mi fornisce un fattore di carico basso (½) ma al tempo stesso va ad interferire con altri possibili hash. Posso quindi implementare un sistema di salto delle celle a numero variabile (a numero fisso rischio di avere una coda circolare) per velocizzare il salvataggio ma devo trovare un modo per riuscire a calcolare il salto finale.

Hashing doppio

Questo metodo mi permette di calcolare il salto variabile. Partendo dal presupposto che ho H(x, i) con i = numero collisioni, posso stabilire dove finirò nella tabella grazie alla formula H(x) + ni dove n indica il passo (es. 2, 3...)

I grafi

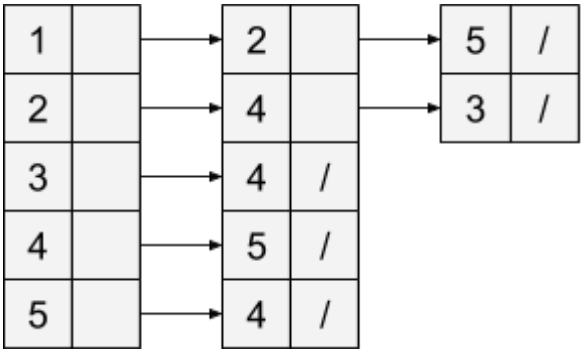
Definizioni

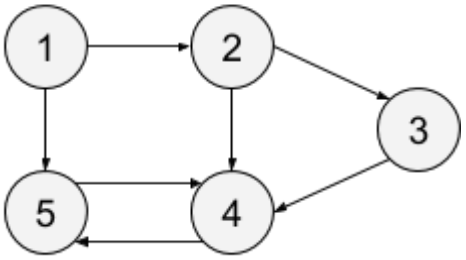
Definizione di grafo: insieme di linee che uniscono coppie di nodi. I nodi sono chiamati vertici (V) e le linee sono dette archi (E). Possiamo quindi dire formalmente che G = (V, E), E ⊆ VxV.

- Un grafo si dice **orientato (o diretto)** se i suoi archi hanno una direzione.
- Il **grado** di un nodo è il **numero di archi** che incidono sul nodo stesso. Può essere valutato sul singolo arco o sul collegamento (A → B e B → A oppure A -- B), se **entrante** o **uscente**.
- Un grafo si dice **completo** se non è possibile aggiungere archi (**V^2 archi**).
- Un grafo si dice **sparso** se il numero di archi è **nettamente inferiore** a quello di un grafo completo.
- Un **cammino** è una **sequenza** di **nodi** $\forall\ i \in \{1, 2 \dots n\text{-}1\}, (v_i, v_{i+1}) \in E$.
- Un cammino si dice **semplice** se **passa** dai nodi **una sola volta**.
- Un cammino si dice **ciclico** se **non è semplice**.
- Un **grafo** si dice **ciclico** se **contiene un cammino ciclico**.

- La **lunghezza** di un cammino è il **numero di archi da attraversare**.
- In un grafo **orientato**, due nodi si dicono **mutuamente raggiungibili se da entrambi i nodi è possibile raggiungere l'altro**.
- L'**insieme massimale** di nodi **mutuamente raggiungibili** si dice **strettamente raggiungibile (componente fortemente connessa)**.

Rappresentazione di un grafo - liste di adiacenza

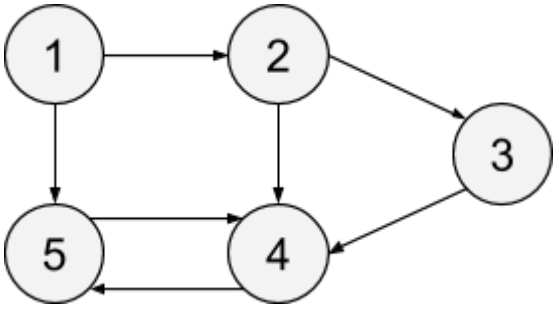




Spazio di memoria occupato: **Θ(V+E)**
Funziona meglio se il grafo è sparso

Rappresentazione di un grafo - matrice di adiacenza

	1	2	3	4	5
1	0	1	0	0	1
2	0	0	1	1	0
3	0	0	0	1	0
4	0	0	0	0	1
5	0	0	0	1	0



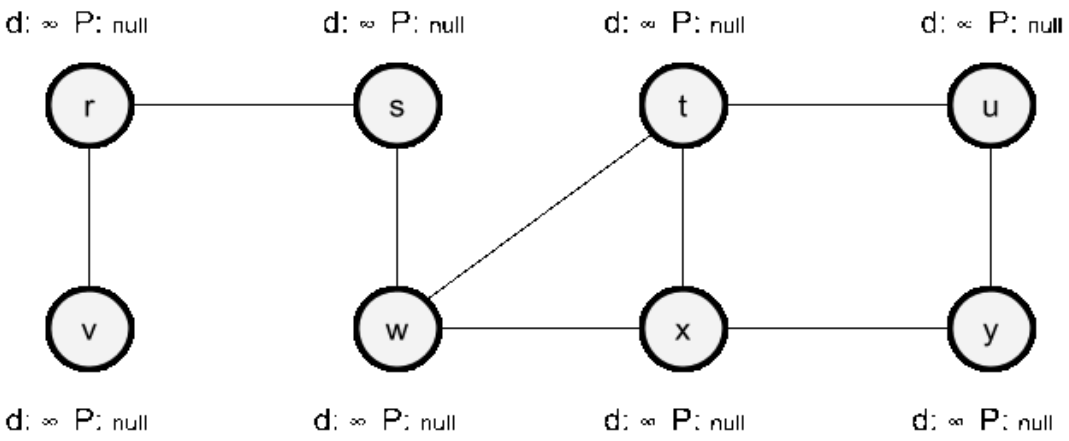
Spazio di memoria occupato: **Θ(V^2)**
Funziona meglio se il grafo è completo

Algoritmi di visita - BFS

Note introduttive: l'algoritmo BFS (Breadth First Search) si basa sull'esplorazione in ampiezza (a livelli) del grafo. Per definire lo stato di visita di un nodo useremo dei colori, il bianco, il grigio ed il nero, per indicare rispettivamente un nodo mai esplorato, uno in visita e uno già esplorato completamente. Useremo anche *P* (o π) per indicare il predecessore di un nodo e ***d*** per indicare la distanza da un nodo ***s*** (source) e il nodo attuale.

```
BFS(G, s):
    for (Vertice u : (V[G] - {s})) {
        color[u] = WHITE; //nodo mai visitato
        d[u] = ∞; //non so come raggiungerlo
        P[u] = null;
    }
    color[s] = GRAY; //ho visitato la radice ma non i suoi vicini
    d[s] = 0; //la distanza della radice da se stessa è 0
    Q = {s}; //coda FIFO che contiene elementi grigi
    while (Q != ∅) {
        u = head(Q);
        for (Vertice v : Adj[u]) {
            if (color[v] == WHITE) { //solo se non l'ho mai visitato
                color[v] = GRAY; //ho scoperto un nuovo nodo
                d[v] = d[u] + 1;
                P[v] = u;
                enqueue(Q, v); //aggiungilo a Q
            }
            dequeue(Q); //togli u da Q, hai visitato tutti i vicini
            color[u] = BLACK; //segnalo come completato
        }
    }
```

Complessità: **Θ(V+E)**



Q = {}

Algoritmi di visita - DFS

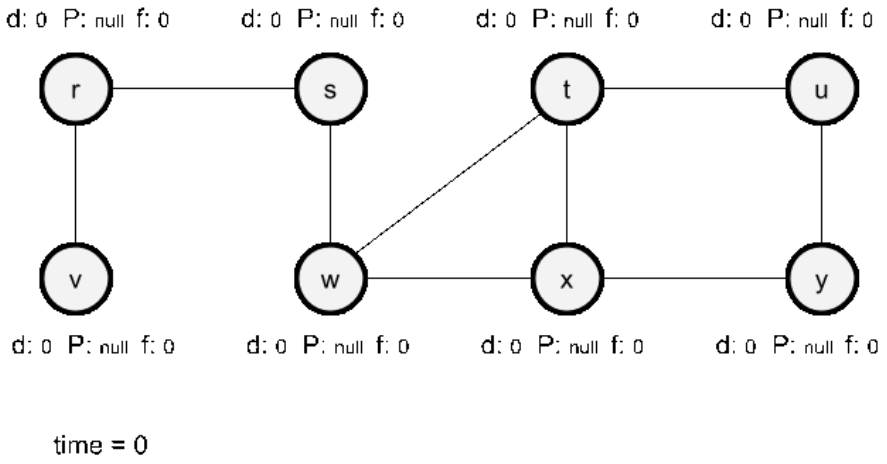
Note introduttive: l'algoritmo DFS (Depth First Search) si basa sull'esplorazione in profondità del grafo. Ciascun nodo, oltre ad avere l'attributo relativo al colore del nodo, avrà altri due attributi, il tempo di inizio visita e il tempo di fine visita. Gli attributi d e f indicano rispettivamente il tempo di inizio e quello di fine visita.

```
DFS(G):
    for (Vertice u : V[G]) {
        color[u] = white; //nodo mai visitato
        P[u] = null;
    }
    time = 0; //contatore globale
    for (Vertice u : V[G]) {
        if (color[u] == white) {
            DFS_visit(u);
        }
    }

DFS_visit(u)
    color[u] = gray; //ho visitato il nodo ma non i suoi vicini
    d[u] = ++time;
    for (Vertice v : Adj[u]) {
        if (color[u] == white) {
            P[v] = u;
            DFS_visit(v);
        }
    }
    color[u] = black;
    f[u] = ++time;
```

Complessità: **Θ(V+E)**

Da questo algoritmo possiamo dedurre che un nodo discende da un altro se e solo se il suo intervallo è completamente contenuto nell'intervallo dell'altro nodo. Con il tempo di inizio visita e quello di fine visita posso sapere in $\Theta(c)$ se un nodo discende da un altro.



Componenti connesse

Una componente connessa di un grafo è un sottografo in cui:

- qualsiasi coppia di vertici è connessa da cammini.
- il sottografo non è connesso a nessun vertice addizionale del grafo.

Per semplicità, possiamo definirla anche in un altro modo: dati due nodi a cui viene dato un valore al campo "componente" ***cc***, essi sono appartenenti alla stessa componente connessa se e solo se hanno lo stesso valore nel campo *cc*. Per fare ciò modifichiamo la DFS.

```
DFS(G):
    for (Vertice u : V[G]) {
        color[u] = white; //nodo mai visitato
        P[u] = null;
    }
    time = 0; //contatore globale
    x = 0; //valore globale per le cc
    for (Vertice u : V[G]) {
        if (color[u] == white) {
            DFS_visit(u);
            x++;
        }
    }

DFS_visit(u)
    color[u] = gray; //ho visitato il nodo ma non i suoi vicini
    d[u] = ++time;
    cc[u] = x; //nella cc numero x
    for (Vertice v : Adj[u]) {
        if (color[u] == white) {
            P[v] = u;
            DFS_visit(v);
        }
```

```
    }
    color[u] = black;
    f[u] = ++time;
```

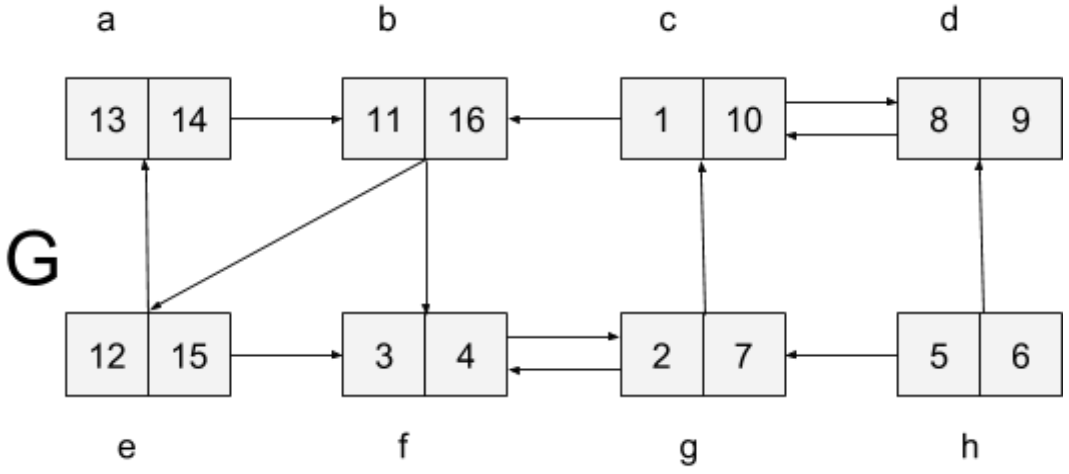
L'algoritmo per trovare le componenti connesse e raggrupparle è il seguente:

```
CC(G):
    for (Vertice v : V[G]) {
        make_set(v); //creo un insieme per ogni nodo
    }
    for (Arco (u,v) : E[G]) { //per ogni arco
        if (find_set(u) != find_set(v)) {
            union(u, v); //fanno parte della stessa componente connessa
        }
    }
```

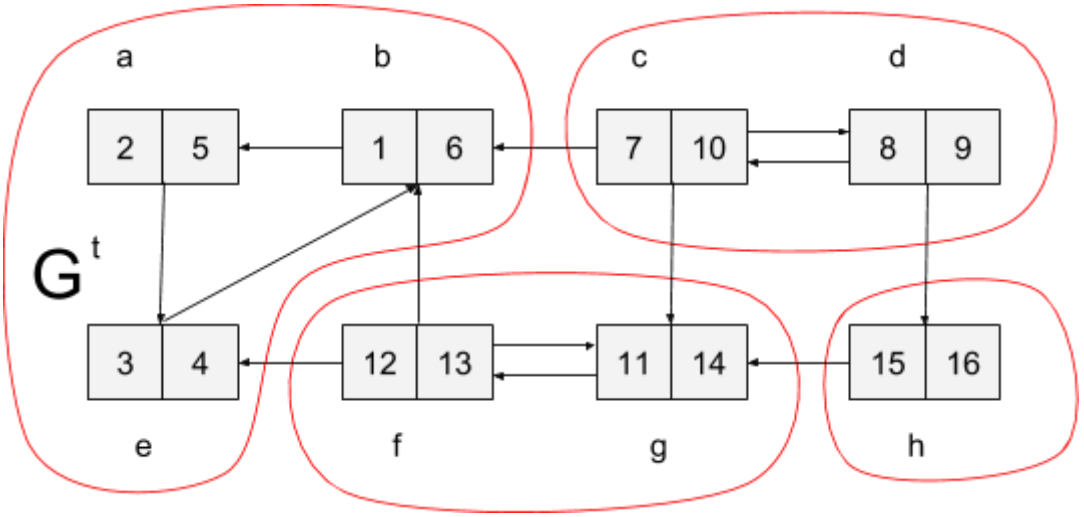
Complessità: **Θ(V+E+V log V)** o **Θ(m+n log n)** con m operazioni e n makeset.

Componenti strettamente connesse

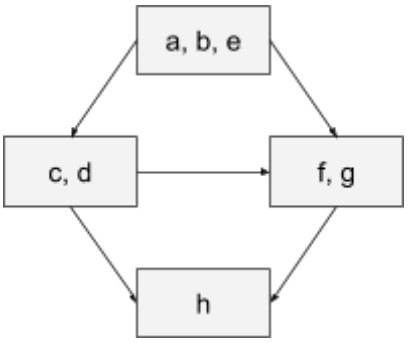
```
SCC(G):componenti fortemente connesse
    //DFS su G per calcolare f
    //DFS su G^t visitando i nodi per f crescente
```



Dopo aver applicato l'algoritmo SCC tutti i nodi di una stessa componente connessa finiscono nello stesso albero.



Da questo grafo possiamo accorpare in gruppi di componenti connesse e collegandole tra loro otteniamo un grafo aciclico e senza mutua raggiungibilità. Se infatti fossero mutuamente raggiungibili tra loro, potremmo raggruppare il grafo in un'unica componente connessa.



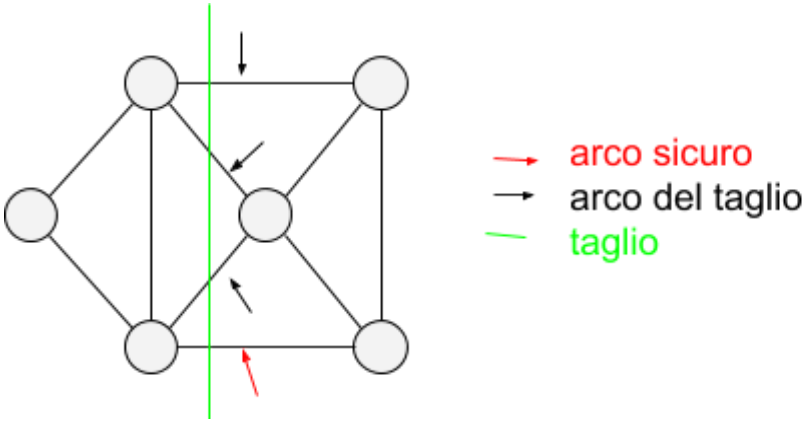
Ordinamento topologico

Ordinamento lineare di tutti i vertici in un grafo **diretto aciclico**

```
Ord_top(G):
    //usa DFS per calcolare f
    //restituisci i nodi per f crescente
```

Albero di copertura di costo minimo (Minimum Spanning Tree)

Dato un grafo non orientato G=(V, E), costruire un grafo non orientato G=(V, A) eliminando alcuni archi e ottenendo un grafo i cui nodi sono ancora tutti raggiungibili e il cammino che li lega ha il costo minimo rispetto a tutti gli altri cammini possibili.
Taglio: **bipartizione** dei nodi.
Arco del **taglio**: archi **interessati** dal taglio.
Arco sicuro per un taglio: arco del taglio di **costo minimo**.



Algoritmo di Kruskal

```
Kruskal(G, w):
    //ordina E per peso non decrescente, ≤
    A = ∅;
    for (Vertice v : V[G]) {
        make_set(v); //creo un insieme per ogni nodo
    }
    for (Arco (u,v) : E[G]) { //per ogni arco
        if (find_set(u) != find_set(v)) { //se non sono uniti
            A = A ∪ (u, v);
            union(u,v);
        }
    }
```

NOTE:

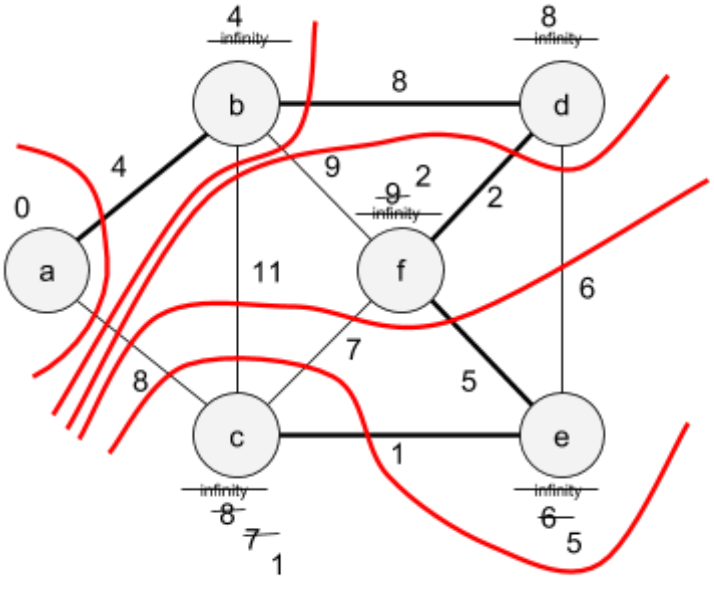
- prendi l'arco che pesa di meno: con un taglio, quello ci sarà di sicuro (non ho bisogno di tagliare effettivamente, il taglio è solo un approccio visuale al problema)
- l'algoritmo ha complessità **Θ(E log E)**

Algoritmo di Prim

```
Prim(G, w, r): //r nodo root di partenza
    Q = V[G]
    for (Vertice v : V[G]) {
        key[v] = ∞;
    }
    key[r] = 0;
    P[r] = null;
    while (Q != ∅) {
        u = extract_min(Q); //prendo il vertice con key minimo in Q
        for (Vertice v : Adj[u]) {
            if (v ∈ Q && w(u,v) < key[v]) {
                P[v] = u;
                key[v] = w(u, v);
            }
        }
    }
```

NOTE:

- vedo il mio nodo root come circondato da una barriera. L'obiettivo è quello di espandere la barriera e inglobare i nodi che mi portano ad avere il MST
- guardo le adiacenze e metto come chiave il costo dell'arco per raggiungere il nodo attraversando la barriera
- prendo l'arco col costo minimo
- l'arco P punta il predecessore che mi ha dato la key
- questo algoritmo ha complessità variabile in base alla struttura scelta, perché la extract_min e la reduce_key cambiano:
    - **Θ(VE)** se uso delle **liste ordinate**
    - **Θ(V^2)** se uso delle **liste non ordinate**
    - **Θ((V + E)log V)** se uso gli **heap**
    - **Θ(V log V + E)** se uso gli **heap di Fibonacci**

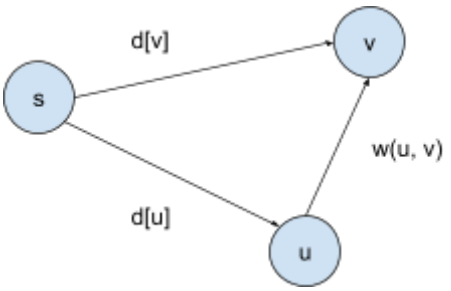


ordine della extract_min(): b → d → f → e → c

Cammini minimi

Dato un albero pesato con partenza nel nodo ***s***, si vuole trovare il cammino minimo verso un qualsiasi nodo ***v***. Il cammino minimo è la somma minima dei pesi degli archi tra i nodi tra ***s*** e ***v***. L'idea di base è il rilassamento del grafo, ovvero la ricerca di cammini, che passino da un altro nodo, che abbiano un costo inferiore a quello dal nodo partenza a quello indicato.

La soluzione al problema della ricerca dei cammini minimi non sempre è possibile, infatti basti pensare al caso in cui esista un ciclo negativo, ovvero un ciclo che abbia archi la cui somma dei pesi risulti essere un numero inferiore a 0; con un ciclo così, il cammino minimo di un grafo sarebbe impossibile da trovare, infatti si potrebbe sempre diminuire passando un'altra volta per il ciclo.



Questi due algoritmi saranno la base per algoritmi successivi che risolveranno il problema.

```
Relax(u, v, w): //d[v] distanza da s a v
    if (d[v] > d[u] + w(u,v)) { //disequazione di Bellmann
        d[v] = d[u] + w(u,v);
        P[v] = u;
    }

Init(G, s):
    for (Vertice v : V[G]) {
        d[v] = ∞;
        P[v] = null;
    }
    d[s] = 0;
```

Algoritmo di Dijkstra

Funziona solo se gli archi hanno tutti peso ≥ 0 (e così da non poter avere cicli negativi).

```
Dijkstra(G, w, s):
    Init(G,s);
    Q = V[G];
    while (Q != ∅) {
        u = extract_min(Q);
        for (Vertice v : Adj[u]) {
            relax(u, v, w);
        }
    }
```

- questo algoritmo ha complessità variabile in base alla struttura scelta:
    - **Θ(VE)** se uso delle **liste ordinate**
    - **Θ(V^2)** se uso delle **liste non ordinate**
    - **Θ((V + E)log V)** se uso gli **heap**
    - **Θ(V log V + E)** se uso gli **heap di Fibonacci**

Algoritmo di Bellman-Ford

Funziona anche se il grafo ha archi negativi MA NON può avere cicli negativi (ritorna false).

```
Bellman_Ford(G, w, s):
    Init(G,s);
    for (i = 1 to (|V[G]| - 1)) {
        for (Edge (u,v) : E) {
            relax(u, v, w);
        }
    }
    for (Edge (u,v) : E) {
        if (d[v] > d[u] + w(u,v)) {
            return false;
        }
    }
    return true;
```

Complessità: **Θ(VE)**

Algoritmo di Johnson

Se ho degli archi con peso negativo

```
Johnson(G, w, s):
    Bellman_Ford(G, w, s);
    h = d;
    Dijkstra(G, w, s)
```

Complessità: **Θ(V^2logV)**

Algoritmo DAG_SP (Directed Acyclic Graph Shortest Path)

Nel caso in cui il grafo sia aciclico possiamo usare un algoritmo **Θ(V + E)**

```
DAG_sp(G, w, s):
    Init(G,s);
    for (Vertice u : V[G]) { //presi in ordine topologico
        for (Vertice v : Adj[u]) {
            relax(u, v, w);
        }
    }
```

Algoritmo Extend_SP

La distanza *d* può essere rappresentata sotto forma di matrice *D* e lo stesso vale per il peso. La dicitura D^k_{ij} indica il costo del cammino minimo tra i nodi *i* e *j* che usa solo nodi in {1 .. k} con $1 \leq i \leq k \leq j$ come nodi intermedi. Complessità **Θ(n^3*logn*)**

```
Extend_SP(D, W):
    n = rows(D);
    for (i = 1 to n) {
        for (j = 1 to n) {
            D^i_ij = ∞;
            for (k = 1 to n) {
                D^i_ij = min(D^i_ij, D_ik + W_kj)
            }
        }
    }
```

Algoritmo Floyd-Warshall

Usando i principi di Extend_SP, trova la soluzione a partire dal grafo anziché dalla matrice. Complessità $\Theta(n^3)$.

```
Floyd_Warshall(G, w):
    D^0 = W;
    n = rows(W);
    for (k = 1 to n) {
        for (i = 1 to n) {
            for (j = 1 to n) {
                D^k_ij = min(D^k-1_ij, D^k-1_ik + D^k-1_kj)
            }
        }
    }
```

Chiusura transitiva di un grafo

La chiusura transitiva di un grafo non è altro che l'appiattimento dei cammini di tutti i nodi di un grafo; per esempio, se in un grafo esistono gli archi E={(a,b),(b,c)} si conclude che per la proprietà transitiva si ha il cammino a→c: la chiusura transitiva di questo grafo aggiunge tutti i cammini appiattiti all'insieme degli archi, quindi in questo caso aggiunge (a,c).

La complessità dell'algoritmo è $\Theta(V(V + E))$, che in un **grafo sparso** diventa $\Theta(V^2)$ mentre in un **grafo completo** diventa $\Theta(V^3)$.

Flusso massimo

Si vuole trasportare un flusso attraverso una rete, senza accumulare ne far fermare il flusso in un particolare nodo. Diamo alcune definizioni:

- **Rete di flusso**: è un grafo G=(V, E) in cui ogni arco in E ha una capacità $c:V\times V\to\mathbb{R}^{\geq 0}$ tale che $\forall u,v\ (u,v)\notin E \Rightarrow c(u,v) = 0$. Possiede due nodi particolari, ***s*** per la partenza del flusso e ***t*** per la destinazione
- **Flusso**: è una funzione $f:V\times V\to\mathbb{R}$ con i seguenti vincoli
    - **capacità**: $\forall u,v\ f(u,v) \leq c(u,v)$ *[il flusso deve essere minore della capacità]*

    - **antisimmetria**: $\forall u,v\ f(u,v) = -f(v,u)$ *[la freccia al contrario è un flusso negativo]*
    - **conservazione**: $\forall u \in V - \{s, t\}\ \sum_{v \in V} f(u,v) = 0$ *[la materia si conserva]*
- **Rete residua**: è un grafo $G^l=(V, E^l)$ con una capacità (residua) $c^l(u,v) = c(u,v) - f(u,v)$
- **Capacità di un taglio**: è la somma delle capacità degli archi tagliati, ovvero in formula $c(Y, Z) = \sum_{(u,v) \in Y \times Z} c(u,v)$

Algoritmo di Ford-Fulkerson
Permette di trovare il flusso massimo che attraversa un grafo da un punto ad un altro.

```
Ford_Fulkerson(G, c):
    f = identicamente 0; // tale che ∀(u,v) f(u,v)=0
    while (∃ cammino aumentante in G_f) {
        P = cammino aumentante;
        x = minima capacità residua su P;
        flusso lungo P += x;
    }
```

Se usiamo la BFS per trovare il cammino aumentante la complessità è **Θ(|f*|(V + E))**.

Algoritmo di Karp

In alcuni casi può essere migliore del Ford-Fulkerson e ha complessità **Θ(VE(V + E))**.

- Grafi $G = (V, E)$ con $E \subseteq V \times V$
    - matrici di adiacenza (meglio completo): $\mathbf{V^2}$
    - liste di adiacenza (meglio sparso): **V + E**
- Visita
    - BFS: **V + E**
    - DFS: **V + E**
- Componenti Connesse: **V + E + VlogV**
- Copertura di costo minimo (Minimum Spanning Tree)
    - Kruskal: **ElogE**
    - Prim
        - Liste ordinate: **VE**
        - Liste non ordinate: $\mathbf{V^2}$
        - Heap: **(V + E)logV**
        - Heap di Fibonacci: **VlogV + E**
- Cammini minimi
    - Dijkstra
        - Liste ordinate: **VE**
        - Liste non ordinate: $\mathbf{V^2}$
        - Heap: **(V + E)logV**
        - Heap di Fibonacci: **VlogV + E**
    - Bellman-Ford: **VE**
    - DAG_SP: **V + E**
    - extend_sp: $\mathbf{n^3 logn}$
    - Floyd-Warshall: $\mathbf{n^3}$
- Chiusura transitiva: **V(V + E)**
- Flusso massimo
    - Ford-Fulkerson: **|f*|(V + E)**
    - Karp: **VE(V + E)**