

- **BUILD-MAX-HEAP**, eseguita in tempo lineare, genera un MAX-HEAP da un array di input non ordinato
- **HEAP-SORT**, eseguita in $O(n \log n)$, ordina sul posto un array
- **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, **HEAP-MAXIMUM**, eseguite in $O(\log n)$ permettono ad un heap di essere utilizzato come una coda di priorità.

HEAPSORT (A)

```

1. BUILD-MAX-HEAP(A) // in questo modo l'elemento maggiore si trova nella radice
2. for i = A.length down to 2 do // itera partendo dalla fine dell'array
3.   swap A[1] with A[heap-size] // scambia il primo elemento (il maggiore) con l'ultimo
4.   A.heap-size = A.heap-size - 1 // diminuisce la dimensione dell'heap in quanto l'elemento maggiore è stato
                                spostato in fondo
5.   MAX-HEAPIFY(A, 1) // per garantire che la proprietà del MAX-HEAP sia mantenuta e che il massimo elemento sia la
                        radice

```

MAX-HEAPIFY: -input: A (array), indice i:

- quando viene chiamata assume che gli alberi binari con radice in $\text{LEFT}(i)$ e $\text{RIGHT}(i)$ siano max-heap, ma che $A[i]$ possa essere più piccolo dei suoi figli, violando la proprietà del MAX-HEAP. Quindi, fa scendere il valore $A[i]$ in modo che il sottoalbero con radice di indice i diventi un max-heap. Ad ogni passo viene determinato il più grande tra $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$, l'indice del maggiore viene memorizzato:
- se $A[i]$ è il più grande, allora il sottoalbero con radice nel modo i è un max-heap e la procedura termina
- se uno dei due figli è l'elemento maggiore, $A[i]$ viene scambiato con $A[\text{max}]$ così da rispettare la proprietà: la funzione viene chiamata ricorsivamente

MAX-HEAPIFY (A, i)

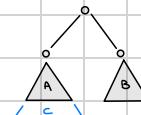
```

1. l = LEFT(i)
2. r = RIGHT(i)
3. if l <= A.heap-size and A[l] > A[i]
4.   massimo = l
5. else massimo = i
6. if r <= A.heap-size and A[r] > A[i]
7.   massimo = r
8. if massimo != i
9.   swap A[i] with A[massimo]
10.  MAX-HEAPIFY(A, massimo)

```

worst case: A e B sono uguali, ma l'albero è sbilanciato sulla

$$\text{sinistra. } A+B+C=N \quad A+C = \frac{2}{3}n$$



Il tempo di esecuzione in un sottoalbero di dimensione m con radice in un modo i è $\Theta(1)$ per sistemare le relazioni fra gli elementi $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$ più il tempo per eseguire MAX-HEAPIFY in un sottoalbero con radice in uno dei figli del modo i (chiamata ricorsiva). I sottoalberi dei figli hanno dimensione che non supera $2m/3$.

+ caso peggiore: l'ultima ziga dell'albero è piena esattamente a metà e la ricorsione diventa:

$$T(m) \leq T(2m/3) + \Theta(1) \Rightarrow T(m) = O(\log m)$$

Possiamo usare MAX-HEAPIFY dal basso verso l'alto per convertire un array $[1..m]$ con $m = A.length$, in un MAX-HEAP. Gli elementi nel Sott阵 A $[(\lfloor m/2 \rfloor + 1)..m]$ sono foglie dell'albero dunque ciascuno di essi è un heap di un solo elemento che possiamo usare come punto di partenza. Quindi partiamo dalle foglie e applichiamo MAX-HEAPIFY in modo rimontante dell'albero.

BUILD-MAX-HEAP (A)

```

1. A.heap-size = A.length
2. for i = A.length down to 1  $\Rightarrow O(n \log n)$ 
3.   MAX-HEAPIFY(A, i)

```

BUILD-MIN-HEAP (P)

```

1. A.heap-size = A.length
2. for i = (A.length/2) down to 1
3.   MIN-HEAPIFY(A, i)

```

MIN-HEAPIFY (A, i)

```

1. l = LEFT(i)
2. r = RIGHT(i)
3. if l <= A.heap-size and A[l] < A[i]
4.   min = l
5. else
6.   min = i
7. if r <= A.heap-size and A[r] < A[i]
8.   min = r
9. if min != i
10.  swap A[i] with A[min]
11.  MIN-HEAPIFY(A, min)

```

- Qual'è l'effetto di chiamare MAX-HEAPIFY(A, i) per $i > A.heap-size/2$?

: Significa che stiamo cercando di eseguire l'operazione su un nodo che è una foglia o si trova al di là delle foglie dell'heap.

In un heap binario completo tutti i nodi al di sotto dell'indice $A.heap-size/2$ sono nodi interni o radici di sottoalberi completi. Tutti i nodi dall'indice $A.heap-size/2$ in poi sono foglie o non entrano affatto. Dal momento che MAX-HEAPIFY opera in modo ascendente (portendo dall'alto verso il basso) eseguirlo su un nodo che è già una foglia non avrà effetti significativi perché il nodo analizzato non ha figli.

- Qual'è l'effetto di chiamare MAX-HEAPIFY(A, i) quando l'elemento $A[i]$ è maggiore dei suoi figli?
- : Non avrà alcun effetto sulla struttura dell'heap. Infatti la procedura è realizzata in modo tale che il passo in analisi sia maggiore dei suoi figli, dal momento che $A[i]$ è già maggiore dei suoi figli non è necessario eseguire alcuna operazione.

CODA DI PRIORITÀ

- Una delle applicazioni più diffuse dell'heap. Ci sono due tipi di code:
 - MAX-PRIORITÀ, basata sul MAX-HEAP
 - MIN-PRIORITÀ, basata sul MIN-HEAP

- Una coda di MAX-PRIORITÀ supporta queste operazioni:

- INSERT(S, x), inserisce l'elemento x nell'insieme S . $S = S \cup \{x\}$ $O(\log m)$
- MAXIMUM(S), restituisce l'elemento di S con la chiave più grande (priorità più alta)
- EXTRACT-MAX(S), rimuove e restituisce l'elemento di S con la chiave più grande $O(\log m)$
- INCREASE-KEY(S, x, k), aumenta il valore dell'elemento x al nuovo valore k , che si suppone sia almeno grande quanto il valore corrente della chiave dell'elemento x . $O(\log m)$

- Una coda di MIN PRIORITÀ supporta: INSERT, MINIMUM, EXTRACT-MIN, DECREASE-KEY

• HEAP-MAXIMUM (A)

1. return $A[1]$

• INCREASE-KEY (A, i, k)

1. if $k < A[i]$ then
2. error " $A[i] > k$ "
3. $A[i] = k$
4. while $i > 1$ and $A[\text{parent}(i)] < A[i]$ do
5. swap $A[i]$ with $A[\text{parent}(i)]$
6. $i = \text{parent}(i)$

• HEAP-EXTRACT-MAX (A)

1. if $A.\text{heap-size} < 1$
2. error
3. $\max = A[1]$
4. $A[1] = A[A.\text{heap-size}]$
5. $A.\text{heap-size} = A.\text{heap-size} - 1$
6. MAX-HEAPIFY($A, 1$)
7. return \max

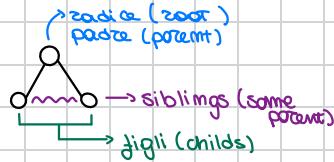
• INSERT-KEY (A, i, k)

1. if $k < A[i]$ then
2. error " $A[i] > k$ "
3. while $i > 1$ and $A[\text{parent}(i)] < A[i]$ do
4. swap $A[i]$ with $A[\text{parent}(i)]$
5. $i = \text{parent}(i)$

BINARY TREE

Alberi → struttura dati composta da:

- Nodi
- RADICI (root/head)
- FIGLI



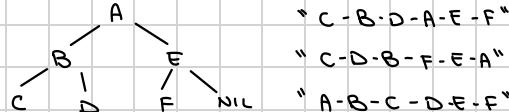
PROFONDITA': distanza tra un nodo x e la radice (lunghezza向日葵 da x a root)

LIVELLO: insieme dei nodi ad una determinata profondità

ALTEZZA: profondità massima di un nodo in un albero

Ogni nodo ha due figli: LEFT[x], RIGHT[x] → array

IN-ORDER → LEFT[x] | x | RIGHT[x]



POST-ORDER → LEFT[x] | RIGHT[x] | x

PRE-ORDER → x | LEFT[x] | RIGHT[x]

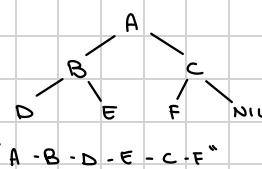
1. **PRE-ORDER?** (x) $\xrightarrow{\text{modo}}$

if ($x \neq \text{NIL}$)

print(x)

PRE-ORDER(LEFT[x])

PRE-ORDER(RIGHT[x])



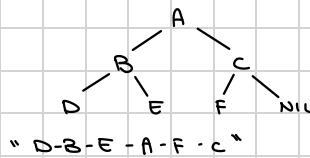
2. **IN-ORDER?** (x)

if ($x \neq \text{NIL}$)

IN-ORDER(LEFT[x])

print(x)

IN-ORDER(RIGHT[x])



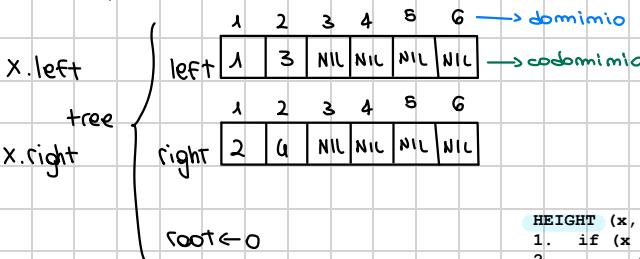
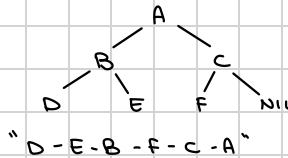
3. **POST-ORDER?** (x)

if ($x \neq \text{NIL}$)

POST-ORDER(LEFT[x])

POST-ORDER(RIGHT[x])

print(x)

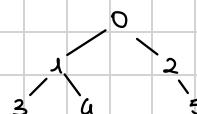


NUMBER-OF-NODES (x)

1. int c = 0
2. if ($x \neq \text{NIL}$)
3. c += NUMBER-OF-NODES(left[x])
4. c += NUMBER-OF-NODES(right[x])
5. return c
- 6.

HEIGHT (x , d) $\xrightarrow{\text{depth}}$

1. if ($x \neq \text{NIL}$)
2. if (left[x] != NIL or right[x] != NIL)
3. r = d
4. l = d
5. if (left[x] != NIL)
6. l = HEIGHT(left[x], d+1)
7. if (right[x] != NIL)
8. r = HEIGHT(right[x], d+1)
9. return max(l, r)
10. return d



Binary Search Tree

Sono STRUTTURE DATI che supportano operazioni sugli insiemi come: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT e DELETE.

Può essere utilizzato come: dizionario o come coda di priorità.

Le operazioni di base richiedono un tempo proporzionale all'h dell'albero (un albero completo con m nodi esige un $\Theta(\log m)$, mentre se un albero è una catena lineare di m nodi le op richiedono $\Theta(m)$)

Negli alberi binari ogni nodo ha al più due figli, può essere rappresentato da una struttura dati concatenata in cui ogni nodo è un oggetto:

PARENT	
LEFT	RIGHT

Se manca un figlio l'attributo viene sostituito con NIL. Se entrambi gli attributi LEFT e RIGHT sono NIL allora si tratta di una foglia. Il modo radice è l'unico modo nell'albero il cui attributo padre è NIL.

PROPRIETÀ DEGLI ALBERI BINARI DI RICERCA:

Sia x un nodo e Key il suo valore:

- y è un nodo nel Sottoalbero Sinistro di x, allora $y \text{ key} < x \text{ key}$
- z è un nodo nel Sottoalbero destro di x, allora $z \text{ key} > x \text{ key}$

$$\text{value}[x] > \text{value}[\text{LEFT}[x]]$$

$$\text{value}[x] < \text{value}[\text{RIGHT}[x]]$$

$$\downarrow \text{value}[x] > \text{value}[y] > \text{value}[z]$$

Questo consente di estrarre ordinatamente tutte le chiavi con l'algoritmo ricorsivo IN ORDER.

Se x è la radice di un Sottoalbero di m nodi, la chiamata INORDER(x) richiede il tempo $\Theta(m)$.

(la procedura viene richiamata ricorsivamente due volte per ogni nodo dell'albero (una per il dx) (una per il sx))

• IN ORDER, radice in mezzo

• PRE ORDER, radice per prima

• POST ORDER, radice alla fine

SEARCH(x, v)

```

1. if x = NIL or value[x] = v
2.   return x
3. if v < value[x]
4.   return SEARCH(left[x], v)
5. else
6.   return SEARCH(right[x], v)

```

O(h)

(l'altezza dell'albero)

MIN(x)

```

1. while left[x] != NIL
2.   x = left[x]
3. return x

```

MAX(x)

```

1. while right[x] != NIL
2.   x = right[x]
3. return x

```

SUCCESSOR (x)

```

1. if right[x] != NIL then
2.   return min(right[x]) // se il sottoalbero destro non è vuoto,
                           allora il successore di x è il nodo più a
                           sinistra nel sottoalbero destro
3. y = parent[x]
4. while y != NIL and x == right[y] do
5.   x = y
6.   y = parent[y]
7. return y

```

• DELETE, tre casi base:

1) il nodo da cancellare x non ha figli, quindi è una foglia

⇒ modifica il padre, x diventa NIL

2) x ha un solo figlio, sostituisco x con il figlio e lo elimino

3) x ha due figli, sostituisco x con il suo successore

(troviamo il successore y di x, che si trova nel sottoalbero destro

di x, y prende il posto di x, la parte restante del sottoalbero destro
originale diventa il sottoalbero destro di y e il sottoalbero sinistro

di y diventa il sinistro di y. Occorre considerare quando y è figlio

destro di x)

INSERT(T, z)

```

1. y = NIL
2. x = root[T]
3. while x != NIL do
4.   y = x
5.   if value[z] < value[x] then
6.     x = left[x]
7.   else
8.     x = right[x]
9. parent[z] = y
10. if y = NIL then
11.   root[T] = z // l'albero T era vuoto
12. else
13.   if value[z] < value[y]
14.     left[y] = z
15.   else
16.     right[y] = z

```

O(h)

```

DELETE (Node* root, int data) //pointer reference alla radice e il valore da eliminare
DELETE(T, z)
1. if left[z] = NIL or right[z] = NIL then //controlla se z ha al massimo un figlio
2.     y = z // ha al massimo un figlio
3. else //ha entrambi i figli
4.     y = successore(z) //trova il nodo più piccolo nel sottoalbero destro di z
5. if left[y] != NIL then
6.     x = left[y]
7. else
8.     x = right[y] // 5-8 assegna x al figlio sinistro di y, se presente, altrimenti lo assegna al figlio destro
9. if x != NIL then
10.    parent[x] = parent[y] //se x non è nullo, aggiorna il genitore di x con il genitore di y
11. if parent[y] = NIL then
12.     root[T] = x // se y non ha un genitore x diventa la radice dell'albero
13. else //y ha un genitore
14.     if y = left[parent[y]] then
15.         left[parent[y]] = x
16.     else
17.         right[parent[y]] = x //14-17 se y è figlio sx del genitore, il figlio sx diventa x, altrimenti il destro
18. if y != z then
19.     value[z] = value[y]
20. return y

```

III caso: trova il minimo nel Sottoalbero destro

- copiare il valore minimo nel modo da Cancellare
- eliminare il duplicato (il minimo nella posizione originale) dal sottoalbero destro

Alberi Rosso-Neri

- Gli ALBERI ROSSO-NERI rappresentano uno dei tanti modi in cui possiamo garantire un albero di ricerca per garantire che le operazioni elementari richiedono un tempo $O(\log n)$
- Gli alberi rosso-neri sono BST con un bit di memoria aggiuntivo per ogni nodo, dove registriamo il colore, Rosso o NERO.
- Tramite i colori si garantisce che ogni percorso dalla radice alle foglie non è più lungo del doppio di un qualsiasi altro percorso (dalla radice alle foglie) nell'albero \Rightarrow l'albero è approssimativamente BILANCIATO
- ogni nodo contiene: COLOR, KEY, LEFT, RIGHT

PROPRIETÀ

- 1) ogni nodo è ROSSO o NERO
- 2) la radice è NERA
- 3) ogni foglia NIL è NERA
- 4) se un nodo è ROSSO, allora entrambi i suoi figli sono NERI
- 5) Per ogni nodo, tutti i cammini semplici che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi MERI

- Definiamo altezza minima di un modo x , indicata con $lh(x)$, il numero di nodi meri lungo un percorso che inizia da x (ma non lo include) e finisce in una foglia.

LEMMA: L'altezza massima di un albero rosso-nero con m nodi interni è $2 \log(m+1)$

- Il sottoalbero con radice in un modo x qualsiasi contiene almeno $2^{lh(x)} - 1$ nodi interni.

DIMOSTRAZIONE PER INDUZIONE SULL'ALTEZZA DI X

- l'altezza di x è 0, allora è una foglia (NIL) \Rightarrow il sottoalbero con radice x contiene: $2^{lh(x)} - 1$ nodi $\rightarrow 2^0 - 1 = 1 - 1 = 0$ \square

- PASSO INDUTTIVO: l'altezza di x è maggiore di 0, allora è un modo interno e ha almeno due figli.

\Rightarrow ogni figlio ha altezza minore pari a: $-lh(x)$ (figlio rosso)

$-lh(x) - 1$ (figlio nero)

Siccome $lh(x) > lh(figlio(x))$, allora possiamo applicare l'ipotesi induttiva per concludere che ogni figlio ha almeno $2^{lh(x)-1} - 1$ nodi interni.

formula del padre con \leftarrow

l'altezza del figlio

Quindi il sottoalbero con radice x contiene almeno:

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 \Leftrightarrow 1 = 2^{bh(x)-1}$$

Sottoalbero
figlio sx Sottoalbero
figlio dx

- Indichiamo con h l'altezza dell'albero. Per la 4° proprietà (un nodo rosso ha entrambi i figli neri)

→ almeno metà dei nodi, in un qualsiasi percorso dalla radice ad una foglia (esclusa la radice) deve essere nera.

Di conseguenza $bh(\text{root})$ deve essere almeno $h/2$

$$2^{bh(x)-1} - 1 \leq m \quad 2^{h/2} - 1 \leq m \implies 2^{h/2} \leq m+1$$

(sottoalbero)

(padre)

$$\Rightarrow \log_2 2^{h/2} \leq \log_2 (m+1)$$

$$\frac{h}{2} \leq \log_2 (m+1)$$

$$h \leq 2 \log_2 (m+1) \quad \rightarrow \text{Complessità asintotica: non ci interessano le costanti} \Rightarrow h \leq \log_2 (m+1)$$

$T(m) =$ INSERT: $O(\log m)$

DELETE: $O(\log m)$

RB-INSERT(T, z)

```

1. y = T.nil
2. x = T.root
3. while x != T.nil
4.     y = x
5.     if z.key < x.key
6.         x = x.left
7.     else x = x.right
8. z.p = y
9. if y == T.nil
10.    T.root = z
11. else if z.key < y.key
12.    y.left = z
13. else y.right = z
14. z.left = T.nil
15. z.right = T.nil
16. z.color = RED
17. RB-INSERT-FIXUP(T, z)

```

B-Alberi

Self-Balancing Tree

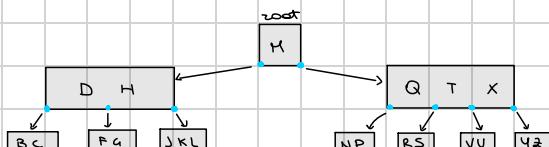
I B-Alberi sono alberi di ricerca bilanciati che operano bene sui dischi e sulle unità di memoria secondaria ad accesso diretto. Possono avere più di due figli, fino a diverse centinaia ed è solitamente determinato dalle caratteristiche dell'unità a disco utilizzata. Ogni B-Albero di M nodi ha altezza $O(\log m)$, può comunque essere più piccola rispetto a quella di un RB albero, dal momento che il grado di ramificazione (la base del logaritmo che esprime la sua altezza) può essere molto più grande.

Un B-albero è un albero radicato (con radice $\text{root}[T]$) ha le seguenti proprietà:

1) ogni nodo x ha i seguenti campi:

- $x.m$ è il numero di chiavi correntemente memorizzate in x
- le $x.m$ chiavi sono memorizzate in ordine crescente t.c. $x.Key_1 \leq x.Key_2 \leq \dots \leq x.Key_{x.m}$
- $x.left$ è un campo booleano con valore true se x è foglia, false altrimenti

2) ogni nodo interno x contiene anche $x.m+1$ PUNTATORI ($x.c_1, x.c_2, \dots, x.c_{x.m}$) ai suoi figli. I nodi foglia non hanno figli perciò i loro attributi c_i non sono definiti



3) La chiavi $x.key_i$ separano gli intervalli delle chiavi memorizzate in ciascun sottoalbero: se K_i è una chiave qualsiasi memorizzata nel sottoalbero con radice $C_i(x)$, allora $K_1 \leq x.key_1 \leq K_2 \leq x.key_2 \leq \dots \leq x.key_{x.m} \leq K_{x.m+1}$

4) Tutte le foglie hanno la stessa profondità, che è l'altezza h dell'albero

5) Ci sono limiti superiori e inferiori per il numero di chiavi che un nodo può contenere. Questi limiti possono essere espressi in termini di un intero $t \geq 2$ chiamato GRADO MINUTO del B-Albero:

a) ogni nodo, tranne la radice, deve avere almeno $t-1$ chiavi. Ogni nodo interno, tranne la radice, quindi ha almeno t figli. Se l'albero non è vuoto, la radice deve avere almeno una chiave.

b) ogni nodo può contenere al massimo $2t-1$ chiavi. Quindi, un nodo interno può avere al massimo $2t$ figli.
Diciamo che un nodo è pieno se contiene esattamente $2t-1$ chiavi.

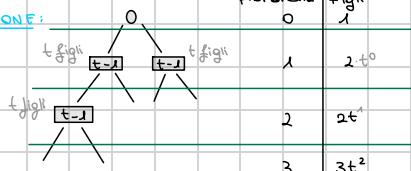
Il B-Albero più semplice si ha quando $t=2$. Ogni nodo interno ha 2, 3 o 4 figli. (Solitamente però t è molto più grande)

TEOREMA (altezza di un B-Albero)

Se $m \geq 1$, allora per qualsiasi B-Albero T di m chiavi, con altezza h e grado minimo $t \geq 2$, si ha

$$h \leq \log_t \frac{m+1}{2}$$

DIMOSTRAZIONE:



almeno $t-1$ chiavi per ogni nodo
 $\sum_{i=1}^n 2^{t-1}$
di nodi

$$m \geq 1 + (t-1) \sum_{i=1}^n 2^{t-1}$$

$$= 1 + 2(t-1) \left(\frac{t^h - 1}{t-1} \right) = 2t^h - 1$$

$$m \geq 2t^h - 1 \quad t^h \leq \frac{m+1}{2} \quad \log_t(t^h) \leq \log_t\left(\frac{m+1}{2}\right) \quad h \leq \log_t\left(\frac{m+1}{2}\right)$$

B-TREE-SEARCH (x, k)

```

1. i = 1
2. while i <= x.n and k > x.key^y
3.   i = i + 1
4.   if i <= x.n and k == x.key^i
5.     return (x, i)
6.   else if x.leaf
7.     return NIL
8.   else DISK-READ(x.c^i)
9.   return B-TREE-SEARCH(x.c^i, k)

```

ricerca come input un puntatore al nodo radice x di un sottoalbero e una chiave

k da cercare nel sottoalbero da chiavi da livello più alto ha quindi la forma

B-TREE-SEARCH($T.root, k$). Se k si trova nel B-Albero, la procedura restituisce la

coppia ordinata (y, i) , formata da un nodo y e da un indice i , tali che $y.key^i = k$,

altrimenti restituisce NIL.

$$T(m) = O(C \log_t m)$$

righe 1-3: trovano il più piccolo indice i tale che $K \leq x.key_i$, altrimenti assegnano $x.m+1$ a i .

righe 4-5: controllano se è stata trovata la chiave, e in tal caso la restituiscono.

righe 6-9: terminano la ricerca senza successo se x è una foglia, oppure ricorsivamente ricorrono nel sottoalbero di x , dopo aver eseguito l'operazione obbligatoria DISK-READ su tale nodo figlio

B-TREE-CREATE (T)

```

1. x = ALLOCATE-NODE()
2. x.leaf = TRUE
3. x.n = 0
4. DISK-WRITE(x)
5. T.root = x

```

Per costituire un B-Albero T , prima utilizziamo B-TREE-CREATE per creare una radice vuota e poi chiamiamo B-TREE-INSERT per aggiungere nuove chiavi. Si utilizza una procedura ausiliaria ALLOCATE-NODE, che alloca una pagina del disco da utilizzare come nuovo nodo. Nel tempo $O(1)$

$$T(m) = O(1)$$

Alberi di Intervalli (Grafo della Scena)

Un intervallo chiuso è una coppia ordinata di numeri reali $[t_1, t_2]$, con $t_1 \leq t_2$. Rappresentiamo un intervallo $[t_1, t_2]$ come un oggetto i avente i campi i.low = t_1 , estremo inferiore

i.high = t_2 , estremo superiore

Un intervallo $[t_1, t_2]$ rappresenta l'insieme $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$

Diciamo che gli intervalli i e i! si sovrappongono se $i.i! \neq \emptyset$, ovvero se $i.low \leq i!.high \& i!.low \leq i.high$



Due intervalli qualsiasi soddisfano la tautomia degli intervalli, ovvero una sola delle seguenti proprietà è vera:

- i e i! si sovrappongono
- i è a sinistra di i!: i!.high < i.low
- i è a destra di i!: i!.high < i.low

Un albero di intervalli è un albero rosso-moro che gestisce un insieme dinamico di elementi in cui ogni elemento x contiene un intervallo x.int.

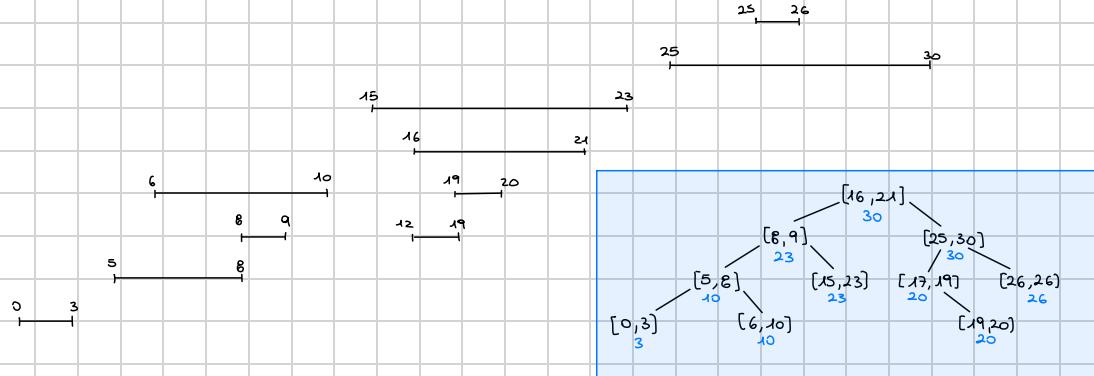
Gli alberi di intervalli supportano:

- INTERVAL-INSERT (T, x): aggiunge l'elemento x, il cui attributo int si suppone contenga un intervallo, all'albero di intervalli di T.
- INTERVAL-DELETE (T, x): rimuove l'elemento x, dall'albero di intervalli di T
- INTERVAL-SEARCH (T, i): restituisce un puntatore a un elemento x nell'albero di intervalli di T tale che x.int si sovrapponga all'intervalllo i; restituisce un puntatore alla sentinella T.nil se non esiste tale elemento nell'insieme.

Oltre agli intervalli, ogni nodo x contiene un valore x.max, ovvero il massimo tra tutti gli estremi degli intervalli memorizzati nel sottoalbero con radice in x. Se conosciamo l'intervallo x.int e i valori max dei figli del nodo x, possiamo determinare x.max.

$$x.\text{Max} = \max(x.\text{int}.high, x.\text{left}.max, x.\text{right}.max)$$

ESEMPIO: prima confrontiamo i low e dopo gli high



```

INTERVAL-SEARCH (T, i)
1. x = T.root
2. while x != T.nil e i non si sovrappone x.int
3.   if x.left != T.nil e x.left.max >= i.low
4.     x = x.left
5.   else
6.     x = x.right
7. return x

```

da ricerca di un intervallo che si sovrappone a i inizia con x nella radice dell'albero e prosegue verso il basso. Termina quando:

- viene trovato un intervallo che si sovrappone a i
- quando x punta alla sentinella T.nil

Ogni iterazione del ciclo di base impiega il tempo $O(1)$ e poiché l'altezza di un albero rosso-nero di m nodi è $O(\log m)$, l'algoritmo impiega il tempo $O(\log m)$.

(Se c'è più di un intervallo prende il primo che trovi)

Statistica d'Ordine

l'i-esima STATISTICA D'ORDINE di un insieme di m elementi è l'i-esimo elemento più piccolo.

• il MINIMO di un insieme di elementi è la PRIMA STATISTICA D'ORDINE ($i=1$)

• il MASSIMO è l'N-ESIMA STATISTICA D'ORDINE ($i=m$)

• la MEDIANA è il "punto di mezzo" dell'insieme

- m è dispari: la mediana è UNICA ed è $i=(m+1)/2$ indipendentemente dalla posita di m orario:
 M pari: le mediane sono due - 1) $i=m/2$
 2) $i=m/2+1$

- la MEDIANA INFERIORE, $i=\lfloor(m+1)/2\rfloor$
 - la MEDIANA SUPERIORE, $i=\lceil(m+1)/2\rceil$

PROBLEMA: selezionare l'i-esima statistica d'ordine da un insieme di m numeri distinti.

↳ INPUT: un insieme A di m numeri (distinti) e un intero i, con $1 \leq i \leq m$

↳ OUTPUT: l'elemento $x \in A$ che è maggiore esattamente di altri $i-1$ elementi di A.

Possiamo modificare gli alberi rosso-neri in modo che qualsiasi statistica d'ordine su un insieme dinamico possa essere determinata nel tempo $O(\log m)$; anche il RANGO di un elemento (la posizione che occupa nella sequenza ordinata degli elementi dell'insieme) può essere determinato nel tempo $O(\log m)$.

Un ALBERO PER STATISTICHE D'ORDINE T è un albero rosso-nero con un'informazione aggiuntiva in ogni nodo, $x.size$, ovvero il numero di nodi (interni) nel sottoalbero con radice in x, incluso x stesso. Quindi $x.size$ è la dimensione del sottoalbero, definendo la dimensione della sentinella a 0, quindi $T.nil.size = 0$, allora abbiamo l'identità $x.size = x.left.size + x.right.size + 1$.

In un albero per statistiche d'ordine non è richiesto che le chiavi siano distinte. In presenza di chiavi uguali definiamo il range come la posizione in cui l'elemento sarebbe elencato in un attraversamento simmetrico dell'albero.

• OS-SELECT(x, i) è un algoritmo che restituisce un puntatore al nodo che contiene l'i-esimo chiave più piccola nel sottoalbero con

radice in x.

```

OS-SELECT(x, i)
1. r = x.left.size + 1
2. if i == r
3.   return x
4. else if i < r
5.   return OS-SELECT(x.left, i)
6. else
7.   return OS-SELECT(x.right, i - r)

```

riga 1: calcolo il range del nodo x nel sottoalbero di radice x

x.left.size è il numero di nodi che precedono x in un attraversamento simmetrico del sottoalbero con radice in x.

riga 2: se $i=r$, allora il nodo x è l'i-esimo elemento più piccolo, quindi la

3 restituisce x.

riga 4: se $i < r$, allora l'i-esimo elemento più piccolo è nel sottoalbero destro di x.

Poiché ci sono r elementi nel sottoalbero con radice in x che precedono il sottoalbero destro di x, l'i-esimo elemento più piccolo è l' $(i-r)$ -esimo elemento più piccolo nel sottoalbero con radice in x.right.

- OS-RANK(T, x), dato un puntatore a un nodo x in un albero per statistiche d'ordine T , restituisce la posizione di x nell'ordine incrementale determinato da un attraversamento simmetrico dell'albero T .

OS-RANK(T, x)

```

1. r = x.left.size + 1
2. y = x
3. while y != T.root
4.     if y == y.p.right
5.         r = r + y.p.left.size +
6.         1
7.     y = y.p
7. return r
  
```

} Il ранг di x può essere considerato come il numero di nodi che precedono x , più 1 per se stesso.
All'inizio di ogni iterazione del ciclo while (3-6) r è il ранг di $x.key$ nel sottoalbero con radice nel nodo y .

Tavole Hash

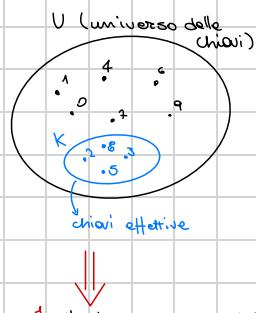
cap. 11.1, 11.2, 11.3, 11.4

- **TAVOLE A INDIRIZZAMENTO DIRETTO**, è una tecnica che funziona quando l'universo U delle chiavi è ragionevolmente piccolo.

$U = \{0, 1, \dots, m-1\}$ con m non troppo grande. Supponiamo che due elementi non possono avere la stessa chiave.

Potendo rappresentare l'insieme dinamico utilizziamo un array che indichiamo con $T[0 \dots m-1]$, dove ogni cella corrisponde a una chiave nell'universo U .

→ la cella K punta a un elemento dell'insieme con chiave K , se l'insieme non contiene l'elemento con chiave K , allora $T[K] = \text{NIL}$.



0	/
1	/
2	2
3	3
4	/
5	5
6	/
7	/
8	8
9	/Nil

array che indica con l'insieme delle chiavi:

$$U = \{0, 1, \dots, |U|-1\}$$

$$T = [0 \dots |U|-1]$$

• SEARCH(T, K)

$$T[X.Key] = X$$

• INSERT(T, x)

$$T[X.Key] = x$$

$$T(m) = O(1)$$

PROBLEMA: ¹ se l'universo delle chiavi U è

troppo grande, memorizzare una tonda.

T di dimensione $|U|$ è inefficiente in termini

di memoria + ² l'insieme K delle chiavi.

effettivamente memorizzare può essere così piccolo rispetto a U che la maggior parte dello spazio allocato per la tonda T sarebbe sprecato.

² le tavole hash richiedono meno spazio di una tonda ad indirizzamento diretto. Con le tavole ad indirizzamento diretto un elemento con chiave K

è memorizzato nella cella K , mentre in una tonda hash è memorizzato in una cella $h(K)$, ovvero utilizziamo una **FUNZIONE HASH** h per

calcolare la cella della chiave K . $h: U \rightarrow \{0, 1, \dots, m-1\}$. Diremo che $h(K)$ è il valore hash della chiave K .

• TAVOLE HASH

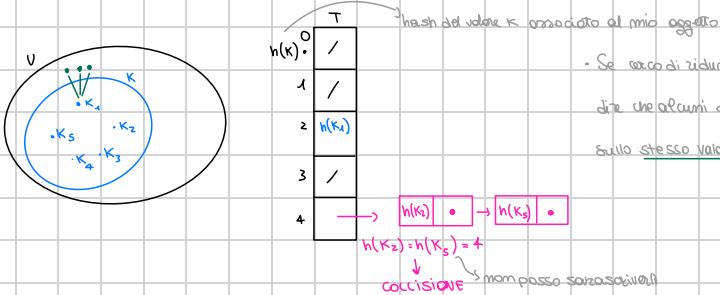
HASHING: funzione che mappa i dati in interi

TAVOLE HASH, utilizzate quando $|K| \ll |U|$

↓ funzione HASH $h: U \rightarrow \{0, 1, \dots, m-1\}$ (es. cittadino → cod. fiscale)
chiavi effettive

• SPAZIO RICHIESTO: $\approx \Theta(|K|)$

• TEMPO x le op.: $\approx O(1)$ (perché semplificano all'indirizzamento diretto).



- Se ci sono di ridurre l'insieme delle chiavi (es. 60000 → 1000), vuol dire che alcuni degli elementi sono uguali e vengono quindi mappati sulla stessa valore di hash *

• INSERT (T, x)

inserisce x in testa alla lista $T[h(\text{key}[x])]$ $O(1)$

• SEARCH (T, k)

ricerca un elemento con chiave k nella lista $T[h(k)]$

• SEARCH (T, x)

ricerca un elemento x nella lista $T[h(\text{key}[x])]$ $T(m)$ proporzionale alla lunghezza della lista nel caso peggiore

• DELETE (T, x)

dipende dalla Search

Cancella x dalla lista $T[h(\text{key}[x])]$ se le liste sono doppiamente concatenate → $O(1)$

Il compito della funzione hash è ridurre l'intervallo degli indici e quindi la dimensione dell'array. (l'array ha dimensione m invece che $|U|$)

PROBLEMA: due chiavi possono essere mappate nella stessa cella. ⇒ **COLLISIONE**
↓ soluzione

CONCATENAMENTO: poniamo tutti gli elementi che sono associati alla stessa cella in una lista concatenata

data una tonda hash T con m celle date sono memorizzati m elementi, definiamo **FATORE DI CARICO** d della tonda T il rapporto m/m , ossia il numero medio di elementi memorizzati in una lista.

le prestazioni dell'hashing nel caso medio dipendono dal modo in cui la funzione hash h distribuisce mediamente l'insieme delle chiavi da memorizzare tra le m celle.

HASHING UNIFORME SEMPLICE: qualsiasi chiave ha la stessa probabilità di essere mappata in una qualsiasi delle m celle, indipendentemente dalle celle in cui sono mappate le altre *

⇒ probabilità che scegliendo chiave a caso questa vada in cella i è $\frac{1}{m} \forall i$

TEOREMA: In una tonda hash le cui collisioni sono risolte con il concatenamento, una qualsiasi ricerca richiede $O(1+d)$ nel caso medio, nell'ipotesi di hashing uniforme semplice.

• se $m = O(m)$, allora $d = \frac{m}{m} = O(m) = O(1)$

⇒ ricerca in $O(1)$ in media

⇒ tutte le operazioni di dizionario in $O(1)$ (in media)

• **COME COSTRUIRE BUONE FUNZIONI HASH:** una buona funzione hash soddisfa l'ipotesi dell'hashing uniforme semplice*. Altre applicazioni però potrebbero richiedere proprietà più vincolanti, ad esempio richiede che le chiavi che sono vicine forniscano valori hash che siano distanti, in questo caso utiliziamo l'**HASHING UNIVERSALE**.

OSS: la maggior parte delle funzioni hash suppone che l'universo delle chiavi sia l'insieme dei numeri naturali, se le chiavi non sono numeri le trasformo.

$$pt \rightsquigarrow \text{ASCII}(p) = 12 \quad \text{ASCII}(t) = 116 \quad \text{scrivo pt in base 128: } (112 \cdot 128) + 116 = 14452$$

-**METODO DELLA DIVISIONE**: una chiave K viene associata a una delle m celle prendendo il resto della divisione fra K e m:

$$h(K) = K \bmod m \quad \text{es. } K=100 \quad m=12 \quad h(K)=4$$

di solito evitiamo certi valori di m, ad esempio non dovrebbe essere una potenza di 2, perché se $m=2^k$, allora $h(K)$ rappresenta proprio i p bit meno significativi di K.

Una buona scelta per m è un numero primo non troppo vicino a una potenza esatta di 2.

ES: dobbiamo allocare una tavola hash per contenere $m=2000$ stringhe di caratteri, dove ogni carattere ha 8 bit.

riteniamo accettabile esaminare in media ≈ 3 elementi in una ricerca senza successo, allora dobbiamo una tavola hash di dimensione $m=701$

→ numero primo vicino a $\frac{m}{d} = \frac{2000}{3}$ ma non a una potenza di 2

$$\Rightarrow h(K) = K \bmod 701$$

-**METODO DELLA MOLTIPLICAZIONE**: moltiplichiamo la chiave K per una costante A nell'intervallo $0 < A < 1$ ed estraiamo la parte frazionaria di KA (avendo, ci concateniamo sulla parte decimale del risultato). Poi moltiplichiamo per m e prendiamo la parte intera inferiore del risultato.

$$h(K) = l(m(KA \bmod 1))$$

→ $KA - LKA$

Qui il valore di m non è critico e solitamente viene scelto come una potenza di 2.

-**HASHING UNIVERSALE**: **PROBLEMA** → Per una qualsiasi funzione hash, potrebbe succedere che m chiavi vengono mappate tutte alla stessa cella, generando un tempo medio di ricerca pari a $\Theta(m)$.

→ **SOLUZIONE**: salvo CASUALMENTE la funzione hash, indipendentemente dalle chiavi che devono essere memorizzate.

Sia H una collezione finita di funzioni hash che associano un dato universo U di chiavi all'intervallo $\{0, 1, \dots, m-1\}$. Questa collezione è detta UNIVERSALE se $\forall K, L \in U$ il numero di funzioni hash H per le quali $h(K) = h(L)$ è al massimo $\lceil \frac{|H|}{m} \rceil$.

coppia di chiavi distinte

$$\forall K, L \in U \text{ con } K \neq L, \lceil |H| : h(K) = h(L) \rceil \leq \frac{|H|}{m}$$

→ scegliendo a caso una funzione da H , la probabilità di collisione tra K e L è $\leq \frac{1}{m} \cdot \frac{1}{m-1} = \frac{1}{m}$

-**TEOREMA**: Utilizzando l'hashing universale e la risoluzione di collisioni tramite concatenamento in una tavola con m celle inizialmente vuote, occorre $\Theta(m)$ per eseguire una qualsiasi sequenza di m operazioni INSERT, SEARCH, DELETE con $O(m)$ INSERT.

-**INDIRIZZAMENTO APERTO**: tutti gli elementi sono memorizzati nella tavola hash stessa, ovvero ogni cella della tavola contiene un elemento dell'insieme di monico o la costante NIL. Non ci sono liste libere elementi memorizzati all'esterno della tavola ed è possibile riempire la tavola al punto che non è possibile fare altri inserimenti.

→ **CONSEGUENZA**: il fattore di collasso d non supera mai 1.

→ **VANTAGGIO**: esclude i puntatori calcolando la sequenza delle celle da esaminare.

→ libera memoria che viene impiegata nell'affrire un maggior numero di celle nella tavola.

-**INSERIMENTO**: esaminiamo in successione le posizioni della tavola (**ISPEZIONE**) finché non troviamo una cella vuota in cui inserire la chiave. Voglio evitare $O(m)$ per la ricerca, perciò faccio dipendere la sequenza d'ispezione dalla chiave.

Quindi estendiamo la funzione hash im modo da includere l'ordine di ispezione (a partire da 0) come secondo input.

$$\text{NW} \quad h: U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$$

Si richiede che per ogni chiave K , la sequenza d'ispezione $\langle h(K, 0), h(K, 1), \dots, h(K, m-1) \rangle$ sia una permutazione di $\langle 0, 1, \dots, m-1 \rangle$, in modo che ogni cella possa essere considerata come possibile cella in cui inserire una nuova chiave mentre la tavola si riempie.

- Supponiamo che gli elementi da inserire siano chiavi (la chiave K è identica all'elemento).

HASH-INSERT(T, k)

```
1. i = 0
2. repeat
3.   j = h(k, i)
4.   if T[j] == NIL
5.     T[j] = k
6.     return j
7.   else i = i + 1
8. until i == m
9. error "overflow della tavola hash"
```

L'algoritmo che ricerca la chiave K esamina la stessa sequenza di celle che ha esaminato l'algoritmo di inserimento quando ha aggiunto K . Però la ricerca può terminare quando trova una cella vuota, perché la chiave K sarebbe stata inserita lì e non dopo (supponendo che le chiavi non vengano cancellate).

HASH-SEARCH(T, k)

```
1. i = 0
2. repeat
3.   j = h(k, i)
4.   if T[j] == k
5.     return j
6.   i = i + 1
7. until T[j] == NIL or i == m
8. return NIL
```

→ in tal caso una soluzione potrebbe essere quella di mettere la cella come

"DELETED" anziché NIL.

↓
quando si ha la necessità di eliminare le chiavi si preferisce il concatenamento,
in quanto nell'indirizzamento aperto i tempi di ricerca non dipendono più dal fattore
di carico d .

HASHING UNIFORME: ogni chiave ha la stessa probabilità di avere come sequenza di ispezione una delle $m!$ permutazioni di

$$\langle 0, 1, \dots, m-1 \rangle$$

→ estende il concetto di HASHING UNIFORME SEMPLICE, in quanto produce un'intera sequenza di ispezione,
piuttosto che un singolo numero.

Per implementare l'hashing uniforme vengono utilizzate delle approssimazioni accettabili, come il DOPPIO HASHING, in quanto
è difficile implementare il vero e proprio hashing uniforme.

DOPPIO HASHING: usa una funzione hash della forma $h(K, i) = (h_1(K) + ih_2(K)) \bmod m$

h_1, h_2 funzioni ausiliarie

Nell'ispezione la posizione di partenza è determinata da n_1 e la distanza fra le posizioni successive da n_2 .

Buone scelte:

- m potenza di 2 e definire n_2 in modo che generi un numero dispari
- m primo e definire n_2 in modo che generi un numero intero positivo minore di m .

Esempio: scegliamo $1 \Rightarrow h_1(K) = K \bmod m \quad h_2(K) = 1 + (K \bmod m^2)$, con $m^2 = m-1$

$$K = 123456, m = 701, m^2 = 700 \Rightarrow h_1(K) = 80 \quad h_2(K) = 257$$

Quindi l'ispezione inizia dalla posizione 80 e si ripete ogni 257 celle (modulo m)

TEOREMA: Nell'ipotesi di hashing uniforme, data una tavola hash a impiantamento aperto con $d \cdot \frac{m}{M} < 1$, il numero atteso di ispezioni è: $\leq \frac{1}{1-d}$

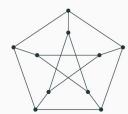
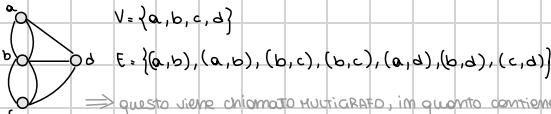
GRAFI

Un grafo $G = (V, E)$ consiste di:

- un insieme $V = \{v_1, \dots, v_m\}$ di vertici (o nodi);
- un insieme $E = \{(v_i, v_j) : v_i, v_j \in V\}$ di coppie (non ordinate) di vertici (distinti), dette archi.

v_i e v_j sono detti estremi dell'arco (v_i, v_j)

ES.



Albero

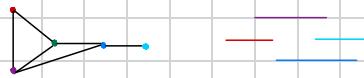
Grafo di Petersen

- **GRAFO PLANARE**: viene rappresentato come:

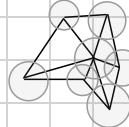
- vertici punti in \mathbb{R}^2
- arco $(u, v) \sim$ curva tra u e v senza autointersezioni
- curve corrispondenti ad archi distinti si intersecano al più negli estremi

- **GRAFO D'INTERSEZIONE**: Sia $S = \{S_1, \dots, S_m\}$ una famiglia di insiemi. Il grafo d'intersezione di S è il grafo $G = (V, E)$ i cui vertici corrispondono agli elementi di S e tale che $(v_i, v_j) \in E$ se e solo se $S_i \cap S_j \neq \emptyset$.

- **GRAFO D'INTERVALLI**: grafo d'intersezione di una famiglia di intervalli chiusi in \mathbb{R}



- **GRAFO DI DISCHI**: grafo d'intersezione di una famiglia di dischi chiusi in \mathbb{R}^2



def. Un **insieme indipendente** in un grafo $G = (V, E)$ è un sottoinsieme $I \subseteq V$ tale che, per ogni coppia $u, v \in I$, si ha che $(u, v) \notin E$.

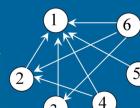
In parole, i vertici in I sono a due a due non adiacenti.

def. Un **grafo diretto** $D = (V, A)$ consiste di:

- un insieme $V = \{v_1, \dots, v_m\}$ di vertici o nodi;
- un insieme $A = \{(v_i, v_j) : v_i, v_j \in V\}$ di coppie ordinate di vertici, dette archi diretti. d'arco (v_i, v_j) si dice uscente da v_i e entrante in v_j .

Esempio: Disegnare il grafo diretto che ha come vertici i primi 6 numeri interi, e ha un arco diretto da x verso y se $x \neq y$ e x è un multiplo di y

$$\Rightarrow V = \{1, \dots, 6\}, A = \{(2,1), (3,1), (4,1), (5,1), (6,1), (4,2), (6,2), (6,3)\}$$



TERMINOLOGIA:

ES:

- $M = \text{numero di vertici}$ (es. $M=10$)

- $m = \text{numero di archi}$ (es. $m=13$)

- due nodi sono detti **ADIACENTI** se sono collegati da un arco (es. l e i)

- un arco (x,y) si dice **INCIDENTE** ad x e ad y (gli **ESTREM**)

- il **GRADO** di un nodo è il numero di archi ad esso incidenti. (es. i ha grado 4; $\delta(i)=4$) $\Rightarrow \sum_{v \in V} \delta(v) = 2m$

- il **grado del grafo** $\Delta(G)$ è pari al **massimo** tra tutti i gradi dei suoi vertici (es. $\Delta(G)=7$)

- un **CAMMINO** in $G=(V,E)$ tra una coppia di nodi (x,y) è una sequenza alternata di nodi e di archi in G che parte da x e arriva in y ,

dove ogni arco è incidente ai due nodi tra cui è compreso nel cammino:

$$\langle v_{i_1} = x, (v_{i_1}, v_{i_2}), v_{i_2}, (v_{i_2}, v_{i_3}), v_{i_3}, \dots, v_{i_{k-1}}, (v_{i_{k-1}}, v_{i_k}), v_{i_k} = y \rangle$$

- un cammino si dice **SEMPLICE** se i nodi che lo compongono sono distinti

- la **LUNGHEZZA** di un cammino è data dal numero di archi che lo compongono (es. $\langle l, i, e, c, b, a \rangle$ cammino SEMPLICE,

LUNGHEZZA: 5, tra l e a)

- se il grafo è **DIRETTO**, il cammino deve rispettare il verso di orientamento degli archi e verrà detto **CAMMINO ORIENTATO**.

- la lunghezza del più **CORTO CAMMINO (SEMPLICE)** tra due nodi si dice **DISTANZA** (es. l ed i hanno distanza 1, mentre l ed a hanno distanza 4) Un nodo è a distanza 0 da sé stesso.

- Il **DIAMETRO** di un grafo è la massima distanza tra due nodi del grafo (es. diametro 4 (distanza tra l e a))

- Se esiste un cammino tra ogni coppia di nodi, allora il grafo si dice **CONNESSO**, altrimenti si dice **DISCONNESSO**.



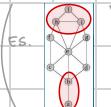
grafo disconnesso formato da due componenti connesse

- Un grafo disconnesso ha diametro **INFINITO**.

- Un cammino **CHIUSO** (avendo un cammino da un nodo a sé stesso) che non contiene altre ripetizioni di nodi si dice **CICLO**.

(es. $\langle l, i, e, h, l \rangle$)

- Un grafo $H=(V', E')$ è un **SOTTOGRAFO** di $G=(V, E) \iff V' \subseteq V$ e $E' \subseteq E$ (es.



- SOTTOGRAFI DEGENERI** di G : $H = \emptyset$ e $H = G$

- Dato un grafo $G=(V, E)$, il **SOTTOGRAFO INDOTTO** da un insieme di nodi $V' \subseteq V$ è il grafo $H[V'] = (V', E')$ ove $E' = \{(x, y) \in E \cdot x, y \in V'\}$

(es. il sottografo indotto da $V' = \{l, h, n, b, a\}$ è $H[V'] = (V', E' = \{(l, n), (l, i), (h, i), (b, a)\})$ ($H[V]$ non è connesso))

def **CRITERIO DI CONNESSIONE**: Sia $G=(V, E)$ un grafo. Per ogni $X \subseteq V$, denotiamo con $\delta(X)$ il sottoinsieme di E contenente gli archi con un estremo in X e l'altro in $V \setminus X$



LEMMA: Un grafo $G=(V, E)$ è connesso se e solo se $\delta(X) \neq \emptyset$ per ogni $\emptyset \neq X \subset V$

• **GRAFO TOTALMENTE DISCONNESSO** è un grafo $G=(V, E)$ tale che $V \neq \emptyset$ ed $E = \emptyset$



• **GRAFO COMPLETO** (o **CLIQUE**): è un grafo tale che per ogni coppia di nodi esiste un arco che li congiunge. Il grafo completo con m nodi verrà indicato con K_m



$$\Rightarrow |E| = m \cdot (m-1)/2$$

→ un grafo senza **CAPPI** (archi da un nodo a se stesso) ma **ARCHI PARALLELI** può avere un numero di archi m compreso tra 0 e $m(m-1)/2 = \Theta(m^2)$. Se il grafo è **CONNESSO**, allora necessariamente $m \geq m-1$. Quindi per grafi connessi:

$$m-1 \leq m \leq m(m-1)/2, \text{ cioè } m = \Omega(m) \text{ ed } m = O(m^2)$$

def Se $m = \Omega(m^2)$, il grafo si dice **DENSO**, mentre se $M = O(m)$, si dice **SPARSO**.

N.B. se un grafo ha $m \geq m-1$ archi, non è detto che sia连通的 (condizione necessaria, ma non sufficiente).

def. Un grafo $G = (V, E)$ si dice **EULERIANO** se e solo se contiene un cammino (non semplice, in generale) che passa una e una sola volta su ciascun arco in E .

TEOREMA DI EULER: Un grafo $G = (V, E)$ CONNESSO è Euleriano se e solo se ha tutti i nodi di grado pari, oppure se ha esattamente due nodi di grado dispari.

def. Un **ALBERO** è un grafo连通的 ed aciclico. Una **FORESTA** è un grafo aciclico.

LEMMA: Sia $G = (V, E)$ una foresta con p componenti connesse. Allora $|V| - |E| + p$

TEOREMA: Sia $G = (V, E)$ un grafo. Le seguenti affermazioni sono equivalenti:

- 1) G è un albero (connesso ed aciclico)
- 2) G contiene un unico connettore tra ogni coppia di vertici
- 3) G è connesso, ma il grafo ottenuto rimuovendo un qualsiasi arco da E è disconnesso
- 4) G è aciclico, ma il grafo ottenuto aggiungendo un qualsiasi arco ad E contiene un ciclo.
- 5) G è aciclico e $|E| = |V| - 1$
- 6) G è connesso e $|E| = |V| - 1$

def. Un **ALBERO RADICATO** è un albero in cui uno dei vertici si distingue da tutti gli altri; tale vertice è detto **RADICE**.

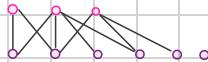
def. Siano T un albero radicato con radice r e x un qualsiasi vertice di T .

Un **ANTENUTO** di x è un qualsiasi vertice nell'unico connettore tra r e x . Se y è un antenuto di x , allora x è un **DISCENDENTE** di y .

Il **SOTTOALBERO** con radice in $x(T)$ è l'albero insito dai discendenti di x con radice in x .

Se l'ultimo arco nell'unico connettore tra r e x è (y, x) , allora y è il **PADRE** di x e x è un **FIOGLIO** di y .

GRAFI BIPARTITI è un grafo $G = (V = (A, B), E)$ tale che ogni arco ha come estremi un nodo in A ed un nodo in B .



• un grafo bipartito si dice **COMPLETO** se per ogni $x \in A$ ed $y \in B$, $(x, y) \in E$

• con $K_{a,b}$ si indica il **grafo bipartito completo di ordine** (a, b) , ovvero tale che $|A|=a$ e $|B|=b$

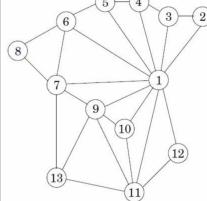
Problemi e Algoritmi sui Grafi

• COLORAZIONE DI UNA MAPPA



Problema:
colorare una mappa usando il minimo numero di colori in modo che due stati confinanti non abbiano lo stesso colore

una mappa può essere rappresentata come un **GRAFO PLANARE**



PROBLEMA: colorare i nodi di un grafo planare usando il minimo numero di colori in modo che due nodi adiacenti non abbiano lo stesso colore.

Def. Il minimo numero di colori per cui una tale colorazione esiste è detto **NUMERO CHROMATICO** di G e si denota con $\chi(G)$.

OSS. G è bipartito $\Leftrightarrow \chi(G) \leq 2$

• PROBLEMA:

Si devono fissare le date di un insieme di esami sotto il vincolo che certi esami non possono essere svolti lo stesso giorno (perché esami dello stesso anno e corso di laurea, o usano la stessa aula multimediale, ecc.). Si vuole minimizzare il numero di giorni utilizzati per fare esami.

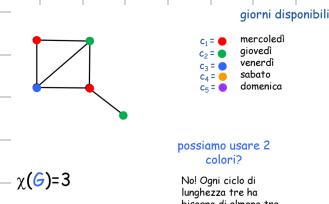
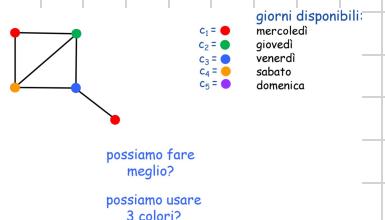


Definisci il seguente grafo: 1- un nodo per ogni esame

2- se due esami sono in conflitto

aggiungi l'arco fra i nodi corrispondenti

Colorare i nodi del grafo risultante usando il minimo numero di colori in modo che due nodi adiacenti non abbiano lo stesso colore \rightarrow Esami/nodi in conflitto non possono essere svolti/colorati lo stesso giorno/color.



Calcolare $\chi(G)$ è un problema algorithmico difficile, anche se si sa già il numero minimo di colori (lo è).

→ progettiamo un algoritmo veloce che colora il grafo usando pochi colori anche se non un numero minimo

FIRST-FIT COLORING

1. Dai un ordine arbitrario ai nodi v_1, v_2, \dots, v_n
2. Dai un ordine arbitrario ai colori c_1, c_2, \dots
3. Per $i = 1, 2, \dots, n$
4. assegna a v_i il primo colore (nell'ordine) che è ammissibile

Rappresentazione dei Grafi

Due metodi standard:

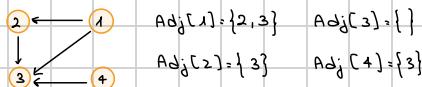
- **LISTE DI ADIACENZA**, permettono di rappresentare in modo compatto i grafhi sparsi, ovvero quelli in cui $|E| \ll |V|^2$.

- **MATRICE DI ADIACENZA**, utilizzate per rappresentare un grafo denso ($|E|$ è prossimo a $|V|^2$), oppure quando dobbiamo dire rapidamente se c'è un arco che collega due vertici particolari.

Possono essere applicati sia ai grafi orientati, sia a quelli non orientati.

• La rappresentazione con liste di adiacenza di un grafo $G = (V, E)$ consiste in un array Adj di $|V|$ liste, una per ogni vertice in V .

Per ogni $u \in V$, la lista di adiacenza $Adj[u]$ contiene tutti i vertici v tali che esiste un arco $(u, v) \in E$. Quello $Adj[u]$ include tutti i vertici adiacenti a u in G . \Rightarrow le liste di adiacenza rappresentano gli archi di un grafo



- Se G è un grafo orientato, la somma delle lunghezze di tutte le liste di adiacenza è $|E|$, perché un arco della forma (u, v) è rappresentato inserendo v in $Adj[u]$.

- Se G è un grafo non orientato, la somma delle lunghezze di tutte le liste di adiacenza è $2|E|$, perché se (u, v) è un arco non orientato, allora v appare nella lista di adiacenza di u e viceversa.

- PROBLEMA: memoria richiesta $\Theta(V + E)$.

- Queste liste possono essere adattate per rappresentare anche i **GRAFI PESATI**, ovvero grafi per i quali ogni arco ha un **PESO** associato,

tipicamente dato da una **funzione peso** $w: E \rightarrow \mathbb{R}$

es. $G = (V, E)$ grafo pesato con la funzione peso w . Il peso $w(u, v)$ dell'arco $(u, v) \in E$ viene memorizzato assieme al vertice v nella lista di adiacenza di u .

SVANTAGGIO: non c'è un modo più veloce per determinare se un particolare arco (u, v) è presente nel grafo che cercare v nella lista di adiacenza $\text{Adj}[u]$.

Possiamo rappresentare il grafo con una **MATRICE DI ADIACENZA**

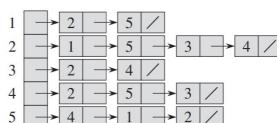
- per la **RAPPRESENTAZIONE CON MATERICI DI ADIACENZA** di un grafo $G = (V, E)$ si suppone che i vertici Siano numerati $1, 2, \dots, |V|$ in modo arbitrario. La rappresentazione di un grafo G consiste in una matrice $A = (a_{ij})$ di dimensione $|V| \times |V|$ tale che:

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{negli altri casi} \end{cases}$$

ES. GRAFO NON ORIENTATO:



(a)



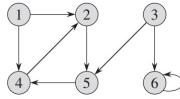
(b)

RAPP. MATRICE DI ADIACENZA

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

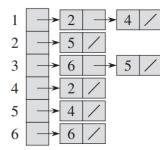
(c)

GRAFO ORIENTATO:



(a)

RAPP. LISTA DI ADIACENZA



(b)

RAPP. MATRICE DI ADIACENZA

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

- da matrice di adiacenza di un grafo richiede $\mathcal{O}(V^2)$ memoria, indipendentemente da $|E|$.

- Poiché (u, v) e (v, u) rappresentano lo stesso arco in un grafo non orientato, la matrice A di un grafo non orientato è uguale alla sua trasposta $A = A^T$. Alcune applicazioni memorizzano soltanto gli elementi che si trovano sopra e lungo la diagonale riducendo la memoria richiesta quasi della metà.

- le matrici di adiacenza possono essere utilizzate per i grafici pesati.

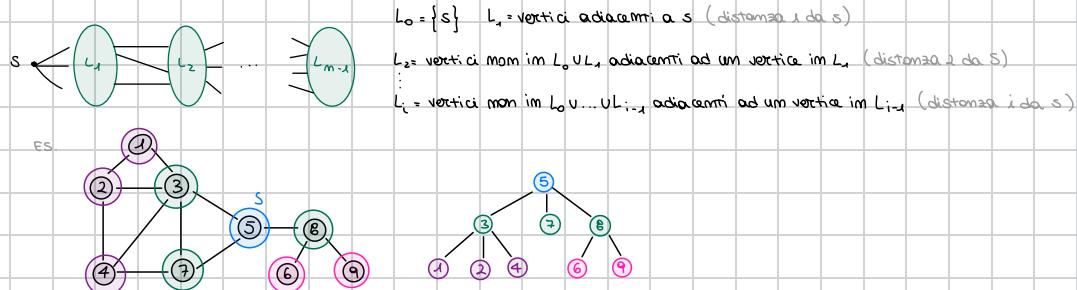
es. $G = (V, E)$ grafo pesato con la funzione peso w , il peso $w(u, v)$ dell'arco (u, v) è w viene memorizzato come l'elemento nella riga u e nella colonna v della mat. Se un arco non esiste — NULL

— $0.0 .00$ (preferibile)

Visita in Ampiezza (BFS)

Dato un grafo $G = (V, E)$ e un vertice distinto s , detto **SORGENTE**, la visita in ampiezza ispeziona sistematicamente gli archi di G per "scoprire" tutti i vertici che sono raggiungibili da s .

- calcola la distanza (il minimo numero di archi) da s a ciascun vertice raggiungibile da s .
- genera un albero BF (breadth-first tree) con radice s che contiene tutti i vertici raggiungibili. Per ogni vertice v raggiungibile da s , il cammino semplice nell'albero BF che va da s a v corrisponde a un "cammino minimo" da s a v in G .
- opera sui grafi orientati e non.
- l'algoritmo di **PRIM** per l'albero di connessione minima e l'algoritmo di **DJIKSTRA** si basano su concetti simili a quelli della visita in ampiezza.



COME FUNZIONA:

- per tenere traccia dei vertici già visitati colora i vertici di: BIANCO, vertici mai visitati (inizialmente tutti i vertici sono bianchi) CIRICO, potrebbero avere qualche modo bianco adiacente NERO, tutti i vertici adiacenti sono neri, quindi già scoperti
- un vertice viene scoperto quando viene incontrato per la prima volta, a quel punto non è più bianco.
- il colore di ogni vertice $u \in V$ è memorizzato nell'attributo $u.\text{color}$ e il predecessore di u è memorizzato in $u.\text{p}$ da distanza dalla sorgente s al vertice u calcolata in $u.d$. L'algoritmo utilizza una coda Q con schema FIFO per gestire i vertici già.

BFS(G, s)

- | | |
|---|--|
| <ol style="list-style-type: none"> 1. for ogni vertice $u \in G.V - \{s\}$ 2. $u.\text{color} = \text{WHITE}$ 3. $u.d = \infty$ 4. $u.\pi = \text{NIL}$ 5. $s.\text{color} = \text{GRAY}$ 6. $s.d = 0$ 7. $s.\pi = \text{NIL}$ 8. $Q = \emptyset$ 9. $\text{ENQUEUE}(Q, s)$ | <ol style="list-style-type: none"> 10. while $Q \neq \emptyset$ 11. $u = \text{DEQUEUE}(Q)$ 12. for ogni $v \in \text{Adj}[u]$ 13. if $v.\text{color} == \text{WHITE}$ 14. $v.\text{color} = \text{GRAY}$ 15. $v.d = u.d + 1$ 16. $v.\pi = u$ 17. $\text{ENQUEUE}(Q, v)$ 18. $u.\text{color} = \text{BLACK}$ |
|---|--|

righe 1-4: ad eccezione del vertice sorgente, tutti i vertici: - vengono colorati di bianco

- la distanza da s viene impostata ∞

- il predecessore viene impostato a NIL

riga 5: s colorato di grigio // dobbiamo ancora analizzare la sua lista di adiacenza

righe 6-7: inizializza la distanza da s a se stesso 0 e assegna al predecessore di s NIL

righe 8-9: inizializziamo la coda Q che contiene solo s

righe 10-18: il while si ripete finché restano vertici grigi le cui liste di adiacenza non sono ancora completamente esaminate.

righe 12-17: il for esamina ciascun vertice v nella lista di adiacenza di u. Se v è bianco, l'algoritmo ha scoperto un nuovo vertice,

lo colora di grigio, v.d viene impostato come la distanza del padre +1, memorizza il padre in v.r e mette v in fondo alla coda Q.

riga 18: $\text{Adj}[u]$ è stata esaminata quindi u.color = black

• TEMPO DI ESECUZIONE BFS

- operazioni di inserimento e cancellazione dalla coda: $O(1) \Rightarrow$ tempo totale alle op. con la coda $O(V)$, poiché la lista di ciascun vertice viene ispezionata quando il vertice viene rimosso, ogni lista viene ispezionata al più una volta.

- il tempo impiegato per ispezionare le liste è $O(E)$, poiché la somma delle lunghezze di tutte le liste è $O(E)$.

- il costo del for (1-4) di inizializzazione è $O(V)$

$$\Rightarrow T(n) \text{ totale} \in O(V+E)$$

• CAMMINI MINIMI: definiamo la distanza di cammino minimo $\delta(s, v)$ da s a v come il numero minimo di archi in un cammino qualsiasi che va dal vertice s al vertice v, se non c'è un cammino da s a v, allora $\delta(s, v) = \infty$.

Un cammino di lunghezza $\delta(s, v)$ da s a v è detto cammino minimo da s a v.

LEMMA: Se $G = (V, E)$ è un grafo (orientato o non) e $s \in V$ è un vertice arbitrario, allora per qualsiasi arco $(u, v) \in E$ si ha:

$$\delta(s, v) \leq \delta(s, u) + 1$$

LEMMA: Sia $G = (V, E)$ un grafo (orientato e non) e supponiamo che BFS venga eseguita sul grafo G da un dato vertice sorgente sEV.

Allora, al termine di BFS, per ogni vertice $v \in V$, il valore v.d calcolato dalla procedura soddisfa la relazione $v.d \geq \delta(s, v)$

LEMMA: Supponiamo che durante l'esecuzione di BFS su un grafo $G = (V, E)$, la coda Q contenga i vertici $\langle v_1, v_2, \dots, v_r \rangle$, dove v_i è l'inizio della coda Q e v_r è la fine. Allora, $v_r.d \leq v_i.d + 1$ e $v_i.d \leq v_{i+1}.d$ per $i = 1, 2, \dots, r-1$

COROLARIO: Supponiamo che i vertici v_i e v_j siano inseriti nella coda durante l'esecuzione di BFS e che v_i sia inserito prima di v_j . Allora, $v_i.d \leq v_j.d$ nell'istante in cui v_j viene inserito nella coda. (conseguenza immediata del lemma³)

TEOREMA (Correttezza della visita in ampiezza)

Sia $G = (V, E)$ un grafo (orientato o non) e supponiamo che la procedura BFS venga eseguita sul grafo G da un dato vertice sorgente sEV. Allora, durante la sua esecuzione, BFS scopre tutti i vertici $v \in V$ che sono raggiungibili dalla sorgente s e, alla fine dell'esecuzione, $v.d = \delta(s, v)$ per ogni $v \in V$. Inoltre, per qualsiasi vertice $v \neq s$ che è raggiungibile da s, uno dei cammini minimi da s a v è un cammino minimo da s a v, seguito dall'arco $(v.r, v)$.

DIM. 1) Supponiamo, per assurdo, che $\exists v \in V$ t.c. $v.d \neq \delta(s, v)$. Sia v il vertice con minimo $\delta(s, v)$ a ricevere il valore errato d. $\rightsquigarrow s \neq v$

$v.d > \delta(s, v)$ poiché $(v.d \leq \delta(s, v)) \wedge v \neq s$ raggiungibile da s (altrimenti, $\delta(s, v) = \infty \geq v.d$)

Sia u il vertice che precede immediatamente v , allora: $\delta(s, v) = \delta(s, u) + 1 \rightsquigarrow u.d = \delta(s, u)$

$$\rightsquigarrow v.d > \delta(s, v) = \delta(s, u) + 1 \\ * \\ = u.d + 1$$

Consideriamo il momento in cui u e' eliminato da Q . In quel momento:

1) v e' bianco: la riga 15 imposta $v.d = u.d + 1$ che contraddice *

2) v e' marrone: allora era già stato rimosso dalla coda e per il COROLARIO $v.d \leq u.d$, che contraddice *

3) v e' grigio: allora era stato colorato di grigio dopo la cancellazione da Q di un vertice w (cancellato prima di u)
t.c. $v.d = w.d + 1$ ma $w.d \leq u.d$ (corollario) $\rightsquigarrow v.d \leq u.d + 1$

$$v.d = \delta(s, v) \quad \forall v \in V$$

2) Se $v.\pi = u \rightsquigarrow v.d = u.d + 1 \rightsquigarrow \delta(s, v) = \delta(s, u) + 1$

\rightsquigarrow cammino più corto da s a $u = v.\pi + \text{arco}(v.\pi, v)$
È
cammino più corto da s a v

*ALBERI BFS: da procedura BFS costruisce un albero BF mentre visita il grafo. L'albero è rappresentato dagli attributi π .

Per un grafo $G = (V, E)$ con sorgente s , definiamo il sottografo dei predecessori di G come $G_\pi = (V_\pi, E_\pi)$ dove:

$$V_\pi = \{v \in V : v.\pi \neq \text{NIL}\} \cup \{s\}$$

$$E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$

G_π è tale che: $-V_\pi$ contiene tutti i vertici raggiungibili da s .

- $\forall v \in V_\pi \exists!$ cammino da s a v in G_π che è il cammino più corto da s a v in G .

DIM: 1) $v.\pi = u \iff (u, v) \in E$ e $\delta(s, v) < \infty$, ovvero se v è raggiungibile da s

2) G_π è un albero (connesso e $|E_\pi| = |V_\pi| - 1$)

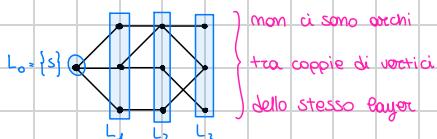
\rightsquigarrow unico cammino da s a $v \rightsquigarrow$ il cammino più corto

Test Grafi Bipartiti

LEMMA: Sono $G = (V, E)$ un grafo connesso, $s \in V$, e L_0, L_1, \dots, L_k i layer prodotti da BFS con input (G, s) . Esattamente una delle seguenti si verifica:

1) Non esiste alcun arco tra coppie di vertici in uno stesso layer, e G è bipartito

2) Esiste un arco tra una coppia di vertici in uno stesso layer, e G contiene un ciclo di lunghezza dispari (quindi G NON è bipartito)



DIM: 1) Un qualsiasi arco di G ha estremi in layer consecutivi. $A = \{v \in L_i : \text{com i pari}\}$ e $B = \{v \in L_i : \text{com i dispari}\}$

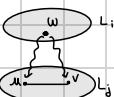
\rightsquigarrow no arco con entrambi estremi in A o in B

$\rightsquigarrow G$ è bipartito

2) Sia $(u, v) \in E$ con $u, v \in L_j$ per un certo j . Sia w l'antecesore comune di u e v nell'albero BFS con minima distanza.

$$d(w, u) = d(w, v) \dots$$

Supponiamo $w \in L_i$:



\rightsquigarrow ciclo di lunghezza $i + (j-i) + (j-i)$ dispari

$\rightsquigarrow G$ non è bipartito

COROLARIO: un grafo è bipartito se e solo se non contiene cicli di lunghezza dispari.

LEMMA: È possibile testare se un grafo G è bipartito ($\Leftrightarrow \chi(G) \leq 2$) in $O(m+n)$

$G = (V, E)$ connesso, $s \in V$ arbitrario. Lanciamo BFS con input (G, s) .

$$\forall (u, v) \in E \text{ verifichiamo se } u.d = v.d \quad \exists (u, v) \in E \text{ con } u.d = v.d \Rightarrow G \text{ non è bipartito}$$

$$\nexists (u, v) \in E \text{ con } u.d = v.d \Rightarrow G \text{ è bipartito}$$

Idea di un generico algoritmo di visita:

La visita parte da una sorgente s .

\rightsquigarrow Aggiunge s a Q , che contiene tutti i nodi da cui la visita può proseguire, ed esplora seguendo una qualche regola uno dei vertici adiacenti ad s .

\rightsquigarrow Al generico passo di visita, viene scelto (secondo una qualche regola) un nodo $u \in Q$, e si visita (secondo una qualche regola) un arco (u, v) di G : se il vertice v viene scoperto per la prima volta mediante tale arco, v viene marcato come scoperto, e viene inserito in Q ; inoltre, u viene marcato come padre di v .

\rightsquigarrow Un vertice v rimane in Q fintantoché non sono stati scoperti tutti i vertici adiacenti a v .

Due scelte naturali su come implementare Q , che influenzano le caratteristiche della visita ed i suoi scopi:

1. Q è implementata come una **coda (FIFO)** \rightsquigarrow **visita in ampiezza (BFS)**
2. Q è implementata come una **pila (LIFO)** \rightsquigarrow **visita in profondità (DFS)**

Visita in Profondità (DFS)

Dato un grafo $G = (V, E)$, DFS (depth-first search) visita G sempre più in profondità fintantoché esistono vertici non scoperti. Se non esistono vertici non scoperti, allora uno di essi viene selezionato come nuovo vertice sorgente e la visita riparte da esso.

Inoltre il sottografo dei predecessori prodotto può essere formato da più alberi ed è definito come: $G_{\text{pr}} = (V_{\text{pr}}, E_{\text{pr}})$, dove

$$E_{\text{pr}} = \{(v_{\text{pr}}, v) : v \in V \text{ e } v.\pi \neq \text{NIL}\}$$

Il sottografo dei predecessori di una DFS forma una **FORESTA DF** (depth-first forest) composta da vari **ALBERI DF** (depth-first tree), gli archi in E_{pr} sono detti **ARCHI D'ALBERO**.

A differenza di BFS, non trova distanze, ma può essere utilizzato per:

- 1 - ordinamento topologico
- 2 - componenti fortemente connesse
- 3 - ricerca di cicli

CHE FUNZIONA DFS:

- i vertici vengono colorati durante la visita: **BIANCO**, il vertice non è ancora stato visitato

garantisce che ogni vertice vada a finire in un solo albero DF, in modo che questi alberi Siano disgiunti

- ogni vertice v ha due informazioni temporali:
 sono numeri intesi compresi fra 1 e $|V|$, poiché ogni vertice viene scoperto una sola volta e la sua visita può essere completata una sola volta +

- v.d registra il momento in cui v viene scoperto (e colorato di grigio)
- v.f registra il momento in cui l'ha visitata completa e l'ispezione della lista di v (e colorato di nero).

v e' WHITE prima del tempo v.d, GRAY fra il tempo v.d e v.f, e BLACK successivamente

DFS (G)

1. for ogni vertice $u \in G.V$
2. $u.color = \text{WHITE}$
3. $u.F = \text{NIL}$
4. $\text{time} = 0$ //variabile globale
5. for ogni vertice $u \in G.V$
6. if $u.color == \text{WHITE}$
7. DFS-VISIT (G, u)

DFS-VISIT (G, u)

1. $\text{time} = \text{time} + 1$
2. $u.d = \text{time}$ //il vertice bianco u è stato appena scoperto
3. $u.color = \text{GRAY}$
4. for ogni $v \in G.\text{Adj}[u]$ //ispeziona l'arco (u, v)
5. if $v.color == \text{WHITE}$
6. $v.F = u$
7. DFS-VISIT (G, v)
8. $u.color = \text{BLACK}$ //colora di nero u , visita completata
9. $\text{time} = \text{time} + 1$
10. $u.F = \text{time}$

DFS: righe 5-7: controllano, uno alla volta, tutti i vertici in V e, quando trovano un vertice bianco, lo visitano con DFS-VISIT, automaticamente il vertice u passato come parametro diventa la radice di un nuovo albero della foresta DF.

Quando DFS termina o ogni vertice u è stato assegnato un TEMPO DI SCOPERTA $u.d$ e un TEMPO DI COMPLETAMENTO $u.F$

DFS-VISIT: righe 4-7: ispezionano ogni vertice v adiacente a u , se v è bianco viene visitato ricorsivamente.

$T(m)$ DFS: (righe 1-2) - (righe 5-7) impiegano $\Theta(v)$, escluso il tempo per chiamare DFS-VISIT

$T(m)$ DFS-VISIT: la procedura viene chiamata esattamente una volta per ogni vertice $v \in V$, in quanto viene invocata solo se il vertice è bianco.

(righe 4-7): viene eseguito $|\text{Adj}[v]|$ volte. Poiché $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$ il costo totale è $\Theta(E)$.

$T(m)$ totale: $\Theta(V+E)$

TEOREMA (ANNIDAMENTO DEGLI INTERVALLI DEI DISCENDENTI)

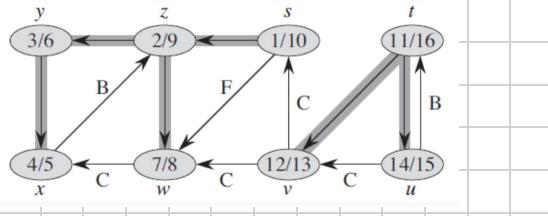
Sia G un grafo (diritto o non). Per qualsiasi visita in profondità di G vale la seguente proprietà: il vertice v è un discendente proprio di u nella foresta DFS se e solo se $u.d < v.d < v.f < u.F$

TEOREMA (TEOREMA DEL CAMMINO BIANCO)

In una foresta DFS di un grafo G (diritto o non), il vertice v è un discendente del vertice u se e solo se, al tempo $u.d$ in cui viene scoperto u , il vertice v può essere raggiunto da u lungo un cammino che è formato esclusivamente da vertici bianchi.

• Sia G_F la foresta DFS prodotta da una visita in profondità di G .

1. **ARCHI D'ALBERO:** archi nella foresta DF di G_F : l'arco (u, v) è un arco d'albero se v viene scoperto la prima volta durante l'esplorazione di (u, v) .
2. **ARCHI ALL'INDIETRO:** sono gli archi (u, v) che collegano un vertice u a un antenato v in un albero DF. I cappi, che possono presentarsi nei grafici orientati, sono considerati archi all'indietro.
3. **ARCHI IN AVANTI:** sono gli archi (u, v) , (diversi dagli archi d'albero) che collegano un vertice u a un discendente v in un albero DF.
4. **ARCHI TRASVERSALI:** tutti gli altri archi. Possono connettere i vertici nello stesso albero DF, purché un vertice non sia un antenato dell'altro, oppure possono connettere vertici di alberi DF differenti.

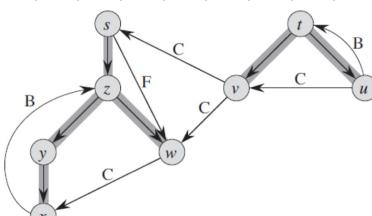


es. (s, z) arco d'albero

(s, u) -F arco in avanti

(x, z) -B arco all'indietro

(w, x) -C arco trasversale



Ogni arco (u, v) può essere classificato in base al colore del vertice v raggiunto quando (u, v) viene ispezionato per la prima volta: 1) **V bianco** \rightarrow (u, v) arco d'albero;

2) **V grigio** \rightarrow (u, v) arco all'indietro;

3) **V blu** \rightarrow (u, v) arco in avanti o trasversale.

TEOREMA: In una visita in profondità di un grafo non diretto G , gli archi di G possono essere archi d'albero o archi all'indietro.

APPLICAZIONE DFS:

• **ORDINAMENTO TOPOLOGICO:** Un ordinamento topologico di un dag (Directed Acyclic Graph) $G = (V, E)$ è un ordinamento lineare di V tale che, se $(u, v) \in E$, allora u appare prima di v nell'ordinamento. Se G contiene un ciclo \rightarrow \nexists ordinamento lineare.

TOPOLOGICAL-SORT (G)

1. call DFS(G) to compute finishing times $v.f$ for each vertex v
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

LEMMA: Un grafo diretto G è acilico se e solo se una visita in profondità di G non genera archi all'indietro.

TEOREMA: Sia G un dag. Topological-Sort (G) produce un ordinamento topologico di G .

Componenti fortemente connesse

Def. Un grafo diretto è **FORTEMENTE CONNESSO** se due vertici qualsiasi sono raggiungibili l'uno dall'altro. Le **COMPONENTI**

FORTEMENTE CONNESE sono le classi di equivalenza dei vertici secondo la relazione "sono mutualmente raggiungibili".

Equivalentemente, sono i sottografi fortemente connessi massimali.

LEMMA: Siano $G = (V, E)$ un grafo diretto ed $s \in V$ arbitrario. G è fortemente connesso se e solo se qualsiasi vertice è raggiungibile da s ed s è raggiungibile da qualsiasi vertice.

• ALGORITMO per determinare se $G = (V, E)$ è fortemente连通的:

1 - Scelgo vertice arbitrario $s \in V$.

2 - DFS-VISIT(s, s).

3 - DFS-VISIT(G^T, s) G^T sono gli archi di G con direzioni invertite

4 - Fortemente连通的 se e solo se tutti i vertici sono scoperti in entrambe le esecuzioni

$$\rightsquigarrow T(m) = \Theta(|V| + |E|)$$



STRONGLY-CONNECTED-COMPONENTS (G)

1. call DFS(G) to compute finishing times $u.f$ for each vertex u
2. compute G^T
3. call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4. output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

LEMMA: Siano C e C' due componenti fortemente连通的 distinte del grafo diretto G . Se $u, v \in C$ e $u', v' \in C'$ ed esiste un cammino $u \rightsquigarrow u'$ in G , allora non può esistere anche un cammino $v \rightsquigarrow v'$ in G .

LEMMA: Siano C e C' due componenti fortemente連通的 distinte del grafo diretto $G = (V, E)$. Supponiamo che esista un arco $(u, v) \in E^T$, dove $u \in C$ e $v \in C'$. Allora $f(C) < f(C')$

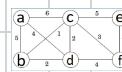
TEOREMA: da procedura STRONGLY-CONNECTED-COMPONENTS (G) calcola correttamente le componenti fortemente連通的 di un grafo diretto G .

Grafi Pesati

def. È un grafo $G = (V, E)$ in cui ad ogni arco viene associato un valore definito da una funzione peso w (tipicamente valori reali) $w: E \rightarrow \mathbb{R}$

Dato un arco $e = (u, v)$, ne denoteremo il peso con $w(e)$, oppure per semplicità, con $w(u, v)$.

Ad esempio, $w(a, c) = 6$



• **CAMMINI MINIMI IN GRAFI PESATI:** Sia $G = (V, E)$ un grafo diretto e pesato con funzione peso reale w . Il peso (o anche costo) di un cammino orientato $p = \langle v_0, v_1, v_2, \dots, v_k \rangle$ è dato da: $w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$.

Un **CAMMINO MINIMO** in G tra una coppia di vertici $u \neq v$ è un cammino orientato in G tra u e v avente costo MINOR o Uguale

a quello di ogni altro cammino $u \neq v$. Tale costo viene detto la **DISTANZA** in G tra u e v , e verrà denotato con $d(u, v)$

$(d(u, v) = \infty$ se u non è raggiungibile da v)

N.B. Il cammino minimo tra due nodi non è necessariamente unico.

• **TRE PROBLEMI FONDAMENTALI:** Dato un grafo G , i tre problemi classici legati ai cammini minimi sono i seguenti:

1) **Cammino minimo tra due nodi:** dati due nodi x e y in G , trovare un cammino minimo in G che congiunge x e y .

2) **Cammini minimi a sorgente singola:** dato un vertice s in G , detto **SORGENTE**, trovare i cammini minimi da s verso tutti i vertici da esso raggiungibili nel grafo G .

3) **Cammini minimi tra tutte le coppie di nodi:** trovare un cammino minimo in G che congiunge ogni coppia di vertici x e y di G .

LEMMA: Se un grafo G contiene un ciclo di **PESO NEGATIVO**, allora non esistono cammini più brevi.

• Un cammino minimo può contenere un ciclo? • ciclo di peso negativo: NO

• ciclo di peso positivo: NO

• **RAPPRESENTAZIONE DEI CAMMINI MINIMI:** Vogliamo non solo i pesi dei cammini minimi ma anche i loro vertici. Anche per algoritmi di cammini minimi rimetteremo gli attributi $x \rightsquigarrow$ sotto grafo dei predecessori G_x , contenente cammino minimo da sorgente s ad ogni vertice raggiungibile da s .

INITIALIZE-SINGLE-SOURCE(G, s)

1. for each vertex $v \in G.V$
2. $v.d = \infty$
3. $v.\pi = \text{NIL}$
4. $s.d = 0$

$v.d$ è limite superiore per $\delta(s, v)$ detto stima del cammino minimo.

RELAX(u, v, w)

1. if $v.d > u.d + w(u, v)$
2. $v.d = u.d + w(u, v)$
3. $v.\pi = u$

Verifico se passando per u ottengo cammino più corto da s a v .

TEOREMA: Un sottocammino di un cammino più breve è un cammino più breve.

CAMMINI MINIMI DA SORGENTE SINGOLA

Input: grafo diretto $G = (V, E)$ con funzione peso $w: E \rightarrow \mathbb{R}$ e vertice sorgente s .

Task: Trovare i cammini minimi da s a tutti i vertici raggiungibili da s .

Vedremo:

1) Algoritmo per GRAFI ACICLICI

2) Algoritmo di BELLMAN-FORD

3) Algoritmo di DISKRETA

tutti utilizzano INITIALIZE-SINGLE-SOURCE e RELAX come subroutine

LEMMA: (Disuguaglianza triangolare) Per qualsiasi arco $(u, v) \in E$, si ha $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

LEMMA: (Proprietà del limite superiore) Per tutti i vertici $v \in V$, si ha sempre $v.d \geq \delta(s, v)$ e, una volta che il limite superiore $v.d$ assume il valore $\delta(s, v)$, esso non cambia più.

LEMMA: (Proprietà dell'assenza di un cammino) Se non c'è un cammino da s a v , allora si ha sempre $v.d = \delta(s, v) = \infty$.

LEMMA: (Proprietà della convergenza) Se $s \leadsto u \rightarrow v$ è un cammino minimo da s a v e se $u.d = \delta(s, u)$ in un istante qualsiasi prima del rilassamento dell'arco (u, v) , allora $v.d = \delta(s, v)$ in tutti gli istanti successivi.

LEMMA: (Proprietà del rilassamento) Se $p = \langle v_0, v_1, \dots, v_k \rangle$ è un cammino minimo da $s = v_0$ a v_k e gli archi di p vengono rilassati nell'ordine $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$, allora $v_k.d = \delta(s, v_k)$ (indipendentemente dagli altri rilassamenti).

LEMMA: (Proprietà del sottografo dei predecessori) Una volta che $v.d = \delta(s, v)$ per ogni $v \in V$, il sottografo dei predecessori G_π è un albero di cammini minimi radicato in s .

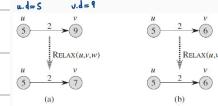
GRAFI ACICLICI

DAG-SHORTEST-PATHS(G, w, s)

1. topologically sort the vertices of G $\Theta(|V| + |E|)$
2. INITIALIZE-SINGLE-SOURCE(G, s) $\Theta(|V|)$
3. for each vertex u , taken in topologically sorted order: $\Theta(|E|)$
4. for each vertex $v \in G.\text{Adj}[u]$
5. RELAX(u, v, w)

$T(m) + \Theta(|V| + |E|)$

TEOREMA: Se $G = (V, E)$ è un dag con funzione peso $w: E \rightarrow \mathbb{R}$ e sorgente s , allora al termine di DAG-SHORTEST-PATHS, $v.d = \delta(s, v)$ per ogni $v \in V$ e G_π è un albero di cammini minimi.



Algoritmo Bellman-Ford

L'algoritmo restituisce un valore booleano che indica se esiste o meno un ciclo di peso negativo raggiungibile da s.

- 1) Se esiste, il problema non ha soluzione
- 2) Se non esiste, l'algoritmo trova cammini minimi con rispettivi pesi

BELLMAN-FORD (G, w, s)

```

1. INITIALIZE-SINGLE-SOURCE( $G, s$ )  $\Theta(|V|)$ 
2. for  $i = 1$  to  $|G.V|-1$  {
3.   for each edge  $(u, v) \in G.E$  }  $\Theta((|V|-1) \cdot |E|)$ 
4.   RELAX  $(u, v, w)$ 
5.   for each edge  $(u, v) \in G.E$  }  $\Theta(|E|)$ 
6.   if  $v.d > u.d + w(u, v)$ 
7.     return false
8. return true

```

$T(m) = \Theta(|V| \cdot |E|)$

riga 1: inizializzazione dei valori d e π di tutti i vertici

righe 2-4: viene effettuato un rilassamento per ciascun arco del grafo

righe 5-8: controllano se esiste un ciclo di peso negativo e restituiscono il valore booleano appropriato

TEOREMA: Supponiamo che $G = (V, E)$ non contenga cicli di peso negativo raggiungibili da s. Allora, dopo $|V|-1$ iterazioni del ciclo for (righe 2-4), si ha $v.d = \delta(s, v)$ per ogni $v \in V$ raggiungibile da s. BELLMAN-FORD restituisce true e G_{π} è un albero di cammini minimi radicato in s.

TEOREMA: Supponiamo che $G = (V, E)$ contenga un ciclo di peso negativo raggiungibile da s. Allora, BELLMAN-FORD restituisce false.

Algoritmo di Dijkstra

Risolve il problema dei cammini minimi da sorgente unica in un grafo pesato $G = (V, E)$ nel caso in cui tutti i pesi degli archi non sono negativi.

L'algoritmo mantiene un insieme S di vertici i cui pesi finali dei cammini minimi dalla sorgente s sono stati già determinati. L'algoritmo ripetutamente seleziona ripetutamente il vertice $u \in V - S$ con la stima minima del cammino minimo, aggiunge u ad S e rilassa tutti gli archi che escono da u .

DJIKSTRA (G, w, s)

```

1. INITIALIZE-SINGLE-SOURCE( $G, s$ ) // inizializza i valori d e  $\pi$ 
2.  $S = \emptyset$  // inizializza S come insieme vuoto
3.  $Q = G.V$  // inizializza la coda di min-priorità Q in modo che contenga tutti i vertici in  $V$ 
4. while  $Q \neq \emptyset$  // inizialmente  $Q = V - S$ 
5.    $u = \text{EXTRACT-MIN}(Q)$  } um vertice  $u$  viene estratto da  $Q = V - S$  e aggiunto all'insieme  $S$ . Il vertice  $u$ , quindi ha la stima minima del cammino minimo di tutti i vertici in  $V - S$ .
6.    $S = S \cup \{u\}$  } rilassano ogni arco  $(u, v)$  che esce da  $u$  e aggiornano la stima  $v.d$  e il predecessore  $v.\pi$ . Se il cammino minimo che arriva a  $v$  può essere migliorato passando per  $u$ 
7.   for ogni vertice  $v \in G.\text{Adj}[u]$  } rilassano ogni arco  $(u, v)$  che esce da  $u$  e aggiornano la stima  $v.d$  e il predecessore  $v.\pi$ . Se il cammino
8.     RELAX  $(u, v, w)$ 

```

• CONFRONTO FRA BELLMAN-FORD & DIJKSTRA

Bellman-Ford si applica ad una classe molto più vasta di grafi diretti. Alcune implementazioni di Dijkstra sono sempre più efficienti di Bellman-Ford. Per esempio, con Fibonacci heap $\sim m + m \log m = O(mm)$. Dijkstra è, al momento, l'algoritmo più efficiente per la classe di grafi a cui si applica. Poiché abbiamo lower bound $\Omega(m+m)$, non è in generale ottimo. Ma è ottimo se $m = \Omega(m \log m)$.

Operazioni con gli Insiemi Disgiunti

def. Una struttura dati per insiemi disgiunti mantiene una collezione $\mathcal{S} = \{S_1, S_2, \dots, S_k\}$ di insiemi dinamici disgiunti. Ciascun insieme è identificato da un rappresentante, che è un elemento dell'insieme.

Ogni elemento è rappresentato da un oggetto. Indicando con x un oggetto, vogliamo supportare le seguenti operazioni:

- **MAKE-SET(x)**: crea un nuovo insieme il cui unico elemento (e rappresentante) è x . Poiché gli insiemi sono disgiunti, x non può trovarsi in qualche altro insieme.
- **UNION(x, y)**: unisce gli insiemi dinamici che contengono x e y in un nuovo insieme che è l'unione di questi due insiemi. Si suppone che i due insiemi siano disgiunti prima dell'operazione. Poiché richiediamo che gli insiemi nella collezione siano disgiunti, "distruggiamo" gli insiemi S_x e S_y eliminandoli dalla collezione.
- **FIND-SET(x)**: restituisce un puntatore al rappresentante dell'insieme (unico) che contiene x .

(NOTAZIONE): M , numero di operazioni MAKE-SET

oss. • # operazioni UNION $\leq M-1$

M' , numero di operazioni MAKE-SET, UNION, FIND-SET.

• $M' \geq M$

• APPLICAZIONE: COMPONENTI CONNESSE DI UN GRAFO

CONNECTED-COMPONENTS(G)

1. For ogni vertice $v \in G.V$ } il pone ciascun vertice v nel proprio insieme
2. MAKE-SET(v) }
3. For ogni arco $(u, v) \in G.E$ }
4. if FIND-SET(u) ≠ FIND-SET(v) } per ogni arco (u, v) unisce gli insiemi che contengono u e v
5. UNION(u, v) }

SAME-COMPONENT(u, v)

1. if FIND-SET(u) == FIND-SET(v)
2. return true
3. else return false

• RAPPRESENTAZIONE DI INSIEMI DISGIUNTI TRAMITE LISTE CONCATENATE

Ciascun insieme è rappresentato dalla sua lista concatenata. L'oggetto di ciascun insieme ha gli attributi `head` (punta al primo oggetto della lista) e `tail` (che punta all'ultimo oggetto). Ogni oggetto nella lista contiene un elemento dell'insieme, un puntatore al successivo oggetto e un puntatore che ritorna all'oggetto dell'insieme. Il rappresentante è l'elemento dell'insieme nel primo oggetto della lista.

MAKE-SET e FIND-SET richiedono un tempo $O(1)$ con l'utilizzo delle liste concatenate.

↳ **MAKE-SET**, creiamo una nuova lista concatenata il cui unico oggetto è x

↳ **FIND-SET**, seguiamo il puntatore da x per arrivare all'oggetto del suo insieme e poi ritornare all'elemento nell'oggetto cui punta `head`.

TEOREMA: Utilizzando la rappresentazione degli insiemi disgiunti tramite liste concorrenti e l'euristica dell'unione pesata, una sequenza di m operazioni, m delle quali sono operazioni MAKE-SET, impiegheranno un tempo $O(m + m \log m)$.

* FORESTA DI INSIEMI DISGIUNTI

Rappresentiamo gli insiemi con alberi radicati, dove ogni nodo contiene un elemento e ogni albero rappresenta un insieme. In una **FORESTA DI INSIEMI DISGIUNTI**, ogni elemento punta soltanto a suo padre. In modo radice di ogni albero contiene il rappresentante ed è padre di se stesso.

↳ **MAKE-SET**, crea un albero con un solo nodo. ↳ **FIND-SET**, segue i puntatori ai padri finché non trova la radice dell'albero.

↳ **UNION**, fa sì che la radice di un albero punti alla radice dell'altro albero.

EURISTICHE:

1. **UNIONE PER RANGO**, l'idea è di fare in modo che la radice dell'albero con meno nodi punti alla radice dell'albero con più nodi.

Per ogni nodo manteniamo un **RANGO**, ovvero un limite superiore per l'altezza del nodo da radice con il ramo più piccolo viene fatto puntare alla radice con il ramo più grande durante l'operazione **UNION**.

2. **COMPRESSESIONE DEL CAMMINO**, utilizzata durante le operazioni di **FIND-SET** per fare in modo che ciascun nodo nel cammino di ricerca punti direttamente alla radice. La compressione del cammino non cambia i ranghi.

FIND-SET(x)

1. if $x \neq x.p$
2. $x.p = \text{FIND-SET}(x.p)$
3. return $x.p$

TEOREMA: Utilizzando la rappresentazione degli insiemi disgiunti tramite foresta di insiemi disgiunti e le euristiche dell'unione per range e della compressione del cammino, una sequenza di m operazioni, m delle quali sono operazioni **MAKE-SET**, impiega un tempo $O(m \cdot d(m))$, dove $d(m)$ è una funzione che cresce molto lentamente ($d(m) \leq 4$ per $m \leq 10^{60}$).

Alberi di Connessione Minimi

Input: $G = (V, E)$ non diretto e connesso con funzione peso $w: E \rightarrow \mathbb{R}$

Task: trovare $T \subseteq E$ aciclico che collega tutti i vertici e con peso totale minimo

T indica un sottografo
connesso

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

Oss. G è connesso se e solo se ammette un albero di connessione

TEOREMA: (formula di Cayley) Il grafo completo con m vertici K_m ha m^{m-2} alberi di connessione distinti

Sia $A \subseteq E$ (sottoinsieme degli archi), vogliamo garantire la seguente invarianta: Prima di ogni iterazione, A è un sottoinsieme di qualche albero di connessione minima.

→ Ad ogni iterazione viene scelto un arco (u, v) e aggiunto ad A se e solo se (u, v) è sicuro per A . Ovvero, se $A \cup \{(u, v)\}$ è ancora un sottoinsieme di qualche albero di connessione minima.

Degli archi taglio $(S, V-S)$ una partizione di V . Un arco (u, v) attraversa il taglio. Il taglio è una partizione dei vertici di G in due insiemi disgiunti. Un arco che attraversa un taglio è detto **LEGGERO** se tra tutti gli archi che attraversano il taglio esso è quello di peso minimo.

GENERIC-MST(G, w)

1. $A = \emptyset$
2. while A non forma un albero di connessione
3. trova un arco (u, v) che è sicuro per A
4. $A = A \cup \{(u, v)\}$
5. return A

TEOREMA: Sia $G = (V, E)$ un grafo connesso non orientato con funzione peso $w: E \rightarrow \mathbb{R}$. Sia A un sottoinsieme di E che è contenuto in qualche albero di connessione minima per G . Sia $(S, S-V)$ un taglio qualsiasi di G che rispetta A . Sia (u, v) un arco leggero che attraversa $(S, S-V)$. Allora, l'arco (u, v) è sicuro per A .

Algoritmo di Kruskal

d'algoritmo di Kruskal trova un arco sicuro da aggiungere alla foresta in costruzione saggiando, fra tutti gli archi che collegano due alberi qualsiasi nella foresta, un arco (u, v) di peso minimo.

MST-KRUSKAL(G, w)

```

1  $A = \emptyset$ 
2 for ogni vertice  $v \in G.V$ 
3   MAKE-SET( $v$ )
4 ordina gli archi di  $G.E$  in senso non decrescente rispetto al peso  $w$ 
5 for ogni arco  $(u, v) \in G.E$ , preso in ordine di peso non decrescente
6   if FIND-SET( $u$ ) ≠ FIND-SET( $v$ )
7      $A = A \cup \{(u, v)\}$ 
8     UNION( $u, v$ )
9 return  $A$ 
```

righe 1-3: inizializziamo l'insieme A come un insieme vuoto e creamo $|V|$ alberi, uno per ogni vertice

righe 5-8: esaminiamo gli archi nell'ordine dal più leggero al più pesante. Il ciclo verifica, per ogni arco (u, v) , se l'estremo u e v appartengono allo stesso albero; in caso affermativo, l'arco (u, v) non può essere aggiunto alla foresta senza generare un ciclo, quindi l'arco viene scartato. Altrimenti i due vertici appartengono ad alberi differenti. In questo caso, l'arco (u, v) viene aggiunto ad A nella riga 7 e i vertici dei due alberi vengono fusi nella riga 8.

$$T(m) = O(|V| \cdot \log |E|)$$

Algoritmo di Prim

d'algoritmo di Prim ha la proprietà che gli archi nell'insieme A formano sempre un albero singolo. d'albero inizia da un arbitrario vertice radice r e si sviluppa fino a coprire tutti i vertici in V . A ogni passo viene aggiunto all'albero A un arco leggero che collega A con un vertice isolato (che non sia estremo di qualche arco in A).

- input: G (grafo connesso) e r (radice dell'albero di connessione minima da costruire)
- i vertici che non si trovano nell'albero risiedono in una coda di min-priorità Q basata su un campo key.
- $v.key$ è il peso minimo di un arco qualsiasi che collega v a un vertice nell'albero
- $v.\pi$ padre di v nell'albero

MST - Prim(G, ω, r)

```

1 for each  $u \in G.V$ :
2    $u.key = \infty$ 
3    $u.\pi = NIL$ 
4    $r.key = 0$ 
5    $Q = G.V$ 
6 while  $Q \neq \emptyset$ :
7    $u = EXTRACT - MIN(Q)$ 
8   for each  $v \in G.Adj[u]$ :
9     if  $v \in Q$  and  $\omega(u, v) < v.key$ :
10        $v.\pi = u$ 
11        $v.key = \omega(u, v)$ 
```

