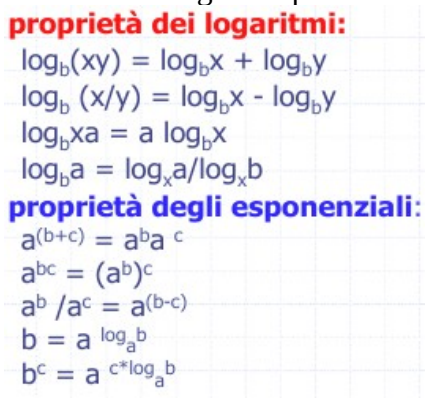
Appunti Algoritmi

Rircorsione di coda: la chiamata ricorsiva è l'ultima istruzione da eseguire
Ricorsione binaria: due chiamate ricorsive per ciascun caso non base
Ricorsione multipla: potenzialmente molteplici chiamate ricorsive per ciascun caso non base

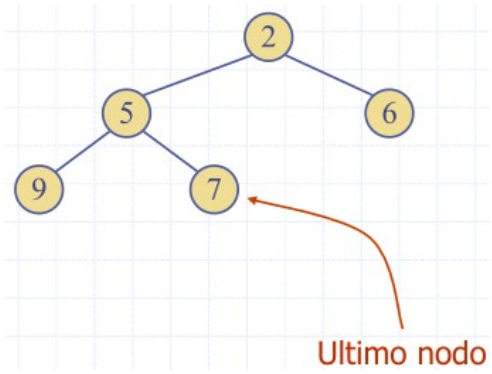
Analisi degli algoritmi

Il tempo di esecuzione di un algoritmo cresce al crescere della dimensione dell'input
La notazione O grande permette di esprimere un **upper bound** al tasso di crescrita di una funzione



Lezioni 12-13 marzo (video online)
Coda di priorità: lista di entry (chiave,valore)
Selection Sort: Usa coda di prorità non ordinata, quindi insert o(1) costante e removemin o(n) poichè devo scandire tutta la coda per trovare il min
Per n operazioni ho un costo quadratico dato da n * o(n) = $o(n^2)$ **NON in place**
Algoritmo: Data una sequenza S, inizio rimuovendo gli elementi dalla sequenza e li aggiungo alla coda di priorità (senza ordinarli), poi procedo ad ordinare gli elementi nella coda di priorità rimuovendo il minimo e inserendolo alla fine della sequenza in output.
In place: lo spazio di memoria non cresce in base all'input, ma è costante
Insertion Sort: Usa coda di prorità ordinata, quindi insert o(n) lineare, devo scandire tutta la coda per trovare l'elemento, mentre removeMin mi costa o(1),
Costo totale quadratico dato da n * o(n) = $o(n^2)$, **NON in place**
Algoritmo: Prelevo un elemento dalla sequenza e lo aggiungo alla coda di priorità (ordinandoli durante l'insert), poi rimuovo il minimo e lo aggiungo alla sequenza finale.
Insertion sort in place: invece di affidarci a una coda di priorità aggiuntiva, applichiamo degli swaps tra gli elementi
Insertion Sort migliore del Selection Sort perchè sa sfruttare meglio gli eventuali preordinamenti degli input.

Heap



Un **Heap** è un albero binario che memorizza chiavi sui nodi e può essere di tipo min-Heap o max-Heap (a seconda dell'ordinamento dei nodi) e soddisfa determinate proprietà: devono soddisfare l'heap order e devono essere degli alberi quasi completi.
Heap order: la chiave di ogni figlio deve essere maggiore o uguale della chiave del padre (Min-Heap, il contrario nel caso di un max-Heap)
Albero quasi completo: albero completo tranne per l'ultimo livello, che però lo deve essere da sinistra verso destra.
Inserimento in un heap: trovo l'ultimo nodo, inserisco la chiave k nel nodo e poi riprisito l'heap order. **Costo O(log n)**
Rimozione da un min-Heap: rimuove la radice e al suo posto ci pone la chiave presente nell'ultimo nodo e infine applica il **Downheap (inverso del UpHeap)** per ripristinare l'ordine corretto. **Costo O(log n)**
Upheap: algoritmo che consente di ripristinare l'heap order in tempo **O(log n)** (poichè tale è l'altezza di un heap), scambiando la nuova chiave k lungo il cammino ascendente dal nodo inserito finchè non viene ripristinato l'ordine o raggiunge la radice.
Negli scambi si predilige il nodo che ha la chiave minore.

Un albero completo ha un altezza minima di 2^k**,** ed ha un numero di nodi pari a $\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$

Da cui dato n numero di nodi: $k = \ln_2(n+1) - 1$

Dato che ci interessa il livello asintotico, riassumiamo dicendo che l'altezza di un albero di n nodi è al massimo n e al minimo $\ln_2(n)$, dato che la base del logaritmo non ha nessuna rilevanza asintotica, si fa direttamente il logaritmo normale in base 10.
Metodi size, isEmpty e min in tempo O(1)

Heap-Sort

Heap-Sort: coda di priorità basata sugli heap, ordinamento in tempo O(n log n), ordinamento in O(log n) invocato n-1 volte
Fusione di due heaps: data una chiave k e due heap, pongo la chiave k come nodo radice e pongo gli heap come figli del nodo k, e poi opero un downheap per ripristinare l'ordine.
rank = indice +1
Implementazione dell'heap con array: dato un nodo all'indice i, il figlio sinistro si trova nella posizione 2i+1 , e il figlio destro si trova a 2i + 2.
Per passare dal figlio al padre invece sottraggo 1 e divido per 2; ed escludendo l'eventuale parte decimale ottengo come parte intera l'indice del padre
Nell'allocazione dell'array per l'heap, inizialmente alloco un piccolo harray non troppo grande, di dimensione n, nel caso si riempisse, raddoppio la dimensione dell'array, (da n passo a 2n, da 2n passo a 4n, da 4n passo a 8n ecc.)
In questo modo ad ogni riempimento ritardo il tempo per una nuova allocazione.

Costruzione bottom-up di un Heap.

Procedo inserendo i nodi dalle foglie fino alla radice.
Inizialmente inserisco il livello delle foglie, poi quando passo a inserire il livello successivo procedo a ripristinare l'heap order, e cosi fino ad arrivare alla radice
Costo: O(n) poiché ogni nodo è attraversato al più da 2 cammini (velocizza la prima parte di heap-sort)

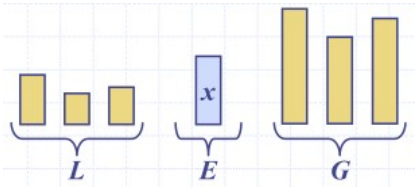
Array2heap: algoritmo per ricostruire l'heap in tempo O(n) utilizzando heap-sort
Si costruisce un heap in base all'array fornitomi da ordinare e si ordinano gli elementi al suo interno come se fosse un max heap, poi iterativamente ed in place, sostituisco il valore massimo che si trova in prima posizione, con l'elemento in posizione n-1 e rimuovo il valore massimo.
Ad ogni iterazione riordino il mio albero in modo che si mantenga l'heap order.
Ogni nodo appartiene al massimo a 2 percorsi, qualsiasi sia la lunghezza del percorso, la sommatoria della lunghezza dei percorsi è n.

Merge-Sort

Algortimo di tipo divide et impera, divide ricorsivamente l'input in 2 sottosequenze, risolve il sottoproblema e poi combina le soluzioni.
Come l'heap sort viene eseguito in tempo O(n log n), log n consiste nella discesa verso il basso a ogni suddivisione, dunque all'altezza dell'albero, mentre n per ordinare le sottosequenze, ma a differenza dell'heap sort non usa una coda di priorità ausiliaria ma si svolge tutto in place.
"**Merge sort ottimizzato non migliora la progressione**" è di tipo **iterativo**, è **efficente nella gestione delle porzioni rimanenti**, **evita continue operazioni di malloc / free** (per questo alloco una sola volta un array temp di taglia n pari alla dimensione dell'input).
Merge sort "in place" non ci riusciamo**.**
Costo spazio del merge sort: input + spazio temp (2n + costante); **spazio lineare**
Nel caso della **ricorsione**, ho uno spazio aggiuntivo dato dalla pila dei record di attivazione pari a: log n spazio(pila)

Quick-Sort (possibile in place)

Anch'esso basato sul divide et impera, si sceglie ricorsivamente un pivot randomicamente(ma secondo un algoritmo), e si partiziona in tre sequenze, una sequenza L contentene tutti gli elementi minori del pivot, una sequenza E contentente tutti gli elementi uguali al pivot e una sequenza G contentente tutti gli elementi maggiori del pivot e li riordina.



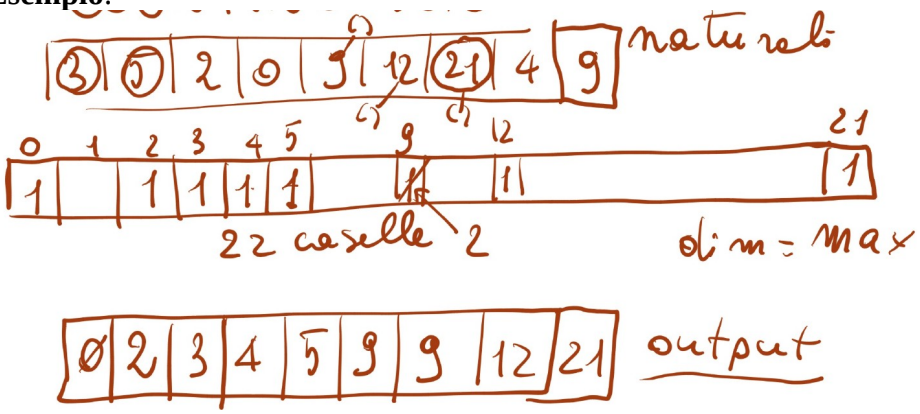
Il **caso peggiore** è quando si sbaglia a scegliere il pivot, quindi ottengo n ricorsioni che mi porta ad avere un **costo quadratico O(n^2)**

Per non scegliere un pivot completamente a random, immaginiamo di dividere l'array di dimensione s in 3 porzioni (ipotetiche come in figura)
Abbiamo una **good call** quando le sottosequenze di sinistra e di destra (L e G) sono minori di ¾ s, per ottenere ciò dobbiamo prendere un buon pivot all'interno **della** zona gialla, abbiamo invece una

bad call quando scegliamo un pivot tra le due zone esterne il che porta ad avere due sottosequenze di dimensione più grande di ¾ s.
Nel caso di una buona chiamata (abbiamo un miglioramento) ho una suddivisione del tipo: ¾ **n** e **n/4** (la meno buona), nel caso di una cattiva chiamata (non abbiamo un miglioramento) del tipo: **n** e **0** (non miglioro la progressione)
Essendo le scelte possibili due, o bad call o good call, la probabilità che esca una good call è pari al 50%, quindi ogni due chiamate riduco l'input di ¼.
Dato dunque un nodo di profondità i, ottengo un fattore di miglioramento ogni due livelli (solo i/2 antenati sono delle buone chiamate), dunque la dimensione della sequenza in input per la chiamata attuale è pari almeno a $(3/4)^{i/2}n$ (dimensione dell'array più grande)
Sia che vada bene, sia che vada male, ottengo su ogni livello complessivamente un lavoro pari a O(n), per un altezza pari a O(log n), dunque arrivo ad un **valore atteso del costo** pari a **O(n log n)**
In place (teoria): prendo un pivot e lo dispongo nella probabile posizione dove dovrà andare e dispongo a sinistra gli elementi minori del pivot e a destra gli elementi maggiori del pivot entrambi disordinati.
In place (pratica): prendo un pivot e lo scambio con l'elemento in prima posizione e prendo due indici i e j, i lo faccio puntare all'inizio della sequenza non ordinata e j alla fine e inizio a fare confronti, se l'elemento in posizione i-esima è minore dell'elemento in posizione j-esima, lascio tutto invariato ed incremento di un'unità i e decremento altrettanto j e procedo con i confronti.
Nel caso l'elemento in posizione i-esima sia maggiore dell'elemento in posizione j-esima, li scambio.
Procedo finchè i e j, si incontrano nella stessa casella o in due caselle adiacenti, nel caso si incontrassero nella stessa casella, confronto l'elemento in quella posizione con il pivot, nel caso sia minore, lo scambio con il pivot, invece nel caso si incontrassero su due caselle adiacenti si prende l'indice più a destra della parte sinistra e poi ricomincio fino a ordinare tutta la sequenza.
L'unica cosa non in place è il record della pila di attivazione
In place si suddivide in 2 parti e non in 3
Quick sort preferito su tutti gli altri

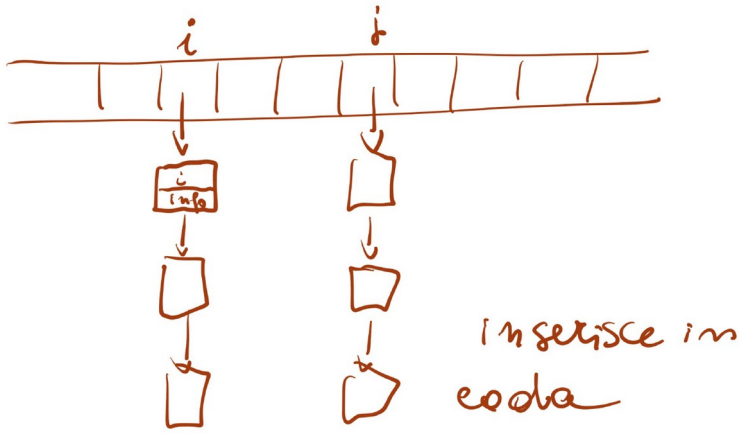
Counting Sort (si limita a lavorare con numeri interi)

Dato un array di numeri natuali, scansiono l'input per vedere qual'è il valore massimo e alloco un array temporaneo di dimensione pari al valore massimo più uno e procedo a riscansionare l'array e per ogni numero incremento la casella corrispondente come indice al valore che sto considerando.
Una volta terminato l'array di input, scansiono l'array temporaneo e scrivo in output tanti valori quanti sono i numeri presenti nelle caselle.
Esempio:

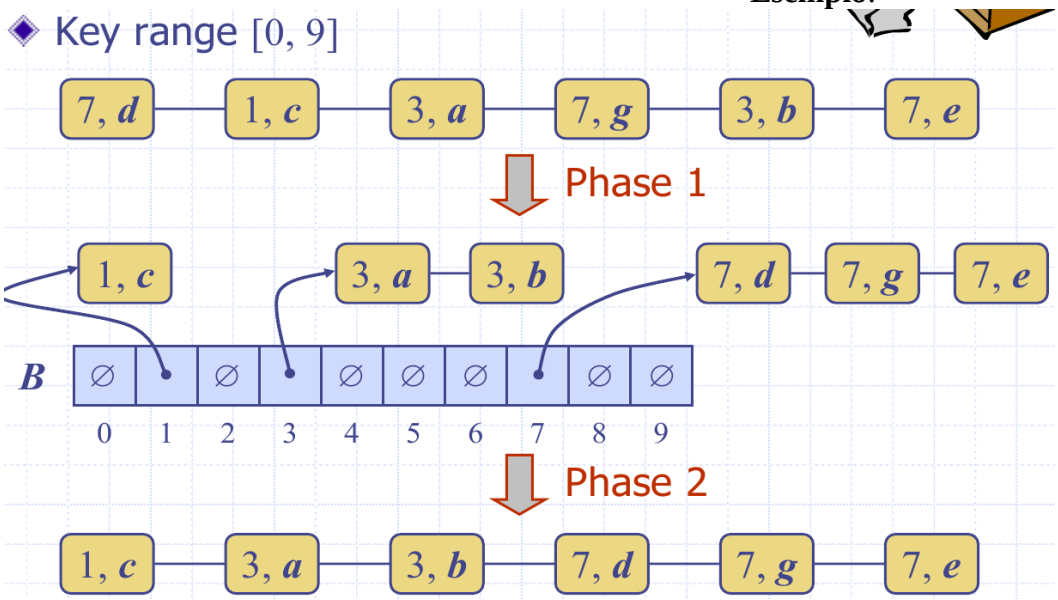


Data la dipendenza del valore massimo il costo dell'algortimo è pari a **O(n + N)** dove N non è in funzione dell'input ma potrebbe essere esponenziale. Nel momemto in cui il valore massimo è limitato, è limitata anche la dimensione dell'array temporaneo e posso considerare il costo lineare O(n)

Bucket Sort (generalizzazione del counting sort, lavora con entry < K,V >)
Possiamo immaginarlo come un array di liste.
Lavora con sequenze di elementi con chiavi nel range [0, N-1]
Procedendo come il counting sort sulla sequenza S fornitami, al posto di incrementare la casella corrispondente, inserisco in coda alla casella i-esima la sequenza corrispondente.
Alla fine scandisco ciò che ho ottenuto e riscrivo l'array dall'inizio e inserisco ogni elemento alla fine di S (per garantire la **proprietà di stabilità: l'ordine di elementi con la stessa chiave viene preservato dopo l'esecuzione dell'algoritmo)**.
Esempio:



Esempio:



Nel caso io abbia valori relativi (minimo m e massimo M) posso ottimizzare l'algoritmo creando un array temporaneo di dimensione M – m +1 per ridurre lo spazio allocato inutilmente dove nel caso volessi prendere una chiave di indice k la dovrei porre nella casella di indice k – m.

Ordine lessicografico

Utile per chiavi multi-dimensionali formati da d-tuple di chiavi.
Confronto tra due tuple x e y: confronto il primo elemento di entrambi, se x primo batte y primo allora la tupla x viene prima della y (e viceversa), se sono uguali continuo con i prossimi elementi delle tuple.
Sorting lessicografico: Data una sequenza di d-tuple, inizio i confronti partendo dall'ultimo elemento delle tuple (mantenendo la stabilità).
Esempio:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

Costo: Dato che per ogni elemento della tupla viene eseguito un algoritmo di stableSorting, l'algortimo gira in tempo **O(dT(n))** dove T(n) è il tempo che impiega l'algortimo di sorting

Radix-Sort

Radix Sort é una specializzazione del sorting lessicografico in cui come algoritmo di stable sorting viene usato il **bucket-sort** in cui dunque la chiavi sono interi nel range [0, N-1]
Costo: **O(d(n + N))** con d numero di componenti della sequenza. (costo esponenziale se non viene limitato)

Radix-Sort per numeri binari (Binary numbers)

Considera una sequenza di n interi di b-bit ed essendo ogni elemento nel range [0,1] dunque limitato e per la precisione abbiamo array di due caselle, possiamo far girare il radix sort in tempo **O(bn)**

Mappe

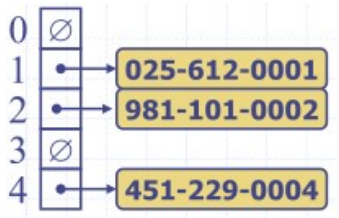
Una mappa modella una collezione in cui si possono cercare entry chiave-valore.
NON sono ammesse entry diverse con la stessa chiave (**nei dizionari si)**
Operazione sulle mappe:
get(k): se la mappa M ha un entry con la chiave k, ritorna il valore associato; altrimenti ritorna null
put(k,v): inserisci l'entry (k,v) nella mappa M; se la chiave k non è già in M, ritorna null; altrimenti ritorna il vecchio valore associato alla chiave k.
remove(k): se la mappa ha un entry con la chiave k, rimuovila e ritorna il valore associato ad essa; altrimenti ritorna null.
Size(), isEmpty().
entrySet(): ritorna una collezione iterabile di entry presenti in M
keySet(): ritorna una collezione iterabile di chiavi presenti in M
values(): ritorna un iteratore di valori presenti in M
Possiamo implementare una mappa usando una **lista non ordinata,** dunque una put viene eseguita in tempo O(1), mentre get e remove in tempo O(n)

Dizionari

uguali alle mappe ma:
get(k): restituisce una qualsiasi entry contentente la chiave k.
remove(k): rimuove una qualsiasi entry contente la chiave k.
remove(k,v): rimuove la chiave che presenta il valore v.

findAll(): restituisce tutte le entry con la chiave k

Tavole hash (per eseguire tutto in tempo sub-lineare, ma necessitano manutenzione)



Struttura ad array di puntatori in cui alcuni sono null e altri puntano alle informazioni.
Una funzione hash assegna a ogni chiave un numero molto preciso nel range [0, N-1] chiamato **pseudo-chiave** per poter fare (in ipotesi) un indirizzamento diretto, con N numero di chiavi e non valore massimo.
Il problema è che le tabbelle hash sono affette da **problemi di collisione** poichè lo spazio delle chiavi non ha una funzione iniettiva. (**principio del picione**: se tu hai n casette per i piccioni e ogni casetta può ospitare un piccione, se tu vuoi ospiare più di N piccioni, avrai casette in cui cercherai di mettere più di un piccione)
Paradosso del compleanno: su 23 persone ho una probabilità maggiore del 50% che almeno due abbiano lo stesso compleanno. Bound del compleanno uguale circa a $\sqrt{N}$
Occorre dunque una funzione hash che sparpagli il più possibile gli elementi sulla tabbella e avere una buona tecnica di gestione delle collisioni, oltre a decidere quanto fare grande la tabella inizialmente.

Funzioni di Hash

Composta da due funzioni: una **funzione di hash (hash code)** che trasformi la chiave in un intero e una funzione di compressione che trasformi l'intero ottenuto dalla funzione di hash in un altro numero nel range [0, N-1]
Il compito delle funzioni di hash è quello di sparpagliare le chiave apparentemente in maniera randomica.

Hash Codes

Memory address: Possiamo interpretare gli indirizzi di memoria di un oggetto chiave come un intero, ciò non va bene per stringhe e numeri.
Integer Cast: Prendo i bit della chiave e li rappresento come un intero (utile per chiavi di lunghezza minore o uguali al numero di bit di un intero (come byte,short, int e float)
Component sum: Paritizio i bit di una chiave in componenti di lunghezza fissa (16 o 32 bit) e li sommo (ignorando gli overflow). Utile per chiavi numeriche più lunghe (long, double)
Accumulazione polinomiale (usato per le stringe): Partiziono i bit di una chiave in una sequenza di componenti di lunghezza fissa (8,16 o 32 bit) e ne calcolo il polinomio associato in un determinato valore z, ignorando gli overflow.

$$p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + \ldots + a_{n-1} z^{n-1}$$

Il polinomio può essere calcolato in tempo O(n), ciò si può notare raccogliendo iterativamente z all'interno del polinomio e risolvendolo con la **regola di Horner (utile per calcolare il valore di un polinomio in un punto in tempo lineare**)
Horner: un ciclo per tutti i valori di i = k (scendendo) dove si sommano al valore a[i] la somma dei precedenti a[i] valori moltiplicati per z
(Esempio: con una scelta di z = 33 otteniamo al massimo 6 collisioni su un set di 50k parole inglesi)

$$P(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_k x^k =$$
$$= a_0 + x\left(a_1 + a_2 x + \cdots + a_k x^{k-1}\right) =$$
$$= a_0 + x\left(a_1 + x\left(a_2 + a_3 x + \cdots + a_k x^{k-2}\right)\right) = \cdots =$$
$$= a_0 + x\left(a_1 + x\left(a_2 + x\left(a_3 \cdots + x\left(a_{k-1} + a_k x\right)\right)\right)\right)$$

Nell'ambito **hash il costo invece di O(k)** (pari al numero di coefficienti del polinomio) **diventa O(1)** poichè nel mondo delle mappe l'input non è la chiave, ma è dato da tutti gli elementi della mappa, indipendentmenete dal numero di elementi che sta dentro la mappa ho una dimensione della chiave che non corrisponde ad un tipo ben definito e dunque il grado del polinomio non cambia al crescere della dimensione dell'input.

Funziondi di Compressione

Divisione: y mod N (resto della divisione intera y % N), N che è la dimensione della tabella di hash viene scelto di solito in modo che sia un numero primo.
Moltiplica, aggiungi e dividi (MAD) funzione pazza: **h(y) = (ay + b) mod N**
Sfrutto due coefficienti a e b interi non negativi tali che a mod N sia diverso da zero (**a non deve essere multiplo di N altrimenti ay mod N sarebbe zero avendo per tutte le chiavi la stessa casella hash**)

Gestione delle collisioni

Due approcci differenti:
- **1° approccio**: cerco di mettere la nuova entry in una nuova posizione
Open addressing: mi dice in quale altra posizione della tavola posizionare l'elemento collidente (non usa una memoria ulteriore come il separate chaining) **sfrutto linear probing** (vedi sotto)

- **2° approccio**: ho a disposizione memoria extra esterna chiamata **separate chaining (concatenazione separata)** che in qualche modo viola la politica delle table hash perchè vado a gestire ogni casella come un puntatore ad una lista collegata, dove aggiungo in coda la nuova entry che mi ha generato la collisione, di conseguenza ho una tavola hash come un array di liste.
In questo caso ho un costo O(1) per gli inserimenti, ma pago durante la ricerca, poichè in O(1) vado alla testa della lista, ma nel caso peggiore la liste potrebbe essere molto lunga ed avere un costo lineare O(n) per scandirla tutta.

Linear probing

Per risolvere la collisione posiziono la nuova entry all'interno della prossima casella libera iterativamente (ciclo nel caso mi trovi nell'ultima casella dell'array)
Ugualmente se eseguo una get, e la funzione mi punta in una casella dove però non trovo la chiave che sto cercando (poichè potrebbe essere stata spostata in basso dall'algoritmo), proseguo iterativamente andando a cercare nella prossima casella, se arrivo in una casella vuota, significa che non ho trovato la chiave e mi fermo.
Osservazioni sul linear probing:
-1) posso scegliere un **passo > 1,** ovvero posso decidere non di andare alla prossima casella, ma ad esempio avanti di 4 caselle (pericoloso se la tavola non fosse un numero primo, poichè facendo I salti non è garantito che io visiti tutte le caselle, ma potrei tornare sulla stessa sequenza)
- 2) si genera il fenomeno del **clustering primario (agglomerazione primaria):** aumenta la probabilità di occupare la prossima casella libera di altre zone già occupate, creando un compattamento di caselle non voluto.
- 3) **gestione delle rimozioni**: devo gestire in maniera accurata le rimozioni, altrimenti interrompo le catene di scansione, per fare ciò quando vado a fare una rimozione, appongo una marcatura sulla casella in cui ho eliminato una entry che mi dice che la casella è disponibile poichè successiva ad un eliminazione in modo da non bloccare la catena di scansione che si ferma solo se trova una casella libera senza marcatura speciale.
Esistono due metodi per segnare la marcatura:
1) gestisco un bit che mi segna se la casella ha una marcatura oppure no
2) si crea una entry speciale chiamata **Dummy** simbolica che viene sostistuita ad una entry eliminata
Con queste tecniche il numero di entry può essere minore o uguale alla dimensione dell'array
$n \leqslant N$

Fattore di carico (load factor) $\alpha = \frac{n}{N}$: percenturale di riempimento della tavola

Quando il fattore di carico supera ½ le prestazioni iniziano a peggiorare esponenzialmente, per risolvere ciò raddoppio subito la tavola e reinserisco i vecchi elementi con la nuova funzione di hashing.

Double Hashing (mi dice di quanto saltare)

Usa una seconda funzione di hashing d(k) che in caso di collisione moltiplica questo valore per **j** che mi indica il numero di tentativi che sto facendo sommato alla chiave i. Il tutto moltiplicato per mod N: **(i + jd(k)) mod N per j = 0,1,…,N – 1**
La dimensione N della tabella anche in questo caso deve essere un numero primo per poter testare tutte le caselle.
Effetto positivo contro il clustering primario poichè vengono fatte scansioni tramite salti grazie alla nuova funzione di hashing.
Funzione di compressione per questa funzione di hash: si sceglie un nuovo valore q (sempre numero primo) tale che esso sia minore di N (q < N) tale da avere come funzione **q – k mod q** che mi consente di avere un valore tra 1 e q
Costo: il tempo stimato di esecuzione per tutte le funzioni di un dizionario è O(1), mentre il caso peggiore O(n) in cui tutte le caselle della tabella sono piene.

Alberi

Un albero è un struttura “libera”, ovvero non esistono vincoli sul numero di figli di un nodo (albero n-ario)
Albero k-ario: albero in cui il numero dei figli non è superiore a k
Radice: nodo senza genitore
Nodo interno: nodo con almeno un figlio
Nodo esterno: nodo senza figli (**foglie**)
Profondità: numero di antenati
Altezza di un albero: massima distanza “radice-foglia” (altezza minima logaritmica e altezza massima lineare), data dalla lunghezza del ramo più lungo
Ramo: percorso che va dalla radice a una foglia

Metodi generici: integer size(), boolean isEmpty(), Iterator elements(), Iterator positions()
Metodi di accesso: posizion root(), position parent(p), positionIterator children(p)
Metodi di query: isInternal(p), isExternal(p), isRoot(p)
Metodi di aggiornamento: object replace(p,o)

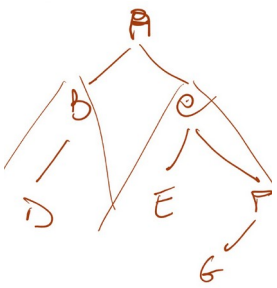
Visita in preordine: dall’alto verso il basso
Visita in postordine: dal basso verso l’alto

BinaryTree (Alberi Binari)

Estensione dei normali alberi dove ogni nodo interno ha al più due figli; I figli di un nodo sono una coppia ordinata (negli alberi k-ari non c’è ordinamento, quindi è sbagliato dire che un albero binario è un albero k-ario con k = 2)
Distinzione tra figlio destro e figlio sinistro non presenti negli alberi generici, ma presente in quelli binari.
Albero proprio: esclude la possibilità che esistano alberi con un figlio solo
Proprietà:
Dati **n numero dei nodi, e numero dei nodi esterni, i numero dei nodi interni ed h altezza dell’albero**:
e = i+1 (il numero dei nodi esterni è pari al numero dei nodi interni + 1)
n = 2e – 1 (il numero dei nodi è pari a due volte il numero dei nosi esterni – 1)
metodi addizionali: position left(p), position right(p), boolean hasLeft(p), boolean hasRight(p)

Tecniche di attraversamento degli alberi

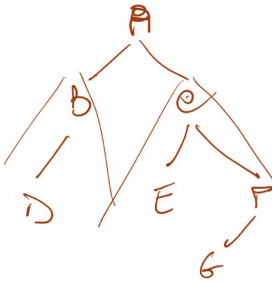
Composto da due famiglie:
Attraversamento in profondità: privilegiano lo spostamento **verticale** con tecniche **DFS (depth first search)** o anche dette a scandaglio.
Suddiviso in sottocategorie:
- **Visita in preordine** (in ordine anticipato): Si comincia dalla radice, visita un nodo e poi visita i suoi sottoalberi
Esempio:



visito A, B, D, C, E, F, G

```
void preOrder(root) {
   if( root == null) return;
      visit(root);
      preOrder(root → leftChild);
      preOrder(root → rightChild);
   return; }
```

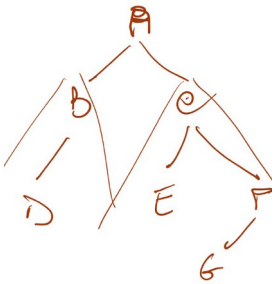
- **Visita in postordine** (in ordine posticipato): D, B, E, G, F, C, A




```
void postOrder(root) {
    if( root == null) return;          #nel caso di espressioni: if (root è foglia) return valore foglia;
    postOrder(root → leftChild);
    postOrder(root → rightChild);
    visit(root);
    return;
}
```

Di utilizzo negli alberi per le espressioni, se invece stampo le parentesi posso usare anche la visia in orderdel TDA tree,
Riassumendo si può dire che lo stampo con la visita in order e lo calcolo con la visita in post order

- **Visita in ordine** (in ordine simmetrico): D, B, A, E, C, G, F

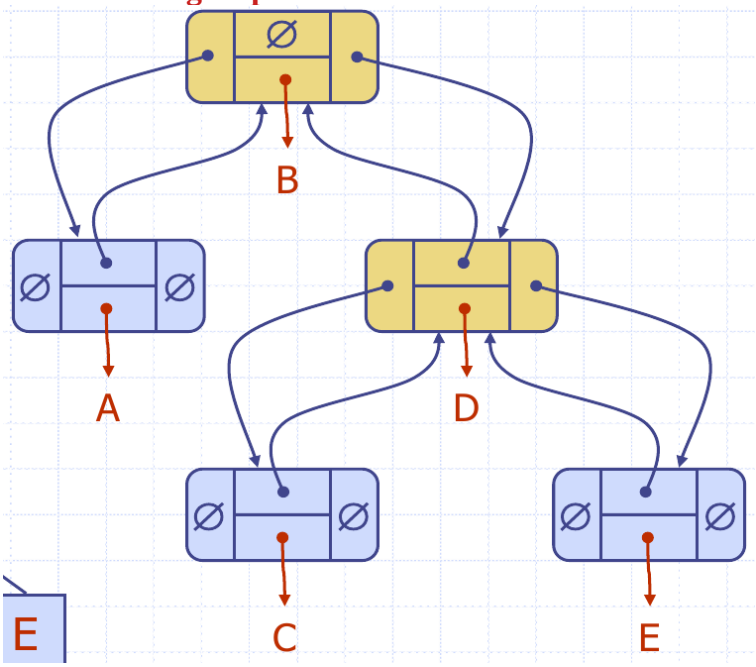



```
void inOrder(root) {
    if( root == null) return;
    inOrder(root → leftChild);
    visit(root);
    postOrder(root → rightChild);
    return;
}
```

Tour di Eulero
Esegue simultaneamente la visita in preorder da sinistra, inorder dal basso e in postorder da destra
passo base
azioni preorder
chiamata a Sx
azioni inorder
chiamata a Dx
azioni postorder

Attraversamento in ampiezza: privilegiano lo spostamento **orizzontale** con tecniche **BFS (breadth first search)** o anche dette a ventaglio.

Strutture collegate per alberi binari



Ogni nodo ha un puntatore al genitore (nella radice è null) raramente utilizzato, un puntatore contenente i dati, e due puntatori verso il figlio sinistro e il figlio destro (null se non presenti)
In c posso usare diverse soluzioni:
- Due struct differenti, una per i nodi e una per la struttura che mi permette attraverso il campo size di conoscere il numero di nodi dell'albero
- Una sola struct che mi definisce i nodi dell'albero che mi consente una gestione migliore durante le ricorsioni, perdo l'informazione size che dovrò, nel caso necessario dichiarare separatamente.
- Una basata su array (già studiata per gli heap) buona sono per alberi completi o quasi completi (nel caso peggiore ho un costo esponenziale) altrimenti è considerato un grave errore.

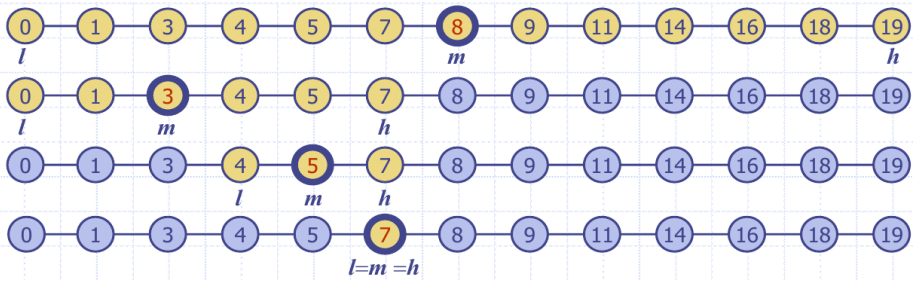
Rappresentazione per alberi n-ari generici (**non trattati approfonditamente per l'esame, lavoreremo con alberi binari)**: una struttura per l'albero che contiene root e size, ma cambia la struttura per il nodo che oltre al campo informativo contiene un campo chiamato **firstChild** e un campo chiamato **nextSibling** (prossimo fratello)

Mappe ordinate

Una mappa ordinata presenta lo spazio delle chiavi totalmente ordinato e delle operazioni aggiuntive quali **predecessore(k)** (data una chiave k, qual'è la chiave che precede k nell'ordinamento, ovvero qual'è la entry nella mappa che tra tutte le chiavi più piccole di k è quella più vicina a k) e **successore(k) (**analogamente è la chiave più piccola tra le chiavi maggiori di k)
non è necessario che la chiave k sia nella mappa
range-query(k1, k2): esistono due vesioni, la versione **conteggio** (che mi dice il numero di entry con le chiavi comprese tra k1 e k2) e la versione **lista** (che mi restituisce la lista delle dentry che hanno le chiavi comprese tra k1 e k2)

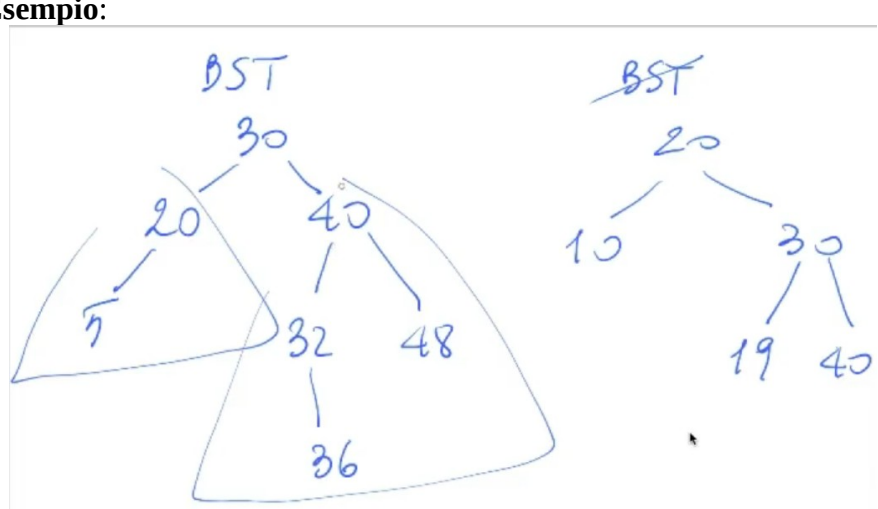
Ricerca binaria

La ricerca binaria è ciò che più si avvicina al concetto di mappa ordinata che viene implementata con un array con le chiavi ordinate che ci consente di trovare una determinata chiave in tempo logaritmico O(log n), pago però nell'inserimento (dato che pago n/2 shift per trovare il posto in cui inserire la nuova chiave) e nella rimozione di una chiave (dato che pago n/2 per ricimpattare le chiavi dopo l'eliminazione). (esempio find(7))



Alberi di ricerca binaria (Binary search trees BST)

Sono alberi binari in cui esiste un **ordinamento** fra le chiavi in cui dato un nodo k, le chiavi associate al sottoalbero **sinitro** sono tutte **minori** o uguali a k e le chiavi associate al sottoalbero **destro** sono tutte **maggiori** o uguali a k (dunque in caso di inserimento di una chiave multipla, posso scegliere arbitrariamente se andare a destra o a sinistra). Questa definizione è ricorsiva quindi deve essere valida per ogni nodo. **Dunque proprietà globale e non solo locale.**
Nei BST la chiave minima è la chiave più a sinistra (il che non vuol dire la più profonda dato che non c'è la completezza come negli heap), e la chiave massima è la chiave più a destra
Esempio:



Il bst a sinistra è valido perchè rispetta l'ordinamento, mentre quello a destra no perchè 19 è minore di 20.
Per avere un **albero proprio**, quindi che ogni nodo abbia due figli, dovremmo inserire delle foglie, non contenenti dati, ma vuote (**aggiungere queste foglie vuol dire raddoppiare il numero dei nodi poichè vogliamo tutti i nodi esterni come foglie vuote**)

Algoritmo di ricerca

Ogni accesso inizia dalla radice e tramite confronti (che ci dicono se andare a esplorare il nodo sinistro o destro) siamo in grado di arrivare al nodo contenente la chiave ricercata o nel caso non sia presente, arrivare ad una foglia e concludere che la chiave non è presente.
In questo modo invece di fare la visita dell'albero, visitiamo un ramo dell'albero che ci permette un costo molto inferiore.

Costo della visita inOrder (vale anche per le altre visite):
- Proprietà fondamentali:
1) ogni nodo è visitato almeno una volta
2) ogni nodo non è visitato due o più volte.
Dimostrazione 1: Andando a vedere il codice, supporre che un nodo non viene visitato, dato che la chiamata al nodo avviene all'interno del genitore, implica che neanche il genitore viene visitato e cosi ricorsivamente fino alla radice. Ma ciò è assurdo perchè quando chiamiamo inOrder lo facciamo proprio sulla radice.
Dimostrazione 2: Supponiamo per assurdo che un nodo venga visitato due volte, vuol dire che la chiamata a visita di quel nodo avviene due volte, ma ciò ugualmente a prima ricorsivamente vuol dire che il genitore è stato chiamato due volte, e cosi fino alla radice. Ma invocare due volte la radice vuol dire eseguire due volte la chiamata inOrder, non una sola, quindi non possiamo con una sola chiamata esplorare due volte la radice.
Da queste proprietà scaturisce un costo **O(n) per la visita**, ammesso che le operazioni di visita siano in tempo costante

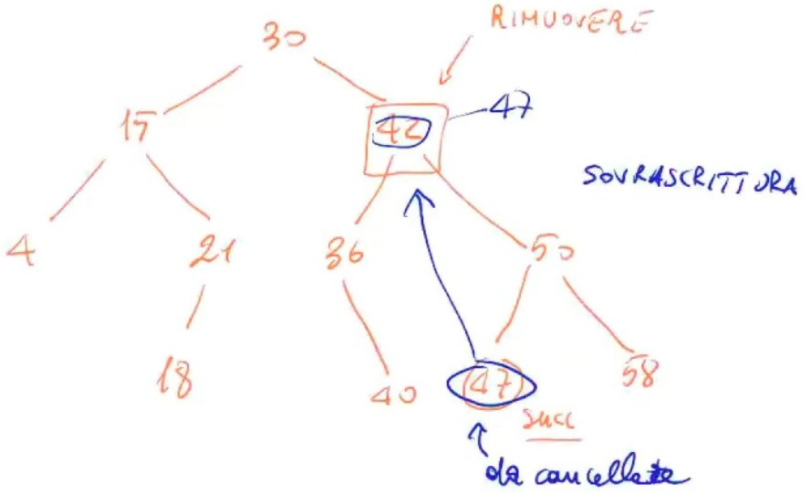
Search
Nella **search** ci concentriamo solo su un ramo e non su tutto l'albero, dunque il costo è dato dalla lunghezza del ramo che nel caso peggiore è l'altezza h dell'albero, quindi il costo è O(h) con h che asintoticamente può variare tra lg(n) e n ($lg(n) \leq h \leq n$)
Il caso peggiore sarebbe dunque O(n) ma questo implicherebbe proprio il caso più sfortunato in cui l'albero ha una forma sfortunata (difficile da incontrare), quindi consideriamo come **costo O(h)** mantenendo limitata l'altezza dell'albero, altrimenti non si noterebbe l'efficenza della ricerca visitando solo un ramo.

Inserimento
Nell'operazione di **inserimento** di una chiave devo rispettare le regole di ordinamento, il che mi porta ad avere ogni inserimento in una foglia.
Il worst case mi porta ad aggiungere una foglia al ramo più lungo, il che mi porta un costo pari a O(h)

Ricerca del predecessore e del successore (si svolge in maniera speculare)
POCO CHIARO, MA NON CI SONO LE SLIDE
Il predecessore di una chiave k è la più grande chiave (nel BST) < k (non è rilevante se k sia presente o no nel BST)
Inizializiamo il candidato a null, poi andiamo alla radice v, **se la chiave dento v < k allora abbiamo un candidato diverso da null**, se la chiave è proprio uguale a k non va bene, è già troppo, come anche una chiave > k è troppo.
Scendiamo verso sinistra nell'albero se la chiave di v è magiore o uguale a k, scendiamo verso destra se la chiave di v è minore di k.
Scendiamo dunque a destra, dato che gli elementi a sinsitra sarebbero più piccoli di k, se troviamo un elemento v' minore di k, impostiamo lui come candidato perchè è meglio di v, continuiamo finchè possiamo e l'ultimo candidato sarà il predecessore cercato.
Se il candidato rimane a null, vuol dire che non abbiamo trovato un predecessore.
Il **costo** di queste operazioni è **il costo di discesa lungo un ramo dell'albero**, dunque **O(h)**

Cancellazione (il prof non prevede le foglie vuote)

L'algoritmo di cancellazione prevede tre casi:
1) **Cancellazione di un nodo foglia**: Attraverso l'algoritmo di ricerca **cerco la chiave del nodo da cancellare** (costo **O(h)**) e poi **metto a null il puntatore che puntava alla foglia** (intervento sul genitore), **costo O(1)** (fare attenzione se era il figlio sinistro o il destro). **Costo complessimo O(h)**
2) **Cancellazione di un nodo con un figlio**: Cerco il nodo w da cancellare, indico con v il genitore e u il figlio; farò puntare v non più a w, ma al figlio u in modo da mantenere le informazioni precedenti che non voglio eliminare e rimangano inalterate le proprietà del BST.
Anche questa operazione nel caso peggiore ha un **costo di O(h)**
3) **Cancellazione di un nodo con due figli**: Usiamo la tecnica del precessore o del successore, è indipendente, avendo il nodo due figli non deve per forza partire dalla radice, ma può partire dal nodo che deve essere rimosso.
Una volta trovato il successore migliore, ne prendiamo il contenuto e lo usiamo per sovrascrivere il nodo da rimuovere mantenendo le proprietà del BST per poi infine cancellare il nodo copiato.
Il predecessore/successore non possono mai avere due figli, perchè uno dei due figli porterebbe a soluzioni migliori. In particolare un successore non può avere il figlio sinistro, perchè li ci sarebbe il successore migliore e un predecessore non può avere il figlio destro per lo stesso motivo.
Anche questa operazione nel caso peggiore ha un costo di **O(h)**
Esempio:



Possiamo dunque riassumere dicendo che tutte e tre le operazioni hanno costo O(h)
Prendendo tutte le permutazioni possibili dell'albero, possiamo dire con molta probabilità che l'altezza dell'albero sarà logaritmica.
Verificare che un albero è un BST in tempo lineare, consiste nel visitare in ordine simmetrico (inorder) e verificare che vengano incontrate chiavi ordinate in maniera crescente. Simmetricamente posso verificare da destra le chiavi in ordine decrescente.

Range Query

Dato un albero e un intervallo [a,b] cercare tutte le chiavi in questo intervallo.
1) Cerco a (potrebbe anche non esserci nell'albero) e se non c'è, la più piccola chiave maggiore di a
2) Una volta trovato il valore interessato, continuo esaminando le chiavi seguendo l'ordine dato da **inOrder**
3) Mi fermo a b, o nel caso non ci fosse, il più grande degli elementi minori di b

```
void inOrder(root) {
    if( root == null) return;
    inOrder(root → leftChild);          #proteggo con un test la chiamata, poichè non voglio
                                        farla se mi porta a sinistra di a
    visit(root);
```

```
    postOrder(root → rightChild);          #proteggo con un test la chiamata, poichè non voglio
                                           farla se mi porta a destra di b
    return;
}
```

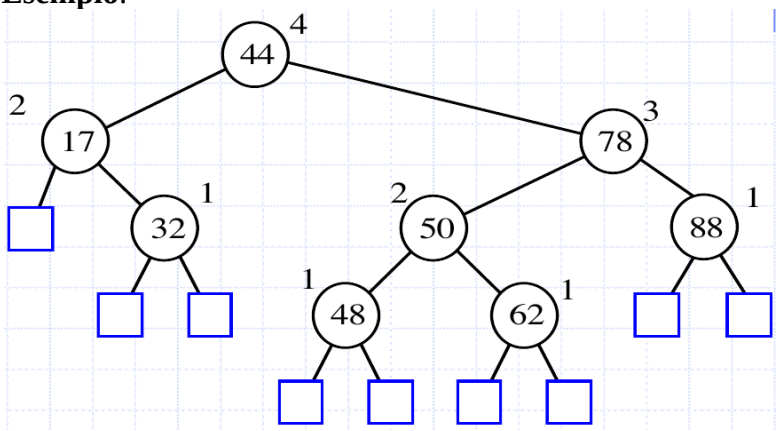
Costo pari all'atezza dell'albero O(h) per trovare a, poi spendo **O(h + K)** con K dimensione delle chiavi (output) nell'intervallo [a,b]

Chiavi Multiple (dizionario, nelle mappe non ci possono essere duplicati)

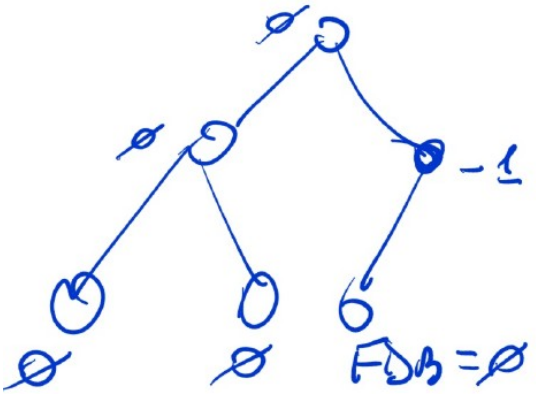
Insert: **le chiavi uguali possono stare sia nel sottoalbero sinistro che in quello destro**, devo dunque usare una politica che mi consenta poi di essere riusata anche nelle altre funzioni quali find.
findAll: per farla funzionare devo immaginarla come una range query per (k – e , k + e) con e componente infinitesima. (in questo modo possiamo non scegliere a priori un criterio di inserimento)

Alberi AVL (alberi bilanciati)

Gli alberi AVL sono alberi binari bilanciati tali che per ogni nodo interno dell'albero la differenza di altezza tra il nodo sinistro e il nodo destro è al più 1.
Esempio:



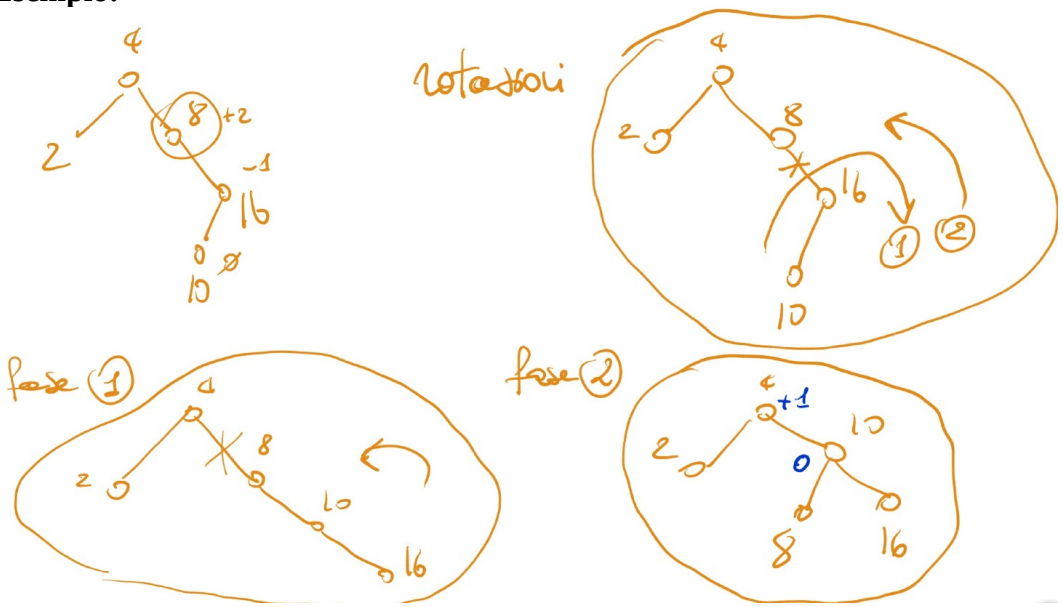
Un albero di 2 livelli ha altezza 1, un albero con solo la radice ha altezza 0 e un albero con 0 livelli ha altezza -1
Fattore di bilanciamento: indica la differenza di livelli tra il nodo destro e quello sinistro
Un albero AVL ha un fattore di bilanciamento compreso tra 1, 0 e -1
Esempio:



Gli AVL hanno sempre altezza O(log n)

Caso limite (alberi di fibonacci): a parità di numero di nodi l'altezza è massima oppure data un'altezza il numero di nodi è minimo e tutti i fattori di bilanciamento, in modulo, per i nodi interni sono pari ad 1
Gli alberi AVL che appartengono al caso limite prendono il nome di **alberi di fibonacci**
Otteniamo un albero di fibonacci più alto prendendo i due alberi precedenti e mettendoli insieme con un nodo radice.
Quindi il numero di nodi di un albero di fibonacci di altezza h è **F(h)** lo possiamo ottenere come:
F(h) = F(h – 1) + F(h – 2) +1
Dimostrando che gli alberi di fibonacci di n nodi hanno altezza O(log n) dimostro anche che tutti gli alberi AVL hanno altezza O(log n) poichè hanno altezza inferiore agli alberi di fibonacci che ne sono il caso limite.
Dimostrazione per induzione: indichiamo con n(h) il numero minimo di nodi di un albero AVL di altezza h.
Possiamo subito notare che n(1) = 1 e n(2) = 2, per n > 2, un albero AVL di altezza h contiene il nodo radice e due sotto alberi, uno di altezza h – 1 e uno di altezza h – 2.
Sapendo che n(h – 1) > n(h – 2), possiamo dire che n(h) > 2n(h – 2) ma è una ricorrenza quindi vale anche n(h) > 4n(h – 4) e in generale **n(h) > 2^in(h – 2i)**
Risolvendo il caso base per h – 2i = 1 e risolvendo rispetto ad i, otteniamo che **n(h) > $2^{h/2-1}$.**
Passando ai logaritmi notiamo il limite logaritmo dell'altezza degli alberi: **h < log(n(h) + 2)**
Possiamo dunque così dire che l'altezza di un albero AVL è pari a O(log n)

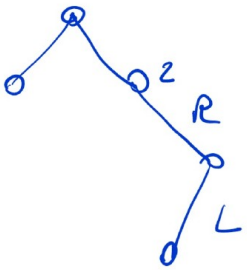
Quando faccio un operazione, ad esempio l'inserimento, se il risultato è ancora un AVL dico che è tutto apposto, se invece si sono sbilanciati uno o più nodi, devo intervenire per ripristinare la condizione di bilanciamento.
Per intervenire devo fare operazioni chiamate **rotazioni**
Esempio:



Fase 1 = inverto 16 con 10
Fase 2 = taglio momentaneamente 8, porto il 10 su, e poi sistemo l'8 alla sua sinistra essendo inferiore a 10.

Esistono quattro tipi di rotazione: LL, RR, LR, RL con L = left e R = right
LL e RR sono le più semplici perchè sono singoli movimenti
LR e RL invece eseguono movimenti doppi
Corrispondono ai movimenti che devo compiere dal primo nodo sbilanciato, fino alla foglia
Il numero di rotazioni che mi servono per bilanciare l'albero è sempre costante

Esempio:



Rimozione da alberi AVL

Si esegue la rimozione come nei normali alberi BST, poi bisogna ribilanciare l'albero nel caso sia stato sbilanciato. Non si guarda il ramo dove avviene lo sbilanciamento, ma il ramo più lungo (sempre partendo dal nodo sbilanciato), per questo non è garantito che avvenga una singola rotazione.
Nel caso peggiore ho h rotazioni pari all'altezza dell'albero.

Costo operazioni AVL di altezza h

Ricerca: O(h)
Inserimento: O(h) per la parte BST **+ O(1)** per le rotazioni, quindi ancora in **O(h)**
Cancellazione: O(h) per la parte BST + **h O(1)** per le rotazioni, quindi **O(h) + O(h)**, che appartiene ancora a **O(h)**
Un AVL di n nodi ha un altezza massima parti all'altezza di un albero di fibonacci di n nodi perché è minore di 2 log(n).
Dunque tutte le operazioni hanno un costo O(log n)

Nei nodi AVL abbiamo un campo in più che corrisponde **all'altezza del suo sottoalbero** che mi permette di calcolare il fattore di sbilanciamento di un nodo in tempo costante.
I dublicati devono poter andare indifferentemente a sinistra o a destra.
Infatti, se cosi non fosse, e si decidesse di mettere tutti i duplicati sempre nel sottoalbero destro avremmo il paradosso di non poter fare rotazioni.

Grafi

Un grafo è una coppia (V, E), dove V è un insieme di nodi detti **vertici** ed E un insieme di **archi / spigoli (edges)**, entrambi aventi informazioni.
Dal punto di vista modellistico, mentre i vertici sono oggetti assestanti che definiscono l'insieme, l'insieme degli archi gode della proprietà che ciascuno degli archi è un collegamento fra due vertici.
Definizione matematica: un arco è una coppia ordinata di vertici. (relazione binaria)
Nei grafi abbiamo **relazioni irriflessive**: ogni nodo non può essere in relazione con se stesso, non ammettiamo self-loop, ma possiamo dire che spesso la relazione è simmetrica (a R b => b R a)
Arco orientato: quando c'è una relazione origine – destinazione, graficamente abbiamo una freccia

Arco non orientato: graficamente non c'è la freccia

Grafo orientato/diretto è un particolare grafo in cui **tutti gli archi sono orientati**
Grafo non orientato/indiretto è un particolare grafo in cui **tutti gli archi NON sono orientati**
Un **grafo semplice** è un grafo non orientato e privo di self-loop (anche privo di archi multipli)
Ogni arco ha due estremità anche dette **endpoints**
Adiacenza di vertici vuol dire che due vertici sono collegati da un arco (adiacenti)
Archi incidenti: un arco è incidente sui due vertici ai quali è collegato
Il **grado** di un vertice è stabilito dal numero di archi incidenti su un vertice
Regolarità di un grafo: un grafo è detto regolare se e solo se per definizione tutti i vertici hanno lo stesso grado
Percorso (path): una sequenza alternata di vertici e archi, inizia con un vertice e ne finisce con un altro
Percorso semplice: un percorso in cui i vertici compaiono solo una volta
Ciclo: caso particolare di percorso in cui il primo vertice corrisponde anche all'ultimo
Ciclo semplice: un ciclo in cui l'unico vertice che si ripete è quello da dove si parte e dove si arriva
La somma dei gradi di tutti i vertici è sempre pari al doppio del numero degli archi
Dato **n numero di vertici, m numero di archi e deg(v) grado del vertice v:**

$$\sum_{v} deg(v) = 2m$$

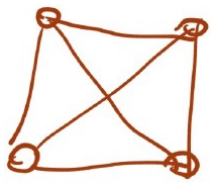
Ciò implica che non esiste un arco che non viene visto e non c'è un arco che non viene visto due volte.
In un grafo non orientato senza self-loops e archi multipli il **massimo rumero di archi** è pari a:

$$m \leq n\frac{n-1}{2}$$

Poichè ogni nodo al massimo è collegato a tutti gli altri nodi ed avrà n – 1 archi.
NON esiste un numero minimo di archi.
Un Grafo connesso non presenta nodi isolati.

Grafo completo K_n con n numero dei vertici

Un grafo completo è un grafo in cui tutti i vertici sono connessi tra di loro
Un grafo può essere rappresentato in infiniti modi, i **disegni non planari** sono quelli in cui due archi si incontrano in un punto che non è un vertice.
Esempio:



Teorema di Kuratowsky (grafo planare)
Un **grafo è planare** (ammette un disegno senza incroci) se non contiene ne $\mathbf{K_5}$ (grafo completo di 5 nodi) ne $K_{3,3}$ (grafo completo **BIPARTITO**, ovvero che i nodi sono divisi in due sottoinsiemi e I nodi sono collegati solo con nodi dell'altro insieme e non dello stesso insieme)
Essendo il numero di archi m:

$0 \leq m \leq n\frac{n-1}{2}$ quindi con $m \in O(n^2)$

I grafi con $O(n^2)$ archi vengono detti **densi**
I grafi con O(n) archi vengono detti **sparsi**
I grafi planari sono tutti sparsi.

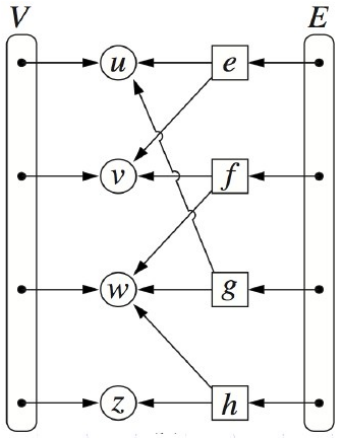
Operazioni sui grafi

numVertices(): ritorna il numero di vertici del grafo
vertices(): ritorna un iteratore di tutti i vertici del grafo

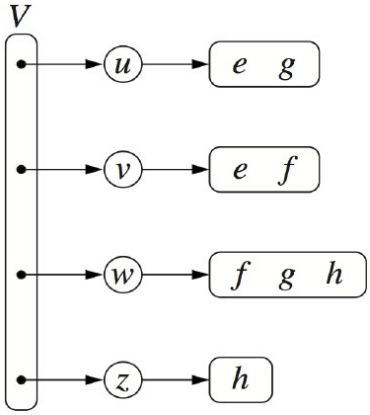
numEdges(): ritorna il numero di archi del grafo
edges(): ritorna un iteratore di tutti gli archi del grafo
getEdge(u,v): ritorna l'arco che collega i vertici u e v, se non esiste ritorna null. Nel caso di grafo indiretto non c'è nessuna differenza tra getEdge(u,v) e getEdge(v,u), ciò non vale nel caso di archi diretti poichè sarebbero due archi con due informazioni diverse.
endVertices(e): Dato un arco e, mi dice chi sono i due vertici che collega. Se lo invoco su un grafo diretto, il primo vertice sarà l'origine e il secondo la destinazione.
opposite(v,e): Dato un vertice v e un arco e mi restituisce il vertice che sta dall'altra parte dell'arco. Da errore se e non è un arco incidente su v.
OutDegree(v): ritorna il numero di archi uscenti dal vertice v nel caso di grafo orientato
inDegree(v): ritorna il numero di archi entranti nel vertice v nel caso di grafo orientato. Nel caso di grafo non orientato ritorna lo stesso valore di OutDegree(v).
outgoingEdges(v): ritorna un iteratore contenente gli archi uscendi dal vertice v
incomingEdges(v): ritorna un iteratore contenente tutti gli archi entranti al vertice v.
insertVertex(x): inserisce un vertice isolato contenente l'informazione x.
insertEdge(u,v,x): crea e ritorna un nuovo arco che collega i vertici u e v, contenente l'informazione x; ritorna un errore se esiste già un arco che collega i due vertici.
removeVertex(v): rimuove il vertice v e tutti i suoi archi incidenti dal grafo
removeEdge(e): rimuoviel'arco e dal grafo

Strutture

Tutte le rappresentazioni hanno in comune una lista di vertici e una lista di archi
Lista di archi: fa vedere il collegamento tra ogni arco e i vertici



Liste di adiacenza: per ciascun vertice è definita una lista degli archi incidenti su di esso



Con le liste di adiacenza mi è facile gestire le informazioni sui percorsi.

Matrice di adiacenza: composta da m vertici e n archi, nella posizione ij troviamo informazioni sull'arco che va da i a j

	0	1	2	3
u → 0		*e*	*g*	
v → 1	*e*		*f*	
w → 2	*g*	*f*		*h*
z → 3			*h*	

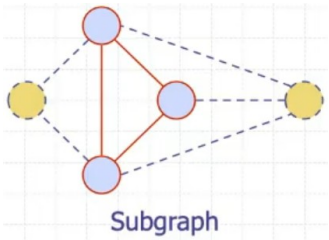
Se il grafo è **non diretto** la matrice è SIMMETRICA rispetto alla diagonale principale che rimane sempre vuota perchè corrisponderebbe ai self-loop che noi non ammettiamo.
Ha senso dunque per grafi non orientati allocare solo mezza matrice attraverso un array di puntatori alle righe.

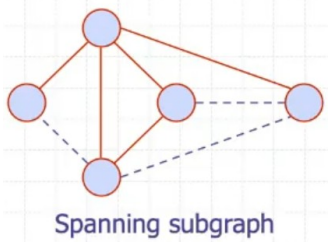
Performance

▪ n vertices, m edges ▪ no parallel edges ▪ no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n + m$	$n + m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Sottografo

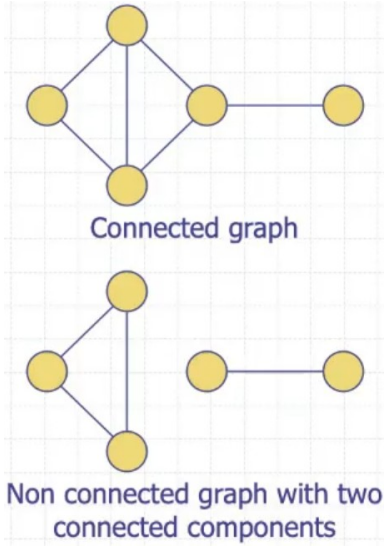
Un sottografo é un **grafo** che presenta un sottoisieme di archi e un sottoinsieme di vertici di un altro grafo.





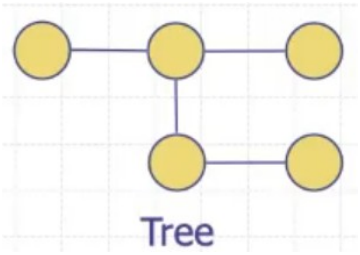
Si chiama **sottografo ricoprente (spanning subgraph)** un sottografo che si ottiene rimuovendo solo archi e nessun vertice.
Un sottografo si dice **indotto** se è ottenuto specificando solamente il sottoinsieme di vertici, posso prendere tutti gli archi che sono incidenti a questi vertici

Un grafo è detto **connesso** se comunque prendo due vertici esiste un percorso che li collega

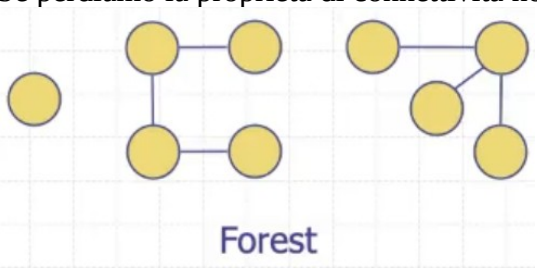


Una **componente connessa** è un sottografo **massimale** che è connesso (immagine sopra)
massimale: non è possibile renderlo più grande senza perdere la proprietà di connessione.

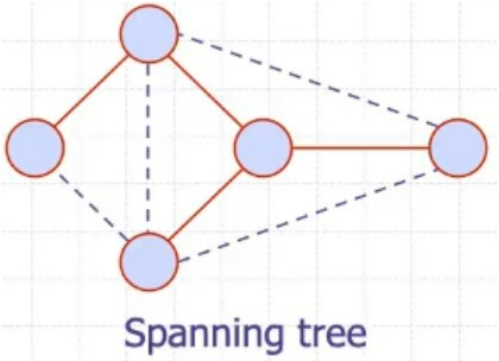
Un albero è un grafo aciclico connesso.
Gli alberi con cui abbiamo lavorato fino ad adesso evidenziavano un nodo particolare, la radice, e questi prendono il nome di **radicati (rooted),** mentre nei grafi non abbiamo bisogno di specificare la radice.



Se perdiamo la proprietà di connettività non abbiamo più un albero ma una **foresta**



Le componenti connesse di una foresta sono alberi.
Albero ricoprente (Spanning tree): è un particolare sottografo che è sia un albero ma anche connesso



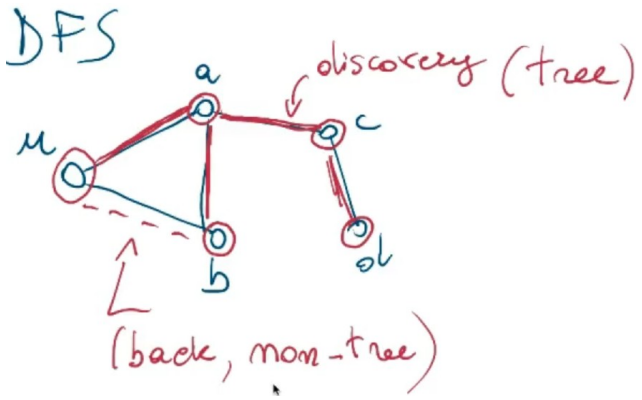
In maniera analoga rinunciando alla propriterà di connettività possiamo parlare di **foresta ricoprente (Spanning forest)**

DFS (Depth-First Search)

Escludiamo self-loops e archi multipli**.**
Una DFS: Visita tutti i vertici e gli archi del grafo G e determina se G è connesso; Nel caso non sia connesso determina le componenti connesse del grafo (determina dunque una spanning forest)
Una DFS su un grafo con **n** vertici e **m** nodi impiega un tempo **O(n + m)** (tempo lineare in funzione dell'input).
Con opportune modifiche si può estendere una DFS per risolvere altri poblemi come trovare un ciclo all'interno del grafo oppure trovare un percorso tra due vertici.
Una DFS nei grafi corrisponde al percorso di eulero negli aberi binari.
L'**algortimo**: Dato in input un grafo G e un vertice u, esplora tutto il grafo raggiungibile dal vertice indicato.
Mentre negli alberi non succedeva che io visitassi due volte lo stesso nodo, nei grafi può succedere che io visiti un nodo già visitato attraverso un percorso differente.
Devo quindi far in modo che ciò non avvenga, per far ciò uso una **marcatura** che **mi indica se il nodo è già stato visitato oppure no**, una pratica comune è associare un numero corrispondente all'ordine di visita dei nodi.

Algorithm DFS(G, u):
Input: A graph G and a vertex u of G
Output: A collection of vertices reachable from u, with their discovery edges
Mark vertex u as visited.
for each of u's outgoing edges, $e = (u, v)$ **do**
if vertex v has not been visited **then**
Record edge e as the discovery edge for vertex v.
Recursively call DFS(G, v).

Inizio markando il nodo u datomi in input come visitato poi eseguo un ciclo for per ogni arco uscente dal nodo u (nel caso di archi bidirezionali gli archi che escono sono uguali a quelli che entrano) e per ogni vertice v che sta dall'altra parte dell'arco, se non è stato visitato marchiamo l'arco che porta alla scoperta di v (in modo da distinguerli da quelli che non abbiamo ancora visitato e di conseguenza anche il vertice) e richiamiamo ricorsivamente l'algoritmo sul vertice appena visitato.
Se l'arco mi porta ad un nodo già visitato non lo marco come discovery (**mi consente di evitare cicli**) ma lo marco come **back**.
A fine ciclo tutti gli archi vengono visitati e vengono annotati come **discovery** se mi portano a nodi non visitati e prendono il nome di **archi tree (poichè non ci sono cicli)**, mentre gli altri che mi hanno portato a nodi già visitati prendono il nome di **archi back o non-tree**.
Esempio percorso:



Non è possibile che l'albero costituito dagli archi discovery non sia connesso poichè se l'albero non è connesso, si dovrebbe avere un salto tra due alberi connessi, il che non è possibile dato che visitiamo solo nodi adiacenti.
Scegliamo il costo di **O(m+n)** poichè se il grafo non è connesso il numero di vertici potrebbe superare il numero degli archi.

Path Finding

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

Dato in input un grafo e due vertici, trova il percorso che li collega.
Usiamo l'algoritmo della DFS e con le scritte in rosso aggiungiamo la parte aggiuntiva per indicare le informazioni sul percorso attraverso una pila.
Iniziamo segnando il vertice iniziale come visitato e lo aggiungiamo alla pila, se siamo arrivati al nodo desiderato ritorniamo gli elementi della pila altrimenti per ogni vertice incidente controlliamo se sia un arco inesplorato, se lo è andiamo a vedere il vertice che c'è dall'altra parte.
Se il vertice è inesplorato, settiamo l'arco come discovery e aggiungiamo l'arco alla pila e chiamiamo ricorsviamente sul nuovo nodo, se invece il vertice è già esplorato,lo impostiamo come back.
Se finiscono i cicli e quindi anche le ricorsioni, arriviamo a S.pop(e) cioè togliamo l'arco e perchè vuol dire che non ci sta instradando verso la strada giusta (come quando nel labirinto arriviamo ad un vicolo cieco e bisogna tornare indietro)
Se alla fine non arrivo alla destinazione, tolgo anche il vertice iniziale con S.pop(v)

Cycle Finding (variante del path finding per individuare i cicli)

```
Algorithm cycleDFS(G, v, z)
    setLabel(v, VISITED)
    S.push(v)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            S.push(e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                pathDFS(G, w, z)
                S.pop(e)
            else
                T ← new empty stack
                repeat
                    o ← S.pop()
                    T.push(o)
                until o = w
                return T.elements()
    S.pop(v)
```

Inizialmente mettiamo in pila tutto ciò che incontriamo, poi per ogni arco incidente sul vertice sotto esame v, se l'arco l'ho già usato, salto e passo al prossimo, se l'arco invece è unexplored, faccio il push dell'arco sulla pila e verifico se il nodo che sta dall'altra parte è inesplorato.
Se risulta inesplorato, imposto l'arco come discovery e chiamo una visita DFS.
Se la chiamata alla DFS non mi ha fatto trovare un ciclo, vuol dire che l'arco e non mi ha portato verso di esso e lo rimuovo dalla pila.
Se il nodo che sta dall'altra parte invece è già stato visitato, vuol dire che sono davanti ad un ciclo.
In questo momento nella pila ho un'alternanza di archi e vertici, perciò creo una nuova pila T in cui vado a inserire gli elementi della pila a ritroso effettuando la pop, finchè non arrivo al nodo w già visitato e ritorno gli elementi della pila dove ho il ciclo (senza gli elementi che mi hanno permesso di arrivare al ciclo).
Se non ho trovato il ciclo alla fine tolgo anche v dalla pila.

DFS per un intero grafo

```
Algorithm DFS(G)
    Input graph G
    Output labeling of the edges of G
        as discovery edges and
        back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
        DFS(G, v)
```

Inizialmente procedo con un doppio ciclo per impostare tutti i vertici e tutti gli archi come inesplorati. Inizializzazione con costo **O(m+n)** per m archi e n nodi.
Poi eseguo un ciclo per tutti i vertici del grafo, se il vertice risulta inesplorato, faccio partire una DFS proprio da li.

```
Algorithm DFS(G, v)
    Input graph G and a start vertex v of G
    Output labeling of the edges of G
        in the connected component of v
        as discovery edges and back edges
    setLabel(v, VISITED)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
            w ← opposite(v,e)
            if getLabel(w) = UNEXPLORED
                setLabel(e, DISCOVERY)
                DFS(G, w)
            else
                setLabel(e, BACK)
```

Per vedere quante sono le componenti connesse del grafo posso usare questo algoritmo per gestire un contatore che mi va a ricordare quante sono state le chiamate a DFS a partire da un vertice.

Grafi orientati/diretti (Digrafi)

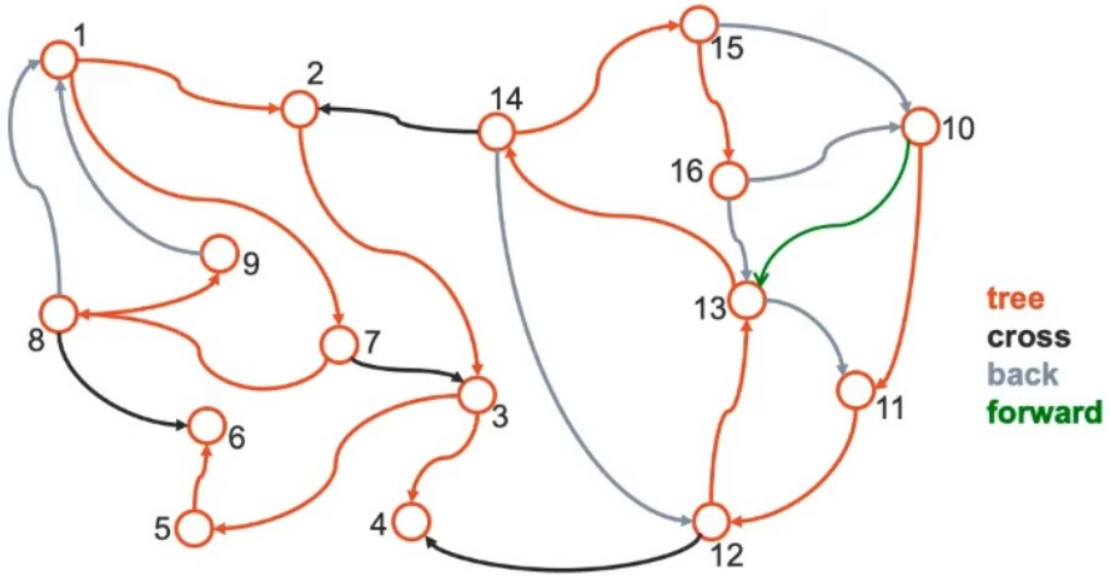
Il numero di archi presenti nei grafi orientati è il doppio rispetto a quelli non orientati:

$$m \leq n(n-1)$$

Le liste di adiacenza possono contenere i nodi adiacenti oppure informazioni sugli archi.
Mentre nei grafi non orientati bisognava aggiungere un nuovo arco ad entrambi i vertici toccati, ora non dobbiamo più farlo dato che il grafo sarà orientato e avrà un'unica direzione e avremo una coppia ordinata di vertici e non un insieme.
É comunque utile in alcuni casi avere due liste di adiacenza per ogni nodo, una per gli archi entranti e una per quelli uscenti.
La matrice di adiacenza nei grafi orientati non sarà più simmetrica come nei grafi non orientati, faremo dunque uso di tutta la matrice, con le caselle sulla diagonale principale che rimarranno comunque vuote dato che escludiamo i self-loops.

DFS diretta (D-DFS)

Come la DFS normale rimane la proprietà per cui ottengo ancora una foresta ricoprente, ma diventa meno interessante.
L'algoritmo non cambia, ma l'etichettatura degli archi diventa più ricca di informazioni: un arco può essere di quattro tipi: **tree, back, cross, forward**
Archi tree: quelli che definiscono l'albero finale
Archi back (ultimo arco prima della chiusura di un ciclo): archi da u a v, essendo u un discendente di v, ovvero che esiste un percorso che parte da v e ritorna a u e denuncia la presenza di un ciclo.
Archi forward (un arco che porta ad un nodo già visitato ma non chiude un ciclo): archi da u a v sapendo già che v è un discendente di u
Archi cross: quando ne u ne v sono discendenti l'uno dell'altro e non sono ne archi back ne archi forward



Algoritmo D-DFS

Introduciamo una doppia etichettatura (int) dei vertici:
Prima marcatura = discovery label: intero progressivo che indica l'ordine di scoperta del vertice
Seconda marcatura = leaving label: intero progessivo che indica l'ordine di abbandono dei vertici (quando chiudiamo il livello ricorsivo), quindi abbiamo già esplorato tutti i nodi raggiungibili a partire da un determinato nodo. (quindi tutti i nodi discendenti)
Dato che settiamo la marca di ingresso all'inizio della chiamata e la marca di uscita alla fine della chiamata, se ci domandiamo se un nodo v sia discendete di un altro nodo u, basta andare a vedere se la marca di u di uscita è segnata o meno, se non lo è vuol dire che v è un discendente di u dato che si troverà all'interno della chiamata di u, se invece fosse diversa da zero vuol dire che abbiamo già visitato tutti i discendenti di u.

Diverso approccio: invece di usare due marcature, posso usare un'unica marcatura che all'inizio della chiamata marca il vertice u come **exploring** e alla fine come **explored**.
Per verificare dunque se un vertice è discendente di u basta vedere se la marca è ancora in stato exploring oppure no.

```
D-DFS(Digraph G, Vertex v) {
 setDiscoveryLabel(v, getNextDLabel());
 for all e in outgoingEdges(v)
  if(getLabel(e) == UNEXPLORED)
   w = opposite(e, v)
   if(getDiscoveryLabel(w)==UNEXPLORED)
    setLabel(e, DISCOVERY)
    D-DFS(G, w)
   else if(getLeavingLabel(w) == 0)
    setLabel(e, BACK)
   else if(getDiscoveryLabel(v)<getDiscoveryLabel(w))
    setLabel(e, FORWARD)
   else setLabel(e, CROSS)
 setLeavingLabel(v, getNextLLabel())
}
```

GetNextDLabel() è una funzione che mi tiene conto di tutte le label già assegnate e mi dice qual è la prossima da assegnare.
Inizio marcando il vertice v con setDiscroveryLable (dunque il primo approccio con due marcature) e per ogni arco e uscente vado a verificare:
Se l'etichetta dell'arco e è inesplorato andiamo a vedere il vertice w opposto al vertice v; se questo vertice ha una discroverylabel UNEXPLORED, gli associamo l'etichetta di DISCOVERY (tree edge) e invochiamo la DFS.
Altrimenti se il vertice w è già stato visitato bisogna vedere il tipo di arco da assegnare ad e:
se la label di uscita di w è uguale a zero (ciò vuol dire che il nodo w è un discendente di v) segno l'arco come **BACK**, altrimenti mi porterebbe ad un ciclo.
Se invece ho scoperto w dopo di v imposto l'arco come **FORWARD**, altrimenti se i precedenti else sono risultati falsi setto l'arco come **CROSS**.
Arrivato alla fine del cilo imposto la lable di v come leaving.

Possiamo riassumere dicendo che per scoprire se un grafo è ciclico/aciclico basta la marcatura minima, se invece vogliamo sapere se gli archi sono di tipo cross o forward serve la doppia marcatura.

Connettività sui grafi diretti (digrafi)

Debole (week): Un grafo viene detto **debolmente connesso** se e solo se il grafo non orientato sottostante è connesso (cioè se eliminando tutti i versi di percorrenza e rendendo il grafo non orientato, esso risulti connesso, possiamo chiamare il grafo orientato di partenza debolmente connesso).
Forte (strong): Un grafo viene detto **fortemente connesso** se e solo se comunque scegliamo una coppia di nodi (u,v) esiste un percorso orientato da u verso v (qualsiasi sequenza vertice arco)
Essendo qualsiasi coppia di vertice vale anche la coppia (v,u), ciò necessità l'esistenza di un ciclo.
Dunque se sappiamo che un grafo è aciclico, esso non può essere fortemente conneso.
Anche se un grafo non è fortemente connesso possiamo vedere se esso ha delle componenti fortemente connesse.
Grafo Trasposto: Dato G grafo chiamiamo un grafo trasposto G^T se ogni suo arco è stato trasposto (ne è stato invertito l'ordine).
Normalmente siccome è costosto creare un nuovo grafo che sia il trasposto di quello precedente, cerchiamo di lavorare sullo stesso grafo interpretandolo come grafo trasposto (basta interpretare gli archi nei versi opposti).

Teorema

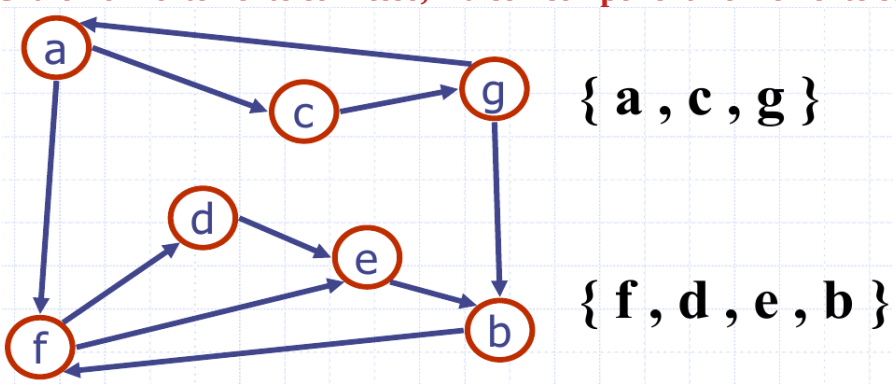
G è fortemente connesso se e solo se G^T è fortemente connesso

La proprietà di connessione forte viene riservata durante la trasposizione.

Algortimo per la connettività forte

Se prendo un vertice v e eseguo una visita DFS e raggiungo tutti i nodi,allora il grafo è fortemente connesso, altrimenti se mi rimane qualche nodo non visitato durante la DFS il grafo non lo è.
Possono esistere nodi per cui non vale il percorso inverso, quindi per avere un algoritmo completo, inizialmente eseguo una prima visita, se questa non mi permettere di raggiungere tutti i nodi del grafo, già so che il grafo non è fortemente connesso.
Se invece riesco a visitare tutti i nodi ho verificato la condizione necessaria affinché il grafo sia fortemente connesso, ma è solo sufficiente, perciò effettuo dallo stesso nodo una visita sul grafo trasposto, se trovo un vertice v non visitato, il grafo non è fortemente connesso,
Se invece ho visitato tuti i nodi anche nel grafo trasposto, posso concludere che il grafo di partenza è fortemente connesso.

Grafo non fortemente connesso, ma con componenti fornemente connesse.



Un grafo con componenti fortemente connesse è composto da sottografi massimali tali che ogni vertice può raggiungere tutti i vertici del sottografo stesso.

Algoritmo stronglyConnectedComponent(G)

Eseguiamo una DFS sul grafo G per calcolare il timestamp di abbandono per ogni vertice V.
Eseguiamo una DGS su G^T considerando tutti i nodi nel main loop in ordine decrescente secondo il timestamp di abbandono calcolato prima. (L'ultimo che abbandono è il primo da cui farò ripartire la visita)
In ouput otterrò ogni albero del grafo trasposto che rappresenta una componente fortemente connessa.

Nodo sorgente: nodo che non ha archi entranti
Nodo pozzo: nodo che non ha archi archi uscenti

DAG (directed acyclic graph)

Ordine parziale come un grafo orientato aciclico.
Se la coppia (q,p) appartiene ad R e (p,q) appartiene a R, possiamo dire che q e p sono **confrontabili**
Se la coppia (q,p) non appartiene ad R e (p,q) non appartiene ad R, possiamo dire che q e p **non sono confrontabili**.
In una relazione **d'ordine parziale** non è garantito che comunque prendo due elementi, questi siano confrontabili. Mentre la relazione **d'ordine è totale** se comunque prendo due elementi questi sono confrontabili.
Possiamo rappresentare entrambe attraverso un grafo dove la presenza di un arco indica la relazione di precedenza.

Arco transitivio corrisponde all'arco forward, poiché rappresentano delle scorciatoie, ma dove non aggiungo informazioni dato che c'era già un percorso orientato tra i due nodi.
Riduzione transitiva (hasse diagram): eliminazione dal grafo di tutti gli archi transitivi, riduce il numero di archi senza perdere informazioni.
Chiusura transitiva: aggiunge il massimo numero di archi transitivi
Se un grafo orientato descrive un **POSET** (partially ordered set) allora è aciclico
In un DAG esiste sempre almeno una sorgente ed esiste sempre almeno un pozzo.
Per assurdo se non esistesse un pozzo, si tornerebbe su un nodo già visitato e quindi si avrebbe un ciclo, il che andrebbe contro la struttura del DAG.

Ordinamento topologico

Vuol dire trovare un ordinamento totale (nuove comparabilità che magari prima non esistevano) che sia coerente con l'ordinamento parziale esistente.
Ordinamento tale che siano rispettate le precedenti relazioni, ad esempio:
Se A < E, A < B, C < E, D < E; A D C E B è un ordinamento totale valido, come anche A C D E B
Non esiste necessariamente un ordinamento topologico unico, ma ce ne possono essere molteplici.
Un grafo diretto ammette un ordinamento topologico se e solo se è un DAG

```
Algorithm TopologicalSort(G)
    H ← G          // Temporary copy of G
    n ← G.numVertices()
    while H is not empty do
        Let v be a vertex with no outgoing edges
        Label v ← n
        n ← n − 1
        Remove v from H
```

Effettua una copia del grafo di partenza e dato n il numero dei vertici del grafo, finché la copia non è vuota, trova un pozzo, dagli l'etichetta massima n, diminuisci l'etichetta e rimuove il pozzo dal grafo temporaneo.
L'ordine con cui rimuoviamo i pozzi descrive l'ordinamento topologico
Costo O(n+m)

Implementazione con DFS

```
Algorithm topologicalDFS(G)
    Input dag G
    Output topological ordering of G
  n ← G.numVertices()
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      topologicalDFS(G, v)
```

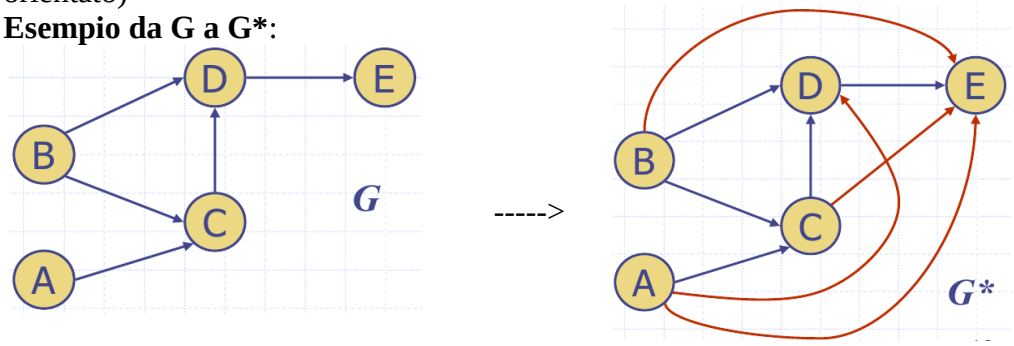
A sinistra abbiamo l'inizializzazione che poi lancerà l'ordinamento topologico.
A destra la solita DFS in cui assegnamo la marcatura di uscita n.

```
Algorithm topologicalDFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the vertices of G
    in the connected component of v
  setLabel(v, VISITED)
  for all e ∈ G.outEdges(v)
    { outgoing edges }
    w ← opposite(v,e)
    if getLabel(w) = UNEXPLORED
      { e is a discovery edge }
      topologicalDFS(G, w)
    else
      { e is a forward or cross edge }
  Label v with topological number n
  n ← n - 1
```

Chiusura transitiva indicata con *

Aggiungo un arco transitivo tra nodi che hanno un percorso di raggiungibilità che li collega, ma non hanno già un arco tra di loro. (creo una scorciatoia tra due nodi a patto che esista il percorso orientato)

Esempio da G a G*:



Gli archi che aggiungo (archi forward) per creare la chiusura transitiva non sono intesi come archi da essere percorsi nelle future DFS, quindi mi conviene etichettarli finché non finisco il processo di chiusura transitiva.

Algoritmo di Floyd-Warshall (alternativa) programmazione dinamica

Idea: Inizio numerando i vertici da 1 a n, poi considero i percorsi che considerano solamente i vertici numerati da 1 a k come vertici intermedi.

Esempio: dati due nodi i e j mi domando se ci sia un percorso che li unisce, ma che hanno dei numeri limitati, in questo caso da 1 a k-1

Pseudo-codice: L'algoritmo calcola una serie di grafi $G_0, \dots, G_n$ dove $G_0 = G$ (grafo di partenza) e G_n è il grafo della chiusura transitiva.

G_k è il k-esimo grafo generico che ha un arco orientato da i a j se il grafo di partenza G ha un percorso orientato da i a j che usa come vertici intermedi vertici numerati non superiori a k.

Si procede dunque in maniera incrementale sfruttando il passo precedente.

Tempo cubico $O(n^3)$

assumendo che la funzione per i nodi di adiacenza si svolga in tempo costante O(1), il che ci esclude le liste di adiacenza e predilige le matrici di adiacenza.

```
Algorithm FloydWarshall(G)
  Input digraph G
  Output transitive closure G* of G
  i ← 1
  for all v ∈ G.vertices()
    denote v as v_i
    i ← i + 1
  G_0 ← G
  for k ← 1 to n do
    G_k ← G_{k-1}
    for i ← 1 to n (i ≠ k) do
      for j ← 1 to n (j ≠ i, k) do
        if G_{k-1}.areAdjacent(v_i, v_k) ∧
            G_{k-1}.areAdjacent(v_k, v_j)
          if ¬G_k.areAdjacent(v_i, v_j)
            G_k.insertDirectedEdge(v_i, v_j, k)
  return G_n
```

Il primo passo numera i vertici (in alcuni casi non necessaria perché già indotta dalla struttura di dati, esempio se uso una matrice di adiacenza, essa ha già associato righe e colonne ai vertici)
Procediamo poi ad inizializzare G_0 come G e poi procediamo ad un unico ciclo for con k che varia da 1 a n.
Per inizializzare G_k sfrutto il precedente G_{k-1} per poi aggiungere archi, il che mi permette di sfruttare il lavoro fatto sul grafo precedente e che mi consente di osservare solo i vertici intermedi che tengono conto anche del k-esimo vertice.
Per trovare questi archi mancanti effettuo un doppio ciclo su tutte le coppie i,j diversi tra loro da 1 a n, se nel grafo precedente i vertici i e k sono adiacenti (se c'è un arco tra i due) e lo sono anche i vertici j e k significa che concatenando i due io ho un percorso per andare da i a j, quindi è un candidato a essere rappresentato esplicitamente nel nuovo grafo.
Se l'arco non è già presente lo aggiungo.
Alla fine del ciclo sono pronto a costruire il prossimo grafo fino ad arrivare a G*

Non c'è bisogno di usare tutti i grafi volta per volta; ci sono varie scuole di pensiero:
1) Utilizzo solo due grafi, quello che ho già costruito prima e quello che sto costruendo, quindi i precedenti non esistono più.
2) Utilizzo un grafo solo dove durante l'inserimento dell'arco aggiungo un numero che mi tiene conto della fase attuale.

Visita in ampiezza di un grafo (BFS Breadth-First Search) valido anche per gli alberi

Si basa su un sistema di visita a livelli in cui gli ultimi a essere visitati saranno i nodi più lontani dal nodo di partenza.
Come nella DFS, con una BFS si visita tutti i vertici e gli archi di un dato grafo G, si determina se G è connesso, si possono calcolare tutte le componenti connesse di G e si può calcolare una foresta ricoprente di G.
Ha sempre un costo pari ad O(n + m) con n vertici e m archi.
Utile per trovare i percorsi a cammino minimo dati due vertici in input oppure trovare un ciclo (più efficiente della DFS).

Algoritmo BFS

La fase di inizializzazione del grafo è la stessa della DFS

```
Algorithm BFS(G)
    Input graph G
    Output labeling of the edges
        and partition of the
        vertices of G
  for all u ∈ G.vertices()
   setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
   setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
   if getLabel(v) = UNEXPLORED
       BFS(G, v)
```

```
Algorithm BFS(G, s)
  L0 ← new empty sequence
  L0.addLast(s)
  setLabel(s, VISITED)
  i ← 0
  while ¬Li.isEmpty()
    Li+1 ← new empty sequence
    for all v ∈ Li.elements()
      for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
          w ← opposite(v,e)
          if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            setLabel(w, VISITED)
            Li+1.addLast(w)
          else
            setLabel(e, CROSS)
    i ← i +1
```

La BFS utilizza un insieme di code chiamate liste che sono numerate da zero fino a un certo k che dipende dalle caratteristiche del grafo. Ogni numero delle liste indica la distanza dalla sorgente.
Diametro del grafo: massima distanza tra due nodi

Costo pari a O(n+m) stesso costo della DFS.
Se non mi interessassero le liste dei nodi alle varie distanze, posso usare un'unica coda su cui aggiungo in coda i vari vertici in base all'ordine di visita, ma ciò mi fa perdere il numero di livelli. Posso ovviare a questo problema inserendo nella coda un marcatore attraverso un nodo fittizio che non viene aggiunto al grafo e che viene inserito a ogni cambio livello che nell'estrazione dalla coda indica il livello corrispondente.
Il problema di questo meccanismo è che la coda non sarà mai vuota, quindi non posso usare il precedente test, ma devo verificare se l'ultimo nodo estratto è un nodo fittizio.
I percorsi che uniscono il vertice sorgente ai vertici di una data lista hanno tutti la stessa lunghezza ed è pari al numero d'ordine di quella lista.

Cammini minimi (Shortest Paths)

Grafi pesati (weighted graphs): grafi che presentano un peso/costo (ma anche distanze, tempi ecc) associato ad un arco da pagare nel caso volessimo percorrere quell'arco. (peso negativo mi rappresenta un guadagno, ma non vanno bene tutti gli algoritmi per i grafi pesati)
Distanza fra due nodi: lunghezza del percorso più breve tra due nodi
Esistono tre diverse categorie di problemi sui percorsi minimi (in base ai parametri che si passano):
- **SP(G,u,v):** trovare il percorso a lunghezza minima nel grafo G, partendo da u ed arrivando a v
- **SSSP(G,s)**: **single source shortest path**, dato un grafo di partenza G e una sorgente s, si vogliono trovare tutti i percorsi minimi dalla sorgente s a ogni nodo destinazione possibile (esempio router).
- **APSP(G): all pears shortest path**, dato un grafo G determinare tutti i percorsi a cammino minimo per qualunque coppia di nodi che possiamo definire sull'insieme dei vertici.

Proprietà

1) Un frammento di percorso di un percorso minimo è esso stesso un percorso minimo
Dimostrazione: Dato un percorso da u a v, prendiamo in esame un percorso da p a q interno al percorso da u a v, se esistesse un percorso migliore, allora il percorso da u a v, usando quest'altro percorso diventerebbe più breve di quello precedente.
2) Dato un nordo sorgente, l'insieme dei percorsi minimi ottenuto con la SSSP può essere descritto da un albero che prende il nome di **albero dei cammini minimi o albero dei percorsi minimi (shortest path tree)**. (non ci sono cicli, poiché anche se ci fossero, i percorsi del ciclo avrebbero stessa lunghezza, quindi l'arco che crea il ciclo sarebbe superfluo)

Algoritmo basato su BFS per grafi non pesati

```
BFS(Graph G)
  forall v∈V(G) v.label=unexplored
  forall e∈E(G) e.label=unexplored
  forall v∈V(G)
    if(v.label==unexplored) BFS(G,v)

BFS(Graph G, Vertex v)
  Q = new empty queue()
  v.pred=0
  v.label=visited
  Q.insert(v)
  while(!Q.isEmpty())
    w=Q.dequeue()
    forall e∈w.incidentEdges()
      if(e,label==unexplored)
        x=e.opposite(w)
        if(x.label==unexplored)
          x.pred=w
          e.label=tree
          x.label=visited
          Q.enqueue(x)
        else e.label=cross
```

Possiamo gestire la visita con un'unica coda Q, esiste un nuovo campo nella struttura del nodo, ovvero **predecessore** (**pred**) che ci permette di costruire in maniera compatta l'albero della BFS dove il pred è ovviamente il padre del nodo.
Possiamo attraverso esso risalire alla radice ed ottenere anche il percorso minimo
Siccome stiamo lavorando su grafi non pesati, la foresta ricoprente ottenuta mi da anche il percorso a cammini minimi di una SSSP, ma solo perche su grafo non pesato, altrimenti non vale.

Algoritmo di Dijkstra (SSSP) con grafi pesati (peso degli archi non negativi)

Cerca di procedere per livelli come la DFS, ma questa volta ci sono i pesi associati.
Parte dalla sorgente e trova tutti i nodi in ordine crescente di distanza (l'ultimo nodo è quello più lontano dalla sorgente).
Ogni volta che trovo un nodo, tiene traccia di quale è stato il percorso per trovare quel nodo, il quale definisce l'albero dei percorsi minimi per quel nodo.
Assunzioni: **il grafo è connesso e i pesi degli archi non sono negativi**
Si realizza una "nuvola (cloud)" crescente che ricopre tutti i vertici. Ciò avviene associando a ciascun vertice del grafo una stima d(v) che rappresenta la distanza di quel vertice dalla sorgente s, inizializzata all'inizio a + infinito e man mano verrà sostituita dalla stima minore che si sarà ottenuta visitando il grafo.
L'algortimo prevede dunque n-1 passi dove la nuvola ingloba a ogni passo il nodo n-esimo
Almeno un nodo fuori dalla nuvola ha già la stima esatta.
Inglobare un nodo può anche migliorare le stime di altri nodi.
Il miglioramento di una stima viene chiamato **rilassamento di un arco**
d(z) = min { d(z) già calcolata, d(u) + weight(e)} ovvero se ho un nodo z fuori dalla nuvola, la distanza di d(z) corrisponde al minimo tra la d(z) già calcolata e la distanza di un nodo u, già inglobato nella nuvola, quindi con distanza già definita e minima e il peso dell'arco e che mi porta a z.

Algorithm ShortestPath(G, s):
Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.
Output: The length of a shortest path from s to v for each vertex v of G.
Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  u = value returned by Q.remove_min()
  for each vertex v adjacent to u such that v is in Q **do**
    {perform the ***relaxation*** procedure on edge (u, v)}
    if $D[u] + w(u, v) < D[v]$ **then**
      $D[v] = D[u] + w(u, v)$
      Change to $D[v]$ the key of vertex v in Q.
return the label $D[v]$ of each vertex v

Usiamo una coda di priorità Q contenente tutti i vertici del grafo G usando la stima della distanza D come chiavi e iniziamo il processo di svuotamento della coda.
Ogni volta che si estrae un elemento di fatto stiamo aggiungendo un vertice alla nuvola.
Estraiamo il minimo e lo mettiamo nella nuvola; per tutti gli adiacenti esegui la procedura di rilassamento degli archi.
Se la distanza di u che è una stima corretta poiché u è appena stato preso, più la lunghezza dell'arco tra u e v, è minore della precedente distanza, aggiorno la distanza e prendo nota del predecessore.
Infine cambio la chiave del vertice all'interno della coda Q.
Infine ritorniamo tutte le distanze di ogni vertice ottenendo cosi l'albero dei cammini minimi.

Costi: per creare la coda spendo O(n), all'interno del ciclo while, ho un costo di O(log n) per trovare il minimo nella coda Q e lo faccio n volte, quindi ho un costo di O(n log n).
Il ciclo for viene eseguito per tutti gli adiacenti, se uso liste di adiacenze mi costa grad(v), quindi nel ciclo for la somma di tutti i gradi sarà pari a 2n.
All'interno del for dunque, sommando tutti i costi ci troviamo a fare O (m log n) con m somma di tutti i gradi e log n dato dai rilassamenti.
Abbiamo dunque un costo complessivo pari a **(m + n) log n**
Se il grafo è connesso domina m, dunque il costo è **O(m log n)**

Per i grafi non connessi devo modificare il test del while.
L'algoritmo di dijkstra così strutturato (prendendo il cammino minimo) **funziona perché** ad esempio: supponiamo che ad un certo punto l'algoritmo sbaglia ed inserisce dentro la nuvola un vertice che non è quello più vicino (la stima della distanza non è ancora corretta).
Andiamo a vedere il nodo D precedente che quando è stato inserito aveva una distanza corretta, dato che stiamo considerando il nodo F attuale come il primo errore.
Dato che le stime non possono mai essere più piccole delle distanze reali, vuol dire che esiste un altro percorso con distanza minore, che però avremmo già trovato prima di aggiungere D attraverso altri rilassamenti.
Di fatto la distanza stimata dai vertici dentro la nuvola è già quella migliore perché consideriamo sempre i vertici più vicini, non esitono percorsi alternativi che vanno a spasso fuori dalla nuvola, perché lo avremmo già inglobato nella nuvola.
Di conseguenza non ci sono ulteriori possibilità per migliorare F e se non esistono evidentemente la stima è corretta.
Archi negativi potrebbero infliggere dei costi nuovi a vertici che sono già dentro la nuvola.

Algoritmo di Bellman-Ford (**percorsi minimi con archi con peso negativo, no cicli < 0**)

```
BellmanFord(Grapg G, Vertex source)
  forall v∈V(G)
    dist[v]=infinite
    pred[v]=null
  dist[source]=0
  for i from 1 to |V(G)|-1
    forall (u,v)∈E(G)
      w=weight(u,v)
      if(dist[u]+w<dist[v])
        dist[v]=dist[u]+w
        pred[v]=u
  forall (u,v)∈E(G)
    w=weight(u,v)
    if(dist[u]+w<dist[v])
      error "Graph contains negative-weight cycle"
  return dist[], pred[]
```

Cerca di fare molti più rilassamenti "brutali" rispetto a dijkstra.
Tenta tutti i possinili rilassamenti n-1 volte (i percorsi minimi saranno percorsi al più da n-1 archi)
Funzionamento: viene fornito in input un grafo G e un vertice sorgente e viene inizializzata la stima di ogni nodo ad infinito e il predecessore di ogni nodo a null. Poi alla fine posta la stima della sorgente a 0.
Poi parte il ciclo di n-1 passi che prende tutti gli archi del grafo e per ognuno se ne considera il peso w; se la stima di u (ovvero la stima dalla sorgente fino al nodo u) + il costo incrementale w necessario ad u per raggiungere v, è minore della distanza di v dalla sorgente (ovvero ad avere un

miglioramento della stima della distanza di v), allora aggiorno la stima della distanza di v e imposto u come suo predecessore.
É un passaggio a tappeto per migliorare tutte le stime, non c'è la costruzione di una nuvola.
Infine si esegue un altro for uguale al precedente perché non dovrebbe esserci nessun nuovo miglioramento della stima, se avviene vuol dire che il grafo contiene un **ciclo di peso totale negativo**.
Costo totale dell'algoritmo è pari a O(m n) dato che m sono il numero di archi per le n-1 iterazioni

Nessun algoritmo vuole un ciclo di costo totale < 0, infatti, supponiamo di voler andare da un vertice S a un vertice T minimizzando il costo, nel caso fosse presente un ciclo con un arco a peso negativo, mi converrebbe percorrerlo nuovamente nel maggior numero di volte, ma ciò mi porterebbe a percorrerlo all'infinito.

Algoritmo di Floyd-Warshall per APSP con pesi arbitrati (**inclusi pesi negativi, no cicli < 0**)
Simile a quello per la chiusura transitiva, basato sulla programmazione dinamica (migliora gradualmente provando tutti i vertici ottimali passando per soluzioni sub-ottimali) e fa uso di due matrici come strutture dati di appoggio, una chiamata **dist** (che mi raccoglie le distanze fra le coppie di vertici) e una **next** (che mi raccoglie i successori) entrambe di dimensione n x n
La matrice next contiene nella cella [i][j] l'indice del nodo successore di i che mi consentirà di arrivare a j, nello SP da i a j.

```
FloydWarshall(Graph G)
  dist = new n×n matrix of distances
    // (all infinite)
  next = new n×n matrix of vertex indices
    // (all zeros)
  forall (u,v)∈E(G)
    dist[u][v]=w(u,v)
    next[u][v]=v
  for k from 1 to n
    for i from 1 to n
      for j from 1 to n
        if(dist[i][j]>dist[i][k]+dist[k][j])
          dist[i][j]=dist[i][k]+dist[k][j]
          next[i][j]=next[i][k]


Path(Vertex u, Vertex v)
  if(next[u][v]==0) return []
  path=[u]
  while(u!=v)
    u=next[u][v]
    path.append(u)
  return path
```

Supponendo tutti i numeretti dei vertici da 1 a n (zero non mi rappresenta nessun vertice, quindi inizializzo gli elementi di next in questo modo).
Per ogni arco da u a v calcolo le distanze e il nodo successore.
Triplo ciclo for con una tripla k i j che vale per ogni nodo:
se la distanza di [i][j] nota è maggiore della distanza per andare da [i][k] + la distanza [k][j] (cerco un nodo k che mi consenta di diminuire la distanza da i a j), aggiorno la distanza e il successore.
La funzione path mi stampa il percorso
Costo totale $O(n^3)$

RIASSUNTO Shortest-Path

- Caso s-t (**sorgente, destinazione**): nel caso **non pesato uso una BFS "interrotta"**, mi fermo quando arrivo a t, **costo m+n;** nel caso **pesato la BFS non funziona, quindi userò dijkstra "interrotto"** mi fermo quando la nuvola ingloba t, costo **m log n**.
- Caso SSSP: nel caso di grafo **non pesato, uso BFS a partire dalla sorgente**, **costo m**; nel caso di grafo **presato uso Dijkstra**, **costo m log n**.
Nel caso di archi negativi Bellman Ford costo nxm
- Caso APSP: mi importa poco in questo caso se il grafo sia pesato o non pesato, ma divido tra applicare n volte dijkstra o floyd-warshall

Percosi minimi in un DAG (assumendo il grafo orientato)

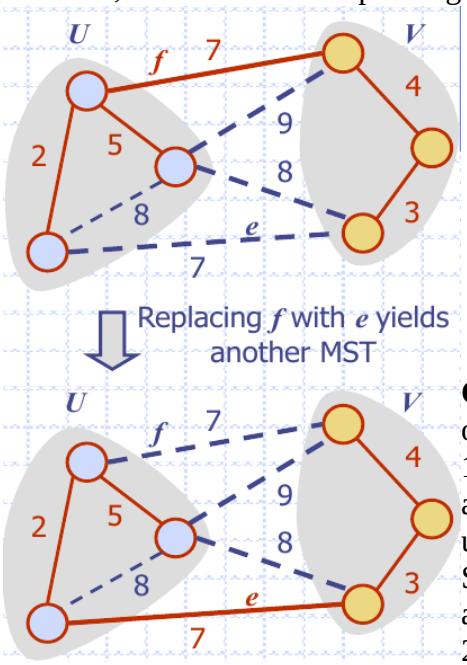
Funziona anche con gli archi a peso negativo, Costo O(n+m)

```
Algorithm DagDistances(G, s)
  for all v ∈ G.vertices()
    if v = s
      setDistance(v, 0)
    else
      setDistance(v, ∞)
  { Perform a topological sort of the vertices }
  for u ← 1 to n do    {in topological order}
    for each e ∈ G.outEdges(u)
      { relax edge e }
      z ← G.opposite(u,e)
      r ← getDistance(u) + weight(e)
      if r < getDistance(z)
        setDistance(z,r)
```

Iniziamente viene eseguito un ordinamento topologico sui vertici per poi inizializzare le distanze tra i vertici.
Prendiamo ora in considerazione i nodi in ordine topologico e per ogni arco uscente andiamo a vedere cosa c'è dall'altra parte; vediamo quanto ci costa arrivare dall'altra parte utilizzando l'arco corrente e se questa distanza migliora la stima di z, possiamo aggiornarne la stima e ricordare chi ne è il predecessore.

Minimum Spanning Trees (alberi ricoprenti minimi)

La somma dei pesi degli archi che sono presenti nello spanning tree è minima.
Nei grafi non pesati qualunque spanning tree è minimo.
Proprietà di ciclo: L'inserimento di un qualunque arco che non fa parte del mio spanning tree crea un ciclo. Tutti gli archi di questo ciclo hanno un peso minore dell'arco e appena inserito
Dimostrazione per contraddizione: se il peso di f fosse maggiore di e, allora avremmo un altro spanning tree che contiene e.
Proprietà di partizionamento: Partizioniamo i vertici del grafo in due sottoinsiemi U e V e prendiamo in considerazione un arco di peso minimo (ce ne possono essere molteplici) **e** fra le due partizioni, allora esiste uno spanning tree che contiene e.



Dimostrazione: Prendiamo uno spanning tree minimo, se per caso accade che l'albero non contiene e, consideriamo il ciclo che si forma aggiungendo l'arco e.
A causa della proprietà di ciclo, il peso degli archi di tutto il ciclo sono minori o uguali ad e.
Ma per costruzione dato che abbiamo preso e come uno degli archi minimi che si trova tra i due sottoinsiemi, allora avremo un'uguaglianza, dunque togliendo f e mettendo e abbiamo un altro spanning tree, ma con lo stesso peso.

Costruzione di uno spanning tree: esistono due filosofie opposte.
1) **Incrementale**: Parto da un nodo arbitrario e attraverso un algoritmo (tipo djikstra) calcolare l'albero che alla fine sarà uno spanning tree. (algoritmo di Prim)
Secondo questo approccio dunque la nuvola parte da un nodo e acquisice gli altri
2) **Distribuita**: Parto dai nodi, e considero la foresta ricoprente in cui non ho nessun arco, per poi aggiungere in maniera furba i migliori archi per collegare coppie di nodi.
Più si va avanti diminuisce il numero di componenti della foresta, finisco quando la foresta sarà vuota e sarà diventato un singolo albero ricoprente. (algoritmo di Kruskal)

Algoritmo di Prim-Jarnik

Prendo un nodo arbitrario e a partire da esso facciamo partire una nuvola aggiungendo nodi che sono connessi da archi che man mano che vengono aggiunti garantiscono il fatto di aggiungere il minimo incremento di peso totale possibile.
Se questa proprietà viene mantenuta per tutto il processo fino all'inserimento dell'ultimo nodo, allora avremo anche la minimalità rispetto alla somma totale alla fine del processo risultando in uno spanning tree.
Simile a dijkstra ma cambia il modo in cui viene rilassato l'arco, ad ogni vertice associamo una stima che rappresenta il più piccolo peso tra tutti gli archi che connettono l'ultimo nodo con i vertici della nuvola, **introduciamo così il concetto di distanza di un vertice dalla nuvola**
Detto ciò, procediamo ad aggiungere alla nuvola il vertice con l'etichetta di distanza più bassa possibile e aggiorniamo le distanze dei vertici fuori dalla nuvola collegati all'ultimo nodo aggiunto alla nuvola.

Algorithm PrimJarnik(G):
Input: An undirected, weighted, connected graph G with n vertices and m edges
Output: A minimum spanning tree T for G

Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T

Prendo un vertice qualsiasi dal grafo G, imposto la distanza del nodo scelto s a zero e gli altri ad infinito.
Inizializiamo l'insieme vuoto T che costituirà l'albero finale ricevendo gli archi mano a mano che l'algoritmo avanza.
Inizializiamo la coda di priorità Q con priorità la distanza.
Come con dijkstra l'algoritmo procede estraendo uno alla volta i vertici dalla coda e andiamo a collegare il vertice u a T usando un vertice e (la prima volta non ha senso, poiché non abbiamo archi, ma solo il vertice).
Per ogni arco che collega il vertice u ad un nodo v adiacente che non è ancora presente nella nuvola.
Procediamo a vedere se la distanza di v dalla nuvola T può essere migliorata con l'arco che collega u e v.
Se lo fa, sostituisco la vecchia distanza e cambio la chiave all'interno della coda insieme al nuovo valore.
Costo uguale a djikstra: costo inizialare lineare per l'inizializzazione, poi costo O(n log n) per le estrazioni, costo O(m log n) per il rilassamento degli archi.
Costo totale O((m+n) log n)
Essendo il grafo connesso il costo totale è O(m log n)

Approccio di Kruskal

Basato sull'idea di far crescere una foresta di alberi che si connettono fra loro con archi scelti opportunamente nel miglior modo possibile fino a che non si converge verso il MST.
Ordinamento fra gli archi indotto dal peso degli stessi.

Algorithm Kruskal(G):
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G

for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n - 1$ edges **do**
  $(u, v) =$ value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u, v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

Cluster (agglomerato): insieme di nodi selezionati durante l'interconnessione attraverso un albero minimo.
Inizialmente si costruisce un cluster per ogni vertice. (costo di inizializzazione pari ad n)
Inizializzo una coda di priorità Q che contiene tutti gli archi del grafo G usando come chiave il loro peso.
In virtù del fatto che possiamo costruire il bottom heap in tempo lineare
Inizializzo un insieme T che conterrà i miei archi come insieme vuoto e imposto un ciclo che mi dice che finché T non cotiene tutti gli archi necessari (n-1).
Prendo l'arco di peso minimo dalla coda di priorità e prendiamo in considerazione il cluster che contiene u e il cluster che contiene v,
Se i due cluster sono diversi (poiché potrebbe avvenire che anche se abbiamo un arco con un buon peso, esso colleghi due nodi dello stesso cluster), aggiungiamo l'arco che collega i due cluster di u e di v all'insieme T e unisci insieme i due cluster in uno solo.
Procedendo con l'algoritmo otterremo un unico cluster.
Siccome non ci sono cicli, quando abbiamo raggiunto n-1, siamo sicuri che abbiamo raggiunto un albero ricoprente e per le scelte che abbiamo fatto è anche di peso minimo, dato che per le scelte effettuate abbiamo addizionato un arco di peso minimo.
La correttezza la abbiamo per via della proprietà di partizione, considerando come partizioni i due cluster che vanno uniti.

Struttura dati per l'algortimo di Kruskal

L'algoritmo mantiene una foresta di alberi; una coda di priorità estrae gli archi incrementando il peso; un arco viene accettato solo se connette alberi distinti.
Dunque ci serve una struttura che mantenga partizioni, ad esempio una collezione disgiunta di set con operazioni quali:
makeSet(u): crea un set con solo u all'interno (costo O(1))
find(u): ritorna il set contenente u (costo O(h) con h altezza dell'albero)
union(A,B): rimpiazza i due set A e B con un set che è costituito dalla loro unione
Cerco di fare un unione collegando le radici in modo da ottenere un costo di una find il più piccolo possibile facendomi guidare da un rank (pari all'altezza). (**union-by-rank**), esiste anche **union-by-size** che cerca di mettere l'albero più piccolo dentro l'albero più grande.

Disjoint sets (collezioni disgiunte)

Rappresento ciascuno degli insieme come alberi dove i figli puntano al padre. (non conta come vengono rappresentati gli alberi, qualsiasi albero va bene)
Mettiamo in un albero gli elementi che appartengono allo stesso cluster in modo da averli in modo compatto.
Possiamo dunque disporre un array di dimensione n-1 corrispondente al vertice e all'interno della casella, l'indice del genitore. (la radice punta a se stesso)
Singleton: insieme di un solo elemento.
Path Compression: Nel caso di un albero sfavorevole, con un ramo molto alto, durante la find, procediamo a puntare i nodi di quel ramo alla radice, in modo da diminuire l'altezza dell'albero e ottimizzare le prossime find.

Implementazione efficiente

Facciamo uso di due array, uno che tiene conto di chi sono i genitori e uno che tiene conto delle altezze:

Make-set(x):

p[x] = x	ovvero il padre di x è se stesso
rank[x] = 0	l'altezza h di x è ovviamente zero essendo il nodo radice-foglia

Union(x,y):

Link(Find-set(x),Find-set(y))	x e y non devono necessariamente essere gli identificatori dei rispettivi insiemi, ma basta che appartengono a quegli insiemi

Link(x,y):

if rank[x] > rank[y]	va a confrontare le altezze dei due alberi
then p[y] = x	attacco i due sottoalberi
else p[x] = y	
if rank[x] = rank[y]	se le due alteze sono uguali
then rank[y] = rank[y] + 1	incremento l'altezza della radice

FIND-SET(x):

if x diverso da p[x] then	se non è la radice chiamo ricorsivamente find-set
p[x] = FIND-set(p[x])	tutti i nodi del ramo avranno come effetto quello di avere come padre la radice dell'albero
return p[x]	

A causa del mancato aggiornamento del rank all'interno della find, il rank dentro da link non indica esattamente l'altezza del sottoalbero, ma ne è un upper-bound (se non è esatta, ne è diminuita).

Costo: **O(m a(n))**
dove m è il numero di operazioni effettuate, tra le 3 sopra ed a(n) è l'inverso della funzione di Ackermann (funzione che cresce in maniera spaventosa, molto più dell'esponenziale), essendo l'inversa cresce in maniera lentissima e per questo il risultato di a(n) sarà sempre minore di 4

Algoritmo di Baruvka

Anch'esso come kruskal fa crescere una foresta di alberi

```
Algorithm BaruvkaMST(G)
    T ← V  {just the vertices of G}
  while T has fewer than n − 1 edges do
    for each connected component C in T do
      Let edge e be the smallest-weight edge from C to another component in T
      if e is not already in T then
        Add edge e to T
  return T
```

Inizialmente ho tutti i vertici di G dentro T ed eseguo il ciclo per tutti gli archi che per ciascuna componente connessa C mi va a sceegliere un arco e di peso minimo che va dalla componente connessa C ad un'altra componente connesssa T.
Se l'arco e non è già presente in T, lo aggiungo. (cambia dunque l'ordine con cui vengono aggiunti gli archi)
Costo complessivo: O(m log n)

Domande di esame

D: Definire gli alberi di fibonacci e spiegare perché sono interessanti. Quanti nodi hanno gli alberi di fibonacci di altezza 6? Perchè gli alberi di Fibonacci di una data altezza hanno tutti lo stesso numero di nodi?
R: Un albero di fibonacci è un particolare albero AVL in cui ogni nodo interno ha fattore di bilanciamento + o – 1. Il fatto che sia un AVL implica che sia bilanciato in altezza e che sia un albero di ricerca.
Sono interessanti perché pur se per definizione sono bilanciati, tra tutti gli alberi AVL sono quelli più vicini alla condizione di sbilanciamento proprio perché tutti i fattori di bilanciamento dei nodi interni sono +1 o -1, quindi un inserimento o un eliminazione può provocare uno sbilanciamento dell'albero.
A parità di numero di nodi, l'albero di fibonacci è l'albero AVL più alto.
A parità di altezza, l'albero di fibonacci è l'albero AVL che ha meno nodi.
Sono interessanti perché la loro forma particolare permette di studiare la relazione tra numero di nodi e altezza, e siccome sono un caso limite, se verifico che l'altezza dell'albero di fibonacci è log n, ottengo come conseguenza che anche tutti gli altri alberi AVL hanno altezza logaritmica perché tutti gli altri AVL con lo stesso numero di nodi avranno un altezza che è minore od uguale ad un albero di fibonacci.
 # nodi $n_{h+2} = n_h + n_{h+1} + 1$ (in questo caso con 6 nodi ho 33)
tabella nodi
h = 0 → n = 1
h = 1 → n = 2
h = 2 → n = 4
h = 3 → n = 7
h = 4 → n = 12
h = 5 → n = 20
h = 6 → n = 33
Gli alberi di Fibonacci di una data altezza hanno tutti lo stesso numero di nodi perché viene imposto dalla proprietà di bilanciamento. (anche se possono avere forme differenti, non cambia il numero di nodi)

D: Si consideri un albero binario completo di altezza h, con chiavi associate ai nodi e tale che la chiave associata a ogni nodo interno sia il max fra le chiavi dei due figli (tale tipologia di albero binario è detta torneo). Naturalmente la radice contiene la massima chiave dell'albero.
Quanti confronti sono necessari e sufficienti per determinare la seconda chiave in ordine di grandezza (assumere per semplicità che tutte le chiavi siano distinte). E quanti per determinare la terza? É possibile derivare un algoritmo di ordinamento? Se si, quale sarà il suo costo?
R: Devo cercare la seconda chiave più grande, dunque la chiave che ha perso con il vincitore e ha vinto tutti gli altri incontri. Confronto tutti le chiavi che si sono scontrate con la chiave max e prendo il massimo tra i perdenti. **Costo h-1**
Tra l'altezza h e il numero di nodi dell'albero c'è una relazione logaritmica. (h = log n)
Per ordinare ho un costo **n log n**, dato che faccio n volte l'altezza. (anche per l'estrazione)
Per costruire l'albero mi costa n perché ho un numero lineare di incontri.

D: Dato un grafo semplice e connesso, con n nodi ed m archi, è possibile determinare in tempo costante se esso sia acicliclo o meno? Spegare. N.B il grafo è connesso per ipotesi
R: Mi basta fare una visita, se ho un arco back ho trovato un ciclo, ma dato che mi dicono che è sicuramente connesso, conoscendo m ed n mi permettono di dire se lo è o meno perché un grafo che è sia aciclico che connesso mi rappresenta un albero.
Un albero di n nodi ha esattamente n-1 archi, dunque posso verificare in tempo costante se è aciclico o meno. É aciclico se m = n-1, se invece m > n-1 presenta dei cicli.

(Se fosse stato orientato avrei dovuto fare la visita e cercare se ci fosse il ciclo, non avrei potuto farlo in tempo costante)

D: Esiste un unico albero minimo ricoprente per il grafo in figura o può esisterne più di uno?
R: Può esistere più di un albero minimo ricoprente solo se il grafo presenta più di un arco con lo stesso peso, altrimenti no.

D: Si consideri un grafo G = (V,E) orientato completo (per ogni coppia di vertici u,v appartenenti a V, esistono entrambi gli archi diretti (u,v) e (v,u)) di n vertici. Com'è l'labero di visita di una DFS applicata a G? Si descriva la sua struttura.
R: Inizio la DFS da un nodo qualsiasi e completo la mia DFS, anche se ho un grafo pieno di archi, la mia visita comunque ignorerà gli archi che portano a nodi già visitati, quindi in ogni caso avrò un albero di visita pari ad n nodi con n – 1 archi e mi blocco solo alla fine.
Parto da un nodo, vado dove mi pare e mi blocco solo alla fine quando ho finito i nodi, ciò vuol dire che ogni albero di visita di un grafo di questo genere è composto da una sequenza lineare (una catena), quindi tutti i nodi dell'albero hanno grado 2,tranne i nodi esterni che hanno grado 1. (ovvero non si ramifica, ma degenera in una catena lineare) questo succede ogni volta che facciamo la visita di un grafo completo.

D: Dobbiamo realizzare una mappa (rispetto a chiavi intere). Una volta popolata con coppie (chiave, valore),la mappa dovrà gestire le seguenti operazioni (tra parentesi, la frequenza prevista per ciascuna operazione):
i) search(k) (40% di tutte le operazioni);
ii) range_query(k1, k2) (60% di tutte le operazioni).
Sia n il numero di coppie presenti dopo il popolamento della struttura dati. Assumendo che, tipicamente, k2 − k1 = O(log n) nelle operazioni di range query, quale struttura dati usereste tra una tabella hash e un AVL? Motivare la risposta.
R: In una tavola hash con un fattore di carico basso una ricerca mi costa O(1), mentre la range-query mi chiede in ogni caso la scansione di tutta la tabella, quindi mi costa O(n)
Nella AVL la search mi costa O(log n) e range_query mi costa O(log n + k) con k dimensione dell'ouput. Dato che mi viene detto che k2 – k1 = O(log n) la dimensione dell'output k è esattamente k = log n, quindi ho un costo di O(log n + log n) dunque un costo asintotico di O(log n)
Dato che la search viene eseguita il 40% delle volte e la range-query il 60% su m operazioni,
Avremo dunque 2/5 m search e 3/5 m range il che mi porta ad avere:
Per 2/5m search, avremo per le tavole hash un costo di 2/5 m (dato da O(1) per 2/5 m), mentre le gli AVL un costo di 2/5 m log n
Per 3/5m range, avremo per le tavole hash un costo di 3/5 m n, mentre per gli AVL un costo di 3/5 m log n
Costi totali:
Per le tavole hash: 2/5 m + 3/5 m n = m/5 (2 + 3n) per un costo totale di O(m n)
Per gli AVL: 2/5 m log n +3/5 m log n = m log n per un costo toale di O(m log n)
Per questo motivo mi conviene l'AVL

D: Mostrare come è possibile modificare un albero AVL in modo da implementare una coda di priorità con le stesse prestazioni (asintotiche) di un heap rispetto alle consuete operazioni di ricerca/ estrazione del minimo, inserimento/cancellazione di una chiave. Non si richiede la descrizione degli algoritmi che implementano le operazioni menzionate sopra, ma soltanto di descrivere chiaramente come queste ultime(e la struttura dati sottostante) vanno modificate per conseguire l'obiettivo
R: Per la insert sappiamo che in un AVL abbiamo un costo di O(log n).
Il min nell'AVL lo otteniamo in tempo costante dato che si trova nell'ultima foglia a sinistra, quindi per trovarlo ci basta scendere lungo il ramo più a sinistra in tempo O(log n).

Per potenziare l'AVL, insieme ad esso decido di considerare anche una variabile che chiamiamo min che contiene un puntatore al minimo, in questo modo posso eseguire la funzione min in tempo costante O(1), ciò però mi porta a dover fare il controllo sul min anche durante la insert, poiché nel caso di inserimento di una chiave che è più piccola dell'attuale minimo, devo aggiornare la variabile min, che però non mi costa molto perché so dove sarà il nuovo minimo (l'ultimo nodo a sinistra).
Delete-min nel min-heap costa log n, dato che dobbiamo ripristinare la condizione dell'heap, in un AVL delete-min costa O(log n) per cancellare la chiave, poi un altro O(log n) per cercare il nuovo minimo e aggiornare la variabile min, per un costo totale comunque di O(log n)
Se aggiungiamo anche una variabile max, possiamo avere contemporanemente un AVL che svolge le stesse funzionalità sia di un min-heap, che di un max-heap.

MAX-GAP IN-PLACE

```
MAX-GAP( Nodo v, int best){
    if (v == null) return (best, +infinito)
    (lg, lm) = MAX-GAP(v→left, best)
    if((v→key – lm) > lg) best = v→key – lm
    else best = lg
    return MAX-GAP(v→right, best)
}
```

lg = left gap (max-gap trovato nel sottoalbero sinistro)
lm = left max (massimo trovato nel sottoalbero sinistro)

Max-gap revisionato, il precedente è sbagliato ma non si sa mai

```
int magGap(Nodo v){
    (k,g) = _maxGap(v, + infinito, -1);
    return g;
}

_maxGap(v, maxK,maxG){
    if (v == null) return (maxK, maxG)
    (lk, lg) = _maxGap(v→left, maxK,maxG)
    if((v→key – lk) > lg) maxG = v→key – lk
    else maxG = lg
    return _macGap(v→right, v→key,maxG)
}
```

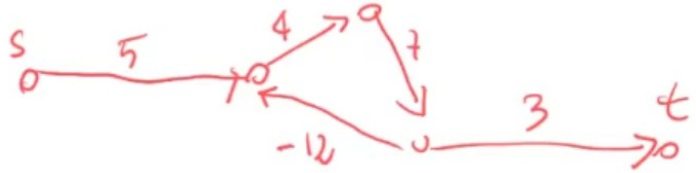
il livello ricorsivo prende in input anche la max chiave incentrata fino al momento della chiamata.

D: Si supponga di avere una tabella hash di dimensione iniziale n (ossia, la tabella può inizialmente contenere fino a n coppie (chiave, valore)). La tabella è inizialmente vuota. Si supponga che la tabella hash sia implementata particolarmente bene e che per fattori di carico non superiori a 0.5 il suo comportamento in termini di complessità delle operazioni sia ideale (questa è ovviamente una semplificazione, ma abbastanza realistica, almeno in media). Si supponga ora che siano inserite in sequenza n/2 coppie (chiave, valore) e che a seguito di un ulteriore inserimento (l'(n/2 +1)-esimo) la dimensione della tabella venga portata a 2n – 1 (in modo da mantenere non superiore a 0.5 il fattore di carico). Ciò premesso:
i) Si valuti il costo dell'operazione (n/2 +1)-esima;
ii) Si determini il costo medio di ciascun inserimento, pari a (costo totale di tutte le operazioni)/(numero di operazioni).
R: Con il raddoppiamento il fattore di carico passa da ½ ad ¼

i) Il costo per il raddoppiamento, che comporta la riallocazione dello spazio che viene gestito autonomamente mi costa O(n) e il rehashing dei precedenti valori da reinserire nella tavola che è pari ad O(n) poiché pagheremmo una costante O(1) per ognuna delle n/2 operazioni. Avremmo dunque un costo complessivo in tempo lineare.
ii) Abbiamo fatto n/2 + 1 inserimenti, quindi andando a fare la media, ovvero la valutazione del costo ammortizzato ci troviamo a dividere n/2 +1 per n/2 +1, dunque il costo ammortizzato è costante.
Ricordandoci però che il costo ammortizzato non ci da nessuna garanzia sulla singola operazione.

D: Si consideri un min-heap inizialmente vuoto. Si supponga che, in sequenza, vengano inserite nell'heap n coppie (chiave, valore), in ordine crescente rispetto all'ordinamento delle chiavi. Si valuti il costo complessivo delle operazioni di inserimento, nel caso peggiore. Giustificare la risposta. [assumere che le chiavi siano usate come priorità]
R: Se nel min-heap inseriamo chiavi in ordine crescente, durante l'inserimento se si effettua l'upHeap per riportare l'ordine, nota subito che è già tutto soddisfatto e non deve andare a fare altri confronti. Dunque le operazioni di inserimento avvengono in tempo costante perciò il costo di n operazioni di inserimento è proprio pari ad n, costo lineare.

D: Mostrare un'istanza del problema dei cammini minimi s-t (sorgente-destinazione) su un grafo pesato diretto che non ammette una soluzione a causa della presenza di (almeno) un ciclo di lunghezza negativa. Istanza significa che occorre specificare il grafo, i pesi sugli archi, i nodi sorgente e destinazione.
R: Nel caso di archi con pesi negativi, gli algoritmi (floyd-wharshall) terrebbero a percorrere infinite volte il ciclo che ha un arco con peso negativo che mi da un peso totale del ciclo < 0, e quindi l'algoritmo non si fermerebbe mai perché terrebbero a ripercorrere all'infinito il ciclo.



L'arco negativo deve essere maggiore della somma degli altri due archi, deve avere un costo totale < 0, in questo caso avrei -12 +11 = -1, andava bene anche 5+7 -13 = -1 purché il costo rimanga negativo altrimenti l'algoritmo non lo percorrerebbe all'infinito.

D: Esiste un unico albero minimo ricoprente per il grafo in figura o può esisterne più di uno?
R: Condizione necessaria affinché esistano più minimum spanning tree (MST) diversi tra loro è la presenza di archi con lo stesso peso.