Ricerca binaria ricorsiva: Supponiamo che l'array in cui cerchiamo il target sia ordinato. In ogni passaggio elimino metà array, in totale avrò $\log_2 n + 1$ chiamate della funzione binarySearch, ciascuna con costo c.

Ricorsione lineare: si ha quando il corpo del metodo contiene al massimo una invocazione ricorsiva.

Ricorsione doppia: un esempio è il calcolo dei numeri di fibonacci, il costo è molto svantaggioso rispetto ad una versione con ricorsione lineare.

Code di priorità: Nelle code classiche il primo elemento ad entrare è anche il primo ad uscire. In alcuni casi c'è da considerare anche la priorità dell'elemento. Le code di priorità sono collezioni di coppie (chiave, valore). Le operazioni principali da considerare sono: insert, removeMin, min, size, isEmpty. All'interno della coda gli elementi vengono ordinati in base alle chiavi.

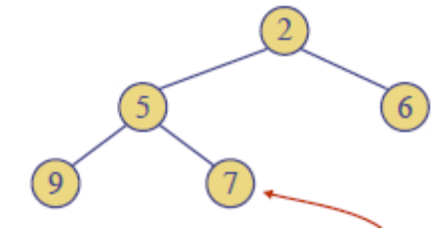
Lista ordinata: ordinare insieme S in array A attraverso una coda Q

while (!S.isEmpty) Q.insert (S.remove); while (!Q.isEmpty) A.add (Q.removeMin);

Se la coda è NON ordinata mi costerà O(n) il primo ciclo e $O(n^2)$ il secondo ciclo (insert costante e removemin/min lineare). Se la coda è ordinata mi costerà $O(n^2)$ il primo ciclo e O(n) il secondo ciclo (inserimento lineare, removemin/min costante). Apparentemente il costo totale sarà uguale in entrambi i casi. In realtà nel primo caso per eseguire removeMin la funzione dovrà sempre controllare tutta la coda; nel secondo caso per eseguire la insert la funzione controllerà nel caso peggiore tutti gli elementi ma nella realtà sarà più rapida.

Consideriamo la seguente equazione ricorsiva di costo: $C(n)<=c+C(n/2) \rightarrow C(n)=O(\log_2 n)$. Se la associamo alla ricerca di un elemento in un array, il costo sarà logaritmico nella dimensione dell'input (considerata come il numero di elementi di un array). Se la associamo a una funzione di elevamento a potenza con n parametro di elevazione, il costo sarà lineare nella dimensione dell'input (considerata come il numero di bit necessari a salvare in memoria l'input, ovvero l'esponente).

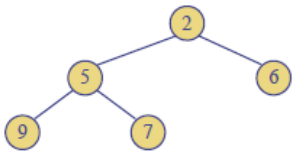
Heap:

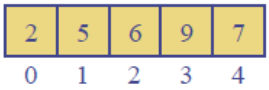


Viene utilizzato per rappresentare code di priorità. Contiene elementi del tipo (chiave, valore). È un albero binario con chiavi associate ai vertici. In ogni nodo, eccetto la radice, la chiave associata al nodo è <= di quelle associate ai figli. L'altezza sarà uguale alla parte intera del $\log_2 n$, dove n è il numero di elementi dell'heap.

Inserimento: si inserisce l'elemento nell'ultimo posto disponibile e si riordina l'heap. Mantenendo un riferimento al primo spazio libero, l'operazione di inserimento avrà sempre costo costante. Per riordinare l'heap si scambia l'elemento inserito con il padre se è maggiore e si continua così (upheap). Il costo massimo è logn.

RemoveMin: in un heap l'elemento con chiave minima si trova sempre nella radice. Viene rimosso e sostituito con l'ultimo elemento dell'heap. Per riordinare si scambia la nuova radice con il minore fra i due figli e si continua così (downheap). Il costo massimo è logn.

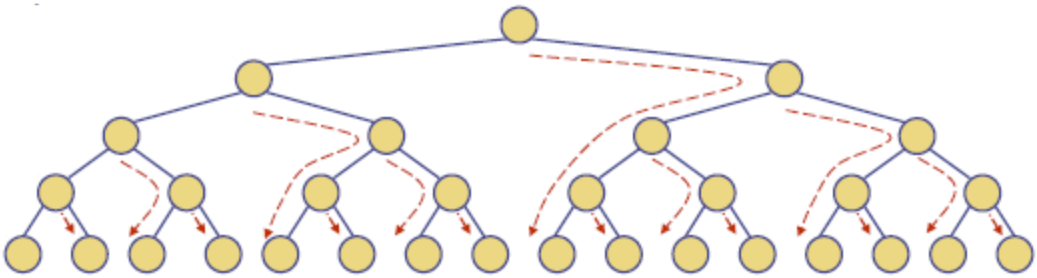




Utilizzando una rappresentazione con array si mette il figlio sx in posizione 2i+1 e il figlio destro in 2i+2, il genitore sarà in (i-1)/2. La radice ha indice 0, l'ultimo figlio è nell'ultima posizione nell'array, i figli sinistri sono in posizione dispari e i figli destri in posizione pari.

Heapsort: algoritmo di ordinamento attraverso un heap. Si inseriscono tutti gli elementi da ordinare in un heap (costo nlogn) e si rimuovono con removeMin (costo nlogn). In totale avrà costo nlogn

Heapify: creazione bottom-up di un heap. Creiamo degli heap (anche composti da un solo elemento) e li uniamo a coppie aggiungendo contemporaneamente un elemento come radice. Per rispettare la proprietà di heap eseguiamo downheap. Il costo totale sarà O(n). Infatti il costo è dato dalla somma di tutti i downheap che avverranno. Nel caso peggiore i downheap salteranno un livello quando unisco due singoli, due livelli quando unisco due triplette, ecc… In totale il numero di esecuzioni di downheap sarà minore del numero di archi nell'heap (che è numero di nodi – 1).



Il problema di rappresentare strutture dati tramite array ha degli svantaggi: primo fra tutti è la difficoltà nell'aumentarne le dimensioni nel caso si riempia. In questo caso bisogna creare un nuovo array e copiarci tutti gli elementi. Si usa una tecnica che consiste nel raddoppiare le dimensioni ogni volta che l'array si riempie. Facciamo l'analisi dei costi: nel caso ci siano n-1 elementi, l'inserimento di un elemento avrà costo O(n) perché sarà necessario copiarli tutti nel nuovo array inizializzato. Questo caso però si verifica raramente, per cui facciamo un'analisi dei costi ammortizzata (si considera una sequenza di operazioni e si divide il costo totale per il numero di operazioni totali). Consideriamo di voler inserire n elementi in un array inizialmente vuoto. In totale ci troveremo a fare $\log_2 n$ raddoppi ogni 2^{i-1} inserimenti. Consideriamo gli inserimenti fra il 2^i-esimo raddoppio e il 2^{i+1}-esimo escluso. Avremo 2^i inserimenti in totale di cui uno con costo 2^{i+1} e i restanti 2^i-1 ciascuno con costo logaritmico nella dimensione attuale dell'array che consideriamo in ogni caso al massimo 2^{i+1} ($\log_2 2^{i+1}=i+1$). Il costo complessivo nell'i-esimo intervallo sarà $2^{i+1}+2^i(i+1)$ (costo raddoppio + 2^i volte costo inserimento normale). L'ultimo intervallo avrà lunghezza $n-2^{\log n}$ per cui il costo totale sarà O(nlogn). Considerando che abbiamo fatto n inserimenti possiamo dividere il costo trovato ed otteniamo O(logn) come costo medio logaritmico del singolo inserimento.

Per sostituire la chiave di un elemento all'interno dell'heap si può usare la funzione replace che ha costo O(logn) dato dalle operazioni di upheap e downheap.

Se creo un heap inserendo n chiavi crescenti, il costo delle operazioni di inserimento sarà sempre costante e quindi il costo totale sarà O(n). Inserendo invece n chiavi decrescenti dovrò eseguire ogni volta upheap con costo massimo O(logn) e in totale il costo degli n inserimenti sarà O(nlogn).

Ordinamento:

Insertion sort: ordinamento tramite code di priorità rappresentate da liste ordinate $O(n^2)$

Selection sort: ordinamento tramite code di priorità rappresentate da liste non ordinate $O(n^2)$

Bubble sort

Quick sort: è un allgoritmo che opera scelte casuali. Sceglie un elemento pivot a caso e divide gli elementi in tre sottoinsiemi, gli elementi minori uguali e maggiori del pivot. Ricorsivamente ordina i due sottoinsiemi minori e maggiori e infine unisce tutto. L'esecuzione è rappresentata da un albero binario. Essendo la scelta del pivot casuale possiamo ottenere ad ogni iterazione casi buoni o cattivi. Definiamo caso buono quello in cui le dimensioni di minore e maggiore sono minori di 3s/4, caso cattivo quello in cui minore o maggiore ha hanno dimensione maggiore di 3s/4. Una attivazione è buona con probabilità 1/2. Posso allora dividere l'array in pivot buoni e cattivi noto come avrò una possibilità di 1⁄2 di ottenere un pivot buono. Quindi per un nodo di profondità i ci aspettiamo che i suoi i/2 antenati abbiano chiamato pivot buoni e la dimensione della sequenza sarà al più $(3⁄4)^{i/2}n$ quindi per un nodo di profondità $2\log_{4/3}n$ avrò una dimensione attesa del nodo di 1, cioè ho profondità di O(logn) e visto che ad ogni livello il lavoro per trovare tutti i nodi è O(n) avrò un costo complessivo di O(nlogn). Il caso peggiore avrà costo totale $O(n^2)$, il caso medio O(nlogn). Per ottenere il caso migliore dovremmo scegliere come pivot l'elemento medio. Per avvicinarci a questo caso possiamo prendere 3 o 5 elementi a caso e scegliere come pivot l'elemento mediano fra questi.

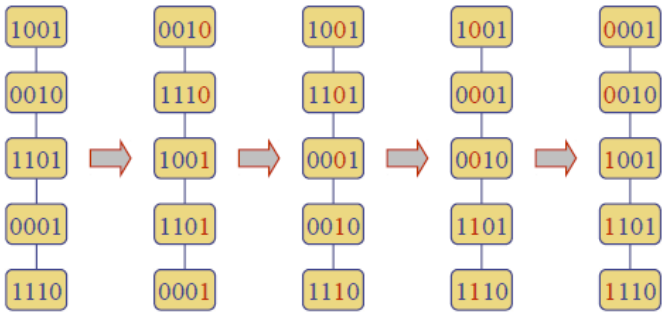
Merge sort iterativo: il costo complessivo è O(nlogn). L'idea fondamentale del merge sort è che l'operazione di fusione di due array ordinati ha costo lineare rispetto alla dimensione degli array.

Heap sort in-place: non si crea un heap ma si ordina l'array per far sì che sia un maxheap. Poi scambio la radice con l'ultimo elemento, dimentico l'ultimo elemento ed eseguo downheap. Itero questa procedura finchè la size dell'heap rimasto non è pari a uno. Il costo totale dell'operazione sarà O(nlogn)

Gli algoritmi basati sul confronto hanno costo minimo O(nlogn) ma esistono altri algoritmi di ordinamento con costo minore. Ad esempio usando un heap la complessità è O(nlogn).

Gli algoritmi in-place non utilizzano memoria ausiliaria o ulteriori strutture dati.

Bucket sort: non è basato sul confronto. Creiamo un array con abbastanza spazio per contenere tutte le chiavi possibili (consideriamo noto l'intervallo in cui sono le chiavi). Consideriamo chiavi nell'intervallo (0,9) e mettiamo ogni chiave nel suo indice del bucket (array creato prima). Se abbiamo delle coppie con chiave uguale le inseriamo in una lista con riferimento alla testa nel bucket. Svuotando il bucket in ordine ho ordinato il mio array. Il costo di questo algoritmo è O(N+n) dove N è il numero di possibili chiavi e n è il numero di elementi da ordinare. Se N<n ho O(n).



Radix sort: è un algoritmo di ordinamento non basato sul confronto. Si applica quando la chiave può essere vista come tupla di chiavi ordinabili. Ad esempio una serie di caratteri composti da 8 bit è ordinabile invocando 8 volte bucketsort su ogni indice della sequenza di bit. Se 2 elementi hanno l'i-esimo bit uguale mantengo l'ordinamento che avevano. Il costo sarà O(n+N) per ogni bucketsort moltiplicato per la grandezza della tupla, in totale O(d(n+N)).

Mappe

Sono un dato astratto che rappresenta coppie (chiave, valore). La ricerca di una chiave è l'operazione fondamentale e vogliamo una struttura efficiente nel compiere questa operazione. La ricerca sarà poi funzionale all'inserimento e alla cancellazione. Supponiamo che non esistano 2 elementi con la stessa chiave. Operazioni fondamentali: get(k), put(k,v), remove(k), size, isEmpty. Iteratori: entryset (coppie), keyset (chiavi), values (valori).

Realizzazione con lista ordinata: get, put (controlla se c'è già la chiave) e remove avranno costo O(n). È efficiente per mappe di piccole dimensioni.

Un insieme è una lista non ordinata di elementi priva di duplicati. L'operazione fondamentale è la verifica di appartenenza di un elemento. Un multiinsieme permette duplicati. Operazioni: add, remove, contains, iterator hanno costo O(n); addall (unione), retainall (intersezione), removeall (sottrazione).

Una multimappa permette di associare più valori ad un'unica chiave. Viene implementata come una mappa in cui ogni chiave ha come valore un insieme di valori ad essa associati nella multimappa. Operazioni: get(k), put(k,v), remove(k,v), removeAll(k), size (restituisce il totale dei valori presenti).

Le mappe si possono rappresentare con tabelle Hash. Utilizzando un array a cui lego ad ogni indice la sua chiave rischio di avere molti buchi; con una tabella hash posso ridurre l'intervallo di chiavi e mappare ogni chiave originale sul nuovo intervallo più piccolo. Bisogna cercare di sparpagliare più possibile la chiavi rimappate per evitare che vadano a finire sullo stesso nuovo indice. Generalmente le funzioni hash si avvalgono del modulo/resto h(x)=x%N dove N è la grandezza del nuovo array. Questa funzione restituirà valori sempre minori di N. h(x) si chiama valore hash della chiave x.

Una tabella Hash è composta da una funzione hash h e un array (detto tabella) di dimensione N. L'obiettivo è memorizzare la coppia (k,v) in posizione h(k) nell'array. I problemi sorgono per chiavi non intere e per le eventuali collisioni (la funzione h potrebbe restituire lo stesso valore con due diverse chiavi in input). Per gestire chiavi di qualsiasi tipo (ad esempio alfanumeriche) si trasforma la chiavi in un intero (hashcode) con una funzione h1 sfruttando il fatto che qualsiasi elemento è già codificato in binario per permetterne la memorizzazione. Successivamente si utilizza una funzione h2 di compressione. h(x)=h2(h1(x)). Si cerca di ottenere quasi iniettività, ovvero chiavi distinte mappate su interi distinti.

Un esempio riguardante stringhe come chiavi. Si possono sommare i codici ASCII dei caratteri che compongono la stringa per poi passarli alla funzione di compressione. Il problema è che gli anagrammi di una parola avranno tutti lo stesso hashcode e quindi saranno mappati con lo stesso indice dando luogo a collisioni. Per questo si utilizzano metodi più complessi come il metodo dei polinomi: si partiziona la chiave in componenti con lunghezza fissa di bit (8,16,32,…) e si considerano come interi (a0, a1, …); successivamente si calcola il polinomio $p(z)=a0+a1z+a2z^2+\dots$ dopo aver fissato un valore per z e trascurando l'overflow. Abbiamo descritto un metodo per il calcolo di hashcode che risulta essere molto

efficiente, su 50000 termini inglesi va incontro a sole 6 collisioni utilizzando z=33. Un altro metodo è quello degli shift ciclici: si somma una rappresentazione a 32 bit con se stessa shiftata di qualche bit (1000101→0010110).

Ora che abbiamo definito metodi per calcolare hashcode abbiamo bisogno di funzioni di compressione. La funzione modulo (resto) h2(x)=xmodN rende la dimensione della tabella pari a N. Scegliendo N numero primo si ottengono migliori risultati nelle collisioni. MAD (multiply, add, divide) è una funzione di compressione h2(x)=((ax+b)modp)modN, p è un numero primo p>N, a<p e b<p sono scelti indipendentemente e a caso. Garantisce poca collisione. Anche h2(x)=(ax+b)modN dà risultati accettabili.

Gestione delle collisioni. Ho collisione se due chiavi hanno lo stesso hashcode o se la funzione di compressione mappa due hashcode nello stesso indice della tabella. Ci sono più modalità di gestire collisioni:

- Si può associare una lista ad ogni elemento dell'array (memoria esterna alla tabella), le liste di trabocco. Con poche collisioni avrò un costo quasi costante, con molte collisioni avrò un costo alto per cercare l'elemento giusto all'interno di una lista di trabocco.
- Si cerca un'altra posizione libera (memoria interna alla tabella), Indirizzamento aperto
    - Scansione lineare (linear probing): si inserisce l'elemento che genera collisione nella prima posizione libera successiva. Genera agglomerazione primaria (sequenza di celle occupate). In questo caso nella ricerca posso trovare una coppia con chiave k, una cella vuota o posso controllare tutto l'array senza trovare posizioni vuote. Quando cerco un elemento devo controllare dalla posizione che dovrebbe avere a tutte le celle successive fino a che non ne trovo una vuota. Quando rimuovo un elemento, lo sostituisco con un valore standard deciso in partenza (Defunct), questo è necessario perché se inserisco un elemento in posizione 2 e poi devo mettercene un altro ma la 2 è già occupata e la 3 anche allora lo metto in 4; se poi rimuovo quello in posizione 3 avrò un buco in mezzo e rischio di fermarmi nella ricerca. Se la tabella è piena devo aumentarne la dimensione.
    - Scansione quadratica: se una cella è piena si mette nella prima libera generata da (h(k)+f(i))modN per i che va da 1 a N-1 e dove $f(i)=i^2$. Può dar luogo ad agglomerazione secondaria. Con N primo e almeno metà celle vuote ho la garanzia di trovare un posto libero.
    - Hashing doppio (double hashing): si usa una seconda funzione di hash d(k). La coppia si inserisce nella prima cella disponibile nelle posizioni (i+jd(k))modN con j=0,…,N-1 ho la garanzia che vengano esaminate tutte le celle. Generalmente si sceglie d(k)=q-kmodq con q<N e primo

Fattore di carico (load factor): è il tasso di riempimento della tabella ed è pari al numero di chiavi presenti diviso N. Bisogna tenerlo sicuramente <1. Inoltre per mantenere prestazioni simili a quelle di una tabella ideale è meglio se <0.5 per scansione lineare e hashing doppio, <0.75/0.9 con liste di trabocco.

Il rehashing è l'aumento delle dimensioni di una tabella hash. Con liste di trabocco non è necessario anche se le prestazioni scendono nettamente per fattori di carico >1. Con scansione lineare è necessario se la tabella si riempie. Gli hashcode delle chiavi non vanno ricalcolati, quella che va riapplicata è la funzione di compressione per tenere conto del nuovo N. Effettuare il rehashing sparpaglia nuovamente gli elementi nella tabella hash. Ha costo lineare nella dimensione della tabella hash.

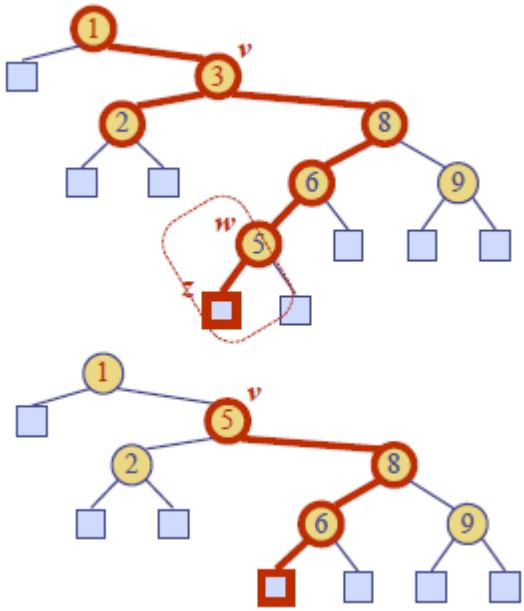
Ogni chiave ha probabilità 1/N di essere mappata in una posizione dell'array. Supponiamo di inserire k1,…,kn n chiavi di seguito e che la gestione delle collisioni sia con liste di trabocco. Vogliamo calcolare la lunghezza media (valore atteso) della lista di trabocco in posizione i. Fissiamo una posizione i e consideriamo l'inserimento j-esimo, gli associamo una variabile binaria xj =1 se la j-esima coppia è in posizione i, =0 altrimenti. X è il numero di chiavi assegnate alla posizione i della tabella, X=sommatoria con j che va da 1 a n di xj. Il valore atteso di X (lunghezza di una lista di trabocco) è E=n/N. Questo valore può essere sempre <1 se la tabella viene ridimensionata ogni volta che il fattore di carico supera un certo valore (ad esempio 0.5).

Supponiamo una tabella di dimensione iniziale 7, fattore di carico massimo 0.5 e rehashing con raddoppio. Consideriamo n inserimenti che causano l raddoppi, dopo n inserimenti la dimensione della tabella è $S_l=7*2^l$ e considerando un fattore di carico massimo 0.5 abbiamo $S_l>=2n$ e $S_{l-1}<2n$. $S_l=7*2^l$ → $7*2^l>=2n$ → l=parte intera inferiore di $\log_2(2n/7)$. Abbiamo ottenuto il numero di raddoppi dovuto all'inserimento di n elementi. Il costo dell'i-esimo raddoppio è pari all'allocazione di una nuova tabella + la copia di ogni chiave dopo aver applicato nuovamente la funzione di compressione. Il costo complessivo di l raddoppi è 8cn-7c, ovvero O(n) e quindi lineare nel numero di inserimenti. Di conseguenza il costo medio di rehashing imputabile a ciascun inserimento è O(1).

Inserendo n chiavi in una tabella di dimensione N avrò al massimo n-1 collisioni (il primo inserimento andrà sicuramente liscio) e minimo n-N collisioni (ovvero tutte quelle necessariamente accadute dopo aver riempito l'array).

Mappe ordinate. Operazioni: get(k), put(k,v), remove(k), subMap(k1,k2) o rangeQuery(k1,k2). Potremmo usare un array su cui applicare ricerca binaria con costo logaritmico. Per eseguire subMap si può fare due ricerche binarie e ottenuti gli indici delle due chiavi delimitatrici restituire la porzione di array richiesta. Costo operazioni: get logaritmico, put lineare se assente o logaritmico se già presente, remove lineare, submap O(logn+S) dove S è il numero di chiavi nell'intervallo. Utilizzando questa struttura si ha una lista che occupa spazio O(n), una add sarà costosa per mettere l'elemento al posto giusto, un Merge (per unione o intersezione) avrà costo lineare nella somma delle dimensioni degli insiemi da unire.

Alberi binari di ricerca si usano per rappresentare mappe ordinate. Con liste avevamo add/remove con costo lineare e ricerca con costo logaritmico. In un albero binario di ricerca BST ho chiavi sottoalbero sinistro <= chiave radice <= chiavi sottoalbero destro. Si può fare una visita simmetrica in-order e corrisponderà a una visita con chiavi in ordine crescente (prima sottoalbero sinistro, poi radice e infine sottoalbero destro, il tutto ricorsivamente). Come operazioni fondamentali consideriamo ricerca, inserimento, cancellazione e subMap. Per la ricerca si cerca nel sottoalbero destro o sinistro se la chiave cercata è maggiore o minore della radice, se si arriva ad una foglia la ricerca non ha prodotto risultati. Per la put si aggiunge appena si trova una foglia. La remove è più complicata: se l'elemento rimosso ha come uno dei due figli una foglia allora posso semplicemente eliminarlo e sostituirlo con l'altro figlio; se invece ha due figli devo trovare il nodo con due foglie come figli più a sinistra nel sottoalbero destro dell'elemento rimosso e metterlo al suo posto. Il costo della remove dipende dall'altezza dell'albero ed è variabile fra O(n) e O(logn) in base al bilanciamento dell'albero. La range query ha complessità O(h+s) dove h è l'altezza e s è il numero di chiavi nell'intervallo k1, k2. Per avere efficienza in tutte queste operazioni c'è bisogno di alberi ben bilanciati e quindi di un'altezza più possibile vicina al minimo teorico (ovvero $\log_2 n$).

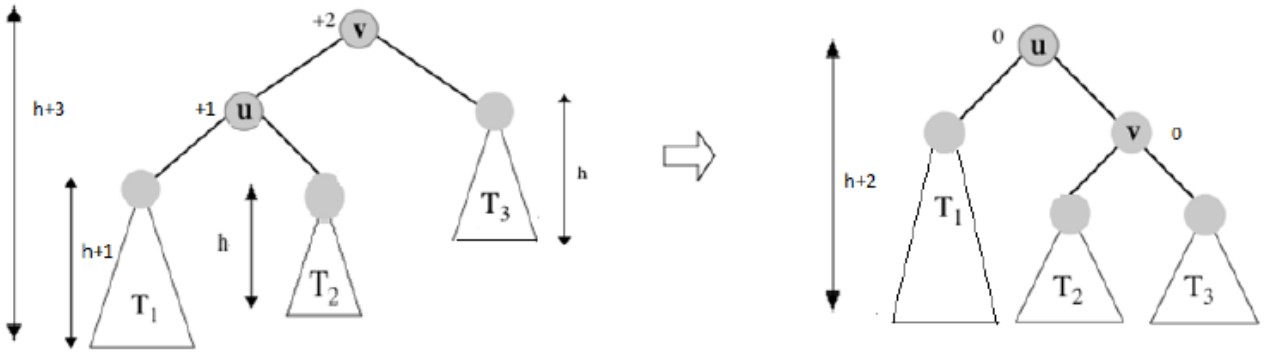


Le funzioni max e min sono molto semplici perché vanno semplicemente sempre nel sottoalbero destro o sempre nel sinistro fino a trovare una foglia. Definiamo predecessore di un nodo il massimo valore nel sottoalbero sinistro e successore il minimo valore nel sottoalbero destro. Se non ho figli sinistri devo risalire al padre se sono figlio sx fino a che non sono figlio destro e restituisco il genitore. Tutte queste operazioni hanno costo O(h) dove h è l'altezza dell'albero (O(n) nel caso peggiore, O(logn) quando ho un albero perfettamente bilanciato). Il nostro scopo è avere un albero di altezza sempre ottimale, ovvero sempre ben bilanciato, per avere costo di tutte queste operazioni sempre logaritmico.

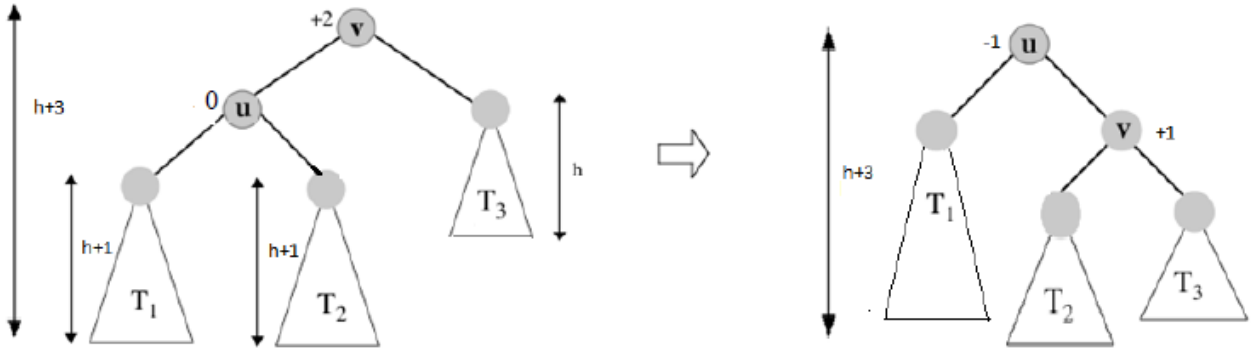
Alberi AVL. Il fattore di bilanciamento β di un nodo è h(sx)-h(dx). Un albero bilanciato in altezza ha $|\beta|<=1$ per ogni nodo. β viene mantenuto come informazione nel record relativo a ciascun nodo. Se un albero è AVL la sua altezza è $O(\log_2 n)$. Un albero in cui tutti i β sono =0 ha il massimo dei nodi per quella altezza. Gli alberi AVL più sbilanciati sono gli alberi di fibonacci che hanno il numero minimo di nodi per la propria altezza, ovvero il massimo dell'altezza per il proprio numero di nodi. Si parte da altezza 0 (solo un nodo), poi si crea albero di altezza 1 con figlio sinistro l'albero di h=0, poi si crea albero di altezza 2 mettendo figlio sinistro l'albero di h=1 e figlio destro l'albero di h=0, poi si crea albero di altezza 3 mettendo figlio sinistro l'albero di h=2 e figlio destro l'albero di h=1 e così via. $n(h)>=c*a^h$ con a=radice(2) e c=1/2 → il numero di nodi è esponenziale rispetto all'altezza, di conseguenza l'altezza è comunque logaritmica rispetto al numero di nodi. Un albero AVL con n nodi ha altezza h=O(logn), dobbiamo però ripensare insert e remove per far mantenere la condizione di bilanciamento tipica degli AVL.

Aggiungendo un elemento in un albero AVL cambiano i fattori di bilanciamento di +-1 dei nodi lungo il cammino dalla radice al nodo aggiunto. Manteniamo il bilanciamento attraverso rotazioni. Se ho radice v (sx=u dx=t3) e u (sx=t1 dx=t2) diventa con una rotazione destra radice u (sx=t1 dx=v) e v (sx=t2 dx=t3). Una rotazione sinistra riporta alla situazione iniziale. Questa rotazione mantiene le proprietà di BST e ha costo costante O(1). Sia il nodo v di profondità massima con fattore di bilanciamento β=+-2, il sottoalbero che sbilancia (t) può essere sinistro del figlio sinistro di v SS o dx del figlio dx DD (simmetrici fra di loro); può essere sx del figlio dx SD o dx del figlio sx di v DS (simmetrici fra di loro).

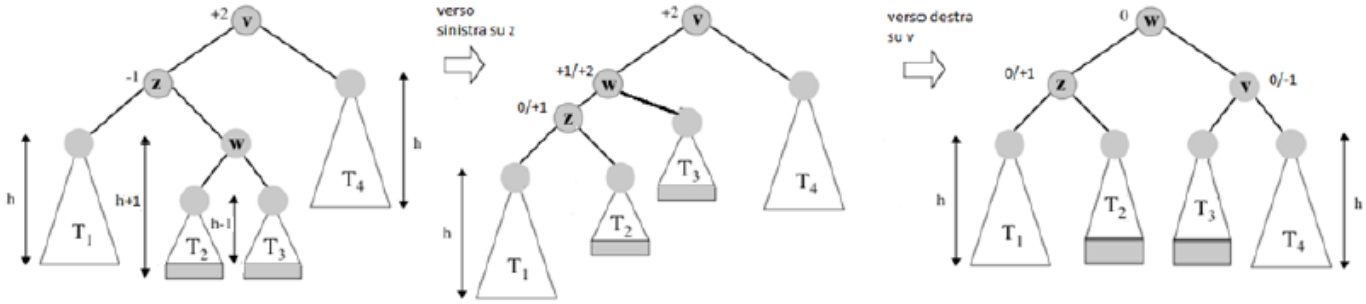
- Caso SS: faccio una rotazione a destra su v
    - Se T2 ha altezza h ho risolto e ridotto l'altezza (caso possibile con inserimento o rimozione)



    - Se T2 ha altezza h+1 ho risolto e l'altezza è rimasta uguale (caso possibile con rimozione)



- Caso DD è simmetrico a SS
    - Caso SD: facciamo una rotazione sx in z, a questo punto sono nel caso SS e so come procedere, ovvero facendo una rotazione destra su v (caso possibile con inserimento o rimozione)



Come strutturiamo una insert: creo nodo, lo inserisco come nei BST semplici, ricalcolo i fattori di bilanciamento dei nodi nel cammino dalla radice al nodo aggiunto, eseguo opportune rotazioni sul nodo più profondo con fattore di bilanciamento +-2 (lo chiamo nodo critico). È sempre sufficiente un solo ribilanciamento per inserimento.

Come strutturiamo una delete: cancella nodo come in BST, ricalcola fattori di bilanciamento del padre del nodo eliminato fisicamente ed esegui eventuali ribilanciamenti. Itero questa procedura: se l'altezza del ribilanciato è uguale a quella pre-eliminazione allora termino, se è diminuita vado nel padre del sottoalbero ribilanciato, calcolo il suo fattore di bilanciamento e applico opportuni ribilanciamenti.

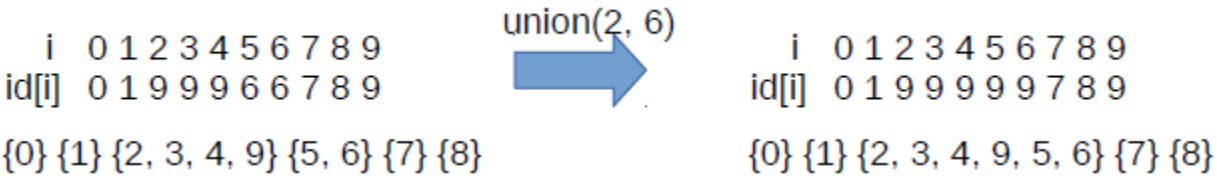
Tutte queste operazioni hanno costo O(logn) perché nella delete faccio al massimo tante rotazioni quanta è l'altezza dell'albero e ciascuna rotazione ha costo costante. Bisogna sempre tener conto del fatto che abbiamo considerato di conoscere i fattori di bilanciamento in tempo costante e che il calcolo dei fattori di bilanciamento di tutti i nodi di un percorso da radice a foglia abbia costo O(logn). Per mantenere queste premesse bisogna avere un campo che associ ad ogni nodo l'altezza del sottoalbero che ha lui stesso come radice.

Tipi di visite:

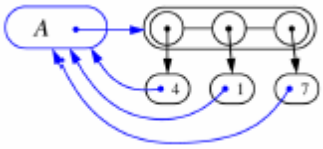
- In-order restituisce chiavi crescenti (sx, visita radice, dx)
- Post-order si usa per le espressioni aritmetiche (sx, dx, visita radice)
- Visita di Eulero è un caso generale del pre-ordine e post-ordine in cui ogni nodo è visitato tre volte: inizialmente da sopra e poi da sotto sia da destra che da sinistra

Union-Find

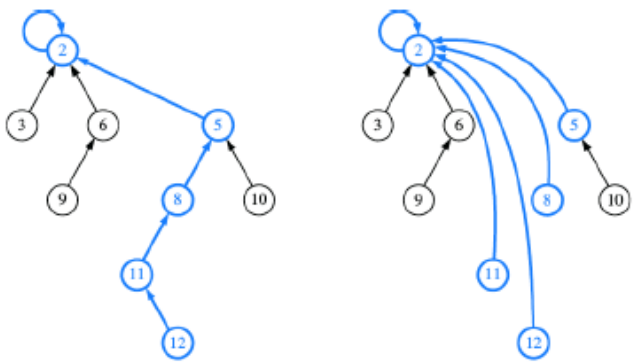
Abbiamo insiemi di oggetti in partizioni disgiunte. Le operazioni fondamentali sono: makeSet(x) o makeCluster(x) crea l'insieme, union(a,b) sostituisce a quegli insiemi la loro unione, find(x) restituisce il leader dell'insieme contenente x. Ci concentriamo su una rappresentazione basata su array. Possiamo ad esempio mantenere due array della stessa lunghezza: in uno teniamo gli elementi e nell'altro, negli indici corrispondenti a ciascun elemento, mettiamo un id che può essere ad esempio uno degli elementi del sottoinsieme a cui appartiene (leader). La find ha costo costante, makeCluster ha costo costante (deve solo aggiungere l'elemento in entrambi gli array, sarà leader di se stesso), union ha costo lineare (deve scorrere tutti gli id e modificare quelli di uno dei due sottoinsiemi che vogliamo unire impostando il leader dell'altro sottoinsieme). Svantaggi degli array sono spazio e ridimensionamento, oltre che eventuali buchi se mappiamo ogni intero dell'insieme nel corrispondente indice dell'array.



Si può usare una implementazione collegata in cui abbiamo una testa della lista (A) per ogni sottoinsieme che contiene anche il numero di elementi del sottoinsieme. MakeCluster e find hanno costo costante, union(a,b) deve modificare i riferimenti di ogni elemento del sottoinsieme più piccolo con costo pari a min(a.size,b.size). Facendo un'analisi di costo ammortizzato otteniamo che una serie di k makeCluster, find e union ha costo O(k + nlogn) se inizio con una partizione vuota. Le k operazioni coinvolgono al più n elementi k>=n. Il confronto fra a.size e b.size ha costo c, la modifica della size del leader della lista più grande ha costo c, la modifica di p.leader per ogni elemento della lista minore ha costo c*min(a.size, b.size). Il costo per ora è c*k (le prime due operazioni hanno costo c moltiplicato per k esecuzioni) + sommatoria da i a l di min(a(i).size, b(i).size) dove a(i) e b(i) sono i due insiemi coinvolti nella union e l è il numero totale di operazioni di union. Chiamo mj il numero totale di volte che il riferimento a leader di un singolo elemento è stato modificato. La sommatoria da i a l di min(a(i).size, b(i).size) diventa quindi uguale alla sommatoria su tutti gli elementi dell'insieme di mj. Ogni volta che aggiorno il leader la lista è almeno raddoppiata, quando faccio makecluster la lista ha inizialmente size=1. n >= dimensione lista in cui sta elemento j >= 2^{mj} quindi mj <= $\log_2 n$ per ogni elemento j. Il costo totale sarà quindi minore di c*k + sommatoria di mj <= c*k + n*$\log_2 n$, da cui otteniamo il costo O(k+nlogn) indicato sopra.



Implementazione con alberi: ogni elemento è un nodo dell'albero che ha come campi: element (valore), size (dimensione dell'albero a cui appartiene), parent (riferimento al genitore). Nodi dello stesso albero corrispondono allo stesso sottoinsieme. Il leader è la radice. Makecluster ha costo O(1), find ha costo proporzionale al cammino fino alla radice (che è anche il leader) e nel caso perrgiore è O(n), union ha costo O(1) ma comprende prima una find. I leader hanno come riferimento al genitore un riferimento a se stessi, quando si fa union questo riferimento va ad un elemento dell'altro sottoinsieme a cui si lega. Si può però ottimizzare la union: union-by-size (si aggiorna il leader dell'albero minore) e compressione (aggiorno il genitore di ogni nodo visitato fino al leader, in questo modo avrò alberi più bassi). Si può dimostrare che con queste due ottimizzazioni il costo complessivo di k operazioni è O(k log*n) dove log* è il minimo x tale che log(log…(logn))<2 con x volte log. Se n è 16, log*n=3 perché log(log(log16))=1<2 e ho usato 3 log. Non è una funzione costante ma cresce molto più lentamente di un logaritmo, è quasi costante.

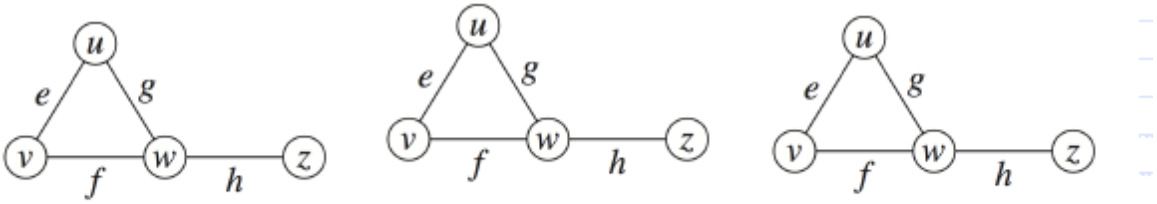


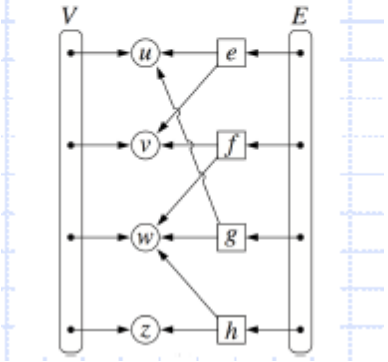
Esercizi:

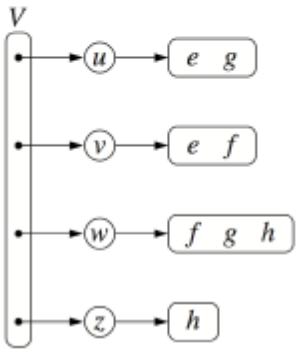
1. Mostrare come usare un albero AVL per ordinare per chiave n coppie (k,v) in tempo O(nlogn): è necessario popolare inizialmente l'albero e il costo dell'inserimento in un AVL di un elemento è logn, per n elementi otteniamo O(nlogn). Il costo della visita in-order è O(n). Combinando i due costi ottengo O(nlogn)
2. Dimostrare che non è possibile inserire in successione n chiavi in un AVL inizialmente vuoto con costo complessivo O(nlogn) usando operazioni su chiavi basate sul confronto. Se fosse possibile AVLsort avrebbe costo O(nlogn) e non è possibile per un algoritmo di ordinamento basato sul confronto.
3. Mostrare come è possibile modificare un albero AVL in modo da implementare una coda di priorità con le stesse prestazioni asintotiche di un heap. Si suppongano tutte le chiavi distinte. Non abbiamo problemi con i costi eccetto nell'estrazione del min che nell'heap ha costo costante e negli AVL logaritmico. Posso risolvere questo problema mantenendo sempre un riferimento al minimo nella radice che va eventualmente modificato nel caso di insert e remove.

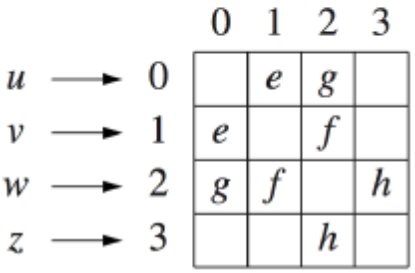
Grafi: possibili rappresentazioni

- Lista di nodi + lista di archi (con riferimento ai nodi)
- Lista di adiacenza: lista di nodi con ciascuno una lista dei vertici adiacenti
- Matrice di adiacenza: nodi in righe e colonne, archi nelle celle

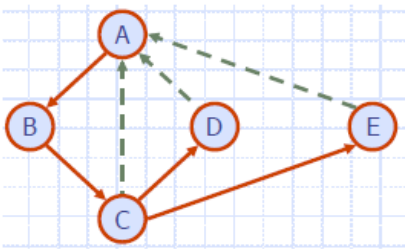




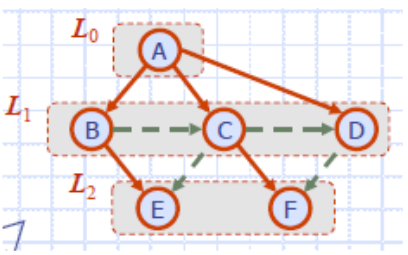




Un grafo può essere connesso o non connesso, può essere diretto o non diretto. Un albero è un particolare grafo connesso, non diretto e senza cicli. Una foresta è un insieme di alberi, di conseguenza è un grafo non connesso. Dato un grafo connesso posso sempre trovare un albero ricoprente.



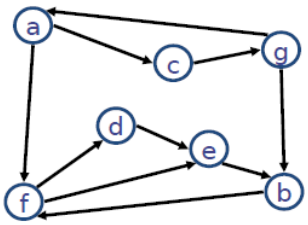
Ricerca in profondità DFS: determina se è connesso e determina una foresta ricoprente. Si usa per trovare un cammino fra due archi se esiste e per trovare un ciclo nel grafo. Visita tutti i vertici e archi della componente connessa del vertice iniziale V. Gli archi etichettati formano un albero ricoprente della componente connessa di V. Si può modificare per visitare sottografi non connessi e trovare una foresta ricoprente.



Ricerca in ampiezza BFS: determina il cammino più breve per raggiungere un nodo da quello di partenza.

Grafi diretti (digrafi). Vengono usati ad esempio per lo scheduling, ovvero quando abbiamo un insieme di attività che devono essere eseguite in un certo ordine. Se G è semplice (ovvero fra due nodi esiste al massimo un arco) m<=n*(n-1) dove m è il numero di archi e n è il numero di nodi. Può essere utile per mantenere liste di adiacenza diverse per archi entranti e uscenti. Una DFS diretta calcola il sottoinsieme di vertici raggiungibili dal vertice iniziale s.

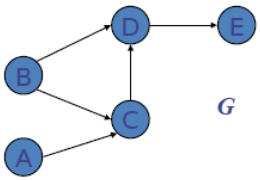
La proprietà di connettività forte si verifica quando ciascun vertice è raggiungibile da ogni altro vertice. Si può controllare se un grafo rispetta questa proprietà scegliendo un vertice e facendo una DFS, se al termine di questa esiste ancora un vertice non visitato allora la proprietà non è rispettata; successivamente si invertono tutti i vertici del grafico e si esegue nuovamente una DFS, se al termine esiste ancora un vertice non visitato allora la proprietà non è rispettata. Se entrambe le DFS vanno a buon fine allora possono restituire vero. La complessità di questo algoritmo è O(n+m).



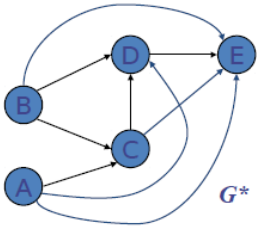
{ a , c , g }

{ f , d , e , b }

Si possono trovare anche le componenti (sottografi) fra di loro fortemente connesse in un grafo. Si utilizza l'algoritmo di Kosaraju: si esegue inizialmente una DFS del grafo inserendo i vertici visitati in una pila. Successivamente si esegue una DFS sul grafo trasposto per ogni vertice nella pila, l'insieme dei nodi trovati è una componente fortemente connessa. Complessità è O(n+m).

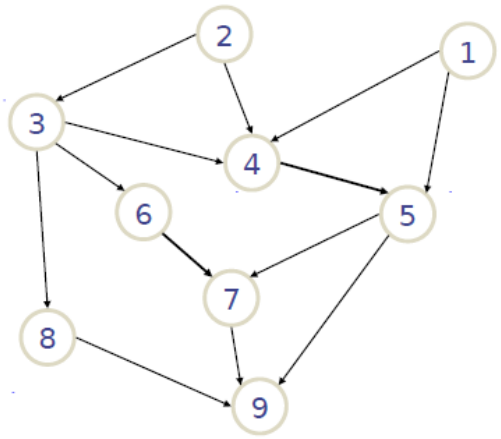


G



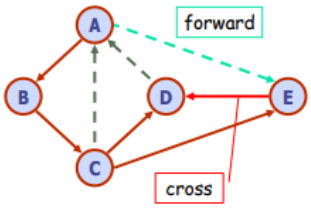
*G**

La chiusura transitiva di un digrafo G è il digrafo G* tale che abbia gli stessi vertici e se G ha un cammino diretto da u a v, allora G* ha un arco diretto da u a v. Ci permette di sapere in tempo costante se un vertice è raggiungibile da un altro vertice (utilizzando ad esempio una matrice di adiacenza). Si può fare la DFS da ognuno dei vertici ma ha costo O(n(n+m)), altrimenti si può considerare la matrice di adiacenza. Il quadrato della matrice di adiacenza ci dice quali nodi posso raggiungere in due passi, il problema principale è l'utilizzo di spazio per memorizzare le matrici, oltre al costo del calcolo dei prodotti fra matrici. Si può usare anche l'algoritmo di Floyd-Warshall per la chiusura transitiva. Si numerano i vertici, si considerano i cammini che usano 1,2,…,k come vertici intermedi. C'è un arco diretto da i a j in G_k se: c'è un arco diretto (i,j) in G_{k-1} o se ci sono archi diretti (i,k) e (k,j) in $G_{k'1}$. Il costo di questo algoritmo è vario, può essere migliore o peggiore dei metodi visti prima.

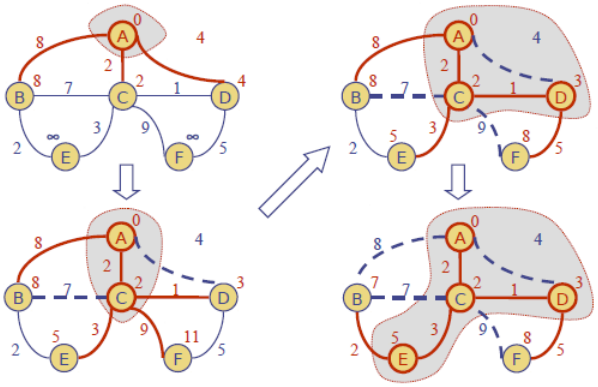


Ordinamento topologico: numera i vertici in modo tale che (u,v) in E implichi u<v. Un digrafo ammette un ordinamento topologico se e solo se è un DAG (digrafo aciclico). Se un grafo è un DAG esiste almeno un vertice che non ha archi entranti e almeno un vertice che non ha archi uscenti. Se si può definire un ordinamento topologico allora il grafo sarà privo di cicli. Si prende il vertice che non ha archi entranti e gli si assegna l'etichetta 1, poi si rimuove e rimane un altro DAG con un vertice senza nodi entranti a cui si assegna l'etichetta 2. Si prosegue così fino ad ottenere un ordinamento topologico del grafo iniziale. Si può eseguire questa procedura anche inversa, partendo dai vertici senza nodi uscenti e numerandoli a partire da n. Nella realtà non si crea veramente un secondo grafo da cui rimuovere di volta in volta i vertici ma si assegna a ciascuno di loro una variabile binaria 'rimosso'. Il costo di questo algoritmo è O(n+m). Si può implementare anche modificando una DFS.

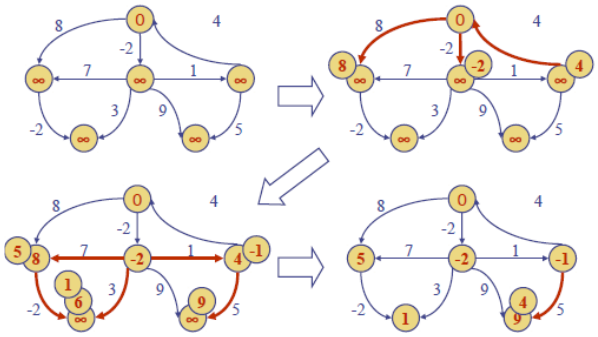
- Archi DFS tree:
  - *Tree, Discovery edges*
- Archi non-tree (in Digraph):
  - *Back edges*, verso antenato
  - *Forward edges*, verso discendente
  - *Cross edges*, verso altro vertice
- In Graph (non orientato): solo discovery e back



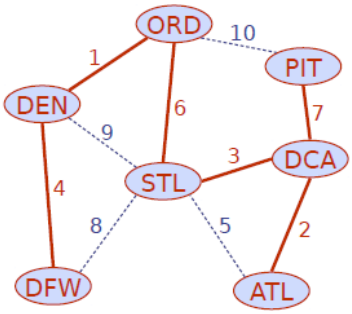
Cammini minimi in un grafo pesato. La visita in ampiezza BFS di un grafo a partire da un nodo s identifica cammini con il numero minimo di archi da s a qualsiasi altro vertice raggiungibile. Ci interessa però il caso in cui ogni arco abbia un peso diverso. Definiamo w(u,v) il peso dell'arco (u,v). Nel caso dei grafi diretti può non esistere un cammino fra due vertici ma possono anche esistere più cammini minimi di peso uguale. Ogni sottocammino di un cammino minimo è a sua volta un cammino minimo. Dato un grafo ed un nodo sorgente generalmente viene richiesto di calcolare un albero di cammini minimi dalla sorgente a tutti i vertici raggiungibili. Il risultato di quest'algoritmo è l'etichettatura di ogni nodo con la lunghezza del cammino dalla sorgente. d[u] è la distanza minima dalla sorgente s al nodo u se e solo se per ogni (v,u) d[u]<=d[v]+w(v,u). Algoritmo di Dijkstra: dato un vertice calcolare l'albero dei cammini minimi da s ad ogni altro vertice. Assumiamo il grafo connesso, i pesi non negativi e il grafo non diretto. Si imposta tutte le d[u] ad un valore massimo. Si fa crescere un sottoinsieme di vertici S che all'inizio include solo s e alla fine include tutti i vertici. Ad ogni passo si aggiunge a S il vertice u che non gli appartiene e che ha d[u] minima. Si aggiornano anche le etichette dei vertici adiacenti a u. La d di un vertice va diminuendo finchè alla fine è quella del cammino minimo. Una volta messo un nodo in S non modifico più la sua d[u]. Il passo chiave di questo algoritmo è il rilassamento, ovvero quando aggiorno



d[z]=min(d[z], d[u]+w[u,z]). Si implementa con una coda di priorità che rende meno costosa la scelta del nodo con d[u] minore, è da considerare che avremo bisogno di modificare le chiavi per cui serve un heap flessibile. La complessità dell'algoritmo di Dijkstra è O((m+n)logn) sfruttando una coda di priorità. Se voglio calcolare le distanze minime fra ogni coppia di vertici posso eseguire l'algoritmo di Dijkstra n volte (una per ogni possibile vertice sorgente) con costo O(n(n+m)logn).

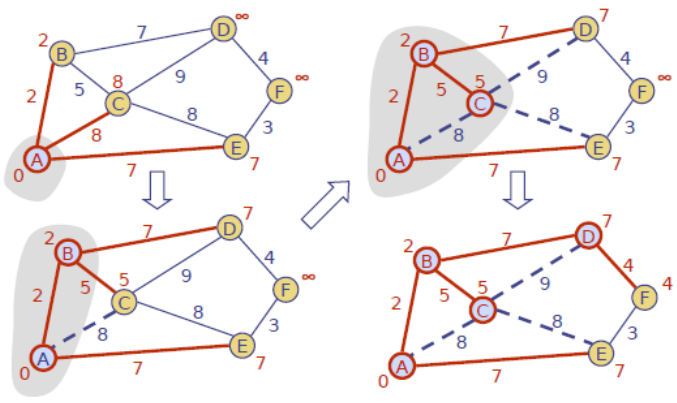


L'algoritmo di Bellman-Ford funziona nel caso di grafi diretti privi di cicli con pesi negativi e per grafi non diretti con pesi positivi. Ha complessità O(m*n). Per mantenere l'albero dei cammini minimi devo salvare in v anche il nodo che mi ci ha fatto arrivare con percorso minimo (lo faccio durante il rilassamento se aggiorno d[v]). Altrimenti possiamo salvare il predecessore di ogni vertice nell'albero dei cammini minimi in un array esterno.

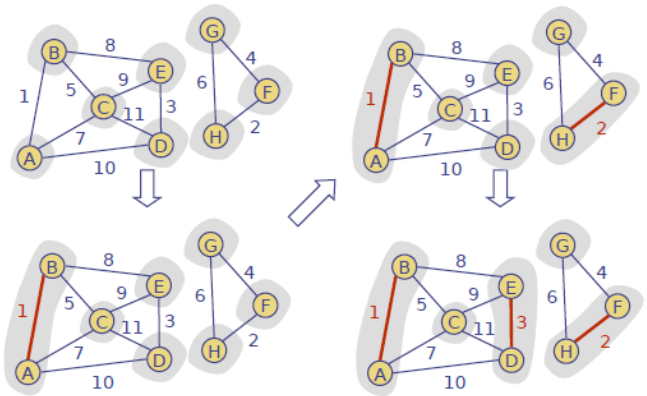


Alberi minimi ricoprenti (Minimum Spanning Tree). Consideriamo grafi non diretti con pesi positivi. Vogliamo un albero che tocchi tutti i nodi tale che la somma dei pesi degli archi scelti sia minima (peso complessivo). Proprietà:

- Di ciclo: per ogni arco f di C w(f)<=w(e) (con un MST T, un arco e non in T, un ciclo T unito ad e C), ovvero il peso di e deve essere maggiore degli altri, altrimenti potremmo ottenere un albero ricoprente di peso minore.
- Di taglio: un taglio è una partizione di V in due sottoinsiemi V1 e V2, l'arco del taglio è qualsiasi arco con un estremo in V1 e l'altro in V2. Sia e l'arco di peso minimo del taglio (V1, V2), allora esiste un MST che contiene e.



Algoritmo di Prim-Jarnik: prendiamo un qualsiasi nodo di partenza e tutti i suoi archi. Assegniamo a lui valore zero e agli altri il peso dell'arco. Poi si sceglie il nodo collegato dall'arco di peso minimo e si mette come visitato, si aggiunge l'arco al MST. Si aggiornano le distanze dei nodi non visitati a lui adiacenti se sono minori di quelle dal primo nodo. Poi scelgo l'arco con peso minore che collega un nodo visitato e uno non visitato e lo aggiungo al MST, segno il nodo come visitato. Proseguo fino ad aver visitato tutti i nodi e ottengo un MST. Il costo di quest'algoritmo è O((n+m)logn).



Algoritmo di Kruskal: ogni nodo parte da solo, poi si sceglie l'arco di peso minimo e si uniscono due nodi con quello. Poi si controlla il successivo arco con peso minimo e, se non unisce due nodi già legati, si usa per unire due nodi o gruppi di nodi. Si prosegue fino ad ottenere un MST. Ha complessità O((m+n)logn).