ALGORITMI E STRUTTURE DATI

Algoritmi

Un **algoritmo** è procedimento di calcolo che prende come **input** un insieme di valori e genera un valore o un insieme di valori, come **output**.
Può essere considerato anche come uno strumento per risolvere un **problema computazionale** ben definito.

La sequenza di input è detta ***istanza*** del problema dell'ordinamento.

Si parla di **sintesi** per indicare la creazione di un algoritmo per risolvere un problema.

Un algoritmo si dice **corretto** se, per ogni istanza di input, termina con l'output corretto.
L'**analisi** di un algoritmo consiste nel verificare se l'algoritmo risolve il problema per cui è stato disegnato e considera quindi la **correttezza** e la **complessità** (tempo e spazio) dell'algoritmo stesso.
La **complessità** indica la quantità di risorse consumate per eseguire un algoritmo:

- tempo computazionale (valutare il numero di operazioni che l'algoritmo esegue);
- spazio occupato;
- dispositivi hardware utilizzati.

Un consumo eccessivo di queste risorse pregiudica l'effettivo utilizzo dell'algoritmo stesso.

La classificazione consiste nel classificare gli algoritmi in base alla quantità di risorse che utilizzano per risolvere il problema.

Misure di calcolo della complessità

Il numero di operazioni elementari eseguite dall'algoritmo A sull'istanza x indica il **tempo di calcolo $T_A(x)$.** Esso per esempio, dipende dal numero di elementi che voglio ordinare.

Il numero di celle di memoria utilizzate dall'algoritmo su x viene detto **spazio di memoria $S_A(x)$**.

Si considera il tempo di esecuzione di un algoritmo nel **caso peggiore** o nel **caso medio**.
Il **caso peggiore** costituisce un limite superiore al tempo di esecuzione per qualsiasi input .
Conoscendo questo tempo, abbiamo la garanzia che l'algoritmo non potrà impiegare di più.
La complessità nel caso peggiore fornisce spesso una valutazione troppo pessimistica.

Il **caso medio** prende in considerazione i tempi di calcolo di tutte le istanze per poi calcolarne la media.
Spesso però questo caso è "brutto" quanto quello peggiore.
La complessità del caso medio assume una distribuzione uniforme sulle istanze.

Progettare gli algoritmi

l problema **dell'ordinamento** può essere definito nel modo seguente:

Input: una sequenza di *n* numeri <a1 , a2 , . . . , an>.
Output: una permutazione (riordinamento) <a1', a2', . . . , an'> della sequenza di input tale che a1'<= a2'<= ... <= an'.

Di algoritmi di ordinamento ce ne sono diversi. La scelta dell'algoritmo più appropriato dipende dal numero di elementi da ordinare, dal livello di ordinamento iniziale degli elementi, da eventuali vincoli sui valori degli elementi e dal tipo di unità di memorizzazione da usare.

METODO DIVIDE ET IMPERA
Gli algoritmi che utilizzano questo metodo sono **algoritmi ricorsivi.** In particolare con questa tecnica

suddividono il problema in vari sottoproblemi, che sono simili al problema originale, ma di dimensioni piccole, risolvono i sottoproblemi in modo ricorsivo e, poi, combinano le soluzioni per creare una soluzione del problema originale.

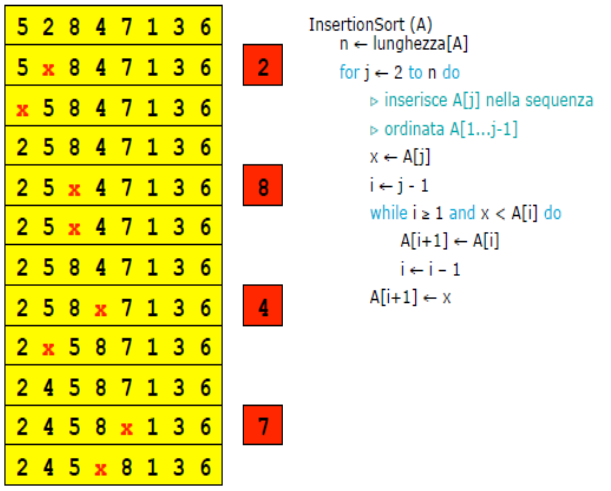
Questo metodo prevede tre passi:

Divide: il problema viene diviso in un certo numero di sottoproblemi.
Impera: i sottoproblemi vengono risolti in modo ricorsivo.
Combina: le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema originale.

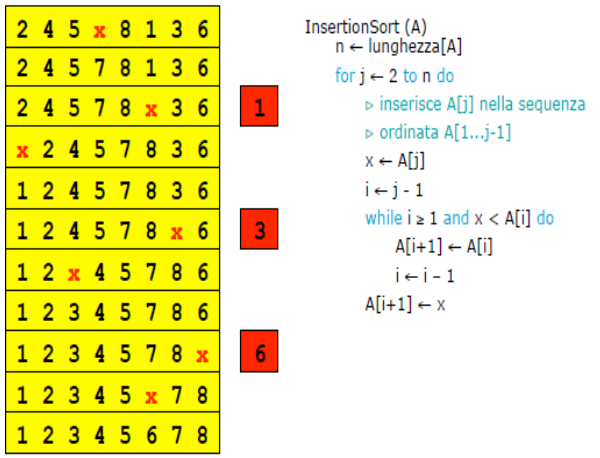
Algoritmo Insertion-Sort

Risolve il problema dell'ordinamento visto in precedenza ed è efficiente per ordinare un piccolo numero di elementi.
La sequenza di numeri da ordinare sono anche detti **chiavi**.
Nel caso peggiore la sua complessità è O(n^2), nel caso migliore invece è Ω(n).
Prende come parametro un array A[1 . . n] contenente una sequenza di lunghezza n che deve essere ordinata .
I numeri di input vengono ordinati sul posto: i numeri sono risistemati all'interno dell'array A, con al più un numero costante di essi memorizzati all'esterno dell'array in qualsiasi istante.
L'array di input A contiene la sequenza di output ordinata quando la procedura INSERTION -SORT è completata.

Esempio di Esecuzione (1)



Esempio di Esecuzione (2)



Algoritmo Merge-Sort

Questo algoritmo per l'ordinamento utilizza la tecnica **Divide et impera**. Infatti:

Divide: divide la sequenza degli n elementi da ordinare in due sottosequenze di n/2 elementi ciascuna.
Impera: ordina le due sottosequenze in modo ricorsivo utilizzando l'algoritmo merge sort.
Combina: fonde le due sottosequenze ordinate per generare la sequenza ordinata.

La procedura MERGE impiega un tempo Θ(n), dove n = r − p + 1 è il numero di elementi da fondere .
Nel caso peggiore impiega Θ(n log n).
L'idea consiste nel porre in fondo a ogni mazzo una carta sentinella, che contiene un valore speciale che usiamo per semplificare il nostro codice. In questo esempio usiamo ∞ come valore sentinella, in modo che quando si presenta una carta con ∞, essa non può essere la carta più piccola.

L'InsertionSort è meglio del MergeSort per array quasi ordinati sufficientemente grandi e anche per array piccoli.

Ricorrenze

La complessità può essere espressa mediante una relazione di ricorrenza che può essere risolta attraverso tre metodi:

- **sostituzione;**
- **albero di ricorsione;**
- **metodo dell'esperto**.

Metodo di sostituzione
Il metodo di sostituzione prima di tutto ipotizza un limite asintotico e successivamente verifica l'ipotesi fatta con una dimostrazione per induzione matematica.
Può essere usato per determinare il limite superiore e inferiore di una ricorrenza.

Metodo dell'albero di ricorsione
Questo metodo consiste nel costruire un albero di ricorsione i cui nodi rappresentano il costo di un singolo sottoproblema. Sommiamo i costi all'interno di ogni livello dell'albero per ottenere un insieme di costi per livello; poi sommiamo tutti i costi per livello per determinare il costo totale di tutti i livelli della ricorsione. Gli alberi di ricorsione sono particolarmente utili quando la ricorrenza descrive il tempo di esecuzione di un algoritmo divide et impera.
Questo metodo è un ottimo modo per formulare una ipotesi che poi sarà verificata con il metodo di sostituzione.

Metodo dell'esperto
Questo metodo viene utilizzato per risolvere equazioni ricorsive con la forma:

$$\mathbf{T(n) = aT(n/b) + f(n)}$$

Il metodo dell'esperto dipende dal **teorema dell'esperto.**

HeapSort

L'HeapSort è un altro tipo di algoritmo di ordinamento che utilizza gli attributi migliori sia dell'InsertionSort sia del MergeSort.
È un algoritmo che utilizza una struttura dati detta: **heap**.

L'**HEAP** è composto da un array considerabile come un albero binario. Ogni nodo dell'albero corrisponde al valore memorizzato nella relativa cella dell'array. L'array che rappresenta l'heap è caratterizzato dalla lunghezza dell'array ovvero dal numero di elementi dell'array e dalla dimensione dell'heap ovvero dal numero di elementi dell'heap registrati nell'array (heap – size [A]).
L'indice i identifica i nodi e il nodo genitore viene indicato con PARENT[i], il figlio sinistro: LEFT[i], quello destro: RIGHT[i].
Si possono avere due tipi di heap:

- **max heap**
- **min heap**

Max heap: il contenuto di un nodo padre è sempre maggiore o al massimo uguale al contenuto dei suoi nodi figli. In sostanza il valore maggiore si trova nella radice.
Min heap: il contenuto del nodo padre è minore o al massimo uguale al contenuto dei suoi figli.

Per l'algoritmo **heapSort** si utilizza il **max heap.**

Si definisce **altezza di un nodo** il numero di archi del cammino più lungo che a partire da un nodo scende fino a un nodo foglia.
Mentre l'**altezza di un heap** è l'altezza della sua radice.

MAX – HEAPIFY
È una sottoroutine che permette di manipolare i max – heap. Si suppone che l'albero su cui opera sia un max heap dove però viene violata la proprietà del max heap. In sostanza il valore in un nodo padre è più piccolo del valore del nodo figlio.
Questa sottoroutine prende a ogni passaggio il valore di ogni nodo e guarda se i suoi figli sono minori. Se lo sono la procedura termina altrimenti se i suoi figli hanno valore più grande scambia il figlio con valore maggiore con il padre. Questo accade ricorsivamente per ogni sottoalbero.

Il tempo di esecuzione è: $T(n) = O(\log n)$.

BUILT – MAX – HEAP
Costruzione di un max-heap a partire da un array utilizzando la procedura Max-Heapify dal basso verso l'alto. Si inizia un ciclo for che va da [lunghezza/2] down to 1. Per esempio se la lunghezza dell'albero è 10 (10 nodi) allora parto dal 5° nodo e lo considero come radice. Chiamo Max-Heapify e guardo se il padre è più grande dei figli. Se lo è passo al successivo nodo radice (nel nostro esempio sarà il 4°) se invece non è prendo il figlio con valore massimo è lo scambio con il padre.
Tutto questo avviene fino al nodo 1 ricorsivamente per tutti i sottoalberi dell'heap.

HEAP SORT

Questo algoritmo di ordinamento inizialmente chiama la procedura BUILT – MAX HEAP per un array in dimensione [1,..., n].
Dopo di che si fa un ciclo for che va da n down to 2 e dato che A[1] è l'elemento maggiore lo si scambia con il minore: A[n]. A[n] viene eliminato e di conseguenza la lunghezza dell'array diventa A[1...n-1].
A questo punto se la nuova radice viola la proprietà del max heap chiamo la procedura MAX-HEAPIFY creando così un max heap.
La complessità di questo algoritmo è $O(n \log n)$ perchè la chiamata a BUILT-MAX HEAP è $O(n)$ e quella al MAX-HEAPIFY è $O(\log n)$.

CODE DI PRIORITA'
Un'applicazione degli heap sono le code di priorità. Anch'esse, analogamente agli heap, possono essere code a max-priorità e code min-priorità.
Una **coda di priorità** è una struttura dati che mantiene un insieme di elementi S ai quali viene associato una chiave e che vengono estratti poi dalla coda in base alla priorità più alta.

Una coda max-priorità supporta le seguenti operazioni:
INSERT(S, x): inserisce l'elemento x in S
MAXIMUM(S): restituisce l'elemento in S con chiave più grande
EXTRACT – MAX(S): elimina e restituisce l'elemento S con chiave più grande
INCREASE – KEY (S, x, k): aumenta il valore della chiave di x al valore k. K dovrebbe essere almeno uguale alla chiave di x.

Una coda min-priorità supporta invece le seguenti operazioni:
INSERT
MINIMUM
EXTRACT - MIN
DECREAS – KEY

Dato che gli heap possono implementare le code di priorità, occorre memorizzare un **handle (aggancio)** per ogni elemento dell'heap che serva a collegare l'oggetto dell'applicazione. Tipicamente questi handle sono gli indici di array ed è per questo che serve mantenerli aggiornati ogni qual volta l'oggetto dell'applicazione viene modificato.

Un heap può svolgere qualsiasi operazione con le code di priorità su un insieme di dimensione n nel tempo $O(\lg n)$.

QuickSort

È un algoritmo di ordinamento che ordina sul posto il mio array. Utilizza, inoltre, il paradigma **divide et impera** (come nel merge-sort). In sostanza ho un array A [p....r] e lo **divido** in due sottoarray: A[p.... q-1] e A[q+1.... r]. L'elemento A[p] deve essere < = A [q] <= A[q+1... r].
Successivamente si chiama Quicksort per l'ordinamento.
Alla fine avremo l'array A[p... r] già ordinato a causa del fatto che quicksort esegue un ordinamento sul posto.
L'indice q viene determinato dalla procedura **PARTITION.**
Questa procedura partiziona l'array in base a un elemento detto **pivot** che corrisponde all'ultimo elemento presente nell'array. In sostanza, controlla se l'elemento di posizione j è <= all'elemento pivot. Tiene inoltre traccia dell'ultimo elemento non minore del pivot memorizzando il suo indice nella variabile i. A questo punto se l'elemento A[j] risulta essere <= al pivot (x) allora la i si incrementa di una posizione e viene scambiato l'elemento A[i] con A[j] altrimenti si riparte il ciclo fino all'elemento A[r-1]. Arrivati a quest'ultimo elemento si esce dal ciclo e si prende x (A[r]) e lo si mette nella posizione A[i+1]. Come ultima cosa si ritorna il valore i incrementato nuovamente di 1.
La complessità di questa procedura è $\Theta(n)$ dove $n = r - p + 1$.

QUICKSORT
L'algoritmo ha una complessità che è determinata dal fatto che il partizionamento sia bilanciato o sbilanciato.

PARTIZIONAMENTO BILANCIATO: se ad ogni partizionamento il numero di elementi che metto da una parte e dall'altra è costante.
PARTIZIONAMENTO SBILANCIATO: se ad ogni passo il numero di elementi che metto da una parte e dall'altra varia.

Nel **caso peggiore** da una parte avrò tutti gli elementi e dall'altra niente. La complessità sarà $O(n^2)$.
Nel **caso migliore** avremo un bilanciato perfetto delle due metà dell'array. La complessità sarà O(n log n).
Nel **caso medio** avremo una ripartizione con proporzionalità costante. La complessità sarà O(n log n).

Ordinamento in tempo lineare

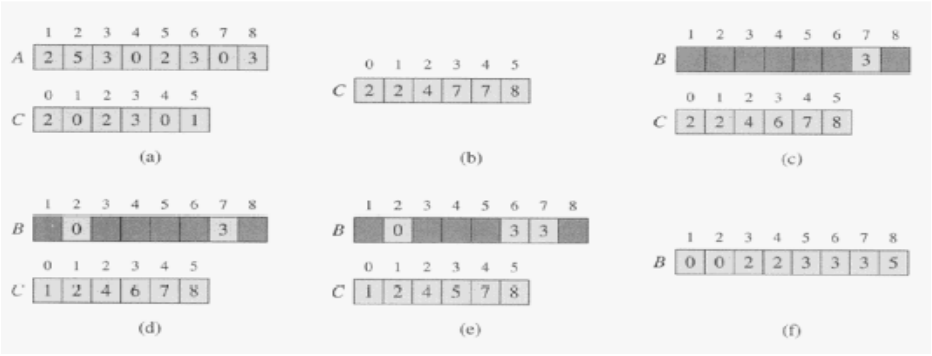
Gli **algoritmi di confronto (di ordinamento)** visti fino ad ora per ordinare n elementi operavano dei confronti tra gli elementi. Ora, invece, opereremo come se gli input fossero elementi distinti.

ALBERI DI DECISIONE
Gli ordinamenti per confronti possono essere visti astrattamente come **alberi di decisione** che sono alberi binari completi che rappresentano i confronti effettuati da un algoritmo di ordinamento che opera su n input.
Ogni nodo interno è rappresentato da $i:j$ per qualche $1 \leq i, j \leq n$.
Ogni nodo foglia è rappresentato da una permutazione $<\pi(1), \pi(2), \ldots, \pi(n)>$.
Nel ramo sinistro dell'albero ci saranno gli elementi $ai \leq aj$ e in quello destro gli elementi $ai > aj$.

Qualsiasi algoritmo di ordinamento per confronti richiede $\Omega(n \lg n)$ confronti nel caso peggiore. È sufficiente determinare l'altezza dell'albero di decisione dove ogni permutazione appare come una foglia raggiungibile.

COUNTING SORT
Questo algoritmo di ordinamento suppone che ogni numero n di input sia un numero intero compreso nell'intervallo [0, k]. Per ogni elemento di input x, conta il numero di elementi minori di x, in modo tale che si possa inserire x nella sua posizione dell'array di output.
In input avremo un array A[1, n] con lunghezza n e occorrono altri due array: B[1, n] che rappresenta l'output ordinato e C[0, k] che consiste in una memoria temporanea di lavoro.
La complessità sarà $\Theta(n+k)$.
Questo algoritmo di ordinamento mantiene l'ordine iniziale tra gli elementi di valore uguale infatti si dice che esso è **stabile**.

Esempio di esecuzione di CountingSort



RADIX SORT

E' un algoritmo di ordinamento che serve ad ordinare un array A di n elementi in una certa base b dove ogni elemento ha d cifre. La cifra 1 è quella con ordine più basso mentre la cifra d è quella di ordine più alto. In sostanza le cifre di ogni elemento sono comprese nell'intervallo [1, k].

L'ordinamento viene effettuato eseguendo d volte un algoritmo di ordinamento stabile (CountingSort) per ordinare l'array rispetto a ciascuna delle d cifre partendo dalla mento significativa.

Complessità: $T(n, d, k) = \Theta(d(n + k))$.

Questo algoritmo non è da preferire se lo spazio di memoria principale è limitato e quindi è meglio un algoritmo di ordinamento sul posto come quicksort.

Esempio di esecuzione di RadixSort

	4 3 2 1	4 3 2 1	4 3 2 1	4 3 2 1	4 3 2 1
A[1]	1 4 2 7	9 8 9 0	8 2 2 3	0 0 3 9	0 0 3 9
A[2]	0 2 4 1	0 2 4 1	7 5 2 5	3 1 6 2	0 2 4 1
A[3]	7 5 2 5	3 1 6 2	1 4 2 7	8 2 2 3	1 2 3 9
A[4]	3 1 6 2	8 2 2 3	1 2 3 9	1 2 3 9	1 4 2 7
A[5]	9 8 9 0	7 5 2 5	0 0 3 9	0 2 4 1	3 1 6 2
A[6]	1 2 3 9	1 4 2 7	0 2 4 1	1 4 2 7	7 5 2 5
A[7]	8 2 2 3	1 2 3 9	3 1 6 2	7 5 2 5	8 2 2 3
A[8]	0 0 3 9	0 0 3 9	9 8 9 0	9 8 9 0	9 8 9 0

BUCKET SORT

E' un algoritmo veloce perchè fa ipotesi sull'input (come countingsort) supponendo che esso sia generato da un processo casuale che distribuisce gli elementi uniformemente nell'intervallo [0,1). Quindi gli elementi sono numeri reali.

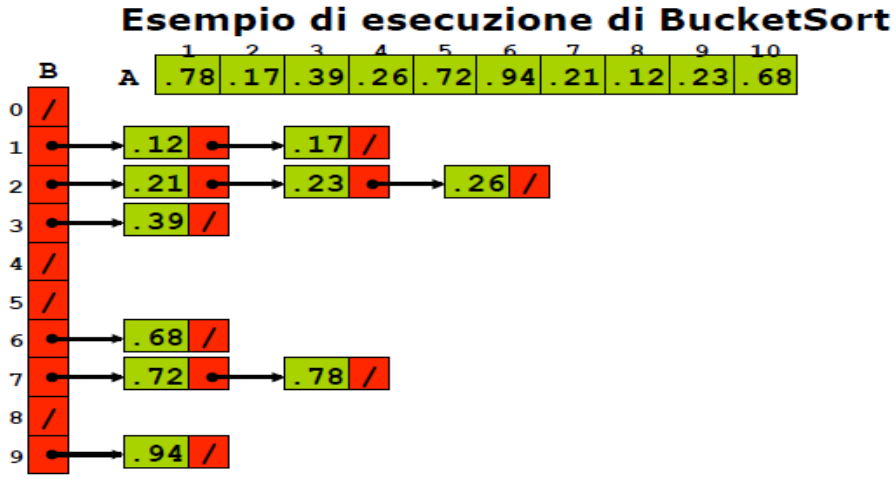
L'intervallo viene poi suddiviso in n sottointervalli con uguale dimensione detti ***bucket.***

Gli n numeri di input vengono poi distribuiti nei vari bucket.

Come output l'algoritmo tornerà i bucket ordinati poi verranno esaminati elencando gli elementi in ciascuno di essi.

Il codice di questo algoritmo suppone di prendere in input un array A di n elementi dove ogni elemento A[i] deve soddisfare la seguente proprietà: $0 \leq A[i] < 1$. Inoltre, l'algoritmo, si serve di un ulteriore array ausiliario B[0...n-1] di liste concatenate (bucket) supponendo ci sia un meccanismo per mantenere tali liste.

La complessità di esecuzione è $\Theta(n)$ quindi viene eseguito in un tempo lineare.



Strutture dati

Una ***struttura dati*** è un modo per memorizzare i dati, organizzarli e semplificarne l'accesso e la modifica.

STACK e CODE

Gli stack e le code sono insiemi dinamici dove l'elemento rimosso con DELETE è determinato a priori.

STACK
Con l'operazione DELETE viene eliminato l'elemento inserto per ultimo. Utilizza quindi la politica **FIFO** (First – In, First – Out).

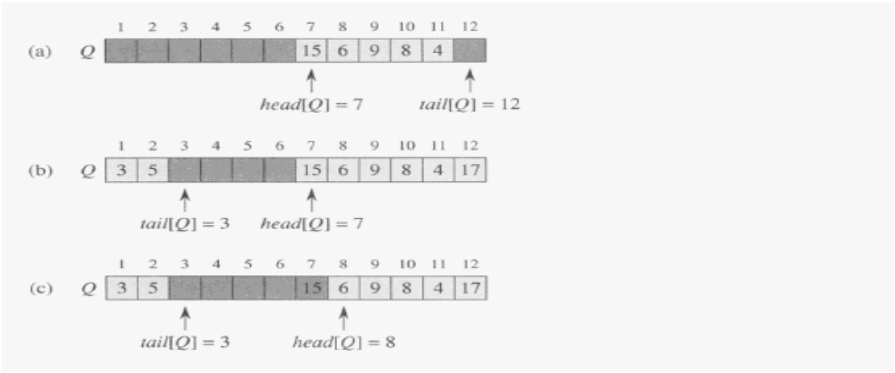
Con lo stack o **pila** l'operazione INSERT è chiamata PUSH (S, x) e la DELETE è chiamata POP(S).
Lo stack viene implementato con un array S di n elementi: S[1..n]. top[S] è l'indice dell'elemento inserito più di recente. L'array quindi sarà: S[1..top[S]]; dove S[1] è l'elemento in fondo allo stack. S[top[S]] invece è l'elemento in cima. Se top[s] = 0 lo significa che lo stack è vuoto. STACK-EMPTY(S) controlla appunto se lo stack è vuoto e se lo è significa che si è verificato un underflow e quindi un errore.
Se top[S] è > n allora si verifica un overflow che però nel codice non viene considerato.
La complessità di queste operazioni è O(1).
Ricordiamo che la pila ha una dimensione fissa.

CODA
Con l'operazione DELETE viene eliminato l'elemento inserito per primo. Utilizza quindi la politica **LIFO** (Last-In, First-Out).

L'operazione di INSERT è chiamata ENQUEUE (Q, x) e quella di DELETE: DEQUEUE(Q).
La coda ha un inizio detto **head** e una fine detta **tail**.
Viene implementata attraverso un array di n-1 elementi al massimo. L'array è Q[1..n].
Head[Q] punta all'inizio della coda e **tail[Q]** punta alla prossima posizione di inserimento.

Se head[Q] = tail[Q] significa che la coda è vuota e in caso voglia eliminare un elemento che non può esserci allora si verifica un **underflow**.
Se head[Q] = tail[Q] + 1 significa che la coda è piena e in caso di un nuovo inserimento si verifica un overflow.

Esempio di enqueue e dequeue



LISTE CONCATENATE
La **lista** è una struttura dati che contiene oggetti disposti in ordine lineare. L'ordine è determinato da un puntatore in ogni oggetto. Questo significa che gli elementi sono collegati da un puntatore.

I campi di una lista possono essere:

- **key**: campo chiave per la ricerca dei dati;
- **prev:** puntatore al precedente elemento;
- **next**: puntatore al prossimo elemento.

Se prev[x] = NIL significa che x non ha un predecessore quindi x è il primo elemento della lista detto head.
Se next[x] = NIL significa che x non ha un successore quindi x è l'ultimo elemento della lista detto tail.
Se head[L] = NIL significa che la lista è vuota.

Ci sono diversi tipi di liste:

- **singolarmente concatenate:** hanno i campi key e next;
- **doppiamente concatenate:** campi key, prev e next;
- **ordinata:** l'ordinamento lineare della lista corrisponde all'ordinamento lineare delle chiavi memorizzate negli elementi della lista. L'elemento minimo è la testa e quello massimo la coda.
- **Non ordinata:** gli elementi si possono presentare in qualsiasi ordine.
- **Circolare:** l'elemento prev della testa punta alla coda; l'elemento next della coda punta alla testa.

Noi considereremo solo le liste **non ordinate e doppiamente concatenate.**

Operazioni:

- LIST – SEARCH (L, x): trova il primo elemento con chiave = k in L restituendo un puntatore a questo elemento. Se x non c'è restituisce NIL. O(n) nel caso peggiore.
- LIST – INSERT (L, x): inserisce x la cui chiave deve essere già stata impostata, davanti alla lista concatenata. O(1)
- LIST – DELETE (L,x): deve ricevere linearmente un puntatore a x, poi elimina x e aggiorna i puntatori. O(1).

SENTINELLA: serve a rendere la DELETE più semplice. Consiste in un elemento finto sempre presente nella lista (nil[L]). Trasforma una lista doppiamente concatenata in una lista circolare doppiamente concatenata. La sentinella è posta tra la testa e la coda.

ALBERO RADICATO
Gli elementi di questo albero sono dinamici ed hanno una relazione gerarchica. La radice può avere 0 o più alberi collegati.
Ogni nodo avrà un campo chiave (key) e puntatori agli altri nodi che variano a seconda del tipo di albero.

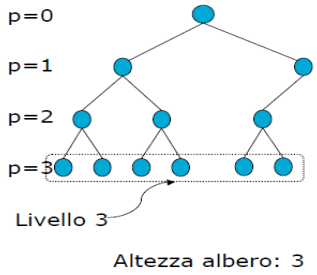
I puntatori sono:

- **p:** puntatore al padre;
- **left**: puntatore al figlio sinistro;
- **right:** puntatore al figlio destro.

Se left[x] = NIL significa che non esiste il figlio sinistro e così anche per quello destro.
root[T] punta alla radice dell'intero albero T e se è = NIL allora l'albero è vuoto.

La **profondità o livello** di un **nodo** è il numero di archi che bisogna attraversare per raggiungerlo dalla radice.
L'**altezza** di un **albero** è la massima profondità a cui si trova una foglia.

Esempio profondità, livello, altezza



Tecniche di visita degli alberi

- **Visita in profondità:** vengono visitati i rami uno dopo l'altro. Tre varianti: **visita in preordine, inordine, postordine.**
- **Visita in ampiezza:** a livelli, partendo dalla radice.

Visita in preordine: si visita prima la radice, poi si fanno le chiamate ricorsive sul figlio sinistro e poi destro.
Inordine: si fa prima la chiamata ricorsiva sul figlio sinistro, poi si visita la radice, e poi si fa la chiamata ricorsiva sul figlio destro.
Postordine: si fanno prima le chiamate ricorsive sul figlio sinistro e destro e poi si visita la radice.

Hashing

Dizionario: insieme dinamico A di elementi aventi un campo key e altri campi in cui vengono memorizzate altre informazioni legate alla chiave.

TABELLE A INDIRIZZAMENTO DIRETTO
La tecnica di indirizzamento diretto funziona bene se l'insieme universo U delle chiavi è piccolo.

Avremo una tabella T[0..m-1] dove ogni posizione o **slot** ha una chiave di U.
La tabella contiene i puntatori agli elementi k.

Le operazioni SEARCH, INSERT e DELETE vengono eseguite nel tempo O(1).

SVANTAGGI:
- serve riservare una memoria sufficiente per tante celle quante sono le chiavi possibili.
- Se U è troppo grande non è possibile utilizzare l'indirizzamento diretto.
- Se le chiavi effettivamente usate sono una piccola parte di U la memoria viene sprecata.
- Prima di utilizzare una tavola ad indirizzamento diretto occorre inizializzare tutti gli elementi a NIL. Tempo O(U).

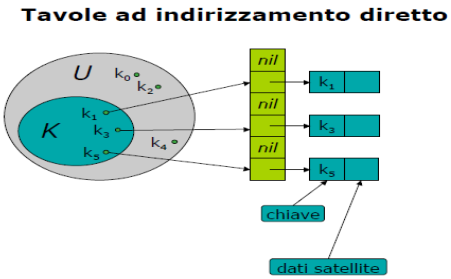


TABELLE HASH
Sono array contenenti un insieme di *m* celle molto minore della dimensione dell'universo.
Viene utilizzata una **funzione hash h** per calcolare lo slot della chiave k.
L'elemento con chiave k è h(k) che può essere considerato anche come valore hash di k.
Dato che vengono utilizzati per le chiavi solo m valori e non tutto U allora lo spazio di memoria usato sarà meno rispetto alle tabelle a indirizzamento diretto.

Problema: due chiavi possono essere associate allo stesso slot e questa viene detta **collisione.**
Per risolvere il problema delle collisioni vengono adottati diversi approcci tra cui quello del **concatenamento.**
Con questa tecnica tutti gli elementi di uno slot vengono messi in una lista concatenata. L'operazione INSERT sarà eseguita in un tempo O(1) come la DELETE (T, x) per la quale non occorre prima cercare l'elemento x perchè gli viene passato subito x e non la chiave k.

Data una tabella hash T con *m* slot in cui sono memorizzati *n* elementi, si definisce fattore di carico della tabella (**alfa**) il rapporto *n/m* che indica il numero medio degli elementi memorizzati nella catena. Alfa può essere minore, uguale o maggiore di 1.

SEARCH viene eseguita nel caso peggiore nel tempo $\Theta(n)$, quindi quando tutte le chiavi sono associate allo stesso slot.

Si suppone che ogni elemento abbia la stessa probabilità di essere associato a uno qualsiasi degli m slot, indipendentemente dallo slot cui sarà associato qualsiasi altro elemento. Questa è la base dell'**hashing uniforme semplice** (tutte le chiavi hanno probabilità 1/m)**.**
Le operazioni vengono eseguite nel tempo O(1).

Funzioni hash

Una buona funzione hash dovrebbe soddisfare l'ipotesi dell'**hashing uniforme semplice** vista sopra. Però non è possibile sempre verificare questa condizione.
Un dei metodi per buone funzioni hash è il **metodo della divisione** che calcola il valore hash come resto della divisione tra la chiave e un determinato numero primo.

Supporremo che l'insieme universo delle chiavi sia l'insieme N dei numeri naturali.

METODO DELLA DIVISIONE
Questo metodo, che serve a creare una funzione hash, consiste nel inserire una chiave k in uno degli m slot calcolato come resto della divisione di k per m. Ovvero h(k) = (k mod m).
Questo è un metodo molto veloce perchè fa solo una operazione di divisione.

Il valore m dovrebbe preferisci diverso da una potenza di 2 (2^p) quindi sarebbe meglio un numero primo non troppo vicino a una potenza esatta di 2.

METODO DELLA MOLTIPLICAZIONE
Il metodo della moltiplicazione consiste di 2 passi:

1. si moltiplica la chiave k per una certa costante A, $0 < A < 1$, estraendo la parte frazionaria kA;

2. moltiplico la parte frazionaria per m e prendo la parte intera inferiore del risultato.

In sostanza avremo che: $h(k) = m \lfloor (kA \bmod 1) \rfloor$.

In questo caso m non è critico. Esso può assumere un valore 2^p (potenza di 2).
Però, si ha la probabilità che funzioni bene questo metodo quando, in particolare, il valore di A si avvicina al valore: $(\sqrt{5} -1)/2$.

INDIRIZZAMENTO APERTO
Gli elementi sono memorizzati nella stessa tabella hash. Ogni slot potrà contenere un elemento dell'insieme dinamico oppure il valore NIL.
Per ricercare un elemento si scansiona ogni slot della tabella fino a quando non lo si trova.

Un difetto è che la tabella può riempirsi senza così fare altri inserimenti.
Il fattore di carico alfa non sarà mai > 1.

Il vantaggio di questa tecnica è che esclude i puntatori. Calcola la sequenza degli slot da esaminare senza seguire i puntatori.
Dato che i puntatori non ci sono posso avere più spazio per la tabella hash, a parità di memoria occupata.
Questo comporta ricerche più veloci e meno collisioni.

L'operazione INSERT scansiona la tabella fino a trovare uno slot vuoto per inserire la chiave. La sequenza di scansione dipende dalla chiave da inserire. Si chiede che per ogni chiave la sequenza di scansione sia una permutazione di < 0, 1,.., m-1> e che ogni posizione della tabella hash venga considerata come slot per una nuova chiave mentre la tabella si riempie.

La cancellazione è più complicata. Se cancello da uno slot i una chiava k non posso mettere nello slot il valore NIL perchè potrebbe essere impossibile ritrovare qualsiasi chiave k durante il cui inserimento abbiamo esaminato lo slot i ed era occupato. Una soluzione: marcare lo slot al valore speciale DELETED. Lo slot DELETED viene considerato dalla INSERT come vuoto.

Ci sono delle approssimazioni accettabili per implementare l'hashing uniforme semplice:
- **scansione (o ispezione) lineare;**
- **scansione quadratica;**
- **doppio hashing.**

SCANSIONE LINEARE
La funzione di hash si ottiene da una funzione hash ordinaria definita come **funzione hash ausiliaria.**

Questo tipo di implementazione presenta un problema: **clustering primario**.
In pratica si formano lunghe file di slot occupati e quindi il tempo per la ricerca si allunga.

SCANSIONE QUADRATICA
Parto con in input una chiave, applico la funzione di hash che mi restituisce una cella e se essa è occupata rieseguo la funzione di hash con i che va da 1 a m-1.
Nella tabella si fanno salti più lunghi.

È meglio di quella lineare anche se presenta anch'essa un problema anche se minore.
Problema: **clustering secondario:** celle occupate una dopo l'altra distanziate di un certo passo.
Come per la scansione lineare, anche in questa, ho solo m sequenze di scansione distinte.

DOPPIO HASHING
Risolve il problema dell'addensamento (clustering).

Si utilizzano due funzioni di hash.

Genera anch'essa sequenze di ispezione diverse ma molte di più rispetto alla scansione lineare e quadratica: m^2.
Sparpaglia maggiormente le chiavi nella tabella.

Alberi binari di ricerca

Gli **alberi binari di ricerca** sono strutture dati che possono eseguire molte operazioni su insiemi dinamici come: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE.
Possono essere utilizzati sia come dizionario, sia come coda di priorità.
Il tempo per le operazioni di base è proporzionale all'altezza dell'albero.

In un **albero binario** ogni nodo può avere al più due figli e viene rappresentato da record.

Un **albero binaro di ricerca** viene rappresentato da una struttura dati concatenata in cui ogni nodo è un oggetto.
In un albero binario di ricerca ogni nodo contiene i seguenti campi:

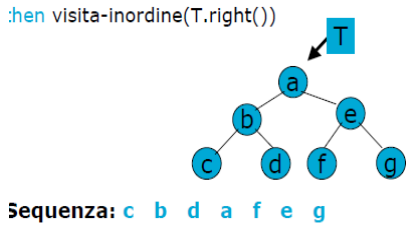
- key;
- dati satellite;
- left;
- right;
- p.

Se manca un figlio o il padre i rispettivi campi assumono il valore NIL.
La radice è l'unico nodo ad avere p = NIL.

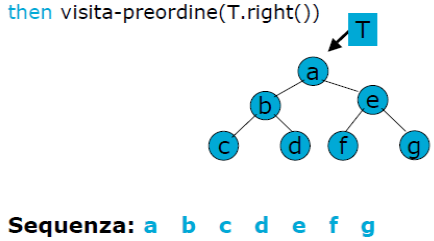
In questi alberi la memorizzazione delle chiavi deve soddisfare le proprietà degli alberi binari di ricerca:

- tutti i valori dei nodi del sottoalbero sinistro sono minori della radice;
- tutti i valori dei nodi del sottoalbero destro sono maggiori della radice.

Questa proprietà permette di visualizzare ordinatamente tutte le chiavi di un albero binario di ricerca con un semplice algoritmo di ricorsione di **attraversamento simmetrico di un albero (inorder)**. Questo algoritmo permette di visualizzare la radice fra i valori del suo sottoalbero sinistro e destro.

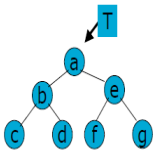


L'algoritmo di **attraversamento anticipato di un albero (preorder)** invece permette di visualizzare la radice prima dei valori dei suoi sottoalberi.



L'algoritmo di **attraversamento posticipato di un albero (postordine)** visualizza la radice dopo i valori dei suoi sottoalberi.

'visita T"



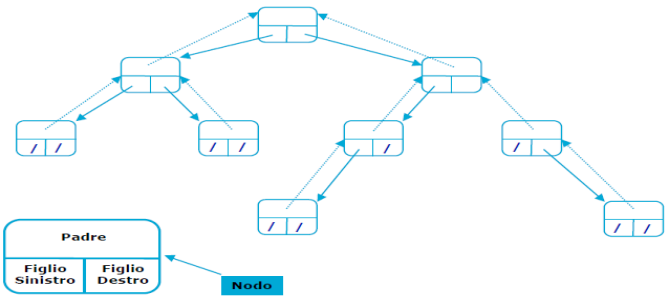
Sequenza: c d b f g e a

La procedura INORDER – TREE - WALK (ROOT [T]) stampa i nodi in ordine nel tempo O(n) con n nodi.
La procedura TREE – SEARCH dato un puntatore alla radice e una chiave k, restituisce un puntatore a un nodo con chiave k se esiste, altrimenti restituisce il valore NIL. Tempo O(h).
Con la procedura TREE – MINIMUM si seguono i puntatori left dei figli a sinistra, fino a quando viene incontrato NIL. Restituisce un puntatore all'elemento minimo del sottoalbero con radice in un nodo x.
TREE – MAXIMUM equivale al TREE – MINIMUM. Tempo O(h).
TREE – SUCCESSOR: il successore di un nodo x è il nodo con la più piccola chiave che è maggiore di key[x]. Restituisce il successore di un nodo x in un albero binario di ricerca, se esiste, o NIL se x ha la chiave massima nell'albero. TREE – PREDECESSOR è analogo. Tempo O(h).
La TREE – INSERT riceve un nodo z per il quale key[z] = v, left[z] = NIL e right[z] = NIL; modifica l'albero di ricerca T e qualche campo di z in modo che z sia inserito in una posizione appropriata nell'albero.
Inizia dalla radice dell'albero e segue un percorso verso il basso. Tempo O(h).
La TREE – DELETE richiede come argomento un puntatore al nodo z da cancellare.
Se:

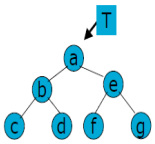
- z non ha figli, il padre p[z] viene modificato per sostituire z con NIL come suo figlio;
- z ha un figlio solo, tolgo z creando un nuovo collegamento tra suo figlio e suo padre;
- z ha due figli, si crea un nuovo collegamento per rimuovere il suo successore y che non ha un figlio sinistro, poi si sostituisce la chiave e i dati satelliti di z con la chiave e i dati satelliti di y.

Tempo O(h).

Esempio di albero binario



Esempio di visita in ampiezza



Sequenza: a b e c d f g

Alberi Red-Black

Gli alberi di ricerca Rosso – Neri sono alberi binari che utilizzano un bit in più di memoria per ogni nodo per identificare il nodo ovvero il **colore** associato a quel determinato nodo: **Rosso** o **Nero.**
Questi alberi sono approssimativamente bilanciati: assegnando vincoli al modo in cui possono essere colorati i nodi, garantisce che nessun percorso sia più di due volte più lungo di qualsiasi altro.

Ogni nodo contiene i seguenti campi:

- **parent**: puntatore al genitore;
- **left, right:** puntatore ai figli;
- **color**: colore del nodo;
- **key, data**: chiave e dati.

Un albero di ricerca per essere Rosso – Nero deve soddisfare le seguenti proprietà:

1- il nodo radice è sempre **NERO**;

2- i valori NIL sono **NERI**;
3- se un nodo padre è rosso, allora entrambi i suoi figli devono essere **ROSSI**;
4- per ogni nodo, tutti i percorsi da quel nodo fino alle foglie discendenti contengono lo stesso numero di nodi neri.

I nodi NIL vengono rappresentati da una sola **sentinella** che semplifica gli alberi rosso – neri.
Questo nodo sentinella evita di trattare diversamente i puntatori ai nodi dai puntatori nil.
Al posto di un puntatore nil, si usa un puntatore ad un nodo nil.

Definiamo **altezza nera** di un nodo x, indicata da **bh(x)**, il **numero di nodi neri** lungo un percorso che inizia dal nodo x (ma non lo include) e finisce in una foglia. Per la proprietà 4, tutti i percorsi da un nodo hanno lo stesso numero di nodi neri.
Per definizione: l'altezza nera di un albero è l'altezza nera della sua radice.
Tutte le operazioni effettuate (ricerca, massimo, minimo, precedente e successivo) su di un albero Rosso-Nero, hanno una complessità pari all'altezza dell'albero stesso ovvero h = O(log n).
Le operazioni di inserimento e cancellazione anche anch'essa questa complessità però dato che modificano l'albero potrebbero violare le proprietà degli alberi Rosso-Neri con la necessità quindi di ripristinare queste proprietà.
Per ripristinare queste proprietà occorre modificare il colore di alcuni nodi e modificare la struttura dei puntatori. Per modificare la struttura dei puntatori occorre fare delle rotazioni verso destra o sinistra.

ROTAZIONI PER L'INSERIMENTO O(lg n)

Caso 1: lo zio y di z è rosso.
Caso 2: lo zio di z è nero e z è un figlio destro.
Caso 3: lo zio di z è nero e z è un figlio sinistro.

ROTAZIONI PER LA CANCELLAZIONE O(lg n)

La procedura RB – DELETE rimuove un nodo e successivamente chiama la RB – DELETE – FIXUP che cambia i colori e fa delle rotazioni per ripristinare le proprietà red – black.

Se il nodo y rimosso è nero:
- se y era la radice e un figlio rosso di y diventa la nuova radice si viola la proprietà 1;
- se x e p[y] (ora anche p[x]) erano rossi, allora si viola la proprietà 3;
- se qualsiasi percorso che conteneva y ha un nodo nero in meno, la proprietà 4 è violata da qualsiasi antenato di y. Si corregge dicendo che x ha un nodo nero extra.

L'obbiettivo è di spostare il nero extra in alto nell'albero finchè:
- x punta a un nodo rosso e nero e di allora coloro di nero x;
- x punta alla radice e allora il nero extra può essere rimosso;
- vengono fatte rotazioni e ricolorazioni.

Caso 1: il fratello w di x è rosso
Dato che w deve avere figli neri possiamo cambiare i colori di w e p[x] e fare una rotazione sinistra di p[x]. Il nuovo fratello di x, adesso è nero, quindi si trasforma il caso 1 nel caso 2, 3 o 4.

Caso 2: il fratello w di x è nero ed entrambi i figli di w sono neri
Dato che anche w è nero, tolgo un nero sia da x che da w, lasciando x con un solo nero e w rosso. Poi aggiungo un nero extra a p[x] per compensare la mancanza del nero.
Il nuovo nodo x viene colorato di nero.

Caso 3: il fratello w di x è nero, il figlio sinistro di w è rosso e il figlio destro di w è nero
Possiamo scambiare i colori di w e di suo figlio sinistro left[w] e poi effettuare una rotazione destra di w. Il nuovo fratello w di x ora è nero con un figlio destro rosso, quindi abbiamo trasformato il caso 3 nel caso 4.

Caso 4: il fratello w di x è nero e il figlio destro di w è rosso
Si cambia qualche colore e si fa una rotazione sinistra di p[x], si rimuove il nero extra da x rendendolo singolarmente nero.

Strutture dati per insiemi disgiunti

Gli insiemi disgiunti sono insiemi i cui elementi sono tutti distinti.
Gli insiemi disgiunti organizzati in collezioni vengono mantenuti in strutture dati. Ogni insieme viene identificato da un elemento particolare detto rappresentante. Ogni elemento degli insiemi viene rappresentato da un oggetto.
Dato un oggetto x, posso eseguire le seguenti operazioni:

- MAKE – SET (x): viene creato un nuovo insieme con un solo membro che è x (quindi x è anche il rappresentante). X non deve essere presente in nessun altro insieme dato che gli elementi devono essere disgiunti.
- UNION (x, y): unisce due insiemi contenenti x e y in un nuovo insieme. Gli insiemi devono essere disgiunti.
- FIND – SET (x): restituisce il puntatore al rappresentate dell'insieme (unico) che contiene x.

Una semplice rappresentazione di questi insiemi è data dalle **liste concatenate.**
Il primo oggetto di ogni lista viene utilizzato come rappresentante.
Ogni oggetto nella lista contiene:

- info: un elemento dell'insieme;
- rappr: puntatore all'indietro al rappresentante;
- succ: un puntatore all'oggetto contenuto nel successivo elemento dell'insieme.

MAKE – SET: crea una nuova lista concatenata, con un unico oggetto x. O(1).
FIND – SET: restituisce il puntatore da x al rappresentante. O(1).
UNION: eseguita appendendo la lista di x alla fine di quella di y e il rappresentante del nuovo insieme è l'elemento che era il rappresentante dell'insieme di y. Purtroppo deve essere aggiornato il puntatore al rappresentante per ogni nodo della lista più lunga. teta(n2) dove n2 è la lunghezza della seconda lista.

Supponendo che ogni rappresentante contenga anche la lunghezza (lung) della lista e che si appenda sempre la lista più corta a quella più lunga, si parla di EURISTICA PER L'UNIONE PESATA.

Un altro tipo di rappresentazione ma più efficiente e veloce è data dagli **alberi radicati**. Ogni nodo contiene un elemento ed ogni albero rappresenta un insieme.
In una foresta di insiemi disgiunti ogni elemento ha un puntatore solo al padre. La radice contiene il rappresentante.

MAKE – SET: crea un albero con un solo nodo;
FIND – SET: eseguita controllando i puntatori al padre fino a che non viene trovata la radice dell'albero.
UNION: la radice di un albero punta alla radice di un altro albero.

Euristica dell'unione per rango: simile a quella per l'unione pesata vista per le liste. La radice dell'albero più basso deve puntare a quella dell'albero più alto.
Per ogni nodo si tiene un rango che è un limite superiore all'altezza dell'albero.
In sostanza, la radice con rango più piccolo punta alla radice con rango maggiore nel corso dell'UNION. Se eseguita individualmente: O(m log n) dove m sono il numero di operazioni e n il numero di operazioni MAKE – SET.

Euristica della compressione dei cammini: usata per le FIND – SET per far si che ogni nodo sul cammino di accesso punti direttamente alla radice.
Se eseguita individualmente:

teta (n + f log n)	se f < n;
teta (f $\log_{(1 + f/n)}$ n)	se f >= n.

dove m sono il numero di operazioni, n quelle MAKE – SET e f quelle FIND – SET.

Alberi di connessione minima

Si vuole trovare un sottoinsieme aciclico che connetta tutti i vertici di un grafo e tale che venga minimizzato il peso totale. Questo sottoinsieme aciclico deve formare un **albero di copertura.** Il problema di costruire un albero di copertura si può risolvere con due algoritmi: **Kruskal e Prim.** Questi due algoritmi vengono eseguiti nel tempo O (E lg V) se usano heap binari mentre quello di Prim se usa heap di Fibonacci viene eseguito nel tempo O(E + V lg V).
I due algoritmi illustrano inoltre un'euristica di ottimizzazione: **la strategia greedy.** Questa strategia prescrive di fare la scelta più conveniente quando bisogna scegliere tra varie alternative durante l'esecuzione di un algorimo.

Sia dato un grafo non orientato G = (V, E) con una funzione peso w: E ---> R, si vuole costruire un albero di connessione minima.
La strategia greedy utilizza un algoritmo generico che fa crescere l'albero di copertura minima di un arco alla volta.
L'algoritmo considera un sottoinsieme A di un qualche albero di copertura minimo e ad ogni passo viene determinato un arco che può essere aggiunto ad A senza violare la proprietà di A (deve essere un sottoinsieme di un qualche albero di copertura minimo). Questo arco è detto **arco sicuro per A** perché può essere aggiunto ad appunto senza violare la sua proprietà.

Si dice **TAGLIO (S, V-S)** di un grafo non orientato, una partizione di V.
Un arco attraversa il taglio se uno dei suoi estremi è in S e l'altro in V-S.
Un taglio rispetta un insieme A di archi se nessun arco di A attraversa il taglio.
Si dice **ARCO LEGGERO**: l'arco che attraversa il taglio ed il suo peso è minimo tra i pesi degli archi che attraversano quel taglio.

KRUSKAL (caso speciale dell'algoritmo generico)
L'insieme A è una foresta e l'arco sicuro che viene aggiunto ad A è sempre un arco di peso minimo che connette due componenti distinte.
Vengono esaminati gli archi ed ordinati in ordine di peso dal più piccolo al più grande. Successivamente per ogni arco si controlla se le estremità appartengono allo stesso albero; se è così l'arco viene scartato perchè non può essere aggiunto alla foresta senza creare cicli. Se invece i due vertici appartengo a due alberi diversi, viene aggiunto l'arco e i due vertici vengono fusi in un unico insieme.
Tempo O (E lg E).

PRIM (caso speciale dell'algoritmo generico)
A forma un singolo albero e l'arco sicuro aggiunto ad A è sempre quello peso minimo che connette l'albero a un vertice non appartenente all'albero.

Proprietà: gli archi di A formano un singolo albero.

L'albero parte da un arbitrario vertice radice e cresce fino a quando l'albero non copre tutti i vertici in v.
Ad ogni passo, viene aggiunto un arco leggero.

Durante l'esecuzione dell'algoritmo tutti i vertici che non sono nell'albero risiedono in una coda con priorità Q basata su di un campo chiave key.

- Si inizializza la coda con tutti i vertici ponendo al valore infinito la key di ogni vertice, tranne nella radice dove viene posta a zero.
- Il predecessore della radice viene posto a NIL.
- Vengono poi aggiornati i campi chiave e predecessore di ogni vertice v adiacente ad u ma non appartenente all'albero.

Il colore dei vertici è inizialmente bianco e diventa nero quando il vertice viene aggiunto all'albero.
Complessità: O(V lg V + E lg E).

Cammini minimi da sorgente unica

Dato un grafo G = (V, E) orientato e pesato, il **peso di un cammino** è la somma dei pesi associati agli archi che compongono il cammino.
Il **peso minimo da u a v** è uguale al minimo peso tra i cammini uv se esiste un cammino da u a v altrimenti è uguale a infinito.

Varianti del calcolo dei cammini minimi (casi):

1. cammini minimi da **sorgente unica** a tutti gli altri vertici;
2. cammini minimi da **ogni vertice ad un'unica destinazione**;
3. cammini minimi da **un'unica sorgente** ad **un'unica destinazione**;
4. cammini minimi da **ogni vertice** ad **ogni altro vertice**.

Con i cammini minimi da sorgente unica si possono avere archi con **pesi negativi**.
Se il grafo :

- non contiene cicli di peso negativo raggiungibili dalla sorgente s, allora per ogni vertice il peso del cammino minimo rimane ben definito anche se ha valore negativo;
- contiene cicli di peso negativo raggiungibili da s, allora il peso dei cammini minimi non rimangono più ben definiti. Nessun cammino da s a un vertice nel ciclo può essere minimo, dato che un cammino di peso minore può essere sempre trovato seguendo il cammino minimo proposto e quindi percorrendo il ciclo negativo. Il peso del cammino negativo è – infinito.

Se un vertice è raggiungibile da un cammino – infinito allora anche quel vertice avrà valore di – infinito.

Spesso è utile calcolare oltre ai cammini minimi anche i vertici che li compongono.
Dato un grafo si mantiene per ogni vertice v un predecessore che può essere un altro nodo o NIL.
Alcuni algoritmi gestiscono i predecessori in modo tale da ripercorrere all'indietro un cammino da un nodo all'altro.

Ricordiamo inoltre, che un cammino minimo, non può contenere alcun tipo di cicli.

Se un cammino è minimo anche i suoi sottocammini allora sono cammini minimi.

TECNICA DEL RILASSAMENTO
La tecnica del rilassamento è utilizzata dagli algoritmi che studieremo a breve.
Questa tecnica consiste nel associare ad ogni vertice un attributo d[v] chiamato **stima di cammino minimo** che costituisce un limite superiore al peso di cammino minimo da s a v; e a diminuire ripetutamente questo limite superiore al reale peso di cammino minimo di ogni vertice, finché il limite superiore non diventa uguale al peso di cammino minimo stesso.

Gli attributi d[v] e il predecessore vengono inizializzati sempre rispettivamente a infinito e NIL.
d[v] = 0 per v = s.

Si applica su un arco e verifica se passando per u è possibile diminuire la distanza per v.

```
s                 v                    s                 v
5 -----2-------> 9                     5 -----2------->6
         |                                    |
         | relax                              | relax
         |                                    |
5 -----2-------> 7                     5 ----2-------->6
```

Nell'algoritmo di Dijkstra e nell'algoritmo dei cammini minimi per grafi orientati aciclici, ogni arco viene rilassato una sola volta; mentre con Bellman – Ford viene rilassato più volte.

DIJKSTRA
Questo algoritmo risolve il problema di cammini minimi con sorgente singola su un grafo orientato e pesato se tutti i pesi degli archi **non sono negativi**.

Seleziona ripetutamente il vertice u che appartiene a V-s con la minima stima di cammino minimo, inserisce u in S e rilassa tutti gli archi uscenti da u.
L'algoritmo mantiene una coda a priorità Q contenente tutti i vertici di V-S, usando come chiave i rispettivi valori d.
Si assume, inoltre, che il grafo sia rappresentato con liste di adiacenza.
Utilizza la strategia greedy vista in precedenza.
Complessità: O(n^2).

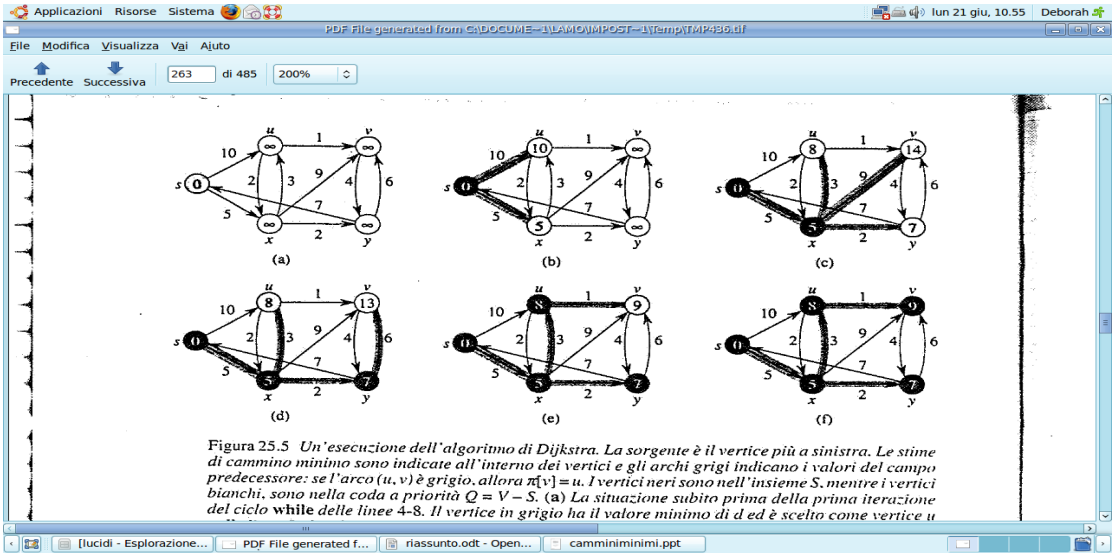


Figura 25.5 *Un'esecuzione dell'algoritmo di Dijkstra. La sorgente è il vertice più a sinistra. Le stime di cammino minimo sono indicate all'interno dei vertici e gli archi grigi indicano i valori del campo predecessore: se l'arco (u, v) è grigio, allora π[v] = u. I vertici neri sono nell'insieme S, mentre i vertici bianchi, sono nella coda a priorità Q = V – S.* (a) *La situazione subito prima della prima iterazione del ciclo* **while** *delle linee* 4-8. *Il vertice in grigio ha il valore minimo di d ed è scelto come vertice u*

BELLMAN – FORD
Questo algoritmo risolve il problema di cammini minimi con sorgente singola su un grafo orientato e pesato se tutti i pesi degli archi possono essere **negativi**.

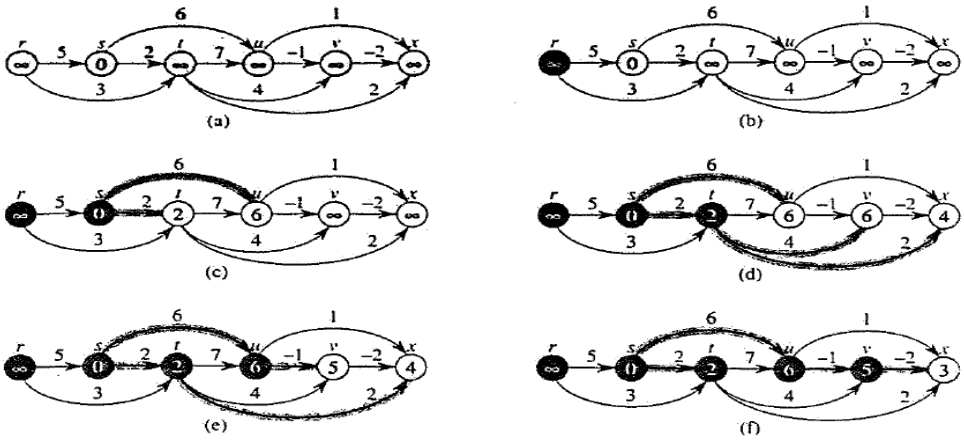
Restituisce un valore booleano che indica se esiste o no un ciclo di peso negativo raggiungibile dalla sorgente. Se esiste l'algoritmo indica che non esiste soluzione altrimenti se non esiste produce i cammini minimi e i loro pesi.

Questo algoritmo usa anch'esso, come Dijkstra, la tecnica del rilassamento.
Esegue V-1 passate ma esegue molti rilassamenti inutili.

Complessità: O(VE).

GRAFI ORIENTATI ACICLICI
I cammini minimi da sorgente unica possono essere calcolati nel tempo O(V+E) rilassando gli archi di un grafo orientato aciclico (DAG) pesato secondo un ordinamento topologico dei suoi vertici.
Nei dag non possono esistere cicli di peso negativo anche se esistono archi negativi.

Funzionamento dell'algoritmo:
ordina il dag topologicamente: se c'è un cammino da u a v allora u precede v.
Esegue una sola passata sui vertici seguendo l'ordinamento topologico: quando un vertice viene elaborato, tutti gli archi uscenti da esso vengono rilassati.



Cammini minimi tra tutte le coppie di vertici in un grafo

Dato un grafo orientato e pesato che associa ad ogni arco un peso a valore nei reali, si vuole trovare per ogni coppia di vertici un cammino minimo restituendo come output una tabella.
Il problema dei cammini minimi tra tutte le coppie si risolve con un algoritmo dei cammini minimi da sorgente unica. In particolare se i pesi degli archi non sono negativi si utilizza Dijkstra mentre se i pesi sono negativi si usa Bellman – Ford una volta per ogni vertice.

Il grafo viene rappresentato con una **matrice di adiacenza** dove:
se i = j allora w(i,j) = 0;
se i =! j e (i,j) appartengono a E allora w(i,j) = peso dell'arco orientato (i,j);
se i =! j e (i,j) non appartengono a E allora w(i,j) = infinito.

Sono ammessi archi di peso negativo.

Oltre alla matrice di adiacenza si deve calcolare anche la **matrice dei predecessori** e da essa, per ogni vertice, si definisce il **sottografo dei predecessori**.

Algoritmo Floyd – Warshall

Con questo algoritmo si possono avere archi negativi ma si assume di non avere cicli di peso negativo.
Esso considera i **vertici intermedi** di un cammino minimo ovvero: <V2, V3, …., Vv-1>.

Dato un insieme di vertici V di G si consideri un sottoinsieme di vertici da 1 a k per un qualche k.
Per ogni coppia di vertici ij, si considerino tutti i cammini da i a j cui i vertici intermedi stanno nel sottoinsieme e sia p un cammino di peso minimo tra di essi.
Se k:

- **non è** un vertice intermedio di p, allora tutti i vertici intermedi sul cammino p sono in (1, 2, …, k-1);
- **è** un vertice intermedio su p cammino minimo, allora spezzo k in p1 (cammino minimo da i a k) con vertici intermedi in (1, …, k-1) e p2 (cammino minimo da k a j) con vertici intermedi in (1, …, k-1).

Complessità: teta(n^3).

SOLUZIONE RICORSIVA
Dato il peso di un cammino minimo da i a j con vertici intermedi in (1, …, k), quando:

- k = 0: si ha un cammino che non ha vertici intermedi e quindi ha, al massimo, un arco.

COSTRUIRE UN CAMMINO MINIMO

Ci sono diversi metodi:

- calcolo la matrice D dei pesi di cammino minimo e ricavo la matrice dei predecessori da D;
- si costruisce la matrice dei predecessori "in linea" mentre l'algoritmo Floyd Marshall calcola D.

Se:

- k = 0: (cammino minimo senza vertici intermedi)
  - se i = j o il peso del cammino minimo ij = infinito, allora il predecessore ij = NIL;
  - se i =! j e il peso del cammino minimo ij < infinito, allora il predecessore ij = i.
- k >= 1: se prendo il cammino i---> k----> j allora il predecessore di j = predecessore del cammino minimo da k con vertici intermedi (1,...., k-1). Altrimenti scelgo lo stesso predecessore j scelto in un cammino minimo dai i a v con vertici (1, …, k-1).

CHIUSURA TRANSITIVA DI UN GRAFO

Consiste nel generare un altro grafo con gli stessi nodi e con archi da i a j se nel grafo precedente esiste un percorso da i a j.

Due soluzioni:

1. Assegno un peso pari a 1 ad ogni arco in E ed applico Floyd-Warshall e se esiste un cammino da i a j allora dij < n; altrimenti dj = ∞ ⇒ complessità Θ(n3)
2. Sostituisco le operazioni di min e + dell'algoritmo di Floyd-Warshall con le operazioni logiche $\vee$ (or) e $\wedge$ (and) Per i,j,k=1,…,n, definiamo tij (k) pari a 1 se esiste in G un cammino da i a j con tutti i vertici intermedi in {1,…,k}; 0 altrimenti.

Problema del flusso massimo

Una **rete di flusso** è un grafo orientato in cui ogni arco ha una capacità positiva quindi maggiore o uguale di zero.

La rete di flusso ha due vertici significativi:

- s: la sorgente;
- t: il pozzo, vertice di destinazione.

Per ogni vertice c'è un cammino da s a t.

Il **flusso** è una funzione a valori reali che deve soddisfare delle proprietà:

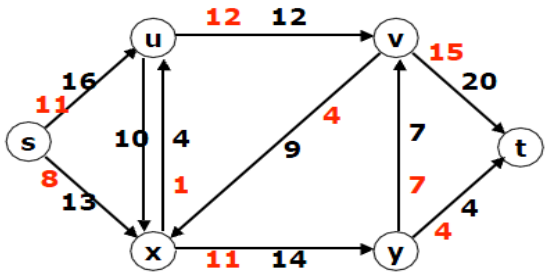
- **vincolo di capacità:** il flusso non può superare la capacità dell'arco, flusso <= capacità;
- **antisimmetria:** il flusso da u a v posso vederlo come un flusso negativo da v a u;
- **conservazione del flusso:** il flusso non può perdersi lungo il tragitto.

Il flusso può non essere sempre una quantità positiva (posso avere più sorgenti e più pozzi).

Dato rete di flusso, una sorgente s e un pozzo t, si vuole trovare un **flusso di valore massimo da s a t.**

Il **flusso totale netto** è la differenza fra il flusso totale uscente e il flusso totale entrante ed è uguale a zero.

Esempio di rete di flusso



Valore del flusso $|f| = 19$

METODO DI FORD – FULKERSON

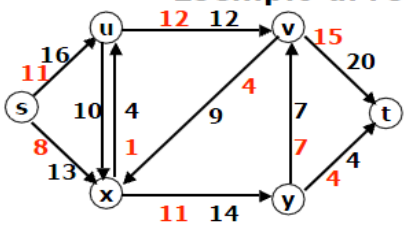
Si definisci un metodo e non un algoritmo perchè può avere diverse implementazioni eseguite in tempi diversi però tutte basate su tre concetti:

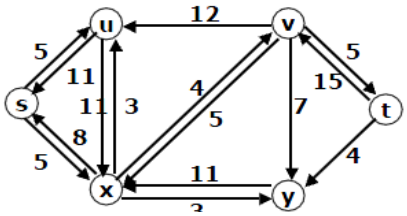
- **cammini aumentanti:** cammino non saturo che posso utilizzare ancora per trasmettere;
- **rete residua:** grafo con archi di capacità residua diversa da zero. Gli archi sdoppiano se non sono saturi;
- **taglio di una rete di flusso:** partiziono l'insieme dei nodi in due insiemi; sorgente e pozzo devono stare in due sottoinsiemi diversi. La capacità del taglio è la capacità degli archi che stanno in tutti e due i sottoinsiemi.

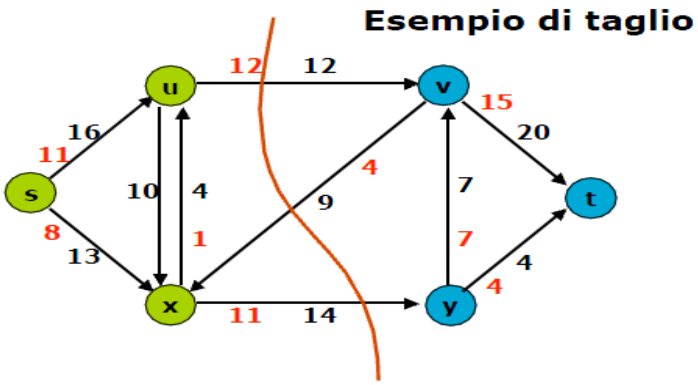
Si parte con f(u,v) = 0 per ogni u e v e quindi si aumenta iterativamente il valore del flusso cercando un cammino dalla sorgente s al pozzo t lungo il quale sia possibile inviare ulteriore flusso. Il procedimento termina quando non vi sono più **cammini aumentanti.**

Ford-Fulkerson-Method(G, s, t)
for "ogni u,v ∈ V[G]" **do**
    f(u,v) ← 0
while "esiste un cammino aumentante p" **do**
    "aumenta il flusso lungo p"
return f

Esempio di rete residua







FordFulkerson(G, s, t)
for “ogni uv ∈ E[G]” **do**
f(u,v) ← f(v,u) ← 0
while “esiste un cammino P da s a t in Gf” **do**
“calcola c(P) = min{cf(u,v) : uv arco di P}”
for “ogni arco uv di P” **do**
f(u,v) ← f(u,v) + c(P)
f(v,u) ← - f(u,v)
return f

Complessità: O(E f) dove E è il numero di archi e f è il flusso massimo.

Se le capacità legate agli archi non sono numeri interi, l'algoritmo appena visto potrebbe non terminare.

ALGORITMO DI EDMONDS – KARP
Questo algoritmo utilizza la visita in ampiezza.
Complessità: O(E^2 V)

Abbinamento massimo nei grafi bipartiti
L'insieme dei vertici si può dividere in S e D. Gli archi stanno sia in S che in D.
Devo trovare in insieme massimo di coppie distinte.

Il grafo non è orientato, i vertici rappresentano stazioni di commutazione, gli archi linee di trasmissione ed ogni arco è associato ad un numero che indica la probabilità che la linea di trasmissione trasmetta correttamente un messaggio.

Devo determinare il cammino più affidabile tra una coppia di vertici dati.

Grafo

Un grafo G = (V,E) è costituito da un insieme di vertici V ed un insieme di archi E ciascuno dei quali connette due vertici in V detti estremi dell'arco.

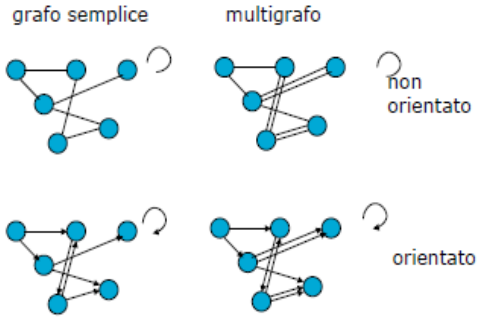
Grafo Orientato: Quando vi è un ordine tra i due estremi degli archi. In questo caso il primo estremo si dice coda e il secondo testa.

Cappio: è un arco i cui estremi coincidono.

Grafo non orientato semplice: se non ha cappi e non ci sono due archi con gli stessi estremi.

Grafo orientato semplice: se non ci sono due archi con gli stessi estremi. In caso contrario si parla di

multigrafo.



Archi:

Se un grafo è semplice indentifichiamo un arco con la coppia dei suoi estremi:

$e = uv \in E$ (diciamo che l'arco e è incidente in u e in v)

Se il grafo è orientato la coppia uv è ordinata (v è adiacente a u) altrimenti si dice che l'arco e esce da u ed entra in v (adiacenza simmetrica).

Grado di un nodo:

Il grado del vertice v è il numero di archi incidenti in v. Se il grafo è orientato si suddivide in un grado entrante (archi entranti) ed un grado uscente (archi uscenti).

Percorso in un grafo:

Un percorso di lunghezza k dal vertice u al vertice v in un grafo è una sequenza di k+1 ($x0 = u$ e $xk = v$) vertici. Se $k > 0$ e $x0 = xk$ il percorso è chiuso.
Un cammino è un percorso i cui vertici sono tutti distinti con la possibile eccezzione di $x0 = xk$ nel qual cosa esso è un ciclo. Un ciclo di lunghezza k=1 è un cappio. Un grafo aciclico è un grafo che non contiene cicli.

Raggiungibilità e componenti connesse:

Quando esiste almeno un cammino dal vertice u al vertice v diciamo che il vertice v è accessibile o raggiungibile da u.
Un grafo non orientato si dice connesso se esiste almeno un cammino tra ogni coppia di vertici. Le componenti connesse di un grafo sono le classi di equivalenza dei suoi vertici rispetto alla relazione di raggiungibilità.

Componenti fortemente connesse:

Un grafo orientato si dice fortemente connesso se esiste almeno un cammino da ogni vertice u ad ogni altro vertice v. Le componenti fortemente connesse di un grafo sono le classi di equivalenza dei suoi vertici rispetto alla relazione di mutua accessibilità.

Sottografo:

Un sottografo del grafo $G = (V,E)$ è un grafo $G' = (V',E')$ tale che $V' \subseteq V$ e $E' \subseteq E$.
Il sottografo di $G = (V,E)$ indotto da $V' \subseteq V$ è il grafo $G' = (V',E')$ tale che:

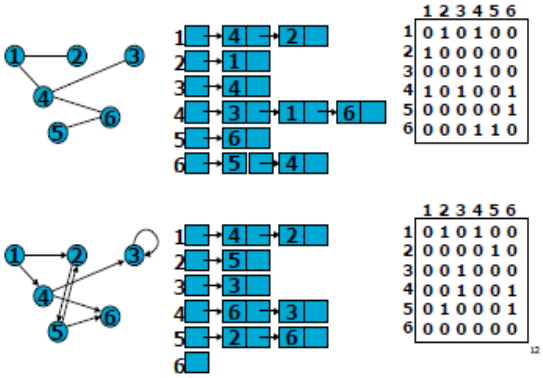
$E' = \{uv: uv \in E \text{ e } u,v \in V'\}$

Rappresenteazione di grafi: Liste di adiacenza

E' costituita da una lista Adj[u] (per ogni vertice u) che contiene i vertici adiacenti al vertice u.
Quantita memoria richiesta (sia orientato che non): O(|V|+|E|)
Velocità per determinare se un arco uv è presente: O(|V|)

Rappresentazione di grafi: Matrice delle adiacenze

Assume che i vertici siano numerati 1,2,...,|V| in modo arbitrario. La martrice è booleana e segna 1 se i vertici sono adiacenti altrimenti 0.
Quantita di memoria richiesta: O(|V|^2)
Velocita per determinare se un arco uv è presente: accesso diretto.



Visita in ampiezza (BFS):

Dato un grafo e un vertice scelto come sorgente, la visita in ampiezza parte da s e visita sistematicamente il grafo per scoprire tutti i vertici che sono raggiungibili da s.
Calcola la distanza (minima) di ogni vertice del grafo dalla sorgente s. Produce anche un albero BF i cui rami sono cammini di lunghezza minima. La visita espande uniformemente la forntiera tra i vertici scoperti e quelli non ancora scoperti.

Vertici colorati:

1) Bianco (vertici non ancora raggiunti)
2) Grigio (vertici raggiunti che stanno sulla frontiera)
3) Nero (vertici raggiunti che non stanno sulla frontiera)

Algoritmo BFS:

Assume che il grafo sia rappresentato con liste delle adiacenze. Usa una coda Q di vertici in cui memorizza la frontiera.
Complessità: O(n+m)

Proprietà delle distanze:

Indichiamo con d(u,v) la distanza del vertice v dal vertice u: la lunghezza di un cammino minimo che congiunge u e v.

$$d(x,v) \leq d(x,u) + 1 \text{ per ogni } x \in V \text{ e ogni } uv \in E$$

Proprietà del limite superiore e della coda:

Per ogni vertice u e per tutta l'esecuzione di BFS vale la diseguaglianza:

$$d[u] \geq d(s,u)$$

Se la coda Q non è vuota e contiene i vertici (v1,v2,...,vt) allora per ogni i=1, ..., t-1:

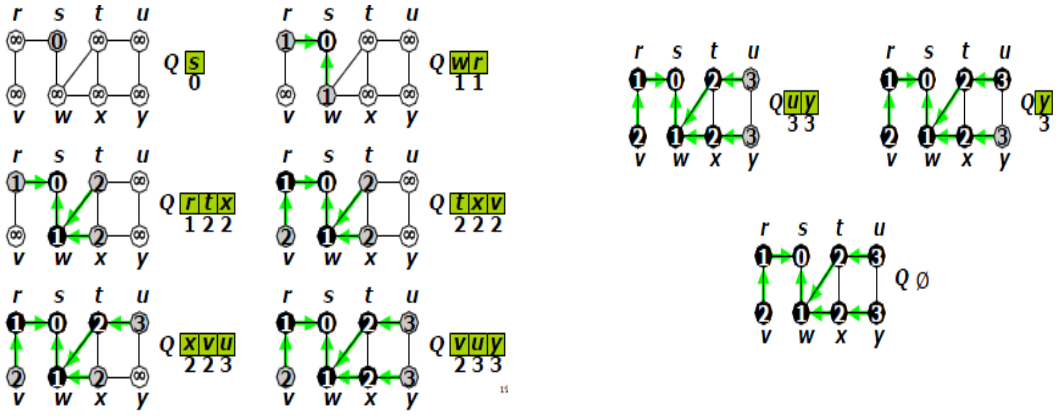
$$d[v_i] \leq d[v_{i+1}]$$

ed inoltre

$$d[v_t] \leq d[v_1] + 1$$

Correttezza di BFS

BFS visita tutti i vertici raggiungibili da s e quando termina d[v] = d(s,v) per ogni vertice v del grafo.



Visita in profondità (DFS):

Lo scopo è avanzare in profondità nella ricerca finchè è possibile. Si esplorano gli archi uscenti dal vertice u raggiunto per ultimo. Se viene scoperto un nuovo vertice v ci si sposta su tale vertice. Se tutti gli archi uscenti da u portano a vertici già scoperti si torna indietro e si riprende esplorando archi uscenti dal vertice cui u è stato scoperto. Il procedimento contiuna finchè si sono scoperti tutti i vertici raggiungibili dal vertiche iniziale scelto. Se non sono stati raggiunti tutti i vertici del grafo si ripete il procedimento partendo da un vertice non ancora raggiunto (si sceglie una nuova sorgente).

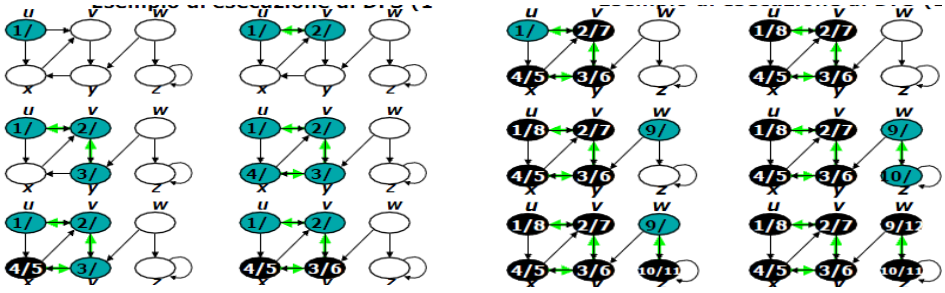
Vertici colore:

1) Bianco (vertici non ancora raggiunti)
2) Grigio (vertici scoperti)
3) Nero (vertici la cui lista delle adiacenze è stata completamente esplorata)

Marcatempi:

1) d[u] registra quando il vertice viene scoperto e colorato di grigio.
2) f[u] registra quando il vertice è stato completato e viene colorato di nero.

Complessità: O(|V| + |E|)



Classificazione degli archi:

1) Archi d'albero: archi uv con v scoperto visitando le adiacenze di u.
2) Archi all'indietro: archi uv con u=v oppure v ascendente di u in un albero della foresta di ricerca in profondità.
3) Archi in avanti: archi uv con v discendente di u in un albero della fortesta.
4) Archi trasversali: archi uv in cui v ed u appartengono a rami o alberi distinti della foresta.

Ordinamento topologico:

Un ordinamento topologico di un grafo orientato aciclico è un ordinamento lineare dei suoi vertici tale che:

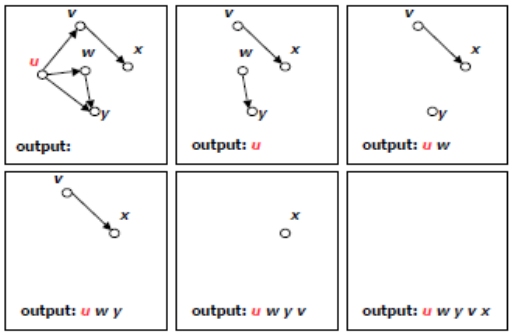
1) Per ogni arci uv ∈ E il vertice u precede il vertice v.
2) Per transitività, ne consegue che se v è raggiungibile da u, allora compare prima di v nell'ordinamento.

L'ordinamento topologico si usa per determinare un ordine in cui eseguire un insieme di attività in presenza di vincoli di precedenza.

Determinare l'ordinamento topologico:

Soluzione diretta:

1) Trovare ogni vertice che non ha alcun arco incidente in ingresso.
2) Stampare tale vertice e rimuoverlo insieme ai suoi archi.
3) Ripetere la procedura finchè tutti i vertici risultano rimossi.



Complessità: O(|V| + |E|)
(Un grafo orientato è aciclico se e solo se nella visita in profondita non si trova nessun arco all'indietro)

Componenti fortemente connesse di un grafo orientato:

Si possono calcolare con la visita in profondità. Una componente fortemente connessa di un grafo orientato è un insieme massimale di vertici $U \subseteq V$ tale che per ogni $u,v \in U$ esiste un cammino da u a v ed un cammino da v ad u.
Sono calcolate in tre fasi:

1) Si usa la visita in profondità in G per ordinare i vertici in ordine di tempo di completamento f decrescente (come per l'ordinamento topologico).
2) Si calcola il grafo trasposto del grafo G.
3) Si esegue una visita in profondita nel grafo trasposto usando l'ordine dei vertici calcolato nella prima fase nel ciclo principale.

Gli alberi della visita in profondità nel grafo trasposto rappresentano le componenti fortemente connesse.

Grafo delle componenti fortemente connesse:

Dato un grafo orientato G, il grafo delle componenti fortemente connesse di G è il grafo orientato H che ha:

1) Come vertici le componenti fortemente connesse di G
2) Un arco da una cfc C ad una cfc C' se e solo se in G vi è un arco che connette un vertice di C ad un vertice di C'.

Altre applicazioni della DFS:

Catatterizzazione di alcune importanti proprietà dei grafi (non orientati):

1) Ciclo di Eulero: ciclo in un grafo che visita ogni arco di G una volta.
2) Ciclo di Hamilton: ciclo in un grafo G che visita ogni nodo di G una volta.

Programmazione dinamica e greedy

Fasi per risolvere un problema:

1) Classificazione del problema
2) Caratterizzazione della soluzione
3) Tecnica di progetto
4) Utilizzo di strutture dati

Classificazione del problema:

Problemi decisionali: Il dato di ingresso soddisfa una certa proprietà?
Esempio: Stabilire se un grafo è connesso
Problemi di ricerca: Spazio di ricerca (insieme di soluzioni possibili) e soluzione ammissibile (soluzione che rispeta certi vincoli).
Esempio: posizione di una sottostringa in una stringa.

Caratterizzazione della soluzione:

Definire la soluzione dal punto di vista matematico. Le caratteristiche formali della soluzione possono suggerire una possibile tecnica.

Tecnica di progetto:

1) Divide-et-impera: Un problema viene suddiviso in sotto-problemi indipendenti, che vengono risolti ricorsivamente (top-down).
2) Programmazione dinamica: La soluzione viene costruita (bottom-up) a partire da un insieme di sotto-problemi potenzialmente ripetuti. I sotto-problemi in cui esso si può scomporre non sono indipendenti.
3) Greedy: Ad ogni passo si sceglie la soluzione che è localmente ottima.

Divide-et-impera vs programmazione dinamica:

Divide-et-impera: E' una tecnica ricorsiva vantaggiosa quando i sottoproblemi sono indipendenti altrimenti, gli stessi sottoproblemi possono venire risolti più volte.
Programmazione dinamica: E' una tecnica iterativa vantaggiosa quando ci sono sottoproblemi in comune.

Quando applicare la programmazione dinamica?

Sottostruttura ottimale: E' possibile combinare le soluzioni dei sottoproblemi per trovare la soluzione di un problema più grande. Le decisioni prese per risolvere un problema rimangono valide quando esso diviene un sottoproblema di un problema più grande.
Sottoproblemi ripetuti: Un sottoproblema può occorrere più volte.
Spazio dei sottoproblemi: Deve essere polinomiale.

Fasi della programmazione dinamica:

1) Definire la struttura di una soluzione ottima. Mostrare che una soluzione ottima si ottiene da soluzioni ottime di sottoproblemi.
2) Definire ricorsivamente il valore di una soluzione ottima. La soluzione ottima ad un problema contiene le soluzioni ottime ai sottoproblemi.
3) Calcolare il valore di una soluzione ottima con strategia "bottom up" (cioè calcolando prima le soluzioni dei casi più semplici).
4) Costruzione della soluzione ottima a partire dalle informazioni calcolate.

Algoritmi greedy:

Ogni volta si fa la scelta migliore localmente.

Elementi della strategia greedy:

Sottostruttura ottima: Ogni soluzione ottima non elementare si compone di soluzioni ottime di sottoproblemi.
Prorietà della scelta greedy: La scelta ottima localmente non pregiudica la possibilità di arrivare a una soluzione globalmente ottima.

Codici di Huffman:

Vengono ampiamente usati nella compressione dei dati. Permettendo un risparmio compreso tra il 20% e 90% secondo il tipo di file. Sulla base delle frequenze con cui ogni carattere appare nel file, l'algoritmo greedy di Huffman trova un codice ottimo. Associa ad ogni carattere una sequenza di bit detta parola codice.

Codici prefissi:

Un codice prefisso è un codice in cui nessuna parola è prefisso da una ulteriore parola.

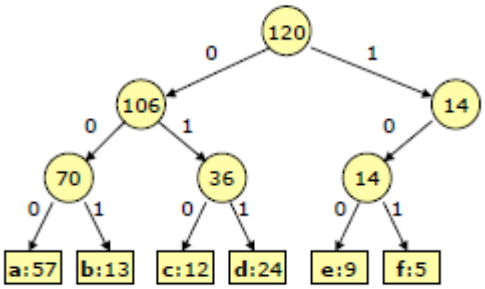
Decodifica di un codice prefisso:

1) Individuare la prima parola codice del file codificato.
2) Tradurla nel carattere originale e aggiungere tale carattere al file decodificato.
3) Rimuovere la parola codice dal file codificato.
4) Ripetere l'operazione per i successivi caratteri.

Per facilitare la suddivisione del file codificato in parole codice è comodo rappresentare il codice con un albero binario.

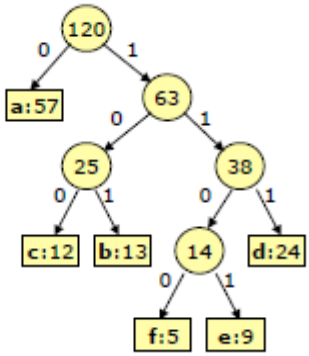
Codice a lunghezza fissa:

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. fisso	000	001	010	011	100	101



Codice a lunghezza variabile:

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5
cod. var.	0	101	100	111	1101	1100



Definizione formale del problema:

Codice ottimo: Dato un file F, un codiceC è ottimo per F se non esiste un altro codice tramite il quale F possa essere compresso impiegando un numero inferiore di bit.
Nota: Il codice ottimo dipende dal particolare file e possono esistere più soluzioni ottime.
Teorema: I codici a prefisso ottimi sono rappresentati da un albero in cui tutti i nodi interni hanno due figli.

Algoritmo greedy di Huffman:

Principio del codice di Huffman:

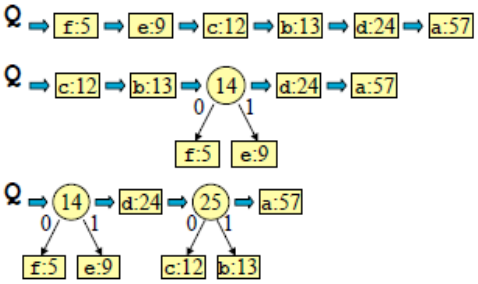
1) Minimizzare la lunghezza dei caratteri che compaiono più frequentemente.
2) Assegnare ai caratteri con la frequenza minore i codici corrispondenti ai percorsi più lunghi all'interno dell'albero.

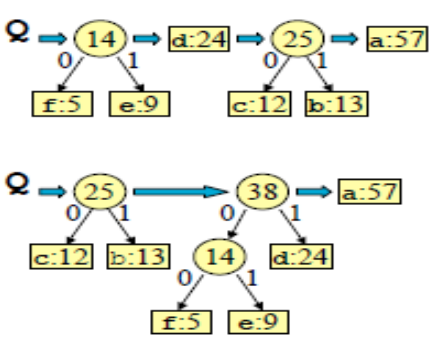
Un codice è progettato per un file specifico:

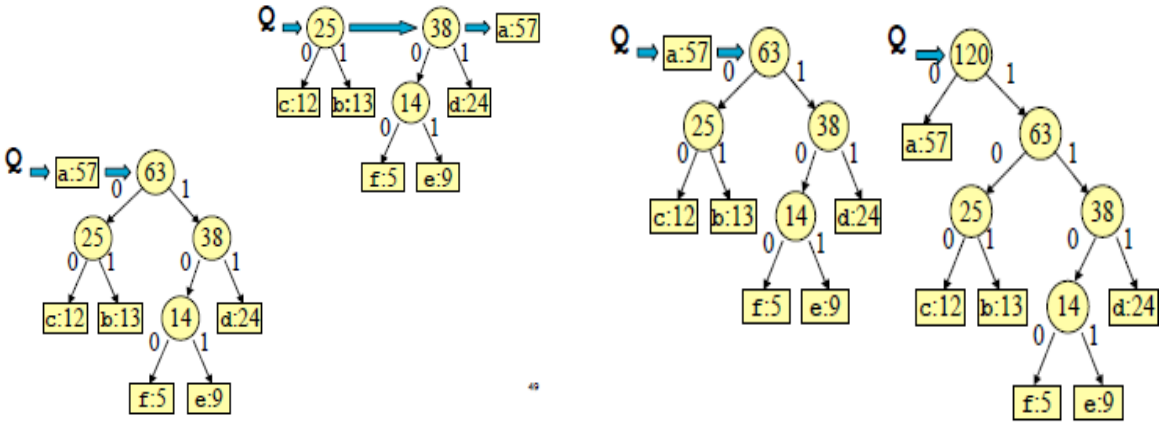
1) Si ottiene la frequenza di tutti i caratteri.
2) Si costruisce il codice.
3) Si rappresenta il file tramite il codice.
4) Si aggiunge al file una rappresentazione del codice.

Esempio:

carattere	a	b	c	d	e	f
frequenza	57	13	12	24	9	5







Complessità algoritmo di Huffman: O(n log n)
L'algoritmo di Huffman è gready perchè ad ogni passo costruisce il nodo interno avente frequenza minima possibile.