

Algoritmi e Strutture Dati

▼ Strutture dati fondamentali

Stack & Queue

Sono due tipi di strutture dati fondamentali che modellano collezioni di oggetti.

Le operazioni ammissibili sono:

- inserimento
- rimozione
- iterazione
- verifica se la collezione è vuota

Si differenziano in base alla logica di rimozione degli oggetti:

- Stack ⇒ permette di esaminare e rimuovere l'elemento inserito più di recente (\Logica LIFO)
- Queue ⇒ permette di esaminare e rimuovere l'elemento inserito meno recentemente (\Logica FIFO)

Stack

Stack (linked list)

Uno stack può essere implementato tramite una linked list.

In particolare, la linked list deve mantenere il riferimento al primo nodo della lista (**top dello stack**) e inserire i nuovi elementi **sempre prima** del primo nodo.

Quando si deve effettuare la rimozione sarà prelevato il primo elemento della lista.

Operazione Push:

- salvo il primo nodo in una variabile
- creo un nuovo nodo con l'oggetto da inserire e lo inserisco come primo elemento
- prendo l'elemento precedentemente salvato e lo assegno come successivo del nuovo primo elemento

Operazione Pop:

- salvo l'elemento che devo restituire (quello in prima posizione)
- Assegno l'elemento successivo a quello da restituire come nuovo primo elemento, così facendo elimino il primo nodo
- Restituisco l'elemento precedentemente salvato

Ogni operazione impiega un tempo costante nel worst case

Stack (array)

Proviamo ad implementare uno stack di capacità fissa con un array:

- Usiamo un array $s[]$ per memorizzare N elementi nello stack
- Push \Rightarrow inseriamo un elemento in posizione N
- Pop \Rightarrow rimuoviamo l'elemento in posizione $N-1$

Definiamo:

- **underflow** \Rightarrow lancia un'eccezione se viene effettuato pop su uno stack vuoto
- **overflow** \Rightarrow eccezione se viene effettuato un push su uno stack pieno
- **null item** \Rightarrow possibilità di inserire elementi null nello stack
- **Loitering** \Rightarrow mantenimento di un riferimento ad un oggetto anche quando non necessario

Implementazione con array con ridimensionamento

Abbiamo visto come nella precedente implementazione fosse necessario per il client specificare la dimensione dello stack da creare, ma ciò va contro i principi di encapsulamento e information hiding: il client non deve sapere com'è implementato lo stack.

L'obiettivo è quello di realizzare uno **stack di dimensione variabile**, una prima idea potrebbe essere quella di usare un array che aumenta la dimensione dopo ogni push e la diminuisce dopo ogni pop, ma ciò è troppo dispendioso perché:

- bisogna copiare tutti gli elementi in un nuovo array dopo ogni operazione
- gli accessi all'array per inserire N elementi sono $\sim N^2$

Per questi motivi il resizing dell'array deve essere meno frequente possibile.

Una delle possibilità può essere **raddoppiare** la dimensione dell'array quando è **pieno**, creando così un array di dimensione doppia e copiando gli elementi.

In questo modo il numero di accessi non sarà più quadratico ma lineare

Per la **riduzione della dimensione**:

- dimezzare la size dell'array quando è pieno a metà: troppo dispendioso nel worst case, infatti considerando la sequenza di operazioni push-pop quando l'array è pieno ogni operazione impiega un tempo proporzionale ad N
- **dimezzare la size quando l'array è pieno per 1/4**, così facendo l'array sarà sempre pieno tra il 25% e il 100%

L'analisi ammortizzata è un metodo di analisi di complessità di un algoritmo che viene utilizzato quando il worst case è troppo pessimistico.

Si parte da una struttura dati vuota e si calcola il tempo medio di esecuzione per ogni operazione su una sequenza di operazioni nel worst case.

Partendo da uno stack vuoto: ogni sequenza di M push e pop richiede un tempo proporzionale ad M.

Confronto tra le implementazioni

Linked List:

- ogni operazione richiede un tempo costante nel worst case
- Necessari spazio e tempo extra per i link

Resizing array:

- Non è garantito un tempo di esecuzione costante delle operazioni, solo per un'analisi ammortizzata
- Occupa meno spazio

La scelta dell'implementazione dipende da ciò che si deve realizzare: se si opera su un sistema con poca memoria conviene dare priorità alla complessità spaziale e viceversa.

Queue - implementazione con Linked List

In questo caso si mantiene un **puntatore al primo (first)** e all'**ultimo nodo della lista (last)**

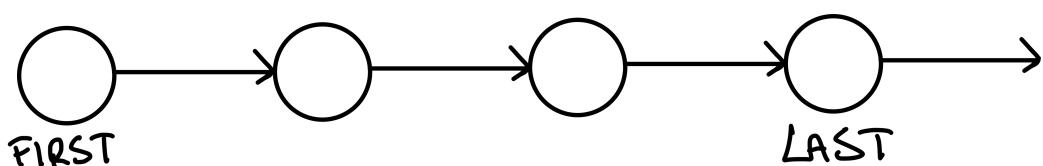
L'elemento **first** è quello che verrà **rimosso** con operazioni di **dequeue** mentre con le operazioni di **enqueue** verranno **inseriti** elementi **dopo** l'elemento **last**.

Il metodo **dequeue** è l'equivalente del pop dello Stack:

- salvo l'elemento da restituire (first, cioè l'elemento in testa alla coda)
- elimino il primo nodo assegnando al suo posto il successivo
- se la coda è vuota \Rightarrow last = first = null
- restituisco l'elemento salvato

Operazione di **enqueue**:

- salvo l'ultimo nodo della coda
- creo il nuovo ultimo nodo contenente l'elemento da inserire in coda e imposto il successivo a null
- se la coda è vuota \Rightarrow referenzio il nodo anche con la variabile del primo nodo (first)
- altrimenti linko il nuovo nodo al termine della lista e lo assegno come nuovo last



Implementazione con array di dimensione fissa

La logica è simile a prima:

- l'elemento in **testa** alla coda è il **primo elemento non nullo (head)**
- l'elemento del **retro** della coda è l'**ultimo elemento non nullo (tail)**

Operazioni:

- **enqueue** ⇒ aggiungiamo un nuovo elemento nella posizione **q[tail]**
- **dequeue** ⇒ rimuoviamo l'elemento in posizione **q[head]**
- Aggiorniamo head e tail con un'operazione di modulo con la capacità dell'array

Anche in questo caso si deve effettuare un ridimensionamento dell'array per evitare underflow e overflow.

Strutture dati generiche

Fino ad ora abbiamo visto come implementare Stack e Queue per stringhe, ma vorremmo implementare strutture dati generiche.

L'approccio (sbagliato) di Java 1.5 e versioni precedenti era quello di implementare una classe per ogni tipo, ma ciò comporterebbe la propagazione di possibili errori (codice duplicato).

Un'altra alternativa sarebbe quella di creare una classe Stack di Object, questa soluzione richiede l'utilizzo di tanti cast che potrebbero portare a degli errori a runtime nel caso in cui si dovesse effettuare tra oggetti divergenti.

L'ultima soluzione, quella ideale, è usare i **generics**, una funzionalità di Java che permette di evitare i cast e scoprire eventuali errori di tipo a tempo di compilazione.

Gestione dei tipi primitivi: per collezionare i tipi primitivi basta usare le loro classi wrapper, così avverrà un autoboxing cioè un cast automatico tra il tipo primitivo e il suo tipo wrapper.

Iterazione, iteratori

L'obiettivo è supportare l'iterazione sugli elementi di uno stack senza rivelarne la rappresentazione interna.

La soluzione consiste nel far implementare l'interfaccia **Iterable** allo stack e di conseguenza implementare un metodo iterator() che restituisce al client un iterator specifico dello Stack.

L'iterator è l'istanza di una classe che implementa l'interfaccia Iterator, cioè implementa i metodi next, hasNext, ecc...

Per quanto riguarda l'implementazione dello Stack tramite array, si utilizza un Iterator che itera gli elementi dell'array in maniera inversa, ciò perché la cima dello Stack è l'ultimo elemento dell'array.

Modifica Concorrente

Se un client modifica la struttura dati durante l'iterazione viene sollevata un'eccezione di tipo ConcurrentModificationException.

Per rilevare questa eccezione l'iterator conta il numero di operazioni di push e pop nello Stack e salva il conteggio nella sottoclassse Iterator all'atto della creazione.

Se quando viene chiamato next e hasNext il conteggio corrente non è uguale a quello memorizzato allora viene lanciata l'eccezione

Applicazione nelle chiamate a funzione

Come un compilatore implementa la chiamata a funzione:

- Chiamata a funzione: operazione di push del local environment e return address
- Return: operazione di pop del return address e local environment

Risoluzione di espressione aritmetiche

Due stack:

- Stack dei valori
- Stack degli operatori

L'algoritmo esegue diverse operazioni a seconda del "carattere incontrato":

- Valore ⇒ push del valore nello stack dei valori
- Operatore ⇒ push dell'operatore nello stack degli operatori
- Chiusura parentesi ⇒ pop di un operatore e pop di due valori, esecuzione dell'operazione e successivo push del risultato nello stack dei valori

L'algoritmo funziona anche se l'operatore si trova dopo due valori invece che in mezzo, le parentesi sono ridondanti

▼ Analisi di algoritmi

L'analisi di algoritmi è effettuata utilizzando il metodo scientifico così come per i fenomeni naturali

Gli algoritmi vengono analizzati per:

- predire performance
- confronto con altri algoritmi
- fornire garanzie

Il tempo di esecuzione di un algoritmo può essere misurato manualmente, in modo automatico (stopwatch) o tramite un'analisi empirica

Analisi Empirica

Viene fornito un input di dimensioni diverse e si misurano i tempi di esecuzione per ciascuno di essi.

Si effettua un plot del tempo di esecuzione in funzione della dimensione dell'input, in particolare il plot può essere anche fatto utilizzando scale logaritmiche per entrambi gli assi.

Nel caso di plot logaritmici si ottiene una retta del tipo: $\ln(T(N)) = b * \ln(N) + c$

Quindi $T(N) = aN^b \Rightarrow$ legge di potenza

Ipotesi del raddoppio

Un modo per stimare b in una legge di potenza consiste nell'eseguire il programma raddoppiando la dimensione dell'input, infine si calcola il rapporto tra $T(2N)$ e $T(N)$ e il relativo log in base 2.

Modelli matematici per il tempo di esecuzione

Il tempo di esecuzione è dato dal prodotto tra la somma dei costi e la frequenza di tutte le operazioni.

Costi delle operazioni:

- le operazioni primitive impiegano un tempo costante che dipende dall'hardware

- le operazioni non primitive possono impiegare un tempo superiore che dipende dalla dimensione di input N

Prima semplificazione \Rightarrow usare un modello di costo sfruttando il tempo di esecuzione delle operazioni di base

Seconda semplificazione $\sim \Rightarrow$ Stima il tempo di esecuzione (o memoria occupata) come funzione della dimensione di input N, ma si ignorano i termini della funzione di ordine inferiore.

Esempio: se la funzione è di ordine 3 allora approssimiamo prendendo solo il termine del terzo ordine.

Classificazione degli ordini di crescita

Se $f(N) \sim c g(N)$ con costante $c > 0$, allora l'ordine di crescita di $f(N)$ è proprio $g(N)$:

- ignoriamo il coefficiente
- ignoriamo i termini di ordine inferiore

È possibile descrivere l'ordine di crescita degli algoritmi più comuni sfruttando il seguente insieme di funzioni:

- 1 \rightarrow costante
- $\log N$ \rightarrow logaritmico
- N \rightarrow lineare
- $N \log N$ \rightarrow linearitmico
- N^2 \rightarrow quadratico
- N^3 \rightarrow cubico
- 2^N \rightarrow esponenziale

▼ Teoria degli algoritmi

Tipi di analisi:

- **Best case:** limite inferiore di costo (lower bound)
 - Determinato dall'input migliore
 - Utile per capire le prestazioni migliori che l'algoritmo può offrire
- **Worst case:** limite superiore di costo (upper bound)
 - Determinato dall'input peggiore

- Utile per conoscere le prestazioni peggiori dell'algoritmo
- **Average case:** costo atteso per un input casuale

Un algoritmo è **ottimale** se **lower bound = upper bound**

In generale quando si analizzano le prestazioni di un algoritmo è possibile abbassare l'upper bound sviluppando un nuovo algoritmo, mentre è più difficile migliorare il lower bound.

Analisi delle prestazioni - Memoria

Consideriamo una macchina a 64 bit con puntatori di 8 byte.

Tipici costi in memoria:

- Tipi primitivi
- Array monodimensionali
- Array bidimensionali

In Java ogni oggetto ha un overhead in memoria di 16 byte, mentre ogni reference occupa 8 byte.

Padding ⇒ ogni oggetto deve occupare in memoria un multiplo di 8 byte pertanto verranno aggiunti un numero di byte tale da ottenere un multiplo di 8.

2 tipi di calcolo:

- **shallow memory usage:** non si prende in considerazione la memoria degli oggetti referenziati
- **deep memory usage:** si considera ricorsivamente anche la memoria degli oggetti referenziati

Sommario

Analisi empirica

Eseguiamo i programmi come esperimento, assumiamo una legge di potenza e formuliamo un'ipotesi sul tempo di esecuzione. Il modello creato ci permette di fare predizioni sul tempo di esecuzione dell'algoritmo.

Analisi matematica

L'analisi viene effettuata contando la frequenza delle operazioni, viene usata la tilde notation per semplificare l'analisi.

Metodo scientifico

Il modello matematico è indipendente dal sistema e si applica anche per macchine non ancora costruite. Viene utilizzata l'analisi empirica per validare i modelli matematici e fare predizioni.

▼ Union-Find

Problema di connessione dinamica

Dato un insieme di N oggetti vogliamo realizzare due operazioni:

- connettere due oggetti
- verificare l'esistenza di un percorso che connette due oggetti

Può essere utile e conveniente nominare gli oggetti da 0 a $N-1$, in questo modo trascuriamo dettagli inutili e possiamo utilizzare una tabella dei simboli per tradurre i nomi in interi e viceversa.

La connessione è una **relazione di equivalenza**, cioè rispetta le seguenti proprietà:

- Riflessiva
- Simmetrica
- Transitiva

Una **componente connessa** è un insieme massimale di oggetti mutuamente connessi.

Vogliamo implementare le seguenti operazioni:

- **Find:** in quale componente connessa si trova l'oggetto p ?
- **Connected:** p e q si trovano nella stessa componente connessa?
- **Union:** rimpiazzare le componenti che contengono p e q con la loro unione

In particolare l'implementazione del metodo connected sfrutta il metodo find verificando che p e q siano nella stessa componente.

Quick Find

È l'approccio Eager (rigoroso)

La struttura dati a disposizione è un array di interi $\text{id}[]$ di lunghezza N che contiene gli id delle componenti connesse

Ogni elemento $\text{id}[p]$ rappresenta l'id della componente che contiene l'elemento p

Operazioni:

- **find** \Rightarrow conoscere l'id della componente che contiene p
- **connected** \Rightarrow verificare se p e d hanno lo stesso id, $\text{id}[p] == \text{id}[d]$
- **union** \Rightarrow unire le due componenti connesse, cioè rendere uguali i due id.
Il problema che si presenta è che si devono cambiare i valori degli altri elementi appartenenti alla stessa componente connessa.

Esempio: se si vuole effettuare la union di 6 e 1, dove $\text{id}[6] = 0$ e $\text{id}[1] = 1$ allora dovrà andare a modificare $\text{id}[6]$ con il valore di $\text{id}[1]$, la modifica non si ferma a $\text{id}[6]$ ma a tutte le componenti che hanno $\text{id} = 0$

https://www.youtube.com/watch?v=o3izwV_joyM&list=PL1fp4FtjD21Vqu2rOMBk28VcOtev7gvpN&index=5

Alcune osservazioni:

- Inizializziamo l'array degli id con numeri crescenti da 0 a $N-1$, quindi effettuiamo N accessi
- Nella find restituiamo l'id dell'elemento cercato
- Nella union è necessario controllare tutti gli elementi che condividono l'id di p e poi cambiarli tutti con l'id di q , quindi in totale avremo al massimo $2N + 2$ accessi ad array.

Prestazioni

Valutiamo le prestazioni prendendo in considerazione gli accessi all'array:

- Inizializzazione $\Rightarrow N$
- Union $\Rightarrow N$
- Find $\Rightarrow 1$

- Connected \Rightarrow 1

Concludiamo che questo algoritmo è dispendioso per operazioni di Union, infatti necessita di N^2 accessi ad array per processare una sequenza di N operazioni di union su N oggetti.

Una caratteristica degli algoritmi con ordine di crescita quadratico è che si ha una bassa scalabilità, anche con hardware prestante.

Quick Union

Approccio lazy (pigro)

La struttura dati è sempre un array di interi id[] di lunghezza N, dove:

- l'elemento id[i] è il genitore di i
- la radice di i è id[id[...]], cioè si continua a cercare finché l'id non cambia

Come prima, l'array è inizializzato sempre con id crescenti da 0 a N

Operazioni:

- find \Rightarrow conoscere la radice dell'elemento p
- connected \Rightarrow verificare se le radici di p e q siano uguali
- union \Rightarrow settare l'id della radice di p uguale all'id della radice di q, così facendo si cambia un solo valore

Alcune osservazioni:

- inizializzazione con numeri crescenti da 0 a N, quindi abbiamo N accessi ad array
- Nella **find** cerchiamo la radice di p andando a controllare gli id dei genitori finché `id[i]==i`, abbiamo un numero di accessi pari alla profondità di i
- Nella **union** cambiamo l'id di p settandolo uguale alla radice di q, abbiamo un numero di accessi pari alla profondità di p e q
- Se `id[i]==i` allora quella è una radice

<https://www.youtube.com/watch?v=-y6cwaj43qA>

Prestazioni:

Valutiamo le prestazioni considerando il numero di accessi all'array:

- Inizializzazione: N
- Union: N (worst case)
- Find: N (worst case)
- Connected: N (worst case)

Confronto tra quick union e find:

- **Quick-find:**
 - operazione di union dispendiosa (N accessi ad array)
 - gli alberi sono piatti, cioè non esistono
- **Quick-union:**
 - Gli alberi possono diventare profondi
 - Operazione di find e connected più dispendiosa (fino ad N accessi)

Possibili miglioramenti

Weighted quick union

Modifichiamo la quick-union in modo da non avere alberi troppo profondi.

A parità di numero di operazioni otterremo alberi di profondità minore e quindi minor numero di operazioni per find.

Utilizziamo la stessa struttura della quick-union ma aggiungiamo un array di interi sz[] che memorizza il numero di elementi nell'albero.

Operazioni:

- find e connected sono invariate
- union viene modificata:
 - collegiamo la radice dell'albero più piccolo alla radice dell'albero più grande
 - aggiorniamo l'array sz[]

Prestazioni

- Find \Rightarrow tempo di esecuzione proporzionale alla profondità di p, quindi passiamo ad un **ordine di crescita logaritmico** perché possiamo affermare con certezza che la profondità di qualsiasi nodo x è al più è $\lg(x)$

- Union \Rightarrow tempo costante, date le radici

Path Compression

In questo caso si modifica il metodo find()

2 approcci

- **two pass:** si aggiunge un secondo loop alla find che imposta gli id[] dei nodi esaminati alla radice
- **one pass (path halving):** nello stesso loop, con una sola linea di codice in più si assegna iterativamente come $id[i]$ l'id del nodo padre di i

WQUPC

L'unione del **Weighted Quick Union** e **Path Compression** crea un nuovo algoritmo che data una struttura dati vuota, qualsiasi sequenza di **M** operazioni union-find su **N** oggetti produce un numero $\leq c (N + M\lg^*(N))$ accessi ad array.

\lg^* \rightarrow logaritmo iterato

$M\lg^*N$ è un approssimazione della sequenza union find, considerando path compression e weighted quick union.

In teoria l'algoritmo **WQUPC** **NON** è **lineare** ma nella pratica sì, non è possibile creare un algoritmo che abbia un tempo di esecuzione lineare

Una delle applicazioni dell'algoritmo **WQUPC** è nella teoria della percolazione.

Consideriamo un modello astratto:

- Griglia NxN di siti
- ogni sito è **aperto** con probabilità p e **bloccato** con probabilità $1-p$
- il sistema percola se e solo se top e bottom della griglia sono connessi da siti aperti

Per grandi valori di N esiste un valore di soglia p^* per il quale:

- $p > p^*$ allora è molto probabile che ci sia percolazione
- $p < p^*$ allora è molto probabile che ci sia percolazione

Montecarlo Simulation

È possibile utilizzare l'union-find per verificare se un sistema percola

Si inizializzano tutti i siti (nello stato bloccato) in una griglia NxN, si dichiarano siti aperti in modo random finché il top e il bottom sono connessi e si stima il valore p^*

I passaggi sono:

- creare un oggetto per ogni sito da 0 a N^2-1
- se esiste un collegamento tra più siti aperti, allora faranno parte della stessa componente连通 e quindi avranno lo stesso id
- il sistema **percola** se ogni sito nella riga di bottom è connesso ad almeno un sito della riga di top

▼ Algoritmi di ordinamento

L'obiettivo di un algoritmo di ordinamento è quello di ordinare qualsiasi tipo di dato secondo il concetto di **ordine totale**, cioè una relazione binaria \leq che soddisfa le seguenti proprietà:

- Antisimmetria \rightarrow se $v \leq w$ e $w \leq v \Rightarrow v = w$
- Transitività \rightarrow se $v \leq w$ e $w \leq x \Rightarrow v \leq x$
- Totalità $\rightarrow v \leq w$ oppure $w \leq v$ oppure entrambe

Per garantire il funzionamento di un sort() con qualsiasi tipo di dato, si utilizza il concetto di **callback**.

Il metodo sort() chiama il metodo compareTo() degli oggetti che a loro volta devono implementare Comparable.

Il compareTo() definisce un ordine totale e restituisce un `int`, in particolare:

- negativo se $v < w$
- positivo se $v > w$
- zero se $v = w$

Selection Sort

L'array da ordinare viene scansionato da sx verso dx in modo che a sinistra della posizione i-esima troviamo il final order, mentre a destra si effettua uno **swap** con l'elemento minimo.

Questo algoritmo richiede $\sim \frac{N^2}{2}$ confronti, anche se i dati in input sono già ordinati, invece lo swap è lineare.

In ogni caso il tempo di esecuzione è **quadratico**

Si dice che il selection sort non è **adattivo** perché il tempo di esecuzione dipende solo dalla dimensione dell'array.

È **instabile** perché **NON** preserva le posizioni relative di elementi uguali all'interno dell'array

Il **Selection Sort** è particolarmente utile quando si devono ordinare dati con **record molto grandi** (quindi spostarli fisicamente è costoso) e **chiavi piccole** (il confronto tra le chiavi è rapido).

In questi casi, poiché il numero di **scambi** è ridotto e i confronti sono relativamente economici, il Selection Sort può essere preferibile rispetto ad altri algoritmi che eseguono più spostamenti o scambi di dati.

<https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/2c1083aa-a768-483e-81ea-752a8a493cd1/21DemoSelectionSort.mov>

Insertion sort

Consiste nel navigare l'array fino a quando non si trova l'elemento in posizione $i+1$ minore dell'elemento in posizione i e in tal caso si effettua uno **swap** se l'elemento in posizione j è minore dell'elemento in posizione $j-1$.

Complessità:

- **best case** → lineare
- **average** → quadratica
- **worst case** → quadratica

Quindi per ordinare un array con elementi casuali l'algoritmo usa in media $\sim \frac{N^2}{4}$ confronti e $\frac{N^2}{4}$ scambi.

Se l'array è parzialmente ordinato allora il tempo di esecuzione diventa lineare, per cui l'algoritmo è **adattivo**.

È anche **stabile**

È possibile effettuare dei miglioramenti

- **dimezzare gli scambi:** si effettuano degli shift degli elementi invece di effettuare scambi
- **binary insertion sort:** conviene utilizzare la ricerca binaria per trovare il punto di inserzione

Dato che il tempo di esecuzione è **quadratico** sia nel worst che nell'average case, l'algoritmo viene usato per array piccoli.

<https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/d36576aa-8e1d-49eb-814c-2ddebeee8445/21DemoInsertionSort.mov>

Shell sort

È un'estensione dell'algoritmo insertion sort, l'idea è quella di spostare gli elementi più di una posizione alla volta facendo un **h-sorting** dell'array e non soltanto con quelli adiacenti.

Un array h-sorted è un array costituito da sottosequenze ordinate.

Questo algoritmo esegue un **h-sort** dell'array per sequenze di valori decrescenti di h.

Ad ogni iterazione si effettua lo **scambio** dell'elemento in posizione i con ogni elemento più grande posto h posizioni alla sua sinistra.

L'h-sort è realizzato con un insertion sort con **passo** di lunghezza h.

Un array è **h-sorted** se per ogni i si ha che $a[i-h] \leq a[i]$

<https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/80f876ee-9aea-4514-adfe-7b99063f386e/21DemoShellSort.mov>

La scelta ottimale della legge di incremento di h è $3x + 1$. ($h < \frac{N}{3}$)

Prestazioni:

- **best case** $\rightarrow N \log_3(N)$
- **worst case** $\rightarrow N^{\frac{3}{2}}$

Questo algoritmo è utile se utilizzato su array di dimensioni non elevate oppure se utilizzato in sistemi embedded perché non utilizza memoria aggiuntiva.

L'obiettivo da raggiungere è avere un algoritmo di ordinamento che abbia un ordine di crescita **linearitmico** nel caso **peggiore e average**, mentre **lineare** nel caso **migliore**.

Non è possibile scendere al di sotto di un ordine di crescita lineare perché si deve come minimo accedere a tutti gli elementi dell'array per poterlo ordinare.

Shuffling e ShuffleSort

L'obiettivo è **mescolare** un array, ossia cambiare l'ordine degli elementi in modo da ottenere una **permutazione uniformemente casuale** di questi ultimi, cioè tutte le permutazioni sono equiprobabili.

ShuffleSort prevede di generare un numero reale random per ogni elemento dell'array per cui è importante che la funzione random sia ben implementata in modo da avere numeri più casuali possibili (problema dei numeri pseudocasuali) e in base a questi ordinare l'array.

Una possibile soluzione potrebbe essere creare una **casualità** nel compareTo()

Knuth Shuffle

Permette di effettuare uno shuffle in un tempo lineare.

In pratica ad ogni iterazione di i si sceglie un intero random r (compreso tra 0 e i) e si scambia $a[i]$ con $a[r]$

Nella sua implementazione non sono presenti loop annidati per cui è lineare.

<https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/b18574df-6771-4c8c-995f-b935548414be/21DemoKnuthShuffle.mov>

Merge Sort

Quest'algoritmo si basa sul principio **divide et impera**, in particolare effettua 3 operazioni:

- Dividere a metà l'array
- Ordinare ricorsivamente ogni metà
- Unire (merge) le due metà

Dopo aver diviso a metà l'array, si confrontano i minimi dei subarray e il minimo tra questi viene inserito nella prima posizione disponibile dell'array di appoggio, si fa scorrere il puntatore del sub-array dell'elemento minimo e si riparte con il nuovo confronto finché gli elementi dell'array non terminano

Quest'algoritmo ha prestazioni **linearitmiche** in tutti i casi possibili, in pratica verranno fatti $\frac{N}{2}$ confronti per ogni metà dell'array e N confronti dovuti alle operazioni di merge

Si forma un albero di ricerca binaria, in cui ogni nodo ha due figli, la navigazione in profondità di quest'albero è $\lg(N)$.

Questo valore va moltiplicato per N, cioè le operazioni di merging.

<https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/926f65d2-3734-4f07-9750-af9a46dc106c/22DemoMerge.mov>

Complessità spaziale

È il primo algoritmo **non in place** che analizziamo

Un algoritmo di ordinamento è **in place** se utilizza uno spazio di memoria extra $\leq c \log N$

Merge sort **NON** è in place perché utilizza uno spazio **proporzionale ad N** in quanto viene utilizzato un array ausiliario aux[] che deve essere di lunghezza N per poter effettuare l'operazione di merge finale tra i due subarray principali

Possibili miglioramenti

- Per subarray di piccole dimensioni si effettua l'insertion sort, evitando così l'overhead del merge sort. Si dice che si effettua un **cut off ad insertion sort**

- Nel caso in cui abbiamo a che fare con i due sotto array già ordinati, non effettuiamo il merge, per fare ciò si controlla che $a[mid] < a[mid + 1]$
- Eliminare la copia ausiliaria dell'array, cioè usa l'array di supporto per ordinare direttamente l'array, questa soluzione risolve il problema della complessità temporale ma non spaziale.

Merge sort di tipo bottom-up

Questo merge sort è una soluzione per evitare l'overhead dovuto alla ricorsione.

L'idea di base è quella di scorrere l'array ed effettuare il merge di subarray di dimensione $sz=1$, ripetere per subarray di dimensione crescente $sz = 2, 4, 8, \dots$

In realtà anche non usando la ricorsione, quest'implementazione risulta essere circa il 10% più lenta

Natural merge sort

Un'altra idea per migliorare il merge sort è quella di **identificare** l'ordine pre esistente, in particolare si identificano quali sono le sequenze già in ordine e si effettuano i merge.

Trade off: si effettuano meno passi, ma si hanno più confronti per identificare le sequenze preordinate esistenti.

Tim sort

È un algoritmo che implementa l'idea del **natural merge sort** e utilizza un **insertion sort binario** per le esecuzioni iniziali (se necessario).

Si ottiene così un tempo di esecuzione lineare su molti array con ordinamenti pre esistenti.

Complessità di ordinamento

La complessità computazionale è un framework utile a studiare l'efficienza degli algoritmi.

Le caratteristiche da analizzare sono:

- modello computazionale → operazioni ammissibili
- modello di costo → numero di operazioni
- upper bound → costo garantito da un algoritmo nel caso peggiore

- lower bound \rightarrow limite minimo di costo tra tutti gli algoritmi che risolvono il problema

Algoritmo ottimale \rightarrow lower bound e upper bound coincidono

In questo caso vogliamo verificare se il merge sort può essere definito ottimale

Gli algoritmi di ordinamento si basano sui decision tree perché accedono alle informazioni mediante confronti.

Ogni nodo corrisponde ad ogni confronto possibile nell'algoritmo, l'altezza dell'albero corrisponde al numero di confronti effettuati nel worst case.

Ogni nodo foglia corrisponde ad un possibile ordinamento degli elementi e sono $N!$, cioè tutte le possibili permutazioni.

Allora dimostriamo che qualsiasi algoritmo di ordinamento basato su confronti deve utilizzare un numero di **confronti** almeno pari a $\log(N!) \sim N \lg N$ nel **worst case**.

Possiamo affermare che un albero di ricerca binaria ha minimo $N!$ e massimo 2^h foglie:

$$N! \leq \text{numero foglie} \leq 2^h$$

Applichiamo \lg ad ogni membro:

$$\lg(N!) \leq h$$

Per la formula di Stirling possiamo scrivere: $\lg(N!) \sim N \lg(N)$

Possiamo concludere che nel nostro caso, per il problema dell'ordinamento abbiamo:

- modello computazionale: Decision Tree
- modello di costo: numero di operazioni \rightarrow numero di confronti
- upper bound: $\sim N \lg N$ per il merge sort
- lower bound: $N \lg N$

Quindi il merge sort è ottimale rispetto al numero di confronti perché coincidono lower e upper bound, ma non è ottimale rispetto allo spazio utilizzato.

Comparator

Questa interfaccia permette di utilizzare strategie di ordinamento alternative rispetto al compareTo

Il metodo utilizzato da Comparator è **compare** e anche questo metodo deve rispettare la relazione di ordine totale (antisimmetria, transitività, totalità)

In Java per utilizzare un comparator con l'algoritmo di sort standard è sufficiente creare un oggetto che implementa Comparator e passarlo come secondo argomento della funzione.

Volendo utilizzare un Comparator, nel metodo sort va passato un array di Object e non Comparable.

Stabilità

Un algoritmo si definisce **stabile** se preserva l'ordinamento iniziale nel caso di elementi con **chiavi uguali**.

Quindi anche dopo l'implementazione del Comparator va preservata la stabilità.

Esempi:

- Insertion sort → stabile
- Selection e Shell → non stabile, perché a causa degli spostamenti di elementi a lunga distanza è possibile che NON venga preservato l'ordine relativo degli elementi uguali
- Merge sort è stabile perché l'operazione di merge prende le chiavi uguali sempre dal **subarray sinistro**

Quick Sort

È uno dei principali algoritmi di ordinamento, ideato da Tony Hoare.

Passi principali:

- **Shuffle** dell'array per cercare di evitare un array ordinato
- **Partizionare** l'array in modo che per un certo valore di j:

- $a[j]$ è in place (elemento pivot), cioè non ci sono elementi **maggiori alla sinistra** di $a[j]$ e NON ci sono elementi **minori alla destra**
- **Ordinare** ogni sub-array ricorsivamente

L'algoritmo si basa su due concetti: **in place** e **ricorsione**.

La ricerca dell'elemento **in place** avviene nel metodo di **partitioning**

Si utilizzano due variabili per scorrere l'array, in particolare **i** da sx verso dx e **j** da dx verso sx.

Si parte considerando come **pivot** l'elemento in posizione 0 e si scorre l'array finché i due puntatori non si **incrociano**

Si **incrementa** i finché si trovano elementi **minori** del pivot

Si **decrementa** j finché si trovano elementi **maggiori** del pivot

Quando si trova un elemento minore e uno maggiore del pivot si effettua lo **scambio** tra $a[i]$ e $a[j]$

Quando i e j si **superano**, basterà scambiare il pivot con l'elemento in posizione j e continuare ad ordinare ricorsivamente i due sub-array.

La ricorsione viene utilizzata nel metodo **sort privato**. In pratica, dopo aver partizionato l'array, viene richiamato il metodo sort sia per il subarray sinistro che per quello destro

Il **numero di confronti** per partizionare un array di lunghezza N è $\sim N \lg N$

In generale, è possibile usare un array extra per rendere l'operazione di partizionamento più semplice e stabile, ma questa soluzione non vale il costo.
Conviene usare un approccio **in place**

Chiavi uguali

Quando ci sono elementi duplicati conviene **fermare la scansione** (del metodo partitioning) sulle chiavi uguali a quella del pivot in modo da distribuire equamente i due subarray a sinistra e a destra dei duplicati

Prestazioni

Da analisi empiriche il quick sort migliora ulteriormente le prestazioni rispetto al merge sort, oltre a dare il vantaggio di una complessità spaziale minore.

Tuttavia, il quick sort ha prestazioni peggiori se ci sono duplicati, solitamente in questo caso si cambia tipo di partizionamento, cioè si posizionano vicine tra loro le chiavi duplicate e si partiziona a destra e sinistre di queste.

Best case: l'elemento pivot finisce sempre nella posizione a metà dell'array, si ricade nel caso del merge sort, cioè un numero di confronti $\sim N \lg N$

Worst case: si ha quando l'array è **già ordinato** perché il pivot sarebbe già in place e a destra ci saranno $N-1$ elementi mentre 0 a sinistra. Ne consegue che il numero di confronti è $\frac{1}{2}N^2$

Non si possono garantire prestazioni linearitmiche con il quick sort anche se il worst case è poco probabile.

Il **merge sort** garantisce complessità temporale linearitmica mentre il **quick sort** garantisce complessità spaziale costante.

Average case: è stato dimostrato che nel caso medio il quick sort effettua il 39% di confronti in più ma fa un minor spostamento di dati. Quindi nel complesso il quick sort ha prestazioni migliori

In place → il quick sort **NON** supera in termini di complessità spaziale un livello superiore a $c \log(N)$

Questo perché l'unica componente dell'algoritmo ad usare spazio extra è la parte ricorsiva.

Dato che la profondità dei subarray che si creano è al massimo logaritmica, possiamo dire che lo spazio usato nello stack del programma per garantire tale ricorsione sarà sicuramente minore di quello linearitmico

Miglioramenti

- **Cutoff a insertion sort** → come già accadeva con il merge, anche in questo caso un miglioramento di facile implementazione è il cutoff ad insertion sort. In particolare si passa all'insertion sort quando si formano subarray di circa 10 elementi, evitando così l'overhead di quick sort
- **Scelta del pivot** → È importante ragionare sulla scelta del pivot dato che quest'ultimo è un elemento di fondamentale importanza per la complessità dell'algoritmo.
Sappiamo che il best case si ha quando si sceglie come pivot l'elemento **mediano**

Per aumentare la probabilità, si prendono randomicamente 3 sample e si calcola la mediana. L'elemento mediano tra i 3 viene scelto come pivot e portato in prima posizione

Gestione delle chiavi duplicate

Abbiamo visto che l'obiettivo del sorting in presenza di chiavi duplicate è quello di raggruppare gli elementi con chiavi uguali.

Il quick sort ha numerosi problemi quando deve analizzare array con un numero di elementi distinti minore della dimensione.

Finora abbiamo visto che la funzione di partizionamento si ferma quando incontra elementi uguali. Se consideriamo un array di chiavi tutte duplicate:

- se l'algoritmo non si ferma → numero di confronti $\sim 1/2 N^2$
- se l'algoritmo si ferma sulle chiavi uguali → numero di confronti $\sim N \lg N$

La scelta migliore sarebbe mettere insieme tutte le chiavi uguali in place.

Possiamo concludere che il **quick sort** migliora le prestazioni del merge sort, ad eccezione di quelle nel worst case, è in place ma NON è stabile.

Algoritmo della Bandiera Olandese (3-way partitioning)

Dato un array di n bucket, ognuno contenente un colore della bandiera olandese, si vuole ordinare l'array per colore.

Le operazioni consentite sono:

- $\text{swap}(i,j) \rightarrow$ scambia il colore nel bucket di i con quello di j
- $\text{color}(i) \rightarrow$ verifica il colore nel bucket i

Requisiti:

- esattamente n chiamate di $\text{color}()$
- al massimo n chiamate di $\text{swap}()$
- spazio extra costante

L'idea è di applicare il 3-way partitioning, cioè partizionare l'array in 3 parti tali che:

- gli elementi compresi tra le posizioni `lt` e `gt` (zona centrale) siano **uguali** all'elemento pivot
- non ci siano elementi più grandi alla sinistra di `lt` ⇒ tutti gli elementi con chiave minore del pivot
- non ci siano elementi più piccoli alla destra di `gt` ⇒ tutti gli elementi con chiave maggiore del pivot

Multi-pivot

A partire da java 7, il quick sort presente in libreria è stato sostituito dal quick sort con **2 elementi pivot**

Dual pivot quick sort

Ci sono due pivot `p1` e `p2`:

- `p1` dovrà essere nella posizione finale `lt` e alla sua sinistra ci saranno gli elementi con chiave **minore**
- `p2` dovrà essere nella posizione finale `gt` e alla sua destra ci saranno gli elementi con chiave maggiore

Le chiavi comprese tra `p1` e `p2` saranno nella zona centrale, se `p1 = p2` allora l'algoritmo diventa quello della bandiera olandese.

Three pivot quick sort

Si utilizzano 3 pivot

La tecnica multi-pivot ha migliori prestazioni rispetto a quella a singolo pivot perché permette meno **cache miss**, in particolare si riduce il coefficiente N della funzione linearitmica.

La libreria standard di Java prevede il metodo `Array.sort()` overloaded, per i tipi primitivi viene utilizzato il dual pivot quick sort, mentre per i tipi referenziati viene usato il timsort

▼ Selection

Si intende la famiglia di algoritmi che dato un array di N elementi seleziona il k -esimo.

Questo significa che dato un array di elementi non ordinati, si cerca di rintracciare l'elemento in posizione k dell'array se fosse ordinato

Quick Select

È un algoritmo che può essere considerato come una variante del quick sort.

Quest'algoritmo **NON** restituisce l'intero array ordinato, ma solo la posizione dell'elemento in posizione k dell'array ordinato.

L'idea è di **partizionare** l'array in modo che l'elemento $a[j]$ sia nella posizione corretta, ossia:

- **nessun** elemento alla sinistra di esso sia maggiore
- **nessun** elemento alla destra sia minore

Si ripete il procedimento in subarray fin quando j (dove finisce il pivot) diventa uguale a k .

In pratica, se il pivot supera l'elemento in posizione k allora è possibile **non** considerare il subarray di destra, viceversa si esclude quello di sinistra.

Prestazioni

Possiamo dire che nell'average case la quick select ha prestazioni **lineari**.

Intuitivamente ogni passo di partizionamento divide l'array approssimativamente a metà $\sim 2N$ confronti.

In generale, gli algoritmi di **selezione basati sui confronti** hanno un tempo di esecuzione nel **worst case lineare**

In conclusione possiamo dire che non ci si dovrebbe accontentare di un algoritmo con prestazioni lineari (anche nel worst case), ma finché non verrà scoperto conviene utilizzare la quick select se non c'è bisogno di un ordinamento completo.

▼ Collezioni

In generale una **collection** è un tipo di dato che colleziona gruppi di altri dati

I tipi principali sono:

| data type | core operations | data structure |
|----------------|-----------------------|---------------------------------------|
| stack | PUSH, POP | <i>linked list, resizing array</i> |
| queue | ENQUEUE, DEQUEUE | <i>linked list, resizing array</i> |
| priority queue | INSERT, DELETE-MAX | <i>binary heap</i> |
| symbol table | PUT, GET, DELETE | <i>binary search tree, hash table</i> |
| set | ADD, CONTAINS, DELETE | <i>binary search tree, hash table</i> |

▼ Code a priorità - priority queue

Le operazioni sono le stesse delle queue ⇒ inserimento e rimozione.

Ciò che le distingue dalle code normali è che la **rimozione** rimuove l'elemento con **priorità massima**.

Creare una struttura dati che svolge le attività della coda a priorità può essere oneroso se non si usano i giusti elementi, infatti:

- utilizzare un array **non** ordinato porta ad effettuare N confronti nel worst case dato che potrebbero essere confrontati tutti gli elementi, quindi sarebbe un problema dal punti di vista del remove
- utilizzare un array **ordinato** potrebbe essere un problema per l'insert

Quindi, le prestazioni migliori si hanno con un array **parzialmente** ordinato

Una coda a priorità viene implementata utilizzando come struttura dati un **binary heap**

Binary Heap

È la rappresentazione di un **binary tree** sotto forma di **array**.

La caratteristica fondamentale del binary tree è che cresce di altezza ogni volta che il numero di elementi è una potenza di 2.

Un albero è **perfettamente bilanciato** se l'altezza del sottoalbero sx è uguale all'altezza del sottoalbero dx (ad eccezione dell'ultimo livello)

Se un albero è bilanciato ed ha N elementi, allora avrà **altezza** pari a $\lg N$

Un **binary heap** è un albero binario completo **heap ordered**, cioè:

- ogni nodo rappresenta una chiave
- il nodo **genitore** di una chiave **NON** può essere minore delle chiavi dei nodi figli

La rappresentazione con array è caratterizzata da:

- **indici** che partono da 1
- i **figli** del nodo **k** sono nelle posizioni $2k$ e $2k + 1$
- il **genitore** di un nodo **k** si trova nella posizione $k/2$ (vale per pari e dispari)

L'elemento con chiave maggiore si trova in posizione $a[1] \Rightarrow$ **radice dell'albero binario**

Le **operazioni** consentite sono **inserimento** e **rimozione**, non sono banali perché si deve rispettare l'heap order.

Inserimento

Quando si inserisce un nuovo elemento, esso viene inserito inizialmente nell'ultima posizione dell'array.

Se l'elemento viola l'heap order (cioè è più grande del nodo genitore) allora si effettua uno **swim-up (promozione)** dell'elemento

Promozione:

- si confronta il nodo inserito con il genitore, se è maggiore di questo allora si effettua uno scambio
- si continua a confrontare fino a che non si verifica la proprietà di heap order

Il **costo** della promozione è pari a $1 + \lg N$ perché l'altezza dell'albero è $\lg N$, quindi nel worst case andrebbe attraversato l'intero albero in profondità.

È dovuto al fatto che inserendo il nuovo nodo si potrebbe raggiungere un numero che è una potenza di 2 e quindi si genera un nuovo livello dell'albero.

Rimozione

Con quest'operazione viene prelevato l'elemento con priorità maggiore, la rimozione può portare ad una violazione dell'heap order, in particolare può accadere che una **chiave** diventa minore di almeno uno dei **suoi figli**.

In questo caso, va eseguita un'operazione di **demotion**, l'operazione che porta il nodo **genitore** ad essere declassato come **nodo figlio**

Passi per rimuovere un elemento:

- si **scambia** l'elemento **radice** con quello in **ultima posizione** e si **rimuove** l'ultimo elemento
- si effettua il **sink down** (demotion) della nuova radice, cioè:
 - si confronta il nodo con i due nodi figli e si effettua uno **scambio** con il nodo figlio maggiore
 - si continuano i confronti finché il nodo non è maggiore di entrambi i figli

Loitering → si riferisce alla situazione in cui un oggetto è ancora presente nella memoria, nonostante non sia più accessibile o utilizzato nel programma. L'oggetto è abbandonato nella memoria senza essere liberato dal garbage collector.

Prestazioni

Con il binary heap sappiamo con certezza che migliorano le prestazioni rispetto ad avere un array ordinato e/o non ordinato, dove le prestazioni sono lineari.

Il binary heap garantisce un tempo di esecuzione con ordine di crescita **logaritmico** sia per la promotion che per la demotion.

In entrambe le operazioni si **naviga** l'albero in profondità per cui nel worst case il costo sarà $\sim \lg N$

L'operazione di **restituzione** dell'elemento **massimo** è costante, perché è sempre l'elemento in posizione 1.

Rimozione di un elemento random dal binary heap

L'obiettivo è rimuovere un elemento qualsiasi in un tempo **logaritmico**.

L'algoritmo è uguale a quello di rimozione del massimo:

- si sceglie un indice r compreso tra 1 e n
- si scambia $pq[r]$ con l'elemento in ultima posizione e si effettua il sink down (se l'elemento è minore di almeno uno dei nodi figli) o lo swim up (se l'elemento è maggiore del nodo padre)

Possibili miglioramenti

- **Half exchange** → questo metodo riduce gli accessi in memoria

- **Euristica bounce di Floyd** → si evitano i sink down preferendo gli swim up, in questo modo si fanno meno confronti ma più scambi
Viene fatto il sink della radice fino all'ultima posizione (1 confronto per nodo) e poi si esegue lo swim up della chiave
- **Heap multiway** → un ultimo miglioramento è quello di basarsi non più su alberi binari ma su un albero **d-way** (ogni nodo ha al massimo d figli).
La regola di base è sempre la stessa, cioè la chiave del nodo genitore è sempre maggiore delle chiavi dei figli.
Per poter mantenere la rappresentazione ad array occorre sempre un numero di figli costanti.
Con questa soluzione, l'**altezza** dell'albero sarà: $\log_d N$, di conseguenza, il numero di confronti nel worst case sarà peggiore rispetto all'albero binario:
 - **insert** $\sim \log_d N$
 - **delete max** $\sim d \log_d N$
 È importante scegliere un valore di d non troppo elevato per minimizzare il numero di confronti, un valore tipico è 4.

Gestione di underflow e overflow

Underflow → si lancia un'eccezione quando si tenta di eliminare un nodo da una **priority queue** vuota.

Overflow → si aggiunge un **costruttore senza argomenti** e si utilizza un **array ridimensionabile**, ciò porterà ad un tempo ammortizzato di $\log N$ per operazione.

Immutabilità delle chiavi

Potrebbero esserci problemi se un client modifica il valore delle chiavi all'interno della struttura dati.

Per evitare il problema, si rendono le chiavi **immutabili**

Quindi si utilizzano come chiavi oggetti String, wrapper di tipi primitivi o oggetti di classi che usano variabili di istanza dichiarate come private final.

Heap sort

È un algoritmo di ordinamento basato su un binary heap.

Vediamo l'heap sort **in place**

L'heap sort si compone di 3 passi:

- **generare** un albero binario
- **ordinare** l'albero come un max heap
- **prelevare** ricorsivamente il nodo radice

Costruzione del max heap

Viene costruito con una logica bottom up

Passaggi:

- si parte con $k = \frac{N}{2}$ in quanto si considerano i nodi foglia come dei mini heap di dimensione 1
- si effettua il sink down
- si passa al nodo in posizione $k - 1$ e si ripetono i passi finché l'albero non sarà ordinato

Alla fine del ciclo, l'array sarà **max heap o rdered**

Ora passiamo alla rimozione del nodo

https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/8fbe9371-672e-4867-b2de-7b8540e28f89/24De_moHeapsort.mov

Sort Down

Si sfrutta la costruzione del max heap, cioè viene prelevato l'elemento massimo ogni volta.

Per il sort down si eseguono ripetutamente i seguenti passi:

- si scambia la radice con l'ultimo elemento (non si elimina, quindi l'algoritmo è *in place*)
- si fa il sink down della radice

Alla fine si ottiene l'array ordinato

Prestazioni

Per la fase di costruzione del max heap si effettua un numero di scambi al più pari a N , mentre un numero di confronti al più $2N$.

Si dimostra che nel **worst** case, heapsort utilizza un numero di confronti e scambi $\leq 2N \log N$

Quindi le prestazioni sono **linearitmiche** dal punto di vista temporale, mentre è in place dal punto di vista spaziale dato che non utilizza spazio extra

CONFRONTO CON I MIGLIORI SORTING

| | MERGE | QUICK | HEAP |
|-------------|------------|-------|------------|
| IN PLACE | ✗ | ✓ | ✓ |
| COMPLESSITÀ | $N \log N$ | N^2 | $N \log N$ |

In conclusione, heapsort è ottimale sia dal punto di vista temporale che spaziale, ma:

- presenta cicli più lunghi
- non usa in maniera efficiente la cache
- non è stabile (come il quick) a causa dei tanti spostamenti

Intro Sort

È una particolare implementazione utilizzata in alcune librerie che mette insieme diversi algoritmi per cercare di prendere i punti di forza di ognuno.

Quest'algoritmo esegue il **quicksort**, però:

- effettua un cutoff ad **Heap Sort** se la profondità dello stack di chiamate ricorsive supera $2 \lg N$
- effettua un cutoff ad **insertion sort** se $N = 16$

▼ Symbol Tables - Tabelle dei simboli

Le tabelle dei simboli sono anche conosciute come **mappe, dizionari o array associativi**.

Sono una generalizzazione del concetto di chiave valore dell'array in cui la chiave è l'indice e il valore è l'elemento.

La symbol table è una struttura dati composta da coppie **chiave-valore** in cui possiamo:

- inserire un valore associato ad una specifica chiave
- data una chiave, cercare il valore corrispondente

Un'applicazione tipica delle tabelle dei simboli è utilizzata per il DNS

API di una symbol table

L'astrazione utilizzata è quella di un array associativo, cioè si associa un solo valore ad ogni chiave.

Questa classe si basa su alcune convenzioni:

- i valori **NON** sono null
- il metodo `get()` ritorna null se la chiave **NON** è presente
- il metodo `put()` sostituisce un vecchio valore con un nuovo valore

Chiavi e valori

I valori possono essere di **qualsiasi tipo**

Invece le chiavi devono essere necessariamente univoche.

Per i tipi delle chiavi facciamo alcune assunzioni per confrontarle:

- se assumiamo chiavi di tipo **comparable**, sfruttiamo il `compareTo()`
- se assumiamo chiavi di tipo generico, sfruttiamo `equals()` per testarne l'uguaglianza e `hashCode()` per codificarle

Una best practice è quella di utilizzare **chiavi di tipi immutabili** (wrapper di tipi primitivi, String, ecc)

Test di uguaglianza - metodo equals()

Tutte le classi Java ereditano dalla superclasse Object() il metodo equals()

Il requisito di base dell'implementazione di tale metodo richiede di rispettare le seguenti proprietà:

- **Riflessività:** `x.equals(x)` è vero
- **Simmetria:** `x.equals(y) ⇒ y.equals(x)`
- **Transitività:** if `x.equals(y)` AND `y.equals(z)` allora `x.equals(z)`
- **Non-null:** `x.equals(null)` è falso

L'implementazione di default presente nella classe Object prevede un test di uguaglianza basato sull'identità, ossia verificare l'uguaglianza dei riferimenti (operatore `==`)

Nelle altre classi tale metodo è ridefinito in maniera personalizzata tramite il meccanismo di **overriding**.

Inoltre, è importante rendere il compareTo consistente con l>equals.

Implementazioni

Con linked list non ordinata

L'implementazione più facile da usare è questa, in cui in ogni nodo della lista abbiamo:

- riferimento al **valore**
- riferimento alla **chiave**
- **puntatore al nodo successivo**

Le operazioni sono:

- **ricerca:** cerco le chiavi fino a che non trovo quella desiderata e restituisco il valore
- **inserimento:** cerco tra le chiavi se è già presente, se non c'è alcun match allora inserisco la chiave in testa alla linked list

Si intuisce che questo metodo di inserimento non è il migliore perché per **inserire** un elemento bisogna navigare l'intera linked list e questo porta ad avere una **complessità lineare**

In entrambe le operazioni occorre una **ricerca sequenziale delle chiavi**

L'insert ha un costo lineare garantito (sia nel worst case che nell'average case)

Anche la **search** ha un costo lineare garantito, ma nell'average case è $N/2$

Con array ordinato

L'idea migliore è quella di utilizzare un array ordinato di coppie chiave valore per avere la possibilità di utilizzare la **ricerca binaria**, la quale ha un costo **logaritmico** e non lineare.

Avremo 2 array: **uno per le chiavi e uno per i valori**.

In particolare per ogni i avremo che $\text{val}[i]$ sarà l'elemento associato alla chiave $\text{key}[i]$.

Funzione ausiliaria di rank \Rightarrow calcoliamo quante chiavi $< k$

Nel codice implementativo si utilizza compareTo e non equals.

Questa implementazione garantisce un costo **logaritmico** per l'operazione di **search** (sia worst che average).

Il problema di questa struttura si ha con l'inserimento, dove il costo resta lineare (N worst case e $N/2$ nell'average) dato che bisogna preservare l'ordine delle chiavi, per cui quando si inserisce una nuova chiave bisogna fare uno **shift** di tutte le chiavi maggiori.

Vantaggi

Con questo tipo di implementazione vengono migliorate molte delle operazioni, che avrebbero prestazioni in ogni caso **lineari** con la linked list.

Con l'array posso:

- vedere qual è la chiave minima, in quanto essa si trova in $\text{pos}[0]$
- vedere qual è la chiave massima e minore rispetto alla chiave che io specifico
- andare a vedere qual è la settima chiave partendo dal primo
- vedere l'intervallo determinato da due chiavi

| | sequential search | binary search |
|-------------------|-------------------|---------------|
| search | N | $\log N$ |
| insert / delete | N | N |
| min / max | N | 1 |
| floor / ceiling | N | $\log N$ |
| rank | N | $\log N$ |
| select | N | 1 |
| ordered iteration | $N \log N$ | N |

▼ Binary Search Tree

Un **BST** è un albero binario che rispetta la condizione di **ordine simmetrico**.

Cioè **ogni nodo** ha una chiave che è maggiore di tutte le chiavi del **sotto-albero sx** e contemporaneamente **minore** di tutte le chiavi del **sotto-albero dx**.

L'albero binario è definito ricorsivamente come:

- un albero vuoto
- un nodo collegato a due alberi binari disgiunti \Rightarrow sottoalbero sinistro e destro

Algoritmo di ricerca di un elemento in un albero binario

Si parte dalla radice, si controlla il nodo corrente e:

- se l'elemento che cerco è $<$ del nodo corrente allora passo al **sottoalbero sinistro**
- se l'elemento che cerco è $>$ del nodo corrente allora passo al **sottoalbero destro**
- se l'elemento è **uguale** \Rightarrow **search hit**

BST in Java

In **Java**, rappresentiamo un **BST** utilizzando un semplice riferimento ad un oggetto di tipo **Node** che costituirà la radice dell'albero.

Esso sarà composto da 4 campi:

- chiave
- valore
- riferimento ad un Nodo left (sottoalbero sx)
- riferimento ad un Nodo right (sottoalbero dx)

Metodo Get

Per questo metodo bisogna navigare l'albero fin quando non si trova il valore richiesto.

Quindi nel **worst case** questo metodo attraverserà l'albero in tutta la sua profondità, per cui la complessità sarà **logaritmica**.

Metodo Put

Ha un'implementazione più complessa, infatti questo metodo ha al suo interno un altro metodo privato che effettua ricorsivamente la ricerca della chiave (se già presente) e:

- se la chiave è già **presente** nell'albero, **resetta** il valore
- se la chiave **NON** è presente, **aggiunge** un nuovo nodo

Anche in questo caso il numero di confronti è pari a $1 + \text{livello profondità del nodo}$.

Forma dell'albero

Molti alberi binari di ricerca possono essere rappresentati usando forme diverse dell'albero a seconda dell'ordine di inserimento degli elementi.

Distinguiamo 3 casi particolari:

- **Best Case:** albero perfettamente bilanciato, quindi l'altezza dell'albero è $\lg N$, il sottoalbero destro e sinistro hanno a loro volta la stessa altezza e per ogni foglia ho lo stesso numero di archi necessari per risalire fino alla **radice**
- **Caso tipico:** fornendo le chiavi randomicamente l'albero potrebbe diventare non bilanciato, ma non così sbilanciato come nel caso peggiore

- **Worst Case:** l'albero è completamente sbilanciato, l'altezza coincide con il numero di elementi $N \Rightarrow$ ogni nodo ha un solo ramo

Altre operazioni

Così come lo abbiamo implementato è possibile fare:

- **ricerca chiave maggiore**
- **ricerca chiave minore**
- **floor:** operazione che trova **la chiave maggiore** tra tutte quelle minori o uguali alla chiave data
- **ceiling:** operazione che trova **la chiave minore** tra tutte quelle maggiori o uguali alla chiave data
- **rank:** restituisce il numero di chiavi minori o uguali a una data
- **select:** restituisce la **chiave** che si trova in quella posizione dell'albero ordinato
- **size:** memorizziamo il numero di nodi presenti nei sottoalberi di cui quel nodo è radice, così da implementare il metodo di restituzione in modo semplice, facendo restituire il valore della variabile d'istanza

Attraversamento in order

L'attraversamento in order è un algoritmo che permette di attraversare l'albero con il seguente ordine:

- **attraversare prima il sottoalbero sinistro**
- inserire in coda la chiave
- **attraversare il sottoalbero destro**

Questo attraversamento ha la seguente **proprietà**: restituisce le chiavi in **ordine crescente**

Symbol Table con BST

L'implementazione tramite **BST riduce il costo di tutte le operazioni**, infatti tutte dipendono dall'altezza dell'albero.

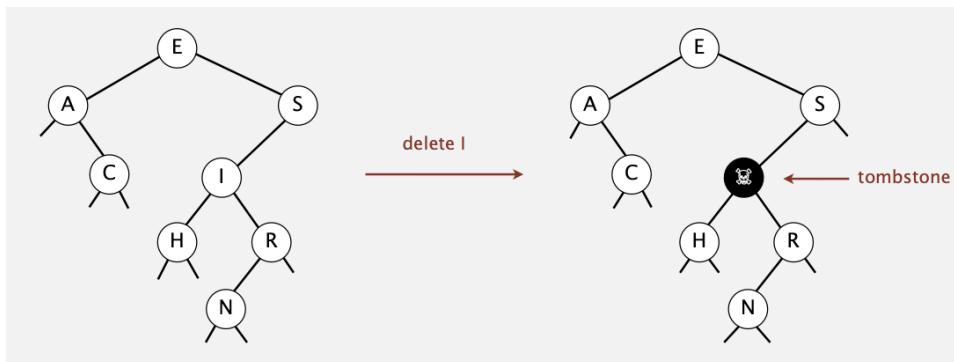
Se assumiamo che le chiavi vengano inserite in modo randomico, allora è proporzionale a $\lg N$

Operazione di deletion (Approccio lazy, pigro)

Data una chiave, per rimuovere il rispettivo nodo è necessario:

- settare il valore a null
- lasciare la chiave nell'albero per guidare la ricerca, ma non viene considerata nell'operazione di ricerca

L'approccio è lazy perché il nodo non viene effettivamente eliminato ma settato a null

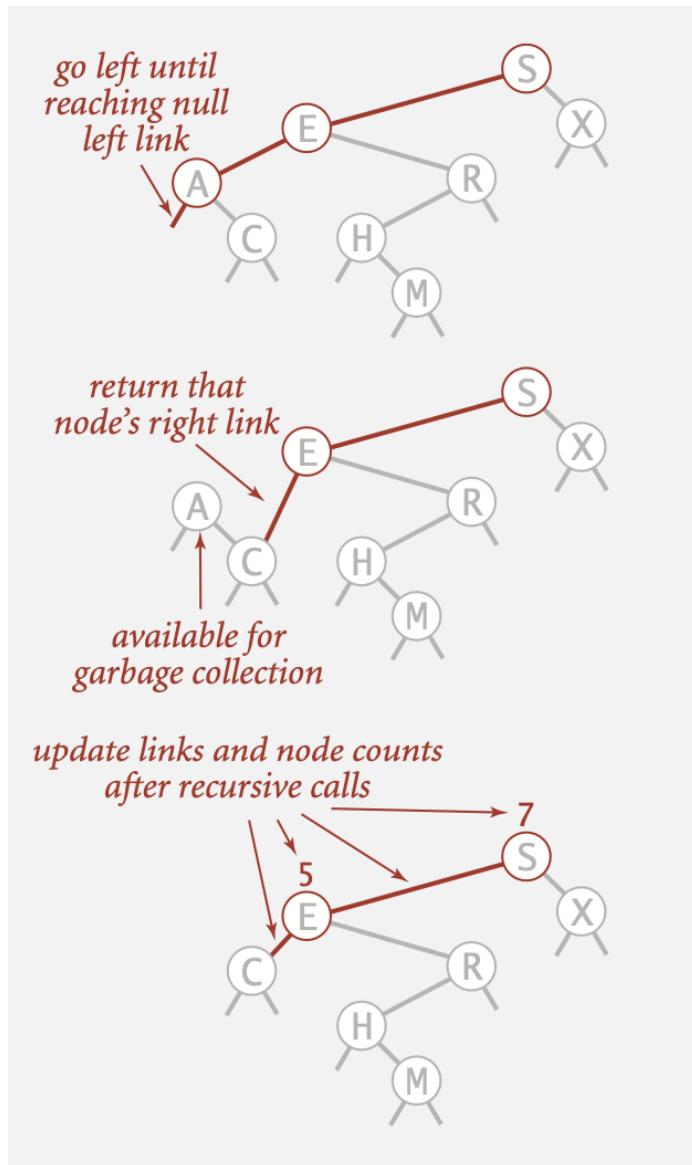


Eliminazione del minimo

La chiave minima del **BST** si trova nel nodo foglia più a sinistra.

Quindi per trovare il minimo basta:

- spostarsi nel sottoalbero sinistro fino a trovare un nodo il cui riferimento al sottoalbero sinistro è **null**.
- sostituire quel nodo con il suo riferimento al sottoalbero destro (il nodo padre sarà ora collegato alla sua sinistra con il ramo destro del nodo eliminato).
- aggiornare i conteggi dei sottoalberi



HIBBARD DELETION

Quest'operazione consiste nell'**eliminare uno specifico nodo** con una specifica chiave k

Distinguiamo vari casi:

- **Caso 0:** il nodo ha **0 figli**

Si elimina il nodo settando il collegamento del nodo padre a null

- **Caso 1:** il nodo ha **1 figlio**

Si elimina il nodo sostituendo il collegamento del nodo padre collegando ad esso il nodo figlio del nodo eliminato

- **Caso 2:** il nodo da **2 figli**

Una volta trovato il nodo da eliminare, bisogna cercare un altro nodo tale da

essere il minimo del sottoalbero destro del nodo da eliminare.

Questo nodo trovato sarà il successore del nodo eliminato

Per trovare il nodo minimo, si va nel sottoalbero destro e poi si va sempre a sinistra finché non si raggiunge un nodo che ha un sottoalbero sinistro posto a null

Infine si mette il nodo minimo al posto del nodo da eliminare

Questa soluzione però non è soddisfacente perché non è **simmetrica**.

Gli alberi non sono randomici, per cui la delete ha prestazioni \sqrt{N}

Non esiste ancora un'implementazione semplice ed efficiente per l'operazione di delete di un nodo in un BST

| implementation | guaranteed | | | average case | | | ordered ops? | operations on keys |
|------------------------------------|------------|--------|--------|----------------|----------------|----------------|--------------|--------------------------|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (linked list) | N | N | N | $\frac{1}{2}N$ | N | $\frac{1}{2}N$ | | <code>equals()</code> |
| binary search (ordered array) | $\lg N$ | N | N | $\lg N$ | $\frac{1}{2}N$ | $\frac{1}{2}N$ | ✓ | <code>compareTo()</code> |
| BST | N | N | N | $1.39 \lg N$ | $1.39 \lg N$ | \sqrt{N} | ✓ | <code>compareTo()</code> |

other operations also become \sqrt{N}
if deletions allowed

▼ Balanced Search Tree

L'obiettivo è quello di garantire prestazioni **logaritmiche** per le operazioni di **search-insert-delete** in un albero binario di ricerca sia nel worst case che nell'average case.

Albero 2-3 (2-3 search tree)

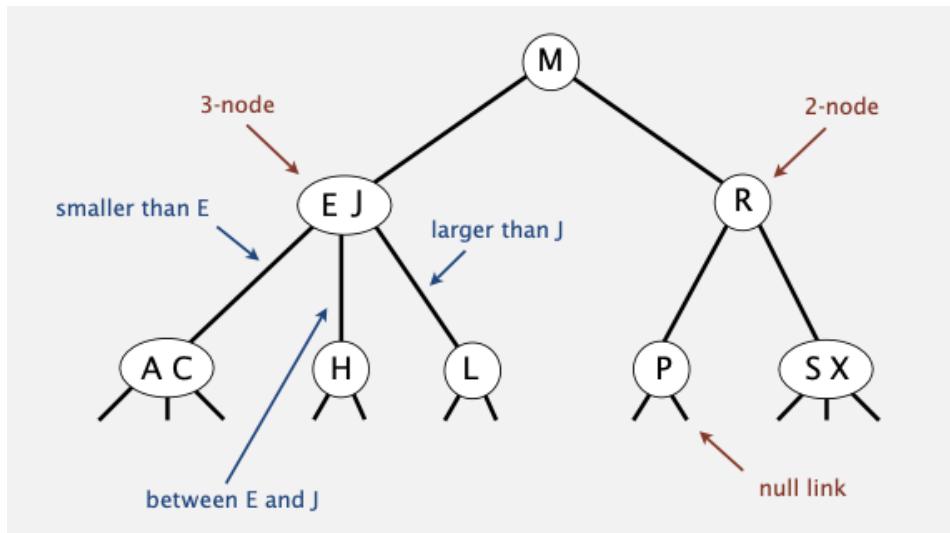
Un albero 2-3 è un albero che ammette **1 o 2 chiavi per nodo**.

Distinguiamo quindi i nodi in:

- **2-node** che ha 1 chiave e 2 figli
- **3-node** che ha 2 chiavi e 3 figli

Nel 3-node, il figlio centrale rappresenta i nodi che hanno chiavi comprese tra le due chiavi del nodo padre.

Esempio: se il 3-node ha come chiavi E e J, il ramo figlio centrale avrà i nodi con chiavi comprese tra E e J.



L'ordine rispettato è sempre l'ordine simmetrico che permette di avere le chiavi in ordine crescente mediante un attraversamento **inorder**.

Con un albero 2-3 si ottiene un albero **perfettamente bilanciato**, infatti ogni percorso dalla **root** ad un nodo **foglia** ha sempre la stessa lunghezza.

Operazioni

Search

È uguale a quella dell'albero binario con l'unica differenza che si tiene conto anche del terzo link, ciò significa che nel **worst case** vengono fatti 3 confronti per ogni nodo.

Insert

Distinguiamo gli inserimenti in base al tipo di nodo foglia:

- **Inserimento in un 2-node:** si aggiunge la nuova chiave al 2-node e si crea in questo modo un 3-node

https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/8fe0f412-25fa-4482-aaab-9bc622831625/Inserimento_2-3_2-node.mov

- **Inserimento in un 3-node:** si aggiunge la nuova chiave al 3-node creando temporaneamente un 4-node, poi si sposta la chiave centrale del 4-node nel nodo genitore e se si raggiunge la radice e questa è un 4-node, allora viene divisa in 3 2-node

https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/23244189-30b4-4e38-b178-34d8a11b8975/Inserimento_2-3_3-node.mov

Lo split di un 4-node è una **trasformazione locale**, infatti richiede un numero costante di operazioni, ogni trasformazione mantiene l'ordine simmetrico e il bilanciamento perfetto dell'albero.

Prestazioni

Il numero di figli per nodo può variare da 2 a 3, di conseguenza il numero di foglie andrà da 2^h a 3^h , per cui l'altezza dell'albero andrà da $\log_2 N$ a $\log_3 N$

Il **worst case** si ha quando i nodi sono tutti **2-node**

Invece, il **best case** si ha quando i nodi sono tutti **3-node**

In entrambi i casi le prestazioni sono **logaritmiche**, il problema è che questo tipo di albero (2-3) è difficile da implementare

Albero LLRB (Left Leaning Red-Black BST)

È un modo per rappresentare un 3-node tramite un BST.

In particolare, ogni volta che inserisco un nodo, il link che collega la radice al nuovo nodo viene marcata come **rosso**.

Distinguiamo 3 casi:

- se il link rosso è a destra allora viene fatto uno **swap** tra padre e figlio (left rotation)
- se ci sono due rami rossi sulla sinistra, il nodo centrale diventa il padre degli altri due nodi (right rotation)

- se ci sono rami rossi sia a destra che a sinistra, questi vengono marcati come neri e il ramo che collega il nodo al padre viene marcato come rosso (se è un ramo destro vedere primo caso)

https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/baa4bd01-597e-4f4b-86ad-4e2ccde3f960/33De_moRedBlackBST.mov

Un albero 2-3 può essere trasformato in un **BST** in cui si usano dei link rossi che fanno da **collante** tra i nodi che rappresentano le 2 chiavi di un 3-node.

Si usa un left link che collega alla sinistra della chiave maggiore, la chiave minore del 3-node

Possiamo trarre una definizione di albero binario di ricerca equivalente:

- Nessun nodo può avere 2 rami rossi connessi ad esso
- Ogni percorso dalla radice ad un nodo **foglia** ha lo stesso numero di rami neri (perfetto bilanciamento dei rami neri)
- I rami rossi sono sempre dei rami verso sinistra (dalla chiave maggiore a quella minore del 3-node)

Implementazione red-black del BST

Ogni nodo è puntato da un solo link proveniente dal nodo genitore, per rappresentare il colore del link viene aggiunto un attributo booleano alla classe Node.

Search

La ricerca è identica a quella effettuata nei BST elementari, infatti viene ignorato il colore dei link, è più veloce grazie al bilanciamento dell'albero

Insert

Ricordiamo che per ogni operazione interna bisogna mantenere l'ordine simmetrico e il perfetto bilanciamento dei black link.

Durante le operazioni di insert bisogna correggere eventuali situazioni intermedie che non rispettano le proprietà di ordine simmetrico e bilanciamento perfetto:

- red link verso destra
- 4-node temporaneo:
 - 2 nodi figli collegati al genitore da un ramo rosso
 - doppio ramo rosso consecutivo a sinistra (nodo padre - nodo - nodo figlio)
 - doppio ramo rosso consecutivo sinistra-destra

Le operazioni per aggiustare la simmetria e il bilanciamento sono:

- **left rotation**, si applica quando c'è un ramo rosso a destra
- **right rotation**, si applica quando ci sono due rami rossi consecutivi a sinistra
- **color flip**, si applica quando c'è un ramo rosso sia a destra che a sinistra di un nodo

Bilanciamento in un albero LLRB

L'altezza dell'albero LLRB è $\leq 2 \lg N$ nel **worst case**

Dimostrazione:

- ogni percorso che va dalla root ad un link null ha sempre lo stesso numero di rami neri
- non possono mai esserci 2 rami rossi consecutivi

Di solito l'altezza dell'albero è $\sim 1.0 \lg N$

Prestazioni

Dato che l'albero è sempre bilanciato, con questa implementazione possiamo dire che nel **worst case** l'altezza dell'albero sarà $2 \lg N$, mentre nell'**average case** tende a $\lg N$

Con questa implementazione riusciamo a migliorare le prestazioni di tutte le operazioni del BST, in particolare la hibbard deletion passa da avere complessità \sqrt{N} a $\lg N$

| implementation | guarantee | | | average case | | | ordered ops? | key interface |
|---|-----------|-----------|-----------|----------------|----------------|----------------|--------------|--------------------------|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (unordered list) | N | N | N | $\frac{1}{2}N$ | N | $\frac{1}{2}N$ | | <code>equals()</code> |
| binary search (ordered array) | $\lg N$ | N | N | $\lg N$ | $\frac{1}{2}N$ | $\frac{1}{2}N$ | ✓ | <code>compareTo()</code> |
| BST | N | N | N | $1.39 \lg N$ | $1.39 \lg N$ | \sqrt{N} | ✓ | <code>compareTo()</code> |
| 2-3 tree | $c \lg N$ | $c \lg N$ | $c \lg N$ | $c \lg N$ | $c \lg N$ | $c \lg N$ | ✓ | <code>compareTo()</code> |
| red-black BST | $2 \lg N$ | $2 \lg N$ | $2 \lg N$ | $1.0 \lg N^*$ | $1.0 \lg N^*$ | $1.0 \lg N^*$ | ✓ | <code>compareTo()</code> |

B-Tree

Un B-tree è una struttura dati che appartiene alla famiglia dei **Balanced Search Tree**

I B-Tree sono molto usati in contesti di sistemi operativi ed in particolare nella gestione della **memoria virtuale**, in quanto molto utili per la ricerca delle pagine

Un B-albero è una generalizzazione degli alberi 2-3 che consente di avere fino a M-1 coppie di chiave-link per ogni nodo, ossia:

- nella radice ci sono almeno 2 coppie chiave-link
- negli altri nodi almeno $M/2$ coppie chiave-link (al più $M-1$)

Il valore di M va scelto il più grande possibile in modo che M link possano entrare in una pagina.

Distinguiamo due tipi di nodi:

- **nodi esterni** → contengono le effettive coppie chiave-valore da memorizzare
- **nodi interni** → contengono le copie delle chiavi che indicano dove trovare i dati cercati, fungono da guida per la ricerca

In particolare, nei nodi interni è presente una particolare chiave chiamata **sentinella** che modella tutti i valori delle chiavi minori

Infatti, per **cercare** tutte le chiavi minori di una data chiave k basterà **muoversi** nel ramo indicato dalla chiave sentinella.

Vediamo le operazioni

Search

Si parte dalla root:

- si cerca l'intervallo che consente di trovare la chiave e si esegue il link corrispondente
- la ricerca termina quando si raggiunge un nodo esterno

Insert

Come negli alberi 2-3, l'inserimento dovrà mantenere il bilanciamento dell'albero, per cui:

- si cerca la nuova chiave
- si inserisce la nuova chiave nel livello finale
- se c'è un overflow delle chiavi del nodo si effettua uno split dal basso verso l'alto

Se accade uno split del nodo radice verrà creata una nuova root.

Prestazioni

Data una collezione di N elementi da disporre in un B-Tree di livello M , sappiamo che il numero di figli che può avere un nodo è $M/2$ fino ad un massimo di $M - 1$.

Ciò vuol dire che l'altezza va da $\log_{M/2} N$ a $\log_{M-1} N$

▼ Hash Tables

Possiamo migliorare l'implementazione dei red-black BST sfruttando un altro sistema di accesso ai dati, cioè l'**hashing**.

L'idea è quella di salvare gli elementi in una **tabella indicizzata dalla chiave**, in cui l'indice è calcolato attraverso una funzione della chiave chiamata funzione di **hash**.

Una funzione di hash è un metodo che consente di **calcolare l'indice (posizione) nell'array a partire dalla chiave**

Questo meccanismo può portare ad alcuni problemi:

- il calcolo della funzione di hash deve essere **più efficiente** possibile perché è necessario per ogni operazione
- il test di uguaglianza tra due chiavi deve essere implementato **in accordo** con la funzione di hash, cioè due chiavi uguali devono avere lo stesso hashcode
- quando chiavi diverse restituiscono lo stesso indice si ha il problema delle **collisioni**

Come realizzare una funzione di hash

L'ideale sarebbe realizzare una funzione **il più efficiente** possibile che codifichi le chiavi uniformemente, ossia restituisca in output valori equiprobabili, cioè ben distribuiti tra gli indici dell'array da riempire.

Tutte le classi Java ereditano il metodo hashCode() dalla classe Object, la cui implementazione di default restituisce l'indirizzo di memoria di un oggetto x sotto forma di valore intero a 32 bit

Molte altre classi Java di libreria (classi wrapper dei tipi primitivi, String, File, Date, etc...) presentano implementazioni personalizzate del metodo hashCode()

Per i tipi definiti dall'utente, si combinano i valori delle variabili d'istanza usando la regola $31x + y$, cioè si moltiplica il valore dell'hashCode della variabile corrente e poi sommo l'hash della successiva.

Distinguiamo 4 casi:

- Se il campo è di tipo primitivo → uso il metodo hashCode della rispettiva classe Wrapper
- Se il campo è di tipo reference → uso il metodo hashCode della classe
- Se il campo è null → return 0
- Se il campo è un array, applico la funzione ad ogni entry dell'array, in alternativa utilizzo il metodo Arrays.deepHashCode()

Questa regola è molto utilizzata nelle classi di libreria Java, ma dato che le chiavi sono stringhe di bit è possibile pensare ad una funzione di hash universale.

Hashing Modulare

L'hashCode è un intero compreso tra -2^{31} e $+2^{31} - 1$, ma la funzione di hash ha come scopo quello di mappare i valori di hash con gli indici di un array compreso tra 0 e M-1, dove M è tipicamente un numero primo o una potenza di 2.

L'implementazione più corretta del modular hashing è la seguente:

```
private int hash(Key key){  
    return (key.hashCode() & 0x7fffffff) % M;  
}  
  
// si effettua & bit a bit con 0x7fffffff per ottenere un numero  
// si effettua % con M per ottenere un numero compreso tra 0 e M-1
```

Assunzione di hashing uniforme: ogni chiave deve restituire un valore intero **equiprobabile** compreso tra 0 e $M-1$ in modo che ogni posizione dell'array abbia la stessa probabilità di essere riempita da una chiave

Gestione delle collisioni

Una **collisione** avviene se due chiavi **distinte** producono uno stesso indice come risultato della funzione di hash, quindi andrebbero ad occupare la stessa posizione nell'hash table.

Invece di sovrascrivere i valori si utilizza la tecnica del **separate chaining**

Separate Chaining

Questa tecnica consiste nell'implementare l'hash table con un array di M linked list.

Ogni linked list sarà lunga $\frac{N}{M}$ (N =numero di elementi, M =lunghezza dell'array)

In pratica, ogni posizione dell'array conterrà una reference ad una linked list di chiavi che condividono lo stesso indice di hash (collisione)

In questo modo, in caso di collisione, il nuovo oggetto sarà messo in coda all'oggetto che occupa la stessa posizione.

La ricerca sarà più efficiente perché si effettua la ricerca della chiave nella linked list contenuta nella posizione dell'indice restituito dalla funzione di hash, per cui non c'è bisogno di ricercare nell'intera tabella.

È importante la scelta di M , infatti:

- M troppo grande fa sì che ci siano linked list vuote \Rightarrow spreco di memoria

- M troppo piccola fa sì che ci siano chain troppo lunghe, si perde il vantaggio di fare pochi confronti tendendo così alla ricerca sequenziale

La scelta tipica è $\sim \frac{N}{4}$ che garantisce operazioni in un tempo costante.

Resizing dell'hash table

Per avere una lunghezza delle liste costante pari a $\frac{N}{M}$ è necessario:

- Raddoppiare la dimensione M dell'array quando il rapporto è ≥ 8
- Dimezzare M quando il rapporto è ≤ 2
- Rieffettuare l'hash di tutte le chiavi ogni volta che avviene un resize (a cambiare non saranno gli hashCode ma soltanto gli indici della funzione hash, perché dipende da M)

Tali operazioni sono onerose ma così facendo si ottengono catene meno lunghe che ottimizzeranno le operazioni di search e insert.

Operazione di delete

Per eliminare una chiave (e il valore associato) basterà considerare soltanto la catena che contiene la chiave da eliminare ed effettuare una rimozione del nodo che la contiene (collegare il nodo precedente a quello successivo)

Sommario delle implementazioni di una symbol table

| implementation | guarantee | | | average case | | | ordered ops? | key interface |
|---|-----------|-----------|-----------|-----------------|-----------------|-----------------|--------------|--|
| | search | insert | delete | search hit | insert | delete | | |
| sequential search (unordered list) | N | N | N | $\frac{1}{2} N$ | N | $\frac{1}{2} N$ | | <code>equals()</code> |
| binary search (ordered array) | $\lg N$ | N | N | $\lg N$ | $\frac{1}{2} N$ | $\frac{1}{2} N$ | ✓ | <code>compareTo()</code> |
| BST | N | N | N | $1.39 \lg N$ | $1.39 \lg N$ | \sqrt{N} | ✓ | <code>compareTo()</code> |
| red-black BST | $2 \lg N$ | $2 \lg N$ | $2 \lg N$ | $1.0 \lg N$ | $1.0 \lg N$ | $1.0 \lg N$ | ✓ | <code>compareTo()</code> |
| separate chaining | N | N | N | $3-5^*$ | $3-5^*$ | $3-5^*$ | | <code>equals()</code> <code>hashCode()</code> |

* under uniform hashing assumption

L'implementazione di **hash table** con separate-chaining garantisce prestazioni lineari nel caso peggiore, ma considerando l'assunzione di hashing uniforme (tutte le posizioni sono equiprobabili) abbiamo che nell'average case tutte le operazioni avranno un costo costante, non dipendente dal numero di elementi all'interno della struttura.

Linear probing hash table - indirizzamento aperto

Questo metodo non prevede di mappare due elementi con stesso codice hash nella stessa posizione ma di inserire il **nuovo elemento** nel primo **slot successivo vuoto**.

Questa tecnica è realizzabile solo se la dimensione M dell'array è **maggior**e rispetto al numero N di coppie chiave-valore.

Per **cercare** l'elemento, se esso non sarà nella posizione corretta, si verificano le posizioni successive fino a trovare l'elemento (**search hit**) o fino al trovare una posizione non occupata (**search miss**).

Operazioni:

- **hash**: mappare le chiavi con interi i compresi tra 0 e M-1
- **insert**: inserire la chiave all'indice **i** se libero altrimenti provare per **i+1, i+2, ...**
- **search**: cercare la chiave all'indice **i**, se la posizione è occupata cercare in **i+1, i+2, ...**

Clustering

Un **cluster** è un blocco contiguo di elementi, le nuove chiavi da inserire hanno una grande probabilità di venir mappate nel mezzo dei grandi cluster.

Problema del parcheggio di Knuth

Consideriamo un'auto che arriva in una strada con M spazi di parcheggio.

Ognuno desidera un posto random i tale che se il posto è occupato allora si prova il posto **i+1, i+2, ...**

Il **displacement** è lo spazio che bisognerà percorrere per trovare un posto libero (il numero di posti da scorrere).

Quando il parcheggio è pieno a metà, cioè $\frac{M}{2}$ avremo un **displacement** pari a $\sim \frac{3}{2}$

Se il parcheggio è completamente pieno avremo un **displacement** pari a $\sim \sqrt{\pi \frac{M}{8}}$

Se consideriamo l'assunzione di **hashing uniforme**, il numero medio di probe in una hash table di tipo linear probing di dimensione M che contiene $M = \alpha M$ chiavi è:

$$\sim \frac{1}{2} \left(1 + \frac{1}{1 - \alpha}\right) \text{ per un search hit}$$

$$\sim \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2}\right) \text{ per un search miss / insert}$$

Ciò significa che:

- se M è troppo grande \rightarrow ci saranno troppe entry vuote
- se M è troppo piccolo \rightarrow ci saranno molti blocchi contigui e quindi il tempo per la ricerca aumenta considerevolmente
- La scelta tipica nel caso di linear probing è $\alpha = \frac{N}{2} \sim \frac{1}{2}$

Così facendo si ha un numero di probe per la ricerca:

- **search hit** $\sim \frac{3}{2}$
- **search miss** $\sim \frac{5}{2}$

Resizing

Per mantenere il valore del rapporto $\alpha \leq \frac{1}{2}$ occorre:

- raddoppiare la dimensione M quando il rapporto $\frac{N}{M} \geq \frac{1}{2}$
- dimezzare la dimensione M quando il rapporto $\frac{N}{M} \leq \frac{1}{8}$
- effettuare un rehash delle chiavi quando c'è un resizing

Operazione di Delete

Per effettuare la delete non basta eliminare l'entry dell'array perché se si effettuasse un'operazione di search si rischierebbe di stoppare in quella posizione diventata vuota la ricerca di una chiave.

Per risolvere questo problema si può utilizzare un **place holder** che indica che la posizione era occupata precedentemente, oppure rieffettuare l'hashing delle chiavi.

Conclusioni Symbol Table

Il linear probing e il separate chaining non garantiscono prestazioni **sublineari** nel caso peggiore.

Nel caso medio, invece, con un'assunzione di hashing uniforme le prestazioni delle operazioni sono costanti perché non dipendono dal numero di elementi presenti nella struttura.

- **Separate Chaining**

- Le prestazioni degradano gradualmente
- Il clustering è meno sensibile a funzioni hash mal progettate

- **Linear Probing**

- Meno spazio sprecato nell'array → più entry occupate
- Migliori prestazioni della cache

Varianti dell'hashing

- **Two probe hashing:** variante del separate chaining:

- L'hash produce due posizioni e si inserisce la chiave nella catena più piccola
- Riduce la lunghezza attesa della catena più lunga a $\log \log N$

- **Double hashing:** variante del linear probing

- Si usa il linear probing ma si salta un numero di posizioni variabile e non 1 per volta
- Efficace nell'eliminare il clustering
- Permette di avere una tabella quasi piena
- Rende difficile l'implementazione di delete

- **Cuckoo hashing:** variante del linear probing

- L'hash produce due posizioni e si inserisce la chiave in una delle due, se è occupata si reinserisce la chiave nella posizione alternativa e così via

ricorsivamente

- Tempo di ricerca costante nel **worst case**

▼ Confronto tra Hash Table e BST

Le **hash tables** sono:

- semplici da programmare
- più veloci con chiavi semplici (richiedono poche operazioni aritmetiche vs un numero di confronti logaritmico degli alberi)
- miglior supporto per le stringhe in Java

I **BST**:

- migliori prestazioni garantite (nel **worst case**)
- supporto per le operazioni ordinate della Symbol Table
- più semplice implementare correttamente il metodo `compareTo()` rispetto ai metodi `equals()` e `hashCode()`

BST red-black → `java.util.TreeMap` e `TreeSet`

Hash Table → `HashMap` e `IdentityHashMap`

▼ Grafi non orientati

Un **grafo** è un insieme di **nodi** connessi da un insieme di **archi**.

Un **path** è una sequenza di **nodi** connessa da archi.

Un **ciclo** è un cammino che inizia e termina nello stesso nodo (primo e ultimo nodo coincidono).

Il **grado** di un nodo è pari al numero di nodi a cui è connesso

Due **nodi** si dicono connessi se esiste almeno un cammino che li unisce.

Una **componente连通** di un grafo è costituita da un **insieme massimale** di nodi raggiungibili da uno qualsiasi degli altri nodi appartenenti alla stessa componente.

Definiamo:

- **self loop** → se un vertice è collegato con sè stesso.
- **archi paralleli** → una coppia di nodi è collegata da più archi.

Un grafo può essere rappresentato tramite:

- **un insieme di archi**, cioè come **lista degli archi (array o linked list)** → implementazione non efficiente
- **matrice di adiacenza N x N** → composta da elementi booleani dove per ogni arco $v-w$ nel grafo: `ad[v][w] = ad[w][v] = true`

La **mda** è più efficiente per trovare i nodi adiacenti ad un dato nodo ma utilizza molto spazio in memoria perché vengono create delle matrici **fortemente sparse** (molte entry = 0), mentre per ogni arco si hanno due entry poste a **true**.

I tempi di esecuzione per trovare i nodi adiacenti sono proporzionali ad N .

Un'altra implementazione è quella delle **liste di adiacenza**: ad esempio usiamo un array di liste, ognuna di esse contiene i nodi di adiacenti ad un dato nodo.

Esempio: nella posizione 0 abbiamo la lista dei nodi adiacenti al nodo 0.

Così facendo il tempo di esecuzione per trovare i nodi adiacenti sarà proporzionale al grado del vertice.

| representation | space | add edge | edge between v and w ? | iterate over vertices adjacent to v ? |
|------------------|---------|----------|-------------------------------|--|
| list of edges | E | 1 | E | E |
| adjacency matrix | V^2 | 1 * | 1 | V |
| adjacency lists | $E + V$ | 1 | $degree(v)$ | $degree(v)$ |

* disallows parallel edges

Algoritmo di ricerca in profondità (depth-first search)

Immaginiamo un labirinto come un grafo dove **ogni nodo** è un'intersezione e **ogni arco** è un passaggio, l'obiettivo è esplorare tutti i nodi del labirinto

L'algoritmo di **Tremaux** si basa sui seguenti passi:

- srotolare un gomitolo durante il percorso e segnare ogni intersezione e passaggio visitati
- quando si sono visitati tutti i percorsi a partire da un nodo, bisogna effettuare una retrace (torna indietro) del cammino

L'algoritmo **depth-first-search** ha come obiettivo l'attraversamento di un grafo basato proprio sull'esplorazione del labirinto, sfruttando lo stack delle chiamate di funzione in maniera analoga all'utilizzo del gomitolo.

Tipiche applicazioni:

- trovare tutti i nodi connessi ad un dato nodo sorgente
- trovare un cammino tra due nodi

Consideriamo un **grafo non orientato**, partiamo dal nodo 0 e visitiamo ricorsivamente tutti i nodi non segnati adiacenti al nodo attraversato.

Utilizziamo un **array di booleani** per segnare se un nodo è stato attraversato e un **array di interi** che contiene gli archi traversati per arrivare ad un dato nodo.

Ogni volta che sono in un nodo mai esplorato, continuo, invece quando si arriva in un nodo già visitato torno indietro (al chiamante).

Design pattern per il processing di un grafo

Disaccoppiare il tipo di dati del grafo dal suo processing:

- creare un oggetto **Graph**
- passare l'oggetto ad una routine di **graph-processing**
- interrogare la routine per ottenere informazioni

Per visitare un nodo v:

- segnare il nodo v come visitato
- visitare ricorsivamente tutti i nodi adiacenti a esso che non sono segnati

Strutture dati utilizzate:

- array di booleani `marked[]` in cui segnare i nodi visitati
- array di interi `edgeTo[]` che tiene traccia dei cammini presi (`edgeTo[w] == v` significa che l'arco `v-w` è stato utilizzato per visitare `w` per la prima volta)
- Usare lo stack delle chiamate di funzione per la ricorsione per fare il retrace

<https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/1288a41c-650a-44e2-924e-595678d14105/41DemoDept hFirstSearch.mov>

Proposizione: la DFS marchia tutti i nodi connessi al nodo sorgente s in un tempo proporzionale alla somma dei loro gradi (+ il tempo per inizializzare l'array marked[])

In particolare **ogni nodo connesso al nodo s sarà visitato una e una sola volta.**

Dopo aver effettuato la DFS, è possibile controllare se il nodo v è connesso ad s in un tempo costante e trovare un cammino v-s (se esiste) in un tempo proporzionale alla sua lunghezza.

Algoritmo di ricerca in ampiezza (breadth-first search)

Utilizziamo una **queue** per segnare i nodi non esplorati che sono raggiungibili da un dato nodo.

Finché la coda non è vuota:

- rimuovere il nodo v dalla coda
- aggiungere alla coda tutti i nodi **non segnati** adiacenti a v e segnarli

Utilizziamo 3 strutture dati:

- array di booleani `marked[]` per segnare i nodi visitati
- array di interi `edgeTo[]` per segnare il nodo da cui si arriva
- array di interi `distTo[]` per segnare le distanze tra i nodi
- coda di supporto nel metodo BFS

<https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/67347b7c-6a3c-4b5d-a2d4-2fbb879e1f75/41DemoBread thFirstSearch.mov>

La BFS esamina i nodi grazie ad un ordinamento basato sulla distanza crescente dalla sorgente s.

La coda contiene sempre **un numero ≥ 0** di nodi di **distanza k** da s, seguiti da un **numero ≥ 0** di nodi di **distanza k+1**.

In ogni grafo connesso G, la **BFS** calcola i cammini minimi da s fino a tutti gli altri nodi in un tempo proporzionale a E+V (numero di archi + numero di vertici)

Con **cammino minimo** si intende come **numero di archi percorsi**, perché gli archi **NON** sono pesati.

Componenti Connesse

Query di connettività

Sappiamo che la **connessione è una relazione di equivalenza** (riflessiva, simmetrica e transitiva) e che due vertici v e w sono connessi se esiste un cammino che li unisce.

L'obiettivo è preprocessare il grafo in modo da poter rispondere a una query di connettività del tipo "v è connesso a w?" in un tempo **costante**.

Conosciamo il concetto di **componente connessa**, ossia un insieme massimale di vertici connessi. Possiamo quindi partizionare i nodi del grafo secondo le componenti connesse.

Una volta che abbiamo le componenti connesse del grafo, possiamo rispondere a delle query di connettività in un tempo costante.

Implementazione java

Usiamo l'array di booleani `marked[]` per segnare i nodi attraversati dalla **DFS**, l'array di `id[]` per segnare l'id della componente a cui appartiene un vertice attraversato dalla **DFS** e una variabile `count` che conterrà il numero delle componenti del grafo.

<https://prod-files-secure.s3.us-west-2.amazonaws.com/e1eb52d3-764b-46b3-9d9f-ff895f4b8814/ced32575-c7b0-4c6f-a546-6f41bf7a2314/41DemoConnectedComponents.mov>

Sfide di graph processing

Tipici problemi di graph processing sono:

- Verificare se un grafo è **bipartito**: un grafo si dice bipartito se l'insieme dei suoi nodi può essere partizionato in due sottoinsiemi tali che ogni nodo di una delle due parti è collegato soltanto a nodi dell'altra partizione.
- Ricerca di un ciclo nel grafo
- Problema dei ponti di **Konigsberg**: verificare se esiste un ciclo che attraversa ogni arco esattamente una volta (**ciclo o circuito euleriano**)
Un grafo connesso è **euleriano** se e solo se tutti i nodi sono di grado **pari** (sono connessi ad un numero di nodi pari)
- Verificare se esiste un **ciclo o circuito hamiltoniano**, cioè ogni ciclo visita ogni nodo una sola volta
- **Isomorfismo** dei grafi: verificare se due grafi sono identici
- Disporre un grafo su un piano senza far incrociare gli archi → esiste un algoritmo basato su **DFS** che risolve questo problema in un tempo lineare

Conclusioni: DFS e BFS sono due algoritmi che introducono soluzioni efficienti per molti problemi riguardanti i grafi non orientati, ma non per tutti.

▼ Grafi orientati

Un **digrafo** è un insieme di nodi, connessi da archi **orientati**.

Tipici problemi sui digrafi:

- **s→t path**: verificare se esiste un cammino tra s e t
- **Shortest s→t path**: trovare il cammino minimo tra s e t
- **Ordinamento topologico**: verificare se il digrafo può essere ordinato in modo che tutti i nodi vengano prima di tutti i nodi collegati ai propri archi uscenti.
- **Connettività forte**: verificare l'esistenza di un cammino orientato tra coppie di nodi
- **Chiusura transitiva**: per quali nodi v e w esiste un percorso diretto tra v e w
- **PageRank**: calcolare l'importanza di una pagina web

Rappresentazione con liste di adiacenza

Si può rappresentare un digrafo mantenendo un **array di liste indicizzato** in base ai nodi

(ogni indice rappresenta un nodo del digrafo)

Per ogni indice i abbiamo la lista dei nodi raggiungibili partendo dal nodo i

È la rappresentazione più usata nella **pratica**, questo perché

- molti algoritmi si basano sull'iterazione tra nodi **adiacenti** ad un nodo v
- i digrafi tratti dal mondo reale tendono ad essere molto sparsi (molti nodi ma di basso grado).

Questa rappresentazione permette di avere:

- un'operazione di **inserimento** di un arco in un tempo costante
- limitare lo spazio occupato (rispetto alla matrice di adiacenza)

Ricerca in un digrafo

Problema di raggiungibilità: trovare tutti i nodi raggiungibili da un nodo s seguendo un cammino di archi orientati

Si utilizza la **DFS** in maniera analoga ai grafi non orientati.

Questo è possibile perché i grafi indiretti sono un caso particolare di grafo orientato: ogni arco può essere scomposto in una coppia di archi in entrambe le direzioni ⇒ **DFS è in realtà un algoritmo per digrafi.**

Applicazione del problema di raggiungibilità: analisi del controllo del flusso di un programma

Ogni programma può essere rappresentato come un digrafo dove:

- nodi → blocchi di istruzione sequenziali
- archi → salti

Gli obiettivi sono:

- eliminazione del **dead-code**: trovare e rimuovere codice irraggiungibile che non verrà mai eseguito
- rivelazione dei **cicli infiniti**: determinare se il nodo di uscita del digrafo è irraggiungibile

Conclusioni: la DFS permette di risolvere problemi semplici sui digrafi (raggiungibilità, ricerca di cammino, ordinamento topologico, rilevamento dei cicli diretti).

Per risolvere problemi più difficili è necessario tener conto delle seguenti basi:

- Soddisfacibilità booleana (**2 satisfiability**)
- Cammini euleriani diretti
- Componenti fortemente connesse

Breadth first search nei digrafi

Come per la DFS, anche la BFS può essere realizzata in maniera analoga per i digrafi
⇒ **anche la BFS è in realtà un algoritmo per i digrafi**

La BFS calcola i **cammini minimi** (come numero di archi percorsi) da s a tutti gli altri nodi del digrafo in un tempo proporzionale a $E+V$

La BFS può essere utilizzata per trovare i cammini minimi con **multipli vertici sorgente**, per fare ciò si effettua una enqueue di tutti i vertici sorgente.

Ordinamento topologico

L'ordinamento topologico è un ordinamento dei nodi di un grafo che prevede che ogni nodo preceda sempre tutti i nodi raggiungibili dai suoi archi uscenti.

Un'applicazione è lo **scheduling** delle attività: dato un insieme di task da completare con dei vincoli di precedenza si deve stabilire l'ordine di scheduling delle task, si utilizza un modello a digrafo:

- nodi → task da eseguire
- archi → vincoli di precedenza

Questo tipo di ordinamento può essere eseguito **solo su grafi che NON presentano cicli** → grafo orientato aciclico **DAG (DIRECTED ACYCLIC GRAPH)**

Dato un grafo orientato aciclico, l'ordinamento topologico consiste nel **ridisegnare il grafo** in modo che **tutti gli archi puntino nella stessa direzione**

Si utilizza la **DFS**

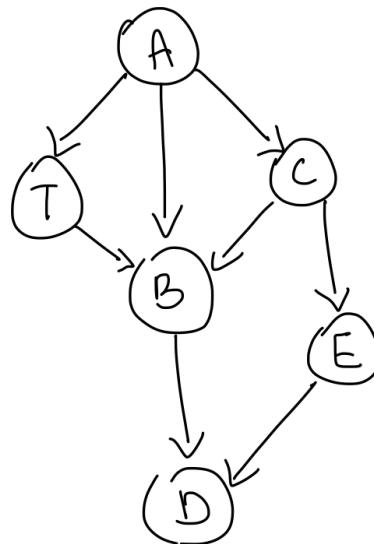
Il depth-first ordering può avere tre tipi di ordinamenti:

- **preorder** → mettere i nodi in una coda prima delle chiamate ricorsive, cioè si marca un nodo come visitato appena viene raggiunto

- **postorder** → mettere i nodi in una coda dopo le chiamate ricorsive, cioè si marca un nodo come visitato solo dopo aver esplorato tutti gli archi uscenti da esso
- **reverse postorder** → mettere i nodi in uno stack dopo le chiamate ricorsive.

Quest'ultimo permette proprio l'ordinamento topologico

Esempio:



PRE ORDER: A, C, B, D, E, T

POST ORDER: D, B, E, C, T, A

REVERSE POST ORDER: A, T, C, E, B, D

Perché l'algoritmo di ordinamento topologico funziona?

- Il **primo nodo** nel postorder ha grado di uscita (numero di vertici ad esso raggiungibili) pari a 0
- I nodi dal secondo all'ultimo possono puntare solo all'ultimo nodo

Proposizione: il reverse postorder della DFS è un ordinamento topologico di un grafo orientato aciclico.

Dimostrazione: consideriamo un qualsiasi arco $v \rightarrow w$. Quando viene chiamato il metodo $\text{dfs}(v)$:

1. $\text{dfs}(w)$ è già stata chiamata e ha terminato l'esecuzione $\Rightarrow w$ è stato attraversato definitivamente prima di v
2. $\text{dfs}(w)$ non è ancora stata chiamata: verrà chiamata direttamente o indirettamente da $\text{dfs}(v)$ e terminerà prima di essa $\Rightarrow w$ verrà completato prima di v
3. $\text{dfs}(w)$ è stata chiamata, ma non è ancora terminata. Questo scenario è impossibile in un grafo orientato aciclico, poiché lo stack delle chiamate a funzione conterrebbe il percorso da w a v , e l'arco $v \rightarrow w$ formerebbe un ciclo.

Rilevazione di cicli diretti

Un digrafo ha un ordinamento topologico **se e solo se NON presenta cicli:**

- se c'è un **ciclo diretto** \rightarrow ordinamento topologico impossibile
- se non ci sono cicli diretti, l'algoritmo basato su DFS può trovare un ordinamento topologico

Per questo motivo possiamo utilizzare la DFS per **rilevare la presenza** di un ciclo diretto nel digrafo.

Componenti fortemente connesse

Due nodi v e w si dicono **fortemente connessi** se esiste un cammino diretto che va da v a w e un cammino diretto che va da w a v .

La connettività forte è una relazione di **equivalenza**:

- Riflessiva
- Simmetrica
- Transitiva

Una **componente fortemente connessa** è un sottoinsieme massimale di nodi fortemente connessi in un digrafo.

Anche le componenti fortemente connesse possono essere **calcolate** grazie all'algoritmo di DFS, segnando in un array l'id della componente a cui ogni vertice appartiene.

In questo modo è possibile verificare in tempo costante la connettività forte tra due nodi di un digrafo.

Algoritmi sulle componenti fortemente connesse

Algoritmo di Kosaraju-Sharir

Questo algoritmo riesce a trovare le **componenti fortemente connesse** di un digrafo in due semplici passi.

Le intuizioni fondamentali sono:

- le **componenti fortemente connesse** di un digrafo G sono le stesse del grafo trasposto (ottenuto invertendo gli archi) G^R
- **kernel DAG** → contrarre ogni componente fortemente connessa in un singolo vertice

L'idea è quindi di:

- calcolare l'ordinamento topologico del kernel DAG
- eseguire la DFS nel digrafo di partenza G considerando i vertici in base all'ordinamento topologico

Fase 1 → per ottenere un ordinamento topologico del digrafo inverso è necessario calcolare il reverse postorder di G^R usando la DFS.

Fase 2 → eseguo la DFS sul digrafo originario G , considerando i vertici in base all'ordinamento topologico del digrafo inverso appena calcolato, durante la DFS verranno segnati gli id delle componenti fortemente connesse di ogni vertice attraversato.

Così facendo si ottengono tutte le componenti fortemente connesse del digrafo G .

Prestazioni: questo algoritmo calcola le componenti fortemente connesse di un digrafo in un tempo proporzionale ad $E+V$ (somma del numero di archi e nodi)

Il bottleneck nel **tempo di esecuzione** è dato dalla duplice esecuzione della DFS + il tempo per calcolare il digrafo inverso G^R

La **correttezza** dell'algoritmo è data dal fatto che riesco a scoprire quali sono i nodi raggiungibili sia nel grafo inverso che quello originale.

L'**implementazione** dell'algoritmo è molto semplice e molto simile a quella delle componenti connesse nei grafi non orientati.

Riassunto video maestro

È un algoritmo che ci permette di trovare le **componenti fortemente connesse** in un grafo orientato.

Ci serviremo di **due** DFS:

- La prima troverà l'ordinamento topologico
- La seconda la utilizzeremo dopo aver trasposto il grafo (direzione degli archi invertita)

Consideriamo tutti i vertici nell'ordinamento topologico e costruiamo le componenti fortemente connesse.

Utilizzeremo uno stack, la **prima DFS** andrà a popolare lo stack, successivamente dopo aver trasposto il grafo si faranno dei pop() sullo stack e si farà la **seconda DFS**, così si costruiscono le componenti fortemente connesse.

Ogni nodo che possiamo raggiungere con l'aiuto della DFS nel grafo formerà una singola componente fortemente connessa.

https://www.youtube.com/watch?v=Qdh6-a_2MxE&list=PL1fp4FtjD21Vqu2rOMBk28VcOtev7gvpN&index=8

Conclusioni sulla DFS

L'algoritmo DFS è molto importante perché risolve numerosi problemi riguardanti i **digrafi**, raggruppati in 3 categorie:

- **Problemi di raggiungibilità a singolo nodo sorgente di un digrafo** → si utilizza una singola DFS
- **Ordinamento topologico in un grafo orientato aciclico (DAG)** → si utilizza una singola DFS
- **Trovare le componenti fortemente connesse in un digrafo** → algoritmo di Kosaraju-Sharir (doppia DFS)

▼ Minimum Spanning Tree (MST) - Alberi di copertura di costo minimo

Un **albero di copertura** di un grafo G è un sottografo T che è **connesso, aciclico** e che include tutti i nodi di G .

Una soluzione al problema dell'MST potrebbe essere l'utilizzo di una tecnica **brute force** che consiste nel cercare tutti gli alberi di copertura di un dato grafo, e scegliere quello con peso minimo, ma questa è **una soluzione non efficiente**.

Algoritmo Greedy

Un algoritmo **greedy** è un algoritmo che date N soluzioni, sceglie quella ottimale nello spazio di ricerca locale.

Assumiamo delle semplificazioni:

- il grafo è connesso
- i pesi degli archi sono distinti

Queste semplificazioni fanno in modo che il **MST** esiste ed è unico.

Definiamo **taglio** in un grafo una partizione dei suoi nodi in due sottoinsiemi disgiunti non vuoti.

Ogni taglio determina un **insieme di taglio** (cut-set), cioè un insieme di **crossing edge**, ossia archi che connettono un nodo di un insieme con un nodo dell'altro insieme.

Proprietà di taglio → dato un qualsiasi taglio, il **crossing edge** di peso minimo deve essere nell'albero di copertura minima.

L'algoritmo greedy di base per la ricerca dell'MST si basa sui seguenti passi:

1. si inizia con tutti gli archi grigi
2. si cerca un taglio che non ha crossing edge neri e si colora il crossing edge di peso minimo di nero
3. si ripete fino a quando non si hanno $V-1$ archi neri

Di questo algoritmo greedy esistono diverse implementazioni che si differenziano a seconda della scelta dei tagli o sulla ricerca dell'arco di peso minimo:

- **Algoritmo di Kruskal**
- **Algoritmo di Prim**
- **Algoritmo di Boruvka**

Cosa succede se eliminiamo le assunzioni di semplificazione?

- I pesi degli archi non sono tutti distinti → l'algoritmo greedy resta corretto anche in presenza di pesi uguali

- Se il grafo non è connesso → si cerca la **foresta di copertura di peso minimo**, cioè si cerca il MST per ogni componente connessa.

Rappresentazione di un grafo con archi pesati con liste di adiacenza

Rappresentiamo il grafo mantenendo un array di liste indicizzato dai nodi del grafo → in ogni posizione dell'array avremo la lista degli archi incidenti nel nodo considerato.

<https://www.youtube.com/watch?v=4ZIRH0eK-qQ>

Algoritmo di Kruskal

È l'implementazione che utilizza una **priority queue** per mantenere gli archi, in particolare sceglie di volta in volta l'arco di **peso minimo** verificando se l'aggiunta di esso crea o meno un **ciclo** nel MST.

La priority queue avrà come radice l'arco col peso minimo.

Si considerano gli archi in **ordine crescente di peso** inseriti nella priority queue.

Come verificare se l'aggiunta di un arco all'albero crea o meno un ciclo?

Abbiamo diverse soluzioni:

- **usare la DFS** → l'albero avrà al più $V-1$ archi, l'algoritmo ha costo proporzionale al numero di nodi V
- **usare una struttura dati Union-Find** → soluzione più efficiente (proporzionale al logaritmo iterato di V)
 - Si mantiene un insieme per ogni componente connessa all'albero T
 - Se due nodi v e w sono nello stesso insieme, allora l'aggiunta dell'arco $v-w$ creerebbe un ciclo nell'albero
 - Per aggiungere l'arco $v-w$ a T basta unire gli insiemi che contengono v e w

Prestazioni: questo algoritmo calcola il **MST** in un tempo proporzionale a $E \log E$ nel **worst case**, ciò è dovuto all'utilizzo della priority queue, infatti ogni volta che si effettua un **delete min** la coda va riordinata in un tempo $\log E$, con frequenza pari a E (numero di archi)

Se gli archi sono già ordinati, l'ordine di crescita è pari a $E \log^* V$ (valore ≤ 5 per $V < 2^{65536}$), ciò è dovuto all'operazione di find.

Algoritmo di Prim

È un'altra implementazione di tipo greedy per trovare **alberi di copertura di peso minimo** in un **grafo non orientato** a pesi non negativi.

Si parte dal vertice iniziale (convenzione 0) e si va ad accrescere l'albero T aggiungendo ad esso l'arco di **peso minimo** che incide esattamente in uno dei nodi di T .

Si ripetono queste operazioni fino a che T non avrà $V-1$ archi.

La sfida principale è trovare l'arco di peso minimo che ha esattamente un endpoint nell'albero T nel minor tempo possibile, abbiamo diverse complessità:

- Costo proporzionale ad E se si fa una **ricerca esaustiva (eager)** su tutti gli archi del grafo
- Costo linearitmico $E \log E$ se si utilizza una **priority queue (approccio lazy)**

Distinguiamo due principali implementazioni dell'algoritmo:

- **Implementazione Lazy**
- **Implementazione Eager**

Implementazione Lazy

Si utilizza una **min priority queue** che contiene gli archi che incidono su almeno un nodo dell'albero T :

- Le chiavi della priority queue sono gli archi, la logica di priorità è data dal peso degli archi
- Si utilizza l'operazione di **delete min** per determinare l'arco successivo da aggiungere all'albero T
- Si ignorano gli archi che hanno entrambi gli endpoint già presenti nell'albero T
- Se l'arco non è da ignorare, sia w il nodo endpoint dell'arco ancora non inserito nell'albero T :
 - Si aggiunge alla priority queue ogni arco incidente in w (assumendo che gli altri endpoint non siano in T)
 - Si aggiunge w all'albero T e si segna w

Prestazioni: questa implementazione calcola il MST in un tempo proporzionale ad $E \log E$ e utilizza uno spazio di memoria extra proporzionale ad E nel **worst case**

Implementazione Eager

Quest'approccio utilizza una **priority queue** dei nodi connessi da un arco all'albero T.

La priorità di un nodo v è il peso dell'arco più corto che connette v all'albero T.

Anche in questo caso l'algoritmo termina una volta aggiunti esattamente $V-1$ archi all'albero T.

Per questo algoritmo abbiamo bisogno di una **min priority queue** che supporti le seguenti operazioni:

- Inserimento e rimozione della chiave minima
- Operazione di **decrease key** per cambiare la priorità di una chiave, dato il suo indice

Implementazione tramite **binary heap**: si parte dall'implementazione di una semplice MinPQ, oltre a mantenere 3 array paralleli `keys[]`, `pq[]` e `qp[]`:

- `keys[i]` è la priorità della chiave con indice `i`
- `pq[i]` è l'indice della chiave che indica la posizione del binary heap (indica la posizione in `qp`)
- `qp[i]` è la posizione nell'heap della chiave con indice `i`

Si utilizza `swim(qp[i])` per implementare l'operazione di **decreaseKey**

Prestazioni: dipendono da come è implementata la priority queue:

- L'implementazione con array non ordinato è ottimale per grafi densi perché i costi sono proporzionali al numero dei vertici e non di archi
- Il **binary heap** è molto veloce per grafi sparsi
- Un **heap 4-way** è adatto in situazioni performance-critical
- Il **Fibonacci heap** è il migliore dal punto di vista teorico secondo un'analisi ammortizzata, ma ha una complessità implementativa notevole rispetto alle altre

▼ Clustering

K-clustering: lo scopo è dividere un insieme di oggetti in k gruppi coerenti. Si utilizza una **distance function** che permette di calcolare numericamente la vicinanza tra due oggetti.

L'obiettivo è dividere gli oggetti in cluster tali che gli oggetti posti in cluster diversi siano molto distanti.

Un esempio è il **single-link** clustering:

- **Single-link:** la distanza tra due cluster è calcolata come la distanza tra i due oggetti più vicini appartenenti ai due cluster diversi
- **Single-link clustering:** dato un intero k, trovare k-clustering che massimizzi tale distanza tra i due cluster più vicini

Questo è proprio l'algoritmo di **Kruskal**, che termina quando si ottengono k componenti connesse:

- inizialmente si formano V cluster contenenti un solo oggetto
- Si cerca la coppia di oggetti più vicina e si fa un merge di cluster
- Si ripete finché non si ottengono k cluster

▼ Algoritmi di cammino minimo

Dato un grafo orientato, quindi un **digrafo**, vogliamo trovare il cammino minimo che collega un nodo sorgente **s** e un nodo destinazione **t**.

Una delle applicazioni è per esempio la ricerca del percorso più breve (Google Maps).

Esistono numerose varianti di cammino minimo:

- in base ai nodi:
 - **Singola sorgente** → cammino minimo da s a tutti gli altri nodi
 - **Singolo link** → da ogni nodo s ad un nodo t
 - **Sorgente-sink** → da un qualsiasi nodo s ad un altro t
 - **Tutte le coppie** → cammino minimo tra tutte le possibili coppie di nodi
- in base alle **restrizioni sugli archi**:
 - Pesi **non negativi**
 - Pesi **euclidei**
 - Pesi **arbitrari**

- in base alla presenza di **cicli**:
 - Cicli orientati **non ammessi**
 - Cicli **non negativi**

Effettuiamo la seguente **semplificazione**: esiste sempre un cammino minimo tra un nodo sorgente s e ogni altro nodo v del digrafo.

Implementazione: rappresentazione con liste di adiacenza

Si utilizza la rappresentazione con **array di liste di adiacenza**, per ogni nodo i abbiamo una lista di archi orientati che partono da tale nodo

SHORTEST PATHS TREE - SPT

Per implementare un algoritmo che calcoli i **cammini minimi** possiamo pensare ad un albero dei cammini minimi, detto **shortest paths tree - SPT**.

L'albero dei cammini minimi può essere rappresentato con due array indicizzati rispetto ai nodi:

- $\text{distTo}[v] \rightarrow$ array che mantiene le lunghezze dei cammini minimi dal nodo s ad ogni nodo v
- $\text{edgeTo}[v] \rightarrow$ array che mantiene l'ultimo arco del cammino minimo tra s e v

Gli indici di questi array sono i nodi dell'SPT.

Rilassamento di un arco

Dato un arco $e = v \rightarrow w$, siano:

- $\text{distTo}[v]$: la **lunghezza** del cammino minimo conosciuto tra s e v
- $\text{distTo}[w]$: la **lunghezza** del cammino minimo tra s e w
- $\text{edgeTo}[w]$: l'ultimo arco entrante in w nel cammino minimo tra s e w

Se $e = v \rightarrow w$ ci dà il cammino minimo per w attraverso v, allora aggiornare $\text{distTo}[w]$ e $\text{edgeTo}[w]$

"Rilassare l'arco" significa verificare se il miglior cammino per andare da s a w è quello che va da s ad un certo nodo v e poi passare dall'arco che unisce v e w, se ciò è vero allora è possibile aggiornare la struttura dati.

Condizioni ottimali dei cammini minimi

Sia G un digrafo con archi pesati: $\text{distTo}[]$ è l'array che mantiene le distanze dei cammini minimi a partire dal vertice s se e solo se:

- $\text{distTo}[s] = 0$, cioè la distanza da s a s è 0
- Per ogni vertice v , $\text{distTo}[v]$ è la lunghezza di un certo cammino da s a v
- Per ogni arco $e = v \rightarrow w$ si ha che $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$

Algoritmo generico per trovare i cammini minimi

Algoritmo per calcolare **SPT** a partire da s :

- Inizializzare $\text{distTo}[s] = 0$ e $\text{distTo}[v] = \infty$ per ogni vertice v
 - ripetere fino a quando non sono soddisfatte le condizioni ottimali
 - rilassare ogni arco

Di questo algoritmo esistono diverse implementazioni:

- **Algoritmo di Dijkstra**: per digrafi con pesi non negativi
- **Algoritmo con ordinamento topologico**: per digrafi che non hanno cicli diretti
- **Algoritmo di Bellman-Ford**: per digrafi che non hanno cicli negativi

▼ Algoritmo di Dijkstra

Si inizializza l'algoritmo ponendo $\text{distTo}[s] = 0$ e tutte le altre distanze ad un valore **infinito** positivo.

Successivamente si rilassano ripetutamente gli archi e si aggiungono ai SPT i nodi, si continua fino a quando:

- non sono stati inseriti tutti i nodi
- oppure se non ci sono più nodi da inserire con un valore di distTo finito.

Questo algoritmo garantisce il calcolo dell'SPT in un digrafo con pesi non negativi, infatti:

- ogni arco $e = v \rightarrow w$ è rilassato esattamente **una volta** (quando il nodo v è rilassato), così facendo la proprietà $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$ è rispettata.
- la diseguaglianza è rispettata fino al termine dell'algoritmo perché:

- $\text{distTo}[w]$ non potrà aumentare perché abbiamo **valori monotoni decrescenti**
- $\text{distTo}[v]$ non può cambiare perché **scegliamo il valore di $\text{distTo}[]$ minore ad ogni passo**

```

if (distTo[w] ≤ distTo[v] + e.weight()):
    vai_sempre()
else:
    distTo[w] = distTo[v] + e.weight()

```

Quindi al termine dell'algoritmo la condizione ottimale dei cammini minimi è rispettata.

Questo algoritmo utilizza una priority queue per scegliere l'arco da aggiungere al cammino minimo

Quale implementazione di Priority Queue scegliere?

Dipende dai costi delle operazioni di insert, delete min e decrease key

| PQ implementation | insert | delete-min | decrease-key | total |
|------------------------|-------------|------------------|--------------|------------------|
| unordered array | 1 | V | 1 | V^2 |
| binary heap | $\log V$ | $\log V$ | $\log V$ | $E \log V$ |
| d-way heap | $\log_d V$ | $d \log_d V$ | $\log_d V$ | $E \log_{E/V} V$ |
| Fibonacci heap | 1^\dagger | $\log V^\dagger$ | 1^\dagger | $E + V \log V$ |

\dagger amortized

Le osservazioni sono le stesse fatte per l'algoritmo di Prim, infatti i due algoritmi si somigliano perché fanno parte della stessa famiglia.

La differenza sta nella regola usata per scegliere il successivo nodo da aggiungere nell'albero:

- **Prim** → prende il nodo più vicino all'albero tramite archi non orientati

- **Dijkstra** → prende il nodo più vicino alla sorgente secondo un cammino orientato

▼ Algoritmo per digrafi pesati aciclici (DAG)

Dato un digrafo che non ha cicli diretti, la ricerca dei cammini minimi da un nodo sorgente a tutti gli altri nodi è più semplice, infatti basterà:

- considerare i nodi in base ad un ordinamento **topologico**
- rilassare gli archi uscenti da ognuno di essi

L'algoritmo basato su ordinamento topologico calcola lo SPT in un qualsiasi digrafo pesato aciclico in un tempo **proporzionale ad E+V** (somma di archi e nodi), ma i pesi possono essere negativi.

- Ogni arco $e = v \rightarrow w$ è rilassato esattamente **una volta** (quando il nodo v è rilassato), così facendo la proprietà $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$ è rispettata.
- La disuguaglianza è rispettata fino al termine dell'algoritmo perché:
 - $\text{distTo}[w]$ non potrà aumentare perché abbiamo **valori monotoni decrescenti**
 - $\text{distTo}[v]$ non cambia grazie all'ordinamento **topologico**, nessun arco entrante in v verrà rilassato prima che vengano rilassati i suoi archi uscenti.

Ricerca cammini massimi

Il problema di ricerca dei cammini massimi si risolve in maniera equivalente a quello dei cammini minimi, basterà invertire il segno della disuguaglianza nel metodo relax, e di conseguenza:

- negare i pesi
- trovare i cammini minimi
- negare i pesi risultanti

L'algoritmo basato su ordinamento topologico funziona anche con pesi negativi.

▼ Algoritmo di Bellman-Ford

L'algoritmo di Dijkstra non funziona con digrafi a pesi negativi perché si potrebbe selezionare un cammino minimo sbagliato per un dato nodo se si incontrano archi a peso negativo.

Definiamo **ciclo negativo** un ciclo di archi orientati la cui somma dei pesi è **negativa**.

Uno SPT esiste se e solo se il digrafo non presenta cicli negativi (assumendo che tutti i nodi siano raggiungibili dalla sorgente s).

L'**algoritmo di Bellman-Ford** si basa sui seguenti passi:

- inizializzare $distTo[s] = 0$ e $distTo[v] = \infty$
- ripetere V volte:
 - rilassare ogni arco

Questo algoritmo calcola l'SPT di un digrafo pesato che non ha cicli negativi in un tempo proporzionale a $E \times V$.

Infatti, dopo il passo i -esimo si trova il cammino che è breve tanto quanto un qualsiasi cammino minimo contenente un numero di archi $\leq i$.

Se la distanza $distTo[v]$ non cambia durante il passo i -esimo, non c'è bisogno di rilassare alcun arco uscente da v nel passo $i+1$

Implementazione FIFO: è possibile mantenere una **coda** dei nodi (presi una sola volta) la cui distanza dalla sorgente non è cambiata.

Ne segue che il tempo di esecuzione resta proporzionale ad $E \times V$ nel **worst case**, ma nell'average case l'algoritmo risulta molto più veloce, infatti tende a $E + V$

Sommario delle prestazioni dei vari algoritmi

| algorithm | restriction | typical case | worst case | extra space |
|-------------------------------|---------------------|--------------|------------|-------------|
| topological sort | no directed cycles | $E + V$ | $E + V$ | V |
| Dijkstra (binary heap) | no negative weights | $E \log V$ | $E \log V$ | V |
| Bellman-Ford | no negative cycles | $E V$ | $E V$ | V |
| Bellman-Ford (queue-based) | | $E + V$ | $E V$ | V |

I cicli **diretti e pesi negativi** rendono il problema più complesso e le soluzioni meno prestanti, mentre i cicli **negativi** rendono il problema **irrisolvibile**.

L'algoritmo di Bellman-Ford può essere usato per trovare eventuali cicli negativi in un digrafo pesato con alcuni pesi negativi, infatti se esiste un ciclo negativo l'algoritmo si bloccherà in un loop infinito aggiornando gli array `distTo[]` e `edgeTo[]` dei nodi nel ciclo.

Quindi se un qualsiasi nodo v viene aggiornato al passo V , allora esiste un ciclo negativo e sarà sufficiente fare un trace back degli archi `edgeTo[v]` per trovarlo.

Sommario degli algoritmi di cammino minimo

- Digrafi con pesi NON negativi → Dijkstra, complessità quasi lineare
- Digrafi pesati aciclici → algoritmo basato su ordinamento topologico, complessità lineare e i pesi possono essere negativi
- Digrafi con pesi negativi e/o cicli negativi → Bellman-Ford
 - se NON sono presenti cicli negativi → trovare i cammini minimi con Bellman-Ford
 - se sono presenti → trovare il ciclo negativo con Bellman-Ford