ALGORITMI

Problemi e Algoritmi

Problema

Un problema computazionale esprime una relazione tra un'istanza, cioè l'insieme dei valori di input (in termini assoluti) **e una soluzione**, cioè l'insieme dei valori di output (in termini dell'istanza)

Algoritmo

Un algoritmo è una procedura di calcolo che dato un input, produce una soluzione e si dice ***corretto*** se per ogni istanza è in grado di raggiungere la fine e produrre un output adeguato, cioè che presenta la corretta relazione che intercorre tra istanza e soluzione.
L'algoritmo è dunque una procedura di calcolo ben definita ed è basata sulla macchina ideale RAM.

Random Access Machine

La Random Access Machine (più elementare di Von Neumann) **è un modello ideale di calcolatore** (mono-processore), che tiene conto della presenza di una memoria RAM. In questo modello, le operazioni (`read`, `load`, `add`, `store`) costituiscono un linguaggio macchina elementare, sono eseguite in ordine sequenziale e hanno costo unitario.
Tali operazioni possono essere descritte attraverso uno pseudocodice elementare.

Algoritmi di ordinamento

I problemi di ordinamento hanno come input un'istanza di elementi, che deve essere ordinata secondo una relazione d'ordine e come output una permutazione dell'istanza, che risulta così ordinata.

Gli algoritmi che risolvono questi problemi possono presentare proprietà diverse e basarsi su strategie diverse.

Proprietà

Operazione in loco

Gli algoritmi di ordinamento si dice che **operano in loco se non necessitano di copiare l'istanza fornita in strutture dati diverse da quella di input**.

Stabilità

Gli algoritmi di ordinamento si dicono **stabili se non modificano l'ordine degli elementi che hanno stesso valore**.

Incrementali

Gli algoritmi di ordinamento si dicono **incrementali se sono basati sulla strategia per cui la soluzione viene costruita incrementalmente a partire da un insieme più piccolo di elementi**. Si basano quindi sul fatto che molti problemi hanno la caratteristica per cui la soluzione di un'istanza di dimensione $n-1$ può essere utile per risolvere un'istanza di dimensione n.

Un algoritmo di ordinamento basato su questa strategia è l'Insertion Sort.

Strategia

Tecnica Greedy

La tecnica greedy ("golosa") **consiste nel scegliere l'alternativa più appetibile, cioè fare una scelta localmente ottima**, ma che non sempre lo è anche a livello globale. Tale scelta viene fatta **guardando solo una parte dell'input** e non l'intera istanza, producendo una parte della soluzione e non quella dell'intero problema.

Un algoritmo di ordinamento basato su questa tecnica è il Selection Sort.

Divide et impera

La tecnica divide et impera (o divide and conquer) **consiste nel suddividere il problema in più sottoproblemi che possono essere risolti in maniera ricorsiva**, suddividendoli a loro volta **fino** ad arrivare ad un problema la cui soluzione è banale, che rappresenta il **caso base**. Risolvere quindi i piccoli problemi separatamente e poi unire le soluzioni per ottenere quella del problema principale.

È una tecnica ricorsiva, la cui ricorsione ad ogni passo

- *divide* l'istanza in due o più minori
- *impera* lanciando l'algoritmo sulle istanze minori
- *combina* unendo le soluzioni delle istanze più piccole e producendo la soluzione dell'istanza maggiore.

Un algoritmo di ordinamento basato su questa tecnica è il Merge Sort.

Algoritmi noti

Algoritmo Selection Sort

L'algoritmo Selection Sort utilizza la *tecnica greedy* per l'ordinamento di un array. La strategia che usa consiste nel selezionare l'elemento più piccolo (più grande, se l'ordinamento è decrescente) dell'istanza e posizionarlo all'inizio dell'array, poi continuare ad eseguire tale operazione sugli altri elementi, escludendo ogni volta quelli già ordinati, che si trovano all'inizio.

Questo algoritmo **opera in loco ed è stabile**.
Il costo è **quadratico** $\boldsymbol{\vartheta(n^2)}$, nel ***caso migliore, peggiore e medio***, in quanto la sua implementazione (non ricorsiva) presenta cicli annidati.

Algoritmo Insertion Sort

L'algoritmo Insertion Sort utilizza la *strategia incrementale* ed è basato sull'operazione di inserimento.

Questo algoritmo **opera in loco ed è stabile**.
Inoltre, è

- **efficiente su piccole istanze**, data la sua complessità;
- **adattivo**, cioè è più veloce su istanze già parzialmente ordinate
- **online**, cioè è utilizzabile anche con input disponibile parzialmente.

Il costo è

- nel ***caso peggiore*** **quadratico** $\boldsymbol{\vartheta(n^2)}$: la sequenza è ordinata inversamente
- nel ***caso migliore*** **lineare** $\boldsymbol{\vartheta(n)}$: la sequenza è già ordinata
- nel ***caso medio*** **quadratico** $\boldsymbol{\vartheta(n^2)}$: l'inserimento avviene nel mezzo.

Algoritmo Merge Sort

L'algoritmo Merge Sort utilizza la *strategia divide et impera*, prendendo come caso base, quello in cui l'istanza da ordinare è costituita da un solo elemento, quindi può essere considerata ordinata.
La procedura completa consiste nel dividere l'istanza in due e sostituire l'istanza originale con l'unione ordinata delle due.

Questo algoritmo **non opera in loco, ma è stabile**.
Il costo è $\boldsymbol{\vartheta(n \log n)}$, nel ***caso migliore, peggiore e medio***.

Algoritmo Quick Sort

L'algoritmo Quick Sort utilizza la *strategia divide et impera*, prendendo come caso base, quello in cui l'istanza da ordinare è costituita da due elementi ordinati oppure due non ordinati.
La procedura completa consiste nel prendere un elemento di riferimento, chiamato ***pivot***, e dividere l'istanza in due minori contenenti una i valori

minori o uguali al pivot e una quelli maggiori. Si lancia poi la procedura ricorsivamente e al termine l'array risulterà già ordinato (senza combinare).

Questo algoritmo **opera in loco, ma non è stabile**.
Il costo è

- nel ***caso peggiore*** $\boldsymbol{\vartheta(n \log n)}$: il pivot è il valore minimo o massimo e quindi l'algoritmo ricorrerà su un array con meno di un elemento e uno degenere
- nel ***caso migliore*** **quadratico** $\boldsymbol{\vartheta(n^2)}$: il pivot è il valore medio e quindi l'algoritmo ricorrerà su due array bilanciati
- nel ***caso medio*** $\boldsymbol{\vartheta(n \log n)}$: il pivot è un valore diverso dai precedenti e quindi l'algoritmo ricorrerà su due array sbilanciati (non eccessivamente).

Per evitare che i costi peggiori non coincidano con disposizioni notevoli degli elementi, si introduce una *randomizzazione per la scelta del pivot*, in quanto il costo dell'intero algoritmo dipende da questa scelta nella procedura di partition (costo $\vartheta(n)$).
È possibile gestire eventuali costi elevati con un'*ammortizzazione* della complessità: il costo di una partition sbilanciata può essere assorbito da quello di una bilanciata.

Algoritmo Heap Sort

L'algoritmo Heap Sort utilizza la *struttura heap*.
La procedura, per l'ordinamento di un array, consiste nel ciclare l'array riducendo, ad ogni estrazione del primo elemento, la dimensione dell'array, fino a quando non resta un solo elemento, che per definizione è ordinato. Ad ogni ciclo trasformare l'array in un heap, lanciando la procedura build_max_heap, poi estrarre il primo elemento per posizionarlo in coda e lanciare la procedura max_heapify per ripristinare le proprietà dell'heap. Il ripetuto spostamento del primo elemento in fondo all'array fa si che alla fine dell'intera operazione, gli elementi siano in ordine.

Questo algoritmo **opera in loco, ma non è stabile**, in quanto potrebbe alterare una eventuale relazione di parentela tra gli elementi.
Il costo è $\boldsymbol{\vartheta(n \log n)}$, nel ***caso migliore, peggiore e medio***.

Algoritmo Tree Sort

L'algoritmo Tree Sort utilizza la *struttura alberi binari di ricerca*.
La procedura, per l'ordinamento di un array, consiste nel ciclare l'array per crearne il relativo albero, poi eseguire lanciare la procedura tree_to_array per riversare l'albero in un array, che quindi risulterà ordinato.

Il costo è nel ***caso peggiore*** **quadratico** $\boldsymbol{\vartheta(n^2)}$, oppure $\boldsymbol{\vartheta(n \log n)}$ se l'albero è bilanciato.

Tabella di confronto

caso	*migliore*	*medio*	*peggiore*	*opera in loco*	*è stabile*
Selection Sort	$\vartheta(n^2)$			Si	Si
Insertion Sort	$\vartheta(n)$	$\vartheta(n^2)$		Si	Si
Merge Sort	$\vartheta(n \log n)$			No	Si
Quick Sort	$\vartheta(n \log n)$		$\vartheta(n^2)$	Si	No
Heap Sort	$\vartheta(n \log n)$			Si	No
Tree Sort	$\vartheta(n \log n)$ *oppure* $\vartheta(n^2)$			Si	Si

STRUTTURE DATI

Strutture di dati

Una struttura di dati è un contenitore di dati organizzato secondo una strategia o disciplina, che ottimizza l'accesso all'input e la produzione di output.

Con queste strutture è possibile definire Tipi Astratti di Dato, derivati da quelli primitivi già esistenti.

Abstract Data Type

Un ADT è una descrizione di un tipo di dato indipendente dalla sua realizzazione in un linguaggio di programmazione. È caratterizzato da un dominio di interesse e altri di supporto; un insieme di costanti; una collezione di operazioni.

Realizzazione

Un ADT può essere realizzato, quindi implementato in un linguaggio attraverso la definizione di tipi e strutture corrispondenti ai domini; costrutti che rappresentano le costanti; funzioni che realizzano le operazioni.

Stack

La pila è un ADT basato sulla strategia LIFO Last-In-First-Out.

Caratteristiche

- domini:
  - di interesse: P insieme delle pile
  - di supporto:
    - Z insieme degli elementi della pila
    - B insieme dei booleani $\{true, false\}$
- costanti: pila vuota
- operazioni:
  - is_empty: $P \rightarrow B$ verifica se la pila è vuota
  - push: $P \times Z \rightarrow P$ inserisce un elemento in testa alla pila
  - pop: $P \rightarrow P \times Z$ rimuove e restituisce l'elemento affiorante
  - top: $P \rightarrow Z$ legge l'elemento affiorante
  - empty: $P \rightarrow P$ svuota la pila
  - size: $P \rightarrow Z$ restituisce il numero di elementi presenti

Tutte le operazioni hanno costo costante $\vartheta(1)$.

Realizzazione

Una pila può essere realizzata attraverso un oggetto formato da un array, che ne rappresenta gli elementi **e un campo**, che punta all'elemento affiorante.

Queue

La coda è un ADT basato sulla strategia FIFO First-In-First-Out.

Caratteristiche

- domini:
  - di interesse: Q insieme delle code
  - di supporto:
    - Z insieme degli elementi della coda
    - B insieme dei booleani {true, false}
- costanti: coda vuota
- operazioni:
  - is_empty: $Q \rightarrow B$ verifica se la coda è vuota
  - enqueue: $Q \times Z \rightarrow Q$ inserisce un elemento in fondo alla coda
  - dequeue: $Q \rightarrow Q \times Z$ rimuove e restituisce l'elemento più vecchio
  - front: $Q \rightarrow Z$ legge l'elemento più vecchio
  - empty: $Q \rightarrow Q$ svuota la coda
  - size: $Q \rightarrow Z$ restituisce il numero di elementi presenti

Realizzazione

Una coda può essere realizzata attraverso un oggetto formato da un array, che ne rappresenta gli elementi **e da due campi**, che puntano uno all'elemento più vecchio e uno alla prima posizione utile per l'inserimento.

Liste

La lista è un ADT in cui gli oggetti sono disposti secondo un ordine sequenziale importante.

Caratteristiche

- domini:
  - di interesse: L insieme delle liste
  - di supporto:
    - I insieme degli iteratori
    - Z insieme degli elementi della lista
    - B insieme dei booleani {true, false}
- costanti:
  - lista vuota
  - iteratore di default con valore non valido -1

- operazioni:
  - modifica
    - insert: L×Z→L inserisce un elemento in testa alla lista
    - add: L×Z→L inserisce un elemento in coda alla lista
  - aggiornamento
    - delete: L×I→L rimuove l'elemento indicato dall'iteratore
    - delete: L×Z→L rimuove l'elemento indicato dal valore
    - insert_before: L×I×Z→L inserisce un elemento prima dell'iteratore
    - add_after: L×I×Z→L inserisce un elemento dopo l'iteratore
    - empty:L→L svuota la lista
  - consultazione
    - head:L→I restituisce il primo elemento
    - last:L→I restituisce l'ultimo elemento
    - next:L×I→I restituisce l'elemento dopo l'iteratore
    - prev:L×I→I restituisce l'elemento prima dell'iteratore
    - info:L×I→Z restituisce l'elemento associato all'iteratore
    - search:L×Z→I restituisce l'iteratore associato all'elemento
    - is_empty:L→B verifica se la lista è vuota
    - size:L→Z restituisce il numero di elementi presenti

Realizzazione

Una lista può essere realizzata in due modi: tramite array o tramite oggetti e riferimenti. Inoltre, **può essere gestita attraverso tre strategie:**

- *lista semplicemente concatenata*, che consente uno scorrimento efficiente, ma solo in avanti e non indietro;
- *lista doppiamente concatenata*, che consente uno scorrimento efficiente in avanti e indietro;
- *lista con accesso agli estremi*, che consente un accesso veloce al primo e all'ultimo elemento.

Tutte le strategie consento accesso ai campi in tempi costanti $\vartheta(1)$.

Liste tramite array

Liste con tre array

Una lista può essere realizzata attraverso tre array, ad indicare gli elementi e i loro rispettivi precedenti e successivi, **e un campo**, che punta al primo elemento.

Questa strategia non si basa su alcuna disciplina, infatti le colonne sono indipendenti dall'ordine della lista e possono quindi esserci posizioni vuote o

con valori privi di significato (diversamente da pila e coda). Tale struttura permette dunque, di eseguire **operazioni semplici in tempi costanti**. Inoltre, **è possibile condividere i tre array con più liste**, in quanto non interferirebbero l'una con l'altra poiché utilizzerebbero posizioni diverse.

Per individuare le posizioni libere è però, necessaria una ulteriore lista, con i rispettivi tre array, i cui elementi non hanno significato, in quanto indicano posizioni utilizzabili. Risulta così che un inserimento nella lista principale equivale ad una rimozione dalla lista libera e una cancellazione dalla lista principale ad un inserimento nella lista libera. In questo modo è possibile svolgere le due *operazioni in tempo costante*, in quanto si evitano le ricerche delle celle libere lungo l'intera lista.

Liste con un array

La lista può anche essere realizzata attraverso un solo array, dato dall'unione dei tre, in modo da permettere la gestione di elementi eterogenei (di dimensioni di verse), introducendo un campo ne indica le dimensioni (celle successive occupate). Per l'inserimento è necessaria però, la ricerca dello spazio adeguato.

Le liste tramite array hanno lo stesso funzionamento dell'heap.

Liste tramite oggetti e riferimenti

Singly Linked List

Una lista semplicemente concatenata può essere realizzata attraverso un oggetto, che rappresenta l'intera lista, con l'attributo ***head***, puntatore al primo elemento. Ogni elemento è anch'esso un oggetto con gli attributi ***info*** a indicarne il valore e ***next*** puntatore al successivo.

Doubly Linked List

Una lista doppiamente concatenata può essere realizzata attraverso un oggetto, che rappresenta l'intera lista, con l'attributo ***head***, puntatore al primo elemento e l'attributo ***tail***, puntatore all'ultimo elemento. Ogni elemento è anch'esso un oggetto con gli attributi ***info*** a indicarne il valore, ***next*** puntatore al successivo e ***prev***, puntatore al precedente.

Sentinelle

Le liste realizzate in questo modo permettono di introdurre speciali iteratori, chiamati **sentinelle** (nodo fittizio), cioè il **primo iteratore, sempre presente, che non ha elemento associato** e coincide quindi con l'iteratore non valido.

Liste circolari

Tale iteratore consente di introdurre le **liste circolari**, in cui il precedente della sentinella è l'ultimo della lista e il successivo dell'ultimo è la sentinella. Si possono quindi facilmente ottenere il primo e l'ultimo elemento della lista a partire dalla sentinella: $\boldsymbol{NULL + next = first}$ & $\boldsymbol{NULL + prev = last}$. Si evita così lo scorrimento dell'intera lista e molte procedure risultano semplificate.

Alberi radicati

Definizione

Un albero radicato è un insieme di nodi su cui è definita una relazione binaria per cui ogni nodo ha un solo genitore, ad eccezione della radice che non ne ha, ed **esiste un cammino diretto da ogni nodo alla radice**.

Risulta perciò che non ci sono cammini ciclici.
Una radice può rappresentare, da sola, un albero.

Proprietà

Se un albero ha n nodi, ci sono $n - 1$ **archi** (relazioni figlio-genitore). I nodi si contano a partire da 0 e gli archi a partire da 1.
Il **cammino** è la sequenza di nodi in cui uno è genitore del successivo. La lunghezza del cammino è pari al numero di archi che intercorrono tra i nodi.
La **profondità** è la lunghezza del cammino dal nodo alla radice e il nodo più profondo ha profondità pari all'altezza dell'albero.
L'**altezza** dell'albero è pari alla profondità del nodo più profondo.

Ogni nodo ha un **grado**, dato dal numero di figli che ha.
Si dicono **fratelli**, i nodi che hanno stesso genitore e si trovano sullo stesso livello.
Si dice **foglia**, un nodo che non ha figli e quindi ha grado zero.
Si dice **nodo interno**, un nodo che non è foglia.

Un **albero ordinato** è un albero in cui l'ordine dei figli di ogni nodo è significativo.

Relazione di discendenza

Ogni nodo sul cammino è **antenato** di quello da cui si parte, che è a sua volta **discendente**.

L'insieme costituito da un nodo e i suoi discendenti costituisce un **sottoalbero radicato**. Il sottoalbero della radice è l'albero stesso; il sottoalbero di una foglia è la foglia stessa. Dunque, **ogni nodo ha un sottoalbero**.

Esempi di applicazione

Queste strutture dati vengono, ad esempio, usate in

- rapporti di ereditarietà (programmazione orientata agli oggetti)
- gerarchie organizzative di controllo
- rapporti di contenimento (directory di File System, nodi di una rete …)
- struttura sintattica di una frase.

Visite di un albero

Un albero può essere visitato secondo tre discipline:

- due ricorsive
  - **visita in preordine**: si processa un *nodo poi i suoi figli*, partendo dal sinistro e si effettuano le operazioni in ordine ***top-down***
  - **visita in postordine**: si processa un *nodo dopo i suoi figli* e si effettuano le operazioni in ordine ***bottom-up***
- una terza valida solo per gli alberi binari
  - **visita simmetrica**: si processa prima il *figlio sinistro, poi il nodo e poi il figlio destro*.

Cammino euleriano

Il **cammino euleriano** è il cammino eseguito sul "bordo" dell'albero, che permette di capire che *in tutte le visite l'ordine dei nodi visitati è lo stesso, mentre a cambiare è solo il momento in cui avvengono le computazioni sul nodo*.

- nella visita *in preordine*, le operazioni avvengono sulla ***sinistra*** del nodo, cioè quando si tocca il nodo per la prima volta
- nella visita *in postordine*, le operazioni avvengono sulla ***destra*** del nodo, cioè quando si lascia il nodo
- nella visita *simmetrica*, le operazioni avvengono ***sotto*** il nodo, cioè al termine del processo sul sottoalbero sinistro e prima di processare il sottoalbero sinistro.

Il cammino viene eseguito in tempi lineari $\vartheta(n)$.

Alberi di grado arbitrario

Un albero di grado arbitrario è un albero il cui numero massimo di figli dei nodi non è noto e l'ordine dei nodi non è significativo.

Caratteristiche

- domini:
  - di interesse: T insieme degli alberi
  - di supporto:
    - R insieme dei riferimenti che indicano le posizioni
    - Z insieme degli elementi dell'albero
    - B insieme dei booleani {true, false}
- costanti: albero vuoto
- operazioni:
  - modifica
    - add_root: $T \times Z \rightarrow T$ inserisce un nodo come radice
    - add_left: $T \times R \times Z \rightarrow T$ inserisce un nodo come figlio sinistro
    - add_right: $T \times R \times Z \rightarrow T$ inserisce un nodo come figlio destro
    - delete_leaf: $T \times R \rightarrow T$ rimuove una foglia
    - empty:$T \rightarrow T$ svuota l'albero
  - consultazione
    - root:$T \rightarrow R$ restituisce il riferimento alla radice
    - first_child:$T \times R \rightarrow R$ restituisce il riferimento al figlio sinistro
    - next_sibling:$T \times R \rightarrow R$ restituisce il riferimento al fratello destro
    - info:$T \times R \rightarrow Z$ restituisce valore del nodo
    - search:$T \times Z \rightarrow R$ restituisce il riferimento al nodo cercato
    - is_empty:$T \rightarrow B$ verifica se l'albero è vuoto
    - size:$T \rightarrow Z$ restituisce il numero di nodi presenti

Realizzazione

Un albero di grado arbitrario può essere realizzato attraverso due strategie**:** usando una lista per i figli di ogni nodo (prolissa) oppure usando la **strategia figlio sinistro fratello destro** (sintetica). Solitamente si usa questa seconda strategia, in quanto più sintetica e permette un facile accesso a tutti i nodi, anche senza conoscere la profondità dell'albero.

La radice avrà riferimento solo al figlio sinistro (e non anche al destro) e ogni nodo avrà riferimento al figlio sinistro e al fratello destro (oltre che al genitore).

Alberi binari

Un albero binario è un albero ordinato in cui ogni nodo ha solo figlio sinistro e destro, quindi grado massimo 2.

Proprietà

Un albero binario si dice **completo** se tutti i nodi hanno due figli.
Un albero binario si dice **incompleto** se l'ultimo livello è incompleto nella parte destra.

Se l'albero ha altezza h, allora ha
- 2^h foglie, quindi $h = \log_2 n°foglie$
- $2^h - 1$ nodi interni
- $2^{h+1} - 1$ nodi totali
- $2^{profondità}$ nodi nel livello.

Un albero binario completo o incompleto ha sempre un *numero dispari di nodi*.

Caratteristiche

- domini:
  - di interesse: T insieme degli alberi
  - di supporto:
    - R insieme dei riferimenti che indicano le posizioni
    - Z insieme degli elementi dell'albero
    - B insieme dei booleani {true, false}
- costanti: albero vuoto
- operazioni:
  - modifica
    - add_root: T×Z→T — inserisce un nodo come radice
    - add_left: T×R×Z→T — inserisce un nodo come figlio sinistro
    - add_right: T×R×Z→T — inserisce un nodo come figlio destro
    - delete_leaf: T×R→T — rimuove una foglia
    - empty:T→T — svuota l'albero
  - consultazione
    - root:T→R — restituisce il riferimento alla radice
    - left:T×R→R — restituisce il riferimento al figlio sinistro
    - right:T×R→R — restituisce il riferimento al figlio destro
    - info:T×R→Z — restituisce valore del nodo
    - search:T×Z→R — restituisce il riferimento al nodo cercato
    - is_empty:T→B — verifica se l'albero è vuoto
    - size:T→Z — restituisce il numero di nodi presenti

Realizzazione

Un albero binario può essere realizzato attraverso un oggetto, che rappresenta l'intero albero, con l'attributo ***root***, puntatore alla radice. Ogni nodo è anch'esso un oggetto con gli attributi ***info*** a indicarne il valore, ***parent*** puntatore al genitore, ***left*** puntatore al figlio sinistro e ***right*** puntatore al figlio destro.

Alberi binari di ricerca

Un albero binario di ricerca è un albero radicato in cui ogni nodo ha i discendenti nel sottoalbero sinistro con valore minore e i discendenti nel sottoalbero **destro con valore maggiore** al suo.

Operazioni

Le operazioni che si possono svolgere su un ABR sono le stesse di un albero binario, ma seguono alcune regole.

- modifica
  - inserimento

    Ogni nuovo elemento viene inserito come foglia, ma
    - se l'albero è vuoto, viene inserito come radice
    - se l'oggetto è uguale al nodo corrente, viene rifiutato
    - se l'oggetto è minore del nodo corrente, viene inserito a sinistra
    - se l'oggetto è maggiore del nodo corrente, viene inserito a destra.
  - cancellazione
    - una foglia si può sempre rimuovere
    - un nodo con un solo figlio si può rimuovere, collegando il genitore con il figlio
    - se il nodo ha due figli si cerca il minore nel sottoalbero destro e si scambiano e si rimuove il nodo.

    La cancellazione può essere implementata attraverso la procedura `tree_delete` (costo $\vartheta(h)$) e `tree_bypass` (costo $\vartheta(1)$) se il nodo ha solo un figlio.
- consultazione
  - calcolo del minimo o massimo
    - **il minimo è sempre la foglia più a sinistra**
    - **il massimo è sempre la foglia più a destra**.
  - ricerca valore

    La ricerca può essere effettuata ricorsivamente o meno e
    - se l'oggetto è minore del nodo corrente, viene cercato a sinistra
    - se l'oggetto è maggiore del nodo corrente, viene cercato a destra.

- verifica consistenza
  Un albero è ABR se una visita simmetrica produce una sequenza crescente di valori.
  La verifica può essere implementata attraverso la procedura abr_sym, che consiste nel contare i nodi dell'albero, riversarlo in un array e poi verificare che sia ordinato.

Le operazioni sugli ABR hanno tutti complessità $\vartheta(h)$, in particolare:
- nel ***caso peggiore*** **lineare** $\boldsymbol{\vartheta(n)}$: l'albero è sbilanciato
- nel ***caso migliore*** **logaritmico** $\boldsymbol{\vartheta(\log n)}$: l'albero è bilanciato
- nel ***caso medio*** **logaritmico**, per inserimento e ricerca.

Alberi rosso-neri

Definizione

Un albero rosso-nero è un ABR le cui proprietà garantiscono che sia bilanciato. Vengono, perciò usati come *strategia per mantenere gli ABR bilanciati* e quindi avere tempi di esecuzione logaritmici per tutte le operazioni. La strategia si basa sull'intento di *evitare che ci siano nodi con un solo figlio*, ponendo il figlio mancante come puntatore alla **sentinella**, nodo con valore nullo.

Proprietà

Un albero rosso-nero è un albero in cui
1. ogni nodo è rosso o nero
2. radice e sentinella sono nere
3. se un nodo è rosso, i figli sono neri
4. tutti i cammini dalla radice alla sentinella contengono lo stesso numero si nodi neri
   - ogni cammino ha almeno $k-1$ archi, dove k è il numero di nodi neri
   - il cammino più lungo alterna nodi neri e rossi e ha $2(k-1)$ archi.

L'altezza dell'albero è pari alla lunghezza del cammino più lungo $h = h + 1$ e contiene un sottoalbero completo di profondità $h' = h/2 - 1$.

Operazioni

Le operazioni che si possono svolgere su un ARN sono le stesse di un ABR, ma siccome non garantiscono la conservazione delle proprietà, sono necessarie, per inserimento e cancellazione, ***procedure di ripristino a valle***.
- consultazione: ricerca valore – calcolo minimo e massimo
- modifica: tree_insert – tree_delete

Le operazioni hanno costo logaritmico $\vartheta(\log n)$.

Rotazioni

Le rotazioni sono operazioni, eseguite in tempi costanti $\vartheta(1)$, **alla base del ripristino delle proprietà** dell'ARN, senza che queste o i colori dei nodi vengano alterati.

- left_rotate(t,x) si effettua scambiando il figlio destro con il nodo, che diventa figlio sinistro con i discendenti del figlio, scambiati di ordine
- right_rotate(t,y) si effettua scambiando il figlio sinistro con il nodo, che diventa figlio destro con i discendenti del figlio, scambiati di ordine.

Ripristino delle proprietà

Il ripristino delle proprietà di un ARN si basa sulle **rotazioni** e eventuali **ricolorazioni** a seconda del caso che si presenta.

Un **inserimento di un nodo foglia di colore rosso**

- se l'albero era vuoto, ***viola la proprietà 2***, per cui è sufficiente *ricolorare la radice di nero*
- altrimenti ***viola la proprietà 3***, per cui si presentano **tre casi**:
  (considerando che *esiste sempre un nodo zio* del nuovo nodo)
  1. il nuovo nodo è un figlio rosso sinistro (o destro) e lo zio è un figlio nero destro (o sinistro): c'è *disomogeneità* della posizione
  2. il nuovo nodo è un figlio rosso destro (o sinistro) e lo zio è un figlio nero destro (o sinistro): c'è *omogeneità* della posizione
  3. lo zio è anch'esso un figlio rosso.

  Per ripristinare le proprietà difronte a queste situazioni si procede così:
  1. per risolvere il **caso 1**, si effettua una *ricolorazione del nodo padre e nonno* e una *rotazione destra sul nonno*
     La complessità di questa soluzione è costante $\vartheta(1)$.
  2. per risolvere il **caso 2**, si effettua una *rotazione sinistra sul nodo padre* e si eseguono le operazioni per risolvere il *caso 1*
     La complessità di questa soluzione è costante $\vartheta(1)$.
  3. per risolvere il **caso 3**, si effettua una *ricolorazione del nodo padre, nonno e zio*, poi uno *scambio del nuovo nodo con il nonno*, che diventa il nuovo e si analizza la nuova situazione e si applica l'adeguata *soluzione* (uno dei tre casi). *Nel caso peggiore, si rientrerà sempre nel caso 3, risalendo alla radice*, dove sarà sufficiente la ricolorazione. Al termine ogni cammino risulterà incrementato di un arco.
     La complessità di questa soluzione è logaritmica $\vartheta(\log n)$.

Heap

L'heap è un array in cui gli elementi sono in rapporto con la loro posizione nell'array.

Proprietà

L'heap può rappresentare alberi binari completi o quasi completi.
Se codifica un albero quasi completo con n elementi, allora gli elementi da 0 a $\boldsymbol{\lfloor n/2 \rfloor - 1}$ **sono** ***nodi interni***.

A partire da un nodo con posizione nell'array i è possibile determinare i suoi attributi: parent $= \frac{\boldsymbol{i-1}}{\boldsymbol{2}}$; left $= \boldsymbol{2i+1}$; right $= \boldsymbol{2i+2}$ e i nodi che hanno posizione $\boldsymbol{i \geq \lfloor n/2 \rfloor}$ **sono** ***foglie***.

L'heap può essere di due tipi:
- **max-heap**, in cui tutti gli elementi hanno sempre *valore maggiore o uguale a quello dei figli*
- **min-heap**, in cui tutti gli elementi hanno sempre *valore minore o uguale a quello dei figli*.

Realizzazione

L'heap può essere realizzato **attraverso un oggetto**, che rappresenta l'intera struttura, con l'attributo ***size*** ad indicare la dimensione dell'array e il riferimento all'**array** stesso.
La dimensione dell'heap può essere gestita con una *gestione telescopica*.

Procedure

Le procedure caratteristiche dell'heap, con costi logaritmici, sono:
- max_heapify
  trasforma il sottoalbero radicato in un heap, se i due sottoalberi radicati sinistro e destro sono già heap, lavorando sul nodo dall'alto verso il basso
- build_max_heap
  trasforma un array in un heap, eseguendo max_heapify sui nodi non foglie, lavorando sull'heap dal basso verso l'alto.

Le procedure hanno costi logaritmici $\vartheta(\log n)$ e $\vartheta(n \log n)$.

Code di priorità

La coda di priorità è una collezione di elementi, ad ognuno dei quali è associato un valore di priorità, che ne definisce l'ordine.

Esempi di applicazione

Queste strutture dati vengono, ad esempio, usate in

- allocazione di risorse condivise ai processi, in cui gli elementi sono rappresentati dalle richieste di risorse e i processi in esecuzione che generano nuove richieste con priorità rappresentano l'inserimento
- sistema ad eventi, in cui gli eventi, con il loro tempo di accadimento, rappresentano gli elementi e la simulazione di un evento provoca l'inserimento di altri.

Caratteristiche

- domini:
  - di interesse: Q insieme delle code di priorità
  - di supporto:
    - Z insieme degli elementi della coda
    - B insieme dei booleani {true, false}
- costanti: coda vuota
- operazioni:
  - is_empty: Q→B — verifica se la coda è vuota
  - new_queue: Q→Q — crea una coda vuota
  - maximum: Q→Z — legge l'elemento con priorità massima
  - insert: Q×Z→Q — inserisce un elemento in fondo alla coda
  - extract_max: Q→Q — rimuove e restituisce l'elemento con priorità max

Le prime tre operazioni hanno costo lineare $\vartheta(1)$.

Realizzazione

Una coda di priorità può essere realizzata, in modo efficiente, **attraverso l'heap**, che permette di eseguire le operazioni in tempi lineari o logaritmici.

Potrebbero anche essere realizzate attraverso liste ordinate oppure liste non ordinata, ma non risulterebbe efficiente, in quanto non garantiscono di eseguire le operazioni di inserimento e rimozione entrambe con bassi costi.
Non vengono utilizzati gli alberi perché non sarebbe possibile identificare l'ultimo elemento e quindi non risulterebbe una strategia efficiente.
Possono però, essere usati gli *alberi binari di ricerca*, in quanto permettono di svolgere operazioni in tempi logaritmici, come l'heap.

Operazioni

Implementando le code di priorità attraverso gli heap, le operazioni risulteranno le stesse, ma necessitano un ripristino delle proprietà:

- insert
  aggiunge il nuovo elemento all'ultimo posto, poi risalendo viene riposizionato correttamente nella posizione $\lfloor i/2 \rfloor$
- extract_max
  rimuove il primo elemento (priorità max) e lo sostituisce con l'ultimo, che viene riposizionato correttamente chiamando max_heapify.

Le due operazioni hanno costi logaritmici $\vartheta(\log n)$.

Array associativi

Un array associativo (o mappa o dizionario) **è un insieme di coppie** <chiave,valore>, in cui le variabili omogenee sono accedute tramite chiavi omogenee, ma che possono essere di qualsiasi tipo, purché siano univoche.

Esempi di applicazione

Queste strutture dati vengono, ad esempio, usate in

- dati fiscali dei contribuenti, in cui la chiave è il codice fiscale e il valore è composto dai dati anagrafici, contributivi, ecc…
- dati satellite associati ad oggetti software, in cui la chiave è un oggetto e il valore è un insieme di dati specifici associati all'oggetto.

Caratteristiche

- domini:
  - di interesse: A insieme degli array associativi
  - di supporto:
    - K insieme delle chiavi
    - V insieme dei valori
    - B insieme dei booleani {true, false}
- costanti: array associativo vuoto
- operazioni:
  - put: $A \times K \times V \rightarrow A$ aggiunge una coppia all'array
  - get: $A \times K \rightarrow V$ restituisce il valore associato alla chiave
  - delete: $A \times K \rightarrow A$ rimuove una coppia dall'array
  - exists: $A \times K \rightarrow B$ verifica se una chiave è utilizzata

Le operazioni di inserimento hanno costo costante $\vartheta(1)$ ed un'eventuale ricerca binaria ha costo logaritmico.

Realizzazione

Un array associativo può essere realizzato, in modo efficiente, **attraverso un albero binario di ricerca**, che presenta algoritmi basati su confronti e quindi permette di eseguire le operazioni in tempi logaritmici.
Con uno stesso array è possibile avere più alberi di altezze diverse, ma maggiore è l'altezza maggiore è lo sbilanciamento tra i lati dell'albero.

Può essere, però realizzato, in modo più efficiente, **attraverso le tabelle hash**, che presenta algoritmi non basati su confronti, quindi permette di eseguire le operazioni in tempi lineari e costanti.

Potrebbero anche essere realizzati attraverso liste o array ordinati oppure non ordinati, ma non risulterebbero efficienti, in quanto non garantiscono di eseguire le operazioni di inserimento e ricerca entrambe con bassi costi.

Tabelle Hash

Una tabella hash è una struttura dati usata per mettere in corrispondenza una data chiave con un dato valore e presenta algoritmi non basati su confronti, che permettono di eseguire le operazioni in tempi lineari e costanti.

Realizzazione array associativi

Problemi con le chiavi

Nella realizzazione degli array associativi attraverso le tabelle hash, si presentano due problematiche:

- la *tipologia delle chiavi*, in quanto le chiavi possono essere di qualsiasi tipo, quindi non possono essere usate per indicizzare gli elementi in un array
- la *numerosità delle chiavi*, in quanto le possibili chiavi sono troppe per essere tutte contenute in un array.

È per questi due motivi che si fa affidamento sull'esistenza di un ***insieme universo delle chiavi*** K e il suo sottoinsieme delle chiavi utilizzate a runtime, che non sono note e vengono associate univocamente ai dati dell'insieme m.

Realizzazione

La tabella hash può essere realizzata attraverso un array di dimensione pari al numero di dati associati ($m \ll K$) che deve memorizzare, indicizzato da un numero intero, generato da una funzione.

Funzione hash

La funzione hash genera l'indice della posizione nell'array in cui vanno inseriti i dati a seconda della loro chiave, **trasformando le chiavi in interi** compresi nell'intervallo $[0, m-1]$.

Proprietà

La funzione hash deve avere la proprietà di essere ***deterministica***, cioè di produrre lo stesso intero ogni volta che viene lanciata su una stessa chiave e di essere calcolabile in ***tempi costanti*** (rispetto agli elementi).

Ipotesi di distribuzione uniforme

La funzione hash genera in modo pseudocasuale gli interi, basandosi sull'**ipotesi di distribuzione uniforme**, per cui la ***probabilità che la chiave vengo inserita in una cella è la stessa per ogni cella*** dell'array, indipendentemente dalle altre chiavi già inserite.
Ciò implica che per chiavi simili vengano generati indici diversi.

Problema delle collisioni

La funzione hash ha un codominio m molto più piccolo del dominio K, perciò è inevitabile che si presentino **collisioni**, cioè casi in cui viene generato lo ***stesso intero per diverse chiavi***.
Per ovviare a questo problema si realizzano gli elementi dell'array attraverso delle liste semplicemente concatenate (almeno semplicemente).

Liste di trabocco

Gli elementi dell'array rappresentante la tabella **vengono realizzati attraverso liste di trabocco**, i cui oggetti sono composti da **tre campi**: key valore della chiave, $info$ valore del dato associato e $next$ riferimento al nodo successivo.

Fattore di carico

La dimensione di queste liste non è nota a priori, ma è possibile definire il ***numero medio di elementi*** che contengono, chiamato **fattore di carico**, pari al rapporto tra il numero di elementi memorizzati e il numero di posizioni disponibili $\alpha = n/m$.

Operazioni

Le operazioni sulle tabelle avverranno quindi lavorando sulle liste:

- la ***ricerca di un elemento per chiave*** avverrà ricercando l'elemento con la chiave indicata scorrendo la lista che si trova nella posizione data dalla funzione hash
- l'***inserimento di una nuova coppia*** avverrà inserendo la coppia in testa alla lista che si trova nella posizione data dalla funzione hash oppure sovrascrivendo il valore del dato associato alla chiave, se questa è già in uso

– la ***cancellazione di un elemento*** per chiave avverrà cancellando il nodo dalla lista che si trova nella posizione data dalla funzione hash.

Le funzioni che quindi sono necessarie in queste operazioni sono:
– EQUAL, che confronta due array per verificare se sono uguali, cioè hanno tutte le chiavi uguali
– HASH, che genera l'intero per indicizzare l'array a partire dalle chiavi.

Le operazioni hanno tutte complessità dipendente dal costo di queste due funzioni:
- nel ***caso migliore*** $\boldsymbol{\vartheta(1)} = hash\ \vartheta(1) + equal\ \vartheta(1)$: la lista è vuota o ha un solo elemento
- nel ***caso peggiore*** $\vartheta(n) = hash\ \vartheta(1) + ricerca\ in\ lista\ \vartheta(n) + equal\ \vartheta(n)$: si sono presentate tutte collisioni e gli elementi sono tutti nella stessa lista
- nel ***caso medio*** $\vartheta(n)$ se non è garantita una distribuzione uniforme, mentre ***se è garantita una distribuzione uniforme*** delle chiavi, la lunghezza delle liste coincide con la lunghezza media indicata dal fattore di carico e quindi il costo è $\boldsymbol{\vartheta(\alpha)}$, che se α resta costante sotto una certa soglia, diventa $\vartheta(1)$, in quanto viene effettuata una **gestione telescopica della memoria**: *per mantenere α costante* deve essere incrementato anche m, quindi *viene raddoppiata la dimensione della tabella e* di conseguenza *si dimezza il fattore di carico*. Il costo del raddoppio viene assorbito dagli inserimenti precedenti di costo costante.

Metodi di implementazione

La funzione hash può essere implementata secondo diverse strategie a seconda della tipologia di chiave:
- ***per interi***
  - **metodo della divisione** $h(k) = k \bmod m$
    - se le chiavi sono pseudocasuali, si usa per riportare l'intero nell'intervallo $[0, m-1]$
    - se le chiavi non sono pseudocasuali, si adotta m come numero primo lontano dalle potenze di 2 per limitare le collisioni
  - **metodo della moltiplicazione**
    - se le chiavi appartengono all'intervallo (0,1), $h(k) = \lfloor mk \rfloor$
    - se le chiavi sono qualsiasi, $h(k) = \lfloor m(k\ irr - \lfloor k\ irr \rfloor) \rfloor$ con $irr = \frac{\sqrt{5}-1}{2}$.
- ***per stringhe***

- **funzione** simple_hash esegue la somma dei codici ASCII dei caratteri della stringa ed è *veloce* (complessità lineare), ma *non efficiente*, in quanto restituisce la stessa chiave per permutazioni diverse
- **funzione** DJB2_hash esegue la somma dei codici ASCII dei caratteri della stringa, assumendo come valore iniziale il numero primo $hash = 5381$ e effettuando, prima della somma, una traslazione di 5 celle $hash * 33$

- ***per oggetti*** si usa il metodo della divisione sui singoli campi dell'oggetto e poi se ne esegue la somma.

Insiemi

Un insieme è una collezione di elementi omogenei in cui non c'è valore, ma solo chiave, che coincide quindi con il valore.

Caratteristiche

- domini:
  - di interesse: I insieme degli insiemi
  - di supporto:
    - E insieme degli elementi
    - B insieme dei booleani {true, false}
- costanti: insieme vuoto
- operazioni:
  - add: I×E→I aggiunge un elemento all'insieme
  - remove: I×E→I rimuove un elemento dall'insieme
  - contains: I×E→B verifica se un elemento appartiene all'insieme

Le operazioni di inserimento hanno costo costante $\vartheta(1)$.

Realizzazione

Un insieme può essere realizzato attraverso le tabelle hash (senza il valore).

Può anche essere realizzato, in modo più efficiente con costi logaritmici, attraverso *alberi rosso-neri* al posto delle liste, ma risulta molto più complesso.

Grafi

Un grafo è un ADT costituito da nodi e archi.

Proprietà

Gli elementi di un grafo sono rappresentabili dalle coppie G(V,E) o G(E,V).

Dato un nodo n si dice

- ***arco uscente***, l'arco (n, v), in cui il primo elemento della coppia è il nodo
- ***arco entrante***, l'arco (v, n), in cui il primo elemento della coppia è l'arco
- ***nodo adiacente*** a n, il nodo m per cui esiste un arco uscente da n a m

- ***grado di uscita***, il numero di archi uscenti
- ***grado di entrata***, il numero di archi entranti
- ***nodo sorgente***, se non ha archi entranti
- ***nodo pozzo***, se non ha archi uscenti.

Un **cammino** è una sequenza di nodi per cui esiste un arco per ogni coppia consecutiva di nodi. Il cammino si dice ***semplice*** se tutti i nodi sono distinti. La ***lunghezza*** del cammino è pari al numero di archi che ci sono lungo il percorso.
Un **ciclo** è un cammino non semplice in cui il primo e l'ultimo nodo coincidono. Il ciclo si dice ***semplice*** se solo il primo e l'ultimo nodo coincidono.
È detto cappio o **loop**, il ciclo di un solo arco e un solo nodo.
Il **grafo** si dice **semplice** se non ha cappi e si dice **aciclico** se non ha cicli.

I grafi possono essere orientati o non orientati.

Caratteristiche

- domini:
  - di interesse: G insieme dei grafi
  - di supporto:
    - NI insieme degli iteratori per i nodi
    - AI insieme degli iteratori per gli archi
    - B insieme dei booleani {true, false}
- costanti: grafo vuoto
- operazioni:
  - first_node: G→NI — trova il primo nodo del grafo
  - next_node: G×NI→NI — trova il prossimo nodo
  - add_node: G→NI — aggiunge un nodo al grafo
  - first_edge: G→AI — trova il primo arco del grafo
  - next_edge: G×AI→AI — trova il prossimo arco
  - add_edge: G→AI — aggiunge un arco al grafo
  - adj_node: G×NI×AI→NI — trova il nodo adiacente tramite l'arco
  - are_adj: G×NI×NI→B — verifica se due nodi sono adiacenti

Visite dei grafi

Le visite dei grafi sono svolte attraverso algoritmi che permettono di visitare tutti i nodi raggiungibili a partire da un nodo di partenza. I nodi vengono raggiunti secondo un ordine dato dalla forma del grafo, ma non tutti i nodi vengono raggiunti, mentre altri vengono raggiunti più volte.

Per questi motivi tali algoritmi fanno uso di marcatori.

Marcatori

I marcatori sono valori associati ai nodi, generalmente interi con il nome di ***colore***.

La **marcatura** dei grafi avviene

- se il grafo è identificato da un intero (liste), aggiungendo al grafo un array di interi di dimensione pari al numero di nodi
- se il grafo è identificato da un riferimento, aggiungendo all'oggetto nodo il campo intero color.

Breadth-First Search

La visita in ampiezza BFS viene svolta a partire da un nodo, visitando tutti i nodi raggiungibili nell'ordine imposto dalla distanza, quindi prima quelli più vicini, poi quelli più lontani.

La strategia prevede l'utilizzo di una **coda**, inizializzata con il nodo di partenza e finché questa non risulta vuota, si estrae l'elemento e si analizzano i nodi adiacenti: se il nodo è marcato si salta, altrimenti viene marcato e messo in coda.

La complessità dell'algoritmo è nel ***caso peggiore*** $\boldsymbol{\vartheta(n+m)}$, in quanto ogni nodo viene aggiunto e rimosso una sola volta e ogni arco viene considerato due volte dal nodo sorgente e destinazione.

Depth-First Search

La visita in profondità DFS viene svolta a partire da un nodo, percorrendo il grafo finché si trovano nodi non visitati.

La strategia prevede la visita di un nodo adiacente al nodo, continuando ricorsivamente finché non si incontra un nodo già visitato e quindi si torna indietro per cercare nodi non visitati fino a tornare al nodo di partenza.

Analizzando gli archi del grafo è possibile generare un albero, in cui non si troverà mai un collegamento tra un nodo della sezione sinistra e uno della sezione destra.

La complessità dell'algoritmo è nel ***caso peggiore*** $\boldsymbol{\vartheta(n+m)}$, in quanto su ogni nodo viene lanciata la visita una sola volta e ogni arco viene considerato due volte dal nodo sorgente e destinazione.

Grafi orientati

Un grafo orientato o diretto è un grafo in cui tutti gli archi sono diretti e ogni elemento è rappresentato da una relazione binaria.

Realizzazione

Il grafo orientato può essere realizzato attraverso due strategie:

- la **matrice delle adiacenze**, le cui celle (i, j) rappresentano gli archi tra i nodi i e j e hanno valore 1 se l'arco esiste, valore 0 se non esiste
  Le operazioni hanno costo quadratico $\vartheta(n^2)$.
- la **lista delle adiacenze**, per cui un array rappresenta i nodi attraverso la posizione e ogni suo elemento è una lista dei nodi a cui esso è collegato.
  Le operazioni hanno costo $\mathcal{O}(n) + \mathcal{O}(m)$, nel caso peggiore, se il grafo è denso $m \in \mathcal{O}(n^2)$, altrimenti $m \in \mathcal{O}(n)$.

Grafi non orientati

Un grafo non orientato o non diretto è un grafo in cui nessun arco è diretto e ogni elemento è rappresentato da una relazione a doppio senso.

Realizzazione

Il grafo non orientato può essere realizzato attraverso due strategie:

- la matrice delle adiacenze, come per i grafi orientati, ma è simmetrica
- la lista delle adiacenze, come per i grafi orientati, ma ogni nodo che in lista ha il suo adiacente, si trova anche nella lista di questo.

Le operazioni hanno costi lineari.

Proprietà

Un grafo non orientato si dice ***connesso*** se per ogni coppia di nodi esiste un cammino da uno all'altro. Un nodo si dice ***raggiungibile*** da un altro nodo se esiste un cammino da uno all'altro. La proprietà di raggiungibilità è una proprietà di equivalenza, le cui classi di equivalenza sono chiamate ***componenti connesse***.

Procedure

Per i grafi orientati si possono implementare ulteriori procedure:

- ***verifica connettività***: a seguito di una visita BFS, se alcuni nodi non sono marcati si può dire che il grafo non è connesso
- ***visita in BFS o DFS***: a seguito di una prima visita, si rilancia la stessa con lo stesso array di marcatori su un diverso nodo

- ***calcolo componenti connesse***: si cicla finché esiste un nodo non visitato, incrementando un contatore e lanciando su tale nodo una visita BFS.

Grafi pesati

Un grafo pesato è un grafo in cui ad ogni arco è associato un peso.

Realizzazione

Il grafo pesato può essere realizzato attraverso due strategie:

- la matrice delle adiacenze, come per i grafi non pesati, ma si usa il peso dell'arco al posto del valore 1
- la lista delle adiacenze, come per i grafi non pesati, ma ogni elemento della lista ha anche il campo $weight$ (oltre a $prev$ key e $next$).

Grafi tramite oggetti e riferimenti

I grafi possono essere realizzati tramite oggetti e riferimenti, così da evitare la riallocazione della memoria, che le altre due strategie obbligano quando si effettuano le operazioni di inserimento e cancellazione.

Il grafo viene rappresentato da **un oggetto**, con **due** campi, entrambi **liste doppiamente concatenate** a rappresentare i nodi e gli archi presenti nel grafo.
Ogni elemento di queste liste ha i campi $\boldsymbol{prev}$ riferimento al precedente e $\boldsymbol{next}$ riferimento al successivo, ma anche un campo che a sua volta è un **oggetto**, a rappresentare il nodo o l'arco, nelle diverse liste.
L'oggetto **nodo** ha i campi id $label$ e $\boldsymbol{archi}$, una lista degli archi entranti e uscenti del nodo.
L'oggetto **arco** ha i campi id $weight$, $\boldsymbol{from}$ riferimento al nodo sorgente e $\boldsymbol{to}$ riferimento al nodo destinazione.
Questa rappresentazione è valida per tutti i tipi di grafi, solo che nei grafi non orientati non si fa differenza tra il nodo di origine e di destinazione, quindi i campi *from* e *to* sono intercambiabili.

Operazioni

Le operazioni sui grafi avverranno quindi lavorando sulle liste:

1. ***inserimenti***

- l'inserimento di un ***nuovo nodo*** avverrà creando l'oggetto nodo e inserendolo in testa alla lista nodi del grafo
- l'inserimento di un ***nuovo arco*** avverrà creando l'oggetto arco e inserendolo in testa alle liste archi del grafo e dei nodi sorgente e destinazione

2. ***cancellazioni***

- la rimozione di un ***arco***, attraverso il suo identificatore, avverrà rimuovendo l'arco dalle liste archi dei nodi sorgente e destinazione, poi dalla lista archi del grafo
- la rimozione di un ***nodo***, attraverso il suo identificatore, avverrà rimuovendo prima tutti gli archi adiacenti poi rimuovendo il nodo dalla lista nodi del grafo.

Queste operazioni avvengono in tempi quadratici, dunque non sono efficienti.

Per ridurre i costi si aggiungono agli oggetti degli **ulteriori campi**:

- all'oggetto **nodo** si aggiunge il campo ***pos*** riferimento all'elemento della lista dei nodi del grafo
- all'oggetto **arco** si aggiungono i campi ***pos*** riferimento all'elemento della lista archi del grafo, ***frompos*** riferimento all'elemento nella lista archi del nodo sorgente, ***topos*** riferimento all'elemento nella lista archi del nodo destinazione.

Attributi opzionali del grafo

La struttura del grafo può essere arricchita introducendo **altri campi** al fine di semplificare alcune operazioni:

- *numero_nodi* per non dover scorrere la lista nodi del grafo per contarli
- *numero_archi* per non dover scorrere la lista archi del grafo per contarli
- max *_id_nodi* per non dover verificare se l'id è già in uso per un nodo
- max *_id_archi* per non dover verificare se l'id è già in uso per un arco.

Sommario

ALGORITMI 1
Problemi e Algoritmi 1
Problema 1
Algoritmo 1
Random Access Machine 1
Algoritmi di ordinamento 1
Proprietà 1
Operazione in loco 1
Stabilità 1
Incrementali 2
Strategia 2
Tecnica Greedy 2
Divide et impera 2
Algoritmi noti 3
Algoritmo Selection Sort 3
Algoritmo Insertion Sort 3
Algoritmo Merge Sort 3
Algoritmo Quick Sort 3
Algoritmo Heap Sort 4
Algoritmo Tree Sort 4
Tabella di confronto 5
STRUTTURE DATI 6
Strutture di dati 6
Abstract Data Type 6
Realizzazione 6
Stack 6
Caratteristiche 6
Realizzazione 7
Queue 7
Caratteristiche 7
Realizzazione 7
Liste 7
Caratteristiche 7
Realizzazione 8
Liste tramite array 8
Liste con tre array 8

Liste con un array 9
Liste tramite oggetti e riferimenti 9
Singly Linked List 9
Doubly Linked List 9
Alberi radicati 10
Definizione 10
Proprietà 10
Relazione di discendenza 10
Esempi di applicazione 11
Visite di un albero 11
Cammino euleriano 11
Alberi di grado arbitrario 12
Caratteristiche 12
Realizzazione 12
Alberi binari 13
Proprietà 13
Caratteristiche 13
Realizzazione 14
Alberi binari di ricerca 14
Operazioni 14
Alberi rosso-neri 15
Definizione 15
Proprietà 15
Operazioni 15
Rotazioni 16
Ripristino delle proprietà 16
Heap 17
Proprietà 17
Realizzazione 17
Procedure 17
Code di priorità 18
Esempi di applicazione 18
Caratteristiche 18
Realizzazione 18
Operazioni 19
Array associativi 19
Esempi di applicazione 19

Caratteristiche 19
Realizzazione 20
Tabelle Hash 20
Realizzazione array associativi 20
Problemi con le chiavi 20
Realizzazione 20
Funzione hash 20
Proprietà 21
Ipotesi di distribuzione uniforme 21
Problema delle collisioni 21
Metodi di implementazione 22
Insiemi 23
Caratteristiche 23
Realizzazione 23
Grafi 23
Proprietà 23
Caratteristiche 24
Visite dei grafi 25
Marcatori 25
Breadth-First Search 25
Depth-First Search 25
Grafi orientati 26
Realizzazione 26
Grafi non orientati 26
Realizzazione 26
Proprietà 26
Procedure 26
Grafi pesati 27
Realizzazione 27
Grafi tramite oggetti e riferimenti 27
Operazioni 27
Attributi opzionali del grafo 28