

Introduzione

Una struttura dati è un'informazione organizzata allo scopo di migliorare l'efficienza dell'algoritmo di un programma. Un tipo astratto di dati (TDA) definisce con rigore matematico una struttura dati.

Il TDA Stack (pila)

Lo stack è un TDA che immagazzina oggetti arbitrari. Inserimenti e cancellamenti avvengono secondo lo schema LIFO (last in, first out), quindi un nuovo oggetto viene inserito in cima (top) allo stack e l'elemento cancellato è sempre quello che si trova in cima.

Le operazioni supportate dal TDA Stack sono:

- `push(x)`: inserisce l'elemento `x` nello stack
- `pop()`: cancella e restituisce in output l'ultimo elemento inserito
 - Se `pop()` viene invocato su uno stack vuoto genera un errore
- `top()`: restituisce l'ultimo elemento **senza** cancellarlo
 - Se `top()` viene invocato su uno stack vuoto genera un errore
- `size()`: restituisce il numero di elementi dello stack
- `isEmpty()`: restituisce TRUE/FALSE a seconda che lo stack sia vuoto o meno

Un esempio di TDA Stack è la cronologia delle pagine web di un browser o un registro chiamate.

Stack implementati con array

Gli elementi vengono aggiunti da sinistra verso destra mentre una variabile tiene traccia dell'indice dell'elemento al top. [Uno stack di `n` elementi richiede spazio $O(n)$]

Esistono certi limiti nell'implementazione con array (non del TDA), ovvero:

- La dimensione massima dello stack dev'essere stabilita a priori e non può essere cambiata;
- L'array può riempirsi totalmente;
- Un'operazione di push su uno stack pieno provoca (throw) un `FullStackException`;

NOTA SULLE ECCEZIONI:

Tutte le eccezioni usate nelle implementazioni dei TDA sono definite come sottoclassi della classe

RuntimeException, inoltre le eccezioni che non sono sottoclassi di **RuntimeException** devono essere dichiarate nella clausola **throws**.

Vantaggi: Implementazione semplice ed efficiente

- Memoria usata: $O(N)$
- Complessità dei metodi (`push`, `pop`, `top`, `size`, `isEmpty`): $O(1)$

Svantaggi: bisogna fissare *a priori* un limite superiore `N` alla taglia massima dello stack

Dunque si possono verificare due casi: O un'applicazione potrebbe richiedere una capacità superiore ad `N`, in tal caso lo stack lancia

una `FullStackException` che arresta l'applicazione; Oppure un'applicazione occupa una capacità *molto inferiore* ad `N` ci sarà uno spreco di memoria.

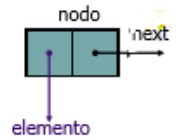
Stack basati su array dinamici

Quando l'array è pieno, `push()` rimpiazza l'array con uno più grande. La decisione su quanto più grande dev'essere l'array dipende da due tipi di strategie:

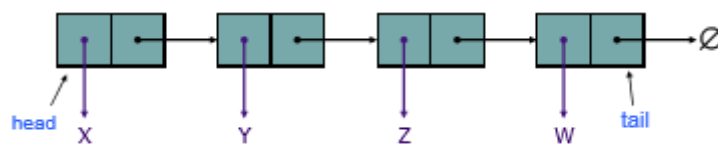
- Strategia incrementale: aumenta la dimensione di una costante c ;
- Strategia del raddoppio: raddoppia la dimensione

Implementazione di Stack con lista a puntatori singoli

Una lista a puntatori singoli consiste in una sequenza di nodi. Ciascun nodo contiene un riferimento all'elemento corrente ed un riferimento al prossimo nodo (next link)



I campi next determinano un ordinamento lineare sui nodi mentre il primo e l'ultimo nodo sono detti rispettivamente *head* e *tail* [testa e coda]



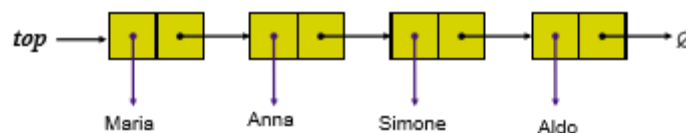
L'elemento al top è salvato nel primo nodo della lista. Per uno stack di n elementi si usa uno spazio di dimensioni $O(n)$, mentre ciascuna operazione del TDA Stack richiede tempo $O(1)$.

Per l'inserimento di un elemento E in una lista concatenata, si crea un nuovo nodo e si definisce il riferimento all'elemento E , si definisce poi il riferimento al nodo successivo (il vecchio head) e si aggiorna il riferimento al nodo in cima alla pila.

La somiglianza con l'array sta nell'ordinamento lineare, mentre la differenza sta nel fatto che la lista concatenata non ha bisogno di una taglia fissata a priori come accade per l'array (può usare uno spazio proporzionale al numero dei suoi elementi)

Vantaggi: non si deve specificare a priori un limite superiore alla taglia della pila o della coda (una pila o una coda di N elementi richiederà sempre uno spazio di dimensione $O(N)$); tutte le operazioni richiedono tempo $O(1)$.

Svantaggi: lo spazio di memoria richiesto da ciascun elemento è leggermente superiore rispetto alla soluzione con array, così come l'implementazione è leggermente meno semplice di quella tramite array



Queue

Il TDA Code (Queue) è un TDA in cui inserimento e cancellazione avvengono secondo una strategia FIFO (First-In, First-Out), dunque durante un **inserimento** un nuovo oggetto viene inserito dietro la coda, in caso di **cancellazione** viene cancellato l'elemento davanti alla coda mentre per una **ricerca** viene restituito il prossimo elemento che sarebbe cancellato. Le operazioni supportate dal TDA Coda sono:

- Enqueue(element): inserisce un nuovo elemento dietro (rear) la coda
- Dequeue(): restituisce l'elemento in testa (front) alla coda (quello che è restato più a lungo in coda) e lo rimuove
- Front() restituisce l'elemento in testa alla coda
- Size() restituisce il numero di elementi in coda
- isEmpty() restituisce TRUE se la coda è vuota, FALSE altrimenti

Size e Front possono generare un **EmptyQueueException** se invocati su una coda vuota

Implementazione coda

L'implementazione della coda avviene tramite array, in cui l'elemento in testa alla coda è il primo elemento di quest'ultimo; si inseriscono gli oggetti da sinistra a destra e ci sono due variabili F,R (Front, Rear) che tengono traccia degli elementi presenti all'interno della coda: F indica il primo elemento presente nell'array, mentre R indica la prossima cella libera; nel caso di $F=R$ allora si dice che la coda sia vuota.

Ad ogni inserimento viene incrementata r, mentre ad ogni cancellazione viene incrementata f. Nel caso in cui ci siano chiamate continue di enqueue() e dequeue(), si avranno le variabili F,R sulla fine dell'array e un ulteriore inserimento verrebbe negato nonostante l'array abbia comunque dei posti liberi. Per ovviare a questo problema, si può far in modo che le variabili F,R vengano incrementate "circolarmente", ovvero $F=(F+1) \bmod N$.

Si pone a $N-1$ il limite massimo di oggetti che la coda può contenere, perché se ci fossero N enqueue() consecutivi si arriva ad avere $F=R$, ovvero la condizione in cui la coda è vuota, generando quindi un errore.

Vantaggi: implementazione semplice ed efficiente

- memoria usata: $O(N)$
- complessità dei metodi (size, isEmpty, front, enqueue, dequeue): $O(1)$

Svantaggi: bisogna fissare *a priori* un limite superiore N alla massima taglia della coda

Deque

Il TDA *Deque* è un TDA simile alla *Coda/Queue* che però supporta cancellazioni ed inserimenti ad entrambe le estremità. Le operazioni supportate sono:

- `AddFirst(e)` inserisce “e” all’inizio del deque
- `addLast(e)` inserisce “e” alla fine del deque
- `removeFirst()` restituisce e rimuove il primo elemento del deque
- `removeLast()` restituisce e rimuove l’ultimo elemento del deque
- `getFirst()` restituisce il primo elemento del deque
- `getLast()` restituisce l’ultimo elemento del deque
- `size()` restituisce la size del deque
- `isEmpty()` restituisce true se il deque è vuoto, false altrimenti

(richiede la definizione di `EmptyDequeException`)

Un’implementazione di Deque basata sulle liste

- Si usano liste doppiamente linkate
 - Ciascun nodo contiene un riferimento (`next`) al nodo successivo e un riferimento (`prev`) al nodo precedente



- **Vantaggio:** cancellazione in tempo costante → `removeLast()` in tempo costante
 - Nelle liste a link singoli la cancellazione di un nodo X richiede tempo proporzionale al numero di nodi che precedono X
- Per semplificare la scrittura del codice si usano due nodi “sentinella”
 - Header:
 - Il campo `next` contiene un riferimento al **primo nodo** della deque
 - Il campo `prev` è settato a `NULL`
 - Trailer:
 - Il campo `prev` contiene un riferimento **all’ultimo nodo** della deque
 - Il campo `next` è settato a `NULL`

Metodi dei nodi

Un nodo di una lista doppiamente linkata deve supportare i seguenti metodi:

- `setElement(E elem)`
- `setNext(DLNode<E> newNext)`
- `setPrev(DLNode<E> newPrev)`
- `getElement()`
- `getNext()`
- `getPrev()`

Design Pattern

- Un design pattern fornisce un metodo generale per risolvere situazioni differenti
- Un design pattern descrive
 - Gli scenari in cui può essere applicato
 - Il modo in cui va applicato
 - Il risultato della sua applicazione

Adapter

- Metodologia di programmazione che consiste nell'adattare le funzionalità di una classe preesistente a quelle di una nuova classe che si intende costruire
 - In genere la nuova classe contiene un'istanza della vecchia come variabile di istanza nascosta
 - I metodi della nuova classe sono implementati usando i metodi della vecchia invocati per variabile di istanza nascosta

Esempi dell'uso dell'Adapter

- Specializzare una classe per implementare una classe più semplice
 - ESEMPIO: adattiamo la classe che implementa Deque in modo che possa essere usata per implementare Stack
- Specializzare una classe affinché funzioni con un certo tipo di dati
 - ESEMPIO: adattiamo la classe ArrayStack in modo che lo Stack contenga solo stringhe

Stack e Code implementate mediante Deque

	Top() implementato con getLast()
STACK	Push() implementato con addLast()
	Pop() implementato con removeLast()
<hr/>	
	Front() implementato con getFirst()
CODA	Enqueue() implementato con addLast()
	Dequeue() implementato con removeFirst()

ArrayList (vettore)

Queue, Stack e Deque possono essere visti come sequenze particolari in cui possiamo inserire, cancellare e avere accesso solo ad elementi particolari (primo/ultimo)

L'ArrayList è un contenitore di elementi organizzati secondo un ordinamento lineare, in cui cancellazioni e inserimenti sono possibili in posti arbitrari. L'accesso agli elementi avviene per mezzo del loro indice, e l'indice di un elemento in un ArrayList è il numero di elementi che lo precedono; infatti l'elemento i -esimo ha indice $i-1$.

Naturalmente l'indice di un elemento si può modificare in seguito a cancellazioni e inserimenti

Operazioni su ArrayList

- `Get(i)` restituisce l'elemento di indice " i " di V , senza rimuoverlo
- `Set(i, e)` restituisce e rimpiazza in V l'elemento di indice " i " con l'elemento " e "
- `Add(i, e)` inserisce in V l'elemento " e " con indice " i "
- `Remove(i)` restituisce e rimuove da V l'elemento di indice " i "
- `Size()` restituisce il numero di elementi presenti nell'ArrayList
- `isEmpty()` restituisce TRUE se l'ArrayList è vuoto, FALSE altrimenti

ADD E REMOVE (In pratica vengono "shiftati" gli elementi presenti nell'arraylist)

- `Add(i,e)`: viene incrementato di 1 l'indice di tutti gli elementi che prima dell'inserimento avevano indice $\geq i$
- `Remove(i)`: viene decrementato di 1 l'indice di tutti gli elementi prima della rimozione che avevano indice $> i$

L'arraylist viene implementato con un array A di N elementi e una variabile $n < N$ che indica il numero di elementi nell'array list. `Remove(i)` e `add(i)` hanno tempo di esecuzione $\Theta(n)$ nel caso pessimo, ovvero quando $i=0$

Quando l'operazione `add(i,e)` dà luogo ad un overflow ($n=N$), anziché lanciare l'eccezione, rimpiazzando l'array con uno più grande:

1. Allochiamo un nuovo array B di capacità $2N$;
2. Copiamo $A[i]$ in $B[i]$ per ogni $i=0\dots N-1$;
3. Poniamo $A=B$;
4. Inseriamo il nuovo elemento e in A

Richiede l'eccezione `IndexOutOfBoundsException` che viene lanciata quando si specifica come argomento di un metodo un indice sbagliato

Array estensibili: tecnica del raddoppio

- Ogni volta che l'array si riempie, lo sostituiamo con un altro grande il doppio, dunque se il numero di elementi n è uguale alla lunghezza dell'array allora l'operazione di ADD richiederà tempo $O(n)$ per trasferire gli elementi nel nuovo array.
- N OPERAZIONI DI ADD (effettuate a partire da un vettore vuoto) RICHIEDONO $O(n)$ PER TRASFERIRE GLI ELEMENTI
- In generale, per N operazioni ADD pago $3 \cdot n\$ \rightarrow$ tempo di esecuzione delle N operazioni di ADD = $O(N)$
- SPESE DI EVENTUALI SHIFT ESCLUE

Array estensibili

Nell'operazione `REMOVE(i)` se il numero di elementi è sceso al di sotto di una certa soglia, allora l'array può essere rimpiazzato da uno più piccolo. Così si evitano sprechi di spazio, inoltre una buona scelta è quella di dimezzare la dimensione dell'array quando questo è riempito per meno di un quarto della sua lunghezza.

Node List (lista)

Analogamente al TDA ArrayList, il TDA NodeList rappresenta una sequenza di elementi disposti secondo un ordine lineare, è la versione orientata agli oggetti della struttura dati concreta lista a puntatori.

La versione astratta del nodo è il TDA **Position** [Position nasconde il modo in cui la lista è implementata]

La lista di posizioni (position list) è un contenitore di oggetti che memorizza ciascun elemento in una posizione, e mantiene le stesse degli oggetti in un ordinamento lineare; il concetto di posizione formalizza l'idea di **posto** di un elemento

TDA Position

La lista di posizioni (positionList) è un contenitore di oggetti che memorizza ciascun elemento in una posizione, e mantiene quest'ultime in un ordinamento lineare. Il concetto di posizione formalizza l'idea di **posto** di un elemento. *[la posizione non è altro che il nome che permette di riferirsi all'elemento ad esso associato]*

La posizione è associata sempre allo stesso elemento e non cambia mai, neanche in seguito a cancellazioni o inserimenti.

La posizione, inoltre, definisce essa stessa un tipo astratto di dati che supporta l'unico metodo element(), che restituisce l'elemento nella posizione corrente.

Metodi del TDA Node List

- Metodi generici
 - Size(): restituisce il numero di elementi memorizzati nella lista
 - isEmpty(): restituisce TRUE solo se la lista è vuota
- Metodi di accesso
 - First(): restituisce la posizione del primo elemento della lista. Se è vuota, si verifica un errore.
 - Last(): restituisce la posizione dell'ultimo elemento della lista. Se è vuota, si verifica un errore.
 - Prev(p): restituisce la posiz. Dell'elemento che si trova PRIMA dell'elemento P.
Se P è nella prima posizione si verifica un errore.
 - Next(p): restituisce la posiz. dell'elemento che si trova DOPO l'elemento P.
Se P è nell'ultima posizione si verifica un errore.
- Metodi di aggiornamento
 - addBefore(p,e): inserisce l'elemento E nella posizione prima di quella di P e ne restituisce la posizione
 - addAfter(p,e): inserisce l'elem. E nella posizione dopo di quella di P e ne restituisce la posizione
 - addFirst(e): inserisce l'elemento E nella prima posizione della lista
 - addLast(e): inserisce E nell'ultima posizione della lista
 - remove(p): rimuove e sostituisce l'elemento in posizione P
 - set(p, e): sostituisce con E l'elemento in posizione P restituendolo in output

Le eccezioni in cui può incorrere questo TDA sono:

- InvalidPositionException
 - Viene lanciata quando viene specificata una posizione non valida come argomento
 - Ad esempio se P=NULL, P è la posizione di una lista differente o P è già eliminato dalla lista
- BoundaryViolationException
 - Viene lanciata quando si tenta di accedere ad una posizione fuori della lista
 - Ad esempio viene lanciata dal metodo PREV quando riceve come argomento la prima posizione della lista.
- EmptyListException
 - Quando i metodi FIRST() e LAST() vengono invocati su una lista vuota.
(NB sono metodi della classe NodePositionList)

Per l'implementazione basata sulle liste doppiamente concatenate, i nodi implementano le posizioni del TDA NodeList, quindi implementano l'interfaccia Position

TDA Sequence (solo per gli ex studenti del corso De Bonis)

È un TDA per rappresentare un insieme di elementi disposti secondo un ordine lineare; fornisce le funzionalità del TDA NodeList e del TDA ArrayList. È possibile far riferimento ad un elemento sia attraverso il suo indice che attraverso la sua posizione.

Oltre ai metodi del TDA ArrayList e NodeList, il TDA Sequence supporta i seguenti metodi:

- `getFirst`: restituisce l'elemento che si trova nella prima posizione
- `getLast`: restituisce l'elemento che si trova nell'ultima posizione
- `removeFirst`: cancella il primo elemento della sequenza restituendolo in output
- `removeLast`: cancella l'ultimo elemento della sequenza restituendolo in output

Esistono inoltre due metodi "ponte" che mettono in relazione indici e posizioni:

- `atIndex(i)`: restituisce la posizione dell'elemento di indice i
- `indexOf(p)`: restituisce l'indice dell'elemento in posizione P

Implementazioni di Sequence

- Lista doppiamente linkata: un nodo contiene un elemento della sequenza
- Array di posizioni
 - Il TDA Position viene implementato in modo da memorizzare, oltre all'elemento anche il suo indice
 - Se l'array è usato in modo circolare, allora i metodi `ADDFIRST` e `REMOVEFIRST` possono essere eseguiti in tempo $O(1)$

Implementazione basata su array

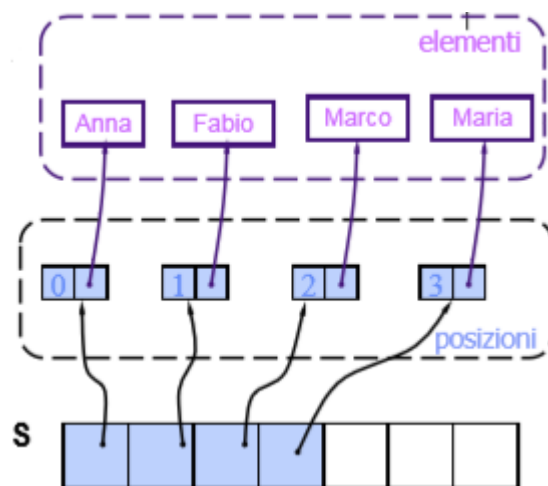
Ogni locazione contiene un oggetto di tipo Position

Un oggetto di tipo Position memorizza:

- **Elemento**
- **Indice**

Implementazione basata su array circolare

(stessa immagine a fianco, solo che la prima cella è di un indice F mentre la prima libera a destra è indicata da una certa R)



Iterator / Iteratore

Un iteratore permette la scansione di un qualsiasi TDA che astrae una collezione ordinata di elementi. Può essere visto infatti come un'estensione del concetto di posizione (position): il TDA Position incapsula il concetto di "posto", mentre un iteratore incapsula il concetto di "posto" e di "posto successivo".

Un iteratore dunque può essere definito come un TDA che supporta i due seguenti metodi:

- `hasNext()`: determina se ci sono altri elementi nell'iteratore
- `next()`: restituisce l'elemento successivo in un iteratore

Supporta l'eccezione **NoSuchElementException** quando `next()` viene invocato su una collezione vuota o sull'ultimo elemento della collezione.

Java fornisce un iteratore con l'interfaccia **java.util.iterator**. Per quanto riguarda la sua implementazione:

- un iteratore viene solitamente associato ad uno dei TDA ad accesso sequenziale (arraylist, nodelist)
- un iteratore permette di accedere a tutti gli oggetti della collezione nell'ordine lineare stabilito
- il primo elemento di un iteratore si ottiene dalla prima chiamata del metodo `next()`, assumendo che l'iteratore contenga almeno un elemento.

Bisogna fare in modo che tutti i TDA Ad accesso sequenziali supportino il metodo `Iterator<E> iterator()`, che restituisce un iteratore degli elementi della collezione.

Il tipo Iterable

Se vogliamo che il TDA `NodeList` supporti il metodo `iterator()` dobbiamo ridefinire l'interfaccia `PositionList`, quindi facciamo in modo che l'interfaccia `positionList` estendi l'interfaccia `iterable`).

Se non ci fossero gli iteratori si cicla sulla prima posizione con una variabile `Position` finché non si raggiunge l'ultimo nodo.

Il metodo positions()

Nei TDA che supportano la nozione di `Position` possiamo aggiungere il metodo **`Iterable <Position<E>> positions()`** che restituisce un oggetto di tipo `Iterable` (collezione iterabile) contenente le posizioni dei TDA come elementi

Se invochiamo **`ITERATOR()`** sulla collezione restituita da **`POSITIONS()`** otteniamo un iteratore delle posizioni del TDA

Se vogliamo che il TDA `NodeList` supporti il metodo `POSITIONS()` dobbiamo ridefinire l'interfaccia `PositionList` facendo un **`extends Iterable<E>`** e aggiungendo il metodo **`public Iterable <Position <E>> positions();`**

Implementare un iteratore mediante una struttura dati

- Possiamo utilizzare una qualsiasi struttura dati che consente l'accesso sequenziale ai suoi elementi:
 - In genere si utilizza una lista linkata o un array
- Per creare un iteratore bisogna copiare tutti gli elementi della struttura dati che si vuole scandire in quella usata per implementare l'iteratore
 - `Iterator()` richiede tempo $O(n)$, dove N è il numero di elementi.

Implementazione mediante lista linkata

Si copiano gli elementi in una lista linkata

- Qui usiamo una lista a puntatori singoli
- La classe che implementa `Iterator` in questo caso si chiama `LinkedListIterator`

Input <a,b,c>

Lista: HEAD→[a,-]→[b,-]→[c,]

Implementazione dell'iteratore mediante un cursore

In questa implementazione, l'iteratore opera sulla collezione che si vuole scandire invece che su una sua copia. Si usano una variabile che fa riferimento alla collezione di oggetti, ed una variabile di istanza che tiene traccia dell'elemento corrente (cursore); Per creare un iteratore bisogna semplicemente inizializzare le due variabili di istanza [iterator() richiede tempo $O(1)$]

È un iteratore *semi-dinamico* in quanto tiene conto di alcune modifiche che vengono effettuate sulla lista

Il metodo Iterator() di List

Usando l'implementazione precedente di ElementIterator è semplicissimo scrivere il metodo iterator() di una classe che implementa PositionList

Implementazione mediante struttura dati VS implementazione con cursore

- Implementazione mediante struttura dati
 - Vantaggio: le classi che implementano Iterator in questo modo permettono di creare iteratori per una qualsiasi struttura dati
 - Svantaggio: la creazione di un iteratore richiede tempo lineare nel numero di elementi della struttura dati da scandire
- Implementazione mediante cursore
 - Vantaggio: la creazione di un iteratore richiede tempo costante
 - Svantaggio: per ciascuna struttura dati che vogliamo scandire, dobbiamo fornire un'apposita classe che implementa iterator per quella struttura dati

Priority Queue - Definizione informale

Una coda a priorità è un contenitore di elementi a ciascuno dei quali è assegnata una chiave

- Questa chiave viene assegnata nel momento in cui l'elemento è inserito nella coda
- Le chiavi determinano la priorità degli elementi, ovvero l'ordine in cui vengono rimossi dalla coda

Priority scheduling

Ad ogni processo viene associata una priorità. I processi in attesa di essere eseguiti sono inseriti in una coda a priorità, dalla quale viene estratto ed eseguito il processo con la priorità più grande

- Problema: STARVATION – i processi con priorità più bassa non vengono mai eseguiti
- Soluzione: AGING: le priorità dei processi in coda vengono gradualmente aumentate

Il TDA PriorityQueue

È un contenitore di oggetti, ognuno dei quali è una coppia (key, **element**) → Entry. La chiave specifica la priorità dell'oggetto all'interno della collezione. Generalmente, viene assegnata ad un elemento nel momento in cui l'elemento viene inserito e può essere successivamente modificata. Ad ogni istante può essere rimosso solo l'elemento con la più alta priorità, e per poter stabilire qual è l'oggetto con la più alta priorità è necessario definire una **regola per confrontare le chiavi**, ovvero una Relazione d'ordine.

Una relazione d'ordine è una relazione binaria \leq su un insieme X che è riflessiva, antisimmetrica e transitiva. Una relazione d'ordine \leq si dice totale se $k \leq k'$ oppure $k' \leq k$ per ciascuna coppia di elementi k, k' in X. **Se su una collezione è definita una relazione d'ordine totale, allora è ben definito il più piccolo valore della collezione (chiave più piccola).** Il TDA della coda a priorità è definito attraverso i metodi principali (su una priority queue P):

- Insert(k,o):
 - Inserisce l'entrata (k,o) in P restituendola in output; se K non è una chiave valida si verifica un errore.
- removeMin(): rimuove e restituisce l'entrata con chiave (key) più piccola; se la coda è vuota si ha un errore.

Metodi aggiuntivi:

- min(P): restituisce, senza rimuoverla, l'entrata con chiave più piccola della coda a priorità; se la coda è vuota, si genera un errore
- size(P), isEmpty(P);

Le eccezioni sono:

- **EmptyPriorityQueueException:** lanciato se il metodo min() o removeMin() viene chiamato su una coda a priorità vuota
- **InvalidKeyException:** lanciato se il metodo insert(k,o) viene chiamato con una chiave k non valida che rende impossibile il confronto con il Comparator della coda a priorità (k null oppure k è di una classe incompatibile con le classi delle altre chiavi)

Nel caso in cui si implementasse la coda a priorità con una lista doppiamente concatenata non ordinata, si avrebbe un inserimento veloce ed una cancellazione lenta

- Insert(k,o) richiede tempo $O(1)$ perché possiamo semplicemente creare un nuovo oggetto entry $e=(k,o)$ e usare insertLast(e)
- Min() e removeMin() richiedono tempo $O(n)$ perché richiedono lo scorrimento della lista (non ordinata) per determinare l'elemento con chiave minima

Con una lista ordinata, invece, si avrebbe una cancellazione veloce ($O(1)$ perché la chiave minima è all'inizio della lista) e un inserimento lento perché bisogna inserire il nuovo oggetto nel posto giusto per mantenere l'ordine non decrescente delle chiavi

Per tener traccia dell'associazione valore-chiave, si usa il **composition pattern**: ovvero che un singolo oggetto (entry) è definito come una composizione di altri oggetti.

Per confrontare le chiavi in modo da determinare la più piccola, invece, si adopera il **comparator pattern**: invece di basarci sulle chiavi, per stabilire la regola di confronto definiamo un oggetto comparator, esterno alle chiavi. Ogni volta che viene costruita una nuova coda a priorità, viene fornito un oggetto Comparator; il vantaggio è che si può dare un'implementazione generale di coda a priorità che funziona in vari contesti.

Il TDA Comparator fornisce un meccanismo di confronto basato sull'unico metodo compare(a,b), che restituisce un intero i tale che $i < 0$ se $a < b$; $i = 0$ se $a = b$, $i > 0$ se $a > b$.

Si verifica una condizione di errore se a e b non possono essere confrontati.

L'interfaccia che definisce Comparator è java.util.comparator, che include anche il metodo equals(), per confrontare un comparatore con un altro comparatore

TDA Tree

Un albero è una struttura dati che contiene oggetti organizzati **gerarchicamente** (a differenza di nodelist e arraylist, i cui elementi sono organizzati linearmente). Ciascun elemento dell'albero (tranne la radice) ha un unico padre e zero o più figli.

I nodi che non hanno figli sono chiamati **foglie**, se sono figli dello stesso padre sono detti **fratelli**. I nodi che hanno almeno un figlio sono chiamati **nodi interni**, mentre un nodo U è un **antenato** di un nodo V se:

- U=V oppure
- U è un antenato del padre di V

Un nodo V è un nodo **discendente** di un nodo U se U è un antenato di V, naturalmente. Un **sottoalbero** di un albero T radicato a V è un albero che consiste di tutti i discendenti di V.

La **profondità** di un nodo V è il numero dei suoi antenati (se stesso escluso), mentre l'**altezza** di un nodo V è il *massimo* numero dei suoi discendenti

La visita **preorder** di un albero T è una visita dei nodi T che si può definire ricorsivamente nel modo:

- Prima si visita la radice T
- Poi si fa una visita preorder per ciascuno dei sottoalberi radicati nei suoi figli

La visita **postorder** di un albero T è una visita dei nodi di T che si può definire ricorsivamente nel modo:

- Prima si fa una visita postorder dei sottoalberi radicati nei figli della radice
- Poi si visita la radice T

Se escludiamo il tempo per le chiamate ricorsive, gli algoritmi **preorder** e **postorder** impiegano tempo $O(1+c_v)$, dove c_v è il numero di figli di v.

Se cominciamo la visita dal nodo w, l'algoritmo viene invocato su tutti i discendenti di w

$$\bullet \text{ TEMPO TOTALE} = \sum_{v \in T_w} O(c_v + 1) = O(|T_w|)$$

La visita di tutto l'albero richiede tempo $O(|T|)$

Come il TDA lista, anche il TDA albero usa la nozione di posizione per memorizzare i suoi elementi, quindi anche qui un oggetto position offre il metodo element(), che restituisce l'oggetto memorizzato nella posizione corrente. Altri metodi presenti nel TDA albero sono:

- Metodi generici
 - Size(): numero di nodi dell'albero
 - isEmpty(): verifica se l'albero è vuoto
 - iterator(): restituisce un iteratore degli elementi nei nodi dell'albero
 - positions(): restituisce un iteratore dei nodi dell'albero
- Metodi di accesso
 - root(): restituisce la radice dell'albero; si verifica un errore se l'albero è vuoto
 - parent(v): restituisce il padre di V; si verifica un errore se V è la radice
 - children(v): restituisce un iteratore delle posizioni dei figli di V
- Metodi di interrogazione
 - isInternal(v): verifica se V è un nodo interno
 - isExternal(v): verifica se V è una foglia
 - isRoot(v): verifica se V è la radice
- Metodi di modifica
 - replace(v,e): rimpiazza con E l'elemento del nodo V e restituisce il vecchio elemento
 - insertChild(v,e): aggiunge un nuovo figlio al nodo V (il nuovo figlio conterrà l'elemento E)

Le eccezioni del TDA albero sono:

- **InvalidPositionException:** ogni metodo che prende in input una posizione, lancia questa eccezione se la posizione è invalida
- **BoundaryViolationException:** `parent(v)` lancia questa eccezione se il nodo `V` è la radice dell'albero
- **EmptyTreeException:** lanciata da `root()` se invocato un albero vuoto
- **NonEmptyTreeException:** lanciata da `addRoot(e)` se invocato su un albero non vuoto

Il TDA albero viene implementato mediante una struttura linkata; l'oggetto `TreePosition` è un oggetto con tre riferimenti: `parent`, `element` e collezione dei figli (può essere una lista o un array)

Proposizione

Altezza di un albero non vuoto = profondità massima delle foglie

Giustificazione: Altezza dell'albero = numero di archi lungo il percorso dalla radice alla foglia più distante

- Definiamo:
 - C_v = numero di figli di v
 - T_w = sottoalbero con radice w
 - $|T_w|$ = numero di nodi in T_w
- Si ha: $\sum_{v \in T} C_v = |T| - 1$
 - **DIM:** ad eccezione di w , ciascun nodo in T_w è figlio di un unico nodo di T_w e quindi dà un contributo pari ad 1 alla sommatoria

Il TDA `Position` è implementato dalla classe `TreeNode` che contiene riferimenti a:

- Elemento
- Nodo genitore
- Collezione dei nodi figlio

La classe `LinkedTree`: il metodo `checkPositions`

- `checkPositions(Position<E> v)`
 - lancia `InvalidPositionException` se
 - v è null
 - v non è di tipo `TreePosition`

Il metodo `positions()` restituisce una collezione iterabile contenente i nodi dell'albero nell'ordine in cui sono visitati durante una visita preorder. La visita richiede tempo $O(n)$

Binary Tree – Albero Binario

Un albero binario è un albero (ordinato) in cui ciascun nodo può avere al massimo due figli (figlio destro e figlio sinistro). Si parla di albero binario **proprio** quando ogni nodo ha esattamente due figli.

Un albero binario T o è vuoto o è formato da:

- Un nodo R detto radice;
- Un albero binario detto sottoalbero sinistro di T ;
- Un albero binario detto sottoalbero destro di T ;

Il TDA albero binario è un caso speciale del TDA albero, i metodi aggiuntivi sono:

- $\text{left}(v)$: restituisce il figlio sinistro di V ; si verifica un errore se V non ha il figlio sinistro
- $\text{right}(v)$: restituisce il figlio destro di V ; si verifica un errore se V non ha il figlio destro
- $\text{hasLeft}(v)$: verifica se V ha il nodo sinistro
- $\text{hasRight}(v)$: verifica se V ha il nodo destro

Visto che ogni nodo può avere al massimo 2 figli, ad un dato livello i (0 è il livello del nodo radice) ci saranno al più 2^i nodi.

La struttura utilizzata per un albero binario è sempre una struttura concatenata, e in una visita inorder un nodo viene visitato DOPO il suo sottoalbero sinistro e PRIMA del suo sottoalbero destro

La classe **LinkedBinaryTree** ha un costruttore che restituisce un albero binario vuoto. A partire da quest'ultimo, si può costruire un albero binario qualsiasi prima aggiungendo la radice E , poi via via gli altri nodi usando i seguenti metodi aggiuntivi:

- $\text{addRoot}(e)$: crea e restituisce un nuovo nodo R contenente l'elemento E , R diventa la radice dell'albero; si ha una condizione di errore se l'albero non è vuoto
- $\text{insertLeft}(v,e)$: crea e restituisce un nuovo nodo U contenente l'elemento E ; U diventa il figlio sinistro di V ; si ha una condizione di errore se V ha già il figlio sinistro
- $\text{insertRight}(v,e)$: crea e restituisce un nuovo nodo U contenente l'elemento E ; U diventa il figlio destro di V ; si ha una condizione di errore se V ha già il figlio destro
- $\text{remove}(v)$: rimuove il nodo V e lo sostituisce con suo figlio, se esiste, e restituisce l'elemento memorizzato in V ; si ha una condizione di errore se V ha due figli
- $\text{attach}(v,T1,T2)$: attacca $T1$ e $T2$ rispettivamente come sottoalbero sinistro e destro del nodo foglia V ; si ha una condizione di errore se V è un nodo interno

Heap

Un heap è un albero binario che memorizza una collezione di entrate nei suoi nodi e soddisfa due proprietà:

1. Proprietà di ordinamento – ogni nodo diverso dalla radice memorizza un elemento la cui chiave è maggiore o uguale di quella del suo padre
2. Proprietà strutturale – in ogni livello dell'albero c'è il massimo numero di nodi possibile, tranne che nell'ultimo che è riempito da sinistra a destra, cioè deve essere un *albero binario completo*.

Il massimo numero di nodi di un heap di altezza h : $1+2+4+\dots+2^{h-1}+1 = 2^h-1+1 = 2^h$

Quindi: $2^h \leq n \leq 2^{h+1}-1 \rightarrow \log(n+1)-1 \leq h \leq \log n \rightarrow h = \lfloor \log n \rfloor$ {limite inferiore}

Dunque un heap con N entrate ha altezza $H = \log N$

Il TDA Coda a priorità può essere implementato efficientemente con un heap, e consiste dei seguenti elementi:

- Un **heap**, cioè un albero binario completo che memorizza nei suoi nodi entrate (chiave, valore) le cui chiavi soddisfano l'ordinamento dell'heap
- Un **comparatore**, cioè un oggetto che definisce una relazione d'ordine totale tra le chiavi

ADD

Per l'inserimento di una nuova entrata con una chiave k , si esegue l'operazione di add che preserva la proprietà di albero binario completo ma non quella di ordinamento; poi si effettuano gli scambi con i nodi discendenti fino a che non si avrà un ordine con le chiavi. Questa tecnica è anche chiamata UPHEAP

Nel caso pessimo, si eseguono un numero di scambi pari all'altezza dell'albero, ovvero $O(\log n)$

REMOVEDMIN

Si copia l'entrata E dell'ultimo nodo nella radice, si cancella l'ultimo nodo e si continua a scambiare l'entrata E del nodo U con quella del figlio di U avente la più piccola chiave fino a che la proprietà di ordinamento non viene ristabilita. Questa tecnica è anche chiamata DOWNHEAP

Nel caso pessimo, si eseguono un numero di scambi pari all'altezza dell'albero, ovvero $O(\log n)$

COMPLESSITÀ

Size, isEmpty e min hanno complessità $O(1)$, mentre insert e removeMin hanno complessità $O(\log n)$

Adaptable Priority Queue (solo per gli ex studenti del corso De Bonis)

Il TDA Adaptable Priority Queue (APQ) estende il TDA Priority Queue con le seguenti operazioni:

- Remove(e): rimuove e sostituisce l'entrata E dalla coda a priorità
- ReplaceValue(E, V): rimpiazza con V il valore dell'entrata E restituendo in output il vecchio valore
- replaceKey(E, K): rimpiazza con K la chiave dell'entrata E restituendo in output la vecchia chiave

Motivazione: Alcuni algoritmi richiedono di cancellare un'entrata qualsiasi o di aggiornare l'elemento o la chiave di un'entrata qualsiasi

Implementazione

I metodi remove, replaceElement e replaceKey richiedono di conoscere il posto in cui l'entrata si trova all'interno dell'heap

- Soluzione inefficiente: il metodo replaceKey effettua la ricerca dell'entrata da modificare/cancellare
- Soluzione efficiente: indichiamo al metodo replaceKey il posto dell'entrata da modificare/cancellare

Locator

- Il TDA Priority Queue non supporta la nozione di Position sebbene possa essere implementato mediante un contenitore posizionale, abbiamo bisogno di un meccanismo per accedere direttamente ad un'entrata della coda
- Possiamo aggiungere alla classe che implementa Entry un campo che tiene traccia del posto (location) dove si trova l'entrata nella struttura dati usata per implementare la coda
- Se la struttura dati usata per implementare la coda supportata la nozione di Position allora location sarà di tipo Position
 - Implementazione con una lista: location contiene il riferimento alla Position della lista in cui è contenuta l'entrata
 - Implementazione con Heap: location contiene il riferimento al nodo dell'heap contenente l'entrata
- Quando un'entrata viene inserita nella coda, la sua location viene inizializzata con il riferimento alla sua Position nella struttura dati
- La location viene aggiornata ogni volta che l'entrata cambia posizione nella struttura dati

BinarySearchTree (solo per gli ex studenti del corso De Bonis)

Un albero di ricerca binario è un albero binario che memorizza in ciascun nodo una chiave in modo tale che se u , v e w sono 3 nodi interni tali che u si trova nel sottoalbero sinistro di v e w si trova nel sottoalbero destro di v , allora

$$\text{Key}(u) \leq \text{key}(v) \leq \text{key}(w)$$

(sulle chiavi è definita una relazione di ordine totale)

Operazioni supportate da un BST

- Cancellazione
- Inserimento
- Ricerca di una chiave
- Ricerca del minimo
- Ricerca del massimo
- Ricerca del successore
- Ricerca del predecessore

Visita inorder di un BST

- Una visita inorder di un albero di ricerca binario visita le chiavi in ordine non decrescente.
 - Possiamo ottenere la sequenza ordinata dalle chiavi

Algoritmo di ricerca

Input: la chiave da cercare k e un nodo v del BST

Output: un nodo w del sottoalbero radicato in v , tale che o w è un nodo interno contenente k o w è una foglia che si trova nel posto in cui si troverebbe k se appartenesse al sottoalbero

Algoritmo di inserimento

Input: una chiave k , un valore x , un nodo v

Output: un nuovo nodo del sottoalbero adicato in v che contiene la coppia (k, x)

- `insertAtExternal(w,e)` inserisce e nella foglia w e trasforma w in un nodo interno avente come figli due nuove foglie. Se v è un nodo interno, allora si verifica un errore

Algoritmo di cancellazione

Input: la chiave k da rimuovere \rightarrow `remove(k)`

Se k è contenuta in un nodo interno v allora distinguiamo due casi: uno dei figli di v è una foglia OPPURE entrambi i figli di v sono nodi interni

Caso 1: il nodo v da cancellare ha un figlio w che è una foglia

- l'algoritmo di cancellazione rimuove i nodi v e w invocando `removeExternal(w)`
- `removeExternal(w)` rimuove la foglia w e il padre di w dall'albero e sostituisce il padre di w con il suo fratello; se w non è una foglia allora si ha un errore.

Caso 2: la chiave da cancellare è contenuta in un nodo v in cui i figli sono entrambi nodi interni

- troviamo il primo nodo interno w visitato dopo v nella visita inorder (è il nodo interno più interno a sinistra nel sottoalbero destro di w)
- copiamo la chiave contenuta in w nel nodo v
- rimuoviamo w e il suo figlio sinistro z (che è una foglia) mediante `removeExternal(z)`

Dizionari implementati come BST

Possiamo implementare il TDA Dictionary mediante un albero di ricerca binario. Ogni entrata viene salvata in un nodo interno e c'è bisogno di un comparatore per confrontare le chiavi

Complessità dei metodi di Dictionary

Consideriamo un dizionario con n entrate implementato con un BST di altezza h

- lo spazio usato è $O(n)$
- i metodi `find`, `insert` e `remove` impiegano tempo $O(h)$

nel caso pessimo $h=O(n)$; nel caso ottimo $h=O(\log n)$

Set & Partition (solo per gli ex studenti del corso De Bonis)

Il TDA Set è un contenitore di oggetti, che sono a due a due distinti e in generale non è definita una relazione di ordine sugli elementi. Le operazioni del TDA Set sono

- `union(B)`
 - invocato sull'insieme A , rimpiazza A con l'unione di A e B
- `intersect(B)`
 - invocato sull'insieme A , rimpiazza A con l'intersezione di A e B
- `subtract(B)`
 - invocato sull'insieme A , rimpiazza A con la differenza di A e B

il modo più semplice per implementare il TDA insieme è quello di memorizzare gli elementi dell'insieme in una sequenza (`positionList`, `IndexList`, `Sequence`)

Implementazione di Set con una sequenza ordinata

Per rendere più efficienti le operazioni insiemistiche è utile definire una relazione d'ordine totale sull'insieme

- gli elementi dell'insieme vengono memorizzati in una sequenza ordinata
- VANTAGGIO: è possibile usare lo schema della procedura merge (usata nel mergesort) per implementare i metodi `union`, `intersect` e `subtract`
- La classe che implementa set avrà

- Una variabile di istanza del tipo usato per rappresentare la sequenza (indexList, PositionList o Sequence)
 - Una variabile di istanza di tipo Comparator usata per confrontare gli elementi dell'insieme
- Possiamo utilizzare un algoritmo generico (template method) merge per effettuare tre operazioni insiemistiche
- $A.union(B)$, $A.intersect(B)$ e $A.subtract(B)$, implementate con uno schema generico di merge, richiedono tempo $O(n+m)$

Algoritmo generico di merge di due liste

- Template method **genericMerge**
- Metodi ausiliari
 - `alsLess`
 - `blsLess`
 - `bothAreEqual`
- tempo $O(n_A+n_B)$ se i metodi ausiliari sono eseguiti in tempo $O(1)$

Uso di Generic Merge per le operazioni insiemistiche

- Intersection: copia solo gli elementi che appaiono in entrambe le liste
 - Occorre riscrivere il metodo `bothAreEqual`
- Union: copia ogni elemento e butta via duplicati
 - Ovvero riscrivere tutti e tre i metodi ausiliari
- Subtract: copia solo gli elementi che appaiono in A e non appaiono in B
 - Occorre riscrivere `alsLess`

TDA Partition

Una partizione è una collezione di insiemi S_1, \dots, S_k a due a due disgiunti, dunque $S_i \cap S_j = \emptyset$ per ogni $i \neq j$

Le operazioni supportate dal TDA Partition sono:

- `makeSet(x)`
 - crea l'insieme contenente il solo elemento x e lo aggiunge alla partizione
- `union(A,B)`
 - aggiunge alla partizione l'insieme unione di A e B distruggendo gli insiemi A e B
- `find(x)`
 - restituisce l'insieme che contiene l'elemento x

una semplice implementazione di Partition consiste nel memorizzare gli insiemi della partizione all'interno di una sequenza (IndexList, PositionList, Sequence).

La sequenza contiene oggetti di tipo Set (set è a sua volta implementato tramite una sequenza)

Implementazione efficiente di Partition: **union()**

dato che in una partizione gli insiemi sono due a due disgiunti, siamo sicuri che un elemento apparterrà solo ad un insieme. Di conseguenza possiamo effettuare l'unione di due insiemi velocemente.

Aggiungiamo all'implementazione di Set i metodi

- `public Set<E> fastUnion(Set B)`
- `public E fastInsert(E x)`
- `public Set<E> fastInsert(E x)`
 - inserisce x nell'insieme su cui è invocato senza verificare se x appartiene ad A
 - complessità $O(1)$

- `public Set<E>fastUnion(Set<E> B)`
 - inserisce gli elementi di B nell'insieme su cui è invocato senza verificare se appartengono già ad A
 - complessità $O(|B|)$

quando si esegue un'unione, spostiamo sempre gli elementi dall'insieme più piccolo a quello più grande (unione pesata)

- ogni volta che spostiamo un elemento, esso va a finire in un insieme che è almeno due volte più grande dell'insieme da cui proviene
- in questo modo, un elemento può essere spostato al più $O(\log n)$ volte

per ogni elemento spostato, aggiorniamo la voce corrispondente nella mappa

Implementazione efficiente di Partition

- `union(A,B)`
 - se $|A| > |B|$
 - aggiungiamo gli elementi di B ad A
 - rimuoviamo B dalla partizione
 - altrimenti
 - aggiungiamo gli elementi di A a B
 - rimuoviamo A dalla partizione
- invece di cercare in partizione la position P relativa all'insieme da rimuovere, inseriamo nella classe che implementa Set una variabile di tipo Position che tiene traccia della posizione in cui si trova l'insieme.

Proposizione: è possibile implementare Partition in modo tale che **una sequenza di n operazioni di makeset, find e union, eseguite a partire da una partizione vuota, richieda tempo ammortizzato $O(n \log n)$**

NB: la proposizione è vera per l'implementazione basata su una sequenza che usa

1. l'euristica dell'unione pesata,
2. un meccanismo per associare gli elementi agli insiemi di appartenenza che permetta di effettuare in tempo $O(1)$ le operazioni di find e di aggiornamento (dell'insieme associato ad un elemento)
3. un'implementazione di Set che permetta di tenere traccia della posizione di ciascun insieme nella partizione

tale preposizione non vale per l'implementazione in cui si usa una tabella hash per gestire l'associazione tra elementi e insiemi. In quell'implementazione le operazioni di find e di aggiornamento richiedono tempo costante solo **nel caso medio**.

Grafi

Il TDA Graph è una collezione di elementi memorizzati nelle position del grafo. Ciascuna position del grafo corrisponde ad un vertice o ad un arco.

Corrisponde al grafo non direzionato, per quello direzionato occorre definire dei metodi aggiuntivi. I metodi di accesso e di interrogazione sono:

- `numVertices()`: restituisce il numero di vertici del grafo
- `numEdges()`: restituisce il numero di archi del grafo
- `endVertices(e)`: restituisce un array contenente le due estremità di e
- `opposite(v, e)`: restituisce il vertice incidente su e opposto al vertice v
- `areAdjacent(v,w)`: restituisce TRUE solo se i vertici v e w sono adiacenti

Metodi che restituiscono collezioni iterabili

- `incidentEdges(v)`: restituisce una collezione iterabile degli archi incidenti su v
- `vertices()`: restituisce una collezione iterabile dei vertici nel grafo
- `edges()`: restituisce una collezione iterabile degli archi nel grafo

Metodi di aggiornamento

- `replace(v,x)`: sostituisce l'elemento nel vertice `v` con `x` e restituisce l'elemento memorizzato precedentemente in `v`
- `replace(e,x)`: sostituisce l'elemento nell'arco `e` con `x` e sostituisce l'elemento memorizzato precedentemente in `e`
- `insertVertex(o)`: inserisce restituendo in output un vertice che memorizza l'elemento `o`
- `insertEdge(v,w,o)`: inserisce restituendo in output un arco `(v,w)` che memorizza l'elemento `o`
- `removeVertex(v)`: cancella il vertice `v` ed i suoi archi incidenti e restituisce in output l'elemento di `v`
- `removeEdge(e)`: cancella l'arco `e` e restituisce in output l'elemento di `e`

TDA Directed Graph

In aggiunta ai metodi del TDA Graph, il TDA Directed Graph ha anche i seguenti metodi:

- `isDirected(e)`
 - restituisce `TRUE` se l'arco è direzionato
- `insertDirectedEdges(u,v,o)`
 - aggiunge al grafo l'arco direzionato `(u,v)` avente come elemento `o`

Bisogna notare che se `E` è un arco direzionato, allora il metodo `endVertices(E)` deve restituire un array `A` tale che `A[0]` contiene l'origine dell'arco e `A[1]` contiene la destinazione dell'arco.

Potrebbe essere utile aggiungere i metodi:

- `inIncidentEdges(v)`
 - restituisce una collezione iterabile degli archi entranti in `V`
- `outIncidenceEdges(v)`
 - restituisce una collezione iterabile degli archi uscenti da `V`

Grafi pesati e non

Si possono rappresentare i grafi pesati e quelli non pesati nello stesso modo. Nel caso di un grafo pesato, l'elemento dell'arco rappresenterà il peso dell'arco. Nel caso di un grafo NON pesato, l'elemento dell'arco sarà `NULL`

Implementazioni di Graph

Le loro implementazioni cambiano a seconda della rappresentazione che si utilizza per Graph

- lista di archi (edge list)
- matrice di adiacenza (adjacent matrix)
- liste di adiacenza (adjacent list)

Liste di adiacenza

Consiste di una collezione di tutti i vertici, di tutti gli archi e per ciascun vertice `v`, è una collezione `I(v)` degli archi incidenti su `v`

Ciascun vertice `v` contiene un riferimento alla collezione `I(v)` degli archi incidenti su `v`. Ciascun arco `e=(u,v)` contiene i riferimenti alle posizioni relative ad `E` nelle collezioni `I(u)` e `I(v)`.

Tipicamente le collezioni `I(v)` vengono rappresentate con delle liste.

Implementazione di Vertex e di Edge

- l'implementazione di Vertex contiene
 - riferimento all'elemento
 - riferimento a `I(v)`
 - `protected PositionList<Edge<E>> incEdges;`

- riferimento alla sua posizione nella lista di vertici
 - `protected Position<Vertex<V>> loc;`
- l'implementazione di Edge contiene
 - riferimento all'**elemento**
 - riferimenti alle posizioni relative all'arco nelle liste di incidenza della sua estremità
 - `protected Position<Edge<E>>[] inc;`
 - riferimento alla sua posizione nella lista degli archi
 - `protected Position<Edge<E>> inc;`
 - riferimenti alle posizioni dei vertici u e v nella lista dei vertici
 - `protected MyVertex<V>[] endVertices;`

Rappresentazione mediante lista di archi

Il grafo è rappresentato tramite due liste

- Una lista (NodeList) o un vettore (ArrayList) conserva i vertici del grafo
- Un'altra lista (NodeList) o vettore (ArrayList) conserva gli archi del grafo


Per effettuare facilmente la ricerca degli archi si potrebbe utilizzare un dizionario per rappresentare la collezione degli archi invece che una lista

- L'entrata del dizionario associata all'arco E ha come chiave l'elemento dell'arco E e come valore l'arco E
- Si può fare la stessa cosa per i vertici.

Richiede inoltre di scandire tutti gli archi del grafo per trovare gli archi incidenti su un certo vertice

Matrice di adiacenza

Permette di scoprire velocemente se due vertici sono adiacenti, richiede spazio $O(n^2)$

$n = \# \text{ nodi}$ $m = \# \text{ archi}$ 			
	Lista di archi	Liste di Adiacenza	Matrice di Adiacenza
Spazio	$n + m$	$n + m$	n^2
<code>incidentEdges(v)</code>	m	1	n
<code>areAdjacent(v, w)</code>	m	$\min(\text{grado}(v), \text{grado}(w))$	1
<code>insertVertex(o)</code>	1	1	n^2
<code>insertEdge(v, w, o)</code>	1	1	1
<code>removeVertex(v)</code>	m	$\text{grado}(v)$	n^2
<code>opposite(v, e), endVertices(e)</code>	1	1	1
<code>replace(e, x), replace(v, x)</code>	1	1	1
<code>removeEdge(e)</code>	1	1	1

Il Design Pattern Decorator

Serve ad "attaccare" informazioni extra ad un oggetto; le informazioni vengono aggiunte dinamicamente e ciascuna di esse consiste di una coppia (attributo, valore).

ESEMPIO: per implementare la DFS e la BFS è utile "decorare" i vertici e gli archi con l'attributo **esplorato** a cui è associato un valore booleano

Alternative:

- Avremmo potuto inserire il campo **esplorato** di tipo boolean nell'implementazione di Vertex ed Edge
 - Problema: l'implementazione del grafo non è più generica ma dipende dall'uso che intendiamo farne.
Ad esempio, nel nostro caso vogliamo effettuare una visita DFS del grafo
- Avremmo potuto usare una tabella hash contenente gli archi e i vertici esplorati
 - Problema: nel caso peggiore il tempo non è costante per marcare "esplorato" un vertice o un arco

TDA Decorable Position

Possiamo realizzare un decoratore per un contenitore posizionale definendo il TDA Decorable Position che unisce i metodi del TDA Map a quelli del TDA Position:

- Element()
- Size()
- isEmpty()
- entries(): restituisce tutte le coppie (attributo, valore) associate alla posizione
- get(a): restituisce il valore dell'attributo A
- put(a,x): pone il valore di K uguale ad X
- remove(a): rimuove dalla posizione l'attributo A e il suo valore

Mappa

Una mappa è un contenitore di elementi del tipo (K,V) dove K è la chiave e V il suo corrispondente valore. Ogni elemento (K,V) viene detto entrata /entry della mappa, e le entrate multiple con la stessa chiave NON sono permesse; le chiavi, oltre ad essere il mezzo per accedere agli elementi, hanno lo scopo di rendere efficiente la ricerca. I metodi fondamentali del TDA Mappa sono:

- Get(K): se la mappa M ha un'entrata con chiave K, restituisce il valore ad essa associata; altrimenti restituisce NULL
- Put(K,V): se la chiave K non è già in M, inserisce l'entrata (K,V) nella mappa M e restituisce NULL; altrimenti rimpiazza con V il valore esistente e restituisce il valore associato a K
- Remove(K): se la mappa M ha un'entrata con chiave K, la rimuove da M e restituisce il valore associato; altrimenti restituisce NULL
- Keys(): restituisce una collezione iterabile delle chiavi in M
- Values(): restituisce una collezione iterabile dei valori in M
- Entries(): restituisce una collezione iterabile delle entrate chiave-valore di M
- Size(), isEmpty()

L'implementazione del TDA Mappa avviene tramite lista doppiamente concatenata non ordinata, ed è conveniente solo per mappe di piccola taglia

Complessità

- PUT() richiede il tempo necessario per verificare che la chiave non sia già nella mappa ($O(n)$ nel caso pessimo); poiché la lista non è ordinata possiamo inserire la nuova entrata all'inizio o alla fine della lista
- GET() e REMOVE() richiedono tempo $O(n)$ (nel caso pessimo la chiave non viene trovata) per scandire l'intera lista

Uno dei metodi più efficienti per implementare una mappa è usare una tabella **hash**. Una tabella hash, per un dato tipo di valore, è composta da un array (chiamato bucket array) e una funzione hash H.

Nel bucket array, ciascuna cella è un contenitore di coppie (K,V), e le chiavi sono tutte distinte e scelte nell'intervallo $[0, n-1]$; per questo si ha bisogno di un bucket array di dimensione n, l'entrata con chiave k verrà così inserita nella k-esima cella dell'array, mentre tutte le operazioni richiedono tempo $O(1)$

Sorgono due problemi:

1. Lo spazio sarà sempre proporzionale ad n (numero di tutte le chiavi possibili): se n è molto più grande del numero di chiavi realmente presenti si ha un notevole spreco di memoria
2. Non sempre le chiavi sono interi nell'intervallo $[0, n-1]$

Per ovviare a tutto ciò, si usa una funzione hash h , che associa chiavi di tipo arbitrario ad interi in un intervallo fissato $[0, N-1]$. Un esempio: $h(x) = x \bmod N$ è una semplice funzione hash per chiavi intere; l'intero $h(x)$ viene chiamato **valore hash** della chiave x .

In generale, una funzione hash opera in due fasi

- *Hash code*: trasforma oggetti arbitrari in interi (per risolvere il problema 2)
- *Compressione*: trasforma interi in un intervallo arbitrariamente grande in interi dell'intervallo $[0, N-1]$ (per risolvere il problema 1)

Quando implementiamo una mappa con una tabella hash, lo scopo è di memorizzare l'entrata (K, V) all'indice $i=h(k)$ dell'array. Nel caso in cui si hanno due entrate distinte con lo stesso valore hash, verrebbero associate alla stessa cella, generando quindi una collisione. Per evitare ciò si implementa la tabella di hash secondo uno dei due seguenti criteri:

- Linear Probing
 - Quando si tenta di inserire un'entrata (K, V) in una cella $A[i]$ già occupata (cioè tale che $h(k)=i$), si prova con la cella $A[(i+1) \bmod N]$, anche se questa è occupata si passa alla cella $A[(i+2) \bmod N]$ e così via, fino a trovarne una vuota
- Separate Chaining
 - Ciascuna cella $A[i]$ ospita una lista L_i contenente tutte le entrate (K, V) tali che $h(k)=i$

La probabilità di avere una collisione fra chiavi dipende dal load factor "lambda", definito come il rapporto tra il numero delle chiavi memorizzate sulla capacità del bucket array; conviene mantenere tale rapporto <0.5 per il linear probing e <0.9 per il separate chaining. Quando non ci si trova più nel giusto range, bisogna effettuare il rehash, cioè ridefinire la funzione di hash con nuovi parametri, raddoppiare la capacità del bucket array e generare nuovi fattori di scale e shift mediante i quali sarà possibile creare una nuova mappa con lambda nel giusto range.

II TDA Dizionario

È una collezione di entrate del tipo (chiave, elemento). Le operazioni principali sono di ricerca, inserimento e cancellazione. **Si possono avere più elementi con la stessa chiave, contrariamente alle mappe.** I metodi del TDA Dizionario sono:

- Metodi di accesso
 - Find(k): Restituisce un'entrata del dizionario D con chiave k. Se non esiste alcuna entrata con chiave k restituisce NULL.
 - findAll(k): restituisce una collezione iterabile di tutte le entrate che hanno chiave k.
 - Entries(): Restituisce una collezione iterabile di tutte le entrate del dizionario
- Metodi di aggiornamento
 - Insert(k,o): Inserisce e restituisce in output l'entrata (k,o)
 - Remove(e): Rimuove e restituisce in output l'entrata e. Se e non appartiene al dizionario, restituisce NULL
- Metodi generici
 - Size()
 - isEmpty()

Se ai metodi find, findAll e insert viene passata in input una chiave non valida, allora si verifica un errore.

Se al metodo remove viene passata un'entrata non valida si verifica un errore.

2 Tipi di dizionari

- Dizionari non ordinati
 - Non è definita alcuna relazione di ordinamento sulle chiavi
 - Si può stabilire se sono due chiavi uguali o meno
- Dizionari ordinati
 - Sulle chiavi è definita una relazione di ordine totale
 - Al costruttore del dizionario viene passato un comparatore come argomento
 - Si possono definire metodi aggizionali che fanno riferimento all'ordinamento delle chiavi

Log file

Un log file è un dizionario non ordinato implementato con una lista (non ordinata); le entrate sono memorizzate nella lista in un ordine arbitrario.

- Insert richiede tempo $O(1)$ se implementata con `addFirst` o `addLast`
- Find and remove richiedono tempo $O(n)$ in quanto nel caso pessimo, che si verifica quando la chiave non è presente nel dizionario, si scandisce l'intera sequenza per trovare l'entrata con la chiave desiderata

È indicato implementare un dizionario come log file solo se si tratta di un dizionario di piccola dimensione, o nel caso in cui si effettuano molti inserimenti e poche operazioni di cancellazione e ricerca

Ordered Search table

Una ordered search table è un dizionario ordinato implementato con un array list (ordinato); le entrate vengono immagazzinate in un arraylist nel quale sono ordinate in base al valore delle chiavi. Si usa un comparatore esterno per le chiavi

- Find richiede tempo $O(\log n)$ con la ricerca binaria
- Insert richiede tempo $O(n)$ perché occorre shiftare a destra tutte le entrate con chiave maggiore della chiave della nuova entrata
- Remove richiede tempo $O(n)$ perché occorre shiftare a sinistra tutte le entrate che seguono l'entrata che deve essere rimossa
- È indicato implementare un dizionario come ordered search table solo se si tratta di un dizionario piccolo o se si effettuano molte operazioni di ricerca e poche operazioni di inserimento e cancellazione senza aver trovato un'entrata con chiave k