INDICE degli appunti

TREE

Premesso che, il TDA **Position** rappresenta l'astrazione di nodo e fornisce l'unico metodo **element()**
che restituisce l'elemento memorizzato in quella posizione;
il TDA Tree usa il TDA Position per memorizzare e localizzare i nodi nella struttura gerarchica Tree.

- i metodi dell'intefaccia del TDA Tree: Tree<E> -

metodi generici:
size() e **isEmpty()** : già li conosciamo.
iterator() : restituisce un iteratore degli elementi dell'albero.
positions() : restituisce una collezione iterabile delle posizioni degli elementi dell'albero.
metodi di accesso:
root() : restituisce la posizione della radice
parent(p) : restituisce la posizione del padre del nodo di posizione p.
children(p) : restituisce una collezione iterabile delle posizioni dei figli del nodo in posizione p.
metodi di interrogazione:
isInternal(p)
isExternal(p)
isRoot(p)
metodi di aggiornamento:
replace(p,e) : rimpiazza l'elemento in posizione **p** con **e**, restituendo il vecchio elemento
di seguito l'interfaccia dei sudetti metodi:

```
public interface Tree<E> {
public int size();  // già conosciuto
public boolean isEmpty();   // già conosciuto
/** Restituisce un iteratore degli elementi memorizzati nell'albero. */
   public Iterator<E> iterator();
/** Restituisce una collezione iterabile delle posizioni dei nodi. */
   public Iterable<Position<E>> positions();
/** Rimpiazza l'elemento memorizzato in un dato nodo. */
   public E replace(Position<E> v, E e)   throws InvalidPositionException;
/** Restituisce la radice dell'albero. */
   public Position<E> root()   throws EmptyTreeException;
/** Restituisce il padre di un dato nodo. */
   public Position<E> parent(Position<E> v) throws InvalidPositionException, BoundaryViolationException;
/** Restituisce una collezione iterabile delle posizioni dei figli di un dato nodo di posizione v. */
   public Iterable<Position<E>> children(Position<E> v)   throws InvalidPositionException;
/** Ci dice se un dato nodo è interno. */
   public boolean isInternal(Position<E> v)   throws InvalidPositionException;
/** Ci dice se un dato nodo è esterno (una foglia). */
   public boolean isExternal(Position<E> v)   throws InvalidPositionException;
/** Ci dice se un dato nodo è la radice di questo albero. */
   public boolean isRoot(Position<E> v)   throws InvalidPositionException;
}    // fine interfaccia
```

FINE

Binary Tree

Il TDA BinaryTree estende, ossia specializza, il TDA Tree
Di seguito l'interfaccia di BinaryTree:

```
public interface BinaryTree<E> extends Tree<E> {
public Position<E> left(Position<E> v) throws InvalidPositionException, BoundaryViolationException;
public Position<E> right(Position<E> v) throws InvalidPositionException, BoundaryViolationException;
public boolean hasLeft(Position<E> v) throws InvalidPositionException;
public boolean hasRight(Position<E> v) throws InvalidPositionException;
} // fine interfaccia
```

In una implementazione con lista concatenata, **LinkedBinaryTree**
ogni nodo contiene i seguenti campi-riferimento-a:
un reference all' elemento stesso
un reference al nodo padre
un reference al figlio sinistro
un reference al figlio destro

*la sua classe **BTNode**:*

```
public class BTNode<E> implements BTPosition<E> {
private E element;
private BTPosition<E> left, right, parent;    // reference al figlio sinistro, al figlio destro e al padre
// il costruttore
public BTNode(E element, BTPosition<E> parent, BTPosition<E> left, BTPosition<E> right) {
        this.element = element;
        this.parent = parent;
        this.left = left;
        this.right = right;
      }
public E element() { return element; }
public void setElement(E o) { element=o; }
public BTPosition<E> getLeft() { return left; }
public void setLeft(BTPosition<E> v) { left=v; }
public BTPosition<E> getRight() { return right; }
public void setRight(BTPosition<E> v) { right=v; }
public BTPosition<E> getParent() { return parent; }
public void setParent(BTPosition<E> v) { parent=v; }
} // fine classe del nodo
```

di seguito l'interfaccia Position situata nel package position:

```
public interface Position<E> {
E element();
}
```

di seguito il TDA Position (BTPosition<E>) implementato dalla classe BTNode<E>

```
public interface BTPosition<E> extends Position<E> {
public void setElement(E o);
public BTPosition<E> getLeft();
public void setLeft(BTPosition<E> v);
public BTPosition<E> getRight();
public void setRight(BTPosition<E> v);
public BTPosition<E> getParent();
public void setParent(BTPosition<E> v);
}// fine interfaccia di BTPosition
```

tre sono i metodi per le visite: preorder, inorder, postorder tutti di complessità O(n).
si differenziano per l'ordine con cui il nodo viene visitato;

Algorithm PreOrder(v)
visit(v)
if hasLeft (v)==true
  PreOrder (leftChild (v))
if hasRight (v)==true
  PreOrder (rightChild (v))

Algorithm inOrder(v)
if hasLeft (v)==true
  inOrder (leftChild (v))
visit(v)
if hasRight (v)==true
  inOrder (rightChild (v))

Algorithm PostOrder(v)
if hasLeft (v)==true
  PostOrder (leftChild (v))
if hasRight (v)==true
  PostOrder (rightChild (v))
visit(v)

FINE

CompleteBinaryTree

Un albero binario completo è un albero binario in cui ogni livello, fino al penultimo, è completamente riempito. L'ultimo livello è riempito da sinistra a destra; l'ultimo livello può contenere un numero di nodi inferiore al massimo possibile, ma deve essere riempito da sinistra a destra

Il TDA CompleteBinaryTree estende, ossia specializza, il TDA BinaryTree
e aggiunge due metodi:
Position<E> add(E e); che inserisce una foglia contenente l'elemento **e**, restituendo sua la posizione
E remove(); rimuove l'ultimo nodo dell'albero restituendone l'elemento.
di seguito l'interfaccia CompleteBinaryTree:

```
public interface CompleteBinaryTree <E> extends BinaryTree <E> {
   public Position <E> add(E elem);
   public E remove();
} // fine interfaccia
```

L'implementazione coi vettori prevede l'uso di un ArrayList T
le operazioni di add e remove richiedono tempo O(1), viene coinvolto solo l'ultimo elemento dell'arraylist
la radice è memorizzata all'indice 1 e per ogni nodo v memorizzato all'indice i:
- il suo figlio sinistro è memorizzato all'indice 2i
- il suo figlio destro è memorizzato all'indice 2i + 1

FINE

Binary Search Tree (BST) (Alberi di Ricerca Binari)
E' un albero binario (= albero con al più due figli) con la seguente caratteristica:
per ogni *nodo interno v* si ha che:
le chiavi k di ogni *nodo* ***s*** del sottoalbero sinistro sono minori o uguali a quella contenuta nel nodo **v** e
le chiavi k di ogni *nodo* ***d*** del sottoalbero destro sono maggiori o uguali a quella contenuta nel nodo **v** ,

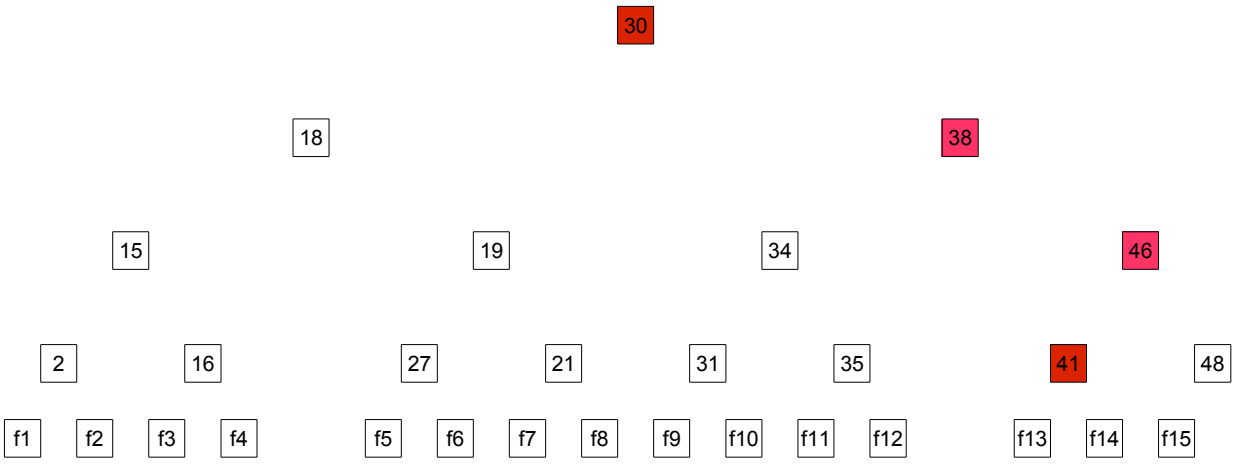
key(s) ≤ key(v) ≤ key(d)

(cioè sulle chiavi è definita questa particolare relazione di ordine totale)

Assumeremo inoltre che il BST sia proprio, cioè ogni nodo o è una foglia o ha due figli, ad esclusione delle foglie che sono vuote, cioè non contengono valori, quindi i dati sono memorizzati solo nei nodi interni.
nota: per come è strutturato, il BST, visitato con inorder produce la lista ordinata delle chiavi k.
le operazioni principali supportate da un BST sono:
Inserimento: **treeInsert (k,x,v)** inserisce una entrata (k,x) nel sottoalbero radicato in v
Cancellazione: **remove(k)** sono due i casi bene illustrati nelle slide n.237 DeBonis e n.492 DeMarco.
Ricerca di una chiave: **treeSearch (k,v)** ricerca la chiave k a partire da un nodo v
la complessità è O(h) (dove h = altezza dell'albero) per tutti e tre i metodi.
Ricerca del minimo:
Ricerca del massimo:
Ricerca del successore:
Ricerca del predecessore:
insertAtExternal(w,(k,v)): è un metodo ausiliario usato da TreeInsert() che viene richiamato nel caso in cui w (=nodo in cui inserire) è una foglia, in tal caso il metodo trasforma il nodo w in un nodo interno con due figli-foglia e lo fà richiamando a sua volta il metodo expandExternal(Position<E> v, E l, E r))

<u>Si può implementare un dizionario con un BST grazie alla sua caratteristica di ricerca veloce,</u>
la complessità in questo caso è:
O(n) nel caso pessimo che è quello in cui il BST è tutto sbilanciato da un solo lato.
O(log n) nel caso ottimo quello invece in cui l'albero e perfettamente bilanciato.
nell'implementazione di un dizionario è previsto l'uso di un comparatore per il confronto delle chiavi k.

illustrazione della fase di ricerca: cerca la chiave k = 41: **treeSearch (41, 30)**



FINE

HEAP

Albero binario che contiene entrate (k,v) nei propri nodi e che gode di due proprietà:

1. **(proprietà di ordinamento) ogni nodo contiene un elemento, la cui chiave k è maggiore o uguale a quella di suo padre.**
2. **(proprietà strutturale) ogni livello ha il massimo numero di nodi possibile, tranne l'ultimo cioè quello delle foglie che è riempito nel verso da sinistra a destra (*albero binario completo*).**

L'altezza di un heap che contiene n entrate(nodi) è data da: h = log n .

lo heap si presta bene all'implementazione delle code a priorità.

FINE

PriorityQueue (code a priorità)

E' un contenitore di oggetti fatti di entrate del tipo chiave-valore (k,v); la chiave specifica la priorità dell'oggetto nella collezione. Può essere rimosso solo l'elemento con la più alta priorità.
Il tipo di priorità (numerica, lessicografica,...) è stabilito da un comparatore (è lui che stabilisce la regola di priorità).
Ogni volta che viene costruita una nuova coda a priorità, viene fornito un oggetto Comparator che sia per così dire *personalizzato* a quel tipo di priorità.

Implementazione con lista doppiamente concatenata:

Con lista non ordinata:

Caratteristiche: inserimento veloce O(1) alla fine della lista e cancellazione lenta, scorrimento della lista O(n) per cercare l'elemento con chiave minima.

Con lista ordinata:

Caratteristiche: inserimento lento O(n), deve scorrere la lista per l'inserimento ordinato e cancellazione veloce O(1) perchè l'elemento con chiave minima da eliminare è in prima posizione.

Implementazione con lo heap:

ma l' implementazione più efficiente di una coda a priorità si ottiene utilizzando uno heap
e si basa sull'idea di memorizzare i dati entrate chiave-valore (k,v) in un albero binario, servono un heap e un comparatore, oggetto quest' ultimo che serve a definire una relazione d'ordine totale sulle chiavi k. Ogni entrata è un oggetto Entry(k,v) formato da due oggetti: chiave – valore, e la chiave piu' piccola si trova nella radice dell'albero.

i metodi fondamentali: (slide n. 370 dell'unico De Marco)

```
public Entry<K,V> insert(K key, V value) throws InvalidKeyException
```

l'inserimento prevede l'aggiunta della nuova entrata alla fine dello heap e, successivamente, il confronto col genitore con spostamento all' insù dell'entrata al posto del genitore, fin quando la nuova entrata non risulta <= del padre. *Complessità log n nel caso pessimo.*

```
public Entry<K,V> removeMin() throws EmptyPriorityQueueException
```

1) copiamo l'entrata < *e* > dell'ultimo nodo nella radice
2) cancelliamo l'ultimo nodo
3) continuiamo a scambiare l'entrata < *e* > nel nodo *u* con quella del figlio di *u* avente la più piccola chiave fino a che la proprietà di ordinamento non verrà ristabilita.

Complessità log n nel caso pessimo.

L'ADT Comparator fornisce un meccanismo di confronto basato sul seguente unico metodo:

int compare(E a, E b): restituisce un intero *i* tale che:

i < 0 se a < b
i = 0 se a = b
I > 0 se a > b

FINE

AdaptablePriorityQueue

Questo TDA estende (quindi specializza) le code a priorità aggiungendo i metodi:

Remove (e) : che rimuove e restituisce l'entrata **e** dalla coda a priorità

replaceKey(e,k): rimpiazza con **k** la chiave dell'entrata **e** restituendo in output la vecchia chiave.

replaceValue(e,v) : rimpiazza con **v** il valore dell'entrata **e** restituendo in output il vecchio valore.

domanda: perchè c'è bisogno di questa particolare estensione delle code a priorità?

risposta: perchè in alcuni algoritmi (Prim e Dijkstra) c'è l'esigenza di aggiornare le priorità (le key) oppure i valori.

Ora, siccome nelle code a priorità non è supportato il concetto di *position*, per accedere agli elementi nella coda si aggiunge un campo detto **locator**, che tiene traccia della posizione delle entrate.

protected Position <Entry<K,V>> loc;

Aggiungiamo, alla classe che implementa Entry, un campo che tiene traccia del posto (location) in cui si trova l'entrata nella struttura dati usata per implementare la coda.

Se la struttura dati usata per implementare la coda supporta la nozione di Position allora location sarà di tipo Position:

l'implementazione con una lista: location contiene il riferimento alla Position della lista in cui è contenuta l'entrata

l'implementazione con Heap: location contiene il riferimento al nodo dello heap che contiene l'entrata

Implementazione di Prim e Dijkstra con AdaptablePriorityQueue

Uso della coda a priorità negli algoritmi Prim e Dijkstra:

Scelta greedy basata sul valore delle chiavi assegnate ai vertici

i vertici del grafo vengono inseriti in una coda a priorità;

ad ogni passo viene estratto dalla coda il vertice con priorità più alta (chiave più piccola)

negli algoritmi Prim e Dijkstra si utilizza una versione specializzata delle code a priorità:-

- **AdaptablePriorityQueue**, perchè in questo tipo di algoritmi sono previste modifiche alle chiavi **k**, o agli elementi **v** delle entrate **(k,v)** e quindi questa versione particolare delle code a priorità prevede con metodi ad hoc tali modifiche.

FINE

Map

Cos'è una Mappa ?

La mappa è un contenitore di elementi del tipo chiave-valore (k,v) dove l'elemento (k,v) è detto entrata; le chiavi, a differenza dei dizionari, devono essere uniche nella tabella, cioè non sono ammesse due chiavi con lo stesso valore.
Le chiavi rappresentano il mezzo per accedere agli elementi ed hanno lo scopo di rendere efficiente la ricerca;

i metodi fondamentali:
– **size()**, **isEmpty()** : i soliti metodi che non hanno bisogno di spiegazioni
– **get(k)**: se la mappa M ha una entrata con chiave k, restituisce il valore associato alla chiave, 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 vecchio valore associato a k
– **remove(k)**: se la mappa M ha una entrata con chiave k, la rimuove da M e restituisce il valore ad essa associato; altrimenti, restituisce null.
– **keys()**: restituisce una collezione iterabile delle chiavi in M, per es:
(**keys().iterator** restituisce un iteratore sulle chiavi)
– values(): restituisce una collezione iterabile dei valori in M, per es:
(**values().iterator()** restituisce un iteratore sui valori)
– entries(): restituisce una collezione iterabile delle entrate chiave-valore di M, per es:
(**entries().iterator()** restituisce un iteratore sulle entrate)

L'ADT Mappa può essere implementato con lista doppiamente concatenata (non ordinata)
qui i metodi get(), put(), remove() richiedono, nel caso pessimo, O(n); nella put() il tempo si perde per verificare se la chiave esiste già, mentre l'inserimento è rapidissimo(costante), perchè essendo lista non ordinata, esso va posto in coda;
Questa implementazione va bene per mappe di piccola taglia.

Un modo molto efficiente di implementare le mappe è quello in cui si utilizzano la *tabelle hash* partendo prima da una implementazione iniziale che prevede un array particolare, detto *buchet array* (leggi *bochet*) in cui ogni cella non contiene un valore unico, ma è essa stessa un contenitore di due valori, l'entrata (k,v), chiave-valore (buchet = contenitore) .
Questa implementazione però costringe a creare un array grande almeno quanto la chiave più grande, essendo proprio la chiave l'indice di posto che contiene il valore associato a quella chiave, e questo rappresenta un enorme spreco di memoria pur fornendo un accesso ai valori rapidissimo O(1).
Lo spreco consiste nel fatto che, anche per pochi elementi, io portei aver necessità di creare un array enorme, inoltre le chiavi potrebbero anche non essere dei numeri;
ed ecco, allora, che entrano in gioco le tabelle hash.
Una **tabella hash** è rappresentata oltre che da un array (*buchet array*), anche da una funzione hash .

RICORDA quindi che **le tabelle hash servono ad evitare lo spreco di memoria**

all'interno della classe **HashTableMap** che implementa **Map**
la funzione hash di una chiave **k** viene calcolata dal metodo **public int hashValue(K key)** che opera in due fasi:-
1). hash-code (codifica): che trasforma una chiave di qualsiasi tipo di oggetto, in un numero intero.
2). compressione: dispone *uniformemente* le chiavi in un intervallo predefinito più piccolo;

questa soluzione però presenta qualche problema:-
- per quanto efficiente, una funzione hash, potrebbe in taluni casi emettere una chiave già presente, per cui si verificherebbe in tal caso una *collisione di chiavi* uguali, allora due le soluzioni:
Separate chaining: ogni cella del buchet array è un reference (praticamente un puntatore) a lista concatenata, in cui vengono aggiunte tutte le entrate con chiavi uguali.
Linear probing: (Open addressing):quando,in inserimento, l'indice calcolato dalla funzione hash è già presente, vuol dire che la funzione hash "ha sbagliato" allora si verifica una collisione: è accaduto quindi che due chiavi distinte hanno lo stesso valore hash e risultano associate alla stessa cella.
si inizia allora a scorrere in modo lineare l'array, cercando una postazione libera per memorizzare la chiave, e così va bene; però ulteriore complicazione: quando si vuole eliminare una chiave, bisogna scalare all'indietro tutte le chiavi collise per quella entrata, con una enorme complicazione del codice, perchè se così non si facesse, si dovrebbe lasciare vuota la postazione, ma poi quando si andrebbe in ricerca, il programma sarebbe indotto in errore, perchè di fronte ad una cella vuota riterrebbe conclusa la ricerca, uscendo erroneamente dalla fase di ricerca.
a tale problema, però, si può ovviare semplicemente contrassegnando le postazioni cancellate con un simbolo di cella disponibile (**AVAILABLE**); in tal modo, in ricerca queste celle contrassegnate verrebbero interpetrate come celle diponibili e non indicanti la fine della ricerca.
Comunque le collisioni andrebbero sempre evitate utilizzando una funzione hash molto efficiente, siccome però, come già detto, la perfezione non esiste, si può aiutare la stessa funzione hash ad essere performante, mantenendo **il rapporto n/N** , cioè numero di chiavi presenti nell'array **n** fratto la grandezza dello stesso array **N** ad un valore tale da minimizzare al massimo l'evento della collisione; **tale rapporto è detto fattore di carico (load factor).**

$\lambda <$ **0.5** nel linear probing e $\lambda <$ **0.9** nel separate chaining

nel probe il modulo (calcolo del resto) mi serve per creare una struttura circolare, perchè quando trovo la cella occupata e vado in ricerca successiva per cercare una locazione libera, se arrivo alla fine senza esito, ritorno indietro attraverso il calcolo del mod N.
La funzione hash che java mette a disposizione non va bene perchè utilizza i valori presenti nelle chiavi e quindi andrebbe bene per valori primitivi e non per chiavi di tipo non primitivo; infatti in tal caso utilizzerebbe l'indirizzo per calcolare il numero intero h (il valore hash), ma così facendo, se per caso due chiavi-oggetto fossero distinte in locazioni diverse di memoria, ma avessero lo stesso stato (quindi fossero uguali), la funzione darebbe valori diversi per due chiavi uguali, il che vìola la proprietà fondamentale della funzione hash, secondo cui per chiavi uguali la funzione hash deve produrre lo stesso valore hash h.
metodi alternativi al linear probing sono:
quadrating probing: $\mathbf{h(k,i)=(h(k)+c1i+c2i2) \bmod N}$ con c1, c2≠0 costanti
double hashing: $\mathbf{c(k,i) = [h1(k) + i \cdot h2(k)] \bmod m}$ per $0 \le i < m$, e con h1 e h2 funzioni hash

FINE

MAP

L' ADT map è un contenitore che memorizza oggetti (detti entrate) del tipo chiave-valore (k,v)
Le chiavi sono uniche, inserimenti di chiavi già esistenti non sono permessi; le chiavi rappresentano il mezzo per reperire i valori associati, ma servono pure a rendere efficiente la ricerca.
Due sono le tecniche implementative:

1. ***Mediante lista doppiamente linkata delle posizioni:***

PositionList<Entry<K,V>> entries = new NodePositionList<Entry<K,V>>()
la ricerca di una chiave K si fa così:

```
public Iterable<K> keys() {
// crea una lista di chiavi del tipo dell'interfaccia PositionList e doppiamente linkata(NodePositionList)
 PositionList<K> keys = new NodePositionList<K>();
//  si dichiara iterabile la lista principale di nome “entries”
 Iterable<Entry<K,V>> entries = this.entries();
// con un ciclo di for each si scandisce la lista e si caricano solo le chiavi K nella lista iterabile “keys”.
 for(Entry<K,V> p: entries){
   keys.addLast(p.getKey());
  }
     return keys;
}
```

Complessità:

- put
  richiede il tempo necessario per verificare che la chiave non sia già nella mappa (***O***(***n***) nel caso pessimo) (qui 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 si scandisce l'intera lista)

L'implementazione mediante lista non ordinata è conveniente solo per mappe di piccola taglia o mappe in cui le operazioni più frequenti sono le *put*, e dove invece, ricerche e rimozioni sono più rare (per esempio, record delle login a una workstation).

2. ***mediante tabella hash***

Uno dei modi più efficienti di implementare una Mappa è di usare una **tabella hash**
una tabella hash per un dato tipo di chiave è composta da:
- un array (chiamato *bucket array (si pronuncia: bocket)* = array di contenitori o di secchi)
- una funzione hash **h**

Qui ogni cella dell'array è un contenitore di coppie (k,v)
L'idea iniziale era quella di creare un array di dimensione pari al valore della chiave più grande, ma ciò comportava un notevole spreco di memoria: per esempio:
nel caso di studenti si può pensare di memorizzare i dati di uno studente di indice pari al suo numero di matricola, ma questi non sempre partono da zero! Quindi memoria sprecata.
Inoltre c'è da dire che non sempre le chiavi sono di tipo intero(possono essere infatti alfanumeriche, ossia di tipo string).

La funzione hash (hash = tritare)

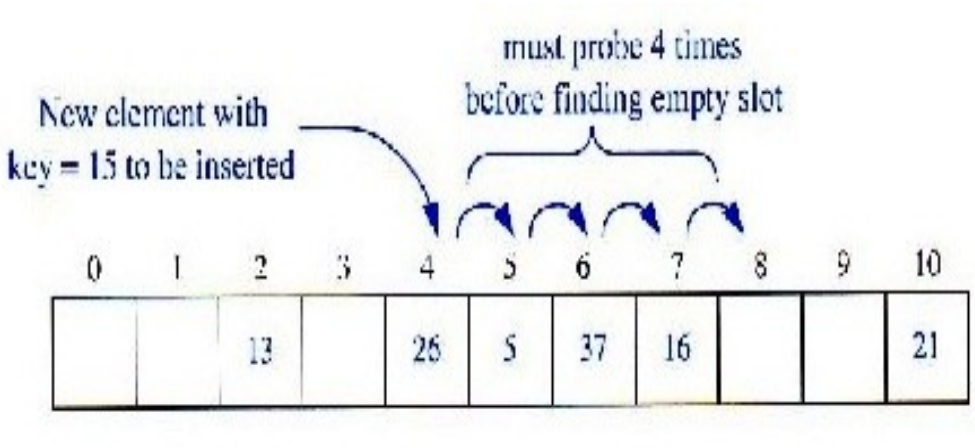
Lo scopo di una funzione hash è quello di distribuire le chiavi uniformemente nell'intervallo [0. N-1].
la funzione hash è rappresentata da una sorta di algoritmo di trasformazione che, fissato un intervallo, associa chiavi di qualsiasi tipo in un intero incluso in quell'intervallo,
per esempio: **h(x) = x mod N** che è un esempio di funzione hash per numeri interi compresi nell'intervallo [0, **N**-1] e dove **h(x)** è il valore hash della chiave **x;**
la trasformazione avviene in due fasi:
1. *hash code(codifica)*: trasformazione di chiavi di tipo qualsiasi in numeri interi
2. *compressione*: riporta l'intero ad un valore dell'intervallo scelto[0,N-1]

ora con lo stesso sistema esposto sopra, memorizziamo l'entrata (la coppia (k,v)) all'indice i = h(x) dell'array. Però può sorgere un problema: nessuna funzione hash è perfetta, per cui si potrebbe verificare che su chiavi diverse la funzione calcoli valori hash uguali, per cui nella fase di inserimento di un elemento, si trovi la posizione-indice occupata, tale particolare evento si dice **collisione**.
La f.hash implementata da java ha un problema: per due chiavi uguali può generare hash code diversi, mentre sappiamo che una funzione hash per chiavi uguali deve generare lo stesso hash code; questo succede perchè in taluni casi usa l'indirizzo di memoria dell'oggetto da trasformare in intero e quindi capita spesso che oggetti uguali memorizzati in locazioni differenti producano un hash code differente.

Trattamento delle collisioni

Separate Chaining (catene separate)
tale tecnica prevede che ogni cella dell'array sia un rif. ad una lista che contiene tutte le entrate di coppie (k,v) che hanno lo stesso h(k) = i , cioè i valori di hash uguali.
Una lista può essere anche un log file che altro non è che un semplice file di testo con estensione .log

Linear Probing (esplorazione lineare) detto anche metodo *open addressing*
questa tecnica invece funziona così: se si verifica una collisione, si cerca agli indici di posto successivi l'esistenza di una postazione libera e, appena trovata, si deposita la coppia (k,v)
esempio grafico: con funzione hash => **h(x) = x mod N** (x = elem. da inserire ed N dim Array = 11)
nell'array abbiamo già inserito: 13,26,5,37,16,21 (che hanno già avuto collisioni) ed ora inseriamo 15



C'è una complicazione in questa tecnica:
riguarda la rimozione di un elemento dall'array memorizzato a seguito di una collisione;
infatti, perchè tutto funzioni, bisogna eliminare l'elemento scalando a ritroso anche e soltanto tutti gli elementi che hanno avuto la stessa collisione (quelli per i quali la funzione hash ha assegnato la stessa posizione) e dal punto di vista implementativo la cosa è molto sconveniente in termini di efficienza;
Inoltre quando si va in ricerca l'algoritmo si arresterebbe se trovasse una cella vuota, intendendo aver controllato tutte le entrate con chiave k e questo è chiaramente sbagliato. Allora cosa si fa?
Si risolverebbe inserendo delle sentinelle chiamate “AVAILABLE” nelle posizioni da cui è stato eliminato un elemento, così quando si ripassa in ricerca su queste celle per inserimento o per lettura, tali celle contrassegnate vengono semplicemente ignorate;
tuttavia, c'è da dire, che questa non rappresenta un soluzione ottimale, l'array diventerebbe, presto pieno di celle non più utilizzabili, rallentando le relative operazioni.

Load factor (fattore di carico)
E' chiaro che la probabilità di avere collisioni fra chiavi, dipende da tre fattori:
1. dalla grandezza **N** dell' array
2. dal numeri di chiavi **x** già presenti nell'array
3. dalla bontà della funzione hash: h(k)

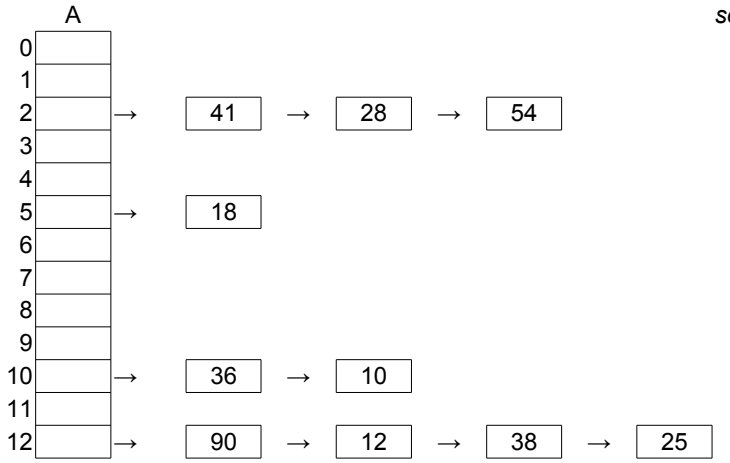
Ora, a prescindere, dal terzo fattore (la bontà del calcolo della funzione hash) bisogna stabilire un controllo sugli altri due fattori: x ed N, o meglio, bisogna tenere fissato entro un certo valore il relativo rapporto, così da mantenere sempre la stessa efficienza dal punto di vista delle collisioni.

λ = x / N <= ***load factor***

ossia il rapporto tra numero di chiavi presenti e dimensione del bucket array .
Si può dimostrare che conviene mantenere tale rapporto:

λ < **0.5** nel linear probing
λ < **0.9** nel separate chaining

separate chaining: il modello grafico



rehashing

Per mantenere λ (load factor) al di sotto di una certa soglia, si effettua il rehashing delle entrate di una tabella in una tabella più grande (creandola in genere di dimensione doppia). Il rehashing consiste nel ricalcolare gli hash code della tabella per riadattarli alla nuova tabella.

Modi diversi di computare l'hash code: (= calcolo di un codice hash)

Cast ad intero: se le chiavi k sono del tipo la cui rappresentazione binaria sia al più 32 bit, come ad esempio i tipi: byte, short, char si può fare direttamente un cast ad intero.
se la rappresentazione binaria superasse i 32 bit si avrebbe, nel cast, perdita di informazione perchè il cast taglierebbe a 32 bit (peraltro prenderebbe i meno significativi) aumentando la probailità di hash uguali.

Somma delle parti: il numero binario corrispondente della chiave viene suddiviso in **m** segmenti ognuno dei quali tradotto in intero, la somma di questi interi ci dà hash code.
il problema di questo metodo è dato dal fatto che se le parti prese singolarmente sono uguali a quelle di un un'altra chiave, si diversa, ma con le parti singole permutate in ordine differente, la cui somma però è sempre uguale.

Hash code polinomiale:
Il polinomio p(a) può essere valutato in tempo O(n) usando la regola di Horner.

La funzione di compressione utilizzata dalla funzione hash

dato: **i** = hash code di k; e **N** = capacità bucket array
Una buona funzione di compressione dovrebbe garantire una probabilità 1/N che due chiavi collidano

Metodo della divisione:
| i | mod N , collisioni meno probabili se N è un primo
genera molte collisioni se molte chiavi hanno un hash code della forma pN+q per diversi valori di p

Metodo MAD (multiply, add and divide):
|ai+b| mod N, dove a e b sono costanti intere scelte in modo casuale tali che a>0, b ≥ 0, a mod N ≠0 e dove N è primo.

Il concetto di posizione
La lista di nodi (node list) è un contenitore di oggetti che:
memorizza ciascun elemento in una **`posizione`**
mantiene le posizioni degli oggetti 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 essa associato.
Attenzione però: la position è un concetto immutabile,
Cioè se lo paragoniamo ad un indice avremo che mentre l'indice può cambiare il suo valore in caso di inserimenti o cancellazioni, la position invece, in caso di modifiche della lista, conserva sempre quello stesso nome identificativo.
`Morale:` continueremo a riferirci allo stesso elemento con la stessa posizione
La posizione definisce essa stessa un tipo astratto di dati che supporta il
seguente unico metodo: element(): che restituisce l'elemento in *questa* posizione

FINE

DICTIONARY

L'ADT Dictionary ha le stesse proprietà dell' ADT Map, con una sola differenza:
nel Dizionario le chiavi possono anche non essere uniche
cioè esistono chiavi di ugual valore.

l'interfaccia Dictionary:

```
public interface Dictionary<K,V> {
public int size();
public boolean isEmpty();
public Entry<K,V> find(K key) throws InvalidKeyException;
public Iterable<Entry<K,V>> findAll(K key) throws InvalidKeyException;
public Entry<K,V> insert(K key, V value) throws InvalidKeyException;
public Entry<K,V> remove(Entry<K,V> e) throws InvalidEntryException;
public Iterable<Entry<K,V>> entries();
}
```

Esistono dizionari ordinati e non ordinati

dizionari ordinati implementati con un arrayList:
usano un comparatore per definire una relazione di ordine totale sulle chiavi ed utilizzano un arrayList per ospitare le entrate (k,v).
il metodo find() richede un tempo O(log n) perchè utilizza l'algoritmo di ricerca binaria
mentre i metodi: insert() e remove() richiedono un misero O(n) a causa dello spostamento degli elementi .

dizionari non ordinati (detti anche **log file**) implementati con le liste:
sono implementati attraverso l'uso delle liste, ovviamente senza comparatore.
qui il metodo di inserimento addFirst() oppure addLast() richiedono tempo costante: O(1)
ma quelli che richedono una ricerca, come find() e remove() che devono necessariamente cercare l'elemento per agire su di esso, richiedono un tempo lineare: O(n)

entrambi i tipi di implementazione, però, si prestano bene solo a dizionari di piccole dimensioni, o che necessitano maggiormente delle operazioni 'convenienti' in termini di complessità.

Un'implementazione efficiente dei dizionari si ha attraverso l'uso di tabelle hash.
qui con una buona funzione di hash e un buon controllo del load factor, si possono avere efficienze dell'ordine di O(1), e laddove si verifichino agglomerati di celle prodotti da collisioni, si fa ricorso all' **hashing doppio** che riduce questo effetto negativo di lunghi gruppi di celle consecutive occupate che rallentano la scansione;
di seguito la formula dell' hashing doppio (double hashing):-

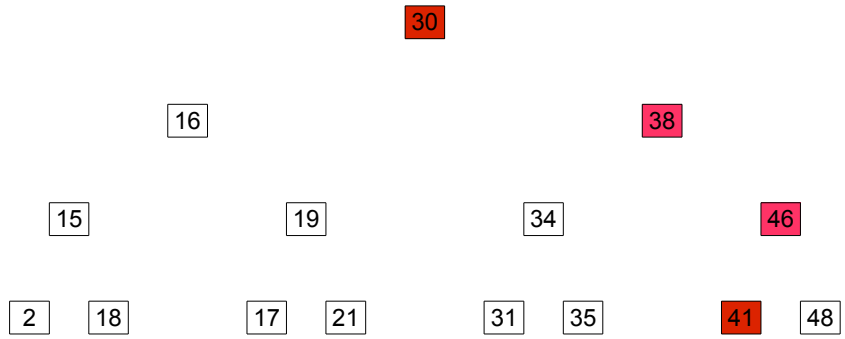
$$c(k,i) = [h1(k) + i \cdot h2(k)] \bmod m$$

per $0 \le i < m$, e con h1 e h2 che sono funzioni hash

*un dizionario si può implementare anche con l'uso di un BST (**B**inary**S**earch**T**ree):*
In un Binary Search Tree, utilizzato per implementare un Dizionario,
tutti i valori del sotto-albero sinistro di un nodo *v* sono minori o uguali del valore di *v*
tutti i valori del sotto-albero destro di un nodo *v* sono maggiori o uguali del valore di *v*
e questo vale per tutti i nodi.
nella figura sopra ho omesso le foglie che in un BST sono senza valore, ma ci devono essere

RICERCA DI UN NODO:
così se per esempio voglio cercare il nodo 41, allora partendo dalla radice lo confronto con essa e scelgo di procedere a destra, in quanto è risultato essere 41 > 30 e siccome i valori >30 sono tutti alla destra dello stesso 30 la scelta è ovvia.
Scendo a destra nel confronto 41>38 per lo stesso motivo,
ed in ultimo a sinistra nel confronto 41< 46 .
e qui trovo il valore cercato 41.

di seguito un esempio illustrato:



FINE

SEQUENCE

Il TDA Sequence si usa per rappresentare un insieme di elementi disposti secondo un ordine lineare.
I TDA utilizzati per rappresentarlo sono: NodeList e ArrayList
Qui un elemento è individuabile sia attraverso l'indice che attraverso la sua posizione, all'uopo esistono due metodi detti, appunto, metodi-ponte: atIndex (i) e indexOf (p) che permettono di passare da un modo all'altro, cioè da indice a posizione e viceversa.
Oltre ai metodi dell'interfaccia NodeList ci sono anche i metodi:
getFirst(), getLast(), removeFirst(), removeLast()

Due sono i metodi di implementazione del TDA Sequence implementato con lista DL:NodePositionList
I metodi basati sulle posizioni che richiedono tempo costante
I metodi basati sugli indici che richiedono una ricerca dal nodo header al trailer o viceversa e quindi hanno bisogno di tempo lineare con Array di Posizioni usato in modo circolare e dove l'oggetto Position memorizza due dati: indice (è quello dell'array) ed elemento.

FINE

SET & PARTITION

Il TDA **Set**.

Iniziamo dal concetto di insieme, in inglese **Set**, è esso un TDA rappresentato da una sorta di contenitore in cui gli elementi sono a due a due distinti e sui quali generalmente, non è definita alcuna relazione d'ordine. (slide n.443 di De Marco)

Tre sono i metodi principali, dai cui nomi è facile intuirne il significato matematico:

public Set<E> union(Set<E> B)

invocato sull'insieme A, rimpiazza A con l'unione di A e B A.union(B)

gli elementi contenuti A ed in B presi una volta sola.

public Set<E> intersect(Set<E> B)

invocato sull'insieme A, rimpiazza A con l'intersezione di A e B A.intersect(B)

gli elementi contenuti sia in A e sia in B

public Set<E> subtract(Set<E> B)

invocato sull'insieme A, rimpiazza A con la differenza di A e B A.subtract(B)

gli elementi di A non contenuti in B

i soliti metodi size() e isEmpty completano i metodi dell'interfaccia.

Implementazione di Set con una sequenza ordinata

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

Cioè gli elementi dell'insieme vengono memorizzati in una sequenza ordinata;

il vantaggio di questo ordinamento consiste nel rendere possibile l'uso dello schema della procedura merge (usata nel merge sort) per implementare i metodi union, intersect e subtract.

La classe che implementa Set avrà:

- una var. di istanza del tipo usato per rappresentare la sequenza (IndexList, PositionList, Sequence)
- una variabile di istanza di tipo Comparator usata per confrontare gli elementi dell'insieme.

Algoritmo Generico di Merge di due liste (in pseudocodice).

```
Algorithm genericMerge(A, B)
  S ← empty sequence
  while !(A.isEmpty()) ∧ !(B.isEmpty()) {
       a ← A.first().element();
       b ← B.first().element()
       if a < b
         aIsLess(a, S); A.remove(A.first())
       else if b < a
         bIsLess(b, S); B.remove(B.first())
       else // se b = a
         bothAreEqual(a, b, S)
         A.remove(A.first()); B.remove(B.first())
   }
  while !(A.isEmpty())
       aIsLess(a, S); A.remove(A.first())
  while !(B.isEmpty())
       bIsLess(b, S); B.remove(B.first())
return S
```

considerato n_A : il numero di elementi in A e n_B : il numero di elementi in B

considerando i tempi O(1) dei metodi ausiliari, il tempo sarà $O(n_A + n_B)$

per *union* vanno riscritti tutti e tre i metodi: **aIsLess, bIsLess, bothAreEqual**
cioè per ognuno dei tre casi si deve inserire un elemento nell'insieme-unione S.
per *intersection* va riscritto il metodo ausiliario **bothAreEqual**
qui si inserisce l'elemento si S solo nel caso di a == b.
per *subtract* va riscritto **aIsLess** (*aisLess* vuol dire *a è minore*)
*qui invece si inserisce in S solo l'elemento **a*** che risulta minore o maggiore di elementi **b.**

aIsLess(a,S) : aggiunge l'elemento a alla sequenza (insieme) S
bIsLess(b,S) : aggiunge l'elemento b alla sequenza (insieme) S
bothAreEqual(a,b,S) : aggiunge l'elemento **a** e non il suo duplicato **b** alla sequenza (insieme) S
una ottima illustrazione grafica del funzionamento del merge generico inizia a slide n.452 di DeMarco

Il TDA **Partition**
Il TDA Partition è, a sua volta, una collezione di insiemi a due a due disgiunti, esso supporta i metodi:
1. **makeSet(x)** crea l'insieme contenente il solo elemento x e lo aggiunge alla partizione
2. **union(A, B)** aggiunge alla partizione l'insieme unione di A e B distruggendo gli insiemi A e B
3. **find(x)** restituisce l'insieme che contiene l'elemento x

i soliti metodi size() e isEmpty completano i metodi dell'interfaccia.

Ora, mentre prima, gli elementi di confronto erano gli elementi degli insiemi,qui gli elementi di confronto sono gli insiemi stessi e cioè oggetti di tipo Set implementati sempre attraverso l'uso dei tipi:- (PositionList, IndexList, Sequence)
una implementazione efficiente può essere quella che fà uso delle mappe, rappresentate queste da una tabella hash, che associa quell'elemento al relativo insieme, in cui la coppia è formata da (elemento, insieme) e dove insieme è rappresentato da una variabile di istanza di tipo Set.
Find ha tempo medio O(1) se si usa una buona funzione hash e si tiene il load factor sotto controllo
Partition può essere implementata **con una lista** (lista i insiemi) **ListPartition** oppure implementata **con gli alberi** dove la radice individua l'insieme e i restanti nodi gli elementi dell'insieme;
qui la **Union** si ottiene facendo puntare la radice di un albero alla radice dell'altro,
e la **Find** si realizza seguendo il percorso dal nodo che contiene l'elemento fino al nodo radice.

FINE

Grafi

I grafi sono una particolare struttura dati usata spesso in informatica
Esempio più noto di grafo: World Wide Web
gli algoritmi che lavorano su grafi sono fondamentali in molti campi dell'informatica
Esempio: i metodi di routing per l'instradamento dei pacchetti su Internet hanno bisogno di conoscere
i cammini più brevi per andare da un nodo all'altro della rete.
Un grafo G=(V,E) è una coppia di insiemi
V = insieme di nodi, chiamati vertici
E = insieme di coppie di nodi, chiamati archi
per esempio:
V = {1,2,3,4,5}
E = {(1,2),(2,3),(2,4),(3,4),(4,5),(5,1)}

Dal punto di vista delle strutture dati è un contenitore di elementi memorizzati
nelle posizioni del grafo: i vertici V e gli archi E
un grafo può essere orientato e non, a seconda che i suoi archi abbiano o no un verso (arco = freccia)

metodi di un grafo non orientato:

metodi iteratore:
– **vertices()**: restituisce una collezione iterabile su tutti i vertici del grafo
– **edges()**: restituisce una collezione iterabile su tutti gli archi del grafo
– **incidentEdges(v)**: restituisce una collezione iterabile sugli archi incidenti su **v**

Metodi di accesso
– **endVertices(e)**: restituisce un array dei due vertici estremità dell'arco **e**
– **opposite(v, e)**: restituisce il vertice opposto a **v** sull'arco **e**
– **areAdjacent(v, w)**: vero se e solo se **v** e **w** sono adiacenti
– replace(v, x): rimpiazza l'elemento memorizzato nel vertice **v** con **x**
– **replace(e, x)**: rimpiazza l'elemento memorizzato nell'arco **e** con **x**

Metodi di aggiornamento
– **insertVertex(o)**: inserisce un vertice che contiene l'elemento o
– **insertEdge(v, w, o)**: inserisce un arco (v,w) che contiene l'elemento o
– **removeVertex(v)**: rimuove il vertice v (ed i suoi archi incidenti)
– **removeEdge(e)**: rimuove l'arco *e*

Terminologia
Estremità di un arco: vertici collegati dall'arco u e v
Archi incidenti su un vertice u: archi con una delle due estremità uguali a u
Vertici adiacenti: vertici che sono le estremità di uno stesso arco
Grado di un vertice: numero di archi che incidono sul vertice
Autociclo: arco che ha entrambe le estremità uguali ad uno stesso vertice
Percorso: sequenza che comincia e finisce con un vertice e in cui si alternano vertici ed archi
ciascun arco è situato tra le sue estremità
Percorso semplice: percorso in cui tutti i vertici e tutti gli archi sono distinti
ciclo: percorso che inizia e finisce nello stesso vertice
ciclo semplice: ciclo in cui tutti i vertici sono distinti ad eccezione del primo e dell'ultimo

Il **decorator pattern** è un design pattern che serve ad inserire informazioni extra (colori) ad un oggetto
Una decorazione consiste di:
- una chiave (che identifica il tipo di decorazione: colore, booleano true/false, ect.)
- un valore associato alla chiave (bianco, true, ect.)
esempio:
algoritmi come BFS e DFS richiedono che i nodi e/o gli archi del grafo vengano colorati per tenere traccia dell'avvenuta o meno esplorazione;
Si può usare un decorator pattern per “decorare” i nodi del grafo:
ad ogni nodo sarà associata una coppia (chiave, valore) dove la chiave identifica l'attributo esplorato e il valore dove la chiave identifica l'attributo esplorato e il valore (ad essa associato) è un booleano;
esplorato è l'attributo (chiave)
true o false è il valore
in pratica memorizza l'informazione: esplorato si - esplorato no
un altro esempio può essere: chiave = colore e attributo = white

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 a x
remove(a): rimuove dalla posizione l'attributo a e il suo valore

allora ne definisco l'interfaccia estendendo Position e Map
public interface DecorablePosition<T> extends Position<T>, Map<Object,Object> { }
e poi definisco le interfacce Vertex ed Edge estendendo in esse l'interfaccia DecorablePosition
public interface Vertex<V> extends DecorablePosition<V> { }
public interface Edge<E> extends DecorablePosition<E> { }

su un oggetto Vertex<V> si possono usare i metodi della mappa per gestire le colorazioni, per es:-
final Object COLOR = new Object();
final Object WHITE = new Object();
v.put(COLOR, WHITE); // dove COLOR è la chiave e WHITE è il valore
if(v.get(COLOR) != WHITE)
Ciascun vertice del grafo è rappresentato da un oggetto di tipo *vertex*, e tutti gli oggetti vertex sono memorizzati in un contenitore V (lista o vettore (ArrayList))
Ciascun arco del grafo è rappresentato da un oggetto di tipo *edge*, e tutti gli oggetti edge sono memorizzati in un contenitore E (una seconda lista o vettore (ArrayList))
L'oggetto *vertex* immagazzina due informazioni:
- l'elemento *o* (quindi un reference ad esso)
- un reference alla propria posizione all'interno del contenitore.
L'oggetto e*dge* (per un arco *e = (u,v)*) immagazzina tre informazioni:
- un riferimento a *o*
- un riferimento agli oggetti-vertex per *u* e *v*
- un riferimento alla posizione dell'oggetto edge nel contenitore E degli archi del grafo

Vantaggi:
- implementazione semplice
- accesso diretto dagli archi ai vertici su cui essi sono incidenti
  (endVertices(e) e opposite(v,e) sono semplici da implementare e richiedono tempo O(1))

Svantaggi:
- l'accesso agli archi che sono incidenti su un dato vertice richiede un'ispezione esaustiva di tutto il contenitore E. I seguenti metodi richiedono un tempo proporzionale al numero di archi del grafo:
- incidentEdges(v)
- areAdjacent(v,w)
- removeVertex(v)

domanda: qual è il massimo numero di archi che un grafo G = (V,E) di n = |V| vertici può avere?
risposta: nei grafi orientati n^2 e in quelli non orientati (n su 2)

Implementazione dei grafi mediante liste di adiacenza o matrici di adiacenza

rappresentazione di un grafo in memoria: grafo G=(V,E)

con **liste di adiacenza** **oppure** con **matrice di adiacenza** slide 543 di DeMarco

liste di adiacenza (per grafi non orientati): AdjacencyListGraph

un array di **|V|** liste, cioè tante liste quanti sono i vertici
per ogni **u** in V la lista **Adj[u]** contiene tutti i vertici **v** *adiacenti* ad **u** nel grafo

liste di adiacenza (per grafi orientati):

un array di **|V|** liste, cioè tante liste quanti sono i vertici
per ogni **u** in V la lista **Adj[u]** contiene tutti i vertici **v** *uscenti* da **u** nel grafo
Spazio di memoria richiesto in entrambi i casi: $\Theta(\max\{|V|,|E|\}) = \Theta(|V|+|E|)$

Grafi pesati e non pesati

Sono quei grafi in cui l'elemento degli archi rappresenta il peso dello stesso arco;
nei grafi non pesati il valore di tale elemento è null.
Il peso **w(u.v)** di ogni arco **(u,v)** diviene un nuovo campo aggiunto nelle liste di **u**
qui per fortuna lo spazio di memoria richiesto è: $\Theta(|V|+|E|)$
però non c'è un modo veloce di riderca di un arco nel grafo, la ricerca è sequenziale in G

Matrice di adiacenza

Un grafo G = (V,E) può essere rappresentato mediante una matrice di adiacenza:
V = {0,1, ... , n – 1}
E = { (i,j) | i ∈ V, j ∈ V }
matrice VxV
La rappr.consiste di una matrice A di dimensione |V|x|V| tale che per l'elemento a_{ii}
in posizione i,j si ha:
a_{ij}=1 se (i.j) ∈ E
a_{ij}= 0 altrimenti
una buona illustrazione alle slide di DeMarco n. 572
nei grafi non orientati la matrice di adiacenza risulta essere simmetrica

Matrice di adiacenza: Vantaggi e svantaggi
Buone notizie
Per determinare se un dato arco (i,j) è presente nel grafo, è sufficiente controllare il valore di a_{ij}
tempo costante

cattive notizie

Spazio di memoria richiesto: $\Theta(|V|^2)$
in caso di grafi sparsi si ha uno spreco di memoria

in definitiva:
liste di adiacenza: è conveniente quando il grafo G è sparso, (|E| molto piu` piccolo di $|V|^2$)

matrice di adiacenza: è conveniente quando il grafo G è denso, (|E| quasi uguale a $|V|^2$)

Metodi di visita dei grafi: BFS, DFS

BFS: Breadth-first-search (visita in ampiezza, cioè visita prima tuti i nodi a distanza minima)
DFS: Depth-first search (visita in profondità, esplora in profondità in grafo G finché è possibile)

Dijkstra

L'algoritmo di Dijkstra è utilizzato per cercare i cammini minimi (o Shortest Paths, SP) in un grafo
Si ottiene il percorso minimo tra un punto di partenza e tutti gli altri punti del grafo.
i valoi dei nodi vengono aggiornati mediante la tecnica del “rilassamento”
l'uso del TDA AdaptablePriorityQueue è giustificato dal fatto che in questo algoritmo sono previste modifiche ad elementi e chiavi, e AdaptablePriorityQueue contiene i metodi giusti per tali modifiche.
L’analisi considera due diverse implementazioni della coda Q:
Mediante un array : Tempo totale: $O(V^2 + E)$ = **$O(V^2)$**
Mediante binary heap : Tempo totale: O(Vlog V + E log V)

nella cartella: 'Strutture Dati Algoritmo di Dijkstra' all'interno di 'SD domande di esame orale' c'è una pagina html che illustra bene il funzionamento di Dijkstra.

Prim

Per la descrizione dell'algoritmo si assume il grafo sia rappresentato utilizzando una struttura dati detta lista delle adiacenze che è solitamente realizzata con un array cui ogni posizione corrisponde un vertice. Ogni elemento dell'array punta ad una generica lista concatenata che conterrà tutti i vertici adiacenti al vertice considerato.Inoltre si assume che ogni vertice v abbia i campi dato chiave[v] e π[v], rispettivamente il valore associato al vertice e il puntatore al padre di quest'ultimo all'interno dell'MST.

Anche qui si usa il TDA AdaptablePriorityQueue perchè anche in questo algoritmo(come in Dijkstra) sono previste modifiche ad elementi e chiavi; AdaptablePriorityQueue contiene i metodi giusti per tali modifiche.

Inizialmente si pongono tutti i campi chiave[v] a +∞ e tutti i campi π[v] a NIL.
Si prende un vertice qualsiasi come radice dell'albero e si pone la sua chiave a 0.
Si inseriscono tutti i vertici rimasti in una struttura dati appropriata
(tipicamente una coda di priorità) e li si estrae in ordine crescente.

Si scorre quindi la lista delle adiacenze del vertice estratto (u) considerando solo i vertici (v) ancora all'interno della struttura ausiliaria.
Per ognuno di essi tale che la sua distanza da u sia la minore tra tutti quelli considerati, si pone π[v] uguale ad u inserendo di fatto v nell'MST.
Si conclude il ciclo aggiornando il campo chiave[v] con il valore della distanza tra u e v.
L'utilizzo di una struttura dati ausiliaria è auspicabile per evitare di dover controllare continuamente se il vertice considerato in uno dei passi dell'algoritmo sia già stato visto precedentemente.
In altre parole, dato un grafo G=(V,E) (V è l'insieme dei vertici o nodi, E è l'insieme degli archi) ed un

albero di soluzione S in cui porremo i nodi raggiunti nei vari passi dell'algoritmo procediamo nel seguente modo: pongo in S un nodo di partenza (arbitrario) dal quale poi sceglierò l'arco incidente di peso minimo non collegato a nodi facenti parte dell'albero di soluzione S. Effettuata la scelta del ramo dal passo precedente includerò in S il vertice collegato al vertice di partenza dal ramo appena scelto. Ad ogni vertice che includo in S anche i rami incidenti di quel vertice si aggiungeranno ai rami tra cui sceglierò quello meno costoso che collega ad un vertice non appartenente ad S. L'algoritmo termina quando la cardinalità di S è pari a quella di G.

complessità: Quindi il tempo totale di esecuzione dell'algoritmo di Prim è : O(E * log V) se si utilizza una coda a priorità. (E = insieme degli archi; V = inseme dei nodi)

Kruskal

premesso che: *l'albero ricoprente è il grafo che connette tutti i nodi del grafo senza creare cicli*
Lo scopo dell'algoritmo è quello di trovare un albero ricoprente di peso minimo,
cioè quello in cui la somma dei pesi sia minima.
L'algoritmo di Kruskal si basa sulla seguente semplice idea:
ordiniamo gli archi per costi e poi li esaminiamo in questo ordine e li inseriamo man mano nella soluzione che stiamo costruendo, se non formiamo cicli con archi precedentemente selezionati.
Notiamo che ad ogni passo, se abbiamo più archi con lo stesso costo, è indifferente quale viene scelto.

Implementazione dell'algoritmo:
Una possibile implementazione dell'algoritmo di Kruskal potrebbe essere la seguente:
- Creare la foresta di grafi.
- Ordinare tutti gli archi del grafo in ordine crescente.
- Scandire gli archi ordinati, uno per volta, inserendoli se opportuno nella foresta
- L'arco per essere aggiunto alla foresta deve collegare due alberi disgiunti
- L'arco deve essere sicuro
- L'arco non deve generare cicli, sempre vero se l'arco unisce due alberi disgiunti.

L'implementazione che viene descritta utilizza il TDA Partition e il TDA PriorityQueue.
Il **TDA Partition** viene utilizzato per mantenere un insieme di oggetti disgiunti (insiemi disgiunti).
Il **TDA PriorityQueue** invece viene utilizzato per mantenere la lista degli archi ordinati dal più piccolo al più grande.

Kruskal-complessità: O(E log E)

FINE

Cammino Euleriano

- E` una visita generica di un albero binario
- Casi speciali: visita preorder, postorder e inorder
- Attraversa l'albero ispezionando ogni nodo tre volte:

- a sinistra: prima della visita del sottoalbero sinistro (preorder)
- da sotto: tra le visite ai due + sottoalberi (inorder)
- a destra: dopo la visita del sottoalbero destro (postorder)

EulerTour è chiamato ricorsivamente sul figlio sinistro e destro
Un oggetto TourResult con campi left, right e out tiene i risultati delle chiamate ricorsive a EulerTour

Specializzazione di EulerTour:
valutazione espressioni, cioè calcolarne il risultato, assumendo però che: -
- Nodi interni contengono gli operatori (+ - * :)
- Nodi esterni contengono gli operandi (le variabili che contengono i valori)

Il concetto alla base di un cammino euleriano è che, dato un albero, tutti i nodi vengano toccati da questo cammino per tre volte, la prima da sinistra (scendendo lungo l'albero), la seconda dal basso e la terza da destra (risalendo).

L'algoritmo eulerTour:

```
Algorithm eulerTour(T,v)
effettua visita di v a sinistra
if v ha un figlio sinistro u then
eulerTour(T,u)
effettua visita di v da sotto
if v ha un figlio destro w then
  eulerTour(T,w)
effettua visita di v a destra
```

La classe TourResult:

```
/* I campi left e right servono a tenere traccia dei risultati della chiamate di eulerTour sul figlio
  sinistro e sul figlio destro di v. Il campo out tiene traccia del valore computato
  dalla chiamata di eulerTour su v
*/
public class TourResult<R> {
public R left;
public R right;
public R out;
}
```

FINE

Analisi ammortizzata

Nella costruzione degli algoritmi, per garantirne la bontà, se ne analizza, in genere, il comportamento nel caso peggiore. Tale stima però può non essere tanto realistica, ed ecco perchè:-

- Un algoritmo potrebbe avere tante operazioni poco costose, per es. con complessità O(1), ed averne una sola molto costosa, diciamo O(n); in un caso del genere, preso come caso limite, analizzare l'algoritmo nel caso peggiore ci darebbe una stima, non del tutto realistica ma per così dire troppo pessimistica.
  Allora analizziamo la sequenza tipica delle operazioni richieste dall'algoritmo e, **su tale sequenza, calcoliamo la media dei tempi richiesti**

Quindi, **IMPORTANTE:** non si considera la media, bensì la media sulla sequenza delle operazioni, e dove la **sequenza** comprende operazioni costose e meno costose.

Si considera il tempo richiesto per eseguire, nel caso pessimo, un'intera sequenza di operazioni
Se operazioni più costose sono poco frequenti, allora il loro costo può essere ammortizzato dalle operazioni meno costose.
Per esempio:
Supponiamo di avere una sequenza di σ operazioni su una particolare struttura dati:

• La sequenza richiede tempo totale O(σ t) per essere eseguita
• Diremo che il tempo ammortizzato per ogni operazione della sequenza è O(t)
• Media sulla sequenza: O(σ t)/σ = O(t)

• Nota: anche se una singola operazione potrebbe richiedere più di O(t) nel caso peggiore!

è come se noi volessimo diluire i costi maggiori di alcune operazioni su altre operazioni, più frequenti, e sulla sequenza di tali operazioni eseguire una media.
Il risultato di una tale analisi ci consentirà una valutazione più realistica dell' algoritmo.
E' chiaro che se nella sequenza considerata, l'operazione dal costo più eleveto compare spesso, in tal caso l'analisi ammortizzata tenderà ad avvicinarsi all'analisi nel caso peggiore.

FINE

Merge Sort ricorsivo

L'algoritmo di Merge Sort ricorsivo

data una sequenza di numeri:

(**10** **3** **15** **2** **1** **4** **9** **0**)

si divide la sequenza in due parti uguali, se dispari allora la prima parte contiene l'elemento in più

(**10** **3** **15** **2**) (**1** **4** **9** **0**)

si divide ancora (ricorsione) in parti uguali

(**10** **3**) (**15** **2**) (**1** **4**) (**9** **0**)

e ancora fino all'unico elemento

(**10**) (**3**) (**15**) (**2**) (**1**) (**4**) (**9**) (**0**)

si prende, per ogni coppia di elementi in parentesi quello minimo e si scambia di posto se necessario

(**3** **10**) (**2** **15**) (**1** **4**) (**0** **9**)

ancora si prende, per ogni coppia di elementi in parentesi quello minimo e si scambia di posto se necessario

(**2** **3** **10** **15**) (**0** **1** **4** **9**)

ancora si prende, per ogni gruppo di elementi in parentesi quello minimo e si inserisce nell'array finale ordinato

(**0** **1** **2** **3** **9** **10** **15**)

complessità: O(n)