Raccolta temi d'esame del corso di Laboratorio di Algoritmi e Strutture Dati

Roberto Posenato

27 gennaio 2010

Parte I

Anno accademico 2000/2001

1 Appello del 18/12/2000

Esercizio 1.1. Data un'implementazione (con liste ordinate) dell'ADT Set, scrivere una procedura che, dati due oggetti Set, restituisce un oggetto che è l'unione dei due oggetti.

La procedura deve essere scritta in Java.

Suggerimento: l'implementazione offre il metodo `public Iterator iterator()`.

2 Appello del 15/01/2001

Esercizio 2.1.

Prima parte

Data un'implementazione a liste dell'ADT Set, scrivere una procedura esterna che, dati due oggetti Set, restituisce un oggetto Set che è l'intersezione dei due oggetti in input.

Seconda parte

Presentare la stessa procedura nel caso in cui l'implementazione è a liste ordinate.

3 Appello del 15/03/2001

Esercizio 3.1.

Si consideri il problema della massima sotto sequenza comune di due stringhe e si supponga di avere disponibile il metodo (realizzato con una tecnica di programmazione dinamica) `static int[][] maxLunghezza(String x, String y)` che restituisce la matrice delle lunghezze delle sotto sequenze massime per tutti i sotto problemi (in `c[x.length()][y.length()]` c'è la lunghezza della sotto sequenza massima cercata).

Scrivere una procedura `public static String maxSSC(String x, String y)` (in Java) che restituisce la stringa della massima sotto sequenza comune.

Parte II
Anno accademico 2001/2002

4 Appello del 19/03/2002

Esercizio 4.1.
Si vuole realizzare un'implementazione della struttura dati hash table a indirizzamento aperto con doppio hashing per gestire un insieme di dati costituito da coppie *(CodiceFiscale, Reddito)*, dove *CodiceFiscale* un oggetto di tipo `Stringa` e *Reddito* è un oggetto di tipo sconosciuto.

1. Si proponga una soluzione per il calcolo della posizione di inserimento di un elemento a partire dal *CodiceFiscale*. Si fornisca l'implementazione in Java della soluzione proposta.
2. Supponendo di voler mantenere un carico costante della tabella, si fornisca un algoritmo (e sua codifica in Java) per l'adeguamento della dimensione della tabella hash in caso di sovraccarico.

5 Appello del 17/06/2002

Esercizio 5.1.
Dato lo schema della seguente classe

```
public class Persona {
  private String          nome, cognome;
  private java.util.Date dataNascita;
  private int             reddito;
  ...
}
```

1. Si fornisca la codifica in Java opportunamente commentata del o dei metodi necessari per poter eseguire il confronto fra oggetti di questa classe e che implementi il criterio di confronto lessicografico sul meta attributo *nome+cognome*; come si deve modificare la dichiarazione di classe? (max 7/30)
2. Si fornisca la codifica di un algoritmo di ordinamento che dato un array di object di tipo *Persona* restituisca l'array ordinato secondo il criterio di ordinamento naturale della classe. (max 7/30)
3. Si fornisca la codifica in Java opportunamente commentata delle classi esterne necessarie per implementare i seguenti criteri di confronto: piùGiovane, minorReddito. (max 8/30)
4. Si fornisca la codifica dell'algoritmo ShellSort che dato un array di object di tipo *Persona* restituisca l'array ordinato secondo il criterio di ordinamento passato come secondo parametro. (max 8/30)

6 Appello del 15/07/2002

Esercizio 6.1.
Si vuole implementare una struttura dati per rappresentare dei grafi orientati pesati sfruttando anche la seguente classe

```
public class Arco {
  private Object nodoP, nodoA, peso;
  ...
}
```

Tale implementazione sarà rappresentata dalla classe `Grafo`.

1. Si arricchisca la classe `Arco` (con codice Java commentato) in modo che implementi l'interfaccia `Comparable`. Il criterio di confronto da realizzare è dato dal confronto dei pesi e, a parità di peso, dal confronto dell'attributo `nodoP` (che assunzioni si devono fare?). Dire come deve essere modificato il metodo `equals()`. (max 7/30)
2. Scrivere in linguaggio Java un'interfaccia che dichiari le operazioni fondamentali su un grafo, giustificando le scelte fatte. (max 7/30)
3. Scrivere in linguaggio Java una possibile implementazione dell'interfaccia del punto precedente, limitandosi alla specifica degli attributi (con giustificazioni) e al metodo `remove(Object x)` dove il parametro rappresenta un nodo. (max 8/30)

4. Scrivere in linguaggio Java, con le giustificazioni opportune, il metodo che restituisce l'insieme di tutti gli archi ordinati. (max 8/30)

7 Appello del 02/09/2002

Esercizio 7.1. Si presenti in linguaggio Java commentato un'implementazione dell'ADT 'albero' generico con radice che sia il più compatta possibile (da un punto di vista uso classi e attributi... no di descrizione o commenti...). Tale implementazione deve poter rappresentare come elementi associati ad un nodo qualsiasi oggetto. (max 7/30)

Esercizio 7.2. In base alla classe del punto 1, si scriva in linguaggio Java un metodo pubblico della classe che effettui la visita in post-ordine dell'albero applicando il metodo `visita(Object x)` ad ciascun nodo visitato. Il metodo `visita(Object x)` però non è della classe, ma di un oggetto passato come parametro. Come deve essere la dichiarazione di questo parametro per poter far si che si possa accettare 'qualsiasi' oggetto garantendo comunque che questo abbia il metodo `visita(Object x)`? (max 7/30)

Esercizio 7.3. Sempre in base alla classe del punto 1, si scriva un'implementazione di una classe che rappresenta un interatore (java.util.Iterator) su l'albero.

Esercizio 7.4. Si consideri il problema della massima sotto sequenza comune di due stringhe: date due stringhe X e Y, determinare la sotto sequenza comune Z di lunghezza massima.

È noto che, data una massima sotto sequenza comune Z, valgono le seguenti proprietà:

- $x_m = y_n \rightarrow z_k = x_m = y_n$ e Z_{k-1} è sotto sequenza massima di X_{m-1} e Y_{n-1}.
- $x_m \neq y_n \rightarrow$
  - o $z_k \neq x_m \rightarrow Z_k$ è sotto sequenza massima di X_{m-1} e Y_n
  - o $z_k \neq y_n \rightarrow Z_k$ è sotto sequenza massima di X_m e Y_{n-1}.

Si determini un algoritmo per la risoluzione del problema, specificando la tecnica di costruzione utilizzata, le strutture dati necessarie e una pseudo codifica dell'algoritmo stesso. L'algoritmo non deve ricorrere alla tecnica della ricerca esaustiva. (max 8/30)

Parte III
Anno accademico 2007/2008

8 Appello del 14/03/2008

Si consideri la seguente classe:

```
/** Rappresenta un messaggio generico. */
public class Messaggio {
  /** Nome del mittente */
  private String mittente;
  /** Data di invio */
  private java.util.Date dataInvio;
  /** Lunghezza del messaggio in char*/
  private int lunghezza;
  ...
}
```

Esercizio 8.1 (*max 7 punti*)**.** Scrivere la codifica Java (opportunamente commentata) del o dei metodi necessari per poter eseguire il confronto fra oggetti di questa classe che implementi il criterio di confronto lessicografico sul campo `mittente` usando solamente i seguenti metodi della classe `String`: `char charAt(i)`, che restituisce il carattere nella posizione data dall'intero i e `int length()` che restituisce la lunghezza della stringa.

Esercizio 8.2 (*max 7 punti*)**.** Scrivere la codifica Java (opportunamente commentata) delle classi esterne necessarie per implementare i seguenti criteri di confronto tra oggetti di tipo `Messaggio`: *PiuVecchio* e *PiuBreve*.

Per il campo `dataInvio` è ammesso usare `Date.compareTo()`.

Esercizio 8.3 (*max 8 punti*)**.** Scrivere la codifica Java (opportunamente commentata) di un algoritmo di ordinamento (differente da quello dell'esercizio 4) che dato un array di tipo `Messaggio` restituisca l'array ordinato secondo il criterio interno alla classe sopra implementato.

Esercizio 8.4 (*max 8 punti*)**.** 1. Si commenti il seguente algoritmo di ordinamento. Di quale algoritmo si tratta? (NB: La risposta corretta alla domanda non è considerata valida senza i commenti al codice)

```
public static short[] sort(short a[])
        throws IllegalArgumentException {
    int i = 0, max = 0;
    // verifico che 'a' soddisfi le condizioni
    for (i = 0; i < a.length; i++) {
        if (a[i] < 0)
          throw new IllegalArgumentException(
            "Il vettore a contiene un elemento negativo");
        if (max < a[i]) max = a[i];
    }

    short[] temp = new short[max + 1];

    for (i = 0; i < a.length; i++)
        temp[a[i]] += 1;

    for (i = 1; i <= max; i++)
        temp[i] += temp[i - 1];

    short[] out = new short[a.length];

    for (i = a.length - 1; i >= 0; i--) {
        out[temp[a[i]] - 1] = a[i];
        temp[a[i]] -= 1;
    }
    return out;
}
```

2. Modificare il codice dato (su un foglio a parte) in modo che ordini un array di object di tipo `Messaggio` secondo un criterio a vostra scelta di quelli proposti negli esercizi precedenti. Giustificare la scelta.

9 Appello del 01/04/2008

Esercizio 9.1 (*max 7 punti*)**.** Si vuole implementare l'Abstract Data Types *Insieme* che rappresenta il concetto matematico di insieme. Buona norma è fornire prima di tutto una specifica delle operazioni minime che permettano di definire e modificare oggetti di tipo *Insieme* rispetto agli elementi costituenti e le operazioni tipiche che operano sugli insiemi. Si scriva tale specifica in Java con commenti in Javadoc assumendo che un insieme possa contenere un elemento null al più e che non possa contenere sé stesso.
Nota: ogni commento Javadoc poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante costa 2 punti.

Esercizio 9.2 (*max 7 punti*)**.** Si assuma di avere a disposizione la classe `LinkedList` che implementa l'ADT *Lista* visto a lezione in grado anche di gestire elementi null. Fornire un'implementazione in Java commentato della specifica dell'esercizio 1 usando tale classe limitandosi a presentare 3 metodi: uno è l'aggiunta di un elemento, uno è l'intersezione e il terzo a scelta. Per ciascun metodo, indicare qual è l'ordine di grandezza del tempo di calcolo richiesto nel caso peggiore.

Esercizio 9.3 (*max 8 punti*)**.** Si dia un'implementazione in Java commentato della specifica dell'esercizio 1 che permetta di determinare nel medesimo ordine di tempo sia se un elemento appartiene all'insieme sia qual è l'elemento minimo dell'insieme.
Che assunzioni si devono fare per poter costruire questa implementazione? Ci sono restrizioni rispetto alla specifica del punto 1? Non è obbligatorio implementare tutti i metodi dichiarati nell'esercizio 1: ci si può limitare a fornire la struttura dati di supporto e i metodi necessari per rispondere alla domanda dell'esercizio. Indicare qual è l'ordine di grandezza del tempo di calcolo richiesto dalle due operazioni.

Esercizio 9.4 (*max 8 punti*)**.** Si supponga di avere una classe `Tree` che implementi l'ADT *Albero radicato generico* visto a lezione. Per comodità, si ricorda che la classe `GenericNode` usata da `Tree` contiene i metodi:

```
/** ...*/
public int degree() {...}
/** ...*/
public GenericNode getSon(int i) throws IndexOutOfBoundsException {...}
/** ...*/
public void setSon(int i, Object o) throws IndexOutOfBoundsException {...}
/** ...*/
public void removeSon(int i) throws IndexOutOfBoundsException {...}
/** ...*/
public GenericNode getFather() {...}
/** ...*/
public void setFather(Node newPadre) {...}
```

Si arricchisca la classe `Tree` implementando il metodo `postOrdine()` che esegue una visita in post-ordine dell'albero. Il metodo deve essere di tipo iterativo e scritto in Java completo di Javadoc e commenti nei punti chiave.

10 Appello del 25/06/2008

Si vuole implementare l'algoritmo di Bellman-Ford per il problema dei cammini minimi da sorgente unica. A tal fine si risponda di conseguenza ai seguenti quesiti ricordando che ogni risposta deve essere adeguatamente commentata.

Esercizio 10.1 (*max 7 punti*)**.** Scrivere le interfacce degli ADT `Node` e `Edge` in modo che:

- entrambi devono poter rappresentare un valore associato di tipo confrontabile;
- l'arco deve essere diretto;

Nota: le interfacce devono essere scritte in Java con commenti in Javadoc. Ogni commento poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante costa 2 punti.

Esercizio 10.2 (*max 7 punti*)**.** Scrivere l'interfaccia dell'ADT `WeightedGraph` dove il grafo è da intendersi diretto. L'interfaccia deve usare i tipi definiti nell'esercizio 1. *Osservare la stessa nota esercizio 1.*

Esercizio 10.3 (*max 8 punti*)**.** L'algoritmo di Bellman-Ford prevede di inizializzare un array (che chiameremo `distance`) di upper bound per la distanza del cammino minimo dalla sorgente a ciascun nodo e un array di predecessori (che chiameremo `predecessor`) che indica per ciascun nodo qual è il predecessore nel cammino minimo.

Si implementi tale procedura (che chiameremo `initialiseSingleSource`) in Java commentato comprensivo della gestione delle opportune eccezioni sui parametri assumendo:

- che i nodi siano identificati solo dal nome: quindi `distance` e `predecessor` non sono array ma...;
- di avere in input: g di tipo `WeightedGraph` (il grafo), s di tipo `Node` (nodo sorgente), `distance` di tipo ... e `predecessor` di tipo ...;
- che il peso del cammino sia un intero;

Considerato che l'implementazione **dovrà** usare uno o più metodi `WeightedGraph` o `Node` o `Edge`, si dia l'implementazione in Java commentato anche dei metodi usati.

Esercizio 10.4 (*max 8 punti*)**.** Si implementi quindi la procedura `relax(WeightedGraph g, Node u, Node v, ... distance, ... predecessor)` in Java commentato con la gestione delle opportune eccezioni.

A questo punto l'algoritmo di Bellman-Ford è praticamente scritto. Se ne dia l'implementazione usando le procedure fino a questo punto sviluppate.

11 Appello del 22/07/2008

Si vuole implementare l'ADT heap binario. A tal fine si risponda in modo adeguato ai seguenti quesiti.

Esercizio 11.1 (*max 7 punti*)**.** Per semplicare la gestione, assumiamo che uno heap generico sia costituito da nodi, ciascuno dei quali rappresenta un elemento. Scrivere l'interfaccia `HeapNode`, che rappresenta un nodo di uno heap, in modo che:

- rappresenti un elemento in modalità sola lettura;
- il tipo dell'elemento deve essere il più generale possibile (compatibilmente con le esigenze della struttura);
- permetta il confronto del nodo stesso con altri nodi (indicando all'implementatore che il confronto dovrà essere fatto sul valore).

Nota: l'interfaccia deve essere scritte in Java con commenti in Javadoc. Ogni commento poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante costa 2 punti.

Esercizio 11.2 (*max 7 punti*)**.** Scrivere l'interfaccia Java per l'ADT `Heap` generico in modo che:

- rappresenti le operazioni fondamentali per operare su uno heap; per ciascuna operazione si deve restituire l'esito della stessa (se significativo) come valore booleano e si deve specificare un lancio di eccezione quando...
- permetta di ottenere un riferimento all'elemento **minimo** dello heap;
- usi `HeapNode` per rappresentare gli elementi;
- **non** rappresenti le possibili operazioni di manipolazione di più heap (come la union di 2 heap);

Osservare la stessa nota esercizio 1.

Esercizio 11.3 (*max 8 punti*)**.** Si vuole ora realizzare un'implementazione dell'interfaccia dell'esercizio 2.
Tra i possibili heap, si consideri lo heap **binario** in cui lo heap è rappresentato come un albero binario quasi completo che soddisfa la proprietà del min-heap: ogni nodo diverso dalla radice ha valore maggiore del nodo padre.
Si descriva la struttura dati necessaria per un'implementazione efficiente dello heap **binario**.
Si formalizzi la struttura dati proposta in Java con il commento Javadoc appropriato.

Esercizio 11.4 (*max 8 punti*)**.** Si dia l'implementazione del metodo `boolean insert(HeapNode n)` dell'interfaccia dell'esercizio 2 all'interno della classe dell'esercizio 3 con i commenti del caso in modo tale che la complessità sia $O(n+\log n)$ nel caso peggiore, dove n è il numero di elementi presenti nello heap.

12 Appello del 02/09/2008

Si vuole definire una struttura dati specializzata e il codice necessario per implementare l'algoritmo **bucket-sort** in grado di ordinare interi di qualsiasi valore finito. A tal fine si risponda in modo adeguato ai seguenti quesiti.

Esercizio 12.1 (*max 7 punti*)**.** L'algoritmo prevede di suddividere gli elementi in *bucket*, ordinare questi bucket e poi costruire l'array ordinato estraendo gli elementi dai bucket.
Visto che ciascuno di questi bucket deve contenere un numero arbitrario di oggetti `Integer`, scrivere l'interfaccia `Bucket` che rappresenti l'ADT.

Nota: l'interfaccia deve essere scritta in Java con commenti in Javadoc che spieghino anche il perché dei metodi. Ogni commento poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante costa 2 punti.

Esercizio 12.2 (*max 7 punti*)**.** Si presenti ora una parte di una possibile implementazione, che chiameremo `BucketImpl`, dell'interfaccia del punto 1 in modo che:

- la struttura dati di supporto sia ben giustificata;
- contenga il metodo per permetta di aggiungere un elemento;
- contenga il metodo che permetta di accedere al k-esimo elemento in tempo $O(1)$.

Osservare la stessa nota esercizio 1.

Esercizio 12.3 (*max 8 punti*)**.** Si vuole estendere la classe presentata nell'esercizio 2 in modo che il metodo che aggiunge l'elemento, lo aggiunga in modo ordinato (ordine crescente) rispetto agli elementi eventualmente già presenti nel bucket.

Si scriva quindi tutta la classe `OrderedBucket` come *estensione* della classe `BucketImpl`.

Esercizio 12.4 (*max 8 punti*)**.** Si dia ora l'implementazione dell'algoritmo `Integer[] bucketSort(Integer a)` utilizzando la classe `OrderedBucket` realizzata nell'esercizio 3.

13 Appello del 13/01/2009

Si vuole definire una struttura dati e il codice necessario per realizzare un'implementazione efficiente di due semplici algoritmi per grafi: determinazione *pozzo universale* (nodo con nessun arco uscente e con $|V|-1$ archi entranti) e *classificazione degli archi* in base al loro tipo rispetto alla ricerca in profondità. A tal fine si risponda in modo adeguato ai seguenti quesiti.

Esercizio 13.1 (*max 7 punti*)**.** Si assuma che i nodi siano rappresentati da interi e che gli archi non siano pesati.
Si definisca un'interfaccia, `SimpleGraph`, che rappresenti le operazioni tipiche di gestione di un grafo orientato in cui i nodi non possono essere rimossi: inserimento nodo, inserimento/cancellazione di un arco dati i due nodi, ecc.
Suggerimento: inserire un nodo in un tale grafo non deve richiedere nessun parametro e deve restituire l'indice del nodo inserito.
NOTA: L'INTERFACCIA DEVE ESSERE SCRITTA IN JAVA CON COMMENTI IN JAVADOC CHE SPIEGHINO ANCHE IL PERCHÉ DEI METODI. OGNI COMMENTO POCO PRECISO O INCOMPLETO COSTA 1 PUNTO. OGNI METODO FONDAMENTALE MANCANTE COSTA 2 PUNTI. ATTENZIONE A PORRE LE GIUSTE ECCEZIONI!

Esercizio 13.2 (*max 7 punti*)**.** Si presenti ora una parte di una possibile implementazione, che chiameremo `SimpleGraphImpl`, dell'interfaccia del punto 1 in modo che:

- la struttura dati di supporto sia ben giustificata;
- contenga il metodo per permetta di aggiungere un nodo;
- contenga il metodo `boolean isEdge(int u, int v)` che indica se esiste un arco tra i due nodi dati. Questo metodo deve avere complessità in tempo $\Theta(1)$.

Osservare la stessa nota esercizio 1.

Esercizio 13.3 (*max 8 punti*)**.** Si scriva il codice Java del metodo `public int getUniversalSink()` che restituisce l'indice del nodo *pozzo universale*, se esiste, `-1` altrimenti. Il metodo deve avere complessità in tempo $O(|V|)$ e usare solo il metodo `isEdge(int u,int v)` dell'esercizio precedente.

Esercizio 13.4 (*max 8 punti*)**.** Si scriva il codice Java del metodo `public char[][] classifyEdges()` che restituisce la matrice di adiacenza in cui gli archi sono etichettati in base al loro tipo in seguito alla visita in profondità del grafo. Si ricorda che il tipo di un arco (u,v) è definito come:

- `'A'`(lbero) se v viene scoperto per la prima volta durante l'esplorazione (u,v);
- `'B'`(ack) se v è un antenato di u in un albero di profondità;
- `'F'`(orward) se l'arco non di è tipo A e v è un discendente di u;
- `'C'` se non è di un tipo precedente.

Parte IV
Anno accademico 2008/2009

14 Appello del 02/04/2009

Premessa Si vuole definire una parte della struttura dati e il codice necessario per realizzare un'implementazione dell'ADT *Coda di priorità*. A tal fine si risponda in modo adeguato ai seguenti quesiti ricordando che le interfacce devono essere scritte in Java con commenti in Javadoc significativi. Ogni commento poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante costa 2 punti.

Esercizio 14.1 (*max 7 punti*)**.** Si scriva l'interfaccia per l'ADT `Coda` che può contenere solo oggetti costituiti da un campo di nome `key` e uno di nome `weight`. I due campi servono, rispettivamente, a etichettare e a pesare (per poi confrontare) un oggetto. *Suggerimento: Forse è necessario strutturare l'interfaccia...*

Esercizio 14.2 (*max 7 punti*)**.** L'ADT `Coda di max-priorità` richiede che l'elemento restituito da `dequeue()` sia sempre quello con `weight` maggiore rispetto a quello degli altri presenti nella coda. Si riscriva quindi la specifica di quei metodi di `Coda` che debbono essere modificati in modo che `Coda` diventi la nuova interfaccia `CodaDiMaxPriorità`. Si aggiunga, infine, il metodo `boolean increase(key, weight)` che assegna `weight` al peso dell'elemento identificato da `key` se il parametro è maggiore del peso corrente dell'elemento...

Esercizio 14.3 (*max 8 punti*)**.** Si presenti ora una parte di una possibile implementazione di `CodaDiMaxPriorità` che usi come struttura dati di supporto lo *heap*. In particolare, si scriva il codice Java di: (1) la dichiarazione della struttura dati della classe (inclusa l'eventuale struttura dati della classe interna) e (2) l'implementazione del metodo `maxHeapify(int i)` che sistema l'elemento di indice i nella giusta posizione assumendo che il suo sottoalbero sia uno heap.

Esercizio 14.4 (*max 8 punti*)**.** Si scriva il codice Java del metodo `increase(key, weight)` assumendo di avere a disposizione solo il metodo `parentIndex(index)` e che la ricerca dell'elemento con chiave key possa avvenire in tempo $O(n)$ dove n è il numero di elementi dello heap.

15 Appello del 29/06/2009

Premessa Si vuole definire una parte della struttura dati e il codice necessario per realizzare un'implementazione dell'algoritmo di Floyd-Warshall. A tal fine si risponda in modo adeguato ai seguenti quesiti ricordando che le interfacce devono essere scritte in Java con commenti in Javadoc significativi. Ogni commento poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante costa 2 punti.

Esercizio 15.1 (*max 7 punti*)**.** Si scriva l'interfaccia per l'ADT `ArcoOrientato` che dichiari le operazioni fondamentali per un arco orientato che può avere un'`etichetta` e un `costo` entrambi modificabili. L'etichetta non identifica l'arco. Gli estremi dell'arco devono essere di tipo `int` mentre il costo di tipo `double`.

Esercizio 15.2 (*max 7 punti*)**.** Si scriva la struttura dati della classe `DigrafoPesato` che permetta di gestire grafi orientati pesati in cui i nodi sono rappresentati da `int` e gli archi da oggetti di tipo `ArcoOrientato`. La struttura dati deve garantire l'accesso ad un arco in tempo $O(1)$ dati i suoi estremi. Assumendo che i nodi assumano sempre numerazione $1, 2, \ldots, n-1, n$, si presenti il codice del metodo `int addNode()` che aggiunge un nuovo nodo al grafo e restituisce l'identificatore del nodo aggiunto. Si presenti poi l'implementazione del metodo `boolean addEdge(ArcoOrientato a)` che aggiunge o aggiorna l'arco creando eventualmente i due nodi se non sono presenti e sono ammissibili secondo la numerazione assunta; il metodo deve restituire vero se l'arco è stato aggiunto, falso se è stato aggiornato.

Esercizio 15.3 (*max 8 punti*)**.** Si implementi il metodo `double[][] floydWarshall()` (all'interno della classe `DigrafoPesato`) che determina la matrice delle distanze minime tra tutti i nodi secondo l'algoritmo di Floyd Warshall. Si implementi quindi il metodo `boolean negativeCycle(double[][] distance)` che restituisce vero se la matrice contiene cicli negativi, falso altrimenti.

Esercizio 15.4 (*max 8 punti*)**.** Si estenda il metodo `double[][] floydWarshall()` aggiungendo le righe necessarie affinché il metodo determini anche la matrice dei predecessori. Per semplificare l'implementazione, si assume che la matrice dei predecessori sia definita come variabile di istanza `int[][] predec;`.

16 Appello del 15/07/2009

Premessa Si vuole definire una parte della struttura dati e il codice necessario per realizzare un'implementazione dell'algoritmo di Dijkstra. A tal fine si risponda in modo adeguato ai seguenti quesiti ricordando che le interfacce

devono essere scritte in Java con commenti in Javadoc significativi. Ogni commento poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante costa 2 punti.

Esercizio 16.1 (*max 7 punti*)**.** Si scrivano due interfacce: una per l'ADT `Node` e una per l'ADT `Edge`. La prima deve contenere le operazioni fondamentali per un nodo in cui l'identità è basata sul valore dell'attributo `name` di tipo `String` che non può essere modificato. La seconda deve contenere le operazioni fondamentali per un arco pesato orientato in cui l'identità è basata sul valore degli estremi e in cui il peso è di tipo `Integer`.

Esercizio 16.2 (*max 7 punti*)**.** Si scriva la struttura dati **commentata** della classe `WeightedGraph` che permetta di gestire grafi orientati pesati in cui i nodi sono rappresentati da oggetti di tipo `Node` e gli archi da oggetti di tipo `Edge`. La struttura dati deve garantire l'accesso ad un nodo in un tempo mediamente indipendente dalla dimensione del grafo.

Si presenti il codice del metodo `Node[] getNodes()` che restituisce i nodi del grafo in un tempo $O(n)$ e del metodo `Edge[] getAdj(Node)` che restituisce gli archi uscenti dal nodo dato.

Esercizio 16.3 (*max 8 punti*)**.** Si implementi il metodo `boolean SPSSDijkstra(WeightedGraph g, Node s, Map distance, Map predecessor)` (metodo di una nuova classe) che determina le distanze dei cammini minimi secondo l'algoritmo di Dijkstra. `s` è il nodo di partenza. `distance` è la mappa `(String, Comparable)` che, alla fine, dovrà contenere le distanze minime. `predecessor` è la mappa `(String, Node)` che, alla fine, dovrà contenere il predecessore per ciascun nodo nel cammino minimo.

Il metodo restituisce vero se l'algoritmo termina correttamente, falso altrimenti (quando deve SEMPRE restituire falso?).

In questo esercizio si può ricorrere alla classe `MinPriorityQueue` che contiene i metodi
`void enqueue(String key, Comparable value)`,
`Entry dequeue()`, dove Entry è classe con due campi: `Entry.key, Entry.value`
`boolean decrease(String key, Comparable newValue)`
la cui semantica è quella classica delle operazioni sull'ADT Coda di priorità minima.

Esercizio 16.4 (*max 8 punti*)**.** Si scriva il metodo `String shortestPath(WeightedGraph g, Node s, Node d, Map distance, Map predecessor)` che restituisce la stringa che rappresenta il cammino minimo tra `s` e `d` nel formato "`s->`v_i`->`v_j`->...->d: <costo>` ricorrendo alla struttura dati più indicata per costruire tale stringa.

17 Appello del 07/09/2009

Premessa Argomento di questa prova sono gli alberi binari di ricerca. Come in passato, si chiede di rispondere in modo adeguato ai seguenti quesiti ricordando che il codice deve essere scritto in Java con commenti in Javadoc. Ogni commento mancante o poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante nelle interfacce costa 2 punti.

Esercizio 17.1 (*max 7 punti*)**.** Si scrivano due interfacce: `BinaryNode` e `BinaryTree`. La prima deve contenere le operazioni fondamentali per un nodo di un albero binario di ricerca. Si ricordi che ogni nodo deve contenere un valore confrontabile in sola lettura e che l'interfaccia NON deve richiedere l'accesso al padre. La seconda deve contenere le operazioni fondamentali per gestire un albero binario di ricerca in cui i nodi sono di tipo `BinaryNode`. Per comodità, il metodo che indica se un **valore** è presente deve essere chiamato `find(...)`.

Esercizio 17.2 (*max 7 punti*)**.** Si scriva (con commenti) la struttura dati e i costruttori che si ritengono opportuni (si omettano gli altri metodi) della classe `BinaryNodeImpl` che implementa l'interfaccia `BinaryNode` dell'esercizio 1 assumendo di adottare una rappresentazione linkata e fissando il tipo del valore contenuto a `String`. Analogamente, si scriva poi la struttura dati e il metodo `find` della classe `BinaryTreeImpl` che usa `BinaryNodeImpl`.

Esercizio 17.3 (*max 8 punti*)**.** Si supponga ora di voler arricchire `BinaryTreeImpl` con un metodo (nome `traversal(...)`) che permetta di visitare l'albero in cui sia il tipo di visita sia l'operazione che si vuole eseguire su ciascun nodo devono essere passati come parametri.

Quali sono le strutture dati necessarie (interfacce o classi) per poter specificare tali parametri (se non sono già state dichiarate nell'interfaccia del 1 esercizio)?
Qual è la segnatura finale del metodo?

Non proporre la soluzione vista a lezione: non è sufficiente!

Esercizio 17.4 (*max 8 punti*)**.** Si scriva un'implementazione completa dei metodi specificati nell'esercizio precedente (`traversal(...)` e tutti quelli richiamati) assumendo che l'operazione da eseguire su ciascun nodo sia quello di stampare il valore contenuto nel nodo e il numero dei nodi visitati fino a quel momento.

Con quale visita i valori ottenuti dall'operazione implementata indicano, per ciascun nodo, quanti sono i nodi con valori inferiori?

18 Appello del 21/09/2009

Premessa Argomento di questa prova è l'implementazione di un'estensione di QuickSort e una di MergeSort. Si chiede di rispondere in modo adeguato ai seguenti quesiti ricordando che il codice deve essere scritto in Java con commenti in Javadoc. Ogni commento mancante o poco preciso o incompleto costa 1 punto. Ogni metodo fondamentale mancante nelle interfacce costa 2 punti. **Per risolvere gli esercizi, non è ammesso usare o richiamare classi della libreria java.lang.* o java.util.*** se non espressamente richiesto.
Si ricorda che l'interfaccia `java.util.Comparator` dichiara la segnatura di due metodi: `int compare(Object o1, Object o2)` e `boolean equals(Object obj)`. Il metodo `compare(o1,o2)` deve avere la medesima semantica di `o1.compareTo(o2)` (assumendo `o1` di tipo `Comparable`). Il metodo `equals()` deve indicare se il corrente oggetto comparatore impone lo stesso ordinamento di quello dato come parametro.

Esercizio 18.1 (*max 7 punti*)**.** Si scriva l'interfaccia per il tipo `Node` in cui sia possibile gestire le seguenti informazioni: `key` in sola lettura, `weight`, `value` e `spread`, in lettura e scrittura. Quest'ultimi tre devono essere di tipo confrontabile. Infine, non devono essere ammessi valori nulli.

Si scriva quindi una classe (`NodeImpl`) che implementi l'interfaccia `Node` limitandosi alla struttura dati e ai costruttori che si ritengono opportuni (si omettano gli altri metodi).

Esercizio 18.2 (*max 7 punti*)**.** Si scriva un'interfaccia (`NodeComparator`) che estenda `java.util.Comparator` richiedendo un overload di `compare(...)` che accetti come parametri due oggetti di tipo `Node`.

Si scrivano quindi due classi (`WeightCmp` e `ValueCmp`) che implementano l'interfaccia `NodeComparator` (limitatamente al metodo `compare(...)`): la prima deve realizzare un ordinamento decrescente rispetto all'informazione `weight`, la seconda un ordinamento crescente rispetto l'informazione `value` prevedendo, in caso di uguale valore, di ordinare secondo `WeightCmp`.

Esercizio 18.3 (*max 8 punti*)**.** Si scriva un'implementazione di QuickSort per una array di oggetti `Node` tale che: (i) l'elemento di pivot è l'elemento di posizione centrale, (ii) usi un oggetto di tipo `NodeComparator` (passato come argomento) per confrontare gli oggetti.

Esercizio 18.4 (*max 8 punti*)**.** Si scriva un'implementazione di MergeSort per una array di oggetti `Node` che usi un oggetto di tipo `NodeComparator` (passato come argomento) per confrontare gli oggetti.

Tornando all'esercizio 2, ha senso fare l'overriding del metodo `equals` nelle due classi (`WeightCmp` e `ValueCmp`) in modo da rispondere alla specifica di `java.util.Comparator` che richiede di restituire vero quando l'argomento determina lo stesso tipo di confronto del oggetto corrente? Perché? Se si lascia l'implementazione di default, cosa si perde?