Laboratorio di Algoritmi e Strutture Dati
Anno Accademico 2005/06
Prof. Carlo Blundo

Raccolta di esercizi
Versione del 26 ottobre 2005

Alcuni di questi esercizi sono tratti dai libri di testo del corso ("*Data structures and algorithms*" di Goodrich e Tamassia e "*Introduction to Algorithms*" di Cormen, Leiserson, Rivest, Stein) Altri esercizi sono stati presi in vari siti di corsi di Algoritmi e Strutture Dati o Laboratorio di ASD di cui non ricordo i riferimenti esatti. Infine, alcuni esercizi sono stati proposti da me durante le varie edizioni del corso di LASD o durante le sessioni di esame. Se qualcuno, non vuole che il proprio esercizio venga inserito in questa raccolta oppure vuole che la fonte dell'esercizio venga citata opportunamente è pregato di contattarmi.

Carlo Blundo

Stack

1. Implementare l'interfaccia **Stack** (scrivere la classe **ArrayStack**) usando un array di tipo **Object** di lunghezza fissata (e.g., private static final int CAPACITY = 1024;). Se lo stack è pieno lanciare l'eccezione **StackFullException**.

2. Testare l'implementazione della classe **ArrayStack** scrivendo un programma **TestStack** che utilizzi tutti i metodi dell'interfaccia **Stack**.

3. Modificare la classe **ArrayStack** in modo che se l'array è pieno, allora venga allocato dello spazio sufficiente a contenere nuovi elementi e non venga lanciata l'eccezione **StackFullException.** Lo stack deve essere *estendibile.*

4. Usando la struttura dati **ArrayStack**, invertire una stringa ricevuta in input. Ad esempio, dando in input la stringa "casa" in output deve essere fornita la stringa "asac".

5. Scrivere una funzione Java che ricevuto in input uno stack restituisce un nuovo stack che corrisponde ad una copia palindroma dello stack passato per argomento. Lo stack originale non deve essere modificato. Ad esempio se lo stack passato in input è <1, 2, 4, 5>, allora lo stack ritornato come output della funzione sarà <1, 2, 4, 5, 5, 4, 2, 1>.

6. Si scriva una funzione Java **Stack inverti(Stack s)** che restituisca un nuovo stack ottenuto invertendo il contenuto di **s**. Si osservi che la funzione deve lasciare inalterato il contenuto di **s** alla fine dell'esecuzione (lo può modificare durante).

7. Scrivere la funzione Java **Stack StackSum(Stack s)** che restituisce un nuovo stack ottenuto a partire da s aggiungendo dopo ogni elemento la somma degli elementi che lo precedono in **s**. Ad esempio se lo stack **s** passato in input è <1, 2, 4, 5> (il valore 1 è al top dello stack), allora lo stack restituito in output sarà: <1, 1, 2, 3, 4, 7, 5, 12>. La funzione deve lasciare inalterato il contenuto di **s** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità della funzione proposta? Il problema può essere risolto in tempo lineare nel numero degli elementi nello stack **s**?

8. In un'espressione aritmetica sono ammesse indifferentemente le parentesi tonde "()" o le parentesi quadre "[]", purché siano bilanciate correttamente. Scrivere un funzione che, ricevendo in input una stringa, restituisce **true** se e solo se le parentesi sono bilanciate correttamente. Ad esempio le parentesi nelle seguenti espressioni sono bilanciate
   a. Rd(3e)d(d(i)d)
   b. C(f(f(f(g(t))d)iu)d)dfe

   Ma le parentesi nelle seguenti espressioni non sono bilanciate
   c. Rwe(fd)(fd(ffs)c
   d. Fdr(((d)(d)d(f(id(i))f)(d))

9. Aggiungere alla classe **ArrayStack** il metodo **multiPop**(int **num**) che restituisce un array contenente i primi **num** elementi presenti nello stack. Il metodo deve cancellare dallo stack gli elementi restituiti. Inoltre, deve lanciare l'eccezione **NotEnoughElements** se nello stack vi sono meno di **num** elementi.

10. Si aggiunga alla classe **ArrayStack** (implementa l'interfaccia **Stack** usando un array) il metodo **Stack clone()** che restituisce un nuovo stack avente lo stesso contenuto dello stack

su cui è invocato. Il metodo deve lasciare inalterato il contenuto dello stack su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

11. L'interfaccia **DoppioStack,** che gestisce due stack contemporaneamente, consente le seguenti operazioni:
    - **push1():** inserimento nel primo stack.
    - **push2():** inserimento nel primo stack.
    - **pop1():** rimozione e restituzione dell'elemento nel primo stack.
    - **pop2():** rimozione e restituzione dell'elemento nel secondo stack.
    - **top1():** restituzione (senza rimozione) dell'elemento nel primo stack.
    - **top2():** restituzione (senza rimozione) dell'elemento nel secondo stack.

    Si implementi la classe **ArrayDoppioStack** usando un array di tipo **Object** di lunghezza fissata (e.g., private static final int CAPACITY = 1024;).

12. Testare l'implementazione della classe **ArrayDoppioStack** scrivendo un programma **TestArrayDoppioStack** che utilizzi tutti i metodi dell'interfaccia **DoppioStack**.

13. Aggiungere alla classe **ArrayStack** il metodo **String toString()** che restituisce una stringa rappresentante il contenuto dello stack su cui è invocato. Gli elementi dello stack dovranno essere separati da virgola e racchiusi da parentesi quadre. L'elemento al top dello stack deve essere preceduto dalla parola top. Ad esempio se lo stack è <1, 6, 4, 7>, allora il metodo deve restituire la stringa "[top 1, 6, 4, 7]". Il metodo deve lasciare inalterato il contenuto dello stack su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

14. Si consideri la seguente interfaccia:

```
public interface ExtendedStack {
    // Restituisce il numero degli elementi nello stack
    public int size();

    // Restituisce true se lo stack è vuoto, false altrimenti
    public boolean isEmpty();

    // Restituisce l'elemento al top dello stack.
    // Se lo stack è vuoto  lancia l'eccezione StackEmptyException
    public Object top()  throws StackEmptyException;

    // Inserisce nello gli elementi nel vettore elements
    // elements[0] è il primo elemento inserito
    // elements[elements.length] è il primo elemento inserito
    public void mulitPush(Object[] elements);

    // Rimuove i primi n elementi al top dello stack
    // restituendoli in un array. Se lo stack è vuoto lancia
    // l'eccezione StackEmptyException. Se ci sono meno di n elementi
    // lancia l'eccezione NotEnoughElements
    public Object multiPop(int n) throws
                    StackEmptyException, NotEnoughElements;
}
```

    Implementare l'interfaccia **ExtendedStack**. La classe **ArrayExtendedStack** che implementa tale interfaccia deve far uso di una sola variabile istanza di tipo **ArrayStack**.

15. Testare l'implementazione della classe **ArrayExtendedStack** scrivendo un programma **TestArrayExtendedStack** che utilizzi tutti i metodi dell'interfaccia **ExetendedStack**.

16. Aggiungere alla classe **ArrayStack** il metodo **void removeAll(Object elem)** che cancella dallo stack tutte le occorrenze dell'elemento **elem**. Se lo stack è vuoto il metodo deve lanciare l'eccezione **StackEmptyException** (modificare opportunamente la firma del metodo). Quale è la complessità del metodo proposto? Giustificare la risposta.

17. Scrivere la funzione **void removeAllFromStack(Object elem, Stack s)** che cancella dallo stack **s** tutte le occorrenze dell'elemento **elem**. Se lo stack **s** è vuoto il metodo deve lanciare l'eccezione **StackEmptyException** (modificare opportunamente la firma del metodo). Quale è la complessità del metodo proposto? Giustificare la risposta.

18. Scrivere una funzione Java **Stack concat(Stack s1, Stack s2)**. La funzione restituisce la concatenazione dei due stack **s1** ed **s2** ricevuti in input (gli elementi di **s1** vengono inseriti prima degli elementi di **s2**). Se **s1** è <1, 6, 4, 7> mentre **s2** è <12, 16, 44, 72>, allora lo stack restituito sarà: <12, 16, 44, 72, 1, 6, 4, 7>La funzione deve lasciare inalterato il contenuto degli stack **s1** ed **s2** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità di tempo di tale programma?

Coda

1. Implementare l'interfaccia **Queue** (scrivere la classe **ArrayQueue**) usando un array di tipo Object di lunghezza fissata (e.g., public static final int CAPACITY = 1024;). Se la coda è piena lanciare l'eccezione **QueueFullException**.

2. Testare dell'implementazione della classe **ArrayQueue** scrivendo un programma **TestQueue** che utilizzi tutti i metodi dell'interfaccia **Queue**.

3. Modificare la classe **ArrayQueue** in modo che se la coda è piena, allora venga allocato dello spazio sufficiente a contenere nuovi elementi e non venga lanciata l'eccezione **QueueFullException**. (La coda deve essere estendibile).

4. Sia **ArrayQueue** la classe che implementa l'interfaccia **Queue** mediante un array. Aggiungere il metodo **void reverse()** che inverte il contenuto della coda.

5. Scrivere una funzione Java che ricevuto in input una coda q restituisce in output true se q è palindroma. Ad esempio, le code <1,2,3,2,1> <1,2,2,1>sono palindrome; mentre la coda <1,2,3,3,1> non lo è. La funzione deve lasciare inalterato il contenuto di **q** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità della funzione proposta?

6. Descrivere come implementare il TDA Stack usando due code. Quale è la complessità di tempo dei metodi pop() e push()?

7. La classe **StackQueue** implementa l'interfaccia **Stack** utilizzando come variabili istanza solo due code **q1** e **q2**. Supponendo che i valori contenuti nello stack sono conservati nella coda **q1**, una bozza della classe **StackQueue** è la seguente:

```
public class StackQueue implements Stack {
        Queue q1, q2;
        public boolean isEmpty() { return q1.isEmpty(); }
        public boolean size() { return q1.size(); }
                ...
}
```

completare l'implementazione della classe **StackQueue**.

8. Si aggiunga alla classe **ArrayQueue** (implementa l'interfaccia **Queue** usando un array) il metodo **Queue clone()** che restituisce un nuova coda avente lo stesso contenuto della coda su cui è invocato. Il metodo deve lasciare inalterato il contenuto della coda su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

9. Aggiungere alla classe **ArrayQueue** il metodo **String toString()** che restituisce una stringa rappresentante il contenuto della coda su cui è invocato. Gli elementi della coda dovranno essere separati da virgola e racchiusi da parentesi quadre. L'elemento al rear della coda deve essere seguito dalla parola rear. L'elemento al front della coda deve essere preceduto dalla parola front. Ad esempio se la coda è <1, 6, 4, 7>, allora il metodo deve restituire la stringa "[front 1, 6, 4, 7 rear]". Il metodo deve lasciare inalterato il contenuto della coda su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

10. Scrivere una funzione **EstraiDaCoda(Queue q, int n)** che ricevuto in input una coda **q** di interi (**Integer**) ed un numero intero (**Integer**) **n** restituisca una coda **r** contenente tutti gli elementi di **q** che sono multipli di n e maggiori di 27. Gli elementi di **r** devono comparire nello stesso ordine in cui comparivano in **q**. La funzione deve lasciare inalterato il contenuto di **q** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità della funzione proposta?

19. Aggiungere alla classe **ArrayQueue** il metodo **void removeAll(Object elem)** che cancella dalla coda tutte le occorrenze dell'elemento **elem**. Se la coda è vuoto il metodo deve lanciare l'eccezione **QueueEmptyException** (modificare opportunamente la firma del metodo). Quale è la complessità del metodo proposto? Giustificare la risposta.

20. Scrivere la funzione **void removeAllFromStack(Object elem, Queue q)** che cancella dalla coda **q** tutte le occorrenze dell'elemento **elem**. Se la coda **q** è vuota il metodo deve lanciare l'eccezione **QueueEmptyException** (modificare opportunamente la firma del metodo). Quale è la complessità del metodo proposto? Giustificare la risposta.

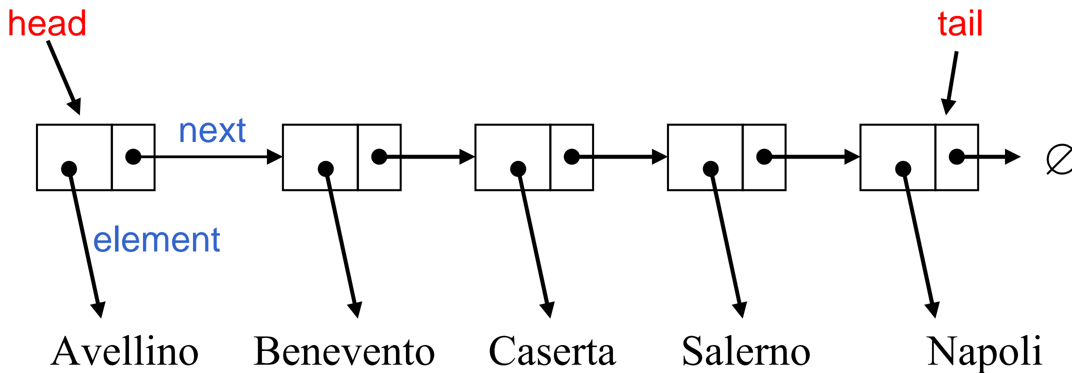
21. Scrivere una funzione Java **Queue concat(Queue q1, Queue q2)**. La funzione restituisce la concatenazione delle due code **q1** e **q2** ricevute in input (gli elementi di **q1** vengono prima degli elementi di **q2**). La funzione deve lasciare inalterato il contenuto delle code **q1** e **q2** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità di tempo di tale programma?

22. Scrivere una funzione Java **Queue concatNoDup(Queue q1, Queue q2)**. La funzione restituisce la concatenazione senza duplicati delle due code **q1** e **q2** ricevute in input (gli elementi di **q1** vengono prima degli elementi di **q2**). La concatenazione senza duplicati indica il fatto che se un elemento non può comparire due volte nella coda restituita in output. Per semplicità si può supporre che le code contengono numeri interi. La funzione deve lasciare inalterato il contenuto delle code **q1** e **q2** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità di tempo di tale programma?

23. Scrivere una funzione Java **Queue dammiPari(Queue q)** che ricevuta in input una coda **q** di elementi qualsiasi restituisce in output una coda contenente solo gli elementi in posizione pari. La funzione deve lasciare inalterato il contenuto della coda **q** alla fine dell'esecuzione (lo può modificare durante).

Lista Lincata

Una lista lincata non è il TDA Lista, ma una versione semplificata della struttura dati Lista. Una lista lincata nella sua forma più semplice è una collezione di **nodi** che formano un ordine lineare. Ogni **nodo** è un oggetto composto che memorizza un riferimento ad un elemento ed un riferimento ad un altro **nodo**. Il primo nodo della lista è detto *head*, l'ultimo *tail*. Ad esempio,



Una possibile implementazione del nodo è la seguente:

```
public class Node {
        private Object element;
        private Node next;

        public Node() {  this(null, null); }

         public Node(Object e, Node n) { element = e;  next = n; }

        public Object getElement() { return element;   }

        public Node getNext() {  return next; }

        public void setElement(Object newElem) { element = newElem;  }

        public void setNext(Node newNext) { next = newNext; }
}
```

1. Implementare il TDA Stack attraverso una lista lincata (scrivere la classe **LinkedStack**).

```
public class LinkedStack implements Stack {
                private Node top;    //riferimento al nodo head
                private int size;       // numero di elementi nello stack

                public LinkedStack()  // inizializza uno stack vuoto
                . . .
}
```

2. Aggiungere alla classe **LinkedStack** (implementa l'interfaccia **Stack** usando una lista lincata) un costruttore che ricevuto in input un array A di oggetti, inizializzi lo stack con gli elementi dell'array (il primo elemento da inserire nello stack è A[0], il secondo A[1], ...).

3. Implementare il TDA Queue attraverso una lista lincata (scrivere la classe **LinkedQueue**).

```
public class LinkedQueue implements Queue {
        private Node front;  // riferimento al nodo head
        private Node rear;        // riferimento al nodo tail
        private int size;           // numero di elementi nella coda
        public LinkedQueue()  // inizializza una coda vuota
        . . .
}
```

4. Sia **LinkedQueue** la classe che implementa l'interfaccia **Queue** mediante una lista lincata. Aggiungere il metodo **void reverse()** che inverte il contenuto della coda.

5. Modificare il metodo **int size()** della classe **LinkedQueue** in modo che conti in maniera ricorsiva i nodi presenti nella coda.

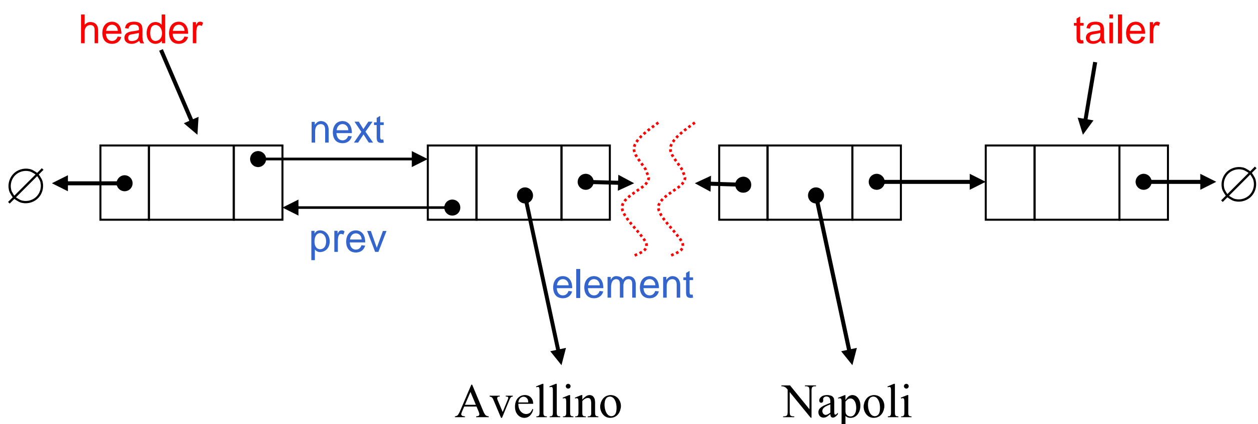
Lista Doppiamente Lincata

Una lista doppiamente lincata è una versione semplificata della struttura dati Lista. Una lista doppiamente lincata nella sua forma più semplice è una collezione di **nodi** che formano un ordine lineare. Ogni **nodo** è un oggetto composto che memorizza un riferimento ad un elemento e due riferimenti ad altri due **nodi**: il nodo che lo precede nella lista ed il nodo che lo segue nella lista.

Per migliorare sia l'efficienza del codice sia per semplificare la programmazione conviene usare due nodi sentinella

- header (È un nodo che punta alla testa (head) della lista. Ha il campo prev settato a null)
- trailer (È un nodo che punta alla coda (tail) della lista. Ha il campo next settato a null)

Ad esempio



Una possibile implementazione del nodo è la seguente:

```
public class DLNode {
  private Object element;
  private DLNode next, prev;

  public DLNode() { this(null, null, null); }

  public DLNode(Object e, DLNode p, DLNode n) {
    element = e;
    next = n;
    prev = p;
  }

  public void setElement(Object newElem) {  element = newElem; }

  public void setNext(DLNode newNext) { next = newNext; }

  public void setPrev(DLNode newPrev) { prev = newPrev; }

  public Object getElement() { return element; }

  public DLNode getNext() { return next; }

  public DLNode getPrev() { return prev; }

}
```

Deque

1. Implementare l'interfaccia **Deque** implementando la classe **DLinkedDeque** usando una lista doppiamente lincata formata di nodi di tipo **DLNode**.

2. Testare la vostra implementazione della classe **DLinkedDeque** scrivendo un programma **TestDeque** che utilizzi tutti i metodi dell'interfaccia **Deque**.

3. Implementare il TDA **Stack** usando come variabile d’istanza della classe una coda a doppio ingresso. Cambia la complessità di tempo dei vari metodi implementati? In caso affermativo, indicare come giustificando la risposta.

4. Implementare il TDA **Queue** usando come variabile d’istanza della classe una coda a doppio ingresso. Cambia la complessità di tempo dei vari metodi implementati? In caso affermativo, indicare come giustificando la risposta.

5. Si aggiunga alla classe **DLinkedDeque** (implementa l’interfaccia **Deque** usando una coda doppiamente lincata) il metodo **Deque clone()** che restituisce un nuova deque avente lo stesso contenuto della deque su cui è invocato. Il metodo deve lasciare inalterato il contenuto della deque su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

6. Aggiungere alla classe **DLinkedDeque** il metodo **String toString()** che restituisce una stringa rappresentante il contenuto della deque su cui è invocato. Gli elementi della deque dovranno essere separati da virgola e racchiusi da parentesi quadre. Il primo elemento della deque deve essere preceduto dalla parola first; mentre l’ultimo elemento della deque deve essere seguito dalla parola last. Ad esempio se la deque è <1, 6, 4, 7>, allora il metodo deve restituire la stringa “[first 1, 6, 4, 7 last]”. Il metodo deve lasciare inalterato il contenuto della deque su cui è invocato. Quale è la complessità del metodo proposto (giustificare la risposta).

7. Sia **DLinkedDeque** la classe che implementa l'interfaccia **Deque**. Aggiungere il metodo **void reverse()** che inverte il contenuto della deque.

8. Aggiungere alla classe **DLinkedDeque** il metodo void palindroma() che restituisce true se la deque su cui è invocato è palindroma. Ad esempio, le deque <1,2,3,2,1> e <1,2,2,1> sono palindrome; mentre la deque <1,2,3,3,1> non lo è. Quale è la complessità della funzione proposta?

9. Aggiungere alla classe **DLinkedDeque** il metodo **Object cancella(Object e)** che cancella la prima occorrenza dell'elemento **e** dalla deque. Il metodo deve restituire l'elemento cancellato oppure lanciare un'eccezione opportuna se l'elemento non è presente nell'istanza di **DLinkedDeque** .

10. Aggiungere alla classe **DLinkedDeque** il metodo **Object cancellaTutti(Object e)** che cancella tutte le occorrenze dell'elemento **e** dalla deque. Il metodo deve restituire l'elemento cancellato oppure lanciare un'eccezione opportuna se l'elemento non è presente nell'istanza di **DLinkedDeque**.

11. Scrivere la funzione Java **Object cancella(Deque d, Object e)** che cancella la prima occorrenza dell'elemento **e** dalla deque **d**. Il metodo deve restituire l'elemento cancellato

oppure lanciare un'eccezione opportuna se l'elemento non è presente nell'istanza di **DLinkedDeque**. Il resto della deque **d** deve rimanere inalterato dopo l'invocazione della funzione cancella.

12. Scrivere la funzione Java **Object cancellaTutti(Deque d, Object e)** che cancella tutte le occorrenze dell'elemento **e** dalla deque **d**. Il metodo deve restituire l'elemento cancellato oppure lanciare un'eccezione opportuna se l'elemento non è presente nell'istanza di **DLinkedDeque**. Il resto della deque **d** deve rimanere inalterato dopo l'invocazione della funzione cancella.

Vector

1. Implementare l'interfaccia **Vector** (scrivere la classe **ArrayVector**) usando come rappresentazione interna della classe un array di tipo **Object**

2. Testare l'implementazione dell'interfaccia **Vector** scrivendo un programma **TestVector** che utilizzi tutti i metodi dell'interfaccia **Vector**

3. Scrivere la classe **VectorDeque** che implementa l'interfaccia **Deque**. La classe **VectorDeque** deve usare come rappresentazione interna un'istanza della classe **ArrayVector**. Quale è la complessità dei metodi implementati (giustificare la risposta).

4. Si scriva la funzione **Vector estrai(Queue Q, Object o1, Object o2, Comparator C)** che restituisce il vettore di tutti gli oggetti presenti in **Q** compresi tra **o1** ed **o2** inclusi (per confrontare gli oggetti si deve usare il comparatore **C**). Usare solo i metodi dell'interfaccia **Queue** (però non è possibile utilizzare i metodi **elements()** e **positions()**). Si osservi che la funzione deve lasciare inalterato il contenuto di **Q** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità del metodo proposto (giustificare la risposta).

5. Scrivere la funzione **boolean contiene(Vector V)** che restituisce true se il vettore V contiene due elementi la cui somma è pari a 7. Si supponga che **V** contenga interi positivi. La funzione **contiene** deve avere una complessità di tempo pari ad O(n) dove n è il numero degli elementi contenuti in **V**. Illustrare la complessità di tempo della funzione proposta. Soluzioni con complessità di tempo superiore a O(n) saranno considerate nulle.

6. Si scriva una funzione **ArrayVector fromRankToRank(ArrayVector V, int r1, int r2)** che restituisce un nuovo **ArrayVector** contenente gli elementi di **V** che hanno rango compreso tra **r1** ed **r2**. La funzione deve lasciare inalterato il contenuto di **q** alla fine dell'esecuzione e lanciare l'eccezione **OutOfRankExceprion** se **r1** o **r2** non sono ranghi validi per **V**. Quale è la complessità del metodo proposto (giustificare la risposta).

List

1. Implementare l'interfaccia **List** (scrivere il codice della classe **NodeList**).

2. Testare l'implementazione di **NodeList** scrivendo un programma **TestNodeList** che utilizzi tutti i metodi dell'interfaccia **List**

3. Aggiungere alla classe **LinkedList** il metodo **Object getAtIndex(int i)** che restituisce l'elemento i-esimo della lista. Se tale elemento non esiste deve lanciare l'eccezione **ElementNotFound**.

4. Il tipo astratto **DoppioStack** consente le seguenti operazioni:
   - **headPush():** inserimento in testa.
   - **tailPush():** inserimento in coda.
   - **headPop():** rimozione e restituzione dell'elemento in testa.
   - **tailPop():** rimozione e restituzione dell' elemento in testa.
   - **headTop():** restituzione (senza rimozione) dell'elemento in testa.
   - **tailTop():** restituzione (senza rimozione) dell'elemento in coda.

   Si implementi in Java usando come variabile d'istanza della classe un oggetto di tipo **LinkedList**.

5. Aggiungere alla classe **NodeList** (che implementa il TDA **List** mediante una lista doppiamente linkata) il metodo: **void RemoveOdd**( ) che rimuove dalla lista tutti gli elementi di ordine dispari.

6. Si scriva la funzione **List estrai(List L, Object o1, Object o2, Comparator C)** che restituisce la lista di tutti gli oggetti presenti in **L** compresi tra **o1** ed **o2** inclusi (per confrontare gli oggetti si deve usare il comparatore **C**). Usare solo i metodi dell'interfaccia **List** (però non è possibile utilizzare i metodi **elements()** e **positions()**). Si osservi che la funzione deve lasciare inalterato il contenuto di **L** alla fine dell'esecuzione (lo può modificare durante). Quale è la complessità del metodo proposto (giustificare la risposta).

Sequence

1. Implementare l'interfaccia **Sequence** (scrivere la classe **NodeSequence**) usando una lista doppiamente lincata.

2. Implementare l'interfaccia **Sequence** (scrivere la classe **ArraySequence**) usando un array.

3. Aggiungere alla classe **NodeSequence** (implementa l'interfaccia **Sequenze** usando una lista doppiamente lincata) il metodo **Iterator elementsUpTo**(int **n**), che ricevuto in input un intero **n** restituisce in output un iteratore sugli elementi che hanno rango minore od uguale ad **n**. Il metodo deve lanciare l'eccezione **notEnoughElements** se non ci sono **n** elementi nella sequenza.

4. Scrivere una funzione Java **maggiori(Sequence a, Object val, Comparator comp)** che restituisce in output un ObjectIterator sugli elementi di **a** che sono maggiori di **val** secondo il comparatore **comp**.

5. Scrivere la funzione Java
   concat(Sequence a, Sequence b)
   che riceve in input due sequenze e restituisce in output una sequenza che corrisponde alla concatenazione delle sequenze **a** e **b**. Ad esempio se **a**=(2,1,6,3,8) e **b**=(6,4,2,9,10), come output viene restituita la sequenza (2,1,6,3,8, 6,4,2,9,10). Al termine della funzione, le sequenze **a** e **b** devono risultare immutate ma possono essere modificate durante l'esecuzione di **concat**.

6. Scrivere una funzione Java **concat(Sequence a, Sequence b)** che riceve in input due sequenze e restituisce in output una sequenza che corrisponde alla concatenazione delle sequenze **a** e **b** senza ripetizione. Ad esempio se **a**=(2,1,6,3,8) e **b**=(6,4,2,9,10), in output viene restituita la sequenza (2,1,6,3,8,4,9,10).

Dictionary

1. Sia **LogFile** l'implementazione dell'interfaccia **Dictionary** attraverso un'istanza LF della classe **NodeSequence**. Implementare il metodo public Object **findElement(**Object key**)** che restituisce l'elemento associato alla chiave key. Si supponga che le chiavi sono di tipo stringa e che due chiavi sono uguali se coincidono oppure se sono della stessa lunghezza. Implementare inoltre il metodo boolean **isEqualTo(**Object x, Object y**);** della classe **EqualityTester**. Discutere della complessità di tempo del metodo implementato.

2. Aggiungere alla classe D, che implementa un dizionario non ordinato, il metodo
   public Iterator EQ().
   Il metodo **EQ** restituisce un iteratore di tutti gli elementi il cui valore è uguale a quello della propria chiave. (Si tenga presente che un elemento può essere di un tipo che non è confrontabile con quello delle chiavi). Il metodo deve essere scritto senza fare assunzioni sul modo in cui la classe D implementa l'interfaccia **Dictionary.**

3. Si supponga che la classe **LogFile** implementi l'interfaccia **Dictionary** attraverso un'istanza **LF** della classe **NodeSequence**. Implementare i seguenti metodi:

   a. Il metodo **public Object findElement(Object key)** della classe **LogFile** che restituisce l'elemento associato alla chiave **key**. Le chiavi devono essere confrontate tramite l'istanza **EQ** di **EqualityTester.** Quale è la complessità del metodo proposto? Giustificare la risposta.

   b. Il metodo **boolean isEqualTo(Object x, Object y)** della classe **StringEqualityTester** che implementa un l'interfaccia **EqualityTester** per chiavi tipo stringa. Si assuma che due chiavi sono uguali se sono identiche oppure se sono della stessa lunghezza.

4. Aggiungere alla classe **D** che implementa l'interfaccia **Dictionary** il metodo **Iterator deleteAllElements(Object key)** che cancella dal dizionario tutti gli elementi che hanno chiave uguale a **key**. Il metodo restituisce un iteratore su tutte le voci cancellate dal dizionario. Se non ci sono elementi con con chiave uguale a **key**, allora viene lanciata l'eccezione **NoSuchKey**. Quale è la complessità del metodo proposto (giustificare la risposta).

5. Aggiungere alla classe **LinearProbingHTable**, che implementa l'interfaccia **Dictionary** con una tabella hash in cui le collisioni sono risolte con il linear probing, il seguente metodo:
   public int[] removeEltsWithKey(Object k)
   che rimuove tutti gli elementi con chiave **k** e restituisce un array di lunghezza pari alla dimensione del dizionario e contenente nelle prime locazioni gli indici delle entrate della tabella che si sono liberate. Se la tabella non contiene elementi con chiave **k** allora l'array deve contenere -1 nella prima locazione. Il metodo deve lanciare **InvalidKeyException** se **k** non è una chiave valida.

Priority Queue

1. Aggiungere alla classe **UnsortedSequencePriorityQueue** (implementa l'interfaccia **PriorityQueue** usando un'istanza della classe **NodeSequence**) il metodo
   ObjectIterator getInterval(Object k_1**, Object** k_2**)**
   due chiavi k_1 e k_2, con k_1 minore di k_2, restituisca un iteratore sugli elementi di Q che hanno chiave k tale che $k_1 \leq k \leq k_2$.
   a. L'elemento di Q a priorità minima è quello che ha come chiave la stringa più corta
   b. Si supponga che **Comp** sia il comparatore utilizzato per confrontare le chiavi nella coda a priorità
2. Mostrare come implementare il tipo di dati astratto **Stack** usando solo una coda a priorità. (Scrivere la classe **StackPQ** che implementa l'interfaccia Stack usando come sola variabile d'istanza una variabile di tipo **PriorityQueue**).

3. Scrivere una funzione Java **PQ-Sort** che ricevuti in input una sequenza ed un comparatore restituisce in output una sequenza ordinata. La funzione **PQ-Sort**, per l'ordinamento, deve far uso di una coda a priorità. Discutere della complessità di tempo della funzione.

4. Aggiungere alla classe **P** che implementa l'interfaccia **PriorityQueue** il metodo
   Object decreaseKey(Object elem, Object NewKey).
   Se il valore di **NewKey** e` inferiore a quello della chiave associata ad **elem** allora il metodo pone il valore della chiave associata ad **elem** uguale a **NewKey** e restituisce **elem;** in caso contrario il metodo non effettua alcun aggiornamento e restituisce **null**. Il metodo deve essere scritto senza fare assunzioni sul modo in cui la classe P implementa **PriorityQueue.**

5. Scrivere la funzione
   void updateKey(PriorityQueue Q, Object oldKey, Object newKey).
   La funzione deve aggiornare la chiave di tutti gli elementi della coda a priorità **Q** con chiave **oldKey** al valore indicato da **newKey**. La funzione deve utilizzare esclusivamente i metodi nell'interfaccia **PriorityQueue** ad eccezione del metodo **decreaseKey**. Si osservi che la funzione può modificare il contenuto di **Q** durante la sua esecuzione. Analizzare la complessità di tempo della funzione proposta.

6. Aggiungere alla classe **MyPriorityQueue** che implementa il TDA **PriorityQueue** il metodo
   Iterator removeLargestElts(int m)
   che rimuove gli **m** elementi con chiave più grande dalla coda a priorità e restituisce un iteratore degli elementi rimossi. Si tenga presente che l'elemento con priorità massima è l'elemento con minimo valore della chiave. La coda a priorità non deve essere dall'esecuzione del metodo.

Map

1. Implementare l'interfaccia **Map** utilizzando come variabile d'istanza un variabile di tipo **Vector**.

2. Implementare la seguente interfaccia

```
interface Pull {
        public void add(Objec e);       //aggiunge l'elemento e al contenitore
        public Object remove();         //cancella un elemento arbitrario dal contenitore
        public boolean isEmpty();       //restituisce true se il contenitore è vuoto
        public int size();              //restituisce il numero di elementi nel contenitore
}
```

   Tutti i metodi devono avere una complessità pari a $O(1)$, giustificare la risposta.

Tree

1. Implementare l'interfaccia **Tree** (scrivere il codice della classe **LinkedTree**) usando la classe **TreeNode** quale implementazione dell'interfaccia **Position**. (Il codice di **TreeNode** è sul sito del corso).

2. Testare l'implementazione di **LinkedTree** scrivendo un programma **TestNLinkedTree** che utilizzi tutti i metodi dell'interfaccia **Tree**

3. Modificare il metodo **getChildren** di **TreeNode** in modo da fargli restituire un iteratore ai figli del nodo.

4. Aggiungere alla classe **LinkedTree** il metodo **int depth(v)** che restituisce la profondità del nodo **v**

5. Aggiungere alla classe **LinkedTree** il metodo **int height()** che restituisce l'altezza dell'albero.

6. L'arità (arity) di un albero è il numero massimo di figli di un nodo dell'albero. Aggiungere alla classe **LinkedTree** il metodo **int arity()**.

7. Implementare l'interfaccia **Tree** (scrivere il codice della classe **LCRSTree**) usando la classe **LCRSNode** quale implementazione dell'interfaccia **Position.** (Il codice di **LCRSNode** è sul sito del corso).

8. Testare l'implementazione di **LCRSTree** scrivendo un programma **TestLCRSTree** che utilizzi tutti i metodi dell'interfaccia **Tree**

9. Valutare le differenze della complessità di tempo dei metodi implementati in **LinkedTree** e **LCRSTree**

10. Aggiungere alla classe **LinkedTree** (implementa l'interfaccia **Tree** usando come nodo la classe **TreeNode**) il metodo **ObjectIterator nodes**(int k) che restituisce in output un iteratore sui nodi dell'albero che hanno grado al più k (hanno al più k figli).

11. Aggiungere alla classe **LinkedTree** (implementa l'interfaccia **Tree** usando come nodo la classe**TreeNode**) il metodo **int numNodes(TreeNode a, TreeNode b)** che restituisca in output il numero di nodi nella path dal nodo **a** al nodo **b**. Il metodo deve restituire –1 se il nodo **a** non è antenato del nodo **b** o viceversa . Dire quale sia la complessità del metodo proposto nel caso peggiore, motivando la risposta.

12. Aggiungere alla classe **LinkedTree** (implementa l'interfaccia **Tree** usando come nodo la classe **TreeNode**) il metodo: **CountNodes**() che restituisce il numero di nodi dell'albero che hanno almeno tre figli.

13. Aggiungere alla classe **LinkedTree** (implementa l'interfaccia **Tree** usando come nodo la classe **TreeNode**) il metodo **boolean isAncestor(TreeNode n1, TreeNode n2)** che restituisca inoutput *true* qualora **n1** sia antenato di **n2**, false altrimenti. Dire quale sia la complessità del metodo proposto nel caso peggiore, motivando la risposta.

14. Aggiungere alla classe **LinkedBinaryTree** (implementa l'interfaccia **BinaryTree** usando come nodo la classe **BTNode**) il metodo **ObjectIterator leaves**() che restituisce in output un iteratore sulle foglie dell'albero

15. Si scriva la funzione **Iterator sameDepth(Tree t, int k)** che ricevuti in input un albero **T** ed un intero **k**, restituisca in output un iteratore su tutti i nodi di **T** che si trovano a profondità **k**. Per l'implementazione non è possibile utilizzare i metodi **elements()** e **positions().** Quale è la complessità del metodo proposto (giustificare la risposta).

16. Aggiungere alla classe **LinkedTree** il metodo **int [] conta()** che restituisce in output un array **A** di interi contenente nella posizione **i** il numero di nodi dell'albero che hanno un numero di figli pari ad **i** (in **A[i]** deve essere memorizzato il numero di nodi dell'albero che hanno **i** figli). Per l'implementazione non è possibile utilizzare i metodi **elements()** e **positions().** Quale è la complessità del metodo proposto? Giustificare la risposta.

17. Aggiungere alla classe **LinkedTree** il metodo **int arity()** che restituisce l'arità dell'albero. L'arità (arity) di un albero è il numero massimo di figli di un nodo dell'albero. Quale è la complessità del metodo proposto (giustificare la risposta).

18. Aggiungere alla classe **LinkedTree** (implementa l'interfaccia **Tree** usando come nodo la classe **TreeNode**) il metodo: **allLeaves**() che restituisce un iteratore sugli elementi contenuti nelle foglie.

19. Scrivere la funzione

    public Object max(BinaryTree T, Comparator c)

    che prende in input un albero binario e un comparatore e restituisce l'elemento più grande dell'albero. I confronti tra gli elementi dell'albero devono essere effettuati mediante il comparatore **c**.

20. Aggiungere alla classe **LinkedTree** il metodo **Iterator preorderVisit(**) che restituisce un iteratore sugli elementi dell'albero elencati secondo la visita preorder dell'albero. Quale è la complessità del metodo proposto (giustificare la risposta).

21. Aggiungere alla classe **LinkedTree** il metodo **Iterator postorderVisit(**) che restituisce un iteratore sugli elementi dell'albero elencati secondo la visita postorder dell'albero. Quale è la complessità del metodo proposto (giustificare la risposta).

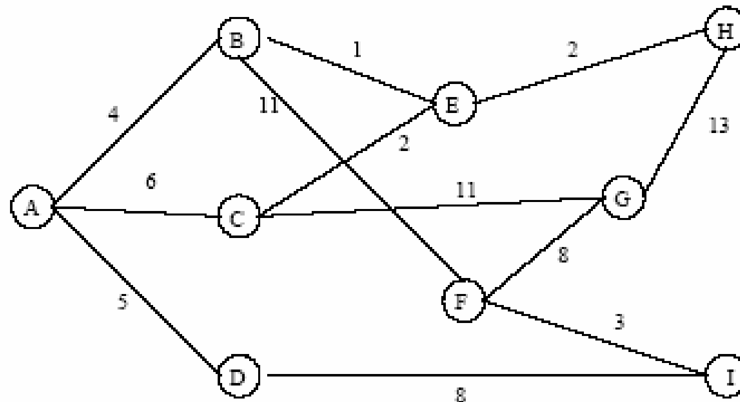
22. Aggiungere alla classe **LinkedTree** il metodo **int height(TreeNode v)** che restituisce l'altezza del sottoalbero avente per radice **v**.

23. Aggiungere il metodo **boolean balanced(BTNode v) a**lla classe **LinkedBinaryTree** che verifichi se il sottoalbero avente radice **v** sia perfettamente bilanciato o meno. Per semplicità si assuma la seguente definizione di albero binario perfettamente bilanciato: tutti i livelli (compreso quello delle foglie) devono essere completi.

24. Aggiungere il metodo **boolean balanced()** alla classe **BinTree** che verifichi se l'albero binario sia perfettamente bilanciato o meno. Per semplicità si assuma la seguente definizione di albero binario perfettamente bilanciato: tutti i livelli (compreso quello delle foglie) devono essere completi.

Grafi

1. Si consideri il grafo pesato in figura



   Rappresentare il grafo sia mediante liste delle adiacenze (nelle liste i nodi vengono inseriti in ordine lessicografico) sia mediante matrice delle adiacenze.

2. Sia $G = (V,E)$ un grafo orientato con $n = |V|$ vertici ed $m = |E|$ archi. Si supponga che puntatori e interi richiedano 32 bit. Indicare per quali valori di n ed m la lista delle adiacenze richiede meno memoria della matrice delle adiacenze.

3. Il trasposto di un grafo orientato $G = (V,E)$ è il grafo $G^T = (V,E^T)$ in cui $E^T = \{(u,v) : (v,u) \in E\}$. Descrivere due algoritmi efficienti per calcolare G^T con la rappresentazione con liste delle adiacenze e con la rappresentazione con matrice delle adiacenze.

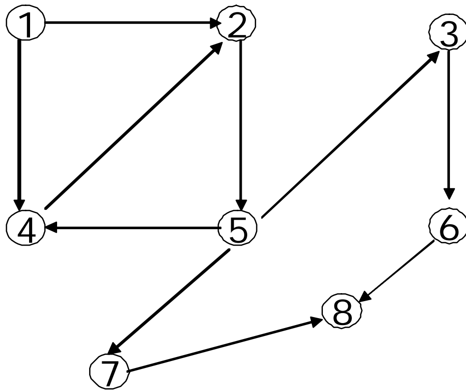
4. Data una rappresentazione con liste di adiacenza di un grafo orientato, quanto tempo occorre per calcolare: il grado uscente di ogni vertice (numero di archi uscenti dal vertice); il grado entrante di ogni vertice (numero di archi entranti nel vertice). Cambia il tempo necessario per risolvere il problema precedente se il grafo non è orientato? Se si, come?

5. La matrice delle incidenze di un grafo orientato G=(V,E) è una matrice $B=(b_{i,j})$ di dimensione |V|X|E| tale che: $b_{i,j} = -1$ se l'arco j esce dal vertice i; $b_{i,j} = 1$ se l'arco j entra nel vertice i; 0 negli altri casi. Descrivere, che cosa rappresentano gli elementi del prodotto matriciale BB^T, dove B^T è la matrice trasposta di B.

6. Il quadrato di un grafo orientato G = (V,E) è il grafo $G^2 = (V,E^2)$ tale che $(u,w) \in E^2$ se e soltanto se, per qualche $v \in V$, si abbia $(u,v) \in E$ e $(v,w) \in E$. Ovvero G^2 contiene un arco fra u e w, se G contiene un cammino con due soli archi fra u e w. Descrivere degli algoritmi efficienti per calcolare G^2 da G, rappresentando G sia con le liste di adiacenze sia con la matrice di adiacenza. Analizzare i tempi di esecuzione degli algoritmi proposti.

7. Data una rappresentazione con liste di adiacenza di un multigrafo G = (V,E), descrivere un algoritmo con tempo O(V + E) per calcolare la rappresentazione con liste di adiacenza del grafo non orientato G' = (V,E'), dove E' è formato dagli archi di E con tutti gli archi multipli fra due vertici distinti sostituiti da un singolo arco e con tutti i cappi rimossi. Per la soluzione, è possibile utilizzare una struttura di appoggio.

8. Si consideri il grafo in figura



Rappresentare il grafo mediante lista delle adiacenze (in ciascuna lista i nodi vengono inseriti in ordine crescente) e mediante matrice delle incidenze.

9. Scrivere lo pseudocodice dell'algoritmo per la visita in ampiezza di un grafo (BFS). Commentare l'uso delle strutture dati utilizzate.

10. Indicare, giustificando la risposta, la complessità dell'algoritmo BFS.

11. Data la rappresentazione mediante lista delle adiacenze del grafo nell'esercizio al punto 8, disegnare l'albero BFS avente come radice il nodo 1.

12. Provare il seguente lemma "In un grafo pesato, indicando con $\delta(s,v)$ il minimo numero di archi in un cammino da s a v si ha che per ogni $(u,v) \in E$ $\delta(s,v) \leq \delta(s,u) + 1$ "

13. Provare il seguente lemma "Dopo che la BFS è terminata, si ha che per ogni $v \in V$ $d[v] \geq \delta(s,v)$ ".

14. Qual è il tempo di esecuzione della procedura BFS se il suo grafo di input è rappresentato da una matrice delle adiacenze e l'algoritmo viene modificato per gestire questa forma di input?
    a. Scrivere l'algoritmo modificato
    b. Calcolare il tempo di esecuzione dell'algoritmo giustificando la risposta

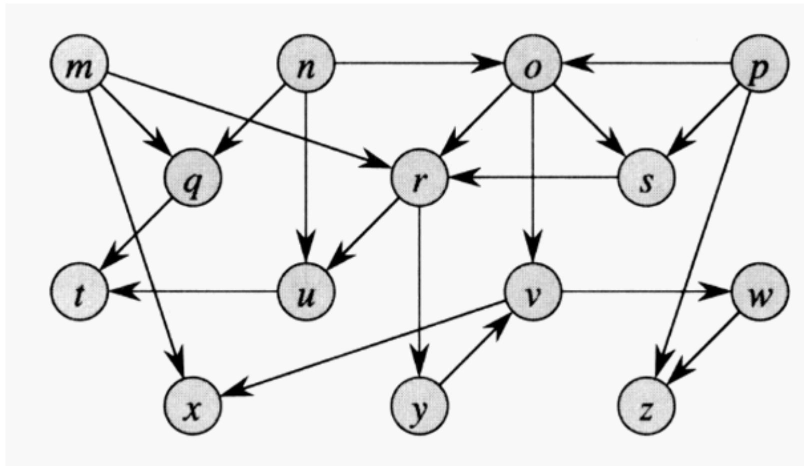
15. Scrivere lo pseudocodice dell'algoritmo per la visita in profondità di un grafo (DFS). Commentare l'uso delle strutture dati utilizzate.

16. Indicare, giustificando la risposta, la complessità dell'algoritmo DFS.

17. Data la rappresentazione mediante lista delle adiacenze del grafo nell'esercizio al punto 8, disegnare l'albero DFS avente come radice il nodo 1.

18. Scrivere lo pseudocodice di un algoritmo per il calcolo di tutte le componenti connesse di un grafo non orientato (odificare opportunamente una delle visite analizzate - BFS o DFS. Discutere della complessità dell'algoritmo proposto e delle strutture dati utilizzate.

19. Eseguire l'ordinamento topologico sul seguente DAG.



20. Illustrare l'algoritmo di Prim per risolvere il problema del Minimo Albero Ricoprente. Servirsi dello pseudocodice e di commenti relativi alle strutture dati utilizzate. Quale è la complessità dell'algoritmo? Giustificare la risposta.

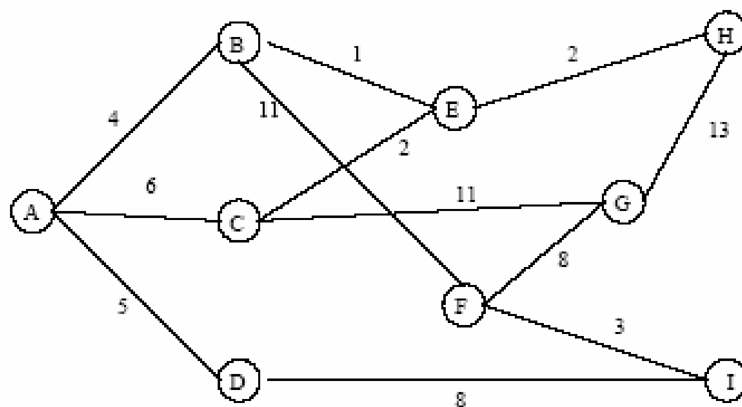
21. Provare il seguente teorema. Se si fa riferimento ad altri lemmi e/o corollari è necessario enunciarli.

    Teorema:

    Sia G=(V,E) un grafo connesso non direzionato con una funzione di peso *w* a valori reali definita su E. Sia A un sottoinsieme di E che è incluso in un minimo albero ricoprente per G, sia (S, V-S) un qualsiasi taglio di G che rispetta A, e sia (u,v) un arco leggero (*light edge*) che attraversa (S, V-S). Allora, l'arco (u,v) è un arco sicuro (*safe edge*) per A.

22. Scrivere lo pseudocodice dell'algoritmo di Dijkstra. Quale è la sua complessità? Giustificare la risposta.

23. Si consideri il grafo pesato in figura



Rappresentare il grafo mediante liste delle adiacenze (nelle liste i nodi vengono inseriti in ordine lessicografico) ed eseguire l'algoritmo di Dijkstra su di esso per il calcolo dei cammini minimi, a partire dal nodo A. Si riporti per ogni ciclo principale dell'algoritmo: il contenuto dell'insieme Q dei nodi non ancora visitati con le relative priorità; il contenuto dell'insieme R dei nodi già visitati; il vettore delle distanze.

24. Provare il seguente teorema. Se si fa riferimento ad altri lemmi e/o corollari è necessario enunciarli.
    Teorema: Se f è un flusso in una rete di flusso G con sorgente s e destinazione t, allora le seguenti condizioni sono equivalenti:
    1. f è un massimo flusso in G
    2. G_f non contiene augmenting path
    3. $|f| = c(S,T)$ per un taglio (S,T)

25. Descrivere un algoritmo per individuare il minimo taglio (S,T) di una rete di flusso. Fornire lo pseudo-codice dell'algoritmo proposto indicando la strategia adottata per computare i cammini aumentanti. Analizzare la complessità dell'algoritmo proposto.

26. Descrivere le quattro versioni del problema dei cammini minimi in un grafo orientato (SSSP, SDSP, SPSP, APSP).

27. Mostrare, come un algoritmo utilizzato per risolvere il problema SSSP possa essere utilizzato per risolvere i problemi SDSP, SPSP, APSP.

28. Scrivere lo pseudocodice dell'algoritmo per la visita in ampiezza di un grafo (BFS). Commentare l'uso delle strutture dati utilizzate. Indicare inoltre, giustificando la risposta, la complessità dell'algoritmo BFS.

29. Provare il seguente teorema. Se si fa riferimento ad altri lemmi e/o corollari è necessario enunciarli.
    Sia G=(V,E) un grafo connesso non direzionato con una funzione di peso *w* a valori reali definita su E. Sia A un sottoinsieme di E che è incluso in un minimo albero ricoprente per G, sia (S, V-S) un qualsiasi taglio di G che rispetta A, e sia (u,v) un arco leggero (*light edge*) che attraversa (S, V-S). Allora, l'arco (u,v) è un arco sicuro (*safe edge*) per A.

30. Codificare l'algoritmo di Prim in Java (scrivere una funzione Java che implementi l'algoritmo). Il codice deve far riferimento alle interfacce illustrate durante il corso.

31. Codificare l'algoritmo di Dijkstra in Java (scrivere una funzione Java che implementi l'algoritmo). Il codice deve far riferimento alle interfacce illustrate durante il corso.

32. Codificare l'algoritmo BFS in Java (scrivere una funzione Java che implementi l'algoritmo). Il codice deve far riferimento alle interfacce illustrate durante il corso.

33. Codificare l'algoritmo DFS in Java (scrivere una funzione Java che implementi l'algoritmo). Il codice deve far riferimento alle interfacce illustrate durante il corso.