I Appello – 23 Gennaio 2017

Esercizio 1

Si consideri la seguente implementazione della struttura dati di albero binario bilanciato:

```
public interface Albero{
  Albero add(String s);          // ritorna l'albero con un nodo in piu' con s nel campo info
  void stampa();                 // stampa inline dell'albero
  boolean presente(String s);    // ritorna true se s e' presente in un nodo
  int nNodi();                   // numero di nodi, cioe' di stringhe, contenuti nell'albero
}

public interface Iteratore {
    boolean hasNext();      // dice se c'e' un prossimo elemento su cui iterare
    String next();          // restituisce la stringa contenuta nel prossimo elemento
                            // e sposta avanti l'iteratore
}

class AlberoVuoto implements Albero {
    public Albero add(String s){
      return new AlberoImpl(s,new AlberoVuoto(),new AlberoVuoto());
    }
    public void stampa(){ System.out.print("- "); }
    public boolean presente(String s){return false;}
    public int nNodi(){ return 0; }
}

class AlberoImpl implements Albero {
    private String info;
    private Albero figlioSin;
    private Albero figlioDx;

    AlberoImpl(String s){ info=s; figlioSin=new AlberoVuoto(); figlioDx=new AlberoVuoto(); }
    AlberoImpl(String s, Albero fs, Albero fd){ info=s; figlioSin=fs; figlioDx=fd; }

    public int nNodi(){ return figlioSin.nNodi() + figlioDx.nNodi() + 1; }

    public Albero add(String s){
        int ns=figlioSin.nNodi();
        int nd=figlioDx.nNodi();
        if(ns > nd) figlioDx=figlioDx.add(s);
        else figlioSin=figlioSin.add(s);
        return this;
    }

    public void stampa(){
      figlioSin.stampa(); System.out.print(info+" "); figlioDx.stampa();
    }
    public boolean presente(String s){
      return (s.equals(info) || figlioSin.presente(s) || figlioDx.presente(s));
    }
    ...
}
```

```
class Main {
 public static void main(String[] arg){
  Albero a=new AlberoImpl("pippo");
  a.add("pluto").add("paperino").add("minnie").add("topolino").add("gastone").add("paperone");
  a.stampa(); // - minnie - pluto - gastone - pippo - topolino - paperino - paperone -
    ....
} }
```

(A) Completare il metodo `main` in modo tale che avvii 3 thread concorrenti `t1, t2, t3` rispettando le seguenti specifiche:

- il thread `t1` aggiunge all'albero `a` la stringa `qui`, poi fa passare un po' di tempo, poi aggiunge all'albero `a` la stringa `quo` ed infine stampa l'albero.
- il thread `t2` controlla se `quo` è presente nell'albero `a` e stampa `BIANCO` in caso positivo o `NERO` in caso negativo, dopodiché fa passare un po' di tempo e poi stampa la stringa `ROSSO`.
- il thread `t3` controlla se `qui` è presente nell'albero `a` e stampa `UNO` in caso positivo o `DUE` in caso negativo, dopodiché fa passare un po' di tempo e poi stampa la stringa `TRE`.
- l'aggiunta di una stringa all'albero non deve interferire con gli accessi concorrenti in lettura all'albero. **Giustificare brevemente** perché il codice scritto rispetta questo vincolo.
- le operazioni di *test-and-set* effettuate dai thread `t2, t3` devono essere corrette: ad esempio, quando `t2` stampa `NERO` non è possibile che nel frattempo `t1` abbia inserito la stringa `quo`. **Giustificare brevemente** perché il codice scritto rispetta questo vincolo.
- le due stampe dei colori effettuate da `t2` devono essere mutuamente esclusive rispetto alle due stampe dei numeri effettuate da `t3` ma possono essere concorrenti rispetto alla stampa dell'albero a effettuata dal thread `t1`. Ad esempio l'output `NERO ...stampa albero... ROSSO UNO TRE` è ammissibile mentre `BIANCO UNO TRE ROSSO ...stampa albero...` non lo è. **Giustificare brevemente** perché il codice scritto rispetta questo vincolo.
- **ATTENZIONE: non è possibile modificare la definizione della classe `AlberoImpl`** ma **SOLO** scrivere codice all'interno del metodo `main`.

(B) Aggiungere alla sola classe `AlberoImpl` il metodo `public Iteratore itera()` che restituisce un iteratore sull'albero non vuoto fatto in modo tale per cui le successive chiamate del suo metodo `next()` restituiscano le stringhe contenute nel cammino più a sinistra dell'albero di invocazione.
Aggiungere al `main` il codice di invocazione del metodo `itera()` sull'albero `a` ed il codice che ne effettua l'iterazione completa producendo in output la sequenza `pippo pluto minnie qui`, che corrisponde al cammino più a sinistra dell'albero `a`.

(C) Scrivere una versione distribuita dell'applicazione scritta nel punto (A) rispettando le seguenti specifiche:

- L'albero `a` deve essere un oggetto di tipo `Remote`. Una macchina Server crea l'albero `a`, lo riempie, ne pubblica il riferimento sul registro RMI e poi realizza quello che faceva il thread `t1` del punto (A).
- I thread `t2, t3` descritti nel punto (A) devono essere avviati su una macchina Client che prima di tutto recupera il riferimento all'albero remoto `a`.
- Per quanto riguarda i vincoli di comportamento descritti nel punto (A), in questa versione distribuita è **possibile modificare il codice della classe `AlberoImpl`**. In particolare devono essere rispettate le seguenti specifiche:
  - l'aggiunta di una stringa all'albero non deve interferire con gli accessi concorrenti (e remoti) in lettura all'albero. **Giustificare brevemente** perché il codice scritto rispetta questo vincolo.
  - **non è necessario** che le operazioni di *test-and-set* effettuate dai thread `t2, t3` siano corrette: ad esempio, quando `t2` stampa `NERO` è possibile che nel frattempo il server abbia inserito la stringa `quo`. **Indicare se e perché** il codice scritto permette che in qualche esecuzione le operazioni di *test-and-set* non siano corrette.
  - le due stampe dei colori effettuate da `t2` devono essere mutuamente esclusive rispetto alle due stampe dei numeri effettuate da `t3` ma possono essere concorrenti rispetto alla stampa dell'albero `a` effettuata dal thread `t1`. **Giustificare brevemente** perché il codice scritto rispetta questo vincolo.
- Descrivere **TUTTE le possibili stampe** prodotte dall'applicazione distribuita realizzata.