Esercizio II Parziale 2022-2023

Esercizio 1

Realizzare un metodo costruttore della classe LinkedBinaryTree<E>

public LinkedBinaryTree (List<E> list)

che prende in input una lista di oggetti di tipo E e costruisce un nuovo albero binario come una catena lineare tale che ogni nodo ha esattamente un figlio e l'oggetto in posizione i nella lista è collocato nel nodo di livello i dell'albero. In fase di costruzione dell'albero, si assuma che per ogni nodo la probabilità di avere un figlio sinistro (rispettivamente destro) sia ½.

```
public LinkedBinaryTree(List<E> list){

        root = new Node<E>(list.get(0));
        Random rand = new Random();
        int x;
        Node<E> current = root;
        //se x è 0 figlio sinistro altrimenti destro
        for(int i = 1; i < list.size(); i++){
            x = rand.nextInt(2);
            if(x == 0){

                current.setLeft(new Node<E>(list.get(i)));
                current = current.left;

            }
            else{
                current.setRight(new Node<E>(list.get(i)));
                current = current.right;
            }

        }

    }
```

Esercizio 2

Implementare la classe UndirectedUnweightedNetwork<Vertex> come sottoclasse della classe UnweightedNetwork<Vertex> per rappresentare grafi non orientati e non pesati.

```java
public class UndirectedUnweightedNetwork<Vertex> extends UnweightedNetwork<Vertex>{
    @Override
    public boolean equals (Object obj)
    {
        if ((obj == null) || !(obj instanceof UndirectedUnweightedNetwork<?>))
            return false;
        UndirectedUnweightedNetwork<?> other = (UndirectedUnweightedNetwork<?>)obj;
        return adjacencyMap.equals (other.adjacencyMap);
    }
    @Override
    public int edgeSize(){
        return super.edgeSize()/2;
    }
    @Override
    public boolean addEdge (Vertex v1, Vertex v2)
    {
        return super.addEdge(v1,v2) &&  super.addEdge(v2,v1);
    }

    public boolean removeEdge(Vertex v1, Vertex v2){
        return super.removeEdge(v1,v2) && super.removeEdge(v2,v1);
    }
    public boolean isConnected()
    {
            // Count the vertices reachable from v.
            Vertex v = adjacencyMap.firstKey();
            Iterator<Vertex> itr = new BreadthFirstIterator (v);
            int count = 0;
            while (itr.hasNext())
            {
                itr.next();
                count++;
            } // while
            if (count < adjacencyMap.size())
                return false;
            return true;
    }
}
```

I Appello 2022-2023

Esercizio 3

Aggiungere alla classe UndirectedNetwork<> un metodo

public UndirectedNetwork<Vertex> complementary()

che restituisce una nuova istanza di tipo UndirectedNetwork<Vertex> che rappresenta il grafo

“complementare” al grafo corrente (assegnare peso unitario a tutti gli archi).

Dato un grafo G = (V, E), il uso grafo complementare ha gli stessi nodi di G, ma contiene solo gli archi che non sono presenti in G.

```
public UndirectedNetwork<Vertex> complementary() {
         UndirectedNetwork<Vertex> compGraph = new UndirectedNetwork<Vertex>();
         TreeMap<Vertex,Double> NMap;
         Set<Vertex> V = adjacencyMap.keySet(); //vertici del grafo
         //creazione grafo completo
         for (Vertex v: V) {
            compGraph.adjacencyMap.put(v, new TreeMap<Vertex,Double>());
            NMap = compGraph.adjacencyMap.get(v);
            for (Vertex x: V) NMap.put(x, 1.0);
         }
         //cancellazione degli archi contenuti in this
         for (Vertex v: V) {
            NMap = compGraph.adjacencyMap.get(v);
            for (Vertex x: adjacencyMap.get(v).keySet()) NMap.remove(x);
         }
         return compGraph;
        }
```

I Appello 2021-2022

Esercizio 2

Realizzare il metodo statico

public static <E> void printLeafs(BinaryNode<E> root)

che stampa il contenuto di tutte le foglie dell’albero binario radicato in root,

seguendo l’ordine da destra a sinistra.

```
public static <E> void recursivePrintLeafs(BinaryNode<E> root){
        if(!(root.hasLeft() || root.hasRight()))
            System.out.println(root.getData());
        if(root.hasRight())
            recursivePrintLeafs(root.right);
        if(root.hasLeft())
            recursivePrintLeafs(root.left);
    }
```

Esercizio 3

Aggiungere alla classe Network<> un nuovo metodo public Network<Vertex> transpose() che restituisce una nuova istanza di tipo Network<Vertex> che rappresenta il grafo trasposto al grafo corrente.

Si ricorda che dato un grafo G = (V, E), il uso grafo trasposto Gt = (V, Et) ha gli stessi nodi di G e gli archi orientati in senso opposto, i.e., Et = {(v, u) | (u, v) ∈ E}.

```
public Network<Vertex> transpose(){
            Network<Vertex> tran;
            for(Vertex v : this.adjacencyMap.keySet())
                tran.addVertex(v);

            for(Vertex v : this.adjacencyMap.keySet()){
                for(Map.Entry<Vertex,Double> u : adjacencyMap.get(v).entrySet())
                    addEdge(u.getKey(),v,u.getValue());
            }
            return tran;
        }
```

V Appello 2022-2023

Esercizio 2

Realizzare un metodo ricorsivo interno alla classe LinkedBinaryTree<> che calcola

l'altezza dell'albero binario corrente.

```
public int recursiveHeight(Nodo<E> nodo){
      if(nodo == null)
        return 0;
      if(!(nodo.hasLeft() || nodo.hasRight()))
        return 0;
      return Math.max(recursiveHeight(nodo.left), recursiveHeight(nodo.right)) + 1;
    }
```