Homework format:

- Name your file HW2.hs and include the first line:
  `module HW2 where`
- Use the the tree and graph data types in the file HW2types.hs by including the line
  `import HW2types`
- To receive full credit your code must be commented including a header with your name & date.

Problem 1: Trees

You will use the data type below from HW2Types.hs for this problem. Assume there are no duplicate values in a tree that is no two nodes have the same value.

```
data Tree = Node Int Tree Tree | Leaf
    deriving Show
```

a) Write a Haskell function called **sizeTree** that takes as input a tree and returns an integer that represents the number of nodes in the tree. Assume the size of a Leaf is 0. For example:

```
*Main> sizeTree Leaf
0
*Main> t1
Node 4 (Node 3 (Node 2 (Node 1 Leaf Leaf) Leaf) Leaf) Leaf
*Main> sizeTree t1
4
*Main> t3
Node 4 (Node 2 Leaf Leaf) (Node 10 (Node 9 (Node 7 Leaf Leaf) Leaf) Leaf)
*Main> sizeTree t3
5
*Main> sizeTree (Node 1 Leaf Leaf)
1
```

b) Write a Haskell function called **height** that takes as input a tree and returns an integer that represents the height of the tree. Define the height of a tree to be the number of edges in the longest path from root to leaf. Assume the height of a Leaf is -1 and the height of a singleton is 0.

```
*Main> height (Node 5 (Node 3 Leaf Leaf) (Node 4 Leaf Leaf))
1
*Main> height (Node 4 Leaf Leaf)
0
*Main> height Leaf
-1
*Main> t2
Node 1 Leaf (Node 2 Leaf (Node 3 Leaf (Node 4 Leaf Leaf)))
*Main> height t2
3
*Main> t3
Node 4 (Node 2 Leaf Leaf) (Node 10 (Node 9 (Node 7 Leaf Leaf) Leaf) Leaf)
*Main> height t3
3
```

c) Write a Haskell function called **treeSum** that takes as input a tree and returns an integer that represents the sum of all nodes in the tree. Assume the treeSum of a Leaf is 0.

```
*Main> treeSum Leaf
0
*Main> treeSum t1
10
*Main> t1
Node 4 (Node 3 (Node 2 (Node 1 Leaf Leaf) Leaf) Leaf) Leaf
*Main> treeSum t3
32
*Main> t3
Node 4 (Node 2 Leaf Leaf) (Node 10 (Node 9 (Node 7 Leaf Leaf) Leaf) Leaf)
```

d) Overload the equivalence (==) operator to return True if two trees contain the same values and False otherwise. Assume two Leaf are equivalent.

Use: instance Eq Tree where

(==) …….

```
*Main> Leaf == Leaf
True
*Main> Leaf == (Node 1 Leaf Leaf)
False
*Main> t1
Node 4 (Node 3 (Node 2 (Node 1 Leaf Leaf) Leaf) Leaf) Leaf
*Main> t2
Node 1 Leaf (Node 2 Leaf (Node 3 Leaf (Node 4 Leaf Leaf)))
*Main> t1 == t2
True
*Main> (Node 3 Leaf (Node 5 Leaf Leaf)) == (Node 5 (Node 3 Leaf Leaf) Leaf)
True
*Main> (Node 3 Leaf (Node 5 Leaf Leaf)) == (Node 7 (Node 3 Leaf Leaf) Leaf)
False
```

e) Write a function called **mergeTrees** that takes as input a two trees t1 and t1 and returns a tree t3 containing all of the values in t1 and t2 .

```
*Main> t1
Node 5 (Node 2 Leaf Leaf) (Node 8 (Node 6 Leaf Leaf) Leaf)
*Main> t2
Node 4 (Node 3 Leaf Leaf) (Node 7 Leaf (Node 10 (Node 9 Leaf Leaf) Leaf))
*Main> let t3 = mergeTrees t1 t2
*Main> t3
Node 5 (Node 2 Leaf (Node 4 (Node 3 Leaf Leaf) Leaf)) (Node 8 (Node 6 Leaf (Node 7 Leaf Leaf))
(Node 10 (Node 9 Leaf Leaf) Leaf))
```

```
*Main> mergeTrees tree2 tree4
Node 6 (Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf (Node 4 Leaf Leaf))) (Node 8 Leaf Leaf)
*Main> mergeTrees tree1 tree2
Node 5 (Node 2 Leaf (Node 3 Leaf Leaf)) (Node 6 Leaf (Node 8 Leaf Leaf))
*Main> mergeTrees (Node 5 Leaf Leaf) (Node 5 Leaf Leaf)
Node 5 Leaf Leaf
*Main> mergeTrees (Node 5 Leaf Leaf) Leaf
Node 5 Leaf Leaf
*Main> mergeTrees Leaf Leaf
Leaf
```

f) Write a function called **isBST** that takes a tree t as input and returns True if the tree is a binary search tree and False otherwise. If you want to review the definition of a binary search tree BST reference this page https://en.wikipedia.org/wiki/Binary_search_tree. *Note: isBST Leaf = True.*

```
*Main> tree1
Node 5 Leaf Leaf
*Main> isBST tree1
True
*Main> tree2
Node 8 (Node 6 Leaf Leaf) (Node 2 (Node 3 Leaf Leaf) (Node 7 Leaf Leaf))
*Main> isBST tree2
False
*Main> tree3
Node 8 (Node 2 Leaf (Node 3 Leaf Leaf)) (Node 10 (Node 9 Leaf Leaf) Leaf)
*Main> isBST tree3
True
```

g) Write a Haskell function called **convertBST** that takes as input an arbitray tree t and returns a tree t' that is formatted as a binary search tree such that t and t' are equivalent.

```
*Main> tree2
Node 8 (Node 6 Leaf Leaf) (Node 2 (Node 3 Leaf Leaf) (Node 7 Leaf Leaf))
*Main> convertBST tree2
Node 7 (Node 2 Leaf (Node 3 Leaf (Node 6 Leaf Leaf))) (Node 8 Leaf Leaf)
*Main> isBST (convertBST tree2)
True
*Main> tree2 == (convertBST tree2)
True
```

Problem2: Graphs

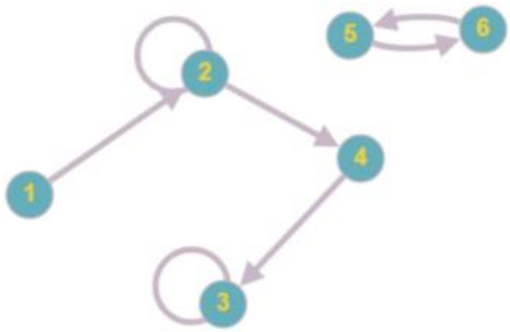
A simple way to represent a directed graph is through a list of edges. An edge is given by a pair of vertices. For simplicity, vertices are represented by integers.

```
type Vertex  = Int
type Edge    = (Vertex,Vertex)
type Graph   = [Edge]
```

(We ignore the fact that an edge list cannot represent isolated vertices without self-loops)

Consider, for example, the following directed graph1.

graph1 = [(1,2),(2,2),(2,4),(4,3),(3,3),(5,6),(6,5)]



a) Write a function called **numVE** that takes as input a graph and returns a tuple that represents the number of vertices and number of edges. For example:

```
*Main> numVE graph1
(6,7)
*Main> numVE []
(0,0)
*Main> numVE [(1,2),(2,3),(3,4),(1,1),(1,3)]
(4,5)
*Main> numVE [(1,1)]
(1,1)
```

b) Write a function called **removeLoops** that removes all self loops from a graph. It takes as input a graph G as input and returns a graph G' which is equivalent to G with the self loops removed.

```
*Main> removeLoops graph1
[(1,2),(2,4),(4,3),(5,6),(6,5)]
*Main> removeLoops [(2,3),(2,5)]
[(2,3),(2,5)]
*Main> removeLoops [(2,3),(2,5), (1,1), (3,5),(3,3)]
[(2,3),(2,5),(3,5)]
*Main> removeLoops [(1,1)]
[]
```

c) Write a function called **removeVertex** that removes a vertex and incident edges from a graph. It takes as input a vertex v and a graph G and returns a graph G' which is equivalent to G without any edges incident to v.

```
*Main> removeVertex 1 graph1
[(2,2),(2,4),(4,3),(3,3),(5,6),(6,5)]
*Main> removeVertex 2 graph1
[(4,3),(3,3),(5,6),(6,5)]
*Main> removeVertex 8 graph1
[(1,2),(2,2),(2,4),(4,3),(3,3),(5,6),(6,5)]
*Main> removeVertex 1 []
[]
*Main> removeVertex 1 [(1,2),(2,1)]
[]
```