Test your coding skill

Write a program to ask the user to enter a first name, a last name and an email. When there is an input error, the program has to stop and display appropriate error.

Below are possible errors:

- First name is empty
- Last name is empty
- Email in wrong format

```
Enter first name:
Error: First name must not be empty
```

```
Enter first name: John
Enter last name:
Error: Last name must not be empty
```

```
Enter first name: John
Enter last name: Smith
Enter email: blah
Error: Invalid email
```

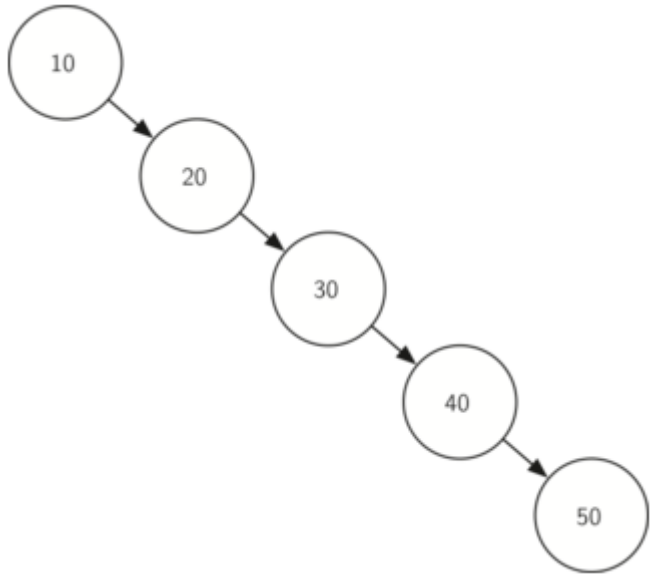
```
Enter first name: Green
Enter last name: Frog
Enter email: frog@pond.com
Thank you for your input
```

Binary Search Tree: Design

- TreeNode Class:
  - hasLeftChild, hasRightChild
  - isLeftChild, isRightChild
  - isRoot, isLeaf
  - hasAnyChildren, hasBothChildren
  - spliceOut
  - findSuccessor, findMin
  - replaceNodeData
- BinarySearchTree Class
  - length, __len__
  - put, _put, __setitem__
  - gut, _gut, __getitem__, __contains__
  - delete, __delitem__, remove

Binary Search Tree Analysis

- The search algorithm computation complexity is implicitly related to the performance of the put Method.
- A perfectly balanced tree has the same number of nodes in the left subtree as the right subtree.
- Hence, the worst-case performance of put is O(log_2 n), where n is the number of nodes in the tree.
- When adding new node, the maximum number of comparisons need in put method is log_2 n
- Nevertheless, it is possible to construct a search tree that has height n.
- It is simply inserting the keys in sorted order, for example, the figure in this slide.
- Hence, the performance of the put method in this case is is O(n) .



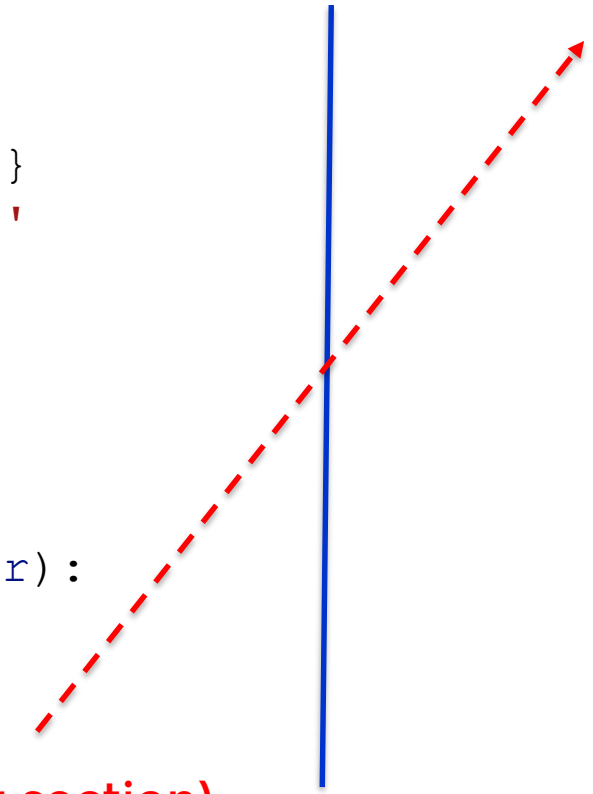
Breadth-First Search Function

We first modify some codes in graph.py

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}
        self.colour = 'white'
        self.distance=0
        self.pred = None

      ....

    def setColour(self,colour):
        self.colour =colour
```



```
def getColour(self):
    return self.colour

def setDistance(self,distance):
    self.distance=distance
def getDistance(self):
    return self.distance
def setPred(self,pred):
    self.pred=pred
def getPred(self):
    return self.pred
```

(Continue the code in the left section)

Breadth-First Search Function

```
from graph.py import Graph, Vertex   # from the previous examples (graph.py)
from queue.py import Queue  # from the previous examples (queue.py)

def bfs(g,start):
  list1 = []
  start.setDistance(0)
  start.setPred(None)
  vertQueue = Queue()
  vertQueue.enqueue(start)
  while (vertQueue.size() > 0):
    currentVert = vertQueue.dequeue()
    for nbr in currentVert.getConnections():
      if (nbr.getColor() == 'white'):
        nbr.setColor('gray')
        nbr.setDistance(currentVert.getDistance() + 1)
        nbr.setPred(currentVert)
        vertQueue.enqueue(nbr)
    currentVert.setColor('black')
    list1.append(currentVert)
  return list1
```

Breadth-First Search Function

```
g = Graph()
for i in range(6):
    g.addVertex(i)
g.addEdge(0,1,5)
g.addEdge(0,5,2)
g.addEdge(1,2,4)
g.addEdge(2,3,9)
g.addEdge(3,4,7)
g.addEdge(3,5,3)
g.addEdge(4,0,1)
g.addEdge(5,4,8)
g.addEdge(5,2,1)
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
start= g.getVertex(0)
list1 = bfs(g,start)
for i in list1:
    print(i)
```

Output:

```
( 0 , 1 )
( 0 , 5 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
0 connectedTo: [1, 5]
1 connectedTo: [2]
5 connectedTo: [4, 2]
2 connectedTo: [3]
4 connectedTo: [0]
3 connectedTo: [4, 5]
```