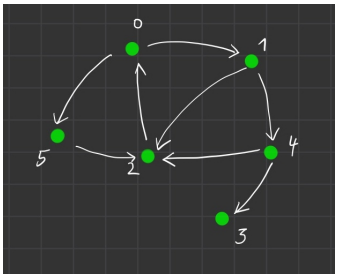
Breitensuche
Betrachten Sie den Graphen im unten angegebenen Bild. Dort finden sich 6 Knoten mit Kanten zu anderen Knoten. Wir codieren den Graphen mit der unten angegebenen Liste von Listen. Der nullte Eintrag der Liste enthält die Elemente 1 und 5, weil der Knoten 0 nach 1 und 5 zeigt und so weiter...

```
[[1, 5],
 [2,4],
 []
 [],
 [3, 2,],
 [2]]
```



Schreiben Sie einen Breitensuche-Algorithmus in Python, der gestartet an dem Knoten 0, einen Weg zum Knoten 3 findet und stellen Sie vor wie er funktioniert.

Hinweis: Arbeiten Sie mit dem Datentyp `list` und verwenden Sie die Methoden `.append()`, `.copy()`, `.pop()`.

Solution:

```python
#!/usr/bin/env python
# coding: utf-8
# <h1>Table of Contents<span class="tocSkip"></span></h1>
# <div class="toc"><ul class="toc-item"></ul></div>
# Code found on:
# https://www.educative.io/edpresso/how-to-implement-a-breadth-first-search-in-python
graph = [
  [1, 5],
  [2, 4],
  [],
  [],
  [3,2],
  [2]
]
def bfs(graph, node):
    """
    Performs a breadth first search and returns the breadth first tree (which is a special
    graph) as list.
    :param visited:
    """
    bfsTree = []     # Stores the breadth first graph
    visited = []  # List to keep track of visited nodes.
                # no node is visited twice!
    queue = []    # Initialize a queue, which collects the next nodes
                # which are visited
    visited.append(node) # The start node has been visited
    queue.append(node)   # The start node is appended to the queue
    while queue: # As long there is something in the queue
        s = queue.pop(0) # We read and delete the first element
        bfsTree.append([].copy()) # We get a new node in the bfs-graph
        for neighbour in graph[s]:
          # We will visit all neighbours of s, unless they have been visited already
          if neighbour not in visited:
            visited.append(neighbour) # The neighbour has now been visited
            queue.append(neighbour)   # and is appended to the queue
```

```
            bfsTree[s].append(neighbour) # Store that we got from s -> neighbbour
    return bfsTree
# Driver Code
bfsTree = bfs(graph, 0)
# The path from 0 to 3 is automatically the shortest path in the given breadth first tree.
# So we just need to find the connection from 0 to 3 in the bfsTree.
# To that end we just follow the given graph and find the correct one.
print("  ", bfsTree[0])
print("   /   | ")
print(bfsTree[bfsTree[0][0]], bfsTree[bfsTree[0][1]])
print(" /  | ")
print(bfsTree[bfsTree[1][0]], bfsTree[bfsTree[1][1]])
```