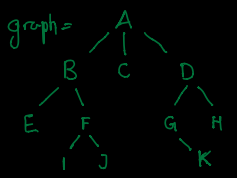
Input: A Node class that has a name and an array of optional children nodes

graph=



// When put together, nodes form an acyclic tree-like structure

Output: array // Traverse the tree using BFS approach (specifically navigating the tree from left to right), stores all the nodes' names in the input array

["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K"]

```
class Node {
  constructor(name) {
    this.name = name;
    this.children = [];
  }

  addChild(name) {
    this.children.push(new Node(name));
    return this;
  }

  // O(v + e) time | O(v) space
  breadthFirstSearch(array) {
    let queue = [this];
    while (queue.length > 0) {
      const current = queue.shift();
      array.push(current.name);
      for (const children of current.children) {
        queue.push(children);
      }
    }
    return array;
  }
}
```

Idea: In BFS, we use a queue to keep track of what node we explore next. We first add the root node to the queue (enqueue) and the shift() it off (dequeu) with this node, we add it's children (Left to right) to the queue. Once added, we grab the next value in the queue by dequeing it and then repeat.

Time: O(v+e) (where v is the vertices and e are the edges) since at each vertex (node) we traverse, we explore it's edges

Space: O(v) since we store every node in an array
since we (at worst) we could have a one branch graph and before our queue would have all the vertices at once (v-1 or O(v))