LeetCamp

A project dedicated to DS&A

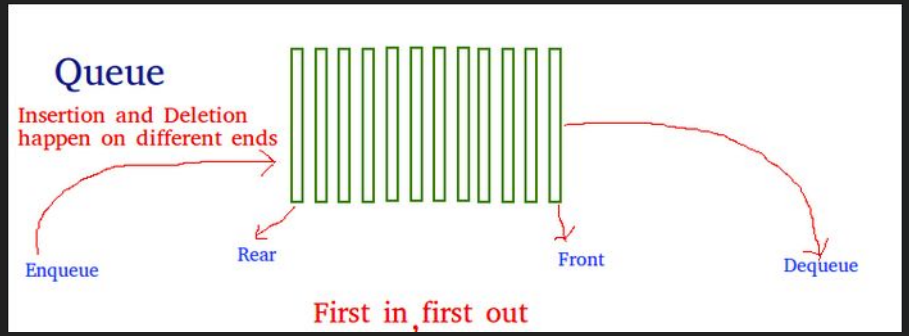
Agenda

1. Queues
2. BFS on Trees
3. Practice

Queues

Queues

- Linear data structure
- First In First Out (FIFO)
- Operations:
  - Enqueue - add item to the queue
  - Dequeue - remove item from the queue
  - Front - get item at the front
  - Rear - get item at the rear
- Recommended Python implementation:
  - Collections.deque
  - i.e. ***from collections import deque***



Queue Implementation

```
from collections import deque

# Initializing a queue
q = deque()

# Adding elements to a queue
q.append('a')
q.append('b')
q.append('c')

print("Initial queue")
print(q)

# Removing elements from a queue
print("\nElements dequeued from the queue")
print(q.popleft())
print(q.popleft())
print(q.popleft())

print("\nQueue after removing elements")
print(q)
```

Console Output:

- deque([a, b, c])

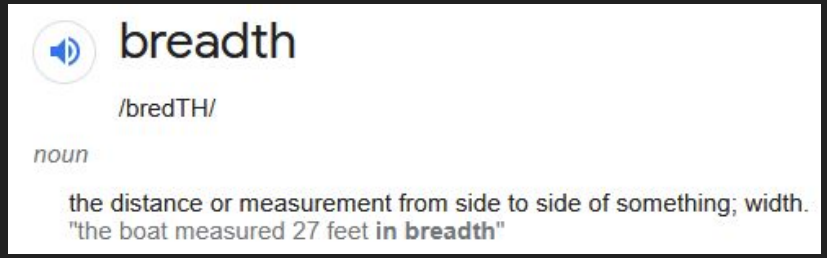
- a
- b
- c

- deque([])

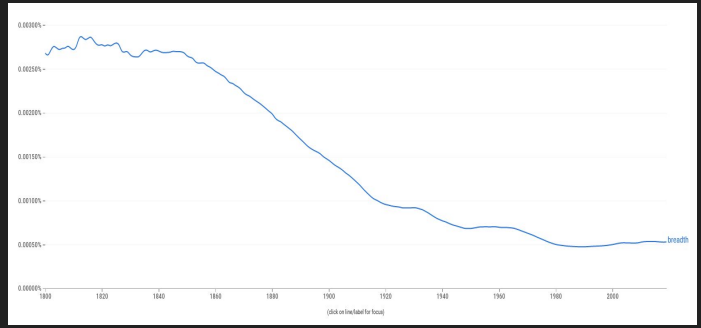
BFS on Trees

What is Breadth-First Search (BFS)?

- Breadth-first search (BFS) is used when you want to search *far and wide* before moving farther down a path
- Extra memory (up to O(w), where w is the diameter of the tree) *may be needed*
- Rather than “going down the rabbit whole”, we check everything at our current level first...



What is “breadth”?

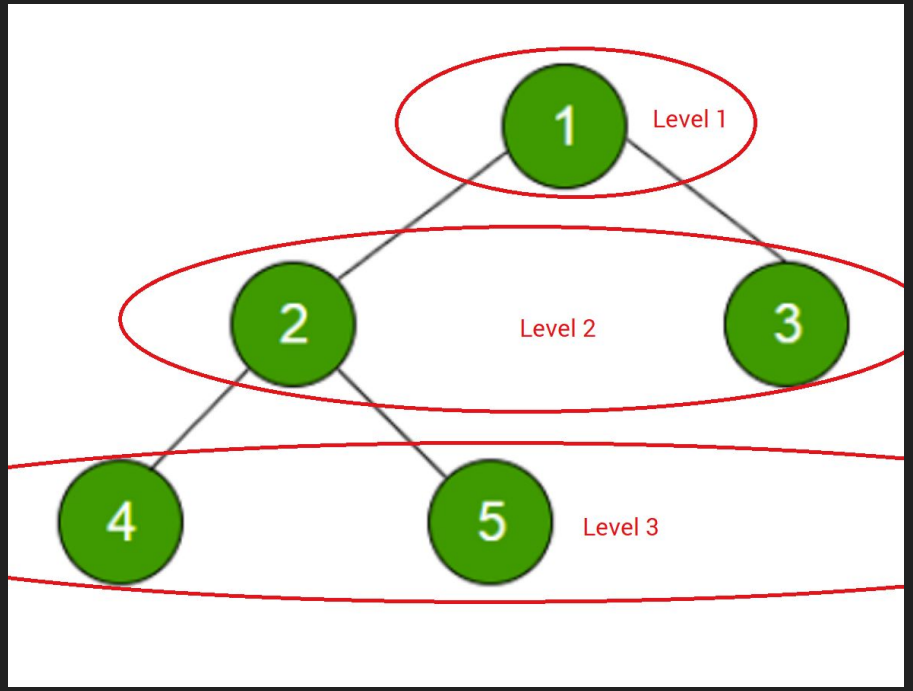


Use of the word “breadth”

BFS on Trees

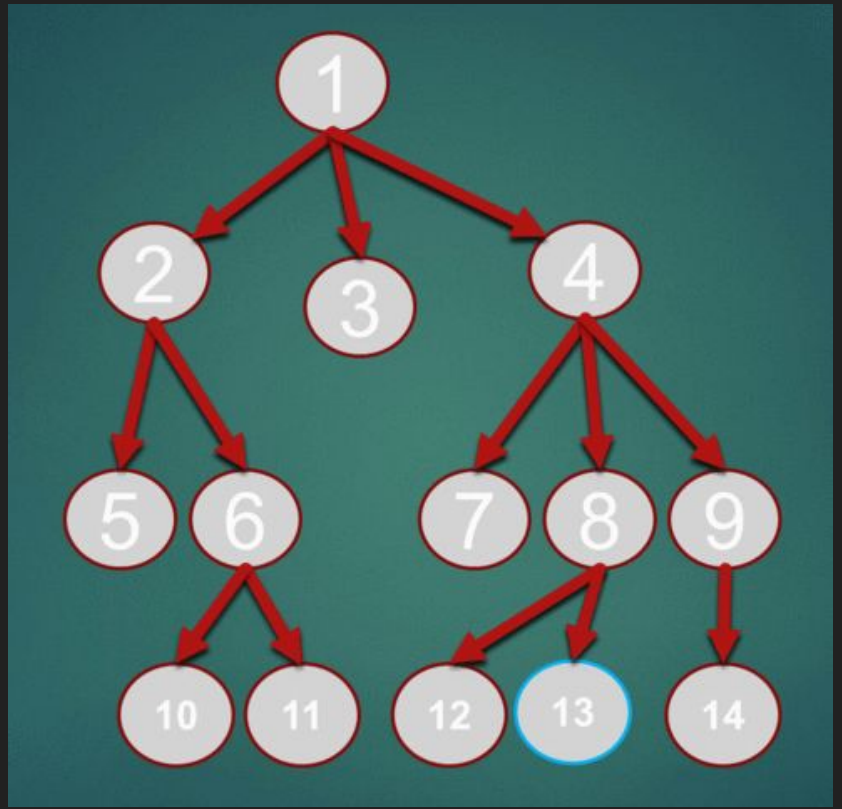
- Trees have “levels”

- To search using BFS, we want to look at **ALL NODES** on a given level before moving further down



Why did we talk about Queues today?!

1. Start with your root node
2. Enqueue all of its children
3. *While* the queue is not empty
    a. popleft() from queue
    b. Is this the node you are searching for?
        i. If so, return the node
        ii. Else, enqueue it's children
4. If the queue is empty and you have not found the node, it is must not be in the tree.



Boilerplate BFS

- Make sure you import deque
  - **from collections import deque**
- You have to start somewhere, make sure to enqueue the root
- If you are going to look at node attributes (node.left, node.right, node.val)
  - *ensure that no NULL values are ever in the queue*

```
##############################################
####         Boilerplate Functions         ####
##############################################
# from collections import deque
def bfs(root: TreeNode, target):
    """Breadth First Search (BFS)"""
    # Initialize a queue
    q = deque()
    # If the root exists, enqueue it
    if root:
        q.append(root)

    while len(q) > 0:
        node = q.popleft()
        # If the current node is the target
        if node.val == target:
            return node
        # If the node has a left child, enqueue it
        if node.left:
            q.append(node.left)
        # If the node has a right child, enqueue it
        if node.right:
            q.append(node.right)
    # If there are no more nodes in the queue
    # and we haven't found the target, it is not in the tree
    return None
```

Practice

700. Search in a Binary Search Tree

- Pretend this is a regular Binary Tree, use BFS
- Find a target node (called “val”) in the tree

- Hint: use the 1 through 4 steps to implement your solution

700. Solution

- Using BFS, this is the correct solution for a Binary Tree

- This can be improved because it is a binary search tree... but that is not today's lesson!

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def searchBST(self, root: Optional[TreeNode], val: int) -> Optional[TreeNode]:
        """Breadth First Search (BFS)"""
        # Initialize a queue
        q = deque()
        # If the root exists, enqueue it
        if root:
            q.append(root)

        while len(q) > 0:
            node = q.popleft()
            # If the current node is the target
            if node.val == val:
                return node
            # If the node has a left child, enqueue it
            if node.left:
                q.append(node.left)
            # If the node has a right child, enqueue it
            if node.right:
                q.append(node.right)
        # If there are no more nodes in the queue
        # and we haven't found the target, it is not in the tree
        return None
```

102. Binary Tree Level Order Traversal

- Handle one level at a time, from left to right, using BFS

- Hint 1: when the while loop starts, the current length of the queue ***is the length of the current level...***

- Hint 2: Get the length of the queue *at the start of the while loop*, use a nested for loop to loop that many times before adding elements to your output

102. Solution

- n = <number of elements in the current level>

- Append “level” list after all elements in the level are added

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        q = deque()
        if root:
            q.append(root)

        res = []
        while len(q) > 0:
            n = len(q)
            level = []
            for _ in range(n):
                node = q.popleft()
                level.append(node.val)
                if node.left: q.append(node.left)
                if node.right: q.append(node.right)
            res.append(level)
        return res
```

993. Cousins in Binary Tree

- You can enqueue more than just a single element in a queue

- What if you enqueued a tuple, containing multiple pieces of information...

- i.e. `q.append((node, parent_node, depth))`

993. Solution

- We should enqueue more than just the nodes...
- Enqueue (node, parent, depth)

- Check that the parents of x and y are not the same, but that they are on the same level

```python
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def isCousins(self, root: Optional[TreeNode], x: int, y: int) -> bool:
        q = deque()
        if root:
            q.append((root, None, 0))

        res = []
        while len(q) > 0:
            if len(res) == 2: # x and y have already been found
                break
            # get node, parent, and depth out of the current node
            node, parent, depth = q.popleft()
            # check if it is either x or y
            if node.val == x or node.val == y:
                # if it is, add it's parent and depth to res
                res.append((parent, depth))

            # If left child exists, enqueue it (with it's parent and depth)
            if node.left:
                # remember, node is parent and depth is current + 1 depth
                q.append((node.left, node, depth + 1))
            # If right child exists, enqueue it (with it's parent and depth)
            if node.right:
                # remember, node is parent and depth is current + 1 depth
                q.append((node.right, node, depth + 1))

        # make sure both x and y have been found (res will be length 2)
        if len(res) == 2:
            node_x, node_y = res
            # if the nodes parents ARE NOT the same node,
            # but they exist on the same level, then they are cousins!
            return node_x[0] != node_y[0] and node_x[1] == node_y[1]

        # if both x and y were not found
        return False
```

Q&A Time!