csgator Follow
Jan 4, 2018 · 4 min read

Leetcode Pattern 1 | BFS + DFS == 25% of the problems — part 1

It is amazing how many graph, tree and string problems simply boil down to a DFS (Depth-first search) / BFS (Breadth-first search). Today we are going to explore this basic pattern in a novel way and apply the intuition gained to solve some medium problems on Leetcode.

Let us build on top of pattern 0. First of, a tree can be thought of as a connected acyclic graph with N nodes and N-1 edges. Any two vertices are connected by *exactly one* path. So naturally the question arises, what about a DFS or a BFS on binary trees ? well there are 6 possible DFS traversals for binary trees (3 rose to fame while the other 3 are just symmetric)

1. left, right, root (Postorder) ~ 4. right, left, root
2. left, root, right (Inorder) ~ 5. right, root, left
3. root, left, right (Preorder) ~ 6. root, right, left

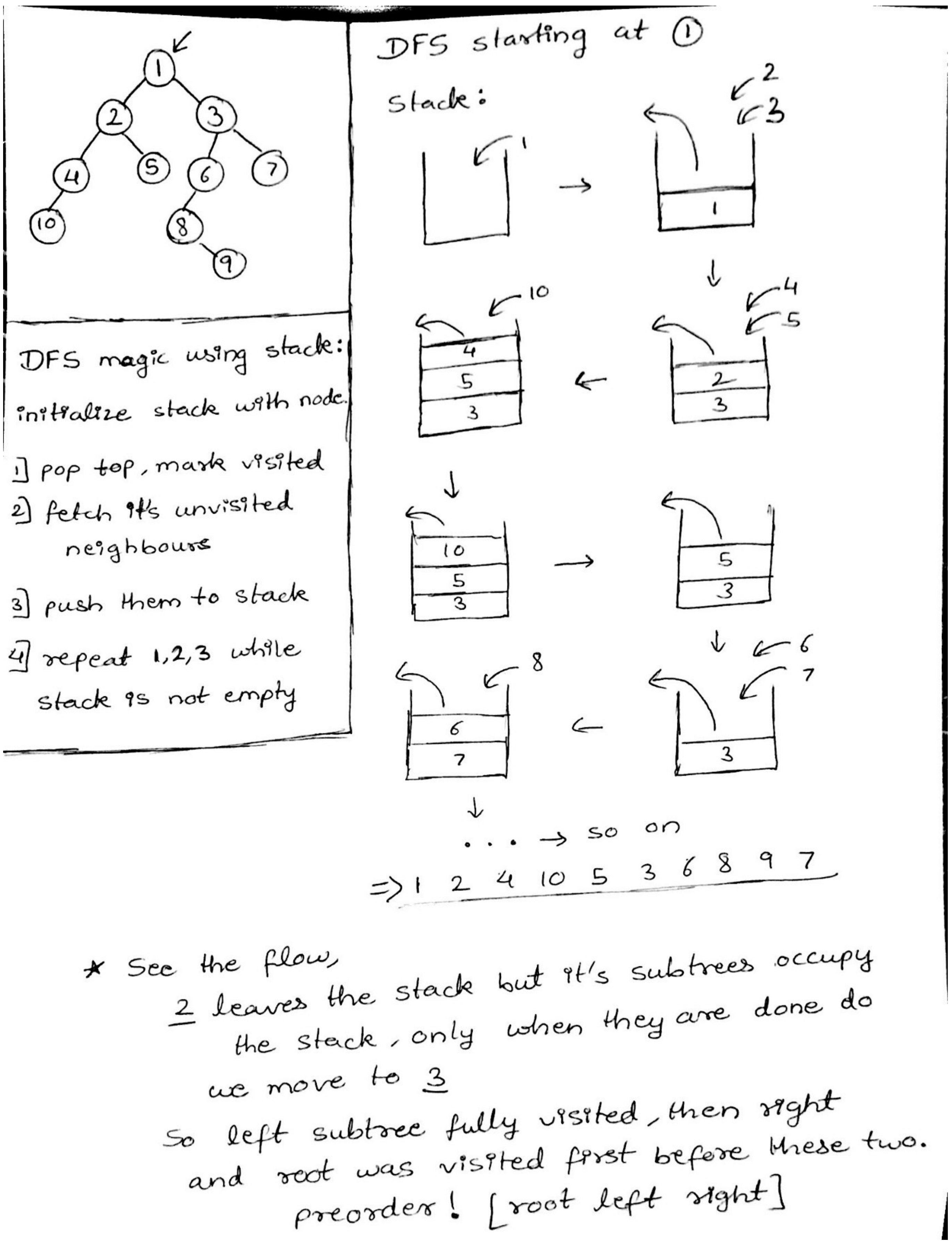
And there is one BFS which is the level order traversal (can be done using queue). Let us analyze the DFS pattern using a stack which we are already familiar with.

DFS is all about diving as deep as possible before coming back to take a dive again. Below is the iterative DFS pattern using a stack that will allow us to solve a ton of problems. (iterative introduced first so as to develop intuition)

> *DFS magic spell: 1]push to stack, 2] pop top , 3] retrieve unvisited neighbours of top, push them to stack 4] repeat 1,2,3 while stack not empty. Now form a rap !*

144. Preorder Traversal

Let’s walk through the above spell using an example tree.



DFS magic using stack:
initialize stack with node.
1] pop top, mark visited
2] fetch it's unvisited neighbours
3] push them to stack
4] repeat 1,2,3 while stack is not empty

★ See the flow,
2 leaves the stack but it's subtrees occupy the stack, only when they are done do we move to 3
So left subtree fully visited, then right and root was visited first before these two.
preorder! [root left right]

```cpp
class Solution {
public:
    // DFS magic : initialize stack and do the following
    // pop top, retrieve neighbours for top, push unvisited neighbours to stack | repeat
    // because this is a tree no need to keep track of visited as no cycles possible.

    vector<int> preorderTraversal(TreeNode* root) {
        stack<TreeNode*> s;
        vector<int> result;

        s.push(root);

        while(!s.empty()){
            TreeNode* current = s.top(); s.pop();  // pop top

            if(current != NULL){
                // push unvisited neighbours to stack | order matters here, if you rever
                // it would still be a DFS but a symmetric one to preorder out of the 6
                s.push(current->right);
                s.push(current->left);

                result.push_back(current->val);
            }
        }

        return result;
    }
};
```

preorder.cpp hosted with ❤ by GitHub view raw

Two things to ponder on as we move further:

1] Why are we using stack for DFS , couldn't we use a queue ? (always remember : stack for DFS, imagine a vertical flow | queue for BFS, horizontal flow, more on this later)

2] How do we extend this DFS process to general graphs or graphs disguised as matrices (as in most LC problems). (Include a mechanism to track visited)

323. Number of Connected Components in an Undirected Graph

The base problem upon which we will build other solutions is this one which directly states to find the number of connected components in the given graph. We could use DFS / BFS to solve this. Below is the DFS code using the stack spell.

```
class Solution {
public:
    int countComponents(int n, vector<pair<int, int>>& edges) {
        vector<bool> visited(n, false);
        vector<vector<int>> adjList(n, vector<int>(0));
        stack<int> dfs;

        int count = 0;
        int ans = 0;

        // building the graph's adjacency list
        for(auto edge : edges){
            int from = edge.first;
            int to   = edge.second;

            adjList[from].push_back(to);
            adjList[to].push_back(from);
        }

        // dfs on nodes to find connected components
        for(int i = 0; i < n; i++){
            if(!visited[i]){
                ans++;
                dfs.push(i);

                while(!dfs.empty()){
                    int current = dfs.top(); dfs.pop();
                    visited[current] = true;

                    for(int neighbour : adjList[current]){
                        if(!visited[neighbour]) dfs.push(neighbour);
                    }
                }
            }
        }

        return ans;
    }
};
```

countComponents.cpp hosted with ❤ by GitHub view raw

200. Number of Islands

This falls under a general category of problems where in we just need to find the number of connected components but the details could be twisted.

The intuition here is that once we find a "1" we could initiate a new group, if we do a DFS from that cell in all 4 directions we can reach all 1's connected to that cell and thus belonging to same group. We could mark these as visited and move on to count other groups. One optimization here is that we don't need the matrix later so we could as well destroy the visited cells by placing a special character saving us extra memory for the visited array.

Even if we needed the matrix later one could always restore the original value after the dfs call. Below is a simple recursive DFS solution.

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int ans = 0; // number of groups

        // iterating through given grid to find a '1'
        for(int i = 0; i < grid.size(); i++){
            for(int j = 0; j < grid[0].size(); j++){
                if(grid[i][j] == '1'){
                    ans++; // start a group and visit all members of this group using df
                    dfs(grid, i, j);
                }
            }
        }

        return ans;
    }

    void dfs(vector<vector<char>>& grid, int y, int x){
        // if out of bounds or at a cell with '0' or '*', simply stop and return | end t
        if(x < 0 || x >= grid[0].size() || y < 0 || y >= grid.size() || grid[y][x] != '1

        grid[y][x] = '*'; // destroying visited cells so they are not re-visited

        // recursive dfs on all neighbours
        dfs(grid, y + 1, x);
        dfs(grid, y - 1, x);
        dfs(grid, y, x + 1);
        dfs(grid, y, x - 1);
    }
};
```

numIslands.cpp hosted with ❤ by GitHub view raw

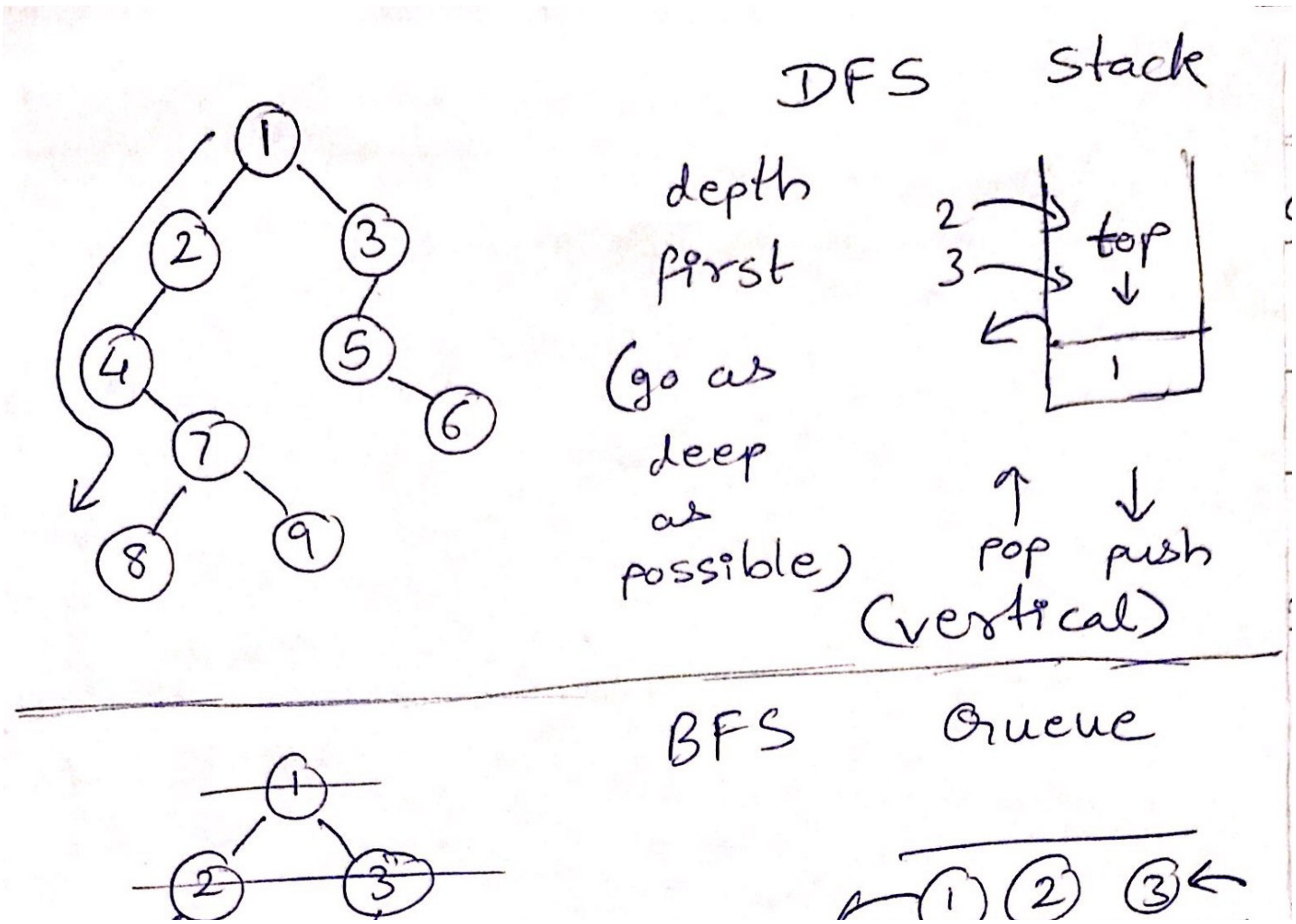
547. Friend Circles

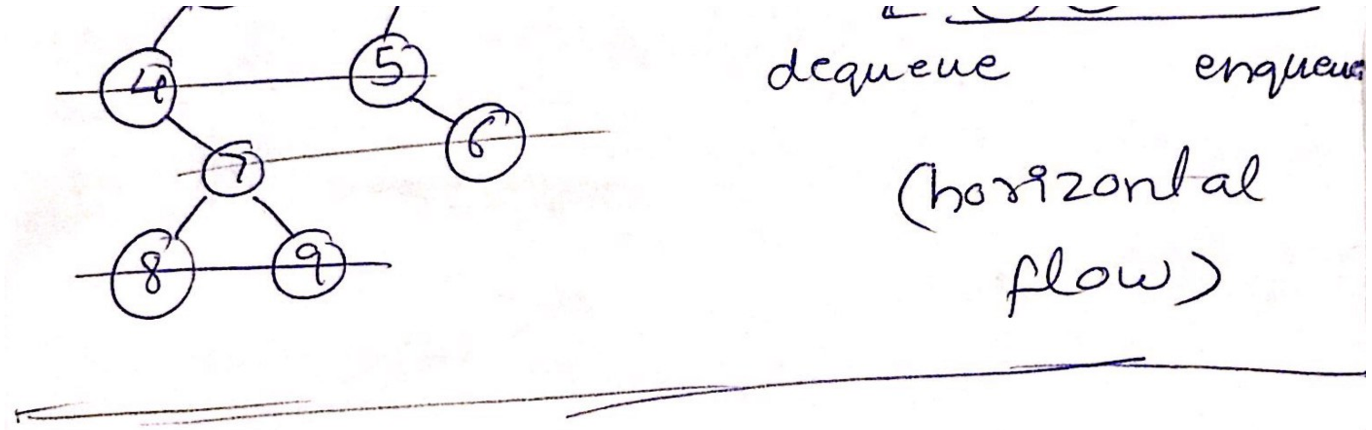
Same concept, find connected components. The only twist is that the connected neighbors are presented in a different form. Here we don't destroy the matrix, but use an array to keep track of visited (friendZoned). Notice the stack pattern, it is exactly the same as in connected components problem. All that changes is the way neighbors are defined. If we were to write an iterative version for the islands problems, it would also be very similar.

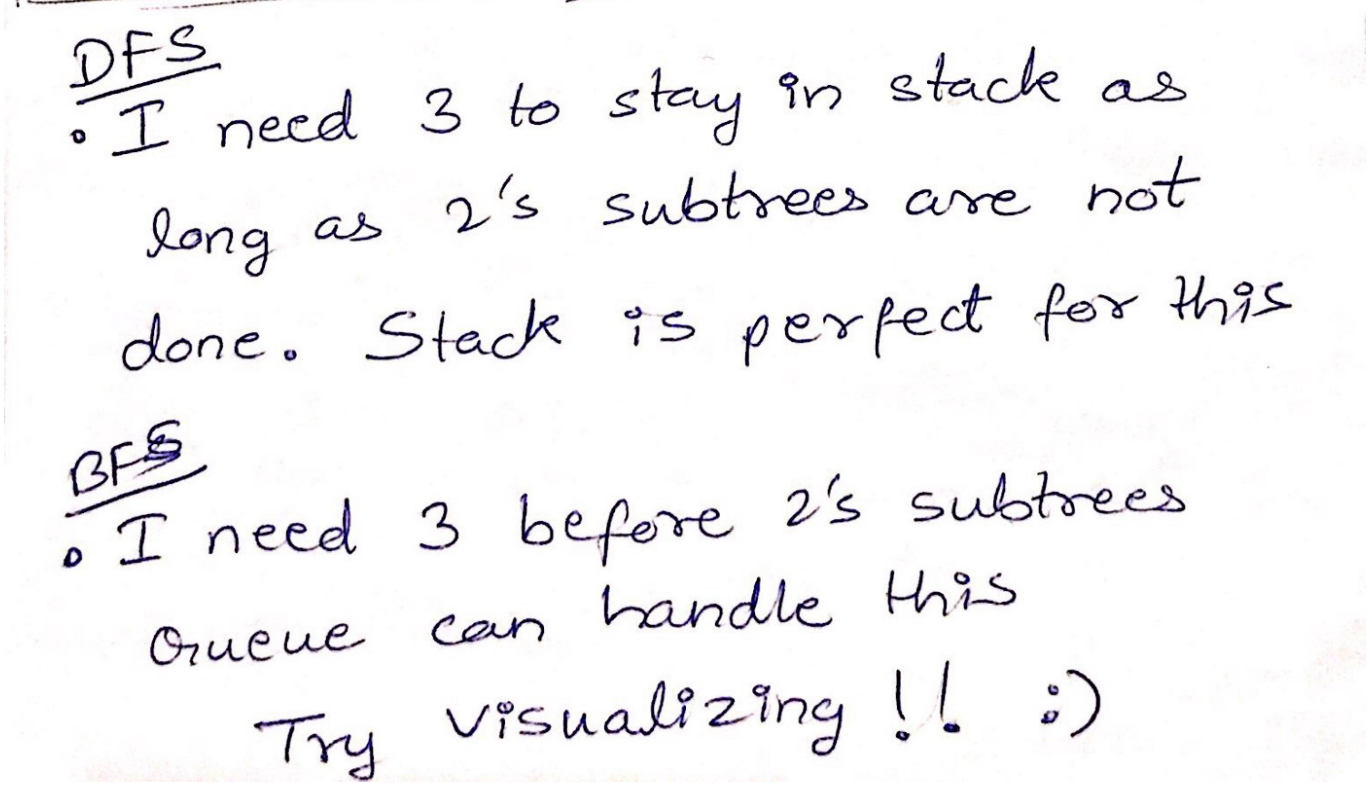
```

class Solution {
public:
    // M is the friend matrix provided | basic intuition: we need connected components i
    // why does this work ? because applying transitive property to this : a friend of b
    // a,b,c,d should fall into 1 friend circle, just imagine a,b,c,d to be nodes of gra
    // are friends else not connected by an edge, so simply finding the total groups / c
    int findCircleNum(vector<vector<int>>& M) {
        vector<bool> friendZoned(M.size(), false);                          // a
        stack<int> dfs;                                                     // s

        int circles = 0;                                                    // f

        for(int i = 0; i < M.size(); i++){
            if(!friendZoned[i]){
                circles++;  // // person i unvisited so start new friend circle

                // DFS magic : { push to stack - pop top - retrieve neighbours - repeat
                dfs.push(i);

                while(!dfs.empty()){
                    int current = dfs.top(); dfs.pop(); // pop top
                    friendZoned[current] = true;        // mark node visited

                    // retrieve it's unvisited neighbours and push them to stack
                    for(int j = 0; j < M[current].size(); j++){
                        if(!friendZoned[j] && M[current][j] == 1){
                            dfs.push(j);
                        }
                    }
                }
            }
        }

        return circles;
    }
};
```

friendCircleNum.cpp hosted with ❤ by GitHub view raw

I broke this post into 2 parts as it got too long, I will visit BFS in the next part. So let's conclude this part by pondering on why we always use stack for dfs and queue for bfs. Have you ever wondered why we don't use queue for dfs or stack for bfs? questions like these really give us some insights on the difference between stacks and queues.







DFS
- I need 3 to stay in stack as long as 2's subtrees are not done. Stack is perfect for this

BFS
- I need 3 before 2's subtrees Queue can handle this
Try visualizing !! :)

So using a stack I could pop 2 and push it's kids and keep doing so eventually exhausting 2's subtrees, 3 stays calmly in the stack just below the part where the real push-pop action is going, we pop 3 when all subtrees of 2 are done. This feature of stack is essential for DFS.

While in a queue, I could dequeue 2 and enqueue it's subtrees which go behind 3 as it was already sitting in the queue. So the next time I dequeue I get 3 and only after that do I move on to visiting 2's subtrees, this is essentially a BFS !

For me this revelation was pure bliss. Take a moment to celebrate the history of Computer Science and the geniuses behind these simple yet powerful ideas.

http://people.idsia.ch/~juergen/bauer.html

Do comment your views and feel free to connect with me on LI : https://www.linkedin.com/in/sourabh-reddy

Programming Leetcode Whiteboarding Algorithms Datastructures