csgator Follow
Jan 6, 2018 · 4 min read

Leetcode Pattern 1 | DFS + BFS == 25% of the problems — part 2

Hola again ! Thanks for all the positive feedback. It really motivates me to keep writing daily. Many people actually asked me to write on DP patterns next as that is the most dreaded topic in interview prep. I would dedicate the next few posts to the same, building intuition on some non-trivial DP problems but for today let's complete BFS. Some useful tips for DP to help you out till then:

1. Solve 3 DP problems each day for 2 weeks and you'll start getting a hang of the underlying patterns. Struggle a bit, think hard, lookup the solution after an hour or two, **understand the intuition deeply**, repeat.
2. Think hard on the classic DP problems (which are only a handful), discuss / explain the intuition to a peer, draw it out on paper (very important) and you would then be able to solve most DP problems.
3. Read solutions to DP problems from geeksforgeeks or watch Tushar Roy's playlist on DP, the whole point is to accustom yourself with solutions to standard problems and then apply that intuition elsewhere.

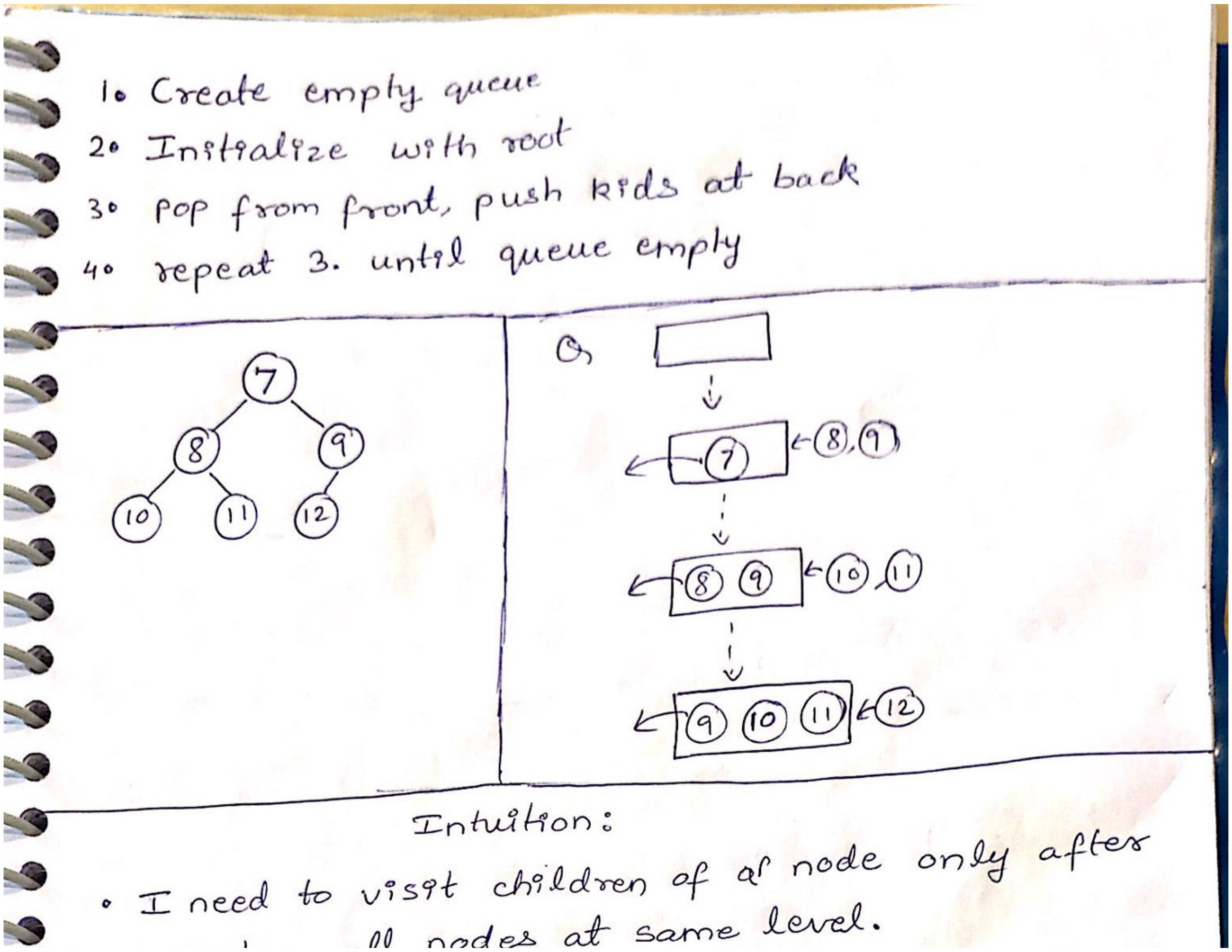
Honestly DP is overrated, it is totally doable. Just break out of the 'DP is wicked hard' phase and start solving problems. Over the next few days I am going to follow the above outlined techniques and share some insights every day.

BFS:

Again let's start with a tree, cause we are so obsessed with trees! level order traversal is simply a BFS and a rather simple one at that !

```cpp
void levelOrder(TreeNode* root) {
    queue<TreeNode*> bfs;
    bfs.push(root);

    while(!bfs.empty()){
        TreeNode* current = bfs.front(); bfs.pop();

        if(current){
            cout << current->val << " ";

            bfs.push(current->left);
            bfs.push(current->right);
        }
    }

    cout << endl;
}
```

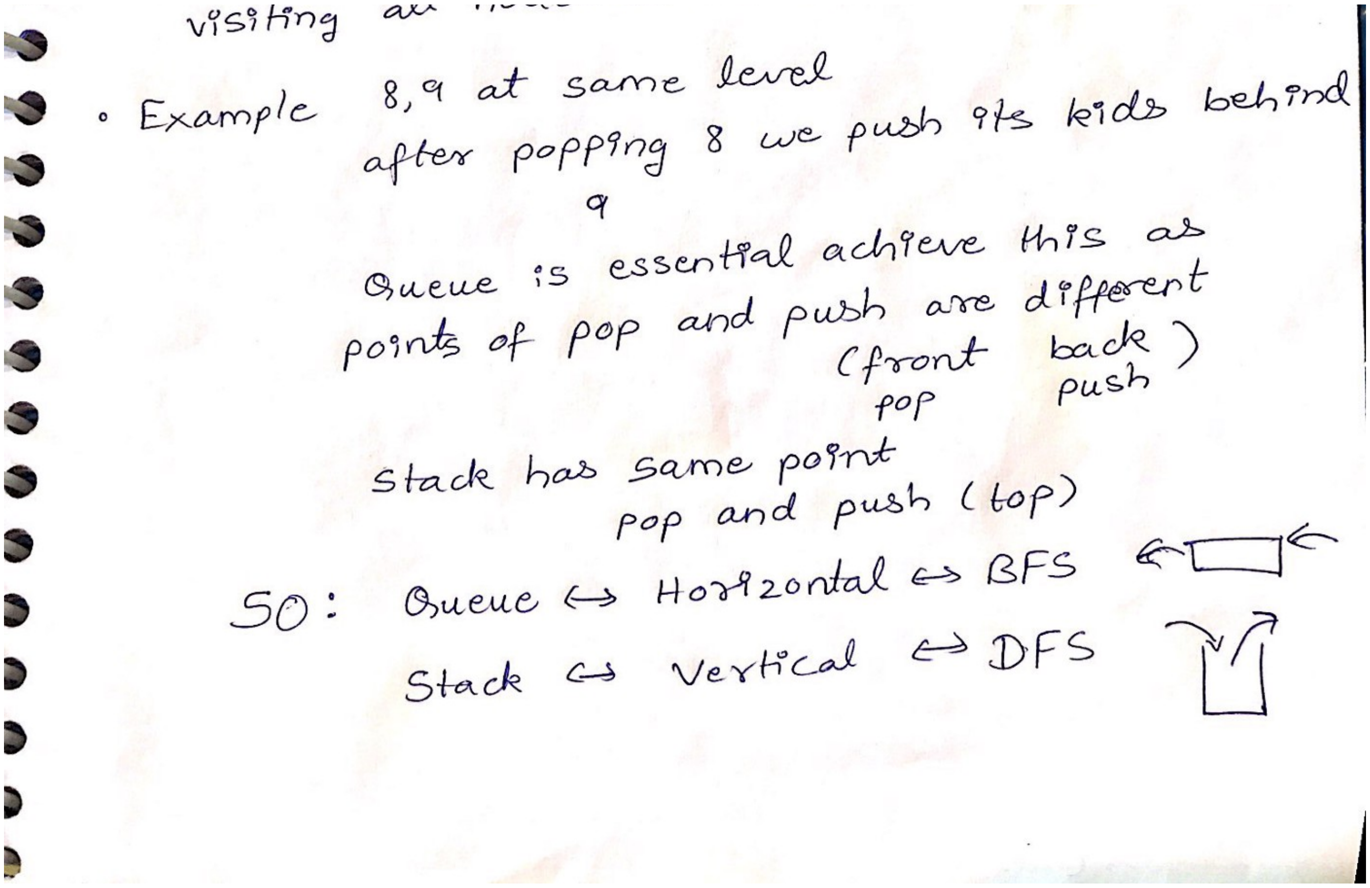
treebfs.cpp hosted with ❤ by GitHub view raw



1. Create empty queue
2. Initialize with root
3. Pop from front, push kids at back
4. repeat 3. until queue empty

Intuition:

- I need to visit children of a node only after
ll nodes at same level.



visiting

• Example 8,9 at same level
after popping 8 we push its kids behind
9

Queue is essential achieve this as
points of pop and push are different
(front back)
pop push

Stack has same point
pop and push (top)

SO: Queue ↔ Horizontal ↔ BFS
Stack ↔ Vertical ↔ DFS

102. Binary Tree Level Order Traversal

The leetcode problem on level order traversal is a bit more involved than the above mentioned simple traversal. They require you to store each level result in an array and return final result as array of arrays. Just imagine somebody told you to put a line break after printing each level, that would essentially be the same problem.

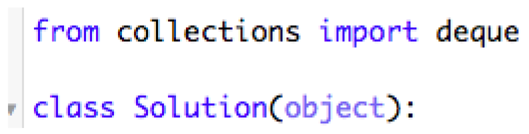
1st approach

Using the above simple code it is not possible to know when a new level starts. I originally solved this problem using 2 queues, but I found this amazing approach in discuss and I have adopted it since then. (discuss is where the true learning happens ;)). We use a dummy node as marker to mark level ends. Try visualizing the horizontal queue push- pop action going on in BFS and figure out how you could use an extra dummy node to mark level ends, then go through below code. This approach simply blew my mind !

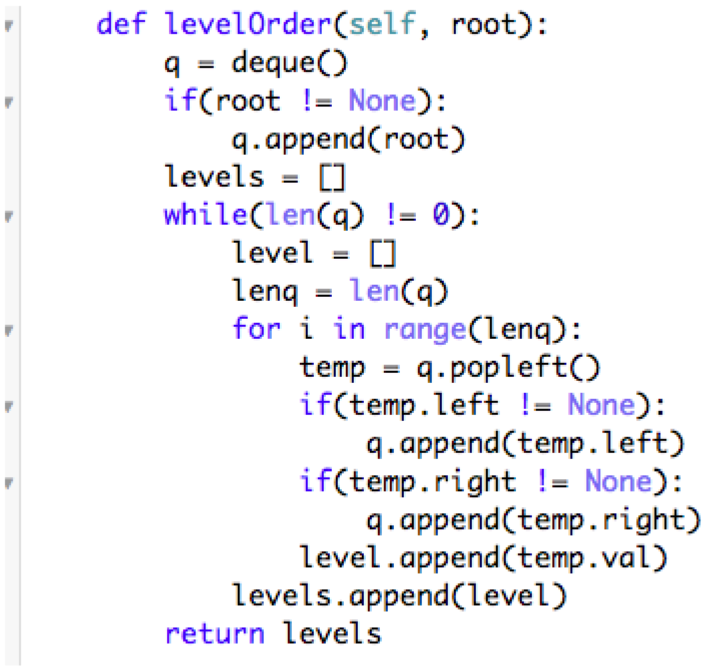
```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result; // array of arrays to store final result
        vector<int> levelresult;    // array storing result for each level temporarily

        queue<TreeNode*> bfs;       // bfs queue
        TreeNode* marker = new TreeNode(INT_MIN);  // dummy node to mark end of level

        bfs.push(root);
        bfs.push(marker);

        while(!bfs.empty()){
            TreeNode* current = bfs.front(); bfs.pop(); // pop from front or standard de

            if(current){
                // if dummy node popped from queue means a level has ended
                if(current->val == INT_MIN){
                    result.push_back(levelresult); // store level result
                    levelresult = vector<int>(0);  // initialize levelresult for next le

                    // if queue not empty after removing marker, meaning a next level ex
                    if(!bfs.empty()) bfs.push(marker);
                }
                else{
                    levelresult.push_back(current->val); // storing current level values

                    // kids pushed at back of queue ( standard enqueue operation )
                    bfs.push(current->left);
                    bfs.push(current->right);
                }
            }
        }

        result.pop_back();
        return result;
    }
};
```

levelOrder.cpp hosted with ❤ by GitHub view raw

EDIT: As Shad Khan suggested on LI, we could eliminate the dummy node using the size of queue to keep track of level. Here is his Python code:



```
from collections import deque

class Solution(object):
```



```python
    def levelOrder(self, root):
        q = deque()
        if(root != None):
            q.append(root)
        levels = []
        while(len(q) != 0):
            level = []
            lenq = len(q)
            for i in range(lenq):
                temp = q.popleft()
                if(temp.left != None):
                    q.append(temp.left)
                if(temp.right != None):
                    q.append(temp.right)
                level.append(temp.val)
            levels.append(level)
        return levels
```

2nd approach

Let's play a **game of 2 queues**. We keep 2 queues for even and odd levels of the tree. So starting with 0th level i.e root, **initialize even queue with root**. Now imagine holding the even queue in your right hand and the odd queue in your left (just 2 boxes which allow entry from only one side and exit from the opposite side).

1. Tilt your right hand so all the contents of even queue start falling out. **Whenever a node falls out of a queue push it's children into the other queue.**

2. Now tilt your left hand emptying contents of odd queue and adding kids of falling out nodes into even queue.

3. Repeat until both queues are empty.

```cpp
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        queue<TreeNode*> even;
        queue<TreeNode*> odd;
        vector<vector<int>> result;

        even.push(root);

        while(!even.empty() || !odd.empty()){
            vector<int> level;

            if(odd.empty()){
                // while even not empty, transfer kids of falling out nodes to odd
                while(!even.empty()){
                    TreeNode* curr = even.front(); even.pop();

                    if(curr){
                        level.push_back(curr->val);

                        odd.push(curr->left);
                        odd.push(curr->right);
                    }
                }
            }
            else if(even.empty()){
                // if even is empty, transfer kids of falling out nodes to even | keep j
                while(!odd.empty()){
                    TreeNode* curr = odd.front(); odd.pop();

                    if(curr){
                        level.push_back(curr->val);

                        even.push(curr->left);
                        even.push(curr->right);
                    }
                }
            }

            result.push_back(level);
        }

        result.pop_back();
        return result;
    }
};
```

levelOrder2.cpp hosted with ❤ by GitHub view raw

101. Symmetric Tree problem also can be solved using 2 queue method in a slightly different way, but enough with trees already! Let's see an actual graph (matrix) problem using BFS.

542. 01 Matrix

How does one know that we need BFS here and not DFS, which is a very true dilemma is many problems, well the first thought that crosses my mind seeing this problem is if somehow I could iterate through all 0's in matrix and start a recursive action at these cells updating distances of neighboring cells by 1, keep doing so and stop only if the cell under consideration is already closer to another 0.

For this to be successful we need all those actions to execute only 1 round at a time (visit 4 neighbors) and then **wait for all others** to execute their 1 rounds so recursion doesn't work (DFS failed only 1 option left BFS). See how this is so similar to the tree case when we needed the kids of the current node **only after** we have visited nodes at same level, it's a BFS , bingo! The reason we need this here is if we don't wait for other actions to execute a round how would we know if the current cell is already closer to another 0 in which case we need to stop BFS operation for that cell.

```
class Solution {
public:
    vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
        queue<vector<int>> bfs;
        int dist = 0;
        int m = matrix.size(), n = matrix[0].size();

        for(int i = 0; i < matrix.size(); i++){
            for(int j = 0; j < matrix[i].size(); j++){
                if(matrix[i][j] == 0) bfs.push(vector<int>({i, j}));    // keep track of
                else matrix[i][j] = INT_MAX;                            // else initiali
            }
        }

        vector<vector<int>> directions({{1,0}, {-1, 0}, {0, 1}, {0, -1}});

        while(!bfs.empty()){
            vector<int> current = bfs.front(); bfs.pop();

            for(auto d : directions){
                int i = current[0] + d[0];
                int j = current[1] + d[1];

                // if new cell is out of bounds or is already closer to another 0, stop
                if(i < 0 || i >= m || j < 0 || j >= n || matrix[i][j] <= matrix[current[

                bfs.push(vector<int>({i, j}));                       // put in queue for f
```

```cpp
                    matrix[i][j] = matrix[current[0]][current[1]] + 1; // update new smaller
                }
            }

            return matrix;
        }
};
```

01matrix.cpp hosted with ❤ by GitHub view raw

Concluding thoughts on BFS :

1. Problems in which you have to find shortest path are most likely calling for a BFS.
2. For graphs having unit edge distances, shortest paths from any point is just a BFS starting at that point, no need for Dijkstra's algorithm.
3. Maze solving problems are mostly shortest path problems and every maze is just a fancy graph so you get the flow.

Enjoy and stay tuned !

Algorithms Data Structures Leetcode Interview Programming