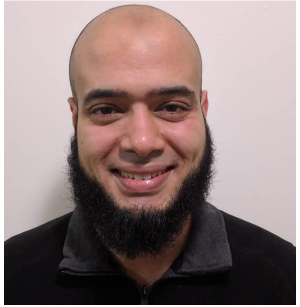
Data Structures

Level Order Traversal

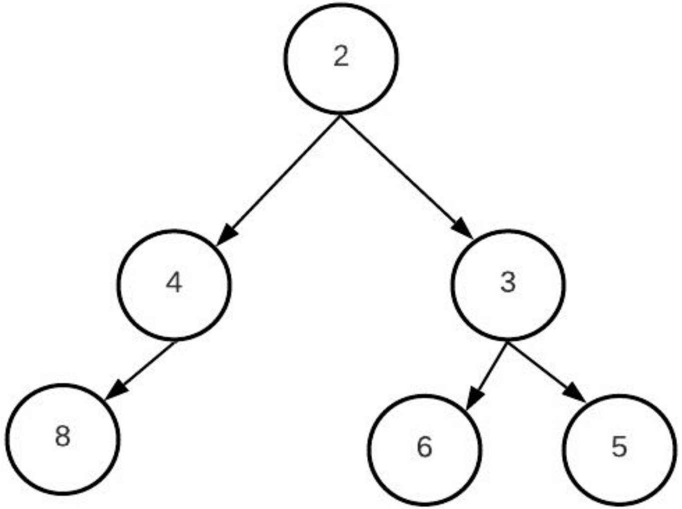
Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



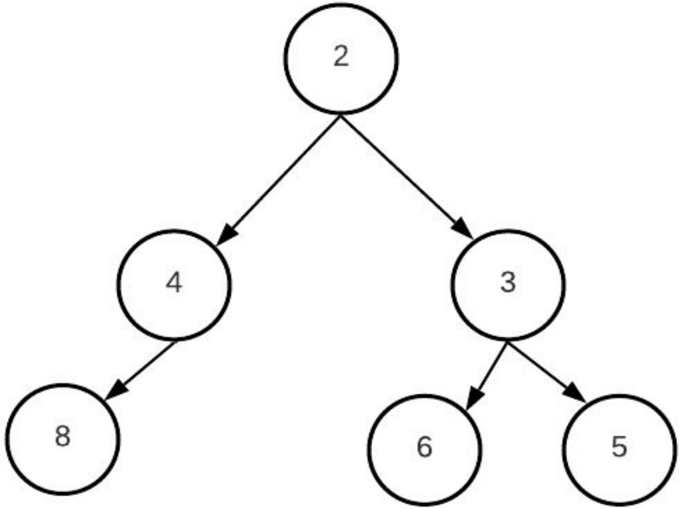
Level Order Traversal

- We learned 3 recursive traversal methods that keep going deep tell no more way
  - We call them depth first (go deeper)
  - Inorder here is: 8 4 2 6 3 5
- In level order traversal, we print the tree level by level
  - Level 0: 2
  - Level 1: 4 3
  - Level 2: 8 6 5
  - We call it: breadth first
- Depth vs Breadth



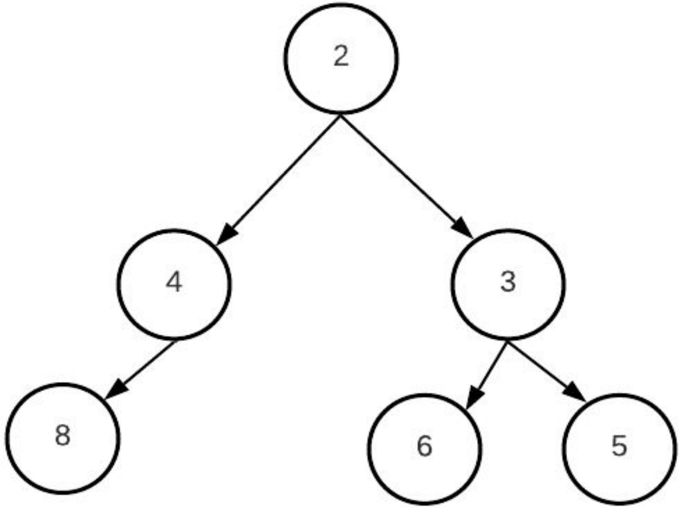
Traversing level by level

- Although we can use recursion to find the levels one by one, it will be very impractical
- One of the great applications of the **queue** is to iterate on a tree level by level
- Start a queue with the root.
- Pop it and push its children and so on
- Can you finish idea and implement it?



Traversing level by level

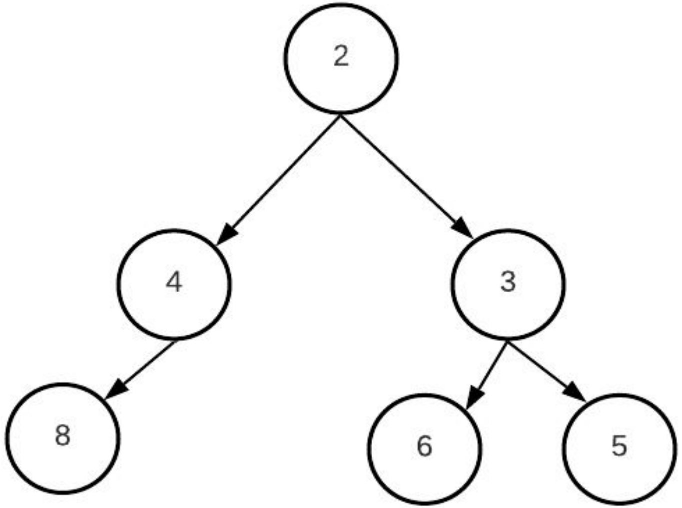
- Add the root node to the queue
- While not empty
  - Get node
  - Print it
  - Add its available children



2				

Traversing level by level

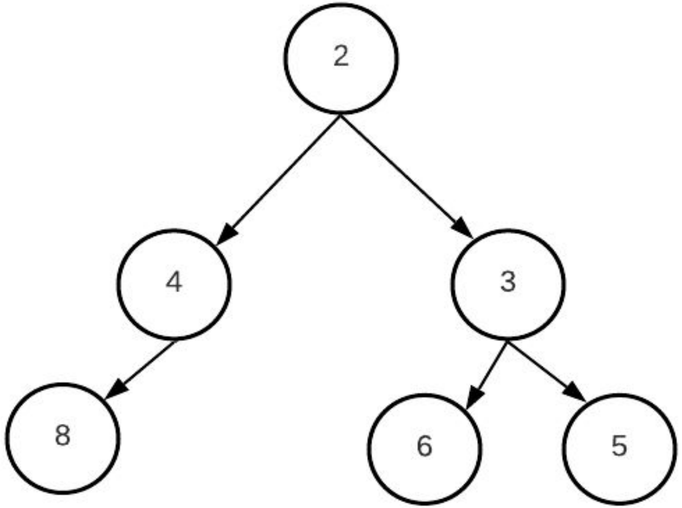
- Pop: Tree(2)
- Add children: 4, 3
- **Printed so far**: 2



4	3			

Traversing level by level

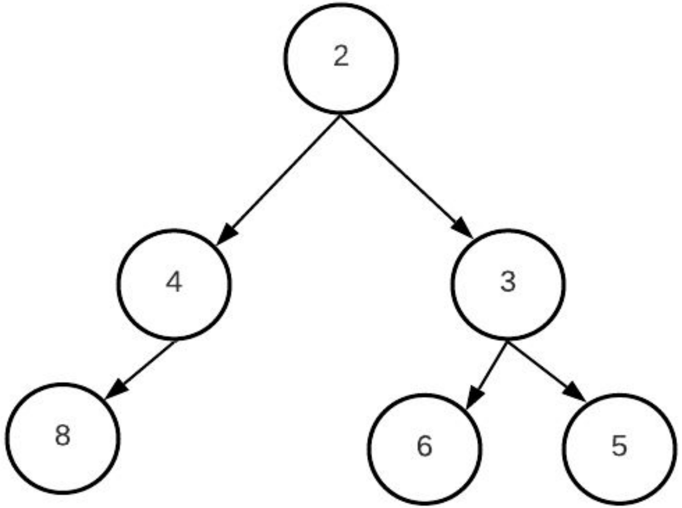
- Pop: Tree(4)
- Add children: 8
- **Printed so far**: 2 4



3	8			

Traversing level by level

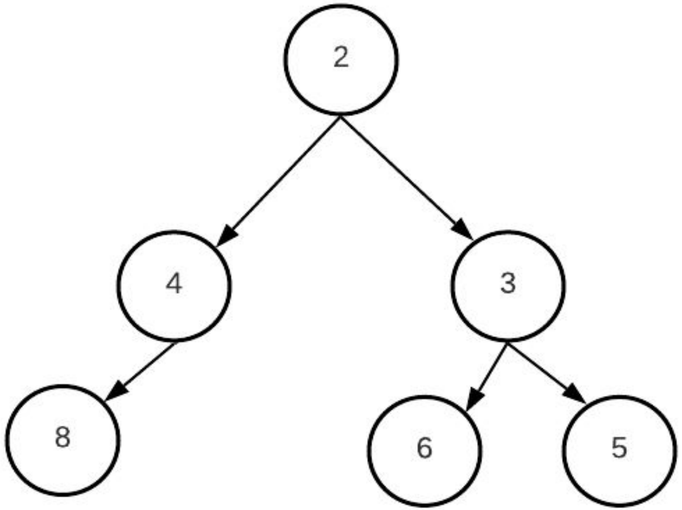
- Pop: Tree(3)
- Add children: 6, 5
- **Printed so far**: 2 4 3



8	6	5		

Traversing level by level

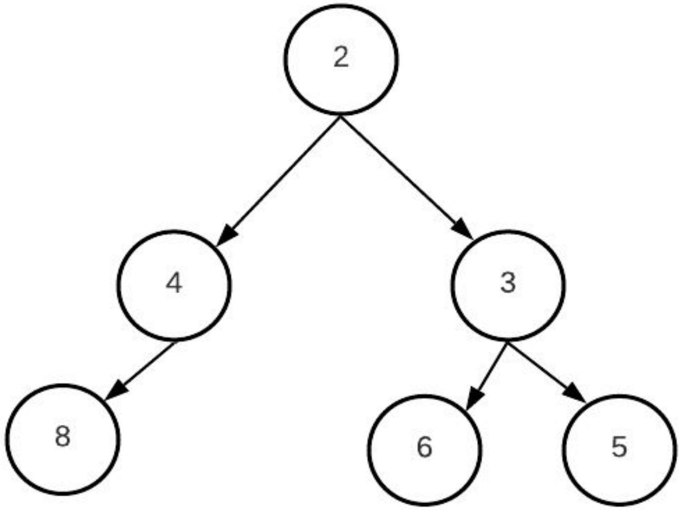
- Pop: Tree(8)
- Add children: None
- **Printed so far**: 2 4 3 8



6	5			

Traversing level by level

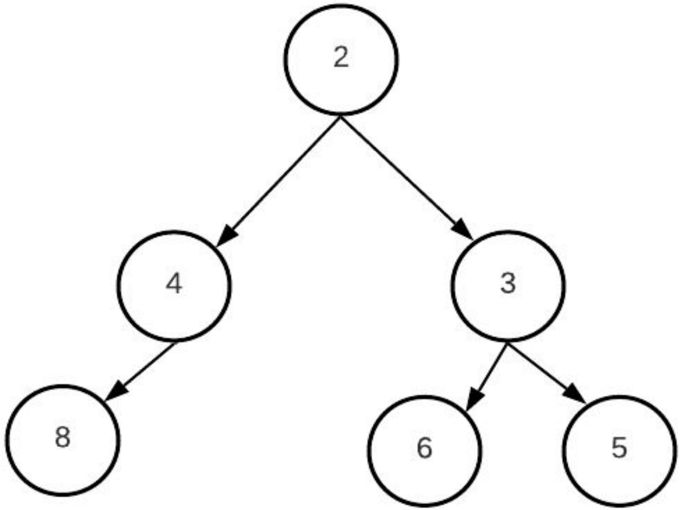
- Pop: Tree(6)
- Add children: None
- **Printed so far**: 2 4 3 8 6



5				

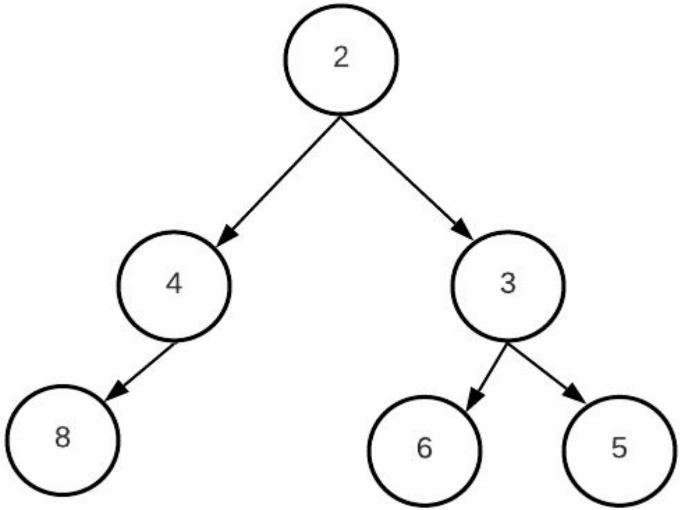
Traversing level by level

- Pop: Tree(5)
- Add children: None
- Empty: queue: stop
- **Printed so far**: 2 4 3 8 6 5



The queue content

- What is happening? We iterate on nodes one by one, adding its children to the current queue
- The queue will be in 1 of 2 cases
  - Either all current nodes of 1 specific level
  - Or 2 consecutive levels

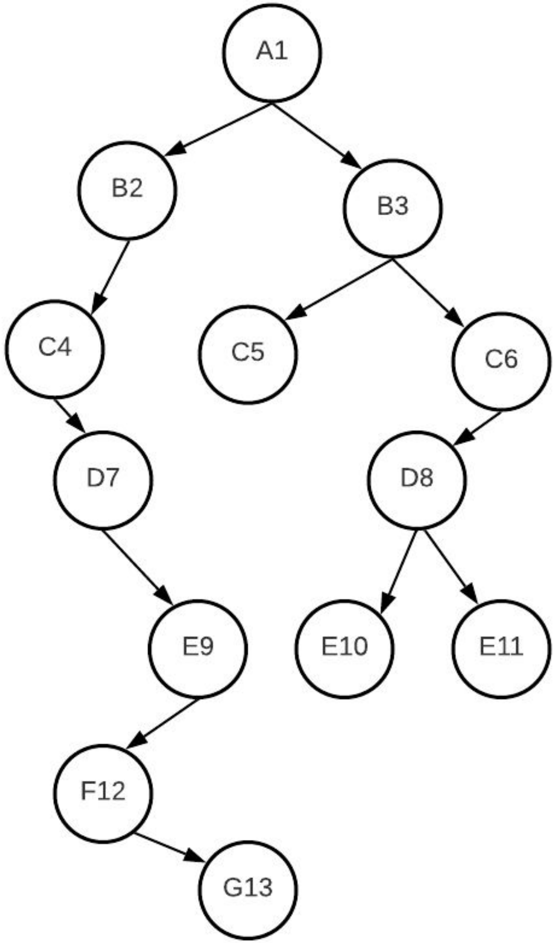


3	8			

8	6	5		

Let’s check the queue

- A1 : remove A1, add B2, B3
- B2, B3 : remove B2, add C4
- B3, C4 : remove B3, add C5, C6
- **C4, C5, C6** : remove C4, add D7
- **C5, C6, D7** : remove C5, add nothing
- C6, D7 : remove C6, add D8
- D7, D8 : remove D7, add E9
- D8, E9 : remove D8, add E10, E11
- E9, E10, E11 : remove E9, add F12
- E10, E11, F12
- F12
- G13



Implementation v1

- Just simulate the process using the code
- Although we print level by level, but we don't know the level of each node!
- 2 ways
    - In the queue, add also the level
    - Or smartly, process level by level

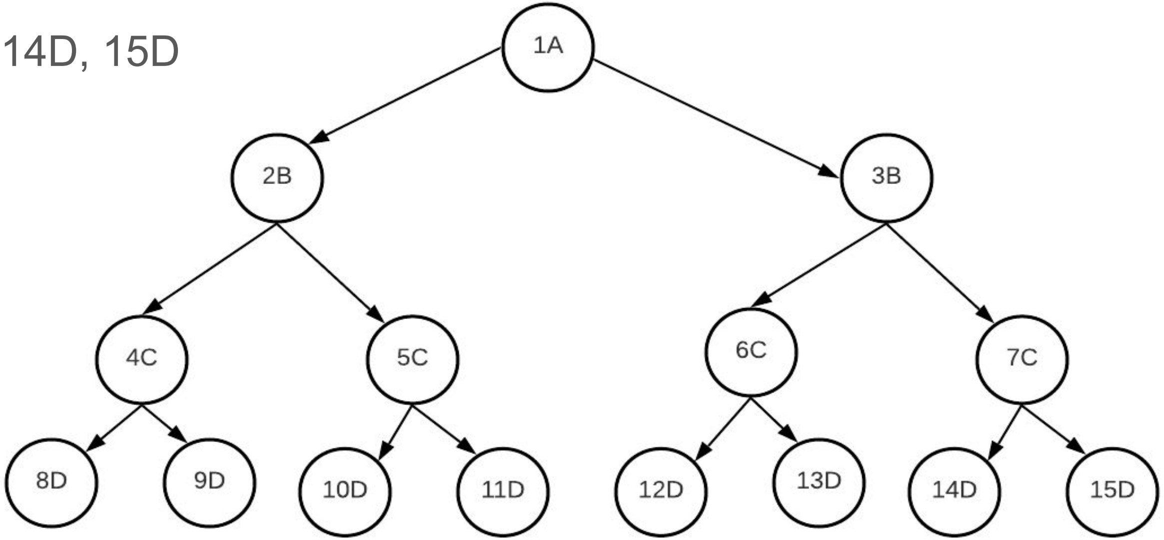
```
void level_order_traversal1() {
    queue<BinaryTree*> nodes_queue;
    nodes_queue.push(this);

    while (!nodes_queue.empty()) {
        BinaryTree*cur = nodes_queue.front();
        nodes_queue.pop();

        cout << cur->data << " ";
        if (cur->left)
            nodes_queue.push(cur->left);
        if (cur->right)
            nodes_queue.push(cur->right);
    }
    cout << "\n";
}
```

Print level by level, knowing level

- Let’s assume the queue now has ONLY nodes from level 5
  - Assume they are 4 nodes. Let’s call it sz
- Process sz # of times the queue
  - Now the sz (4) nodes are removed!
  - Only their children are added

Process based on current size

- 1A sz = 1, Process 1 step
- 2B, 3B sz = 2, Process 2 steps
- 4C, 5C, 6C, 7C sz = 4
- 8D, 9D, 10D, 11D, 12D, 13D, 14D, 15D



Implementation v2

- Now can trivially know which level we are
- In each step
  - We process all current parents
  - Add all their children
  - Hence, always one level in the queue
- Both methods are O(n) time
  - We iterate on each node: ~n
  - We move through each edge: ~n
    - A tree has n-1 edges

```cpp
void level_order_traversal2() {
    queue<BinaryTree*> nodes_queue;
    nodes_queue.push(this);

    int level = 0;
    while (!nodes_queue.empty()) {
        int sz = nodes_queue.size();

        cout<<"Level "<<level<<": ";
        while(sz--) {
            BinaryTree*cur = nodes_queue.front();
            nodes_queue.pop();

            cout << cur->data << " ";
            if (cur->left)
                nodes_queue.push(cur->left);
            if (cur->right)
                nodes_queue.push(cur->right);
        }
        level++;
        cout << "\n";
    }
}
```

Time Complexity

- Fact: A tree of nodes has always n-1 edges (think about it)
- Time complexity
  - In both recursive and level traversals: we iterate on each node ⇒ ~n steps
  - From each node, we pass on its children. **Total** edges ~n
    - Don’t just say it is 2 max as constant! Think total here
  - So total O(n) time

Memory Complexity

- In recursion, we have a **stack** of depth h. So O(h)
- But for level order? We have a queue of items
- We know, the queue will never have more than n nodes, so o(n)
    - But actually, will have only subset of them: the max level per a tree
- Ok, then in a perfect tree, we have max of 2^h nodes in last level so o(2^h)
    - But if the tree is degenerate, this means we have 2^n nodes while in fact we have only O(1)
- Overall, this should encourage the following choices
    - The best case: O(1) for degenerate tree
    - The worst case: For a perfect tree we have O(2^h). As h = ~log n. Then again O(n)
        - Math Tip: 2 ^ logn = n
    - **Overall: a better representation is O(n) memory complexity**

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."