List of Depths: Given a binary tree, design an algorithm that creates a linked list of all the nodes at each depth.

If you have a tree of depth D, you'll have D number of linked lists.

Ex)

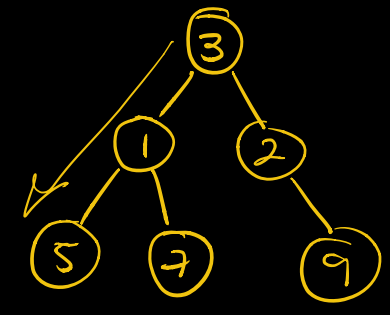


Fig 1.
A binary tree

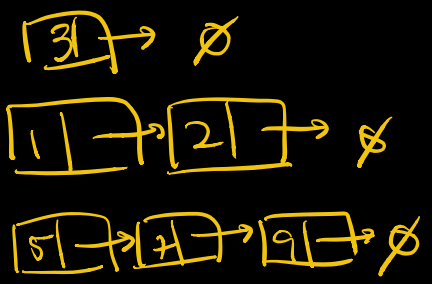
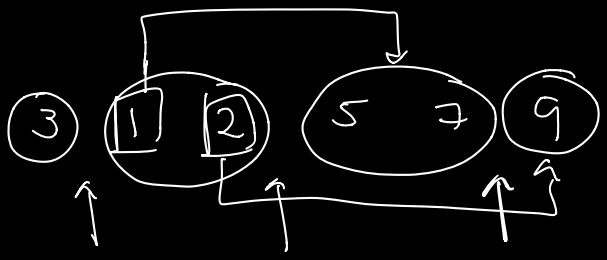


Fig 2.
Linked lists for each depth of Fig 1.

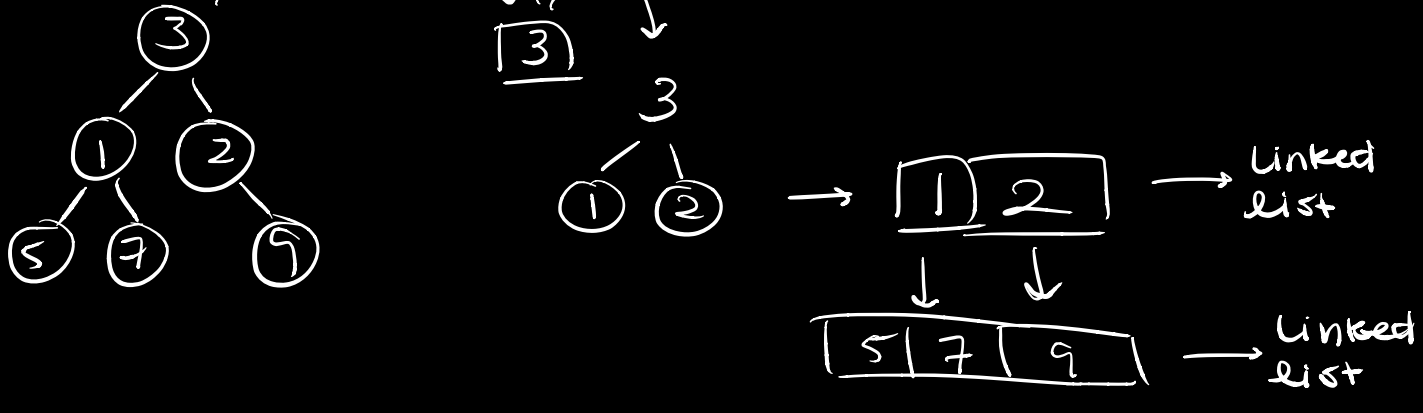
D = 3,
3 linked lists



Depth first search queue

How do we know these are the partitions?

make a linked list



- Have a list hold all linked lists
- At each level, create a linked list of all nodes in that level
  - starting with the root, store its children
  - reset linked list after visiting each level

Breadth-First Search Solution (Iterative)

```
function createLinkedLists(root) {
    let results = [ ]; // Array for containing linked lists
    let currentLevel = new LinkedList();
    if(root !== null) {
        currentLevel.append(root);
    }
    while(currentLevel.size > 1 ){
        results.push(currentLevel);
        let parents = currentLevel;
        currentLevel = new LinkedList();
        for(parent in parents){
            if(parent.left !== null){
                currentLevel.append(parent.left);
            }
            if(parent.right !== null){
                currentLevel.append(parent.right);
            }
        }
    }
    return results;
}
```

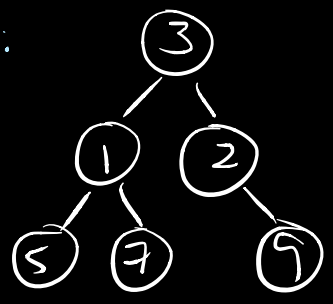
1
2/4
3/5

Complexity Analysis:

- Time: O(n), we end up visiting each node in the tree once.
- Space: O(n), for the creation of the linked lists that gets returned

Illustrated Test:

given:



1. results = [ ]
   currentLevel = [3] → ∅
   ↓
2. enter while loop
   results = [[3] → ∅]
   parents = [3] → ∅
   ↓
3. enter for loop for parents
   currentLevel = [1] → [2] → ∅
4. back to the beginning of the while loop
   results = [[3] → ∅ , [1] → [2] → ∅]
   parents = [1] → [2] → ∅
5. currentLevel = [5] → [7] → [9] → ∅
6. back to the beginning of the while loop, again
   results = [[3] → ∅ , [1] → [2] → ∅ , [5] → [7] → [9] → ∅]

– end –

Recursive Solution (depth-first search via preorder traversal)

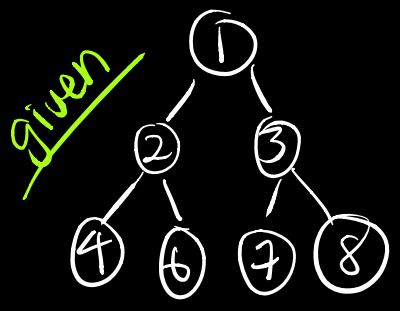
```
function createLinkedLists(root) {
    let results = [ ];
    createLinkedList(root, results, 0);
    return results;
}

function createLinkedList(root, results, level){
    if (root === null){
        return;
    }
    let result = null;
    if( results.size == level) {
        result= new LinkedList();
        results.push(result);
    } else {
        result = results.get(level);
    }
    result.append( root );
    createLinkedList(root.left, results, level +1);
    createLinkedList(root.right, results, level +1);
}
```

Complexity Analysis

Time: O(n), we visit each node in the tree once

Space: O(n), although the recursive calls result in O(log n) of the stack to be used, we are still returning linked lists of the same size as the tree that is given. Therefore, the larger complexity trumps the smaller one.

running through the function:



① root = 1
results = [ ] = 0
level = 0
results = [∅]
result = 1 → ∅

② root = 2
results = [1 → ∅] = 1
level = 1
results = [1 → ∅, ∅]
result = 2 → ∅

③ root = 4
results = [1 → ∅, 2 → ∅]
level = 2
results = [
result = 4 → ∅

③ root = 6
results = [1 → ∅, 2 → ∅, 4 → ∅]
level = 2
go to else statement!
result = 4 → 6 → ∅

② root = 3
results = [1 → ∅, 2 → ∅, 4 → 6 → ∅]

level = 1
result = 2 → 3 → ∅