DFS

- DFS is a traversal method for binary search trees that explores as far as possible along each branch before backtracking.
- It prioritizes depth over breadth, traversing down left or right subtrees to their ends.
- Common DFS methods include Inorder, Preorder, and Postorder traversals.
- Best implemented using recursion, though iterative approaches with a stack are possible but more complex.

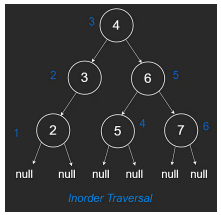
Depth First Search (DFS) is a way of traversing binary search trees that prioritizes depth rather than breadth. The idea is to keep traversing down either the left subtree or the right subtree until there are no more nodes left. There are various methods under which depth-first search is performed. These methods visit the nodes - root, $\text{left}_c hild, right_c hildindifferentorders. Thesethreemethodsare$: $Inorder, Preorder, PostorderDepthfirstsearchisbestimplementedusingrecursion. Again, youcouldus$

Inorder Traversal

Inorder Traversal

- Visits left-child, parent, then right-child. Results in sorted order for BSTs.
- Inorder traversal will result in the nodes being visited in a sorted order.
- Order: [2,3,4,5,6,7] for a tree with nodes [4,3,6,2,null,5,7]

```
def inorder(root):
    if not root:
        return
    inorder(root.left)
    print(root.val)
    inorder(root.right)
```

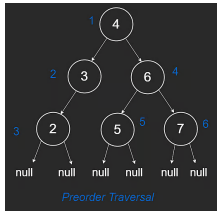


An inorder traversal prioritizing left before right will visit the left-child first, then the parent node and then the right-child. Inorder traversal will result in the nodes being visited in a sorted order. The order in which these nodes will be visited is [2,3,4,5,6,7], which is sorted. It is important to note that this only works when left is prioritized before right. If we prioritize right before left, we will end up with a reverse sorted array. The reason the nodes will print in a sorted order is because of the BST property. Since we know all values to the left of a node are smaller, this means we won't hit our base case until we reach the left-most node which is also the smallest node. After visiting this, we will traverse up, visit the parent and then visit the right-subtree. The visual below shows this process.

Preorder Traversal

- A preorder traversal prioritizing left before right will visit the parent, the left-child and the right-child, in that order.
- Order: [4,3,2,6,5,7] for a tree with nodes [4,3,6,2,null,5,7]

```
def preorder(root):
    if not root:
        return
    print(root.val)
    preorder(root.left)
    preorder(root.right)
```

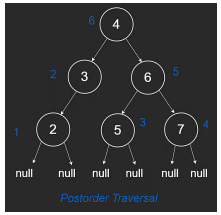


Postorder Traversal

Postorder Traversal

- A post-order traversal prioritizing left before right will visit the left_child, the right_child and the parent, in that order.
- Order: [2,3,5,7,6,4] for a tree with nodes [4,3,6,2,null,5,7]

```
def postorder(root):
    if not root:
        return
    postorder(root.left)
    postorder(root.right)
    print(root.val)
```



Concept

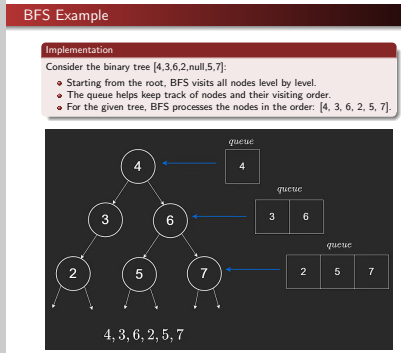
BFS

- In BFS, unlike Depth-First Search, we prioritize breadth, focusing on visiting all nodes at one level before moving to the next.
- BFS is generally implemented iteratively, using a queue to keep track of the nodes at the current level and to visit their children in order.

```
def bfs(root):
    queue = deque()

    if root:
        queue.append(root)

    level = 0
    while len(queue) > 0:
        print("level: ", level)
        for i in range(len(queue)):
            curr = queue.popleft()
            print(curr.val)
            if curr.left:
                queue.append(curr.left)
            if curr.right:
                queue.append(curr.right)
        level += 1
```

In depth-first search, we prioritized depth. For breath-first serach, we prioritize breadth. We focus visiting all the nodes on one level before moving on to the next level. Generally, breadth-first search is implemented iteratively and that is the implementation we will be covering in this course. You can write it recursively but it is a lot more challenging.

BFS makes use of a queue data structure, more specifically, a deque, allowing us to remove elements both from the head and the tail in O(1) time.



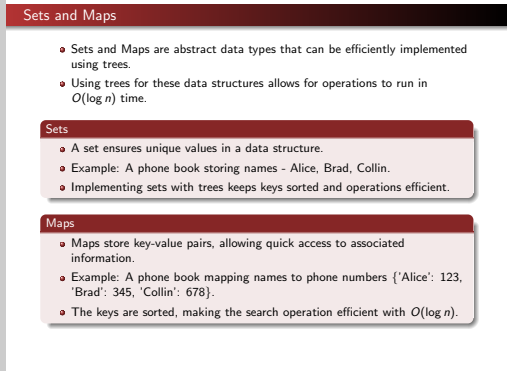
Let's take an example of a binary tree, [4,3,6,2,null,5,7] and apply the BFS algorithm. Remember, our goal is to visit all the nodes on one level before moving to the next.
If we append our root node to our queue and loop through the queue such that at any given time, our queue only holds the nodes on a certain level, we will ensure that we visit the levels in order and that we don't mix up the levels either. This is exactly what the code inside of the while loop achieves. As long as our queue is not empty, we remove the node(s) that is present in our queue and add its children to the queue (which would be the next level). Therefore, when we remove root, we add its children, which are 3,6 to the queue. Next, we remove 3 and add its child 2. We then remove 6 and add 5,7 to the queue. Because of the FIFO nature of a queue, we ensure that we visit the nodes from left to right.

Our queue becomes empty once we have visited all of the nodes.

Time Complexity

- The work done by BFS is proportional to the number of nodes in the tree, denoted as n.
- Since it performs a constant amount of work per node, the total time complexity is $O(n)$.

Technically, the total work done is cn where n is the number of nodes in the tree and c is the amount of work we perform at each node. We performed a total of three operations per node - printing the node, appending the node, and removing it. This is what the c represents. For the case of asymptotic analysis, we can drop this constant, meaning the algorithm belongs to O(n).



Imagine we have a phone book with three names - Alice, Brad, Collin. We could store these using dynamic arrays but a set ensures that we have unique values in our data structure, and implementing it using a tree ensures that our keys are sorted alphabetically.

A map on the other hand operates on key-value pairs. Going back to our phone book example, not only can we store names, but also map them to their phone numbers. Again, the data structure is sorted by the key. Because the key maps to a value, we can find all of the information associated with a key. The value here doesn't have to be a phone number, it can also be an object, etc.

The operations performed on a TreeMap run in O(log n) time. The mapping for the phonebook would look like the following. 'Alice' : 123, 'Brad' : 345, 'Collin' : 678

Backtracking

- Backtracking is a systematic way to iterate through all possible configurations of a search space.
- It is similar to Depth-First Search (DFS) but focuses on solving problems by exploring all possible combinations and backtracking from unsuccessful attempts.
- Useful for problems where an exhaustive search is required, such as finding all combinations of a pattern lock.

Backtracking is an algorithm that's similar to DFS on binary trees, which we have already discussed. It operates on a brute-force approach. Imagine that we had to search for "all possible combinations of a pattern lock". We would have to do an exhaustive search to find all possible combinations i.e. there really is not a better algorithm to get all combinations than to search for all of the combinations one by one. This is the idea behind backtracking. We explore a possible way to perform a task and if we do not succeeed, we backtrack and explore other ways until we find a solution.

Motivation with Example

Problem Statement

Determine if a path exists from the root of a tree to a leaf node that does not contain any zeroes.

- Depth-First Search (DFS) can be used, focusing on paths that avoid nodes with value 0.
- If a leaf node is reached without encountering a 0 value, a valid path exists.

Q: Determine if a path exists from the root of the tree to a leaf node. It may not contain any zeroes. The problem is basically asking us if we can traverse from the root node to the leaf node without having a value of 0. We return true if there exists a path and false if there does not.

The first thing that comes to mind is using depth-first search. Our constraint is that we cannot have a node with value 0 in our path. We also know that if the tree is empty, then there cannot exist a valid path either. Finally, if we reach a leaf node and have not returned false, we can return true since it means there is a path that exists from root to leaf.

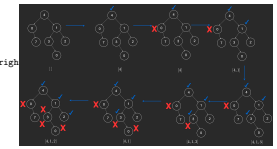
For the sake of this problem, let's assume that there exists exactly one path, so it must exist either in the right-subtree or the left-subtree. Arbitrarily, we choose to try the left side before right. If the answer was not found in the left-subtree, the algorithm will search in the right-subtree and if the path exists, it will return true.

Given the tree, [4,0,1,null,7,2,0], the valid path would look like the following, as shown in the visual. A path is invalid if it has a 0 in it.

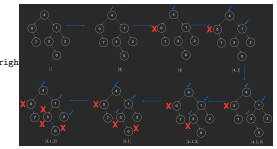
```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

def canReachLeaf(root):
    if not root or root.val == 0:
        return False

    if not root.left and not root.right:
        return True
    if canReachLeaf(root.left):
        return True
    if canReachLeaf(root.right):
        return True
    return False
```

Returning Path Value

```
def leafPath(root, path):
    if not root or root.val == 0:
        return False
    path.append(root.val)

    if not root.left and not root.right:
        return True
    if leafPath(root.left, path):
        return True
    if leafPath(root.right, path):
        return True
    path.pop()
    return False
```



```
def leafPath(root, path):
    if not root or root.val == 0:
        return False
    path.append(root.val)

    if not root.left and not root.righ
        return True
    if leafPath(root.left, path):
        return True
    if leafPath(root.right, path):
        return True
    path.pop()
    return False
```



In this problem, we can pass a parameter path, which is a list to store all the nodes that are in the valid path. So, given the tree [4,0,1,null,7,3,2,null,null,null,0], we first add the root node to our list.

Since there is only one valid path, it will either be in the left-subtree or the right-subtree. Prioritizing left over right, the left-subtree is invalid because 4's left child is 0. We return false and now recursively check the right-subtree. Going to the right, 1 is valid, so we add it to our list. Now, we check 3, which is valid, so it gets added to our list. 3's left child is null, so we return false. Checking 3's right child, we again hit the base case. Now, we must remove 3 from our stack because if there existed a valid path, we would have returned true already. We go back up to the 3's parent, which is 1, and check its right subtree. We add 2 to our list. We then explore 2 but 2 is a leaf node, which makes the recursive call return true, after which the function returns true. Our valid path is [4,1,2].