DFS-BFS

Presented by Yasin Ceran

February 21, 2024

Depth-First Search (DFS)

DFS

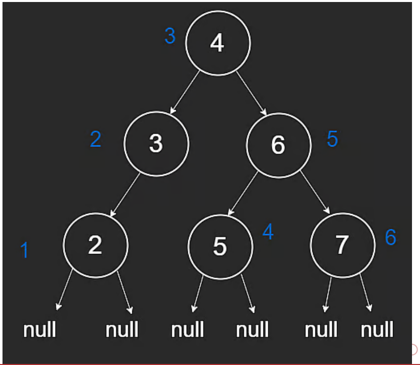
- DFS is a traversal method for binary search trees that explores as far as possible along each branch before backtracking.
- It prioritizes depth over breadth, traversing down left or right subtrees to their ends.
- Common DFS methods include Inorder, Preorder, and Postorder traversals.
- Best implemented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

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)
```

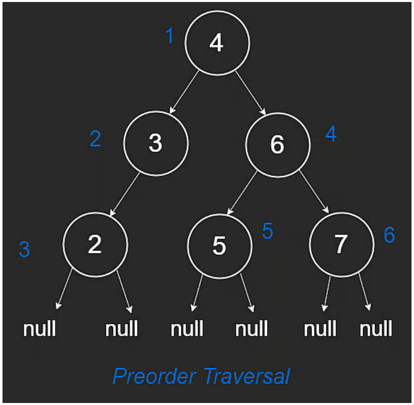


Preorder Traversal

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)
```

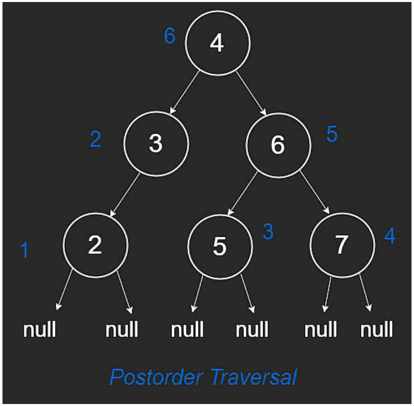


Postorder Traversal

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)
```



Time and Space Complexity

- All nodes must be visited, leading to a time complexity of $O(n)$ for each traversal method.
- Sorting an array via BST insertion and Inorder traversal yields a time complexity of $O(n \log n)$, considering n insertions at $O(\log n)$ each, plus n steps for traversal.
- The effective complexity for sorting with a BST and DFS traversal is $O(n \log n)$.

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.

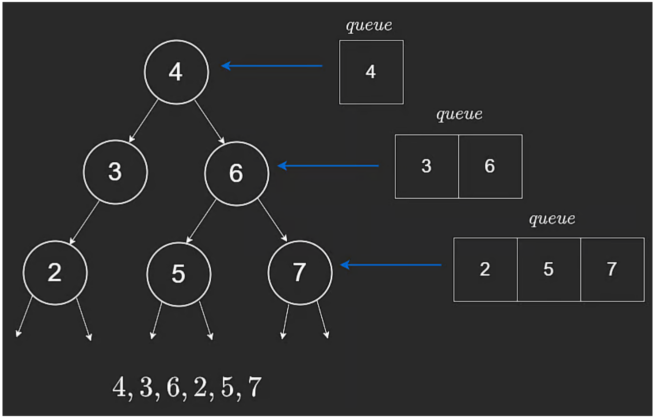
```
def bfs(root):
    queue = deque()

    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
```

BFS Example

Implementation

Consider the binary tree [4,3,6,2,null,5,7]:

- Starting from the root, BFS visits all nodes level by level.
- The queue helps keep track of nodes and their visiting order.
- For the given tree, BFS processes the nodes in the order: [4, 3, 6, 2, 5, 7].



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 $O(n)$.

Sets and Maps

- Sets and Maps are abstract data types that can be efficiently implemented using trees.
- Using trees for these data structures allows for operations to run in $O(\log n)$ time.

Sets

- A set ensures unique values in a data structure.
- Example: A phone book storing names - Alice, Brad, Collin.
- Implementing sets with trees keeps keys sorted and operations efficient.

Maps

- Maps store key-value pairs, allowing quick access to associated information.
- Example: A phone book mapping names to phone numbers {'Alice': 123, 'Brad': 345, 'Collin': 678}.
- The keys are sorted, making the search operation efficient with $O(\log n)$.

Implementation in Different Languages

Python

```
from sortedcontainers import SortedDict
treemap = SortedDict({'c': 3, 'a': 1, 'b': 2})
```

Java

```
TreeMap<String, String> treeMap =
    new TreeMap<String, String>();
```

JavaScript

```
const TreeMap = require("treemap-js");
let map = new TreeMap();
```

C++

```
map<string, string> treeMap;
```

Backtracking

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.

Motivation with Example

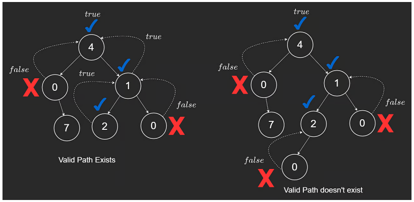
Problem Statement

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.

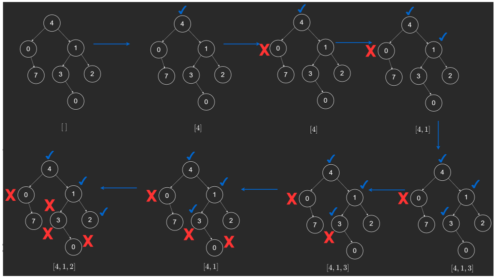
Motivation with Example-Solution

```
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
        return True
    if leafPath(root.left, path):
        return True
    if leafPath(root.right, path)
        return True
    path.pop()
    return False
```



Time Complexity

- For a binary tree with n nodes, the time complexity of backtracking algorithms like the ones presented is $O(n)$.
- This is because, in the worst case, every node in the tree needs to be visited.

Summary:

- **Depth-First Search (DFS):** Explores as far as possible along branches before backtracking. Ideal for exploring tree or graph structures, identifying paths, and solving puzzles.
- **Breadth-First Search (BFS):** Explores all neighbors at the current depth before moving to nodes at the next depth level. Used for shortest path problems and hierarchy exploration.
- **BST for Sets and Maps:**
    - *Sets* - Utilize the unique value property of BSTs for efficient search, insertion, and deletion.
    - *Maps* - Key-value pairs are sorted by key in a BST, allowing for efficient retrieval and maintenance of associative arrays.
- **Tree Maze:** A conceptual application combining tree traversal techniques to navigate mazes, where decisions at branches determine the path through the maze.