Breadth First Search (BFS) for Binary Trees - Complete Technical Guide

Table of Contents

Introduction

Breadth First Search (BFS) is a tree/graph traversal algorithm that explores nodes level by level, visiting all nodes at depth `d` before moving to nodes at depth `d+1`. For binary trees, BFS is also known as **level-order traversal**.

Key Characteristics:

- Uses a **queue** data structure (FIFO - First In First Out)
- Explores nodes horizontally before going deeper
- Guarantees shortest path in unweighted trees
- Natural choice for level-based problems

When to Use BFS:

- Level-order traversal needed
- Finding shortest path/minimum depth
- Processing nodes level by level
- Zigzag traversal patterns
- Finding nodes at specific distances

Core Implementation

Basic Tree Node Structure

cpp

```cpp
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
```

Standard BFS Template

cpp

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

// Basic BFS - Print all nodes level by level
void bfsTraversal(TreeNode* root) {
    if (!root) return;

    queue<TreeNode*> q;
    q.push(root);

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

        cout << current->val << " ";

        if (current->left) q.push(current->left);
        if (current->right) q.push(current->right);
    }
}

// Level-by-Level BFS (returns vector of vectors)
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> currentLevel;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            currentLevel.push_back(node->val);
```

```
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        result.push_back(currentLevel);
    }

    return result;
}
```

Complexity Analysis

Time Complexity: O(n)

- Each node is visited exactly once
- Each node is enqueued and dequeued once
- n = total number of nodes in the tree

Space Complexity: O(w)

- w = maximum width of the tree (maximum nodes at any level)
- Worst case: Complete binary tree → O(n/2) ≈ O(n)
- Best case: Skewed tree → O(1)
- Average case for balanced tree: O(n/2) at the last level

Interview Question Categories

1. Basic Level-Order Traversal

- Print nodes level by level
- Return 2D vector of levels
- Level order in single array

2. Directional Traversals

- Zigzag/spiral level order
- Bottom-up level order
- Right-to-left traversal

3. Level-Based Queries

- Average of each level
- Maximum value in each level
- Find level with maximum sum
- Count nodes at each level

4. Specific Node Finding

- Right side view of tree
- Left side view of tree
- Find deepest/leftmost leaf
- Nodes at distance K

5. Tree Properties

- Minimum/maximum depth
- Check if tree is complete
- Check if tree is perfect
- Width of binary tree

6. Tree Modifications

- Populate next right pointers
- Connect nodes at same level
- Level-order successor/predecessor

7. Cousin/Relative Problems

- Check if two nodes are cousins
- Find all cousins of a node
- Check if nodes are siblings

Common Patterns & Solutions

Pattern 1: Level-by-Level Processing

cpp

```cpp
// Average of Levels in Binary Tree
vector<double> averageOfLevels(TreeNode* root) {
    vector<double> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        double sum = 0;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
            sum += node->val;

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        result.push_back(sum / levelSize);
    }

    return result;
}
```

Pattern 2: Zigzag/Spiral Traversal

cpp

```cpp
// Binary Tree Zigzag Level Order Traversal
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);
    bool leftToRight = true;

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> currentLevel(levelSize);

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            // Find position to insert
            int index = leftToRight ? i : (levelSize - 1 - i);
            currentLevel[index] = node->val;

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        leftToRight = !leftToRight;
        result.push_back(currentLevel);
    }

    return result;
}
```

Pattern 3: Tree Views (Right/Left Side)

cpp

```cpp
// Binary Tree Right Side View
vector<int> rightSideView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            // Last node of each level
            if (i == levelSize - 1) {
                result.push_back(node->val);
            }

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }

    return result;
}

// Binary Tree Left Side View
vector<int> leftSideView(TreeNode* root) {
    vector<int> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
```

```
            q.pop();

            // First node of each level
            if (i == 0) {
                result.push_back(node->val);
            }

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }

    return result;
}
```

Pattern 4: Minimum/Maximum Depth

cpp

```
// Minimum Depth of Binary Tree
int minDepth(TreeNode* root) {
    if (!root) return 0;

    queue<TreeNode*> q;
    q.push(root);
    int depth = 1;

    while (!q.empty()) {
        int levelSize = q.size();

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            // First leaf node found
            if (!node->left && !node->right) {
                return depth;
            }

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        depth++;
    }

    return depth;
}

// Maximum Depth of Binary Tree (BFS approach)
int maxDepth(TreeNode* root) {
    if (!root) return 0;

    queue<TreeNode*> q;
    q.push(root);
    int depth = 0;

    while (!q.empty()) {
        int levelSize = q.size();
        depth++;
```

```
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }

    return depth;
}
```

Pattern 5: Cousin Problems

cpp

```cpp
// Check if Two Nodes are Cousins
bool isCousins(TreeNode* root, int x, int y) {
    if (!root) return false;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        bool xFound = false, yFound = false;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            // Check if x and y are children of same parent
            if (node->left && node->right) {
                if ((node->left->val == x && node->right->val == y) ||
                    (node->left->val == y && node->right->val == x)) {
                    return false; // Same parent, not cousins
                }
            }

            // Check if x or y found at this level
            if (node->val == x) xFound = true;
            if (node->val == y) yFound = true;

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        // If both found at same level but not same parent
        if (xFound && yFound) return true;
        if (xFound || yFound) return false; // Only one found
    }

    return false;
}
```

Pattern 6: Bottom-Up Level Order

cpp

```cpp
// Binary Tree Level Order Traversal II (Bottom-Up)
vector<vector<int>> levelOrderBottom(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> currentLevel;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

            currentLevel.push_back(node->val);

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        result.push_back(currentLevel);
    }

    // Reverse the result
    reverse(result.begin(), result.end());
    return result;
}
```

Pattern 7: Nodes at Distance K

cpp

```cpp
// All Nodes Distance K from Target (requires parent tracking)
vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
    if (!root) return {};

    // Build parent map using BFS
    unordered_map<TreeNode*, TreeNode*> parent;
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();

        if (node->left) {
            parent[node->left] = node;
            q.push(node->left);
        }
        if (node->right) {
            parent[node->right] = node;
            q.push(node->right);
        }
    }

    // BFS from target node
    unordered_set<TreeNode*> visited;
    queue<TreeNode*> bfs;
    bfs.push(target);
    visited.insert(target);
    int distance = 0;

    while (!bfs.empty()) {
        if (distance == k) {
            vector<int> result;
            while (!bfs.empty()) {
                result.push_back(bfs.front()->val);
                bfs.pop();
            }
            return result;
        }

        int levelSize = bfs.size();
```

```
        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = bfs.front();
            bfs.pop();

            // Check left child
            if (node->left && !visited.count(node->left)) {
                bfs.push(node->left);
                visited.insert(node->left);
            }

            // Check right child
            if (node->right && !visited.count(node->right)) {
                bfs.push(node->right);
                visited.insert(node->right);
            }

            // Check parent
            if (parent.count(node) && !visited.count(parent[node])) {
                bfs.push(parent[node]);
                visited.insert(parent[node]);
            }
        }

        distance++;
    }

    return {};
}
```

Pattern 8: Populate Next Right Pointers

cpp

```cpp
// Node definition with next pointer
struct Node {
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node(int x) : val(x), left(nullptr), right(nullptr), next(nullptr) {}
};

// Populate Next Right Pointers (Perfect Binary Tree)
Node* connect(Node* root) {
    if (!root) return nullptr;

    queue<Node*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        Node* prev = nullptr;

        for (int i = 0; i < levelSize; i++) {
            Node* node = q.front();
            q.pop();

            if (prev) {
                prev->next = node;
            }
            prev = node;

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        // Last node of level points to nullptr (already set)
    }

    return root;
}
```

Pattern 9: Check Complete Binary Tree

cpp

```cpp
// Check if Binary Tree is Complete
bool isCompleteTree(TreeNode* root) {
    if (!root) return true;

    queue<TreeNode*> q;
    q.push(root);
    bool nullSeen = false;

    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();

        if (!node) {
            nullSeen = true;
        } else {
            if (nullSeen) return false; // Node after null
            q.push(node->left);
            q.push(node->right);
        }
    }

    return true;
}
```

Pattern 10: Maximum Width of Binary Tree

cpp

```cpp
// Maximum Width of Binary Tree
int widthOfBinaryTree(TreeNode* root) {
    if (!root) return 0;

    queue<pair<TreeNode*, long long>> q;
    q.push({root, 0});
    long long maxWidth = 0;

    while (!q.empty()) {
        int levelSize = q.size();
        long long leftmost = q.front().second;
        long long rightmost = leftmost;

        for (int i = 0; i < levelSize; i++) {
            auto [node, idx] = q.front();
            q.pop();
            rightmost = idx;

            if (node->left) {
                q.push({node->left, 2 * idx});
            }
            if (node->right) {
                q.push({node->right, 2 * idx + 1});
            }
        }

        maxWidth = max(maxWidth, rightmost - leftmost + 1);
    }

    return maxWidth;
}
```

Advanced Techniques

1. Multi-Source BFS

Used when starting from multiple nodes simultaneously.

cpp

```cpp
// Shortest Distance from Multiple Sources
int shortestDistance(TreeNode* root, vector<int>& sources) {
    if (!root) return -1;

    queue<TreeNode*> q;
    unordered_set<int> sourceSet(sources.begin(), sources.end());

    // Initialize with all source nodes
    queue<TreeNode*> init;
    init.push(root);
    while (!init.empty()) {
        TreeNode* node = init.front();
        init.pop();

        if (sourceSet.count(node->val)) {
            q.push(node);
        }

        if (node->left) init.push(node->left);
        if (node->right) init.push(node->right);
    }

    // Regular BFS from all sources
    // ... rest of implementation
    return 0;
}
```

2. Bidirectional BFS

Search from both ends simultaneously to reduce time complexity.

3. BFS with State Tracking

Track additional state information during traversal.

cpp

```
struct State {
    TreeNode* node;
    int sum;
    int level;
};

void bfsWithState(TreeNode* root) {
    queue<State> q;
    q.push({root, root->val, 0});

    while (!q.empty()) {
        State current = q.front();
        q.pop();

        // Process with state information
        // ...
    }
}
```

Interview Tips

Common Mistakes to Avoid:

1. **Forgetting to check for nullptr before pushing to queue**
2. **Not capturing level size before the loop** (queue size changes during iteration)
3. **Off-by-one errors in level counting**
4. **Forgetting to handle empty tree case**
5. **Memory leaks when working with dynamically allocated nodes**

Optimization Strategies:

1. Use `reserve()` for vectors when size is known
2. Avoid unnecessary copying of data structures
3. Consider iterative vs recursive trade-offs
4. Use references to avoid copying large structures

Testing Checklist:

- Empty tree (nullptr)
- Single node tree
- Left-skewed tree
- Right-skewed tree
- Complete binary tree
- Perfect binary tree
- Tree with only left children

- Tree with only right children

Practice Problem List

Easy:

1. Binary Tree Level Order Traversal
2. Average of Levels in Binary Tree
3. Minimum Depth of Binary Tree
4. Maximum Depth of Binary Tree
5. Binary Tree Right Side View
6. Cousins in Binary Tree

Medium:

1. Binary Tree Zigzag Level Order Traversal
2. Binary Tree Level Order Traversal II
3. Populating Next Right Pointers
4. All Nodes Distance K in Binary Tree
5. Maximum Width of Binary Tree
6. Check Completeness of Binary Tree
7. Find Bottom Left Tree Value
8. Add One Row to Tree

Hard:

1. Serialize and Deserialize Binary Tree
2. Vertical Order Traversal of Binary Tree
3. Binary Tree Maximum Path Sum (with BFS approach)

Conclusion

BFS is a fundamental algorithm for binary tree problems, especially when dealing with level-based operations. The key is recognizing when a problem requires level-by-level processing and applying the appropriate pattern. Master the core template, understand the common patterns, and practice extensively to excel in technical interviews.