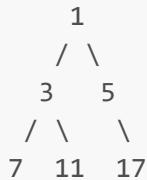


Certainly! The `levelordertraversal` function performs a level-order traversal (also known as a breadth-first traversal) on a binary tree. Let's break down the code step by step, especially considering the tree you mentioned:

Tree Structure:



Step-by-Step Explanation:

1. Initialization:

```
queue<node*>q;  
q.push(root);  
q.push(NULL);
```

- A queue `q` is initialized to assist with the level-order traversal.
- The root node (`1`) is pushed into the queue, followed by a `NULL` marker. This `NULL` is used to indicate the end of the current level.

2. While Loop:

```
while (!q.empty())
```

- The loop runs as long as there are nodes in the queue.

3. Process Each Node:

```
node* temp = q.front();  
q.pop();
```

- The front node of the queue is assigned to `temp` and removed from the queue.

4. Check for Level Completion:

```
if (temp == NULL)
```

- If `temp` is `NULL`, it means we have completed processing all nodes at the current level.

5. Level Completion Actions:

```
cout << endl;
if (!q.empty()) {
    q.push(NULL);
}
```

- Print a newline to separate levels.
- If the queue is not empty (indicating there are more nodes to process in subsequent levels), push another `NULL` marker to denote the end of the next level.

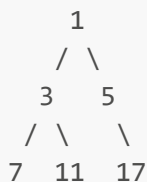
6. Process Non-NULL Nodes:

```
else {
    cout << temp->data << " ";
    if (temp->left) {
        q.push(temp->left);
    }
    if (temp->right) {
        q.push(temp->right);
    }
}
```

- Print the data of the current node.
- If the left child exists, push it to the queue.
- If the right child exists, push it to the queue.

Example Execution:

Given tree:



Let's traverse this step by step:

1. Initial Queue: `[1, NULL]`

2. Process Node 1:

- Print `1`
- Push left child `3`
- Push right child `5`

- Queue: [NULL, 3, 5]
- 3. **End of Level 1:**
 - Encounter NULL
 - Print newline
 - Push NULL
 - Queue: [3, 5, NULL]
- 4. **Process Node 3:**
 - Print 3
 - Push left child 7
 - Push right child 11
 - Queue: [5, NULL, 7, 11]
- 5. **Process Node 5:**
 - Print 5
 - No left child
 - Push right child 17
 - Queue: [NULL, 7, 11, 17]
- 6. **End of Level 2:**
 - Encounter NULL
 - Print newline
 - Push NULL
 - Queue: [7, 11, 17, NULL]
- 7. **Process Node 7:**
 - Print 7
 - No left or right child
 - Queue: [11, 17, NULL]
- 8. **Process Node 11:**
 - Print 11
 - No left or right child
 - Queue: [17, NULL]
- 9. **Process Node 17:**
 - Print 17
 - No left or right child
 - Queue: [NULL]
- 10. **End of Level 3:**
 - Encounter NULL
 - Print newline
 - Queue: [] (empty)

Final Output:

```
1
3 5
7 11 17
```

This traversal ensures that each level of the tree is printed on a new line, and all nodes at the same level are printed together.

How the temp and Queues null is working step by step:

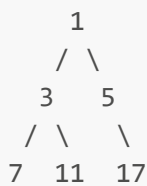
Let's clear up the sequence of events in the level-order traversal to ensure we understand how `temp` can sometimes be `NULL` and sometimes not.

Key Clarification:

- `temp` can be `NULL` after `q.pop()`, but only if the front of the queue was `NULL`.
- When we pop `NULL` from the queue, it signifies the end of a level.
- If the front of the queue is not `NULL`, we pop a node and process it in the `else` block.

Step-by-Step Detailed Walkthrough:

Given the tree:



Initial Setup:

- Queue: `[1, NULL]`

1. First Iteration:

```
node* temp = q.front(); // temp is 1
q.pop();           // Queue becomes [NULL]
```

- `temp` is `1`, which is not `NULL`.
- Execute the `else` block:

```
cout << temp->data << " "; // Output: 1
if (temp->left) q.push(temp->left); // Push 3
if (temp->right) q.push(temp->right); // Push 5
```

- Queue: `[NULL, 3, 5]`

2. Second Iteration:

```
node* temp = q.front(); // temp is NULL
q.pop();           // Queue becomes [3, 5]
```

- `temp` is `NULL`.
- Execute the `if (temp == NULL)` block:

```
cout << endl; // Output: newline
if (!q.empty()) q.push(NULL); // Queue is not empty, push NULL
```

- Queue: `[3, 5, NULL]`

3. Third Iteration:

```
node* temp = q.front(); // temp is 3
q.pop(); // Queue becomes [5, NULL]
```

- `temp` is `3`, which is not `NULL`.
- Execute the `else` block:

```
cout << temp->data << " "; // Output: 3
if (temp->left) q.push(temp->left); // Push 7
if (temp->right) q.push(temp->right); // Push 11
```

- Queue: `[5, NULL, 7, 11]`

4. Fourth Iteration:

```
node* temp = q.front(); // temp is 5
q.pop(); // Queue becomes [NULL, 7, 11]
```

- `temp` is `5`, which is not `NULL`.
- Execute the `else` block:

```
cout << temp->data << " "; // Output: 5
if (temp->left) ; // No left child
if (temp->right) q.push(temp->right); // Push 17
```

- Queue: `[NULL, 7, 11, 17]`

5. Fifth Iteration:

```
node* temp = q.front(); // temp is NULL
q.pop(); // Queue becomes [7, 11, 17]
```

- `temp` is `NULL`.
- Execute the `if (temp == NULL)` block:

```
cout << endl; // Output: newline
if (!q.empty()) q.push(NULL); // Queue is not empty, push NULL
```

- Queue: `[7, 11, 17, NULL]`

6. Subsequent Iterations:

- Continue processing nodes `7`, `11`, and `17` similarly, updating the queue and outputting their values. Each `NULL` marker signifies the end of a level.

Summary:

- When `temp` is `NULL`, we know we've reached the end of a level, and we handle it accordingly.
- When `temp` is not `NULL`, it means we're processing an actual node, and we handle its children.

The queue only contains `NULL` temporarily as a marker. After `q.pop()`, `temp` will be `NULL` only if the front of the queue was `NULL`. Otherwise, `temp` will be the next node in the queue, and the `else` block will execute.

Therefore, after popping a node (like `1` or `3`), `temp` is not `NULL`, and the `else` block executes. When we pop a `NULL`, it marks the end of the level, and we handle that in the `if (temp == NULL)` block.