**Breadth-First Traversal**

Breadth first traversal relies on the queue data structure. Queues allow us to keep a reference to nodes that we want to come back to, even though we have not visited them yet.

A common term for nodes that we add to our queue is **discovered nodes;** a discovered node is one that we add to our queue, whose location we know, but we have yet to actually visit. This is exactly what makes a queue the perfect structure for solving the breadth-first traversal problem.

We start off by adding the root node to our queue, since that is the first node we have access to. This means the root node is the only **discovered node** to start.

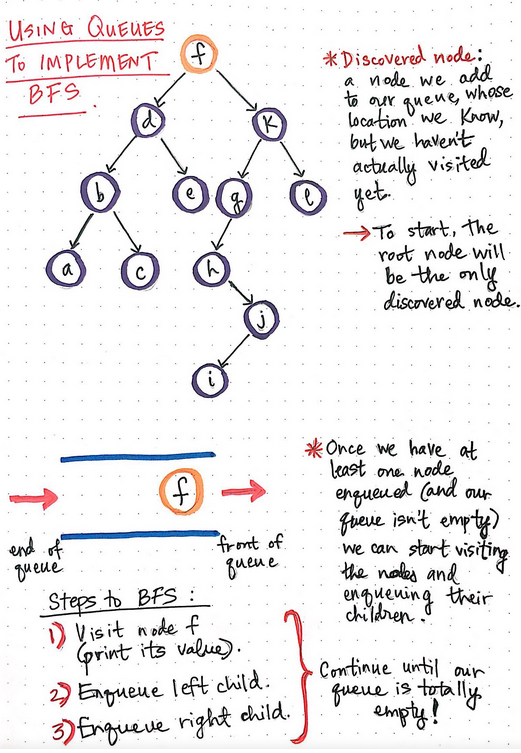
Once we have one node queued, we can start the process of visiting their children nodes. Before we remove the root node from the queue, we add their children to the queue from left to right.

We do the same with the children. Before removing a child from the queue, we add their children to the queue.

So for every node in our queue, starting with the root node, we will want to do three things:

1. **Visit** the node, which usually means printing out its value,
2. **Add** the node’s **left child** to our queue,
3. **Add** the node’s **right child** to our queue.

Once we do these three things, we can remove the node from our queue. We basically need to keep doing this repeatedly until we get to the point where our queue is empty.



|  |  |  |
| --- | --- | --- |
| **Action** | **Queue** | **Visited Node** |
| Add **f** to the queue | **f** |  |
| Add **f’s** children to the queue **(d, k)** | **kdf** |  |
| Remove **f** from the queue | **kd** | **f** |
| Add **d’s** children to the queue **(b, e)** | **ebkd** |  |
| Remove **d** | **ebk** | **d** |
| Add **k’s** children to the queue **(g, l)** | **lgebk** |  |
| Remove **k** | **lgeb** | **k** |
| Add **b’s** children to the queue **(a, c)** | **calgeb** |  |
| Remove **b** | **calge** | **b** |
| Add **e’s** children to the queue (none) | **calge** |  |
| Remove **e** | **calg** | **e** |
| Add **g’s** children to the queue **(h)** | **hcalg** |  |
| Remove **g** | **hcal** | **g** |
| Add **l’s** children to the queue (none) | **hcal** |  |
| Remove **l** | **hca** | **l** |
| Add **a’s** children to the queue (none) | **hca** |  |
| Remove **a** | **hc** | **a** |
| Add **c’s** children to the queue (none) | **hc** |  |
| Remove **c** | **h** | **c** |
| Add **h’s** children to the queue **(j)** | **jh** |  |
| Remove **h** | **j** | **h** |
| Add **j’s** children to the queue **(i)** | **ij** |  |
| Remove **j** | **i** | **j** |
| Add **i’s** children to the queue (none) | **i** |  |
| Remove **i** |  | **i** |

**C++ Implementation of Breadth-First Traversal with Queues**

void BreadthFirstTravseral(struct node\* root)

{

queue<node\*> q;

if (!root) {

return;

}

q.push(root);

while (!q.empty()) {

const node \* const temp\_node = q.front();

cout<<temp\_node->data << " ";

if (temp\_node->left) {

q.push(temp\_node->left);

}

if (temp\_node->right) {

q.push(temp\_node->right);

}

q.pop();

}

}