Breadth First Search (BFS) Graph Traversal in Data Structures

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-86

In the last lecture, we learned about what graph traversal is as well as two of the traversal techniques in brief. These were the Breadth-First Search and the Depth First Search. Today, we'll be dealing solely with the Breadth-First Search abbreviated as BFS in greater detail. I'll also provide you all with some techniques to derive the Breadth-First Search of any graph while only looking at it.

Before we actually move on to writing the algorithm of the Breadth-First Search or programming its implementation, we'll first visualize how Breadth-First Search works.

Graph traversal refers to the process of visiting (checking and/or updating) each vertex(node) in a graph. And since traversing a very large graph manually becomes impossible, some algorithms are used to accomplish this task. One such algorithm of graph traversal is Breadth-First Search.

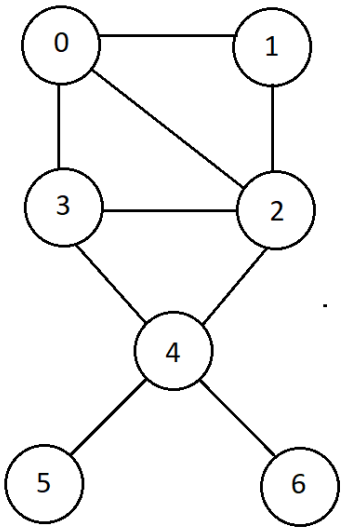
What is Breadth-First Search?

In Breadth-First Search, we start with a node (not necessarily the smallest or the largest) and start exploring its connected nodes. The same process is repeated with all the connecting nodes until all the nodes are visited.

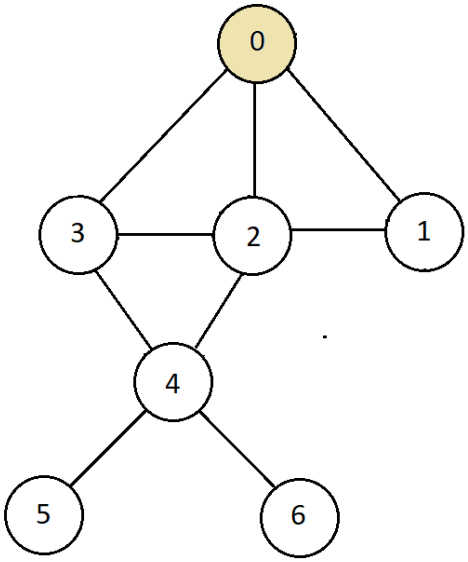
We should first learn the concept of the BFS spanning tree in order to understand the Breadth-First Search in a very intuitive way.

Method 1: BFS Spanning Tree:

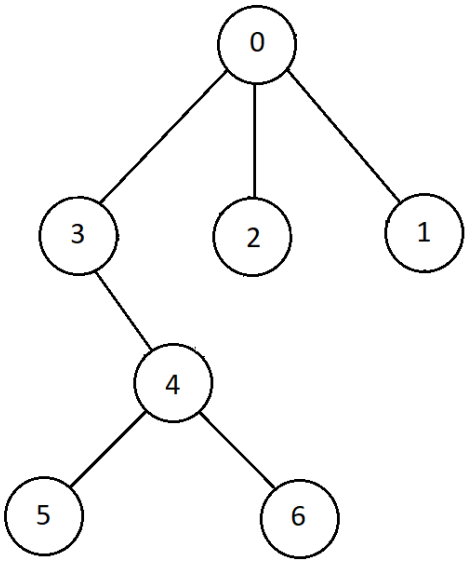
To understand what BFS Spanning Tree means, consider the graph I've illustrated below.



Now, choose any node, say 0, and try to construct a tree with this chosen node as its root. Or basically, hang this whole tree in a gravity-driven environment with this node and assume those edges are not rigid but flexible. So, the graph would now look something like this.



Now, mark dashed or simply remove all the edges which are either sideways or duplicate (above a node) to turn this graph into a valid tree, and as you know for a graph to be a tree, it shouldn't have any cycle. So, we can remove the edges between nodes 2 and 3, and then between nodes 1 and 2 being sideways. Then also between 2 and 4. You could have rather removed the one between node 3 and 4 instead of 2 and 4, but both ways work since these are duplicate to node 4. The tree we receive after we do these above-mentioned changes is,



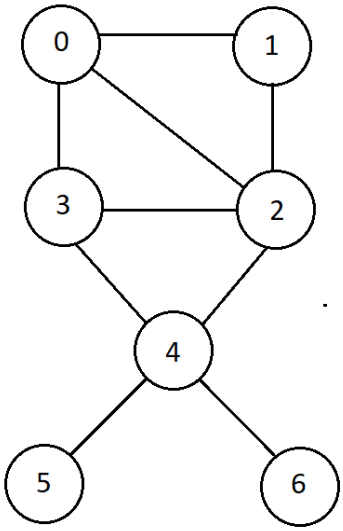
This constructed tree above is known as a BFS Spanning Tree. And surprisingly, the level order traversal of this BFS spanning tree gives us the Breadth-First Search traversal of the graph we started originally with. And if you remember what a level order traversal of a tree is, we simply write the nodes in the same level from left to right. So, the level order traversal of the above BFS Spanning Tree is **0, 3, 2, 1, 4, 5, 6.**

And a BFS Spanning Tree is not unique to a graph. We could have removed, as discussed above, the edge between nodes 3 and 4, instead of nodes 2 and 4. That would have yielded a different BFS Spanning Tree.

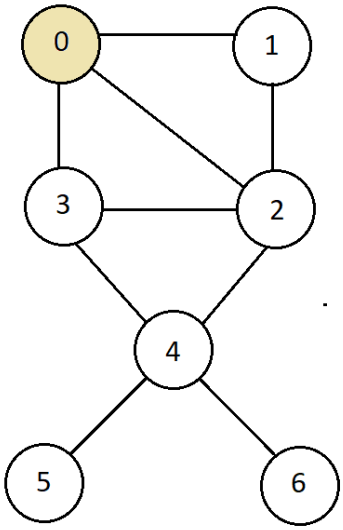
Therefore, given a graph, this is probably the easiest way, I believe, to write the Breadth-First Search traversal of the graph. Here, the chances of making a mistake are minimal. Despite the simplicity of the above technique, there is a very convenient method we follow when we implement the Breadth-First Search traversal algorithm in our program. Let us now discuss that.

Method 2: Conventional Breadth-First Search Traversal Algorithm:

Consider the same graph we covered above. Let me illustrate that again.

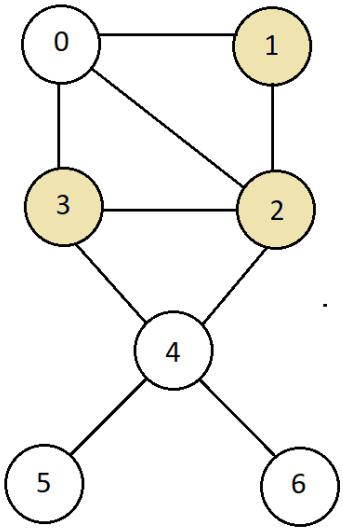


Considering we could begin with any source node; we'll start with 0 only. Let's define a queue named **exploration queue** which would hold the nodes we'll be exploring one by one. We would maintain another array holding the status of whether a node is already **visited** or not. Since we are starting with node 0, we would enqueue 0 into our exploration queue and mark it visited.



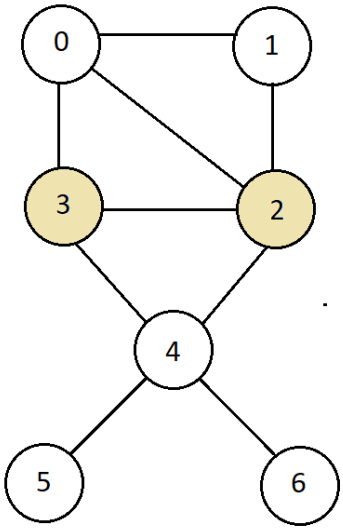
Visited: 0
Exploration Queue: 0

Now, we'll start visiting all the nodes connected to node 0, and remove node 0 from the exploration queue, enqueuing all the currently visited nodes which were nodes 1, 2, and 3. We are pushing them inside the exploration queue because these might further have some unvisited nodes connected to them. Mark these nodes visited as well.



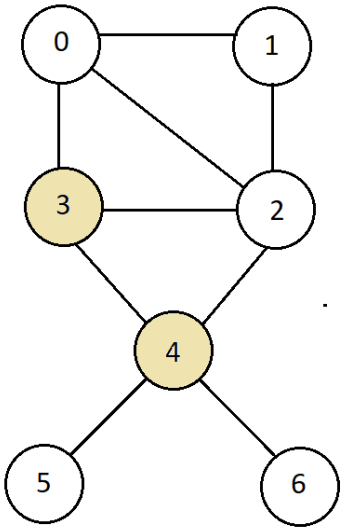
Visited: 0 1 2 3
Exploration Queue: 1 2 3

After this, we have node 1 at the top in the exploration queue, so we'll take it out and visit all unvisited nodes connected to it. Unfortunately, there aren't any. Therefore, we'll continue exploring further.



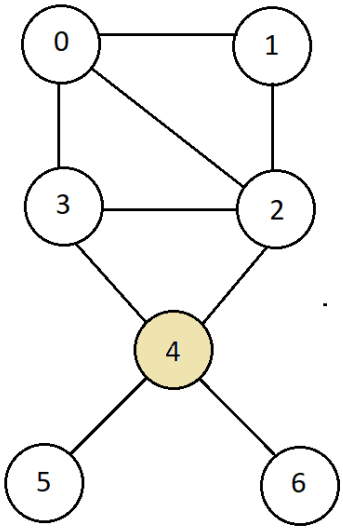
Visited: 0 1 2 3
Exploration Queue: 2 3

Next, we have node 2. And the only unvisited node connected to node 2 is node 4. So, we'll mark it visited and will also enqueue it in our exploration queue.



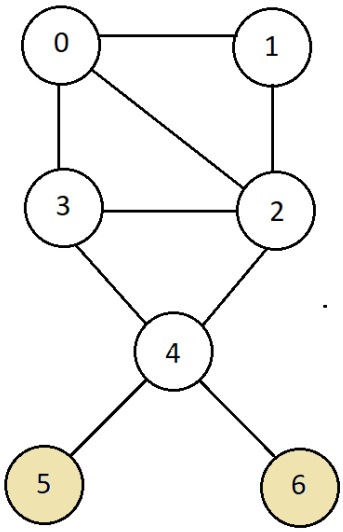
Visited: 0 1 2 3 4
Exploration Queue: 3 4

Node 3 is the next in line. Since, all nodes 1, 2, and 4 which are the nodes connected to it are already visited, we have nothing to do here while we are on node 3.



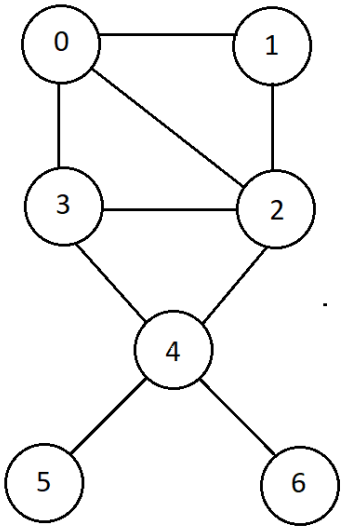
Visited: 0 1 2 3 4
Exploration Queue: 4

Next, we have node 4 on the top in the exploration queue. Let's get it out and see what nodes are connected and unvisited to it. So, we got nodes 5 and 6. Mark them visited and push them inside the exploration queue.



Visited: 0 1 2 3 4 5 6
Exploration Queue: 5 6

And now we can explore the other two nodes left in the queue, and since all nodes are already visited, we'll get nothing in them. And this got our queue emptied and every node traversed in Breadth-First Search manner.



Visited: 0 1 2 3 4 5 6
Exploration Queue:

And the order in which we marked our nodes visited is the Breadth-First Search traversal order. Here, it is **0, 1, 2, 3, 4, 5, 6.** So basically, the visited array maintains whether the node itself is visited or not, and the exploration queue maintains whether the nodes connected to a node are visited or not. This was the difference.

Let's now see how the process we did manually above can be automated in C although we will be using pseudocode for now. In the next lecture, we'll discuss it in more detail.

1. We'll take a whole graph and the information about its nodes and edges as the input along with the source node *s*.
2. We'll mark the node *s* visited and then create a queue, and enqueue *s* in it.
3. We'll then initiate a while loop which will run until the queue is not empty. At each iteration, we will take out the first element out of the queue, and visit all the vertices already not visited and connected to it while enqueueing every new node we visit in the queue.
4. Below is the pseudocode, and I encourage you all to read it and try to implement the same in C.

```
- Input: A graph G = (V,E) and source code s in V
- Algorithm:

- Mark all nodes v  in V as unvisited
- Mark source node s as visited
- enq(Q,s) //First-in first-out Queue
- while(Q is not empty)
      {
          u:=deq(Q);
          for each unvisited neighbour v of u {
              mark v as visited;
              enq(Q,v);
          }
      }
```

Few important points before we leave:

1. We can start with any vertex.
2. There can be multiple Breadth-First Search results for a given graph
3. The order of visiting the vertices can be anything.
4. **Quiz**: Try to find another valid Breadth-First Search for the same graph we discussed above. (Hint: Start with nodes other than 0).

We are now just left with the programming part, which will be covered super soon in the next lecture. Keep revising the concept until then.

Thank you for being with me throughout the session. I hope you enjoyed it. If you appreciate my work, please let your friends know about this channel too. Lectures on graphs have just started, and if you haven't saved this already, you just have to move on to codewithharry.com or my YouTube channel to access them. See you all there in the next lecture where we'll see the implementation of the Breadth-First Search algorithm in C. Till then keep learning.