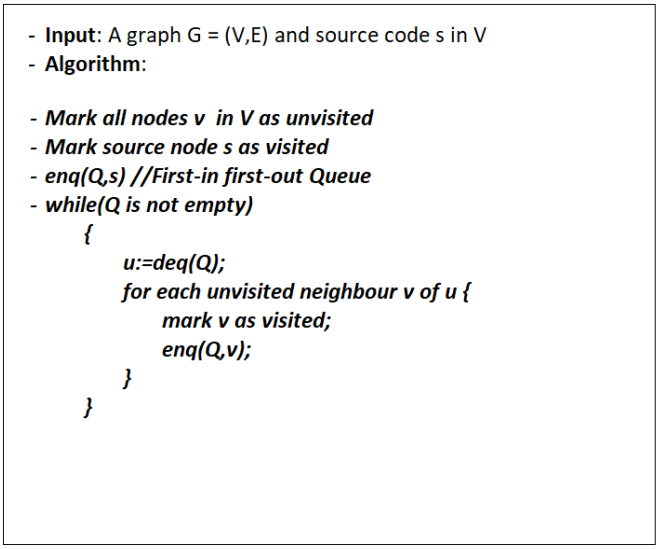
BFS Implementation in C | C Code For Breadth First Search

codewithharry.com/videos/data-structures-and-algorithms-in-hindi-87

In the last lecture, we learned about the concepts of Breadth-First Search in graph traversal. We saw two of its methods, first one using the BFS spanning tree, and the second one being one of the most conventional ones where we maintained a visited array and an exploration queue while we explored new nodes to see what we have visited already and what we have yet to explore respectively. Today we will implement Breadth-First Search in C language.

If you recall, we gave you all a pseudocode in the end and asked you all to read and try implementing the same in C. So, let me know if you could. The pseudocode was:



1. Now, the first thing we did was, we took the input. The input comprises the information concerning the graph, its nodes/vertices, and edges, and the source node we'll start the traversal with.
2. Then, we'll mark the source node *s* visited and then create an exploration queue, and enqueue the source code *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 using the utility function dequeue and suppose that element is *u*. Then, we visit all the vertices which are directly connected to *u* and are already not visited.
4. And every time we visit a new node, we enqueue them in our exploration queue. Eventually, the queue becomes empty, and our traversal ends. And the visited array is the order of our Breadth-First Search traversal.

So, this was the theory part. We are now ready to move on to our programming segment. Having said that, let's move directly to our editors. I have attached the source code below. Follow it as we proceed.

Understanding the source code below:

1. The first thing you should do is, bring the implementation part of the queue from our previous lectures. In this way, we can save time and reduce repetition. I have literally copied everything from my previous snippet including the queue, and its utility functions implemented using arrays.

Initializing a queue:

2. Next thing you should do is initialize a queue, and dynamically assign it a memory in heap using malloc and define its size, say 400. Mark both its front and rear at zeroth index.

```
// Initializing Queue (Array Implementation)
 struct queue q;
 q.size = 400;
 q.f = q.r = 0;
 q.arr = (int*) malloc(q.size*sizeof(int));
```

Code Snippet 1: Initializing a queue

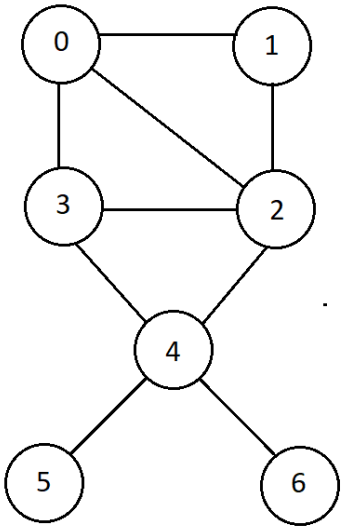
BFS Implementation:

3. Before we start programming the BFS implementation of the graph, we should first define a graph. And out of all the methods we have studied to represent a graph, we will be using the adjacency matrix one to define our graph here.

4. Define an integer variable *node* which we will use to traverse the graph. Define another integer variable *i* which is the node we are currently at. So, since we would be the default starting with any node of our choice, we would initialize it with node 1.

5. Now, define an integer array *visited* to store the status of a node (of size 7 here), and the values corresponding to a node is 0, if it is unvisited and 1, if it is visited. And since, no node is already visited when we first start, therefore we'll fill this array with all zeros.

6. Create an adjacency matrix corresponding to the graph I've illustrated below.



An adjacency matrix holds a value 1 for a cell that is at the intersection of the ith row and jth column if there is an edge between node i and node j. So, the adjacent matrix corresponding to the graph above is:

	0	1	2	3	4	5	6
0	0	1	1	1	0	0	0
1	1	0	1	0	0	0	0
2	1	1	0	1	1	0	0
3	1	0	1	0	1	0	0
4	0	0	1	1	0	1	1
5	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0

7. Next, we would print the source node we have chosen, here node 1. Also, mark this node visited, which means make the ith index of the visited array 1. And since we are now interested in exploring the nodes connected to this source node, we would first enqueue this node into the queue *q* we created in step 2.

8. Create a while loop, and make it run while our queue is not empty. Queue becoming empty is the condition where we finish our traversal. We use the utility function *isEmpty* for the same.

Inside this loop, pluck the top element of the queue using *dequeue* and store it in an integer variable *node*. Run another *for* loop using *j* as the iterator, and since the size of our graph is 7, we'll make this loop run from 0 to 7.

Now, every time we find edges *node & j* connected (that is cell *a[node][j]* equal to 1) and the node *j* unvisited, we mark it visited, and enqueue this newly visited node *j* in the queue. We'll print these nodes while we visit them to determine the BFS traversal order of the graph.

This way we will explore all the nodes, and the queue will ultimately become empty.

```
// BFS Implementation
int node;
int i = 1;
int visited[7] = {0,0,0,0,0,0,0};
int a [7][7] = {
    {0,1,1,1,0,0,0},
    {1,0,1,0,0,0,0},
    {1,1,0,1,1,0,0},
    {1,0,1,0,1,0,0},
    {0,0,1,1,0,1,1},
    {0,0,0,0,1,0,0},
    {0,0,0,0,1,0,0}
};
printf("%d", i);
visited[i] = 1;
enqueue(&q, i); // Enqueue i for exploration
while (!isEmpty(&q))
{
    int node = dequeue(&q);
    for (int j = 0; j < 7; j++)
    {
        if(a[node][j] ==1 && visited[j] == 0){
            printf("%d", j);
            visited[j] = 1;
            enqueue(&q, j);
        }
    }
}
```

Code Snippet 2: Implementing Breadth-First Search

Here is the whole source code:

```c
#include<stdio.h>
#include<stdlib.h>

struct queue
{
    int size;
    int f;
    int r;
    int* arr;
};


int isEmpty(struct queue *q){
    if(q->r==q->f){
        return 1;
    }
    return 0;
}

int isFull(struct queue *q){
    if(q->r==q->size-1){
        return 1;
    }
    return 0;
}

void enqueue(struct queue *q, int val){
    if(isFull(q)){
        printf("This Queue is full\n");
    }
    else{
        q->r++;
        q->arr[q->r] = val;
        // printf("Enqued element: %d\n", val);
    }
}

int dequeue(struct queue *q){
    int a = -1;
    if(isEmpty(q)){
        printf("This Queue is empty\n");
    }
    else{
        q->f++;
        a = q->arr[q->f];
    }
    return a;
}

int main(){
    // Initializing Queue (Array Implementation)
    struct queue q;
    q.size = 400;
    q.f = q.r = 0;
    q.arr = (int*) malloc(q.size*sizeof(int));

    // BFS Implementation
    int node;
    int i = 1;
```

```
    int visited[7] = {0,0,0,0,0,0,0};
    int a [7][7] = {
        {0,1,1,1,0,0,0},
        {1,0,1,0,0,0,0},
        {1,1,0,1,1,0,0},
        {1,0,1,0,1,0,0},
        {0,0,1,1,0,1,1},
        {0,0,0,0,1,0,0},
        {0,0,0,0,1,0,0}
    };
    printf("%d", i);
    visited[i] = 1;
    enqueue(&q, i); // Enqueue i for exploration
    while (!isEmpty(&q))
    {
        int node = dequeue(&q);
        for (int j = 0; j < 7; j++)
        {
            if(a[node][j] ==1 && visited[j] == 0){
                printf("%d", j);
                visited[j] = 1;
                enqueue(&q, j);
            }
        }
    }
    return 0;
}
```

Code Snippet 3: Implementing BFS using queue in C

We'll see if our algorithm actually works, and if it reverts the Breadth-First Search traversal order of the graph we fed into the program. As you could observe, we have started our traversal considering node 0 as the root node. And when we ran the program, the output we received was:

```
0123456
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 1: Output when the root node is 0

But if we change our root node from 0 to 1, the output changes to:

```
1023456
PS D:\MyData\Business\code playground\Ds & Algo with Notes\Code>
```

Figure 2: Output when root node is 1

I encourage you all to try this yourself. It shall work for any node you start with, and any graph you feed into the program. I hope I made this clear for you all. You can go through the lectures again for a better understanding of things. Depth First Search is what we'll cover next.

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 explore the next traversal method, the Depth First Search Till then keep learning.