**Project Report : Breadth First Search – graph theory**

**Author: Nikhil Bhirud, Sachin Badgujar, Bhsuhan Suhas Deshpande**

**-----------------------------------------------------------------------------------------------------------------------------------------------------------**

**How does your demonstration algorithm improve the understanding of the algorithm?**

Following points make our demonstration better:

1. Our demonstration includes step by step procedure to execute the algorithm.
2. Every iteration and recursion is explained with the visual help of graph structure and array structure.
3. We have made sure to mention the line in the code which is related to the results and the values of variables which outputs the results represented visually.
4. Also the code implementation is modular with proper names to functions and variables to clearly understand the purpose of statements.
5. In addition, we have added some pre-requisite information about adjacency matrix and how it is represented in an array.

**What is Breadth first search?**

It is a level by level search where one entire level of a tree is searched before searching the next one.

7

15

26

77

24

e.g BFS of adjacent tree with 7 as starting node would be

7, 15, 26, 77, 24.

In case of graphs, all the Neighboring nodes of a node are visited first then neighbors of neighbors and so on.

e.g. BFS of this graph with starting node as 7 would be,

,15 ,22 ,3 ,55 ,

7

9

.

7

15

3

22

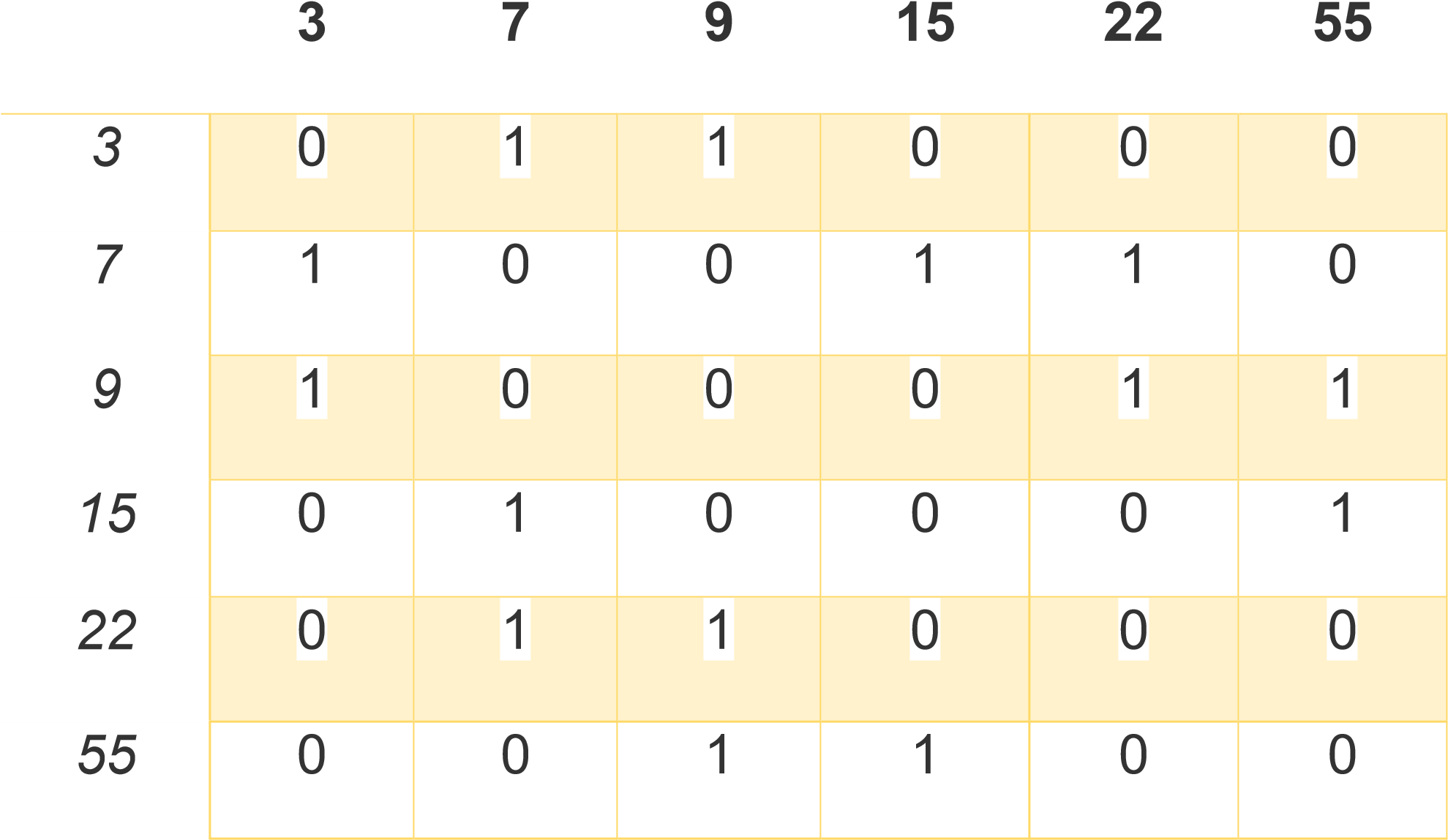
55

9

**What is adjacency Matrix?**

If there r ‘n’ nodes then adjacency matrix is the is a n by n matrix where intersection with some value other than ‘0’ means an edge between those vertices.

e.g for the above shown graph , the matrix would be as follows . We are assuming weight on each edge is 1, since is not mentioned.



**BFS Implementation:**

//This class implements the BFS Algorithm for undirected graphs import java.util.Arrays; import java.util.Scanner;

public class BFS {

public static void main(String args[]){ int [][] adjacencyMatrix = null;

//Queue is a user defined data structure

MyQueue queue = new MyQueue();

Scanner scan = new Scanner(System.in);

//Taking input from console.

System.out.println("Enter the number of nodes: "); int numberOfNodes = scan.nextInt();

adjacencyMatrix = new int[numberOfNodes+1][numberOfNodes+1];

//Accepting the adjacency matrix

System.out.println("Enter the adjecency matrix"); for(int i=1;i<numberOfNodes+1;i++) for(int j=1;j<numberOfNodes+1;j++) adjacencyMatrix[i][j] = scan.nextInt();

System.out.println("Please enter the source node: "); int source = scan.nextInt(); scan.close();

//Logic for BFS

int visited[] = new int[numberOfNodes+1];

int BFSTree[] = new int[numberOfNodes]; ……………………….. (1) queue.add(source);

visited[source] = 1;

BFSTree[0] = source; int x = 1;

while(!queue.isEmpty()){ ………………………….(2) int i = queue.remove(); for(int j=1;j<=numberOfNodes; j++){

if (adjacencyMatrix[i][j] == 1 && visited[j] != 1){ ………………….(3)

visited[j] = 1; BFSTree[x++] = j; queue.add(j);

if(x== numberOfNodes){ // It means all nodes are visited ………..(4) queue= new MyQueue(); // empty queue.

}

}

}

}

//Printing the BFS result if the graph is complete if(Arrays.toString(BFSTree).indexOf('0') == -1){

System.out.println("\nBFS: "+Arrays.toString(BFSTree));

}

else{

System.out.println("\nThe graph entered is not connected");

System.out.println("Node "+ BFSTree[Arrays.toString(BFSTree).indexOf('0')-1]+" is disconnected");

}

}

}

//Class Link is used as a base structure for Queue

// \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// | | | | | |

// | data | next |-------->| data | next |

// |\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_\_\_|\_\_\_\_\_\_\_\_\_\_\_| class Link {

int data;

Link next;

public Link(int data) {

this.next = null; this.data = data;

}

}

//Class MyQueue is an implementation of queue class MyQueue { private Link last; private Link first; public MyQueue() {

first = null; last = null;

}

boolean isEmpty(){ return first==null;

}

void add(int data){ Link element = new Link(data); if (last != null) last.next = element; element.next = null; last = element; if (first == null)

first = element;

}

int remove(){ if (first==null){

return -1;

}

int data = first.data; first = first.next;

return data;

}

}

Let’s take an example to completely understand the BFS implementation.

5

2

4

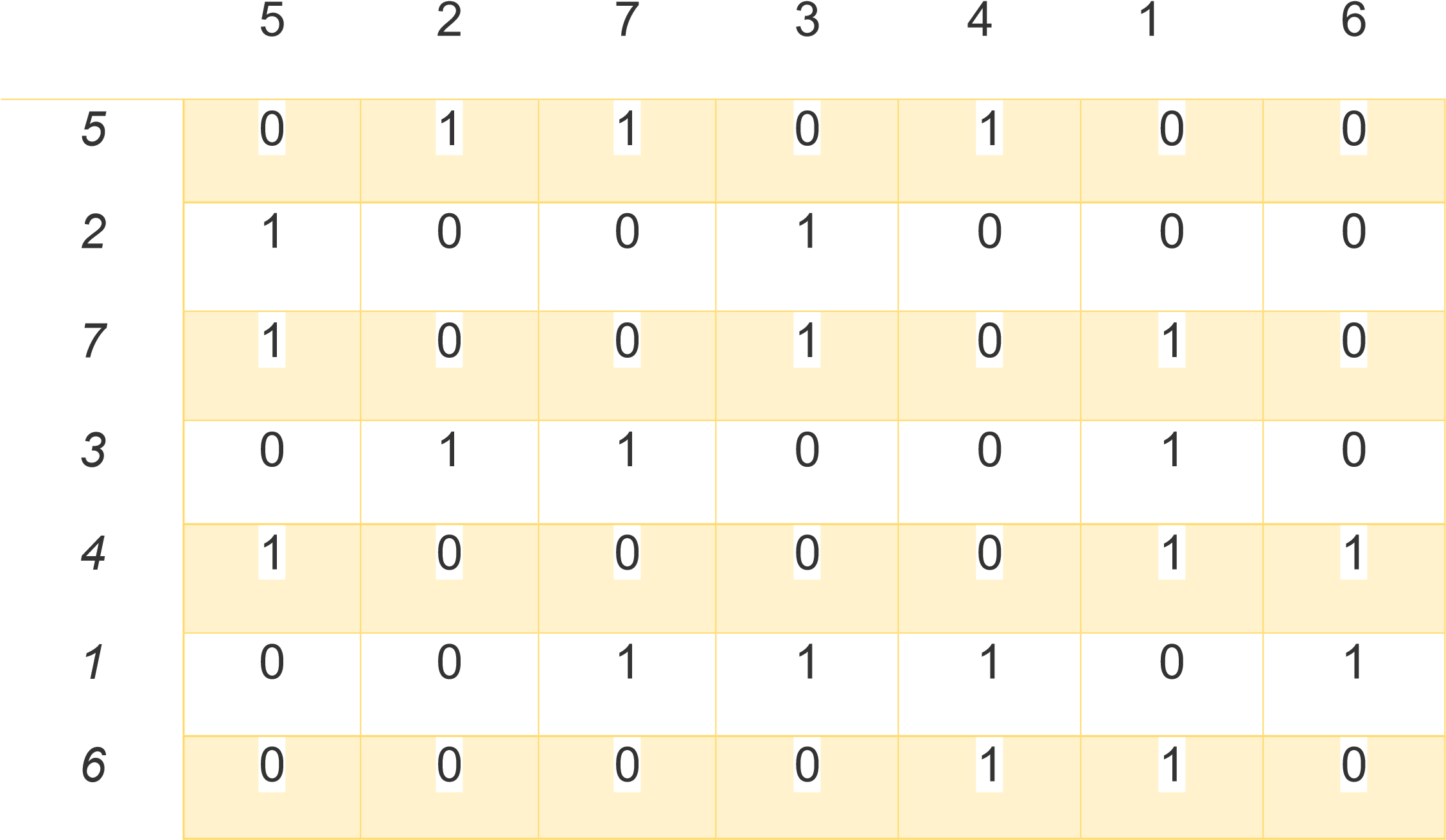
7

3

1

6

Adjaceny matrix for it would be like this



There are two arrays and a queue as follows 1) Initially: - refer to (1) in the code.

Let the source node be (5)

Source node

5

2

4

7

3

1

6

BFS Tree (result Array)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **5** |  |  |  |  |  |  |

Visited Array: (This makes sure that we are not revisiting a vertices) (indexed from 1-10)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  | **1** |  |  |

Queue: (maintains the vertices to be visited next)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **5** |  |  |  |  |  |  |

Remove this from queue and find its neighbors.

Iteration starts (refer (2) from implementations)

Iteration 1:

It removes 5 from the queue. Run loop over all the nodes to find if there is a edge using adjacency matrix.

If edge is found , add that node to BFS tree and mark it as visited.

2,4,7 are added to queue.

Refer implementation (3)

Neig

hbor

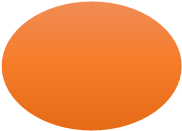
Neig

hbor

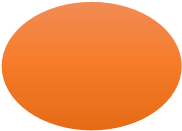
Neig

hbor

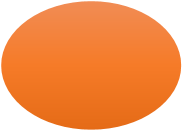
5



2



4



7

3

1

6

BFS Tree (result Array)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **5** | **2** | **4** | **7** |  |  |  |

Visited Array: (This makes sure that we are not revisiting a vertices) (indexed from 1-10)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** |  | **1** | **1** |  | **1** |

Queue: (maintains the vertices to be visited next)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **2** | **4** | **7** |  |  |  |  |

Remove this from queue and find its neighbors.

Iteration 2:

It removes 2 from the queue. Run loop over all the nodes to find if there is an edge using adjacency matrix.

If edge is found, add that node to BFS tree and mark it as visited.

Node 3 is added.

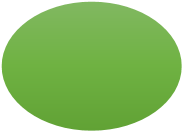
Refer implementation (3)

Completed tree is in green.

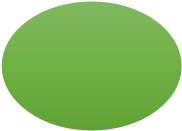
Neig

hbor

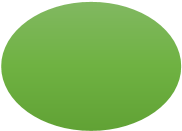
5



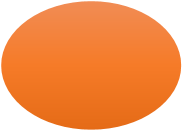
2



4



7



3

1

6

BFS Tree (result Array)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **5** | **2** | **4** | **7** | **3** |  |  |

Visited Array: (This makes sure that we are not revisiting a vertices) (indexed from 1-10)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **1** | **1** | **1** |  | **1** |

Queue: (maintains the vertices to be visited next)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **4** | **7** | **3** |  |  |  |  |

Remove this from queue and find its neighbors.

Iteration 3:

It removes 4 from the queue. Run loop over all the nodes to find if there is an edge using adjacency matrix.

If edge is found, add that node to BFS tree and mark it as visited.

Nodes 1,6 are added.

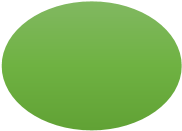
Refer implementation (3)

Completed tree is in green.

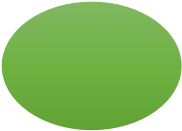
Neighbor

Neighbor

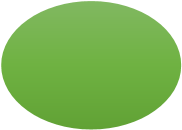
5



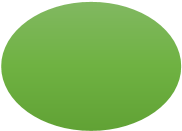
2



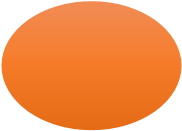
4



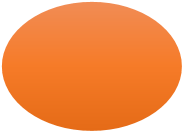
7



3



1



6

BFS Tree (result Array)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **5** | **2** | **4** | **7** | **3** | **1** | **6** |

Visited Array: (This makes sure that we are not revisiting a vertices) (indexed from 1-10)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **1** | **1** | **1** | **1** | **1** | **1** | **1** |

All Vertices are visited now.

Queue: (maintains the vertices to be visited next)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **7** | **3** | **1** | **6** |  |  |

Now implementation reaches the point shown by (4).

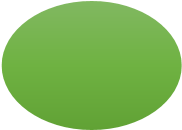
It renews the queue and hence emptying it and the while loop ends.

# Final BFS Sequence

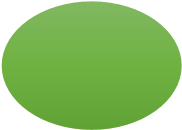
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **5** | **2** | **4** | **7** | **3** | **1** | **6** |

Completed tree is in green.

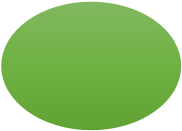
5



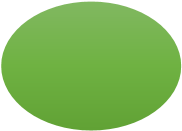
2



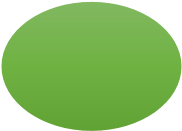
4



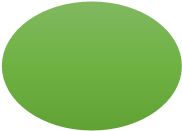
7



3



1



6