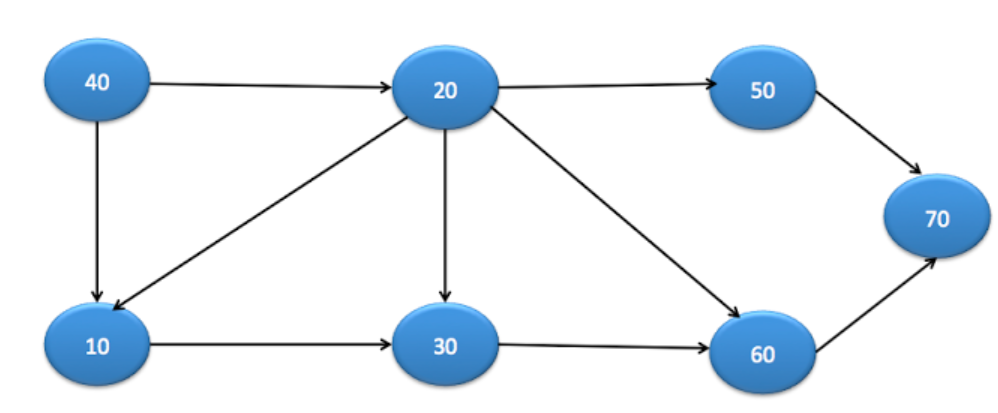
**Algorithm Implementation – Graph Traversals:**

**Part 1 – Implementing a Graph in Java:**

A graph can be respresented by a list of points and an adjacency matrix. The list of points describes what nodes are in the graph and the adjacency matrix describes how they are connected. Consider the following graph.



This graph can be implemented by initializing the list of points and adjacency matrix as shown below.

|  |
| --- |
| **public** **static** **void** main(String arg[]){  List<Node> nodes = **new** ArrayList<Node>();  nodes.add(**new** Node(40)); //This node gets index 0  nodes.add(**new** Node(10)); //This node gets index 1  nodes.add(**new** Node(20)); //This node gets index 2  nodes.add(**new** Node(30)); //This node gets index 3  nodes.add(**new** Node(60)); //This node gets index 4  nodes.add(**new** Node(50)); //This node gets index 5  nodes.add(**new** Node(70)); //This node gets index 6    **int** adjacencyMatrix[][] = {  {0,1,1,0,0,0,0}, // Adjacency list for node at index 0 (40)  {0,0,0,1,0,0,0}, // Adjacency list for node at index 1 (10)  {0,1,0,1,1,1,0}, // Adjacency list for node at index 2 (20)  {0,0,0,0,1,0,0}, // Adjacency list for node at index 3 (30)  {0,0,0,0,0,0,1}, // Adjacency list for node at index 4 (60)  {0,0,0,0,0,0,1}, // Adjacency list for node at index 5 (50)  {0,0,0,0,0,0,0}, // Adjacency list for node at index 6 (70)  };    Graph graph = **new** Graph(nodes, adjacencyMatrix);  } |

* To get the adjacency list of any node, call adjacencyMatrix[nodes.indexOf(nodeOfInterest)]
* The adjacency matrix of node n specifies which nodes n can traverse to. For example, the only node 10 can traverse to is node 30. Node 10 has the adjacency list: {0,0,0,1,0,0,0}. This means that the only node 10 can traverse to in the graph is at index 3 (since index 3 is the only position with a one in the adjacency list). The element at index 3 is node 30. This is how you would interpret an adjacency list in Java.
* A neighbour of a node ‘n’ is a node that can be traversed to b n.

**Part 2 – Graph Class (Discluding Traversals) and Node Class:**

|  |
| --- |
| **public** **class** Graph{  **public** List<Node> nodes;  **public** **int**[][] adjacencyMatrix;  **public** Graph(List<Node> nodes, **int**[][] adjacencyMatrix){  **this**.nodes = nodes;  **this**.adjacencyMatrix = adjacencyMatrix;  }  **public** **boolean** isNeighbour(**int** currentNodeIndex, **int** otherNodeIndex){  **return** adjacencyMatrix[currentNodeIndex][otherNodeIndex] == 1;  }  **public** **boolean** eligibleToVisit(Node neighbour){  **return** (neighbour != **null** && !neighbour.visited);  }  **public** List<Node> visitEligibleNeighbours(Node node){  ArrayList<Node> visitedNodes = **new** ArrayList<Node>();  **int** nodeIndex = nodes.indexOf(node);  **for** (**int** i = 0; i < nodes.size(); i++) {  **if**(isNeighbour(nodeIndex, i)){  Node neighbour = nodes.get(i);  **if**(eligibleToVisit(neighbour)){  neighbour.visited = **true**;  visitedNodes.add(neighbour);  }  }  }  **return** visitedNodes;  }  } |

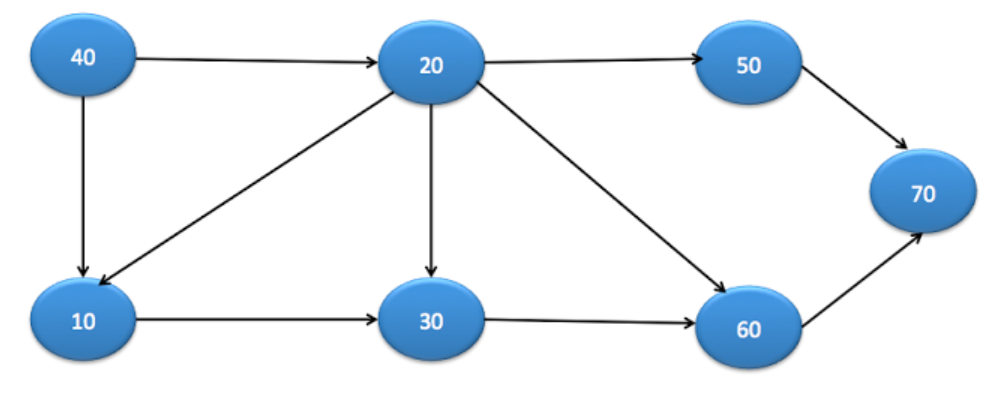
* The method “eligible to visit” determines if the neighbour node in the paramter is eligible to visit. For it to be eligible to visit, it must not be null and not be already visited.
* The “visit elligible neighbours” method visits all the neighbours of the paramter node that are eligible to visit. This method also returns a list of the nodes that we have visited. So to summarize, this method takes in a node, visits all its elligible neighbours, and returns them in a list.

**Part 3 – Breadth First Search:**

|  |
| --- |
| **public** **void** bfs(Node startNode){  Queue<Node> queue = **new** LinkedList<Node>();  startNode.visited = **true**;  queue.add(startNode);    **while** (!queue.isEmpty()){  Node node = queue.remove();  System.*out*.print(node.data + " ");  List<Node> visitedNeighbours = visitEligibleNeighbours(node);  queue.addAll(visitedNeighbours);  }  } |

* First you take the start node, visit it, and add it to the queue. You go into the while loop with one node in the queue (the start node) which is visited).
* While the queue is not empty, you remove an item from the queue, print it. Then you visit all its elligible neighbours. Then you add those elligible neighbours to the queue.
* What happens is, when you print an item, that item is completely gone from the queue. However, before you lose reference to that item, you add all its elligible neighbours to the queue. Eventually then you must handle each one of those elligible neighbours one by one.
* Notice how you always visit nodes before you add them to the queue. This is because once you add them to the queue, they will guaranteed get printed (when the get to the front of the queue). If you add items that aren’t visited in the queue, they will get added again (because someone else will consider them eligible). This will result in duplicate items in the queue leading to duplicate prints when they reach the front.
* The program terminates once everything has been visited once (which results in no eligible neighbours and then an empty queue).

**Part 5 – Walk Through Example (Start Node is 40):**

****

|  |
| --- |
| **Initial:** We visit the start node which is 40. We add the start node the queue. We then enter the while loop.  **Iteration 1:**  We remove an item (the 40) from the queue. We print it. The console is now [40 ].  We identify and visit all the eligible neighbours of (the 40) which are {10,20}  After we visit and add all the eligible neighbours of (the 40) the queue is: {20 🡪 10}  **Iteration 2:**  We remove an item (the 10) from the queue. We print it. The console is now [40 10 ].  We identify and visit all the eligible neighbours of (the 10) which are {30}  After we visit and add all the eligible neighbours of (the 10) the queue is: {30 🡪 20}  **Iteration 3:**  We remove an item (the 20) from the queue. We print it. The console is now [40 10 20 ].  We identify and visit all the eligible neighbours of (the 20) which are {60,50}  After we visit and add all the eligible neighbours of (the 20) the queue is: {50 🡪 60 🡪 30}    **Iteration 4:**  We remove an item (the 30) from the queue. We print it. The console is now [40 10 20 30 ].  We identify and visit all the eligible neighbours of (the 30) which are {}  After we visit and add all the eligible neighbours of (the 30) the queue is: {50 🡪 60}  **Iteration 5:**  We remove an item (the 60) from the queue. We print it. The console is now [40 10 20 30 60 ].  We identify and visit all the eligible neighbours of (the 60) which are {70}  After we visit and add all the eligible neighbours of (the 60) the queue is: 70 🡪 50}  **Iteration 6:**  We remove an item (the 50) from the queue. We print it. The console is now [40 10 20 30 60 50 ].  We identify and visit all the eligible neighbours of (the 50) which are {}  After we visit and add all the eligible neighbours of (the 50) the queue is: {70}  **Iteration 7:**  We remove an item (the 70) from the queue. We print it. The console is now [40 10 20 30 60 50 70 ].  We identify and visit all the eligible neighbours of (the 70) which are {}  After we visit and add all the eligible neighbours of (the 70) the queue is: {}  **Final**: 40 10 20 30 60 50 70 |

**Important Note:** A key component of the algorithm how do you choose the order of eligible neighbours.The order in which you add all the items in the queue affects the order of the final print. For example, we know after the start node 40, we must go to the eligible neighbour nodes (the 10 and the 20). Do you go to the 10 first? Or the 20 first? If you choose 10 and then 20, then this will result in 10’s elligible neighbours being added to the queue first, and then 20’s eligible neighbours being added to the queue which will continue to affect the orde of the printing. This will result in 10’s elligible neighbours (the 30) being printed before 20’s elligible nighbours (the 50, 60 and coincidentally includes the 30).

In this implementation, the way to decide the order of the eligible neighbours is by the order of the elements in the node list. Since 10 comes in the node list before 20, it will be chosen first. This happens because we do

|  |
| --- |
| **for** (**int** i = 0; i < nodes.size(); i++) {  **if**(isNeighbour(nodeIndex, i)){  Node neighbour = nodes.get(i); |

So as we iterate over the node list to determine which ones are eligible neighbours, we will fill up the a list and return that list. This list will then be added to the queue and that is the order of the prints.

**Part 6 – Time and Space Complexity of Breadth First Search:**