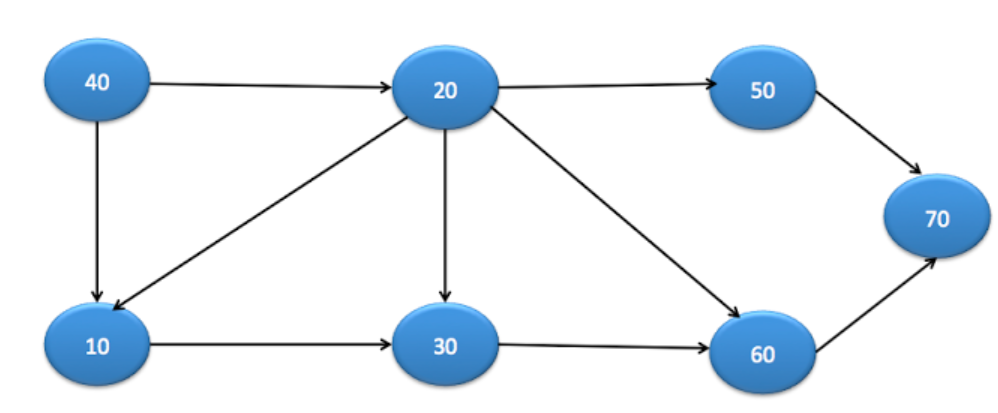
**Algorithm Implementation – Breadth First Search for Graphs:**

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

A graph can be respresented by a list of nodes. Each node contains a list of nodes adjacent to it (i.e nodes it can traverse to in one step).



|  |
| --- |
| **public** **class** Node{  **public** **int** data;  **public** **boolean** visited;  **private** List<Node> adjacentNodes;    **public** Node(**int** data){  **this**.data=data;  adjacentNodes = **new** ArrayList<Node>();  }    **public** **void** addAdjacentNode(Node node){  adjacentNodes.add(node);  }  } |

So if you have a list of nodes that contains the values {10, 20, 30, 40, 50, 60, 70} and each node contains a list of its adjacent nodes, then you can specifiy the entire graph. For example, node 40 has a list of adjacent nodes that contain {10, 20} which means there are two edges that can allow the graph to traverse from 40 to either 10 or 20.

**Part 2 – Using Adjacency Matrices to Specify Edges:** You can use an adjacency matrix to specifiy all the edges in the graph in a concise fashion.

|  |
| --- |
| **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)  }; |

* 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 adjacency matrix of node 10 is: {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.
* Adjacency Matrices make it easier and quicker to build a graph with a lot of edges since this will only make more ones in the double array as opposed to more lines of code.

**Part 3 – Setting Adjacent Nodes:** Once you have your adjacency matrix filled out, you can pass it as a parameter to the method “set adjacent nodes”. This method will actually ensure each node has its adjacent nodes in its list of adjacent nodes.

|  |
| --- |
| **public** **static** **void** main(String arg[]){  List<Node> nodes = **new** ArrayList<Node>();  Node node40 = **new** Node(40);  Node node10 = **new** Node(10);  Node node20 = **new** Node(20);  Node node30 = **new** Node(30);  Node node60 = **new** Node(60);  Node node50 = **new** Node(50);  Node node70 = **new** Node(70);    nodes.add(node40);  nodes.add(node10);  nodes.add(node20);  nodes.add(node30);  nodes.add(node60);  nodes.add(node50);  nodes.add(node70);    **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)  };    *setAdjacentNodes*(nodes, adjacencyMatrix);  Graph graph = **new** Graph(nodes);  graph.bfs(node40);  }    **public** **static** **void** setAdjacentNodes(List<Node> nodes, **int**[][] adjacencyMatrix){  **for**(**int** i = 0; i < nodes.size(); i++){//For each node in the graph  **for**(**int** j = 0; j < adjacencyMatrix[i].length; j++){//Go through its adjaceny list  **if**(adjacencyMatrix[i][j] == 1){ //If you find a one in its list  nodes.get(i).addAdjacentNode(nodes.get(j)); //add an adjacent node  }  }  }  } |

* After calling “set adjacent nodes” you now have a list of nodes that each have their list of adjacent nodes correctly initialized. The “adjacency matrix” was just a tool used to neatly set the adjacent nodes and is no longer need. The graph is only specified by its list of nodes (its only parameter).
* Once you call “set adjacent nodes”, you can initialize your graph and call breadth first search.

**Part 4 – Graph Class (Discluding Traversals) and Node Class:**

|  |
| --- |
| **public** **class** Node{  **public** **int** data;  **public** **boolean** visited;  **private** List<Node> adjacentNodes;    **public** Node(**int** data){  **this**.data=data;  adjacentNodes = **new** ArrayList<Node>();  }    **public** **void** addAdjacentNode(Node node){  adjacentNodes.add(node);  }  }  **public** **class** Graph{  **public** List<Node> nodes;  **public** Graph(List<Node> nodes){  **this**.nodes = nodes;  }  **public** **boolean** eligibleToVisit(Node adjacentNode){  **return** (adjacentNode != **null** && ! adjacentNode.visited);  }    **public** List<Node> visitEligibleAdjacentNodes(Node node){  ArrayList<Node> visitedNodes = **new** ArrayList<Node>();  **for** (Node adjacentNode : node.adjacentNodes) {  **if**(eligibleToVisit(adjacentNode)){  adjacentNode.visited = **true**;  visitedNodes.add(adjacentNode);  }  }  **return** visitedNodes;  }  } |

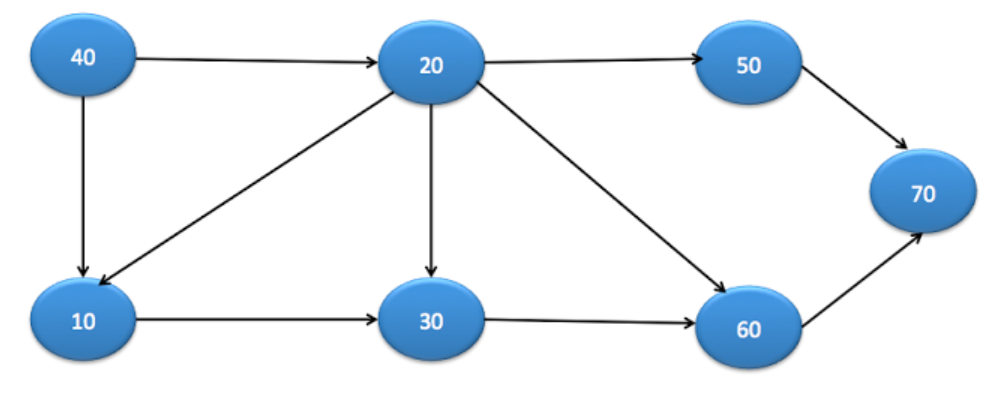
* An adjacent node is eligible to visit if it is not null and if it has not been visited.
* The method “visit eligible adjacent nodes” visits all the nodes that are adjacent to the node in the parameter and eligible to visit. It returns a list of all the nodes that were visited.

**Part 5 – 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> visitedNodes = visitEligibleAdjacentNodes(node);  queue.addAll(visitedNodes);  }  } |

* 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 adjacent nodes. Then you add those elligible adjacent nodes 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 adjacent nodes to the queue. Eventually then you must handle each one of those elligible adjacent nodes 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 adjacent nodes and then an empty queue).

**Part 6 – 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 adjacent nodes of (the 40) which are {10,20}  After we visit and add all the eligible adjacent nodes 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 adjacent nodes of (the 10) which are {30}  After we visit and add all the eligible adjacent nodes 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 adjacent nodes of (the 20) which are {60,50}  After we visit and add all the eligible adjacent nodes 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 adjacent nodes of (the 30) which are {}  After we visit and add all the eligible adjacent nodes 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 adjacent nodes of (the 60) which are {70}  After we visit and add all the eligible adjacent nodes 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 adjacent nodes of (the 50) which are {}  After we visit and add all the eligible adjacent nodes 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 adjacent nodes of (the 70) which are {}  After we visit and add all the eligible adjacent nodes 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 adjacent nodes. 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 adjacent 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 eligible adjacent nodes being added to the queue first, and then 20’s eligible adjacent nodes being added to the queue which will continue to affect the order of the printing. This will result in 10’s elligible adjacent nodes (the 30) being printed before 20’s eligible adjacent nodes (the 50, 60 and coincidentally includes the 30).

In this implementation, the way to decide the order of the eligible adjacent nodes is by the order of the elements in the original node list in the graph. Since 10 comes in the node list before 20, it will be chosen first.

To illustrate how this happens, we first add the nodes in the node list (in main) below.

nodes.add(**new** Node(10)); //This node gets index 1

nodes.add(**new** Node(20)); //This node gets index 2

Since we aribtrarily added 10 before 20 in the node list, 10 had a lower index then 20. We later iterate around this adjacency matrix from left to right using

**for**(**int** j = 0; j < adjacencyMatrix[i].length; j++){//Go through its adjaceny list

**if**(adjacencyMatrix[i][j] == 1){ //If you find a one in the list

nodes.get(i).addAdjacentNode(nodes.get(j)); //add an adjacent node

}

}

With ‘i’ at this point set to 0. This means we add the 10 and the 20 into the adjacency nodes using ‘nodes.get(j)’. Remember, whoever got added first to the list ‘nodes, has the lower list index, which means it gets added to the adjacency list first.

Later, we iterate around the adjacency nodes list and if they are eligible to visit, we add them to a visited nodes list. This list gets added to the queue and this is the order of the print. So to summarize,

1. The order of the 10,20 in the original list specifies the order in which they will be added to the adjacent nodes list of 40.
2. The order of the adjacent nodes list of 40 specifies the order of the visited nodes list in “visit eligible nodes”.
3. The order of the “visitied nodes” list specifies the order in which they get added to the queue.
4. The order in which they get added to the queue specify the order in which they get printed.

**Part 6 – Time and Space Complexity:**

**Time Complexity:**

* The while loop in the “breadth first search” method gets executed V times where V is the number of nodes/vertices in the graph. This is because each node enters the queue once, removed once, and printed once. Since we see that there is a print in every iteration, we know there are V iterations.
* To add to the last point, the boolean variable “visited nodes” ensures that an item only gets added to the queue once. Since there is one item being removed every iteration, and V items that pass through the queue, then it will take V iterations to empty the queue. We keep iterating until the queue is empty so we iterate V times. To further prove this, notice how there are 7 iterations and 7 nodes in the example above.
* The code inside the while loop in the “breadth first search” is constant except the “visit eligible adjacent nodes” call. So we can use the following mathematical representation to represent the time complexity.
* We can split the sums like how did below.
* The first sum simplifies to C×V since we are adding C a total of V times. This simplifies the equation to the one below.
* Now let’s analyze the second sum. We are thinking about the time complexity of calling “visit adjacent nodes” for each node/vertice in the graph. This happens because each node gets removed from the queue once, and its that same node that we use to call “visit adjacent nodes”. So imagine calling the function below with every possible node in the graph.

|  |
| --- |
| **public** List<Node> visitEligibleAdjacentNodes(Node node){  ArrayList<Node> visitedNodes = **new** ArrayList<Node>();  **for** (Node adjacentNode : node.adjacentNodes) {  **if**(eligibleToVisit(adjacentNode)){  adjacentNode.visited = **true**;  visitedNodes.add(adjacentNode);  }  }  **return** visitedNodes;  } |

* The complexity of calling this function once is O(number Of Adjacent Nodes). So the complexity of calling this function for every node in the graph is represented below.
* If you think about it, the sum of adjacent nodes for all nodes is equal to the number of edges for a directed graph. This is because if there is an edge between to nodes (i.e. there is an edge from 40 to 10), this increases the number of adjacent nodes othat 40 has by one.
* So the complexity of breadth first searach Is O(V + E).
* In addition, if V > E, then (V + E ) < ( V + V ) = ( 2V) which shows that the complexity would be O(V). The same is also true if E > V, the complexity would be O(E). Thus the final time complexity is O(V + E) or O(max(E,V)).

**Part 7 – Different Implementations:**

Be aware that there are different implementations for graphs. What you used above is refered to as “Graphs with Adjacency Lists”. However, you could have an implementation for graphs with only adjacency matrix. What this means is that you have a list of nodes (that don’t have attributes for adjacent nodes, and only have data attributes) and then your graph class has an adjacency list. So when you try to visit all eligible adjacent nodes, you would have to iterate over all the nodes (in the list of nodes) and check if they are A) eligible and B) neighbours. This would cost O(V \* V). Just be aware of how the graphs are designed before you are quick to assign the time complexity.