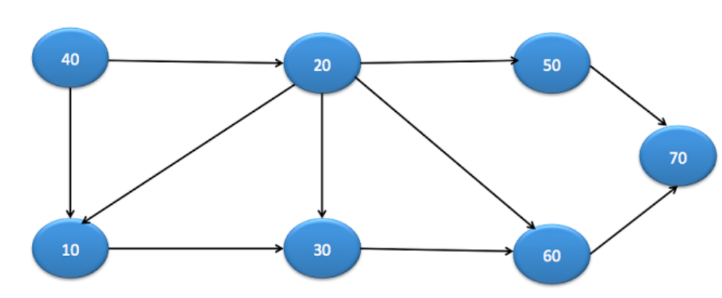
**Algorithm Implementation – Depth First Search for Graphs:**

**Part 1 – Implementation:**

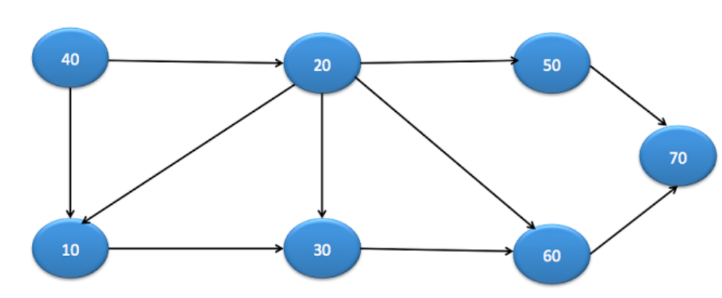
* Review Breadth First Search to know how to implement and initialize graphs in Java.



|  |
| --- |
| **public** **void** dfsUsingStack(Node startNode){  Stack<Node> stack = **new** Stack<Node>();  stack.add(startNode);  startNode.visited = **true**;  **while** (!stack.isEmpty()){  Node node = stack.pop();  System.*out*.print(node.data + " ");  List<Node> visitedNodes = visitEligibleAdjacentNodes(node);  stack.addAll(visitedNodes);  }  } |

* The idea with breadth first search is you start at the node 40, and get all the eligible adjacent nodes (after you mark them visited). This will return you a list of the eligible adjacent nodes (10 and 20) in the order of which they were added in the graph (this is discussed in the breadth first search document). However, since we are using a stack, we pop the item 20 (since it is last in first out) and that is the node we print next. We continue this process.
* What this process does is goes as deep as possible. The 40 has two eligible adjacent nodes (10 and 20) and we go to the latter (since it is LIFO). Then the 20 has 3 eligible adjacent nodes (and we go to the 50). And then the 50 has one eligible adjacent node (the 70) and we traverse to that node.
* Breadth first search first goes to a start node, traverses to the start nodes neighbours, then the start nodes neighbour’s neighbours, and so on. It traverses a whole neighbourhood and keeps increasing the radius of this neighbourhood until it covers the entire graph. Depth first search goes door to door (i.e. start node -> arbitrary neighbour -> arbitrary neighbour of previous neighbour ) until it reaches a dead end in which it begins to go back again.
* Suppose the start node has two eligible adjacent nodes A and B. Then suppose we visit A first and also add A’s eligible adjacent nodes (Aneighbour). Depth first search prioritizes Aneighbour over B due to the stack LIFO property, while Breadth First search prioritizes node B over Aneighbour due to the queue FIFO property. When you consistently choose the first eligible adjacent node’s neighbour over the second eligible adjacent node, you are going to go as deep as possible.
* Both these search algorithms are blind searches. If you are looking for a particular node, it is a matter of luch of where and when you will find it.

**Part 2 – Execution Example:**



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

**Part 3 – Time Complexity:**

Time Complexity is the same as breadth first search for the same reasons.