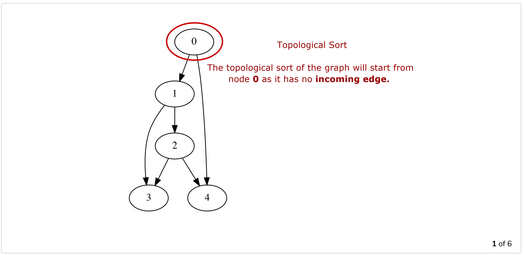
What is Topological Sort?

Topological Sort is a way to order a **directed acyclic** graph. A **directed** graph has edges that are *incoming* or *outgoing*, meaning that they have a specific direction. An **acyclic** graph has *no cycles*, i.e., a node is not reachable from its ancestors. A topological sort takes a graph and finds the order of its nodes so it **always** starts from a node that has no incoming edges and then traverses the adjacent nodes. Note that the current node is before its adjacent. The illustration below will help explain this concept better.



Implementing the Code

The code below illustrates how to implement this process using recursion. First, let's examine the code and then move on to its explanation.

Modify the edges by using the `addEdge` function and the number of vertices, `nVertices`, to create your own graph, `g`.Then, run the `topologicalSorting` on these variables to see how it works.

```
class TopologicalSortClass {

    static class Graph {
        int numVertices;
        LinkedList<Integer>[] tempList;

        Graph(int numVertices) {
            this.numVertices = numVertices;
            tempList = new LinkedList[numVertices];
            for (int i = 0; i < numVertices ; i++) {
                tempList[i] = new LinkedList<>();
            }
        }

        // Method to add an edge between 2 nodes in the Graph
        // fromNode 2 toNode 4 ==> 2 -> 4
        public void addEgde(int fromNode, int toNode) {
            tempList[fromNode].addFirst(toNode);
        }

        public void topologicalSorting() {
            boolean[] visited = new boolean[numVertices];
            Stack<Integer> ts = new Stack<>();

            //visit from each node if not already visited
            for (int i = 0; i < numVertices; i++) {
                if (!visited[i]) {
                    topologicalSortRecursive(i, visited, ts);
                }
            }
            System.out.println("Topological Sort: ");
            int size = ts.size();
            for (int i = 0; i < size; i++) {
                System.out.print(ts.pop() + " ");
            }
        }

        public void topologicalSortRecursive(int start, boolean[] visited, Stack<Integer> ts) {
            visited[start] = true;
            for (int i = 0; i < tempList[start].size(); i++) {
                int vertex = tempList[start].get(i);
                if (!visited[vertex])
                    topologicalSortRecursive(vertex, visited, ts);
            }
            ts.push(start);
        }

    }

    public static void main( String args[] ) {
        System.out.println( "Path after Topological Sorting: " );

        int nVertices = 5;

        Graph g = new Graph(nVertices);

        g.addEgde(0, 1);
        g.addEgde(0, 4);
        g.addEgde(1, 2);
        g.addEgde(1, 3);
        g.addEgde(2, 3);
        g.addEgde(2, 4);

        // Topological function called here
        g.topologicalSorting();

    }
}
```

Run | Save | Reset

Understanding the Code

The recursive code can be broken down into two parts. The first is the recursive method and the second is the main where the method is called. Before examine the actual `topologicalSortRecursive` method let's quickly go over some of the details from the class `Graph`.

- The class `Graph` on **line 4 and 5** has two variables: `numVertices` which depicts the total number of vertices in the graph, and a list that contains the elements of type **Integer**, called `tempList`. The *addEdge* method from **line 17 to 19** takes in two nodes as its arguments and creates an edge between them. This class also uses the `topologicalSorting` and `topologicalSortRecursive` methods, which will run the Topological Sorting on the graph.

Let's examine the two divisions of code.

Driver Method

First, let's examine the driver method, which calls the recursive code.

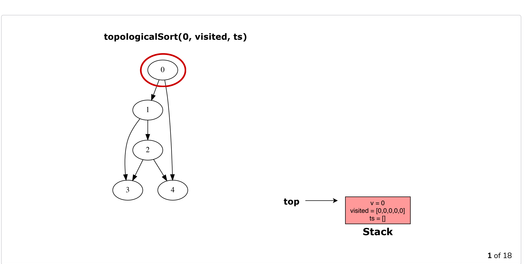
- In the `main` method, on **line 53**, the `nVertices` variable is initialized .This serves to depicts the number of vertices in the graph.
- On **line 55**, a new Graph is created with the number of vertices, `nVertices`, as its argument.
- From **lines 57 to line 62**, multiple edges are created, and each edge takes 2 integers, `fromNode` 2 to a `toNode` 4 to create an edge that points from **Node 2** to **Node 4**.
- On **line 65**, the `topologicalSorting()` function is called.

Recursive Method

Now that we know how the recursive code is called, let's look at the `topologicalSorting` and the `topologicalSortRecursive` in detail. This is the code segment between **lines 21 and 46** in the code snippet above.

- The `topologicalSorting()` first creates a type `boolean` array, `visited`, to mark the vertices that have been visited. and also creates a new Stack,called `stack`.
- The for-loop from **line 26 to line 30** continues until all the vertices are covered . It then marks ones that have not been visited by calling the main `topologicalSortRecursive` method.
- The `topologicalSortRecursive` method takes in three arguments. The first is vertex `start` which needs to be traversed as the first argument. The second is `visited` as the second one, which stores whether a node has been traversed or not. The third argument is the `stack`.
- This method takes the vertex `start` and makes a recursive call to its unvisited neighboring vertices. It chooses from one of the children nodes and then moves down and mark the children node of that vertex. If it finds that a certain vertex has no neighbors or unvisited neighbors, it pushes this vertex into the stack and traverses back to the last vertex. This process checks if the vertex it has other unvisited children nodes. The same cycle then continues once all the nodes have been checked.
- The for-loop from **line 31 to line 35** prints the vertices that were pushed in the stack during the `topologicalSortRecursive`. This enables us to see the directed order of nodes traversed using topological sort.

Understanding through a Stack



Now that we have learned how to carry out a Topological Sort on a graph using recursion in Java, the next lesson offers a short quiz on all the concepts of this chapter!