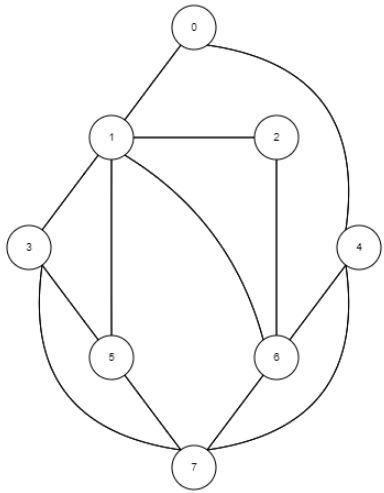
**Lab 4: Depth-First Search (Recursive and non-recursive implementation)**

You will be given the adjacency list representation of a graph. The *Graph* class reads in a text file which contains the edges and creates the adjacency list representation of the associated graph.

Graph’s logical representation **Adjacency List**



0: 1 > 4

**Step1:** DFS class takes an adjacency list representation of a graph to traverse through its edges:

Graph G= new Graph("edges2.txt", false);

DFS dfSearcher= new DFS(G.getAdjList());

Now, open DFS.java and you will see that a boolean array is used to keep track of the visited vertices. Implement following dfs method which traverses the graph using “**recursion”**. No stack!

public void dfs(int i) {

...

}

**Question 1:** Write the adjacency list representation of this graph for selected 3 vertices (only write “1”s) (10p)

**Question 2:** Starting at vertex 0, traverse the graph by depth-first-search and draw the traversal stack. (15p)

|  |  |  |
| --- | --- | --- |
| **Node** | **Stack in\_order** | **Stack out\_order** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Question 3:** In the main method (in Test.java), call dfs method with a starting index of “0” (0 as keyboard input), write down graph traversal. You should paste your dfs method code below. (25p)

**Question 3:** Using DFS, can you find if a given directed graph is cyclic? If you can find it, explain it step by step (Pseudo-code). (25p)

**Question 4:** See the non-recursive version of the dfs method using the “stack” data structure. Which version of the DFS better, non-recursive version or stack version? Why? Explain it. Compare time complexities. (25p)

public void dfs\_no\_recursion(int root) {

Stack<Integer> s = new Stack<Integer>();

s.push(root);

int current\_node;

visited[root] = true;

System.out.print(root + " ");

Repeats the followings until there are no elements in the stack:

At each iteration, looks at the last added element, and gets an unvisited neighbor of that element.

If there is such an element is found, marks that element as visited and pushes it onto the stack. This element then becomes the last added element in the stack.

If there is no such element, pops the last element from the stack.

int neighbor;

while (!s.isEmpty()) {

current\_node = s.peek();

neighbor = getUnvisitedNeighbor(current\_node);

if (neighbor != -1) { // root has a neighbor

visited[neighbor] = true;

System.out.print(neighbor + " ");

s.push(neighbor);

} else

s.pop();

} // end while

}

private int getUnvisitedNeighbor(int root) {

for (int neighbor : adjList[root]) {

if (!visited[neighbor]) {

return neighbor;

} // end if

} // end for each

return -1; // no unvisited neighbor has been found

}