Depth-first Search of a Digraph
(implemented using a stack)

Global variables and initializations:

S:	A stack of vertices. The vertices on *S* will be the vertices on the path from the starting vertex up to and including the current vertex.
adjList:	An array of lists. *adjList*[*v*] is the adjacency list of vertex *v*. For simplicity we allow the adjacency lists to be destroyed by the algorithm. *removeElement*(*adjList*[*v*]) will remove an element from *adjList*[*v*] and return it.
color:	an array of vertices. *color*[*v*] will be the color of vertex *v*. Initially *color*[*v*] = *white* for all vertices.
time:	relative time, initially 0.
d:	an array of vertices. *d*[*v*] will be the "discover time" of vertex *v*, i;e., the time at which *v* is pushed onto the stack, colored gray, and preprocessed.
f:	an array of vertices. *f*[*v*] will be the "finish time" of vertex *v*, i.e., the time at which *v* is postprocessed, colored black, and popped from the stack.

Depth first search of entire graph: *Perform depth-first search from some white vertex, then from a second white vertex, a third, etc., until no white vertices remain.*

```
void depthFirst( Digraph G)
    Initialize variables as above;
    for ( each vertex v of G )
        if ( color[v] == white )
            depthFirstFromVertex(G, v);
    return;
```

Depth first search from a single vertex: *Performs depth-first search from a specific starting vertex.*

```
void depthFirstFromVertex( Digraph G, Vertex start)
    discoverVertex( start);
    while ( not empty(S) )
        if ( adjList[top(S)] is nonempty )
            adjacent = removeVertex( adjList[top(S)]);
            if ( color[adjacent] == white )
                discoverVertex( adjacent);
            else if ( color[adjacent] == gray )
                Process back edge (top(S),adjacent);
            else if ( d[adjacent] < d[top(S)] )
                Process cross edge (top(S),adjacent);
            else
                Process forward edge (top(S),adjacent);
        else
            finishTopVertex();
    return;
```

__Discovering of a new vertex__: *Discover a new vertex x, adjacent to top(S). The new vertex is pushed on the stack, colored gray, and its dicover time is recorded. It undergoes any application-specific preorder processing.*

```
void discoverVertex( Vertex x)
    push( S, x);
    color[top(S)] = gray;
    d[top(S)] = ++time;
    Preorder process vertex top(S);
    return;
```

__Exiting from the current vertex__: *Exit a vertex after all edges out of the vertex have been explored. Any application-specific postorder processing of the vertex is performed. The vertex is colored black, and its finish time is recorded. It is then popped from the stack.*

```
void finishTopVertex( )
    Postorder process vertex top(S).
    f[top(S)] = ++time;
    color[top(S)] = black;
    pop(S)
    return;
```

The running time of this algorithm is $\Theta(n+e)$, apart from the time for application-specific processing of the vertices and edges.