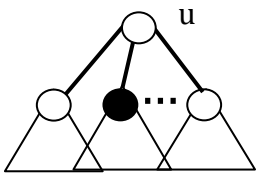
Iterative depth-first search
David Gries

You know that during execution of the recursive version of dfs, the call stack contains a frame for each call on dfs that has not completed. An iterative version of dfs also needs a stack. We develop an iterative dfs that maintains a stack of Nodes.

```
/** Visit every node reachable along a path of unvisited nodes from node u.
    Precondition: u has not been visited. */
public static void dfsIterative(Node u)
```



Here's the invariant of the loop that we will write:

> Invariant: all nodes (and only those nodes) that have to be visited are reachable along a path of unvisited nodes from some node is s.

The bit about (only those nodes) is necessary. Without it, placing *all* nodes in the stack initially would truthify the invariant, even nodes, for example, that are not connected to u.

From the method spec, we see that the invariant is truthified by starting with a stack that contains node u.

If s is empty, then by the invariant, all nodes that have to be visited have been visited.

In the body of the loop we first pop the top element from the stack and store it in u. We could have used a fresh local variable instead, but there is no need for that since the initial value of parameter u is not needed any longer.

If the popped node already visited, there's nothing to do. Otherwise, u is visited, and then, all neighbors of u are pushed onto the stack in order to again truthify the invariant.

That ends the development of an iterative version of depth-first search.

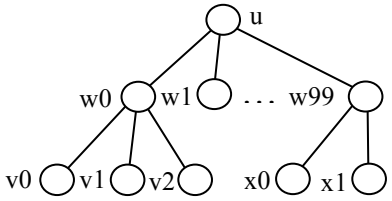
Some comments on this version of dfs.

1. The order in which neighbors are processed. The recursive dfs visits the neighbors from left to right: first w0 (and all nodes reachable along unvisited paths from it), then w1, and then w2.

The iterative version pushes the neighbors onto stack s in the order w0, w1, w2. This means that w2 will processed —and visited if necessary— first, and it's easy to see that the neighbors are thus visited in reverse order.

To get the same order, in the iterative version, push the neighbors onto s in reverse order.

2. A more efficient stack. Particularly when the out-degree of nodes can be large, pushing all neighbors of a node onto the stack is not efficient. We describe a better implementation of the stack. Assume the graph is undirected.



Let's assume that the neighbors of a node are maintained in an ArrayList. Using this graph as an example, assume that

> ArrayLists b, c, d, e contain the children of nodes u, w1, w2, w99:

ArrayLists of neighbors:
b is (w0, w1, …, w99)
c is (y0, y1, y2, u)
d is (u)
e is (x0, x1, u)

The iterative dfs first pushes u on the stack. Then, it pops u from the stack and pushed its neighbors on the stack —lets put them in in reverse order, so the stack now contains w99, …, w0.

Our alternative data structure is a stack t. The method starts by visiting u and then pushing on the stack an object that contains (a pointer to) b, which contains the neighbors of u, and the number 0. This denotes ArrayList elements b[0..].

We show one iteration of the main loop, using the two different stacks. In the original implementation, w0 is popped from the stack; it is visited; and v0, v1, v2, and u are pushed onto the stack.

In the implementation using stack t, a peek at top stack element (b, 0) results in b[0], so w0 is visited and (b, 0) is changed to (b, 1). Then, since w0 was visited, (c, 0), which contains w0's neighbors, is pushed onto stack t.

So you see that stacks s and t are just different ways to describe the same set of nodes. We leave it to you to implement iterative dfs using this second stack t.