COMP20007 Design of Algorithms

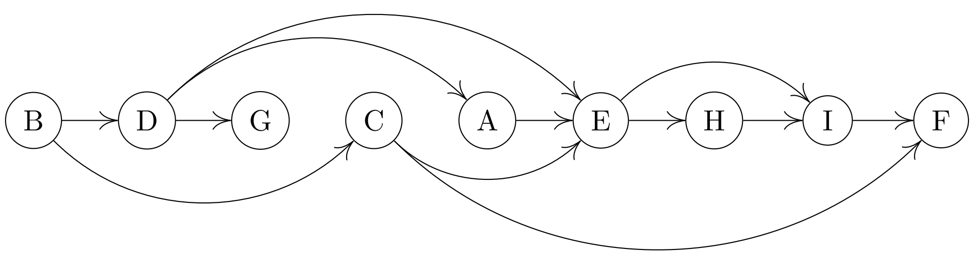
Workshop 6 Solutions

Tutorial

1. Topological sorting Running depth-first search from A results in the following sequence of operations (and resulting stacks):

push A	$S = \{A\}$
push E	$S = \{A, E\}$
push H	$S = \{A, E, H\}$
push I	$S = \{A, E, H, I\}$
push F	$S = \{A, E, H, I, F\}$
pop F	$S = \{A, E, H, I\}$
pop I	$S = \{A, E, H\}$
pop H	$S = \{A, E\}$
pop E	$S = \{A\}$
pop A	$S = \{\}$
push B	$S = \{B\}$
push C	$S = \{B, C\}$
pop C	$S = \{B\}$
push D	$S = \{B, D\}$
push G	$S = \{B, D, G\}$
pop G	$S = \{B, D\}$
pop D	$S = \{B\}$
pop B	$S = \{\}$

Taking the order of nodes popped and reversing it we get a topological ordering. So if we use a depth-first search starting from A, we get the topological ordering B, D, G, C, A, E, H, I, F. Rearranged into this order, the graph's edges all point from left to right:



2. Binary tree traversals The traversal orders of the example binary tree provided are:

Inorder: $0, 3, 4, 6, 7, 8, 9$

Preorder: $6, 4, 0, 3, 8, 7, 9$

Postorder: $3, 0, 4, 7, 9, 8, 6$

3. Level-order traversal of a binary tree The level-order traversal will visit the nodes in the tree in left to right order starting at the root, then doing the first level, the second level *etc.* For the binary tree in Question 3 the level-order traversal will be: 6, 4, 8, 0, 7, 9, 3.

We can use breadth-first search to traverse a binary tree in level-order, as long as we break ties by selecting left children first.

```
function LevelOrder(root)
    init(queue)
    enqueue(queue, root)
    while queue is not empty do
        node ← dequeue(queue)
        visit node
        if leftChild(node) is not NULL then
            enqueue(queue, leftChild(node))
        if rightChild(node) is not NULL then
            enqueue(queue, rightChild(node))
```

4. Binary tree sum Our recursive Sum algorithm will return the sum of the subtree T. We'll process the nodes in pre-order traversal order.

```
function Sum(T)
    if T is non-empty then
        sum ← value(T_root)
        sum ← sum + Sum(T_left)
        sum ← sum + Sum(T_right)
        return sum
    else
        return 0
```