Exercise – Graph Traversal

To traverse a graph, we have 2 basic methods, a "Depth First" and a "Breadth First". We're going to take a look at a Breadth First algorithm.

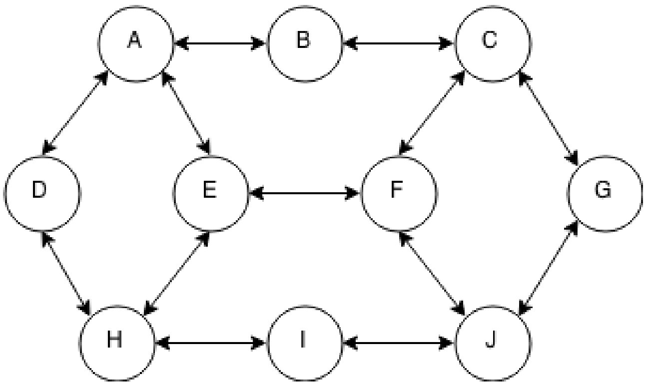
Consider the following graph implementation. The below code defines a structure for nodes and edges. Each Node contains a collections of Edges. Defined are 2 functions, AddNode and AddConnection. The add connection function creates an edge connecting the nodes.

```
class Graph
{
public:
	// predefine classes
	struct Node;

	struct Edge
	{
		Node *connection;

		// default constructor
		Edge() : connection(NULL) { }

		// overloaded construct
		Edge( Node *node ) : connection(node)  { }
	};

	struct Node
	{
		int value;
		std::vector< Graph::Edge > connections;

		// default constructor
		Node() : value(0) {}

		// overloaded constructor
		Node(int val) : value(val) {}
	};

	~Graph()
	{
		// make sure to delete all nodes added to the graph
		for(auto iter = m_nodes.begin(); iter != m_nodes.end(); iter++)
			delete (*iter);

		m_nodes.clear();
	}

	Node *AddNode( int value )
	{
		m_nodes.push_back( new Node(value) );
		return m_nodes.back();
	}

	void AddConnection( Node *n1, Node *n2)
	{
		n1->connections.push_back( Edge(n2) );
		n2->connections.push_back( Edge(n1) );
	}

private:

	std::vector< Node * > m_nodes;

};
```

The above graph node's can store an integer value. Lets define a graph structure. Create a new console application, add the graph class above, and add the following.

```
int main(int argc, char **argv)
{
	Graph *pGraph = new Graph();

	Graph::Node *a, *b, *c, *d, *e, *f, *g, *h, *i, *j;

	a = pGraph->AddNode('a');     b = pGraph->AddNode('b');
	c = pGraph->AddNode('c');     d = pGraph->AddNode('d');
	e = pGraph->AddNode('e');     f = pGraph->AddNode('f');
	g = pGraph->AddNode('g');     h = pGraph->AddNode('h');
	i = pGraph->AddNode('i');     j = pGraph->AddNode('j');

	pGraph->AddConnection( a, b );        pGraph->AddConnection( a, d );
	pGraph->AddConnection( a, e );        pGraph->AddConnection( b, c );
	pGraph->AddConnection( d, h );        pGraph->AddConnection( e, h );
	pGraph->AddConnection( e, f );        pGraph->AddConnection( f, c );
	pGraph->AddConnection( f, j );        pGraph->AddConnection( g, c );
	pGraph->AddConnection( g, j );        pGraph->AddConnection( i, j );
	pGraph->AddConnection( i, h );


	delete pGraph;

	return 0;
};
```

The above will represent a graph that can be visually represented as:



Let's take a look at pseudo code for the BFS algorithm. We require a stack that contains a list of nodes needing to be processed.

While the stack is not empty, take the first node, remove it from the stack and add each of its children to the stack for processing.

We need to repeat this process until the stack is empty... If our graph was ACyclic, than this process would work perfectly. However, our graph is Cyclic, so this will end up adding children that we have already visited, causing the stack to continually grow until we run out of memory.

For this reason, we also need to keep track of nodes that have been visited, making sure we dont add them to the stack.

We have 2 options.

1. Add a Boolean "visited" variable to the graph node. When we process the node, set its visited variable to true.

2. Create a list of nodes already visited, called the "closed list". Don't add the node to the graph if it's on the closed list.

Here some more formal pseudo code, I'll use the second option for recording which items have already been visited.

```
Procedure BFS(startNode)

    Let openList be a Queue
    Let closedList be a List

    Add startNode to openList

    While openList is not empty

        Let currentNode = next item in openList

        // Process the node, do what you want with it. EG:
        Print value of currentNode to console

        remove currentNode from openList
        Add currentNode to closedList

        for all connections c in currentNode
            Add c to openList if not in closedList
```

Using a simular approach to the above, a DFS can also be achieved by changing the open list from a Queue to a Stack. Remember, a Queue is a FIFO structure (First in First Out) where as a stack is a LIFO (Last in First Out).

Exercise:

Implement the above pseudo code for both BFS and a DFS. Print the value of each node before it's added to the open List.

```
void PrintBFS(Graph::Node *startNode)
{
        // TODO: code goes here
}

void PrintDFS(Graph::Node *startNode)
{
        // TODO: code goes here
}
```

Exercise

Looking at these algorithms, the only real difference is how the open List is sorted. Rather than treating the open List as a stack or queue collection. Let's just treat it as a standard list and preform a sort operation.

We will need to store an integer value in the node class, representing degrees Of Seporation.

```
struct Node
{
        int value;
        int dos;
        std::vector< Graph::Edge > connections;

        // default constructor
        Node() : value(0), dos(0) {}

        // overloaded constructor
        Node(int val, int ds) : value(val), dos(ds) {}
};
```

when we add a node to the open List, increment the added nodes degrees Of Separation.

This will allow us to sort the open List by degrees of.

The BFS is where the open List is sorted in ascending order, meaning we process the next node on the open list as the one with the lowest degrees of separation.

The DFS is where the open List is sorted in descending order, meaning we process the next node on the open list as the one with the highest degrees of separation.

Updated pseudo code:

```
Procedure BFS_DFS(startNode)

    Let openList be a List
    Let closedList be a List

    Add startNode to openList

    While openList is not empty

        if processing as BFS
            sort openList ascending by degreesOfSeportation

        if processing as DFS
            sort openList descending by degreesOfSeportation


        Let currentNode = first item in openList

        // Process the node, do what you want with it. EG:
        Print value of currentNode to console

        remove currentNode from openList
        Add currentNode to closedList

        for all connections c in currentNode
            if c is not on closedList
               let c.degreesOfSeporation = currentNode.degreesOfSeporation + 1
```

```
            Add c to openList
```

This altered approach is an important concept to understand, as this is the base concept underlying how Dijkstras and AStar Path finding algorithms work.