7.1
A complete graph is an undirected graph where each distinct pair of vertices has an unique edge connecting them. This is intuitive in the sense that you are basically choosing 2 vertices from a collection of n vertices simplified to C(n,2). Using the formula for binomial coefficients, C(n,2) = n(n-1)/2. In terms of big O it is O(n^2)

7.2
d of a V is the number of m connected to it
D/2 = m because in a connected graph you have vertices getting connections from two other vertices

7.3
1 - 2 - 3 - 1
Pros - transversal
Cons - Each edge needs twice as much storage space.

7.4
The size of the adjacency matrix representation is O(V * V) and the size of the adjacency list representation is O(|V|+|E|)

7.5
```
LinkedList<GraphE> [] adjList
int n
public adjacencyList( int n)
{
  This.n = n
  This.adjList =  LinkedList<GraphE> [] new LinkedList<> [n]
  for (int i = 0; i < n; i++)
        {
          adjList[i] = new LinkedList<GraphE>
        }
}
```
7.6

By checking the outputs we could find out whether the graph is a connected tree. If the output of numConnectedComponents is 1 and the output of exist cycle is false then the graph is a connected tree.

7.8
We need to modify the code to add a counter variable to the breadthFirstMatrix method and we have to increment it at the end of the program after line 9

7.9

Because V^2 is the largest term we can really discard the other elements. An adjacency list is more efficient in terms of storage because it only stores what is necessary.

7.12
Vertex 0 leads to an infinite loop because it just iterates through the same elements, we can solve this by adding a counter to check if all nodes were visited

7.13

Order the DFS
Loop through the graph
Get visited list
At i you need to check all of the vertices that it are connected to vertex at i
Recursively check its connected vertices