```go
func NewGraph(vertices int) *Graph { // initializes an undirected graph with a given number of vertices
    adjList := make([]*Node, vertices)
    return &Graph{
        vertices: vertices,
        adjList:  adjList,
    }
}

func (g *Graph) AddEdge(v1, v2 int) { // adds an undirected edge between two vertices
    if v1 >= 0 && v1 < g.vertices && v2 >= 0 && v2 < g.vertices {
        nodeV1 := &Node{vertex: v2, next: g.adjList[v1]}
        g.adjList[v1] = nodeV1
        nodeV2 := &Node{vertex: v1, next: g.adjList[v2]}
        g.adjList[v2] = nodeV2
    }
}

func (g *Graph) PrintList() { // displays adjacency list
    for vertex, node := range g.adjList {
        fmt.Printf("Vertex %d -> ", vertex)
        for node != nil {
            fmt.Printf("%d ", node.vertex)
            node = node.next
        }
        fmt.Println()
    }
}

// --- USES ---
    // LINEAR time complexity of O(n)
    // space complexity of O(numVertex + numEdge)
        // uses less space compared to an ADJACENCY MATRIX for any given dataset
```

Tree

```go
// --- TREE ---
    // non-linear collection of nodes (which store data) organised in a hierachy, where nodes are connected by edges
    // root node: top-most node with no incoming edges
    // leaf node: bottom-most nodes with no outgoing edges
    // branch nodes: nodes in the middle with both incoming and outgoing edges
    // parent nodes: any node with an outgoing edge
    // child nodes: any node with an incoming edge
    // sibling nodes: any nodes sharing the same parent node
    // subtree: smaller tree nested within a larger tree
    // size of tree => total number of nodes
    // depth of node => number of edges below root node
    // height of node => number of edges above furthest leaf node
```