Recall the ADT binary tree: a tree structure used mainly to represent 1 to 2 relations, i.e. each item has at most two immediate successors.

Limitations of tree structures:
- an item in a tree can appear in one position only
- an item has exactly one predecessor (parent node) unless it is the root (none)
- the only available links are parent/child links

E.g.: Attempting to represent a transport link data with a tree structure:

BUT: Nodes would appear in more than one place;
Nodes would have more than one predecessor (e.g. Miami)
Definition

- The **ADT Graph** is a graph treated as abstract data type, consisting of
  - a set \( V \) of items (nodes), and
  - a set \( E \) of edges, each linking 2 nodes.

In the previous example:

\[
V = \{\text{Toronto, Vancouver, New York, Washington, Miami, Los Angeles}\}
\]

\[
E = \{(\text{Vancouver, New York}), (\text{Vancouver, Los Angeles})
\text{(Vancouver, Miami)}, (\text{Los Angeles, Toronto})
\text{(Toronto, Washington)}, (\text{Washington, New York})
\text{(Miami, Los Angeles)}\}
\]

- Edges do no indicate a direction, unless the graph is **directed**. The example is an
  undirected graph, where an edge (Vancouver, New York) implies also the edge
  (New York, Vancouver). In directed graphs both directions must be specified if
  both are to apply.

A graph consists of a set \( V \) of items (nodes) and a set \( E \) of edges each linking two nodes. Two nodes of a graph are said adjacent if they are joined by an edge. In the example above Miami and Los Angeles are adjacent nodes. A “path” between two nodes is a sequence of edges that begins at one node and it ends at the other node.

The example given in the previous slide is a undirected graph. Graphs can be directed or undirected. In the latter case the edges do not indicate a direction. That is in an undirected graph we can travel in either directions along the edges between nodes. In contrast, each edge in a directed graph has a direction, and we can only travel along the edges in the direction indicated. Therefore, if we want to consider both directions between two given nodes we will have to specified two directed edges between these nodes one for each direction.

An ADT Graph is essentially a graph treated as Abstract Data Type.
Access Procedures

**Constructor operations** to construct a graph:
1. `createGraph()`  
   // post: creates an empty graph
2. `addNode(newItem)`  
   // post: adds newItem in a graph. No change if newItem already exists.
3. `addEdge(Node1, Node2)`  
   // post: adds an edge between two existing nodes (Node1 and Node2)

**Predicate operations** to test graph
1. `isEmpty()`  
   // post: determines whether a graph is empty
2. `isLink(Node1, Node2)`  
   // post: returns true if edge (Node1, Node2) is present in the graph
3. `addEdge(Node1, Node2)`  
   // post: adds an edge between two existing nodes (Node1 and Node2)

**Selector operations** to select items of a queue
1. `deleteNode(Node)`  
   // post: remove a node and any edges between this node and other nodes.
2. `deleteEdge(Node1, Node2)`  
   // post: delete the edge between the two given nodes Node1, Node2.

This slide provides the list of the main access procedures and operations for the ADT graph. Note that insertion and deletion operations are different from those seen so far for the other ADTs, in that they can apply to either nodes or edges. Nodes of a graph may or may not contains a value. In the example considered so far the nodes do contain a value. Graphs whose nodes do not contain values are used mainly to represent relationships.

Another important procedure used with ADT graphs is “findPath”. This is defined as

```
findPath(Node1, Node2)
//pre: Node1 and Node2 are existing nodes in the graph
//post: it returns the list of nodes on a path from Node1 to Node2.
```
The two most common implementations of a graph are the **adjacency matrix** and the **adjacency list**. The first is a static implementation whereas the second is a dynamic implementation.

For a graph with N nodes, is an N x N array matrix such that matrix[i][j]=1 (or true) is there is an edge from node i to node j, and 0 (or false) otherwise. This slide shows the adjacent matrix representation of the graph given in the first slide.

Note that this matrix is symmetrical, that is matrix[i][j] = matrix[j][i]. This is not necessarily the case for a directed graph. Moreover, the matrix itself does not include the nodes values. We have used them here as index values. You could implement this by using a second array values[0..N-1] where each location values[i] contains the value of the node i in the graph.

This implementation presents, however, some limitations. Firstly, it has to include a value for each entry in the matrix. This can cause waste of memory in particular when the graph includes only few of its possible edges. Secondly, it is quite an inflexible structure in the sense that for some operations like adding a new node to the graph it requires the whole matrix to be redefined.

On the other hand, this structure allows the operation isLink of determining whether there is an edge between two given nodes Node1 and Node2 to be performed very efficiently. It would simply consist of checking the value of matrix[i][j], where i is Node1 and j is Node2.
Dynamic Implementation of a Graph

For N nodes, we have N linked lists. Each node includes two reference variables, one for the linked list of all the edges starting from it, and the other to reference to other nodes in the graph so to allow graph traversal.

What are the Java type declarations for this ADT?

The dynamic implementation of a graph with N nodes consists of N linked lists each referenced by each node in the graph. The i\textsuperscript{th} linked list specifies all the edges in the graph that start from the i\textsuperscript{th} node, together with the value of the nodes adjacent to the i\textsuperscript{th} node.

The “node” data structure representing each node in the graph includes two reference variables, and the actual value of the node. The first reference variable is used to reference the linked list of its adjacent nodes, whereas the second reference variable can be used to reference to other nodes of the graph in order to allow going from one node to another in the graph.
Pros and Cons of Dynamic Implementation

**Advantages:**

- Easy to retrieve all the adjacent nodes of a given node (just traverse the linked list associated with the node).
- Easy to check whether a node has edges starting from it
- Less memory space is needed.

**Limitation:**

- More complex operation for checking whether there is an edge between two given nodes.

This data structure implementation has various advantages with respect to the static implementation. In particular, one of the typical operations done on a graph, which is finding all nodes that are adjacent to a given node, can be performed very efficiently. In this case it would be sufficient to return the linked list referenced by the node under consideration. If the referenced linked list is empty than it means that the node does not have any adjacent node. In the static implementation, this operation would be much less efficient as it would require the traversal of the row in the matrix that corresponds to the particular node in consideration. This would be necessary also when all the entries in this row are equal to zero!

Another important advantage is that the adjacency list takes less memory space than the adjacency matrix. This is simply because in the first case you need to represent the nodes and only the edges that are in the graph. In the matrix case you will always have to create \( n^2 \) entries independently on how few edges are in the graph.

However, the operation of checking if there is an edge between two given nodes is less efficient for the adjacency list because we have to consider the linked list associated with one of these two nodes and traverse it in order to determine if the other node is in this list. So the operation is not as direct as it is in the case of the adjacency matrix.

Hence, looking at these two different types of implementation of a graph we can conclude that they respectively have their pros and cons. The choice ultimately depends on the application and the type of operations that the application will most frequently require.
**Definition**

- A **Weighted Graph** is a graph in which each edge has a weight value associated with it; depending on the application this may represent distance, cost, or other variable.
- Path searching applications may have to take into account the weight of each edge in determining a solution.

**Representation of the weight of an edge:**
- The adjacency matrix may have 0 (=no edge) or the weight at each entry.
- In adjacency list, the node for each edge now holds the weight too.

San Francisco 2600 New York  
900 Los Angeles 150 Providence

The ADT Graphs
Graph Traversals

Use the same depth-first and breadth-first traversal algorithms seen for the binary trees.

Differences between graph and tree traversals:

• Tree traversal always visit all the nodes in the tree
• Graph traversal visits all the nodes in the graph only when it is connected. Otherwise it visits only a subset of the nodes. This subset is call the connected component of the graph.

Recursive and iterative implementations of the algorithms.

Iterative: Use a stack for the depth-first search (dfs)
Use a queue for the breadth-first search (bfs)
We can now conclude this second part of the course with the same overview slide I gave you in the introduction lecture, but this time as a summary slide of what we have seen and what you are supposed to know.
The End