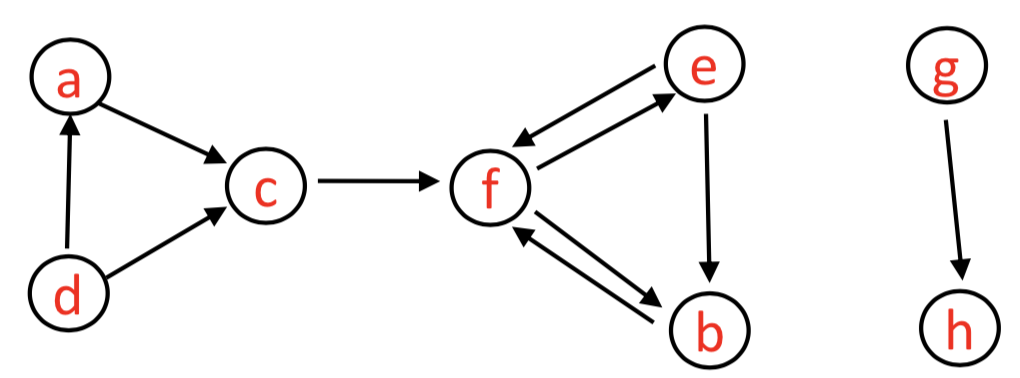
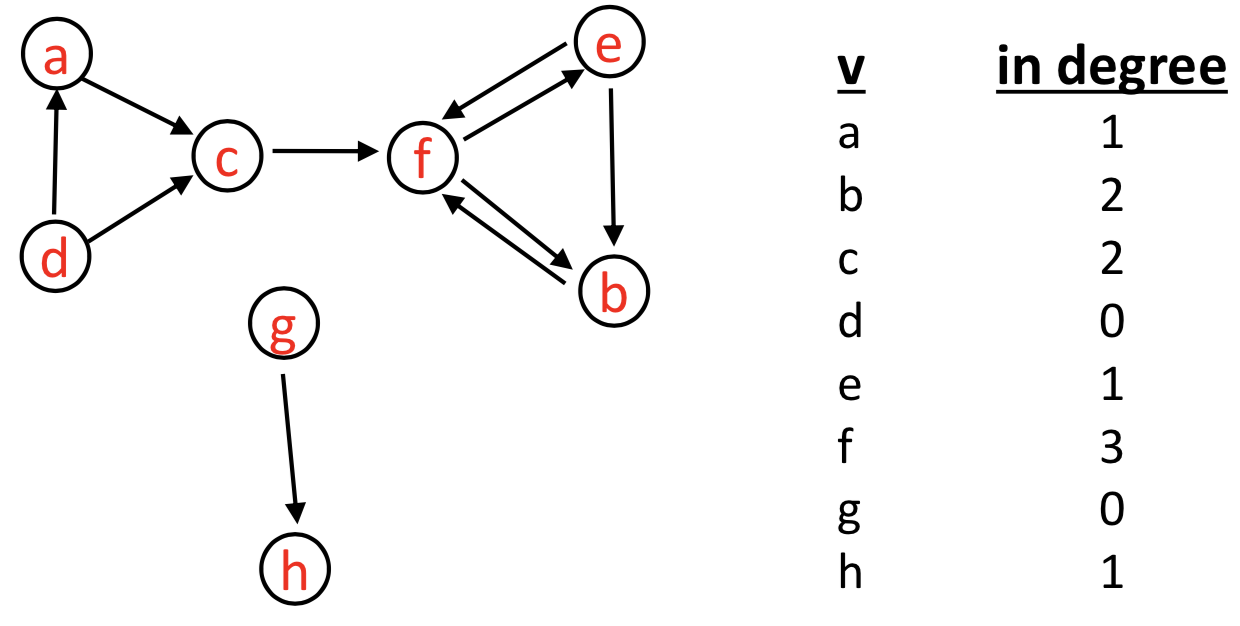
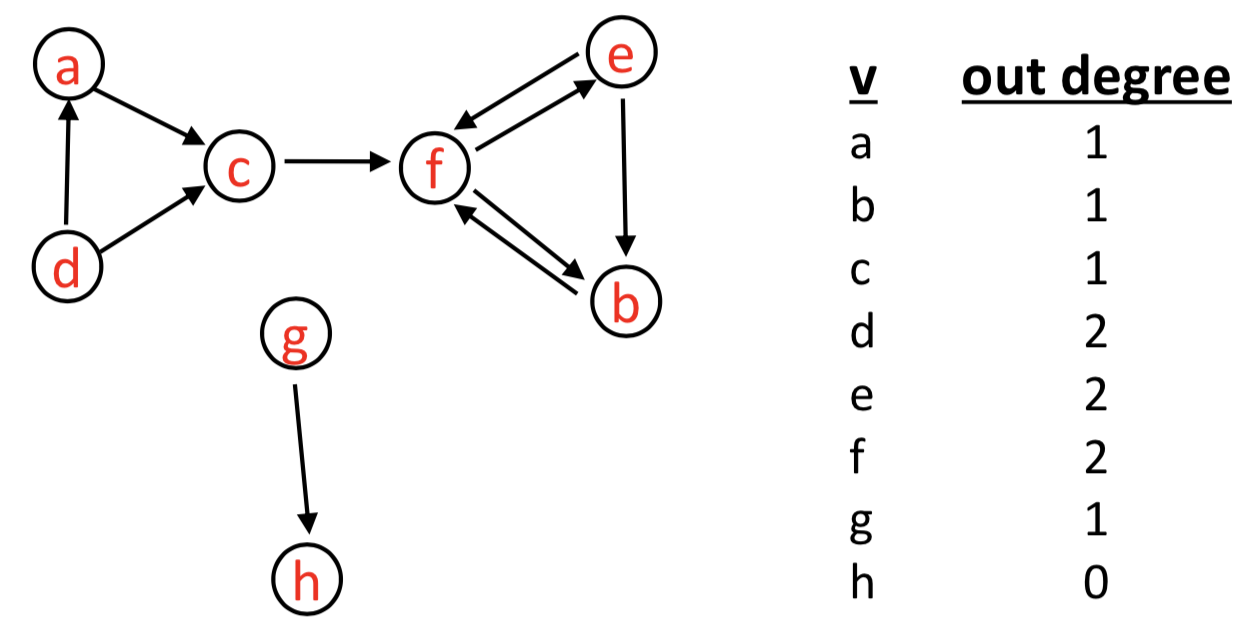
COMP250-Graph



A *directed graph* is a set of **vertices** 𝑉= {𝑣𝑖 : 𝑖∈1,...,𝑛} and set of ordered pairs of these vertices called **edges** 𝐸={ (𝑣i ,𝑣j )∶𝑖, 𝑗∈1,...,𝑛 }.

A *undirected graph* is a set of **vertices** 𝑉= {𝑣𝑖 : 𝑖∈1,...,𝑛} and set of unordered pairs called **edges** 𝐸={ (𝑣i ,𝑣j )∶𝑖, 𝑗∈1,...,𝑛 }.

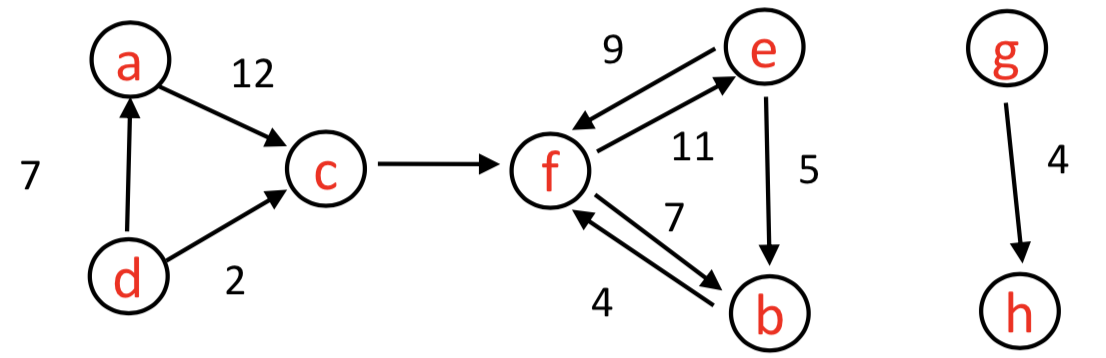
A *path* is a sequence of edges such that end vertex of one edge is the start vertex of the next edge. No vertex may be repeated except first and last.

Examples: • acfeb • dac • febf

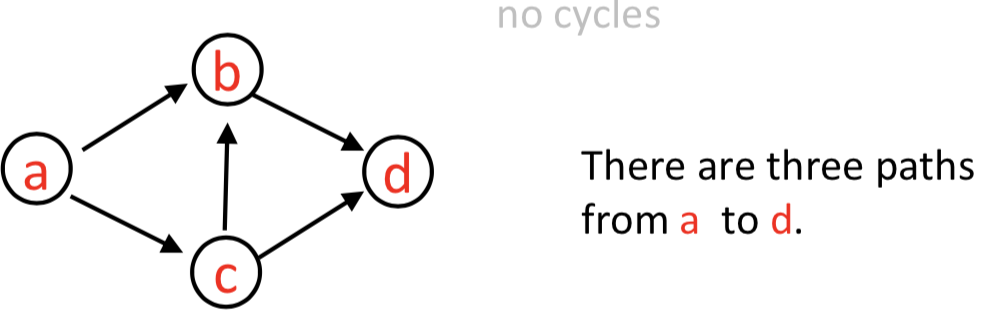
A *cycle* is a path such that the last vertex is the same as the first vertex.

Example: • febf • efe • fbf

[Weighted Greaph]



[Directed Acyclic Graph] NO CYCLES



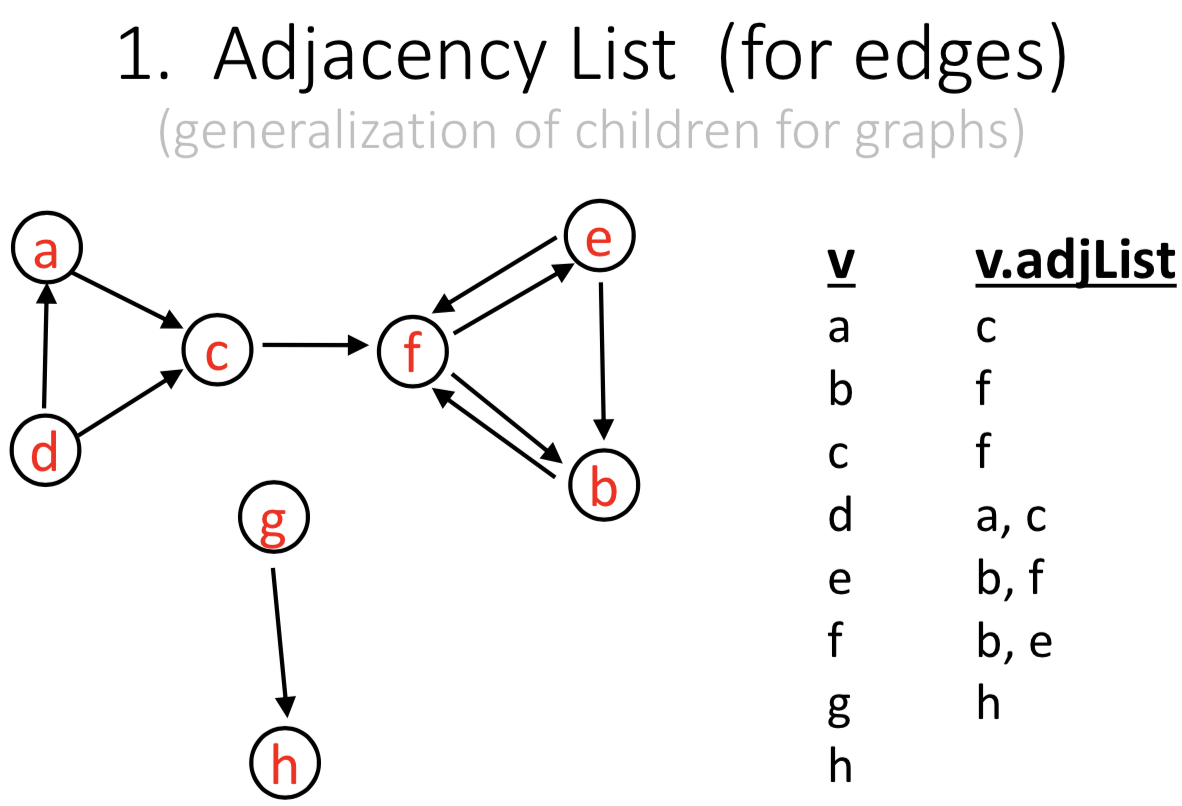
Used to capture **dependencies**. e.g. a implies b, or a must happen before b can happen, etc.

Graph ADT

Graphs are a generalization of trees, but a graph does NOT have a root vertex.

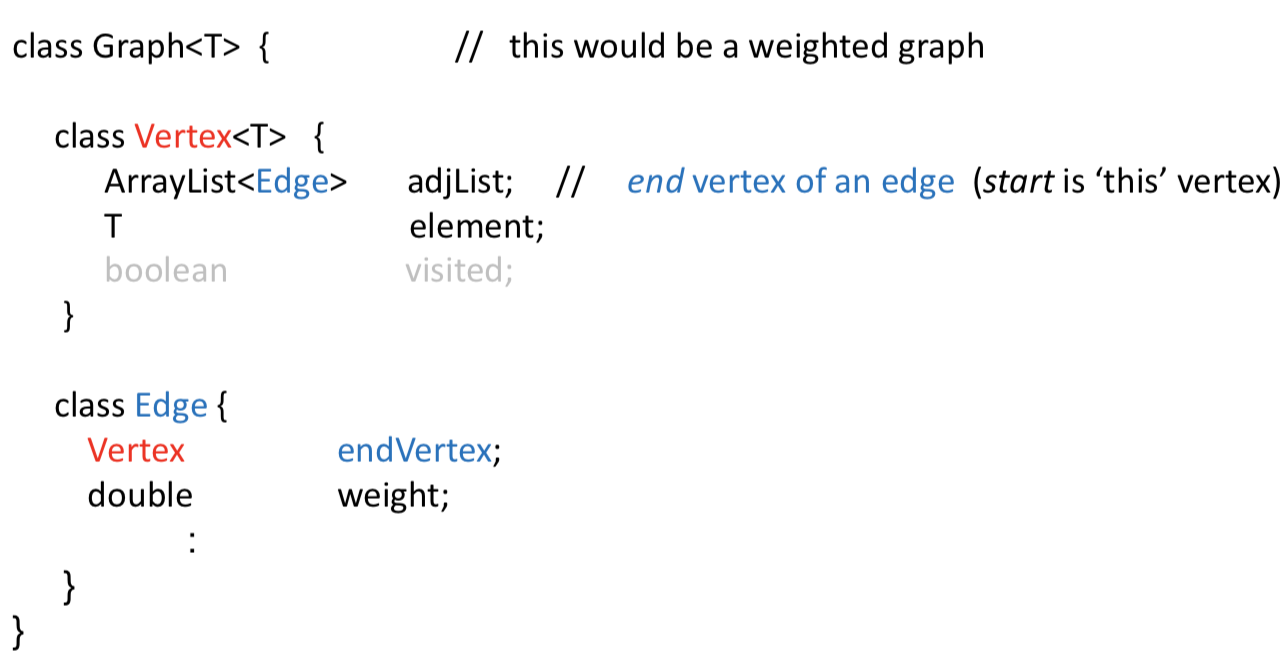
Outgoing edges from a vertex in a graph are like children of a vertex in a tree. Incoming edges are like parent(s).

**1. Adjacency List (for edges)** generalization of children for graphs



Here each adjacency list is sorted, but that is not always possible (or necessary).





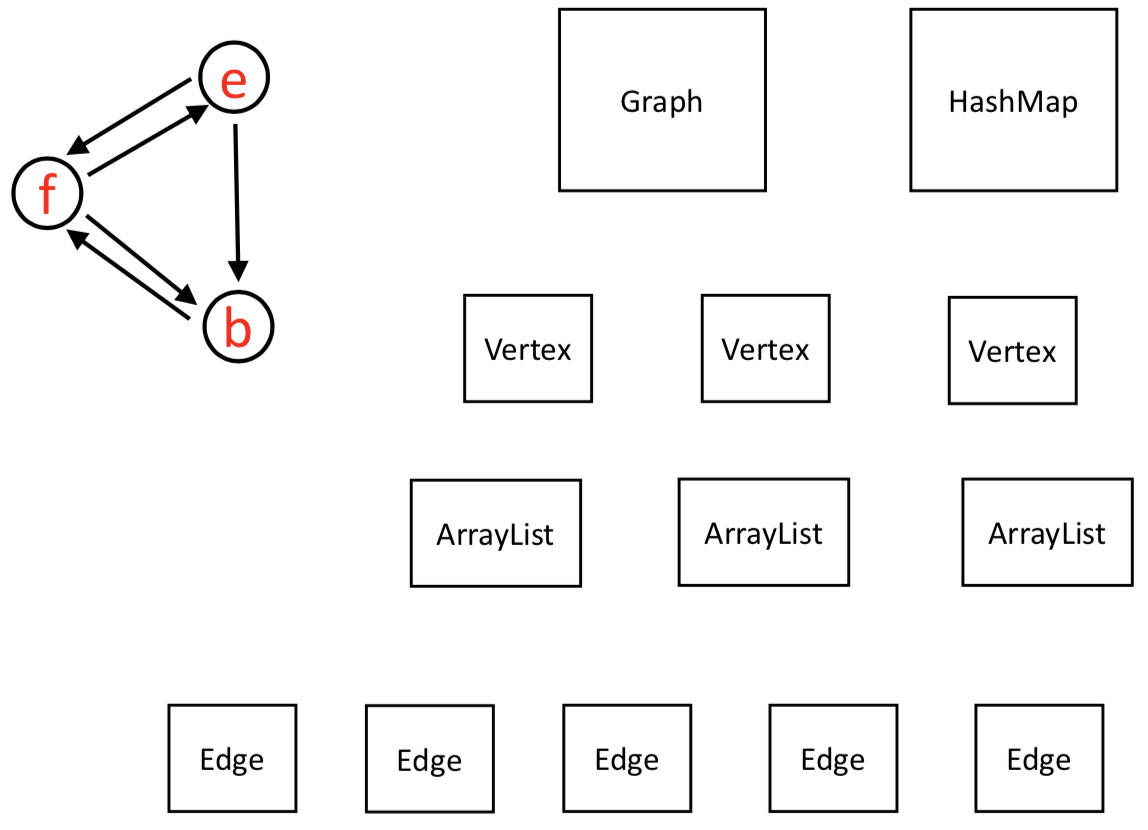
How to access vertices?

We can have a string name (key) for each vertex. e.g. YUL for Trudeau airport

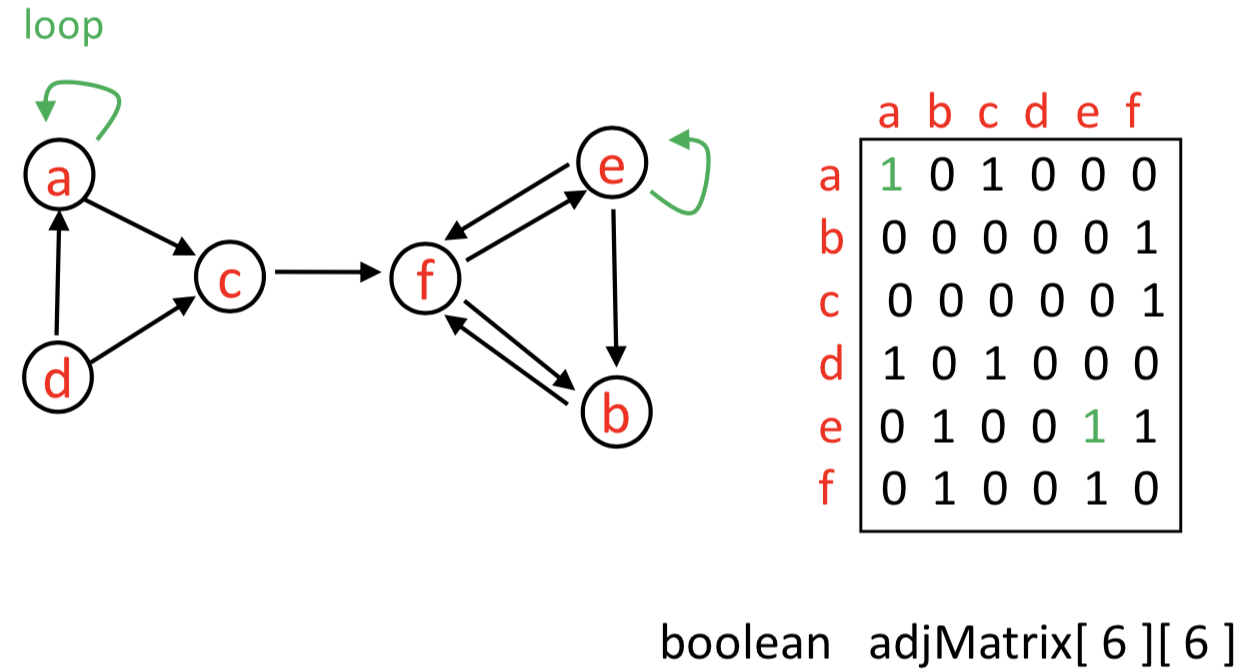


We could also just have a list of vertices.

How many objects?



**2.Adjacency Matrix**



Assume we have a list of vertex names i.e. a unique mapping from vertex names to 0, 1, .... , n-1 (not a hashmap).

Suppose a graph has 𝑛 vertices. (The following are not formal definitions.)

* the graph is dense if number of edges is close to 𝑛2.
* the graph is sparse if number of edges is close to 𝑛.

[Adjacency list VS adjacency matrix]

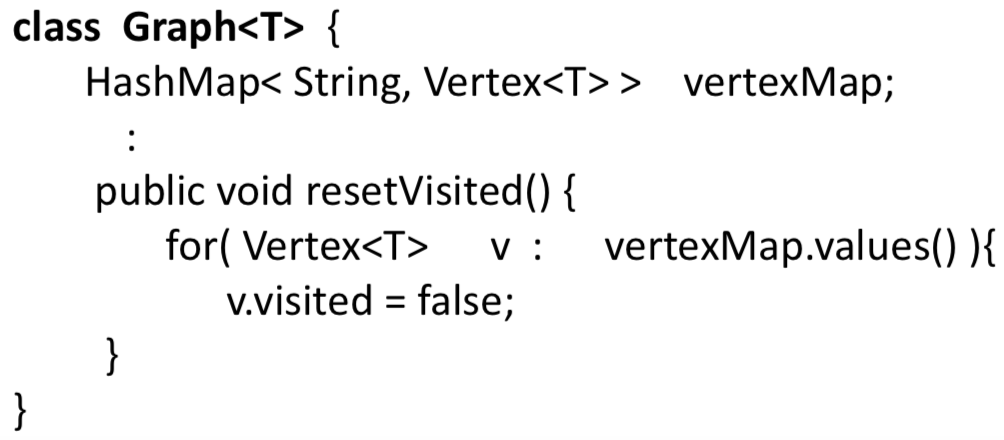
* The graph is sparse e.g. 10,000 vertices and 20,000 edges and we want to use as little space as possible.
* The graph is dense e.g. 10,000 vertices and 20,000,000 edges, and we want to use as little space as possible.
* Answer the query areAdjacent() as quickly as possible, no matter how much space you use.
* Perform operation insertVertex( v ).
* Perform operation removeVertex( v ).

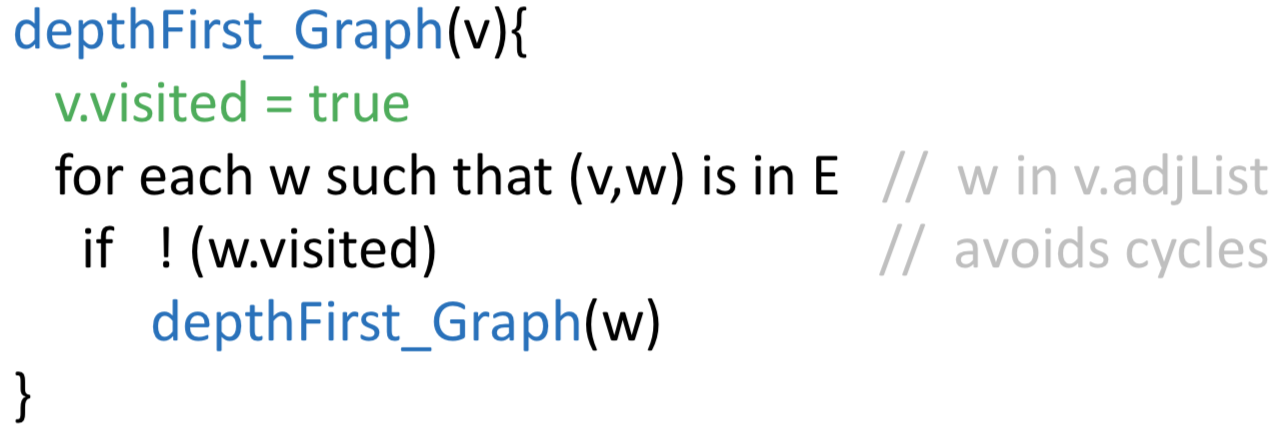
Graph Traversal compare with tree traversal

**Recursive graph traversal** • depth first

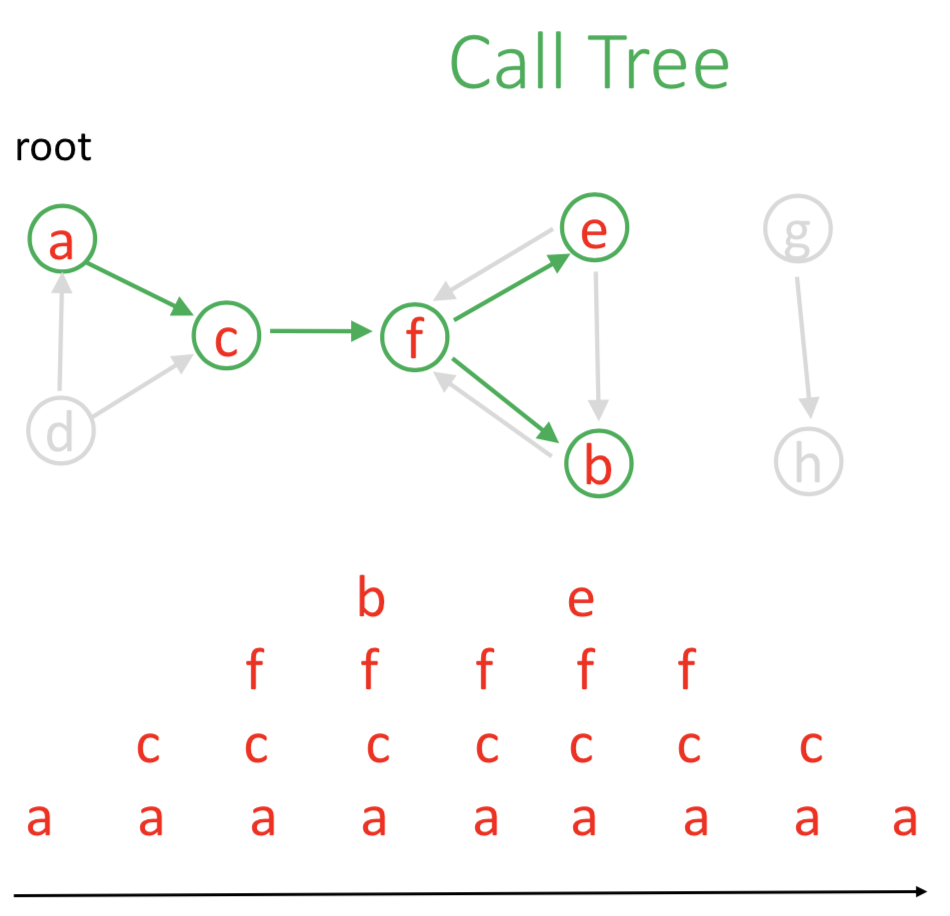
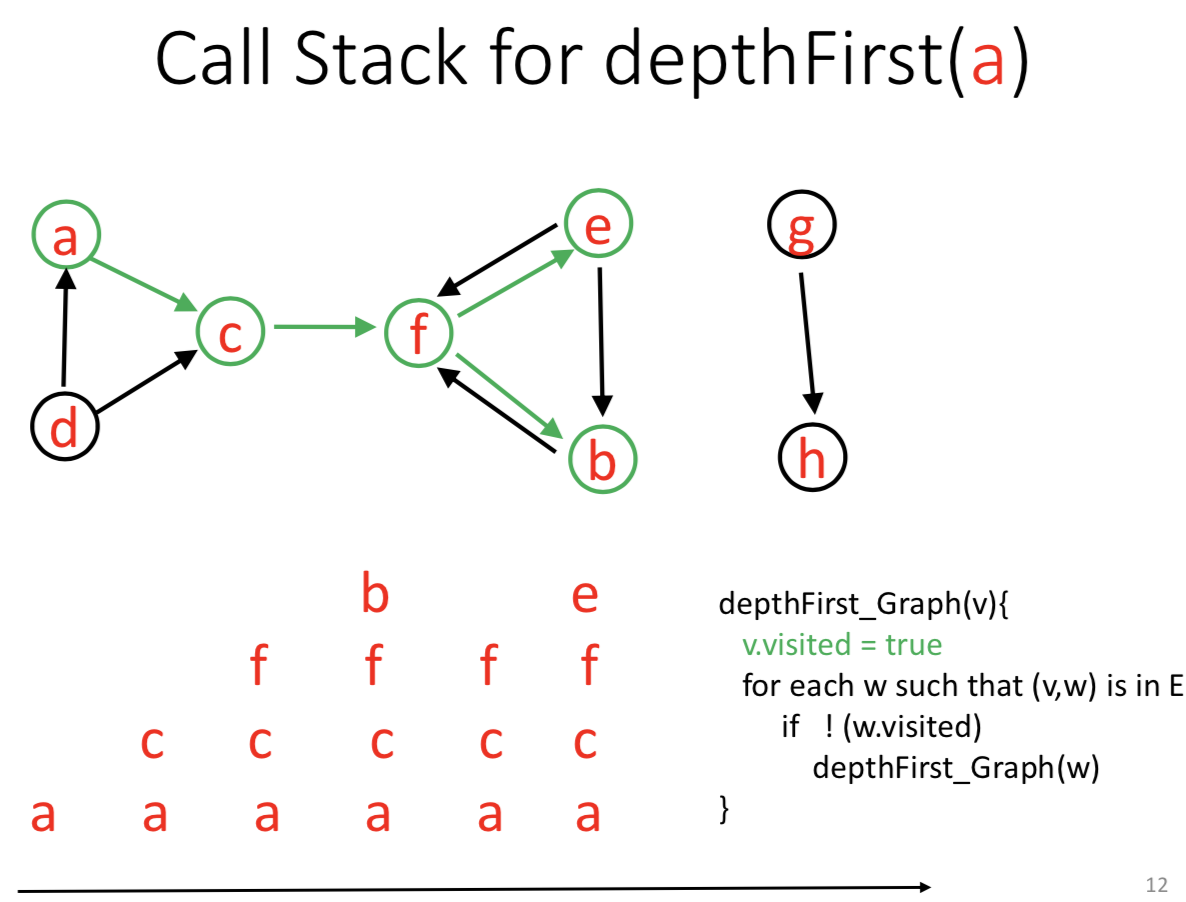
*Need to specify a starting vertex.*

*Visit all nodes that are “reachable” by a path from a starting vertex.*

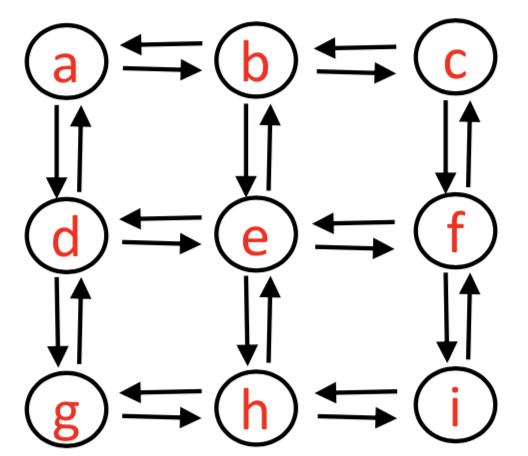
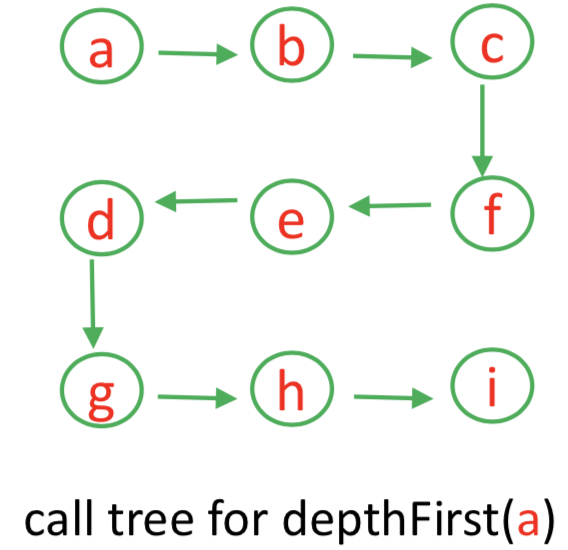




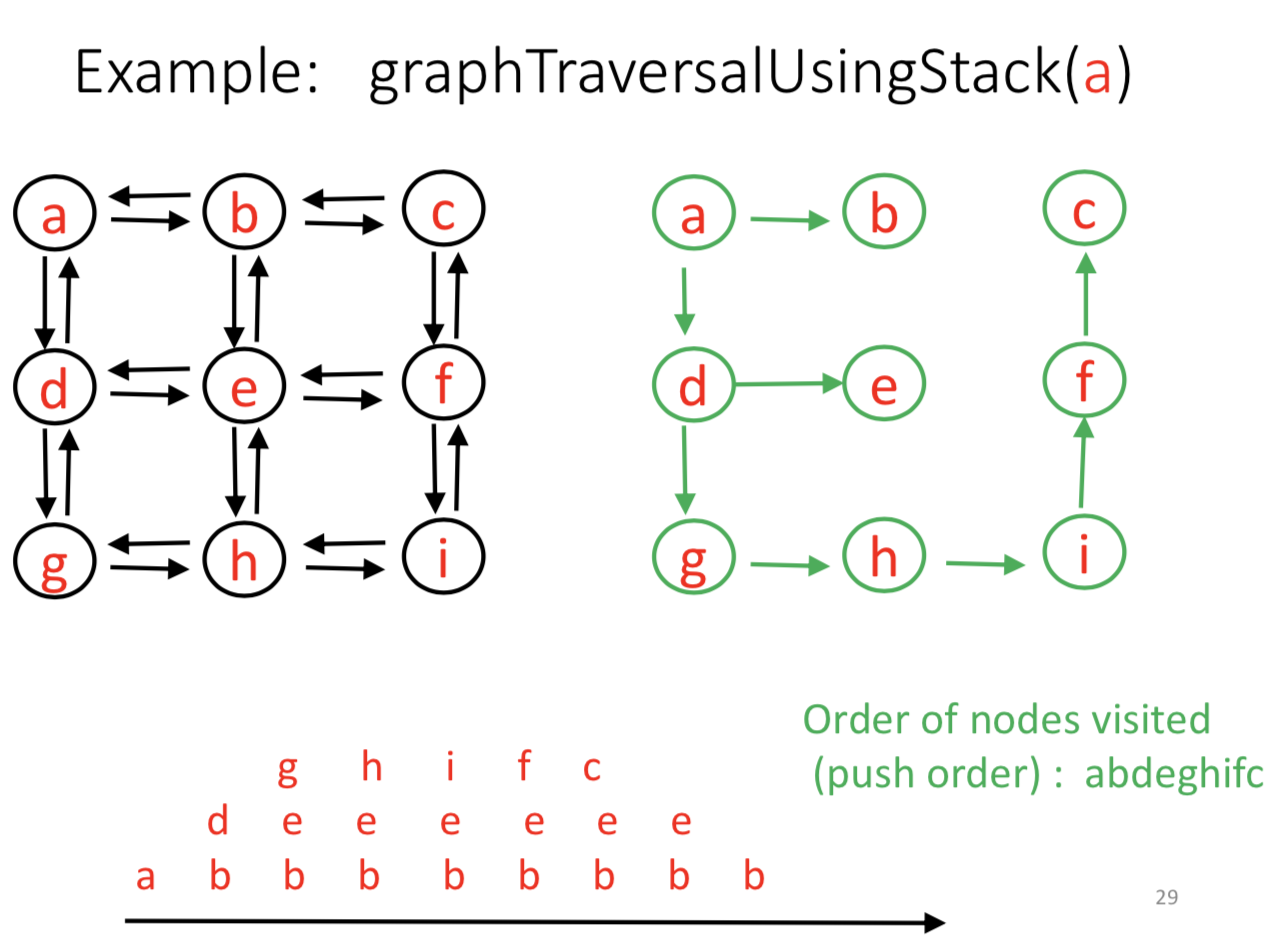
// Here “visiting” just means “reaching”



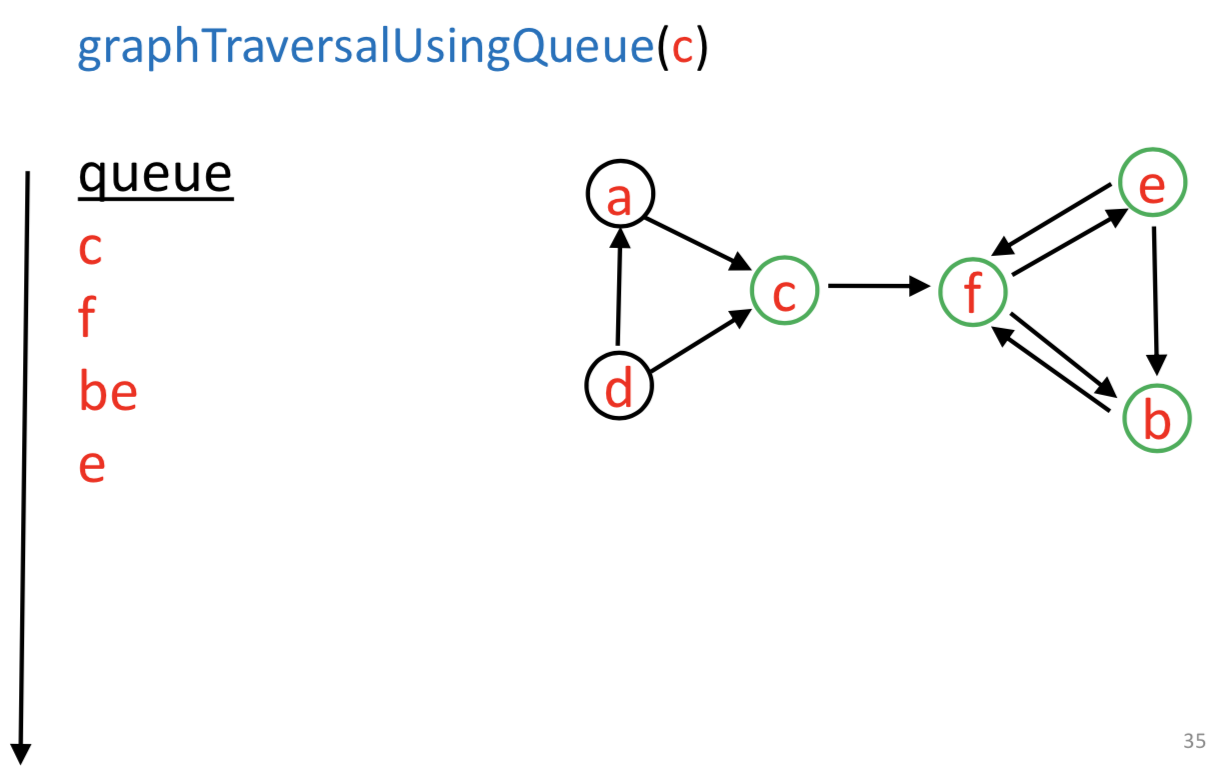
Unlike tree traversal for rooted tree, a graph traversal started from some arbitrary vertex does not necessarily reach all other vertices. The order of nodes visited depends on the order of nodes in the adjacency list.

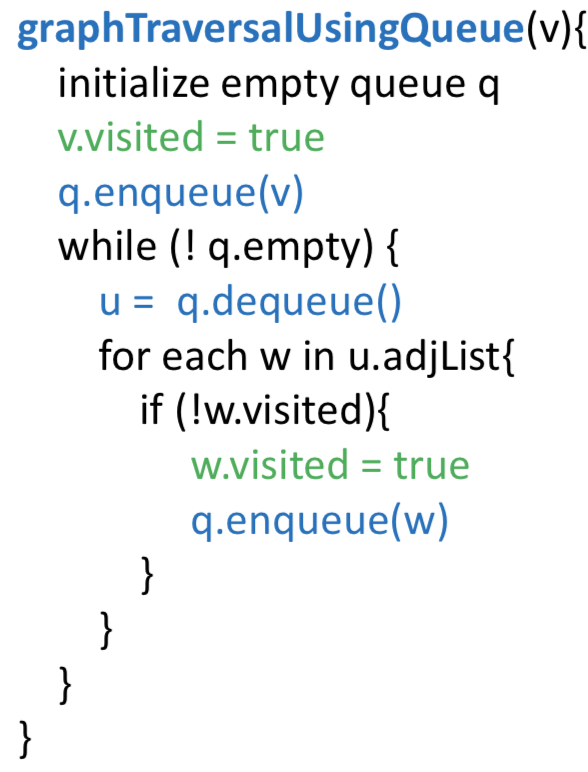


**Non-recursive graph traversal**

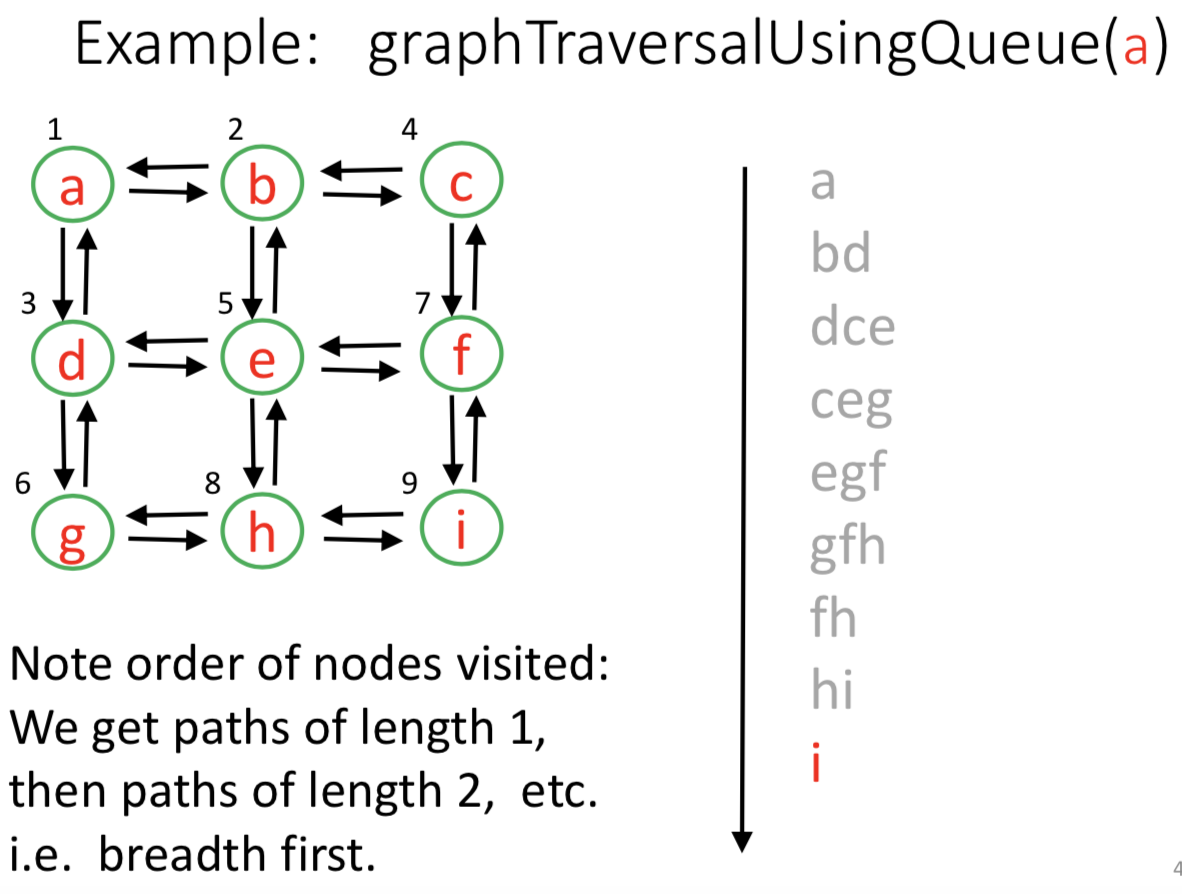
• depth first



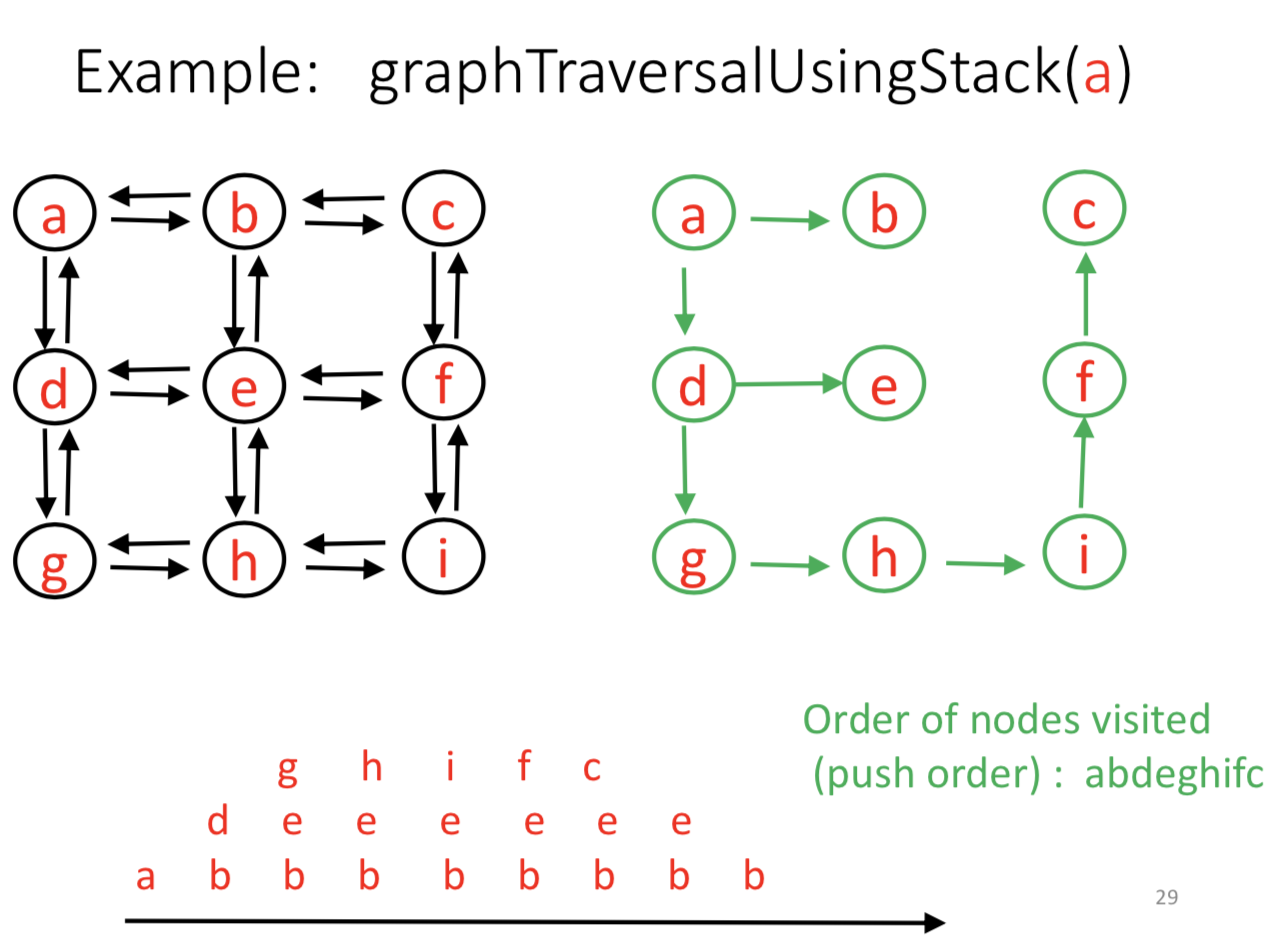
• breadth first



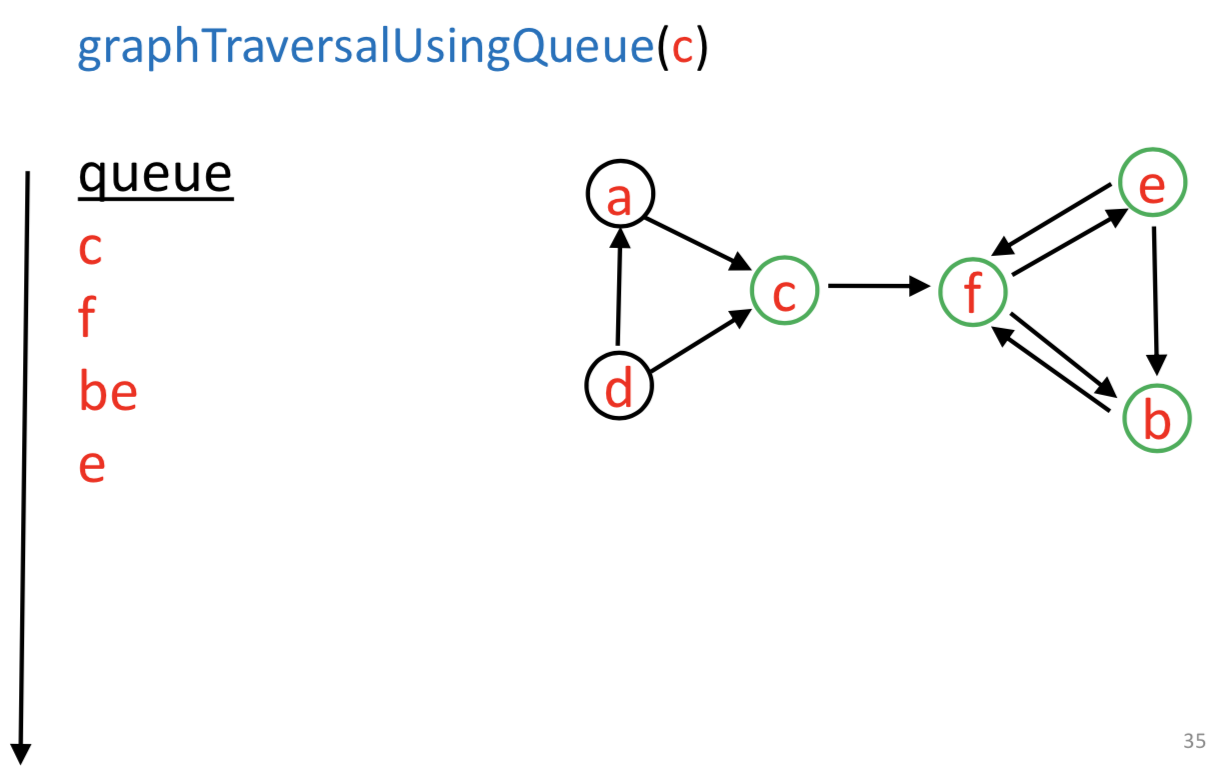
It defines a tree whose root is the starting vertex. It finds the shortest path (number of vertices) to all vertices reachable from starting vertex.

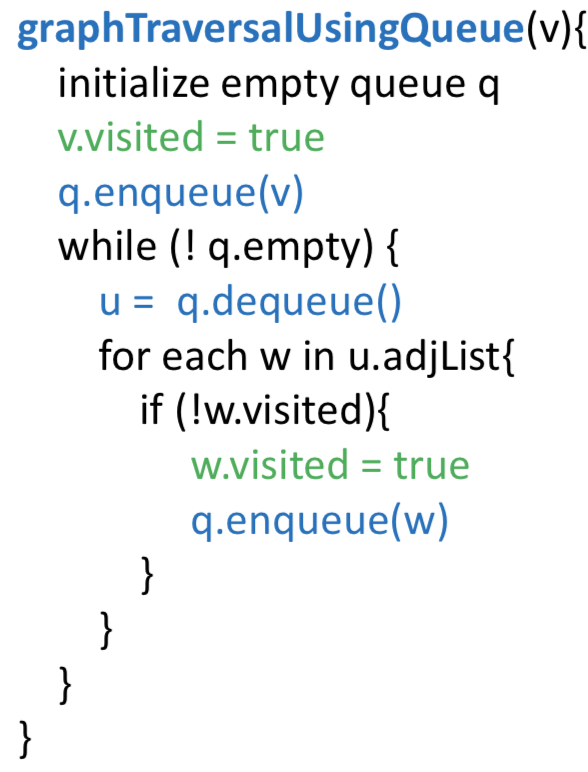


**Non-recursive graph traversal**

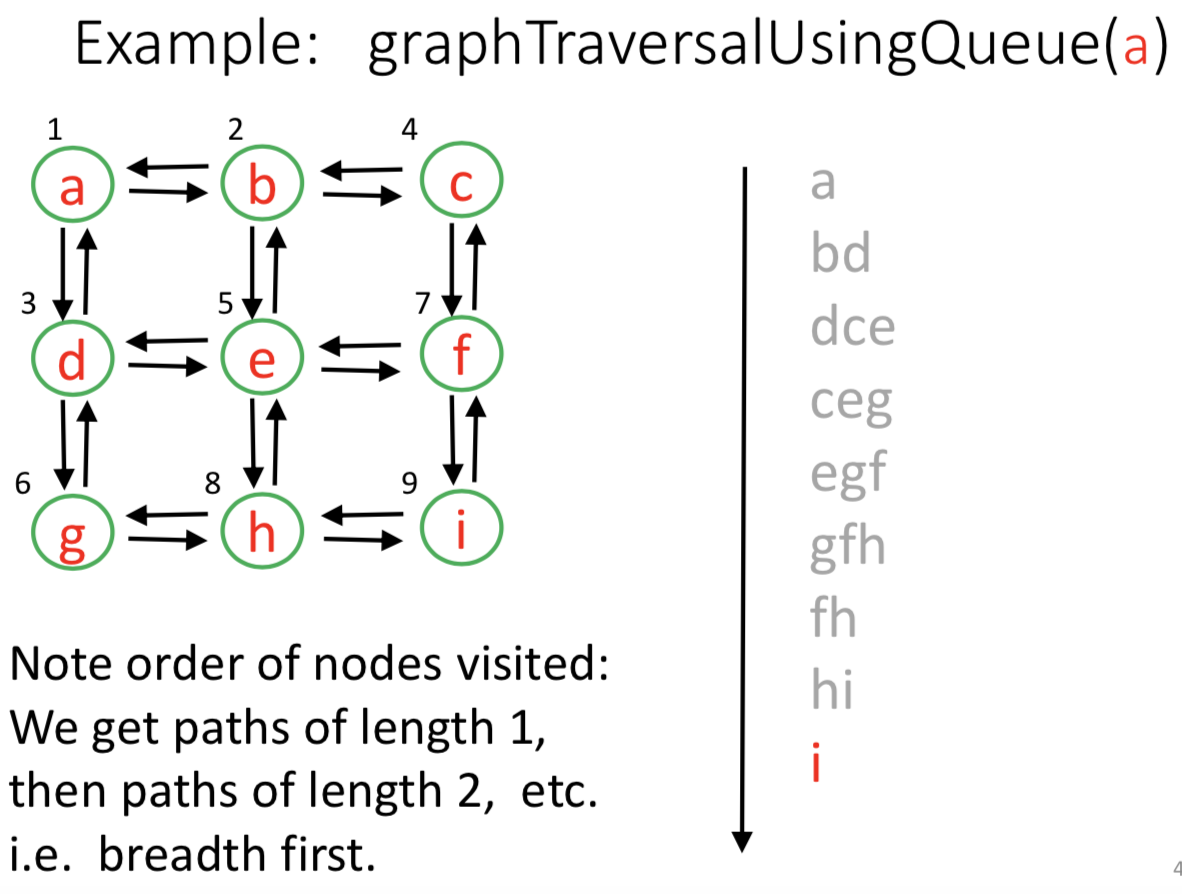
• depth first



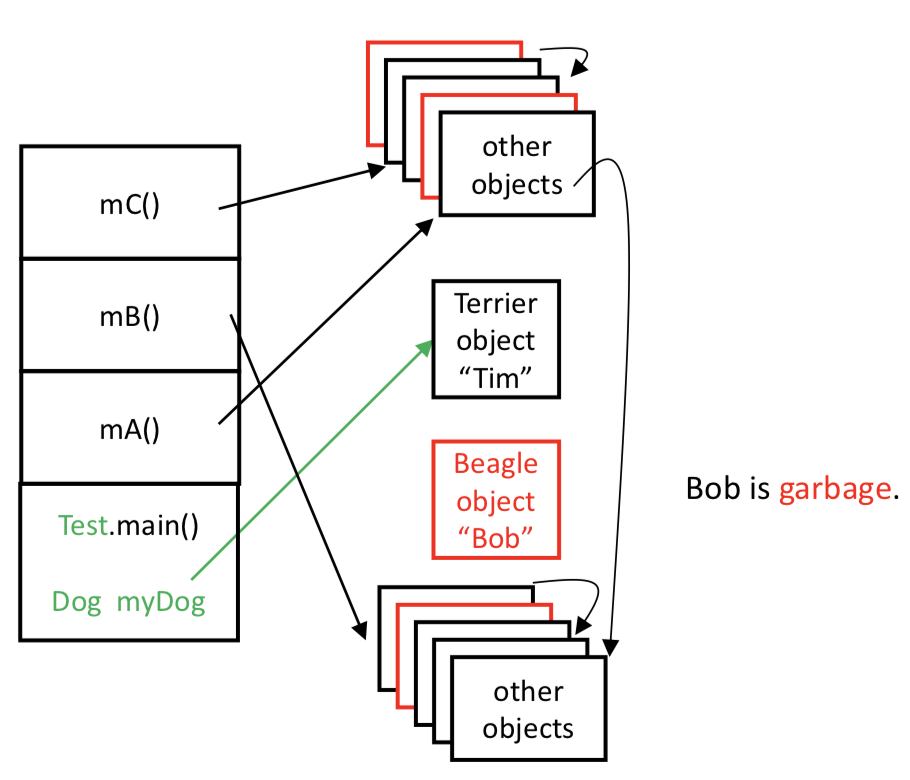
• breadth first



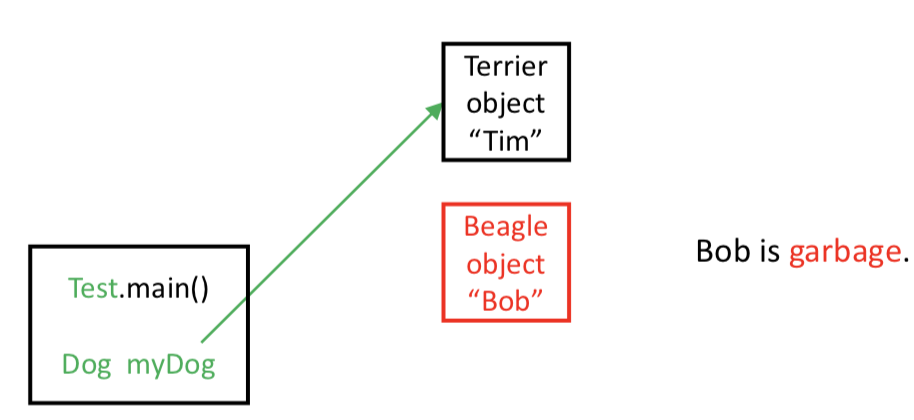
It defines a tree whose root is the starting vertex. It finds the shortest path (number of vertices) to all vertices reachable from starting vertex.



**Application-Garbage Collection**

Dog myDog = new Beagle(“Bob”);

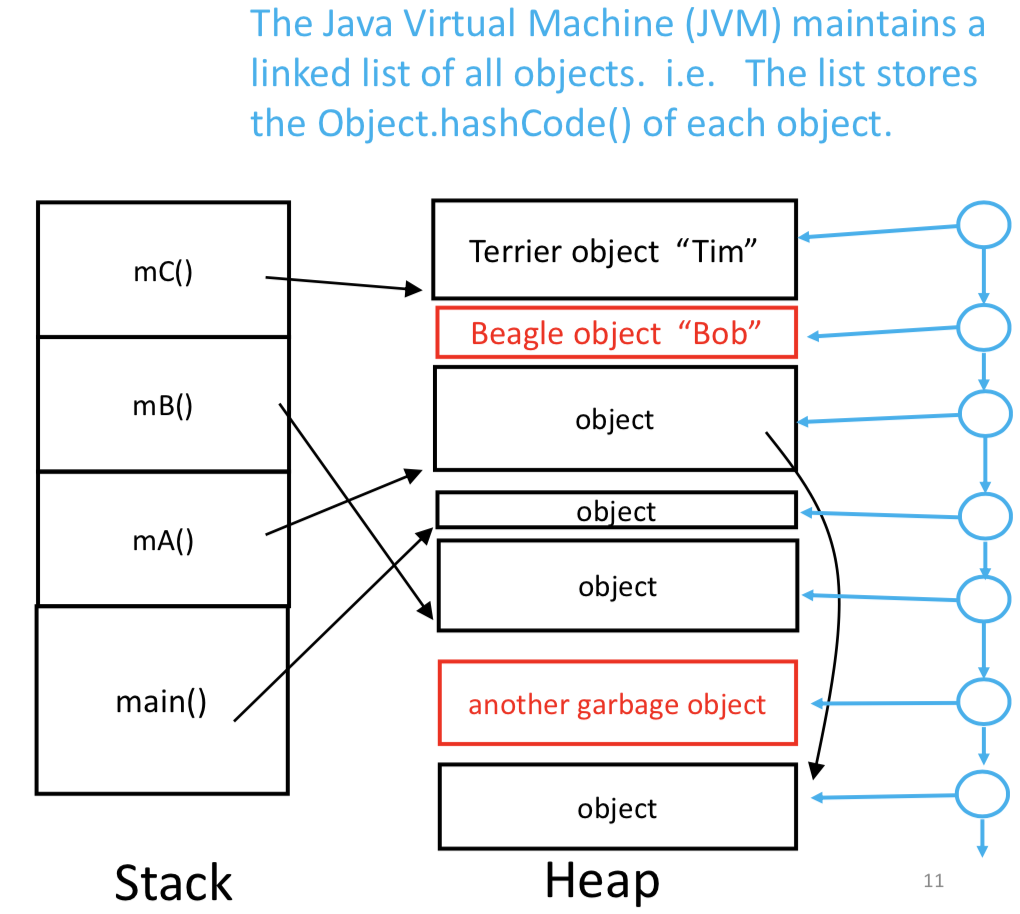
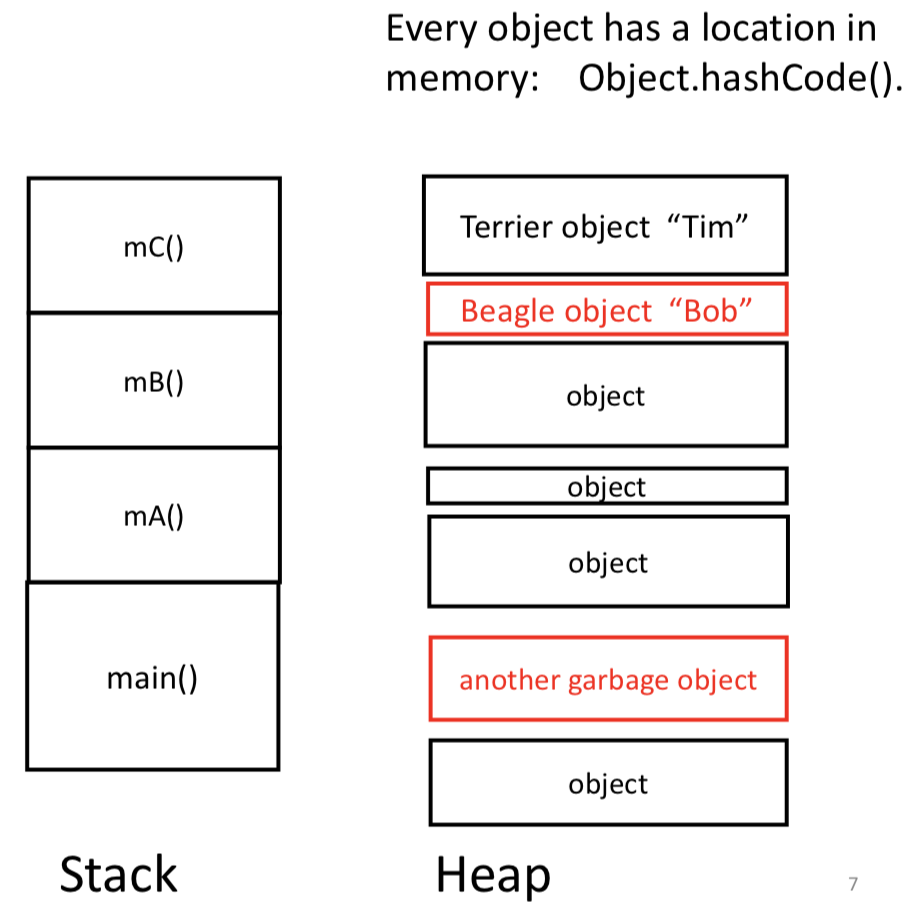
myDog = new Terrier(“Tim”);



heap

call stack

When space fills up, we reuse the space we don’t need. (Garbage collection)



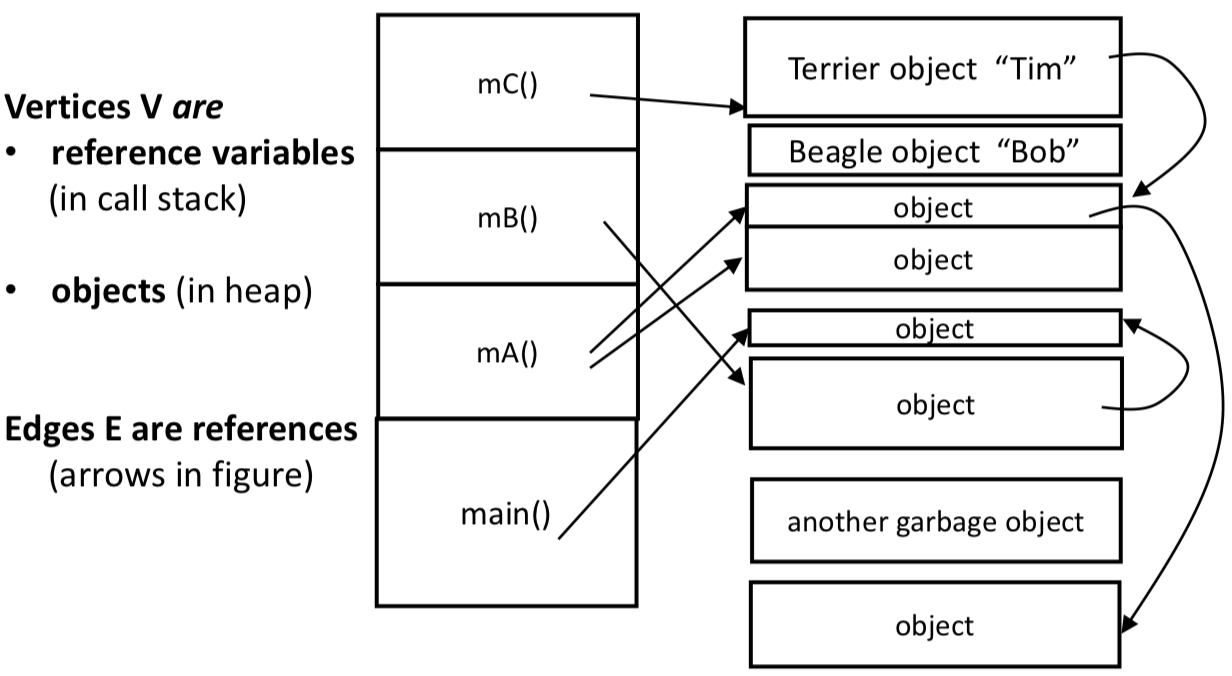
Note: If objects are only referenced by each other, they are garbage, because they will never be used by the program.

**Garbage collection: “Mark and Sweep”**

*1) Build a graph, and identify live objects (“Mark”)*

*2) Remove garbage (“Sweep”)*

build a graph



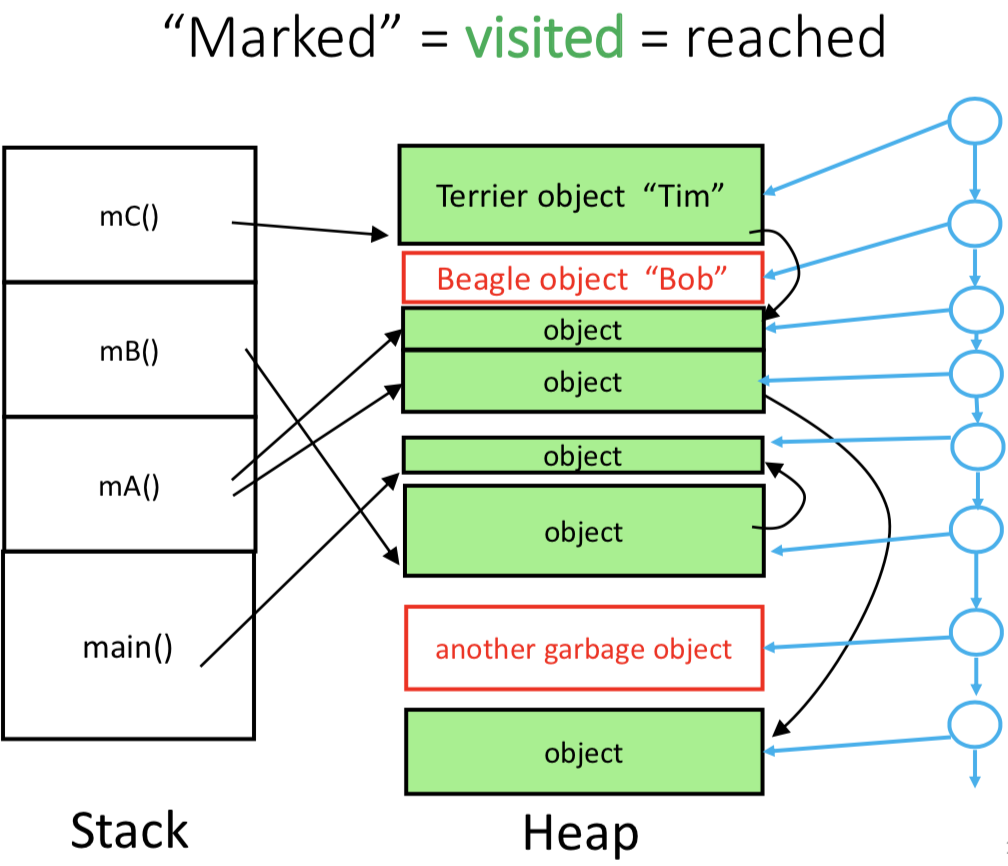
stack

heap

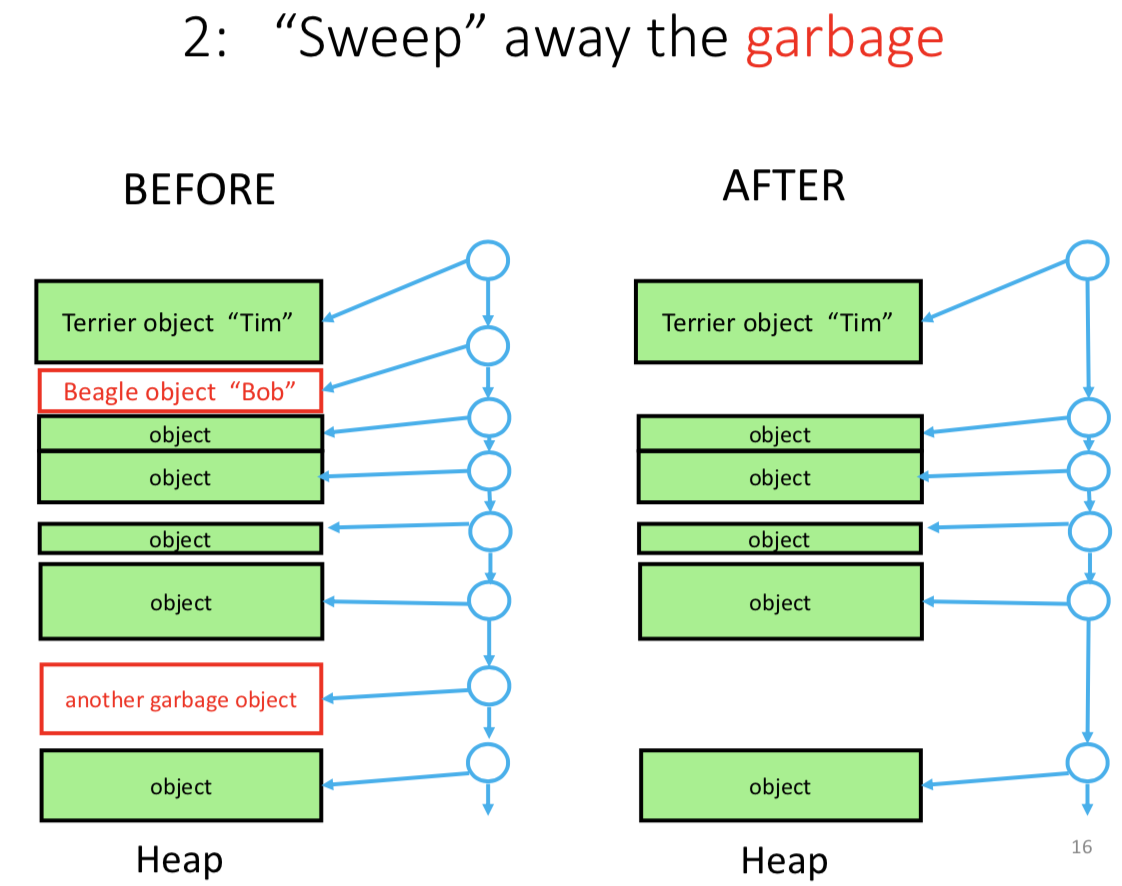
mark

For each reference variable on the call stack {

• traverse the graph, starting from the object that is referenced by that variable, and mark each object that is reached as visited

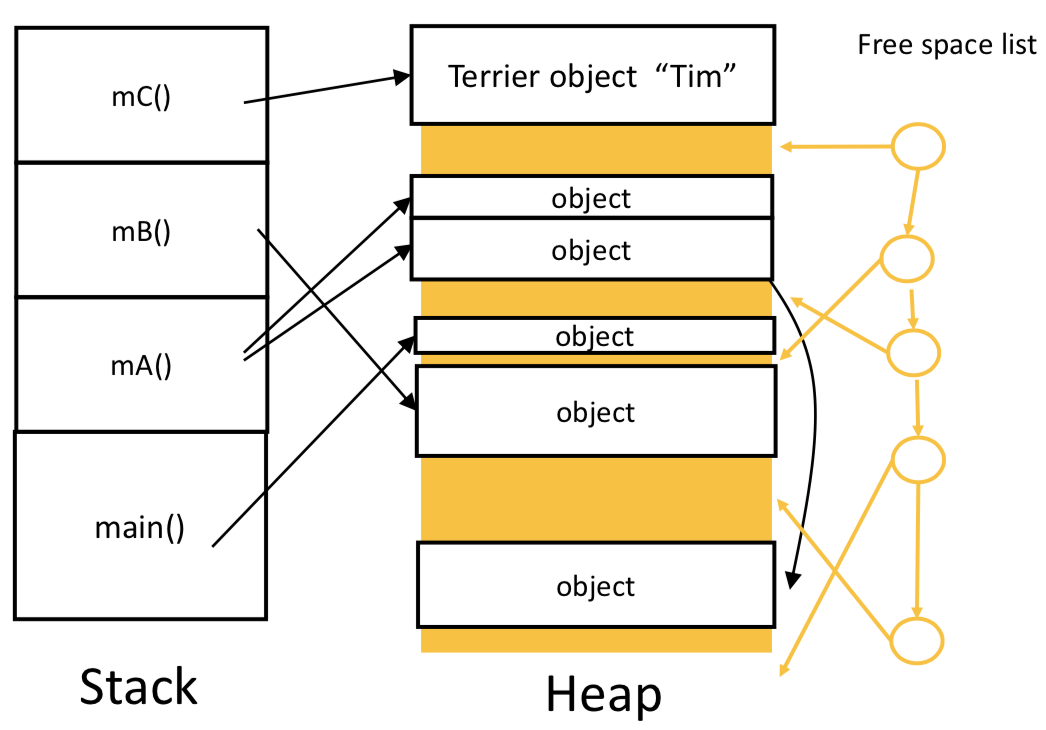


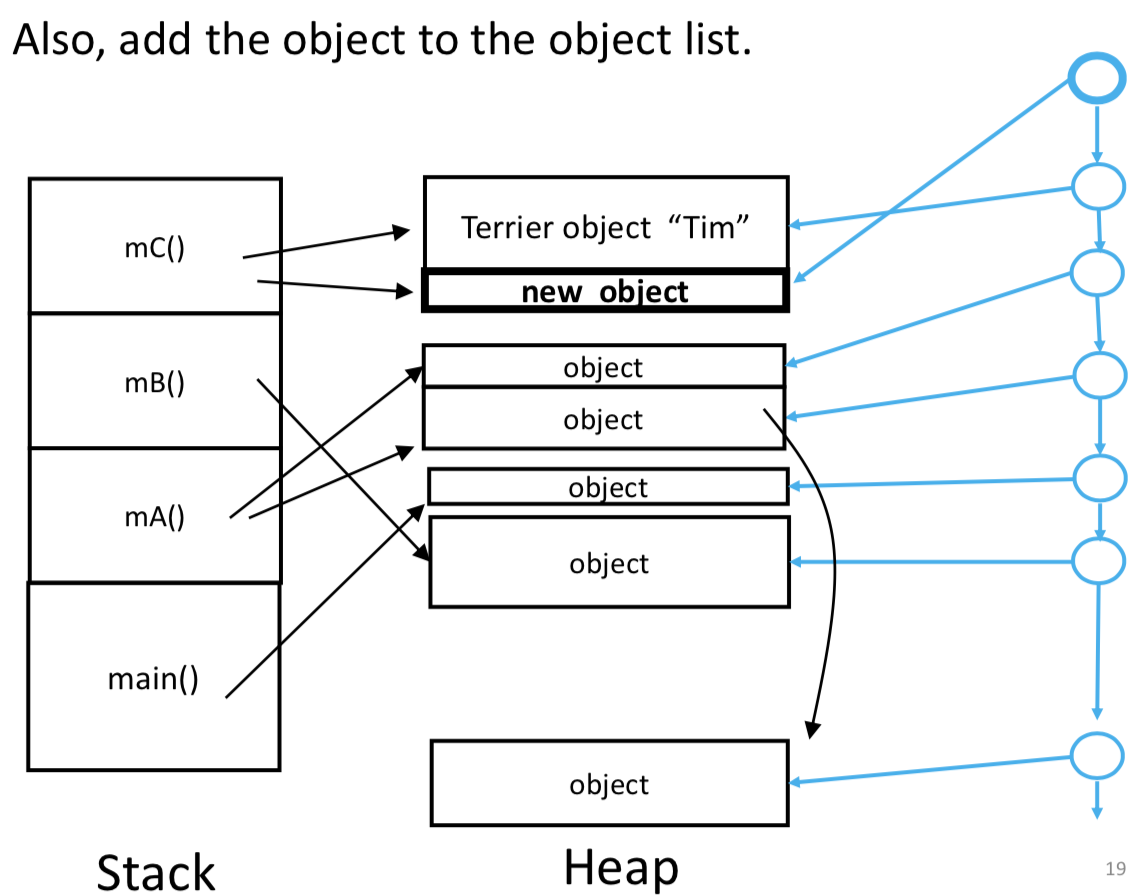
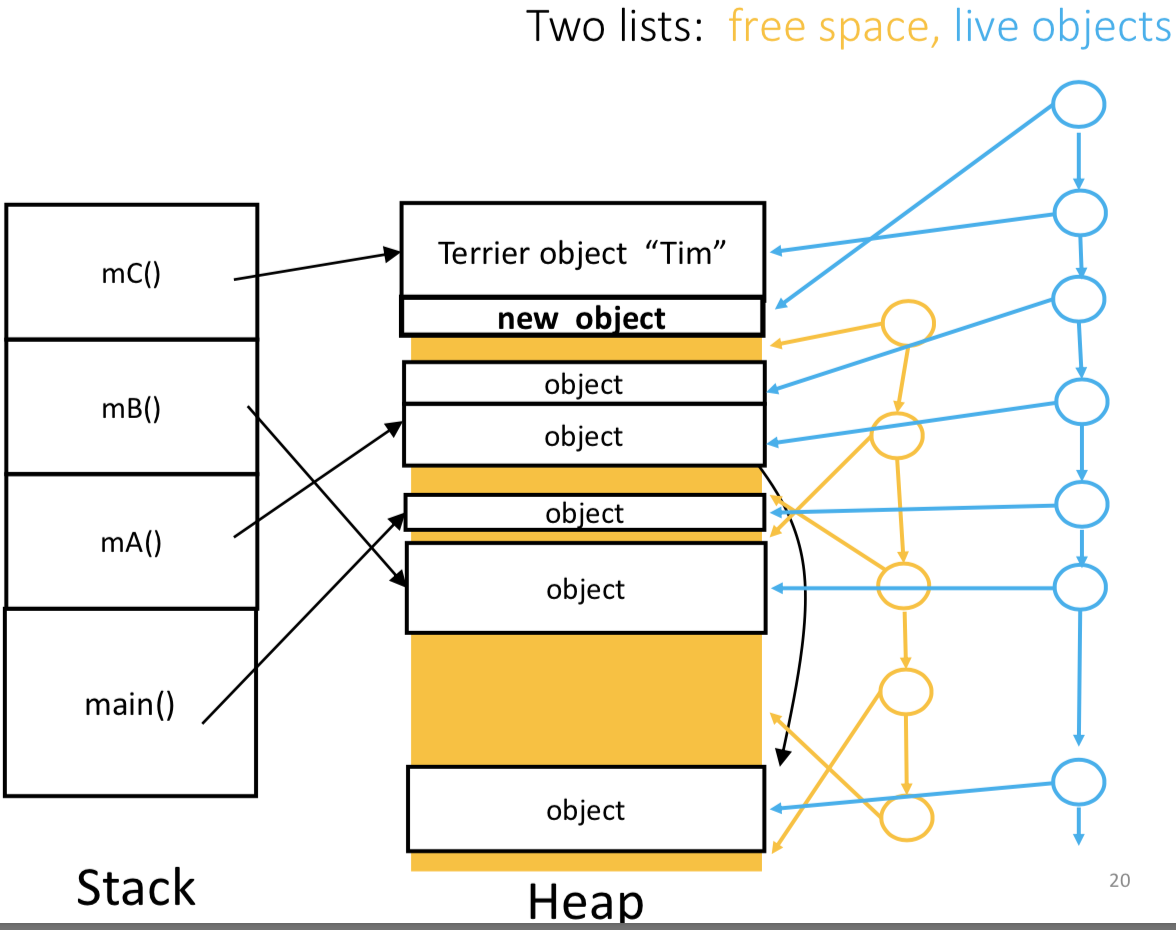
sweep away the garbage



When a new project is created, we can find a place for it. We may need to modify the free space list

Use another list to keep track of free space between objects.





DO NOT NEED TO BE IN ORDER

After garbage collection, continue execution..

* + New objects can be added, where there is a big enough gap in free space.
  + Garbage collection is needed again when there is no gap big enough for the new object. (Objects can also be moved around – if heap space is too fragmented.)
  + Program needs to stop (temporarily) to do garbage collection, which is not good for real time applications.

**Application: Google Page Rank**

Google builds a graph data structure:

• the vertices V are the web pages

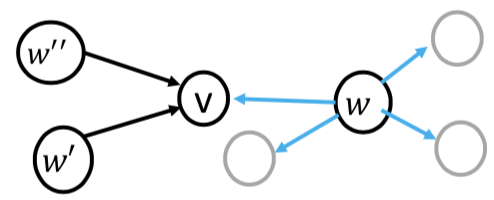
• the edges E are the hyperlinks within the web pages

• the keys K for the vertex map are the URL’s

To define the “page rank” of 𝑣 :

• Let 𝑤 be a vertex such that (𝑤, 𝑣) is an edge.

• Let 𝑁𝑜𝑢𝑡 (𝑤) be the ‘out degree’ of w.





To calculate this, we need a list of the incoming edges to each vertex, similar to an adjacency list but now we list the incoming instead of outgoing edges.

