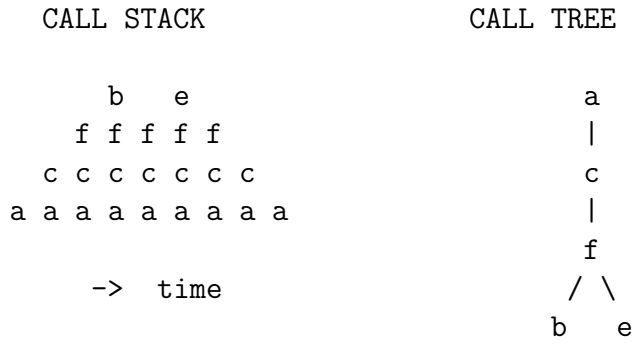
More on depth first search

Let's run the depth first search algorithm on the *directed graph* from last lecture i.e. the graph on the right which had some directed edges between nodes. A depth first search from `a` will not reach the same set of nodes in the undirected case. In particular, it will not reach node `d`.

Below is the call stack (evolving over time) for the DFS recursion, and the corresponding call tree. Note that the call stack is actually constructed (recall lecture 20 page 4) when you run a program that implements this recursive algorithm, whereas the call tree is not constructed.



Next, let's write depth first search using a non-recursive algorithm.
NOTE: This was algorithm modified on April 7. See line with ‘‘`select first w...`’’

```
DepthFirstSearch(v){
   initialize empty stack s
   s.push(v)
   v.visited = true                              //
   while (!s.empty)
     if ((w.visited == false) for some w in adjList(s.top)){
        select first w in adjList(s.top) with (w.visited == false)
        w.visited = true                         //
        s.push(w)}
     else
        s.pop()
     }
}
```

Notice how much more awkward it is, compared to the recursive method described last class. The reason is that we have to explicitly keep track of the "call stack" namely we need to keep track of where we are our search. This situation is quite common, and is one reason why recursion is so powerful.

One final point. Suppose you wanted to keep track of the order in which vertices are traversed. How could you do that? One way is to create a queue (or a second stack). You could add an instruction `q.enqueue(w)` to above code, whenever you assign a vertex (say `v` or `w`) to be visited. (There are two such occurences, marked by `//`.) Then, when the algorithm terminates, the queue would contain the vertices visited in their proper order.

Breadth first search

Like depth first search, "breadth first search" searches for all the nodes that can be reached from a given node v. The difference is that breadth first search attempts to create paths from a given node that are as short as possible. First, it visits nodes that are a distance 1 away. Then it visits nodes that are a distance 2 away, etc.

We have already seen an example of breadth first search, namely level order traversals. (Recall the tree isomorphism question in Assignment 3. A tree is just a particular kind of graph.) Here we are considering a more general sort of level order traversal, where the levels correspond to nodes that are a certain distance away (in terms of number of edges) from a given node.

The new wrinkle in the algorithm below comes from the fact that we are working with a graph rather than a tree, and so we need to make sure we don't visit a vertex more than once. We don't want to go around and around the graph in circles/cycles. That was not possible in a tree, but it is possible in a graph if you are not careful.

```
bfs(u){
  initialize empty queue q
  u.visited = true
  q.enqueue(u)
  while ( !q.empty) {
    v = dequeue()
    for each w in adjList(v)
      if !w.visited{
        w.visited = true
        enqueue(w)
      }
}
```

Notice that we enqueue a vertex only if we have not visited it, and so we cannot enqueue a vertex more than once. Moreover, all vertices that are enqueued are dequeued, since the algorithm doesn't terminate until the queue is empty.

As in the depth first search, if we wanted to keep track of the order in which vertices are traversed, we could use a second queue. For each statement of the form `v.visited = true`, we would add a second statement `q2.enqueue(u)`. Whenever we visit a vertex, we add it to this second queue. In the end, we get a queue holding nodes in the order that they are visited.

Take the same example graph as before, both directed and undirected instances. Below we show the queue `q` as it evolves over time, namely we show the queue at the end of each pass through the `while` loop. We also show the final queue `q2`.

Since this is not a recursive function, we don't have a call tree. But we can still define a tree. Each time we visit a new node, `w in adjList(v)`, we have an edge `(v, w)` where `w` has not yet been visited. This defines a branch in our tree. We can think of the **w** node as a child of the `v` node, which is why we are calling this a tree. Indeed we have a rooted tree since we are starting the tree at some initial node `u` that we are searching from.

```
  UNDIRECTED GRAPH
  ----------------
  QUEUE q     TREE
(snapshots)

  a            a
  cde         / \
  def        c   d
  ef         |
  fb         f
  b         / \
           b   e

   q2 =  acdfbe
```

```
 DIRECTED GRAPH
 ----------------
 QUEUE q    TREE
(snapshots)

 a           a
 c           |
 f           c
 be          |
 e           f
            /   \
           b     e

    q2 = acfbe
```

Another Example

Here is another example, which better illustrates the difference between DFS and BFS. We suppose the graph is undirected.

```
  GRAPH      ADJACENCY LIST       DFS                  BFS

a - b - c    a - (b,d)         a   h - i                a
|   |   |    b - (a,c,e)       |   |                   /   \
d - e - f    c - (b,f)         b   g                 b     d
|   |   |    d - (a,e,g)       |   |               /  \     \
g - h - i    e - (b,d,f,h)     c   d              c   e      g
             f - (c,e,i)       |   |              |   |      |
             g - (d,h)         f   e              f   h      i
             h - (e,g,i)       |   |
             i - (f,h)         e - d              q2 = abdcegfhi
```

Terminology

Finally, here is some basic graph terminology that you should become familiar with:

- *verticies* of an edge: if we have an edge $e = (v_1, v_2)$ then v_1 and v_2 are the *end vertices* of the edge; v_1 and v_2 are *adjacent vertices*, the edge e is *incident* on these vertices.
- *outgoing edges* from v - the set of edges of the form $(v, v') \in E$ where $v' \in V$
- *incoming edges* to v - the set of edges of the form $(v', v) \in E$ where $v' \in V$
- *in-degree of* v - the number incoming edges to v
- *out-degree of* v - the number outgoing edges from v
- *path* - a sequence of verticles $(v_1, \ldots, v_p)$ where $(v_i, v_{i+1}) \in E$
- *cycle* - a path such that the first vertex is the same as the last vertex