7.3.1 Depth-first search

In this implementation, we keep track of two lists:

- **Candidate nodes** nodes that need to be visited next;
- **Visited nodes** nodes already visited.

Both lists are passed (and updated) after every invocation of the traversal function. For example, given a starting node s and a graph g, a general function for depth-first traversal of a graph is:

```
depthFirstSearch          :: Ix a => a -> Graph a -> [a]
depthFirstSearch start g = dfs [start] []
  where
    dfs [] vis    = vis
    dfs (c:cs) vis
      | elem c vis = dfs cs vis
      | otherwise  = dfs ((adjacent g c)++cs) (vis++[c])
```

The first argument of the `dfs` function represents the list of candidate nodes and the second argument the list of visited nodes. To avoid calling the (++) operator in the second argument of the recursive call to `dfs`, visited nodes can be accumulated in reverse order then reversed just before delivering the result:

```
depthFirstSearch'          :: Ix a => a -> Graph a -> [a]
depthFirstSearch' start g = reverse (dfs [start] [])
  where
   dfs [] vis     = vis
   dfs (c:cs) vis
      | elem c vis = dfs cs vis
      | otherwise  = dfs ((adjacent g c)++cs) (c:vis)
```

When applied to the graph in Figure 7.2, the evaluation proceeds as follows:

```
depthFirstSearch' 1 g  ⇒  dfs [1] []
                       ⇒  dfs [2,3,4] [1]
                       ⇒  dfs [3,4] [2,1]
                       ⇒  dfs [6,4] [3,2,1]
                       ⇒  dfs [5,4] [6,3,2,1]
                       ⇒  dfs [4,4] [5,6,3,2,1]
                       ⇒  dfs [4] [4,5,6,3,2,1]
                       ⇒  dfs [] [4,5,6,3,2,1]
                       ⇒  reverse [4,5,6,3,2,1]
                       ⇒  [1,2,3,6,5,4]
```

In this program, adjacent nodes are added and retrieved from the front of the candidate nodes list during the search. Therefore, we can use a stack ADT (see Section 5.2) to hold candidate nodes since they are processed in a last-in-first-out (LIFO) fashion:


```
depthFirstSearch''     :: Ix a => a -> Graph a -> [a]
depthFirstSearch'' start g
                       = reverse (dfs (push start emptyStack) [])
 where
  dfs s vis
   | (stackEmpty s)    = vis
   | elem (top s) vis  = dfs (pop s) vis
   | otherwise         = let c = top s
                         in
                          dfs (foldr push (pop s) (adjacent g c))
                              (c:vis)
```

7.3.2 Breadth-first search

The function that implements breadth-first traversal is identical to the depth-first function except that this time, a queue ADT (see Section 5.3) is used to hold the candidate nodes. This function is defined as:

```
breadthFirstSearch :: Ix a => a -> Graph a -> [a]
breadthFirstSearch start g
                   = reverse (bfs (enqueue start emptyQueue) [])
 where
  bfs q vis
   | (queueEmpty q)= vis
   | elem (front q) vis
                   = bfs (dequeue q) vis
   | otherwise     = let c = front q
                     in
                      bfs (foldr enqueue
                                 (dequeue q)
                                 (adjacent g c))
                          (c:vis)
```

In the previous example, it generates the following reduction sequence:

```
breadthFirstSearch 1 g  ⇒  bfs [1] []
                        ⇒  bfs [2,3,4] [1]
                        ⇒  bfs [3,4] [2,1]
                        ⇒  bfs [4,6] [3,2,1]
                        ⇒  bfs [6] [4,3,2,1]
                        ⇒  bfs [5] [6,4,3,2,1]
                        ⇒  bfs [4] [5,6,4,3,2,1]
                        ⇒  bfs [] [5,6,4,3,2,1]
                        ⇒  reverse [5,6,4,3,2,1]
                        ⇒  [1,2,3,4,6,5]
```