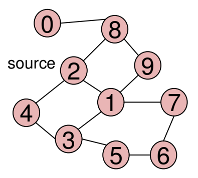
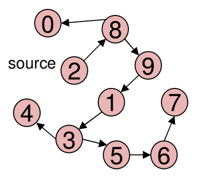
Depth-First Search

1. From the given vertex, visit one of its adjacent vertices and leave others;
2. Then visit one of the adjacent vertices of the previous vertex;
3. Continue the process, visit the graph as deep as possible until:
   - A visited vertex is reached;
   - An end vertex is reached.

Depth-First Traversal

1. Depth-first traversal of a graph:
2. Start the traversal from an arbitrary vertex;
3. Apply depth-first search;
4. When the search terminates, backtrack to the previous vertex of the finishing point,
5. Repeat depth-first search on other adjacent vertices, then backtrack to one level up.
6. Continue the process until all the vertices that are reachable from the starting vertex are visited.
7. Repeat above processes until all vertices are visited.





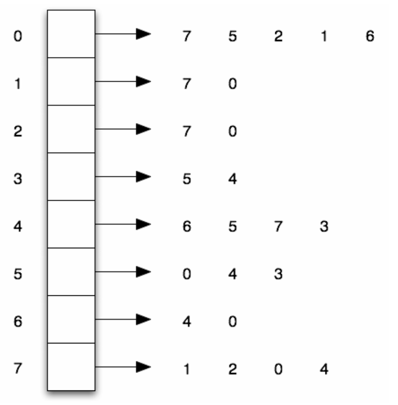
Algorithm

The pseudocode of depth-first traversal algorithm:

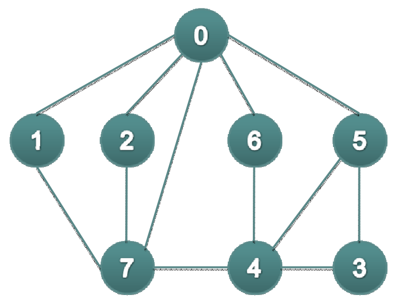
```
Boolean visited[V.size];
void DepthFirst(Graph G) {
     Vertex u;
     for each vertex u in V
        do visited[u] = false;
     for each vertex u in V
        do if visited[u] = false
              then RDFS(u);
}
void RDFS(Vertex u){
     visited[u] = true;
     Visit(u);
     for each vertex w in Adj[u]
        do if visited[w] = false
              then RDFS(w);
}
```

Example: Depth-First Traversal

An adjacent list of a graph:



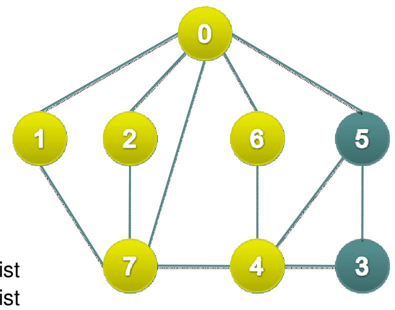
0	7	5	2	1	6
1	7	0			
2	7	0			
3	5	4			
4	6	5	7	3	
5	0	4	3		
6	4	0			
7	1	2	0	4	



Example: Depth-First Traversal

Function calls of depth-first traversal of the graph

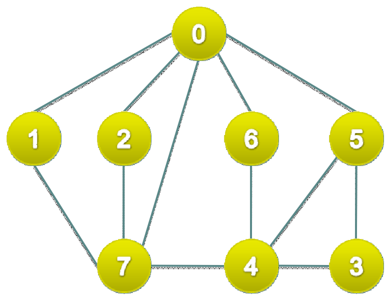
```
visit 0
   visit 7 (first on 0's list)
        visit 1 (first on 7's list)
                check 7 on 1's list
                check 0 on 1's list
        visit 2 (second on 7's list)
                check 7 on 2's list
                check 0 on 2's list
        check 0 on 7's list
        visit 4 (fourth on 7's list)
                visit 6 (first on 4's list)
                        check 4 on 6's list
                        check 0 on 6's list
```



Example: Depth-First Traversal

```
                visit 5 (second on 4's list)
                        check 0 on 5's list
                        check 4 on 5's list
                        visit 3 (third on 5's list)
                                check 5 on 3's list
                                check 4 on 3's list
                check 7 on 4's list
                check 3 on 4's list
   check 5 on 0's list
   check 2 on 0's list
   check 1 on 0's list
   check 6 on 0's list

End recursive calls
```



Using a stack

- DFS can be implemented with stack, since recursion and programming with stacks are equivalent;
- Visit a vertex v
- Push all adjacent unvisited vertices of v onto a stack
- Pop a vertex off the stack until it is unvisited
- Repeat these steps
- If the stack is empty and there is no vertex to push onto the stack, then the traversal process finishes.

Algorithm

The pseudocode of depth-first traversal algorithm:

```
DFS(G,s)
   for each vertex u in V
         do visited[u] = false
   Report(s)
   visited[s] = true

   initialize an empty stack S
   Put(S, s)

   While S is not empty
       do u = Pop(S)
            for each v in Adj[u]
                  do if visited[v] = false
                       then Report(v)
                            visited[v] = true
                            Put(S,v)
```

Quiz 2

- Continue to write a function to traverse the graph using DFS algorithm

  void DFS(Graph* graph, int s, int (*func)(int));

  // func is a pointer to the function that process on the visited vertices

Applications

- The paths traversed by BFS or DFS form a tree (called BFS tree or DFS tree).
- BFS tree is also a shortest path tree starting from its root. i.e. Every vertex v has a path to the root s in T and the path is the shortest path of v and s in G.
- DFS is used to check a the path existence between two vertices. It can be used to determine if a graph is connected.

Path finding with DFS

```
dfs-path(v, w)
  dfs-path(v, w, empty stack)

dfs-path(v, w, S)
  push(S, v)
  for each u in Adj[v]
    if visited[u] = false and visited[w]
     = false
          dfs(u, w, S)
  if visited[w]= false
    pop(S, v)
  return S
```

Quiz 3

- Add a new functionality in your program in order to find a path between two metro stations by modifying DFS.

Cycle detecting: Colored DFS

- All nodes are initially marked white. When a node is encountered, it is marked grey
- When its descendants are completely visited, it is marked black.
- If a grey node is ever encountered, then there is a cycle.

Pseudo algorithme

```
For each vertex u in G
       Color [u] = WHITE,
       Predecessor [u] = NULL;
For each vertex u in G do
       if color [u] = white
           DFS_visit(u);

DFS_visit(u)
       color(u) = GRAY
       For each v in adj[u] do
           if color[v] = GRAY and Predecessor[u] ≠  v
                       return "cycle exists"
           if color[v] = while
               Predecessor[v] = u
               Recursively DFS_visit(v)
       color[u] = Black;
```

Quiz 4:

- Add a functionality to a metro program to check if there is a loop train.