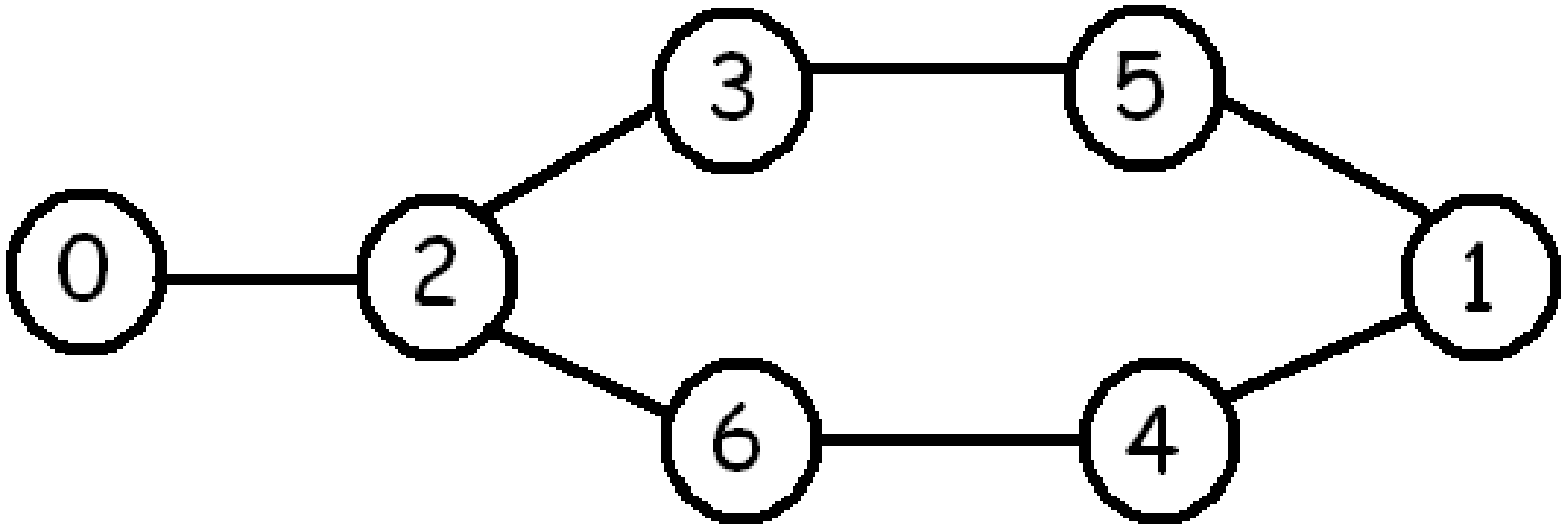
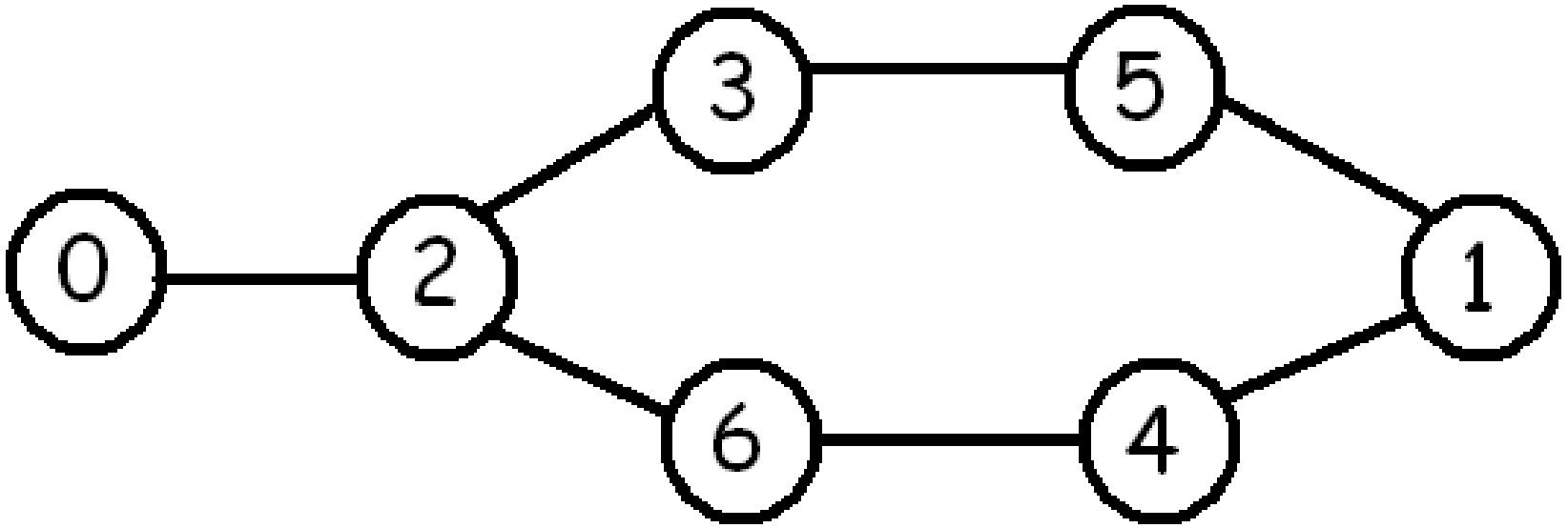
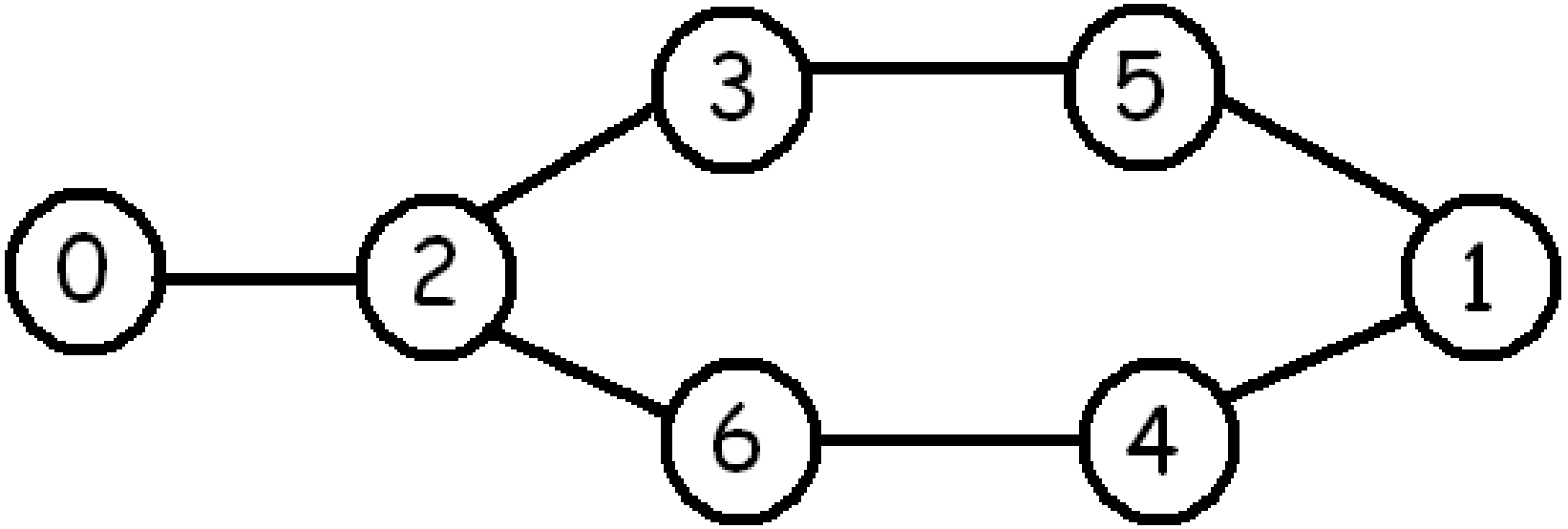
	0	1	2	3	4	5	6
Queue:							
Parent:							
BFI:							
Level:							



	0	1	2	3	4	5	6
Queue:							
Parent:							
BFI:							
Level:							

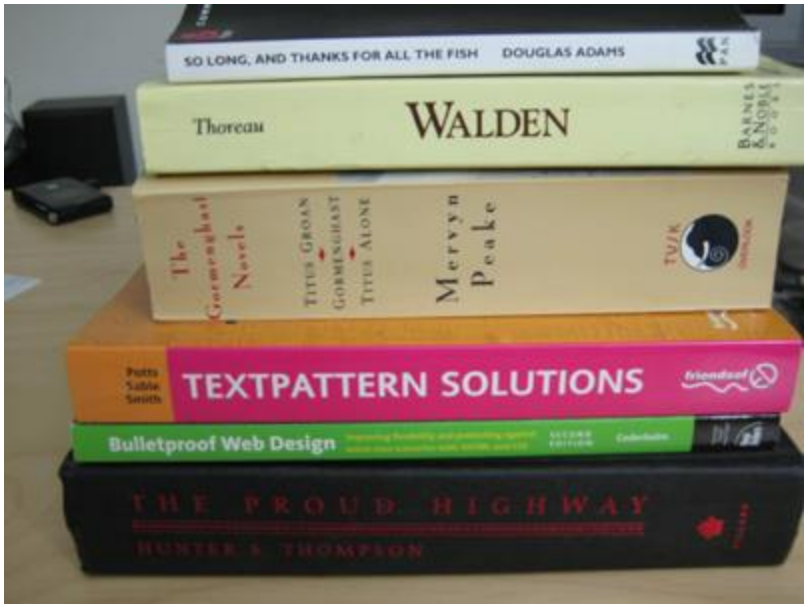




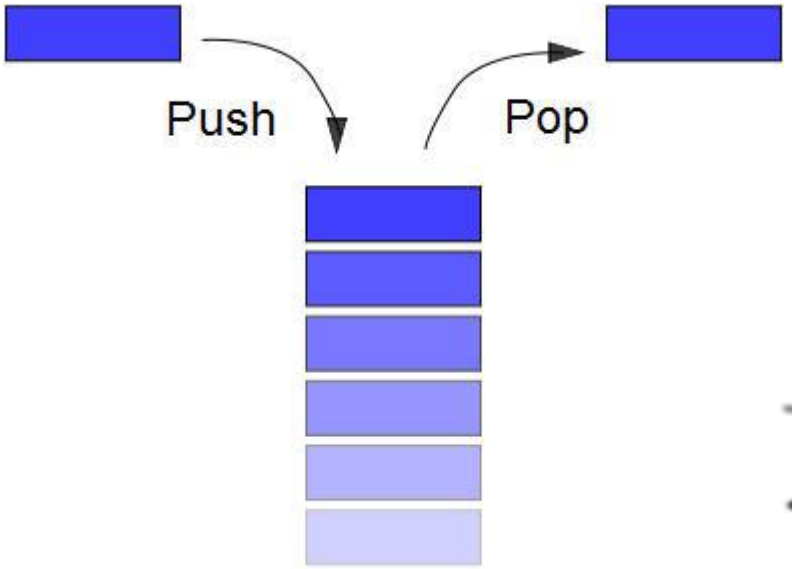
Apply BFS starting at vertex 0. Show the queue, the parent, the the BFI, and the level arrays. Process the neighbours of a vertex in numerical order.

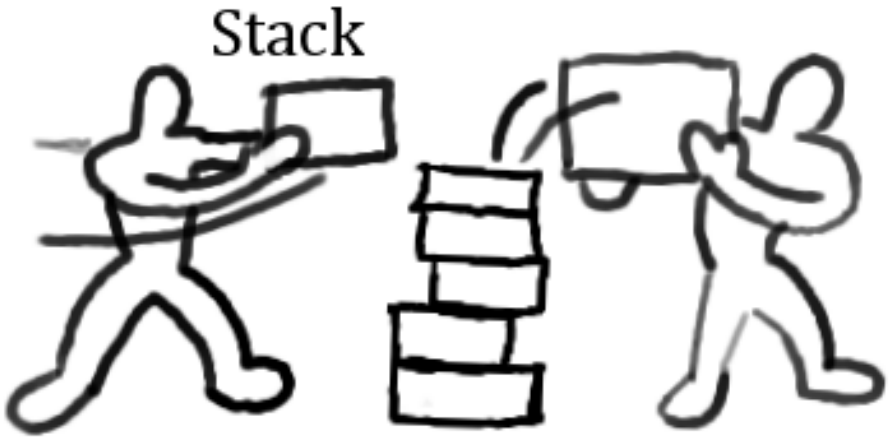
Stacks


corbis


SO LONG, AND THANKS FOR ALL THE FISH
DOUGLAS ADAMS
Thoreau
WALDEN
TEXTPATTERN SOLUTIONS
Bulletproof Web Design
THE PROUD HIGHWAY

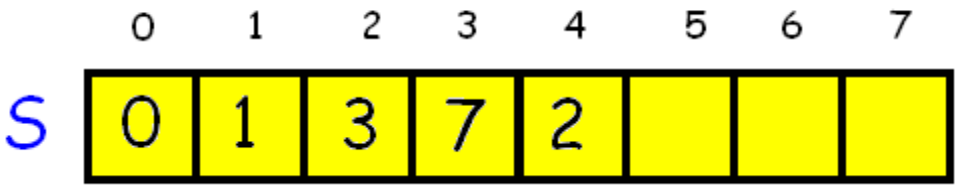
Stack Data Structure: permits push and pop at the top of the stack.





Using an array for a stack:

top=5 = # items in stack



To test if the stack is non-empty:

```
if (top > 0)
```

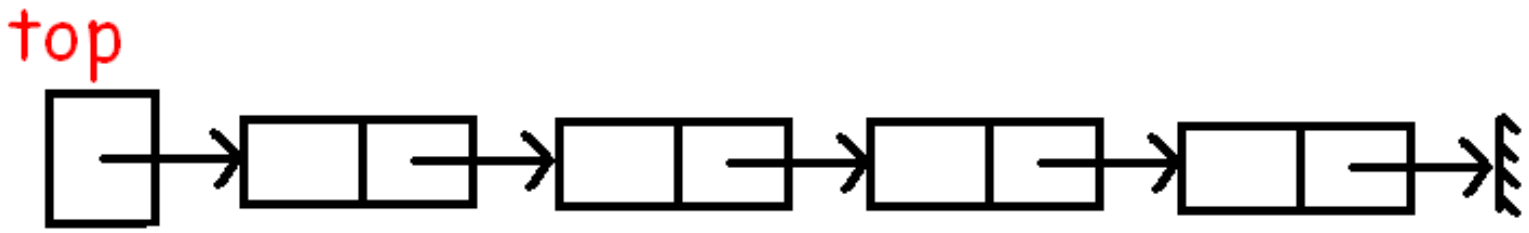
To pop x from the stack:

```
top--; x= S[top];
```

To push x onto the stack:

```
S[top]= x; top++;
```

Using an linked list for a stack:



DFS (Depth First Search) uses a stack instead of a queue.

Data structures:
A stack of edges of the form (p, v) where p is the DFS parent of node v.
visited[i]= true is vertex i has been visited and false if not.
parent[i]= DFS tree parent of node i. The parent of the root s is s.

Major difference between BFS and DFS:

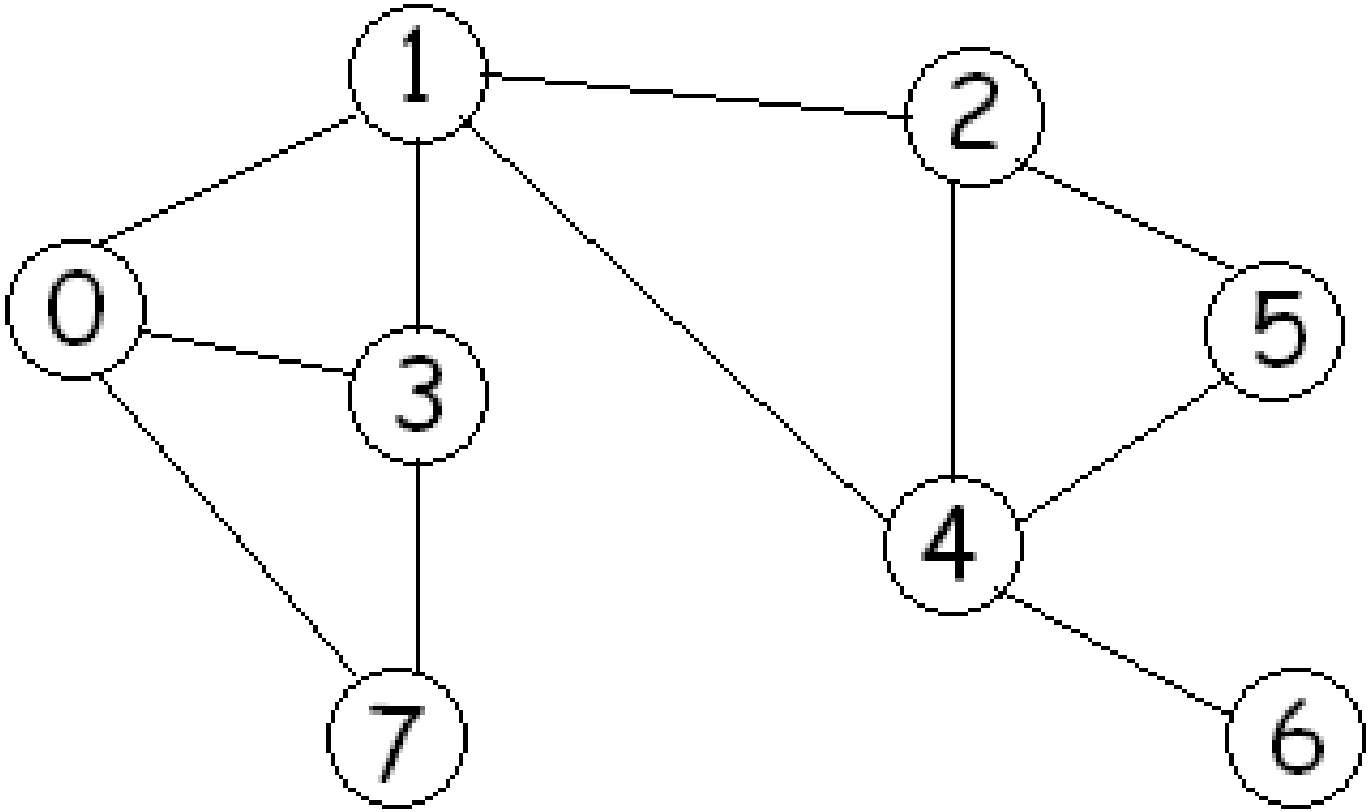
BFS: vertex is marked as visited and the parent is assigned when it is ADDED to the queue.

DFS: vertex is marked as visited and its parent is assigned when it is REMOVED from the stack.

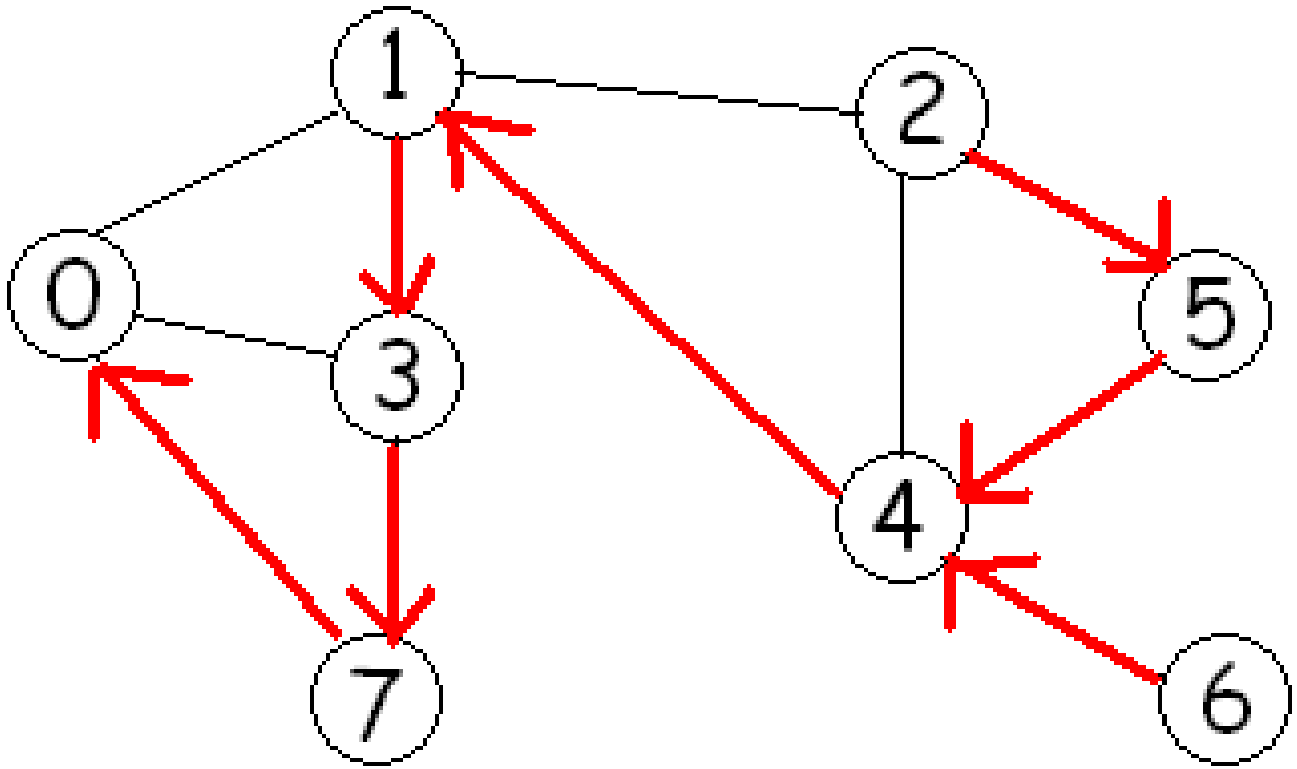
The pseudo code for DFS is:

1. Mark each vertex as unvisited.
2. Push (s, s) on the stack.
3. While the stack is not empty do

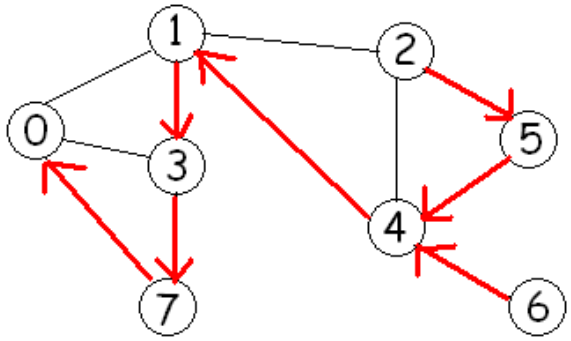
```
        Pop (p, v) from the stack.
        If v is not visited
            mark v as visited
            parent[v]= p;
            for each neighbour u of v do
                if u is not visited
                    push (v, u) on stack.
end while
```


1
2
0
5
3
4
7
6

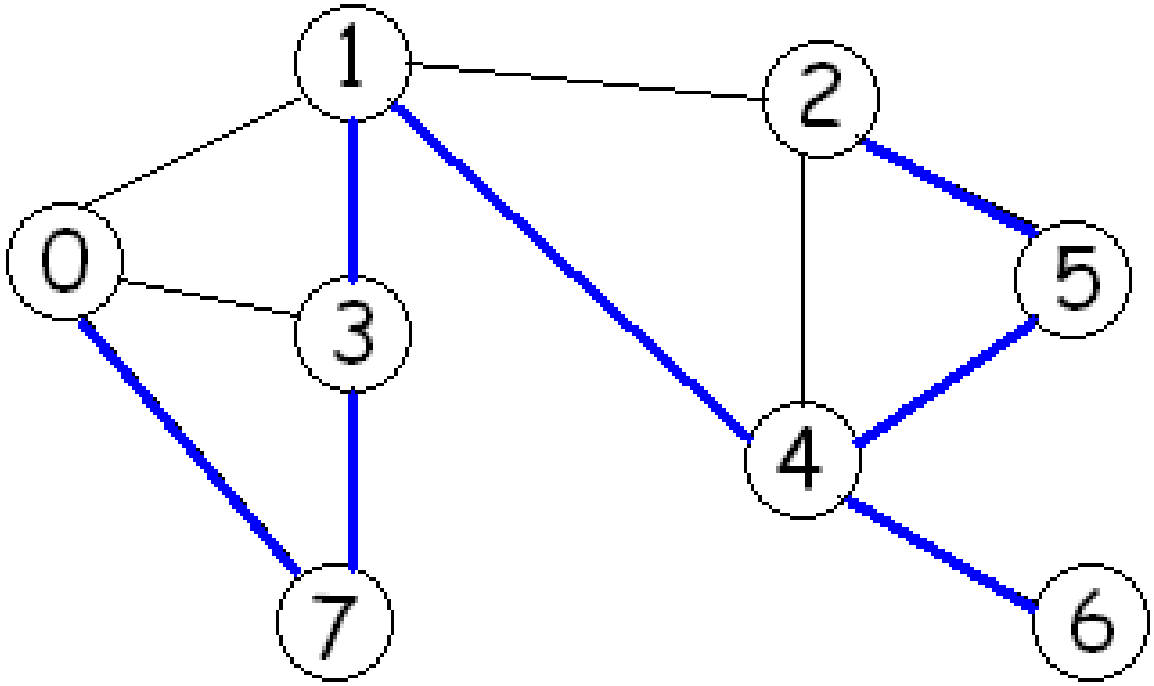
DFS parent information:



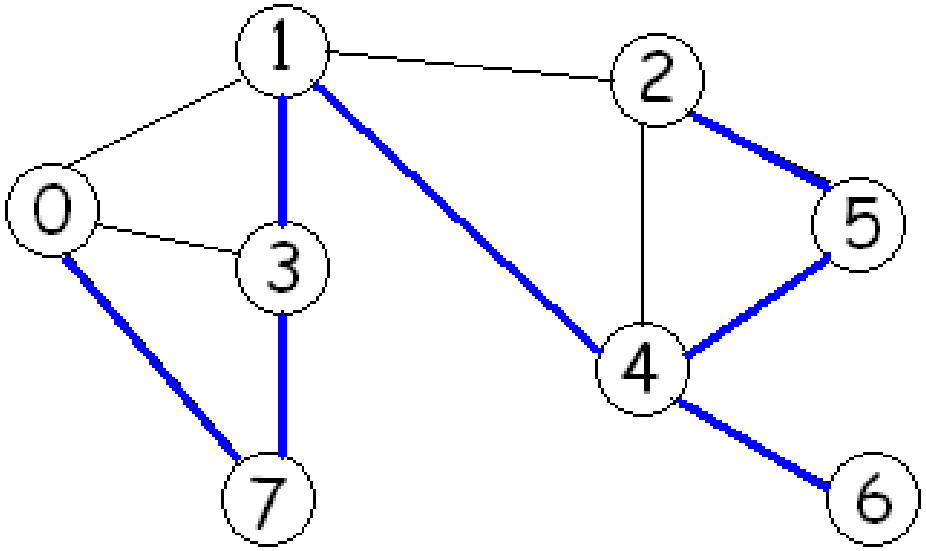
	0	1	2	3	4	5	6	7
DFI	0	3	7	2	4	6	5	1



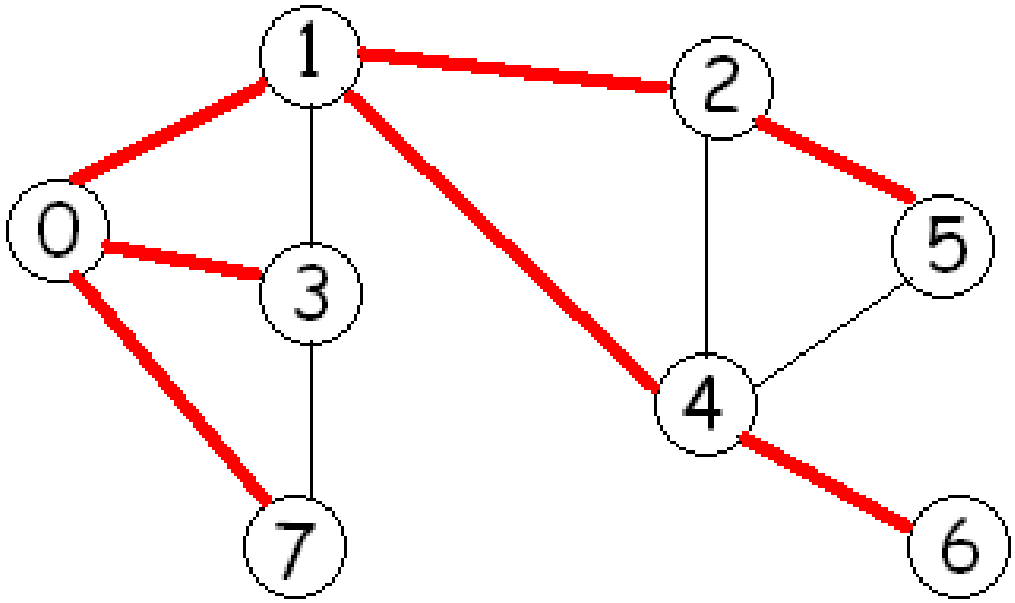
The blue spanning tree is the DFS tree:



Comparing DFS to BFS:



DFS Tree



BFS Tree

Timing analysis:

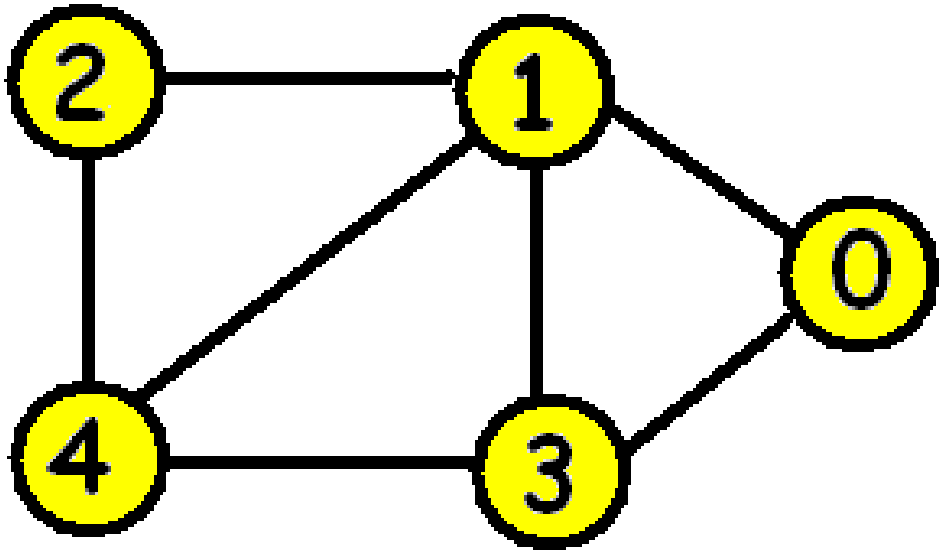
Assume the graph has n vertices and m edges.

Step 1 takes O(n) time.
Step 2 takes O(1) time.

The pseudo code for DFS is:

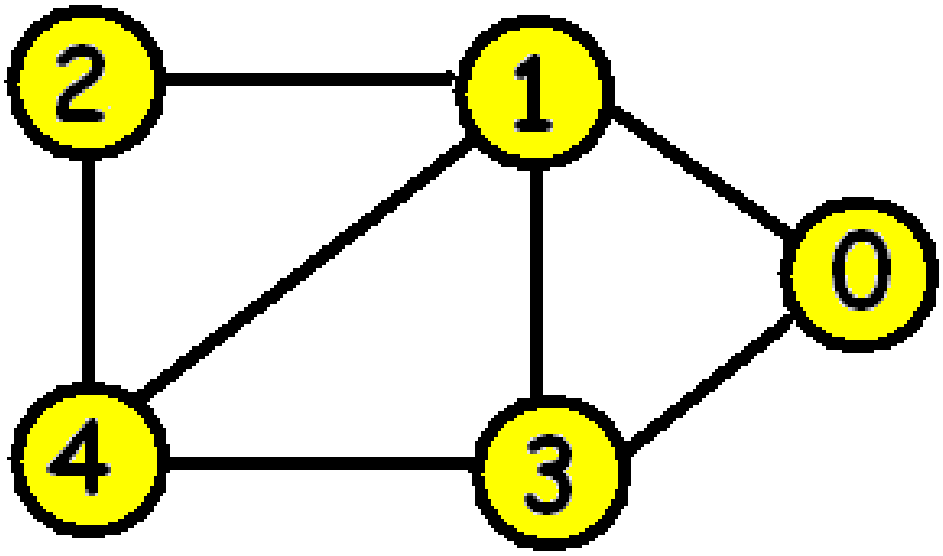
1. Mark each vertex as unvisited.
2. Push (s, s) on the stack.
3. While the stack is not empty do

```
        Pop (p, v) from the stack.
        If v is not visited
            mark v as visited
            parent[v]= p;
            for each neighbour u of v do
                if u is not visited
                    push (v, u) on stack.
end while
```

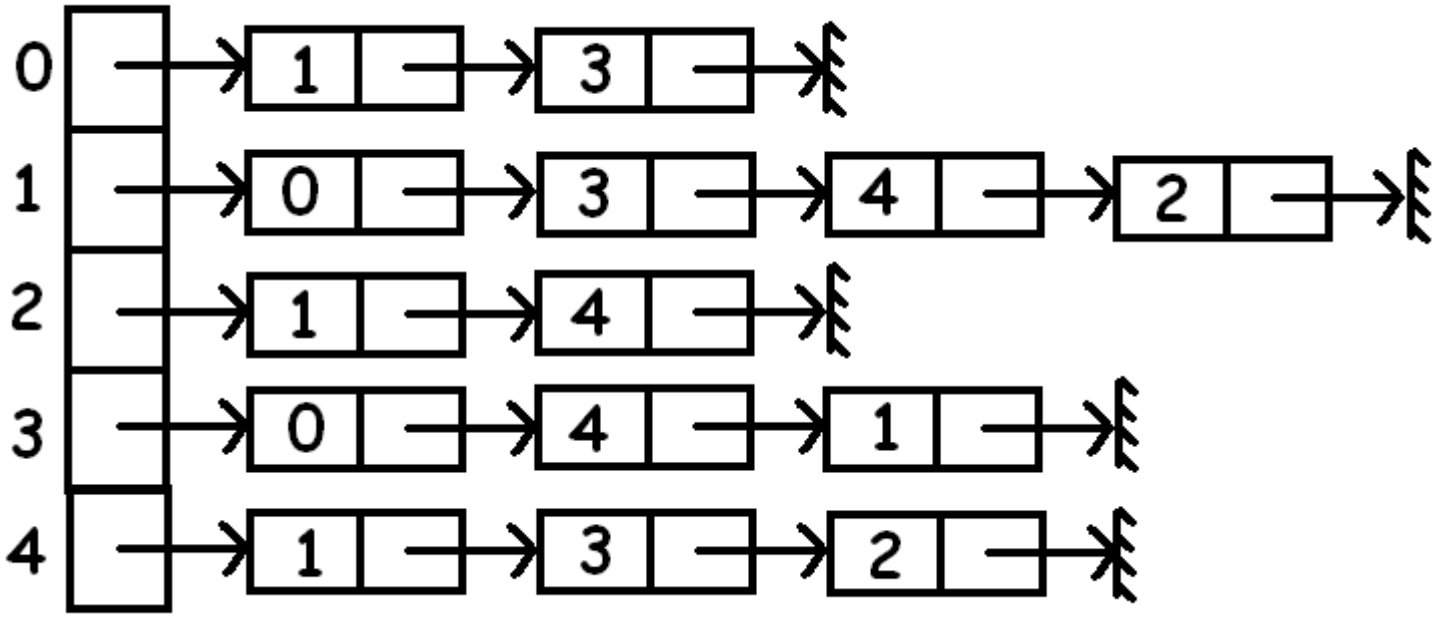


Adjacency matrix:

	0	1	2	3	4
0	0	1	0	1	0
1	1	0	1	1	1
2	0	1	0	0	1
3	1	1	0	0	1
4	0	1	1	1	0



Adjacency list:



To determine the work from step 3, observe first that each vertex is "visited" at most one time, and so we go through its neighbours one time. An arc (p, u) is pushed on the stack because we are visiting u's neighbour p. Thus, at most 2*m+1 items are pushed to the stack [(s,s) and for each edge (u, v), possibly (u, v) and (v, u)]. Thus the while loop can be entered at most 2*m+1 times.

However, the only times where the code takes more than constant time is when a vertex is visited. Each vertex is visited at most once (exactly once for a connected graph) So the total time is $O(n^2)$ with adjacency matrices or $O(m)$ with adjacency lists.

Space analysis:

A stack size of 2*m is adequate because each edge is pushed on the stack at most two times.

The entry (s,s) is pushed on the stack but then it is immediately popped so we do not have to count it in our worst case space analysis.

Can both of (u, v) and (v, u) end up on the stack?

Apply DFS starting at vertex 0.