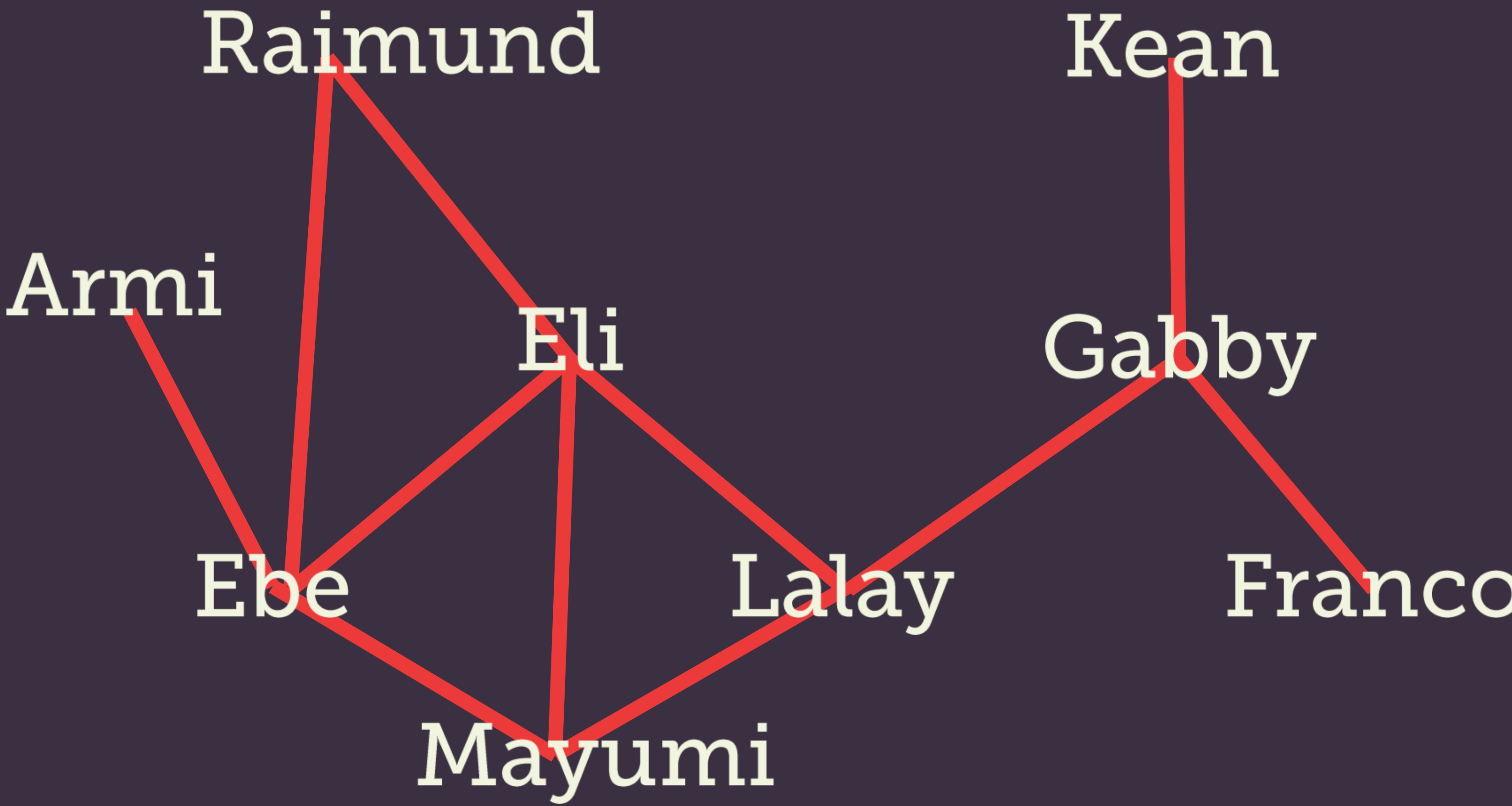
GRAPH ALGORITHMS

+ DATA STRUCTURES

GRAPH

Graph consists of a set of vertices and a set of edges.

$$G = (V, E)$$

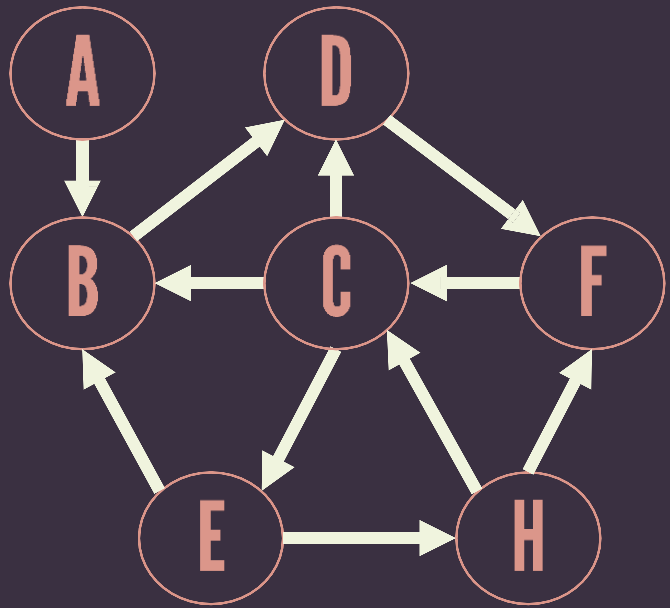

Raimund
Kean
Armi
Eli
Gabby
Ebe
Lalay
Franco
Mayumi

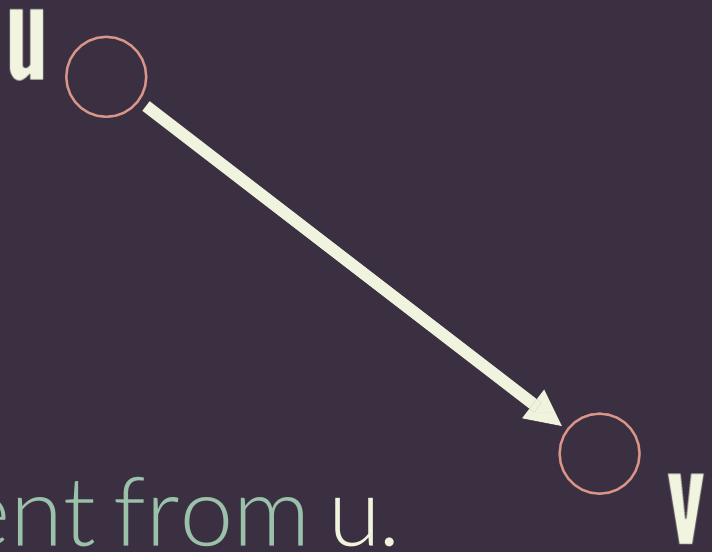
types of

GRAPHS

directed

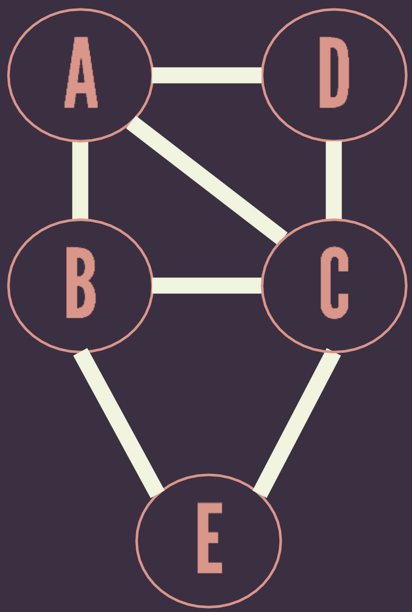
GRAPH

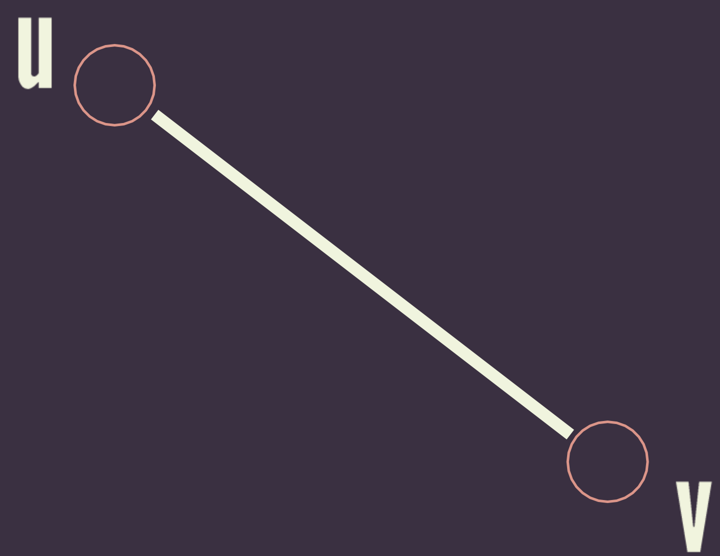




Edge (u,v) is incident from u.
Edge (u,v) is incident to v.
Vertex v is adjacent to vertex u.

undirected
GRAPH

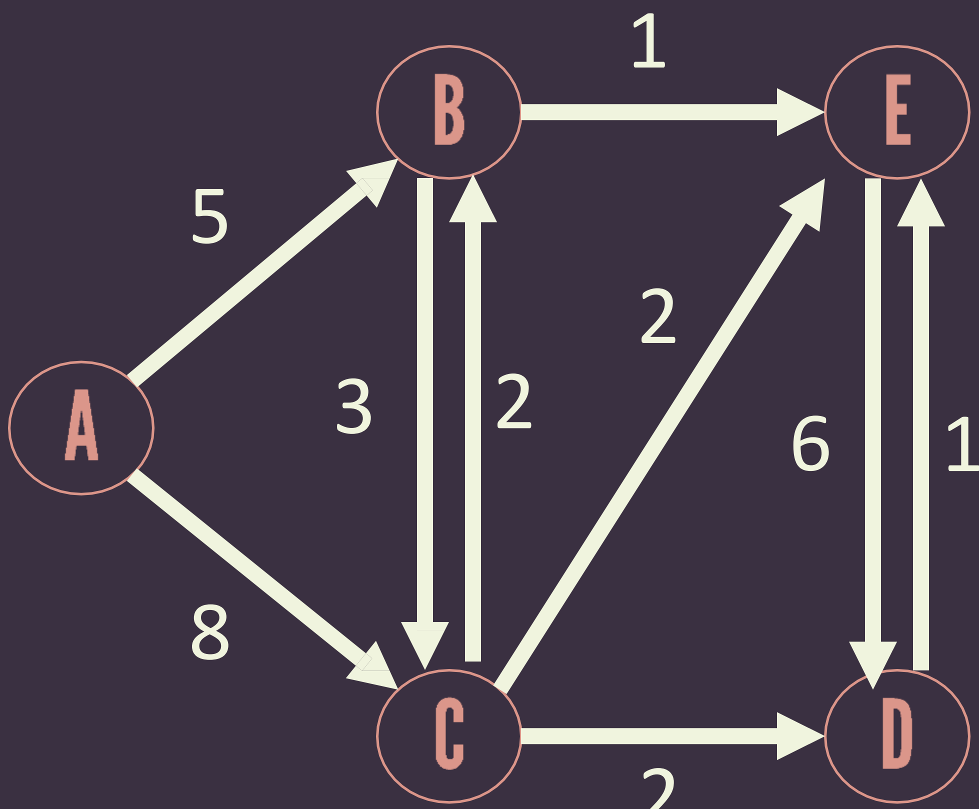




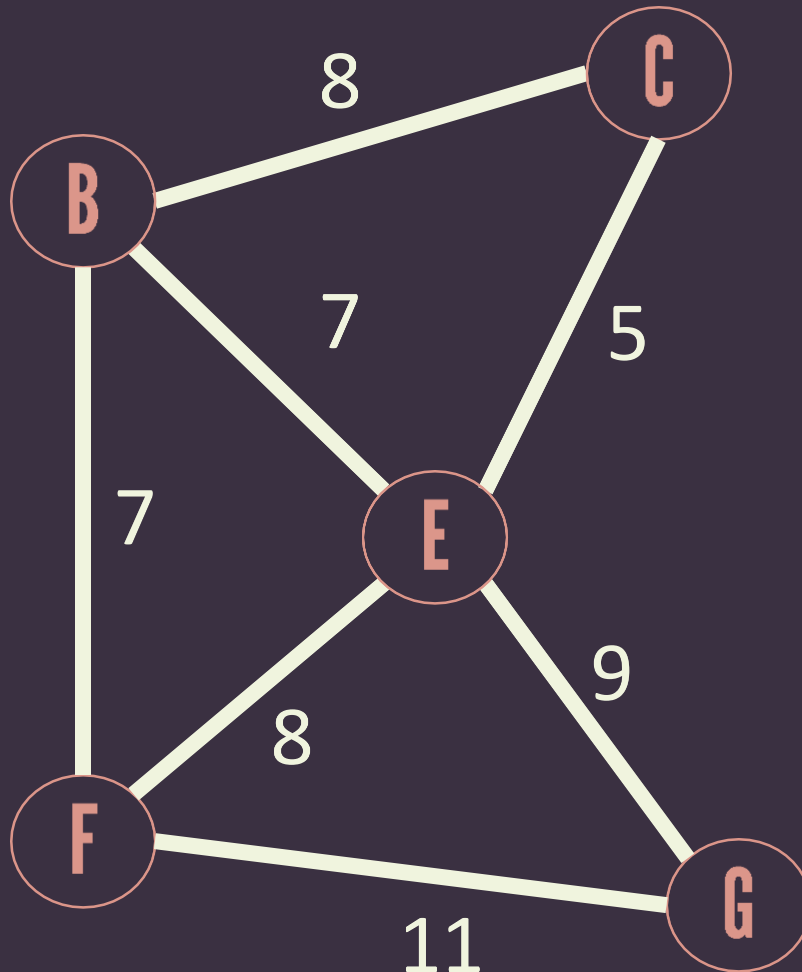
Edge (u,v) or (v,u) is incident on u and v.
Vertex v is adjacent to vertex u.
Vertex u is adjacent to vertex v.

weighted

GRAPH



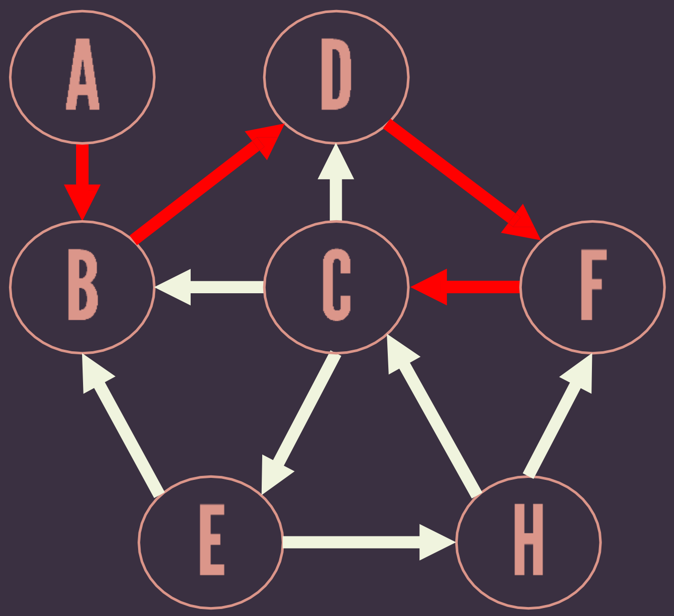
weighted
GRAPH



path in a GRAPH

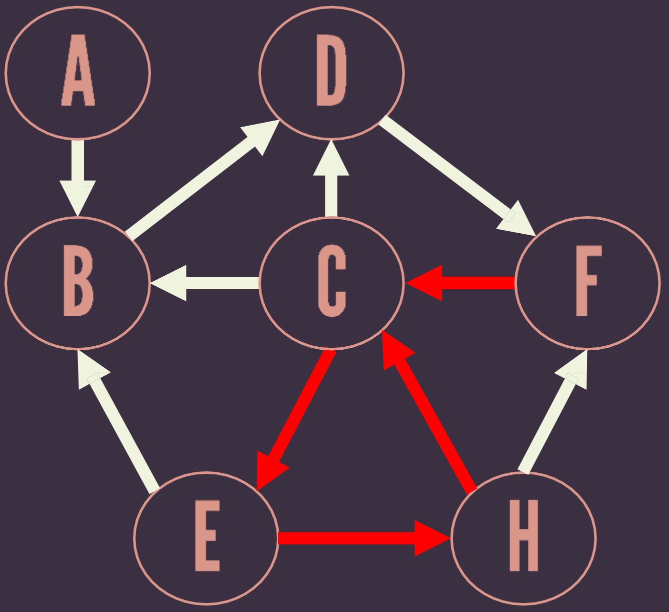
Path represents a sequence of edges between the two vertices.

path in a
GRAPH



path in a

GRAPH



in-degree, $\rho^+(v)$

# of edges incident to v

out-degree, $\rho^-(v)$

# of edges incident from v

degree of vertex v, $\rho(v)$

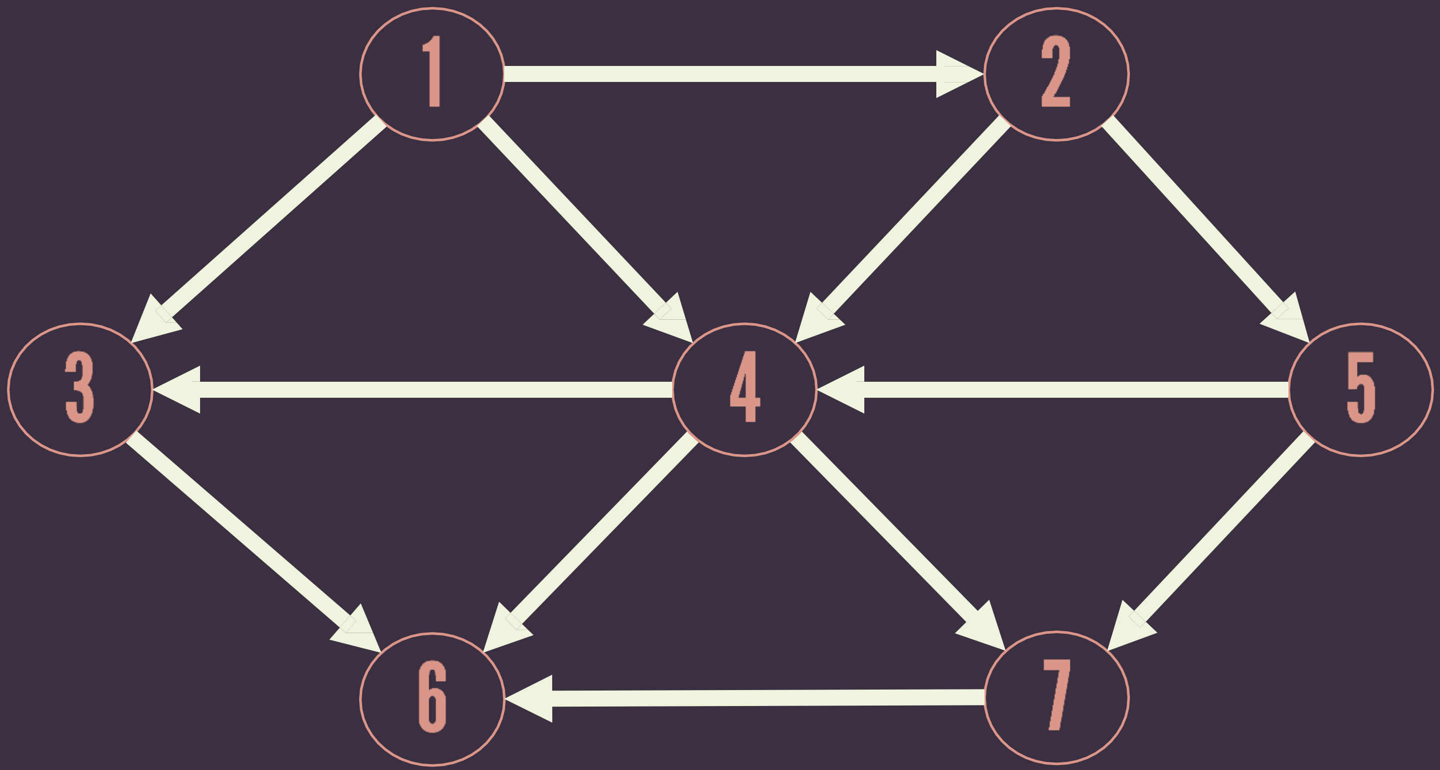
# of edges incident on v

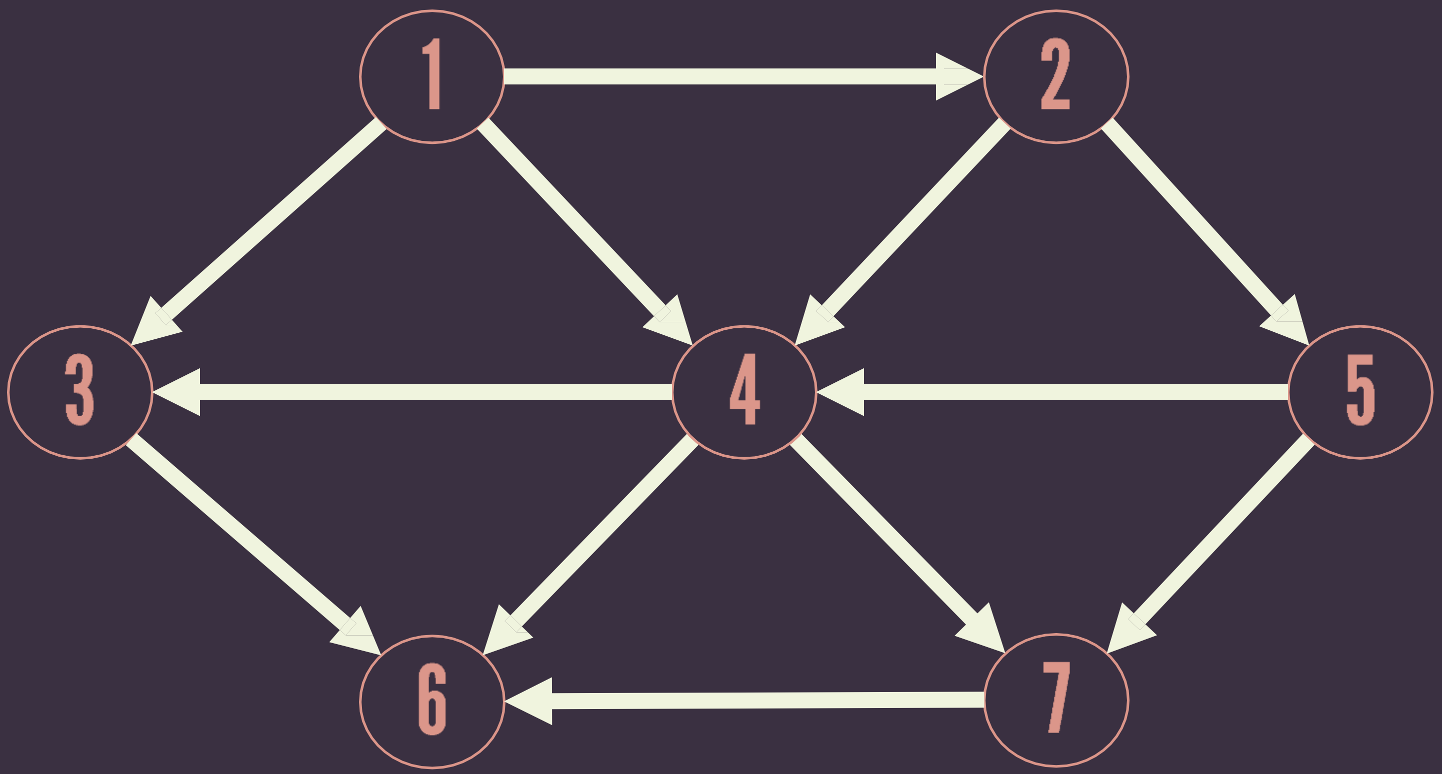
$= \rho^{-}(v) + \rho^{+}(v)$

degree of vertex v, $\rho(v)$

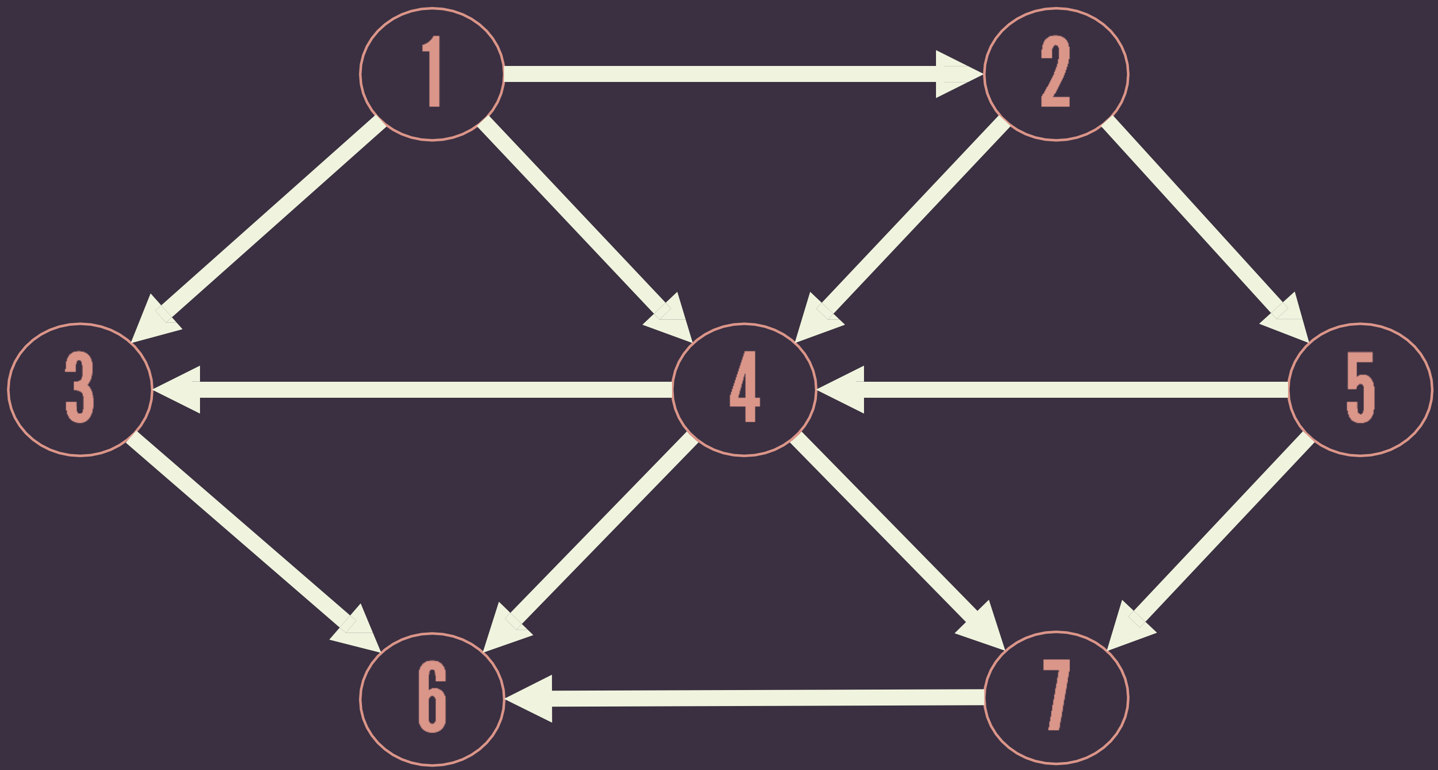
of edges incident on v.

(also applicable for undirected graphs)

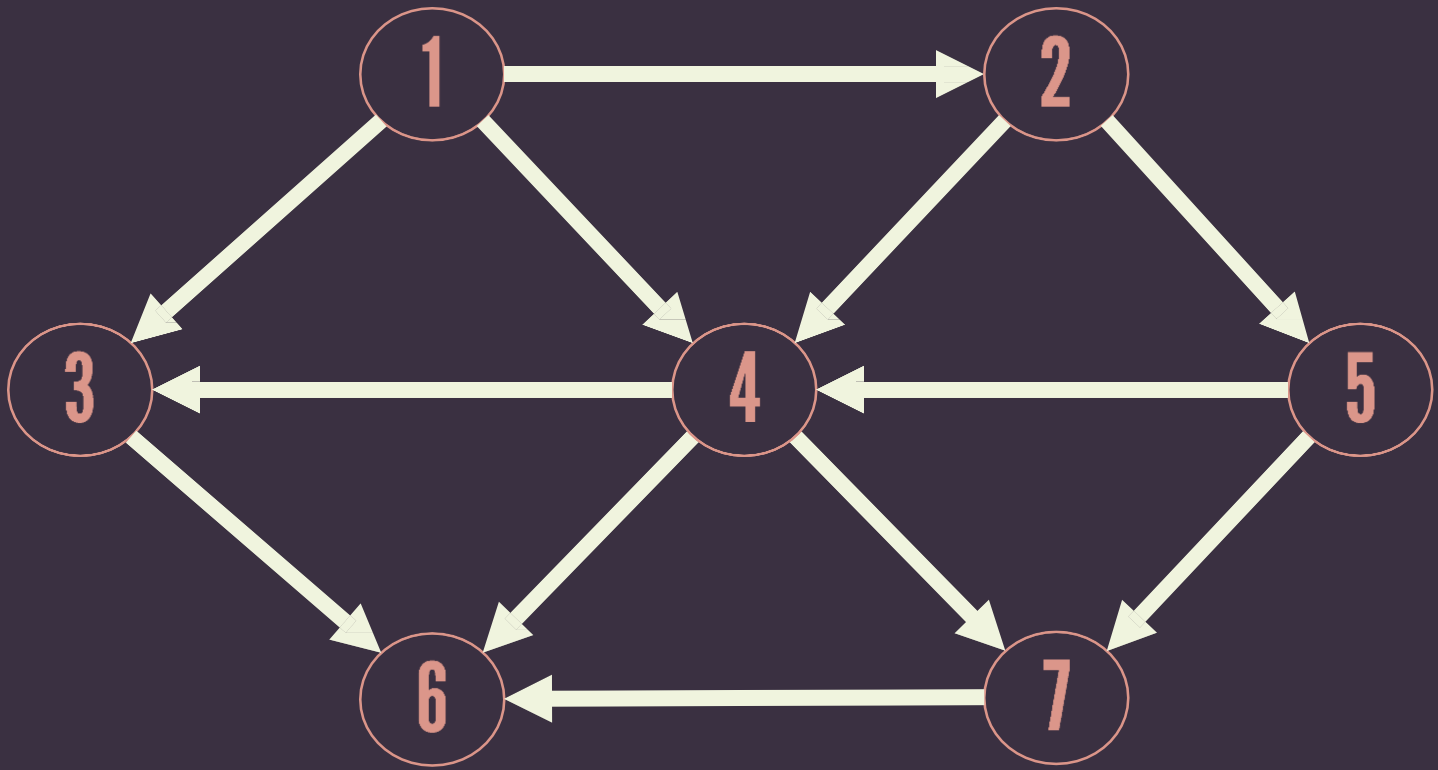

1
2
3
4
5
6
7



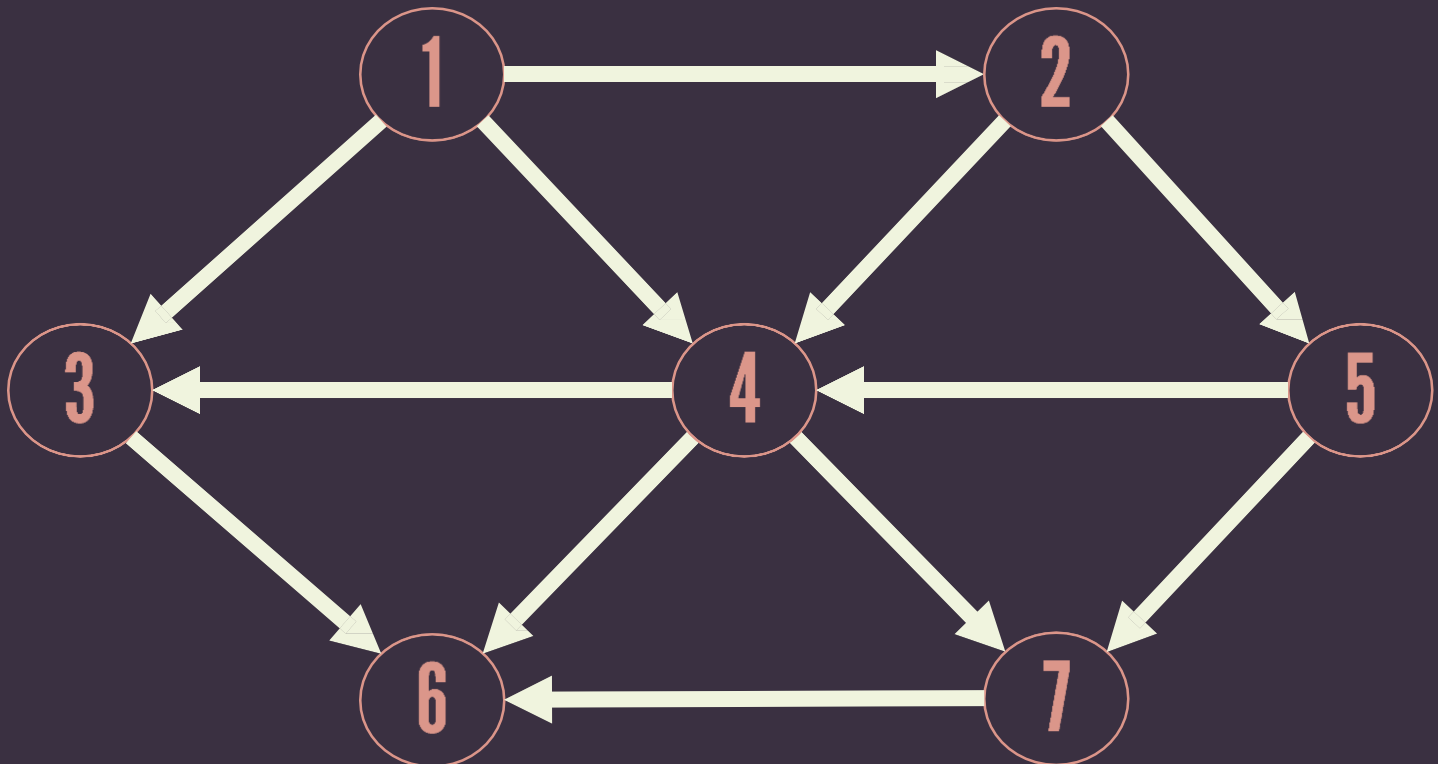
$\rho^{+}(4) : 3$



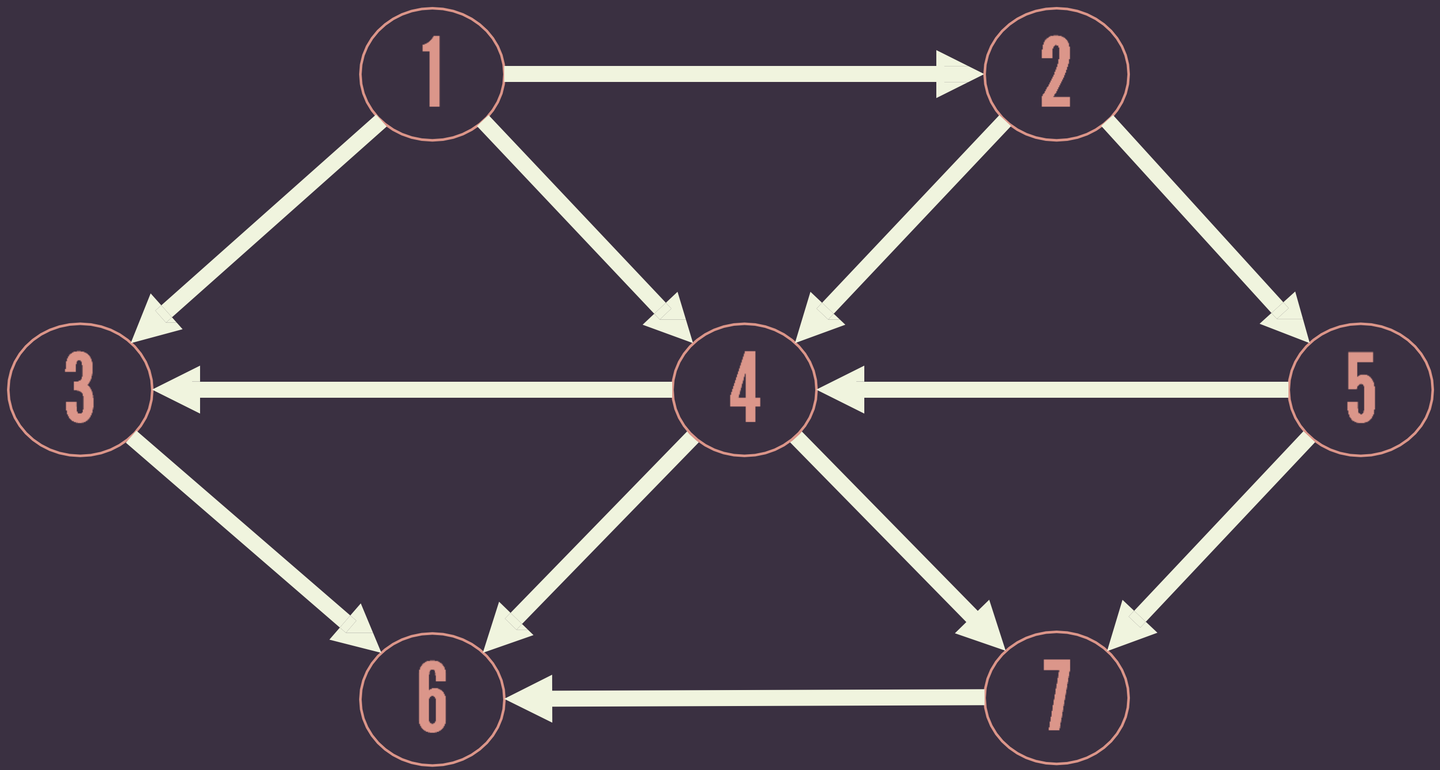
$\rho^{+}(7)$: 2



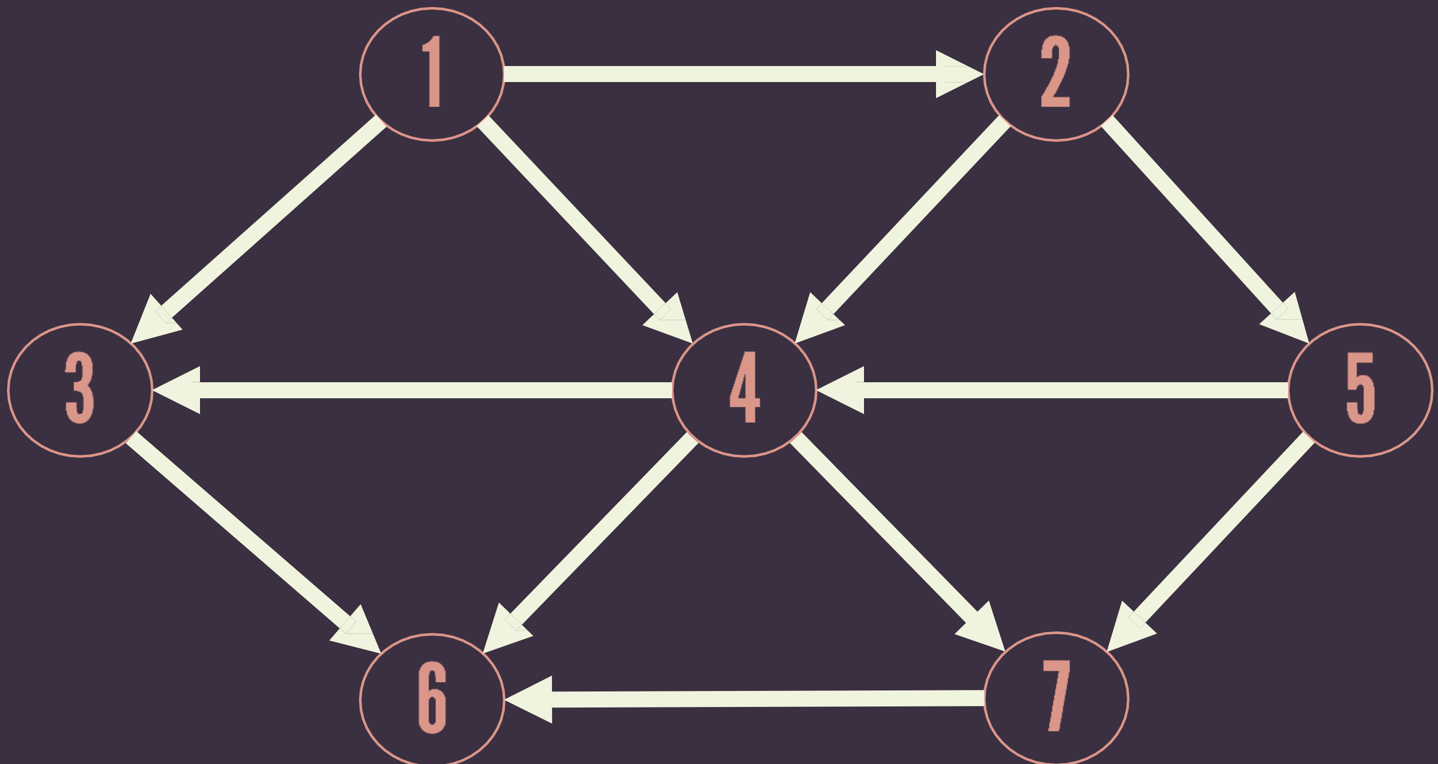
$\rho^{-}(7)$: 1



$\rho^{-}(5)$: 2



$\rho(5)$:   3



$\rho(4)$: 6

GRAPH
Representations

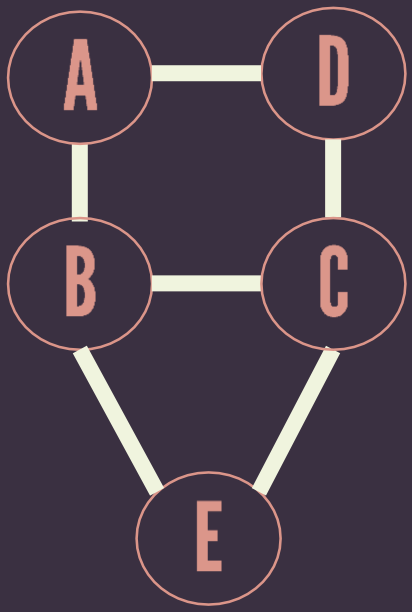
adjacency
MATRIX

2D
ARRAY

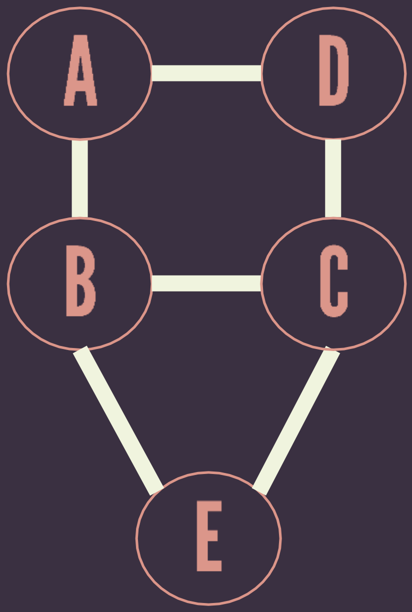
adjacency
LIST

list of
LISTS

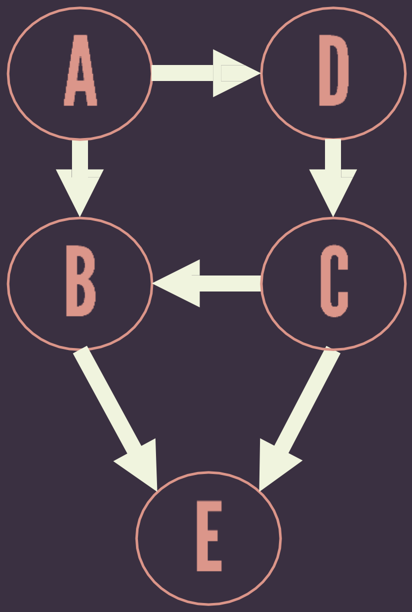
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	0	1
C	0	1	0	1	1
D	1	0	1	0	0
E	0	1	1	0	0



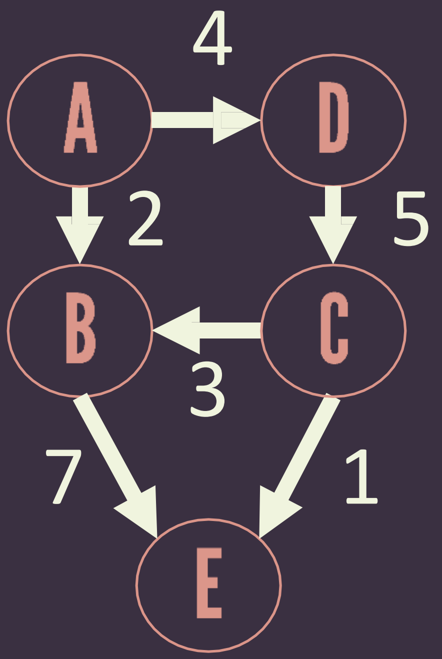
	A	B	C	D	E
A	∞	1	∞	1	∞
B	1	∞	1	∞	1
C	∞	1	∞	1	1
D	1	∞	1	∞	∞
E	∞	1	1	∞	∞

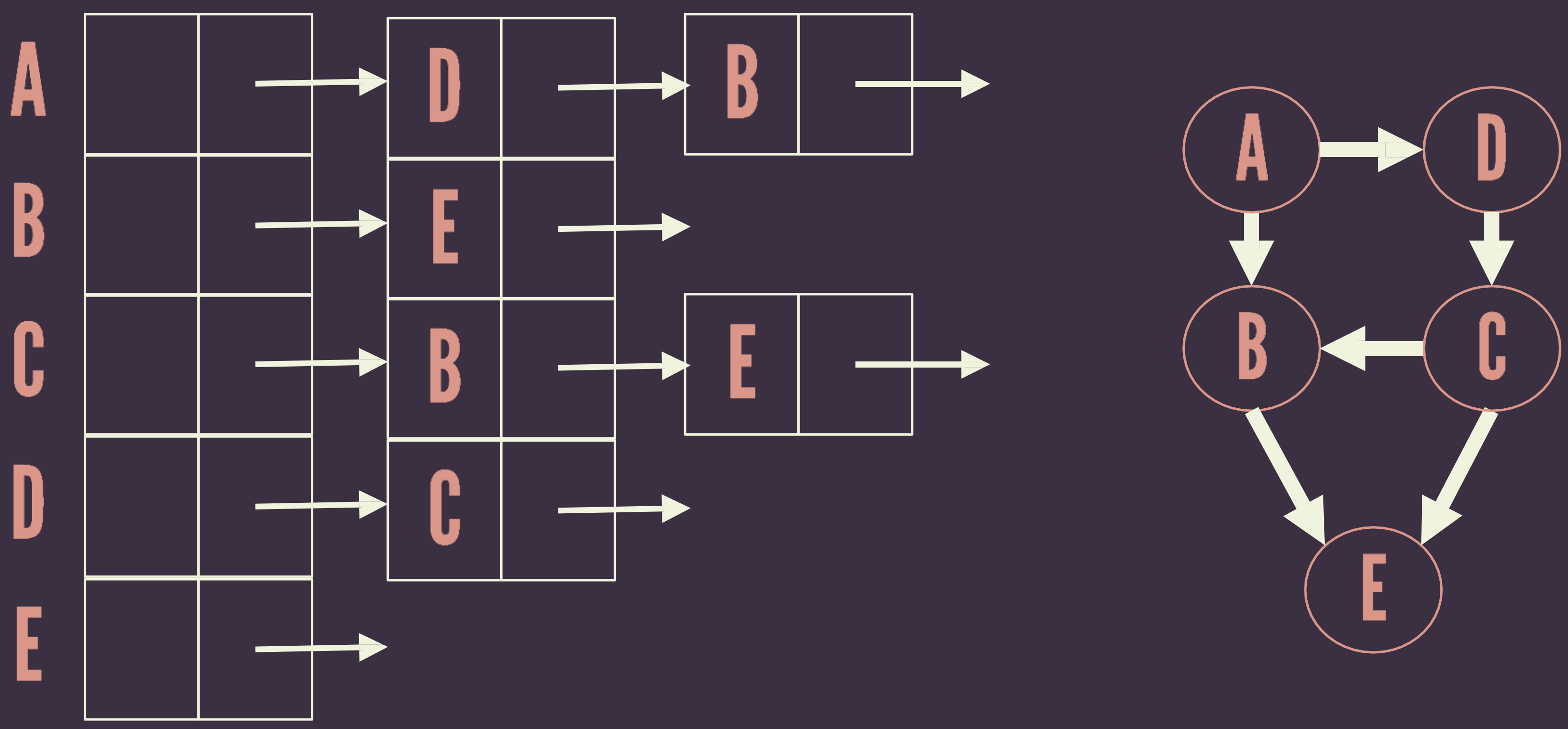


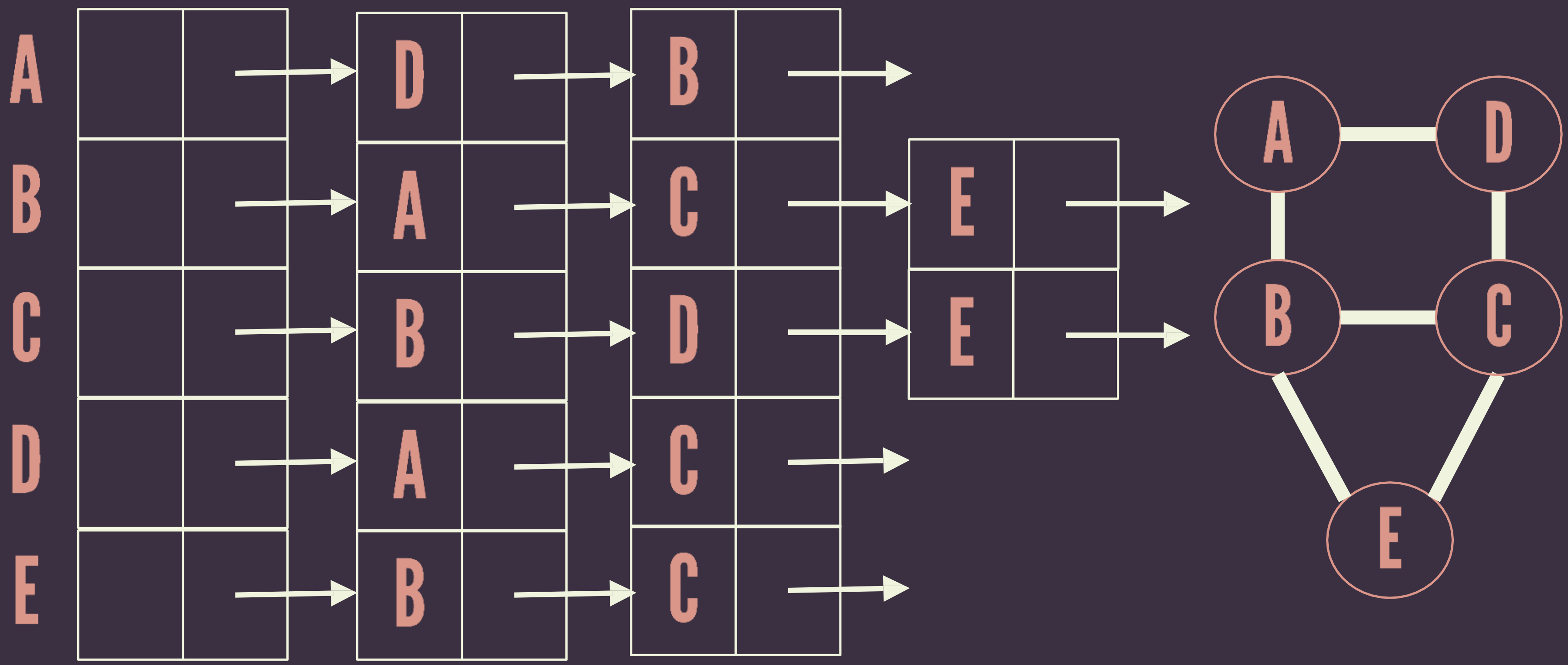
	A	B	C	D	E
A	0	1	0	1	0
B	0	0	0	0	1
C	0	1	0	0	1
D	0	0	1	0	0
E	0	0	0	0	0



	A	B	C	D	E
A	0	2	0	4	0
B	0	0	0	0	7
C	0	3	0	0	1
D	0	0	5	0	0
E	0	0	0	0	0



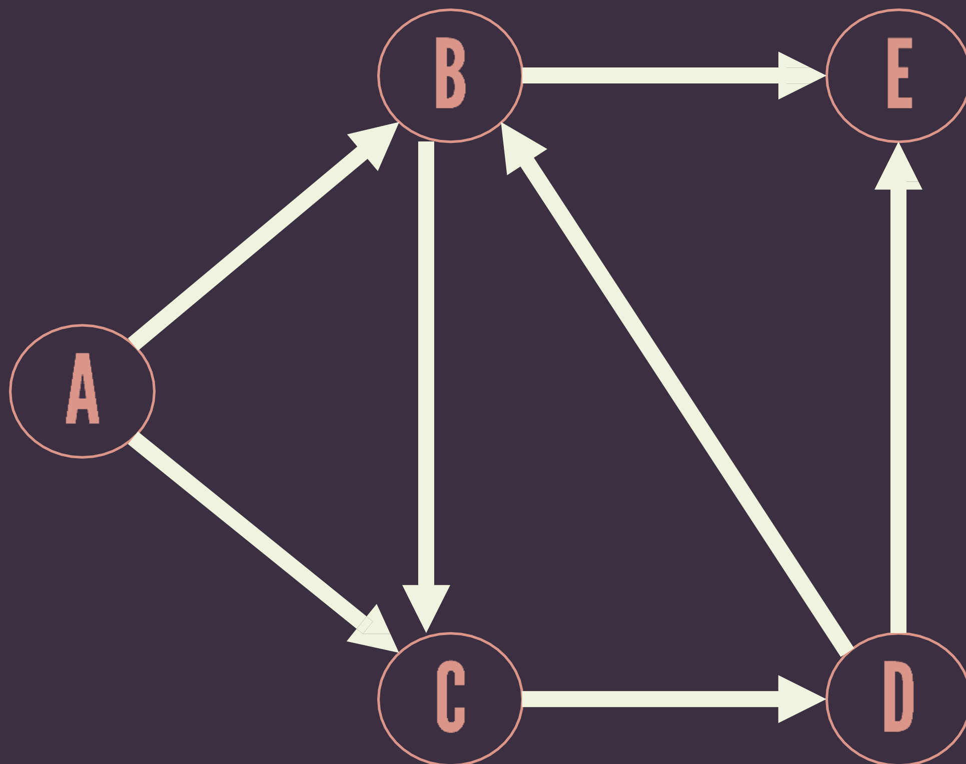

A
B
C
D
E
D
B
E
B
E
C
A
D
B
C
E


A
B
C
D
E
D
B
A
C
E
B
D
A
C
B
C
A
D
B
C
E
E

GRAPH ALGORITHMS

topological **SORT**

An ordering of vertices in a directed acyclic graph such that if there is a path from v_i to v_j, then v_j appears after vi in the ordering.



A, B, C, D, E

A, C, D, B, E

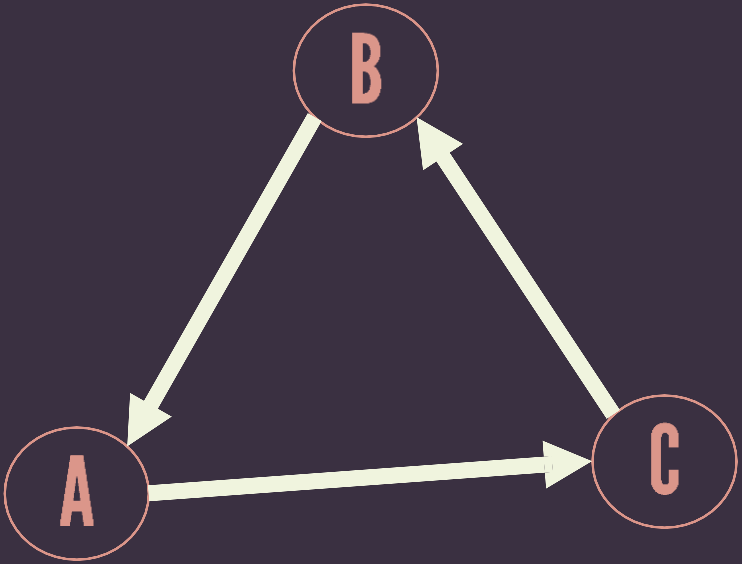
directed
acyclic graph

DAG

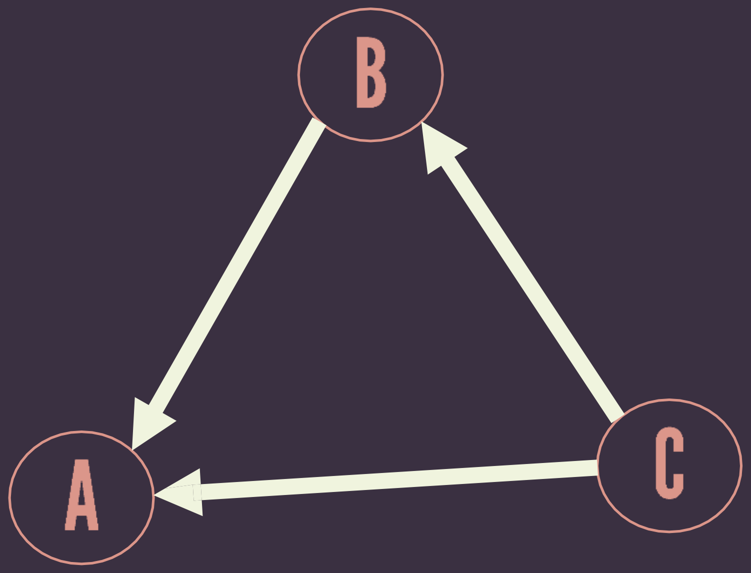
A directed graph with no directed cycles.

directed
acyclic graph

DAG

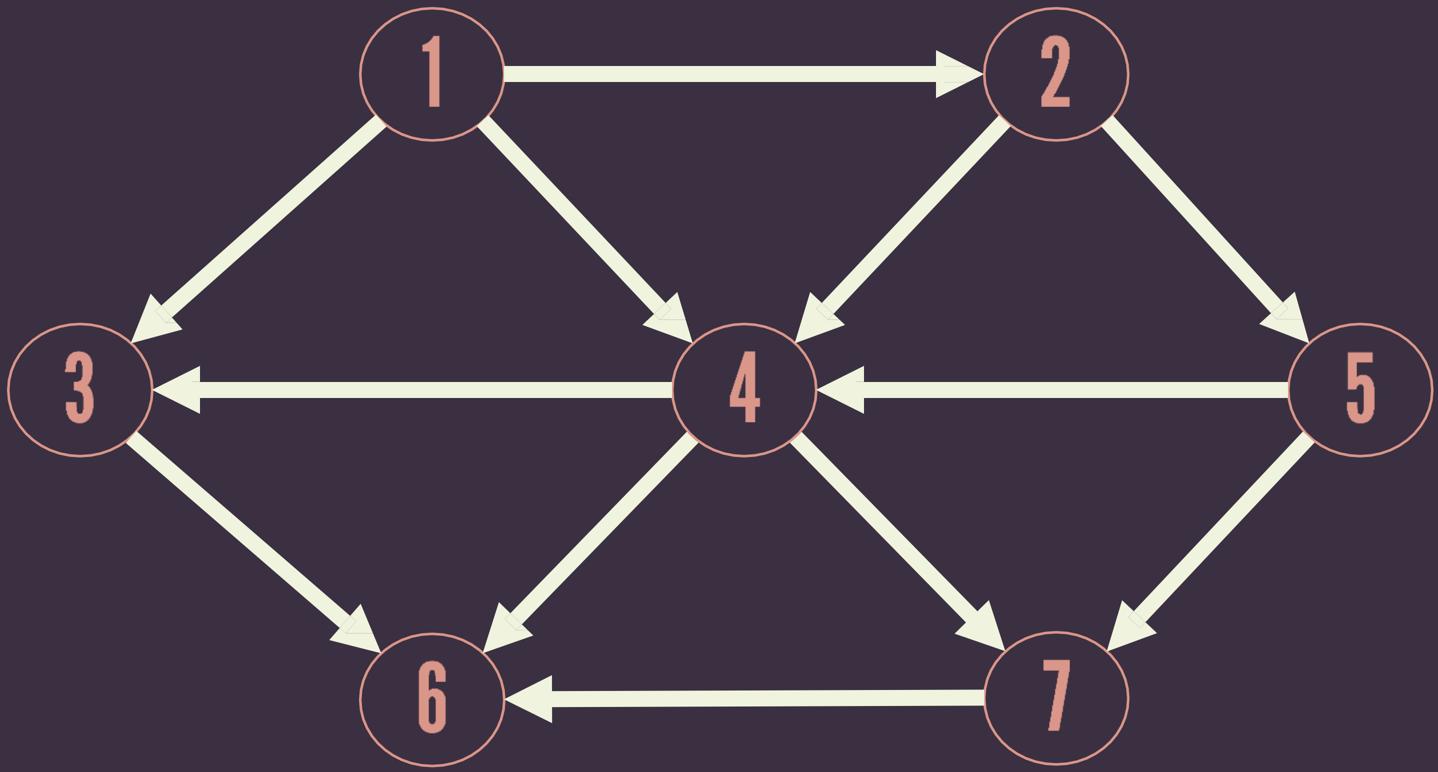


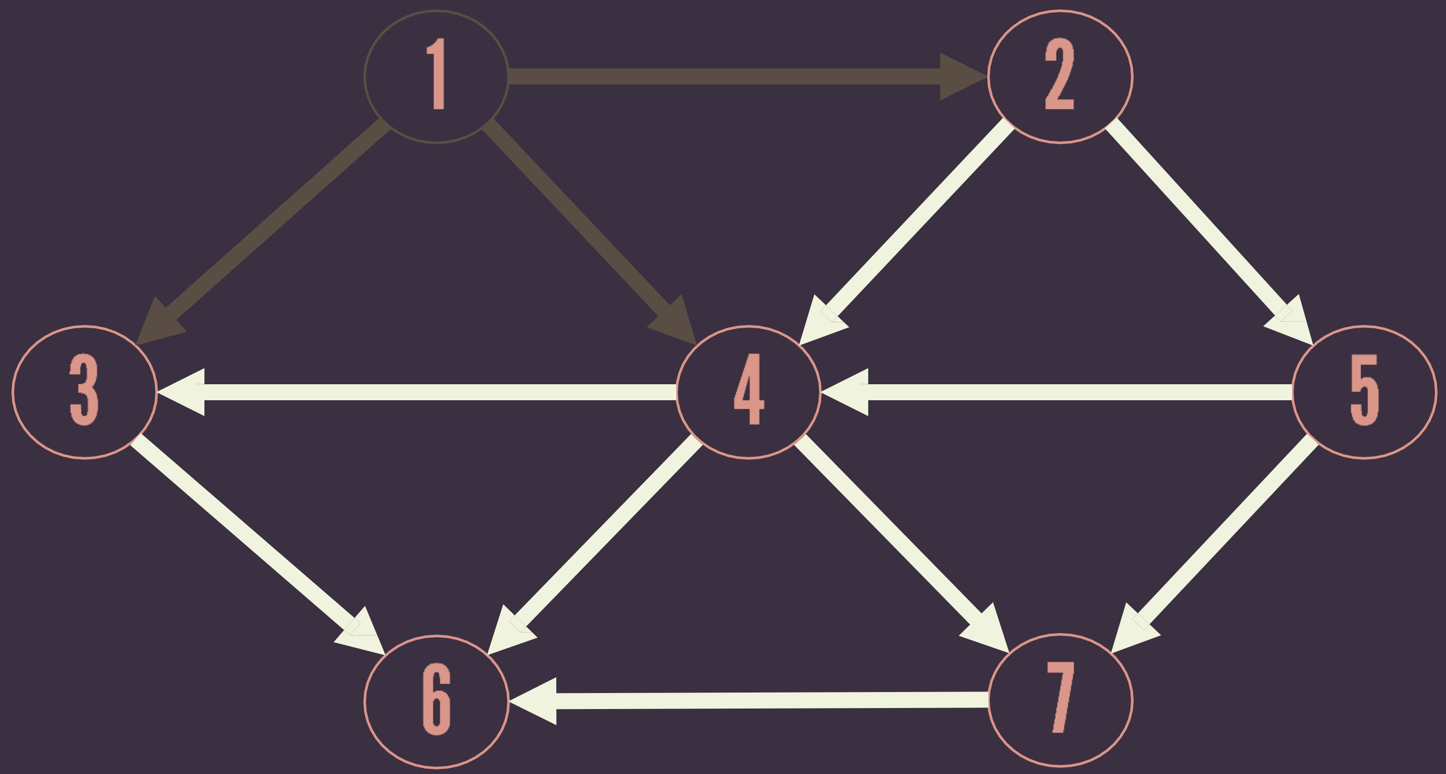
directed
acyclic graph
DAG

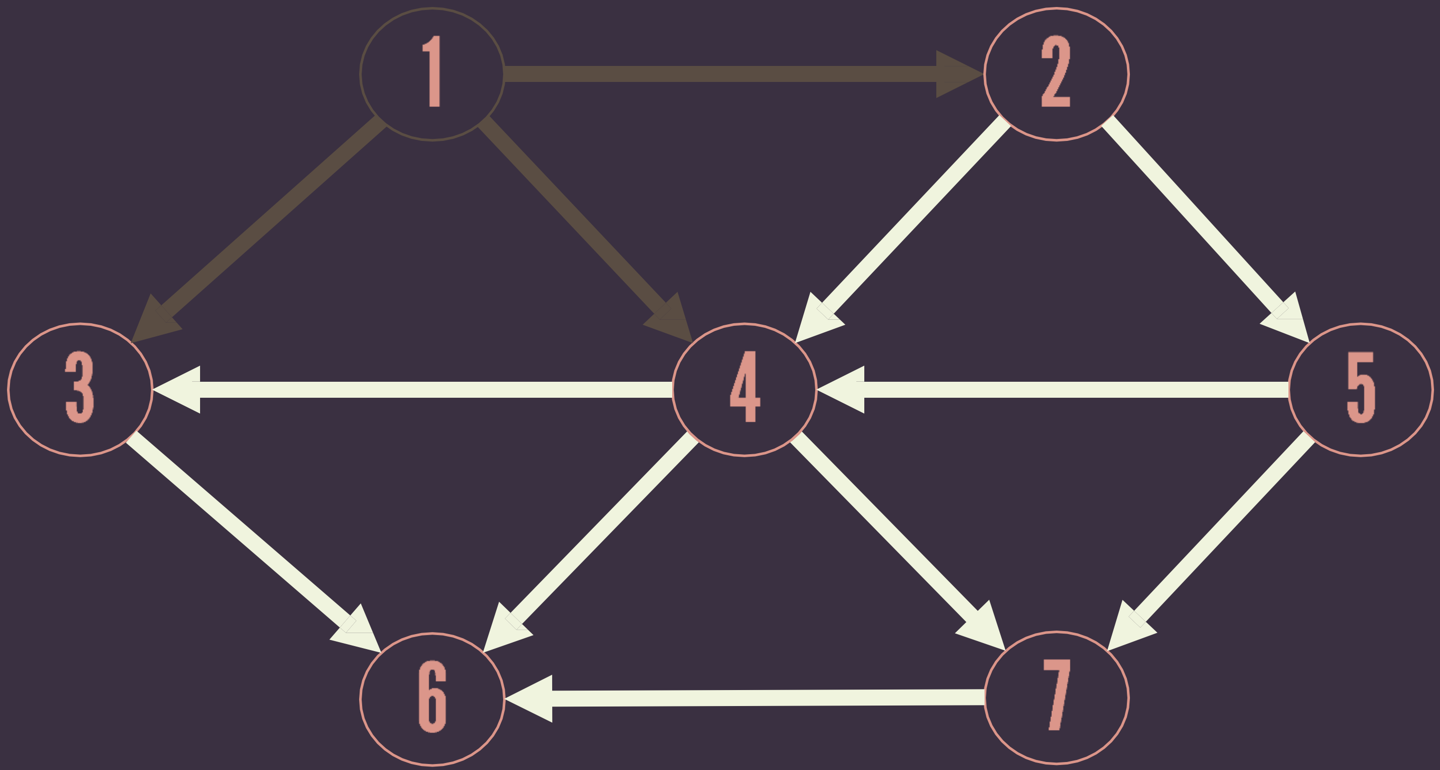


Algorithm:

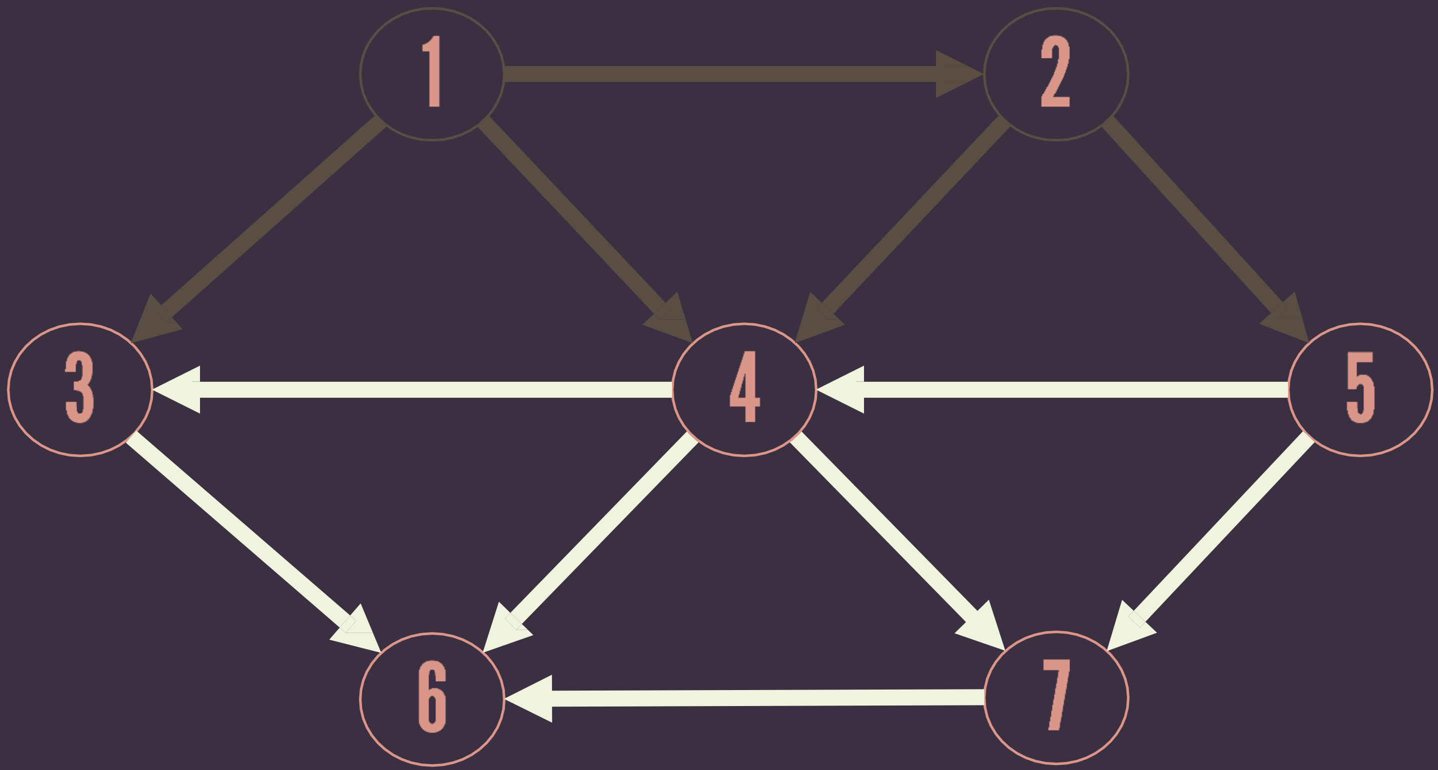
- Find any vertex with no incoming edges (in-degree is 0).
- Print this vertex and remove it along with its edges from the graph.
- Repeat steps above.

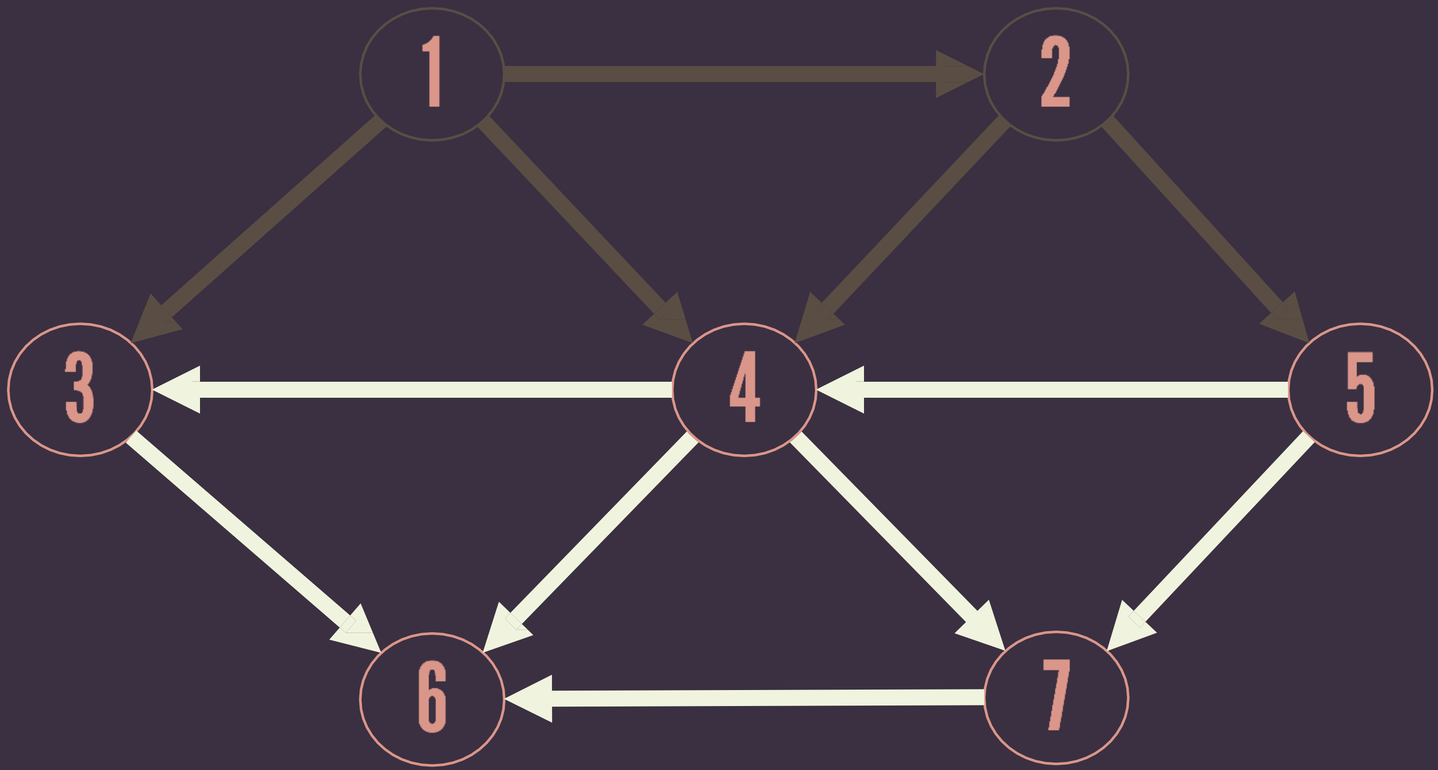

1
2
3
4
5
6
7


1
2
3
4
5
6
7

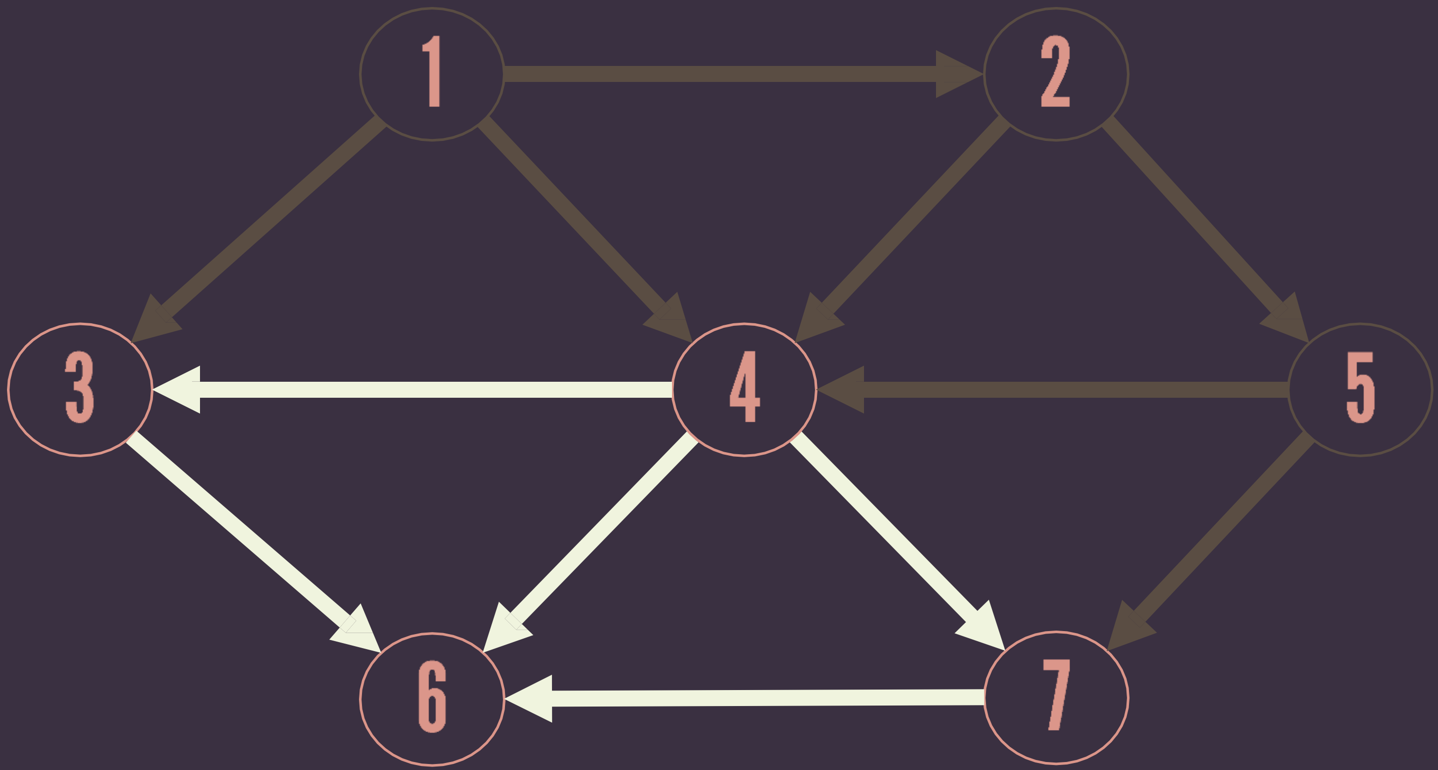


1 2

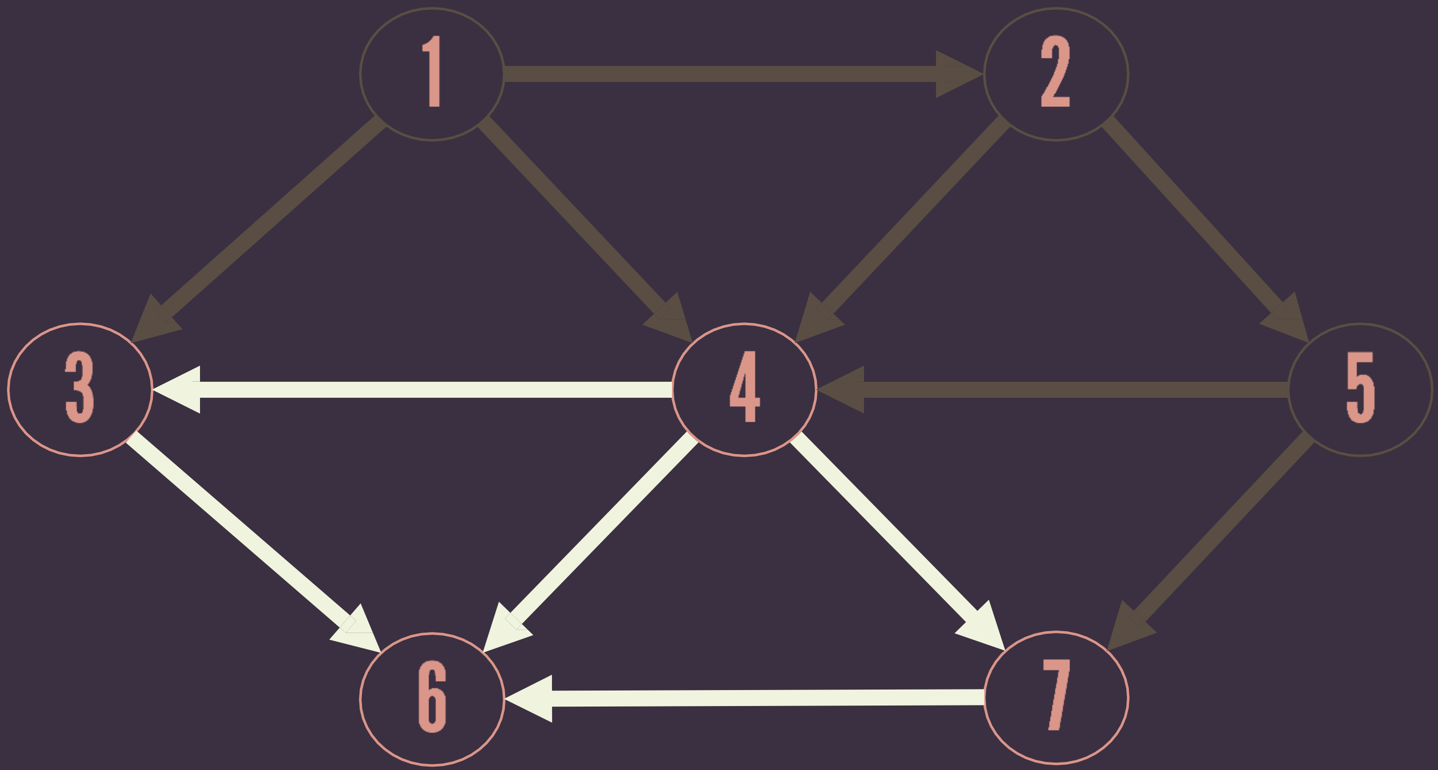

1
2
3
4
5
6
7



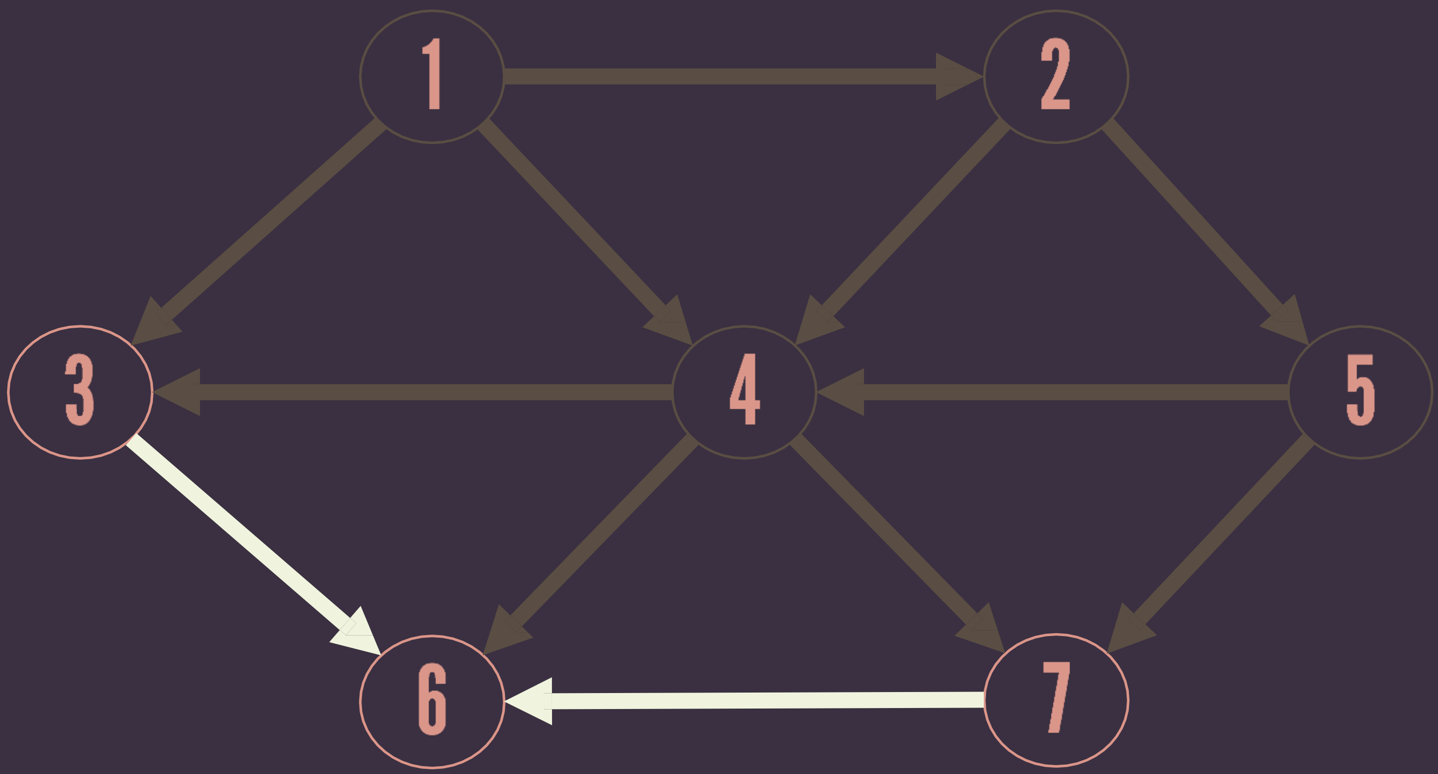
1 2 5



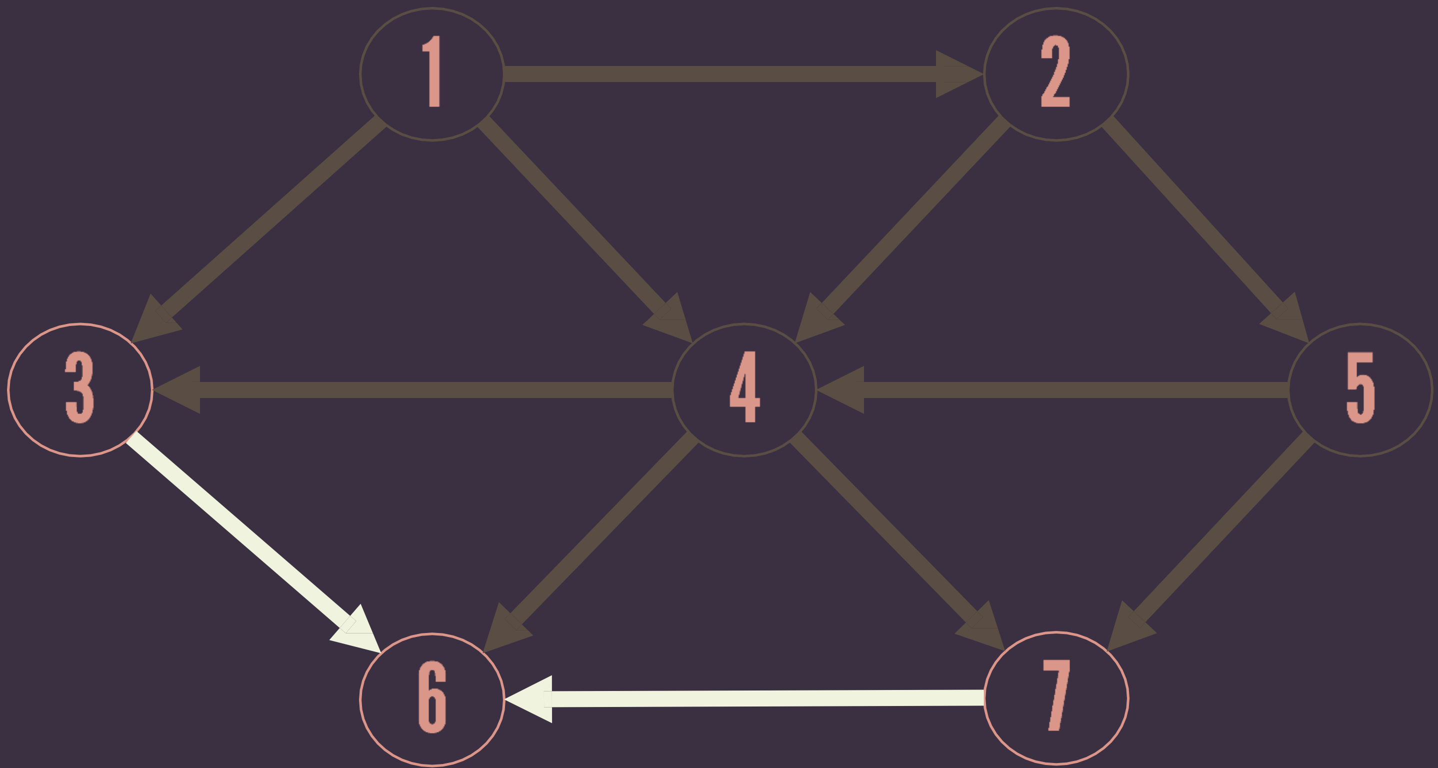
1 2 5



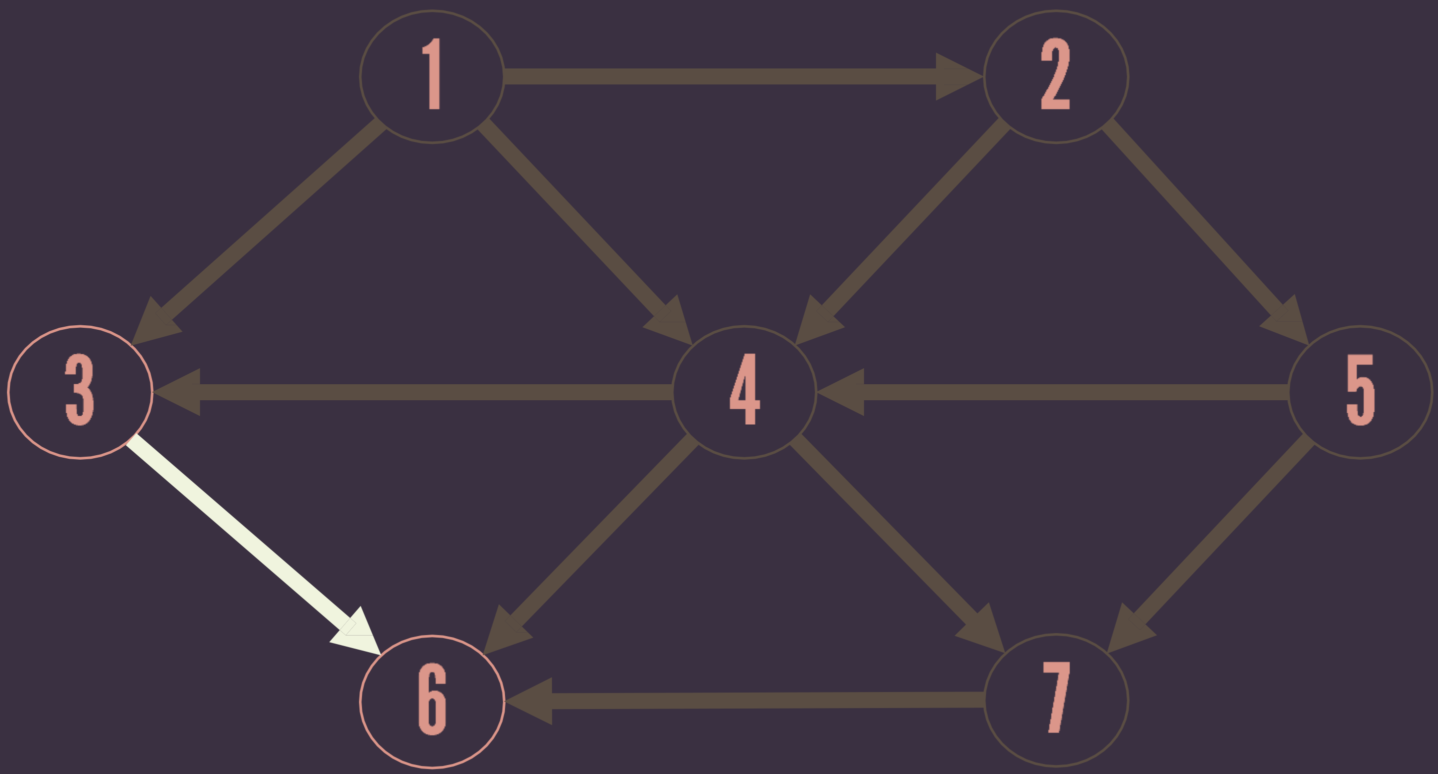
1 2 5 4



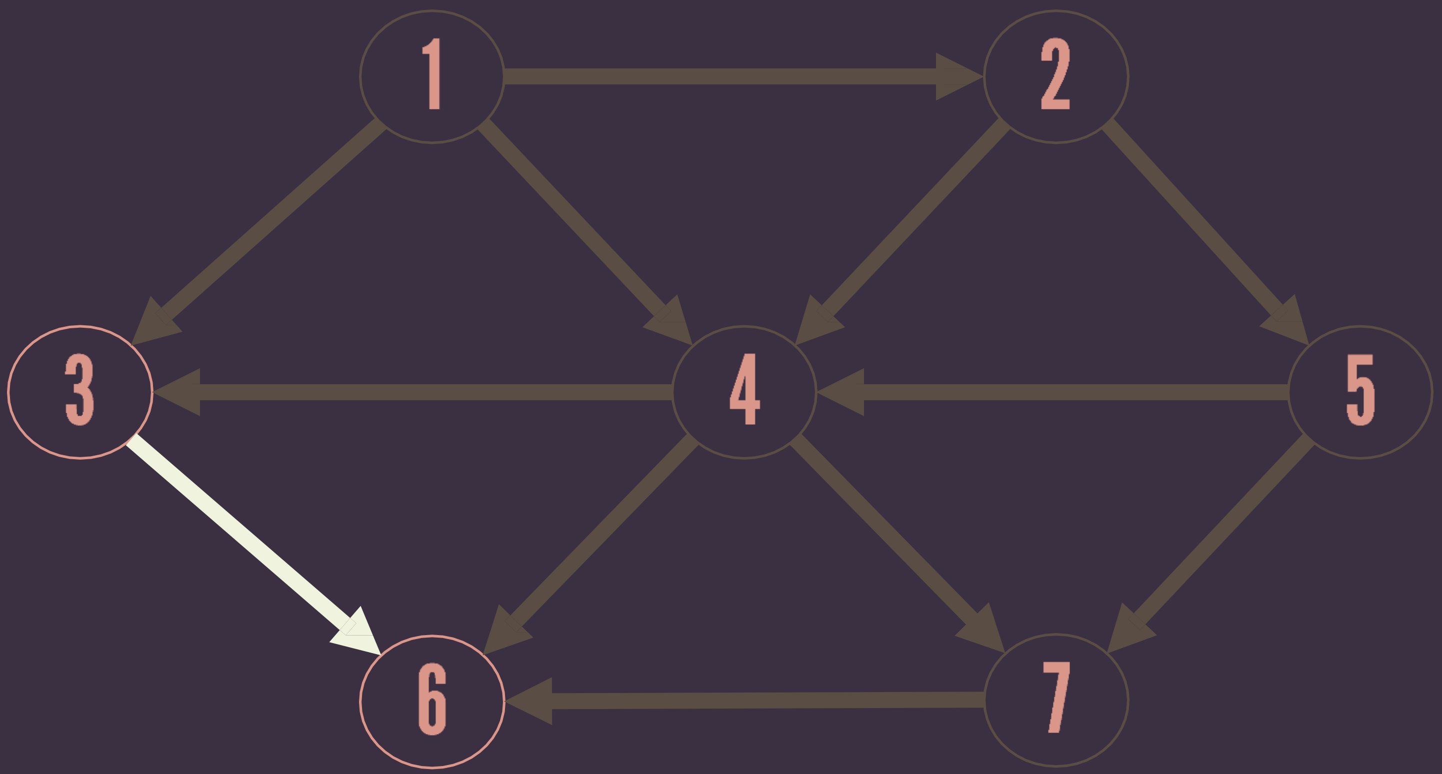
1 2 5 4



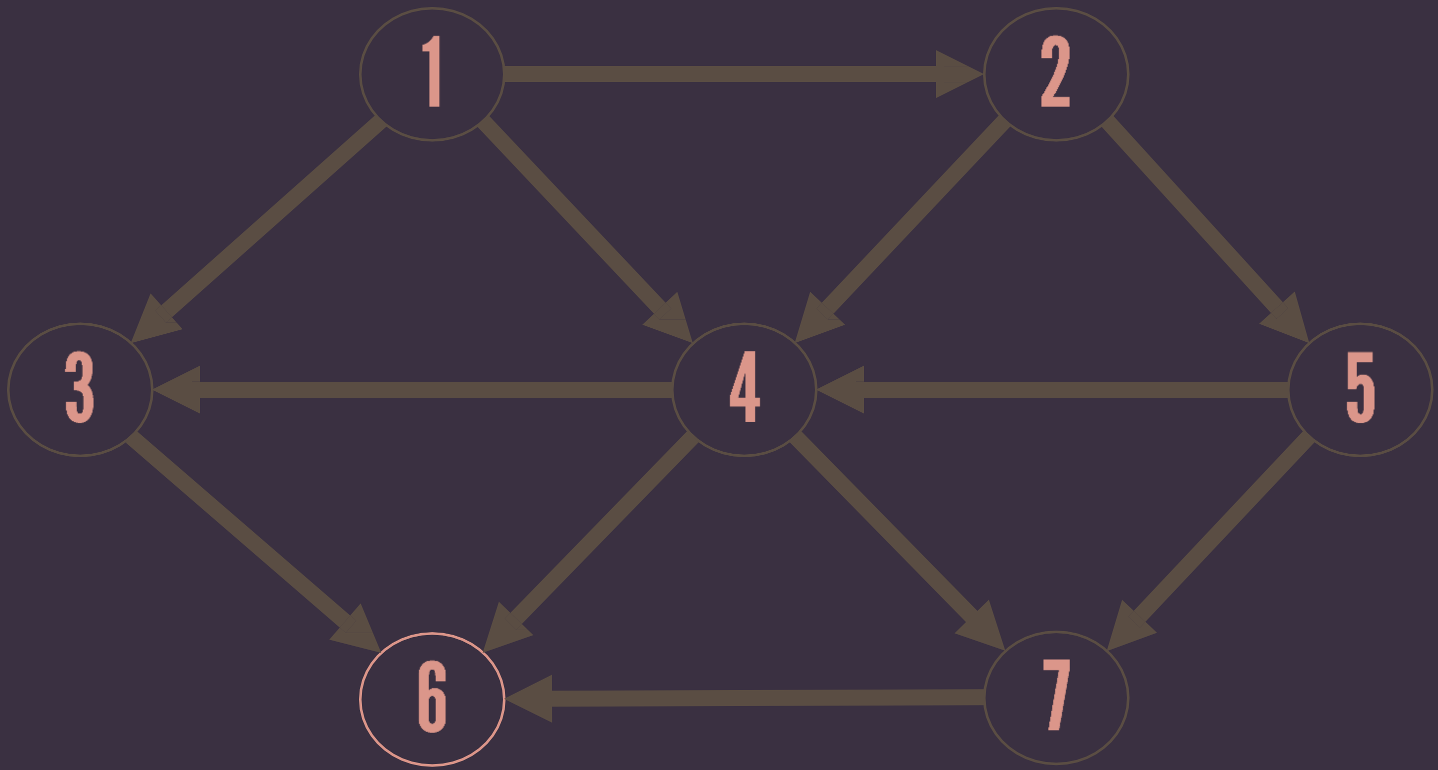
1 2 5 4 7



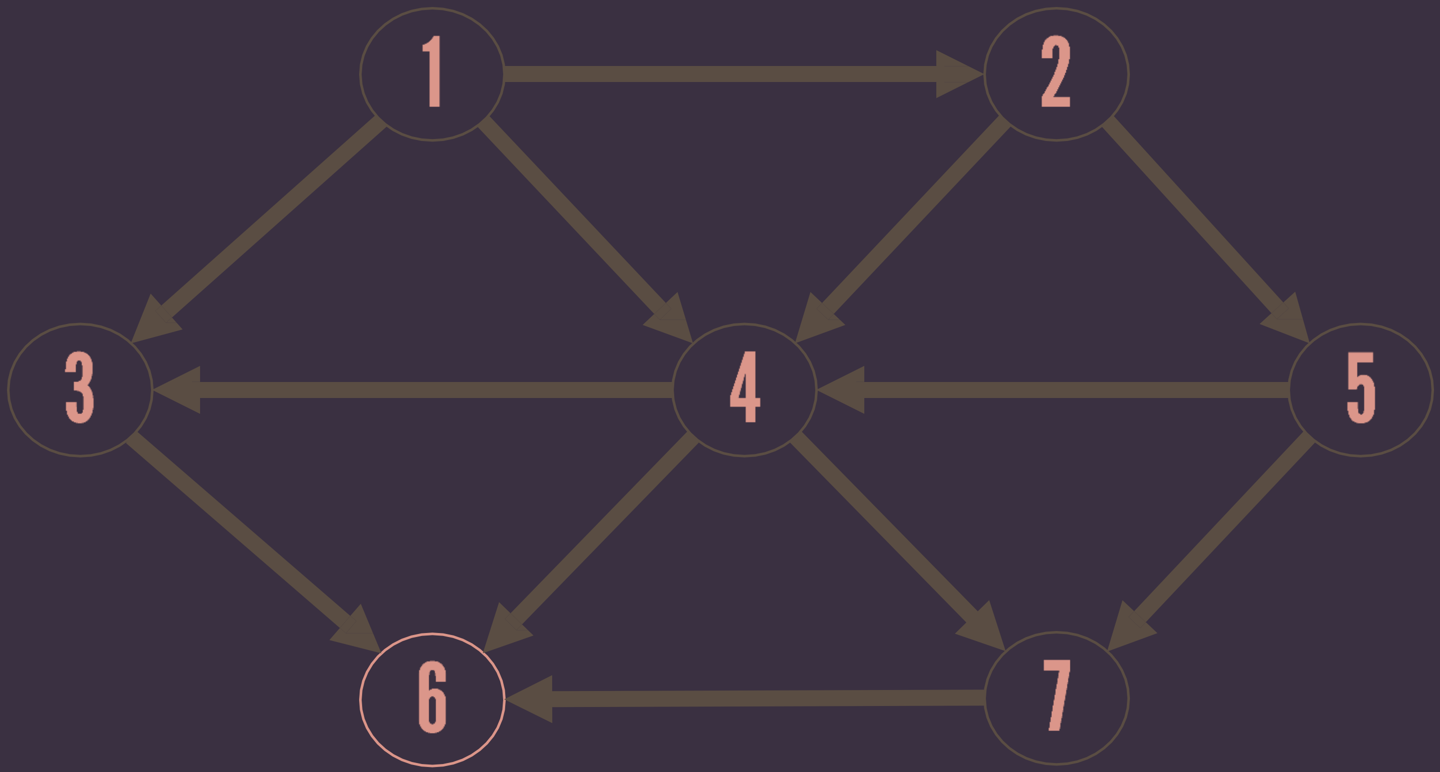
1 2 5 4 7



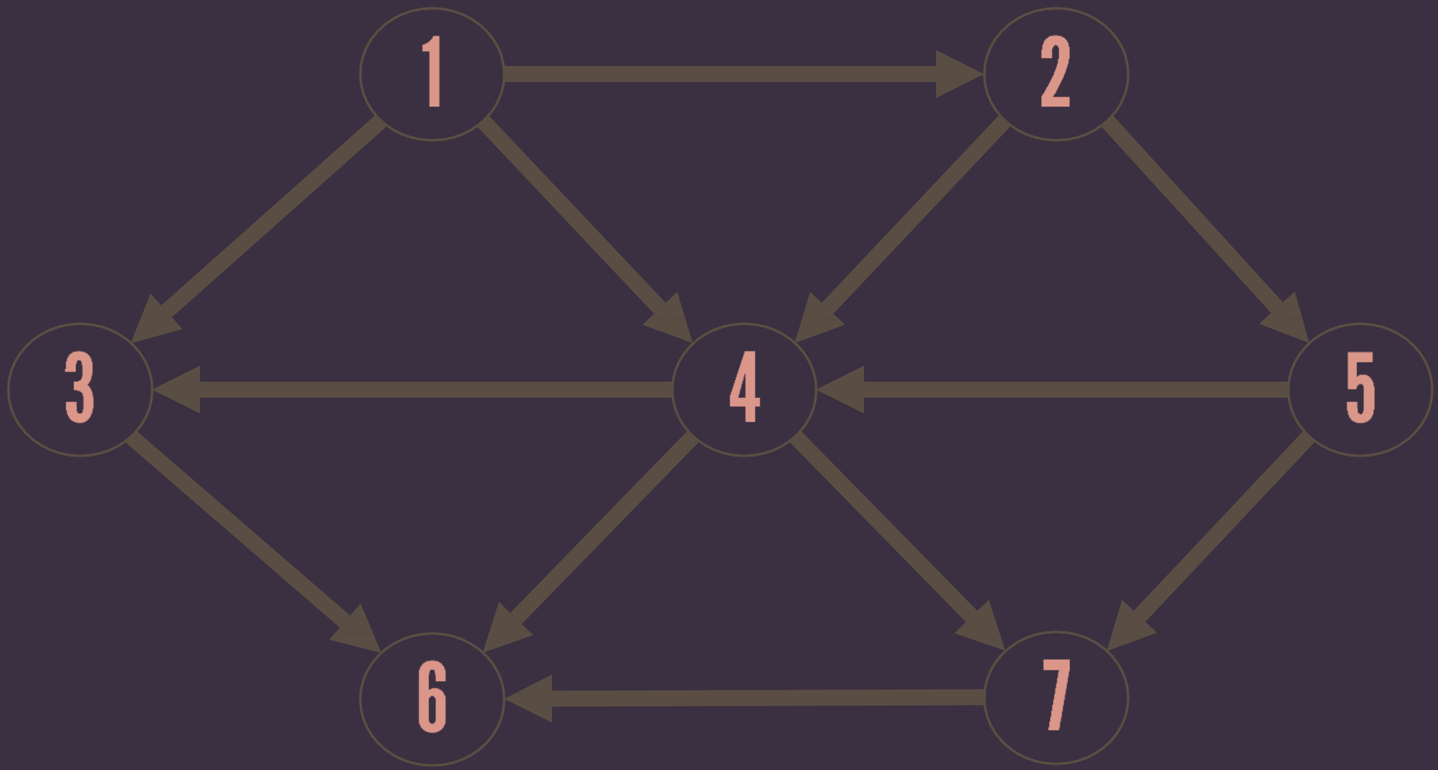
1 2 5 4 7 3



1 2 5 4 7 3



1 2 5 4 7 3 6



1 2 5 4 7 3 6

graph

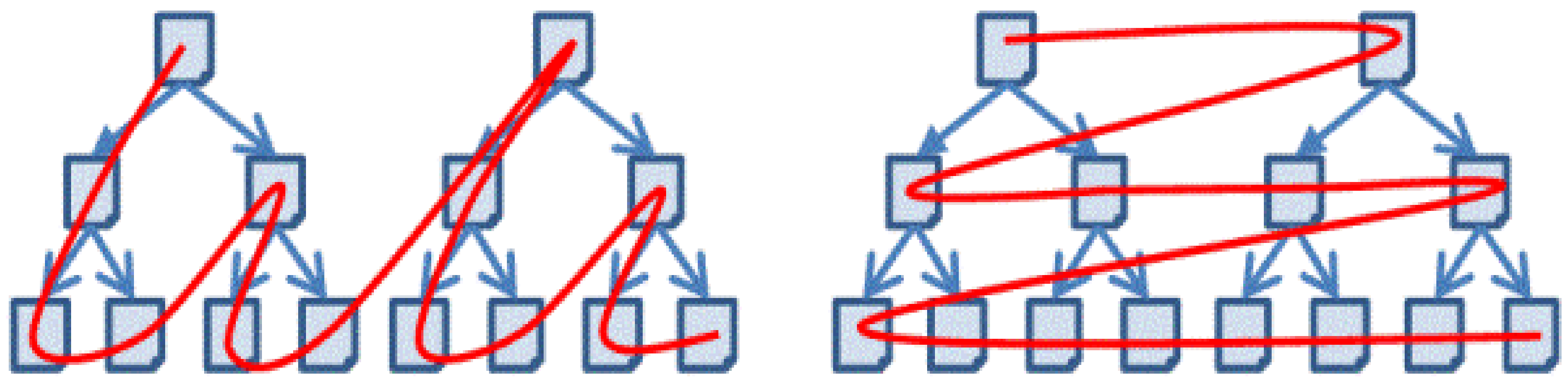
TRAVERSALS

DEPTH FIRST

search

BREADTH FIRST

search



a) b)

ifp.uni-stuttgart.de

DEPTH FIRST
search

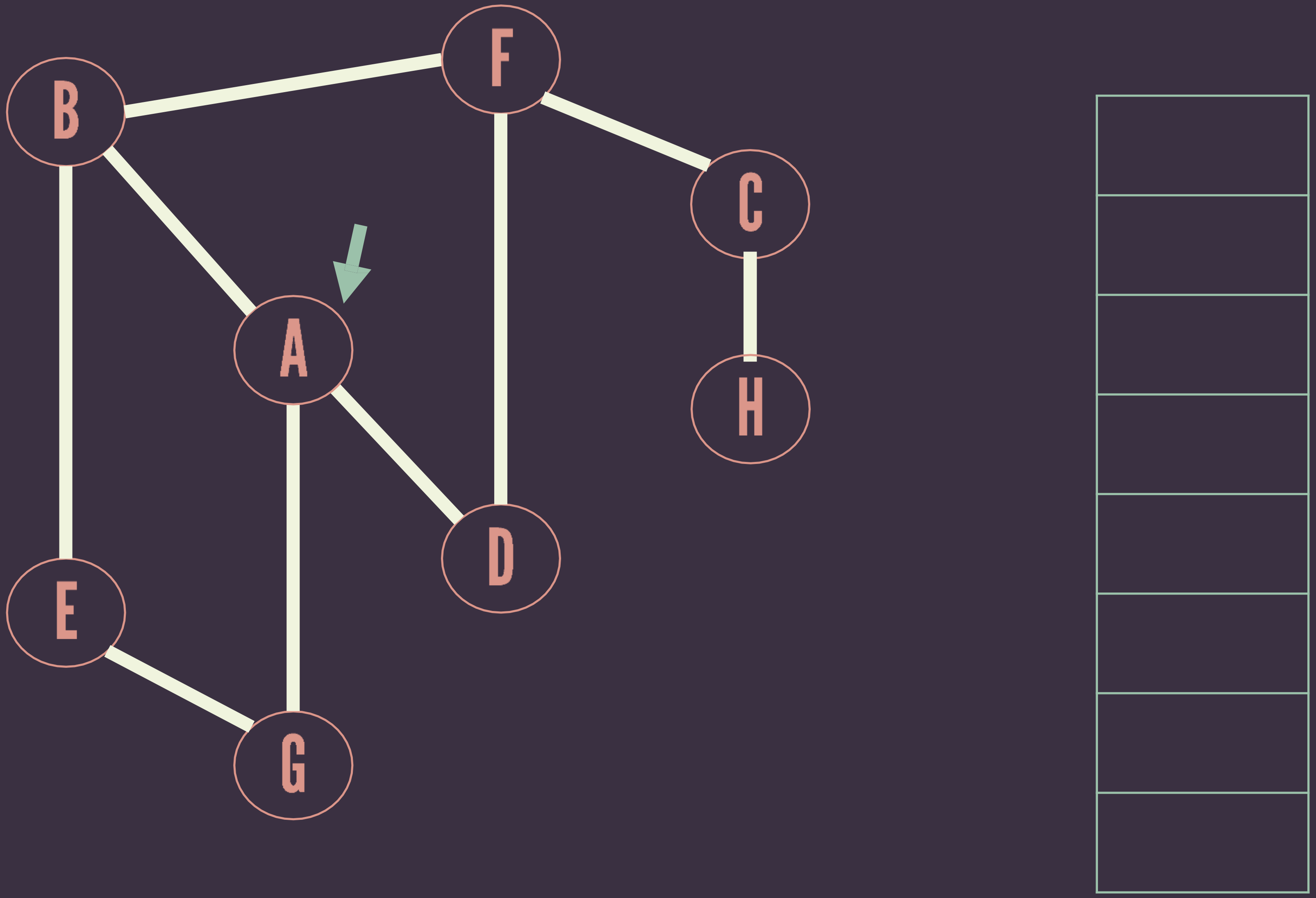
Generalization of preorder traversal.

DEPTH FIRST search

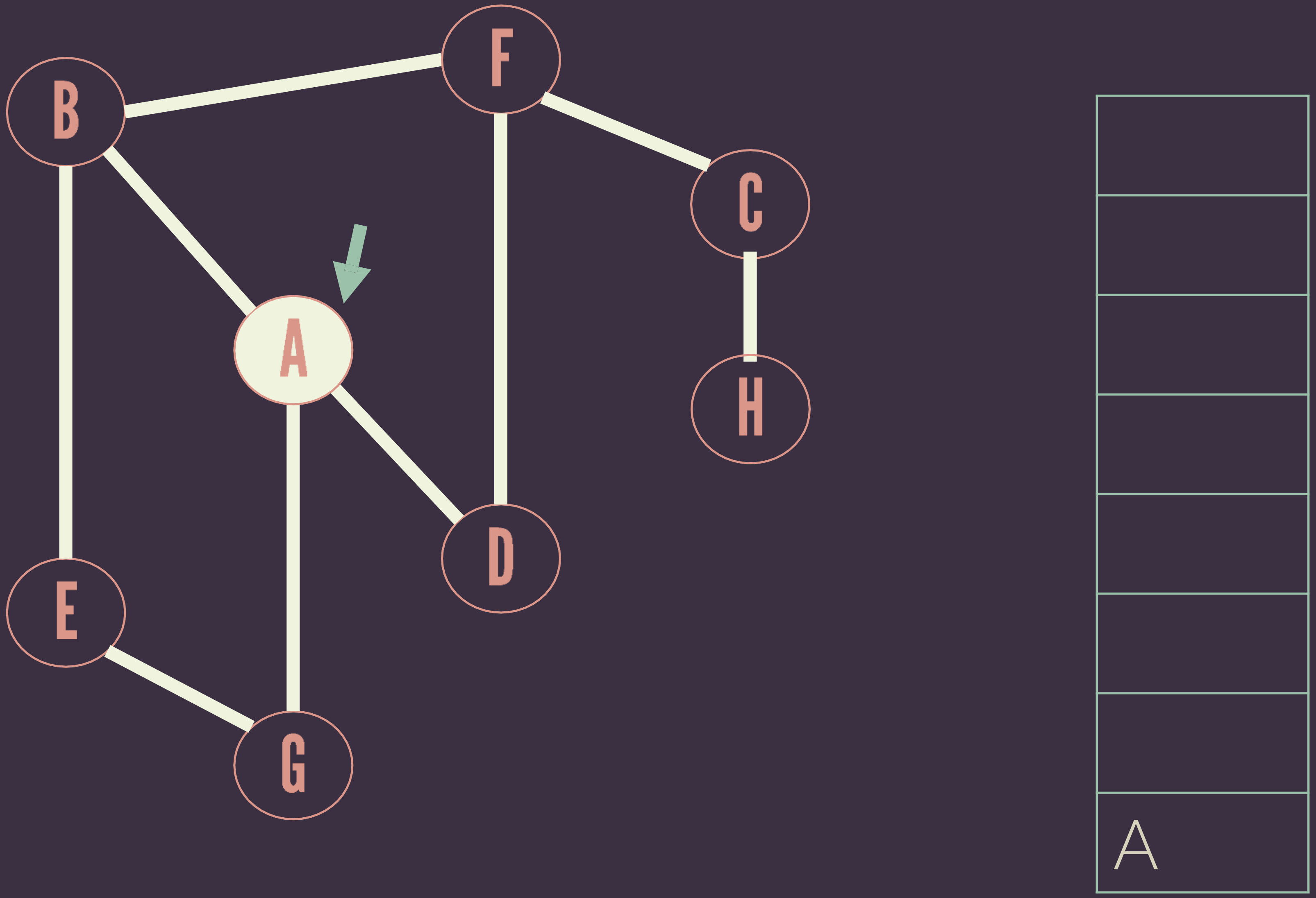
Starting at some vertex v, process v and recursively traverse all vertices adjacent to v.

```
void DFS( vertex v, graph G ){
    print v;
    visited[v] = TRUE;
    for each w adjacent to v:
        if( !visited[w] )
            DFS(w);
}
```

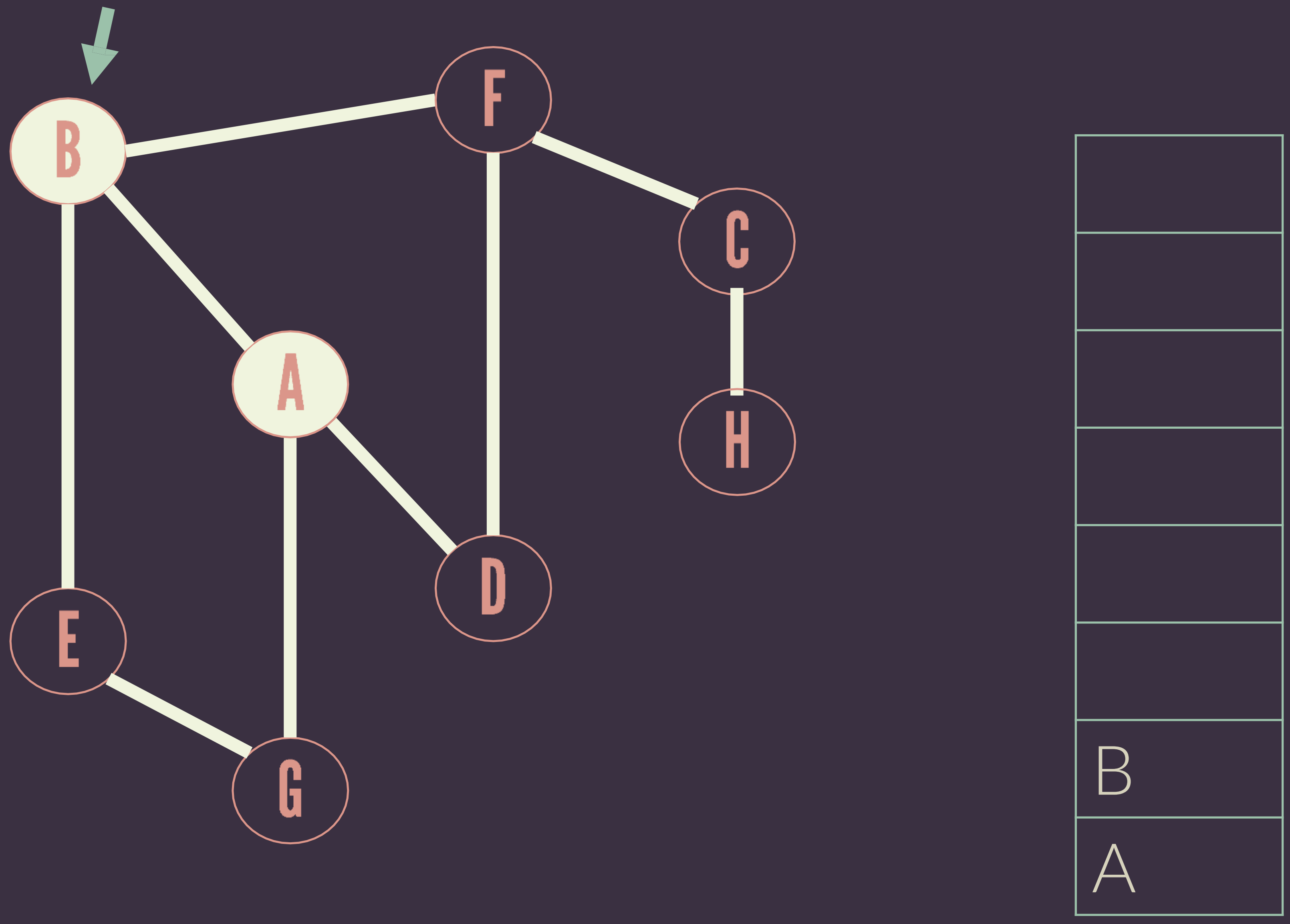
```
void DFS( vertex v, graph G ){
    stack S;
    push(v,S);
    while stack S is not empty{
        v = pop(S);
        if ( !visited[v] ){  print v;
            visited[v] = TRUE;
            for each w adjacent to v:
                if( !visited[w] )
                    push(w, S);
        }
    }
}
```



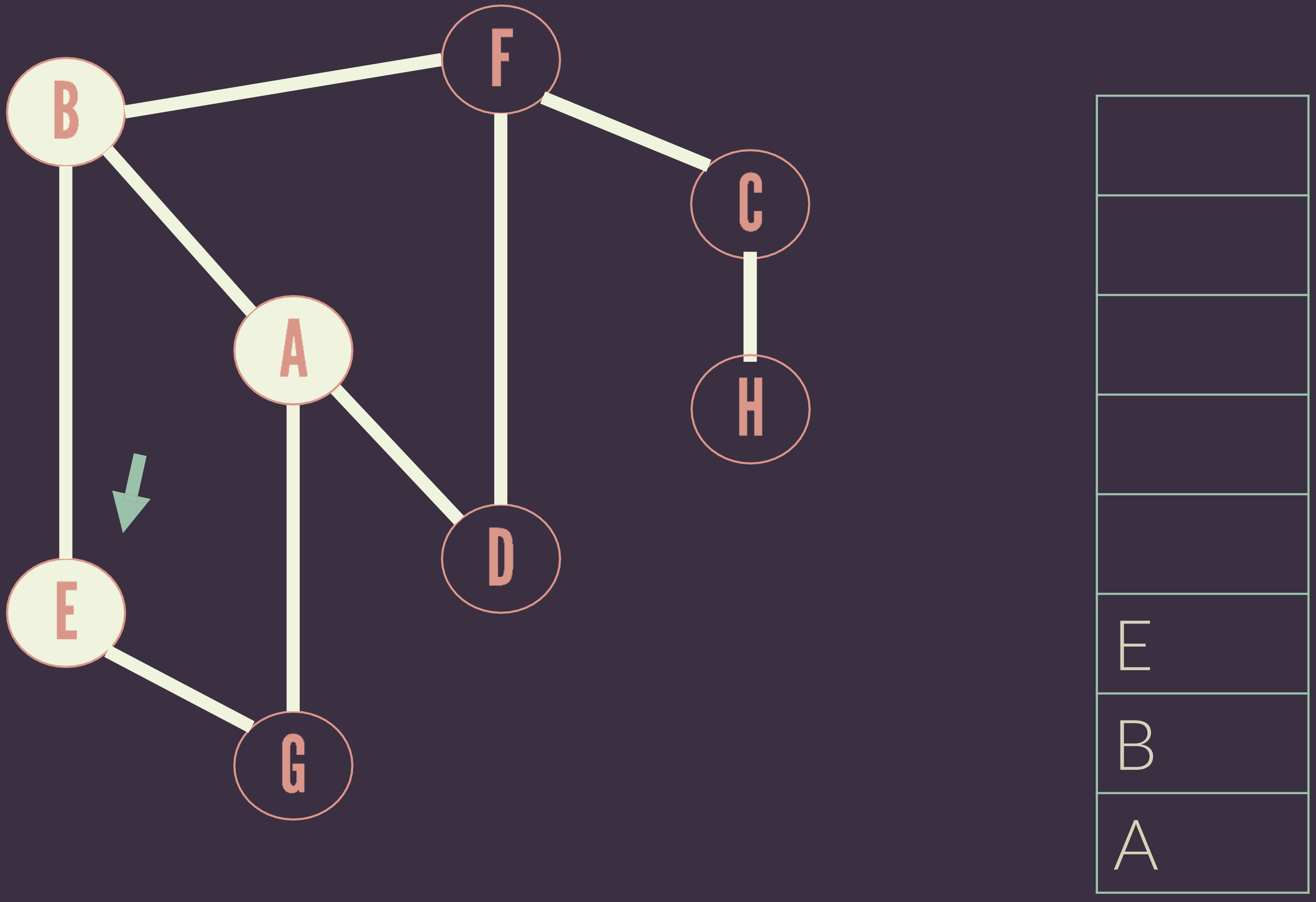
Result:



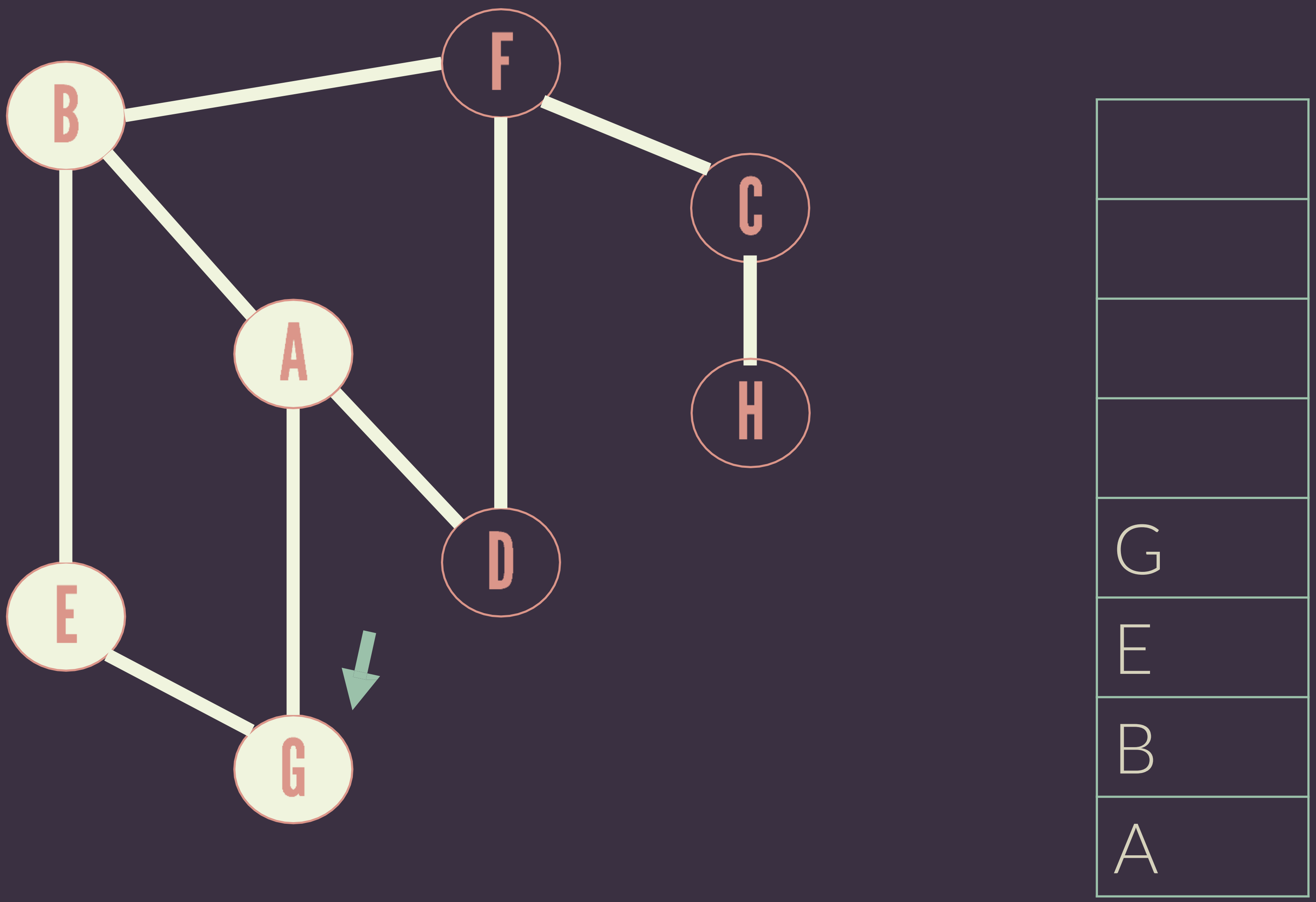
Result: A



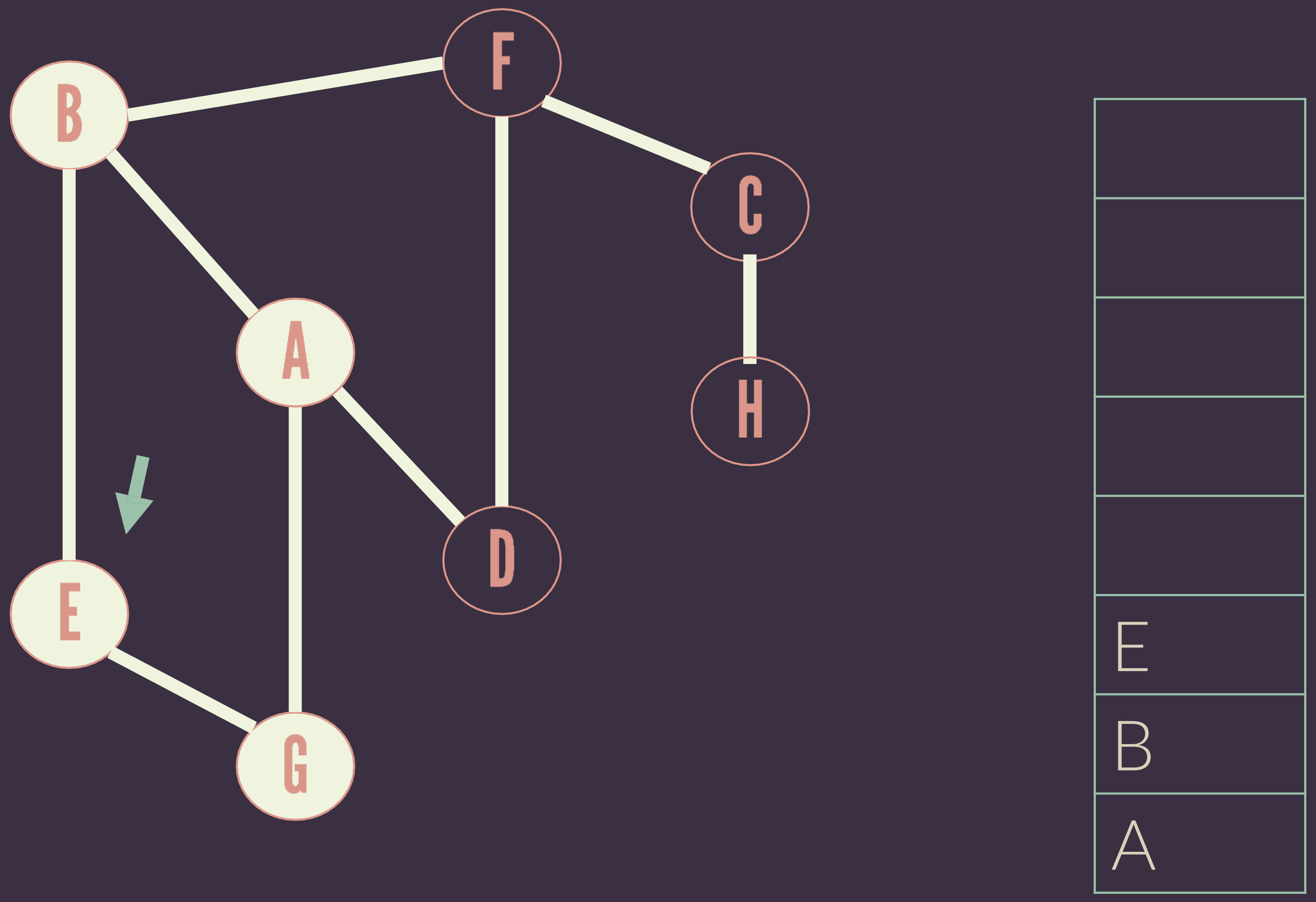
Result: AB



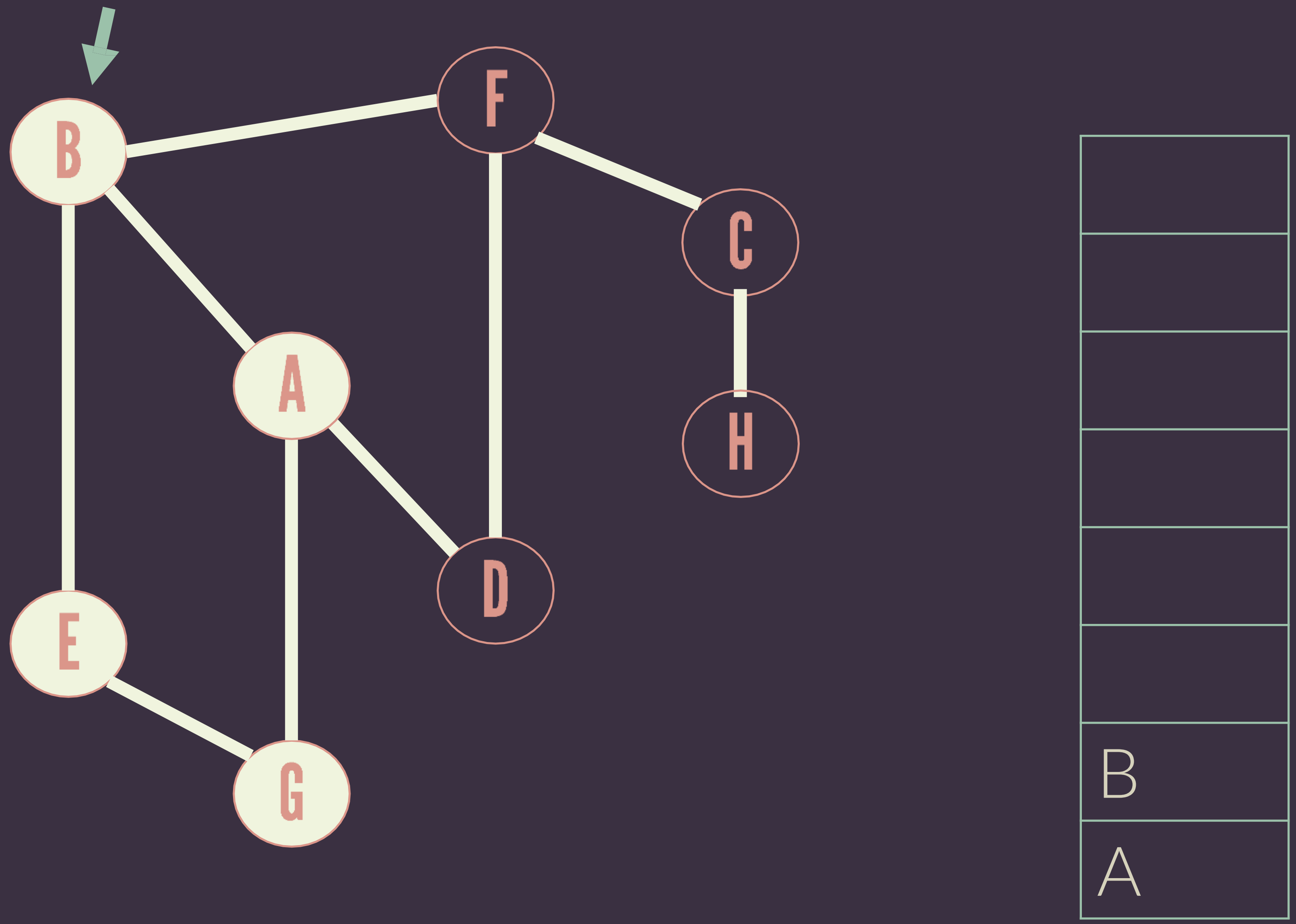
Result: A B E



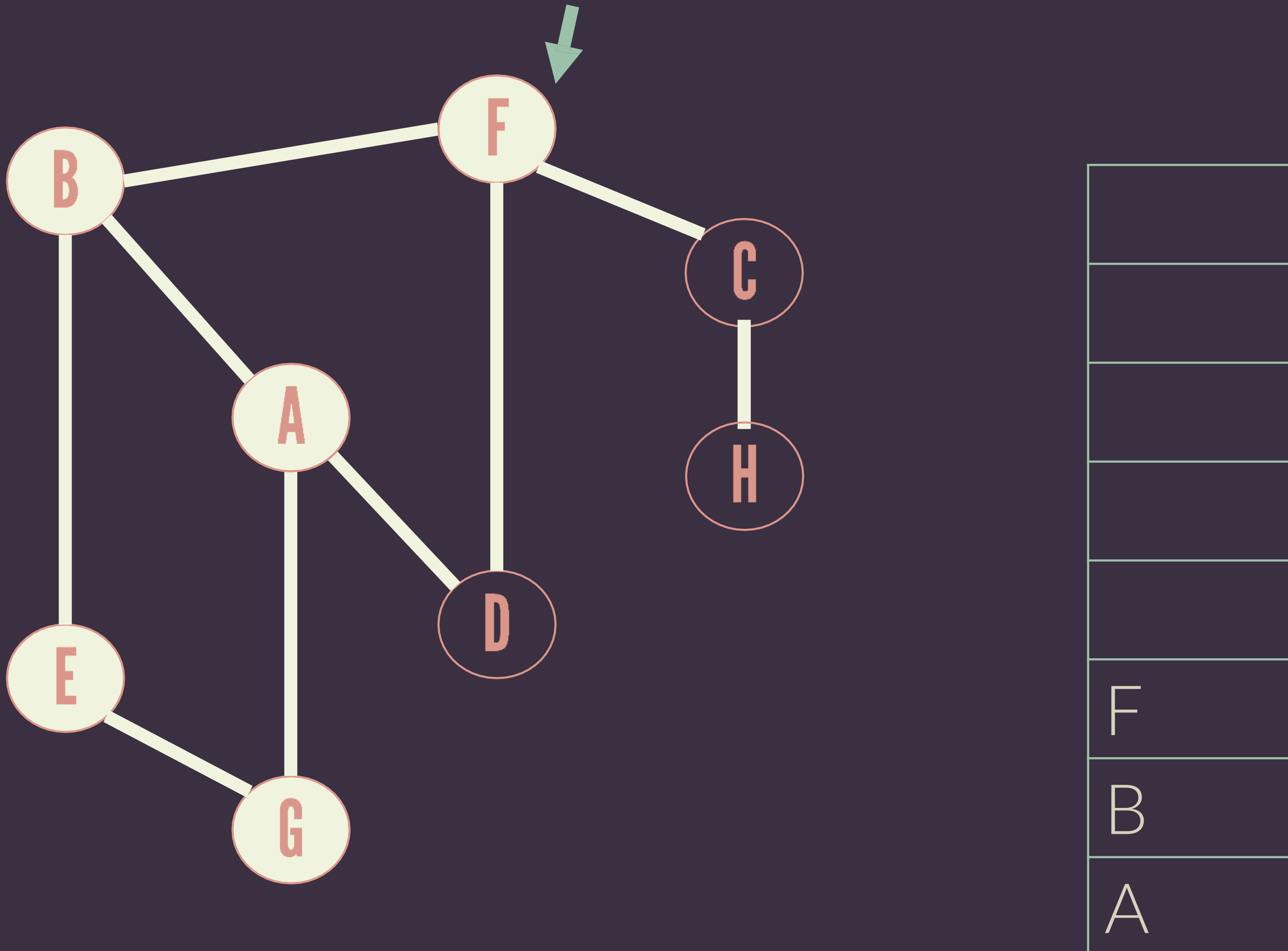
Result: A B E G



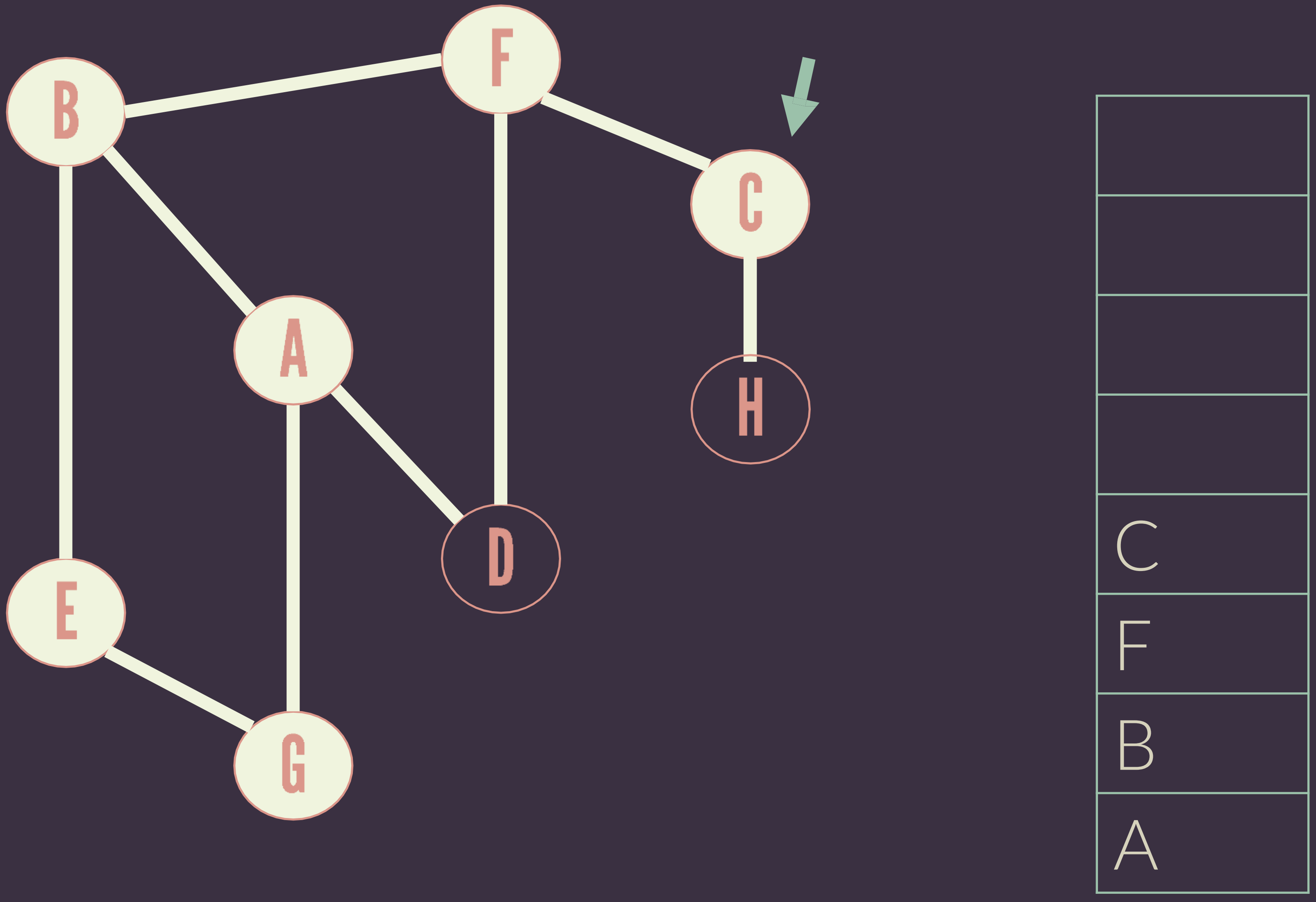
Result: A B E G



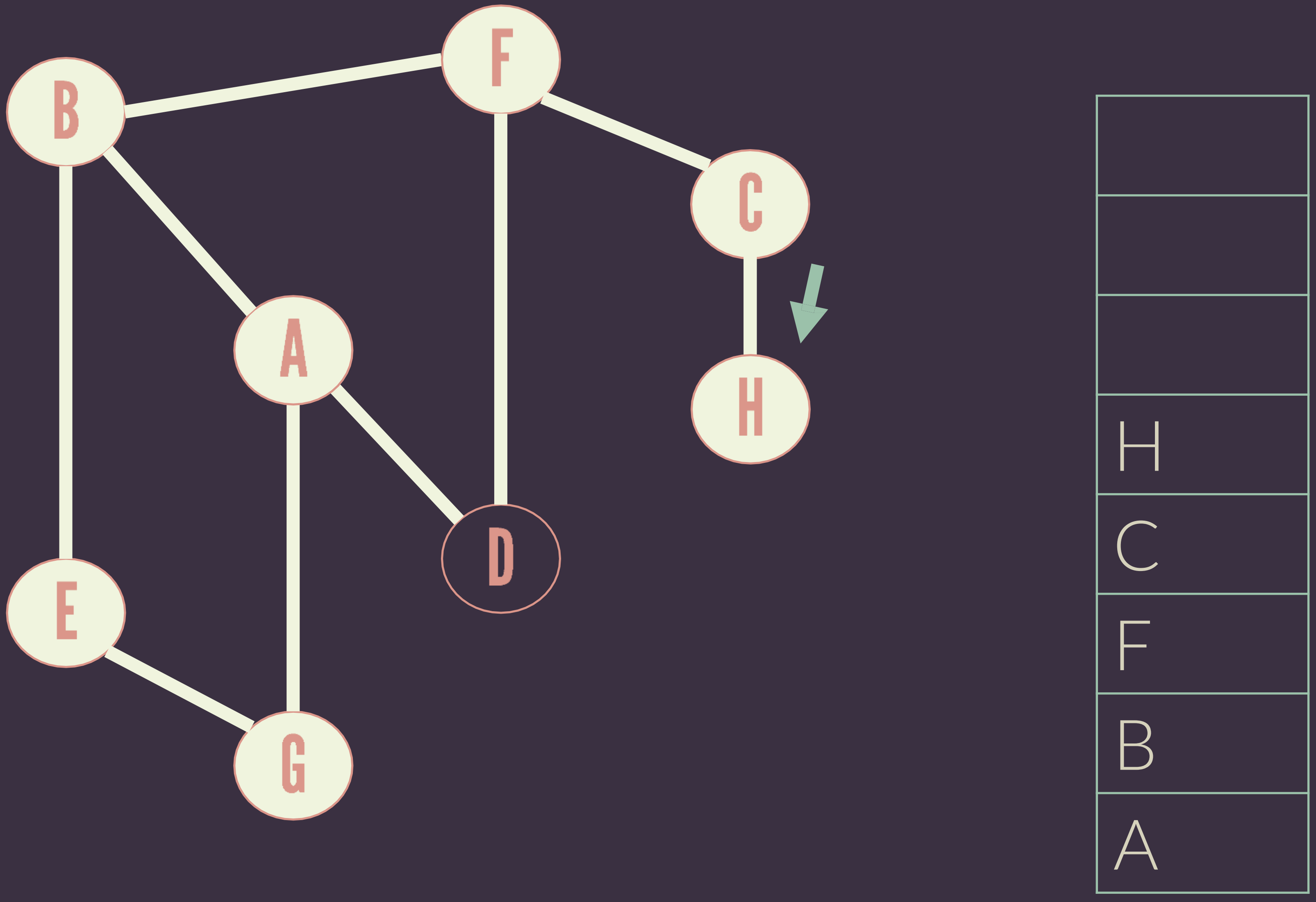
Result: A B E G



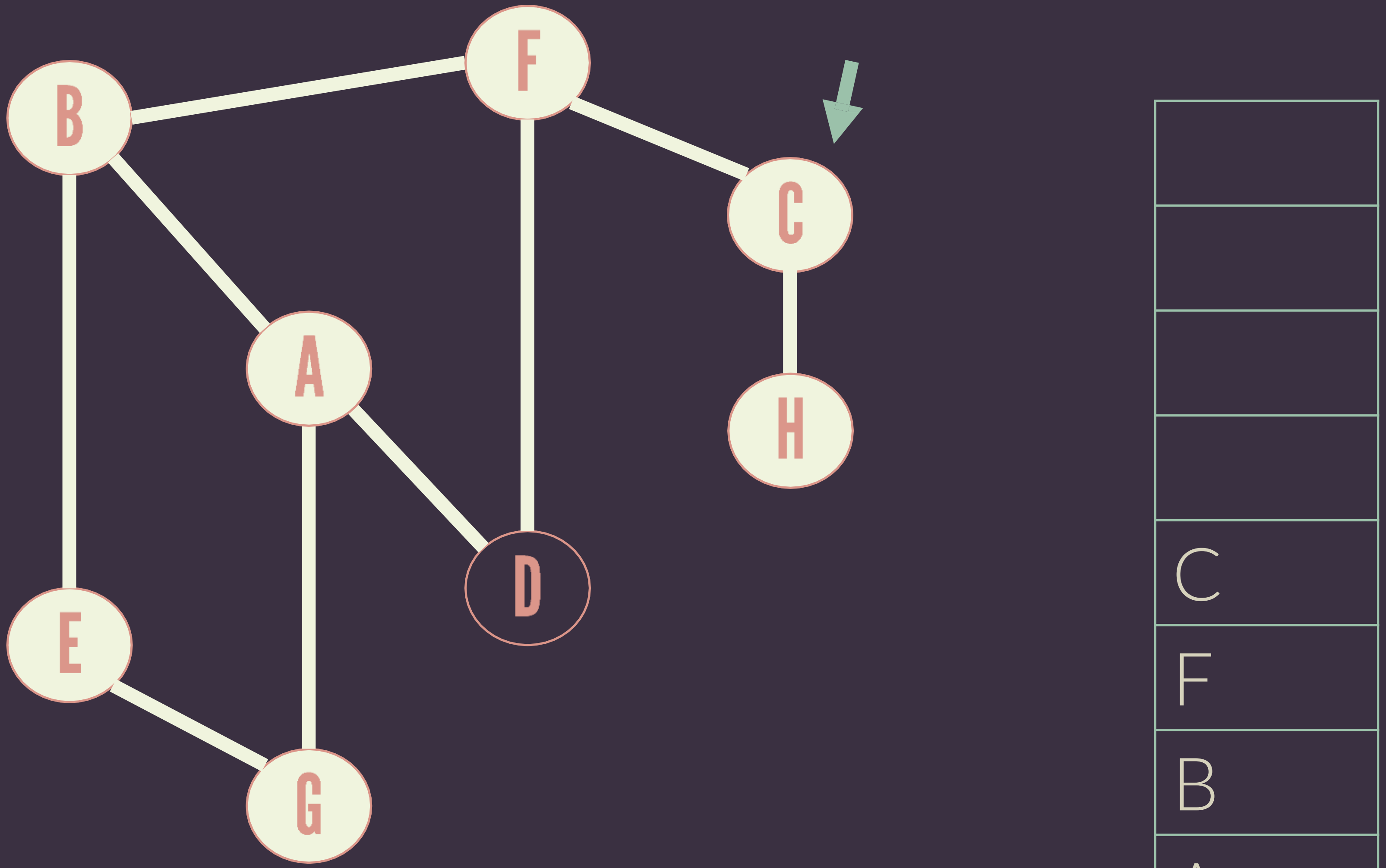
Result: A B E G F



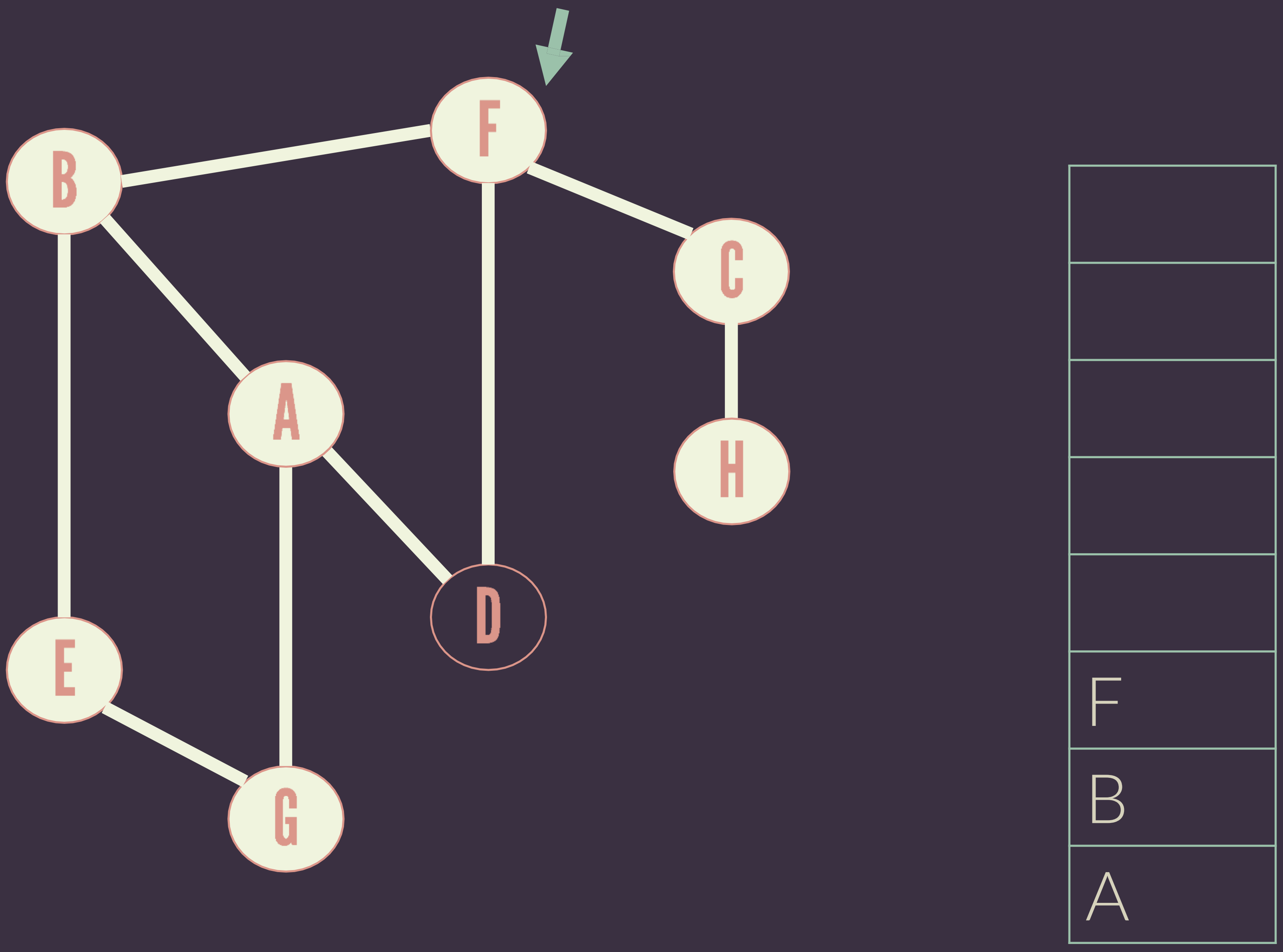
Result: A B E G F C



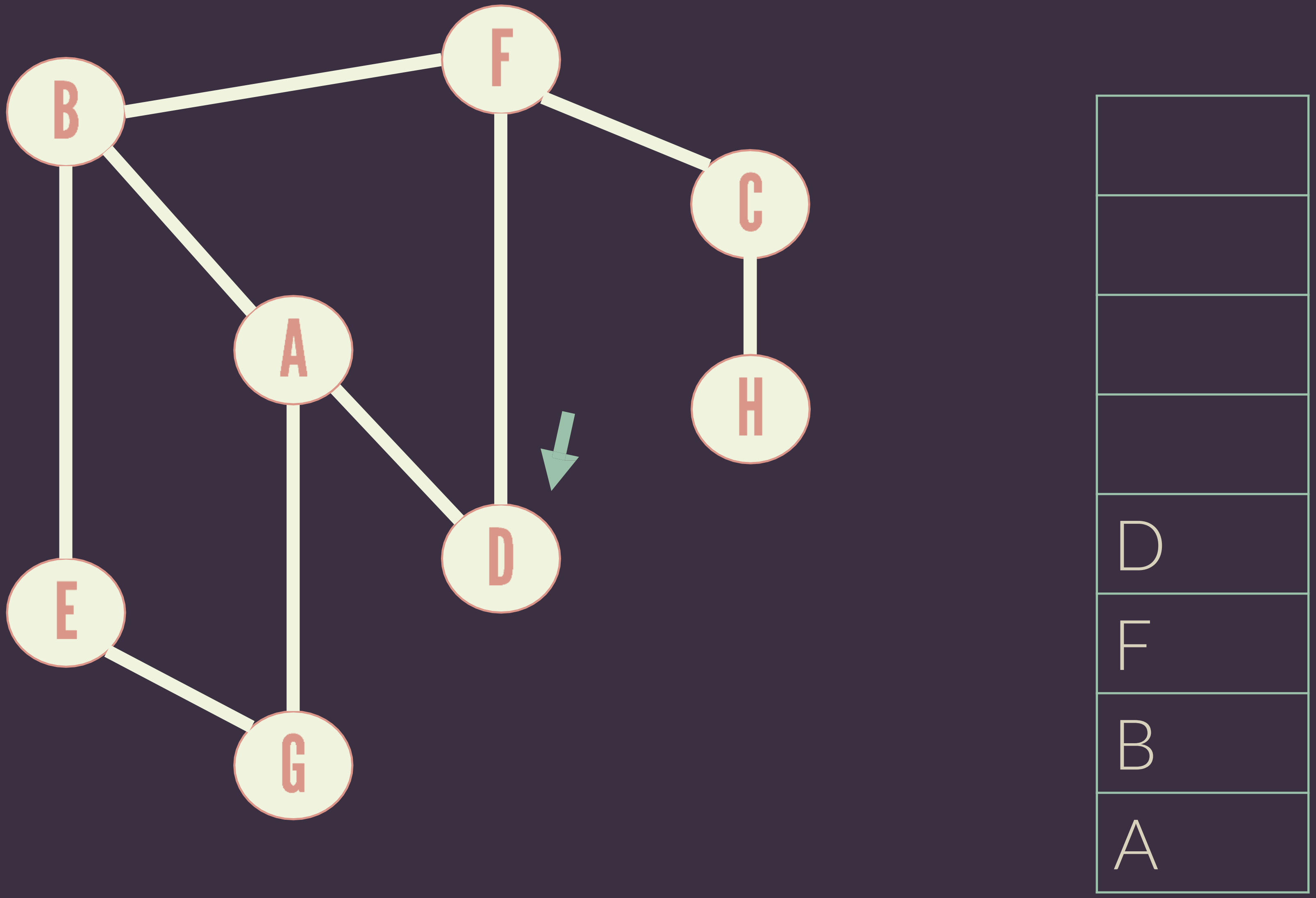
Result: A B E G F C H



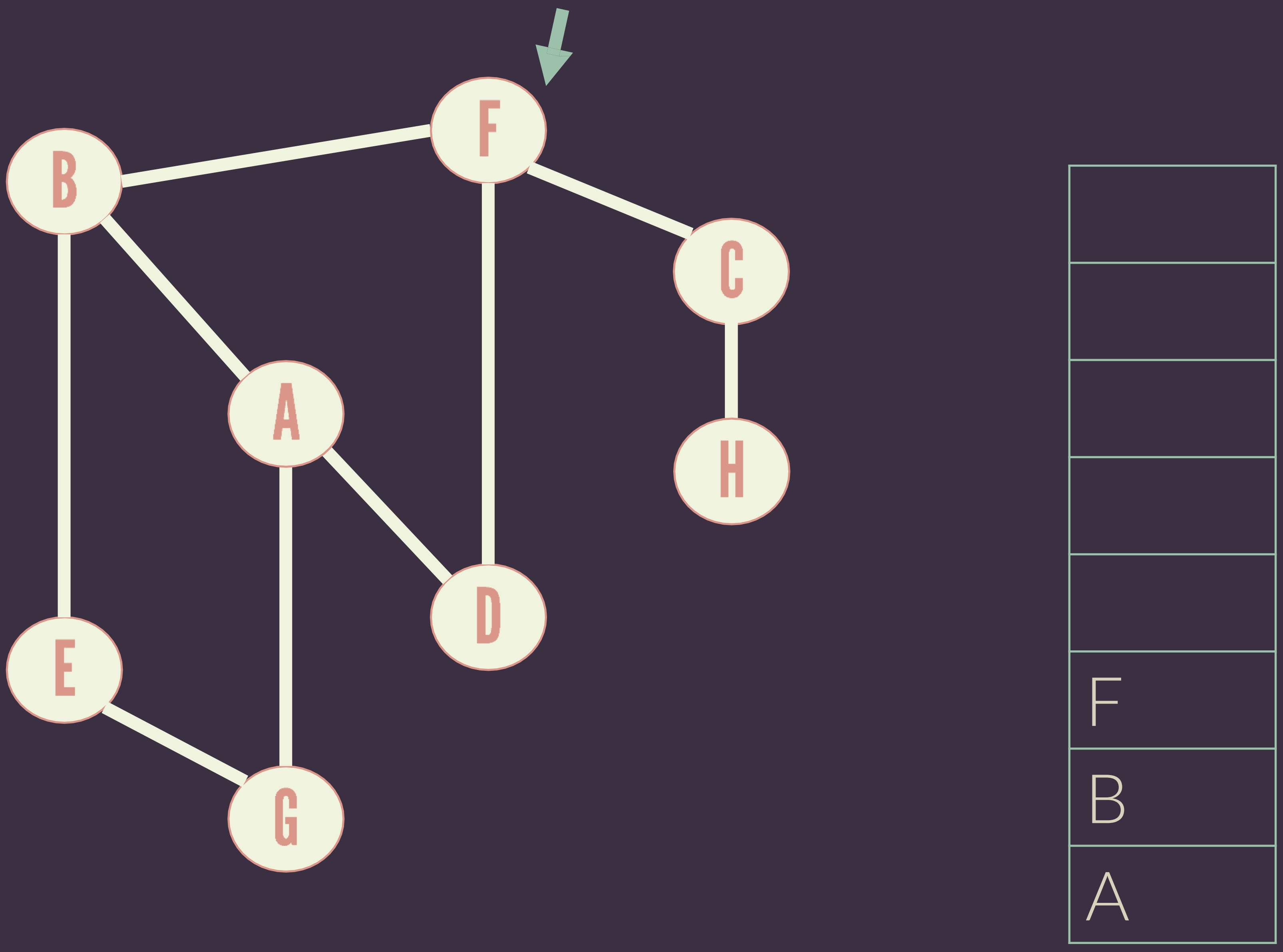
Result: A B E G F C H



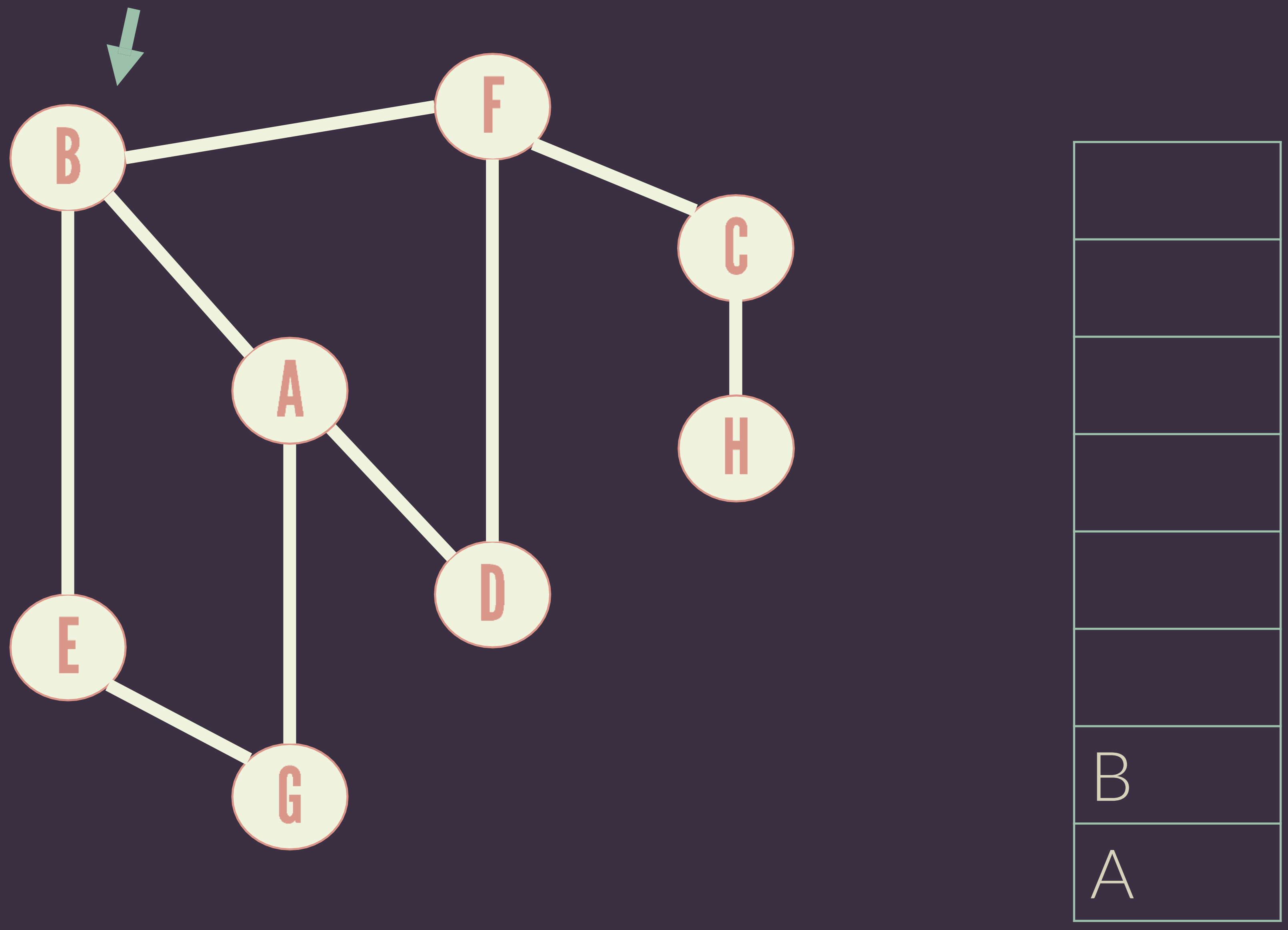
Result: A B E G F C H



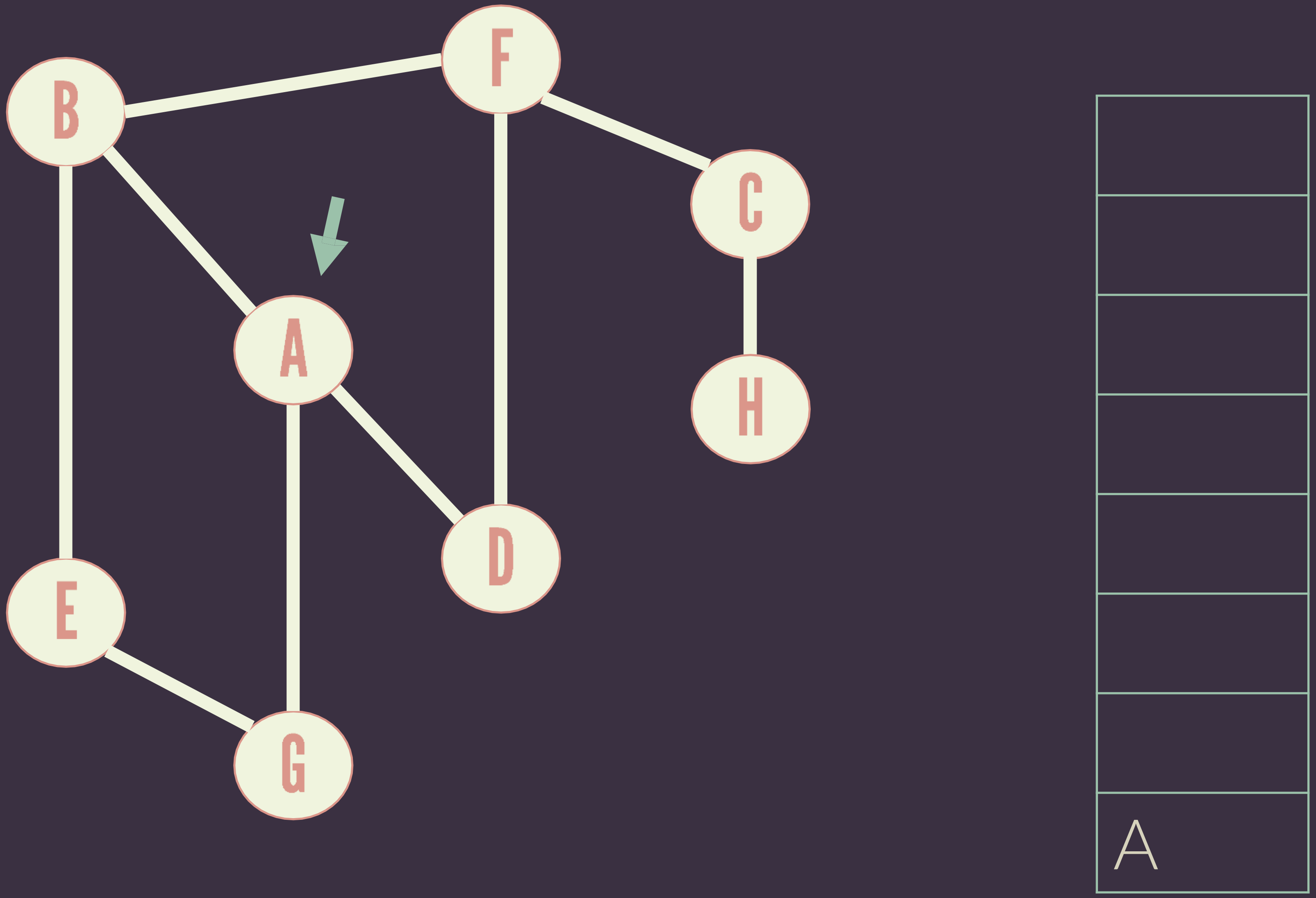
Result: A B E G F C H D



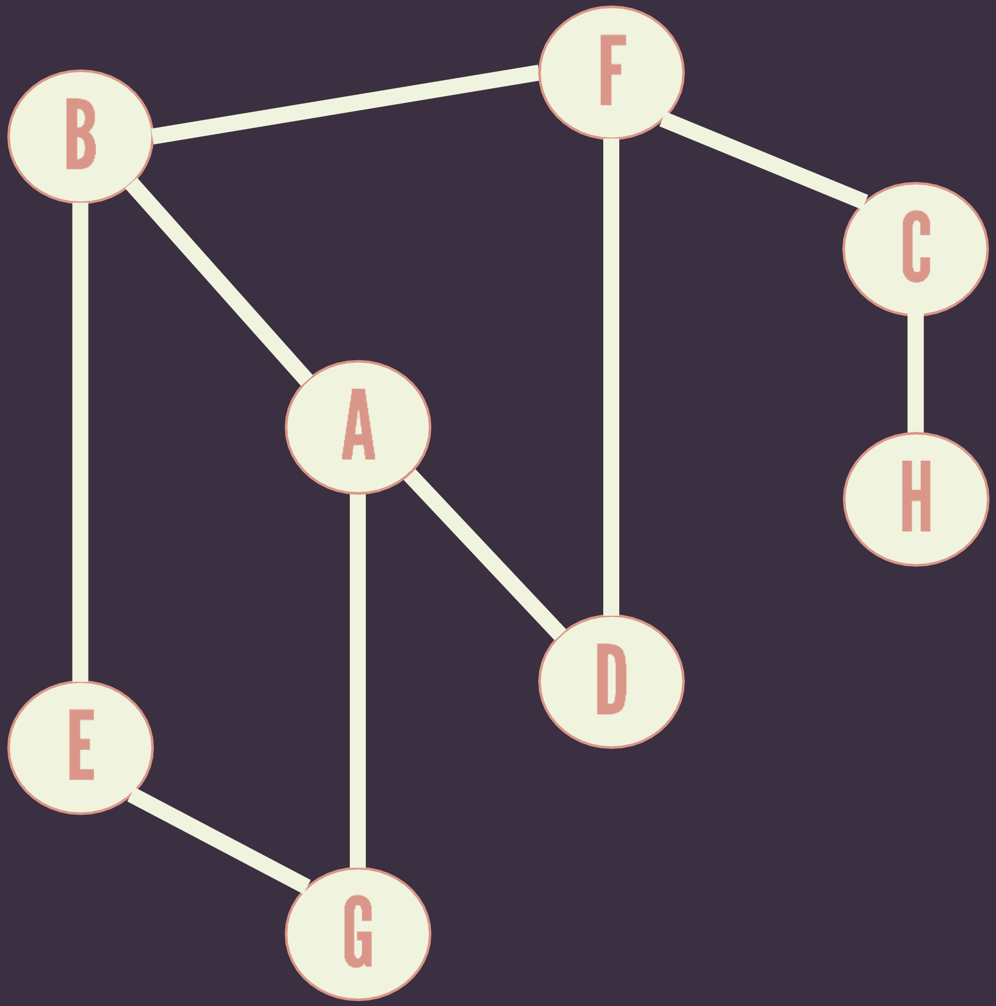
Result: A B E G F C H D



Result: A B E G F C H D



Result: A B E G F C H D



Result: A B E G F C H D

DFS applications

Detecting cycle in a graph.

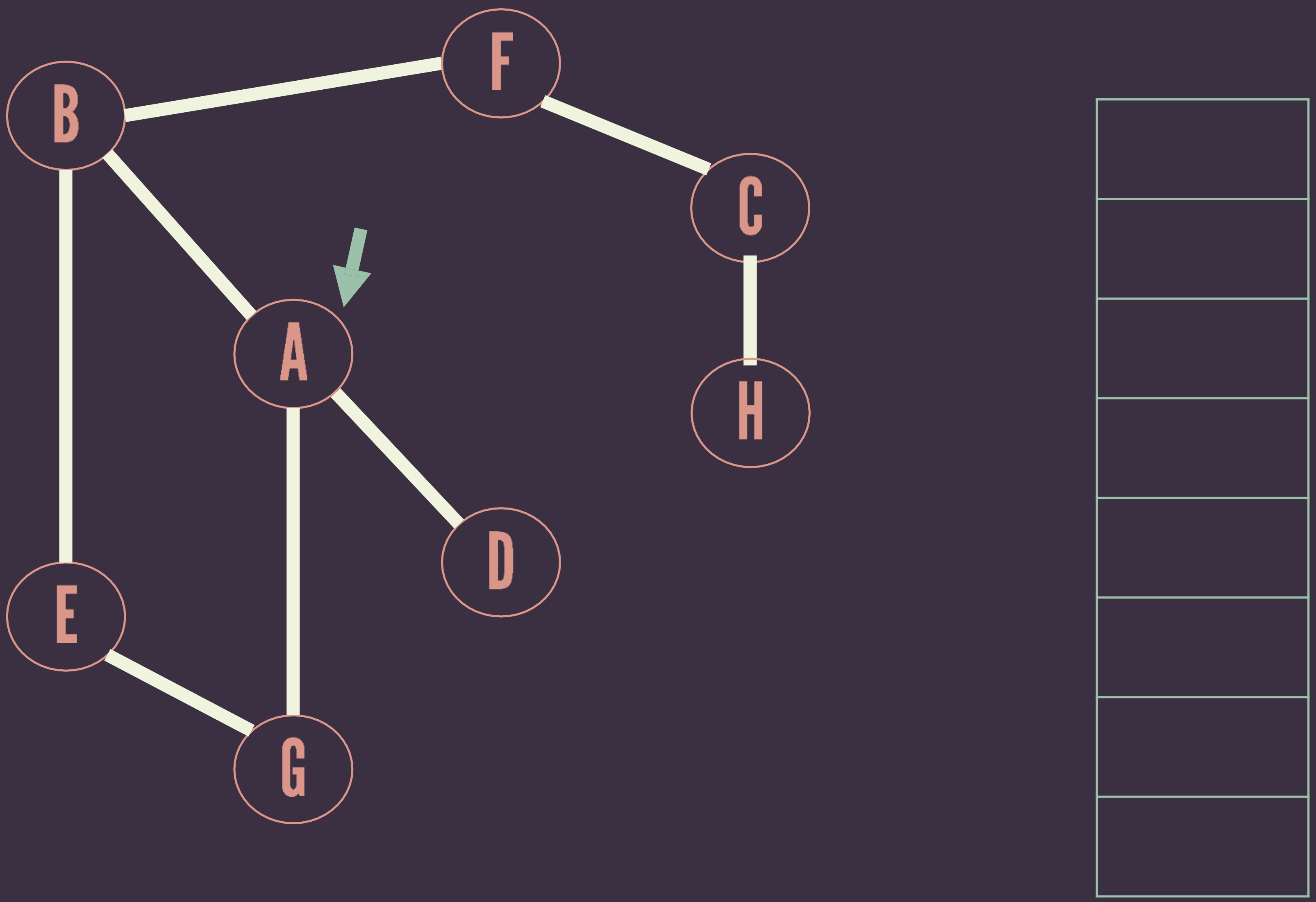
Topological sorting.

Path finding.

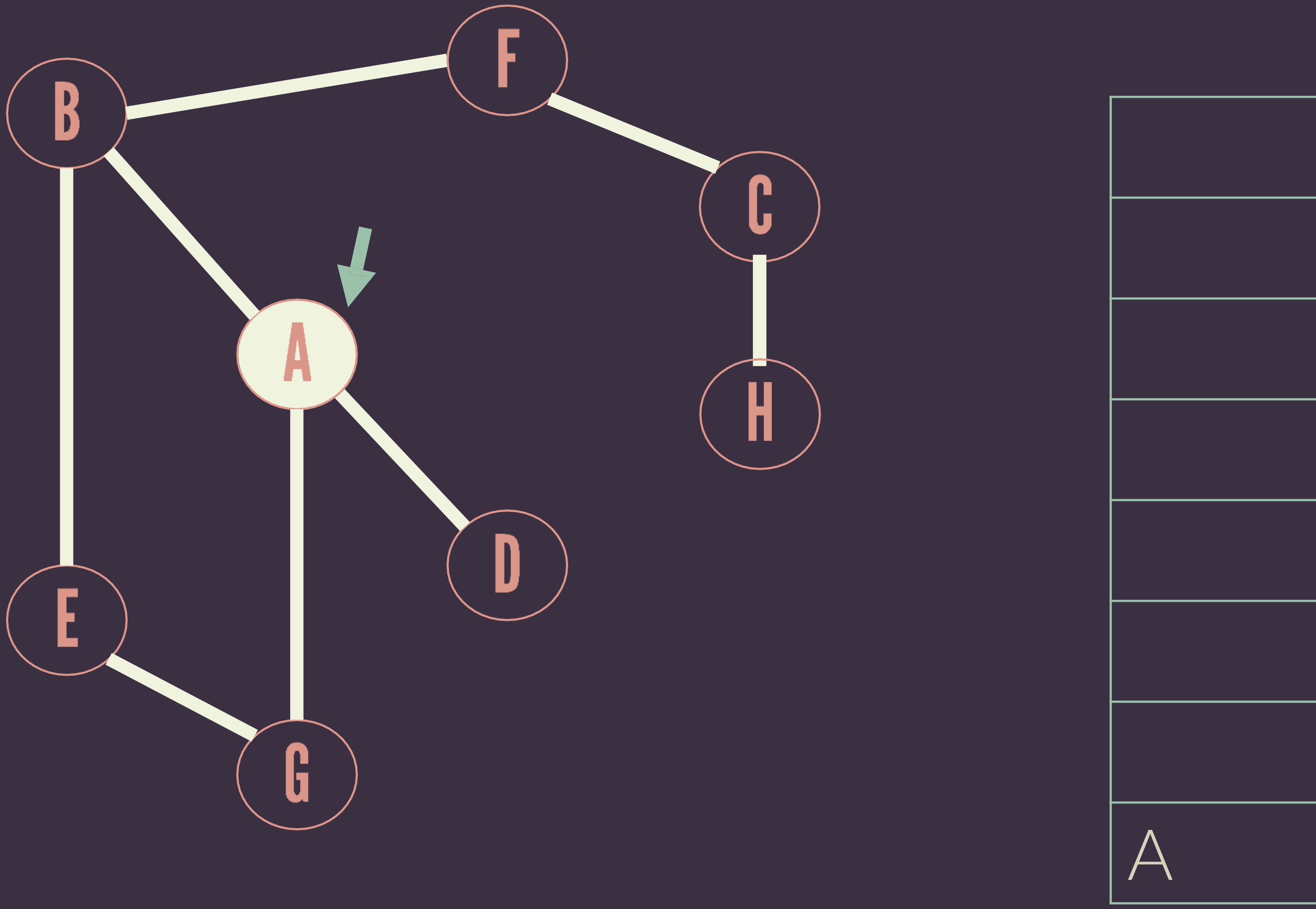
Spanning trees.

DFS
detecting cycle in a graph

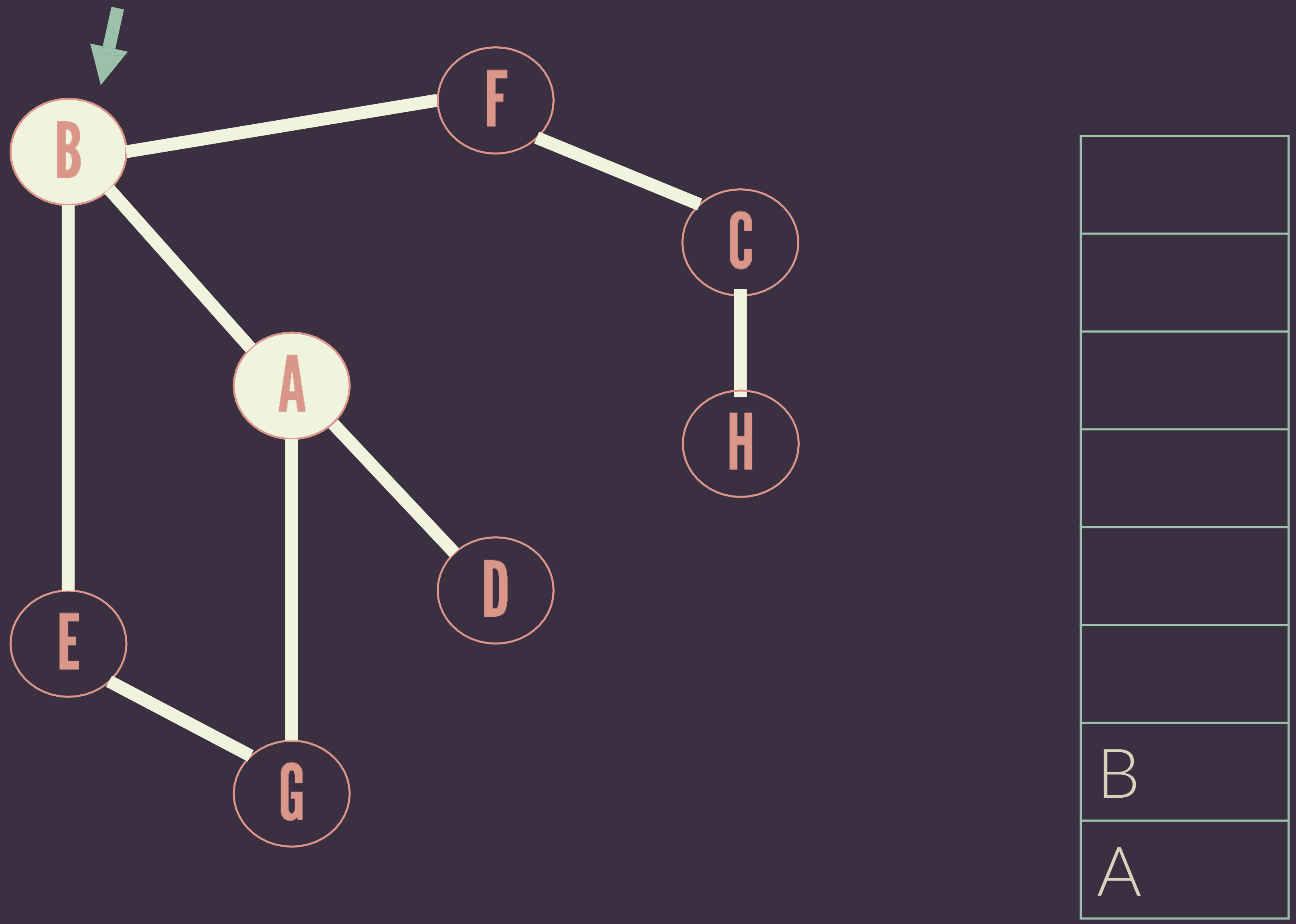
A graph has a cycle if and only if we can find a back edge during DFS.



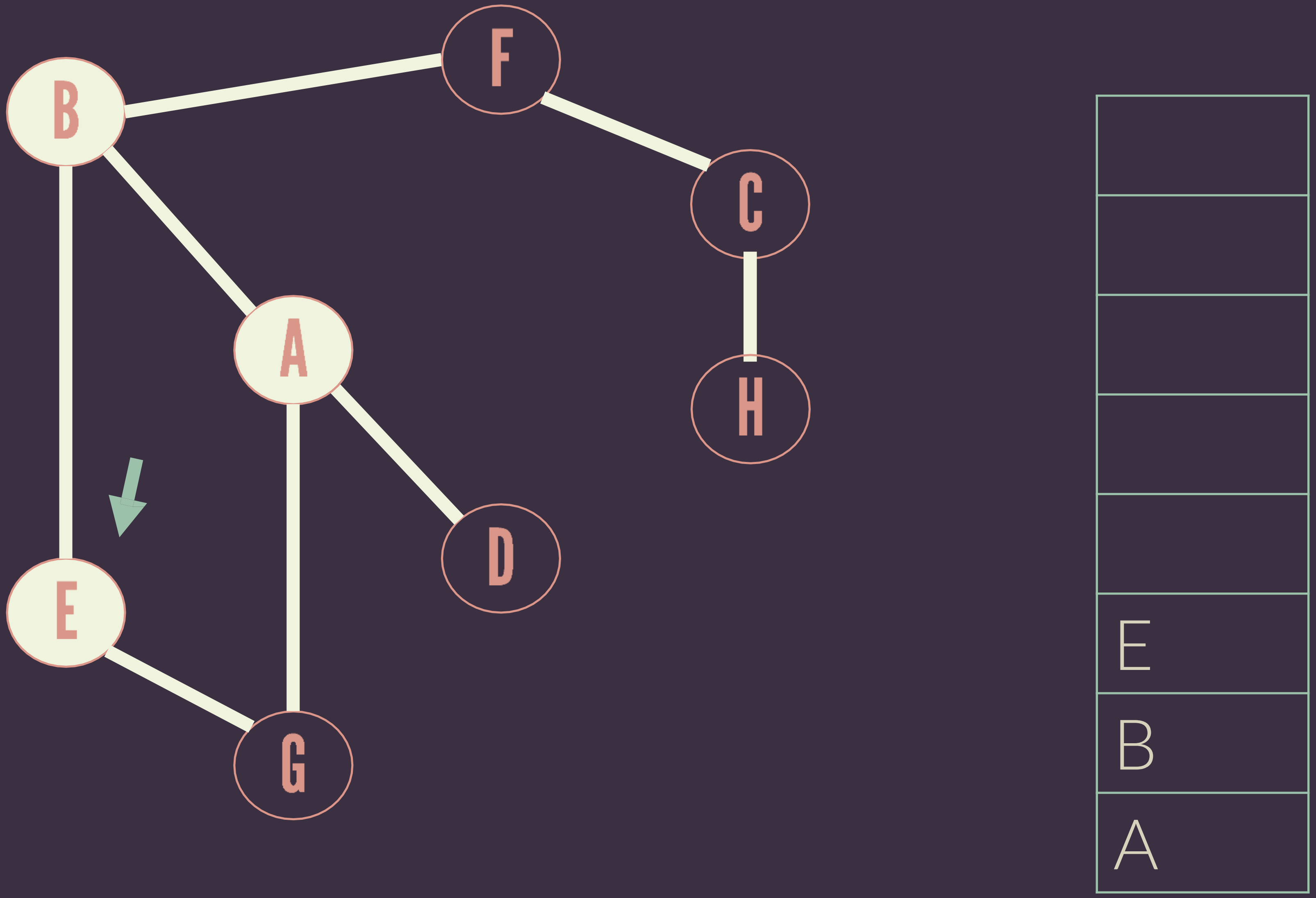
Result/Visited:



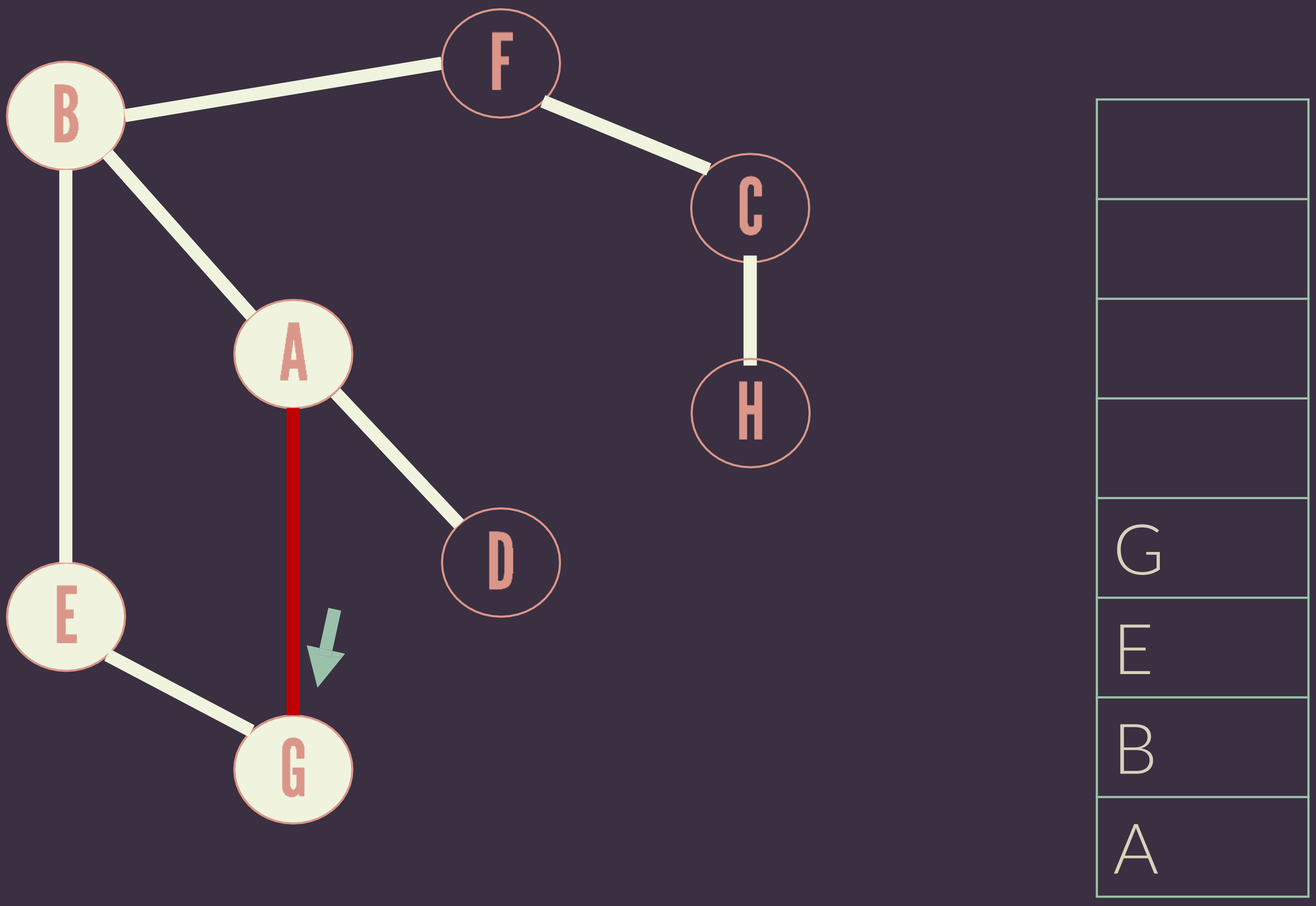
Result/Visited: A



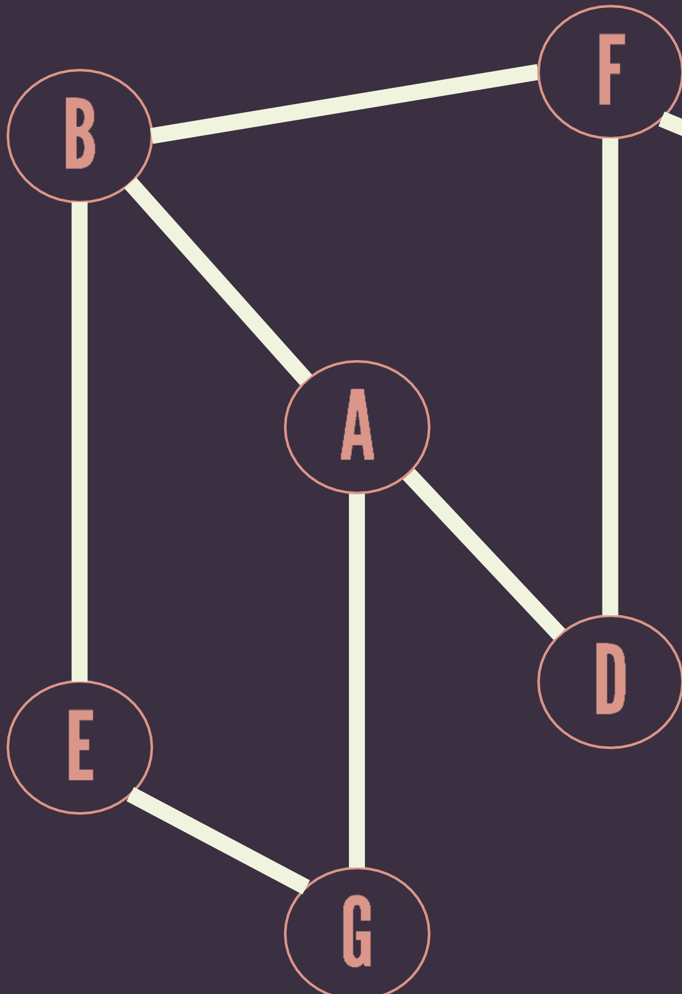
Result/Visited: A B

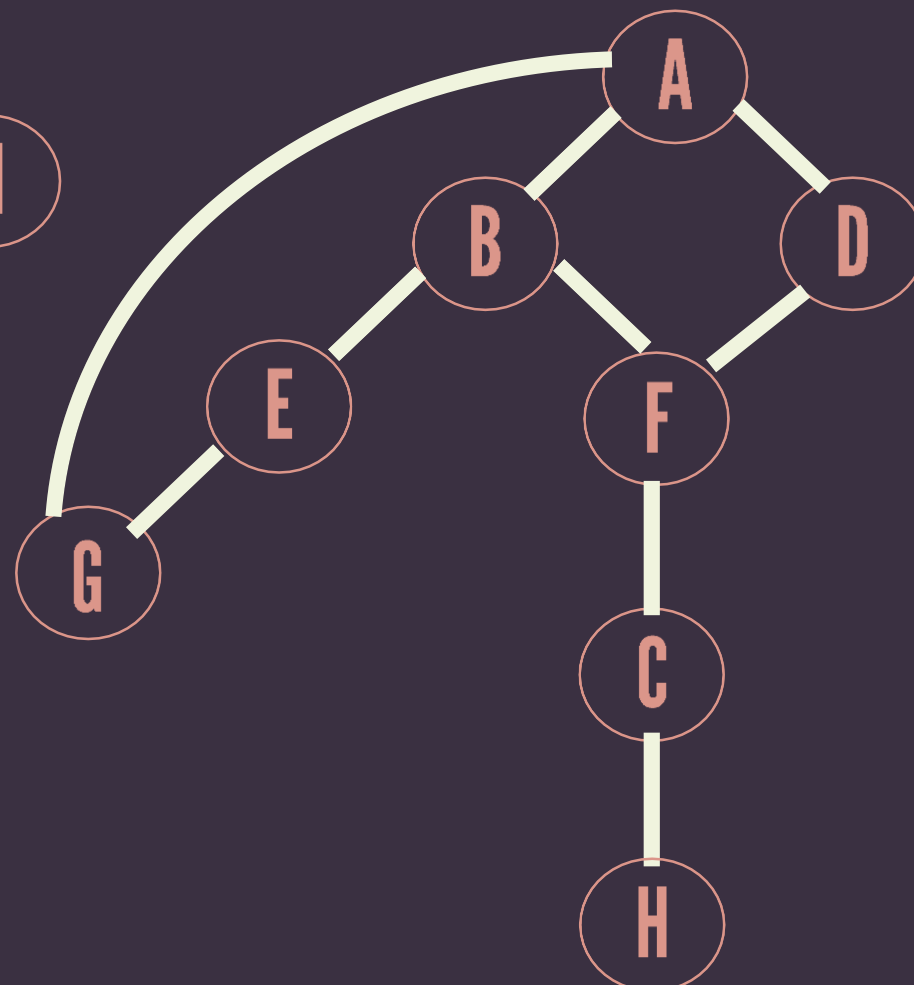


Result/Visited: A B E



Result/Visited: A B E G

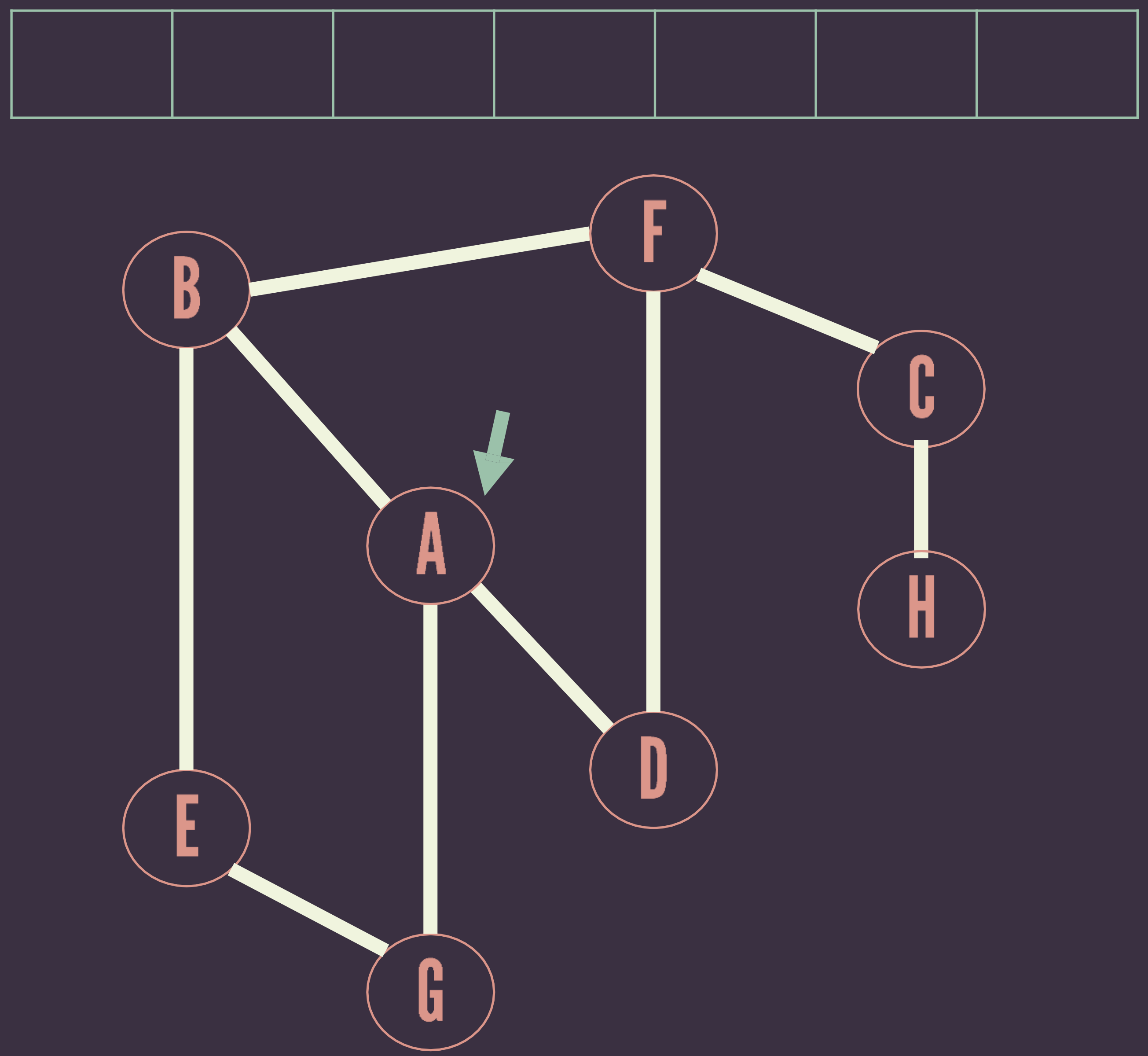

F
B
C
A
H
D
E
G


A
B
D
E
F
G
C
H

BREADTH FIRST search

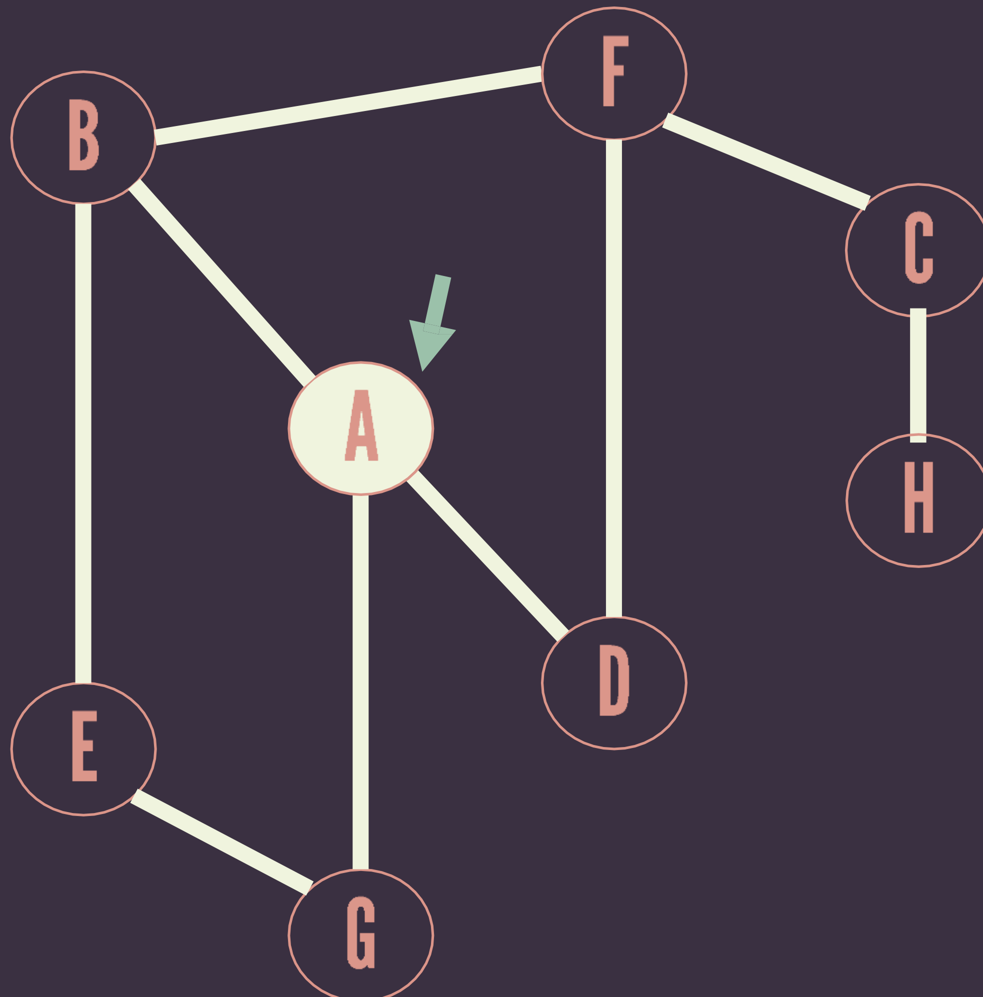
Generalization of level-order traversal of trees.

```
void BFS( vertex v, graph G ){
    queue Q;
    enqueue(v,Q);
    while queue Q is not empty{
        v = dequeue(Q);
        if ( !visited[v] ){
            print v;
            visited[v] = TRUE;
            for each w adjacent to v:
                if( !visited[w] )
                    enqueue(w, Q);
        }
    }
}
```

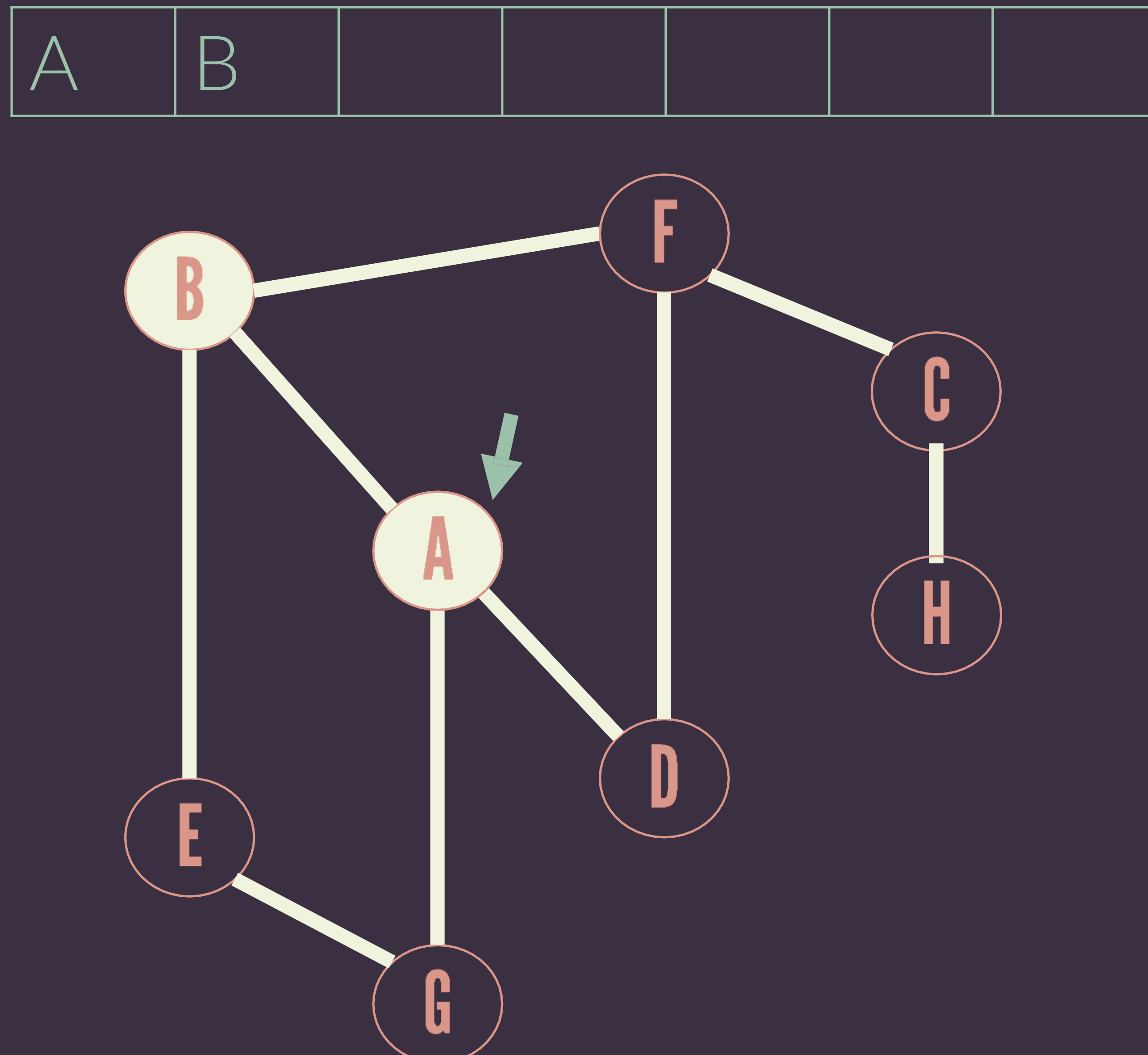


Result: A

A						

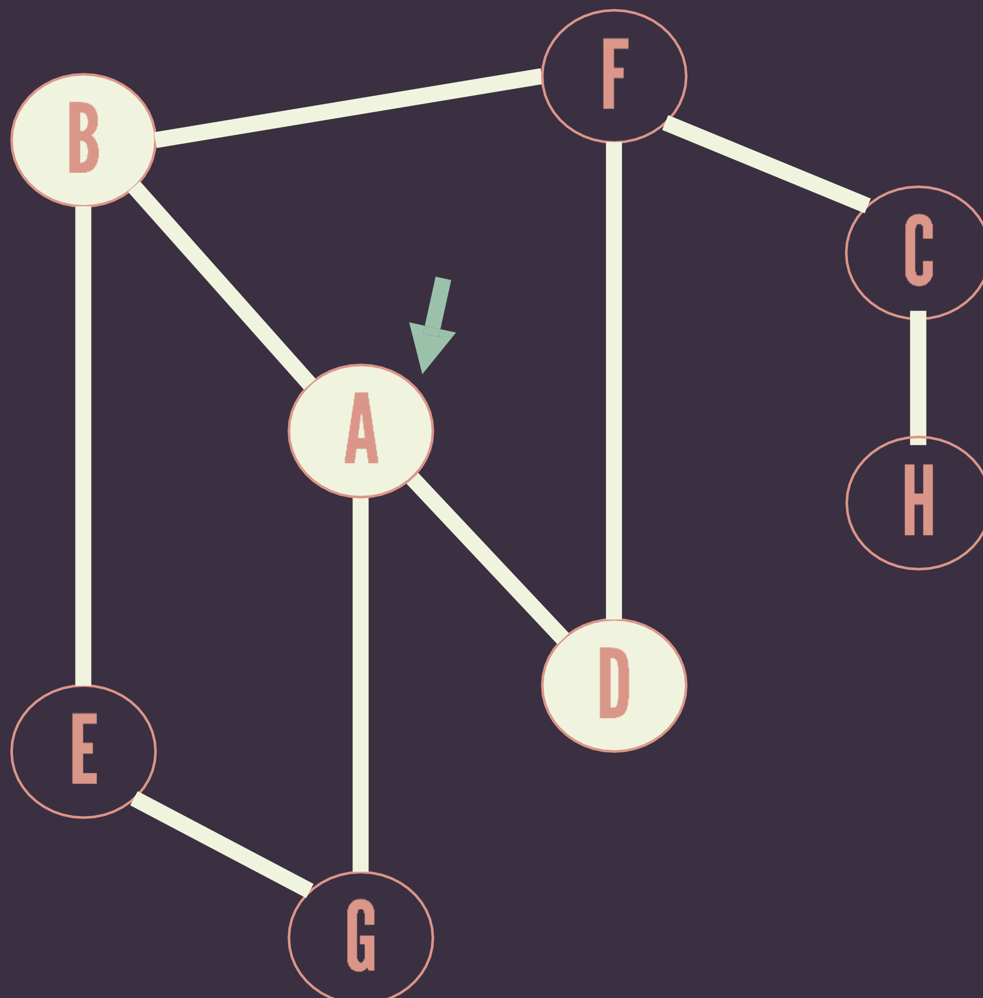


Result: A



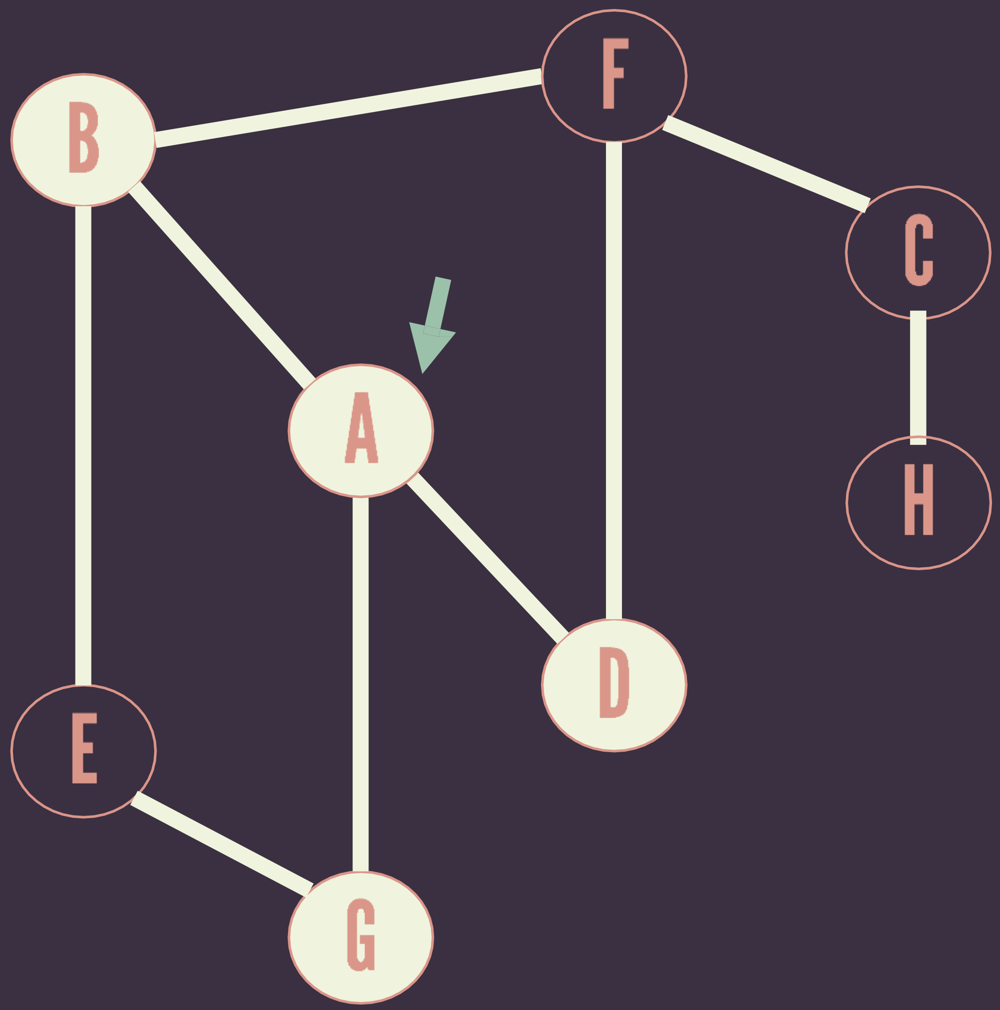
Result: A B

A	B	D				

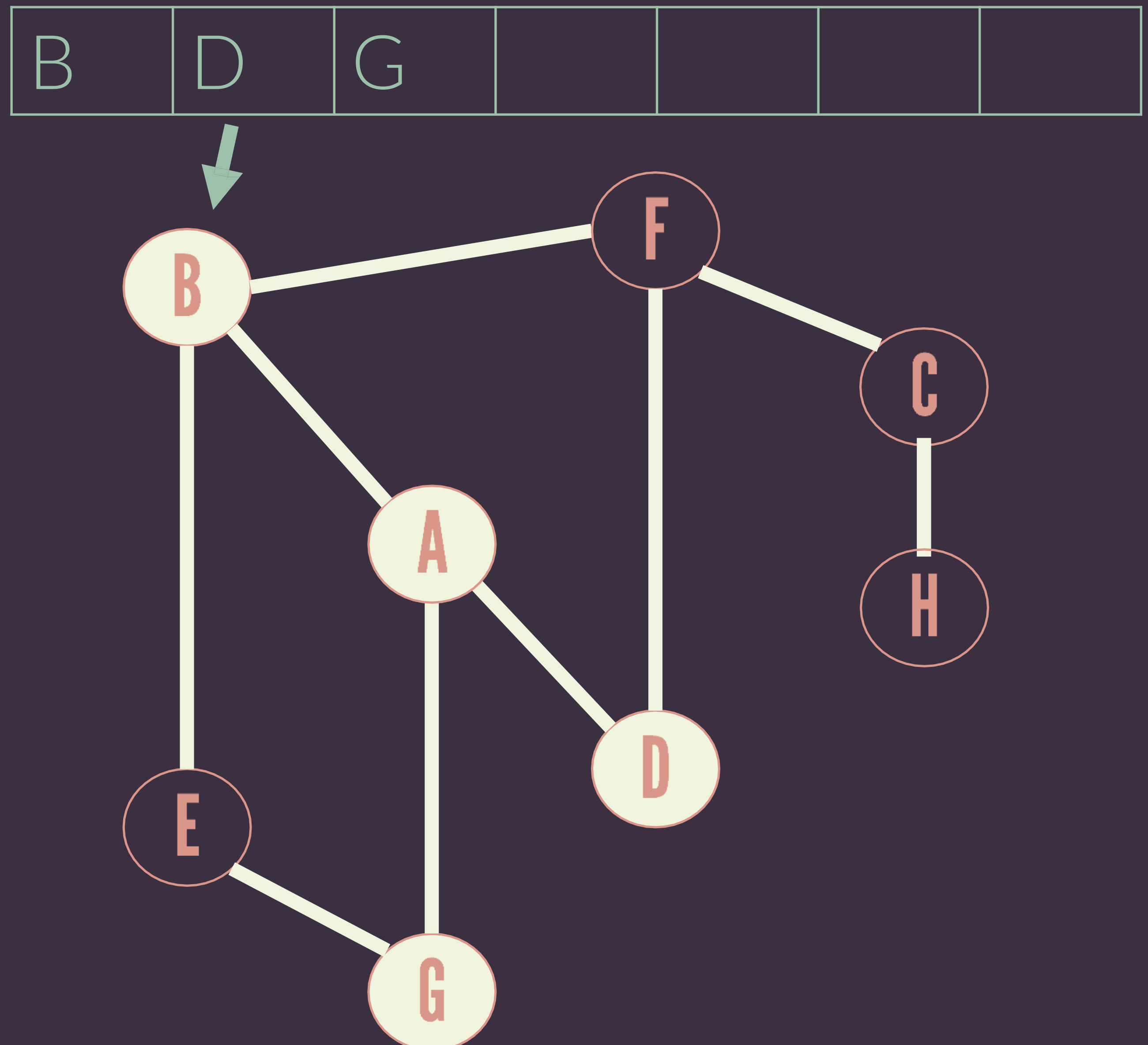


Result: A B D

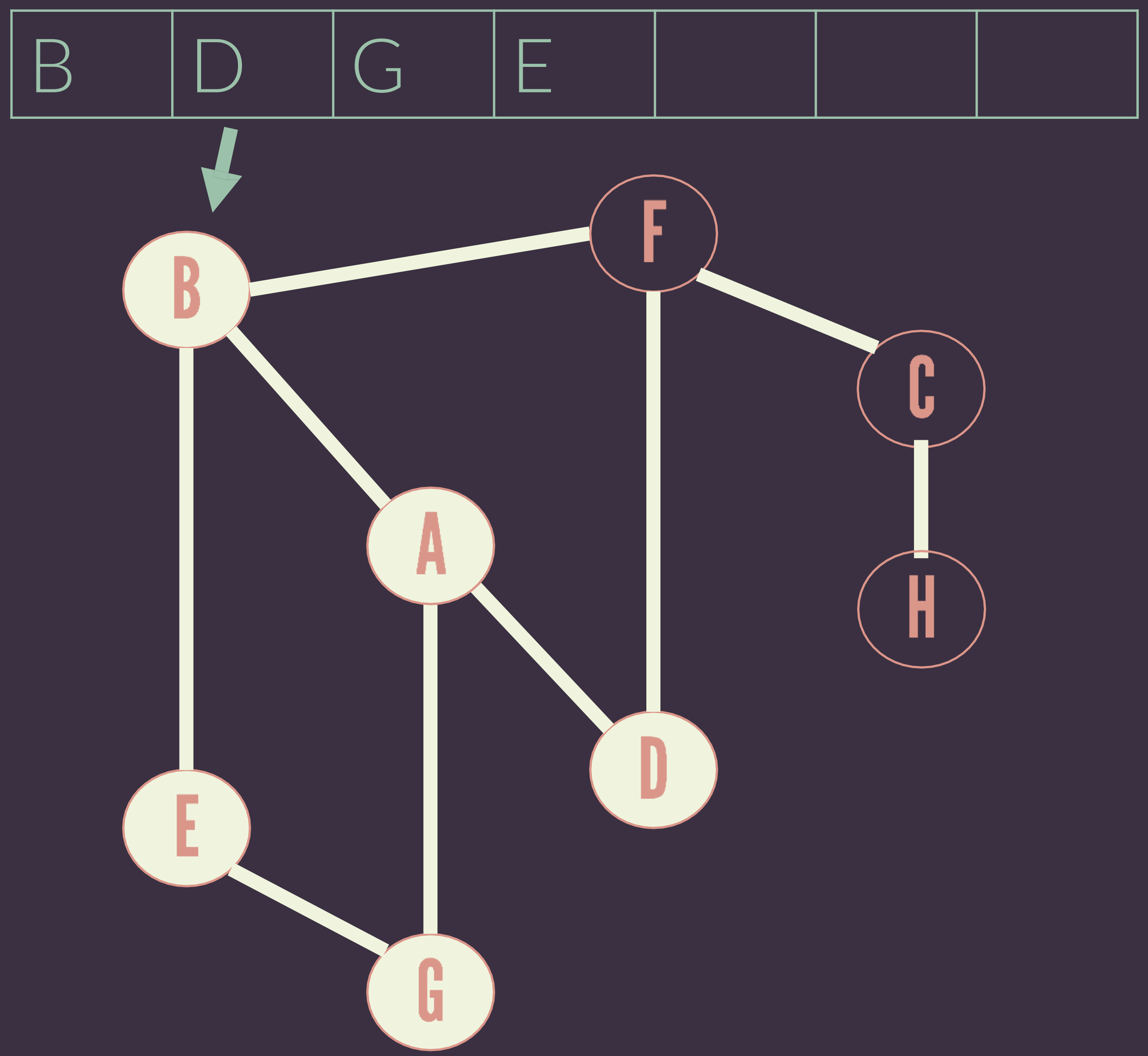
A	B	D	G			



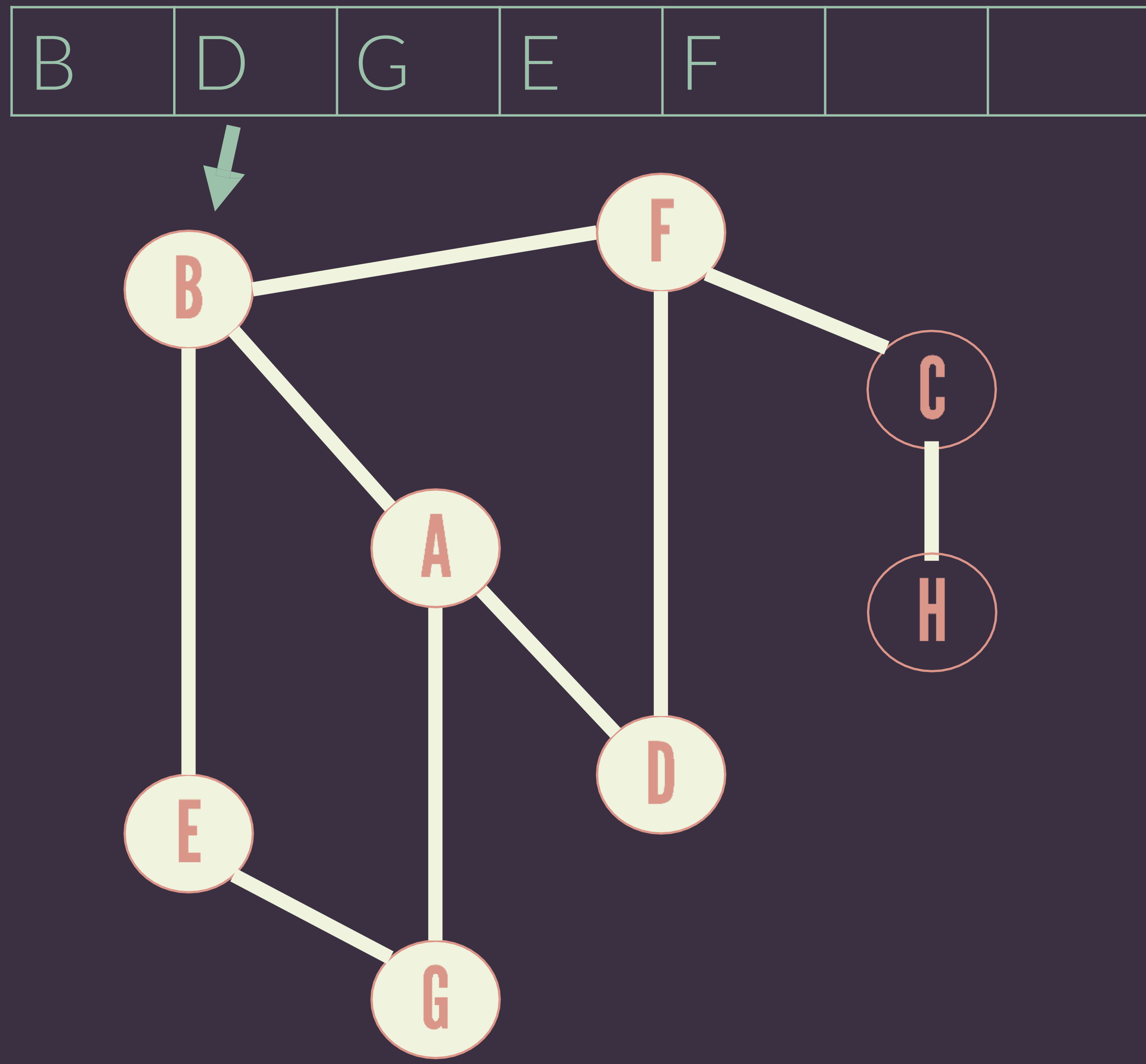
Result: A B D G



Result: A B D G

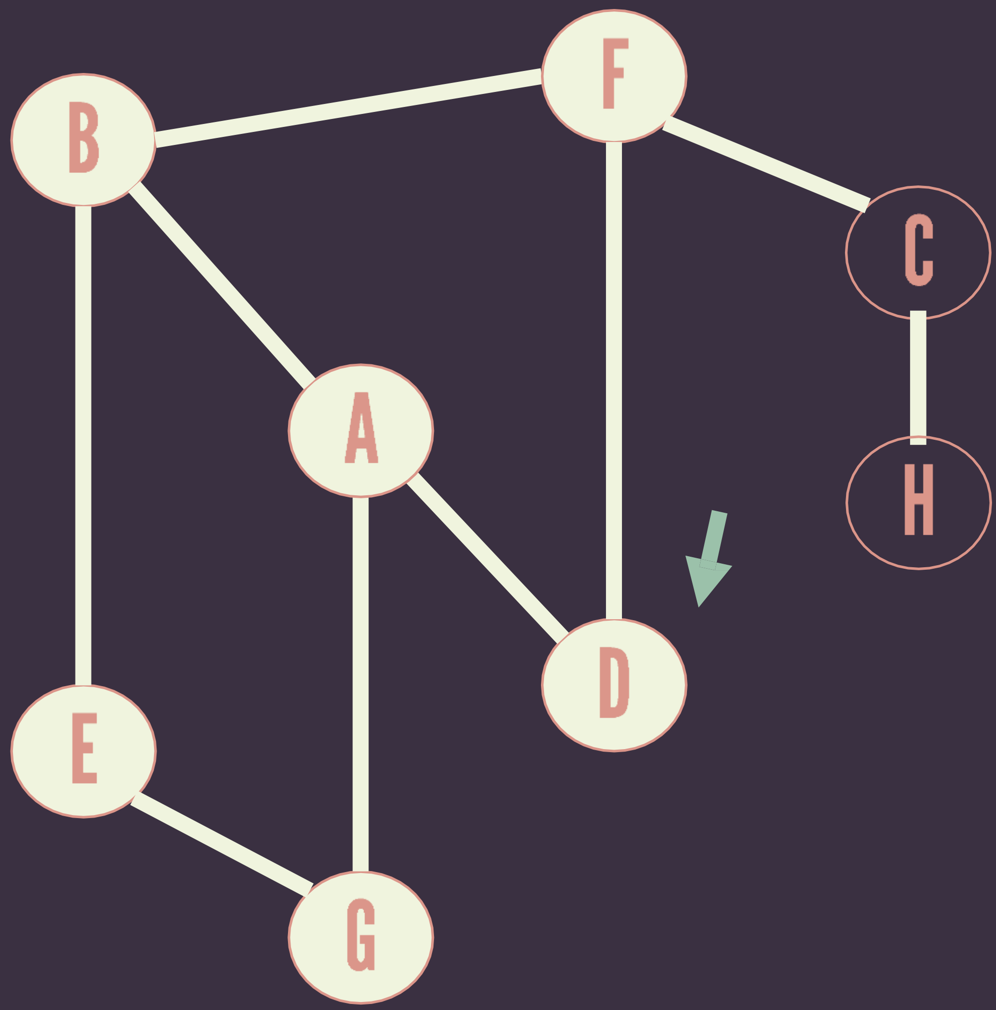


Result: A B D G E

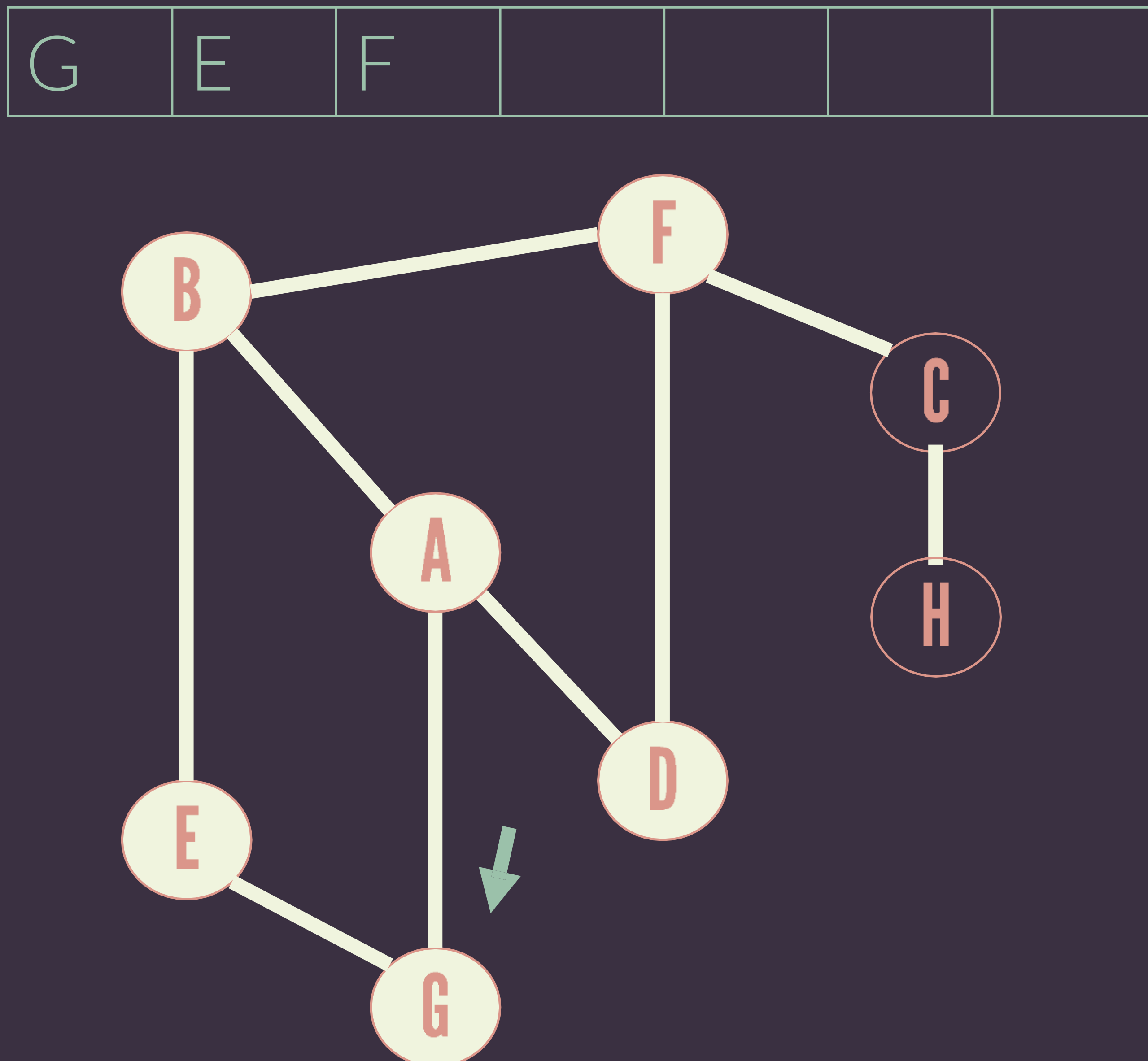


Result: A B D G E F

D	G	E	F			

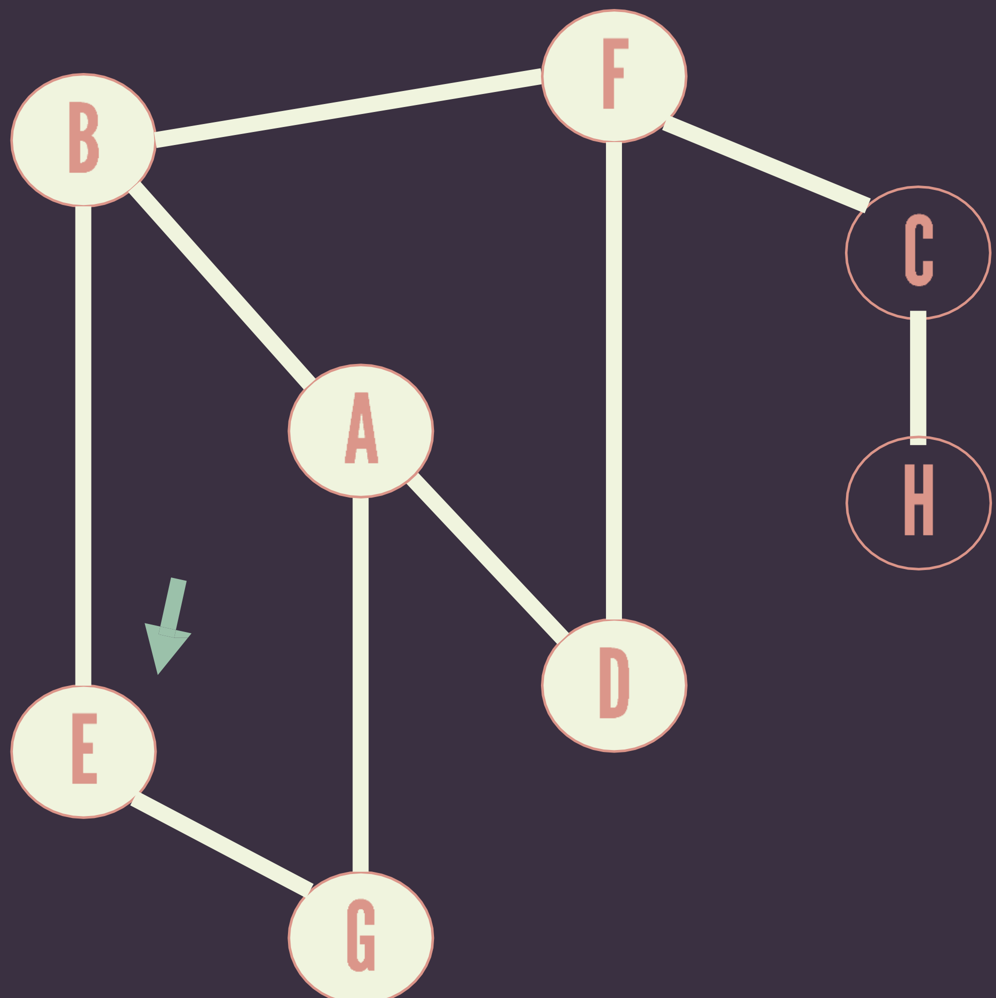


Result: A B D G E F

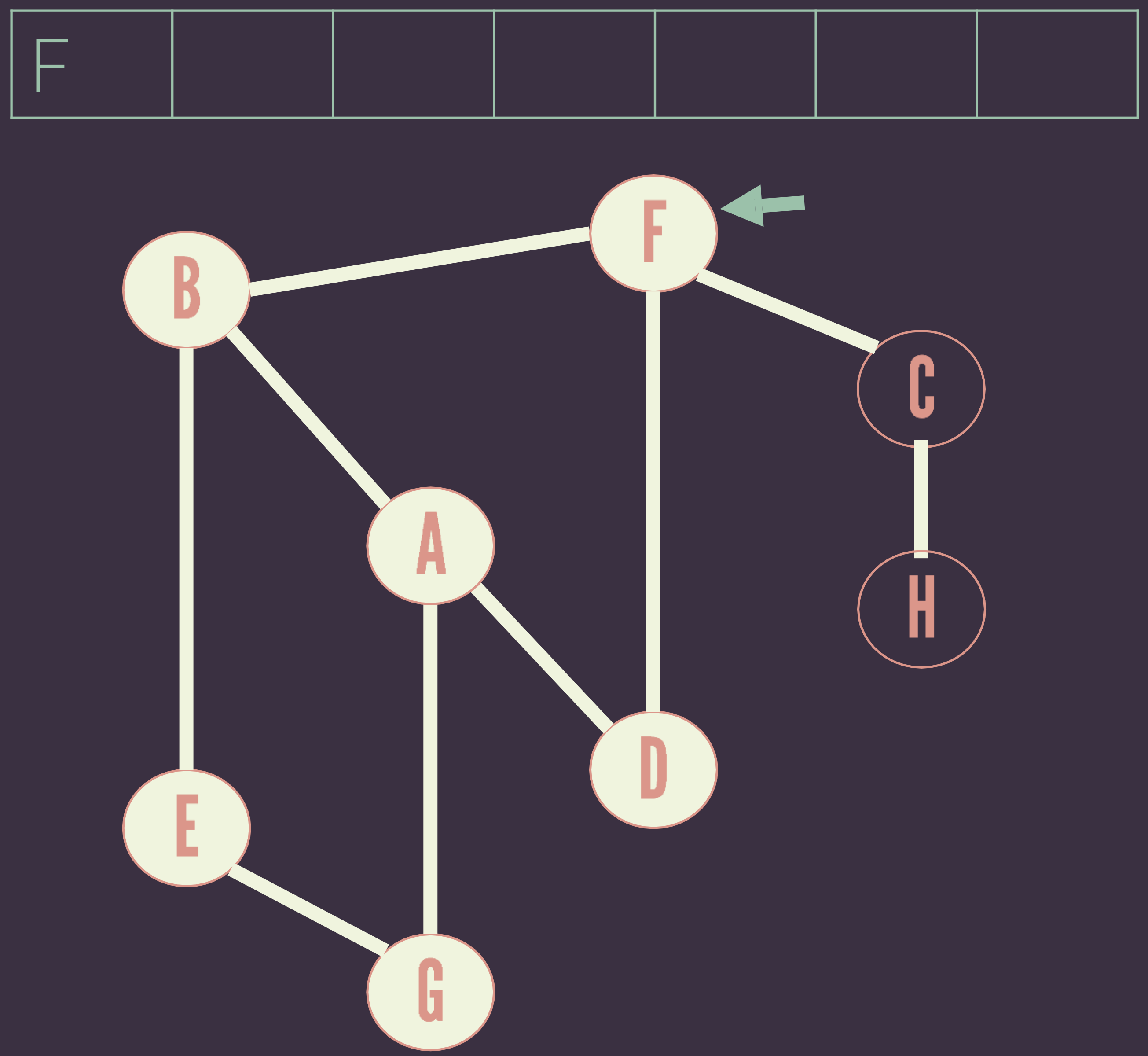


Result: A B D G E F

E	F					

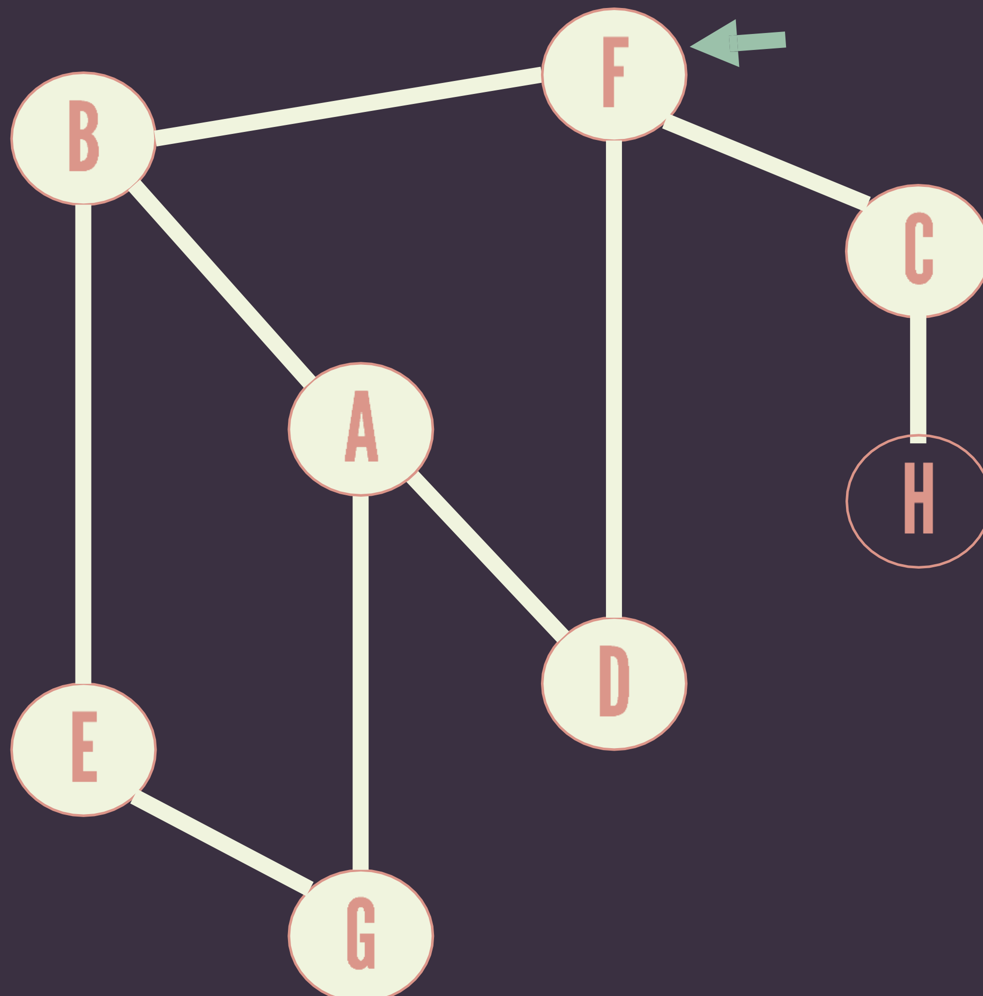


Result: A B D G E F

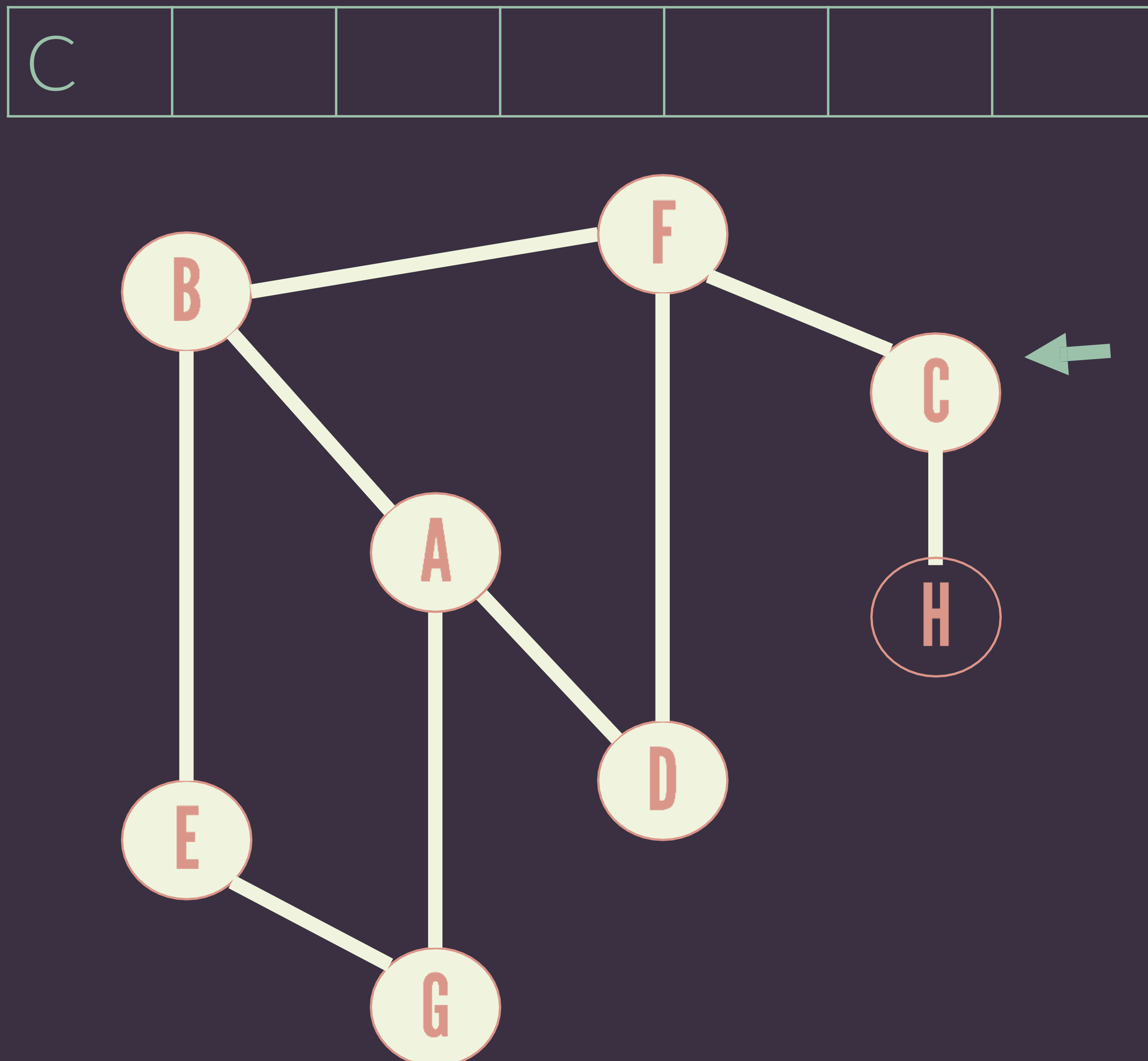


Result: A B D G E F

F	C					

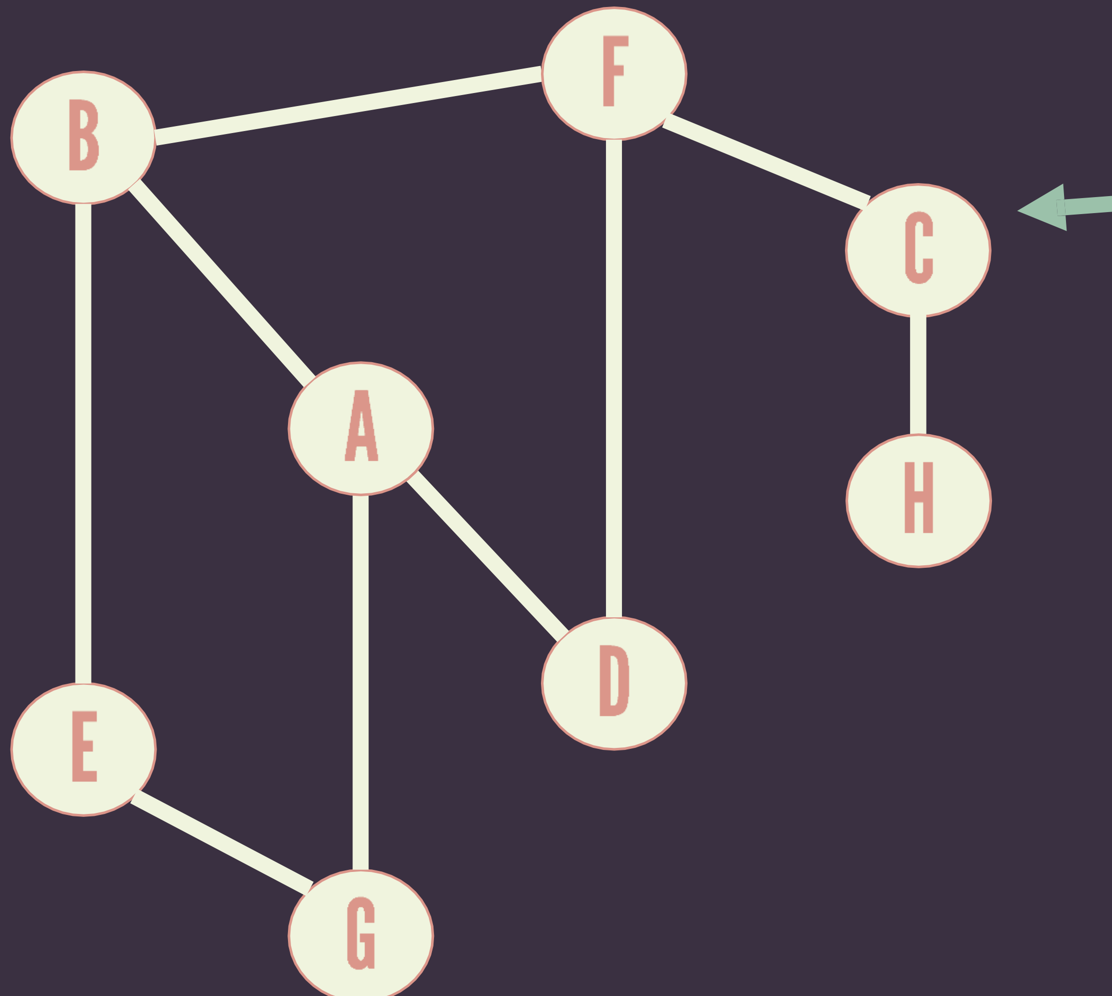


Result: A B D G E F C



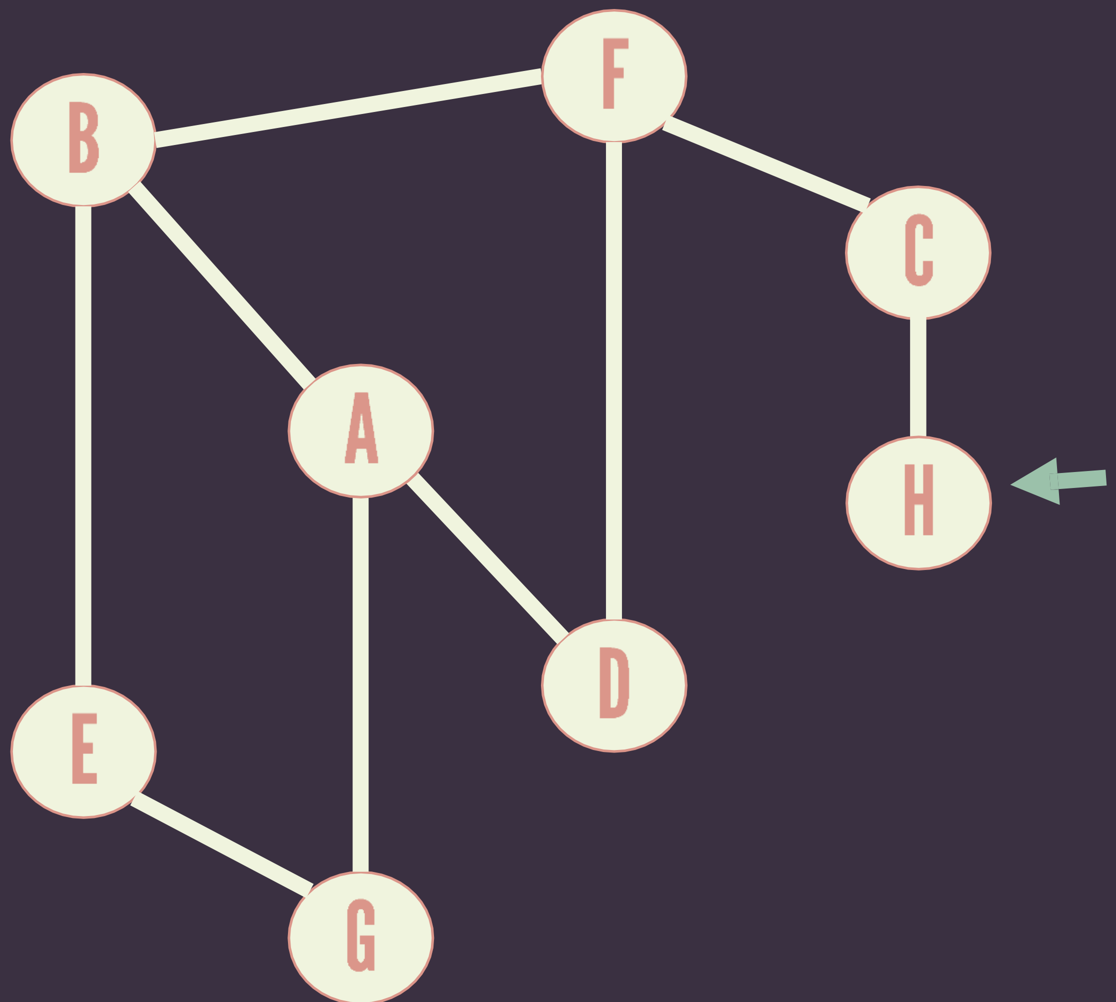
Result: A B D G E F C

C	H					

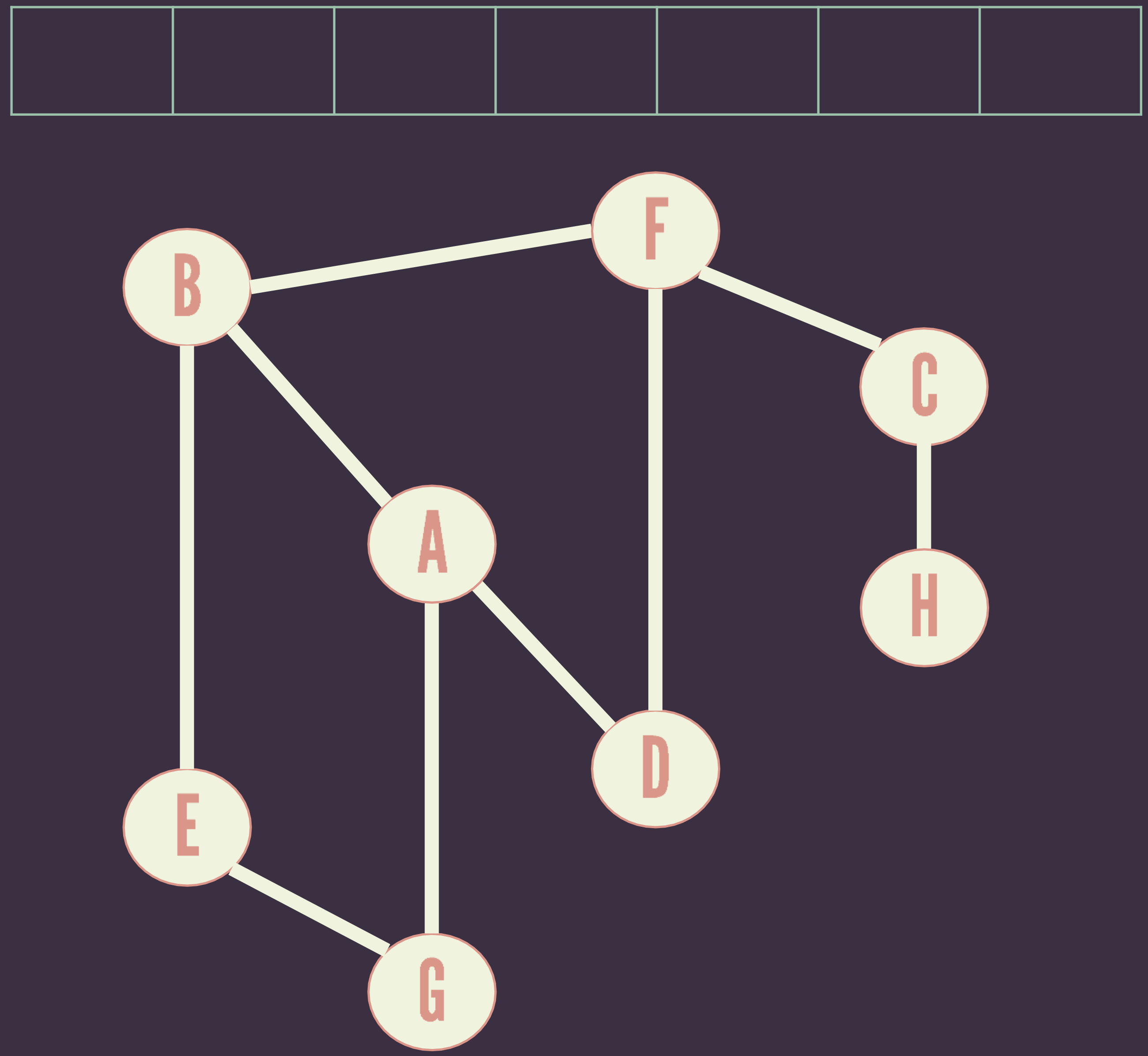


Result: A B D G E F C H

H						



Result: A B D G E F C H



Result: A B D G E F C H

BFS applications

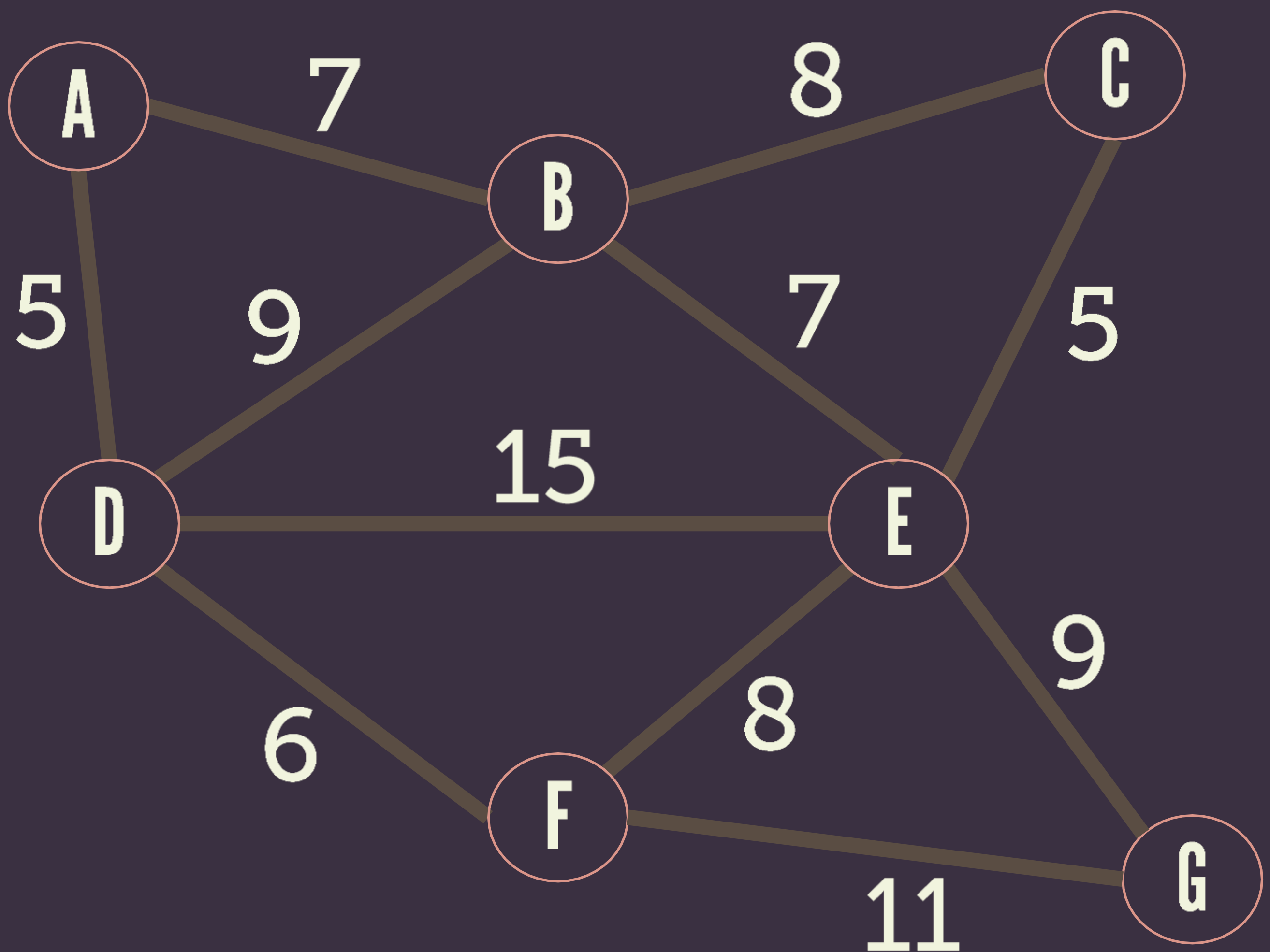
Finding nodes within one.
connected components.
Testing for bipartiteness.
Finding shortest paths.

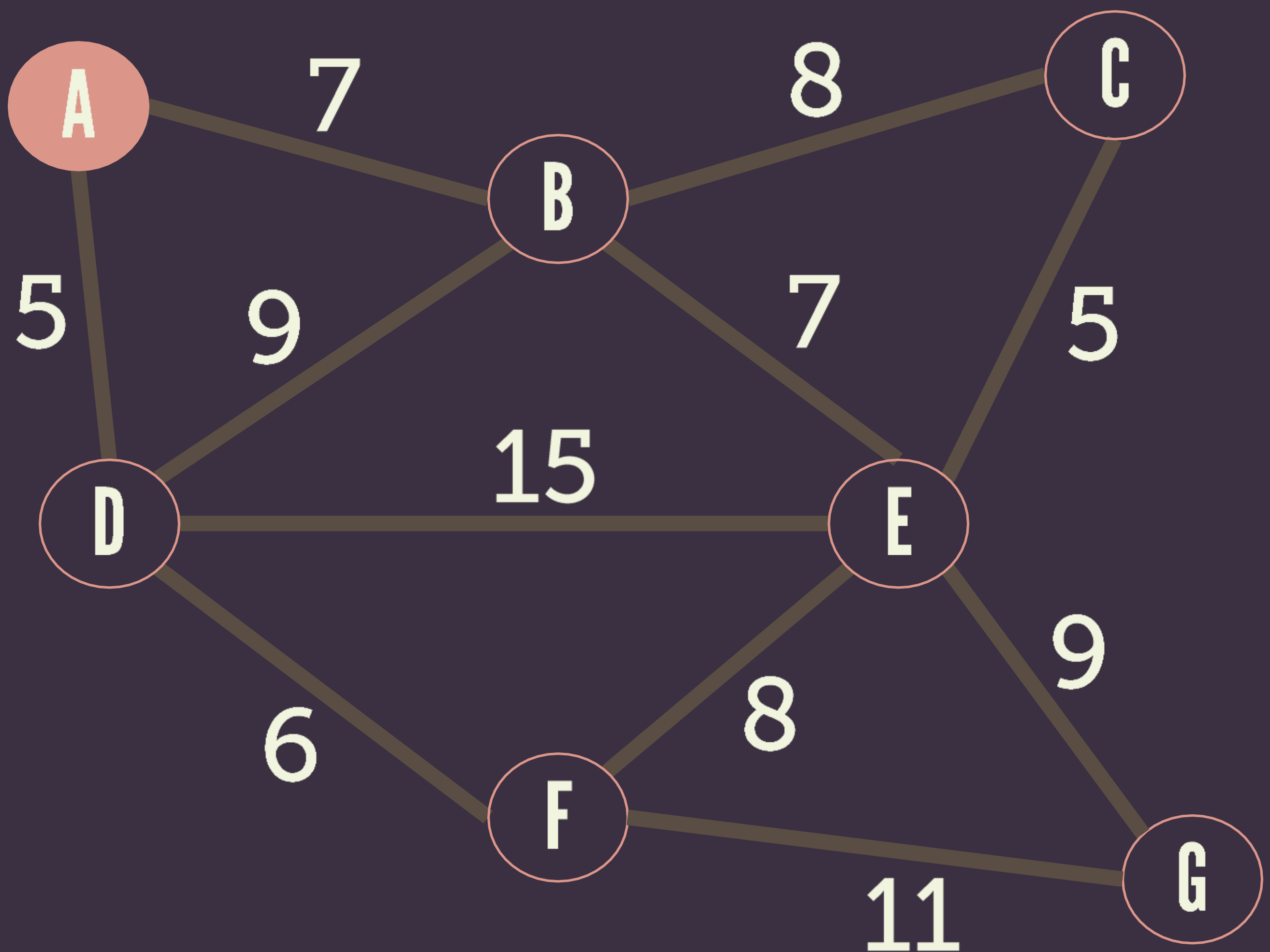
minimum

SPANNING TREES

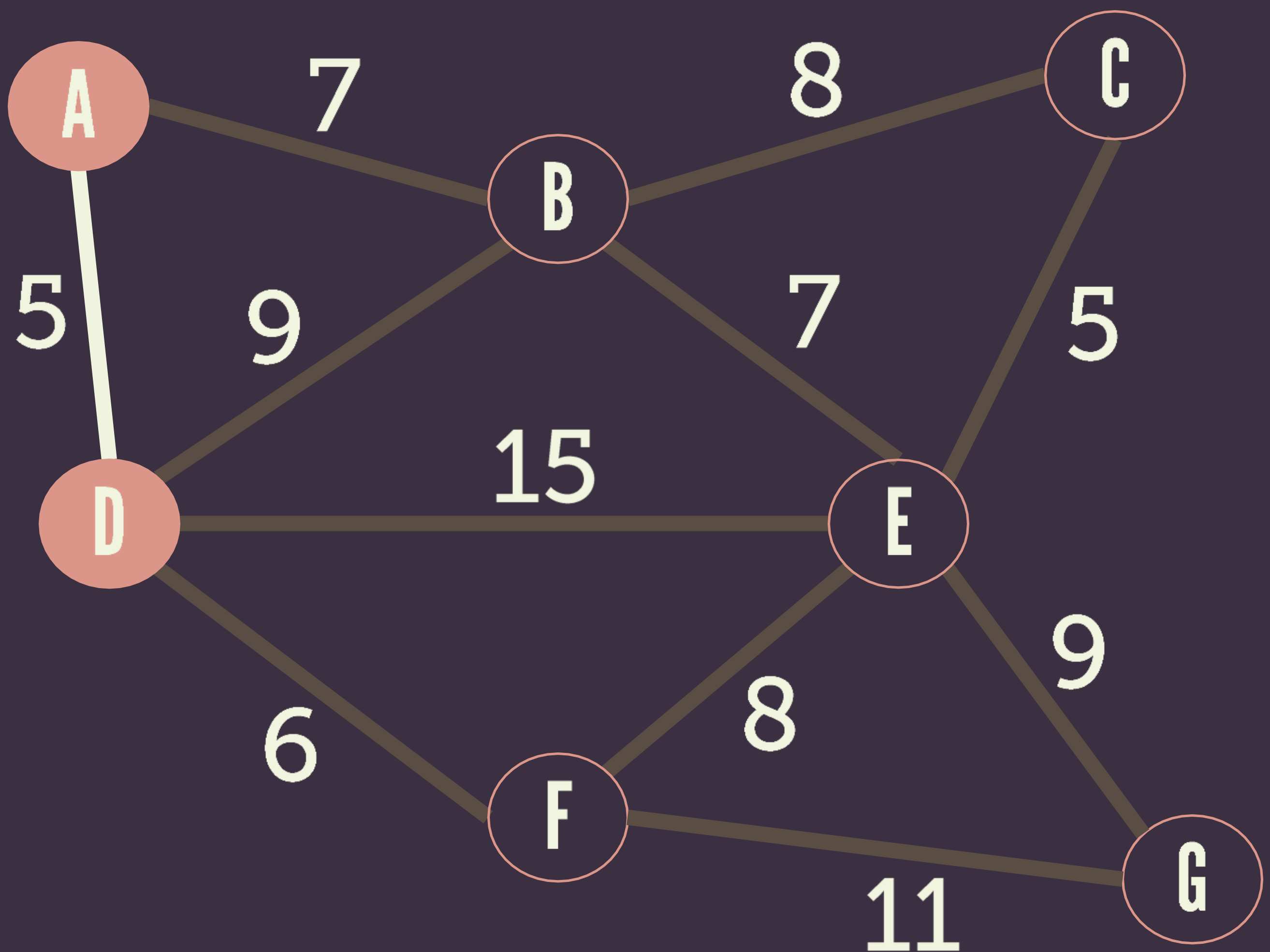
minimum **SPANNING TREE**

A tree formed from graph edges that connects all the vertices of the graph at lowest cost.

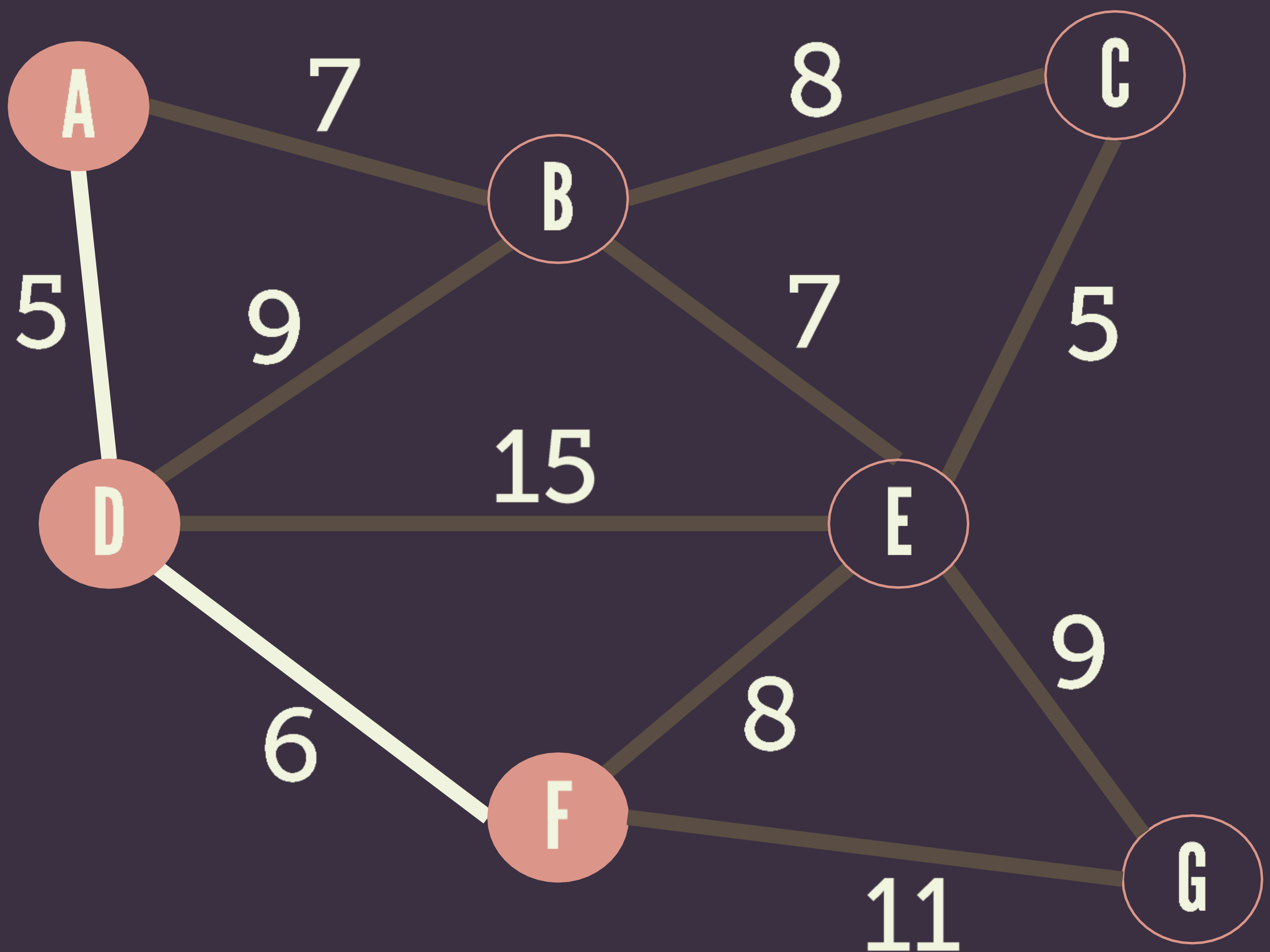

A
B
C
D
E
F
G
7
8
5
9
7
5
15
6
8
9
11



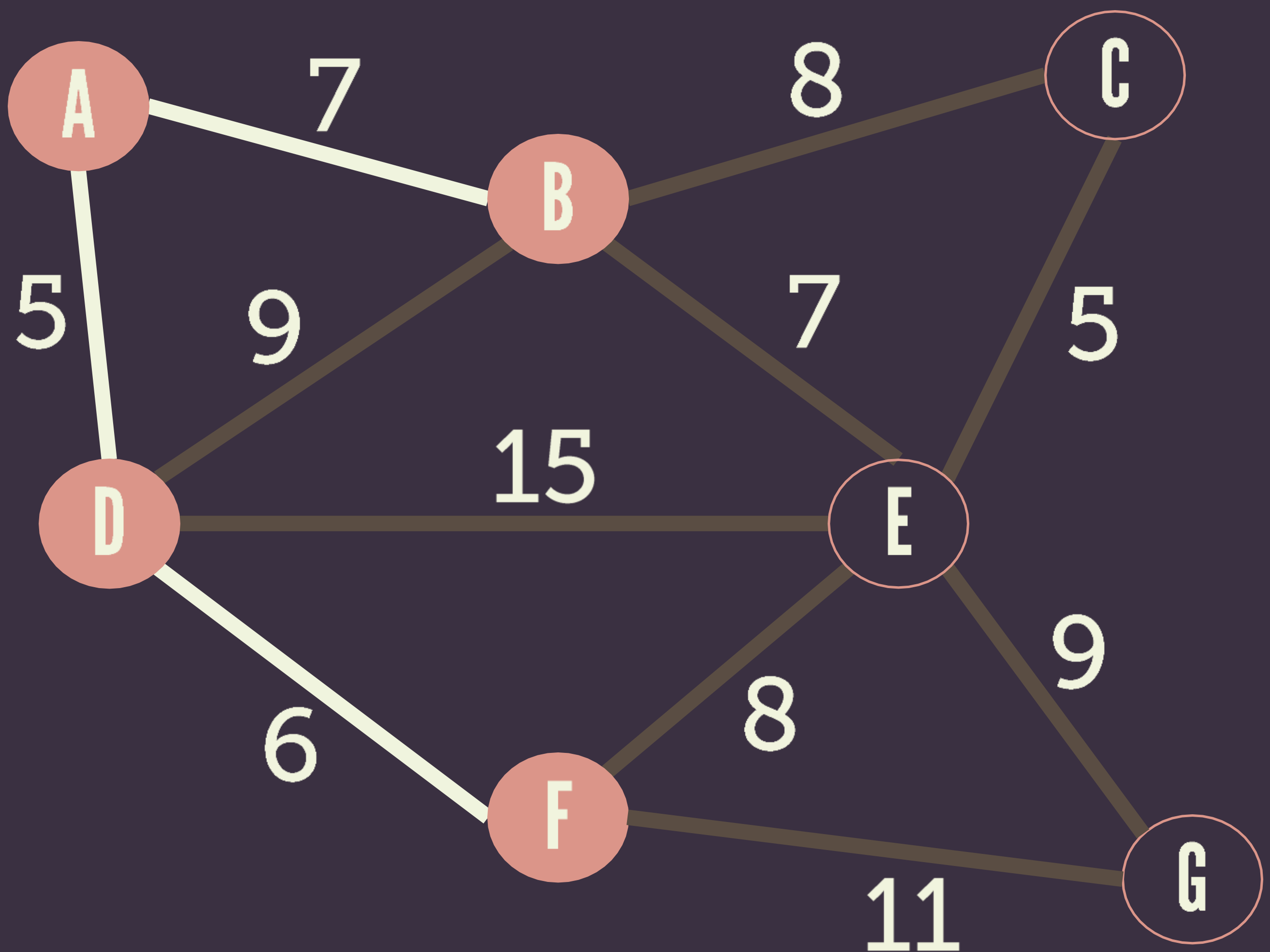
cost = ?



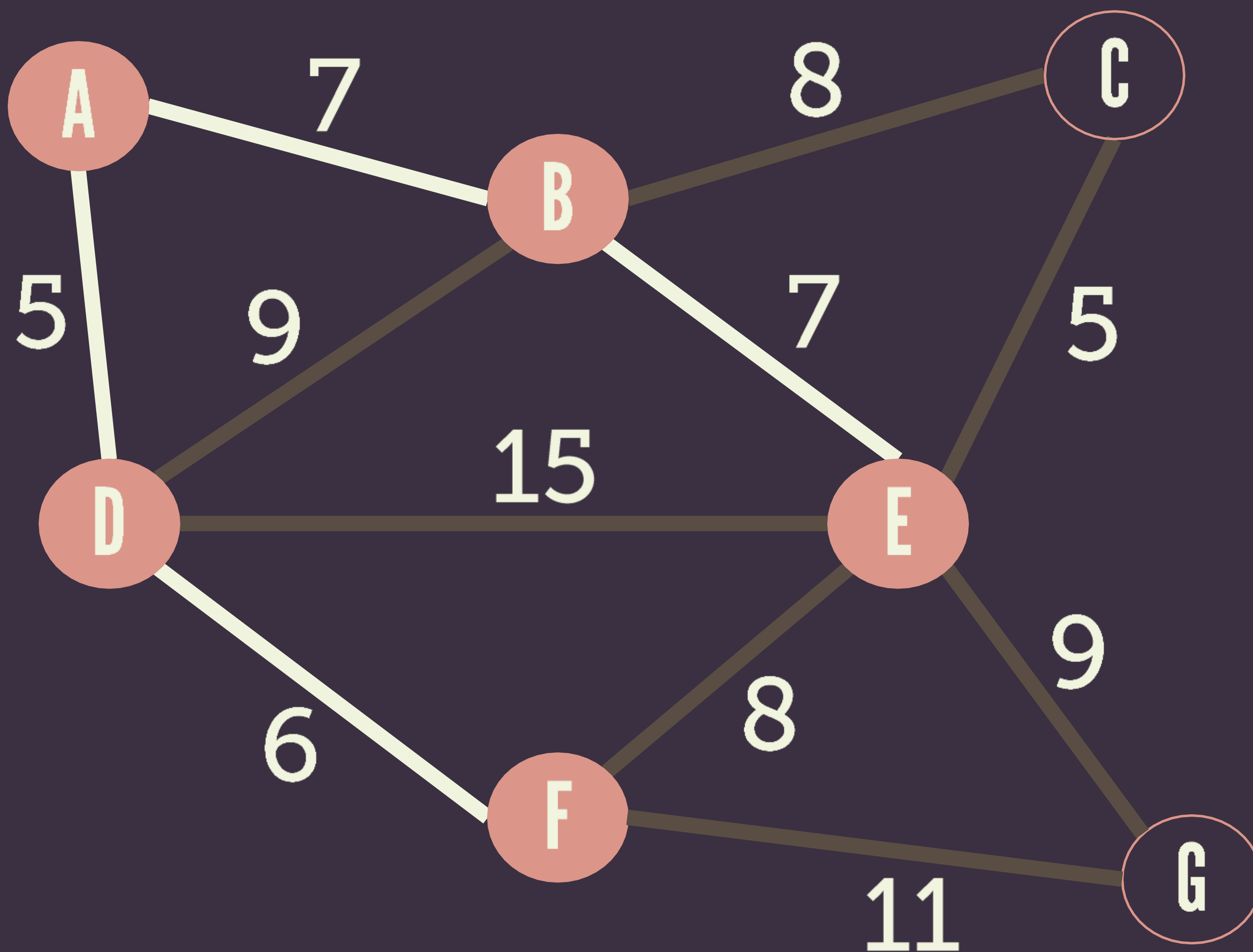
cost = 5



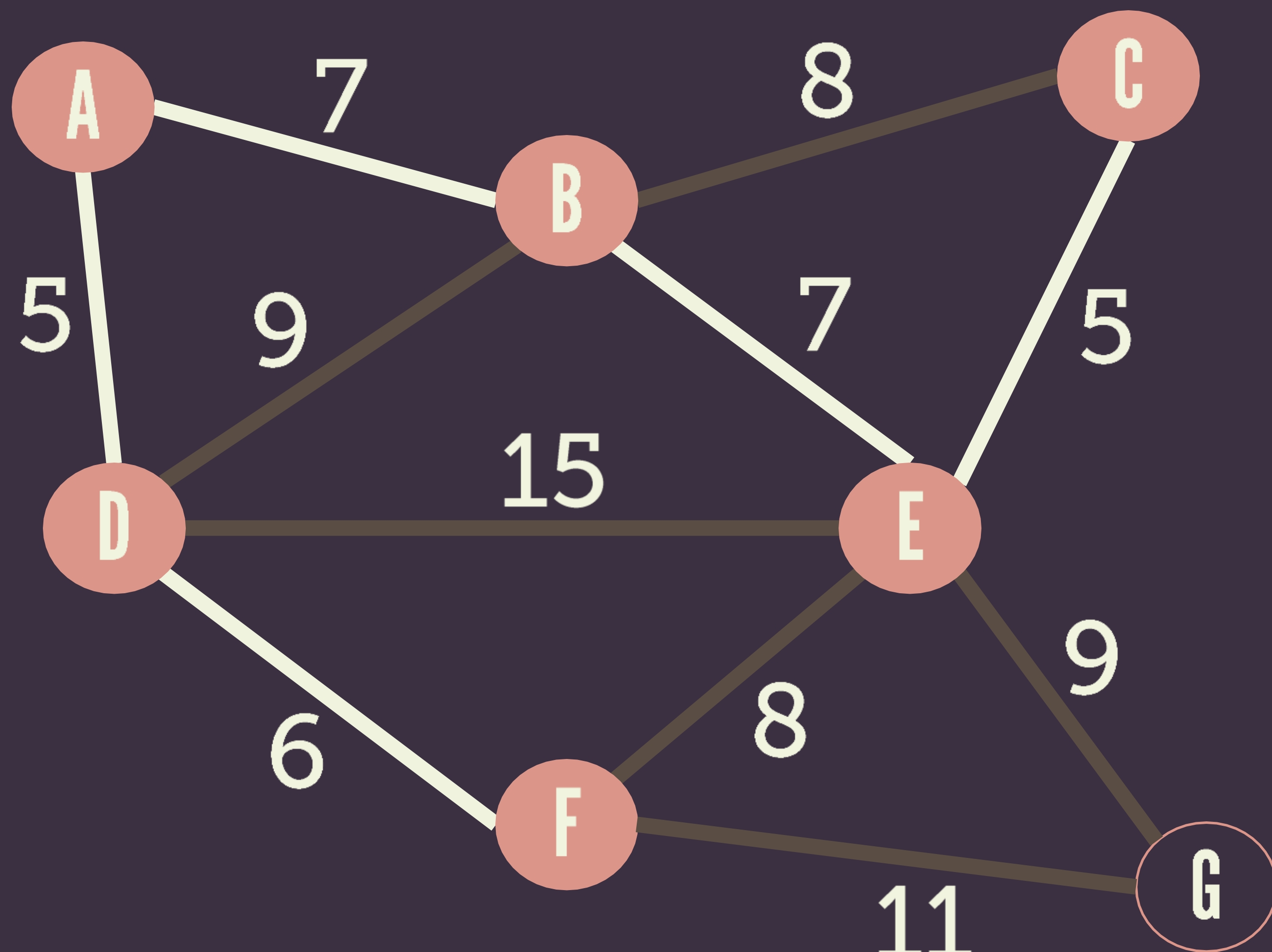
cost = 11



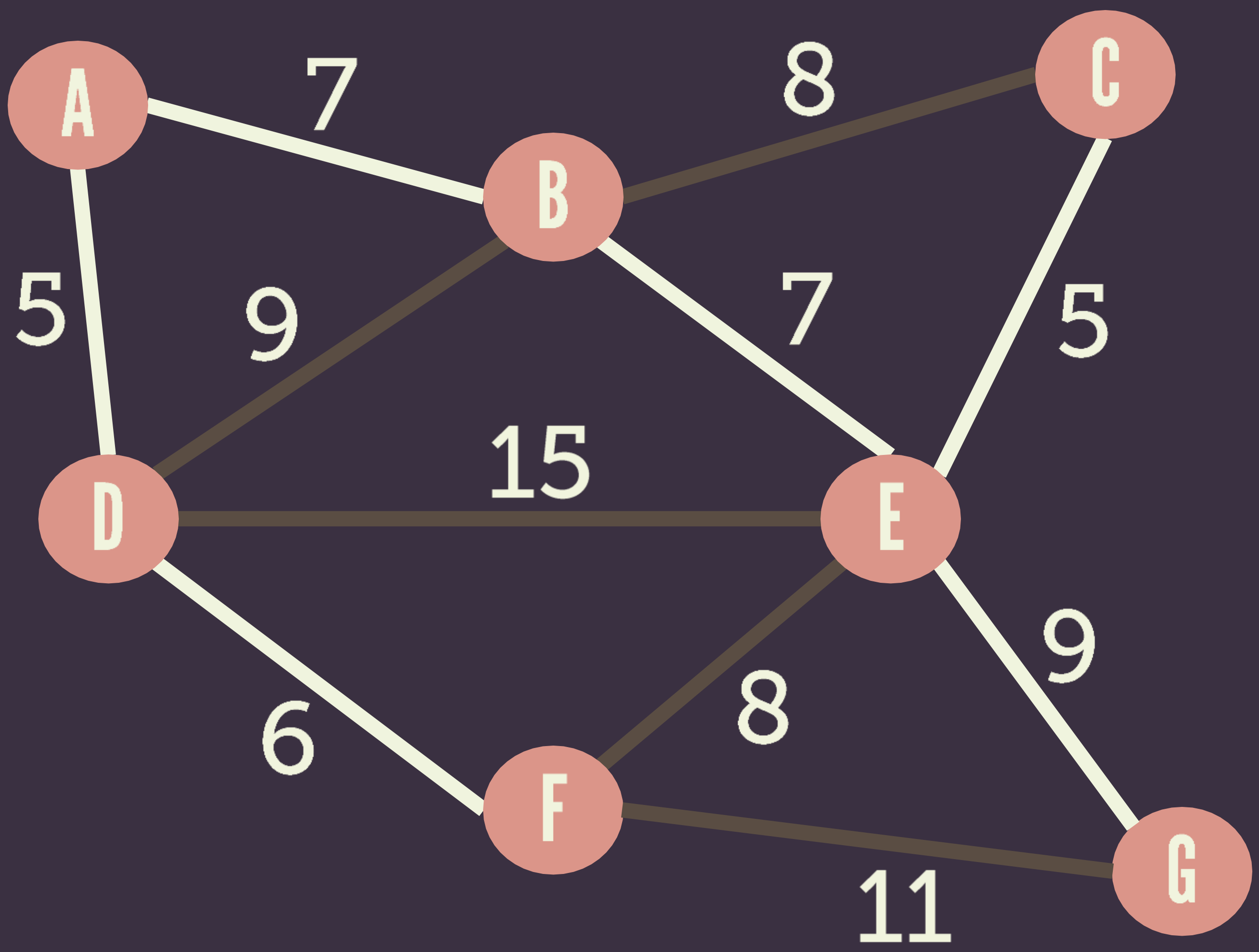
cost = 18



cost = 25



cost = 30



cost = 39