Lecture 7: Graph Algorithms – I

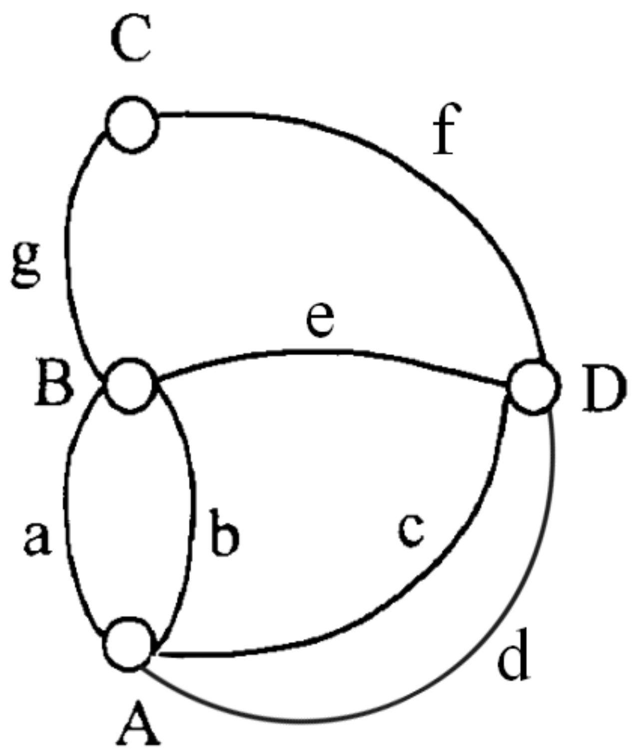
Tsung-Wei (TW) Huang

Department of Electrical and Computer Engineering

University of Utah, Salt Lake City, UT

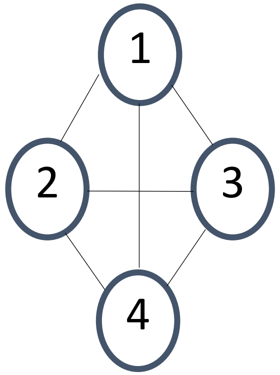
Graph Definition

- **Vertex (V)**
  - A, D, B, D
- **Edge (E)**
  - BC, CD,…
- **Degree (deg)**
  - The branch of a vertex
- **Path (P)**
  - A sequence of connected vertices, e.g. ADCB
- **Cycle (C)**
  - A sequence of connected vertices with same end points

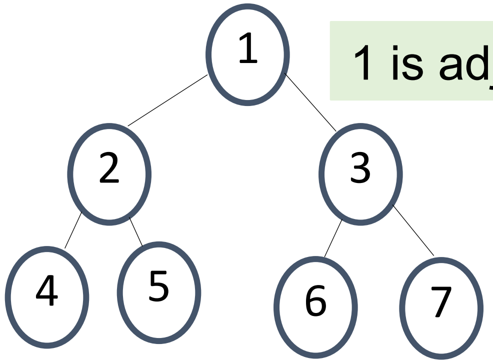


Graph Definition (cont'd)

- Undirected Graph – G1, G2
- Directed Graph – G3
  - Indegree and outdegree

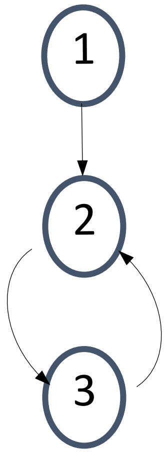


G1



1 is adjacent to 3

G2

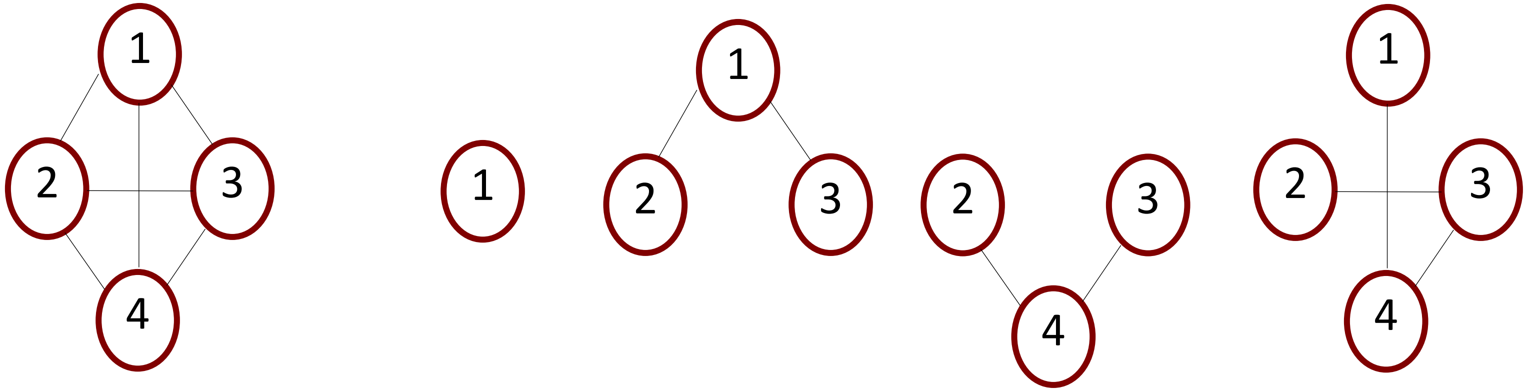


edge is incident to2

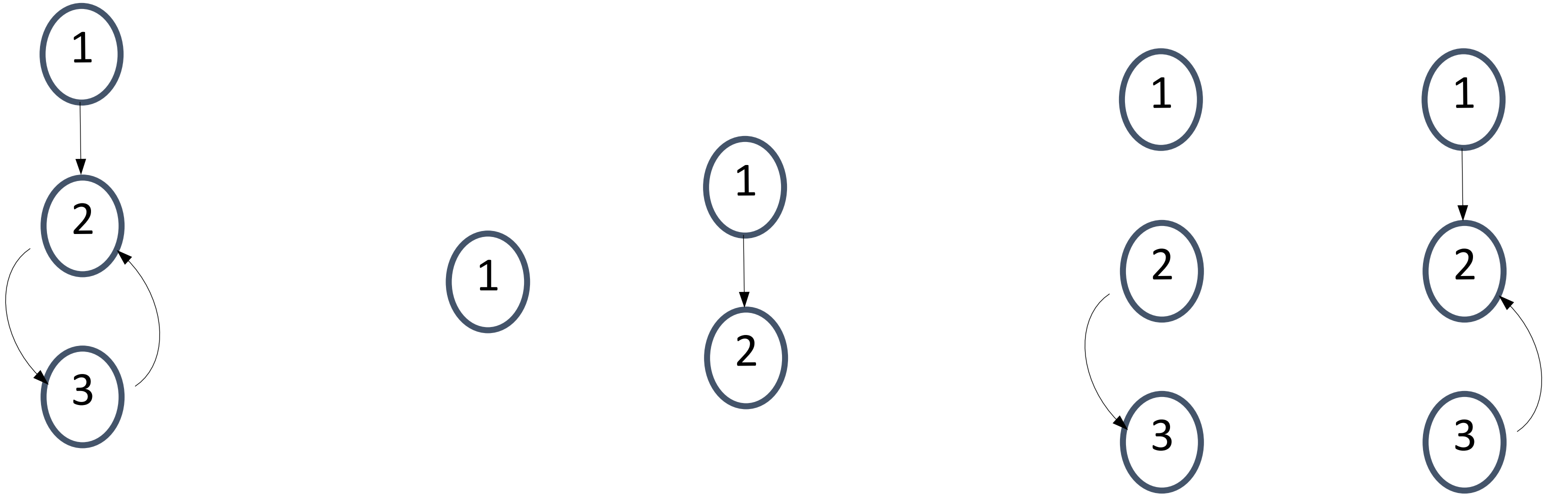
G3

Subgraph

G1

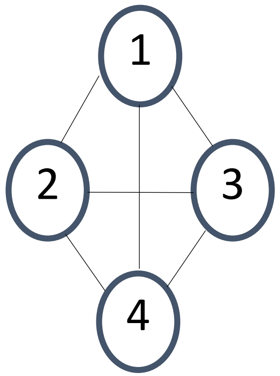


G3

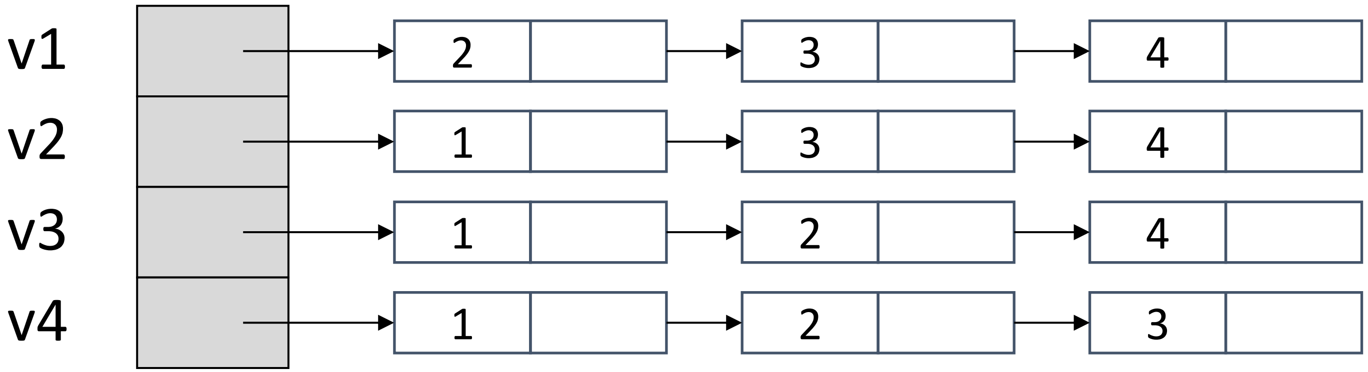


Graph Data Structure

- **Adjacent list**
  - Each node keeps a list of adjacent nodes
  - Commonly, we use `std::vector<std::vector<Vertex>>`

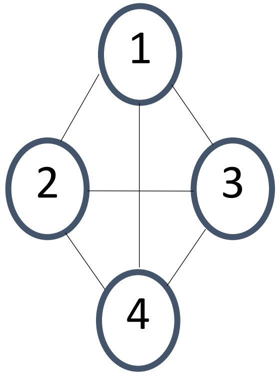


G1



Graph Data Structure (cont'd)

- **Adjacency Matrix**
  - A complete matrix denotes all-pair connections



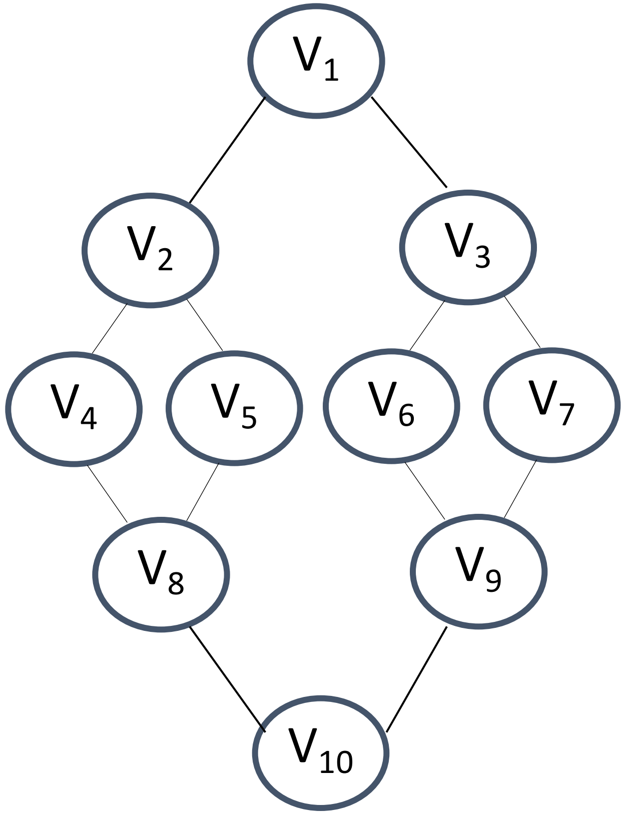
G1

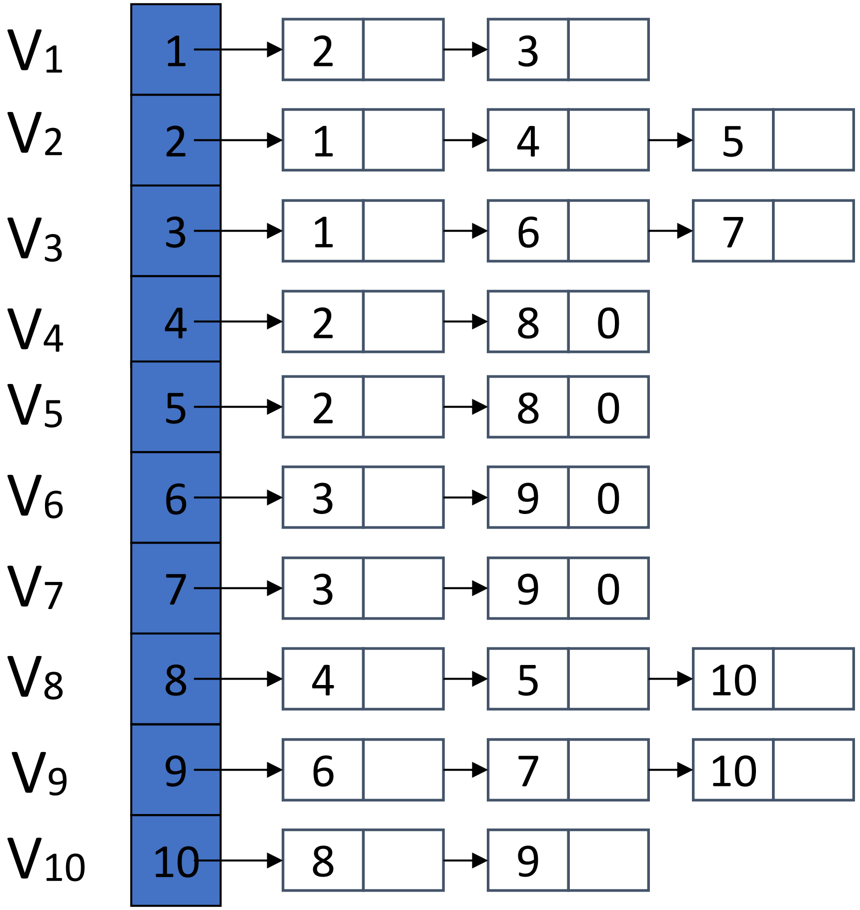
	1	2	3	4
1	0	1	1	1
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

Graph Traversal Algorithms

- **Depth First Search (DFS)**
  - The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking
  - Traversal order is last-in-first-out (stack)
- **Breadth First Search (BFS)**
  - The algorithm starts at the root node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level
  - Traversal order is first-in-first-out (queue)

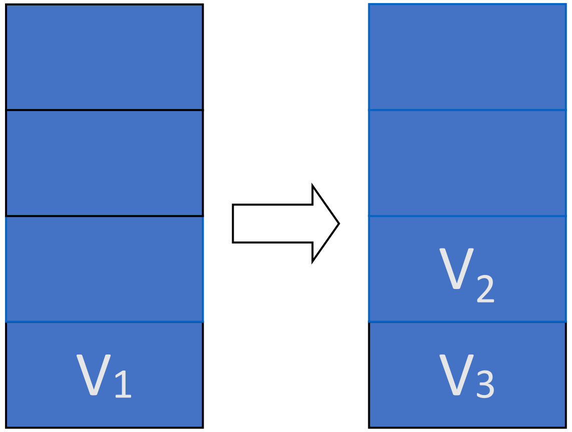
DFS Walkthrough – 1

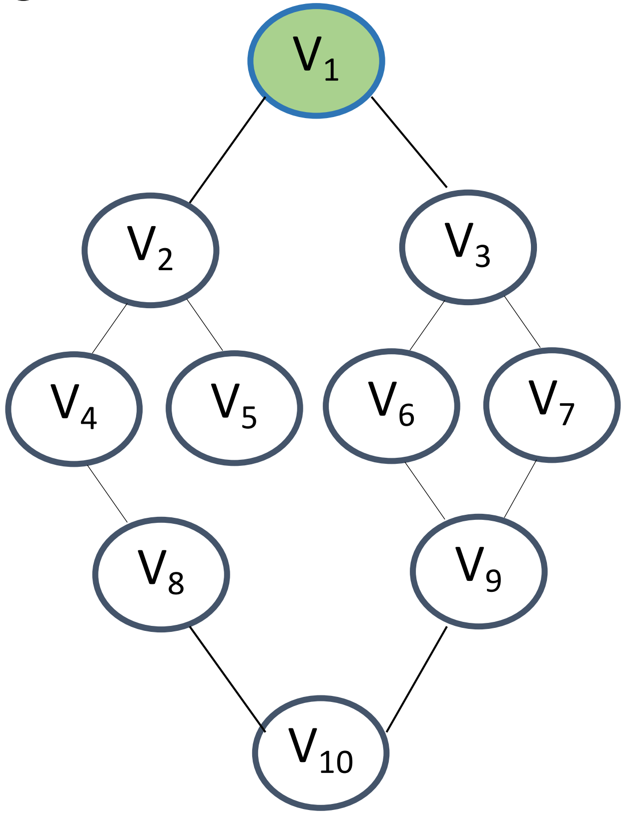




DFS Walkthrough – 2

- **Start from v1 and insert v2 and v3**

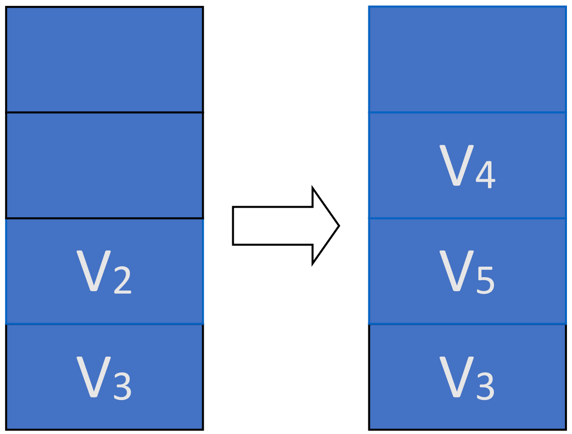


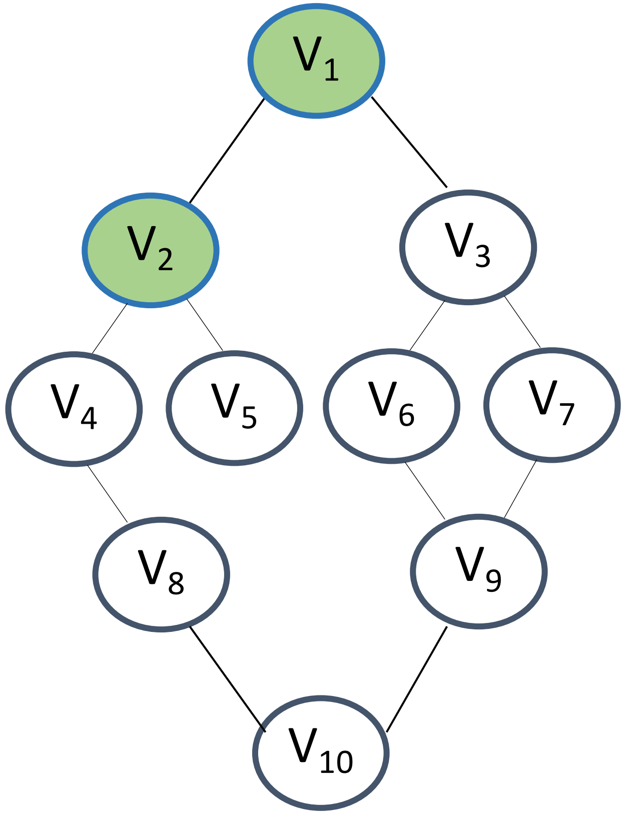


output：V_1

DFS Walkthrough – 3

- **Pop v2 and inserts its v4 and v5**

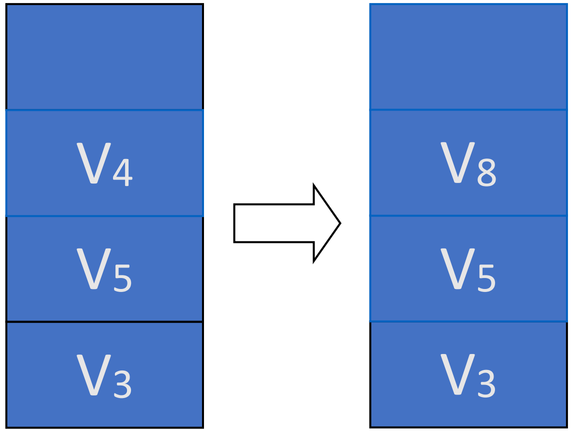


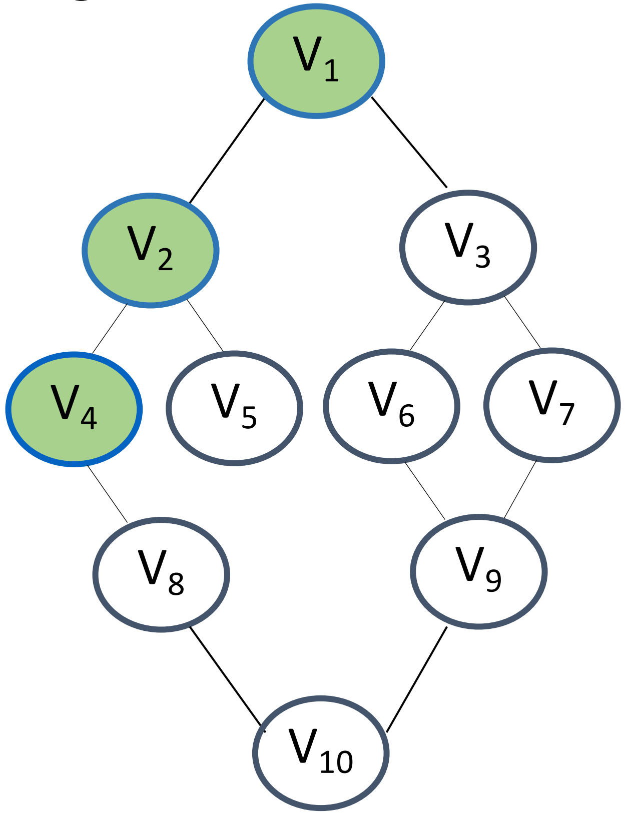


output：V_1 V_2

DFS Walkthrough – 4

- **Pop v4 from the stack and insert v8**

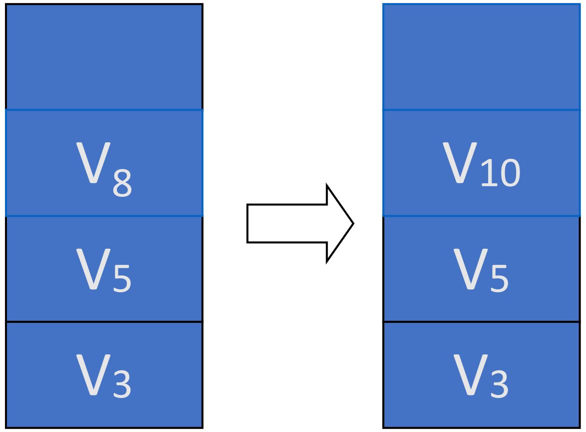




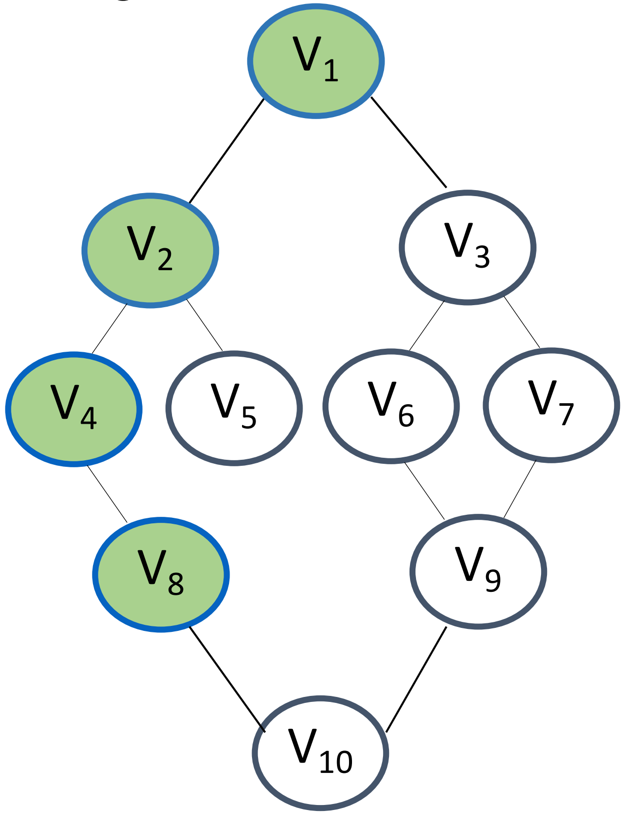
output：V_1 V_2 V_4

DFS Walkthrough – 5

- **Pop v8 from the stack and insert v10**

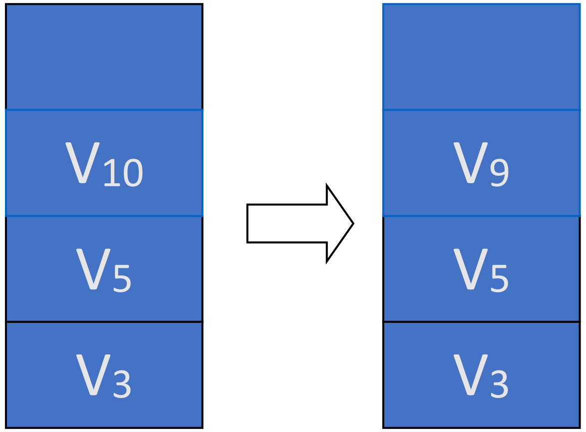


output：V_1 V_2 V_4 V_8

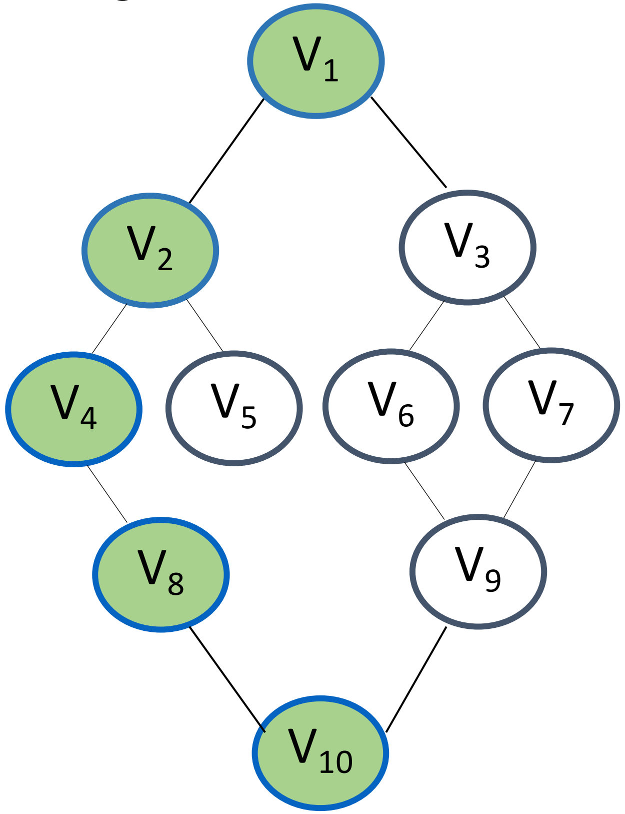


DFS Walkthrough – 6

- **Pop v8 from the stack and insert v10**

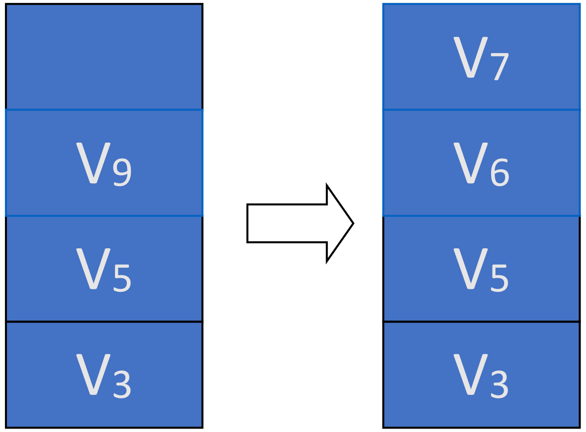


output：V_1 V_2 V_4 V_8 V_{10}

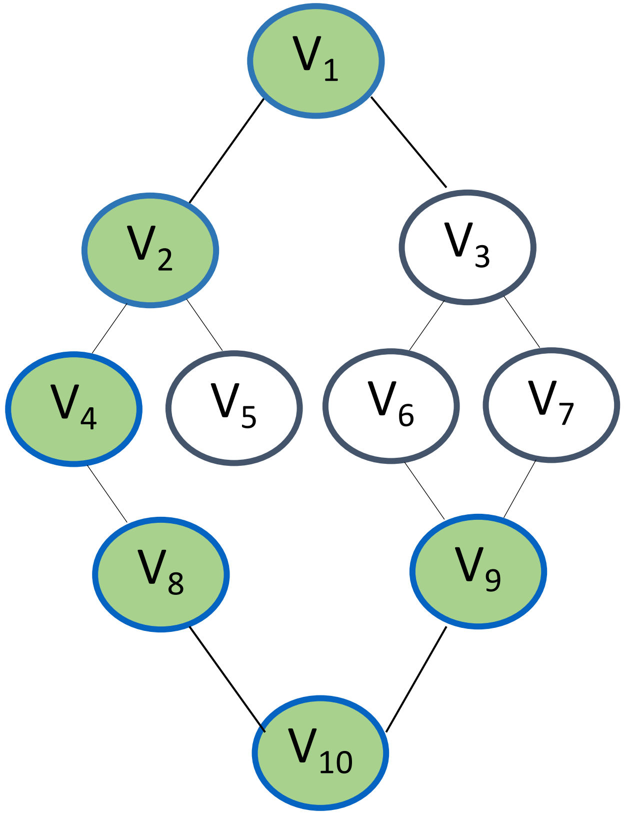


DFS Walkthrough – 7

- **Pop v9 and insert v6 and v7**

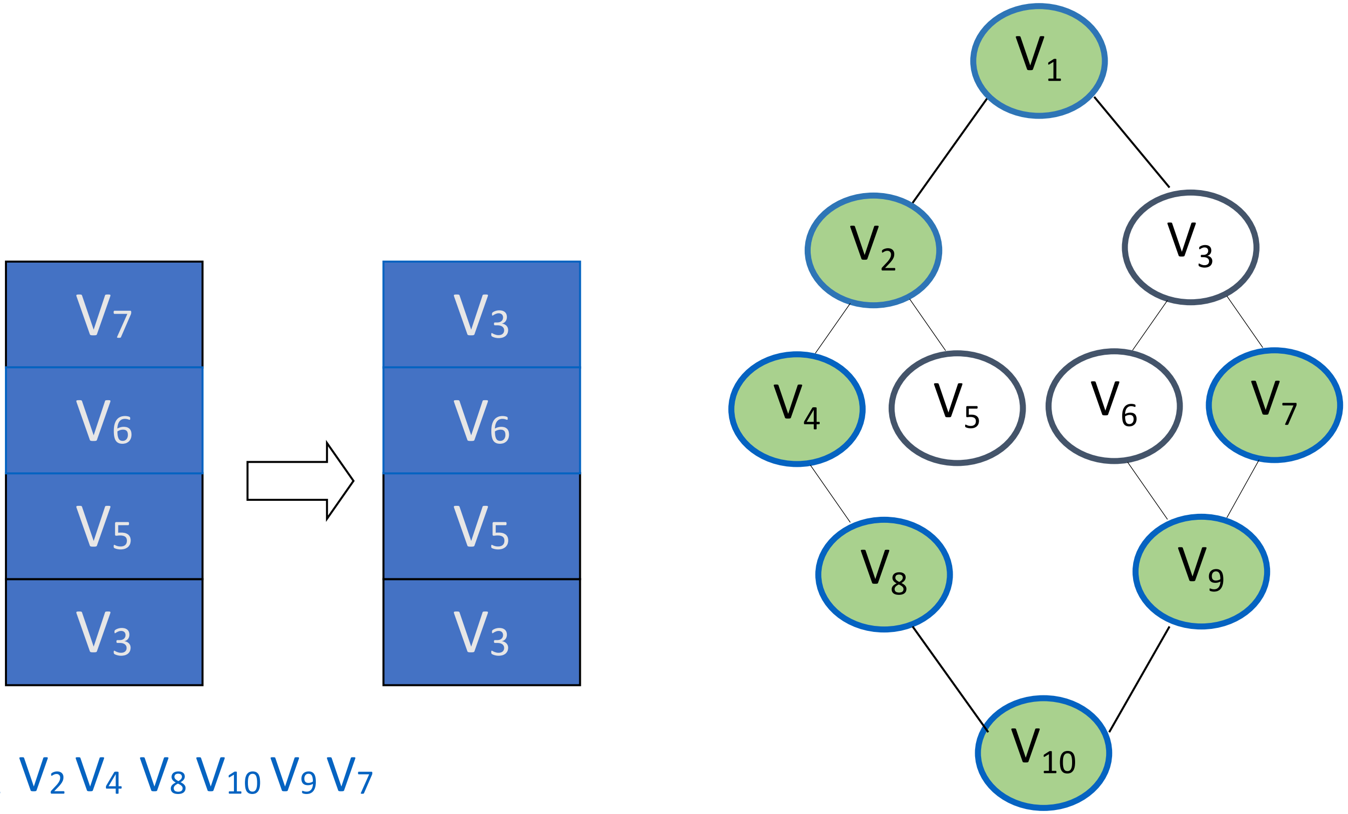


output：V_1 V_2 V_4 V_8 V_{10} V_9



DFS Walkthrough – 8

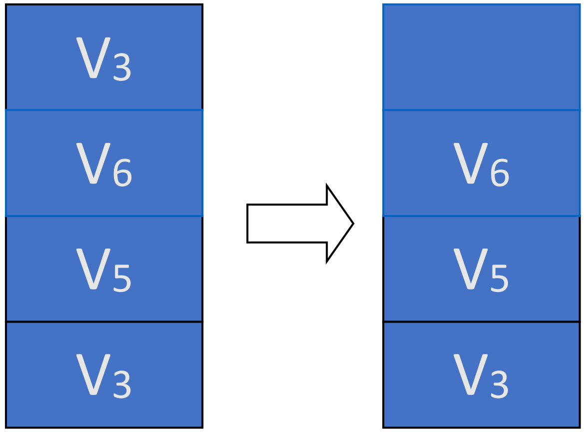
- Pop v7 from the stack and insert v3



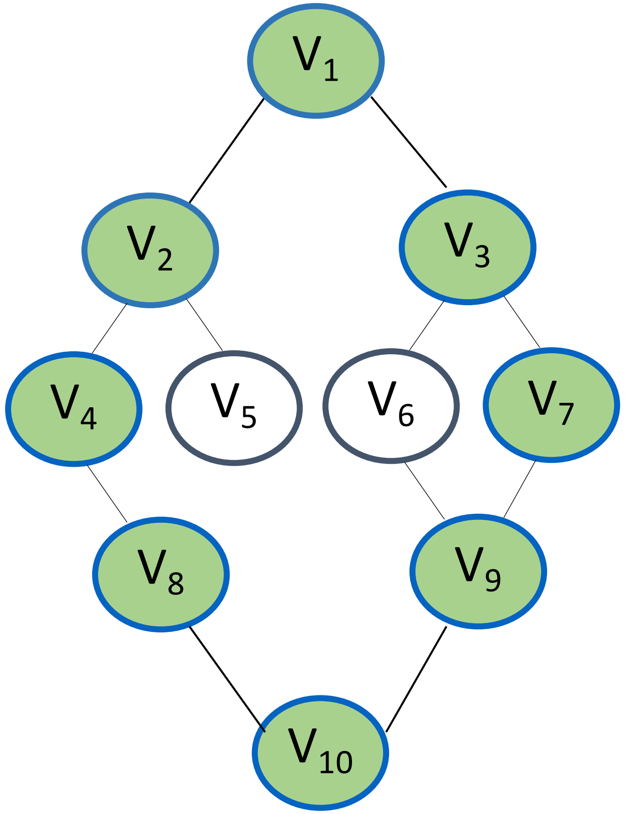
output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7

DFS Walkthrough – 9

- **Pop v3 from the stack**

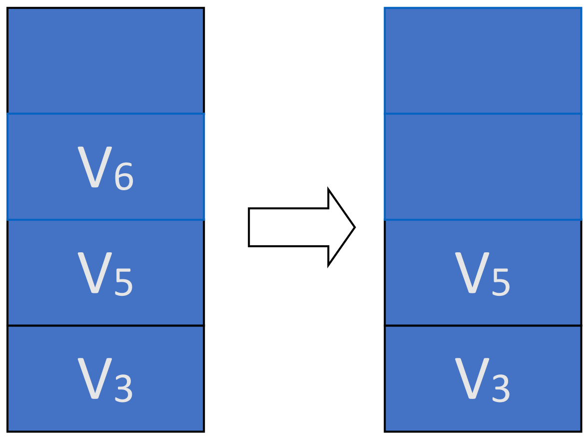


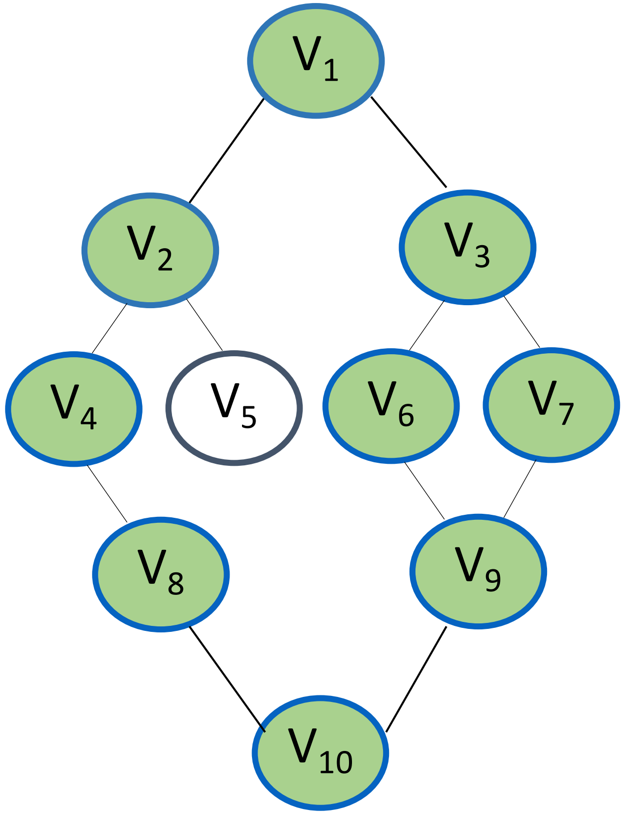
output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7 V_3



DFS Walkthrough – 10

- **Pop v6 from the stack**

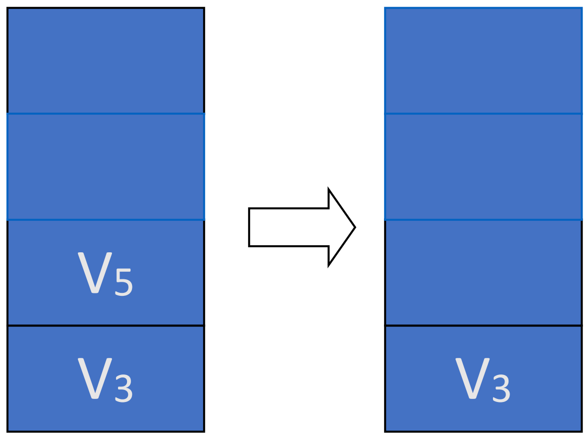


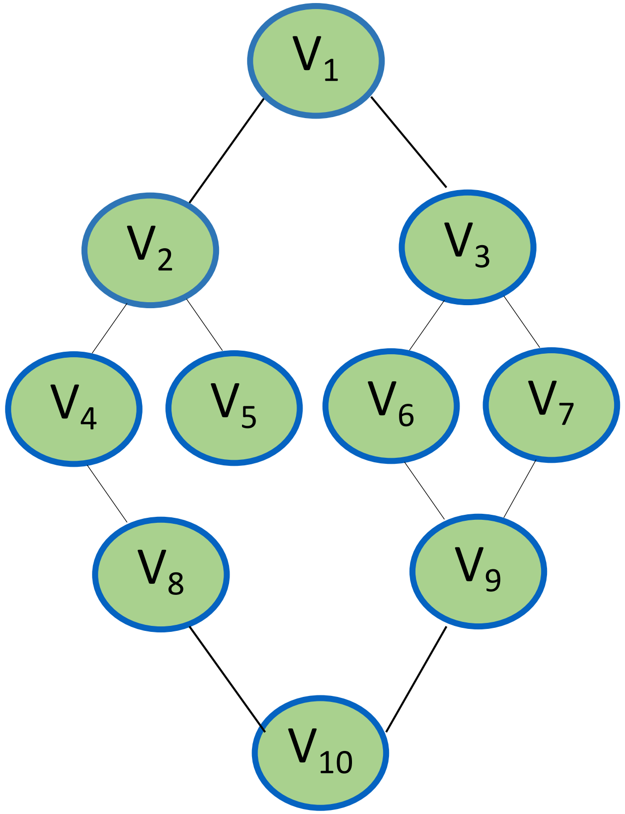


output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7 V_3 V_6

DFS Walkthrough – 11

- **Pop v5 from the stack**

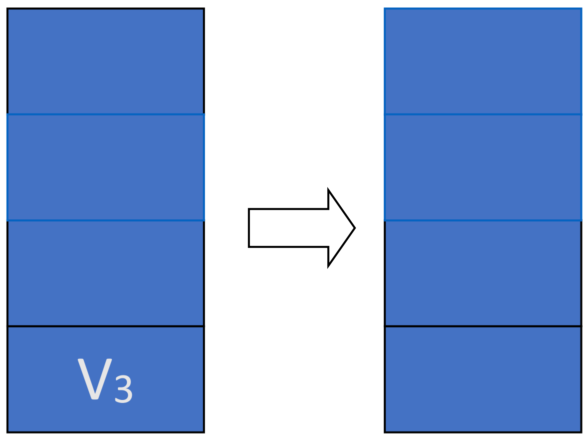


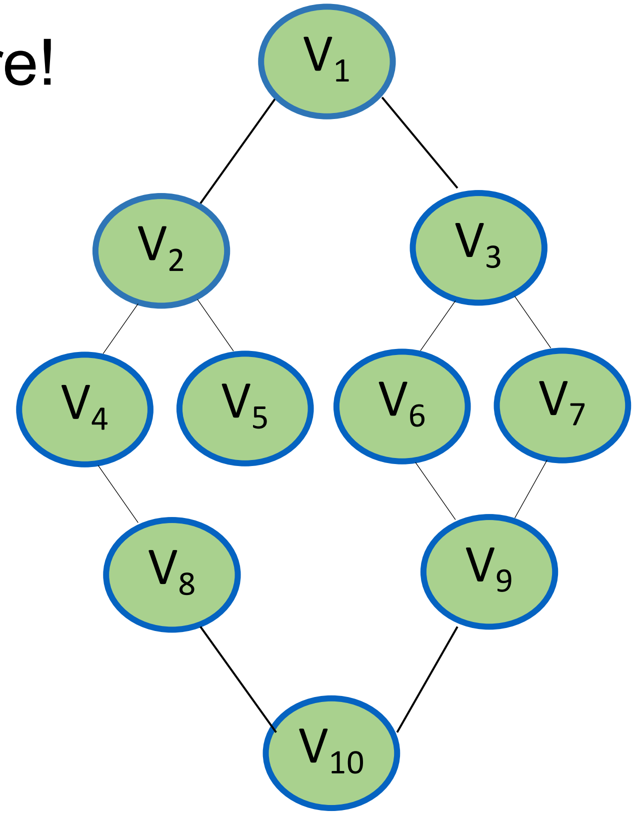


output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7 V_3 V_6 V_5

DFS Walkthrough – 12

- **Pop v3 from the stack**
  - No output for v3 as we visited it before!





output：V_1 V_2 V_4 V_8 V_{10} V_9 V_7 V_3 V_6 V_5

Recursive Implementation of DFS

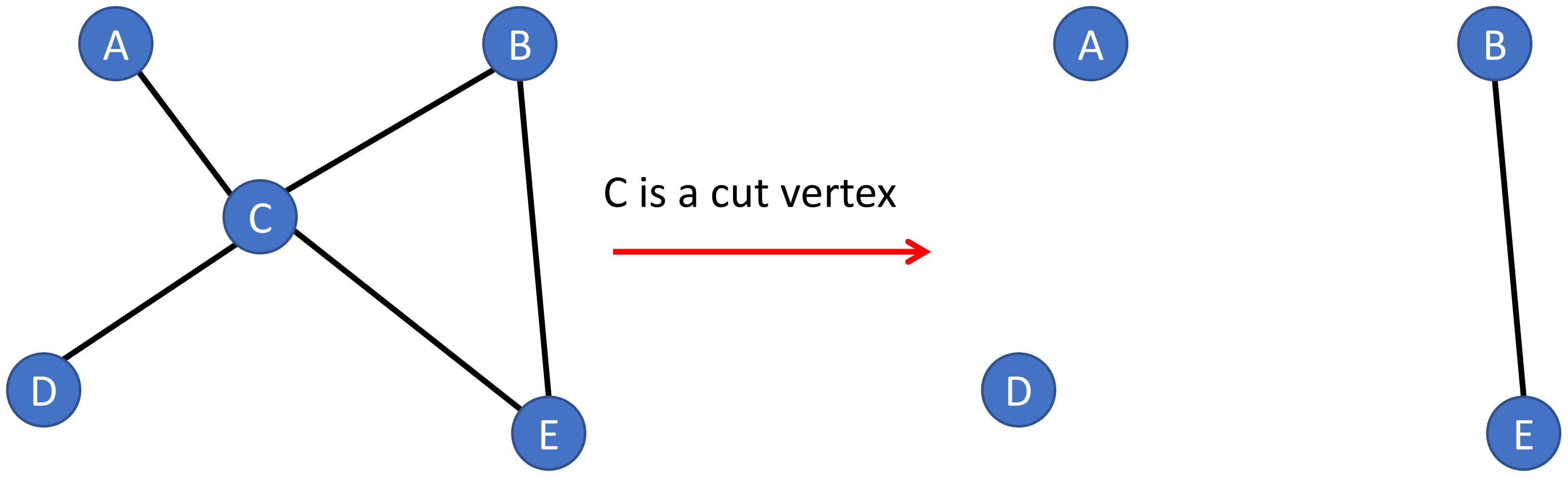
```
void DFS(int v)
{
    // Mark the current node as visited and print it
    visited[v] = true;
    std::cout << v << " ";

    // Recur for all the vertices adjacent to this vertex
    for (auto i = adj[v].begin(); i != adj[v].end(); ++i) {
        if (!visited[*i]) {
            DFS(*i);
        }
    }
}
```

Stack-based Implementation of DFS

```
vector<bool> visited(V, false);
stack<int> stack;
stack.push(s);
while (!stack.empty()) {
    s = stack.top();
    stack.pop();
    // Stack may contain same vertex twice – only print unvisited ones
    if (!visited[s])   {
        cout << s << " ";
        visited[s] = true;
    }
    // Get all nonvisited adjacent vertices of the popped vertex s
    for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
        if (!visited[*i])  stack.push(*i);
}
```

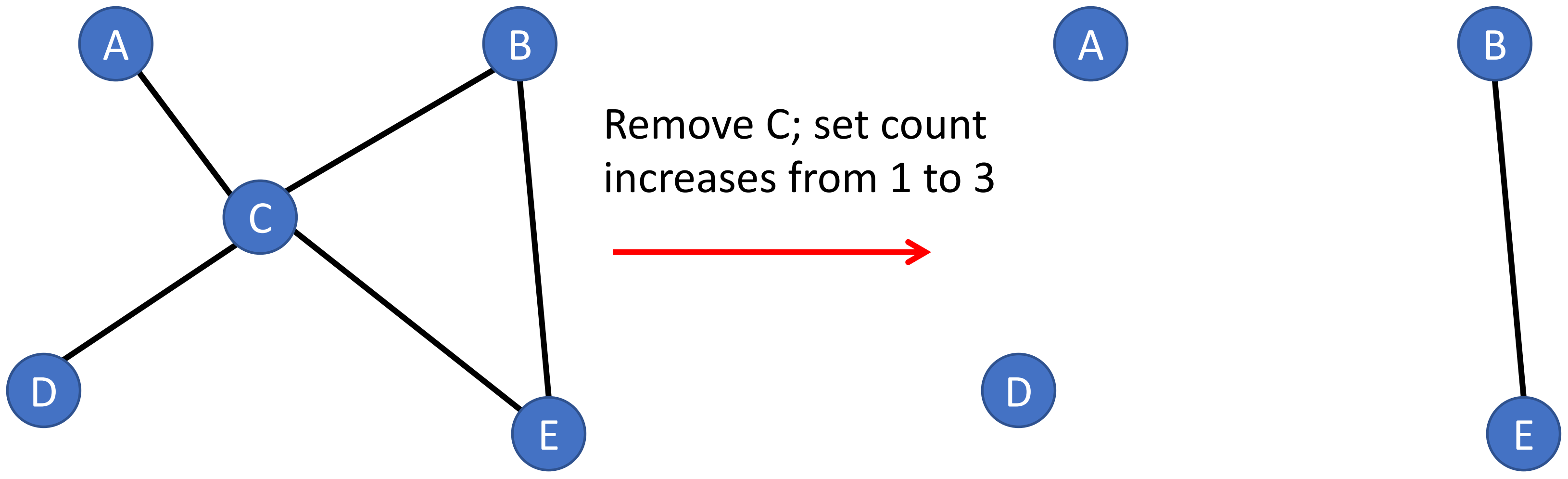
Cut Vertex (Articulation Point)

- A cut vertex or articulation point is a vertex in a graph such that removal of the vertex causes an increase in the number of connected components.
- If the graph was connected before the removal of the vertex, it will be disconnected afterwards.



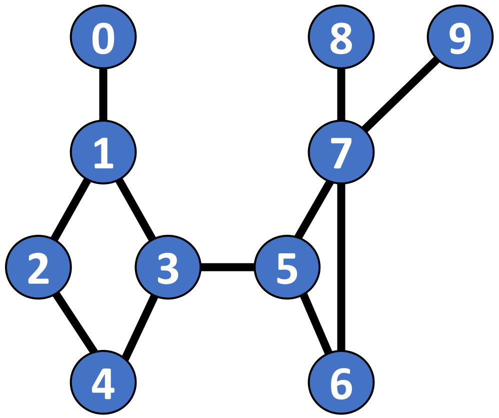
How to Find Cut Vertices? Brute Force?

- Enumerate all vertices O(N)
  - Remove the vertex from the graph
  - Perform union-and-find algorithms to find the number of sets
  - If the number of disjoint sets increases, the vertex is a cut
- Total time complexity is $O(N^2 \log N)$
  - Each union-and-find takes O(NlogN), needs N times



DFS can Do This for us More Efficiently

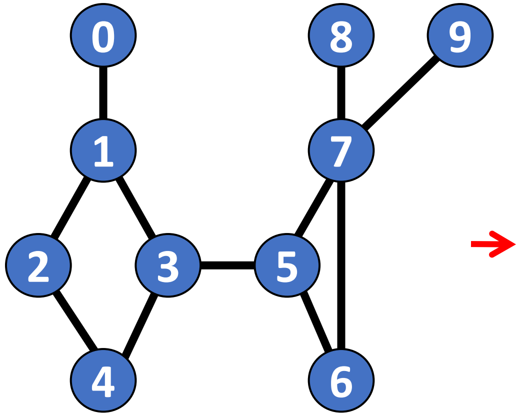
- We have two edge types during DFS
  - Forward edge u→v, v is not visited
  - Backward edge u→v, v is visited (except parent)



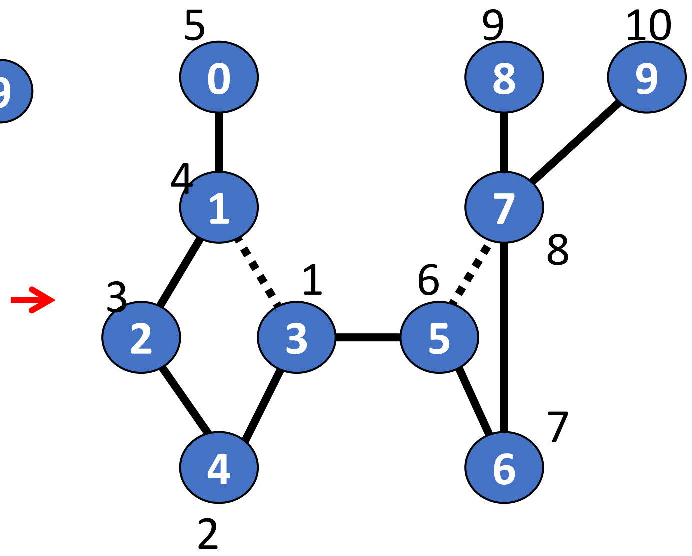
Let's start from 3

DFS can Do this for us

- We have two edge type during DFS
  - Forward edge u→v, v is not visited
  - Backward edge u→v, v is visited



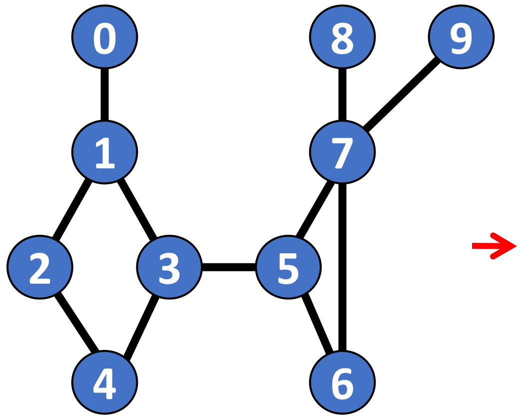
Let's start from 3



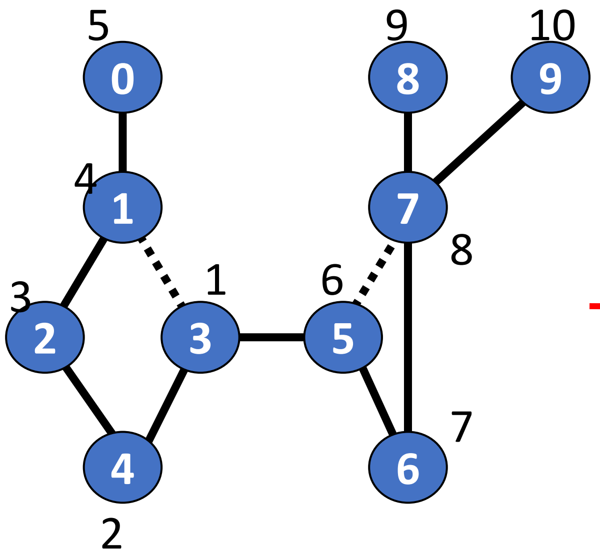
Lebel the order of traversal
in a linear array "dfn"

DFS can Do this for us

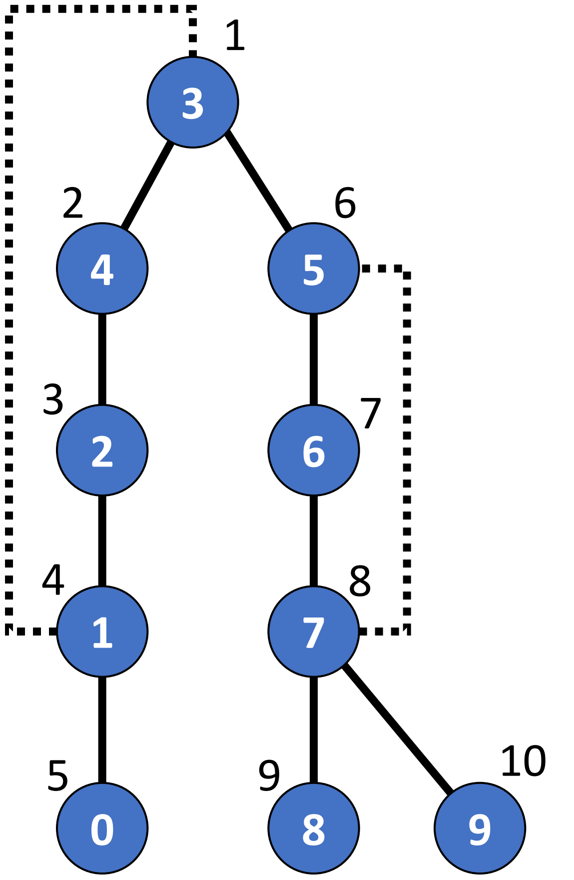
- We have two edge type during DFS
  - Forward edge u→v, v is not visited
  - Backward edge u→v, v is visited



Let's start from 3



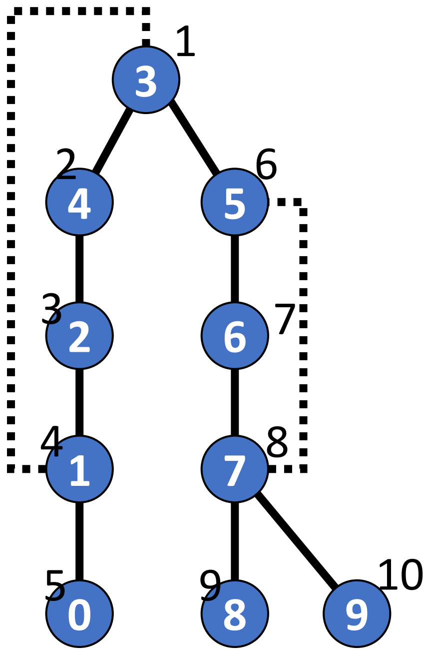
Lebel the order of traversal in a linear array "dfn"



DFS gives us a spanning tree order of vertices

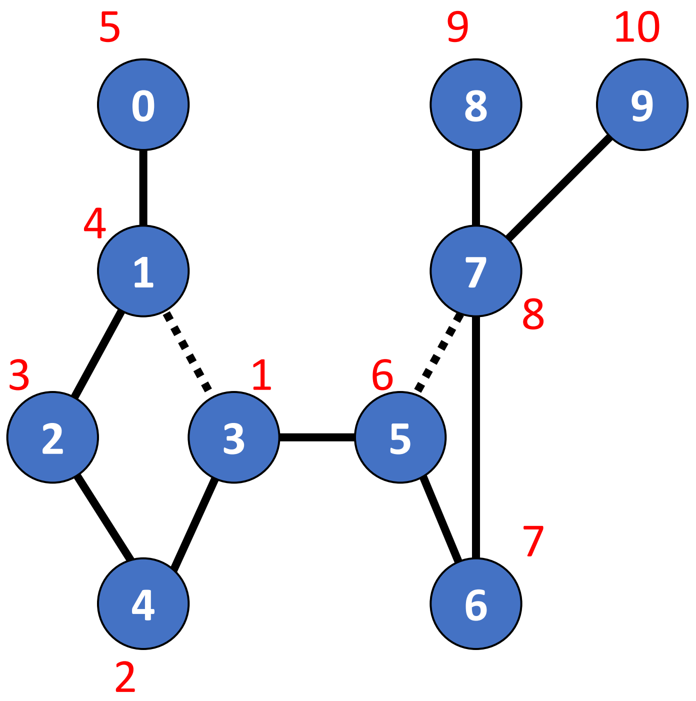
Cut Vertex Property

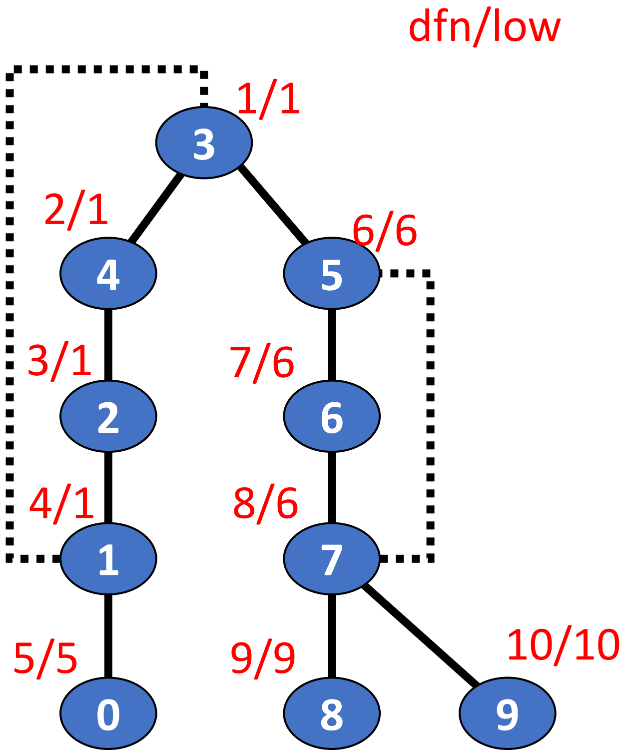
- Observation
  - If root has two children, => the root is a cut vertex
  - If a vertex u has a child v such that v can't go back to u's parent => u is a cut vertex
    - Assume no duplicate edges between vertices
- Let's quantify this
  - low[u]: the minimum dfn value u can reach
    - min{ low[u], low[v] }, foreach edge u→v



Example of dfn[i] and low[i]

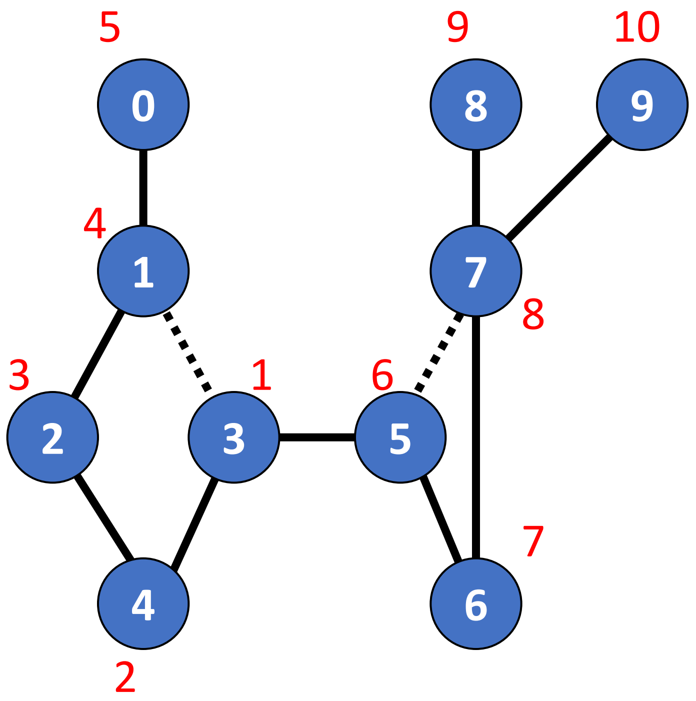
Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	9	10
low	5	1	1	1	1	6	6	6	9	10

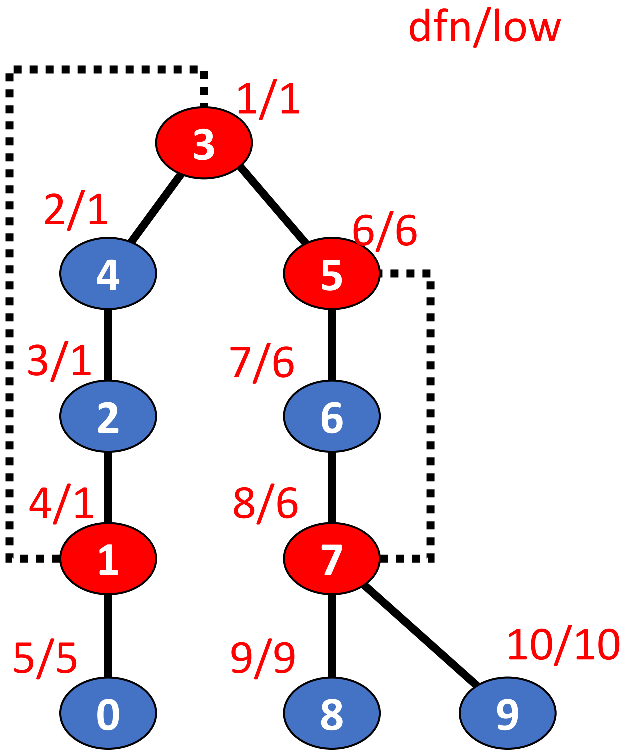




Cut Vertices Identified

Vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	9	10
low	5	1	1	1	1	6	6	6	9	10





Algorithm

```
void dfs(int u)  {
  visited[u]=true;
  low[u]=dfn[u]=____________;
  int child=0;
  for(int i=0; i<adj[u].size(); i++) {
    int v=adj[u][i];
    if(visited[v]==false) {
      child++;
      parent[v]=u;
      dfs(v);
      low[u]=____________;
      if______________________________________________
        cut[u]=true;
    }
    else if(v!=parent[u]) {  // backward edge
        ______________________
    }
}
```

Initialization:
times=0, parent[i]=-1, cut[i]=0, visited[i]=0

- **Observation**
  - If root has two children, => the root is a cut vertex
  - If a vertex u has a child v such that v can't go back to u's parent => u is a cut vertex
    - Assume no duplicate edges between vertices
- **Let's quantify this**
  - low(u): the minimum dfn value u can reach
    - min{ low(u), low(v)}, foreach edge u→v

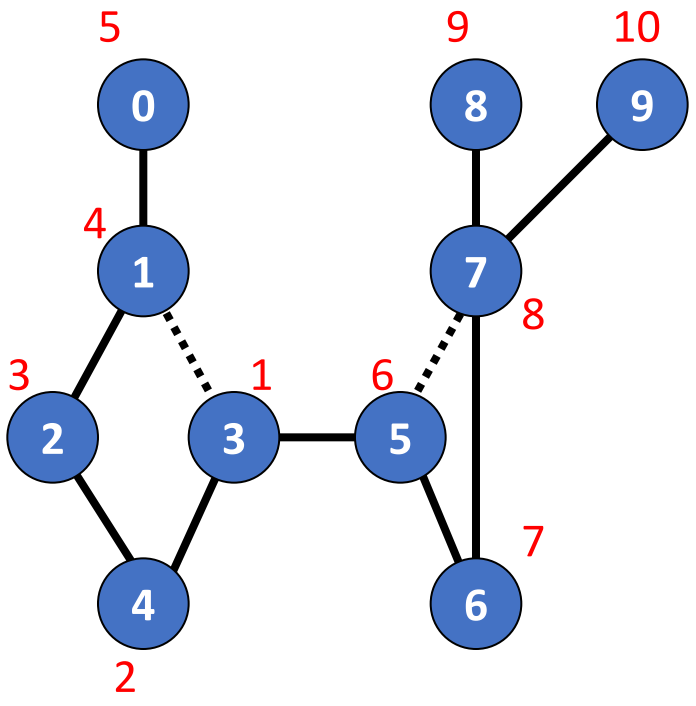
Algorithm (cont'd)

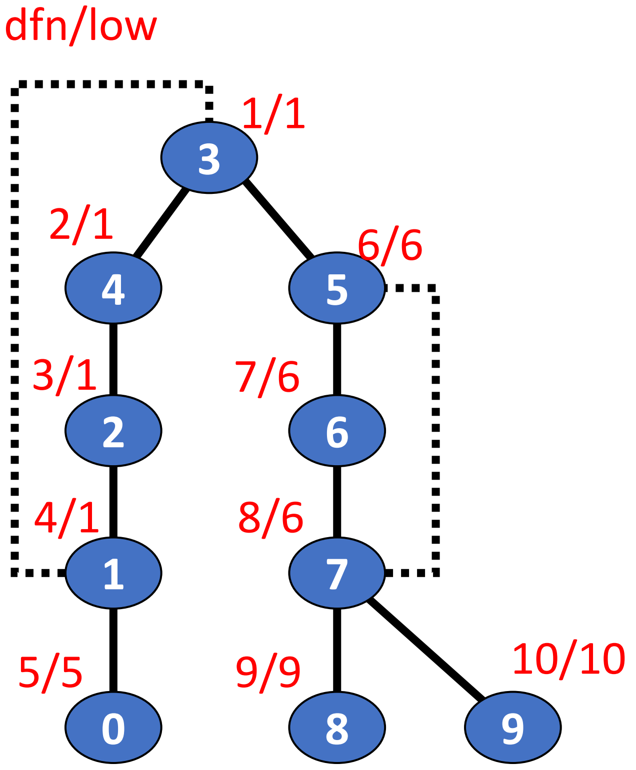
```
void dfs(int u)  {
  visited[u]=true;
  low[u]=dfn[u]=min(low[u], low[v]);
  int child=0;
  for(int i=0; i<adj[u].size(); i++) {
    int v=adj[u][i];
    if(visited[v]==false) {
      child++;
      parent[v]=u;
      dfs(v);
      low[u]= dfn[u]=(++times);;
      if( ((parent[u]!=-1) and ( low[v]>=dfn[u] )) or ( (parent[u]==-1) and (child>1)))
        cut[u]=true;
    }
    else if(v!=parent[u]) {  // backward edge
        low[u]=min(low[u], dfn[v]);
    }
}
```

Initialization:
times=0, parent[i]=-1, cut[i]=0, visited[i]=0

Cut Edge

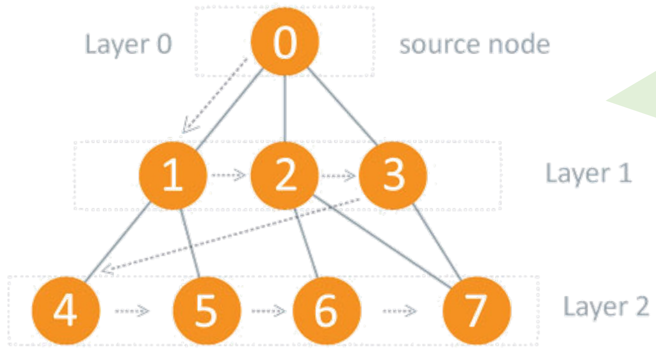
- Observation
  - For each edge u→v, if ____________ **=> cut edge**



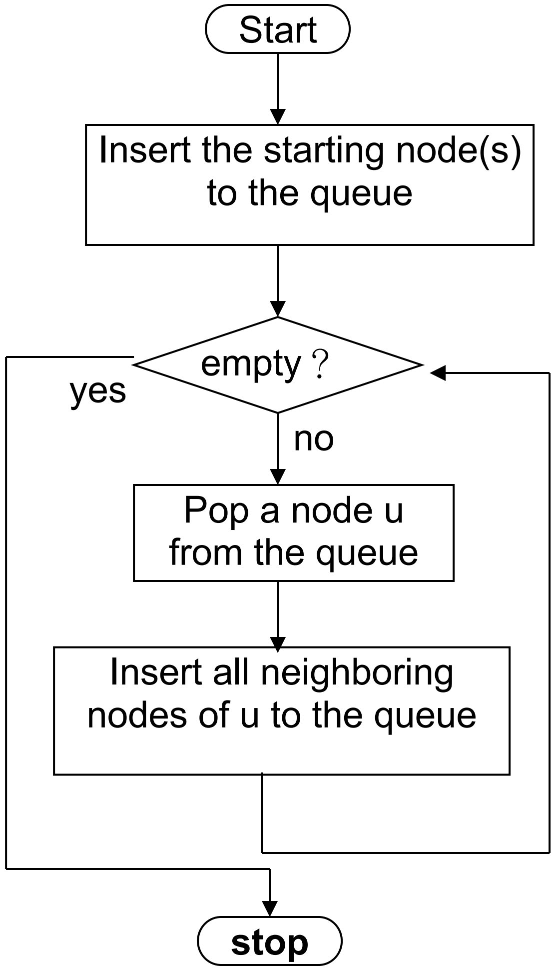


BFS Algorithm

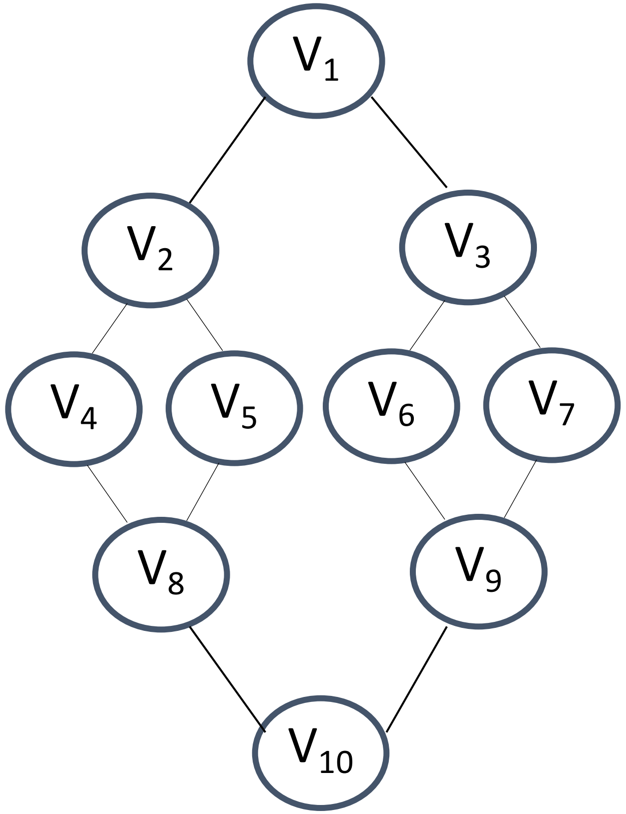
- **Breadth First Search (BFS)**
  - The algorithm starts at the root node, and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level
  - Traversal order is first-in-first-out (queue)

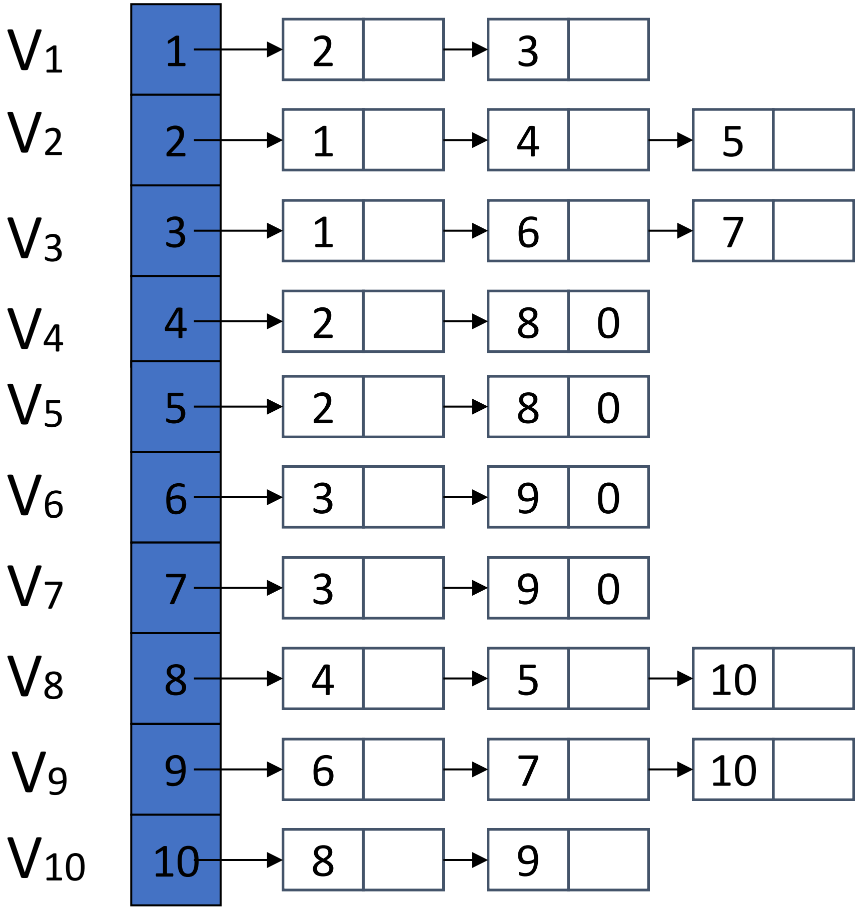


BFS is also known as "level-by-level" traversal (e.g., finding shortest paths in an undirected graph)



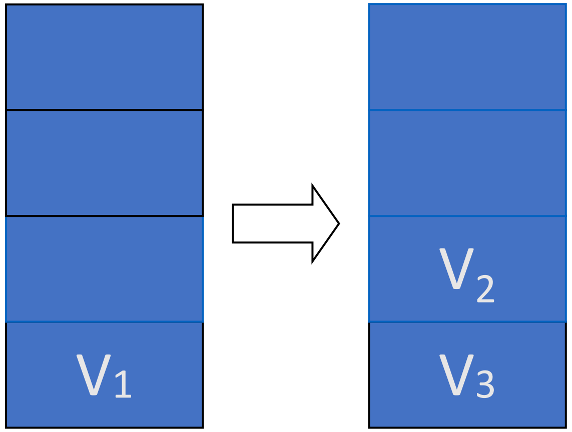
BFS Walkthrough – 1

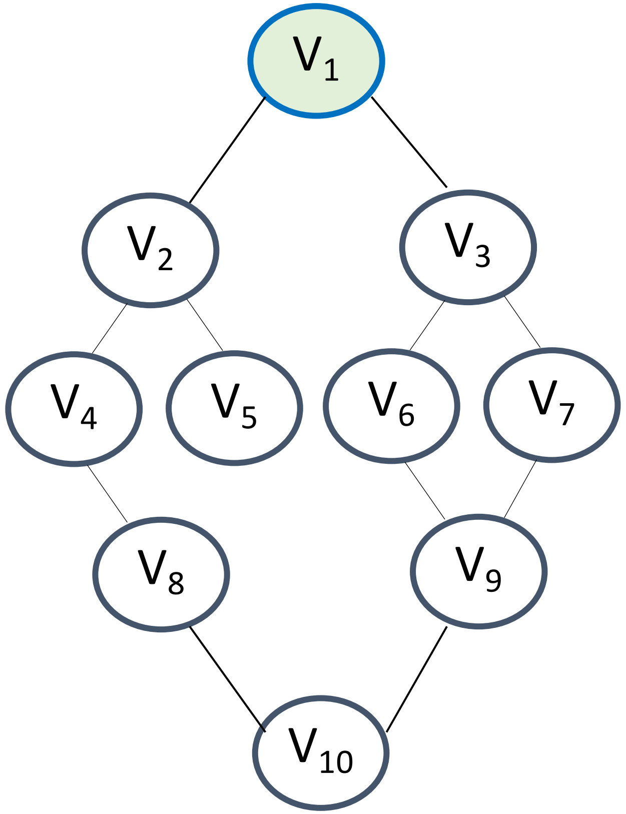




BFS Walkthrough – 2

- **Pop v1 and insert v2 and v3**

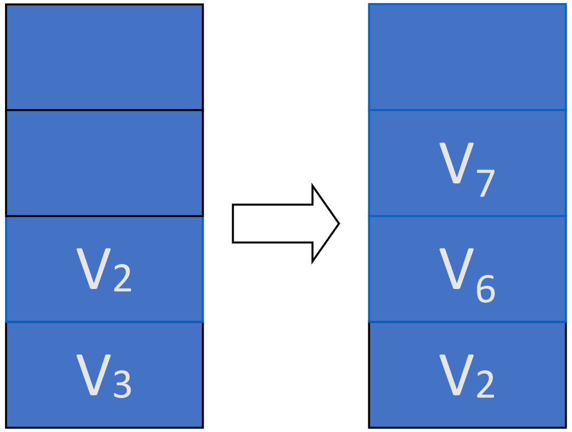


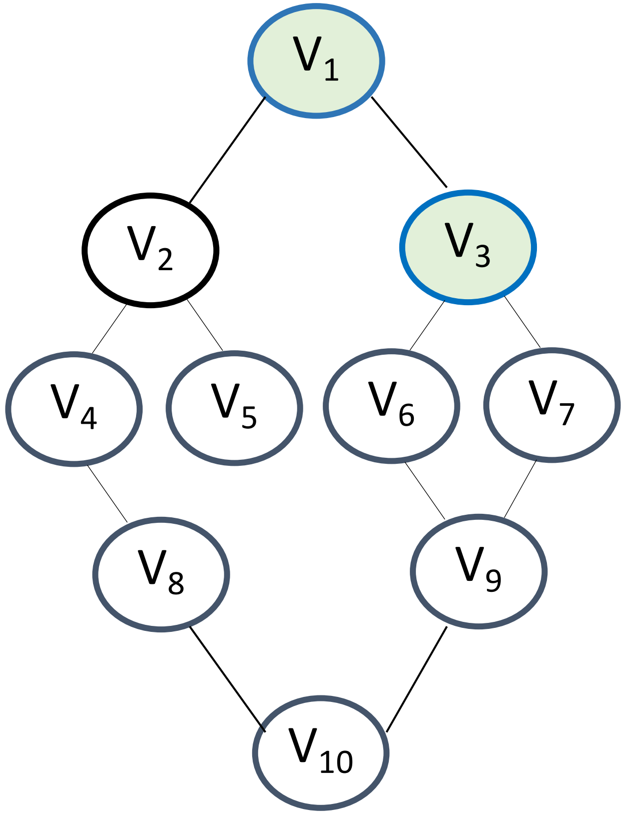


output：V_1

BFS Walkthrough – 3

- Pop v3 and insert v6 and v7

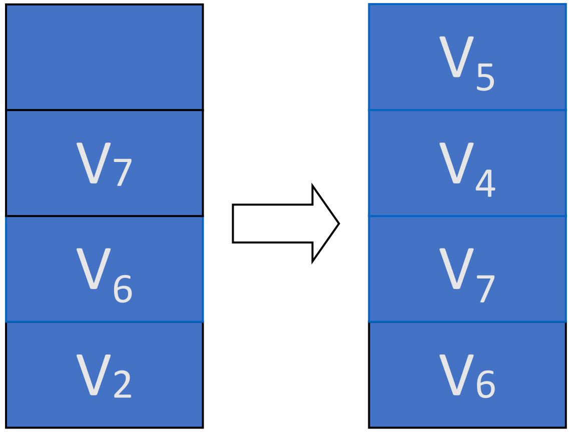


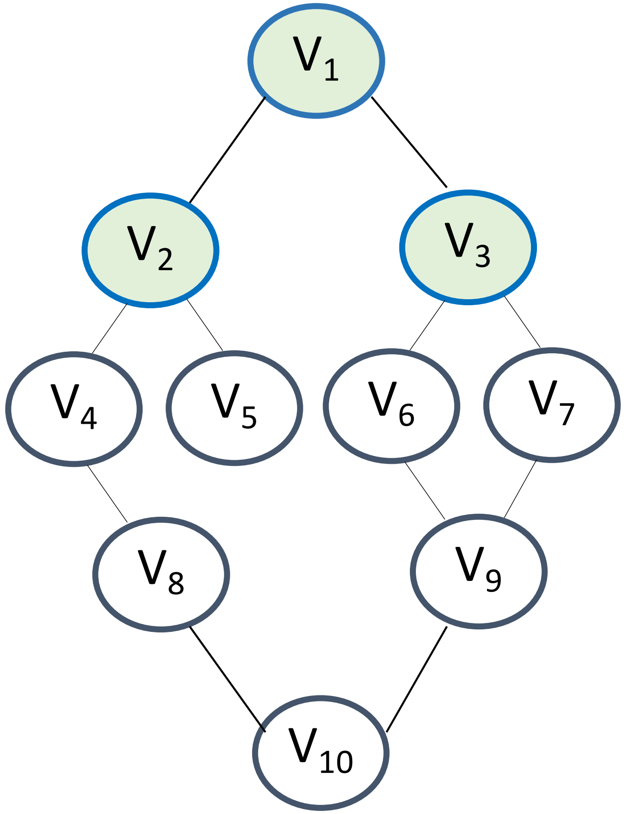


output：V_1 V_3

BFS Walkthrough – 4

- Pop v2 and insert v4 and v5

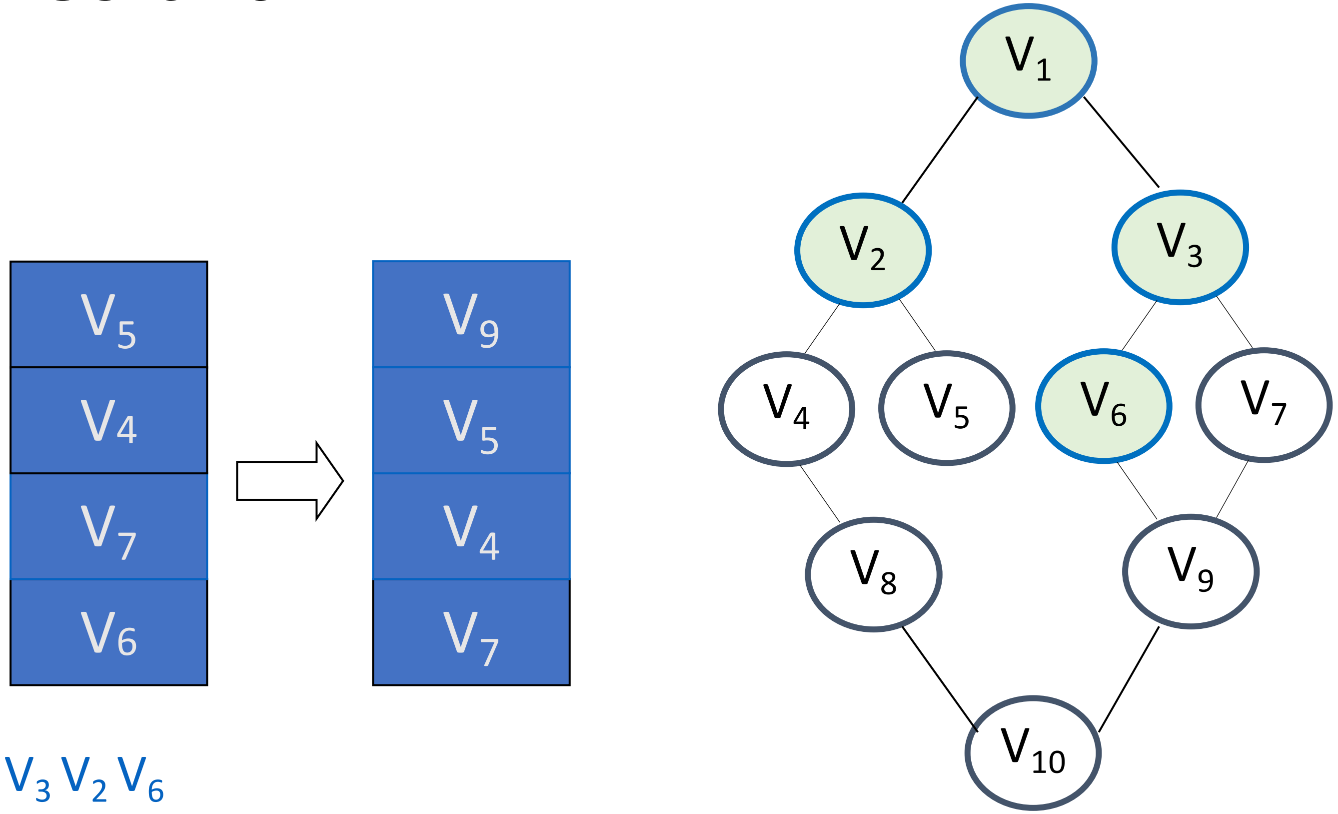




output：V_1 V_3 V_2

BFS Walkthrough – 5

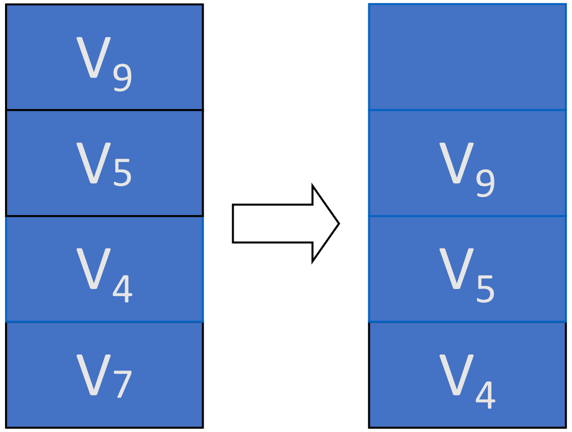
- **Pop v6 and insert v9**



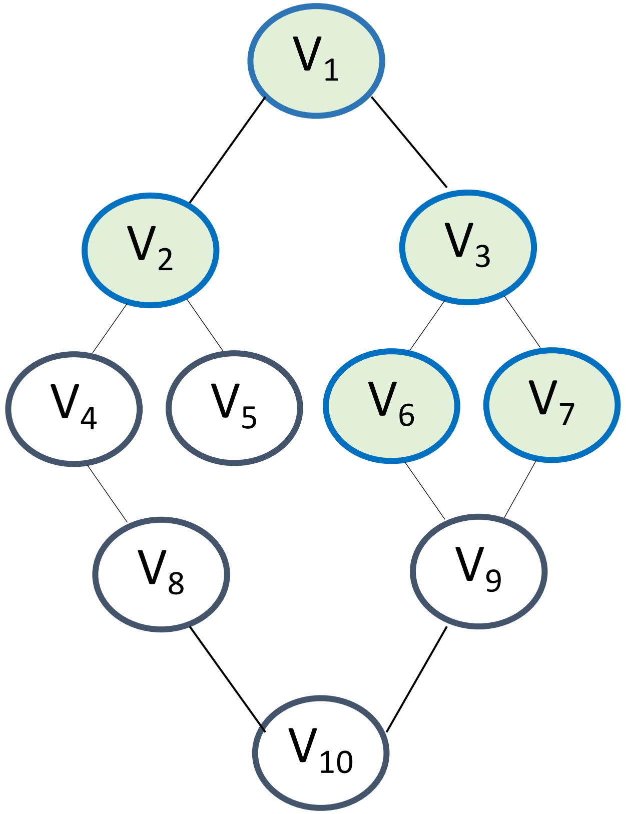
output：V_1 V_3 V_2 V_6

BFS Walkthrough – 6

- **Pop v7 and insert nothing**
  - v9 has been inserted by v6 before!

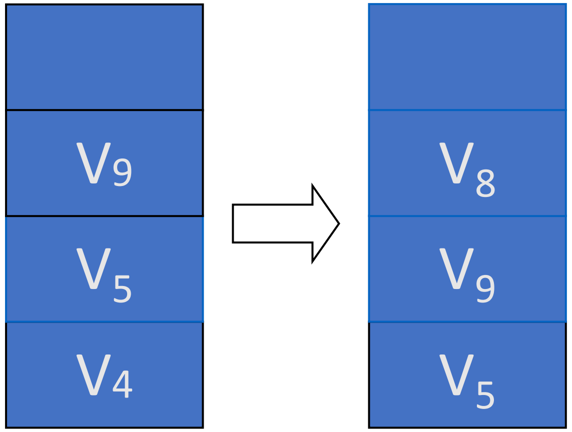


output：$V_1\ V_3\ V_2\ V_6\ V_7$

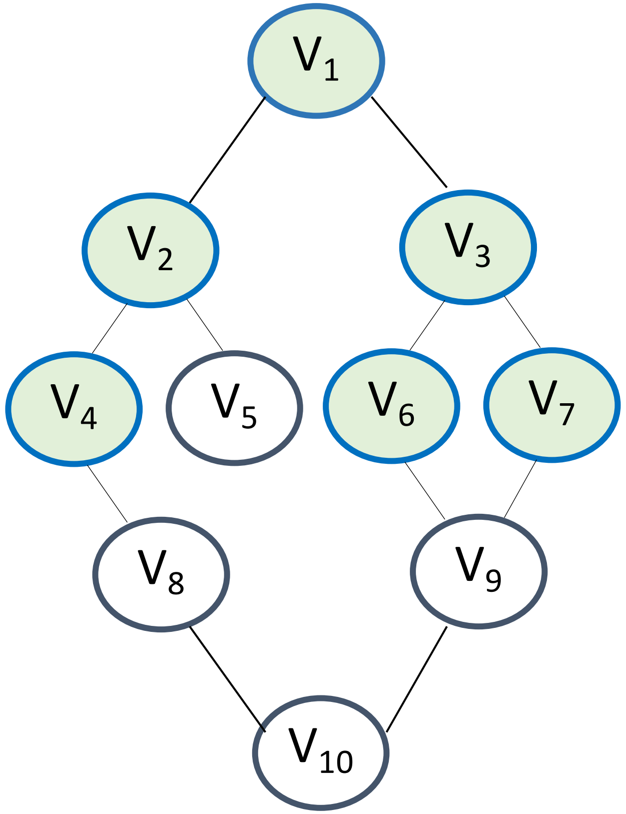


BFS Walkthrough – 7

- **Pop v4 and insert v8**

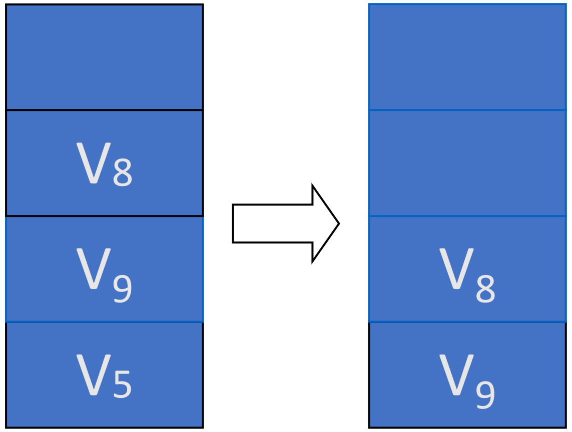


output：V_1 V_3 V_2 V_6 V_7 V_4

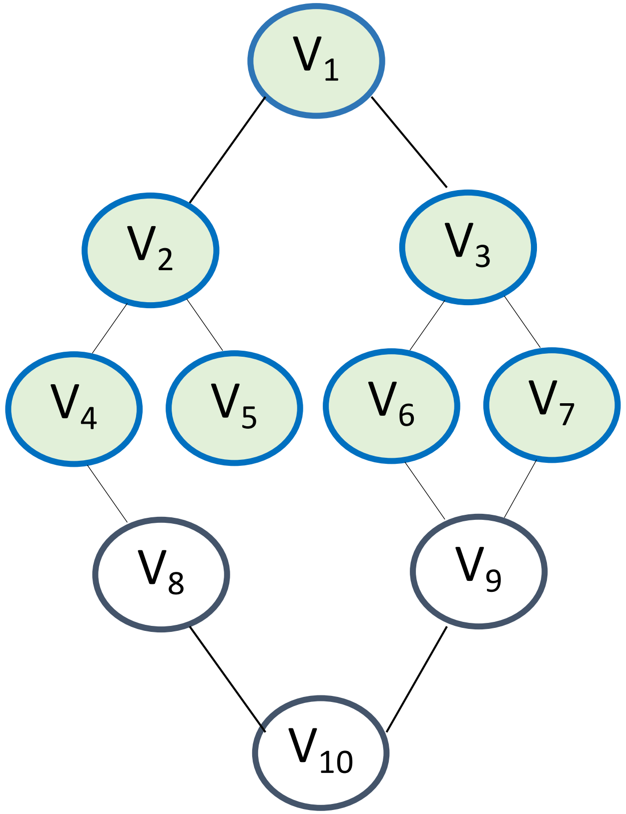


BFS Walkthrough – 8

- **Pop v5**

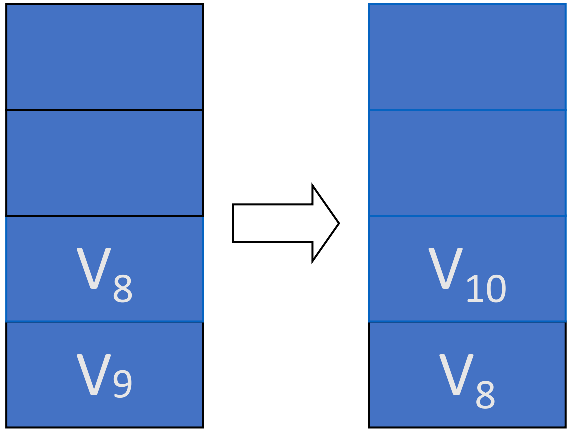


output：V_1 V_3 V_2 V_6 V_7 V_4 V_5

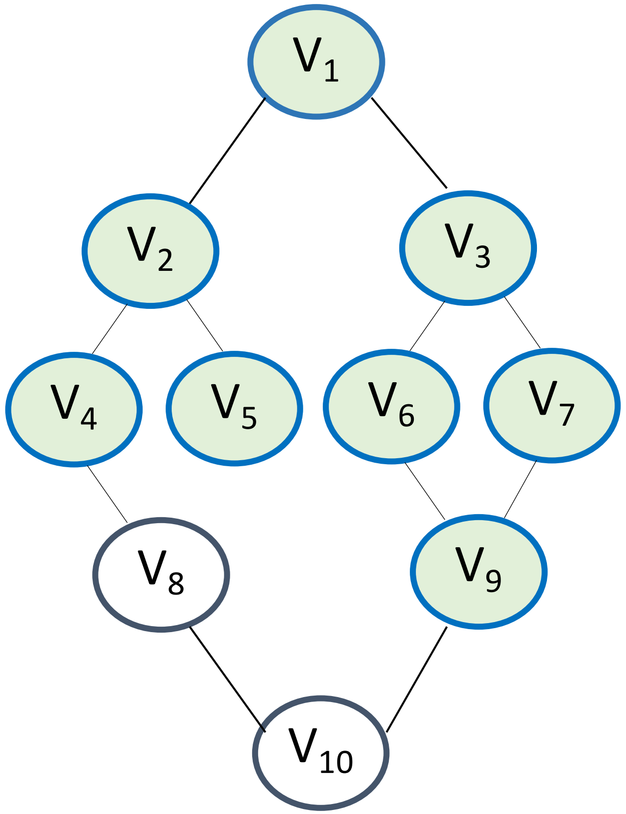


BFS Walkthrough – 9

- **Pop v9 and insert v10**

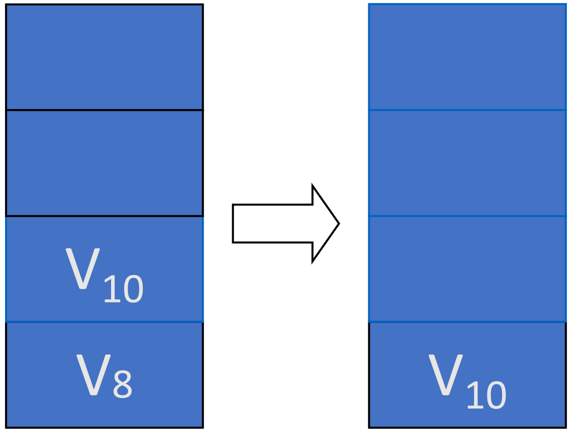


output：V_1 V_3 V_2 V_6 V_7 V_4 V_5 V_9

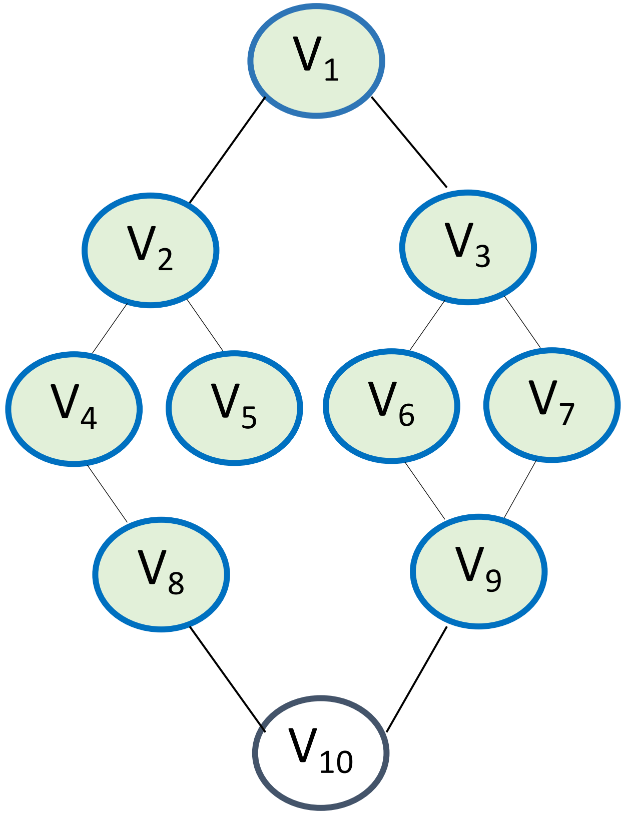


BFS Walkthrough – 10

- **Pop v8 and insert nothing**
  - v10 has been inserted by v9 before!

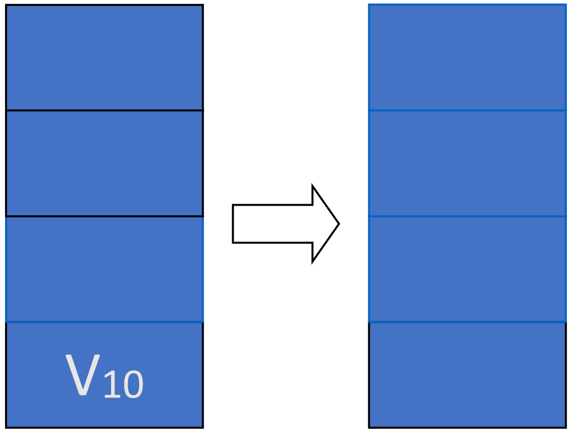


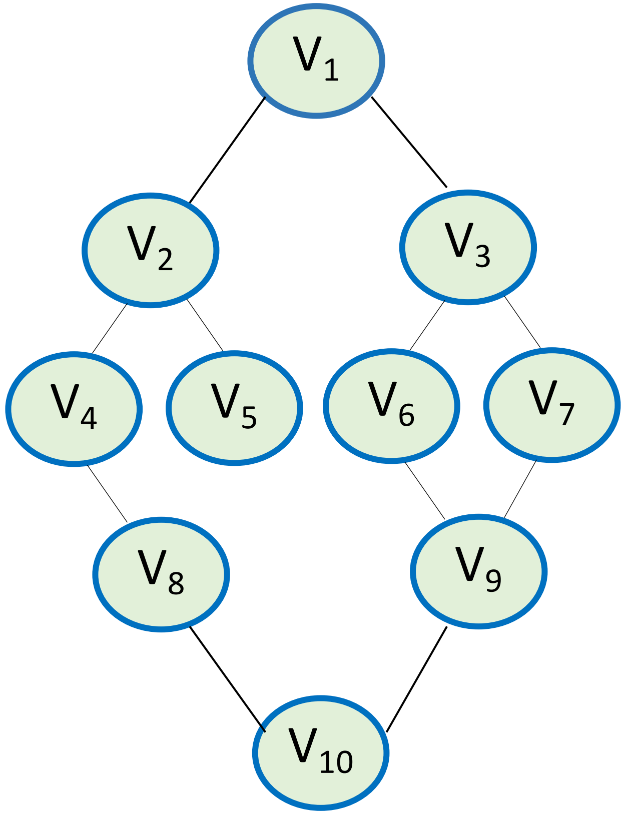
output：$V_1\ V_3\ V_2\ V_6\ V_7\ V_4\ V_5\ V_9\ V_8$



BFS Walkthrough – 11

- **Pop v10**





output：V_1 V_3 V_2 V_6 V_7 V_4 V_5 V_9 V_8 V_{10}

Queue-based Implementation of BFS

```
vector<bool> visited(V, false);
queue<int> queue;
queue.push(s);
while (!queue.empty()) {
    s = queue.top();
    queue.pop();
    // Stack may contain same vertex twice – only print un-visited ones
    if (!visited[s])   {
        cout << s << " ";
        visited[s] = true;
    }
    // Get all un-visited adjacent vertices of the popped vertex s
    for (auto i = adj[s].begin(); i != adj[s].end(); ++i)
        if (!visited[*i])  queue.push(*i);
}
```

Summary

- **We have discussed graph data structures**
  - Vertex and edge definitions
  - Adjacent list and adjacent matrix
- **We have discussed two graph traversal algorithms**
  - Breadth-first-traversal explores vertices in a first-in-first-out order
  - Depth-first-traversal explores vertices in a last-in-first-out order
- **We have discussed two important DFS applications**
  - Cut vertex
  - Cut edge