CSC190: Computer Algorithms and Data Structures

Graphs

Instructor: Pirathayini Srikantha

University of Toronto

March 20, 2017

Readings/References

- Standish (10)

Table of Contents

Table of Contents

Motivation

- A graph consists of a set of nodes that may or may not have **relationships** between one another
- Graphs are a **generalization** of trees with many useful and interesting applications
- **Transportation network**: vertices are cities and links are roads connecting cities
  - Links can represent the distance (**shortest path problem**) or the cost (route with **minimal cost problem**)
- **Oil pipeline network**: vertices are pumping stations and links are pipelines
  - Each link or pipeline can have various diameters (**max flow rate**)
- **Graph exploration**: each vertex is a city and the links are roads
  - **Traversing** a maze (find the exit path from a maze) or the traveller's salesman problem (visit every city before reaching the starting city)

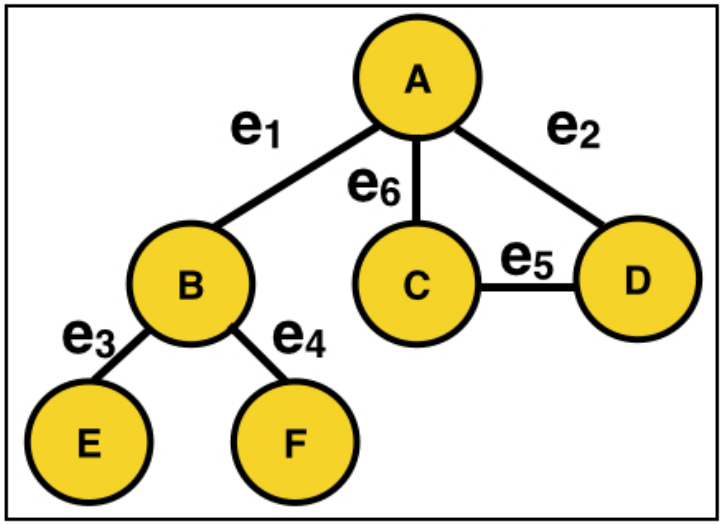
Table of Contents

Edges and Vertices

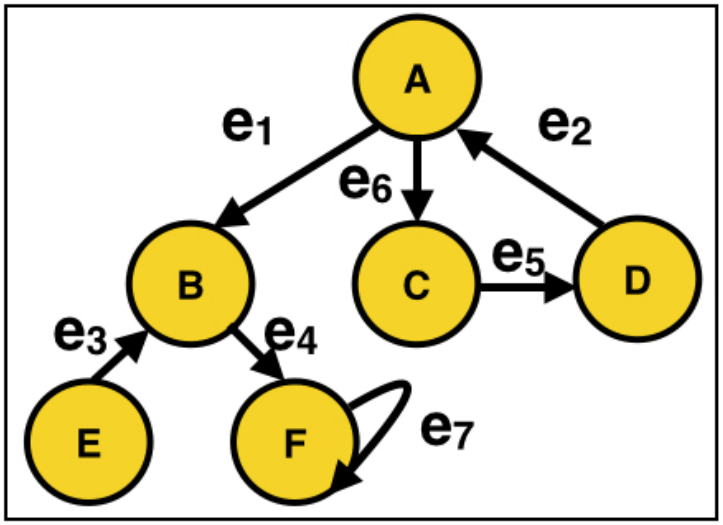
- **Graph** is represented by $G = (V, E)$ where V is a set containing all the vertices in the graph and E contains all the links in the graph
- Suppose that $v_i, v_j \in V$ and if there is a **relationship** between these nodes then this will be represented by an **edge** $e_k \in E$ where $e_k = (v_i, v_j)$
- If the order of the relationship is important then $(v_i, v_j) \neq (v_j, v_i)$ and these graphs are referred to as digraph
  - If $e_k = (v_i, v_j) \in E$ then v_i is the **origin** and v_j is the **terminus** of the edge
- If the order of relationship between nodes need not be preserved then $(v_i, v_j) = (v_j, v_i)$ and these graphs are referred to as **undirected graphs** or just plain graphs

Edges and Vertices

Undirected



Directed



$V=\{A, B, C, D, E, F\}$

$E=\{e_1,e_2,e_3,e_4,e_5,e_6\}$

$e_1=(A,B),\ e_1=(B,A)$

$E=\{e_1,e_2,e_3,e_4,e_5,e_7\}$

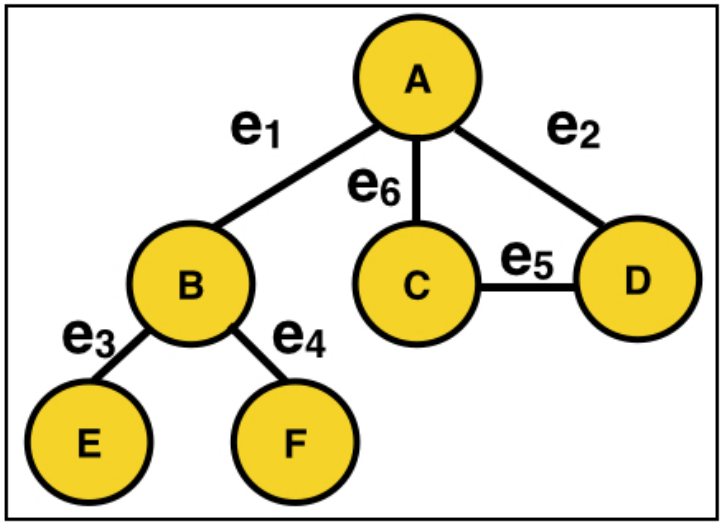
$e_1=(A,B),\ e_1\neq(B,A)$

Paths, Cycles, and Adjacency

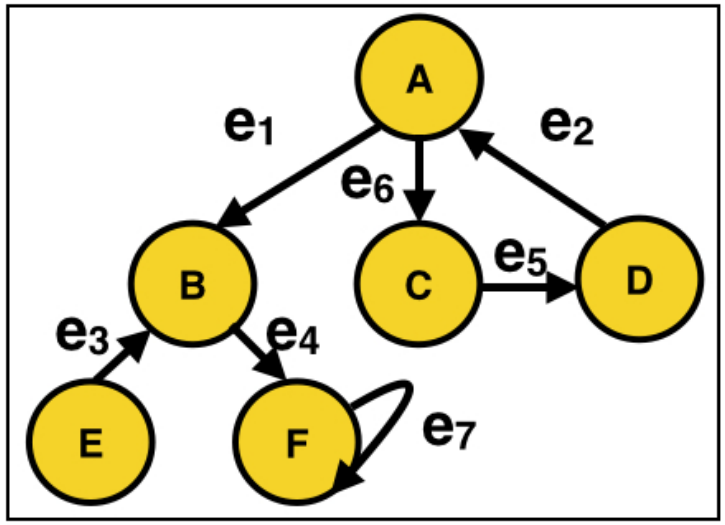
- **Adjacency:** If two vertices are connected by an edge $(v_i, v_j) = e$ then v_j is adjacent to v_i
- **Path:** A collection of edges $p = e_1, e_2, ...e_n$ where at least one node in each edge is adjacent to a node in the consecutive edge
  - E.g. $(v_i, v_j), (v_j, v_k), (v_k, v_l), \ldots$
  - An alternate representation of a path is $p = v_i v_j \ldots v_n$
- **Cycle:** A special type of path $p = v_i v_j \ldots v_n$ where $v_i = v_n$
- Simple cycle: All nodes (with the exception of the first and last nodes) occur only once in a path forming a simple cycle ($n > 3$)

Paths, Cycles, and Adjacency

Undirected



Directed



$V=\{A, B, C, D, E, F\}$

$E=\{e_1,e_2,e_3,e_4,e_5,e_6\}$

$e_1=(A,B)$, $e_1=(B,A)$

$P_{AE}=\{e_1,e_3\}$

$P_{AF}=\{e_1,e_4\}$

Cycle: $P_{AA}=\{e_6,e_5,e_2\}$

$E=\{e_1,e_2,e_3,e_4,e_5,e_7\}$

$e_1=(A,B)$, $e_1\neq(B,A)$

$P_{AE}=NA$

$P_{AF}=\{e_1,e_4\}$

Cycle: $P_{AA}=\{e_6,e_5,e_2\}$

Cycle: $P_{FF}=\{e_7\}$

Adjacency Sets

- **Adjacency Sets**: Provides an equivalent definition of a graph

$$G = (V, E)$$

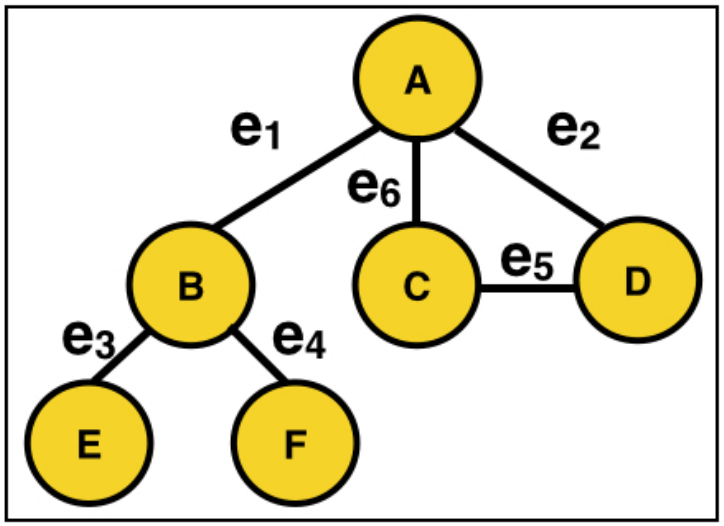
- V_x: set of all vertices adjacent to v_x

$$V_x = \{v_y | (v_x, v_y) \in E\}$$

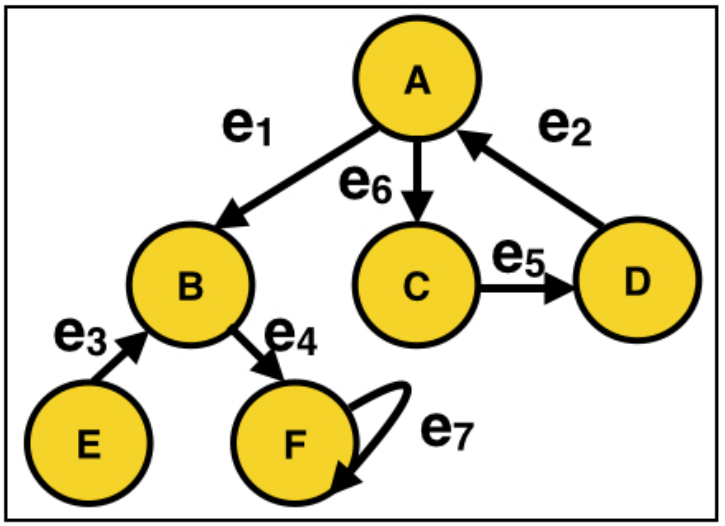
- The set $A = \{V_x | x \in V\}$ completely represents the graph G and therefore A provides an equivalent definition of G

Adjacency Sets

Undirected



Directed



$V=\{A, B, C, D, E, F\}$

$E=\{e_1,e_2,e_3,e_4,e_5,e_6\}$

$e_1=(A,B)$, $e_1=(B,A)$

$V_A=(B,C,D)$

$E=\{e_1,e_2,e_3,e_4,e_5,e_7\}$

$e_1=(A,B)$, $e_1 \neq (B,A)$

$V_A=(B,C)$

Vertex Metrics

Undirected Graph

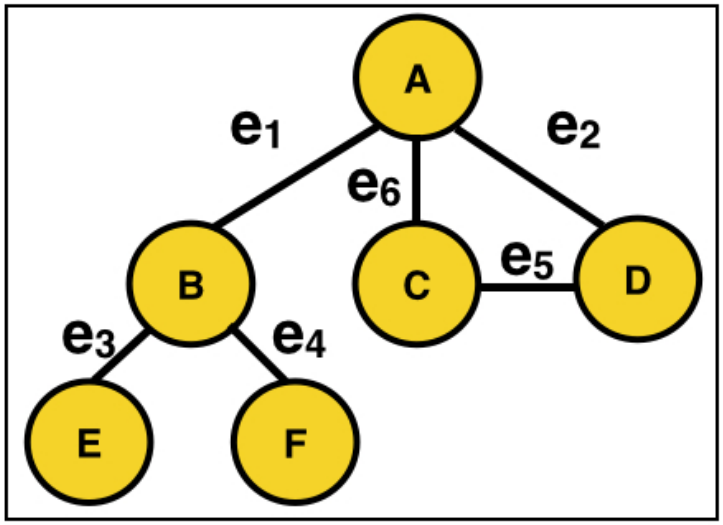
- **Degree** of a vertex v_x: Number of edges in which v_x is one of the endpoint

Directed Graph

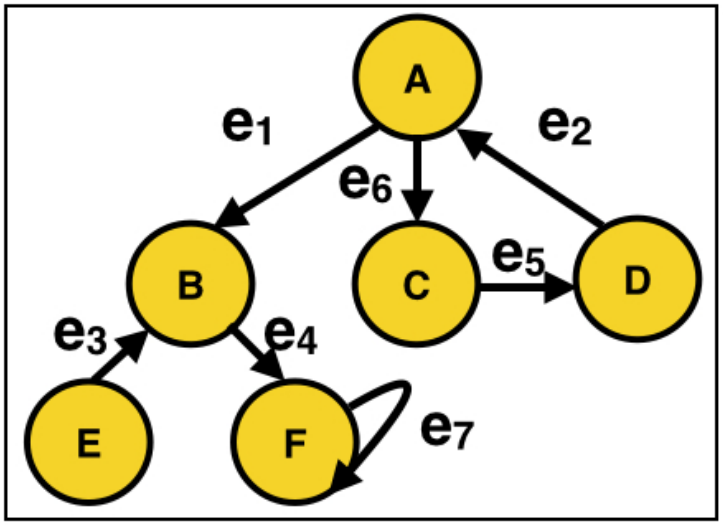
- **Predecessors** of v_x: $Pred(v_x) = \{v_y | (v_y, v_x) \in E\}$
- **Successors** of v_x: $Succ(v_x) = \{v_y | (v_x, v_y) \in E\}$
- **In-degree** of v_x: number of nodes in $Pred(v_x)$
- **Out-degree** of v_x: number of nodes in $Succ(v_x)$

Vertex Metrics

Undirected



Directed



V={A, B, C, D, E, F}

E={e_1,e_2,e_3,e_4,e_5,e_6}

Deg(A)=3

E={e_1,e_2,e_3,e_4,e_5,e_7}

Pred(A)=D

Succ(A)=B,C

In-degree(A)=1

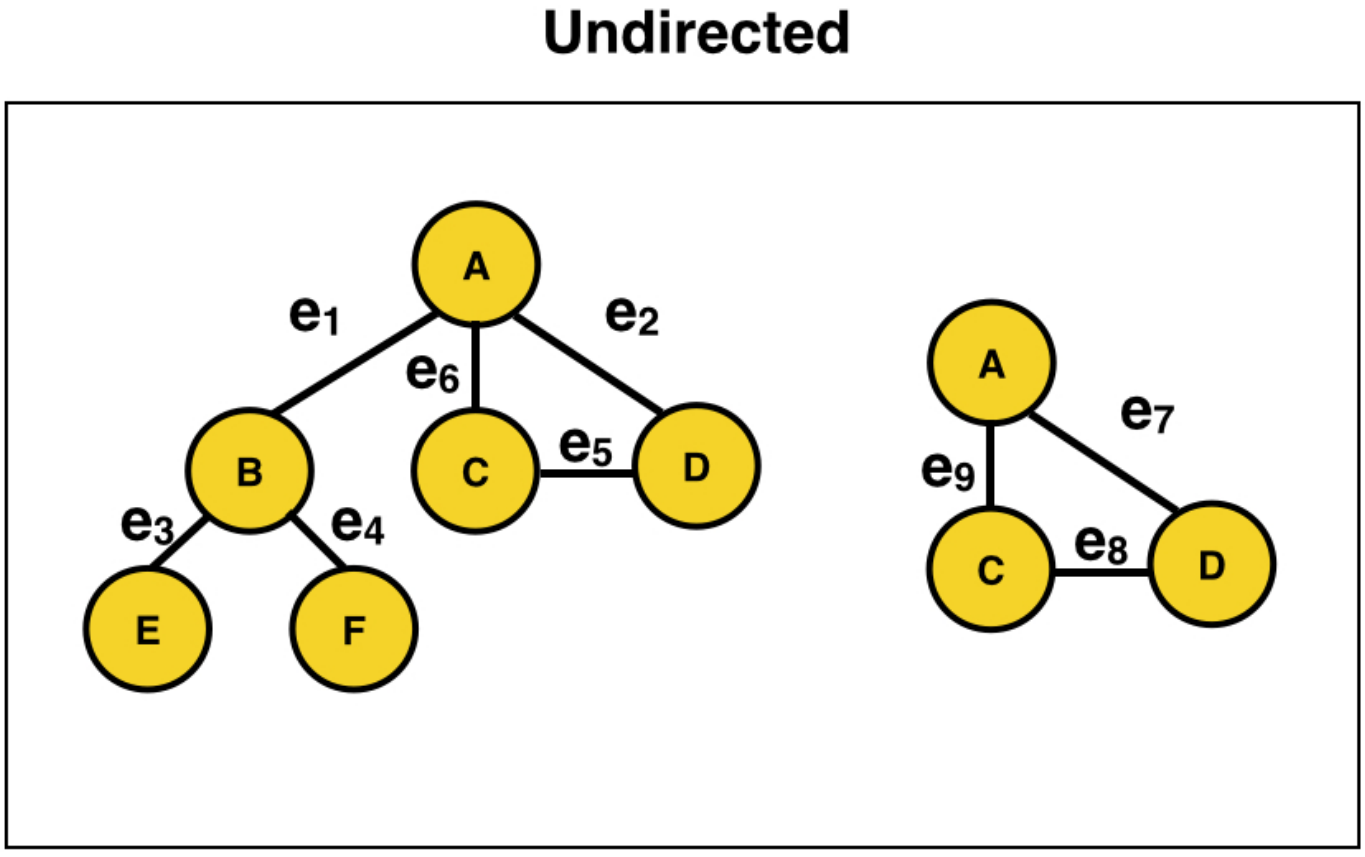
Out-degree(A)=2

Connectivity and Components

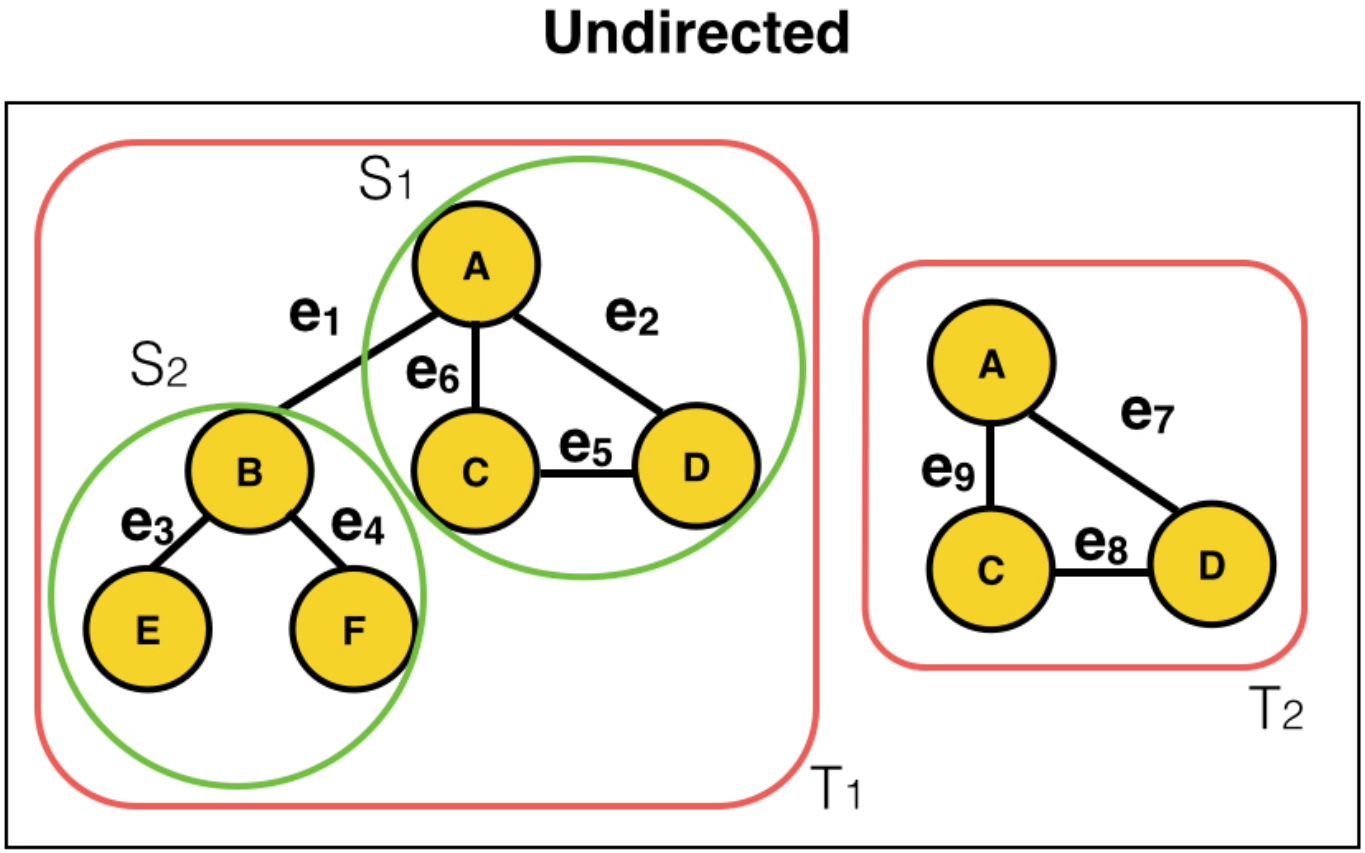
Undirected Graph: Ordering does not matter

- **Connected**: $v_i, v_m \in V$ are connected if there exists a path between these vertices
- **Connected Component S**: $S \subset V$ is a set containing vertices that are all connected to one another
- **Maximal Connected Component T**: Largest subset of nodes that are all connected to one another
- If an undirected graph has distinct $T_1, T_2 \ldots T_k$ then $T_i \cap T_j = \varnothing \ \forall i, j \in \{1 \ldots k\}$

Connectivity and Components



Connectivity and Components



$S_1 \subset T_1$

$S_2 \subset T_1$

$T_2 \cap T_1 = \varnothing$

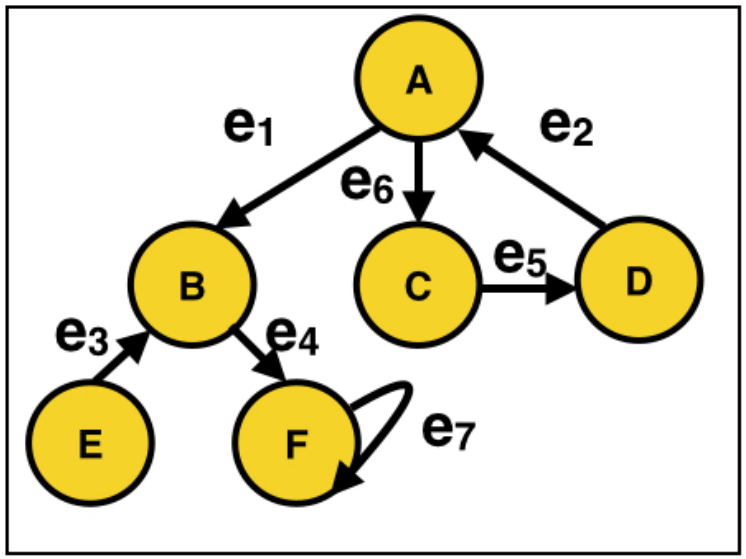
Connectivity and Components

Directed Graph: Ordering does matter

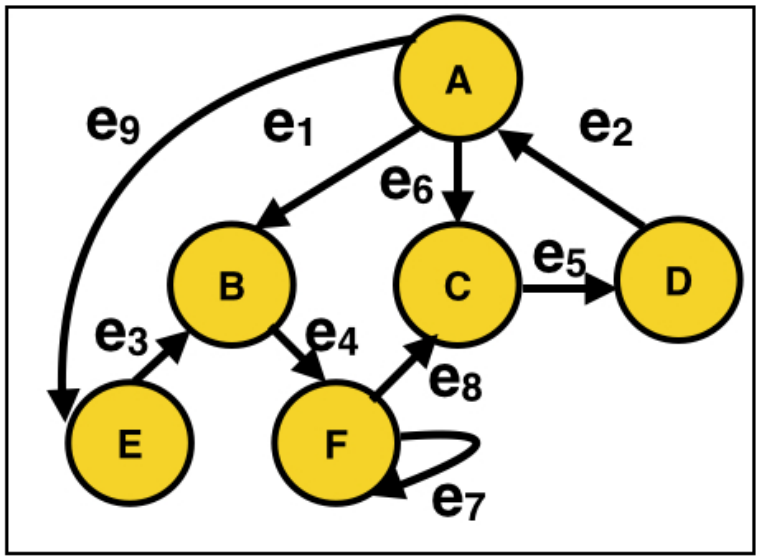
- **Connected**: $v_i, v_m \in V$ are connected if there exists a path between these vertices (direction of edges must be same in the path)
- **Strongly connected S**: All $v_i, v_m \in S$ are connected to one another
- **Weakly connected S**: $v_i, v_m \in S$ where either v_i is connected to v_j or v_j is connected to v_i (both statements are not true at once)

Connectivity and Components

Directed



Weakly Connected



Strongly Connected

Table of Contents

Graph Representation

- In a **graph representation**, it is necessary to include all vertices/nodes (V) and edges present (E) in the graph
- Two different methods can capture all necessary information about a graph
- **Adjacency Matrices**
- **Adjacency Lists**

Adjacency Matrix

Graph Representation

An **adjacency matrix** A representation of a graph is composed of n rows and n columns (i.e. there are n vertices/nodes in the graph)

Undirected Graph

- If there is an edge between (v_i, v_j), then both entries $A[i][j]$ and $A[j][i]$ are set to 1
- Thus A is a **symmetric** matrix

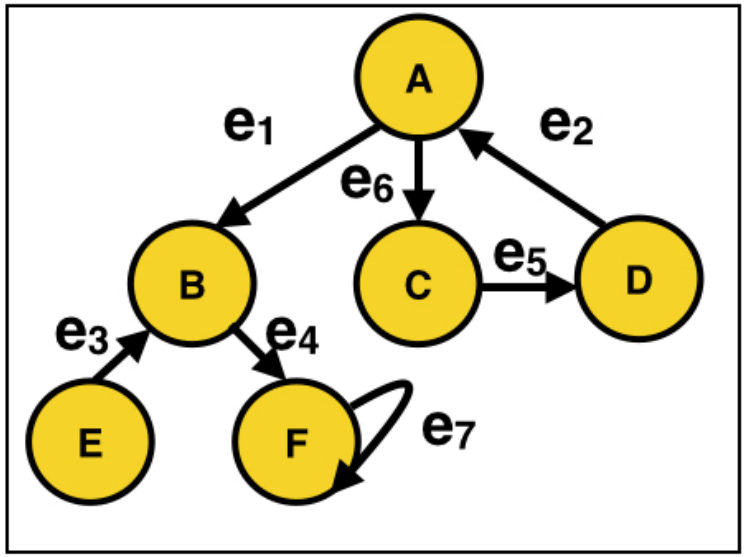
Directed Graph

- Rows are originating nodes and columns are terminus nodes
- If $(v_i, v_j) \in E$, then $A[i][j]$ is set to 1
- When (v_i, v_j) do not form an edge, $A[i][j]$ is set to 0
- A is not always symmetric

If edges have weight, then weights are listed explicitly in the corresponding entry instead of 1

Graph Representation: Adjacency Matrix Example

Directed Graph *G*

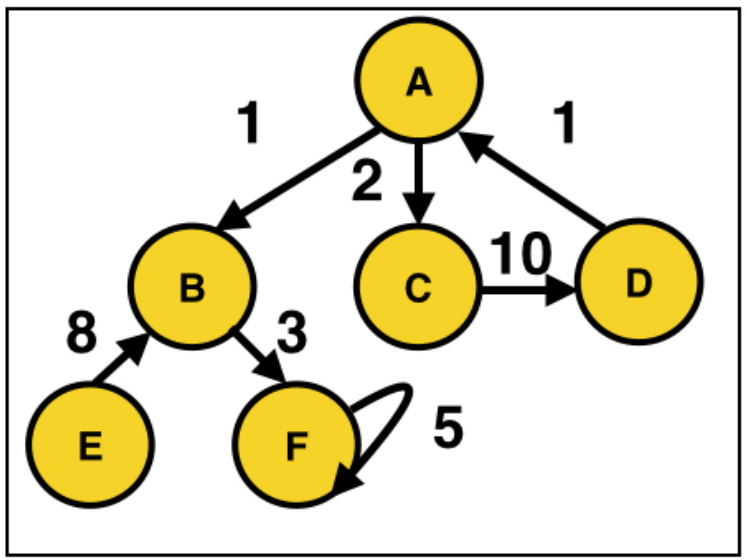


Adjacency Matrix for *G*

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

Graph Representation: Adjacency Matrix Example

Weighted Directed Graph *G*

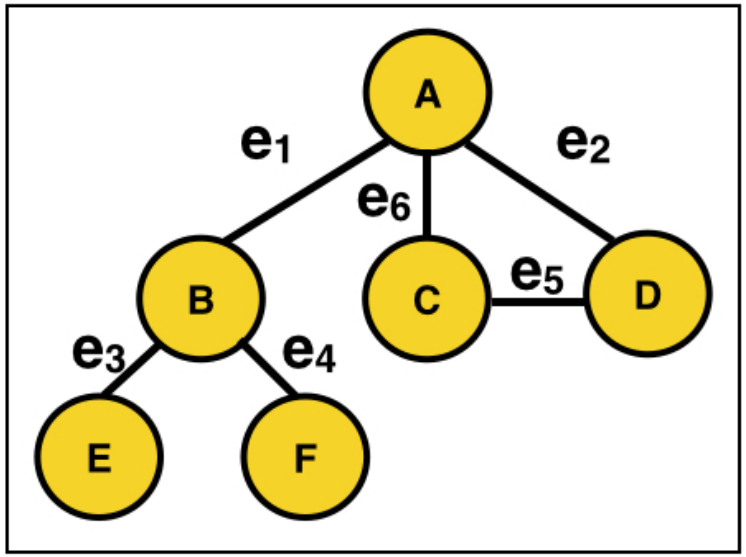


Adjacency Matrix for *G*

	A	B	C	D	E	F
A	0	1	2	0	0	0
B	0	0	0	0	0	3
C	0	0	0	10	0	0
D	1	0	0	0	0	0
E	0	8	0	0	0	0
F	0	0	0	0	0	5

Graph Representation: Adjacency Matrix Example

Undirected Graph *G*

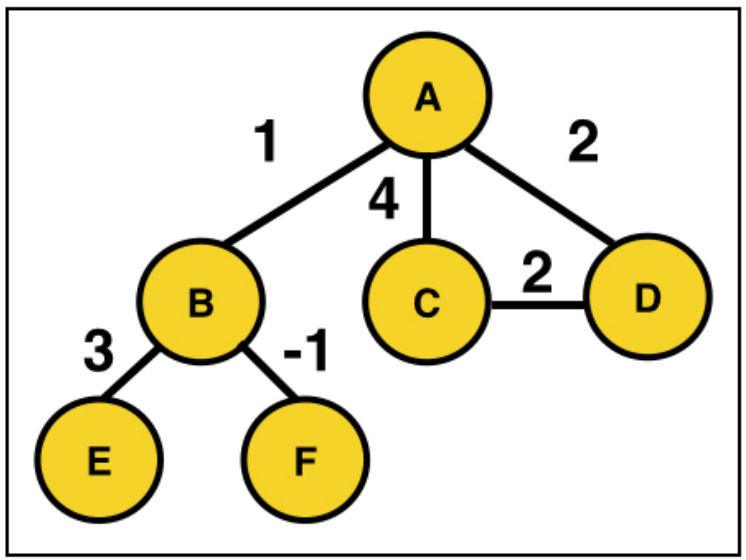


Adjacency Matrix for *G*

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

Graph Representation: Adjacency Matrix Example

Weighted Undirected Graph *G*



Adjacency Matrix for *G*

	A	B	C	D	E	F
A	0	1	4	2	0	0
B	1	0	0	0	3	-1
C	4	0	0	2	0	0
D	2	0	2	0	0	0
E	0	3	0	0	0	0
F	0	-1	0	0	0	0

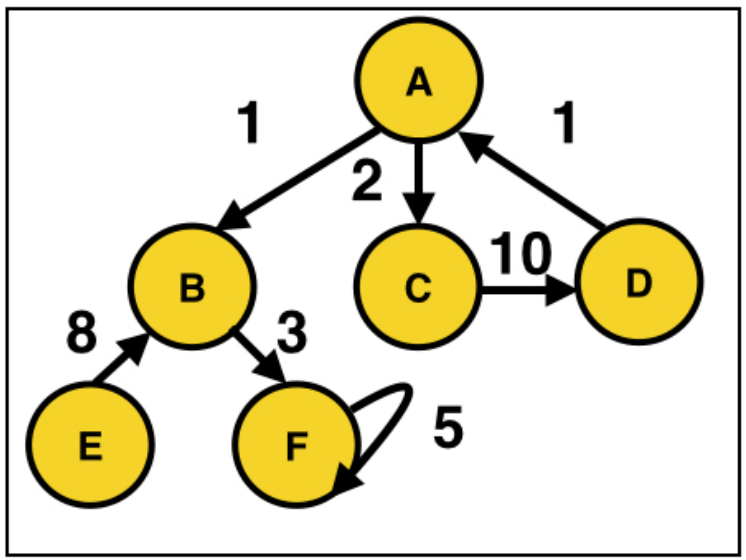
Adjacency List

Graph Representation

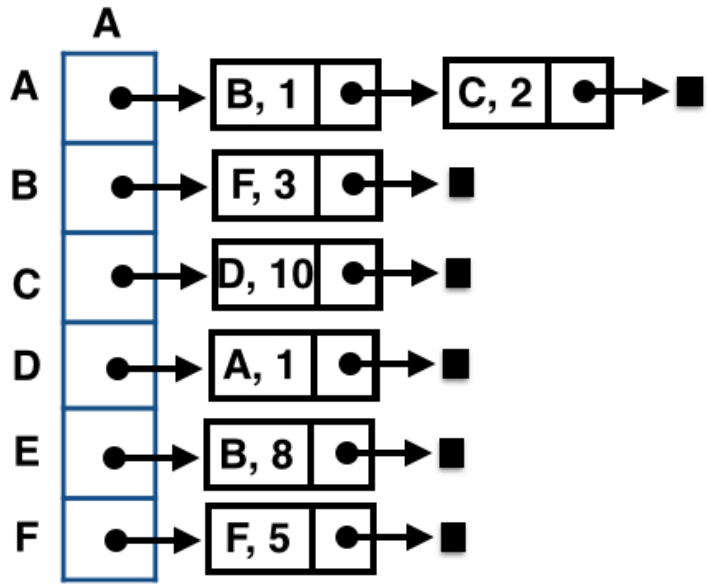
- An **adjacency list** is another method of representing a graph
- **Linked lists** are engaged to form an **adjacency list**
- Information about every combination of edges possible in the graph is **not** stored in adjacency lists
- An array of n pointers is created where each element in the array represents a node in the graph
- A linked list residing in the i^{th} index of the array represents all edges where v_i is the originating node (all non-zero columns for the i^{th} entry in the adjacency matrix)
- The j^{th} node in the i^{th} linked list represents a terminating node forming the edge (v_i, v_j)

Graph Representation: Adjacency List Example

Weighted Directed Graph *G*

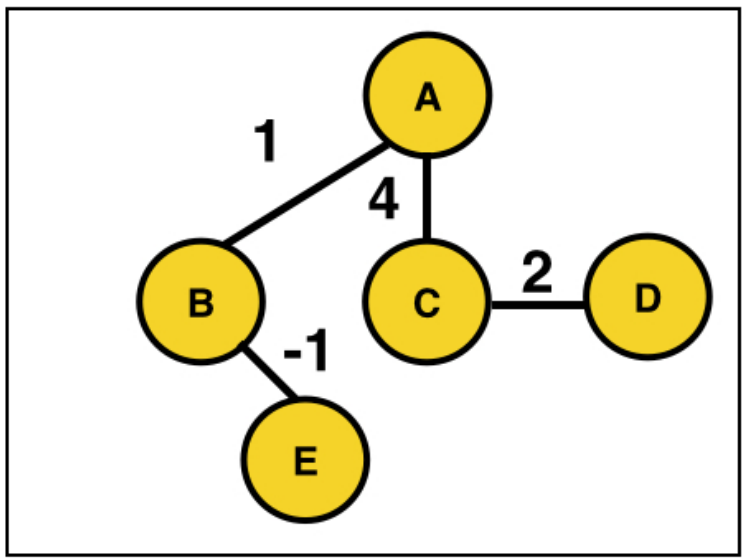


Adjacency Matrix for *G*

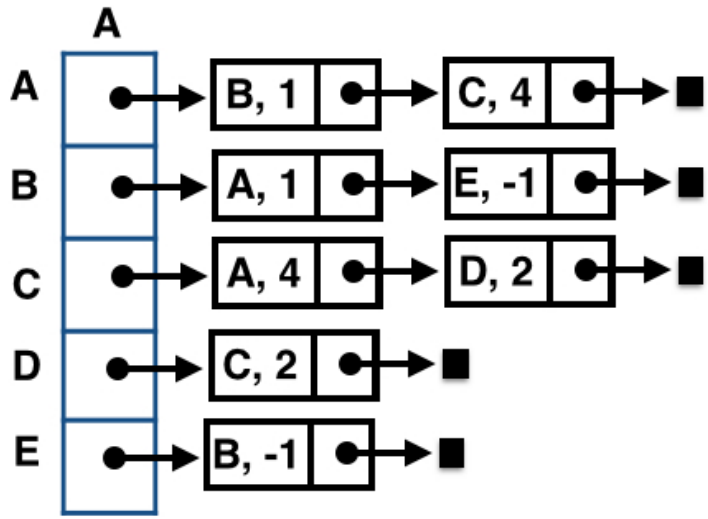


Graph Representation: Adjacency List Example

Weighted Undirected Graph *G*



Adjacency Matrix for *G*



Graph Representation: Definition of Structures

```
#define NODES 8


struct listNode{
    int vertexLabel;
    struct listNode * next;
};
struct adjList{
    int visitedList[NODES];
    struct listNode * ptrArray[NODES];
};

struct adjMatrix{
    int matrix[NODES][NODES];
};
```

Table of Contents

Table of Contents

Graph Traversal

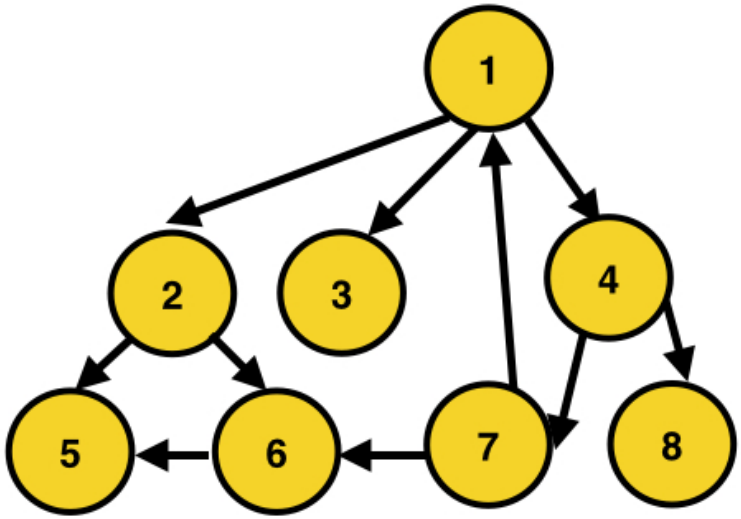
- In order to perform a search on a graph G, it is necessary to **traverse** every node in the graph
- A graph search will begin from a **starting** vertex $\mathtt{v}$
- All nodes in the graph can be traversed from $\mathtt{v}$ only if all other vertices are **accessible** from $\mathtt{v}$
- Suppose that all nodes are accessible from $\mathtt{v}$, can we employ in-order, pre-order or post-order traversals as we did in trees?
- Need to take care of **cycles** to prevent from being stuck in it infinitely by **marking** nodes already visited
- Two types of graph traversals: **breadth-first** and **depth-first**

Graph Traversal

- **Breadth-first** traversal is similar to level-order traversal as all siblings are visited first before visiting the descendants
- Can use **queues** for implementing breadth first traversal
- **Depth-first** traversal visits descendants first and then visits siblings
- Can use **stacks** for implementing depth first traversal

Breadth-First Traversal

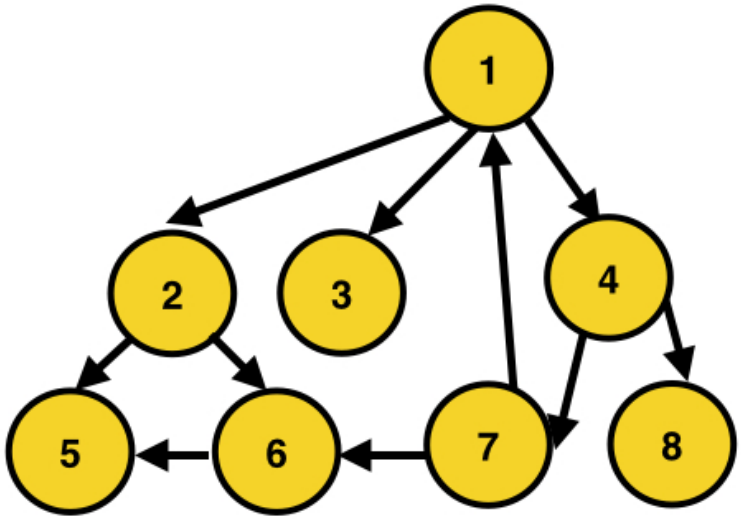
Graph Traversal: Breadth-first

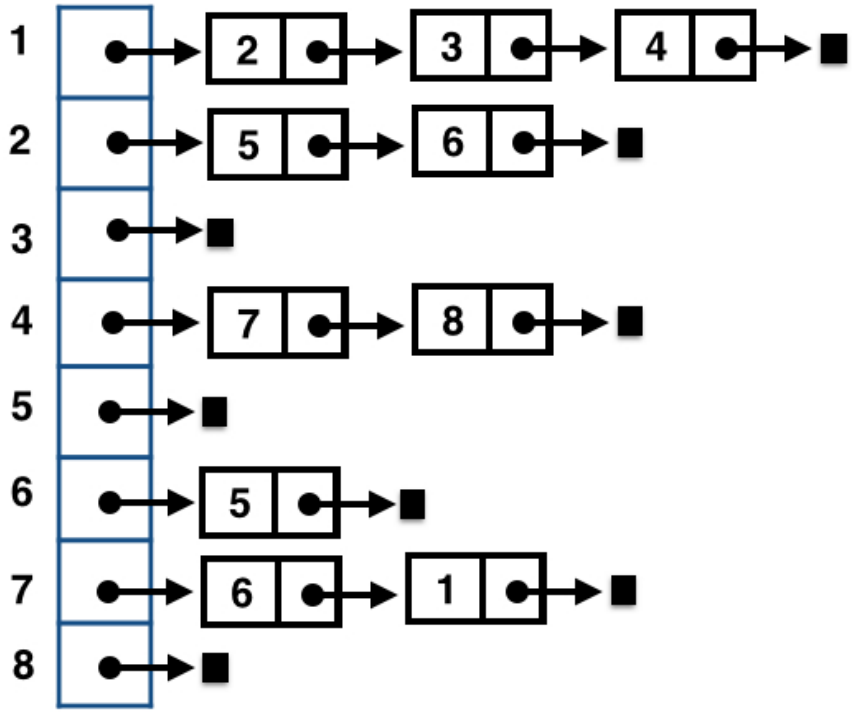


Breadth-First Traversal: 1 2 3 4 5 6 7 8

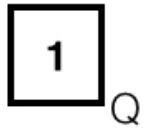
Can perform breadth-first traversal by using
a queue and marking nodes that are visited

Graph Traversal: Breadth-first

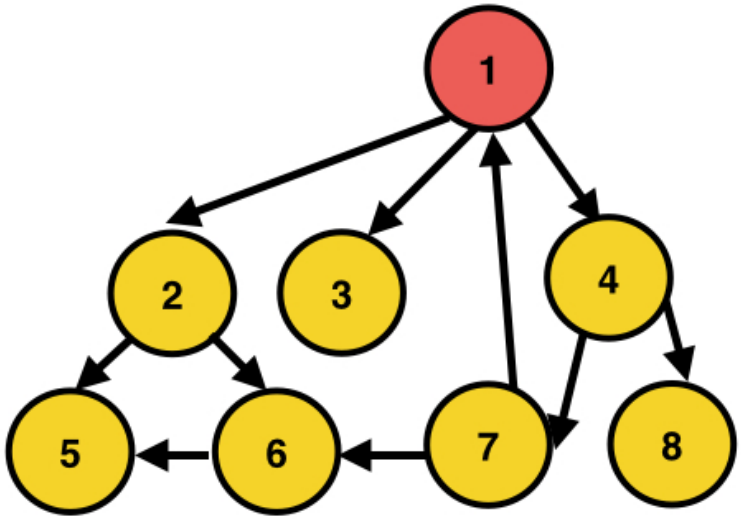


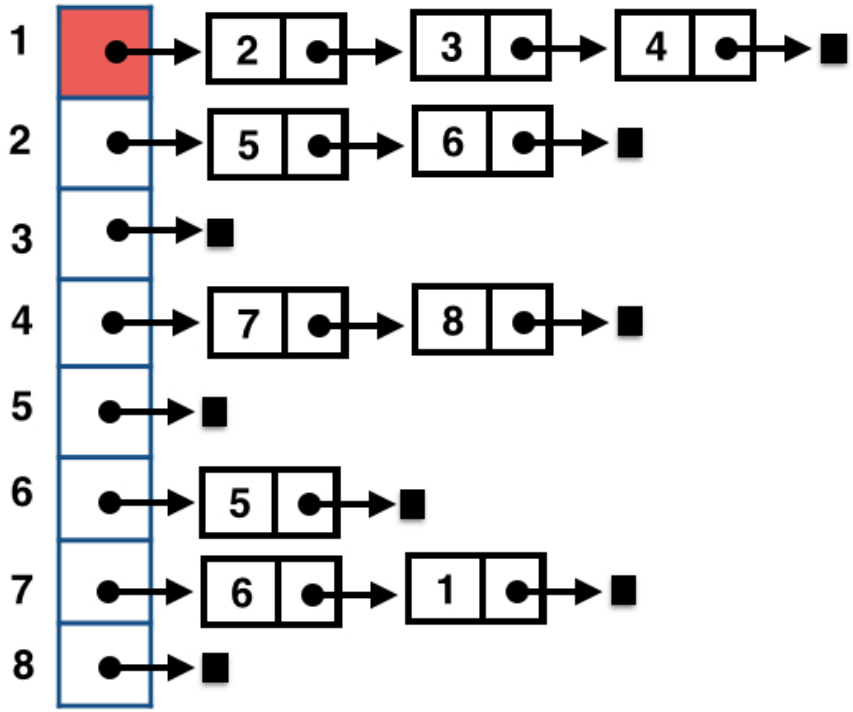


Enqueue the starting node into the queue Q



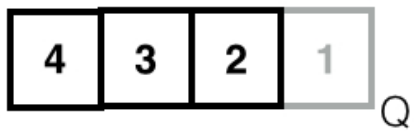
Graph Traversal: Breadth-first



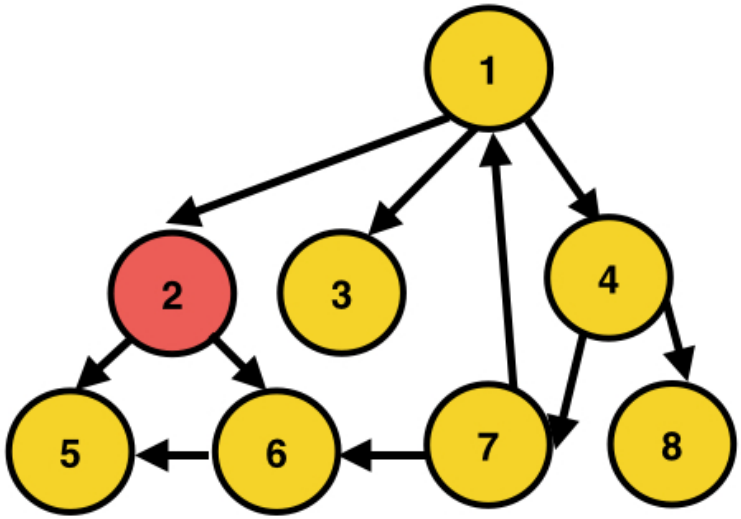


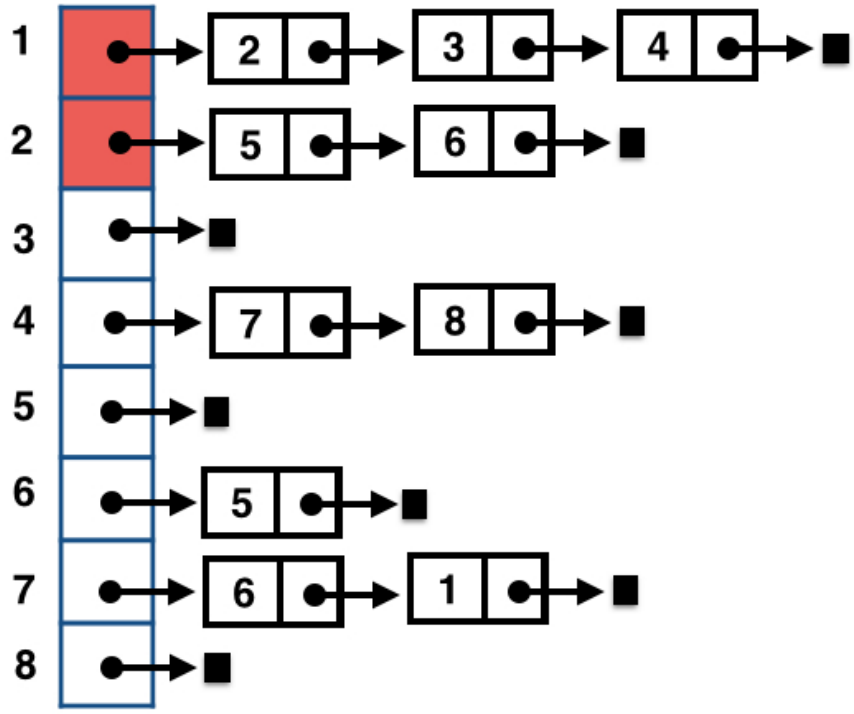
Dequeue a node from queue Q, print its content to the console,
mark the node as visited and enqueue the dequeued node's adjacent nodes into Q

Console: 1



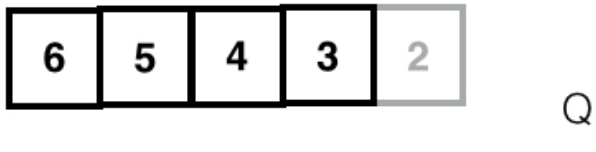
Graph Traversal: Breadth-first



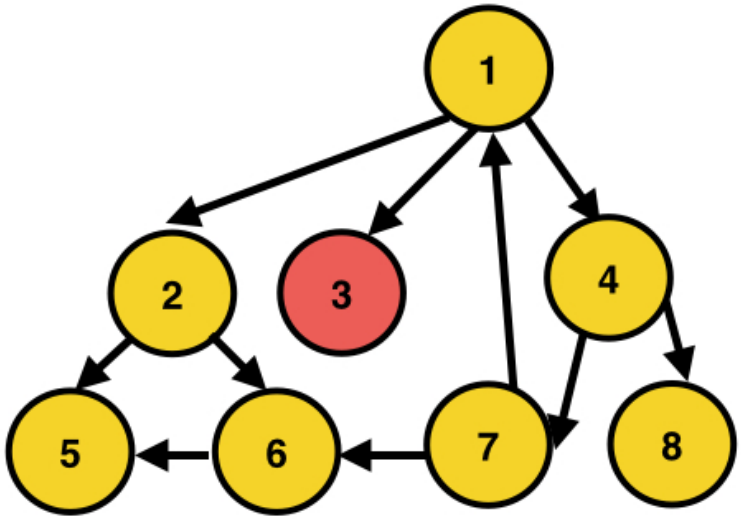


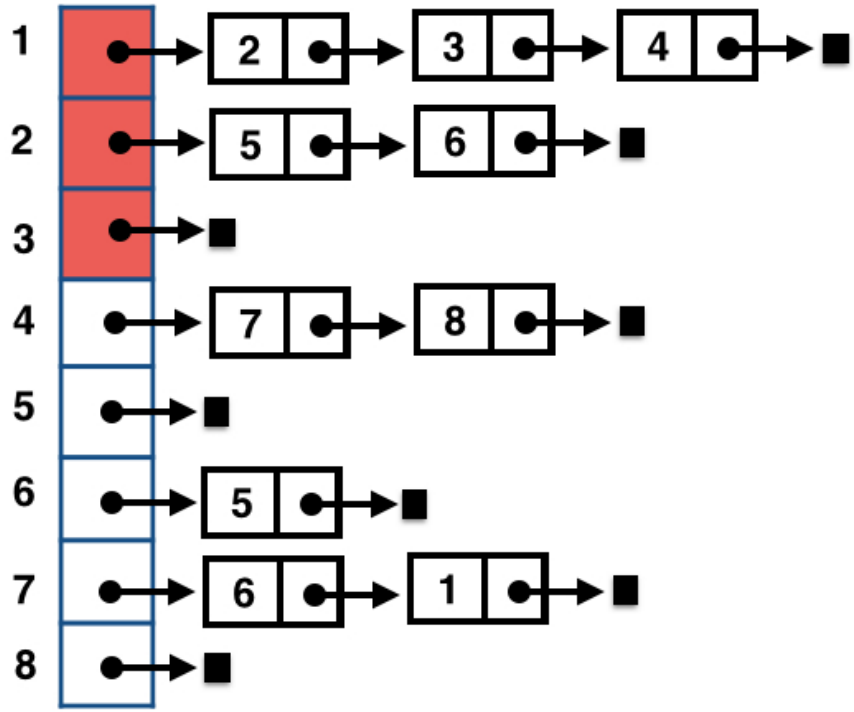
Dequeue a node from queue Q, print its content to the console,
mark the node as visited and enqueue the dequeued node's adjacent nodes into Q

Console: 1 2



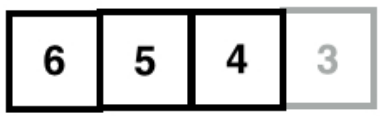
Graph Traversal: Breadth-first





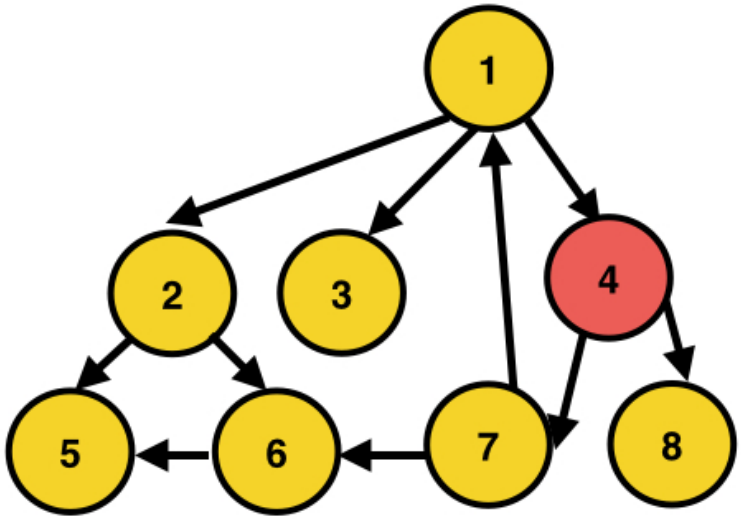
Dequeue a node from queue Q, print its content to the console,
mark the node as visited and enqueue the dequeued node's adjacent nodes into Q

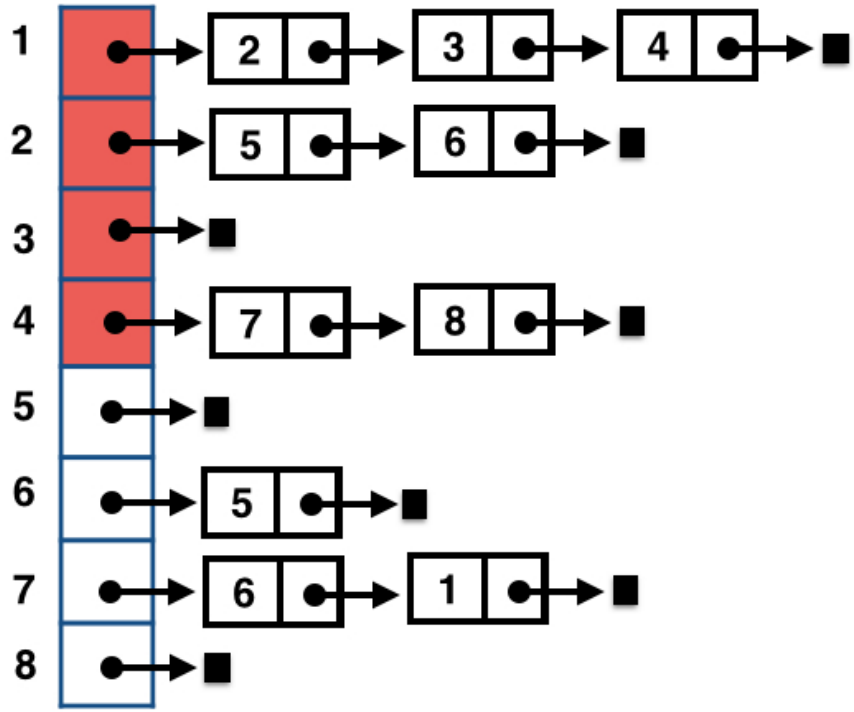
Console: 1 2 3



Q

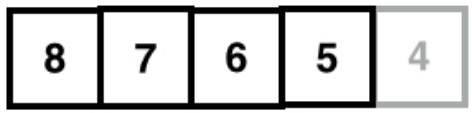
Graph Traversal: Breadth-first





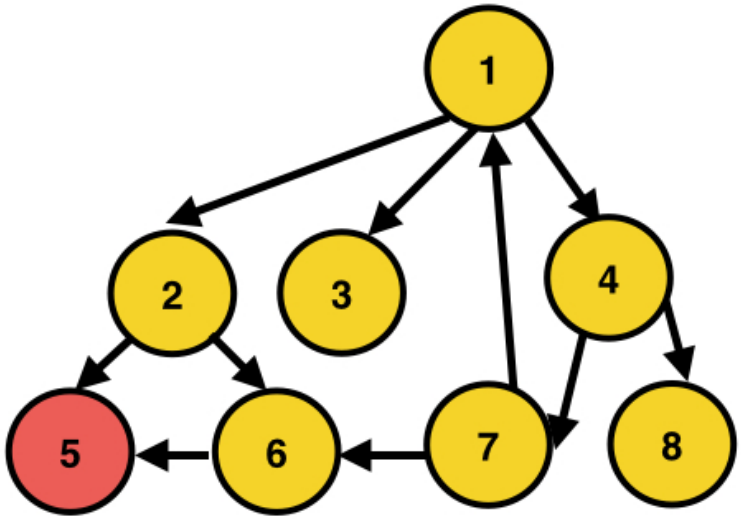
Dequeue a node from queue Q, print its content to the console,
mark the node as visited and enqueue the dequeued node's adjacent nodes into Q

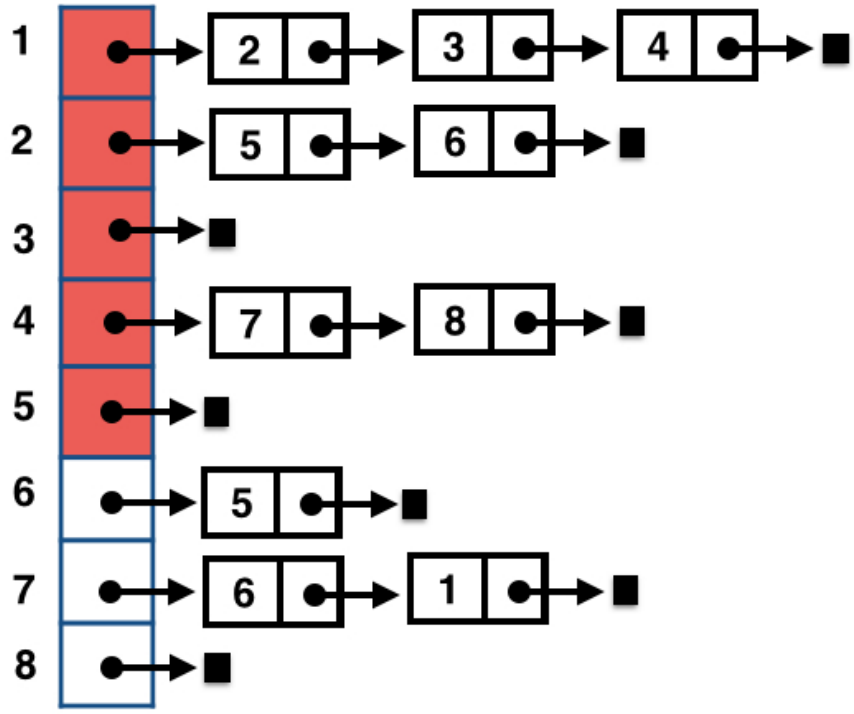
Console: 1 2 3 4



Q

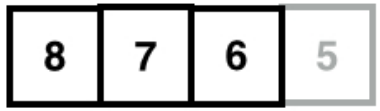
Graph Traversal: Breadth-first





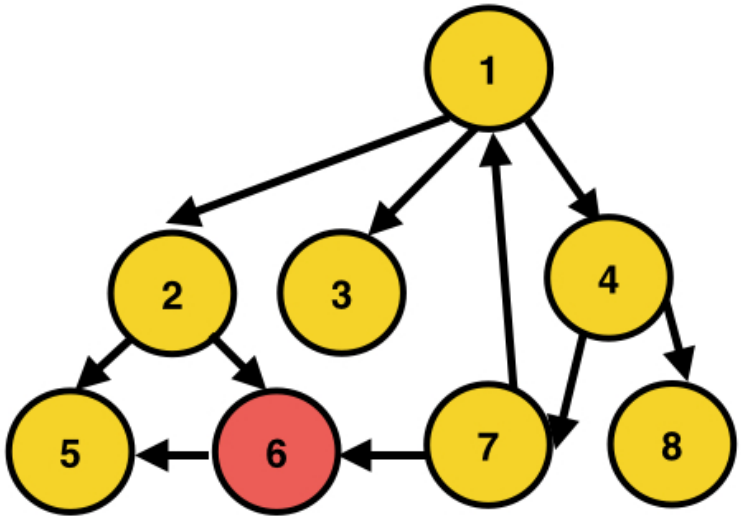
Dequeue a node from queue Q, print its content to the console,
mark the node as visited and enqueue the dequeued node's adjacent nodes into Q

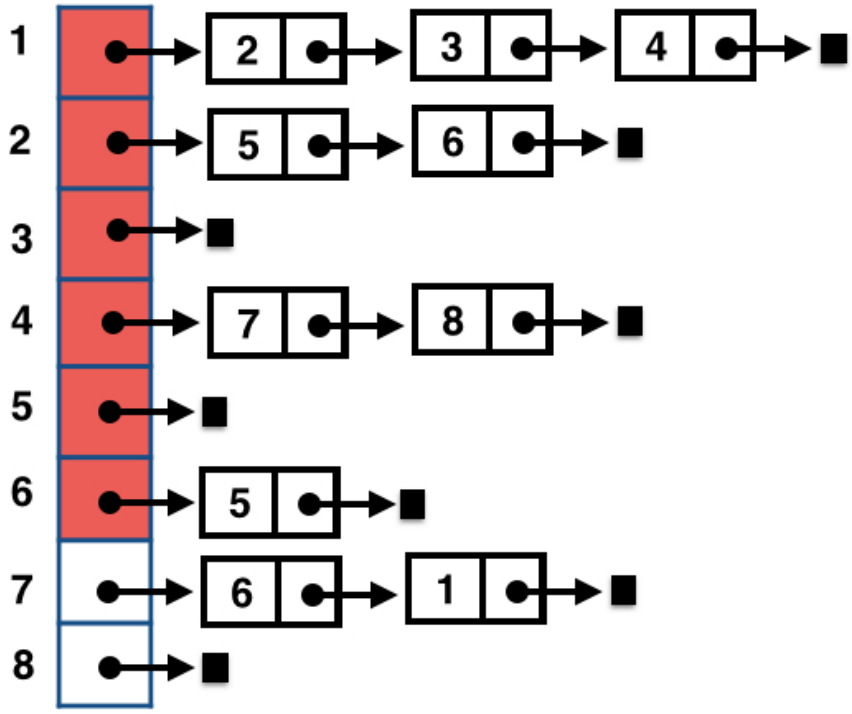
Console: 1 2 3 4 5



Q

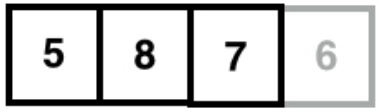
Graph Traversal: Breadth-first





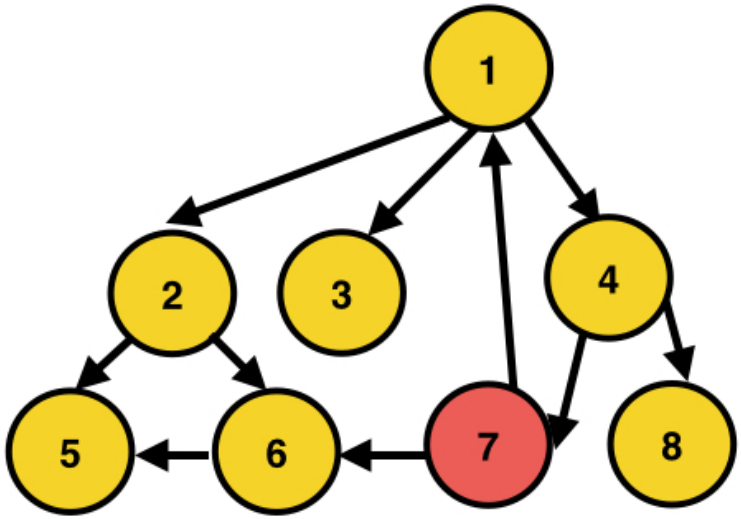
Dequeue a node from queue Q, print its content to the console,
mark the node as visited and enqueue the dequeued node's adjacent nodes into Q

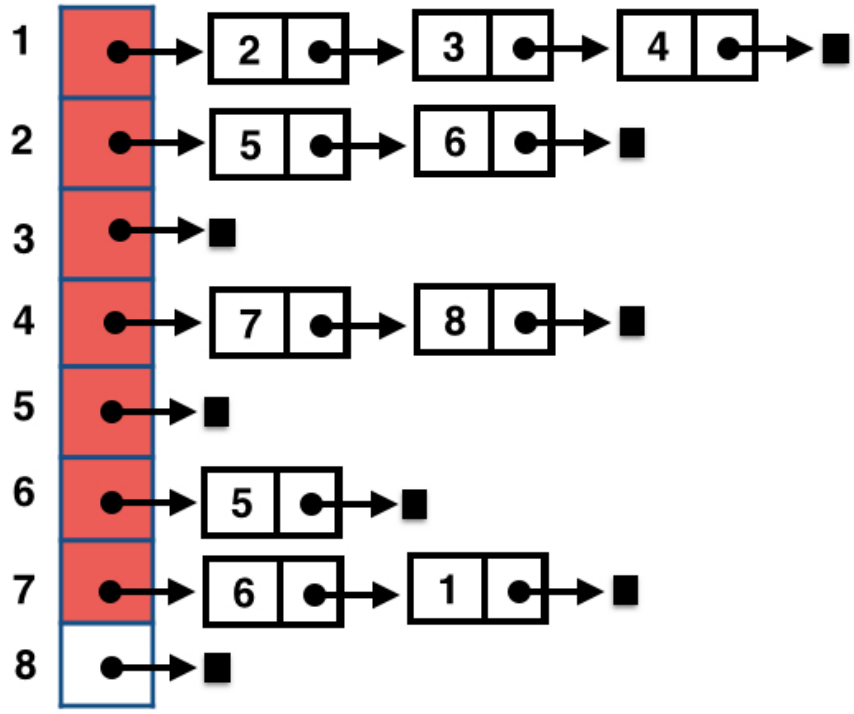
Console: 1 2 3 4 5 6



Q

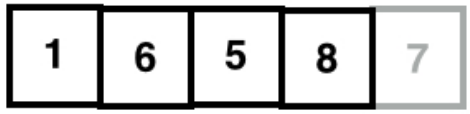
Graph Traversal: Breadth-first





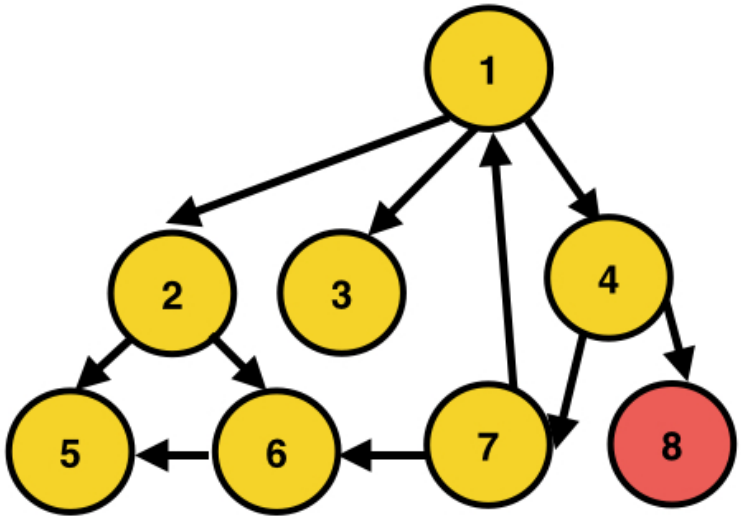
Dequeue a node from queue Q, print its content to the console,
mark the node as visited and enqueue the dequeued node's adjacent nodes into Q

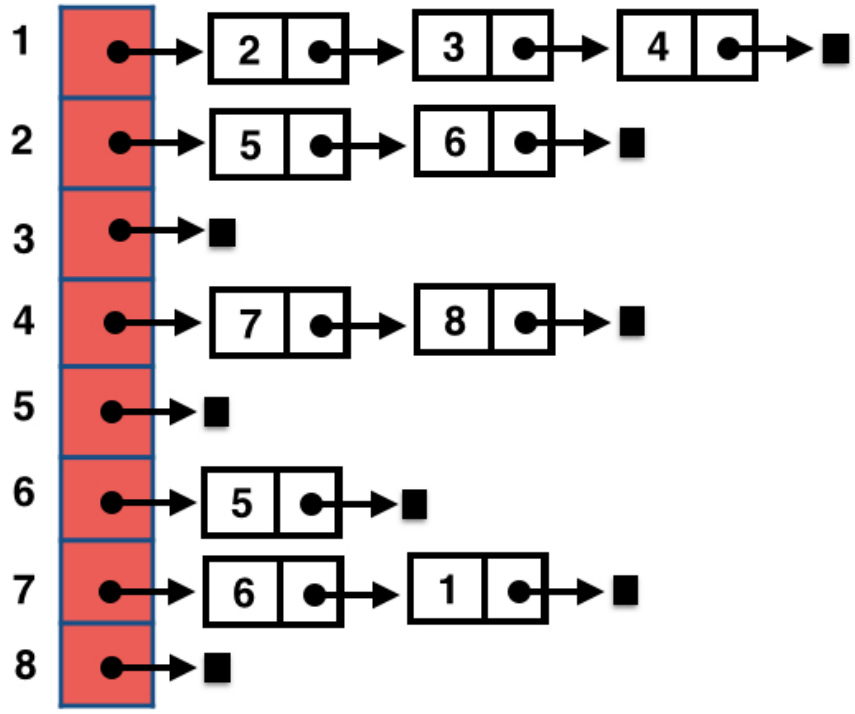
Console: 1 2 3 4 5 6 7



Q

Graph Traversal: Breadth-first

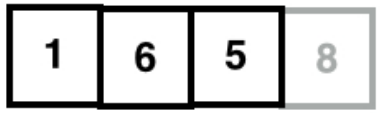




Dequeue a node from queue Q, print its content to the console,
mark the node as visited and enqueue the dequeued node's adjacent nodes into Q

All remaining nodes in the queue have been visited already!

Console: 1 2 3 4 5 6 7 8



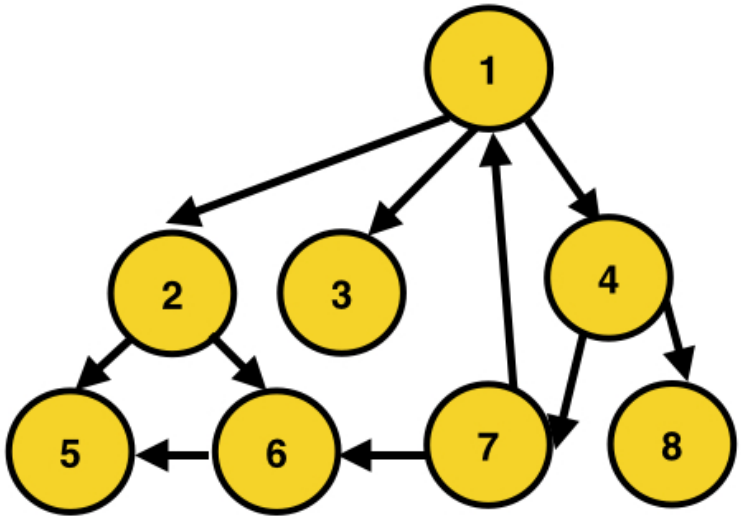
Q

Graph Traversal: Breadth-first

```
void breadthFirstTraversal(struct listNode * n, struct adjList * aL){

    struct Queue * q;
    struct listNode * l, * c;
    int i;
    initQueue(&q);
    enqueue(q,n);
    for (i=0; i<NODES; i++) {
        aL->visitedList[i]=0;
    }
    while(isQueueEmpty(q)!=1){
        dequeue(q,&l);
        if (aL->visitedList[l->vertexLabel-1]==0){
            printf("%d ",l->vertexLabel);
        }
        aL->visitedList[l->vertexLabel-1]=1;
        c=aL->ptrArray[l->vertexLabel-1];
        while (c!=NULL) {
            if (aL->visitedList[c->vertexLabel-1]==0) {
                enqueue(q,c);
            }
            c=c->next;
        }
    }
    free(q);
}
```

Depth-First Traversal

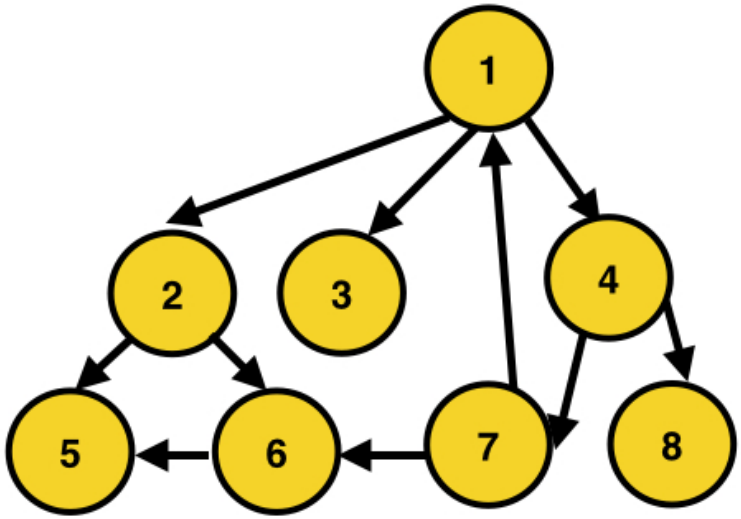
Graph Traversal: Depth-first

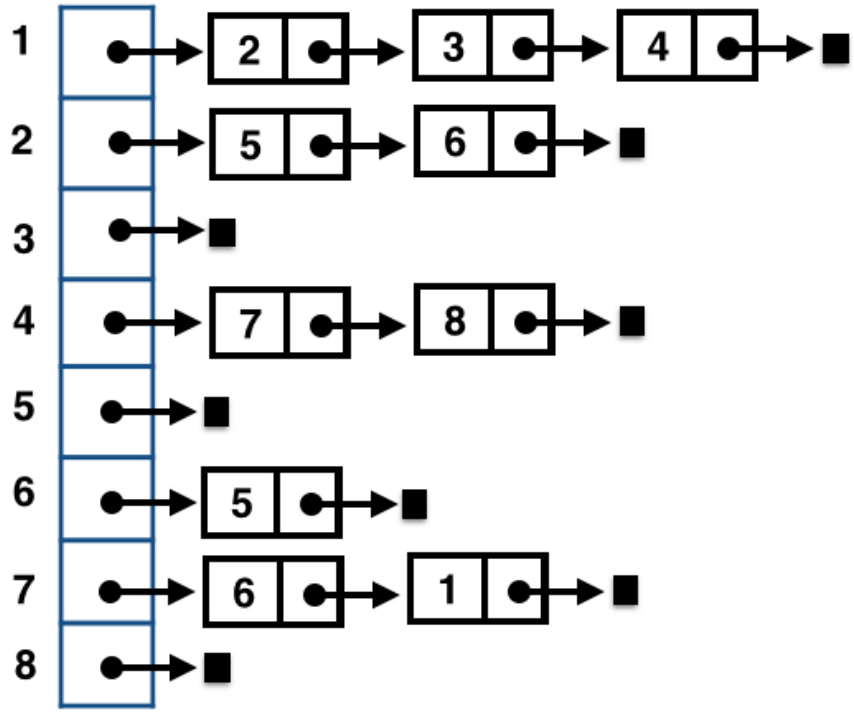


Depth-First Traversal: 1 2 5 6 3 4 7 8

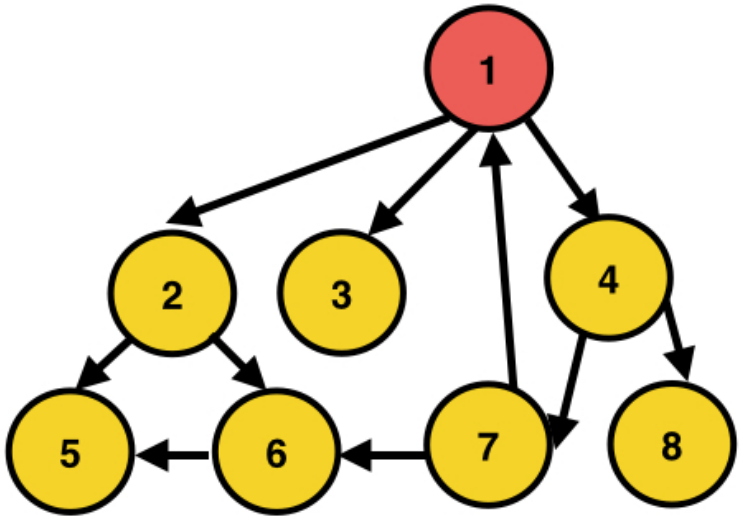
Can perform depth-first traversal by using
two stacks and marking nodes that are visited

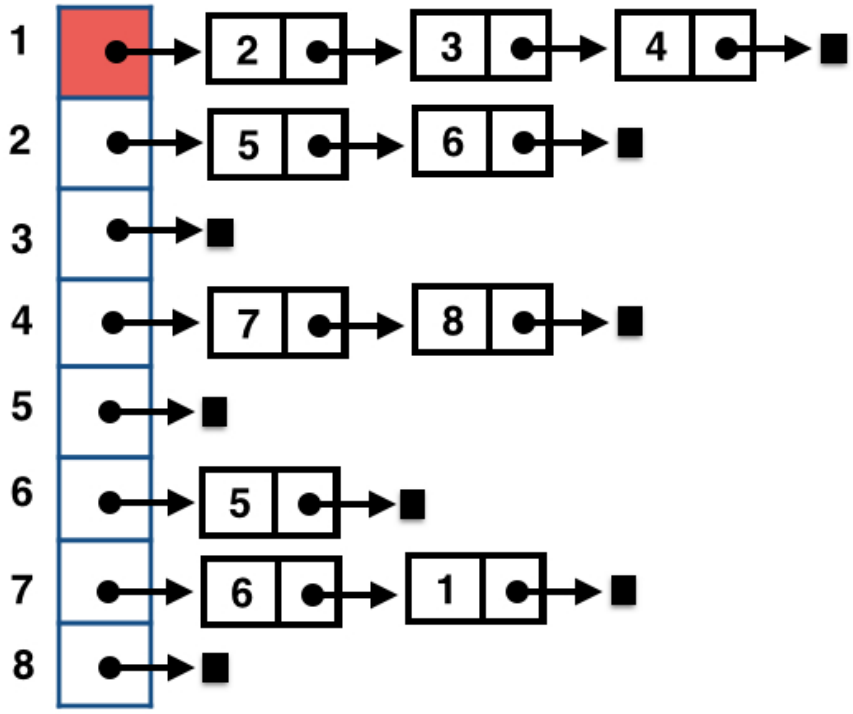
Graph Traversal: Depth-first





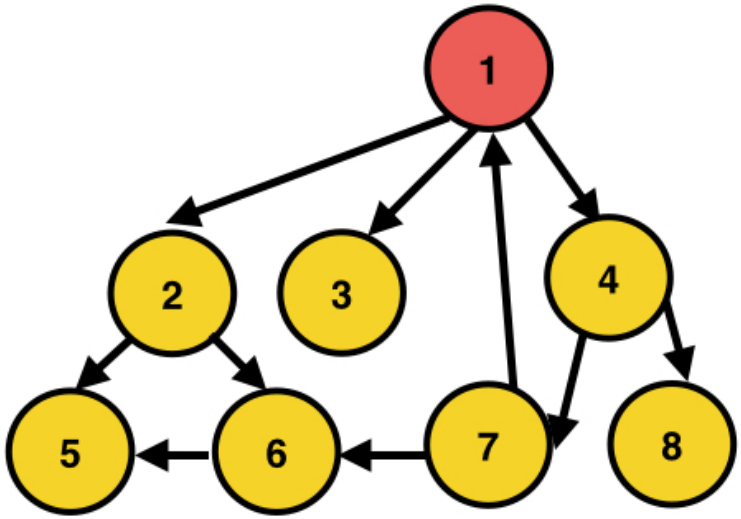
Graph Traversal: Depth-first

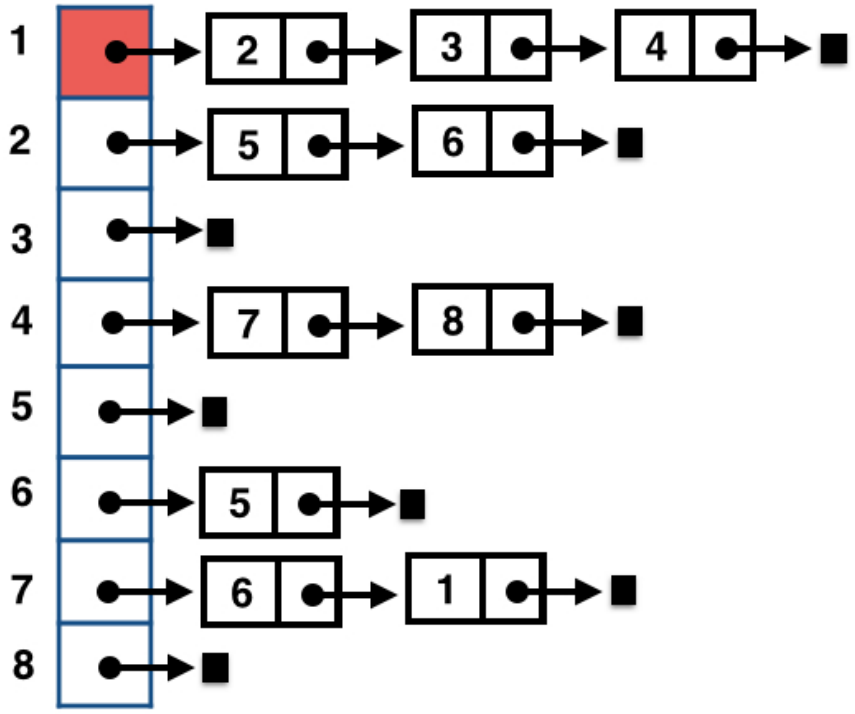




Mark nodes as these are visited. Push the starting node into S

Graph Traversal: Depth-first

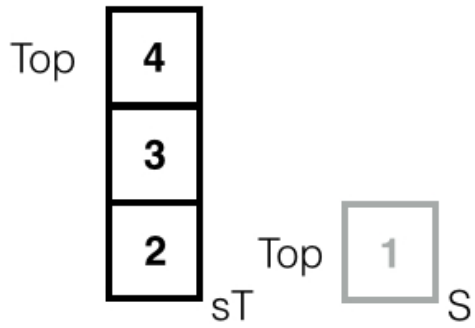




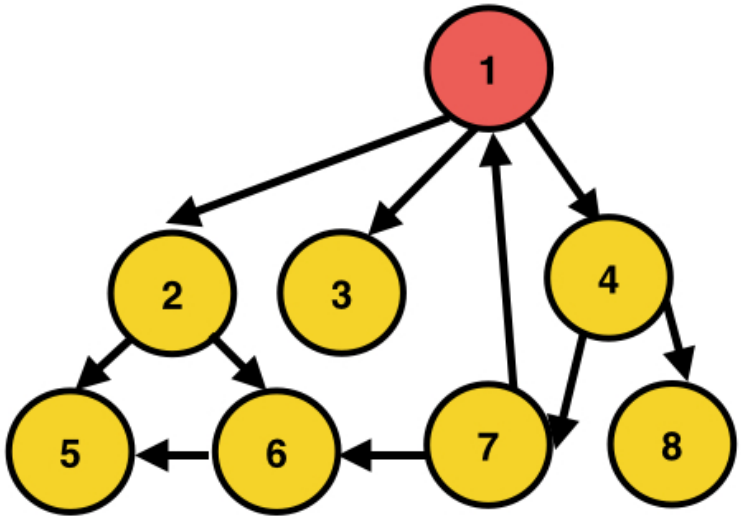
Pop a node from the stack s and push all the nodes adjacent to the popped node into stack sT

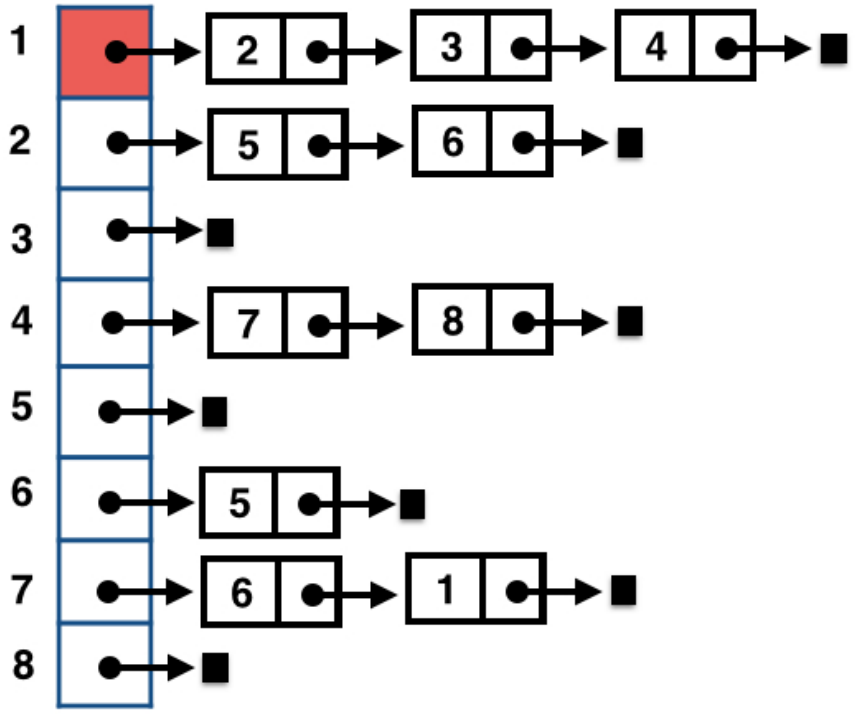
Print out the value of the popped node

Console: 1



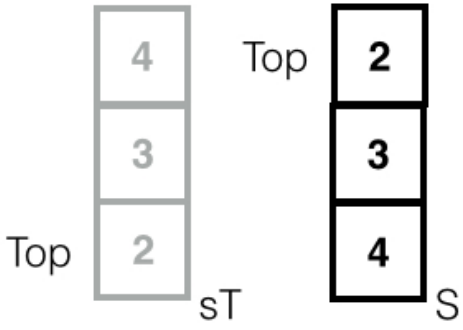
Graph Traversal: Depth-first



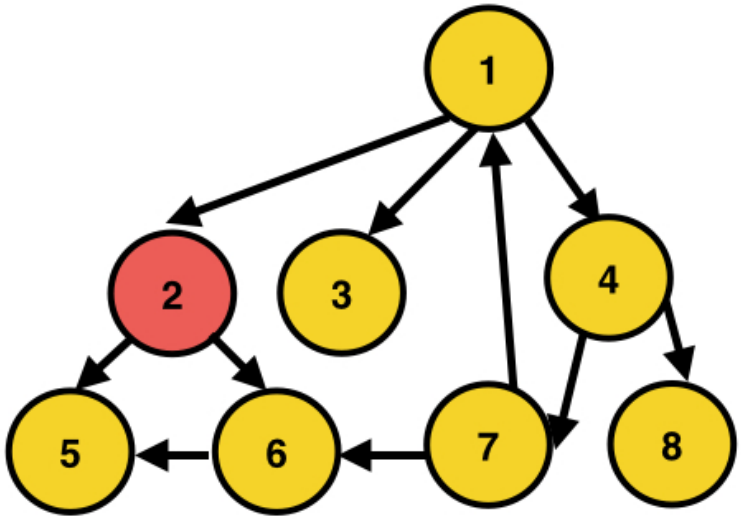


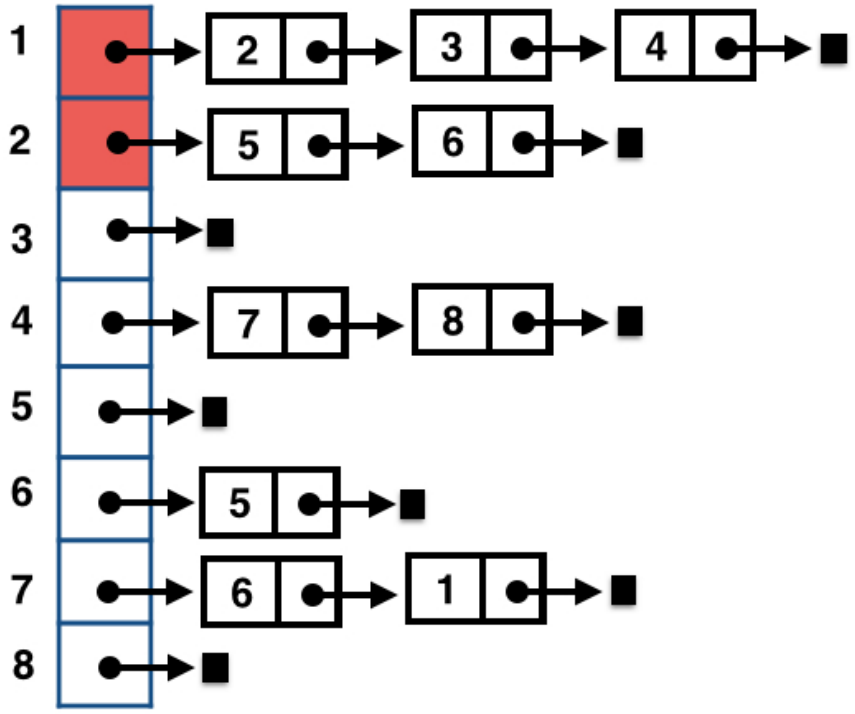
Push all nodes in sT into S

Console: 1



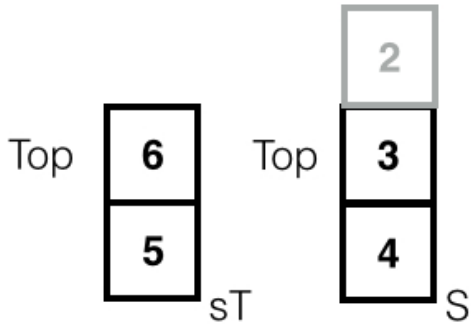
Graph Traversal: Depth-first



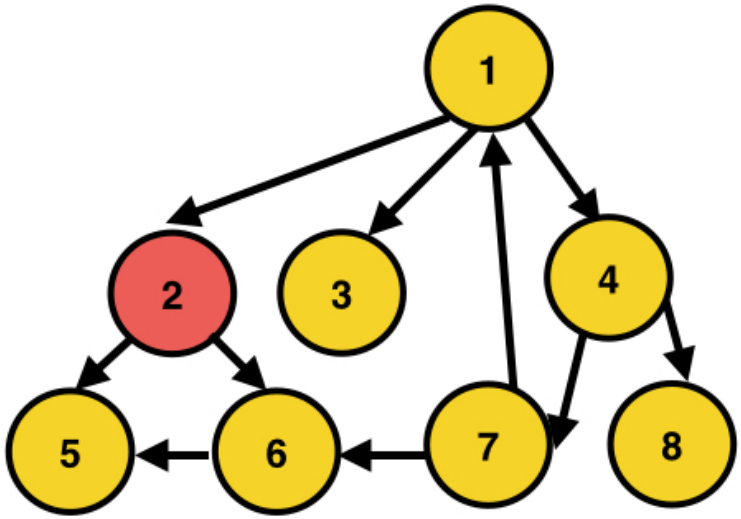


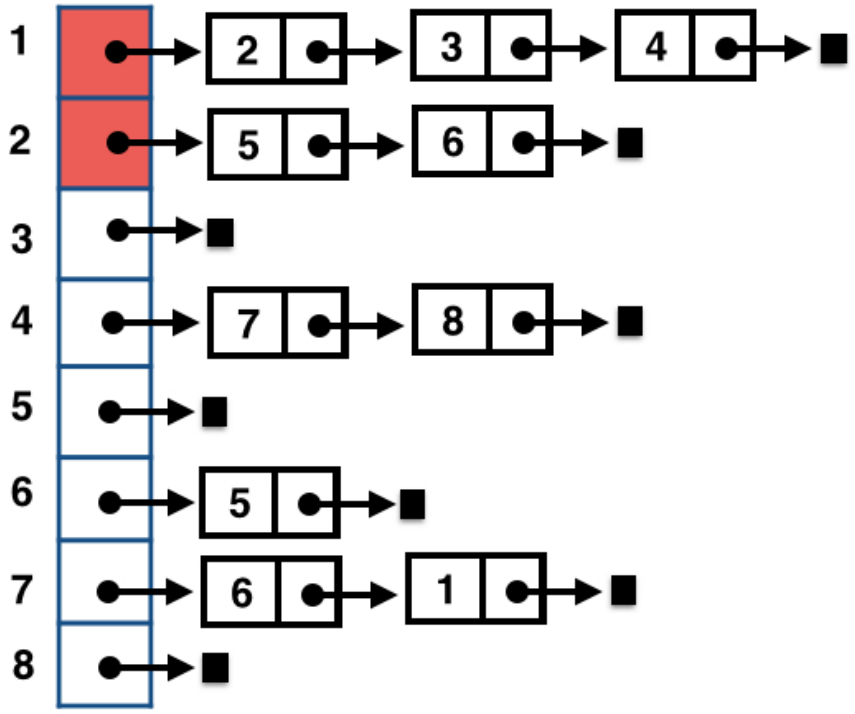
Pop a node from S, print its content to the console, push all adjacent nodes to the popped node into sT and mark the popped node as visited

Console: 1 2



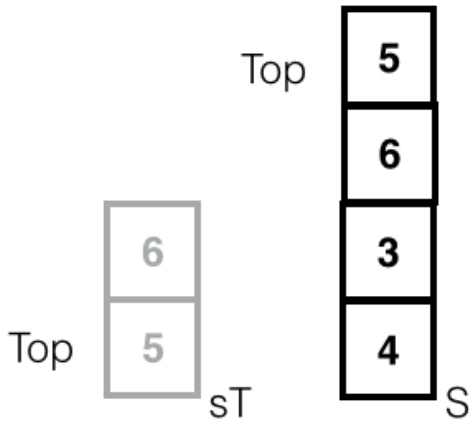
Graph Traversal: Depth-first



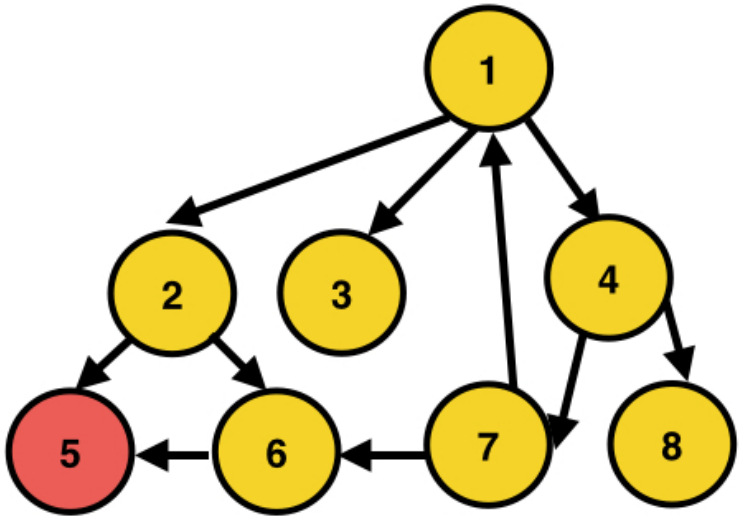


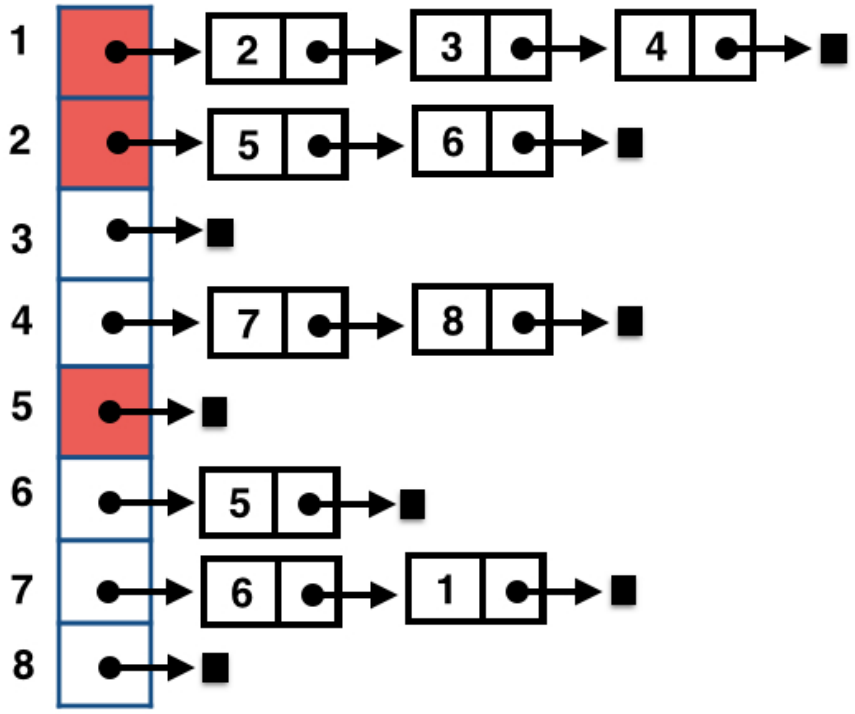
Push all the elements in sT into S

Console: 1 2



Graph Traversal: Depth-first

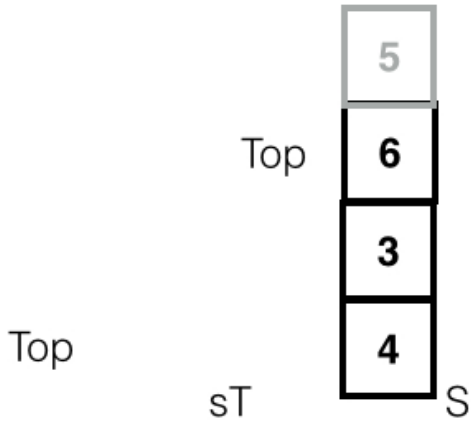




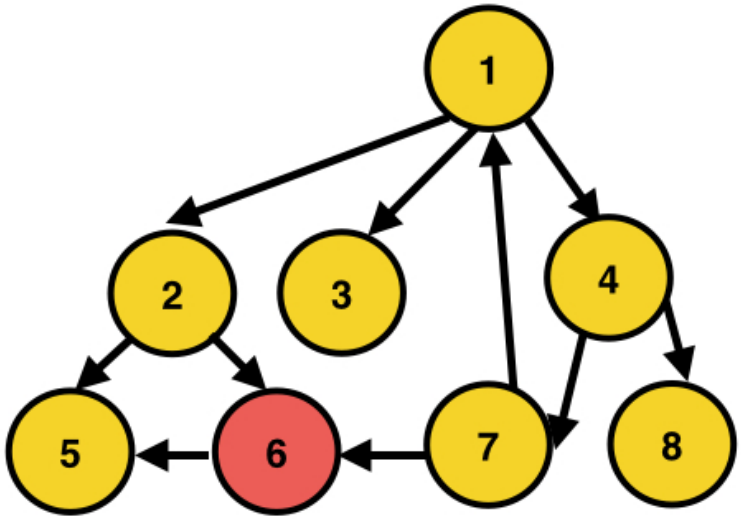
Pop a node from S, print its content to the console and mark the popped node as visited

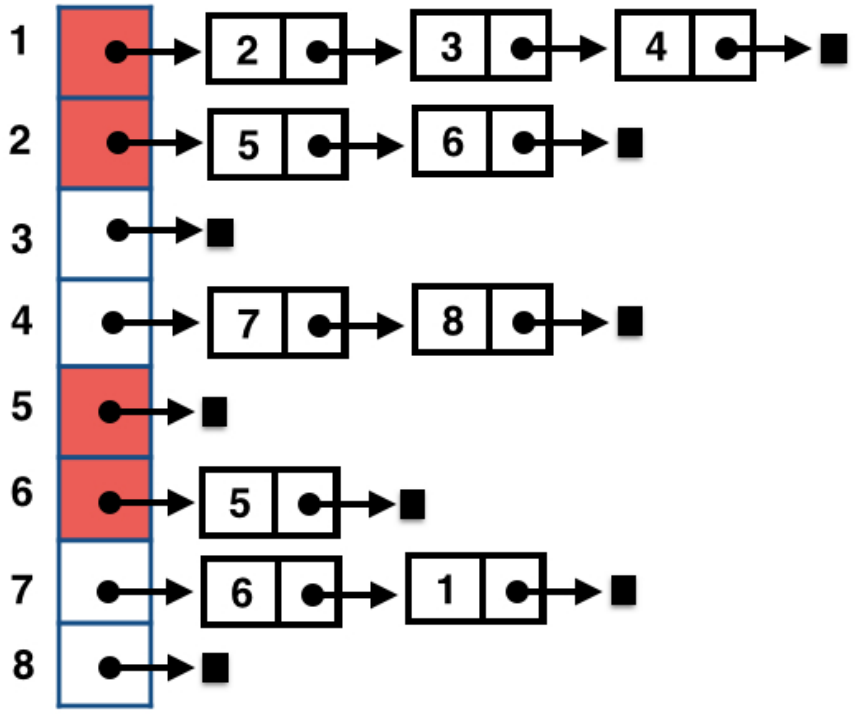
Cannot push all adjacent nodes to the popped node into sT as node 5 has no adjacency list

Console: 1 2 5



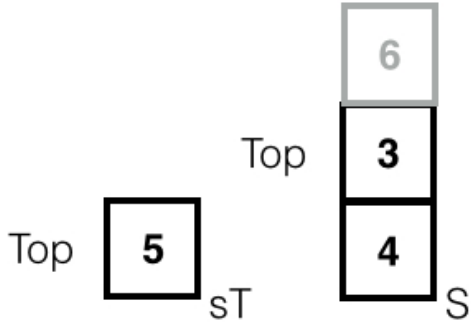
Graph Traversal: Depth-first



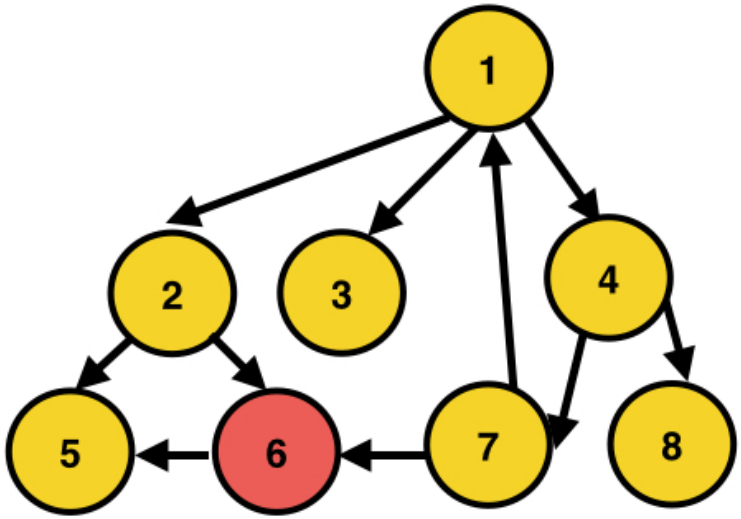


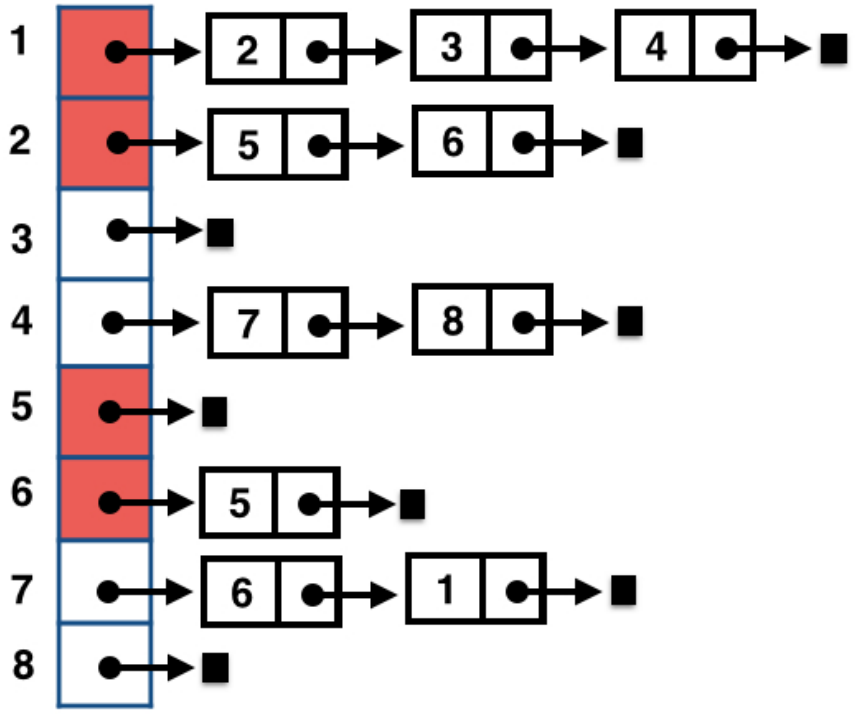
Pop a node from S, print its content to the console and mark the popped node as visited. Push all adjacent nodes to the popped node into sT.

Console: 1 2 5 6



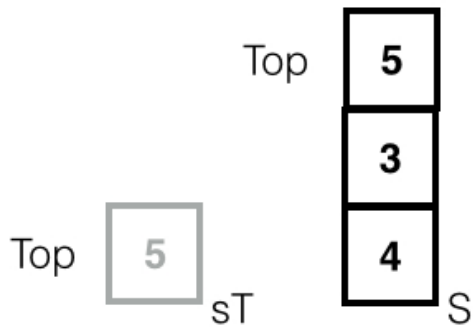
Graph Traversal: Depth-first



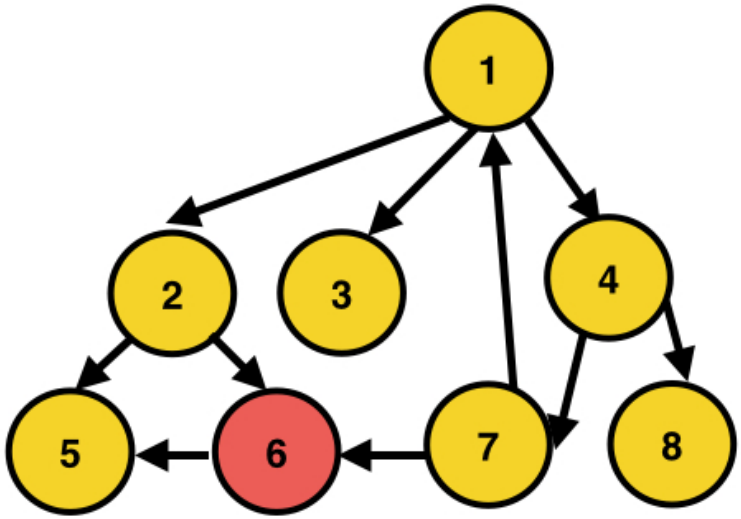


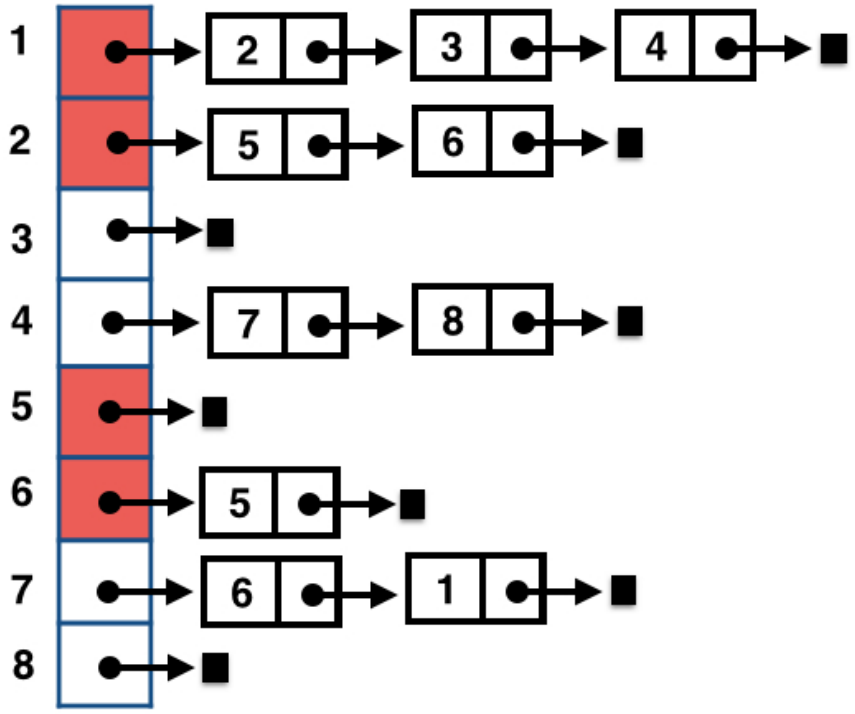
Push all the elements in sT into S

Console: 1 2 5 6



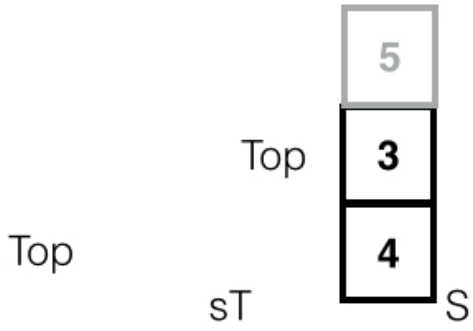
Graph Traversal: Depth-first



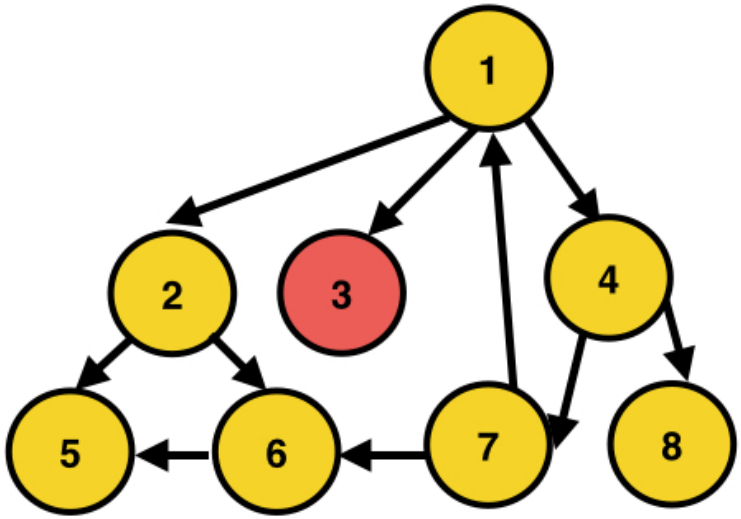


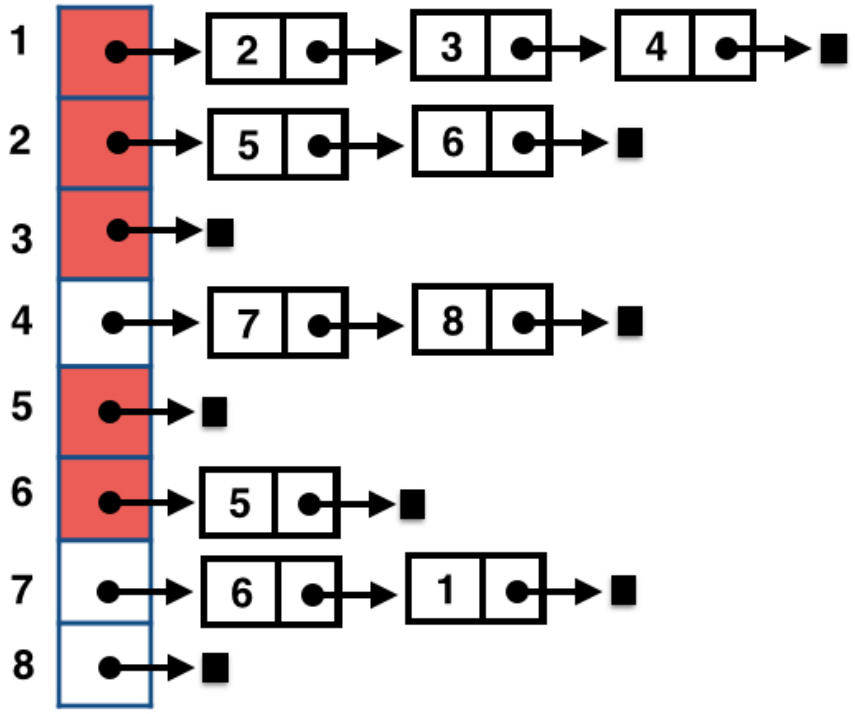
Pop node from S. Since node 5 is already visited, it is not printed

Console: 1 2 5 6



Graph Traversal: Depth-first

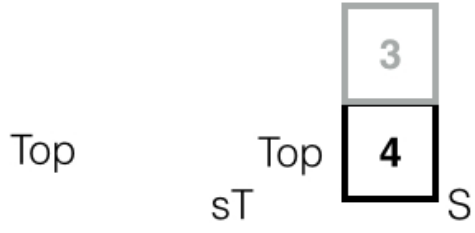




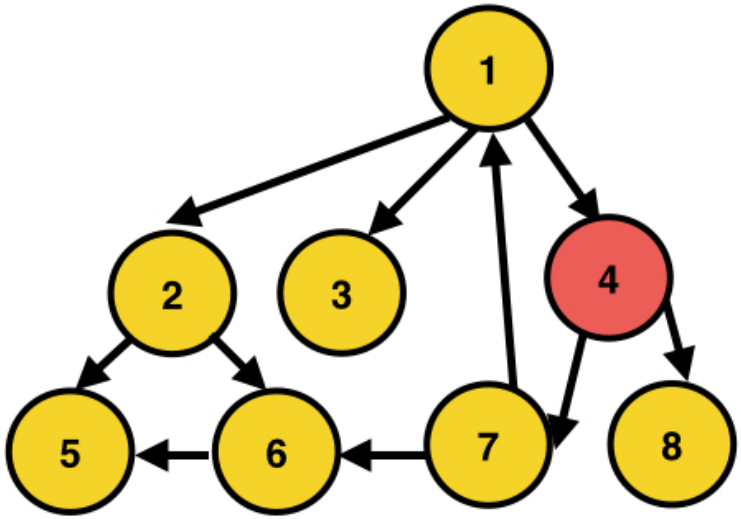
Pop a node from S, print its content to the console and mark the popped node as visited

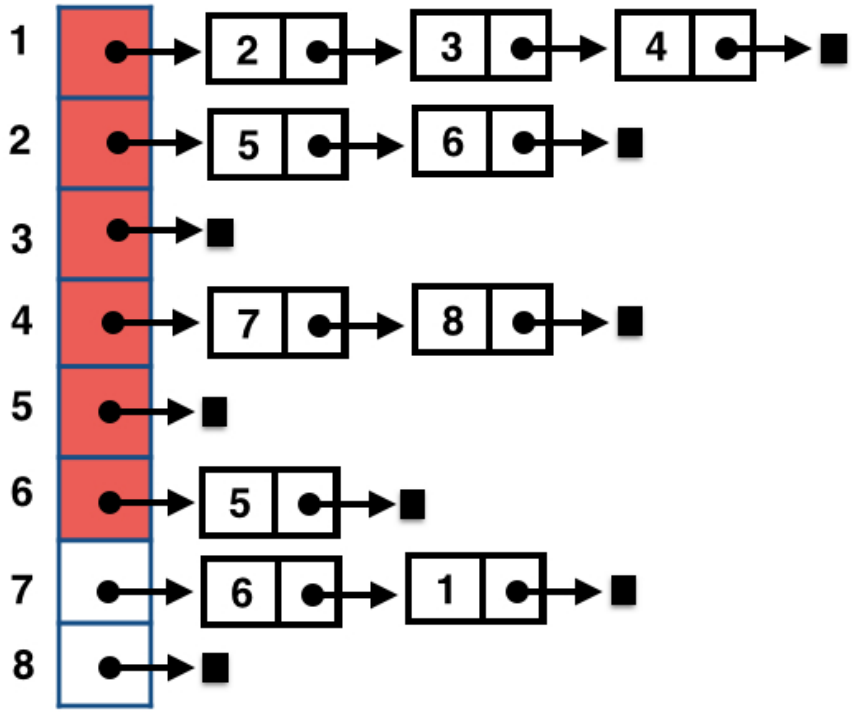
Cannot push all adjacent nodes to the popped node into sT as node 3 has no adjacency list

Console: 1 2 5 6 3



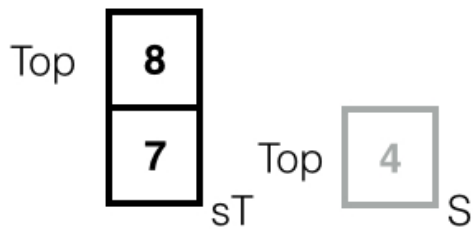
Graph Traversal: Depth-first



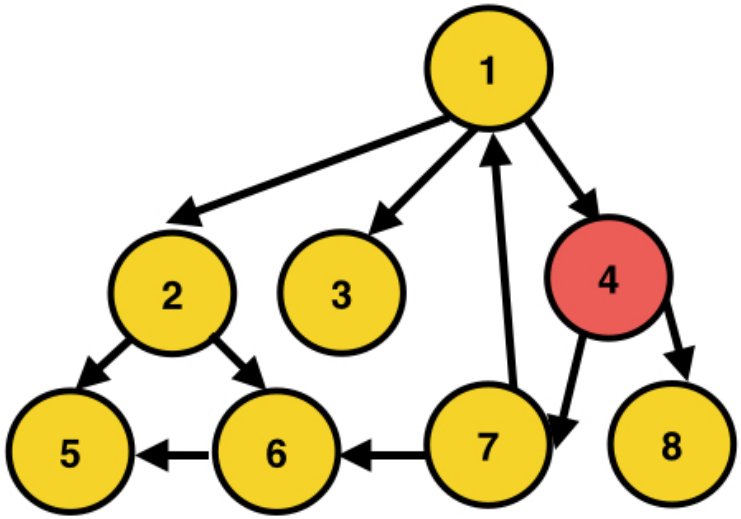


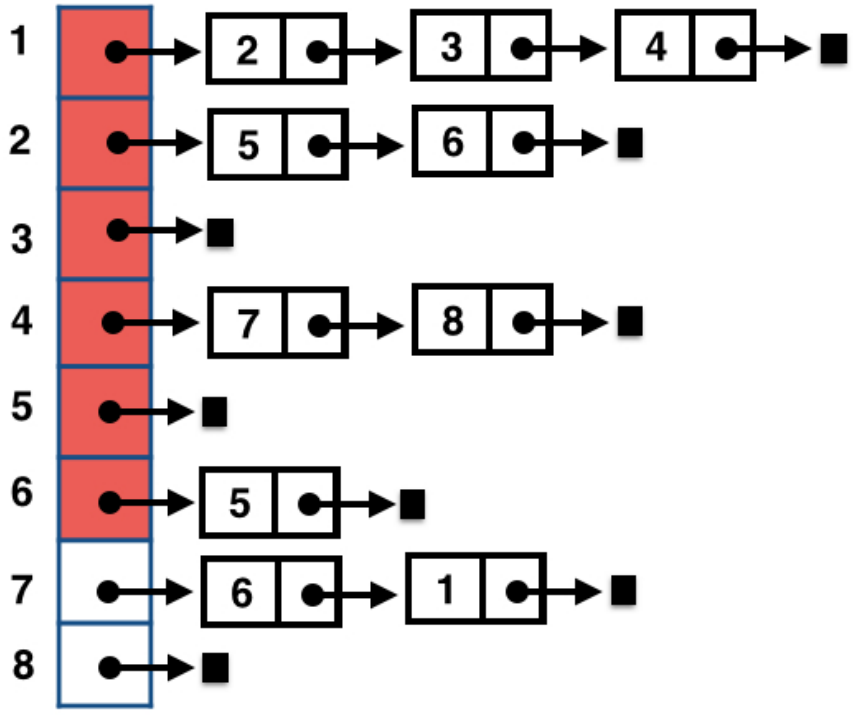
Pop a node from S, print its content to the console, push all adjacent nodes to the popped node into sT and mark the popped node as visited

Console: 1 2 5 6 3 4



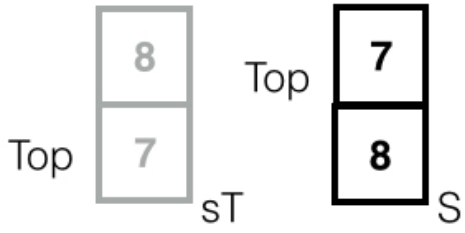
Graph Traversal: Depth-first



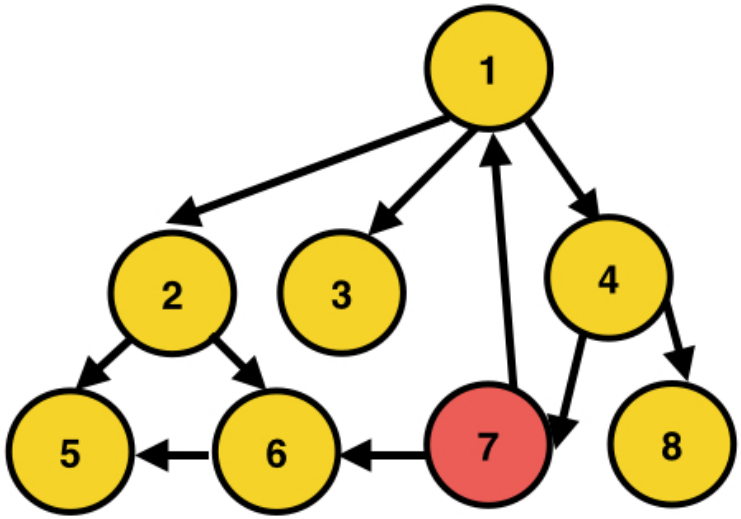


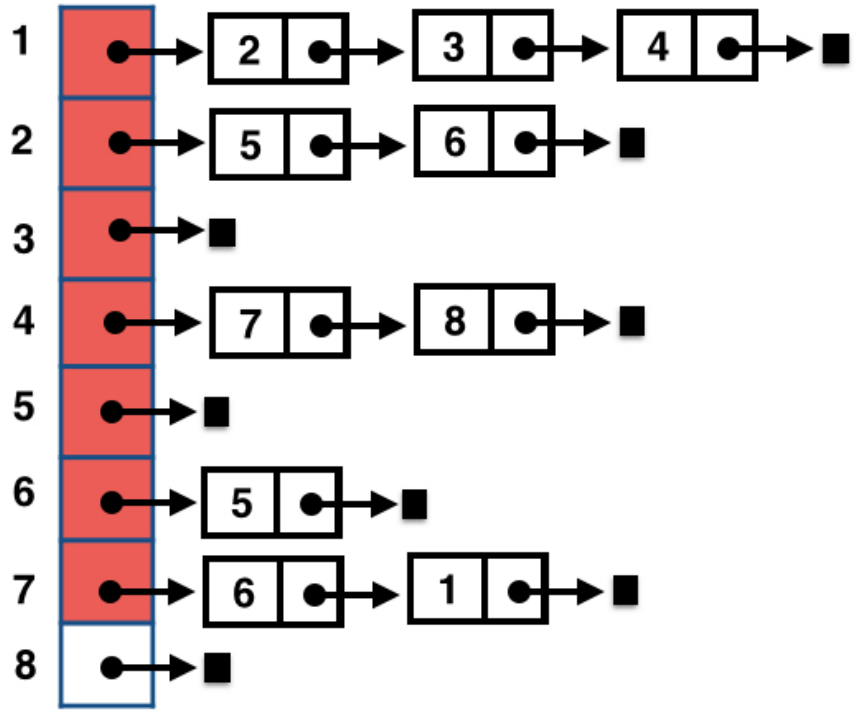
Push all elements in sT into S

Console: 1 2 5 6 3 4



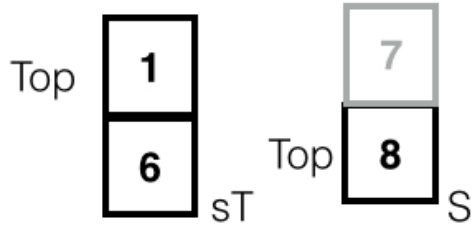
Graph Traversal: Depth-first



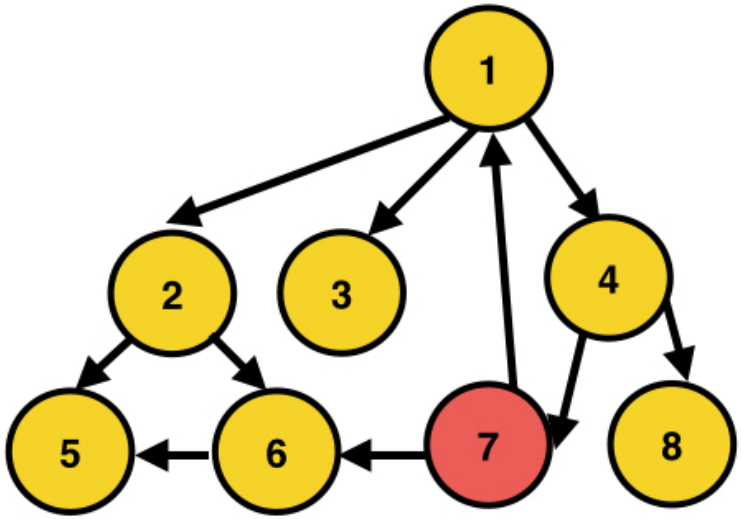


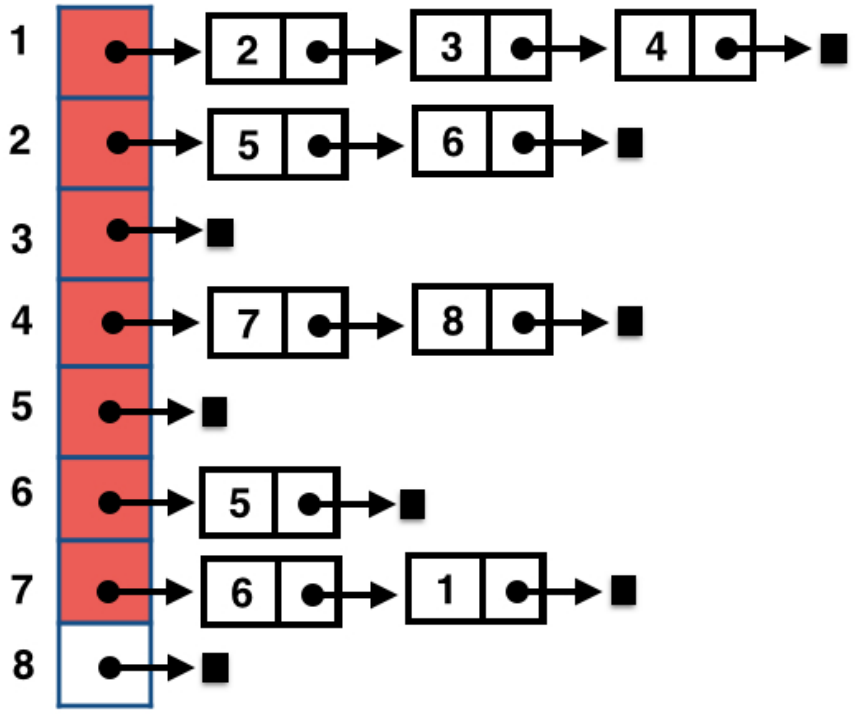
Pop a node from S, print its content to the console and mark the popped node as visited. Push all adjacent nodes to the popped node into sT.

Console: 1 2 5 6 3 4 7



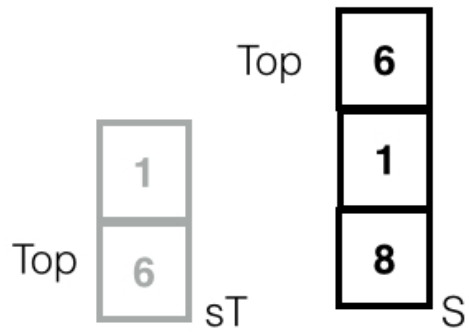
Graph Traversal: Depth-first



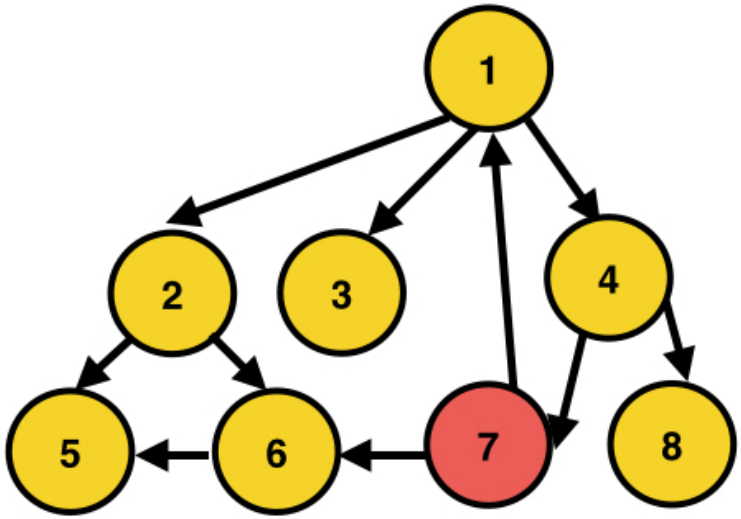


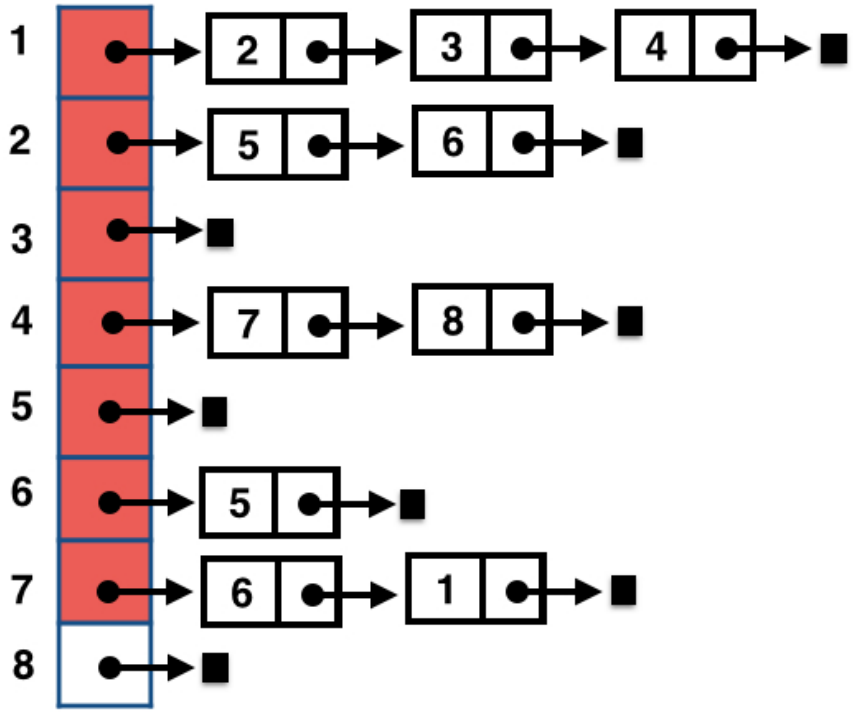
Push all elements in sT into S

Console: 1 2 5 6 3 4 7



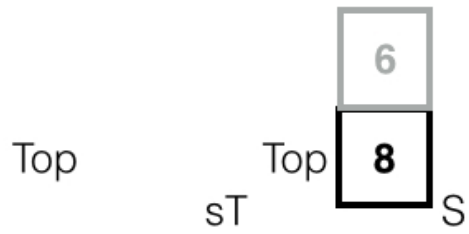
Graph Traversal: Depth-first



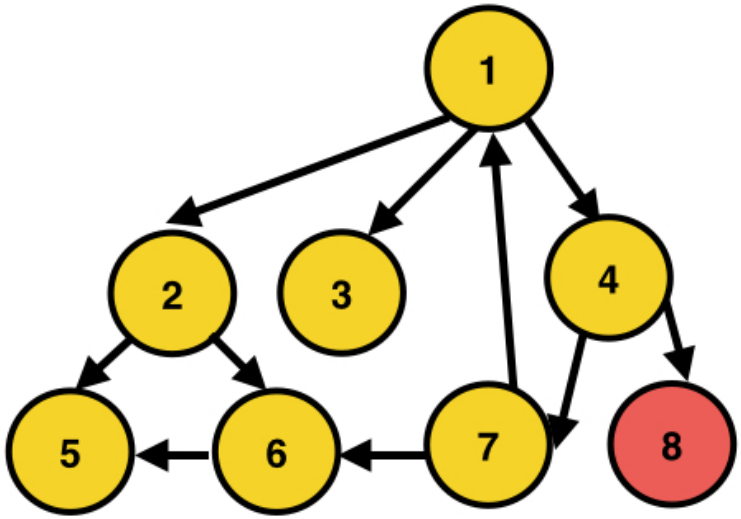


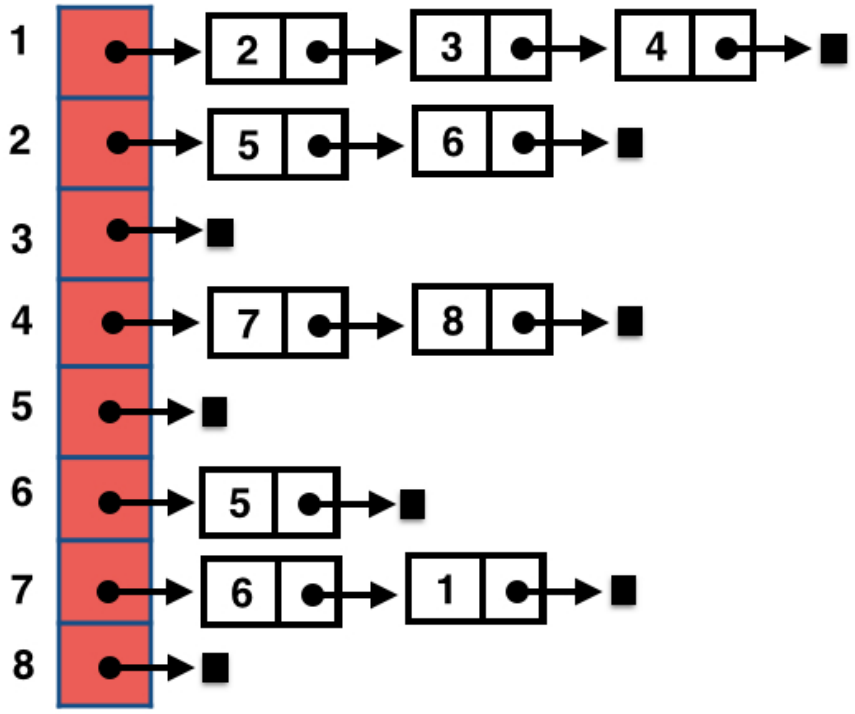
Pop S. Since node 6 is already visited, its value is not printed

Console: 1 2 5 6 3 4 7



Graph Traversal: Depth-first





Pop a node from S, print its content to the console and mark the popped node as visited

Cannot push all adjacent nodes to the popped node into sR as node 8 has no adjacency list

Console: 1 2 5 6 3 4 7 8

Graph Traversal: Depth-first

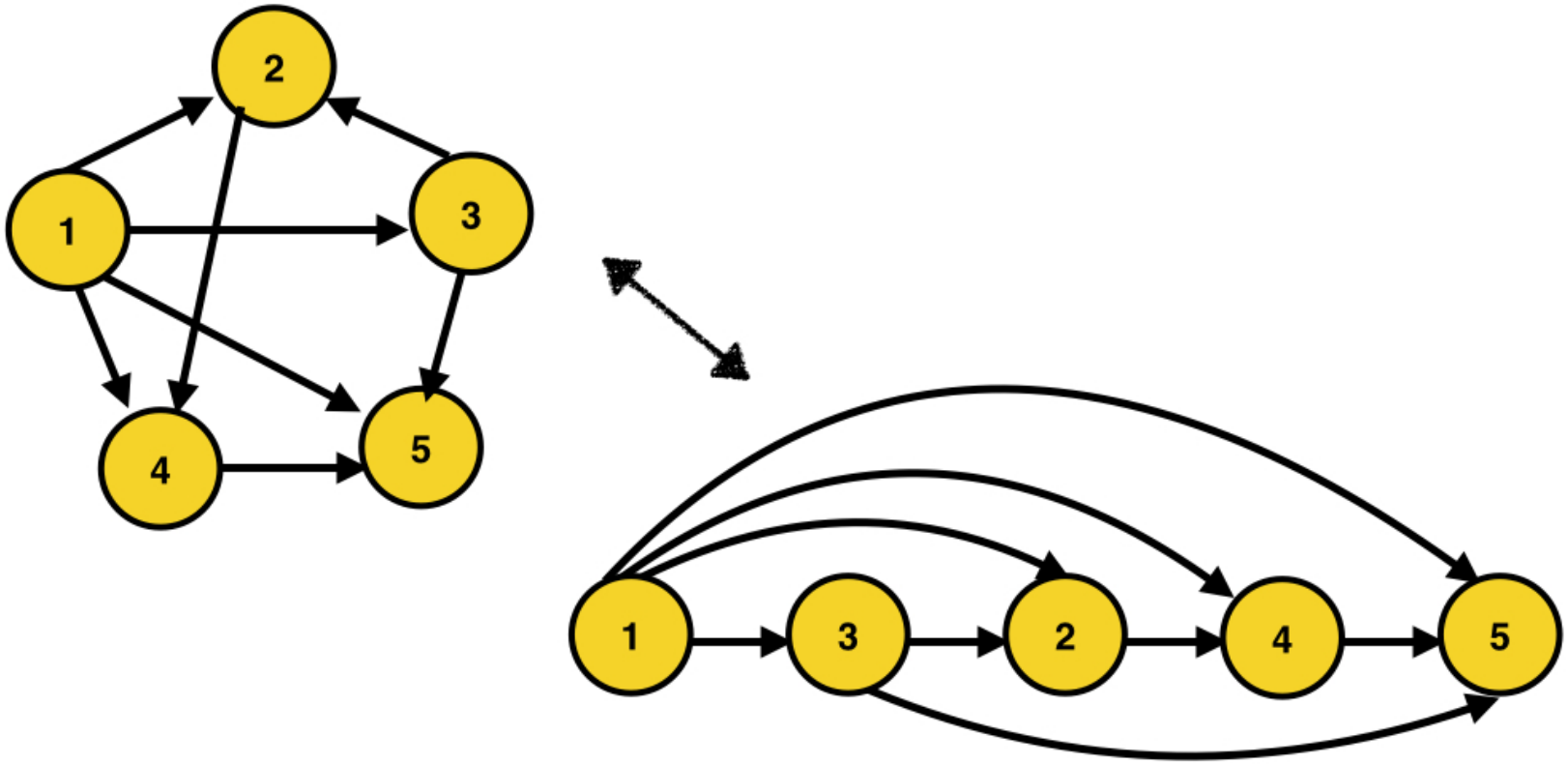
```
void depthFirstTraversal(struct listNode * n, struct adjList * aL){

    struct Stack * s, * sTemp;
    struct listNode * l, * c;
    int i;
    initStack(&s);
    initStack(&sTemp);
    push(s,n);
    for (i=0; i<NODES; i++) {
        aL->visitedList[i]=0;
    }
    while(isEmpty(s)!=1){
        pop(s,&l);
        if (aL->visitedList[l->vertexLabel-1]==0){
            printf("%d ",l->vertexLabel);
        }
        aL->visitedList[l->vertexLabel-1]=1;
        c=aL->ptrArray[l->vertexLabel-1];
        while (c!=NULL) {
            if (aL->visitedList[c->vertexLabel-1]==0) {
                push(sTemp,c);
            }
            c=c->next;
        }
        while (isEmpty(sTemp)!=1) {
            pop(sTemp,&l);
            push(s, l);
        }
    }
    free(s);
}
```

Table of Contents

Topological Ordering in a Graph

- Suppose that there are a set of events that take place in a process
- Some events maybe **contingent** on the completion of other events
- One example that you maybe dealing with currently is course pre-requisites
- Courses can be nodes/vertices and directed edges (v_i, v_j) indicate that v_i is a pre-requisite of v_j
- You may be interested in the order in which you can take these required courses throughout your degree
- This notion of **order** in the graph is referred to as **topological ordering**

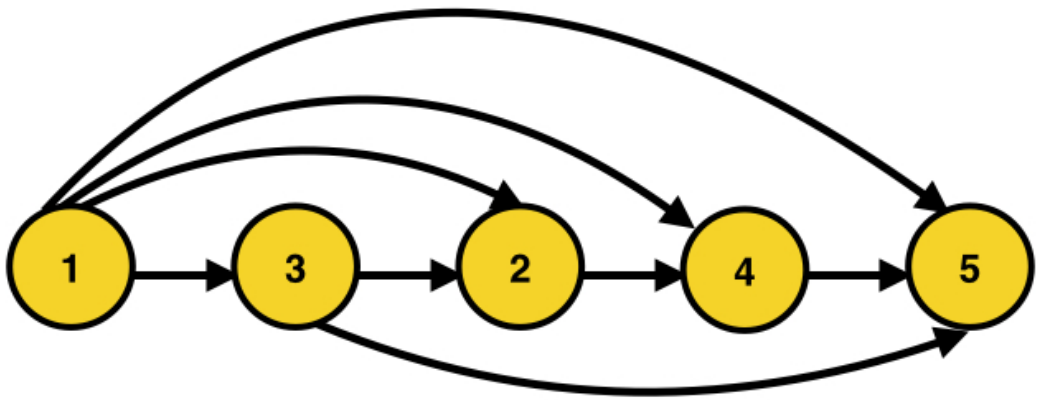
Example of a Pre-requisite Graph



DAGs

Will you be able to find the topological ordering of any graph?

- **No**, there are some **restrictions** on the graph properties
- The graph must be **directed** and **acyclic** (i.e. referred to as DAG *Directed Acyclic Graph*)



Topological Ordering Algorithm

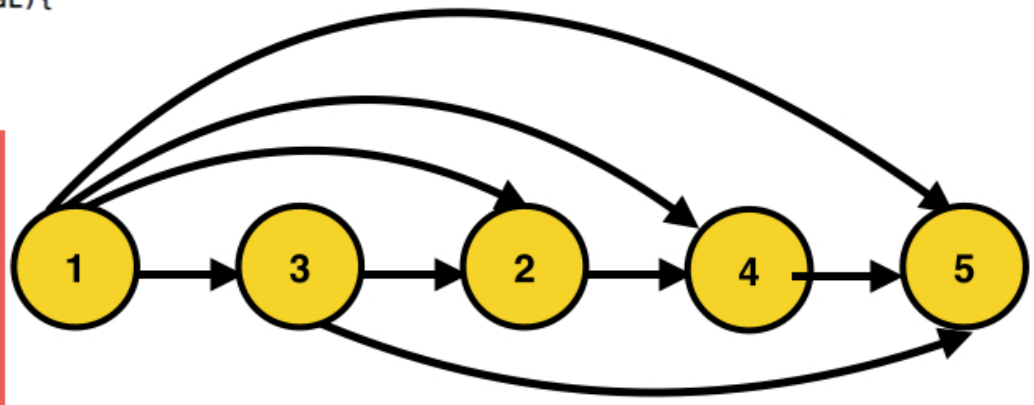
- What courses can you take **without** any pre-requisites?
  - The node v_x with an in-degree of 0
- The **in-degree** of a node indicates the number of pre-requisites required before being able to take that course
- After taking the first pre-requisite of a course v_x, the number of pre-requisites left before v_x can be taken is the $inDegree(v_x) - 1$
- Suppose that the number of pre-requisites left for v_x is recorded in a variable p_x
- Initializing $p_x = inDegree(v_x)$ $v_x \in V$ and every time a course v_y is taken, decrement p_x variable of all $v_x \in Succ(v_y)$ by 1
- The next course that can be taken is the course v_x with $p_x = 0$
- Repeat above until all courses in the graph are taken

Implementation of Topological Ordering of DAGs

```
void topologicalOrdering(struct adjList * aL){
    int inDegree[NODES]={0};
    struct Queue * q;
    struct listNode * lN, * cN;
    int i;
    initQueue(&q);

    markVerticesAsUnvisited(aL);

    for (i=0; i<NODES; i++) {
        lN=aL->ptrArray[i];
        while (lN!=NULL) {
            inDegree[lN->vertexLabel-1]++;
            lN=lN->next;
        }
    }

    for (i=0; i<NODES; i++) {
        if (inDegree[i]==0) {
            enqueue(q, newNode(i+1));
        }
    }

    while (isQueueEmpty(q)!=1) {
        dequeue(q, &lN);
        printf("%d ", lN->vertexLabel);
        aL->visitedList[lN->vertexLabel-1]=1;
        cN=aL->ptrArray[lN->vertexLabel-1];
        free(lN);
        while (cN!=NULL) {
            inDegree[cN->vertexLabel-1]--;
            cN=cN->next;
        }
        for (i=0; i<NODES; i++) {
            if (inDegree[i]==0 && aL->visitedList[i]!=1) {
                enqueue(q, newNode(i+1));
            }
        }
    }
    free(q);
}
```

Implementation of Topological Ordering of DAGs

```
void topologicalOrdering(struct adjList * aL){
    int inDegree[NODES]={0};
    struct Queue * q;
    struct listNode * lN, * cN;
    int i;
    initQueue(&q);

    markVerticesAsUnvisited(aL);

    for (i=0; i<NODES; i++) {
        lN=aL->ptrArray[i];
        while (lN!=NULL) {
            inDegree[lN->vertexLabel-1]++;
            lN=lN->next;
        }
    }

    for (i=0; i<NODES; i++) {
        if (inDegree[i]==0) {
            enqueue(q, newNode(i+1));
        }
    }

    while (isQueueEmpty(q)!=1) {
        dequeue(q, &lN);
        printf("%d ", lN->vertexLabel);
        aL->visitedList[lN->vertexLabel-1]=1;
        cN=aL->ptrArray[lN->vertexLabel-1];
        free(lN);
        while (cN!=NULL) {
            inDegree[cN->vertexLabel-1]--;
            cN=cN->next;
        }
        for (i=0; i<NODES; i++) {
            if (inDegree[i]==0 && aL->visitedList[i]!=1) {
                enqueue(q, newNode(i+1));
            }
        }
    }
    free(q);
}
```



inDegree	0	2	1	2	3
vertices	1	2	3	4	5

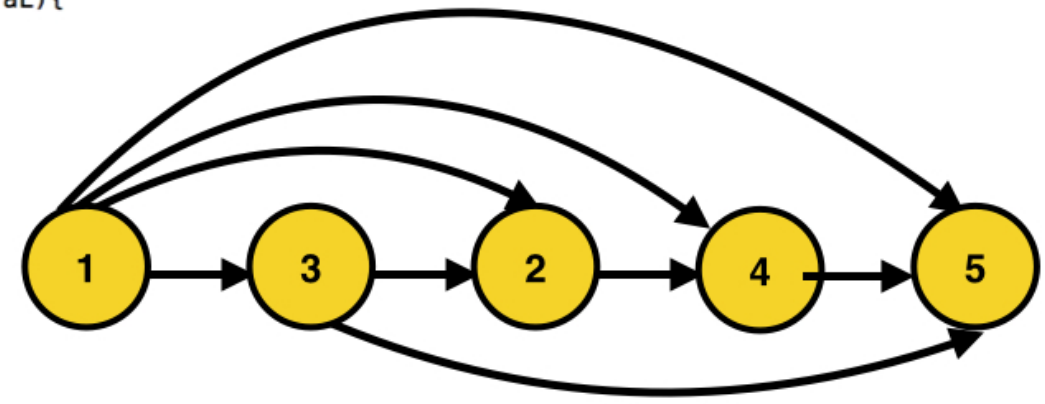
visitedList	0	0	0	0	0
vertices	1	2	3	4	5

Q

Console:

Implementation of Topological Ordering of DAGs

```
void topologicalOrdering(struct adjList * aL){
    int inDegree[NODES]={0};
    struct Queue * q;
    struct listNode * lN, * cN;
    int i;
    initQueue(&q);

    markVerticesAsUnvisited(aL);

    for (i=0; i<NODES; i++) {
        lN=aL->ptrArray[i];
        while (lN!=NULL) {
            inDegree[lN->vertexLabel-1]++;
            lN=lN->next;
        }
    }

    for (i=0; i<NODES; i++) {
        if (inDegree[i]==0) {
            enqueue(q, newNode(i+1));
        }
    }

    while (isQueueEmpty(q)!=1) {
        dequeue(q, &lN);
        printf("%d ", lN->vertexLabel);
        aL->visitedList[lN->vertexLabel-1]=1;
        cN=aL->ptrArray[lN->vertexLabel-1];
        free(lN);
        while (cN!=NULL) {
            inDegree[cN->vertexLabel-1]--;
            cN=cN->next;
        }
        for (i=0; i<NODES; i++) {
            if (inDegree[i]==0 && aL->visitedList[i]!=1) {
                enqueue(q, newNode(i+1));
            }
        }
    }
    free(q);
}
```



inDegree	0	2	1	2	3
vertices	1	2	3	4	5

visitedList	0	0	0	0	0
vertices	1	2	3	4	5

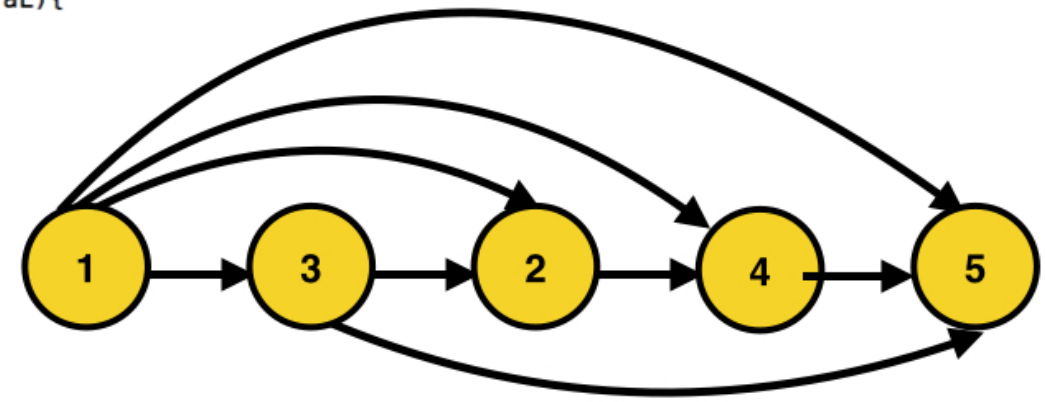
Console:

1

Q

Implementation of Topological Ordering of DAGs

```
void topologicalOrdering(struct adjList * aL){
    int inDegree[NODES]={0};
    struct Queue * q;
    struct listNode * lN, * cN;
    int i;
    initQueue(&q);

    markVerticesAsUnvisited(aL);

    for (i=0; i<NODES; i++) {
        lN=aL->ptrArray[i];
        while (lN!=NULL) {
            inDegree[lN->vertexLabel-1]++;
            lN=lN->next;
        }
    }

    for (i=0; i<NODES; i++) {
        if (inDegree[i]==0) {
            enqueue(q, newNode(i+1));
        }
    }

    while (isQueueEmpty(q)!=1) {
        dequeue(q, &lN);
        printf("%d ", lN->vertexLabel);
        aL->visitedList[lN->vertexLabel-1]=1;
        cN=aL->ptrArray[lN->vertexLabel-1];
        free(lN);
        while (cN!=NULL) {
            inDegree[cN->vertexLabel-1]--;
            cN=cN->next;
        }
        for (i=0; i<NODES; i++) {
            if (inDegree[i]==0 && aL->visitedList[i]!=1) {
                enqueue(q, newNode(i+1));
            }
        }
    }
    free(q);
}
```



inDegree	0	1	0	1	2
vertices	1	2	3	4	5

visitedList	1	0	0	0	0
vertices	1	2	3	4	5

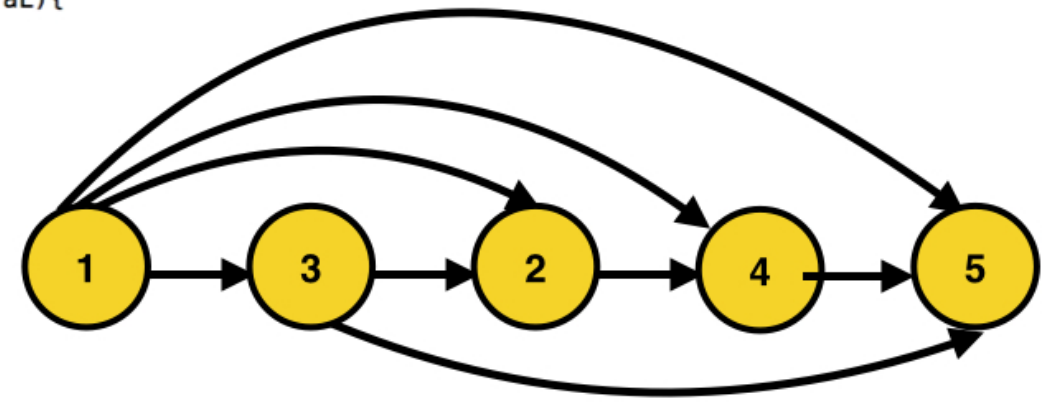
Console:
1

Q

3

Implementation of Topological Ordering of DAGs

```
void topologicalOrdering(struct adjList * aL){
    int inDegree[NODES]={0};
    struct Queue * q;
    struct listNode * lN, * cN;
    int i;
    initQueue(&q);

    markVerticesAsUnvisited(aL);

    for (i=0; i<NODES; i++) {
        lN=aL->ptrArray[i];
        while (lN!=NULL) {
            inDegree[lN->vertexLabel-1]++;
            lN=lN->next;
        }
    }

    for (i=0; i<NODES; i++) {
        if (inDegree[i]==0) {
            enqueue(q, newNode(i+1));
        }
    }

    while (isQueueEmpty(q)!=1) {
        dequeue(q, &lN);
        printf("%d ", lN->vertexLabel);
        aL->visitedList[lN->vertexLabel-1]=1;
        cN=aL->ptrArray[lN->vertexLabel-1];
        free(lN);
        while (cN!=NULL) {
            inDegree[cN->vertexLabel-1]--;
            cN=cN->next;
        }
        for (i=0; i<NODES; i++) {
            if (inDegree[i]==0 && aL->visitedList[i]!=1) {
                enqueue(q, newNode(i+1));
            }
        }
    }
    free(q);
}
```



inDegree	0	0	0	1	1
vertices	1	2	3	4	5

visitedList	1	0	1	0	0
vertices	1	2	3	4	5

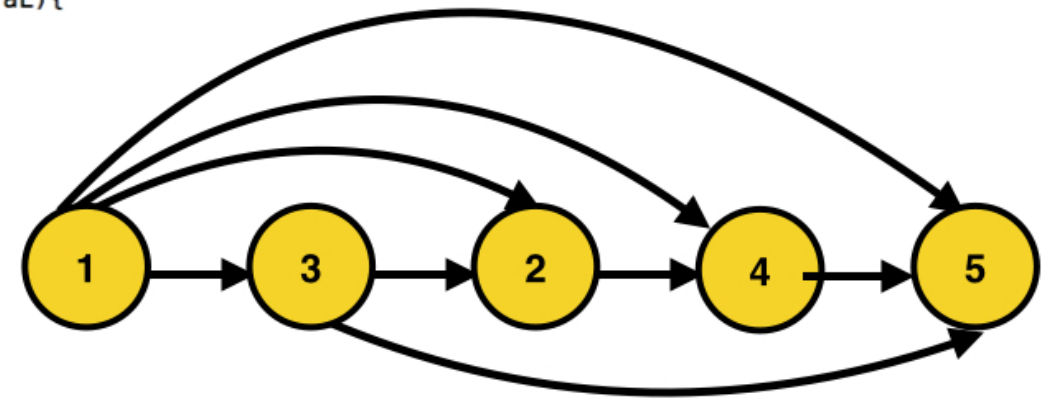
Console:
1 3

2

Q

Implementation of Topological Ordering of DAGs

```
void topologicalOrdering(struct adjList * aL){
    int inDegree[NODES]={0};
    struct Queue * q;
    struct listNode * lN, * cN;
    int i;
    initQueue(&q);

    markVerticesAsUnvisited(aL);

    for (i=0; i<NODES; i++) {
        lN=aL->ptrArray[i];
        while (lN!=NULL) {
            inDegree[lN->vertexLabel-1]++;
            lN=lN->next;
        }
    }

    for (i=0; i<NODES; i++) {
        if (inDegree[i]==0) {
            enqueue(q, newNode(i+1));
        }
    }

    while (isQueueEmpty(q)!=1) {
        dequeue(q, &lN);
        printf("%d ", lN->vertexLabel);
        aL->visitedList[lN->vertexLabel-1]=1;
        cN=aL->ptrArray[lN->vertexLabel-1];
        free(lN);
        while (cN!=NULL) {
            inDegree[cN->vertexLabel-1]--;
            cN=cN->next;
        }
        for (i=0; i<NODES; i++) {
            if (inDegree[i]==0 && aL->visitedList[i]!=1) {
                enqueue(q, newNode(i+1));
            }
        }
    }
    free(q);
}
```



inDegree	0	0	0	0	1
vertices	1	2	3	4	5

visitedList	1	1	1	0	0
vertices	1	2	3	4	5

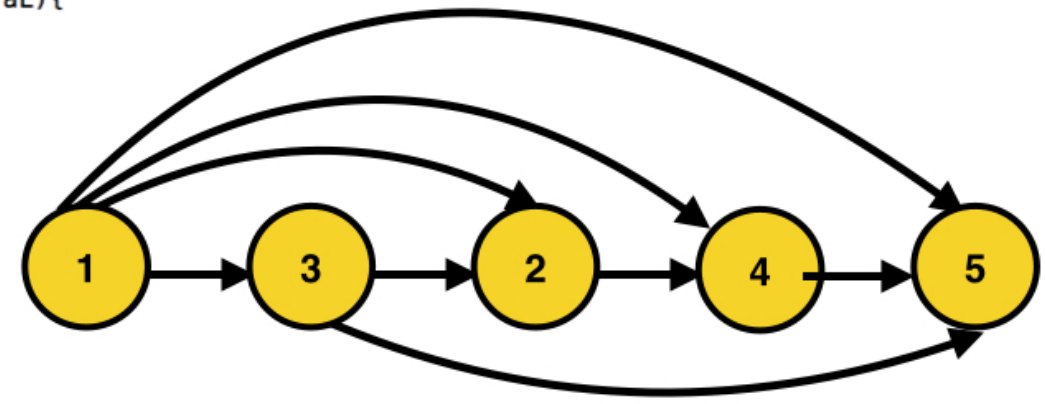
Console:
1 3 2

4

Q

Implementation of Topological Ordering of DAGs

```
void topologicalOrdering(struct adjList * aL){
    int inDegree[NODES]={0};
    struct Queue * q;
    struct listNode * lN, * cN;
    int i;
    initQueue(&q);

    markVerticesAsUnvisited(aL);

    for (i=0; i<NODES; i++) {
        lN=aL->ptrArray[i];
        while (lN!=NULL) {
            inDegree[lN->vertexLabel-1]++;
            lN=lN->next;
        }
    }

    for (i=0; i<NODES; i++) {
        if (inDegree[i]==0) {
            enqueue(q, newNode(i+1));
        }
    }

    while (isQueueEmpty(q)!=1) {
        dequeue(q, &lN);
        printf("%d ", lN->vertexLabel);
        aL->visitedList[lN->vertexLabel-1]=1;
        cN=aL->ptrArray[lN->vertexLabel-1];
        free(lN);
        while (cN!=NULL) {
            inDegree[cN->vertexLabel-1]--;
            cN=cN->next;
        }
        for (i=0; i<NODES; i++) {
            if (inDegree[i]==0 && aL->visitedList[i]!=1) {
                enqueue(q, newNode(i+1));
            }
        }
    }
    free(q);
}
```



inDegree	0	0	0	0	0
vertices	1	2	3	4	5

visitedList	1	1	1	1	0
vertices	1	2	3	4	5

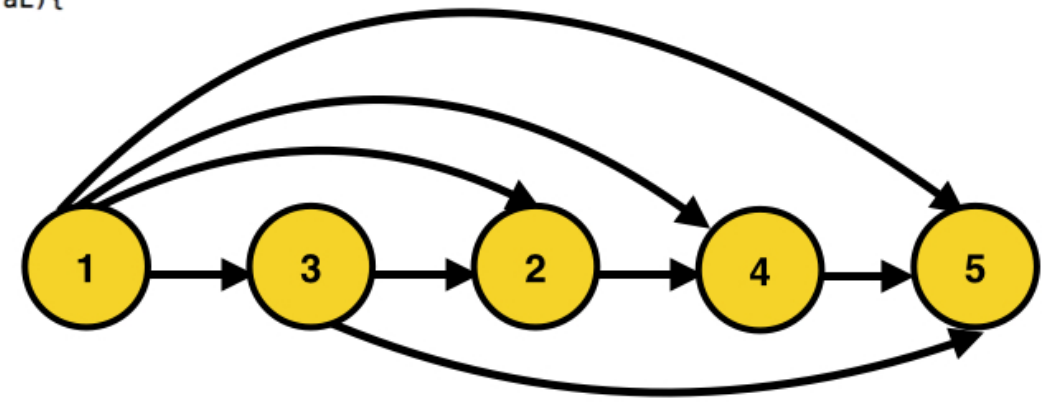
Console:
1 3 2 4

5

Q

Implementation of Topological Ordering of DAGs

```
void topologicalOrdering(struct adjList * aL){
    int inDegree[NODES]={0};
    struct Queue * q;
    struct listNode * lN, * cN;
    int i;
    initQueue(&q);

    markVerticesAsUnvisited(aL);

    for (i=0; i<NODES; i++) {
        lN=aL->ptrArray[i];
        while (lN!=NULL) {
            inDegree[lN->vertexLabel-1]++;
            lN=lN->next;
        }
    }

    for (i=0; i<NODES; i++) {
        if (inDegree[i]==0) {
            enqueue(q, newNode(i+1));
        }
    }

    while (isQueueEmpty(q)!=1) {
        dequeue(q, &lN);
        printf("%d ", lN->vertexLabel);
        aL->visitedList[lN->vertexLabel-1]=1;
        cN=aL->ptrArray[lN->vertexLabel-1];
        free(lN);
        while (cN!=NULL) {
            inDegree[cN->vertexLabel-1]--;
            cN=cN->next;
        }
        for (i=0; i<NODES; i++) {
            if (inDegree[i]==0 && aL->visitedList[i]!=1) {
                enqueue(q, newNode(i+1));
            }
        }
    }
    free(q);
}
```



inDegree	0	0	0	0	0
vertices	1	2	3	4	5

visitedList	1	1	1	1	1
vertices	1	2	3	4	5

Console:
1 3 2 4 5

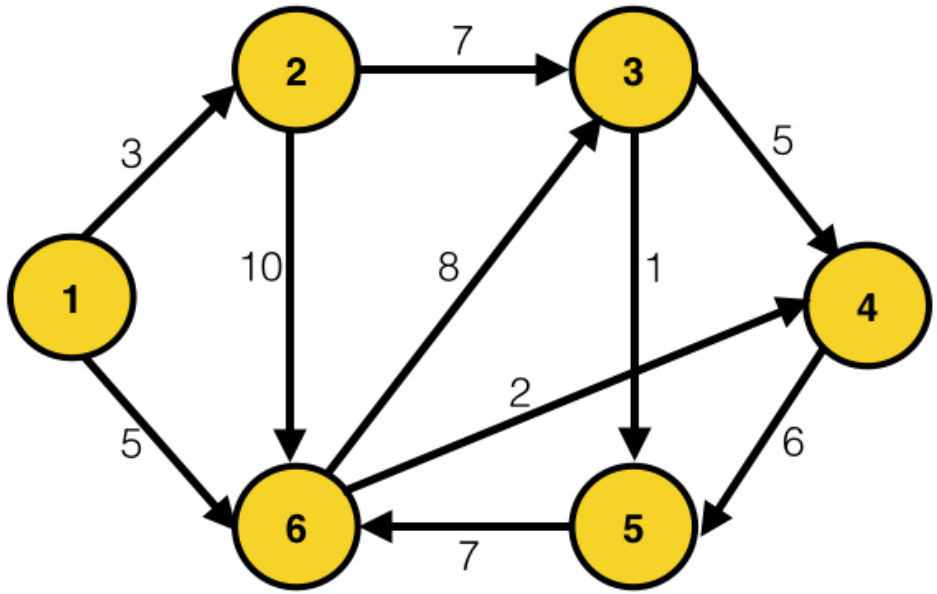
Q

Table of Contents

Shortest Path in a Graph

- **Weighted** digraphs are graphs containing **directed** edges that are associated with values/**weights**
- Weights can be associated with distances, costs, times, etc
- Examples of weighted digraphs are:
  - Vertices/nodes representing cities, edges depicting roads connecting cities and weights on edges denoting the cost/distances/time
- Given a weighted digraph, one important question that may arise is: What is the **shortest path** from city A to city B?
- A **brute force** method may require exploring all possible permutations of nodes in the graph to select the least cost path

An Example of a Weighted Digraph



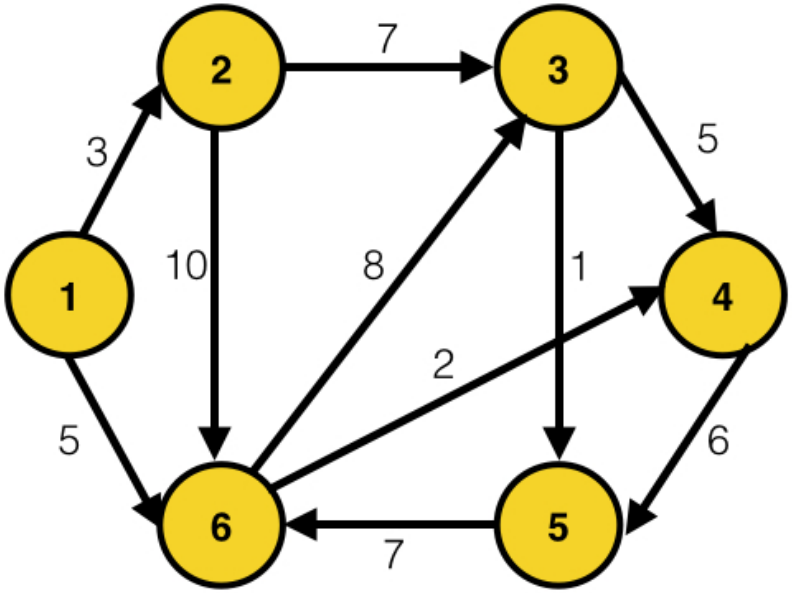
Shortest Path in a Graph

- A very well known algorithm that tackles this problem of finding the shortest path is called the **Dijkstra's** algorithm
- In order to implement this algorithm in C, we shall make use of **adjacency matrix** representation of the graph
  - $A[i,j] = w_{i,j}$ if $e_k = (v_i, v_j) \in E$ and $weight(e_k) = w_{i,j}$
  - Else $A[i,i] = 0$ otherwise $A[i,j] = \infty$
- If the start s and destination d nodes are not directly connected by an edge, the Dijkstra's algorithm seeks for the next best candidate that can serve as an intermediate node connecting s and d at every stage of the algorithm

Summary of Dijkstra's Approach

- Two different sets of vertices W and V are maintained
- At every stage, a new node w from V is added to W if it has **minimal distance** to a node in W
- W initially contains the node s and is **enlarged** over the course of the algorithm with *good* candidates that can form a path from s to d until all vertices in the graph are added to W
- Suppose that node w is added to W and the shortest distance of w from s is sDw
- Since a new node has been added to W the shortest distance of all other nodes in $v \in V$ to s may be better if these can connect via w
- Suppose that the current shortest distance of v from s is sDv and the distance of v to s via w is $sDw + A[w][v]$
- sDv is updated to $sDw + A[w][v]$ if this has a smaller value than the current sDv (this means that the cost of connecting to s via w is smaller than the cost of connecting to s in the previous stage)

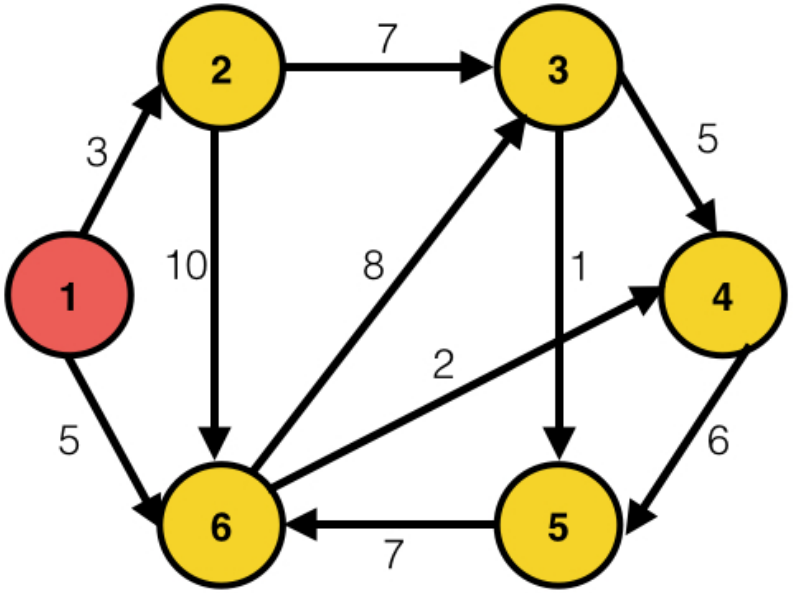
Application of Dijkstra's Algorithm



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Find the distance of the shortest path from 1 to 5

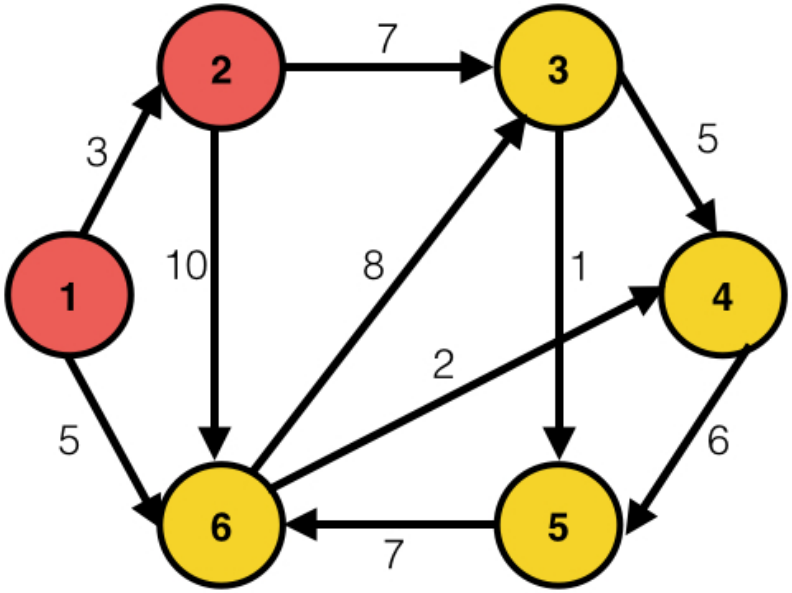
Application of Dijkstra's Algorithm



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Stage	W	V	w	sDw	sD1	sD2	sD3	sD4	sD5	sD6
1	1	2,3,4,5,6	1	0	0	**3**	∞	∞	∞	5
2										
3										
4										
5										
6										

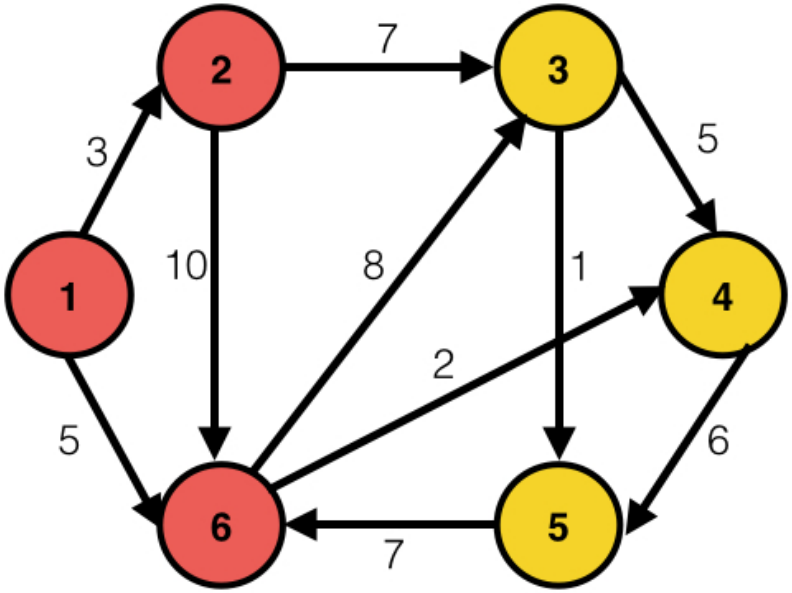
Application of Dijkstra's Algorithm



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Stage	W	V	w	sDw	sD1	sD2	sD3	sD4	sD5	sD6
1	1	2,3,4,5,6	1	0	0	3	∞	∞	∞	5
2	1,2	3,4,5,6	2	3	0	3	10	∞	∞	**5**
3										
4										
5										
6										

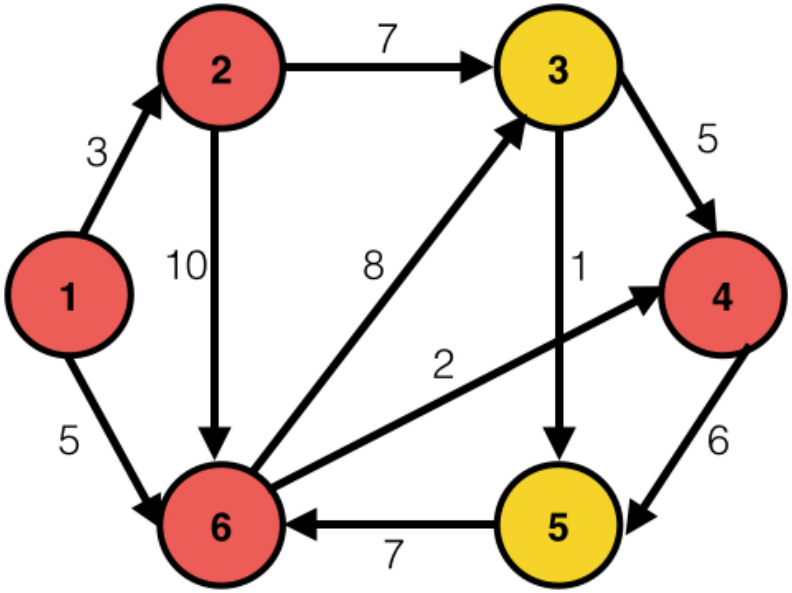
Application of Dijkstra's Algorithm



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Stage	W	V	w	sDw	sD1	sD2	sD3	sD4	sD5	sD6
1	1	2,3,4,5,6	1	0	0	3	∞	∞	∞	5
2	1,2	3,4,5,6	2	3	0	3	10	∞	∞	5
3	1,2,6	3,4,5	6	5	0	3	10	**7**	∞	5
4										
5										
6										

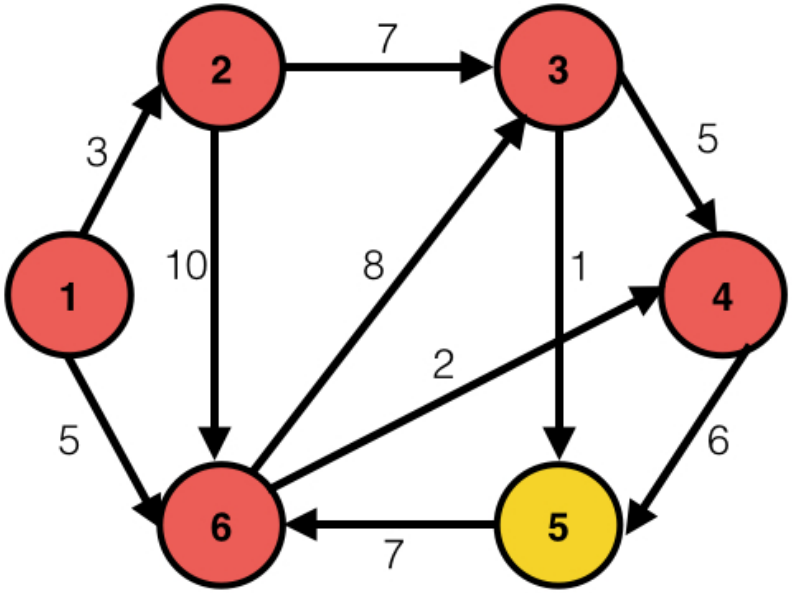
Application of Dijkstra's Algorithm



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Stage	W	V	w	sDw	sD1	sD2	sD3	sD4	sD5	sD6
1	1	2,3,4,5,6	1	0	0	3	∞	∞	∞	5
2	1,2	3,4,5,6	2	3	0	3	10	∞	∞	5
3	1,2,6	3,4,5	6	5	0	3	10	7	∞	5
4	1,2,6,4	3,5	4	7	0	3	**10**	7	13	5
5										
6										

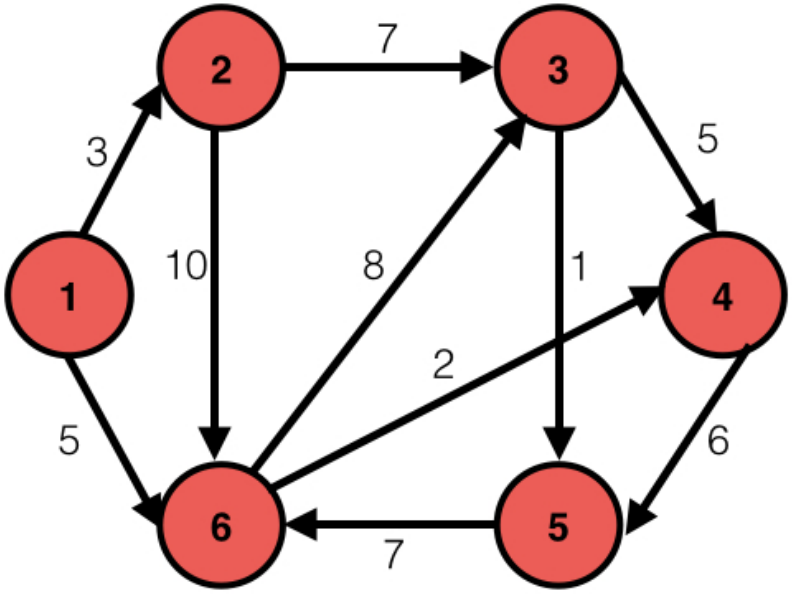
Application of Dijkstra's Algorithm



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Stage	W	V	w	sDw	sD1	sD2	sD3	sD4	sD5	sD6
1	1	2,3,4,5,6	1	0	0	3	∞	∞	∞	5
2	1,2	3,4,5,6	2	3	0	3	10	∞	∞	5
3	1,2,6	3,4,5	6	5	0	3	10	7	∞	5
4	1,2,6,4	3,5	4	7	0	3	10	7	13	5
5	1,2,6,4,3	5	3	10	0	3	10	7	**11**	5
6										

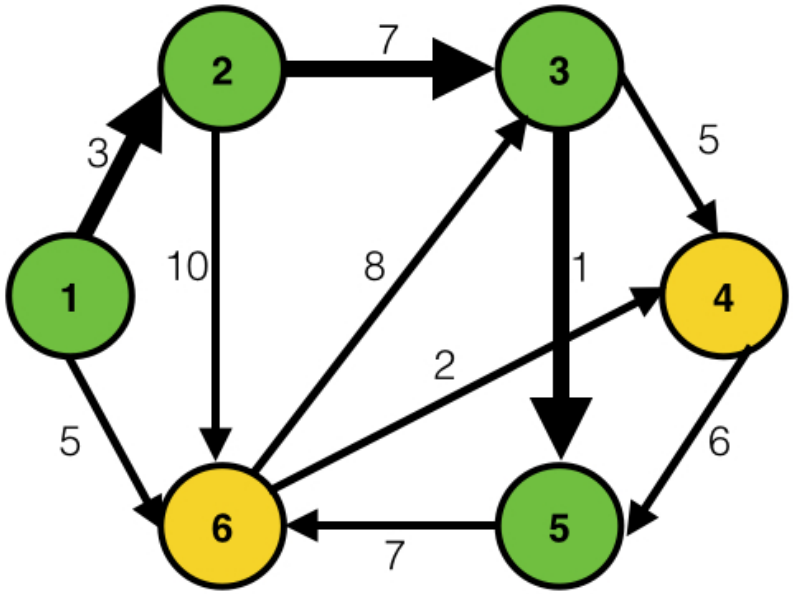
Application of Dijkstra's Algorithm



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Stage	W	V	w	sDw	sD1	sD2	sD3	sD4	sD5	sD6
1	1	2,3,4,5,6	1	0	0	3	∞	∞	∞	5
2	1,2	3,4,5,6	2	3	0	3	10	∞	∞	5
3	1,2,6	3,4,5	6	5	0	3	10	7	∞	5
4	1,2,6,4	3,5	4	7	0	3	10	7	13	5
5	1,2,6,4,3	5	3	10	0	3	10	7	11	5
6	1,2,6,4,3,5		5	**11**	0	3	10	7	11	5

Application of Dijkstra's Algorithm



	1	2	3	4	5	6
1	0	3	∞	∞	∞	5
2	∞	0	7	∞	∞	10
3	∞	∞	0	5	1	∞
4	∞	∞	∞	0	6	∞
5	∞	∞	∞	∞	0	7
6	∞	∞	8	2	∞	0

Stage	W	V	w	sDw	sD1	sD2	sD3	sD4	sD5	sD6
1	1	2,3,4,5,6	**1**	0	0	**3**	∞	∞	∞	5
2	1,2	3,4,5,6	**2**	3	0	3	**10**	∞	∞	5
3	1,2,6	3,4,5	6	5	0	3	10	7	∞	5
4	1,2,6,4	3,5	4	7	0	3	10	7	13	5
5	1,2,6,4,3	5	**3**	10	0	3	10	7	**11**	5
6	1,2,6,4,3,5		5	**11**	0	3	10	7	11	5

Implementation of Dijkstra's Algorithm

Implementation Summary:

1. Initialization:
   i. Create two lists for V and W
   ii. Set the starting vertex to be w
   iii. Insert w into W and insert remaining vertices into V
2. while-loop conditioned on V not being empty
   i. Find the node with the minimal distance to s, set it to be w, remove it from V and insert it into W
   ii. Update the distance metric of all nodes remaining in V to $sDv = min(sDv, sDw + A[w][v])$

Implementation of Dijkstra's Algorithm

```
struct list * shortestPath(struct adjMatrix * aM, int startVertex){
    struct list * V = initList();
    struct list * W = initList();
    struct listNode * currNode;
    int w=startVertex, minDistance, i;
    int shortestDistance[NODES];
    insertList(W, startVertex);
    for (i=0; i<NODES; i++) {
        if (i+1!=startVertex) {
            insertList(V, i+1);
        }
        shortestDistance[i]=aM->matrix[w-1][i];
    }
    while (V->l!=NULL) {
        minDistance=INF;
        currNode=V->l;
        while (currNode!=NULL) {
            if (minDistance > shortestDistance[currNode->vertexLabel-1]) {
                minDistance=shortestDistance[currNode->vertexLabel-1];
                w=currNode->vertexLabel;
            }
            currNode=currNode->next;
        }
        insertList(W, w);
        removeList(V, w);
        for (i=0; i<NODES; i++) {
            shortestDistance[i]=min(shortestDistance[i],
                                    shortestDistance[w-1]+aM->matrix[w-1][i]);
        }
    }
    for (i=0; i<NODES; i++) {
        printf("shortest distance of %i from 1 is %d \n", i+1, shortestDistance[i]);
    }
    free(V);
    return W;
}
```

Performance: $O(n^2)$

Implementation of Dijkstra's Algorithm

Initialization

```
struct list * V = initList();
struct list * W = initList();
struct listNode * currNode;
int w=startVertex, minDistance, i;
int shortestDistance[NODES];
insertList(W, startVertex);
for (i=0; i<NODES; i++) {
    if (i+1!=startVertex) {
        insertList(V, i+1);
    }
    shortestDistance[i]=aM->matrix[w-1][i];
}
```

i Create two lists for V and W

ii Set the starting vertex to be w

iii Insert w into W and insert remaining vertices into V

Implementation of Dijkstra's Algorithm

While-loop

```
while (V->l!=NULL) {
    minDistance=INF;
    currNode=V->l;
    while (currNode!=NULL) {
        if (minDistance > shortestDistance[currNode->vertexLabel-1]) {
            minDistance=shortestDistance[currNode->vertexLabel-1];
            w=currNode->vertexLabel;
        }
        currNode=currNode->next;
    }
    insertList(W, w);
    removeList(V, w);
    for (i=0; i<NODES; i++) {
        shortestDistance[i]=min(shortestDistance[i],
                                shortestDistance[w-1]+aM->matrix[w-1][i]);
    }
}
```

i Find the node with the minimal distance to s, set it to be w, remove it from V and insert it into W

ii Update the distance metric of all nodes remaining in V to $sDv = min(sDv, sDw + A[w][v])$

Proving that Dijkstra's Algorithm Works via Induction

Base case: Prove that Dijkstra's algorithm works for the stage 2 (where $W = v_s$)

- A new addition into W at Stage 2 is the one that is the closest to v_s

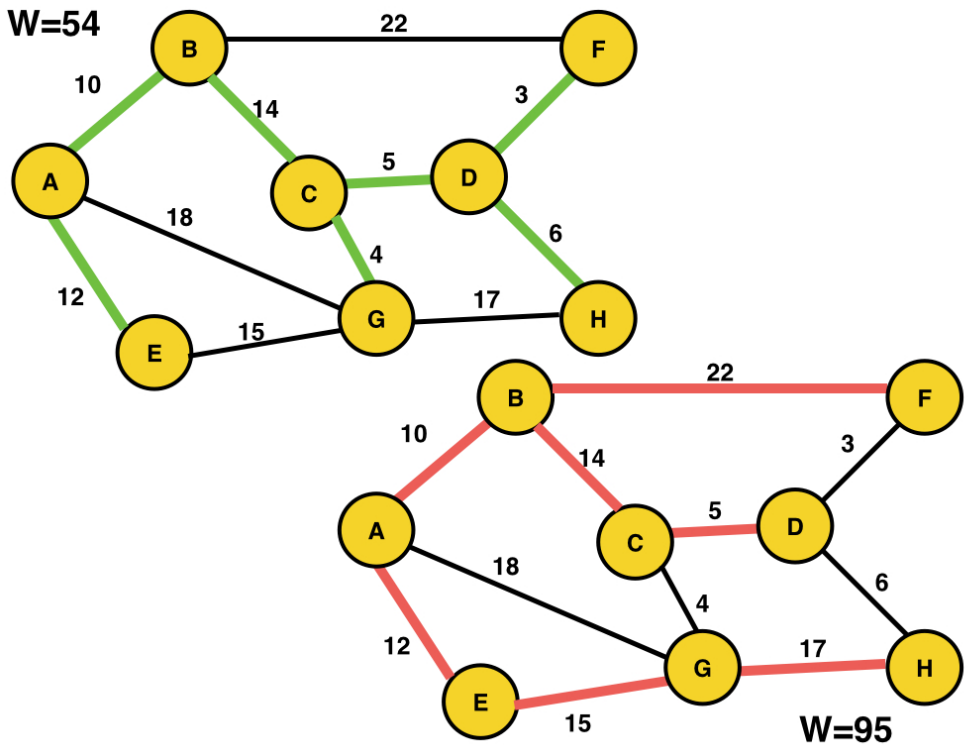
Inductive step: Prove that if the Dijkstra's algorithm holds for the i^{th} stage then it also holds for $(i+1)^{th}$ stage

- Suppose that node w is selected by the Dijkstra's algorithm to be the next node to be inserted into W at Stage $i+1$ but there exists another node r with a shorter path from v_s (**Proof by contradiction**)
- This cannot be true due to the following reasons:
  - All nodes added to W up to stage i do result in the shortest path (based on our assumption)
  - Shortest distances of all other nodes in $x \in V$ are updated based on $min(sD_x, sD_w + T[w][x])$
  - Since $T[w][x]$ is not a negative number, the next candidate selected based on this metric must have the minimal distance to the set W

Table of Contents

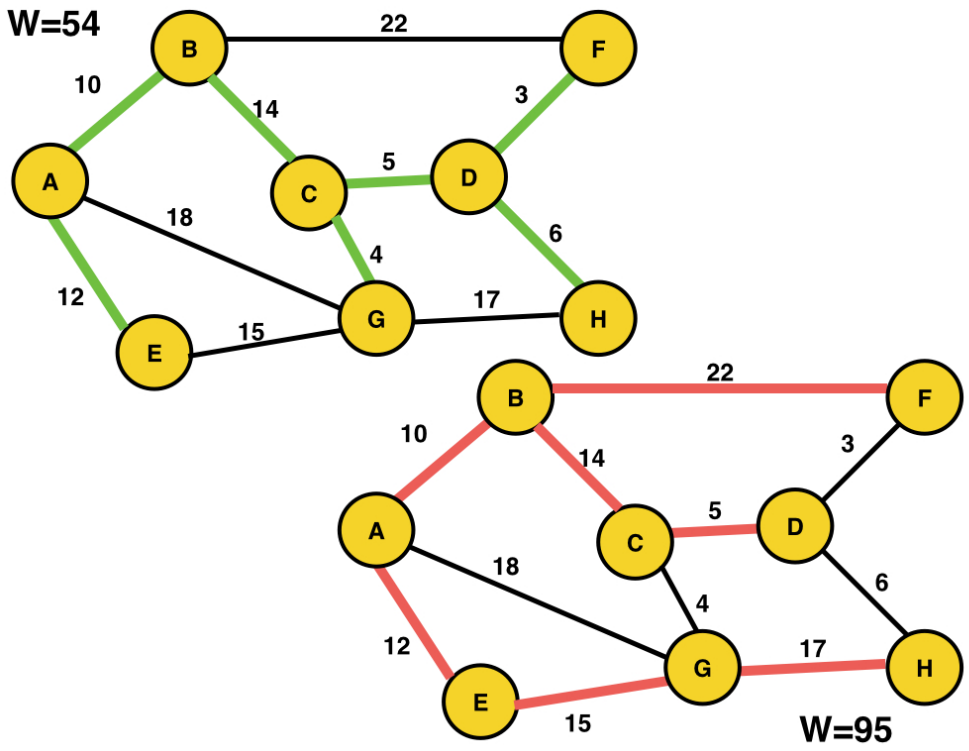
Spanning Trees

- Consider an **undirected**, **connected** graph $G(V, E)$ with **weighted** edges
- A **spanning tree** is a tree constructed using some edges in E that connects all vertices in the graph
- The sum of all weights on the edges used to create the spanning tree is the **weight** W of the spanning tree



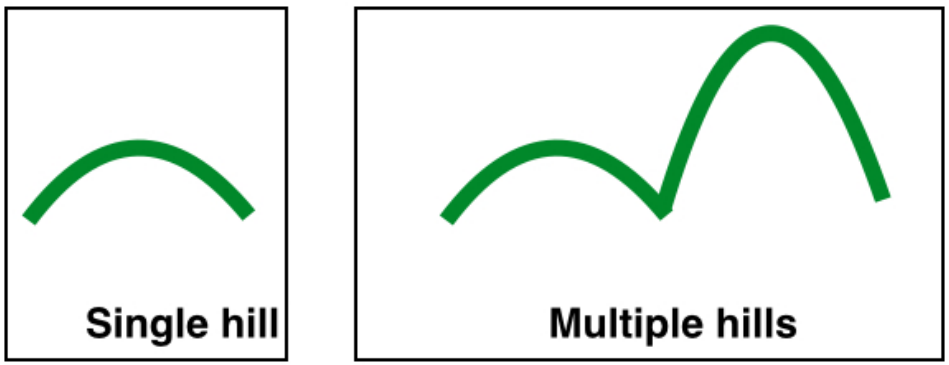
Minimal Spanning Trees

- The spanning tree with **minimal** such summed weight W is called the **minimal spanning tree**
- What are the applications of these?
  - Circuits (need to connect separate electric terminals but cost of wiring is important)
  - Communications: broadcasting signals

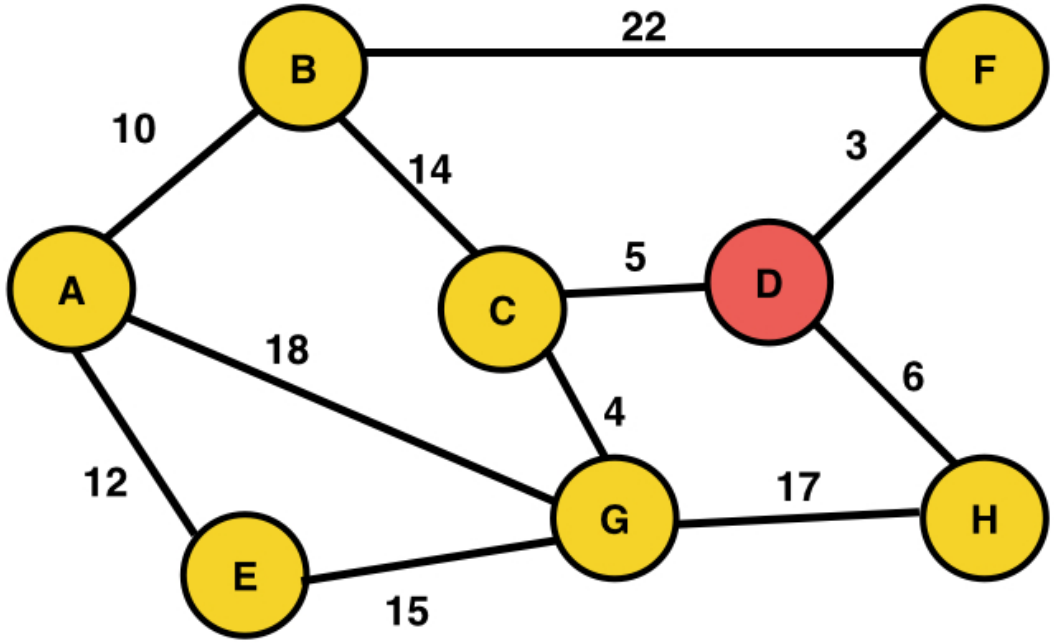


Finding the Minimal Spanning Tree

- **Prim's algorithm** is used widely to construct the minimal spanning tree of a graph (named after Dr Robert C. Prim)
- Prim's algorithm is also called the **greedy** algorithm as decisions are made without taking the full picture into account (local decisions)
- This is one of the **few** greedy algorithm that results in the **globally optimal solution**
  - Suppose that there only exists one hill in a region
  - How can you reach the top/summit of the tallest hill?
  - One way is to travel in the direction that is locally steep (ascent is fastest)
  - This will not work if there are multiple hills in the region - you might end up at a hill that is not the tallest
- Typically greedy algorithms do **not** work for most applications



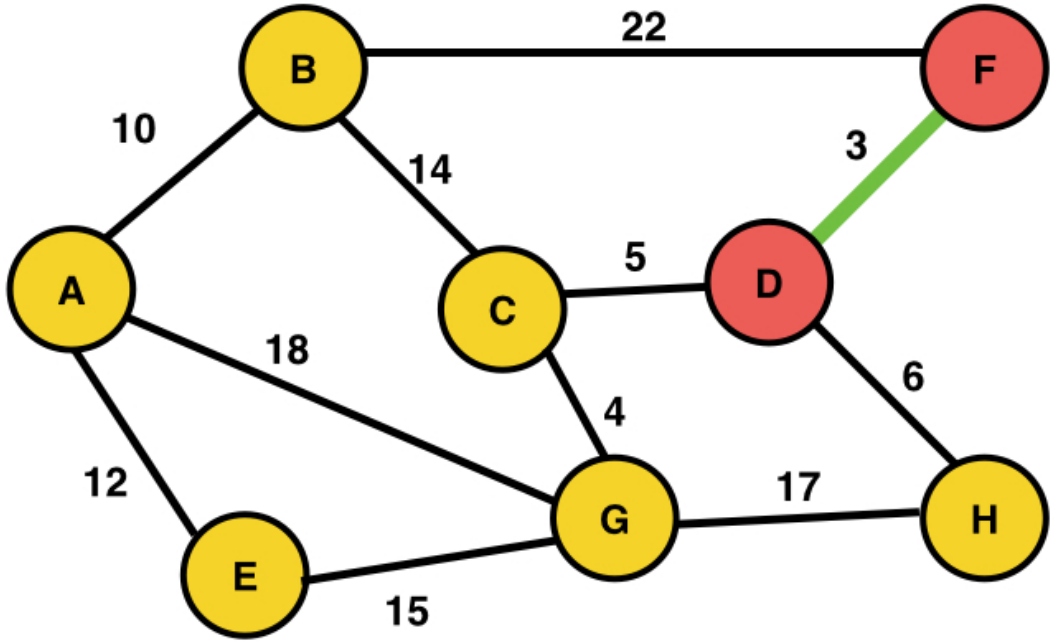
Prim's Algorithm in Action



T: D

U:

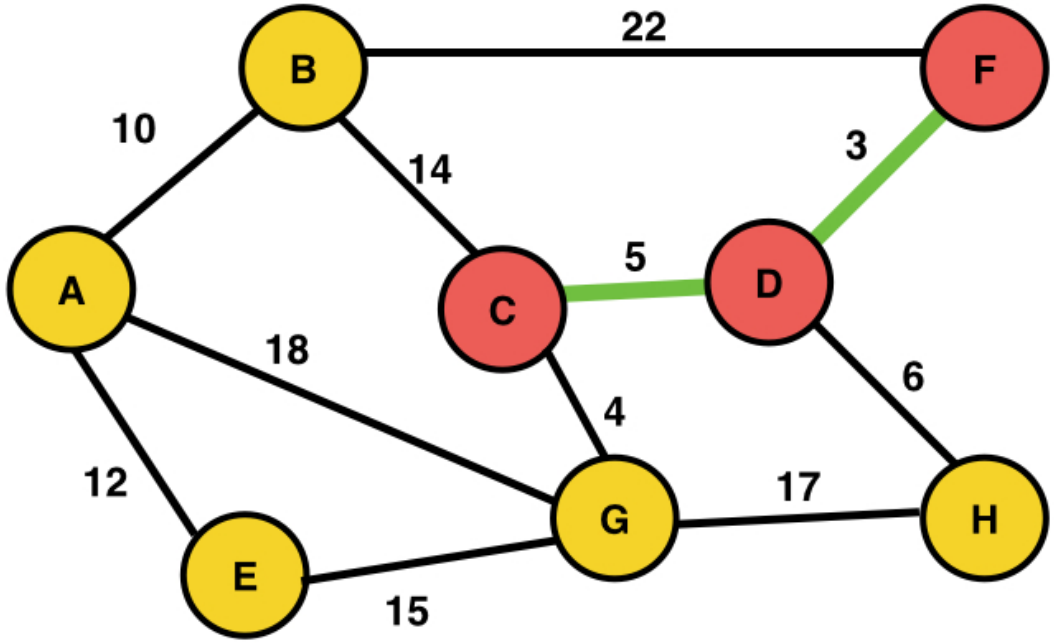
Prim's Algorithm in Action



T: D, F

U: (D,F)

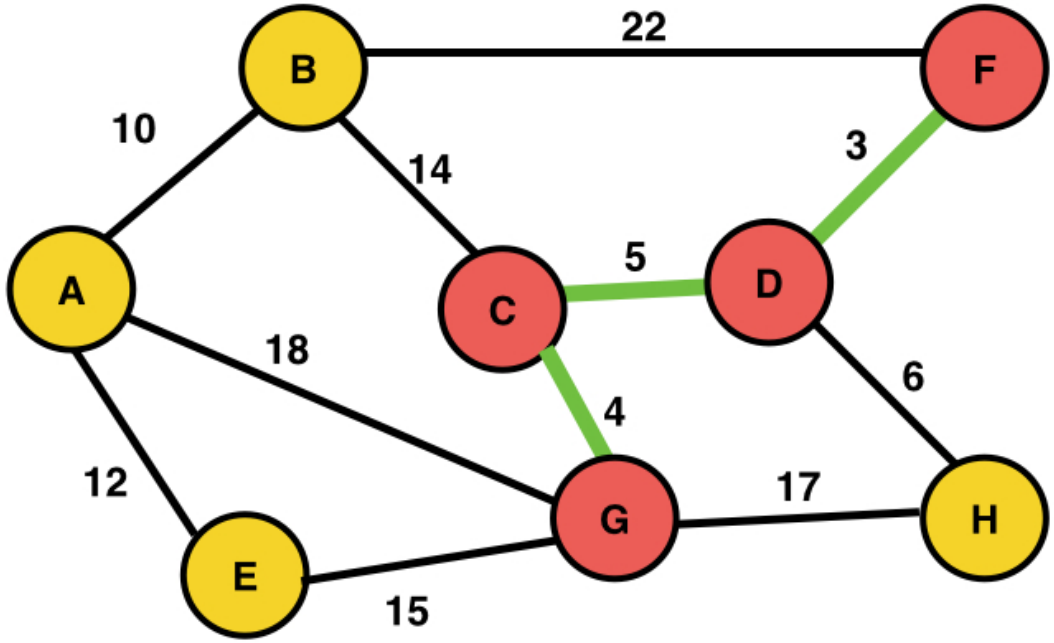
Prim's Algorithm in Action



T: D, F, C

U: (D,F), (D,C)

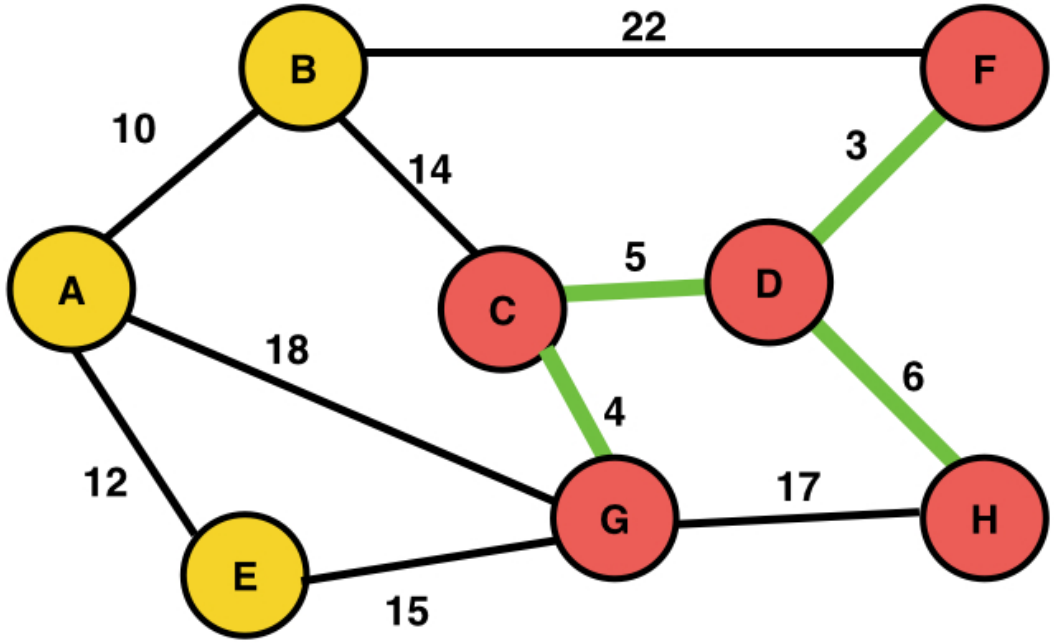
Prim's Algorithm in Action



T: D, F, C, G

U: (D,F), (D,C), (C,G)

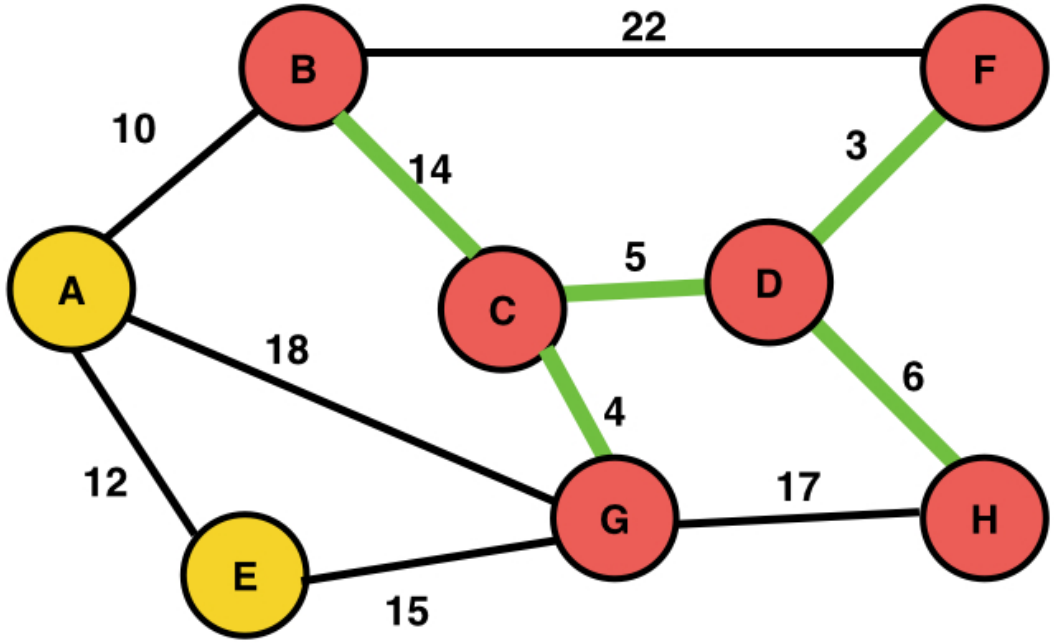
Prim's Algorithm in Action



T: D, F, C, G, H

U: (D,F), (D,C), (C,G), (D,H)

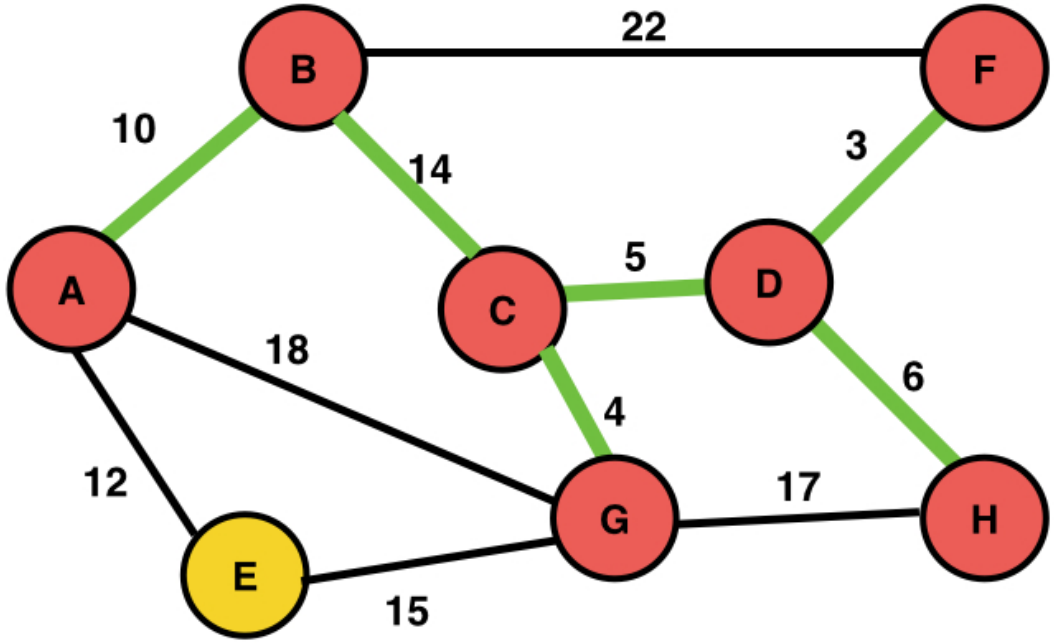
Prim's Algorithm in Action



T: D, F, C, G, H, B

U: (D,F), (D,C), (C,G), (D,H), (C,B)

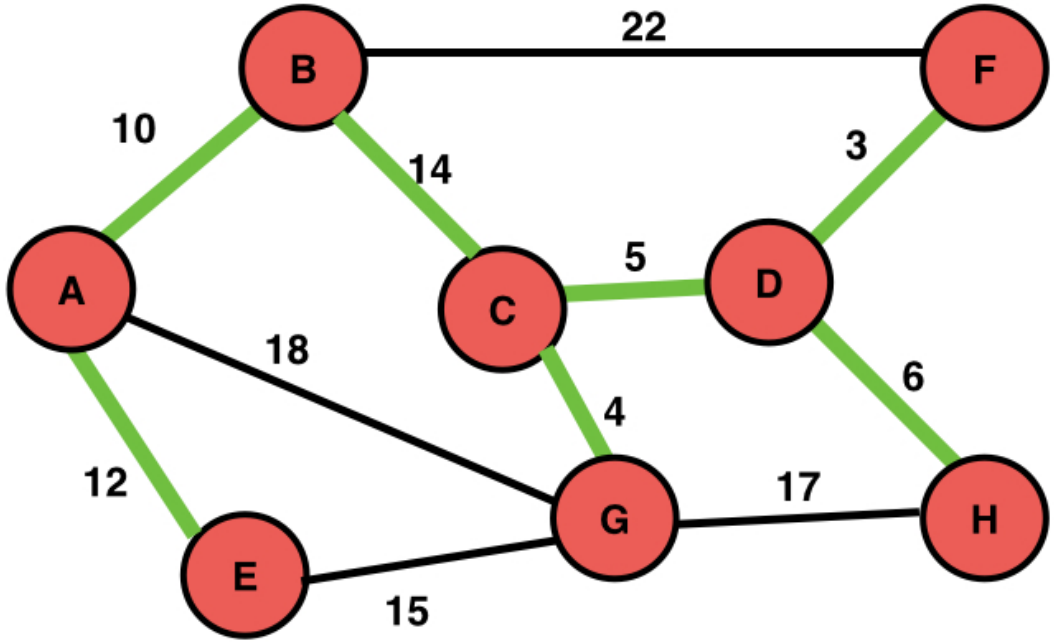
Prim's Algorithm in Action



T: D, F, C, G, H, B, A

U: (D,F), (D,C), (C,G), (D,H), (C,B), (B,A)

Prim's Algorithm in Action



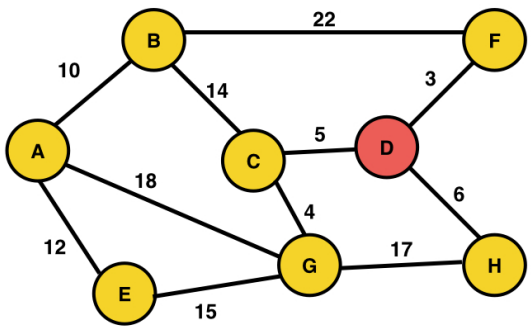
T: D, F, C, G, H, B, A, E

U: (D,F), (D,C), (C,G), (D,H), (C,B), (B,A), (A,E)

Summary of the Prim's Algorithm

Sets T (vertex set) and U (edge set) are maintained throughout this algorithm

1. First any node in the graph is selected and added to T
2. At every iteration of this algorithm, an edge with minimal weight that connects a node not in T to a node in T is added to the edge set U
3. The node in the newly added edge that is not in T is added to T
4. This is repeated until all vertices in G are added to T



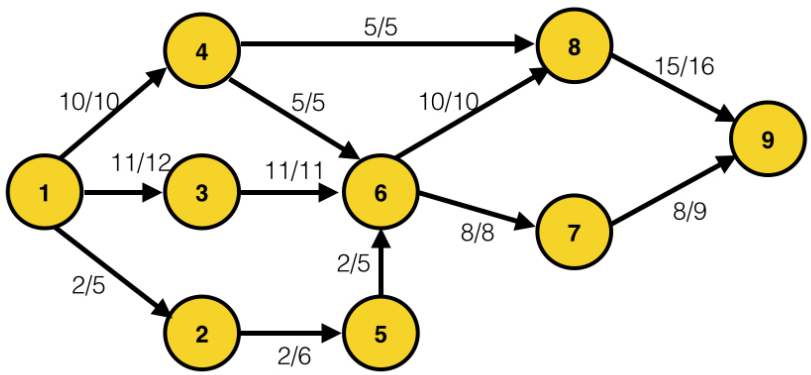
T: D

U:

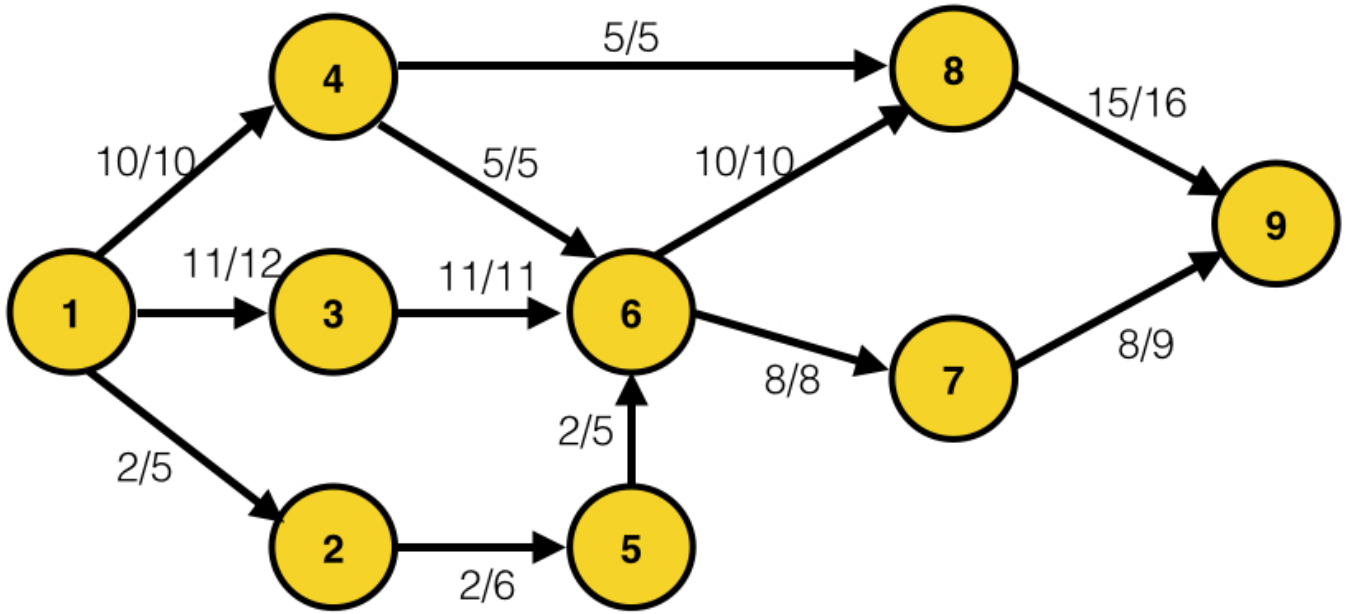
Table of Contents

What are Flow Networks?

- In **flow networks**, substance originates at S (**source**) and is consumed at D (**destination**)
- Consider an oil pipeline network in which vertices are pumping stations and edges are pipelines
- Substance can flow through pipelines $e_{i,j}$ starting at S and reach D as long as the flow along each pipeline is within the flow capacity
- Weights on edges of a flow network depicts the current flow $f_{i,j}$ and the capacity $c_{i,j}$ of the edge and is labelled as $f_{i,j}/c_{i,j}$
- What is the **maximum** flow possible throughout the flow network starting from a **source** to a **destination**?



Constraints on a Flow Network



Flow Constraints: $0 \leq f_{i,j} \leq c_{i,j} \ \forall i, j \in V$

Flow Conservation: $\sum_{i \in V} f_{i,j} = \sum_{i \in V} f_{j,i} \forall j \in \{V / \ s, d\}$

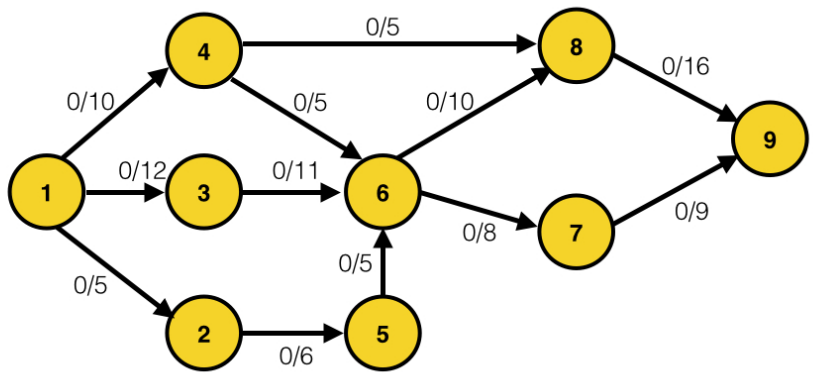
Back Flow: $0 \leq f_{j,i} \leq f_{i,j}$

Ford-Fulkerson Algorithm

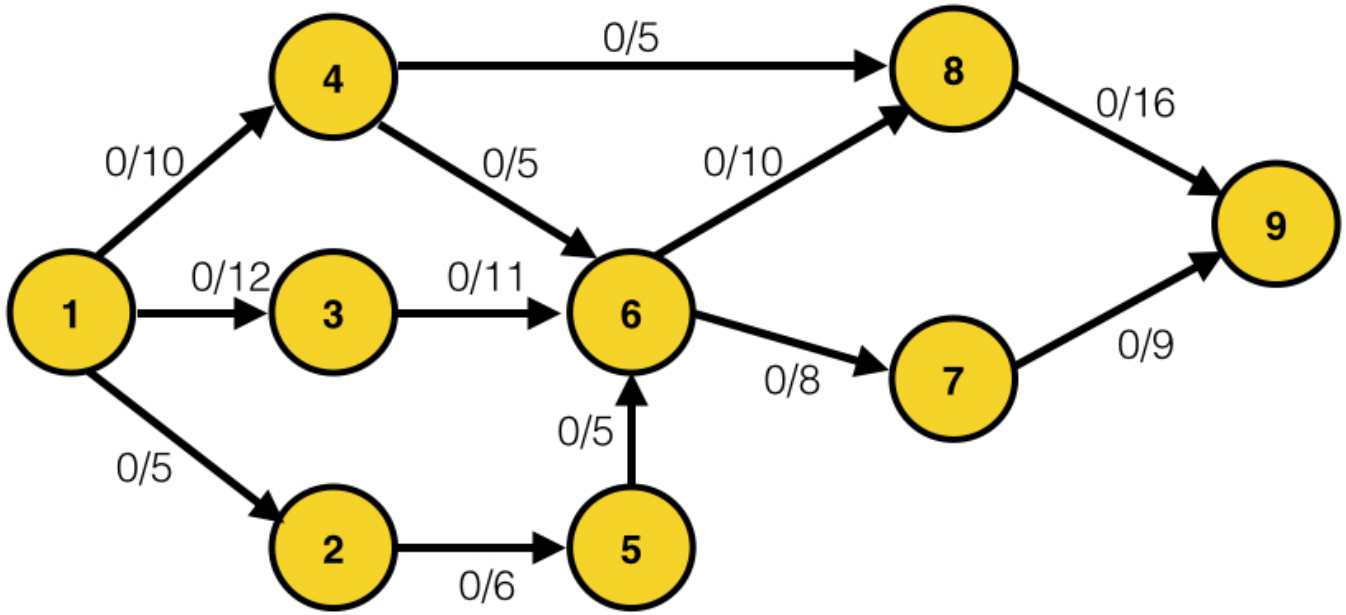
Find an **augmenting path** from S-D

- Path in which **positive flow** can be added to edges that connect S-D i.e. no edges have reached full capacity
- **Breadth-first** path finding algorithm (shortest path from S-D with space to augment)

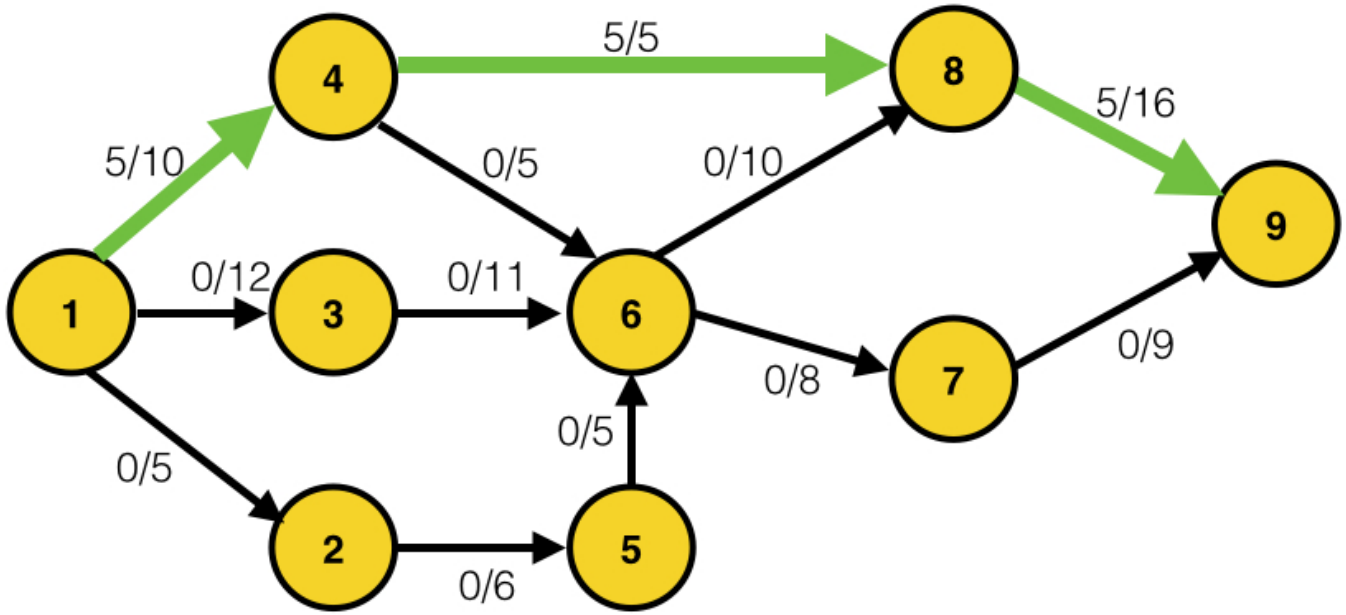
1. **Increment** flow on all edges in the path by the same value which heeds flow constraints on these edges
2. Repeat until no more augmenting path exists



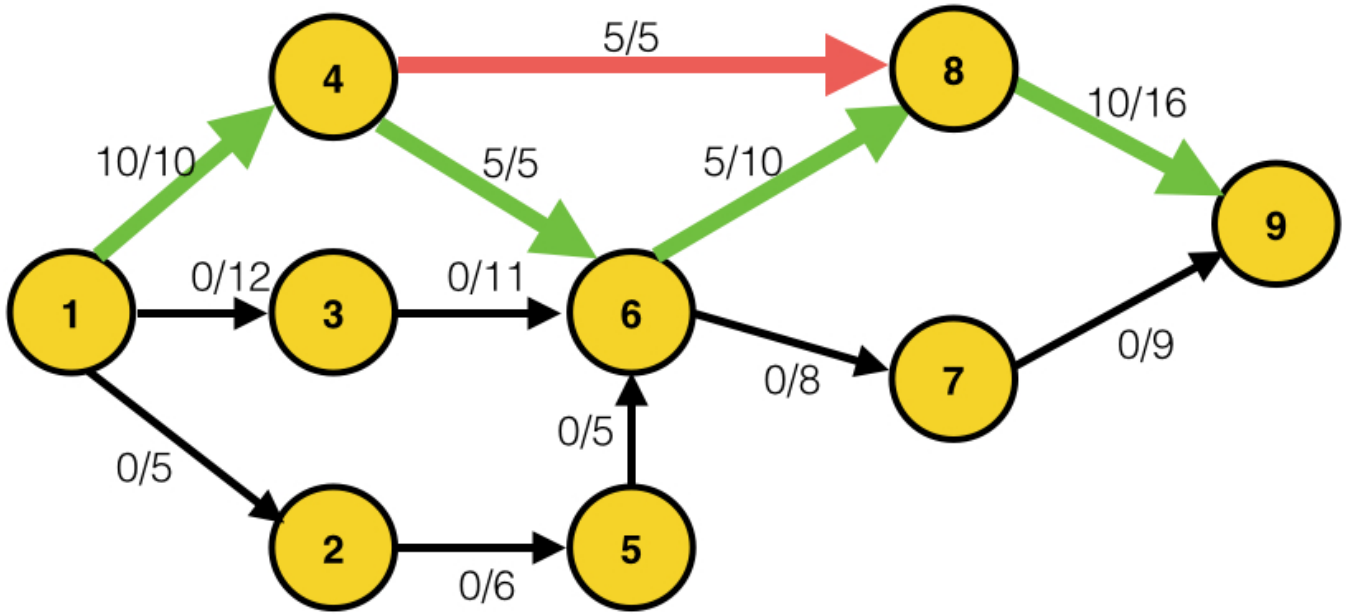
Ford-Fulkerson Algorithm in Action



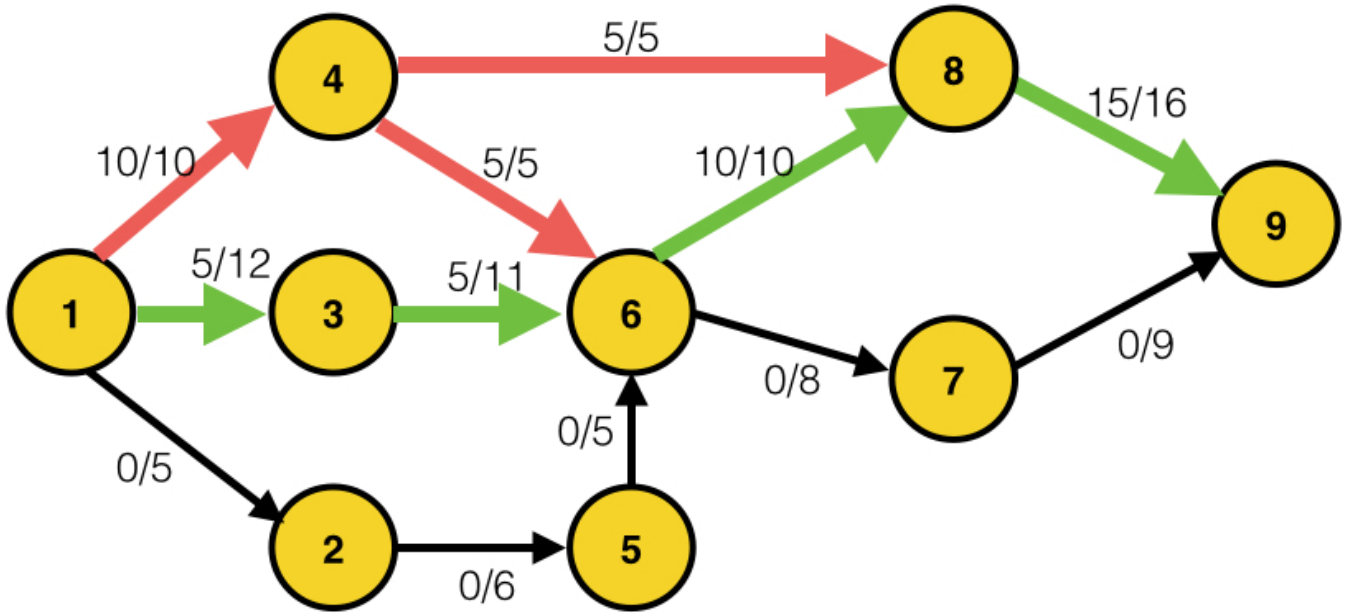
Ford-Fulkerson Algorithm in Action



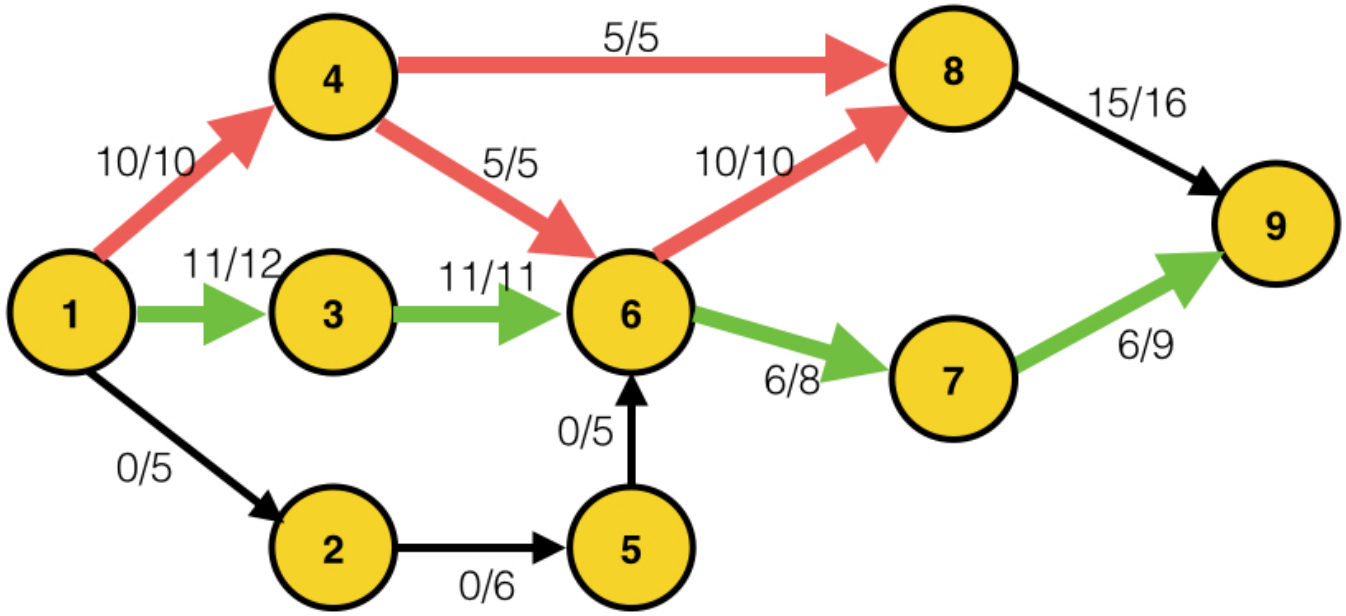
Ford-Fulkerson Algorithm in Action



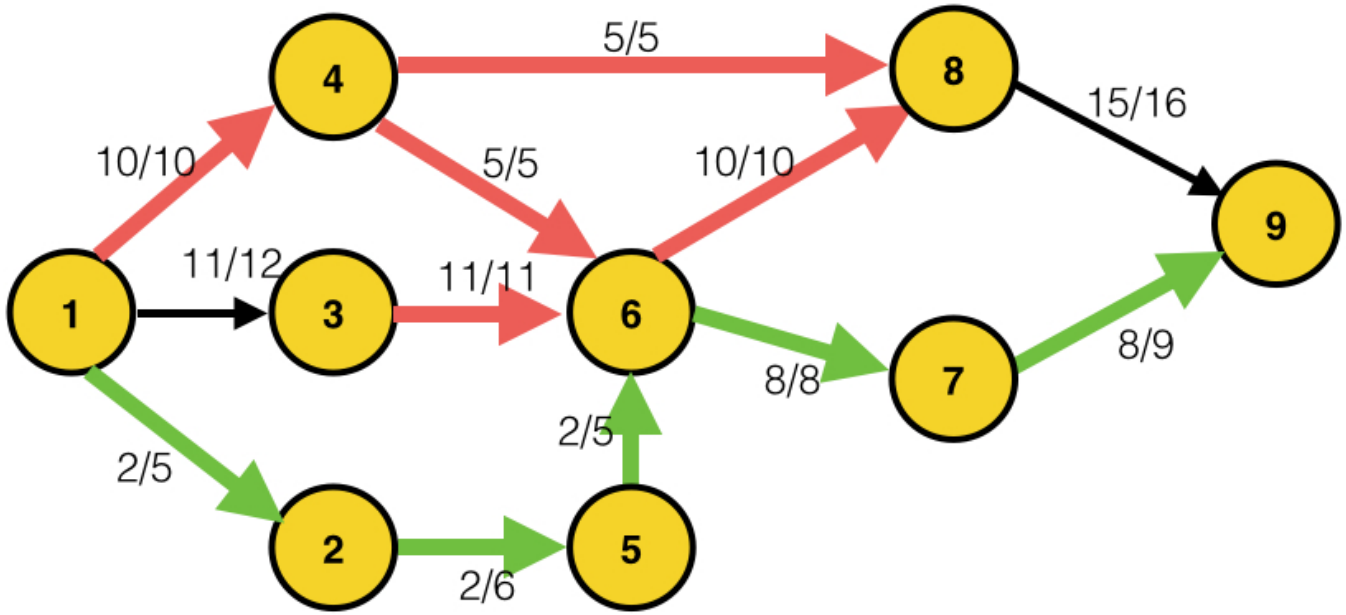
Ford-Fulkerson Algorithm in Action



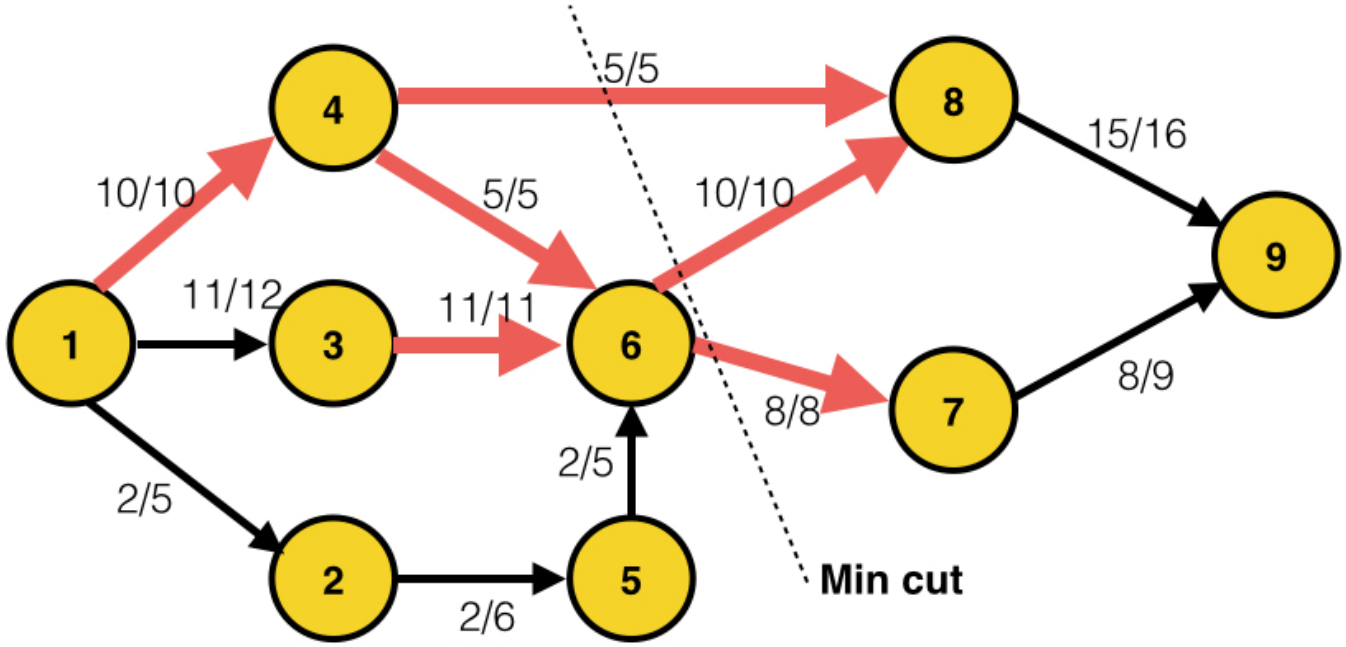
Ford-Fulkerson Algorithm in Action



Ford-Fulkerson Algorithm in Action



Ford-Fulkerson Algorithm in Action

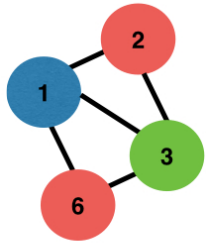


Max Flow: 23

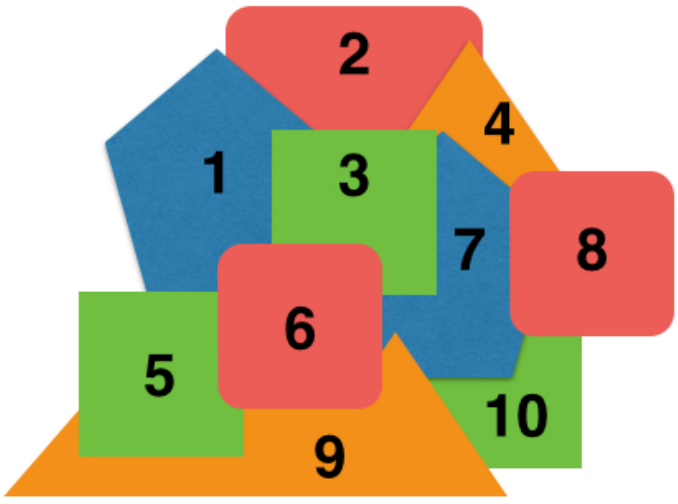
Table of Contents

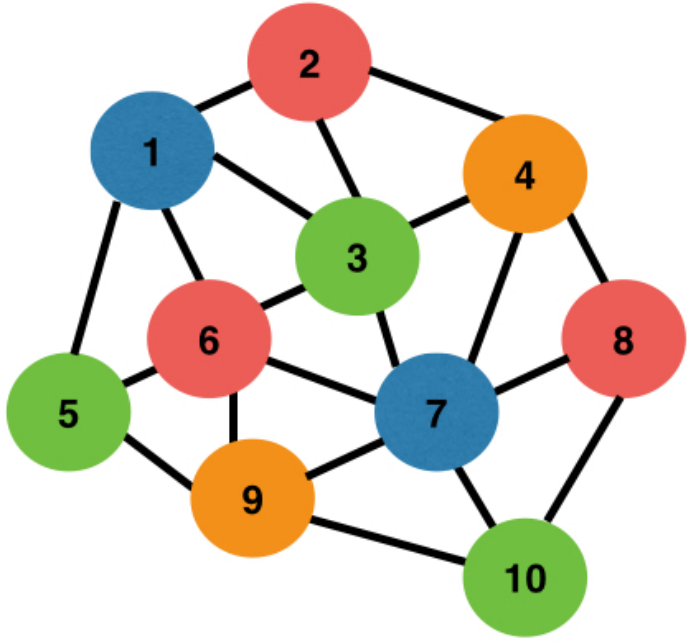
Proper Colouring in Graph

- **Proper colouring** refers to the colouring of vertices in a graph in which no vertices of the same colour are connected by an edge
- The **graph colouring** problem examines the minimum number of colours required to achieve proper colouring
  - If the minimum number of colours required to colour a graph is k, then the graph is called **k-colorable**
- **Four-colouring** is used typically by cartographers to colour countries that are adjacent to one another with different colours belonging to a four colour set
- Intuitively obvious but a formal proof took many years to achieve
- The run-time of the algorithm required to solve this problem is exponential (i.e. NP-complete)



Example: Four-colour Problem



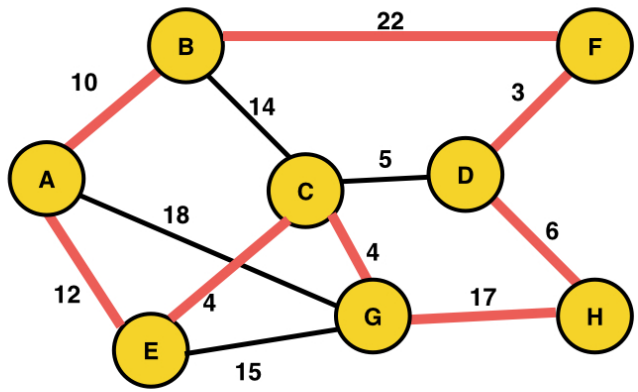


Can convert a map into a planar graph where each city is a vertex and if two cities are adjacent, these are connected by an edge

Table of Contents

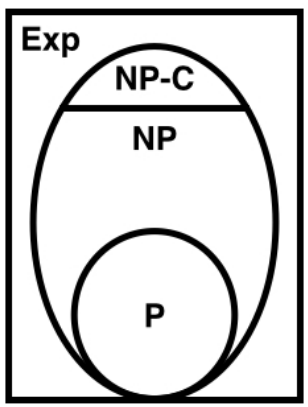
Hamiltonian Circuit Problem

- This is a **decision problem** that examines whether it is possible find a path that visits every vertex in the graph only once before returning to the starting vertex
- This is referred to as the **Hamiltonian cycle**
- What is the complexity of finding a solution to this problem?
- Can use the theory of **satisfiability** (beyond the scope of this course) or **translate** the problem to a well-known problem for which complexity is already established
- We shall try to translate the problem to the travelling salesman problem



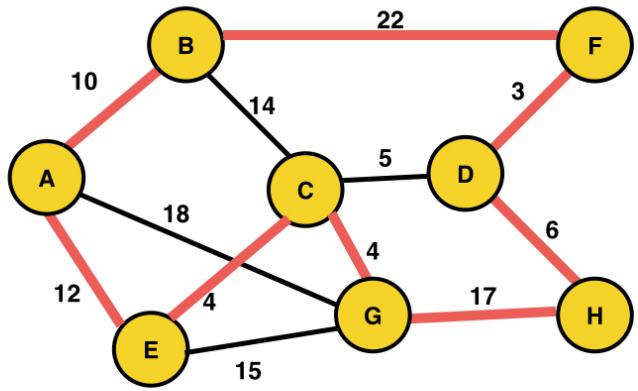
Side Note: NP vs NP-Hard vs NP-Complete

- **P**: Problems can be solved in polynomial time
- **NP**: If the solution to the problem can be verified in polynomial time
- **Reducible**: If a problem A can be translated into problem B with a polynomial time reduction algorithm R
- **NP-Hard**: If any NP problems can be reduced to the NP problem B (this is at least as difficult all difficult problems can get)
  - If a problem A is in NP and it is reducible to problem B (i.e. $B \leq_p A$) in polynomial time



One Version of the Travelling Salesman Problem

- The **travelling salesman problem** is a well-know problem that has many varieties
- The one we will consider here is: find a path in which a salesman can visit every city once and end up at the starting city with cost less than k
- It can be shown via Cook's theorem that the Travelling Salesman Problem is NP-complete (need the notion of **satisfiability**)
- Many problems such as **Hamiltonian Circuit problem** can be shown to be NP-complete as these can be *reduced* to this version of Travelling Salesman problem



An example of Reduction

Converting hamilton circuit problem to TSP:

- Suppose that the hamilton circuit problem is defined for G
- Define G' which contains the same V as G in which there exists an edge for every pair of vertex in V
- There are more edges in G' than G
- Assign a weight of 1 to edges in G' that exist in G and the value of 2 to other edges
- Now pose the problem as a TSP with K=n
- This reduction is polynomial in complexity and therefore the Hamilton Circuit problem is NP-Complete!