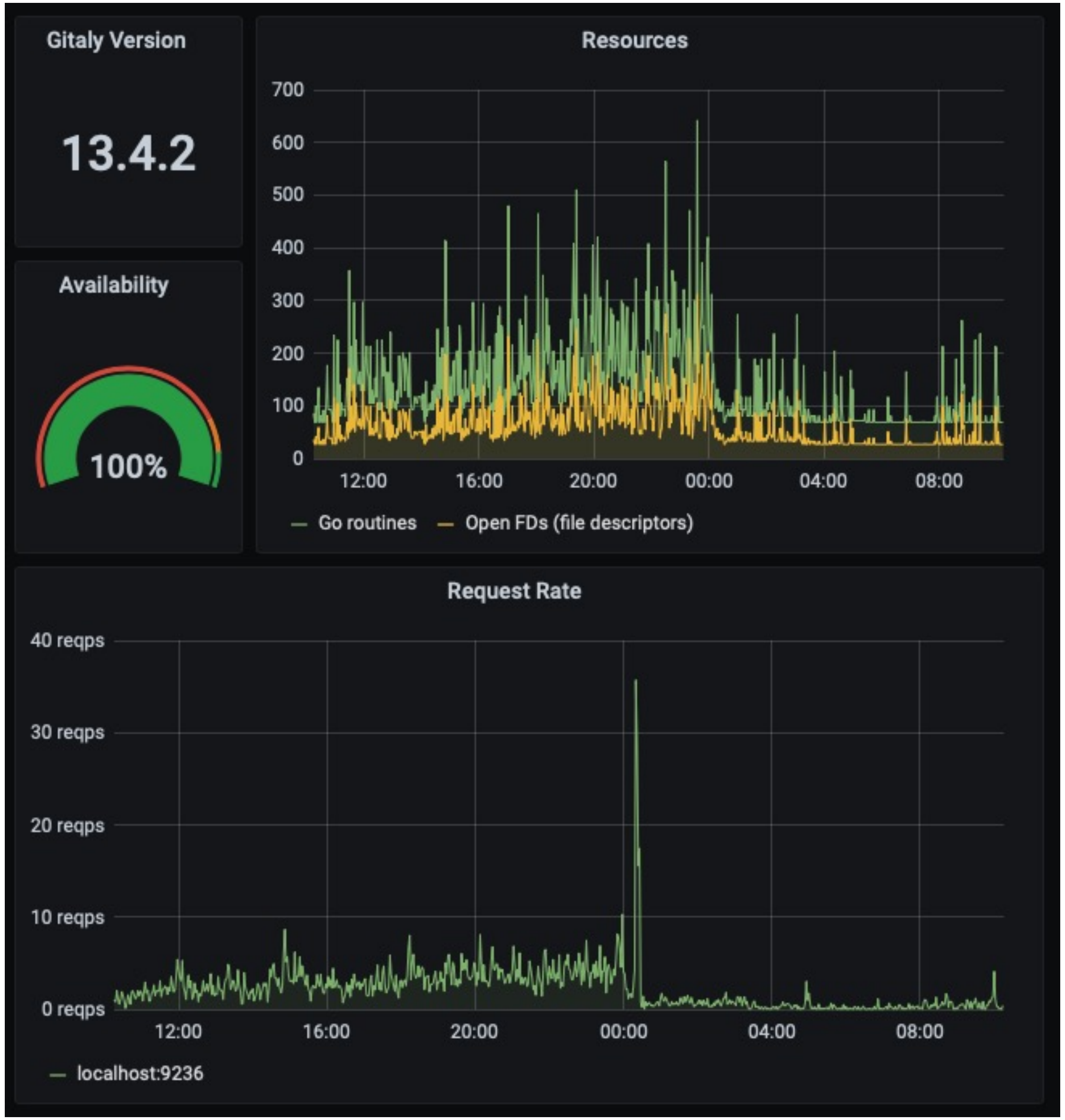
Graphs

Prof. Darrell Long

CSE 13S

Is this a graph?

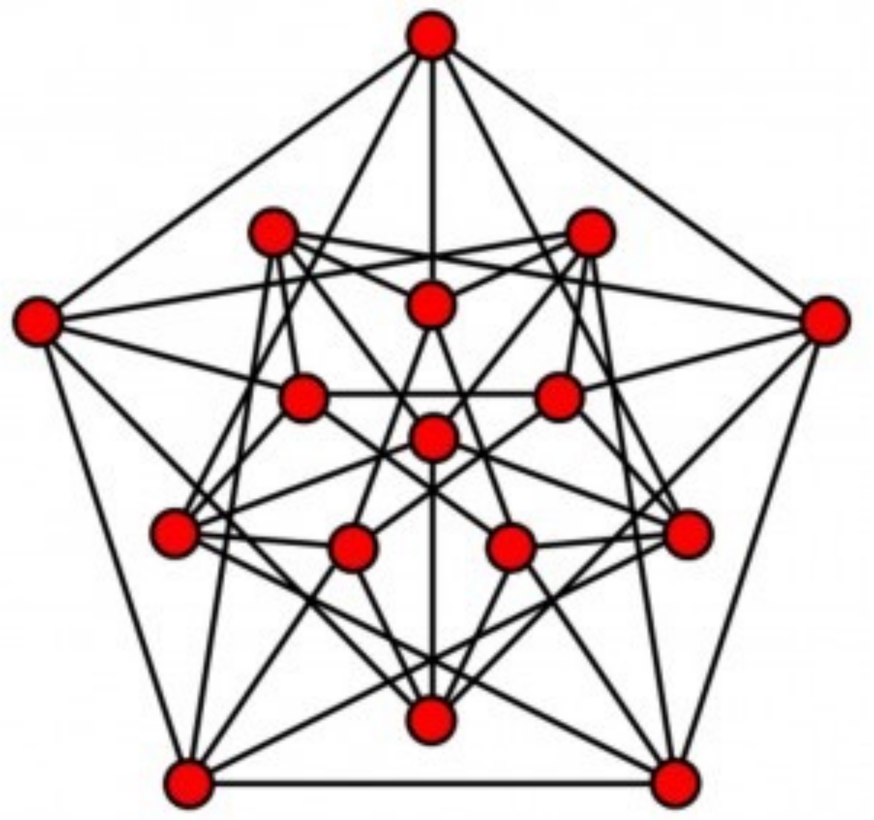
- Yes, this is commonly called a graph, but it is not what we are talking about here…
- We will call these depictions of numerical quantities *plots*.

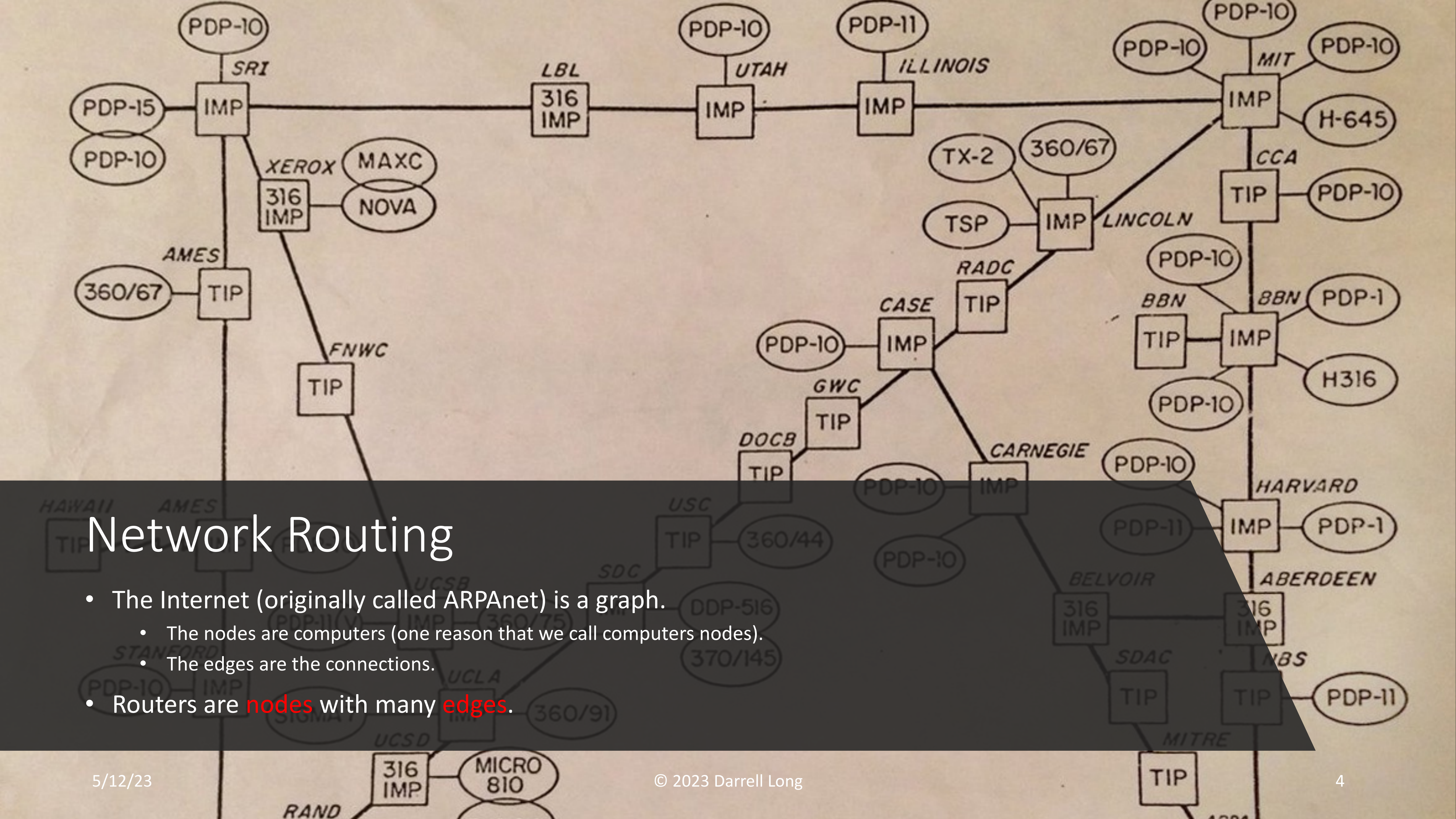


Is this a graph?

- This is an example of what we will call a *graph*.
- There is an entire branch of mathematics dedicated to *graph theory*.
- Graphs of this type are used in mathematics, computer science, physics, and even sociology.

Vertices and Edges

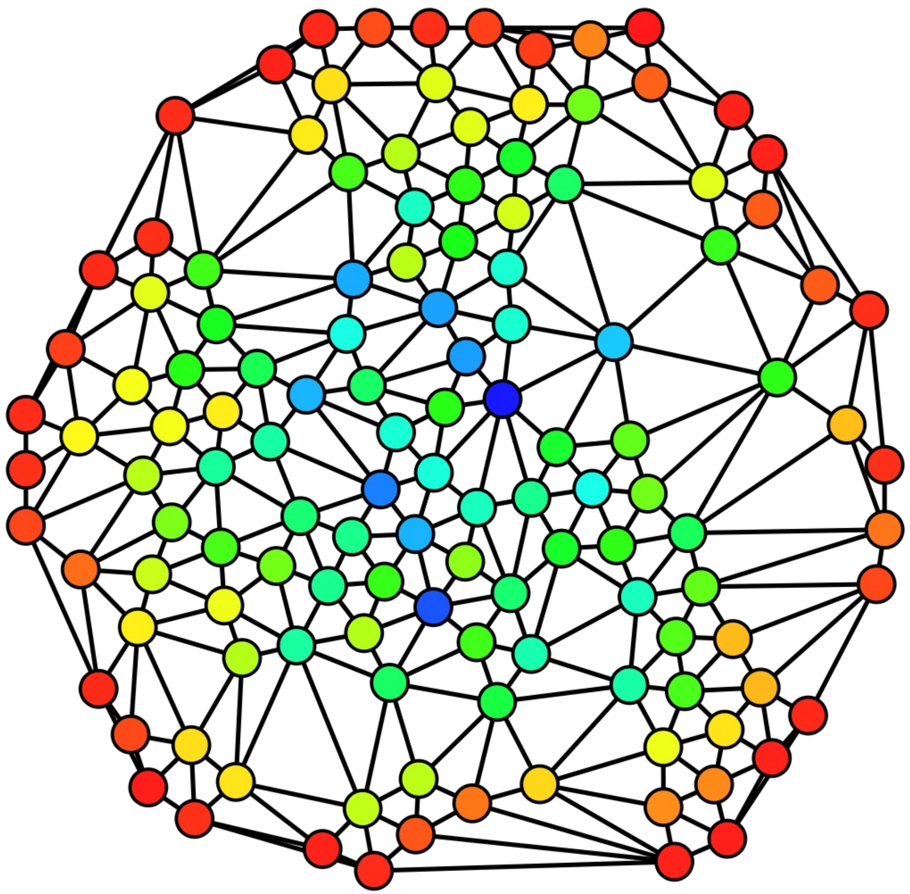




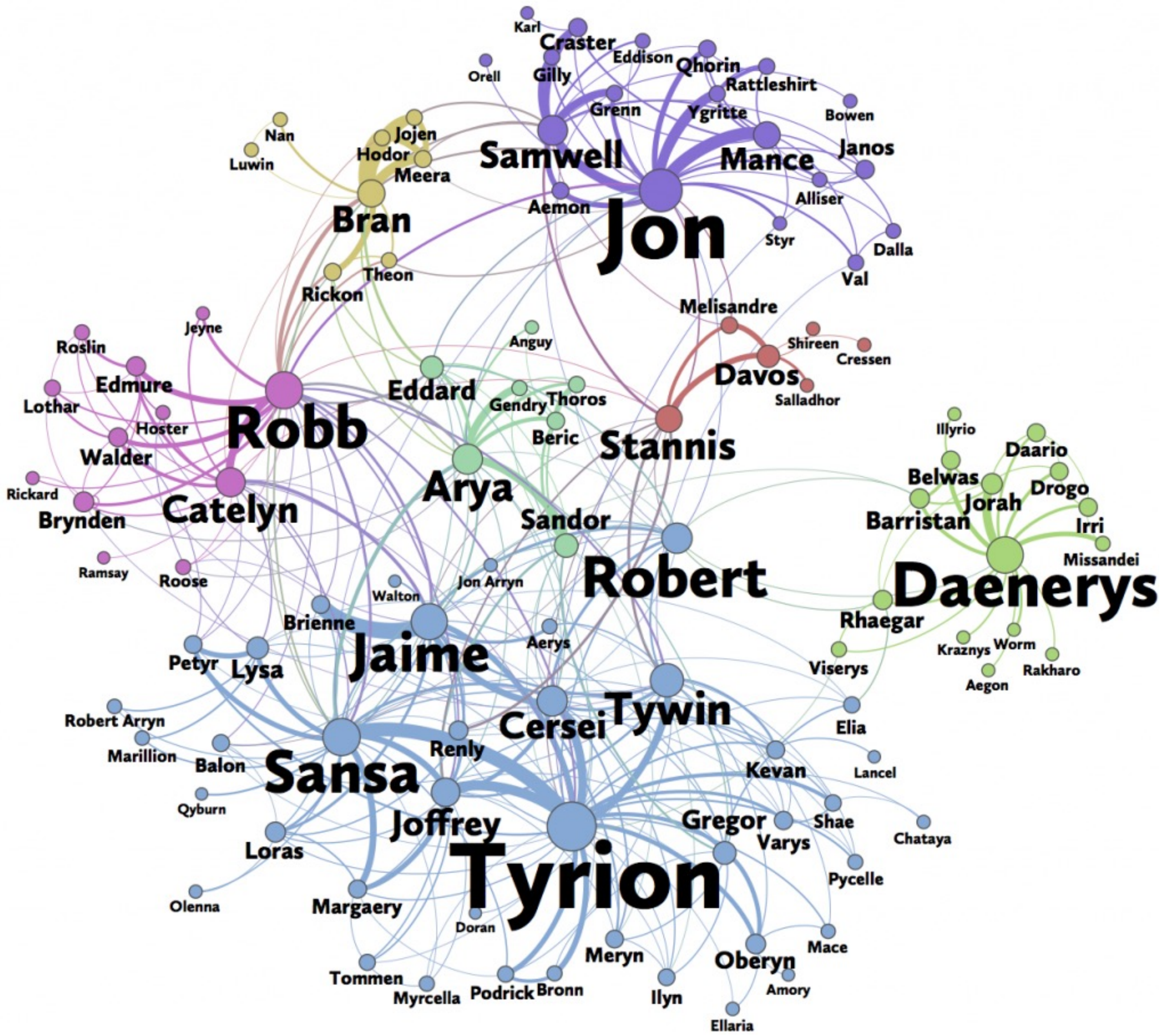
Network Routing

- The Internet (originally called ARPAnet) is a graph.
  - The nodes are computers (one reason that we call computers nodes).
  - The edges are the connections.
- Routers are nodes with many edges.

Social Networks



Network of Throne



A Formal Definition

- We formally define a graph G as follows:
  - $G = < V, E >$
  - A graph is defined by its ***vertices*** and ***edges***.

- $\boldsymbol{V}$ is the set of vertices.
  - $V = \{v_1, v_2, \cdots, v_n\}$

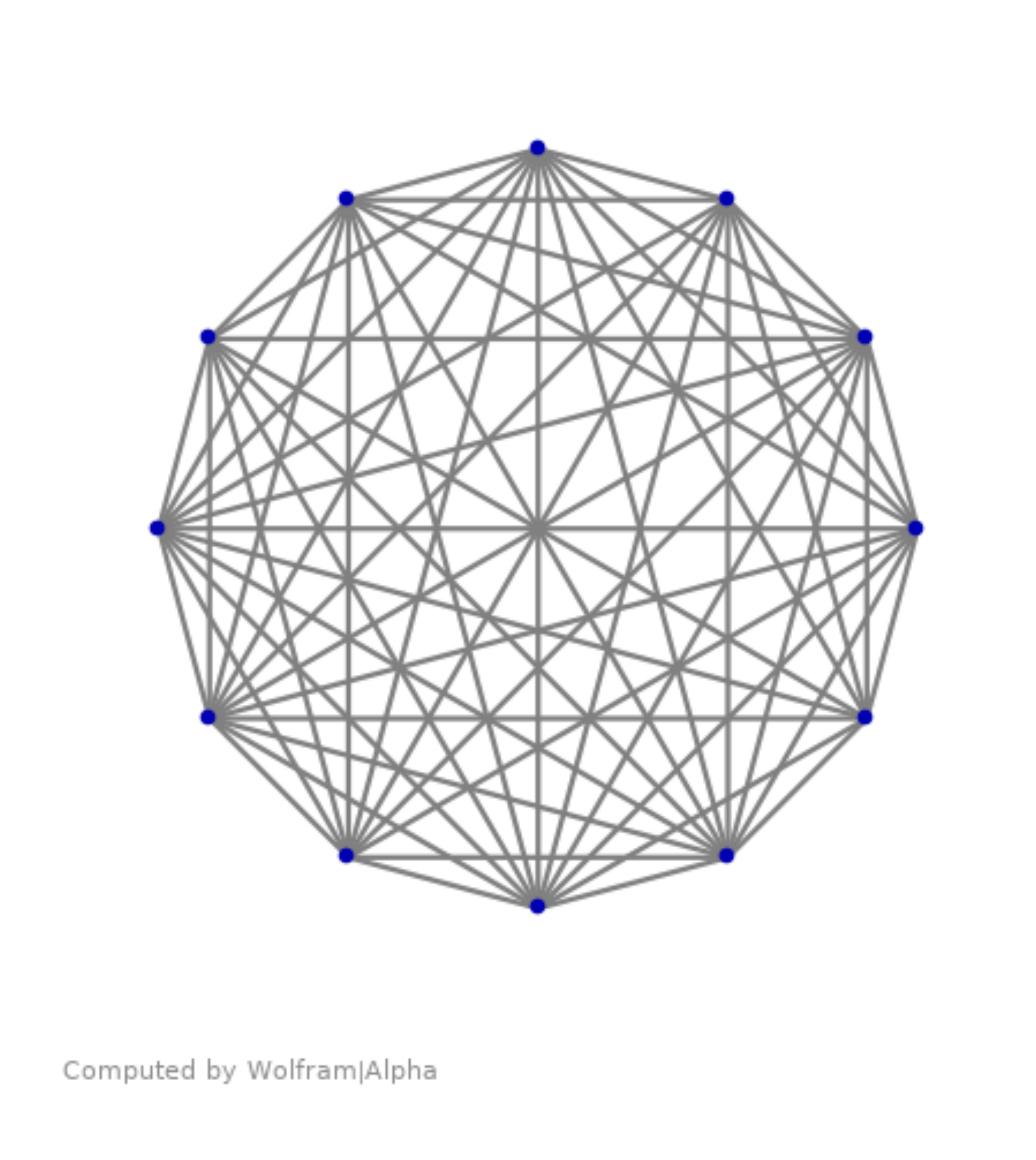
- $\boldsymbol{E}$ is the set of edges.
  - Each edge is a tuple of vertices.
  - $E = \{< v_i, v_j >, < v_p, v_q >, \cdots, < v_s, v_t >\}$

Directed and Undirected Graphs

- Edges may have a direction, $n_1 \rightarrow n_2$, and we call that a *directed graph*.
- Edges may have no direction (or both directions), $n_1 \leftrightarrow n_2$, and we call that an *undirected graph*.
- The edges may have *weights*, which represent capacity, strength, or cost.

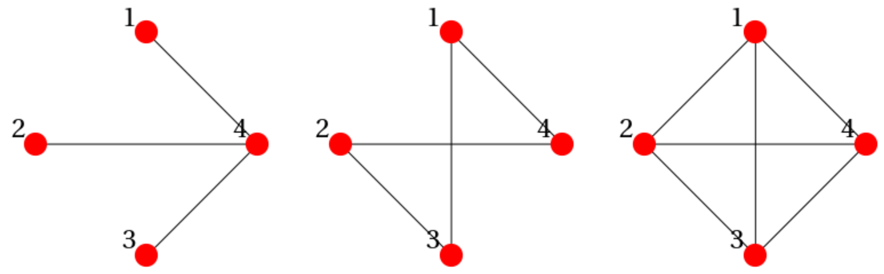
Representing a graph

- Adjacency Matrix
  - $n \times n$ matrix
  - Binary: edges present or absent
  - Weighted: $n \neq 0$
- Adjacency List
  - Column array for nodes
  - Linked list of edges from each node
  - May contain weights



Adjacency Matrix (Used in Assignment 4)

- A non-zero entry in $M_{i,j}$ means there is an edge $n_i \to n_j$.
- A matrix that is symmetric around the diagonal represents an *undirected* graph.
- The entry can specify not only the existence of an edge, but also its weight.
- Requires $O(n^2)$ space.
  - Sparse matrix techniques can improve it.



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Computed by Wolfram|Alpha

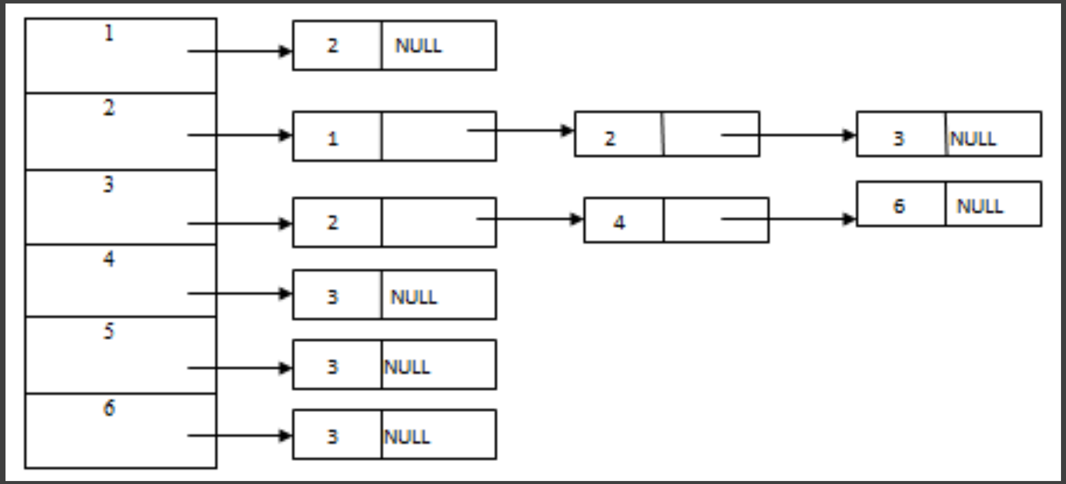
A graph in C

- Uses dynamically sized adjacency matrix.
- Adding an edge is O(1).
- Checking for the existence of an edge is also O(1).

```c
typedef struct {
    uint32_t vertices; // Number of vertices in graph.
    uint32_t **matrix; // Adjacency matrix.
} Graph;

// Constructor for a graph with dynamically sized adjacency matrix.
// Performs no error checking with memory allocation.
Graph *graph_create(uint32_t vertices) {
    Graph *g = (Graph *)malloc(sizeof(Graph));
    g->vertices = vertices;
    g->matrix = (uint32_t **)calloc(vertices, sizeof(uint32_t *));
    for (uint32_t i = 0; i < vertices; i += 1) {
        g->matrix[i] = (uint32_t *)calloc(vertices, sizeof(uint32_t));
    }
    return g;
}

// Adds an edge <i, j> of weight k.
// Does not check if vertices are valid.
void graph_add_edge(Graph *g, uint32_t i, uint32_t j, uint32_t k) {
    g->matrix[i][j] = k;
}

// Checks if edge <i, j> exists.
// Does not check if vertices are valid.
bool graph_has_edge(Graph *g, uint32_t i, uint32_t j) {
    return g->matrix[i][j] > 0;
}
```

Adjacency List

- Each node is represented as an entry in a column vector.
    - Each entry is the head of a linked list.
- The list elements contain:
    - The destination node, and
    - The *weight* of the edge.
- Why would you prefer this over an adjacency matrix?
    - An adjacency matrix is $O(n^2)$ space,
    - An adjacency list will be more space efficient for sparse graphs.



Another graph in C

- Uses adjacency lists.
  - Implemented using *linked lists*.
- Adding an edge is O(1).
- Checking for the existence of an edge is O(n).
  - Must traverse entire list!

```c
#include "ll.h" // Linked list ADT interface.

typedef struct {
    uint32_t vertices;  // Number of vertices in graph.
    LinkedList **lists; // Adjacency lists.
} Graph;

// Constructor for a graph using adjacency lists.
// Performs no error checking with memory allocation.
Graph *graph_create(uint32_t vertices) {
    Graph *g = (Graph *)malloc(sizeof(Graph));
    g->vertices = vertices;
    g->lists = (LinkedList **)calloc(vertices, sizeof(LinkedList *));
    for (uint32_t i = 0; i < vertices; i += 1) {
        g->lists[i] = ll_create(); // Initialize each to be empty adjacency list.
    }
    return g;
}

// Adds an edge <i, j> of weight k.
// Does not check if vertices are valid.
void graph_add_edge(Graph *g, uint32_t i, uint32_t j, uint32_t k) {
    // Inserts a new node containing j as the vertex and k as the weight.
    ll_insert(g->lists[i], j, k);
}

// Checks if edge <i, j> exists.
// Does not check if vertices are valid.
bool graph_has_edge(Graph *g, uint32_t i, uint32_t j) {
    // Searches the list, looking for a node that contains j as the vertex.
    return ll_find(g->lists[i], j);
}
```

Basic Graph Algorithms

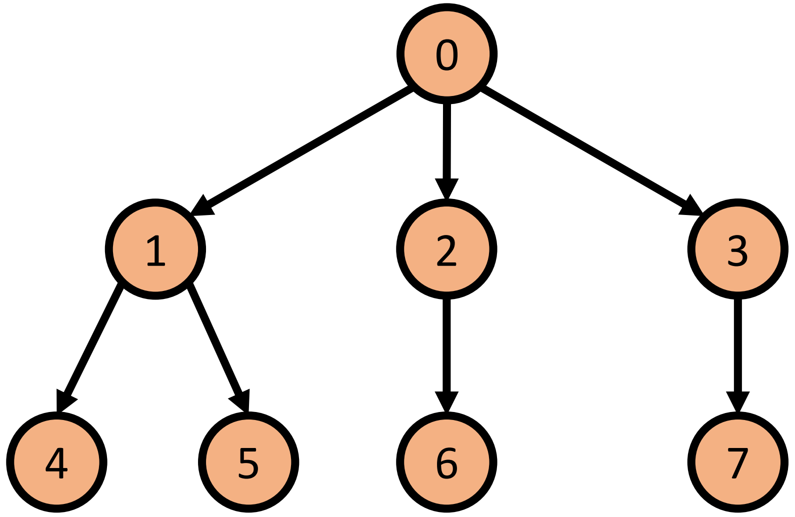
- We often want to find something in a graph.
- Two ways of searching a graph:
  1. Breadth-first search
     - Uses a queue.
     - Explore the set of vertices immediately reachable.
       - Then repeat the process for each vertex in the set.
     - Also known as "level-order" traversal.
  2. Depth-first search
     - Uses recursion or a stack.
     - Search as far as possible before backing up.
     - We will showcase *iterative* DFS using a stack.

Depth-First Search

- With a Tree
  - Basic recursive traversal
  - Visit all of the nodes
- With a Graph
  - Can be recursive
  - But need to mark which nodes have been visited
    - The recursive algorithm's "Base Case"

Depth-First-Search (DFS)

```
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                  // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

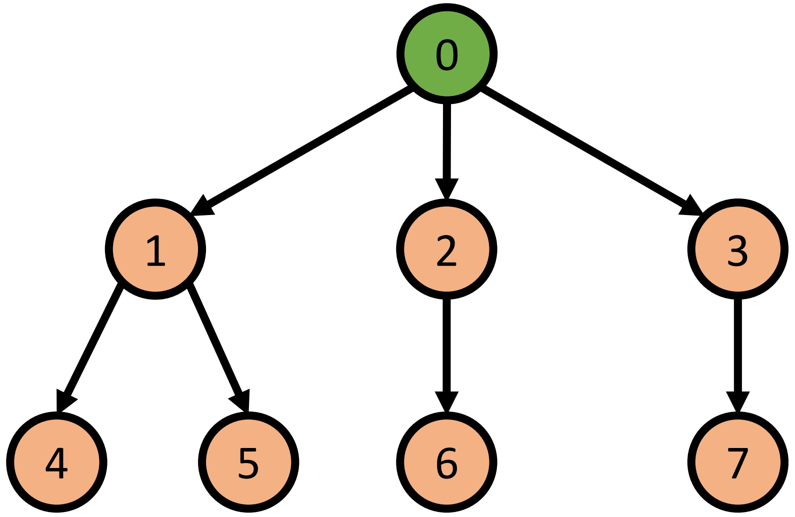


Stack: 0

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                   // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

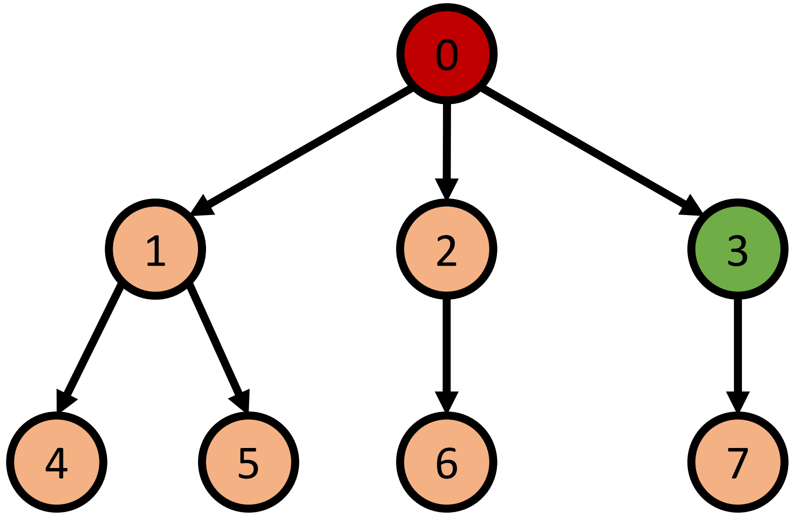


Stack: 1, 2, 3

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```c
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                   // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

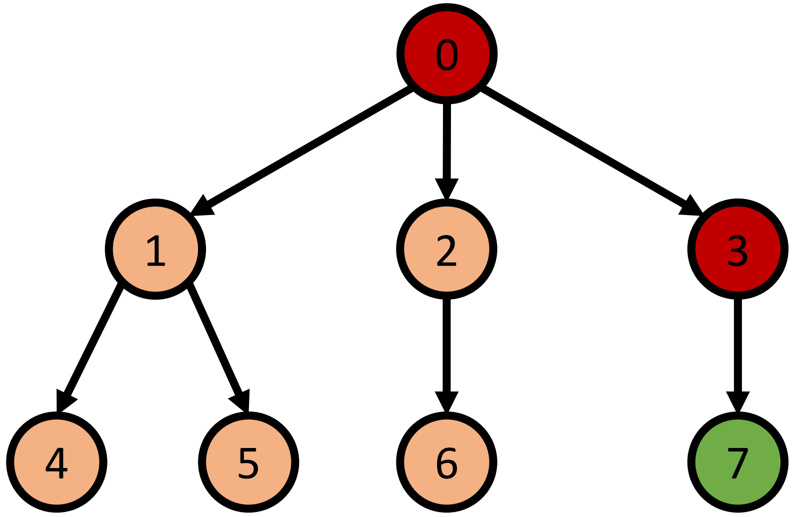


Stack: 1, 2, 7

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```c
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

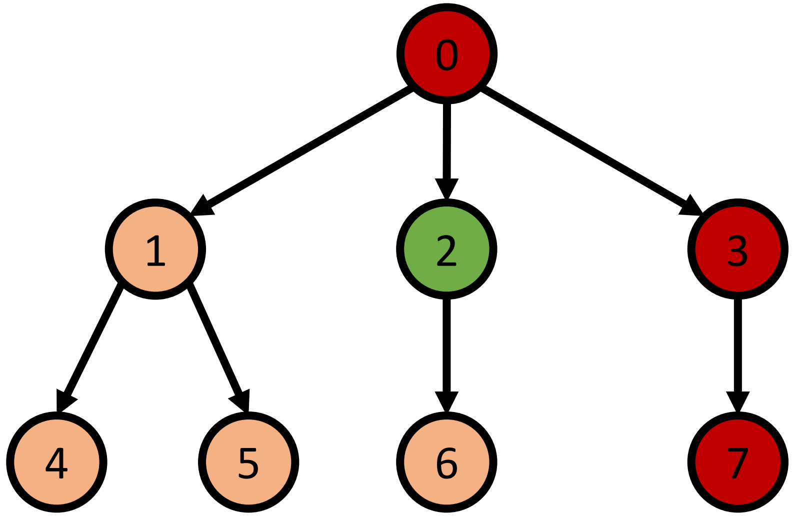


Stack: 1, 2

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

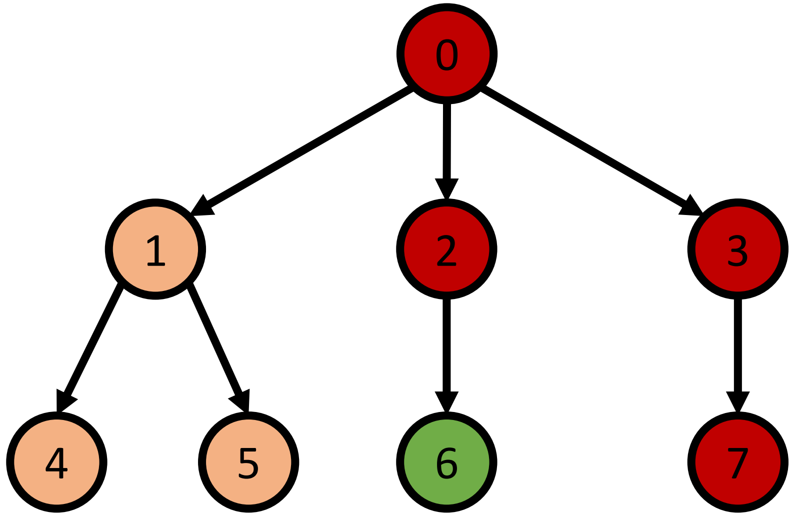


Stack: 1, 6

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                     // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

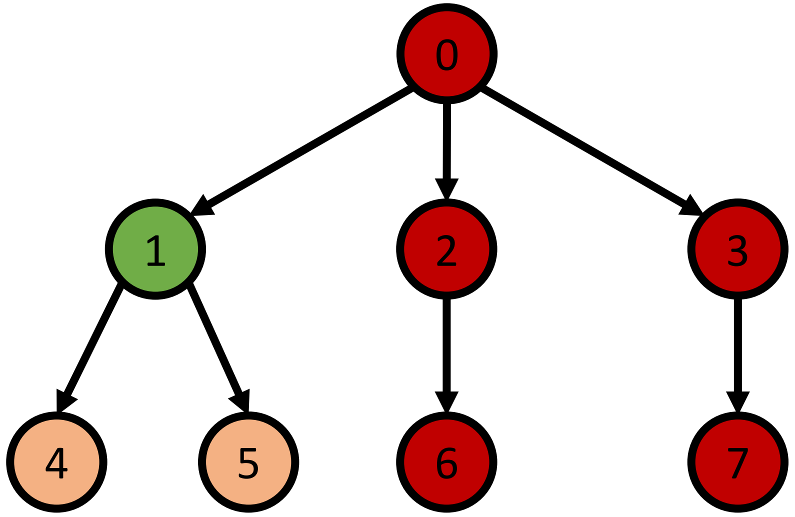


Stack: 1

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

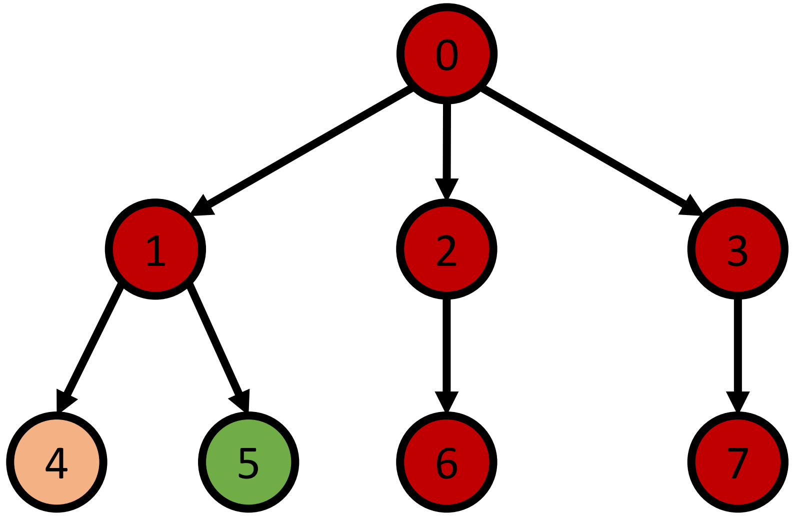


Stack: 4, 5

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

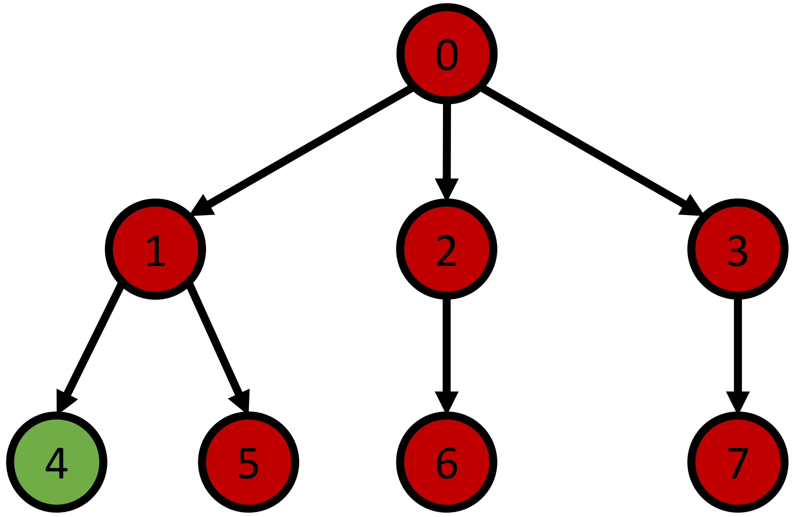


Stack: 4

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```c
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

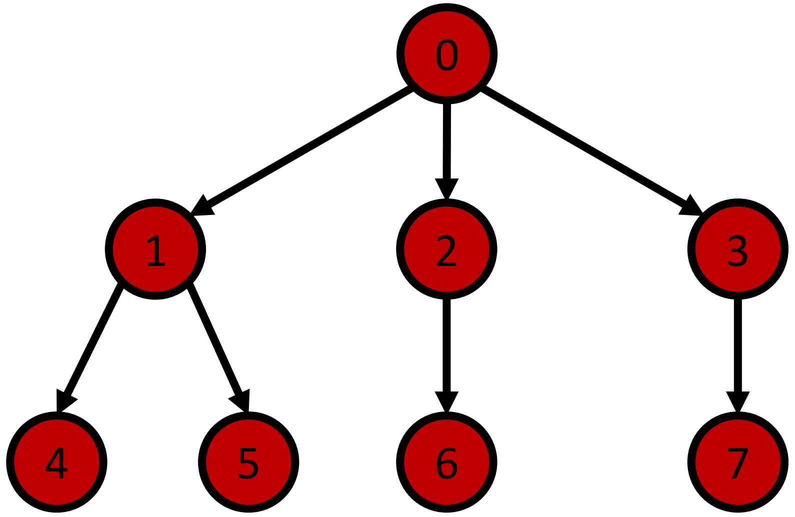


Stack: 4

Top of stack will be the rightmost element.

Depth-First-Search (DFS)

```
void dfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Track where we've been.
    Stack *s = stack_create(graph_vertices(g)); // Capacity of stack is number of vertices.
    stack_push(s, v);

    while (!stack_empty(s)) {
        stack_pop(s, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                stack_push(s, u);
            }
        }
    }

}
```

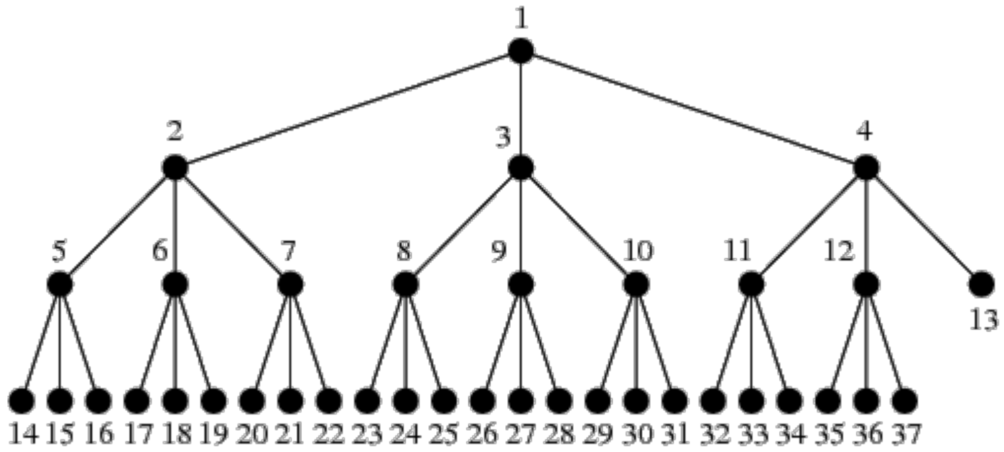


Stack:

Top of stack will be the rightmost element.

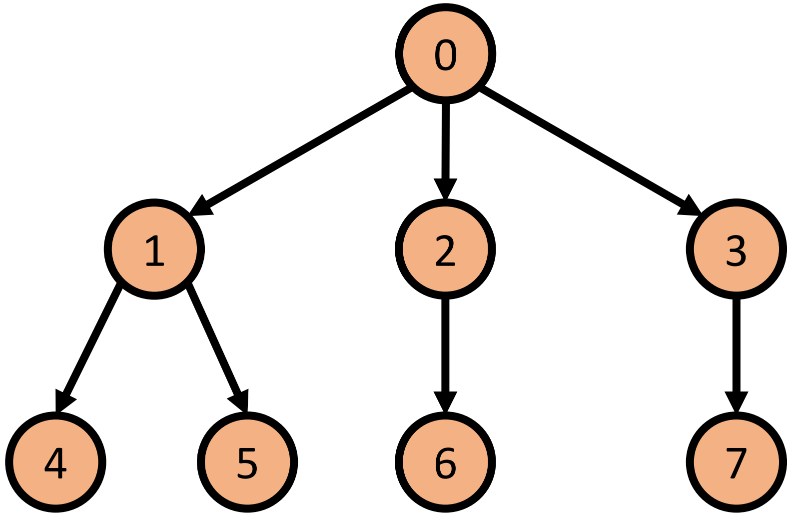
Trees

- Trees are form of *acyclic* graph.
    - Acyclic means that if you follow the edges, there are *no loops* (cycles).
- You will often hear the term *DAG*, which stands for:
    - Directed
    - Acyclic
    - Graph



Breadth-First-Search (BFS)

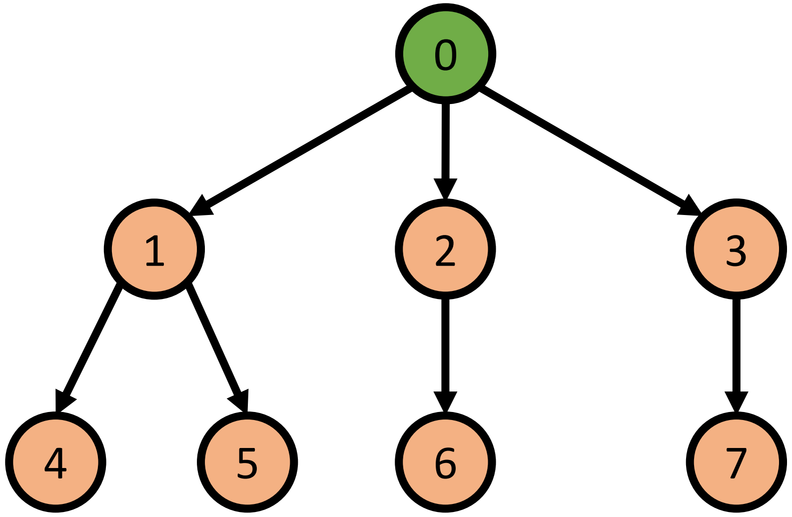
```c
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 0

Breadth-First-Search (BFS)

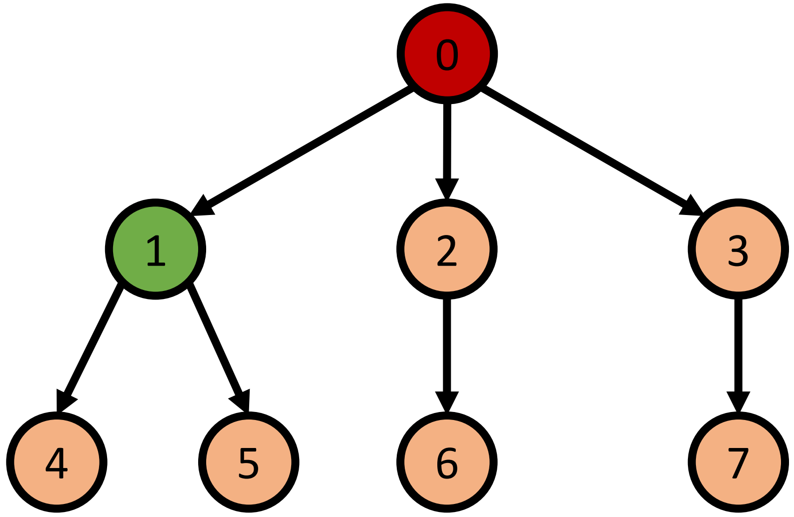
```
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                        // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 1, 2, 3

Breadth-First-Search (BFS)

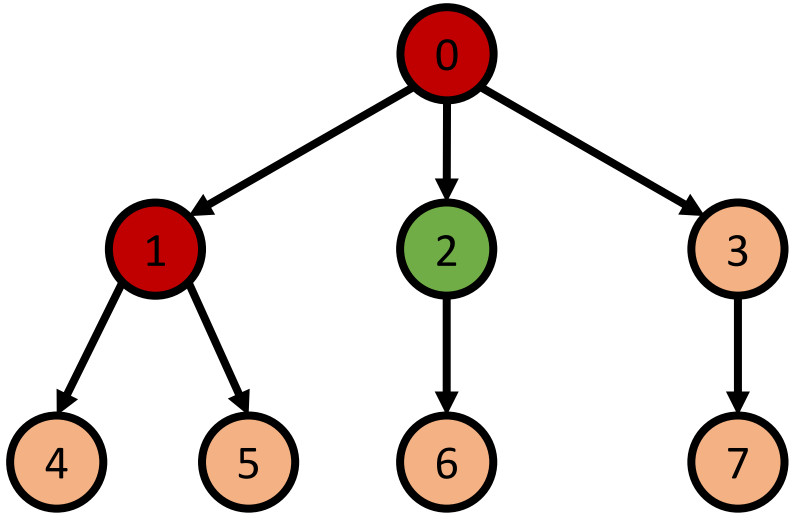
```c
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 2, 3, 4, 5

Breadth-First-Search (BFS)

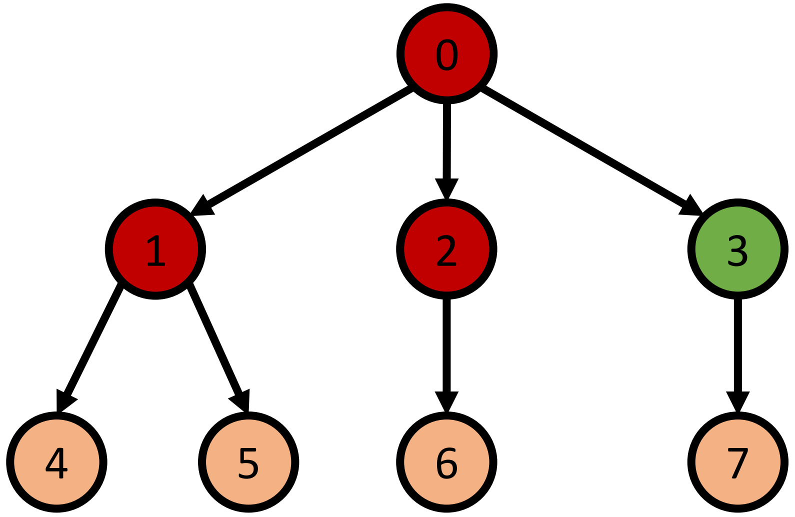
```c
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                   // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 3, 4, 5, 6

Breadth-First-Search (BFS)

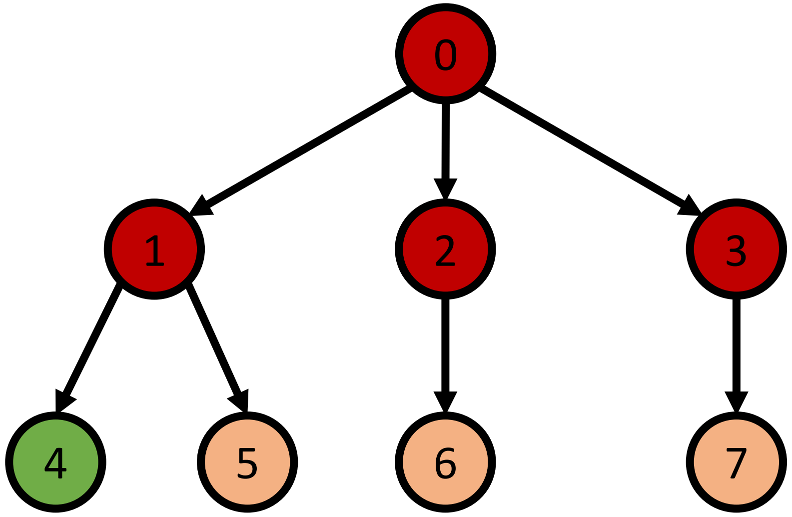
```c
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 4, 5, 6, 7

Breadth-First-Search (BFS)

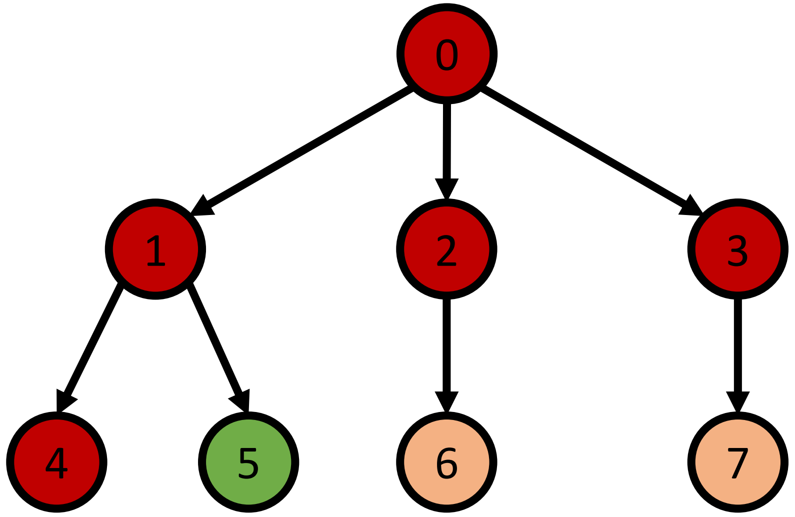
```c
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 5, 6, 7

Breadth-First-Search (BFS)

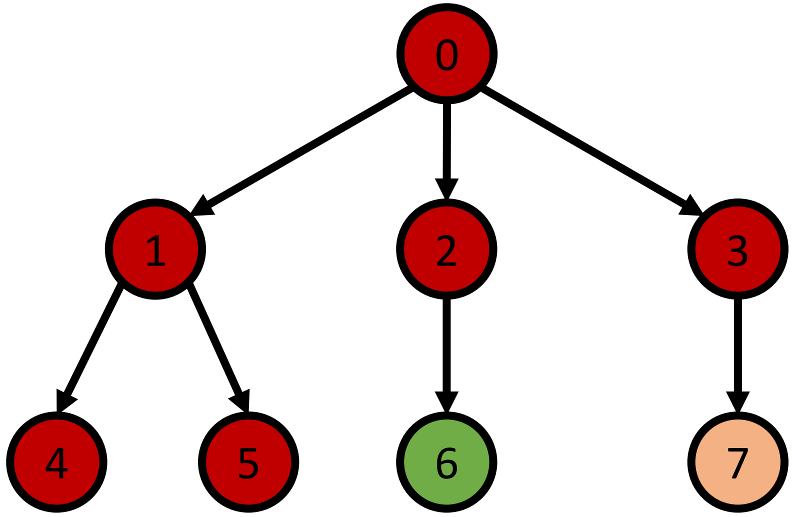
```
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 6, 7

Breadth-First-Search (BFS)

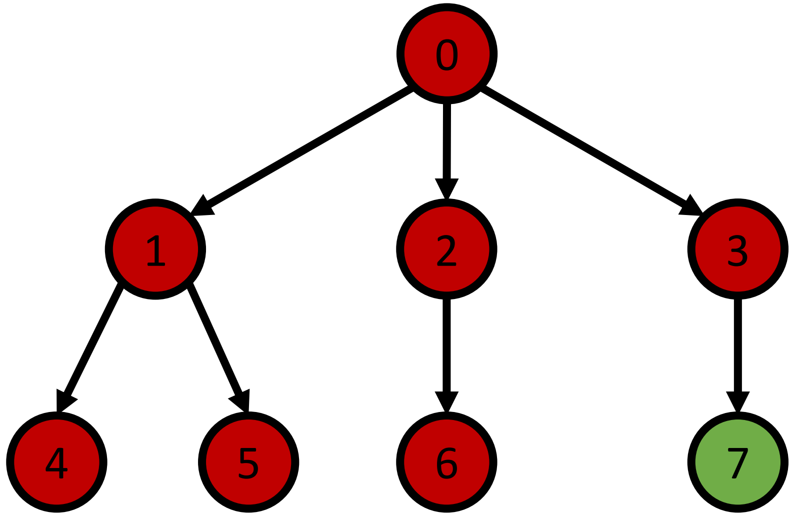
```c
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                     // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 7

Breadth-First-Search (BFS)

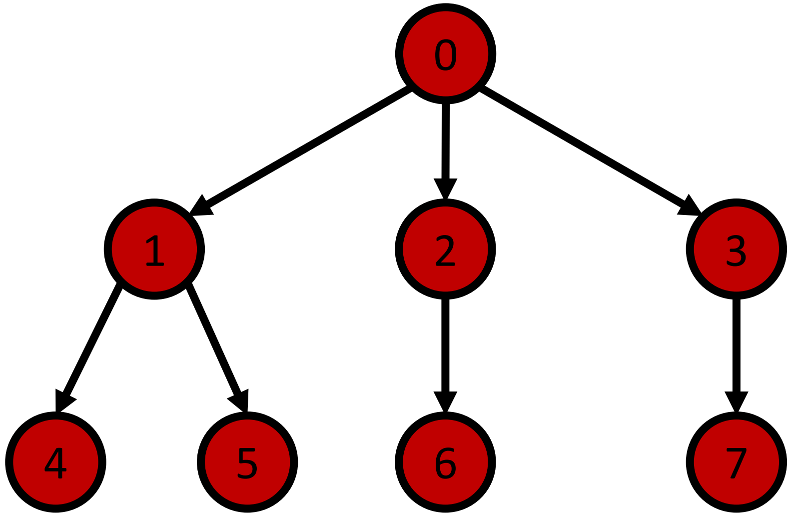
```
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                      // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue: 7

Breadth-First-Search (BFS)

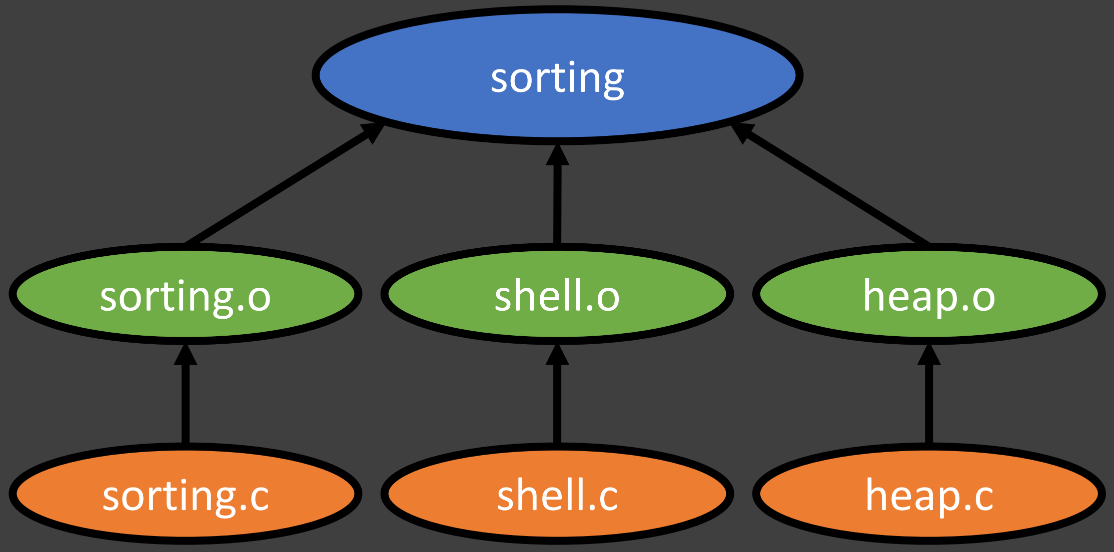
```
void bfs(Graph *g, uint32_t v) {
    Set visited = set_empty();                    // Capacity of queue is number of vertices.
    Queue *q = queue_create(graph_vertices(g)); // Capacity of queue is number of vertices.
    enqueue(q, v);

    while (!queue_empty(q)) {
        dequeue(q, &v);
        visited = set_insert(visited, v);

        for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
            if (!set_member(visited, u) && graph_has_edge(g, v, u)) {
                enqueue(q, u);
            }
        }
    }
}
```



Queue:

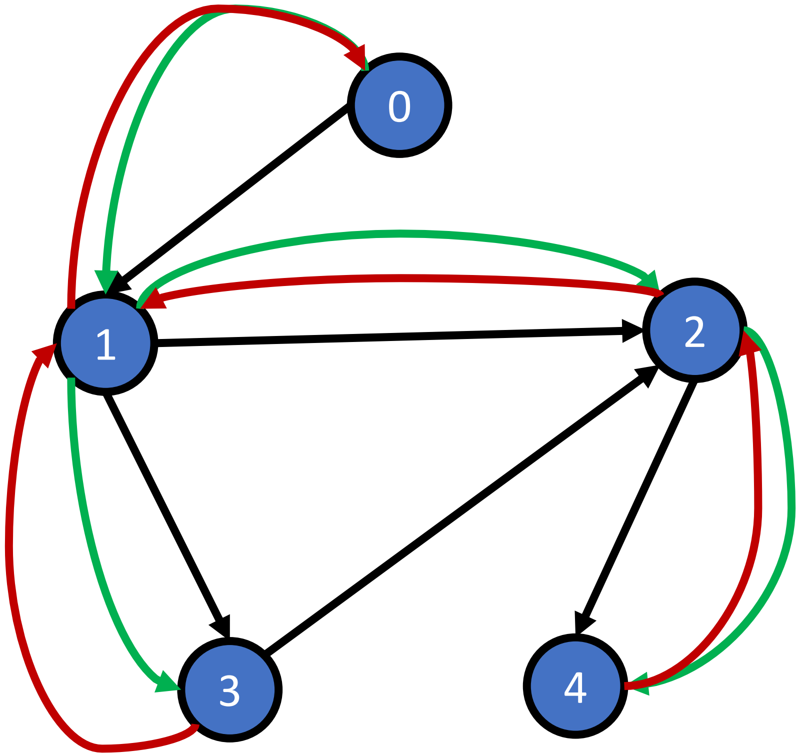
Topological Sort

- Linearly orders vertices in a DAG such that, for any directed edge <u,v>, u appears before v.
- Typically used to indicate ordering of dependencies.
  - Such as in a Makefile!
- Implemented using DFS:
  - Whenever a vertex is finished, it is prepended to a list.
  - Finished vertex: no more exploration possible from that vertex.
- Topological orderings are *not* unique.
  - Could be more than one valid topological ordering.



Modified DFS to Topologically Sort (Recursive)

```c
void toposort(Graph *g, uint32_t v, Set *visited, LinkedList *order) {
    *visited = set_insert(*visited, v);

    for (uint32_t u = 0; u < graph_vertices(g); u += 1) {
        if (!set_member(*visited, u) && graph_has_edge(g, v, u)) {
            toposort(g, u, visited, order);
        }
    }

    ll_insert(order, v);
}

int main(void) {
    Graph *g = graph_create(8);
    graph_add_edge(g, 0, 1, 5); // <0, 1>, weight = 5
    graph_add_edge(g, 1, 2, 2); // <0, 1>, weight = 2
    graph_add_edge(g, 1, 3, 2); // <0, 1>, weight = 2
    graph_add_edge(g, 3, 2, 3); // <0, 1>, weight = 3
    graph_add_edge(g, 2, 4, 4); // <0, 1>, weight = 4

    Set visited = set_empty();
    LinkedList *order = ll_create();
    toposort(g, 0, &visited, order);
    return 0;
}
```



Order: 0 → 1 → 3 → 2 → 4

Kahn's Algorithm

Input: A directed acyclic graph $G = \langle V, E \rangle$.
Output: A list L containing a topological ordering of G.

```
KAHN(G)
L ← []      // List storing topological ordering.
S ← {}      // Set of vertices with no incoming edges.
for v ∈ V
    if v has no incoming edges
        S ← S ∪ {v}
while |S| > 0
    remove some v ∈ S
    L ← L + [v]
    for each u ∈ V where e is <v,u> and e ∈ E
        E ← E − {e}
        if u has no incoming edges
            S ← S ∪ {u}
return L
```

Single-Source Shortest Paths (SSSP)

- Assume some graph $G\ =< V, E >$ and source vertex $s \in V$.
  - We want the shortest path from s to any $v \in V$.
- SSSP algorithms:
  - Bellman-Ford
  - Dijkstra's
    - We will focus on Dijkstra's algorithm.

Dijkstra's Algorithm

Input: A directed graph $G = < V, E >$ and source vertex $s \in V$.
Output:

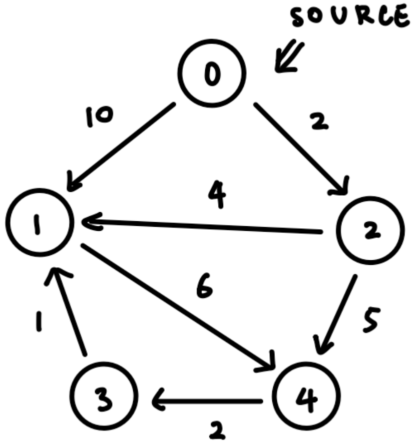
1. Distances from s for each vertex $v \in V$.
2. Predecessors for each $v \in V$ in shortest paths from s.

```
DIJKSTRA(G, s)
S ← {s}
for v ∈ G
      D[v] ← ∞        // Distance from source is infinite.
      P[v] ← −1       // Predecessor of vertex is unknown.
D[s] ← 0              // Distance from s to s.
while S ≠ V
      w ← vertex in V − S such that D[w] is minimum
      S ← S ∪ {w}
      for each v ∈ ADJACENT(G, w)
            t ← D[w] + WEIGHT(G, w, v)
            if t < D[v]           // If new distance from s to v is shorter.
                  D[v] ← t        // Update distance.
                  P[v] ← w        // Update predecessor.
return D, P
```

$$D = \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ [\infty, \infty, \infty, \infty, \infty] \end{array}$$

$$P = [-1, -1, -1, -1, -1]$$

$$S = \{ \}$$



Dijkstra’s Algorithm

Input: A directed graph $G = < V, E >$ and source vertex $s \in V$.
Output:

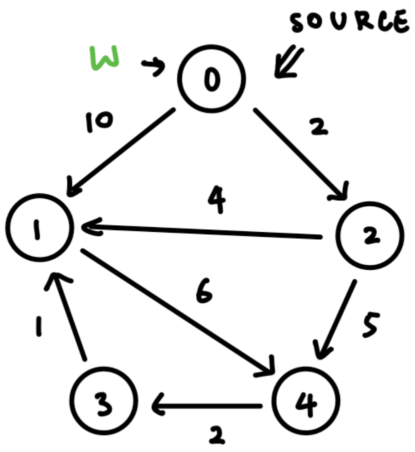
1. Distances from s for each vertex $v \in V$.
2. Predecessors for each $v \in V$ in shortest paths from s.

```
DIJKSTRA(G, s)
S ← {s}
for v ∈ G
    D[v] ← ∞        // Distance from source is infinite.
    P[v] ← −1       // Predecessor of vertex is unknown.
D[s] ← 0            // Distance from s to s.
while S ≠ V
    w ← vertex in V − S such that D[w] is minimum
    S ← S ∪ {w}
    for each v ∈ ADJACENT(G,w)
        t ← D[w] + WEIGHT(G,w,v)
        if t < D[v]          // If new distance from s to v is shorter.
            D[v] ← t         // Update distance.
            P[v] ← w         // Update predecessor.
return D, P
```

$$\begin{array}{ccccccc} & 0 & 1 & 2 & 3 & 4 & \\ D = [& 0, & 10, & 2, & \infty, & \infty &] \end{array}$$

$P = [-1, 0, 0, -1, -1]$

$S = \{0\}$



adjacent(G, 0) = { 1, 2 }

vertex 1

t = 0 + 10

is 10 < ∞ ? ✓ (update)

vertex 2

t = 0 + 2

is 2 < ∞ ? ✓ (update)

Dijkstra's Algorithm

Input: A directed graph $G = \langle V, E \rangle$ and source vertex $s \in V$.
Output:

1. Distances from s for each vertex $v \in V$.
2. Predecessors for each $v \in V$ in shortest paths from s.

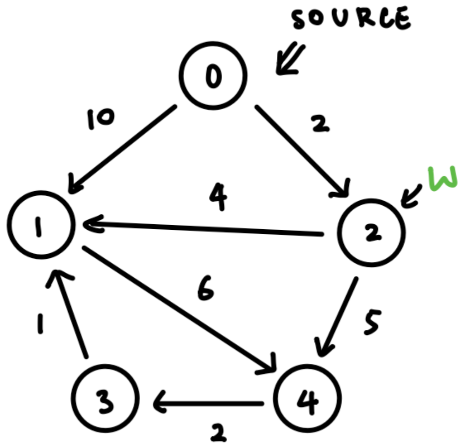
DIJKSTRA(G, s)

$S \leftarrow \{s\}$
for $v \in G$
    $D[v] \leftarrow \infty$ // Distance from source is infinite.
    $P[v] \leftarrow -1$ // Predecessor of vertex is unknown.
$D[s] \leftarrow 0$ // Distance from s to s.
while $S \neq V$
    $w \leftarrow$ vertex in $V - S$ such that $D[w]$ is minimum
    $S \leftarrow S \cup \{w\}$
    for each $v \in$ ADJACENT(G, w)
        $t \leftarrow D[w] +$ WEIGHT(G, w, v)
        if $t < D[v]$ // If new distance from s to v is shorter.
            $D[v] \leftarrow t$ // Update distance.
            $P[v] \leftarrow w$ // Update predecessor.
return D, P

$$\begin{array}{cccccc} & 0 & 1 & 2 & 3 & 4 \\ D = [& 0, & 6, & 2, & \infty, & 7 \;] \end{array}$$

$P = [-1, 2, 0, -1, 2]$

$S = \{0, 2\}$



adjacent$(G, 2) = \{1, 4\}$

vertex 1

$t = 2 + 4$

is $6 < 10$? ✓ (update)

vertex 4

$t = 2 + 5$

is $7 < \infty$? ✓ (update)

Dijkstra’s Algorithm

Input: A directed graph $G = < V, E >$ and source vertex $s \in V$.
Output:

1. Distances from s for each vertex $v \in V$.
2. Predecessors for each $v \in V$ in shortest paths from s.

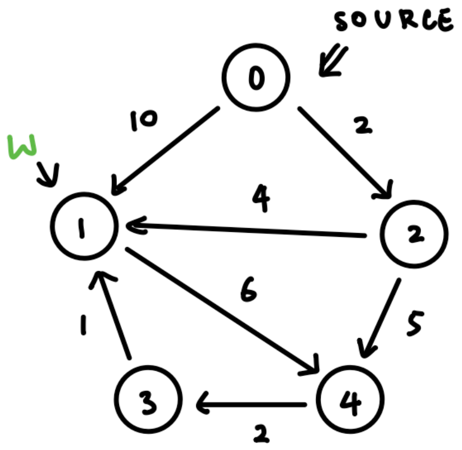
```
DIJKSTRA(G, s)
S ← {s}
for v ∈ G
    D[v] ← ∞        // Distance from source is infinite.
    P[v] ← −1       // Predecessor of vertex is unknown.
D[s] ← 0            // Distance from s to s.
while S ≠ V
    w ← vertex in V − S such that D[w] is minimum
    S ← S ∪ {w}
    for each v ∈ ADJACENT(G, w)
        t ← D[w] + WEIGHT(G, w, v)
        if t < D[v]           // If new distance from s to v is shorter.
            D[v] ← t          // Update distance.
            P[v] ← w          // Update predecessor.
return D, P
```

$$\begin{array}{cccccc} & 0 & 1 & 2 & 3 & 4 \\ D = [& 0, & 6, & 2, & \infty, & 7\ ] \end{array}$$

$P = [-1, 2, 0, -1, 2]$

$S = \{0, 2, 1\}$

adjacent(G, 1) = { }



Dijkstra’s Algorithm

Input: A directed graph $G = \langle V, E \rangle$ and source vertex $s \in V$.
Output:

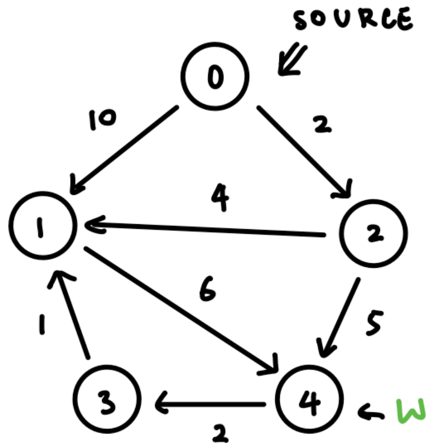
1. Distances from s for each vertex $v \in V$.
2. Predecessors for each $v \in V$ in shortest paths from s.

```
DIJKSTRA(G, s)
S ← {s}
for v ∈ G
    D[v] ← ∞          // Distance from source is infinite.
    P[v] ← −1         // Predecessor of vertex is unknown.
D[s] ← 0              // Distance from s to s.
while S ≠ V
    w ← vertex in V − S such that D[w] is minimum
    S ← S ∪ {w}
    for each v ∈ ADJACENT(G, w)
        t ← D[w] + WEIGHT(G, w, v)
        if t < D[v]           // If new distance from s to v is shorter.
            D[v] ← t          // Update distance.
            P[v] ← w          // Update predecessor.
return D, P
```

$$\begin{array}{cccccc} & 0 & 1 & 2 & 3 & 4 \\ D = & [0, & 6, & 2, & 9, & 7] \end{array}$$

$P = [-1, 2, 0, 4, 2]$

$S = \{0, 2, 1, 4\}$



adjacent(G, 4) = { 3 }

vertex 3

t = 7 + 2

is 9 < ∞ ? ✓ (update)

Dijkstra's Algorithm

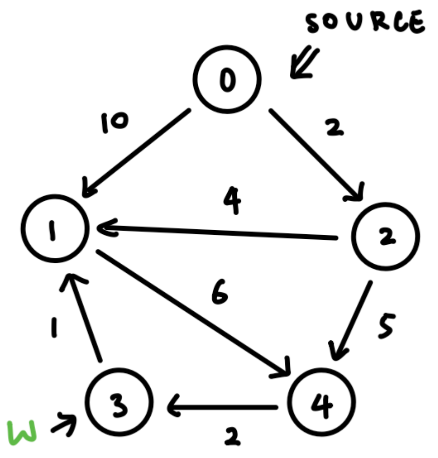
Input: A directed graph $G = \langle V, E \rangle$ and source vertex $s \in V$.
Output:

1. Distances from s for each vertex $v \in V$.
2. Predecessors for each $v \in V$ in shortest paths from s.

DIJKSTRA(G, s)

$S \leftarrow \{s\}$
for $v \in G$
  $D[v] \leftarrow \infty$ // Distance from source is infinite.
  $P[v] \leftarrow -1$ // Predecessor of vertex is unknown.
$D[s] \leftarrow 0$ // Distance from s to s.
while $S \neq V$
  $w \leftarrow$ vertex in $V - S$ such that $D[w]$ is minimum
  $S \leftarrow S \cup \{w\}$
  for each $v \in$ ADJACENT(G, w)
    $t \leftarrow D[w] + $ WEIGHT(G, w, v)
    if $t < D[v]$ // If new distance from s to v is shorter.
      $D[v] \leftarrow t$ // Update distance.
      $P[v] \leftarrow w$ // Update predecessor.
return D, P

$$\begin{array}{lllll} 0 & 1 & 2 & 3 & 4 \end{array}$$
$$D = [0, 6, 2, 9, 7]$$
$$P = [-1, 2, 0, 4, 2]$$
$$S = \{0, 2, 1, 4, 3\}$$



adjacent$(G, 3) = \{ 1 \}$

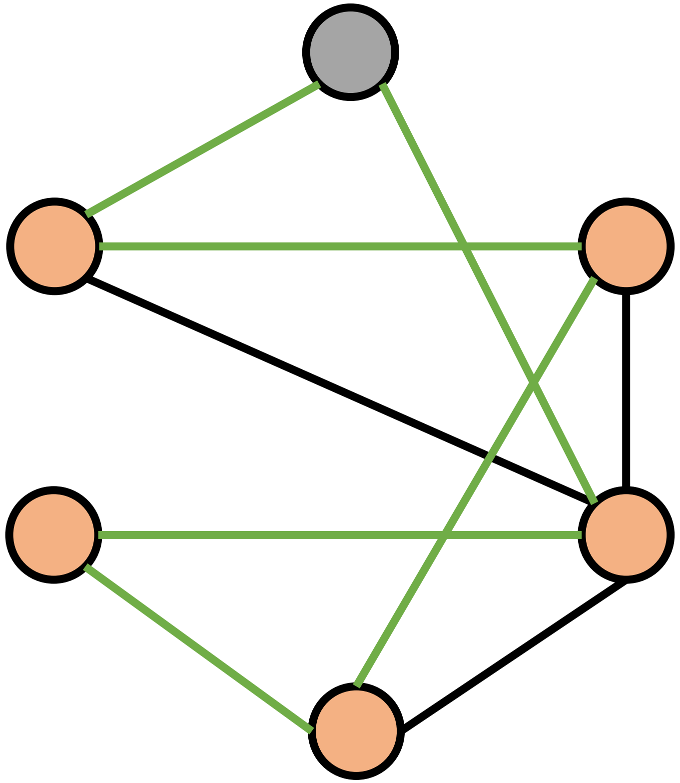
<u>vertex 1</u>

$t = 9 + 1$

is $10 < 6$? ✗ (no update)

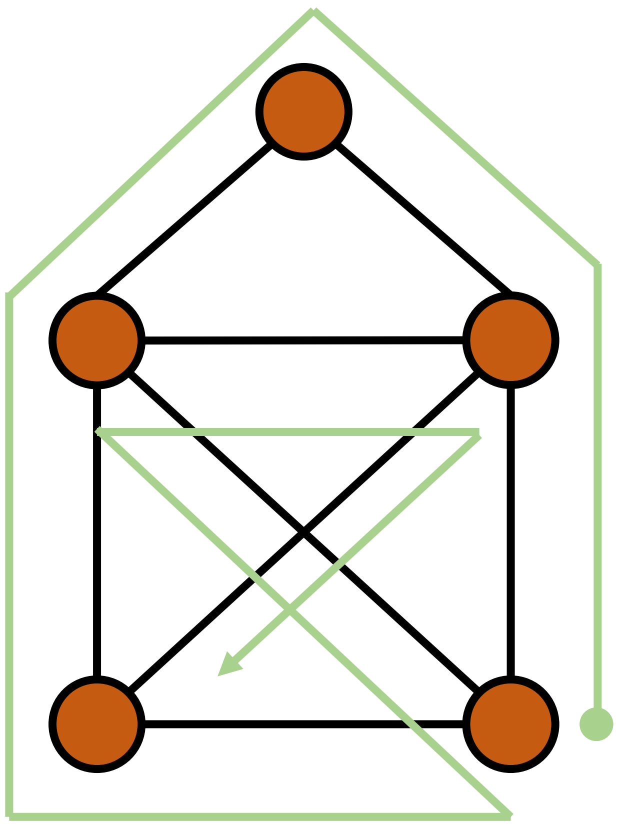
Hamiltonian Path

- A path in an undirected or directed graph that visits each *vertex* exactly once.
  - Must start from an origin vertex and end up back at the origin.



Eulerian Path

- A path in an undirected or directed graph that visits each *edge* exactly once.
    - Must start from an origin vertex and end up back at the origin.



Summary

- Graphs pervade computer science.
  - Shortest path finding
  - Graph coloring
  - Network flow
  - Dependency ordering
  - … and so much more!
- Come in undirected and directed forms.
- Used generally to indicate relationships between entities.
- Can be represented using either an adjacency matrix or using adjacency lists.