Divide-and-Conquer

In-Class Exercise: Finding a Local Minimum

distinct

- Suppose you are given an unsorted array of *N* integers

$A = [\quad]$ $O(N)$

- You want to find a local minimum: an element that is smaller than both of its neighbors (or if there is only one neighbor, then it only has to be smaller than that)

$[\quad]$

In-Class Exercise: Finding a Local Minimum

[y x z]

FLM(A):
  if mid(A) is a local min:
    return mid
  else:
    FLM(left half)
    FLM(right half)
    return either of those

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n^0)$$

$a, b = 2$

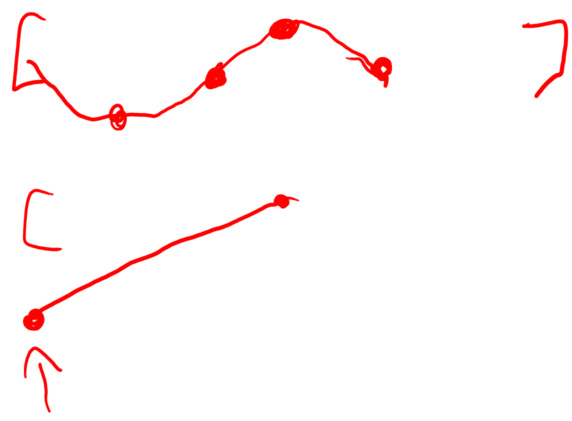
$d = 0$

$\log_b a = 1 > d$

$O(n^1)$

In-Class Exercise: Finding a Local Minimum

$A = [\quad y\ x\ z \quad]$



if $x < y, z$: return x

otherwise, $x > y$ or $x > z$

if $x > y$: recurse on left half

else if $x > z$: recurse on right half

$[1\ 4\ 5\ 2\ 3]$

$T(n) = 1 \cdot T(\frac{n}{2}) + O(n^0)$

$a = 1$ $b = 2$ $d = 0$

$\log_2 1 = 0 = d$

$O(\log n)$

Summary of Divide-and-Conquer

- Think about how to split a large problem into smaller problems
  - Sometimes you have to redefine the problem ~~(like we did for medians)~~
  - Sometimes you have to be creative about the input ~~(like for FFT)~~
- Sometimes you break the problem into (similar-sized pieces); sometimes you just remove (one element (like Merge in MergeSort)) → MM does not apply

MM often applies

reduce-and-conquer

Graphs

Map-Coloring

- Suppose you are trying to pick colors for a map
- If two countries are next to each other, they should have different colors
- How do you pick the colors?

Graphs

- A natural representation for this problem is a graph
- Graph data structure:
  - vertex
  - Nodes/Vertices (entities, individuals, places, things)
    - Let $V = \{v_1, v_2, \ldots\}$ be the set of vertices

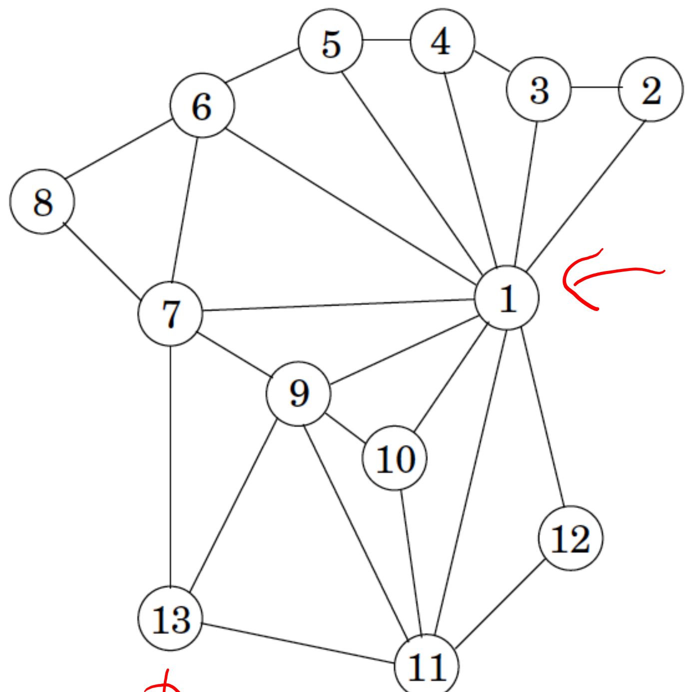
      N = # nodes
  - Edges/Links (relationships, friendships, connections)
    - Let $E = \{e_1, e_2, \ldots\}$ be the set of edges

      M = # edges
    - An edge is usually written as (v_1, v_2), where v_1 and v_2 are vertices
- Can be weighted: different edges have different strengths
- Can be directed: edges point in one direction

Graphs

Examples of vertices: 1, 8, 13

Examples of edges: (1, 10), (2, 3), (13, 9)

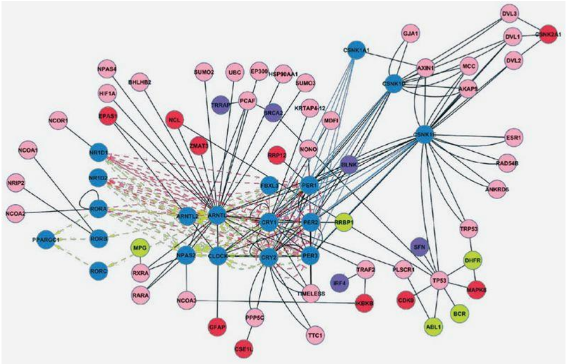


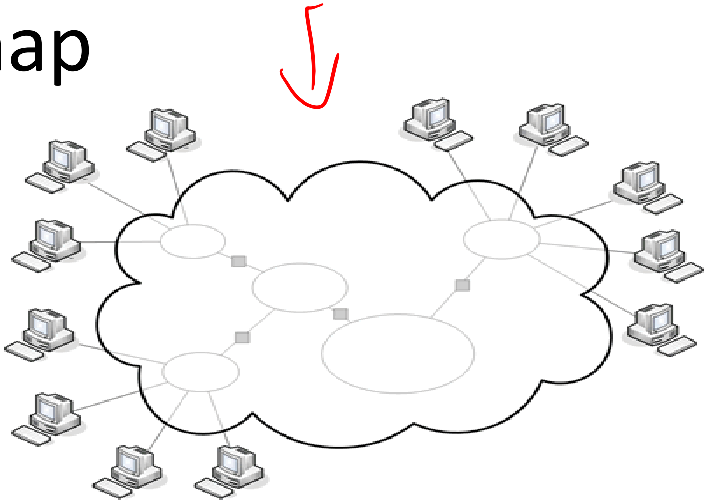
(1, 10) =

(10, 1)

Examples of Graphs

- A social network
- The World Wide Web
- Friendship network
- Co-worker relationships
- Road map





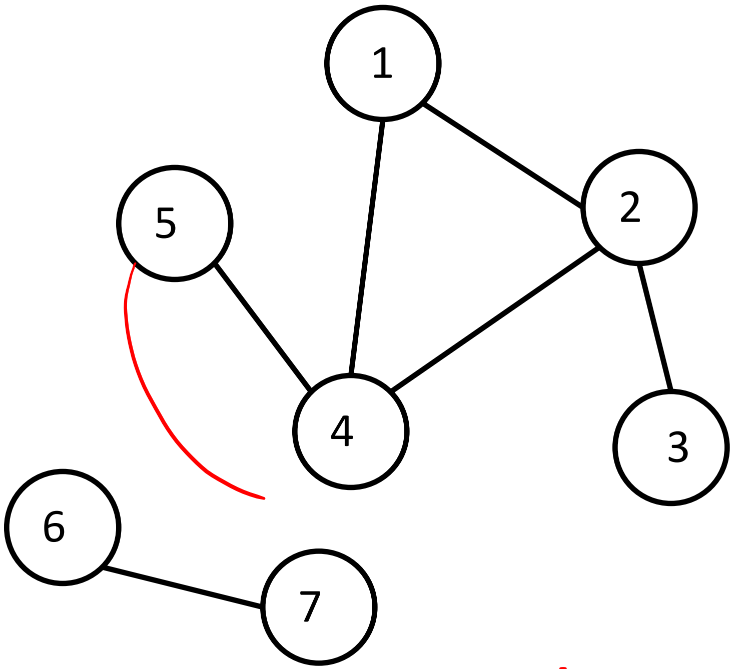
Graph Representation

adjacent (neighboring)

- How do we represent a graph?
- Adjacency matrix:
  - Suppose there are $n = |V|$ vertices
  - Create an n x n matrix A (adjacency)

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

Graph Representation



0	1	0	1	0	0	0
1	0	1	1	0	0	0
0	1	0	0	0	0	0
1	1	0	0	1	0	0
0	0	0	1	0	0	0
0	0	0	0	0	0	1
0	0	0	0	0	1	0

How many values need to be stored?

$O(n^2)$

$n^2/2$

Graph Representation

- How much space does the adjacency matrix take? $O(n^2)$

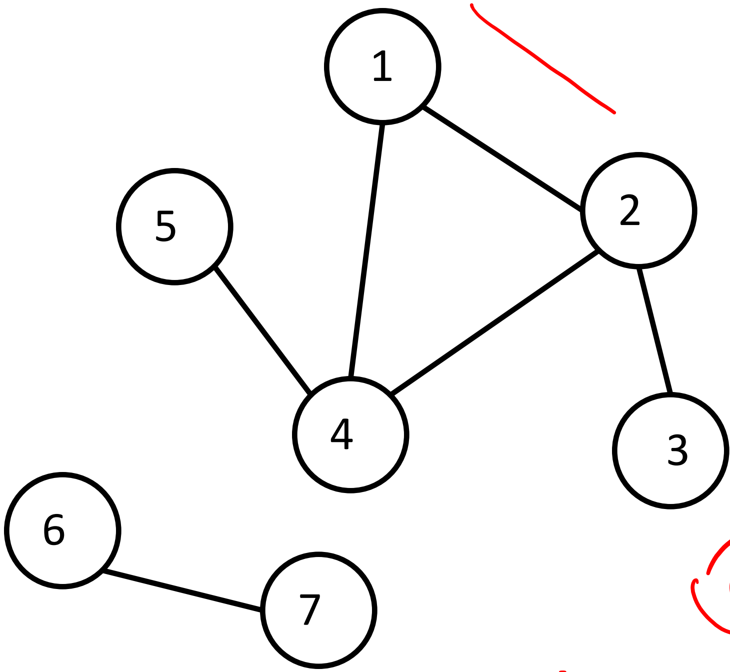
- Once created, how long does it take to check if an edge exists? $O(1)$

- Once created, how long does it take to add an edge? $O(1)$

Graph Representation

- Adjacency list
  - One linked list for each vertex
  - Each linked list stores the neighbors of that vertex

Graph Representation



8

9

10

$m << n$

1: 2 -> 4
2: 1 -> 3 -> 4 -> 4
3: 2
4: 1 -> 2 -> 5 -> 2
5: 4
6: 7
7: 6
8:
9:
10:

How many values need to be stored?

$O(n+m)$

nodes edges

Graph Representation

[ ] $\{1, 1, 3\} : \{1, 3\} = \{3, 1\}$

- How much space does the adjacency list take?

  $O(n+m)$

- Once created, how long does it take to check if an edge exists? $O(n)$

  #neighbors

  $O(\text{degree of node})$

- Once created, how long does it take to add an edge

  $O(n)$

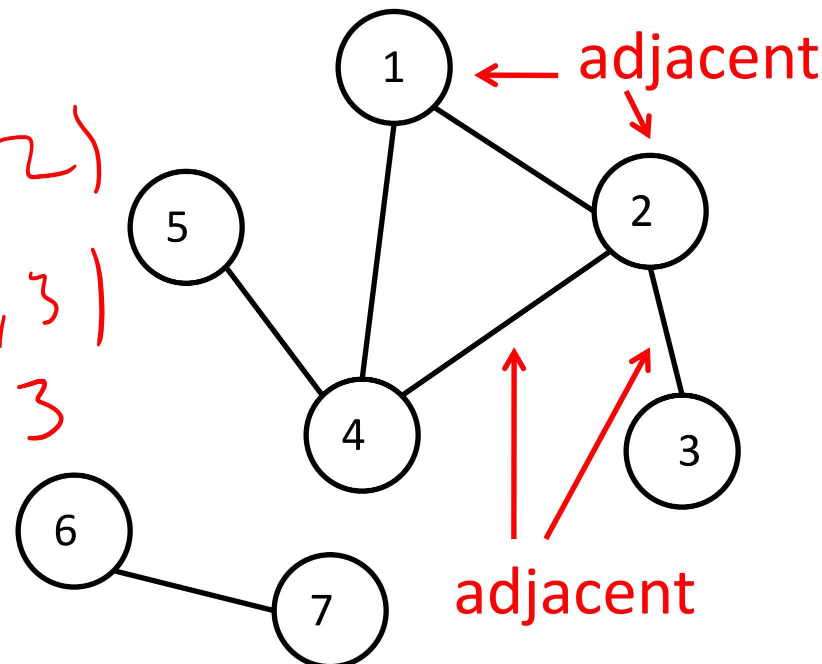
  $O(\text{degree of node})$

Some Basic Terminology

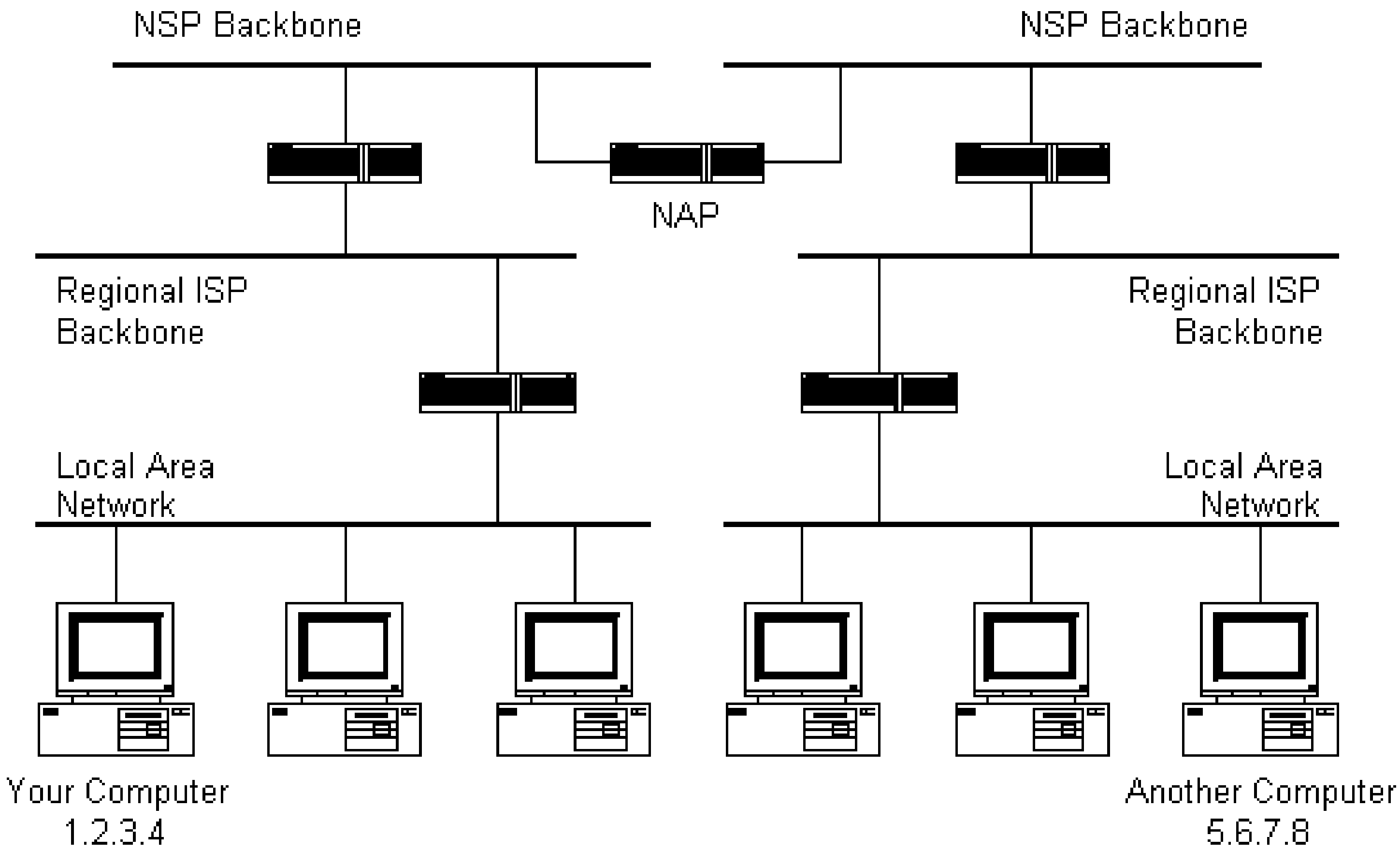
- Two nodes are adjacent if they are connected by an edge
- Two edges are adjacent if they have one vertex in common
- A path is a sequence of adjacent edges from a vertex *v* to another vertex *u*

(5, 4) (4, 1) (1, 2)
(2, 3)

$5 \to 4 \to 1 \to 2 \to 3$



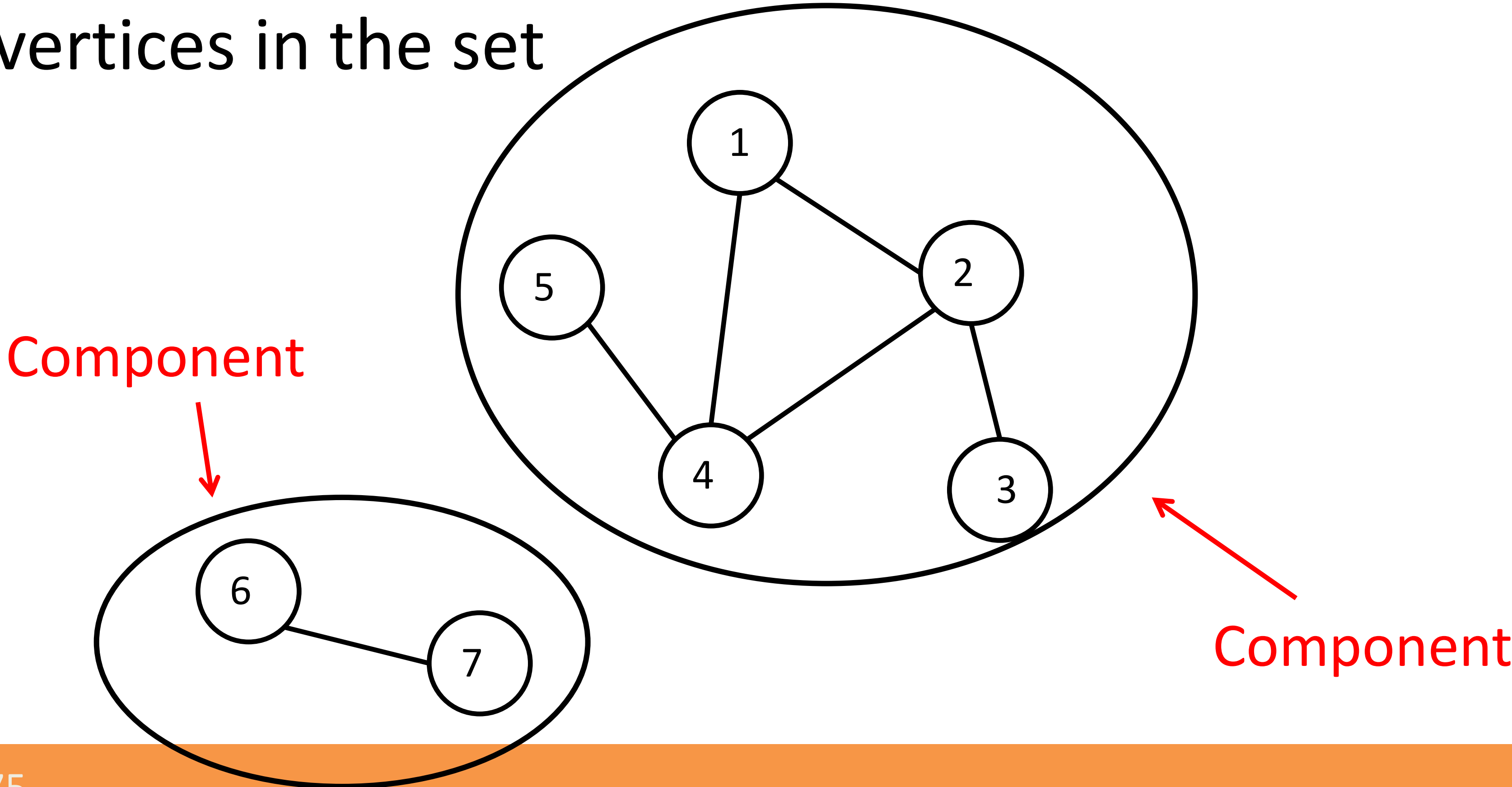
Internet Routing



Some Basic Terminology

connected (CCs)

- A component is a set of vertices such that there exists a path between every pair of vertices in the set



In-Class Exercise

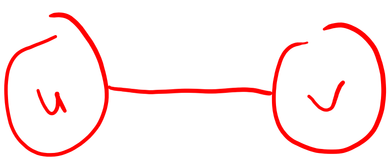
- Suppose you are given a graph *G* in adjacency matrix form
- Goal: given a vertex *u*, output all other vertices that are reachable from *u* (in the same component)

Searches in Graphs

- Suppose we are given a vertex *u* and we want to find all vertices *v* that are reachable from *u* (i.e., there is a path from *u* to *v*)

From u, list all vertices adj. to u

~~Recursively~~ Perform same proc. on those neighbors,



Etc.

Depth First Search

heart of DFS

procedure explore(G, v)

Input: $G = (V, E)$ is a graph; $v \in V$

Output: visited(u) is set to true for all nodes u reachable from v

previsit(v)

visited(v) = true

for each edge $(v, u) \in E$:

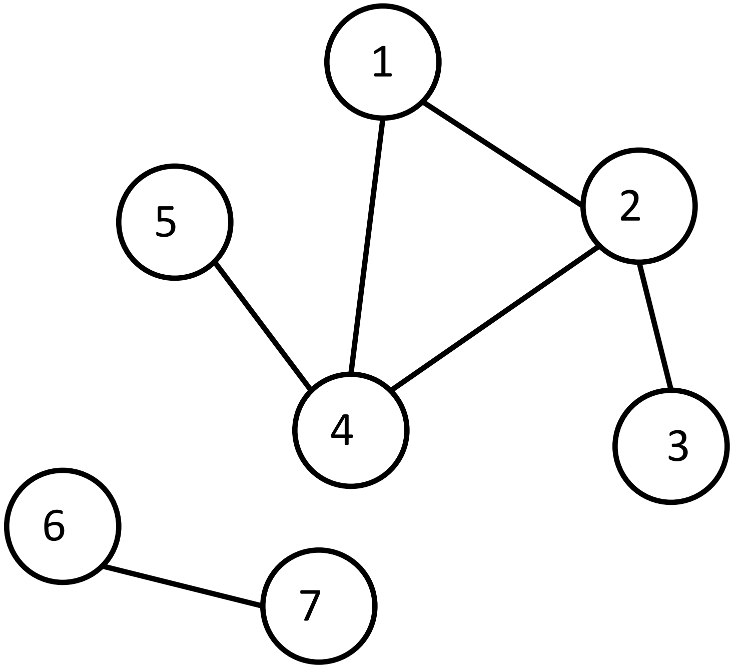
    if not visited(u): explore(u)

postvisit(v)

Ignore for now

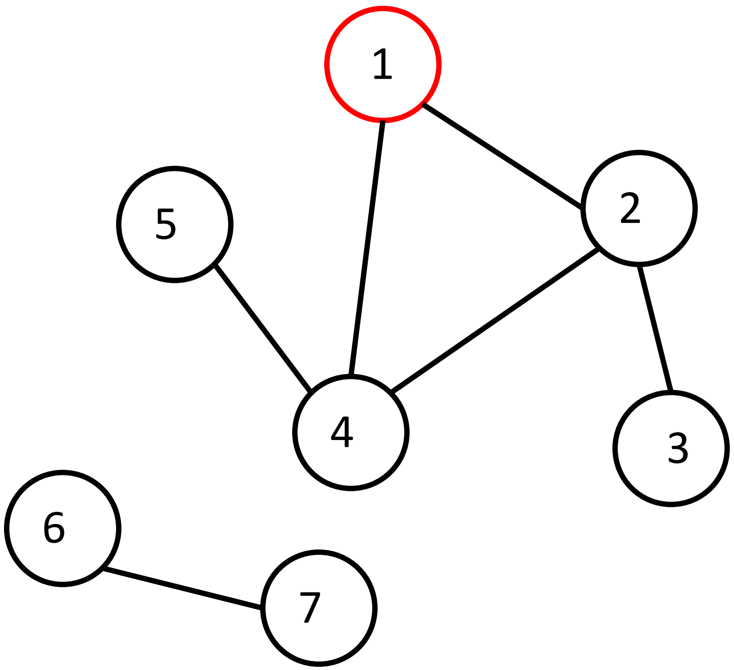
What is the running time of Depth First Search?

Depth First Search: Example



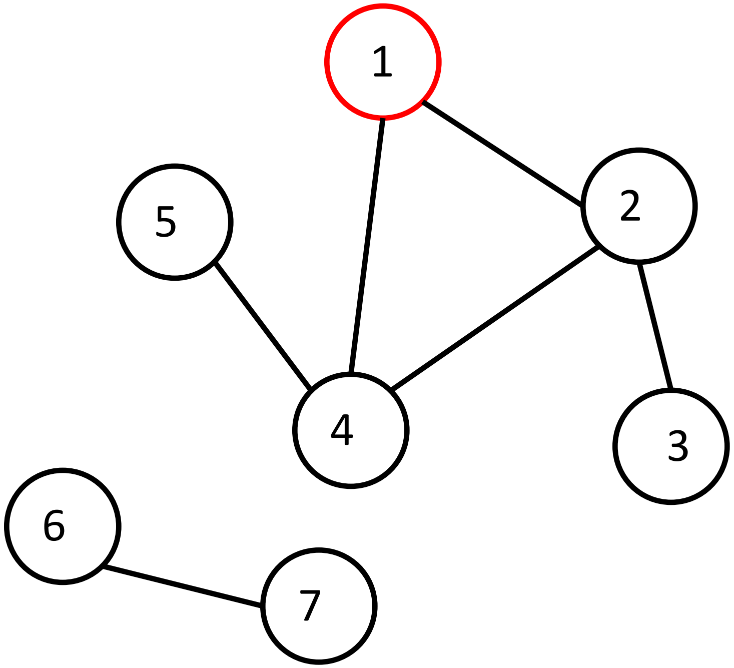
$v = 1$

Depth First Search: Example



Mark *v* as visited

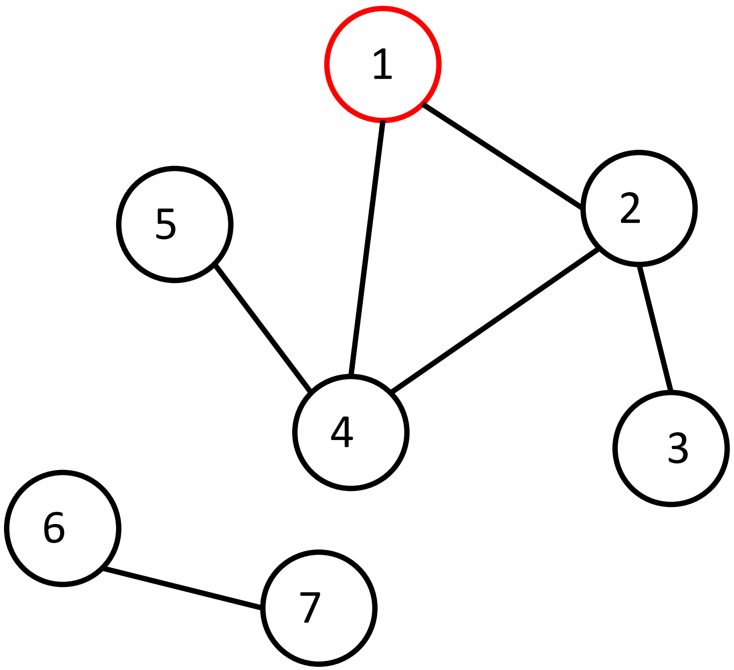
Depth First Search: Example



For every u_1 adjacent to v, if u_1 has not yet been visited, explore u_1

$u_1 = 2, 4$

Depth First Search: Example

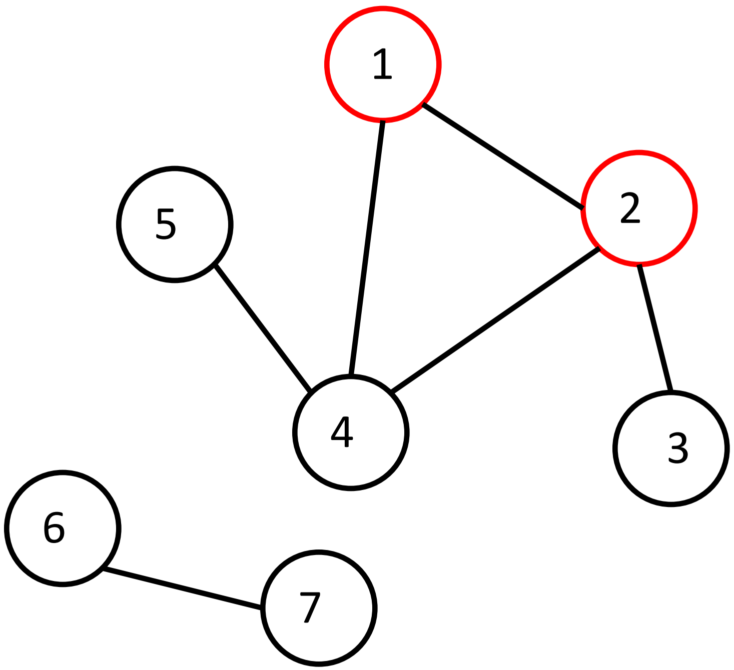


For every u_1 adjacent to v, if u_1 has not yet been visited, explore u_1

u_1 = 2, 4

Let's start with u_1 = 2

Depth First Search: Example



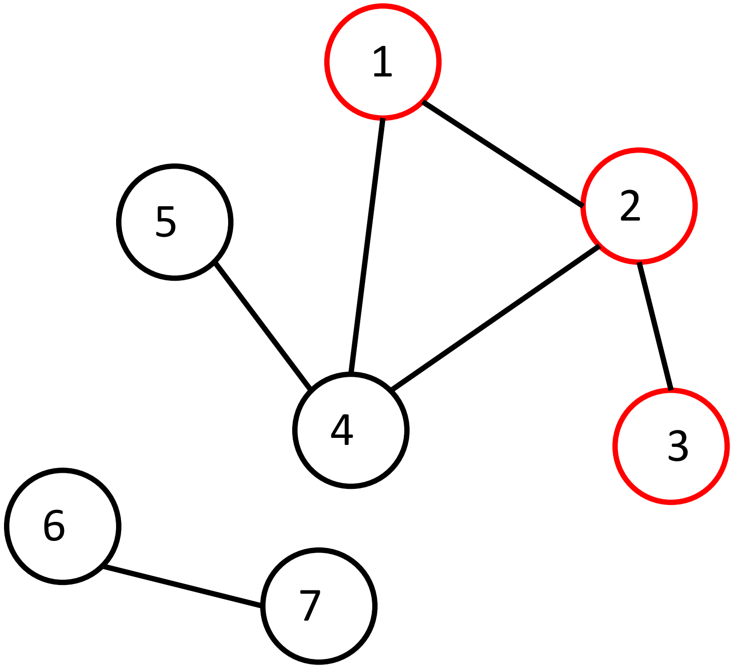
Mark $u_1 = 2$ as visited

For every u_2 adjacent to u_1, if u_2 has not been visited, explore u_2

$u_2 = 3, 4$

Let's start with $u_2 = 3$

Depth First Search: Example

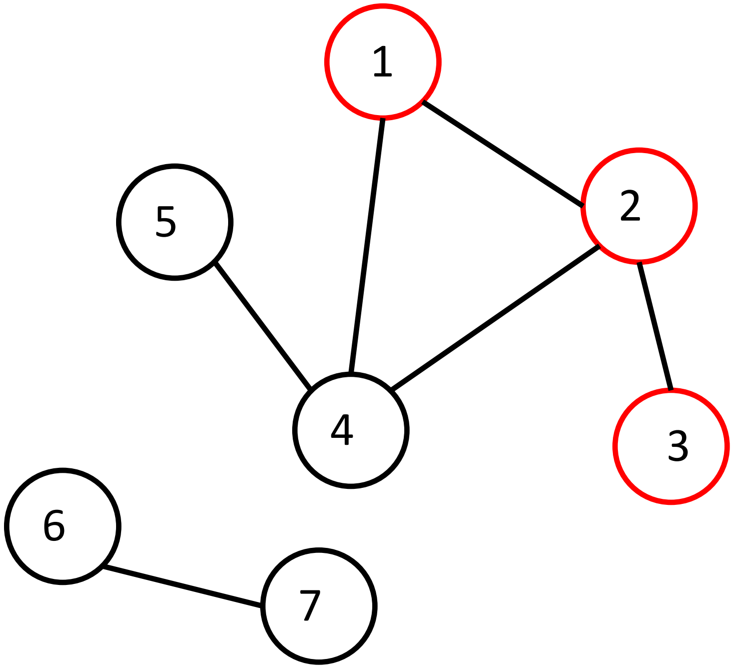


Mark $u_2 = 3$ as visited

For every u_3 adjacent to u_2, if u_3 has not been visited, explore u_3

Everything adjacent to u_3 has been visited already!

Depth First Search: Example

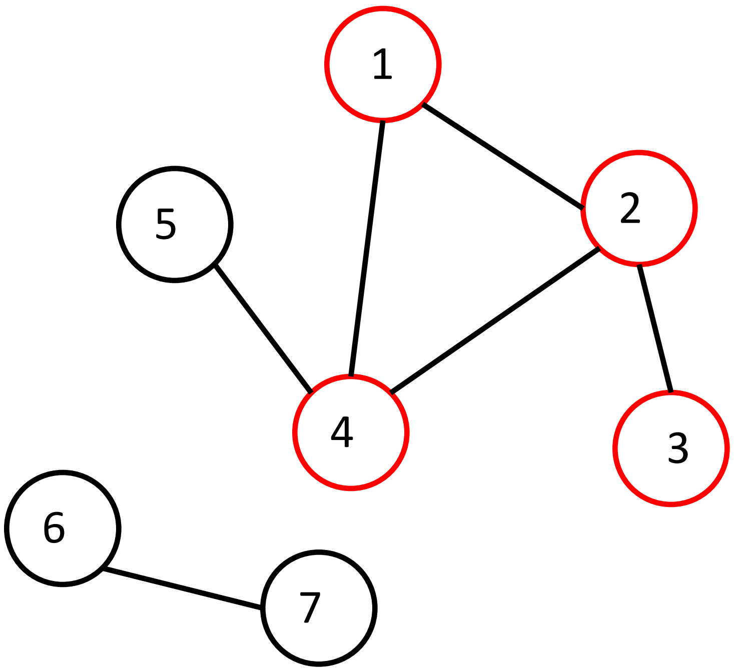


So where were we…?

After visiting $u_1 = 2$, we said we'd have to visit $u_2 = 3, 4$.

3 is done, so time for $u_2 = 4$.

Depth First Search: Example

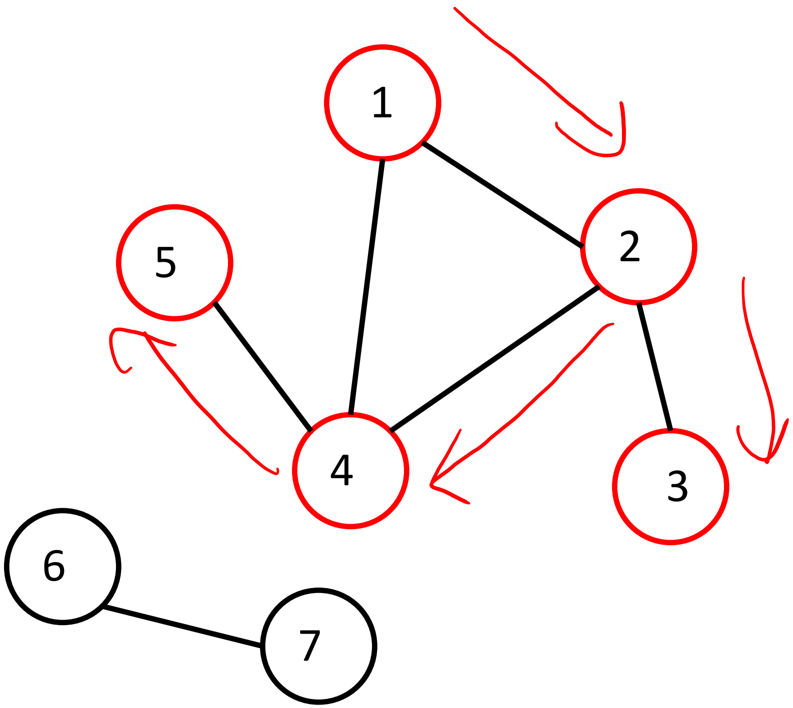


Mark $u_2 = 4$ as visited

For every u_3 adjacent to u_2, if u_3 has not been visited, explore u_3

1, 2, 5 are adjacent to u_2. But 1 and 2 are already visited, so that leaves $u_3 = 5$.

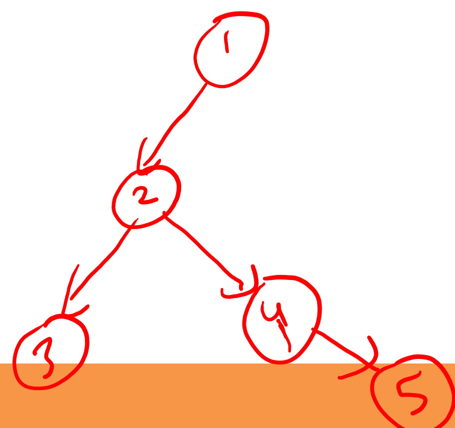
Depth First Search: Example



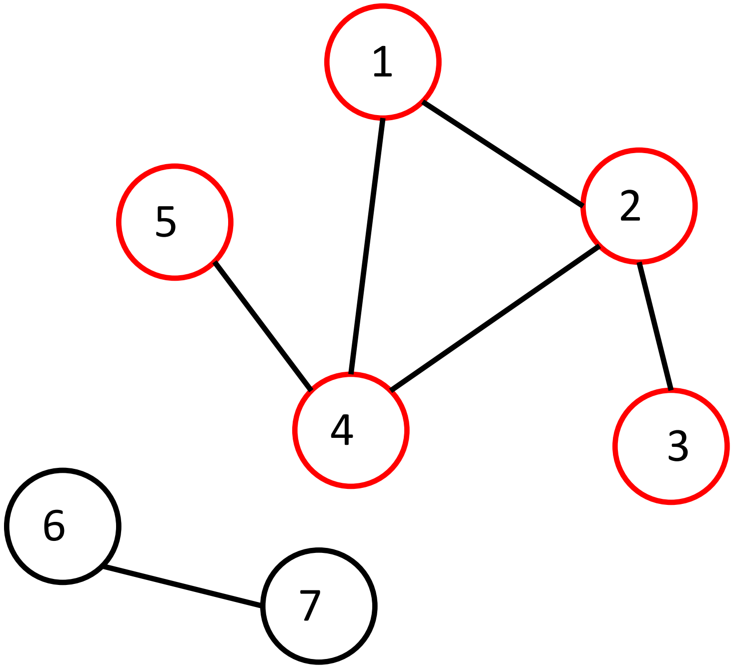
Mark $u_3 = 5$ as visited

For every u_4 adjacent to u_3, if u_4 has not been visited, explore u_4

Everything adjacent to u_3 has been visited!



Depth First Search: Example



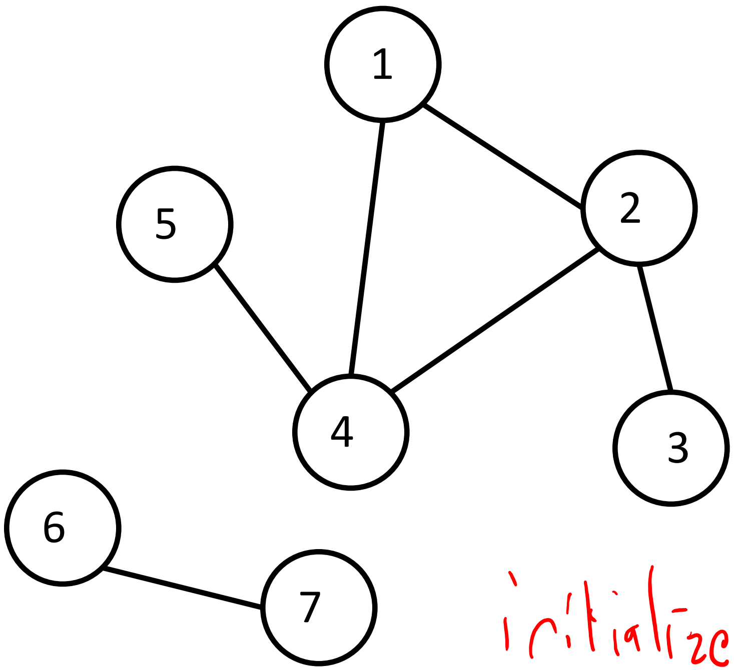
So where were we…?

After visiting $v = 1$, we said we'd have to visit $u_1 = 2, 4$.

We visited 2, and during the process of visiting 2, we visited 4.

So everything is done!

Depth First Search: Example with Stacks



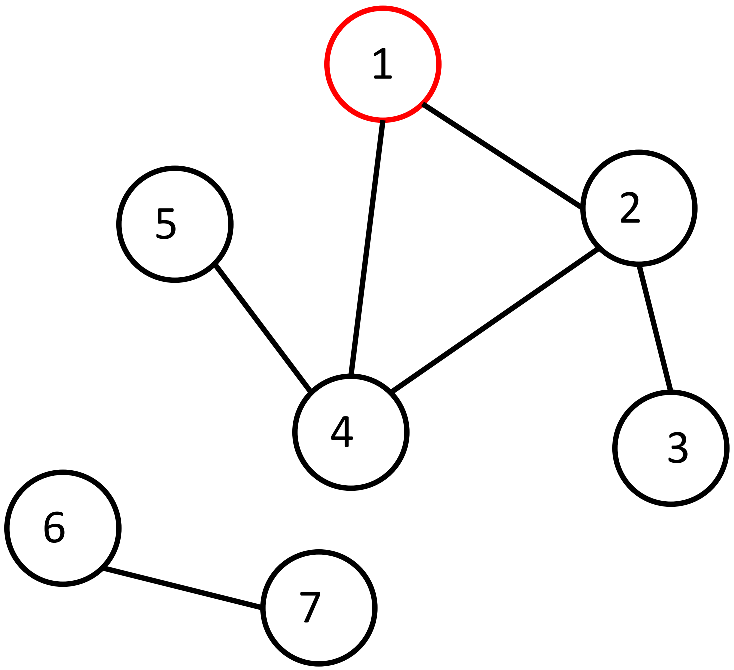
$v = 1$

Start by putting 1 on the stack

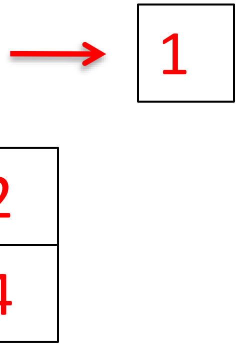
initialize stack

1

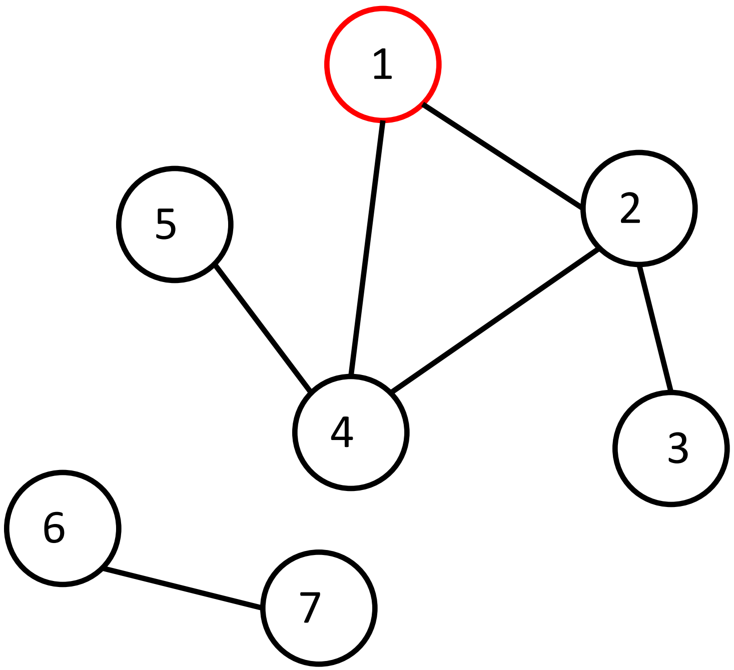
Depth First Search: Example with Stacks



Which node is first on the stack? Pop it from the stack, mark it as explored, put its unexplored neighbors on the stack.



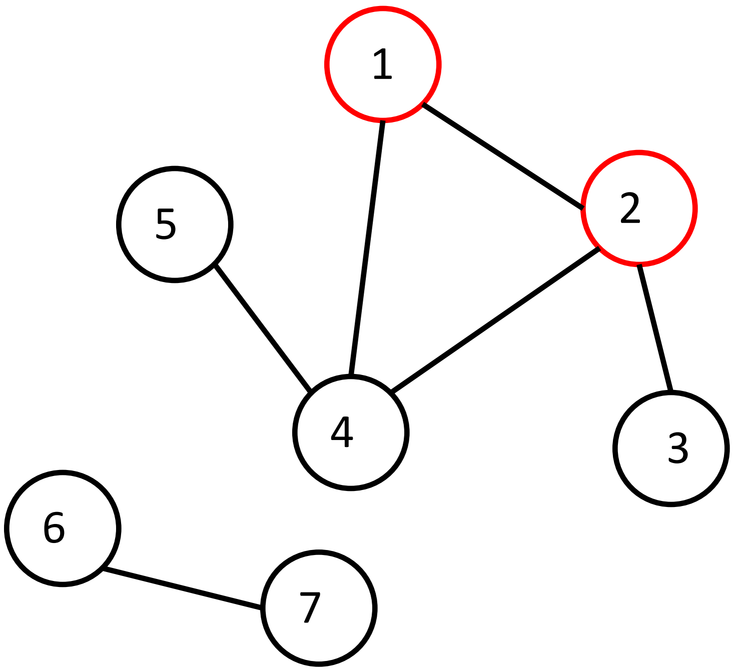
Depth First Search: Example with Stacks



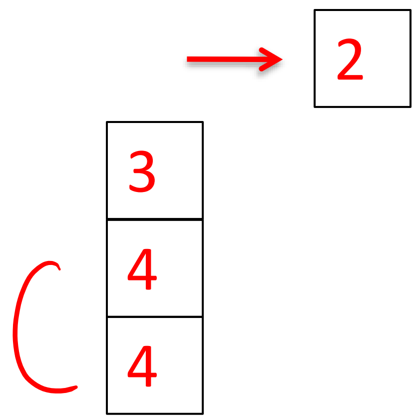
Which node is first on the stack? Pop it from the stack, mark it as explored, put its unexplored neighbors on the stack.

2
4

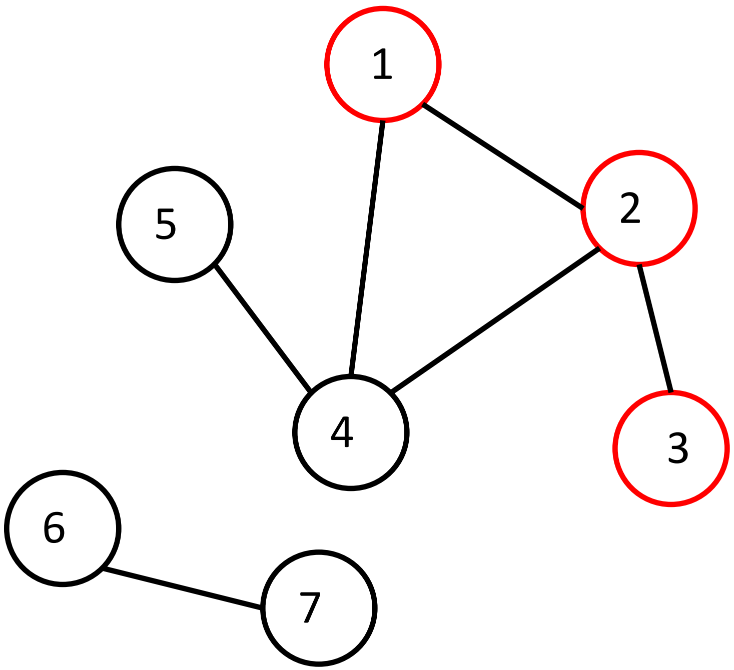
Depth First Search: Example with Stacks



Which node is first on the stack? Pop it from the stack, mark it as explored, put its unexplored neighbors on the stack.



Depth First Search: Example with Stacks



Which node is first on the stack? Pop it from the stack, mark it as explored, put its unexplored neighbors on the stack.

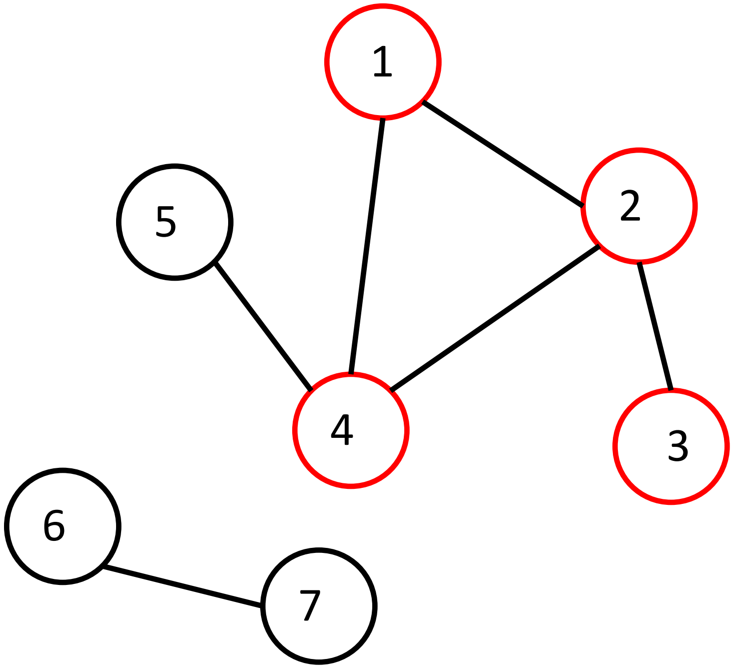
No unexplored neighbors!

→ 3

4 → put on by 2

4

Depth First Search: Example with Stacks

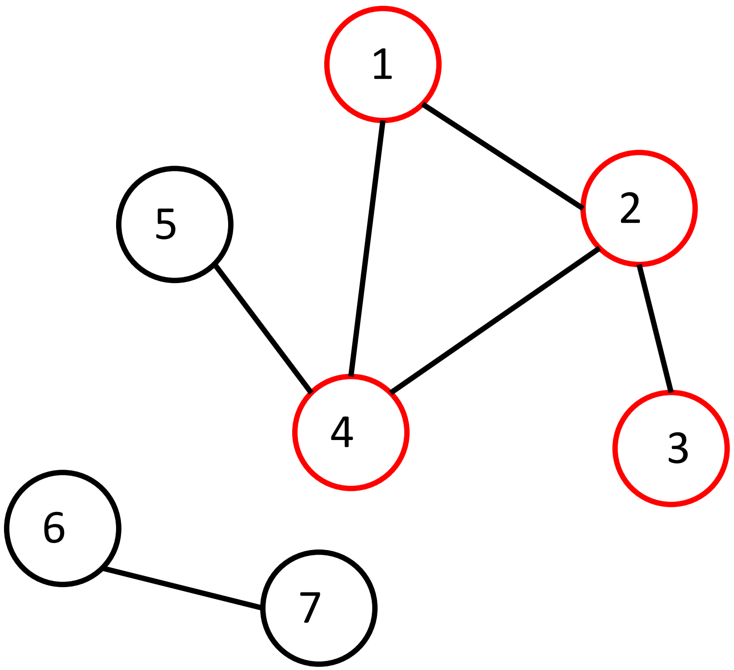


Which node is first on the stack? Pop it from the stack, mark it as explored, put its unexplored neighbors on the stack.

→ 4

5
4

Depth First Search: Example with Stacks



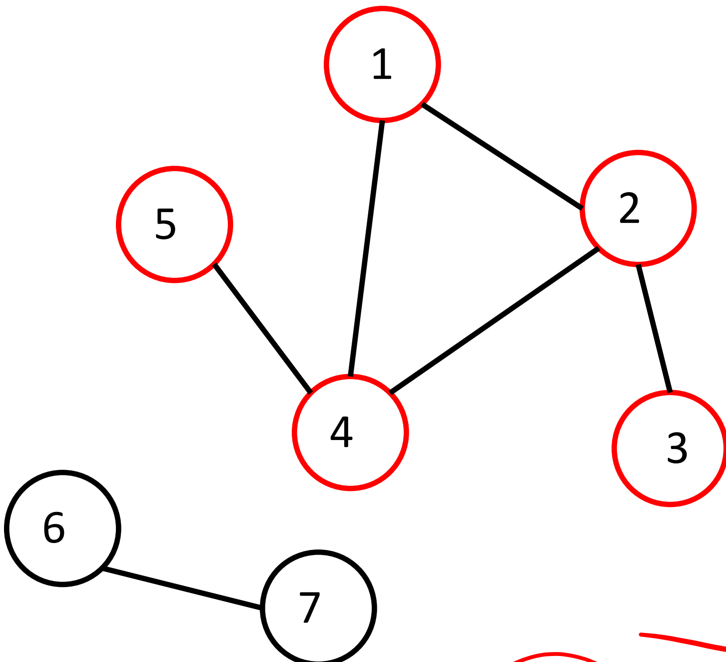
Which node is first on the stack? Pop it from the stack, mark it as explored, put its unexplored neighbors on the stack.

No unexplored neighbors!

→ 5

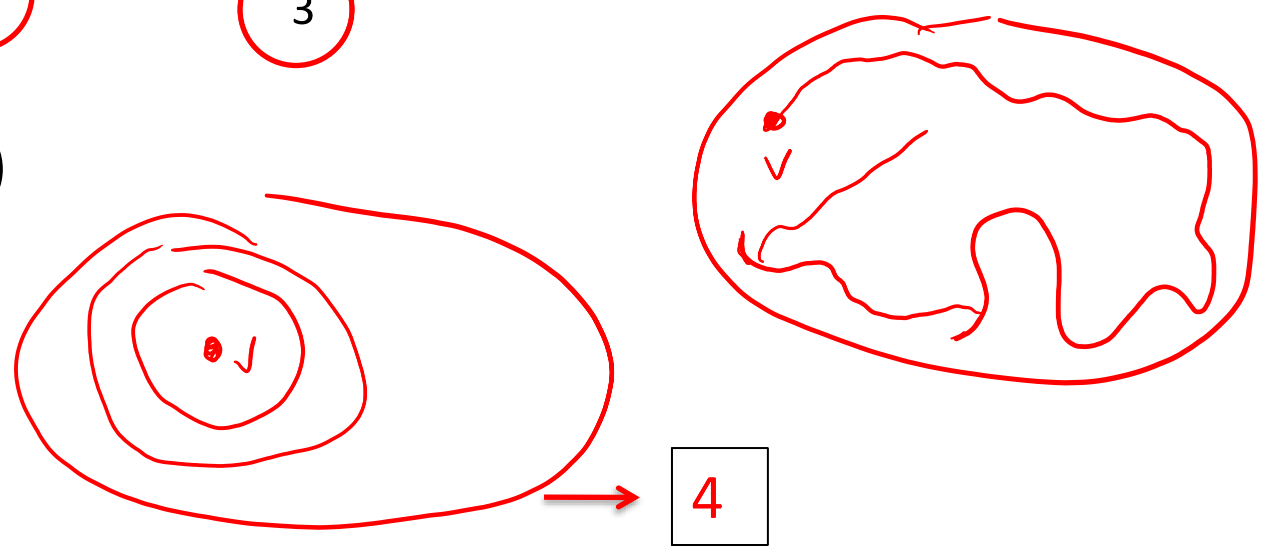
4

Depth First Search: Example with Stacks



Which node is first on the stack? 4 is, but it has already been explored!

Done!



Timekeeping

```
procedure previsit(v)
pre[v] = clock
clock = clock + 1
```

```
procedure postvisit(v)
post[v] = clock
clock = clock + 1
```

pre[*u*] to post[*u*] is the time *u* is on the stack

For all nodes *u*, *v*, either [pre[*u*], post[*u*]] is completely within [pre[*v*], post[*v*]], or the other way around, or there is no overlap. Why?