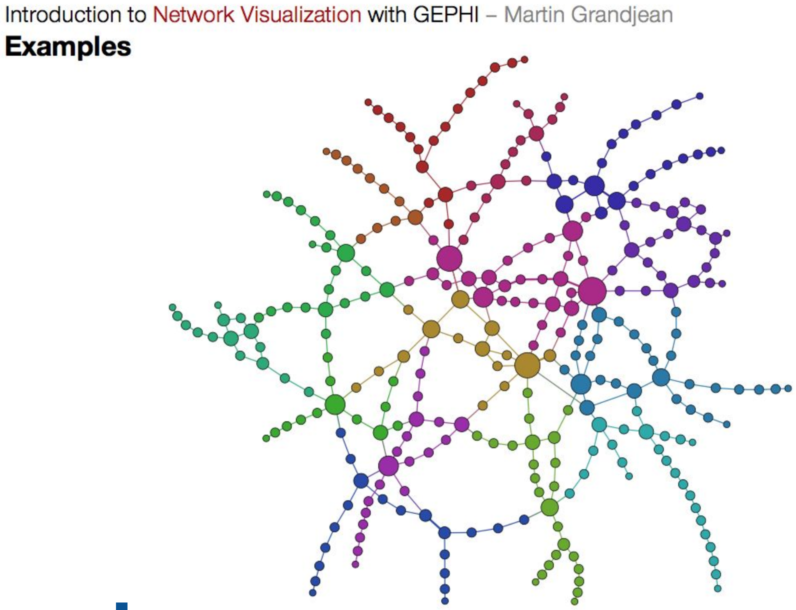



Lecture 22 (Graphs 1)

# Graphs and Traversals

**CS61B, Spring 2024 @ UC Berkeley**

Slides credit: Josh Hug

# Tree Definition

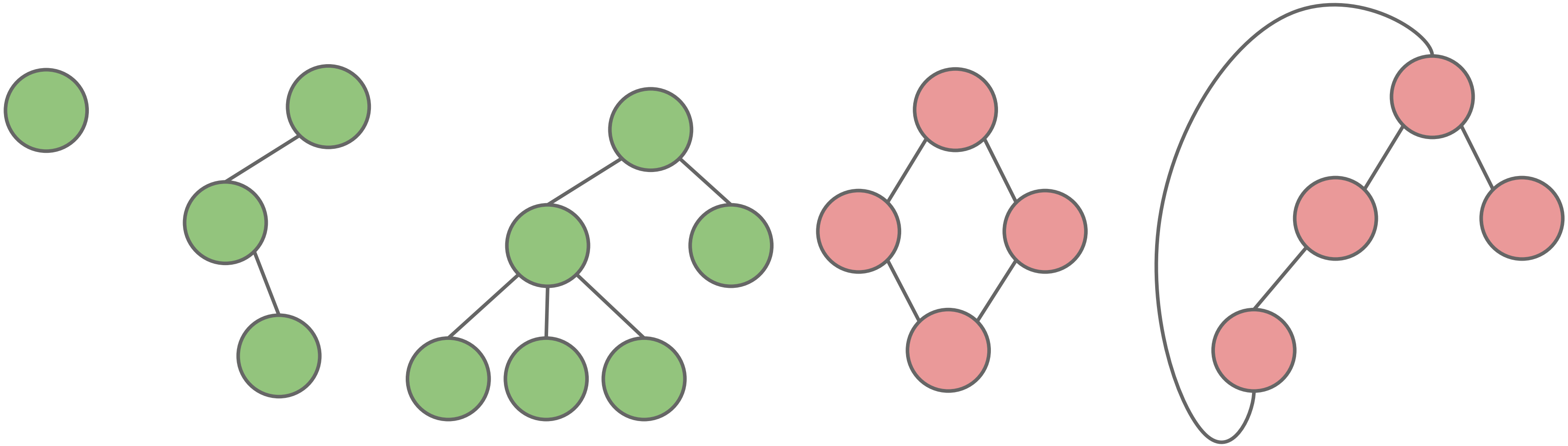
Lecture 22, CS61B, Spring 2024

# Tree Definition (Reminder)

A tree consists of:

- A set of nodes.
- A set of edges that connect those nodes.
  - Constraint: There is exactly one path between any two nodes.

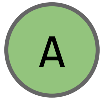
Green structures below are trees. Pink ones are not.



# Rooted Trees Definition (Reminder)

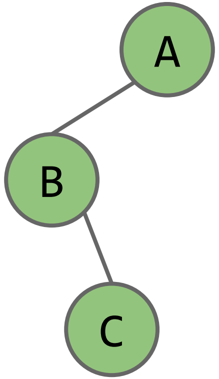
A rooted tree is a tree where we’ve chosen one node as the “root”.

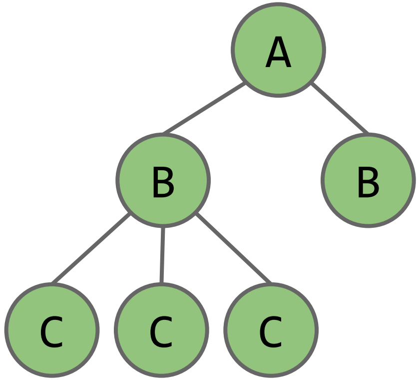
- Every node N except the root has exactly one parent, defined as the first node on the path from N to the root.
- A node with no child is called a leaf.




For each of these:

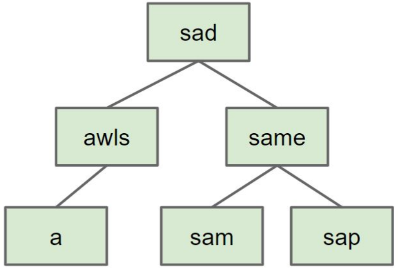
- A is the root.
- B is a child of A. (and C of B)
- A is a parent of B. (and B of C)

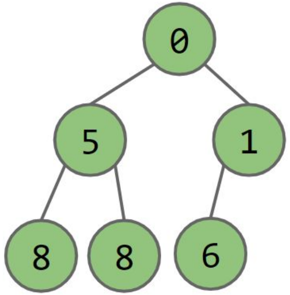


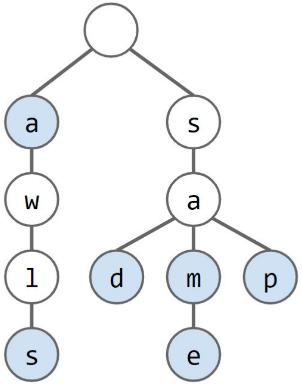



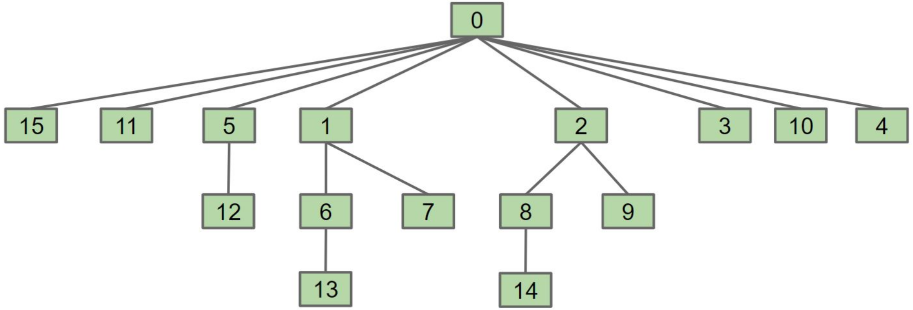
# Trees

We’ve seen trees as nodes in a specific data structure implementation: Search Trees, Tries, Heaps, Disjoint Sets, etc.



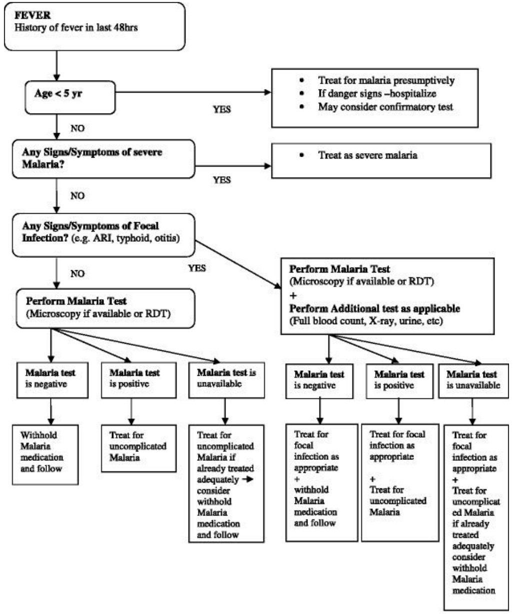



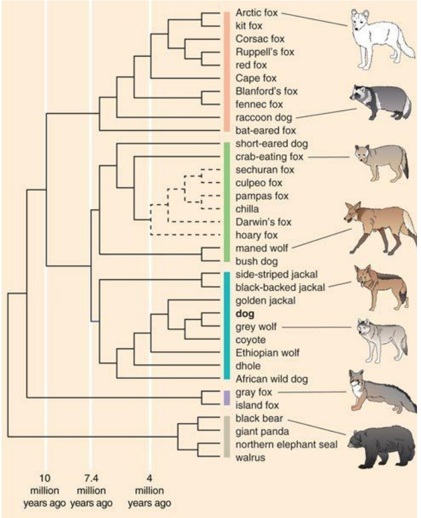
# Trees

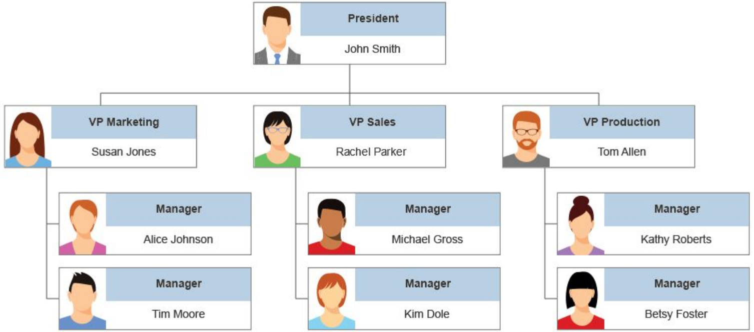
Trees are a more general concept.

- Organization charts.
- Family lineages* including phylogenetic trees.
- MOH Training Manual for Management of Malaria.




Source: MOH (2009) Training Manual for the Management of Malaria at Health Facilities in Ghana




*: Not all family lineages are trees!

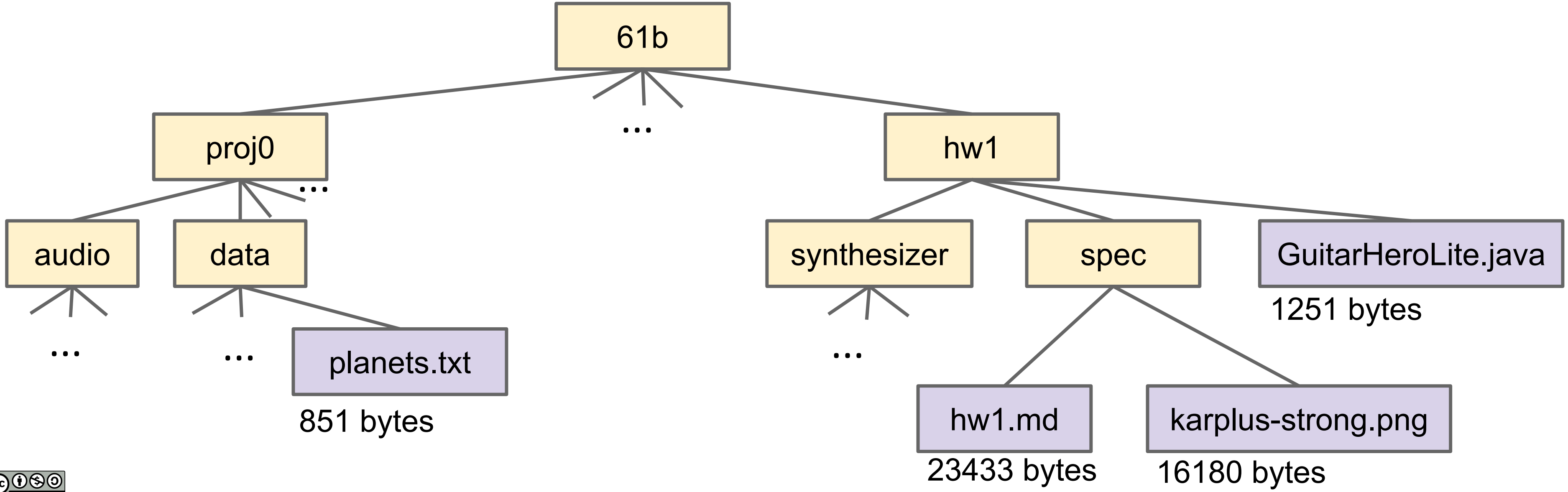
# Tree Traversals

Lecture 22, CS61B, Spring 2024

# Example: File System Tree

Sometimes you want to iterate over a tree. For example, suppose you want to find the total size of all files in a folder called 61b.

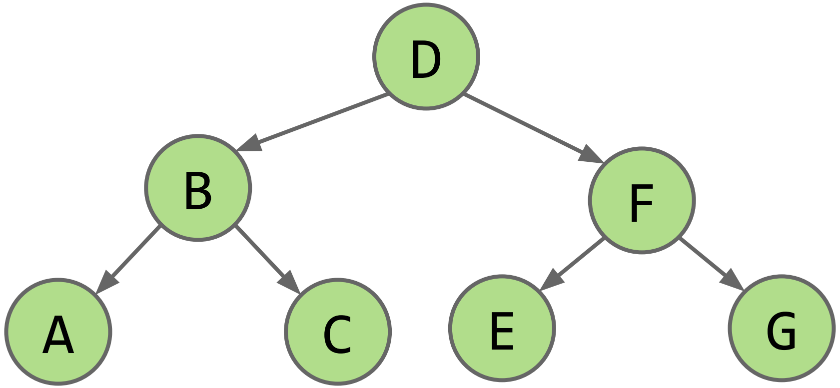
- What one might call “tree iteration” is actually called “tree traversal.”
- Unlike lists, there are many orders in which we might **visit** the nodes.
    - Each ordering is useful in different ways.



# Tree Traversal Orderings

Level Order

- Visit top-to-bottom, left-to-right (like reading in English): DBFACEG



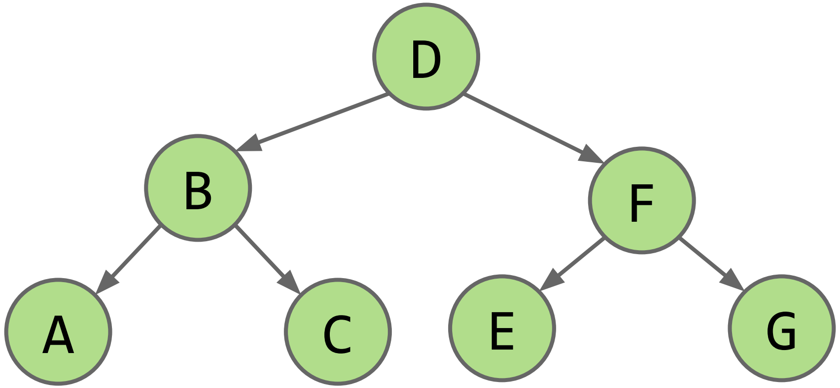
# Tree Traversal Orderings

Level Order

- Visit top-to-bottom, left-to-right (like reading in English): DBFACEG

Depth First Traversals

- 3 types: Preorder, Inorder, Postorder
- Basic (rough) idea: Traverse “deep nodes” (e.g. A) before shallow ones (e.g. F).
- Note: Traversing a node is different than “visiting” a node. See next slide.



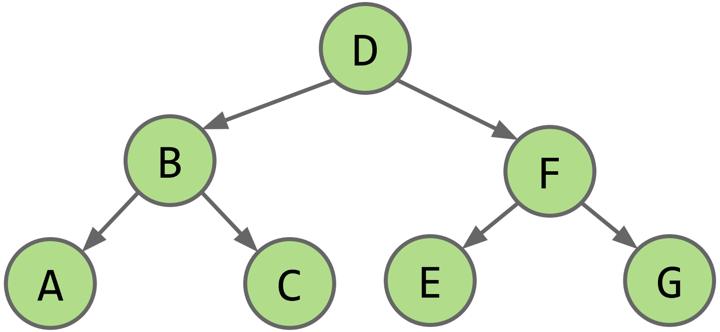
# Demo: Preorder Depth-First Tree Traversal

Preorder: "Visit" a node, then traverse its children:

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

`preOrder(D)`



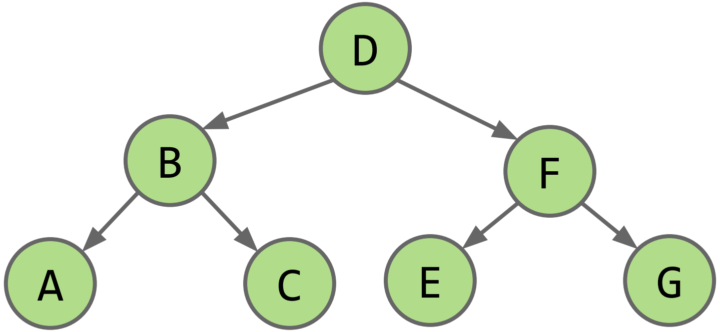
# Demo: Preorder Depth-First Tree Traversal

Preorder: "Visit" a node, then traverse its children: D

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

`preOrder(D)`



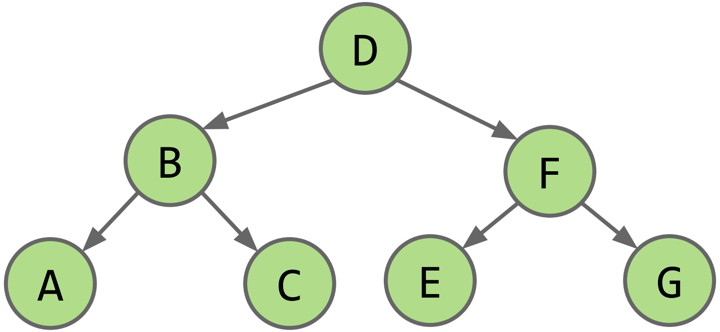
## Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: D

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

preOrder(D)



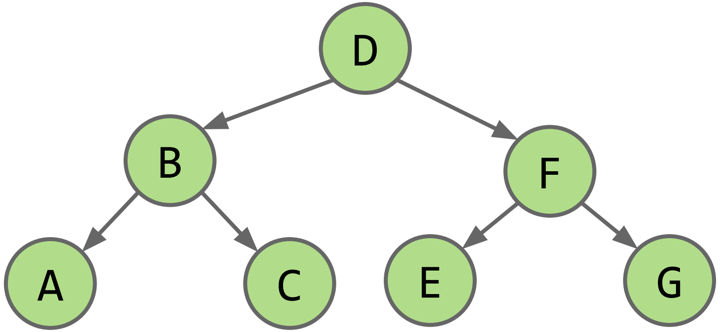
# Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: D

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(B)
```



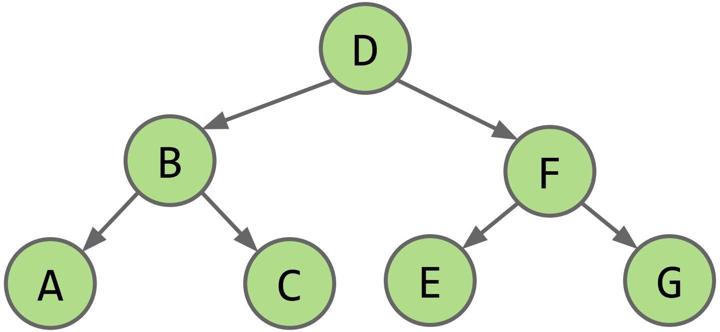
# Demo: Preorder Depth-First Tree Traversal

Preorder: "Visit" a node, then traverse its children: DB

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(B)
```



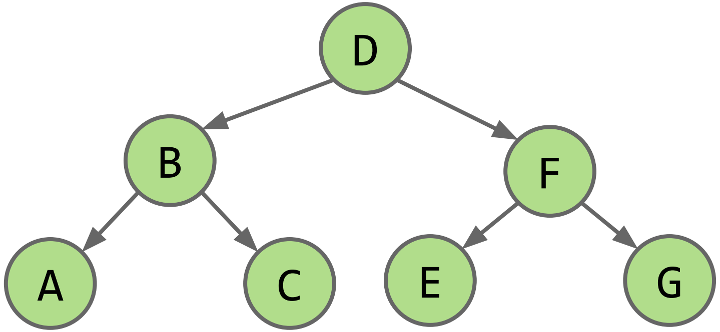
# Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: DB

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(B)
```



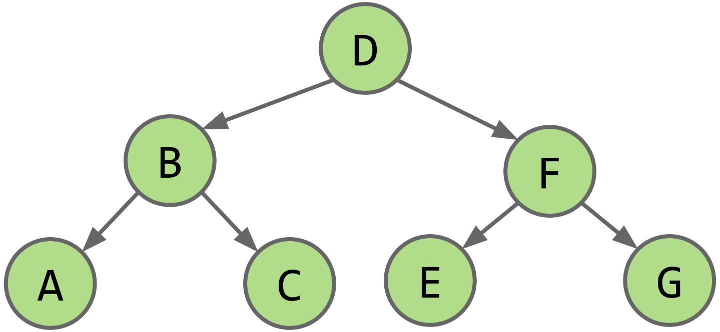
# Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: DB

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(B)
    preOrder(A)
```



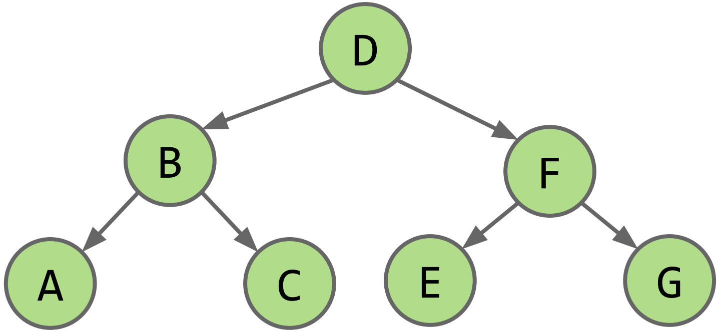
# Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: DBA

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

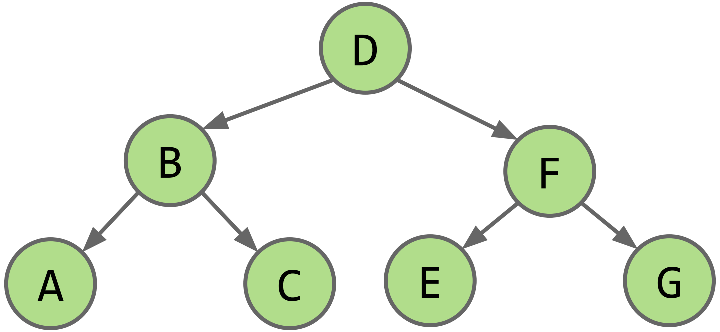
```
preOrder(D)
  preOrder(B)
    preOrder(A)
```



# Demo: Preorder Depth-First Tree Traversal

Preorder: "Visit" a node, then traverse its children: DBA

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```




Call stack:

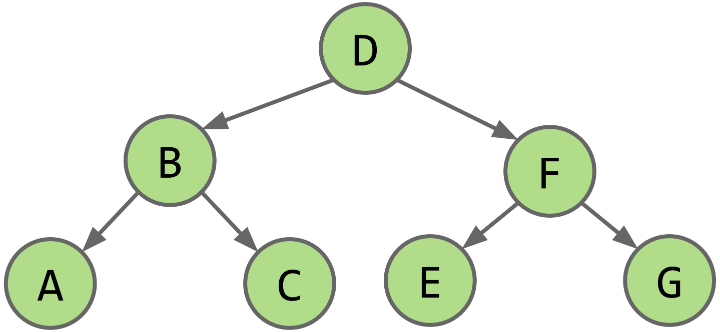
```
preOrder(D)
  preOrder(B)
    preOrder(A)
```

Skipping over the steps of `preOrder(null)` for brevity.

# Demo: Preorder Depth-First Tree Traversal

Preorder: "Visit" a node, then traverse its children: DBA

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```




Call stack:

```
preOrder(D)
  preOrder(B)
    preOrder(A)
```

Skipping over the steps of `preOrder(null)` for brevity.

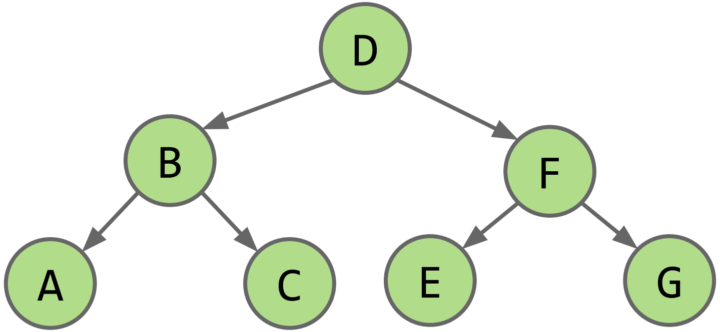
# Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: DBA

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(B)
```



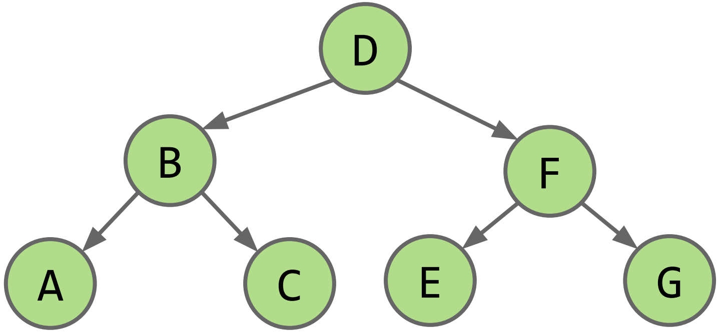
# Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: DBAC

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(B)
```




Skipping over the steps of `preOrder(C)` for brevity.

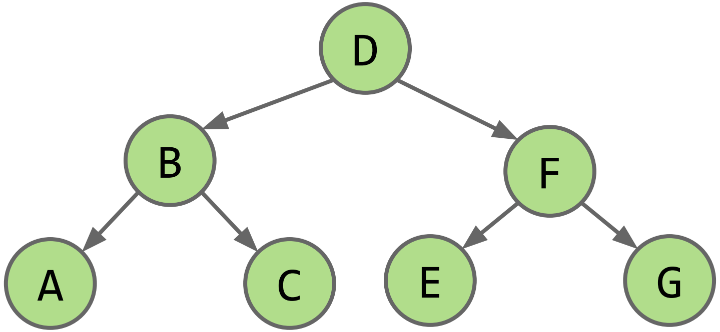
# Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: DBAC

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

`preOrder(D)`



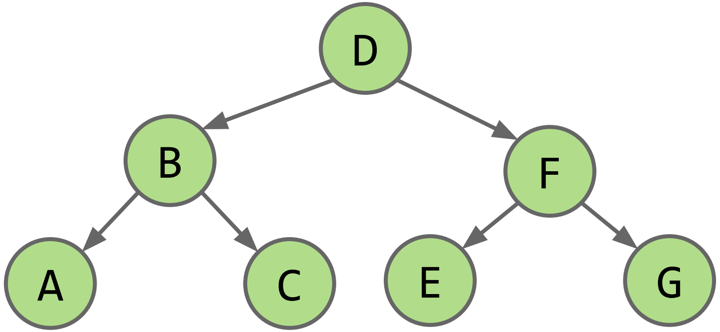
# Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: DBAC

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(F)
```



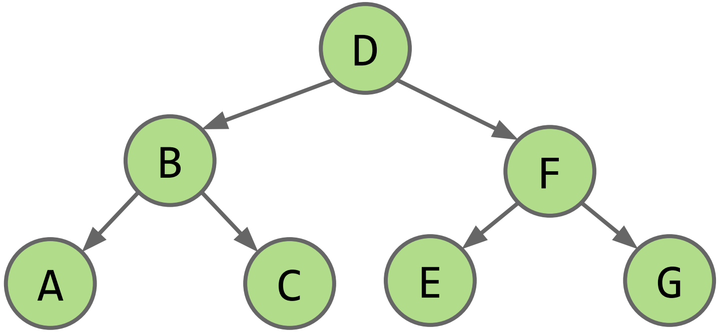
## Demo: Preorder Depth-First Tree Traversal

Preorder: “Visit” a node, then traverse its children: DBACF

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(F)
```



# Demo: Preorder Depth-First Tree Traversal

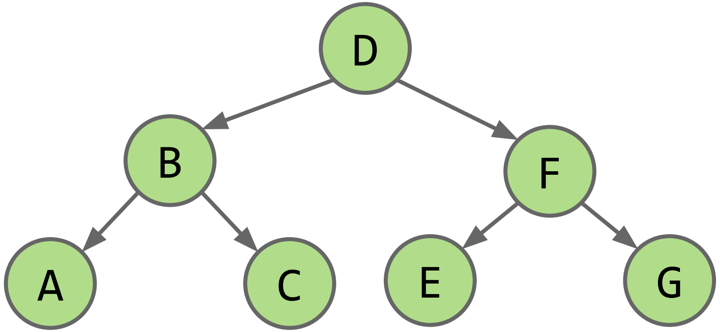
Preorder: “Visit” a node, then traverse its children: DBACFE

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

`preOrder(D)`

`    preOrder(F)`




Skipping over the steps of `preOrder(E)` for brevity.

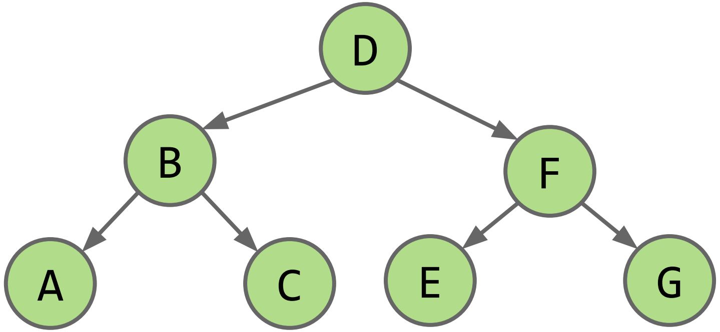
# Demo: Preorder Depth-First Tree Traversal

Preorder: "Visit" a node, then traverse its children: DBACFEG

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

Call stack:

```
preOrder(D)
  preOrder(F)
```




Skipping over the steps of `preOrder(G)` for brevity.

# Depth First Traversals

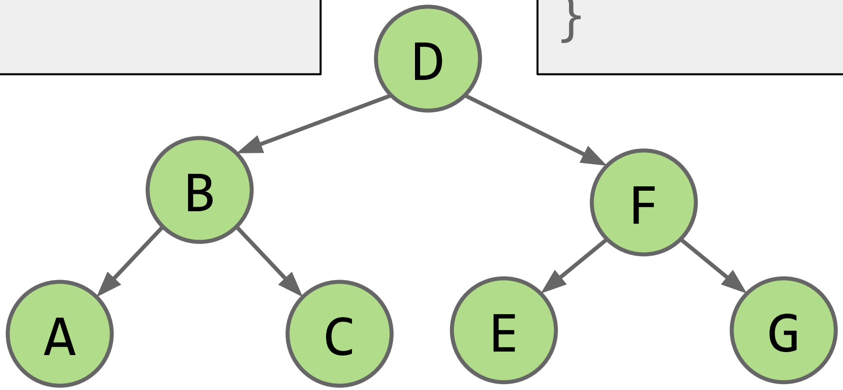
Preorder traversal: “Visit” a node, then traverse its children:  DBACFEG

Inorder traversal: Traverse left child, visit, then traverse right child:

ABCDEFG

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

```
inOrder(BSTNode x) {
    if (x == null) return;
    inOrder(x.left)
    print(x.key)
    inOrder(x.right)
}
```



# Depth First Traversals http://yellkey.com/yes

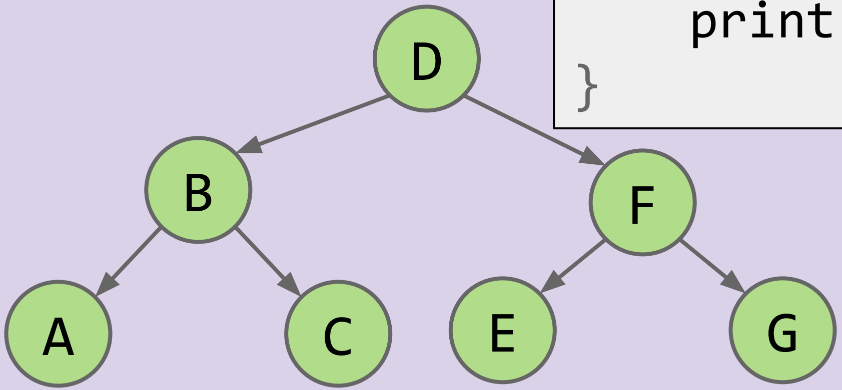
Preorder traversal: “Visit” a node, then traverse its children: DBACFEG

Inorder traversal: Traverse left child, visit, traverse right child: ABCDEFG

Postorder traversal: Traverse left, traverse right, then visit: ???????

1. DBACEFG
2. GFEDCBA
3. GEFCABD
4. ACBEGFD
5. ACBFEGD
6. Other

```
postOrder(BSTNode x) {
    if (x == null) return;
    postOrder(x.left)
    postOrder(x.right)
    print(x.key)
}
```



# Depth First Traversals

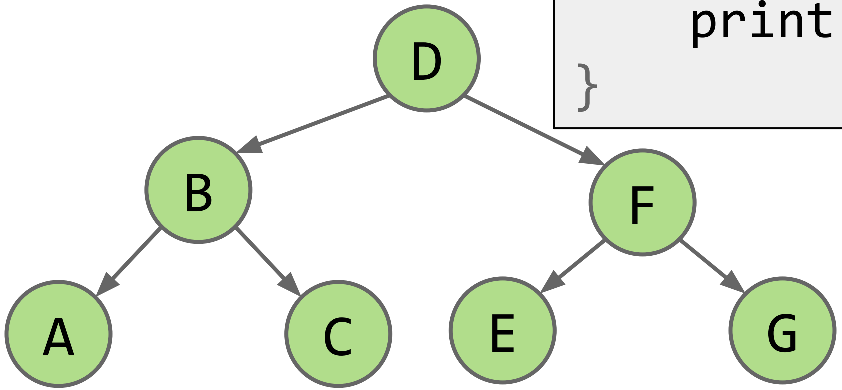
Preorder traversal: "Visit" a node, then traverse its children: DBACFEG

Inorder traversal: Traverse left child, visit, traverse right child: ABCDEFG

Postorder traversal: Traverse left, traverse right, then visit: ACBEGFD

1. DBACEFG
2. GFEDCBA
3. GEFCABD
4. **ACBEGFD**
5. ACBFEGD
6. Other

```
postOrder(BSTNode x) {
    if (x == null) return;
    postOrder(x.left)
    postOrder(x.right)
    print(x.key)
}
```

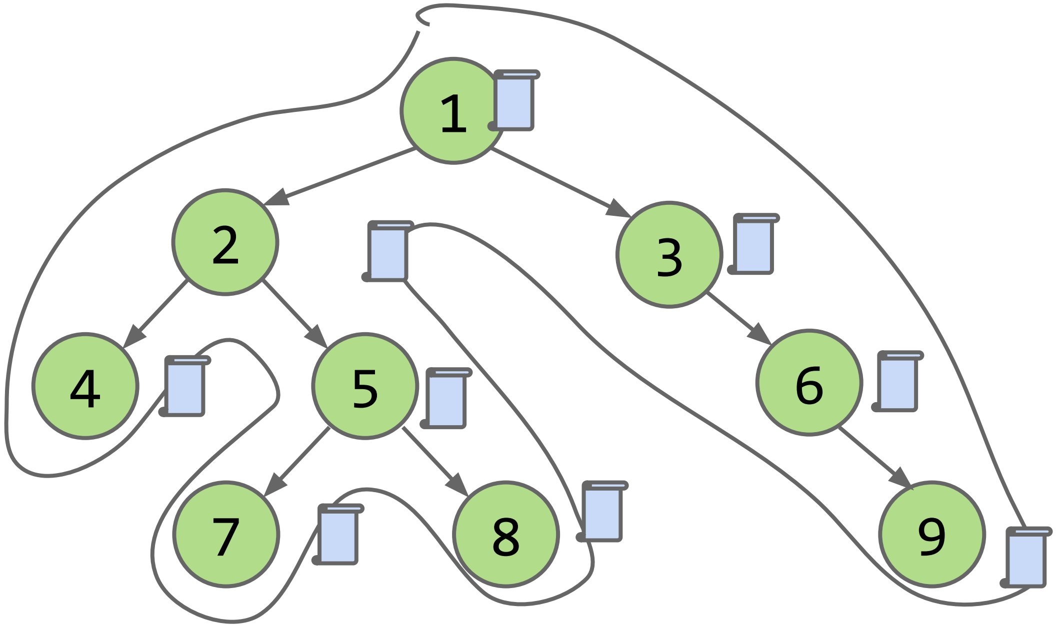


# A Useful Visual Trick (for Humans, Not Algorithms)

- Preorder traversal: We trace a path around the graph, from the top going counter-clockwise. “Visit” every time we pass the LEFT of a node.
- Inorder traversal: “Visit” when you cross the bottom of a node.
- Postorder traversal: “Visit” when you cross the right a node.

Example: Post-Order Traversal

- 4 7 8 5 2 9 6 3 1



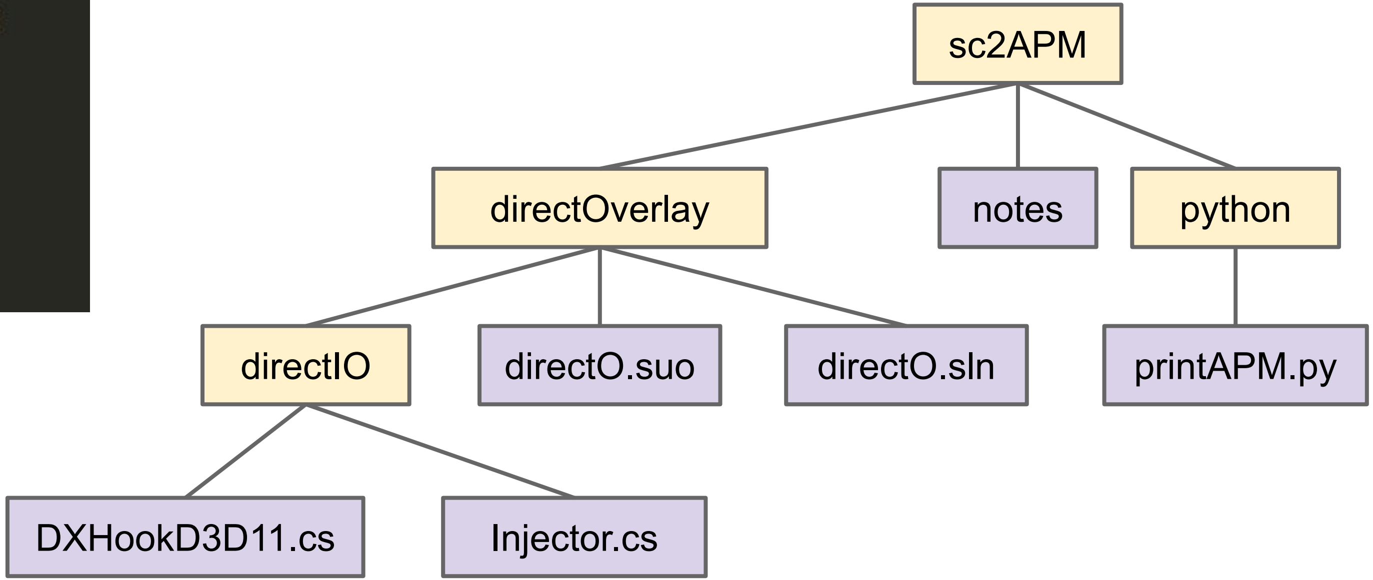
# Usefulness of Tree Traversals

Lecture 22, CS61B, Spring 2024

# What Good Are All These Traversals?

Example: Preorder Traversal for printing directory listing:

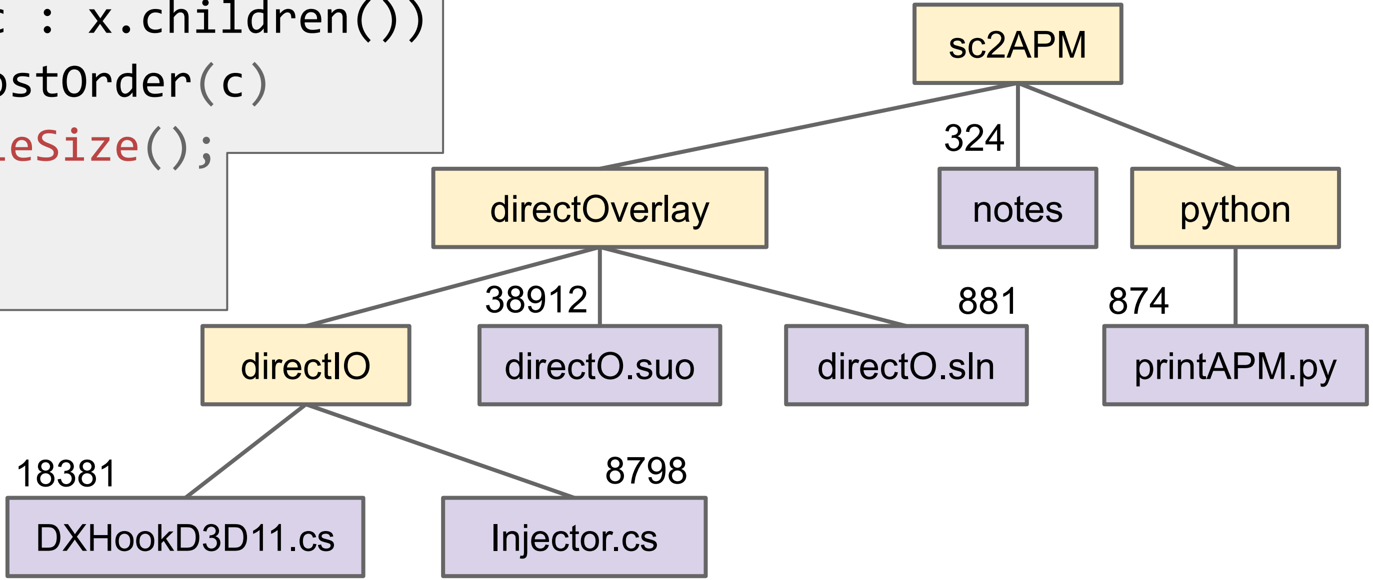
```
sc2APM/
  directOverlay/
    directIO/
      DXHookD3D11.cs
      Injector.cs
    directO.suo
    directO.sln
  notes
  python/
    printAPM.py
```



# What Good Are All These Traversals?

Example: Postorder Traversal for gathering file sizes.

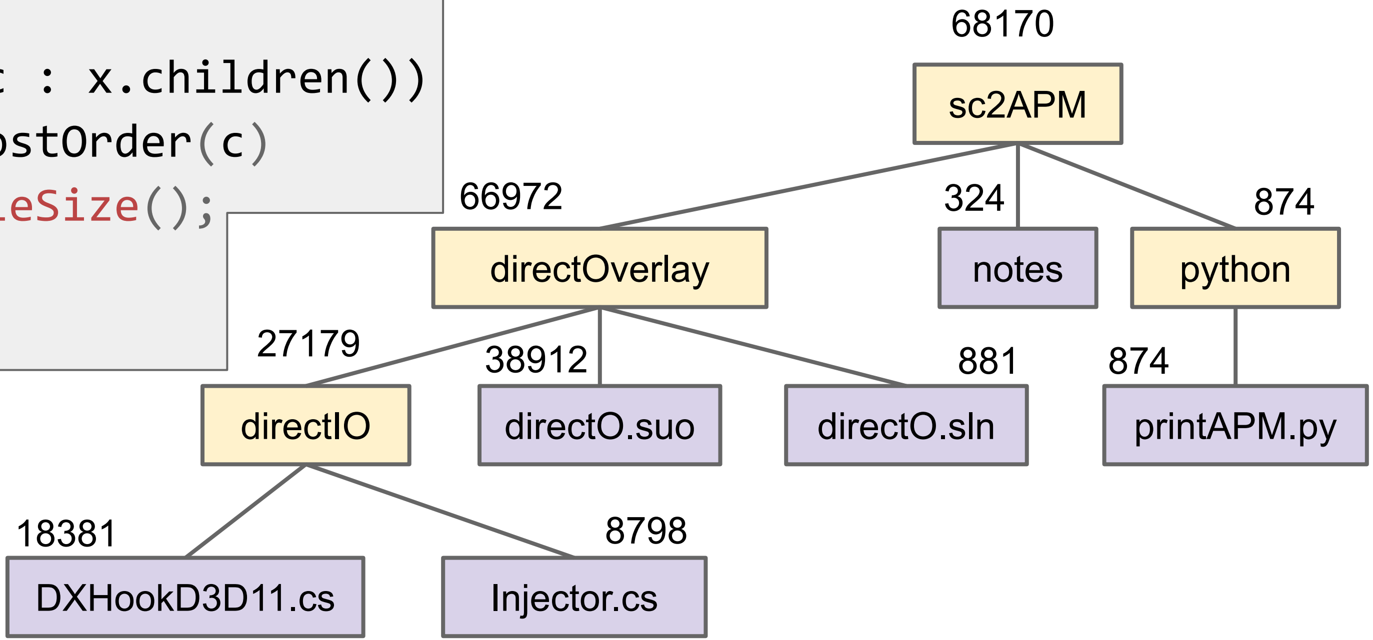
```
postOrder(BSTNode x) {
    if (x == null) return 0;
    int total = 0;
    for (BSTNode c : x.children())
        total += postOrder(c)
    total += x.fileSize();
    return total;
}
```



## What Good Are All These Traversals?

Example: Postorder Traversal for gathering file sizes.

```
postOrder(BSTNode x) {
    if (x == null) return 0;
    int total = 0;
    for (BSTNode c : x.children())
        total += postOrder(c)
    total += x.fileSize();
    return total;
}
```



# Graph Definition

Lecture 22, CS61B, Spring 2024

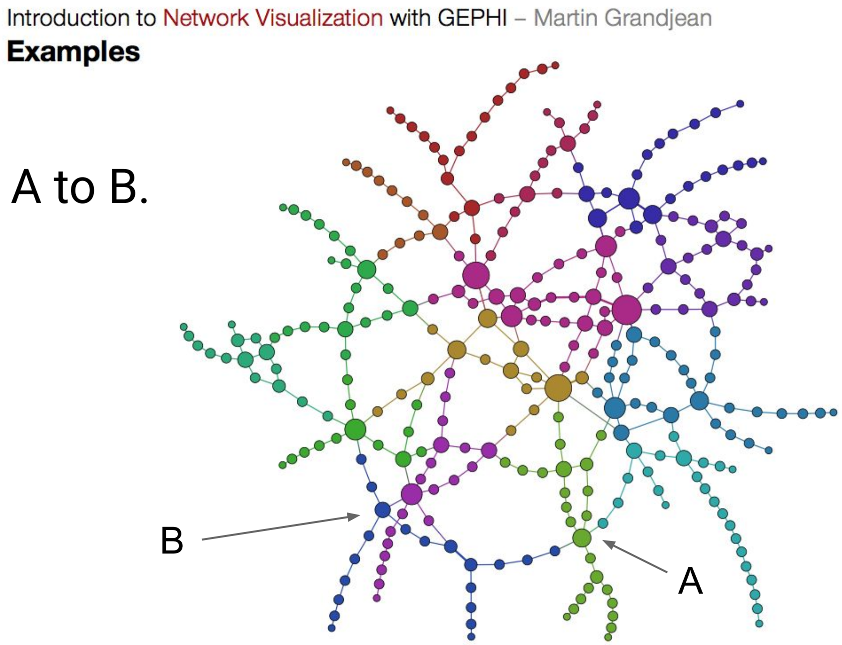
# Trees and Hierarchical Relationships

Trees are fantastic for representing strict hierarchical relationships.

- But not every relationship is hierarchical.
- Example: Paris Metro map.

This is not a tree: Contains cycles!

- More than one way to get from A to B.

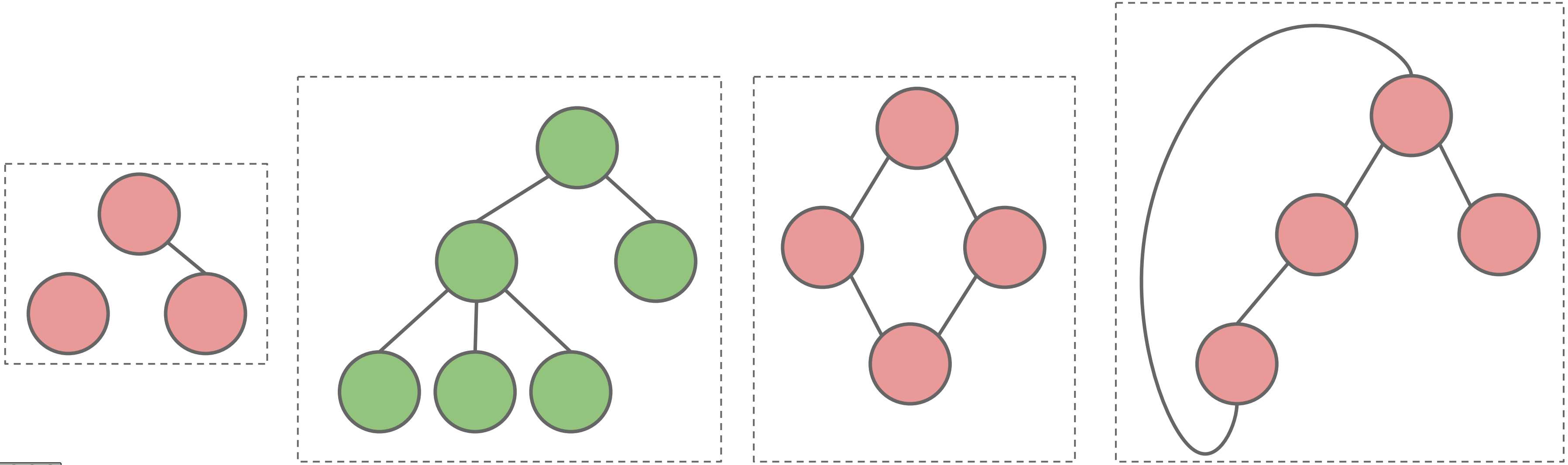


# Tree Definition (Revisited)

A tree consists of:

- A set of nodes.
- A set of edges that connect those nodes.
  - Constraint: There is exactly one path between any two nodes.

Green structures on slide are trees. Pink ones are not.



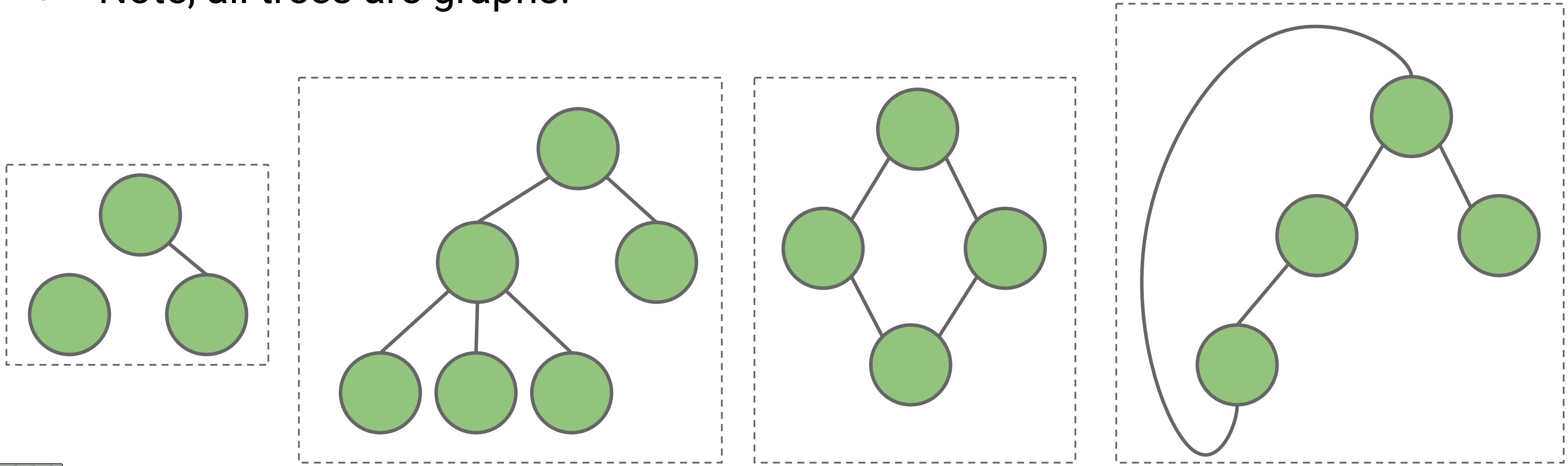
## Graph Definition

A graph consists of:

- A set of nodes.
- A set of zero or more edges, each of which connects two nodes.

Green structures below are graphs.

- Note, all trees are graphs!



# Graph Example: BART

Is the BART graph a tree?

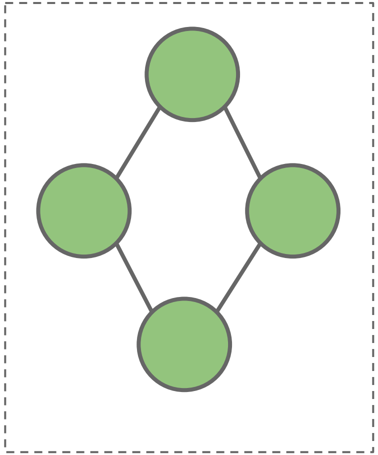
- No, has one cycle.
  - San Bruno
  - SFO
  - Millbrae

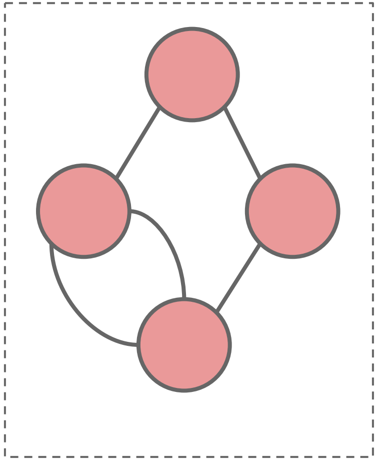
## Graph Definition

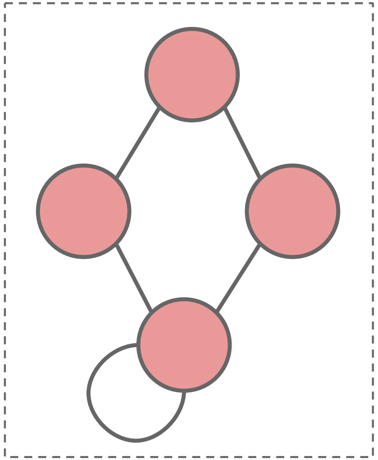
A simple graph is a graph with:

- No edges that connect a vertex to itself, i.e. no “length 1 loops”.
- No two edges that connect the same vertices, i.e. no “parallel edges”.

Green graph below is simple, pink graphs are not.







# Graph Definition

A simple graph is a graph with:

- No edges that connect a vertex to itself, i.e. no “loops”.
- No two edges that connect the same vertices, i.e. no “parallel edges”.

In 61B, **unless otherwise explicitly stated, all graphs will be simple.**

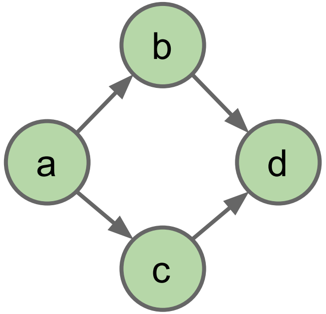
- In other words, when we say “graph”, we mean “simple graph.”

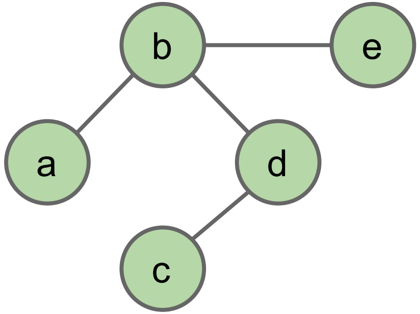
## Graph Types

Directed

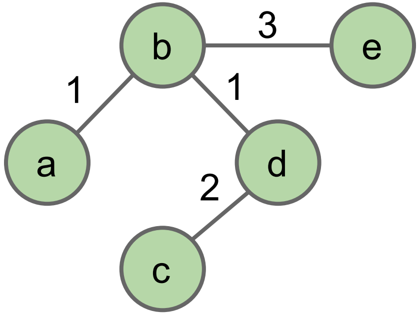
Undirected

Acyclic:

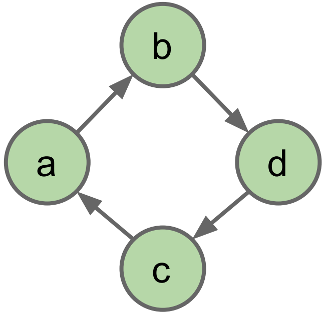


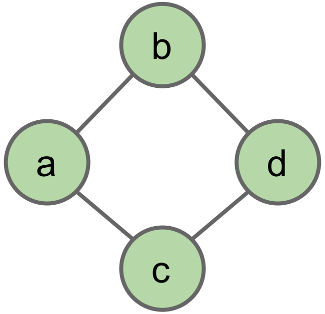




With Edge Labels




Cyclic:



# Graph Terminology

- Graph:
  - Set of ***vertices***, a.k.a. ***nodes***.
  - Set of ***edges***: Pairs of vertices.
  - Vertices with an edge between are ***adjacent***.
  - Optional: Vertices or edges may have ***labels*** (or ***weights***).
- A ***path*** is a sequence of vertices connected by edges.
  - A ***simple path*** is a path without repeated vertices.
- A ***cycle*** is a path whose first and last vertices are the same.
  - A graph with a cycle is 'cyclic'.
- Two vertices are ***connected*** if there is a path between them. If all vertices are connected, we say the graph is connected.

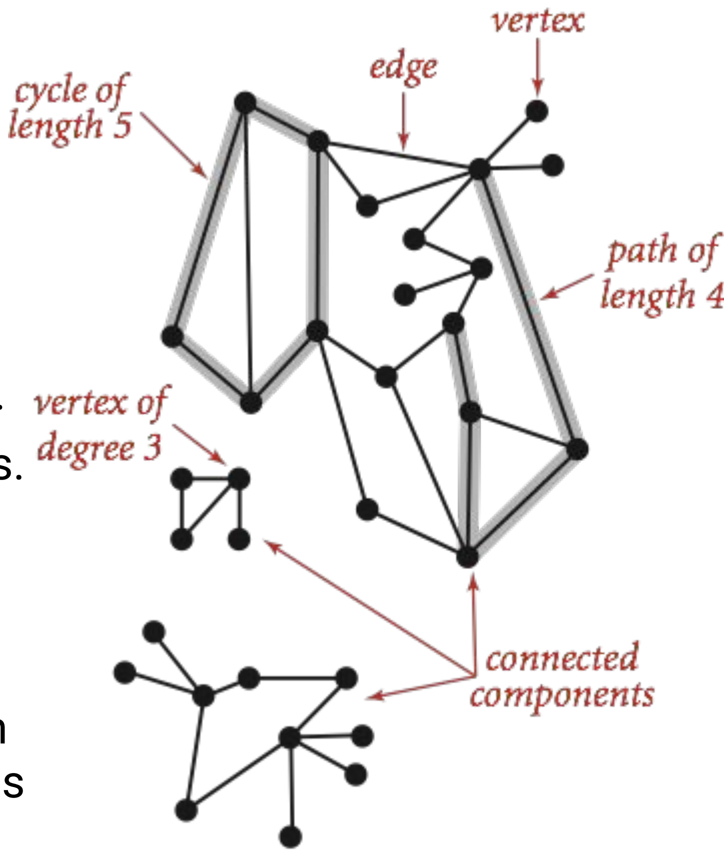
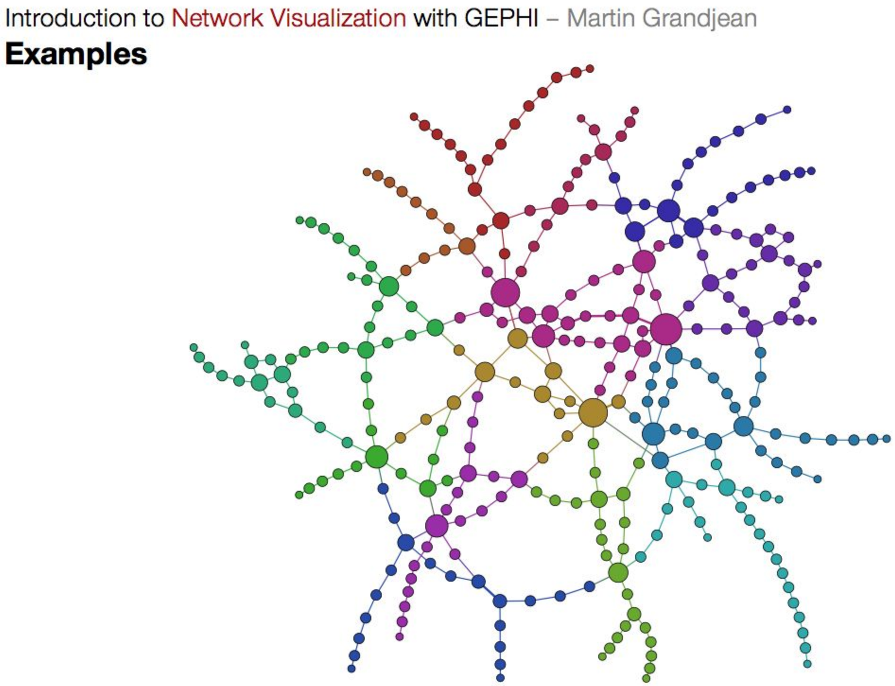



Figure from Algorithms 4th Edition

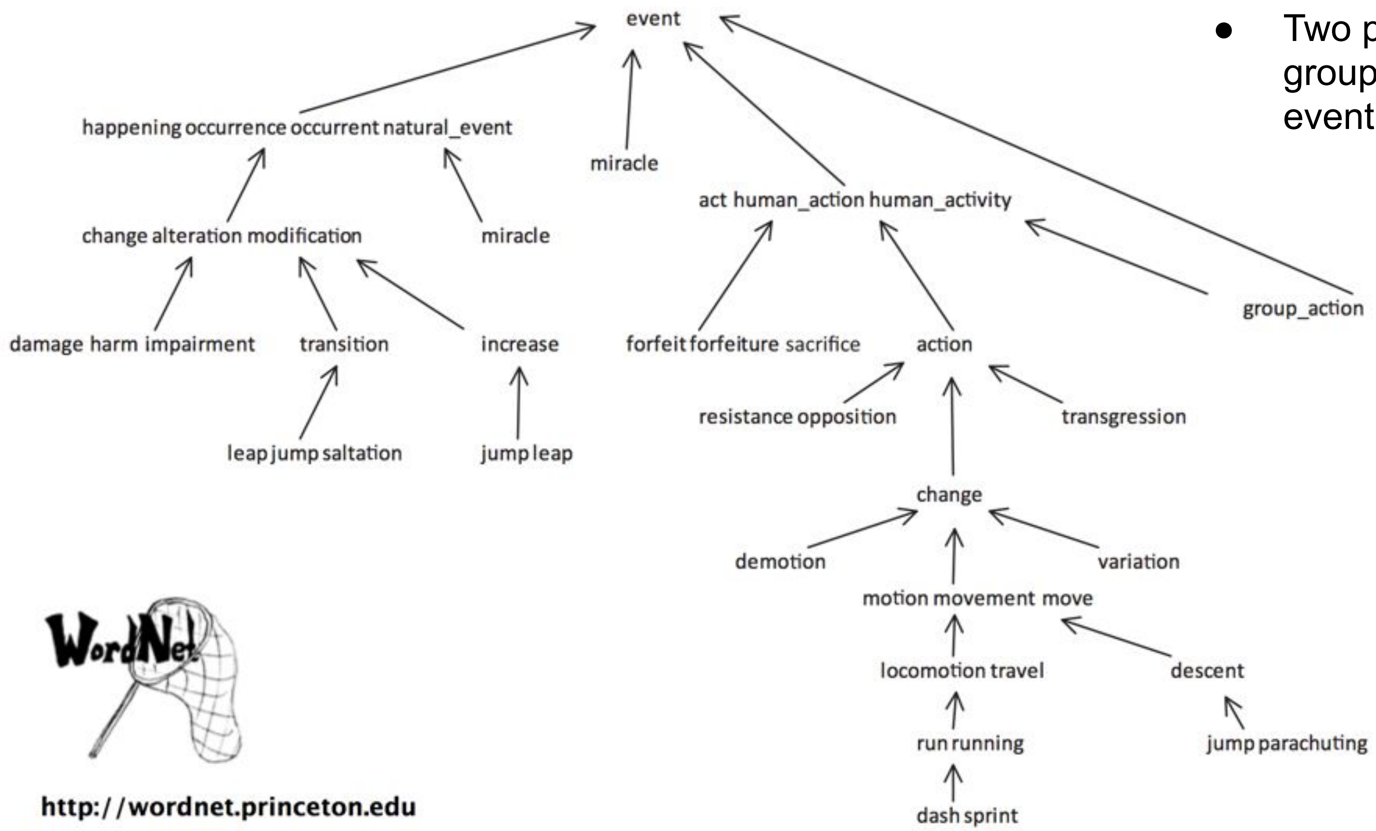
# Graph Example: The Paris Metro

This schematic map of the Paris Metro is a graph:

- Undirected
- Connected
- Cyclic (not a tree!)
- Vertex-labeled (each has a color).



# Directed Graph Example




http://wordnet.princeton.edu

Not a tree!

- Two paths from group_action to event.

Edge captures ‘is-a-type-of’ relationship. Example: descent is-a-type-of movement.

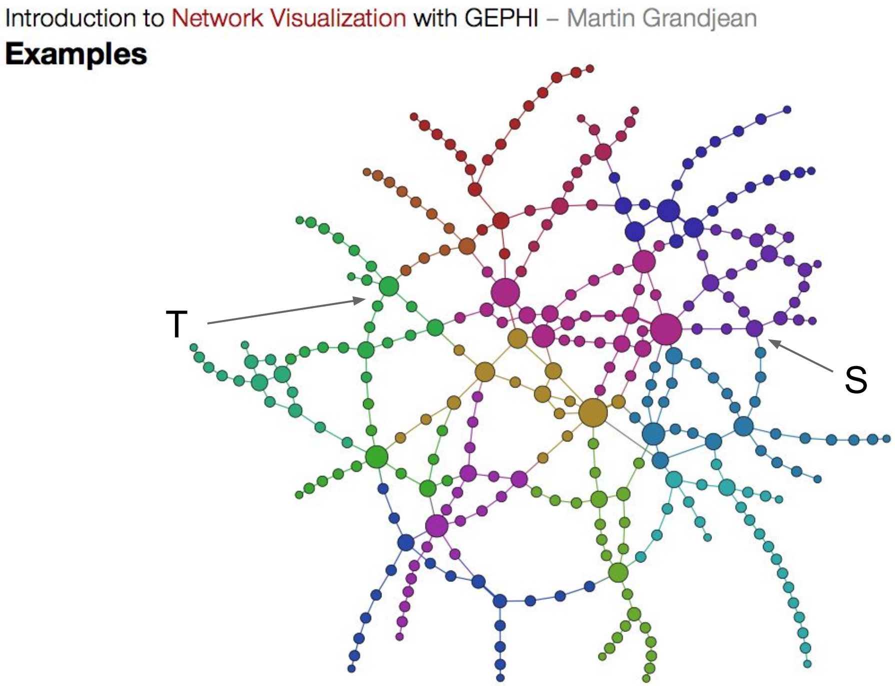
# Some Famous Graph Problems

Lecture 22, CS61B, Spring 2024

## Graph Queries

There are lots of interesting questions we can ask about a graph:

- What is the shortest route from S to T? What is the longest without cycles?
- Are there cycles?
  - Is there a tour you can take that only uses each node (station) exactly once?
  - Is there a tour that uses each edge exactly once?



# Graph Queries More Theoretically

Some well known graph problems and their common names:

- **s-t Path**. Is there a path between vertices s and t?
- **Connectivity.** Is the graph connected, i.e. is there a path between all vertices?
- **Biconnectivity.** Is there a vertex whose removal disconnects the graph?
- **Shortest s-t Path.** What is the shortest path between vertices s and t?
- **Cycle Detection.** Does the graph contain any cycles?
- **Euler Tour.** Is there a cycle that uses every edge exactly once?
- **Hamilton Tour.** Is there a cycle that uses every vertex exactly once?
- **Planarity**. Can you draw the graph on paper with no crossing edges?
- **Isomorphism**. Are two graphs isomorphic (the same graph in disguise)?

Often can't tell how difficult a graph problem is without very deep consideration.

# Graph Problem Difficulty

Some well known graph problems:

- **Euler Tour.** Is there a cycle that uses every edge exactly once?
- **Hamilton Tour.** Is there a cycle that uses every vertex exactly once?

Difficulty can be deceiving.

- An efficient Euler tour algorithm O(# edges) was found as early as 1873 [Link].
- Despite decades of intense study, no efficient algorithm for a Hamilton tour exists. Best algorithms are exponential time.

Graph problems are among the most mathematically rich areas of CS theory.

# Motivation for Graph Traversals: s-t Connectivity

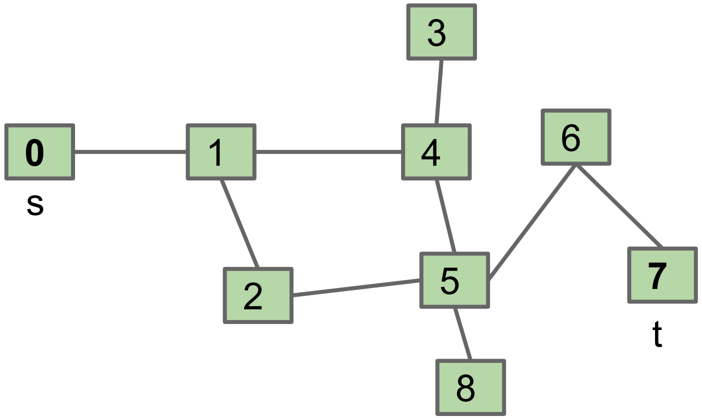
Lecture 22, CS61B, Spring 2024

# s-t Connectivity

Let's solve a classic graph problem called the s-t connectivity problem.

- Given source vertex s and a target vertex t, is there a path between s and t?

Requires us to traverse the graph somehow.



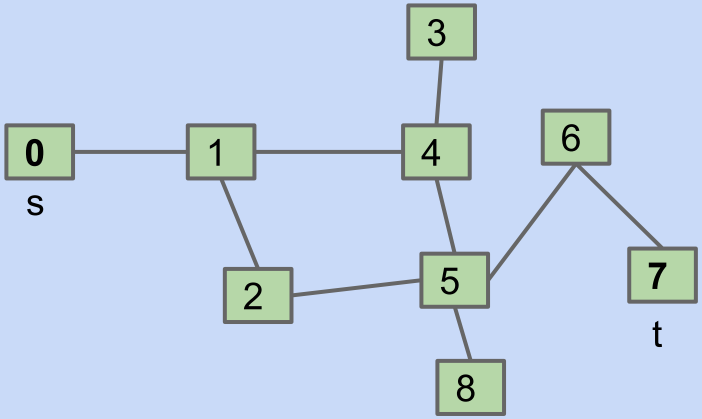
# s-t Connectivity

Let's solve a classic graph problem called the s-t connectivity problem.

- Given source vertex s and a target vertex t, is there a path between s and t?

Requires us to traverse the graph somehow.

- Try to come up with an algorithm for connected(s, t).

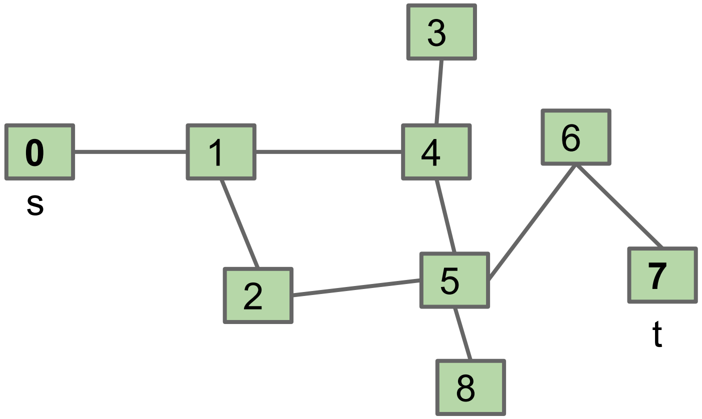


# s-t Connectivity

One possible recursive algorithm for connected(s, t).

- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any neighbor v of s, return true.
- Return false.

What is wrong with the algorithm above?

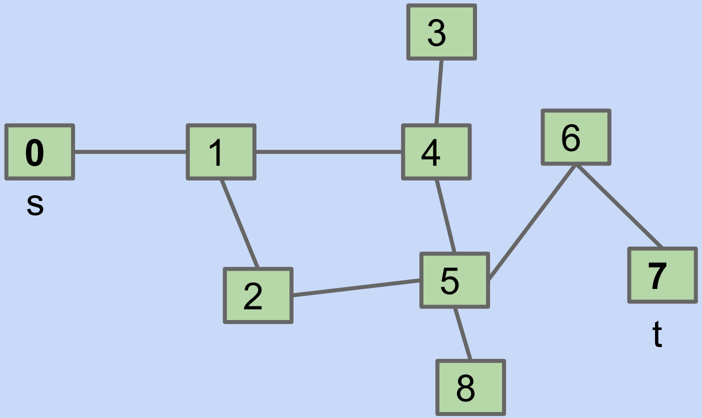


# s-t Connectivity

One possible recursive algorithm for connected(s, t).

- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any neighbor v of s, return true.
- Return false.

What is wrong with the algorithm above?



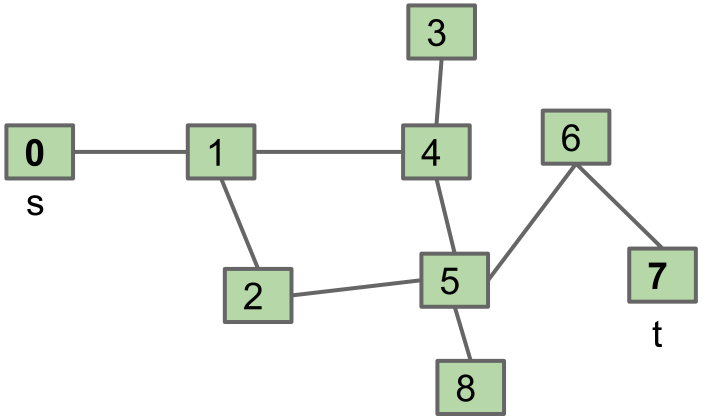
## s-t Connectivity

One possible recursive algorithm for connected(s, t).

- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any neighbor v of s, return true.
- Return false.

What is wrong with it? Can get caught in an infinite loop. Example:

- connected(0, 7):
  - Does 0 == 7? No, so...
  - if (connected(1, 7)) return true;
- connected(1, 7):
  - Does 1 == 7? No, so...
  - If (connected(0, 7)) … ← Infinite loop.



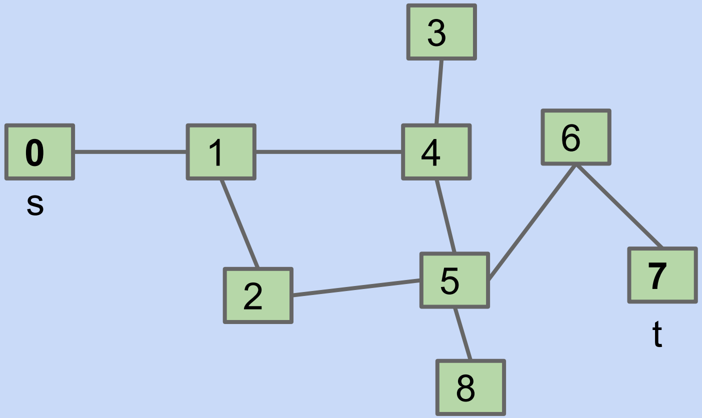
# s-t Connectivity

One possible recursive algorithm for connected(s, t).

- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any neighbor v of s, return true.
- Return false.

What is wrong with it? Can get caught in an infinite loop.

- How do we fix it?



# Depth First Search

Lecture 22, CS61B, Spring 2024

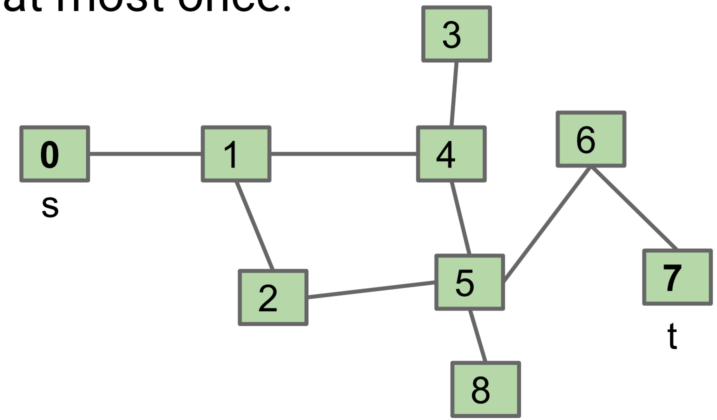
# s-t Connectivity

One possible recursive algorithm for connected(s, t).

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Basic idea is same as before, but visit each vertex at most once.

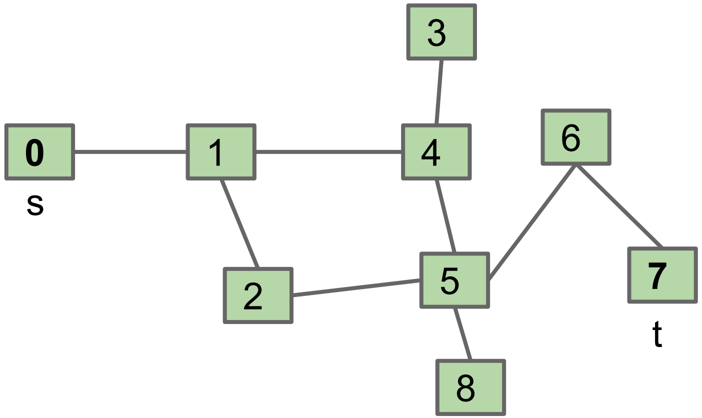
- Marking nodes prevents multiple visits.



## Demo: s-t Connectivity

connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.



# Demo: s-t Connectivity

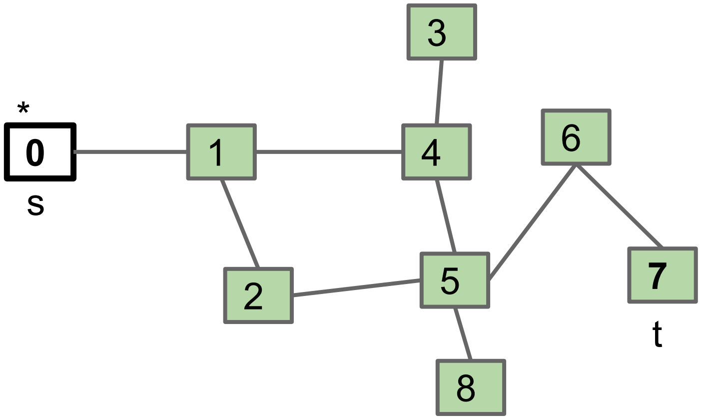
connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0

mark(0).
Is 0 == 7? No.

isMarked(1)? No.
- Check connected(1, 7).



# Demo: s-t Connectivity

connected(s, t):

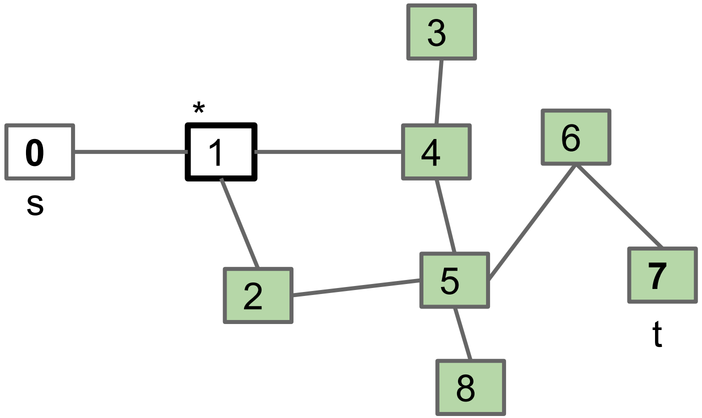
- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1

mark(1).
Is 1 == 7? No.

isMarked(0)? Yes.
isMarked(2)?

- Check connected(2, 7).



# Demo: s-t Connectivity

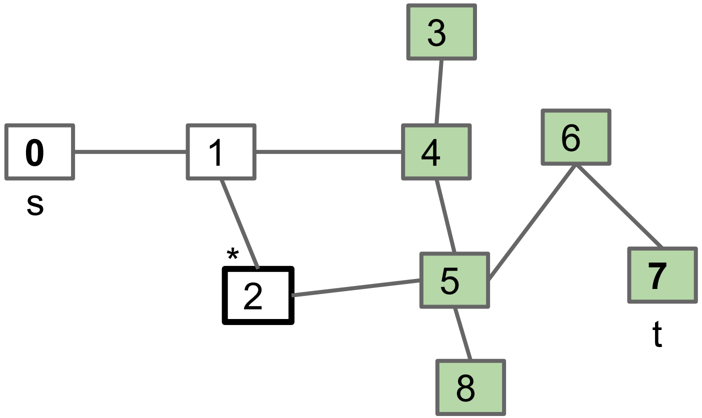
connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2

mark(2).
Is 2 == 7? No.

isMarked(1)? Yes.
isMarked(5)?
- Check connected(5, 7).



# Demo: s-t Connectivity

connected(s, t):

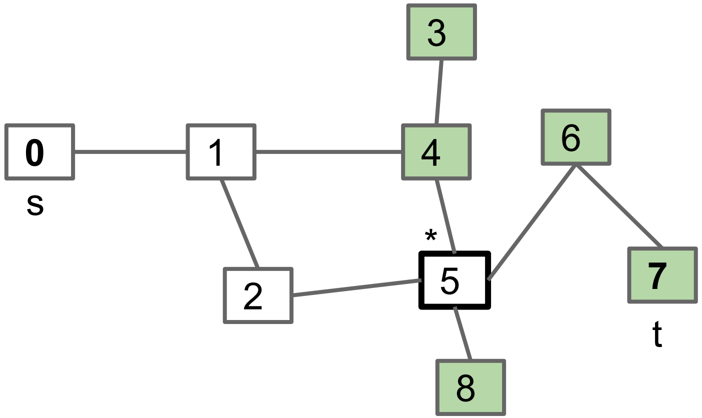
- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2 → 5

mark(5).
Is 5 == 7? No.

isMarked(2)? Yes.
isMarked(4)?

- Check connected(4, 7).



# Demo: s-t Connectivity

connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

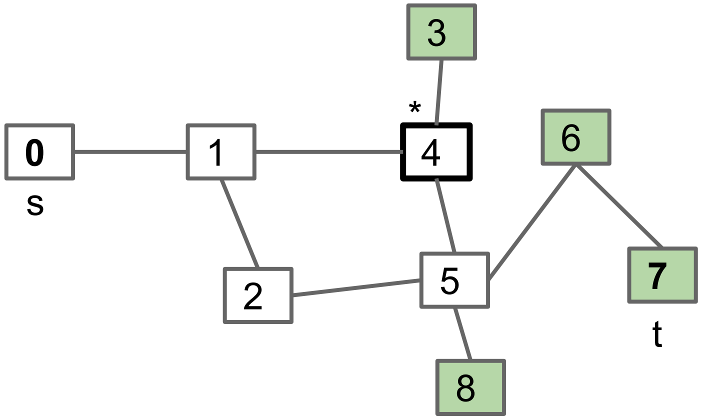
Call stack: 0 → 1 → 2 → 5 → 4

mark(4).
Is 4 == 7? No.

isMarked(1)? Yes.

isMarked(3)? No.

- Check connected(3, 7).



# Demo: s-t Connectivity

connected(s, t):

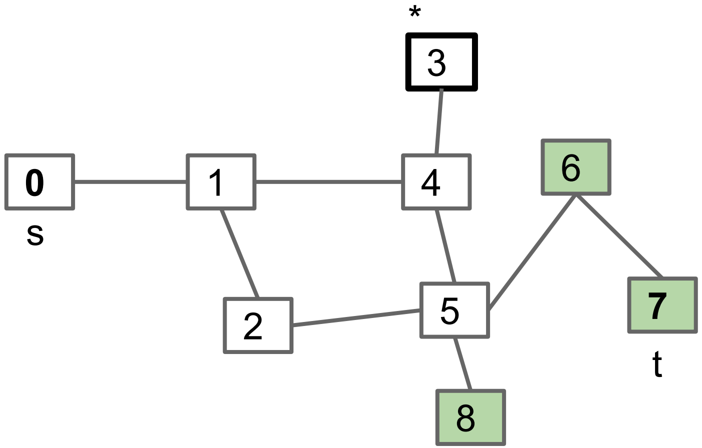
- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2 → 5 → 4 → 3

mark(3).
Is 3 == 7? No.

isMarked(4)? Yes.

No more neighbors! Return false.



# Demo: s-t Connectivity

connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2 → 5 → 4

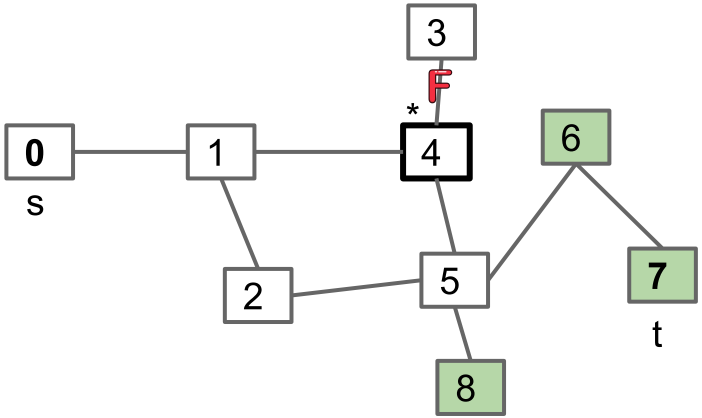
mark(4).
Is 4 == 7? No.

isMarked(1)? Yes.

isMarked(3)? No.

- Check connected(3, 7). Answer was false.

isMarked(5)? Yes.
No more neighbors, so return false.



# Demo: s-t Connectivity

connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2 → 5

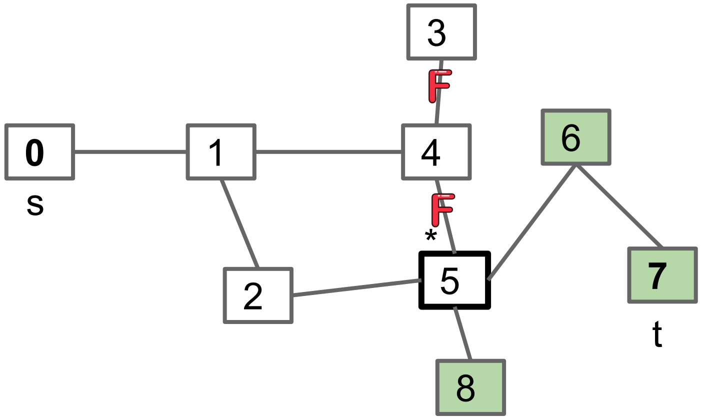
mark(5).
Is 5 == 7? No.

isMarked(2)? Yes.
isMarked(4)?

- Check connected(4, 7). Answer was false, so keep checking neighbors.

isMarked(6)?

- Check connected(6, 7).



# Demo: s-t Connectivity

connected(s, t):

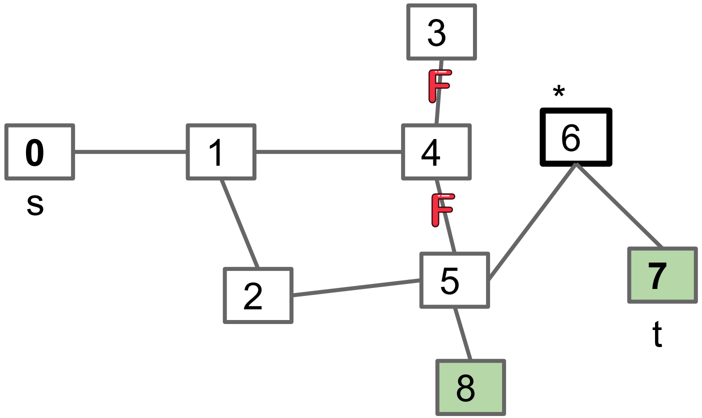
- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2 → 5 → 6

mark(6).
Is 6 == 7? No.

isMarked(5)? Yes.
isMarked(7)? No.

- Check connected(7, 7).



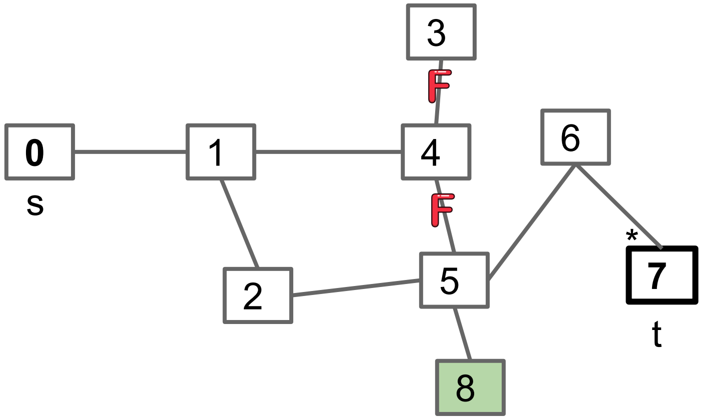
# Demo: s-t Connectivity

connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2 → 5 → 6 → 7

mark(7).
Is 7 == 7? Yes. Return true!



# Demo: s-t Connectivity

connected(s, t):

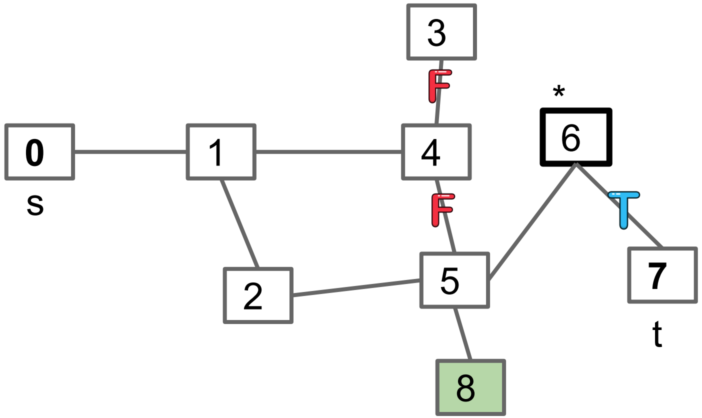
- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2 → 5 → 6

mark(6).
Is 6 == 7? No.

isMarked(5)? Yes.
isMarked(7)? No.

- Check connected(7, 7). Answer was true, so return true.



## Demo: s-t Connectivity

connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2 → 5

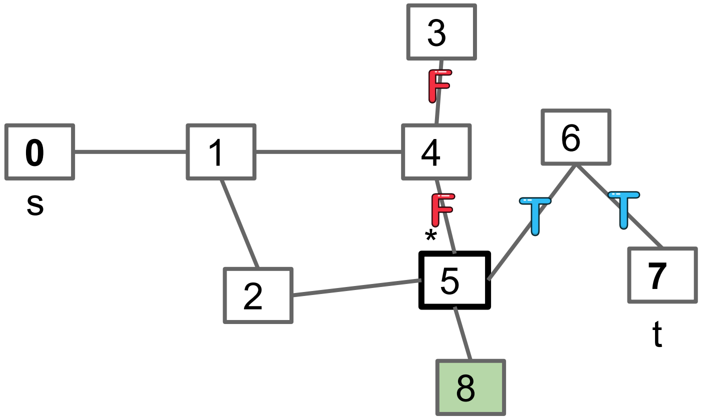
mark(5).
Is 5 == 7? No.

isMarked(2)? Yes.
isMarked(4)?

- Check connected(4, 7). Answer was false, so keep checking neighbors.

isMarked(5)? Yes.
isMarked(6)?

- Check connected(6, 7): Return true!



# Demo: s-t Connectivity

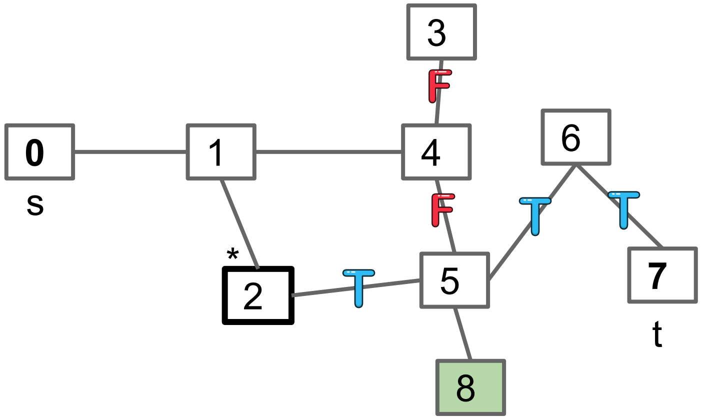
connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1 → 2

mark(2).
Is 2 == 7? No.

isMarked(1)? Yes.
isMarked(5)?
- Check connected(5, 7). Answer was true, so return true!



# Demo: s-t Connectivity

connected(s, t):

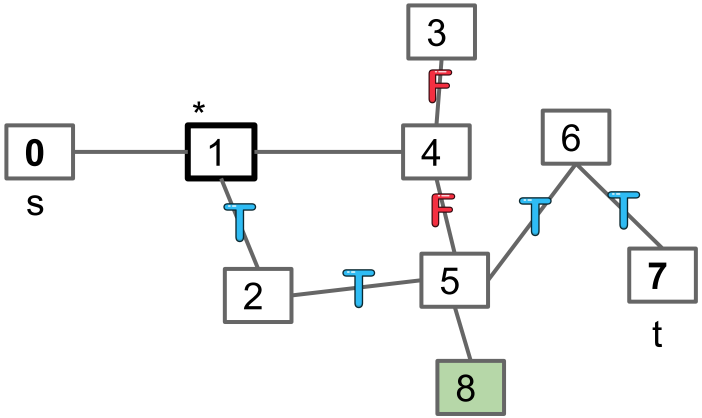
- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0 → 1

mark(1).
Is 1 == 7? No.

isMarked(0)? Yes.
isMarked(2)?

- Check connected(2, 7). Answer was true, so return true!



# Demo: s-t Connectivity

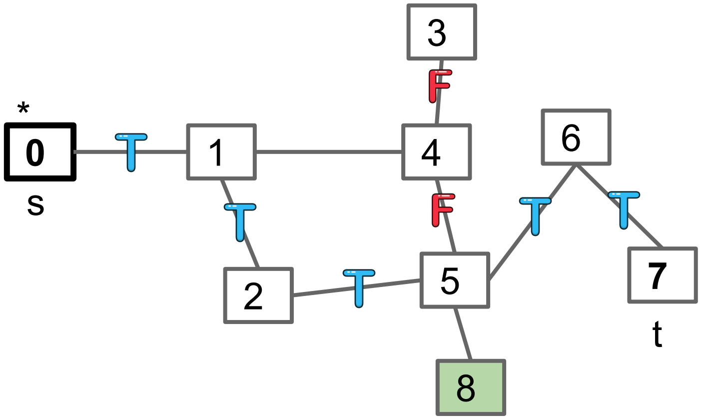
connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.

Call stack: 0

mark(0).
Is 0 == 7? No.

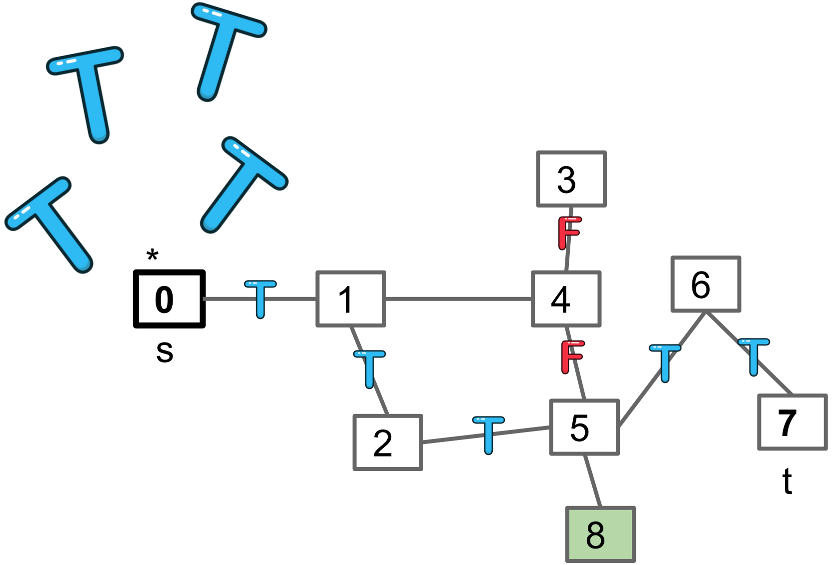
isMarked(1)? No.
- Check connected(1, 7). Answer was true, so return true!



## Demo: s-t Connectivity

connected(s, t):

- Mark s.
- Does s == t? If so, return true.
- Otherwise, if connected(v, t) for any unmarked neighbor v of s, return true.
- Return false.




mark(0).
Is 0 == 7? No.

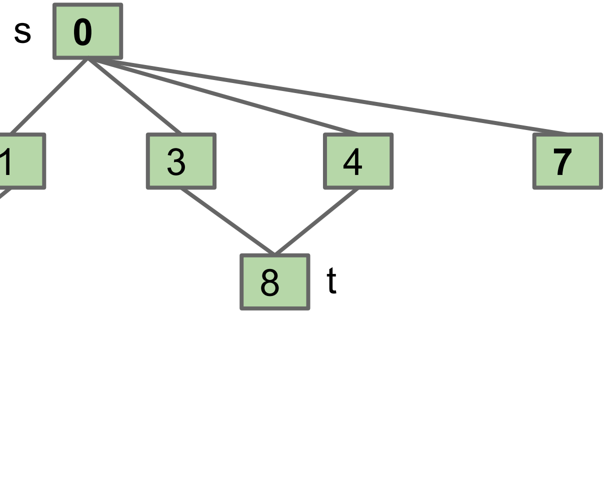
isMarked(1)? No.

- Check connected(1, 7). Answer was true, so return true!

# Depth First Traversal

This idea of exploring a neighbor's entire subgraph before moving on to the next neighbor is known as Depth First Traversal or Depth First Search.

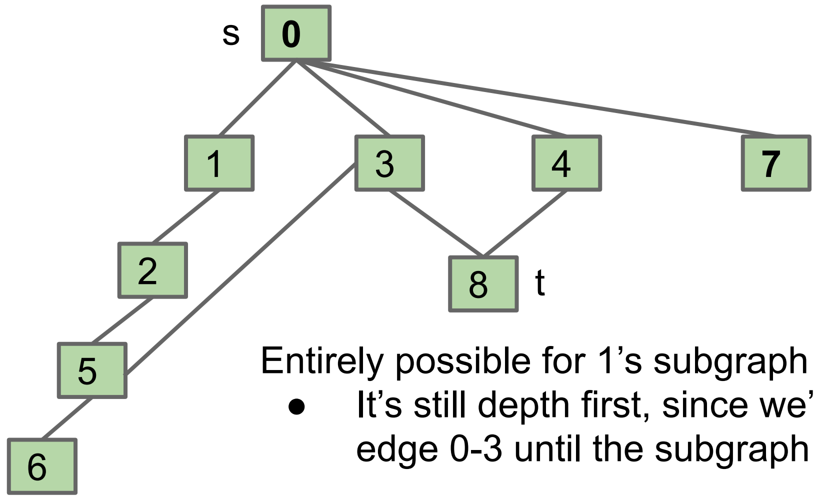
- Example: Explore 1's subgraph completely before using the edge 0-3.
- Called "depth first" because you go as deep as possible.



# Depth First Traversal

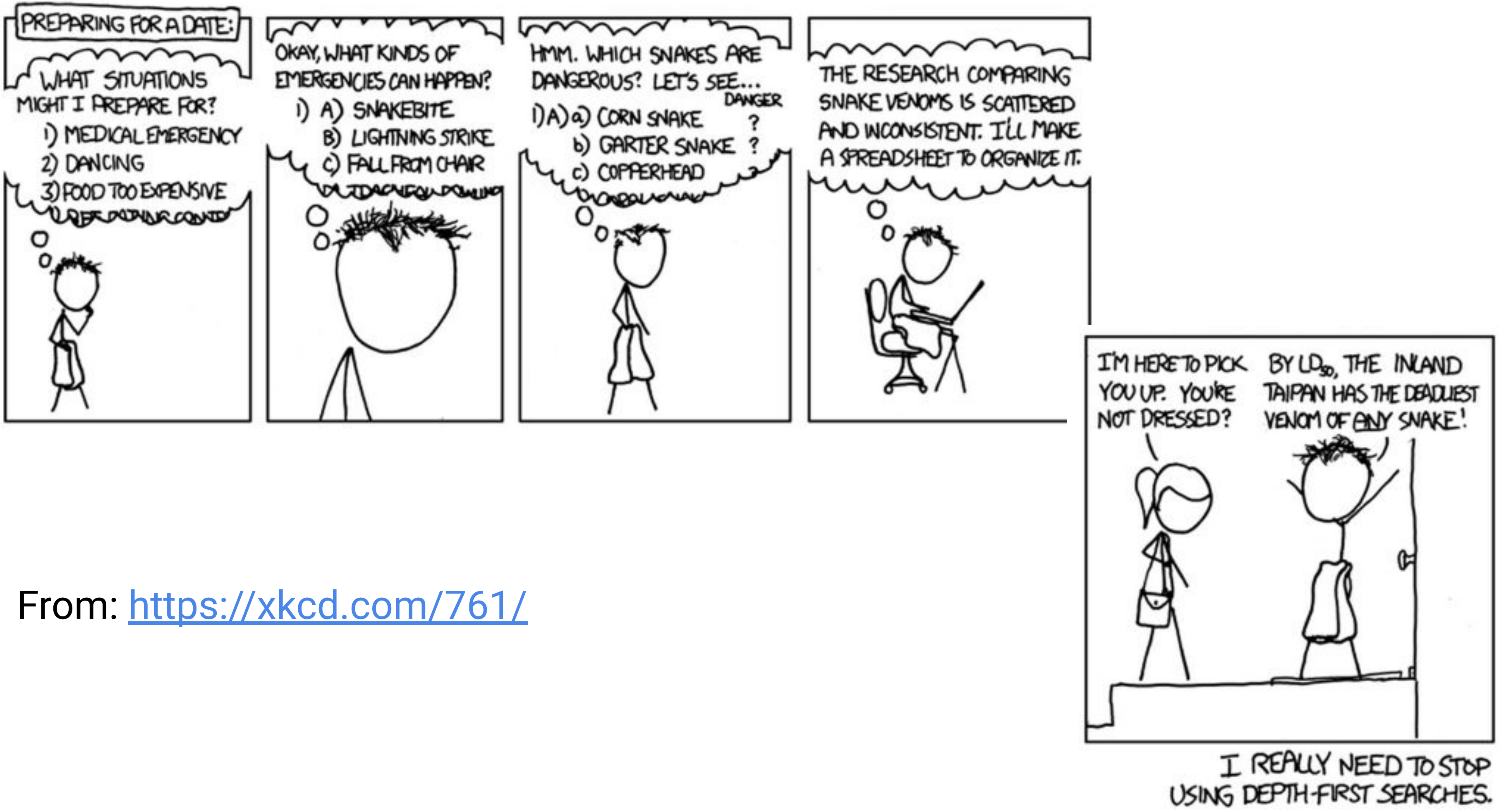
This idea of exploring a neighbor's entire subgraph before moving on to the next neighbor is known as Depth First Traversal.

- Example: Explore 1's subgraph completely before using the edge 0-3.
- Called "depth first" because you go as deep as possible.




Entirely possible for 1's subgraph to include 3!

- It's still depth first, since we're not using the edge 0-3 until the subgraph is explored.




From: https://xkcd.com/761/

# The Power of Depth First Search

DFS is a very powerful technique that can be used for many types of graph problems.

Another example:

- Let’s discuss an algorithm that computes a path to every vertex.
- Let’s call this algorithm DepthFirstPaths.
- Goal: Find a path from s to every other reachable vertex, visiting each vertex at most once.

# Demo: DepthFirstPaths

dfs(v):

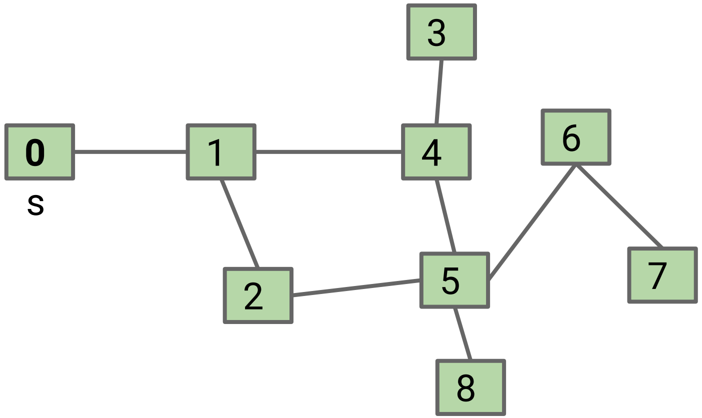
- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0)

Order of dfs calls: 0

| # | marked | edgeTo |
|---|---|---|
| 0 | F | - |
| 1 | F | - |
| 2 | F | - |
| 3 | F | - |
| 4 | F | - |
| 5 | F | - |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

Start by calling dfs(0).




Order of dfs returns:

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
    - set edgeTo[w] = v.
    - dfs(w)

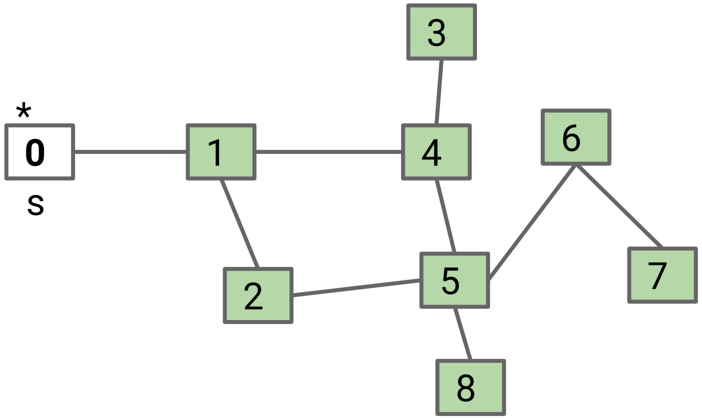
Call stack:
dfs(0)

Order of dfs calls: 01

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | F | 0 |
| 2 | F | - |
| 3 | F | - |
| 4 | F | - |
| 5 | F | - |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

dfs(0):
  mark(0).

  isMarked(1)? No.
  - edgeTo[1] = 0. **dfs(1).**




Order of dfs returns:

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

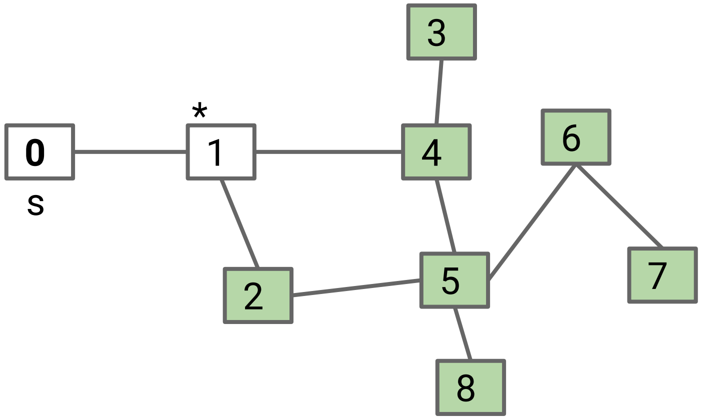
Call stack:
dfs(0) → dfs(1)

Order of dfs calls: 012

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | F | 1 |
| 3 | F | - |
| 4 | F | - |
| 5 | F | - |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

dfs(1):
  mark(1).

  isMarked(0)? Yes.
  isMarked(2)? No.
  - edgeTo[2] = 1. **dfs(2).**




Order of dfs returns:

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

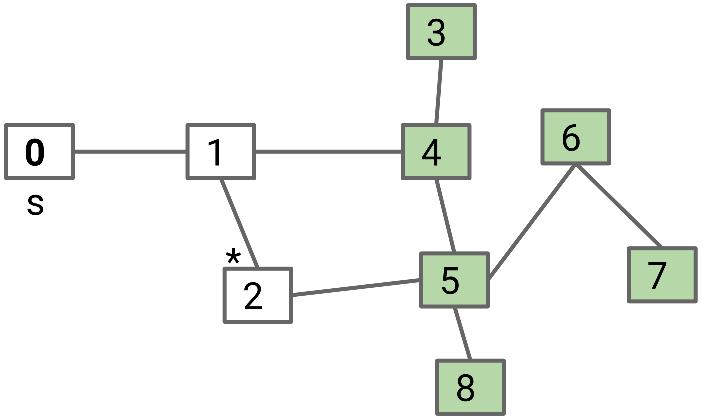
Call stack:
dfs(0) → dfs(1) → dfs(2)

Order of dfs calls: 0125

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | F | - |
| 4 | F | - |
| 5 | F | 2 |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

dfs(2):
mark(2).

isMarked(1)? Yes.
isMarked(5)? No.
- edgeTo[5] = 2. **dfs(5).**




Order of dfs returns:

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5)

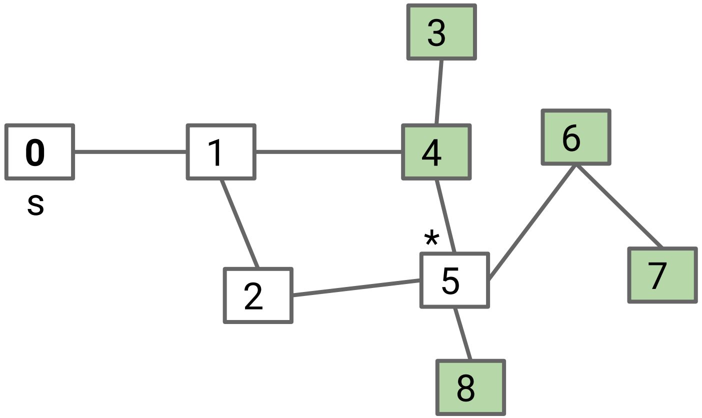
Order of dfs calls: 01254

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | F | - |
| 4 | F | 5 |
| 5 | T | 2 |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

dfs(5):
mark(5).

isMarked(2)? Yes.
isMarked(4)? No.

- edgeTo[4] = 5. **dfs(4).**




Order of dfs returns:

## Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
    - set edgeTo[w] = v.
    - dfs(w)

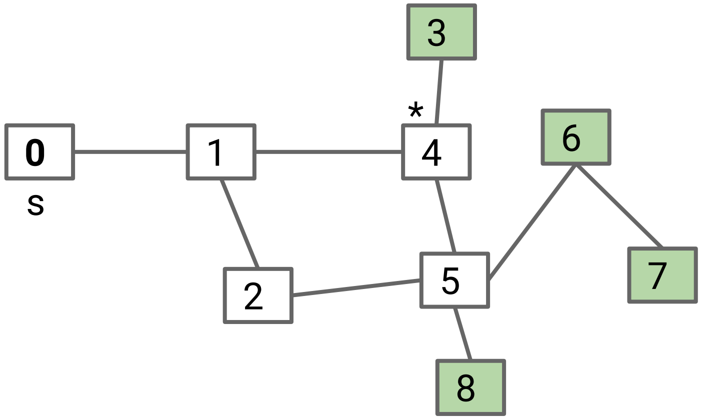
Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5) → dfs(4)

Order of dfs calls: 012543

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | F | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

dfs(4):
mark(4).

isMarked(1)? Yes.
isMarked(3)? No.
- edgeTo[3] = 4. **dfs(3).**




Order of dfs returns:

## Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5) → dfs(4) → dfs(3)

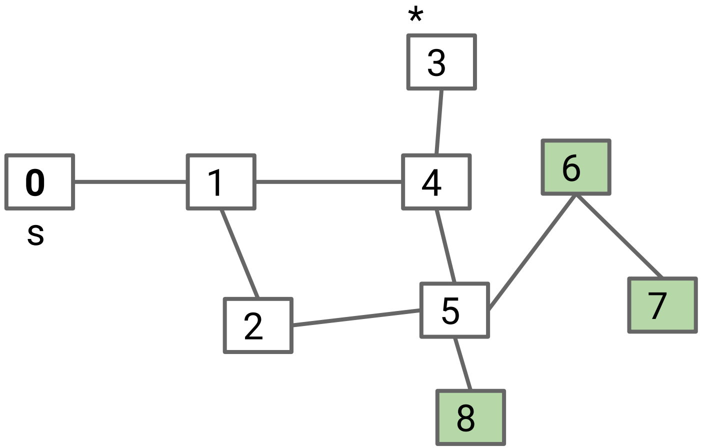
Order of dfs calls: 012543

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

dfs(3):
mark(3).

isMarked(4)? Yes.

No more children, so return.




Order of dfs returns: 3

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5) → dfs(4)

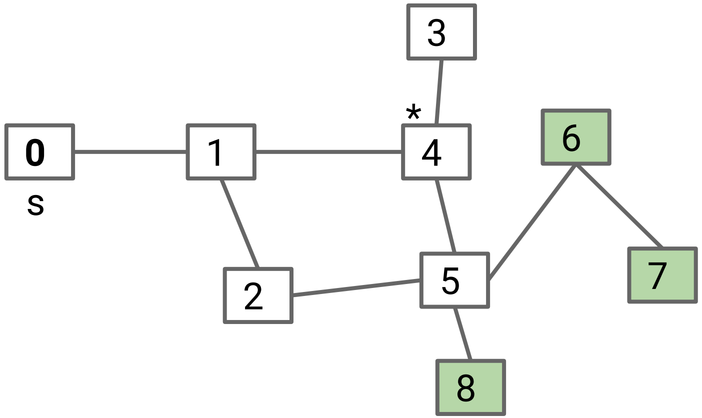
Order of dfs calls: 012543

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | F | - |
| 7 | F | - |
| 8 | F | - |

dfs(4):
mark(4).

isMarked(3)? No.
- edgeTo[3] = 4. dfs(3).

No more children, so return.




Order of dfs returns: 34

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) → dfs(5)

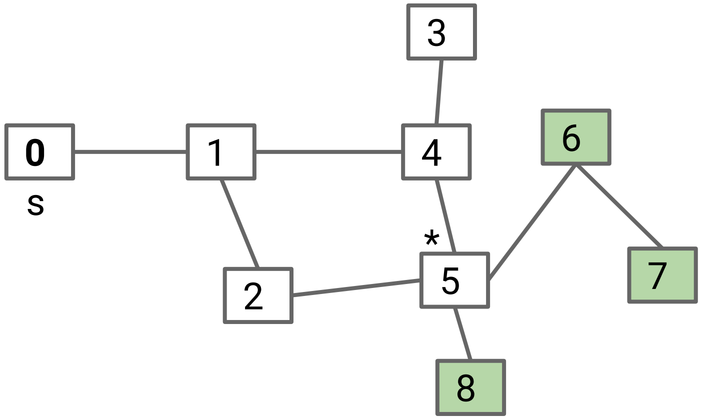
Order of dfs calls: 0125436

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | F | 5 |
| 7 | F | - |
| 8 | F | - |

dfs(5):
mark(5).

isMarked(2)? Yes.
isMarked(4)? No.
- edgeTo[3] = 4. dfs(4).

isMarked(6)? No.
- edgeTo[6] = 5. **dfs(6).**




Order of dfs returns: 34

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

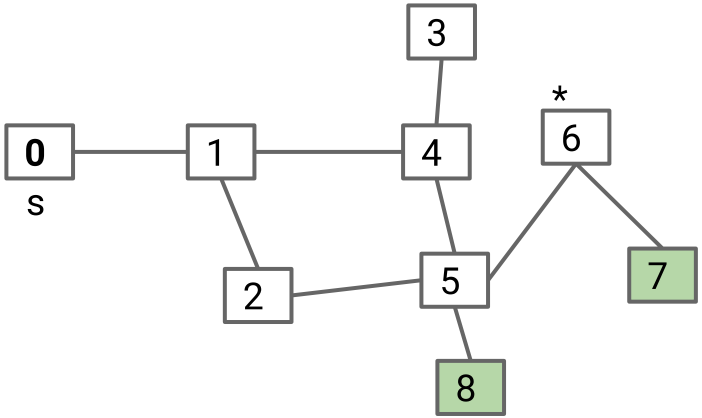
Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5) → dfs(6)

Order of dfs calls: 01254367

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | F | 6 |
| 8 | F | - |

dfs(6):
  mark(6).

  isMarked(5)? Yes.
  isMarked(7)? No.
  - edgeTo[7] = 6. **dfs(7).**




Order of dfs returns: 34

## Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5) → dfs(6) → dfs(7)

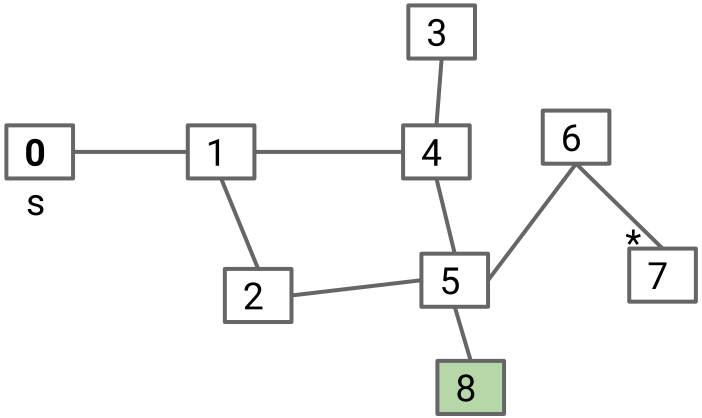
Order of dfs calls: 01254367

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | T | 6 |
| 8 | F | - |

dfs(7):
mark(7).

isMarked(6)? Yes.

No more children, so return.




Order of dfs returns: 347

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5) → dfs(6)

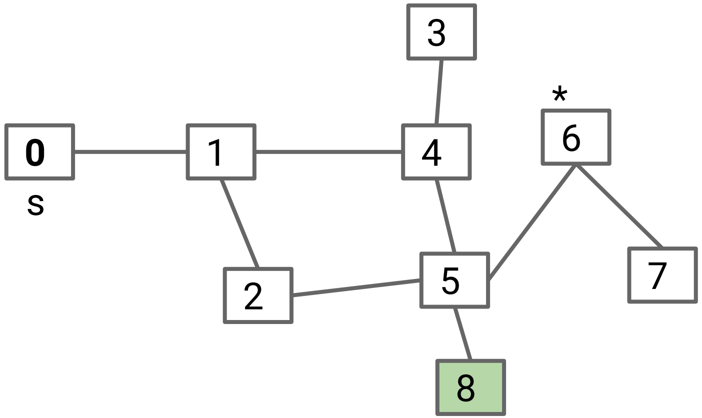
Order of dfs calls: 01254367

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | T | 6 |
| 8 | F | - |

dfs(6):
mark(6).

isMarked(5)? Yes.
isMarked(7)? No.
- edgeTo[7] = 6. dfs(7).

No more children, so return.




Order of dfs returns: 3476

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) → dfs(5)

Order of dfs calls: 012543678

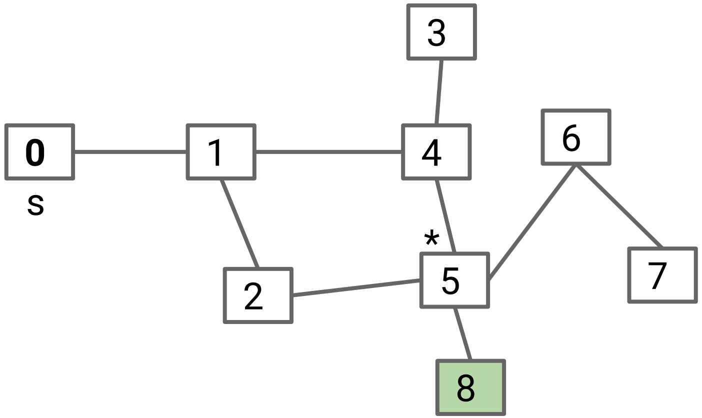
| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | T | 6 |
| 8 | F | 5 |

dfs(5):
mark(5).

isMarked(2)? Yes.
isMarked(4)? No.
- edgeTo[3] = 4. dfs(4).

isMarked(6)? No.
- edgeTo[6] = 5. dfs(6).

isMarked(8)? No.
- edgeTo[8] = 5. **dfs(8).**




Order of dfs returns: 3476

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5) → dfs(8)

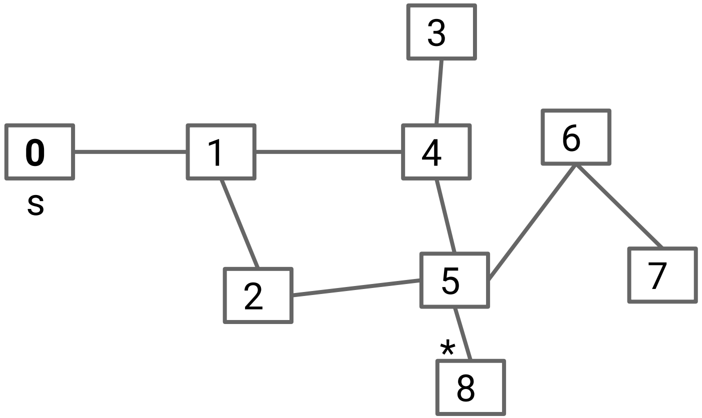
Order of dfs calls: 012543678

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | T | 6 |
| 8 | T | 5 |

dfs(8):
mark(8)

isMarked(5)? Yes.

No more children, so return.




Order of dfs returns: 34768

## Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2) →
dfs(5)

Order of dfs calls: 012543678

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | T | 6 |
| 8 | T | 5 |

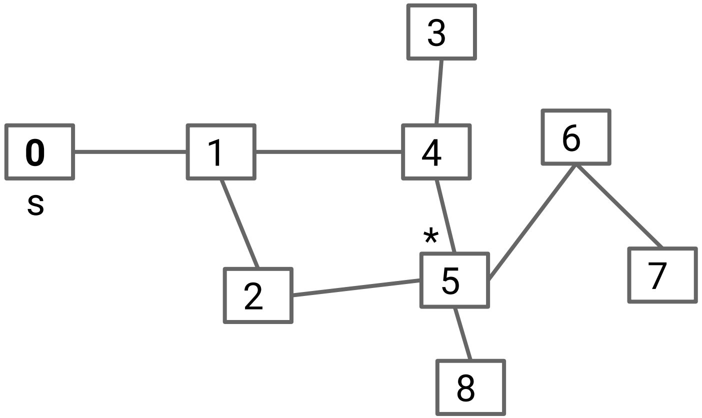
dfs(5):
mark(5).

isMarked(2)? Yes.
isMarked(4)? No.
- edgeTo[3] = 4. dfs(4).

isMarked(6)? No.
- edgeTo[6] = 5. dfs(6).

isMarked(8)? No.
- edgeTo[8] = 5. dfs(8)

No more children, so return.




Order of dfs returns: 347685

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1) → dfs(2)

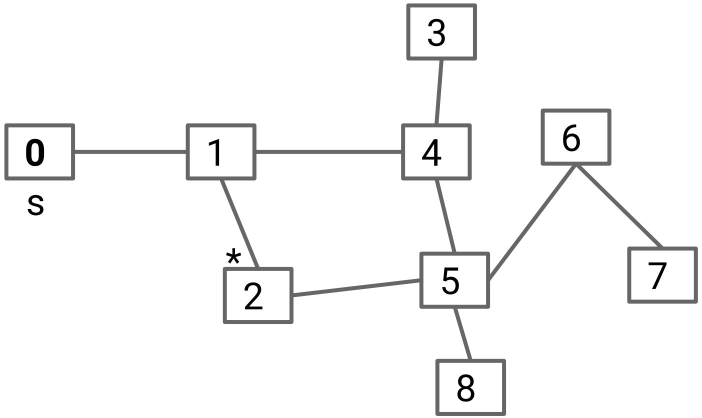
Order of dfs calls: 012543678

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | T | 6 |
| 8 | T | 5 |

dfs(2):
mark(2).

isMarked(1)? Yes.
isMarked(5)? No.
- edgeTo[5] = 2. dfs(5).

No more children, so return.




Order of dfs returns: 3476852

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0) → dfs(1)

Order of dfs calls: 012543678

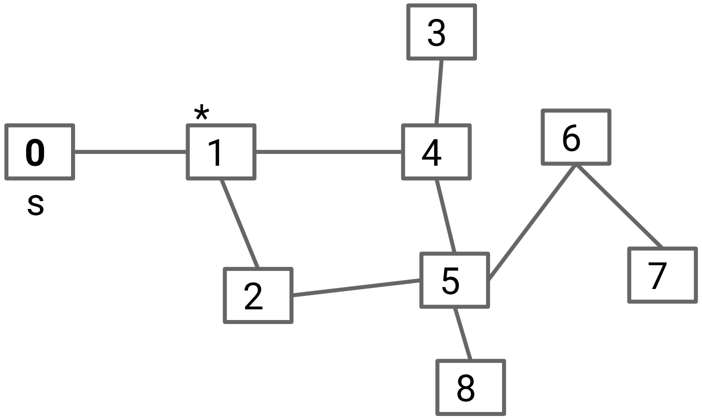
| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | T | 6 |
| 8 | T | 5 |

dfs(1):
mark(1).

isMarked(0)? Yes.
isMarked(2)? No.
- edgeTo[2] = 1. dfs(2).

isMarked(4)? Yes.

No more children, so return.




Order of dfs returns: 34768521

# Demo: DepthFirstPaths

dfs(v):

- Mark v.
- For each unmarked adjacent vertex w:
  - set edgeTo[w] = v.
  - dfs(w)

Call stack:
dfs(0)

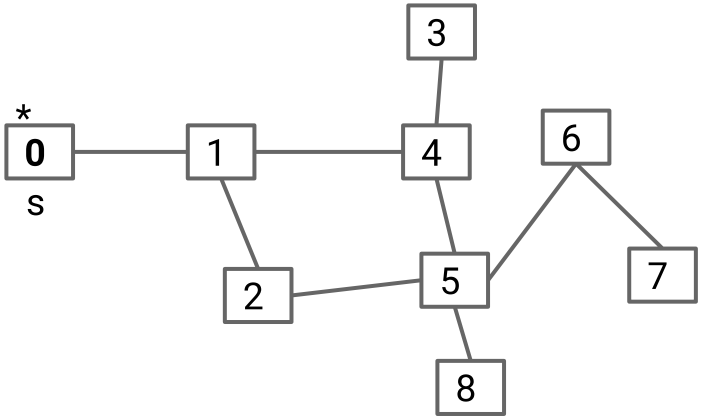
Order of dfs calls: 012543678

| # | marked | edgeTo |
|---|---|---|
| 0 | T | - |
| 1 | T | 0 |
| 2 | T | 1 |
| 3 | T | 4 |
| 4 | T | 5 |
| 5 | T | 2 |
| 6 | T | 5 |
| 7 | T | 6 |
| 8 | T | 5 |

dfs(0):
  mark(0).

  isMarked(1)? No.
  - edgeTo[1] = 0. dfs(1).

  No more children, so return.




Order of dfs returns: 347685210

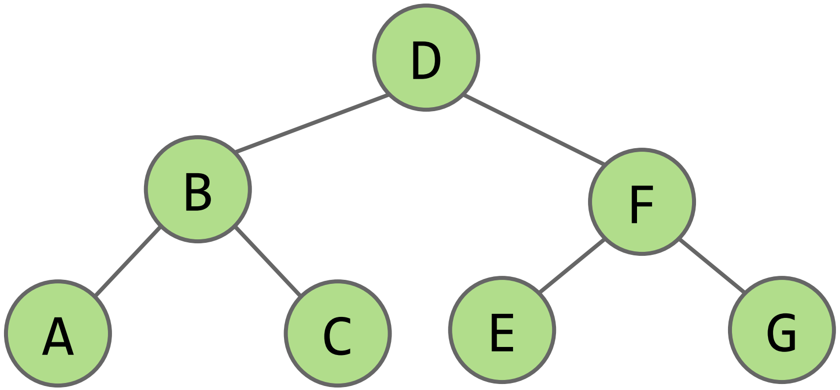
# Tree vs. Graph Traversals

Lecture 22, CS61B, Spring 2024

# Tree Traversals

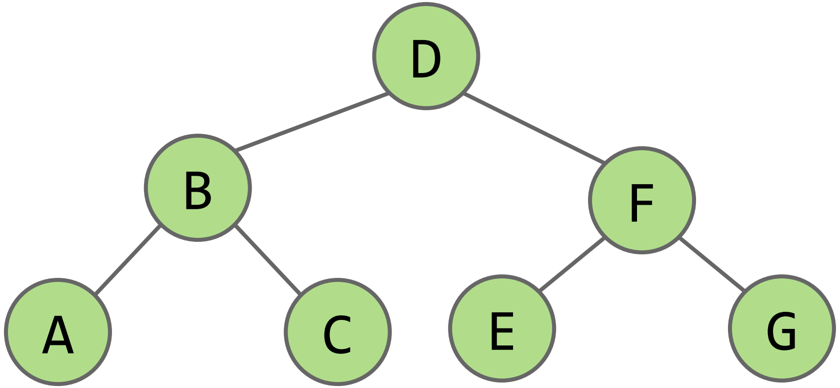
There are many tree traversals:

- Preorder: DBACFEG
- Inorder: ABCDEFG
- Postorder: ACBEGFD
- Level order: DBFACEG



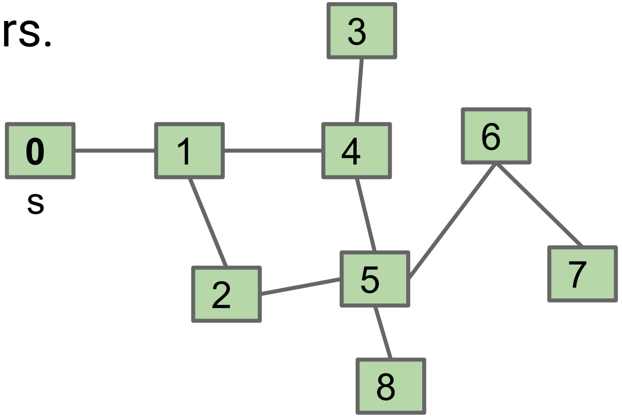
There are many tree traversals:

- Preorder: DBACFEG
- Inorder: ABCDEFG
- Postorder: ACBEGFD
- Level order: DBFACEG



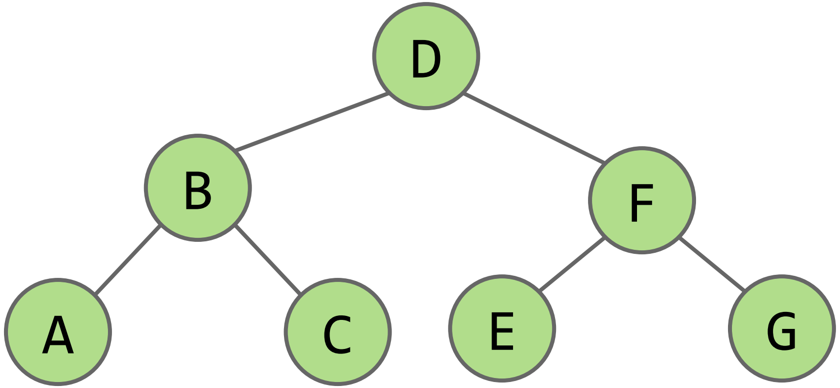

What we just did in DepthFirstPaths is called “**DFS Preorder**.”

- **DFS Preorder**: **Action** is **before DFS** calls to neighbors.
    - Our action was setting edgeTo.
    - Example: edgeTo[1] was set before DFS calls to neighbors 2 and 4.
- One valid DFS preorder for this graph: 012543678
    - Equivalent to the order of dfs calls.



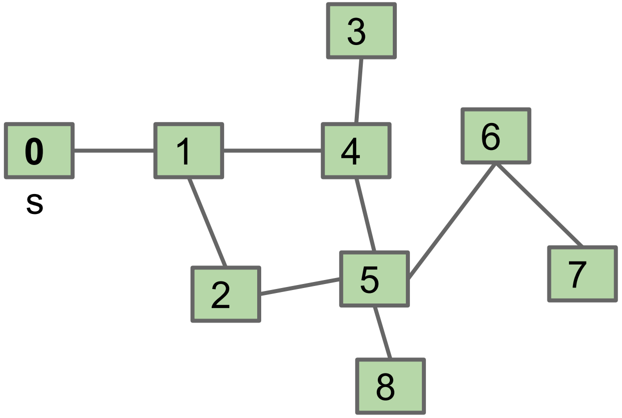
There are many tree traversals:

- Preorder: DBACFEG
- Inorder: ABCDEFG
- Postorder: ACBEGFD
- Level order: DBFACEG




Could also do actions in **DFS Postorder**.

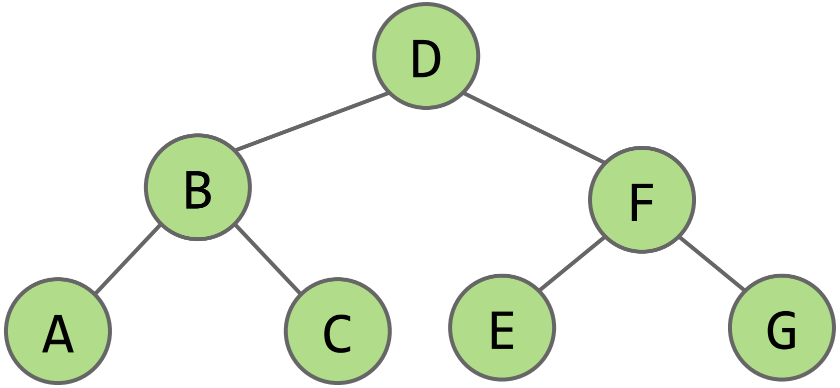
- **DFS Postorder: Action** is **after DFS** calls to neighbors.
- Example: dfs(s):
  - mark(s)
  - For each unmarked neighbor n of s, dfs(n)
  - print(s)
- Results for dfs(0) would be: 347685210
- Equivalent to the order of dfs returns.



## Graph Traversals

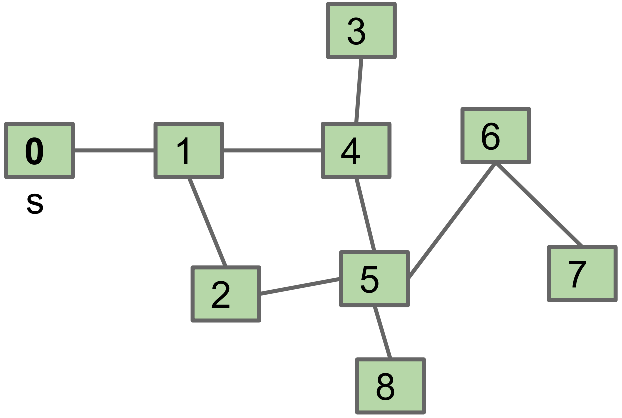
Just as there are many tree traversals:

- Preorder: DBACFEG
- Inorder: ABCDEFG
- Postorder: ACBEGFD
- Level order: DBFACEG




So too are there many graph traversals, given some source:

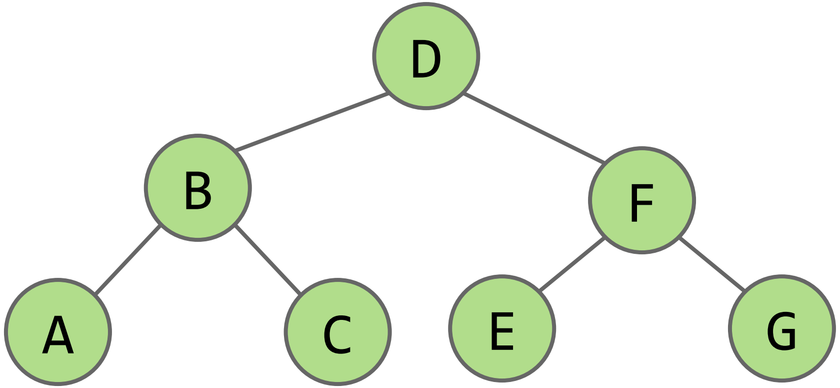
- DFS Preorder: 012543678 (dfs calls).
- DFS Postorder: 347685210 (dfs returns).



# Graph Traversals

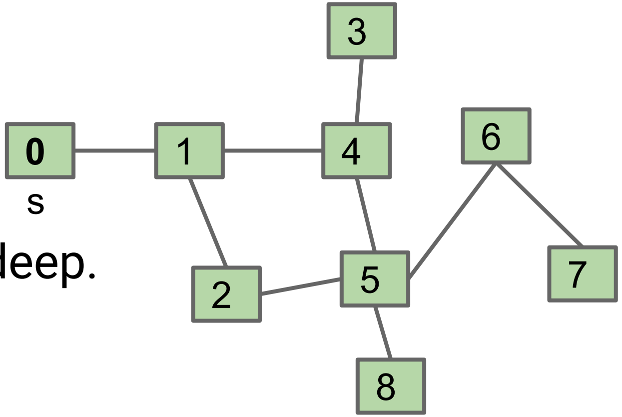
Just as there are many tree traversals:

- Preorder: DBACFEG
- Inorder: ABCDEFG
- Postorder: ACBEGFD
- Level order: DBFACEG




So too are there many graph traversals, given some source:

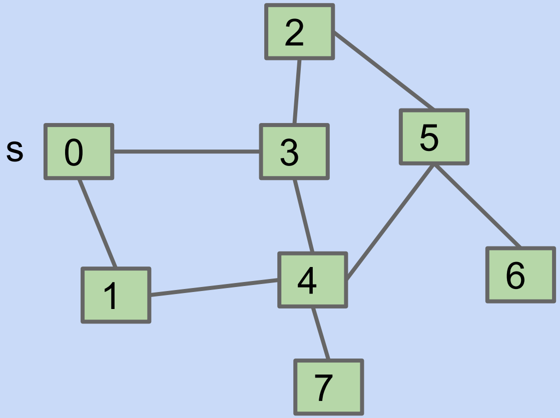
- DFS Preorder: 012543678 (dfs calls).
- DFS Postorder: 347685210 (dfs returns).
- BFS order: Act in order of distance from s.
  - BFS stands for “breadth first search”.
  - Analogous to “level order”. Search is wide, not deep.
  - 0 1 24 53 68 7



# Challenge: Invent Breadth First Search

Lecture 22, CS61B, Spring 2024

# Shortest Paths Challenge Before Next Lecture




Goal: Given the graph above, find the length of the shortest path from s to all other vertices.

- Give a general algorithm.
- Hint: You'll need to somehow visit vertices in BFS order.
- Hint #2: You'll need to use some kind of data structure.

Will discuss a solution in the next lecture.

## Summary

Graphs are a more general idea than a tree.

- A tree is a graph where there are no cycles and every vertex is connected.
- Key graph terms: Directed, Undirected, Cyclic, Acyclic, Path, Cycle.

Graph problems vary widely in difficulty.

- Common tool for solving almost all graph problems is traversal.
- A traversal is an order in which you visit / act upon vertices.
- Tree traversals:
  - Preorder, inorder, postorder, level order.
- Graph traversals:
  - DFS preorder, DFS postorder, BFS.
- By performing actions / setting instance variables during a graph (or tree) traversal, you can solve problems like s-t connectivity or path finding.