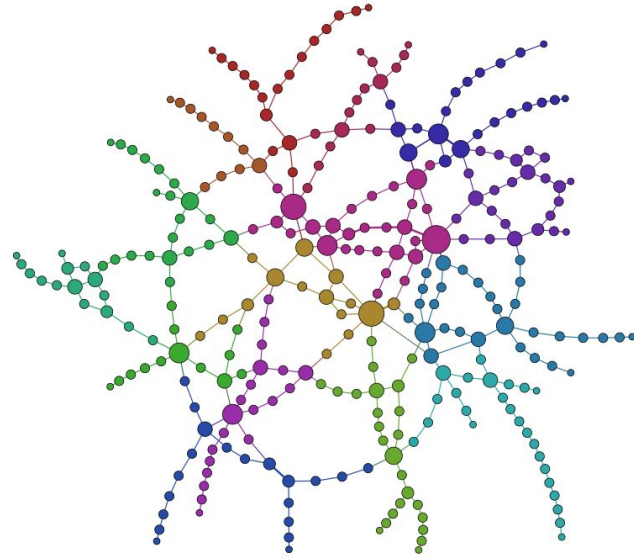


Examples



CS61B, 2021

Lecture 21: Graphs and Traversals

- Tree Traversals
- Graphs
- Depth First Search
- Breadth First Search

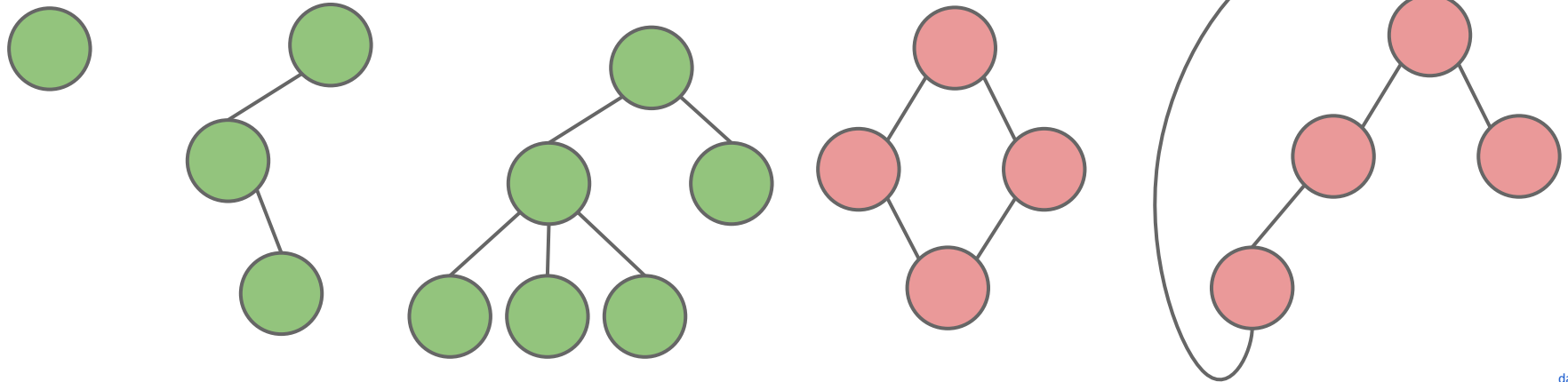
Trees and Traversals

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)

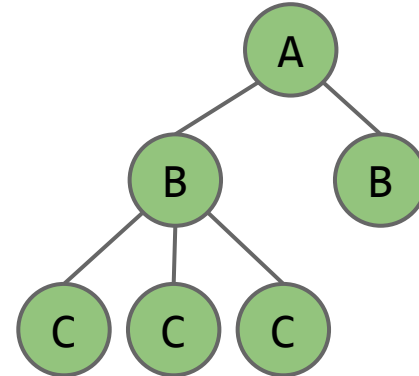
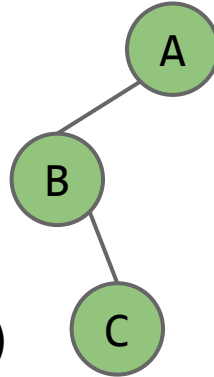
A rooted tree is a tree where we've chosen one node as the “root”.

- 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.



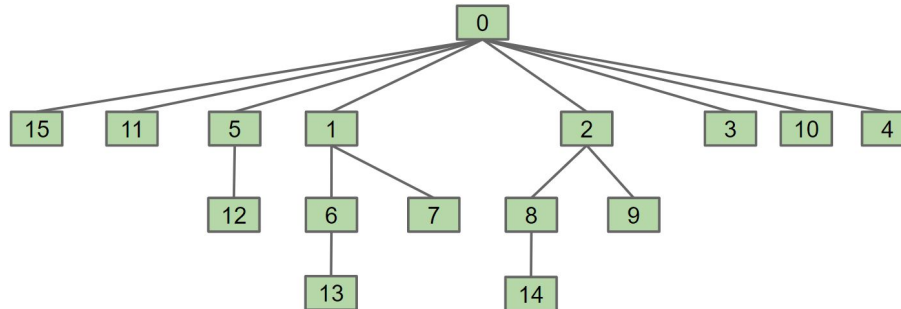
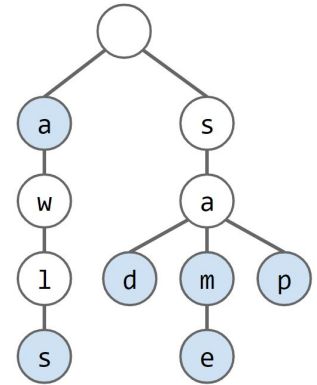
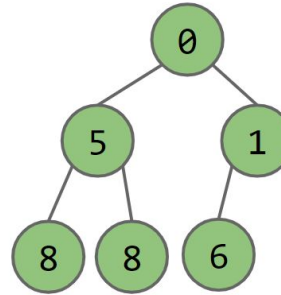
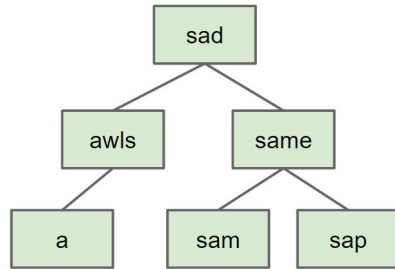
For each of these:

- 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

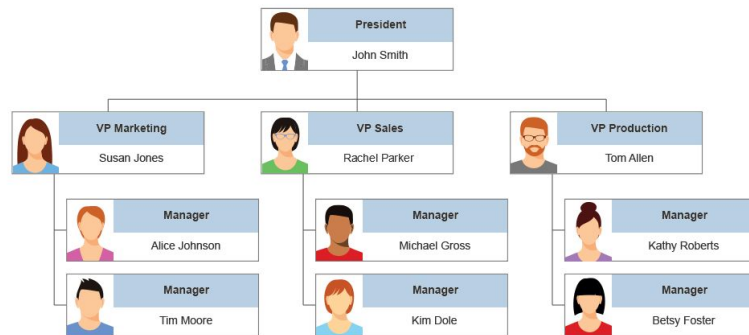
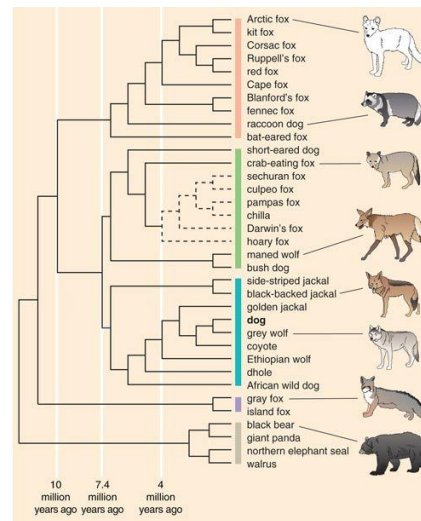
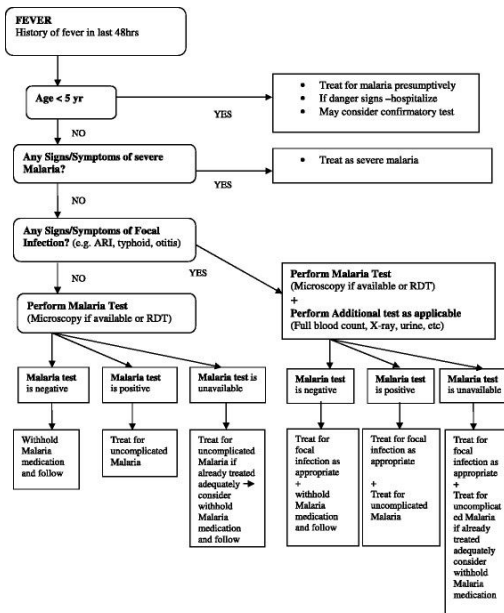
We've seen trees as nodes in a specific data structure implementation: Search Trees, Tries, Heaps, Disjoint Sets, etc.



Trees

Trees are a more general concept.

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

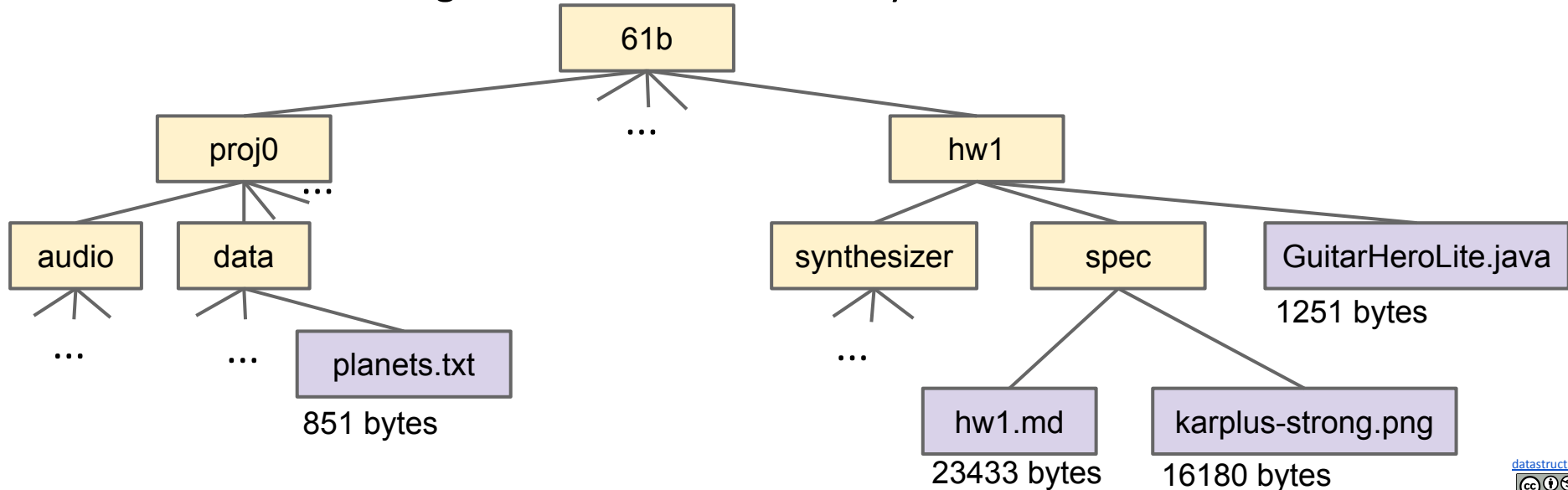


*: Not all family lineages are trees!

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.

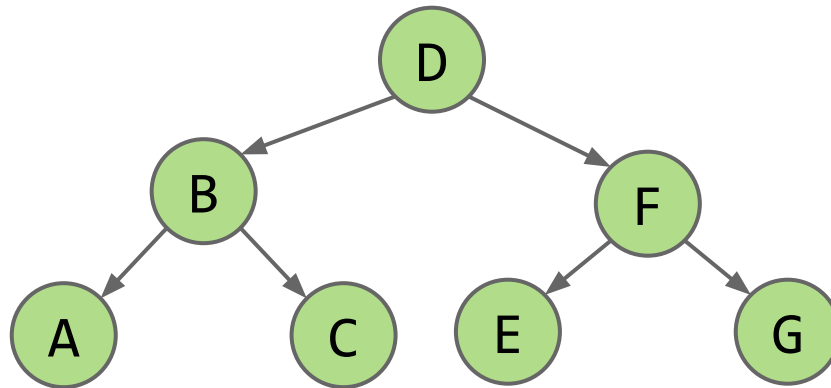
- What one might call “tree iteration” is actually called “tree traversal.”
- Unlike lists, there are many orders in which we might **visit** the nodes.
 - Each ordering is useful in different ways.



Tree Traversal Orderings

Level Order

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



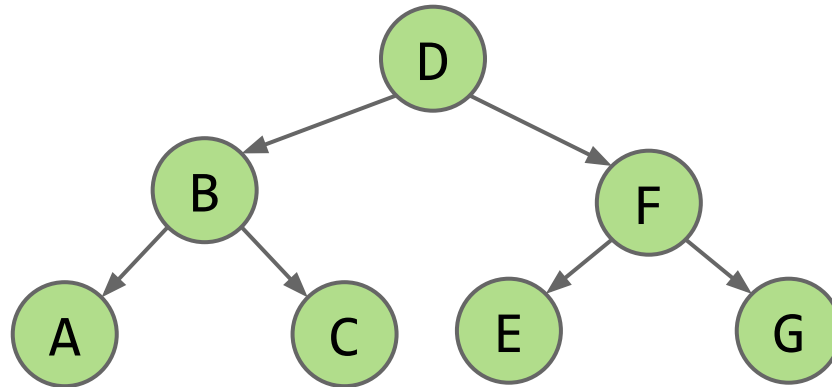
Tree Traversal Orderings

Level Order

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

Depth First Traversals

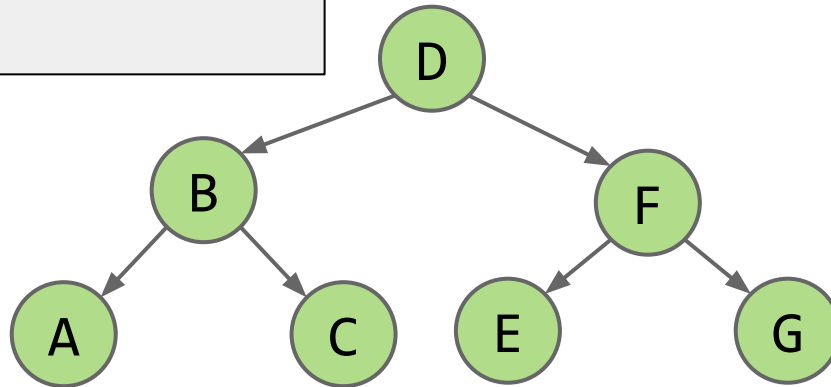
- 3 types: Preorder, Inorder, Postorder
- Basic (rough) idea: Traverse “deep nodes” (e.g. A) before shallow ones (e.g. F).
- Note: Traversing a node is different than “visiting” a node. See next slide.



Depth First Traversals

Preorder: "Visit" a node, then traverse its children: D B A C F E G

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



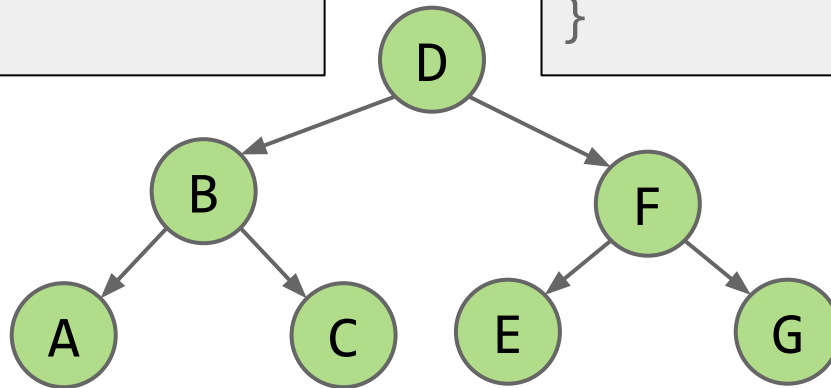
Depth First Traversals

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

Inorder traversal: Traverse left child, visit, then traverse right child: A B C D E F G

```
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/drop>

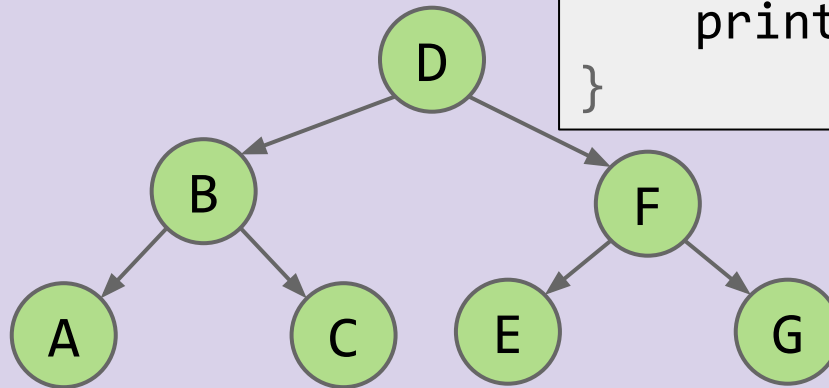
Preorder traversal: "Visit" a node, then traverse its children: DBACFEG

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

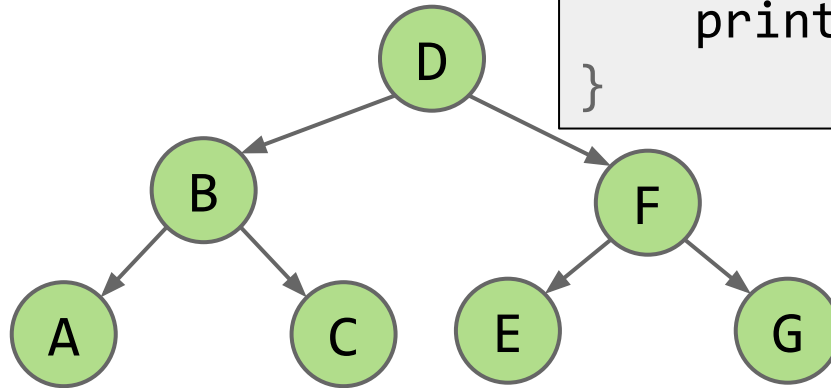
Preorder traversal: "Visit" a node, then traverse its children: DBACFEG

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)  
}
```

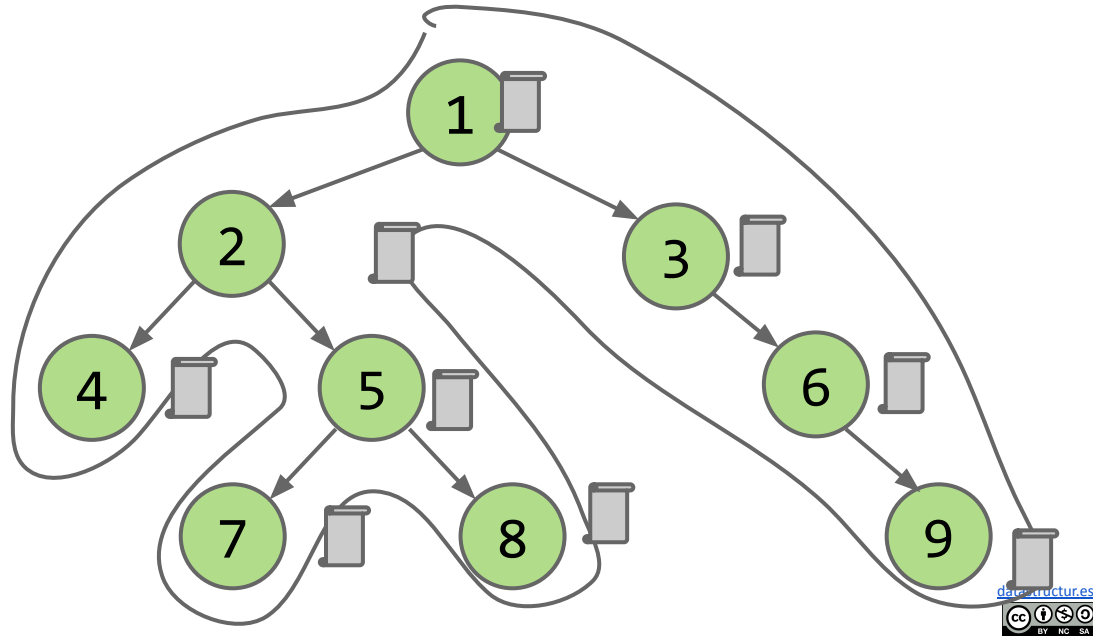


A Useful Visual Trick (for Humans, Not Algorithms)

- 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 of a node.

Example: Post-Order Traversal

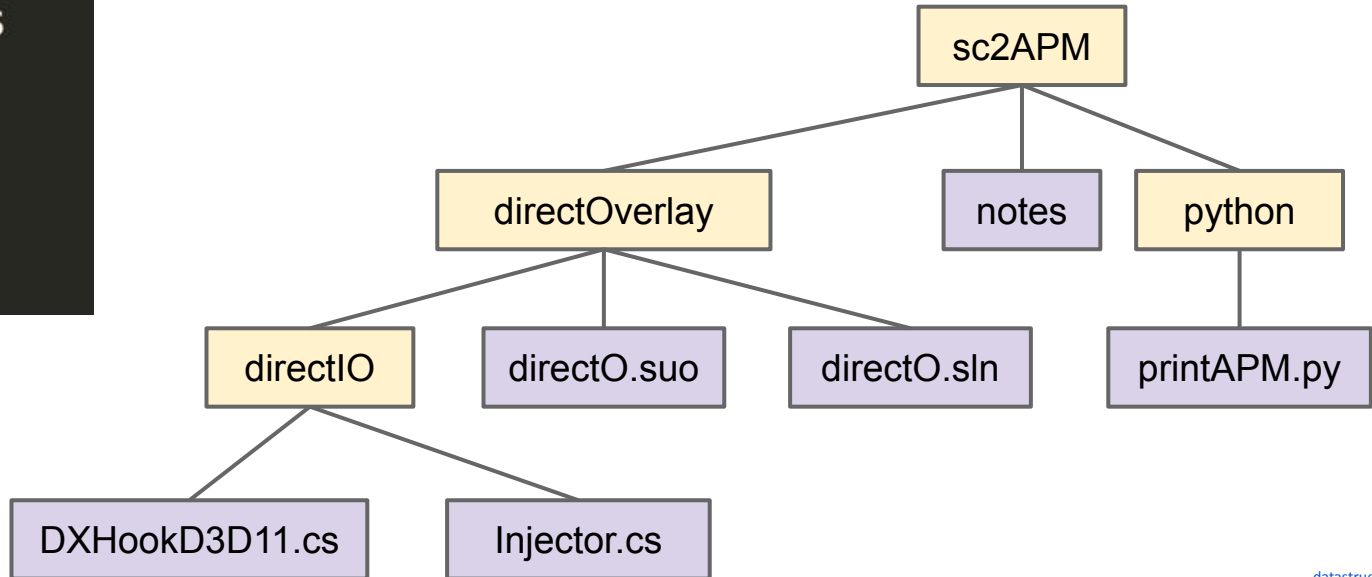
- 4 7 8 5 2 9 6 3 1



What Good Are All These Traversals?

Example: Preorder Traversal for printing directory listing:

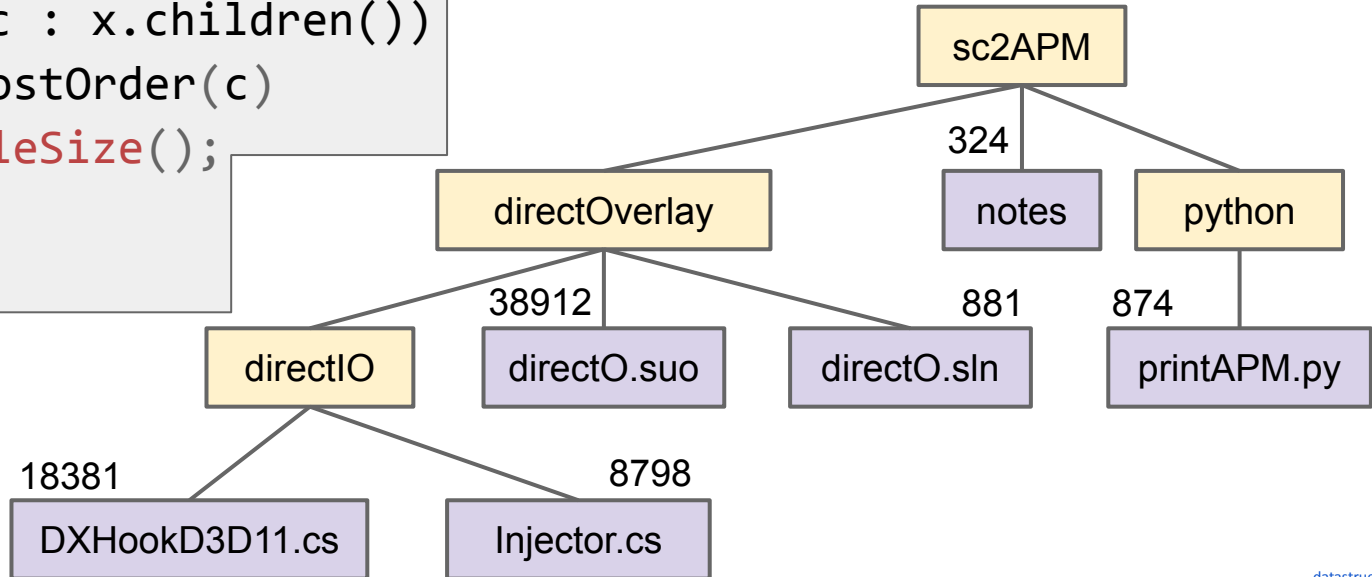
```
sc2APM/  
  directOverlay/  
    directIO/  
      DXHookD3D11.cs  
      Injector.cs  
    directO.suo  
    directO.sln  
  notes  
  python/  
    printAPM.py
```



What Good Are All These Traversals?

Example: Postorder Traversal for gathering file sizes.

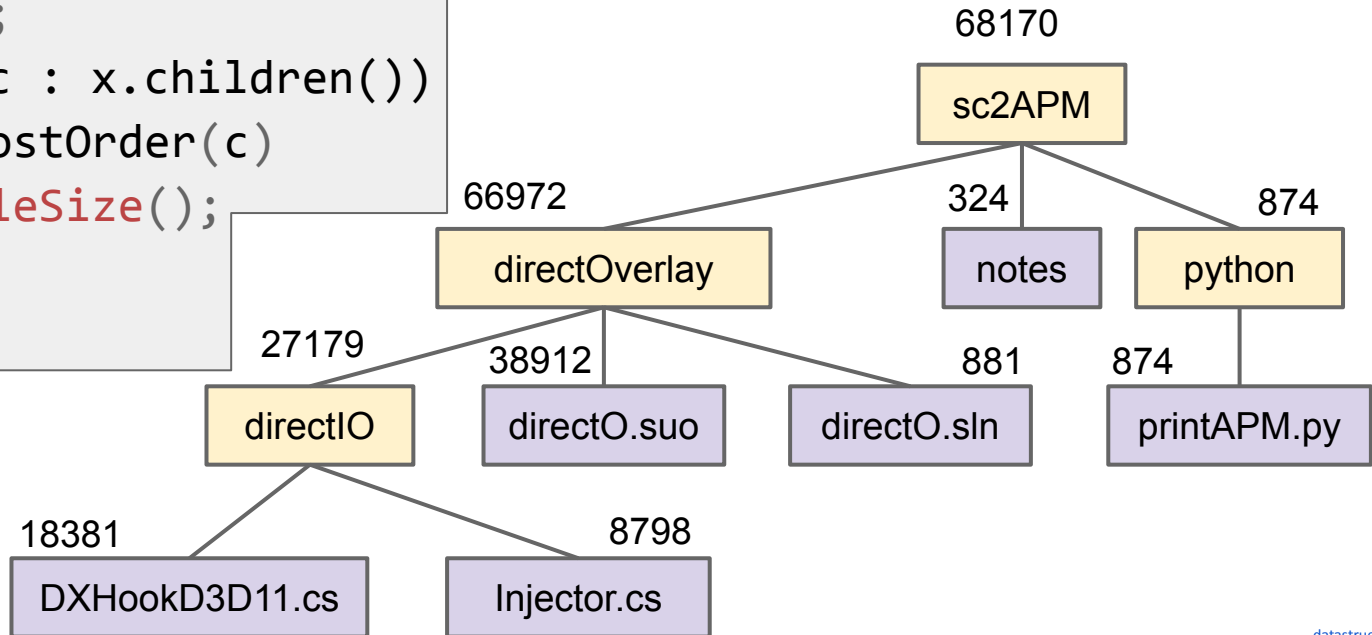
```
postOrder(BSTNode x) {  
    if (x == null) return 0;  
    int total = 0;  
    for (BSTNode c : x.children())  
        total += postOrder(c)  
    total += x.fileSize();  
    return total;  
}
```



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;  
}
```



Graphs

Trees and Hierarchical Relationships

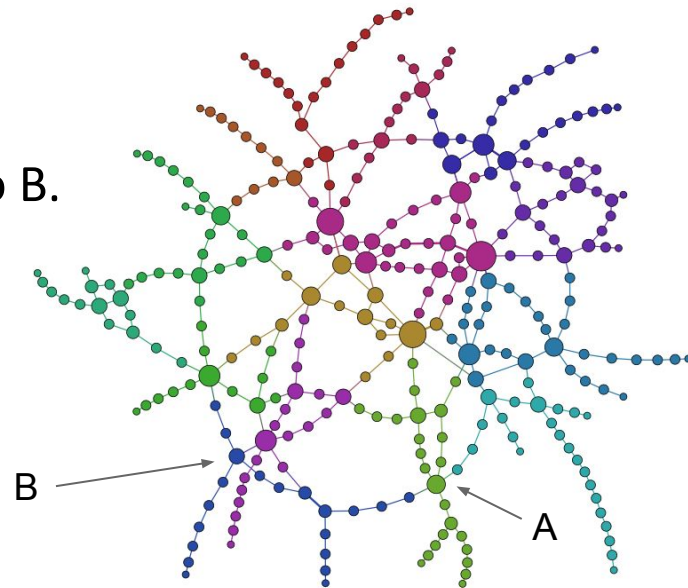
Trees are fantastic for representing strict hierarchical relationships.

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

Introduction to **Network Visualization** with Gephi – Martin Grandjean
Examples

This is not a tree: Contains cycles!

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

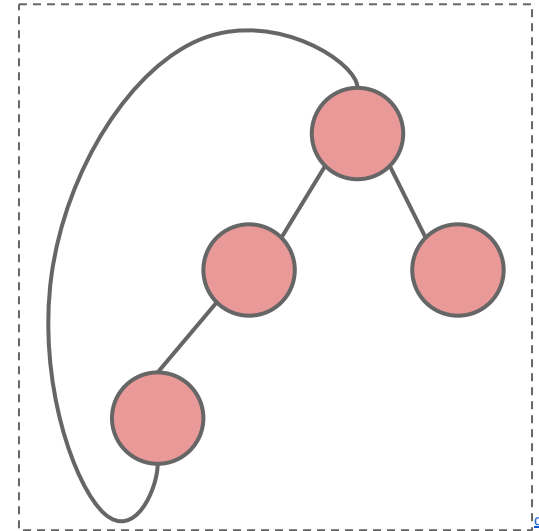
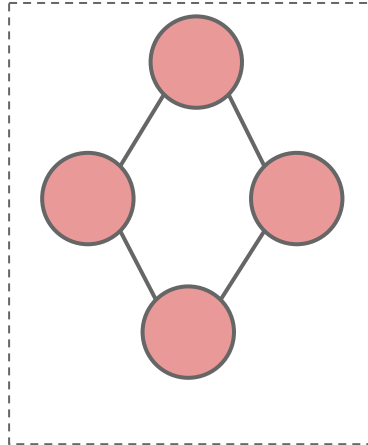
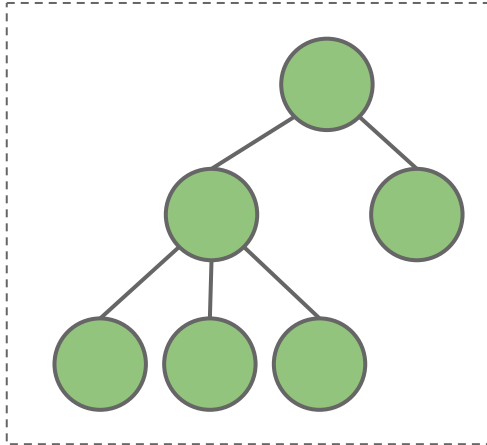
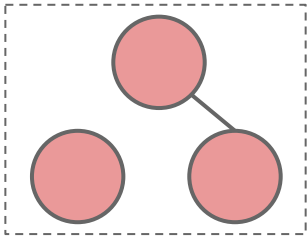


Tree Definition (Revisited)

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.



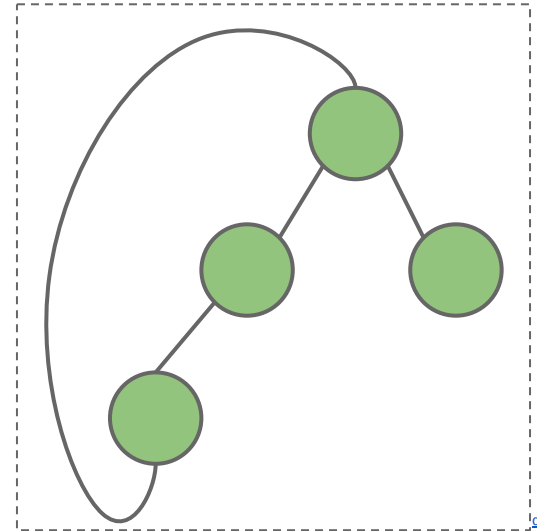
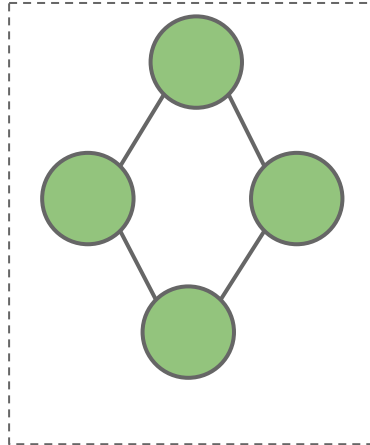
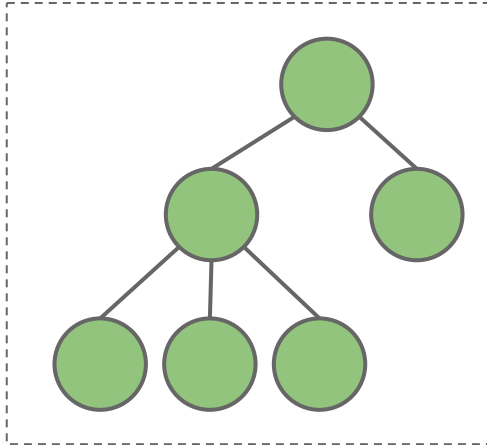
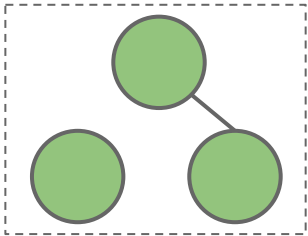
Graph Definition

A graph consists of:

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

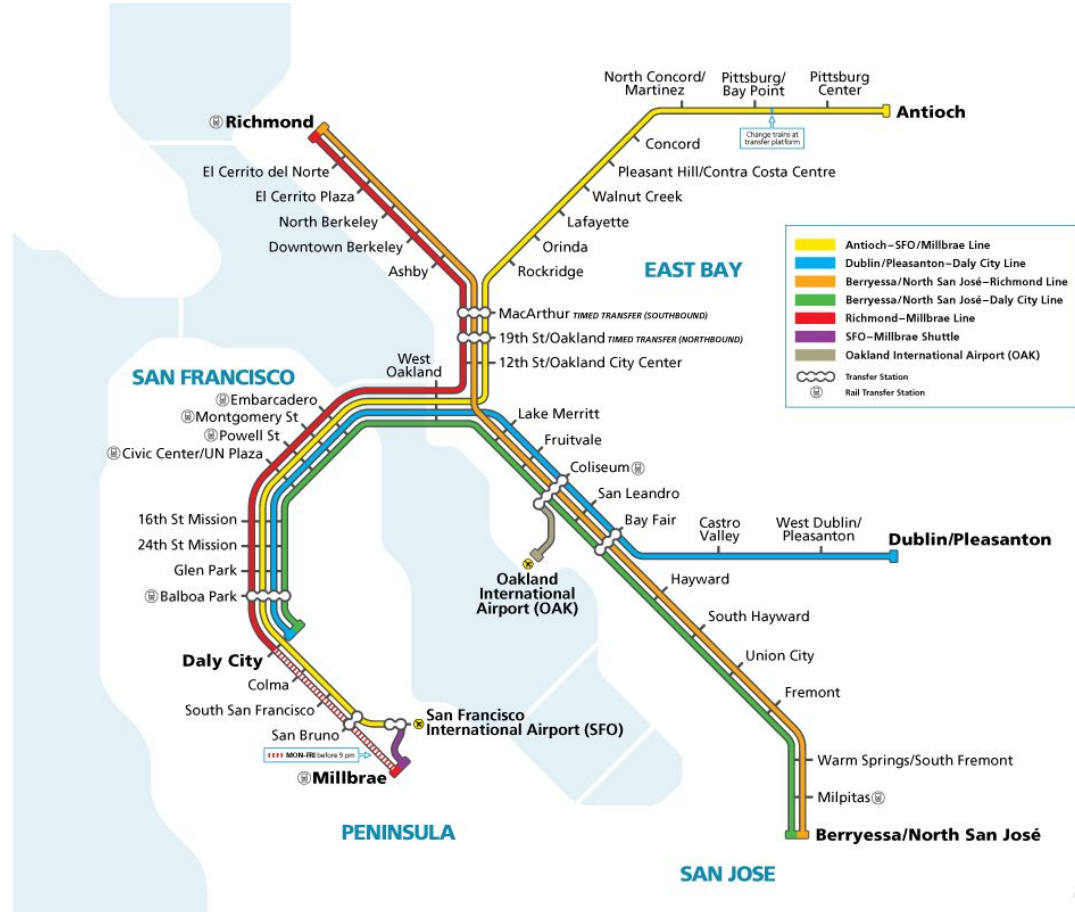
Green structures below are graphs.

- Note, all trees are graphs!



Graph Example: BART

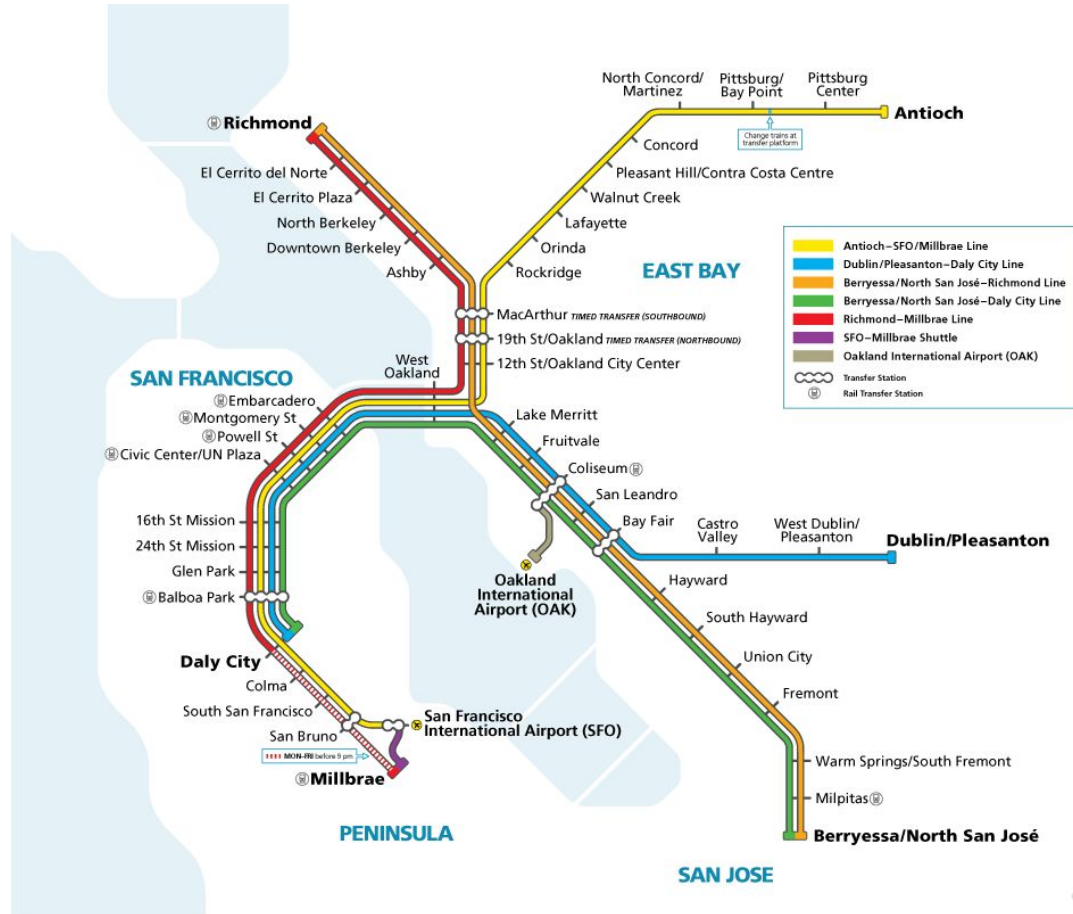
Is the BART graph a tree?



Graph Example: BART

Is the BART graph a tree?

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

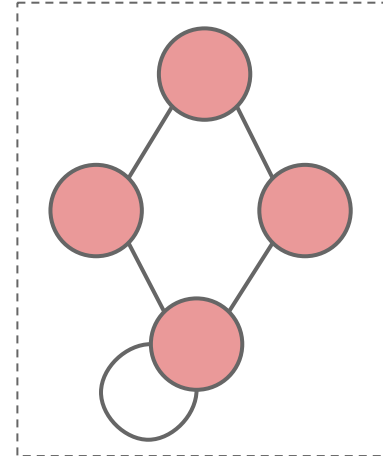
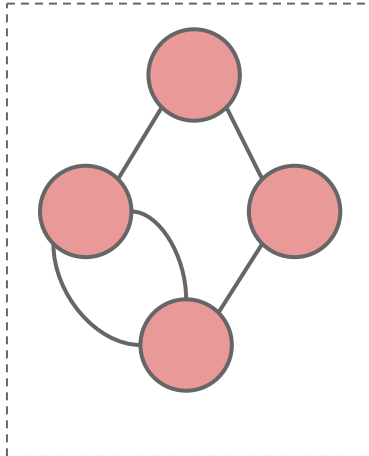
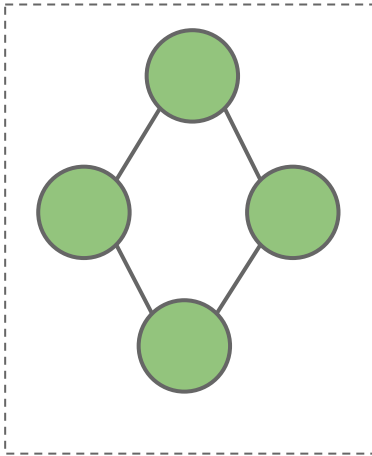


Graph Definition

A simple graph is a graph with:

- No edges that connect a vertex to itself, i.e. no “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.**

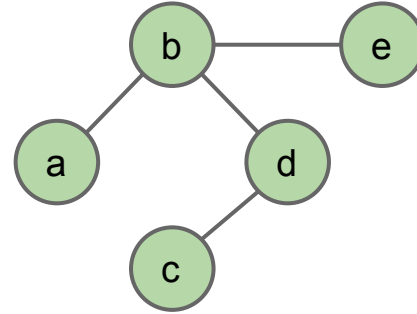
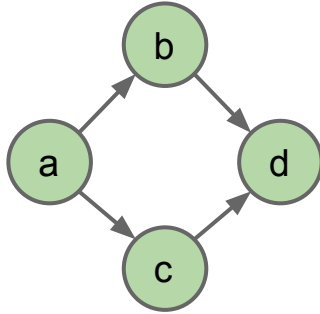
- In other words, when we say “graph”, we mean “simple graph.”

Graph Types

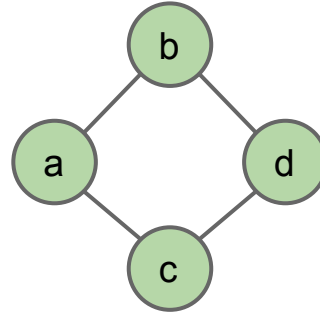
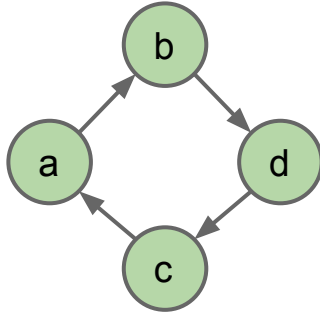
Directed

Undirected

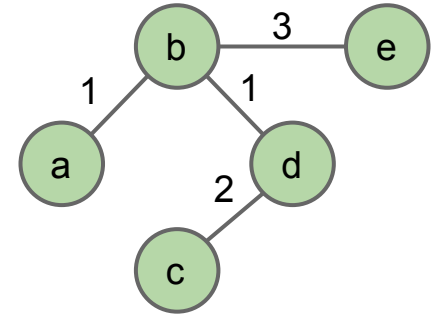
Acyclic:



Cyclic:



With Edge Labels



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.

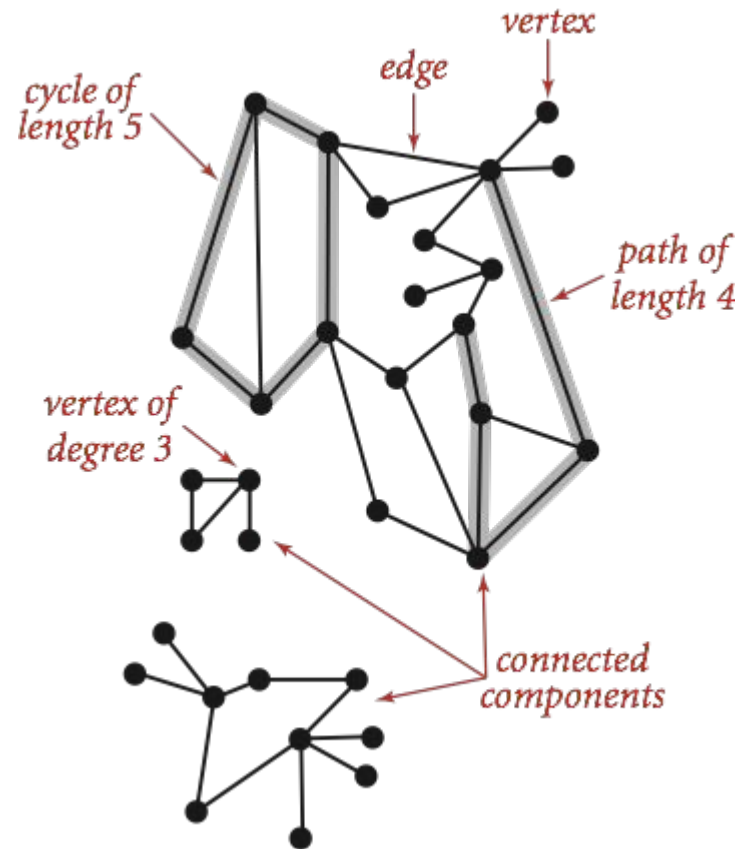


Figure from Algorithms 4th Edition

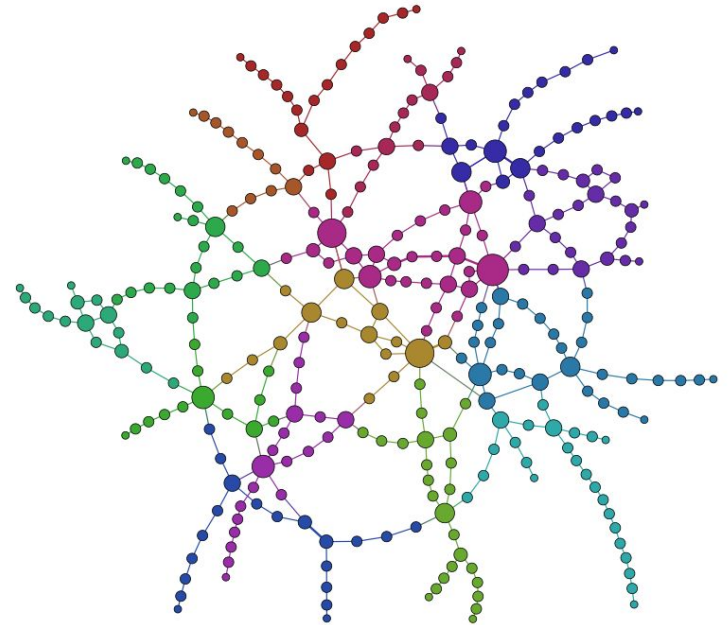
Graph Example: The Paris Metro

This schematic map of the Paris Metro is a graph:

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

Introduction to **Network Visualization** with GEPHI – Martin Grandjean

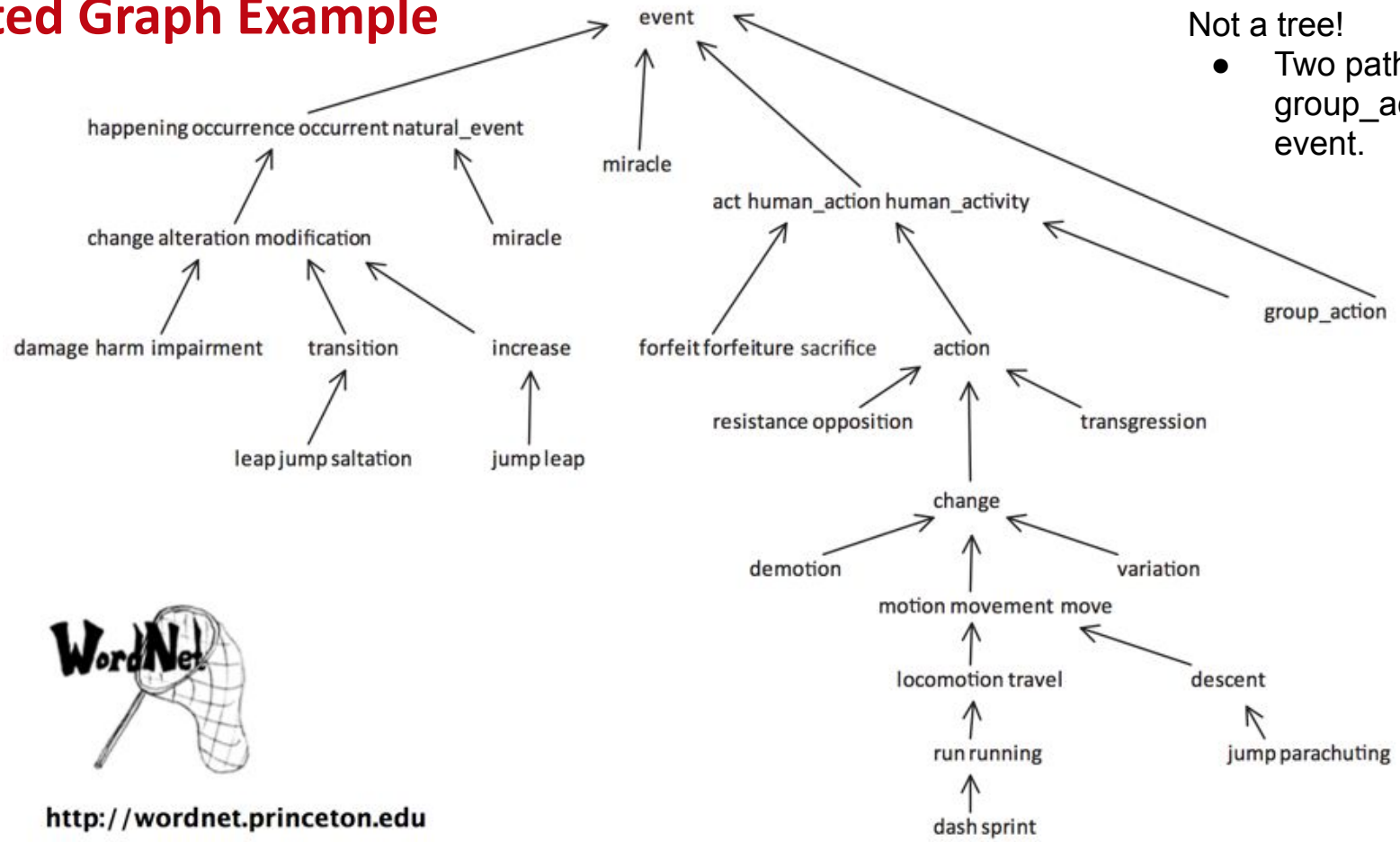
Examples



Directed Graph Example

Not a tree!

- Two paths from group_action to event.



<http://wordnet.princeton.edu>

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

Graph Problems

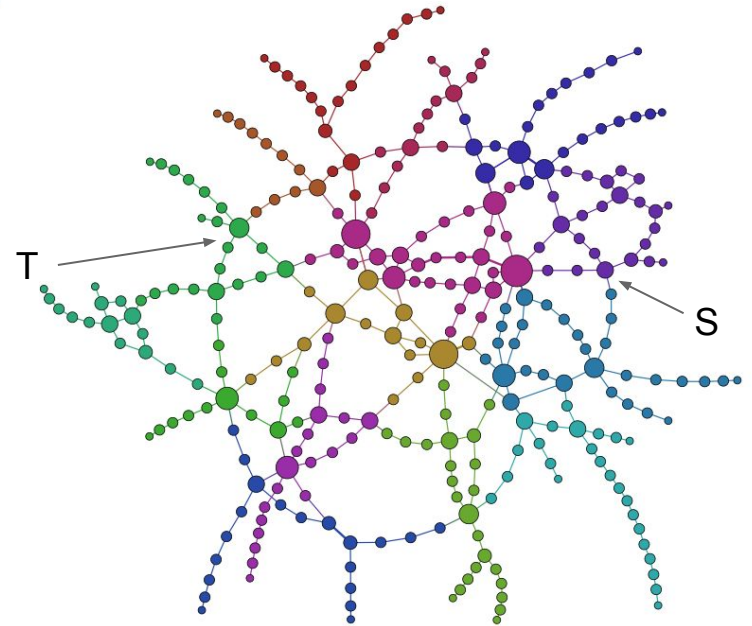
Graph Queries

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?

Introduction to **Network Visualization** with GEPHI – Martin Grandjean

Examples



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 $O(\# \text{ edges})$ 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.

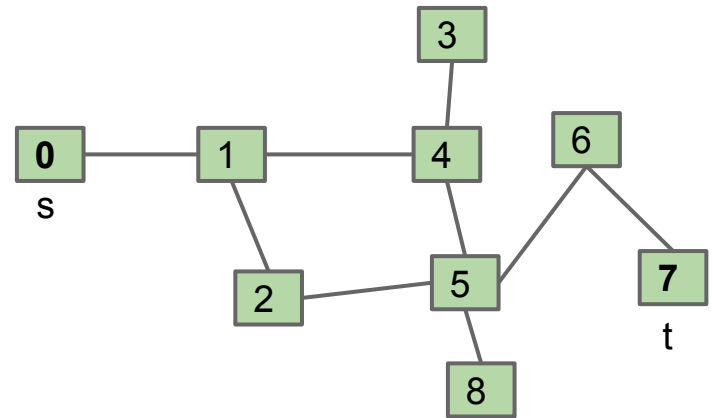
Depth-First Traversal

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.



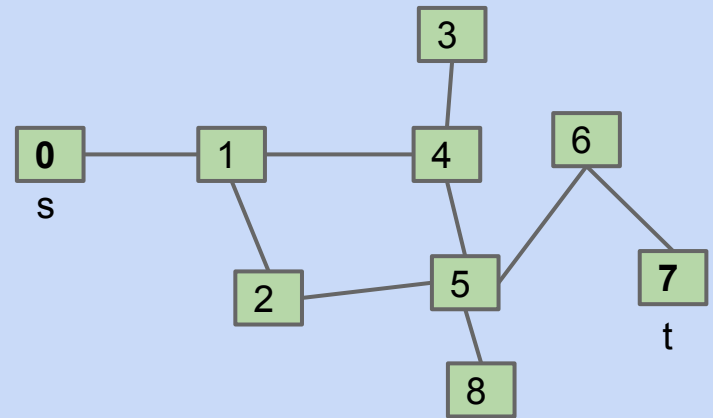
s-t Connectivity

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)`.

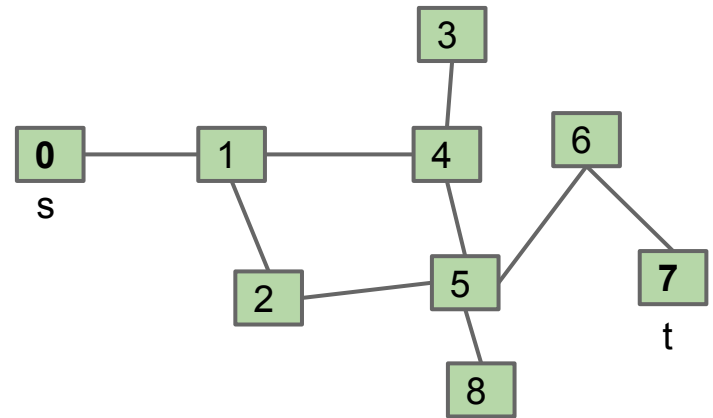


s-t Connectivity

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?

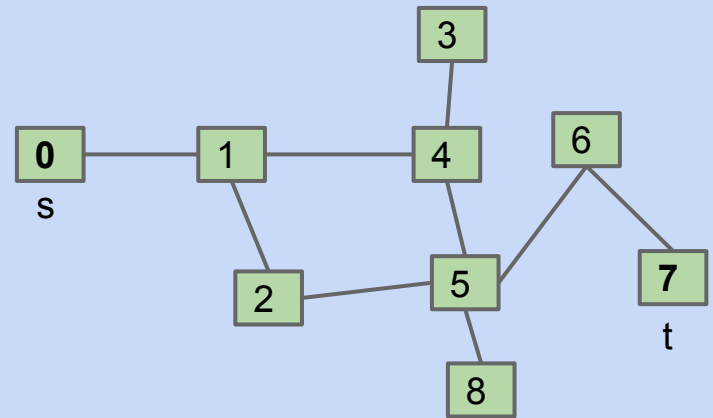


s-t Connectivity

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?



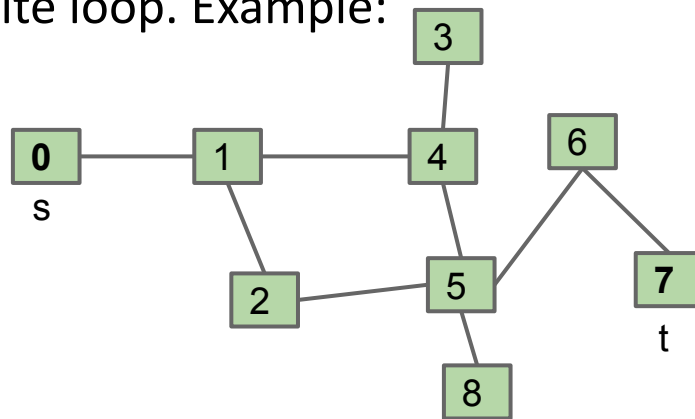
s-t Connectivity

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.



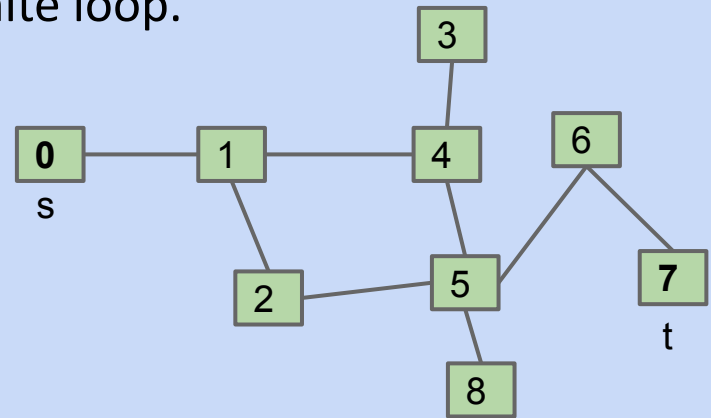
s-t Connectivity

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?



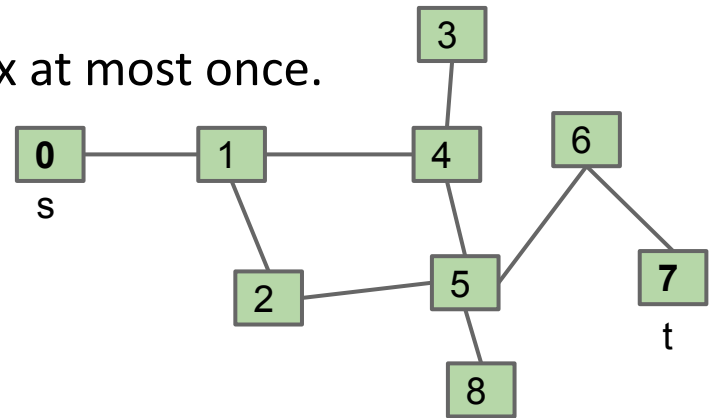
s-t Connectivity

One possible recursive algorithm for $\text{connected}(s, t)$.

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

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

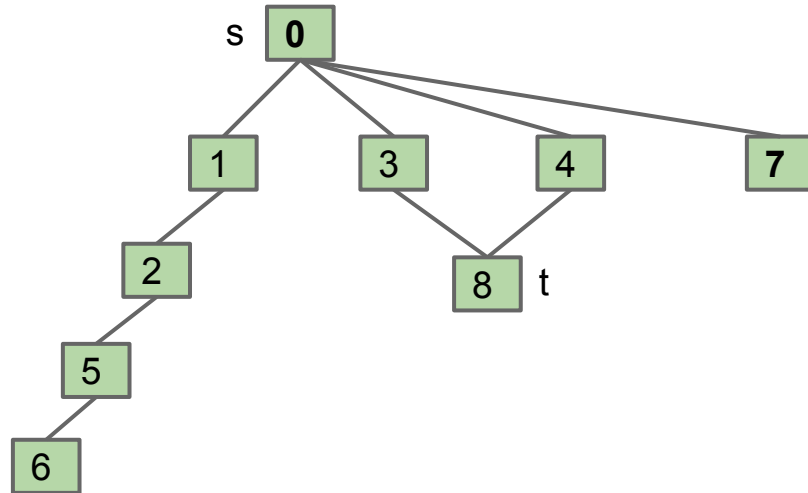
- Marking nodes prevents multiple visits.
- Demo: [Recursive s-t connectivity](#).



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.

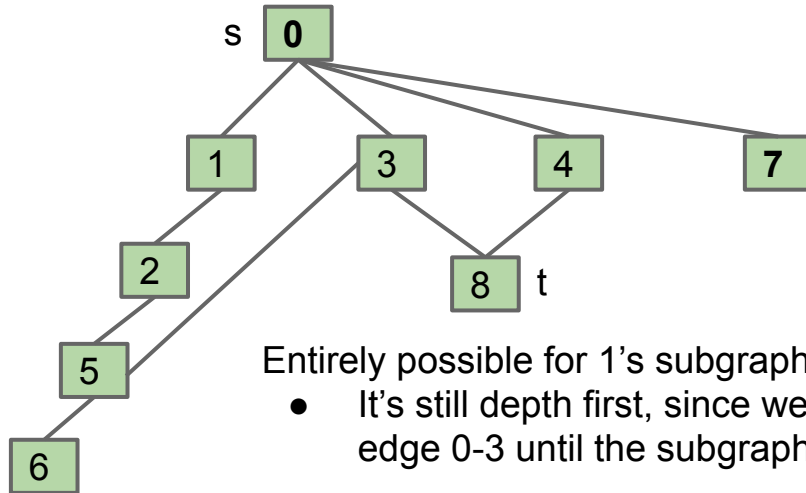
- Example: Explore 1's subgraph completely before using the edge 0-3.
- Called "depth first" because you go as deep as possible.



Depth First Traversal

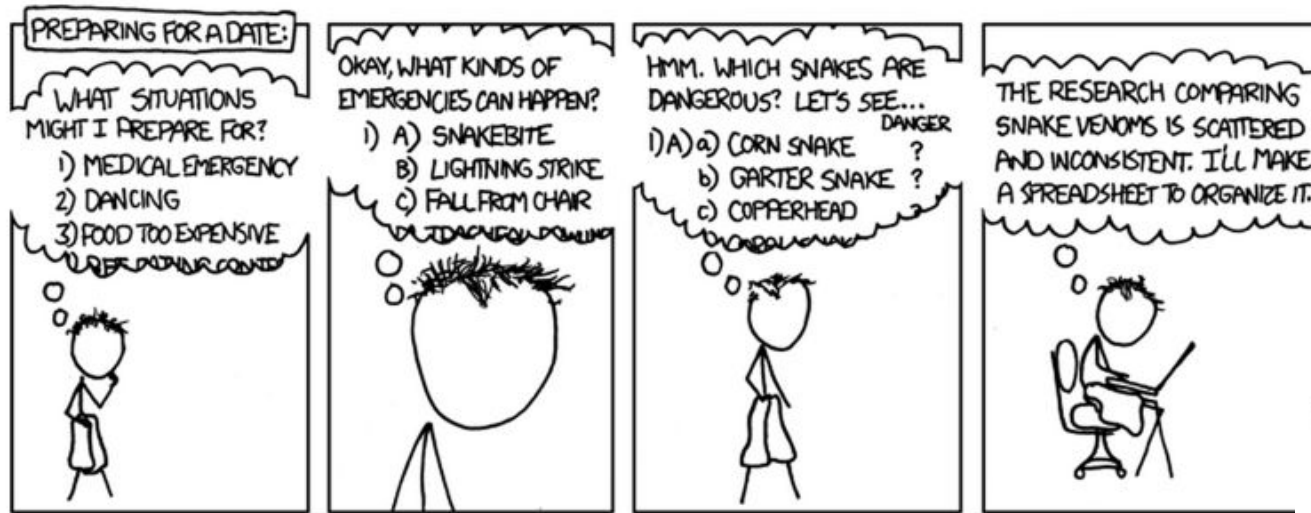
This idea of exploring a neighbor's entire subgraph before moving on to the next neighbor is known as Depth First Traversal.

- 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/>

I REALLY NEED TO STOP USING DEPTH-FIRST SEARCHES.

The Power of Depth First Search

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

Another example:

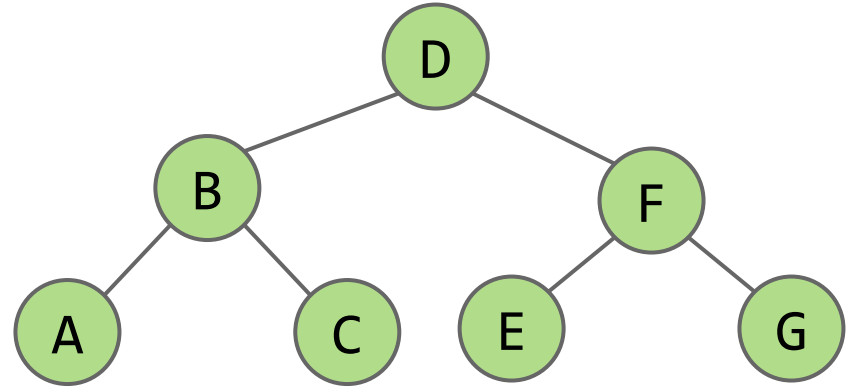
- Let's discuss an algorithm that computes a path to every vertex.
- Let's call this algorithm DepthFirstPaths.
- Demo: [DepthFirstPaths](#).

Tree Vs. Graph Traversals

Tree Traversals

There are many tree traversals:

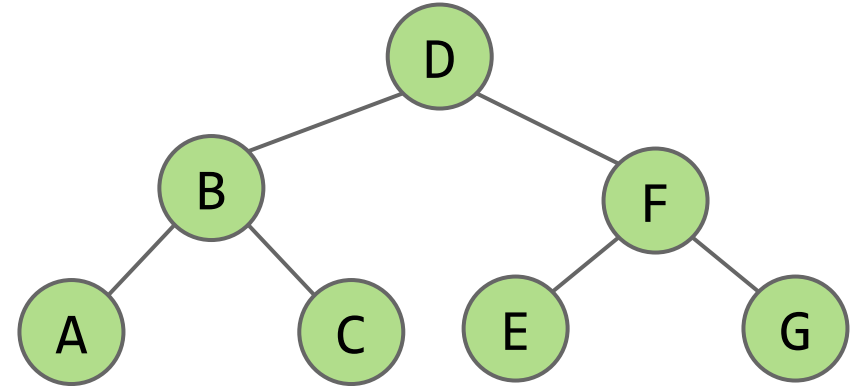
- Preorder: DBACFEG
- Inorder: ABCDEFG
- Postorder: ACBEGFD
- Level order: DBFACEG



Graph Traversals

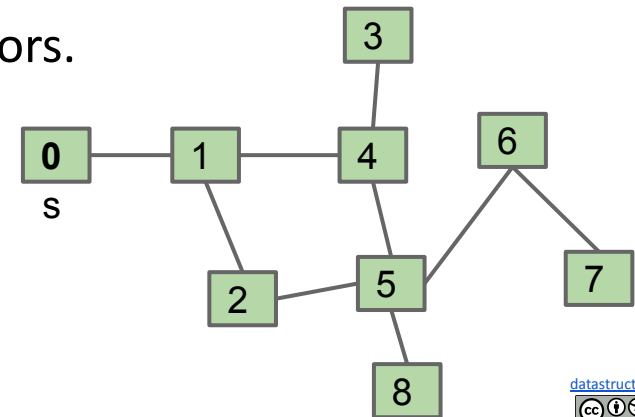
There are many tree traversals:

- 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.



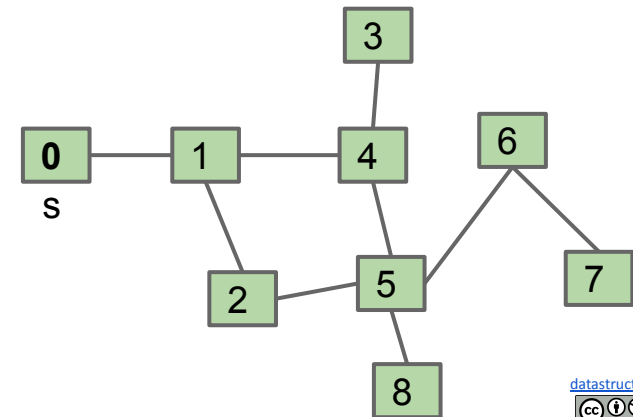
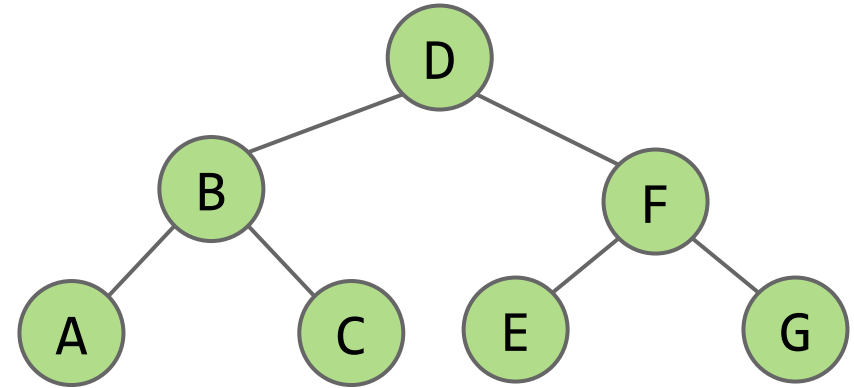
Graph Traversals

There are many tree traversals:

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

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

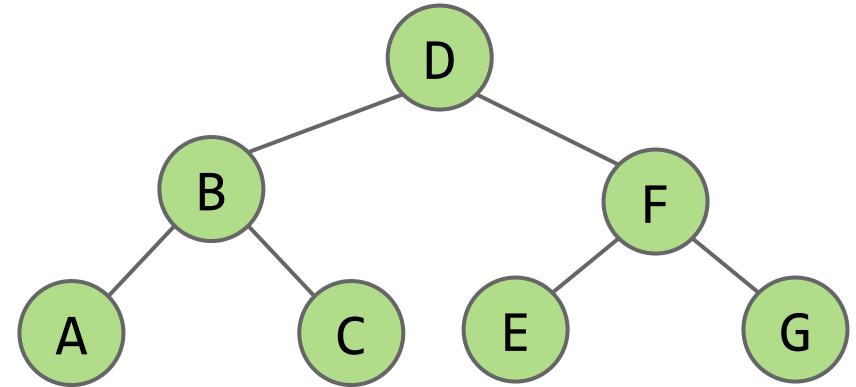
- **DFS Postorder: Action** is **after DFS** calls to neighbors.
- Example: `dfs(s)`:
 - `mark(s)`
 - For each unmarked neighbor `n` of `s`, `dfs(n)`
 - `print(s)`
- Results for `dfs(0)` would be: 347685210
- Equivalent to the order of `dfs` returns.



Graph Traversals

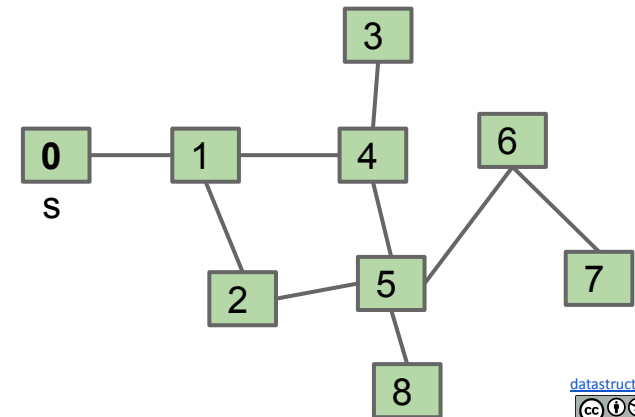
Just as there are many tree traversals:

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



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

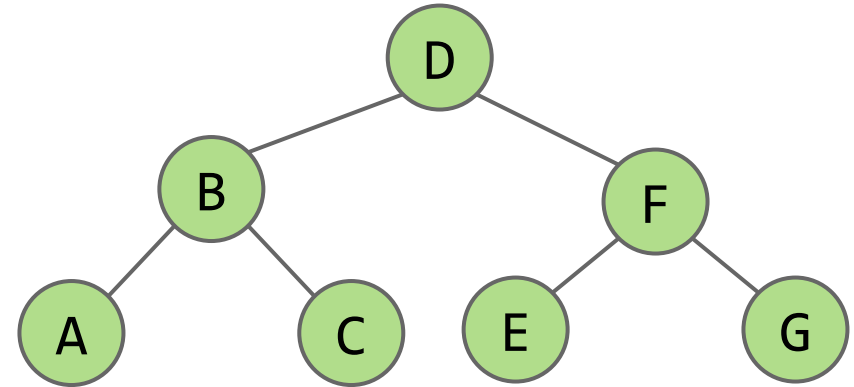
- DFS Preorder: 012543678 (dfs calls).
- DFS Postorder: 347685210 (dfs returns).



Graph Traversals

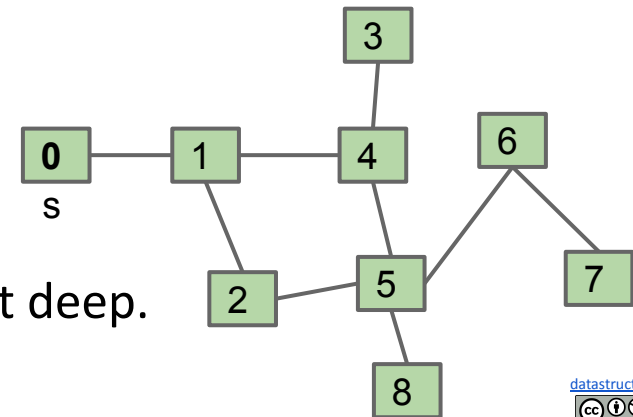
Just as there are many tree traversals:

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

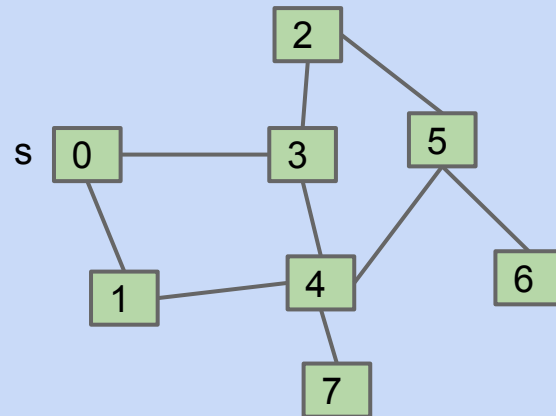


So too are there many graph traversals, given some source:

- 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



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

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.