Tree Traversal Orderings

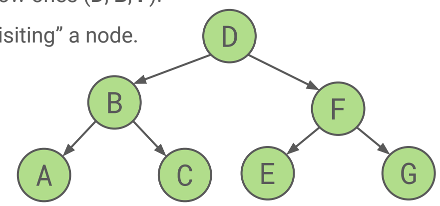
Level-Order Traversal. Visit top-to-bottom, left-to-right (like reading in English): **DBFACEG**

Depth-First Traversals.

Traverse deep nodes (**A**, **C**, **E**, **G**) before shallow ones (**D**, **B**, **F**).

Note: “Traversing” a node is different than “visiting” a node.

3 types: **Preorder**, **Inorder**, **Postorder**.

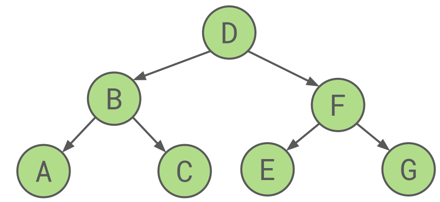


Depth-First Traversals

Preorder Traversal.
“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)
}
```

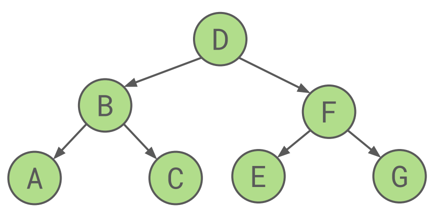


Depth-First Traversals

Inorder Traversal.
Traverse left child, “visit”, then traverse right child.

A B C D E F G

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

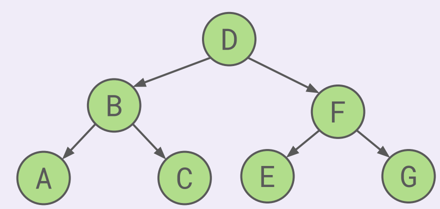


Q

Depth-First Traversals

Postorder Traversal.
Traverse left, traverse right, then “visit.”

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



Depth-First Traversals: Visual Trick (for humans)

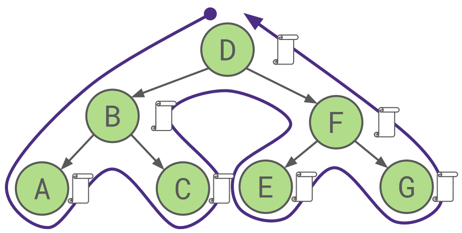
First, trace a path around the graph from the top going counter-clockwise.

Preorder. “Visit” when passing the left.

Inorder. “Visit” when passing the bottom.

Postorder. “Visit” when passing the right.

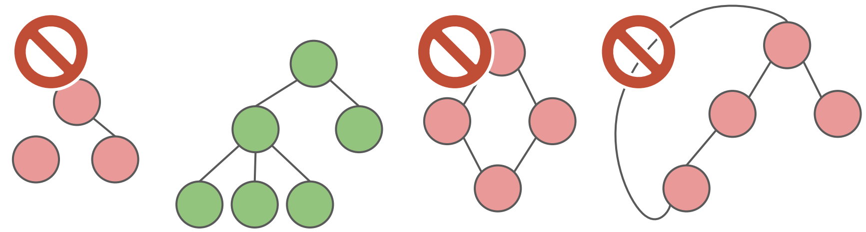
A C B E G F D



Alternate Tree Definition

Tree. Consists of a set of nodes and a set of edges that connect those nodes.

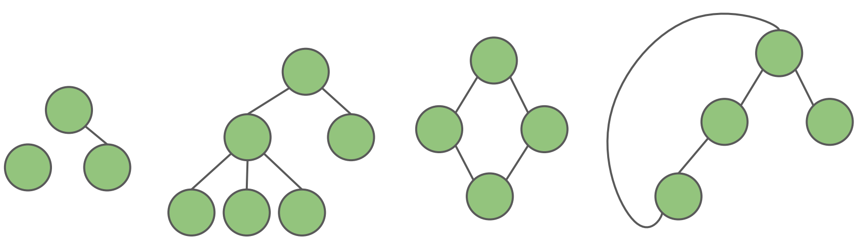
Invariant. There is exactly one path between any two nodes.



Graph Definition

Graph. Consists of a set of nodes and a set of zero or more edges.

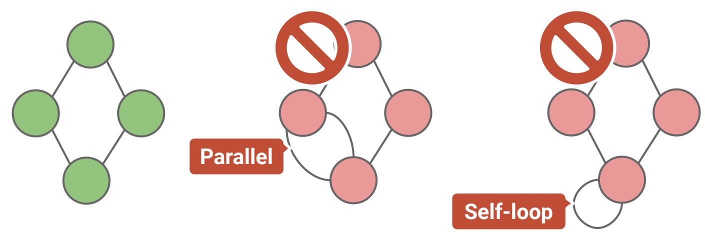
Each edge connects any two nodes. Not all nodes need to be connected.

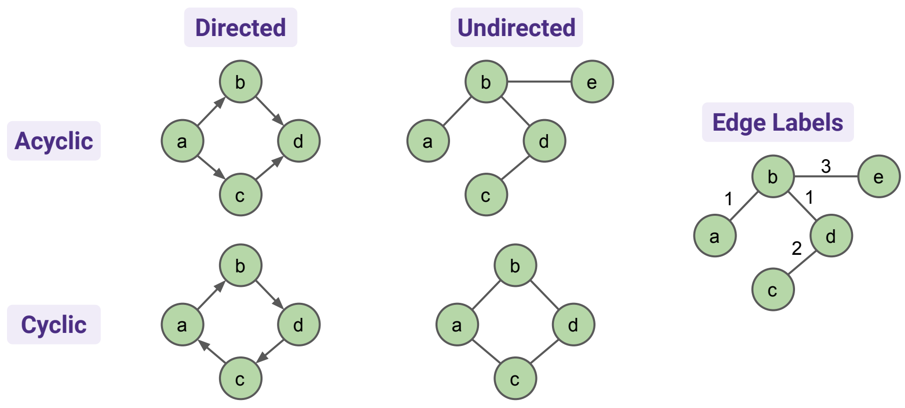


Simple Graph Definition

Simple Graph. A graph with no **self-loops** and no **parallel edges**.

Unless otherwise stated, **all graphs in this course are simple graphs**.





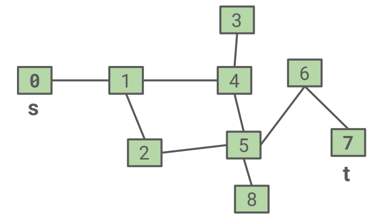
Types of Graphs

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**, does there exist a path between **s** and **t**?

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

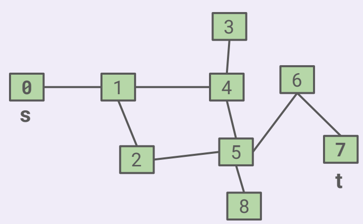


Q Applying Tree Traversal

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

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

What is problematic about this algorithm?



Depth-First Search

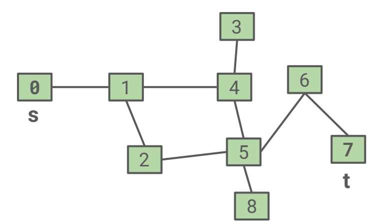
Demo

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

1. **Mark s** as visited.
2. Does **s** == **t**? If so, return true.
3. Otherwise, if connected(**v**, **t**) for any **unmarked** neighbor **v** of **s**, return true.
4. Return false.

Each vertex visited at most once.

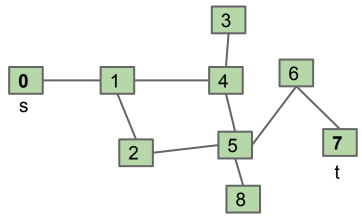
Depth-First Search.



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.



DepthFirstPaths Demo

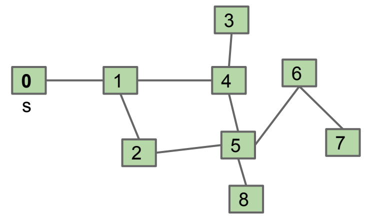
Goal: Find a path from s to every other reachable vertex, visiting each vertex at most once. dfs(v) is as follows:

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

#	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 calls: 0



Order of dfs returns: