Lecture 7

COMP 3760

Data Structures and Graphs

Text chapter 1.4, 3.5, 5.3

Fundamental Data Structures

(Chapter 1.4)

Data Structures

- A *data structure* is a particular way of storing and organizing data
- Data structures and algorithms are often deeply interconnected
  - The way you organize data affects the performance of your algorithm
- We've *mostly* been using arrays … so far

Fundamental Data Structures

- Linear Data Structures
  - Array
  - Linked list
  - Stack
  - Queue
- Set
- Dictionary (Map)
- Tree
- Graph

Arrays

- A sequence of *n* items of the same type, accessed by an index

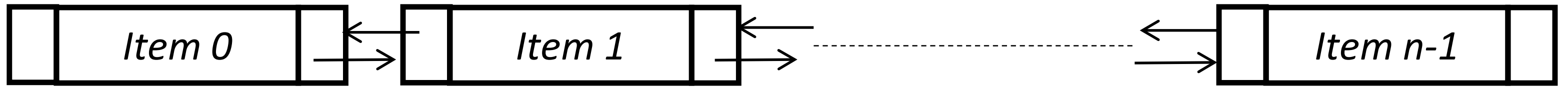
Item[0]	*Item[1]*	…	*Item[n-1]*

- The good:
  - Each item accessed in same constant time
- The bad:
  - Size is fixed
  - Insertion / deletion in an array is time consuming – all the elements following the inserted element must be shifted appropriately

Linked Lists

- (singly) A sequence of zero or more elements called *nodes,* consisting of data and a pointer

- (doubly) Pointers in each direction

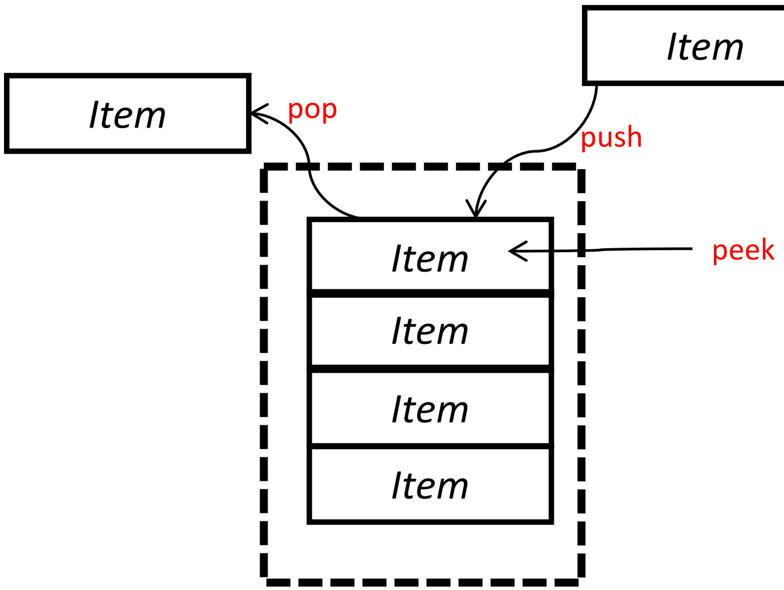


Linked Lists

- Linked lists provide two key advantages over arrays
  - Dynamic size
  - Ease of insertion/deletion

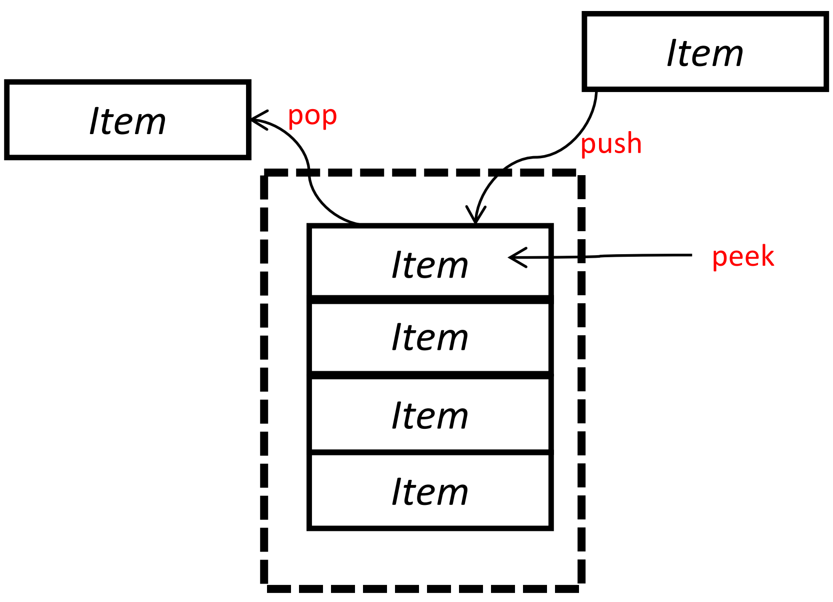
- Linked lists have some drawbacks:
  - Random access is not allowed

Do you know what this is?



Stack

- Like a stack of plates
- Last-in-first-out (LIFO)



Operations on a stack

- Insert operation is called Push
- Delete operation is called Pop
- Examining the top item is Peek

- Example application:
  - Analysis of languages (e.g. properly nested brackets)
  - Properly nested: (())
  - Wrongly nested: ())(()

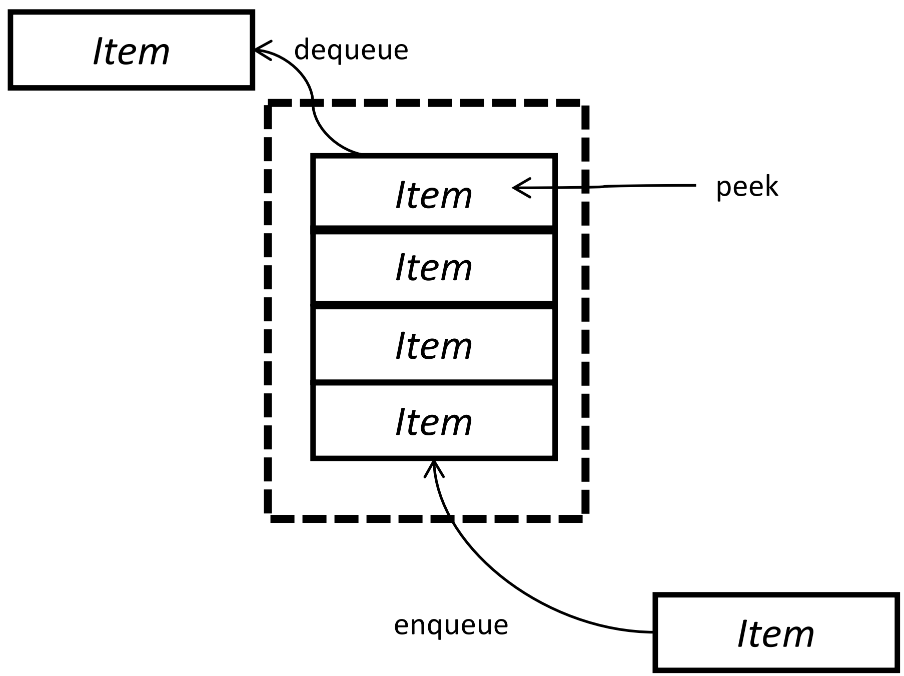
Stack

```
Algorithm CheckBalancedParenthesis(expr)
n = length(expr)
Create a stack s
for i = 0 to n-1 do
   if (expr[i] is '(' ) do
      s.Push(expr[i])
   else if (expr[i] is ')' )
      if (s is empty) or
            (s.Peek() does not pair with expr[i])
         return False
      else
         s.Pop()
if (s is empty)
   return True
else
   return False
```

Abstract Data Type

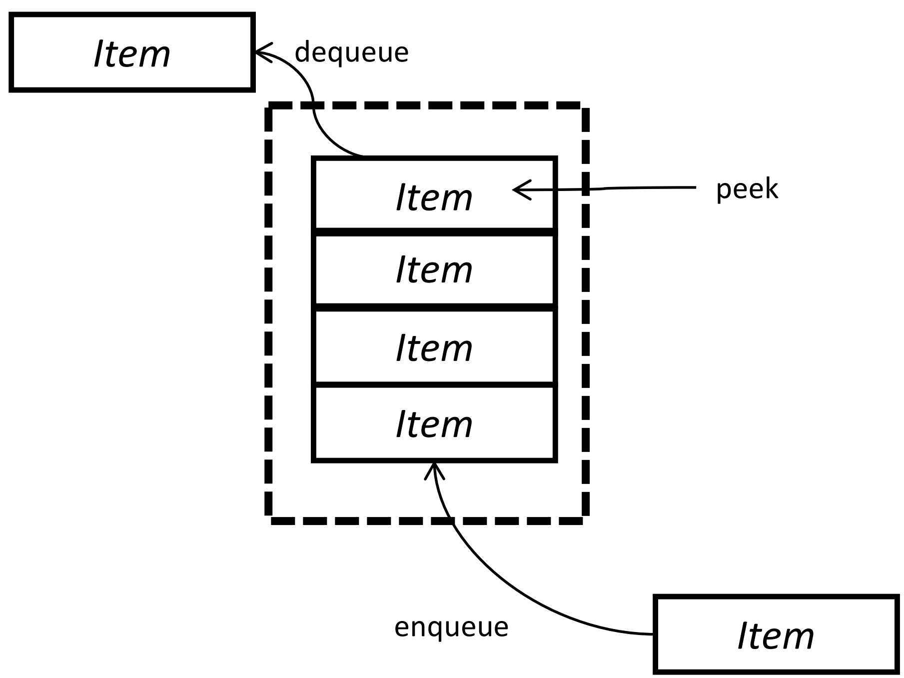
- Often a data structure is closely associated with a set of available operations
- Data structure + operations = *abstract data type*
  - From an OOP perspective, think about members (methods) of a class
- Example 1: priority queue
  - Underlying implementation was a heap
  - Operations were Insert and deleteMax
- Example 2: stack
  - Operations are push, pop, peek

How about this one?



Queues

- Like a line-up
- First-in-first-out (FIFO)



Operations on a queue

- Adding to the queue is Enqueue
- Removing from the queue is Dequeue
- The top/front element is the Head (sometimes there is a “Peek” method)

Set

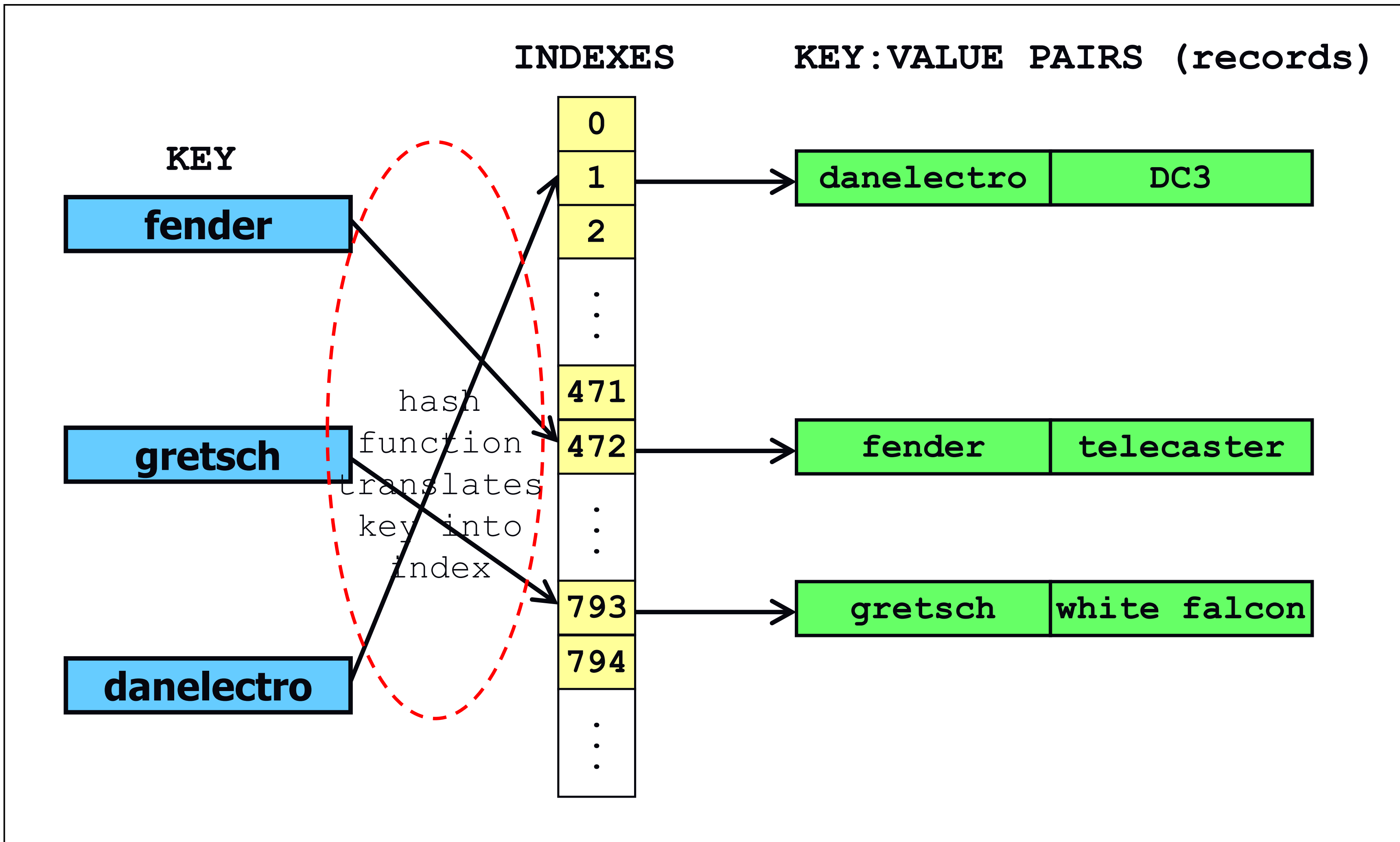
- Like a set in math, e.g. {1, 2, 3, 4}
  - Sets cannot contain duplicate items
- Operations on a Set:
  - Add an item to it
  - Remove an item from it
  - Check if an item is in it
  - Iterate over it (loop based on all items, one-by-one)

Set in Java

- Different ways to implement a set
  - HashSet
    - Faster implementation, but it is unordered
  - TreeSet
    - Slower, but the items are available in order

Map (as a hash table)

- A ***Map*** is a lookup table that takes a ***key*** and returns a ***value***
  - the most common implementation is as a hashtable (hashmap)

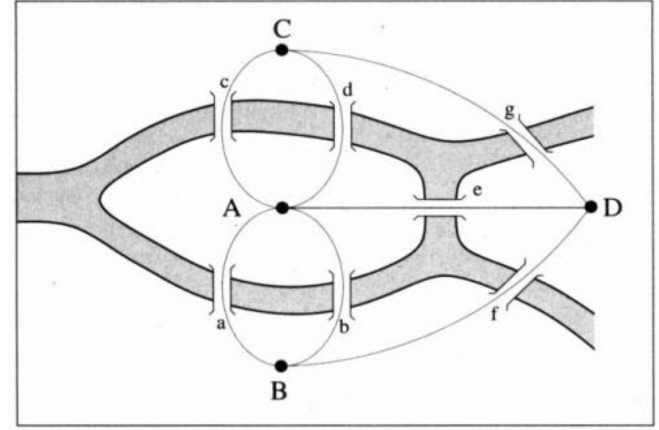


Graphs

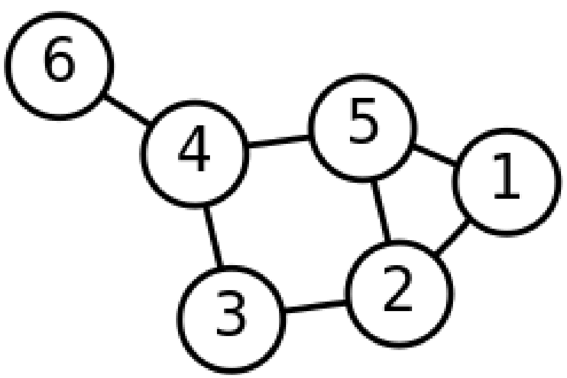
(Still in Chapter 1.4)

Graphs

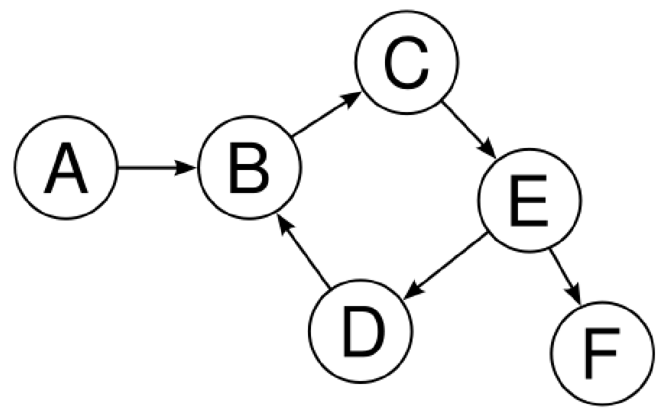
- G = (V, E)
  - V is a set of *vertices*
  - E is a set of *edges*



Motivation: Real world connections



Undirected



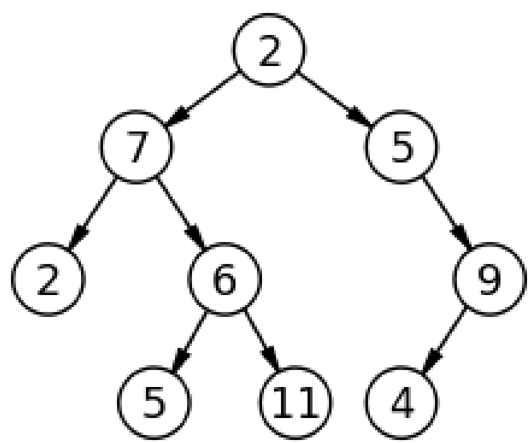
Directed

Some special graphs

- *Connected graph*
  - A graph where there is a path available between any two vertices
- *Cyclic graph*
  - A graph containing at least one cycle
- *Acyclic graph*
  - A graph containing no cycles
- *Tree*
  - Any connected + acyclic graph
- *Complete graph*
  - Every pair of vertices is connected by an edge
- *Weighted graph*
  - Every edge has an associated value

Trees

- Connected, acyclic graphs
  - Usually we think of trees as having a root
- Representing data in a tree can speed up your algorithms in many natural problems

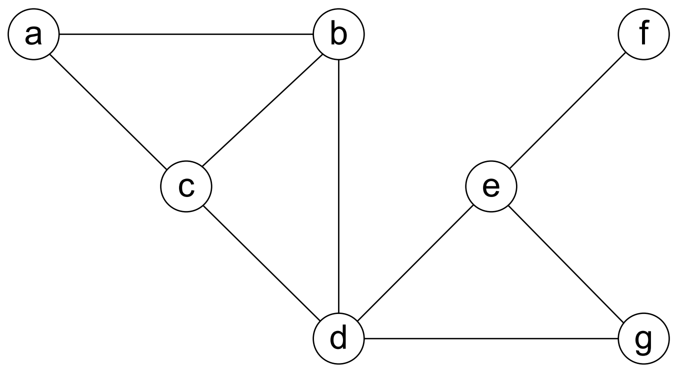


Representation of graphs

- Two common ways to represent graphs:
  - Adjacency matrix
    - |V| x |V| matrix
    - Cell i, j represents an edge from vertex i to j
  - Adjacency lists
    - |V| linked lists – one for each vertex, showing all the neighbours of that vertex

Representation: Adjacency Matrix

- For this graph:



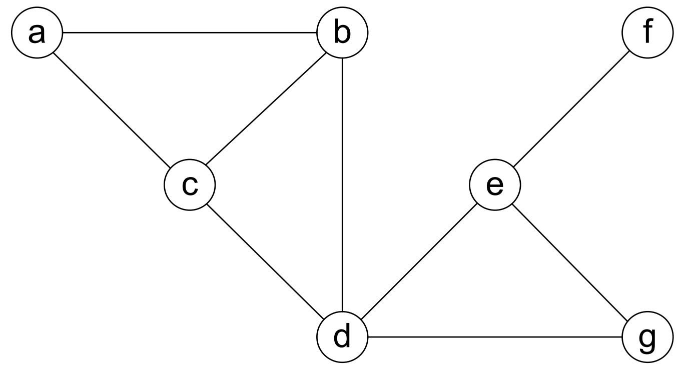
- Adjacency matrix is the following:

O(|v|^2)

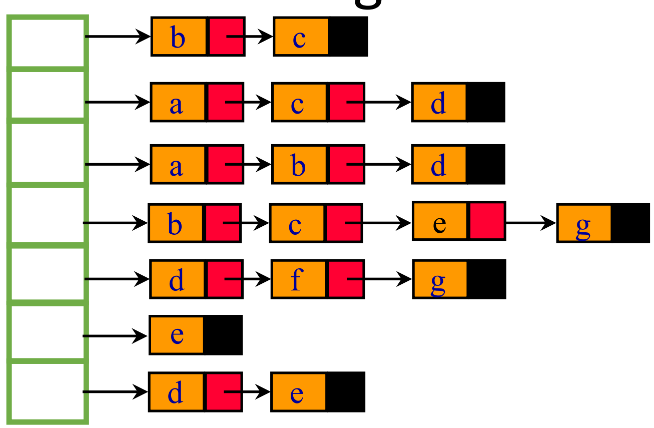
	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	1	0	0	0
c	1	1	0	1	0	0	0
d	0	1	1	0	1	0	1
e	0	0	0	1	0	1	1
f	0	0	0	0	1	0	0
g	0	0	0	1	1	0	0

Representation: Adjacency List

- For the same graph:



- Adjacency list is the following:
  - *For vertex a*:
  - *For vertex b*:
  - *For vertex c*:
  - *For vertex d*:
  - *For vertex e*:
  - *For vertex f*:
  - *For vertex g*:



O(|E|)

Representing Graphs

1. Adjacency matrix
   - Or Weight Matrix for weighted graphs

2. Adjacency lists
   - A list of vertices connected to each vertex

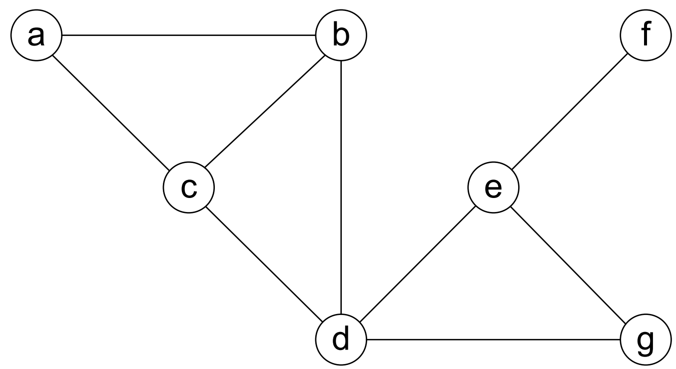
- Which one to use?
  - Depends on the nature of the graph (sparse or not)
  - Depends on the algorithm

Graph Algorithms

(Chapter 3.5)

Graph Traversal

- Many real-world problems require processing of each vertex (or edge) in a graph



  - Routing a message on a network
  - Web crawling
  - Social networking
  - Garbage collection
  - Solving puzzles

Graph Traversal Algorithms

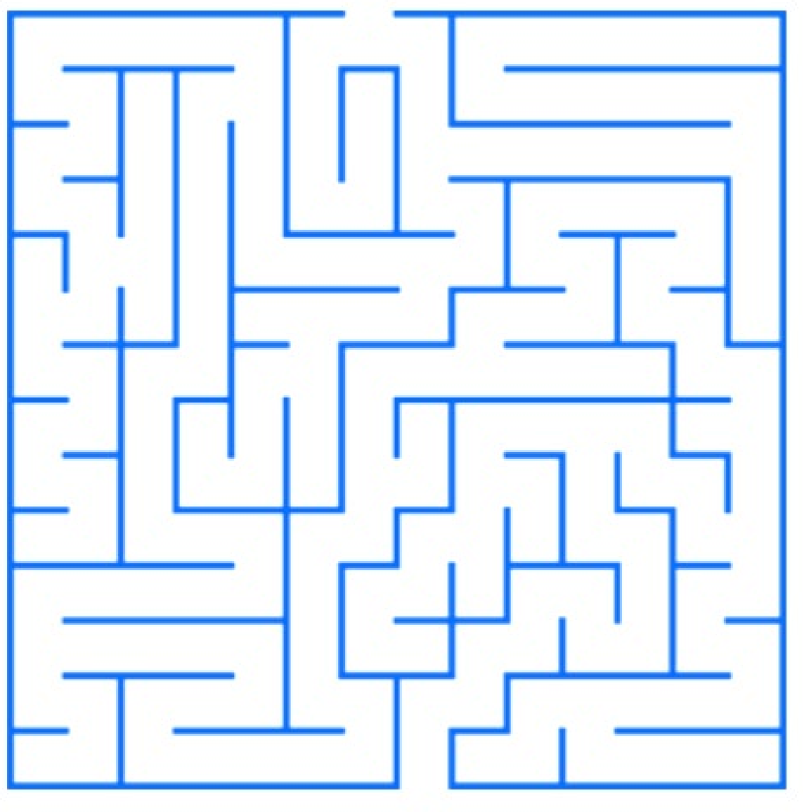
- Graph traversal algorithms give a method for *systematically processing* all vertices

Basic idea: "visit" all the vertices, one at a time, *marking* them as we visit them

- Two approaches:
  - Depth-First Search (DFS)
  - Breadth-First Search (BFS)

Depth-first search (DFS)

- Think about how you might try to find your way through a maze



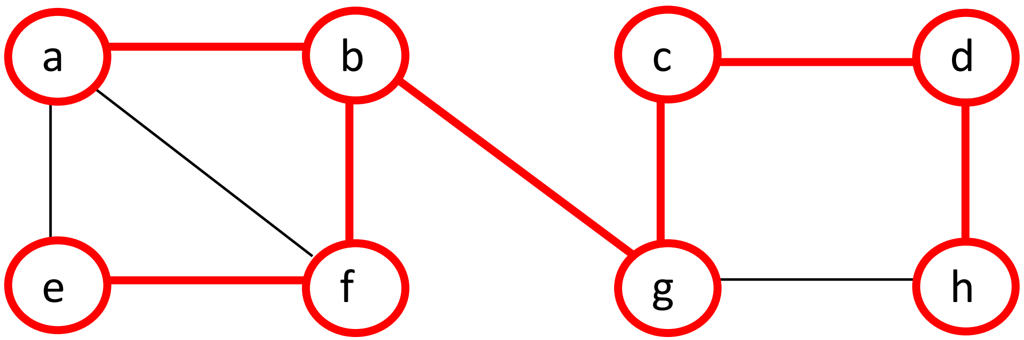
Depth-first search (DFS)

- Visits all vertices by always <u>*moving away*</u> from the last vertex visited (if possible)
  - Backtracks at “dead-ends” (no adjacent, unvisited vertices)

- Implementation often uses a stack of vertices being processed

- Follows a tree-like route throughout the graph

DFS example



Backtrack/finish order: e f h d c g b a

DFS order: a b f e g c d h

Some notes on DFS

- To track the progress of the algorithm we use a stack
  - When we make a recursive call, e.g. dfs(v), we push v onto the stack
  - When v is a dead-end (i.e. no more neighbors to visit) it is popped off the stack
- Our convention: break ties for “next neighbor” by using some natural order
- Typical results from running DFS can be:
  - List of vertices in order visited
  - List of vertices in order of “dead-ends” (when popped from stack)
  - DFS Tree – tree containing all the edges that were used to visit nodes
    - Unused edges of G (edges not in DFS tree) are called “back edges”

DFS algorithm

```
Algorithm Depth_First_Search(Graph G)
// Graph G = {V,E}
   initialize visited to false for all vertices
   for each vertex v in V
      if v has not been visited
         dfs_helper(v)

function dfs_helper(Vertex v)
   visit node v
   for each vertex w in V adjacent to v
      if w has not been visited
         dfs_helper(w)
```

- “Visit node v” means doing whatever you need to do at each node

Common uses of DFS

- Find a spanning tree of a graph
- Find a path between two vertices v and u
- Find a path out of a maze
- Determine if a graph has a cycle
- Find all the connected components of a graph
- Search the state-space of problems for a solution (AI)
- Many more!

Efficiency of DFS

- The basic operation is the if statement in dfs_helper():

```
for each vertex w in V adjacent to v
    if w has not been visited
        dfs(w)
```

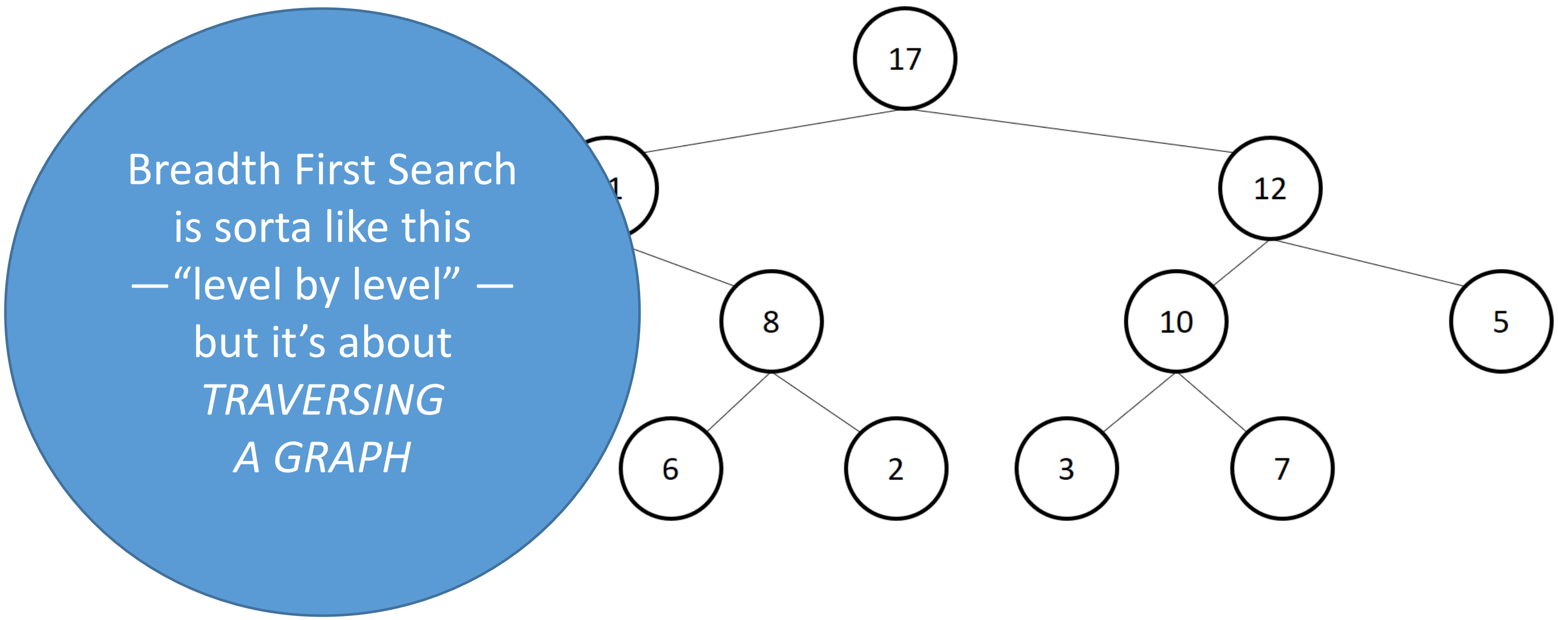
- Each time dfs_helper() is called, this loop examines all the neighbors of some vertex v … eventually it looks at ALL the neighbors of ALL the vertices
    - Therefore the number of basic operations depends on the data structure used to implement the graph
- Basically we need to visit each element of the data structure exactly once. So the efficiency must be:
    - $O(|V|^2)$ for adjacency matrix
    - $O(|V|+|E|)$ for adjacency lists

Which is better/worse?

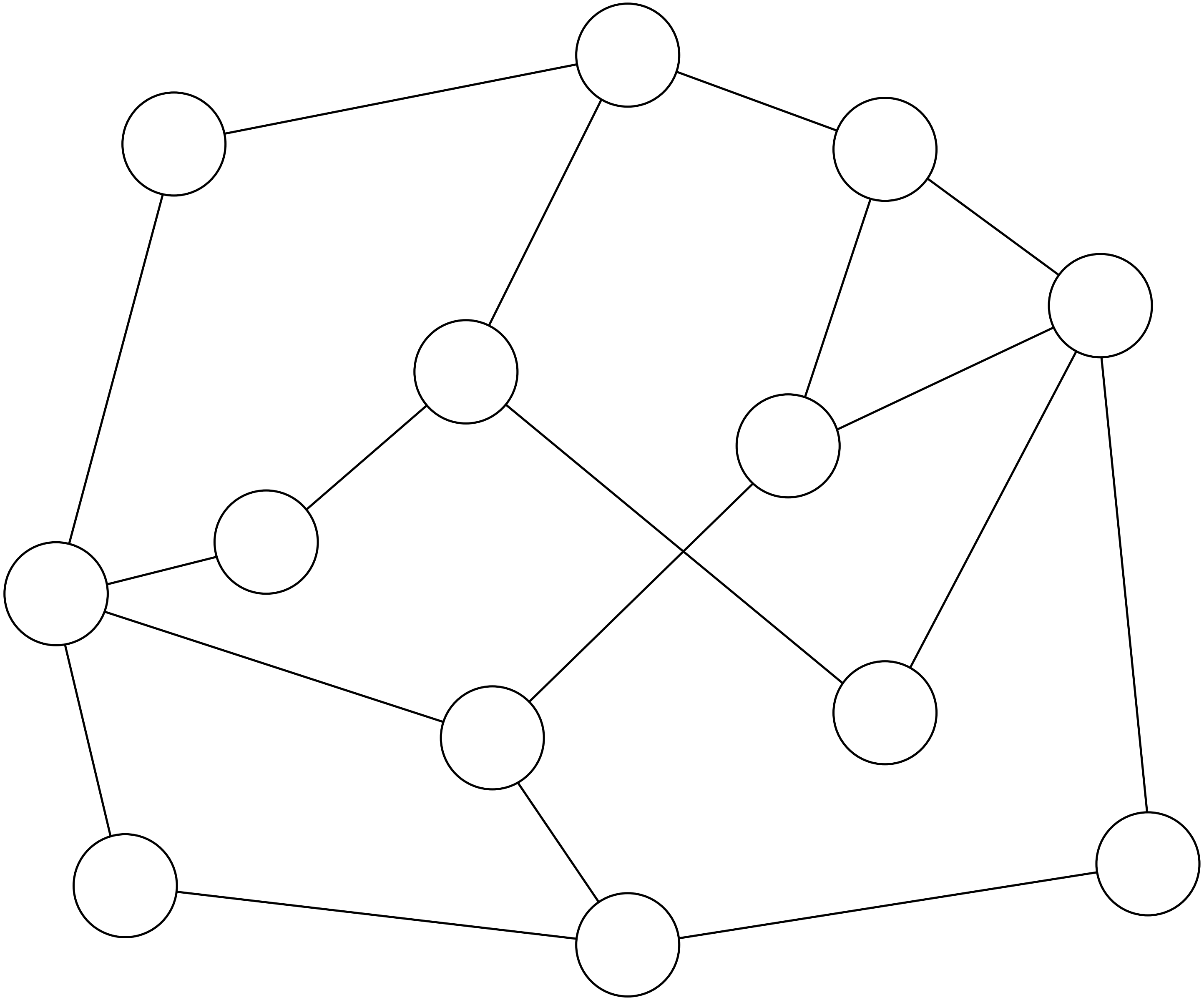
- $O(|V|^2)$
- $O(|V|+|E|)$

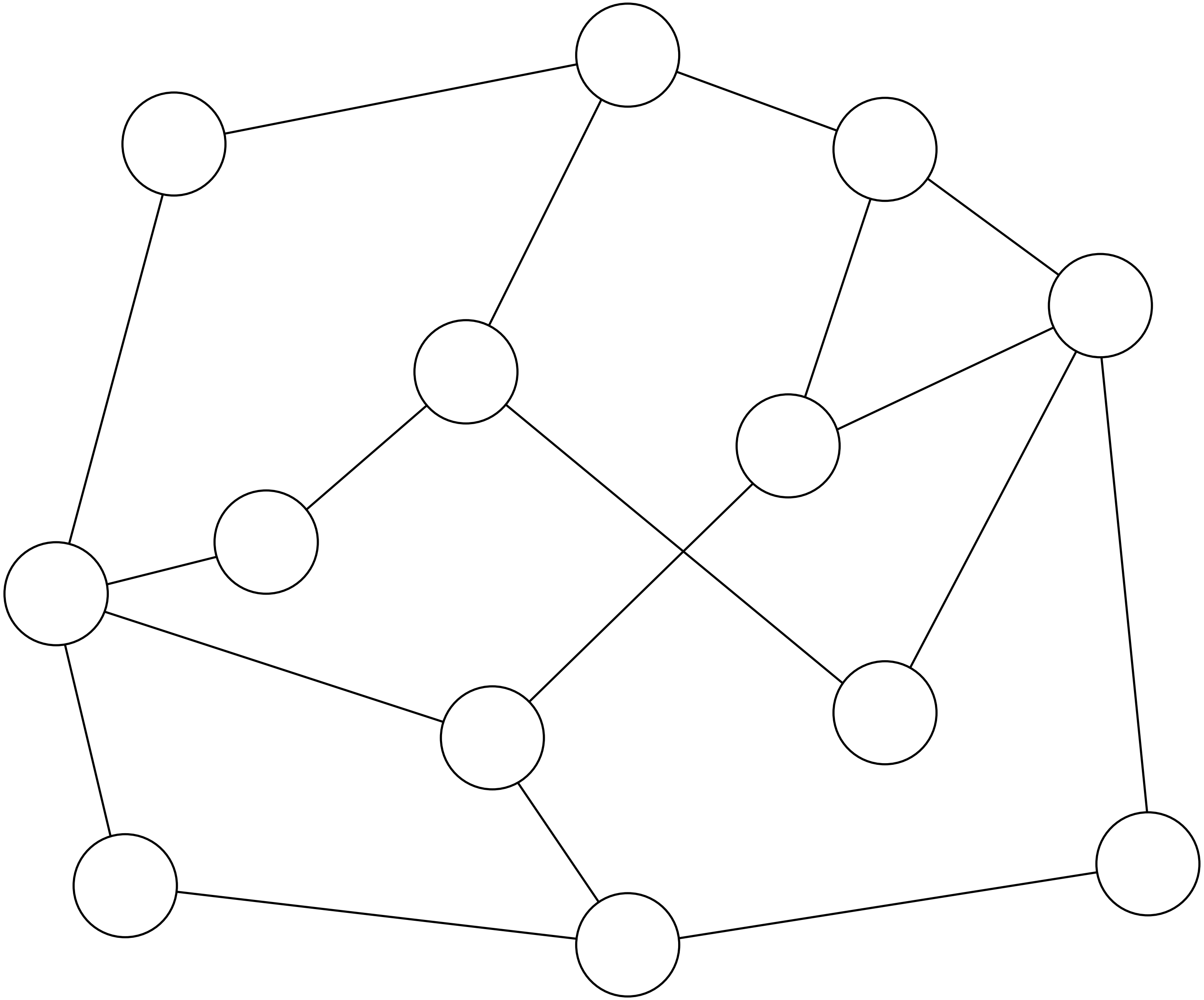
Breadth-first search (BFS)

- Recall the “trick” where we used an array to store a (very specific type of) binary tree (a heap).



17 11 12 9 8 10 5 1 4 6 2 3 7





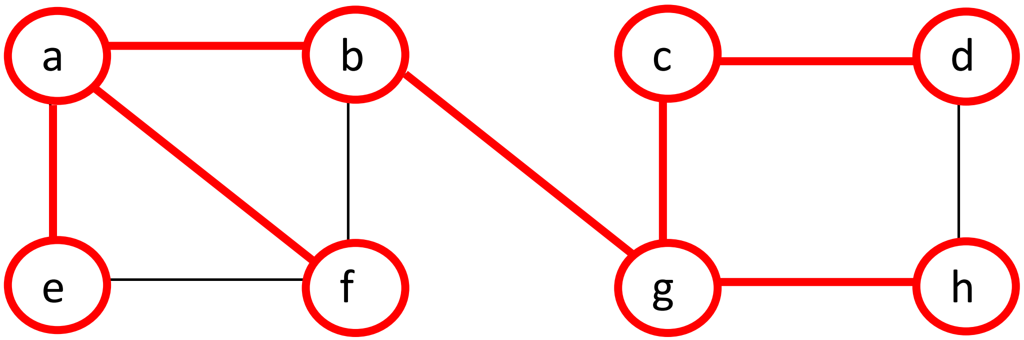
Breadth-first search (BFS)

- Visit all neighbors “the same distance” from starting vertex
  - Visit immediate neighbors first
  - Then the neighbors of all those vertices
  - Etc.

- Instead of a stack, BFS uses a queue

- Follows a tree-like route throughout the graph, but perhaps a different tree than DFS

BFS example



BFS order: a b e f g c h d

BFS algorithm

```
Algorithm Breadth_First_Search(Graph G)
// Graph G = {V,E}
   initialize visited to false for all vertices
   for each vertex v in V
      if v has not been visited
         bfs_helper(v)

function bfs_helper(Vertex v)
   visit node v
   initialize a queue Q
   add v to Q
   while Q is not empty
      for each w adjacent to Q.head
         if w has not been visited
            visit node w
            add w to Q
      Q.dequeue()
```

- Uses a queue (FIFO) to determine which vertex to visit next
- Edges that are in G but not in the resulting BFS tree are called "cross-edges"

Notes on BFS

- Same efficiency as DFS:
  - Adjacency matrix: $O(|V|^2)$
  - Adjacency list: $O(|V|+|E|)$

- Yields just one ordering of vertices (order added/deleted from queue is the same)
  - Whereas with DFS, the order that vertices are *visited* may be different from the order they get *finished* (become dead-ends)

BFS applications

- Really the same as DFS
- Sometimes one or the other may be better for specific problems

Graph Algorithms: Binary tree traversal

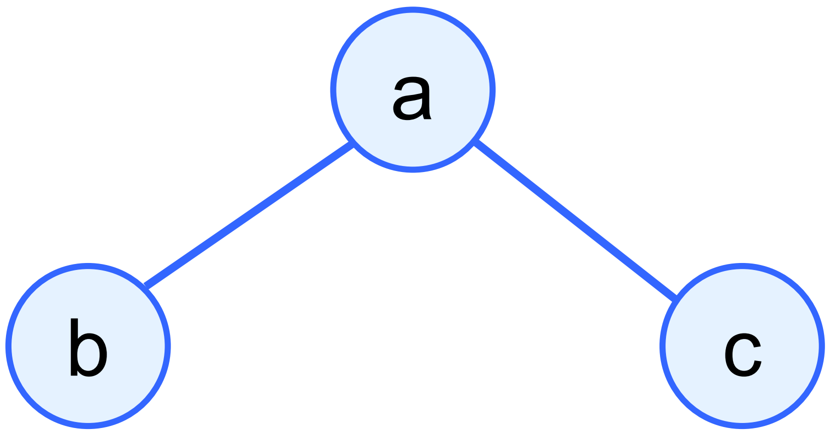
Textbook: Chapter 5.3

Tree traversal

- Traversing a tree means to visit all of the nodes of the tree
- We’ve already seen DFS and BFS (for graphs)
- Here are a few traversals specific to *binary trees*:
  - Preorder – root *before* the children
  - Inorder – root *between* the children
  - Postorder – root *after* the children

Preorder traversal

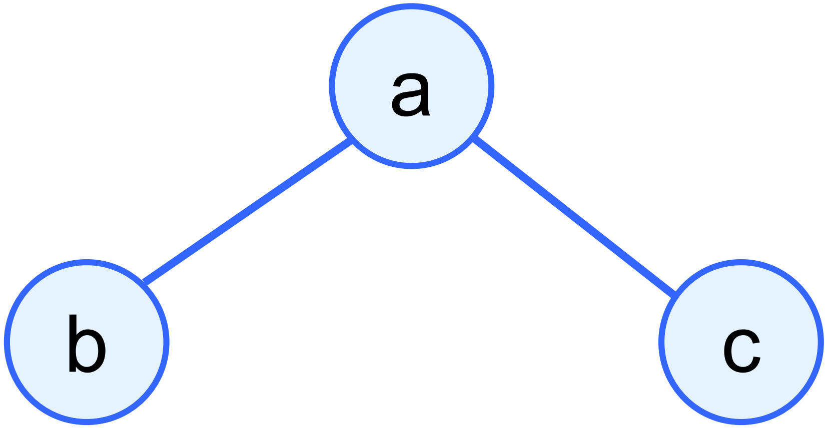
1. Visit the root
2. Traverse the left subtree
3. Traverse the right subtree



Preorder traversal is: a b c

Inorder traversal

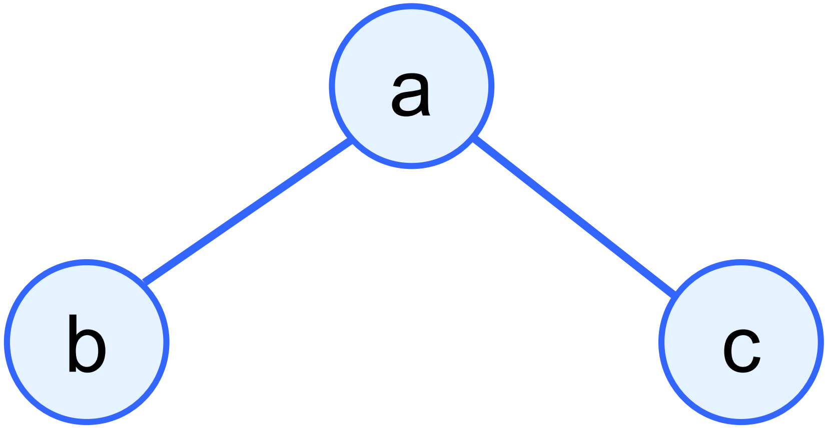
1. Traverse the left subtree
2. Visit the root
3. Traverse the right subtree



Inorder traversal is: b a c

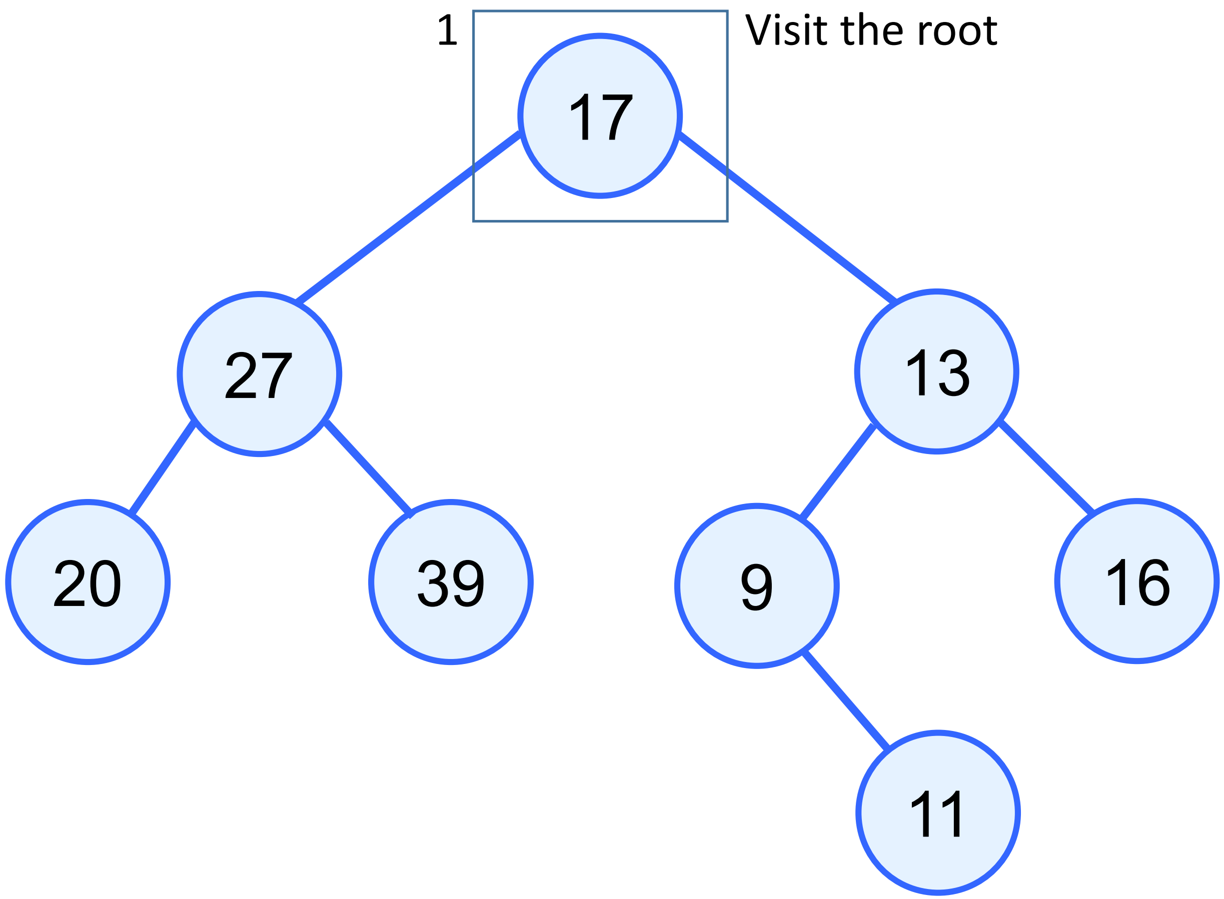
Postorder traversal

1. Traverse the left subtree
2. Traverse the right subtree
3. Visit the root



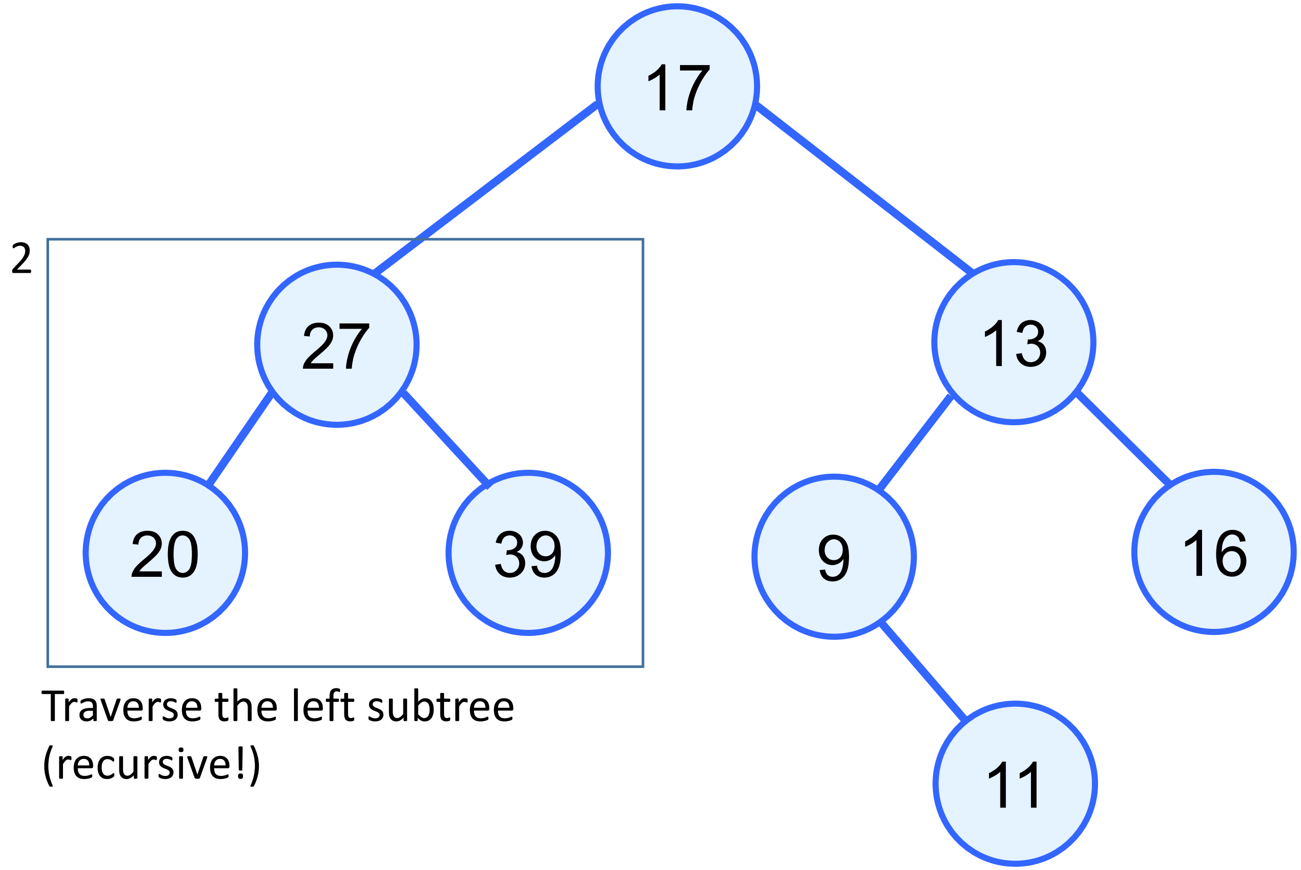
Postorder traversal is: b c a

Preorder example



17

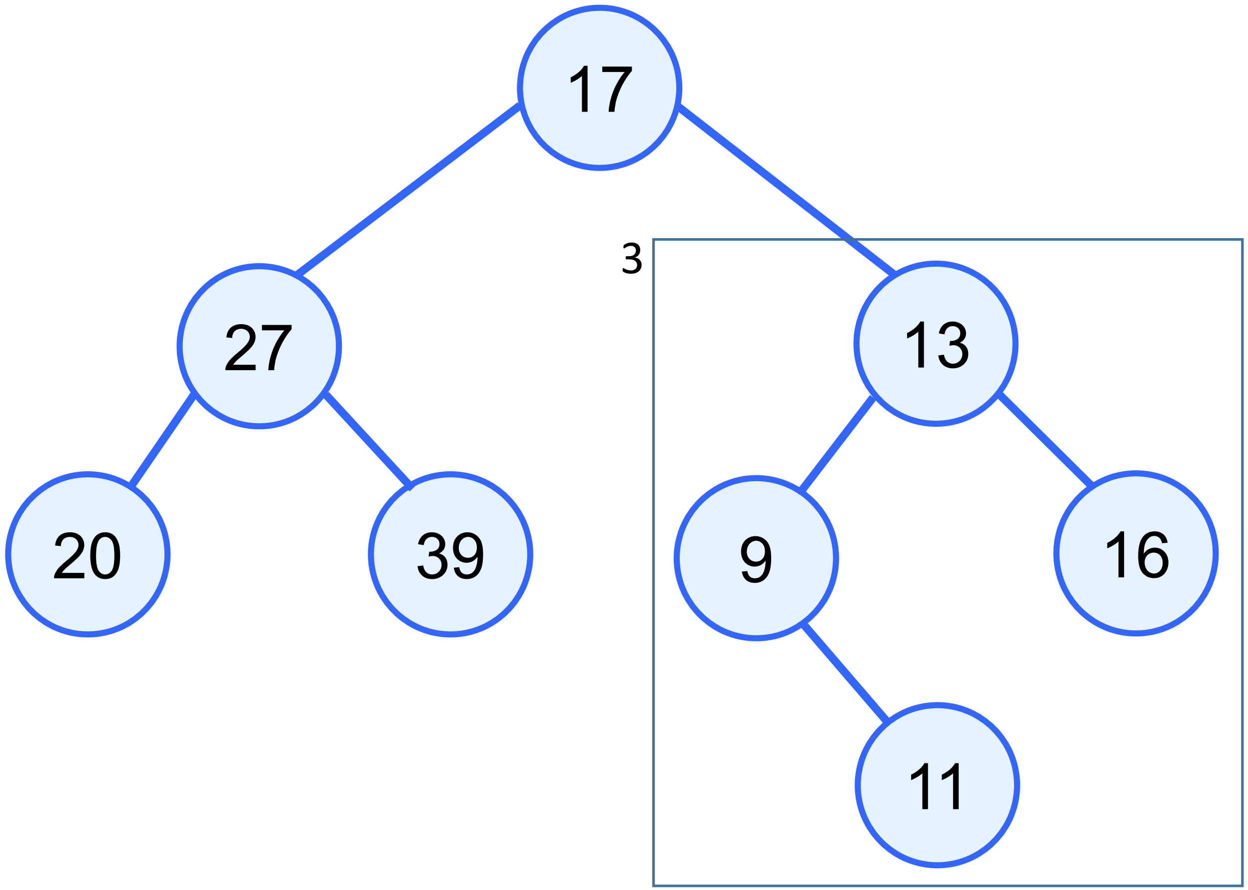
Preorder example



Traverse the left subtree (recursive!)

	root	left	right
17	27	20	39

Preorder example

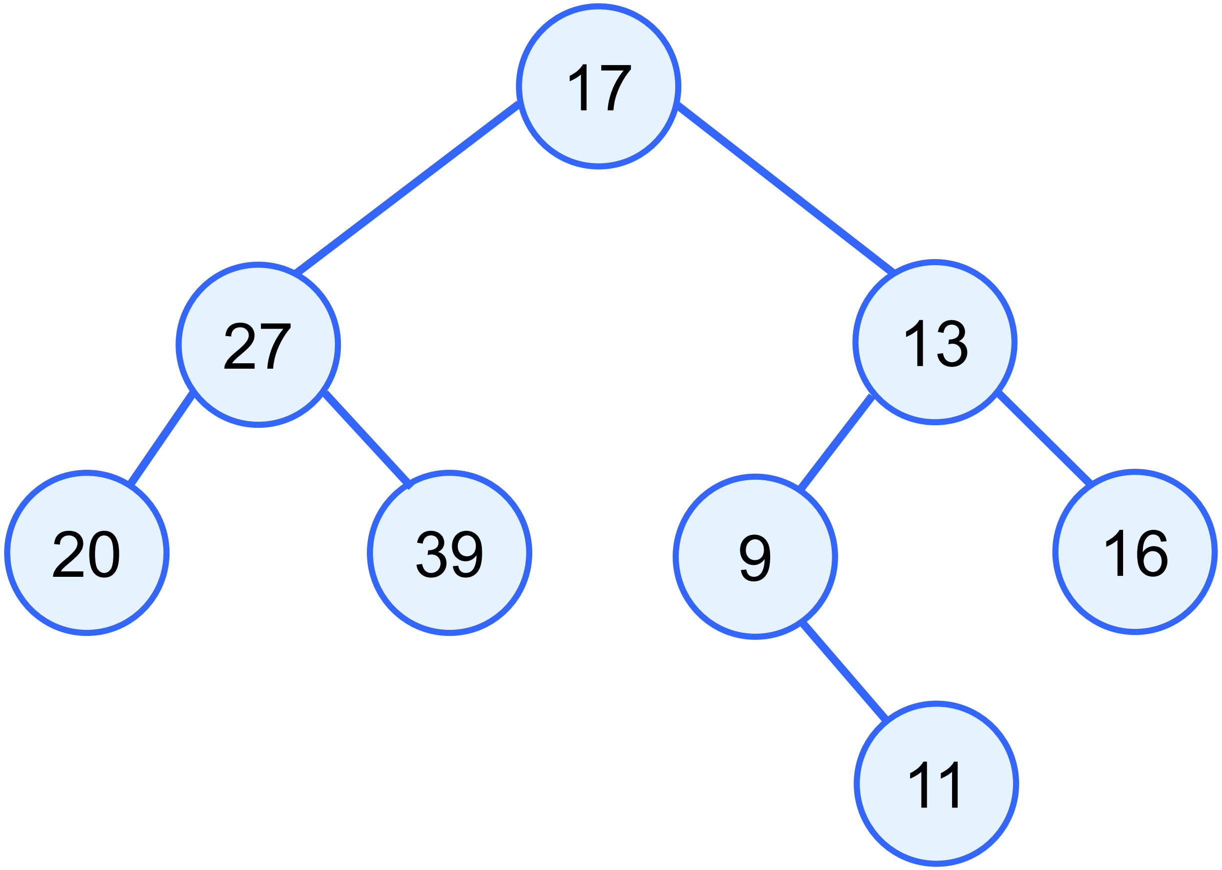


Traverse the right subtree (recursive within recursive!)

17 27 20 39

root	left		right
	root (left)	right	
13	9	11	16

Preorder example

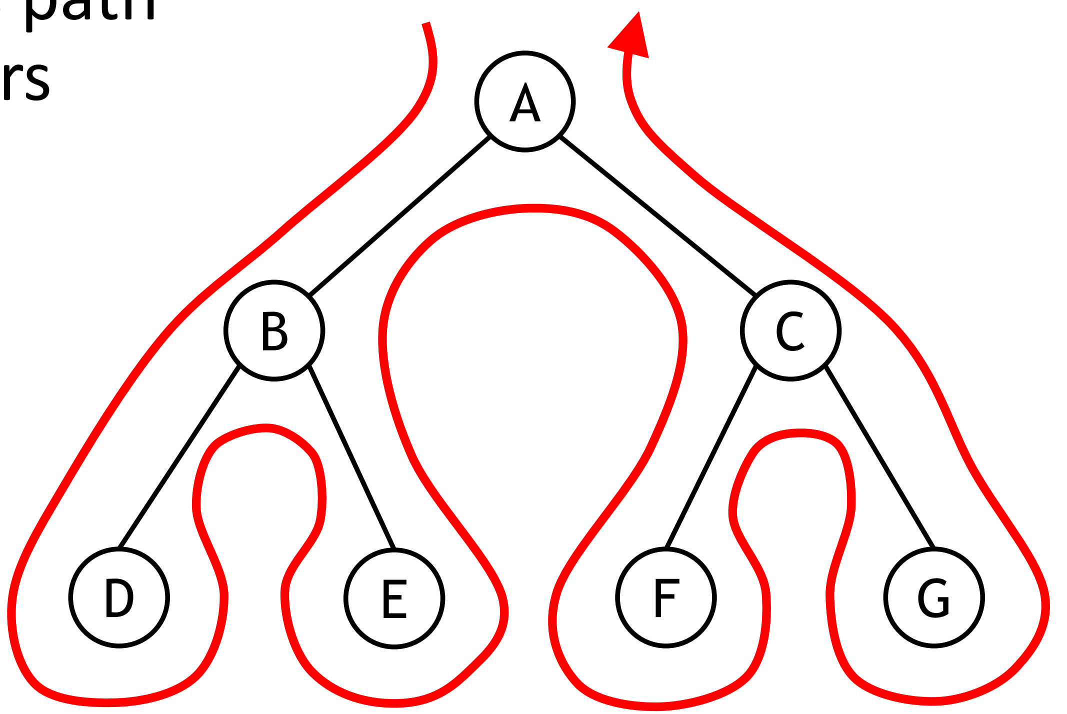


17 27 20 39 13 9 11 16

Another way to think about it

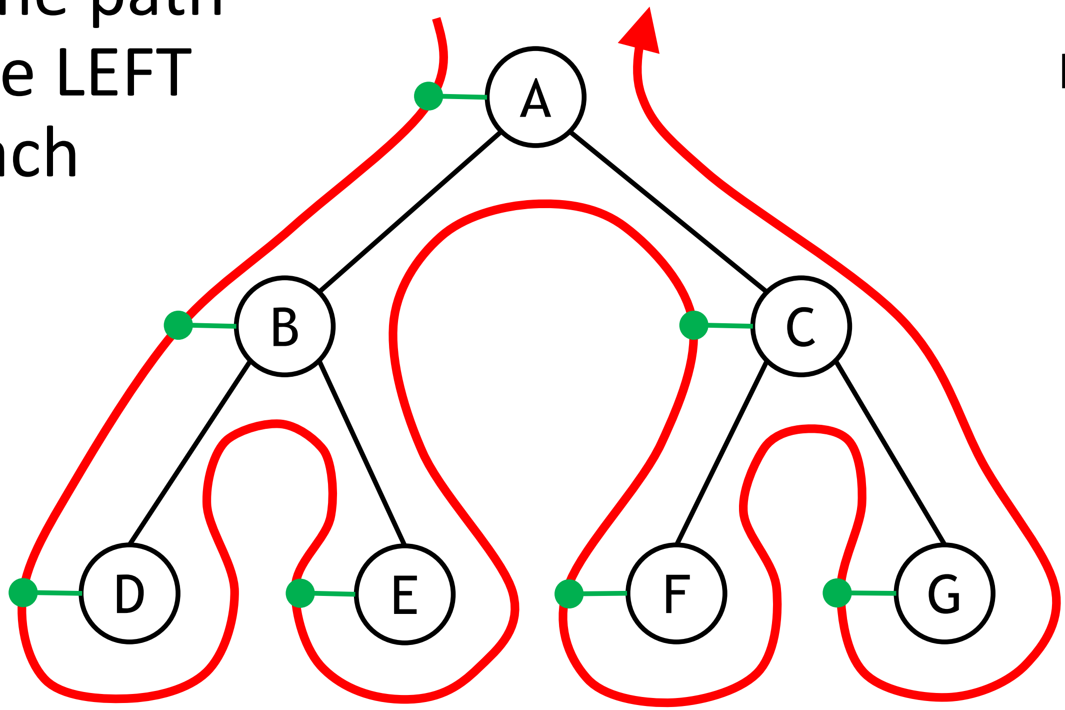
- Consider this path that meanders past all of the nodes

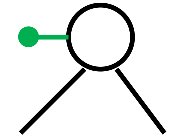
- ALL three traversals follow this path!



Preorder traversal

- Is when the path passes the LEFT side of each node

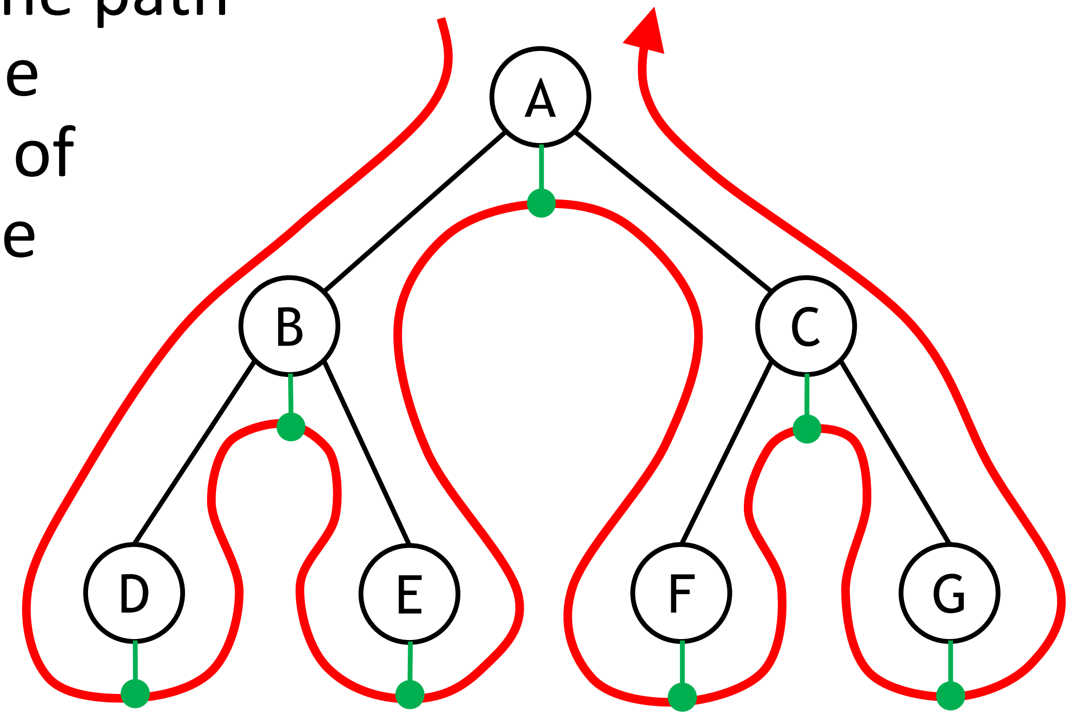


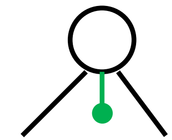


Preorder: A B D E C F G

Inorder traversal

- Is when the path passes the BOTTOM of each node

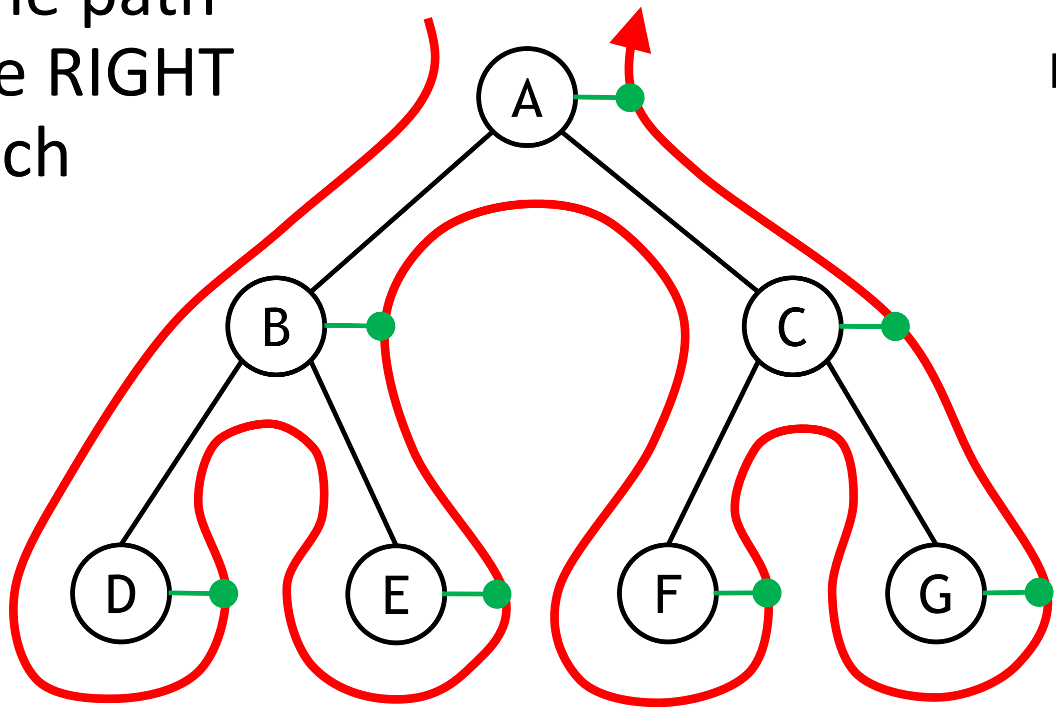


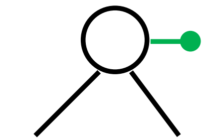


Inorder: D B E A F C G

Postorder traversal

- Is when the path passes the RIGHT side of each node





Postorder: D E B F G C A

Pseudocode

```
Algorithm inOrder(Node N)
if N != null
   inOrder(N.leftChild)
   Print N.value
   inOrder(N.rightChild)
```

```
Algorithm preOrder(Node N)
if N != null
   Print N.value
   preOrder(N.leftChild)
   preOrder(N.rightChild)
```

```
Algorithm postOrder(Node N)
if N != null
   postOrder(N.leftChild)
   postOrder(N.rightChild)
   Print N.value
```

What if I told you

Preorder → A B D E C F G

Inorder → D B E A F C G

Fun facts about pre/in/postorder

- Given pre + in, you can reconstruct the tree
  - (and also determine postorder)
- Given post + in, you can reconstruct the tree
  - (and also determine preorder)
- Given pre + post, you can only *sometimes* reconstruct the tree
  - For you to ponder: under what condition(s)?

Some problems solvable with graph traversal

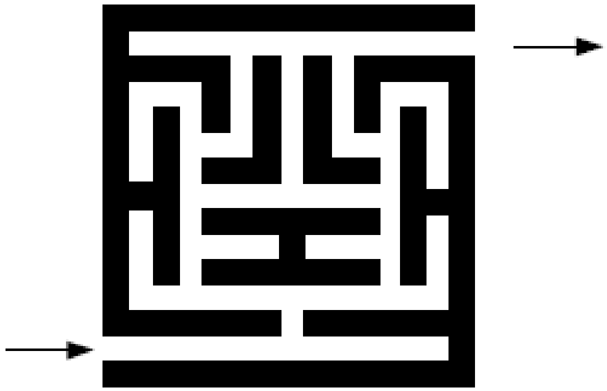
Problem 1: Spanning Tree

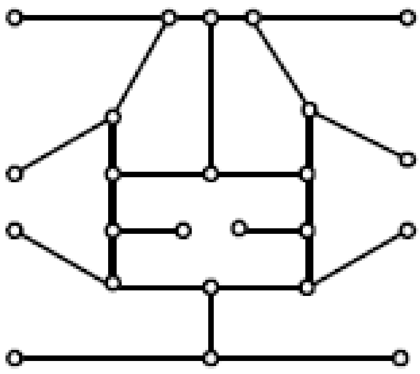
- Given a connected graph G, find a spanning tree T
  - This is a straight-up application of BFS (or DFS)
  - Build a new graph (the spanning tree) as we go
    - Initialize a new graph T
    - Each time we visit a vertex, add the edge we used to T

- BFS or DFS?
  - BFS usually gives shorter paths between vertices

Problem 2: Solving a Maze

- Represent maze as a graph
  - Nodes for start, finish, intersections, and dead-ends
  - Find a path from *start* to *finish*
- BFS or DFS?
  - If interested in end-result:
    - BFS will find the shortest total path
  - If actually walking while solving:
    - DFS tends to result in less actual walking
    - BFS backtracks to parent nodes too often





Problem 3: Shortest Path

- Find the shortest path between two vertices u and w
- BFS or DFS?
  - BFS will find a shortest path
  - DFS will find a path – maybe not the shortest one

- Idea of algorithm (and why it works):
  - First, use bfs(u) to create a spanning tree T with u as the root. Note that all paths that appear in T are the shortest paths from u to their respective vertices
  - Then, use DFS on T to find a path from u to w (as in the maze problem)

Problem 4: Determine Connectivity

- Determine if a graph is connected
- BFS or DFS?
  - Either will work
  - Think about this yourself!
  - What modification(s) do you need to make to the algorithm to answer this question?