Python for Data Scientists
L11 : Invariants, Graph Traversal, BFS, and DFS

Shirin Tavara

Outline of the lecture

Applications of trees and graphs

Recap of graphs, trees, binary search tree

Invariants

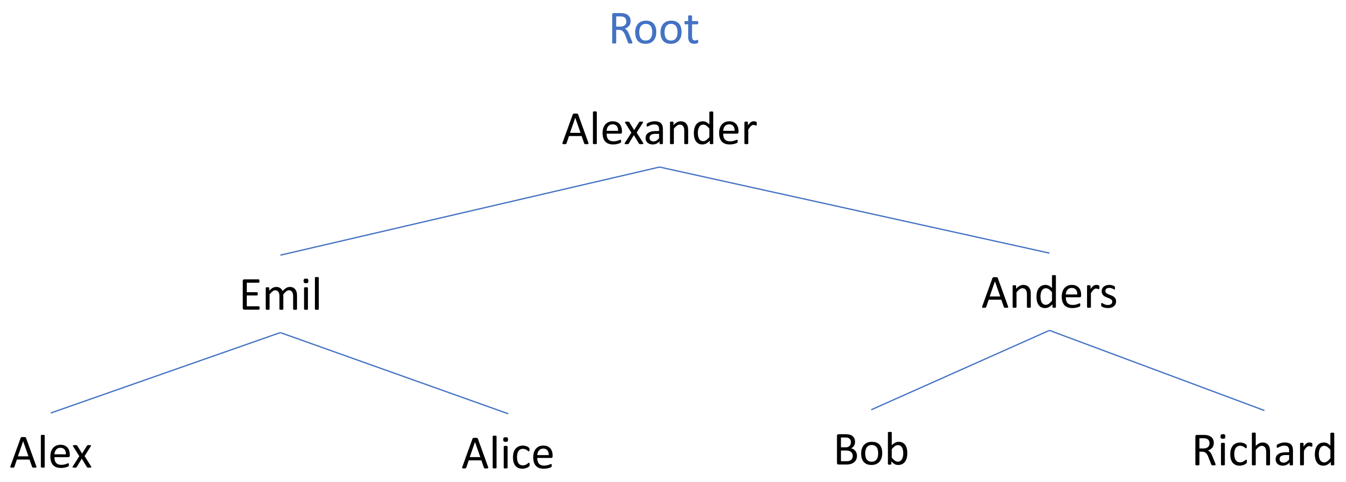
Graph traversal

- Breadth first search, depth first search

ReCap - Binary Trees

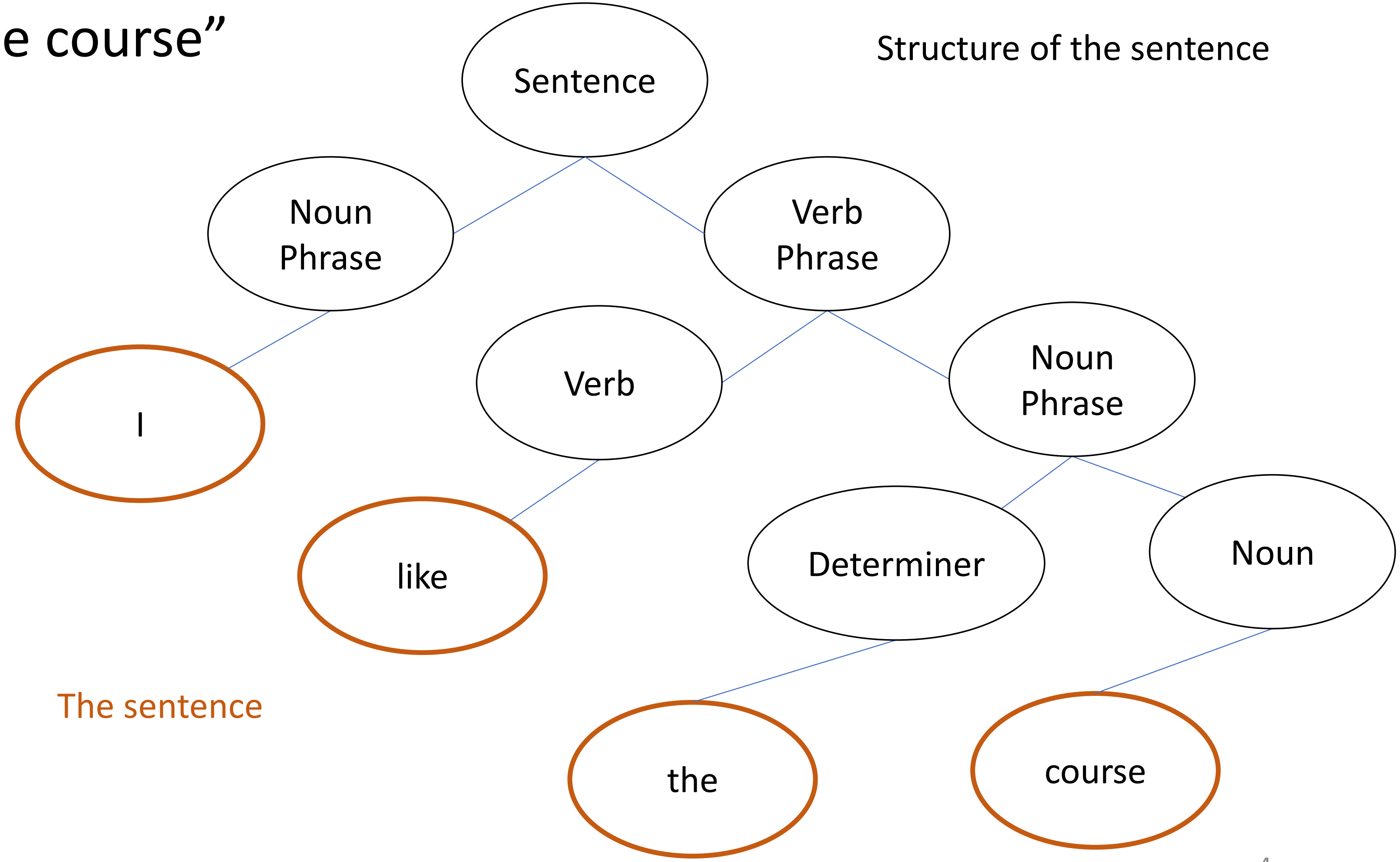
Binary tree represents the nodes connected by edges.

- A Root node.
- Every node has one parent node, except the root node.
- Each node has at most two child nodes.



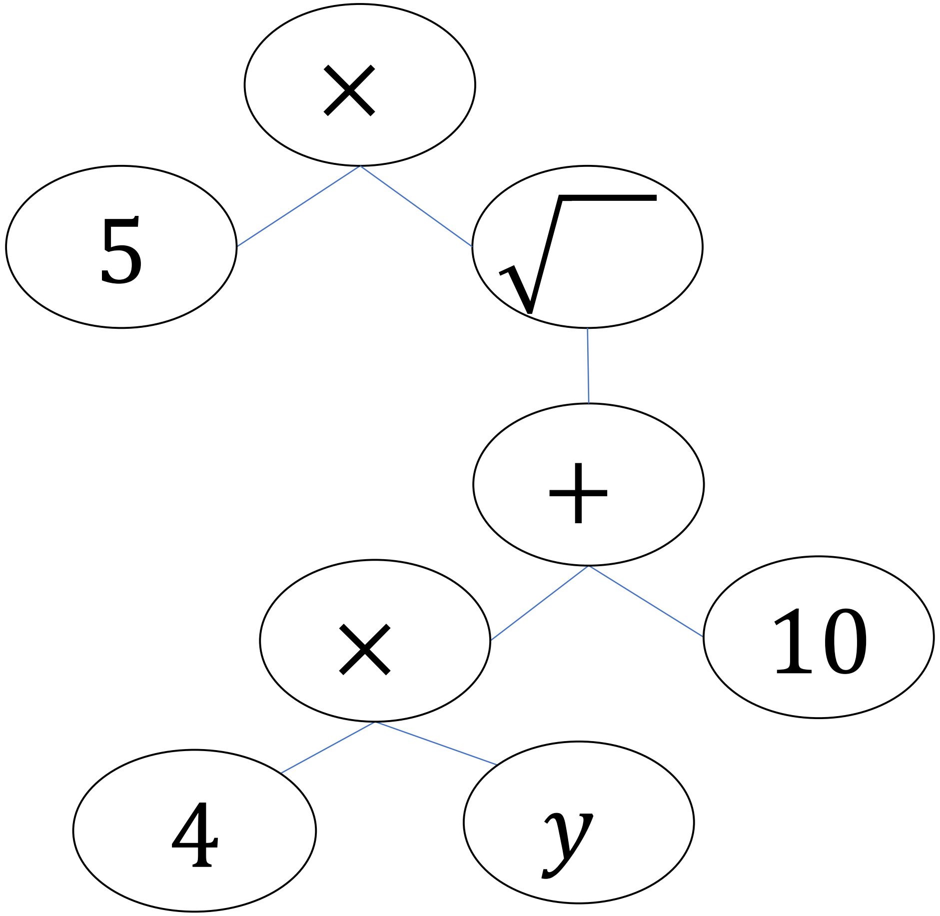
Syntax Tree

Sentence: “I like the course”



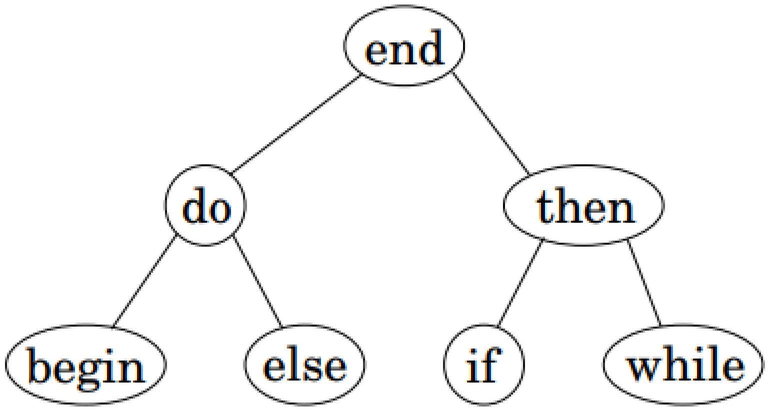
Syntax Tree

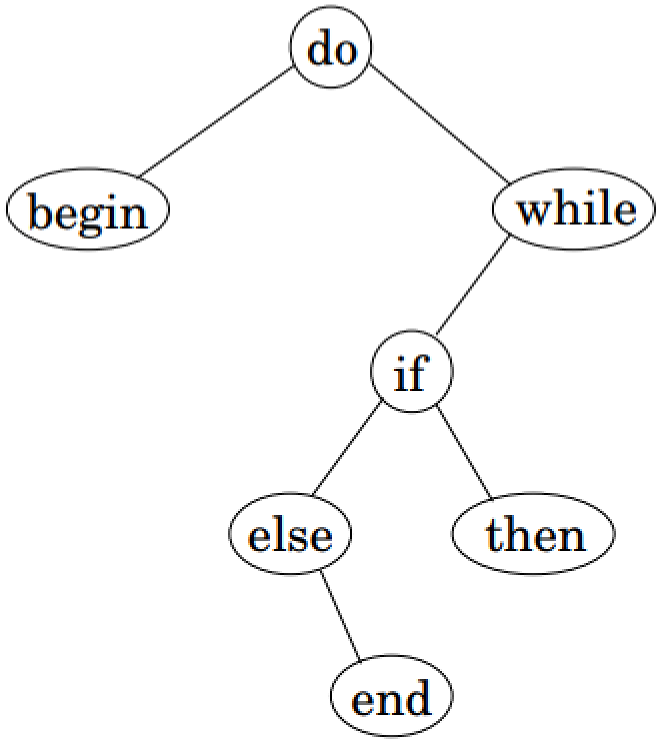
Expression: $5\sqrt{4y + 10}$



Binary Search Tree

BST for accessing keywords of programming languages with different costs considering the frequency that keywords are accessed

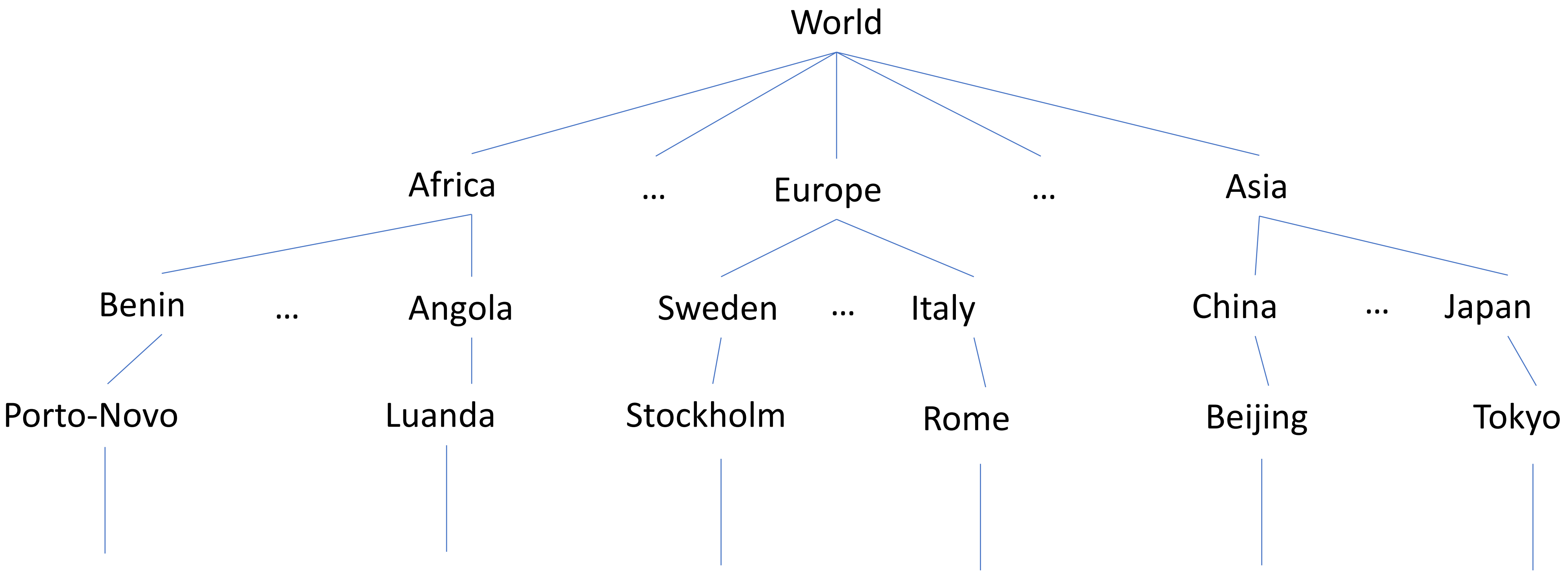




Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. Algorithms (1st Edition). McGraw-Hill Higher Education. 2008.

Hierarchy

Geographical hierarchy

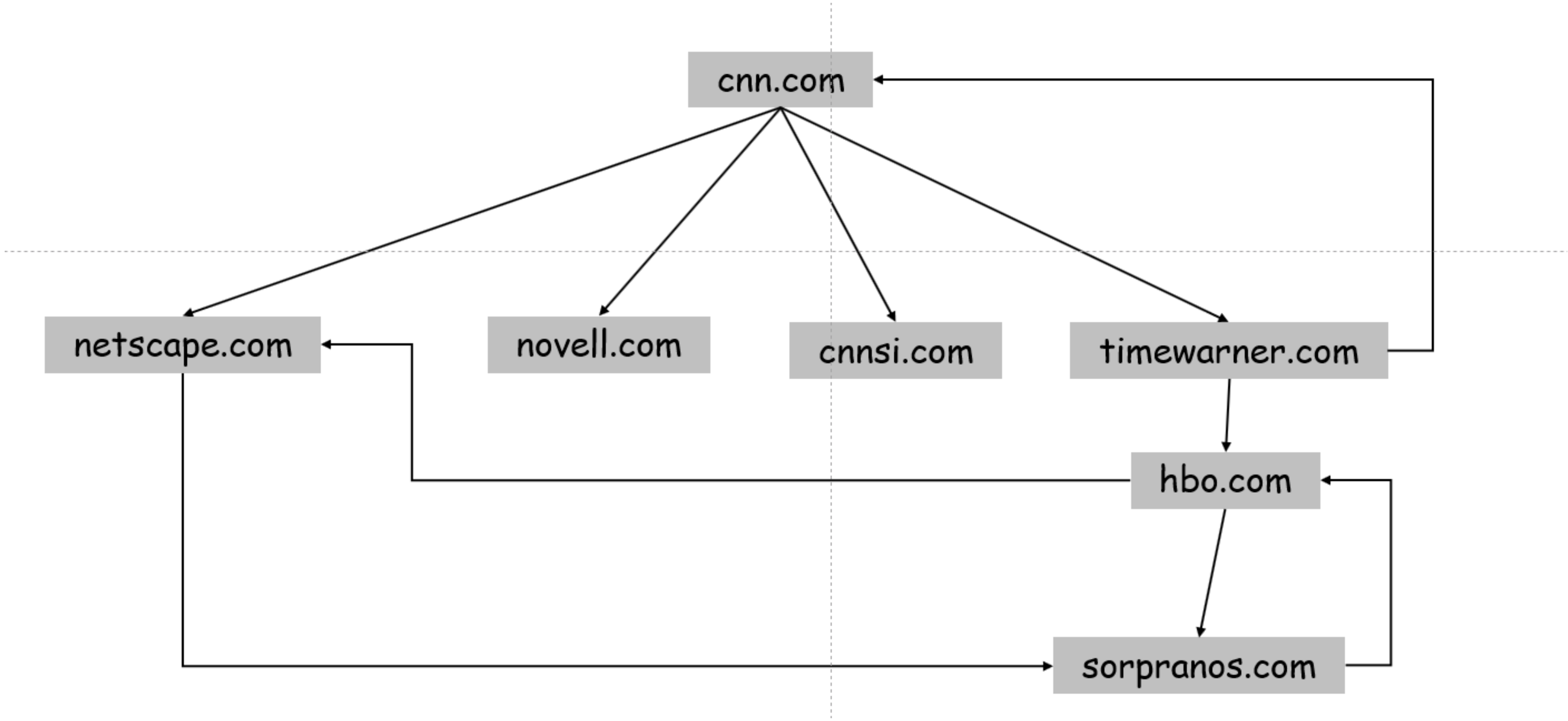


Graph Applications - World Wide Web

Web Graph

Node: web page.

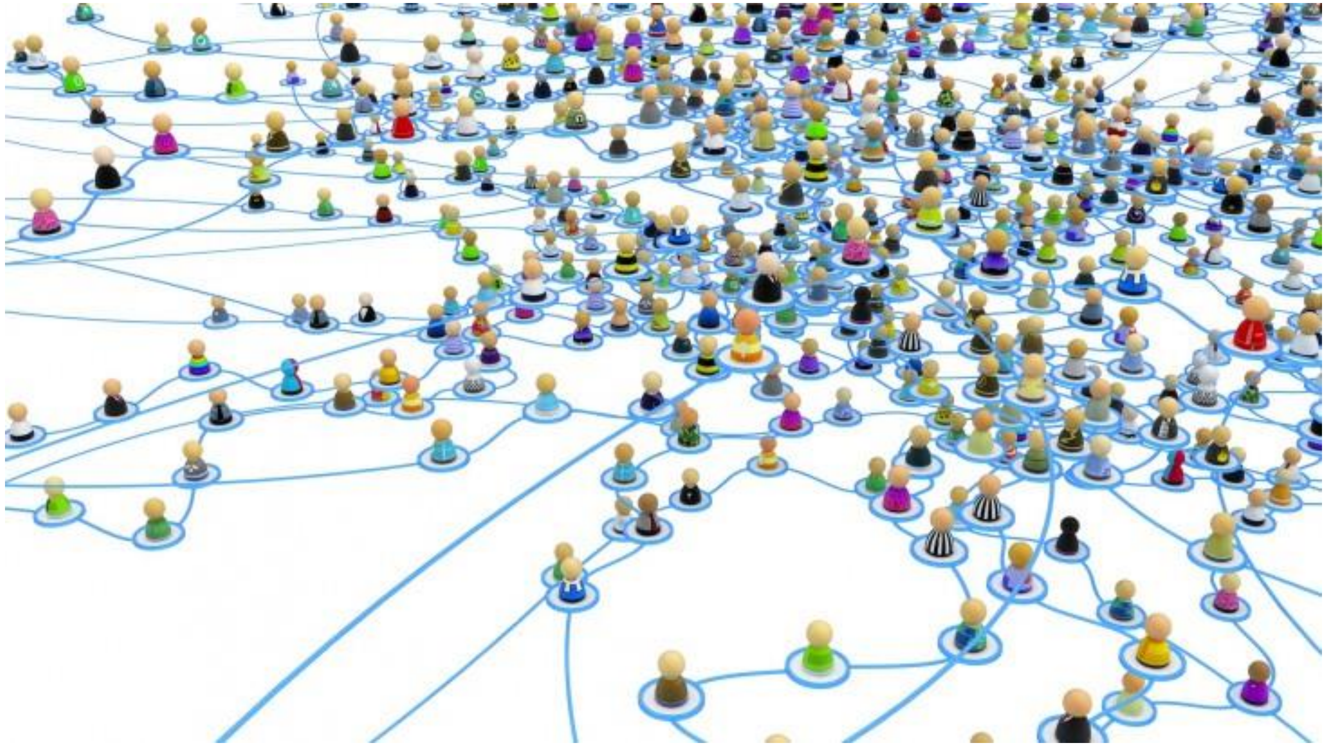
Edge: hyperlink from one page to another.



Graph Applications - Social Network

Social network graph.

- Node: people.
- Edge: relationship between two people.

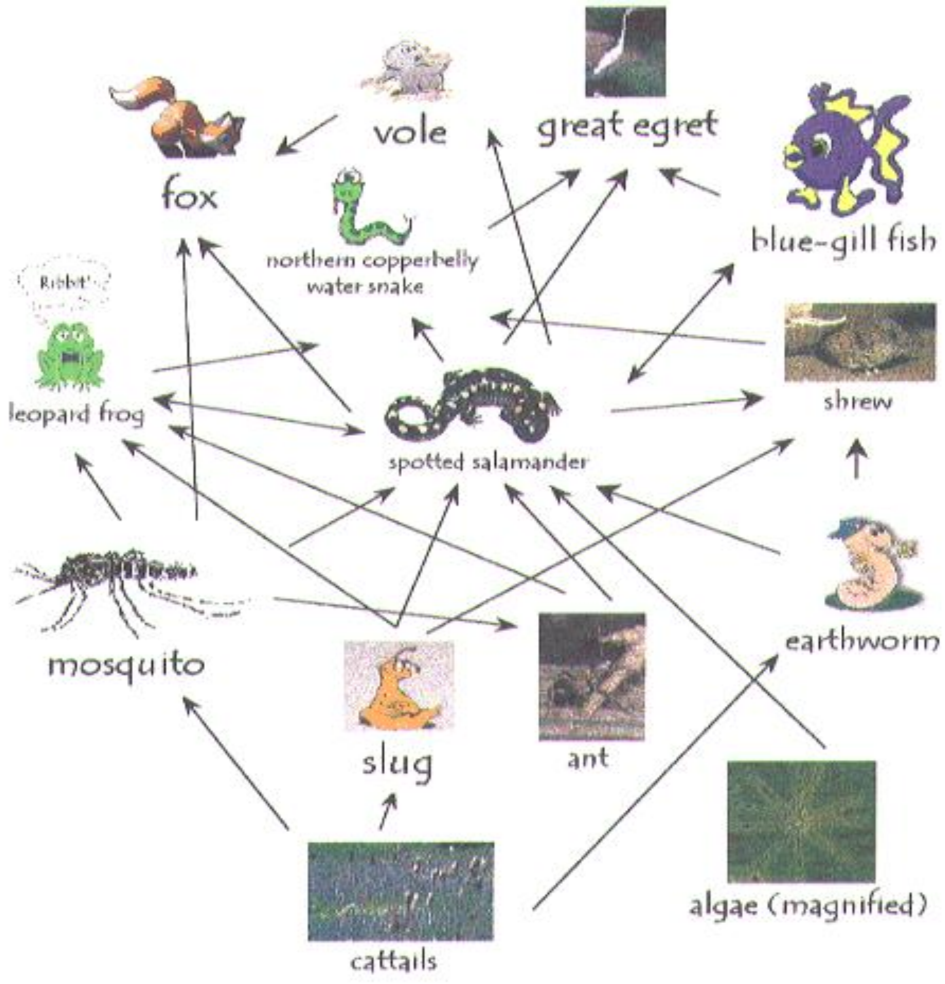


https://cdn0.tnwcdn.com/wp-content/blogs.dir/1/files/2013/11/social-network-links-730x410.jpg

Graph Applications - Ecological Food Web

Food web graph

- Node = species.
- Edge = from prey to predator. (victim to killer)



Some Other Graph Applications

Graph	Nodes	Edges
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

Traversal order in a binary tree

- PreOrder traversal

  Root-left subtree- right subtree

- InOrder traversal

  Left subtree- root- right subtree

- PostOrder traversal

  Left subtree- right subtree - root

Binary tree : binarytree package

```python
from binarytree import Node
root = Node(2)
root.left = Node(4)
root.right = Node(8)

# Getting binary tree
print('Binary tree :', root)

# Getting list of nodes
print('List of nodes :', list(root))

# Getting inorder of nodes
print('Inorder of nodes :', root.inorder)

# Checking tree properties
print('Size of tree :', root.size)
print('Height of tree :', root.height)

# Get all properties at once
print('Properties of tree : \n', root.properties)
```

```
Binary tree :
  2
 / \
4   8

List of nodes : [Node(2), Node(4), Node(8)]

Inorder of nodes : [Node(4), Node(2), Node(8)]

Size of tree : 3
Height of tree : 1
Properties of tree :
 {'height': 1, 'size': 3, 'is_max_heap': False, 'is_min_heap': True, 'is_perfect': True, 'is_strict': True, 'is_complete': True, 'leaf_count': 2, 'min_node_value': 2, 'max_node_value': 8, 'min_leaf_depth': 1, 'max_leaf_depth': 1, 'is_bst': False, 'is_balanced': True, 'is_symmetric': False}
```

Binary tree from a given list

```python
from binarytree import build


# List of nodes
nodes =[2, 4, 8, 16, 32, 64, None]

# Builidng the binary tree
binary_tree = build(nodes)
print( binary_tree)

# Getting list of nodes from binarytree
print(binary_tree.values)
```

```
    ___2___
   /       \
  _4       _8
 /  \      /
16   32   64
```

```
[2, 4, 8, 16, 32, 64]
```

Random binary tree

```python
from binarytree import tree

# Create a random binary tree of any height
root = tree()
print(root)

# Create a random binary tree of given height
rootRandom = tree(height = 3)
print(rootRandom)

# Create a random perfect binary tree of given height
rootRandomPer = tree(height = 3, is_perfect = True)
print(rootRandomPer)
```

```
      5
     / \
  __4   3__
 /         \
11          10
  \        /  \
   7      0    9
```

```
      __4______
     /         \
   __3        _14__
  /   \      /     \
 6     2   _13      0
/ \        /       / \
8  1      10      7   5
```

```
        ________8_____
       /              \
   ___10__           __14__
  /       \         /      \
 _13       12      0        6
/   \     /  \    / \      / \
11   9   2    4  7   3    1   5
```

Binary Search Tree

```python
from binarytree import bst

# Create a random BST of any height
root = bst()
print('BST of any height : \n', root)

# Create a random BST of given height
root2 = bst(height = 2)
print('BST of given height : \n', root2)

# Create a random perfect BST of given height
root3 = bst(height = 2, is_perfect = True)
print('Perfect BST of given height : \n', root3)
```

```
BST of any height :

        ____13
       /      \
    __11       14
   /    \
  8      12
 / \
5   9


BST of given height :

    __6
   /
  1
 / \
0   3


Perfect BST of given height :

    __3__
   /     \
  1       5
 / \     / \
0   2   4   6
```

Binary Search Tree

```
class Node:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

    """ find a specific value in the tree """
    def findval(root, lkpval):
        if lkpval < root.data:
            if root.left == None:
                return str(lkpval) + " Not Found"
            return root.left.findval(lkpval)
        elif lkpval > root.data:
            if root.right == None:
                return str(lkpval) + " Not Found"
            return root.right.findval(lkpval)
        else:
            print(str(root.data) + ' is found’)

def insert(node, data):
   # 1. If the tree is empty, return a new, single
node
   if node == None:
      return (Node(data))
   else:
      # 2. Otherwise, recur down the tree
      if data <= node.data:
         node.left = insert(node.left, data)
      else:
         node.right = insert(node.right, data)
      return node

def minValue(node):
   current = node
   # Loop down to find the lefmost leaf
   while(current.left is not None):
      current = current.left
   return current.data

root = None
root = insert(root,7)
insert(root,1)
insert(root,6)
insert(root,8)
insert(root,4)
insert(root,9)
print ("Minimum value in BST is
:",(minValue(root)))
print(root.findval(9))
print(root.findval(11))
```

Binary Search Trees – Measuring in 4 languages

C

Generating random array with 2,000,000 values	Filling tree with 2,000,000 nodes	Traversing all 2,000,000 nodes	Peak memory used
22 ms	12.86 sec	128 ms	53 MB

C#

Version	Generating random array with 2,000,000 values	Filling tree with 2,000,000 nodes	Traversing all 2,000,000 nodes	Peak memory used
.NET 4.6/C# 4.0	25 ms	13.07 sec	128 ms	58 MB

Java (1.8)

Generating random array with 2,000,000 values	Filling tree with 2,000,000 nodes	Traversing all 2,000,000 nodes	Peak memory used
21 ms	12.2 sec	126 ms	70 MB

Python

Version	Generating random array with 2,000,000 values	Filling tree with 2,000,000 nodes	Traversing all 2,000,000 nodes	Peak memory used
Python 2.7.10	1.9 sec	98 sec	750 ms	377 MB
Python 3.5.0	2.55 sec	114.23 sec	828 ms	220 MB

https://aaronjwood.com/articles/binary-search-trees/

Graphs

A graph consists of:

- Nodes or vertices
- The links between the nodes, a set of pairs of vertices that are connected

Graph representation:

- Adjacency matrix
- Adjacency list

Graph	Adjacency Matrix	Adjacency List
Space complexity	$O(\|V\|^2)$ or $O(n^2)$	$O(\|V\| + \|E\|)$
IsConnected(n_i, n_j)	$O(1)$	$O(\|\mathrm{V}\|)$
Add(n_k)	$O(\|V\|^2)$	$O(1)$
GetAdjacent(n_k)	$O(\|\mathrm{V}\|)$	$O(\|\mathrm{E}\|)$

Graphs Implementation

```
# Create the dictionary with graph
elements
graph = { "1" : ["2","3"],
          "2" : ["1", "3", "4", "5"],
          "3" : ["1", "2", "5", "7", "8"],
          "4" : ["2", "5"],
          "5" : ["2", "3", "4", "6"],
          "6" : ["5"],
          "7" : ["3", "8"],
          "8" : ["3", "7"]
         }

# Print the graph
print(graph)
```

```
{'1': ['2', '3'], '2': ['1', '3', '4', '5'], '3': ['1', '2',
'5', '7', '8'], '4': ['2', '5'], '5': ['2', '3', '4', '6'],
'6': ['5'], '7': ['3', '8'], '8': ['3', '7']}
```

https://insideaiml.com/article-details/Python---Graphs-928

Graphs: Display Graph Vertices

```
class graph:
    def __init__(self,gdict=None):
        if gdict is None:
            gdict = []
        self.gdict = gdict

# Get the keys of the dictionary
    def getVertices(self):
        return list(self.gdict.keys())

gElements = { "1" : ["2","3"],
          "2" : ["1", "3", "4", "5"],
          "3" : ["1", "2", "5", "7", "8"],
          "4" : ["2", "5"],
          "5" : ["2", "3", "4", "6"],
          "6" : ["5"],
          "7" : ["3", "8"],
          "8" : ["3", "7"]
         }
g = graph(gElements)
print(g.getVertices())
```

['1', '2', '3', '4', '5', '6', '7', '8']

https://insideaiml.com/article-details/Python---Graphs-928

Graphs: Display Distinct Graph Edges

```python
class graph:
    def __init__(self,gdict=None):
        if gdict is None:
            gdict = {}
        self.gdict = gdict

    def edges(self):
        return self.findedges()

# Find the distinct list of edges
    def findedges(self):
        edgename = []
        for vrtx in self.gdict:
            for nxtvrtx in self.gdict[vrtx]:
                if {nxtvrtx, vrtx} not in edgename:
                    edgename.append({vrtx, nxtvrtx})
        return edgename
```

```python
gElements = { "1" : ["2","3"],
              "2" : ["1", "3", "4", "5"],
              "3" : ["1", "2", "5", "7", "8"],
              "4" : ["2", "5"],
              "5" : ["2", "3", "4", "6"],
              "6" : ["5"],
              "7" : ["3", "8"],
              "8" : ["3", "7"]
            }
g = graph(gElements)
print(g.edges())
```

[{'1', '2'}, {'1', '3'}, {'2', '3'}, {'2', '4'}, {'5', '2'}, {'5', '3'}, {'3', '7'}, {'8', '3'}, {'5', '4'}, {'5', '6'}, {'8', '7'}]

https://insideaiml.com/article-details/Python---Graphs-928

Graphs : Adding A Vertex

```python
class graph:
    def __init__(self,gdict=None):
        if gdict is None:
            gdict = {}
        self.gdict = gdict
    def getVertices(self):
        return list(self.gdict.keys())

# Add the vertex as a key
    def addVertex(self, vrtx):
       if vrtx not in self.gdict:
            self.gdict[vrtx] = []
```

```python
gElements = { "1" : ["2","3"],
              "2" : ["1", "3", "4", "5"],
              "3" : ["1", "2", "5", "7", "8"],
              "4" : ["2", "5"],
              "5" : ["2", "3", "4", "6"],
              "6" : ["5"],
              "7" : ["3", "8"],
              "8" : ["3", "7"]
            }
g = graph(gElements)
print(g.getVertices())
g.addVertex("9")
print(g.getVertices())
```

['1', '2', '3', '4', '5', '6', '7', '8']
['1', '2', '3', '4', '5', '6', '7', '8', '9']

https://insideaiml.com/article-details/Python---Graphs-928

Graphs : Adding an edge

```
class graph:
    def __init__(self,gdict=None):
        if gdict is None:
            gdict = {}
        self.gdict = gdict

    def edges(self):
        return self.findedges()

    def AddEdge(self, edge):
        edge = set(edge)
        (vrtx1, vrtx2) = tuple(edge)
        if vrtx1 in self.gdict:
            self.gdict[vrtx1].append(vrtx2)
        else:
            self.gdict[vrtx1] = [vrtx2]
    +def findedges(self):
      #(definition in slide 22)
```

```
gElements = { "1" : ["2","3"],
              "2" : ["1", "3", "4", "5"],
              "3" : ["1", "2", "5", "7", "8"],
              "4" : ["2", "5"],
              "5" : ["2", "3", "4", "6"],
              "6" : ["5"],
              "7" : ["3", "8"],
              "8" : ["3", "7"]
            }
g = graph(gElements)
print(g.edges())
g.AddEdge({'1','9'})
g.AddEdge({'5','9'})
print(g.edges())
```

[{'2', '1'}, {'1', '3'}, {'2', '3'}, {'2', '4'}, {'2', '5'}, {'3', '5'}, {'3', '7'}, {'8', '3'}, {'4', '5'}, {'6', '5'}, {'8', '7'}]

[{'2', '1'}, {'1', '3'}, **{'1', '9'}**, {'2', '3'}, {'2', '4'}, {'2', '5'}, {'3', '5'}, {'3', '7'}, {'8', '3'}, {'4', '5'}, {'6', '5'}, {'8', '7'}, **{'9', '5'}**]

https://insideaiml.com/article-details/Python---Graphs-928

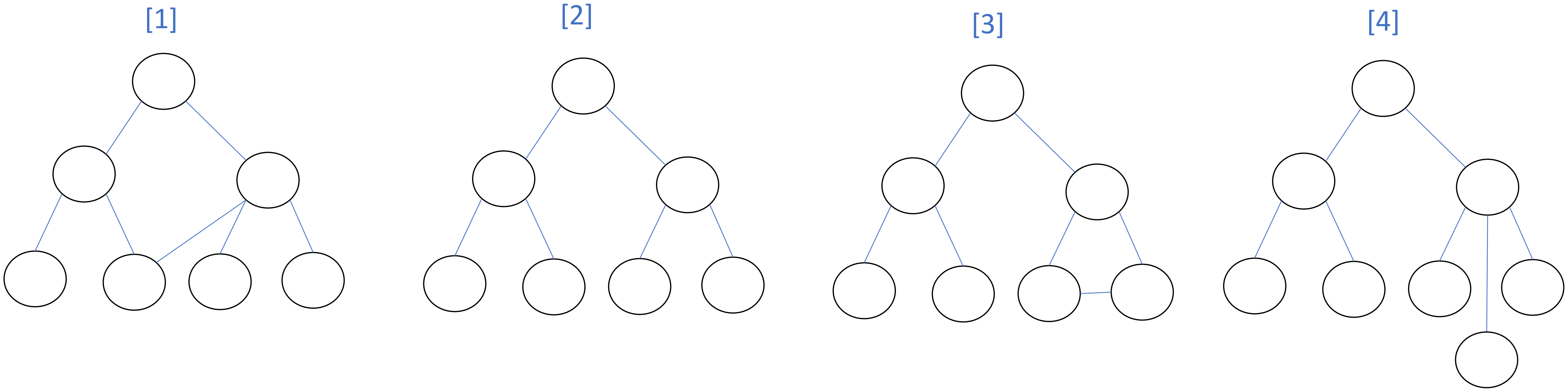
Advantages of trees and graphs

- Search complexity:
  - Array or linked list: they are linear structures and the time required to search a “linear” list is proportional to the size of the data set.
  - Trees: fast search (*O*(log n) comparisons to find a particular node)

- Representation:
  - Linked lists: a node could at most have two pointers (one to its next and one to its previous node)
  - Graphs: a node could have more than two pointers.

Poll

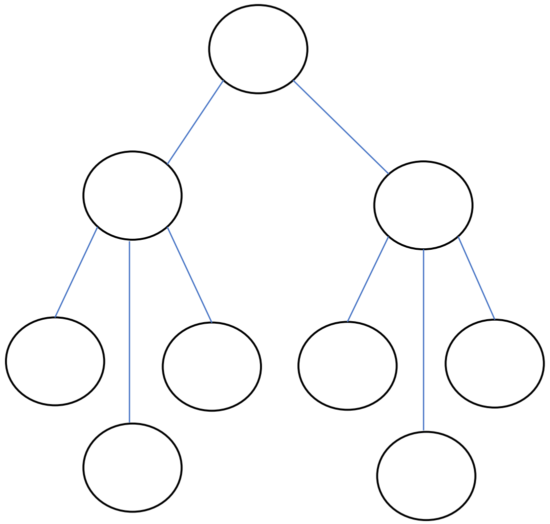
Which of the following graphs are/is a tree?



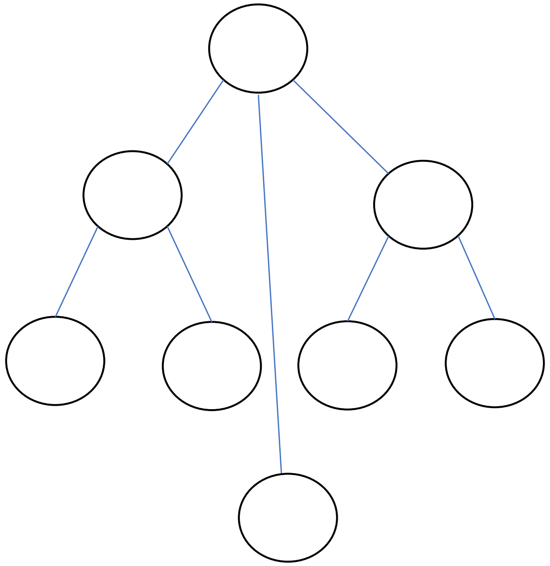
Poll

Which of the following graphs are/is a binary tree?

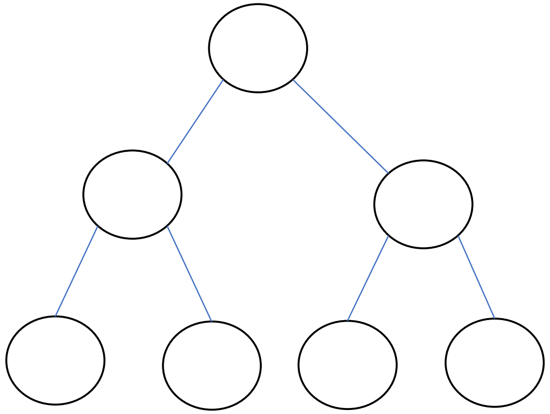
[1]



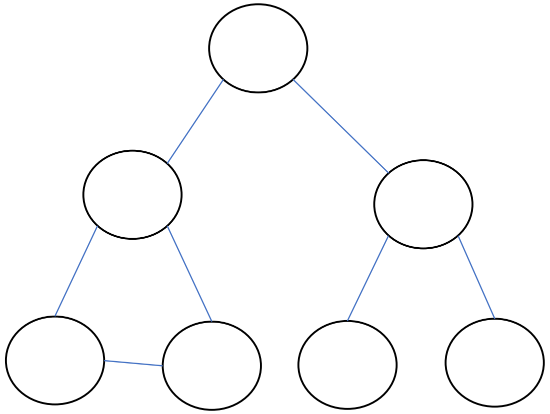
[2]



[3]

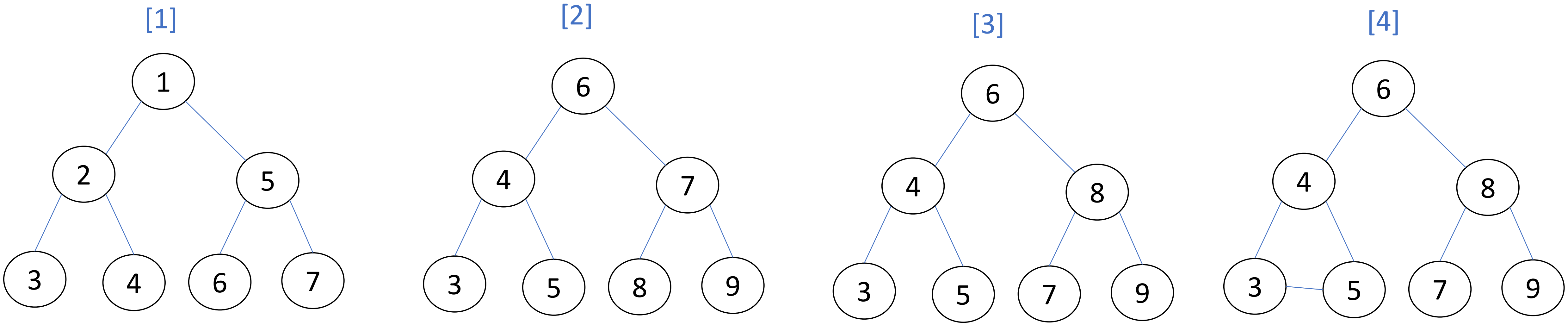


[4]



Poll

Which of the following graphs are/is a binary search tree?



Invariants

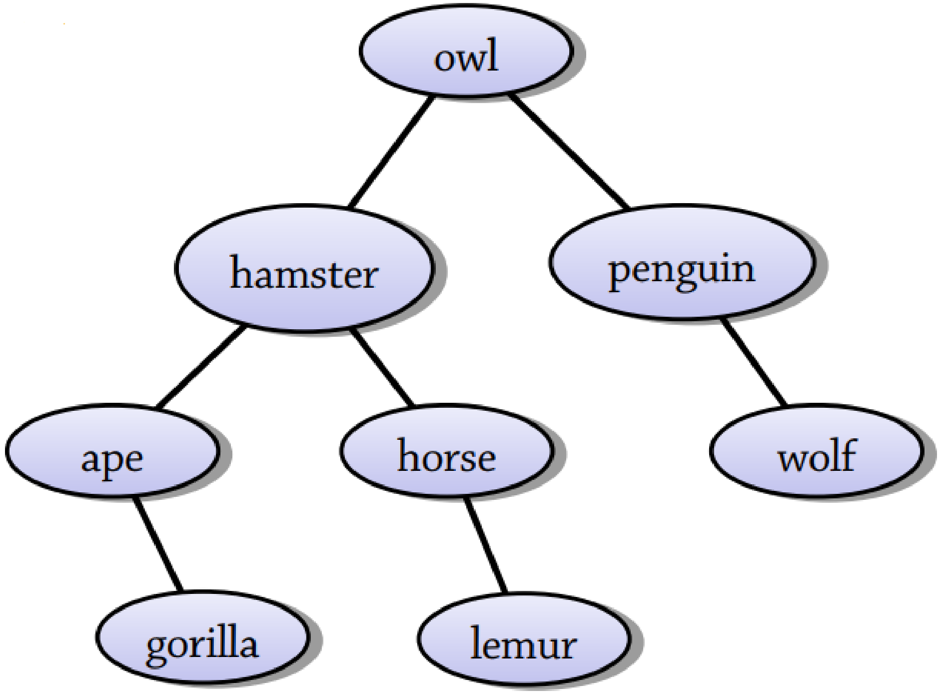
Some slides adapted from

Robert Sedgewick, Kevin Wayne, Peter Ljunglöf, and Nick Smallbone

Back to binary search trees

A binary search tree (BST) is a binary tree where:

- Each node has a key
- Each node's key is greater than all the keys in the left subtree, and less than all the keys in the right subtree



Invariants

- “unchanged by specified mathematical or physical operations or transformations” [Merriam-Webster dictionary]

- A set of properties or conditions that will hold before and after conducting each step of an operation/algorithm.

Invariants

The property

“Each node’s key is greater than all the keys in the left subtree, and less than all the keys in the right subtree ”

is an example of a data structure invariant

- A property that the data structure designer wants to always hold

- The invariant affects the whole design!

- In search (get), we rely on the invariant holding in order to find the value efficiently

- When modifying the data structure, we must take care to make the invariant still holds afterwards – this is called maintaining the invariant

  e.g., in insertion and deletion

The goal: find an invariant which is useful but also easy to maintain!

Checking the invariants

What happens if we fail to maintain the invariant?

- For example, inserting something in the wrong place

Answer: at first, maybe nothing! But later operations may fail

- A later call to get/max/floor/... may return the wrong answer!
- Or a call to put may fail to find an existing key... and end up with the same key appearing twice in the BST!

These kind of bugs are a nightmare to track down!

Solution: check the invariant

Checking the invariants

Define a method

```
boolean invariant()
```

that returns *true* if the invariant holds

```
boolean invariant() {
    return isBST(root);
}
```

Then, in the implementation of every operation, do

```
assert invariant() : “Invariant failed”;
```

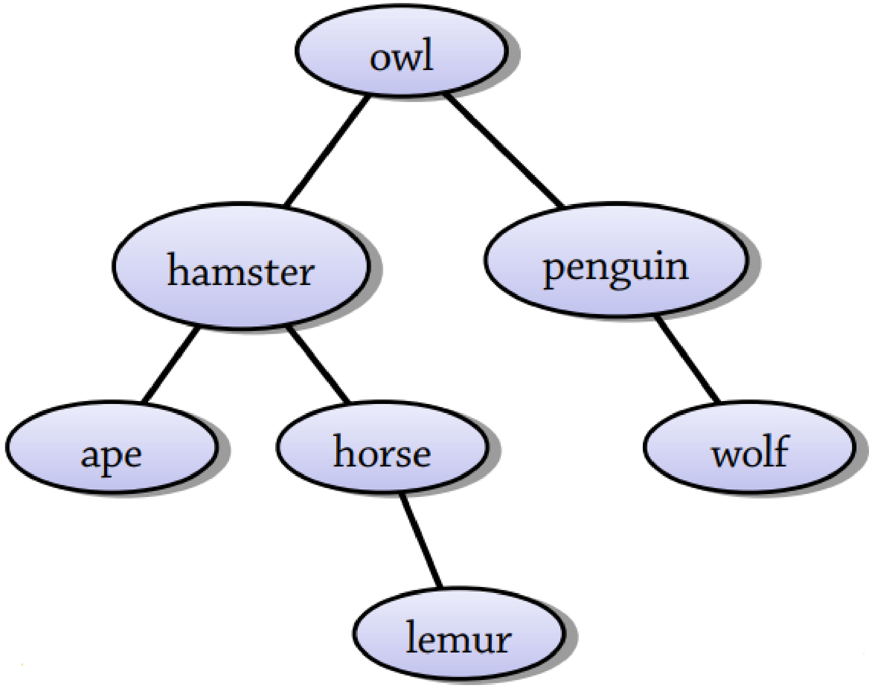
This will *throw an exception* if the invariant doesn't hold!

This will find lots and lots of bugs!

Rank and select

Let's try to add two more operations on BSTs:

- `int rank(Key key):` returns the number of items in the BST less than key
  - e.g. `rank(horse) = 2`
- `Key select(int n):` returns the nth-smallest item in the BST, counting from 0
  - e.g. `select(2) = horse`
  - We count from 0 so that select and rank are inverses



How can we implement these?
What invariant can we add?

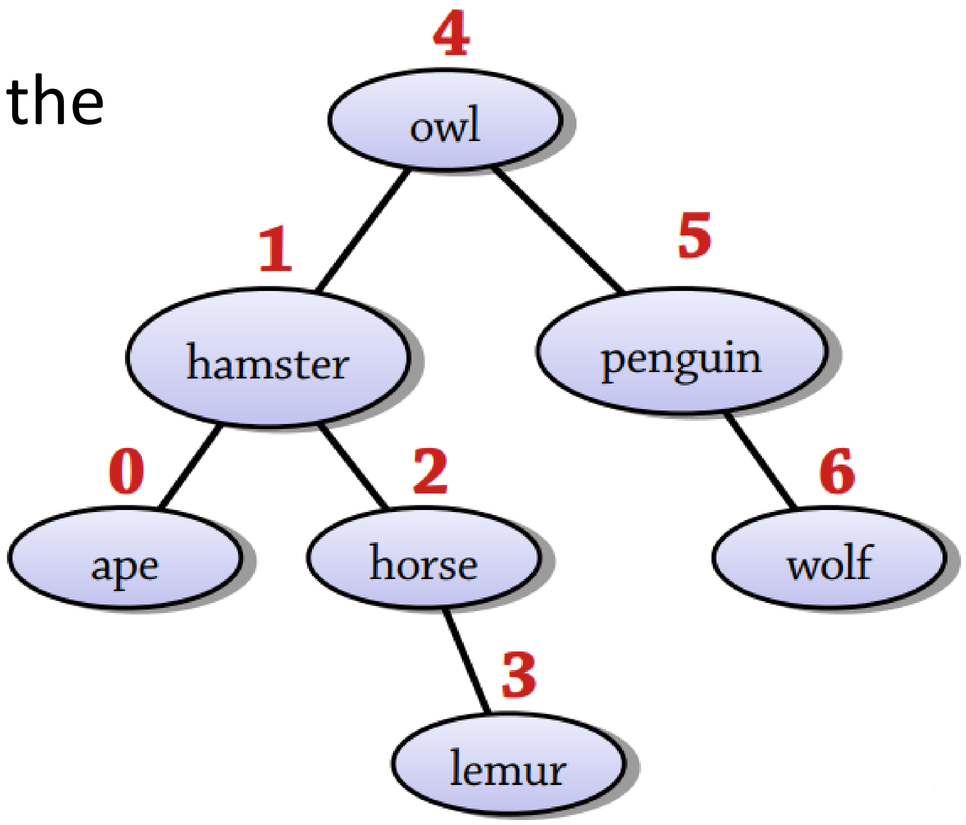
A first attempt

Let's store the rank as an extra field in each node

To implement `rank(key)`, we'll just find the correct node and then return its rank field.

Is this a good idea?
What's the problem?

This is a bad idea.



A first attempt

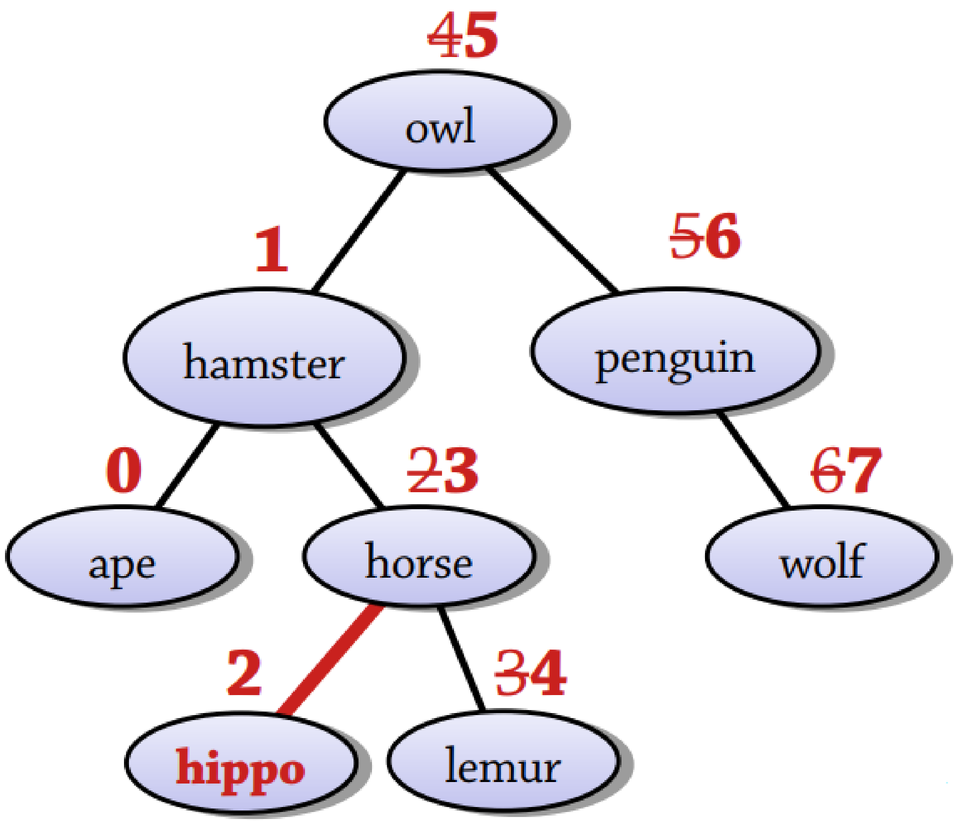
This variant is expensive to maintain

Let’s store the rank as an extra field in each node

To implement rank(key), we’ll just find the correct node and then return its rank field.

This is a bad idea.
What’s the problem?

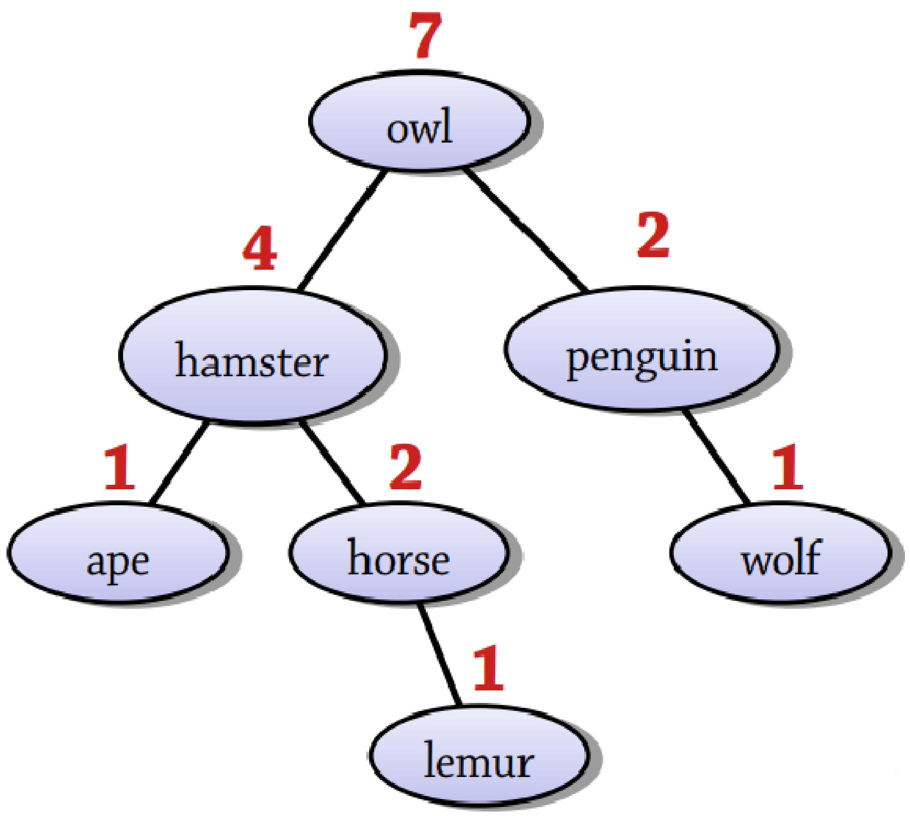
Whenever we modify the BFS, we’ll need to update the rank fields of lots of nodes (linear time)



A better answer

Make each node record the *size of its subtrees.*
Observation: *rank of root = size of left subtree.*
This leads to a recursive algorithm for rank!

```
int rank(Key key, Node x) {
        if (x == null) return 0;
        int cmp = key.compareTo(x.key);
        if (cmp < 0)
                return rank(key, x.left);
        else if (cmp > 0)
                return 1 + size(x.left) +
                        rank(key, x.right);
        else return size(x.left);
}
```

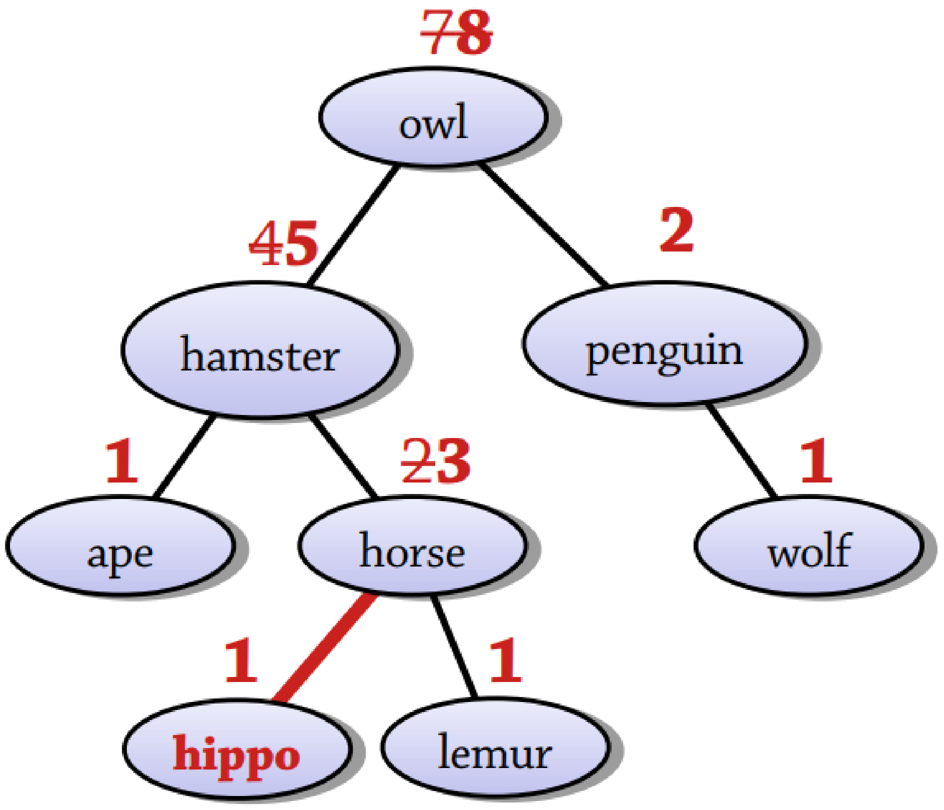


A better answer

What's more, when we insert a new value, we only need to update the size fields of the new node and its ancestors

- Number of nodes changed = height of tree
- Logarithmic, if BST is balanced!
- Invariant is *cheap to maintain*

(P.S. this invariant also supports select)



Invariants are important

Once we chose the invariant, we were forced to implement the operations a certain way:

- Given the BST invariant, there's only one reasonable way to implement search
- Also, only one way to implement insertion so as to preserve the invariant
- Given the invariant about labelling nodes with their size, there's only one reasonable way to implement rank
- And insertion/deletion must update the size field so as to preserve the invariant

The main creative step was choosing the invariant!

Invariants

When designing a data structure your first question should be:

What is the invariant?

- **How can I use it to efficiently compute stuff?**
- **How can I maintain it when updating the data structure?**

Because the invariant often expresses the main idea of the data structure!

A good invariant adds some *extra structure* that:

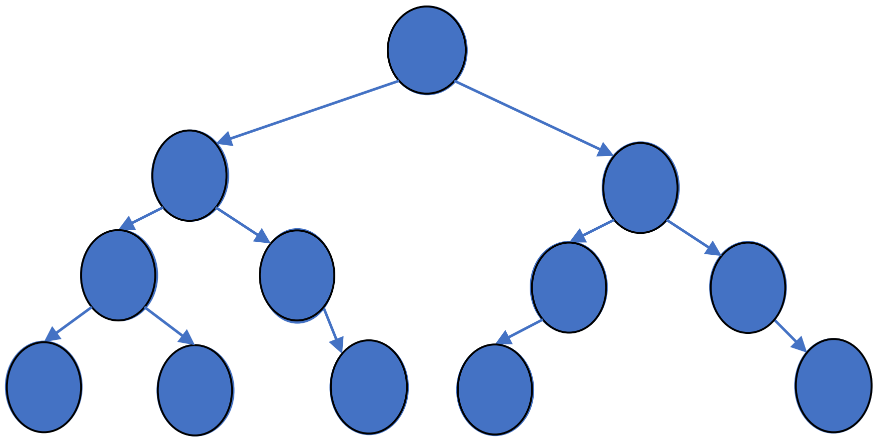
- makes it easy to *get* at the data (the invariant is useful)
- but doesn't make it too hard to maintain the invariant when *updating* the data

Graph Traversal

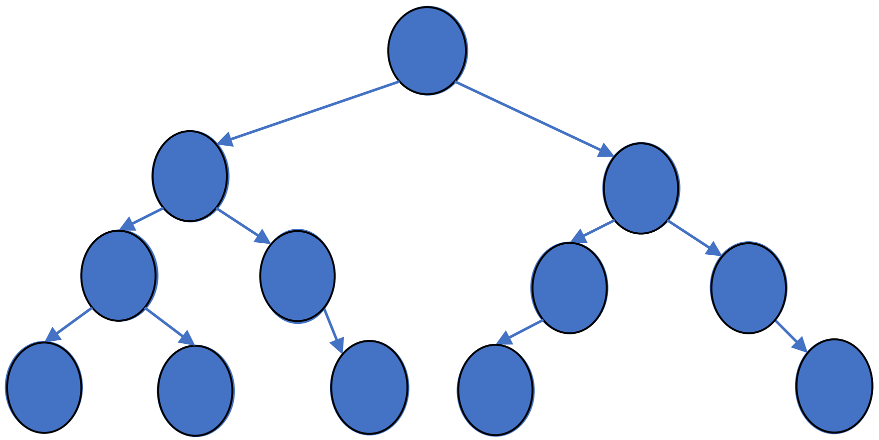
Walking a graph <-> Visiting nodes of a graph <-> Traversing nodes of a graph in a particular order

- Breadth-First-Search (BFS)
  - Explore all the nodes at one depth level before exploring the nodes of the next depth level
- Depth-First-Search (DPS)
  - Explore all the nodes in one subtree before exploring a sibling subtree

- BFS



- DFS



Breadth-First-Search (BFS)

Breadth First Search (BFS):

BFS(*s*): Find a BFS tree rooted at s, which includes all nodes reachable from node s.

```
Create a Boolean array Discovered[1…n], Set Discovered[s] = true
and Discovered[v] = false for all other v.                      O(n)

Create an empty FIFO queue Q, add node s to Q.                  O(1)

while Q is not empty                                            O(?)
    dequeue a node u from Q                                     O(1)
    for each node v adjacent to node u                          O(?)
        if Discovered[v] is false then                          O(1)
            add node v to Q, set Discovered[v] to true          O(1)
        endif
    endfor
endwhile
```

Question – Poll

What is the time complexity of BFS?

V: number of vertices, E: number of edges

1) *O*(E)
2) *O*(V)
3) *O*(V+E)
4) *It depends which representation of the graph is used!*

Breadth-First-Search (BFS)

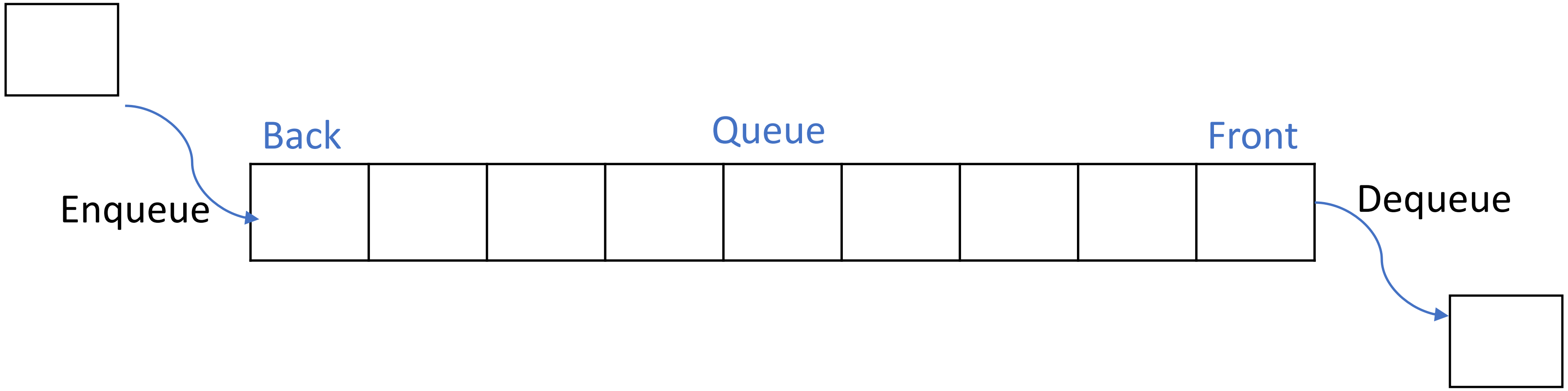
Analysis:

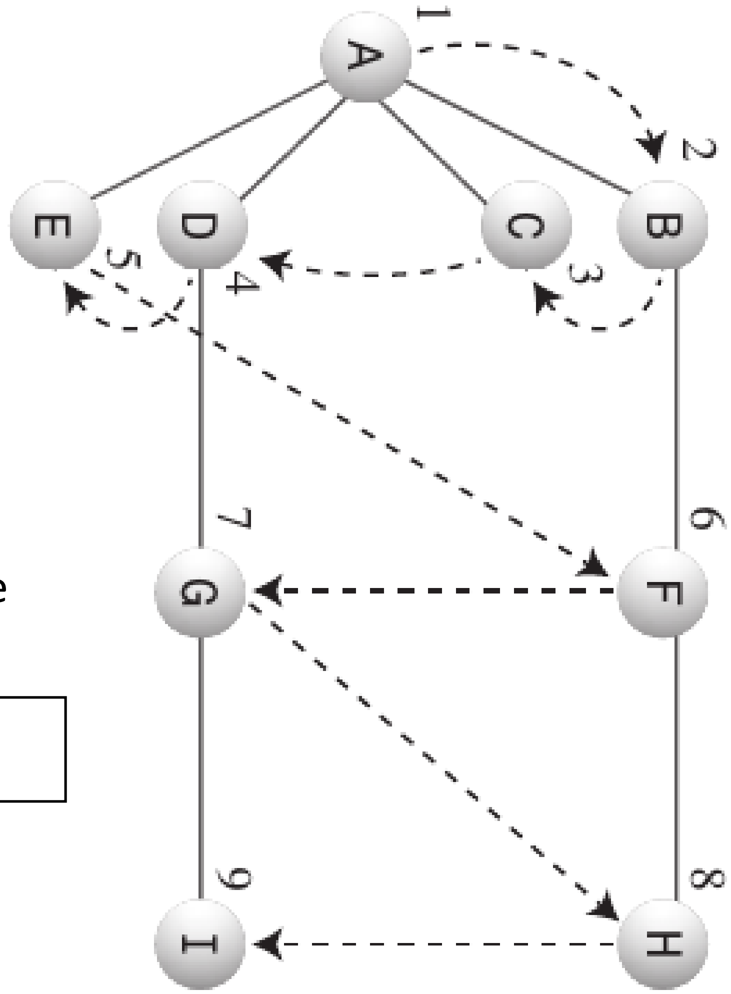
- A node *u* enters Q at most once, and the for loop needs nodes adjacent to every such u
- Graph representation will effect the analysis
- Finding all *v* adjacent to *u*:
  - **Adjacency Matrix:**
    - ➢ we have to check all matrix entries in u's row: O(n)
    - ➢ total time required to process all rows of the Matrix: $O(n^2)$
  - **Adjacency List:**
    - ➢ when we consider node u, there are deg(u) incident edges (u, v)
    - ➢ total time processing all the edges is $\Sigma_{u \in V} \deg(u) = 2m \Rightarrow O(m)$
    - ➢ setup time for the array Discovered is O(n), $\Rightarrow O(m + n)$
    - ➢ m is at least n-1 for connected graph, m dominates $\Rightarrow O(m)$

each edge (u, v) is counted exactly twice in sum: once in deg(u) and once in deg(v)

Breadth-First-Search (BFS)

	A	B	C	D	E	F	G	H	I
Discovered	false	false	false	false	false	false	false	false	false





http://i.stack.imgur.com/TjhfH.png

BFS Tree:

Create an empty tree T

Add edge *(u, v)* to the tree *T,* when v is discovered the first time

Breadth-First-Search (BFS)

A	B	C	D	E	F	G	H	I
True	True	True	True	True	True	True	True	True

Enqueue A → A

Enqueue B, C, D, E → E D C B ~~A~~ | Dequeue-> A → : A

Enqueue F → F E D C ~~B~~ ~~A~~ | Dequeue-> B → : A, B

F E D ~~C~~ ~~B~~ ~~A~~ | Dequeue-> C → : A, B, C

Enqueue G → G F E ~~D~~ ~~C~~ ~~B~~ ~~A~~ | Dequeue-> D → : A, B, C, D

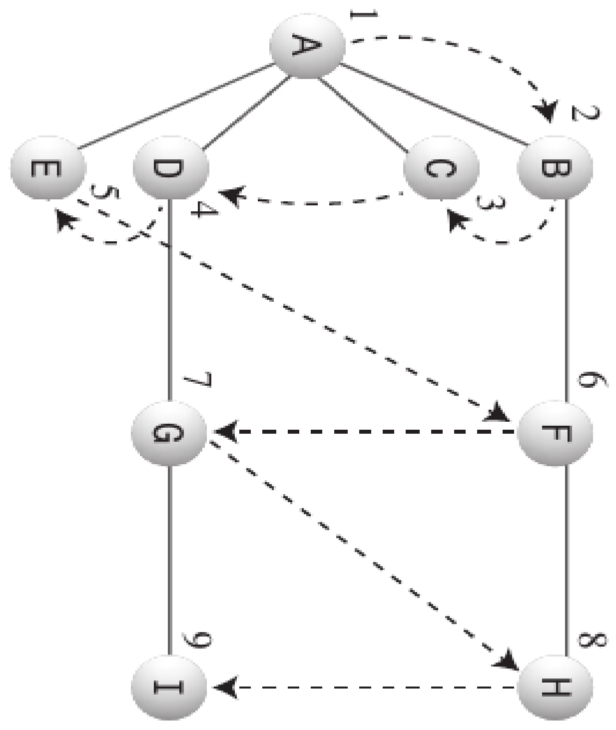
G F ~~E~~ ~~D~~ ~~C~~ ~~B~~ ~~A~~ | Dequeue-> E → : A, B, C, D,E

Enqueue H → H G ~~F~~ ~~E~~ ~~D~~ ~~C~~ ~~B~~ ~~A~~ | Dequeue-> F → : A, B, C, D, E, F

Enqueue I → I H ~~G~~ ~~F~~ ~~E~~ ~~D~~ ~~C~~ ~~B~~ ~~A~~ | Dequeue-> G → : A, B, C, D, E, F, G

I ~~H~~ ~~G~~ ~~F~~ ~~E~~ ~~D~~ ~~C~~ ~~B~~ ~~A~~ | Dequeue-> H → : A, B, C, D, E, F, G, H

~~I~~ ~~H~~ ~~G~~ ~~F~~ ~~E~~ ~~D~~ ~~C~~ ~~B~~ ~~A~~ | Dequeue-> I → : A, B, C, D, E, F, G, H, I

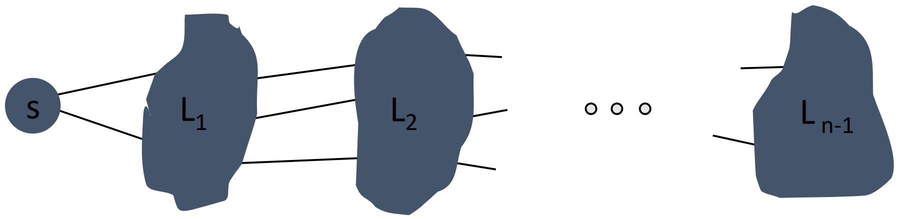


http://i.stack.imgur.com/TjhfH.png

Breadth First Search: Properties

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.

BFS algorithm partitions the nodes into layers:



- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

➢ Implementation using Queue processes the nodes exactly layer by layer

➢ explores in order of distance from s.

Question- Poll

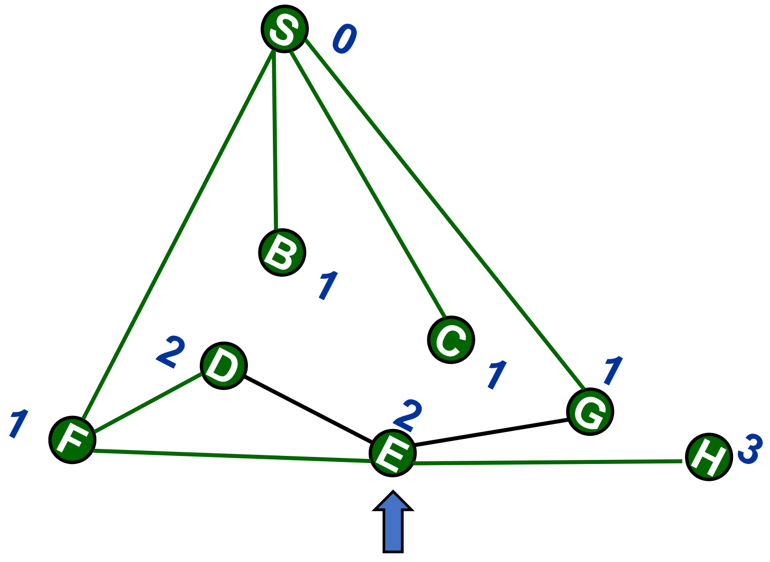
Suppose we have a tree in which each edge has length 1. In the implementation of Breadth-First-Search using queues, what is the maximum distance between two nodes in the queue?

1) 0
2) At most 1
3) It depends on the shape of the tree
4) Can be anything

Breadth First Search: Properties

Property. Let T be a BFS tree of G = (V, E),
nodes u, v belong to T, and
let (u, v) be an edge of G.
Then the level of u and v differ by at most 1.

(Edges discarded by BFS are those which connect
nodes of the same layer e.g., DE, or
nodes from adjacent layers e.g., EG)



Let u, v belong to layers L_i and L_j respectively.

Suppose $i < j - 1$. (Negate the conclusion)

- When BFS examines the edges incident to u, since u belongs to layer L_i, the only nodes discovered from u belong to layers L_{i+1} and earlier;
- hence, if v is a neighbor of u, then it should have been discovered by this point at the latest, and
- should belong to layer L_{i+1} or earlier (a contradiction)

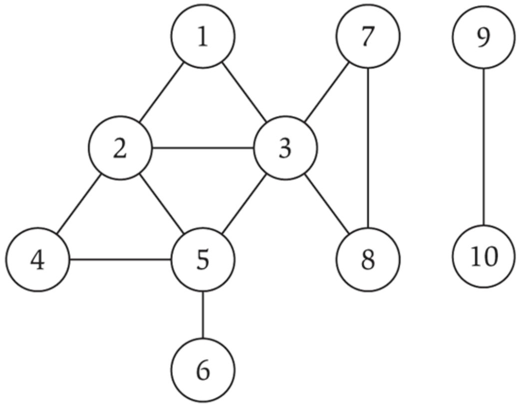
Breadth First Search: Properties

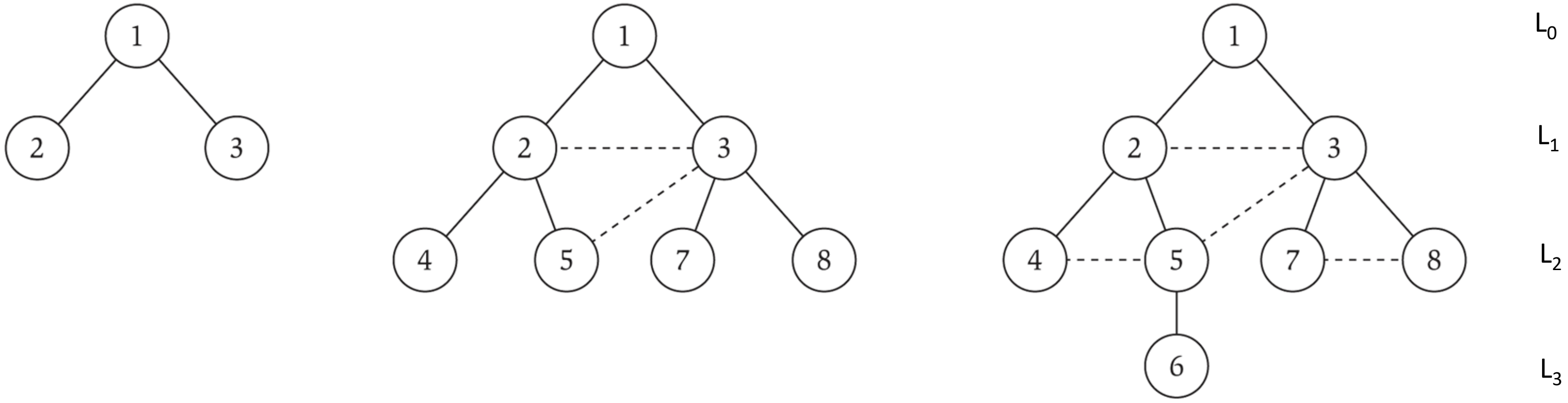
- What follows:

➢ For each i, L_i consists of all nodes at distance exactly i from s.

➢ There is a path from s to t if t appears in some layer.

➢ Moreover: s-t is a shortest path.





(a) (b) (c)

Depth First Search (DFS)

Depth First Search (DFS):

Create a Boolean array Explored[1…*n*], initialized to false for all.

DFS(u)

set Explored[*u*] to true

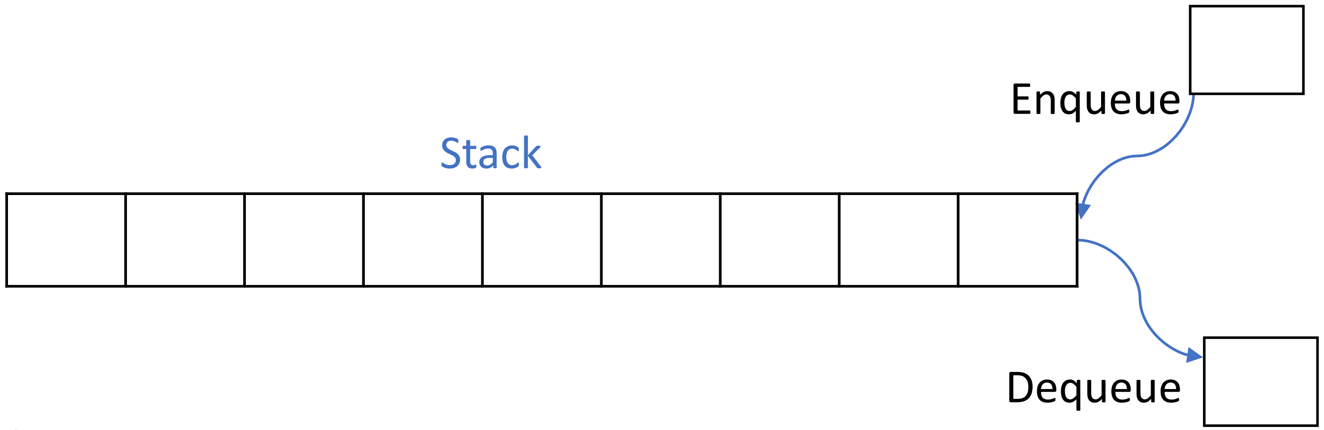
for each node *v* adjacent to node *u*

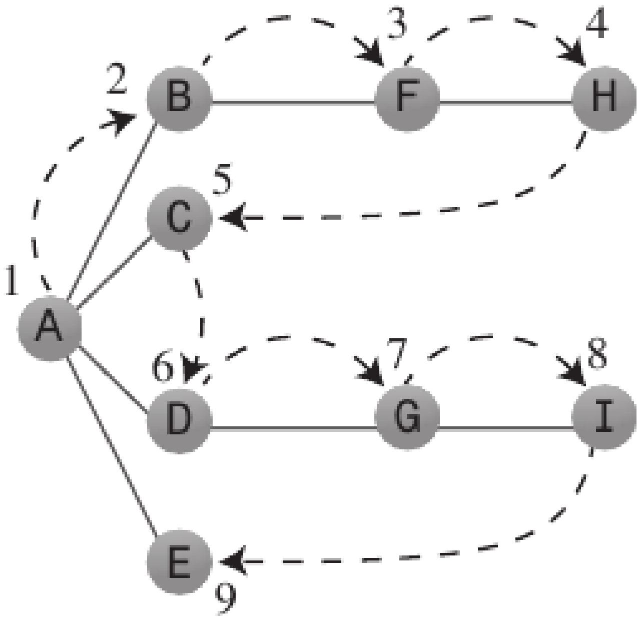
```
        if Explored[v] is false then
                DFS(v)
        endif
endfor
```

- **Call DFS(s)**
  - each recursive call is done only after termination of the previous call, this gives the desired depth first behavior.

Depth First Search (DFS)

	A	B	C	D	E	F	G	H	I
Discovered	false	false	false	false	false	false	false	false	false





Reference: http://i.stack.imgur.com/gh0T1.png

DFS tree:

Take an array parent,

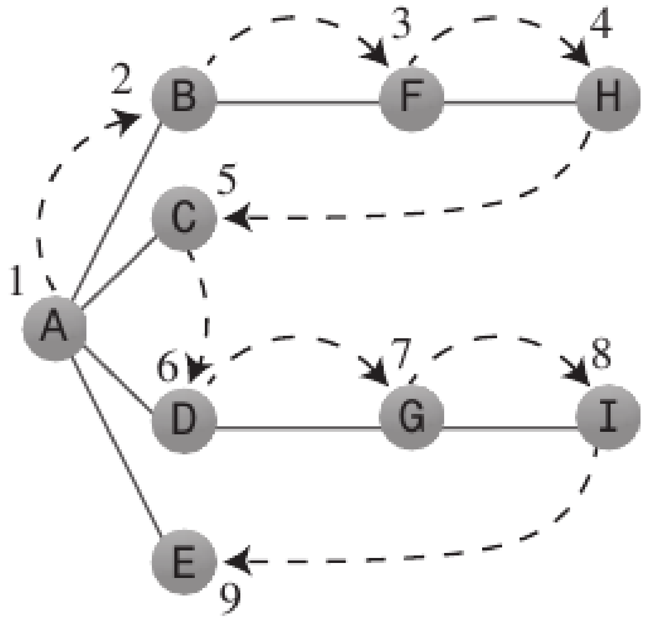
set parent[*v*]= *u* when calling DFS(v) due to edge *(u, v)*.

When setting u *(u ≠ s)* as Explored,

add the edge *(u*, parent[*u*]*)* to the tree.

Depth First Search (DFS) using stack

Push A	A				Visited Nodes: **A**
Push B	B	A			: **A ,B**
Push F	F	B	A		: **A, B, F**
Push H	H	F	B	A	: **A, B, F, H**
Pop H	~~H~~	F	B	A	: A, B, F, H
Pop F	~~H~~	~~F~~	B	A	: A, B, F, H
Pop B	~~H~~	~~F~~	~~B~~	A	: A, B, F, H
Push C	C	A			: **A, B, F, H, C**
Pop C	~~C~~	A			: A, B, F, H, C
Push D	D	A			: **A, B, F, H, C, D**
Push G	G	D	A		: **A, B, F, H, C, D, G**
Push I	I	G	D	A	: **A, B, F, H, C, D, G, I**
Pop I, Pop G, Pop D	~~I~~	~~G~~	~~D~~	A	: A, B, F, H, C, D, G, I
Push E	E	A			: **A, B, F, H, C, D, G, I, E**
Pop E, Pop A	~~E~~	~~A~~			: A, B, F, H, C, D, G, I, E



Reference: http://i.stack.imgur.com/gh0T1.png

Question – Poll

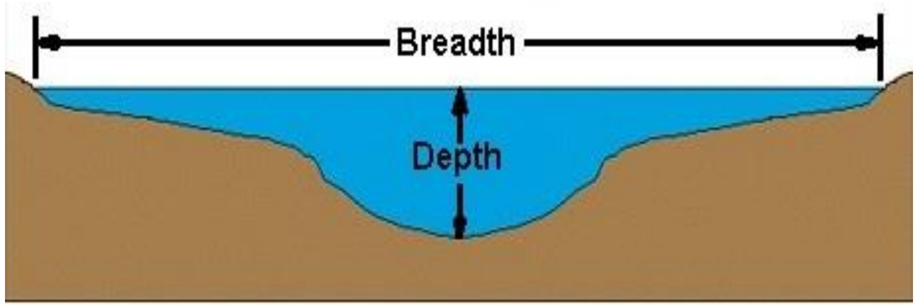
Suppose we have a tree in which each edge has length 1. In the implementation of Depth-First-Search using stacks, what is the maximum distance between two nodes in the stack?

1) 0
2) At most 1
3) It depends on the shape of the tree
4) Can be anything

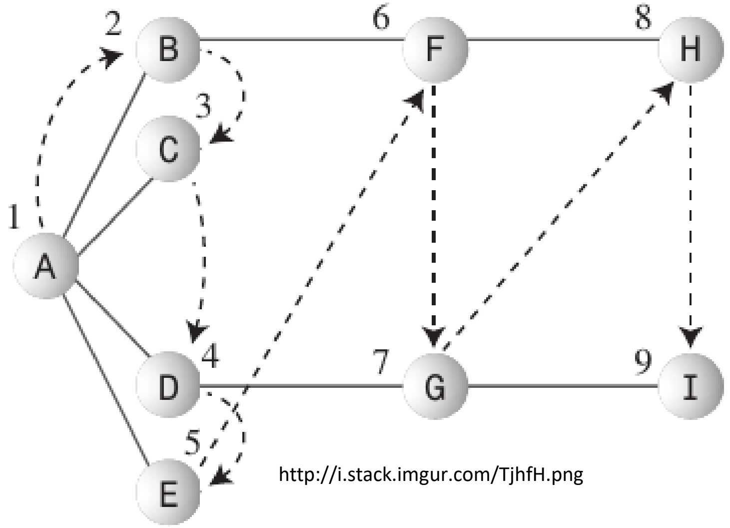
BFS vs. DFS

BFS: Put unvisited vertices on a queue.

- Examines vertices in increasing distance from s.
- Using adjacency list requires O(m).



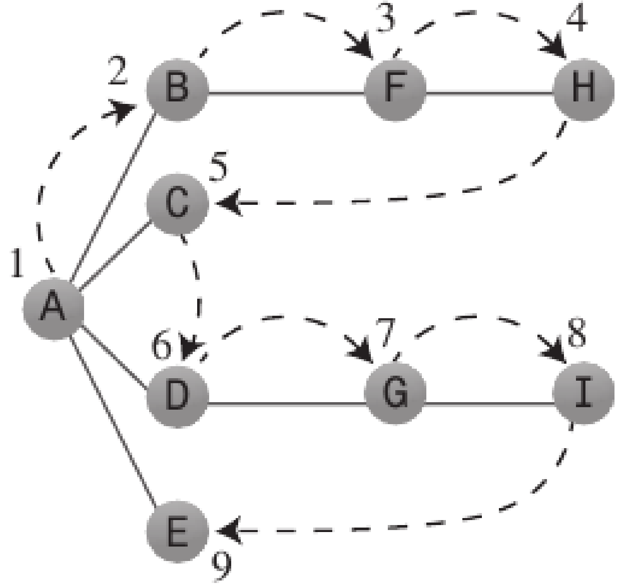
http://i.stack.imgur.com/QtYo8.jpg



http://i.stack.imgur.com/TjhfH.png

DFS: Put unvisited vertices on a stack.

- tries to explore as deeply as possible
  - *Mimics maze exploration.*
- O(m) due to similar reasoning.



http://i.stack.imgur.com/gh0T1.png