A-A-WEEK-8

- ## Assessment Structure
-
  - 1 hour, 40 minutes
- - Mixture of multiple choice (15-20), free response (1-3) and VSCode (1-3) problems, each with multiple specs.
- - Free response just requires enough detail to answer the question, 1-3 sentences. As long as you are able to explain the concept and answer all aspects that it asks, you are good.
- - Coding problems will have specs to run (`npm test`) and check your work against
- - Standard assessment procedures
- - You will be in an individual breakout room
- - Use a single monitor and share your screen
- - Only have open those resources needed to complete the assessment:
- - Zoom
- - VSCode
- - Browser with AAO and Progress Tracker (to ask questions)
- - Approved Resources for this assessment:
- - MDN: https:>developer.mozilla.org/en-US/docs/Web/JavaScript
-

1. the study guide - Anki cards & Code Implementations
2. BFS & DFS for a graph
3. BFS & DFS for a binary search tree
4. know the difference between BF(S/T) and DF(S/T) - search vs traversal

IP SUITE

IDENTIFY THE CORRECT FIELDS OF AN IPV6 HEADER. COMPARED TO IPV4'S 14 (YIKES!) HEADER FIELDS, WE NOW ONLY USE 8 FIELDS. THEY ARE ALL LISTED HERE, BUT THE MAIN THINGS TO BE FAMILIAR WITH ARE THE VERSION TO INDICATE IPV4 VS IPV6, AS WELL AS THE SOURCE AND DESTINATION IP ADDRESS FIELDS:

VERSION: JUST LIKE IPV4, THE FIRST FOUR BITS OF OUR PACKET HEADER REPRESENT OUR VERSION NUMBER, WHICH IS 0110, REPRESENTING OUR DECIMAL 6.

TRAFFIC CLASS: IDENTIFIES THE TYPE OF PACKET DATA

FLOW LABEL: ADDS PACKET SEQUENCING TO IP, BUT IS EXPERIMENTAL

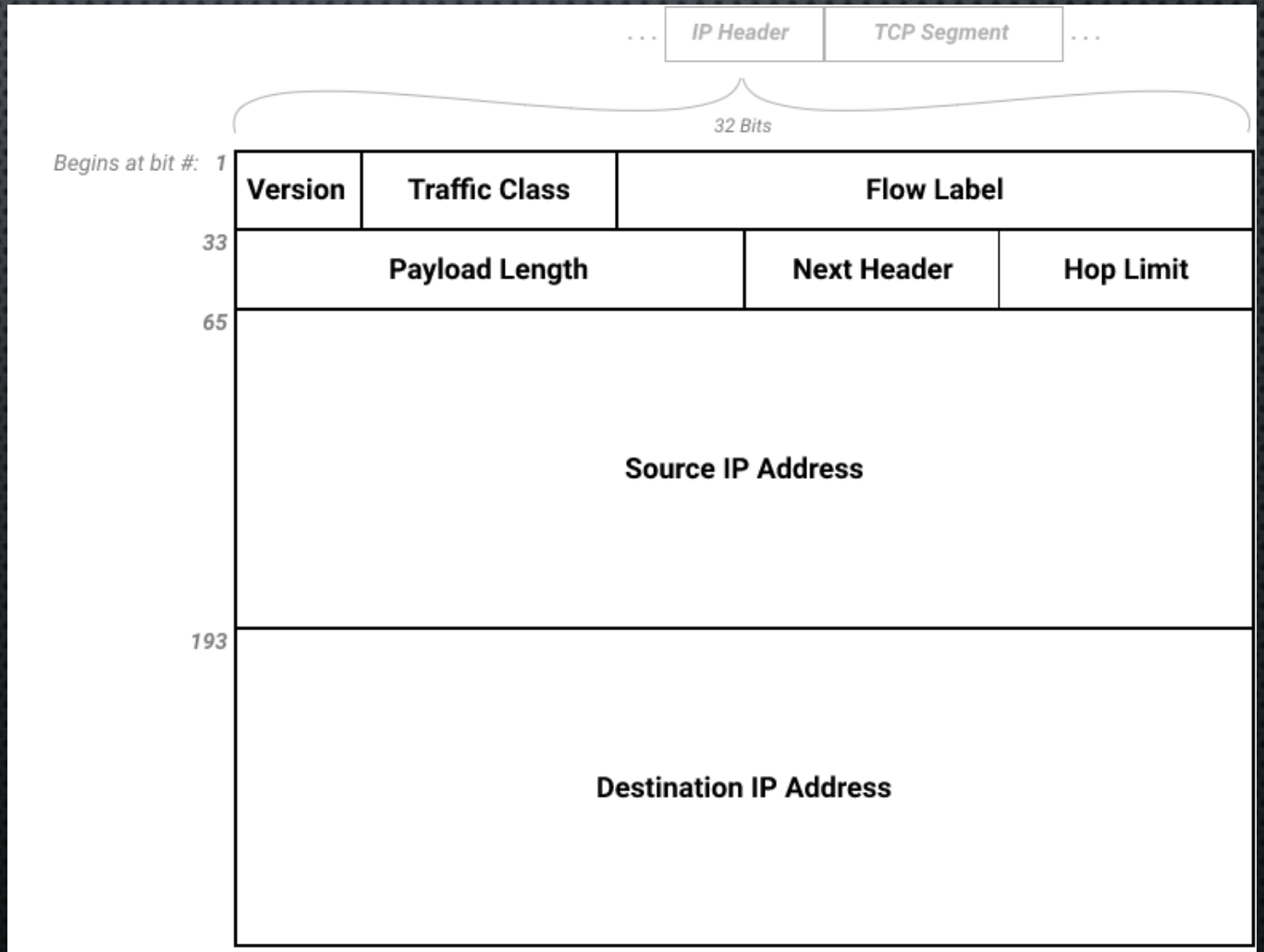
PAYLOAD LENGTH: SPECIFIIES THE SIZE OF THIS PACKET

NEXT HEADER: TYPICALLY IDENTIFIES THE TRANSPORT LAYER PROTOCOL (TCP, UDP, ETC.). IT CAN ALSO INDICATE AN EXTENSION HEADER IS PRESENT, WHERE EXTRA OPTIONS FOR THE PACKET ARE SPECIFIED (SIMILAR CONCEPT TO THE OPTIONS FIELD OF IPV4, BUT A DIFFERENT, CHAINABLE IMPLEMENTATION)

HOP LIMIT: DECREMENTED BY ONE EACH TIME THIS PACKET PASSES THROUGH AN INTERMEDIARY. PREVENTS THE PACKET FROM BEING PASSED AROUND FOREVER.

SOURCE IP ADDRESS

DESTINATION IP ADDRESS

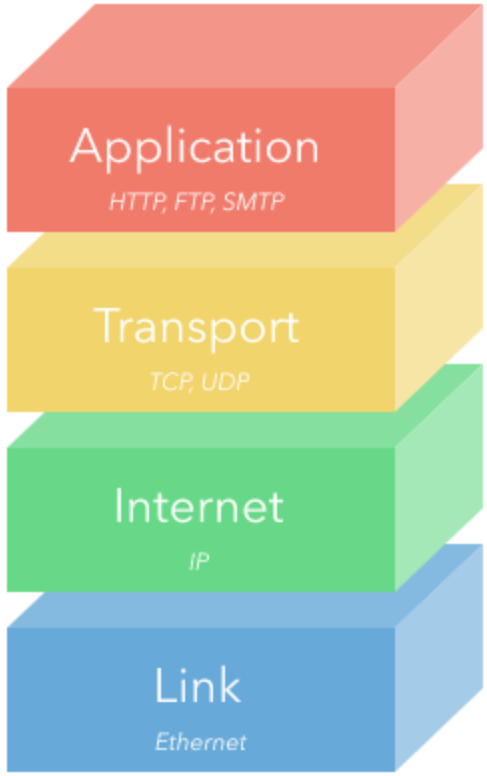


IP Address (Internet Protocol Address)	Support end-to-end connections without a central system
IP Address (Internet Protocol Address)	A number that uniquely identifies each computer or device connected to the Internet.
Is IP's packet switching concerned about packet ordering or lost packets?	No - TCP is responsible for ensuring the correct packet order, and for acknowledging the receipt of packets (to gracefully handle lost packets)
List the two needs for computer-to-computer communications provided by TCP	1. Reliable Data Transfer 2. Persistent and Consistent Connections
TCP stands for what	Transmission Control Protocol
Transport Layer	This IP Suite (TCP/IP Model) layer supports connectivity between clients and server

What is the bottom layer of the TCP/IP model?	The link or network access
What is the second layer of the TCP/IP model?	Internet
What is the third layer of the TCP/IP model?	Transport
What is top layer of the TCP/IP model?	Application

1. Describe the structure and function of network models from the perspective of a developer.

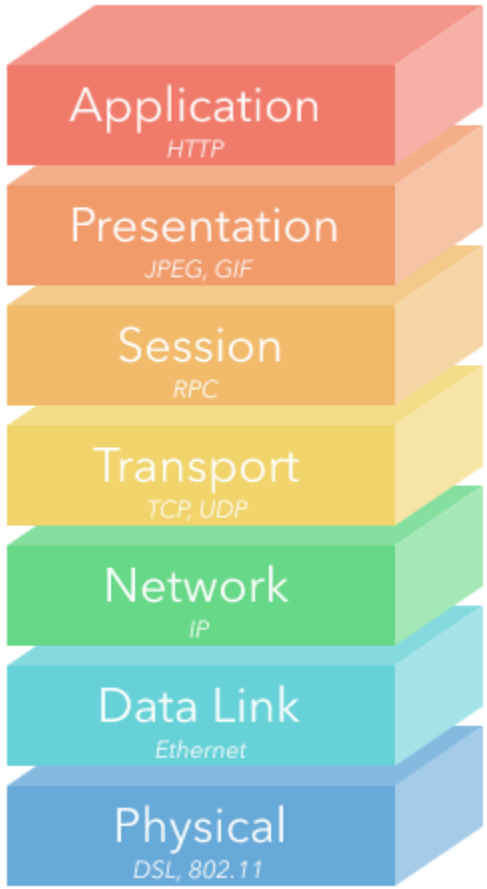
- Recognize a given model as the TCP/IP model



- Give a brief description of each layer (What is its major concern and an example)
  - Application: User-facing data, such as HTTP or FTP (file transfer)
  - Transport: Connectivity between clients and servers, such as TCP or UDP
  - Internet: Routing between separate networks, such as IP
  - Link: Low-level communication between local resources on a network, such as Ethernet

REFERENCE
NETWORK OSI

- Recognize a given model as the OSI model



- Give a brief description of each layer (What is its major concern and an example)
- Application (Layer 7)
  - Example: HTTP
  - Information used by client-side software
- Presentation (Layer 6)
  - Example: JPEG, GIF
  - Data gets translated into a presentable format
  - Often called the syntax layer since it translates machine-readable syntax into human-readable syntax
- Session (Layer 5)
  - Example: RPC (Remote Procedure Call)
  - Authentication and data continuity
  - Authorize actions, reestablish session from dropped connections
- Transport (Layer 4)
  - Example: TCP, UDP
  - Mirrors TCP/IP's Transport Layer
  - Focused on data integrity and connectivity
- Network (Layer 3)
  - Example: IP
  - Mirrors TCP/IP's Internet Layer
  - Manages connections between different remote networks
- Data Link (Layer 2)
  - Example: Ethernet
  - Connections between one network interface to another
  - Primarily used by machines in a local network (ie targeting different MAC addresses)
- Physical (Layer 1)
  - Example: DSL, 802.11 (Wi-Fi)
  - Translating from electrical signals to bits of data

NETWORK OSI

Application layer (OSI)	Layer 7 of the OSI model, it provides application services to a network. An important, and an often-misunderstood concept, is that end-user applications do not reside at the application layer. Instead, the application layer supports services used by end-user applications. Another function of the application layer is advertising available services.
Application layer (OSI)	This OSI layer includes information used by client side software
Application, Presentation, Session	Which three layers of the OSI model are comparable in function to the application layer of the TCP/IP model? (Choose three.)
Data Link	Which layer in the OSI model do MAC addresses and switches use?
Data Link	This layer in the OSI model is responsible for dealing with connections directly from one machine's network interface to another machine's network interface

Data Link	This OSI layer deals with connections directly between two machine's network interfaces. This layer is mostly used by machines in a local network (i.e. Ethernet)
Network Layer (OSI)	This OSI Layer manages connections between remote networks (i.e. IP)
Network Layer (OSI)	Logical addressing, routing, and path determination.
Physical Layer (OSI)	Defines the electrical, optical, cabling, connectors, and procedural details required for transmitting bits, represented as some form of energy passing over a physical medium.
Physical Layer (OSI)	This OSI layer is responsible for managing low-level protocols like WiFi and DSL

Presentation Layer (OSI)	Defines the format and organization of data. Includes encryption.
Presentation Layer (OSI)	-performs 3 main functions: -data formatting: converts data to/from standardized formats -data compression/expansion: reduces large amounts of data into smaller file sizes -data encryption/decryption:makes data unreadable w/o the correct key (apps don't understand encryption, so this layer does it) -responsible for presenting information to the application layer in the manner expected
Session Layer (OSI)	This OSI layer includes protocols responsible for authentication and data continuity (i.e. RPC)
Session Layer (OSI)	Provides the mechanism for managing the dialogue between end-user application processes.-NetBIOS
Transport Layer (OSI)	responsible for providing communication with the application by acknowledging and sequencing the packets to and from the application

Transport Layer (OSI)	The OSI layer that accepts data from the upper layers, and breaks it up into smaller units known as segments, passes them on to the lower layers, and ensures that all segments arrive correctly at the other end.
Transport Layer (OSI)	This OSI Layer utilizes transport protocols in order to provide data integrity and connectivity (TCP and UDP for example)

NETWORK DNS

CNAME record	A Canonical Name record within DNS, used to provide an alias for a domain name.
CNAME record	Which resource record defines an alias for a host name?
CNAME record	This DNS record type acts like an alias linking together resources by domain instead of IP.
MX record	mail exchange record
MX record	Records within DNS servers that are used by SMTP servers to determine where to send mail

MX record	Which record do you use to specify an organization's mail server.
NS record	A record that points to a name server
A record	Used to resolve the name in an NS record to its IP address.
SOA record	Start of Authority record. This record identifies the primary name server for the zone. The SOA record contains the host name of the server responsible for all DNS records within the namespace, as well as the basic properties of the domain.

NETWORK TC/IP

IP Address (Internet Protocol Address)	Support end-to-end connections without a central system
IP Address (Internet Protocol Address)	A number that uniquely identifies each computer or device connected to the Internet.
Is IP's packet switching concerned about packet ordering or lost packets?	No - TCP is responsible for ensuring the correct packet order, and for acknowledging the receipt of packets (to gracefully handle lost packets)
List the two needs for computer-to-computer communications provided by TCP	1. Reliable Data Transfer 2. Persistent and Consistent Connections
TCP stands for what	Transmission Control Protocol

Transport Layer	Model) layer supports connectivity between clients and server
What is the bottom layer of the TCP/IP model?	The link or network access
What is the second layer of the TCP/IP model?	Internet
What is the third layer of the TCP/IP model?	Transport
What is top layer of the TCP/IP model?	Application

Vocab Binary Tree

Term	Definition
Tree	graph with no cycles
Binary tree	a tree where each node has at most 2 child nodes.
root	➢ the ultimate parent, ➢ the single node of a tree that can access every other node through edges; ➢ by definition the root will not have a parent
Internal node	a node that has children
Leaf	a node that does not have any children
path	a series of nodes that can be traveled through edges.

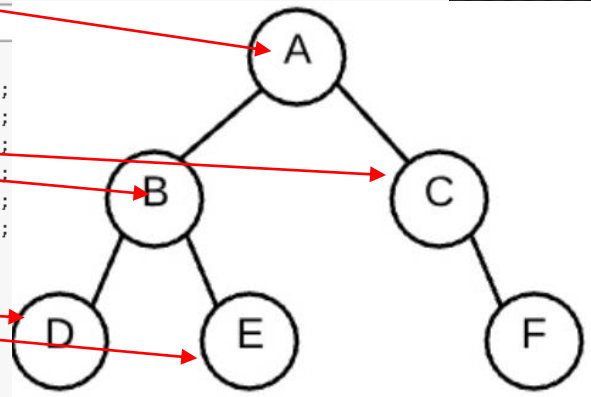
1. Explain and implement a Binary Tree

A Binary Tree is a Tree where nodes have at most 2 children, usually we represent these as 'left' and 'right'.

```
class TreeNode {
  constructor(val) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
}
```

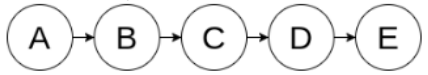
```
let a = new TreeNode('a');
let b = new TreeNode('b');
let c = new TreeNode('c');
let d = new TreeNode('d');
let e = new TreeNode('e');
let f = new TreeNode('f');

a.left = b;
a.right = c;
b.left = d;
b.right = e;
c.right = f;
```

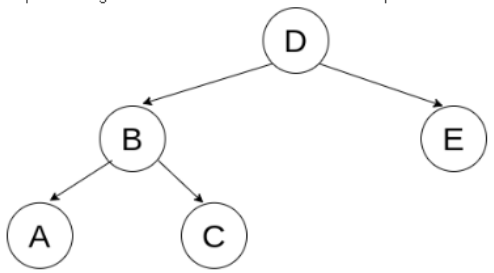


's a complete implementation of a binary search tree

- Two extreme implementations:



- Implementing in a chain results in max number of parents and min number of leaves: 4 parents, 1 leaf



- Implementing as a balanced tree results in min number of parents and max number of leaves: 2 parents, 3 leaves

- All that we need in order to implement a binary tree is a TreeNode class that can store a value and references to a left and right child.

```
class TreeNode {
  constructor(val) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
}
```

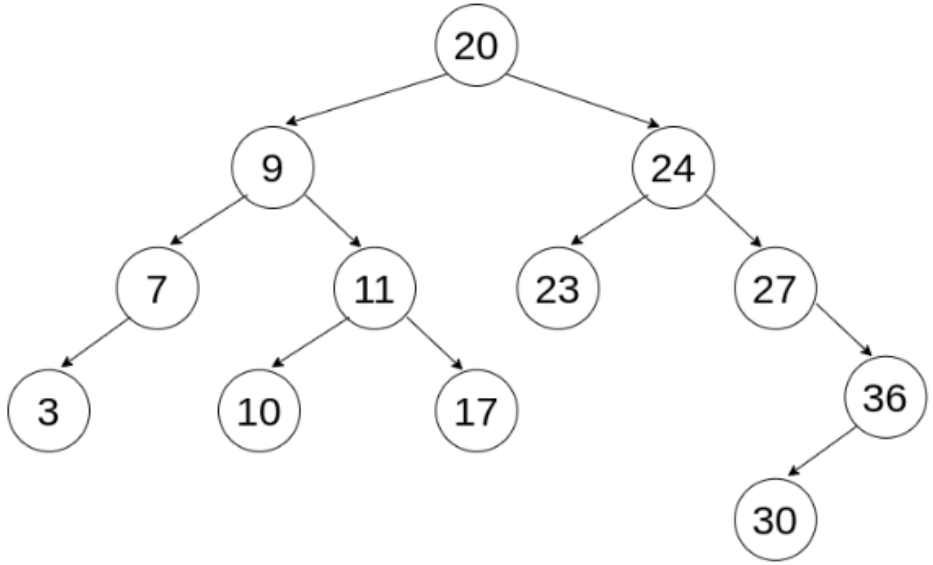
Binary Trees

1. Explain and implement a Binary Tree.

- A tree is a collection of nodes and edges between them.
- It cannot have any cycles, which are edges that form a loop between nodes.
- We also only consider rooted trees in computer science, which is a tree that has one root node that is able to access all other nodes.
- For a tree to be a binary tree, each node can have a maximum of two children.
- It's important to be able to identify and explain tree terminology as well. If given a tree, be able to point out each component.
  - root: The single node of a tree that can access every other node through edges.
  - parent node: A node that is connected to lower nodes in the tree. If a tree only has one node, it is not a parent node because there are no children.
  - child node: A node that is connected to a higher node in the tree. Every node except for the root is a child node of some parent.
  - sibling nodes: Nodes that have the same parent.
  - leaf node: A node that has no children (at the ends of the branches of the tree)
  - internal node: A non-leaf node (aka a parent)
  - path: A series of nodes that can be traveled through edges.
  - subtree: A smaller portion of the original tree. Any node that is not the root node is itself the root of a subtree.
- Know the basics of each term
  - A non-empty tree has to have a root.
  - A tree doesn't have any parent nodes if there are no children.
  - What's the min/max number of parent and leaf nodes for a tree with 5 nodes?

2. Identify the three types of tree traversals: pre-order, in-order, and post-order.

- Pre-order: Values are accessed as soon as the node is reached.
- In-order: Values are accessed after we have fully explored the left but before we explore the right branch.
- Post-order: Values are accessed after all of our children have been accessed.
- *Breadth First: The previous three are types of Depth First Traversals. Breadth first accesses values of nodes by level, left to right, top to bottom
- Given a tree, be able to determine the order of each traversal type:



  - Breadth First: 20, 9, 24, 7, 11, 23, 27, 3, 10, 17, 36, 30
  - Pre-order: 20, 9, 7, 3, 11, 10, 17, 24, 23, 27, 36, 30
  - In-order: 3, 7, 9, 10, 11, 17, 20, 23, 24, 27, 30, 36

3. Explain and implement a Binary Search Tree.

- A binary search tree is a binary tree with the added stipula
- Example of a BST with an insert method. You won't be as

```
class BST {
  constructor() {
      this.root = null;
  }

  insert(val, currentNode=this.root) {
    if(!this.root) {
      this.root = new TreeNode(val);
      return;
    }

    if (val < currentNode.val) {
      if (!currentNode.left) {
        currentNode.left = new TreeNode(val);
      } else {
        this.insert(val, currentNode.left);
      }
    } else {
      if (!currentNode.right) {
        currentNode.right = new TreeNode(val);
      } else {
        this.insert(val, currentNode.right);
      }
    }
  }
}
```

- Nodes
  - Similar to our linked list or tree implementations
  - Track the value and the neighbors array as instance variables on the node
  - We don't have a reference to the overall graph with this implementation

```
class GraphNode {
  constructor(val) {
    this.val = val;
    this.neighbors = [];
  }
}
```

Graphs

1. Explain and implement a Graph.

- A good place to start with explaining a graph is comparing to a tree:
  - A graph can:
    - Consist of any collection of nodes and edges (no limits on connections)
    - Have cycles
    - Have disconnected portions (a forest, with multiple trees, for example)
    - Be missing a root node (don't have to have one node that connects to everything)
  - In a tree, we had an idea of children and parents, in a graph we have neighbors (no hierarchy)
- Just like how we could represent trees in multiple ways, we can represent graphs many ways as well, with advantages/disadvantages to each:
  - Adjacency Matrix - 2D Array
    - Visually clear what's going on
    - One axis (outside array) has an entry (inner array) for each node in the graph. If one node is connected to another node in the graph,

```
let matrix = [
/*          A      B      C      D      E      F   */
/*A*/    [true,  true,   true,  false,  true,  false],
/*B*/    [false, true,   false, false,  false, false],
/*C*/    [false, true,   true,  true,   false, false],
/*D*/    [false, false,  false, true,   false, false],
/*E*/    [true,  false,  false, false,  true,  false],
/*F*/    [false, false,  false, false,  true,  true]
];
```

  - Adjacency List - POJO
    - Object where every value in the graph has a key
    - Value for the key is an array with each other node that it is connected to (neighbors)
    - Easy to iterate through
    - Doesn't take up as much space as an Adjacency Matrix or Node
    - Can refer to the entire graph by referencing the object

```
let list = {
  a: ['b', 'c', 'e'],
  b: [],
  c: ['b', 'd'],
  d: [],
  e: ['a'],
  f: ['e']
};
```

TRAVERSE A GRAPH.
WE CAN USE RECURSION OR ITERATION TO TRAVERSE EACH NODE.

WE GENERALLY WANT TO KEEP TRACK OF EACH NODE THAT WE'VE VISITED ALREADY SO THAT WE DON'T GET TRAPPED IN CYCLES. EASIEST WAY TO DO THIS IS TO KEEP A SET VARIABLE THAT WE UPDATE AS WE TRAVERSE TO EACH NODE.

THE PROJECTS FROM W08D03 AND THEIR SOLUTIONS ARE A GREAT RESOURCE HERE.

BE COMFORTABLE WITH TAKING EITHER AN ITERATIVE OR A RECURSIVE APPROACH TO TRAVERSING A GRAPH, AS WELL AS BEING ABLE TO WORK WITH EITHER AN ADJACENCY LIST (LIKE IN THE FRIENDSOF PROBLEM) OR A NODE CLASS (LIKE IN THE BREADTHFIRSTSEARCH OR

- MAXVALUE PROBLEMS).
  PRACTICE TAKING THE IMPLEMENTATION THAT YOU DID IN THE PROJECT AND CONVERTING IT TO A DIFFERENT IMPLEMENTATION. YOU PROBABLY USED RECURSION FOR FRIENDSOF, SO TRY USING ITERATION WITH A STACK ARRAY, ETC. THE INTENTION OF ALL OF THESE CODE BLOCKS IS NOT TO MEMORIZE THEM! YOU SHOULD BE COMFORTABLE WITH REASONING OUT WHY WE ARE IMPLEMENTING THEM DIFFERENTLY.

  THE MAIN DIFFERENCE BETWEEN A NODE IMPLEMENTATION AND AN ADJACENCY LIST IS THAT WE ARE ACCESSING THE NODE'S NEIGHBORS ATTRIBUTE JUST LIKE WE ARE ACCESSING THE VALUES ON THE LIST (IE, WITH AN ADJACENCY LIST SAVED TO A GRAPH VARIABLE, GRAPH[NODE] GIVES ALL OF NODE'S NEIGHBORS).
  THE MAIN DIFFERENCE BETWEEN A DEPTH-FIRST AND BREADTH-FIRST IS UTILIZING A STACK VS A QUEUE.
  NODES AND OTHER OBJECTS WILL HAVE THEIR VALUE ACCESSED USING . NOTATION OR [] NOTATION. THE SAME IS NOT TRUE IF THE DATA SET YOU ARE WORKING WITH CONTAINS STRINGS OR NUMBERS. THEIR 'VALUE' IS JUST THE STRING/NUMBER ITSELF (NOT ACCESSED VIA .VAL).
  SOME POSSIBLE EXAMPLE IMPLEMENTATIONS (NOTE: THE SOLUTION TO THE ADJACENCY LIST PROBLEM FROM THE LOS-EMPTY IS IN HERE WITH SOME MINOR TWEAKING)

  USING A NODE IMPLEMENTATION WITH RECURSION:

```
// If you are unfamiliar, a Set is a data structure that does not allow for repeated values
// It makes sense to use here because it has constant lookup time with its `has` method
// and our visited nodes should never have repeats.
// We could have accomplished the same thing with a different data structure
// (object, array, etc.), but a Set makes sense with what we are tracking.
function depthFirstRecur(node, visited=new Set()) {
    // if this node has already been visited, then return early
    if (visited.has(node.val)) return;

    // otherwise it hasn't yet been visited,
    // so print it's val and mark it as visited.
    console.log(node.val);
    visited.add(node.val);

    // then explore each of its neighbors
    node.neighbors.forEach(neighbor => {
        depthFirstRecur(neighbor, visited);
    });
}

depthFirstRecur(f);
Using a NODE implementation with ITERATION:
// This is easy to swap to a breadth-first approach by using a queue instead of a stack!
// Instead of popping from the top, we can shift from the front
function depthFirstIter(node) {
    let visited = new Set();
    let stack = [ node ];

    while (stack.length) {
        let node = stack.pop();

        // if this node has already been visited, then skip this node
        if (visited.has(node.val)) continue;

        // otherwise it hasn't yet been visited,
        // so print it's val and mark it as visited.
        console.log(node.val);
        visited.add(node.val);

        // then add its neighbors to the stack to be explored
        stack.push(...node.neighbors);
    }
}
```

```
depthFirst(graph);
Using an ADJACENCY LIST with ITERATION:
// With starting node, not exploring all nodes, only the connected ones
function depthFirstIter(graph, startNode) {
  // Just like our node implementation, if we want to operate breadth-first, we
  // can utilize a queue instead of a stack, shifting instead of popping
  let stack = [startNode];
  let visited = new Set();

  while (stack.length > 0) {
    let node = stack.pop();
    if (visited.has(node)) continue;
    console.log(node)
    visited.add(node);
    stack.push(...graph[node]);
  }
}

depthFirstIter(f);
//Using an ADJACENCY LIST with RECURSION:
//One advantage of an adjacency list is that, since we have a reference to the whole graph,
//we can access nodes that aren't connected to our starting point.
//This may or may not be desired, so we can implement our functions differently to account for this feature.
function depthFirst(graph) {
    let visited = new Set();

    // This loop allows us to access every node/vertex, even if it wasn't connected
    // to where we started.
    // If we only wanted to reach points from a starting location, we could take in
    // that value as an argument and use it as the node directly with our helper
    // function, no need to loop.
    for (let node in graph) {
        _depthFirstRecur(node, graph, visited);
    }
}

function _depthFirstRecur(node, graph, visited) {
    if (visited.has(node)) return;

    console.log(node);
    visited.add(node);

    graph[node].forEach(neighbor => {
        _depthFirstRecur(neighbor, graph, visited);
    });
}
```

DEPTH FIRST SEARCH

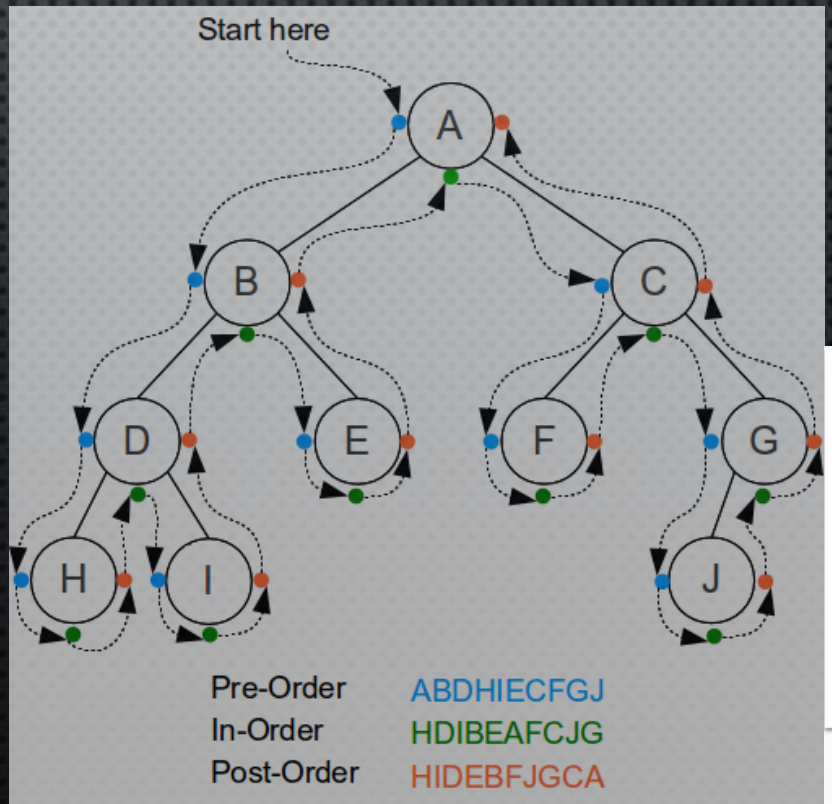
These are all depth first traversals, which means using recursion.

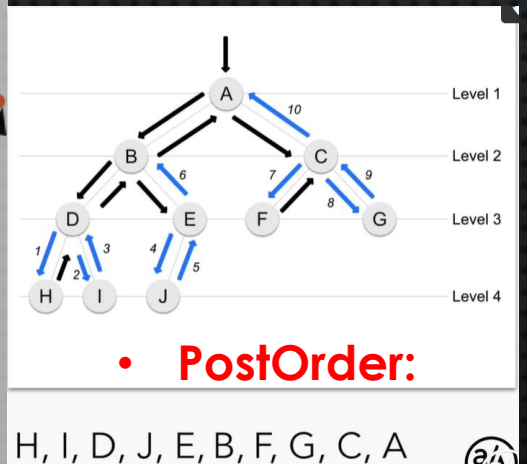
When doing a "Depth-first search" (DFS):

- **the search tree is deepened as much as possible on each child before going to the next sibling.**

- **There are three common ways to traverse them in depth-first order: in-order, pre-order and post-order.**

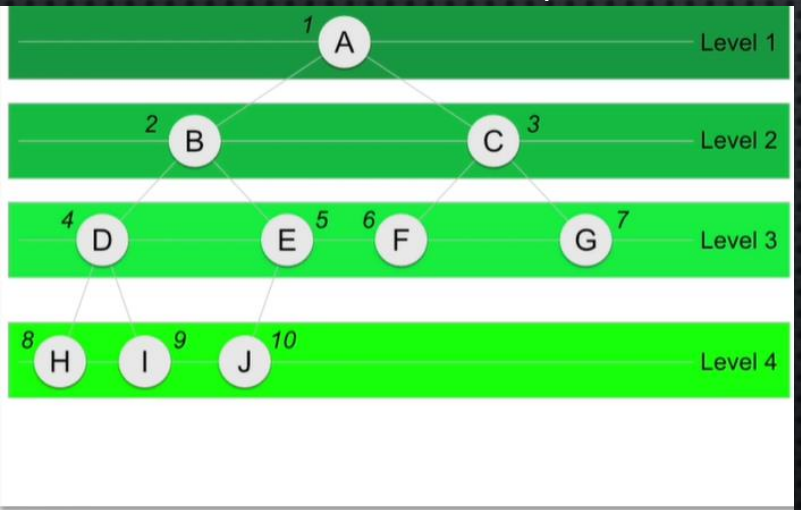
- **InOrder:**
  - **1: Recursively visit the left sub tree**
  - **2: Access the data of the current node**
  - **3: Recursively visit the right sub tree**
- **PreOrder:**
  - **1: Access the data of the current node**
  - **2: Recursively visit the left sub tree**
  - **3: Recursively visit the right sub tree**
- **PostOrder:**
  - **1: Recursively visit the left sub tree**
  - **2: Recursively visit the right sub tree**
  - **3: Access the data of the current node**





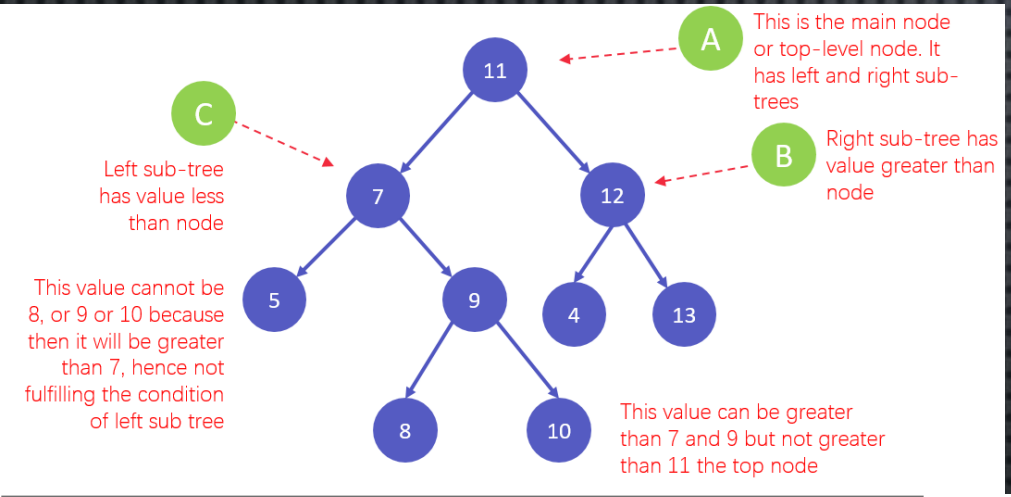
BREADTH FIRST SEARCH

Breadth-first Search on the adjacency list graph. When doing breadth first, iterative is simpler and easier. We can use a queue to do this.

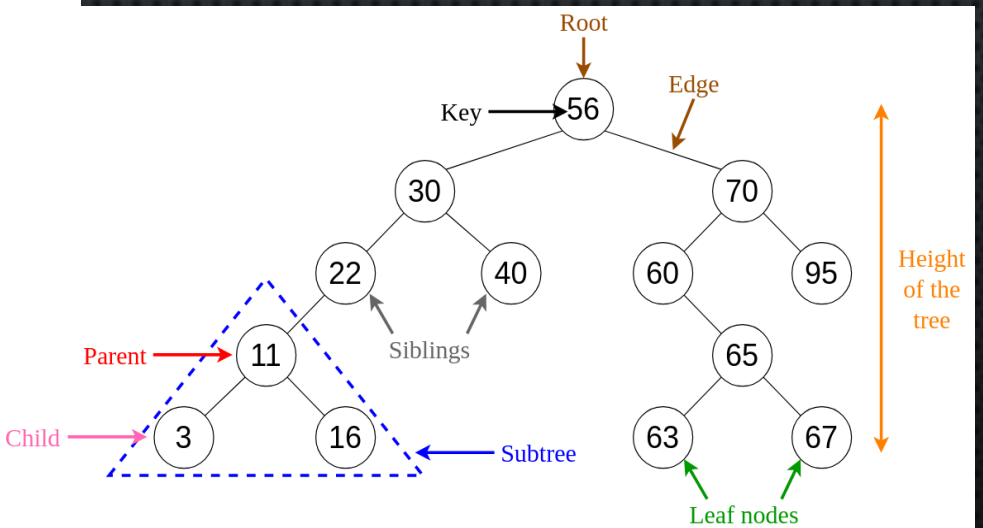


A, B, C, D, E, F, G, H, I, J

Vocab Binary Search Tree



Binary Search Tree if:
the left subtree contains values less than the root AND
the right subtree contains values greater than or equal to the root AND
the left subtree is a Binary Search Tree AND
the right subtree is a Binary Search Tree

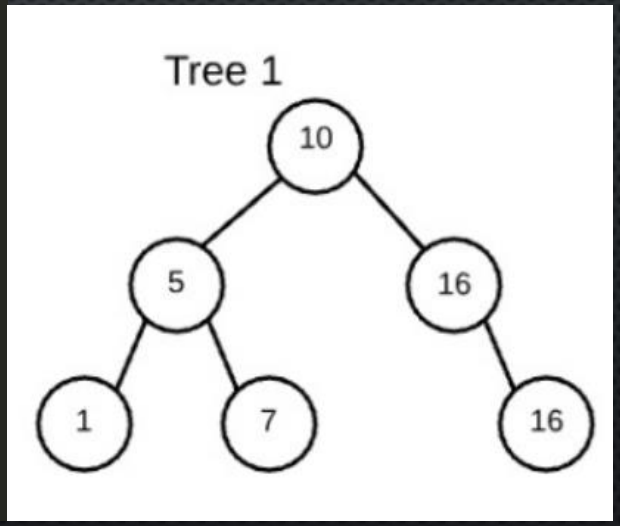


Explain and implement a binary search tree

- A Binary Search Tree is a special kind of Binary Tree where the following is true:
- given any node of the tree, the values in the left subtree must all be strictly less than the given node's value.
- and the values in the right subtree must all be greater than or equal to the given node's value
- Or to say it with recursion: the left subtree contains values less than the root
- AND the right subtree contains values greater than or equal to the root
- AND the left subtree is a Binary Search Tree
- AND the right subtree is a Binary Search Tree
- Some definitions of binary search trees allow duplicates and some do not. We can write our code to deal with the duplicates though.

```
class TreeNode {
    constructor(val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
let ten = new TreeNode("10");
let five = new TreeNode("5");
let sixteen = new TreeNode("16");
let one = new TreeNode("1");
let seven = new TreeNode("7");
let sixteenDuplicate = new TreeNode("16");
ten.left = five;
ten.right = sixteen;
five.left = one;
five.right = seven;
sixteen.right = sixteenDuplicate;
console.log("ten: ", ten);
console.log("----------ten above^----------");
console.log("five: ", five);
console.log("----------five----------");
console.log("sixteen: ", sixteen);
console.log("----------sixteen----------");
console.log(" one: ", one);
console.log("----------one----------");
console.log("seven: ", seven);
console.log("----------seven----------");
console.log("sixteenDuplicate: ", sixteenDuplicate);
```

```
ten:  TreeNode {
  val: '10',
  left:
   TreeNode {
     val: '5',
     left: TreeNode { val: '1', left: null, right: null },
     right: TreeNode { val: '7', left: null, right: null } },
  right:
   TreeNode {
     val: '16',
     left: null,
     right: TreeNode { val: '16', left: null, right: null } } }
----------ten above^----------
five:  TreeNode {
  val: '5',
  left: TreeNode { val: '1', left: null, right: null },
  right: TreeNode { val: '7', left: null, right: null } }
----------five----------
sixteen:  TreeNode {
  val: '16',
  left: null,
  right: TreeNode { val: '16', left: null, right: null } }
----------sixteen----------
 one:  TreeNode { val: '1', left: null, right: null }
----------one----------
seven:  TreeNode { val: '7', left: null, right: null }
----------seven----------
sixteenDuplicate:  TreeNode { val: '16', left: null, right: null }
```



BST Implementation

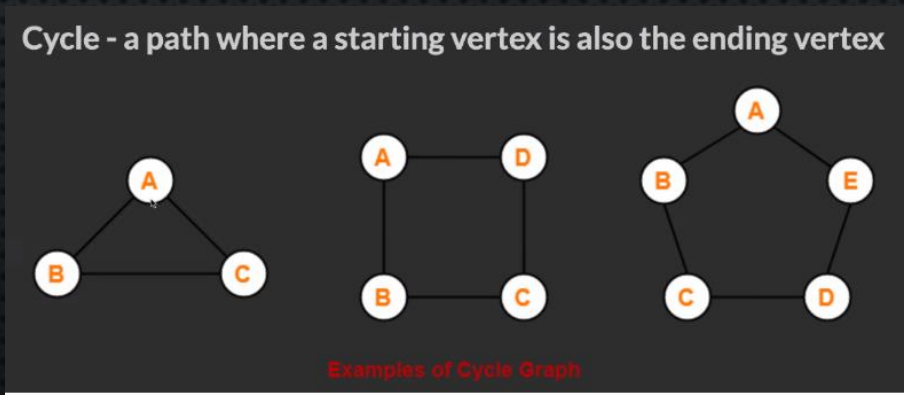
```
class TreeNode {
    constructor(val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
class BST {
    constructor() {
        this.root = null;
    }
    insert(val, currentNode = this.root) {
        if (!this.root) {
            this.root = new TreeNode(val);
            return;
        }
        if (val < currentNode.val) {
            if (!currentNode.left) {
                currentNode.left = new TreeNode(val);
            } else {
                this.insert(val, currentNode.left);
            }
        } else {
            if (!currentNode.right) {
                currentNode.right = new TreeNode(val);
            } else {
                this.insert(val, currentNode.right);
            }
        }
    }
```

```
    searchRecur(val, currentNode = this.root) {
        if (!currentNode) return false;
        if (val < currentNode.val) {
            return this.searchRecur(val, currentNode.left);
        } else if (val > currentNode.val) {
            return this.searchRecur(val, currentNode.right);
        } else {
            return true;
        }
    }
    searchIter(val) {
        let currentNode = this.root;
        while (currentNode) {
            if (val < currentNode.val) {
                currentNode = currentNode.left;
            } else if (val > currentNode.val) {
                currentNode = currentNode.right;
            } else {
                return true;
            }
        }
        return false;
    }
}
```


```
class TreeNode {
    constructor(val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
class BST {
    constructor() {
        this.root = null;
    }
    insert(val, currentNode = this.root) {
        if (!this.root) {
            this.root = new TreeNode(val);
            return;
        }
        if (val < currentNode.val) {
            if (!currentNode.left) {
                currentNode.left = new TreeNode(val);
            } else {
                this.insert(val, currentNode.left);
            }
        } else {
            if (!currentNode.right) {
                currentNode.right = new TreeNode(val);
            } else {
                this.insert(val, currentNode.right);
            }
        }
    }
    searchRecur(val, currentNode = this.root) {
        if (!currentNode) return false;
        if (val < currentNode.val) {
            return this.searchRecur(val, currentNode.left);
        } else if (val > currentNode.val) {
            return this.searchRecur(val, currentNode.right);
        } else {
            return true;
        }
    }
    searchIter(val) {
        let currentNode = this.root;
        while (currentNode) {
            if (val < currentNode.val) {
                currentNode = currentNode.left;
            } else if (val > currentNode.val) {
                currentNode = currentNode.right;
            } else {
                return true;
            }
        }
        return false;
    }
}
```

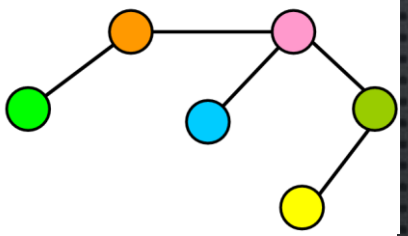
Graph

- A graph is **any** collection of nodes and edges.
- It doesn't need to have a root node (not every node needs to be accessible from a single node)
- It can have cycles (a group of nodes whose paths begin and end at the same node)
  - Cycles are not always "isolated", they can be one part of a larger graph.
    - You can detect them by starting your search on a specific node
    - and finding a path that takes you back to that same node.
- Any number of edges may leave a given node
- A Path is a sequence of nodes on a graph

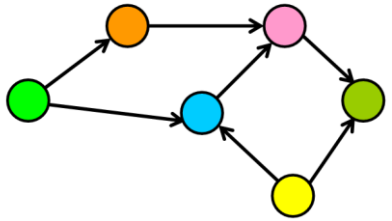


Graph

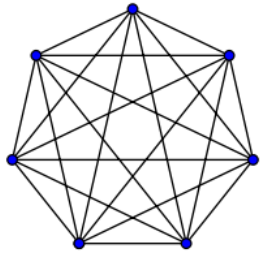
Sparse Graph



Dense Graph



Complete Graph

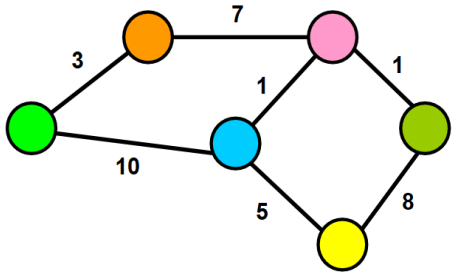


- All nodes connect to all other nodes

- Sparse Graph - Few edges
- When the number of edges in the graph is significantly fewer than the maximum number of edges, the graph is sparse.

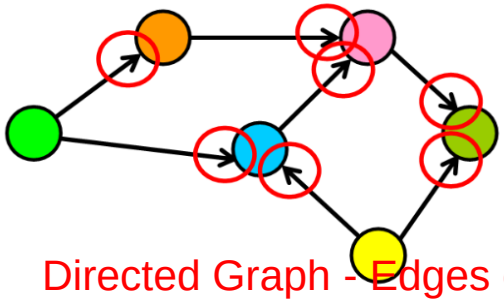
When the number of edges in the graph is close to the maximum number of edges, the graph is dense.

Weighted Graph



- Weighted Graph - Edges have a cost or a weight to traversal

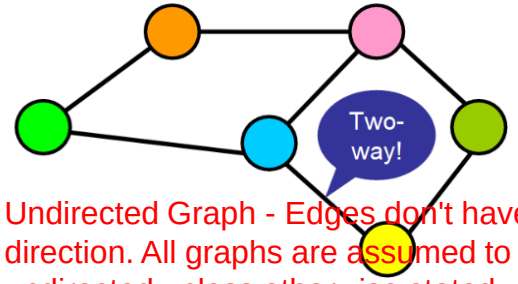
Directed Graph



Directed Graph - Edges only go one direction

- Directed Graph - Edges have a direction

Undirected Graph



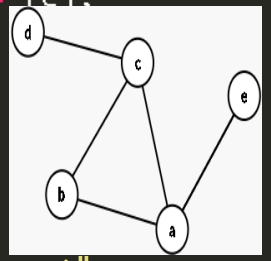
Undirected Graph - Edges don't have a direction. All graphs are assumed to be undirected unless otherwise stated

- Undirected Graph - Edges don't have a direction. All graphs are assumed to be undirected unless otherwise stated

WAYS TO REFERENCE GRAPH NODES

- ADJACENCY LIST:

```
class GraphNode {
    constructor(val) {
        this.val = val;
        this.neighbors = [];
    }
}
let a = new GraphNode("a");
let b = new GraphNode("b");
let c = new GraphNode("c");
let d = new GraphNode("d");
let e = new GraphNode("e");
let f = new GraphNode("f");
a.neighbors = [e, c, b];
c.neighbors = [b, d];
e.neighbors = [a];
f.neighbors = [e];
console.log(
    "a:-----> ",
    a,
    "\n",
    "a.val----->",
    a.val,
    "\n",
    "a.neighbours----->",
    a.neighbors,
    "\n"
);
console.log(
    "--------------------------a--"
);
```




```
~ graphs : (master) node graph.js
a:-----> GraphNode {
  val: 'a',
  neighbors:
   [ GraphNode { val: 'e', neighbors: [Array] },
     GraphNode { val: 'c', neighbors: [Array] },
     GraphNode { val: 'b', neighbors: [] } ] }
 a.val-----> a
 a.neighbours-----> [ GraphNode { val: 'e', neighbors: [ [GraphNode] ] },
  GraphNode { val: 'c', neighbors: [ [GraphNode], [GraphNode] ] },
  GraphNode { val: 'b', neighbors: [] } ]

--------------------------a--
b:-----> GraphNode { val: 'b', neighbors: [] }
 b.val-----> b
 b.neighbours-----> []

--------------------------b--
c:-----> GraphNode {
  val: 'c',
  neighbors:
   [ GraphNode { val: 'b', neighbors: [] },
     GraphNode { val: 'd', neighbors: [] } ] }
 c.val-----> c
 c.neighbours-----> [ GraphNode { val: 'b', neighbors: [] },
  GraphNode { val: 'd', neighbors: [] } ]

--------------------------c--
d:-----> GraphNode { val: 'd', neighbors: [] }
 d.val-----> d
 d.neighbours-----> []

--------------------------d--
e:-----> GraphNode {
  val: 'e',
  neighbors: [ GraphNode { val: 'a', neighbors: [Array] } ] }
 e.val-----> e
 e.neighbours-----> [ GraphNode {
    val: 'a',
    neighbors: [ [GraphNode], [GraphNode], [GraphNode] ] } ]
```

Adjacency Matrix

The row index will corespond to the source of an edge and the column index will correspond to the edges destination.

- When the edges have a direction, `matrix[i][j]` may not be the same as `matrix[j][i]`
- It is common to say that a node is adjacent to itself so `matrix[x][x]` is true for any node
- Will be O(n^2) space complexity

let matrix = [

	A	B	C	D	E	F
A	[True,	True,	True,	False,	True,	False]
B	[False,	True,	False,	False,	False,	False]
C	[False,	True,	True,	True,	False,	False]
D	[False,	False,	False,	True,	False,	False]
E	[True,	False,	False,	False,	True,	False]
F	[False,	False,	False,	False,	True,	True]

];

> Adjacency List

- We use an object where keys represent the node labels.
- The values associated with the keys will be an array containing all adjacent nodes,
- ie, the nodes which this instance (represented by a key) should hold reference to.
- An adjacency list is easy to implement and allows us to refer to the entire graph by simply referencing the object.

> The space required for an adjacency list is the number of edges in the graph.

BASIC-GRAPH-CLASS

```
class Graph {
    constructor() {
        this.adjList = {};
    }
```

```
addVertex(vertex) {
        if (!this.adjList[vertex]) this.adjList[vertex] = [];
    }
addEdges(srcValue, destValue) {
        this.addVertex(srcValue);
        this.addVertex(destValue);
        this.adjList[srcValue].push(destValue);
        this.adjList[destValue].push(srcValue);
    }
buildGraph(edges) {
        edges.forEach((ele) => {
            this.addEdges(ele[0], ele[1]);
        });
        return this.adjList;
    }
breadthFirstTraversal(startingVertex) {
        const queue = [startingVertex];
        const visited = new Set();
        const result = new Array();
        while (queue.length) {
            const value = queue.shift();
            if (visited.has(value)) continue;
            result.push(value);
            visited.add(value);
            queue.push(...this.adjList[value]);
        }
        return result;
    }
```

```
depthFirstTraversalIterative(startingVertex) {
        const stack = [startingVertex];
        const visited = new Set();
        const result = new Array();
        while (stack.length) {
            const value = stack.pop();
            if (visited.has(value)) continue;
            result.push(value);
            visited.add(value);
            stack.push(...this.adjList[value])
        }
        return result;
    }

depthFirstTraversalRecursive(
        startingVertex,
        visited = new Set(),
        vertices = []
    ) {
        if (visited.has(startingVertex)) return [];
        vertices.push(startingVertex);
        visited.add(startingVertex);
        this.adjList[startingVertex].forEach((vertex) => {
            this.depthFirstTraversalRecursive(vertex, visited, vertices);
        });
        return [...vertices];
    }
```

```
class Graph {
    constructor() {
        this.adjList = {};
    }
    addVertex(vertex) {
        if (!this.adjList[vertex]) this.adjList[vertex] = [];
    }
    addEdges(srcValue, destValue) {
        this.addVertex(srcValue);
        this.addVertex(destValue);
        this.adjList[srcValue].push(destValue);
        this.adjList[destValue].push(srcValue);
    }
    buildGraph(edges) {
        edges.forEach((ele) => {
            this.addEdges(ele[0], ele[1]);
        });
        return this.adjList;
    }
    breadthFirstTraversal(startingVertex) {
        const queue = [startingVertex];
        const visited = new Set();
        const result = new Array();
        while (queue.length) {
            const value = queue.shift();
            if (visited.has(value)) continue;
            result.push(value);
            visited.add(value);
            queue.push(...this.adjList[value]);
        }
        return result;
    }
    depthFirstTraversalIterative(startingVertex) {
        const stack = [startingVertex];
        const visited = new Set();
        const result = new Array();
        while (stack.length) {
            const value = stack.pop();
            if (visited.has(value)) continue;
            result.push(value);
            visited.add(value);
            stack.push(...this.adjList[value]);
        }
        return result;
    }
    depthFirstTraversalRecursive(
        startingVertex,
        visited = new Set(),
        vertices = []
    ) {
        if (visited.has(startingVertex)) return [];
        vertices.push(startingVertex);
        visited.add(startingVertex);
        this.adjList[startingVertex].forEach((vertex) => {
            this.depthFirstTraversalRecursive(vertex, visited, vertices);
        });
        return [...vertices];
    }
}
```

Implement-graph

```
class GraphNode {
    constructor(val) {
        this.val = val;
        this.neighbors = [];
    }
}
let a = new GraphNode("a");
let b = new GraphNode("b");
let c = new GraphNode("c");
let d = new GraphNode("d");
let e = new GraphNode("e");
let f = new GraphNode("f");
a.neighbors = [b, c, e];
c.neighbors = [b, d];
e.neighbors = [a];
f.neighbors = [e];
```

```
// Recursive case given a node
function depthFirstNode(node, visited = new Set()) {
    // We utilize a Set for our memo instead of a regular object
    // A Set is like an object where the keys have no values
    if (visited.has(node.val)) return;
    console.log(node.val);
    visited.add(node.val);
    node.neighbors.forEach((neighbor) => depthFirstNode(neighbor, visited));
}
// We invoke this with an instance of the graph node class
depthFirstNode(f);
// depthFirstNode(node, visited);   // 1st frame,   node.val = 'f', visited = {'f'}
// depthFirstNode(node, visited);   // 2nd frame,   node.val = 'e', visited = {'f', 'e'}
// depthFirstNode(node, visited);   // 3rd frame,   node.val = 'a', visited = {'f', 'e', 'a'}
// depthFirstNode(node, visited);   // 4th frame,   node.val = 'b', visited = {'f', 'e', 'a', 'b'}, popped off stack
// depthFirstNode(node, visited);   // 5th frame,   node.val = 'c', visited = {'f', 'e', 'a', 'b', 'c'}
// depthFirstNode(node, visited);   // 6th frame,   node.val = 'd', visited = {'f', 'e', 'a', 'b', 'c', 'd'}, popped off stack
// 5th frame,     node.val = 'c'     returns undefined,     popped off the stack
// 4th frame,     node.val = 'e'     vistied.has(node.val) = true,    returns undefined,     popped off the stack
// 3rd frame,     node.val = 'a'     returns undefined,     popped off the stack
// 2nd frame,     node.val = 'e'     returns undefined,     popped off the stack
// 1st frame,     node.val = 'f'     returns undefined,     popped off the stack
// Final return value: undefined
```

```
~ graphs : (master)
f
e
a
b
c
d
~ graphs : (master)
```

```
class GraphNode {
    constructor(val) {
        this.val = val;
        this.neighbors = [];
    }
}
let a = new GraphNode("a");
let b = new GraphNode("b");
let c = new GraphNode("c");
let d = new GraphNode("d");
let e = new GraphNode("e");
let f = new GraphNode("f");
let adjList = {
    a: ["b", "c", "e"],
    b: [],
    c: ["b", "d"],
    d: [],
    e: ["a"],
    f: ["e"],
};
```

```
function depthFirstIter(node) {
    let visited = new Set(); // a set doesn't
    let stack = [node];

    while (stack.length) {
        let node = stack.pop();

        if (visited.has(node.val)) continue;

        console.log(node.val);
        visited.add(node.val);

        stack.push(...node.neighbors);
    }
}

depthFirstIter(f);
console.log(
```

```
function depthFirstAdj(graph, node, visited = new Set()) {
    // if this node has already been visited, then return early
    if (visited.has(node)) return;
    // otherwise it hasn't yet been visited,
    // so print it's val and mark it as visited.
    console.log(node);
    visited.add(node);
    // then explore each of its neighbors
    graph[node].forEach((neighbor) => {
        depthFirstAdj(graph, neighbor, visited);
    });
}
//! We invoke this with the adjacency list and a key from the list.
depthFirstAdj(adjList, "f");
console.log(`\n----------\n`);
depthFirstAdj(adjList, "a");

// We see that when we invoke this function with 'a' we are not able to access the 'f' node
```

```
~ graphs
f
e
a
b
c
d

----------

a
b
c
d
e
```

```
function completeDepthFirst(graph) {
    let visited = new Set();
    for (let node in graph) {
        _depthFirstRecur(node, graph, visited);
    }
}
function _depthFirstRecur(node, graph, visited) {
    // if this node has already been visited, then return early
    if (visited.has(node)) return;
    // otherwise it hasn't yet been visited,
    // so print it's val and mark it as visited.
    console.log(node);
    visited.add(node);
    // then explore each of its neighbors
    graph[node].forEach((neighbor) => {
        _depthFirstRecur(neighbor, graph, visited);
    });
}
// We invoke this with the adjacency list and a key from the list.
completeDepthFirst(adjList, "f");
console.log(`\n----------\n`);
completeDepthFirst(adjList, "a");
```

```
~ graphs
a
b
c
d
e
f

----------

a
b
c
d
e
f
~ graphs
```

PLAYGROUND

```
class _Vertex {
    constructor(value) {
        this.value = value;
        this.neighbors = new Array();
    }
}
class Graph {
    constructor() {
        this.graff = new Object();
    }
addVertex(value) {
      this.graff[value] = new _Vertex(value);
  }
  addEdge(value1, value2) {
      // if they do not exist in graff, we have to add the
      if (!this.graff[value1]) this.addVertex(value1);
      if (!this.graff[value2]) this.addVertex(value2);
      // just create vars for each vertex
      const value1Vertex = this.graff[value1];
      const value2Vertex = this.graff[value2];
      // make sure the neighbor connection is accomplished
      value2Vertex.neighbors.push(value1Vertex);
      value1Vertex.neighbors.push(value2Vertex);
  }
getVertices() {
      return Object.keys(this.graff);
  }
  getEdges() {
      const li = new Array();
      for (let srcValue in this.graff) {
          const vertex = this.graff[srcValue];
          // push the connection between two vertices i
          vertex.neighbors.forEach((neiVertex) =>
              li.push([srcValue, neiVertex.value])
          );
      }
      return li;
  }
```

```
adjList() {
      const adjList = new Object();
      for (let srcValue in this.graff) {
          // sets up var and grabs vertex
          const vertex = this.graff[srcValue];
          // sets up the "neighbors" array
          adjList[srcValue] = new Array();
          // we want push each neighboring vertex in the "neighbors" array
          vertex.neighbors.forEach((neighbor) =>
              adjList[srcValue].push(neighbor.value)
          );
      }
      return adjList;
  }
```

```
      breadthFirstTraversal(startValue) {
          const adjList = this.adjList();
          const queue = [startValue]; // queue = ["b", "c", "d"]
          const resLi = new Array(); // ["a"]
          const visited = new Set();
          // we want use queue to pop off and continually add neighbors to iterate
          while (queue.length > 0) {
              // shift off current value in "front of line"
              const value = queue.shift(); // "b"
              if (visited.has(value)) continue;
              resLi.push(value);
              visited.add(value);
              // use spread operator to spread "neighbor" values into the queue
              queue.push(...adjList[value]); // "b", "c", "d"
          }
          return resLi;
      }
```

```
    depthFirstTraversal(value, visited = new Set(), resLi = new Array()) {
          // THIS PSEUDO CODE WILL GRAB EVERY VERTEX
          // const vertices = getVertices();
          // vertices.forEach.....
          //   vertex.neighbors.forEach......
          // THIS CODE HAS A STARTING POINT
          if (visited.has(value)) return;
          const adjList = this.adjList();
          // add to "visited" set to not have to visit nodes that have already been visi
          visited.add(value);
          resLi.push(value);
          // iterating through all neighbors of that current or starting "value"
          adjList[value].forEach((neighbor) =>
              this.depthFirstTraversal(neighbor, visited, resLi)
          );
          return resLi;
      }
  }
```

```
const unweightedUndirectedAdjList = {
    a: ["b", "c", "d"],
    b: ["a", "c", "e"],
    c: ["a", "b", "f", "g"],
    d: ["a", "g"],
    e: ["b"],
    f: ["c", "g"],
    g: ["c", "f", "g"],
    h: [],
};
```

PLAYGROUND TESTING

```
const unweightedUndirectedAdjList = {
    a: ["b", "c", "d"],
    b: ["a", "c", "e"],
    c: ["a", "b", "f", "g"],
    d: ["a", "g"],
    e: ["b"],
    f: ["c", "g"],
    g: ["c", "f", "g"],
    h: [],
};
const graph = new Graph();
graph.addEdge("a", "b");
graph.addEdge("c", "g");
graph.addEdge("f", "g");
graph.addEdge("a", "c");
graph.addEdge("a", "d");
graph.addEdge("c", "f");
graph.addEdge("d", "g");
graph.addEdge("b", "c");
graph.addEdge("b", "e");
graph.addVertex("h");
console.log(
    "-----------------------------------Before printing-
);
console.log(graph.getVertices());
console.log(
    "console.log( graph.getVertices() ): _________________
);

console.log(graph.getEdges());
console.log(
    "console.log( graph.getEdges() ): ___________________
);
console.log(graph.adjList());
console.log(
    "console.log( graph.adjList() ): ____________________
);
console.log("BFS", graph.breadthFirstTraversal("a"));
console.log("DFS", graph.depthFirstTraversal("a"));
```

```
╗ ~ graphs : (master) node implement-graph.js
-----------------------------------Before printing--
[ 'a', 'b', 'c', 'g', 'f', 'd', 'e', 'h' ]
console.log( graph.getVertices() ): _____________________
[ [ 'a', 'b' ],
  [ 'a', 'c' ],
  [ 'a', 'd' ],
  [ 'b', 'a' ],
  [ 'b', 'c' ],
  [ 'b', 'e' ],
  [ 'c', 'g' ],
  [ 'c', 'a' ],
  [ 'c', 'f' ],
  [ 'c', 'b' ],
  [ 'g', 'c' ],
  [ 'g', 'f' ],
  [ 'g', 'd' ],
  [ 'f', 'g' ],
  [ 'f', 'c' ],
  [ 'd', 'a' ],
  [ 'd', 'g' ],
  [ 'e', 'b' ] ]
console.log( graph.getEdges() ): ____________________
{ a: [ 'b', 'c', 'd' ],
  b: [ 'a', 'c', 'e' ],
  c: [ 'g', 'a', 'f', 'b' ],
  g: [ 'c', 'f', 'd' ],
  f: [ 'g', 'c' ],
  d: [ 'a', 'g' ],
  e: [ 'b' ],
  h: [] }
console.log( graph.adjList() ): _____________________
BFS [ 'a', 'b', 'c', 'd', 'e', 'g', 'f' ]
DFS [ 'a', 'b', 'c', 'g', 'f', 'd', 'e' ]
╗ ~ graphs : (master)
```

PROJECTS: #1

```
//------------------Binary-Tree-Proj-1-----------------------------------
class TreeNode {
    constructor(val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
//-----------------------------tree-order-------------------------
function preOrderArray(root) {
    if (!root) return [];
    return [
        root.val,
        ...preOrderArray(root.left),
        ...preOrderArray(root.right),
    ];
}
function inOrderArray(root) {
    if (!root) return [];
    return [...inOrderArray(root.left), root.val, ...inOrderArray(root.right)];
}
function postOrderArray(root) {
    if (!root) return [];
    return [
        ...postOrderArray(root.left),
        ...postOrderArray(root.right),
        root.val,
    ];
}
```

```
//------------------------------dfs-----------------------------------
// Iterative Solution
function dfs(root) {
    if (!root) return [];
    Let stack = [root];
    Let vals = [];
    while (stack.length) {
        Let node = stack.pop();
        vals.push(node.val);
        if (node.right) stack.push(node.right);
        if (node.left) stack.push(node.left);
    }
    return vals;
}
// Recursive Solution
const dfsRec = (root) => {
    if (!root) return [];
    return [root.val, ...dfsRec(root.left), ...dfsRec(root.right)];
};
```

```
//------------------------BFS---------------------
function bfs(root) {
    if (!root) return [];
    Let queue = [root];
    Let vals = [];
    while (queue.length) {
        Let node = queue.shift();
        vals.push(node.val);
        if (node.left) queue.push(node.left);
        if (node.right) queue.push(node.right);
    }
    return vals;
}
```

PROJECT 1-LEETCODE 105

Problem:

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

For example, given

```
preorder = [3,9,20,15,7]
inorder = [9,3,15,20,7]
```

Return the following binary tree:

```
    3
   / \
  9  20
    /  \
   15   7
```

```
//---------------Build-Tree-----leetcode---require treeNode-------------------
function buildTree(preorder, inorder) {
    if (!preorder.length && !inorder.length) return null;
    // the first element in the preorder array is the root
    Let root = new TreeNode(preorder[0]);
    Let rootIdx = inorder.indexOf(preorder[0]);
    // the left subtree is everything in the inorder array to the left of the root
    Let leftInorder = inorder.slice(0, rootIdx);
    // the right subtree is everything in the inorder array to the right of the root
    Let rightInorder = inorder.slice(rootIdx + 1);

// to build the left subtree, the values in the leftPreorder array have to be the same as the ones in the leftInorder array
    Let leftPreorder = preorder.filter((val) => leftInorder.includes(val));

// to build the left subtree, the values in the leftPreorder array have to be the same as the ones in the leftInorder array
    Let rightPreorder = preorder.filter((val) => rightInorder.includes(val));
    root.left = buildTree(leftPreorder, leftInorder);
    root.right = buildTree(rightPreorder, rightInorder);
    return root;
}
```

PROJECT 2 BST

```
┐ ~ Code-Snippets : (master) node sept-study-guide.js
sam: TreeNode { val: 'Sam', left: null, right: null }
riv: TreeNode {
  val: 'River',
  left: TreeNode { val: 'Dean', left: null, right: null },
  right: null }
tree.banana:---> undefined
tree 2---> [ 'sam', 'river', 'Barry', 'dean' ]
tree 2--->after---> [ 'sam', 'river', 'Barry', 'dean', child: 'sam' ]
┐ ~ Code-Snippets : (master) ∏
```

```
class TreeNode {
    constructor(val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
```

```
class BST {
    constructor() {
        this.root = null;
    }
    insert(val, currentNode = this.root) {
        if (!this.root) {
            this.root = new TreeNode(val);
            return;
        }
        if (val < currentNode.val) {
            if (!currentNode.left) {
                currentNode.left = new TreeNode(val);
            } else {
                this.insert(val, currentNode.left);
            }
        } else {
            if (!currentNode.right) {
                currentNode.right = new TreeNode(val);
            } else {
                this.insert(val, currentNode.right);
            }
        }
    }
```

```
function getHeight(root) {
    if (!root) return -1;
    return 1 + Math.max(getHeight(root.left), getHeight(root.right));
}
```

```
    searchRecur(val, currentNode = this.root) {
        if (!currentNode) return false;
        if (val < currentNode.val) {
            return this.searchRecur(val, currentNode.left);
        } else if (val > currentNode.val) {
            return this.searchRecur(val, currentNode.right);
        } else {
            return true;
        }
    }
    searchIter(val) {
        let currentNode = this.root;
        while (currentNode) {
            if (val < currentNode.val) {
                currentNode = currentNode.left;
            } else if (val > currentNode.val) {
                currentNode = currentNode.right;
            } else {
                return true;
            }
        }
        return false;
    }
}
```

```
const sam = new TreeNode("Sam");
console.log("sam:", sam);
const riv = new TreeNode("River");
const barry = new TreeNode("Barry");
const dean = new TreeNode("Dean");
const tree = new BST();
tree.root = sam;
sam.left = riv;
sam.right = barry;
riv.left = dean;
console.log("riv:", riv);
console.log("tree.banana:--->", tree.banana);
const tree2 = ["sam", "river", "Barry", "dean"];
console.log("tree 2--->", tree2);
tree2.child = "sam";
console.log("tree 2--->after--->", tree2);
```

```
let tree = new BST();
tree.insert(10);
tree.insert(5);
tree.insert(16);
tree.insert(1);
tree.insert(7);
tree.insert(16);
console.log("tree", tree);
/*
expect(tree.root.val).to.equal(10);
            expect(tree.root.left.val).to.equal(5);
            expect(tree.root.right.val).to.equal(16);
            expect(tree.root.left.left.val).to.equal(1);
            expect(tree.root.left.right.val).to.equal(7);
            expect(tree.root.right.right.val).to.equal(16);
*/

console.log("tree.root.left.val: ", tree.root.left.val);
console.log("tree.root.right.val: ", tree.root.right.val);
console.log("tree.root.left.left.val: ", tree.root.left.left.val);
console.log("tree.root.left.right.val: ", tree.root.left.right.val);
console.log("tree.root.right.right.val: ", tree.root.right.right.val);
/*
  tree.root.left.val:  5
tree.root.right.val:  16
tree.root.left.left.val:  1
tree.root.left.right.val:  7
tree.root.right.right.val:  16
*/
```

PROJ-2-LEETCODE-108

```
//---------------------------------proj2 leetcode 108-----------------------------
function sortedArrayToBST( nums ) {
    if ( !nums.length ) return null;
    Let midIdx = Math.floor( nums.length / 2 );
    // we use the middle element of the array as our root
    Let root = new TreeNode( nums[ midIdx ] );
    // the root's left subtree is a recursive call on the left side of the array
    root.left = sortedArrayToBST( nums.slice( 0, midIdx ) );
    // the root's right subtree is a recursive call on the right side of the array
    root.right = sortedArrayToBST( nums.slice( midIdx + 1 ) );
    return root;
}
//----------OR-------------------------------------
Let sortedArrayToBST = function ( nums ) {
    return generate( nums, 0, nums.length - 1 );
};
Let generate = function ( nums, start, end ) {
    if ( start > end ) {
        return null;
    }
    Let midIndex = start + parseInt( ( end - start ) / 2 );
    Let midVal = nums[ midIndex ];
    Let node = new TreeNode( midVal );
    node.left = generate( nums, start, midIndex - 1 );
    node.right = generate( nums, midIndex + 1, end );
    return node;
};
```

```
function sortedArrayToBST(nums) {
    if (nums.length === 0) return null;
    let height = 0;
    let midIdx = Math.floor(nums.length/2);
    let node = new TreeNode(nums[midIdx]);
    let leftTree = nums.slice(0, midIdx);
    let rightTree = nums.slice(midIdx + 1);
    node.left = sortedArrayToBST(leftTree);
    node.right = sortedArrayToBST(rightTree);
    height++;
    return node;
}

let nums = [-10, -3, 0, 5, 9];
console.log(sortedArrayToBST(nums));
```

PROJ-2-LEETCODE-110

```
function getHeight(root) {
    if (!root) return -1;
    return 1 + Math.max(getHeight(root.left), getHeight(root.right));
}

function isBalanced(root) {
    if (!root) return true;

    // check to see if the top level of the tree is balanced
    Let heightDifference =
        Math.abs(getHeight(root.left) - getHeight(root.right)) <= 1;

    // also check to see if the left and right subtrees are balanced
    return heightDifference && isBalanced(root.left) && isBalanced(root.right);
}
```

PROJ-2-LEETCODE-450

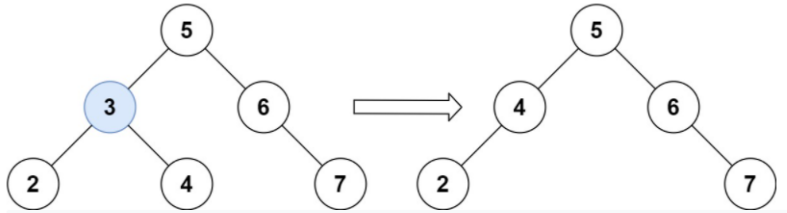
Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

1. Search for a node to remove.
2. If the node is found, delete the node.

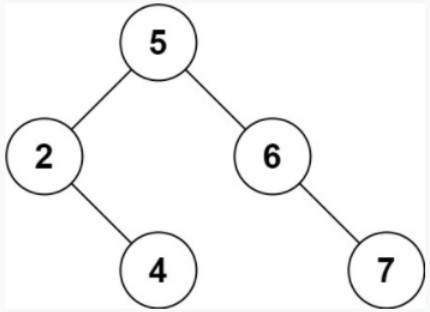
Follow up: Can you solve it with time complexity `O(height of tree)`?

Example 1:




```
Input: root = [5,3,6,2,4,null,7], key = 3
Output: [5,4,6,2,null,null,7]
Explanation: Given key to delete is 3. So we find the node with value 3 and delete it.
One valid answer is [5,4,6,2,null,null,7], shown in the above BST.
Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.
```

```
Input: root = [5,3,6,2,4,null,7], key = 3
Output: [5,4,6,2,null,null,7]
Explanation: Given key to delete is 3. So we find the node with value 3 and delete it.
One valid answer is [5,4,6,2,null,null,7], shown in the above BST.
Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.
```



Example 2:

```
Input: root = [5,3,6,2,4,null,7], key = 0
Output: [5,3,6,2,4,null,7]
Explanation: The tree does not contain a node with value = 0.
```

Example 3:

```
Input: root = [], key = 0
Output: []
```

Constraints:

- The number of nodes in the tree is in the range `[0, 10^4]`.
- `-10^5 <= Node.val <= 10^5`
- Each node has a **unique** value.
- `root` is a valid binary search tree.
- `-10^5 <= key <= 10^5`

```javascript
// View the full problem and run the test cases at:
//  https://leetcode.com/problems/delete-node-in-a-bst/
function deleteNode(root, key) {
    if (!root) return null; // return null if key isn't in the tree
    if (key < root.val) {
        root.left = deleteNode(root.left, key); // recurse on left subtree
    } else if (key > root.val) {
        root.right = deleteNode(root.right, key); // recurse on right subtree
    } else {
        // you've found the key!
        if (!root.left) {
            return root.right; // the right subtree will be shifted up a level if there's no left subtree
        } else if (!root.right) {
            return root.left; // the left subtree will be shifted up a level if there's no right subtree
        } else {
            // if the key node has a right and left node, we do the following:
            // 1) find the min-value node on the right subtree
            // 2) set the left value on the right min node to the key's left subtree
            // 3) return the key's updated right subtree (which will be shifted up a level)
            let rightMin = root.right;
            while (rightMin.left) rightMin = rightMin.left; // add findMin function
            rightMin.left = root.left;
            return root.right;
        }
    }
    return root;
}
```

PROJECT 3 -1

```
class Graph {
  constructor() {
    this.adjList = {};
  }
  addVertex(vertex) {
    if (!this.adjList[vertex]) {
      this.adjList[vertex] = [];
    }
  }
  addEdges(srcValue, destValue) {
    this.addVertex(srcValue);
    this.addVertex(destValue);
    this.adjList[srcValue].push(destValue);
    this.adjList[destValue].push(srcValue);
  }
  buildGraph(edges) {
    edges.forEach((edge) => {
      const srcValue = edge[0];
      const destValue = edge[1];
      this.addEdges(srcValue, destValue);
    });
    return this.adjList;
  }
```

```
  breadthFirstTraversal(startingVertex) {
    // Kevin Bacon
    let foundNodes = [startingVertex];
    let i = 0;
    while (i < foundNodes.length) {
      const thisNode = foundNodes[i];
      i = i + 1;
      let neighbors = this.adjList[thisNode];
      neighbors = neighbors.filter((neighbor) => {
        return !foundNodes.includes(neighbor);
      });
      foundNodes = [...foundNodes, ...neighbors];
    }
    return foundNodes;
  }
```

```
  depthFirstTraversalIterative(startingVertex) {
    const visited = new Set();
    const vertices = [];
    const stack = [startingVertex];
    while (stack.length) {
      let currentVertex = stack.pop();
      if (visited.has(currentVertex)) continue;
      visited.add(currentVertex);
      vertices.push(currentVertex);
      stack.push(...this.adjList[currentVertex]);
    }
    return vertices;
  }
  depthFirstTraversalRecursive(
    startingVertex,
    visited = new Set(),
    vertices = []
  ) {
    if (visited.has(startingVertex)) return;
    visited.add(startingVertex);
    vertices.push(startingVertex);
    for (let neighbor of this.adjList[startingVertex]) {
      this.depthFirstTraversalRecursive(neighbor, visited, vertices);
    }
    return vertices;
  }
}
```

```
describe("Graph Implementation", () => {
  describe("#constructor()", () => {
    it("should initialize an `adjList` to an empty object", () => {
      let graph = new Graph();
      expect(graph).to.have.property("adjList");
      expect(graph.adjList).to.eql({});

  describe("#addVertex(vertex)", () => {
    context("when the vertex is not in the graph", () => {
      it("should initialize a value of a new vertex as an empty array", () => {
        let graph = new Graph();
        graph.addVertex("Kevin Bacon");
        expect(graph.adjList).to.eql({ "Kevin Bacon": [] });
      });
    });
context("when the vertex is already in the graph", () => {
    it("should not add the vertex", () => {
      let graph = new Graph();
      graph.addVertex("James McAvoy");
      graph.addVertex("Kevin Bacon");
      graph.adjList["James McAvoy"] = ["Kevin Bacon"];
      graph.adjList["Kevin Bacon"] = ["James McAvoy"];
      graph.addVertex("James McAvoy"); // second entry
      expect(graph.adjList).to.eql({
        "James McAvoy": ["Kevin Bacon"],
        "Kevin Bacon": ["James McAvoy"],
      });
describe("#addEdges(edge1, edge2)", () => {
  context("when the graph does not include the edges", () => {
    it("should add the edges into the graph first", () => {
      let graph = new Graph();
      graph.addEdges("Kevin Bacon", "James McAvoy");
      expect(graph.adjList).to.eql({
        "Kevin Bacon": ["James McAvoy"],
        "James McAvoy": ["Kevin Bacon"],
      });
      graph.addEdges("Kevin Bacon", "Edward Asner");
      expect(graph.adjList).to.eql({
        "Kevin Bacon": ["James McAvoy", "Edward Asner"],
        "James McAvoy": ["Kevin Bacon"],
        "Edward Asner": ["Kevin Bacon"],
      });
```

WHITE BOARDING PROBLEMS 1:

```
## Problem 1: Breadth First Search on a Graph

given the adjacency list below, how many friends would Joe visit if he were
trying to get to Jesse using Breadth-First Traversal?

NOTE: your function should return the number of friends visited, not including Joe himself

```javascript
const adjacencyList = {
  'derek':['selam', 'dean'],
  'joe':['selam'],
  'selam': ['derek', 'joe', 'dean', 'evan'],
  'dean': ['derek', 'evan', 'selam'],
  'sam': ['jen'],
  'evan': ['selam', 'jesse', 'dean'],
  'jen':['sam', 'javier'],
  'javier':['jen'],
  'chris':[],
  'jesse': ['evan'],
};

function joesFriendsBFS(adjacencyList, startName, endName) {
  // We're doing BFS so we know we need a QUEUE
  let queue = [startName];
  // Track our visited friends.
  let visited = new Set();
  // While there are friends left to visit
  while (queue.length) {
    // We remove a friend from the queue
    let name = queue.shift();
    // Check if we've already visited them. if we have, we move on
    if (visited.has(name)) continue;
    // If we haven't, we add them to the visited Set
    visited.add(name);
    // Have we reached Jesse yet?
    if (name === endName) {
      // if so, we know we're done and we can return the number of friends we've
      // visited, which is the length of the visited set (minus Joe)
      return Array.from(visited).length - 1
    }
    // Otherwise let's add all the friends at our current friend, and run this again
    queue.push(...adjacencyList[name]);
  }
  // if our whole loop happened and we didn't get to Jesse, it means he's not a
  // part of the graph or there's no path between Joe and Jesse
  return "Jesse's not here :(";
}
```

Problem 1: Breadth First Search on a Graph

given the adjacency list below, how many friends would Joe visit if he were trying to get to Jesse using Breadth-First Traversal?

NOTE: your function should return the number of friends visited, not including Joe himself

```
const adjacencyList = {
  'derek':['selam', 'dean'],
  'joe':['selam'],
  'selam': ['derek', 'joe', 'dean', 'evan'],
  'dean': ['derek', 'evan', 'selam'],
  'sam': ['jen'],
  'evan': ['selam', 'jesse', 'dean'],
  'jen':['sam', 'javier'],
  'javier':['jen'],
  'chris':[],
  'jesse': ['evan'],
};
```
```

WHITE BOARDING PROBLEMS 2:

```
// PROBLEM 2

function joesFriendsDFS(adjacencyList, startName, endName) {
    // we're doing DFS so we know we need a STACK
    let stack = [startName];
    // Track visited
    let visited = new Set();
    // While there are friends to search through, we're going to contine
    while (stack.length) {
        // We remove the current friend we're comparing to Jesse
        let name = stack.pop();
        // Check if we've already visited them. if we have, we move on.
        if (visited.has(name)) continue;
        // if we haven't, add them to the visited list
        visited.add(name);
        // check if we have reached Jesse
        if (name === endName) {
            // If we have, we can return the list of friends we visited along the way
            const friendsList = Array.from(visited);
            // but we must first remove Joe from the list. we know he's first because
            // he was the first person popped off the stack.
            friendsList.shift();

            return friendsList;
        }

// if we didn't see Jesse, we add the friends to the top of the stack and keep searchi
        stack.push(...adjacencyList[name]);
    }
    // if our whole loop happened and we didn't get to Jesse, it means he's not a
    // part of the graph or there's no path between Joe and Jesse
    return "Jesse's not here :(";
}
```

Problem 2: Depth First Search on a Graph

Given the adjacency list below, which friends would Joe visit if he were trying to get to Jesse using Depth-First Traversal?

NOTE: your function should return a list of friends visited, not including Joe himself.

```
const adjacencyList = {
  'derek':['selam', 'dean'],
  'joe':['selam'],
  'selam': ['derek', 'joe', 'dean', 'evan'],
  'dean': ['derek', 'evan', 'selam'],
  'sam': ['jen'],
  'evan': ['selam', 'jesse', 'dean'],
  'jen':['sam', 'javier'],
  'javier':['jen'],
  'chris':[],
  'jesse': ['evan'],
};
```

WHITE BOARDING PROBLEMS 2:

```
// PROBLEM 2

function joesFriendsDFS(adjacencyList, startName, endName) {
    // we're doing DFS so we know we need a STACK
    let stack = [startName];
    // Track visited
    let visited = new Set();
    // While there are friends to search through, we're going to contine
    while (stack.length) {
        // We remove the current friend we're comparing to Jesse
        let name = stack.pop();
        // Check if we've already visited them. if we have, we move on.
        if (visited.has(name)) continue;
        // if we haven't, add them to the visited list
        visited.add(name);
        // check if we have reached Jesse
        if (name === endName) {
            // If we have, we can return the list of friends we visited along the way
            const friendsList = Array.from(visited);
            // but we must first remove Joe from the list. we know he's first because
            // he was the first person popped off the stack.
            friendsList.shift();

            return friendsList;
        }

// if we didn't see Jesse, we add the friends to the top of the stack and keep searchi
        stack.push(...adjacencyList[name]);
    }
    // if our whole loop happened and we didn't get to Jesse, it means he's not a
    // part of the graph or there's no path between Joe and Jesse
    return "Jesse's not here :(";
}
```

Problem 2: Depth First Search on a Graph

Given the adjacency list below, which friends would Joe visit if he were trying to get to Jesse using Depth-First Traversal?

NOTE: your function should return a list of friends visited, not including Joe himself.

```
const adjacencyList = {
  'derek':['selam', 'dean'],
  'joe':['selam'],
  'selam': ['derek', 'joe', 'dean', 'evan'],
  'dean': ['derek', 'evan', 'selam'],
  'sam': ['jen'],
  'evan': ['selam', 'jesse', 'dean'],
  'jen':['sam', 'javier'],
  'javier':['jen'],
  'chris':[],
  'jesse': ['evan'],
};
```

WHITE BOARDING PROBLEMS 3:

```
const hasPathSum = function (root, sum) {
    // If there's no root, we know that it's impossible to reach the sum
    if (!root) return false;

// we must keep track of our current value, which is the difference between our final sum and
    Let currSum = sum - root.val;
    // our base case. if we reach the bottom of the tree, and the difference between
    // our final value and our current value is zero, we know we've found the sum along this
    // so we can return true.
    if (!root.left && !root.right) {
        if (currSum == 0) return true;
    }
    // Now that we've established our base case and recursive step, we call the recursion
    Let leftSum = hasPathSum(root.left, currSum);
    Let rightSum = hasPathSum(root.right, currSum);
    // we want these functions to return a number, and we want to check both directions
    return leftSum || rightSum;
};

const pathSum = (root, sum, curSum = root.val) => {
    if (!root) return false;
    if (sum === curSum) return true;

    const leftRecur = pathSum(root.left, sum, curSum + root.val);
    const rightRecur = pathSum(root.right, sum, curSum + root.val);
    console.log(leftRecur || rightRecur);
    return leftRecur || rightRecur;
};
```

Problem 3: Path Sum of Binary Tree

Given the binary tree below and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

Use the following test-case: pathSum(5, 22) where 5 is the root node and 22 is the sum.
Note: this function should return a boolean value indicating whether or not the sum is possible to achieve.

```
//        5
//       / \
//      4   8
//     /   / \
//    11  13  4
//   /  \      \
// 7    2      1
```

WHITE BOARDING PROBLEMS 4:

Problem 4: Max Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Note: the depth at the root is 0.

```
//    3
//   / \
// 9  20
//    /  \
//   15   7
```

```
// PROBLEM 4
// DFS Recursive
function maxDepth(root) {
    // Our base case. we've reached the bottom of the tree.
    // Also stops us from running this on an empty tree
    if (!root) return -1;
    // we know this function returns a number, so we just add one to that number
    // to count the 'depth' of our tree.
    // we do this to the left and right of each tree node because we want to ensure
    // we are finding the maximum depth
    const leftHeight = 1 + maxDepth(root.left);
    const rightHeight = 1 + maxDepth(root.right);
    // our function returns the larger number: whichever one has the larger depth.
    return Math.max(leftHeight, rightHeight);
}
```

PlayGround 2

```
class _Node {
    constructor(value) {
        this.value = value;
        this.left = null;
        this.right = null;
        this.parent = null;
    }
}
```

```
class BST {
    constructor() {
        this.root = null;
    }
    insert(value) {
        if (this.root) {
            this._insert(this.root, value);
        } else {
            this.root = new _Node(value);
        }
    }
    _insert(root, value) {
        if (value < root.value) {
            if (root.left) {
                this._insert(root.left, value);
            } else {
                root.left = new _Node(value);
                root.left.parent = root;
            }
        } else {
            if (root.right) {
                this._insert(root.right, value);
            } else {
                root.right = new _Node(value);
                root.right.parent = root;
            }
        }
    }
    find(value) {
        if (this.root) {
            return this._find(this.root, value);
        }
    }
    _find(root, value) {
        if (root.value === value) {
            return root;
        } else if (value < root.value && root.left) {
            return this._find(root.left, value);
        } else if (value >= root.value && root.right) {
            return this._find(root.right, value);
        }
        return `not here fam`;
    }
    getHeight() {
        // gets the depth of tree
    }
    validBST() {
        // is it balanced?
    }
    buildTree() {
        // performs the balancing
    }
    delete(value) {
        // deletes node
        // 1. leaf node
        // 2. you have one child
        // 3. you have two children
    }
}
```

```
    bfsPrint() {
        if (this.root) {
            return this._bfsPrint();
        } else {
            return `your tree is empty fam`;
        }
    }
    _bfsPrint() {
        const nodes = [this.root]; // [root.left, root,right]
        const li = new Array();
        while (nodes.length) {
            const node = nodes.shift();
            li.push(node.value);
            if (node.left) nodes.push(node.left);
            if (node.right) nodes.push(node.right);
        }
        return li;
    }
    dfsPreorderPrint() {
        if (this.root) {
            return this._dfsPreorderPrint(this.root);
        } else {
            return `your tree is empty fam`;
        }
    }

    _dfsPreorderPrint(root, li = new Array()) {
        li.push(root.value);
        if (root.left) this._dfsPreorderPrint(root.left, li);
        if (root.right) this._dfsPreorderPrint(root.right, li);
        return li;
    }
    dfsInorderPrint() {
        if (this.root) {
            return this._dfsInorderPrint(this.root);
        } else {
            return `your tree is empty fam`;
        }
    }
    _dfsInorderPrint(root, li = new Array()) {
        if (root.left) this._dfsInorderPrint(root.left, li);
        li.push(root.value);
        if (root.right) this._dfsInorderPrint(root.right, li);
        return li;
    }
    dfsPostorderPrint() {
        if (this.root) {
            return this._dfsPostorderPrint(this.root);
        } else {
            return `your tree is empty fam`;
        }
    }
    _dfsPostorderPrint(root, li = new Array()) {
        if (root.left) this._dfsPostorderPrint(root.left, li);
        if (root.right) this._dfsPostorderPrint(root.right, li);
        li.push(root.value);
        return li;
    }
```

```
const tree = new BST();
tree.insert(6);
tree.insert(2);
tree.insert(7);
tree.insert(1);
tree.insert(4);
tree.insert(3);
tree.insert(5);
tree.insert(9);
tree.insert(8);
console.log(tree.bfsPrint());
console.log(tree.dfsPreorderPrint());
console.log(tree.dfsInorderPrint());
console.log(tree.dfsPostorderPrint());
console.log(tree.find(10));
console.log(tree.find(6));
console.log(tree.find(8));
```

```
╗ ~ Code-Snippets : (master) node sept-study-guide.js
[ 6, 2, 7, 1, 4, 9, 3, 5, 8 ]
[ 6, 2, 1, 4, 3, 5, 7, 9, 8 ]
[ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
[ 1, 3, 5, 4, 2, 8, 9, 7, 6 ]
not here fam
_Node {
  value: 6,
  left:
   _Node {
     value: 2,
     left:
      _Node { value: 1, left: null, right: null, parent: [Circular] },
     right:
      _Node { value: 4, left: [_Node], right: [_Node], parent: [Circular] },
     parent: [Circular] },
  right:
   _Node {
     value: 7,
     left: null,
     right:
      _Node { value: 9, left: [_Node], right: null, parent: [Circular] },
     parent: [Circular] },
  parent: null }
_Node {
  value: 8,
  left: null,
  right: null,
  parent:
   _Node {
     value: 9,
     left: [Circular],
     right: null,
     parent:
      _Node { value: 7, left: null, right: [Circular], parent: [_Node] } } }
╗ ~ Code-Snippets : (master)
```

FRIENDS-OF

```
/**
 * In this file, you will implement the friendsOf function that will calculate
 * the group of friends that a person has a certain distance from them.
 *
 * @param {Object} adjacencyList - The adjacency list that describes the graph,
 * never null or undefined
 * @param {string} name - The name of the person from where you will start your
 * search, never null or undefined
 * @param {number} distance - The distance away that you will traverse to find
 * the person's friends-of list, always a value greater than or equal to 1
 *
 * You will also need to implement a friendsOfRecursion function that will
 * recurse through your friends graph. friendsOf will call this.
 *
 * @param {string} name - the name of the person from where you will start
 * your search, never null or undefined.
 * @param {Object} adjacencyList - The adjacency list that describes the graph,
 * never null or undefined
 * @param {Set} visited - A list of all the graph nodes we have visited
 * @param {number} maxDistance - the maximum distance you want to recurse into
 * the graph, for example 1 should find immediate friends and 3 should find
 * immediate friends, friends of immediate friends, and friends of those friends
 * @param {number} currentDistance - the current distance we are at during our
 * recursion
 *
 * Hint: You can convert a Set into an Array using the `Array.from()` method.
 * https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/from
 *
 * Hint: refer to the documentation of Set on MDN here:
 * https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set
 */
function friendsOfRecursion( name, adjacencyList, visited,maxDistance,currentDistance) {
    // iterate until max distance is reached
    if (currentDistance  >  maxDistance) return;
    // add name to our visted set
    visited.add(name);
    // loop through the friend list @ adjacent list keyed into the name
    for (let friend of adjacencyList[name]) {
        // increment count to reach max distance for base case
        friendsOfRecursion(friend,adjacencyList,visited,maxDistance,currentDistance + 1);
  }
}
function friendsOf(adjacencyList, name, distance) {
    // keying in the name in adjacent list
    // if name is not or adjacency list is empty>> return undefined
    if (name in adjacencyList) {
        //init new set to store friends of var name
        let visited = new Set();
        // looping through friends of name var and passing to recusive helper func
        for (let currentFriend of adjacencyList[name]) {
            friendsOfRecursion(currentFriend,adjacencyList,visited,distance,0);
        }
        // delete var name from friend list
        visited.delete(name);
        // return the friend list as an array derived from the visited set object
        return Array.from(visited);
  }
  //
}
//
```