Binary Trees and Binary Search Trees

1. Explain and implement a Binary Tree

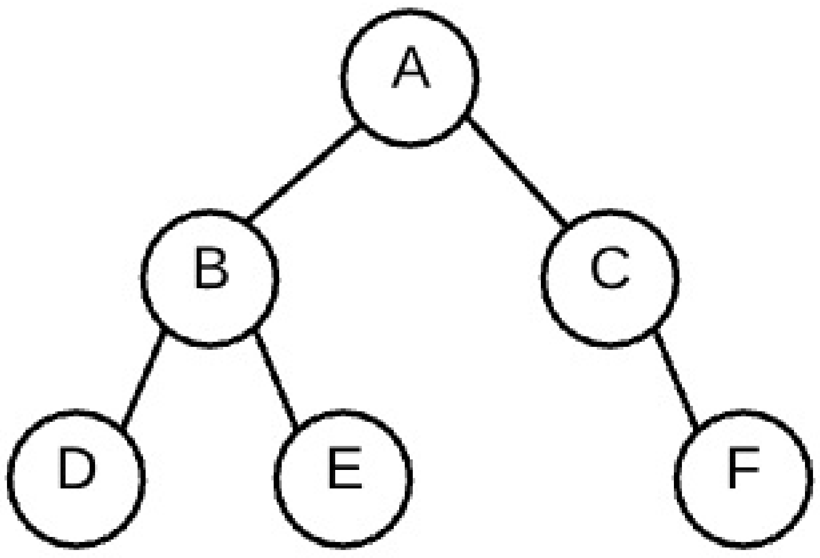
A Binary Tree is a Tree where nodes have at most 2 children, usually we represent these as 'left' and 'right'.

```
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;
```

Creates the following Binary Tree



Here's a complete implementation of a binary search tree

```
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;
    }
}

module.exports = {
    TreeNode,
    BST
};
```

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

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

Pre-order

1. Access the data of the current node
2. Recursively visit the left sub tree

3. Recursively visit the right sub tree

In-Order

1. Recursively visit the left sub tree
2. Access the data of the current node
3. Recursively visit the right sub tree

Post-Order

1. Recursively visit the left sub tree
2. Recursively visit the right sub tree
3. Access the data of the current node

3. 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

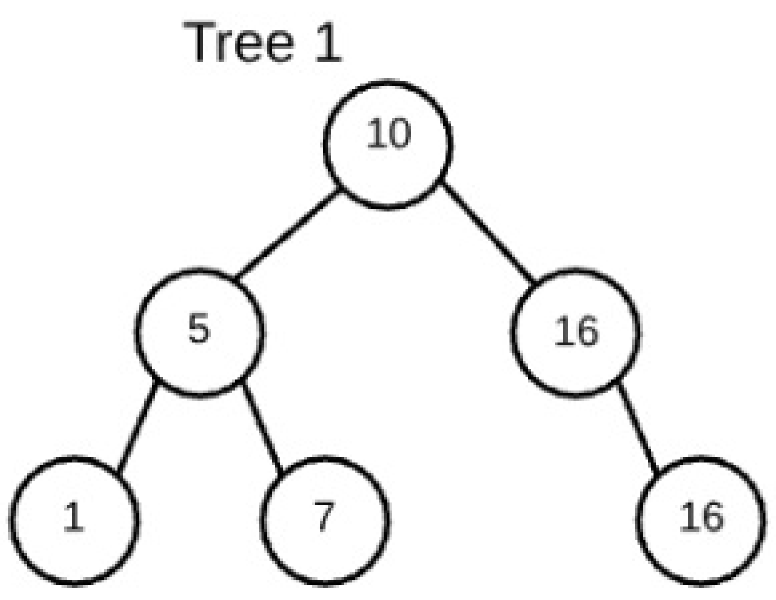
Some definitions of binary search trees allow duplicates and some do not. We can write our code to deal with the duplicates though.

Example of a Binary Search Tree:

```
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;
```

Generates this tree:



Usually we will write a class for our BST so we can have an "insert" method.
This way we can control the order of how we insert the nodes to make sure the tree is a binary search tree and also a balanced BST.

Graphs

1. Explain and implement a Graph

A graph is a collection of nodes and any edges between those nodes. It is a broad category. Linked lists and trees are both subclasses of graphs.

We can build a graph out of objects by making a GraphNode class

```
// Class based 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');

a.neighbors = [b];
b.neighbors = [c];
c.neighbors = [a];
```

Adjacency Lists are another way to make a graph

```
// adjacency list (Non-class based)
let graph = {
    'a': ['b'],
    'b': ['c'],
    'c': ['a']
}
```

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.

Extra Challenge, use the Queue class we made last week as the queue to use inside of this function, instead of just using an array as a queue

```
function breadthFirstSearch(graph, startingNode, targetVal) {
    // Populate our queue with the starting Node
    let queue = [ startingNode ];
    // Create a new empty Set to hold the nodes we've visited
    let visited = new Set();

    //  Keep going until the queue is empty
    while(queue.length) {
        // Dequeue the first thing from the queue
        let node = queue.shift();

        // If we've visited this node before, then just continue, which goes
        // back up to the while loop
        if(visited.has(node)) continue;
        // Add the node to the visited set.
        visited.add(node);

        // Check to see if the node is the one we are looking for, if it is
        // return true
        if (node === targetVal) return true;
        // Enqueue the node's neighbors from the adjacency graph onto the queue
        queue.push(...graph[node]);
    }
    // If we made it through the loop without finding one, return false
    return false;
}

breadthFirstSearch(graph, 'a', 'c'); // true
```

Depth-first Search on the adjacency list graph.

It is easier to do a recursive solution for depth first.
Depth first usually uses a stack, in this case the recursive call-stack is acting as our stack.

```
function depthFirstSearch(graph, startingNode, targetVal, visited=new Set()) {
    // If we found the node, return true
    if (startingNode === targetVal) {
        return true;
    }

    let neighbors = graph[startingNode];
    for (let neighbor of neighbors) {
        // If the neighbor has already been visited, we can
        // skip it.
        if (visited.has(neighbor)) continue;

        // Add the neighbor to the visited set
        visited.add(neighbor);

        // Now we recurse to check the neighbor and return the result
        return depthFirstSearch(graph, neighbor, targetVal, visited);
    }
    // If we didn't find it, return true
    return false;
}

console.log(depthFirstSearch(graph, 'a', 'c'));
```

Network Models Objectives

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

OSI is a reference model and doesn't match up very well to how things actually work in the real world. It has seven layers. Some descriptions of the OSI model try to fit our existing tech into the seven layer model but it doesn't match up exactly with how networks work today.

OSI Network Model

1. Application - HTTP
2. Presentation - JPEG/GIF
3. Session - RPC
4. Transport - TCP/UDP
5. Network - IP
6. Data Link - Ethernet
7. Physical - DSL, 802.11

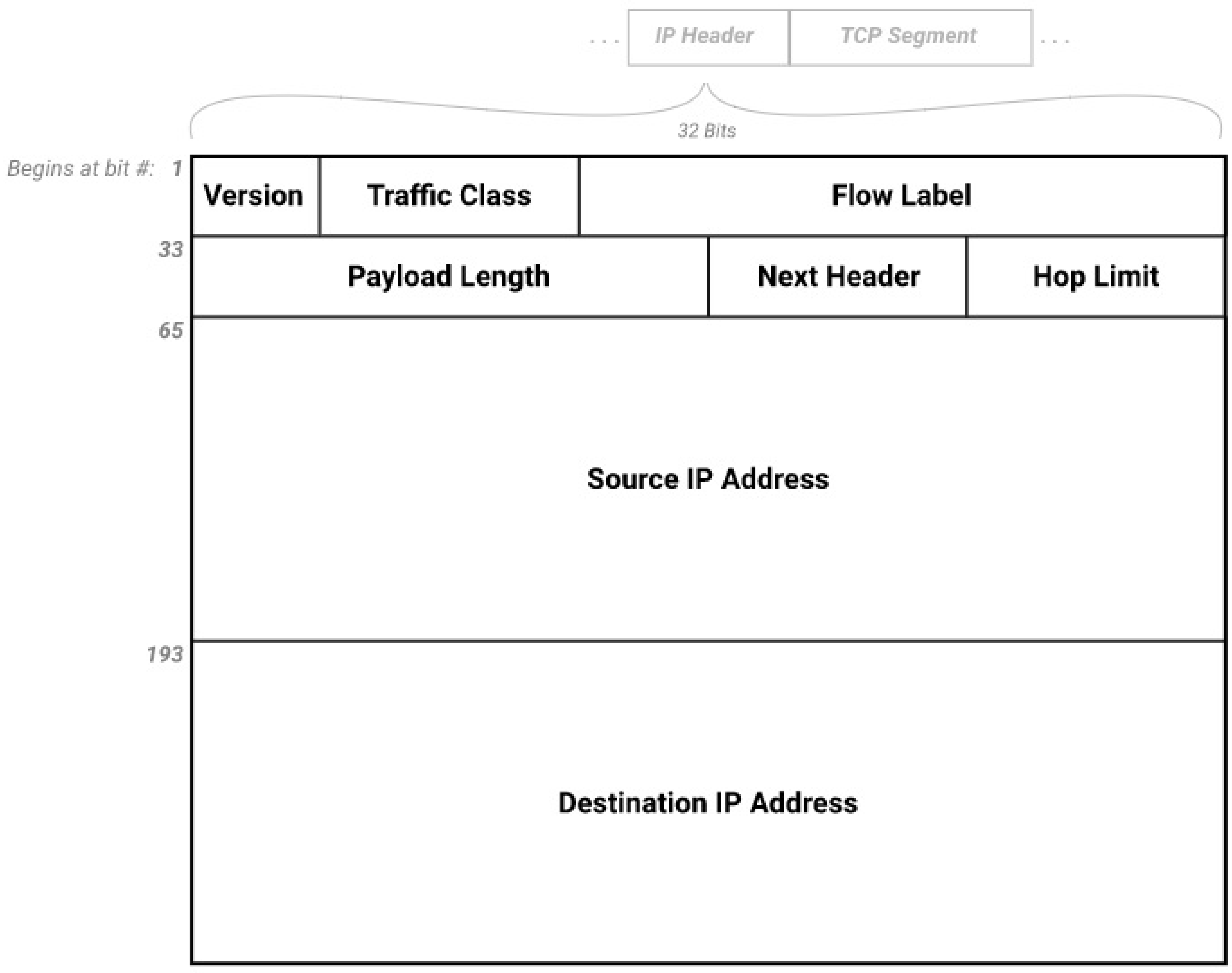
The TCP/IP model is the actual way networks today work, it is simpler than the OSI model and has only four layers.

TCP/IP Network Model

1. Application - HTTP, HTTPS, FTP, SMTP, etc
2. Transport - TCP or UDP
3. Internet - IP
4. Link - Ethernet

Internet Protocol Suite Objectives

1. Identify the correct fields of an IPv6 header



2. Distinguish an IPv4 packet from an IPv6

The Version number is stored in the headers of IP packets as a binary number.

```
IPv4 = 4 = 0100
IPv6 = 6 = 0110
```

IPv4 addresses are made up of 4 octets, each a 8-bit binary number converted to decimal.

Example: `192.168.1.1`

IPv6 addresses are made up of a 128bit number. It is usually resprentened in hexidecimal, with every four digits separated by a `:`

Example: `2001:0db8:85a3:0000:0000:8a2e:0370:7334`

In IPv6 addresses you can also compress the zeros to make it shorter to write:

The rules are:

- An entire string of zeros can be removed, you can only do this once.
- 4 zeros can be removed, leaving only a single zero.
- Leading zeros can be removed.

Example: `2001:db8:85a3::0:8a2e:370:7334`

3. Describe the following subjects and how they relate to one another: IP Addresses, Domain Names, and DNS

- *IP Address* - The internet protocol address assigned to a particular networking device (Ethernet adapter, Wi-Fi Adapter, etc). IPv4 example: `192.168.1.1`.
- *Domain Name* - A human readable name assigned to an IP address. Examples: `google.com` or `appacademy.io`
- *DNS* - Domain Name System: A protocol (on UDP port 53) that allows our computer to talk to a DNS Server and *resolve* a Domain Name into an IP Address. Example: `google.com` might resolve to `172.217.6.142`

Common DNS Record Types are:

- A - Directly maps a domain name to an IPv4 Address
- AAAA - Directly maps a domain name to an IPv6 Address
- CNAME - Maps a domain name to another domain name
- MX - Defines the mail server for a domain
- NS - Defines the DNS Servers for a zone (domain)
- SOA - Defines which DNS Server is the authority for a zone(domain)

4. Identify use cases for the TCP and UDP protocols

TCP - Transmission Control Protocol

TCP is used when you want reliable connections and you want the packets to reach the destination in the correct order. Web Browsing, Downloading Files, fetching Email from a server, Streaming Music or Video are all examples of TCP

UDP - user Datagram Protocol

UDP is used when you don't mind an more unreliable connection, but where real time interactivity is more important. If you drop a few packets, no big deal. Examples are Voice Over IP Telephone calls, and Video Chat systems like Facetime or Zoom.

5. Describe the following subjects and how they relate to one another: MAC Address, IP Address, and a port

- *MAC Address* - A hardware address assigned to every physical networking device on a network. These are assigned usually at the time the device was manufacturered, although in some case they can be changed via software. They look like a series of Hexidecimal values separated by `:` characters. Example: `ea:de:36:d9:5a:b8`. They are used to communicate on the local network *only*.
- *IP Address* - An address assigned to a networking device. These are usually assigned in software, and may be automatically assigned by an ISP or by a router on a local network using DHCP. They are used to `route` connections across multiple networks. Example: IPV4 Address `192.168.1.1`. A Router will map IP Addresses to MAC Addresses to keep track of connections.
- *Port* - Represents a TCP/UDP connection on an actual computer. Valid ports are numbers in the range from 0-65535. Used by the operating system of a computer to route TCP connections to the right program running on a computer. These programs can be said to be "listening" on a port. No two programs are allowed to listen on the same port at once. The default ports for web servers are HTTP(80) and HTTPS(443).

6. Identify the fields of a TCP segment

This was covered in the optional lectures and therefore isn't on the assessment

7. Describe how a TCP connection is negotiated

This was covered in the optional lectures and therefore isn't on the assessment

8. Explaining the difference between network devices like a router and a switch

1. *Hub* - A device which hooks multiple computers together over ethernet and blindly repeats ethernet packets to all the other devices on a local area network. These are not used much anymore
2. *Switch* - A device which intelligently hooks multiple computers together over ethernet and sends ethernet packets to the correct devices on a local area network based on MAC Addresses.
3. *Router* - A device which is reponsible for routing IP packets BETWEEN different networks.