Data Structure and Algorithm

Data Structures

Data Structures are specialized formats for organizing, processing, retrieving, and storing data. They are fundamental to efficient algorithm design and problem-solving in computer science.

Types of Data Structures

1. **Linear Data Structures**: Data elements are arranged sequentially or linearly, where each member element is connected to its previous and next element.
    - **Arrays**: Collection of elements identified by index or key.
    - **Linked Lists**: Series of connected nodes where each node contains data and a reference to the next node.
    - **Stacks**: Linear structure which follows LIFO (Last In First Out) principle.
    - **Queues**: Linear structure which follows FIFO (First In First Out) principle.
2. **Non-Linear Data Structures**: Data elements are arranged in a hierarchical or interconnected manner.
    - **Trees**: Hierarchical structure with a root node and child nodes, where each node has at most one parent.
        - **Binary Trees**: Each node has at most two children.
        - **Binary Search Trees**: Binary tree with sorted order properties.
    - **Graphs**: Consist of nodes (vertices) and edges connecting them.
        - **Directed Graphs**: Edges have a direction.
        - **Undirected Graphs**: Edges do not have a direction.
3. **Hash Tables**: Key-value pairs are stored in a hash table and provide efficient data retrieval through hashing.

Basics of Memory Allocation and Memory Leak in JavaScript

Memory management is a crucial aspect of programming, and understanding how it works can help you write more efficient and bug-free code. In JavaScript, memory allocation and memory leaks are important concepts to grasp.

Memory Allocation

Memory allocation in JavaScript happens in two main areas: the stack and the heap.

Stack: Used for static memory allocation. This is where primitive values (numbers, strings, booleans, null, undefined, and symbols) are stored. Function calls and their local variables are also stored in the stack.

Heap: Used for dynamic memory allocation. This is where objects and functions are stored. Whenever a new object or function is created, it is allocated memory in the heap.

Example of Memory Allocation

```
function example() {
    // Stack allocation
    let a = 10;
    let b = "Hello";

    // Heap allocation
    let obj = { name: "Alice", age: 25 };
    let arr = [1, 2, 3, 4];
}

example();
```

In the above example:

- `a` and `b` are primitive values and are allocated memory on the stack.
- `obj` and `arr` are complex objects and arrays, and they are allocated memory on the heap.

Garbage Collection

JavaScript has an automatic memory management system known as garbage collection. The garbage collector automatically frees up memory that is no longer in use. The most common algorithm used for garbage collection in JavaScript is Mark-and-Sweep.

Mark-and-Sweep Algorithm

1. **Mark Phase**: The garbage collector traverses all objects that are reachable and marks them.
2. **Sweep Phase**: The garbage collector removes all objects that are not marked, thus freeing up memory.

Memory Leak

A memory leak occurs when a program does not release memory that is no longer needed. This can lead to increased memory usage and eventually slow down or crash the program.

Common Causes of Memory Leaks in JavaScript

1. **Global Variables**: Variables declared without `var`, `let`, or `const` become global and are not garbage collected.
2. **Closures**: Improper use of closures can lead to memory leaks if references to variables are not properly managed.
3. **Event Listeners**: Unremoved event listeners can cause memory leaks because they keep references to the DOM elements.
4. **Timers**: Unstopped intervals or timeouts can prevent garbage collection.

Example of a Memory Leak

```
let leakedArray = [];

function memoryLeak() {
    // Every call adds a new element to the array without rel
```

```
easing the previous ones
    leakedArray.push("leak");
}

// Call the function repeatedly
setInterval(memoryLeak, 1000);
```

In this example, the `leakedArray` keeps growing as `memoryLeak` is called every second, leading to a memory leak because the array keeps holding references to the string "leak".

How to Prevent Memory Leaks

1. **Use `let`, `const`, or `var` for Variable Declarations**: Avoid creating global variables inadvertently.

   ```
   let x = 10; // Good
   y = 20;    // Bad (global variable)
   ```

2. **Manage Closures Carefully**: Ensure that closures do not unintentionally hold references to variables that are no longer needed.

   ```
   function createClosure() {
       let largeArray = new Array(1000).fill("data");
       return function() {
           console.log(largeArray);
       };
   }
   ```

3. **Remove Event Listeners**: Always remove event listeners when they are no longer needed.

```
let button = document.getElementById("myButton");

function handleClick() {
    console.log("Button clicked");
}

button.addEventListener("click", handleClick);

// Remove event listener when no longer needed
button.removeEventListener("click", handleClick);
```

4. **Clear Timers**: Clear intervals and timeouts when they are no longer needed.

```
javascriptCopy code
let intervalId = setInterval(() => {
    console.log("Interval running");
}, 1000);

// Clear interval when done
clearInterval(intervalId);
```

Monitoring Memory Usage

Modern browsers come with developer tools that help in monitoring memory usage and detecting memory leaks. For example, in Google Chrome:

1. Open Developer Tools (F12 or right-click → Inspect).
2. Go to the "Memory" tab.
3. Use the "Heap Snapshot" to take snapshots and analyze memory usage over time.

Understanding memory allocation and how to avoid memory leaks is crucial for building efficient and reliable JavaScript applications. By following best practices and using tools to monitor memory usage, you can prevent memory-related issues in your code.

Complexity Analysis

Complexity Analysis is the study of how the performance of an algorithm changes with the size of the input. There are two primary types of complexity we consider:

- **Time Complexity**: How the runtime of the algorithm scales with input size.
- **Space Complexity**: How the memory usage of the algorithm scales with input size.

Asymptotic Analysis (Big-O Notation)

Asymptotic Analysis is a way of describing the behavior of algorithms as the input size grows. Big-O Notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows.

Common Big-O Notations:

- **O(1)**: Constant time
- **O(log n)**: Logarithmic time
- **O(n)**: Linear time
- **O(n log n)**: Linearithmic time
- **O(n^2)**: Quadratic time
- **O(2^n)**: Exponential time
- **O(n!)**: Factorial time

Complexity of Common Operations for Data Structures

Let's explore the complexity of common operations for different data structures using JavaScript examples.

1. Arrays

- **Access**: O(1)

- **Search**: O(n)
- **Insertion (at end)**: O(1)
- **Insertion (at beginning or middle)**: O(n)
- **Deletion (at end)**: O(1)
- **Deletion (at beginning or middle)**: O(n)

```
let arr = [1, 2, 3, 4, 5];

// Access
console.log(arr[2]); // O(1)

// Search
console.log(arr.indexOf(3)); // O(n)

// Insertion (at end)
arr.push(6); // O(1)

// Insertion (at beginning)
arr.unshift(0); // O(n)

// Deletion (at end)
arr.pop(); // O(1)

// Deletion (at beginning)
arr.shift(); // O(n)
```

2. Linked Lists

- **Access**: O(n)
- **Search**: O(n)
- **Insertion**: O(1)

- **Deletion**: O(1)

```
class Node {
    constructor(value) {
        this.value = value;
        this.next = null;
    }
}

class LinkedList {
    constructor() {
        this.head = null;
    }

    // Insertion (at beginning)
    insertAtBeginning(value) {
        const newNode = new Node(value);
        newNode.next = this.head;
        this.head = newNode; // O(1)
    }

    // Search
    search(value) {
        let current = this.head;
        while (current) {
            if (current.value === value) return true;
            current = current.next;
        }
        return false; // O(n)
    }

    // Deletion (by value)
    delete(value) {
        if (!this.head) return;
```

```
        if (this.head.value === value) {
            this.head = this.head.next; // O(1)
            return;
        }

        let current = this.head;
        while (current.next && current.next.value !== value)
{
            current = current.next;
        }

        if (current.next) {
            current.next = current.next.next; // O(1)
        }
    }
}

let ll = new LinkedList();
ll.insertAtBeginning(1);
ll.insertAtBeginning(2);
ll.insertAtBeginning(3);

console.log(ll.search(2)); // O(n)
ll.delete(2); // O(1)
```

Linked List

A linked list is a data structure used in computer science to store a collection of elements, where each element (called a node) contains a reference (or link) to the next element in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations; instead, they use pointers to connect elements, allowing for efficient insertion and deletion operations.

Components of a Linked List

1. **Node**: Each node in a linked list contains two components:
    - **Data**: The value or data the node holds.
    - **Next**: A reference to the next node in the list.
2. **Head**: A reference to the first node in the linked list.
3. **Tail**: Optionally, a reference to the last node in the list (useful for certain operations).

Types of Linked Lists

1. **Singly Linked List**: Each node points to the next node. The last node points to `null`.
2. **Doubly Linked List**: Each node points to both the next and the previous node.
3. **Circular Linked List**: The last node points back to the first node, forming a circle.

Diagram of a Singly Linked List

```
Head
 |
 v
+------+   +------+   +------+
| Data |-->| Data |-->| Data |--> null
| Next |   | Next |   | Next |
+------+   +------+   +------+
```

Explanation

1. **Node Class**: Defines a node with `data` and `next` properties.
2. **LinkedList Class**: Manages the linked list with methods to add nodes and print the list.
    - `add(data)`: Creates a new node and adds it to the end of the list.

- `printList()`: Traverses the list and prints the data of each node.

Singly Linked List

In a singly linked list, each node points to the next node in the sequence.

```
// Node class to create nodes of the linked list
class Node {
    constructor(data) {
        this.data = data; // Data stored in the node
        this.next = null; // Reference to the next node
    }
}

// SinglyLinkedList class to handle operations on the linked
list
class SinglyLinkedList {
    constructor() {
        this.head = null; // Points to the head of the linked
list
    }

    // Method to insert a node at the end of the linked list
    insertAtEnd(data) {
        const newNode = new Node(data);
        if (!this.head) {
            this.head = newNode;
        } else {
            let current = this.head;
            while (current.next) {
                current = current.next;
            }
            current.next = newNode;
        }
    }
```

```
    // Method to display the linked list
    display() {
        let current = this.head;
        while (current) {
            console.log(current.data);
            current = current.next;
        }
    }
}

// Example usage:
const sll = new SinglyLinkedList();
sll.insertAtEnd(1);
sll.insertAtEnd(2);
sll.insertAtEnd(3);
sll.display(); // Output: 1 2 3
```

Doubly Linked List

In a doubly linked list, each node has references to both the next and the previous nodes.

```
// Node class for doubly linked list
class Node {
    constructor(data) {
        this.data = data; // Data stored in the node
        this.next = null; // Reference to the next node
        this.prev = null; // Reference to the previous node
    }
}

// DoublyLinkedList class to handle operations on the linked
list
class DoublyLinkedList {
    constructor() {
```

```
        this.head = null; // Points to the head of the linked
list
        this.tail = null; // Points to the tail of the linked
list
    }

    // Method to insert a node at the end of the linked list
    insertAtEnd(data) {
        const newNode = new Node(data);
        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            let current = this.tail;
                  current.next = newNode;
                  this.tail = newNode;
            }
    }

    // Method to display the linked list
    display() {
        let current = this.head;
        while (current) {
            console.log(current.data);
            current = current.next;
        }
    }
}

// Example usage:
const dll = new DoublyLinkedList();
dll.insertAtEnd(1);
dll.insertAtEnd(2);
dll.insertAtEnd(3);
dll.display(); // Output: 1 2 3
```

Convert array to a linked list

```
// Node class for singly linked list
class Node {
    constructor(data) {
        this.data = data; // Data stored in the node
        this.next = null; // Reference to the next node
    }
}

// Function to convert an array into a singly linked list and
display it
function arrayToLinkedList(arr) {
    if (!arr || arr.length === 0) {
        return null;
    }

    // Create the head of the linked list
    let head = new Node(arr[0]);
    let current = head;

    // Iterate through the array to create nodes and link the
m
    for (let i = 1; i < arr.length; i++) {
        const newNode = new Node(arr[i]);
        current.next = newNode;
        current = newNode;
    }

    // Display the linked list
    current = head;
    while (current !== null) {
        console.log(current.data);
        current = current.next;
    }
```

```
    return head;
}

// Example usage:
const arr = [1, 2, 3, 4, 5];
const linkedListHead = arrayToLinkedList(arr);
```

Adding a Node at the Beginning

```
// Define the Node class
class Node {
    constructor(data) {
        this.data = data;
        this.next = null;
    }
}

// Define the LinkedList class
class LinkedList {
    constructor() {
        this.head = null;
    }

    // Method to insert a new node at the beginning
    insertAtBeginning(data) {
        // Create a new node with the given value
        const newNode = new Node(data);

        // Make the new node point to the current head
        newNode.next = this.head;

        // Update the head to be the new node
        this.head = newNode;
    }
```

```
    // Method to display the linked list
    display() {
        let current = this.head;
        while (current !== null) {
            console.log(current.data);
            current = current.next;
        }
    }
}

// Example usage
const list = new LinkedList();
list.insertAtBeginning(3);
list.insertAtBeginning(2);
list.insertAtBeginning(1);

list.display(); // Output: 1 2 3
```

Deleting a node with a specified value in a singly linked list

```
// Node class for singly linked list
class Node {
    constructor(data) {
        this.data = data; // Data stored in the node
        this.next = null; // Reference to the next node
    }
}

// SinglyLinkedList class to handle operations on the linked
list
class SinglyLinkedList {
    constructor() {
        this.head = null; // Points to the head of the linked
list
```

```
    }

    // Method to insert a node at the end of the linked list
    insertAtEnd(data) {
        const newNode = new Node(data);
        if (!this.head) {
            this.head = newNode;
            return;
        }
        let current = this.head;
        while (current.next) {
            current = current.next;
        }
        current.next = newNode;
    }

    // Method to delete a node with a specified value from th
e linked list
    deleteNode(value) {
        if (!this.head) {
            return;
        }

        // If the node to be deleted is the head node
        if (this.head.data === value) {
            this.head = this.head.next;
            return;
        }

        let current = this.head;
        while (current.next) {
            if (current.next.data === value) {
                current.next = current.next.next;
                return;
            }
            current = current.next;
```

```
        }
    }

    // Method to display the linked list
    display() {
        let current = this.head;
        const result = [];
        while (current) {
            result.push(current.data);
            current = current.next;
        }
        console.log(result.join(' -> '));
    }
}

// Example usage:
const sll = new SinglyLinkedList();

sll.insertAtEnd(1);
sll.insertAtEnd(2);
sll.insertAtEnd(3);
sll.insertAtEnd(4);
sll.insertAtEnd(5);

sll.display(); // Output: 1 -> 2 -> 3 -> 4 -> 5

sll.deleteNode(3); // Delete node with value 3

sll.display(); // Output: 1 -> 2 -> 4 -> 5
```

Inserting a Node After a Node with X Data

```
// Node class for singly linked list
class Node {
```

```
    constructor(data) {
        this.data = data; // Data stored in the node
        this.next = null; // Reference to the next node
    }
}

// SinglyLinkedList class to handle operations on the linked
list
class SinglyLinkedList {
    constructor() {
        this.head = null; // Points to the head of the linked
list
    }

    // Method to insert a node at the end of the linked list
    insertAtEnd(data) {
        const newNode = new Node(data);
        if (!this.head) {
            this.head = newNode;
            return;
        }
        let current = this.head;
        while (current.next) {
            current = current.next;
        }
        current.next = newNode;
    }

    // Method to insert a node after a node with specific dat
a
    insertAfter(data, x) {
        const newNode = new Node(data);
        let current = this.head;
        while (current) {
            if (current.data === x) {
                newNode.next = current.next;
```

```
                current.next = newNode;
                return;
            }
            current = current.next;
        }
        console.log(`Node with data ${x} not found.`);
    }

    // Method to display the linked list
    display() {
        let current = this.head;
        const result = [];
        while (current) {
            result.push(current.data);
            current = current.next;
        }
        console.log(result.join(' -> '));
    }
}

// Example usage:
const sll = new SinglyLinkedList();

sll.insertAtEnd(1);
sll.insertAtEnd(2);
sll.insertAtEnd(3);
sll.insertAtEnd(5); // Inserting a node with data 5
sll.insertAtEnd(6);

sll.display(); // Output: 1 -> 2 -> 3 -> 5 -> 6

sll.insertAfter(4, 3); // Insert node with data 4 after node
with data 3
```

```
sll.display(); // Output: 1 -> 2 -> 3 -> 4 -> 5 -> 6
```

Inserting a Node Before a Node with X Data

To insert a node before a node with specific data, you typically need to keep track of the previous node while traversing the list.

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedList {
  constructor() {
    this.head = null;
  }

  insertAtBegin(data) {
    const newNode = new Node(data);
    newNode.next = this.head;
    this.head = newNode;
  }

  insertBeforeX(value, data) {
    const newNode = new Node(data);

    if (!this.head) {
      console.log("List is empty");
      return;
    }
```

```
    // If the node to insert before is the head node
    if (this.head.data === value) {
      newNode.next = this.head;
      this.head = newNode;
      return;
    }

    let current = this.head;
    while (current.next) {
      if (current.next.data === value) {
        newNode.next = current.next;
        current.next = newNode;
        return;
      }
      current = current.next;
    }

    console.log("Node with value " + value + " not found.");
  }

  display() {
    let current = this.head;
    while (current) {
      console.log(current.data);
      current = current.next;
    }
  }
}

const list = new LinkedList();
list.insertAtBegin(67);
list.insertAtBegin(23);
list.insertAtBegin(12);
list.insertAtBegin(13);

list.insertBeforeX(23, 56);
```

```
list.display();
```

Singly Linked List Implementation

```
// Node class for singly linked list
class Node {
    constructor(data) {
        this.data = data; // Data stored in the node
        this.next = null; // Reference to the next node
    }
}

// SinglyLinkedList class to handle operations on the linked
list
class SinglyLinkedList {
    constructor() {
        this.head = null; // Points to the head of the linked
list
    }

    // Method to insert a node at the end of the linked list
    insertAtEnd(data) {
        const newNode = new Node(data);
        if (!this.head) {
            this.head = newNode;
            return;
        }
        let current = this.head;
        while (current.next) {
            current = current.next;
        }
        current.next = newNode;
```

```
    }

    // Method to display the linked list in forward order
    displayForward() {
        let current = this.head;
        const result = [];
        while (current) {
            result.push(current.data);
            current = current.next;
        }
        console.log("Forward order:", result.join(' -> '));
    }

    // Method to display the linked list in reverse order
    displayReverse() {
        const stack = [];
        let current = this.head;
        while (current) {
            stack.push(current.data);
            current = current.next;
        }
        const result = [];
        while (stack.length > 0) {
            result.push(stack.pop());
        }
        console.log("Reverse order:", result.join(' -> '));
    }
}

// Example usage:
const sll = new SinglyLinkedList();

sll.insertAtEnd(1);
sll.insertAtEnd(2);
sll.insertAtEnd(3);
sll.insertAtEnd(4);
```

```
sll.insertAtEnd(5);

sll.displayForward(); // Output: Forward order: 1 -> 2 -> 3 -
> 4 -> 5
sll.displayReverse(); // Output: Reverse order: 5 -> 4 -> 3 -
> 2 -> 1
```

To remove duplicates from a sorted singly linked list

```
// Node class for singly linked list
class Node {
    constructor(data) {
        this.data = data; // Data stored in the node
        this.next = null; // Reference to the next node
    }
}

// SinglyLinkedList class to handle operations on the linked
list
class SinglyLinkedList {
    constructor() {
        this.head = null; // Points to the head of the linked
list
    }

    // Method to insert a node at the end of the linked list
    insertAtEnd(data) {
        const newNode = new Node(data);
        if (!this.head) {
            this.head = newNode;
            return;
        }
        let current = this.head;
```

```
        while (current.next) {
            current = current.next;
        }
        current.next = newNode;
    }

    // Method to remove duplicates from a sorted linked list
    removeDuplicates() {
        let current = this.head;
        while (current && current.next) {
            if (current.data === current.next.data) {
                current.next = current.next.next;
            } else {
                current = current.next;
            }
        }
    }

    // Method to display the linked list
    display() {
        let current = this.head;
        const result = [];
        while (current) {
            result.push(current.data);
            current = current.next;
        }
        console.log(result.join(' -> '));
    }
}

// Example usage:
const sll = new SinglyLinkedList();

sll.insertAtEnd(1);
sll.insertAtEnd(2);
sll.insertAtEnd(2);
```

```
sll.insertAtEnd(3);
sll.insertAtEnd(4);
sll.insertAtEnd(4);
sll.insertAtEnd(4);
sll.insertAtEnd(5);
sll.insertAtEnd(5);
sll.insertAtEnd(6);

sll.display(); // Output: 1 -> 2 -> 2 -> 3 -> 4 -> 4 -> 4 ->
5 -> 5 -> 6

sll.removeDuplicates();

sll.display(); // Output: 1 -> 2 -> 3 -> 4 -> 5 -> 6
```

Only the function of operations :

```
insertAtBeginning(data) {
        // Create a new node with the given value
        const newNode = new Node(data);

        // Make the new node point to the current head
        newNode.next = this.head;

        // Update the head to be the new node
        this.head = newNode;
    }


insertAtEnd(data) {
        const newNode = new Node(data);
        if (!this.head) {
            this.head = newNode;
            return;
```

```
        }
        let current = this.head;
        while (current.next) {
            current = current.next;
        }
        current.next = newNode;
    }


insertBeforeX(data, x) {
        const newNode = new Node(data);
        if (this.head && this.head.data === x) {
            newNode.next = this.head;
            this.head = newNode;
            return; // Exit after insertion
        }
        let current = this.head;
        let prev = null;
        while (current) {
            if (current.data === x) {
                newNode.next = current;
                if (prev) {
                    prev.next = newNode;
                }
                return; // Exit after insertion
            }
            prev = current;
            current = current.next;
        }
        console.log(`Node with data ${x} not found.`);
    }
}

 insertAfter(data, x) {
        const newNode = new Node(data);
        let current = this.head;
```

```
        while (current) {
            if (current.data === x) {
                newNode.next = current.next;
                current.next = newNode;
                return;
            }
            current = current.next;
        }
        console.log(`Node with data ${x} not found.`);
    }
```

Function to replace a character from a string

```
function ReplaceChar(str, char, n) {
let strChar = str.split('');
for (let i = 0; i < strChar.length; i++) {
    if (strChar[i] === char) {
        strChar[i] = n;
    }
}

console.log(strChar.join(''));
}

ReplaceChar('helloWorld', 'l', 'B');
```

Binary Search

```
function binarySearch(arr, target) {
    let left = 0;
    let right = arr.length - 1;
```

```
    for (; left <= right;) {
        const mid = Math.floor((left + right) / 2); // Always pu
        if (arr[mid] === target) {
            return console.log(mid);
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
}

// Example usage
const arr = [1, 2, 3, 4, 5, 6, 7, 28, 56, 78];
binarySearch(arr, 28); // Output: 7
```

Recursion is a programming technique where a function calls itself in order to solve a problem. The recursive function generally has two main parts:

- **Base case**: A condition that stops the recursion.
- **Recursive case**: The part of the function where the function calls itself with modified arguments.

Workout 1: Factorial Calculation

The factorial of a non-negative integer nnn is the product of all positive integers less than or equal to nnn. It's denoted as n!n!n!.

Factorial Function

```
function factorial(n) {
    // Base case
    if (n === 0 || n === 1) {
        return 1;
```

```
    }
    // Recursive case
    return n * factorial(n - 1);
}

// Example usage
console.log(factorial(5)); // Output: 120
console.log(factorial(0)); // Output: 1
```

Workout 2: Fibonacci Sequence

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1.

Fibonacci Function

```
function fibonacci(n) {
    // Base cases
    if (n === 0) {
        return 0;
    }
    if (n === 1) {
        return 1;
    }
    // Recursive case
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// Example usage
console.log(fibonacci(5)); // Output: 5
console.log(fibonacci(10)); // Output: 55
```

Workout 3: Sum of Array Elements

Calculate the sum of elements in an array using recursion.

Sum of Array Function

```
function sum(arr){
  if(!arr){
    return `No elements to sum`
  }
  if(arr.length==1){
    return arr[0];
  }
  return arr[0] + sum(arr.slice(1));
}

console.log(sum([1,2,3,4,5,6,6,7,8,10,9]))
```

Reverse a string using recursion

```
function reverseString(str) {
// Base case: If the string is empty or has one character
if (str.length <= 1) {
return str;
}
// Recursive case: Reverse the substring and append the first cl
return reverseString(str.slice(1)) + str[0];
}
```

```
// Example usage
console.log(reverseString("hello")); // Output: "olleh"
console.log(reverseString("recursion")); // Output: "noisrucer
```

Palindrome Function

```
function isPalindrome(str) {
    // Base case: If the string is empty or has one characte
r, it's a palindrome
    if (str.length <= 1) {
        return true;
    }
    // Check if the first and last characters are the same
    if (str[0] !== str[str.length - 1]) {
        return false;
    }
    // Recursive case: Check the substring without the first
and last characters
    return isPalindrome(str.slice(1, -1));
}

// Example usage
console.log(isPalindrome("racecar")); // Output: true
console.log(isPalindrome("hello"));   // Output: false
```

2. Recursive Operations on an Array

Here are two common array operations you can perform using recursion:

2.1. Calculating the Sum of Array Elements

```
function sum(arr){
  if(!arr){
    return "Array is empty"
  }
  if(arr.length===1) return arr[0];

  return sum(arr.slice(1))+arr[0];
}
```

```
console.log(sum([1,2,3,4,5,6,7,8,9,10]));
```

2.2. Finding the Maximum Value in an Array

```
function findMax(arr, n) {
    // Base case: If the array has only one element
    if (n === 1) {
        return arr[0];
    }
    // Recursive case: Find the maximum in the rest of the ar
ray and compare with the last element
    return Math.max(findMax(arr, n - 1), arr[n - 1]);
}

// Example usage
const numbers2 = [3, 5, 7, 2, 8];
console.log(findMax(numbers2, numbers2.length)); // Output: 8
```

Reverse Array Using Recursion

To reverse an array recursively, you can use the following approach:

1. **Base Case**: If the array is empty or has one element, it's already reversed.
2. **Recursive Case**: Swap the first and last elements, then recursively reverse the subarray that excludes these elements.

Here's a JavaScript function that accomplishes this:

```
function revArray(arr){
  if(arr.length===0){
    return [];
  }
  return [arr.pop()].concat(revArray(arr))

}
```

```
const arr =[1,2,3,4,5];
console.log(revArray(arr))
```

Palindrome Number Check Using Recursion

```
function isPalindrome(num){
  let str = num.toString();
  if(str.length <= 1){
    return 'isPalindrome';
  }
  if(str[0]!==str[str.length-1]){
    return 'notPalindrome';
  }
  return isPalindrome(str.slice(1,-1));
}
const num = 12231;
console.log(isPalindrome(num))
```

Finding the sum of N elements using recursion

```
function sumOfN(n){
if(n===0){
return 0;
}
return n+ sumOfN(n-1);
}
console.log(sumOfN(10));
```

Inserting a node at index

```
class Node {
  constructor(value) {
    this.value = value;
    this.next = null;
```

```
  }
}

class LinkedList {
  constructor() {
    this.head = null;
    this.size = 0;
  }

  insertAt(index, value) {
    if (index < 0 || index > this.size) {
      console.log("Index out of range");
      return;
    }

    const newNode = new Node(value);

    if (index === 0) {
      newNode.next = this.head;
      this.head = newNode;
    } else {
      let current = this.head;
      let previous;
      let count = 0;

      while (count < index) {
        previous = current;
        current = current.next;
        count++;
      }

      newNode.next = current;
      previous.next = newNode;
    }

    this.size++;
```

```
  }

  printList() {
    let current = this.head;
    while (current) {
      console.log(current.value);
      current = current.next;
    }
  }
}

// Example usage:
const ll = new LinkedList();
ll.insertAt(0, 10); // Insert at the beginning
ll.insertAt(1, 20); // Insert at the end
ll.insertAt(1, 15); // Insert at index 1

ll.printList();
// Output:
// 10
// 15
// 20
```

Advantages of Linked List Over Array

1. **Dynamic Size:**
   - **Linked List:** Can grow and shrink dynamically as elements are added or removed.
   - **Array:** Fixed size once declared (unless using dynamic arrays like in higher-level languages which involve resizing).
2. **Efficient Insertions/Deletions:**
   - **Linked List:** Inserting or deleting an element involves changing a few pointers and can be done in O(1) time if the position is known.

- **Array:** Inserting or deleting an element requires shifting elements, leading to O(n) time complexity.

3. **Memory Usage:**
   - **Linked List:** Memory is allocated as needed. No need to allocate memory for unused elements.
   - **Array:** Pre-allocates memory which can lead to wastage if many elements are unused.
4. **No Contiguous Memory Requirement:**
   - **Linked List:** Does not require contiguous memory allocation.
   - **Array:** Requires contiguous memory allocation which can be a limitation in systems with fragmented memory.

Reverse a Linked List

```
class Node {
  constructor(value) {
    this.value = value;
    this.next = null;
  }
}

class LinkedList {
  constructor() {
    this.head = null;
  }

  reverse() {
    let prev = null;
    let current = this.head;
    let next = null;

    while (current !== null) {
      next = current.next;
```

```
      current.next = prev;
      prev = current;
      current = next;
    }

    this.head = prev;
  }

  insert(value) {
    const newNode = new Node(value);
    newNode.next = this.head;
    this.head = newNode;
  }

  printList() {
    let current = this.head;
    while (current !== null) {
      console.log(current.value);
      current = current.next;
    }
  }
}

// Usage example
const list = new LinkedList();
list.insert(10);
list.insert(20);
list.insert(30);
list.printList(); // Output: 30, 20, 10
list.reverse();
list.printList(); // Output: 10, 20, 30
```

Advantages of Recursion

1. **Simpler Code:**

- Often leads to more readable and concise code for problems that have a recursive nature (e.g., tree traversal, factorial calculation).

2. **Divide and Conquer:**
   - Recursion naturally implements divide and conquer algorithms which break the problem into smaller subproblems (e.g., merge sort, quicksort).
3. **State Management:**
   - Uses the call stack to manage state across recursive calls, eliminating the need for explicit state management.

Linear Search vs Binary Search

1. **Linear Search:**
   - **Time Complexity:** O(n)
   - **Search Method:** Sequentially checks each element of the list until the target is found or the list ends.
   - **Applicability:** Works on both sorted and unsorted lists.
   - **Advantage:** Simple and easy to implement, no need for the list to be sorted.
2. **Binary Search:**
   - **Time Complexity:** O(log n)
   - **Search Method:** Repeatedly divides the sorted list in half and eliminates one half from consideration.
   - **Applicability:** Only works on sorted lists.
   - **Advantage:** Much faster for large lists due to logarithmic time complexity.

Summary Table

Feature	Linear Search	Binary Search
Time Complexity	O(n)	O(log n)

List Requirement	Unsorted or Sorted	Must be Sorted
Method	Sequential	Divide and Conquer
Implementation	Simple	Requires additional steps to sort if not already sorted
Use Cases	Small or unsorted lists	Large and sorted lists

Deleting nth element from a LinkedList

```
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

class LinkedList {
  constructor() {
    this.head = null;
    this.size = 0;
  }

  insertAtEnd(data) {
    const newNode = new Node(data);
    if (!this.head) {
      this.head = newNode;
    } else {
      let current = this.head;
      while (current.next) {
        current = current.next;
      }
      current.next = newNode;
```

```
    }
    this.size++;
  }

  deleteAtIndex(index) {
    if (index < 0 || index >= this.size) {
      console.log("Index out of range");
      return;
    }
    if (index === 0) {
      this.head = this.head.next;
    } else {
      let current = this.head;
      let count = 0;
      while (count < index - 1) {
        current = current.next;
        count++;
      }
      current.next = current.next.next;
    }
    this.size--;
  }

  display() {
    if (!this.head) {
      console.log("The list is empty");
      return;
    }
    let current = this.head;
    while (current) {
      console.log(current.data);
      current = current.next;
    }
  }
}
```

```
const list = new LinkedList();
list.insertAtEnd(10);
list.insertAtEnd(20);
list.insertAtEnd(30);
list.display();  // Output: 10 20 30
list.deleteAtIndex(2);
list.display();  // Output: 10 20
```

1. Find the Number with Maximum Occurrence from an Array

To find the number with the maximum occurrence, you can use a hash map to count the occurrences of each number.

```
function findMaxOccurrence(arr) {
  const countMap = {};
  let maxElement = arr[0];

  for (const num of arr) {
    countMap[num] = (countMap[num] || 0) + 1;
    if (countMap[num] > countMap[maxElement]) {
      maxElement = num;
    }
  }

  return maxElement;
}

const arr = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5];
const maxOccurrence = findMaxOccurrence(arr);
console.log(`Element with the maximum occurrence: ${maxOccurr
ence}`);
```

2. Implement Binary Search

Binary search requires a sorted array and works by repeatedly dividing the search interval in half.

```
function binarySearch(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left <= right) {
    const mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) return mid;
    if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }
  return -1;  // Not found
}

const sortedArray = [1, 3, 5, 7, 9];
console.log(binarySearch(sortedArray, 7));  // Output: 3
```

3. Big O Notation

Big O notation describes the performance or complexity of an algorithm. For example:

- O(1): Constant time (e.g., accessing an element in an array)
- O(n): Linear time (e.g., traversing an array)
- O(log n): Logarithmic time (e.g., binary search)
- O(n^2): Quadratic time (e.g., bubble sort)

4. Stack Overflow Error

A stack overflow error occurs when the call stack (which tracks function calls) exceeds its limit. This typically happens with deep recursion. For example:

```
function recurse() {
  recurse();  // Infinite recursion
}

recurse();  // Causes stack overflow error
```

5. Find the Sum of an Array Using Recursion

You can sum an array using a recursive function:

```
function sumArray(arr) {
  if (arr.length === 0) return 0;
  return arr[0] + sumArray(arr.slice(1));
}

const numbers = [1, 2, 3, 4, 5];
console.log(sumArray(numbers));  // Output: 15
```

6. Doubly Linked List Operations

A doubly linked list node has pointers to both the next and previous nodes.

```
class DoublyNode {
  constructor(data) {
    this.data = data;
    this.next = null;
    this.prev = null;
  }
}

class DoublyLinkedList {
  constructor() {
```

```
    this.head = null;
    this.tail = null;
  }

  insertAtEnd(data) {
    const newNode = new DoublyNode(data);
    if (!this.head) {
      this.head = this.tail = newNode;
    } else {
      this.tail.next = newNode;
      newNode.prev = this.tail;
      this.tail = newNode;
    }
  }

  display() {
    let current = this.head;
    while (current) {
      console.log(current.data);
      current = current.next;
    }
  }
}

const dll = new DoublyLinkedList();
dll.insertAtEnd(10);
dll.insertAtEnd(20);
dll.insertAtEnd(30);
dll.display();  // Output: 10 20 30
```

7. Practice Problems from Blind 75

Blind 75 is a set of 75 problems considered fundamental for coding interviews. Practice problems often include:

- Two Sum
- Median of Two Sorted Arrays
- Longest Substring Without Repeating Characters
- Merge Intervals
- Valid Parentheses

8. Linked List Workout

Common exercises include:

- Reversing a linked list
- Finding the middle node
- Detecting a cycle in a linked list
- Merging two sorted linked lists

9. Check if Array is Palindrome

An array is a palindrome if it reads the same forward and backward.

```
function isPalindrome(arr){
  if(!arr) return false;
  if(arr.length === 1) return true;
  if(arr[0]!==arr[arr.length-1]) return false;
  return isPalindrome(arr.slice(1,-1))
}
const arr = [1,2,3,2,1,2];
console.log(isPalindrome(arr))
```

10. String Workout

Common string problems include:

- Reverse a string
- Check for anagrams

- Find the longest substring without repeating characters
- String compression (e.g., "aaabb" to "a3b2")

11. Garbage Collection in JavaScript

JavaScript uses automatic garbage collection to manage memory. The JavaScript engine periodically scans memory to reclaim space used by objects that are no longer referenced. The most common algorithm used is the **Mark-and-Sweep** algorithm.

12. Reverse a String Using Recursion

Here's how to reverse a string using recursion:

```
function reverseString(str) {
  if (str === "") return "";
  return reverseString(str.substring(1)) + str[0];
}

const str = "hello";
console.log(reverseString(str));  // Output: "olleh"
```

1. Sparse Array

A sparse array is an array in which most of the elements are zero or have a default value. It's often used to save memory when dealing with large datasets with a lot of zero values. Instead of storing all elements, a sparse array only stores non-zero elements along with their indices.

Example of Sparse Array in JavaScript:

```
class SparseArray {
  constructor() {
    this.elements = {};
  }
```

```
  set(index, value) {
    if (value !== 0) {
      this.elements[index] = value;
    } else {
      delete this.elements[index];
    }
  }

  get(index) {
    return this.elements[index] || 0;
  }
}

const sparse = new SparseArray();
sparse.set(2, 10);
sparse.set(1000, 5);

console.log(sparse.get(2));     // Output: 10
console.log(sparse.get(1000));  // Output: 5
console.log(sparse.get(3));     // Output: 0
```

2. Jagged Array

A jagged array is an array of arrays where the inner arrays can have different lengths. Unlike a two-dimensional array, each sub-array in a jagged array does not need to be of the same size.

Example of a Jagged Array in JavaScript:

```
javascriptCopy code
const jaggedArray = [
  [1, 2, 3],
  [4, 5],
  [6, 7, 8, 9]
```

```
];

console.log(jaggedArray[0]);  // Output: [1, 2, 3]
console.log(jaggedArray[1]);  // Output: [4, 5]
console.log(jaggedArray[2]);  // Output: [6, 7, 8, 9]
```

3. Application of Linked List

Linked lists are useful in various scenarios due to their dynamic nature and ease of insertion and deletion:

- **Implementing Stacks and Queues**: Linked lists are used to implement data structures like stacks and queues.
- **Memory Management**: They are used in memory management systems for free space lists.
- **Graph Representation**: Adjacency lists for graphs can be represented using linked lists.
- **Dynamic Data Structures**: Useful when the number of elements can change frequently.

4. Recursion vs Iteration

Recursion involves a function calling itself to solve a problem. **Iteration** uses loops to repeat a set of instructions until a condition is met. Here are the differences:

- **Recursion**:
  - Simplifies code for problems that can be broken into similar sub-problems (e.g., factorial, Fibonacci).
  - Can lead to stack overflow if the recursion depth is too deep.
  - Often more elegant but may use more memory due to function call overhead.
- **Iteration**:
  - Generally more efficient in terms of memory and performance.

- Avoids the overhead of multiple function calls.
- Preferred for problems where the number of repetitions is known.

Example of Both:

Factorial Calculation:

Recursion:

```
function factorialRec(n) {
  if (n === 0) return 1;
  return n * factorialRec(n - 1);
}
console.log(factorialRec(5));  // Output: 120
```

Iteration:

```
function factorialIter(n) {
  let result = 1;
  for (let i = 1; i <= n; i++) {
    result *= i;
  }
  return result;
}
console.log(factorialIter(5));  // Output: 120
```

5. Big Omega (Ω), Big Theta (Θ), and Big O (O) Notation

- **Big O Notation (O)**: Describes the upper bound of an algorithm's running time. It represents the worst-case scenario.
  - Example: `O(n)` means the running time increases linearly with input size.

- **Big Omega Notation (Ω)**: Describes the lower bound of an algorithm's running time. It represents the best-case scenario.
  - Example: `Ω(n)` means the running time is at least proportional to the input size.
- **Big Theta Notation (Θ)**: Describes the exact bound of an algorithm's running time. It provides both the upper and lower bounds.
  - Example: `Θ(n)` means the running time grows linearly with input size in both the worst and best cases.

6. Binary Search and Implementation

Binary search is an efficient algorithm for finding an item from a sorted array. It repeatedly divides the search interval in half.

Binary Search Implementation:

```
function binarySearch(arr, target) {
  let left = 0;
  let right = arr.length - 1;

  while (left <= right) {
    const mid = Math.floor((left + right) / 2);
    if (arr[mid] === target) return mid;
    if (arr[mid] < target) left = mid + 1;
    else right = mid - 1;
  }
  return -1;  // Not found
}

const sortedArray = [1, 3, 5, 7, 9];
console.log(binarySearch(sortedArray, 7));  // Output: 3
```

1. Types of Memory Allocation

1.1. Static Memory Allocation:

- Memory is allocated at compile-time and cannot be changed during runtime.
- Example: Array size fixed at compile time.

```
const fixedArray = new Array(10);  // Size fixed at compile-t
ime
```

1.2. Dynamic Memory Allocation:

- Memory is allocated during runtime, allowing flexibility.
- Example: Using arrays or objects where size can change during execution.

```
let dynamicArray = [];
dynamicArray.push(1);  // Memory allocated at runtime
dynamicArray.push(2);
```

2. Doubly Linked List Operations

2.1. Definition of Doubly Linked List

A doubly linked list is a type of linked list where each node has pointers to both the next and previous nodes.

```
class DoublyNode {
  constructor(data) {
    this.data = data;
    this.next = null;
    this.prev = null;
  }
}

class DoublyLinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
```

```
  }
}
```

2.2. Insertion at End

```
insertAtEnd(data) {
  const newNode = new DoublyNode(data);
  if (!this.head) {
    this.head = this.tail = newNode;
  } else {
    this.tail.next = newNode;
    newNode.prev = this.tail;
    this.tail = newNode;
  }
}
```

2.3. Insertion at Beginning

```
insertAtBeginning(data) {
  const newNode = new DoublyNode(data);
  if (!this.head) {
    this.head = this.tail = newNode;
  } else {
    newNode.next = this.head;
    this.head.prev = newNode;
    this.head = newNode;
  }
}
```

2.4. Insertion at Index

```
insertAtIndex(index, data) {
  if (index < 0 || index > this.size) return;

  if (index === 0) {
    this.insertAtBeginning(data);
    return;
  }

  if (index === this.size) {
    this.insertAtEnd(data);
    return;
  }

  const newNode = new DoublyNode(data);
  let current = this.head;
  let count = 0;

  while (count < index - 1) {
    current = current.next;
    count++;
  }

  newNode.next = current.next;
  newNode.prev = current;
  current.next.prev = newNode;
  current.next = newNode;
}
```

2.5. Insert at Specific Value

```
insertAfterValue(existingValue, newValue) {
  let current = this.head;
```

```
  while (current) {
    if (current.data === existingValue) {
      const newNode = new DoublyNode(newValue);
      newNode.next = current.next;
      if (current.next) current.next.prev = newNode;
      current.next = newNode;
      newNode.prev = current;
      if (newNode.next === null) this.tail = newNode;
      return;
    }
    current = current.next;
  }
}
```

2.6. Deletion at Index

```
deleteAtIndex(index) {
  if (index < 0 || index >= this.size) return;

  if (index === 0) {
    this.head = this.head.next;
    if (this.head) this.head.prev = null;
    if (index === this.size - 1) this.tail = null;
  } else {
    let current = this.head;
    let count = 0;
    while (count < index) {
      current = current.next;
      count++;
    }

    current.prev.next = current.next;
    if (current.next) current.next.prev = current.prev;
```

```
    if (current === this.tail) this.tail = current.prev;
  }

  this.size--;
}
```

2.7. Deletion at Beginning

```
deleteAtBeginning() {
  if (!this.head) return;

  this.head = this.head.next;
  if (this.head) this.head.prev = null;
  if (this.head === null) this.tail = null;
}
```

2.8. Deletion at End

```
deleteAtEnd() {
  if (!this.tail) return;

  if (this.head === this.tail) {
    this.head = this.tail = null;
  } else {
    this.tail = this.tail.prev;
    this.tail.next = null;
  }
}
```

2.9. Deletion in the Middle

```
deleteAtIndex(index) {
  if (index < 0 || index >= this.size) return;

  if (index === 0) {
    this.deleteAtBeginning();
    return;
  }

  if (index === this.size - 1) {
    this.deleteAtEnd();
    return;
  }

  let current = this.head;
  let count = 0;
  while (count < index) {
    current = current.next;
    count++;
  }

  current.prev.next = current.next;
  current.next.prev = current.prev;
}
```

2.10. Reverse the Doubly Linked List

```
reverse() {
  let current = this.head;
  let temp = null;

  while (current) {
    temp = current.prev;
```

```
    current.prev = current.next;
    current.next = temp;
    current = current.prev;
  }

  if (temp) this.head = temp.prev;
}
```

3. Contiguous vs Non-Contiguous Memory

3.1. Contiguous Memory:

- Memory is allocated in a single, continuous block.
- Example: Arrays.

```
const contiguousArray = [1, 2, 3, 4, 5];  // Contiguous block
of memory
```

3.2. Non-Contiguous Memory:

- Memory is allocated in scattered blocks.
- Example: Linked lists.

```
// Nodes in a linked list are scattered in memory
const linkedList = {
  head: { data: 1, next: { data: 2, next: null } }
};
```

4. Hierarchical Data Structure

Hierarchical data structures organize data in a tree-like format, where each element is a node with a parent-child relationship.

Example: File System

```
const fileSystem = {
  name: 'root',
  children: [
    { name: 'folder1', children: [{ name: 'file1.txt' }] },
    { name: 'folder2', children: [{ name: 'file2.txt' }] }
  ]
};
```

5. Advantage and Disadvantage of Recursion

Advantages:

- **Simplifies Code**: Especially for problems like tree traversals, factorial calculation.
- **Readable**: Easier to understand for problems that naturally fit a recursive solution.

Disadvantages:

- **Stack Overflow**: Deep recursion can lead to stack overflow errors.
- **Performance Overhead**: Function calls add overhead and can be less efficient than iteration.

6. Heterogeneous Array

A heterogeneous array is an array that can hold elements of different types.

Example:

```
const heterogeneousArray = [1, 'text', true, { key: 'value' }, [1, 2, 3]];

console.log(heterogeneousArray);  // Output: [1, 'text', true, { key: 'value' }, [1, 2, 3]]
```

7. Advantage and Disadvantage of Linked List

Advantages:

- **Dynamic Size**: Easily grows and shrinks as needed.
- **Efficient Insertions/Deletions**: Inserting or deleting elements is generally faster and doesn't require shifting elements.

Disadvantages:

- **Memory Overhead**: Requires extra memory for pointers.
- **Access Time**: Accessing an element by index is slower compared to arrays because it requires traversal from the head node.

Week 14

1. Bubble Sort

Concept:
Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

JavaScript Implementation:

```
function bubbleSort(arr){
  let swapped;
  do{
    swapped = false
    for(let i =0;i<arr.length-1;i++){
      if(arr[i]>arr[i+1]){
        let temp = arr[i];
        arr[i] = arr[i+1];
        arr[i+1] = temp;
        swapped = true;
      }
    }
```

```
  }
    while(swapped)
}
const arr = [3,-2,5,1,0,4]
bubbleSort(arr);
console.log(arr);
```

Sample Workouts:

1. Sort the array `[64, 34, 25, 12, 22, 11, 90]`.
2. Sort the array `[5, 1, 4, 2, 8]`.
3. Sort the array `[3, 0, 2, 5, -1, 4, 1]`.

2. Insertion Sort

Concept:
Insertion Sort builds the final sorted array one item at a time. It picks the next item and places it in the correct position among the previously sorted items.

JavaScript Implementation:

```
function insertionSort(arr) {
    for (let i = 1; i < arr.length; i++) {
        let key = arr[i];
        let j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
    return arr;
}
```

Sample Workouts:

1. Sort the array `[12, 11, 13, 5, 6]`.
2. Sort the array `[2, 1, 9, 76, 4]`.
3. Sort the array `[3, 1, 2, 10, 7, 6]`.

3. Selection Sort

Concept:
Selection Sort repeatedly selects the minimum element from the unsorted portion of the list and moves it to the end of the sorted portion.

JavaScript Implementation:

```
function selectionSort(arr) {
    let n = arr.length;
    for (let i = 0; i < n - 1; i++) {
        let minIndex = i;
        for (let j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        [arr[i], arr[minIndex]] = [arr[minIndex], arr[i]]; // Swap
    }
    return arr;
}
```

Sample Workouts:

1. Sort the array `[29, 10, 14, 37, 13]`.
2. Sort the array `[64, 25, 12, 22, 11]`.
3. Sort the array `[5, 2, 9, 1, 5, 6]`.

4. Quick Sort

Concept:
Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

JavaScript Implementation:

```
function quickSort(arr){
  if(arr.length < 2){
    return arr;
  }
  let pivot = arr[arr.length-1];
  let left = [];
  let right = [];
  for(let i = 0 ;i<arr.length-1;i++){
    if(arr[i]<pivot){
      left.push(arr[i]);
    }else{
      right.push(arr[i]);
    }
  }
  return [...quickSort(left),pivot,...quickSort(right)];
}

const arr = [8,20,-2,4,-6];
console.log(quickSort(arr));
```

Sample Workouts:

1. Sort the array `[3, 6, 8, 10, 1, 2, 1]`.
2. Sort the array `[12, 4, 7, 9, 2, 5, 1]`.
3. Sort the array `[1, 3, 2, 8, 7, 6, 5, 4]`.

5. Merge Sort

Concept:
Merge Sort is a divide-and-conquer algorithm that divides the unsorted list into

two approximately equal parts, recursively sorts both parts, and then merges the two sorted parts.

JavaScript Implementation:

```
function mergeSort(arr){
  if(arr.length < 2){
    return arr;
  }
  const mid = Math.floor(arr.length/2);
  const leftArr = arr.slice(0,mid);
  const rightArr = arr.slice(mid);
  return merge(mergeSort(leftArr),mergeSort(rightArr));
}

function merge(leftArr,rightArr){
  const sortedArr = [];
  while(leftArr.length && rightArr.length){
    if(leftArr[0]<rightArr[0]){
      sortedArr.push(leftArr.shift());
    }else{
      sortedArr.push(rightArr.shift());
    }
  }
  return [ ...sortedArr, ...leftArr, ...rightArr ];
}
const arr = [9,8,7,6,5,4,3,2,1];
console.log(mergeSort(arr));
```

Sample Workouts:

1. Sort the array `[38, 27, 43, 3, 9, 82, 10]`.
2. Sort the array `[5, 2, 4, 6, 1, 3]`.
3. Sort the array `[12, 11, 13, 5, 6, 7]`.

You can implement these algorithms and solve the sample workouts to practice and understand how each sorting method works.

Valid Parenthesis

```
function isValid(s){
  const stack = [];
  const map = {
  '}' : '{',
  ']' : '[',
  ')' : '(',
  };

  for(let i = 0; i < s.length; i++){
    let char = s[i];
    if(char=== '[' || char === '{' || char === '('){
      stack.push(char);
    }else if(char === ']' || char === '}' || char === ')'){
      if(stack.length === 0 || stack.pop() !== map[char]){
        return false
      }
    }
  }
  return stack.length === 0;
}

const s = '([{}])';
console.log(isValid(s));
```

Queue Implementation

A queue is a linear data structure that follows the FIFO (First In, First Out) principle.

```

class Queue {
```

```
    constructor() {
        this.items = [];
    }

    // Enqueue: Add an element to the back of the queue
    enqueue(element) {
        this.items.push(element);
    }

    // Dequeue: Remove an element from the front of the queue
    dequeue() {
        if (this.isEmpty()) {
            return "Queue is empty";
        }
        return this.items.shift();
    }

    // Peek: Get the front element of the queue
    peek() {
        if (this.isEmpty()) {
            return "Queue is empty";
        }
        return this.items[0];
    }

    // isEmpty: Check if the queue is empty
    isEmpty() {
        return this.items.length === 0;
    }

    // Size: Get the number of elements in the queue
    size() {
        return this.items.length;
    }

    // Print: Display the queue elements
```

```
    print() {
        console.log(this.items.toString());
    }
}

// Example usage:
const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
queue.print(); // Output: 1,2,3
console.log(queue.dequeue()); // Output: 1
queue.print(); // Output: 2,3
```

Stack Implementation

A stack is a linear data structure that follows the LIFO (Last In, First Out) principle.

```
class Stack {
    constructor() {
        this.items = [];
    }
    // Push: Add an element to the top of the stack
    push(element) {
        this.items.push(element);
    }

    // Pop: Remove an element from the top of the stack
    pop() {
        if (this.isEmpty()) {
            return "Stack is empty";
        }
        return this.items.pop();
    }
```

```
    // Peek: Get the top element of the stack
    peek() {
        if (this.isEmpty()) {
            return "Stack is empty";
        }
        return this.items[this.items.length - 1];
    }

    // isEmpty: Check if the stack is empty
    isEmpty() {
        return this.items.length === 0;
    }

    // Size: Get the number of elements in the stack
    size() {
        return this.items.length;
    }

    // Print: Display the stack elements
    print() {
        console.log(this.items.toString());
    }
}

// Example usage:
const stack = new Stack();
stack.push(1);
stack.push(2);
stack.push(3);
stack.print(); // Output: 1,2,3
console.log(stack.pop()); // Output: 3
stack.print(); // Output: 1,2
```

Hash Table Implementation

A hash table (also called a hash map) is a data structure that implements an associative array, a structure that can map keys to values.

```
class HashTable {
    constructor(size = 50) {
        this.table = new Array(size);
        this.size = size;
    }

    // Hash function to compute an index
    _hash(key) {
        let hash = 0;
        for (let i = 0; i < key.length; i++) {
            hash += key.charCodeAt(i);
        }
        return hash % this.size;
    }

    // Set: Insert a key-value pair into the hash table
    set(key, value) {
        const index = this._hash(key);
        if (!this.table[index]) {
            this.table[index] = [];
        }
        this.table[index].push([key, value]);
    }

    // Get: Retrieve a value by its key
    get(key) {
        const index = this._hash(key);
        if (this.table.has(index)) {
            return this.table.get(index).get(key);
        }
```

```
        return undefined;
    }

    // Remove: Delete a key-value pair by its key
     remove(key) {
        const index = this._hash(key);
        this.table[index] = undefined
    }

    // Print: Display the hash table
    print() {
        for (let i = 0; i < this.table.length; i++) {
            if (this.table[i]) {
                console.log(i, this.table[i]);
            }
        }
    }
}

// Example usage:
const hashTable = new HashTable();
hashTable.set('name', 'John');
hashTable.set('age', 25);
hashTable.set('city', 'New York');
console.log(hashTable.get('name')); // Output: John
hashTable.print();
hashTable.remove('age');
hashTable.print();
```

Summary of Key Operations:

- **Queue**:
  - `enqueue(element)`: Adds an element to the back.
  - `dequeue()`: Removes and returns the front element.

- `peek()`: Returns the front element without removing it.
- `isEmpty()`: Checks if the queue is empty.
- `size()`: Returns the number of elements.
- `print()`: Prints the elements.

- **Stack**:
  - `push(element)`: Adds an element to the top.
  - `pop()`: Removes and returns the top element.
  - `peek()`: Returns the top element without removing it.
  - `isEmpty()`: Checks if the stack is empty.
  - `size()`: Returns the number of elements.
  - `print()`: Prints the elements.
- **Hash Table**:
  - `_hash(key)`: Computes an index for a given key.
  - `set(key, value)`: Inserts a key-value pair.
  - `get(key)`: Retrieves the value for a given key.
  - `remove(key)`: Deletes a key-value pair.
  - `print()`: Prints the hash table content.

These implementations are basic but can be extended with more features or optimized based on specific use cases.

Linked List Stack

A **Linked List Stack** is a stack data structure implemented using a linked list. Unlike an array-based stack, a linked list stack does not have a fixed size and can dynamically grow or shrink as elements are added or removed.

Operations:

1. **Push:** Adds an element to the top of the stack.
2. **Pop:** Removes the top element from the stack.

3. **Peek:** Returns the top element without removing it.
4. **isEmpty:** Checks if the stack is empty.
5. **Display:** Displays all the elements in the stack.

JavaScript Implementation:

```
class Node {
    constructor(value) {
        this.value = value;
        this.next = null;
    }
}

class LinkedListStack {
    constructor() {
        this.top = null;
    }

    push(element) {
        let newNode = new Node(element);
        newNode.next = this.top;
        this.top = newNode;
    }

    pop() {
        if (this.isEmpty()) {
            console.log("Stack is empty");
            return null;
        }
        let poppedNode = this.top;
        this.top = this.top.next;
        return poppedNode.value;
    }
```

```
    peek() {
        if (this.isEmpty()) {
            console.log("Stack is empty");
            return null;
        }
        return this.top.value;
    }

    isEmpty() {
        return this.top === null;
    }

    display() {
        let current = this.top;
        while (current) {
            console.log(current.value);
            current = current.next;
        }
    }
}

// Sample Workouts
let stack = new LinkedListStack();
stack.push(10);
stack.push(20);
stack.push(30);
stack.display(); // Output: 30, 20, 10
console.log(stack.pop()); // Output: 30
stack.display(); // Output: 20, 10
```

Linked List Queue

A **Linked List Queue** is a queue data structure implemented using a linked list. Like the stack, it can dynamically grow or shrink as elements are enqueued or dequeued.

Operations:

1. **Enqueue:** Adds an element to the end of the queue.
2. **Dequeue:** Removes the front element from the queue.
3. **Front:** Returns the front element without removing it.
4. **isEmpty:** Checks if the queue is empty.
5. **Display:** Displays all the elements in the queue.

JavaScript Implementation:

```
class Node {
    constructor(value) {
        this.value = value;
        this.next = null;
    }
}

class LinkedListQueue {
    constructor() {
        this.front = null;
        this.rear = null;
    }

    enqueue(element) {
        let newNode = new Node(element);
        if (this.rear === null) {
            this.front = this.rear = newNode;
        } else {
            this.rear.next = newNode;
            this.rear = newNode;
        }
    }

    dequeue() {
```

```
        if (this.isEmpty()) {
            console.log("Queue is empty");
            return null;
        }
        let dequeuedNode = this.front;
        this.front = this.front.next;
        if (this.front === null) {
            this.rear = null;
        }
        return dequeuedNode.value;
    }

    frontElement() {
        if (this.isEmpty()) {
            console.log("Queue is empty");
            return null;
        }
        return this.front.value;
    }

    isEmpty() {
        return this.front === null;
    }

    display() {
        let current = this.front;
        while (current) {
            console.log(current.value);
            current = current.next;
        }
    }
}

// Sample Workouts
let queue = new LinkedListQueue();
queue.enqueue(10);
```

```
queue.enqueue(20);
queue.enqueue(30);
queue.display(); // Output: 10, 20, 30
console.log(queue.dequeue()); // Output: 10
queue.display(); // Output: 20, 30
```

Reverse a string using Stack

```
class Stack {
    constructor() {
        this.items = {};
        this.count = 0;
    }

    // Push: Add an element to the top of the stack
    push(element) {
        this.items[this.count] = element;
        this.count++;
    }

    // Pop: Remove an element from the top of the stack
    pop() {
        if (this.isEmpty()) {
            return "Stack is empty";
        }
        this.count--;
        const item = this.items[this.count];
        delete this.items[this.count];
        return item;
    }

    // isEmpty: Check if the stack is empty
    isEmpty() {
        return this.count === 0;
    }
```

```
    // Size: Get the number of elements in the stack
    size() {
        return this.count;
    }
}

function reverseString(str) {
    const stack = new Stack();
    let reversedStr = '';

    // Push all characters of the string onto the stack
    for (let char of str) {
        stack.push(char);
    }

    // Pop all characters from the stack to build the reverse
d string
    while (!stack.isEmpty()) {
        reversedStr += stack.pop();
    }

    return reversedStr;
}

// Example usage:
const originalString = "hello";
const reversedString = reverseString(originalString);
console.log(reversedString); // Output: "olleh"
```

Explanation:

1. **Stack Class**:
    - `push(element)`: Adds an element to the top of the stack.

- `pop()`: Removes and returns the top element of the stack.
- `isEmpty()`: Checks if the stack is empty.
- `size()`: Returns the number of elements in the stack.

2. **reverseString Function**:
   - Takes a string as input.
   - Iterates over the string and pushes each character onto the stack.
   - Pops each character from the stack, appending it to `reversedStr` to form the reversed string.
   - Returns the reversed string.

Time and space complexity of Soring

1. **Bubble Sort**

- **Best Case:** O(n)(Occurs when the array is already sorted; only one pass is needed with no swaps.)
- **Average Case:** $O(n^2)$
- **Worst Case:** $O(n^2)$(Occurs when the array is sorted in reverse order; maximum number of swaps required.)

2. Insertion Sort

- **Best Case:** O(n)(Occurs when the array is already sorted; only one pass is needed to confirm the order.)
- **Average Case:** $O(n^2)$
- **Worst Case:** $O(n^2)$(Occurs when the array is sorted in reverse order.)

3. Selection Sort

- **Best Case:** $O(n^2)$
- **Average Case:** $O(n^2)$
- **Worst Case:** $O(n^2)$(Selection Sort always performs the same number of comparisons, regardless of the initial order of elements.)

4. Quick Sort

- **Best Case:** O(n log n)(Occurs when the pivot splits the array into two nearly equal halves at each step.)
- **Average Case:** O(n log n)
- **Worst Case:** $O(n^2)$(Occurs when the pivot is the smallest or largest element, resulting in unbalanced partitions. However, with good pivot selection strategies, this is rare.)

5. Merge Sort

- **Best Case:** O(n log n)
- **Average Case:** O(n log n)
- **Worst Case:** O(n log n)(Merge Sort always divides the array into two halves and requires a merge operation, so its complexity is consistent across cases.)

Summary Table

Algorithm	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)

- **Space Complexity:**
  - Bubble Sort, Insertion Sort, and Selection Sort are in-place sorting algorithms with O(1) space complexity.
  - Quick Sort has O(log n) space complexity due to the recursive call stack.
  - Merge Sort has O(n) space complexity because of the additional arrays used during the merge process.

1. Stack

Advantages:

- **LIFO Order**: The stack operates in a Last In, First Out (LIFO) order, making it ideal for scenarios where you need to reverse an order, such as undo operations in text editors, or navigating through web browser history.
- **Simple to Implement**: The stack is easy to implement and understand, using either an array or a linked list.
- **Memory Efficient**: When using a stack for function call management (as in recursion), it provides a clear, structured way to track function execution with minimal overhead.

Disadvantages:

- **Limited Access**: A stack only allows access to the top element. You cannot directly access elements deeper in the stack without popping all the elements above it.
- **Fixed Size (if implemented with an array)**: If a stack is implemented using a static array, it can lead to wasted memory or a stack overflow if the stack exceeds its allocated size.
- **Not Ideal for Complex Data Structures**: Stacks are not suitable for complex data structures where you need to access elements in the middle or the bottom.

2. Queue

Advantages:

- **FIFO Order**: The queue operates in a First In, First Out (FIFO) order, which is ideal for scenarios where you need to maintain the order of elements, such as task scheduling, breadth-first search in graphs, or managing print jobs.
- **Easy to Implement**: Similar to a stack, a queue is also simple to implement and can be done using an array, linked list, or circular buffer.
- **Parallel Processing**: Queues are used in real-time systems for managing tasks where parallel processing or order of execution is critical.

Disadvantages:

- **Limited Access**: Like a stack, a queue restricts access to its elements. You can only access the front and rear elements; there's no direct access to the

elements in between.

- **Inefficient in Some Implementations**: In certain implementations, such as using a simple array, dequeuing can be inefficient if it involves shifting all elements.
- **Not Suitable for Complex Data Manipulation**: Queues are not designed for scenarios where you need to frequently access or modify elements in the middle of the structure.

3. Hash Table

Advantages:

- **Fast Lookup**: Hash tables provide an average time complexity of O(1) for search, insert, and delete operations, making them highly efficient for large datasets.
- **Flexible Keys**: Unlike arrays, hash tables allow the use of non-integer keys, like strings, providing flexibility in how data is stored and accessed.
- **Efficient Memory Use**: Hash tables can be designed to use memory efficiently, particularly when managing large datasets with many unique keys.

Disadvantages:

- **Hash Collisions**: When two keys hash to the same index, it causes a collision. Handling collisions (via chaining or open addressing) adds complexity and can degrade performance.
- **Unordered**: Hash tables do not maintain any order of elements, so if you need to traverse elements in a specific order, hash tables are not suitable.
- **Space Complexity**: Hash tables may require more memory, especially if they are not properly sized, leading to sparse tables with lots of empty buckets.
- **Complexity of Hash Function**: The efficiency of a hash table largely depends on the quality of its hash function. A poorly designed hash function can lead to many collisions and thus poor performance.

Applications

Stack

- Function call management (recursion)
- Expression evaluation and conversion
- Undo/redo functionality
- Backtracking algorithms
- Syntax parsing

Queue

- Task scheduling
- Breadth-First Search (BFS)
- Data streaming
- Asynchronous task management
- Simulation systems

Hash Table

- Database indexing
- Caching
- Symbol tables in compilers
- Associative arrays and dictionaries
- Load balancing
- Cryptographic applications

Deterministic and nondeterministic

- **Deterministic**: If something is deterministic, it means that the outcome is predictable. Every time you give it the same input, you'll get the same result. It's like following a recipe exactly and always getting the same dish.
- **Nondeterministic** (or Undeterministic): If something is nondeterministic, it means the outcome can vary. Even with the same input, you might get

different results each time. It’s like rolling a dice—you don't know what number you'll get, even though you roll the same dice each time.

1. Stable and Unstable

- **Stable**: A sorting algorithm is stable if it preserves the relative order of equal elements in the sorted output. For example, if two elements are equal, their order relative to each other remains the same after sorting.
- **Unstable**: An unstable sorting algorithm does not guarantee the preservation of the relative order of equal elements.

2. Stack Overflow and Underflow

- **Stack Overflow**: This occurs when you try to push an element onto a stack that is already full. It often happens with fixed-size stacks.
- **Stack Underflow**: This occurs when you try to pop an element from an empty stack.

3. Divide and Conquer Strategy

- **Definition**: Divide and conquer is a strategy where a problem is broken down into smaller subproblems, solved individually, and then combined to get the final solution. It typically involves three steps: divide the problem, conquer (solve the subproblems), and combine the results.

4. Hash Functions

- **Definition**: A hash function maps input data (keys) to a fixed-size value (hash value). It is used to quickly locate a data record in a hash table.

5. Hash Values

- **Definition**: The hash value is the result produced by a hash function. It represents the position in the hash table where the corresponding data is stored.

6. Collision

- **Definition**: A collision occurs in a hash table when two different keys produce the same hash value and thus map to the same index in the hash table.

7. Methods to Prevent Collisions

- **Chaining**: Uses linked lists to store multiple elements at the same index. Each index points to a list of entries that hash to that index.
- **Linear Probing**: A method where, upon collision, the algorithm searches for the next available slot in a linear sequence.
- **Double Hashing**: Uses a second hash function to determine the step size for finding the next available slot when a collision occurs.

8. Load Factor

- **Definition**: The load factor of a hash table is the ratio of the number of elements to the number of slots (buckets) in the table. It helps determine when to resize the hash table.

9. Array vs Hashtable

- **Array**: Provides fast access by index but requires a fixed size and does not handle associative key-value pairs.
- **Hashtable**: Allows for dynamic resizing and provides fast access and management of key-value pairs, but may have overhead due to hashing and collision handling.

10. Priority Queue

- **Definition**: A priority queue is a data structure where each element has a priority. Elements are served based on priority rather than the order they were added. Common implementations use heaps.

11. Find Longest Character in String

- **Problem**: Find the most frequent character in a string.
- **Solution**:

```
function findLongestChar(str) {
    const freqMap = {};
    let maxChar = '';
    let maxCount = 0;

    for (const char of str) {
        freqMap[char] = (freqMap[char] || 0) + 1;
        if (freqMap[char] > maxCount) {
            maxCount = freqMap[char];
            maxChar = char;
        }
    }
    return { maxChar, maxCount };
}

// Example usage:
console.log(findLongestChar("aabcdeefgh")); // Output: { maxC
har: 'a', maxCount: 2 }
```

12. Frequency of String Using Hash Map

- **Problem**: Count the frequency of each character in a string.
- **Solution**:

```
function charFrequency(str) {
    const freqMap = {};
    for (const char of str) {
        freqMap[char] = (freqMap[char] || 0) + 1;
    }
    return freqMap;
}
```

```
// Example usage:
console.log(charFrequency("aabcdeefgh")); // Output: { a: 2,
b: 1, c: 1, d: 1, e: 2, f: 1, g: 1, h: 1 }
```

13. Delete Middle Element of Stack

- **Problem**: Remove the middle element of a stack.

```
function deleteMiddle(stack) {
    const mid = Math.floor(stack.length / 2);
    stack.splice(mid, 1);
    return stack;
}

// Example usage:
let stack = [1, 2, 3, 4, 5];
console.log(deleteMiddle(stack)); // Output: [1, 2, 4, 5]
```

14. Queue Reverse in Recursion

- **Problem**: Reverse a queue using recursion.
- **Solution**:

```
function reverseQueue(queue) {
    if (queue.length === 0) return;
    const item = queue.shift();
    reverseQueue(queue);
    queue.push(item);
}

// Example usage:
let queue = [1, 2, 3, 4, 5];
reverseQueue(queue);
console.log(queue); // Output: [5, 4, 3, 2, 1]
```

15. Find Duplicates Using Hash Table

- **Problem**: Identify duplicates in an array.
- **Solution**:

```
function findDuplicates(arr) {
    const freqMap = {};
    const duplicates = [];

    for (const item of arr) {
        if (freqMap[item]) {
            duplicates.push(item);
        } else {
            freqMap[item] = true;
        }
    }
    return duplicates;
}

// Example usage:
console.log(findDuplicates([1, 2, 3, 2, 4, 5, 1])); // Outpu
t: [2, 1]
```

Use of Each Sorting Algorithm

Sorting algorithms are used to arrange data in a specific order, typically ascending or descending. Here are some common sorting algorithms and their uses:

1. **Bubble Sort**
    - **Use Case**: Educational purposes, small datasets.
    - **Characteristics**: Simple but inefficient for large datasets ($O(n^2)$ time complexity).
2. **Selection Sort**
    - **Use Case**: Simple tasks where the overhead of more complex algorithms is not justified, small datasets.

- **Characteristics**: Inefficient for large datasets ($O(n^2)$ time complexity), but it has a simple implementation.

3. **Insertion Sort**
   - **Use Case**: Small datasets or nearly sorted data, online sorting.
   - **Characteristics**: Efficient for small or partially sorted datasets ($O(n^2)$ time complexity).
4. **Merge Sort**
   - **Use Case**: Large datasets, stable sort is required.
   - **Characteristics**: Efficient ($O(n \log n)$ time complexity), stable, works well for large datasets, and can be used for linked lists.
5. **Quick Sort**
   - **Use Case**: General-purpose sorting for large datasets.
   - **Characteristics**: Efficient average case ($O(n \log n)$ time complexity), but can be inefficient in the worst case ($O(n^2)$). Not stable but widely used in practice.
6. **Heap Sort**
   - **Use Case**: Situations where in-place sorting is required.
   - **Characteristics**: Efficient ($O(n \log n)$ time complexity), not stable, and can be used for large datasets.
7. **Counting Sort**
   - **Use Case**: When dealing with integers within a limited range.
   - **Characteristics**: Very efficient for specific cases ($O(n + k)$ time complexity), stable, but not suitable for large ranges of values.
8. **Radix Sort**
   - **Use Case**: When sorting integers or strings with fixed-length keys.
   - **Characteristics**: Efficient for large datasets with fixed-length keys ($O(nk)$ time complexity), stable, but requires additional memory.
9. **Bucket Sort**

- **Use Case**: When the input is uniformly distributed over a range.
- **Characteristics**: Efficient for specific cases (O(n + k) time complexity), stable, but depends on the distribution of data.

Resolution for Collision in Hash Tables

Collision resolution techniques are used to handle situations where two keys hash to the same index in a hash table. Here are common methods:

1. **Chaining**
   - **Description**: Each index of the hash table points to a linked list of entries. Collisions are resolved by adding the new entry to the list at that index.
   - **Advantages**: Simple to implement, can handle a large number of collisions.
   - **Disadvantages**: Can use more memory, performance degrades if many collisions occur.
2. **Linear Probing**
   - **Description**: When a collision occurs, the algorithm searches for the next available slot in a linear sequence (i.e., incrementing the index by a fixed amount until an empty slot is found).
   - **Advantages**: Simple and efficient for small hash tables with low load factors.
   - **Disadvantages**: Can lead to clustering (many elements grouped together), and performance degrades as the table becomes more full.
3. **Quadratic Probing**
   - **Description**: Similar to linear probing, but the interval between slots increases quadratically (i.e., 1^2, 2^2, 3^2, etc.).
   - **Advantages**: Reduces clustering compared to linear probing.
   - **Disadvantages**: Can still have issues with clustering, and finding an empty slot may be more complex.
4. **Double Hashing**

- **Description**: Uses a second hash function to determine the step size for probing. This helps to reduce clustering.
- **Advantages**: Provides a more uniform distribution of slots.
- **Disadvantages**: More complex to implement and requires a good choice of hash functions.

5. **Rehashing**
    - **Description**: When the load factor of the hash table exceeds a certain threshold, the table is resized, and all existing entries are rehashed to new indices in the larger table.
    - **Advantages**: Helps maintain efficient performance as the number of entries grows.
    - **Disadvantages**: Can be computationally expensive during rehashing, and requires additional memory.

Week 15

Tree Concepts Overview

A **tree** is a data structure that consists of nodes, where each node contains a value and references to its child nodes. The topmost node is called the **root**. Trees are hierarchical structures and can be used to model various real-world scenarios like file systems, organizational structures, etc.

Key Terms:

- **Node**: The basic unit of a tree.
- **Root**: The topmost node in a tree.
- **Leaf**: A node with no children.
- **Edge**: The connection between one node and another.
- **Height**: The length of the longest path from the root to a leaf.
- **Depth**: The length of the path from the root to a specific node.

Binary Search Tree (BST) Concepts Overview

A **Binary Search Tree (BST)** is a type of binary tree where each node has at most two children, typically referred to as the left and right children. BSTs have a special property: for any given node, the left subtree contains only values less than the node's value, and the right subtree contains only values greater than the node's value. This property makes BSTs particularly useful for searching, insertion, and deletion operations.

Key Operations:

- **Insertion**: Adds a node to the BST while maintaining the BST property.
- **Search (Contains)**: Checks if a value exists in the BST.
- **Deletion**: Removes a node from the BST and re-organizes the tree to maintain the BST property.
- **Traversals**:
  - **Inorder**: Left, Root, Right (yields sorted order for BST).
  - **Preorder**: Root, Left, Right.
  - **Postorder**: Left, Right, Root.

Sample Workouts in JavaScript

1. Create a Binary Search Tree with Insertion, Contains, and Deletion

```
class Node {
  constructor(data) {
    this.data = data;
    this.left = null;
    this.right = null;
  }
}

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

```
  }

  isEmpty() {
    return this.root === null;
  }

  insert(data) {
    const newNode = new Node(data);
    if (this.isEmpty()) {
      this.root = newNode;
    } else {
      this.insertNode(this.root, newNode);
    }
  }

  insertNode(root, newNode) {
    if (newNode.data < root.data) {
      if (root.left === null) {
        root.left = newNode;
      } else {
        this.insertNode(root.left, newNode);
      }
    } else {
      if (root.right === null) {
        root.right = newNode;
      } else {
        this.insertNode(root.right, newNode);
      }
    }
  }

  search(root, value) {
    if (!root) {
      return false;
    } else {
      if (root.data === value) {
```

```
        return true;
      } else if (value < root.data) {
        return this.search(root.left, value);
      } else {
        return this.search(root.right, value);
      }
    }
  }

  preOrder(root) {
    if (root) {
      console.log(root.data);
      this.preOrder(root.left);
      this.preOrder(root.right);
    }
  }

  inOrder(root) {
    if (root) {
      this.inOrder(root.left);
      console.log(root.data);
      this.inOrder(root.right);
    }
  }

  postOrder(root) {
    if (root) {
      this.postOrder(root.left);
      this.postOrder(root.right);
      console.log(root.data);
    }
  }

  BFS() {
    const queue = [];
    queue.push(this.root);
```

```
    while (queue.length) {
      let curr = queue.shift();
      console.log(curr.data);
      if (curr.left) {
        queue.push(curr.left);
      }
      if (curr.right) {
        queue.push(curr.right);
      }
    }
  }

  min(root) {
    if (!root.left) {
      return root.data;
    } else {
      return this.min(root.left);
    }
  }

  max(root) {
    if (!root.right) {
      return root.data;
    } else {
      return this.max(root.right);
    }
  }

  delete(value) {
    this.root = this.deleteNode(this.root, value);
  }

  deleteNode(root, value) {
    if (root === null) {
      return root;
    }
```

```
    if (value < root.data) {
      root.left = this.deleteNode(root.left, value);
    } else if (value > root.data) {
      root.right = this.deleteNode(root.right, value);
    } else {
      if (!root.left && !root.right) {
        return null;
      }
      if (!root.left) {
        return root.right;
      } else if (!root.right) {
        return root.left;
      }
      root.data = this.min(root.right);
      root.right = this.deleteNode(root.right, root.data);
    }
    return root;
  }

  findClosestValue(root, target, closest = null) {
    if (!root) {
      return closest;
    }

    if (closest === null || Math.abs(target - root.data) < Ma
th.abs(target - closest)) {
      closest = root.data;
    }

    if (target < root.data) {
      return this.findClosestValue(root.left, target, closes
t);
    } else if (target > root.data) {
      return this.findClosestValue(root.right, target, closes
t);
    } else {
```

```
      return closest;
    }
  }

  isValidBST(root, min = -Infinity, max = Infinity) {
    if (!root) {
      return true;
    }

    if (root.data <= min || root.data >= max) {
      return false;
    }

    return this.isValidBST(root.left, min, root.data) && thi
s.isValidBST(root.right, root.data, max);
  }
}

// Usage
const bst = new BST();
bst.insert(10);
bst.insert(5);
bst.insert(15);
bst.insert(3);
bst.insert(7);

console.log("Is the tree empty? :", bst.isEmpty());
console.log("Search for 3:", bst.search(bst.root, 3));

console.log("Pre Order Traversal:");
bst.preOrder(bst.root);

console.log("In Order Traversal:");
bst.inOrder(bst.root);

console.log("Post Order Traversal:");
```

```
bst.postOrder(bst.root);

console.log("BFS Traversal:");
bst.BFS();

console.log("Minimum value:", bst.min(bst.root));
console.log("Maximum value:", bst.max(bst.root));

const target = 6;
console.log(`Closest value to ${target}:`, bst.findClosestVal
ue(bst.root, target));

bst.delete(3);
console.log("2nd Post Order Traversal:");
bst.postOrder(bst.root);

console.log("Is the tree a valid BST? :", bst.isValidBST(bst.
root));
```

1. Pre-order Traversal

- **Order:** `Root → Left → Right`
- **Steps:**
  1. Visit the root node.
  2. Traverse the left subtree in pre-order.
  3. Traverse the right subtree in pre-order.

Example:

Given the tree:

```
    10
   /  \
  5    15
```

```
  / \      \
 3   7      18
```

- **Pre-order traversal:** `10, 5, 3, 7, 15, 18`

2. In-order Traversal

- **Order:** `Left → Root → Right`
- **Steps:**
  1. Traverse the left subtree in in-order.
  2. Visit the root node.
  3. Traverse the right subtree in in-order.

Example:

Given the same tree:

```
     10
    /  \
   5    15
  / \     \
 3   7     18
```

- **In-order traversal:** `3, 5, 7, 10, 15, 18`
- **Note:** For a BST, in-order traversal visits nodes in **sorted order**.

3. Post-order Traversal

- **Order:** `Left → Right → Root`
- **Steps:**
  1. Traverse the left subtree in post-order.
  2. Traverse the right subtree in post-order.

3. Visit the root node.

Example:

Given the same tree:

```
    10
   /  \
  5    15
 / \     \
3   7     18
```

- **Post-order traversal:** `3, 7, 5, 18, 15, 10`

Learning Concepts of Heap

Heap is a special tree-based data structure that satisfies the **heap property**:

- **Min Heap:** The value of each node is greater than or equal to the value of its parent. The smallest element is at the root.
- **Max Heap:** The value of each node is less than or equal to the value of its parent. The largest element is at the root.

Key Operations:

- **Build Heap:** Create a heap from an unsorted array.
- **Insert:** Add a new element to the heap while maintaining the heap property.
- **Remove (Extract):** Remove the root element (smallest in a min heap, largest in a max heap) while maintaining the heap property.

Implementing Min Heap and Max Heap in JavaScript

Min Heap Implementation

```
class MinHeap {
```

```
  constructor() {
    this.heap = [];
  }

  // Helper methods
  parentIndex(index) {
    return Math.floor((index - 1) / 2);
  }

  leftChildIndex(index) {
    return 2 * index + 1;
  }

  rightChildIndex(index) {
    return 2 * index + 2;
  }

  swap(index1, index2) {
    [this.heap[index1], this.heap[index2]] = [this.heap[index
2], this.heap[index1]];
  }

  // Insert an element into the heap
  insert(value) {
    this.heap.push(value);
    this.heapifyUp(this.heap.length - 1);
  }

  // Heapify up to maintain heap property after insertion
  heapifyUp(index) {
    let parent = this.parentIndex(index);
    while (index > 0 && this.heap[index] < this.heap[parent])
{
      this.swap(index, parent);
      index = parent;
      parent = this.parentIndex(index);
```

```
    }
  }

  // Remove the root (minimum element)
  remove() {
    if (this.heap.length === 0) return null;
    if (this.heap.length === 1) return this.heap.pop();

    const root = this.heap[0];
    this.heap[0] = this.heap.pop();
    this.heapifyDown(0);
    return root;
  }

  // Heapify down to maintain heap property after removal
  heapifyDown(index) {
    let left = this.leftChildIndex(index);
    let right = this.rightChildIndex(index);
    let smallest = index;

    if (left < this.heap.length && this.heap[left] < this.hea
p[smallest]) {
      smallest = left;
    }
    if (right < this.heap.length && this.heap[right] < this.h
eap[smallest]) {
      smallest = right;
    }
    if (smallest !== index) {
      this.swap(index, smallest);
      this.heapifyDown(smallest);
    }
  }

  // Build a heap from an array
  buildHeap(array) {
```

```
    this.heap = array;
    for (let i = Math.floor(this.heap.length / 2); i >= 0; i-
-) {
      this.heapifyDown(i);
    }
  }

  // Get the heap as an array
  getHeap() {
    return this.heap;
  }
}

const minHeap = new MinHeap();

// Insert values into the heap
minHeap.insert(10);
minHeap.insert(5);
minHeap.insert(20);
minHeap.insert(3);
minHeap.insert(8);

console.log("Heap after insertions:", minHeap.getHeap()); //
Should print the heap structure

// Remove the minimum element (root)
console.log("Removed element:", minHeap.remove()); // Should
remove the minimum (3)
console.log("Heap after removal:", minHeap.getHeap()); // Sho
uld print the heap structure

// Build a heap from an array
minHeap.buildHeap([15, 5, 20, 1, 17, 10]);
console.log("Heap after building from array:", minHeap.getHea
p()); // Should print the heap structure
```

Max Heap Implementation

```
class MaxHeap {
  constructor() {
    this.heap = [];
  }

  parentIndex(index) {
    return Math.floor((index - 1) / 2);
  }

  leftChildIndex(index) {
    return 2 * index + 1;
  }

  rightChildIndex(index) {
    return 2 * index + 2;
  }

  swap(index1, index2) {
    [this.heap[index1], this.heap[index2]] = [this.heap[index
1], this.heap[index2]];
  }

  insert(value) {
    this.heap.push(value);
    this.heapifyUp(this.heap.length - 1);
  }

  heapifyUp(index) {
    let parent = this.parentIndex(index);
    while (index > 0 && this.heap[index] > this.heap[parent])
{
      this.swap(index, parent);
      index = parent;
```

```
      parent = this.parentIndex(index);
    }
  }

  remove() {
    if (this.heap.length === 0) return null;
    if (this.heap.length === 1) return this.heap.pop();

    const root = this.heap[0];
    this.heap[0] = this.heap.pop();
    this.heapifyDown(0);
    return root;
  }

  heapifyDown(index) {
    let left = this.leftChildIndex(index);
    let right = this.rightChildIndex(index);
    let largest = index;

    if (left < this.heap.length && this.heap[left] > this.hea
p[largest]) {
      largest = left;
    }
    if (right < this.heap.length && this.heap[right] > this.h
eap[largest]) {
      largest = right;
    }
    if (largest !== index) {
      this.swap(index, largest);
      this.heapifyDown(largest);
    }
  }

  buildHeap(array) {
    this.heap = array;
    for (let i = Math.floor(this.heap.length / 2); i >= 0; i-
```

```
-) {
      this.heapifyDown(i);
    }
  }

  getHeap() {
    return this.heap;
  }
}
```

Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses the binary heap data structure to sort elements. It has a time complexity of O(nlogn)O(n \log n)O(nlogn) and works in two main phases:

1. **Building a Max Heap** (or Min Heap for descending order)
2. **Extracting the Maximum Element** repeatedly from the heap and rebuilding the heap to maintain the heap property.

Steps of Heap Sort

1. **Build a Max Heap** from the input data.
2. **Swap** the root (maximum element) with the last element of the heap.
3. **Reduce the heap size** by one, effectively removing the last element (sorted part).
4. **Heapify** the root to maintain the max heap property.
5. Repeat steps 2-4 until all elements are sorted.

Heap Sort Implementation in JavaScript

```
function heapify(arr, n, i) {
  let largest = i;
  let left = 2 * i + 1;
```

```
  let right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest]) {
    largest = left;
  }

  if (right < n && arr[right] > arr[largest]) {
    largest = right;
  }

  if (largest !== i) {
    [arr[i], arr[largest]] = [arr[largest], arr[i]];
    heapify(arr, n, largest);
  }
}

function heapSort(arr) {
  let n = arr.length;

  // Build a max heap
  for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
    heapify(arr, n, i);
  }

  // Extract elements from heap one by one
  for (let i = n - 1; i > 0; i--) {
    // Move current root to the end
    [arr[0], arr[i]] = [arr[i], arr[0]];

    // Call max heapify on the reduced heap
    heapify(arr, i, 0);
  }

  return arr;
}
```

```
// Example usage:
let array = [12, 11, 13, 5, 6, 7];
console.log("Unsorted array:", array);

let sortedArray = heapSort(array);
console.log("Sorted array:", sortedArray);
```

Sample Workouts

1. **Workout 1:** Sort an array of integers using heap sort.
   - **Input:** `[12, 11, 13, 5, 6, 7]`
   - **Output:** `[5, 6, 7, 11, 12, 13]`
2. **Workout 2:** Sort an array of floating-point numbers using heap sort.
   - **Input:** `[4.5, 2.3, 8.7, 3.3, 1.2]`
   - **Output:** `[1.2, 2.3, 3.3, 4.5, 8.7]`
3. **Workout 3:** Sort a large array (e.g., 10,000 elements) to test the performance of heap sort.
   - **Input:** Randomly generated array of 10,000 integers.
   - **Output:** Sorted array in ascending order.

Graph Concepts Overview

A **Graph** is a collection of nodes (vertices) and edges connecting pairs of nodes. Graphs can be directed (edges have direction) or undirected (edges have no direction), and can have weights associated with edges.

Key Operations:

- **Add Vertex**: Add a new node to the graph.
- **Add Edge**: Connect two nodes with an edge.
- **Remove Vertex**: Remove a node and all edges connected to it.
- **Remove Edge**: Remove the connection between two nodes.

Graph Traversals

1. Breadth-First Search (BFS)

BFS explores nodes level by level and is useful for finding the shortest path in an unweighted graph.

```
function bfs(graph, start) {
    let visited = new Set();
    let queue = [start];
    while (queue.length > 0) {
        let node = queue.shift();
        if (!visited.has(node)) {
            visited.add(node);
            console.log(node);
            for (let neighbor of graph[node]) {
                if (!visited.has(neighbor)) {
                    queue.push(neighbor);
                }
            }
        }
    }
}
```

2. Depth-First Search (DFS)

DFS explores nodes by going as deep as possible along each branch before backtracking.

```
function dfs(graph, start, visited = new Set()) {
    if (!visited.has(start)) {
        visited.add(start);
        console.log(start);
        for (let neighbor of graph[start]) {
```

```
            if (!visited.has(neighbor)) {
                dfs(graph, neighbor, visited);
            }
        }
    }
}
```

Sample Workouts for Graphs

1. Graph Representation

```
class Graph {
  constructor(){
    this.adjacencyList = {}
  }

  addVertex(vertex){
    if(!this.adjacencyList[vertex]){
      this.adjacencyList[vertex] = new Set();
    }
  }
  addEdge(vertex1, vertex2){
    if(this.adjacencyList[vertex1]){
      this.addVertex[vertex1]
    }
    if(this.adjacencyList[vertex2]){
      this.addVertex[vertex2]
    }
    this.adjacencyList[vertex1].add(vertex2)
    this.adjacencyList[vertex2].add(vertex1)
  }
  display(){
    for(let vertex in this.adjacencyList){
```

```
      console.log(vertex + "->" + [...this.adjacencyList[vert
ex]]);
    }
  }
hasEdge(v1,v2){
  return this.adjacencyList[v1].has(v2) && this.adjacencyList
[v2].has(v1)
}
  removeEdge(v1,v2){
    this.adjacencyList[v1].delete(v2)
    this.adjacencyList[v2].delete(v1)
  }
  removeVertex(vertex){
    if(!this.adjacencyList[vertex]){
      return
    }
    for(let adjacentVertex of this.adjacencyList[vertex]){
      this.removeEdge(vertex, adjacentVertex);
    }
    delete this.adjacencyList[vertex];
  }

}

const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addEdge("A","B")
graph.addEdge("B","C")
graph.display();
console.log(graph.hasEdge("A","B"));
graph.removeEdge("A","B");
graph.display()
```

```
graph.removeVertex("B");
graph.display();
```

Practice Problems

1. **Heap**: Try problems like "Kth Largest Element in an Array" or "Find Median from Data Stream" on competitive coding platforms.
2. **Trie**: Look for problems such as "Add and Search Word - Data structure design" or "Replace Words".
3. **Graph**: Solve problems like "Number of Islands", "Clone Graph", or "Course Schedule".

For each of these data structures, you can find many practice problems on platforms

Learning the Concept of Trie

A **Trie** (pronounced "try") is a tree-like data structure used to efficiently store and retrieve keys in a dataset of strings. It is particularly useful for tasks like autocomplete, spell checking, and prefix-based searching.

Key Concepts of Trie

- **Nodes:** Each node in a Trie represents a single character of a string. The root node is empty, and each path down the tree represents a word or a prefix of a word.
- **Edges:** Edges connect nodes, representing the sequence of characters in the stored strings.
- **End of Word Marker:** Some nodes are marked to indicate the end of a word.

Trie Operations

1. **Insert:** Add a word to the Trie.
2. **Search:** Check if a word exists in the Trie.
3. **StartsWith:** Check if there is any word in the Trie that starts with a given prefix.

4. **Delete:** Remove a word from the Trie (optional and more complex).

Implementing Trie in JavaScript

```
class TrieNode {
  constructor() {
    this.children = {};
    this.isEndOfWord = false;
  }
}

class Trie {
  constructor() {
    this.root = new TrieNode();
  }

  // Insert a word into the Trie
  insert(word) {
    let node = this.root;
    for (let char of word) {
      if (!node.children[char]) {
        node.children[char] = new TrieNode();
      }
      node = node.children[char];
    }
    node.isEndOfWord = true;
  }

  // Search for a word in the Trie
  search(word) {
    let node = this.root;
    for (let char of word) {
      if (!node.children[char]) {
        return false;
      }
```

```
      node = node.children[char];
    }
    return node.isEndOfWord;
  }

  // Check if any word starts with the given prefix
  startsWith(prefix) {
    let node = this.root;
    for (let char of prefix) {
      if (!node.children[char]) {
        return false;
      }
      node = node.children[char];
    }
    return true;
  }
}

// Example usage:
const trie = new Trie();
trie.insert("apple");
console.log(trie.search("apple"));   // true
console.log(trie.search("app"));     // false
console.log(trie.startsWith("app")); // true
trie.insert("app");
console.log(trie.search("app"));     // true
```

Sample Workouts

1. **Workout 1: Insert and Search Words**
   - Insert words: "hello", "world", "trie", "tree"
   - Search for words: "world" (should return true), "trie" (should return true), "tried" (should return false)
2. **Workout 2: Prefix Search**

- Insert words: "banana", "band", "bandit", "bank"
- Check if the Trie has words starting with the prefix "ban" (should return true) and "bat" (should return false)

3. **Workout 3: Deleting Words (Advanced)**
    - Implement a function to delete a word from the Trie and ensure it handles cases where part of the word is shared with other words.

Tree vs. Graphs:

- **Tree**: A tree is a special case of a graph. It is a connected, acyclic, and directed graph with N nodes and N-1 edges. Trees have a strict hierarchy (parent-child relationships) and no cycles.
- **Graph**: A graph is a broader structure that can contain cycles and can be either directed or undirected. It is a set of vertices (or nodes) connected by edges. Graphs do not have to be connected or acyclic like trees.

Delete Node in Graph in JavaScript:

To delete a node in a graph (represented by an adjacency list) in JavaScript, you need to:

1. Remove all edges that reference the node (i.e., remove the node from the adjacency lists of other nodes).
2. Delete the node from the adjacency list.

Example code in JavaScript:

```
function deleteNode(graph, nodeToDelete) {
    // Remove the node from other nodes' adjacency lists
    for (let node in graph) {
        graph[node] = graph[node].filter(neighbor => neighbor
!== nodeToDelete);
    }
    // Delete the node itself
    delete graph[nodeToDelete];
```

```
}

let graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
};

deleteNode(graph, 'C');
console.log(graph);
```

1. Tree vs. Binary Search Tree (BST):

- **Tree**: A tree is a hierarchical data structure where nodes are connected by edges. A tree can have any structure and any number of children per node.
- **Binary Search Tree (BST)**: A special type of binary tree where every node has at most two children, and the tree maintains a specific order:
  - The left child's value is less than the parent node.
  - The right child's value is greater than the parent node.
- **Key difference**: A BST is organized for efficient searching, insertion, and deletion operations, while a general tree does not enforce such ordering.

2. Heap Sort:

- **Heap Sort** is a comparison-based sorting algorithm that utilizes the properties of a binary heap (either max-heap or min-heap). The basic steps:
  1. Build a heap from the input data.
  2. Repeatedly extract the root element from the heap (the largest or smallest, depending on the heap type) and place it at the end of the sorted array.
  3. Adjust the heap (heapify) after every extraction to maintain the heap property.
- **Time Complexity**: O(n log n) in all cases (best, worst, and average).

3. Concepts of Trie:

- A **Trie** is a tree-like data structure used to efficiently store and search a collection of strings.
- **Nodes and Edges**: Each node represents a single character, and edges represent the transition between characters.
- **End of Word**: Some nodes are marked to signify the end of a word.
- **Applications**: Autocomplete, spell checking, and prefix-based searching.
- **Time Complexity**: Searching, inserting, or deleting a word of length `L` takes O(L), where `L` is the length of the word.

4. Time Complexity of Heap:

- **Insertion and Deletion (Heapify)**: O(log n), because the height of the heap is log n, and both operations might need to adjust the tree from the root to a leaf or vice versa.
- **Building a Heap**: O(n). While it may seem like building a heap should be O(n log n), the process of heap construction can be optimized to O(n).
- **Extracting the Max or Min**: O(log n).

5. Balanced Tree:

- A **Balanced Tree** is a type of binary tree where the height of the left and right subtrees of any node differ by at most one. This ensures that operations like searching, insertion, and deletion remain efficient (O(log n) time complexity).
- Examples: AVL Tree, Red-Black Tree.

6. Red-Black Tree:

- A **Red-Black Tree** is a self-balancing binary search tree with additional properties:
    1. Each node is either red or black.
    2. The root is always black.

3. No two consecutive red nodes are allowed (a red node cannot have a red child).
4. Every path from a node to its descendant NULL nodes must have the same number of black nodes.

- **Time Complexity**: Insertion, deletion, and search take O(log n).

7. Max-Heap vs Min-Heap:

- **Max-Heap**: In a max-heap, the value of the root node is greater than or equal to the values of its children. Thus, the largest element is at the root. Used in scenarios where you need to repeatedly retrieve the largest element (e.g., heapsort).
- **Min-Heap**: In a min-heap, the value of the root node is less than or equal to the values of its children. Thus, the smallest element is at the root. Used when you need to repeatedly retrieve the smallest element (e.g., priority queues).
- **Key Difference**: The order of elements is reversed; max-heap focuses on the largest values, min-heap on the smallest.

8. AVL Tree:

- An **AVL Tree** is a type of self-balancing binary search tree where the difference between heights of left and right subtrees cannot be more than one for all nodes. If at any time the balance factor (height difference) becomes more than one, rotations are used to rebalance the tree.
- **Time Complexity**: Insertion, deletion, and search take O(log n) due to the tree being balanced.

9. Height of a Binary Tree:

- The **height of a binary tree** is the number of edges on the longest path from the root to a leaf node.
- **Calculation**: The height can be found recursively:

```
function height(node) {
```

```
    if (node === null) return -1;  // Base case: empty tre
e has height -1
    return Math.max(height(node.left), height(node.right))
+ 1;
}
```

- For an empty tree, the height is considered -1, and for a tree with just one node (root), the height is 0.

1. Adjacency List:

- An **adjacency list** is a way to represent a graph. Each vertex in the graph has a list of adjacent vertices (nodes it is connected to by an edge). This representation is efficient for sparse graphs.
- **Example in JavaScript:**

```
let graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
};
```

2. Types of Graphs:

- **Directed Graph (Digraph)**: Edges have a direction (from one vertex to another). E.g., A → B.
- **Undirected Graph**: Edges have no direction; the connection is bidirectional. E.g., A - B.
- **Weighted Graph**: Edges have weights (costs or distances). E.g., A --(5)-- B.
- **Unweighted Graph**: Edges do not have weights.

- **Cyclic Graph**: Contains at least one cycle (a path that starts and ends at the same vertex).
- **Acyclic Graph**: Contains no cycles.
- **Connected Graph**: There is a path between every pair of vertices.
- **Disconnected Graph**: Some vertices are not connected by any path.

3. Cycles and Loops:

- **Cycle**: A path in a graph that starts and ends at the same vertex with no repetitions of edges or vertices (other than the start/end vertex). E.g., A → B → C → A.
- **Loop**: An edge that connects a vertex to itself. In simple graphs, loops are not allowed.

4. Applications of Graphs:

- **Social Networks**: Represent relationships between users.
- **Routing Algorithms**: Used in GPS and network routing (e.g., Dijkstra's algorithm).
- **Recommendation Systems**: Suggestions based on user preferences and interactions.
- **Network Topology**: Representing and analyzing network structures.
- **Dependency Resolution**: In package managers and build systems.

5. Types of Trees:

- **Binary Tree**: Each node has at most two children.
- **Binary Search Tree (BST)**: A binary tree where left child nodes are smaller, and right child nodes are larger.
- **AVL Tree**: A self-balancing BST where the height difference between left and right subtrees is at most 1.
- **Red-Black Tree**: A self-balancing BST with specific color properties to maintain balance.

- **Trie**: A tree used for storing strings, allowing for efficient retrieval of words based on prefixes.
- **Segment Tree**: Used for range queries and updates on arrays.
- **Fenwick Tree (Binary Indexed Tree)**: Used for cumulative frequency tables.

6. Complete Tree:

- A **Complete Tree** (or complete binary tree) is a binary tree where all levels, except possibly the last, are fully filled, and all nodes are as far left as possible.

7. Full Tree:

- A **Full Tree** (or proper binary tree) is a binary tree where every node other than the leaves has exactly two children.

8. Perfect Tree:

- A **Perfect Tree** is a binary tree where all internal nodes have exactly two children, and all leaf nodes are at the same level.

9. Heap:

- A **Heap** is a special binary tree-based data structure that satisfies the heap property:
  - **Max-Heap**: Every parent node is greater than or equal to its children.
  - **Min-Heap**: Every parent node is less than or equal to its children.
- **Operations**: Insertion, deletion, and heapify operations are efficient (O(log n)).

10. Trie:

- A **Trie** is a tree-like data structure used for storing a dynamic set of strings. Each node represents a single character, and paths from the root to a node represent prefixes of strings.
- **Operations**: Insertion, search, and prefix matching are typically O(L), where L is the length of the string.

11. BST Insertion Using Recursion:

- Inserting a node into a Binary Search Tree (BST) recursively involves:
  1. Comparing the value to be inserted with the current node's value.
  2. Recursively moving to the left or right child based on the comparison.
  3. Inserting the new node when a null child is reached.

Example in JavaScript:

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

function insert(root, value) {
    if (root === null) {
        return new TreeNode(value);
    }
    if (value < root.value) {
        root.left = insert(root.left, value);
    } else {
        root.right = insert(root.right, value);
    }
    return root;
}

let root = new TreeNode(10);
root = insert(root, 5);
root = insert(root, 15);
```