# Data Structures

## Linked List

Create a class that implements a basic singly linked list, allowing you to perform common operations on it, such as appending, prepending, inserting, removing, and printing elements.

**Task Instructions:**

1. Define a `Node` class:
    - Create a `Node` class with a constructor that takes one parameter, `data`, and initializes two properties: `data` and `next`.
    - The `data` property should store the data for the node.
    - The `next` property should initially be set to `null`.
2. Define a `LinkedList` class:
    - Create a `LinkedList` class with a constructor that initializes two properties: `head` and `size`.
    - The `head` property should initially be set to `null`, indicating an empty list.
    - The `size` property should initially be set to `0` to keep track of the number of elements in the list.
3. Implement the following methods in the `LinkedList` class:
    - `append(data)`: Adds a new node with the given data to the end of the list.
    - `prepend(data)`: Adds a new node with the given data to the beginning of the list.
    - `insert(data, index)`: Inserts a new node with the given data at the specified index in the list. If the index is out of bounds, return `false`.
    - `removeAt(index)`: Removes the node at the specified index and returns its data. If the index is out of bounds, return `null`.
    - `remove(data)`: Removes the first occurrence of a node with the specified data from the list. If the data is not found, return `false`.
    - `getSize()`: Returns the current size of the linked list.
    - `isEmpty()`: Returns `true` if the linked list is empty (size is 0), otherwise returns `false`.
    - `printList()`: Logs the data of all nodes in the linked list to the console.

```
const list = new LinkedList();
list.append(1);
list.append(2);
list.prepend(0);
list.insert(3, 3);
list.printList(); // Output: 0 1 2 3
```

```
console.log("Size:", list.getSize()); // Output: Size: 4
list.remove(2);
list.removeAt(2);
list.printList(); // Output: 0 1
console.log("Size:", list.getSize()); // Output: Size: 2
```

## Stack

Create a class of a stack data structure with the following functionalities:

1. push: Add an element onto the stack.
2. pop: Remove and return the top element from the stack.
3. top: Retrieve the top element of the stack without removing it.
4. isEmpty: Check if the stack is empty.
5. getSize: Get the number of elements in the stack.

**Implementation Options:**

1. **Using an Array:**
   - Create a class named `Stack`.
   - Initialize an empty array as the stack's storage.
   - Implement the push, pop, peek, isEmpty, and getSize methods.
   - Test the stack by pushing and popping elements.

   Example Usage:

   ```
   const stack = new Stack();
   stack.push(1);
   stack.push(2);
   console.log(stack.pop()); // Should print 2
   console.log(stack.top()); // Should print 1
   console.log(stack.isEmpty()); // Should print false
   console.log(stack.getSize()); // Should print 1
   ```

2. **Using a Linked List:**
   - Create a class named `Stack`.
   - Initialize a variable (`top`) to represent the top of the stack as null and a size counter.
   - Implement the push, pop, peek, isEmpty, and getSize methods using a linked list.
   - Test the stack by pushing and popping elements.

   Example Usage:

   ```
   const stack = new Stack();
   stack.push(1);
   stack.push(2);
   console.log(stack.pop()); // Should print 2
   ```

```
console.log(stack.top()); // Should print 1
console.log(stack.isEmpty()); // Should print false
console.log(stack.getSize()); // Should print 1
```

## Queue

Task Title: JavaScript Queue Implementation

Task Description: Implement a queue data structure in JavaScript using a class called `Queue`. The queue should have the following functionalities:

1. `enqueue(element)`: Add an element to the back of the queue.
2. `dequeue()`: Remove and return the front element of the queue.
3. `front()`: Return the front element of the queue without removing it.
4. `isEmpty()`: Check if the queue is empty.
5. `size()`: Return the number of elements in the queue.
6. `print()`: Print the elements of the queue.

Your implementation should use an array to store the elements of the queue. Ensure that the queue follows the first-in-first-out (FIFO) principle, where elements are removed from the front of the queue.

Example Usage:

```
const queue = new Queue();

queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);

queue.print(); // Output: 1 2 3

console.log("Front element: " + queue.front()); // Output: Front element:
1

queue.dequeue();
queue.print(); // Output: 2 3

console.log("Queue size: " + queue.size()); // Output: Queue size: 2
```

## Map & Set

- Create a class, that will have `addUsers` and `getUserById` and will use a Map to store user data, where the keys are user IDs, and the values are user objects.

```
const mappedUser = new Users([
  { id: 1, name: "Alice", email: "alice@example.com" },
  { id: 2, name: "Bob", email: "bob@example.com" },
]);
```

```
mappedUser.getUserById(1); // { id: 1, name: "Alice", email:
"alice@example.com" };
mappedUser.addUsers([{ id: 3, name: "Ann", email: "ann@example.com" }]);
mappedUser.getUserById(3); // { id: 3, name: "Ann", email:
"ann@example.com" }
```

- Implement memoization for a recursive function using a Map to store previously computed results.
- Write a function that will remove duplicates from an array