Data Structures and Algorithms

Linear Data Structures

Engage and Think

Imagine you and your friends have **college lockers** arranged in a row. Each locker has a unique number, making it easy to find your belongings.
Now, think about this:

- What if a new student joins mid-year and needs a locker?
- What if someone moves out and leaves their locker empty?
- How do you make sure your locker stays yours, no matter where it’s placed?

Have you ever had to store, find, or shift things around in a limited space like this? How do you think computers handle something like this with data?

Learning Objectives

By the end of this lesson, you will be able to:

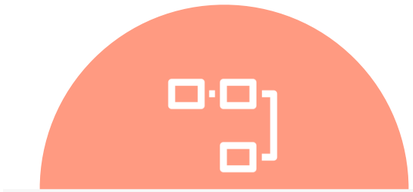
- Identify the suitable linear data structure for various scenarios to streamline data management
- Implement stacks using arrays and linked lists to understand diverse data storage methods
- Implement CRUD (create, read, update, and delete) operations on a singly linked list to establish fundamental data manipulation techniques
- Operate CRUD on a queue for handling data with First-In, First-Out (FIFO) principles
- Combine the functionalities of stacks and queues using a deque to master versatile data handling techniques

Introduction to Linear Data Structures

Linear Data Structures

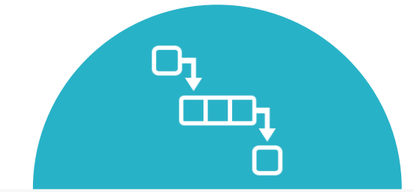
It stores elements in sequence, with each element connected to the next, allowing traversal in one pass.
Some common types are:

Array



Linked list

Stack



Queue

Understanding Arrays

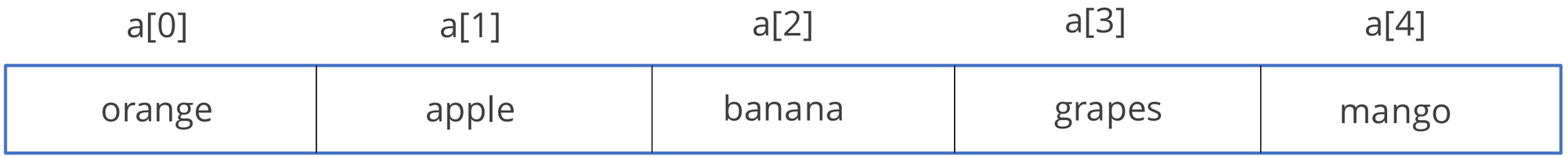
Arrays

Arrays collect elements of the same data type and store them in contiguous and adjacent memory locations.

Here is an example of an array with indexing:

Example:

```
let a= new Array("orange", "apple", "banana", "grapes", "mango");
```



They work on an index system starting from 0 to (n-1), where n is the total number of elements.

How to Declare Arrays in JavaScript?

Arrays can be declared using two methods:

Using array literal

Creates an array using square brackets and logs all elements to the console

Example:

```
let fruits = ["apple", "mango", "banana"];
console.log(fruits);
```

Output:

```
["apple", "mango", "banana"]
```

Using new keyword

Creates an array using the new Array() constructor and logs elements by index

Example:

```
let fruits = new Array("apple", "banana",
"mango");
console.log(fruits);
```

Output:

```
["apple", "mango", "banana"]
```

Accessing Elements in an Array

Arrays use indices, which are numerical positions, to retrieve elements. These indices can be categorized into:

Positive indices

Negative indices

Example:

```
let fruits = ["apple", "banana", "mango"];
console.log(fruits[0]);
console.log(fruits[1]);
console.log(fruits[2]);

Output:
apple
banana
mango
```

Example:

```
let fruits = ["apple", "banana", "mango"];
console.log(fruits[fruits.length - 1]);
console.log(fruits[fruits.length - 2]);
console.log(fruits[fruits.length - 3]);

Output:
mango
banana
apple
```

Positive indices are used to retrieve elements starting from index 0. Since JavaScript does not natively support negative indices, one should use the **.length** property to access elements in the reverse order.

Array Methods

The table below outlines the standard methods of the array object:

Method	Description
keys()	Returns an array iterator object with the keys of the original array
toString( )	Converts an array to a string and returns the result
entries( )	Provides a key/value pair array iterator object
reverse( )	Reverses the order of the elements in an array
indexOf( )	Searches the array for an element and returns its first index
isArray( )	Checks if the passed value is an array and returns true or false

Assisted Practice

Finding the Missing Number

Duration: 10 Min.

Problem statement:
You have been assigned a task to design an algorithm and build a JavaScript program that identifies a missing integer in a sequence from 1 to n for enhancing logical thinking and foundational algorithm skills.

Outcome:
By the end of this demo, you will be able to create an algorithm for identifying a missing number from the first **n** natural numbers.

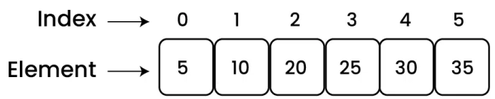
Note: Refer to the demo document for detailed steps:
01_Finding_the_Missing_Number

Assisted Practice: Guidelines

Steps to be followed:

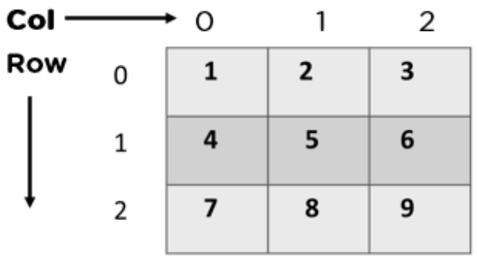
1. Create a working directory within the lab environment
2. Create a JavaScript file and execute it

Types of Arrays



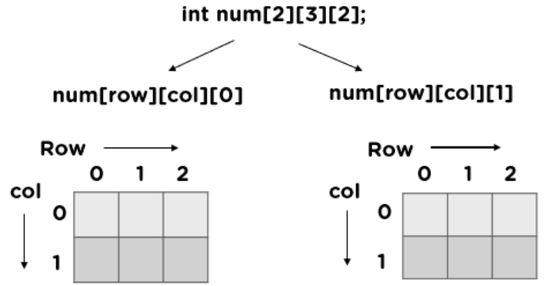
One-dimensional (1D) arrays

Stores elements in a single sequence, accessed using a single index



Two-dimensional (2D) arrays

Arranges elements in rows and columns, accessed using two indices



Three-dimensional (3D) arrays

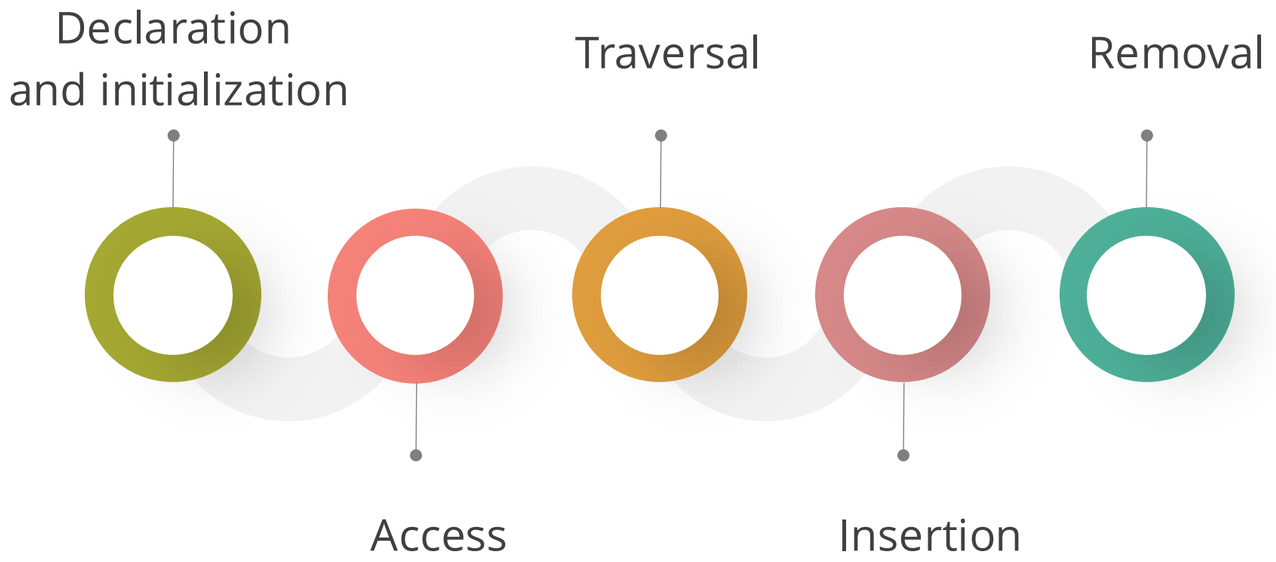
Extends 2D arrays by adding depth, accessed using three indices

Note: 2D Arrays will be covered in detail in this lesson as they are widely used in real-world applications

Operating 2D Arrays

Operating 2D Arrays

It involves steps for data storage, retrieval, and manipulation. The fundamental operations for an efficient 2D array are:



Declaration and Initialization of 2D Arrays

A 2D array represents a structured grid of elements, where memory is allocated for rows and columns.
The two primary ways to declare and initialize a 2D array are:

- Initializer list
- Dynamic assignment with loops

Initialization Using Initializer List

In JavaScript, literal notation provides a straightforward way to initialize a 2D array using square brackets to nest arrays within an array.

Example:

```
let Score = [
    ["Bar", 20, 60, "A"],
    ["Foo", 10, 52, "B"],
    ["Joey", 5, 24, "F"],
    ["John", 28, 43, "A"],
];
console.log(Score);
```

Output:

```
["Bar", 20, 60, "A"],
["Foo", 10, 52, "B"],
["Joey", 5, 24, "F"],
["John", 28, 43, "A"],
```

Initialization Using Loops

A 2D array in JavaScript can be initialized dynamically using nested loops, storing data in a structured format by iterating through rows and columns.

Example:

```
let sample = [];
let row = 3;
let col = 3;
let h = 0

// Loop to initialize 2D array
elements.
for (let i = 0; i < row; i++) {
    sample[i] = [];
    for (let j = 0; j < col;
j++) {
        sample[i][j] = h++;
    }
}
console.log(sample);
```

Output

```
[[0,1,2],
[3,4,5],
[6,7,8]]
```

Accessing Elements in 2D Arrays

In JavaScript, accessing elements in a 2D array requires two indices: the first index specifies the row, and the second index specifies the column.

Example:

```
let twoDArray = [
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
];

// Accessing elements
console.log(twoDArray[0][0]); // Outputs (first row, first column)
console.log(twoDArray[1][2]); // Outputs (second row, third column)
```

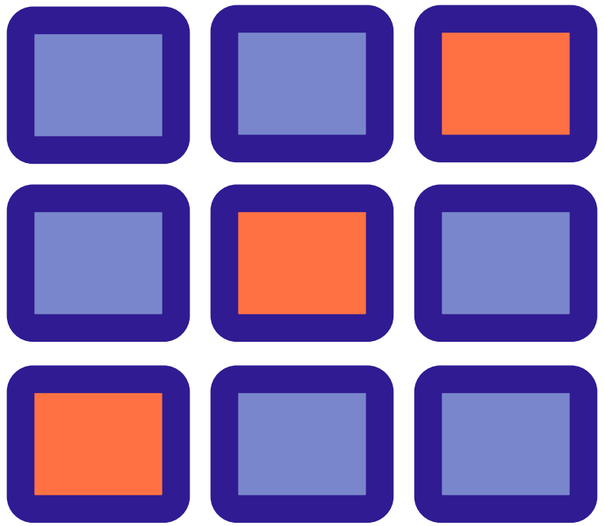
Output:

```
1
6
```

This allows direct retrieval of a specific element from the array.

Traversing Elements in a 2D Array

Traversing elements in a 2D array in JavaScript involves iterating through its elements using nested loops, where the outer loop moves through the rows, and the inner loop processes each column in that row.



```
let twoDArray = [
  [1, 2, 3],
  [4, 5, 6],
];

// Traverse the 2D array
for (let i = 0; i < twoDArray.length; i++) {
  for (let j = 0; j < twoDArray[i].length; j++)
{
    console.log(`Element at (${i}, ${j}):
${twoDArray[i][j]}`);
  }
}
```

```
Element at (0, 0): 1
Element at (0, 1): 2
Element at (0, 2): 3
Element at (1, 0): 4
Element at (1, 1): 5
Element at (1, 2): 6
```

Inserting Elements in 2D Arrays

To insert elements into a 2D array in JavaScript, users can use the **push** method to add a new row to the array or add elements to an existing row:

push()

Example:

```
let studentsData = [['Jack', 24], ['Sara', 23]];
studentsData.push(['Peter', 24]);
console.log(studentsData);
```

Output:

```
[["Jack", 24], ["Sara", 23], ["Peter", 24]
```

Example:

```
let studentsData = [['Jack', 24], ['Sara', 23]];
studentsData[1].push('hello'); // adding an
element to an existing row
console.log(studentsData);
```

Output:

```
[["Jack", 24], ["Sara", 23, "hello"]]
```

Inserting Elements in 2D Arrays

Users can also use **index notation** to assign values to specific positions or the **splice method** to insert elements at a specified index:

Index notation

Example:

```
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData[1][2] = 'hello'; // adding 'hello'
at the third position of the second sub-array
console.log(studentsData);
```

Output:

```
[["Jack", 24], ["Sara", 23, "hello"]]
```

splice()

Example:

```
let studentsData = [['Jack', 24], ['Sara', 23],];
studentsData.splice(1, 0, ['Peter', 24]);
// adding element at 1 index without deleting any
elements
console.log(studentsData);
```

Output:

```
[["Jack", 24], ["Peter", 24], ["Sara", 23]]
```

Removing an Element from 2D Arrays

Elements can be removed from a 2D array using the following methods:

Using **pop()** to remove from the end of a row:

Example:

```
let studentsData = [['Jack', 24], ['Sara', 23]];
studentsData[1].pop();
console.log(studentsData);
```

Output:

```
[["Jack", 24], ["Sara"]]
```

Using **pop()** to remove the last row:

Example:

```
let studentsData = [['Jack', 24], ['Sara', 23]];
studentsData.pop();
console.log(studentsData);
```

Output:

```
[["Jack", 24]]
```

Using **splice()** to remove a specific row:

Example:

```
let studentsData = [['Jack', 24], ['Sara', 23]];
studentsData.splice(1, 1);
console.log(studentsData);
```

Output:

```
[["Jack", 24]]
```

Assisted Practice

Working with 2D Arrays

Duration: 10 Min.

Problem statement:
You have been assigned a task to demonstrate essential methods for working with 2D arrays in JavaScript for effective data organization and manipulation.

Outcome:
By the end of this demo, you will be able to gain hands-on experience with 2D array manipulation in JavaScript.

Note: Refer to the demo document for detailed steps:
02_Working_with_2D_Array

Assisted Practice: Guidelines

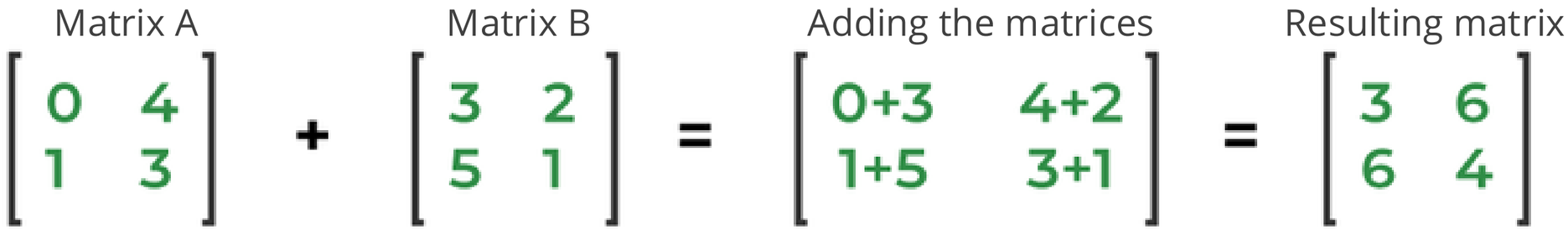
Steps to be followed:

1. Create a JavaScript file and execute it

Matrix Operations Using 2D Array

Matrix Addition

It is performed by adding the corresponding elements from two matrices of the same dimensions. Each element in the resulting matrix is the sum of elements from the same position in the original matrices.



Each number from Matrix A is added to its corresponding number in Matrix B to form a new, resulting matrix.

Implementing Matrix Addition in JavaScript

The function matrixAddition accepts two matrices as inputs and returns a new matrix containing the sums of their corresponding elements.

Example:

```
// Defining a function for adding two matrices.

function matrixAddition(matrixA, matrixB) {
    let result = [];  // Initialize the result matrix

    for (let i = 0; i < matrixA.length; i++) {  // Loop through
each row
        let row = [];  // Start a new row for the result
        for (let j = 0; j < matrixA[i].length; j++) {  // Loop
through each column
            row.push(matrixA[i][j] + matrixB[i][j]);  // Add
corresponding elements and store in the new row
        }
        result.push(row);  // Add the completed row to the result
matrix
    }
    return result;  // Return the resulting matrix of summed
elements
}
```

Example:

```
let matrixA = [[1, 2], [3, 4]];
let matrixB = [[5, 6], [7, 8]];
console.log(matrixAddition(matrixA,
matrixB));
```

Output:

```
[[6, 8],
[10, 12]]
```

Matrix Subtraction

It is performed by subtracting each element of one matrix from the corresponding element of another matrix of the same size.

$$A = \begin{bmatrix} 12 & -3 \\ 2 & 15 \end{bmatrix} \text{ and } B = \begin{bmatrix} 6 & 1 \\ 11 & -8 \end{bmatrix}$$

$$A - B = \begin{bmatrix} 12 - 6 & -3 - 1 \\ 2 - 11 & 15 - (-8) \end{bmatrix} = \begin{bmatrix} 6 & -4 \\ -9 & 23 \end{bmatrix}$$

Implementing Matrix Subtraction in JavaScript

To implement matrix subtraction in JavaScript, users can follow a similar approach to the one used for matrix addition. Here, the function **matrixSubtraction** is defined using 2D arrays:

Example:

```
function matrixSubtraction(matrixA, matrixB) {
    let result = []; // Initialize the resulting matrix
    for (let i = 0; i < matrixA.length; i++) {
        let row = []; // Initialize a new row for the result
        for (let j = 0; j < matrixA[i].length; j++) {
            row.push(matrixA[i][j] - matrixB[i][j]); // Subtract
corresponding elements
        }
        result.push(row); // Add the completed row to the result
matrix
    }
    return result; // Return the resulting matrix of subtracted
elements
}
```

Example:

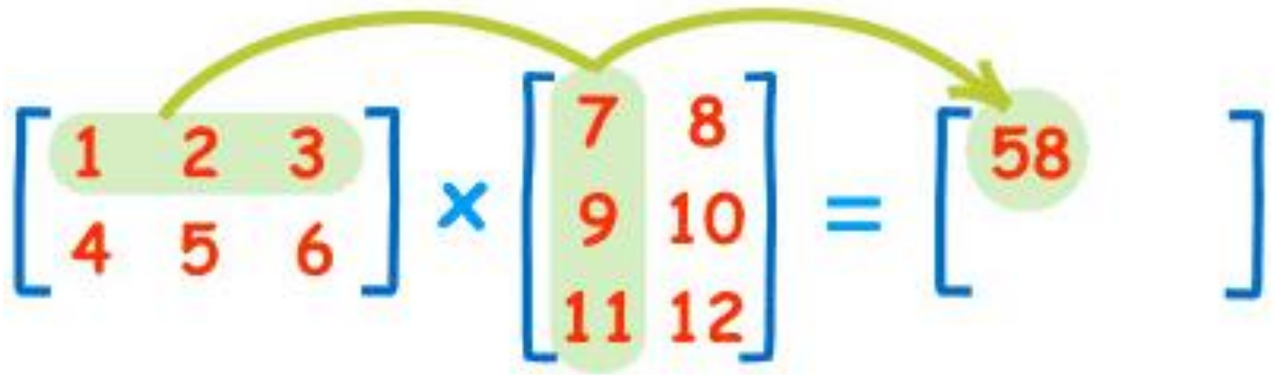
```
let matrixA = [[8, 5], [4, 3]];
let matrixB = [[3, 2], [2, 1]];
console.log(matrixSubtraction(matrixA,
matrixB));
```

Output:

```
[[5, 3],
[2, 2]]
```

Matrix Multiplication

It involves combining two matrices through a specific calculation between the rows of the first matrix and the columns of the second matrix. It results in a new matrix where each element is the sum of the products of corresponding elements.



Each element in the resulting matrix is computed by taking the dot product of the rows from Matrix A and the columns from Matrix B.

Implementing Matrix Multiplication in JavaScript

The JavaScript function **matrixMultiplication** demonstrates how to multiply two matrices by computing the dot products required for each element of the resulting matrix:

Example:

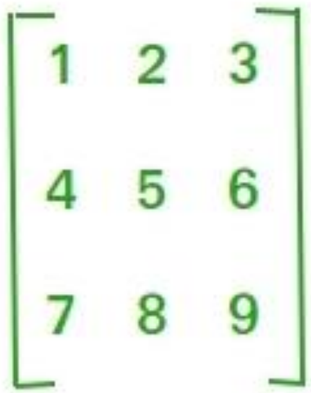
```
function matrixMultiplication(matrixA, matrixB) {
    const result = [];
for (let i = 0; i < matrixA.length; i++) {
        const row = [];
        for (let j = 0; j < matrixB[0].length; j++) {
            let sum = 0;
            for (let k = 0; k < matrixB.length; k++) {
                sum += matrixA[i][k] * matrixB[k][j];
            }
            row.push(sum);
        }
        result.push(row);
    }

    return result;
}
const resultMatrixMult = matrixMultiplication(matrixC,
matrixD);
console.log(resultMatrixMult);
```

Example:

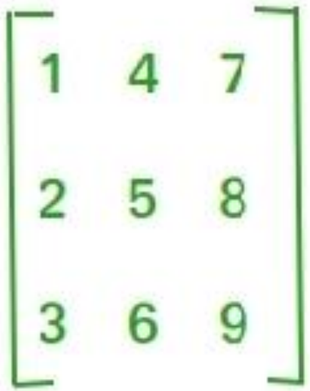
```
matrixC = [[1, 2], [3, 4]];
matrixD = [[5, 6], [7, 8]];

Output:
[[19, 22],
[43, 50]]
```

Matrix Transposition

It is the process of switching the rows and columns of a matrix to form a new matrix. This operation flips the matrix over its diagonal, turning the rows of the original matrix into columns in the new matrix.



Input



Output

Implementing Matrix Transposition Using 2D Arrays

Here is a JavaScript function with an example code demonstrating the transpose of a matrix:

Example:

```
function transposeMatrix(matrix) {
    const result = [];

    for (let i = 0; i < matrix[0].length; i++) {
        const row = [];
        for (let j = 0; j < matrix.length; j++) {
            row.push(matrix[j][i]);
        }
        result.push(row);
    }

    return result;
}

const transposedMatrix = transposeMatrix(matrixE);
console.log(transposedMatrix);
```

Example:

```
matrixE = [[1, 2, 3], [4, 5, 6]];
```

Output:

```
[[1, 4],
[2, 5],
[3, 6]]
```

Assisted Practice

Balancing an Array

Duration: 10 Min.

Problem statement:
You have been assigned a task to determine whether an even-length array can be split into two halves with equal sums and unique elements in each half, reinforcing practical skills in algorithm development and array evaluation.

Outcome:
By the end of this demo, you will be able to check if a given even-length array is balanced, that is, if it can be divided into two halves with equal sums and distinct elements.

Note: Refer to the demo document for detailed steps:
03_Balancing_an_Array

Assisted Practice: Guidelines

Steps to be followed:

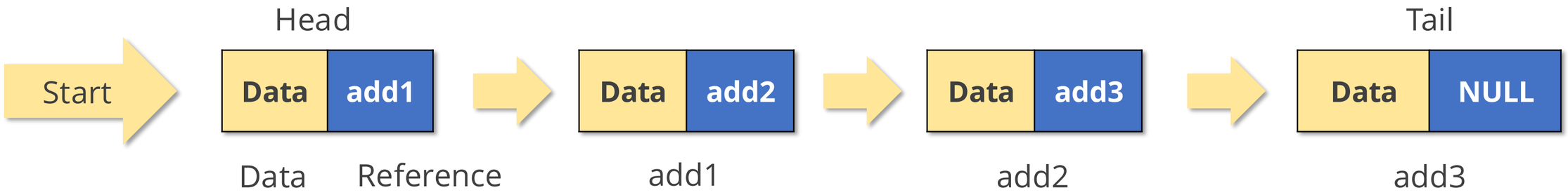
1. Create the algorithm and run it

Understanding Linked Lists

Linked List

It is a collection of elements called nodes, arranged sequentially. Each node links to the next through references. There are two main parts in each node:

1. Data → Stores the actual value
2. Reference to next node → Stores the address of the next node in the sequence



The **Head** of the list is the first node, while the **Tail** is the last node, which points to NULL, indicating the end of the list.

Initialization and Declaration of the Linked List

A linked list is initialized by creating a **LinkedList** class with a head pointer set to null. Nodes are declared using a **Node** class that stores data and a reference to the next node.

```
// User-defined class for a node
class Node {
  constructor(element) { // Constructor
    this.data = element;
    this.next = null;
  }
}

class LinkedList {
  // Constructor and methods
}

let myLinkedList = new LinkedList();
```

In JavaScript, a linked list is typically implemented using objects to represent nodes.

Types of Linked Lists

01 Singly linked list

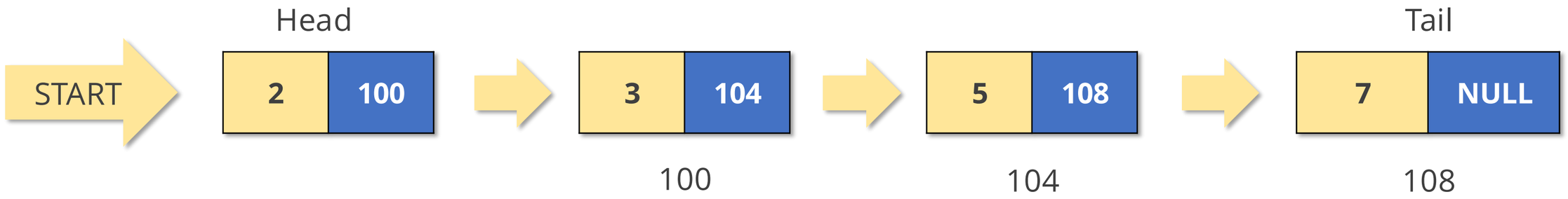
02 Doubly linked list

03 Circular linked list

04 Circular doubly linked list

Singly Linked List

It is a unidirectional data structure that allows traversal in only one direction, from the head to the tail.



Assisted Practice

Implementing CRUD Operations on a Singly Linked List

Duration: 10 Min.

Problem statement:
You have been assigned a task to create a singly linked list in JavaScript with CRUD functionalities such as node addition, traversal, value modification, and node deletion to reinforce understanding of dynamic data structures.

Outcome:
By the end of this demo, you will be able perform the CRUD operations on a singly linked list. The add() method adds a new node at the end of the list, the read() method traverses and prints the list, the update() method changes the value at a given position, and the delete() method removes a node at a specified position. These are essential operations when working with dynamic data structures.

Note: Refer to the demo document for detailed steps:
04_Implementing_CRUD_Operations_on_a_Singly_Linked_List

Assisted Practice: Guidelines

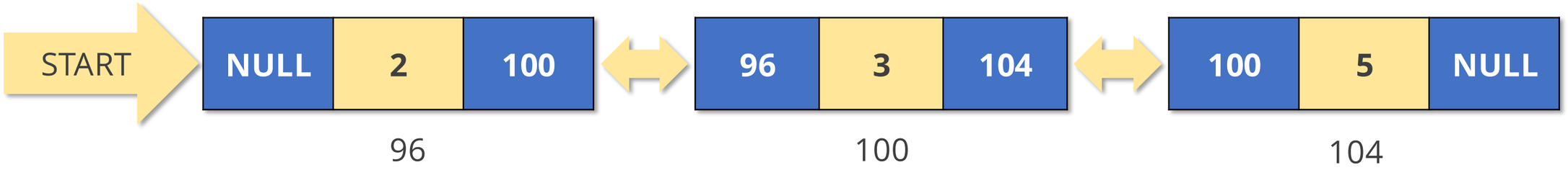
Steps to be followed:

1. Create a JavaScript file and execute it

Doubly Linked List

It is a linear data structure where each element or node contains data and two pointers: one pointing to the next node and another to the previous node.

Node



Assisted Practice

Implementing CRUD Operations on a Doubly Linked List

Duration: 10 Min.

Problem statement:
You have been assigned a task to create a doubly linked list in JavaScript with CRUD functionalities such as node addition, traversal, value modification, and deletion to enhance understanding of bidirectional data structures.

Outcome:
By the end of this demo, you will be able to perform the CRUD operations on a doubly linked list using JavaScript. This includes adding new nodes, traversing the list, updating node values, and deleting nodes, which are key operations for managing and manipulating bidirectional data structures.

Note: Refer to the demo document for detailed steps:
05_Implementing_CRUD_Operations_on_a_Doubly_Linked_List

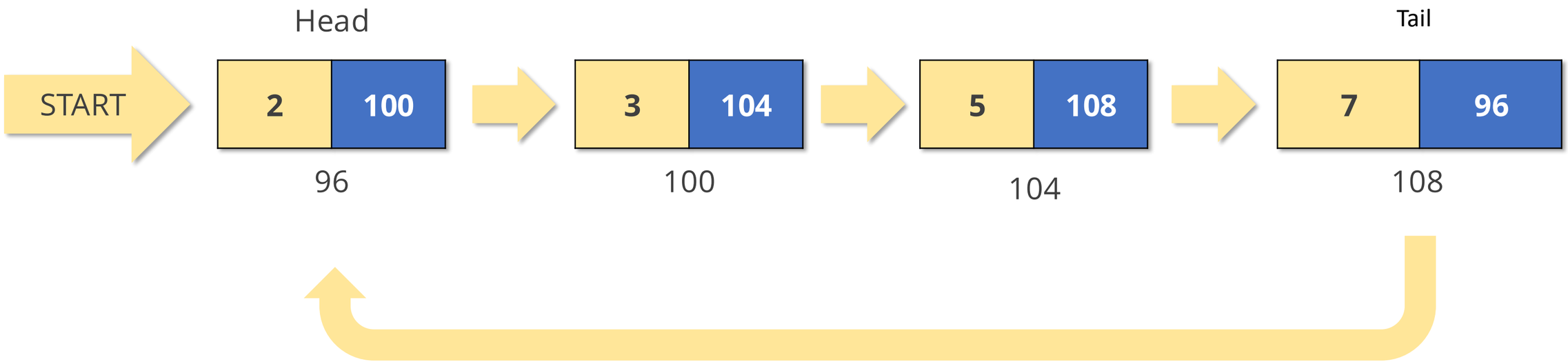
Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Circular Linked List

It is a data structure in which the nodes form a closed loop, where the last node returns to the first node, creating a circular structure.



Assisted Practice

Implementing CRUD Operations on a Circular Linked List

Duration: 10 Min.

Problem statement:
You have been assigned a task to implement a circular linked list in JavaScript, with CRUD functionalities including node addition, traversal, value modification, and node deletion to strengthen the understanding of circular data structure operations.

Outcome:
By the end of this demo, you will be able to perform the CRUD operations on a circular linked list. In this process, the add() method appends a new node at the end of the list, the read() method traverses and prints the list, the update() method modifies the value at a given position, and the delete() method removes a node at a specified position.

Note: Refer to the demo document for detailed steps:
06_Implementing_CRUD_Operations_on_a_Circular_Linked_List

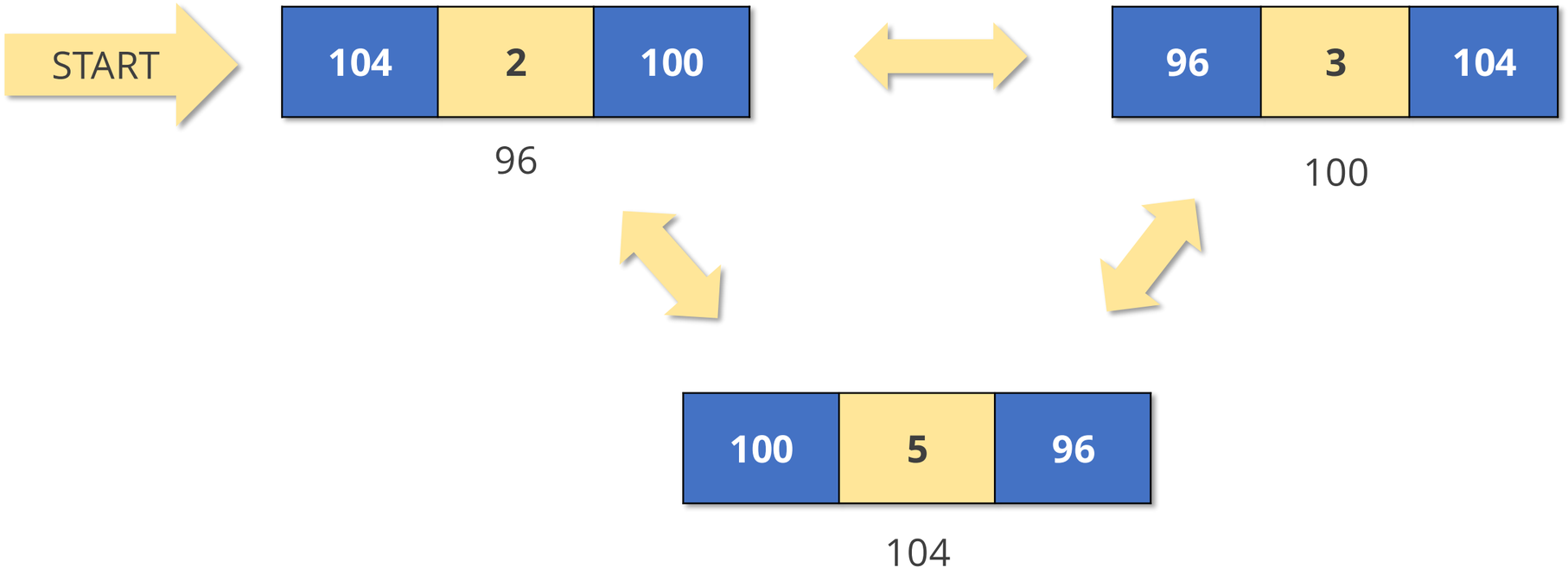
Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Circular Doubly Linked List

It is a data structure where each node contains two pointers: one pointing to the next node and another to the previous node, with the last node linking back to the first node, forming a circular structure.



It is a bidirectional linked list that can be traversed in both directions.

Assisted Practice

Detecting a Cycle in a Linked List

Duration: 10 Min.

Problem statement:
You have been assigned a task to determine whether a linked list contains a cycle by implementing a JavaScript solution that uses pointer-based traversal to enhance system stability, prevent resource leakage, and maintain data integrity in software applications.

Outcome:
By the end of this demo, you will be able to implement a JavaScript solution to detect cycles in a linked list.

Note: Refer to the demo document for detailed steps:
07_Detecting_a_Cycle_in_a_Linked_List

Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Assisted Practice

Implementing CRUD Operations on a Circular Doubly Linked List

Duration: 10 Min.

Problem statement:
You have been assigned a task to implement a circular doubly linked list in JavaScript with CRUD operations including node addition, traversal, value modification, and deletion, reinforcing understanding of advanced linked list structures.

Outcome:
By the end of this demo, you will be able to executed CRUD operations on a circular doubly linked list. The add() method inserts a new node at the end, the read() method traverses and prints the list, the update() method modifies the value at a specified position, and the delete() method removes a node from the given position.

Note: Refer to the demo document for detailed steps:
08_Implementing_CRUD_Operations_on_a_Circular_Doubly_Linked_List

Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Operating Linked Lists

Linked List Operations

There are two operations to perform on a linked list:

Insertion

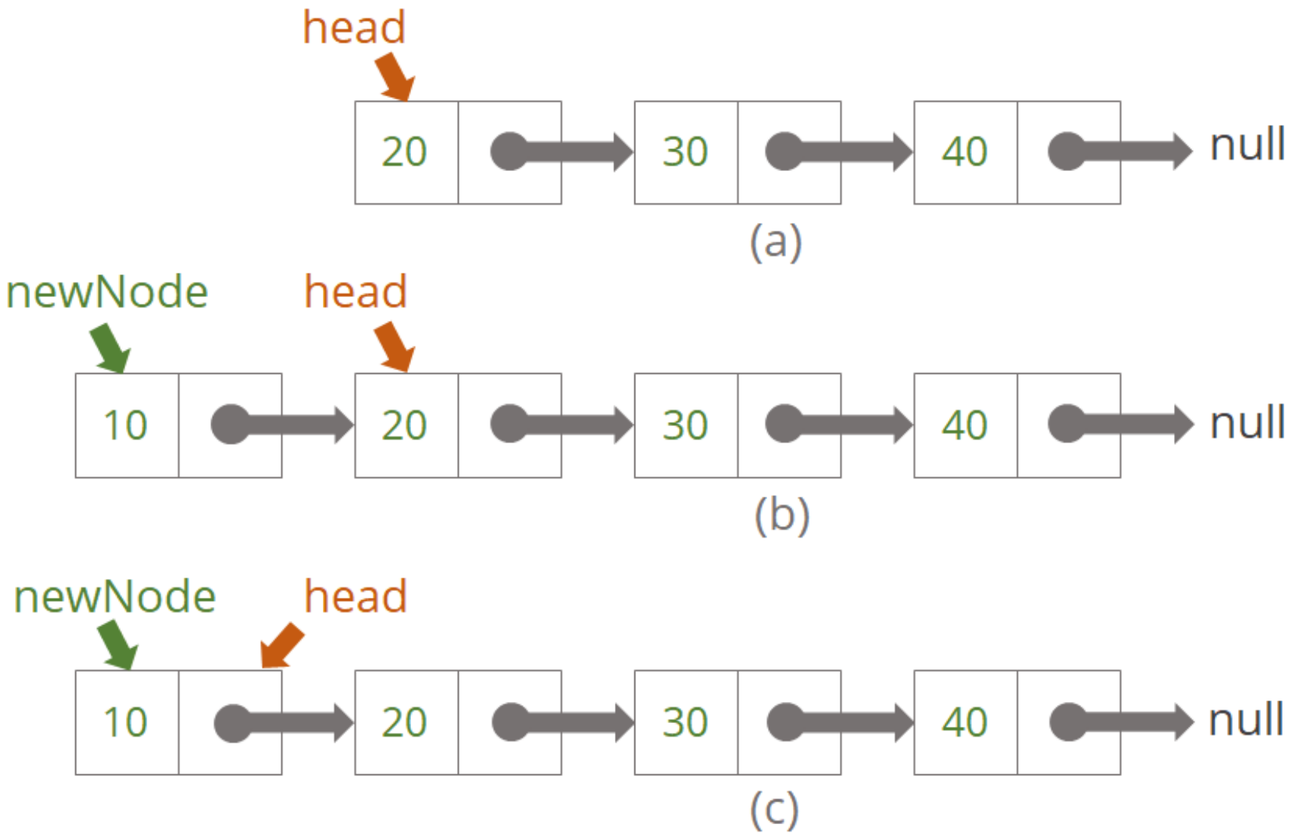
- At the beginning
- At the end
- At a specific position

Deletion

- At the beginning
- At the end
- At a specific position

Singly Linked List Insertion: At the Beginning

To insert a new element at the beginning of a singly linked list, a new node must be created. Then, the head of the list must be updated to point to this new node.



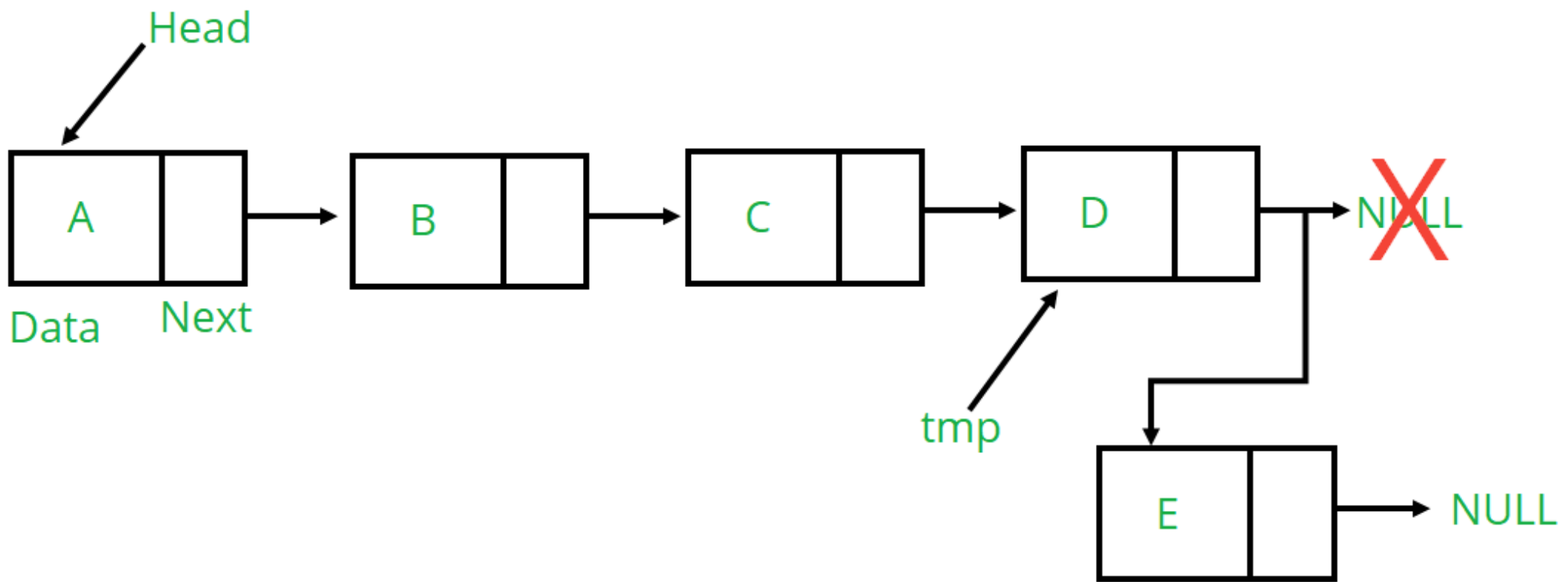
Singly Linked List Insertion: At the Beginning

Here is a JavaScript code snippet demonstrating how to insert an element at the beginning of a singly linked list:

```
function insertAtHead(linkedList, value) {
    const newNode = new Node(value);// Create a new node
    newNode.next = linkedList.head; // Point new node to the
current head
    linkedList.head = newNode;// Set new node as the new head
}
```

Singly Linked List Insertion: At the End

Appending a new element to the end of a singly linked list requires traversing to the last node, creating a new node that points to null, and updating the last node's next pointer to reference the new node.



This diagram shows the process of adding a new node **E** at the end of the list. **tmp** is used to find the last node, **D**, which is then linked to **E**.

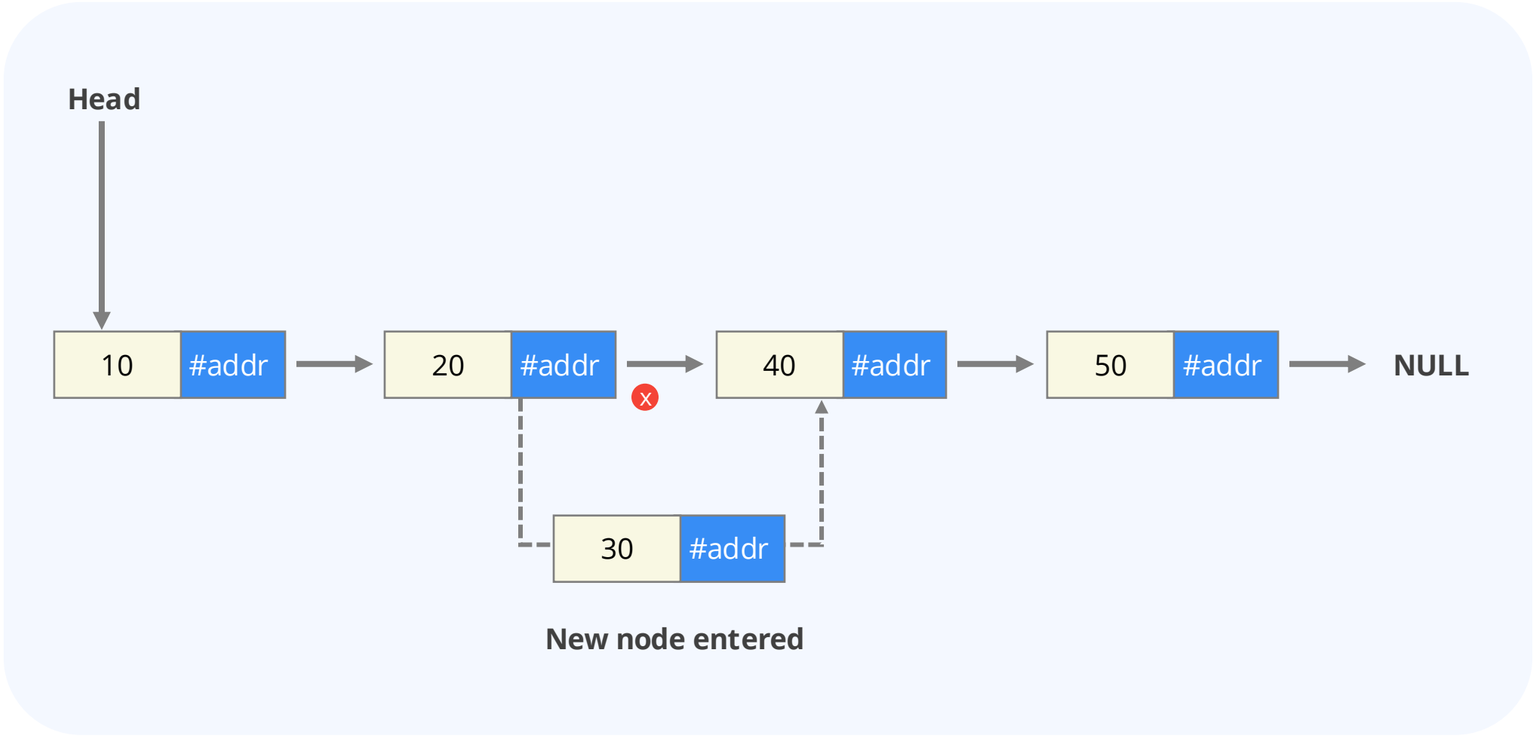
Singly Linked List Insertion: At the End

Here is the JavaScript code for inserting an element at the end of a singly linked list:

```
function append(linkedList, value) {
    const newNode = new Node(value);  // Create
a new node

    // If the linked list is empty, make the new
node the head
    if (!linkedList.head) {
        linkedList.head = newNode;
    } else {
        // Traverse the list to find the last
node
        let current = linkedList.head;
        while (current.next) {
            current = current.next;
        }

        // Link the last node to the new node
        current.next = newNode;
    }
}
```

Singly Linked List Insertion: At a Specific Position

To insert a node at a specific position in a singly linked list, the user must traverse the list to (position - 1) and adjust the pointers to place the new node between the current and next nodes.



Singly Linked List Insertion: At a Specific Position

Here is the code for inserting an element at the specific position of a singly linked list in JavaScript:

```
function insertAtPosition(linkedList, value, position) {
    const newNode = new Node(value); // Create a new node with the
given value
    let current = linkedList.head; // Start at the head of the list
    // Traverse the list to the position just before where you want to
insert
    for (let i = 1; i < position - 1; i++) {
        if (current == null) {
            console.error("Position out of bounds.");
            return;
        }
        current = current.next;
    }
    // Insert the new node
    newNode.next = current.next; // Link new node to the next node in
the list
    current.next = newNode; // Link the current node to the new node
}
```

Quick Check

Emma is managing a product list using a singly linked list. She needs to add a new product at a specific position.

Which of the following steps correctly inserts the new product in the list?

A. Set the new node’s next to null and update the head

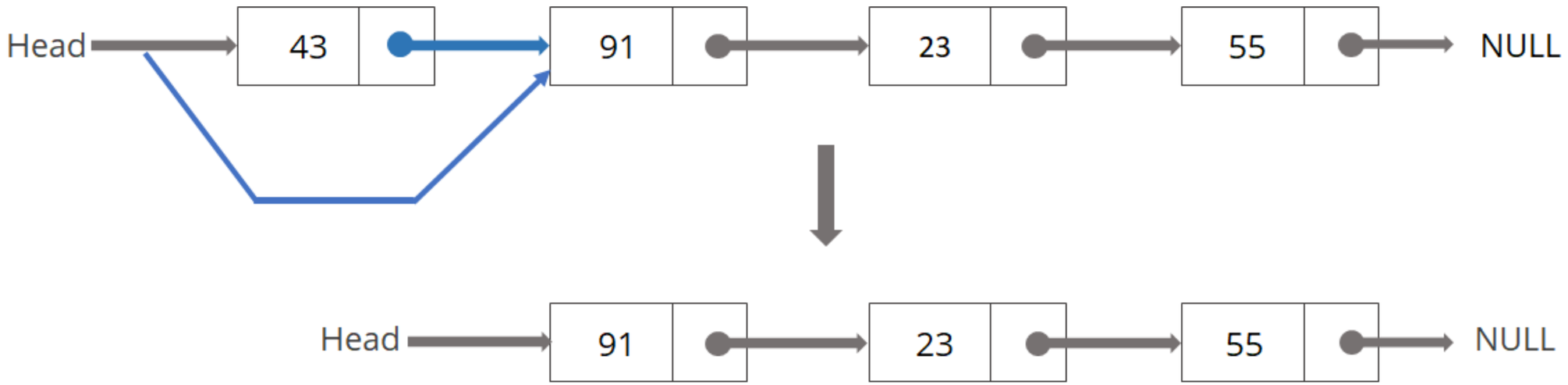
B. Find the position, update the previous node’s next to the new node, and link the new node to the next one

C. Always add the new node at the end

D. Replace the existing product at that position with the new one

Singly Linked List Deletion: At the Beginning

Removing the first element from a singly linked list requires updating the head pointer to the second node, effectively detaching the first node.



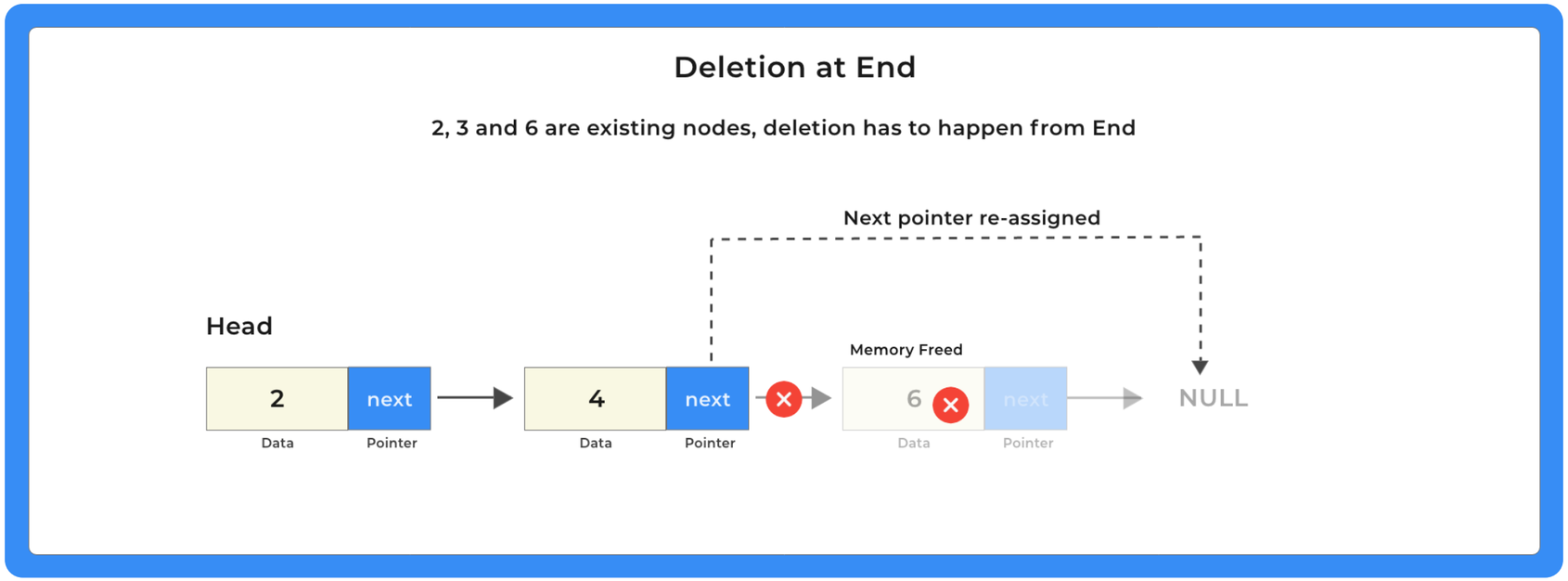
Singly Linked List Deletion: At the Beginning

Here is the JavaScript code for deleting an element at the beginning of a singly linked list:

```
function deleteAtHead(linkedList) {
    // Check if the list is not empty
    if (linkedList.head) {
        // Update the head to point to the
second node, effectively removing the first node
        linkedList.head = linkedList.head.next;
    }
}
```

Singly Linked List Deletion: at the End

Removing the last element from a singly linked list requires traversing to the second-to-last node and updating its next pointer to null, detaching and freeing the last node.



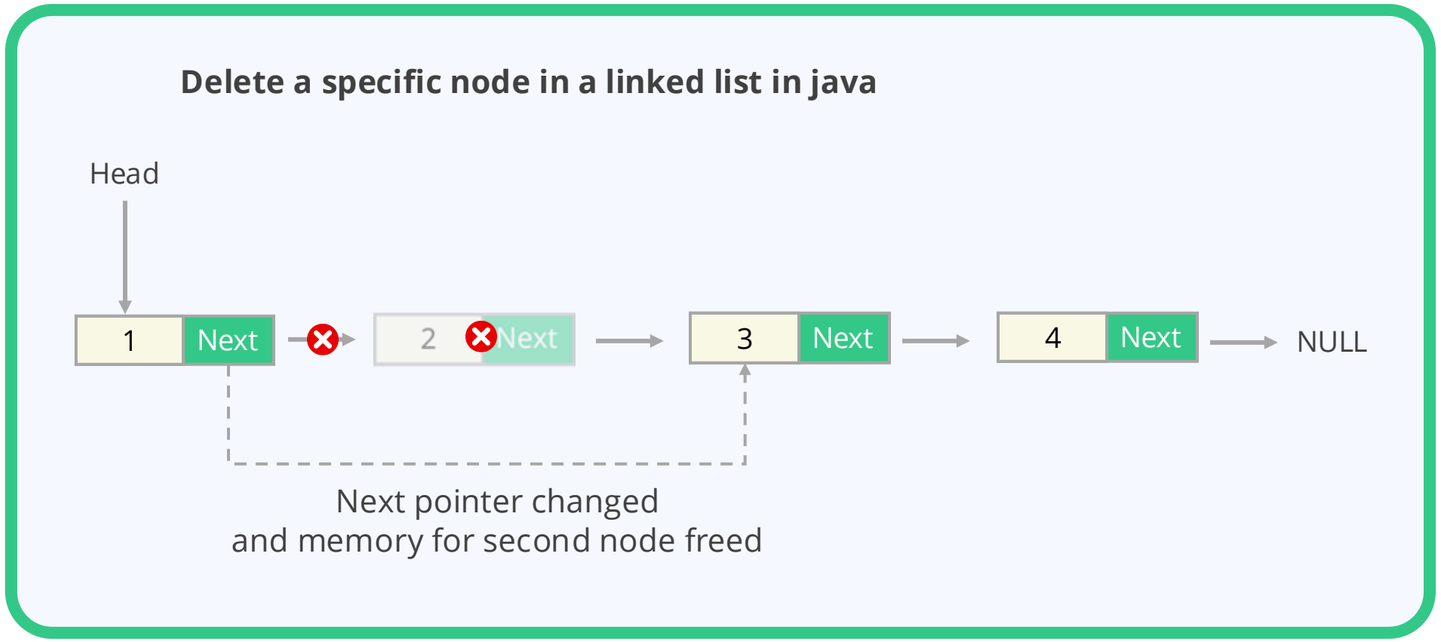
Singly Linked List Deletion: at the End

Here is the JavaScript code for deleting an element at the end of a singly linked list:

```
function deleteAtEnd(linkedList) {
    // Check if the list is empty
    if (!linkedList.head) {
        return;  // Exit if the list is empty, nothing to delete
    }
    // Check if the list has only one node
    if (!linkedList.head.next) {
        linkedList.head = null;  // Remove the only node by setting head to null
        return;
    }
    let current = linkedList.head;  // Start at the head of the list
    // Traverse the list to find the second-to-last node
    while (current.next.next) {
        current = current.next;  // Move to the next node
    }
    current.next = null;  // Set the second-to-last node's next to null, removing
the last node
}
```

Singly Linked List Deletion: at a Specific Position

Deleting a node at a specific position in a singly linked list requires locating the preceding node and updating its next pointer to skip the target node, effectively unlinking and freeing it.



Singly Linked List Deletion: At a Specific Position

Here is the JavaScript code for deleting an element-specific position of a singly linked list:

```
function deleteAtPosition(linkedList, position) {
    // Check if the list is empty
    if (!linkedList.head) {
        return;  // If list is empty, exit function
    }
    // Special case to delete the head node
    if (position === 1) {
        linkedList.head = linkedList.head.next;  // Update the head to the next node
        return;
    }
    let current = linkedList.head;
    // Traverse to the node just before the one to be deleted
    for (let i = 1; i < position - 1 && current; i++) {
        current = current.next;
    }
    // Check if the position is valid (not out of bounds)
    if (!current || !current.next) {
        console.error("Position out of bounds.");  // Error message if position is invalid
        return;
    }
    // Bypass the node at the given position
    current.next = current.next.next;  // Link the current node to the node after the next node
}
```

Assisted Practice

Merging Two Sorted Linked Lists

Duration: 10 Min.

Problem statement:
You have been assigned a task to write a function that merges two sorted linked lists into a single sorted linked list by splicing together their nodes, for practicing efficient list manipulation and understanding merging techniques used in algorithms.

Outcome:
By the end of this demo, you will be able to combine two sorted linked lists into a single sorted list. This implementation is concise and leverages the existing order of the lists to minimize operations.

Note: Refer to the demo document for detailed steps:
09_Merging_two_sorted_linked_list

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Quick Check

Emma, a junior MERN stack developer, is implementing a JavaScript-based inventory management system for an e-commerce store. The product list is stored as a single linked list, each node representing a product. When a product is sold out, she needs to remove it from the end of the list.

Which of the following steps correctly describe how Emma should delete the last product (node) from the linked list?

A. Traverse the list to find the last node and set it to **null**

B. Traverse the list to find the second-to-last node and set its **next** pointer to **null**

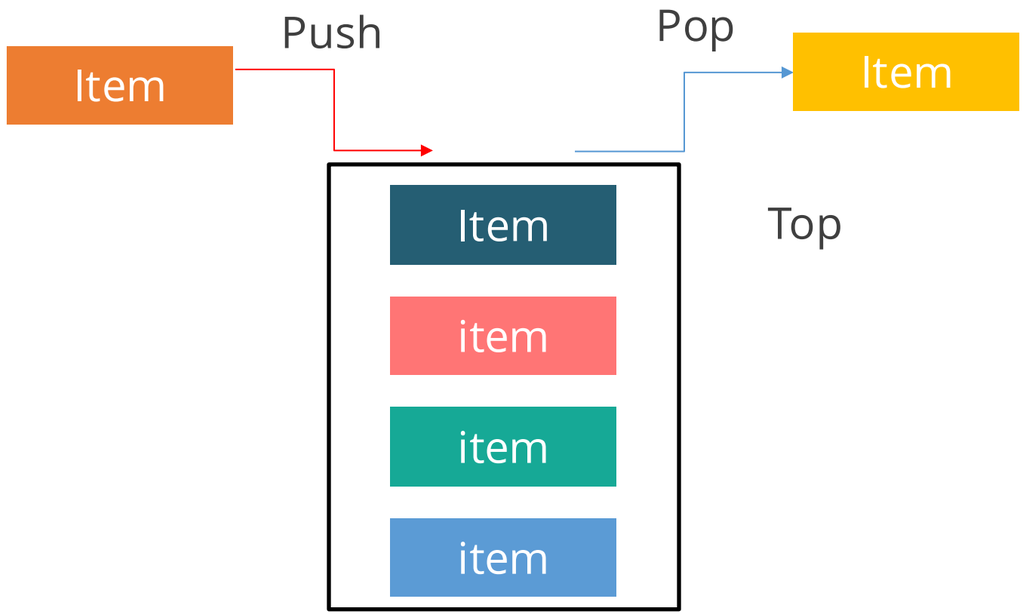
C. Directly set **linkedList.head** to **null**, regardless of the list size

D. Always delete the head node, as it represents the last element in a singly linked list

Understanding Stacks

Stack

It is a linear structure where items are added and removed from the top, following a Last-In, First-Out (LIFO) order.



Example

A stack of plates in a cafeteria: The last plate placed on the stack is the first one taken.

Stack Operations

Operations	Description
push()	The push operation adds an element to the top of the stack.
pop()	The pop operation removes the element from the top of the stack.
peek()	The peek operation retrieves the element at the top of the stack without removing it.
isEmpty()	The isEmpty operation checks if the stack is empty.
getSize()	The getSize operation returns the number of elements in the stack.

Stack Operations

The following code includes various stack operations:

Example:

```
let myStack = [];

push(myStack, 10);
push(myStack, 20);
push(myStack, 30);

console.log("Stack:", myStack);
console.log("Peek:", peek(myStack)); // 30
console.log("Pop:", pop(myStack)); // 30
console.log("Stack:", myStack);
console.log("isEmpty:", isEmpty(myStack)); // false
console.log("getSize:", getSize(myStack)); // 2
```

Output:

```
Stack: [10, 20, 30]
Peek: 30
Pop: 30
Stack: [10, 20]
isEmpty: false
getSize: 2
```

Assisted Practice

Implementing CRUD Operations on a Stack

Duration: 10 Min.

Problem statement:
You have been assigned a task to implement CRUD operations on a stack, showcasing how to push, pop, peek, display, and clear elements within a stack structure for strengthening understanding of linear data structures and practical stack manipulation.

Outcome:
By the end of this demo, you will be able to manage a stack in JavaScript through various CRUD operations, enhancing your ability to manipulate data structures and apply fundamental programming concepts in practical scenarios.

Note: Refer to the demo document for detailed steps:
10_Implementing_CRUD_Operations_on_a_Stack

Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Implementation of Stacks Using the Arrays

Here is the code that includes the implementation of a stack using the arrays, including methods for adding, removing, viewing the top element, and checking if the stack is empty or clearing it:

Example:

```
class Stack {
    constructor() {
        this.items = [];
    }
push(element) {
        this.items.push(element);// Push
element onto the stack

    }
pop() {
        if (this.isEmpty()) {
            console.error("Stack underflow:
Cannot pop from an empty stack."); // Pop
element from the stack
            return;
}
return this.items.pop();
 }
```

Example:

```
peek() {// Peek operation: Return the top
element without removing it
    if (this.isEmpty()) {
      return "Stack is empty";
    }
    return this.items[this.items.length - 1];
  }
isEmpty() {// Check if the stack is empty
    return this.items.length === 0;
  }
size() {// Get the size (number of elements) of
the stack
    return this.items.length;
  }
clear() { // Clear the stack
this.items = [];}}
```

Implementation of Stacks Using the Arrays

Here is the code that implements a stack using the arrays:

Example:

```
// Example usage:
const stack = new Stack();

stack.push(10);
stack.push(20);
stack.push(30);

console.log("Stack:", stack.items); // [10, 20, 30]
console.log("Size:", stack.size()); // 3
console.log("Top:", stack.peek()); // 30

stack.pop();
console.log("Stack after pop:", stack.items); // [10, 20]
```

Output:

```
Stack: [10, 20, 30]
Size: 3
Top: 30
Stack after pop: [10, 20]
```

Assisted Practice

Implementing Stacks Using Arrays

Duration: 10 Min.

Problem statement:
You have been assigned a task to implement a stack using arrays in JavaScript, covering key operations such as push, pop, peek, and display to enhance data structure manipulation skills.

Outcome:
By the end of this demo, you will be able to implement a stack using arrays in JavaScript, covering essential operations like push, pop, peek, and displaying the stack's contents, while strengthening your understanding of data structure manipulation.

Note: Refer to the demo document for detailed steps:
11_Implementing_Stacks_Using_Array

Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Implementation of Stacks Using a Linked List

The given code demonstrates managing a stack in JavaScript using a linked list structure, detailing operations like adding, removing, and viewing the top element, as well as checking if the stack is empty:

Example:

```
class Node {
    constructor(data) {
        this.data = data;
        this.next = null;
    }
}
class Stack {
    constructor() {
        this.top = null;
        this.size = 0;
    }
push(data) {// Push element onto the stack
        const newNode = new Node(data);
        newNode.next = this.top;
        this.top = newNode;
        this.size++; }
```

Example:

```
pop() {// Pop operation: Remove and return the
top element from the stack
    if (this.isEmpty()) {
      return "Underflow: Stack is empty";
    }
    const poppedData = this.top.data; // Get
the data from the top node
    this.top = this.top.next; // Move the top
pointer to the next node
    return poppedData;
  }
peek() {// Peek operation: Return the top
element without removing it
    if (this.isEmpty()) {
      return "Stack is empty";
    }}
```

Implementation of Stacks Using the Linked List

The given code shows how to manage a stack with a linked list in JavaScript, including push, pop, peek, and clear operations.

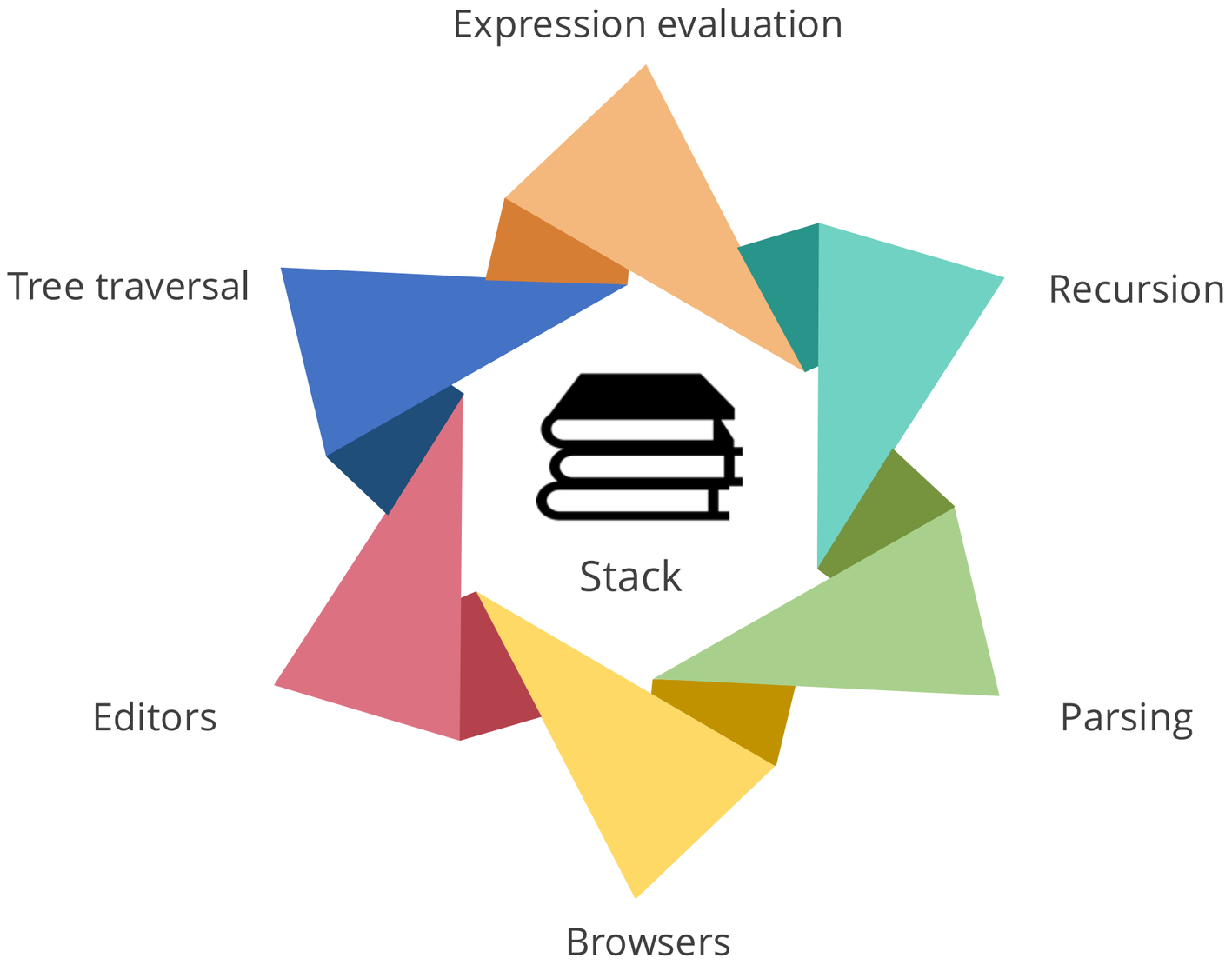
Example:

```
 isEmpty() {
    return this.top === null;
  }
clear() {// Clear the stack
    this.top = null;
  }
}
const stack = new Stack();
stack.push(10);
stack.push(20);
stack.push(30);
console.log("Stack:", stack); // Display the stack
console.log("Top:", stack.peek()); // Peek at the top element
console.log("Popped:", stack.pop()); // Pop the top element
```

Output:

```
Stack: Stack { top: Node { data: 30,
next: Node { data: 20, next: Node {
data: 10, next: null } } } }
Top: 30
Popped: 30
```

Applications of Stacks



Assisted Practice

Implementing Stacks Using a Linked List

Duration: 10 Min.

Problem statement:
You have been assigned a task to demonstrate the implementation of a stack using a linked list in JavaScript, covering operations like push, pop, peek, display, and clear to develop a strong grasp of dynamic data structure manipulation.

Outcome:
By the end of this demo, you will be able to master the implementation and manipulation of a stack using a linked list in JavaScript, covering operations like push, pop, peek, display, and clear and gaining valuable insights into dynamic data structure operations.

Note: Refer to the demo document for detailed steps:
12_Implementing_Stacks_Using_a_Linked_List

Assisted Practice: Guidelines

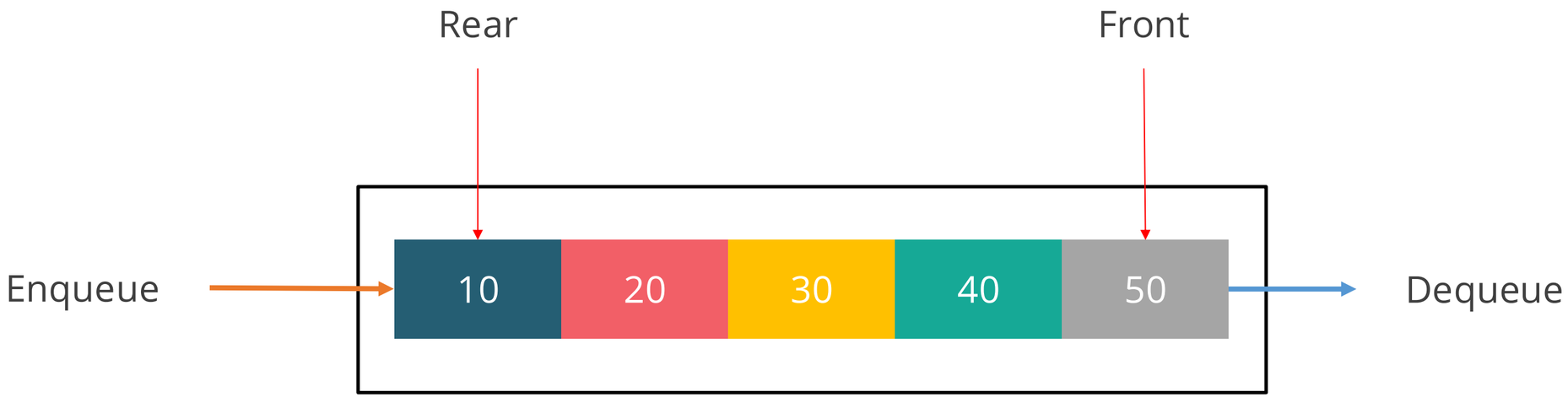
Steps to be followed:

1. Create a JavaScript file and execute it

Understanding Queue

Queue

It is a linear structure where items are added at the back end, called the rear, and removed from the front, following a First-In, First-Out (FIFO) order.



Example

Supermarket checkouts: Customers line up in order, and the first person in line is served first.

Operations on a Queue

Queue

Enqueue(): Insertion of elements to the queue

Dequeue(): Removal of elements from the queue

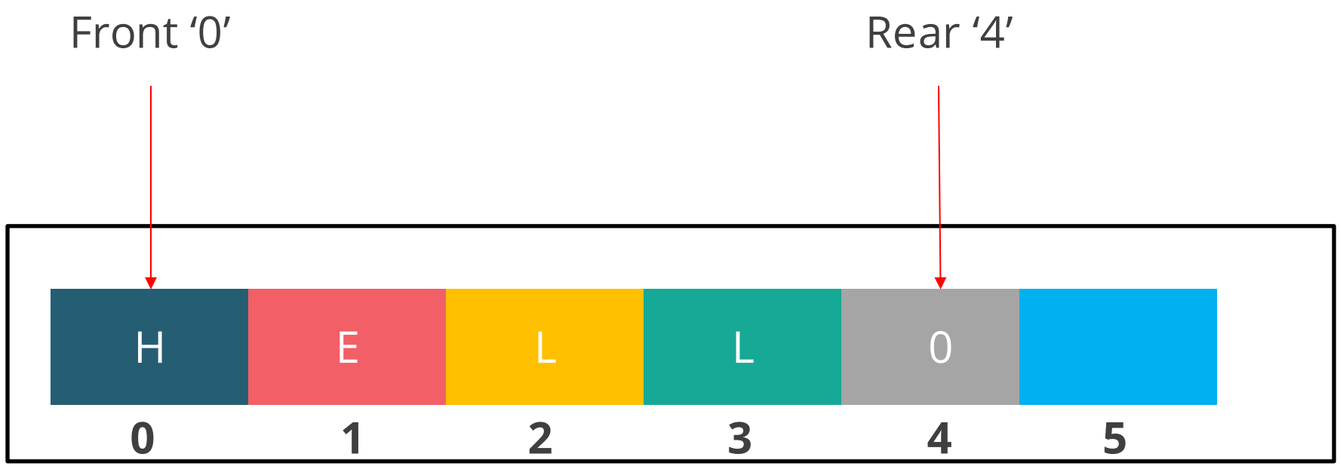
Peek(): Acquires the data element available at the front node of the queue without deleting it

isFull(): Validates if the queue is full

isNull(): Checks if the queue is empty

Operations on a Queue

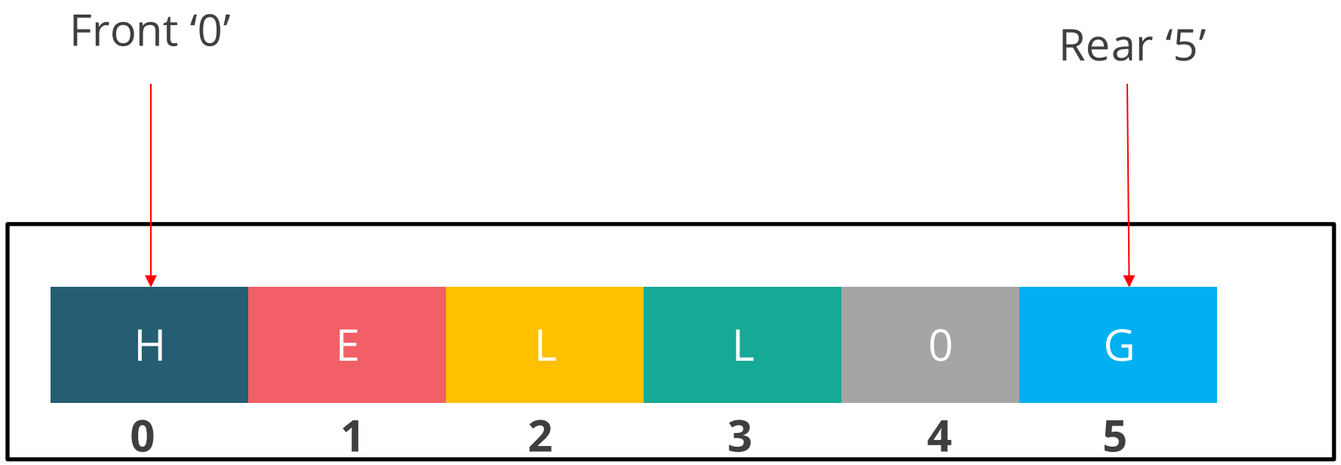
The queue before inserting an element:



Queue

Operations on a Queue

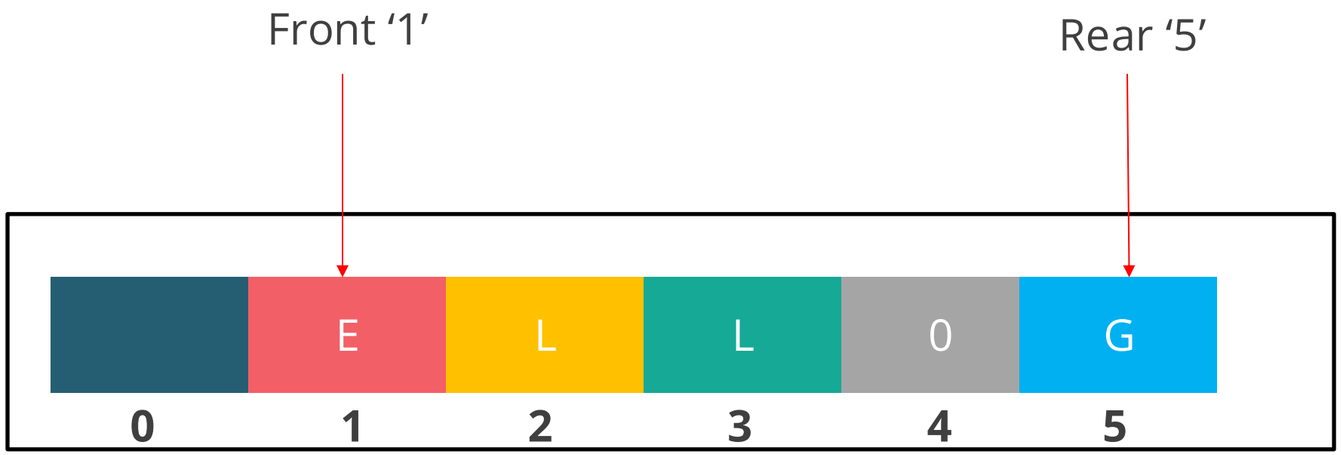
The queue after inserting an element:



Queue

Operations on a Queue

The queue after deleting an element:



Queue

Assisted Practice

Implementing CRUD Operations on a Queue

Duration: 10 Min.

Problem statement:
You have been assigned a task to implement CRUD operations on a queue in JavaScript, including enqueue, dequeue, accessing elements, and checking the queue's size and emptiness, to reinforce understanding of linear data structures.

Outcome:
By the end of this demo, you will be able to implement and execute CRUD operations on a queue in JavaScript, enhancing your ability to manage and interact effectively with this fundamental data structure.

Note: Refer to the demo document for detailed steps:
13_Implementing_CRUD_Operations_on _a _Queue

Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Implementation of a Queue Using the Arrays

The following code implements a queue using the arrays, featuring methods to add, remove, peek, and assess the size and emptiness of the queue.

Example:

```
class Queue {
  constructor() {
    this.items = []; // Array to store elements
  }

  // Enqueue operation: Add an element to the
rear of the queue
  enqueue(element) {
    this.items.push(element);
  }
// Dequeue operation: Remove and return the
element from the front of the queue
  dequeue() {
    if (this.isEmpty()) {
      return "Underflow: Queue is empty";
    }    return this.items.shift();}
```

Example:

```
 // Peek operation: Return the front element
without removing it
  peek() {
    if (this.isEmpty()) {
      return "Queue is empty";
    }
    return this.items[0];
  }
// Check if the queue is empty
  isEmpty() {
    return this.items.length === 0;
  }
// Get the size (number of elements) of the
queue
  size() {
    return this.items.length;}
```

Implementation of a Queue Using the Arrays

The following example demonstrates the implementation of a queue using arrays:

Example:

```
// Example usage:
const queue = new Queue();

queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);

console.log("Front:", queue.peek()); // Peek at the front
element
console.log("Dequeued:", queue.dequeue()); // Dequeue the front
element
console.log("Queue size:", queue.size()); // Get the current
size of the queue
```

Output:

```
Front: 10
Dequeued: 10
Queue size: 2
```

Implementation of a Queue Using a Linked List

The following example demonstrates the code that implements a queue using a linked list:

Example:

```
class Node {
  constructor(data) {
    this.data = data; // Data stored in the
node
    this.next = null; // Pointer to the next
node
  }
}
class Queue {
  constructor() {
    this.front = null; // Initialize the front
of the queue as null (empty)
    this.rear = null; // Initialize the rear of
the queue as null (empty)
    this.size = 0; // Initialize the size of
the queue as 0
```

Example:

```
enqueue(data) {// Enqueue operation: Add an
element to the rear of the queue
    const newNode = new Node(data); // Create a
new node
    if (this.isEmpty()) {
      // If the queue is empty, set both front
and rear to the new node
      this.front = newNode;
      this.rear = newNode;
    } else {
      // Otherwise, update the rear to the new
node
      this.rear.next = newNode;
      this.rear = newNode;
    }
    this.size++; // Increment the size }
```

Implementation of a Queue Using a Linked List

The following example demonstrates the implementation of a queue using a linked list:

Example:

```
 // Dequeue operation: Remove and return the
element from the front of the queue
  dequeue() {
    if (this.isEmpty()) {
      return "Underflow: Queue is empty";
    }
    const removedData = this.front.data; // Get
the data from the front node
    this.front = this.front.next; // Move the
front pointer to the next node
    this.size--; // Decrement the size
    if (this.isEmpty()) {
      // If the queue becomes empty, also
update the rear to null
      this.rear = null;
    } return removedData;}}
```

Example:

```
peek() {// Peek operation: Return the front
element without removing it
    if (this.isEmpty()) {
      return "Queue is empty";
    }
    return this.front.data;
  }
isEmpty() {// Check if the queue is empty
    return this.size === 0;
  }

  // Get the size (number of elements) of the
queue
  getSize() {
    return this.size;
  }}
```

Implementation of a Queue Using a Linked List

The following example demonstrates the implementation of a queue using a linked list:

Example:

```
// Example usage:
const queue = new Queue();

queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);

console.log("Front:", queue.peek()); // Peek at
the front element
console.log("Dequeued:", queue.dequeue()); //
Dequeue the front element
console.log("Queue size:", queue.getSize()); //
Get the current size of the queue
```

Output:

```
Front: 10
Dequeued: 10
Queue size: 2
```

Applications of Queue

To perform asynchronous data transfer

To achieve synchronization in CPUs

To handle interrupts in operating systems

To assign waiting lists for single-shred resources like a printer or a disk

To add or remove the songs in a playlist

To arrange the data in a sequence in an mp3 or a CD player

Assisted Practice

Implementing Stack and Queue using Deque

Duration: 10 Min.

Problem statement:
You have been assigned a task to demonstrate the implementation of both stack and queue functionalities using a deque (double-ended queue) in JavaScript, showcasing its versatility in supporting multiple linear data operations.

Outcome:
By the end of this demo, you will be able to implement stack and queue operations using a deque in JavaScript, broadening your understanding of versatile data structures and their applications in programming.

Note: Refer to the demo document for detailed steps:
14_Implementing_Stack_and_Queue_using_Deque

Assisted Practice: Guidelines

Steps to be followed:

1. Create a JavaScript file and execute it

Key Takeaways

- A two-dimensional array is structured as a matrix, represented by a collection of rows and columns.
- A linked list is a linear data structure where elements can be traversed using pointers. Each element, called a node, consists of two parts: data and a reference to the next node, allowing for dynamic data management.
- Stack is a linear structure where items are added and removed from the top, following a Last-In, First-Out (LIFO) order.
- A queue is a linear structure that enables the user to insert the elements from the REAR end and delete them from the FRONT end.

Thank You