

Data Structures and Algorithms



Linear Data Structures



A Day in the Life of a MERN Stack Developer

As a MERN stack developer, you have been hired by an organization for a development project. Your first task in this project is to process a series of tasks in a specific order. This requirement indicates the need for a data structure that supports sequential processing.

As a developer, it's crucial to understand which data structure is most suitable for this purpose and how to use it effectively.

In this lesson, you will learn concepts that will help you determine the best solution for processing a series of tasks in order. This knowledge will enable you to implement the project's requirements successfully.



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
- 🕒 Operate CRUD (create, read, update, and delete) on a queue for handling data with First-In-First-Out (FIFO) principles
- 🕒 Implement CRUD operations on a singly linked list to establish fundamental data manipulation techniques
- 🕒 Combine the functionalities of stacks and queues using a deque to master versatile data handling techniques

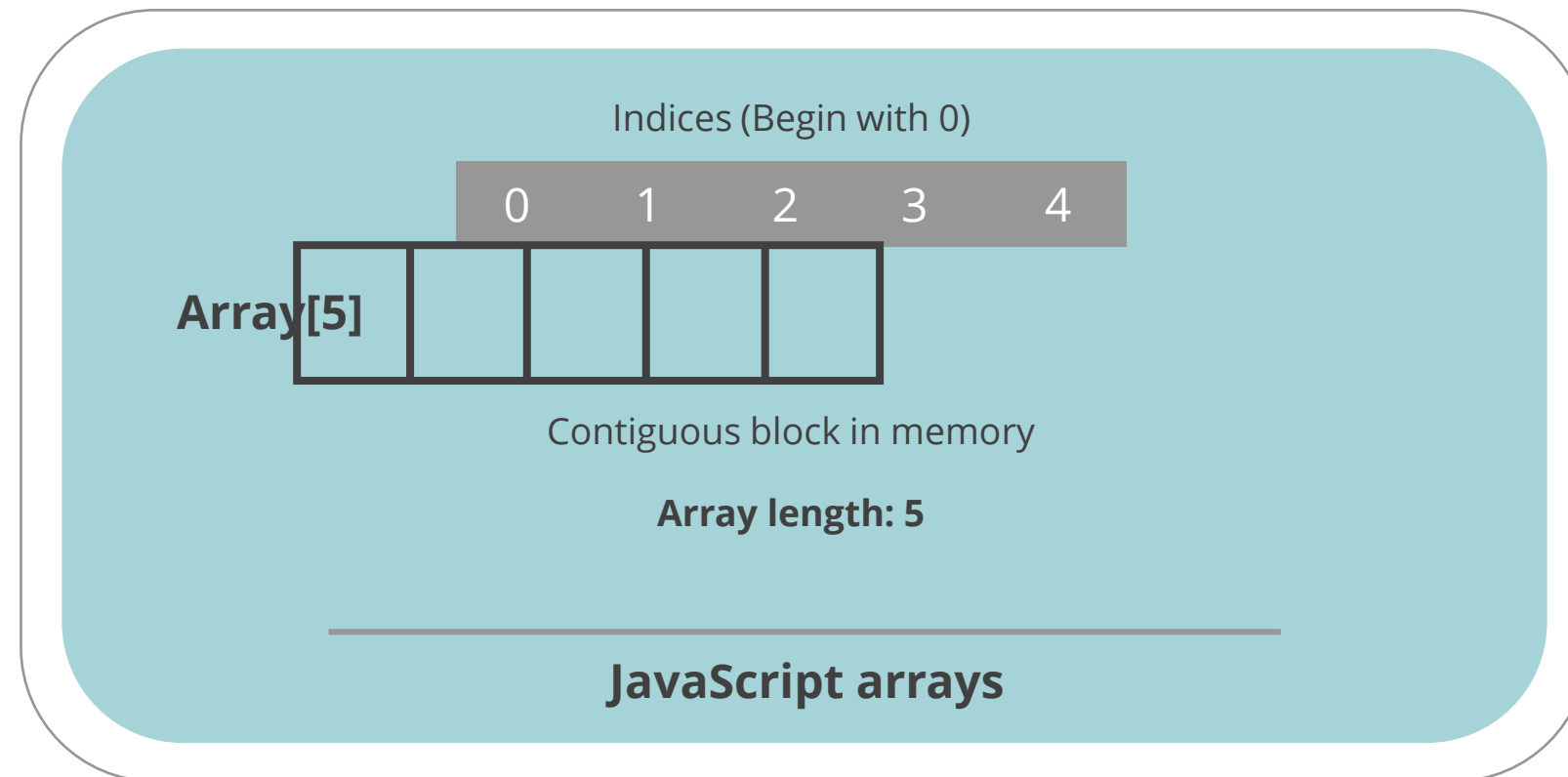




Arrays

Introduction to Arrays

An array is a linear data structure that stores elements of the same data type and stores them in contiguous and adjacent memory locations.



Declaration of Arrays

Arrays can be declared using two methods:

Using array literal

Example:

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

Output:

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

Using new keyword

Example:

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

Output:

```
mango  
banana  
apple
```

Accessing Elements in Arrays

Positive indices retrieve elements from the start. It's important to note that JavaScript does not natively support negative indices. To access elements in reverse order, you can use the length property.

Positive indices

Example:

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

Output:

```
apple  
banana  
mango
```

Negative indices

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
```


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 |

Types of Arrays


Arrays primarily fall into two categories:

One-dimensional



A one-dimensional array is the simplest form, consisting of a single line of elements. Access to each element is through a single index.

Multi-dimensional



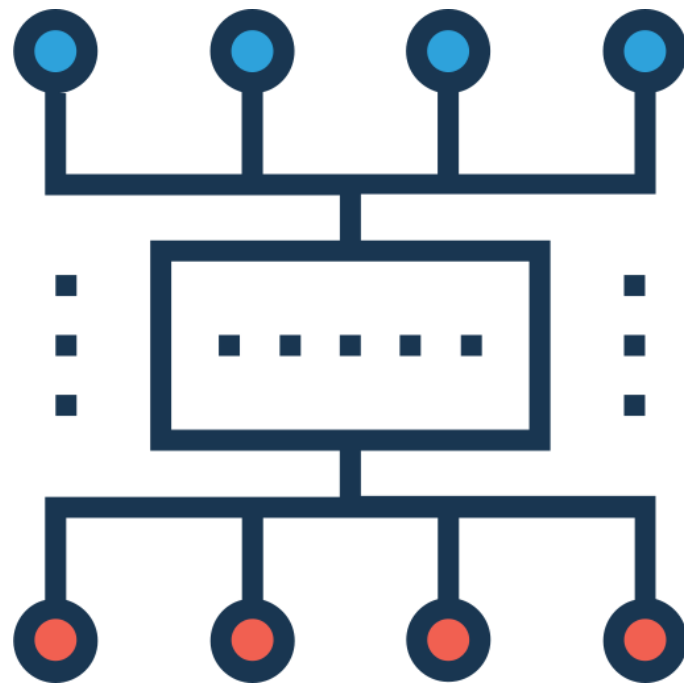
A multi-dimensional array consists of arrays within arrays and may extend beyond two dimensions. This structure allows for the representation of more complex data relationships.



Multi-Dimensional Arrays

What Are Multi-Dimensional Arrays?

A multi-dimensional array has more than one level or dimension.



There are primarily two types of multi-dimensional arrays:

- **Two-dimensional arrays:** These arrays have two dimensions, often visualized as a grid or matrix.
- **Three-dimensional arrays:** These extend to three dimensions, resembling a cube of data.

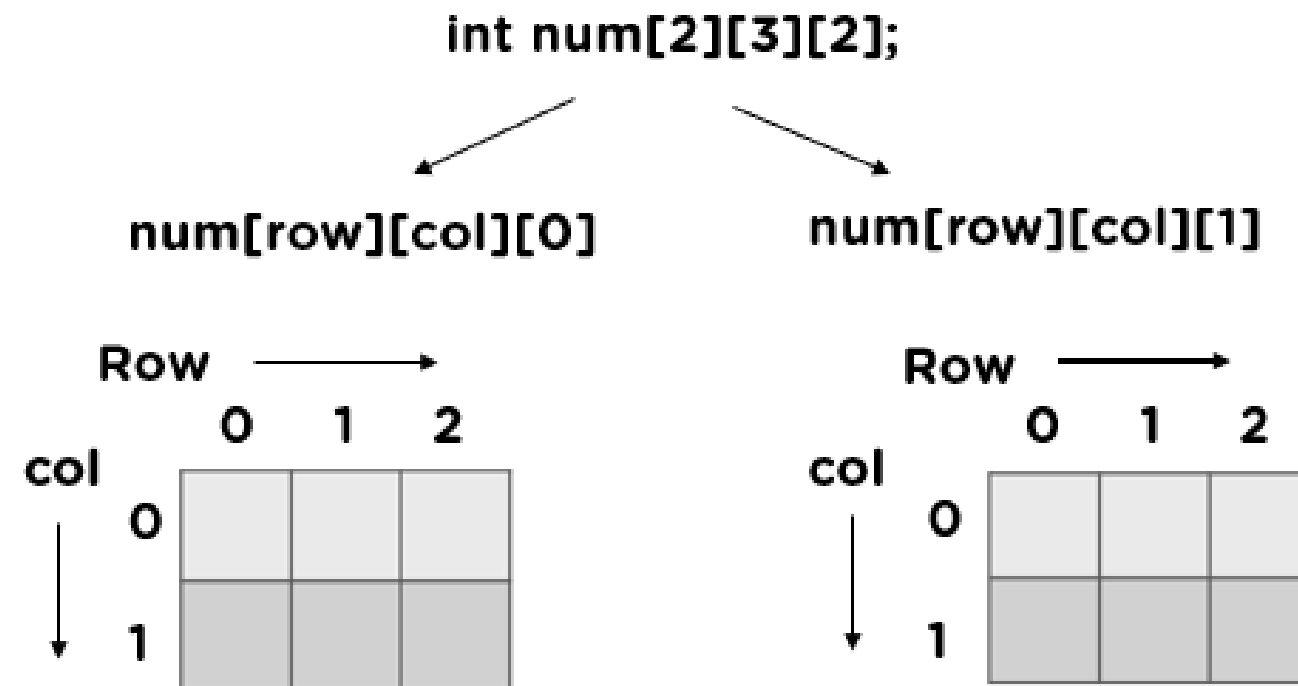
Two-Dimensional Arrays

A two-dimensional array is structured as a matrix, represented by a collection of rows and columns.

| Col → | | 0 | 1 | 2 |
|-----------------|---|---|---|---|
| Row ↓ | 0 | 1 | 2 | 3 |
| | 1 | 4 | 5 | 6 |
| | 2 | 7 | 8 | 9 |

Three-Dimensional Arrays

A three-dimensional array extends a two-dimensional array by adding depth, creating a multi-layered structure of rows, columns, and layers.

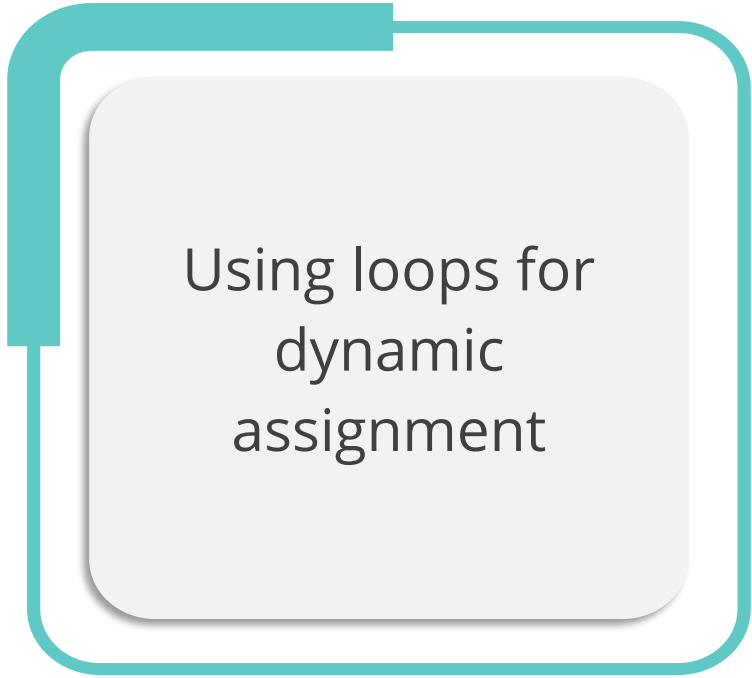


Declaration and Initialization of 2D Arrays

A 2d array can be declared and initialized in the following ways:



Using an initializer
list



Using loops for
dynamic
assignment

Initialization Using Initializer List

Initialize a 2D array using literal notation in JavaScript by nesting arrays within an array.

```
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 Initializer List

To initialize a 2D array using a for loop in JavaScript, users can use nested loops to iterate over the rows and columns.

```
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

Accessing elements in a 2D array in JavaScript involves using two indices: one for the row and another for the column.

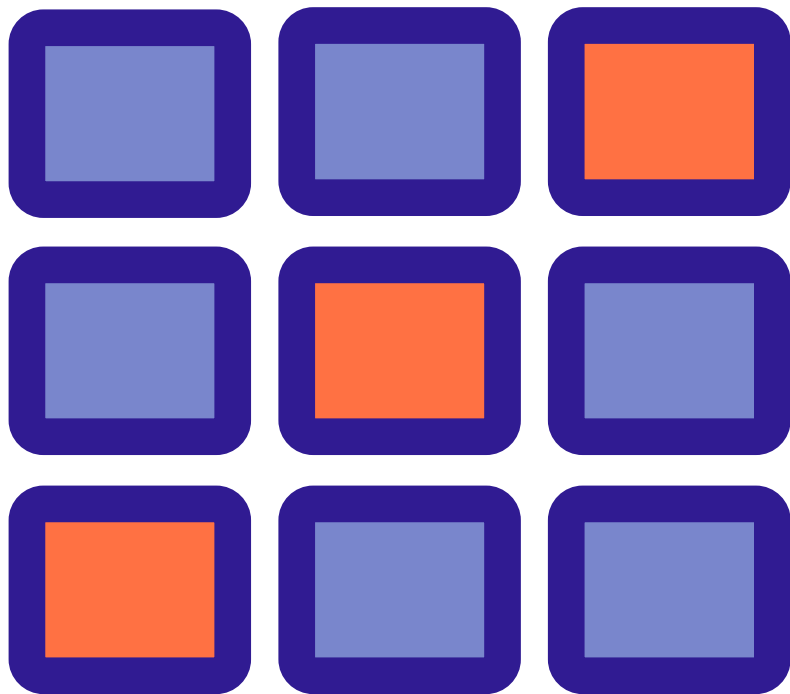
```
let twoDArray = [  
  [1, 2, 3],  
  [4, 5, 6],  
  [7, 8, 9]  
];  
  
// Accessing elements  
console.log(twoDArray[0][0]); // Outputs (first row, first column)  
console.log(twoDArray[1][2]); // Outputs (second row, third column)
```

Output:

1
6

Traversing through 2D Arrays

Traversing through a 2D array in JavaScript involves using nested loops to iterate over rows and columns.



```
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, users can use the following methods:

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');  
console.log(studentsData);
```

Output:

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

Inserting Elements in 2D Arrays

To insert elements into a 2D array, users can use the following methods:

Index notation

Example:

```
let studentsData = [['Jack', 24], ['Sara', 23],];  
studentsData[1][2] = 'hello';  
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  
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]]
```

Matrix Addition

Matrix addition involves adding corresponding elements of two matrices together.
Here's a simple JavaScript function for matrix addition:

Example:

```
function matrixAddition(matrixA, matrixB) {  
  const result = [];  
  
  for (let i = 0; i < matrixA.length; i++) {  
    const row = [];  
    for (let j = 0; j < matrixA[i].length; j++) {  
      row.push(matrixA[i][j] + matrixB[i][j]);  
    }  
    result.push(row);  
  }  
  
  return result;  
}  
  
const resultMatrix = matrixAddition(matrixA, matrixB);  
console.log(resultMatrix);
```

Example:

```
matrixA = [[1, 2], [3, 4]];  
matrixB = [[5, 6], [7, 8]];
```

Output:

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

Matrix Multiplication

Matrix multiplication involves taking the dot product of each row in the first matrix with each column in the second matrix to obtain the elements 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]]
```


Transpose of the Matrix

The transpose of a matrix involves swapping its rows with columns.
Here's a JavaScript function for 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]]
```

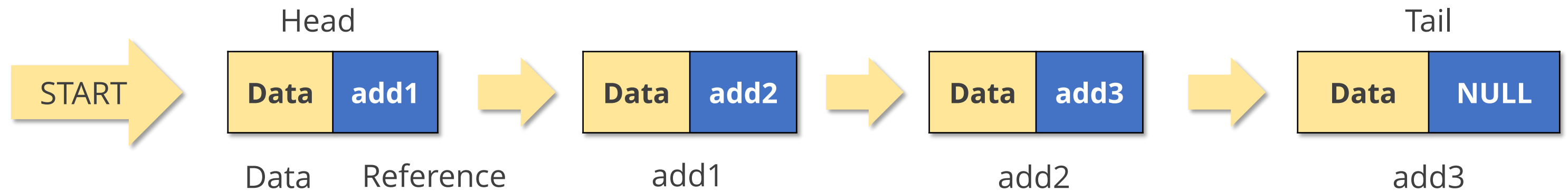
Linked List

What Is the Linked List?

A linked list is a linear data structure whose elements can be traversed using pointers. It consists of nodes, each of which has two parts.

1. Data

2. The reference to next node



Why Do We Need the Linked List?



A linked list is a linear data structure, similar in nature to an array.

- **Dynamic size:**

Unlike an array, which has a fixed size, a linked list can dynamically increase its size to overcome this limitation.

- **Efficient insertions and deletions:**

A linked list performs operations like insertions and deletions with $O(1)$ efficiency, in contrast to an array, which requires shifting all subsequent elements after the operation.

Initialization and Declaration of the Linked List

Here is an example of how to initialize and declare a linked list in JavaScript:

```
// 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. Each node contains a value and a reference to the next node in the list.

Types of Linked Lists

01

Singly linked list

02

Doubly linked list

03

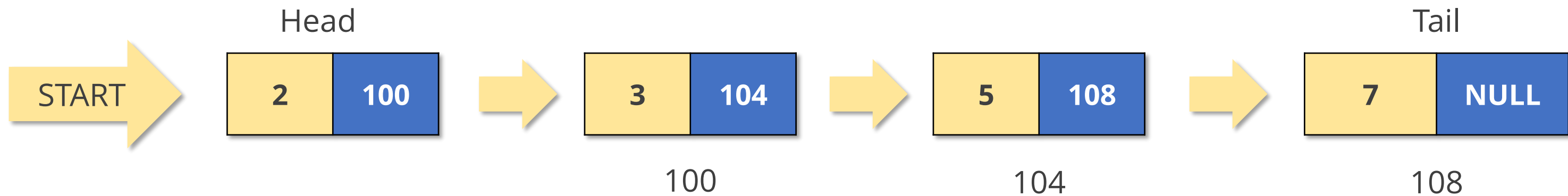
Circular linked list

04

Circular doubly linked list

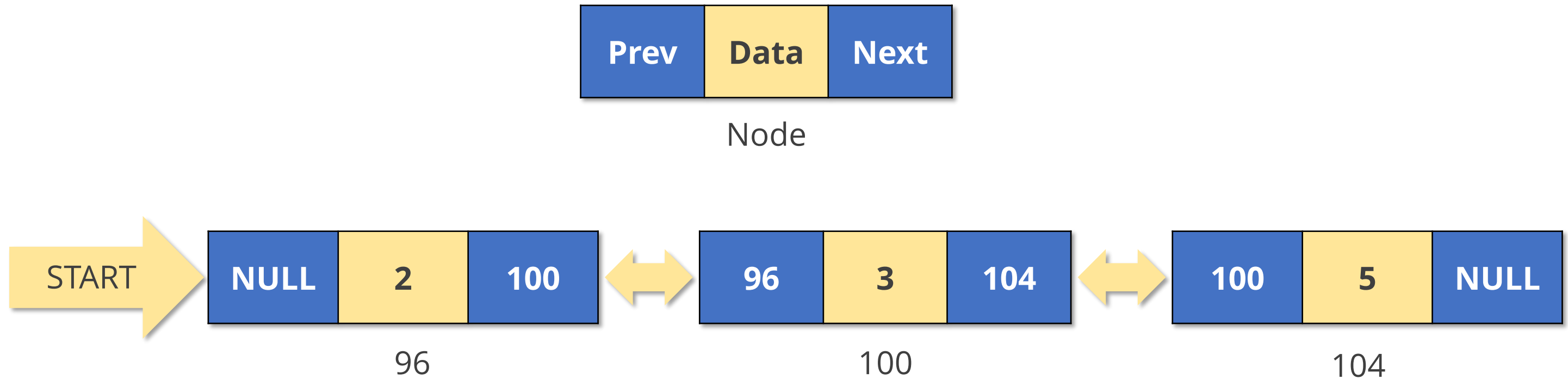
Singly Linked List

A singly linked list is a unidirectional data structure. It allows traversal in only one direction, from the head to the tail.



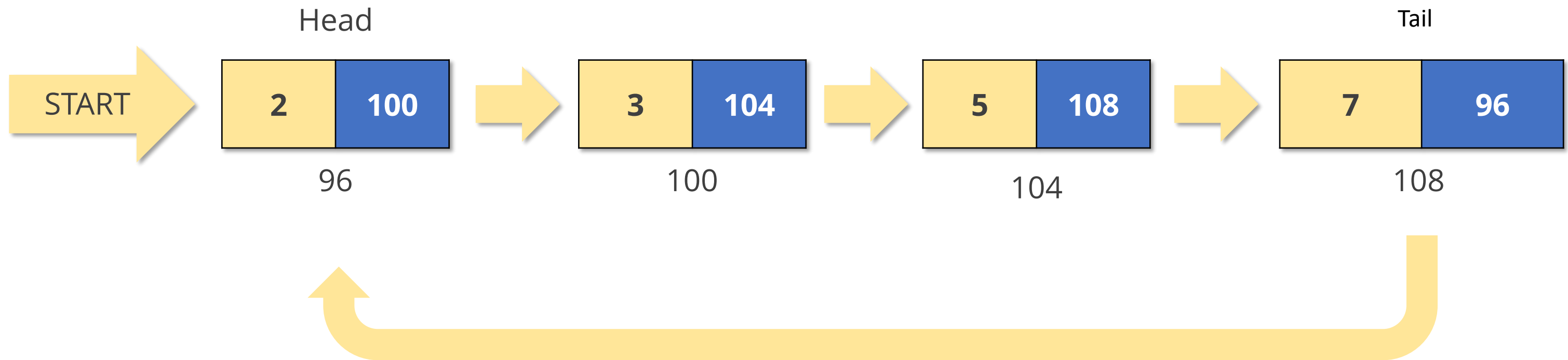
Doubly Linked List

A doubly linked list 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.



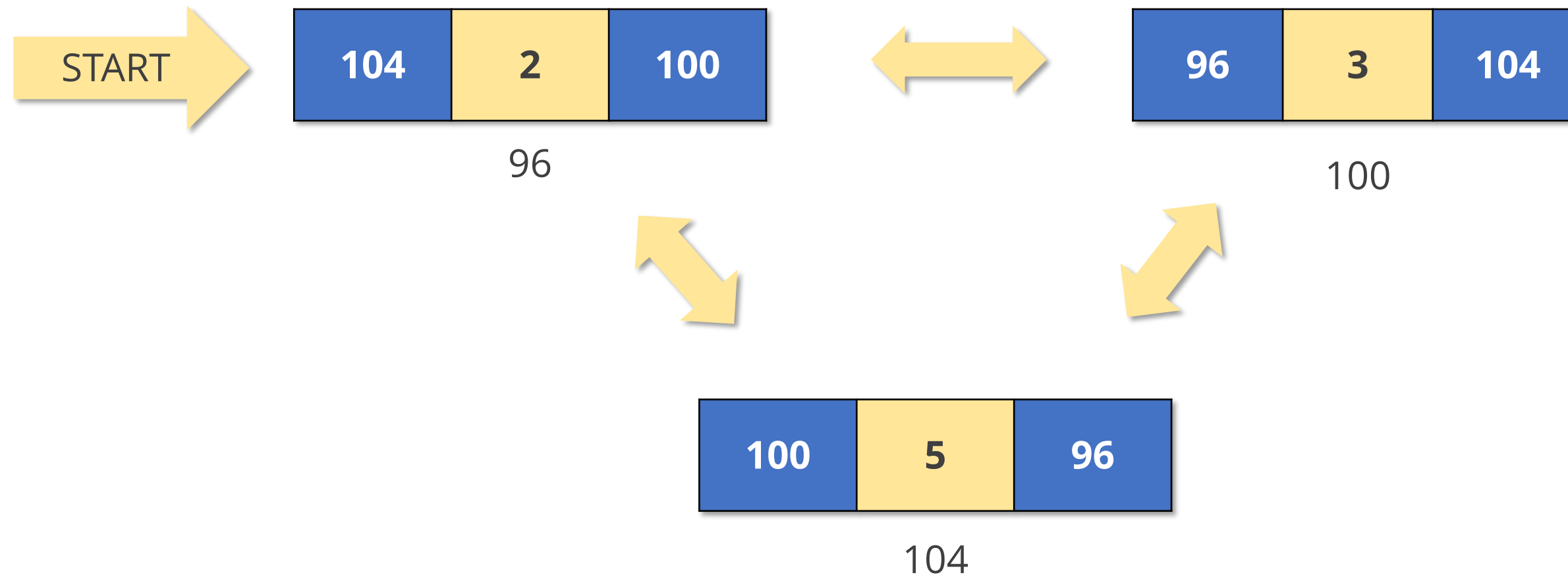
Circular Linked List

A circular linked list is a data structure in which the nodes form a closed loop. The last node points back to the first node, creating a circular structure.



Circular Doubly Linked List

A circular doubly linked list is a data structure in which each node contains two pointers, one to the next node and another pointing to the previous node.



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



Operations on 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

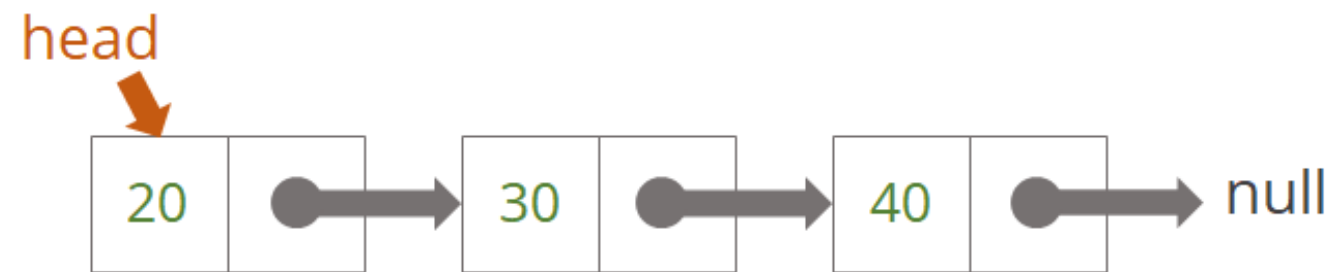
Deletion

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

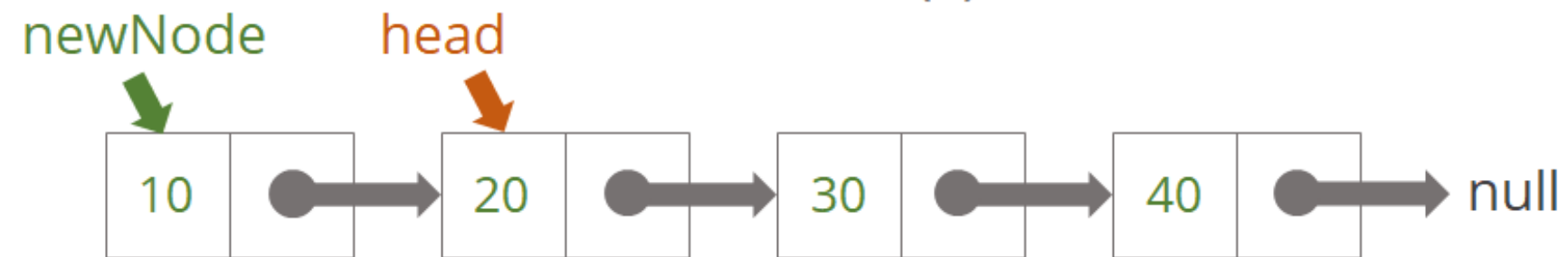


Singly Linked List Insertion: at the Beginning

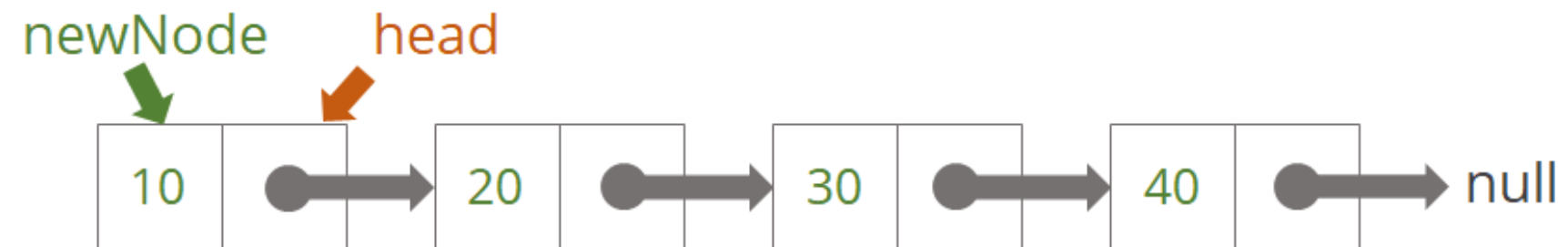
The essential technique for adding a new element to the front of a singly linked list is as follows:



(a)



(b)



(c)

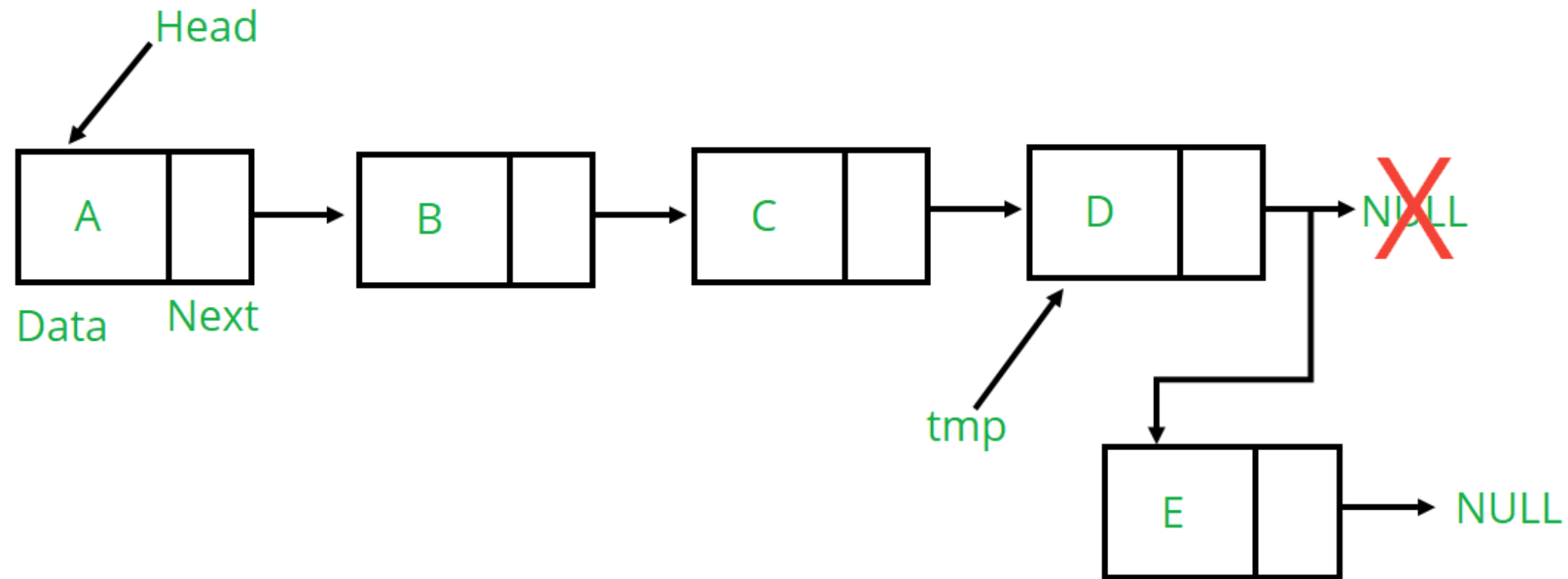
Singly Linked List Insertion: at the Beginning

Below is the JavaScript code for inserting an element at the beginning of a singly linked list:

```
function insertAtHead(linkedList, value) {  
    const newNode = new Node(value);  
    newNode.next = linkedList.head;  
    linkedList.head = newNode;  
}
```

Singly Linked List Insertion: at the End

The fundamental technique for appending a new element to the end of a singly linked list is as follows:



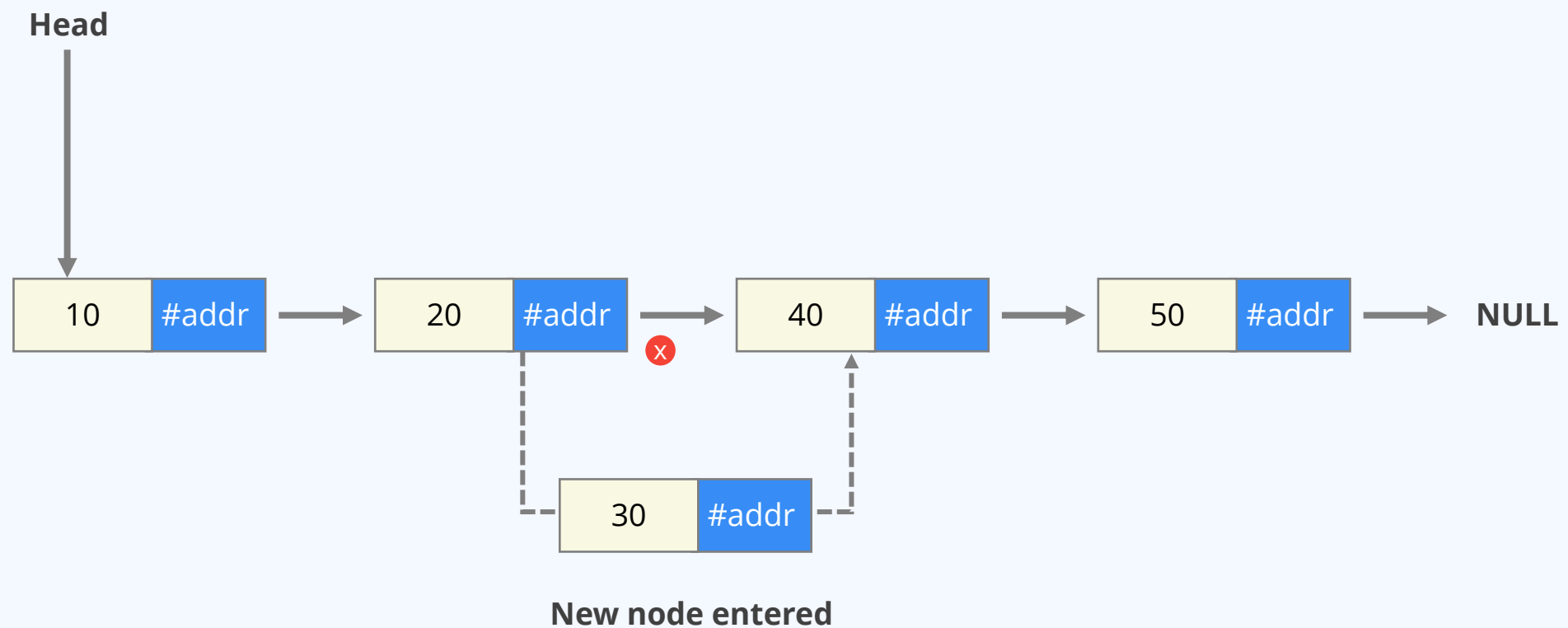
Singly Linked List Insertion: at the End

Below 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);  
    if (!linkedList.head) {  
        linkedList.head = newNode;  
    } else {  
        let current = linkedList.head;  
        while (current.next) {  
            current = current.next;  
        }  
        current.next = newNode;  
    }  
}
```


Singly Linked List Insertion: at a Specific Position

The fundamental technique for inserting a new element at a specific position in a singly linked list is as follows:



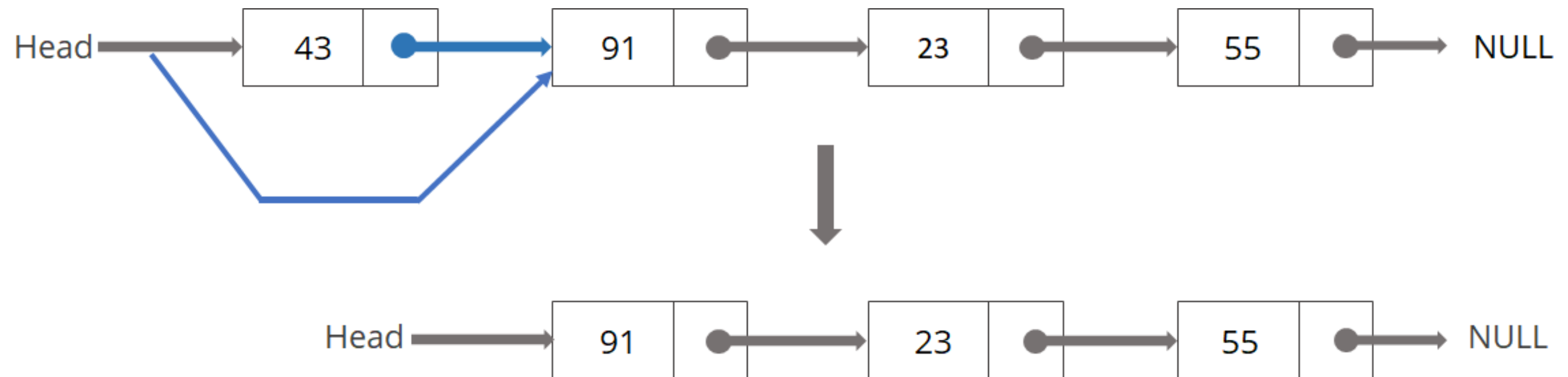
Singly Linked List Insertion: at a Specific Position

Below is the code for inserting an element at the end of a singly linked list in JavaScript.

```
function insertAtPosition(linkedList, value, position) {  
    const newNode = new Node(value);  
    let current = linkedList.head;  
    for (let i = 1; i < position - 1 && current;  
i++) {  
        current = current.next;  
    }  
    if (!current) {  
        console.error("Position out of  
bounds.");  
        return;  
    }  
    newNode.next = current.next;  
    current.next = newNode;  
}
```

Singly Linked List Deletion: at the Beginning

The essential technique for deleting an element from the first node of a singly linked list is as follows:



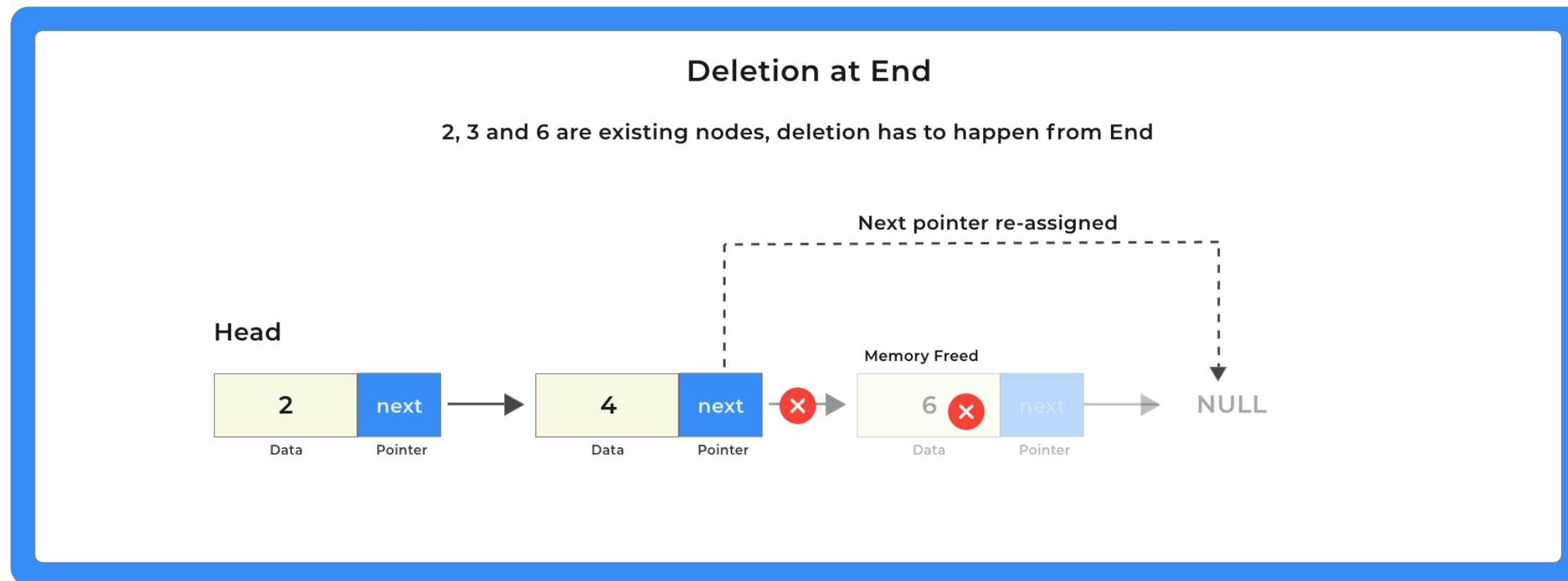
Singly Linked List Deletion: at the Beginning

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

```
function deleteAtHead(linkedList) {  
    if (linkedList.head) {  
        linkedList.head = linkedList.head.next;  
    }  
}
```

Singly Linked List Deletion: at the End

The essential technique for deleting an element from the end of a singly linked list is as follows:



Singly Linked List Deletion: at the End

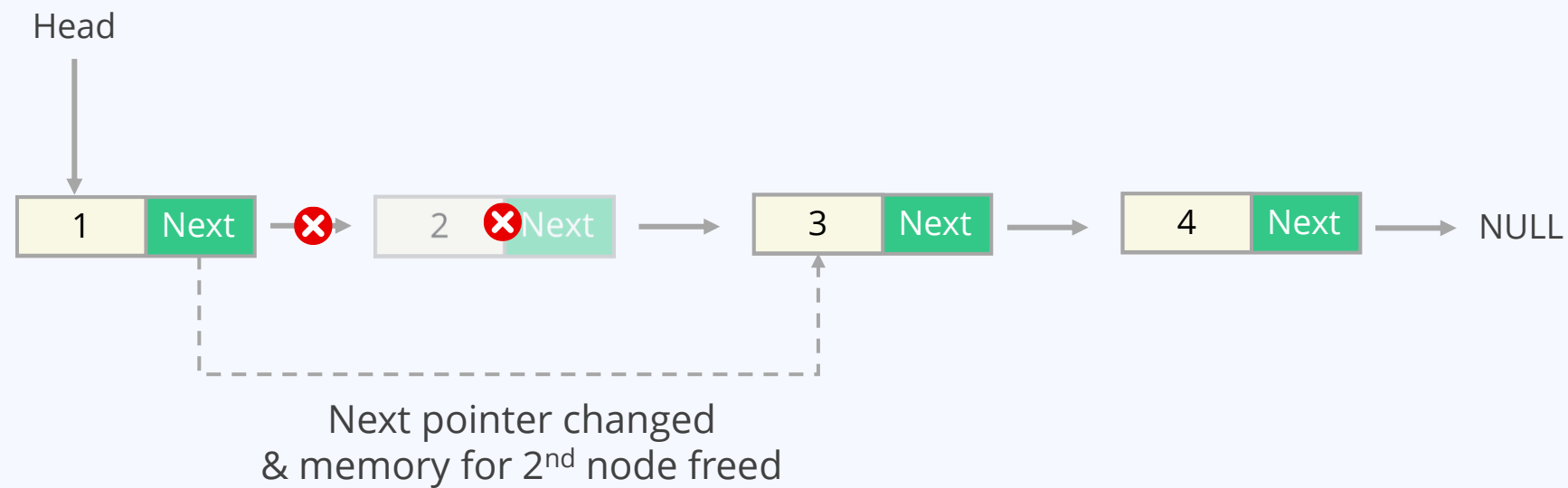
Below is the JavaScript code for deleting an element at the end of a singly linked list:

```
function deleteAtEnd(linkedList) {  
    if (!linkedList.head) {  
        return;  
    }  
    if (!linkedList.head.next) {  
        linkedList.head = null;  
        return;  
    }  
    let current = linkedList.head;  
    while (current.next.next) {  
        current = current.next;  
    }  
    current.next = null;  
}
```

Singly Linked List Deletion: at a Specific Position

The fundamental technique for deleting an element from a specific position in a singly linked list is as follows:

Delete a specific node in a linked list in java



Singly Linked List Deletion: at a Specific Position

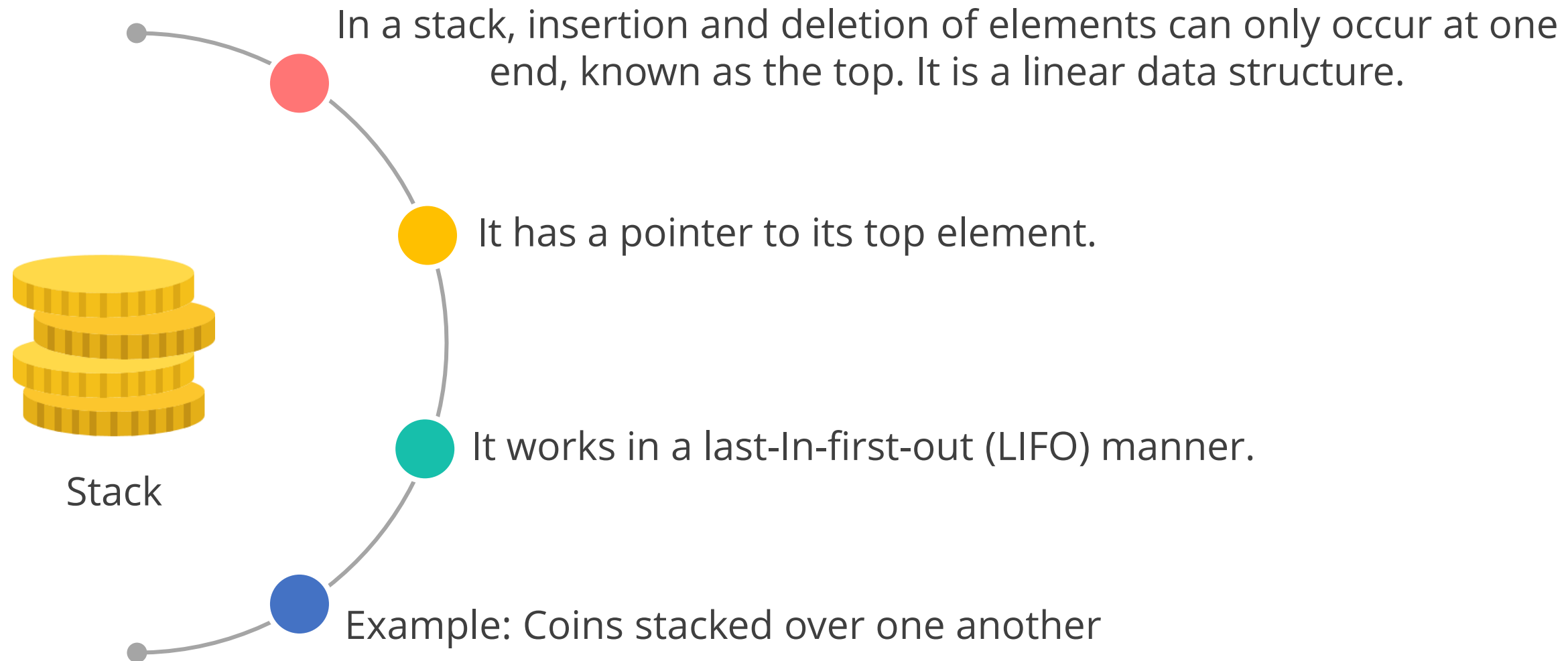
Below is the JavaScript code for deleting an element at the end of a singly linked list:

```
function deleteAtPosition(linkedList, position)
{
    if (!linkedList.head) {
        return;
    }
    if (position === 1) {
        linkedList.head = linkedList.head.next;
        return;
    }
    let current = linkedList.head;
    for (let i = 1; i < position - 1 && current;
i++) {
        current = current.next;
    }
    if (!current || !current.next) {
        console.error("Position out of
bounds.");
        return;
    }
    current.next = current.next.next;}
```



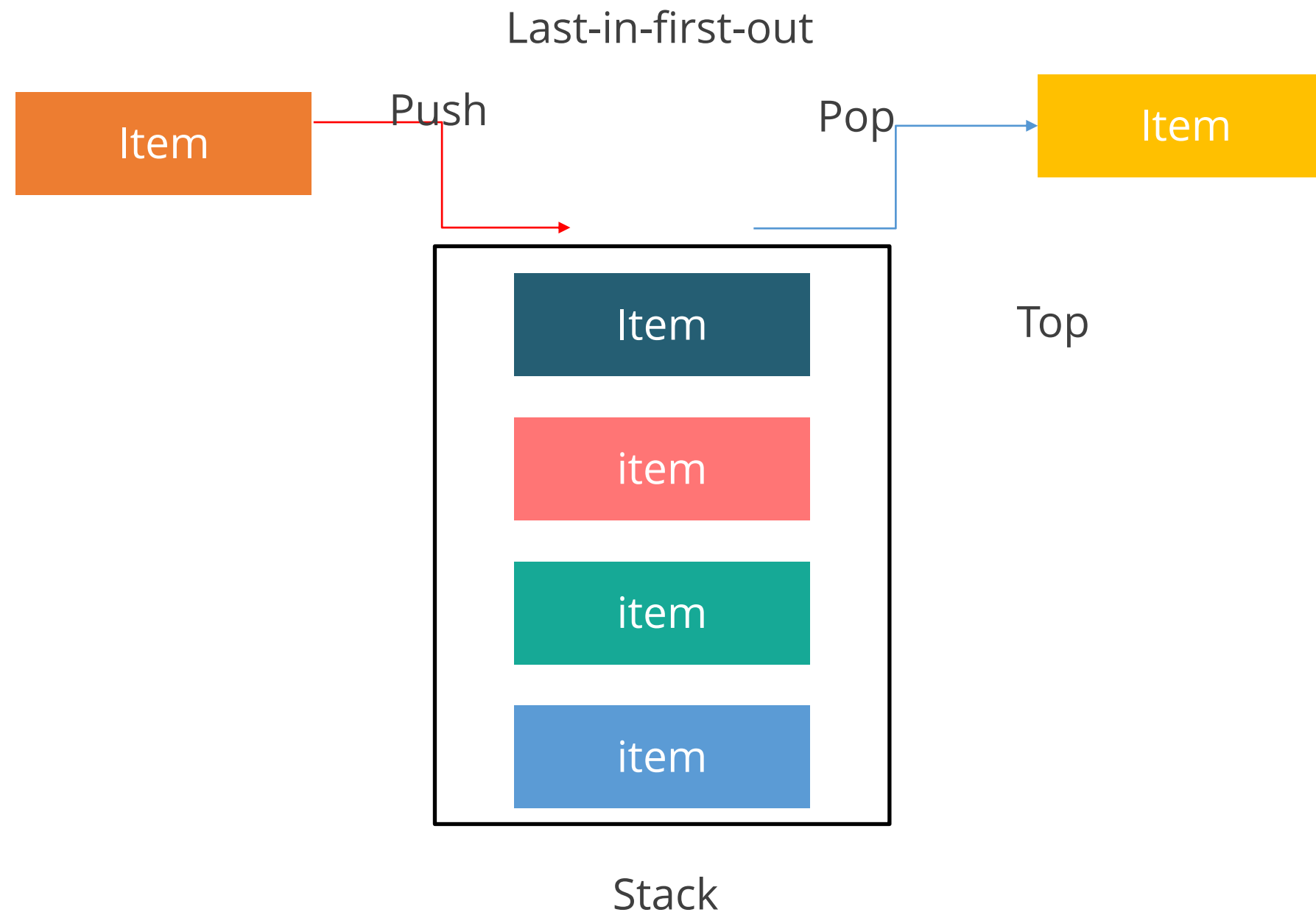

Stacks

What Is a Stack?



Stack Representation

Representation of a stack in data structures:



Stack Operations

These are the most common 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 code below 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
```

Implementation of Stacks Using the Arrays

Below is the code that includes the implementation of a stack using the arrays:

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

Below 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]
```

Implementation of Stacks Using a Linked List

Below is the code that includes the implementation of stacks using a linked list:

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

Below is the code that includes the implementation of stacks using a linked list:

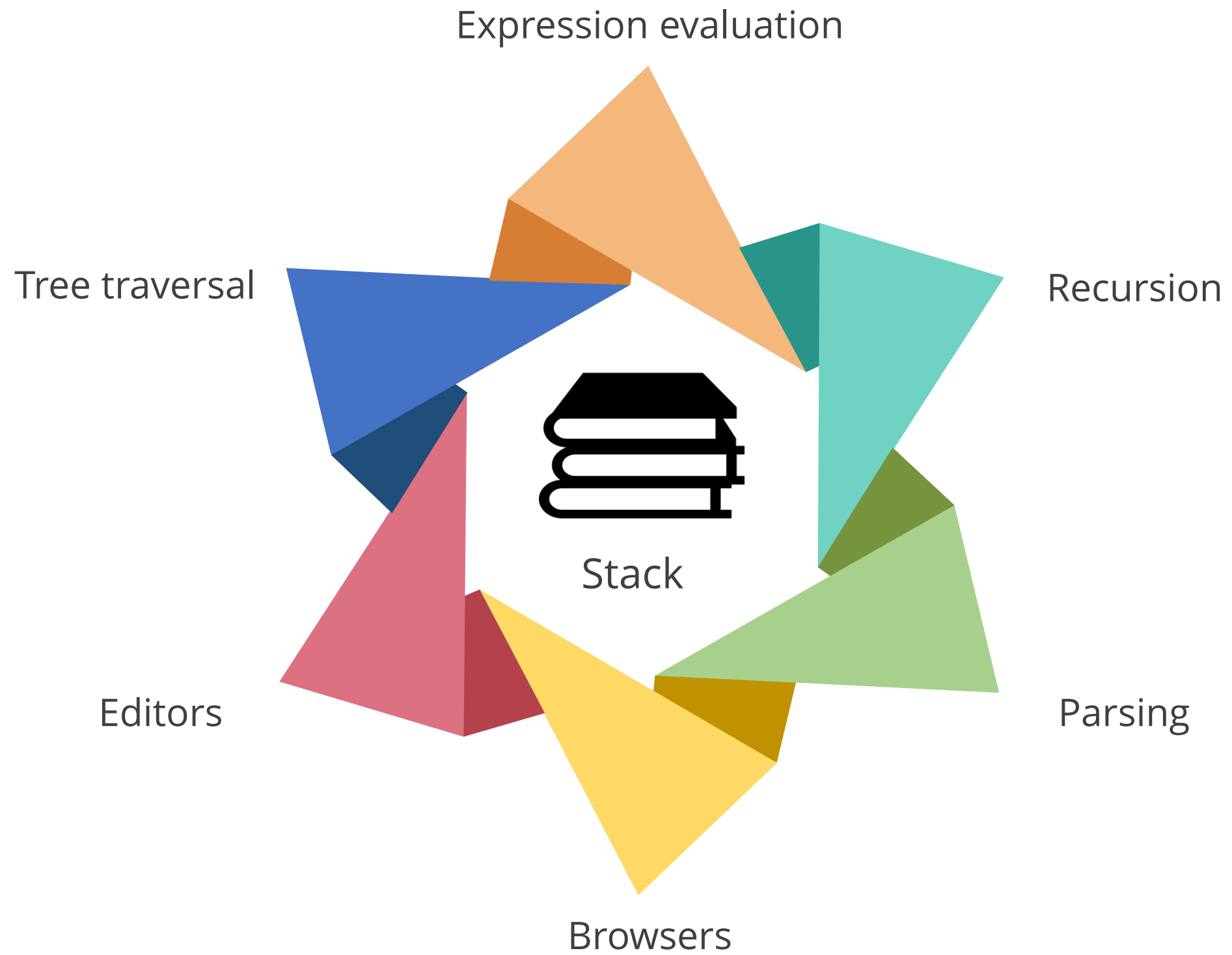
Example:

```
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





Queues

What Is a Queue?

A queue is a linear data structure that enables the user to insert the elements from the *rear* end and delete them from the *front* end.

It works in a first-in-first-out (FIFO) manner.

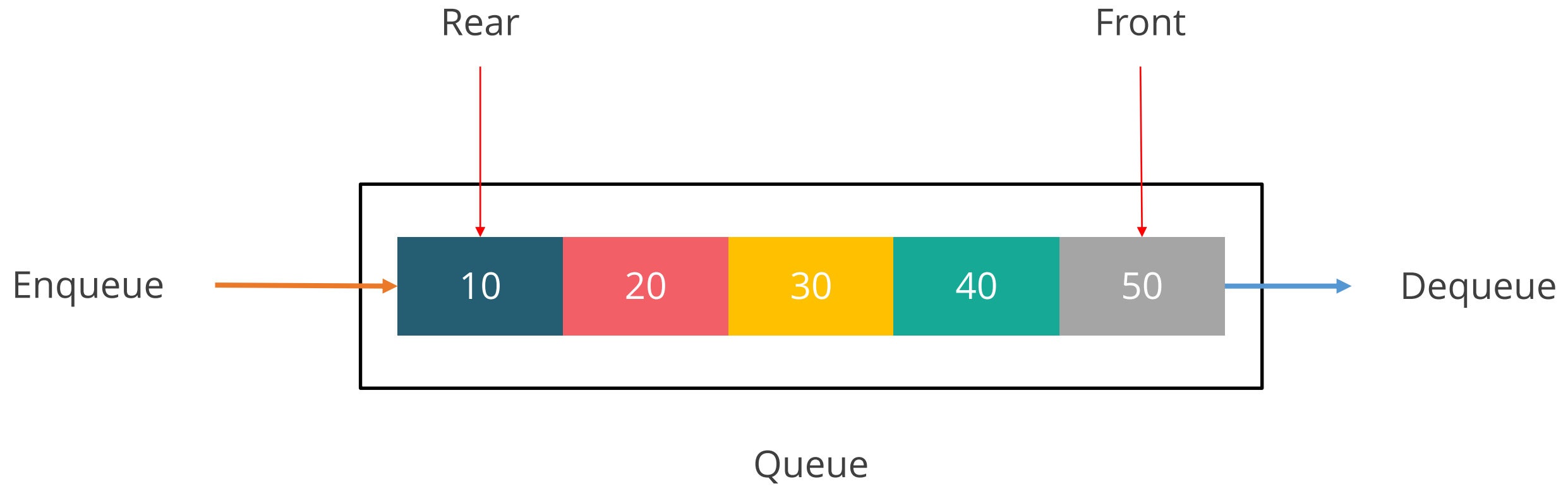


The difference between the stack and the queue is that the elements are removed from opposite ends of the data structure

Example: People waiting in a queue to vote.

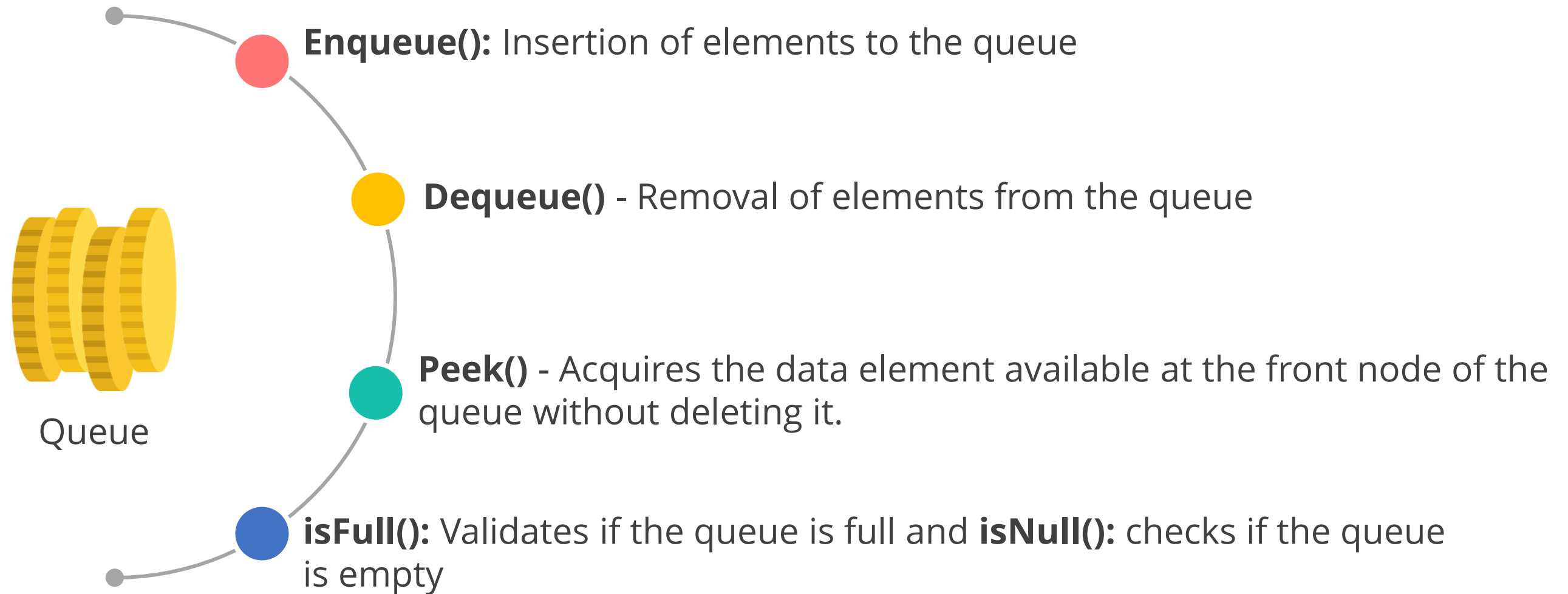
Representation of a Queue

A representation of queue in data structures:



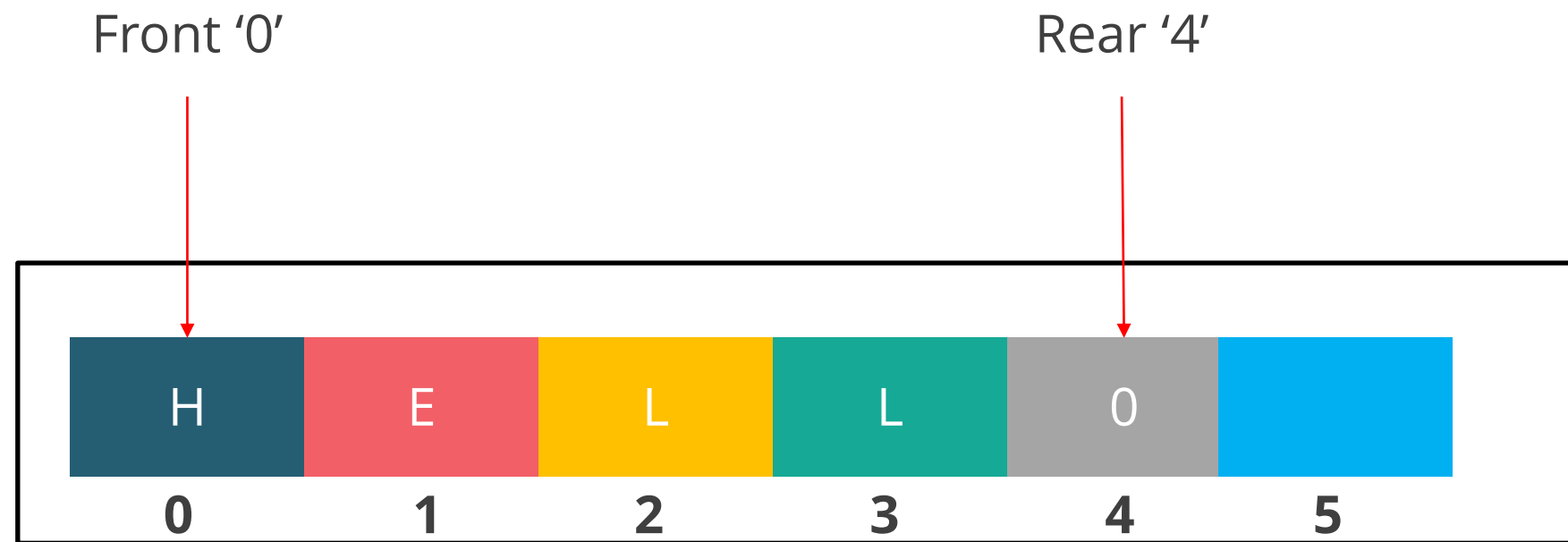
Operations on a Queue

Some operations performed by queue are:



Operations on a Queue

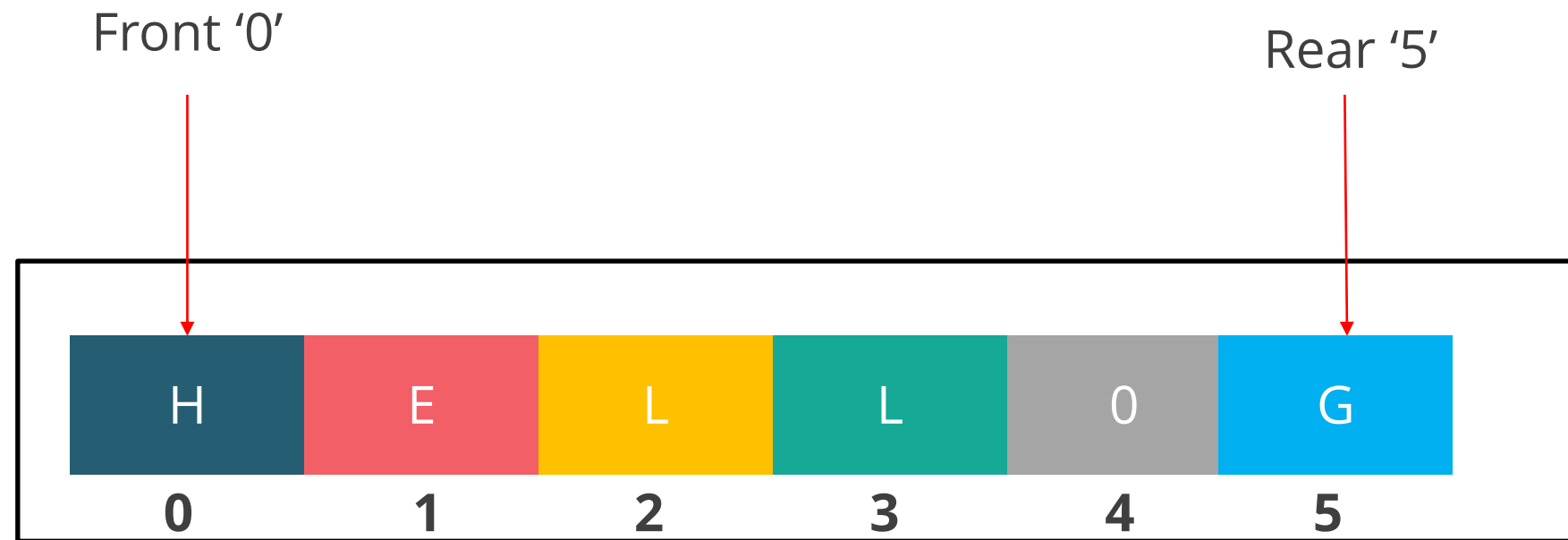
Queue before inserting an element:



Queue

Operations on a Queue

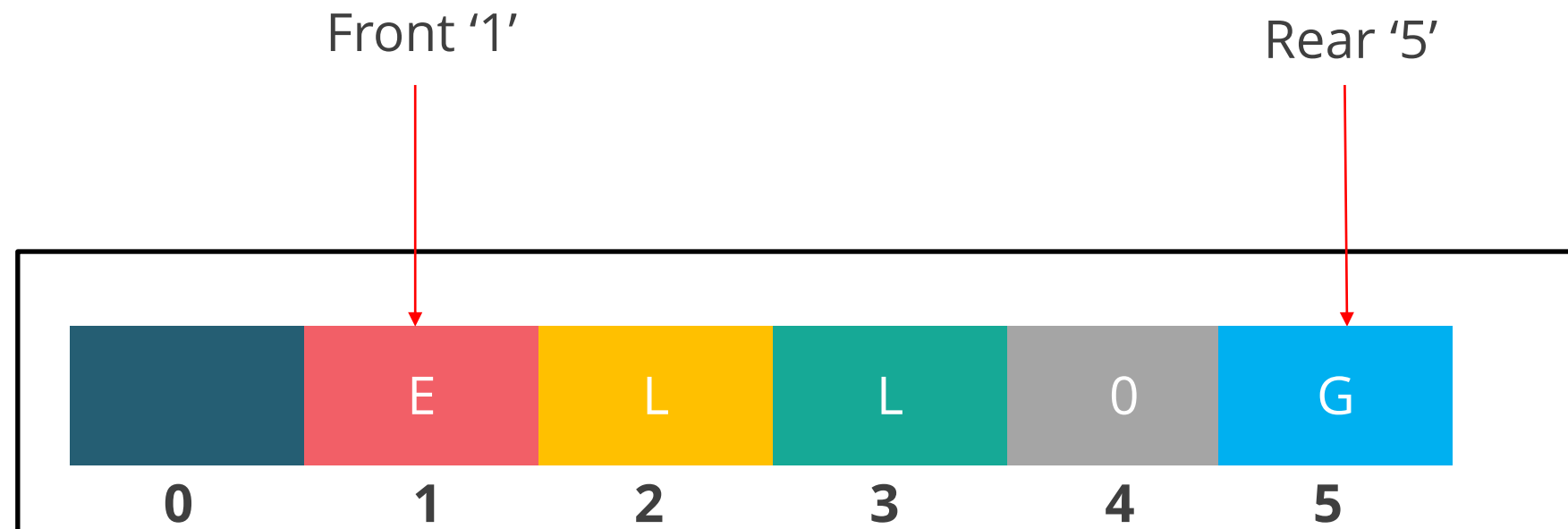
Queue after inserting an element:



Queue

Operations on a Queue

Queue after deleting an element:



Queue

Types of Queues

Simple queue

Circular queue



Priority queue

Double ended queue

Implementation of a Queue Using the Arrays

Below code implements a queue using the arrays:

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

Below code implements a queue using the 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

Below is 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

Below is the code that includes 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

Below is the code that includes the implementation of a queue using a linked list:

Example:

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

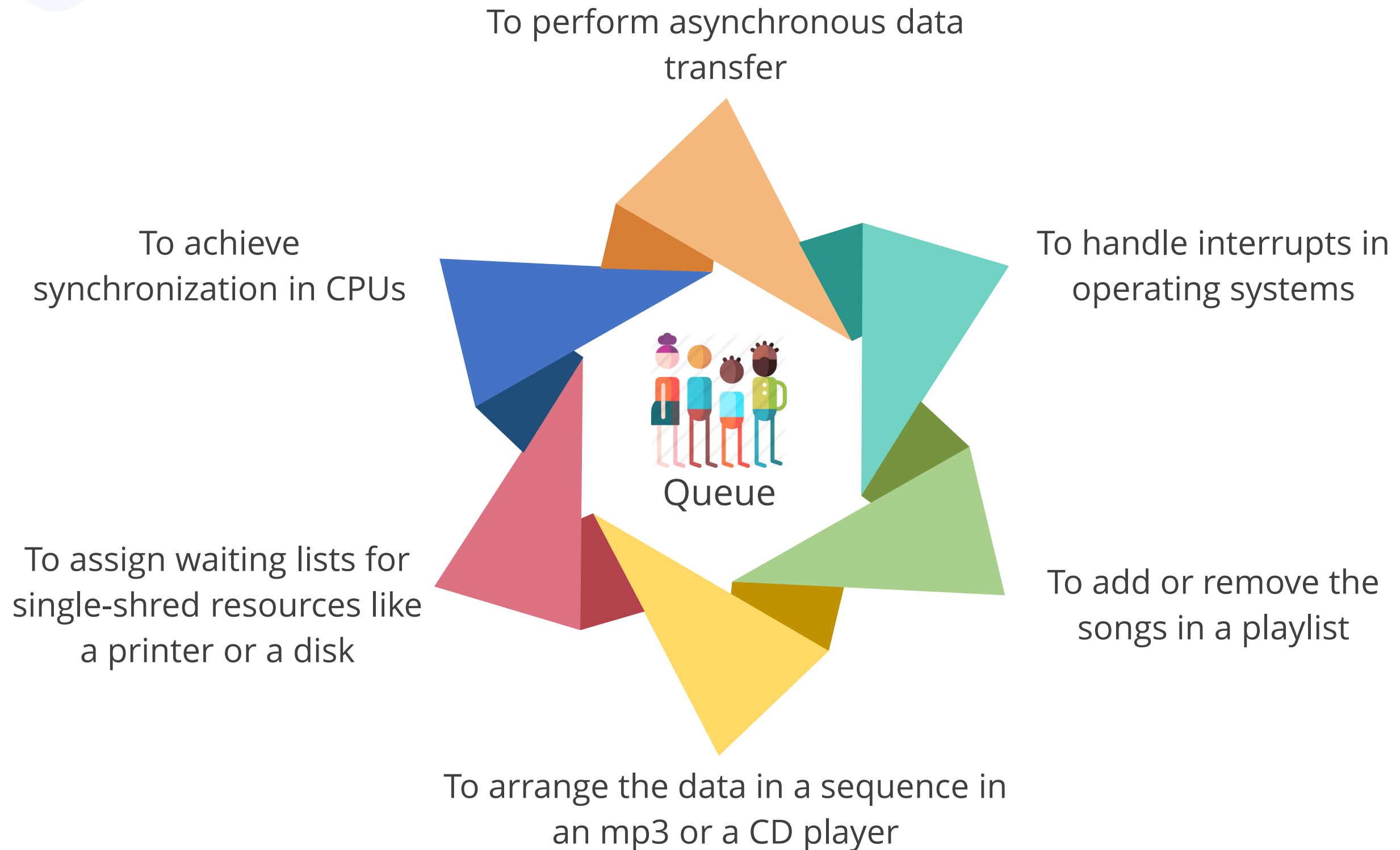
queue.enqueue(10);
queue.enqueue(20);
queue.enqueue(30);

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



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 whose elements can be traversed using pointers. It consists of nodes, each of which has two parts.
- Stack is a linear data structure in which insertion and deletion of elements can be done only at one end called the top.
- 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