Module 01

Purwadhika
Digital Technology School

Full Stack Web Development

Data Structure

Job Connector Program

Outline

- Data Structure
- Stack
- Queue
- Set
- Hash Table / Map
- Linked List

Basic Data Structure

Data structures, at a high level, are techniques for **storing and organizing data** that make it easier to **modify, navigate, and access**. Data structures determine how data is collected, the functions we can use to access it, and the relationships between data.

Data structures are **used in almost all areas of computer science and programming**, from operating systems to artificial intelligence.

Use of Data Structure

Data structures enable us to:

- **Manage and utilize** large datasets
- **Search** for particular data from a database
- **Design algorithms** that are tailored towards particular programs
- **Handle** multiple requests from users at once
- **Simplify and speed up** data processing

Primitive & Non-Primitive Data Structure

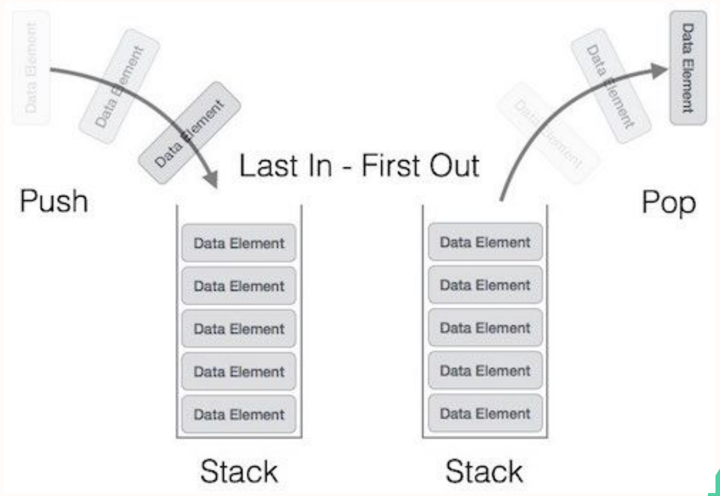
JavaScript has primitive and non-primitive data structures. **Primitive data structures** and data types are native to the programming language. These include boolean, null, number, string, etc.

Non-primitive data structures are not defined by the programming language but rather by the programmer. These include linear data structures, static data structures, and dynamic data structures, like queue and linked lists. These include array and object.

Stack

A **stack** is a data structure that holds a list of elements. A stack allows operations at one end only. This feature makes it **LIFO** data structure. **LIFO** stands for **Last-in-first-out**.

Here, the element which is **placed (inserted or added) last, is accessed first**. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

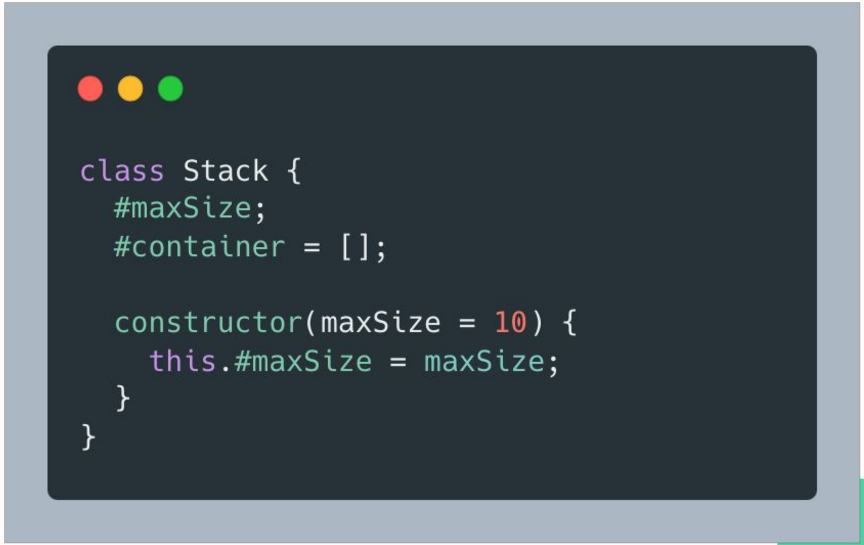


Use of Stack in Real Life

- An undo mechanism in text editors
- Back / forward stacks on browsers
- Reverse a string
- Recursive function
- Etc ...

Stack Implementation

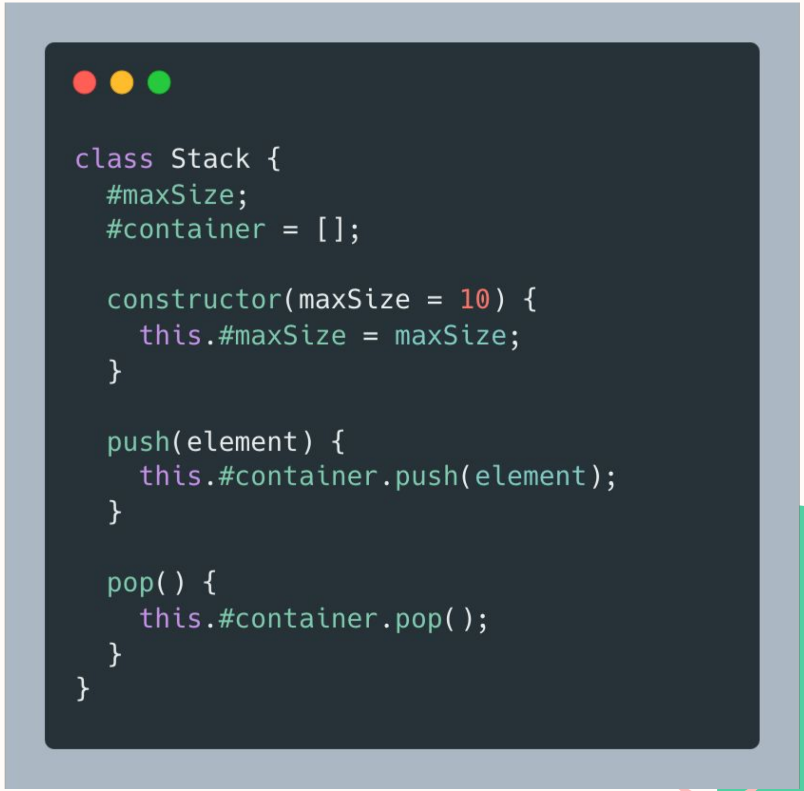
- Create a class named as **Stack**.
- Create private properties named **maxSize** and **container**
- Define default value for **maxSize** and **container**.



```
class Stack {
  #maxSize;
  #container = [];

  constructor(maxSize = 10) {
    this.#maxSize = maxSize;
  }
}
```

Stack Implementation

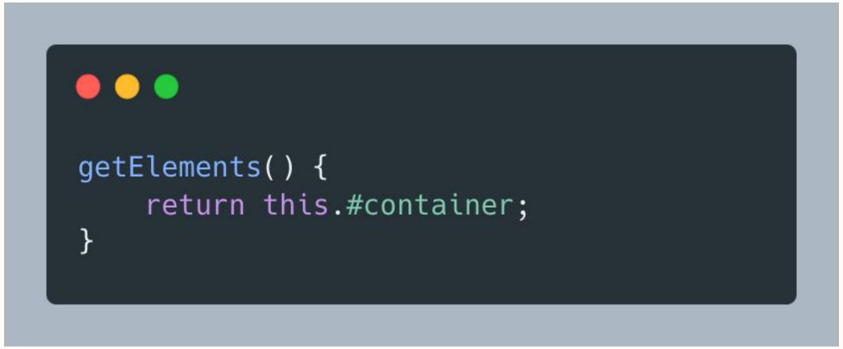
- Create methods **push** and **pop** to modify container.
- **Push** will insert the data into last index of container.
- **Pop** will remove the last value inside the container.



```
class Stack {
  #maxSize;
  #container = [];

  constructor(maxSize = 10) {
    this.#maxSize = maxSize;
  }

  push(element) {
    this.#container.push(element);
  }

  pop() {
    this.#container.pop();
  }
}
```

Stack Implementation

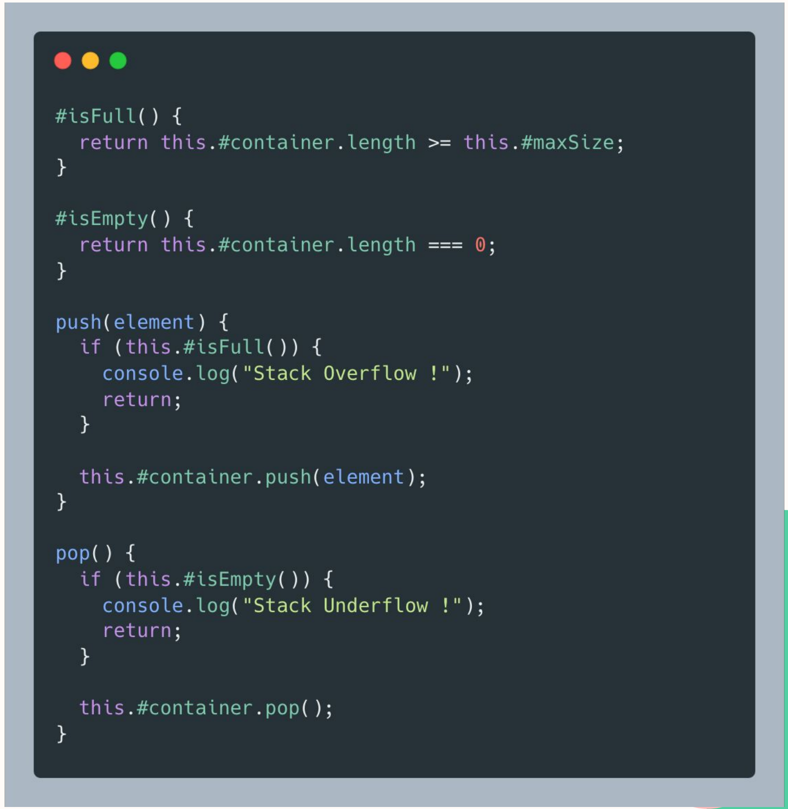
- Create method **getElements** to get all element in container.



```
getElements() {
    return this.#container;
}
```

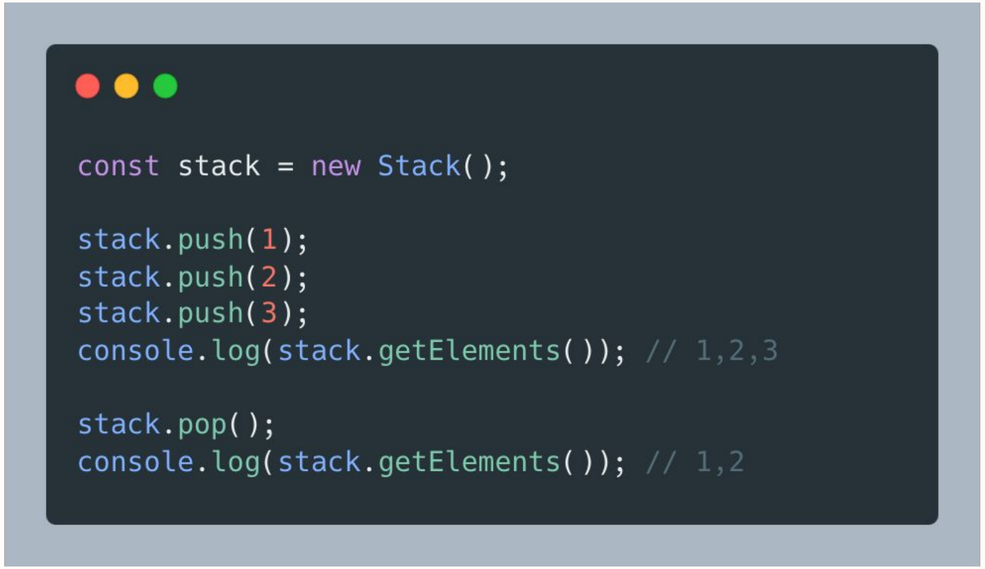
Stack Implementation

- Validate push and pop methods so that it does not exceed **maxSize**.
- **isFull** will check is the container length is same as the **maxSize**.
- **isEmpty** will check is there any value inside the container.



```
#isFull() {
  return this.#container.length >= this.#maxSize;
}

#isEmpty() {
  return this.#container.length === 0;
}

push(element) {
  if (this.#isFull()) {
    console.log("Stack Overflow !");
    return;
  }

  this.#container.push(element);
}

pop() {
  if (this.#isEmpty()) {
    console.log("Stack Underflow !");
    return;
  }

  this.#container.pop();
}
```

Stack Implementation



```
const stack = new Stack();

stack.push(1);
stack.push(2);
stack.push(3);
console.log(stack.getElements()); // 1,2,3

stack.pop();
console.log(stack.getElements()); // 1,2
```

Queue

The **queue** is an abstract data structure, somewhat similar to Stacks. Unlike stacks, **a queue is open at both its ends**. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).

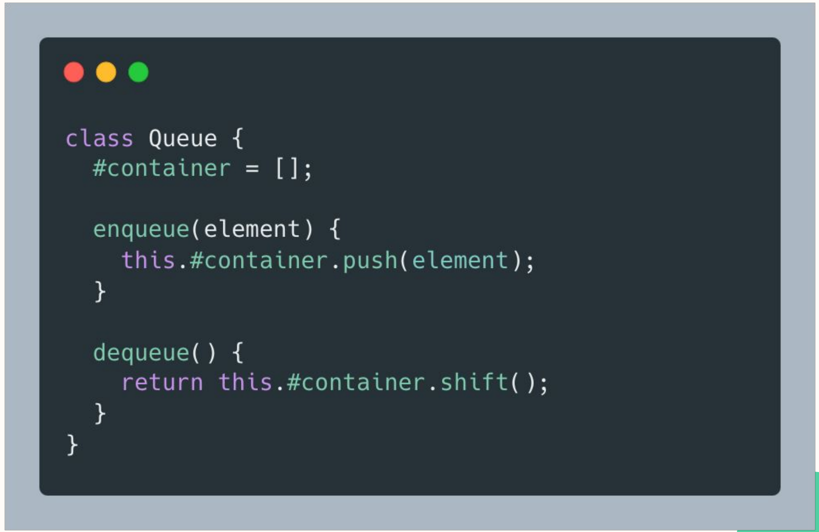
Queue follows **First-In-First-Out** methodology, i.e., the data item stored first will be accessed first.

Use of Queue in Real Life

- Serving requests on a single shared resource, like a printer & CPU task scheduling
- Applied on WhatsApp when we send messages to our friends and they don't have an internet connection then these messages are queued on the server of WhatsApp
- Queues in routers/ switches
- Email queues
- Etc ...

Queue Implementation

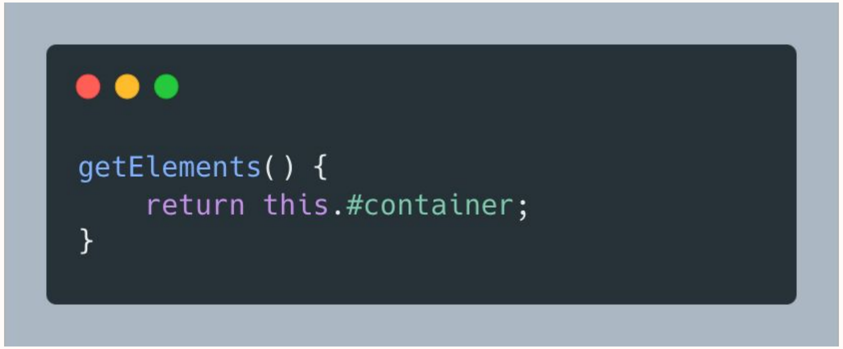
- Create a class named as **Queue**.
- Create private properties named **container** to store all element
- Create methods **enqueue** and **dequeue** to modify container.
- **Enqueue** will insert the data into last index of container.
- **Dequeue** will remove the first value inside the container. We will use **Array.shift()** to removes the first element from container and return it.



```
class Queue {
  #container = [];

  enqueue(element) {
    this.#container.push(element);
  }

  dequeue() {
    return this.#container.shift();
  }
}
```

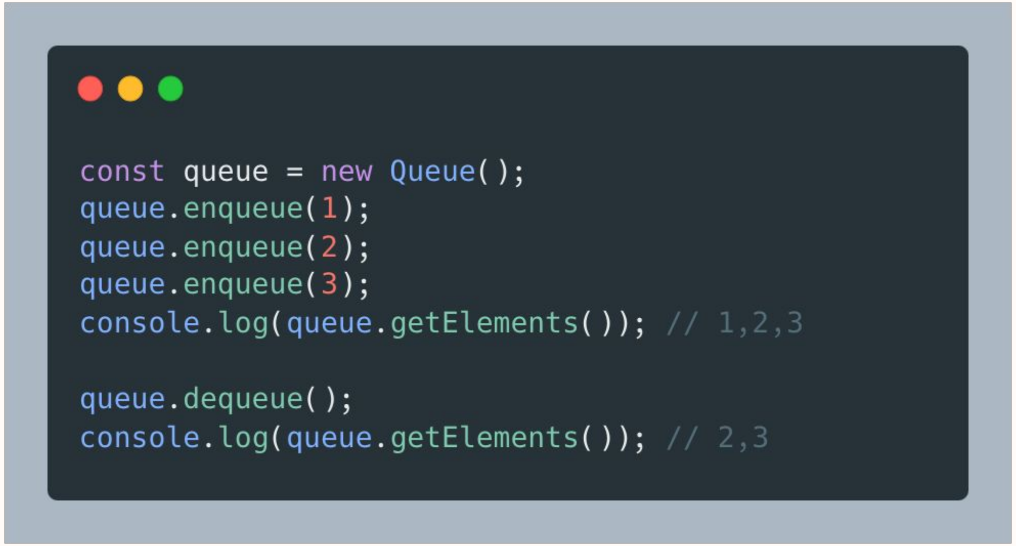
Queue Implementation

- Create method **getElements** to get all element in container.



```
getElements() {
    return this.#container;
}
```

Queue Implementation



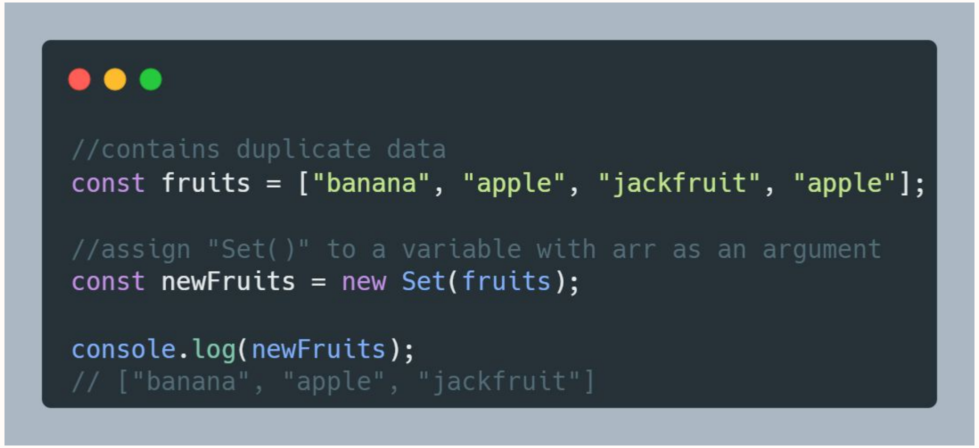
```
const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
console.log(queue.getElements()); // 1,2,3

queue.dequeue();
console.log(queue.getElements()); // 2,3
```

Set

The main Feature of **Set** is it **doesn’t allow duplicate values (unique values)**. ES6 provides a new data structure named **Set** that stores a collection of unique values of any type, whether primitive values (like strings or integers) or object references (like arrays or function).

Set Example

As you can see in example below, there is duplicate data in variable arr. Then assign new Set(arr) to variable set, and you can see when we console.log set, that data become Set data structure and there is no duplicate data.



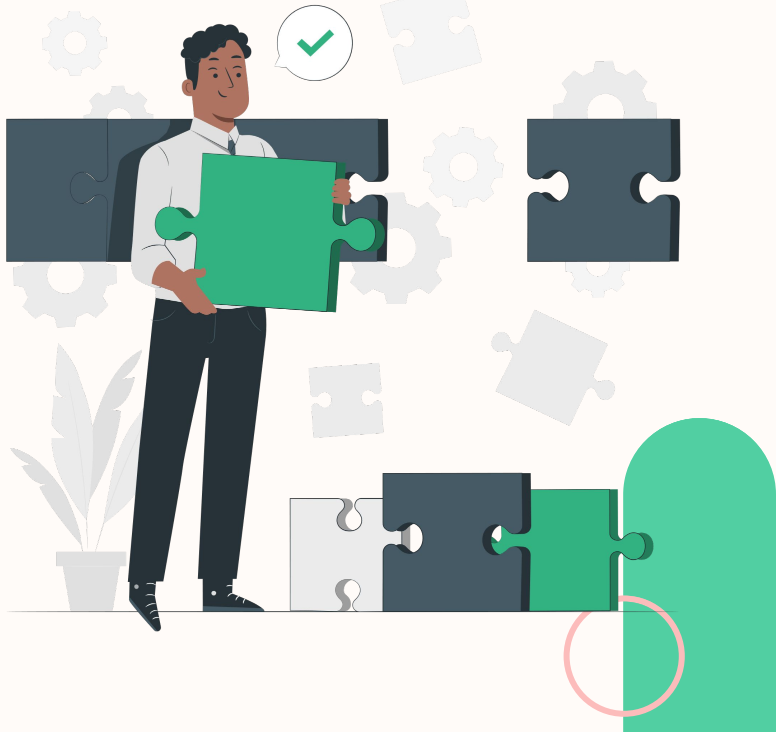
```
//contains duplicate data
const fruits = ["banana", "apple", "jackfruit", "apple"];

//assign "Set()" to a variable with arr as an argument
const newFruits = new Set(fruits);

console.log(newFruits);
// ["banana", "apple", "jackfruit"]
```

Set Built-in Methods

- add
- delete
- has
- clear
- size
- entries
- forEach

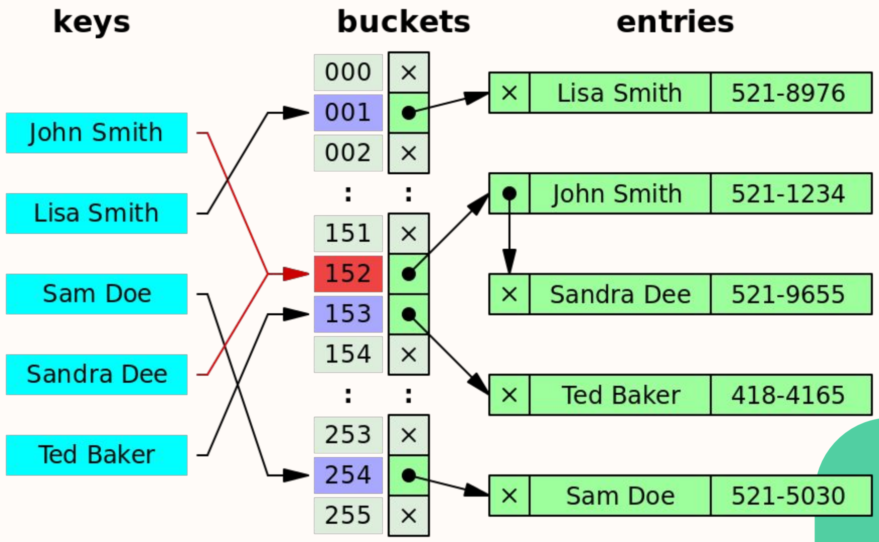
Hash Table / Map

A **hash table** (often called a *hash map*) is a data structure that maps keys to values. Hash tables combine **lookup**, **insert**, and **delete** operations in an efficient way. The key is sent to a hash function that performs arithmetic operations on it. The result (called the hash value or hash) is an index of the key-value pair.



Hash Table / Map

Think of this like a signature on a **block of data that allows us to search in constant time**. A hash table operates like a dictionary that we can ***map*** to get from the hash to the desired data.



Use of Hash Tables

Hash tables **provide access to elements in constant time**, so they are highly recommended for **algorithms that prioritize search and data retrieval operations**. Hashing is ideal for large amounts of data, as they take a constant amount of time to perform insertion, deletion, and search.

Hash Table Collisions

Sometimes, a hash function can **generate the same index for more than one key**. This scenario is referred to as a **hash collision**. Collisions are a problem because every slot in a hash table is supposed to store a single element.

Hash collisions are usually handled using four common strategies.

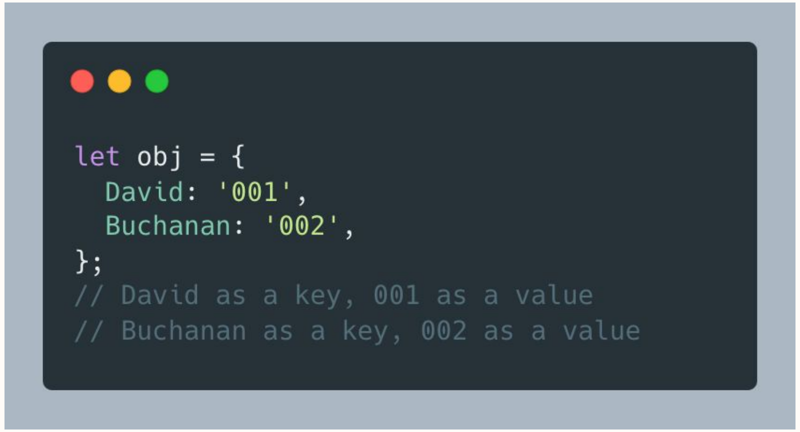
- Linear Probing
- Chaining
- Resizing of the Array or List
- Double Hashing

Hash Table Implementation - Object

The most common example of a Hash Table in JavaScript is the **Object** data type, where you can pair the object's property value with a property key.

But JavaScript's Object type is a special kind of Hash Table implementation for two reasons:

- It has properties added by the Object class. **Keys you input may conflict and overwrite default properties inherited from the class.**
- **The size of the Hash Table is not tracked.** You need to manually count how many properties are defined by the programmer instead of inherited from the prototype.

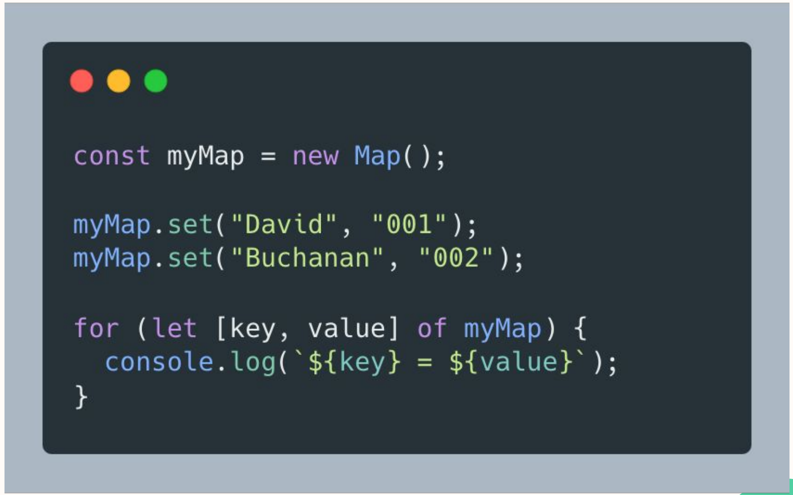


```
let obj = {
  David: '001',
  Buchanan: '002',
};
// David as a key, 001 as a value
// Buchanan as a key, 002 as a value
```

Hash Table Implementation - Map

Map allows you to store key-value pairs inside the data structure.

Unlike the Object type, Map requires you to use the **set()** and **get()** methods to define and retrieve any key-pair values that you want to be added to the data structure.



```
const myMap = new Map();

myMap.set("David", "001");
myMap.set("Buchanan", "002");

for (let [key, value] of myMap) {
  console.log(`${key} = ${value}`);
}
```

Linked List

A **linked list** is a linear data structure, in which the elements are not stored at contiguous memory locations.

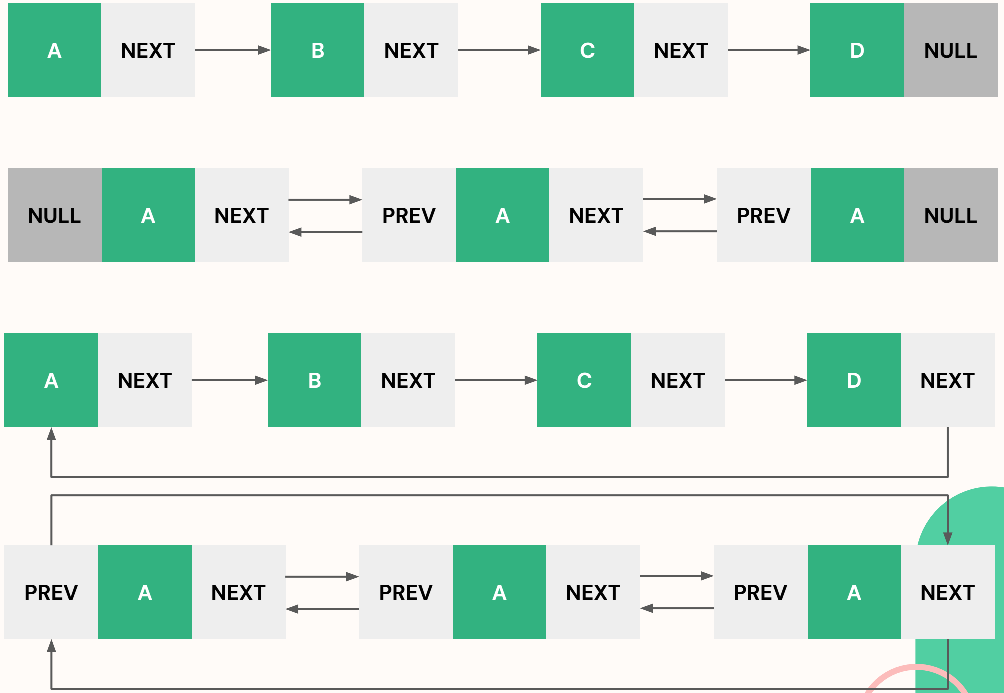
In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Use of Linked List in Real Life

- Image viewer – Previous and next images are linked, hence can be accessed by next and previous button.
- Previous and next page in web browser – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- Music Player – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.
- Etc …

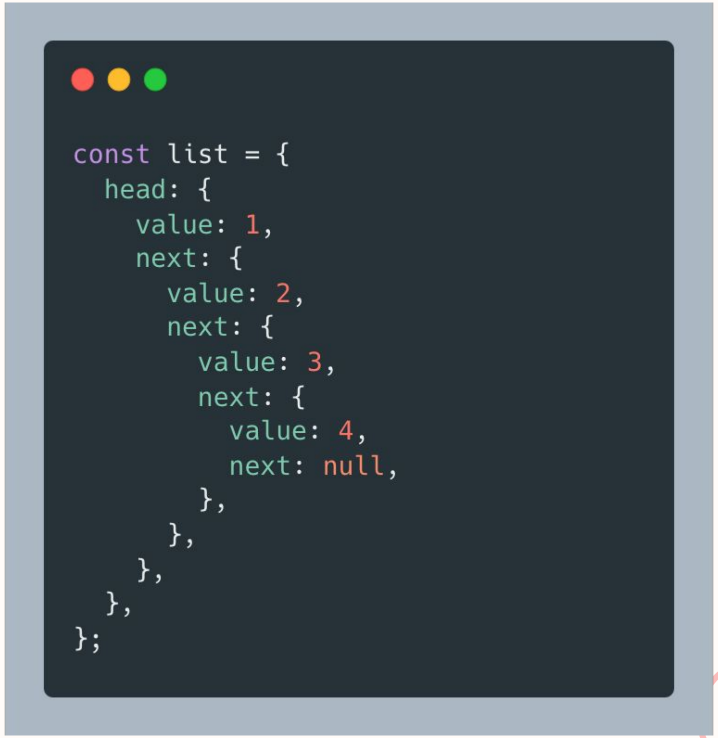
Type of Linked List

a. Single linked lists
b. Double linked lists
c. Circular linked lists
d. Circular double linked lists



Linked List

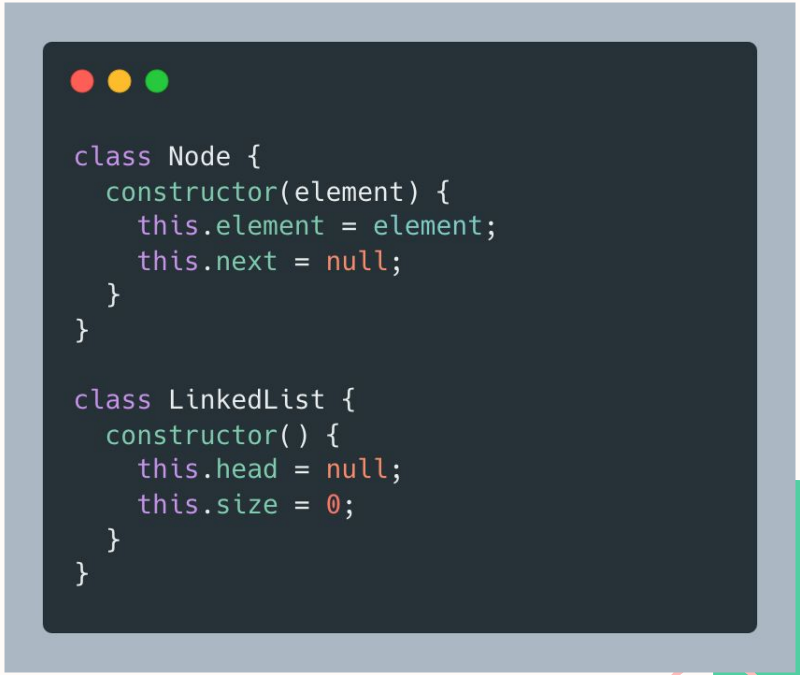
In JavaScript, a single linked list look like this :



```
const list = {
  head: {
    value: 1,
    next: {
      value: 2,
      next: {
        value: 3,
        next: {
          value: 4,
          next: null,
        },
      },
    },
  },
};
```

Linked List Implementation

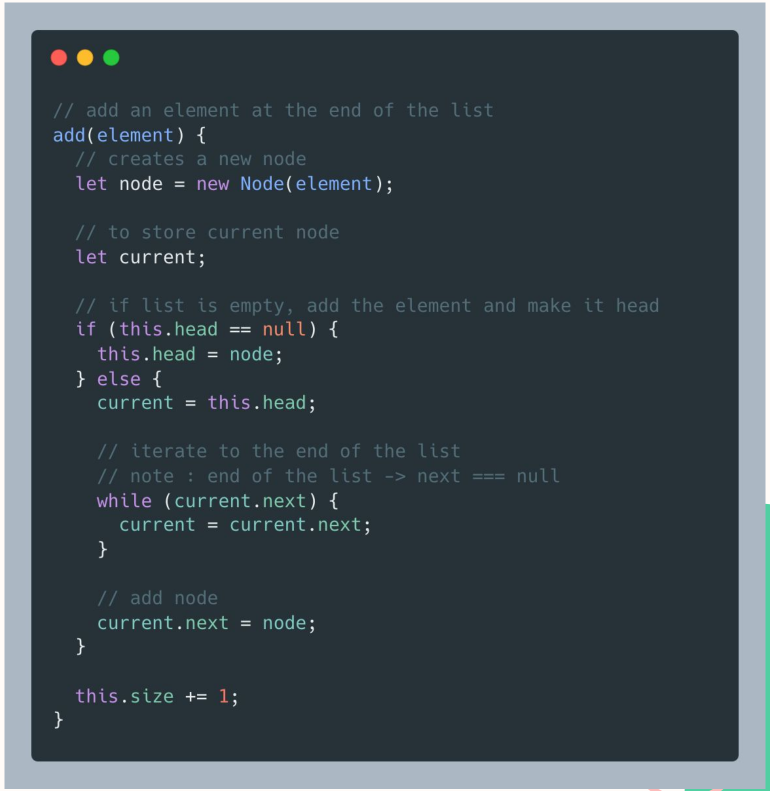
- Create a class named as **Node**.
- Create properties named **element** and **next**
- Create a class named as **LinkedList**.
- Create private properties named **head** and **size**. Set default size to 0.



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

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

Linked List Implementation

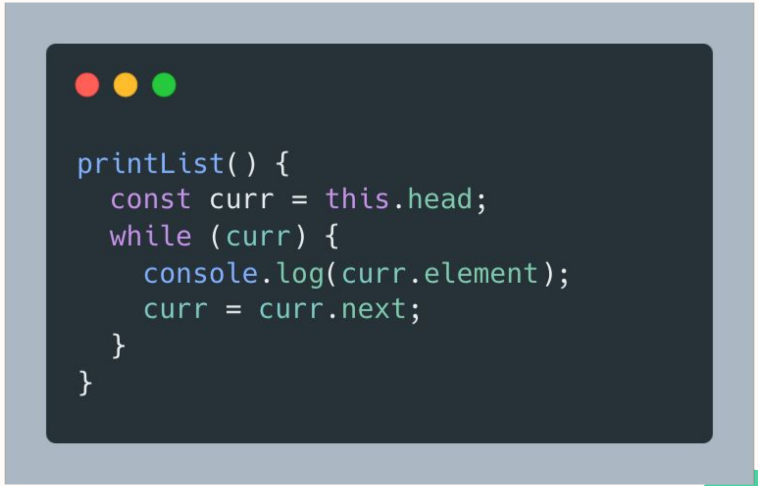
- Create **add** element method in **LinkedList** class, to add element at the end of the list.
- Create new node. If the list is empty, then set the element as a head. Otherwise, set element at the end of the list.



```
// add an element at the end of the list
add(element) {
  // creates a new node
  let node = new Node(element);

  // to store current node
  let current;

  // if list is empty, add the element and make it head
  if (this.head == null) {
    this.head = node;
  } else {
    current = this.head;

    // iterate to the end of the list
    // note : end of the list -> next === null
    while (current.next) {
      current = current.next;
    }

    // add node
    current.next = node;
  }

  this.size += 1;
}
```

Linked List Implementation

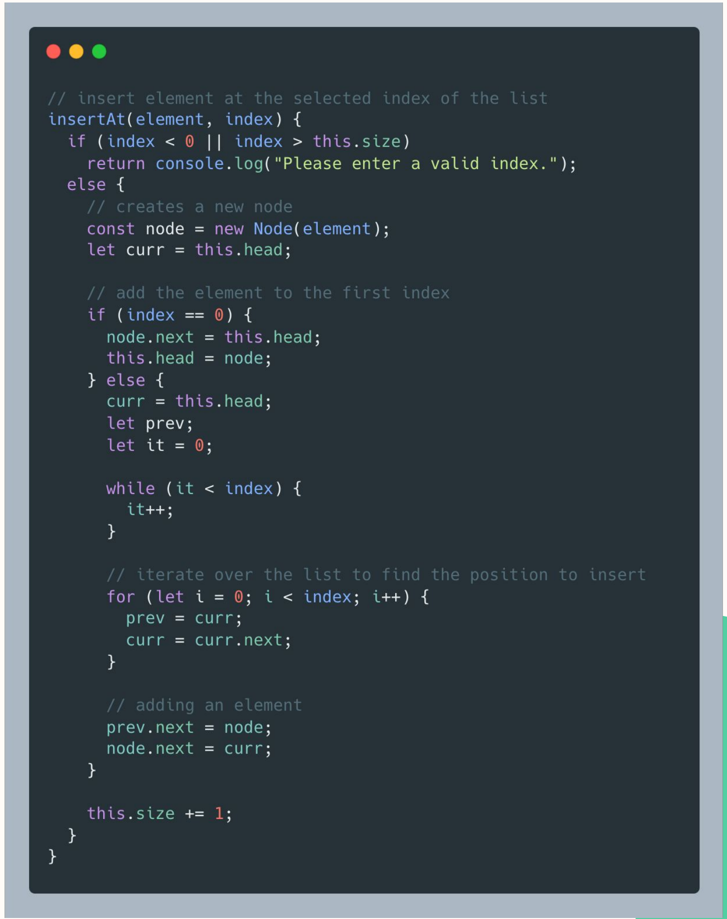
- Create **printList** method in **LinkedList** class, to print the contents of the list.



```
printList() {
  const curr = this.head;
  while (curr) {
    console.log(curr.element);
    curr = curr.next;
  }
}
```

Linked List Implementation

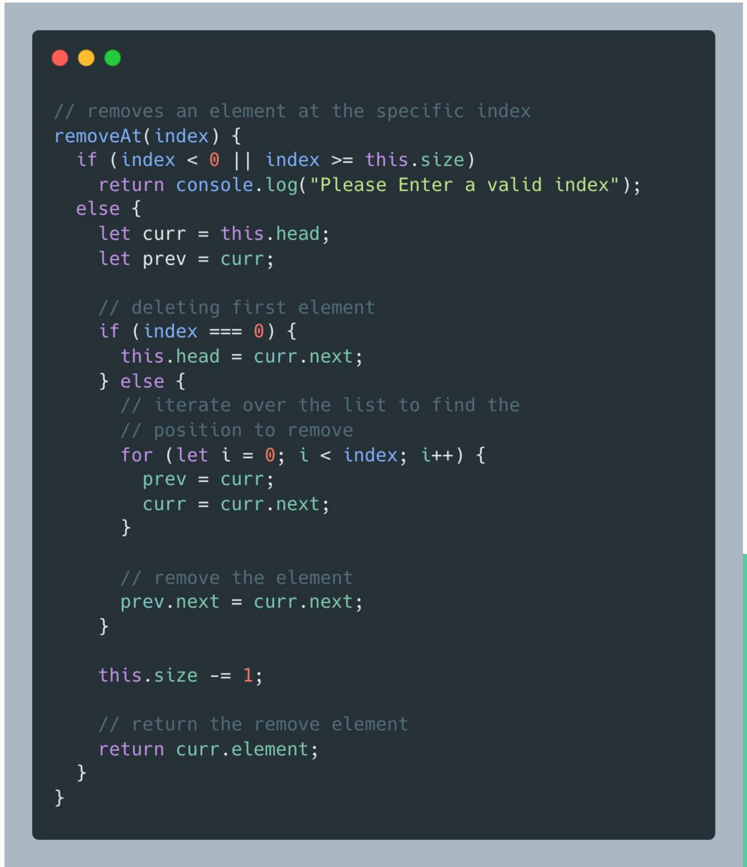
- Create **insertAt** method in **LinkedList** class, to insert element at specific index.



```
// insert element at the selected index of the list
insertAt(element, index) {
  if (index < 0 || index > this.size)
    return console.log("Please enter a valid index.");
  else {
    // creates a new node
    const node = new Node(element);
    let curr = this.head;

    // add the element to the first index
    if (index == 0) {
      node.next = this.head;
      this.head = node;
    } else {
      curr = this.head;
      let prev;
      let it = 0;

      while (it < index) {
        it++;
      }

      // iterate over the list to find the position to insert
      for (let i = 0; i < index; i++) {
        prev = curr;
        curr = curr.next;
      }

      // adding an element
      prev.next = node;
      node.next = curr;
    }

    this.size += 1;
  }
}
```

Linked List Implementation

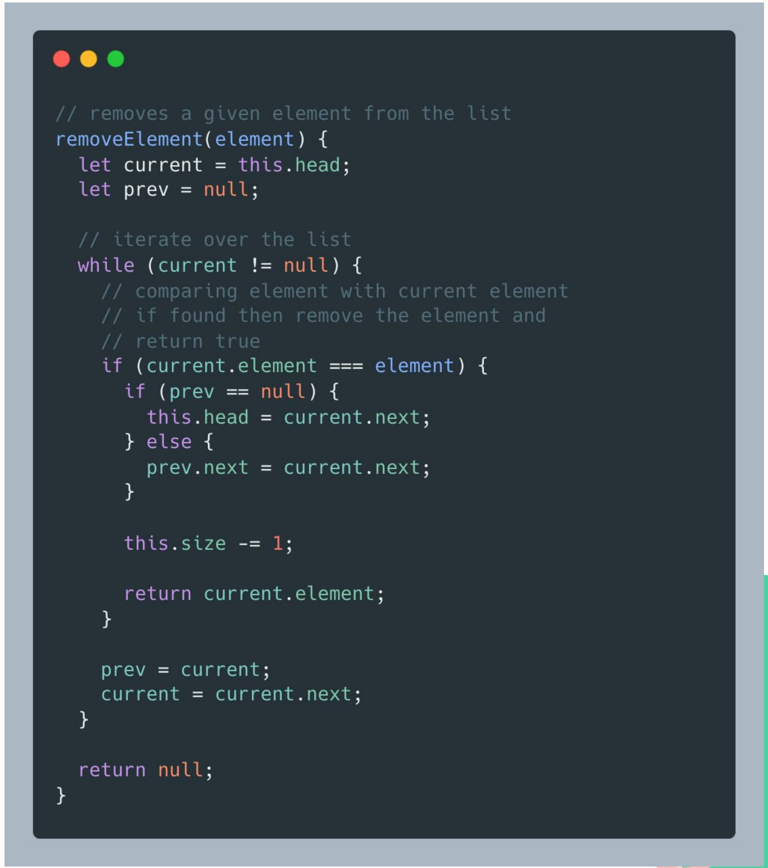
- Create **removeAt** method in **LinkedList** class, to remove element at specific index from the list.



```
// removes an element at the specific index
removeAt(index) {
  if (index < 0 || index >= this.size)
    return console.log("Please Enter a valid index");
  else {
    let curr = this.head;
    let prev = curr;

    // deleting first element
    if (index === 0) {
      this.head = curr.next;
    } else {
      // iterate over the list to find the
      // position to remove
      for (let i = 0; i < index; i++) {
        prev = curr;
        curr = curr.next;
      }

      // remove the element
      prev.next = curr.next;
    }

    this.size -= 1;

    // return the remove element
    return curr.element;
  }
}
```

Linked List Implementation

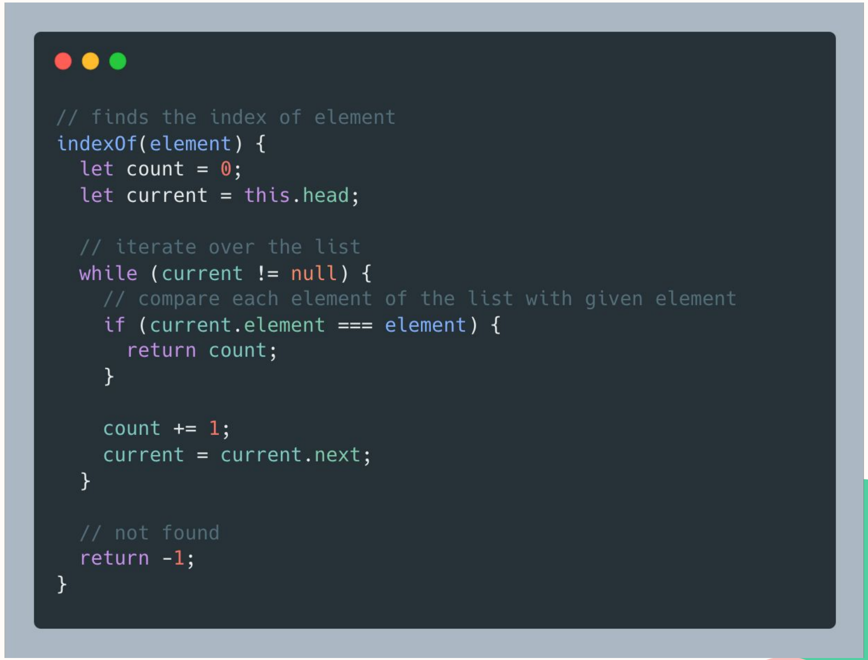
- Create **removeElement** method in **LinkedList** class, to remove element from the list.



```
// removes a given element from the list
removeElement(element) {
  let current = this.head;
  let prev = null;

  // iterate over the list
  while (current != null) {
    // comparing element with current element
    // if found then remove the element and
    // return true
    if (current.element === element) {
      if (prev == null) {
        this.head = current.next;
      } else {
        prev.next = current.next;
      }

      this.size -= 1;

      return current.element;
    }

    prev = current;
    current = current.next;
  }

  return null;
}
```

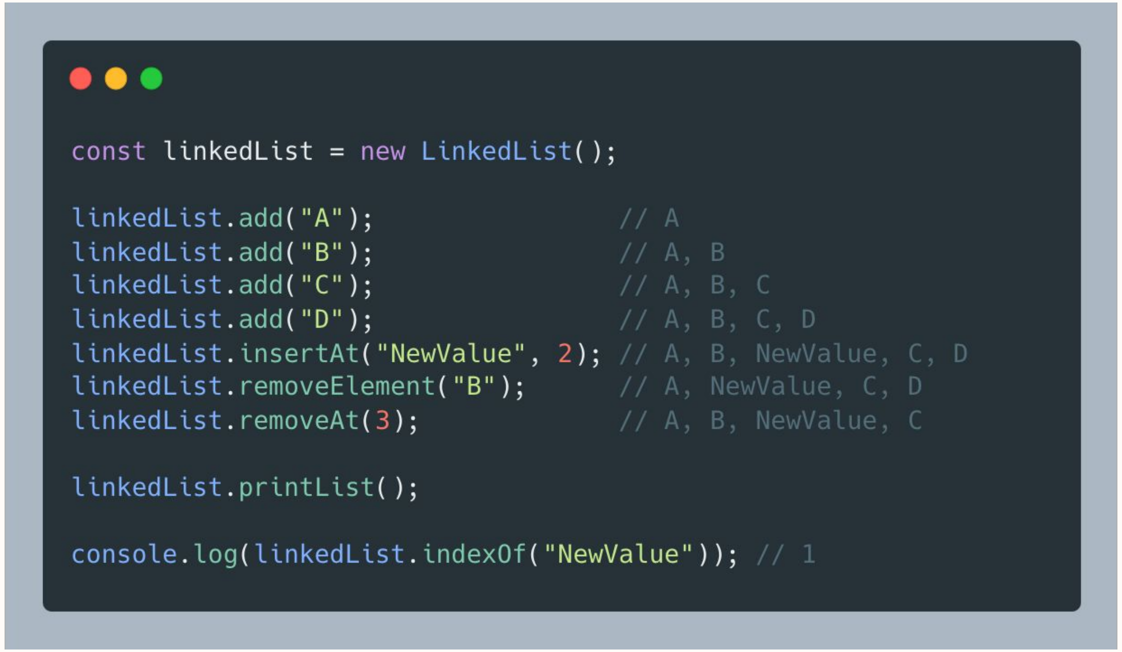
Linked List Implementation

- Create **indexOf** method in **LinkedList** class, to get index of the element.
- Create **size** method in **LinkedList** class, to get the size of the list.



```
// finds the index of element
indexOf(element) {
  let count = 0;
  let current = this.head;

  // iterate over the list
  while (current != null) {
    // compare each element of the list with given element
    if (current.element === element) {
      return count;
    }

    count += 1;
    current = current.next;
  }

  // not found
  return -1;
}
```

Linked List Implementation



```
const linkedList = new LinkedList();

linkedList.add("A");                    // A
linkedList.add("B");                    // A, B
linkedList.add("C");                    // A, B, C
linkedList.add("D");                    // A, B, C, D
linkedList.insertAt("NewValue", 2); // A, B, NewValue, C, D
linkedList.removeElement("B");          // A, NewValue, C, D
linkedList.removeAt(3);                 // A, B, NewValue, C

linkedList.printList();

console.log(linkedList.indexOf("NewValue")); // 1
```

Reference

https://www.youtube.com/watch?v=d_XvFOkQz5k

Thank You!