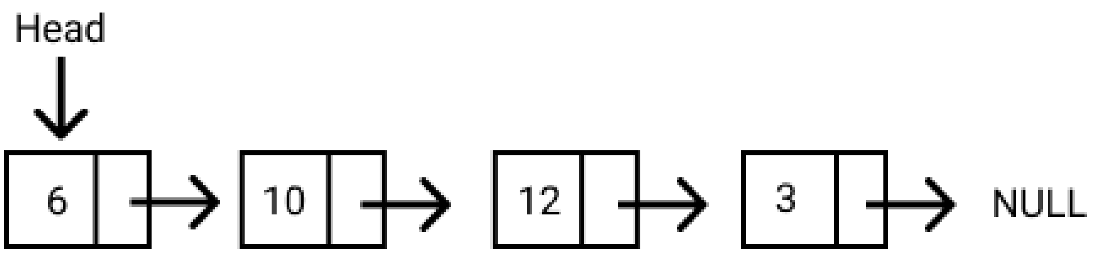
Agenda

- **What is a Linked List**
- **Intro to Singly Linked Lists**

What is a Linked List?

A linked list is a linear data structure similar to an array. However, unlike arrays, elements are not stored in a particular memory location or index. Rath each element is a separate object that contains a pointer or a link to the next object in that list.

Each element (commonly called nodes) contains two items: the data stored and a link to the next node. The data can be any valid data type. You can se this illustrated in the diagram below.



The entry point to a linked list is called the head. The head is a reference to the first node in the linked list. The last node on the list points to null. If a li is empty, the head is a null reference.

In JavaScript, a linked list looks like this:

```
const list = {
    head: {
        value: 6
        next: {
            value: 10
            next: {
                value: 12
                next: {
                    value: 3
                    next: null
                    }
                }
            }
        }
    }
};
```

An advantage of Linked Lists

- Nodes can easily be removed or added from a linked list without reorganizing the entire data structure. This is one advantage it has over arrays.

Disadvantages of Linked Lists

- Search operations are slow in linked lists. Unlike arrays, random access of data elements is not allowed. Nodes are accessed sequentially starting from the first node.
- It uses more memory than arrays because of the storage of the pointers.

Types of Linked Lists

There are three types of linked lists:

- **Singly Linked Lists**: Each node contains only one pointer to the next node. This is what we have been talking about so far.
- **Doubly Linked Lists**: Each node contains two pointers, a pointer to the next node and a pointer to the previous node.
- **Circular Linked Lists**: Circular linked lists are a variation of a linked list in which the last node points to the first node or any other node before it, thereby forming a loop.

Implementing a List Node in JavaScript

As stated earlier, a list node contains two items: the data and the pointer to the next node. We can implement a list node in JavaScript as follows:

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

Implementing a Linked List in JavaScript

The code below shows the implementation of a linked list class with a constructor. Notice that if the head node is not passed, the head is initialized null.

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

Putting it all together

Let's create a linked list with the class we just created. First, we create two list nodes, `node1` and `node2` and a pointer from node 1 to node 2.

```
let node1 = new ListNode(2)
let node2 = new ListNode(5)
node1.next = node2
```

Next, we'll create a Linked list with the node1.

```
let list = new LinkedList(node1)
```

Let's try to access the nodes in the list we just created.

```
console.log(list.head.next.data) //returns 5
```

Each element of a linked list data structure must have the followin properties:

- `value` : The value of the element
- `next` : A pointer to the next element in the linked list (`null` if there is none)

Some LinkedList methods

Next up, we will implement four helper methods for the linked list. They are:

1. size()
2. clear()
3. getLast()

4. getFirst()

1. size()

This method returns the number of nodes present in the linked list.

```
size() {
    let count = 0;
    let node = this.head;
    while (node) {
        count++;
        node = node.next
    }
    return count;
}
```

2. clear()

This method empties out the list.

```
clear() {
    this.head = null;
}
```

3. getLast()

This method returns the last node of the linked list.

```
getLast() {
    let lastNode = this.head;
    if (lastNode) {
        while (lastNode.next) {
            lastNode = lastNode.next
        }
    }
    return lastNode
}
```

4. getFirst()

This method returns the first node of the linked list.

```
getFirst() {
    return this.head;
}
```

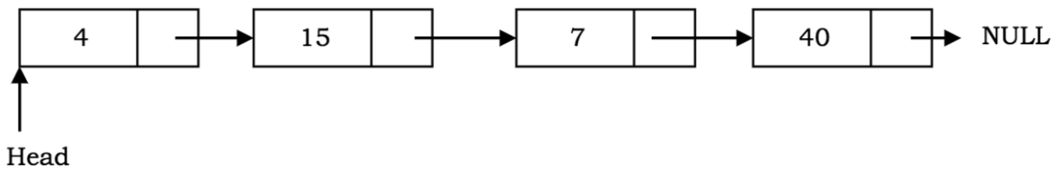
The main operations of a linked list data structure are:

- `insertAt` : Inserts an element at the specific index
- `removeAt` : Removes the element at the specific index
- `getAt` : Retrieves the element at the specific index
- `clear` : Empties the linked list

- `reverse` : Reverses the order of elements in the linked list

Singly linked list

A singly linked list is collection of nodes wherein each node has 2 parts: data and a pointer to the next node. The list terminates with a node pointir at `null` .



Main operations on a linked list are: *insert and delete.*

Before going into the details of these two operations, lets define a node class and see how this node class, along with the linked list class will help building a linear list of nodes.

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

In the above code, a Node class is defined. When an instance of the Node class is formed, the constructor function will be called to initialize the obje with two properties, data and a pointer named `next` . The pointer `next` is initialized with a default value of `null` , incase no value is passed as a argument.

What follows next is a linked list class which maintains the `head` pointer of the list.

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

In the above code, a LinkedList class is defined. When an instance of the LinkedList class is formed, the constructor function will be called to initialize tl object with a property, `head` . The `head` pointer is assigned a value of `null` because when a linked list object is initially created it does not conta any nodes. It is when we add our first node to the linked list, we will assign it to the `head` pointer.

To create an instance of the LinkedList class, we will write:

```
// A list object is created with a property head, currently pointing at null

let list = new LinkedList();
```

After creating a node class and a linked list class, lets now look at the insert and delete operations performed on a singly linked list.

Insert operation on a singly linked list

An insert operation will insert a node into the list. There can be three cases for the insert operation.

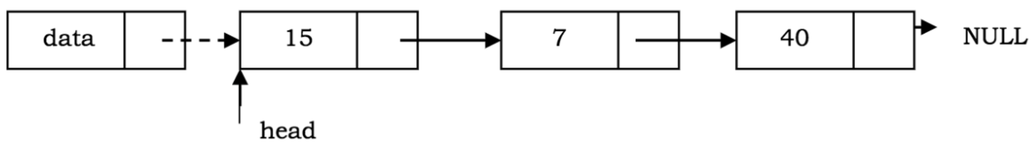
- Inserting a new node before the head (at the beginning of the list).
- Inserting a new node after the tail (i.e. at the end of the list).
- Inserting a new node in the middle of the list (at a given random position).

Inserting a node at the beginning of the singly linked list.

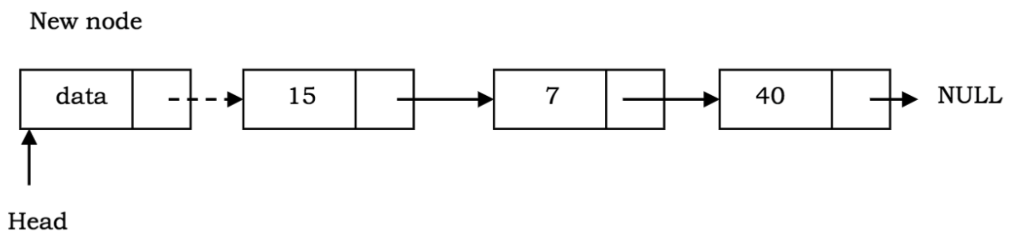
In this case, a new node is added before the current head node. To fulfill this operation we will first create a node. The newly created node will be havir
two properties as defined in the constructor function of the Node class, data and next.

```
LinkedList.prototype.insertAtBeginning = function(data){
// A newNode object is created with property data and next = null
    let newNode = new Node(data);
// The pointer next is assigned head pointer so that both pointers now point at the same node.
    newNode.next = this.head;
// As we are inserting at the beginning the head pointer needs to now point at the newNode.

    this.head = newNode;
    return this.head;
}
```



```
newNode.next=this.head;
```



```
this.head=newNode;
```

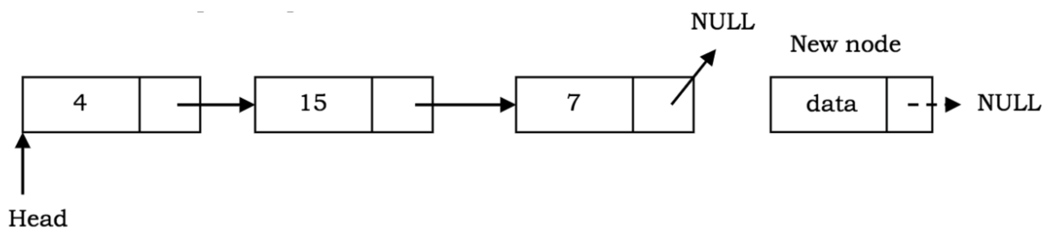
Inserting a node at the end of the singly linked list

In this case, a new node is added at the end of the list. To implement this operation we will have to traverse through the list to find the tail node ar
modify the tail's next pointer to point to the newly created node instead of `null` .

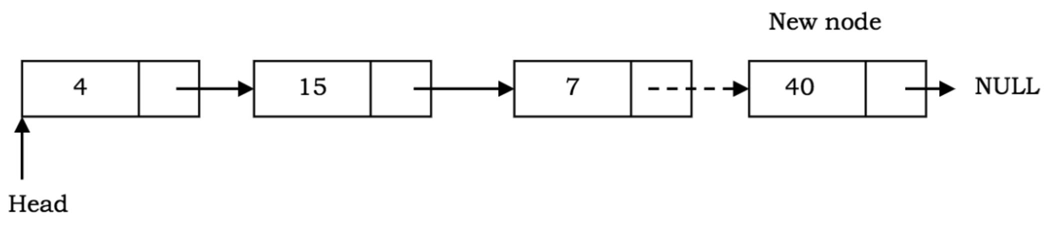
Initially, the list is empty and the `head` points to null.

```
LinkedList.prototype.insertAtEnd = function(data){
// A newNode object is created with property data and next=null

    let newNode = new Node(data);
// When head = null i.e. the list is empty, then head itself will point to the newNode.
    if(!this.head){
        this.head = newNode;
        return this.head;
    }
// Else, traverse the list to find the tail (the tail node will initially be pointing at null), and update the tail's ne
   let tail = this.head;
   while(tail.next !== null){
        tail = tail.next;
   }
   tail.next = newNode;
   return this.head;
}
```

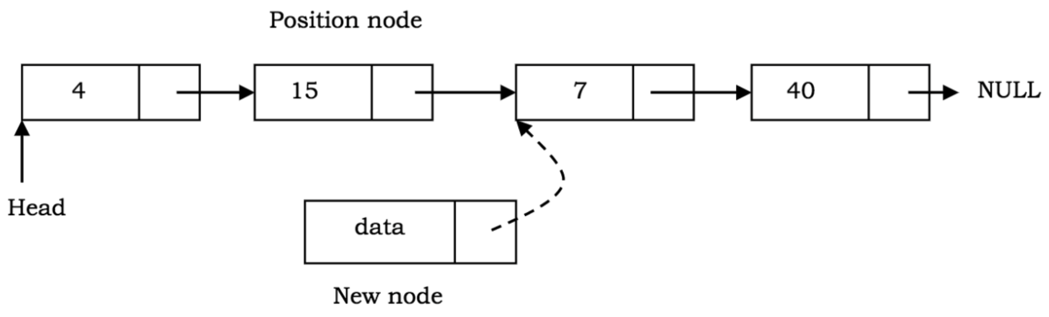


```
Traverse the linked list to find the last node (tail).
```

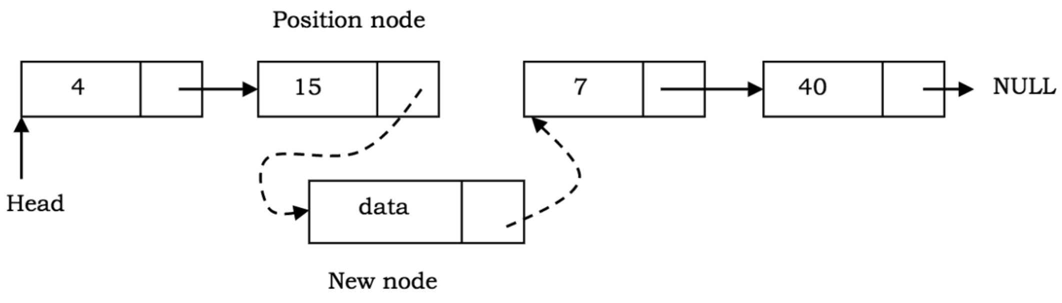


Inserting a node at given random position in a singly linked list

To implement this operation we will have to traverse the list until we reach the desired position node. We will then assign the newNode's next pointer the next node to the position node. The position node's next pointer can then be updated to point to the newNode.



```
newNode.next = previous.next;
```



```
previous.next = newNode;
```

```
// A helper function getAt() is defined to get to the desired position. This function can also be later used for perform
```

```
    LinkedList.prototype.getAt = function(index){
        let counter = 0;
        let node = this.head;
        while (node) {
            if (counter === index) {
               return node;
            }
            counter++;
            node = node.next;
        }
        return null;
    }
// The insertAt() function contains the steps to insert a node at a given index.
    LinkedList.prototype.insertAt = function(data, index){
// if the list is empty i.e. head = null
        if (!this.head) {
            this.head = new Node(data);
            return;
        }
// if new node needs to be inserted at the front of the list i.e. before the head.
        if (index === 0) {
           this.head = new Node(data, this.head);
           return;
        }
// else, use getAt() to find the previous node.
        const previous = this.getAt(index - 1);
        let newNode = new Node(data);
        newNode.next = previous.next;
        previous.next = newNode;

        return this.head
   }
```

Delete operation on a singly linked list

A delete operation will delete a node from the list. There can be three cases for the delete operation.

- Deleting the first node.
- Deleting the last node.
- Deleting a node from the middle of the list (at a given random position).

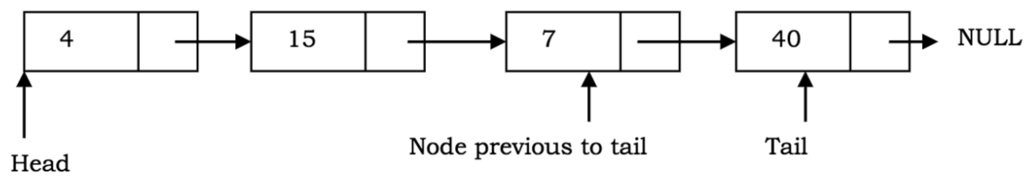
Deleting the first node in a singly linked list

The first node in a linked list is pointed by the `head` pointer. To perform a delete operation at the beginning of the list, we will have to make the ne node to the head node as the new `head` .

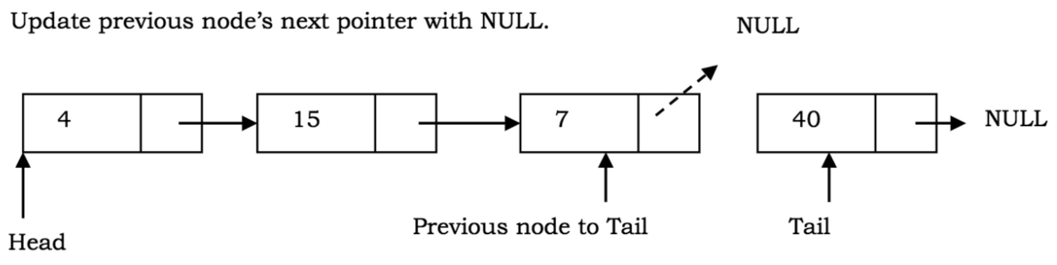
```
LinkedList.prototype.deleteFirstNode = function(){
    if(!this.head){
        return;
    }
    this.head = this.head.next;
    return this.head;
}
```

Deleting the last node in a singly linked list

To remove the last node from the list, we will first have to traverse the list to find the last node and at the same time maintain an extra pointer to point the node before the last node. To delete the last node, we will then set the next pointer of the node before the last node to null.



```
Traverse the list to find the tail and the node previous to the tail.
```



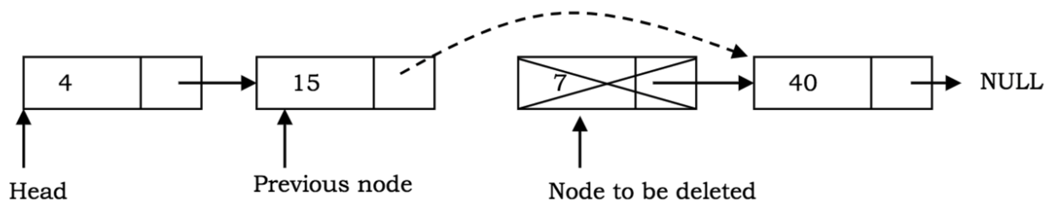
```
LinkedList.prototype.deleteLastNode = function(){
    if(!this.head){
        return null;
    }
    // if only one node in the list
    if(!this.head.next){
        this.head = null;
        return;
    }
   let previous = this.head;
   let tail = this.head.next;

   while(tail.next !== null){
       previous = tail;
       tail = tail.next;
   }

   previous.next = null;
   return this.head;
}
```

Deleting a node from given random position in a singly linked list

Similar to the above case, we will first have to traverse the list to find the desired node to be deleted and at the same time maintain an extra pointer point at the node before the desired node.



```
previous.next = previous.next.next;
```

```
LinkedList.prototype.deleteAt = function(index){
// when list is empty i.e. head = null
    if (!this.head) {
         this.head = new Node(data);
         return;
     }
// node needs to be deleted from the front of the list i.e. before the head.
    if (index === 0) {
        this.head = this.head.next;
        return;
    }
// else, use getAt() to find the previous node.
    const previous = this.getAt(index - 1);

    if (!previous || !previous.next) {
        return;
    }

    previous.next = previous.next.next;
    return this.head
}
```

Deleting the singly linked list

Now, lets delete the complete linked list. This can be done by just one single line of code.

```
LinkedList.prototype.deleteList = function(){
    this.head = null;
}
```

Lets have a look at another example of singly linked list

Create a Node:

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

Create a LinkedList class:

```
class LinkedList{
    //Creates a linkedlist with passed value.
    constructor(value){
        //Creates a head node
        this.head = {
            value: value,
            next : null
        };
        //As there is only one element in the list, head is also a tail
        this.tail = this.head;
        //Length would be 1
        this.length = 1;
    }
}
```

Initializing the Linked List:

```
const myLinkedList = new LinkedList(20);
```

Print the LinkedList

```
printList(){
    //Creates an empty array.
    let printArrayList = [];

    //Pointer which points to the head node
    let currentNode = this.head;

    //Start iterating from the first node till you reach the last node
    while(currentNode !== null){
        //Add every node's value to the array
        printArrayList.push(currentNode.value);

        //Point pointer to the next node
        currentNode = currentNode.next;
    }
    //Return the array
    return printArrayList.join(' -> ');
}
```

Add a new tail node:

```
//Add the node at the tail of the linkedlist
append(value){

    //Create a new Node by creating a instance of a Node class
    const newNode = new Node(value);

    // Check if head is present or not, if head is empty creates a head
    if (this.head == null){
        this.head = node;
    }
    else{ //Else creates a tail

    //We need to point current tail's next to the newNode
    this.tail.next = newNode;

    //Now make newNode a tail node
    this.tail = newNode;

    //Increase the length by 1
    this.length++;
    }
    return this;
}
```

```
myLinkedList.append(40).append(50);
```

Adding a new head node

```
//Add the node as a head of the linkedlist
prepend(value){
    //Create a new Node by creating a instance of a Node class
    const newNode = new Node(value);

    //Points this node's next to the head
    newNode.next = this.head;

    //Now make this node a head node
    this.head = newNode;

    //Increase the length by 1
    this.length++;

    return this;
}
```

```
myLinkedList.prepend(10);
```

Inserting a node by index

```
//Insertes a node at specified index, say we want to insert 30 at index 2
//Current list: 10 -> 20 -> 40 -> 50
insert(index, value){
    //Create a new Node by creating a instance of a Node class
    const newNode = new Node(value);

    //Counter to loop
    let count = 1;

    //Create a temp node to traverse through list, which will start from the head node
    //Pointing to 10
    let previousNode = this.head;

    //Traverse the list one node before the specified index, which is previous node
    while(count < index){
        //Point previous node to next for iterating
        previousNode = previousNode.next;

        //Increase the count to compare it with index;
        count++;
    }
    //When the loop ends you will be able to have a previous node. Which is 20 in this example.

    //First, points new node's next to the previous node's next, so it can hold the list ahead of its index
    //New node = 30, So new node will be 30 -> 40 -> 50
    newNode.next = previousNode.next;

    //Now just point previous node's next to new node.
    //Merge left side of the list, 10 -> 20 -> 30 -> 40 -> 50
    previousNode.next = newNode;
    return this;
}
```

```
myLinkedList.insert(2,30);
myLinkedList.insert(1,15);
```

Delete head:

```
deleteHead(){
     this.head = this.head.next;
     this.length--;
     return this;
}
```

```
myLinkedList.deleteHead();
```

Delete Tail:

```
deleteTail(){
     let secondLastNode = this.head;
     while(secondLastNode.next.next !== null){
         secondLastNode = secondLastNode.next;
     }
     secondLastNode.next = null;
     this.length--;
     return this;
}
```

```
myLinkedList.deleteTail();
```

Delete a node by index:

```
deleteAtIndex(index){
   //Check if index is head
   if(index === 0){
    //Appoint head to the next element
    this.head = this.head.next;
    this.length--;
    return this;
   }
let count = 1;
let previousNode = this.head;
while(count < index){
    previousNode = previousNode.next;
    count++;
}
previousNode.next = previousNode.next.next;
this.length--;
return this;
}
```

```
myLinkedList.deleteAtIndex(2);
```

Delete a node by value:

```
deleteNodeByValue(value){
    //Current node to loop through the list
    let currentNode = this.head;

    //Previous node to update its pointer to next.next node
    let previousNode = null;

    while(currentNode != null){

        //Check if we find the value we are looking for
        if(currentNode.value === value){

            //Check if it is a head or not by comparing previous node with null
            if (previousNode === null) {
                //If it is head, then update head to nextnode
                this.head = currentNode.next;
            }
            else{
                //If it is not head then simply update previous node
                previousNode.next = currentNode.next;
            }
            //Reduce length by 1
            this.length--;
        }

        //Previous node will point to this node with every iteration
        previousNode = currentNode;
        //Current node will point to next node for iteration
        currentNode = currentNode.next;
    }
    return this;
}


myLinkedList.deleteNode(40);
```

Search an element:

```
searchElement(value){
    let currentNode = this.head;
    while(currentNode !== null){
        if(currentNode.value === value) return true;
        currentNode = currentNode.next;
    }
    return false;
}

console.log(myLinkedList.searchElement(20));
```

Thank You !