What is a linked list?

A linked list is an ordered collection of data elements. A data element can be represented as a node in a linked list. Each node consists of two parts: data & pointer to the next node.

Unlike arrays, data elements are not stored at contiguous locations. The data elements or nodes are linked using pointers, hence called a linked list.

A linked list has the following properties:

```
Successive nodes are connected by pointers.
The last node points to null.
A head pointer is maintained which points to the first node of the list.
A linked list can grow and shrink in size during execution of the program.
It can be made just as long as required.
It allocates memory as the list grows. Unlike arrays, which have a fixed
size. Therefore, the upper limit on the number of elements must be known
in advance. Generally, the allocated memory is equal to the upper limit
irrespective of the usage. This is one the key advantages of using a
linked list over an array.
```

Another advantage of a linked list

In contrast to an array, which stores data contiguously in memory, a linked list can easily insert or remove nodes from the list without reorganization of the entire data structure.

Few drawbacks:

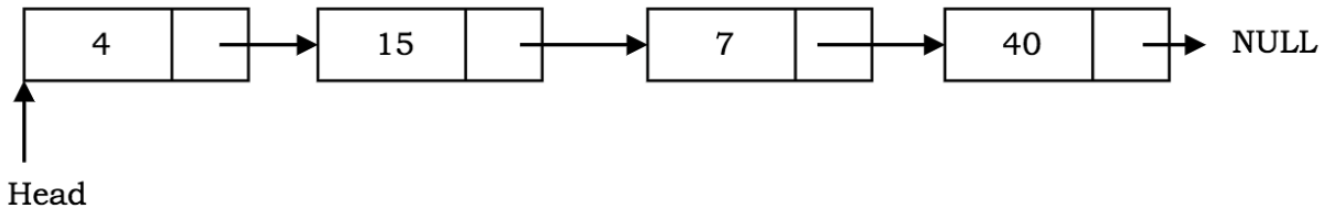
```
Random access of data elements is not allowed. Nodes must be accessed
sequentially starting from the first one. Therefore, search operation is
slow on a linked list.
It uses more memory than arrays because of the storage used by their
pointers.
```

Types of Linked lists

There are a few different types of linked lists. But the most popular ones are: singly, doubly and circular.

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 pointing 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 in 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 object 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 an 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 the 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 contain 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();
```

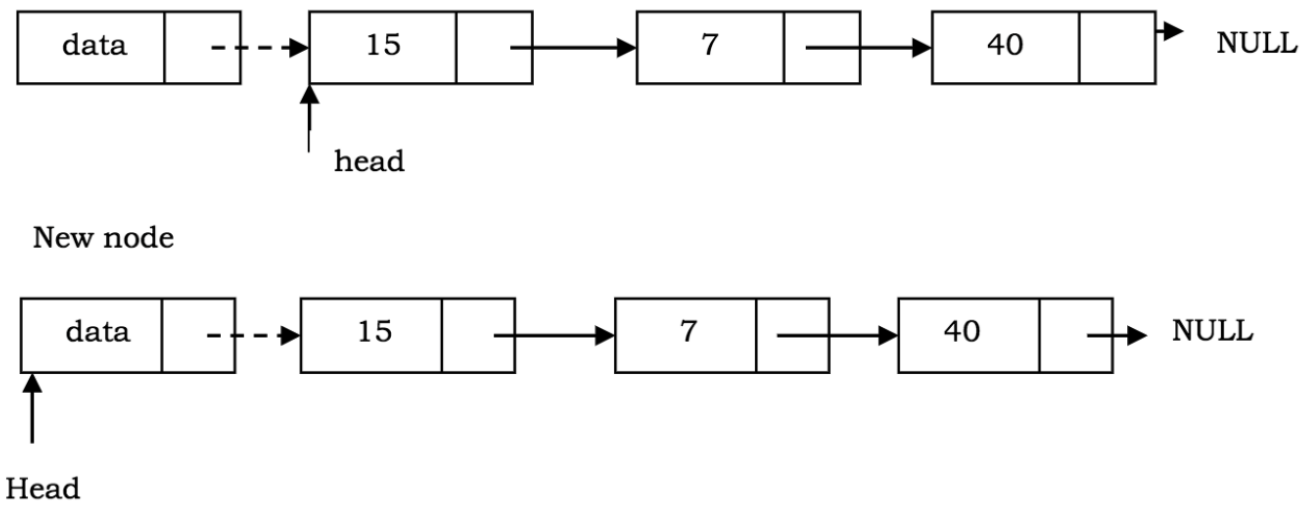
After creating a node class and a linked list class, lets now look at the insert and delete operations performed on a singly linked list.

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 having two properties as defined in the constructor function of the Node class, data and next.

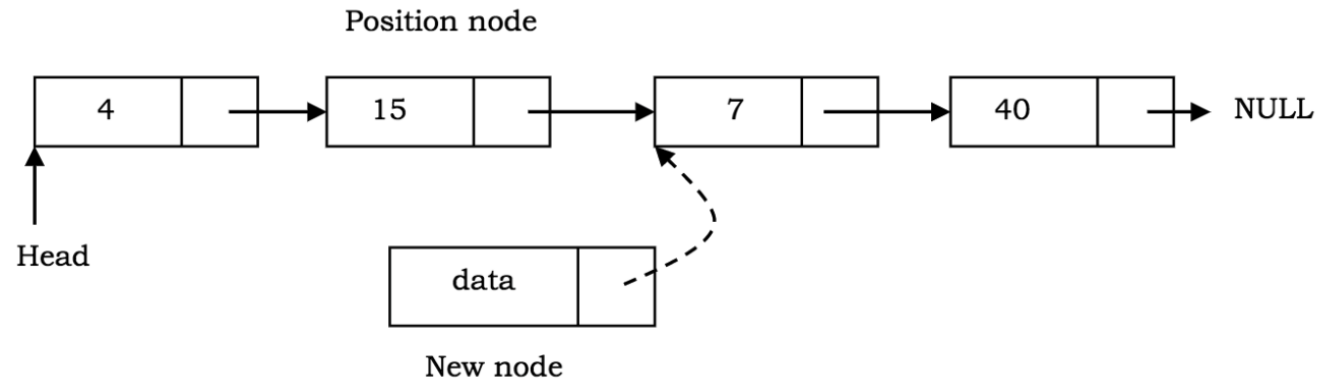
```
LinkedList.prototype.insertAtBeginning = function(data){
// A newNode object is created with property data and next = null
    let newNode = new Node(data);
// The pointer next is assigned head pointer so that both pointers now
point at the same node.
    newNode.next = this.head;
// As we are inserting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

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

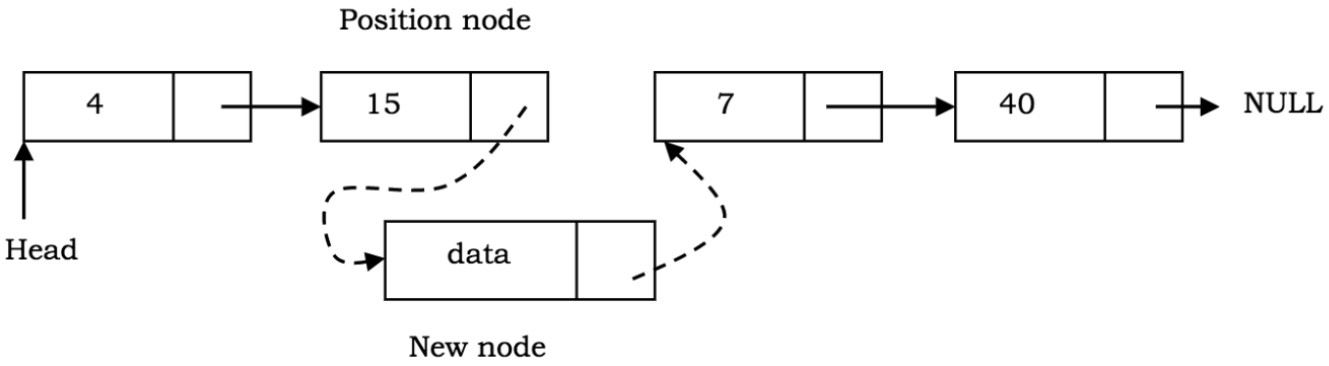


Inserting a node at the end of the singly linked list.

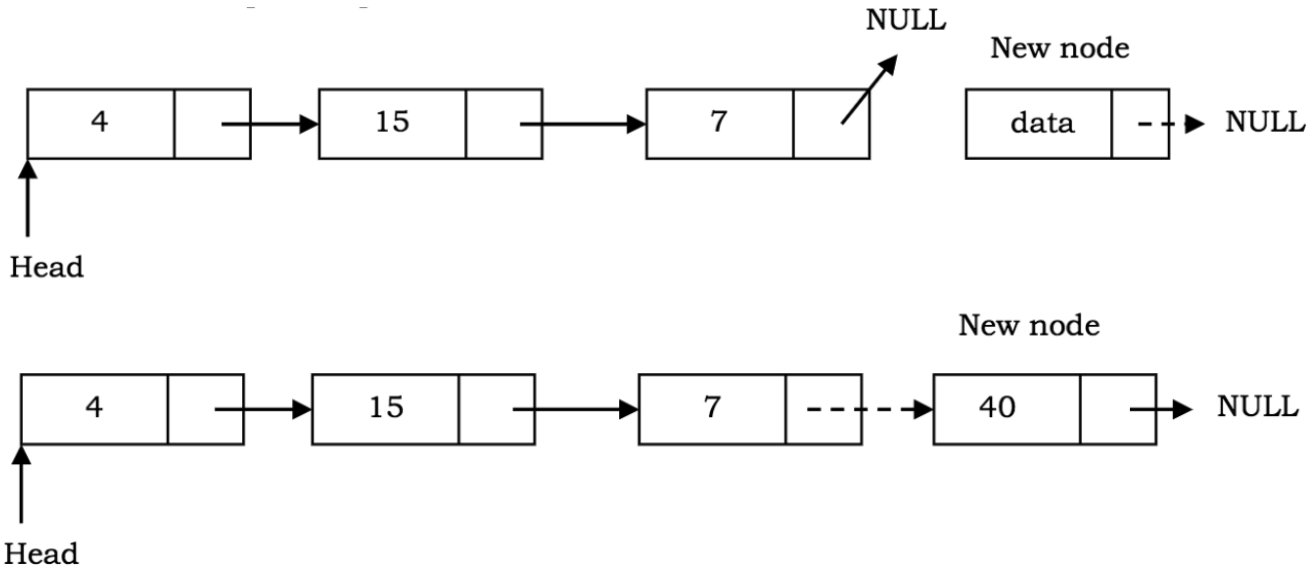
In this case, a new node is added at the end of the list. To implement this operation we will have to traverse through the list to find the tail node and modify the tail's next pointer to point to the newly created node instead of null.

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 next pointer.
   let tail = this.head;
   while(tail.next !== null){
        tail = tail.next;
   }
   tail.next = newNode;
   return this.head;
}
```



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 to 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 performing delete operation from
a given position.
    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.

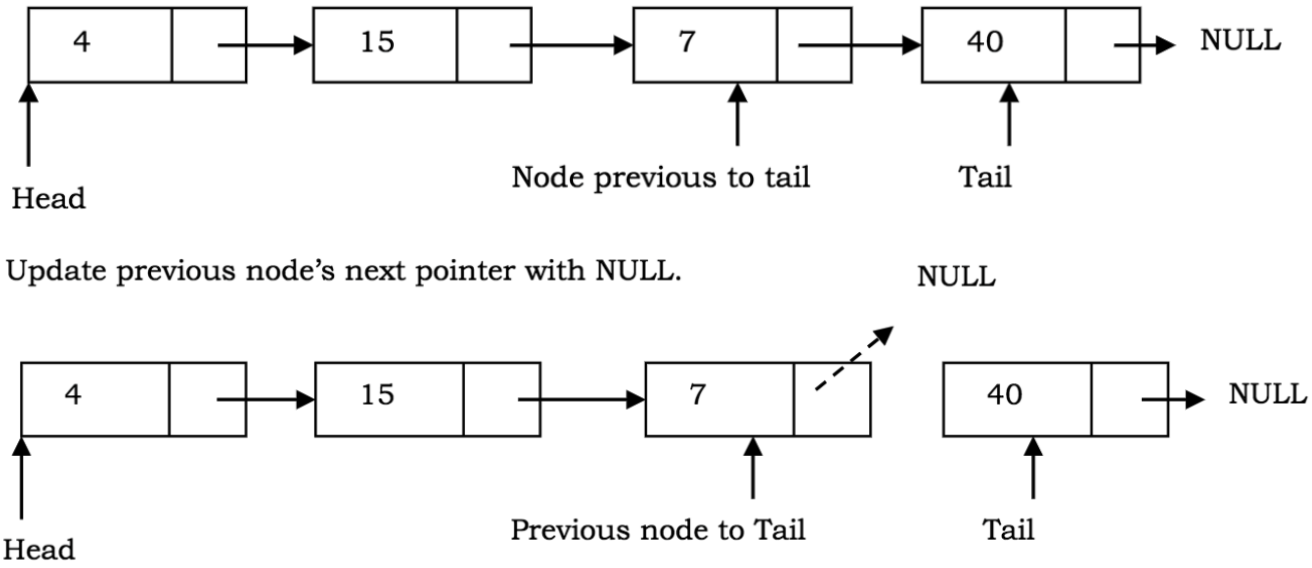
```
Deleting the first node.
Deleting the last node.
Deleting a node from the middle of the list (at a given random position).
Deleting the first node in a singly linked list
```

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 next 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 at 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.

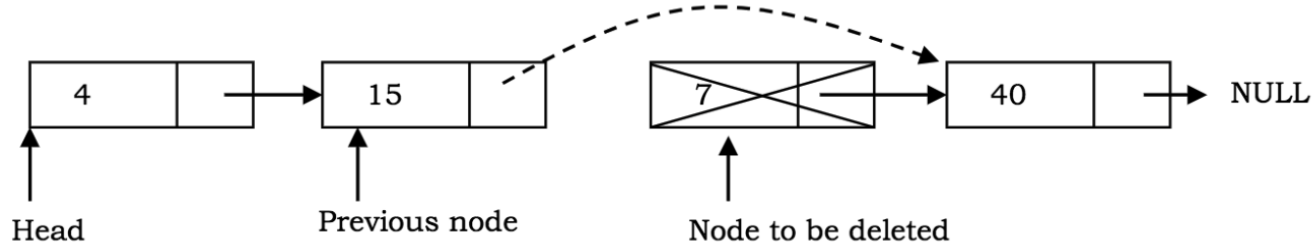


Traverse the list to find the tail and the node previous to the tail.

```
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 to 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;
}
```