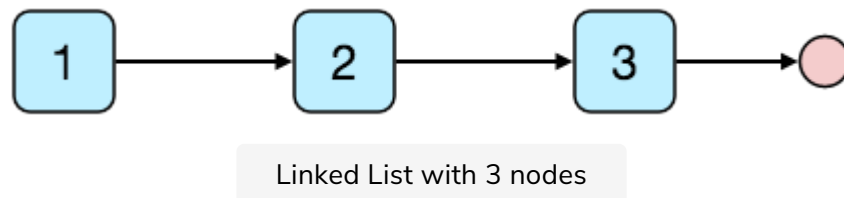


Linked Lists

Linked lists are an ordered set of nodes where each node contains a link to its successor.

Introduction

A linked list is a data structure that holds a sequence of linked nodes. Each node contains data and a reference, where reference points to its successor node in the Linked List.



The above diagram shows a typical example of a singly linked list. It consists of 3 nodes. The first node consists of a value of 1 and contains a pointer to the next node. The next node in the list has a value of 2 and subsequently consists of a reference to the last node in the list and so on. The last node in a linked list contains a value 3 but it doesn't point to anything.

When a reference doesn't point to any valid object, programming languages typically have a "NULL" value to represent the missing value. In Javascript, we use "null" for that purpose - and it's called a *null reference*.

Reference is also sometimes called a pointer. This comes from the C/C++ world where variables can point to memory addresses. In Javascript, we cannot point to memory addresses. Instead we have references to objects.

The first node in the linked list is called **Head Node**. Head node is our handle to the linked list. We traverse the linked list starting at the head node and then move to the node pointed by the current node. We'll visit the traversal in detail later.

Why another 'List' when we already have an array?

It's a valid question. We already have a built-in type Array that acts as a container similar to a linked list. Turns out that array is not an efficient data structure in all cases. Arrays are implemented in programming languages in a certain way which makes it very efficient for certain cases while they are highly inefficient for other cases.

Given how arrays are stored in memory, it is usually inefficient to insert elements in the middle.

However, linked list also has drawbacks. You cannot access an element randomly and have to traverse from the first node to the next and so on until you find your desired node.

If you need to access elements at random positions in the list, don't use a Linked List.

The Node element in JavaScript

As shown above , a Linked List consists of Node elements. The below code snippet shows the typical code structure for a Node element.

```
var node = {  
  data: 0,           // some data value  
  next: null // reference to next node  
}
```



You can see that the Node element consists of two fields:

1. Data field
2. Reference to next Node

Initially the next field will point to null . As we create more nodes in the chain, we will update the next reference to point to the next node in the linked list.

First Take – Manually Build a Linked List

First Take - Manually Build a Linked List

Okay, let's look at a very simplistic example of how we can implement Linked Lists in JavaScript. We will create three nodes as shown in the above diagram. We'll then traverse the linked list and print the values on the console.

> *Run the code below and see how it works*

JavaScript

HTML

CSS (SCSS)

```
var firstNode = {
  data:100,
  next: null
};




var secondNode = {
  data: 200,
  next: null
};

var thirdNode = {
  data:300,
  next: null
};

firstNode.next = secondNode;
secondNode.next = thirdNode;

var currentNode = firstNode;

// Iterate over the Linked List and print nodes
while (currentNode != null) {
  console.log("Node value: " + currentNode.data);
  currentNode = currentNode.next;
}
```



Console

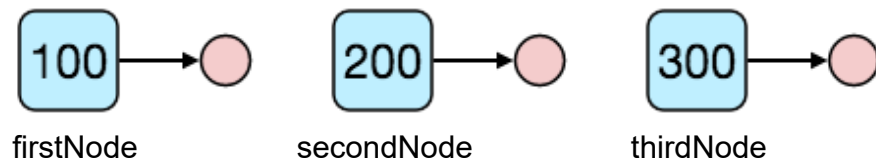
 Clear

Node value: 100

Node value: 200

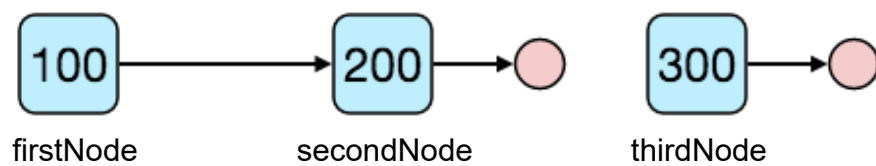
Node value: 300

The below animation shows the steps that we followed to create the linked list



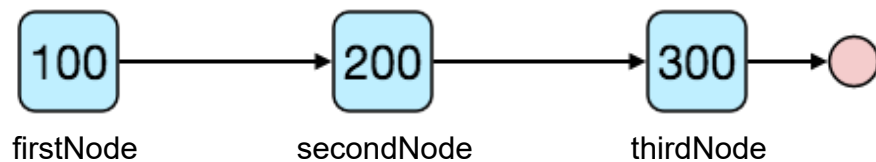
First, we create 3 nodes.
Next references for all nodes point to *null*.

1 of 3



Connect firstNode to secondNode.

2 of 3



Connect secondNode to thirdNode.
thirdNode is the last node and keeps pointing to null.

3 of 3



Let's take a quick quiz.

Quiz #1

Q

What is the general declaration of a node in a linked list

Check Answers

Generalizing Linked List Operations

It's obvious from the above code that if we wanted to create multiple nodes, we will have to write the same lines of code over and over again (to iterate over the linked list, find the last element and extend the linked list by adding a new node). Instead, we will create a `LinkedList` class that provides such functions over the linked list.

The below code snippet shows how linked lists can be generalized in JavaScript.

> *Run the code below and see how it works*

JavaScript

HTML

CSS (SCSS)

```
function Node(data) {
  this.data = data;
  this.next = null;
}

function LinkedList() {
  this._length = 0;
  this._head = null;
}

LinkedList.prototype.push = function(data) {
  // Create a new node with Data
  var node = new Node(data);
```

```

// We are inserting the first node in the list
if (this._head === null) {
    this._head = node;
} else {
    // Find the last node
    var current = this._head;

    while (current.next) {
        current = current.next;
    }

    current.next = node;
}

// Increment the length
this._length++;
}

// We follow the 0 based indexes just like Arrays
LinkedList.prototype.itemAt = function(index) {
    // Ensure that the index is within bounds
    if (index < 0 || index >= this._length) {
        // Return Null when index is out of bounds
        return null;
    }

    var current = this._head;
    var counter = 0;

    while (counter < index) {
        current = current.next;
        counter++;
    }

    return current.data;
}

// Returns Size of Current Linked List
LinkedList.prototype.size = function() {
    return this._length;
}

// Let's create a Linked List and add 3 nodes
var list = new LinkedList();
list.push(100);
list.push(200);
list.push(300);

for (i = 0; i < list.size(); i++) {
    console.log("Node value: " + list.itemAt(i));
}

```



Console

Clear

Node value: 100

Node value: 200

The below animation shows what the code does to create a linked list



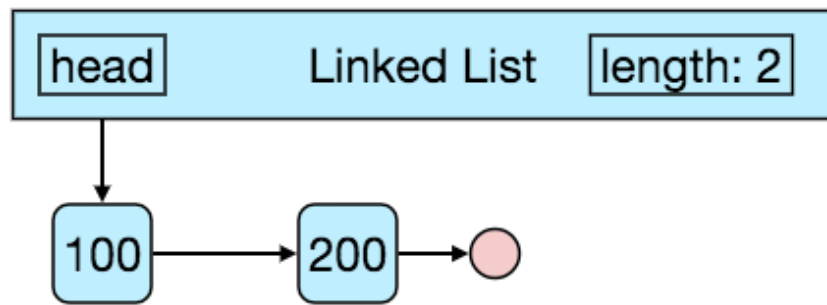
Initial state of linked list object.
Head points to *null* and length is 0.

1 of 4



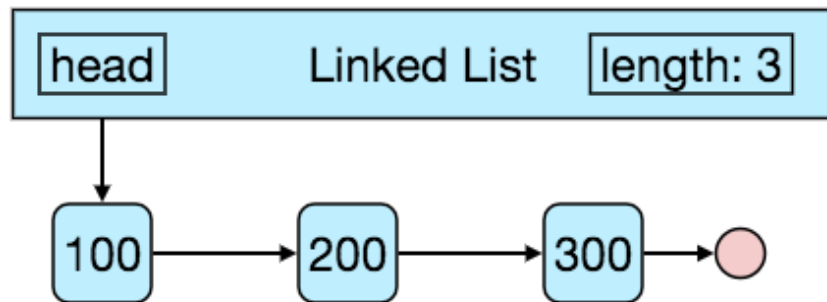
Inserting the first node with value 100.
Head points to the first node and length is 1.

2 of 4



Inserting the second node with value 200.
First node points to second node and length is 2.

3 of 4



Inserting the third node with value 300.
Second node points to third node and length is 3.

4 of 4



Taking a closer look at the Linked List Code

Feel free to skip to Quiz #2 below if you've already understood the above code.

The first section of code created our *Node* class. The function takes the data value, assigns it to the node via the 'this' attribute. Initially, the function will cause the node to point to nothing because at this point in time we don't know where the node needs to be inserted in the linked list.

```
function Node(data) {  
  this.data = data;
```




```
this.next = null;  
}
```

Next we defined our linked list. The linked list will initially have a length of 0 and the first element or the head element will point to nothing. Simple enough.

```
function LinkedList() {  
  this._length = 0;  
  this._head = null;  
}
```



Next we add a *push* function to our linked list class. This function will be used to append nodes to the linked list. There are two cases while appending a node to the linked list.

1. If the linked list is empty, then we are inserting the first node. This node will become the head node.
2. Linked List already has one or more nodes. In this case, node will be inserted at the tail of the linked list.

We create the node and then insert the node at the right position. Note how we traverse to the last element in the linked list. We have a temporary variable called *current* which is updated in the loop to the next element in the list. When *current.next* becomes null, it means that we have reached the end of the list. Here, the while loop will terminate. We point the *next* of the last node to the newly created node.

As the last step, we increment the size of the linked list.

```
LinkedList.prototype.push = function(data) {  
  // Create a new node with Data  
  var node = new Node(data);  
  
  // We are inserting the first node in the list  
  if (this._head === null) {  
    this._head = node;  
  } else {  
    // Find the last node  
    var current = this._head;  
  
    while (current.next) {  
      current = current.next;  
    }  
  }  
}
```



```

        current.next = node;
    }

    // Increment the length
    this._length++;
}

```

Similar to the *push* method, we introduced *itemAt* method. This method retrieves and returns the value of node at the given index. It's similar to array's lookup method. As the first step, we ensure that the Linked List has enough elements. If index is too low (less than 0) or too high (greater than the length), we simply return null.

We could have considered throwing an exception here as well (many programming languages throw `OutOfBoundsException` in such cases).

Now, if the linked list has enough items, we traverse to the desired index, starting from the head node. Remember what we discussed in the beginning as a `LinkedList` v.s. `Array` tradeoff. In `Array`, we could have randomly accessed a given index. Here, we have to iterate through the linked list in a serial order until we get to the desired node. Once we are at the node, we return the data contained by that node.

```

// We follow the 0 based indexes just like Arrays
LinkedList.prototype.itemAt = function(index) {
    // Ensure that the index is within bounds
    if (index < 0 || index >= this._length) {
        // Return Null when index is out of bounds
        return null;
    }

    var current = this._head;
    var counter = 0;

    while (counter < index) {
        current = current.next;
        counter++;
    }

    return current.data;
}

```



Quiz #2 - Generalized Linked List

1

What does the code line `this.head` normally refer to

2

When adding a node to a linked list there is normally a special case condition which is identified by the code snippet `if (this._head === null){ this._head = node;}`. What does this code snippet actually do?

[Check Answers](#)

Removing a Node from the Linked List

Let's see how to remove nodes from a linked list. Our remove function will take the index of the element to be removed. If the node to be removed happens to be the head node, it will update the head node as well.

> *Run the code below and see how it works.*

JavaScript

HTML

CSS (SCSS)

```
// Removes the element and returns the data
// in the node that was removed
LinkedList.prototype.remove = function(index) {
  // Ensure that the index is within bounds
```

```

    if (index < 0 || index >= this._length) {
        return null;
    }

    var current = this._head;

    if (index === 0) {
        // Special case for removing the head node.
        this._head = current.next;
    } else {
        // Track previous element
        var previous = null;
        var counter = 0;

        while (counter < index) {
            previous = current;
            current = current.next;
            counter++;
        }

        // Set previous node's next
        // to the node after the deleted node
        previous.next = current.next;
    }

    this._length--;
    return current.data;
}

// Let's create a Linked List and add 3 nodes
var list = new LinkedList();
list.push('Stacks');
list.push('Queues');
list.push('Arrays');
list.push('Sets');

console.log("Length before removal: " + list.size());

// Remove the 3rd element
var removed = list.remove(2);
console.log("removed: " + removed);

// Remove the head node
var removed = list.remove(0);
console.log("removed: " + removed);

console.log("Length after removal: " + list.size());

```



Console

⊗ Clear

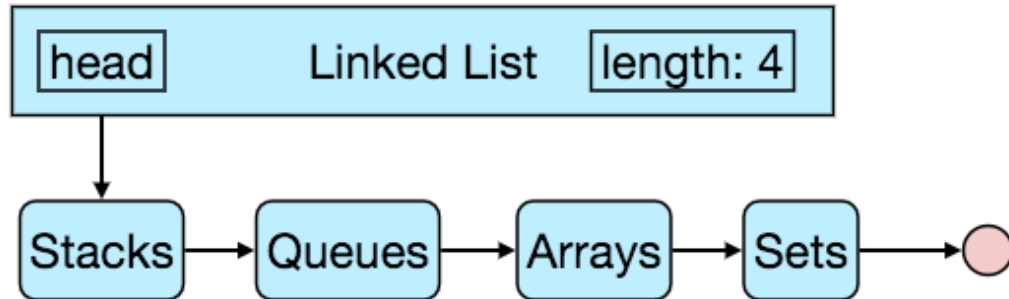
Length before removal: 4

removed: Arrays

removed: Stacks

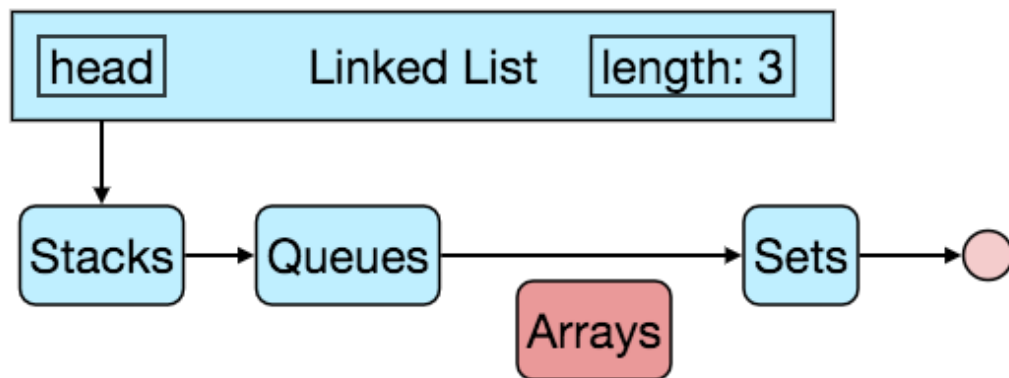
Length after removal: 2

Below visualization shows how nodes were removed in the code.



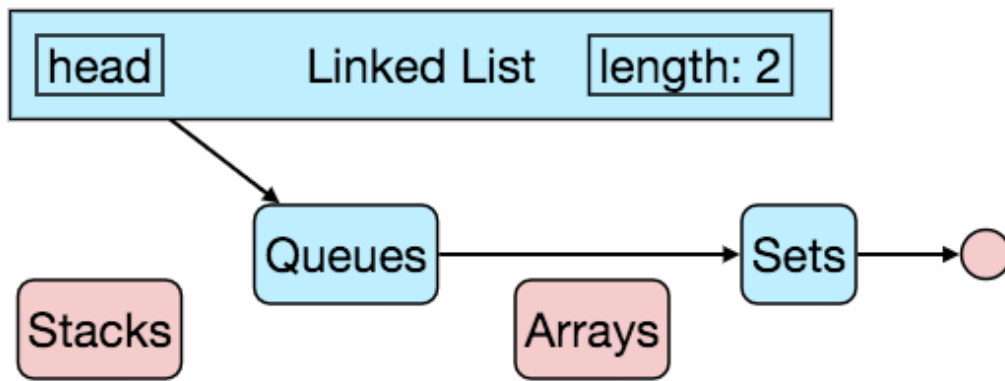
Linked List before any element is removed

1 of 3



Removing node at Index 2.
Node at Index 1 now points to Node at Index 3

2 of 3



Removing node at Index 0.
With current head node removed, head now points to the new head.

3 of 3



Let's go through the deletion code. Feel free to skip to Quiz #3.

We start at the head and traverse to the node to be deleted. If the node to be deleted is the head, then we just point the *_head* to the second node in the list (which could be null if it's a single node linked list).

If the node to be deleted is not the head, then we traverse to the node that is going to be deleted. While traversing to the node, we keep track of the previous node as well. Why? Because in a singly linked list, there is no way to access the previous node of a given node. Hence, we need to explicitly track the previous node as we traverse. Once we reach the desired node, we connect the *previous node's next* to *current node's next*. Think of it as if the current node was a middleman and now previous node is making a direct connection to the node after the current node (or metaphorically speaking, we are removing the middleman).

After setting the references, we then reduce the length and return the data in the current node.

Here's the code for deletion.



```
// Removes the element and returns the data
// in the node that was removed
LinkedList.prototype.remove = function(index) {

    // Ensure that the index is within bounds
    if (index < 0 || index >= this._length) {
        return null;
    }

    var current = this._head;

    if (index === 0) {
        // Special case for removing the head node.
        this._head = current.next;
    } else {
        // Track previous element
        var previous = null;
        var counter = 0;

        while (counter < index) {
            previous = current;
            current = current.next;
            counter++;
        }

        // Set previous node's next
        // to the node after the deleted node
        previous.next = current.next;
    }

    this._length--;
    return current.data;
}
```

Quiz #3 - Removing nodes from a linked list

1

Why do we track the previous node while removing

2

What should the previous.next point to if we just removed the last node?

Check Answers

Exercise

Let's do an exercise. We now want to implement the search in a linked list. Search function takes in a value and either returns the index of the node if the value exists or returns null if value doesn't exist in the linked list.

> *Run the following code. Implement search method so that the tests pass.*

JavaScript

HTML

CSS (SCSS)

```
// Returns the index of the node containig
// the data (assume unique data for simplicity)
LinkedList.prototype.search = function(data) {
  ////////// TODO //////////
  ////////// IMPLEMENT THIS //////////
  ////////// RUN AFTER IMPLEMENTING THIS //////////
  ////////// TESTS SHOULD PASS //////////

  return undefined;
}

// This is the function that runs test cases.
runEvaluation();
```



Console

Clear

*** There is some bug lurking there. See failed test cases ***

Here are the tests that ran:

here are the tests that fail:

Test case FAILED for list.search(100). Result: undefined. Expected: null
--

Test case FAILED for list.search(81). Result: undefined. Expected: 9
--

Test case FAILED for list.search(49). Result: undefined. Expected: 7
--

Test case FAILED for list.search(16). Result: undefined. Expected: 4
--

Summary

- Linked list comprises of a chain of nodes.
- Each node contains a data element and the reference to the next node in the chain.
- Linked lists are better for scenarios where you need to insert or remove elements from the middle.
- Unlike arrays, linked lists don't provide random access. Hence, look up and retrieval in linked lists are slower than an array.