What is a data structure?

A data structure is a way of organizing data that is stored in a computer so that it can be used efficiently.

A *Linked List*, that as its name says, is a linked list of nodes that are represents by a head that is the first node in the list and the tail that is the last one. Each node has a reference/pointer to the previous and the next node.

The linked list data structure have two types, the first one is single linked list, the nodes of this type have a pointer to the next one but not for their previous node.

In this article we are going to explore the *Doubly Linked List, that the nodes have a next and previous pointer (head has a next pointer but not previous and the tail node has a prev pointer but not a next one).*

Advantages:

Although a linked list is similar to an array, it is not restricted to a declared number of elements. Additionally, unlike an array which stores data contiguously in memory or on disk, a linked list can easily **insert or remove** elements without reallocation or reorganization of the entire structure because the data items need not be stored contiguously.

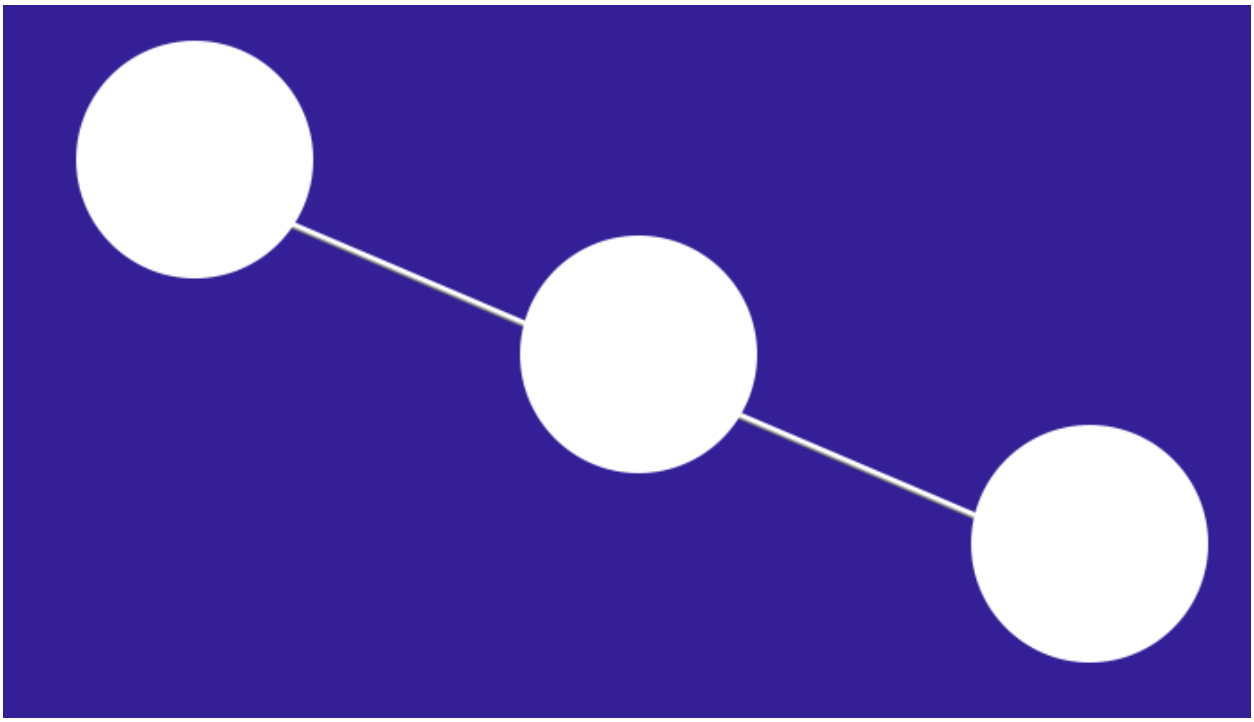
Linked List Drawbacks:

1) Random access is not allowed. We must access nodes sequentially starting from the first one. Therefore, we cannot do a

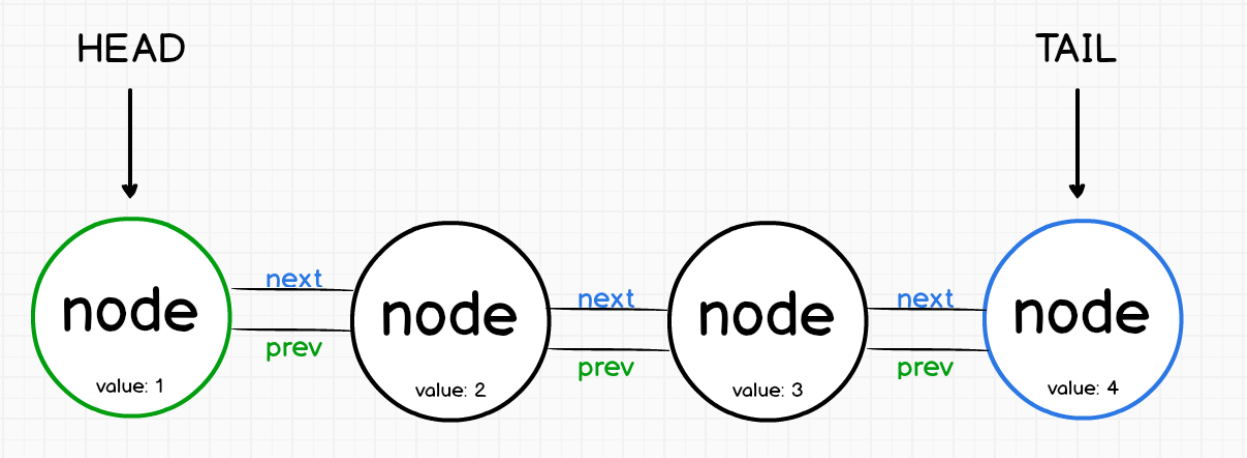
binary search on a linked list. **So for searching element is slow.**

2) Extra memory space for a link is required for each element of the list.

The **linked list** data structure is often used to implement other data structures.



In this article we are going to have an approach to the *linked list* data structure.



So we are going to create our two constructor functions:

```
function LinkedList() {
  this.head = null;
  this.tail = null;
}
function Node(value, next, prev) {
  this.value = value;
  this.next = next;
  this.prev = prev;
}
```

As you see we are represent the image sample in our constructor. Our *LinkedList function* have the `head` and the `tail`, and **why they are null?** Because at the beginning with don't have any node.

So, now we are going to create our *addToTail* method.

Creating head nodes

```
LinkedList.prototype.addToHead = function(value) {
  const newNode = new Node(value, this.head, null);
  if (this.head) this.head.prev = newNode;
  else this.tail = newNode;
```

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

As you see we created the method inside the LinkedList prototype, **why?** well this tecnique is useful because we are going to create many objects, and if we have not create our methods in the prototype we would be duplicating all the methods for each object which meant an expenditure in memory that could be harmful.

Let's review each line

`const newNode = new Node(value, this.head, null);` This is going to store in the variable newNode a new Node object. `value` is going to be the value that we pass in the addToHead method, `this.head` is null at first, so the `next`property is null, and the `prev` attribute is going to be null because we pass it in the third parameter.
`if (this.head) this.head.prev = newNode;` Alright, this line means that if exist a *head* node their prev value is going to be the newNode (that is the new head). if there is not a node, the actual node that we are creating is going to be the head and also the tail as we saw in the **THIRD** image sample.

So if we now create, for example, two nodes:

```
const list = new LinkedList();
list.addToHead(100);
list.addToHead(200);
console.log(list);
```

We will have this output:

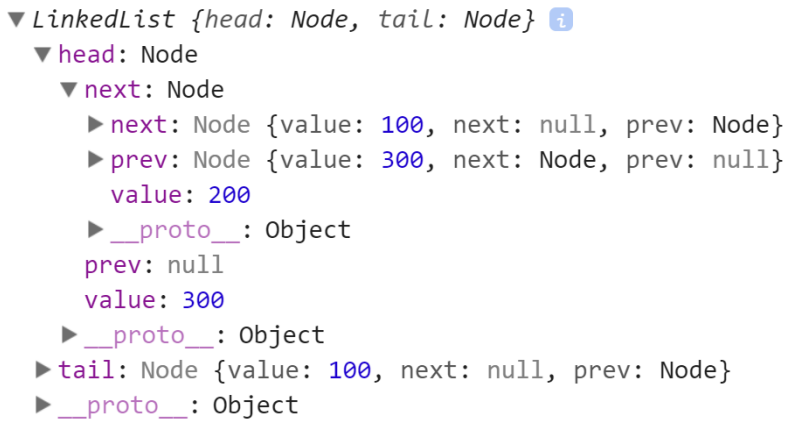
```
▼ LinkedList {head: Node, tail: Node}
  ▶ head: Node {value: 200, next: Node, prev: null}
  ▶ tail: Node {value: 100, next: null, prev: Node}
  ▶ __proto__: Object
```

The head node has a `value` of 200, the `next` property is the tail object (the next one in the list), and there is not a *prev* object because the *head* is the first one.

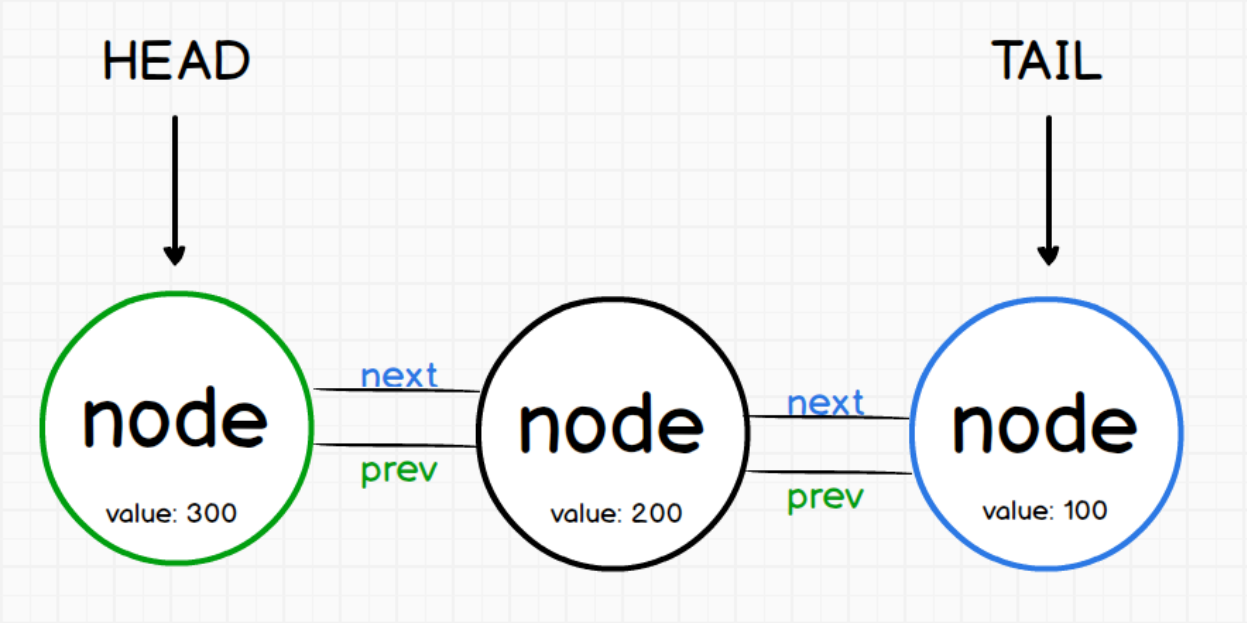
Now, imagine this:

```
const otherlist = new LinkedList();
otherlist.addToHead(100);
otherlist.addToHead(200);
otherlist.addToHead(300);
console.log(otherlist);
```

The output will be:



Or something like this:



So, if you want it to access to the middle node you can do this:

```
console.log(`Middle node value: ${otherlist.head.next.value}`);
```

Remember that the *addToHead* method add the node to the start, then the only thing that you need to do is to decompose the object in your console!

Try it out!

So, now we are going to create our *addToTail* method.

Creating tail nodes

Actually, this method is pretty much the same as we did in the addToHead example.

```
LinkedList.prototype.addToTail = function(value) {
  const newNode = new Node(value, null, this.tail);
  if (this.tail) this.tail.next = newNode;
  else this.head = newNode;
  this.tail = newNode;
}
```

Use the same logic that we did in the last example, the essence is similar, only that using the inverse logic.

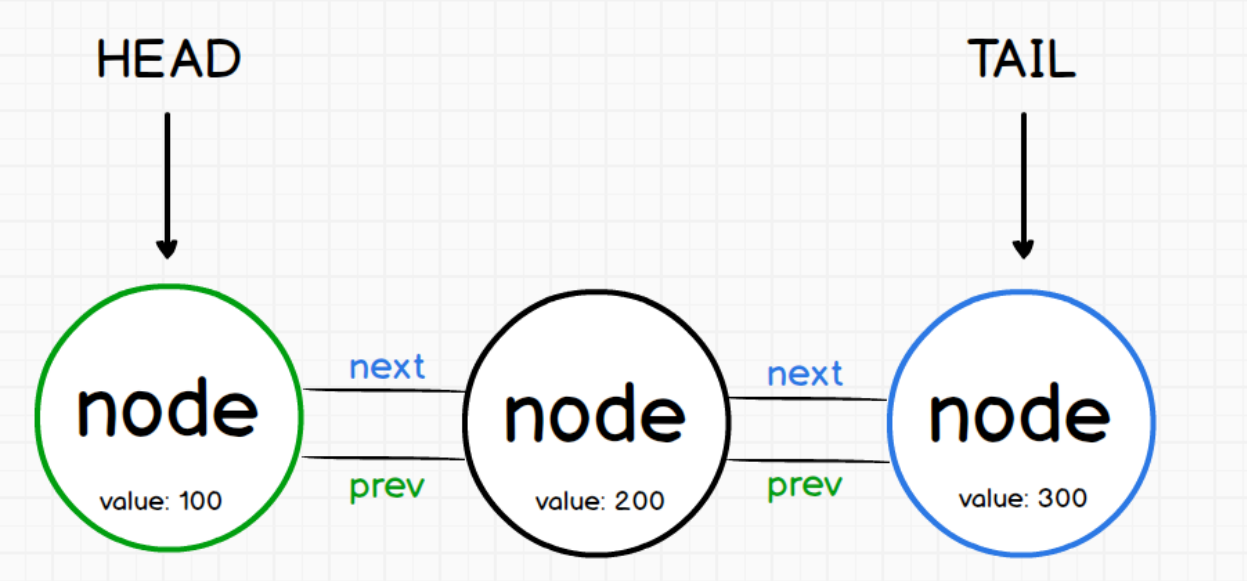
So now, if we do the same example that we did in the addToHead method:

```
const list = new LinkedList();
list.addToTail(100);
list.addToTail(200);
list.addToTail(300);
console.log(list);
```

Now the last added is going to be the Tail (the last one) unlike the other method that the last writing was added as the first node (Head).

```
▼ LinkedList {head: Node, tail: Node}
  ▶ head: Node {value: 100, next: Node, prev: null}
  ▶ tail: Node {value: 300, next: null, prev: Node}
  ▶ __proto__: Object
```

Or like this:



Testing both methods:

```
const list = new LinkedList();
list.addToHead(1);
list.addToTail(2);
console.log(list);
```

```
▼ LinkedList {head: Node, tail: Node}
  ▼ head: Node
    ▼ next: Node
        next: null
      ▶ prev: Node {value: 1, next: Node, prev: null}
        value: 2
      ▶ __proto__: Object
      prev: null
      value: 1
    ▶ __proto__: Object
  ▶ tail: Node {value: 2, next: null, prev: Node}
  ▶ __proto__: Object
```

Removing Nodes

Imagine that we have this nodes:

```
const list = new LinkedList();
list.addToHead(200);
list.addToHead(100); // remember this is the head now!
list.addToTail(300);
console.log(list);
```

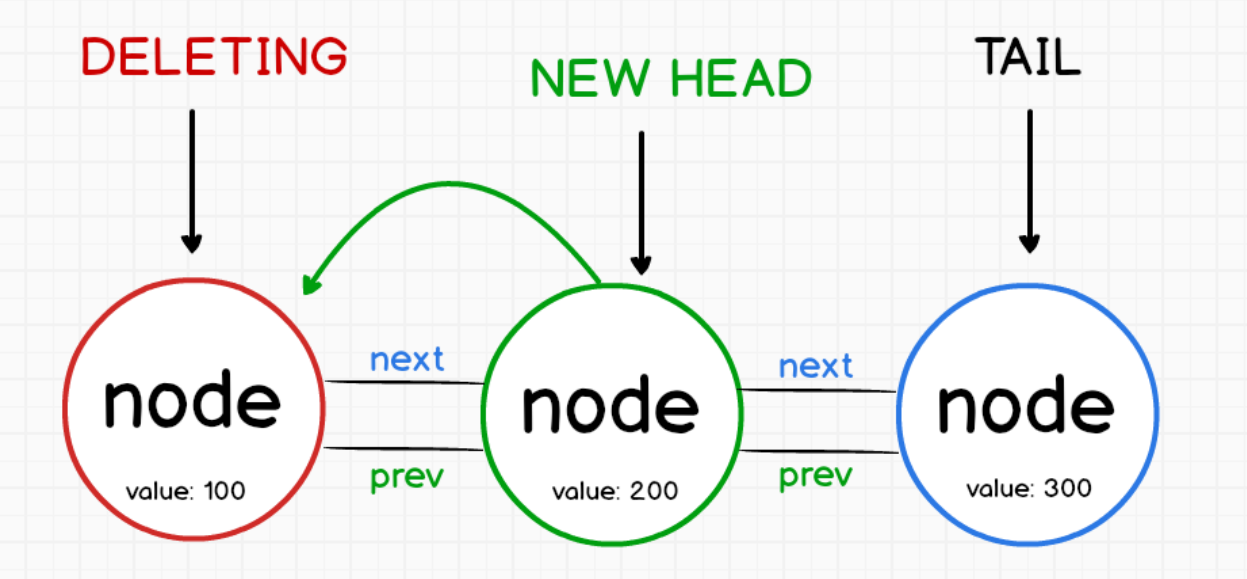
The method for delete head nodes:

```
LinkedList.prototype.removeHead = function() {
  if (!this.head) return null;
  let value = this.head.value;
  this.head = this.head.next;

  if (this.head) this.head.prev = null;
  else this.tail = null;

  return value;
}
```

Les't see, the first line is going to validate if there exist any head, if not return null. Then we save the *value* of the *head node* and we set the new head node with the this line: `this.head = this.head.next;` So at this point we have this:



And in the last lines of code we just re-set the *prev* to null, because the new head mustn't have a prev value (because is the first node).

An return the remove value.

The method for delete tail nodes:

```
LinkedList.prototype.removeTail = function() {
  if (!this.tail) return null;
  let value = this.tail.value;
  this.tail = this.tail.prev;

  if (this.tail) this.tail.next = null;
  else this.head = null;

  return value;
}
```

Applies the same logic for this method, because it is the same but with the opposite effect.

Searching nodes:

```
LinkedList.prototype.search = function(searchValue) {
  let currentNode = this.head;

  while(currentNode) {
    if (currentNode.value === searchValue) return currentNode;
    currentNode = currentNode.next;
  }
  return null;
}
```

So here, we save in the *currentNode* variable the value of `this.head` , then *while* the currentNode are not *undefined* we compare if exist a *node* with the *value* that we are passing, if not we return *null*.

So, if we have this:

```
const list = new LinkedList();
list.addToHead(1);
list.addToTail(2);
console.log(list.search(1)); // true
console.log(list.search(2)); // true
console.log(list.search(3000)); // false
```