tuts+

CODE > JAVASCRIPT

Data Structures With JavaScript: Singly-Linked List and Doubly-Linked List

by Cho S. Kim Sep 17, 2015

Read Time: 16 mins Languages: English

This post is part of a series called Data Structures in JavaScript.

Data Structures With JavaScript: Stack and Queue

Data Structures With JavaScript: Tree

What You'll Be Creating

Two of the most commonly taught data structures in computer science are the singly-linked list and doubly-linked list.

When I was taught these data structures, I asked my peers for analogies to conceptualize them. What I heard were several examples, such as a list of groceries and a train. These analogies, as I eventually learned, were inaccurate. A grocery list was more analogous a queue; a train was more analogous to an array.

As more time passed, I eventually discovered an analogy that accurately described a singly-linked list and a doubly-linked list: a scavenger hunt. If you're curious about the relationship between a scavenger hunt and a linked list, then read below for the answer!

A Singly-Linked List

In computer science, a singly-linked list is a data structure that holds a sequence of linked nodes. Each node, in turn, contains data and a pointer, which can point to another node.

Nodes of a singly-linked list are very similar to steps in a scavenger hunt. Each step contains a message (e.g. "You've reached France") and pointers to the next step (e.g. "Visit these latitude and longitude coordinates"). When we start sequencing these individual steps to form a sequence of steps, we are creating a scavenger hunt.

Now that we have a conceptual model of a singly-linked list, let's explore the operations of a singly-linked list.

Operations of a Singly-Linked List

Since a singly-linked list contains nodes, which can be a separate constructor from a singly-linked list, we outline the operations of both constructors: `Node` and `SinglyList`.

Node

- `data` stores a value.
- `next` points to the next node in the list.

`next` points to the next node in the list.

SinglyList

- `_length` retrieves the number of nodes in a list.
- `head` assigns a node as the head of a list.
- `add(value)` adds a node to a list.
- `searchNodeAt(position)` searches for a node at n-position in our list.
- `remove(position)` removes a node from a list.

Advertisement

Implementation of a Singly-Linked List

For our implementation, we will first define a constructor named `Node` and then a constructor named `SinglyList`.

Each instance of `Node` needs the ability to store data and the ability to point to another node. To add this functionality, we will create two properties: `data` and `next`, respectively.

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

Next, we need to define `SinglyList`:

```
function SinglyList() {
        this._length = 0;
        this.head = null;
}
```

Each instance of `SinglyList` will have two properties: `_length` and `head`. The former is assigned the number of nodes in a list; the latter points to the head of the list—the node at the front of the list. Since every new instance of `SinglyList` does not contain a node, the default value of `head` is `null` and the default value of `_length` is `0`.

Methods of a Singly-Linked List

We need to define methods that can add, search, and remove a node from a list. Let's start with adding a node.

1 of 3: `add(value)`

Awesome, let's now implement the functionality to add nodes to a list.

```
SinglyList.prototype.add = function(value) {
        var node = new Node(value),
                currentNode = this.head;

        // 1st use-case: an empty list
        if (!currentNode) {
                this.head = node;
                this._length++;

                return node;
        }

        // 2nd use-case: a non-empty list
        while (currentNode.next) {
                currentNode = currentNode.next;
        }

        currentNode.next = node;

        this._length++;

        return node;
};
```

Adding a node to our list involves many steps. Let us start from the beginning of our method. We use the argument of `add(value)` to create a new instance of a `Node`, which is assigned to a variable named `node`. We

also declare a variable named `currentNode` and initialize it to the `_head` of our list. If there are no nodes in the list, then the value of `head` is `null`.

After this point in our code, we handle two use cases.

The first use case considers adding a node to an empty list. If `head` does not point to a node, then assign `node` as the head of our list, increment the length of our list by one, and return `node`.

The second use case considers adding a node to a non-empty list. We enter the `while` loop, and during each iteration, we evaluate if `currentNode.next` points to another node. (During the first iteration, `currentNode` is always pointing to the head of a list.)

If the answer is no, we assign `node` to `currentNode.next` and return `node`.

If the answer is yes, we enter the body of the `while` loop. Inside the body, we reassign `currentNode` to `currentNode.next`. This process is repeated until `currentNode.next` no longer points to another node. In other words, `currentNode` points to the last node of our list.

The `while` loop breaks. Finally, we assign `node` to `currentNode.next`, we increment `_length` by one, and then we return `node`.

2 of 3: `searchNodeAt(position)`

We can now add nodes to our list, but we cannot search for nodes at specific positions in our list. Let's add this functionality and create a method named `searchNodeAt(position)`, which accepts an argument named `position`. The argument is expected to be an integer that indicates a node at n-position in our list.

```
SinglyList.prototype.searchNodeAt = function(position) {
        var currentNode = this.head,
                length = this._length,
                count = 1,

                message = {failure: 'Failure: non-existent node in this list.'};

        // 1st use-case: an invalid position
        if (length === 0 || position < 1 || position > length) {
                throw new Error(message.failure);
        }

        // 2nd use-case: a valid position
        while (count < position) {
                currentNode = currentNode.next;
                count++;
        }

        return currentNode;
};
```

The `if` statement checks for the first use case: an invalid position is passed as an argument.

If the index passed to `searchNodeAt(position)` is valid, then we reach the second use case—the `while` loop. During each iteration of the `while` loop, `currentNode`—which first points to `head`—is reassigned to the next node in the list. This iteration continues until count is equal to position.

When the loop finally breaks, `currentNode` is returned.

3 of 3: `remove(position)`

The final method we will create is named `remove(position)`.

```
SinglyList.prototype.remove = function(position) {
        var currentNode = this.head,
                length = this._length,
                count = 0,
                message = {failure: 'Failure: non-existent node in this list.'},
                beforeNodeToDelete = null,
                nodeToDelete = null,
                deletedNode = null;

        // 1st use-case: an invalid position
        if (position < 0 || position > length) {
                throw new Error(message.failure);
        }

        // 2nd use-case: the first node is removed
        if (position === 1) {
                this.head = currentNode.next;
```

```
        this.head = currentNode.next;
        deletedNode = currentNode;
        currentNode = null;
        this._length--;

        return deletedNode;
    }

    // 3rd use-case: any other node is removed
    while (count < position) {
        beforeNodeToDelete = currentNode;
        nodeToDelete = currentNode.next;
        count++;
    }

    beforeNodeToDelete.next = nodeToDelete.next;
    deletedNode = nodeToDelete;
    nodeToDelete = null;
    this._length--;

    return deletedNode;
};
```

Our implementation of `remove(position)` involves three use cases:

1. An invalid position is passed as an argument.
2. A position of one (`head` of a list) is passed as an argument.
3. An existent position (not the first position) is passed as an argument.

The first and second use cases are the simplest to handle. In regards to the first scenario, if the list is empty or a non-existent position is passed, an error is thrown.

The second use case handles the removal of the first node in the list, which is also `head`. If this is the case, then the following logic occurs:

1. `head` is reassigned to `currentNode.next`.
2. `deletedNode` points to `currentNode`.
3. `currentNode` is reassigned to `null`.
4. Decrement `_length` of our list by one.
5. `deletedNode` is returned.

The third scenario is the hardest to understand. The complexity stems from the necessity of tracking two nodes during each iteration of a `while` loop. During each iteration of our loop, we track both the node before the node to be deleted *and* the node to be deleted. When our loop the node to be deleted *and* the node to be deleted. When our loop eventually reaches the node at the position we want to remove, the loop terminates.

At this point, we hold references to three nodes: `beforeNodeToDelete`, `nodeToDelete`, and `deletedNode`. Prior to deleting `nodeToDelete`, we must assign its value of `next` to the next value of `beforeNodeToDelete`. If the purpose of this step seems unclear, remind yourself that we have a list of linked nodes; just removing a node breaks the link from the first node of the list to the last node of the list.

Next, we assign `deletedNode` to `nodeToDelete`. Then we set the value of `nodeToDelete` to `null`, decrement the length of the list by one, and return `deletedNode`.

Complete Implementation of a Singly-Linked List

The complete implementation of our list is here:

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

function SinglyList() {
        this._length = 0;
        this.head = null;
}

SinglyList.prototype.add = function(value) {
        var node = new Node(value),
                currentNode = this.head;

        // 1st use-case: an empty list
        if (!currentNode) {
                this.head = node;
                this._length++;

                return node;
        }

        // 2nd use-case: a non-empty list
        while (currentNode.next) {
                currentNode = currentNode.next;
        }

        currentNode.next = node;

        this._length++;

        return node;
}
```

```
};

SinglyList.prototype.searchNodeAt = function(position) {
    var currentNode = this.head,
        length = this._length,
        count = 1,
        message = {failure: 'Failure: non-existent node in this list.'};

    // 1st use-case: an invalid position
    if (length === 0 || position < 1 || position > length) {
        throw new Error(message.failure);
    }

    // 2nd use-case: a valid position
    while (count < position) {
        currentNode = currentNode.next;
        count++;
    }

    return currentNode;
};

SinglyList.prototype.remove = function(position) {
    var currentNode = this.head,
        length = this._length,
        count = 0,
        message = {failure: 'Failure: non-existent node in this list.'},
        beforeNodeToDelete = null,
        nodeToDelete = null,
        deletedNode = null;

    // 1st use-case: an invalid position
    if (position < 0 || position > length) {
        throw new Error(message.failure);
    }

    // 2nd use-case: the first node is removed
    if (position === 1) {
        this.head = currentNode.next;
        deletedNode = currentNode;
        currentNode = null;
        this._length--;

        return deletedNode;
    }

    // 3rd use-case: any other node is removed
    while (count < position) {
        beforeNodeToDelete = currentNode;
        nodeToDelete = currentNode.next;
        count++;
    }

    beforeNodeToDelete.next = nodeToDelete.next;
    deletedNode = nodeToDelete;
    nodeToDelete = null;
    this._length--;

    return deletedNode;
};
```

From Singly to Doubly

From Singly to Doubly

Awesome, our implementation of a singly-linked list is complete. We can now use a data structure that adds, removes, and searches nodes in a list that occupy non-contiguous space in memory.

However, at this moment, all of our operations begin from the beginning of a list and run to the end of a list. In other words, they are uni-directional.

There may be instances where we want our operations to be bi-directional. If you considered this use case, then you have just described a doubly-linked list.

A Doubly-Linked List

A doubly-linked list takes all the functionality of a singly-linked list and extends it for bi-directional movement in a list. We can traverse, in other words, a linked list from the first node in the list to the last node in the list; and we can traverse from the last node in the list to the first node in the list.

In this section, we will maintain our focus primarily on the differences between a doubly-linked list and a singly-linked list.

Operations of a Doubly-Linked List

Our list will include two constructors: `Node` and `DoublyList`. Let us outline their operations.

Node

- `data` stores a value.
- `next` points to the next node in the list.
- `previous` points to the previous node in the list.

- `previous` points to the previous node in the list.

DoublyList

- `_length` retrieves the number of nodes in a list.
- `head` assigns a node as the head of a list.
- `tail` assigns a node as the tail of a list.
- `add(value)` adds a node to a list.
- `searchNodeAt(position)` searches for a node at n-position in our list.
- `remove(position)` removes a node from a list.

Advertisement

Implementation of a Doubly-Linked List

Let write some code!

For our implementation, we will create a constructor named `Node`:

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

To create the bi-directional movement of a doubly-linked list, we need properties that point in both directions of a list. These properties have been named `previous` and `next`.

Next, we need to implement a `DoublyList` and add three properties: `_length`, `head`, and `tail`. Unlike a singly-linked list, a doubly-linked list has a reference to both the beginning of a list and the end of a list. Since every instance of a `DoublyList` is instantiated without nodes, the default values of `head` and `tail` are set to `null`.

```
function DoublyList() {
        this._length = 0;
        this.head = null;
        this.tail = null;
}
```

Methods of a Doubly-Linked List

We will now explore the following methods: `add(value)`, `remove(position)`, and `searchNodeAt(position)`. All of these methods were used for a singly-linked list; however, they must be rewritten for bi-directional movement.

1 of 3: `add(value)`

```
DoublyList.prototype.add = function(value) {
        var node = new Node(value);

        if (this._length) {
                this.tail.next = node;
                node.previous = this.tail;
                this.tail = node;
        } else {
                this.head = node;
                this.tail = node;
        }

        this._length++;

        return node;
};
```

In this method, we have two scenarios. First, if a list is empty, then assign to its `head` and `tail` the node being added. Second, if the list contains nodes, then find the tail of the list and assign to `tail.next` the node being added; likewise, we need to configure the new tail for bi-directional movement. We need to set, in other words, `tail.previous` to the original tail.

2 of 3: `searchNodeAt(position)`

The implementation of `searchNodeAt(position)` is identical to a singly-linked list. If you forgot how to implement it, I've included it below:

```
DoublyList.prototype.searchNodeAt = function(position) {
        var currentNode = this.head,
                length = this._length,
                count = 1,
                message = {failure: 'Failure: non-existent node in this list.'};

        // 1st use-case: an invalid position
        if (length === 0 || position < 1 || position > length) {
                throw new Error(message.failure);
        }

        // 2nd use-case: a valid position
        while (count < position) {
                currentNode = currentNode.next;
                count++;
        }

        return currentNode;
};
```

---

3 of 3: `remove(position)`

This method will be the most challenging to understand. I'll display the code and then explain it.

```
DoublyList.prototype.remove = function(position) {
        var currentNode = this.head,
                length = this._length,
                count = 1,
                message = {failure: 'Failure: non-existent node in this list.'},
                beforeNodeToDelete = null,
                nodeToDelete = null,
                deletedNode = null;

        // 1st use-case: an invalid position
        if (length === 0 || position < 1 || position > length) {
                throw new Error(message.failure);
        }

        // 2nd use-case: the first node is removed
        if (position === 1) {
                this.head = currentNode.next;

                // 2nd use-case: there is a second node
                if (!this.head) {
```

```
        if (!this.head) {
            this.head.previous = null;
        // 2nd use-case: there is no second node
        } else {
            this.tail = null;
        }

    // 3rd use-case: the last node is removed
    } else if (position === this._length) {
        this.tail = this.tail.previous;
        this.tail.next = null;
    // 4th use-case: a middle node is removed
    } else {
        while (count < position) {
            currentNode = currentNode.next;
            count++;
        }

        beforeNodeToDelete = currentNode.previous;
        nodeToDelete = currentNode;
        afterNodeToDelete = currentNode.next;

        beforeNodeToDelete.next = afterNodeToDelete;
        afterNodeToDelete.previous = beforeNodeToDelete;
        deletedNode = nodeToDelete;
        nodeToDelete = null;
    }

    this._length--;

    return message.success;
};
```

`remove(position)` handles four use cases:

1. The position being passed as an argument of `remove(position)` is non-existent. In this case, we throw an error.
2. The position being passed as an argument of `remove(position)` is the first node (`head`) of a list. If this is the case, we will assign `deletedNode` to `head` and then reassign `head` to the next node in the list. At this moment, we must consider if our list has more than one node. If the answer is no, `head` will be assigned to `null` and we will enter the `if` part of our `if-else` statement. In the body of `if`, we must also set `tail` to `null`—in other words, we return to the original state of an empty doubly-linked list. If we are removing the first node in a list and we have more than one node in our list, we enter the `else` section of our `if-else` statement. In this case, we must correctly set the `previous` property of `head` to `null`—there are no nodes before the head of a list.

3. The position being passed as an argument of `remove(position)` is the tail of a list. First, `deletedNode` is assigned to `tail`. Second, `tail` is reassigned to the node previous to the tail. Third, the new tail has no node after it and needs its value of `next` to be set to `null`.
4. A lot is happening here, so I will focus on the logic more than each line of the code. We break our `while` loop once `currentNode` is pointing to the node at the position being passed as an argument to `remove(position)`. At this moment, we reassign the value of `beforeNodeToDelete.next` to the node after `nodeToDelete` and, conversely, we reassign the value of `afterNodeToDelete.previous` to the node before `nodeToDelete`. In other words, we remove pointers to the removed node and reassign them to the correct nodes. Next, we assign `deletedNode` to `nodeToDelete`. Finally, we assign `nodeToDelete` to `null`.

Finally, we decrement the length of our list and return `deletedNode`.

Complete Implementation of a Doubly-Linked List

Here's the entire implementation:

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

function DoublyList() {
        this._length = 0;
        this.head = null;
        this.tail = null;
}

DoublyList.prototype.add = function(value) {
        var node = new Node(value);

        if (this._length) {
                this.tail.next = node;
                node.previous = this.tail;
                this.tail = node;
        } else {
                this.head = node;
                this.tail = node;
        }

        this._length++;

        return node;
```

```
        return node;
};

DoublyList.prototype.searchNodeAt = function(position) {
        var currentNode = this.head,
                length = this._length,
                count = 1,
                message = {failure: 'Failure: non-existent node in this list.'};

        // 1st use-case: an invalid position
        if (length === 0 || position < 1 || position > length) {
                throw new Error(message.failure);
        }

        // 2nd use-case: a valid position
        while (count < position) {
                currentNode = currentNode.next;
                count++;
        }

        return currentNode;
};

DoublyList.prototype.remove = function(position) {
        var currentNode = this.head,
                length = this._length,
                count = 1,
                message = {failure: 'Failure: non-existent node in this list.'},
                beforeNodeToDelete = null,
                nodeToDelete = null,
                deletedNode = null;

        // 1st use-case: an invalid position
        if (length === 0 || position < 1 || position > length) {
                throw new Error(message.failure);
        }

        // 2nd use-case: the first node is removed
        if (position === 1) {
                this.head = currentNode.next;

                // 2nd use-case: there is a second node
                if (!this.head) {
                        this.head.previous = null;
                // 2nd use-case: there is no second node
                } else {
                        this.tail = null;
                }

        // 3rd use-case: the last node is removed
        } else if (position === this._length) {
                this.tail = this.tail.previous;
                this.tail.next = null;
        // 4th use-case: a middle node is removed
        } else {
                while (count < position) {
                        currentNode = currentNode.next;
                        count++;
                }

                beforeNodeToDelete = currentNode.previous;
                nodeToDelete = currentNode;
                afterNodeToDelete = currentNode.next;

                beforeNodeToDelete.next = afterNodeToDelete;
                afterNodeToDelete.previous = beforeNodeToDelete;
```

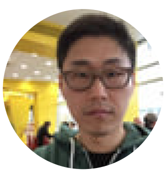
```
            afterNodeToDelete.previous = beforeNodeToDelete;
            deletedNode = nodeToDelete;
            nodeToDelete = null;
        }

        this._length--;

        return message.success;
};
```

Conclusion

We have covered a lot of information in this article. If any of it appears confusing, read it again, and experiment with the code. When it eventually makes sense to you, feel proud. You have just uncovered the mysteries of both a singly-linked list and a doubly-linked list. You can add these data structures to your coding tool-belt!

Advertisement

JavaScript Programming Fundamentals



Cho S. Kim
San Francisco, CA

Cho is a full-stack web-application developer. He dislikes mean people but likes the MEAN stack (MongoDB, ExpressJS, AngularJS, Node.js). During a typical week, he'll be coding in JavaScript, writing about JavaScript, or watching movies NOT about JavaScript.

choskim

FEED LIKE FOLLOW

Weekly email summary

Subscribe below and we'll send you a weekly email summary of all new Code tutorials. Never miss out on learning about the next big thing.

Update me weekly

View on GitHub

Advertisement

QUICK LINKS - Explore popular categories

ENVATO TUTS+ +

JOIN OUR COMMUNITY +

HELP +

tuts+

30,415 Tutorials 1,316 Courses 49,884 Translations

Certified B Corporation

Envato Envato Elements Envato Market Placeit by Envato Milkshake All products Careers Sitemap

© 2021 Envato Pty Ltd. Trademarks and brands are the property of their respective owners.