Computer science in JavaScript: Linked list

Posted at April 13, 2009 09:00 am by Nicholas C. Zakas

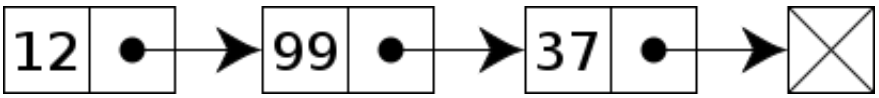
Tags: Computer Science, JavaScript, Linked List, Programming

When I started writing the first edition of Professional JavaScript, my working title was JavaScript for Web Applications and it featured a lot of content that didn't make the final cut. I actually have several chapters worth of content just sitting around on my computer. Several of these chapters discuss implementing common computer science patterns and algorithms in JavaScript. At the time, I thought this would make a good addition to the book, but ultimately ended up holding them back as they didn't fit the final vision for the book. Instead of letting that content sit on my computer, I've decided to start sharing on this blog.

One of the first data structures you learn in computer science is the linked list. As a quick refresher, here's the Wikipedia description of a linked list:

> It consists of a sequence of nodes, each containing arbitrary data fields and one or two references ("links") pointing to the next and/or previous nodes. The principal benefit of a linked list over a conventional array is that the order of the linked items may be different from the order that the data items are stored in memory or on disk, allowing the list of items to be traversed in a different order. A linked list is a self-referential datatype because it contains a pointer or link to another datum of the same type. Linked lists permit insertion and removal of nodes at any point in the list in constant time, but do not allow random access.

Linked lists are often used in computer science programs to help introduce the concept of pointers. The list itself is just a pointer to the head node, which in turn points to the next node, and so on. Each node consists of two fields: a `data` field containing the value for that location in the list and a `next` field containing a pointer to the next node in the list (or an empty pointer if it's the last item).



To begin a JavaScript implementation, start with creating a single node. This can be done most easily by using an object literal:

```
var firstNode = {
    data: 12,
    next: null
};
```

When you want to create a list, create a new node and assign it to this node's `next` property:

```
//attach to first node to create list
firstNode.next = {
    data: 99,
    next: null
};
```

Once you have a list, you can traverse by following the `next` property on each node to get to a specific point in the list. Of course, doing all of this by hand is annoying and error prone, so it's better to create a custom type. Here's the start:

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

The `LinkedList` constructor creates an object with “private” properties: `_length`, which contains the number of items in the list, and `_head`, which points to the first item in the list. Initially, `_head` is set to `null` because the list is empty.

Adding an item into a linked list requires walking the structure to find the correct location, creating a new node, and inserting it in place. The one special case is when the list is empty, in which case you simply create a new node and assign it to `_head`:

```
LinkedList.prototype = {

    add: function (data){

        //create a new node, place data in
        var node = {
                data: data,
                next: null
            },

            //used to traverse the structure
            current;

        //special case: no items in the list yet
        if (this._head === null){
            this._head = node;
        } else {
            current = this._head;

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

            current.next = node;
        }

        //don't forget to update the count
        this._length++;

    },

    //more methods here
};
```

The most complicated part of this method is traversing an already-existing list to find the correct spot to insert the new node. Traditional algorithms use two pointers, a `current` that points to the item being inspected and a `previous` that points to the node before `current`. When `current` is `null`, that means `previous` is pointing to the last item in the list. I’ve recreated this algorithm in JavaScript though there are several other (arguably better) alternatives for tradition’s sake.

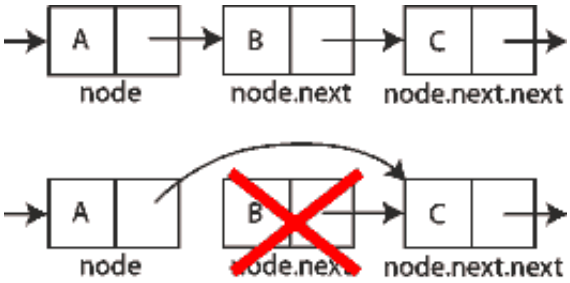
Retrieving a value from the list involves the same type of traversal:

```
LinkedList.prototype = {
    //more methods

    item: function(index){

        //check for out-of-bounds values
        if (index > -1 && index < this._length){
            var current = this._head,
                i = 0;

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

```
            return current.data;
        } else {
            return null;
        }
    },

    //more methods here
};
```

The `item()` method checks to ensure that the index being specified is within a valid range before traversing the list. The `while` loop is used to figure out the correct place to stop in the list to find the data being requested.

Removing a node from a linked list is a little bit tricky. You need to find the node to remove then set the previous node's `next` property to appropriate next node. This "skipping over" of the appropriate node results in it's removal from the list.



The typical implementation of linked list node removal is to have two pointers, a `current` pointer that indicates the node being inspected and a `previous` pointer that points to the node just prior to `current`. When `current` is the node to remove, then `previous.next` must be set to `current.next` to execute the removal. The code:

```
LinkedList.prototype = {
    //more methods

    remove: function(index){

        //check for out-of-bounds values
        if (index > -1 && index < this._length){

            var current = this._head,
                previous,
                i = 0;

            //special case: removing first item
            if (index === 0){
                this._head = current.next;
            } else {

                //find the right location
                while(i++ < index){
                    previous = current;
                    current = current.next;
                }

                //skip over the item to remove
                previous.next = current.next;
            }

            //decrement the length
            this._length--;

            //return the value
            return current.data;

        } else {
            return null;
        }

    },
```

```
    //more methods here
};
```

Note the one special case is in the removal of the first item. In that case, you need to set `_head` equal to `_head.next`, moving the pointer for the start of the list to next item.

Once complete, you can use the linked list implementation like this:

```
var list = new LinkedList();
list.add("red");
list.add("orange");
list.add("yellow");

alert(list.item(1));    //"orange"

list.remove(1);

alert(list.item(1));    //"yellow"
```

This basic implementation of a linked list can be rounded out with a `size()` method to return the length of the list and a `toArray()` method to convert into a regular array. The full source code is available on GitHub at my Computer Science in JavaScript project. I'll be updating the project with each blog post and hopefully build up a nice collection of implementations for reference. Just to be clear, I'm not advocating using this in production code; the native `Array` object serves all of our needs quite well. This is purely an academic exercise and should be treated as such.

Disclaimer: Any viewpoints and opinions expressed in this article are those of Nicholas C. Zakas and do not, in any way, reflect those of my employer, my colleagues, Wrox Publishing, O'Reilly Publishing, or anyone else. I speak only for myself, not for them.

Both comments and pings are currently closed.

Related Posts

- Computer science in JavaScript: Doubly-linked lists
- Computer science in JavaScript: Binary search tree, Part 2
- Computer science in JavaScript: Binary search tree, Part 1
- Computer science in JavaScript: Quicksort
- Computer science in JavaScript: Merge sort

Further reading

- 

- 

- 

16 Comments

Thanks for posting this. I'm very interested in design patterns and how they can be practically implemented in JavaScript. Please keep these posts coming.

So is this an example of what is happening underneath the hood when using the Array object?

Benjamin Toll on April 16th, 2009 at 1:37 am

[...] up in the "remember that data structures weeder Comp Sci class?" moment is the linked list: It consists of a sequence of nodes, each containing arbitrary data fields and one or two [...]

Ajaxian » Giving you that CompSci 101 feeling with JavaScript on April 16th, 2009 at 7:33 am

[...] up in the "remember that data structures weeder Comp Sci class?" moment is the linked list: It consists of a sequence of nodes, each containing arbitrary data fields and one or two [...]

Giving you that CompSci 101 feeling with JavaScript | Guilda Blog on April 16th, 2009 at 8:35 am

Are you interested in contributions in other areas of CS? Eg, I have a Javascript Y Combinator implementation that I'd be happy to pass along.

anders pearson on April 16th, 2009 at 10:20 am

In **LinkedList.add()** you wrote this:

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

current.next = node;
```

But `if ( !current ) then current.next == invalid, not? Maybe while ( current.next ) { ...`

FARKAS MÃ¡tÃ© on April 16th, 2009 at 1:23 pm

Okay, I can see it is good within your js file: http://github.com/nzakas/computer-science-in-javascript/blob/45fd079352a2256679357a4cac423d5efd242609/data-structures/linked-list.js

FARKAS MÃ¡tÃ© on April 16th, 2009 at 1:27 pm

Hi Nicholas

I wrote some non-recursive traversal methods some time ago. Falls in the CS category and you can include them if you like. Have a look at http://www.jslab.dk/articles/non.recursive.preorder.traversal for more info.

Dok on April 16th, 2009 at 2:59 pm

@Benjamin – Linked lists are one possible implementation for arrays. I've not dug through the various JavaScript engines to figure out what they're actually using, though it appears that V8's Array implementation is actually written in JavaScript.

@FARKAS – thanks for pointing that out. I've fixed the problem (copy-paste error!).

@Dok and Anders – I'll take a look at your implementations. I'm still digging though all of the material on my computer, but as soon as I get through that, I'll start looking at contributions.

Nicholas C. Zakas on April 16th, 2009 at 11:49 pm

This reminds me the lessons of data structure in school

å¸ƒé‡Œæ–¯ç on April 17th, 2009 at 8:02 am

[...] my last post, I discussed creating a linked list in JavaScript. This basic data structure is frequently used in [...]

Computer science in JavaScript: Doubly-linked lists | NCZOnline on April 21st, 2009 at 9:01 am

I've never seen a linked list implementation with "length" stored.

adam on April 22nd, 2009 at 6:48 pm

@adam – Now you have.

Nicholas C. Zakas on April 23rd, 2009 at 11:35 pm

Some things of note:

You should keep track of your linked list's tail. This would greatly simplify the add() method and would be considerably more efficient as you could add elements directly without traversing the entire list.

You may want to keep "dummy" head and tail nodes for the list. This would enable you to prevent checking for the special cases where the head or tail is null, reducing some checks and simplifying your code.

You probably shouldn't base your remove() and item() methods on indexes. While they're reasonable to have as convenience functions, it may be better to implement the primary remove() method by having it take an object so it won't require the user to keep track of indexes. For iterating the list, you may want to consider an iterator which is ideal for non-index-based structures (and this is where you will generally be retrieving the elements).

Also, linked lists really would not be good implementations for array. The advantage of arrays is that you can access elements directly by index in O(1) time; linked lists require O(n). Arrays are usually implemented as chunks of contiguous bytes so that an index at position n can be instantly retrieved with n*(element size). This would, of course, make arrays inefficient at adding/removing elements since the entire array would have to be shifted to accommodate such changes. This is where linked lists shine–they're great structures if you're going to be manipulating the list frequently. Since array and linked lists are so different in terms of use, it wouldn't make much sense to use one as the implementation for another.

Regards,
Brian

Brian on May 6th, 2009 at 1:57 pm

@Brian – Thanks for the feedback. My goal isn't to create the best, most-performant linked list implementation, just one that's practical and makes sense for study. You'll note that in my post on doubly-linked lists, I keep track of the tail as well as the head.

Nicholas C. Zakas on May 6th, 2009 at 11:29 pm

Hi Nicholas,

I was playing around with your code and created a very simple reverse function and thought it might be useful as an academic exercise:

```
reverse: function () {
```

```
var aux,
prev = null;

while (this._head) {
aux = this._head.next;
this._head.next = prev;
prev = this._head;
this._head = aux;
}
this._head = prev;
}
```

I know there are plenty of better codes but I just want to keep it as simple as possible for clarity sake.

Regards,

Marcel

Marcel Duran on July 28th, 2009 at 11:22 pm

An excellent recipie for memory leaks. Given the differences in JS implementations across all the various browsers (especially IE), one cannot assume that simply removing all references to an object will cause it to be deallocated (for those of you learning this stuff, didn't you want to know what happened to the memory used by a deleted object?). For best practice, the discarded object really should be explicitly deleted as part of the remove method.

Sorry if that sounded grumpy, still havent got rid of my new year hangover yet :p

Wolfy on January 3rd, 2011 at 8:24 pm

Comments are automatically closed after 14 days.


Maintainable
JavaScript
O'REILLY


Professional
JavaScript
for Web Developers


High Performance
JavaScript
O'REILLY

Payin' Bills