

## JavaScript Data Structures

### Arrays of Records

There is no separate `struct` type in JavaScript.

But we can think of a structure (or *record*) as a methodless object.

#### *Example*

Here we create an array of instances of the top-level object, `Object`.

Each object in the array has `name` and `age` properties (“fields”).

We’ll sort the array on the `age` field of the records using insertion sort.

We first initialize the array `ar` to contain only the sentinel, with an `age` value less than any valid age.

Function `inputArray` prompts for name-age pairs, adding each pair as a record to the end of the array.

The prompt for the name is in the condition of the `while` loop.

When the user clicks the **cancel** button, `prompt` returns `undefined`, which counts as `false`.

This gives a convenient way to exit the loop.

Function `insertSort` is a standard insertion sort.

Function `outputArray` outputs the records one at a time, both field values enclosed in `{...}`’s.

```
var ar = [ { name: "Sentinel", age: -1 } ];

inputArray( ar );
insertSort( ar );
outputArray( ar );

function inputArray( arr )
{
    var name, age;
    while ( name =
        window.prompt(
            "Name\nClick 'cancel' to exit." ) ) {
        age = parseInt( window.prompt( "Age" ) );
        arr[ arr.length ] =
            { name: name, age: age };
    }
}

function insertSort( arr )
{
    var temp;
    for ( var i = 2; i < arr.length; i++ ) {
        temp = arr[ i ];
        var j = i;
        while ( arr[ j-1 ].age > temp.age ) {
            arr[ j ] = arr[ j-1 ];
            j--;
        }
        arr[ j ] = temp;
    }
}
```

Continued next page

Continued from previous page

```
function outputArray( arr )
{
  document.writeln( "<p>The sorted array:<br>" );
  for ( var i = 1; i < arr.length; i++ )
    document.writeln( "{" + arr[i].name + ", "
                      + arr[i].age + "}  " );
  document.writeln( "</p>" );
}
```

This code is the entire contents of the JavaScript file `sortArray.js`.

The HTML file that is loaded to execute this program has little more than a `script` element with a `src` attribute whose value is the name of this file:

```
<html>
<head>
<title>Sorting an Array</title>
<script type = "text/javascript"
        src = "sortArray.js">
</script>
</head>
</html>
```

We supplied the following pairs of values in the order listed:

```
John  35
Bill  25
Sue   27
Ed    24
Al    30
```

The rendering :

```
The sorted array:
{Ed, 24} {Bill, 25} {Sue, 27} {Al, 30} {John, 35}
```

Using instances of `Object` for records requires little overhead in code.

But it's advantageous to define our own object type when there are certain operations naturally performed on the records.

### *Example*

This is just like the previous example except that we define a `Record` constructor.

When invoked with no arguments, it produces the sentinel record.

It's normally invoked with two arguments: name and age.

`Record` has one method, `toString`, which outputs fields within `{...}`'s.

We can now dispense with the `outputArray` function.

When an array occurs in a position where a string is expected, its elements are output in sequence, using their `toString` methods.

The price we pay is that the sentinel (the first record) is also output.

```
var ar = [ new Record() ];

inputArray( ar );
insertSort( ar );
document.writeln( "<p>The sorted array:<br>"
                  + ar + "</p>" );

function Record( name, age )
{
    this.name = name || "Sentinel";
    this.age  = age  || -1;
    this.toString =
        function()
        { return "{" + this.name + ", "
          + this.age + "} "; };
}

function inputArray( arr )
{
    var name, age;
    while ( name =
            window.prompt(
                "Name\nClick 'cancel' to exit." ) ) {
        age = parseInt( window.prompt( "Age" ) );
        arr[ arr.length ] = new Record( name, age );
    }
}
```

Continued next page

Continued from previous page

```
function insertSort( arr )
{
  var temp;
  for ( var i = 2; i < arr.length; i++ ) {
    temp = arr[ i ];
    var j = i;
    while ( arr[ j-1 ].age > temp.age ) {
      arr[ j ] = arr[ j-1 ];
      j--;
    }
    arr[ j ] = temp;
  }
}
```

When this is executed with the same pairs of values, we get:

The sorted array:

```
{Sentinel, -1} ,{Ed, 24} ,{Bill, 25} ,{Sue, 27} ,{Al, 30} ,{John, 35}
```

## Reference Types and Garbage Collection

JavaScript types are divided into two groups:

- *primitive type*: numbers, boolean values, and the `null` and `undefined` types
- *reference types*: objects, arrays, and functions

A primitive type has a fixed size in memory.

Reference types do not have a fixed size.

The variable stores a *reference* (some form of pointer) to the value.

Consider the following code fragment:

```
var a = [1, 2, 3]; // Initialize 'a' to refer
                  // to an array.
var b = a; // Copy that reference into 'b'.
a[0] = 4; // Modify the array via 'a'.
alert(b); // Display the modified array
          // [4,2,3] via 'b'.
```

Here no copy is made of the array.

The assignment in the second line copied only the reference.

Thereafter, the same array is accessed or modified via either `a` or `b`.

Strings are actually *immutable* reference types.

Once created, they can't be modified.

So there's no analogue of the above example.

There is no syntax for pointers in JavaScript.

When we talk about an identifier for an object, that identifier is really a reference (pointer) variable.

This is similar to the situation with arrays in C and C++.

Since reference types haven't a fixed size, memory must be allocated dynamically for the values referenced.

This memory for a value is freed automatically when JavaScript determines that there is no program variable that refers to it.

This is *garbage collection*.

This is in contrast to C and C++, where memory must be freed manually.

## Linked Lists

To implement linked lists, we have two prototypes:

`ListNode` and `List`.

A `ListNode` has two instance variables: `data` and `next`.

If the constructor is invoked with fewer than two arguments, `next` defaults to `null`.

If the constructor is invoked with no arguments, `data` has the value `undefined`.

As a general rule, we define `get` and `set` methods for each instance variable.

These are used by everything except instances of the prototype.

We also define a `toString` method, which invokes the `toString` method of the `data` value.

```
function ListNode( data, nextNode )
{
  this.data = data;
  this.next = nextNode || null;
  this.getdata =
    function() { return this.data; };
  this.setdata =
    function( data ) { this.data = data; };
  this.getnext =
    function() { return this.next; };
  this.setnext =
    function( nextNode ) { this.next = nextNode; };
  this.toString =
    function() { return this.data.toString(); };
}
```

The following is the file used to test `ListNode`.

Note that node `nd2` is the value of node `nd1`'s next field.

```
var nd2 = new ListNode( 2 ),
    nd1 = new ListNode( 3, nd2 );

document.writeln( "<p>" + nd1 + " "
                  + nd1.next + "</p>" );

nd1.setdata( 4 );
nd1.setnext( new ListNode( 5 ) );

document.writeln( "<p>" + nd1.getdata() + " "
                  + nd1.getnext().getdata() + "</p>" );
```

The following is the HTML file that executes both these files.

There are two separate `script` elements with `src` attributes.

File `node.js` contains the `ListNode` constructor.

File `testNode.js` contains the test code.

```
<html>
<head>
<title>Tests</title>
<script type = "text/javascript" src = "node.js">
</script>
<script type = "text/javascript" src = "testNode.js">
</script>
</head>
</html>
```

The following is the rendering:

```
3 2
4 5
```

A `List` is a sequence of `ListNodes` linked by their `next` fields.

There is no dummy head or tail node.

A `List` has instance variables `name`, `firstNode`, and `lastNode`.

The constructor has one argument, for the `name`, which defaults to `"a list"` if this argument isn't supplied.

When constructed, a `List` is empty:

`firstNode` and `lastNode` are both `null`.

We define `get` and `set` methods for `name` but not for `firstNode` or `lastNode`.

These are not meant to be accessed by instances of other prototypes.

This implementation uses standard techniques

- to determine whether the list is empty (method `isEmpty`),
- to insert a node at the front or back, and
- to remove a node from the front or back, returning its `data` value.

The `toString` method returns the concatenation of the string representations of the list nodes.

The `isEmpty` property is assigned an anonymous function in the constructor.

The properties corresponding to the other methods are set as properties of the `List.prototype` property outside the constructor.

They are assigned the appropriate functions defined in this file.

```
function List( name )
{
  this.name = name || "a list";
  this.firstNode = this.lastNode = null;
  this.getname =
    function() { return this.name; };
  this.setname =
    function( name ) { this.name = name; };
  this.isEmpty =
    function() { return this.firstNode == null; };
}

function List_insertAtFront( insertItem )
{
  if ( this.isEmpty() )
    this.firstNode = this.lastNode
      = new ListNode( insertItem );
  else
    this.firstNode
      = new ListNode( insertItem, this.firstNode );
}

function List_insertAtBack( insertItem )
{
  if ( this.isEmpty() )
    this.firstNode = this.lastNode
      = new ListNode( insertItem );
  else {
    this.lastNode.setnext(
      new ListNode( insertItem ) );
    this.lastNode = this.lastNode.getnext();
  }
}
```

Continued next page

---

Continued from previous page

```
function List_removeFromFront()
{
    if ( this.isEmpty() )
        return null;

    var removeItem = this.firstNode.getdata();

    if ( this.firstNode == this.lastNode )
        this.firstNode = this.lastNode = null;
    else
        this.firstNode = this.firstNode.getNext();

    return removeItem;
}

function List_removeFromBack()
{
    if ( this.isEmpty() )
        return null;

    var removeItem = this.lastNode.getdata();

    if ( this.firstNode == this.lastNode )
        this.firstNode = this.lastNode = null;
    else {
        var p = this.firstNode;

        while ( p.getNext() != this.lastNode )
            p = p.getNext();

        this.lastNode = p;
        p.setnext( null );
    }

    return removeItem;
}
```

---

Continued next page

Continued from previous page

```
function List_toString()
{
    var p    = this.firstNode,
        str = "";

    while ( p ) {
        str += p + " ";
        p = p.next;
    }

    return str;
}

List.prototype.insertAtFront
    = List_insertAtFront;
List.prototype.insertAtBack
    = List_insertAtBack;
List.prototype.removeFromFront
    = List_removeFromFront;
List.prototype.removeFromBack
    = List_removeFromBack;
List.prototype.toString
    = List_toString;
```

The following is a test file for this implementation:

```
var lst = new List( "test" );
lst.insertAtFront( 2 );
lst.insertAtBack( 3 );
lst.insertAtFront( 1 );
document.writeln( "<p>" + lst + "</p>" );

lst.removeFromFront();
lst.removeFromBack();
document.writeln( "<p>" + lst + "</p>" );
```

The following is the HTML file that executes these files.

File `list.js` contains the `List` definition.

File `testList.js` contains the test code.

We also include `node.js` since `List` uses `ListNode`.

```
<html>
<head>
<title>List Tests</title>
<script type = "text/javascript" src = "node.js">
</script>
<script type = "text/javascript" src = "list.js">
</script>
<script type = "text/javascript" src = "testList.js">
</script>
</head>
</html>
```

The following is the rendering:

```
1 2 3
2
```

It's obvious how to implement stacks and queues as linked lists.

In fact, stacks and queues can be implemented easily using array operations that we'll look at later.

Arrays in JavaScript, as we'll see, are essentially lists.

One can also use standard techniques to implement sorted lists.

But, in many of the situations where we'd use a sorted list (e.g., quick updates and searches), an associative array is a better choice.