Searching Linked Lists in JavaScript

Algorithms and Data Structures I

1 Objectives

In this reading material, the objectives are:

1. Give a brief introduction to objects and object constructors in JavaScript
2. Describe how objects can be used to construct a linked list
3. Describe how a linked list in JavaScript can be searched

2 Objects in JavaScript

We have previously covered JavaScript Arrays, and in doing so, we saw that they are *objects* in JavaScript. What is an object? The concept of an object appears in many programming languages, and in its simplest sense it is a collection of data. Indeed, the simplest kinds of objects are data structures. However, in many languages objects can have *methods*, which allow one to manipulate the data within objects; at this point objects cease to be collections of data (or data structures) and become something more complicated.

For our purposes we will focus on the simplest kinds of objects. In this case an object consists of *properties*: each property has a name and a value. For example, we could take Old McDonald's farm and the properties as animals in the farm (with the values being the number of the animals). The corresponding object would look like this:

```
var farm = {
  cows : 20,
  pigs : 10,
  ducks : 6,
  lambs : 5,
  chickens: 40
}
```

Here we create a variable called `farm` to which we will assign an object. The whole object is inside the curly brackets, and we see a comma-separated list of properties of the form `name:value`: the name of the property is on the left and the value is on the right. Thus the number of cows in Old McDonald's farm is 20, and the number of chickens is 40. The values need not be numbers; they could be any of the simple data types in JavaScript: Booleans, strings, undefined, null or even another object. We will be using the null data type later on, which signifies nothing. One of the quirks of JavaScript is that null is an object.

From the object we can extract the values of particular properties. For example, for `farm`, we can obtain the number of ducks by typing `farm.ducks`. If we type `var ducks = farm.ducks`, this will create a new variable called `ducks` and assign the value 6 to it. In general for an object called `obj` with property `prop`, `obj.prop` will store the value associated with property `prop`. We can think of the property name `prop` of `obj` as a *pointer* telling us where a value is stored, and then we *dereference* the pointer by using the syntax `obj.prop`. This will be very useful when we come to think about linked lists.

We can make our own objects from scratch by creating a new variable and assigning it properties as above. This is cumbersome if we wish to create a lot of objects that are essentially the same. For example, Old McDonald might have a friend called Young McKenzie who has a farm with the same species of animals, but with different numbers. To be able to make multiple similar objects we use something called a *constructor*. Note that in your Introduction to Programming course this is called the *factory pattern*, but constructor is the more standard piece of terminology so I will use it.

Imagine we want to create lots of objects like `farm` but with the numbers of animals changing from farm to farm (this number could also be 0). To do this we create a function that takes the numbers of the answers as arguments and makes an object. For the case of the farms, the function will look like this:

```
function Farm(cow, pig, duck, lamb, chicken) {
  this.cows = cow;
  this.pigs = pig;
  this.ducks = duck;
  this.lambs = lamb;
  this.chickens = chicken
}
```

Note that we are using the syntax `this.name` and assigning it the values in the arguments of the function. The syntax `this.name` just says that the object created by this function will have the property `name`, and we can assign it a particular value. To now use the constructor, consider the following:

```
var farm = new Farm(20, 10, 6, 5, 40);
```

This will create the same variable `farm` as above. The important thing to note is the use of `new`, which says that we are creating a new object, followed the kind of object this will be, or which class it belongs to. In our case, the object `farm` will belong to the class `Farm`. By convention, we use the capital letter at the beginning of class names to indicate something is a class.

3 Linked lists with objects

Recall that a linked list consists of a collection of nodes, with each node containing a data field, and another element containing the value of a pointer to the next node. Therefore, each node of a linked list is a collection of data. As previously mentioned we do not have pointers as a basic data type in JavaScript so we have to be more creative. However, as alluded to in the previous section, we can use the names of properties in an object as a pointer.

Each node of a linked list will be implemented in JavaScript by an object with two properties: the first property is `data`, which is the data stored in the data field of a node; the second property is `next`, which will correspond to a pointer to the next node. The property `next` could be an object corresponding to another node, or it could be `null`.

In the following we are going to create some JavaScript code, so you will need to set up the environment:

1. Create a folder in your "Documents" folder and name the folder `linked`
2. Create an empty JavaScript file called `linked.js` and save it to the folder `linked`
3. Go to your CLI and change the directory to the folder `linked`

Now open up the JavaScript file `linked.js` and write the following code to the file:

```
var head = null;
head = {data : 5, next : null};
console.log(head);
console.log(head.data);
console.log(head.next);
```

Now save the file and run it by entering `node linked` into the CLI.

You should see the following printed:

```
{ data: 5, next: null }
5
null
```

Let's unpack what is going on here. The variable `head` represents the head of the linked list, which will point to the beginning of the linked list. In this code we initially have `head` being assigned the value `null`, thus it is pointing at nothing. In other words, we have an empty linked list. Then in the next line we will assign a new object to `head`, thus corresponding to a single node. We assign to `head` a new object containing two properties: `data` contains the data stored at the node (in this case the number 5), and `next` is the property that points at the next node. In the second line, `next` is assigned `null`, thus adding the new node to the end of the empty list. Therefore we have an implementation of the following linked list:

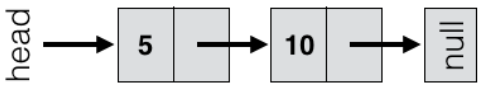
When we return to the code we see that the value stored in `head.data` "dereferences" the head pointer and reveals the value stored in the data field, and then `head.next` stores the pointer to the next node, or null.

Instead of adding a node to the end of a list by hand, we can use a constructor to do this for us. Remove all the current code in the file `linked.js` and put the following:

```
function LLNode(data) {
  this.data = data;
  this.next = null;
}
var head = new LLNode(5);
console.log(head);
console.log(head.data);
console.log(head.next);
```

When you run this from the CLI, you should essentially get the same thing printed as with the previous code (that was deleted) but with `LLNode` added to point out that `head` belongs to this class of objects.

With this constructor, we now have a method for adding elements to the end of a linked list. Consider the following linked list:

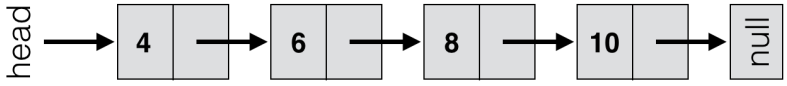


This is the linked list from earlier with a new node added at the end that stores the number 10. We will now implement this linked list based on the code already in `linked.js`. Put the following code under your existing code in `linked.js`:

```
head.next = new LLNode(10);
console.log(head.next.data);
console.log(head.next.next);
```

After running this code, the final two values printed to the console should be `10` and `null` respectively.

Exercise 1: Construct the following linked list through repeated use of the `LLNode` constructor.



4 Searching Linked Lists

To search a linked list we need to *traverse* the list; this is done by updating the value of a temporary pointer (called `temp`) to be the value of the next pointer as we go along the list. Consider the following piece of JavaScript:

```
var temp = head;
while (temp !== null) {
  temp = temp.next;
}
```

This loop will traverse the linked list by assigning a variable `temp` the value stored at `head`, then while `head` is not pointing to null, `temp` is updated to have the value of the next pointer. The loop is exited when `temp` takes the value `null`. We will use this technique to search for a value in the data fields of the nodes in the linked list.

In the JavaScript file `linked.js`, put the following code under the existing code:

```
function searchLL(head, item) {
  var temp = head;
  while (temp !== null) {
    if (temp.data === item) {
      return true;
    }
    temp = temp.next;
  }
  return false;
}
```

This will return `true` if the value `item` is stored in one of the `data` properties of one of the nodes, and `false` otherwise. One of the arguments to `searchLL` is `head`, which will be the object stored at `head` corresponding to the linked list. Test this function on various arguments: on the linked lists constructed above change `item` to have values that are, and are not, in the list.

Exercise 2: Write a function called `numberLL` that will take two input arguments: the head of a list called `head`, and a value called `item`. The function should return the number of times that `item` appears in the list, which should be a non-negative number.