# Linked Lists

For many applications, data are best stored as lists, and lists occur naturally in our day-to-day life – to-do lists, grocery lists, and top-ten lists. In this chapter we explore one particular type of list – the linked list. The chapter starts with an explanation of why we need linked lists, then we explore two different implementations of linked lists – object-based linked lists and array-based linked lists. The chapter finishes up with several examples of how linked lists can be used for solving computer programming problems you may run across.

## Shortcomings of Arrays

The array is the natural data structure to use when working with lists. Arrays provide fast access to stored items and are easy to loop through. And, of course, the array is already part of the language so you don't have the extra overhead of a user-defined data structure.

However, as we've seen, the array is not the perfect data structure. Searching for an item in an unordered array can be slow because you might have to visit every element in the array before finding the element you are looking for. Ordered (sorted) arrays are much more efficient for searching, but insertions and deletions take extra time because you have to shift the array elements up or down to either make space for an insertion or to free up space with a deletion. Moreover, in an ordered array, you have to search for the proper place in the array into which to insert an element.

Furthermore, JavaScript arrays are really objects rather than true arrays, meaning that operations on arrays in JavaScript are not as fast as array operations in other programming languages (Crockford, Chapter 6).

When you determine that the operations performed on an array are too slow for practical use, you can consider using the linked list as an alternative data structure. The linked list can be used in almost every situation when a single-dimension array is used, excepting when you need random access to the items in a list. In that situation, an array is probably the better data structure to use.

## Linked Lists Defined

A linked list is a collection of objects called *nodes*. Each node is linked to its successor node in the list using a reference. A node consists of a field for storing data and a field for the node reference. The reference to another node is called a *link*. An example of a linked list is shown in Figure x.1.

A major difference between an array and a linked list is that where the elements of an array are referenced by their position, the elements of a linked list are referenced by their relationship to the other elements of the linked list. In Figure x.1, we say that "bread" follows "milk", not that "bread" is in the second position. Moving through a linked list involves following the links of the list from the beginning node to the ending node (not including the header node, which is sometimes used as a means of getting into a linked list). Another thing to notice in Figure x.1 is that we mark the end of a linked list by point to the null value.

Marking the beginning of a linked list can be problematic. Many linked list implementations commonly include a special node, called the head, to denote the beginning of the linked list. The linked list shown in Figure x.1 is redesigned with a head node in Figure x.2.

Insertion is a very efficient task when using a linked list. To insert a new node, the link of the node before the inserted node (the previous node) is changed to point to the new node, and the new node's link is set to the node the previous node was pointing to before the insertion. Figure x.3 demonstrates how "cookies" is added to a linked list after "eggs."

Removing an item from a linked list is also easy to do. We simply redirect the link of the node before the deleted node to point to the node the deleted node is pointing to, and we also have the deleted node point to null, which effectively takes the node out of the linked list, since no node points to it and it doesn't point to any other node. Figure x.4 demonstrates how "bacon" is removed from a linked list.

There are, of course, other functions we can perform with a linked list, but insertion and removal are the two functions that best describe why linked lists are so useful.

## An Object-Based Linked List Design

Our design of a linked list will involve creating two objects. We'll create a Node object for adding nodes to a linked list and we'll create a Linked List object, which will provide functions for inserting nodes into the list, removing nodes from the list, displaying the list, and other housekeeping functions.

### The Node Object

A Node object is made up of two properties – element, which stores the node's data, and next, which stores the link to the next node in the list. To create nodes, we'll use a constructor function that sets the value for the two properties:

function Node(element) {

this.element = element;

this.next = null;

}

### The Linked List Object

The Linked List object provides the functionality for a linked list. The object includes methods for inserting new nodes, removing nodes, and finding a particular element in the list. We'll also include a constructor function for creating a new Linked List object. The only property the object holds is a node to represent the head of the list.

Here is the definition for the constructor function:

function LList() {

this.head = new Node("head");

this.find = find;

this.insert = insert;

this.remove = remove;

this.display = display;

}

The head node starts with its next property set to null and is changed to point to the first item inserted into the list. The new node's next property is not changed from null in order to indicate the end of the list.

The first method we'll look at is the insert method, which inserts a node into the list. To insert a new node into the list, you have to specify which node you want to insert the node before or after. Actually, there should be a method for each possibility, so we'll have our insert method insert nodes after the specified node.

To insert a node after an existing node, we have to first find the "after" node. To do this, we create another method, find, which searches through the linked list looking for the specified element. When the method finds the element, it returns the node storing the element. Here is the code for the find method:

function find(item) {

var currNode = this.head;

while (currNode.element != item) {

currNode = currNode.next;

}

return currNode;

}

The find method demonstrates how to move through a linked list. First, we create a new node and assign it the head node. Then we loop through the linked list, moving from one node to the next when the value of the current node's element is not equal to the value we are looking for. As long as the value we are searching for is stored somewhere in the list, the method returns the node storing that value. If the value isn't found, the method returns null.

Once we've found the "after" node, we are ready to insert the new node into the linked list. We do this by setting the new node's next property to the next property of the "after" node, followed by setting the "after" node's next property to a reference to the new node. Here is the definition of the insert method:

function insert(newElement, item) {

var newNode = new Node(newElement);

var current = this.find(item);

newNode.next = current.next;

current.next = newNode;

}

At this point, we're ready to test our linked list code. In order to determine that our code is working, we need a method that displays the elements of the linked list. A display method is defined below:

function display() {

var currNode = this.head;

while (!(currNode.next == null)) {

console.log(currNode.next.element);

currNode = currNode.next;

}

}

We start the definition by linking into the linked list by assigning the head of the list to a new node object. Next, we loop through the list, only stopping when the value of the current node's next property is set to null. To display only the nodes with element data in them (in other words, not the head node), we access the element property of the node the current node is pointing to:

currNode.next.element

Finally, we need to add some code to use the linked list. Here is a simple program fragment that sets up a linked list of cities in western Arkansas:

var cities = new LList();

cities.insert("Conway", "head");

cities.insert("Russellville", "Conway");

cities.insert("Alma", "Russellville");

cities.display()

The output from this program is:

Conway

Russellville

Alma

### Removing Nodes from a Linked List

In order to remove a node from a linked list, we need to find the node just before the node we are trying to remove in order to change its next property to no longer reference the removed node. We can define a method, findPrevious, to perform this task for us. This method works it way through a linked list, stopping at each node to see if the next node stores the element that is to be removed. When the item is found, the method returns this node (the "previous" node) so that its next property can be modified. Here is the code:

function findPrevious(item) {

var currNode = this.head;

while (!(currNode.next == null) &&

(currNode.next.element != item)) {

currNode = currNode.next;

}

return currNode;

}

Now we're ready to write the remove method:

function remove(item) {

var prevNode = this.findPrevious(item);

if (!(prevNode.next == null)) {

prevNode.next = prevNode.next.next;

}

}

The main line of code:

prevNode.next = prevNode.next.next

looks odd but it makes perfect sense; we are just skipping over the node we want to remove to and linking up with the next node in the list.

We are ready now to test our code again, but first we need to modify the constructor function for the linked list to include the new methods:

function LList() {

this.head = new Node("head");

this.find = find;

this.insert = insert;

this.display = display;

this.findPrevious = findPrevious;

this.remove = remove;

}

Here is some code to test the removal of a node:

var cities = new LList();

cities.insert("Conway", "head");

cities.insert("Russellville", "Conway");

cities.insert("Carlisle", "Russellville");

cities.insert("Alma", "Carlisle");

cities.display();

console.log();

cities.remove("Carlisle");

cities.display();

The output of this program before the removal is:

Conway

Russellville

Carlisle

Alma

But Carlisle is in eastern Arkansas so we need to remove it from the list, resulting in the following output:

Conway

Russellville

Alma

Here is a complete listing of the linked list code, including the test program:

function Node(element) {

this.element = element;

this.next = null;

}

function LList() {

this.head = new Node("head");

this.find = find;

this.insert = insert;

this.display = display;

this.findPrevious = findPrevious;

this.remove = remove;

}

function remove(item) {

var prevNode = this.findPrevious(item);

if (!(prevNode.next == null)) {

prevNode.next = prevNode.next.next;

}

}

function findPrevious(item) {

var currNode = this.head;

while (!(currNode.next == null) &&

(currNode.next.element != item)) {

currNode = currNode.next;

}

return currNode;

}

function display() {

var currNode = this.head;

while (!(currNode.next == null)) {

console.log(currNode.next.element);

currNode = currNode.next;

}

}

function find(item) {

var currNode = this.head;

while (currNode.element != item) {

currNode = currNode.next;

}

return currNode;

}

function insert(newElement, item) {

var newNode = new Node(newElement);

var current = this.find(item);

newNode.next = current.next;

current.next = newNode;

}

var cities = new LList();

cities.insert("Conway", "head");

cities.insert("Russellville", "Conway");

cities.insert("Carlisle", "Russellville");

cities.insert("Alma", "Carlisle");

cities.display();

console.log();

cities.remove("Carlisle");

cities.display();

## Doubly Linked Lists

Although traversing a linked list from the first node to the last node is very straightforward, it is not as easy to traverse a linked list backwards. We can simplify this procedure if we add a property to our Node object that stores a link to the previous node. When we insert a node into the list, we'll have to perform more operations to assign the proper links for the next and previous nodes, but we gain efficiency when we have to remove a node from the list, since we no longer have to search for the previous node. Figure x.5 illustrates how a doubly linked list works.

Our first task is to add a previous property to our Node object:

function Node(element) {

this.element = element;

this.next = null;

this.previous = null;

}

The insert method is similar to the insert method for the singly linked list, except we have to set the new node's previous property to point to the previous node. Here is the code for the new insert method:

function insert(newElement, item) {

var newNode = new Node(newElement);

var current = this.find(item);

newNode.next = current.next;

newNode.previous = current;

current.next = newNode;

}

The remove method for a doubly linked list is more efficient because now we don't have to find the previous node. We first need to find the node in the list that is storing the item we want to remove. Then we set that node's previous property to point to the node pointed to by the deleted node's next property. Then we need to redirect the previous property of the node the deleted node points to and point it to the node before the deleted node. Figure x.6 makes this scenario easier to understand. Here is the code for the remove method:

function remove(item) {

var currNode = this.find(item);

if (!(currNode.next == null)) {

currNode.previous.next = currNode.next;

currNode.next.previous = currNode.previous;

currNode.next = null;

currNode.previous = null;

}

}

For different reasons, we might need a utility method that finds the last node in a doubly linked list. One reason for needing this method is if we want to display the elements of a double linked list in reverse order. The following method, findLast(), moves us to the end of a list without going past the end:

function findLast() {

var currNode = this.head;

while (!(currNode.next == null)) {

currNode = currNode.next;

}

return currNode;

}

With the findLast() method in hand, we can write a method to display the elements of a doubly linked list in reverse order. Here is the code for the dispReverse() method:

function dispRevere() {

var currNode = this.head;

currNode = this.findLast();

while (!(currNode.previous == null)) {

console.log(currNode.element);

currNode = currNode.previous;

}

}

The last task to accomplish is to add these new methods to the constructor function for the doubly linked list. Here is that code, along with the rest of the code for implementing a doubly linked list, as well as a short program to test the code:

function Node(element) {

this.element = element;

this.next = null;

this.previous = null;

}

function LList() {

this.head = new Node("head");

this.find = find;

this.insert = insert;

this.display = display;

this.remove = remove;

this.findLast = findLast;

this.dispReverse = dispReverse;

}

function dispReverse() {

var currNode = this.head;

currNode = this.findLast();

while (!(currNode.previous == null)) {

console.log(currNode.element);

currNode = currNode.previous;

}

}

function findLast() {

var currNode = this.head;

while (!(currNode.next == null)) {

currNode = currNode.next;

}

return currNode;

}

function remove(item) {

var currNode = this.find(item);

if (!(currNode.next == null)) {

currNode.previous.next = currNode.next;

currNode.next.previous = currNode.previous;

currNode.next = null;

currNode.previous = null;

}

}

// findPrevious is no longer needed

/\*function findPrevious(item) {

var currNode = this.head;

while (!(currNode.next == null) &&

(currNode.next.element != item)) {

currNode = currNode.next;

}

return currNode;

}\*/

function display() {

var currNode = this.head;

while (!(currNode.next == null)) {

console.log(currNode.next.element);

currNode = currNode.next;

}

}

function find(item) {

var currNode = this.head;

while (currNode.element != item) {

currNode = currNode.next;

}

return currNode;

}

function insert(newElement, item) {

var newNode = new Node(newElement);

var current = this.find(item);

newNode.next = current.next;

newNode.previous = current;

current.next = newNode;

}

var cities = new LList();

cities.insert("Conway", "head");

cities.insert("Russellville", "Conway");

cities.insert("Carlisle", "Russellville");

cities.insert("Alma", "Carlisle");

cities.display();

console.log();

cities.remove("Carlisle");

cities.display();

console.log();

cities.dispReverse();

The output from this program is:

Conway

Russellville

Carlisle

Alma

Conway

Russellville

Alma

Alma

Russellville

Conway

## Circularly Linked Lists

A circularly linked list is similar to a singly linked list and has the same type of nodes. The only difference is that a circularly linked list, when created, has its head node's next property point back to itself. This means that the assignment:

head.next = head

is propagated throughout the circularly linked list so that every new node has its next property pointing to the head of the list. In other words, the last node of the list is always pointing back to the head of the list, creating a circular list.

To create a circularly linked list, change the constructor function of the Linked List object to read:

function LList() {

this.head = new Node("head");

this.head.next = this.head;

this.find = find;

this.insert = insert;

this.display = display;

this.findPrevious = findPrevious;

this.remove = remove;

}

This is the only change we have to make in order to make a singly linked list a circularly linked list. However, some of the other linked list methods will not work correctly unmodified. For example, one method that needs to be modified is display(). As written, if the display() method is executed on a circularly linked list, the method will never stop. The modification that needs to occur is changing the condition of the while loop so that the head element is examined so that the loop will stop right before the head node is encountered. Here is how the new display() method should look:

function display() {

var currNode = this.head;

while (!(currNode.next == null) &&

!(currNode.next.element == "head")) {

console.log(currNode.next.element);

currNode = currNode.next;

}

}

Other methods may need modification also, but we will leave those modifications, if any, as an exercise for the reader.

## Other Linked List Methods

There are several other methods that you might include in order to have a well-functioning linked list. In the upcoming exercises, the reader will have the opportunity to develop some of these methods, including:

* advance(n) – advances n nodes in the linked list
* back(n) – moves n nodes backward in a doubly linked list
* show() – displays the current node only

## Exercises

1. Implement the advance(*n*) method so that when executed, the current node is moved n nodes forward in the list.
2. Implement the back(*n*) method so that when executed, the current node is moved n spaces backwards in the list.
3. Implement the show() method, which displays the element associated with the current node.
4. Write a program that uses a singly linked list to keep track of a set of test grades entered interactively into the program.
5. Later in the book, we will examine associative data structures. An associative data structure stores name/value pairs, so that when the name is given, the value is retrieved. Implement such an associative data structure using a singularly linked list.
6. Rewrite your solution for Exercise 5 using a doubly linked list.
7. According to legend, the 1st-century Jewish historian, Flavius Josephus, were about to be captured along with a band of 40 compatriots by Roman soldiers during the Jewish-Roman war. The Jewish soldiers decided that they preferred suicide to being captured and devised a plan for their demise. They were to form a circle and kill every third soldier until they were all dead. Josephus and one other decided they wanted no part of this and quickly calculated where they needed to place themselves so they would be the last survivors. Write a program that allows you to place *n* people in a circle and specify that every *m*th person will be killed. The program should determine the number of the last two people left in the circle. Use a circularly linked list to solve the problem.