05. DATASTRUCTS Linked Lists.

In previous chapters we discussed the use of lists for storing data. The underlying data storage mechanism we use for lists is the **array**. In this chapter we'll discuss a different type of list, the **linked list**. We'll explain why linked lists are sometimes preferred to arrays, and we'll develop an object-based, linked-list implementation. We'll end the chapter with several examples of how linked lists can solve many programming problems you will encounter.

5.1. Shortcomings of Arrays

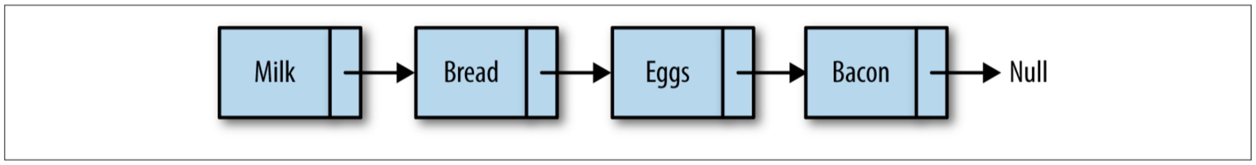
There are several reasons **arrays are not always the best data structure to use for organizing data.** In many programming languages, **arrays are fixed in length**, so it is hard to add new data when the last element of the array is reached. **Adding and removing data from an array is also difficult** because you have to move array elements up or down to reflect either an addition or a deletion. However, these problems do not come up with JavaScript arrays, since we can use the split() function without having to perform additional array element accesses.

The main problem with using JavaScript arrays, however, is that arrays in JavaScript are implemented as objects, causing them to be less efficient than arrays built in languages such as C++ and Java.

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 where a one-dimensional array is used, except when you need random access to the elements of a list. **When random access is required, an array is the better data structure to use.**

5.2. Linked Lists Defined.

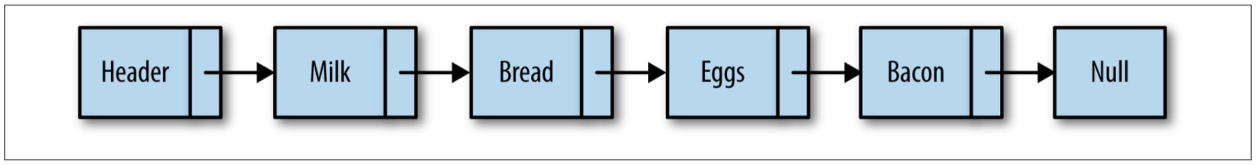
A linked list is a collection of objects called nodes. Each node is linked to a successor node in the list using an object reference. The reference to another node is called a **link**. An example of a linked list is shown here.



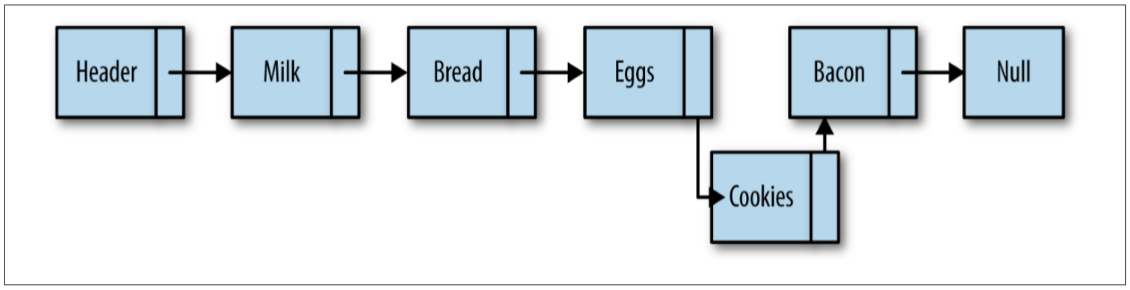
While array elements are referenced by their position, linked list elements are referenced by their relationship to the other elements of the linked list.

In the previous figure, 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 end node (not including the header node, which is sometimes used as a hook for entry into a linked list). Something else to notice in the figure is that we mark the end of a linked list by pointing to a null node.

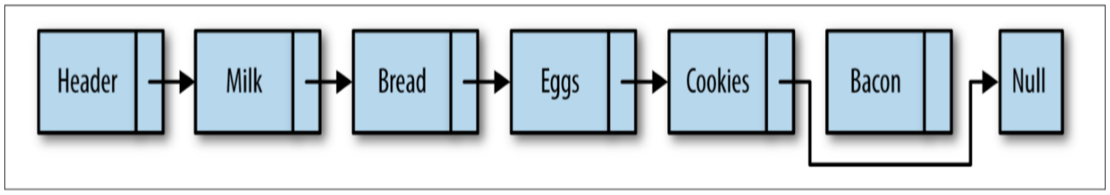
Marking the beginning of a linked list can be a problem. Many linked-list implementations include a special node, called the head, to denote the beginning of a linked list.



Inserting a new node into a linked list is a very efficient task. 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. Next figure illustrates how "*cookies*" is added to the linked list after "*eggs*."



Removing an item from a linked list is also easy to do. The link of the node before the removed node is redirected to point to the node the removed node is pointing to, while also pointing the removed node to null, effectively taking the node out of the linked list. Next figure shows how "*bacon*" is removed from the linked list.



There are 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.

5.3. An Object-Based Linked List Design

Our design of a linked list will involve creating two classes. We'll create a **Node** class for adding nodes to a linked list, and we'll create a **LinkedList** class, which will provide functions for inserting nodes, removing nodes, displaying a list, and other housekeeping functions.

5.2.1. The Node Class

The ***Node*** class consists of two properties: **element**, which store's the node's data, and **next**, which stores a link to the next node in the linked list. To create nodes, we'll use a **constructor** function that sets the values for the two properties:

Example: *Node* class constructor:

```
class Node {
   constructor(element){
      this.element = element;
      this.next = null;
   }
}
```

5.2.2. The Linked List Class.

The *LinkedList* class provides the functionality for a linked list. The class includes functions for inserting new nodes, removing nodes, and finding a particular data value in a list. There is also a constructor function used for creating a new linked list. The only property stored in a linked list is a node to represent the head of the list.

Here is the definition for the constructor function:

Example: *LinkedList* class constructor:

```
class LinkedList {
   constructor(){
      this.head = new Node("head");
   }
}
```

The *head* node starts with its next property set to null and is changed to point to the first element inserted into the list, since we create a new **Node** element but don't modify its next property here in the constructor.

5.2.3. Inserting New Nodes.

The first function we'll examine is the **insert** function, which **inserts a node into a list**. To insert a new node, you have to specify which node you want to insert the new node before or after. We'll start by demonstrating how to insert a new node after an existing node.

To insert a node after an existing node, we first have to find the "after" node. To do this, we create a helper function, **find**(), which searches through the linked list looking for the specified data. When this data is found, the function returns the node storing the data. Here is the code for the **find**() function:

Example: Linked List *find*() method.

```
    find(item) {
       var currNode = this.head;
       while (currNode.element != item) {
          currNode = currNode.next;
       }
       return currNode;
    }
```

The **find**() function demonstrates how to move through a linked list. First, we create a new node and assign it as 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** property is not equal to the data we are searching for. If the search is successful, the function returns the node storing the data. If the data is not found, the function returns null.

Once the "*after*" node is found, the new node is inserted into the linked list. First, the new node's next property is set to the value of the next property of the "after" node.

Then the "after" node's next property is set to a reference to the new node. Here is the definition of the ***insert***() function:

Example: Linked List *insert*() method.

```
    insert(newElement, item) {
       var newNode = new Node(newElement);
       var current = this.find(item);
       newNode.next = current.next;
       current.next = newNode;
    }
```

5.2.4. Display the elements of a Linked List.

We're ready now to test our linked list code. However, before we do that, we need a function that will display the elements of a linked list. The ***display***() function is defined below:

Example: Linked List *display*() method.

```
      display() {
       var currNode = this.head;
       while (!(currNode.next == null)) {
          console.log(currNode.next.element);
          currNode = currNode.next;
       }
    }
```

This function starts by hooking into the linked list by assigning the head of the list to a new node. We then loop through the linked list, only stopping when the value of the current node's next property is set to null. In order to display only nodes with data in them (in other words, not the head node), we access the element property of the node that the current node is pointing to: **currNode.next.element**

5.2.5. Linked List Class Implementation.

Finally, we need to add some code to use the linked list.

Example: *LinkedList* class implementation.

```
class LinkedList {

   constructor(){
      this.head = new Node("head");
   }

    find(item) {
       var currNode = this.head;
       while (currNode.element != item) {
          currNode = currNode.next;
       }
       return currNode;
    }

    insert(newElement, item) {
       var newNode = new Node(newElement);
       var current = this.find(item);
       newNode.next = current.next;
       current.next = newNode;
    }

   display() {
      var currNode = this.head;
      while (!(currNode.next == null)) {
         console.log(currNode.next.element);
         currNode = currNode.next;
      }
   }

}
```

The following code contains a short program that sets up a linked list of cities in western Arkansas that are located on Interstate 40, along with the complete *LinkedList* class definition up to this point.

Example:

```
// main program

var cities = new Linkedlist();

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

cities.display()
```

The output of the previous code is:

Output:

```
Conway
Russellville
Alma
```

5.2.6. Removing Nodes from a Linked List.

In order to remove a node from a linked list, we need to find the node that is just before the node we want to remove. Once we find that node, we change its next property to no longer reference the removed node, and the previous node is modified to point to the node after the removed node. We can define a function, **findPrevious**(), to perform this task. This function traverses the linked list, stopping at each node to see if the next node stores the data that is to be removed. When the data is found, the function returns this node (the "previous" node), so that its next property can be modified. Here is the definition for **findPrevious**():

Example: *findePrevious*() method.

```
 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**() function:

Example: *remove*() method.

```
    remove(item) {
     var prevNode = this.findPrevious(item);
     if (!(prevNode.next == null)) {
         prevNode.next = prevNode.next.next;
     }
   }
```

The main line of code in this function looks odd, but makes perfect sense:

prevNode.next = prevNode.next.next

We are just skipping over the node we want to remove and linking the "*previous*" node with the node just after the one we are removing.

5.2.7. Testing LinkedList class implementation.

Finally, we need a function that lets us know if a queue is empty:

Example: Testing the *remove*() method.

```
var cities = new LinkedList();

cities.insert("Conway", "head");
cities.insert("Russellville", "Conway");
cities.insert("Carlisle", "Russellville");
cities.insert("Alma", "Carlisle");
cities.display();

cities.remove("Carlisle");
cities.display();
```

Output: The output from the previous code is:

```
Conway Russellville Carlisle Alma

Conway Russellville Alma
```

5.2.8.Complete Linked List class implementation.

The following code contains a complete listing of the *Node* class, the *LinkedList* class, and our test program:

Example: Testing the *LinkedList* class implementation.

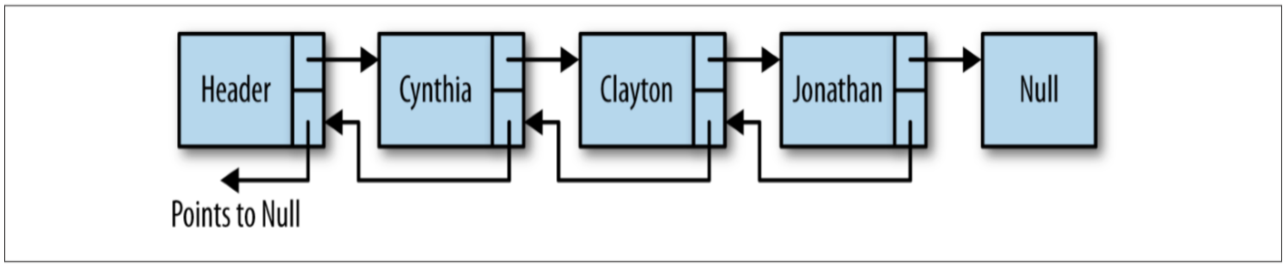
```
// Node class implementation

class Node {
   constructor(element){
      this.element = element;
      this.next = null;
   }
}

// LinkedList class implementation

class LinkedList {

   constructor(){
      this.head = new Node("head");
   }

    find(item) {
       var currNode = this.head;
       while (currNode.element != item) {
          currNode = currNode.next;
       }
       return currNode;
    }

    insert(newElement, item) {
       var newNode = new Node(newElement);
       var current = this.find(item);
       newNode.next = current.next;
       current.next = newNode;
    }

   display() {
      var currNode = this.head;
      while (!(currNode.next == null)) {
         console.log(currNode.next.element);
         currNode = currNode.next;
      }
   }

   remove(item) {
      var prevNode = this.findPrevious(item);
      if (!(prevNode.next == null)) {
         prevNode.next = prevNode.next.next;
```

```
          }
      }

      findPrevious(item) {
         var currNode = this.head;
         while(!(currNode.next == null)&&
                (currNode.next.element != item)){
            currNode = currNode.next;
         }
         return currNode;
      }

    }


    // test program

    var cities = new LinkedList();

    cities.insert("Conway", "head");
    cities.insert("Russellville", "Conway");
    cities.insert("Carlisle", "Russellville");
    cities.insert("Alma", "Carlisle");
    cities.display();

    cities.remove("Carlisle");
    cities.display();
```

5.3. Doubly Linked Lists

Although traversing a linked list from the first node to the last node is straightforward, it is not as easy to traverse a linked list backward. We can simplify this procedure if we add a property to our ***Node*** class 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. Next figure illustrates how a doubly linked list works.



5.3.1. Assigning Partners at a Square Dance

Our first task is to assign a ***previous*** property to our **Node** class:

Example:

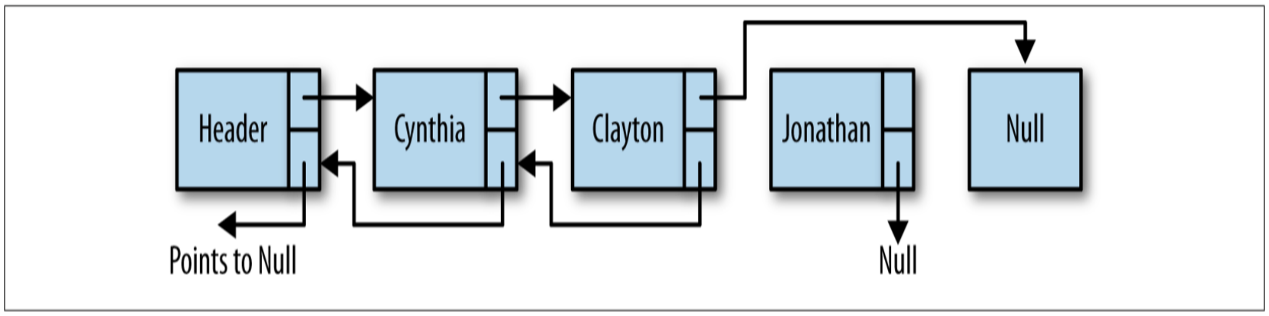
```
class Node {
   constructor(element){
      this.element = element;
      this.next = null;
      this.previous = null;
   }
 }
```

The **insert**() function for a doubly linked list is similar to the **insert**() function for the singly linked list, except that we have to set the new node's previous property to point to the previous node. Here is the definition:

Example:

```
      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**() function for a doubly linked list is more efficient than for a singly linked list because we don't have to find the previous node. We first need to find the node in the list that is storing the data we want to remove. Then we set that node's previous property 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. Next figure makes this situation easier to understand.



Here is the code for the ***remove***() function:

Example:

```
        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;
            }
        }
```

In order to perform tasks such as displaying a linked list in reverse order, we can use a utility function that finds the last node in a doubly linked list. The following function, ***findLast***(), moves us to the last node of a list without going past the end of the list:

Example: *findLast*() method.

```
        findLast() {
          var currNode = this.head;
          while (!(currNode.next == null)) {
             currNode = currNode.next;
          }
          return currNode;
        }
```

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

Example: *dispReverse*() method.

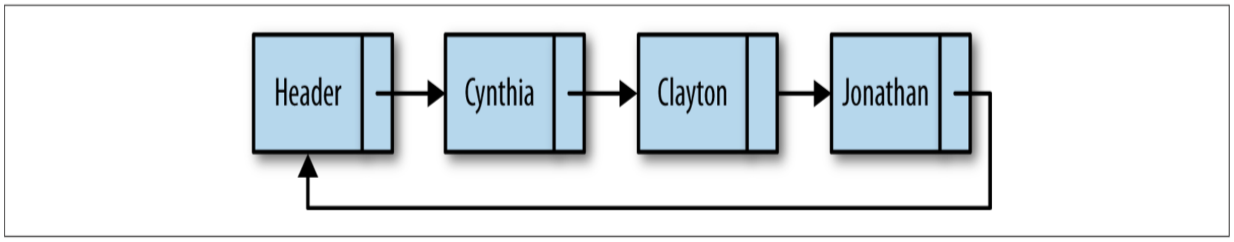
```
        dispReverse() {
          var currNode = this.head;
          currNode = this.findLast();
          while (!(currNode.previous == null)) {
             console.log(currNode.element);
             currNode = currNode.previous;
          }
        }
```

5.4. 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, as shown in the next figure.



The reason you might want to create a circularly linked list is if you want the ability to go backward through a list but don't want the extra overhead of creating a doubly linked list. **You can move backward through a circularly linked list by moving forward through the end of the list to the node you are trying to reach.** To create a circularly linked list, change the constructor function of the *LinkedList* class to read:

Example:

```
class CircularLinkedList {

   constructor(){
      this.head = new Node("head");
      this.head.next = this.head;
   }

   ...
}
```

This is the only change we have to make in order to make a singly linked list into a circularly linked list. However, some of the other linked list functions will not work correctly unmodified. For example, one function that needs to be modified is ***display***(). As written, if the ***display***() function is executed on a circularly linked list, the function will never stop. The condition of the while loop needs to change so that the head element is tested for and the loop will stop when it gets to the head.

Here is how the ***display***() function is written for a circularly linked list:

Example: *display*() method.

```
display() {
   var currNode = this.head;
   while (!(currNode.next == null) &&
          !(currNode.next.element == "head")) {
      console.log(currNode.next.element);
      currNode = currNode.next;
   }
 }
```

5.4.1. Other Linked List methods.

There are several other functions you might include in order to have a well-functioning linked list.

- ***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.