# Lists

Lists are one of the most common organizing tools people use in their day-to-day lives. We have to-do lists, grocery lists, top-ten lists, bottom-ten lists, and many other types of lists. Our computer programs can also make use of lists, especially if we only have a few items to store. Lists are especially useful if we don't have to perform searches on the items in the list or put the items into some type of sorted order. When we need to perform long searches or perform complex sorts, the list becomes less and less useful in relation to more complex data structures.

This chapter presents the implementation of a simple list. We start with a definition of a List ADT and then demonstrate how to implement the ADT. We wrap the chapter up with some problems that are best solved with lists.

Note to reader: this chapter uses the SpiderMonkey JavaScript shell rather than the Node.js shell. Instructions for installing the SpiderMonkey shell can be found in Appendix A.

## A List ADT

In order to design an ADT for a list, we have to provide a definition of a list, including the properties of a list and the operations performed on a list and by a list.

A list is an ordered sequence of data. Each data item stored in a list is called an *element*. In JavaScript, the elements of a list can be of any data type. There is no predetermined number of elements that can be stored in a list, though the practical limit will be constrained by the amount of memory available to the program using the list.

A list with no elements is an *empty* list. The number of elements stored in a list is called the *length* of the list. Internally, the number of elements in a list is kept in a *listSize* variable. You can *append* an element to the end of a list, or you can *insert* an element into a list after an existing element. Elements are deleted from a list using a *remove* operation. You can also *clear* a list so that all the current elements of the list are removed.

The elements of an array are displayed using either a *toString* operation, which displays all the elements of a list at once, or with a *getElement* operation, which displays the value of the *current* element.

Lists have properties to describe location. There is the *front* of a list and the *end* of a list. You can move from one element to the next element using the *next* operation, and you can move backwards through a list using the *prev* operation. You can also move to a numbered position in a list using *moveTo*. The *currPos* property indicates the current position in a list.

The ADT does not specify a storage method for a list, but for our implementation we will use an array named dataStore.

Here is the complete ADT for List:

ADT List:

listSize (property): number of elements in list

pos (property): current position in list

length (method): returns the number of elements in list

clear (method): clears all elements from list

toString (method): returns elements of list

getElement (method): returns value of current position

insert (method): inserts new element after existing element

append (method): adds new element to end of list

remove (method): removes element from list

front (method): sets current position to first element of list

end (method): sets current position to last element of list

prev (method): moves current position backwards one element

next (method): moves current position forward one element

currPos (method): returns the current position

moveTo (method): moves the current position to specified position

## A List Class Implementation

An implementation of a List class can be taken straight from the List ADT we just defined. We'll start with a definition of the constructor function:

function List() {

this.listSize = 0;

this.pos = 0;

this.dataStore = [];

this.clear = clear;

this.find = find;

this.toString = toString;

this.insert = insert;

this.append = append;

this.remove = remove;

this.front = front;

this.end = end;

this.prev = prev;

this.next = next;

this.length = length;

this.currPos = currPos;

this.moveTo = moveTo;

this.getElement = getElement;

this.length = length;

}

The first method we'll implement is the append method. This method appends a new element onto the list at the next available position, which will be equal to the value of the listSize variable:

function append(element) {

this.dataStore[this.listSize++] = element;

}

After the element is appended, the listSize is incremented by one.

Next, let's see how to remove an element from a list. remove is one of the harder methods to implement in the List class. First, we have to find the element in the list and then we have to remove it and adjust the space in the underlying array to fill the hole left by removing an element. To start, let's define a helper method, find, for finding the element to remove:

function find(element) {

for (var i = 0; i < this.dataStore.length; ++i) {

if (this.dataStore[i] == element) {

return i;

}

}

return -1;

}

The find method simply iterates through dataStore looking for the specified element. If the element is found, the method returns the position where the element was found. If the element wasn't found, the method returns -1, which is a standard value to return when an element can't be found in an array. We can use the -1 value for error-checking in our remove method.

The remove method uses the position returned by the find method to begin a process of shifting the elements of dataStore up to fill the hole created by removing an element. This process is illustrated in Figure x.2. After the shifting is finished, listSize is decremented by one to reflect the new size of the list:

function remove(element) {

var foundAt = this.find(element);

if (foundAt > -1) {

for (var i = foundAt; i < this.listSize-1; ++i) {

this.dataStore[i] = this.dataStore[i+1];

}

--this.listSize;

}

}

This is a good time to create a method that allows us to view the elements of a list. Here is the code for a toString method for the class:

function toString() {

var str = "";

for (var i = 0; i < this.listSize; ++i) {

str += this.dataStore[i] + " ";

}

return str;

}

This method loops through the dataStore array, creating a return string by concatenating each list element to each other.

Let's take a break from implementing our List class to see how well it works so far. Here is a short test program that exercises the methods we've created:

var names = new List();

names.append("Cynthia");

names.append("Raymond");

names.append("Barbara");

print(names.toString());

names.remove("Raymond");

print(names.toString());

The output from this program is:

Cynthia Raymond Bryan

Cynthia Bryan

The next method we need to examine is insert. What if, after removing Raymond from the list, we decide we need to put him back in the same place he was in before being removed? An insertion method needs to know where to insert an element, so for now we will say that insertion occurs after a specified element already in the list. With that in mind, here is the definition of insert:

function insert(element, after) {

var insertPos = this.find(after);

for (var i = this.listSize; i > insertPos; --i) {

this.dataStore[i] = this.dataStore[i-1];

}

this.dataStore[insertPos+1] = element;

++this.listSize;

}

insert uses the helper method find to determine the correct insertion position for the new element by finding the element specified as the after argument. Once this position is found, the for loop shifts each element past the element returned by find over one position to make room for the inserted element. After the loop finishes shifting the array elements, the new element is inserted into the array and the listSize variable is incremented by one.

Creating the insert method brings to mind the fact that we might want to insert elements into a list before a specified element. We can do this with an insertBefore method. This method is almost identical to the insert method. The only difference is the inserted element is placed at the position returned by the find method, rather than at the position after the found position. Here is the code:

function insertBefore(element, before) {

var insertPos = this.find(before);

for (var i = this.listSize; i > insertPos; --i) {

this.dataStore[i] = this.dataStore[i-1];

}

this.dataStore[insertPos] = element;

++this.listSize;

}

Next we need to see how to clear out a list so we can put new elements into it:

function clear() {

delete this.dataStore;

this.dataStore = [];

this.listSize = this.pos = 0;

}

The clear method uses the delete operator to delete the dataStore array. The next line recreates dataStore and the last line sets the values of listSize and pos to zero to indicate a new list.

This final set of methods allows movement through the list, and the last method, getElement, displays the current element in the list:

function front() {

this.pos = 0;

}

function end() {

this.pos = this.listSize-1;

}

function prev() {

--this.pos;

}

function next() {

++this.pos;

}

function currPos() {

return this.pos;

}

function moveTo(position) {

this.pos = position;

}

function getElement() {

return this.dataStore[this.pos];

}

Let's create a new list of names to demonstrate how these methods work:

var names = new List();

names.append("Clayton");

names.append("Raymond");

names.append("Cynthia");

names.append("Jennifer");

names.append("Bryan");

names.append("Danny");

Now let's move to the first element and display it:

names.front();

print(names.getElement()); // displays Clayton

Next, we move forward one element and display the element's value:

names.next();

print(names.getElement()); // displays Raymond

Now we'll move forward twice and backwards once, displaying the current element to demonstrate how the prev method works:

names.next();

names.next();

names.prev();

print(names.getElement()); // displays Cynthia

The behavior we've demonstrated is these past few code fragments is captured in the concept of an iterator. We explore iterators in the next section.

## Iterating Through a List

An iterator allows us to traverse a list without referencing the internal storage mechanism of the List class. The methods front, end, prev, next, and currPos provide an implementation of an iterator for our List class. Some advantages to using iterators over using array indexing include:

* Not having to worry about the underlying data storage structure when accessing list elements
* Being able to update the list and not having to update the iterator, where an index becomes invalid when a new element is added to the list
* Providing a uniform means of accessing elements for different types of data stores used in the implementation of a List class

With these advantages in mind, here is how to use an iterator to traverse through a list:

for(names.front(); names.currPos() < names.length(); names.next()) {

print(names.getElement());

}

The for loop starts by setting the current position to the front of the list. The loop continues while the value of currPos is less than the length of the list. Each time through the loop, the current position is moved one element forward through use of the next method.

We can also traverse the list backwards using an iterator. Here's the code:

for(names.end(); names.currPos() >= 0; names.prev()) {

print(names.getElement());

}

This loops starts at the last element of the list and moves backwards using the prev method while the current position is greater than or equal to zero.

## A List-Based Application

To demonstrate how to use lists, we are going to implement most of a system that can be used in the simulation of a video rental kiosk system such as Red Box. Some of the functionality of the system will be implemented in the exercises at the end of the chapter.

### Reading Text Files

In order to get the list of videos available in the kiosk into our program, we have to read the data from a file. We first have to create a text file that contains the list of videos available using any text editor. We name the file: films.txt. Here are the file's contents (these movies are the top 20 movies as voted on by IMDB users):

The Shawshank Redemption

The Godfather

The Godfather: Part II

Pulp Fiction

The Good, the Bad and the Ugly

12 Angry Men

Schindler's List

The Dark Knight

The Lord of the Rings: The Return of the King

Fight Club

Star Wars: Episode V - The Empire Strikes Back

One Flew Over the Cuckoo's Nest

The Lord of the Rings: The Fellowship of the Ring

Inception

Goodfellas

Star Wars

Seven Samurai

The Matrix

Forrest Gump

City of God

Now we need a code fragment to read the contents of the file and store them in our program:

var movies = read('films.txt').split("\n");

This line performs two tasks. First, it reads the contents of our movie text file into the program, read('films.txt'), and second, it splits the file up into individual lines by using the newline character as a delimiter. This output is then stored as an array in the movies variable.

This code fragment works up to a point but is not perfect. When the elements of the text file are split into the array, the newline character is replaced with a space. While a space seems innocuous enough, having an extra space in a string can cause havoc when you are doing string comparisons. So, we need to add a loop that strips the space from each array element using the trim() function. This code will work better in a function, so let's create a function to read data from a file and store it in an array:

function createArr(file) {

var arr = read(file).split("\n");

for (var i = 0; i < arr.length; ++i) {

arr[i] = arr[i].trim();

}

return arr;

}

### Using Lists to Manage a Kiosk

The next step is to take the movies array and store its contents in a list. Here is how we do it:

var movieList = new List();

for (var i = 0; i < movies.length; ++i) {

movieList.append(movies[i]);

}

Now we can write a function to display the movie list available at the kiosk:

function displayList(list) {

for (list.front(); list.currPos() < list.length(); list.next()) {

print(list.getElement());

}

}

Now that we have our movie list taken care of, we need to create a list to store the customers who check out movies:

var customers = new List();

This list will contain customer objects, which include the customer's name and the movie they check out. Here is the constructor for the Customer object:

function Customer(name, movie) {

this.name = name;

this.movie = movie;

}

Next we need a function that allows a customer to check out a movie. This function takes two arguments – the customer's name and the movie they want to check out. If the movie is available, the function removes the movie from the kiosk's list of movies and adds it to the customer's list.

Specifying how the function for checking out movies is supposed to work indicates that the List class is missing a necessary method. We need a way to check to see if an element is in a list, returning a true value if the element is in the list, and a false value otherwise. We'll call the method contains, and here is the definition:

function contains(element) {

for (var i = 0; i < this.dataStore.length; ++i) {

if (this.dataStore[i] == element) {

return true;

}

}

return false;

}

Be sure to include the declaration for this new method in the List class constructor:

this.contains = contains;

Now we are ready to write a method to check out a movie. Here is the definition:

function checkOut(name, movie, filmList, customerList) {

if (movieList.contains(movie)) {

var c = new Customer(name, movie);

customerList.append(c);

filmList.remove(movie);

}

else {

print(movie + " is not available.");

}

}

The method first checks to see if the movie being requested is available. If the movie is available, a Customer object is created with the movie's title and the customer name, the Customer object is appended to the customer list, and the movie is removed from the movie list. If the movie is not available, a simple message is displayed indicating this fact.

Let's test the checkOut method with a short program:

The output of the program displays the movie list with The Godfather removed, but when the customer list is displayed, the program shows:

[object Object]

Clearly, our displayList() method does not work with objects. To fix it, we have to test the first element of the list to see if it is an object, in this case a Customer object. If so, we have to retrieve the properties of the object and use them to retrieve the appropriate value for each property. The following code fragment does the trick:

if (list.getElement() instanceof Customer) {

print(list.getElement()["name"] + ", " +

list.getElement()["movie"]);

}

else {

print(list.getElement());

}

The if statement tests to see if the current element of the list is a Customer object. If it is, we display the values of the current Customer object using the object's properties. If the current element is not a Customer object, we just display the current element. Here is the new definition for displayList():

function displayList(list) {

for (list.front(); list.currPos() < list.length(); list.next()) {

if (list.getElement() instanceof Customer) {

print(list.getElement()["name"] + ", " +

list.getElement()["movie"]);

}

else {

print(list.getElement());

}

}

}

Let's add some titles to our program's output to make it easier to read and some interactive input:

var movies = createArr("films.txt");

var movieList = new List();

var customers = new List();

for (var i = 0; i < movies.length; ++i) {

movieList.append(movies[i]);

}

print("Available movies: \n");

displayList(movieList);

putstr("\nEnter your name: ");

var name = readline();

putstr("What movie would you like? ");

var movie = readline();

checkOut(name, movie, movieList, customers);

print("\nCustomer Rentals: \n");

displayList(customers);

print("\nMovies Now Available\n");

displayList(movieList);

Here is the result of running this new program:

Available movies:

The Shawshank Redemption

The Godfather

The Godfather: Part II

Pulp Fiction

The Good, the Bad and the Ugly

12 Angry Men

Schindler's List

The Dark Knight

The Lord of the Rings: The Return of the King

Fight Club

Star Wars: Episode V - The Empire Strikes Back

One Flew Over the Cuckoo's Nest

The Lord of the Rings: The Fellowship of the Ring

Inception

Goodfellas

Star Wars

Seven Samurai

The Matrix

Forrest Gump

City of God

Enter your name: Jane Doe

What movie would you like? The Godfather

Customer Rentals:

Jane Doe, The Godfather

Movies Now Available

The Shawshank Redemption

The Godfather: Part II

Pulp Fiction

The Good, the Bad and the Ugly

12 Angry Men

Schindler's List

The Dark Knight

The Lord of the Rings: The Return of the King

Fight Club

Star Wars: Episode V - The Empire Strikes Back

One Flew Over the Cuckoo's Nest

The Lord of the Rings: The Fellowship of the Ring

Inception

Goodfellas

Star Wars

Seven Samurai

The Matrix

Forrest Gump

City of God

There is other functionality we can add to make our video rental kiosk system more robust. You will get the opportunity to explore this added functionality in the exercises below.

## Exercises

1. Write a method that inserts an element into a list only if the element to be inserted is larger than any of the elements currently in the list.
2. Write a method that inserts an element into a list only if the element to be inserted is smaller than any of the elements currently in the list.
3. Create a Person object that stores a person's name and their gender. Create a list of at least ten Person objects. Write a method that displays all the people in the list of the same gender (displayByGender(gender)).
4. Modify the video rental kiosk program so that when a movie is checked out it is added to a list of rented movies.
5. Create a check-in method for the video rental kiosk program so that the returned movie is deleted from the rented movies list and is added back to the available movies list.
6. Create a method for the video rental kiosk program that allows the user to view their own list of rented movies.
7. Using the methods created in the chapter, as well as the methods created in the above exercises, create an application that allows a customer to use a video rental kiosk. The customer must be able to see a list of the movies available, check out a movie, return a movie, and see what movies they have checked out.