# Stacks

Data organize naturally as lists. We have already used the array and the associative array for handling data organized as a list. Although those data structures helped us group data in a convenient form for processing, neither structure provides a real abstraction for actually designing and implementing problem solutions.

One list-oriented data structure that provides an easy-to-understand abstraction is the *stack*. Data organized into a stack are added and removed from one end of a list. Stacks are used extensively in programming language implementations for everything from expression evaluation to handling function calls.

## Stack Operations

A stack is a list of elements that are accessible only from one end of the list, which is called the *top*. One common real-world analogy for a stack is the stack of trays at a cafeteria. Trays are always removed from the top, and when trays are put back on the stack, they are placed on the top of the stack. The stack is known as a last-in, first-out (LIFO) data structure.

The two primary operations of a stack are adding elements to the stack and taking elements off of the stack. Elements are added to a stack using the *push* operation. Elements are taken off a stack using the *pop* operation. These operations are illustrated in Figure x.1.

The other primary operation performed on a stack is viewing the top element. The pop operation visits the top element of a stack, but it also removes the element permanently from the stack. Sometimes we want to just know what the top element is without removing it. The *peek* operation is designed to just return the value of the top element of a stack without removing it.

To keep track of where the top element is, as well as where to add a new element, we use a *top* variable that is incremented when we push new elements onto the stack and is decremented when we pop elements off the stack.

While pushing, popping, and peeking are the primary operations associated with a stack, there are other operations we need to perform and properties we need to examine. The *clear* operation removes all the elements from a stack. The number of elements in a stack can be retrieved using the *length* property. We can also define an *empty* property that lets us know if a stack has any elements stored in it, although we will actually just use the length property for this purpose.

## A Stack Implementation

To create a stack implementation, we first need to decide the underlying data structure we will use to store the stack elements. The first data structure to consider is the array. In many languages, this choice would not be possible because the absolute size of an array must be decided when the array is declared. With JavaScript, as we know, this is not the case and arrays can grow beyond their original declaration. Our choice, then, for the data structure to store stack elements is the array.

We'll begin our stack implementation by defining the constructor function for a Stack class:

function Stack() {

this.dataStore = [];

this.top = 0;

this.push = push;

this.pop = pop;

this.peek = peek;

}

The array that stores the stack elements is named dataStore. The constructor sets it to an empty array. The top variable keeps track of the top of the stack and is initially set to 0 by the constructor, indicating that, for now, the 0th position of the array is the top of the stack.

The first method to implement is the *push* method. When we push a new element on the stack we have to store it in the top position and increment the top variable so that the new top is the next position in the array. Here is the code:

function push(element) {

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

}

Pay particular attention to the placement of the increment operator **after** the call to this.top. Placing the operator there ensures that the current value of top is used to place the new element at the top of the stack before it is incremented.

The pop method does the reverse of the push method – it returns the element in the top position of the stack and then it decrements the top variable:

function pop() {

return this.dataStore[--this.top];

}

The peek method returns the top element of the stack by accessing the element at the top-1 position of the array:

function peek() {

return this.dataStore[this.top-1];

}

There will be situations when you need to know how many elements are stored in a stack. The length() method returns this value by returning the value of top:

function length() {

return this.top;

}

Finally, we can clear a stack in order to start over storing elements in it by simply setting the top variable back to 0:

function clear() {

this.top = 0;

}

Now that we have our Stack class defined, we can test it out to see how well it works:

var s = new Stack();

s.push("David");

s.push("Raymond");

s.push("Bryan");

console.log("length: " + s.length());

console.log(s.peek());

var popped = s.pop();

console.log("The popped element is: " + popped);

console.log(s.peek());

s.push("Cynthia");

console.log(s.peek());

s.clear();

console.log("length: " + s.length());

console.log(s.peek());

s.push("Clayton");

console.log(s.peek());

The output from this program is:

length: 3

Bryan

The popped element is: Bryan

Raymond

Cynthia

length: 0

undefined

Clayton

The next-to-last value, undefined, is returned because once a stack is cleared, there is no value in the top position, so when we peek at the top of the stack, undefined is returned.

Here is the complete implementation of the Stack class:

function Stack() {

this.dataStore = [];

this.top = 0;

this.push = push;

this.pop = pop;

this.peek = peek;

this.clear = clear;

this.length = length;

}

function push(element) {

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

}

function peek() {

return this.dataStore[this.top-1];

}

function pop() {

return this.dataStore[--this.top];

}

function clear() {

this.top = 0;

}

function length() {

return this.top;

}

## Using the Stack Class

There are several problems for which a stack is the perfect data structure needed for the solution. In this section, we look at several such problems.

### Multiple Base Conversions

A stack can be used to convert a number from one base to another base. Given a number**, *N***, which we want to convert to a base, *B*, here is the algorithm for performing the conversion:

1. The right-most digit of *N* is *N* % *B*. Push the digit onto the stack.
2. Replace *N* with *N* / *B*.
3. Repeat steps 1 and 2 until *N* = 0 and there are no significant digits remaining.
4. Build the converted number string by popping the stack until the stack is empty.

We can implement this algorithm very simply using a stack in JavaScript. Here is a function definition for converting a number to any base 2 through 9:

function mulBase(num, base) {

var s = new Stack();

do {

s.push(num % base);

num = Math.floor(num /= base);

} while (num > 0);

var converted = "";

while (s.length() > 0) {

converted += s.pop();

}

return converted;

}

Following is a short program to test the function for base 2 and base 8 conversions:

var num = 32;

var base = 2;

var newNum = mulBase(num, base);

console.log(num + " converted to base " + base + " is " + newNum);

num = 125;

base = 8;

var newNum = mulBase(num, base);

console.log(num + " converted to base " + base + " is " + newNum);

The output from this program is:

32 converted to base 2 is 100000

125 converted to base 8 is 175

### Word Palindromes

A word palindrome is a word that is spelled the same forward and backward. For example, "dad" is a palindrome; "racecar" is a palindrome; but "hello" is not a palindrome.

We can use a stack to determine whether or not a given word is a palindrome. We take the original word and push each letter onto a stack, moving from left to right. When we get to the end of the original word, the stack contains the original word in reverse order, with the last letter at the top of the stack and the first letter at the bottom of the stack, as shown in Figure x.2.

Once the complete original word is on the stack, we create a new word using the stack by popping off each letter from the stack. This process will create the original word in reverse order. We then simply compare the original word with the reversed word and if they are equal, the word is a palindrome.

Here is a program, minus the Stack class code, that determines if a word is a palindrome. We create a function, isPalindrome(word), that uses the Stack class:

function isPalindrome(word) {

var s = new Stack();

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

s.push(word[i]);

}

var rword = "";

while (s.length() > 0) {

rword += s.pop();

}

if (word == rword) {

return true;

}

else {

return false;

}

}

var word = "hello";

if (isPalindrome(word)) {

console.log(word + " is a palindrome.");

}

else {

console.log(word + " is not a palindrome.");

}

### Demonstrating Recursion

Stacks are often used in the implementation of computer programming languages. One area where stacks are used is the implementation of recursion. It is beyond the scope of this book to demonstrate exactly how stacks are used to implement recursive procedures, but we can simulate a recursive process.

To demonstrate how recursion is implemented using a stack, let’s consider a recursive definition of the factorial function. First, here is an informal definition of factorial for the number 5:

5! = 5 \* 4 \* 3 \* 2 \* 1 = 120

Here is a recursive function to compute the factorial of a number:

function factorial(n) {

if (n === 0) {

return 1;

}

else {

return n \* factorial(n-1);

}

}

When applied with the argument 5, it returns 120.

To simulate computing *5!* with a stack, we first push the numbers 5 through 1 onto a stack. Then, inside a loop, we pop each number and multiply the number by a running product, resulting in the factorial of 5. Here is the code for the function:

function fact(n) {

var s = new Stack();

while (n > 1) {

s.push(n--);

}

var product = 1;

while (s.length() > 0) {

product \*= s.pop();

}

return product;

}

The following program demonstrates that both functions compute the same value for 5!:

console.log(factorial(5)); // displays 120

console.log(fact(5)); // displays 120

## Exercises

1. A stack can be used to ensure that an arithmetic expression has balanced parentheses. Write a function that takes an arithmetic expression as an argument and returns the position in the equation where a parenthesis is missing.
2. A postfix expression evaluator works on arithmetic expressions taking the following form:

*op1 op2 operator*

Using two stacks, one for the operands and one for the operands, design and implement a JavaScript function that converts infix expressions to postfix expressions, and then use the stacks to evaluate the expression.