# Queues

A *queue* is a type of list where data are inserted at the end of the list and are removed from the front of the list. Queues are used to store data in the order in which they occur. A queue is an example of a first-in, first-out (FIFO) data structure. Queues are used to order processes submitted to an operating system or a print spooler, and simulation applications use queues to model scenarios such as customers standing in the line at a bank or a grocery store.

## Queue Operations

The two primary operations involving queues are inserting a new element into the queue and removing an element from the queue. The operation for inserting a new element is called *enqueue*, and the operation for removing an element is called *dequeue*. The enqueue operation inserts a new element at the end of the queue, and the dequeue operation removes an element from the front of the queue. Figure x.1 illustrates these operations.

Another necessary queue operation is viewing the element at the front of the queue. This operation is called *peek*, just as it is for stacks. The peek operation returns the element stored at the front of the queue without removing it from the queue. Besides examining the front element, we also want to know how many elements are stored in a queue, which we can satisfy with a *length* operation, and we need to be able to remove all the elements of a queue, which is a *clear* operation.

## An Array-Based Queue Class Implementation

Implementing the Queue class using an array is a straightforward operation. Using JavaScript arrays is an advantage that other languages don't have because the implementation of JavaScript arrays includes methods that make it easy to add data to the end of the array, push(), and easy to take away data from the front of the array, shift().

The push() array method places its argument at the first open position of an array, which will be the back of the array at all times, even when there are no other elements in the array. Here is an example:

names = []

name.push("Cynthia");

names.push("Jennifer");

print(names);

which displays:

Cynthia,Jennifer

Then we can remove the element from the front of the array using the shift() method:

names.shift();

print(names);

leaving:

Jennifer

Now let's begin our implementation of the Queue class with the constructor function:

function Queue() {

this.dataStore = [];

this.enqueue = enqueue;

this.dequeue = dequeue;

this.front = front;

this.back = back;

this.toString = toString;

this.empty = empty;

}

The enqueue() method adds an element to the end of a queue:

function enqueue(element) {

this.dataStore.push(element);

}

The dequeue() method removes an element from the front of a queue:

function dequeue() {

return this.dataStore.shift();

}

We can examine the front and back elements of a queue using these methods:

function front() {

return this.dataStore[0];

}

function back() {

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

}

We also need a toString() method to display all the elements in a queue:

function toString() {

var retStr = "";

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

retStr += this.dataStore[i] + "\n";

}

return retStr;

}

Finally, we need a method that lets us know when the queue is empty:

function empty() {

if (this.dataStore.length == 0) {

return true;

}

else {

return false;

}

}

Here is a program to test our implementation:

var q = new Queue();

q.enqueue("Meredith");

q.enqueue("Cynthia");

q.enqueue("Jennifer");

print(q.toString());

q.dequeue("Meredith");

print(q.toString());

print("Front of queue: " + q.front());

print("Back of queue: " + q.back());

The output from this program is:

Meredith

Cynthia

Jennifer

Cynthia

Jennifer

Front of queue: Cynthia

Back of queue: Jennifer

## Using Queues: Assigning Partners at a Square Dance

As we mentioned earlier, queues are often used to simulate situations when people have to wait in line. One scenario we can simulate with a queue is a square dance for singles. When men and women arrive at this square dance, they enter the dance hall and stand in the line for their gender. The dance floor is quite small and there is room for only a few dancing partners at a time. As room becomes available on the dance floor, dance partners are chosen by taking the first man and woman in line. The next man and woman moves to the front of their respective line. As dance partners move onto the dance floor, their names are announced. If a couple leaves the floor and there is not a man and a woman at the front of the line, this fact is announced.

This simulation will store the names of the men and women participating in the square dance in a text file. Here is the data we will use for the simulation:

F Allison McMillan

M Frank Opitz

M Mason McMillan

M Clayton Ruff

F Cheryl Ferenback

M Raymond Williams

F Jennifer Ingram

M Bryan Frazer

M David Durr

M Danny Martin

F Aurora Adney

Each dancer is stored in a Dancer object:

function Dancer(name, sex) {

this.name = name;

this.sex = sex;

}

We will write a function to load the dancers from the file to the program:

function getDancers(males, females) {

var names = read("dancers.txt").split("\n");

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

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

}

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

var dancer = names[i].split(" ");

var sex = dancer[0];

var name = dancer[1];

if (sex == "F") {

femaleDancers.enqueue(new Dancer(name, sex));

}

else {

maleDancers.enqueue(new Dancer(name, sex));

}

}

}

The names are read from the file into an array. Then the function trims the newline character from each line. The second loop splits each line into a two-element array, by sex and name. Then the function examines the sex element and assigns the dancer to the appropriate queue.

The next function pairs up the male and female dancers and announces the pairings:

function dance(males, females) {

print("The dance partners are: \n");

while (!females.empty() && !males.empty()) {

person = females.dequeue();

putstr("Female dancer is: " + person.name);

person = males.dequeue();

print(" and the male dancer is: " + person.name);

}

print();

}

Here is the main program:

// main program

var maleDancers = new Queue();

var femaleDancers = new Queue();

getDancers(maleDancers, femaleDancers);

dance(maleDancers, femaleDancers);

if (!femaleDancers.empty()) {

print(femaleDancers.front().name + " is waiting to dance.");

}

if (!maleDancers.empty()) {

print(maleDancers.front().name + " is waiting to dance.");

}

Here is the output from the program:

The dance partners are:

Female dancer is: Allison and the male dancer is: Frank

Female dancer is: Cheryl and the male dancer is: Mason

Female dancer is: Jennifer and the male dancer is: Clayton

Female dancer is: Aurora and the male dancer is: Raymond

Bryan is waiting to dance.

One change we might want to make to the program is display the number of male and female dancers waiting to dance. We don't have a method that displays the number of elements in a queue, so we need to add it to the Queue class definition:

function count() {

return this.dataStore.length;

}

Be sure to add the following line to the constructor function:

this.count = count;

We can change the main program to this:

// main program

var maleDancers = new Queue();

var femaleDancers = new Queue();

getDancers(maleDancers, femaleDancers);

dance(maleDancers, femaleDancers);

if (maleDancers.count() > 0) {

print("There are " + maleDancers.count() +

" male dancers waiting to dance.");

}

if (femaleDancers.count() > 0) {

print("There are " + femaleDancers.count() +

" female dancers waiting to dance.");

}

The output from this program is:

The dance partners are:

Female dancer is: Allison and the male dancer is: Frank

Female dancer is: Cheryl and the male dancer is: Mason

Female dancer is: Jennifer and the male dancer is: Clayton

Female dancer is: Aurora and the male dancer is: Raymond

There are 3 male dancers waiting to dance.

## Sorting Data with Queues

Queues are not only useful for simulations; they can also be used to sort data. Here is one example (taken from Ford and Topp, 1996).

Back in the old days of computing, programs were entered into a mainframe computer via punch cards, with each card holding a single program statement. The cards were sorted using a mechanical sorter that utilized bin-like structures. We can simulate this process with the use of a set of queues. This sorting technique is called a *radix sort*. It is not the fastest of sorting algorithms, but it does demonstrate an interesting use of queues.

The radix sort works by making two passes over a data set, in this case the set of integers from 0-99. The first pass sorts the numbers based on the 1's digit and the second pass sorts the numbers based on the 10's digit. Each number is placed in a bin based on the digit in each of these two places. Given the numbers:

91, 46, 85, 15, 92, 35, 31, 22

the first pass results in the following bin configuration:

Bin 0:

Bin 1: 91, 31

Bin 2: 92, 22

Bin 3:

Bin 4:

Bin 5: 85, 15, 35

Bin 6: 46

Bin 7:

Bin 8:

Bin 9:

Now the numbers are sorted based on which bin they are in:

91, 31, 92, 22, 85, 15, 35, 46

Next, the list is sorted by the 10's digit into the appropriate bins:

Bin 0:

Bin 1: 15

Bin 2: 22

Bin 3: 31, 35

Bin 4: 46

Bin 5:

Bin 6:

Bin 7:

Bin 8: 85

Bin 9: 91, 92

Finally, take the numbers from the bins and put them back into a list, which results in a sorted set of integers:

15, 22, 31, 35, 46, 85, 91, 92

We can implement this algorithm by using queues to represent the bins. We need nine queues, one for each digit. We will store the queues in an array. We use modulus and integer division for determining the 1's and 10's digits. The remainder of the algorithm entails adding numbers to their appropriate queues, taking the numbers out of the queues to resort them based on the 1's digit, and then repeating the process for the 10's digit. The result is a sorted set of integers.

First, here is the function that distributes numbers by the place digit:

function distribute(nums, queues, n, digit) {

for (var i = 0; i < n; ++i) {

if (digit == 1) {

queues[nums[i]%10].enqueue(nums[i]);

}

else {

queues[Math.floor(nums[i] / 10)].enqueue(nums[i]);

}

}

}

Here is the function for collecting the numbers from the queues:

function collect(queues, nums) {

var i = 0;

for (var digit = 0; digit < 10; ++digit) {

while (!queues[digit].empty()) {

nums[i++] = queues[digit].dequeue();

}

}

}

Here is the main program, and a function for displaying the contents of the array:

function dispArray(arr) {

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

putstr(arr[i] + " ");

}

}

// main program

var queues = [];

for (var i = 0; i < 10; ++i) {

queues[i] = new Queue();

}

var nums = [];

for (var i = 0; i < 10; ++i) {

nums[i] = Math.floor(Math.floor(Math.random() \* 101));

}

print("Before radix sort: ");

dispArray(nums);

distribute(nums, queues, 10, 1);

collect(queues, nums);

distribute(nums, queues, 10, 10);

collect(queues, nums);

print("\n\nAfter radix sort: ");

dispArray(nums);

Here are a couple of runs of the program:

Before radix sort:

45 72 93 51 21 16 70 41 27 31

After radix sort:

16 21 27 31 41 45 51 70 72 93

Before radix sort:

76 77 15 84 79 71 69 99 6 54

After radix sort:

6 15 54 69 71 76 77 79 84 99

## Priority Queues

In the course of normal operations, when an element is removed from a queue that element is always the first element inserted into the queue. There are certain applications of queues, however, that require that elements be removed in an order other than first-in, first-out. When we need to simulate such an application, we need to create a data structure called a *priority queue*.

A priority queue is a queue where elements are removed from the queue based on a priority constraint. For example, the waiting room at a hospital's Emergency Department operates using a priority queue. When a patient enters the Emergency Department, he or she is seen by a triage nurse. This nurse's job is to access the emergence of the patient's condition and assign the patient a priority code. Patients with a high priority code are seen before patients with a lower priority code, and patients that have the same priority code are seen on a first-come, first-served, or first-in, first-out, basis.

To model the operations of a priority queue system, we can use our Queue class with one modification. The dequeue() method must be changed so that it is able to determine the priority of the elements in the queue and remove the element with the highest priority. For this to work, we have to modify the data being stored in the queue so that it contains a variable that holds a priority code.

Let's begin building a priority queue system by first defining an object that will store the elements of the queue:

function Patient(name, code) {

this.name = name;

this.code = code;

}

The value for code will be an integer.

Now we need to define a dequeue() method that removes the element in the queue with the highest priority. We will define the highest priority element as being the element with the lowest code. The dequeue() method will move through the queue's underlying array and find the element with the lowest code. Then the method uses the splice() method to remove the highest priority element. Here is the definition for the method:

function dequeue() {

var priority = this.dataStore[0].code;

for (var i = 1; i < this.dataStore.length; ++i) {

if (this.dataStore[i].code < priority) {

priority = i;

}

}

return this.dataStore.splice(priority,1);

}

The dequeue() method uses a simple "find the minimum value" algorithm to find the element with the lowest (hence the highest priority) priority code. The method returns an array of one element – the element removed from the queue.

The toString() method is also modified to display the more complex Patient object:

function toString() {

var retStr = "";

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

retStr += this.dataStore[i].name + " code: "

+ this.dataStore[i].code + "\n";

}

return retStr;

}

Here is a short program that demonstrates how the priority queue system works:

var p = new Patient("Smith",5);

var ed = new Queue();

ed.enqueue(p);

p = new Patient("Jones", 4);

ed.enqueue(p);

p = new Patient("Fehrenbach", 6);

ed.enqueue(p);

p = new Patient("Brown", 1);

ed.enqueue(p);

print(ed.toString());

var seen = ed.dequeue();

print("Patient being treated: " + seen[0].name);

var seen = ed.dequeue();

print("Patient being treated: " + seen[0].name);

print("Patients waiting to be seen: ")

print(ed.toString());

The output from this program is:

Smith code: 5

Jones code: 4

Fehrenbach code: 6

Brown code: 1

Patient being treated: Jones

Patient being treated: Brown

Patients waiting to be seen:

Smith code: 5

Fehrenbach code: 6

## Exercises

1. Modify the Queue class to create a Deque class. A deque is a queue-like structure which allows elements to be added and removed from both the front and the back of the list. Test your class in a program.
2. Use the Deque class above to determine if a given word is a palindrome.
3. Modify the priority queue example in the chapter so that the higher priority elements have higher numbers rather than lower numbers. Test your implementation with the example in the chapter.
4. Modify the Emergency Department example from the chapter so that the user can control the activity in the ED. Create a menu system that allows the user to choose from the following activities: 1) Patient enters ED; 2) Patient is seen by doctor; 3) Display list of patients waiting to be seen.