04. DATASTRUCTS Queue.

A **queue** is a type of list where **data are inserted at the end and are removed from the front.** Queues are used to store data in the order in which they occur, as opposed to a stack, in which the last piece of data entered is the first element used for processing. Think of a queue like the line at your bank, where the first person into the line is the first person served, and as more customers enter a line, they wait in the back until it is their turn to be served.

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.

4.1. Queue Operations

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

Another important queue operation is **viewing the element at the front of a queue**. This operation is called **peek**. The peek operation returns the element stored at the front of a queue without removing it from the queue.

Besides examining the front element, we also need to know how many elements are stored in a queue, which we can satisfy with the **length** property; and we need to be able to **remove all the elements from a queue**, which is performed with the **clear** operation.

length *(property)*	Returns the number of elements in the stack
clear *(function)*	Clears all elements from stack
enqueue *(function)*	Inserts new element at the top of the stack
dequeue *(function)*	Removes the element at the top of the stack
peek *(function)*	Returns the value stored at the top of the stack

4.2. An Array-Based Queue Class Implementation

Implementing the Queue class using an array is straightforward. Using JavaScript arrays is an advantage many other programming languages don't have because JavaScript contains a function for easily adding data to the end of an array, **push**(), and a function for easily removing data from the front of an array, **shift**(). The *push*() function places its argument at the first open position of an array, which will always be the back of the array, even when there are no other elements in the array.

Example: Here is an example:

```
names = [];

name.push("Cynthia");
names.push("Jennifer");

console.log(names); // displays Cynthia,Jennifer
```

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

Example: Here is an example:

```
names.shift();

console.log(names); // displays Jennifer
```

4.2.1. Queue Constructor.

Now we're ready to begin implementing the **Queue** class by defining the constructor function:

Example: Queue *constructor*() method.

```
class Queue {

   constructor(){
     this.dataStore = [];
   }

...

}
```

4.2.2. Enqueue: Add an Element to the Queue.

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

Example: *enqueue*() method.

```
enqueue(element) {
  this.dataStore.push(element);
}
```

4.2.3. Dequeue: Remove an Element form the Queue.

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

Example: *dequeue*() method.

```
dequeue() {

    return this.dataStore.shift();
}
```

4.2.4. Get first and last elements in the Queue.

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

Example: *front*() and *back*() methods.

```
front() {
   return this.dataStore[0];
 }

back() {
   return this.dataStore[this.dataStore.length-1];
 }
```

4.2.5. Display all elements in the Queue.

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

Example: *toString*() method.

```
toString() {
  var retStr = "";
  for (var i = 0; i < this.dataStore.length; ++i) {
     retStr += this.dataStore[i] + "\n";
  }
  return retStr;
}
```

4.2.6. Check if the Queue is empty.

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

Example: *empty*() method.

```
empty() {
  return (this.dataStore.length==0);
}
```

4.2.7. Full Queue Class Implementation

The following code presents the complete **Queue** class definition along with a test program.

Example: The *Queue* class.

```
class Queue {

   constructor(){
       this.dataStore = [];
   }

   enqueue(element) {
       this.dataStore.push(element);
   }

   dequeue() {
    return this.dataStore.shift();
   }
```

```
        front() {
           return this.dataStore[0];
        }

        back() {
           return this.dataStore[this.dataStore.length-1];
        }

        toString() {
           var retStr = "";
           for (var i = 0; i < this.dataStore.length; ++i) {
              retStr += this.dataStore[i] + "\n";
           }
           return retStr;
        }

        empty() {
           return (this.dataStore.length==0);
        }

    }
```

4.2.8.Testing Queue class implementation.

The following code demonstrates a program that tests this implementation.

Example: Testing the *Queue* class implementation.

```
    // test program

    var q = new Queue();

    q.enqueue("Meredith");
    q.enqueue("Cynthia");
    q.enqueue("Jennifer");
    console.log(q.toString());

    q.dequeue();
    console.log(q.toString());
    console.log("Front of queue: " + q.front());
    console.log("Back of queue: " + q.back());
```

The output from previous code is:

Output:

```
Meredith
Cynthia
Jennifer

Cynthia
Jennifer

Front of queue: Cynthia
Back of queue: Jennifer
```

4.3. Using the Queue Class

As we mentioned earlier, **queues are often used to simulate situations when people have to wait in line.** Once 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. 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 move to the front of their respective lines. As dance partners move onto the dance floor, their names are announced. If a couple leaves the floor and there is not both a man and a woman at the front of each line, this fact is announced.

4.3.1. Assigning Partners at a Square Dance

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

Each dancer is stored in a ***Dancer*** object:

Example:

```
Class Dancer {
   constructor(name, sex){
      this.namer = name;
      this.sex = sex;
   }
 }
```

Next we need a function to load the dancers from the file into the program:

Example:

```
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") {
         females.enqueue(new Dancer(name, sex));
      }
      else {
         males.enqueue(new Dancer(name, sex));
      }
   }
 }
```

The names are read from the text file into an array. The function then trims the newline character from each line. The second loop splits each line into a two-element array, by sex and by 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:

Example:

```
function dance(males, females) {
   console.log("The dance partners are: \n");
   while (!females.empty() && !males.empty()) {
      person = females.dequeue();
      console.log("Female dancer is: " + person.name);
      person = males.dequeue();
      console.log(" and the male dancer is: " + person.name);
   }
 }
```

Let's code a square dance simulation:

Example:

```
// test program

var maleDancers = new Queue();
var femaleDancers = new Queue();

getDancers(maleDancers, femaleDancers);
dance(maleDancers, femaleDancers);

if (!femaleDancers.empty()) {
   console.log(femaleDancers.front().name + " is waiting to
dance.");
}

if (!maleDancers.empty()) {
   console.log(maleDancers.front().name + " is waiting to
dance.");
}
```

The output of the previous square dance simulation is:

Output:

```
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 to display the number of male and female dancers waiting to dance. We don't have a function that displays the number of elements in a queue, so we need to add it to the *Queue* class definition:

Example:

```
count() {
   return this.dataStore.length;
}
```

Let's change the test program to use this new function.

Example:

```
var maleDancers = new Queue();
var femaleDancers = new Queue();

getDancers(maleDancers, femaleDancers);
dance(maleDancers, femaleDancers);

if (maleDancers.count() > 0) {
   console.log("There are " + maleDancers.count() +
               " male dancers waiting to dance.");
}

if (femaleDancers.count() > 0) {
   console.log("There are " + femaleDancers.count() +
               " female dancers waiting to dance.");
}
```

The output of the previous square dance simulation is:

Output:

```
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.
```

4.3.2. Sorting Data with Queues

Queues are not only useful for simulations; they can also be used to sort data. Back in the old days of computing, programs were entered into a mainframe program via punch cards, with each card holding a single program statement. The cards were sorted using a mechanical sorter that utilized bin-like structures to hold the cards. We can simulate this process by using 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 to 99. The first pass sorts the numbers based on the 1s digit, and the second pass sorts the numbers based on the 10s digit. Each number is placed in a bin based on the digit in each of these two places. Given these numbers:

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

The first pass of the radix sort 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 numbers are sorted by the 10s 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 out of the bins and put them back into a list, and you get the following sorted list 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 uses the modulus and integer division operations for determining the 1s and 10s digits. The remainder of the algorithm entails adding numbers to their appropriate queues, taking the numbers out of the queues to re-sort them based on the 1s digit, and the repeating the process for the 10s digit. The result is a sorted set of integers.

First, here is the function that distributes numbers by the place (1s or 10s) digit:

Example:

```
function distribute(nums, queues, n, digit) {
 // digit represents either the 1s or 10s place
```

```
        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]);
            }
        }
    }
```

Example: Here is the function for collecting 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();
        }
    }
}
```

Example: Let's use a function for displaying the contents of an array:

```
function dispArray(arr) {
    for (var i = 0; i < arr.length; ++i) {
        putstr(arr[i] + " ");
    }
}
```

Now let's perform a radix sort.

Example:

```
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));
}
console.log("Before radix sort: ");
dispArray(nums);

distribute(nums, queues, 10, 1);
collect(queues, nums);

distribute(nums, queues, 10, 10);
collect(queues, nums);

console.log("\n\nAfter radix sort: ");
dispArray(nums);
```

Here are a couple of runs of the program:

Output:

```
   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
```

4.3.3. Demonstrating Recursion.

In the course of normal queue operations, **when an element is removed from a queue, that element is always the first element that was 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 one where elements are removed from the queue based on a priority constraint. For example, the waiting room at a hospital's emergency department (ED) operates using a priority queue. When a patient enters the ED, he or she is seen by a triage nurse. This nurse's job is to assess the severity of the patient's condition and assign the patient a priorty 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.

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

Example: *Patient* class

```
class Patient {
   constructor(name, code){
      this.name = name;
      this.code = code;
   }
}
```

The value for code will be an integer that represents the patient's priority, or severity.

Now we need to redefine the **dequeue**() function 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 priority code. This new **dequeue**() function will move through the queue's underlying array and find

the element with the lowest code. Then the function uses the **splice**() function to remove the highest-priority element. Here is the new definition for **dequeue**():

Example: *dequeue*() method with priority.

```
dequeue() {

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

   for (var i = 1; i < this.dataStore.length; ++i) {
      if (this.dataStore[i].code < priority) {
         priority = this.dataStore[i].code;
         index = i;
      }
   }
   return this.dataStore.splice(index,1);
}
```

The **dequeue**() function uses a simple sequential search to find the element with the highest priority code (*the lowest number; 1 has a higher priority than 5*). The function returns an array of one element —the one removed from the queue. Finally, we add a **toString**() function modified to handle *Patient* objects:

Example: *toString*() method in a Queue with priority.

```
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;
}
```

The following code demonstrates how the priority queue system works.

Example: A priority queue implementation.

```
var ed = new Queue();

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

p = new Patient("Ingram", 1);
ed.enqueue(p);

console.log(ed.toString());

var seen = ed.dequeue();
console.log("Patient being treated: " + seen[0].name);
console.log("Patients waiting to be seen: ");
console.log(ed.toString());

// another round

var seen = ed.dequeue();
console.log("Patient being treated: " + seen[0].name);
console.log("Patients waiting to be seen: ");
console.log(ed.toString());
```

The previous code generates the following output:

Output:

```
Smith code: 5
Jones code: 4
Fehrenbach code: 6
Brown code: 1
Ingram code: 1


Patient being treated: Ingram
Patients waiting to be seen:
Smith code: 5
```

```
Jones code: 4
Fehrenbach code: 6
Brown code: 1

Patient being treated: Brown
Patients waiting to be seen:
Smith code: 5
Jones code: 4
Fehrenbach code: 6

Patient being treated: Jones
Patients waiting to be seen:
Smith code: 5
Fehrenbach code: 6
```