Queues - Notes

What is a Queue?5

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.

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.

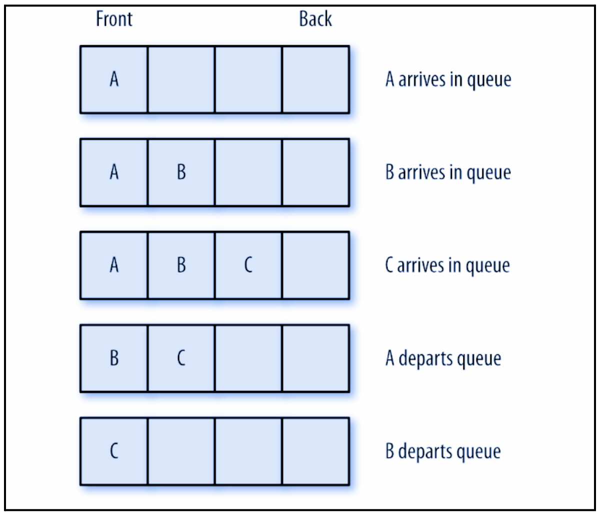

Figure shows inserting and deleting elements from a queue

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.

Queue Class Implementation Using Arrays

Implementing the Queue class using an array is straightforward. Using JavaScript arrays is an advantage many other programming languages don't have because JavaScript con- tains 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. Here is an example:

```
names = [];
name.push("Hello");
names.push("World");
print(names); // displays Hello World
```

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

```
names.shift();
print(names); // displays Hello
```

Now we're ready to begin implementing the Queue class by defining 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() function adds an element to the end of a queue:

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

The dequeue() function 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 functions:

```
function front() {
return this.dataStore[0];}
function back() {
return this.dataStore[this.dataStore.length-1];}
```

We also need a toString() function 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 function that lets us know if a queue is empty:

```
function empty() {
if (this.dataStore.length == 0) {
return true; }
else { return false;}
}
```

Priority Queues

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.

There are two options when implementing a priority queue: you can set the priority and add the element at the correct position, or you can queue the elements as they are added to the queue, and remove them according to priority. For this example, we will add theelements at their correct position, so we can dequeue them by default:

```
function PriorityQueue() {
let items = [];
function QueueElement (element, priority){
// {1}
this.element = element;
this.priority = priority;
}
this.enqueue = function(element, priority){
let queueElement = new QueueElement(element, priority);
let added = false;
for (let i=0; i<items.length; i++){
if (queueElement.priority < items[i].priority){
// {2}
items.splice(i,0,queueElement);
// {3}
added = true;
break;
// {4}
}
}
if (!added){
items.push(queueElement);
//{5}
}
};
this.print = function(){
for (let i=0; i<items.length; i++){
console.log(`${items[i].element} - ${items[i].priority}`); } };
//other methods - same as default Queue implementation }
```

The difference between the implementation of the default Queue and PriorityQueue classes is that we need to create a special element (line {1}) to be added to PriorityQueue. This element contains the element that we want to add to the queue (it can be any type), plus the priority on the queue. First we need to compare its priority to the rest of the elements (line {2}). When we find an item that has a higher priority than the element we are trying to add, then we insert the new element one position before (with this logic, we also respect the other elements with the same priority, but which were added to the queue first). To do this, we can use the splice method from the JavaScript array class that you learned about in Chapter 2, Arrays. Once we find an element with a higher priority, we insert the new element (line {3}), and we stop looping the queue (line {4}). This way, our queue will also be sorted and organized by priority. Also, if the priority we are adding is greater than any priority already added, or if the queue is empty, we simply add to the end of the queue (line {5}):

```
let priorityQueue = new PriorityQueue()
priorityQueue.enqueue("John", 2) ;
priorityQueue. enqueue ("Jack", 1);
priorityQueue.enqueue("Camila", 1) ;
priorityQueue.print();
```

In the previous code, we can see an example of how to use the PriorityQueue class. We

can see each command result in the following diagram (a result of the previous code):

Queue Time Complexity Summary

Queue Operation	**Access**	**Search**	**Peek**	**Insertion**	**Deletion**
Time Complexity	O(n)	O(n)	O(1)	O(1)	O(n^3)