5 Queues and Deques

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).

https://packt.link/EarlyAccess/

We have explored the inner workings of **stacks**, a data structure governed by the *LIFO* (Last in First out) principle. Now, let's turn our attention to **queues**, a similar yet distinct data structure. While stacks prioritize the most recent additions, queues operate on a *FIFO* (First in First out) basis, prioritizing the earliest entries. We will delve into the mechanics of queues and then explore **deques**, a versatile hybrid data structure that combines elements of both stacks and queues. By the end of this chapter, you will have a solid understanding of these fundamental data structures and their practical applications.

In this chapter, we will cover the following topics:

- The queue data structure
- The deque data structure
- Adding elements to a queue and a deque
- Removing elements from a queue and a deque
- Simulating circular queues with the *Hot Potato* game
- Checking whether a phrase is a palindrome with a deque

Different problems we can resolve using queues and deques

The queue data structure

Queues are all around us in everyday life. Think of the line to buy a movie ticket, the cafeteria queue at lunchtime, or the checkout line at the grocery store. In each of these scenarios, the underlying principle is the same: the first person to join the queue is the first one to be served.

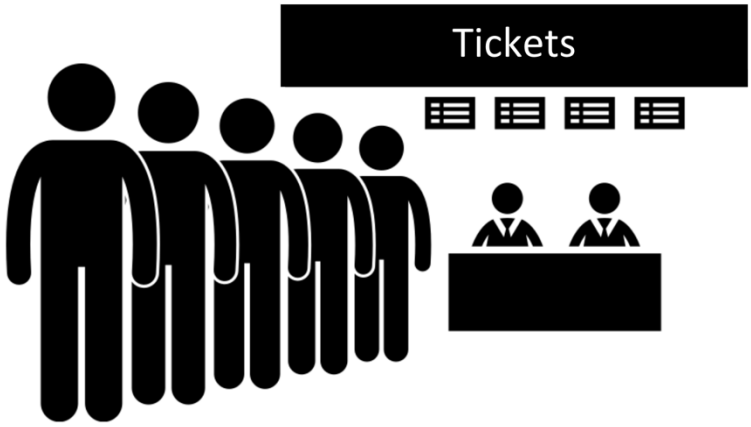


Figure 5.1: Real life queue example: a group of people standing in line to buy a ticket

An extremely popular example in computer science is the printing line. Let's say we need to print five documents. We open each document and click on the *Print* button. Each document will be sent to the print line. The first document that we asked to be printed is going to be printed first and so on, until all the documents are printed.

In the realm of data structures, a queue is a linear collection of elements that adheres to the *First In, First Out* (FIFO) principle, often referred to as *First Come, First Served*. New elements are always added at the rear (end) of the queue, while removal of elements always occurs at the front (beginning).

Let's put these concepts into practice by creating our own Queue class using JavaScript and TypeScript.

Creating the Queue class

We are going to create our own class to represent a queue. The source code for this chapter is available inside the `src/05-queue-deque` folder on GitHub. We will start by creating the `queue.js` file which will contain our class that represents a stack using an array-based approach.

In this book, we will take an incremental approach to building our understanding of data structures. We will leverage the concepts we mastered in the previous chapter and gradually increase the complexity as we progress (so make sure you do not skip chapters). This approach will allow us to build a solid foundation and tackle more intricate structures with confidence.
First, we will declare our `Queue` class:

```
class Queue {
  #items = [];
  // other methods
}
```

We need a data structure that will store the elements of the queue. We can use an array to do this as we are already familiar with the array data structure, and we have also learned in the previous chapter that an array-based approach is preferred compared to an object-based approach.

And again, we will prefix the variable `items` with the hash `#` prefix to indicate this property is private and can only be referenced inside the `Queue` class, hence, allowing us to protect the data and follow the FIFO principle when it comes to the insertion and removal of elements.

The following methods will be available in the `Queue` class:

`enqueue(item)`: This method adds a new item at the end of the queue.
`dequeue()`: This method removes the first item from the beginning of the queue. It also returns the removed item.
`front()`: This method returns the first element from the beginning of the queue. The queue is not modified (it does not remove the element; it only

returns the element for information purposes). This is like the `peek` method from the `Stack` class.

`isEmpty()`: This method returns `true` if the queue does not contain any elements, and `false` if the size of the stack is bigger than 0.

`clear()`: This method removes all the elements of the queue.

`size()`: This method returns the number of elements that the queue contains.

We will code each method in the following subsections.

Enqueueing elements to end of the queue

The first method that we will implement is the `enqueue` method. This method is responsible for adding new elements to the queue, with one especially important detail: we can only add new items at the rear of the queue as follows:

```
enqueue(item) {
  this.#items.push(item);
}
```

As we are using an array to store the elements of the queue, we can use the `push` method from the JavaScript `Array` class that will add a new item to the end of the array. The `enqueue` method has the same implementation as the `push` method from the `Stack` class; from a code standpoint, we are only changing the name of the method.

Dequeuing elements from beginning of the queue

Next, we are going to implement the `dequeue` method. This method is responsible for removing the items from the queue. As the queue uses the FIFO principle, the item at the beginning of the queue (index 0 of our internal array) is removed. For this reason, we can use the `shift` method from the JavaScript `Array` class as follows:

```
dequeue() {
  return this.#items.shift();
}
```

The `shift` method from the `Array` class removes the first element from an array and returns it. If the array is empty, `undefined` is returned and the array is not modified. So this works perfectly with the behavior of the queue.

The `enqueue` method has constant time complexity (*O(1)*). The `dequeue` method can have linear time complexity (*O(n)*), since we are using the `shift` method to remove the first element of an array, it can lead to *O(n)* performance in the worst case as the remaining elements need to be shifted down.

Peeking the element from the front of the queue

Next, we will implement additional helper methods in our class. If we would like to know what the front element of the queue is, we can use the `front` method. This method will return the item that is located at the index 0 of our internal array:

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

Verifying if it is empty, the size and clearing the queue

The next methods we will create are the `isEmpty` method, `size` getter, and the `clear` method. These three methods have the exact same implementation as the `Stack` class:

```
 isEmpty() {
    return this.#items.length === 0;
  }
  get size() {
    return this.#items.length;
  }
  clear() {
    this.#items = [];
  }
```

Using the `isEmpty` method, we can simply verify whether the length of the internal array is 0.

For the size, we create a getter for the `size` of the queue, which is simply the length of the internal array.

And for the `clear` method, we can simply assign a new empty array that will represent an empty queue.

Finally, we have the `toString` method, with the same code as the `Stack` class as follows:

```
toString() {
  if (this.isEmpty()) {
    return 'Empty Queue';
  } else {
    return this.#items.map(item => {
      if (typeof item === 'object' && item !== null) {
        return JSON.stringify(item);
      } else {
        return item.toString();
      }
    }).join(', ');
  }
}
```

Exporting the Queue data structure as a library class

We have created a file src/05-queue-deque/queue.js with our `Queue` class. And we would like to use the `Queue` class in a different file for testing for easy maintainability of our code (`src/05-queue-deque/01-using-queue-class.js`). How can we achieve this?

We covered this in the previous chapter as well. We will use the `module.exports` from `CommonJS` Module to expose our class:

```
// queue.js
class Queue {
  // our Queue class implementation
}
module.exports = Queue;
```

Using the Queue class

Now it is time to test and use our `Queue` class! As discussed in the previous subsection, let's go ahead and create a separate file so we can write as many tests as we like: `src/05-queue-deque/01-using-queue-class.js`.

The first thing we need to do is to import the code from the `queue.js` file and instantiate the `Queue` class we just created:

```
const Queue = require('./queue');
const queue = new Queue();
```

Next, we can verify whether it is empty (the output is `true`, because we have not added any elements to our queue yet):

```
   console.log(queue.isEmpty()); // true
```

Next, let's simulate a printer's queue. Suppose we have three documents open in a computer. And we click on the print button in each document. By doing so, it will enqueue the documents to the queue in the order the print button was clicked:

```
queue.enqueue({ document: 'Chapter05.docx', pages: 20 });
queue.enqueue({ document: 'JavaScript.pdf', pages: 60 });
queue.enqueue({ document: 'TypeScript.pdf', pages: 80 });
```

If we call the `front` method, it is going to return the file `Chapter05.docx`, because it was the first document that was added to the queue to be printed:

```
console.log(queue.front()); // { document: 'Chapter05.docx', pages:
20 }
```

Let's also check the queue size:

```
   console.log(queue.size); // 3
```

Now let's "print" all documents in the queue by dequeuing them until the queue is empty:

```
// print all documents
while (!queue.isEmpty()) {
```

```
  console.log(queue.dequeue());
}
```

The following diagram shows the dequeue operation when printing the first document from the queue:

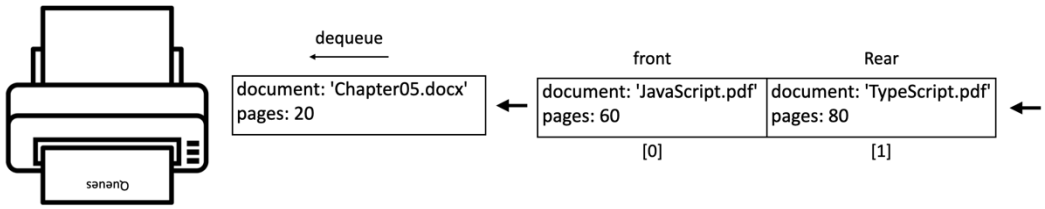


Figure 5.2: Simulation of a printer queue

Reviewing the efficiency of our Queue class

Let's review the efficiency of each method of our queue class by considering the Big O notation in terms of time of execution:

Method	Complexity	Explanation
`enqueue`	*O(1)*	Adding an element to the end of an array is usually a constant-time operation.
`dequeue`	*O(n)*	Removing the first element requires shifting all remaining elements, taking time proportional to the queue's size.
`front`	*O(1)*	Directly accessing the first element by index is a constant-time operation.
`isEmpty`	*O(1)*	Checking the length property of an array is a constant-time operation.
`size`	*O(1)*	Accessing the length property is a constant-time operation.
`clear`	*O(1)*	Overwriting the internal array with an empty array is considered constant time.
`toString`	*O(n)*	Iterating over the elements, potentially stringifying them, and joining them into a string takes time proportional to the number of elements.

Table 5.1:

The `dequeue` operation is generally the most performance-sensitive operation for a queue implemented using an array. This is due to the need to shift elements after removing the first one. There are alternative queue implementations (for example using a linked list, which we will cover in the next chapter, or a **circular queue**) that can optimize the dequeue operation to have constant time complexity in most cases. We will create a circular queue later in this chapter.

The deque data structure

The **deque** data structure, also known as the **double-ended queue**, is a special queue that allows us to insert and remove elements from the end or from the front of the deque.

A deque can be used to store a user's web browsing history. When a user visits a new page, it is added to the front of the deque. When the user navigates back, the most recent page is removed from the front, and when the user navigates forward, a page is added back to the front.

Another application would be an undo/redo feature. We learned we can use two stacks for this feature in the last chapter, but we can also use a deque as an alternative. User actions are pushed onto the deque, and undo operations pop actions off the front, while redo operations push them back on.

Creating the Deque class

As usual, we will start by declaring the `Deque` class located in the file `src/05-queue-deque/deque.js`:

```
class Deque {
  #items = [];
}
```

We will continue using an array-based implementation for our data structure. And given the deque data structure allows inserting and removing from both ends, we will have the following methods:

- `addFront(item)`: This method adds a new element at the front of the deque.
- `addRear(item)`: This method adds a new element at the back of the deque.
- `removeFront()`: This method removes the first element from the deque.
- `removeRear()`: This method removes the last element from the deque.
- `peekFront()`: This method returns the first element from the deque.
- `peekRear()`: This method returns the last element from the deque.

The `Deque` class also implements the `isEmpty`, `clear`, `size`, and `toString` methods (you can check the complete source code by downloading the source code bundle for this book). The code for these is the same as for the `Queue` class.

Adding elements to the deque

Let's check both methods that will allow us to add elements to the front and to the back of the deque:

```
addFront(item) {
  this.#items.unshift(item);
}
addRear(item) {
  this.#items.push(item);
}
```

The `addFront` method inserts an element at index 0 of the internal array by using the `Array.unshift` method.

The `addRear` method inserts an element at the end of the deque. It has the same implementation as the `enqueue` method from the `Queue` class.

Removing elements from the deque

The methods to remove from both the front and the back of the deque are presented as follows:

```
removeFront() {
  return this.#items.shift();
}
removeRear() {
```

```
  return this.#items.pop();
}
```
The `removeFront` method removes and returns the element at the beginning (front) of the deque. If the deque is empty, it returns `undefined`. It has the same implementation as the `dequeue` method from the `Queue` class.

The `removeRear` method removes and returns the element at the end (rear) of the deque. If the deque is empty, it returns `undefined`. It has the same implementation as the `pop` method from the `Stack` class.

Peeking elements of the deque

Finally, let's check the peek methods as follows:

```
peekFront() {
  return this.#items[0];
}
peekRear() {
  return this.#items[this.#items.length - 1];
}
```
The `peekFront` method allows you to look at (peek) the element at the beginning (front) of the deque without removing it. It has the same implementation as the `peek` method from the `Queue` class.

The `peekRear` method allows you to look at (peek) the element at the end (rear) of the deque without removing it. It has the same implementation as the `peek` method from the `Stack` class.

Note the similarities of the implementation of the deque methods with the Stack and Queue classes. We can say the deque data structure is a hybrid version of the stack and queue data structures. Please refer to the Queue and Stack efficiency review to check the time complexity for these methods.

Using the Deque class

It is time to test our Deque class (`src/05-queue-deque/03-using-deque-class.js`). We will use it in the scenario of a browser's "*Back*" and "*Forward*" button functionality. Let's see how this could be implemented using our Deque class:

```
const Deque = require('./deque');
class BrowserHistory {
  #history = new Deque(); // {1}
  #currentPage = null; // {2}
  visit(url) {
    this.#history.addFront(url); // {3}
    this.#currentPage = url; // {4}
  }
  goBack() {
    if (this.#history.size() > 1) { // {5}
      this.#history.removeFront();  // {6}
      this.#currentPage = this.#history.peekFront(); // {7}
    }
  }
  goForward() {
    if (this.#currentPage !== this.#history.peekBack()) { // {8}
      this.#history.addFront(this.#currentPage);  // {9}
      this.#currentPage = this.#history.removeFront(); // {10}
    }
  }
  get currentPage() { // returns the current page for information
    return this.#currentPage;
  }
}
```

Following is the explanation:

> A `Deque` named `history` is created to store URLs of visited pages (`{1}`). The `currentPage` variable keeps track of the currently displayed page (`{2}`). The `visit(url)` method adds the new `url` to the front of the `history` deque (`{3}`) and updates the `currentPage` to the new URL (`{4}`).

The `goBack()` method:
Checks if there are at least two pages in the `history` (current and previous - `{5}`).
If so, it removes the current page from the front of the history deque (`{6}`).
Updates `currentPage` to the now-front element, which represents the previous page (`{7}`).
The `goForward()` method:
Checks if the `currentPage` is different from the last element in the history deque (meaning there is a "next" page - `{8}`).
If so, adds the current page back to the front of the history deque (`{9}`).
Removes and sets `currentPage` to the now-front element, which was the "next" page (`{10}`).

With our browser simulation ready, we can use it:

```
const browser = new BrowserHistory();
browser.visit('loiane.com');
browser.visit('https://loiane.com/about'); // click on About menu
browser.goBack();
console.log(browser.currentPage); // loiane.com
browser.goForward();
console.log(browser.currentPage); // https://loiane.com/about
```

We will simulate visiting the website **https://loiane.com**, which is the author's blog. Next, we will visit the About page so we can add another URL to the browser's history. Then, we can click on the "Back" button to go back to the home page. And we can also click on the "Next" button to go back the About page. And of course, we can peek what is the current page or the history as well. The following image exemplifies this simulation:

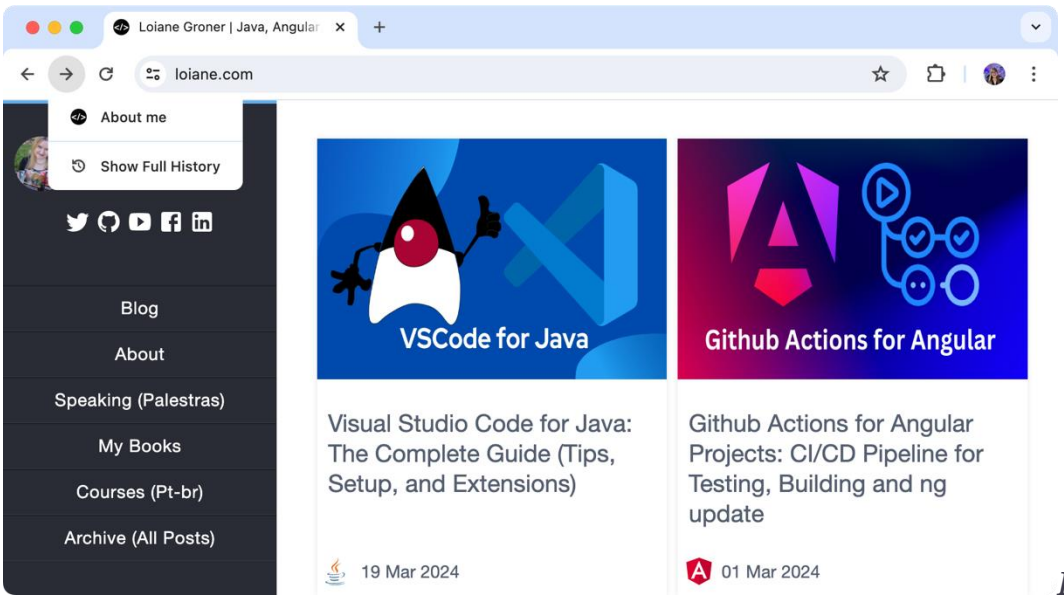


Figure 5.3: Simulation of a browser Back and Next buttons

Creating the Queue and Deque classes in TypeScript

With the JavaScript implementation done, we can rewrite our Queue and Deque classes using TypeScript:

```
// src/05-queue-deque/queue.ts
class Queue<T> {
  private items: T[] = [];
  enqueue(item: T): void {}
  // all other methods are the same as in JavaScript
}
export default Queue;
```

And the `Deque` class:

```
// src/05-queue-deque/deque.ts
class Deque<T> {
  private items: T[] = [];
  addFront(item: T): void {}
  addRear(item: T): void {}
  // all other methods are the same as in JavaScript
}
export default Deque;
```

We will use the generics to make our data structures flexible and have items of only one type. The code inside the methods is the same as the JavaScript implementation.

Solving problems using queues and deques

Now that we know how to use the `Queue` and `Deque` classes, let's use them to solve some computer science problems. In this section, we will cover a simulation of the *Hot Potato* game with queues and how to check whether a phrase is a *palindrome* with deques.

The circular queue: the Hot Potato game

The Hot Potato game is a classic children's game where participants form a circle and pass around an object (the "hot potato") as fast as they can while music plays. When the music stops, the person holding the potato is eliminated. The game continues until only one person remains.

The CircularQueue class

We can perfectly simulate this game using a **Circular Queue** implementation:

```
class CircularQueue {
  #items = [];
  #capacity = 0; // {1}
  #front = 0; // {2}
  #rear = -1; // {3}
  #size = 0; // {4}
  constructor(capacity) { // {5}
    this.#items = new Array(capacity);
    this.#capacity = capacity;
  }
  get size() { return this.#size; }
}
```

A circular queue is a queue implemented using a fixed-size array, meaning a pre-defined capacity (`{1}`), where the front (`{2}`) and rear (`{3}`) pointers can "wrap

around" to the beginning of the array when they reach the end. This efficiently reuses the space in the array, avoiding unnecessary resizing. The `front` pointer is initialized to 0, pointing to the first element's position. The `rear` pointer is initialized to -1, indicating an empty queue. The `size` property (`{4}`) tracks the current number of elements in the queue.

When we create a circular queue, we need to inform how many elements we are planning to store (`{5}`).

Let's review the `enqueue` and `isFull` methods next:

```
enqueue(item) {
  if (this.isFull()) { // {6}
    throw new Error("Queue is full");
  }
  this.#rear = (this.#rear + 1) % this.#capacity; // {7}
  this.#items[this.#rear] = item; // {8}
  this.#size++; // {9}
}
isFull() { return this.#size === this.#capacity; }
```

Before we add any elements to queue, we need to check if it is not full, meaning the size is the same as the capacity (`{6}`). With a fixed capacity, this makes the circular queue predictable in terms of memory usage, but this can also be viewed as a limitation.

If the queue is not full, we increment the `rear` pointer by 1. The key point here is to use the modulo operator (`%`) to wrap rear back to 0 if it reaches the end of the array (`{7}`). Then we insert the item at the new rear position (`{8}`) and increment the size counter (`{9}`).

Finally, we have the `dequeue` and `isEmpty` methods:

```
dequeue() {
  if (this.isEmpty()) { throw new Error("Queue is empty"); } // {10}
  const item = this.#items[this.#front]; // {11}
  this.#size--; // {12}
```

```
  if (this.isEmpty()) {
    this.#front = 0; // {13}
    this.#rear = -1; // {14}
  } else {
    this.#front = (this.#front + 1) % this.#capacity; // {15}
  }
  return item; // {16}
}
isEmpty() { return this.#size === 0;  }
```

To remove the item at the front of the queue, first, we need to check the queue size (`{10}`). If the queue is not empty, we can retrieve the item that is currently stored at the front position (`{11}`) so we can return it later (`{16}`). Then, we decrement the size counter (`{12}`).

If the queue is not empty after dequeuing, we need to increment the `front` pointer by 1 and wrap around it using the modulo operator (`{15}`). If the queue is empty after dequeuing, we reset both `front` (`{13}`) and `rear` (`{14}`) pointers to their initial values.

The biggest advantage of the circular queue is both enqueueing and dequeuing operations are generally *O(1)* (constant time) due to direct manipulation of pointers.

The Hot Potato game simulation

With the new class ready to be used, let's put it in practice the Hot Potato game simulation:

```
function hotPotato(players, numPasses) {
  const queue = new CircularQueue(players.length); // {1}
  for (const player of players) { // {2}
    queue.enqueue(player);
  }
  while (queue.size > 1) { // {3}
    for (let i = 0; i < numPasses; i++) { // {4}
      queue.enqueue(queue.dequeue()); // {5}
    }
```

```
    console.log(`${queue.dequeue()} is eliminated!`); // {6}
  }
  return queue.dequeue(); // {7} The winner
}
```

This function takes an array of players and a number `num` representing the number of times the "potato" is passed before a player is eliminated and it returns the winner.

First, we create a circular queue (`{1}`) with the numbers of players and we enqueue all players (`{2}`).

We run a loop until only one player remains (`{3}`):

- Another loop will pass the potato `numPasses` times (`{4}`) by dequeueing and then re-enqueueing the same player (`{5}`).
- The player at the front is removed and announced as eliminated (`{6}`).

The last remaining player is dequeued and declared the winner (`{7}`).

We can use the following code to try the `hotPotato` algorithm:

```
const players = ["Violet", "Feyre", "Poppy", "Oraya", "Aelin"];
const winner = hotPotato(players, 7);
console.log(`The winner is: ${winner}!`);
```

The execution of the algorithm will have the following output:

```
Poppy is eliminated!
Feyre is eliminated!
Aelin is eliminated!
Oraya is eliminated!
The winner is: Violet!
```

This output is simulated in the following diagram:

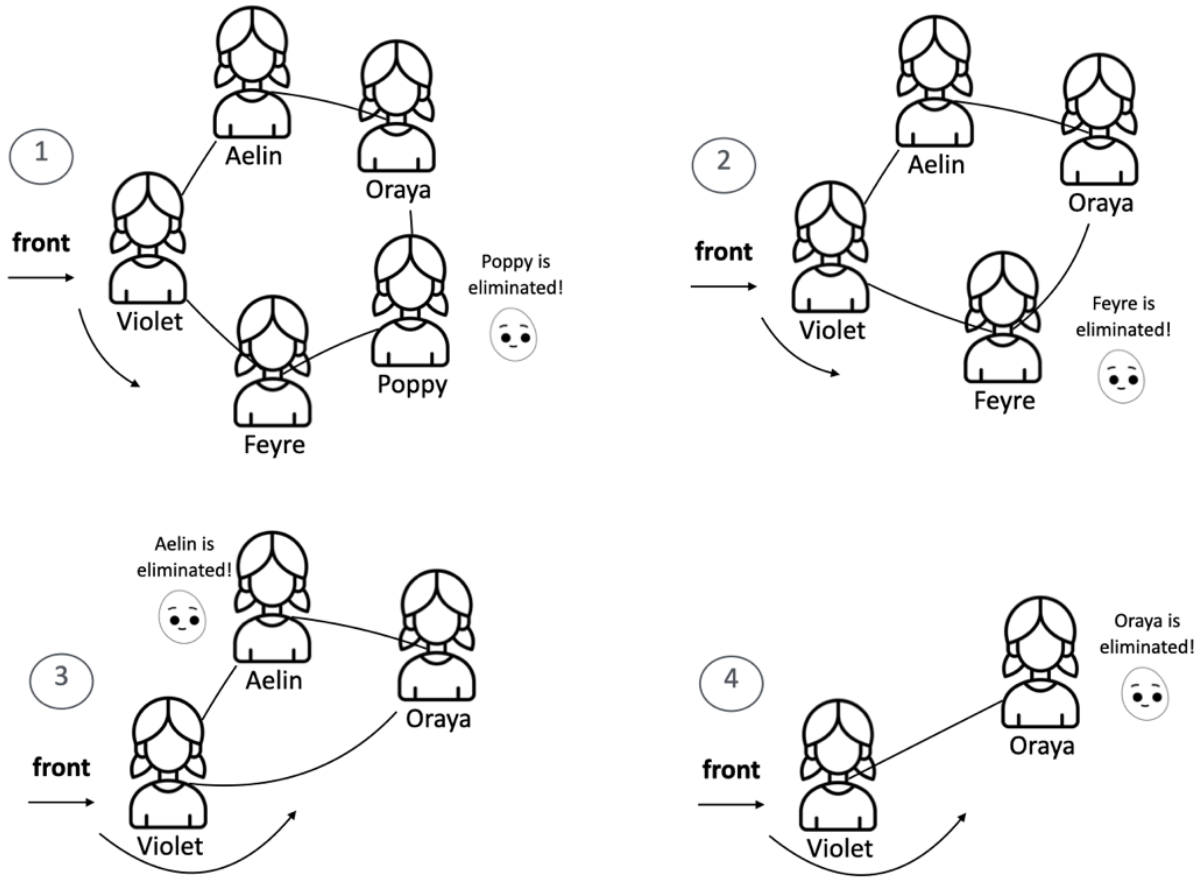


Figure 5.4: The Hot Potato game simulation using circular queue

You can change the number of passes using the `hotPotato` function to simulate different scenarios.

Palindrome checker

The following is the definition of a **palindrome** according to Wikipedia:

A palindrome is a word, phrase, number, or other sequence of characters which reads the same backward as forward, such as madam or racecar.

There are different algorithms we can use to verify whether a phrase or string is a palindrome. The easiest way is reversing the string and comparing it with the original string. If both strings are equal, then we have a palindrome. We can also use a stack to do this, but the easiest way of solving this problem using a data structure is using a deque: characters are added to the deque, and then removed

from both ends simultaneously. If the removed characters match throughout the process, the string is a palindrome.

The following algorithm uses a deque to solve this problem:

```
const Deque = require('./deque');
function isPalindrome(word) {
  if (word === undefined || word === null ||
     (typeof word === 'string' && word.length === 0)) { // {1}
    return false;
  }
  const deque = new Deque(); // {2}
  word = word.toLowerCase().replace(/\s/g, ''); // {3}
  for (let i = 0; i < word.length; i++) {
    deque.addRear(word[i]); // {4}
  }
  while (deque.size() > 1) { // {5}
    if (deque.removeFront() !== deque.removeRear()) { // {6}
      return false;
    }
  }
  return true;
}
```

Before we start with the algorithm logic, we need to verify whether the string that was passed as a parameter is valid with the edge cases (`{1}`). If it is not valid, then we return `false` because an empty string or non-existent word cannot be considered a palindrome.

We will use the Deque class we implemented in this chapter (`{2}`). As we can receive a string with both lowercase and capital letters, we will transform all letters to lowercase, and we will also remove all the spaces (`{3}`). If you want to, you can also remove all special characters such as !?-() and so on. To keep this algorithm simple, we will skip this part. Next, we will enqueue all characters of the string to the deque (`{4}`).

While we have at least two elements in the deque (`{5}` - if only one character is left, it is a palindrome), we will remove one element from the front and one from the back (`{6}`). To be a palindrome, both characters removed from the deque need to match. If the characters do not match, then the string is not a palindrome (`{7}`).

We can use the following code to try the `isPalindrome` algorithm:

```
console.log(isPalindrome("racecar")); // Output: true
```

Exercises

We will resolve an exercise from **LeetCode** using the concepts we learned in this chapter.

Number of Students Unable to Eat Lunch

The exercise we will resolve is the *1700. Number of Students Unable to Eat Lunch* problem available at **https://leetcode.com/problems/number-of-students-unable-to-eat-lunch/**.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function `function countStudents(students: number[], sandwiches: number[]): number {}`, which receives a queue of `students` that would prefer to eat a sandwich 0 or 1 and a stack of `sandwiches`, which will have the same size.

This is a simulation exercise, according to the problem's description:

> If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.
> Otherwise, they will leave it and go to the queue's end.
> This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

The key to resolve this problem is to also treat the stack of sandwiches as a queue (*FIFO*) instead of a stack (*LIFO*).

Let's write the `countStudents` function:

```
function countStudents(students: number[], sandwiches: number[]) {
    while (students.length > 0) { // {1}
      if (students[0] === sandwiches[0]) { // {2}
          students.shift(); // {3}
          sandwiches.shift(); // {4}
      } else {
          if (students.includes(sandwiches[0])) { // {5}
              let num = students.shift(); // {6}
              students.push(num); // {7}
          } else {
              break; // {8}
          }
      }
  }
  return students.length;
}
```

The while loop keeps running as long as there are students in the queue (`{1}`).

If the first student's preference(`students[0]`) matches (`{2}`) the sandwich at the front (top sandwich `sandwiches[0]`), both the student (`{3}`) and sandwich (`{4}`) are removed from the front of their respective queues.

If there is not a match, we check for potential match using the `includes` method from the JavaScript `Array` class to see if any student in the queue would take the current top sandwich (`{5}`). If there is a potential match, the current student (`{6}`) is moved to the back of the queue (`{7}`).

If there is no one left in line who wants the current sandwich, the loop breaks (`{8}`). This indicates that the remaining students will not be able to eat.

The function returns the length of the `students` array. This length represents the number of students still in line who could not get a sandwich they liked.

This solution passes all the tests and resolves the problem. The `includes` check (`{5}`) is important for efficiency. Without it, the code would unnecessarily rotate the queue even when no student would want the top sandwich, so we get bonus points, although this method is not native for a queue data structure.

The time and space complexity of this function is *O(nˆ2)*, where *n* is the number of `students`. This is because the outer loop runs until there are no students left, which in the worst case is *n* iterations.

Inside the loop, there are two operations that can be costly: `students.shift()`, which is *O(n)*, and `students.includes(sandwiches[0])`, which is also *O(n)*. Since these operations are nested inside the loop, the total complexity is *O(nˆ2)*.

The space complexity is *O(1)*, not counting the space needed for the input arrays. This is because the function only uses a fixed amount of additional space to store the `num` variable.

Can you think of a more optimized approach to resolve this problem that might not involve the queue data structure? During technical interviews, it is important to also think about optimizations. Give it a try, and you will find the solution along with the explanation in the source code, along with the 2034. Time Needed to Buy Tickets problem resolution as well.

Summary

In this chapter, we delved into the fundamental concept of queues and their versatile cousin, the deque. We crafted our own queue algorithm, mastering the art of adding (enqueue) and removing (dequeue) elements in a first-in, first-out (FIFO) manner. Exploring the deque, we discovered its flexibility in inserting and deleting elements from both ends, expanding the possibilities for creative solutions.

To solidify our understanding, we applied the knowledge to real-world scenarios. We simulated the classic Hot Potato game, leveraging a circular queue to model its cyclical nature. Additionally, we created a palindrome checker, demonstrating the deque's power in handling data from both directions. We also tackled a simulation challenge from LeetCode, reinforcing the practical applications of queues in problem-solving.

Now, with a firm grasp of these linear data structures (arrays, stacks, queues and deques), we are ready to venture into the dynamic world of linked lists in the next chapter, where we will unlock even greater potential for complex data manipulation and management.

6 Linked Lists

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).

https://packt.link/EarlyAccess/

In previous chapters, we explored data structures stored sequentially in memory. Now, we turn our attention to linked lists, a dynamic and linear data structure with a non-sequential memory arrangement. This chapter delves into the inner workings of linked lists, covering:

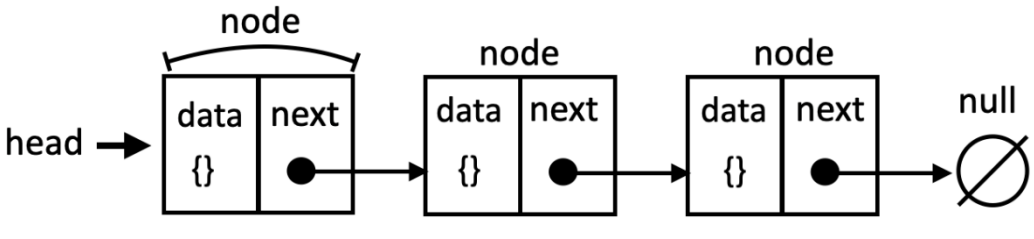
- The linked list data structure
- Techniques for adding and removing elements from linked lists
- Variations of linked lists: doubly linked lists, circular linked lists and sorted linked lists
- How linked lists can be used to implement other data structures
- Implementing other data structures with linked lists
- Exercises using linked lists

The linked list data structure

Arrays, a ubiquitous data structure found in nearly every programming language, offer a convenient way to store collections of elements. Their familiar bracket notation ([ ]) provides direct access to individual items. However, arrays

come with a key limitation: their fixed size in most languages. This constraint makes inserting or removing elements from the beginning or middle a costly operation due to the need to shift remaining elements. While JavaScript provides methods to handle this, the underlying process still involves these shifts, impacting performance.

Linked lists, like arrays, maintain a sequential collection of elements. However, unlike arrays where elements occupy contiguous memory locations, linked lists store elements as nodes scattered throughout memory. Each node encapsulates the element's data (the information or value we want to store) along with a reference (also called a pointer or link) that directs you to the next node in the sequence. The following diagram illustrates this linked list structure:



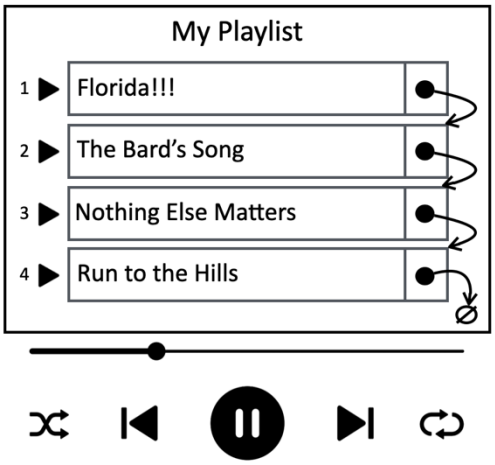
The structure of a linked list data structure with nodes, data and pointers

The first node is called the *head*, and the last node usually points to `null` (or `undefined`) to indicate the end of the list.

A key advantage of linked lists over conventional arrays is the ability to insert or remove elements without the costly overhead of shifting other items. However, this flexibility comes with the tradeoff of using pointers, requiring greater care during implementation. While arrays allow direct access to elements at any position, linked lists necessitate traversal from the head to reach elements in the middle, potentially impacting access time.

However, linked lists are not the best data structure if you need to access elements by their index (like we do with arrays). This is because we need to traverse the list from the beginning, which can be slower. Linked lists also require additional memory for storage, as each node requires extra memory to store the pointer(s), which can be overhead for simple data.

Linked lists have numerous real-world applications due to their flexibility and efficiency in handling dynamic data. One of the popular examples are media players. Media players use linked lists to organize and manage playlists. Adding, removing, and rearranging songs or videos are straightforward operations on a linked list as represented as follows:



A media player representation using linked list as data structure

There are different types of linked lists:

Singly Linked List (or simply Linked List): each node has a pointer to the next node.
Doubly Linked List: Each node has pointers to both the next and previous nodes.
Circular Linked List: The last node points back to the head, forming a loop.

In this chapter, we will cover the linked list as well as these variations, but let's start with the easiest data structure first.

Creating the LinkedList class

Now that you understand what a linked list is, let's start implementing our data structure. We are going to create our own class to represent a linked list. The source code for this chapter is available inside the `src/06-linked-list` folder. We will start by creating the `linked-list.js` file which will contain our class that

represents our data structure as well the node needed to store the data and the pointer.

To start, we define a `LinkedListNode` class, which represents each element (or node) within our linked list. Each node holds the `data` we want to store, along with a reference (`next`) pointing to the subsequent node:

```
class LinkedListNode {
  constructor(data, next = null) {
    this.data = data;
    this.next = next;
  }
}
```

By default, a newly created node's `next` pointer is initialized to `null`. However, the constructor also allows you to specify the next node if it is known beforehand, proving beneficial in certain scenarios.

Next, we declare the `LinkedList` class which represents our linked list data structure:

```
class LinkedList {
  #head;
  #size = 0;
  // other methods
}
```

This class begins by declaring a private `#head` reference, pointing to the first node (element) in the list. To avoid traversing the entire list whenever we need the element count, we also maintain a private `#size` variable. Both properties are kept private using the `#` prefix to ensure encapsulation.

The `LinkedList` class will provide the following methods:

- `append(data)`: adds a new node containing the `data` at the end of the list.
- `prepend(data)`: adds a new node containing the `data` at the beginning (head) of the list.

`insert(data, position)`: inserts a new node containing the `data` at the specified `position` in the list.
`removeAt(position)`: removes the node at the specific `position` in the list.
`remove(data)`: removes the first node containing the specified `data` from the list.
`indexOf(data)`: returns the index of the first node containing the specified `data`. If the `data` is not found, returns `-1`.
`isEmpty()`: returns `true` if the list does not contain any elements, and `false` otherwise.
`clear()`:removes all the elements from the list.
`size()`:returns the number of elements currently in the list.
`toString()`:returns a string representation of the linked list, showing the elements in order.

We will implement each of these methods in detail in the following sections.

Appending elements to the end of the linked list

When appending an element at the end of a LinkedList, we encounter two scenarios:

Empty list: the list has no existing elements, and we are adding the first one.
Non-empty list: the list already contains elements, and we are adding a new one to the end of the list.

The following is the implementation of the `append` method:

```
append(data) {
  const newNode = new LinkedListNode(data);
  if (!this.#head) {
    this.#head = newNode;
  } else {
    let current = this.#head;
    while (current.next !== null) {
      current = current.next;
    }
    current.next = newNode;
```

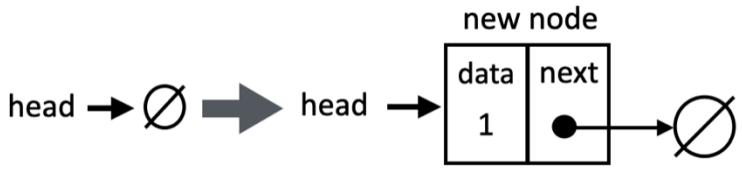
```
  }
  this.#size++;
}
```

Regardless of the list's state, the first step is to create a new node to hold the data.

For the first scenario, we check if the list is empty. The condition `!this.#head` evaluates to `true` if the `#head` pointer is currently `null` (or `undefined`), indicating an empty list.

if the list is empty, the newly created node (`newNode`) becomes the head of the list. Its next pointer will automatically be `null` since it is the only node in the list.

Let's see a visual representation of these steps:



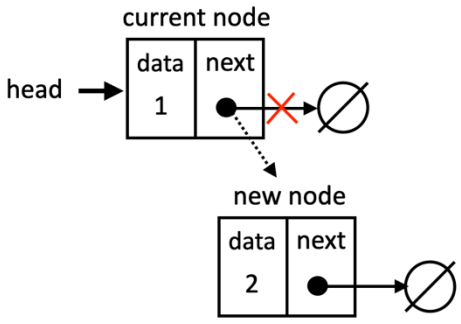
Adding a new element to an empty linked list

In the scenario where our linked list is not empty, we have a reference only to the `head` (the first node). To append a new element to the end, we need to traverse the list:

> We start by assigning a temporary variable, often called current, to the `head` of the list. This variable will act as our pointer as we move through the list.
> Using a while loop, we continuously move current to the next node (`current.next`) as long as `current.next` is not `null`. This means we keep moving until we reach the last node, whose next pointer will be `null`.
> Once the loop terminates, `current` will be referencing the last node. We simply set `current.next` to our new node, effectively adding it to the end of the list.

Finally, we increment the `size` to reflect the addition of the new node.

The following diagram exemplifies appending an element to the end of a linked list when it is not empty:



Adding a new element to the end of a linked list

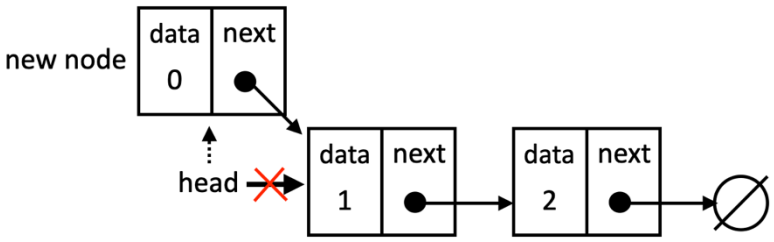
Prepending a new element to the linked list

Adding a new element as the head (beginning) of the linked list is a simple operation:

```
prepend(data) {
  const newNode = new LinkedListNode(data, this.#head);
  this.#head = newNode;
  this.#size++;
}
```

The first step is to create a new node to hold the `data`. Importantly, we pass the current `head` of the list as the second argument to the constructor. This sets the `next` pointer of the new node to the current `head`, establishing the link. If the list is empty, the current head is `null`, so the new node's `next` reference will also be `null`.

Next, we update the `head` of the list to point to the newly created node (`newNode`). Since the new node is already linked to the previous head, the entire list is seamlessly adjusted. And finally, we increase the `size` of the list to reflect the addition of the new node. The following diagram exemplifies this scenario:



Prepending a new element to the linked list

Inserting a new element at a specific position

Now, let's explore how to insert an element at any position within the linked list. For this, we will create an `insert` method, taking both the `data` and the desired `position` as parameters:

```
insert(data, position) {
  if (this.#isInvalidPosition(position)) {
    return false;
  }
  const newNode = new LinkedListNode(data);
  if (position === 0) {
    this.prepend(data);
    return true;
  }
  let current = this.#head;
  let previous = null;
  let index = 0;

  while (index++ < position) {
    previous = current;
    current = current.next;
  }

  newNode.next = current;
  previous.next = newNode;
  this.#size++;
  return true;
}
```

We first verify if the provided `position` is valid using a helper private method, `#isInvalidPosition`. A valid position is one that falls within the bounds of the list (0 to size-1). If the position is invalid, the method returns `false` to indicate the failure to insert. The helper method is declared as follows:

```
#isInvalidPosition(position) {
  return position < 0 || position >= this.size;
}
```

Next, we create the new node that will hold the data we are inserting.

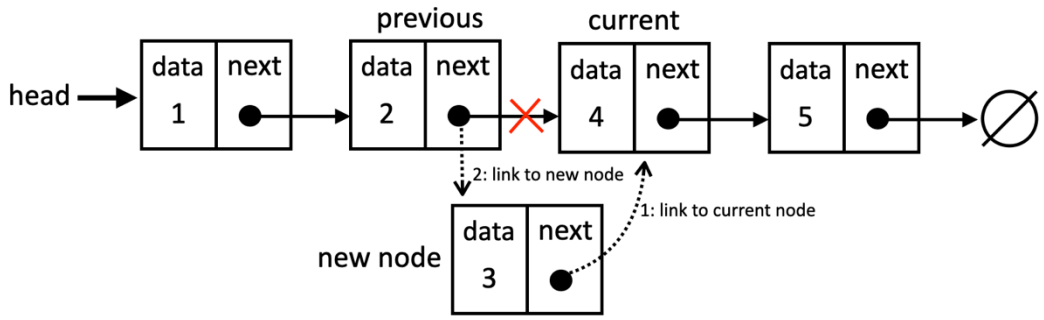
The first scenario is handling the insertion at the head of the list. if the `position` is 0, it means we're inserting at the beginning of the list. In this case, we can simply call the prepend method we defined earlier and return `true` for success.

If not inserting at the head, it means we will need to traverse the list. To do so, we will need a helper variable we will name current and set it to the first node (`head`). We also need a second helper variable to assist with the linkage of the new node we will name `previous`. And we also initialize an index variable to keep track of our position as we traverse.

Then, we will traverse the list until we reach the desired position. To do so, the `while` loop iterates until the index matches the `position`. In each iteration, we move `previous` to the `current` node and the `current` to the `next` node.

After the loop, `previous` points to the node before the insertion point, and `current` points to the node at the insertion point. We adjust the next pointers: `newNode.next` is set to current (the node that was originally at the insertion point), and `previous.next` is set to `newNode`, effectively inserting the new node into the list.

Let's see this scenario in action in the following diagram:



Inserting an element in the middle of a linked list

It is very important to have variables referencing the nodes we need to control so that we do not lose the link between the nodes. We could work with only one variable (`previous`), but it would be harder to control the links between the nodes. For this reason, it is better to declare an extra variable to help us with these references.

Returning the position of an element

Now that we know how to traverse the list until a desired position, it makes easier to traverse the list searching for a particular element and returning its index.

Let's review the `indexOf` method implementation:

```
indexOf(data, compareFunction = (a, b) => a === b) {
  let current = this.#head;
  let index = 0;
  while (current) {
    if (compareFunction(current.data, data)) {
      return index;
    }
    index++;
    current = current.next;
  }
  return -1;
}
```

We start by creating a variable `current` to track the node in the list and it is initially set to the head of the list. We also initialize an index variable to 0, representing the current position in the list.

The `while` loop continues if current is not `null` (we have not reached the end of the list). In each iteration, we check if the data property of the current node matches the element we are searching for, and if so, it returns the index of the element's position.

We can pass a custom comparison function to the `indexOf` method. This function should take two arguments (two different objects) and return `true` if the two objects match according to the desired criteria, or `false` otherwise. With this function, we gain flexibility and the ability to define exactly how elements are compared, accommodating complex data structures and different matching criteria. By default, we simply compare the references of the objects in case no comparison function is informed.

Using a comparison function is also a standard practice in other programming languages. If you prefer, instead of passing the function to the method directly, we can have the function in the constructor of the `LinkedList` class so it can be used whenever needed.
If the element is not found in the current node, it increments the `index` and moves to the next node.

If the loop completes without finding the element, it means the element is not present in the list. In this case, the method returns `-1` (which is an industry convention).

It is useful to have an `indexOf` method as we can use this method to search for elements, and we will also reuse it to remove elements from the list.

Removing an element from a specific position

Let's explore how we can remove elements from our linked list. Similar to appending, there are two scenarios to consider: removing the first element (the head) and removing any other element.

The `removeAt` code is presented as follows:

```
removeAt(position) {
   if (this.#size === 0) {
     throw new RangeError('Cannot remove from an empty list.');
   }
   if (this.#isInvalidPosition(position)) {
     throw new RangeError('Invalid position');
   }
   if (position === 0) {
     return this.#removeFromHead();
   }
   return this.#removeFromMiddleOrEnd(position);
 }
```

We will delve into this code step by step:

1. First, we check if the list is not empty, if it is empty, we return an error.
2. Next, we check if the given position is valid using the `#isInvalidPosition` helper method.
3. Then we check for the first scenario: removing the first element of the list, and if so, we will segregate the logic into a separate private method for better organization and understanding.
4. Finally, if we are not removing the first element, it means we are removing the last element or from the middle of the list. For the singly linked list, both scenarios are similar, so we will handle them in a separate private method.

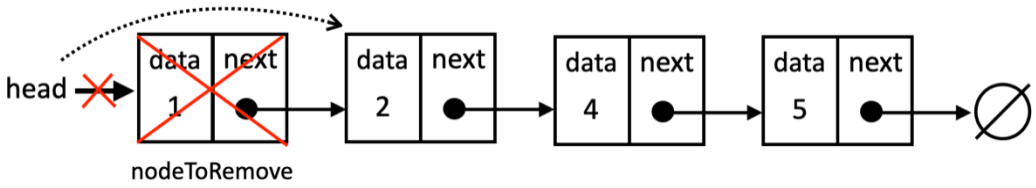
While removing a node from a linked list might seem intricate, breaking down the problem into smaller, more manageable steps can simplify the process. This approach is a valuable technique not only for linked list operations but also for tackling complex tasks in various real-world scenarios.

Let's dive into the `#removeFromHead` method, which is our first scenario to remove the first element from the linked list:

```
#removeFromHead() {
  const nodeToRemove = this.#head;
  this.#head = this.#head.next;
  this.#size--;
  return nodeToRemove.data;
}
```

If the `position` is 0 (indicating the head), we first store a reference to the head node in `nodeToRemove`. Then, we simply shift the `head` pointer to its `next` node, effectively disconnecting the original `head`. Finally, we decrease the list size and return the data of the removed node.

The following diagram exemplifies this action:



Removing the element at the head of the linked list

Next, let's check the code to remove a node from the middle or from the end of a linked list:

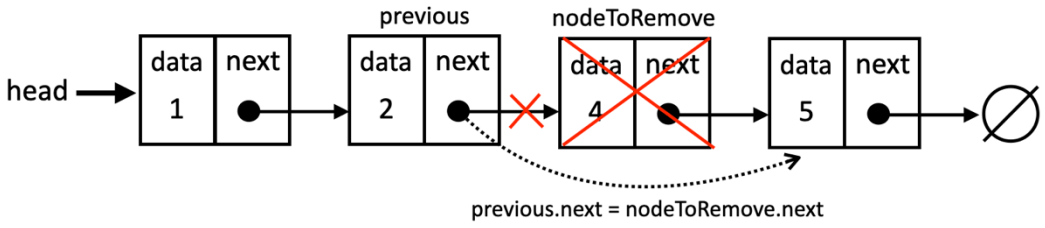
```
#removeFromMiddleOrEnd(position) {
  let nodeToRemove = this.#head;
  let previous;
  for (let index = 0; index < position; index++) {
    previous = nodeToRemove;
    nodeToRemove = nodeToRemove.next;
  }
  // unlink the node to be removed
  previous.next = nodeToRemove.next;
  this.#size--;
  return nodeToRemove.data;
}
```

For any position other than 0, we need to traverse the list to find the node to remove. In previous sections, we used the `while` loop, and we will use the `for` loop now to demonstrate there are different ways of achieving the same result.

We keep two variables, `nodeToRemove` (starting at the first element) and `previous` to navigate through the list. At each iteration, we shift `previous` to the current node (`nodeToRemove`) and move `nodeToRemove` to the next node.

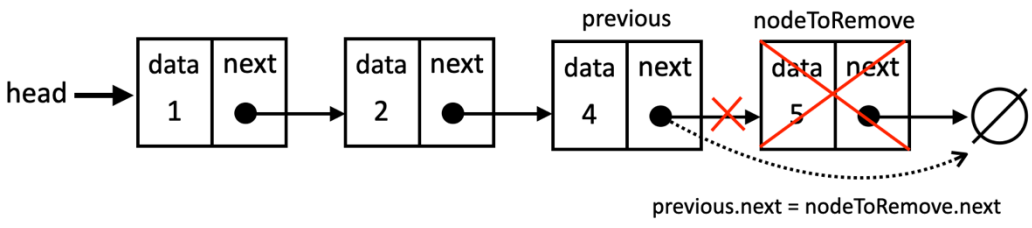
Once we reach the target position, `previous` points to the node before the one we want to remove, and `nodeToRemove` points to the node itself. We adjust the `previous` node's `next` pointer to skip over the `nodeToRemove` node, linking it directly to the node after `nodeToRemove`. This effectively removes the `nodeToRemove` node. Then we can decrement the size of the list and return the removed data.

The following diagram exemplifies removing an element from the middle of the list:



Removing the element from the middle of the linked list

The logic also works for the last element of the list, as the `nodeToRemove`'s next value will be `null` and when `previous.next` receives `null`, it automatically unlinks the last element. The following diagram exemplifies this action:



Removing the last element of the linked list

Now that we know how to remove any element from the list, let's learn how to search for a specific element and then remove it.

Searching and removing an element from the linked list

Sometimes, we need to remove an element from a linked list without knowing its exact position. In this scenario, we require a method that searches for the element based on its data and then removes it. We'll create a `remove` method that accepts the target data and an optional `compareFunction` for custom comparison logic:

```
remove(data, compareFunction = (a, b) => a === b) {
  const index = this.indexOf(data, compareFunction);
  if (index === -1) {
    return null;
  }
  return this.removeAt(index);
}
```

The method first utilizes the `indexOf` method we created earlier to determine the position (`index`) of the first node whose data matches the provided `data` using the optional `compareFunction`.

If `indexOf` returns -1, it means the element is not found in the list. In this case, we return `null`. If the element is found, the method calls `removeAt(index)` to remove the node at that position. The `removeAt` method returns the removed data, which is then returned by the `remove` method as well.

Checking if it is empty, clearing and getting the current size

The `isEmpty`, get `size` and `clear` methods are very similar to the ones we created in previous chapter and provide fundamental operations for managing our linked list. Let's look at them anyways:

```
isEmpty() {
  return this.#size === 0;
```

```
}
get size() {
  return this.#size;
}
clear() {
  this.#head = null;
  this.#size = 0;
}
```

Here is an explanation:

- `isEmpty`: this method checks if the linked list is empty. It does so by simply comparing the private `#size` property to zero. If #size is 0, it means the list has no elements and returns `true`; otherwise, it returns `false`.
- `size`: this method directly returns the value of the private `#size` property, providing the current number of elements in the linked list.
- `clear`: this method is used to completely empty the linked list. It does this by setting the `#head` pointer to `null`, effectively disconnecting all nodes. The `#size` property is also reset to 0.

Transforming the linked list into a string

The last method is the `toString` method, which its primary goal is to provide a string representation of the linked list. This is incredibly useful for debugging, logging, or simply displaying the contents of the list to users:

```
toString() {
  let current = this.#head;
  let objString = '';
  while (current) {
    objString += this.#elementToString(current.data);
    current = current.next;
    if (current) {
      objString += ', ';
    }
  }
  return objString;
}
```

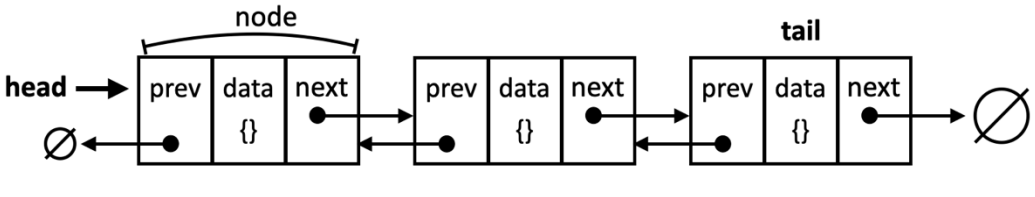
A temporary variable `current` is initialized to point to the `head` of the linked list. This will be our cursor as we traverse the list. An empty string `objString` is created to accumulate the string representation of the list.

The `while` loop continues as long as `current` is not `null`. This means we will iterate over each node in the list until we reach the end (where the last node's next property is `null`).

The `#elementToString` private method (which we have coded in previous chapters) is called to convert the data stored in the `current` node (`current.data`) into a string representation. This string is then appended (added) to the `objString`. We advance the `current` cursor to the `next` node of the list, and if there is a next node (we have not reached the end yet), a comma and a space are appended to the `objString` to separate the elements in the final string representation.

Doubly linked lists

The difference between a doubly linked list and a normal or singly linked list is that in a linked list we make the link from one node to the next one only, while in a doubly linked list, we have a double link: one for the next element and one for the previous element, as shown in the following diagram:



Doubly linked list with previous and next nodes

Let's get started with the changes that are needed to implement the `DoublyLinkedList` class. We will start by declaring the node of our doubly linked list:

```
class DoublyLinkedListNode {
  constructor(data, next = null, previous = null) {
```

```
    this.data = data;
    this.next = next;
    this.previous = previous; // new
  }
}
```

In a doubly linked list, each node maintains two references:

- `next`: a pointer to the next node in the list.
- `previous`: a pointer to the previous node in the list.

This dual linking enables efficient traversal in both directions. To accommodate this structure, we add the `previous` pointer to our `DoublyLinkedListNode` class. The constructor is designed to be flexible. By default, both the `next` and `previous` pointers are initialized to `null`. This allows us to create new nodes that are not yet connected to other nodes in the list. When inserting a node into the list, we explicitly update these pointers to establish the correct links within the list.

Next, we will declare our `DoublyLinkedList` class:

```
class DoublyLinkedList {
  #head;
  #tail; // new
  #size = 0;
  // other methods
}
```

A key distinction of a doubly linked list is that it tracks both the `head` (the first node) and the `tail` (the last node). This bidirectional linking enables us to traverse the list efficiently in either direction, offering greater flexibility compared to a singly linked list.

While the core functionality of a doubly linked list remains similar to a singly linked list, the implementation differs. In a doubly linked list, we must manage two references for each node: `next` (pointing to the following node) and `previous` (pointing to the preceding node). Therefore, when inserting or removing nodes, we need to carefully update not only the next pointers (as in a singly linked list) but also the previous pointers to maintain the correct links throughout the list.

This means that methods like `append`, `prepend`, `insert`, and `removeAt` will require modifications to accommodate this dual linking.

Let's dive into each of the modifications needed.

Appending a new element

Inserting a new element in a doubly linked list is very similar to a linked list. The difference is that in the linked list, we only control one pointer (`next`), and in the doubly linked list we need control both the `next` and `previous` references.

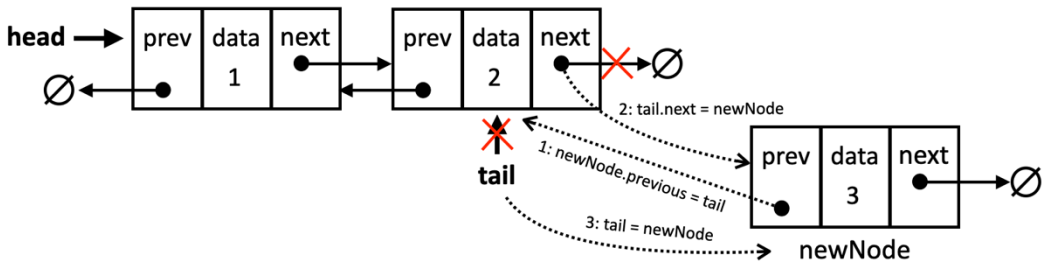
Let's see how the append method behaves in the doubly linked list:

```
append(data) {
  const newNode = new DoublyLinkedListNode(data);
  if (!this.#head) { // empty list
    this.#head = newNode;
    this.#tail = newNode;
  } else { // non-empty list
    newNode.previous = this.tail;
    this.#tail.next = newNode;
    this.#tail = newNode;
  }
  this.#size++;
}
```

When we are trying to add a new element to the end of the list, we run into two different scenarios: if the list is empty or not empty. If the list is empty (`head` is `null`), we create a new node (`newNode`) and set both the `head` and `tail` to this new node. Since it is the only node, it becomes both the start and end.

If the list is not empty, the beauty of the doubly linked list is we do not have to traverse the entire list to get to its end. As we have the `tail` reference, we can simply link the new node's `previous` pointer to the tail, then we will link the tail's next pointer to the new node, and finally, we update the `tail` reference to the new node. The order of this operations is crucial, as if we update the `tail` prematurely, we will lose the reference to the original last node, making it

impossible to correctly link the new node to the end of the list. The following diagram demonstrates the process of adding a new node to the end of the list:



Appending a new node to the doubly linked list

Next, let's review and changes needed to prepend an element to a doubly linked list.

Prepending a new element to the doubly linked list

Prepending a new element to the doubly linked list is not much different then prepending a new element to a singly linked list:

```
prepend(data) {
  const newNode = new DoublyLinkedListNode(data);
  if (!this.#head) {  // empty list
    this.#head = newNode;
    this.#tail = newNode;
  } else { // non-empty list
    newNode.next = this.#head;
    this.#head.previous = newNode;
    this.#head = newNode;
  }
  this.#size++;
}
```

Again, we have two scenarios. In case the list is empty, the behavior is identical to the `append` method, the new node becomes both the `head` and `tail`.

If the list is not empty, we set the `next` pointer of the new node (`newNode`) to the current `head`. We then update the `previous` pointer of the current head to

reference the `newNode`. Finally, we update the `head` to be the `newNode`, as it is now the first node in the list.

In a singly linked list, the prepend operation only requires updating the `next` pointer of the new node and the `head` of the list. However, in a doubly linked list, we must also update the `previous` pointer of the original head node to ensure the bidirectional links are maintained.

Now that we are able to add elements at the head and at the tail of the list, let's checkout how to insert at any position.

Inserting a new element at any position

Inserting an element at an arbitrary position within a doubly linked list requires some additional considerations compared to simply appending or prepending. Let's see how to insert at any position:

```
insert(data, position) {
  if (this.isInvalidPosition(position)) {
    return false;
  }
  if (position === 0) { // first position
    this.prepend(data);
    return true;
  }
  if (position === this.#size) { // last position
    this.append(data);
    return true;
  }
  // middle position
  return this.#insertInTheMiddle(data, position);
}
```

Let's review case by case:

> First, we start by checking if the position if valid, and if not, we return `false` to indicate the insertion was not successful.

Next, we will check if the insertion is at the `head`, and if so, we can reuse the `prepend` method, and return `true` to indicate the insertion was a success.

The next scenario is in case the insertion is at the end of the list, and if so, we can reuse the `append` method and return `true`. Checking for this case will avoid traversing the list to get to its end.

If not prepending and not appending, it means the position is in the middle of the list, and for this case, we will create a private method that will hold the logic.

In case the position is in the middle, we will use the `#insertInTheMiddle` to help us organize the steps better as follows:

```
#insertInTheMiddle(data, position) {
  const newNode = new DoublyLinkedListNode(data);
  let currentNode = this.#head;
  let previousNode;
  for (let index = 0; index < position; index++) {
    previousNode = currentNode;
    currentNode = currentNode.next;
  }
  newNode.next = currentNode;
  newNode.previous = previousNode;
  currentNode.previous = newNode;
  previousNode.next = newNode;
  this.#size++;
  return true;
}
```

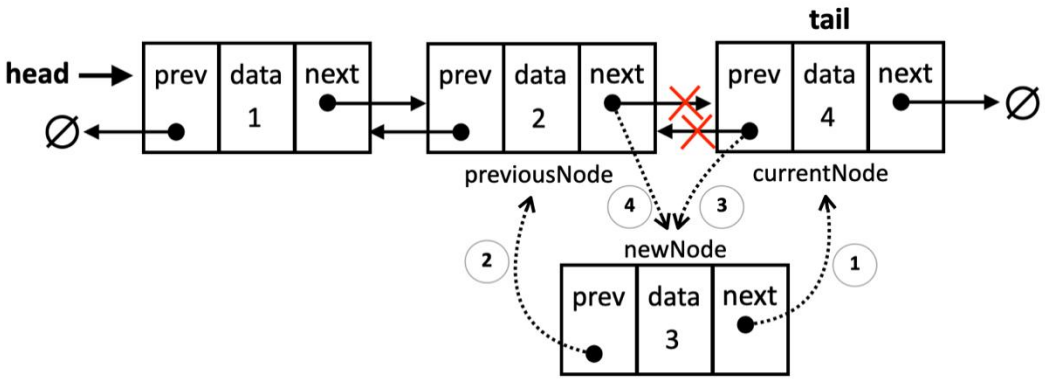
So, we will create a new node and we will traverse the list to the desired position. After the `for` loop, we will insert the new node between the previous and current references with the following steps:

The `newNode`'s `next` pointer is set to the `currentNode`.

The `newNode`'s `previous` pointer is set to the `previousNode`. With these two steps, we have the new node partially inserted to the list. What we need to do now is updating the references of the existing nodes in the list to the new node.

The `currentNode`'s `previous` pointer is updated to point to the `newNode`.

The `previousNode`'s `next` pointer is updated to point to the `newNode`.
The following diagram exemplifies this scenario:



Inserting a new node in the middle of the doubly linked list

Given we have a refence to both the `head` and the `tail` of the list, an improvement we could make to this method is checking if position is greater than size/2, then it would be best to iterate from the end than start from the beginning (by doing so, we will have to iterate through fewer elements from the list).
Now that we have learned the details of how to handle two pointers for inserting nodes in the list, let's see how to remove an element from any position.

Removing an element from a specific position

Let's delve into the details and differences of removing an element from any position of the list:

```
removeAt(position) {
  if (this.#size === 0) {
    throw new RangeError('Cannot remove from an empty list.');
  }
  if (this.#isInvalidPosition(position)) {
    throw new RangeError('Invalid position.');
  }
  if (position === 0) {
    return this.#removeFromHead();
  }
  if (position === this.#size - 1) {
```

```
    return this.#removeFromTail();
  }
  return this.#removeFromMiddle(position);
}
```

We will start by checking if the list is empty and for an invalid position. If the list is empty or if the given position is outside the valid range of the list (0 to size-1), we throw a `RangeError`.

Next, we will check for the three possible scenarios:

- If the removal is from the head (first position of the list)
- If the removal is from the tail (last position of the list)
- or from the middle of the list.

Let's dive into each scenario.

The first scenario is if are removing the first element. Following is the code for the `#removeFromHead` private method:

```
#removeFromHead() {
  const nodeToRemove = this.#head;
  this.#head = nodeToRemove.next;
  if (this.#head) {
    this.#head.previous = null;
  } else {
    this.#tail = null; // List becomes empty
  }
  this.#size--;
  nodeToRemove.next = null;
  return nodeToRemove.data;
}
```

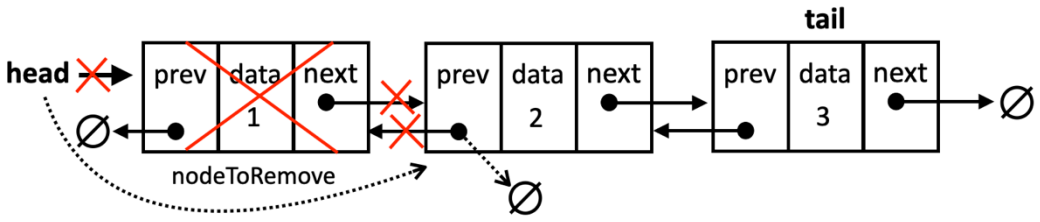
We start by creating a reference (`nodeToRemove`) to the current head node. This is important because we will need to return its data later. Next, the `head` reference is now moved to the `next` node in the list.

If there was only one node, `nodeToRemove.next` will be `null`, and the `head` will become `null`, indicating an empty list.

If the list is not empty after the removal, the `previous` reference of the new `head` node (which was previously the second node) is set to `null`, since it is now the first node and has no predecessor.

If the list is empty, both `head` and `tail` need to be set to `null`. As we have already set the `head` to `null` in the second line of the method, we only need to set the `tail` to `null`.

Finally, we remove the `nodeToRemove` next reference in case there is any. The following diagram demonstrates this scenario:



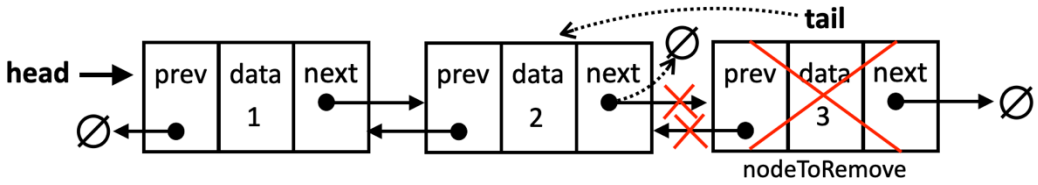
Removing the node at the head of a doubly linked list

The next scenario is checking if we are removing the tail (the last element). Following is the code for the `# removeFromTail` private method:

```
#removeFromTail() {
  const nodeToRemove = this.#tail;
  this.#tail = nodeToRemove.previous;
  if (this.#tail) {
    this.#tail.next = null;
  } else {
    this.#head = null; // List becomes empty
  }
  this.#size--;
  nodeToRemove.previous = null;
  return nodeToRemove.data;
}
```

After creating the reference to the current `tail` node, the `tail` reference is moved to the `previous` node. We need to check if the list if empty after the removal – this is similar to the `removeFromHead` method behavior. If the list is not

empty, we set the new tail's `next` pointer is set to `null`. If the list is empty, we also update the `head` to `null`. The following diagram demonstrates this scenario:



Removing the node at the tail of a doubly linked list

The last scenario is removing the element from the middle of the list. The method `#removeFromMiddle` is listed as follows:

```
#removeFromMiddle(position) {
    let nodeToRemove = this.#head;
    let previousNode;
    for (let index = 0; index < position; index++) {
      previousNode = nodeToRemove;
      nodeToRemove = nodeToRemove.next;
    }

    previousNode.next = nodeToRemove.next;
    nodeToRemove.next.previous = previousNode;
    nodeToRemove.next = null;
    nodeToRemove.previous = null;
    this.#size--;
    return nodeToRemove.data;
  }
```

Since the position is not the `head` or `tail`, we need to traverse the list to find the node and correctly adjust the surrounding nodes' references. We start by declaring a `nodeToRemove`, referencing it to the `head` as the starting point and we will move this node through the list as we iterate. The `previousNode` keeps track of the node just before the `nodeToRemove` and it starts with `null` since the head has no previous node.

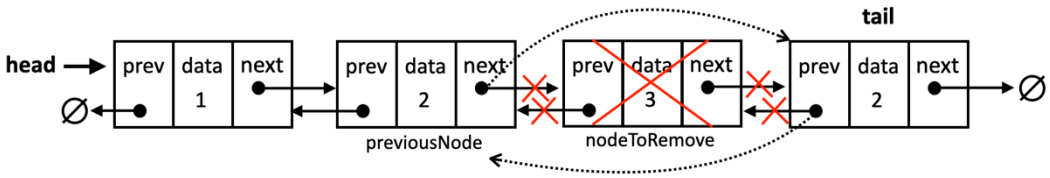
The `for` loop continues until `index` matches the `position` we want to remove from. Inside the loop, we update `previousNode` to be the current node (before we move it) and we mode `nodeToRemove` to the next node.

When the loop stops, `nodeToRemove` will reference the node we want to remove. So, we skip the reference to the `nodeToRemove` by making the `previousNode`'s `next` pointer point to the node after `nodeToRemove`.

Then, `nodeToRemove.next.previous = previousNode` updates the `previous` pointer of the node after `nodeToRemove` to point back to `previousNode`. This step is essential to maintain the doubly linked list's structure.

At last, we remove the `nodeToRemove` next and previous references, we decrease the size of the list and return the removed data.

The following diagram demonstrates this scenario:

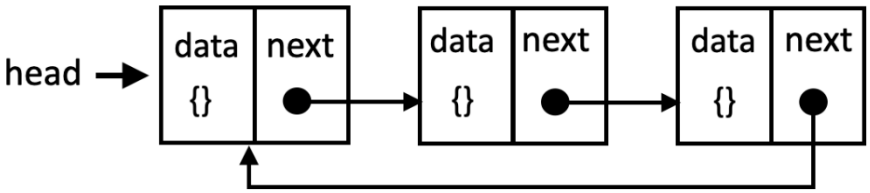


Removing the node from the middle of a doubly linked list

To check the implementation of other methods of the doubly linked list (as they are the same as the linked list), refer to the source code of the book. The download link of the source code is mentioned in the Preface of the book, and it can also be accessed at: **http://github.com/loiane/javascript-datastructures-algorithms**.

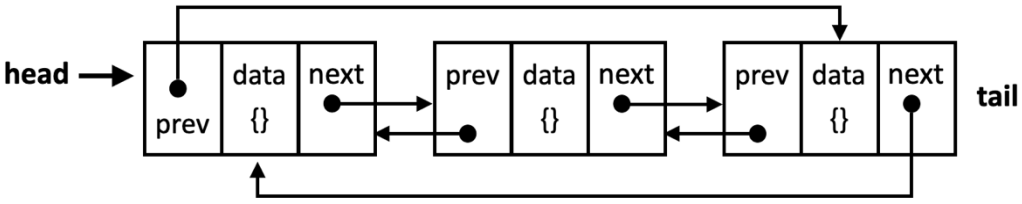
Circular linked lists

A circular linked list is a variation of a linked list where the last node's next pointer (or `tail.next`) references the first node (`head`) instead of being `null` or `undefined`. This creates a closed loop structure., as we can see in the following diagram:



The structure of a circular linked list

A doubly circular linked list has `tail.next` pointing to the head element, and `head.previous` pointing to the `tail` element as showed as follows:



The structure of a doubly circular linked list

The key difference between circular and regular (linear) linked lists is that there is no explicit *end* to a circular linked list. You can continuously traverse the list starting from any node and eventually return to the starting point.

We will implement a singly circular linked list, and you can find the bonus source code for a doubly circular linked list in the source code from this book.
Let's check the code to create the `CircularLinkedList` class:

```
class CircularLinkedList {
  #head;
  #size = 0;
  // other methods
}
```

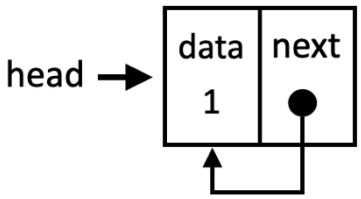
We will utilize the same `LinkedListNode` structure for our `CircularLinkedList` class, as the fundamental node structure remains the same. However, the circular nature of the list introduces some key differences in how we implement operations like `append`, `prepend`, and `removeAt`. Let's explore these modifications in detail.

Appending a new element

Appending a new element to a circular linked list shares similarities with appending to a standard linked list, but with a few key distinctions due to the circular structure. Let's see the code for the `append` method:

```
append(data) {
   const newNode = new LinkedListNode(data);
   if (!this.#head) { // empty list
     this.#head = newNode;
     newNode.next = this.#head; // points to itself
   } else { // non-empty list
     let current = this.#head;
     while (current.next !== this.#head) {
       current = current.next;
     }
     current.next = newNode;
     newNode.next = this.#head; // circular reference
   }
   this.#size++;
 }
```

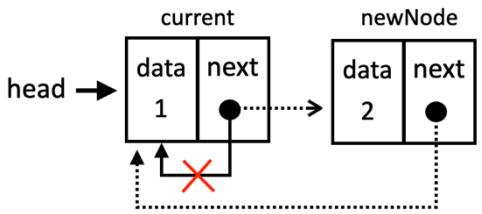
We start by creating the new node to hold that data. Then checks if the list is empty. If so, the new node becomes the `head`, and its next pointer is set to point back to itself, completing the circle. The following diagram exemplifies the first scenario:



Appending an element as the only node in a circular linked list

If the list is not empty, we need to find the last node. This is done by traversing the list starting from the `head` until we find a node whose next pointer points back to the `head`.

Once the last node (`current`) is found, its next pointer is updated to reference the `newNode`. The `newNode`'s next pointer is then set to the `head`, re-establishing the circular link. The following diagram exemplifies the second scenario:



Appending an element in a non-empty circular linked list

Next, let's see how to insert a new node in the beginning of a circular linked list.

Prepending a new element

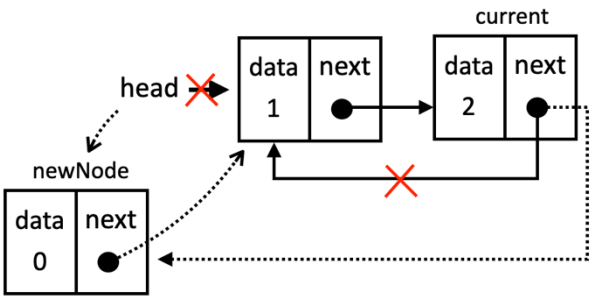
Prepending a new element in a circular linked list involves a few key steps due to the circular nature of the structure. Let's check out the code first:

```
prepend(data) {
  const newNode = new LinkedListNode(data, this.#head);
  if (!this.head) {
    this.head = newNode;
    newNode.next = this.head; // make it circular
  } else {
    // Find the last node
    let current = this.head;
    while (current.next !== this.head) {
      current = current.next;
    }
    current.next = newNode;
    this.head = newNode;
  }
  this.#size++;
}
```

We start by creating a new node to hold the data. The `next` pointer of the new node is immediately set to the current `head` of this list, to maintain the circular nature of the list after the insertion.

If the list is empty, the new node becomes the `head` and we add a self-reference as the next pointer.

In case the list is not empty, we find the last element, so we can update its `next` pointer to the new node, which will become the new `head`. The diagram below exemplifies this action:



Prepending an element in a non-empty circular linked list

If we want to insert a new element in the middle of the list, the code is the same as the `LinkedList` class since no changes will be applied to the last or first nodes of the list.

Removing an element from a specific position

For the removal of an element of a circular linked list, we will cover removing the first and the last elements, since removing an element from the middle is the same behavior as the singly linked list.

First, let's cover removing from the head, with the code as follows:

```
#removeFromHead() {
   const nodeToRemove = this.#head;
   let lastNode = this.#head;
   while (lastNode.next !== this.#head) { // Find the last node
     lastNode = lastNode.next;
   }
   this.#head = nodeToRemove.next; // skip the head
   lastNode.next = this.#head;  // make it circular
```

```
    if (this.#size === 1) {  // only one node
      this.#head = null;
    }
    this.#size--;
    return nodeToRemove.data;
  }
```

Following is the explanation:

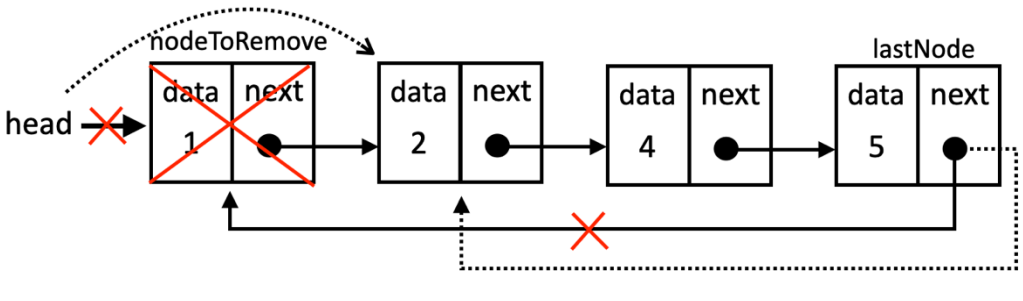
First, we traverse the list to find the last node.

Next, we set the head to the next node to remove the first element.

Then we make the last node point to the new `head`, closing the circle.

Finally, if the removed node was the only one in the list, set the `head` to `null` to reset it.

The diagram below exemplifies this action:



Removing the head from a circular linked list

Now, let's check out how to remove from the end of the list:

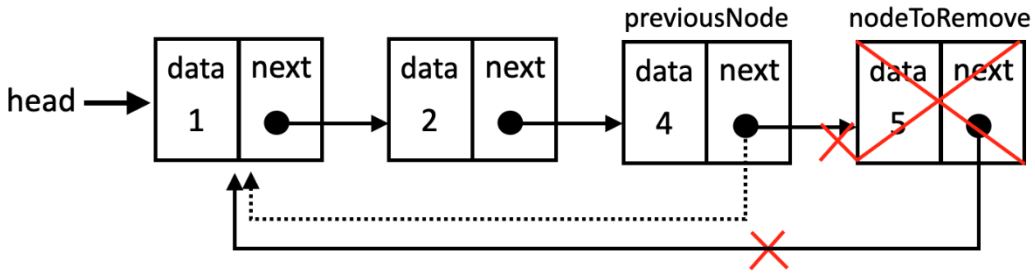
```
#removeFromTail() {
  if (this.#head.next === this.#head) { // single node case
    const nodeToRemove = this.#head;
    this.#head = null;
    this.#size--;
    return nodeToRemove.data;
  } else {
    let lastNode = this.#head;
    let previousNode = null;
    while (lastNode.next !== this.#head) { // Find the last node
      previousNode = lastNode;
      lastNode = lastNode.next;
```

```
    }
    previousNode.next = this.#head; // skip the last node to remove it
    this.#size--;
    return lastNode.data;
  }
}
```

And following is the explanation:

> If there is only one node (`head` points to itself), removing it makes the list empty. We update `head` to `null` and return the removed data.
> If the list is not empty, we need to find the last node and the second last node (`previousNode`). So we traverse the list until we reach its end, updating the `previous` and the `last` node references.
> When the `while` loop is finished, the `lastNode` is the one we want to remove. So we set `previousNode.next = this.#head` to make the second last node point to the head, skipping the last node, which is now removed.
> We decrement the list size and return the removed `data`.

The diagram below exemplifies this action:



Removing the tail from a circular linked list

Now that we know how to add and remove elements from three different types of linked lists, let's put all these concepts learned by creating a fun project!

Creating a media player using a linked list

To solidify our understanding of linked lists and explore new concepts, let's build a real-world application: a media player! This project will leverage the linked list structure and introduce us to additional techniques like working with doubly circular linked lists and ordered insertion.

Before we dive in, let's outline the core features of our media player:

Ordered song insertion: add songs to the playlist based on song title.
Sequential playback: simulate playing songs one after the other in the playlist's order.
Navigation: easily jump to the previous or next song.
Continuous repeat: automatically loop the playlist, playing songs repeatedly.

The following image represents the media player we will develop using linked lists:

A media player using doubly circular linked list

To model our media player's playlist, we will use a custom node structure to represent each song. Following is the `MediaPlayerSong` class:

```
class MediaPlayerSong {
  constructor(songTitle) {
    this.songTitle = songTitle;
    this.previous = null;
    this.next = null;
  }
}
```

Each `MediaPlayerSong` node stores:

`songTitle`: the title of the song.
`previous`: a reference to the previous song node in the playlist (or a reference to the last song, if it is the first song).
`next`: a reference to the next song node in the playlist (or a reference to the first song if it is the last song).

This will allow us to implement the continuous repeat feature of our media player.

Next, let's define the structure of our `MediaPlayer` class:

```
class MediaPlayer {
  #firstSong;
  #lastSong;
  #size = 0;
  #playingSong;
  // other methods
}
```

The `MediaPlayer` class maintains:

- `#firstSong`: a reference to the first `MediaPlayerSong` in the playlist.
- `#lastSong`: a reference to the last `MediaPlayerSong` in the playlist.
- `#size`: the count of songs currently in the playlist.
- `#playingSong`: a reference to the `MediaPlayerSong` that is currently being played.

Before we can start playing songs, we need to be able to add songs to our playlist. Let's see how to do it in the next section.

Adding new songs by title order (sorted insertion)

To maintain an alphabetically ordered playlist, we will implement a method called `addSongByTitle`. This method will insert a new song into the correct position based on its title, ensuring the playlist remains sorted.

Behind the scenes, we are doing a sorted insertion in a doubly circular linked list!
We will start by declaring the method to insert a new song:

```
addSongByTitle(newSongTitle) {
  const newSong = new MediaPlayerSong(newSongTitle);
  if (this.#size === 0) { // empty list
    this.#insertEmptyPlayList(newSong);
  } else {
```

```
    const position = this.#findIndexOfSortedSong(newSongTitle);
    if (position === 0) { // insert at the beginning
      this.#insertAtBeginning(newSong);
    } else if (position === this.#size) { // insert at the end
      this.#insertAtEnd(newSong);
    } else { // insert in the middle
      this.#insertInMiddle(newSong, position);
    }
  }
  this.#size++;
}
```

Here is a brief explanation before we start diving into the details:

> We start by creating a new `MediaPlayerSong` node with the given `newSongTitle`.
> If the playlist is empty, we call a private method `#insertEmptyPlayList` to handle the insertion of the first song.
> For non-empty playlists, we callsa private method `#findIndexOfSortedSong` to determine the correct `position` where the new song should be inserted to maintain alphabetical order.
> Based on the returned `position`, the method dispatches the insertion to one of three private methods:
> `#insertAtBeginning`: inserts the new song at the start of the list.
> `#insertAtEnd`: inserts the new song at the end of the list.
> `#insertInMiddle`: inserts the new song in the middle of the list at the specified position.
> Finally, the `#size` of the playlist is incremented to reflect the addition of the new song.

Let's review each of the steps with more details.

Inserting into an empty playlist

In this scenario, we are handling an insertion into a doubly circular linked list. Let's dive into the `#insertEmptyPlayList` method:

```
#insertEmptyPlayList(newSong) {
  this.#firstSong = newSong;
  this.#lastSong = newSong;
  newSong.next = newSong; // points to itself
  newSong.previous = newSong; // points to itself
}
```

We are assigning the new song to the `firstSong` (head) and the `lastSong` (tail). And to keep the circular references, we set the new song's `next` and `previous` references to itself.

The next step in the logic is to find the position we need to insert the song in case the playlist is not empty.

Finding the alphabetically sorted insertion position

We are dealing with a complex scenario: sorted insertion into a doubly circular linked list. To simplify this, we will tackle it in two phases, the first one determining the correct position (index) where the new song should be inserted to maintain alphabetical order and the second one being the insertion itself. So, for now, let's focus on finding the correct insertion index:

```
#findIndexOfSortedSong(newSongTitle) {
  let currentSong = this.#firstSong;
  let i = 0;
  for (; i < this.#size && currentSong; i++) {
    const currentSongTitle = currentSong.songTitle;
    if (this.#compareSongs(currentSongTitle, newSongTitle) >= 0) {
      return i;
    }
    currentSong = currentSong.next;
  }
  return 0;
}
```

We will traverse the list to find the position of the insertion. To do so, we will use a cursor called `currentSong`. We also need an index counter `i`.

We will loop from the first song up until the last song of the playlist. Inside the loop, we will call a helper method that contains the logic to compare the songs. If the result of the helper method is 0 (duplicate song) or a positive number, it means we found the position.

If the new song does not belong at the current position, we move to the next song in the list. If the loop completes without returning, it means the new song should be inserted at the beginning (index 0).

Next let's check the code for the `#compareSongs` method:

```
#compareSongs(songTitle1, songTitle2) {
  return songTitle1.localeCompare(songTitle2);
}
```

This method is a helper function used to compare two song titles alphabetically, considering locale-specific sorting rules. The localeCompare method returns a number that indicates the sorting relationship between the two strings:

- Negative number: `songTitle1` comes before `songTitle2` in alphabetical order.
- 0 (zero): `songTitle1` and `songTitle2` are considered equal in the current locale.
- Positive number: `songTitle1` comes after `songTitle2` in alphabetical order.

You can modify this method as needed to customize how you would like to compare song titles.

Now that we know the position we need to insert, let's review each of the methods.

Inserting at the beginning of the playlist

Let's check how to prepend a new song in a playlist that is not empty:

```
#insertAtBeginning(newSong) {
  newSong.next = this.#firstSong;
  newSong.previous = this.#lastSong;
```

```
  this.#firstSong.previous = newSong;
  this.#lastSong.next = newSong;
  this.#firstSong = newSong;
}
```

Given our next song, we will point its next reference to the first song (head) and its last reference to the last song (tail). Then, we update the existing first song's previous reference to the new song, and the last song's next reference also to the new song. Finally, we update the first song reference to the new song.

Next, let's see how to append a new song.

Inserting at the end of the playlist

Let's review how we can add new songs at the end of the playlist with the following method:

```
#insertAtEnd(newSong) {
  newSong.next = this.#firstSong;
  newSong.previous = this.#lastSong;
  this.#lastSong.next = newSong;
  this.#firstSong.previous = newSong;
  this.#lastSong = newSong;
}
```

Given the new song, when inserting at the end, we need to link its next reference to the first song and its previous reference to the last song so we can keep the doubly circular references. Then, we need to update the last song's next reference to the new song, the first song's previous reference also to the new song to also keep the circular reference, and finally, update the reference of the last song to the new song.

Now, the last step is to insert a song in the middle of the playlist.

Inserting in the middle of the playlist

Now that we have covered the insertion at the head and at the tail of the doubly circular linked list, let's dive into the details to insert a new element in the middle of the list as follows:

```
#insertInMiddle(newSong, position) {
  let currentSong = this.#firstSong;
  for (let i = 0; i < position - 1; i++) {
    currentSong = currentSong.next;
  }
  newSong.next = currentSong.next;
  newSong.previous = currentSong;
  currentSong.next.previous = newSong;
  currentSong.next = newSong;
}
```

Inserting in the middle is the same as inserting in a doubly linked list. As we have both the previous and next references, we do not need two references. So first, we find the position we are looking for, and we stop at one position before the one we want. Then, we link the new song's next reference to the current's next reference, and the new song's previous reference to the current song. With this step, we have inserted the new song in the list, and now we need to fix the remaining links. So, we fix the current songs' next node's previous reference to the new song, and the current song's next reference to the new song.

With the songs added to the playlist, we can start playing them!

Playing a song

When we select the play song feature of our media player, the goal is to start playing the song. For our simulation, it means assigning the first song to the playing song reference, as described as follows:

```
play() {
  if (this.#size === 0) {
    return null;
```

```
  }
  this.#playingSong = this.#firstSong;
  return this.#playingSong.songTitle;
}
```

If there are no songs in the playlist, we can return null, or if you prefer, you can throw an error as well. Then, we assign the reference of the playing song to the first song, and we return the title we are playing.

Playing the next or previous song

The behavior to play the next or previous song are very similar. The difference is on the reference we are updating: `previous` or `next`. Let's review the behavior for playing the next song first:

```
next() {
  if (this.#size === 0) {
    return null;
  }
  if (!this.#playingSong) {
    return this.play();
  }
  this.#playingSong = this.#playingSong.next;
  return this.#playingSong.songTitle;
}
```

If there are no songs in the playlist, we return `null`. Also, if there is not any song playing at the moment, we play the first song. However, if there are songs playing and we decide we want to play the next song, we simply update the playing song with its next reference and we return the song title.

The code of the previous method is also very similar:

```
previous() {
  if (this.#size === 0) {
    return null;
  }
  if (!this.#playingSong) {
    return this.play();
```

```
  }
  this.#playingSong = this.#playingSong.previous;
  return this.#playingSong.songTitle;
}
```
The difference is if there are songs playing, and we want to play the previous song, we update the current song with its previous reference.

In both cases, when we reach the end of the playlist, or the first song of the playlist, we can keep playing, because of the circular doubly references.

Using our media player

Now that we have built our media player, let's put it to the test. We will start by creating an instance and adding our favorite songs:

```
const mediaPlayer = new MediaPlayer();
mediaPlayer.addSongByTitle('The Bard\'s Song');
mediaPlayer.addSongByTitle('Florida!!!');
mediaPlayer.addSongByTitle('Run to the Hills');
mediaPlayer.addSongByTitle('Nothing Else Matters');
```
After our playlist is created, we can start playing songs and seeing the output:

```
console.log('Playing:', mediaPlayer.play()); // Florida!!!
```
We can select the next song multiple times and check that the continuous playback works as follows:

```
console.log('Next:', mediaPlayer.next()); // Nothing Else Matters
console.log('Next:', mediaPlayer.next()); // Run to the Hills
console.log('Next:', mediaPlayer.next()); // The Bard's Song
console.log('Next:', mediaPlayer.next()); // Florida!!!
```
And if we go the other way around it, selecting the previous button:

```
console.log('Previous:', mediaPlayer.previous()); // The Bard's Song
console.log('Previous:', mediaPlayer.previous()); // Run to the Hills
console.log('Previous:', mediaPlayer.previous()); // Nothing Else
Matters
console.log('Previous:', mediaPlayer.previous()); // Florida!!!
```

If we review the output, we can confirm the songs were inserted in an alphabetical order.
Have fun playing with our media player!

Reviewing the efficiency of the linked lists

Let's review the efficiency of each method by review the Big O notation in terms of time of execution:

Method	Singly	Doubly	Circular	Explanation
`append`	*O(n)*	*O(1)*	*O(n)*	In singly and circular lists, we must traverse to the end to append. Doubly lists have a tail reference for constant time append.
`prepend`	*O(1)*	*O(1)*	*O(n)*	All lists can add a new node as the head directly. However, in circular lists, we must update the last node's next pointer to the new head.
`insert`	*O(n)*	*O(n)*	*O(n)*	For all types, we need to traverse to the position to insert.
`removeAt`	*O(n)*	*O(n)*	*O(n)*	Similar to insertion, traversal to the position is required. Doubly lists have an optimization when removing the tail (*O(1)*), but this is less common than removing from an arbitrary position.
`remove`	*O(n)*	*O(n)*	*O(n)*	Searching for the data takes *O(n)* in all cases, then removal itself is either *O(1)* (if the node is found at the head) or *O(n)* (traversing to the node).
`indexOf`	*O(n)*	*O(n)*	*O(n)*	In the worst case, we might need to traverse the entire list to find the data or determine it's not present.
`isEmpty`	*O(1)*	*O(1)*	*O(1)*	Checking if the list is empty is a simple size reference check.

`size`	*O(1)*	*O(1)*	*O(1)*	The size is tracked as a property and directly accessible.
`clear`	*O(1)*	*O(1)*	*O(1)*	Clearing a list simply involves resetting the head pointer (and tail in doubly linked lists), which is a constant-time operation.
`toString`	*O(n)*	*O(n)*	*O(n)*	Building a string representation requires visiting each node.

Doubly linked lists often have a performance advantage in append due to the tail pointer. Otherwise, all three list types have similar time complexities for most operations, as they all involve some degree of traversal.

Space complexity is *O(n)* for all three types, as the space used is proportional to the number of elements stored.

If we had to compare liked lists to arrays, there are pros and cons to each data structure. Let's review a few key points:

Linked Lists: prefer linked lists when:
You need a dynamic collection where the number of elements changes frequently.
You perform frequent insertions and deletions, especially at the beginning or middle of the list.
You do not require random access to elements.
Arrays: prefer arrays when:
You know the maximum size of the collection beforehand.
You need fast random access to elements by index.
You primarily need to iterate through the elements sequentially.

We have also learned about stacks, queues and deques earlier in this book and we have used arrays internally. These data structures can also be implemented using linked lists. So, what is the best implementation for each? We need to consider these factors when deciding:

Frequency of operations: if you frequently need to access elements by index (random access), arrays might be a better choice. If insertions and deletions at the beginning or middle are common, linked lists could be more suitable.
Memory constraints: if memory is a significant concern and you know the maximum size of your data structure beforehand, arrays might be more memory efficient. However, if the size is highly variable, linked lists can save space by not reserving unused memory.
Simplicity versus flexibility: array implementations are often simpler to code. Linked lists offer more flexibility for dynamic resizing and efficient modifications.

When it comes to answer, it all depends on what operations we will perform the most (and where) and the space we need to store our data.

For stacks and queues, array implementations are often the default choice due to their simplicity. However, if you need to implement a queue with very frequent operations such as push/pop and queue/dequeue, a doubly linked list where we have the head and tail references might be more efficient. For deques, doubly linked lists are a natural fit, as they allow efficient insertion and removal at both ends at constant time.

Since we've covered linked lists, a versatile dynamic data structure, put your knowledge to the test! Try re-implementing classic data structures like stacks, deques, and queues using linked lists instead of arrays. This hands-on exercise will deepen your understanding of both linked lists and these abstract data types. Plus, you can compare the performance and characteristics of your linked list-based versions with their array-based counterparts. For reference, you'll find these linked list implementations within the source code accompanying this book.
Let's put our knowledge into practice with some exercises.

Exercises

We will resolve one exercise from **LeetCode** so we can learn another concept we have not covered in this chapter so far.

However, there are many fun linked list exercises available in LeetCode that we should be able to resolve with the concepts we learned in this chapter. Below are some additional suggestions you can try to resolve, and you can also find the solution along with the explanation within the source code from this book:

2. Add Two Numbers: traverse two linked list and sum each number.
62. Rotate List: remove nodes from the tail and prepend them in the list.
203. Remove Linked List Elements: traverse the list and check for the value that needs to be removed. Tip: keep a previous reference to make the removal easier.
234. Palindrome Linked List: check if the elements of the list are a palindrome.
642. Design Circular Deque: implement the deque data structure.
622. Design Circular Queue implement the queue data structure.

Reverse Linked List

The exercise we will resolve the is the *206. Reverse Linked List* problem available at **https://leetcode.com/problems/reverse-linked-list/description/**.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function `function reverseList(head: ListNode | null): ListNode | null`, which receives the head of a linked list and is expecting a node that represents the head of the reverse list. The `ListNode` class consists of a `val` (number), and the `next` pointer.

Let's write the `reverseList` function:

```
function reverseList(head: ListNode | null): ListNode | null {
  let current = head;
  let newHead = null;
  let nextNode = null;
  while (current) {
    nextNode = current.next;
    current.next = newHead;
    newHead = current;
    current = nextNode;
```

```
  }
  return newHead;
}
```

To better understand what is happening in the code, let's use some diagrams. We will use the example provided by the exercise, which is a linked list with the following values: [1, 2, 3, 4, 5], and is expecting the following result: [5, 4, 3, 2, 1].

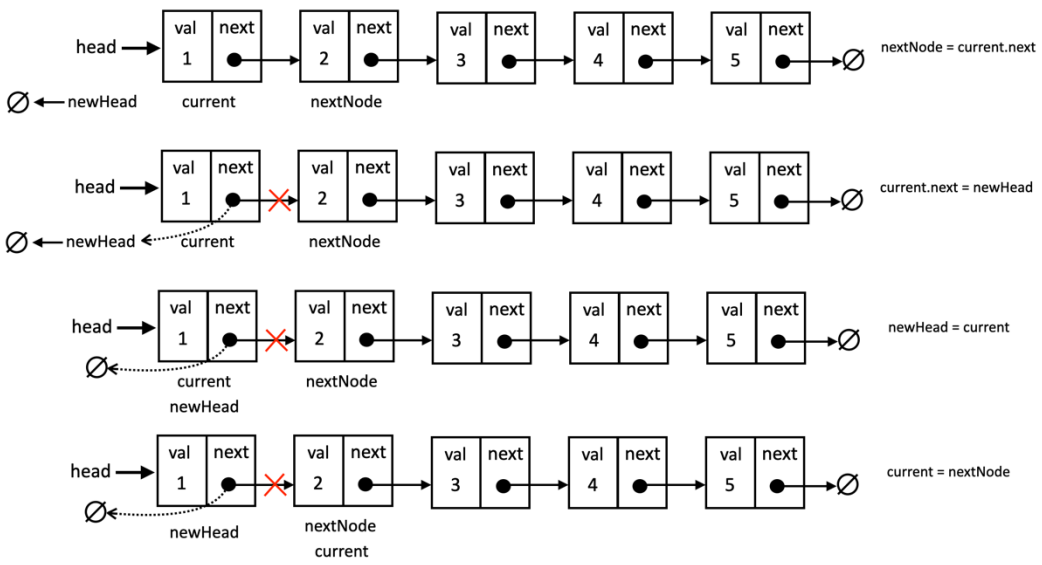
For this exercise, we will use three variables:

- `current` points to the head of the list.
- `newHead` starts as `null`, representing the new head of the reversed list. It is also the variable we will return as a result of the function.
- `nextNode` is a temporary cursor for the next node in the original list.

The logic consists only of a loop, which will traverse the entire list. Inside the loop we have four important operations:

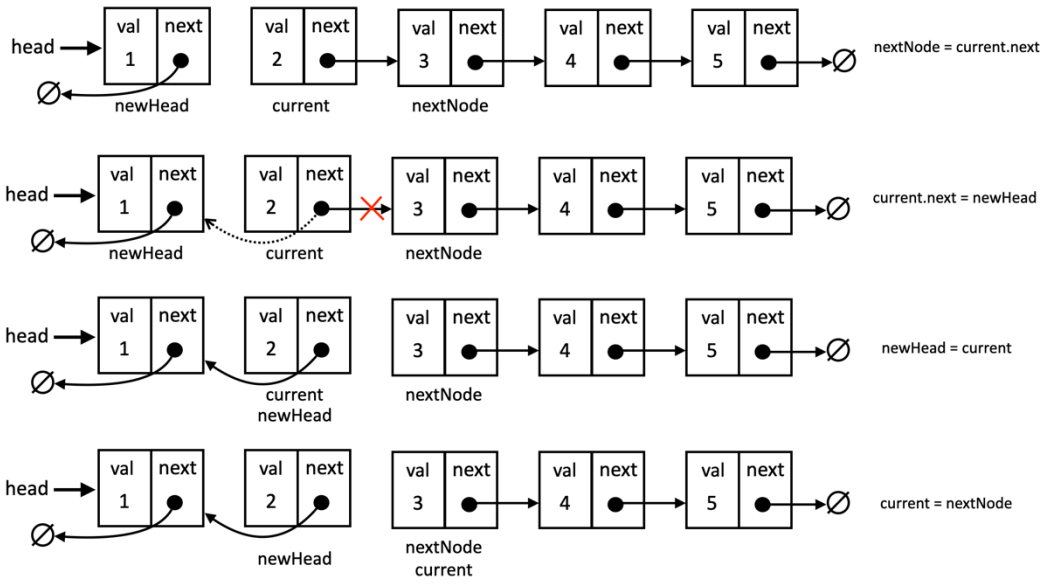
- `nextNode = current.next`: saves the next node before we modify the current node's links.
- `current.next = newHead`: reverses the current node's link to point to the previous node (which is now `newHead`).
- `newHead = current`: moves the `newHead` one step forward, making the `current` node the new head.
- `current = nextNode`: moves current to the next node (which was previously stored in `nextNode`).

After the first pass inside the loop, this is how the list will look like:



The reverse linked list after the first pass inside the while loop

After the second pass on the loop, this is how the list will look like:



The reverse linked list after the second pass inside the while loop

And the process continues until `current` is `null` and the list is reversed. This solution passes all the tests and resolves the problem.

The time complexity of this function is *O(n)*, where *n* is the number of nodes we have in the list. The space complexity is *O(1)*, because we only used the additional

variables to track the nodes and we are not using any additional space, as our solution reverses the linked list in place.

Go back to the `LinkedList`, `DoublyLinkedList` and `CircularLinkedList` classes and create a method to reverse each list in place, following a similar logic we used to resolve this exercise. You will also find this method in the source of this book.

Summary

This chapter explored linked lists and their variations: singly, doubly, and circular. We covered insertion, removal, and traversal techniques, highlighting the advantage of linked lists over arrays for frequent element additions and removals due to their dynamic nature.

To solidify our knowledge, we built a media player, applying concepts like doubly circular and sorted linked lists. We also solved a LeetCode challenge, reversing linked lists in place for an added twist.

Get ready! Next up, we dive into sets, a unique data structure for storing distinct elements.

7 Sets

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).

https://packt.link/EarlyAccess/

Building on your knowledge of sequential data structures, this chapter introduces you to the unique world of **sets**, a collection that stores only distinct values. We will cover the fundamentals of creating sets, adding or removing elements, and efficiently checking for membership. We will then discover how to leverage the power of sets with mathematical operations like union, intersection, and difference. To make things even easier, we will explore JavaScript's built-in `Set` class, providing you with a convenient tool for working with sets directly. So, in this chapter, we will cover:

- Creating a Set class from scratch
- Performing mathematical operations with a set
- JavaScript native Set class
- Exercises

The set data structure

A **set** is a fundamental concept in mathematics and computer science. It is an unordered collection of distinct items (elements). Think of it as a bag where you can put things in, but the order you put them in does not matter, and you cannot have duplicates.

Sets are a fundamental concept in mathematics and computer science, with numerous real-world applications across various fields.

Let's take a look at the mathematical concept of sets before we dive into the computer science implementation of it. In mathematics, a set is a collection of

distinct objects. For example, we have a set of natural numbers, which consists of integer numbers greater than or equal to 0 - that is, N = {0, 1, 2, 3, 4, 5, 6, ...}. The list of the objects within the set is surrounded by {} (curly braces).

There is also the null set concept. A set with no element is called a **null set** or an **empty set**. An example would be a set of prime numbers between 24 and 29. Since there is no prime number (a natural number greater than 1 that has no positive divisors other than 1 and itself) between 24 and 29, the set will be empty. We will represent an empty set with { }.

In mathematics, a set also has some basic operations such as union, intersection, and difference. We will also cover these operations in this chapter.

In computer science for example, sets are used to model relationships between data and to perform operations like filtering, sorting, and searching. Sets are also extremely useful to remove duplicate elements from other collections such as lists.

You can also imagine a set as an array with no repeated elements and no concept of order.

Creating the MySet class

ECMAScript 2015 (ES6) introduced the native Set class to JavaScript, providing a built-in and efficient way to work with sets. However, understanding the underlying implementation of a set is crucial for grasping data structures and algorithms. We will delve into creating our own custom `MySet` class that mirrors the functionality of the native Set, but with additional features like union, intersection, and difference operations.

Our implementation will reside in the `src/07-set/set.js` file. We will start by defining the `MySet` class:

```
class MySet {
  #items = {};
  #size = 0;
}
```
We chose the name `MySet` to avoid conflicts with the native `Set` class. We utilize an object (`{}`) instead of an array to store elements within the `#items` private property. The keys of this object represent the set's unique values, while the corresponding values can be anything (we will use true as a simple placeholder). This choice leverages the fact that JavaScript objects cannot have duplicate keys, naturally enforcing the uniqueness of set elements. Arrays could also be used, but they require additional logic to prevent duplicates and might have slightly slower lookups in some cases. In other languages, this data structure (using a hash table-like approach) is often referred to as a **hash set**. We will also keep track of the number of elements in the set with the property `size`.

Next, we need to declare the methods available for a set:

The `MySet` class will provide the following methods:

- `add(value)`: adds a unique value to the set.
- `delete(value)`: removes the value from the set if it exists.
- `has(value)`: returns true if the element exists in the set and false otherwise.
- `clear()`: removes all the values from the set.
- `size()`: returns how many values the set contains.
- `values()`: returns an array of all the values of the set.
- `union(otherSet)`: combines two sets.
- `intersection(otherSet)`: finds common elements between the two sets.
- `difference(otherSet)`: finds elements unique to one set.

We will implement each of these methods in detail in the following sections.

Finding a value in the set

The first method we implement in our custom `MySet` class is `has(value)`. This method plays a crucial role as a building block for other operations like adding

and removing elements. It allows us to efficiently determine if a given value already exists within the set. Here is the implementation:

```
has(value) {
  return this.#items.hasOwnProperty(value);
}
```

The method directly utilizes JavaScript's built-in `hasOwnProperty` function on the internal `#items` object. This is a highly optimized way to check if a specific key (representing the value) exists in the object.

The `hasOwnProperty` method provides constant time complexity (*O(1)*) on average, making it an extremely fast way to check for existence within the set. This efficiency is a key reason we often prefer using objects over arrays for set implementations in JavaScript.

And now that we have this method, we can proceed with the implementation of the methods for adding and removing values.

Adding values to the set

Next, we will implement the add method in our custom `MySet` class. This method is responsible for inserting a new element into the set, but only if it's not already present (maintaining the set's uniqueness property) as follows:

```
add(value) {
  if (!this.has(value)) {
    this.#items[value] = true; // mark the value as present
    this.#size++;
    return true;
  }
  return false;
}
```

We start by efficiently checking if the value already exists within the set using the `has(value)` method we implemented earlier. If the value is not already present, we insert it into the `#items` object. We use the `value` itself as the key and assign a value of `true` to it. This serves as a simple flag indicating that the value is part of

the set. After a successful insertion, we increment the `#size` property to accurately reflect the new number of elements in the set.

We return `true` to signal that the value was successfully added (it was not already in the set). Otherwise, we return `false` to indicate that the value was not added because it was a duplicate.

Removing and clearing all values

Next, we will implement the `delete` method as follows:

```
delete(value) {
  if (this.has(value)) {
    delete this.#items[value];
    this.#size--;
    return true;
  }
  return false;
}
```

We start by checking if the specified value exists within the set using the previously implemented `has(value)` method. This ensures we only try to delete elements that are present. If the value is found, we use the `delete` operator to remove the corresponding key-value pair from the `#items` object. This directly eliminates the element from the set's internal storage. After a successful deletion, we decrease the `#size` property to maintain an accurate count of elements in the set.

We return `true` to signal that the value was successfully deleted from the set, and `false` to indicate that the value was not found in the set and therefore could not be deleted.

And if we want to remove all the elements from the set, we can use the `clear` method, as follows:

```
clear() {
  this.#items = {};
  this.#size = 0;
}
```
We achieve a complete clearing of the set by directly reassigning the `#items` object to a new, empty object `{}`. This effectively discards all previous key-value pairs (representing the set's elements) and creates a fresh, empty container for future additions. And we also reset the `#size` property back to 0 to accurately reflect that the set now contains no elements.

This implementation is extremely efficient, as reassigning the `#items` object is a constant time operation (*O(1)*). The alternative of iterating and deleting each element individually would be much slower, especially for large sets. This is generally not recommended unless we have a specific reason to track which elements are being removed during the clear operation.

Retrieving the size and checking if it is empty

The next method we will implement is the size method (technically a getter method) as follows:

```
 get size() {
    return this.#size;
  }
```
This method simply returns the size property we are using to keep count.

If we weren't tracking the `#size` property, we could determine the size of the set by:

- Iterating over the keys (elements) of the `#items` object.
- Incrementing a counter for each key encountered.

Here is the code for this alternative approach:

```
getSizeWithoutSizeProperty() {
  let count = 0;
```

```
  for (const key in this.#items) {
    if (this.#items.hasOwnProperty(key)) {
      count++;
    }
  }
  return count;
}
```

The code uses a `for...in` loop to iterate over the keys (which are the values of the set) in the `#items` object. Inside the loop, `hasOwnProperty` is used to ensure we are only counting properties that belong directly to the object (not inherited properties from the prototype chain).

This approach would be less efficient, especially for large sets, as it would involve iterating over all elements, resulting in a time complexity of *O(n)*, where *n* is the number of elements in the set

And to determine if the `MySet` is empty, we implement the `isEmpty()` method, following a pattern consistent with other data structures we have covered in this book:

```
isEmpty() {
  return this.#size === 0;
}
```

This method directly compares the private `#size` property to 0. The property `#size` is meticulously maintained to always reflect the number of elements in the set.

Retrieving all the values

To retrieve an array containing all the elements (values) within our `MySet`, we can implement the `values` method as follows:

```
values() {
  return Object.keys(this.#items);
}
```

We can leverage the built-in `Object.keys()` method for a concise implementation. This built-in JavaScript method takes an object (in our case, `this.#items`) and returns an array containing all its enumerable property keys as strings. Remember, in our `MySet` implementation, we use the keys of the `#items` object to store the actual values that are added to the set.

Now that we have completed the implementation of our custom `MySet` data structure, let's explore how to put it into action!

Using the MySet class

We will dive into practical examples that showcase the utility and flexibility of `MySet`, demonstrating how it can be used to efficiently manage collections of unique elements. Imagine we are building a blog or content management system where users can add tags (keywords) to their articles or posts. We want to ensure that each post has a list of unique tags, with no duplicates.

The source code for this example can be found in the file src/07-set/01-using-myset-class.js. Let's start by defining the article:

```
const MySet = require('./set');
const article = {
  title: 'The importance of data structures in programming',
  content: '...',
  tags: new MySet()  // using MySet to store tags
};
```

Now, let's add some tags to our article:

```
article.tags.add('programming');
article.tags.add('data structures');
article.tags.add('algorithms');
article.tags.add('programming');
```

Note that the first and last tags are duplicates. We can confirm if we have three tags in the set, meaning there are no duplicates:

```
console.log(article.tags.size);  // 3
```

We can also use the `has` method to double check which tags are part of the article:

```
console.log(article.tags.has('data structures'));  // true
console.log(article.tags.has('algorithms'));  // true
console.log(article.tags.has('programming'));  // true
console.log(article.tags.has('javascript'));  // false
```

We can also use the values method to retrieve all the tags:

```
console.log(article.tags.values());
// output: ['programming', 'data structures', 'algorithms']
```

Now, let's say we want to remove the tag programming and add the tag JavaScript instead:

```
article.tags.delete('programming');
article.tags.add('JavaScript');
console.log(article.tags.values());
// output: ['data structures', 'algorithms', 'JavaScript']
```

So, now we have a remarkably similar implementation of the Set class, as in ECMAScript 2015. But we can also enhance our implementation by adding some basic operations such as union, intersection, and difference.

Performing mathematical operations with a set

Sets are a fundamental concept in mathematics with far-reaching applications in computer science, particularly within the realm of **databases**. Databases serve as the backbone of countless applications, and sets play a crucial role in their design and operation.

When we construct a query to retrieve data from a relational database (such as Oracle, Microsoft SQL Server, MySQL, etc.), we are essentially using set notation to define the desired result. The database, in turn, returns a set of data that matches our criteria.

SQL queries allow us to specify the scope of the data we want to retrieve. We can select all records from a table, or we can narrow down the search to a specific subset based on certain conditions. Furthermore, SQL leverages set operations to

perform various types of data manipulation. The concept of *joins* in SQL is fundamentally based on set operations. Here are some common examples:

- **Union**: combining data from two or more tables to create a new set containing all unique rows.
- **Intersection**: identifying rows that are common to multiple tables, resulting in a set containing only the shared data.
- **Difference** (Except/Minus): finding rows that exist in one table but not in another, creating a set of unique rows from the first table.

And beyond the operations used in SQL, there are other essential set operations such as:

- **Subset**: determining if one set is entirely contained within another set. This helps establish relationships between sets and can be useful for various logical and analytical tasks.

Understanding sets and their operations is essential for working with databases and other data-intensive applications. The ability to manipulate sets effectively allows us to efficiently extract, filter, and analyze information from complex datasets. Let's see how we can simulate these operations using our `MySet` class.

Union: combining two sets

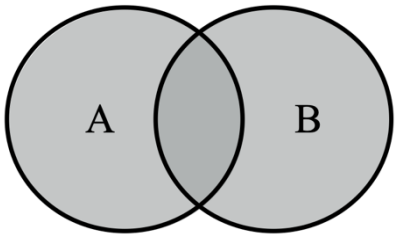
The union of two sets, A and B, is a new set that contains all the unique elements from both sets. It is like combining the contents of two bags into one larger bag, making sure not to put in any duplicates.

For example, consider we have two sets: A and B as follows:

Set A = {1, 2, 3}
Set B = {3, 4, 5}
A ∪ B = {1, 2, 3, 4, 5}

In this example, the value 3 appears in both sets, but it is only included once in the resulting union set because sets cannot contain duplicates.

The union of sets A and B is denoted by the symbol ∪. So, the union of A and B is written as A ∪ B in the mathematical notation. The following diagram exemplifies the union operation:



The union operation of two sets, highlighting all the area of both sets

Now, let's implement the union method in our `MySet` class with the following code:

```
union(otherSet) {
  const unionSet = new MySet();
  this.values().forEach(value => unionSet.add(value));
  otherSet.values().forEach(value => unionSet.add(value));
  return unionSet;
}
```

We need three steps to perform the union of two sets:

1. Create a new empty set: this will be the set to hold the results of the union.
2. Iterate over the first set: add each element from the first set to the new set.
3. Iterate over the second set: add each element from the second set to the new set.

When performing the add operation, it will evaluate if the value is duplicate or not, resulting in a new set containing all the unique elements from the original sets.

It is important to note that the union, intersection, and difference methods we are implementing in this chapter do not modify the current instance of the `MySet` class nor the `otherSet` that is being passed as a parameter. Methods or functions that do not have collateral effects are called **pure functions**. *A pure function does not modify the current instance nor the parameters; it only produces a new result.*

Let's see this in action. Suppose an online advertising platform wants to target users based on their interests, which are collected from various sources (for example: websites visited and social media activity). To be able to launch a campaign, we need:

Collect sets of keywords representing interests from different sources.
Calculate the union of these sets to get a comprehensive list of user interests.
Use this combined set to match users with relevant advertisements.

The following would be the code that would represent this logic. Let's first collect the interest from websites:

```
const interestsFromWebsites = new MySet();
interestsFromWebsites.addAll(['technology', 'politics',
'photography']);
```

Next, let's collect the interested from social media:

```
const interestsFromSocialMedia = new MySet();
interestsFromSocialMedia.addAll(['technology', 'movies', 'books']);
```

With both sources, we can calculate the union to have a list of all interests:

```
const allInterests =
interestsFromWebsites.union(interestsFromSocialMedia);
console.log(allInterests.values());
// output: ['technology', 'politics', 'photography', 'movies',
'books']
```

Now we can try to launch a successful campaign!

To facilitate our examples, we can also create a new method what will take an array of values as the input:

```
addAll(values) {
  values.forEach(value => this.add(value));
}
```

This method will add each element individually so we can save some time during the next examples.

Intersection: identifying common values in two sets

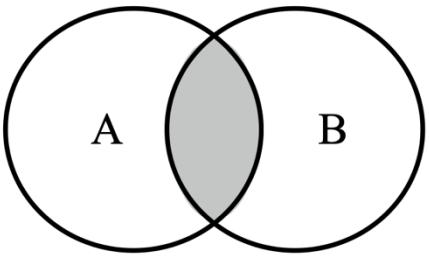
The intersection of two sets, A and B, is a new set that contains only the elements that are common to both sets. Think of it as finding the overlap between the contents of two bags.

For example, consider we have two sets: A and B as follows:

Set A = {1, 2, 3, 4}
Set B = {3, 4, 5, 6}
A ∩ B = {3, 4}

In this example, the values 3 and 4 are present in both sets, so they are included in the resulting intersection set.

The intersection of sets A and B is denoted by the symbol ∩. So, the intersection of A and B is written as A ∩ B. The following diagram exemplifies the intersection operation:



The intersection operation of two sets, highlighting only the middle, which is the shared area of both sets

Now, let's implement the intersection method in our `MySet` class with the following code:

```
intersection(otherSet) {
  const intersectionSet = new MySet();
  this.values().forEach(value => {
    if (otherSet.has(value)) {
      intersectionSet.add(value);
    }
  });
```

```
  return intersectionSet;
}
```

We need three steps to perform the intersection of two sets:

1. Create a new empty set: this will be the set to hold the results of the intersection.
2. Iterate over the first set: for each element in the first set, check if it also exists in the second set.
3. Conditional addition: If the element is found in both sets, add it to the new set.

Let's see this in action. Suppose a job platform wants to match candidates with job postings based on their skills. For this implementation we would need the following logic:

1. Represent a candidate's skills and a job's required skills as sets.
2. Find the intersection of these sets to determine the skills that the candidate possesses, and the job requires.
3. Rank job postings based on the size of the intersection to show the most relevant jobs to the candidate.

The following would be the code that would represent this logic. First, we will define the job postings available:

```
const job1Skills = new MySet();
job1Skills.addAll(['JavaScript', 'Angular', 'Java', 'SQL']);
const job2Skills = new MySet();
job2Skills.addAll(['Python', 'Machine Learning', 'SQL',
'Statistics']);
const jobPostings =
 [{
      title: 'Software Engineer',
      skills: job1Skills
  },
  {
      title: 'Data Scientist',
      skills: job2Skills
  }];
```

The `jobPostings` variable is an array of job objects, each with a `title` and a `MySet` named skills containing the required skills for that job.

Next, we will define the candidate with the name and their skills:

```
const candidateSkills = new MySet();
candidateSkills.addAll(['JavaScript', 'Angular', 'TypeScript',
'AWS']);
const candidate = {
  name: 'Loiane',
  skills: candidateSkills
};
```

The `candidate` is an object representing a job seeker with a name and a `MySet` named skills containing their skills.

Then, we can create a function that will calculate the best potential matches between the candidate and the job postings available:

```
function matchCandidateWithJobs(candidate, jobPostings) {
  const matches = [];
  for (const job of jobPostings) {
      const matchingSkillsSet =
candidate.skills.intersection(job.skills);
      if (!matchingSkillsSet.isEmpty()) {
          matches.push({
              title: job.title,
              matchingSkills: matchingSkillsSet.values()
          });
      }
  }
  return matches;
}
```

Here is an explanation of the `matchCandidateWithJobs` function:

- Takes the `candidate` and the `jobPostings` as input.
- Initializes an empty array `matches` to store the matching jobs.
- Iterates through each job in the `jobPostings` array.

For each job, it calculates the intersection of the candidate's skills and the job's required skills.
If the intersection set is not empty (meaning there are matching skills), the job title and the matching skills (as an array) are added to the `matches` array.
Finally, we return the matches array containing the job titles and their matching skills with the candidate.

Putting all together:

```
const matchingJobs = matchCandidateWithJobs(candidate, jobPostings);
console.log(matchingJobs);
// output: [{ title: 'Software Engineer', matchingSkills:
[ 'JavaScript', 'Angular' ] }]
```

We get the output that the best job posting for this candidate would be the Software Engineer job because the candidate also has JavaScript and Angular skills.

The intersection logic we created works perfectly, however, there is an improvement we can make.

Improving the intersection logic

Consider the following scenario:

Set A contains values: {1, 2, 3, 4, 5, 6, 7}
Set B contains values: {4, 6}

In our initial intersection method, we would iterate through all seven elements of Set A and check for their presence in Set B. However, a more efficient approach exists.

We can optimize the intersection method by iterating over the *smaller* of the two sets. This significantly reduces the number of iterations and comparisons needed when one set is considerably smaller than the other. The optimized code is presented as follows:

```
intersection(otherSet) {
  const intersectionSet = new MySet();
  const [smallerSet, largerSet] = this.size <= otherSet.size ? [this,
otherSet] : [otherSet, this];
  smallerSet.values().forEach(value => {
    if (largerSet.has(value)) {
      intersectionSet.add(value);
    }
  });
  return intersectionSet;
}
```

We use a concise ternary expression to determine which set has fewer elements: `this.size <= otherSet.size ? [this, otherSet] : [otherSet, this]`. This assigns the smaller set to `smallerSet` and the larger set to `largerSet`. Then, we iterate over the `values()` of the `smallerSet`. This immediately reduces the number of loop iterations to the size of the smaller set.

In cases where one set is much smaller than the other, this optimization significantly reduces the number of iterations and comparisons, leading to faster execution time. And the overall performance of the intersection operation is enhanced, especially for scenarios with large set size disparities.

Difference between two sets

The difference between two sets, A and B (denoted as A - B or A \ B), is a new set that contains all the elements of A that are not present in B. In other words, it is the set of elements that are unique to set A.

For example, consider we have two sets: A and B as follows:

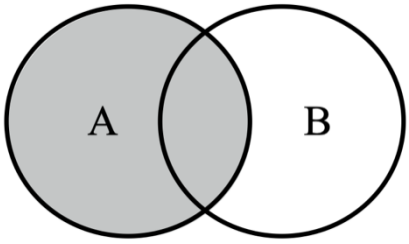
Set A = {1, 2, 3, 4}
Set B = {3, 4, 5, 6}
A - B = {1, 2}
B - A = {5, 6}

In this example, A - B results in the set {1, 2} because these elements are in A but not in B. Similarly, B - A results in {5, 6}.

The difference of sets A and B is written as:

- A - B (sometimes read as *A minus B*)
- A \ B

The following diagram exemplifies the difference operation of A - B:



The difference operation of two sets A - B, highlighting only the area of A not common to B

Now, let's implement the difference method in our `MySet` class with the following code:

```
difference(otherSet) {
  const differenceSet = new MySet();
  this.values().forEach(value => {
    if (!otherSet.has(value)) {
      differenceSet.add(value);
    }
  });
  return differenceSet;
}
```

We need three steps to perform the difference of two sets:

- Create a new empty set: this will hold the result of the difference.
- Iterate over the first set: for each element in the first set, check if it exists in the second set.
- Conditional addition: if the element is *not* found in the second set, add it to the new set.

Let's see this in action. Suppose we are running an online store with a list of subscribers who receive promotional emails. We have segmented the subscribers based on their interests (books, fashion, technology). We want to send a targeted

email campaign about books, but we want to exclude subscribers who have already shown interest in these books.

So, let's start by declaring all the sets we need for this scenario:

```
const allSubscribers = new MySet();
allSubscribers.addAll(['Aelin', 'Rowan', 'Xaden', 'Poppy', 'Violet']);
const booksInterested = new MySet();
booksInterested.addAll(['Aelin', 'Poppy', 'Violet']);
const alreadyPurchasedBooks = new MySet();
alreadyPurchasedBooks.addAll(['Poppy']);
```

We have three sets:

- `allSubscribers`: a set of all email subscribers.
- `booksInterested`: a set of subscribers who have expressed interest in the books.
- `alreadyPurchasedBooks`: a set of subscribers who have already purchased books.

Next, we will find the subscribers interested in books, but have not purchased yet:

```
const targetSubscribers =
booksInterested.difference(alreadyPurchasedBooks);
```

We use `booksInterested.difference(alreadyPurchasedBooks)` to find the subscribers who are interested in books but have not yet made a purchase in that category. This gives us the `targetSubscribers` set.

And finally, we will send the email to the target subscribers:

```
targetSubscribers.values().forEach(subscriber => {
  sendEmail(subscriber, 'New books you will love!');
});
function sendEmail(subscriber, message) {
  console.log(`Sending email to ${subscriber}: ${message}`);
}
```

And the output we will get is:

```
Sending email to Aelin: New books you will love!
Sending email to Violet: New books you will love!
```
We only have one last operation to cover: subsets

Subset: checking if a set contains all the values

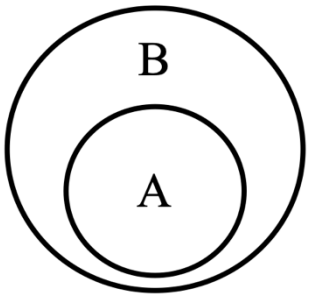
A set A is a subset of another set B if every element of A is also an element of B. In simpler terms, A is completely contained within B.

For example, consider we have two sets: A and B as follows:

Set A = {1, 2}
Set B = {1, 2, 3, 4}
A ⊆ B

In this example, A is a subset of B because every element in A is also in B.

The subset relationship is denoted by the symbol ⊆. So, if A is a subset of B, we write: A ⊆ B. The following diagram exemplifies the subset relationship:



A is a subset of B because element in A is also in B

Now, let's implement the `isSubsetOf` method in our `MySet` class with the following code:

```
isSubsetOf(otherSet) {
  if (this.size > otherSet.size) {
    return false;
  }
  return this.values().every(value => otherSet.has(value));
}
```

We start by checking if the size of the current set (`this.size`) is greater than the size of the `otherSet`. If it is, we know immediately that the current set cannot be a subset of `otherSet` because a subset cannot have more elements than the set it is a part of. In this case, the method returns `false` early, saving unnecessary further checks.

If the size check passes, we call `this.values()` to get an array of all the values in the current set. Then, we use the `every()` method on this array to check if the other set has the value from the current set. If every value in the current set is also found in `otherSet`, then the `every()` method returns `true` (meaning the current set is a subset). If even a single value in the current set is not found in `otherSet`, `every()` returns `false` (meaning it's not a subset).

Let's see this in action. Imagine we are developing a recipe app with a large database of recipes. Each recipe has a set of ingredients. Users can filter the recipes based on the ingredients they have available.

We will start by declaring the sets that store the ingredients of our recipes:

```
const chickenIngredients = new MySet()
chickenIngredients.addAll(['chicken', 'tomato', 'onion', 'garlic',
'ginger', 'spices']);
const spaghettiIngredients = new MySet();
spaghettiIngredients.addAll(['spaghetti', 'eggs', 'bacon', 'parmesan',
'pepper']);
```

Next, we will declare the recipes along with the ingredients:

```
const recipes =
[{
      name: 'Chicken Tikka Masala',
      ingredients: chickenIngredients
  },
  {
      name: 'Spaghetti Carbonara',
      ingredients: spaghettiIngredients
  }];
```

The `recipes` variable is an array of recipe objects, each with a name and a `MySet` named ingredients representing the ingredients required for that recipe.

Then, we also need a set with the list of ingredients we have available:

```
const userIngredients = new MySet();
userIngredients.addAll(['chicken', 'onion', 'garlic', 'ginger']);
```

The next step would be the logic to check if we have a recipe that matches our ingredients:

```
function filterRecipes(recipes, userIngredients) {
  const filteredRecipes = [];
  for (const recipe of recipes) {
      if (userIngredients.isSubsetOf(recipe.ingredients)) {
          filteredRecipes.push({ name: recipe.name });
      }
  }
  return filteredRecipes;
}
```

Here is an explanation of the `filterRecipes` function:

- Takes the `recipes` and `userIngredients` as input.
- Initializes an empty array `filteredRecipes` to store the matching recipe names.
- Iterates over each recipe in the recipes array.
- For each recipe, it checks if `recipe.ingredients.isSubsetOf(userIngredients)`. If `true` (meaning all the recipe's ingredients are present in the user's ingredients), the recipe's name is added to `filteredRecipes`.
- Returns the `filteredRecipes` array.

And finally, putting all together:

```
const matchingRecipes = filterRecipes(recipes, userIngredients);
console.log(matchingRecipes);
```

We will get the following output:

```
[ { name: 'Chicken Tikka Masala' } ]
```
We can also implement the isSupersetOf method, which would check if the current set A is a superset of another set B if every element of B is also an element of A. In simpler terms, B is completely contained within A. Try it, and you can find the source code within the `MySet` class when you download the source code of this book.

Now that we have added some additional logic to the `MySet` class, let's check how the native JavaScript Set class works.

The JavaScript Set class

Let's dive into the native Set class introduced in ECMAScript 2015 (ES6) and explore how to use it effectively.

The Set class provides a built-in, efficient way to work with sets in JavaScript. It offers all the fundamental set operations and is optimized for performance.

Now, let's look at the methods and features available in the native Set class:

Two constructors:
`new Set()`: creates an empty Set.
`new Set(iterable)`: creates a Set from an iterable object (for example, an array).
`add(value)`: adds a value to the set (if it is not already present). Returns the Set object itself for chaining.
`delete(value)`: removes the specified value from the set. Returns `true` if the value was present and removed, otherwise `false`.
`clear()`: removes all elements from the set.
`has(value)`: returns `true` if the value exists in the set, otherwise `false`.
Different methods for iterating the set:
`forEach(callbackFn)`: executes the provided `callbackFn` for each value in the set.
`values()`: returns an iterator over the values of the set.
`keys()`: alias for `values()`.

`entries()`: Returns an iterator over `[value, value]` pairs (since keys and values are the same in a Set).

`size`: property that returns the number of elements in the set.

If we would like to rewrite our example of the article and its tags, can we simply replace `MySet` with `Set` and the code would still work as follows:

```
const article = {
  title: 'The importance of data structures in programming',
  content: '...',
  tags: new Set()
};
article.tags.add('programming');
article.tags.add('data structures');
article.tags.add('algorithms');
article.tags.add('programming');
```

Given the Set class also has a constructor that accepts an array, we could simplify the previous code and pass the tags directly to the constructor:

```
const article = {
  title: 'The importance of data structures in programming',
  content: '...',
  tags: new Set(['programming', 'data structures', 'algorithms'])
};
```

The other methods, such as delete, check the size, has and values would also work as expected.

Building our custom `MySet` class served as a valuable learning exercise, providing insights into the internal workings and mechanics of set data structures. While in everyday JavaScript development, we would likely use the efficient and convenient built-in Set class, the knowledge gained from implementing our own set empowers us to understand the underlying principles, make informed choices between built-in and custom solutions, and troubleshoot set-related issues more effectively.

Reviewing the efficiency of sets

Let's review the efficiency of each method by reviewing the Big O notation in terms of time of execution:

Method	MySet	Set	Explanation
`add(value)`	*O(1)*	*O(1)*	Constant time insertion into the object or underlying data structure.
`addAll(values)`	*O(n)*	*O(n)*	Calls `add(value)` for each value in the input array, where *n* is the size of the array.
`delete(value)`	*O(1)*	*O(1)*	Constant time deletion from the object or underlying data structure.
`has(value)`	*O(1)*	*O(1)*	Object lookup in both cases has constant time
`values()`	*O(n)*	*O(n)*	In `MySet` , it iterates over the object's keys. In `Set` , it creates an iterator that yields each value in linear time.
`size (getter)`	*O(1)*	*O(1)*	Returns the value of the `#size` property or equivalent in the native Set.
`isEmpty()`	*O(1)*	*O(1)*	Checks if `#size` is 0.
`values()`	*O(n)*	*O(n)*	In `MySet` , it iterates over the object's keys. In `Set` , it creates an iterator that yields each value in linear time.
`clear()`	*O(1)*	*O(1)*	Reassigns the `#items` object to an empty object and resets `#size` .

The overall space complexity of sets is considered O(n), where n is the number of unique elements stored in the set. This means that the memory used by a set data structure increases linearly with the number of elements it contains.

Reviewing the time complexity, adding, removing values, and checking if a value exists in a set have constant time. One might ask why not always use sets instead of arrays or lists? While sets excel at specific tasks, there are a few reasons why we would not always choose them over arrays or lists:

If the order of the elements is crucial, arrays are the way to go. Sets do not guarantee any specific order of elements.
If we frequently need to access elements by their position (for example, getting the third item in a list), arrays are much faster due to their direct indexing. Sets require iteration to find a specific element.
If the data naturally contains duplicates, and those duplicates are meaningful, then an array is the more appropriate choice.

Let's put our knowledge into practice with some exercises.

Exercises

We will resolve one exercise from **LeetCode** using the set data structure to remove duplicate values from an array.

Remove duplicates from sorted array

The exercise we will resolve the is the *26. Remove Duplicates from Sorted Array* problem available at **https://leetcode.com/problems/remove-duplicates-from-sorted-array/description/**.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function `function removeDuplicates(nums: number[]): number`, which receives an array of numbers and expects the number of unique elements within the array as a return. For our solution to be accepted, we also have to remove the duplicates from the `nums` array in-place, meaning we cannot simply assign a new reference to it.

Let's write the `removeDuplicates` function using a set data structure to easily remove the duplicates from the array:

```
export function removeDuplicates2(nums: number[]): number {
  const set = new Set(nums);
  const arr = Array.from(set);
  for (let i = 0; i < arr.length; i++) {
    nums[i] = arr[i];
```

```
  }
  return arr.length;
}
```
Here is an explanation of the solution:

We start by creating a new JavaScript Set object, initializing it with the values from the input array `nums`. Sets automatically store only unique values, so any duplicates in `nums` are eliminated.

Next, we convert the set back into a regular array `arr`. This new array contains only the unique elements from the original `nums` array, in sorted order. This step is required because we cannot access each set value directly, like an `array[i]`.

The for loop iterates through the `arr` (unique elements) array and copies each element back into the original `nums` array, overwriting any duplicate values that were present. Since `arr` is guaranteed to be shorter or equal in length to `nums`, we only need to iterate up to the length of arr. This step is a requirement as the problem judge will also check if the `nums` array was modified in-place.

Finally, the method returns arr.length, which is the number of unique elements in the original array. This is the expected output for the LeetCode problem.

The time complexity of this function is *O(n)*, where *n* is the number of values we have in the array `nums`. We are creating the set, adding all the elements (*O(n)*), we convert the set into an array (*O(n)*), and we also have a loop to overwrite the original array (*O(k)*, where *k* is the number of unique elements). Therefore, the overall time complexity is *O(n)*, as it is dominated by the linear-time operations of creating the set and converting it to an array.

The space complexity is *O(k)* because we are creating a set to store the unique elements. In the worst-case scenario where all elements are unique, it will store all *n* elements. However, in most cases, *k* (the number of unique elements) will be smaller than *n*. We also have the array `arr`, which stores only the unique

elements, so its size is *k*. Therefore, the overall space complexity is *O(k)*, where *k* is the number of unique elements in the input array.

While the algorithm is correct and solves the problem, it is not the most *space-efficient* solution. We will resolve this same problem later in this book using a different technique. In the meantime, give it a try and try to solve this problem using *O(1)* space complexity.

Summary

In this chapter, we delved into the inner workings of set data structures by implementing a custom `MySet` class. This hands-on approach mirrors the core functionality of the Set class introduced in ECMAScript 2015, giving you a deeper understanding of how sets operate under the hood. We also extended our exploration beyond the standard JavaScript Set by implementing additional methods like union, intersection, difference, and subset, enriching your toolkit for working with sets.

To put our newfound knowledge into practice, we tackled a real-world LeetCode problem, demonstrating the power of sets in solving algorithmic challenges.

In the next chapter, we will shift our focus to non-sequential data structures, specifically hashes and dictionaries. Get ready to discover how these versatile structures enable efficient data storage and retrieval based on key-value pairs!

8 Dictionaries and Hashes

Before you begin: Join our book community on Discord

Give your feedback straight to the author himself and chat to other early readers on our Discord server (find the "learning-javascript-dsa-4e" channel under EARLY ACCESS SUBSCRIPTION).

https://packt.link/EarlyAccess/

In the previous chapter, we delved into the world of sets, focusing on their ability to efficiently store unique values. Building upon this foundation, we will now explore two more data structures designed for storing distinct elements: dictionaries and hashes.

While sets prioritize the value itself as the primary element, dictionaries and hashes take a different approach. Both structures store data as key-value pairs, allowing us to associate a unique key with a corresponding value. This pairing is fundamental to how dictionaries and hashes work.

However, there is a subtle yet important distinction in implementation. Dictionaries, as we will soon discover, adhere to a strict rule of one value per key. Hashes, on the other hand, offer some flexibility in handling multiple values associated with the same key, opening up additional possibilities for data organization and retrieval.

In this chapter, we will cover:

- The dictionary data structure
- The hash table data structure
- Handling collisions in hash tables
- The JavaScript native `Map`, `WeakMap`, and `WeakSet` classes

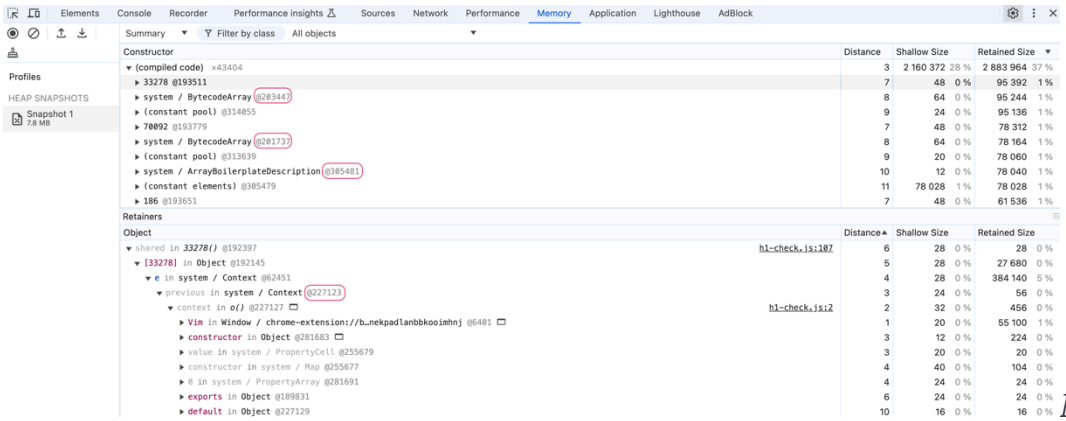
The dictionary data structure

As we have explored, a set is a collection of unique elements, ensuring that no duplicates exist within the structure. In contrast, a **dictionary** is designed to store pairs of keys and values. This pairing enables us to utilize keys as identifiers to efficiently locate specific elements.

While dictionaries share similarities with sets, there is a crucial distinction in the type of data they store. Sets maintain a collection of key-key pairs, where both elements of the pair are identical. Dictionaries, on the other hand, house key-value pairs, associating each unique key with a corresponding value.

It is worth noting that dictionaries are known by various names in different contexts, including **maps**, **symbol tables**, and **associative arrays**. These terms highlight the fundamental purpose of dictionaries: to establish associations between keys and values, facilitating efficient data retrieval and organization.

In computer science, dictionaries are frequently employed to store the reference addresses of objects. These addresses serve as unique identifiers for objects residing in memory. To visualize this concept, consider opening the *Chrome Developer Tools* and navigating to the **Memory** tab. Running a snapshot will reveal a list of objects along with their respective address references, typically displayed in the format @<*number*>. The following screenshot illustrates how dictionaries can be used to associate keys with these memory addresses, enabling efficient object retrieval and manipulation.



Memory tab of a browser displaying the memory allocation for address references

In this chapter, we will also cover some examples of how to use the dictionary data structure in real world projects.

Creating the Dictionary class

In addition to the `Set` class, **ECMAScript 2015** (ES6) introduced the `Map` class, a fundamental data structure often referred to as a dictionary in programming. This native implementation serves as the basis for the custom dictionary class we will develop in this chapter.

The `Dictionary` class we will construct draws heavily from the design principles of the JavaScript Map implementation. As we explore its structure and functionality, you will observe striking similarities to the `Set` class. However, a key distinction lies in the data storage mechanism. Instead of storing only values, as in a Set, our dictionary class will accommodate key-value pairs. This modification allows us to associate unique keys with their corresponding values, thereby unlocking the full power and versatility of dictionaries as a data structure.

Our implementation will reside in the src/08-dictionary-hash/dictionary.js file. We will start by defining the `Dictionary` class:

```
class Dictionary {
  #items = {};
```

```
  #size = 0;
}
```
We utilize an object (`{}`) to store elements within the `#items` private property. The keys of this object represent the unique keys, while the corresponding values can be anything. We will also keep track of the number of elements in the set with the property `size`.

In an ideal scenario, a dictionary would seamlessly store keys of the string type alongside values of any type, whether they are primitive values like numbers or strings, or more complex objects. However, JavaScript's dynamically typed nature introduces a potential challenge. Since we cannot guarantee that keys will consistently be strings, we must implement a mechanism to transform any object passed as a key into a string format. This transformation simplifies the process of searching for and retrieving values within our Dictionary class, enhancing its overall functionality. The same logic can also be applied to the Set class we explored in the previous chapter.

Note that we do not have this issue in the TypeScript implementation, as we can define the type of the key as string.
To achieve this key transformation, we require a function that can reliably convert objects into strings. As a default option, we will leverage the `#elementToString` method we have defined earlier this book in previous data structures. This function provides a reusable solution for stringifying keys, making it adaptable to any data structure we create:

```
#elementToString(data) {
  if (typeof data === 'object' && data !== null) {
    return JSON.stringify(data);
  } else {
    return data.toString();
  }
}
```
This method efficiently converts data into a string representation. If the data is a complex object (excluding `null`), it utilizes `JSON.stringify()` to produce a **JSON**

string. Otherwise, it leverages the `toString` method to ensure a string conversion for any other data type.

Now, let's define the methods that will empower our dictionary/map data structure:

- `set(key, value)`: inserts a new key-value pair into the dictionary. If the specified key already exists, its associated value will be updated with the new value.
- `remove(key)`: removes the entry corresponding to the provided key from the dictionary.
- `hasKey(key)`: determines whether a given key is present in the dictionary, returning `true` if it exists and `false` otherwise.
- `get(key)`: retrieves the value associated with the specified key.
- `clear()`: empties the dictionary, removing all key-value pairs.
- `size()`: returns the count of key-value pairs currently stored in the dictionary, similar to the length property of an array.
- `isEmpty()`: checks if the dictionary is empty, returning `true` if the size is zero and `false` otherwise.
- `keys()`: generates an array containing all the keys present in the dictionary.
- `values()`: produces an array containing all the values stored in the dictionary.
- `forEach(callbackFn)`: iterates over each key-value pair in the dictionary. The `callbackFn` function, which accepts a key and a value as parameters, is executed for each entry. This iteration process can be terminated if the callback function returns `false`, mirroring the behavior of the `every` method in the `Array` class.

We will implement each of these methods in detail in the following sections.

Verifying whether a key exists in the dictionary

The first method we will implement is the `hasKey(key)` method. This method is fundamental, as it will be utilized in other methods like `set` and `remove`. Let's examine its implementation:

```
hasKey(key) {
  return this.#items[this.#elementToString(key)] != null;
}
```
In JavaScript, object keys are inherently strings. Therefore, if a complex object is provided as a key, we must convert it to a string representation. To achieve this, the `#elementToString` method is invoked consistently, ensuring that keys are always treated as strings within our dictionary.

The `hasKey` method checks if there is a value associated with the given key within the items table (the underlying storage for our dictionary). If the corresponding position in the table is not `null` or `undefined`, indicating the presence of a value, the method returns `true`. Otherwise, if no value is found, the method returns `false`.

And now that we have this method, we can proceed with the implementation of the methods for adding and removing values.

Setting a key and value in the dictionary

Next, we will implement the `set` method in our `Dictionary` class. The `set` method serves a dual purpose: it can both add a new key-value pair to the dictionary and update the value of an existing key:

```
set(key, value) {
  if (key != null && value != null) {
    const tableKey = this.#elementToString(key);
    this.#items[tableKey] = value;
    this.#size++;
    return true;
  }
  return false;
}
```
This method accepts a `key` and a `value` as input. If both the key and value are valid (not `null` or `undefined`), the method proceeds to convert the key into a string representation. This is a crucial step because JavaScript object keys can only be strings. This conversion is handled internally by the private

`#elementToString` method, ensuring consistency and reliability across all key types. With the key in string form, the method then stores the value within the dictionary's internal storage (`#items`).

Finally, the method communicates its success by returning `true`, signaling that the key-value pair was successfully inserted or updated and we increment its size. If either the key or value is invalid (`null` or `undefined`), the method returns `false`, signaling that the insertion or update operation failed.

Removing and clearing all values from the dictionary

The delete method's primary function is to remove a key-value pair from the dictionary based on the provided key. It ensures the integrity of the dictionary by checking for the key's existence before attempting removal and updating the size accordingly:

```
delete(value) {
  if (this.has(value)) {
    delete this.#items[value];
    this.#size--;
    return true;
  }
  return false;
}
```

We start by verifying if the provided key exists within the dictionary. This is achieved by calling the `has` method, which checks the dictionary's underlying storage for the presence of the specified key. This check is crucial to prevent errors that might arise from trying to delete a non-existent entry.

If the key is found, the `delete` operator in JavaScript is employed to remove the corresponding key-value pair from the dictionary's internal data structure (`#items`). Following the successful removal of the entry, the dictionary's internal size counter (`#size`) is decremented by one to accurately reflect the change in the number of stored elements.

As a last step, the method signals the outcome of the operation by returning `true` to indicate that the key existed and was successfully deleted, and `false` to indicate that the key was not found and no deletion occurred.

And if we want to remove all the elements from the set, we can use the `clear` method, as follows:

```
clear() {
  this.#items = {};
  this.#size = 0;
}
```

This effectively discards all previous key-value pairs and creates a fresh, empty container for future additions. And we also reset the `#size` property back to 0 to accurately reflect that the set now contains no elements.

Retrieving the size and checking if it is empty

The next method we will implement is the size method as follows:

```
 get size() {
    return this.#size;
  }
```

This method simply returns the size property we are using to keep count.

And to determine if the dictionary is empty, we implement the `isEmpty()` method, following a pattern consistent with other data structures we have covered in this book:

```
isEmpty() {
  return this.#size === 0;
}
```

This method directly compares the private `#size` property to 0. The property `#size` is meticulously maintained to always reflect the number of elements in the set.

Retrieving a value from the dictionary

To search for a specific key within our dictionary and retrieve its associated value, we utilize the `get` method. This method streamlines the process of accessing stored data by encapsulating the necessary logic and is presented as follows:

```
 get(key) {
  return this.#items[this.#elementToString(key)];
}
```

Upon receiving a `key` as input, the `get` method first transforms it into a string representation using the private `#elementToString` function. Next, the method directly accesses the corresponding value from the dictionary's internal storage (`#items`). This is achieved by using the stringified key to index into the #items object, which presumably holds the key-value pairs. The value associated with the given key, if found, is then returned by the method.

Retrieving all the values and all the keys from the dictionary

Let's explore how to retrieve all values and keys from our custom dictionary class in JavaScript. We will start by declaring the method `values`, which will retrieve all the values stored in the Dictionary class as follows:

```
values() {
  return Object.values(this.#items);
}
```

This method is quite straightforward. It leverages the built-in `Object.values()` function, which takes an object (in this case, our private `#items` storage) and returns an array containing all of its values.

Next, we have the `keys` method:

```
keys() {
  return Object.keys(this.#items);
}
```

Similarly, the `keys` method uses the `Object.keys()` function. This function, when given an object, returns an array of all the string-based keys (property names) in that object. Since we ensure that all keys are strings in our dictionary implementation, this works perfectly.

In most cases, these methods have good performance. However, for exceptionally large dictionaries, iterating directly over the `#items` object might be slightly more efficient in some JavaScript engines. Let's see how we can achieve this in the next topic.

Iterating each value-pair of the dictionary with forEach

Thus far, we have not implemented a method that facilitates iteration through each value stored within our data structures. We will now introduce the `forEach` method for the `Dictionary` class, with the added benefit that this behavior can also be applied to other data structures we have previously constructed.

Here is the `forEach` method:

```
forEach(callbackFn) {
  for (const key in this.#items) {
    if (this.#items.hasOwnProperty(key)) {
      callbackFn(this.#items[key], key);
    }
  }
}
```

The `forEach` method is designed to iterate over every key-value pair within our dictionary, applying a provided callback function to each entry. For each key-value pair, the provided `callbackFn` function is executed, receiving the value and key as arguments.

We use a `for...in` loop for iterating over object properties. However, to ensure that we only process the dictionary's own properties (and not inherited ones from its prototype chain), a safeguard is employed. The `hasOwnProperty` method checks whether a property belongs directly to the object. In this case, it verifies if the

current `key` in the loop is an actual key within the `#items` object, the dictionary's underlying storage. Then, we apply the provided callback function to each entry, retrieving the value from the dictionary and passing the key as an argument to the callback.

Now that we have our data structure, let's test it!

Using the Dictionary class

Imagine we are building a simple language learning program. We want to store translations for frequently-used words and phrases to help users quickly look up meanings in different languages.

The source code for this example can be found in the file `src/08-dictionary-hash/01-using-dictionary-class.js`. Let's start by creating the dictionary and adding some values:

```
const translations = new Dictionary();
// Add some translations - English to Portuguese
translations.set("hello", "olá");
translations.set("thank you", "obrigado");
translations.set("book", "livro");
translations.set("cat", "gato");
translations.set("computer", "computador");
```

We use `set` to populate the dictionary with key-value pairs representing word translations. The keys are words in English, and the values are their Portuguese translations.

Next, we will create a function so the user can interact with it to retrieve the translation of a particular word or phrase:

```
function translateWord(word) {
  if (translations.hasKey(word)) {
    const translation = translations.get(word);
    console.log(`The translation of "${word}" is "${translation}"`);
  } else {
```

```
    console.log(`Sorry, no translation found for "${word}"`);
  }
}
```

The `translateWord` function takes a `word` as input. It uses `hasKey` to check if the word exists in the dictionary. If the word is found, it retrieves the translation using the `get` method and prints it. If not found, it displays a "no translation found" message.

We can try this function with the following code:

```
translateWord("hello"); // Output: The translation of "hello" is "olá"
translateWord("dog");  // Output: Sorry, no translation found for
"dog"
```

We can also check all translations available:

```
console.log("All translations:", translations.values());
// All translations: [ 'olá', 'obrigado', 'livro', 'gato',
'computador' ]
```

And alls words we have transactions available:

```
console.log("All words:", translations.keys());
// All words: [ 'hello', 'thank you', 'book', 'cat', 'computer' ]
```

And in case we would like to print the dictionary, we can use the `forEach` method as follows:

```
translations.forEach((value, key) => {
  console.log(`${key}: ${value}`);
});
```

We will get the following output:

```
hello: olá
thank you: obrigado
book: livro
cat: gato
computer: computador
```

So, now that we have a very similar implementation of the native JavaScript Map class,

The JavaScript Map class

ECMAScript 2015 introduced a Map class as part of the JavaScript API. We developed our Dictionary class based on the ES2015 Map class.

At its core, a Map is a collection of key-value pairs, similar to a dictionary or hash table in other programming languages. However, unlike plain JavaScript objects, a Map offers several key advantages as follows:

- The Map class allows keys of any data type, including objects, functions, or even other Map objects. In contrast, object keys are automatically converted to strings.
- The Map class maintains the order in which key-value pairs were inserted, making iteration predictable.
- We can easily get the number of entries using the size property, whereas with objects, we typically need to use `Object.keys(obj).length`.
- The Map class natively supports iteration using `for...of` loops, making it more convenient to work with.

Now, let's take a look at the methods and features available in the native Map class:

- `set(key, value)`: adds or updates a key-value pair.
- `get(key)`: retrieves the value associated with the key.
- `has(key)`: checks if a key exists.
- `delete(key)`: removes a key-value pair.
- `size`: returns the number of entries.
- `clear()`: removes all entries.
- `forEach(callbackFn)`: iterates over all entries.

If we would like to rewrite our translation application example, can we simply replace `Dictionary` with `Map` and the code would still work as follows:

```
const translations = new Map();
translations.set("hello", "olá");
translations.set("thank you", "obrigado");
translations.set("book", "livro");
```

```
translations.set("cat", "gato");
translations.set("computer", "computador");
```
The other methods, such as `get`, check the `size`, `has`, `values` and `forEach` would also work as expected.

Constructing our custom `Dictionary` class has proven to be an enlightening educational endeavor, granting us a deeper understanding of the inner mechanisms of map data structures. While the built-in JavaScript Map class offers efficiency and convenience for most everyday scenarios, the experience of creating our own dictionary equips us with valuable knowledge.

JavaScript also supports a weak version of the Map and Set classes: `WeakMap` and `WeakSet`. Let's briefly take a look at them.

The JavaScript WeakMap and WeakSet classes

In addition to the standard `Map` and `Set` classes, JavaScript offers two specialized collection types known as `WeakMap` and `WeakSet`. These classes provide a unique way to manage object references and can be particularly useful in scenarios where memory management is a concern.

Similar to a `Map`, a `WeakMap` stores key-value pairs. However, the keys in a `WeakMap` *must* be *objects*, and the references to these keys are weak. This means that if the only reference to an object is its presence as a key in a `WeakMap`, the JavaScript garbage collector can remove that object from memory.

A `WeakSet` functions like a `Set`, storing a collection of unique values. However, it can only store `objects`, and the references to these objects are weak. Similar to `WeakMap`, if an object's only reference is its presence in a `WeakSet`, it can be garbage collected.

`WeakMap` and `WeakSet` also have fewer methods than their regular counterparts. They lack `size`, `clear`, and iteration methods (like `forEach` and `keys()`).

Let's review a real-world scenario where we would use these classes. Imagine we are designing a program that provides a Person class. We want to store some sensitive private data associated with each person instance, like their social security number (or tax id) or medical records. However, we do not want to clutter the object itself with these properties, and we want to ensure they can be garbage collected when the Person object is no longer needed. Here is the code to exemplify this scenario:

```
const privateData = new WeakMap();
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
    privateData.set(this, { ssn: 'XXX-XX-XXXX', medicalHistory: [] });
  }
  getSSN() {
    return privateData.get(this)?.ssn;
  }
}
```

A `WeakMap` is created to store the private data. The key is the `Person` object itself (`this`). Inside the `Person` `constructor`, we use `privateData.set(this, { ... })` to associate private data with the newly created person object (`this`). The `getSSN` method retrieves the private SSN data using `privateData.get(this)`. Note the **optional chaining** (`?.`) to safely handle cases where the `Person` object might no longer exist (and this way we do not get a *null pointer* error).

Why use a `WeakMap` instead of a `Map` here? When a Person object becomes inaccessible (no references to it remain), the garbage collector can remove the reference of the object and the associated private data in the `WeakMap`, preventing memory leaks. This can be considered a good practice for managing sensitive or temporary data that does not need to outlive the objects it is associated with.

This pattern also could be used to implement private properties in JavaScript classes before the hash (#) notation was introduced to JavaScript private properties.

Now that we understand the map or dictionary data structure, let's take it to the next level with hash tables.

The hash table data structure

The **hash table** data structure, also known as **hash map**, is a hash implementation of the dictionary or map data structures. A hash table is also a collection of key-value pairs. The key is a unique identifier, and the value is the data you want to associate with that key. Hash tables achieve their speed by using a **hash function**.

This is how hash tables work:

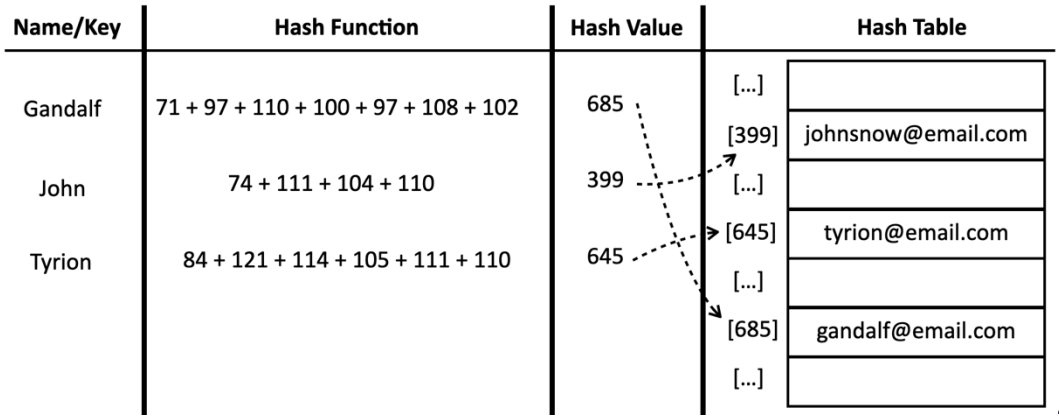
Hash Function: a hash function takes a key as input and produces a unique numerical value called a **hash code** (or **hash value**). This hash code is like a fingerprint of the key.
Storage (buckets/slots): the hash table internally consists of an array (or similar structure like a linked list) with fixed-size buckets or slots. Each bucket can store one or more key-value pairs.
Insertion: when you insert a key-value pair:
The hash function is applied to the key to get its hash code.
The hash code is used to determine the index (bucket) where the key-value pair should be stored.
The pair is placed in that bucket.
Retrieval: when you want to retrieve a value, you provide and key and:
The hash function is applied to the key again, producing the same hash code.
The hash code is used to directly access the bucket where the value should be stored.
The value is found (hopefully) in that bucket.

Hash tables are present in many different places. For example:

Databases: used for indexing data for fast retrieval.
Caches: store recently accessed data for quick lookups.

Symbol Tables: in compilers, used to store information about variables and functions.

One of the most classical examples for a hash table is an email address book. For example, whenever we want to send an email, we look up the person's name and retrieve their email address. The following image exemplifies this process:



A hash table used to store email addresses based on the contact name

For this example, we will use a hash function which will simply sum up the ASCII values of each character of the key length. This is called a **lose-lose hash** function, which is very simple function that can lead into different issues that we will explore in the next sections.

Let's translate this diagram into a source code by creating a `HashTable` class in the new topic so we can dive into this concept.

Creating the HashTable class

Our hash table implementation will be located in the `src/08-dictionary-hash/hash-table.js` file. We begin by defining the `HashTable` class:

```
class HashTable {
  #table = [];
}
```

This initial step simply initializes the private `#table` array, which will serve as the underlying storage for our key-value pairs.

Next, we will equip our `HashTable` class with three essential methods:

- `put(key, value)`: this method either adds a new key-value pair to the hash table or updates the value associated with an existing key.
- `remove(key)`: this method removes the value and its corresponding key from the hash table based on the provided key.
- `get(key)`: this method retrieves the value associated with a specific key from the hash table.

To enable the functionality of these methods, we also need to create a crucial component: the hash function. This function will play a vital role in determining the storage location of each key-value pair within the hash table, making it a cornerstone of our implementation.

Creating the lose-lose hash function

Before implementing the core `put`, `remove`, and `get` methods, we must first establish a `hash` method. This method is fundamental, as it will determine the storage location of key-value pairs within the hash table. The code is presented as follows:

```
hash(key) {
  return this.#loseLoseHashCode(key);
}
```

The `hash` method acts as a wrapper around the `loseLoseHashCode` method, forwarding the provided `key` as its parameter. This wrapper design serves a strategic purpose: it allows for future flexibility in modifying the hash function without impacting other areas of our code that utilize the hash code. The `loseLoseHashCode` method is where the actual hash calculation takes place:

```
#loseLoseHashCode(key) {
  if (typeof key !== 'string') {
    key = this.#elementToString(key);
  }
  const calcASCIIValue = (acc, char) => acc + char.charCodeAt(0);
  const hash = key.split('').reduce((acc, char) => calcASCIIValue, 0);
```

```
  return hash % 37; // mod to reduce the hash code
}
```

Within `loseLoseHashCode`, we begin by checking if the key is already a string. If not, we convert it into a string using the `#elementToString` method we created in previous chapters to ensure consistent handling of keys.

Next, we calculate a hash value by summing the ASCII values of each character in the key string. It leverages two powerful array methods, `split` and `reduce`, to achieve this efficiently. It first splits the string into an array of individual characters. Then, it uses the reduce method to iterate over these characters, accumulating their ASCII values into a single hash value. For each character, we retrieve its ASCII value using the `charCodeAt` method and add it to the hash variable.

Finally, to avoid working with potentially large numbers that might not fit within a numeric variable, we apply a *modulo* operation (the remainder after dividing one number by another) to the hash value using an arbitrary divisor (in this case, 37). This ensures that the resulting hash code falls within a manageable range, optimizing storage and retrieval within the hash table.

Now that we have our hash function, we can start diving into the next methods.

Putting a key and a value in the hash table

Having established our `hash` function, we can now proceed to implement the `put` method. This method mirrors the functionality of the `set` method in the `Dictionary` class, with a slight difference in naming convention to align with customary practice in other programming languages. The `put` method is presented as follows:

```
put(key, value) {
  if (key == null && value == null)  {
    return false;
  }
  const index = this.hash(key);
```

```
  this.#table[index] = value;
  return true;
}
```
The `put` method facilitates the insertion or updating of key-value pairs within the hash table. It first validates the provided key and value, ensuring that neither is `null` or `undefined`. This check prevents the storage of incomplete or meaningless data within the hash table.

If both the `key` and value are deemed valid, we proceed to calculate the hash code for the given key. This hash code, determined by the `hash` function, will serve as the index for storing the value in the underlying `#table` array.

Finally, the put method returns `true` to indicate that the key-value pair was successfully inserted or updated. Conversely, if either the key or value is invalid, the method returns `false`, signifying that the operation was not successful.

Once a value is present in the table, we can try to retrieve it.

Retrieving a value from the hash table

Retrieving a value from the `HashTable` instance is a straightforward process, facilitated by the `get` method. This method enables us to efficiently access data stored within the hash table based on its associated key:

```
get(key) {
  if (key == null) {
    return undefined;
  }
  const index = this.hash(key);
  return this.#table[index];
}
```
We start by validating the input key, ensuring it is not `null` or `undefined`. If the key is indeed valid, we proceed to determine its position within the hash table using the previously defined `hash` function. This function transforms the key into

a numerical hash code, which directly corresponds to the index of the value in the underlying array.

Leveraging this calculated index, the method accesses the corresponding element in the table array and returns its value. This provides a seamless way to retrieve data from the hash table, as the `hash` function eliminates the need for linear search and directly points to the desired value's location.

It is worth noting that in our `HashTable` implementation, we have included input validation to ensure the provided keys and values are not invalid (`null` or `undefined`). This is a recommended practice that can be applied to all data structures we have developed thus far in this book. By proactively validating inputs, we enhance the robustness and reliability of our data structures, preventing errors and unexpected behavior caused by incorrect or incomplete data.
Finally, let's turn our attention to the remaining method in our class: the `remove` method.

Removing a value from the hash table

The final method we will implement for our `HashTable` is the `remove` method, designed to eliminate a key-value pair based on the provided key. This method is essential for maintaining a dynamic and adaptable hash table structure:

```
remove(key) {
  if (key == null) {
    return false;
  }
  const index = this.hash(key);
  if (this.#table[index]) {
    delete this.#table[index];
    return true;
  }
  return false;
}
```

To successfully remove a value, we first need to identify its location within the hash table. This is achieved by obtaining the hash code corresponding to the given key using the `hash` function.

Next, we retrieve the value pair stored at the calculated hash position. If this value pair is not `null` or `undefined`, indicating that the key exists within the hash table, we proceed to remove it. This is accomplished by utilizing the JavaScript `delete` operator, which effectively eliminates the key-value pair from the hash table's internal storage.

To provide feedback on the operation's success, we return `true` if the removal was successful (meaning the key existed and was deleted) and `false` if the key was not found in the hash table.

It is worth noting that, as an alternative to using the delete operator, we could also assign `null` or `undefined` to the corresponding hash position to indicate its vacancy. This approach would still effectively remove the key-value association from the hash table while potentially offering a different strategy for managing empty slots within the array.

Now that the implementation of our class is complete, let's see it in action.

Using the HashTable class

Let's illustrate how our `HashTable` class can be employed to create an email address book:

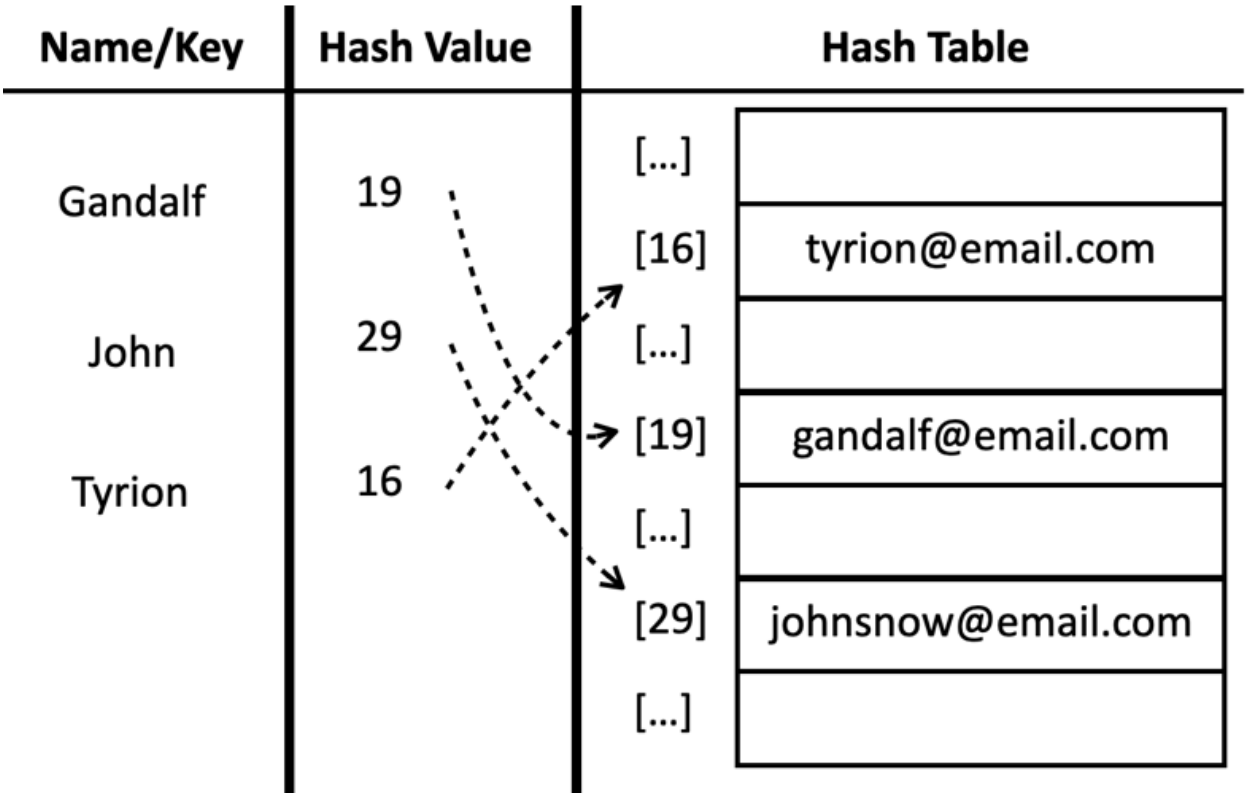
```
const addressBook = new HashTable();
// Add contacts
addressBook.put('Gandalf', 'gandalf@email.com');
addressBook.put('John', 'johnsnow@email.com');
addressBook.put('Tyrion', 'tyrion@email.com');
```

We can gain insights into the internal structure of our hash table by inspecting the hash codes generated for specific keys. For instance, we can observe the hash values calculated for "Gandalf," "John," and "Tyrion" using the hash method:

```
console.log(addressBook.hash('Gandalf')); // 19
console.log(addressBook.hash('John')); // 29
console.log(addressBook.hash('Tyrion')); // 16
```

The resulting hash codes (19, 29, and 16, respectively) reveal how the hash table distributes these keys into different positions within its underlying array. This distribution is crucial for efficient storage and retrieval of values.

The following diagram represents the `HashTable` data structure with these values in it:



A hash table with three contacts

Now let's put our `get` method to the test. By executing the following code, we can verify its behavior:

```
console.log(addressBook.get('Gandalf')); // gandalf@email.com
console.log(addressBook.get('Loiane')); // undefined
```

Since "Gandalf" is a key that exists within our `HashTable`, the get method successfully retrieves and outputs its associated value, "gandalf@email.com". However, when we attempt to retrieve a value for "Loiane," a non-existent key,

the `get` method returns undefined, indicating that the key is not present in the hash table.

Next, let's remove "Gandalf" from the `HashTable` using the `remove` method:

```
console.log(addressBook.remove('Gandalf')); // true
console.log(addressBook.get('Gandalf')); // undefined
```

After removing "Gandalf," calling `hash.get('Gandalf')` now results in `undefined`. This confirms that the entry has been successfully deleted, and the key no longer exists within the hash table.

Occasionally, different keys can result in identical hash values, a phenomenon known as a **collision**. Let's delve into how we can effectively manage collisions within our hash table.

Collisions between keys in a hash table

In certain scenarios, distinct keys may produce identical hash values. We refer to this phenomenon as a collision, as it leads to attempts to store multiple key-value pairs at the same index within the hash table.

For example, let's review at the following email address book:

```
const addressBook = new HashTable();
addressBook.put('Ygritte', 'ygritte@email.com');
addressBook.put('Jonathan', 'jonathan@email.com');
addressBook.put('Jamie', 'jamie@email.com');
addressBook.put('Jack', 'jack@email.com');
addressBook.put('Jasmine', 'jasmine@email.com');
addressBook.put('Jake', 'jake@email.com');
addressBook.put('Nathan', 'nathan@email.com');
addressBook.put('Athelstan', 'athelstan@email.com');
addressBook.put('Sue', 'sue@email.com');
addressBook.put('Aethelwulf', 'aethelwulf@email.com');
addressBook.put('Sargeras', 'sargeras@email.com');
```

To illustrate the collision concept, let's examine the output generated by evoking the `addressBook.hash` method for each name mentioned:

```
4 - Ygritte
5 - Jonathan
5 - Jamie
7 - Jack
8 - Jasmine
9 - Jake
10 - Nathan
7 - Athelstan
5 - Sue
5 - Aethelwulf
10 - Sargeras
```

Notice that multiple keys share the same hash values:

- Nathan and Sargeras both have a hash value of 10.
- Jack and Athelstan both have a hash value of 7.
- Jonathan, Jamie, Sue, and Aethelwulf all share a hash value of 5.

What happens within the hash table after adding all the contacts? Which values are ultimately retained? To answer these questions, let's introduce a `toString` method to inspect the hash table's contents:

```
toString() {
  const keys = Object.keys(this.#table);
  let objString = `{${keys[0]} =>
${this.#table[keys[0]].toString()}}`;
  for (let i = 1; i < keys.length; i++) {
    const value =
this.#elementToString(this.#table[keys[i]]).toString();
    objString = `${objString}\n{${keys[i]} => ${value}}`;
  }
  return objString;
}
```

The `toString` method provides a string representation of the hash table's contents. Since we cannot directly determine which positions in the underlying array contain values, we utilize `Object.keys` to retrieve an array of keys from the

`#table` object. We then iterate through these keys, constructing a formatted string that displays each key-value pair.

Upon invoking `console.log(hashTable.toString())`, we observe the following output:

```
{4 => ygritte@email.com}
{5 => aethelwulf@email.com}
{7 => athelstan@email.com}
{8 => jasmine@email.com}
{9 => jake@email.com}
{10 => sargeras@email.com}
```

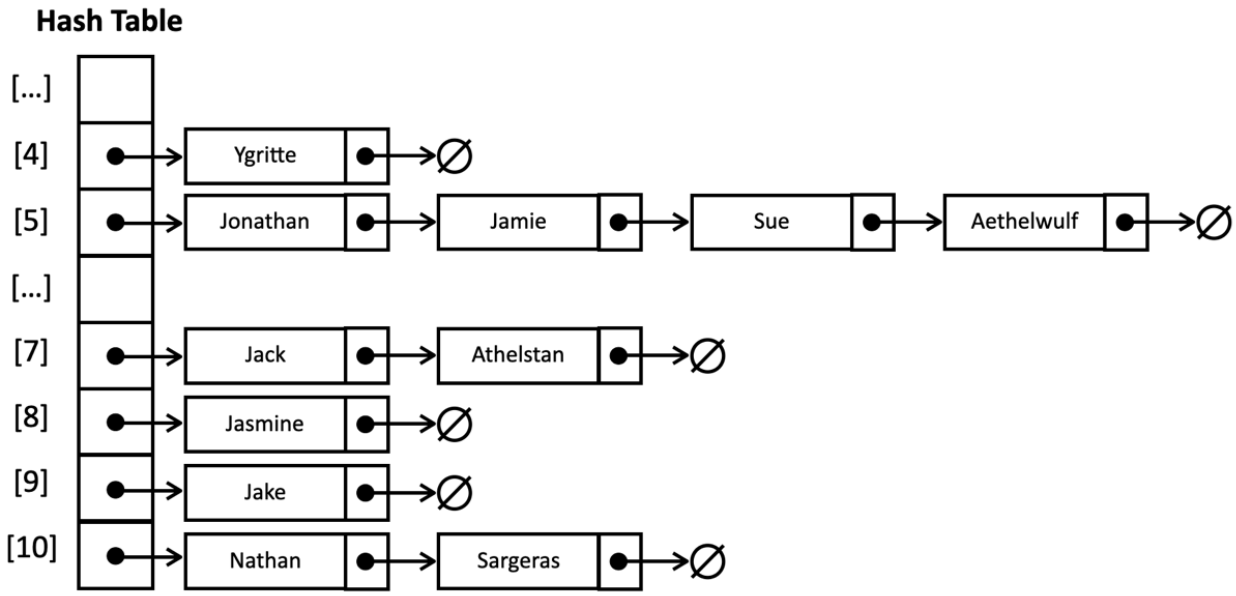
In his example, Jonathan, Jamie, Sue, and Aethelwulf all share the same hash value of 5. Due to the nature of our current hash table implementation, Aethelwulf, being the last added, occupies position 5. The values for Jonathan, Jamie, and Sue have been overwritten. Similar overwriting occurs for keys with other colliding hash values.

Losing values due to collisions is undesirable in a hash table. The purpose of this data structure is to preserve all key-value pairs. To address this issue, we need collision resolution techniques. There are several approaches, including **separate chaining**, **linear probing**, and **double hashing**, we will explore the first two in detail.

Handling collisions with separate chaining technique

Separate chaining is a widely used technique to handle collisions in hash tables. Instead of storing a single value at each index (bucket) of the hash table, separate chaining allows each bucket to hold a *linked list* (or another similar data structure) of values. When a collision occurs (multiple keys hash to the same index), the new key-value pair is simply appended to the linked list at that index.

To visualize this concept, let's consider the code used for testing in the previous section. If we were to apply separate chaining and represent the resulting structure diagrammatically, the output would resemble the following:



A hash table with separate chaining technique

In this representation, position 5 would contain a linked list with four elements, while positions 7 and 10 would each hold linked lists with two elements. Positions 4, 8, and 9 would each house linked lists with a single element. This illustrates how separate chaining effectively handles collisions by storing multiple key-value pairs in linked lists within the same bucket.

There are some advantages of using the separate chaining technique:

- Handles collisions gracefully by not overwriting data when collisions occur.
- The implementation is relatively straightforward to code compared to other collision resolution techniques.
- Linked lists have dynamic size and can grow as needed, accommodating more collisions without requiring a hash table resize.
- As long as the chains (linked lists) remain relatively short, search, insertion, and deletion operations remain efficient (close to *O(1)* in the average case), meaning it has satisfactory performance.

And as any technique, it also has some drawbacks:

- Extra memory overhead as linked lists require additional memory.

In the worst case, if many keys hash to the same index, the linked list could become long, impacting performance.

To demonstrate the practical application of separate chaining, let's create a new data structure called `HashTableSeparateChaining`. This implementation will focus primarily on the put, get, and remove methods, showcasing how separate chaining enhances collision handling within a hash table. To implement separate chaining in our hash table, we begin with the following code, housed within the `src/08-dictionary-hash/hash-table-separate-chaining.js` file:

```
const LinkedList = require('../06-linked-list/linked-list');
class HashTableSeparateChaining {
  #table = [];
}
```

This initial code snippet accomplishes two key tasks:

It imports the `LinkedList` class from another file (`../06-linked-list/linked-list.js`) we previously created in *Chapter 6, Linked Lists*.

It also defines the `HashTableSeparateChaining` class, which will encapsulate our hash table functionality. The class has a private property `#table`, initialized as an empty array. This array will serve as the backbone of our hash table, with each element acting as a bucket that can potentially store a linked list of key-value pairs.

The subsequent steps will involve filling in the core methods (`put`, `get`, and `remove`) that leverage linked lists to efficiently handle collisions and manage key-value pairs within the hash table.

Putting a key and a value with separate chaining technique

Let's implement the first method, the put method using the separate chaining technique as follows:

```
put(key, value) {
  if (key != null && value != null) {
    const index = this.hash(key);
    if (this.#table[index] == null) {
```

```
      this.#table[index] = new LinkedList();
    }
    this.#table[index].append({key, value});
    return true;
  }
  return false;
}
```

The put method in our `HashTableSeparateChaining` class is responsible for inserting or updating key-value pairs. Its first step is to validate the input, ensuring both the key and value are not `null` or `undefined`.

Next, we compute the hash code (index) using the hash function. This index determines the bucket where the key-value pair should be stored.

We then check if the bucket at the calculated index is empty. If it is, a new `LinkedList` is created to store values at this index. This is the core of separate chaining: using linked lists to accommodate multiple values that hash to the same index.

Finally, the key-value pair, encapsulated as an object `{key, value}`, is appended to the linked list at the specified index. If the key already exists in the linked list, its associated value is updated. The method returns `true` upon successful insertion or update, and `false` if the key or value is invalid.

Retrieving a value with separate chaining technique

Now, let's implement the get method to retrieve a value from our `HashTableSeparateChaining` class based on a given key:

```
get(key) {
  const index = this.hash(key);
  const linkedList = this.#table[index];
  if (linkedList != null) {
    linkedList.forEach((element) => {
      if (element.key === key) {
        return element.value;
```

```
      }
    });
  }
  return undefined; // key not found
}
```

In this method, we first hash the provided key to determine its corresponding index in the hash table. We then access the linked list stored at that index.

If there is a linked (`linkedList != null`), we iterate through its elements using a `forEach` loop, passing a callback. For each element in the linked list, we compare its key property to the input key. If we find a match, we return the corresponding value property of that element.

If the key is not found within the linked list, or if the linked list at the calculated index is empty, the method returns `undefined` to indicate that the key was not present in the hash table.

By incorporating the linked list structure and traversal, this `get` method effectively handles potential collisions caused by multiple keys hashing to the same index, ensuring that we retrieve the correct value even in the presence of collisions.

The LinkedList forEach method

Since our previous `LinkedList` class lacked a `forEach` method, we will need to add it for the efficient traversal required in the `HashTableSeparateChaining` class's get method.

Here is the implementation of the `forEach` method for the `LinkedList` class:

```
forEach(callback) {
  let current = this.#head;
  let index = 0;
  while (current) {
    callback(current.data, index);
    current = current.next;
```

```
    index++;
  }
}
```

The `current` variable keeps track of the current node in the list. It starts at the head of the list (`this.#head`). The loop continues as long as there are nodes left to process (`current is not null`). In each iteration, the provided `callback` function is called, passing the element stored in the current node and its index as arguments. The `current` variable is updated to the next node in the list, moving the iteration forward.

Removing a value with separate chaining technique

Removing a value from the `HashTableSeparateChaining` instance presents a slight variation compared to the previous `remove` method we implemented. Due to the utilization of linked lists, we now need to specifically target and remove the element from the relevant linked list within the hash table.

Let's analyze the implementation of the `remove` method:

```
remove(key) {
  const index = this.hash(key);
  const linkedList = this.#table[index];
  if (linkedList != null) {
    const compareFunction = (a, b) => a.key === b.key;
    const toBeRemovedIndex = linkedList.indexOf({key},
compareFunction);
    if (toBeRemovedIndex >= 0) {
      linkedList.removeAt(toBeRemovedIndex);
      if (linkedList.isEmpty()) {
        this.#table[index] = undefined;
      }
      return true;
    }
  }
  return false; // key not found
}
```

We begin by calculating the hash code (`index`) for the given key, similar to the `get` method. It then retrieves the linked list stored at that index. If the linked list exists (`linkedList != null`), we define a comparison function (`compareFunction`) that will be used to identify the element to be removed. This function compares the keys of two objects (`a` and `b`).

Next, we use the `indexOf` method of the linked list to find the index of the element we want to remove. The `indexOf` method takes the element to search for (`{key}`) and the comparison function as arguments. If the element is found, `indexOf` returns its index; otherwise, it returns -1.

If the element is found (`toBeRemovedIndex >= 0`), we remove it from the linked list using the `removeAt` method, which removes the element at the specified index.

After removing the element, we check if the linked list is now empty. If it is, we set the corresponding bucket in the hash table (`this.#table[index]`) to `undefined`, effectively removing the empty linked list. Finally, we return `true` to indicate successful removal.

If the key is not found within the linked list, or if the linked list at the calculated index is empty, we return `false`, signaling that the key was not present in the hash table.

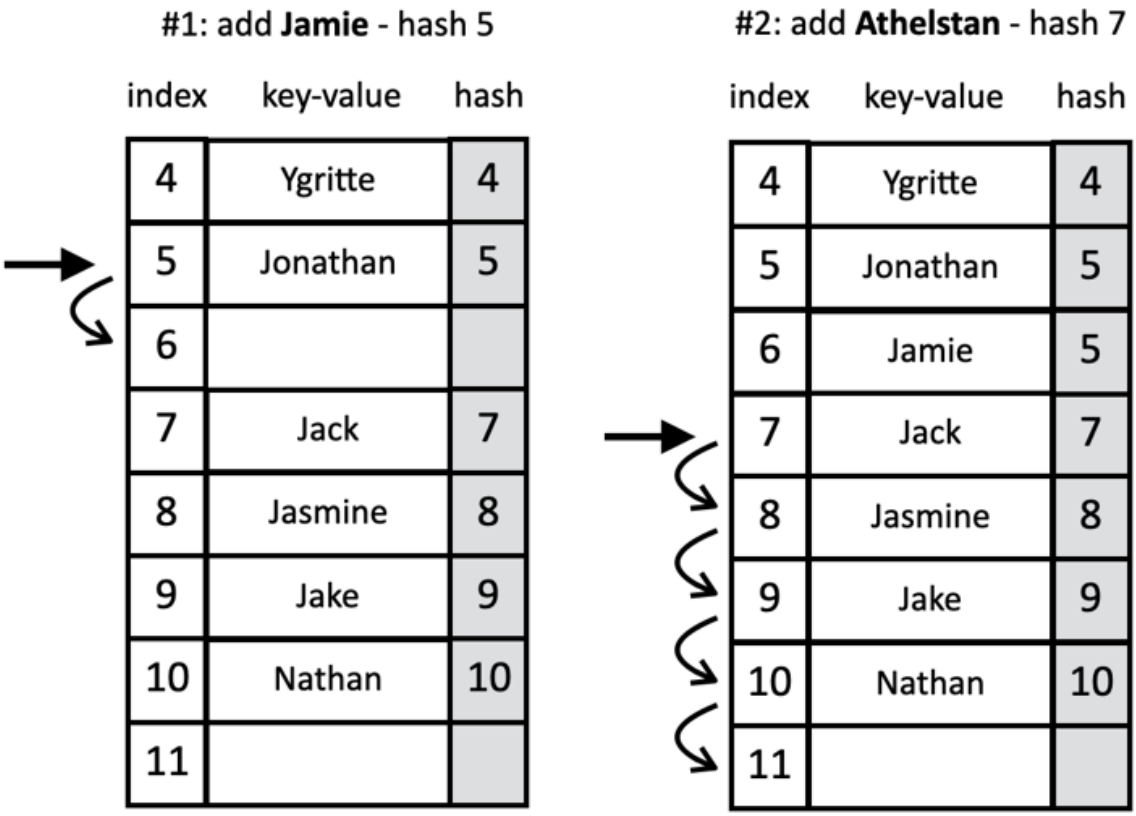
By incorporating linked list removal logic, this enhanced `remove` method seamlessly integrates with the separate chaining approach, enabling efficient removal of key-value pairs even in the presence of collisions. Next, let's delve into a different technique to handle collisions.

Handling collisions with the linear probing technique

Linear Probing is another technique to handle collisions in hash tables, differing from separate chaining in its approach to storing multiple key-value pairs with the same hash code.

Instead of using linked lists, linear probing directly stores all key-value pairs in the hash table array itself. When a collision occurs, linear probing sequentially searches for the next available empty slot in the array, starting from the original hash index.

The following diagram demonstrates this process:



A hash table with linear probing technique

Let's consider a scenario where our hash table already contains several values. When adding a new key-value pair, we calculate the hash for the new key. If the corresponding position in the table is vacant, we can directly insert the value at that index. However, if the position is occupied, we initiate a *linear probe*. We increment the index by one and check the next position. This process continues until we find an available slot or determine that the table is full.

Linear probing offers a simple and space-efficient collision resolution mechanism. However, it can lead to clustering, where consecutive occupied slots decrease performance as the table fills up.

To illustrate linear probing in practice, we will develop a new data structure called `HashTableLinearProbing`. This class will reside in the `src/08-dictionary-hash/hash-table-linear-probing.js` file. We begin by defining the basic structure:

```
class HashTableLinearProbing {
  #table = [];
}
```

This initial structure mirrors the `HashTable` class, with a private property `#table` initialized as an empty array to store key-value pairs. However, we will override the `put`, `get`, and `remove` methods to incorporate the linear probing technique for collision resolution. This modification will fundamentally alter how the hash table handles situations where multiple keys hash to the same index, demonstrating a distinct approach compared to separate chaining.

Putting a key and a value with linear probing technique

Now, let's implement the first of our three core methods: the `put` method. This method is responsible for inserting or updating key-value pairs within the hash table, incorporating the linear probing technique for collision resolution:

```
put(key, value) {
  if (key != null && value != null) {
    let index = this.hash(key);
    // linear probing to find an empty slot
    while (this.#table[index] != null) {
      if (this.#table[index].key === key) {
        this.#table[index].value = value;
        return true;
      }
      index++;
      index %= this.#table.length;
```

```
    }
    this.#table[index] = {key, value};
    return true;
  }
  return false;
}
```

We start by ensuring that both the key and value are valid (not `null`). We then calculate the hash code for the key, which determines the initial position where the value should be stored.

However, if the initial position is already occupied, the linear probing process begins. The method enters a while loop that continues if the current index is occupied by a value. Inside the loop, it first checks if the `key` at the current index matches the provided `key`. If so, the value is updated, and `true` is returned. Otherwise, the index is incremented, and the process wraps around to the beginning of the table if necessary, continuing the search for an empty slot.

Once an empty slot is found, the key-value pair is stored at that index, and `true` is returned to indicate a successful insertion. If either the key or value is invalid, the method returns `false`.

In some programming languages, we need to define the size of the array. One of the concerns of using linear probing is when the array is out of unoccupied positions. When the algorithm reaches the end of the array, it needs to loop back to its beginning and continue iterating its elements - and if necessary, we also need to create a new bigger array and copy the elements to the new array. In JavaScript, we benefit from the dynamic nature of arrays, which can grow automatically as needed. Therefore, we do not have to explicitly manage the table's size or worry about running out of space. This simplifies our implementation and allows the hash table to adapt to the amount of data being stored.

Let's simulate the insertion process within our hash table using linear probing to handle collisions:

> ***Ygritte***: the hash value for "Ygritte" is 4. Since the hash table is initially empty, we can directly insert it at position 4.

Jonathan: the hash value is 5, and position 5 is available, so we insert "Jonathan" there.
Jamie: this also hashes to 5, but position 5 is now occupied. We probe to position 6 (5 + 1), which is empty, and insert "Jamie" there.
Jack: the hash value is 7, and position 7 is empty, so we insert "Jack" without any collisions.
Jasmine: the hash value is 8, and position 8 is available, so "Jasmine" is inserted.
Jake: The hash value is 9, and position 9 is open, allowing us to insert "Jake" without collision.
Nathan: with a hash value of 10 and an empty position 10, "Nathan" is inserted smoothly.
Athelstan: this also hashes to 7, but position 7 is occupied by "Jack." We probe linearly to positions 8, 9, 10 (all occupied), and finally insert "Athelstan" at the first available position, 11.
Sue: hashing to 5, we find positions 5 through 11 occupied. We continue probing and insert "Sue" at position 12.
Aethelwulf: similarly hashing to 5, we probe past occupied positions and insert "Aethelwulf" at position 13.
Sargeras: the hash value is 10, and positions 10 to 13 are occupied. We probe further and insert "Sargeras" at position 14.

This simulation highlights how linear probing resolves collisions by systematically searching for the next available slot in the hash table array. While effective, it is important to note that linear probing can lead to clustering, which can potentially impact the performance of subsequent insertions and retrievals.

Next, let's review how to retrieve a value.

Retrieving a value with linear probing technique

Now that our hash table contains elements, let's implement the get method to retrieve values based on their corresponding keys:

```
get(key) {
  let index = this.hash(key);
  while (this.#table[index] != null) {
    if (this.#table[index].key === key) {
      return this.#table[index].value;
    }
    index++;
    index %= this.#table.length;
  }
  return undefined;
}
```

To retrieve a value, we first need to determine its location within the hash table. We use the hash function to calculate the initial index for the given key. If the key exists within the hash table, its value should be located either at the initial index or somewhere further along due to potential collisions.

If the initial index is not empty (`this.#table[index] != null`), we need to verify whether the element at that position matches the key we are searching for. If the keys match, we return the corresponding value immediately.

However, if the keys do not match, it is possible that the desired value has been displaced due to linear probing. We enter a `while` loop to iterate through subsequent positions in the table, incrementing the index and wrapping around if necessary. The loop continues until either the key is found, or an empty slot is encountered, signaling that the key does not exist.

If, after iterating through the table, the index points to an empty slot (`undefined` or `null`), it means the key was not found, and the method returns `undefined`.

Next, let's review how to remove values using the linear probing technique.

Removing a value with linear probing technique

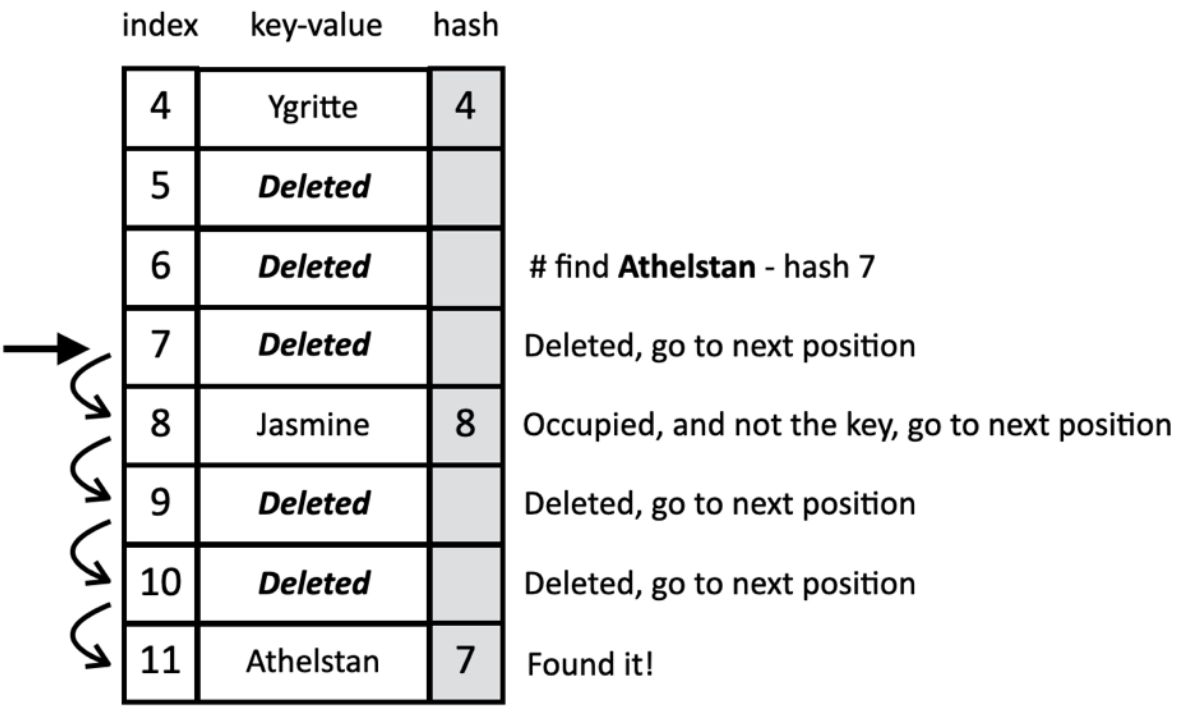
Removing a value from a hash table using linear probing presents a unique challenge compared to other data structures. In linear probing, elements are not necessarily stored at the index directly calculated from their hash value due to

potential collisions. Simply deleting the element at the hash index could disrupt the probing sequence and render other elements inaccessible.

To address this, we need a strategy that maintains the integrity of the probe sequence while still removing the desired key-value pair. There are two primary approaches:

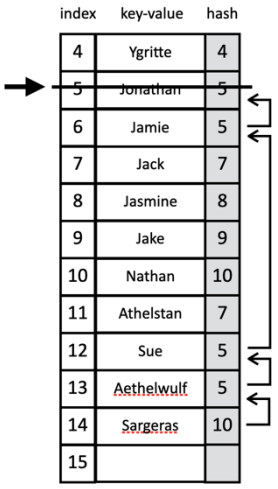
Soft deletion, also known as tombstone marking.
Hard deletion and rehashing the table.

In the soft deletion method, instead of physically removing the element, we mark it as *deleted* using a special value (often called a tombstone or flag). This value indicates that the slot was previously occupied but is now available for reuse. This method is simple to implement, however, this will gradually deteriorate the hash table's efficiency, as searching for key-values will become slower over time. This method also requires additional logic to handle tombstones during insertion and search operations. The following diagram demonstrates the process of the search operation with soft deletion method:



Linear probing removal with soft deletion

The second approach, hard deletion, involves physically removing the deleted element and then rehashing all subsequent elements in the probe sequence. This ensures that the probe sequence remains intact for future searches. In this method, there are no wasted spaces due to tombstones and it maintains an optimal probe sequence. However, it can be computationally expensive, especially for large hash tables or frequent deletions. This implementation is also more complex than soft deletion. When searching for a key, this approach prevents finding an empty spot, but if it is necessary to move elements, this means we will need to shift key-values within the hash table. The following diagram exemplifies this process:



Linear probing removal with rehashing

Both approaches have their pros and cons. For this chapter, we will implement the second approach (rehashing: move one or more elements to a backward position). To check the implementation of the lazy deletion approach (`HashTableLinearProbingLazy` class), please refer to the source code of this book. The download link for the source code is mentioned in the Preface of the book, or it can also be accessed at **http://github.com/loiane/javascript-datastructures-algorithms**.

Let's see the code for the remove method next.

Implementing the remove method with rehashing

The `remove` method in our hash table closely resembles the `get` method but with a crucial difference. Instead of simply retrieving the value, it deletes the entire key-value pair:

```
remove(key) {
  let index = this.hash(key);
  while (this.#table[index] != null) {
    if (this.#table[index].key === key) {
      delete this.#table[index];
      this.#verifyRemoveSideEffect(key, index);
      return true;
    }
    index++;
    index %= this.#table.length;
  }
  return false;
}
```

In the `get` method, upon finding the key, we returned its value. However, in `remove`, we use the `delete` operator to eliminate the element from the hash table. This could be at the original hash position or a different one due to previous collisions.

The challenge arises because we do not know if other elements with the same hash value were placed elsewhere due to a collision. If we simply delete the found element, we might leave gaps in the probe sequence, leading to errors when searching for those displaced elements. To address this, we introduce a helper method, `#verifyRemoveSideEffect`. This method is responsible for managing the potential side effects of removing an element. Its purpose is to move any collided elements backward in the probe sequence to fill the newly created empty spot, ensuring the integrity of the hash table's structure. This process is also known as rehashing:

```
#verifyRemoveSideEffect(key, removedPosition) {
  const size = this.#table.length;
```

```
  let index = removedPosition + 1;
  while (this.#table[index] != null) {
    const currentKey = this.#table[index].key;
    const currentHash = this.hash(currentKey);
    // check if the element should be repositioned
    if (currentHash <= removedPosition) {
      this.#table[removedPosition] = this.#table[index];
      delete this.#table[index];
      removedPosition = index;
    }
    index++;
    index %= size;
  }
}
```

We begin by initializing several variables: the `key` to be removed, the `removedPosition` where the key-value pair was located, the `size` of the hash table array, and an `index` variable to iterate through the table. The index starts at the position immediately after the removed element (`removedPosition + 1`).

The core of the method lies in a `while` loop that continues as long as there are elements in the table to examine. In each iteration, the `key` and its hash value are extracted from the element at the current `index`.

A crucial condition, `currentHash <= removedPosition`, is then evaluated. This checks if the element's original hash value (before linear probing) falls within the range of indices from the start of the table up to the `removedPosition`. If this condition holds `true`, it implies that the element was originally placed further down the probe sequence due to a collision with the removed element. To rectify this, the element at the current `index` is moved back to the now-empty `removedPosition`. Its original position is then cleared, and the `removedPosition` is updated to the current `index`. This ensures that subsequent elements in the probe sequence are also considered for repositioning.

The process repeats, incrementing the index and wrapping around if the end of the table is reached, until all potentially affected elements have been checked and repositioned if necessary. By meticulously evaluating the hash values and

repositioning elements, we guarantee that the probe sequence remains intact after a removal, ensuring the continued functionality and efficiency of the hash table.

This is a simplified implementation, as we could add validations for edge cases and optimize the performance. However, this code demonstrates the core logic of how to manage the side effects of removing an element in a hash table with linear probing.
Let's simulate the removal of "Jonathan" from the hash table we created earlier to demonstrate the process of linear probing with deletion and the subsequent side effect verification.

> ***Locating and removing Jonathan***: we find "Jonathan" at position 5 (hash value 5) and remove it, leaving position 5 empty. Now, we need to assess the side effects of this removal.
> ***Evaluating Jamie***: we move to position 6, where "Jamie" (also with hash value 5) is stored. Since Jamie's hash value is less than or equal to the removed position (5), we recognize that Jamie was originally placed here due to a collision. We copy Jamie to position 5 and delete the entry at position 6.
> ***Skipping Jack and Jasmine***: we continue to positions 7 and 8, where "Jack" (hash value 7) and "Jasmine" (hash value 8) are stored. Since their hash values are greater than both the removed position (5) and the current position (6), we determine that they were not affected by Jonathan's removal and should remain in their current positions.
> We repeat this evaluation for positions 9 through 11, finding no elements that need repositioning.
> ***Repositioning Sue***: at position 12, we find "Sue" (hash value 5). Since the hash value is less than or equal to the removed position (5), we copy Sue to position 6 (the originally vacated position) and delete the entry at position 12.
> ***Repositioning Aethelwulf and Sargeras***: we continue this process for positions 13 and 14, finding that both "Aethelwulf" (hash value 5) and "Sargeras" (hash value 10) need to be moved back. Aethelwulf is copied to position 12, and Sargeras is copied to position 13.

By following these steps, the remove method, along with the `#verifyRemoveSideEffect` helper function, ensures that the removal of

"Jonathan" does not leave any gaps in the probe sequence. All elements are repositioned as necessary to maintain the integrity and searchability of the hash table.

In our examples, we deliberately employed the lose-lose hash function to highlight the occurrence of collisions and illustrate the mechanisms for resolving them. However, in practical scenarios, it is crucial to utilize more robust hash functions to minimize collisions and optimize hash table performance. We will delve into better hash function options in the next section.

Creating better hash functions

A well-designed hash function strikes a balance between performance and collision avoidance. It should be fast to calculate for efficient insertion and retrieval of elements, while also minimizing the likelihood of collisions, where different keys produce the same hash code. While numerous implementations exist online, we can also craft our own custom hash function to suit specific needs.

One alternative to the lose-lose hash function is the `djb2` hash function, known for its simplicity and relatively good performance. Here is its implementation:

```
#djb2HashCode(key) {
  if (typeof key !== 'string') {
    key = this.#elementToString(key);
  }
  const calcASCIIValue = (acc, char) => (acc * 33) +
char.charCodeAt(0);
  const hash = key.split('').reduce((acc, char) => calcASCIIValue,
5381);
  return hash % 1013;
}
```

After transforming the `key` to a string, the key string is split into an array of individual characters. The `reduce` method is then employed to iterate over these characters, accumulating their ASCII values. Starting with an initial value of 5381

(a prime number that is the most common found in this algorithm), the reducer multiplies the accumulator by 33 (used as a magical number) and sums the result with the ASCII code of each character, effectively generating a sum of these codes.

Finally, we will use the remainder of the division of the total by another random prime number (1013), greater than the size we think the hash table instance can have. In our scenario, let's consider 1000 as the size.

Let's revisit the insertion scenario from the linear probing section, but this time using the `djb2HashCode` function instead of `loseloseHashCode`. The resulting hash codes for the same set of keys would be:

807 - Ygritte
288 - Jonathan
962 - Jamie
619 - Jack
275 - Jasmine
877 - Jake
223 - Nathan
925 - Athelstan
502 - Sue
149 - Aethelwulf
711 - Sargeras

Notably, we observe no collisions in this scenario. This is due to the improved distribution of hash values provided by the `djb2HashCode` function compared to the simplistic `loseloseHashCode`.

While not the absolute best hash function available, `djb2HashCode` is widely recognized and recommended within the programming community for its simplicity, effectiveness, and relatively good performance in many use cases. Its ability to significantly reduce collisions in this example underscores the importance of selecting an appropriate hash function for your specific data and application requirements.

Now that we have a solid grasp of hash tables, let's revisit the concept of sets and explore how we can leverage hashing to enhance their implementation and create a powerful data structure known as a **hash set**.

The hash set data structure

A **hash set** is a collection of unique values (no duplicates allowed). It combines the characteristics of a mathematical set with the efficiency of hash tables. Like hash tables, hash sets use a hash function to calculate a hash code for each element (value) we want to store. This hash code determines the index (bucket) where the value should be placed in an underlying array.

We can reuse the code we created in this chapter to create the hash set data structure as well, but with one important detail: we would need to check for duplicate values before the insertion operation.

The benefits of using hash sets are that it is guaranteed that all values in the set are unique. In JavaScript, the native Set class is considered a hash set data structure as well. For example, we could use a hash set to store all the English words (without their definitions).

Maps and TypeScript

Implementing data structures like maps or hash maps in TypeScript can significantly benefit from the language's static typing capabilities. By explicitly defining types for variables and method parameters, we enhance code clarity, reduce the risk of runtime errors, and enable better tooling support.

Let's examine the TypeScript signature for our `HashTable` class:

```
class HashTable<V> {
  private table: V[] = [];
  private loseLoseHashCode(key: string): number { }
  hash(key: string): number { }
  put(key: string, value: V): boolean { }
```

```
  get(key: string): V { }
  remove(key: string): boolean { }
}
```

In this TypeScript implementation, we introduce a generic type parameter `<V>` to represent the type of values stored in the hash table. This allows us to create hash tables that hold values of any specific type (for example: `HashTable<string>`, `HashTable<number>`, and so on). The `table` property is typed as an array of the generic type `V[]`, indicating that it stores an array of values of the specified type.

A significant advantage of using TypeScript becomes evident in the `loseLoseHashCode` method. Since the `key` parameter is explicitly typed as a `string`, we no longer need to check its type within the method. The type system guarantees that only strings will be passed as keys, eliminating the need for redundant checks, and streamlining the code:

```
privaye loseLoseHashCode(key: string) {
  const calcASCIIValue = (acc, char) => acc + char.charCodeAt(0);
  const hash = key.split('').reduce(calcASCIIValue, 0);
  return hash % 37;
}
```

By leveraging TypeScript's type system, we enhance the robustness, maintainability, and readability of our hash table implementation, making it easier to reason about and work within larger projects.

Reviewing the efficiency of maps and hash maps

Let's review the efficiency of each method by reviewing the Big O notation in terms of time of execution:

Method	**Dictionary**	**Hash Table**	**Separate Chaining**	**Linear Probing**
`put(key, value)`	*O(1)*	*O(1)*	*O(1)**	*O(1)**
`get(key)`	*O(1)*	*O(1)*	*O(1)**	*O(1)**
`remove(key)`	*O(1)*	*O(1)*	*O(1)**	*O(1)**

For the `Dictionary` class, all operations are generally *O(1)* in the average case due to direct access to the underlying object using the stringified key.

For the `HashMap` class, similar to the dictionary, all operations are typically *O(1)* in the average case, assuming a good hash function. However, it lacks collision handling, so collisions will cause data loss or overwrite existing entries.

For the `HashTableSeparateChaining`, in the average case, all operations are still *O(1)*. Separate chaining effectively handles collisions, so even with some collisions, the linked lists at each index are likely to remain short. In the worst case (all keys hash to the same index), the performance degrades to *O(n)* as you need to traverse the entire linked list.

Finally, for the `HashTableLinearProbing`, the average case complexity is also *O(1)* if the hash table is sparsely populated (low load factor). However, as the load factor increases and collisions become more frequent, linear probing can lead to clustering, where multiple keys are placed in consecutive slots. This can degrade the worst-case performance to *O(n)*.

Reviewing the execution time, the quality of the hash function significantly affects performance. A good hash function minimizes collisions, keeping performance closer to *O(1)*. In often cases, separate chaining tends to handle collisions more gracefully than linear probing, especially at higher load factors. In a hash table, the **load factor** is a crucial metric that measures how full the table is. It is defined as:

Load Factor = (Number of Elements in the Table) / (Total Number of Buckets)

Next, let's review the space complexity of each data structure:

Data Structure	**Space Complexity**	**Explanation**

Dictionary	*O(n)*	The space used grows linearly with the number of key-value pairs stored. Each pair occupies space in the underlying object.
HashTable	*O(n)*	The array has a fixed size, but you still need space for each stored key-value pair. Unused slots also consume space, especially if there are few collisions.
Separate Chaining	*O(n + m)*	*n* is the number of elements, and *m* is the number of buckets. In addition to the space for elements, each bucket holds a linked list, which adds memory overhead.
Linear Probing	*O(n)*	Similar to the simple hash table, but linear probing tends to use space more efficiently than separate chaining as there are no linked lists.

Which data structure should we use?

If we store 100 elements in a hash table with 150 buckets:

> ***Dictionary***: space usage is proportional to 100 elements.
> ***HashTable*** (no collision handling): space usage is still for 100 elements, plus potentially wasted space in the remaining 50 empty buckets.
> ***HashTableSeparateChaining***: space usage is for 100 elements, plus the overhead of the linked lists in each bucket (which could vary depending on how many collisions there are).
> ***HashTableLinearProbing***: space usage is likely closer to 100 elements, as it tries to fill the array more densely.

At the end of the day, it all depends on the scenario we are working with.

Let's put our knowledge into practice with some exercises.

Exercises

We will resolve one exercise from **LeetCode** using the map data structure to transform integer numbers to roman numbers.

However, there are many fun exercises available in LeetCode that we should be able to resolve with the concepts we learned in this chapter. Below are some additional suggestions you can try to resolve, and you can also find the solution along with the explanation within the source code from this book:

1. Two Sum: given an array of integers, find two numbers that add up to a target sum. This is a classic problem that introduces you to using a hash map to store complements and quickly find matches.
242. Valid Anagram: determine if two strings are anagrams of each other (contain the same characters but in a different order). Hash maps are useful for counting character frequencies.
705. Design HashSet: implement the hash set data structure.
706. Design HashMap: implement the hash map data structure.
13. Roman to Integer: similar to the problem we will resolve, but the other way around.

Integer to Roman

The exercise we will resolve the is the *12. Integer to Roman* problem available at **https://leetcode.com/problems/integer-to-roman/description/**.

When resolving the problem using JavaScript or TypeScript, we will need to add our logic inside the function function intToRoman(num: number): string, which receives a numerical input and returns its corresponding Roman numeral representation as a string.

Let's explore a solution using a map data structure to facilitate the conversion process:

```
function intToRoman(num: number): string {
  const romanMap = {
    M:1000, CM:900, D:500, CD:400, C:100, XC:90,
    L:50, XL:40, X:10, IX:9, V:5, IV:4, I:1
  };
  let result = '';
  for(let romanNum in romanMap){
```

```
    while (num >= romanMap[romanNum]) {
      result += romanNum;
      num -= romanMap[romanNum];
    }
  }
  return result;
}
```

At the heart of this function lies the `romanMap`, which acts as a dictionary, associating Roman numeral symbols with their corresponding integer values. This map includes both standard Roman numerals (M, D, C) and special combinations for subtraction (CM, XC). The arrangement of keys in descending order of value is crucial for the greedy algorithm employed in the conversion process so we do not need to sort the data structure before the conversion process.

Next, we initialize an empty string, `result`, to accumulate the Roman numeral characters. It then enters a loop that iterates through the keys of the `romanMap`.

Within the loop, a nested `while` loop repeatedly checks if the input number (`num`) is greater than or equal to the integer value of the current Roman numeral. If so, the Roman numeral is appended to the `result` string, and its integer value is subtracted from `num`. This process continues until `num` becomes smaller than the value of the Roman numeral, indicating that we need to move on to the next smaller numeral in the map.

By iteratively selecting the largest possible Roman numeral that fits the remaining input value, the function constructs the Roman numeral representation in a *greedy* manner. Once the entire `romanMap` has been traversed, the function returns the completed result string, which now holds the accurate Roman numeral equivalent of the original input integer.

The time complexity of this function is *O(1)*. It iterates over a fixed set of Roman numeral symbols (13 symbols in total). For each symbol, it performs a series of subtractions and concatenations. The number of operations is bounded by the number of symbols and the maximum value of the input number, but since the

set of symbols and their values are constant, the operations do not scale with the input size.

The space complexity is also *O(1)*. The `romanMap` object is a constant and its size does not change with the input, so it contributes a constant space overhead. The result string grows based on the number of Roman numeral characters needed to represent the input number. However, since the maximum number of characters needed to represent any integer in Roman numerals is fixed (for example: 3999 is MMMCMXCIX), this also contributes a constant space overhead. No additional data structures are used that scale with the input size.

We could also use the native `Map` class to store the `romanMap` key-value pairs, however, in the `for` loop, we would need to extract they keys and sort them first. The `Map` class does not guarantee the order of keys, so we would need to extract and sort the keys before iterating, which adds overhead. So, in this case, the simplest data structure works in our favour for a more performant solution.

Summary

In this chapter, we explored the world of dictionaries, mastering the techniques to add, remove, and retrieve elements, while also understanding how they differ from sets. We delved into the concept of hashing, learning how to construct hash tables (or hash maps) and implement fundamental operations like insertion, deletion, and retrieval. Moreover, we learned how to craft hash functions and examined two distinct techniques for handling collisions: separate chaining and linear probing.

We also explored JavaScript's built-in `Map` class, as well as the specialized `WeakMap` and `WeakSet` classes, which offer unique capabilities for memory management. Through a variety of practical examples and a LeetCode exercise, we solidified our understanding of these data structures and their applications.

Equipped with this knowledge, we are now prepared to tackle the concept of **recursion** in the next chapter, paving the way for our exploration of another essential data structure: **trees**.