Solution

Array-based solution

The stack abstract data type can be easily implement in JavaScript with JavaScript `Array`s. The main thing to note when implementing stacks is that that the operations should be O(1). Thankfully, JavaScript `Array`'s `push()` and `pop()` operations are O(1) and the method signatures match the `Stack`'s. Many of the methods are just simple wrappers around `Array` methods.

JavaScript TypeScript

```
export default class Stack {
 constructor() {
  this._items = [];
 }

 /**
  * Pushes an item onto the top of the stack.
  * @param {*} item The item to be pushed onto the stack.
  * @return {number} The new length of the stack.
  */
 push(item) {
  return this._items.push(item);
 }

 /**
  * Remove an item at the top of the stack.
  * @return {*} The item at the top of the stack if it is not empty, `undefined` otherwise.
  */
 pop() {
  return this._items.pop();
 }

 /**
  * Determines if the stack is empty.
  * @return {boolean} `true` if the stack has no items, `false` otherwise.
  */
 isEmpty() {
  return this.length() === 0;
 }

 /**
```

```
  * Returns the item at the top of the stack without removing it from the stack.
  * @return {*} The item at the top of the stack if it is not empty, `undefined` otherwise.
  */
 peek() {
  return this.isEmpty() ? undefined : this._items[this.length() - 1];
 }

 /**
  * Returns the number of items in the stack.
  * @return {number} The number of items in the stack.
  */
 length() {
  return this._items.length;
 }
}
```

Linked list-based solution

Stacks can also be implemented with singly-linked lists which are essentially a chain of connected nodes. To do that, we need to create a `Node` class that has a `prev` pointer, a reference to the `Node` below if it's not the bottom item of the stack.

`push()`-ing involves creating a new `Node` and pointing the current top of the stack to that new `Node`'s `prev` field and updating a reference to the new top of the stack. `pop()`-ing is the reverse where we find the `prev` node of the current top and set it to be the new top of the stack.

Since the time complexity of `length()` has to be O(1) and counting the number of items in a linked list will take O(n), we need to separately track the number of items in the list with a `_length` instance property and update it within the `push()` and `pop()` methods.

```
class Node {
 constructor(value) {
  this.value = value;
  this.prev = null;
 }
}

export default class Stack {
```

```
constructor() {
  this._top = null;
  this._length = 0;
}

/**
 * Pushes an item onto the top of the stack.
 * @param {*} item The item to be pushed onto the stack.
 * @return {number} The new length of the stack.
 */
push(item) {
  const node = new Node(item);
  node.prev = this._top;
  this._top = node;
  this._length++;
  return this._length;
}

/**
 * Remove an item at the top of the stack.
 * @return {*} The item at the top of the stack if it is not empty, `undefined` otherwise.
 */
pop() {
  if (this.isEmpty()) {
    return undefined;
  }

  const node = this._top;
  this._top = node.prev;
  node.prev = null;
  this._length--;
  return node.value;
}

/**
 * Determines if the stack is empty.
 * @return {boolean} `true` if the stack has no items, `false` otherwise.
 */
isEmpty() {
  return this._length === 0;
}

/**
```

```
  * Returns the item at the top of the stack without removing it from the stack.
  * @return {*} The item at the top of the stack if it is not empty, `undefined` otherwise.
  */
 peek() {
   return this.isEmpty() ? undefined : this._top.value;
 }

 /**
  * Returns the number of items in the stack.
  * @return {number} The number of items in the stack.
  */
 length() {
   return this._length;
 }
}
```